Иннопрактика

ФОНД РАЗВИТИЯ ФИЗТЕХ+ ШКОЛ

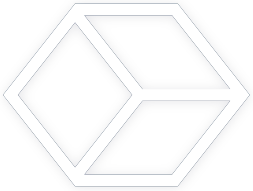
Робофутбол

EduНейро
инфраструктурный центр «Нейронет»

Национальная технологическая инициатива
Пространство возможного

ВВЕДЕНИЕ В ГЛУБОКОЕ ОБУЧЕНИЕ ДЛЯ ШКОЛЬНИКОВ

Данное методическое пособие было создано при поддержке негосударственного института развития «Иннопрактика» в целях работ по созданию методических материалов (методические пособия и видеолекции) в рамках программы инфраструктурного центра «Нейронет»



Оглавление

1. Введение

Всем привет!

В данной методичке мы с вами продолжим изучать различные подходы к решению задач методами глубокого обучения (нейронными сетями). До этого вы обсуждали преимущественно задачи компьютерного зрения, которые наиболее очевидным образом возникают в робототехнике. Здесь же несколько подробнее взглянем на некоторые задачи компьютерного зрения, в частности на задачи семантической сегментации и детекции.

В дополнение к этому мы проанализируем несколько задач из анализа естественного языка (NLP). Посмотрим и обсудим принципиально новый тип данных — последовательности, в частности текст (последовательность слов, токенов). А также поймем, что текст и числа, правильные числа более тесно связаны, чем кажется на первый взгляд.

Коснемся темы обработки звука, в частности речи. Разберем основные подходы к анализу звуковых сигналов и даже теорию рядов Фурье. И решим задачу распознавания ключевого слова (Keyword Spotting), опираясь на уже известную нам информацию и уже известные нам из CV инструменты.

И на сладенькое мы оставили горячо любимые вами игры. А именно мы «преисполнимся» темой обучения с подкреплением (Reinforcement Learning) и научим нашу сеточку играть в простые игры.

Планы грандиозные, настрой тоже. Давайте же начнем!

2. Computer Vision

2.1 Введение

В данном разделе рассмотрены две важные задачи компьютерного зрения — задача семантической сегментации и задача детекции. Приведены одни из наиболее популярных архитектур для их решения, описаны их достоинства и недостатки. Приведена реализация архитектур, субъективно наиболее удачных для процесса обучения и для начала решения прикладных задач с помощью нейронных сетей.

В структуре каждого блока данного раздела можно выделить две основных части:

- Постановка задачи: актуальность, формализация ввода и вывода, требования к сети для практического применения.
- Небольшое введение о том, как задача решается, как в частном случае будет измеряться качество, описание основных идей для решения задачи и улучшение некоторых архитектур.

Для простоты изложения в данных разделах многие формулировки будут подкреплены примерами на кошечках и собачках или простыми картинками, иллюстрирующими утверждение.

2.2 Семантическая сегментация

2.2.1 Постановка задачи

Задача семантической сегментации — это задача о присвоении каждому пикселю изображения метки определенного класса. Например, поиск человека на картинке. Разные люди на картинке в рамках задачи семантической сегментации неразличимы, так как все являются объектом класса «человек» и имеют общую метку. В этом отличие задачи семантической сегментации от задачи так называемой инстанс-сегментации, которая предполагает

помимо выделения класса, выделение также и различных объектов класса, то есть, помимо определения человека на картинке, определение, кому из людей принадлежит конкретный пиксель.

Задача семантической сегментации может быть поставлена несколькими способами — набор классов объектов, которые выделяются на картинке, может варьироваться. Если на картинке несколько классов объектов (фон/столбы/автомобили/люди), мы можем сегментировать произвольное подмножество этих классов.

Рис. 2.1: Пример семантической и инстанс-сегментации [1]

2.2.2 Вход и выход

Определим, что мы подаем на вход и что хотим получать на выходе как решение задачи сегментации.

Итак, мы имеем картинку, которую отдаем на вход. На выходе хотим получить картинку того же размера, на которой каждый пиксель будет сопоставлен с одним из объектов класса (в случае, если мы выделяем на картинке фон и все не интересующие нас классы относим к нему). При этом заранее должен быть определен набор классов, который мы хотим сегментировать.

На выходе сети мы хотим иметь вероятности принадлежности определенному классу для каждого пикселя. Каждый пиксель может с некоторой вероятностью принадлежать любому из классов — применим к выходу Softmax, и тогда сможем посчитать Loss-функцию, в качестве которой возьмем кросс-энтропию. Выбор кросс-энтропии обусловлен тем, что суть задачи именно в классификации пикселей, а для задачи классификации хорошо

подходит кросс-энтропия.

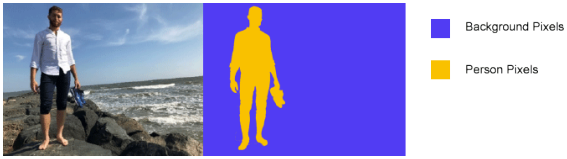


Рис. 2.2: Семантическая сегментация [2]

2.2.3 Метрики качества

Самая простая метрика качества семантической сегментации — попиксельная точность:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

, где TP — количество пикселей, принадлежащих классу и классифицированных верно (true positives), TN — количество пикселей, не принадлежащих классу и классифицированных верно (true negatives), FP — количество пикселей, которые классифицированы как принадлежащие классу, но они таковыми не являются (false positives), FN — количество пикселей, которые принадлежат классу, но классифицированные как не принадлежащие классу (false negatives).

Такая метрика, однако, плохо работает в случае несбалансированности классов. Решается эта проблема введением средней попиксельной точности — вычислением попиксельной точности для каждого класса, и усреднением по количеству классов.

Чаще используется другая метрика — Intersection over Union (IoU).

$$IoU = \frac{TP}{TR + FP + FN}$$

- Часто вычисляется среднее значение метрики IoU по всем классам.

- Среднее значение метрики IoU может вычисляться как взвешенное среднее по значениям, полученным для отдельных классов. Веса соответствуют частотам встречаемости пикселей каждого класса.
- Класс «фон» можно как учитывать, так и не учитывать (можно считать, что фон отсутствует)

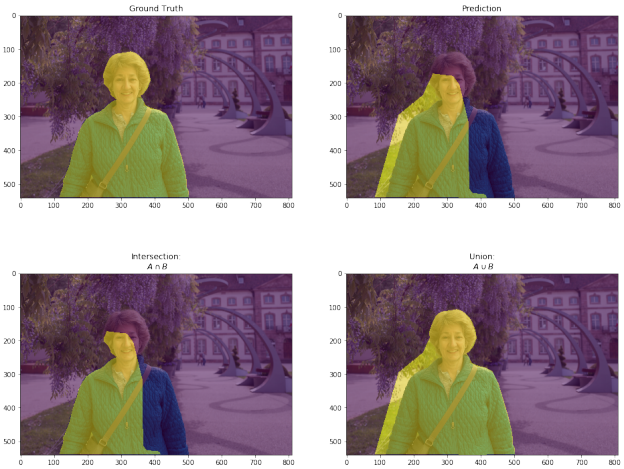


Рис. 2.3: Пример использования метрики IoU для задачи семантической сегментации

2.2.4 Актуальность

Семантическая сегментация используется, например, для сегментирования земной поверхности по снимкам со спутников, в частности с целью выделения, контуров водоемов или нанесения дорог и зданий на карту. Кроме картографических приложений, более локально, снимки свысока могут быть использованы для анализа загруженности дорог, открытых парковок и т.п. Дру-

гое часто упоминаемое применение семантической сегментации — определение с ее помощью дорожной обстановки для автопилотируемых систем. Самоходному автомобилю необходимо определить границы дороги, полосы движения, другие автомобили, людей, животных. При этом ввиду динамичности таких сцен сегментация должна осуществляться очень быстро при ограниченных вычислительных ресурсах.

Еще одно важнейшее приложение семантической сегментации – медицина. Области ее применения обширны. Например, она используется в стоматологии или при анализе МРТ снимков для поиска аномалий, значительно уменьшая вероятность человеческого фактора.

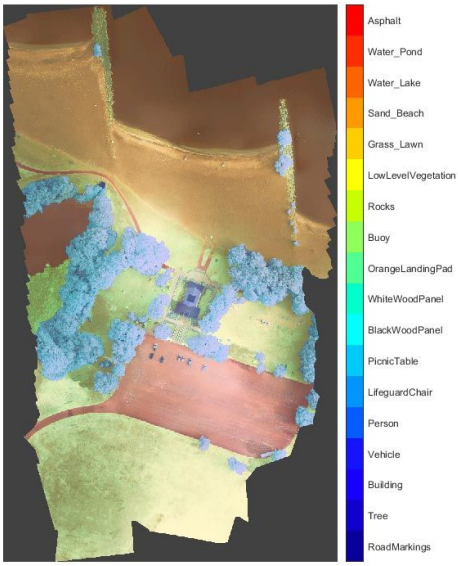


Рис. 2.4: Семантическая сегментация земной поверхности [3]

2.2.5 Скользящее окно

Рассмотрим самые простые идеи семантической сегментации. Установив формат вывода, мы свели задачу семантической сегментации к задаче классификации для каждого пикселя.

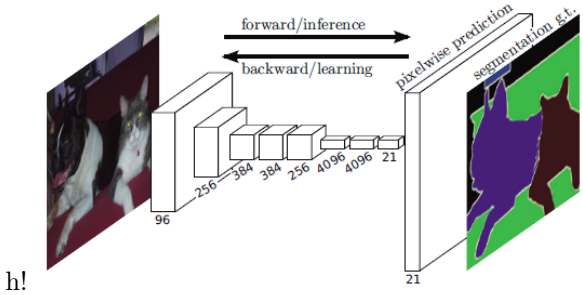
Для того, чтобы классифицировать каждый пиксель, будем рассматривать окно с центром в этом пикселе и, например, размером 10×10. Размер выбирается из конкретной задачи и зависит от датасета и характерного относительного размера объектов – мы хотим, чтобы он примерно соответствовал размеру окна. Каждое такое окошко мы даем на вход реализованному ранее классификатору и, как нетрудно догадаться, на выходе получим сегментированное изображение установленного формата (при условии, что расширили исходное изображение подложкой для получения исходного разрешения на выходе (далее – padding)).

Такая идея решения очень проста, однако имеет очень существенные недостатки:

- Вычислительная сложность — надо классифицировать каждый пиксель и очень много раз обращаться к сети-классификатору, что может быть достаточно затратно.
- В малом окне содержится очень малое количество информации. При этом любое окно никак не учитывает информацию из любых других окон — нужно корректно выбрать размер окна для классификатора, а также требуется, чтобы размеры классифицируемых объектов на изображении были примерно одного размера. Даже человек по части изображения, в которой виден лишь нос животного, не всегда сможет отличить кошку от собаки.

2.2.6 Fully convolutional network (FCN)

Другой подход к решению задачи семантической сегментации – Fully convolution network (FCN). Предлагается использовать сверточную глубокую сеть, получить сжатое изображение, которое можно сегментировать и восстановить размер до исходного. И вычислительно его обработка будет гораздо проще. О том, как



h!

Рис. 2.5: Архитектура FCN [4]

именно мы это сделаем, и пойдет речь в данном разделе.

Итак, будем работать со сверточной сетью, выход которой мы классифицируем и некоторым образом будем обрабатывать. Получится архитектура, похожая на сверточную сеть для классификации изображений, но без полносвязных слоев.

Нетрудно заметить, что такая архитектура не удовлетворяет одному из условий задачи – на выходе мы имеем картинку меньшего размера. Самый простой способ увеличить ее разрешение – развернуть свернутую картинку (далее – upsampling), ближайшими соседями, как указано на 2.6 или увеличить вдвое размер картинки, и по каждому каналу заполнить промежутки между пикселями линейной интерполяцией 2.7.

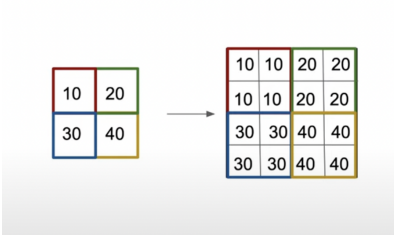


Рис. 2.6: Увеличение разрешения выходной картинки путем upsampling ближайшими соседями [5]

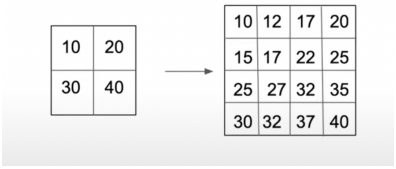


Рис. 2.7: Увеличнеие разрешения выходной картинки путем upsampling билинейной интерполяцией [6]

Реализация FCN

Начинаем с загрузки датасета

```
!wget https://www.dropbox.com/s/k88qukc20ljnbuo/PH2Dataset.rar
```

```
get_ipython().system_raw("unrar x PH2Dataset.rar")
```

Датасет устроен следующим образом:

```
IMD_002/
  IMD002_Dermoscopic_Image/
     IMD002.bmp
  IMD002_lesion/
      IMD002_lesion.bmp
  IMD002_roi/
        ...
IMD_003/
    ...
    ...
```

Каждая папка $IMD\_00\%$ содержит данные об одном наблюдении. В подпапках этого наблюдения находится изображение и сегментированное изображение.

Импортируем библиотеки:

```
import os
import torch
from torch import nn
from skimage.transform import resize
from skimage.io import imread
import numpy as np
from torch.utils.data import TensorDataset, DataLoader
from tqdm.notebook import tqdm
```

Класс для работы с данными:

```
class CustomDataset(TensorDataset):
  def __init__(self,
               root = 'PH2Dataset',
               folder_name = 'PH2 Dataset images',
               pic_size = (256, 256)):
    self.pic_size = pic_size

```

```
    self.images_paths = []
    self.lesions_paths = []
    for root, dirs, files in os.walk(os.path.join(root, folder_name)):
      if root.endswith('_Dermoscopic_Image'):
          self.images_paths.append(os.path.join(root, files[0]))
      if root.endswith('_lesion'):
          self.lesions_paths.append(os.path.join(root, files[0]))

  def __len__(self):
    return len(self.images_paths)

  def __getitem__(self, idx):
    img_feature = imread(self.images_paths[idx])
    img_feature = resize(img_feature,
                         self.pic_size,
                         mode='constant',
                         anti_aliasing=True,)

    img_target = imread(self.lesions_paths[idx])
    img_target = resize(img_target,
                        self.pic_size,
                        mode='constant',
                        anti_aliasing=False,)> 0.5

    return torch.from_numpy(
        np.array(np.rollaxis(img_feature, 2, 0),
                 np.float32)),
    torch.from_numpy(np.array(img_target, np.float32))
```

Создадим объекты класса Dataset, DataLoader для train данных и для валидации.

```
batch_size = 10

train_dataset = CustomDataset()
train_dataloader = DataLoader(train_dataset,
                              batch_size=batch_size,
                              shuffle=True)

val_dataset = CustomDataset()
val_dataloader = DataLoader(val_dataset,
                            batch_size=batch_size,
                            shuffle=True)
```

Определим переменную device, чтобы сделать код универсальным, при возможности для обучения будет использована GPU.

```
device = torch.device(
    'cuda' if torch.cuda.is_available() else 'cpu'
)
print(device)
```

Повторяющийся похожий блок вынесем в отдельный класс FCN_Block. На основе этого класса напишем FCN

```
class FCN_Block(nn.Module):
  def __init__(
        self,
        in_channels,
        out_channels,
        kernel_size=(3, 3),
        padding=1,
        with_pool=True,
        three_conv_layers = False
        ):
    super().__init__()
```

```
12      self.three_conv_layers = three_conv_layers
13      self.with_pool = with_pool
14      self.conv1 = nn.Conv2d(
15          in_channels,
16          out_channels,
17          kernel_size=kernel_size,
18          padding=padding
19          )
20      self.conv2 = nn.Conv2d(
21          out_channels,
22          out_channels,
23          kernel_size=kernel_size,
24          padding=padding
25          )
26      if three_conv_layers:
27          self.conv3 = nn.Conv2d(
28              out_channels,
29              out_channels,
30              kernel_size=kernel_size,
31              padding=padding
32              )
33      if with_pool:
34        self.pool = nn.MaxPool2d((2, 2), (2, 2))
35
36    def forward(self, x):
37      out = self.conv1(x)
38      out = self.conv2(out)
39      if self.three_conv_layers:
40        out = self.conv3(out)
41      if self.with_pool:
42        out = self.pool(out)
43
44      return out
45
```

```
46 class FCN(nn.Module):
47   def __init__(self):
48     super().__init__()
49     self.fcn_block1 = FCN_Block(3, 64)
50     self.fcn_block2 = FCN_Block(64, 128)
51     self.fcn_block3 = FCN_Block(128, 256,
52                                 three_conv_layers=True)
53     self.fcn_block4 = FCN_Block(256, 512,
54                                 three_conv_layers=True)
55     self.fcn_block5 = FCN_Block(512, 512,
56                                 three_conv_layers=True)
57
58     self.equals_linear = FCN_Block(
59         512,
60         4096,
61         kernel_size=(1, 1),
62         padding=0,
63         with_pool=False
64         )
65     self.conv1 = nn.Conv2d(4096, 1, (1, 1))
66
67   def forward(self, x):
68     out = self.fcn_block1(x)
69     out = self.fcn_block2(out)
70     out = self.fcn_block3(out)
71     out = self.fcn_block4(out)
72     out = self.fcn_block5(out)
73     out = self.equals_linear(out)
74
75     out = self.conv1(out)
76     out = torch.nn.functional.upsample(out, scale_factor=32)
77
78     return out
```

Напишем функцию для обучения сети

```
from time import time
from matplotlib import pyplot as plt
from IPython.display import clear_output


def train(model,
          optimizer,
          scheduler,
          loss_fn,
          epochs,
          data_tr,
          data_val):
    X_val, Y_val = next(iter(data_val))
    loss_values = []
    epoch_num = []

    for epoch in range(epochs):
        tic = time()
        print('* Epoch %d/%d' % (epoch+1, epochs))

        scheduler.step()
        avg_loss = 0
        model.train()  # train mode
        for X_batch, Y_batch in tqdm(data_tr):
            X_batch = X_batch.to(device)
            Y_batch = Y_batch.to(device)
            # data to device
            optimizer.zero_grad()
            # set parameter gradients to zero
            # forward
            Y_pred = model(X_batch)
            Y_pred = torch.squeeze(Y_pred)
```

```
            loss = loss_fn(Y_pred, Y_batch) # forward-pass
            loss.backward()  # backward-pass
            optimizer.step()  # update weights
            # calculate loss to show the user
            avg_loss += loss / len(data_tr)
        toc = time()
        loss_values.append(avg_loss.item())
        epoch_num.append(epoch)

        # show intermediate results
        with torch.no_grad():
          model.eval()
          # testing mode
          Y_hat = model(X_val.to(device)).detach().cpu()
          # detach and put into cpu
          Y_hat = torch.squeeze(Y_hat)

          # Visualize tools
          clear_output(wait=True)
          for k in range(6):
              plt.subplot(3, 6, k+1)
              plt.imshow(
                    np.rollaxis(X_val[k].numpy(), 0, 3), cmap='gray'
              )
              plt.title('Real')
              plt.axis('off')

              plt.subplot(3, 6, k+7)
              plt.imshow(Y_hat[k], cmap='gray')
              plt.title('Output')
              plt.axis('off')

              plt.subplot(3, 6, k+13)
```

```
            plt.imshow(Y_val[k], cmap='gray')
            plt.title('Target')
            plt.axis('off')
        plt.suptitle(
          '%d / %d - loss: %f' % (epoch+1, epochs, avg_loss)
        )
        plt.show()
        plt.plot(epoch_num, loss_values)
        plt.title('Loss values')
        plt.xlabel('epoch')
        plt.ylabel('loss')
        plt.show()
        print(
          'loss: %f' % avg_loss, 'LR: %f' % scheduler.get_lr()[0]
        )
```

```
model = FCN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
criterion = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=5,
    gamma=0.95
    )
train(
    model,
    optimizer,
    scheduler,
    criterion,
    100,
    train_dataloader,
    val_dataloader)
```

Ниже приведен пример работы обученной сети:

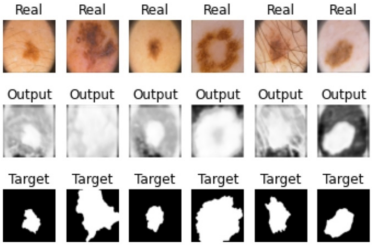


Рис. 2.8: Пример работы FCN

Такая архитектура так же не идеальна и имеет следующие недостатки:

- Метод увеличения разрешения плохо восстанавливает информацию, границы объектов могут получаться нечеткими
- Сжатие картинки и stride могут сильно разрушать пространственную информацию — при восстановлении разрешения невозможно будет отличить две близкие на выходе картинки, которые были разными на входе
- Такая архитектура не очень хорошо работает в случае, когда нужно сегментировать картинки с разной относительной величиной объектов класса — кот может занимать малую часть картинки, а может большую.

Недостатки выхода FCN на самом деле существенны, поскольку карта сегментации из-за столь грубого восстановления разрешения может быть сильно шумной.

Следующий раздел будет посвящен улучшению FCN путем постпроцессинга – обработки выхода сети.

2.2.7 Улучшение FCN

Основная проблема FCN – потеря пространственной информации при сжатии картинки, восстанавливается эта информация достаточно грубо. Рассмотрим способ улучшения выхода сети.

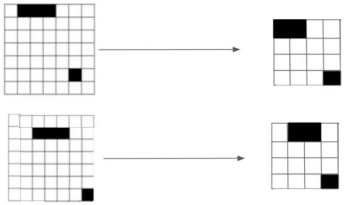


Рис. 2.9: Пример, показывающий, какие два изображения дадут одинаковый результат после upsampling

Рис. 2.10: Пример шумной картинки на выходе FCN

Итак, рассмотрим выходное изображение после каждого сверточного слоя.

Чем глубже слой относительно конечного, тем выше разрешение было на его выходе, тем больше пространственной информации сохранилось к этому выходу. На более близких к концу сверток слоях потеряна пространственная информация, но пиксели на этих изображениях несут в себе информацию о классах объектов. Если объединить выход более глубоких слоев с более мелкими, то сохраним больше информации о границах объектов, получив более качественную сегментацию.

На рисунке ниже приведено сравнение выходных изображений путем объединения выходов разных слоев.

На рисунке 2.12 наглядно виден эффект от использования данной схемы улучшения FCN.

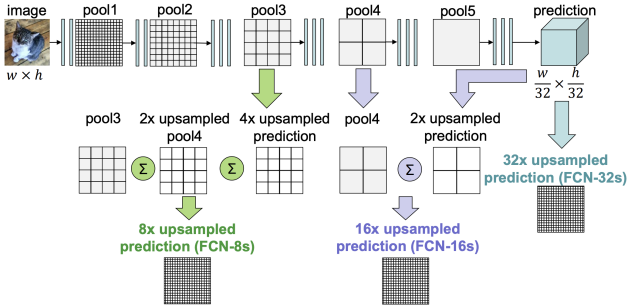


Рис. 2.11: Схема улучшения FCN путем использования промежуточных выходов сверточных слоев [7]

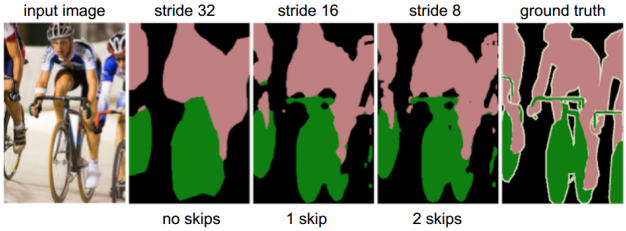


Рис. 2.12: Эффект от улучшения FCN путем использования промежуточных выходов сверточных слоев [8]

Реализпция улучшения качества вывода сети путем обработки выходных изображений сверточных слоев

Задача решается та же самая (задача семантической сегментации), поэтому весь служебный код останется таким же. Единственное, изменится структура сети и то не сильно.

Основная часть архитектуры идентична FCN, но выход сети будет формироваться на основе выходного слоя (переменная out) и на основе выхода 4го блока (out4). Если проследить изменение размеров каналов, то можно увидеть, что out меньше в 32

раза, чем исходное изображение, а out4 в 16 раз меньше, чем входное изображение. Поэтому для лучше точности сегментации мы возьмем и увеличим out в 2 раза, сложим с out4 и увеличим комбинацию в 16 раз.

Некоторое обоснование этого приведено выше. А код модели FCN_16s приведен ниже

```
class FCN_16s(nn.Module):
  def __init__(self):
    super().__init__()
    self.fcn_block1 = FCN_Block(3, 64)
    self.fcn_block2 = FCN_Block(64, 128)
    self.fcn_block3 = FCN_Block(128, 256,
                                three_conv_layers=True)
    self.fcn_block4 = FCN_Block(256, 512,
                                three_conv_layers=True)
    self.fcn_block5 = FCN_Block(512, 512,
                                three_conv_layers=True)

    self.equals_linear = FCN_Block(
        512,
        4096,
        kernel_size=(1, 1),
        padding=0,
        with_pool=False
        )
    self.conv1 = nn.Conv2d(4096, 1, (1, 1))

  def forward(self, x):
    out = self.fcn_block1(x)
    out = self.fcn_block2(out)
    out = self.fcn_block3(out)
    out4 = self.fcn_block4(out)
    out = self.fcn_block5(out4)
```

```
    out = self.equals_linear(out)

    out = self.conv1(out)
    out_2x_sampled = torch.nn.functional.upsample(
        out, scale_factor=2
    )
    out = torch.nn.functional.upsample(
        out_2x_sampled + out4, scale_factor=16
        )

    return out
```

Из аналогичных соображений немного преобразуем FCN, но уже с использованием двух промежуточных и одной выходной карт признаков: выход 4го блока увеличим в 2 раза, выход всей сети увеличим в 4 раза и сложим оба результата с выходом 4го блока. Это модель FCN_8s.

```
class FCN_8s(nn.Module):
  def __init__(self):
    super().__init__()
    self.fcn_block1 = FCN_Block(3, 64)
    self.fcn_block2 = FCN_Block(64, 128)
    self.fcn_block3 = FCN_Block(128, 256,
                                three_conv_layers=True)
    self.fcn_block4 = FCN_Block(256, 512,
                                three_conv_layers=True)
    self.fcn_block5 = FCN_Block(512, 512,
                                three_conv_layers=True)

    self.equals_linear = FCN_Block(
        512,
        4096,
        kernel_size=(1, 1),
```

```
        padding=0,
        with_pool=False
        )
    self.conv1 = nn.Conv2d(4096, 1, (1, 1))

  def forward(self, x):
    out = self.fcn_block1(x)
    out = self.fcn_block2(out)
    out3 = self.fcn_block3(out)
    out4 = self.fcn_block4(out3)
    out = self.fcn_block5(out4)
    out = self.equals_linear(out)

    out_final = self.conv1(out)
    out4_2x_upsampled = torch.nn.functional.upsample(
        out4, scale_factor=2
    )
    out_final_4x_upsampled = torch.nn.functional.upsample(
        out_final,
        scale_factor=4
        )
    out = torch.nn.functional.upsample(
        out4_2x_upsampled + out_final_4x_upsampled + out3,
        scale_factor=8
    )

    return out
```

Далее обучим FCN_16s уже хорошо знакомым вам способом.

```
model = FCN_16s().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
criterion = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.StepLR(
```

```
    optimizer,
    step_size=5,
    gamma=0.95
    )
train(
    model,
    optimizer,
    scheduler,
    criterion,
    100,
    train_dataloader,
    val_dataloader
    )
```

И FCN_8s обучим идентично модели FCN_16s

```
model = FCN_8s().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
criterion = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=5,
    gamma=0.95
    )
train(
    model,
    optimizer,
    scheduler,
    criterion,
    100,
    train_dataloader,
    val_dataloader
    )
```

2.3 Детекция объектов

2.3.1 Постановка задачи

Мы уже знакомы с задачей классификации. Для постановки задачи детекции нам потребуется определить также задачу локализации. Заключается она в определении границ объекта класса, который находится на картинке. Объект помещается в так называемую ограничивающую рамку (или bounding box). Чаще всего эта рамка — прямоугольник со сторонами, параллельными границам картинки. Как нетрудно догадаться, его можно задать, например, правой верхней и левой нижней точками или координатами центральной точки + шириной и высотой. А теперь мы можем сформулировать задачу детекции.

Задача детекции — задача обнаружения объекта класса на картинке и его локализации. Мы должны классифицировать объект на картинке и построить его ограничивающую рамку.

Для решения данной задачи строится набор ограничивающих рамок-гипотез. Предполагается, что в какую-то из них попал детектируемый объект. Затем они классифицируются, границы полученных изображений принимаются за начальные границы объекта, а после они уточняются. Уточнение границ будет описано ниже.

2.3.2 Вход и выход

На вход сети подается изображение с некоторым количеством заранее известных объектов одного класса. На выходе мы получаем изображение, на котором построены ограничивающие рамки вокруг объектов, и эти объекты классифицированы.

То есть сеть должна выделить всех кошек и собак на картинке, ограничить их рамкой, и определить, в каких из них кошка, а в каких собака.

В задаче детекции для оценки предсказаний используется метрика. Intersection over Union (IoU), значение которой расчитывается как отношение площадей пересечения истинной области и предсказания к их объединению. На рисунке 2.13 представлена

наглядная схема расчета IoU.

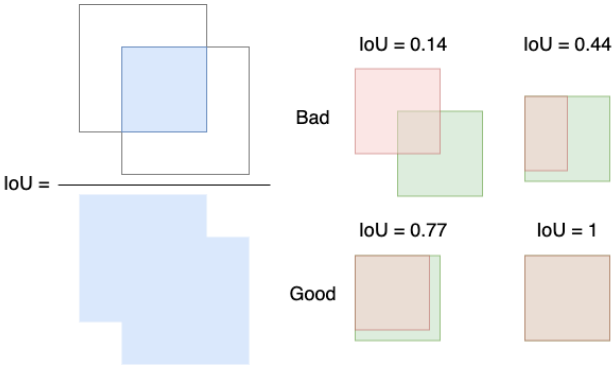


Рис. 2.13: Расчет IoU [9]

2.3.3 Актуальность

Актуальность задачи детекции является логическим продолжением некоторых приложений задачи семантической сегментации. Мы уже научились классифицировать объекты дорожной обстановки для, например, беспилотного автомобиля. Но автомобиль все еще не различает автомобили между собой – не может определить границы движущихся объектов. Решить данные проблемы и позволяет задача детекции.

Алгоритмы, решающие задачу детекции, также могут исопльзоваться в системах видеонаблюдения

2.3.4 R-CNN

R-CNN — Region-based Convolutional Network. Как нетрудно догадаться из названия, он также основан на сверточных нейронных сетях. Работает он следующим образом.

Рис. 2.14: Пример решения задачи детекции автомобиля [10]

Определение рамок-гипотез

На первом шаге алгоритм выделяет регионы, где предположительно может находиться объект. Обычно в этом алгоритме используется selective search. Он выделяет интересные для дальнейшего рассмотрения регионы по интенсивности пикселей, перепаду цветов и контраста. Он может работать уже с сегментированным изображением — на нем четко различаются пиксели разных объектов.

На выходе этого алгоритма мы получаем набор гипотез — регионов (отсюда и region-based), в которых предположительно может находиться объект. Класс объекта на этом шаге не важен. Размеры и соотношение сторон регионов в наборе различны (это не окно фиксированного размера). Количество таких регионов не должно быть очень велико ввиду вычислительных затрат на решение задачи — многие источники ссылаются на использование порядка 2000 регионов на картинку.

В предложенном ниже решении задачи детекции есть существенное упрощение этого шага — рамки-гипотезы выбираются случайным образом в предположении, что какой-то из прямоугольников приемлемым образом ограничит объект.

Векторное представление гипотез

Следующим шагом требуется получить векторное представление конкретного региона. Каждый из регионов подается на вход сверточной нейронной сети, например, AlexNet, но без последнего

softmax-слоя. Последний слой не используется как раз потому, что на выходе данного шага мы хотим получить векторное представление изображения.

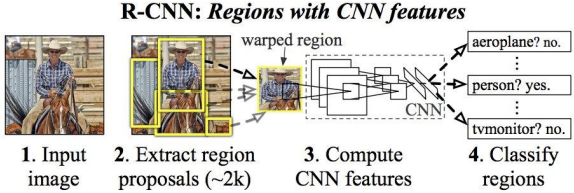


Рис. 2.15: Схема R-CNN [11]

Вход AlexNet имеет размерность $3 \times 227 \times 227$, а размер региона может быть почти любого соотношения сторон и размера — эта проблема решается сжатием или растягиванием входа до нужного размера.

Классификация гипотез

Мы получили для каждой гипотезы вектор. На данном шаге регион еще не классифицирован. Это векторное представление региона классифицируется методом опорных векторов (SVM) — он проводит лучшую плоскость (или n-мерную плоскость), разделяющую два класса, максимизируя расстояние от плоскости до ближайших объектов класса. Рассмотрим его немного подробнее.

