

# Машинное обучение

Никита Трофимов

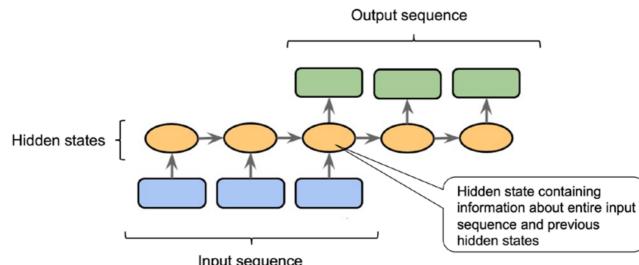
13 ноября 2022 г.

## 16 Transformers – Improving NLP with Attention Mechanisms

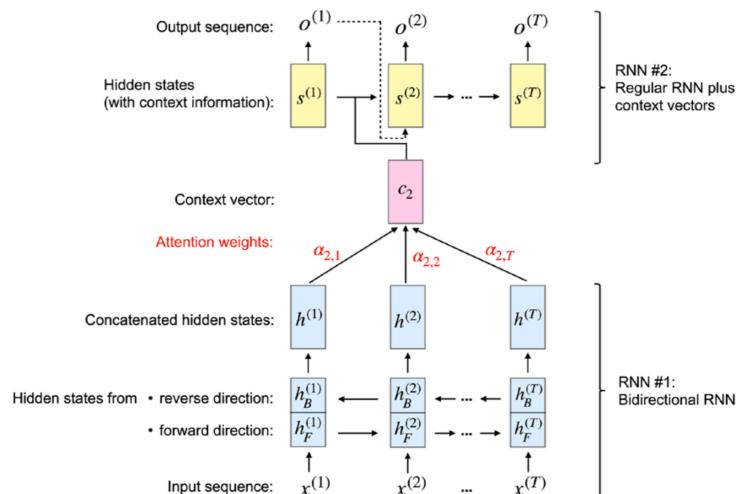
В предыдущей главе мы прошли RNNs и их применения в NLP в sentiment analysis проектах. Однако недавно появилась новая архитектура, которая показывает себя лучше, чем RNN seq2seq модели в некоторых NLP задачах. Это так называемая transformer архитектура. Эта архитектура перевернула мир в задачах NLP, её применяют в автоматическом переводе, моделировании фундаментальных свойств протеиновых последовательностей, также был создан AI, который помогает писать код. В этой главе мы пройдём *attention* и *self-attention* механизмы, а также то, как их применяют в transformer архитектуре. Далее мы посмотрим на самые крутые NLP модели, которые произошли от transformers, а также на посмотрим на large-scale language модель (так называемая BERT модель).

### 16.1 Adding an attention mechanism to RNNs

В этой части обсудим мотивацию для **attention mechanism**, который помогает моделям фокусироваться на конкретных частях входной последовательности больше, чем на других. Представим классическую RNN модель для seq2seq задачи, например, для перевода текста, она парсит весь текст перед тем, как выдать ответ. См. картинку:



Зачем мы парсим весь текст сразу? В разных языках порядок слов разный, например, в немецком и английском. Ограничение seq2seq подхода в том, что RNN пытается запомнить весь вход через один скрытый unit перед тем, как выдать перевод. Такое сжатие приводит к потере информации. Было бы плюсом иметь доступ ко всему входу в любой момент. Attention механизм даёт RNN доступ ко всем входным элементам. Однако такой доступ очень сложный, поэтому этот механизм даёт каждому элементу свой attention вес, он показывает насколько элемент релевантный в момент времени  $t$ . Есть входная последовательность  $x = (x^{(1)}, x^{(2)}, \dots, x^{(T)})$ , у каждого элемента есть свой вес. Чтобы лучше понять устройство RNN с attention механизмом, посмотрим на изображение:



Обсудим сначала первую часть – bidirectional RNN, которая выдаёт контекстные векторы  $c_i$ . О контекстных векторах можно думать как об усилении обычных входов  $x^{(i)}$ . RNN1 обрабатывает вход в обоих направлениях, forward ( $1 \dots T$ ) и backward ( $T \dots 1$ ). Это нужно, потому что элементы могут иметь зависимости как от предыдущих элементов, так и от последующих. Мы имеем два скрытых состояния  $h_F^{(i)}$  и  $h_B^{(i)}$ , затем мы их собираем в состояние  $h^{(i)}$ , его можно понимать как аннотацию к слову в тексте. На вход второй сети мы даём контекстный вектор  $c_i$ :

$$c_i = \sum_{j=1}^T a_{ij} h^{(j)}$$

Вторая сеть – это обычная RNN сеть, её скрытые слои в момент времени  $t$  обозначим как  $s^{(i)}$ , на вход она получает  $c_i$ .  $s_i$  зависит от  $s^{(i-1)}$ ,  $y^{(i-1)}$  и  $c^{(i)}$ , на выходе мы получаем  $o^{(i)}$ .  $y$  – это правильный перевод текста  $x$ , он доступен на момент тренировки. Во время предсказаний мы даём  $o^{(i-1)}$  вместо  $y^{(i-1)}$ . Attention вес  $a_{ij}$  – это нормализованная версия alignment score  $e_{ij}$ , где alignment score вычисляется насколько хорошо вход около позиции  $j$  соответствует выходу на позиции  $i$ . Вот формула:

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})}$$

Эта функция аналогична softmax функции, поэтому  $\sum_{j=1}^T a_{ij} = 1$ . Transformer архитектура также использует attention механизм, однако, в отличие от attention-based RNN, она полагается только на **self-attention** механизм и не включает в себя рекуррентный процесс как в RNN. Другими словами, transformer модель обрабатывает всю входную последовательность сразу вместо того, чтобы считывать и обрабатывать последовательность по одному элементу за раз.