Для простоты описания алгоритма рассмотрим двумерный случай. Пусть мы имеем точки на плоскости — каждая из них задает один из векторов, которые мы классифицируем. Данный алгоритм проводит между ними прямую линию, которая наилучшим образом разделит точки разных классов. В многомерном случае решением является гиперплоскость.

SVM строит функцию

$$F(\mathbf{x}) = sign((\mathbf{w}, \mathbf{x} + b)$$

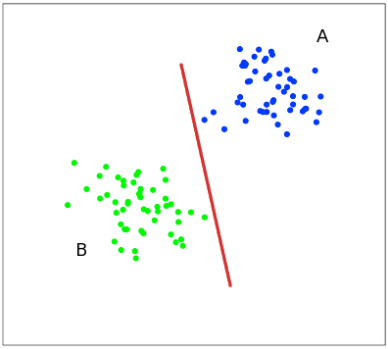


Рис. 2.16: Пример выхода SVM-классификатора [12]

, где $\mathbf{w}$ — нормальный вектор к прямой (или гиперплоскости в многомерном случае). Если $F(\mathbf{x}) = 1$, то мы считаем, что объект принадлежит некоторому классу 1. Если же $F(\mathbf{x}) = -1$, то на картинке класс 2

Этот алгоритм решает задачу бинарной классификации, но нам требуется классифицировать некоторое количество классов N. Для этого мы вводим еще один класс — фон. И тогда учим алгоритм отличать объект класса от фона. То есть класс 1 — объект одного из классов. Класс -1 — класс «фон». Для каждого из классов мы получаем свой классификатор. Каждая из N-моделей дает предсказание о том, принадлежит ли объект некоторому классу. Для получения предсказания используется принцип «один против всех» (OvR - One vs. Rest): для каждого классификатора считается расстояние для полученной прямой (гиперплоскости), и выбирается тот класс, для которого расстояние для соответствующей разделяющей прямой (гиперплоскости) максимально.

Таким образом, на выходе алгоритма мы имеем N+1 мерный вектор, который содержит информацию о том, есть ли объект конкретного класса внутри региона. Запустив алгоритм для всех

регионов, мы получаем матрицу таких векторов, столбцы которой есть предсказание SVM для соответствующих регионов.

Уточнение границ объекта

Некоторый результат можно получить уже из имеющихся данных. Однако важным нюансом является тот факт, что ограничивающие рамки выбираются лишь из тех регионов, что были получены на первом шаге. Понятно, что такие предсказания не очень точны — объекты в рамках могут быть неудачно обрезаны, четко определить границы таким алгоритмом нельзя. Следующий шаг призван уточнить границы регионов для улучшения предсказания.

Границы объектов уточняются с помощью линейной регрессии, на вход авторы алгоритма подают выход из последнего MaxPooling слоя, поскольку карта признаков несет информацию о местоположении объекта, чего нельзя сказать о векторе.

Притом, ввиду возможных особенностей объектов разных классов (характерное соотношение сторон, характерный размер), на каждый из классов строится собственных регрессор для уточнения границ. В каждом классе регрессор представляет собой 4 функции — две на поправки к координатам, две — на поправки к ширине и высоте. Поправки к ширине и высоте следует нормировать на исходную ширину, чтобы избежать зависимости от начального размера рамок.

Стоит оговориться, что нет смысла строить регрессию на границы фона, поэтому она уточняет лишь те предсказания, которые содержат объект некоторого иного класса.

Реализация R-CNN

В данном примере на основе датасета MNIST мы сгенерируем датасет для решения задачи детектирования. Мы создадим набор картинок, где в качестве фона используется простой гауссов шум, а на этом фоне будут находиться несколько чиселок из датасета MNIST (на каждой картинке разное количество чиселок).

Стоит оговориться, что для упрощения примера, в данной

реализации отсутствует регрессия для коррректирования границ объекта — предсказание выбирается из изначально сгенерированных предположений.

Приступим к решению задачи, для начала к генерации датасета.

Импортируем библиотеки

```
import random
import numpy as np
import skimage.filters
from skimage.transform import resize
from torch.utils.data import TensorDataset, DataLoader
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
```

Загрузим, наверняка уже знакомый вам, датасет MNIST. Это сделано с использованием библиотеки tensorflow чисто для удобства.

```
(train_x, train_y), (test_x, test_y) = \
    tf.keras.datasets.mnist.load_data()

train_x = train_x.reshape(-1, 28, 28, 1).astype(np.float32) / 255.
test_x = test_x.reshape(-1, 28, 28, 1).astype(np.float32) / 255.
```

Сгенерируем большую картинку с гауссовым шумом, от которой впоследствии будем отрезать случайные кусочки, чтобы шум на картинках также был несколько разным.

```
bg_source = np.random.rand(1000, 1000, 1).astype(np.float32)
bg_source = skimage.filters.gaussian(bg_source, 4)
bg_source = (bg_source - np.min(bg_source))/\
```

```
(np.max(bg_source) - np.min(bg_source))
bg_source = np.clip(bg_source, 0, 1)
```

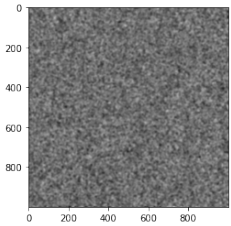


Рис. 2.17: Пример сгенерированного гауссвого шума

Следующей функцией сгенерируем новый датасет. Каждая картинка будет иметь размер new_size. Каждая картинка формируется следующим образом:

1. От большой картинки с шумом отрезается кусок размером new_size.
2. Генерируем случайно количество объектов на картинке num_of_objects
3. Затем в цикле (num_of_objects раз) генерируем индекс картинки из датасета MNIST и вставляем ее в случайное место.

По-хорошему хотелось бы следить за тем, что цифры не накладываются сильно друг на друга, но так как мы взяли достаточно большой размер картинки в нашем датасете new_size = (128, 128) и достаточно маленькое количество чисел на картинке от 1 до 3, то вероятность наложения двух цифр очень маленькая. Из этих соображений не будем обращать внимания на этот нюанс.

```
def gen_od_ds(
        x,
        y,
```

```
        bg_source,
        new_size,
        num_samples,
        max_num_objects
        ):
    x_new = np.zeros(
        (num_samples, new_size[0], new_size[1], 1),
        dtype=np.float32
        )
    y_new = []

    rh = float(x.shape[1]) / new_size[0]
    rw = float(x.shape[2]) / new_size[1]

    for i in range(num_samples):
        oh = random.randint(0, bg_source.shape[0] - new_size[0])
        ow = random.randint(0, bg_source.shape[1] - new_size[1])
        x_new[i] = bg_source[oh:oh+new_size[0],
                                    ow:ow+new_size[1], :]
        num_of_objects = random.randint(1, max_num_objects)
        y_new.append([])
        for _ in range(num_of_objects):
            sample_idx = random.randint(0, x.shape[0]-1)
            xs = x[sample_idx]
            ys = y[sample_idx]
            ofs = (random.randint(0, x_new[i].shape[0]-xs.shape[0]),
                   random.randint(0, x_new[i].shape[1]-xs.shape[1]))
            x_new[i][ofs[0]:ofs[0]+xs.shape[0],\
                     ofs[1]:ofs[1]+xs.shape[1], :] += xs
            ry = float(ofs[0]) / new_size[0]
            rx = float(ofs[1]) / new_size[1]
            y_new[-1].append([ys, ry, rx, rh, rw])

        x_new[i] = np.clip(x_new[i], 0.0, 1.0)
```

```
38
39      return x_new, y_new
```

Собственно, сгенерируем наш датасет (тренировочный и валидационный).

```
new_size = (128, 128)
train_num_samples = 60000
test_num_samples = 10000
max_num_objects = 3
# максимальное кол-во цифр на одном изображении

train_x_det, train_y_det = gen_od_ds(
    train_x,
    train_y,
    bg_source,
    new_size,
    train_num_samples,
    max_num_objects
    )
test_x_det, test_y_det = gen_od_ds(
    test_x,
    test_y,
    bg_source,
    new_size,
    test_num_samples,
    max_num_objects
    )

print(train_x.shape)
print(len(train_y))
print(test_x.shape)
print(len(test_y))
```

Напишем функцию для красивого отображения предсказаний модели.

```
def show_prediction(img, preds):
    import matplotlib.patches as patches

    fig,ax = plt.subplots(1)
    ax.imshow(img[...,0], 'gray', vmin=0, vmax=1,)

    for pred in preds:

        pred_cls = pred[0]
        ry, rx, rh, rw = pred[1:]

        box_y = int(ry * img.shape[0])
        box_x = int(rx * img.shape[1])
        box_w = int(rw * img.shape[1])
        box_h = int(rh * img.shape[0])

        rect = patches.Rectangle(
            (box_x, box_y),
            box_w,
            box_h,
            linewidth=1,
            edgecolor='r',
            facecolor='none'
            )
        ax.add_patch(rect)

        rect = patches.Rectangle(
            (box_x, box_y),
            10,
            -10,
            linewidth=1,
            edgecolor='r',
            facecolor='r'
            )
```

```
        ax.add_patch(rect)
        ax.text(box_x+4, box_y-2, pred_cls)


idx = 0
show_prediction(train_x_det[idx], train_y_det[idx])
```

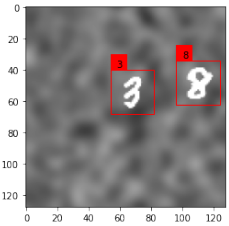


Рис. 2.18: Отрисовка написанной выше функцией предсказания модели

Вообще, вопрос генерации предполагаемых областей не тривиальный. Чаще всего для этого используются умные алгоритмы. Мы на этом останавливаться не будем и просто будем генерировать абсолютно случайные области-гипотезы (proposal). Также добавляя к ним верные области, в которых содержатся нужные объекты.

```
def gen_fake_proposals(
        x_det, y_det, ori_shape, num_proposals
    ):
    all_proposals = []
    for boxes in y_det:
        proposals = []
        for box in boxes:
            proposals.append(box[1:])
```

```
        extra_prop_num = num_proposals - len(proposals)
        for _ in range(extra_prop_num):
            proposal_shape = (
                ori_shape[0] + random.randint(-8, 8),
                ori_shape[1] + random.randint(-8, 8))
            ofs = (random.randint(0,
                                  x_det.shape[1]-proposal_shape[0]),
                   random.randint(0,
                                  x_det.shape[2]-proposal_shape[1]))

            rh = float(proposal_shape[0]) / x_det.shape[1]
            rw = float(proposal_shape[1]) / x_det.shape[2]
            ry = float(ofs[0]) / x_det.shape[1]
            rx = float(ofs[1]) / x_det.shape[2]
            proposals.append([ry, rx, rh, rw])

        random.shuffle(proposals)
        all_proposals.append(proposals)

    return all_proposals

ori_shape = (28, 28)
num_proposals = 8 # количество proposals на одну картинку
test_proposals = gen_fake_proposals(
    test_x_det,
    test_y_det,
    ori_shape,
    num_proposals
    )

```

Напишем функцию для отображения всех proposals на картинке

```
def show_proposals(img, proposals):
    import matplotlib.patches as patches

    fig,ax = plt.subplots(1)
    ax.imshow(img[...,0], 'gray', vmin=0, vmax=1,)

    for proposal in proposals:

        ry, rx, rh, rw = proposal

        box_y = int(round(ry * img.shape[0]))
        box_x = int(round(rx * img.shape[1]))
        box_w = int(round(rw * img.shape[1]))
        box_h = int(round(rh * img.shape[0]))

        rect = patches.Rectangle(
            (box_x, box_y),
            box_w,
            box_h,
            linewidth=1,
            edgecolor='r',
            facecolor='none'
            )
        ax.add_patch(rect)

idx = 0
show_proposals(test_x_det[idx], test_proposals[idx])
```

Следующая функция сгенерирует датасет для классификации. Вообще задача решается в таком виде:

1. Обучается классификатор изображений, умеющий определять класс «фон» и цифры на этом фоне.
2. Этот классификатор будет получать на вход proposals и для каждого из них определять класс.

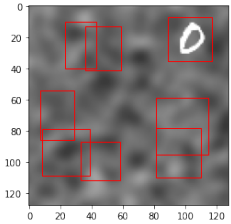


Рис. 2.19: Вывод функции show_proposals – рамки-гипотезы, среди которых сеть ищет объект

3. Те proposals, которые не относятся к классу «фон», те считаются границами задетектированных объектов.

```
def gen_classification_ds(
    x_det,
    y_det,
    inp_size,
    bg_source,
    bg_samples_num
    ):
    imgs = []
    labels = []
    for i in range(len(x_det)):
        if i % 10000 == 0:
            print('[{} / {}]'.format(i, len(x_det)))
        img = x_det[i]
        for box in y_det[i]:
            pred_cls = box[0]
            ry, rx, rh, rw = box[1:]

            box_y = int(round(ry * img.shape[0]))
            box_x = int(round(rx * img.shape[1]))
            box_w = int(round(rw * img.shape[1]))
```

```
            box_h = int(round(rh * img.shape[0]))
            img_sub = img[box_y:box_y+box_h,
                          box_x:box_x+box_w, :]
            if img_sub.shape[0] == inp_size[0] \
                    and \
                    img_sub.shape[1] == inp_size[1]:
                img_inp = img_sub
            else:
                img_inp = resize(img_sub, (inp_size[0], inp_size[1]),
                    order=3, mode='reflect', anti_aliasing=True)
            imgs.append(img_inp)
            labels.append(pred_cls)

    for i in range(bg_samples_num):
        if i % 10000 == 0:
            print('[{} / {}]'.format(i, bg_samples_num))
        ofs = (random.randint(0, bg_source.shape[0]-inp_size[0]),
               random.randint(0, bg_source.shape[1]-inp_size[1]))
        imgs.append(bg_source[ofs[0]:ofs[0]+\
                              inp_size[0], ofs[1]:ofs[1]+inp_size[1], :])
        labels.append(10) # bg

    x_cls = np.stack(imgs)
    y_cls = np.stack(labels)
    return x_cls, y_cls


bg_samples_num = 100000 # Количество образцов "фона"
inp_size = (28, 28) # Размер входа в нейросеть

train_x_cls, train_y_cls = gen_classification_ds(
    train_x_det,
    train_y_det,
    inp_size,
```

```
    bg_source,
    bg_samples_num
    )
```

А дальше уже все будет иметь знакомый нам вид. Напишем свой класс Dataset, в котором для фичей будем переставлять размерность с номером канала в начало.

Создадим объекты классов Dataset и DataLoader.

```
batch_size = 50

class CustomDataset(TensorDataset):
  def __init__(self, features, targets):
    self.features = features
    self.targets = targets

  def __len__(self):
    return len(self.features)

  def __getitem__(self, idx):
    return np.array(np.rollaxis(
        self.features[idx], 2, 0
        ),\
        np.float32),\
        self.targets[idx]

train_dataset = CustomDataset(train_x_cls, train_y_cls)
train_dataloader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True
    )

val_dataset = CustomDataset(test_x_det, test_y_det)
```

```
val_dataloader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    shuffle=True
    )
```

Напишем саму модель R-CNN. Несложно заметить и вспомнить, что ожидается фиксированный размер изображения (это видно из заданного размера входа в self.dense1).

```
import torch
from torch import nn
NUM_CLASSES = 11
class R_CNN(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = nn.Conv2d(1, 32, (5, 5))
    self.pool1 = nn.MaxPool2d((2, 2), (2, 2))

    self.conv2 = nn.Conv2d(32, 64, (5, 5))
    self.pool2 = nn.MaxPool2d((2, 2), (2, 2))

    self.flatten = nn.Flatten()

    self.dense1 = nn.Linear(1024, 256)
    self.dense2 = nn.Linear(256, NUM_CLASSES)

  def forward(self, x):
    out = F.relu(self.conv1(x))
    out = self.pool1(out)

    out = F.relu(self.conv2(out))
    out = self.pool2(out)
    out = self.flatten(out)
```

```
25
26      out = F.relu(self.dense1(out))
27      out = self.dense2(out)
28      return out
```

Напишем также саму функцию применения модели к картинке с некоторым набором proposals. Картинка будет отображаться вместе с регионами только для задетектированных объектов.

```
1   def rcnn_prediction(img, proposals, inp_size, model):
2       predictions = []
3       for proposal in proposals:
4           ry, rx, rh, rw = proposal
5           box_y = int(round(ry * img.shape[0]))
6           box_x = int(round(rx * img.shape[1]))
7           box_w = int(round(rw * img.shape[1]))
8           box_h = int(round(rh * img.shape[0]))
9           img_sub = img[box_y:box_y+box_h,
10                        box_x:box_x+box_w, :]
11          img_inp = resize(img_sub, (inp_size[0], inp_size[1]),
12              order=3, mode='reflect', anti_aliasing=True)
13          img_inp = np.rollaxis(img_inp, 2, 0)
14          img_picture = img_inp[None, ...]
15          x = torch.from_numpy(img_picture).to(device)
16
17          pred = model(x)
18          pred = pred[0]
19          pred_cls = np.argmax(pred.detach().cpu().numpy())
20          if pred_cls != 10:
21              predictions.append([pred_cls] + proposal)
22      return predictions
```

Напишем знакомую вам функцию тренировки сети. Ничего нового опять не наблюдается.

```
from time import time
import torch.nn.functional as F
from matplotlib import pyplot as plt
from IPython.display import clear_output
from tqdm.notebook import tqdm


def train_detection(
    model,
    optimizer,
    scheduler,
    loss_fn,
    epochs,
    data_tr,
    val_data
    ):
    loss_values = []
    epoch_num = []

    for epoch in range(epochs):
        tic = time()
        print('* Epoch %d/%d' % (epoch+1, epochs))

        scheduler.step()
        avg_loss = 0
        model.train()  # train mode
        for X_batch, Y_batch in tqdm(data_tr):
            X_batch = X_batch.to(device)
            Y_batch = Y_batch.to(device)
            # data to device
            optimizer.zero_grad()
            # set parameter gradients to zero

            # forward
```

```
            Y_pred = model(X_batch)
            Y_batch = F.one_hot(Y_batch)

            loss = loss_fn(Y_pred, Y_batch.float()) # forward-pass
            loss.backward()  # backward-pass
            optimizer.step()  # update weights

            # calculate loss to show the user
            avg_loss += loss / len(data_tr)
        toc = time()
        loss_values.append(avg_loss.item())
        epoch_num.append(epoch)

        with torch.no_grad():
          model.eval()  # testing mode
          # Visualize tools
          clear_output(wait=True)
          plt.plot(epoch_num, loss_values)
          plt.title('Loss values')
          plt.xlabel('epoch')
          plt.ylabel('loss')
          plt.show()
          for idx in [0, 50, 1000, 3, 1]:
            img = test_x_det[idx]
            labels_true = test_y_det[idx]
            proposals_img = test_proposals[idx]

            preds = rcnn_prediction(
                img, proposals_img, (28, 28), model
            )
            show_prediction(img, preds)

          print(
```

```
68              'loss: %f' % avg_loss, 'LR: %f' % scheduler.get_lr()[0])
```

Создадим объект класса R_CNN и запустим обучение.

```
device = torch.device(
    'cuda' if torch.cuda.is_available() else 'cpu'
)
print(device)
model = R_CNN().to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
criterion = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=5,
    gamma=0.95
    )
train_detection(
    model,
    optimizer,
    scheduler,
    criterion,
    10,
    train_dataloader,
    val_dataloader
    )
```

Проведем тестовый запуск для объекта idx из тестового датасета.

```
idx = 10
img = test_x_det[idx]
labels_true = test_y_det[idx]
proposals_img = test_proposals[idx]

preds = rcnn_prediction(img, proposals_img, (28, 28), model)
show_prediction(img, preds)
```

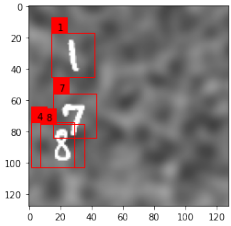


Рис. 2.20: Предсказание модели

Данный алгоритм на практике используется достаточно редко, поскольку имеет существенные недостатки:

- Вычислительная сложность. Один вызов сети может потребовать нескольких минут. Для решения задач, например, автопилотирования, это недопустимо долго.
- Регионы, определенные в начале алгоритма, могут быть очень близкими друг к другу. Выход сети в этом случае будет также близок, а каждый вызов требует немалого количества времени.
- Алгоритм определения начальных регионов никак не обучается, это не позволяет улучшить его качество.

Таким образом, имеет смысл перейти к более продвинутым

вариантам алгоритма, лишенным некоторых из описанных недостатков.

2.3.5 Fast R-CNN

Алгоритм, о котором пойдет речь в этой главе, значительно продвигает самый существенный недостаток R-CNN — время на обучение и вызов сети. В R-CNN каждый из множества выбранных регионов проходит через сеть CNN. Fast R-CNN несет в себе примерно ту же идею со следующими изменениями:

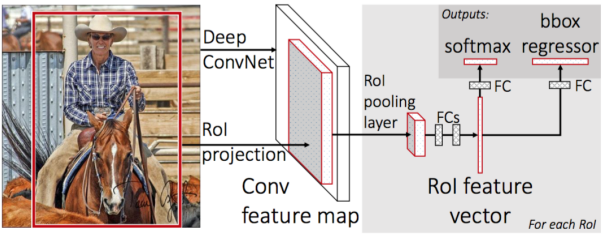


Рис. 2.21: Схема Fast R-CNN [13]

- Карта признаков извлекается не из каждого региона, а из всего изображения
- Аналогично R-CNN некоторым алгоритмом, например, selective-search, находится набор регионов - предположений
- Каждое такое предположение проходит RoI (Region of Interest) слой, который сопоставляет границы рамок с точками на карте признаков, и делит выделенный кусочек карты признаков на заданное количество прямоугольников, применяя на каждом из них max pooling. Таким образом, на выходе слоя мы имеем картинку фиксированного размера и любое предположение может быть входом для полносвязного слоя (2.24).
- Далее каждая гипотеза классифицируется полносвязным слоем и уточняются границы рамок, аналогично R-CNN

Замечание: В R-CNN использовался SVM-классификатор, от выхода которого зависело уточнение координат ограничивающих рамок. В этой же версии алгоритма изображения классифицируются одним softmax выходом размерности N+1. Это позволяет запускать регрессоры на границы рамок и классификаторы параллельно, увеличивая скорость работы алгоритма.

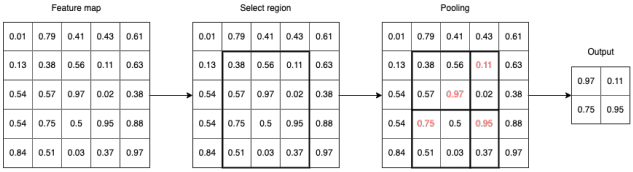


Рис. 2.22: Пример работы RoI слоя [14]

Реализация Fast R-CNN

Здесь определим класс модели. На этапе обучения по сути ничего не меняется, но есть один нюанс уже на этапе применения модели.

Мы не будем прогонять сеть для каждого региона-гипотезы отдельно. Мы рассчитаем карты признаков для всего изображения сразу, затем пропорционально изменению размера уменьшим все proposals и получим карты признаков для каждого из них. Которые уже можно применять для определения лейбла для каждого региона в отдельности.

Это можно сделать благодаря устройству операций пулинга и свертки.

```
from torch import nn
import torch.nn.functional as F
import torch
import torchvision
import matplotlib.pyplot as plt
%matplotlib inline
```

```
from IPython.display import clear_output

NUM_CLASSES = 11

class Fast_R_CNN(nn.Module):

    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(1, 32, (5, 5), padding=2)
        self.conv2 = nn.Conv2d(32, 64, (5, 5), padding=2)
        self.dense1 = nn.Linear(3136, 256)
        self.dense2 = nn.Linear(256, NUM_CLASSES)
        self.max_pool = nn.MaxPool2d((2, 2), (2, 2))
        self.flatten = nn.Flatten()

    def forward(self, inp, proposals=None):

        if proposals is None: # Режим обучения

            out = F.relu(self.conv1(inp))
            out = self.max_pool(out)
            out = F.relu(self.conv2(out))
            out = self.max_pool(out)
            out = self.flatten(out)
            out = F.relu(self.dense1(out))
            out = self.dense2(out)
            return out

        else: # Режим предсказания

            assert inp.shape[0] == 1 # Только batch size = 1
            predictions = []
            roi_pool_size = (7, 7)
```

```
        # Извлечение признаков из всей картинки
        out = F.relu(self.conv1(inp))
        out = self.max_pool(out)
        feat = F.relu(self.conv2(out))

        # Для каждого пропозала
        for proposal in proposals:

            # Отображение координат изображения
            # в координаты пространства признаков
            ry, rx, rh, rw = proposal
            box_y = int(round(ry * int(feat.shape[1])))
            box_x = int(round(rx * int(feat.shape[2])))
            box_w = int(round(rw * int(feat.shape[2])))
            box_h = int(round(rh * int(feat.shape[1])))

            # Вырезаем признаки, относящиеся к пропозалу
            feat_sub = feat[:, :, box_y:box_y+box_h,
                                  box_x:box_x+box_w]

            # Аналог ROI Pooling
            feat_pooled = torchvision.transforms.functional.resize(
              feat_sub,
              (roi_pool_size[0], roi_pool_size[1]),
              torchvision.transforms.InterpolationMode.BILINEAR)

            # Финальная классификация
            out = self.flatten(feat_pooled)
            out = F.relu(self.dense1(out))
            out = self.dense2(out)
            # Фильтрация класса "фон"
            assert out.shape[0] == 1 # Только batch size = 1
            pred = out[0]
```

```
                pred_cls = np.argmax(pred.cpu().detach().numpy())
                if pred_cls != 10:
                    predictions.append([pred_cls] + proposal)

        return predictions
model = Fast_R_CNN()
```

Напишем функцию для инференса Fast-R-CNN. Логика может стать несколько сложнее, поэтому лучше такое выносить в отдельную функцию.

```
def fast_rcnn_prediction(img, proposals, inp_size, model):
    predictions = model(img, proposals)
    return predictions
```

Напишем функцию для тренировки Fast-R-CNN

```
def fast_rcnn_train_detection(
    model,
    optimizer,
    scheduler,
    loss_fn,
    epochs,
    data_tr,
    val_data
    ):
    loss_values = []
    epoch_num = []

    for epoch in range(epochs):
        tic = time()
        print('* Epoch %d/%d' % (epoch+1, epochs))

        scheduler.step()
```

```
        avg_loss = 0
        model.train()  # train mode
        for X_batch, Y_batch in tqdm(data_tr):
            X_batch = X_batch.to(device)
            Y_batch = Y_batch.to(device)
            # data to device
            optimizer.zero_grad()

            # set parameter gradients to zero

            # forward
            Y_pred = model(X_batch)
            Y_batch = F.one_hot(Y_batch)

            loss = loss_fn(Y_pred, Y_batch.float()) # forward-pass
            loss.backward()  # backward-pass
            optimizer.step()  # update weights

            # calculate loss to show the user
            avg_loss += loss / len(data_tr)

        toc = time()
        loss_values.append(avg_loss.item())
        epoch_num.append(epoch)

        with torch.no_grad():
          model.eval()  # testing mode

          # Visualize tools
          clear_output(wait=True)
          plt.plot(epoch_num, loss_values)
          plt.title('Loss values')
          plt.xlabel('epoch')
          plt.ylabel('loss')
```

```
        for idx in [0, 50, 1000, 3, 1]:
          img = np.rollaxis(test_x_det[idx], 2, 0)
          labels_true = test_y_det[idx]
          proposals_img = test_proposals[idx]

          preds = fast_rcnn_prediction(
              torch.Tensor(img[None, ...]),
              proposals_img,
              (28, 28),
              model
              )
          show_prediction(test_x_det[idx], preds)
        plt.show()
        print(
          'loss: %f' % avg_loss, 'LR: %f' % scheduler.get_lr()[0])
```

Наконец, запустим обучение Fast-R-CNN

```
from time import time
from tqdm.notebook import tqdm

device = torch.device(
    'cuda' if torch.cuda.is_available() else 'cpu'
)
print(device)
model = Fast_R_CNN().to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
criterion = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=5,
    gamma=0.95
```

```
    )
fast_rcnn_train_detection(
    model,
    optimizer,
    scheduler,
    criterion,
    10,
    train_dataloader,
    val_dataloader
    )
```

2.3.6 Faster R-CNN

Следующим шагом в улучшении данной архитектуры стала разработка нового метода локализации объекта, который заменил бы selective search. В качестве такого алгоритма авторы предложили Region Proporsal Networks (RPN).

В основе алгоритма лежит система якорей. Якорями авторы называют области изображения, имеющие разные соотношения сторон и разные размеры. Авторы использовали три разных соотношения сторон и три размера.

Пробегая скользящим окном по карте признаков, на основе метрики IoU делается вывод о наличии объекта в текущем регионе. Значит, алгоритм Faster R-CNN работает следующим образом:

- Изображение подается на вход сверточной нейронной сети, на выходе получаем карту признаков
- Карта признаков обрабатывается RPN – скользящее окно проходит по карте признаков, выделяются регионы, где есть объект
- Карта признаков с полученными объектами передается слою ROI с последующей обработкой полносвязными слоями и классификацией

- Далее, также как и в алгоритмах выше, строится регрессия на смещение ограничивающих рамок

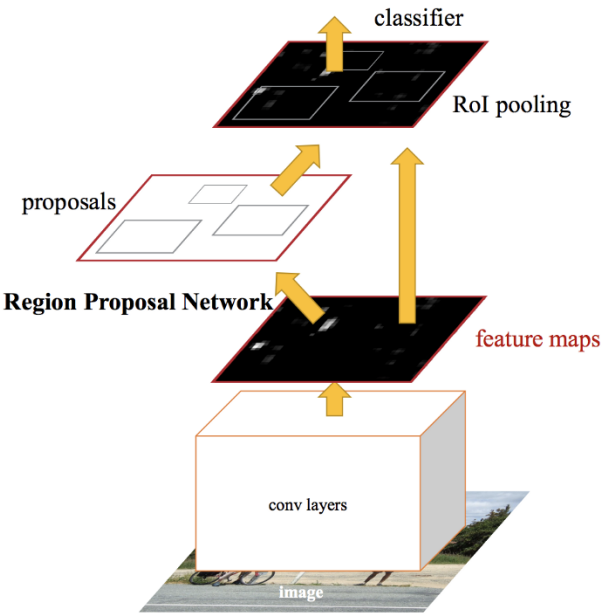


Рис. 2.23: Схема Faster R-CNN [15]

Видно, данный алгоритм отличается от Fast R-CNN лишь совершенствованным алгоритмом определения регионов, где мы ищем объект. Точность определения начальных регионов с использованием этого алгоритма несколько ниже, но его использование позволяет значительно уменьшить время на обучение сети и на ее вызов.

Как было описано выше, высокая скорость работы сети важна для решения задачи детектирования в реальном времени, например, для автопилотируемых систем. Из описанных архитектур

лучшим образом для решения данной задачи подходит именно Faster R-CNN. Для наглядности ниже представлено затрачиваемое на тестовый вызов сети время в секундах

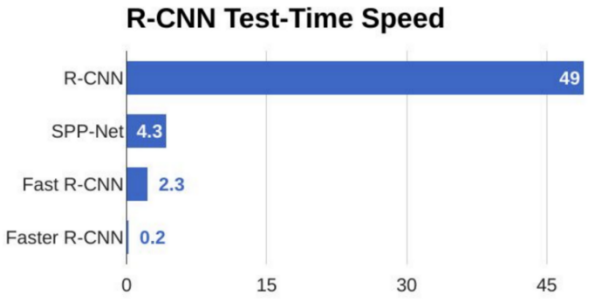


Рис. 2.24: Сравнение затрачиваемого на вызов разных сетей времени [16]

3. Введение в NLP

3.1 Мотивация

Современный человек получает преобладающую часть информацию из трех каналов: фото, звук и текст. Через визуальный канал (т.е. фото и видео) мы получаем 90% всей информации. Следующим по объему передаваемой информации - текст. Как не сложно заметить в нашем доступе очень много текстовой информации, например, книги, страницы в интернете, переписки с друзьями. Также в различных компаниях очень много специфичной текстовой информации, например, логов различных серверов.

Объемы текстовых данных вынуждают нас начать их анализировать, научиться искать в этих объемах информации что-то полезное и качественное, научиться пользоваться этой информацией - золотом XXI века. И мы учимся! И даже научились На данный момент процессы обработки естественного языка плотно вошли в нашу жизнь: алгоритмы поиска в Yandex; процессы коммуникации с чат-ботами в VK, Telegram или ГосУслугах; системы автодополнения в каждом смартфоне, когда подходящее слово появляется по середине клавиатуры; системы фильтрации спама и многое-многое другое.

Для многих неожиданно, но эти системы обработки естественного языка начали закрепляться в нашей жизни сравнительно недавно: примерно в 2013 году. Сначала поиск в поисковых системах напоминал лишь работу с предметным указателем, но на данный момент система является более интеллектуальной. Даже на запрос, сформулированный простым языком, с высокой долей вероятности вы получите достаточно релевантный ответ.

И это результат работы исследователей в течение более чем 8 лет. Если эта область будет развиваться такими темпами, то даже сложно представить, что ожидает нас в будущем.

Давайте изучим основные моменты и подходы, часто используемые при решении задач NLP. Кто знает, может именно вы через года 4 удивите мир своей разработкой

3.2 План

В этой главе мы с вами изучим основы алгоритмов и принципов обработки естественного языка:

- поговорим про препроцессинге текста для решения вышеупомянутых задач;
- поговорим про известное утверждение «король - мужчина + женщина = королева»;
- поговорим о разных типах задач обработки текстов;
- рассмотрим основные архитектуры нейронных сетей для работы с текстами, а вообще говоря и с последовательностями вцелом (ведь текст - это частный случай последовательности).

Только проблема в том, что на первый взгляд текст кажется очень сложным объектом для исследования. Почему? Сразу можно выделить ряд причин

- Зависимость смысла слов от контекста
- Многозначность слов
- Свобода порядка слов в предложении (в разной степени для разных языков)

А как вообще компьютер должен понимать буквы, слова, предложения? Это же не числа, как возраст человека или цена дома, да и не картинки, которые, как мы с вами помним, также являются числами. Так как же работать с текстами? На этот вопрос мы с вами и ответим в этой главе.

3.3 Первичные подходы к NLP

Давайте отвлечемся от наших современных методов и посмотрим, как можно без применения «высокого» интеллекта довольно успешно решать некоторые задачи анализа естественного языка.

Конечно, методы, которые мы обсудим, сегодня не применяются, так как уже есть более мощные аналоги, но для общего понимания стоит разобраться с классическими приемами.

Система автодополнения. Вероятностный подход

Рассмотрим например системы автодополнения, которые сейчас есть в каждом смартфоне. Как построить простую систему, с достаточно высокой вероятностью верно подсказывающую пользователю слова, которые он хочет написать? Подумайте минутку, может у вас получится придумать подход к этой задаче.

Давайте заметим, что в текстах некоторые комбинации слов встречаются чаще, чем другие. Например, если взять очень много вариантов диалога между двумя людьми, то комбинация слов «тебя дела» встретится существенно чаще, чем «тебя бассейн». Таким образом, если пользователь ввел слово «тебя», то система с большей вероятностью будет права, предположив, что следующим словом будет слово «дела», а не «бассейн».

Итак, из теории вероятностей следует, что, если собрать большой датасет с различными текстами из переписок, то можно вполне точно оценивать вероятность того, какое слово пользователь хочет дальше написать. Например, можно оценить вероятность того, что после слова «тебя» пользователь хочет написать слово «дела».

$$P_{\text{дела} \mid \text{тебя}} = P(\text{появления слова "дела" после слова "тебя" }) = \frac{P(\text{"тебя дела" })}{P(\text{"тебя" })}$$

В силу большого количества текстов с хорошей точностью вероятность $P_{\text{дела} \mid \text{тебя}}$ можно оценить частотно: т.е. найти количество случаев «тебя дела» во всем наборе текстов (или корпусе), который у нас есть, это будет N, и количество слов «тебя», это будет M. Тогда, очевидно, что в случае достаточного размера корпуса с достаточной точностью

$$P_{\text{дела} \mid \text{тебя}} \approx \frac{N}{M}$$

где N — количество случаев в корпусе «тебя дела», M — количество случаев «тебя».

Тогда мы можем легко построить систему для автодополнения, потому что мы легко можем посчитать указанную выше вероятность.

- Человек, начинает писать какое-то сообщение
- Система берет последние написанные несколько слов (может одно слов)
- Система по корпусу ищет слово, которое с наибольшей вероятностью подразумевается пользователем после взятых слов

Чат-бот. Регулярные выражения

Наверняка каждый из вас сталкивался с чат-ботами. Иногда это был приятный опыт, иногда не очень, но, вероятно, вам доводилось общаться с различного рода чат-ботами.

Но знакомы ли вы с такой вещью, как «регулярное выражение»?

Регулярное выражение — строковое выражение, записанное в специальном виде. Регулярные выражения позволяют с использованием специальных библиотек наиболее эффективно выстраивать процессы обработки и анализа текстов. Так, с их использованием легко и эффективно можно найти «предложения-приветствия». Например, будем считать приветствием строку, удовлетворяющую следующему регулярному выражению:

```
1 import re
2
3 reg_exp = re.compile(r"(hello|hey|hi)[\ ]*([a-zA-Z]*)")
```

Данному регулярному выражению будет удовлетворять строка, которая начинается со слова hello, hey или hi, после которых будет идти какое угодно число пробелов, а за ними сколько угодно длинная последовательность букв.

Примечание: вообще, регулярные выражения - очень мощный и эффективный инструмент, который используется повсеместно, начиная с анализа данных и заканчивая веб-разработкой. Очень рекомендуется качественно изучить этот материал.

Таким образом, очевидно, как эту информацию и этот подход можно использовать при разработке чат-ботов. Если чат-боту приходит сообщение, которое удовлетворяет данному регулярному выражению (или паттерну), то значит это было приветствие и нужно поздороваться в ответ. Также, зная вид регулярного выражения, можно легко из него извлечь слово, которое, вероятно, является именем адресата сообщения. Так можно, например, определять, адресовано сообщение нашему боту или нет (если мы говорим про групповой чат).

Также для бота можно определить форматы вопросов, на которые он будет уметь отвечать и также эти вопросы можно искать с использованием регулярных выражений.

Заключение: В этом подразделе мы с вами обсудили несколько более старые подходы (вероятностный и с использованием регулярных выражений) к решению задач анализа текста. Как ни странно в современных системах эти подходы достаточно часто используются, конечно, не как основной способ решения задачи, но многие подзадачи эффективно решаются упомянутыми выше методами. А теперь давайте вернемся к современности.

3.4 Препроцессинг текста

Давайте теперь научимся переводить текст на язык понятный компьютеру — язык чисел.

Вообще, как мы с вами помним, компьютер умеет работать только с битами. И уже биты воспринимаются пользователями по-разному: как числа, как буквы или символы, как матрицы чисел(картинки). Строго говоря, текст для компьютера — это уже некоторый набор чисел. Но, к сожалению, такой перевод букв (токенов) в числа не очень хорош для нашей задачи. При обсуждении последующих тем, поймем почему такое численное

представление (или эмбеддинг) не очень подходит для нашей задачи.

Рассмотрим в качестве примера задачу классификации текста. Например, мы хотим научиться классифицировать текст по характеру: позитивный или нет. Для этого нам надо как-то набором чисел описать этот текст (или, как говорят, получить эмбеддинг этого текста). Задача получения эмбеддинга текста, вообще говоря, не простая. Попробуем сделать ее проще:

Давайте поделим текст на части поменьше, переведем эти части в какие-то наборы чисел. Тогда, очевидно, что описательный набор чисел для всего текста будет каким-то образом выражаться через описательные наборы чисел для частей поменьше. Т.е. если мы научимся как-то делить текст на части поменьше и считать эмбединги для этих небольших частей, то можно будет каким-то образом посчитать эмбеддинг для всего текста.

Замечание: самое сложное для любой задачи - научиться считать эмбеддинг для какого-либо объекта. Например, для картинки эмбеддинг можно посчитать при помощи сверточных нейронных сетей, что вы уже умеете. После подсчета эмбеддинга, содержащего всю информацию об объекте, можно легко использовать его для любых задач (например, эмбеддинг картинки можно использовать для генерации описания и тд).

А теперь мы вернемся к конкретному объекту нашего исследования — тексту.

3.4.1 Токенизация

Токенизация — процесс разбиения текста на части «поменьше».

Цель процесса токенизации — разделить текст на части поменьше. И эти части могут быть какие угодно:

1. слова. Этот способ достаточно очевидный, в качестве токенов выступают слова
2. подслова. Здесь для понимания рассмотрим пример: слово «теплоход» будет разбито на два токена «тепло» и «ход», т.е.

очевидно, что эмбединг сложного слова выражается через эмбединги подслов этого сложного слова

3. n-граммы n-грамма — это просто последовательность из n символов

Пример токенизации по словам:

```
>>> sentence = """Thomas Jefferson began
                  building Monticello at the age of 26."""
>>> sentence.split()
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26.']
```

Теперь, если мы знаем эмбеддинги каждого из полученных токенов, то можно легко получить эмбеддинг всего предложения, например, просто объединив эмбеддинги всех токенов. Тоже вполне рабочий вариант. Но, давайте заметим, пару моментов:

1. Например, слово the встречается очень часто и не несет в себе никакой смысловой нагрузки (такие слова называются стоп-слова)
2. Например, форма слова building также не несет в себе очень большой смысловой нагрузки, все равно процесс связан со строительством. Поэтому в качестве токена можно взять начальную форму слова: build. А затем использовать более простой и хорошо изученный эмбеддинг слова build. Процесс замены форма слова на изначальную называется «лемматизация» или «стемминг»

Таким образом, можно догадаться как упростить вычисление эмбеддинга предложения (и текста):

1. убрать стоп-слова в силу их «бессмысленности»
2. привести слова к изначальной форме, обрезав суффиксы, окончания, возможно, полностью изменив слово (better ->

good).

3.4.2 Стоп-слова

Что такое "стоп-слова"? Это слова, которые очень часто встречаются в языке и уже не несут какой-то сильной смысловой нагрузки. Посмотрим на стоп-слова в английском:

```
>>> stop_words = nltk.corpus.stopwords.words('english')
>>> len(stop_words)
153
>>> stop_words[:7]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours']
```

Также к уже отображенным словам добавляются союзы, вспомогательные глаголы и тп.

Подумайте: какие стоп-слова есть в русском языке?

3.4.3 Лемматизация (стемминг)

Лемматизация (стемминг) — процесс приведения токенов (чаще именно слов или подслов) к их изначальной словоформе.

Отличие лемматизации от стемминга лишь в подходе:

- стемминг просто пытается отсечь все лишнее в слове, чаще всего основываясь на правилах;

- лемматизация работает более гибко, часто проверяет связи со словарями

Например, стемминг не уловит для слова better связь со словом good и не произведет соответствующую замену (здесь необходимо сверяться со словарем); от лемматизации, в свою очередь, ожидается, что такая связь будет поймана.

Таким образом, лемматизация и стемминг обозначают один и тот же процесс, только лемматизация определяет более тонкий подход к реализации данного процесса.

3.4.4 Эмбедингы

Как мы с вами помним, эмбеддинг — некоторое численное (возможно, в виде набора чисел) представление некоторого объ-

екта, которое содержит некоторую информацию об этом объекте. Давайте подумаем, какими числами можно описать текст? Прежде чем читать дальше, попробуйте предложить свои варианты наборов чисел, которые как-то смогут описывать некоторый текст. Например, эмбеддингом человека может служить тройка чисел: возраст, рост и вес человека. Очевидно, что это плохой эмбеддинг, так как несколько разных людей могут иметь одинаковую такую тройку чисел.

Предложите свои варианты различных эмбеддингов текстов (их очень много).

Bag of words

Один из первых вариантов подсчета эмбеддинга для текста — bag of words. Данный метод подсчета эмбеддинга ставит тексту в соответствие вектор, в котором напрямую записана информация о том, сколько и каких слов встретилось в тексте.

Первым делом сформируем некоторый список слов (словарь). Возьмем большой набор текстов (корпус текстов), удалим из него стоп-слова и проведем лемматизацию. Затем для каждого слова посчитаем сколько раз это слово встретилось в корпусе, какое-то количество наиболее часто встретившихся слов будем называть «известными» словами. Каким-то образом упорядочим список из этих слов

Следовательно, эмбеддинг, полученный данным методом будет представлять собой вектор, где в индексе i записано количество вхождений i-го слова (из сформированного списка) в конкретный текст.

Для лучшего понимания рассмотрим пример:

Корпус текстов:

1. Привет, как дела? Как твоя жизнь?
2. Приветики, как идут твои дела? Вася открыл свое дело.
3. Приветули, как жизнь?

После фильтрации лишних символов, удаления стоп-слов и проведенной лемматизации текст примет следующий вид:

```
["привет", "как", "дело", "как", "жизнь"]
["привет", "как", "идти", "дела", "Вася", "открыть", "дело"]
["привет", "как", "жизнь"]
```

Очевидно, что в корпусе значительно чаще остальных встречаются слова:

```
["привет", "как", "дело", "жизнь"]
```

Давайте упорядочим эти слова по алфавиту (но это вовсе не обязательно)

```
["дело", "жизнь", "как", "привет"]
```

Получим следующие эмбеддинги на текстах из нашего корпуса (и не только):

Привет, как дела? Как твоя жизнь? - [1, 1, 2, 1]
Приветики, как идут твои дела? Вася, говорят, открыл своё дело. - [2, 0, 1, 1]
Приветули, как жизнь? - [0, 1, 1, 1]
Где ты? - [0, 0, 0, 0]
Что случилось? - [0, 0, 0, 0]

Из приведенных примеров понятно, что одним и тем же эмбедингом могут обладать совсем разные предложения (это значит, что будет невозможно отличить эти предложения по эмбеддингу). Даже для очень большого словаря можно привести такие примеры.

Подумайте, какие еще недостатки у этого метода? Какой важный момент свойственный в особенности русскому языку не учитывается в данной модели?

Ответ: не учитывается информация о порядке слов и контексте. Это важно, ведь, как мы с вами знаем, порядок слов может в корне менять дело.

Например: предложения «Учитель проверял не наши работы» и «Учитель не проверял наши работы» ощутимо отличаются по

смыслу. Это понимали и люди, которые занимались разработкой систем анализа естественного языка, и пытались придумать более подходящие способы подсчета эмбеддингов.

Tf-idf и аналоги

Как мы уже обсудили в методе bag of words исследователей не устраивало отсутствие информации о контексте и порядке слов. Это было наиболее важной, но на тот момент не решаемой проблемой.

Тогда для начала люди просто решили сделать более понятные с точки зрения статистики, более информативные признаки. Понятно, что само по себе количество слов в документе (в тексте) мало что может сказать об этом тексте. Но вот информация о том, в какой доле текстов из всего корпуса встречалось это слово - звучит уже более содержательно.

Из подобных статистических и логических соображений были предложены Tf-idf и подобные ему признаки, которые будут рассмотрены чуть ниже.

Так вот, tf-idf(word, text) - некоторое число, которое содержит в себе информацию о значимости слова word в тексте text, опираясь на то, как часто это слово появляется в данном тексте и во всем корпусе.

$$TF{-}IDF(word, text, corpus) = TF(word, text) \cdot IDF(word, corpus)$$

где

$$TF(word, text) = \frac{\text{количество слов word в text}}{\text{количество слов в text}}$$

$$IDF(word, corpus) = \log \frac{\text{количество текстов в corpus}}{\text{количество текстов в corpus, содержащих word}}$$

Таким образом, чем больше раз слово встречается в документе, тем выше будет значение TF (и, следовательно, TF-IDF). В то же

время по мере увеличения количества документов, содержащих слово, его IDF (и, следовательно, TF-IDF) будет уменьшаться. Итак, теперь у вас есть число — то, с чем компьютер может работать. Но что оно означает? Оно связывает определенное слово с конкретным документом в определенном корпусе, а затем задает числовое значение важности этого слова в данном документе с учетом его встречаемости во всем корпусе.

Получаем ответ на вопрос: как найти эмбеддинг слова в тексте - посчитать TF-IDF для этого слова в этом тексте.

Но пока все ещё не понятно: как посчитать эмбеддинг для всего текста?

Очевидно, хочется, чтобы длины эмбеддингов (длины числовых векторов, характеризующих текст) для разных текстов совпадали. Для этого, например, можно сначала выяснить, сколько в корпусе уникальных значимых токенов (слов). Пронумеровать их и запомнить. А затем для каждого из этих токенов и для каждого документа подсчитать TF-IDF. Так каждый документ (текст) будет охарактеризован вектором размера N, где N — это число уникальных значимых токенов в корпусе. Также i-й элемент вектора для j-го документа будет равен TF-IDF(i-е слово, j-й документ).

word2vec

Мы рассмотрели несколько способов подсчета эмбеддинга слова в тексте, но это все не то, что нам нужно. Такие эмбеддинги не хранят глубокую информацию о значении этого слова, а значит такие эмбеддинги слов достаточно сложно переиспользовать в других задачах, и они будут не очень информативными.

А нам как раз хочется найти для каждого слова такой эмбеддинг (вектор чисел), который бы раскрывал весь глубокий смысл этого слова, как-то согласовывался с интуитивными представлениями об этом слове и логикой. Такое удалось сделать! Но как?

Чтобы изучить различные смыслы слова, взаимосвязи этого

слова с другим, можно пойти следующим путем: научим модель по паре слов предсказывать следующее слово. Задача сложная, но при решении этой задачи модель получит эмбеддинги поступающих слов так, чтобы в них хранилась полезная, реальная информация, потому что иначе не удастся правильно решать поставленную задачу.

Для такой задачи можно найти очень много данных — это абсолютно любые тексты из интернета. Можно обучить модель решать такую задачу и получить необходимые эмбеддинги.

Также не обязательно предсказывать именно следующее слово после пары слов, можно предсказывать слово посередине, зная два слова слева и два слова справа. Или можно просто по предыдущему предсказывать следующее слово. При решении каждой из этих задач изучается смысл слова, его связь с соседними словами, и из этой информации о соседстве слов извлекается информация о глубоком содержании конкретного слова.

Можно инвертировать задачу — по слову посередине, предсказывать окрестность этого слова. При решении каждой из упомянутых задач можно качественно выучить эмбеддинги слов из корпуса.

Также можно решать более простую задачу — научить модель отвечать на вопрос «Может ли пара слов находиться рядом?». Единственное, нужно будет аккуратно набрать данные, не забыв про примеры пар слов, для которых невозможно соседство.

Заметим, что при решении этих задач исходные эмбедингы для слов можно использовать одни и те же. Что имеется в виду? Если удалось найти хорошее решение последней задачи, из которого были получены эмбеддинги слов, то эти вектора могут быть также хорошо использованы для других упомянутых задач.

Если реализовать для одной из задач процесс обучения достаточно аккуратно, то у вас получится матрица эмбедингов слов, которая согласуется с действительностью.

Многие из вас слышали выражение: «Король - Мужчина + Женщина = Королева». Так вот, это не просто слова. Такое соотношение выполняется с хорошей точностью на качественно

и глубоко выученных эмбеддингах, где под разностью и суммой подразумеваются простые векторные операции.

Ниже приведем наглядную демонстрацию того, что было сказано выше. Цветом обозначаются значения вектора эмбеддинга для соответствующего слова.

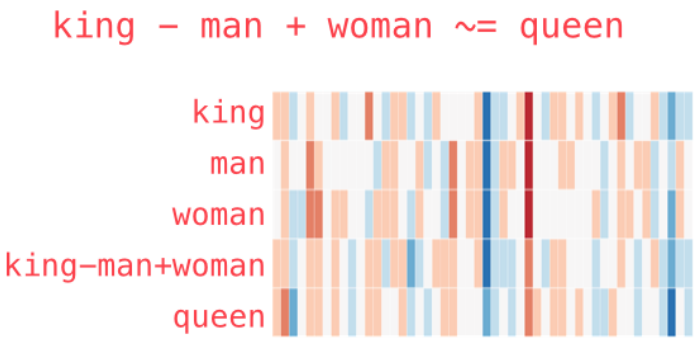


Рис. 3.1: Эмбединги word2vec

Посмотрите на вектор queen и на вектор king — man + woman. Они очень похожи! Что это значит? Это значит, что смыслы слов, глубокие взаимосвязи слов могут быть отображены достаточно хорошо в многомерное векторное пространство (т.е. слово может быть заменено вектором чисел с сохранением информации о значении, роде, числе, истории этого объекта и тд). Что со словами можно оперировать как с обычными числовыми векторами. Какой простор возможностей это нам предоставляет!

3.5 Архитектуры для работы с последовательностями

До данного момента мы с вами рассматривали только сети прямого распространения, т.е. сети, в которых информация идет от начала до конца, строго по «прямой». Также в уже изученных нами архитектурах каждый прогон сети был независим. И это

логично: мы решали задачу классификации изображений, в такой задаче все объекты, очевидно, независимы и уже изученный подход нас устраивает.

Давайте рассмотрим следующую задачу: требуется для каждого слова в предложении предсказать каким членом предложения оно является.

Рассмотрим предложение: «Мама мыла раму...». Для каждого слова нам известен какой-то эмбеддинг, например, методом word2vec можно найти обособленный эмбеддинг для каждого слова.

Тогда, используя найденные эмбеддинги, давайте попробуем решить поставленную задачу с использованием полносвязной нейронной сети. Посмотрим на картинку:

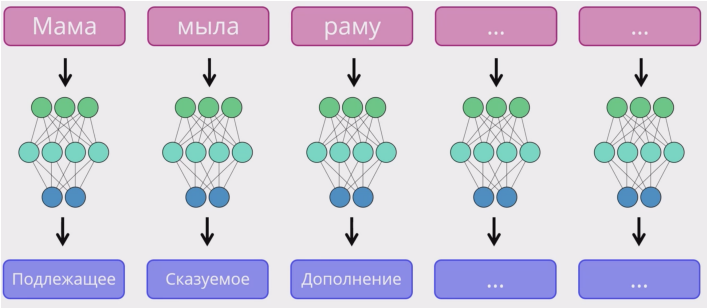


Рис. 3.2: Решение сетью прямого распространения

Что мы будем делать? Берем некоторую полносвязную сеть, прогоняем через неё эмбединг слова «мама», получаем предсказанный член предложения, аналогично с использованием той же сети поступаем со словом «мыла» и со словом «раму». Но этот подход плохой. Подумайте, в чём его проблема?

На самом деле ответ прост: данный метод никак не учитывает контекст слова, т.е. его ближайшее окружение, хотя бы окружение слева справа. А это очевидная проблема, ведь на то, каким членом

предложения является слово, очень сильно влияет контекст этого слова.

Именно из этих соображений приходим к необходимости разработки принципиально новой архитектуры сети, обладающей памятью о своих прошлых итерациях.

3.5.1 RNN (Recurent Neural Network)

Будем решать проблему обособленности прогона нейронной сети на каждом из слов. Как? Давайте при предсказании каким членом предложения является слово будем использовать информацию хотя бы о словах, которые шли в предложении до этого слова. Это значит, что на последующих итерациях на вход нейронной сети будем подавать не только эмбеддинг текущего слова, но и выход скрытого слоя с предыдущей итерации (т.е. с прогона для предыдущего слова). Как на картинке ниже

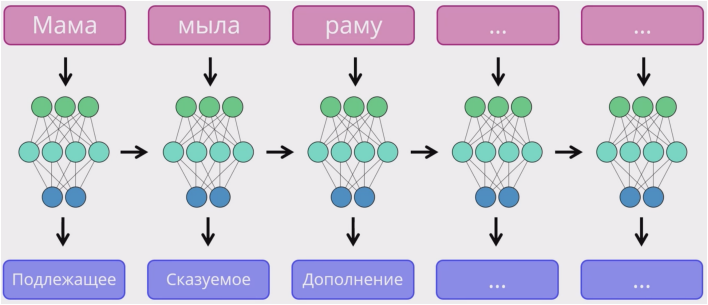


Рис. 3.3: Демонстрация простой RNN

Таким образом, при определении каким членом предложения является конкретное слово будет использована информация о том, какие слова шли в предложении перед этим словом. Конечно, это все еще не очень хорошее решение конкретно для этой задачи (подумайте почему?). Но уже существенно более сильное решение, чем сети прямого распространения.

При работе с произвольными последовательностями этот метод также очень хорошо себя показывает.

Только что мы с вами получили простую рекуррентную нейронную сеть (точнее рекуррентный слой). Для лучшего понимания покажем его в следующем виде:

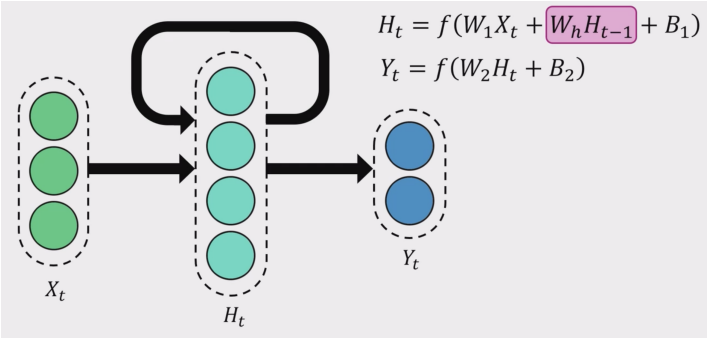


Рис. 3.4: Структура простой рекурентной ячейки

Здесь изображена абсолютно эквивалентная предыдущей нейронная сеть. Чаще в современных материалах и статьях можно увидеть следующее изображение. И схема, изображенная ниже, также эквивалентна двум предыдущим. Присмотритесь и это станет более очевидно)

Маленький светло-зеленый прямоугольник обозначает полносвязный слой с соответствующей функцией активации, в данном случае гиперболическим тангенсом (tanh).

Скажем проще, светло-зеленый прямоугольник на схеме обозначает умножение на некоторую матрицу своего входа (или каждого из своих входов) плюс некоторый вектор сдвига (вектор b) и к результату поэлементно будет применена функция активации, в частности, гиперболический тангенс.

Особо внимательные читатели, вероятно, заметили, что на

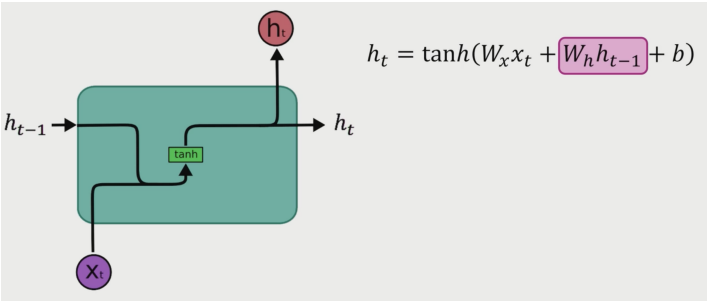


Рис. 3.5: Графовое представление рекурентной ячейки

схеме есть буква t. Она обозначает «момент времени». При работе с последовательностями нам приходится ввести момент времени. Например, с текстами это может быть номер итерации на токене, т.е. при работе с предложением «Мама мыла раму», при работе со словом «мама» можно считать, что $t = 1$, со словом «мыла» - $t = 2$ и тд. Можно также начинать нумерацию с нуля — это абсолютно не важно.

Для лучшего понимания рассмотрим реализацию RNN слоя (для одной последовательности) с использованием полносвязных слоев:

```
import torch
import numpy as np
from torch import nn
import torch.nn.functional as F

class RnnCell(nn.Module):
    def __init__(self, input_size, h_size):
        super().__init__()
        self.h_size = h_size
        self.fcXH = \
            nn.Linear(input_size, self.h_size) # W_x and bias
        self.fcHH = \
            nn.Linear(self.h_size, self.h_size, bias=False) # W_h

    def forward(self, x):
        length, emb_size = x.shape
        h = torch.zeros((self.h_size))
        h_all = [] # список всех получившихся векторов h

        # Цикл по входной последовательности
        for i in range(length):
            h = F.relu(self.fcXH(x[i]) + self.fcHH(h))
            h_all.append(h)

        return h_all
```

Применение линейного слоя в torch к столбцу X, как мы помним, равносильно следующей операции:

$$Y = W \cdot X + b$$

где b - bias (смещение). В линейном слое bias присутствует при условии bias=True (значение по-умолчанию).

Таким образом, в данной части мы с вами рассмотрели простейший рекурентный слой, принципиально отличающийся от

всего вышеизученного, потому что этот слой обладает внутренней памятью.

Но и у этой архитектуры есть некоторые проблемы. Давайте рассмотрим более мощные вариации рекуррентного блока.

3.5.2 LSTM (Long Short-Term Memory)

Предположим, нам нужно создать следующий проект: нужно написать чат-бота, который будет отвечать на вопросы пользователя о товарах. И заказчик просит сделать чат-бота достаточно интеллектуальным для ответа на сложные вопросы. Например:

Пользователь: Привет. Расскажи-ка мне все про смартфон модели <модель телефона>.

Бот: Это смартфон, который хорошо подходит для выполнения повседневных задач, также...

Или вот ещё пример:

Пользователь: Соедини меня с оператором. Хотя нет, лучше сам расскажи мне про эту модель часов.

Бот: <рассказ про конкретную модель часов>

Казалось бы, мы знаем про RNN слой почти все, что нужно, поэтому давайте просто вытащим информацию из всего запроса пользователя этим блоком и на основе этой информации сгенерируем (пока не важно каким образом) ответ бота.

Но тут есть проблема: RNN слой не умеет забывать. Он все помнит, т.е. для запросов по типу первого он отработает хорошо, а для запросов в стиле второго уже могут возникнуть трудности, потому что у текста поменялась мысль. Если сначала пользователь хотел одно, но потом захотел другое, то бот должен уметь в нужный момент забыть то, что было сказано и переключиться на другую мысль.

Так вот, уже изученный нами слой умеет только запоминать, но, к сожалению, не умеет забывать. Это проблема.

Для решения этой проблемы существует ячейка LSTM (Long Short-Term Memory). Данная ячейка была придумана эвристически, т.е. именно на логических соображениях. Рассмотрим этот «умный» слой.

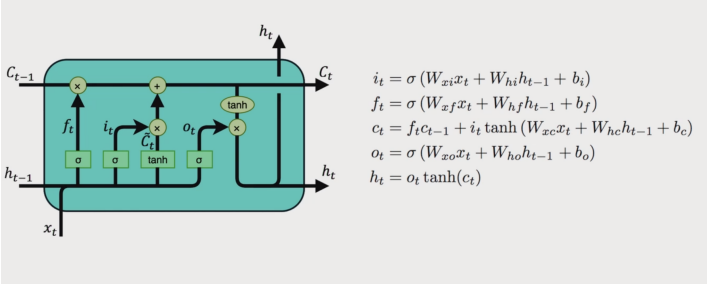


Рис. 3.6: LSTM ячейка

Теперь разберем принцип и логику построения этой ячейки более подробно.

Поймем, что изображено на схеме:

f_t - forget gate

i_t - input gate

o_t - output gate

На вход ячейке подаётся текущее состояние (x_t), предыдущее скрытое состояние (h_{t-1}) и вектор памяти (c_{t-1}).

Для начала нужно вычислить (c_t) - вектор памяти на текущей итерации, т.е. получить ответ на вопрос, что нужно запомнить после текущей итерации. Понятно, что нужно запомнить какую-то информацию с прошлых итераций (т.е. нужно будет учитывать c_{t-1} при вычислении c_t) и информацию с текущей итерации также нужно будет учесть при вычислении c_t).

Так c_t будет являться суммой профильтрованной части с прошлых итераций (преобразованное c_{t-1}) и несколько преобра-

зованной информации с текущей итерации. Таким образом,

$$c_t = filtered\_old\_information + current\_information$$

Теперь придумаем, как фильтровать старую информацию и как преобразовывать текущую информацию, чтобы получить c_t.

Введем коэффициенты забывания — это как раз f_t. f_t вычисляется по формуле с картинки на основе текущего состояния и предыдущего скрытого состояния. Таким образом

$$filtered\_old\_information = f_t \cdot c_{t-1}$$

где подразумевается поэлементное произведение

Аналогичным образом и из аналогичной логики получим второе слагаемое для c_t: $current\_information$

$$current\_information = i_t \cdot \widetilde{c}_t$$

где i_t — также полносвязный слой с сигмоидальной функцией активации от входа ячейки (т.е. несколько преобразованный вход ячейки), а $\widetilde{c}_t$ - полносвязный слой с гиперболическим тангенсом в роли функции активации (коэффициенты фильтрации текущей информации). Таким образом мы научились вычислять вектор памяти, с информацией о том, что нужно будет забывать или помнить впоследствии. Но нам нужно также вычислить скрытое текущее состояние, т.е. актуальную информацию о текущем и прошлых состояниях.

Она будет вычислена с использованием c_t (что логично) и с использованием output gate, т.е. опять же некоторым образом преобразованного входа ячейки. Последние две формулы на картинке точно описывают данный способ получения h_t.

Перечитайте эту информацию еще раз, с первого раза может быть сложно ее осознать. Но если подумать и вникнуть, то можно вполне четко понять логику построения этой ячейки.