## 16.2 Introducing the self-attention mechanism

Мы поняли, что attention механизм помогает RNNs запоминать контекст в длинных последовательностях. Мы сделаем архитектуру, полностью основанную на attention, но без рекуррентных частей RNN, она называется **transformer**. Сначала мы посмотрим на self-attention механизм, который используется в transformers, на самом деле это просто другой вариант attention механизма. Можно думать о attention как о операции, которая соединяет два модуля: кодер и декодер. Self-attention фокусируется только на входных данных и ищет зависимости только между ними, без соединения двух модулей. Есть входная последовательность  $x^{(1)}, \dots, x^{(T)}$ , а также выходная  $z^{(1)}, \dots, z^{(T)}$ .  $o$  – выход всей transformer модели, а  $z$  – это выход self-attention слоя,  $x^{(i)}, z^{(i)} \in \mathbb{R}^d$ , эти вектора содержат информацию о фичах, как в RNNs. Для seq2seq задачи, цель self-attention в том, чтобы смоделировать зависимости между текущим входным элементом и всеми другими элементами входа, поэтому у механизма есть 3 стадии. Сначала мы выводим веса важности на основе похожести текущего элемента со всеми остальными. Далее мы применяем softmax функцию. В конце мы используем эти веса, чтобы посчитать attention значение. Более формально выход self-attention  $z^{(i)}$  можно посчитать так:

$$z^{(i)} = \sum_{j=1}^T a_{ij} x^{(j)}$$

Мы можем думать о  $z^{(i)}$  как о context-aware embedding векторе для вектора  $x^{(i)}$ . Осталось понять, как считать похожесть между элементами. Сначала считаем скалярное произведение:

$$\omega_{ij} = x^{(i)T} x^{(j)}$$

Перед тем, как нормализовать значения  $\omega_{ij}$ , чтобы получить attention веса  $a_{ij}$ , посмотрим как считать  $\omega_{ij}$ . Вот и код:

```
import torch
sentence = torch.tensor([
    0, 7, 1, 2, 5, 6, 4, 3])
torch.manual_seed(123)
embed = torch.nn.Embedding(10, 16)
embedded_sentence = embed(sentence).detach()
print(embedded_sentence.shape) # [8, 16]

omega = torch.empty(8, 8)
for i, x_i in enumerate(embedded_sentence):
    for j, x_j in enumerate(embedded_sentence):
        omega[i, j] = torch.dot(x_i, x_j)
```

```
# более эффективно:
omega_mat = embedded_sentence.matmul(embedded_sentence.T)

# проверка матриц на равенство:
print(torch.allclose(omega_mat, omega)) # True
```

Применим softmax функцию:

```
import torch.nn.functional as F
attention_weights = F.softmax(omega, dim=1)
attention_weights.shape # [8, 8]
attention_weights.sum(dim=1) # все 1.0000
```

Посмотрим как считать контекстный вектор  $z^{(i)}$ , для примера  $i = 2$ , вот код:

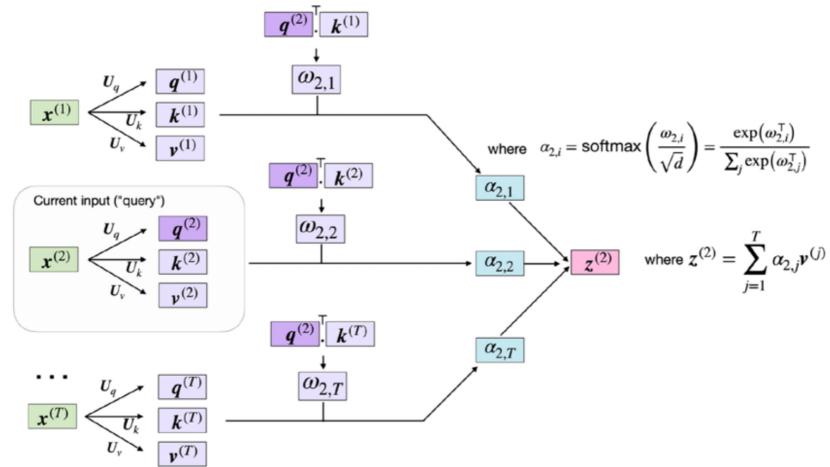
```
x_2 = embedded_sentence[1, :]
context_vec_2 = torch.zeros(x_2.shape)
for j in range(8):
    x_j = embedded_sentence[j, :]
    context_vec_2 += attention_weights[1, j] * x_j
print(context_vec_2)

# более эффективно:
context_vectors = torch.matmul(
    attention_weights, embedded_sentence
)
print(context_vectors.shape) # [8, 16]
print(torch.allclose(context_vec_2, context_vectors[1])) # True
```