Для лучшего понимания алгоритма приведем реализацию ячейки LSTM с использованием линейных слоев, аналогично RNN ячейке:

```
class LSTMCell(nn.Module):
    def __init__(self, x_size, h_size):
        super().__init__()
        self.h_size = h_size

        # input gate
        self.fcXI = nn.Linear(x_size, self.h_size)
        self.fcHI = nn.Linear(self.h_size, self.h_size, bias=False)

        # forget gate
        self.fcXF = nn.Linear(x_size, self.h_size)
        self.fcHF = nn.Linear(self.h_size, self.h_size, bias=False)

        # создание нового кандидата c_t_tilde
        self.fcXC = nn.Linear(x_size, self.h_size)
        self.fcHC = nn.Linear(self.h_size, self.h_size, bias=False)

        # output gate
        self.fcXO = nn.Linear(x_size, self.h_size)
        self.fcHO = nn.Linear(self.h_size, self.h_size, bias=False)

    def forward(self, x):
        length, emb_size = x.shape
        h_t = torch.zeros((self.h_size))
        c_t = torch.zeros((self.h_size))
        h_all = [] # список всех получившихся векторов h
        c_all = []

        # Цикл по входной последовательности
        for i in range(length):
```

```
31          i_t = F.sigmoid(self.fcXI(x) + self.fcHI(h_t))
32          f_t = F.sigmoid(self.fcXF(x) + self.fcHF(h_t))
33          o_t = F.sigmoid(self.fcXO(x) + self.fcHO(h_t))
34          c_t_tilde = F.tanh(self.fcXC(x) + self.fcHC(h_t))
35          c_t = \
36            f_t * c_t + i_t * c_t_tilde # поэлементные операции
37          h_t = o_t * F.tanh(c_t) # поэлементное произведение
38
39          h_all.append(h_t)
40          c_all.append(c_t)
41        return h_t, c_t
```

Ячейки такого типа очень часто используются в задачах тем или иным образом связанных с анализом последовательностей.

3.5.3 GRU (Gated Recurent Unit)

Давайте рассмотрим очередную вариацию RNN ячейки с возможностью «забывать» ненужную информацию. В ячейке типа LSTM целых 4 полносвязных «слоя». Кажется, что это количество можно попробовать уменьшить, опираясь на следующее предположение: пусть в h_t у нас в каждый момент времени будет содержаться одно и то же условное количество информации. Поэтому h_t будем формировать следующим образом: долю z - возьмем из h_{t-1} и x_t, т.е. это можно считать полезной информацией, извлеченной на текущем шаге, а долю $(1-z)$ возьмем из информации с прошлых итераций, т.е. из h_{t-1}.

Из этих соображений можно построить ячейку следующего типа — типа *GRU*.

Вычисляем коэффициенты z_t и r_t. В данном случае коэффициент $(1-z_t)$ — условный коэффициент забывания. В свою очередь, r_t — коэффициент фильтрации информации, для отбора полезной информации из h_{t-1}. Оба коэффициента вычисляются как полносвязный слой с сигмоидой в качестве функции активации и входом $[h_{t-1}, x_t]$. Такой вход в слой, вычисляющий указан-

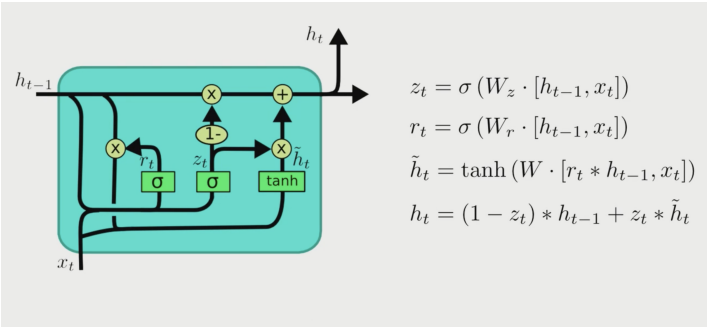


Рис. 3.7: GRU ячейка

ные коэффициенты, кажется достаточно ожидаемым. Подумайте, почему?

Более конкретные формулы для вычисления коэффициентов и выходов приведены на изображении.

Для большей понятности данного слоя приведем реализацию в стиле аналогичном двум предыдущим рекуррентным ячейкам.

```
import torch
import numpy as np
from torch import nn
import torch.nn.functional as F

class GRUCell(nn.Module):
    def __init__(self, x_size, h_size):
        super().__init__()
        self.h_size = h_size

        # input gate
        self.fcZ = \
            nn.Linear(x_size + self.h_size, self.h_size, bias=False)
        self.fcR = \
```

```
            nn.Linear(x_size + self.h_size, self.h_size, bias=False)
        self.fcH = \
            nn.Linear(x_size + self.h_size, self.h_size, bias=False)

    def forward(self, x):
        length, emb_size = x.shape
        h_t = torch.zeros((self.h_size))
        c_t = torch.zeros((self.h_size))
        h_all = [] # список всех получившихся векторов h
        c_all = []

        # Цикл по входной последовательности
        for i in range(length):
            h_t_1_x_t = torch.cat((h_t, x[i]), 0)
            z_t = F.sigmoid(self.fcZ(h_t_1_x_t))
            r_t = F.sigmoid(self.fcR(h_t_1_x_t))

            r_t_h_t_1_x_t = torch.cat((r_t * h_t, x[i]), 0)
            h_t_tilde = F.tanh(self.fcH(r_t_h_t_1_x_t))

            h_t = (1 - z_t) * h_t + z_t * h_t_tilde
            h_all.append(h_t)

        return h_all
```

После внимательного прочтения кода и теории с опорой на прошлый опыт, вероятно, для вас станут более прозрачны принципы работы рекуррентных ячеек, рассмотренных выше. Данные ячейки используются при работе с любыми временными рядами (последовательностями), которыми является текст, цены акций, звук и тд.

Также, давайте вспомним, что как у ячейки GRU, так и у ячейки LSTM есть блок памяти, в отличие от RNN, который позволяет их достаточно успешно использовать для задач с длин-

ными последовательностями. Но также GRU более эффективна в плане затрат ресурсов (время на обучении, инференс), поэтому часто лучше себя показывает на задачах с длинными предложениями (текстами) блок GRU. А блок RNN вообще почти не используется на практике в силу недостатков, которые были упомянуты выше.

3.6 Типы задач работы с последовательностями

Теперь давайте рассмотрим различные типы задач, которые часто стоят перед нами при решении задач, связанных с обработкой с последовательностей.

3.6.1 Many-to-one

Дословный перевод заголовка этого блока — «много в один». Какой вариант задачи, соответствующей так или иначе данному заголовку, приходит вам на ум первым? Ассоциации с какими задачами у вас появляются?

Лично мне первой приходит на ум задача классификации текстов. Потому что есть текст — некоторая последовательность токенов (условное «много»). И по этой последовательности нужно дать только один ответ - класс текста. Это могут быть абсолютно разнообразные классы: позитивный комментарий или негативный; является письмо спамом или нет; текст написан в жанре повести, драмы или сказки. Также к Many-to-one относится задача определения вернет человек кредит или нет по его кредитной истории, которая также может быть рассмотрена как некоторая последовательность.

Замечание: в данном списке были упомянуты не только задачи бинарной классификации (где стоит выбор между двумя классами), но и задачи многоклассовой классификации.

В задаче подобного типа обычно поступают следующим образом: создают некоторый эмбеддинг для всего объекта (текста) и затем на нем решают задачу классификации (даже, например, просто несколькими полносвязными слоями). Но, разумеется,

встает вопрос: «А как получить эмбединг всего текста?». Здесь нам на помощь и придут наши рекуррентные нейронные сети и изученные нами ячейки LSTM и GRU. В скрытом состоянии ячейки хранится вся нужная информация об уже просмотренной части последовательности (текста). Поэтому давайте брать в качестве эмбединга всего текста скрытое состояние (h_t) в момент, когда вся последовательность уже просмотрена нашим рекуррентным блоком. Тогда, используя вектор h_t в качестве признаков, будем решать задачу классификации, например, полносвязной сетью на уже сформированных признаках (на h_t).

Пример задачи

Как мы с вами помним, алгоритм решения задач NLP следующий:

1. Отобрать корпус текстов, соответствующий задаче
2. Определить метрику качества
3. Провести препроцессинг
4. Определить архитектуру нейронной сети

Чаще всего при решении задач с олимпиад или с хакатонов у вас уже есть некоторый корпус текстов, поэтому этот этап рассматривать не будем. Тем более, что организация данного этапа очень сильно зависит от задачи и ваших возможностей.

Рассмотрим более подробно пример решения задачи бинарной классификации отзывов из интернета ("позитивный"и "негативный"). Будем решать данную задачу с использованием популярной библиотеки для анализа естественного языка — модуля nltk.

Первым делом нужно считать данные и провести препроцессинг.

Препроцессинг

В самом начале обычно переобозначают лейблы классов. Если они определены названием, то часто удобнее их пронумеровать, т.е. задать маппинг — назвать класс positive классом 0, класс negative — классом 1. Это первое, что обычно делают при начале

работы с задачей. После этого уже более глубоко погружаются в задачу.

Прежде всего в задаче NLP важно определиться с тем, что вы будете использовать в качестве токенов. Для многих задач решение использовать метод токенизации по словам будет работать неплохо, а дальше уже нужно смотреть по специфике задачи и вашим вычислительным возможностям. Например, часто токенизация по подсловам работает лучше, чем все остальное, но:

1. Требует дополнительного обучения под специфику задачи
2. Увеличивает число обучаемых параметров
3. Увеличивает время прогона сети (инференса)
4. Увеличивает затраты по памяти

Поэтому в качестве первого приближения часто используют токенизацию по словам, а дальше меняют с учетом возможностей и особенностей задачи.

Токенизацию по словам для конкретного текста можно провести следующим образом. Перед токенизацией оставим в тексте только цифры и буквы и приведем текст к нижнему регистру. Кажется, что удаление знаков препинания, лишних символов и приведение текста к нижнему регистру не несет в себе почти никакой принципиальной информации для определения тона текста.

```
import re
import nltk
nltk.download('punkt') # Скачиваем пакет

from nltk.tokenize import word_tokenize

# data cleaning
text =  re.sub('[^А-Яа-яЁё0-9]+', ' ', text)

# lowercase
```

```
text = text.lower()

# tokenization
tokens = nltk.word_tokenize(text) # текст в токены

```

Опять же все эти предположения, весь приведенный пайплайн представлен в универсальном виде, который для большинства задач будет работать достаточно неплохо для базового решения. Но вполне возможно, что для какого-то случая сохранение регистра позволит получить более качественные решения задач. Тут уже нужно смотреть конкретно по задаче.

Проведя токенизацию текстов по словам обычно проводят чистку стоп-слов. Помните ли вы, зачем нужно удалять стоп-слова из текста? На данном этапе очень важно не удалить важной информации. Вы спросите, а как это возможно, удаляя стоп-слова, удалить важную информацию?

Неожиданно, но, например, слово «не» в русском языке (в частности, в модуле nltk) относят к стоп-словам. А это слово часто существенно влияет на отрицательность или позитивность отзыва, да и вообще на смысл предложения.

Например отзывы: «Этот фильм не очень хороший» и «Этот фильм очень хороший». Очевидно, что смыслы отзывов абсолютно противоположны, но отличаются лишь наличием или отсутствием слова «не». Следовательно на этапе удаления стоп-слов для данной задачи важно не удалять слова «не» из текста.

Чистка стоп-слов на том же тексте может быть проведена следующим образом:

```
nltk.download('stopwords')

from nltk.corpus import stopwords

stop_words = stopwords.words('russian') # defining stop_words
```

```
stop_words.remove('не')
text = [word for word in tokens if word not in stop_words]
```

После этого нужно провести лемматизацию (или стемминг, зависит от задачи) каждого получившегося токена. Лемматизация в разных библиотеках по анализу естественного языка работает практически одинаково качественно, поэтому как-то улучшать этап лемматизации стоит в последнюю очередь. А с использованием модуля nltk лемматизация может быть проведена следующим образом

```
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
lemms_tokens = [lemmatizer.lemmatize(word) for word in text]
```

Обернем операцию препроцессинга одного текста в функцию

```
def data_preprocessing(text):
  text =  re.sub('[^А-Яа-яЁё0-9]+', ' ', text)
  text = text.lower()
  tokens = nltk.word_tokenize(text) # converts text to tokens

  text = [word for word in tokens if word not in stop_words]
  lemms_tokens = [lemmatizer.lemmatize(word) for word in text]
  text = ' '.join(lemms_tokens)

  return text, lemms_tokens
```

С препроцессингом разобрались.

Эмбеддинг

Давайте разберемся с подсчетом эмбеддинга токенов и текста. Здесь есть несколько подходов.

Первый и самый простой вариант: подсчитать встречаемость каждого токена в корпусе (технически это можно сделать через Counter из модуля collections), выбрать какое-то количество наиболее часто встречаемых токенов (также на усмотрение исследователя, например, первые 10000), каждому токену поставить в соответствие номер. Задать токен <UNKNOWN>, чтобы уметь работать с неизвестными токенами. Затем перекодировать тексты (последовательность токенов) в последовательность номеров токенов.

Например:
Словарь: [«UNKNOWN> "привет "пока "кот"]
Текст: "Привет! Как твой кот?"
После препроцессинга: ["Привет "как "ты "кот"]
После кодировки индексами в словаре: [1, 0, 0, 3]

Слова «как» и «ты» закодированы нулем, потому что в сло-

варе их нет, поэтому им в соответствие был поставлен токен <UNKNOWN>. И уже полученный в конце вектор будет подаваться на вход нейронной сети, которая должна будет начинаться со слоя torch.nn.Embedding. Этот слой по индексу будет возвращать выученный эмбеддинг этого токена.

Также вместо этого слоя можно взять несколько первых слоев известных предобученных архитектур, например Bert, tiny-bert. Данные сети уже умеют вытаскивать эмбеддинги слов достаточно качественно, с учетом контекста, и останется только дообучить сеть под специфику задачи. Но это также необязательно. Можно даже взять эти слои и «заморозить», т.е. не обучать веса этих слоев. Такой подход со взятие предобученных слоев называется «файнтюнинг» и часто применяется на больших и сложных задачах.

Приведем кусочек кода, который позволяет легко построить словарь заданного размера, который автоматически задает кодировку токена в число (в индекс в этом листе):

```
UNKNOWN_IND = 0 # ind token "<UNKNOWN>""
PAD_IND = 1 # ind token "<PAD>"
DICT_SIZE = 10000

def create_dict(train_data, dict_size=DICT_SIZE):
  freq_counter = Counter()
  for text in tqdm(train_data['text']):
      splitted_text = data_preprocessing(text)[0].split()
      freq_counter.update(dict(Counter(splitted_text)))

  words_list = (['<UNKNOWN>', '<PAD>'] + \
    [word for word, val in freq_counter.most_common(dict_size)])

  return words_list

words_list = create_dict()
```

Часто бывает удобно написать также функции кодирования и декодирования текста из представления в виде индексов в токены и наоборот.

```
def my_ind(token, values):
  try:
    return values.index(token)
  except:
    return UNKNOWN_IND

def encode_text(text, words_list):
  lemms_tokens = data_preprocessing(text)[1]
  encoded_text = [my_ind(token, words_list)
                  for token in lemms_tokens]

  return encoded_text

def decode_text(encoded_inds, words_list):
  return ' '.join([words_list[ind] for ind in encoded_inds])


text_example = train_data['text'][0]
encoded_text = encode_text(text_example, words_list)
text_example, decode_text(encoded_text, words_list)
```

Второй, также часто встречаемый вариант: подсчет эмбеддинга каким-то алгоритмом до обучения сети, в качестве дополнительного этапа препроцессинга. В качестве этого алгоритма могут быть взяты, например, Bag-of-words, Tf-Idf, word2vec или какие-то другие, известные вам. В этом случае уже начинать нейронную сеть со слоя torch.nn.Embedding или предобученных слоев вовсе не нужно.

В рассматриваемой задаче будет использован первый подход со слоем nn.Embedding как наиболее простой. При желании на тему использования других подходов легко можно найти материалы

в интернете.

Батчинг текстов

Прежде чем рассматривать процесс создания и обучения модели, обсудим следующий момент.

Как мы с вами помним из общей теории нейронных сетей, параметры сети подбираются методом градиентного спуска с методом обратного распространения ошибки. Также ранее обсуждалось, что лучше всего делать шаг оптимизации методом градиентного спуска для некоторого подмножества данных (батча) из тренировочного датасета.

В задаче классификации текстов часто обучение также организуется по батчам, но есть один нюанс. Мы помним, что в слои из torch.nn можно передавать сразу батч объектов и считать выход слоя сразу для всего батча. С рекуррентными блоками (RNN, LSTM, GRU) можно поступать аналогично, передавать батч последовательностей и рассчитывать их параллельно. Но это возможно только при условии, что последовательности в батче имеют одинаковую длину.

Данное условие сложно в плане выполнения, так как мы работаем с текстами, длина которых может быть абсолютно разной. В связи с этим часто вводят токен <PAD>, который позволяет производить паддинг последовательностей в батче, т.е. выравнивать все последовательности в батче до одной длины путем добавления этого токена в конец.

Например, на таком батче (тексты после препроцессинга, но до вычисления эмбеддинга) выравниваем последовательности до длины в 4 слова(токена):

привет как ты дела

привет как ты <PAD>

пока до встреча <PAD>

Тогда такой батч уже можно передать в рекуррентный блок, что позволит существенно ускорить процесс обучения сети. Токен «PAD>"использован только для выравнивания последовательностей, и нейронная сеть в ходе обучения усвоит эту информацию.

В принципе, подход с выравниванием всех последовательностей в батче до одной длины вполне может быть использован. Но так не делают на практике. Почему?

Во-первых, если в батч попадут тексты очень разной длины (например, 5 и 100), то в батче будет последовательность, в которой слишком много токенов паддинга. Это явно не скажется хорошо как минимум на производительности.

Во-вторых, есть более тонкая проблема. Если при инференсе рекуррентный блок много раз проходит через токен <PAD>, то при обратном распространении ошибки (при вычислении градиента) полезная информация не будет обработана должным образом.

Для борьбы с двумя упомянутыми проблемами обычно проводят фильтрацию тренировочных данных: строят распределение длин текстов и на усмотрение исследователя выбирается пороговая длина. Таким образом, в трейне будут оставлены только тексты с длиной меньше пороговой. Потому что с длинными текстами (если их мало в трейне) работать нейронная сеть не сможет. Также с учетом второй проблемы наличие слишком длинных текстов усложняет процесс обучения как с точки зрения качества обучения, так и с точки зрения производительности.

На графике ниже с распределением длин текстов в тренировочном датасете наблюдается достаточно классической распределение длин текстов (коротких — много, длинных — мало). И видно, что тексты длиной 800-1000 составляют больше 90% корпуса. Исходя из распределения, для данной задачи можно спокойно выбирать тексты с длиной менее 800 токенов. Это решение также существенно зависит от задачи.

Но такая фильтрация поможет лишь немного решить проблему, ведь все равно в батч могут попасть последовательности сильно разной длины.

Авторы библиотеки torch знали о рассмотренных нами проблемах и создали функцию, которая позволяет оптимизировать работу с текстами не очень разной длины с решением второй проблемы (без лишних прогонов сети через токен паддинга). Это

функция pack_padded_sequence. Эта функция «плотно упаковывает» последовательности для более оптимального использования рекуррентных слоев. Вдаваться в подробности реализации этой функции мы не будем, но далее рассмотрим пример использования этой функции в коде.

Структура сети и обучение. Dataset, DataLoader, NN

Ранее вы уже знакомились с классами Dataset и Dataloader, поэтому знаете зачем они нужны. В задачах NLP часто бывает удобно написать свою реализацию этих классов, примеры реализаций для задачи классификации текстов мы сейчас и приведем.

В нашем случае будем работать следующим образом, мы отфильтровали тексты по длине и будем выравнивать их все до самого длинного в обучающей выборке (т.е. до max_seq_len), добавляя в конец предложения <PAD>. Как мы с вами помним, это не очень хорошая идея в силу ряда проблем, обсужденных выше. Их можно избежать явно, написав свою реализацию класса DataLoader, но это сложно. Поэтому мы просто реализуем нейронную сеть (с использованием функции pack_padded_sequence)

так, чтобы избежать этих проблем и не писать слишком много кода.

```
class CustomDataset(TensorDataset):
    def __init__(self, words_list, data, targets, max_seq_len):
        self.words_list = words_list
        self.texts = data['text'].tolist()
        self.text_lens = data['text_len'].tolist()
        self.targets = targets
        self.max_seq_len = max_seq_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        dd_text = np.array(
            encode_text(
                self.texts[idx],
                self.words_list
            )[:max_seq_len]
        )

        return [self.text_lens[idx],
                torch.nn.functional.pad(
                    torch.from_numpy(dd_text),
                    (0, self.max_seq_len - len(dd_text)),
                    mode='constant',
                    value=PAD_IND)],
                self.targets[idx]

```

Теперь создадим объекты классов CustomDataset и DataLoader

```
max_seq_len = 2539
train_dataset = CustomDataset(
    words_list,
    train_data,
    train_labels,
    max_seq_len=max_seq_len)
val_dataset = CustomDataset(
    words_list,
    val_data,
    val_labels,
    max_seq_len=max_seq_len)

train_dataloader = DataLoader(
    train_dataset, batch_size=50, shuffle=True
)
val_dataloader = DataLoader(
    val_dataset, batch_size=50, shuffle=True
)
```

В датасете будем возвращать для одного объекта его длину, выровненный до max_seq_len эмбеддинг и лейбл. Код, приведенный здесь, наверняка вам уже во многом знаком, подробно на нем останавливаться не будем.

Класс нейронной сети определим следующим образом:

```
class NN_net(nn.Module):
  def __init__(
        self,
        emb_size=16,
        num_layers=3,
        dict_size=DICT_SIZE,
        max_seq_len=max_seq_len):
    super().__init__()
    self.max_seq_len = max_seq_len
```

```
10      self.num_layers = num_layers
11      self.hidden_size = emb_size
12      self.dict_size = dict_size
13
14      self.emb_layer = nn.Embedding(self.dict_size, self.emb_size)
15      self.lstm = nn.LSTM(
16          input_size=self.emb_size,
17          hidden_size=self.hidden_size,
18          batch_first=True,
19          num_layers=self.num_layers,
20          dropout=0.4)
21      self.lin_layer_1 = nn.Linear(self.emb_size, 32)
22      self.lin_layer_2 = nn.Linear(32, 1)
23      self.sigmoid = nn.Sigmoid()
24
25    def forward(self, x, seq_length):
26      out = self.emb_layer(x)
27      # pack, remove pads
28      packed_input = pack_padded_sequence(
29          out,
30          seq_lengths,
31          batch_first=True,
32          enforce_sorted=False)
33
34      # lstm
35      packed_output, (ht, ct) = self.lstm(packed_input, None)
36
37      # unpack, recover padded sequence
38      out = ht[0,:,:]
39
40      out = self.lin_layer_1(out)
41      out = F.relu(out)
42      out = self.lin_layer_2(out)
43      out = self.sigmoid(out)
```

```
44
45      return out
```

Теперь приведем стандартный цикл обучения нейронной сети. В качестве лосс-функции возьмем бинарную кросс энтропию, т.к. решается задача бинароной классификации, в качестве оптимайзера возьмем вариацию градиентного спуска Adam:

```
1  def eval_model(model, data_loader, trashhold=0.5):
2      model.eval() # Set model to eval mode
3      true_preds, num_preds = 0., 0.
4
5      with torch.no_grad():
6          for data_inputs, data_labels in data_loader:
7
8              # Determine prediction of model on dev set
9              seq_lengths, data_inputs = data_inputs
10             data_inputs = data_inputs.to(device)
11             data_labels = data_labels.to(device)
12
13             preds = model(data_inputs, seq_lengths)
14             preds = preds.squeeze(dim=1)
15
16             pred_labels = (preds >= trashhold).long()
17
18             true_preds += (pred_labels == data_labels).sum()
19             num_preds += data_labels.shape[0]
20
21     acc = true_preds / num_preds
22     print(f"Accuracy of the model: {100.0*acc:4.2f}%")
23
24 lr = 0.005
25 n_epochs = 4
26 criterion = nn.BCELoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

for epoch in range(n_epochs):
  for i, inform in tqdm(enumerate(train_dataloader)):
    model.train()
    data, labels = inform
    seq_lengths, data = data

    data = data.to(device)
    labels = labels.to(device)

    model.zero_grad()
    preds = model(data, seq_lengths)
    loss = criterion(preds, labels.unsqueeze(dim=1).float())
    loss.backward()

    optimizer.step()
    if i % 100 == 0:
      eval_model(model, val_dataloader)
```

Таким образом можно обучить сеть для решения задачи бинарной классификации текстов.

Еще несколько вариантов задачи many-to-one: задача генерации изображения по описанию, задача обучения эмбединга текста, задача предсказания прогноза погоды на завтра на основе наблюдений в прошлые дни и многие другие, определение говорящего по голосу и многие другие. В ходе решения каждой из упомянутых задач на входе мы получаем последовательность (текст, звук), а на выходе получаем один объект (класс, эмбеддинг, изображение и тп).

3.6.2 One-to-many

Теперь вопрос чуть сложнее. А какой вариант задачи подойдет для этого пункта, который дословно переводится как «один во много»? Может у вас есть какие-то интуитивные соображения на этот счет?

Тут уже примеры менее очевидны, так как с такими задачами вы до сих пор не сталкивались. Например, задача генерации описания к картинке относится к типу one-to-many. Потому что на входе мы имеем один объект, картинку, а на выходе последовательность - текст.

Да и на олимпиадах и хакатонах такие задачи встречаются существенно реже, поэтому на данном типе мы останавливаться не будем.

3.6.3 Many-to-many или sequence2sequence

Приведем примеры задачи «много во много»: генерация краткого изложения текста, задача генерации ответа на сообщение пользователя также теоретически может быть решена с использованием такого подхода. Данный тип задачи принципиально отличается от предыдущих тем, что здесь по некоторой последовательности нужно сгенерировать другую последовательность.

Решим задачу машинного перевода с английского на русский. Используем англо-русский датасет с сайта с большим количеством данных для решения задач перевода в частности. Нет надобности для примера использовать все данные (их очень много), поэтому возьмем только первые 10000 небольших предложений и на них обучим нашу модель. В качестве токенов будем брать символы алфавита, английского и русского языков.

Скачаем данные и считаем их. В source_texts будут храниться английские тексты и соответветствующие им переводы на русский будут записаны в target_texts.

```
1 import torch
2 import codecs
```

```
import re
import numpy as np
import spacy
import random
from torch import nn
from torch.utils.data import TensorDataset, DataLoader

from tqdm.notebook import tqdm

device = torch.device(
    "cuda:0" if torch.cuda.is_available() else "cpu"
)

# read train data
data_fpath = 'rus.txt'
cnt_sentences = 10000
```

Теперь сохраним списки текстов для обучения из файла data path

```
source_texts = []
target_texts = []
lines = codecs.open(
    data_fpath,
    'r',
    encoding='utf8'
).readlines()[:cnt_sentences]
for line in lines:
    source_text, target_text, = line.split('\t')[:2]
    source_texts.append(source_text.lower())
    target_texts.append(target_text.lower())
```

Создадим словари символов соответственно для английского и русского языков. В нашем словаре в данном решении (вообще это не обязательно) будут присутствовать использованные в корпусе знаки препинания. Для удобства сделаем так, чтобы токен паддинга, начала предложения и конца предложения имели одни и те же индексы в словарях обоих языков, так как изначально эмбедингом токена будет являться его индекс в словаре, по которому вектор эмбединга токена будет извлечен из специального слоя.

```
PAD_TOKEN = '<PAD>'
def prepare_vocab(texts):
    result_text = ''.join(texts).lower().replace(PAD_TOKEN, '')
    vocab = [PAD_TOKEN, '<END>', '<START>'] + \
      sorted(set(result_text))

    vocab_size = len(vocab)
    char2idx = {u:i for i, u in enumerate(vocab)}
    idx2char = {i:u for i, u in enumerate(vocab)}
    return vocab_size, char2idx, idx2char

source_params = prepare_vocab(source_texts)
```

```
SOURCE_VOCAB_SIZE, source_char2idx, source_idx2char = \
        source_params

target_params = prepare_vocab(target_texts)
TARGET_VOCAB_SIZE, target_char2idx, target_idx2char = \
        target_params

# find max_seq_lengths
max_enc_seq_length = max([len(tt) for tt in source_texts])
max_dec_seq_length = max([len(tt) for tt in target_texts])
PAD_IND = source_char2idx[PAD_TOKEN]
```

Определим датасет для работы с данными и даталоадер. Создадим промежуточный датасет CustomDataset, который будет возвращать последовательность токенов для декодера/энкодера (в зависимости от параметров). Эта последовательность также будет выровнена до соответствующей фиксированной длины, которая будет зависеть от режима работы этого класса (encoder или decoder).

```
class CustomDataset:
  def __init__(self, texts, char2idx, type_dataset):
    self.texts = texts
    self.char2idx = char2idx
    self.type_dataset = type_dataset

  def __len__(self):
    return len(self.texts)

  def __getitem__(self, ind):
    if self.type_dataset == 'encoder':
        encoded_text = [self.char2idx[c] for c in self.texts[ind]]
        encoded_text = [
            self.char2idx['<START>'],
```

```
            *encoded_text,
            self.char2idx['<END>']]
        encoded_text = np.array(encoded_text)
        return torch.nn.functional.pad(
            torch.from_numpy(encoded_text),
            (0, max_enc_seq_length - len(encoded_text)),
            mode='constant', value=PAD_IND)
    elif self.type_dataset == 'decoder':
        target_text = [self.char2idx[c] for c in self.texts[ind].lower()]
        input_text_to_decoder = [self.char2idx['<START>']] + \
                                target_text
        input_text_to_decoder = np.array(
            input_text_to_decoder
        )
        input_text_to_decoder = torch.nn.functional.pad(
            torch.from_numpy(input_text_to_decoder),
            (0, max_dec_seq_length - len(input_text_to_decoder)),
            mode='constant', value=PAD_IND)
        target_text_for_decoder = target_text + \
                                  [self.char2idx['<END>']]
        target_text_for_decoder = np.array(
            target_text_for_decoder
        )
        target_text_for_decoder = torch.nn.functional.pad(
            torch.from_numpy(target_text_for_decoder),
            (0, max_dec_seq_length - len(target_text_for_decoder)),
            mode='constant', value=PAD_IND)
        return input_text_to_decoder, target_text_for_decoder
```

Класс EncoderDecoderDataset будет использован напрямую при обучении. Он будет возвращать для каждого объекта следующую информацию: входную последовательность в енкодер, входную последовательность в декодер, объединенные в данные и, в качестве "таргета ожидаемую из декодера последовательность.

```
class EncoderDecoderDataset(TensorDataset):
  def __init__(self, encoder_dataset, decoder_dataset):
    self.decoder_dataset = decoder_dataset
    self.encoder_dataset = encoder_dataset

  def __len__(self):
    return min(len(self.decoder_dataset),
                      len(self.encoder_dataset))

  def __getitem__(self, ind):
    inp_decoder_text, target_text = self.decoder_dataset[ind]
    return [self.encoder_dataset[ind], inp_decoder_text],
              target_text

encoder_dataset = CustomDataset(
    source_texts,
    source_char2idx,
    'encoder')
decoder_dataset = CustomDataset(
    target_texts,
    target_char2idx,
    'decoder')
enc_dec_dataset = \
    EncoderDecoderDataset(encoder_dataset, decoder_dataset)

train_dataloader = DataLoader(
    enc_dec_dataset,
    batch_size=100,
    shuffle=True,
    num_workers=1,
 )
```

Для решения задачи будет использована сеть состоящая из двух блоков: Encoder и Decoder. У каждого из блоков будет своя

функция.

Блок Encoder кодирует информацию из объекта (в частности, последовательности) в числовой вектор (иногда набор векторов) фиксированной размерности. Т.е. производит подсчет более интеллектуального и специфичного эмбединга входного объекта (в нашем случае текста).

Блок Decoder генерирует итоговый объект (в нашем случае последовательность токенов на целевом языке) на основе информации, полученной из Encoder.

Что же происходит глобально? В блоке Encoder информация об исходном объекте кодируется в численном виде, причем остается только принципиально важная информация без специфики исходного представления информации. Затем Decoder, используя только эту важную информацию, декодирует её в специфичный объект (в частности последовательность токенов).

Обычно блоки Encoder и Decoder симметричны. В нашем случае оба блока будут состоять из знакомого вам слоя nn.Embedding и LSTM ячейки. Для того, чтобы избежать однообразия выходных последовательностей, добавим Dropout слой. Приведем используемую нами реализацию модели.

Класс блока Encoder:

```
class Encoder(nn.Module):
    def __init__(
            self,
            vocab_len,
            embedding_dim,
            hidden_dim,
            n_layers,
            dropout_prob):
        super().__init__()

        self.embedding = \
            nn.Embedding(vocab_len, embedding_dim)
```

```
        self.rnn = nn.LSTM(
            embedding_dim,
            hidden_dim,
            n_layers,
            dropout=dropout_prob)

        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, input_batch):
        embed = self.dropout(self.embedding(input_batch))
        outputs, (hidden, cell) = self.rnn(embed)

        return hidden, cell
```

Наш энкодер состоит из простого LSTM блока. Он вернет нам состояния LSTM блока после прохода по всей последовательности, т.е. своего рода эмбеддинг всего поданного на вход текста.

Классы для блока Decoder:

```
class OneStepDecoder(nn.Module):
    def __init__(
            self,
            input_output_dim,
            embedding_dim,
            hidden_dim,
            n_layers,
            dropout_prob):
        super().__init__()
        # self.input_output_dim will be used later
        self.input_output_dim = input_output_dim

        self.embedding = \
            nn.Embedding(input_output_dim, embedding_dim)
        self.rnn = nn.LSTM(
```

```
            embedding_dim,
            hidden_dim,
            n_layers,
            dropout=dropout_prob)
        self.fc = nn.Linear(hidden_dim, input_output_dim)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, target_token, hidden, cell):
        target_token = target_token.unsqueeze(0)
        embedding_layer = \
            self.dropout(self.embedding(target_token))
        output, (hidden, cell) = \
            self.rnn(embedding_layer, (hidden, cell))

        linear = torch.nn.functional.softmax(
            self.fc(output.squeeze(0))
        )

        return linear, hidden, cell


class Decoder(nn.Module):
    def __init__(self, one_step_decoder, device):
        super().__init__()
        self.one_step_decoder = one_step_decoder
        self.device=device

    def forward(self, target, hidden, cell):
        target_len, batch_size = target.shape[0], target.shape[1]
        target_vocab_size = \
            self.one_step_decoder.input_output_dim
        predictions = torch.zeros(
            target_len,
            batch_size,
```

```
            target_vocab_size
        ).to(self.device)
        input = target[0, :]

        # Loop through all the time steps
        for t in range(target_len):
            predict, hidden, cell = self.one_step_decoder(
                input,
                hidden,
                cell)

            predictions[t] = predict
            input = predict.argmax(1)

        return predictions
```

В свою очередь декодер будет брать состояния из энкодера и на основе этого состояния за один шаг генерировать стартовый токен. Затем, уже опираясь на свои состояния и стартовый предсказанный токен, генерирует следующий токен. После этого на основе нового своего состояния и предыдущего своего предсказания генерирует еще один токен. Собственно, шаг декодера будет происходить с использованием объекта класса OneStepDecoder, а весь прогон декодера будет реализован с использованием класса Decoder, который будет возвращать уже всю итоговую последовательность.

Тогда итоговая модель будет иметь следующий вид:

```
class EncoderDecoder(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
```

```
7
8       def forward(self, source, target):
9           hidden, cell = self.encoder(source)
10          outputs = self.decoder(target, hidden, cell)
11
12          return outputs
13
14
15  embedding_dim = 256
16  hidden_dim = 256
17  dropout = 0.5
18  encoder = Encoder(
19      SOURCE_VOCAB_SIZE,
20      embedding_dim,
21      hidden_dim,
22      n_layers=2,
23      dropout_prob=dropout)
24  encoder = encoder.to(device)
25
26  one_step_decoder = OneStepDecoder(
27      TARGET_VOCAB_SIZE,
28      embedding_dim,
29      hidden_dim,
30      n_layers=2,
31      dropout_prob=dropout)
32  one_step_decoder = one_step_decoder.to(device)
33
34  decoder = Decoder(one_step_decoder, device)
35  decoder = decoder.to(device)
36
37  model = EncoderDecoder(encoder, decoder)
38
39  model = model.to(device)
```

Определим оптимизатор и лосс функцию. При вычислении лосс функции мы не хотим учитывать ошибки на токенах паддинга, потому что по большому счету не важно, что выводит модель после токена <END>. Поэтому определение этих объектов будет иметь такой вид.

```
optimizer = torch.optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IND)
```

Тогда цикл обучения модели может выглядеть следующим образом

```
clip = 1
epochs = 100

for epoch in range(1, epochs + 1):
    pbar = tqdm(
        total=epochs,
        bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}',
        unit=' epochs',
        ncols=200)

    # set training mode
    model.train()
    training_loss = []
    # Loop through the training batch
    for i, batch in enumerate(train_dataloader):
        # Get the source and target tokens
        data, target = batch
        src, trg = data

        src = src.to(device)
        trg = trg.to(device)
        target = target.to(device)
```

```
23
24          optimizer.zero_grad()
25
26          # Forward pass
27          output = model(src, trg)
28          # Calculate the loss
29          target = target.reshape(1, -1).squeeze(0)
30          output = output.reshape(-1, output.shape[-1]).float()
31          loss = criterion(output, target)
32
33          # back propagation
34          loss.backward()
35
36          # Gradient Clipping for stability
37          torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
38
39          optimizer.step()
40          training_loss.append(loss.item())
41          loss_cur_val = \
42              round(sum(training_loss) / len(training_loss), 4)
43          pbar.set_postfix(
44              epoch=f" {epoch}, train loss= {loss_cur_val}",
45              refresh=True)
46          pbar.update()
47      # Save model
48      checkpoint = {
49          'model_state_dict': model.state_dict(),
50          'source': source_texts,
51          'target': target_texts
52      }
53      torch.save(checkpoint,
54                 f'nmt-model-lstm-20_epoch_{epoch}.pth')
55
56      pbar.close()
```

Таким образом, можно решать задачу машинного перевода с любого языка на любой язык, только, разумеется, для достаточно высокого качества перевода нужно будет усложнить структуру енкодера и декодера и увеличить объем данных для обучения. Вообще, для данной задачи на данный момент наилучшие решения основываются на архитектуре transformer и, как части этой архитектуры, механизма внимания. Эти аспекты современного глубокого обучения будут рассмотрены более подробно в следующей главе.

3.7 Современные стандарты нейронных сетей

В данном модуле будут рассмотрены несколько особенных ячеек нейронных сетей, которые являются стандартом (как говорят "state of the art") современного (2022 год) глубокого обучения. Ячейки данного типа во многих задачах позволяют добиться существенных улучшений результатов. В данном блоке будут рассмотрены следующие ячейки:

- Attention
- Transformer

3.7.1 Attention

Как очевидно следует из перевода названия, — это «механизм внимания». Но что это значит? Вот это уже менее очевидно, давайте разбираться.

Идея

Рассмотрим задачу перевода текста с английского на русский: в частности, предложение «I am a student». В переводе на русский язык - «Я студент». Для нас все просто и мы знаем, что «I» - относится к местоимению «Я», «am» практически не играет роли при переводе в данной ситуации, «a student» переводится как «студент». Для нас все просто, но для нейронной сети не очевидно, что иногда из двух слов в английском языке при переводе может получиться одно слово в русском языке, а при переводе целых

четырех слов с английского языка получается только два слова на русском.

Другой пример:

Представим, что мы пытаемся перевести предложение

«The animal couldn't cross the road because it was tired.»

Для нас предложение простое и все очевидно:

«Животное не могло перейти дорогу, потому что оно устало.»

Но для алгоритма это достаточно сложная задача, потому что местоимение it может переводится на русский язык совсем разными способами, в зависимости от того, к какому слову оно относится. В данном случае местоимение it относилось к слову animal, т.е. должно переводиться как «оно».

В похожем примере

«The animal couldn't cross the road because it was wide.»

«Животное не могло перейти дорогу, потому что она была широкая.»

Местоимение it уже относится к слову road (дорога) и должно переводиться на русский язык как «она», т.е. понимание такой сложной (для алгоритма) связи слов в предложении критично для правильного перевода.

Из этих соображений вообще хочется научиться говорить нейронной сети: «Вот, смотри, на этом этапе перевода, ты знаешь вот это, ты уже сказала вот это, а сейчас нужно обратить внимание вот на эту информацию из изначального утверждения». Хочется явно указывать при переводе каждого слова, какая исходная информация наиболее важна сейчас, а какая нет. Механизм, который позволяет научить систему так поступать — это механизм внимания.

Внутренние детали

Теперь, поняв интуитивно, что это за механизм и зачем он нужен, давайте немного погрузимся в детали.

Вспомните, каким образом мы решали задачу перевода текста с одного языка на другой с использованием архитектуры Encoder-Decoder? Попробуйте сначала вспомнить самостоятельно.

Мы с вами брали тексты первого языка (с которого надо перевести) прогоняли их через некоторую рекуррентную архитектуру. Затем последнее скрытое состояние энкодера (в котором собрана вся инфомация об исходном предложении) отправлялось в декодер в качестве стартового скрытого состояния для декодера. На основе этой информации генерировалось предложение на втором языке (языке на который нужно перевести).

Но встает вопрос: мы хотели обращать внимание декодера на конкретные токены из исходного текста, именно на те, где хранится информация конкретно для данной итерации декодера являющаяся актуальной. Но как это сделать?

Первое, что приходит на ум, — брать скрытые состояния энкодера на соответствующих итерациях и передавать их дополнительно в декодер. Что значит, для первой итерации декодера брать скрытое состояние, полученное с первой итерации енкодера, для второй итерации декодера будем брать скрытое состояние со второй итерации энкодера. Кажется несложным заметить, что у такого подхода есть ряд проблем:

1. Один токен (слово) одного языка — это далеко не всегда один токен другого языка. Передавая же информацию между соответствующими итерациями энкодера и декодера мы передаем информацию лишь о токене, находящемся на том же месте в предложении, причем в разных языках, что нам вообще не гарантирует никакой смысловой связи.
2. В предложении первого языка может быть меньше токенов, чем должно быть в предложении второго языка, что также будет являться проблемой. Непонятно, что нужно передать на вход для «дополнительных» для второго языка слов.

В связи с приведенными выше утверждениями появляется следующая мысль: а давайте будем на конкретной итерации декодера дополнительно передавать ему на вход некоторую взвешенную сумму скрытых состояний энкодера. Причем веса будут зависеть от текущей в декодере информации. Сложно? Посмотрим на формулы.

Будем передавать дополнительно на вход декодера на итера-

ции j вектор h_j^{dec}:

$$h_j^{dec} = \sum t = 1^N a_j^t \cdot h_t^{enc}$$

где h_t^{enc} — скрытое состояние энкодера в t-й момент времени, т.е. на t-й итерации енкодера, а N — количество итераций энкодера для данной последовательности токенов (текста).

Другими словами, на вход декодеру дополнительно будет подаваться некоторая линейная комбинация скрытых состояний энкодера. Очевидно, хочется, чтобы для более важных токенов a_j^t было «большим» , а для менее важных токенов «маленьким». Таким образом декодеру будто будет сказано «Сейчас важно это, обрати внимание сюда».

Остается вопрос: «А как посчитать a_j^t»? Повторим, что данный коэффициент будет определять важность t-го токена энкодера для j-й итерации декодера. Тогда понятно, что значение этого коэффициента должно зависеть от (j-1) состояния декодера и от состояния энкодера в t-й момент времени, для t-го токена. Данное рассуждение записывается слеующим образом:

$$a_j^t = f(h_t^{enc}, h_{j-1}^{dec})$$

Что это значит? Это значит, что коэффициент должен зависеть от соответствующего скрытого состояния энкодера и от предыдущего состояния декодера.

Остается единственный вопрос: как выбрать функцию f? Например:

1. Простая нейронная сеть (например, один линейный слой)
2. Скалярное произведение
3. Что угодно:)

Затем все коэффициенты a_j^t для вычисления одного h_j^{dec} нормируются на единицу (например, с использованием функции softmax):

$$\sum_{t=1}^{N} a_j^t = 1$$

Таким образом могут быть вычислены коэффициенты внимания, которые позволяют получить информацию о каком-то из типов связи (комбинации типов связи) слов в предложении. Например, о согласовании склонений глаголов, рода прилагательных. Часто этот тип связи сложно определить конкретно, потому что это может быть комбинация разнообразных типов зависимости слов.

Тем не менее, чтобы наиболее качественно улавливать широкий спектр возможных взаимоотношений слов в предложений, часто используют "многоголовый attention» (Multi-head attention). Иначе, с разными параметрами вычисляют несколько наборов коэффициентов a_j^t (в этом случае коэффициентам лучше быть обучаемыми). И дополнительно, на вход декодеру передают уже не пару векторов, а весь набор векторов: предыдущее состояние декодера и векторы, полученные через каждый из наборов коэффициентов. Какой тип связи какому набору коэффициентов соответствует будет определено в процессе обучения автоматически.

Описанный выше подход позволяет передать в декодер информацию о наиболее важных на данный момент с разных точек зрения кусках исходного предложения.

Без этого подхода невозможно представить современные архитектуры, решающие задачи обработки последовательностей (в том числе, задачи NLP). В свое время этот подход позволил сильно ускорить обучение и улучшить результаты решения задач обработки текстов. В том числе существенно улучшить качество машинного перевода.

Через некоторое время на основе этой механики была придумана архитектура transformer. Ее мы рассмотрим ниже. Эта архитектура достигла ещё более высоких результатов, чем просто Attention при решении огромного количества задач NLP. Поэтому transformer на данный момент просто не заменим при решении большинства задач NLP.

3.7.2 Transformer

Transformer — особая архитектура, которая используется для работы с последовательностями. В ее основе лежит концепция внимания. Если быть точным, то концепция «самовнимания» (self-attention). Примечательно, что здесь нет ни единого упоминания о свертках или рекуррентных блоках, т.е. токены могут обрабатываться параллельно, что в разы уменьшает время инференса сети.

Также в данной архитектуре используются только: особая схема кодирования информации, Self-attention и пара других обычных, хорошо известных вам подходов.

Исходная схема

Рассмотрим базовую схему Transformer из оригинальной статьи:

В общих чертах схема должна быть понятна, так как:

- Слои Linear и Softmax вам хорошо знакомы;
- Feed Forward - линейный слой с функцией активации;
- Input Embedding - эмбеддинги входных элементов (например, токенов текста).

Остается разобрать лишь несколько непонятных, простых блоков и в целом архитектура Transformer станет вам понятна. В этом и состоит основная прелесть рассматриваемого нами типа сетей: они состоят из простых, просто связанных блоков, но тем не менее подобные сети сильно выигрывают у других архитектур. Это очень круто!

Блок Position Encoding

Входными эмбеддингами может быть что угодно (например, word2vec). Способ подсчета входного эмбеддинга выбирается произвольно на усмотрение исследователя. Но можно заметить, что ни в одном из уже изученных нами эмбеддингов не использована информация о местоположении токена в тексте. Почему?

Ответ: потому что специально добавлять информацию о местоположении токена было не нужно. Она автоматически подра-

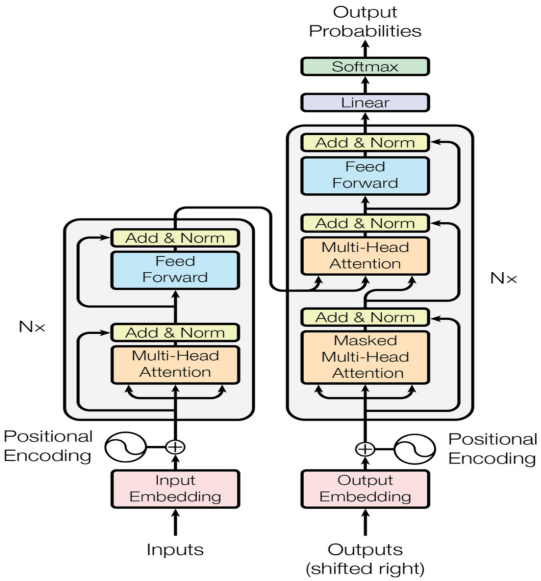


Figure 1: The Transformer - model architecture.

зумевалась и предоставлялась при прогоне рекурентного блока.

В данной архитектуре неявно нигде не извлекается информация о номере элемента последовательности, поэтому нужно её явно руками добавить во входной эмбеддинг. Поэтому предлагается каким-то образом в векторе закодировать информацию о позиции токена в векторе, но к этому способу предъявляются следующие понятные требования:

- Этот вектор должен быть уникальным для каждой позиции
- Расстояние между вектором, кодирующим позицию 2, и вектором, кодирующим позицию 4, должно быть равно расстоянию между векторами для позиции 33 и 35, что ожидаемо. Разумеется, имеется в виду условие для всех возможных пар равноудаленных позиций.

- В данном кодировании не должно быть случайностей. Построение такого вектора должно быть детерминированным.
- Хочется иметь возможность попытаться поработать с предложениями даже сильно длиннее, чем в тренировочных данных.

В связи с этим часто используют в качестве метода позиционного кодирования следующий вариант, который удовлетворяет всем упомянутым выше требованиям.

$$PE_{(pos,2i)} = \sin \frac{pos}{10000^{\frac{2(i+1)}{d_{model}}}}$$

$$PE_{(pos,2i+1)} = \cos \frac{pos}{10000^{\frac{2(i+1)}{d_{model}}}}$$

где pos — позиция слова в тексте, d_{model} — размер вектора позиционного кодирования (совпадает с размером эмбеддинга), $2i$ и $2i+1$ — индексы, начиная с 0, в векторе Positional Encoding.

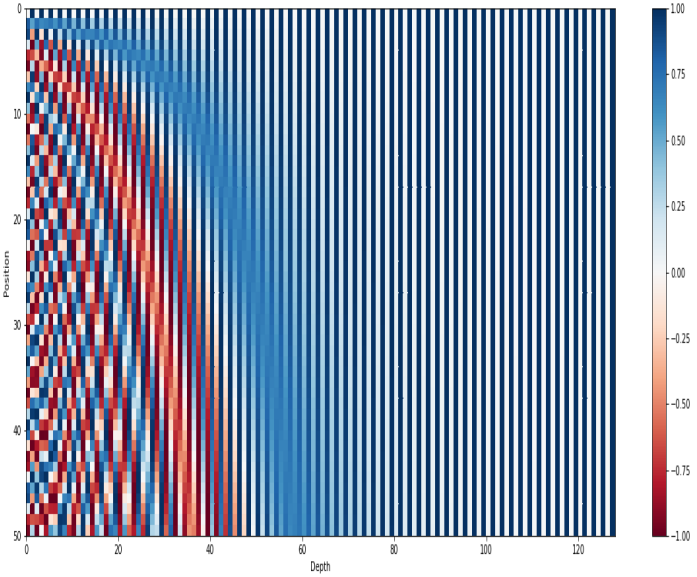
Кроме того, норма вектора PE сохраняется вне зависимости от pos, притом норма этого вектора равна в точности $\sqrt{\frac{d_{model}}{2}}$, что хорошо с точки зрения сохранения дисперсии значений в выходах промежуточных слоев сети.

Посмотрим на 128-мерные вектора Positional Encoding для 50 позиций (по картинке видно, что номер позиции увеличивается сверху вниз).

Из картинки видно, что конкретные компоненты (например, 20я и 40я) векторов с ростом номера меняются все хуже и хуже. Как раз из-за этого у Positional Encoding возникает некоторая проблема: с очень большими предложениями данный подход все равно работает не очень хорошо, так как отличие между векторами (покомпонентное) становится не очень большим.

Мы научились достаточно хорошо получать информацию о позиции токена. Но как ее использовать?

Комбинироваться вектор Positional Encoding и входного эмбеддинга могут абсолютно по-разному: это может быть конкате-



нация, сумма, взвешенная сумма. Чаще всего, и в классической архитектуре, эти вектора просто суммируются.

Блок Self-attention (Multi-Head Attention)

Мы с вами выше обсуждали механизм внимания. Вспомните, в чем он заключается.

Напомню: этот механизм позволяет явно подсказать декодеру, какая информация из энкодера важна конкретно на данной итерации декодера. Другими словами, если проводить аналогию с переводом, то при переводе предложения с английского на русский мы обращаемся к английскому предложению, т.е. к исходному.

Это мы с вами помним. Теперь давайте поймем, что, если предложение длинное и мы просто хотим хорошо понять смысл

этого предложения, то мы частенько смотрим на само это предложение целиком. Иначе обращаем внимание на исходное предложение, чтобы понять более конкретно и более качественно смысл слов в исходном предложении. Этим и занимается механизм self-attention.

С его помощью для текущего элемента текста можно получить контекстный вектор, который охарактеризует связь этого элемента с остальным текстом с некоторой, неведомой нам изначально, точки зрения. Здесь контекстный вектор — это взвешенная сумма входных эмбедингов.

Multi-Head self-attention работает аналогично простому механизму внимания. Здесь будут формироваться несколько контекстных векторов, каждый из которых будет отвечать за свою смысловую связь между элементами текста.

Теперь, собственно, рассмотрим, как работает механизм внимания в энкодере (в декодере работает похожим образом, за исключением пары деталей):

Пусть у нас задана последовательность эмбедингов:

$$x_1, x_2$$

,

И мы хотим подсчитать вектор контекста для x_3. Обозначим вектор контекста как z_3. Подсчитаем его используя механизм внимания (или самовнимания, если быть точным).

$$z_1 = \sum_{i=1}^{2} a_1^i v_i$$

где v_i — вектор value, поставленный в соответствие i-му элементу последовательности (вектор v_i вообще говоря не равен x_i).

Давайте теперь вместе с вектором value (v_i) каждому элементу последовательности поставим в соответствие вектора query (q_i) и key (k_i). Что это за вектора?

q_i — вектор, характеризующий элемент x_i, когда мы смотрим ИЗ него на всю остальную последовательность.

k_i — вектор, характеризующий элемент x_i, когда мы смотрим НА него из другого элемента последовательности.

Иначе говоря, при подсчете коэффициента внимания a_1^2 для вычисления z_1 элемент x_2 будет выглядеть как k_2, а сам элемент x_1 будет выглядеть как q_1. Откуда получаем ожидаемую идею:

$$a_1^2 = f(k_2, q_1)$$

Но какой вид имеет функция f?

Давайте придумаем эту функцию. Очевидно, что характеризовать взаимосвязь элементов x_1 и x_2 можно, например, простым скалярным произведением векторов k_2 и q_1. Чем больше это произведение, тем более сильно связаны эти элементы друг с другом.

$$score_1^2 = (k_2, q_1)$$

Также этот $score_1^2$ нужно поделить на $\sqrt{d}$, где d — размерность векторов k_2 и q_1. Это необходимо для лучшей обучаемости модели (чтобы после применения слоя Self-attention сохранялась дисперсия распределения входных параметров). Таким образом,

$$score_1^2 = \frac{(k_2, q_1)}{\sqrt{d}}$$

Затем, нормируем коэффициенты внимания на единицу (например, при помощи функции softmax).

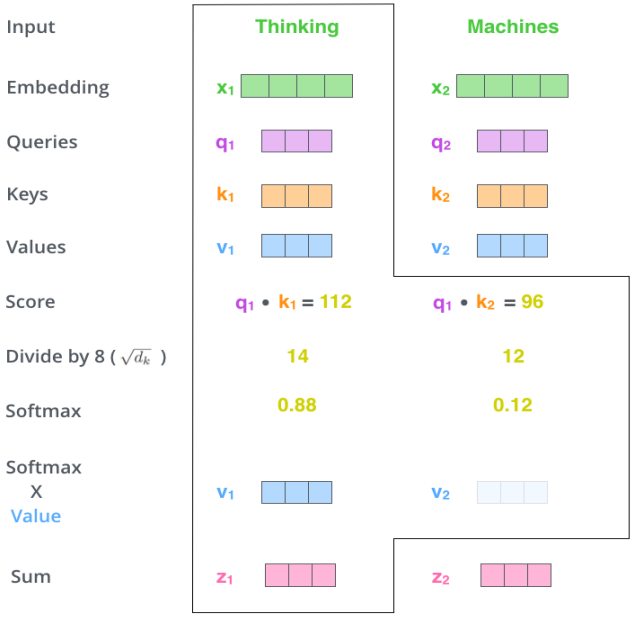
$$\sum_{i=1}^{2} a_1^i = 1$$

Таким образом, если конкретно выписать формулу, то получаем, что

$$a_1^2 = \frac{\exp score_1^2}{\sum_{i=1}^{2} \exp score_1^i}$$

$$a_1^2 = \frac{\exp \frac{(k_2, q_1)}{\sqrt{d}}}{\sum_{i=1}^{2} \exp \frac{(k_i, q_1)}{\sqrt{d}}}$$

Аналогичным образом можно подсчитать и a_1^1 и подсчитать z_1. Для большей наглядности продемонстрируем алгоритм на картинке



Сначала ставим в соответствие x_i вектора v_i, q_i и k_i. На их основе считаем score, делим его на $\sqrt{d}$, берем softmax и подсчитываем z_i.

По сути работаем с тем же самым механизмом внимания за тем лишь исключением, что нужно каким-то образом представлять элемент последовательности, когда мы смотрим из него на всю последовательность и когда смотрим на него из какого-то элемента последовательности.

Кажется, понятно, остался вопрос: где взять v_i, q_i и k_i? На самом деле, они получаются по очень простой формуле:

$$v_i = x_i \cdot W^v$$

;

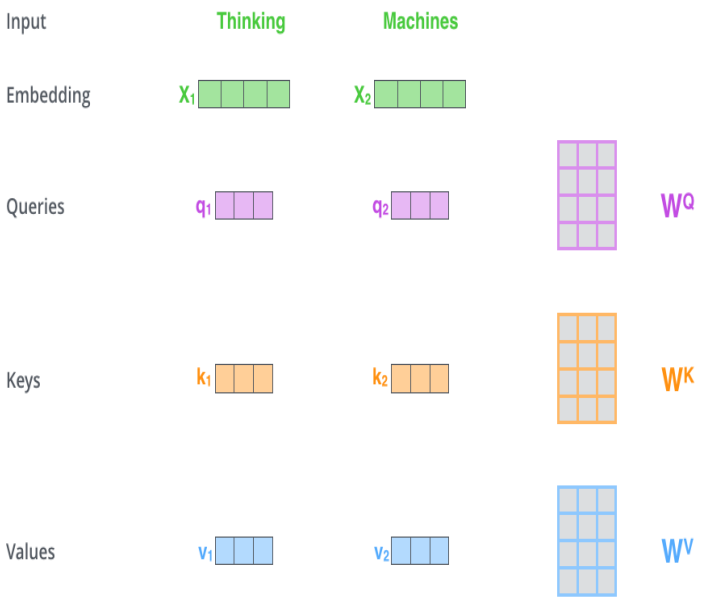
$$q_i = x_i \cdot W^q$$

;

$$k_i = x_i \cdot W^k$$

.

Таким образом, умножением на соответствующие матрицы (одинаковые для всех x_i) можно получить необходимые вектора.



Так для того, чтобы задать, например, «5-головый attention», нужно просто инициализировать и обучать 5 наборов матриц W^v, W^q и W^k.

На самом деле, для лучшей производительности, все операции, описанные выше, просчитываются одновременно для всех элементов последовательности. Это реализуется за счет распараллеливания матричных операций. Также «многоголовый attention» не реализуется как отдельный процесс для каждой «головы», все матрицы стягиваются в одну и оптимальным образом обрабатываются. Затем с обучаемыми весами складываются полученные в каждой голове z_i для конкретного x_i и получается один вектор контекста для каждого элемента последовательности

Но на данный момент в подробности реализации такого типа слоев погружаться не будем. Все происходит так, как описано

выше, только несколько более оптимизировано под вычисления на GPU и CPU.

Так это реализовано в энкодере.

В свою очередь Multi-Head Attention в декодере (который посередине блока декодера) уже смотрит на состояние энкодера R_1, R_2, R_3 и вычисляет «внимание» текущего состояния декодера $\hat{R}_i$ на соответствующее состояние энкодера. Это уже знакомый, обычный Multi-Head Attention, который мы рассмотрели несколько выше. Однако часто это реализуется очень похожим на Multi-Head Attention из энкодера образом:

Из R_1, R_2, R_3 с помощью матриц W^k, W^v вычисляются вектора k_j и v_j. Затем на основе $\hat{R}_i$ и матрицы W^q вычисляется q_i (т.е. кодировка объекта, когда ИЗ него смотрят). И описанным в этом модуле образом вычисляется контекстный вектор для $\hat{R}_i$.

Блок Add & Norm

В этом блоке не наблюдается чего-то особенного.

Как не сложно заметить, в этот блок у нас идут skipped connections (как в простом ResNet), такой подход решает проблему затухающих или взрывающихся градиентов. Затем в этом слое прокинутое значение каким-то образом суммируется с выходом предыдущего блока.

После этого компоненты результирующего вектора нормализуются, т.е. находим среднее компонент по всему итоговому вектору, их дисперсию. Затем из всех компонент вычитается среднее, и результат делится на корень из дисперсии. Это необходимо, чтобы не позволять расти норме векторов, задающих промежуточное состояние сети.

В этом блоке вообще ничего нового не наблюдалось:) Всё вам было хорошо известно и до этого. этот слой необходим для лучшей обучаемости и стабильности сети.

Блок Masked Muti-Head Attention

Этот блок функционирует аналогично блоку Multi-Head Attention в энкодере, но с одной поправкой.

Вектор контекста z_i считается не по всему тексту, а только по уже сгенерированной последовательности, что логично (мы же не умеем заглядывать в будущее). Т.е. вместо

$$z_i = \sum_{k=1}^{N} a_k^i v_k$$

Получаем

$$z_i = \sum_{k=1}^{i} a_k^i v_k$$

Поэтому при вычислении «внимания» на входной вектор будто накладывается маска, которая оставляет эмбединги до текущего номера итерации (включительно) нетронутыми, а эмбеддинги из «будущего» зануляет.

Нюансы обучения

Данная архитектура очень чувствительна к методике обучения. Наиболее часто используется вариация градиентного спуска Adam. В данном подходе в течение первых шагов накапливается информация о градиенте, о норме градиента, о пространстве параметров вокруг стартовой точки, что необходимо для «осознанного» обучения сети.

В начале обучения сеть вообще не владеет информацией о том, куда надо двигаться в пространстве параметров и практически случайно шагает в разные стороны. Архитектура Transformer к этому чувствительна, поэтому коэффициент обучения в процессе тренировки изменяется.

В самом начале (на первых нескольких шагах) часто learning_rate равен нулю, чтобы сеть просто «осмотрелась», стоя на месте. Затем learning_rate линейно, аккуратно увеличивают в течение какого-то времени, а затем начинают уменьшать. Более конкретно в виде формул частый вариант такого подхода обучения можно с некоторого момента времени t_0 представить так:

$$lr(t) = lr_{base} \cdot min(growth(t), decay(t))$$

где $growth(t) = \frac{t-t_0}{T_{warmup}}$ и $decay(t) = \sqrt{\frac{T_{warmup}}{t}}$

Притом если $t < t_0$, то $lr(t) = 0$

Такой процесс называется "прогрев и он просто необходим при обучении архитектуры Transformer. Такой способ обучения также может быть успешно использован и для других «вредных» архитектур.

4. Введение в обработку звука

Звуковой канал получения информации — один из наиболее информативных для современного человека после текста и фото (видео). В связи с этим не удивительно, что в нашем распоряжении находятся огромные запасы звуковых данных. По этой причине мы просто обязаны научиться обрабатывать эту информацию. Так же подумали исследователи.

На данный момент, задача обработки звука достаточно актуальна не только для исследователей, но и для больших компаний. В частности, технологии обработки звука могут быть применены в различных сферах бизнеса:

- голосовые помощники в приложениях
- замена сотрудников call-центров
- голосовые запросы к поисковым системам
- голосовые ассистенты в приставках к телевизору, колонках (Салют, Алиса, Алекса)
- генерация музыки

Поэтому кажется очевидным, что задача обработки звука достаточно актуальна на данный момент. В этой главе мы познакомимся с принципами работы со звуком, в частности с задачей обнаружения ключевого слова.

Но для начала познакомимся с рядами Фурье и с некоторой теорией колебаний.

4.1 Теория

4.1.1 Теория. Колебания

Колебаться (т.е. изменяться) может что-угодно. Например, в математическом маятнике колеблется угол, который составляет нитка с вертикалью. В частности, если в нулевой момент времени мы оттянем маятник в сторону и отпустим, то маятник начнет качаться.

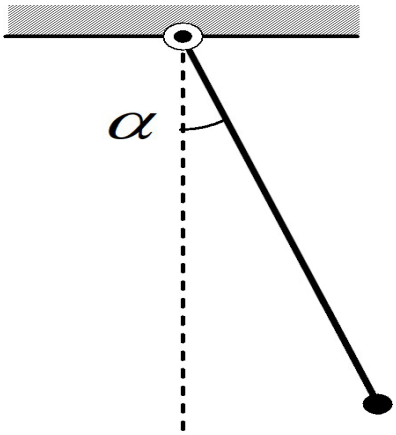


Рис. 4.1: Математический маятник

Угол α будет менять по следующему закону

$$\alpha = \alpha_0 \cos \omega t$$

Это простейший пример колебательной системы. График для альфа выглядит следующим образом (если $\alpha_0 = 10$, $\omega = 2$)

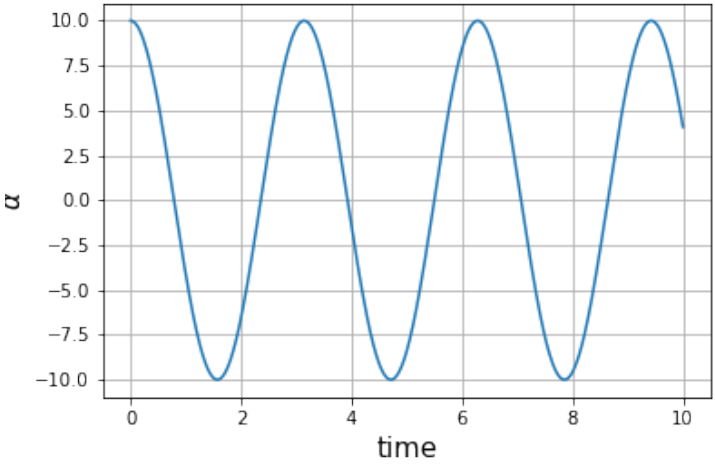


Рис. 4.2: $\alpha(t)$ для математического маятника

Теперь давайте представим, что информации о параметрах функции на α у нас нету, но мы хотим все равно как-то проанализировать поведение маятника. Для этого мы запустим систему и будем замерять угол α по 3 раза в секунду.