Это была базовая версия self-attention, сейчас мы её модифицируем, используя обучаемые матрицы параметров, которые можно оптимизировать во время тренировки NNs. Это более продвинутый **scaled dot-product attention** механизм, который используется в transformer архитектуре. Мы представим три дополнительные весовые матрицы  $U_q$ ,  $U_k$  и  $U_v$ , их используют для проецирования входных данных в query, key, и value sequence elements:

- **Query sequence:**  $q^{(i)} = U_q x^{(i)}$  for  $i \in [1, T]$
- **Key sequence:**  $k^{(i)} = U_k x^{(i)}$  for  $i \in [1, T]$
- **Value sequence:**  $v^{(i)} = U_v x^{(i)}$  for  $i \in [1, T]$

Посмотрим на картинку, на которой показано, как из этих отдельных компонент посчитать context-aware embedding вектор для второго элемента:



Термины query, key и value, которые использовались в оригинальной статье transformer, вдохновлены информационно-поисковыми системами и базами данных. Например, если мы вводим запрос, он сопоставляется со значениями ключей, для которых извлекаются определенные значения. Здесь  $q^{(i)}$  и  $k^{(i)}$  имеют длину  $d_k$ , поэтому матрицы  $U_q$  и  $U_k$  имеют размер  $d_k \times d$ , а вот  $U_v$  имеет размер  $d_v \times d$  ( $d$  – размерность каждого вектора  $x^{(i)}$ ). Для простоты можно сделать  $d_k = d_v = d$ . Посмотрим на код для лучшего понимания:

```
torch.manual_seed(123)
d = embedded_sentence.shape[1]
U_query = torch.rand(d, d)
U_key = torch.rand(d, d)
U_value = torch.rand(d, d)
```

```

x_2 = embedded_sentence[1]
query_2 = U_query.matmul(x_2)
key_2 = U_key.matmul(x_2)
value_2 = U_value.matmul(x_2)

keys = U_key.matmul(embedded_sentence.T).T
values = U_value.matmul(embedded_sentence.T).T

```

```

# проверка:
assert torch.allclose(key_2, keys[1])
assert torch.allclose(value_2, values[1])

```

Посчитаем  $\omega$  и нормализуем их:

```

omega_23 = query_2.dot(keys[2])
print(omega_23) # tensor(14.3667)

omega_2 = query_2.matmul(keys.T)
print(omega_2.shape) # torch.Size([8])

```

Дальше мы нормализуем эти веса:

$$a_{ij} = \text{softmax}\left(\frac{\omega_{ij}}{\sqrt{m}}\right)$$

Обычно выбирают  $m = d_k$ , в таком случае длина весовых векторов будет примерно в том же диапазоне. А вот и код для этого:

```

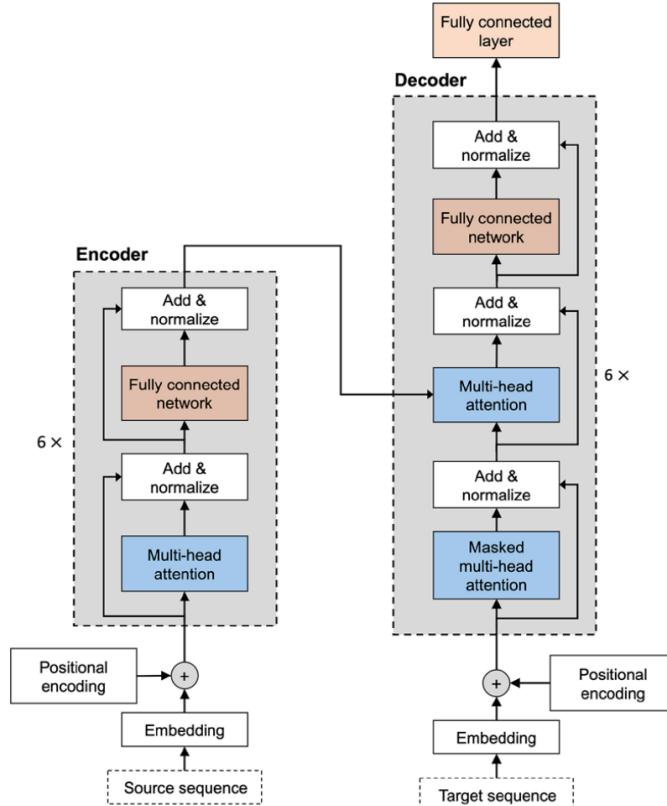
attention_weights_2 = F.softmax(omega_2 / d**0.5, dim=0)
print(attention_weights_2.shape) # torch.Size([8])

context_vector_2 = attention_weights_2.matmul(values)
print(context_vector_2)
# tensor([-1.2226, -3.4387, ..., -1.9917, -3.3499])

```

### 16.3 Introducing the original transformer architecture

Изначально эта архитектура была основана на attention механизме, это нужно было для улучшения генерации текстов с длинными предложениями, однако потом поняли, что лучше удалить рекуррентные слои, так и изобрели transformer архитектуру. Self-attention помогает ей лучше понимать контекст в длинных зависимостях. Изначально её сделали для перевода, но конечно её можно расширить для других задач с текстом. Из неё были выведены популярные языковые модели, такие как BERT и GPT, которые мы обсудим позже. На этом изображении оригинальная transformer архитектура:



На этой схеме два главных блока: кодер и декодер. Кодер получает входную последовательность и кодирует embeddings, используя multi-head self-attention модуль. Декодер принимает обработанный вход и выдаёт выходную последовательность, используя *masked* форму self-attention механизма. Поговорим про кодирование context embeddings через multi-head attention. Конечная цель кодера в том, чтобы промаппить  $X = (x^{(1)}, x^{(2)}, \dots, x^{(T)})$  непрерывному представлению  $Z = (z^{(1)}, z^{(2)}, \dots, z^{(T)})$ , которое дальше идёт в декодер. Кодер состоит из 6 одинаковых слоёв, 6 – это не магическое число, а гиперпараметр. Внутри каждого такого слоя есть два подслоя: один считает multi-head self-attention, а другой – это fc слой. Теперь приступим к разговору о multi-head self-attention, это простая модификация scaled dot-product attention механизма. У нас были 3 матрицы, здесь мы можем думать о множестве этих 3х матриц как о attention head, мы имеем  $h$  таких голов. Если вы помните, CNNs могут иметь несколько ядер, здесь такая же идея. Пусть уже вход прошёл через embedding слой размера  $d$ , тогда входная матрица имеет размер  $T \times d$ . Мы создаём  $h$  наборов query, key и value обучающихся матриц  $(U_{q_1}, U_{k_1}, U_{v_1}), \dots, (U_{q_h}, U_{k_h}, U_{v_h})$ . Первые две матрицы мы умножаем на  $x^{(i)}$ , поэтому  $U_{q_j}$  и  $U_{k_j}$  имеют размер  $d_k \times d$ , а  $U_{v_j} - d_v \times d$ . На практике часто выбирают  $d_k = d_v = m$  для простоты. Пару строчек кода:

```
# было:
torch.manual_seed(123)
d = embedded_sentence.shape[1]
one_U_query = torch.rand(d, d)

# стало:
h=8
multihead_U_query = torch.rand(h, d, d)
multihead_U_key = torch.rand(h, d, d)
multihead_U_value = torch.rand(h, d, d)
```

На практике вместо того, чтобы иметь отдельную матрицу на каждую attention head, в transformer используют одну матрицу для всех голов. Затем головы делят на регионы в этой матрице, которые получают через булевые маски. Это сильно ускоряет вычисления, т.к. несколько умножений матриц теперь делаются за одно умножение. Вычисления делаются так же, как и в прошлой части, например:

$$q_j^{(i)} = U_{q_j} x^{(i)}, j \in \{1, \dots, h\}$$

А вот и пример кода:

```
multihead_query_2 = multihead_U_query.matmul(x_2)
print(multihead_query_2.shape) # [8, 16]
multihead_key_2 = multihead_U_key.matmul(x_2)
multihead_value_2 = multihead_U_value.matmul(x_2)
# 3яя голова для ключа второго примера
print(multihead_key_2[2])
```

Однако нам нужно это вычислить для всех примеров, а не только для второго. Самый простой способ – это расширить исходный массив данных с помощью метода `.repeat()`:

```
# этому методу нужно передать сколько раз повторять
# какую размерность, в этом случае массив просто
# будет скопирован 8 раз
stacked_inputs = embedded_sentence.T.repeat(8, 1, 1)
print(stacked_inputs.shape) # [8, 16, 8]
```

Теперь можно сделать умножение матриц по батчам, используя метод `.bmm()`, в первой размерности будут головы, во второй – размер embedding, в последней – кол-во слов. Так же надо будет транспонировать массив по 2 и 3 размерностям, это более интуитивно. См. код:

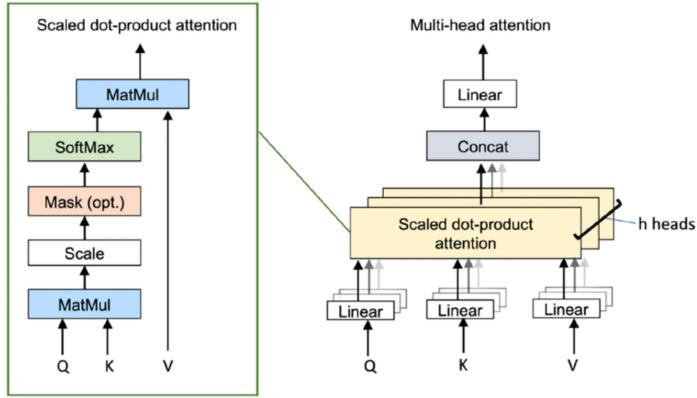
```
multihead_keys = torch.bmm(multihead_U_key, stacked_inputs)
print(multihead_keys.shape) # [8, 16, 8]
multihead_keys = multihead_keys.permute(0, 2, 1)
print(multihead_keys.shape) # [8, 8, 16]
print(multihead_keys[2, 1])

multihead_values = torch.matmul(
    multihead_U_value, stacked_inputs)
multihead_values = multihead_values.permute(0, 2, 1)
```