Таким образом получили бы следующие данные:

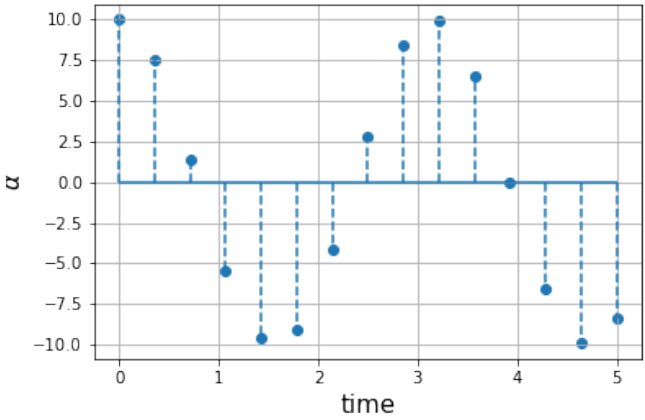


Рис. 4.3: Выборка для $\alpha(t)$ для математического маятника

Получили дискретизированную волну, своего рода сигнал. С частотой дискретизации 3 Гц.

Аналогичным образом можно работать с волнами и сигналами разнообразной природы. Это могут быть колебания грузика на пружинке, колебания электрического тока, колебания барабанной перепонки человека и чего угодно в принципе. Главное — научиться измерять интересующую нас величину с какой-то периодичностью.

Таким образом по полученному набору точек можно приближенно восстановить колебание. В нашем искусственном примере это можно сделать абсолютно точно, но в более реальном случае это уже будет сильно сложнее.

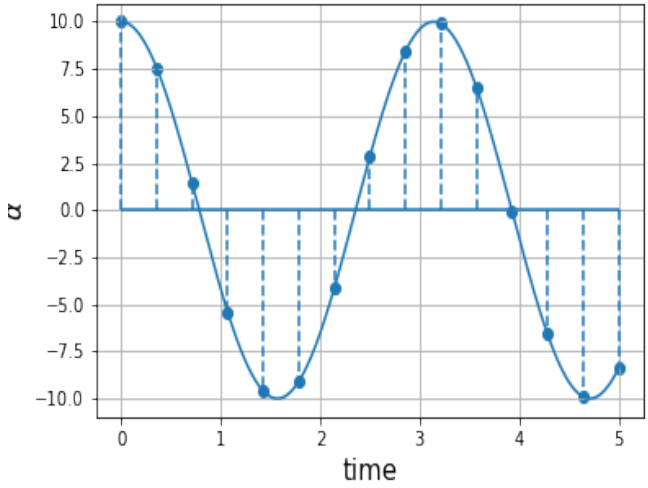


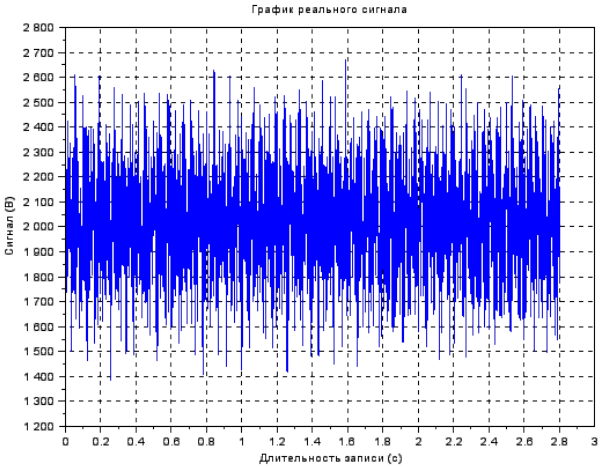
Рис. 4.4: $\alpha(t)$ для математического маятника

Таким образом, можно получить некоторое дискретное приближение практически любого колебания. Вообще, такой набор точек сам по себе не несет много информации. Вот если бы можно было выяснить по этим точкам частоту колебания, тогда мы могли бы все. Для этой задачи существует один метод, но пока мы его обсуждать не будем.

4.1.2 Теория. Ряды Фурье, преобразование Фурье.

Давайте вспомним рассмотренный с маятником пример, тут все очевидно. Простой математический маятник: колебание с одной частотой, колебание абсолютно периодическое. Но далеко не все реальные сигналы также просты.

Вот так выглядит типичный реальный сигнал, например, какого-то высокочастотного шума.



И что же делать с таким сигналом? Здесь нам поможет один очень мощный факт: многие функции, обладающие специальным свойством, могут быть представлены на некотором отрезке в виде суммы следующего вида

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\inf}(a_n \cos n\omega x + b_n \sin n\omega x)$$

Представим теперь тригонометрические функции через соответствующие комплексные экспоненты:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\inf}(a_n \frac{e^{in\omega x} + e^{-in\omega x}}{2} + b_n \frac{e^{in\omega x} - e^{-in\omega x}}{2i}) = \frac{a_0}{2} + \sum_{n=1}^{\inf}((\frac{a_n}{2} + \frac{b_n}{2i})e^{in\omega x} + (\frac{a}{2}$$

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\inf}(c_n e^{in\omega x} + c_{-n} e^{-in\omega x}$$

где $c_n = \frac{a_n}{2} + \frac{b_n}{2i}$, $c_{-n} = \frac{a_n}{2} - \frac{b_n}{2i}$

Вообще говоря, эта сумма может быть и конечной, например,

$$f(x) = 1 + sin(x) = 1 + \frac{1}{2i}e^{ix} - \frac{1}{2i}e^{-ix}$$

уже представлена в виде, приведенном немного выше.

Вы спросите: «Ну и что нам дает это представление? Зачем?». На самом деле возможность такого представления говорит об одном важном моменте: любое периодичное колебание (в том числе звук иногда) — это есть сумма простых колебаний (синусов или косинусов) с кратными частотами, т.е. с частотами ω, 2ω, 3ω и так далее.

Таким образом получаем, что реальный сигнал может быть описан набором частот, на которые он раскладывается, и амплитуд, соответствующих этим частотам (набор коэффициентов c_n и c_{-n}). Такое представление сигнала называется спектром. Обычно он приводится в следующем виде. Например, для $f(x) = 1 + \cos x = 1e^{i0x} + \frac{1}{2}e^{ix} + \frac{1}{2}e^{-ix}$ спектр будет выглядеть следующим образом:

Как видно на графике, при построении спектра по оси y откладывается модуль коэффициента c_n (также и в случае, когда c_n комплексное) перед соответствующей экспонентой, а по оси x откладывается частота соответствующей компоненты.

Еще один пример: $f(x) = 1 + sin(x) = 1 + \frac{1}{2i}e^{ix} - \frac{1}{2i}e^{-ix}$

Спектр данной функции будет выглядеть следующим образом:

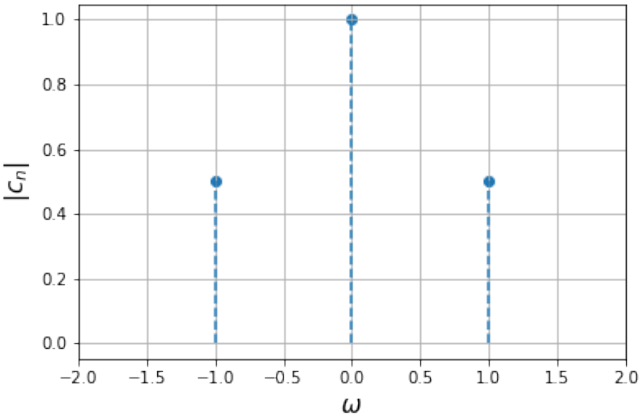


Рис. 4.5: Спектр $f(x) = 1 + \cos x$

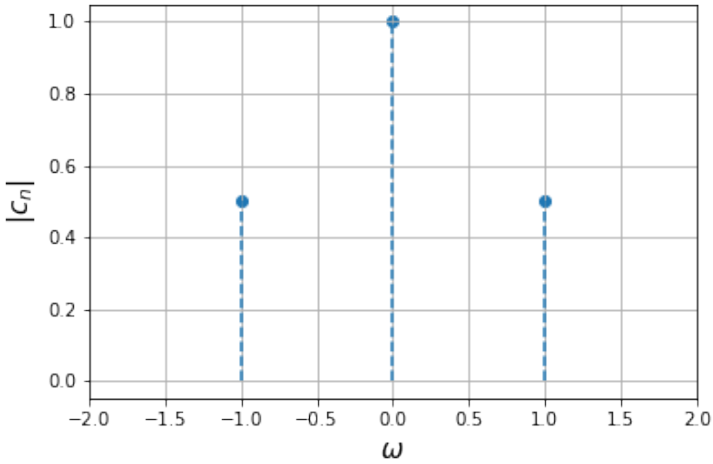


Рис. 4.6: Спектр $f(x) = 1 + \sin x$

Как видно, принципиально этот случай ничем не отличается от предыдущего, что ожидаемо, так как косинус и синус различаются только фазовым сдвигом — $\cos x = \sin \frac{\pi}{2} - x$.

Спектральное представление на самом деле гораздо более информативно, чем представление сигнала в виде набора точек, которое мы получали в самом начале. Оно содержит исключительно важную информацию о сигнале, так как менее значимые компоненты будут иметь меньшие значения амплитуд (коэффициентов c_n), а более важные компоненты — большие значения коэффициентов c_n.

Откровенно говоря, то представление, которое мы с вами обсудили имеет место только для особенно хороших функций и сигналов. Но у такого представления есть более мощный аналог, который работает подобно рассмотренному.

Основополагающее утверждение: любому сигналу $f(t)$ можно поставить в соответствие функцию $c(\omega)$, такую что

$$f(t) = \frac{1}{2\pi} \int_{-\inf}^{inf} (c(\omega) e^{-i\omega t} d\omega$$

Как можно заметить, функция $c(\omega)$ является своего рода аналогом амплитуды (коэффициентов c_n) в рассмотренном ранее разложении. И эта функция также информативна для произвольного сигнала, как и коэффициенты c_n. Но, в отличие от тех коэффициентов, $c(\omega)$ - это уже функция, способная характеризовать произвольный сигнал, а не только обладающий специальными свойствами.

Пример сигнала (одиночный прямоугольный импульс) и его непрерывного спектра (функции $c(\omega)$):

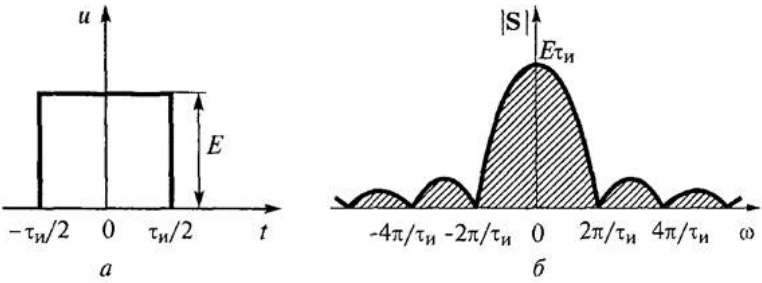


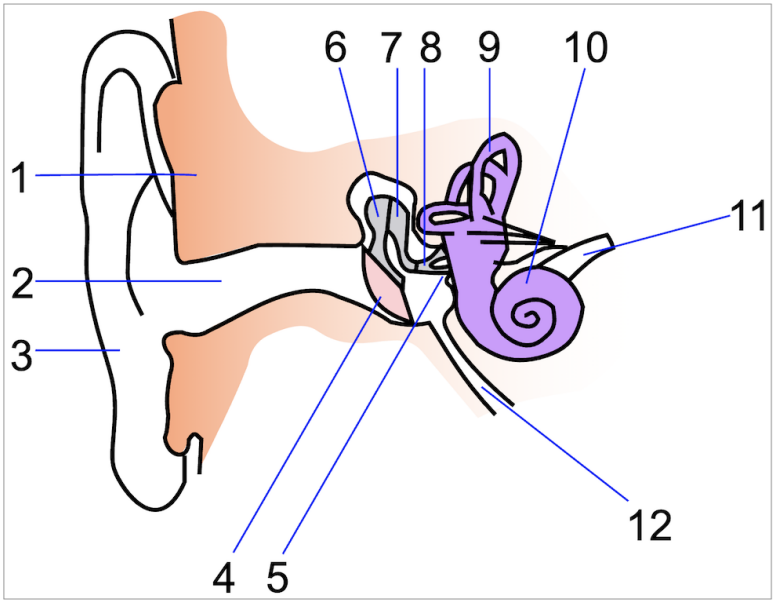
Рис. 4.7: Сигнал в виде прямоугольного одиночного испульса (а) и спектр (непрерывный) этого сигнала (б)

Теперь со спокойной совестью вернемся к конкретному типу колебаниф — к звуку.

4.1.3 Что такое звук?

Звук — это волна определенного диапазона частот, слышимых человеческим ухом. Диапазон частот звуковых волн — от 16—20 Гц до 15—20 кГц.

Звуковая волна распространяется (чаще всего) в упругой среде — воздухе. Мы слышим звук примерно так: волна, прошедшая через ушную раковину, колеблет барабанную перепонку среднего уха, с которой связаны органы: молоточек и наковальня. Они передают колебания во внутреннее ухо с улиточкой и нервами.



1 — височная кость; 2 — слуховой канал; 3 — ушная раковина; 4 — барабанная перепонка; 6 — молоточек; 7 — наковальня; 8 — стремечко; 5 — овальное окно; 9 — полукружные каналы; 10 — улитка; 11 — нервы; 12 — евстахиева труба.

Рис. 4.8: Строение уха

Похожим образом работают цифровые устройства для записи звука: обычно в микрофонах есть мембрана, которая колеблется от звуковых волн. Отклонения мембраны от первоначального положения записываются микрофоном несколько тысяч раз в секунду (обычно от 8000 до 48000, чаще всего 24000). Таким образом получаем дискретизированный звуковой сигнал с соответствующими частотами дискретизации (частотой записи отклонений мембраны), т.е. считываются точки какой-то неизвестной

функции $x(t)$ (отклонений мембраны) с частотой от 8 до 48 кГц (аналогично функции $\alpha(t)$).

Такое представление сигнала называется time-domain представление (аналогично рассмотренному нами изначальному представлению функции $\alpha(t)$). Оно, как мы обсуждали, не очень информативное.

Поэтому для анализа звуковых данных используют другие представления звукового сигнала.

4.1.4 Спектрограмма

Спектрограмма звукового сигнала — это некоторое спектральное представление звука. Что-то похожее на спектры, которые мы обсуждали выше, но все-таки несколько другое.

Для начала еще раз продемонстрируем необходимость создания других признаков. На картинке представлены два достаточно разных сигнала с идентичными спектрами, т.е. если бы мы использовали в качестве признаков спектры сигналов, то эти сигналы нам бы не удалось различить.

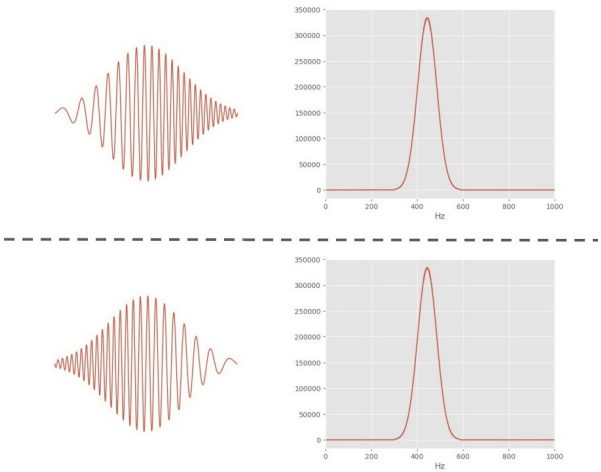


Рис. 4.9: Спектры для отличающихся сигналов

Но эти сигналы хочется уметь различать. Поэтому нужно придумать такое описание звука, которое будет тонко улавливать структуру сигнала (звука). Можете попробовать придумать такой признак для аудио самостоятельно.

Давайте разобьем сигнал на небольшие кусочки (по Δt) и для каждого кусочка будем отдельно искать спектр. Таким образом для каждого промежутка времени Δt у нас будет свой спектр. Что позволит улавливать локальную по времени структуру сигнала. Но звук — это все-таки сигнал, т.е. меняется со временем в каком-то смысле плавно. А мы только что предложили рассматривать кусочки размером Δt независимо друг от друга, что, кажется, не очень хорошо.

Поэтому на практике часто берут эти кусочки перекрывающимися. Что это значит? В первом варианте рассматривались спектры кусочков сигнала $(0, \Delta t), (\Delta t, 2\Delta t)$ и т.д. А теперь пред-

лагается смотреть на спектры перекрывающихся кусочков, например, $(0, \Delta t)$, $(\frac{1}{5}\Delta t, \frac{6}{5}\Delta t)$ и т.д. Что позволяет более плавно следить за изменениями звука. Часто спектрограмму отображают в следующем виде:

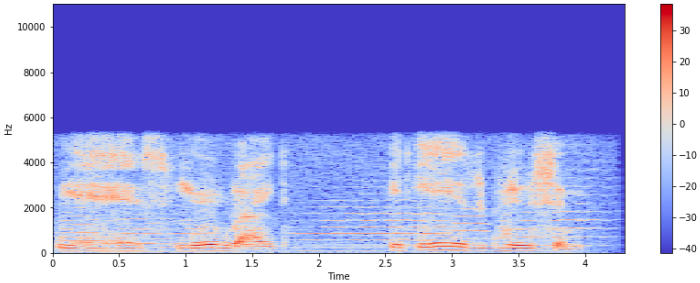


Рис. 4.10: Пример спектрограммы сигнала

По оси Y откладывается частота, по оси X — время, а цветом выражен размер амплитуды в спектре в соответствующий момент времени при соответствующей частоте. Другими словами, для каждого момента времени (интервала времени) построили спектр и отобразили его при помощи различного цвета.

Как можно заметить, половина картинки просто одноцветная. Что это значит? Это значит, что в записи, практически нет компонент с частотой выше 5-6кГц. На самом деле не только в этой записи, но и в обычном голосе редко встречаются такие высокие частоты. В связи с этим часто используют не просто спектрограмму, а Log-спектрограмму, просто переведя ось Y в логарифмический масштаб.

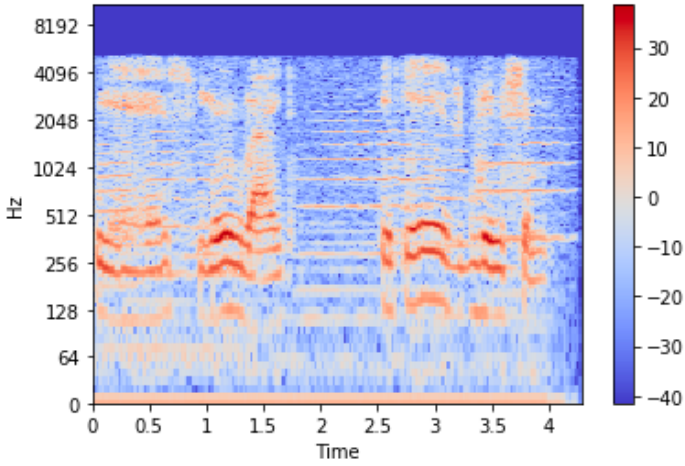


Рис. 4.11: Пример лог-спектрограммы

Посмотрите, значение на каждой последующей метке на оси Y отличается от предыдущей ровно в два раза, т.е. уже большую часть картинки занимает полезная информация, что мы и пытались получить. Это уже лучше, чем было.

4.1.5 Мел-спектрограмма

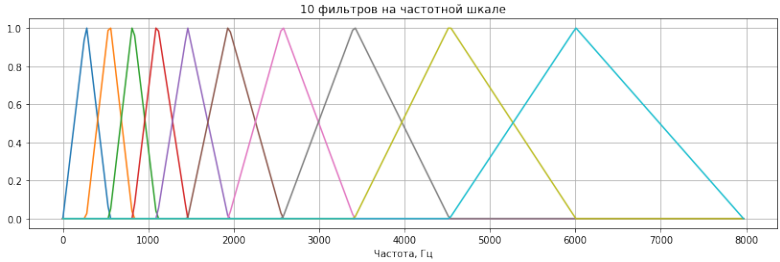
Однако, на практике исходную спектрограмму преобразуют несколько иначе. Наше восприятие звука так устроено, что мы лучше воспринимаем низкочастотные сигналы (до какой-то границы) и хуже воспринимаем высокочастотные сигналы. Что это значит?

Это значит, что если частота сигнала изменится со 100Гц до 120Гц, то человек почти наверняка уловит это изменение, а если частота изменится с 10000Гц до 10020Гц, то вряд ли это изменение будет легко заметить. В связи с этим от частот переходят к мелам.

Мелы вычисляются следующим образом:

1. Строится система треугольных фильтров (например, из 64 фильтров) для частот
2. Каждый фильтр накладывается на спектр для каждого момента времени обособленно
3. Расчет для каждого фильтра даст соответствующий мел коэффициент для соответствующего момента времени

Приведем пример системы из 10 фильтров:



Собственно, для каждого момента времени спектр будет умножаться на этот фильтр, а затем будет производиться суммирование по всем частотам. Так и получится соответствующий фильтру и моменту времени мел-коэффициент.

Очередной пример спектрограммы.

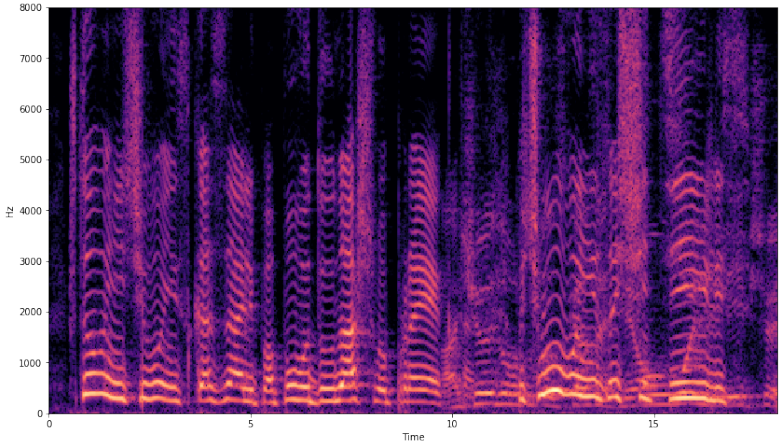


Рис. 4.12: Пример спектрограммы сигнала

А вот она же, но уже в мел единицах (64 фильтра):

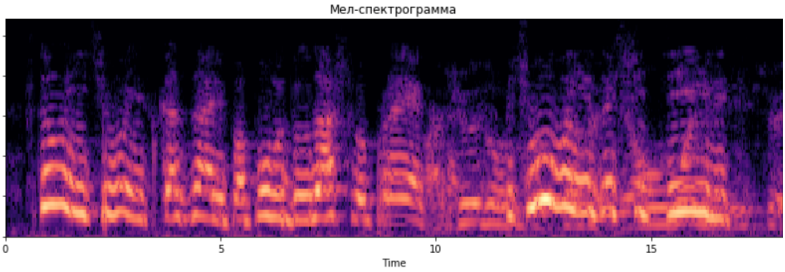


Рис. 4.13: Мел-спектрограмма для того же сигнала

Видно, что нижняя часть картинок (нижние частоты) практически идентичны, а вот в высокочастотной части спектрограммы произошло существенное усреднение. Также после перехода к мел-спектрограмме большая часть картинки содержит полез-

ную информацию. Получили, что и хотели получить от мел-спектрограммы.

Но на мел-спектрограмме также наблюдается большое количество лишней информации (много черных областей). Эту проблему можно немного решить перейдя к лог-мел-спектрограмме, переведя ось Y в логарифмический масштаб.

4.1.6 MFCC

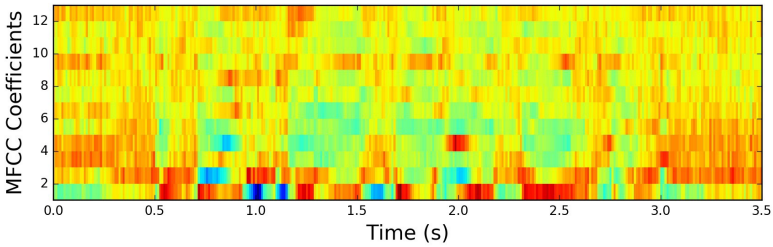
На самом деле и лог мел спектрограмма имеет свои проблемы, но они более глубокие. Как видно из различного типа спектрограмм, соседние пиксели похожи друг на друга, т.е. признаки аудио скоррелированы. Что, как мы с вами помним, не очень хорошо и может влиять на скорость сходимости и качество решения различного типа задач.

В целом для любых данных важно отсутствие скореллированности признаков, описывающих эти данные. В связи с этим на основе мел-спектрограмм часто формируют другие признаки: MFCC (Mel Frequency Cepstral Coefficients). Такие коэффициенты формируются следующим образом:

1. Вычисляем мел коэффициенты в момент времени t
2. Возводим их в квадрат, логарифмируем
3. Вычисляем от полученных коэффициентов дискретное косинусное преобразование (почти аналог преобразования Фурье)

Математически показывается, что полученные коэффициенты уже будут гораздо менее скоррелированы, чем мел коэффициенты. Именно поэтому на данный момент практически во всех актуальных работах используются именно MFCC признаки.

Пример MFCC признаков:



По цвету видно, что признаки не скоррелированы. Удалось добиться того, чего хотели.

Перейдем теперь к решению задачи поиска ключевого слова Keyword Spotting.

4.2 Keyword Spotting (Spotter)

4.2.1 Постановка задачи

Данная задача заключается в том, что нужно определить, есть в аудио ключевое слово или нет. Вообще говоря налицо типичная задача классификации, причем, если слово одно, то это задача бинарной классификации, а если нужно различать два и более слов, то это задача многоклассовой классификации.

С типом задачи разобрались, с ним мы знакомы, но как ее решать? Все зависит от того, в каком виде мы умеем представлять запись. Если мы умеем представлять её в виде числового вектора, то можно решать задачу классификации, например, полносвязной нейронной сетью, как мы уже умеем. Если аудио можно преобразовать в информативное изображение, то давайте решать задачу классификации с использованием сверточных сетей.

Обладая широким аппаратом нейросетевых технологий, достаточно научиться представлять объект (в самом широком смысле этого слова) в наиболее удобном и информативном виде, и работать с ним с использованием известных вам архитектур, как с картинкой, числовым описанием или последовательностью. Это

очень круто.

Разумеется, для большинства задач существуют специфичные структуры и специфичные представления объектов в том или ином виде, которые позволяют наиболее качественно решать конкретную задачу. Но тем не менее поражает, что уже с вашим набором знаний можно придумать, как хоть в каком-то приближении решать почти любую актуальную задачу. Это супер круто, на мой взгляд! Но это все философия.

4.2.2 Один из подходов

Вернемся к задаче Spotter.

Хотим определять есть в записи ключевое слово или нет. Как решать? Давайте переведем запись в картинку и решим на сформированном датасете картинок задачу классификации.

Перевод записи в картинку мы с вами уже долго обсуждаем: это может быть даже time-domain представление сигнала (но оно обладает кучей недостатков), спектрограмма, мел-спектрограмма, лог-спектрограмма, лог-мел-спектрограмма, MFCC и многие другие. Уже этих представлений сигнала достаточно, чтобы научиться решать задачу Spotter в каком-то приближении.

Очевидно, что для этой задачи существуют специфичные архитектуры, которые нужно использовать именно в задаче определения ключевого слова, потому что поиск слова — задача непростая. Но в применении к робототехнике, робофутболу в частности, кажется, наиболее важно определять прозвучал свисток судьи или нет. Согласитесь, сигнал свистка уже намного проще заметить, чем слово, конкретное слово. В связи с этим не будем рассматривать актуальные в промышленных проектах архитектуры, а рассмотрим несложный, работающий пример сети, которая будет неплохо решать задачу Spotter для нескольких слов.

Тогда, если немного адаптировать эту архитектуру под задачу определения только свистка, возможно, даже несколько упростив архитектуру, то, вероятно, она будет достаточно качественно

справляться с поставленной задачей.

4.2.3 Пример задачи

Теперь давайте конкретно рассмотрим пример решения задачи Spotter. Есть такой открытый датасет. В этом датасете есть данные для решения задачи Keyword Spotting для 30+ ключевых слов: bed, bird, cat, dog, down, eight, five, four, go, happy, house, left, marvin, nine и другие.

Нам такой объем данных и классов для демонстрации не нужен. Мы возьмем данные 5 классов: up, down, right, left, stop. Чтобы не было очень скучно, в качестве признаков будем использовать, опять же для примера, мел-спектрограммы. Под другие признаки код легко адаптируется.

Для начала установим sox — это полезная программа, которая позволяет легко и быстро получить некоторую информацию о звуке, например, частоту дискретизации (или как чаще называют sample rate), количество каналов и тд.

Установим sox, скачаем датасет и считаем данные (в нашем случае соответствие класса и набора путей к аудио для этого класса):

```
import os
datadir = "speech_commands"

!apt install sox
tensorflow_url = http://download.tensorflow.org
download_url = \
    f"{tensorflow_url}/data/speech_commands_v0.01.tar.gz"
!wget {download_url} -O speech_commands_v0.01.tar.gz

!mkdir {datadir}
!tar -C {datadir} -xvzf speech_commands_v0.01.tar.gz

samples_by_target = {
    cls: [
        os.path.join(datadir, cls, name)
        for name in os.listdir("./speech_commands/{}".format(cls))]
    for cls in os.listdir(datadir)
    if os.path.isdir(os.path.join(datadir, cls))
}
```

Получим информацию об одном из аудио из этого датасета:

```
!sox --info speech_commands/bed/00176480_nohash_0.wav

Output:

Input File     : 'speech_commands/bed/00176480_nohash_0.wav'
Channels       : 1
Sample Rate    : 16000
Precision      : 16-bit
Duration       : 00:00:01.00 = 16000 samples ~ 75 CDDA sectors
File Size      : 32.0k
Bit Rate       : 256k
```

```
12 Sample Encoding: 16-bit Signed Integer PCM
```

И увидим, что это аудио (и все аудио) является одноканальным с частотой дискретизации 16кГц.

Напишем функцию препроцессинга для конкретного пути к аудио, из которого можно вытащить лейбл класса для этого аудио, а также считать эту запись и обработать ее (в нашем случае рассчитать мел-спектрограмму)

```
1  import librosa
2  import numpy as np
3
4  classes = ("left", "right", "up", "down", "stop")
5
6  def preprocess_sample(filepath, max_length=150):
7      amplitudes, sr = librosa.core.load(filepath)
8      spectrogram = librosa.feature.melspectrogram(
9          amplitudes,
10         sr=sr
11     )[:, :max_length]
12     spectrogram = np.pad(
13         spectrogram,
14         [[0, 0], [0, max(0, max_length - spectrogram.shape[1])]],
15         mode='constant')
16     target = classes.index(filepath.split(os.sep)[-2])
17
18     return np.double(spectrogram), np.int64(target)
```

Сначала считали аудио и sample_rate, затем с использованием модуля librosa рассчитали мел-спектрограмму, которую выровняли до конкретного размера по оси времени. Что это значит? Это значит, что к спектрограмме дописали справа 0 значения, если аудио короче какого-то фиксированного времени, а если длиннее, то обрезали лишнее. Также из пути аудио в этой функции

извлекается target, и данные возвращаются в нужных форматах.

Определим на чем мы будем производить расчеты. На ГПУ или на ЦПУ? Давайте сделаем так, чтобы при доступности ГПУ наш код выполнялся на ГПУ.

```
if torch.cuda.is_available():
  device = torch.device("cuda")
else:
  device = torch.device("cpu")
```

Тогда нужно будет все параметры перенести на один конкретный ресурс (ГПУ или ЦПУ). В частности, саму модели и обучающие данные. Также данные для валидации.

По большому счету — это все, что нам было необходимо, дальше можно делать по-разному. В нашей задаче данных достаточно немного, поэтому можно предподсчитать мел-спектрограммы для всех используемых аудио. Это ускорит обучение, так как признаки уже сгенерированы. Предподсчитать фичи для всех аудио можно следующим образом:

```
from itertools import chain
from tqdm import tqdm
import joblib as jl

all_files = chain(*(samples_by_target[cls] for cls in classes))
spectrograms_and_targets = jl.Parallel(n_jobs=-1)(
    tqdm(
        list(
            map(jl.delayed(preprocess_sample), all_files)
        )
    )
)
X, y = map(np.stack, zip(*spectrograms_and_targets))
X = X.transpose([0, 2, 1])  # to [batch, time, channels]
```

С этого момента вообще можно забыть, что у нас были какие-то аудио и работать с картинками, каждая из которых относится к одному из пяти классов.

Мы же поступим немного иначе. Если данных слишком много и в оперативную память все фичи для всех данных не помещаются, то мы что делаем? Правильно, пишем свой класс Dataset, в котором будут определены методы __len__ и __getitem__. Также этот класс должен наследоваться от класса torch.utils.data.Dataset. Это позволит нам хранить в оперативной памяти только список путей до файлов, а затем при надобности (в методе __getitem__) считывать данные и преобразовывать их так, как нужно. Да, это увеличивает время обучения, но иногда просто нет другого выхода.

```
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
  def __init__(self, filepaths, max_length=150):
    self.filepaths = filepaths
    self.max_length = max_length

  def __len__(self):
    return len(self.filepaths)

  def __getitem__(self, ind):
    features, label = preprocess_sample(
        filepath,
        max_length=self.max_length)
    return features, label

```

Создадим необходимые объекты классов CustomDataset и DataLoader.

```
batch_size = 16
train_dataset = CustomDataset(train_files)
test_dataset = CustomDataset(test_files)


trainloader = DataLoader(train_dataset, batch_size=batch_size,
                         shuffle=True, num_workers=2)
testloader = DataLoader(test_dataset, batch_size=batch_size,
                        shuffle=False, num_workers=2)
```

В нашем случае в классе CustomDataset определен лист с путями до аудио, из которого по индексу берется конкретный путь, который затем обрабатывается известной нам функцией preprocess_sample . Затем на основе этих датасетов создаются стандартные, уже знакомые вам «даталоадеры» (объекты класса DataLoader), которые уже будут использованы непосредственно при обучении наших сетей.

После этого нам остается только написать класс нашей нейронной сети, возможно, сверточной сети, в уже знакомом вам формате

```
import torch.nn as nn
import torch.nn.functional as F

class CustomModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 64, 5)
        self.bn1 = nn.BatchNorm2d(64)
        self.dropout1 = nn.Dropout(0.2)

        self.conv2 = nn.Conv2d(64, 128, 5)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(2)
```

```
14
15          self.conv3 = nn.Conv2d(128, 128, 5)
16          self.bn3 = nn.BatchNorm2d(128)
17          self.pool3 = nn.MaxPool2d(4)
18
19          self.conv4 = nn.Conv2d(128, 128, 5)
20          self.bn4 = nn.BatchNorm2d(128)
21          self.pool4 = nn.MaxPool2d(6)
22
23          self.flatten = nn.Flatten()
24
25          self.dense1 = nn.Linear(256, 256)
26          self.dense2 = nn.Linear(256, 5)
27
28      def forward(self, x):
29          out = self.conv1(x)
30          out = self.bn1(out)
31          out = F.relu(out)
32          out = self.dropout1(out)
33          out = self.conv2(out)
34          out = self.bn2(out)
35          out = F.relu(out)
36          out = self.pool2(out)
37          out = self.conv3(out)
38          out = self.bn3(out)
39          out = F.relu(out)
40          out = self.pool3(out)
41          out = self.conv4(out)
42          out = self.bn4(out)
43          out = F.relu(out)
44          out = self.pool4(out)
45          out = self.flatten(out)
46          out = self.dense1(out)
47          out = F.relu(out)
```

```
        out = self.dense2(out)
        out = F.softmax(out, dim=1)

        return out


model = CustomModel()
model.to(device)
```

Или взять какую-то известную, проверенную архитектуру (например, resnet18), которая в силу своей мощи легко справится с приведенной задачей.

А далее уже обучать модель в давно знакомом вам цикле обучения.

Хорошо, с тем, как обучать модель, мы разобрались. Но как оценивать качество работы модели? Например, можно использовать accuracy, скажете вы. Да, вполне неплохое решение. Но на самом деле есть более показательные метрики.

Вообще зачем нам хочется уметь решать задачу Keyword Spotting? В первую очередь, для вызова голосовых ассистентов, для управления умным домом или тв приставкой. Вы говорите: «Салют, включи фильм Железный Человек», и вам включается фильм. Удобно, правда? Как это работает?

На самом деле, колонка слушает все, что вы говорите (но, разумеется, никуда не сохраняет записи и не отправляет!) и ждет, пока вы скажете ключевое слово. Как только колонка решила, что вы обратились к ней, то она включается во всю и слушает, что вы от нее хотите, и выполняет. Это оптимально. Другой подход привел бы к неоправданно высоким расходам энергии и трафика.

Таким образом, колонка стоит на полочке, слушает все и должна среагировать в нужный момент времени. Получаем, что наиболее важно

1. Уменьшить количество ложных срабатываний
2. Уменьшить количество проигнорированных обращений

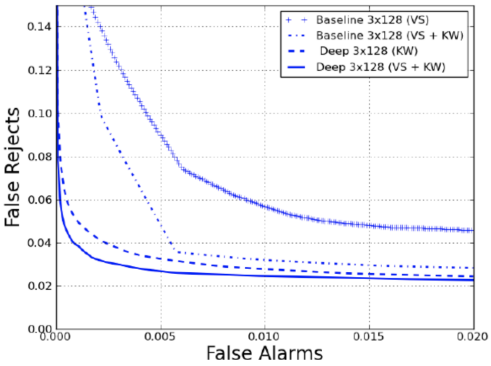
Это два взаимосвязанных момента.

Вообще, алгоритм предсказывает с какой вероятностью в аудио было озвучено ключевое слово. Поэтому нужно подобрать какую-то границу, начиная с которой мы будем считать, что слово прозвучало. И чем выше эта граница, тем чаще мы будем пропускать реальные обращения к колонке, но зато будет меньше ложных срабатываний. Напротив, если эта граница будет слишком низкой, то ложных срабатываний может быть слишком много. В связи с этим приходится искать какой-то компромис.

Это мы с вами поняли. В процессе оптимизации упомянутых величин очень важную роль играет как раз эта самая граница. И, вероятно, для каждого значения этой границы будет своя характеристика как количества ложных срабатываний, так и для количества проигнорированных обращений.

Тогда, очевидно, хочется ввести характеристику для количества ложных срабатываний — это количество ложных срабатываний в час (False activations hour FA/h), и для доли проигнорированных обращений из всех (False rejects / instance т.е. доли ложно отклоненных запросов). И очевидно, что для каждой выбранной границы эти значения будут свои. Вид этих графиков и способ построения чем-то схож с построением ROC-кривой и также очень помогает при выборе оптимальной границы и сравнении моделей.

Вот пример таких кривых из Статьи



Каждая точка кривой соответствует своему граничному значению. И по этой кривой, в зависимости от требований к вашей системе, можно найти оптимальное пороговое значение вероятности, начиная с которого мы будем считать, что ключевое слово прозвучало в записи.

4.2.4 Заключение и наставление

В этом блоке мы рассмотрели задачу Keyword Spotting. На самом деле это далеко не единственная актуальная задача обработки звука. Есть ещё ASR (Automatic Speech Recognition) т.е. задача автоматического распознавания речи, TTS (Text-To-Speech) т.е. преобразование текста в речь, Denoising или чистка шума. Все эти задачи решаются во многих компаниях и по сей день. Но мы их обсуждать не будем, так как эта тема уже более специфичная и требует более крепкой теоретической базы.

С существующими моделями, решающими эти задачи, вы уже однозначно сталкивались, например, когда разговаривали с разными голосовыми асистемтами (Салют, Сири, Алиса и тп), которые по сути своей являются комбинацией различных моделей, решающих различные задачи. Это говорит о том, что специалисты в данной сфере необходимы.

Но главное, мы поняли, что с вашим текущим набором знаний

уже можно с каким-то качеством начать решать практически любую задачу, найти подход практически к любой задаче. Это круто!

Но тем не менее при погружении в конкретную задачу нужно читать и разбираться в ней более глубоко и узко, потому что почти наверняка уже существуют более специфичные подходы к решению именно этой задачи, узнав про которые заранее, вы сэкономите свои время и деньги.

Учитесь искать, фильтровать и изучать информацию из разных источников, преимущественно в интернете. Это просто необходимо сегодня в нашей сфере, где все очень быстро меняется.

Гуглить не стыдно, стыдно не гуглить! Запомните эти слова.

5. Введение в RL

5.1 Введение

До этого перед нами стояли задачи, формулировка и решение которых опирается на работу с заранее подготовленными данными. Чаще всего это были пары «признаки - ответ». Иногда это были пары «картинка - метка класса», «текст на одном языке — текст на другом языке». Следующий тип задач принципиально отличается от рассмотренного ранее.

Ключевой элемент, который отличает обучение с подкреплением от остальных типов МО (с учителем и без учителя) состоит в том, что обучение с подкреплением базируется на концепции обучения через взаимодействие, т.е. модель обучается на основе взаимодействий со средой, чтобы довести до максимума функцию наград.

Из вышесказанного можно понять, что в RL участвует две «сущности»: «агент» (Agent) и «среда» (Environment), а сам процесс происходит так: модель (также называемая агентом) взаимодействует со своей средой и за счет этого генерирует последовательность взаимодействий, которые вместе называются эпизодом. Посредством таких взаимодействий агент накапливает ряд наград, определяемых средой. Награды могут быть положительными или отрицательными, а временами они не раскрываются агенту вплоть до конца эпизода.

Например, представим, что мы хотим научить компьютер играть в шахматы и выигрывать у игроков-людей. Метки (награды) для каждого отдельно взятого шахматного хода, сделанного компьютером, не известны вплоть до конца игры, потому что в течение самой игры мы не знаем, приведет ли конкретный ход в итоге к выигрышу или к проигрышу. Ответная реакция определяется лишь в конце игры. Вероятно, она будет положительной наградой, если компьютер выиграл партию, так как агент до-

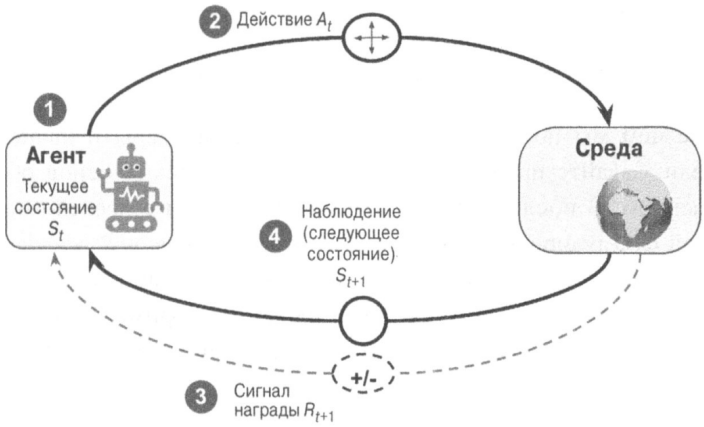


Рис. 5.1: схема взаимодействия Source картинка взята из книги Себастьяна Рашки и Вахида Мирджалили “Python и машинное обучение”

стиг общей желательной цели, и наоборот — отрицательной, если компьютер проиграл.

Во время процесса обучения агент обязан опробовать различные действия (провести исследование), чтобы он мог постепенно узнавать, каким действиям отдавать предпочтение (эксплуатировать знания) и выполнять более чаще для доведения до максимума общей накопленной награды.

В целом эксплуатация будет приводить к выбору действий с большей краткосрочной наградой, тогда как исследование потенциально может обеспечить более высокие награды в долгосрочной перспективе. Компромисс между исследованием и эксплуатацией изучался довольно широко, но до сих пор нет универсального выхода из указанного затруднительного положения при принятии решений.

5.2 Теоретические основы

5.2.1 Математическое ожидание, вероятность и условная вероятность

Вероятность

Есть такая штука, называется случайная величина, обозначим ее X, измерим эту величину и получим значение случайной величины x. Что все это значит?

Например, возьмем обычный игральный кубик. На его гранях условные обозначения 6 цифр — 1, 2, 3, 4, 5, 6. Цифра, выпадающая на кубике и есть случайная величина (X = {1, 2, 3, 4, 5, 6}), и она не предсказуема. Бросив кубик, мы получили какую-то цифру — значение случайной величины, которую показывает кубик (x $\in X$).

Что можно сказать про x?

Если провести эксперимент и бросить кубик очень большое количество раз (например, 10000), то посчитав отношение выпавших, предположим, 1 мы получим $\approx 1/6$, аналогично с другими цифрами на кубике. $1/6$ в этом случае называется вероятностью значения x p(x)

Допустим мы бросили кубик N раз (И снова, N — очень большое число, например, 10000), и из них выпало: n_1 единиц, n_2 двоек и т.д., тогда вероятность каждой величины соответственно можно оценить (интуитивно) $p(1) \approx \frac{n_1}{N}, p(2) \approx \frac{n_2}{N}$.... Если их все сложить, получим:

$$p(1) + p(2) + ... = \sum_{x \in X} p(x) = \frac{n_1 + n_2 + ...}{N} = \frac{\sum_{x \in X} n_x}{N} = 1$$

это очевидно, если понять, что каким бы не было значение x_i (цифра, выпавшая на кубике на i-том броске), оно обязательно принадлежит X ($x \in X$) а так как мы посчитали количество выпадений всех величин X, то мы насчитали ровно N (т.к. нам не может выпасть значение $x \notin X$ и x не может принимать несколько значений из X).

А что, если X бесконечно или несчетно (множество называется счетным, если все его элементы можно пронумеровать

натуральными числами)?

Специально для этого случайные величины разделяют на два типа:

1. Дискретная (прерывная) случайная величина —- принимает отдельно взятые, изолированные значения. Количество этих значений конечно либо бесконечно, но счётно (например цифра, выпадающая на кубике)
2. Непрерывная случайная величина —- принимает все числовые значения из некоторого конечного или бесконечного промежутка. (например, растояние на мишени от точки, в которую попал дротик, до центра в сантиметрах)

Если X — непрерывная случайная величина, то вероятность значения x p(x) = 0. Поэтому для таких случайных величин используют понятие «плотность вероятности». Но это выходит за рамки нашей темы.

Условная вероятность

Иногда можно встретить такую штуку — $p(A|B)$, что означает вероятность наблюдения значения A при условии наблюдения B.

Например, мы знаем, что на кубике выпало четное число, тогда какова вероятность выпадения 5? Ответ 0, так как 5 не четно. А теперь серьезно. Мы знаем, что на кубике выпало четное число, тогда какова вероятность выпадения 4? 1/3, так как на кубике всего 3 четных числа, и каждое из них выпадает с равной вероятностью. В общем случае для наглядности может быть удобным рисовать круги Эйлера

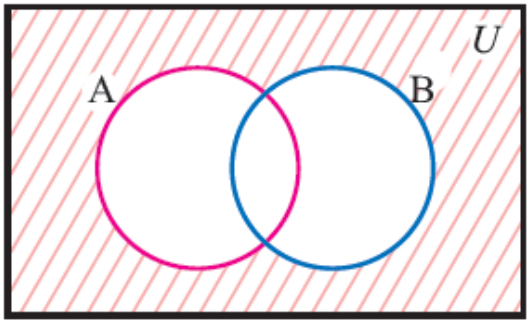


Рис. 5.2: Круги Эйлера

На этом рисунке часть плоскости, на которой расположен рисунок — наше X, красный круг обозначает наше множество A, синий круг множество B. p(A) на рисунке можно интерпретировать, как отношение площадей красного круга к площади U, а p(B) соответственно площадь синего к площади U. Тогда при рассмотрении вероятности наступления события A при условии B мы считаем, что $U = B$ (т.к. известно, что B наступило), т.е. $p(A|B) = \frac{S(AB)}{S(B)}$ — отношение площади пересечения к площади синего круга (множества B). Это кажется интуитивно понятно.

Математическое ожидание

Говоря простым языком, это среднее значение, которое принимает случайная величина при многократном повторении испытаний. Если проведем наш любимый эксперимент с N измерениями и обозначим i-ое измерение как a_i, то математическое ожидание можно записать как:

$$E[X] = \frac{a_0 + a_1 + \ldots + a_{N-1}}{N} = \frac{\sum_{i=0}^{N-1} a_i}{N}$$

Пусть $X = \{x_0, x_1, x_2, \ldots, x_m\}$, а наблюдение i-ой величины произошло n_i раз, то наше выражение можно переписать так:

$$E[X] = \frac{n_0 x_0 + n_1 x_1 + \ldots + n_m x_m}{N} = \frac{\sum_{i=0}^{m} n_i x_i}{N}$$

Воспользовавшись тем, что

$$p(x_i) = \frac{n_i}{N}$$

можно преобразовать мат. ожидание так:

$$E[X] = p(x_0)x_0 + p(x_1)x_1 + \ldots + p(x_m)x_m = \sum_{i=0}^{m} p(x_i)x_i$$

Небольшое упражнение: чему равно мат. ожидание цифры, выпавшей на шестигранном кубике после одного броска?

Решение:

$$E[X] = \sum_{i=0}^{m} p(x_i)x_i = 1 * p(1) + 2p(2) + 3p(3) + 4p(4) + 5p(5) + 6p(6)$$

Тогда очевидно получаем

$$E[X] = \frac{1 + 2 + 3 + 4 + 5 + 6}{6} = 3.5$$

5.2.2 Марковские процессы принятия решений

В общем случае тип задач, с которыми имеет дело обучение с подкреплением, обычно формулируется как марковские процессы принятия решений (Markov Decision Process - MDP)

В сценарии взаимодействий агент/среда при обучении с подкреплением, если мы обозначим начальное состояние агента как S_0, то взаимодействия между агентом и средой дадут в результате последовательность такого вида:

$$\{S_0, A_0, R_1\}, \{S_1, A_1, R_2\}, \{S_2, A_2, R_3\}...$$

Обратите внимание, что фигурные скобки служат только в качестве визуальной подсказки. Здесь S_t и A_t означают состояние и действие, предпринятое на временном шаге t. Посредством R_{t+1} обозначается награда, полученная от среды после выполнения действия A_t. Следует отметить, что S_t, R_{t+1} и A_t представляют собой независимые от времени переменные, которые берут значения из предварительно определенных конечных наборов, обозначаемых с помощью $s \in \hat{S}, r \in \hat{R}$ и $a \in \hat{A}$ соответственно. В марковском процессе принятия решений зависимые от времени переменные S_t и R_{t+1} имеют распределения вероятностей, которые зависят только от их значений на предыдущем временном шаге, t-1 (критерий Марковости). Распределение вероятностей для $S_{t+1} = s'$ и $R_{t+1} = r$ может быть записано как условная вероятность от предыдущего состояния S_t и предпринятого действия A_t:

$$p(s', r|s, a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

Такое распределение вероятностей полностью определяет динамику среды (или модели среды), поскольку на основе этого распределения могут быть рассчитаны вероятности всех переходов среды. Следовательно, динамика среды является центральным критерием для категоризации различных методов обучения с подкреплением. Типы методов обучения с подкреплением, которые требуют модели среды или пытаются изучить модель среды (т.е. выяснить динамику среды), называются методами на основе модели в противоположность методам без модели (об этом позже).

Динамику среды можно считать детерминированной, если определенные действия для заданных состояний приводят каждый раз в одно и то же состояние (например, как в шахматах - если вы сделали ход пешкой на клетку вперед, она не окажется магическим образом на 3 клетки назад), т.е. $p(s', r|s, a) \in \{0, 1\}$. Иначе в более общем случае среда будет обладать стохастическим(случайным) поведением.

Пример стахостической среды — настольная игра монопо-

лия,где каждый ход ты оказываешься с равной вероятностью на 1 клетку вперед или на 6, вне зависимости от своих действий

Чтобы понять такое стохастическое поведение, давайте рассмотрим вероятность наблюдения будущего состояния $S_{t+1} = s'$ при условии, что текущим состоянием является $S_t = s$ и предпринято действие $A_t = a$. Она обозначается как $p(s'|s, a) = P(S_{t+1} = s'|S_t = s, A_t = a)$. Ее можно вычислить как безусловную вероятность, выполнив сумму по всем возможным наградам:

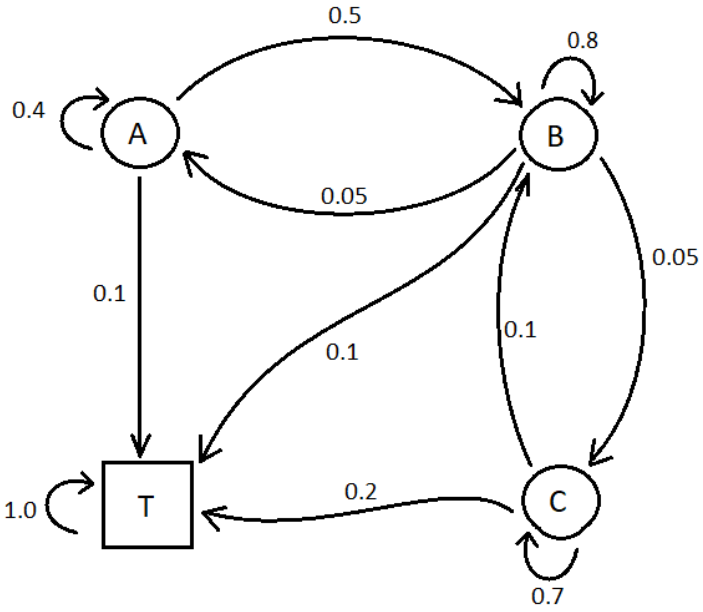
$$p(s'|s, a) = \sum_{r \in \hat{R}} p(s', r|s, a)$$

Такая вероятность называется вероятностью смены состояния. Исходя из вероятности смены состояния, если динамика среды детерминирована, то это значит, что когда агент предпринимает действие $A_t = a$ в состоянии $S_t = s$, переход в следующее состояние, $S_{t+1} = s'$, будет на 100% достоверным, т.е. $p(s'|s, a) = 1$.

Визуализация марковских процессов

Марковский процесс может быть представлен как ориентированный граф, узлы которого соответствуют разным состояниям среды (на самом деле, в RL узлы — это пары «состояние, действие«, но для примера рассмотрим более простой пример). Ребра графа (связи между узлами) представляют вероятности переходов между состояниями. Например, возьмем студента, который делает выбор среди трех состояний: (A) подготовка к сдаче экзамена дома, (B) игра в видеоигры дома или (C) подготовка к сдаче экзамена в библиотеке. Кроме того, имеется заключительное состояние (T) — пойти спать. Решения принимаются каждый час, и в течение этого конкретного часа студент остается в выбранном состоянии. Далее предположим, что находясь дома (состояние A), студент с вероятностью 50% будет переключаться с учебы на игру в видеоигры. С другой стороны, когда студент пребывает в состоянии B (игра в видеоигры) , есть относительно высокий шанс (80%) того , что он продолжит играть в видеоигру в течение последующих часов. Динамика поведения студента показана в

виде марковского процесса на рисунке ниже, включая циклический граф и таблицу переходов. Значения на ребрах графа представляют вероятности переходов в поведении студента, и они также приведены в таблице справа. Рассматривая строки в таблице, имейте в виду, что вероятности переходов из каждого состояния (узла) всегда в сумме дают 1.



	A	B	C	T
A	0.4	0.5	0.0	0.1
B	0.05	0.8	0.05	0.1
C	0.0	0.1	0.7	0.2
T	0.0	0.0	0.0	1.0

5.2.3 Эпизодические или продолжающиеся задачи

По мере взаимодействия агента со средой последовательность наблюдений или состояний формирует траекторию. Траектории бывают двух типов. Если траектория агента может быть разделена на части, так что каждая из них начинается в момент времени t = 0 и заканчивается в заключительном состоянии S_T (при t = T), тогда задача называется эпизодической. С другой стороны, если траектория длится бесконечно, не попадая в заключительное состояние, то задача называется продолжающейся.

Задача, связанная с агентом обучения для игры в шахматы, является эпизодической, в то время как робот-уборщик, который поддерживает чистоту в доме, обычно выполняет продолжающуюся задачу. Сейчас мы будем обсуждать только эпизодические задачи. В эпизодических задачах эпизод представляет собой последовательность или траекторию, которую агент проходит от начального состояния S_0 в заключительное состояние S_T:

$$S_0, A_0, R_1, S_1, A_1, R_2, ..., S_t, A_t, R_{t+1}..., S_{T-1}, A_{T-1}, R_T, S_T$$

В приведенном выше марковском процессе, который изображает задачу студента, готовящегося к сдаче экзамена, мы можем встретить эпизоды вроде следующих трех примеров:

- Эпизод 1: BBCCCCBAT—> экзамен сдан (финальная награда= +1)
- Эпизод 2: ABBBBBBBBBT—> экзамен провален (финальная награда= -1)
- Эпизод 3: BCCCCCT—> экзамен сдан (финальная награда= +1)

А критерий сдачи экзамена останется не известной динамикой среды.

5.2.4 Терминология

Давайте определим несколько дополнительных терминов, специфичных для обучения с подкреплением, которые мы будем применять далее.

Отдача

Так называемая отдача в момент времени t представляет собой накопленные награды, полученные на всем протяжении эпизода. Вспомните, что $R_{t+1} = r$ — это немедленная награда, полученная после выполнения действия A_t, в момент времени t; более поздними наградами являются R_{t+2}, R_{t+3} и т.д. Тогда отдача в момент времени t может быть рассчитана из немедленной награды и более поздних наград следующим образом:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... = \sum_{k=0} \gamma^k R_{t+k+1}$$

Здесь γ — коэффициент дисконтирования с диапазоном $(0, 1]$. Параметр γ указывает, насколько будущие награды «ценны» в текущий момент времени (t). Обратите внимание, что установка $\gamma = 0$ подразумевает, что нас не заботят будущие награды. В таком случае отдача будет равна немедленной награде, а более поздние награды после момента времени t + 1 игнорируются, и агент окажется «близоруким». С другой стороны, если $\gamma = 1$, тогда отдача будет невзвешенной суммой всех более поздних наград. Кроме того следует отметить, что уравнение для отдачи может быть выражено проще с использованием рекурсии:

$$G_t = R_{t+1} + \gamma G_{t+1} = r + \gamma G_{t+1}$$

Смысл в том, что отдача в момент времени t равна немедленной награде r плюс дисконтированная будущая отдача в момент времени t + 1. Это очень важное свойство, которое облегчает вычисления отдачи.

Политика

Политика, обычно обозначаемая как $\pi(a|s)$, представляет собой функцию, которая определяет действие, предпринимаемое следующим, и может быть либо детерминированной, либо стохастической (т.е. давать вероятность выбора следующего действия). Стохастическая политика имеет распределение вероятностей по

действиям, которые агент может предпринимать в заданном состоянии:

$$\pi(a|s) = P[A_t = a|S_t = s]$$

В процессе обучения политика может изменяться по мере того, как агент набирается опыта. Например, агент мог бы начать со случайной политики, где распределение вероятностей всех действий равномерно; в то же время агент благополучно научится оптимизировать свою политику в направлении достижения оптимальной политики. Оптимальной политикой $\pi_*(a|s)$ считается та, которая обеспечивает наивысшую отдачу.

Функция ценности

Функция ценности (value function), также называемая функцией ценности состояния (state-value function), измеряет доброкачественость каждого состояния, другими словами, насколько хорошим или плохим должно быть индивидуальное состояние. Обратите внимание, что критерий доброкачественности базируется на отдаче.

Теперь, основываясь на отдаче G_1, мы определяем функцию ценности состояния s как ожидаемую отдачу (среднюю отдачу по всем возможным эпизодам) после следования политике π:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0} \gamma^{k+1} R_{t+k+1}|S_t = s]$$

В реальной реализации мы обычно оцениваем функцию ценности с применением таблиц поиска, а потому у нас нет нужды повторно вычислять ее много раз. (Это аспект динамического программирования.) Скажем, на практике, когда мы оцениваем функцию ценности, используя такие табличные методы, то сохраняем ценности состояний в таблице, обозначаемой как V(s). В реализации на языке Python ею может быть список или массив NumPy, индексы которого соответствуют различным состояниям, либо словарь Python, чьи ключи отображают состояния на надлежащие ценности.

Кроме того, мы также можем определить ценность для каждой пары «состояние-действие», что называется функцией ценности действия (aciton-value function) и обозначается как $q_\pi(s, a)$. Функция ценности действия ссылается на ожидаемую отдачу G_t, когда агент находится в состоянии $S_t = s$ и предпринимает действие $A_t = a$. Расширив определение функции ценности состояния на пары «состояние-действие», мы получим:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0} \gamma^{k+1} R_{t+k+1}|S_t = s, A_t = a]$$

Подобно обозначению оптимальной политики как $\pi_*(a|s)$, $v_*(s)$ и $q_*(s, a)$ также обозначают оптимальные функции ценности состояния и ценности действия.

Оценка функции ценности является важным компонентом методов обучения с подкреплением. Мы раскроем различные способы вычисления и оценки функций ценности состояния и ценности действия позже в этой главе.

5.2.5 Уравнение Беллмана

Уравнение Беллмана — один из центральных элементов многих алгоритмов обучения с подкреплением. Уравнение Беллмана упрощает расчет функции ценности, так что вместо суммирования в течение множества временных шагов применяется рекурсия, которая похожа на рекурсию для вычисления отдачи.

Базируясь на рекурсивном уравнении для общей отдачи, $G_t = r + \gamma G_{t+1}$ мы можем переписать функцию ценности следующим образом:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[r+\gamma G_{t+1}|S_t = s] = r+\gamma * E_\pi[G_{t+1}|S_t = s]$$

Обратите внимание, что немедленная награда r вынесена из математического ожидания, поскольку она является константой с известной величиной в момент времени t.

Аналогично мы можем записать для функции ценности действия:

$$q_\pi(s) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[r + \gamma G_{t+1}|S_t = s, A_t = a]$$

Тогда, продолжая преобразования, получим

$$q_\pi(s) = r + \gamma E_\pi[G_{t+1}|S_t = s, A_t = a]$$

Для расчета математического ожидания мы можем использовать динамику среды, взяв сумму по всем вероятностям следующего состояния s' и соответствующих наград r (воспользуемся определением мат. ожидания):

$$v_\pi(s) = E_\pi[G_t|S_t = s] = \sum_{a\in\hat{A}} \pi(a|s) E_\pi[G_{t+1}|S_t = s] =$$

$$= \sum_{a\in\hat{A}} \pi(a|s) \sum_{s'\in\hat{S}, r\in\hat{R}} p(s', r|s, a)[r + \gamma E_\pi[G_{t+1}|S_{t+1} = s']]$$

Теперь мы видим, что математическое ожидание отдачи, $E_\pi[G_{t+1}|S_{t+1} = s']$, по существу является функцией ценности состояния, $v_\pi(s')$. Таким образом, мы можем записать $v_\pi(s)$ как функцию от $v_\pi(s')$:

$$v_\pi(s) = \sum_{a\in\hat{A}} \pi(a|s) \sum_{s'\in\hat{S}, r\in\hat{R}} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

если политика и модель среды детерминированы, то выражение принимает гораздо более приятный вид:

$$v_\pi(s) = r + \gamma v_\pi(s')$$

Результат называется уравнением Беллмана, которое связывает функцию ценности для состояния s с функцией ценности для последующего состояния s'. Оно значительно упрощает вычисление функции ценности, потому что устраняет итерационный цикл по оси времени.

Замечание: вообще, способ вывода этой формулы достаточно сложен. Если он для вас не очень понятен, то ничего страшного в этом нет. Самое главное — понять итог теоремы и что значит финальная формула.

5.3 Алгоритмы обучения с подкреплением

Мы рассмотрим несколько моделей, расположенных по мере возрастания их совершенности

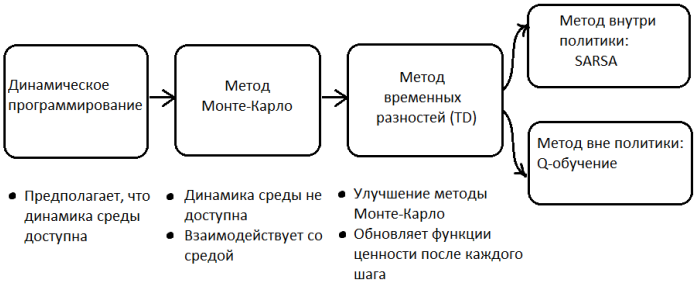


Рис. 5.3: Иерархия алгоритмов

5.3.1 Динамическое программирование

Динамическое программирование не является специальным алгоритмом для обучения с подкреплением и распространено в решении разных других задач в классическом программировании.

Вся идея динамического программирования заключается в том, что для вычисления какого-либо значения в каком-то состоянии мы используем значения, вычисленные для других состояний.

Пример задач на динамическое программирование:

- попробуем подсчитать количество способов дойти до клетки (m, n) на координатной сетке начиная с (0, 0) и с возможностью двигаться только вверх или вправо. Обозначим количество способов дойти до клетки (i, j) как f(i, j). Тогда

можно записать рекуррентную формулу (рекурсивную) f(i, j) = f(i - 1, j) + f(i, j - 1), зная что f(0, 0) = 1 и для значений i, j $<$ 0 функция не определена. заполним табличку (m, n) по правилу в (i, j) клетке значение f(i, j). Это и есть решение при помощи динамического программирования.