Зачем нужно было транспонировать? Нам нужно было первым параметром передать массив батчей, а вторым уже матрицу, на которую умножаем. Теперь надо посчитать контекстные вектора для второго примера для каждой из 8 голов. Для простоты пропустим эти шаги и представим  $z^{(2)}$  случайными числами:

```
multihead_z_2 = torch.rand(8, 16)
```

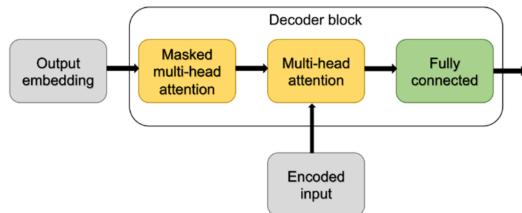
Теперь мы конкатенируем эти вектора в один большой вектор длины  $d_v \times h$  и используем fc слой, чтобы сделать опять вектор длины  $d_v$ . Этот процесс изображён на рисунке:



В коде это можно сделать так:

```
linear = torch.nn.Linear(8*16, 16)
context_vector_2 = linear(multihead_z_2.flatten())
print(context_vector_2.shape) # [16]
```

Получилось, что multi-head – это просто параллельное выполнение нескольких scaled dot-product и их комбинирование. Это очень похоже на несколько ядер в CNNs, которые дают разные каналы. Обсудим обучение языковой модели, а именно декодер и masked multi-head attention. Декодер также содержит повторяющиеся слои. Помимо двух подслой, которые мы уже обсудили, здесь есть masked multi-head attention подслой. Masked attention – это вариация исходного механизма внимания, только здесь на вход подаётся лишь часть входных данных, которая соответствует некоторой маске. Например, во время тренировки переводчика, на позиции  $i$  мы даем модели только правильные переводы с позиций  $1, \dots, i-1$  (чтобы избежать жульничества модели). Вот идея блока декодера:

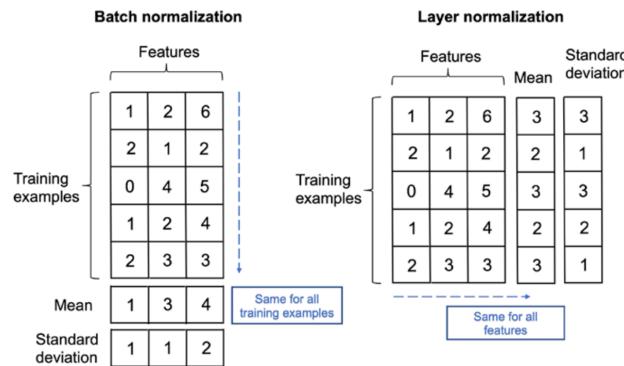


Output embeddings – это предыдущие выходные слова, точнее их кодирование. Сравнивания кодер и декодер, главное отличие в элементах, на которые они обращают внимание. Кодер для каждого слова считает внимание по всем словам, это двойной проход. Декодер конечно получает все эти данные, но вывод он генерирует на основе рассмотренных ранее выводов, это односторонний проход. Осталось поговорить про детали реализации, а именно про positional encodings и layer normalization. Начнём с **positional encodings**, оно помогает улавливать информацию о порядке входных последовательностей, это очень важная часть для transformers, т.к. все слои, которые мы видели на схеме инвариантны к перестановке. Это значит, что без positional encoding, порядок слов игнорируется и не имеет никакого значения для кодирований внимания. Transformers позволяют одинаковым словам на разных позициях иметь немного разные encodings с помощью добавления вектора маленьких значений к входным embeddings. Чаще всего применяют синусоидальное кодирование:

$$PE_{(i,2k)} = \sin(pos/10000^{2k/d_{\text{model}}})$$

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

Здесь  $i$  – это позиция слова,  $k$  – это длина encoding вектора, мы выбираем  $k$  равным размеру входного кодирования, чтобы их можно было потом складывать. Синусоидальные функции используются, чтобы избежать чрезмерно больших значений. В общем, есть два типа positional encodings, который мы уже рассмотрели – это *absolute*, ещё есть *relative*. Первый запоминает абсолютные значения позиций слов и он чувствителен к сдвигам слов в предложении, второй же только поддерживает относительные позиции между словами и его значение инвариантно относительно сдвига. Теперь посмотрим на layer normalization механизм. В то время как batch normalization (обсудим в главе 17) популярен в контексте computer vision, layer normalization используют в контексте NLP, где длина последовательности может меняться. Обычно сложные модели обучаются на больших данных, поэтому посчитать значение по всем примерам довольно сложно, в связи с этим используют этот вид нормализации. Посмотрим на картинку, чтобы понять разницу между layer и batch normalization:

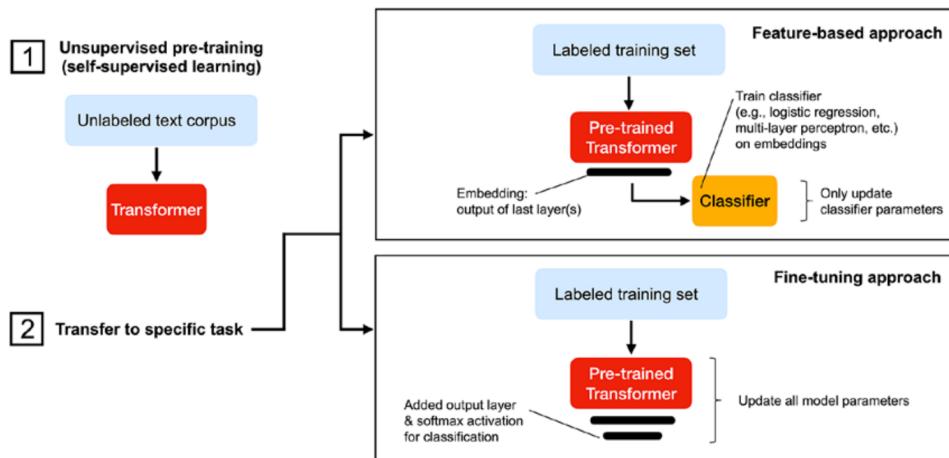


## 16.4 Building large-scale language models by leveraging unlabeled data

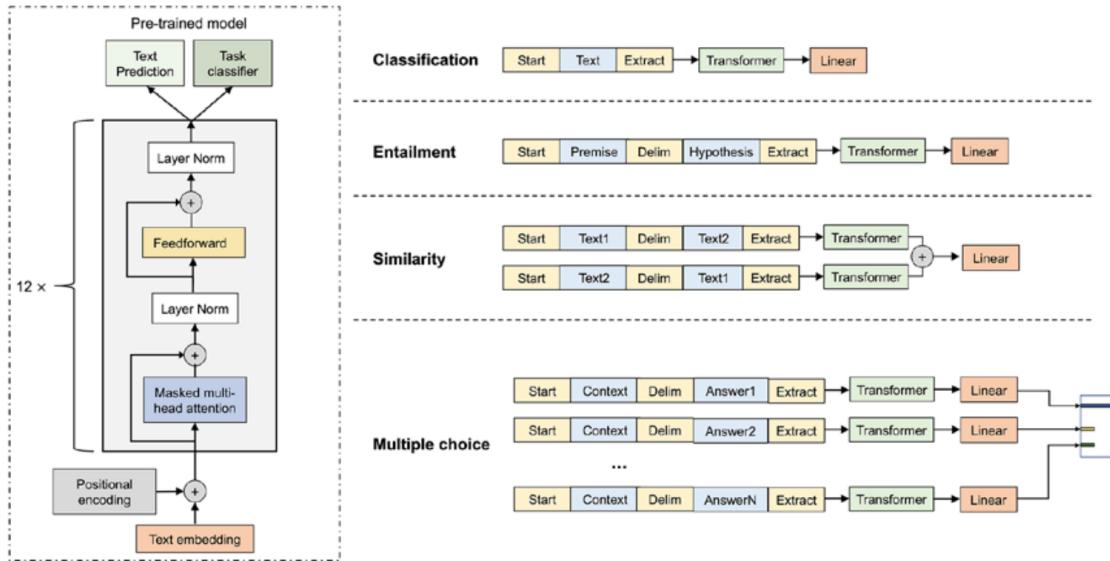
В этой части мы будем обсуждать популярные large-scale transformer модели, которые сделали из обычных transformer моделей. Общая часть этих моделей в том, что они предварительно обучаются на очень больших непомеченных датасетах. Сначала мы обсудим общую процедуру тренировки, а затем уже посмотрим на Generative Pre-trained Transformer (**GPT**), Bidirectional Encoder Representations from Transformers (**BERT**) и Bidirectional and Auto-Regressive Transformers (**BART**). Начнём с предварительного обучения. Для перевода текстов нужен объёмный помеченный датасет, его очень тяжело получить. Интересный вопрос в том, как можно использовать непомеченные данные (книги, сайты) для улучшения качества модели. Секрет заключается в процессе, который называется **self-supervised learning**, он генерирует метки для обычного текста. Имея большой непомеченный текст, мы обучаем модель предсказывать следующее слово. Этот процесс также иногда называют **unsupervised pre-training**, он является очень важным для качественной transformer модели. Он **unsupervised**, потому что мы используем непомеченные данные, но мы, например, генерируем следующее слово, а это уже **supervised** процесс. Пусть у нас есть предложение из  $n$  слов, тогда pre-training процедура состоит из 3х шагов:

1. В момент времени 1 дадим модели правильные слова  $1, \dots, i - 1$ .
2. Предскажем слово на позиции  $i$  и сравним с правильным словом.
3. Обновим модель и время  $i := i + 1$ , перейдём к шагу 1 пока есть слова.