- также к ДП относятся такие алгоритмы как волновой поиск в графе, алгоритм Дейкстры(когда мы считаем значения сразу для всех узлов)

В этом разделе мы сосредоточим внимание на решении задач обучения с подкреплением при следующих допущениях:

- мы обладаем полным знанием динамики среды, т.е. вероятности всех переходов p(s', r |s, a) известны;
- состояние агента имеет марковское свойство(т.е. следующее состояние зависит только от текущего, не от предыдущих), а потому следующее действие и награда зависят только от текущего состояния и выбора действия, которое мы делаем в данный момент или на текущем временном шаге.

Мы должны подчеркнуть, что динамическое программирование не является практичным подходом к решению задач обучения с подкреплением.

Проблема, связанная с использованием динамического программирования, заключается в том, что оно предполагает наличие полного знания динамики среды, которое для большинства реальных приложений обычно необоснованно или нецелесообразно. Тем не менее с образовательной точки зрения динамическое программирование помогает представить обучение с подкреплением в простой манере и служит мотивом к применению более развитых и сложных алгоритмов.

В задачах, описанных в последующих подразделах, преследуются две цели.

1. Получить точную функцию ценности состояния, $v_\pi(s)$. Это также известно как задача прогнозирования и выполняется с помощью оценки политики.
2. Найти оптимальную функцию ценности, $v_*(s)$, что достигается посредством обобщенной итерации политики (generalized

policy iteration — GPI)

Основываясь на уравнении Беллмана, мы можем рассчитать функцию ценности для произвольной политики π с помощью динамического программирования, когда динамика среды известна. При расчете этой функции ценности мы можем адаптировать итеративное решение и начать с ценностей $v^0(s)$, инициализированных нулями для всех состояний. Затем на каждой итерации i + 1 мы обновляем ценности всех состояний на базе уравнения Беллмана, которое в свою очередь основано на ценностях состояний из предыдущей итерации i:

$$v_\pi^{i+1}(s) = \sum_a \pi(a|s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r'|s, a)[r + \gamma v_\pi^i(s')]$$

Можно показать, что с увеличением количества итераций до бесконечности $v_\pi^{i+1}(s)$ сходится в точную функцию ценности состояния $v_\pi(s)$. Также обратите внимание, что нам не нужно взаимодействовать со средой. Причина в том, что мы точно знаем динамику среды. В итоге мы можем задействовать эту информацию и легко оценить функцию ценности.

После расчета функции ценности возникает очевидный вопрос: какую пользу она способна нам принести, если наша политика попрежнему является случайной? Ответ заключается в том, что на самом деле мы можем использовать рассчитанную функцию ценности $v_\pi(s)$ для улучшения нашей политики, как будет показано далее.

Улучшение политики с использованием ожидаемой функции ценности

Теперь, когда мы рассчитали функцию ценности $v_\pi(s)$, следуя существующей политике π, имеет смысл применить $v_\pi(s)$ и улучшить существующую политику π. Это значит, что мы хотим найти новую политику π', которая для каждого состояния s, следующего π', выдавала бы более высокую или хотя бы равную ценность, чем при использовании существующей политики π.

Математически мы можем выразить такую цель для улучшен-

ной политики π' следующим образом:

$$v_{\pi'}(s) \geq v_{\pi}(s) \quad \forall s \in \hat{S}$$

Первым делом вспомните, что политика π определяет вероятность выбора каждого действия a, пока агент находится в состоянии s. Чтобы отыскать политику π', которая всегда имеет лучшую или равную ценность для каждого состояния, мы сначала рассчитываем функцию ценности действия, $q_{\pi}(s, a)$, для каждого состояния s и действия a, базируясь на вычисленной ценности состояния с применением функции ценности $v_{\pi}(s)$. Мы проходим по всем состояниям и для каждого состояния s сравниваем ценность следующего состояния s', куда произошел бы переход в случае выбора действия a.

После получения наивысшей ценности состояния за счет оценки всех пар «состояние-действие»; посредством $q_{\pi}(s, a)$ мы можем сравнить соответствующее действие с действием, выбранным текущей политикой. Если действие, предлагаемое текущей политикой, отличается от действия, предлагаемого функцией ценности действия (т.е. $\arg\max_a q_{\pi}(s, a)$), тогда мы можем обновить политику, переназначив вероятности действий с целью соответствия действию, которое дает наивысшую ценность действия, $q_{\pi}(s, a)$. Такая процедура называется алгоритмом улучшения политики.

Итерация политики

Итерация — каждое выполнение улучшения политики. Если многократно выполнять улучшение политики, то в какой-то момент мы получим, что $v_{\pi}(s) = v_{\pi'}(s) = v_{*}(s)$, что означает получение оптимальной политики $\pi_{*}(s)$. Это и называется итерацией политики.

Итерация ценности

Мы выяснили, что за счет повторения оценки политики (расчета $v_{\pi}(s)$) и $q_{\pi}(s, a)$) и улучшения политики (нахождения π', так что $v_{\pi'}(s) \geq v_{\pi}(s)\ \forall s \in \hat{S}$) можно достичь оптимальной политики.

Однако может оказаться эффективнее объединить две задачи — оценку и улучшение политики — в один шаг.

В следующем уравнении обновляется функция ценности для итерации i + 1 (обозначаемая как v^{i+1}) на основе действия, которое доводит до максимума взвешенную сумму ценности следующего состояния и его немедленной награды $(r + \gamma v^i(s'))$:

$$v'(s) = \max_a \sum_{s',r} p(s, r|s, a)[r + \gamma v^i(s')]$$

В данном случае обновленная ценность для $v^{i+1}(s)$ доводится до максимума путем выбора наилучшего действия из всех возможных действий, тогда как при оценке политики обновленная ценность использовала взвешенную сумму по всем действиям.

5.3.2 Метод Монте-Карло

Метод Монте-Карло в большинстве случаев еще проще, чем динамическое программирование, поэтому широко распространен для решения задач из разных областей: физики, химии и т.д.

Метод Монте-Карло — группа численных методов для изучения случайных процессов. Суть метода заключается в следующем: процесс описывается математической моделью с использованием генератора случайных величин, модель многократно обсчитывается, на основе полученных данных вычисляются вероятностные характеристики рассматриваемого процесса. Например, чтобы узнать методом Монте-Карло, какое в среднем будет расстояние между двумя случайными точками в круге, нужно взять координаты большого числа случайных пар точек в границах заданной окружности, для каждой пары вычислить расстояние, а потом для них посчитать среднее арифметическое.

Как было сказано ранее, динамическое программирование полагается на упрощенческое допущение о том, что динамика среды полностью известна. Отойдя от подхода динамического программирования, мы предположим, что не располагаем знаниями о динамике среды. Таким образом, мы не знаем вероятности

переходов между состояниями среды и взамен хотим, чтобы агент учился через взаимодействие со средой. В случае применения метода Монте-Карло процесс обучения основан на так называемом имитированном опыте (simulated experience).

При обучении с подкреплением на базе метода Монте-Карло мы определяем класс агента, следующего вероятностной политике π, на основе которой агент предпринимает действие на каждом шаге. Результатом будет имитированный эпизод.

Ранее мы определяли функцию ценности состояния, так что ценность состояния указывала ожидаемую отдачу от данного состояния. В динамическом программировании такое вычисление опиралось на знание динамики среды, т.е.$p(s', r|s, a)$.

Тем не менее, в дальнейшем мы будем разрабатывать алгоритмы, для которых динамика среды не требуется. Методы на основе Монте-Карло решают эту задачу, генерируя имитированные эпизоды, где агент взаимодействует со средой. Из таких имитированных эпизодов мы сможем рассчитать среднюю отдачу для каждого состояния, посещенного в заданном имитированном эпизоде.

Оценка функции ценности состояния с использованием метода Монте-Карло

После генерации набора эпизодов для каждого состояния s набор эпизодов, в котором все эпизоды проходят через состояние s, просматривается с целью вычисления ценности состояния s. Давайте предположим, что для получения ценности применяется таблица поиска $V(S_t = s)$, соответствующая функции ценности. Обновления в методе Монте-Карло для оценки функции ценности базируются на общей отдаче, которая получена в данном эпизоде, начиная с посещения состояния s в первый раз. Такой алгоритм называется прогнозированием ценности методом Монте-Карло первого посещения (first-visit Monte-Carlo).

Оценка функции ценности действия с использованием метода Монте-Карло

Когда динамика среды известна, мы можем без труда вывести функцию ценности действия из функции ценности состояния,

заглядывая на один шаг вперед для нахождения действия, которое дает максимальную ценность, как было показано в разделе «Динамическое программирование». Однако это неосуществимо, если мы не знаем динамику среды.

Чтобы решить проблему, мы можем расширить алгоритм для оценки прогноза ценности состояния методом Монте-Карло первого посещения. Скажем, мы можем рассчитать ожидаемую отдачу для каждой пары «состояние-действие» , применяя функцию ценности действия. Чтобы получить ожидаемую отдачу, мы принимаем во внимание посещения каждой пары «состояние-действие» (s, a), что означает пребывание в состоянии s и принятие действия a.

Тем не менее возникает проблема, так как некоторые действия могут никогда не выбираться, что приводит к неполному исследованию. Решить проблему можно несколькими способами. Простейший подход называется исследовательским началом (exploratory start), который предполагает, что каждая пара «состояние-действие» имеет ненулевую вероятность в начале обучения.

Еще один подход к решению проблемы нехватки исследования называется ϵ-жадной политикой ($\epsilon - greedy\ policy$).

В этом подходе на время обучения мы следуем не сгенерированной функцией ценности действия политике, а преобразованной(которая собственно и называется ϵ-жадной). Ее суть в том, что действия, которые не предпринимаются нашей политикой политикой, мы выбираем с вероятностью $\frac{\epsilon}{n}$, где n — количество действий; тогда рекомендованное действие выполняется с вероятностью $1 - \frac{n-1}{n}\epsilon$. Это позволяет нашей модели не зацикливаться на нашей политике и пройти большее количество состояний в меньшие сроки.

Нахождение оптимальной политики с использованием контроля Монте-Карло

Под контролем Монте-Карло (MC control) понимается процедура оптимизации для улучшения политики. Подобно подходу итерации политики из предыдущего раздела («Динамическое

программирование») мы можем многократно чередовать оценку политики и улучшение политики до тех пор, пока не достигнем оптимальной политики. Таким образом, начиная со случайной политики π, процесс чередования оценки политики и улучшения политики может быть проиллюстрирован так:

$$\pi_0 \xrightarrow{\text{Оценить}} q_{\pi_0} \xrightarrow{\text{Улучшить}} \pi_1 \xrightarrow{\text{Оценить}} q_{\pi_1} \xrightarrow{\text{Улучшить}} \pi_2 \ ... \xrightarrow{\text{Оценить}} q_* \xrightarrow{\text{Улучшить}}$$

Улучшение политики — расчет жадной политики из функции ценности действия

Имея функцию ценности действия $q(s, a)$, вот как можно сгенерировать жадную (детерминированную) политику:

$$\pi(s) = \underset{a}{argmax}\ q(s, a)$$

5.3.3 Q-обучение(Q-learning)

Q-обучение очень похоже на метод Монте-Карло, отличие только в том, что мы оцениваем $q_{\pi_n}(s, a)$ не только на основе текущей политики π, но и значений $q_{\pi_{n-1}}(s, a)$ на предыдущей итерации.

Стоит отметить, что в этом разделе мы будем использовать для обозначения ценности действия Q(s, a), а не q(s, a), чтобы подчеркнуть то, что Q не является функцией (в отличие от q), а является, по-сути, массивом со значениями (это сделано для того, чтобы избежать аппроксимации)

А именно, используется метод временных разностей(time differences - TD):

$$\delta_t^i = r_{t+1} + \gamma \underset{a}{max} Q^i(s_{t+1}, a) - Q^i(s_t, a_t)$$

Если просто изменять наше текущее значение $Q^i(s_t, a_t)$

$$Q^{i+1}(s_t, a_t) = Q^i(s_t, a_t) + \delta_t^i$$

то сходимости мы не добьемся, поэтому применяют коэффициент скорости обучения $\alpha \in (0; 1]$:

$$Q^{i+1}(s_t, a_t) = Q^i(s_t, a_t) + \alpha\delta_t^i$$

или

$$Q^{i+1}(s_t, a_t) = (1 - \alpha)Q^i(s_t, a_t) + \alpha \max_a Q^i(s_{t+1}, a)$$

И абсолютно аналогично оценке в методе Монте-Карло политика оценивается и здесь:

$$\pi(s) = \underset{a}{argmax}\, Q(s, a)$$

А может посмотрим на реализацию?

Для начала подключим все используемые библиотеки

```
import numpy as np
from random import random, randint
import matplotlib.pyplot as plt
```

Затем определим нашу среду. Какую задачу мы хотим решать?

Я предлагаю решить задачу, в которой агент путешествует по сетке, на которой расположены ямы (если агент падает в яму, он ломает ногу и не может продолжать путешествие), а его цель пройти эту пересеченную местность как можно быстрее, так как за ним гонится черт и тыкает его вилами каждый ход

```
class Environment:
    # В этом классе мы реализуем нашу среду (от англ.)
    def __init__(self):
        self.__state = np.array([0, 0])
        # запишем атрибут с двумя
        # лидирующими подчеркиваниями,
        # чтобы обозначить, что пользователь
        # (то есть мы при обращении к экземпляру из вне)
        # не имеем
        # доступа к нему (это называется инкапсуляция)
        self.actions = [0, 1, 2, 3]
        # это мы добавим на случай,
```

```
13          # чтобы не прописывать это каждый раз
14          # отдельно, внутри наших агентов
15          # Обозначим 1 - ямы, 0 - свободное пространство
16          self.space = np.array([[0, 0, 0, 0, 0, 1, 0, 0, 0],
17                                 [0, 0, 1, 0, 1, 0, 0, 0, 0],
18                                 [0, 1, 0, 1, 0, 0, 1, 1, 0],
19                                 [0, 0, 0, 0, 0, 0, 0, 0, 1]])
20          self.freedoms = np.array(self.space.shape).prod()
21          # количество степеней свободы, для того же,
22          # что и self.actions
23
24      @property
25      def pos(self):
26      # мы реализуем это для того, чтобы
27      # случайно не изменять это в том месте,
28      # где это не желательно
29          return self.__state.copy()
30
31      @property
32      def state(self):
33      # не важно, какой размерности делать
34      # таблицу Q-значений, но для
35      # простоты реализации сделаем ее одномерной и сделаем
36          return self.__state[0] * self.space.shape[1] + self.__state[1]
37
38      def do(self, action):
39          self.__state += np.array([(1 - action % 2) * (2 * (action//2) - 1),
40                                    (action % 2) *     (2 * (action//2) - 1)])
41          # В этой строке мы преобразуем номер
42          # действия в вектор смещения
43          # и прибавляем к состоянию
44
45          # Проверяем, не вышел ли агент
46          # за границы поля. чтобы строка в
```

```
47          # условии не была громоздской
48          # сделаем две вспомогательные переменные
49          is_valid_x = (self.__state[0] >= 0) and \
50                       (self.__state[0] < self.space.shape[0])
51          is_valid_y = (self.__state[1] >= 0) and \
52                       (self.__state[1] < self.space.shape[1])
53
54          if not is_valid_x or not is_valid_y or \
55             (self.space[tuple(self.__state)] == 1):
56              self.__state = np.array([0, 0])
57              return (-100, True)
58          # Проверяем не попал ли агент в яму
59          elif (self.__state[1] == self.space.shape[1] - 1):
60              self.__state = np.array([0, 0])
61              return (100, True)
62          else:
63              return (-1, False)
```

А теперь по порядку.

Мы обьявили свойство pos для того, чтобы совершенно случайно не изменить в каком-то месте кода состояние среды, а потом недоумевать, почему же все ломается.

Это является хорошей практикой.

В методе do(self, action) мы реализуем взаимодействие агента со средой. Так как мы решили хранить таблицу Q-values как массив Numpy, то действие агента — это не вектор смещения, а цифра, соответственно обозначающая перемещение вверх, влево, вниз, вправо.

Агент не должен выходить за границы поля, поэтому будем считать, что оно окружено ямами.

Собственно за проверку выхода из поля отвечают переменные is_valid_x и is_valid_y, соответственно обозначающие нахождение агента в одной горизонтали с полем и одной вертикали.

Определим вознаграждения:

- за перемещение в свободную клетку -1 (черт тыкает вилами);
- пересекли поле +100 (Ура, опасность позади! О, тут печенька валяется)));
- падение в яму -100 (О нет, я сломал ногу, как так...).

А теперь пришло время определить класс агента.

```
class Agent: # здесь реализуем класс нашего агента
    def __init__(self, env, gamma = 1, lr = 1):
        self.env = env # Запишем экземпляр среды
        self.qvalues = np.zeros((env.freedoms, len(env.actions)))
        self.gamma = gamma # коэффициент дисконтирования
        self.lr = lr

    def get_action(self):
    # получить действие, в соответствии с политикой
        actions = self.qvalues[self.env.state]
        return actions.argmax()

    def do_step(self):
    # Метод совершения агентом 1 (!) временного шага
        return self.env.do(self.env.actions[self.get_action()])

    def get_episode(self, max_steps = None):
    # Получим эпизод взаимодействия со средой
        actions = [self.get_action()]
        r = self.do_step()
        reward = r[0]
        while not r[1]:
            actions.append(self.get_action())
            r = self.do_step()
            reward += r[0]
            if max_steps != None and len(actions) > max_steps:
                return reward, actions
```

```
28          return reward, actions
29
30      def train(self, episodes = 100, greedy = 0.01,
31              testing = 10, max_steps = 150):
32      # Метод обучения агента
33          rewards = [] # будем запоминать награждения
34          steps = [] # количество шагов
35          tests_rewards = []
36          tests_steps = []
37          for i in range(episodes):
38              if (i + 1) % testing == 0:
39                  a, b = self.get_episode(max_steps)
40                  tests_rewards.append(a)
41                  tests_steps.append(len(b))
42                  # Проверка модели не на e-жадной политике
43
44              rewards.append(0)
45              steps.append(0)
46              while True:
47                  steps[-1] += 1
48                  if random() <= greedy:
49                      # Выбираем действие в соответсвии
50                      # с e-жадной политикой
51                      action = self.env.actions[randint(0,
52                                          len(self.env.actions) - 1)]
53                  else:
54                      action = self.get_action()
55                  state_action = (self.env.state, action)
56                  r = self.env.do(env.actions[action])
57
58                  td_target = r[0] +
59                  self.gamma * np.max(self.qvalues[self.env.state])
60                  # Итерация обучения
61                  td_error = td_target - self.qvalues[state_action]
```

```
                self.qvalues[state_action] += td_error * self.lr
                rewards[-1] += r[0]

                if r[1]:
                    # если состояние завершающее,
                    # то закончим эпизод
                    break
        return [[rewards, steps], [tests_rewards, tests_steps]]
```

Метод get_action(self) возвращает нам действие, которое мы должны предпринять согласно нашей жадной политике.

Метод get_episode(self, max_steps) реализует имитацию опыта, на которой мы впоследствии будем обучаться. Обратите внимание, что количество шагов в одном эпизоде ограничено для того, чтобы избежать возникновения бесконечного блуждания.

Метод train соответственно своему названию реализует обучение агента.

Здесь переменные rewards и steps хранят в себе сумму наград и количество шагов за эпизод обучения, а tr и ts — это их значения на всех эпизодах обучения для последующей визуализации.

Определим экземпляры классов среды и агента

```
env = Environment()
agent = Agent(env)
```

И приступим непосредственно к обучению

```
data = agent.train(100, 0.01, 5)
```

И долгожданная визуализация!

```
plots = {'train reward': data[0][0], 'train steps': data[0][1]}
fig, axes = plt.subplots(1, len(plots), figsize=(15, 5))
```

```
for t, (name, plot) in enumerate(plots.items()):
    axes[t].set(title = name)
    axes[t].plot(plot)
    axes[t].set_xlabel('episode')
    axes[t].set_ylabel('reward' if t == 0 else 'steps')
plt.show()
```

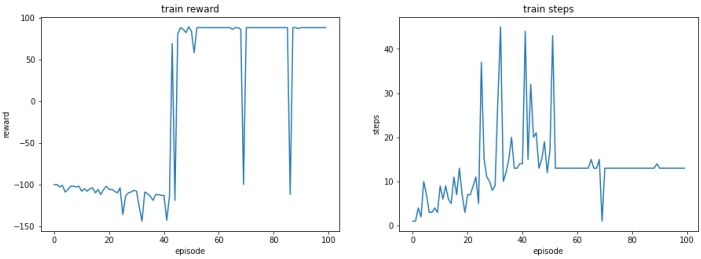


Рис. 5.4: Награды и количество шагов в процессе обучения

Из графика видно, что алгоритм обучается уже к 50-60-му эпизоду, и далее улучшения не происходит

5.3.4 SARSA

SARSA — это аббревиатура от state-action-reward-state-action, что буквально означает обучение на наборах $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Метод SARSA очень похож на Q-обучение с тем отличием, что Q-обучение — это метод вне политики, в то время как SARSA — внутри политики. Это значит, что имитация опыта в Q-обучении происходит следованием жадной политике, в то время как в SARSA для этого используется политика, оцененная на предыдущем шаге.

В SARSA формула обновления ценности действия с этой:

$$Q^{i+1}(s_t, a_t) = (1 - \alpha)Q^i(s_t, a_t) + \alpha \max_a Q^i(s_{t+1}, a)$$

заменяется на эту:

$$Q^{i+1}(s_t, a_t) = (1 - \alpha)Q^i(s_t, a_t) + \alpha Q^i(s_{t+1}, a_{t+1})$$

Все остальное происходит аналогично Q-обучению.

Если на пальцах, то Q-обучение предполагает выбор только наиболее выгодные действия на данный момент (соответственно и полученная политика будет ограничена этим), в то время как SARSA совершенствует нашу политику без этого допущения.

5.4 Так где же тут DL?

С применением глубоких нейронных сетей в обучении с подкреплением распространены два алгоритма: DQN (Deep Q-values Networks) и DDPG (Deep Deterministic Policy Gradient), но в рамках этой книги мы рассмотрим только DQN.

5.4.1 DQN

Не зря в названии есть Q-values, потому что устроено все абсолютно также, с одним лишь отличием: вместо таблицы используется нейронная сеть (она аппроксимирует таблицу)

Импортируем все необходимые пакеты.

```
1 import numpy as np
2 from random import *
3 import matplotlib.pyplot as plt
4 import torch
5 import torch.nn as nn
6 import gym
```

На этот раз мы будем рассматривать задачу не блуждания по перерытой местности, а балансировки стержня на тележке. Нужно удерживать стержень как можно дольше на этой тележке.

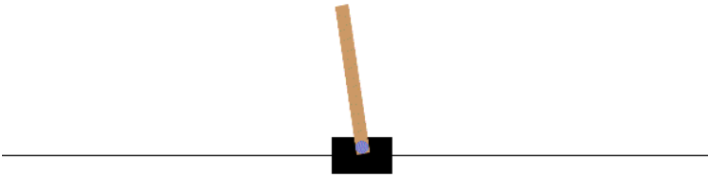


Рис. 5.5: картинка с оффициального сайта

Здесь всего три возможных действия: врубить мотор на полную назад или врубить мотор на полную вперед. Наблюдением являются:

1. Позиция тележки.
2. Скорость тележки.
3. Угол стержня.
4. Угловая скорость стержня.

В качестве награды авторы предлагают +1 каждый временной шаг.

Вообще, при решении задачи обучения с подкреплением среда фиксирована и функция награды для этой среды тоже заранее определена. И наша цель найти оптимальное решение именно для этой среды. Но часто есть соблазн поменять функцию награды на более понятную. Здесь нужно быть аккуратными.

Дело в том, что если вы поменяете функцию награды и найдете классное решение для новой функции, то это вовсе не значит, что с математической и практической точки зрения это будет

классное решение с точки зрения изначальной функции награды. Это тонкая проблема, о которой нужно помнить.

Один из «законных» способов изменения награды описан в следующей статье.

Если кратко, то авторы нашли целый класс таких изменений, которые основаны на методе потенциалов: $R' = R + (\gamma \cdot \Phi(new\_state) - \Phi(state))$, где Φ — потенциал, который зависит только от состояния. Для таких функций доказано, что решение оптимальное для новой награды, и для старой задачи также будет оптимальным.

А теперь реализуем класс нашей нейросети:

```
class AgentNN(nn.Module):
    def __init__(self, states_size, actions):
        super(AgentNN, self).__init__()
        self.fc1 = nn.Linear(states_size, hidden_neurons)
        self.tanh = nn.Tanh()
        self.fc2 = nn.Linear(hidden_neurons, actions)

    def forward(self, x):
        if type(x) != torch.Tensor:
            x = torch.Tensor(x.copy())

        out = self.fc1(x)
        out = self.tanh(out)
        out = self.fc2(out)

        return out
```

Вот наглядная схема архитектуры:

Рис. 5.6: Архитектура нейросети

Казалось бы, что пора перейти к классу нашего агента, но для начала определим класс памяти воспроизведения из которого мы будем извлекать обучающие выборки. Этот класс позволит организовать простое запоминание истории (нескольких эпизодов). Это поможет организовать обучение сети не на одном «прожитом» эпизоде, а на батче эпизодов, что способствует более стабильному обучению сети.

```
class ReplayMemory:
    def __init__(self, buf_size = 512):
        self.buf_size = 512
        self.dtype = [
            ('state', (np.float32, 4)),
            ('action', (np.float32, 1)),
            ('reward', [
                ('val', np.float32),
                ('is_done', bool)]
            ),
            ('new_state',(np.float32, 4))
```

```
        ]

        self.memory = np.array([], dtype = self.dtype)

    def __len__(self):
      return self.memory.shape[0]

    def get_sample(self):
        return np.random.choice(self.memory)

    def get_batch(self, batch_size = 32):
        return np.array(
            [self.get_sample() for i in range(batch_size)],
            dtype = self.dtype)

    def add(self, sars):
        if self.memory.shape[0] - len(sars) >= self.buf_size:
            self.memory = self.memory[:self.buf_size - len(sars)]
        self.memory = np.concatenate(
            [self.memory,
            np.array([sars], dtype = self.dtype)])
```

Вы могли заметить, количество наборов, которые можно хранить одновременно. Это сделано потому, что иногда для обучения может потребоваться 60000 эпизодов. А что, если каждый эпизод длится 2000 итераций? А состояние в вашей задаче описывается картинкой? В большинстве реальных задач это является огромной проблемой, поэтому задумываться о таких мелочах является хорошей практикой.

Так же в этом классе реализовано несколько методов:

- __len__ он реализован для того, чтобы каждый раз, когда мы хотим получить количество элементов, хранимых в памяти, не приходилось писать громоздкую, и часто не понятную или не естественную конструкцию self.memory.shape[0]

- get_sample метод, извлекающий случайный элемент из памяти
- get_batch метод, извлекающий из памяти batch_size случайных элементов (они могут повторятся).
- add метод, добавляющий новые данные в память.

Реализуем класс, который соберет вместе нашу нейросеть, память воспроизведения и принятие решений (политику).

```
class DQN:
    def __init__(
            self,
            states_size,
            actions,
            gamma = 0.99,
            memory_buffer = 512,
            lr = 0.01):
        self.actions = actions
        self.replaymem = ReplayMemory(memory_buffer)
        self.model = AgentNN(states_size, actions)
        self.loss = nn.MSELoss()
        self.optim = torch.optim.Adam(
            self.model.parameters(), lr=lr
        )
        self.gamma = gamma

    def __call__(self, arg):
        return self.model(arg)

    def get_values(self, state):
        return self.model(state)

    def get_action(self, state: np.ndarray):
        if type(state) != np.ndarray:
            state = np.array(state)
```

```
27          return self.model(state.reshape(1, -1)).argmax(
28              axis = 1
29          ).item()
30
31      def update(self, batch_size = 32):
32          if len(self.replaymem) < batch_size:
33              return
34          batch = self.replaymem.get_batch(batch_size)
35          state      = torch.from_numpy(batch['state'].copy())
36          action     = torch.from_numpy(batch['action'].copy())
37          reward     = torch.from_numpy(batch['reward']['val'].copy())
38          not_done   = torch.from_numpy(
39              1 - batch['reward']['is_done'].copy()
40          )
41          new_state = torch.from_numpy(batch['new_state'].copy())
42
43          mask = torch.BoolTensor([
44              [i == x for i in range(self.actions)]
45              for x in action])
46          # print(mask.dtype)
47
48          cur_var = self.model(new_state).max(axis = 1).values
49          td_target = reward + not_done * self.gamma * cur_var
50
51          loss = self.loss(self.model(state)[mask], td_target)
52          self.optim.zero_grad()
53          loss.backward()
54          self.optim.step()
55
56      def add_mem(self, sars):
57        self.replaymem.add(sars)
```

Ну, а теперь напишем нашего агента.

```
1
2  class Agent:
3
4      def __init__(self, env, gamma = 0.9, lr = 0.001):
5          self.env = env
6          self.qvalues = DQN(
7              env.observation_space.shape[0],
8              2,
9              gamma = gamma,
10             lr = lr
11         )
12         self.gamma = gamma
13
14     def get_action(self):
15         return self.qvalues.get_action(self.env.state)
16
17     def do_step(self):
18         return self.env.step(self.get_action())
19
20     def get_episode(self, max_steps = None):
21         actions = [self.get_action()]
22         # print(actions)
23         _, r, is_done, _ = self.do_step()
24         reward = r
25         while not is_done:
26             actions.append(self.get_action())
27             _, r, is_done, _ = self.do_step()
28             reward += r
29             if max_steps != None and len(actions) > max_steps:
30                 return reward, actions
31         self.env.reset()
32         return reward, actions
33
34     def train(
```

```
35              self,
36              episodes = 100,
37              greedy_start = 0.9,
38              greedy_end = 0.01,
39              testing = 10,
40              max_steps = 1000,
41              learn_freq = 5):
42          rewards = []
43          steps = []
44          test_rewards = []
45          test_steps = []
46          for i in range(episodes):
47              self.env.reset()
48              if (i+1) % testing == 0:
49                  reward, actions = self.get_episode(max_steps)
50                  test_rewards.append(reward)
51                  test_steps.append(len(actions))
52
53              rewards.append(0)
54              sam = 0 # step amounts in episode
55              while True:
56                  sars = []
57                  sam += 1
58                  epsilon = greedy_end + \
59                      (greedy_start - greedy_end) * \
60                      np.exp(-i / (episodes - 1))
61                  if random() <= epsilon:
62                      action = round(random())
63                  else:
64                      action = self.get_action()
65                  sars = [self.env.state, action]
66                  new_state, r, is_done, _ = self.env.step(action)
67                  sars += [(r, is_done), new_state]
68                  self.qvalues.add_mem(tuple(sars))
```

```
                rewards[-1] += r
                self.qvalues.update()
                if is_done:
                    break
            steps.append(sam)
        return [[rewards, steps], [test_rewards, test_steps]]

```

Этот класс практически аналогичен классу агента из предыдущей главы.

Теперь создадим переменные и научимся решать конкретную задачу.

```
hidden_neurons = 128

env = gym.make('CartPole-v1')
env.reset()
agent = Agent(env, lr = 0.001)

data = agent.train(2000, testing = 100, learn_freq = 64)

```

Построим график зависимости количества шагов от номера эпизода

```
plt.title('steps for a train')
plt.plot(data[0][1])
plt.xlabel('episode')
plt.ylabel('step')
plt.show()
```

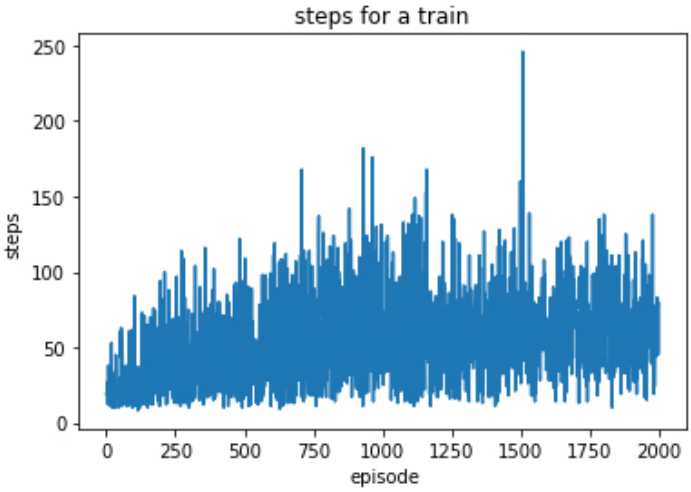


Рис. 5.7: Количество шагов на эпизодах в процессе обучения

По графику видна легкая тенденция к росту количества шагов примерно до 1000 эпизодов. Дальше модель уже не обучается, потому что был достигнут локальный максимум.