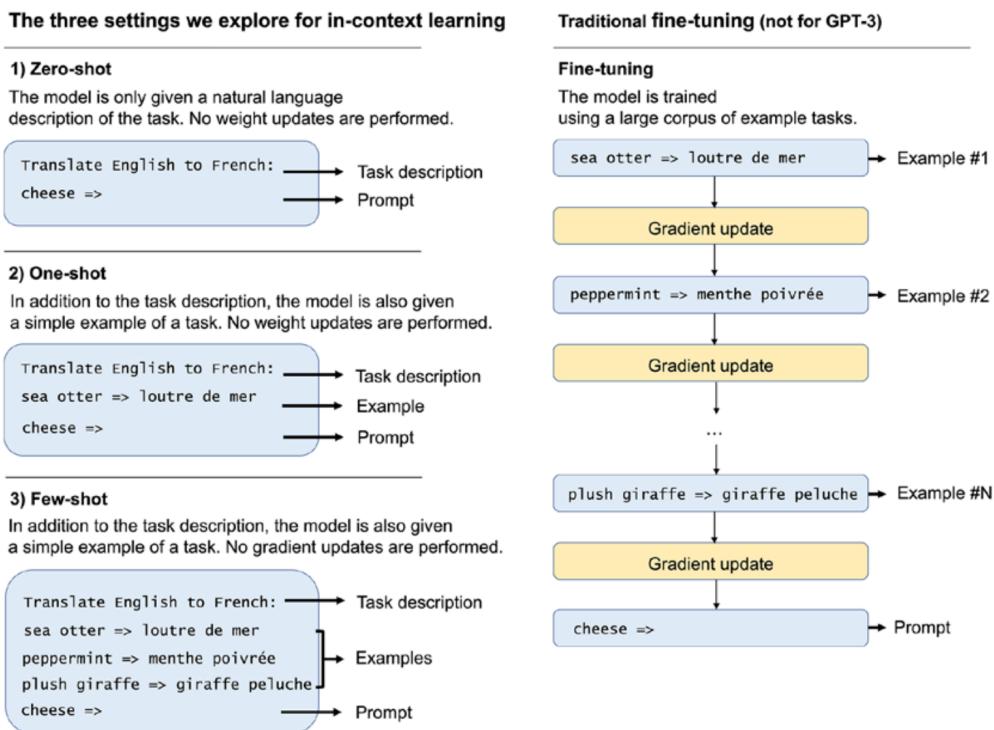
Сейчас есть много разных pre-training техник, например, этот можно рассматривать как одноправленный подход. Полная процедура тренировки для transformer модели состоит из 2х шагов: pre-training на большом непомеченному датасете, далее уже тренировка (или fine-tuning) на специальном помеченному датасете. На первом шаге мы учим общую языковую модель, а уже на втором шаге мы кастомизируем задачу через помеченный датасет. После первого шага есть две стратегии для адаптации модели на специфичную задачу: **feature-based approach** и **fine-tuning approach**. После pre-training мы получаем какие-то скрытые состояния с последнего слоя модели. В первом подходе мы используем эти состояния как дополнительные фичи для помеченного датасета. Но в таком случае надо уметь искать фичи из pre-trained модели. Для этого придумали модель **ELMo** (Embeddings from Language Models). ELMo – это двунаправленная языковая модель, которая выбирает случайно 15% слов из всех слов, а задача модели в предсказании пропусков. То есть о этой технике можно думать как о feature extraction технике, основанной на модели, например, PCA. Второй подход обновляет веса в обычной supervised манере через backpropagation, мы добавляем ещё один fc слой к pre-trained модели. Этот подход реализует популярная модель **BERT**. Идея:



Generative Pre-trained Transformer (**GPT**) – это популярная серия моделей для генерации текста, разработанная OpenAI. Последняя модель GPT-3 даёт настолько хорошие результаты, что их тяжело отличить от человеческого текста. Посмотрим на GPT-1, это transformer, который состоит из декодера (но без кодера) и дополнительного слоя, который добавляют позже для fine-tuning. См. на картинку:



Во время pre-training GPT-1 использует декодер, модель полагается только на предыдущие слова. GPT-1 использует односторонний подход в отличие от BERT (двусторонний), т.к. она больше фокусируется на генерации текста, чем на классификации. GPT-1 хорошо показывает себя на zero-shot задачах – это особая группа задач, где модель во время тестов должна классифицировать примеры между классами, которые она не видела во время тренировки. GPT-2 способна различать задачи между разными типами ввода, ей надо дать лишь контекст, например, [переведи на испанский](#). GPT-3 смешает свой фокус на one-shot и few-shot обучение. Посмотрим на разницу между zero-shot, one-shot, few-shot и fine-tuning процедурами. См. изображение:



Модель GPT-3 очень похожа на GPT-2 за исключением того, что сильно увеличено кол-во параметров, а также использован sparse transformer. В оригинальном (dense) attention механизме каждый пример сравнивается с каждым, это  $O(n^2)$ . **Sparse attention** улучшает эффективность, сравнивая только с подмножеством примеров, обычно его размер пропорционален  $n^{1/p}$ . Посмотрим как использовать GPT-2 для генерации текста через **transformers**, эта библиотека даёт возможность

использовать разные transformer модели для pre-training и fine-tuning. Установим библиотеку и импортируем уже pre-trained GPT модель, которая может генерировать новый текст:

```
! pip install transformers
from transformers import pipeline, set_seed
generator = pipeline('text-generation', model='gpt2')
set_seed(123)
generator("The sun is shining outside today",
         max_length=20, num_return_sequences=3)
```

Как уже было упомянуто, мы можем использовать transformer модель для генерации новых фич для обучения других моделей. См. код:

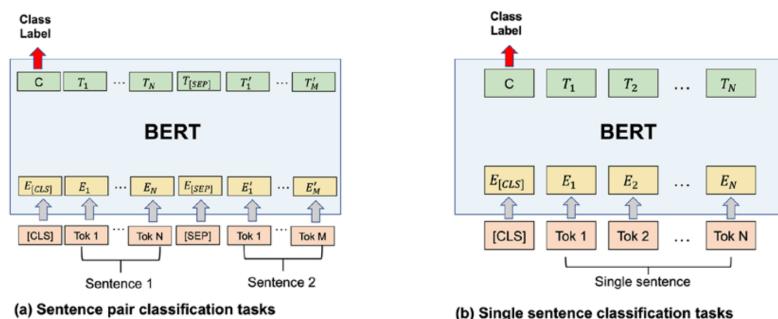
```
from transformers import GPT2Tokenizer
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
text = "Let us encode this sentence"
# return tensors – тип тензора, например, pt – PyTorch
encoded_input = tokenizer(text, return_tensors='pt')
print(encoded_input)
# {'input_ids': tensor([[ 5756,     514, 37773,    428,   6827]]),
#  'attention_mask': tensor([[1, 1, 1, 1, 1]])}

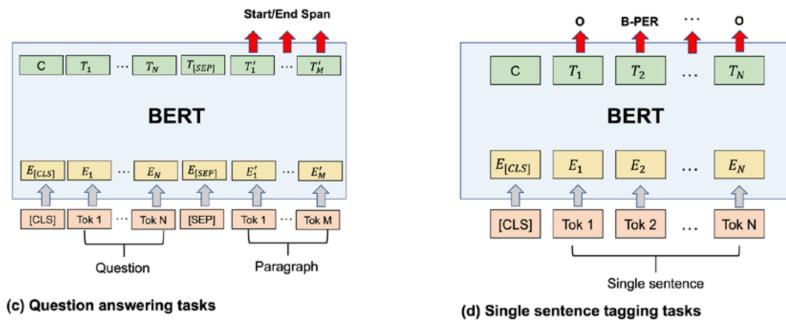
from transformers import GPT2Model
model = GPT2Model.from_pretrained('gpt2')
output = model(**encoded_input)
print(f"GPT-2-based feature encoding size: "
      f"{output['last_hidden_state'].shape}") # [1, 5, 768]
```

Здесь первая размерность – это размера батча, вторая – это длина предложения, третья – длина кодирования. Теперь можно применить это вместо bag-of-words (глава 8) модели ко всему датасету и обучить на нём классификатор. Теперь поговорим про bidirectional pre-training с помощью BERT. **BERT** – это Bidirectional Encoder Representations from Transformers. Как уже понятно из имени, BERT имеет в своей структуре кодер и использует двунаправленную процедуру тренировки (более точно она вообще не имеет направления, т.к. читает все элементы за раз). В transformer кодере токенизация – это сумма positional encodings и token embeddings. В BERT кодере ещё есть segment embedding, который показывает к какому сегменту принадлежит токен. То есть в подготовке входа для BERT кодера три шага: обычная токенизация и embedding, деление на отрезки и позиционный embedding. Зачем нам нужна эта дополнительная информация об отрезках? У BERT есть специальная pre-training задача – *next-sentence prediction*. В этой задаче каждый пример состоит из двух предложений, поэтому и нужно это специальное деление. Pre-training включает в себя две unsupervised задачи: masked language modeling и next-sentence prediction. В masked language model (**MLM**) токены случайно заменяются на *mask tokens* ([MASK]), а модель должна предсказывать эти скрытые слова. Однако в fine-tuning могут быть неожиданные результаты, т.к. в обычном тексте маски не встречаются. Пусть мы выбрали 15% случайных масок, с этими словами делают 3 пункта:

1. Не меняем слово 10% времени.
2. Заменяем изначальный токен слова на случайное слово 10% времени.
3. Заменяем изначальный токен слова на маску 80% времени.

Эта процедура имеет и другие плюсы. С одной стороны мы запоминаем оригинальный токен, с другой стороны мы запоминаем только контекст. Случайная замена помогает избежать ленивости модели – это когда модель возвращает ровно то, что там было. В предсказании следующего предложения модель получает два предложения A и B в формате [CLS]A[SEP]B[SEP]. [CLS] – токен классификации, заполнитель для вывода декодера, а также маркер начала предложения. [SEP] – маркер конца предложения. Задача модели сказать, является ли B следующим предложением для A. BERT претренирован на этих двух задачах, предсказание следующего предложения и [MASK] в одно время. Цель BERT – минимизировать общую loss функцию для обеих задач. В итоге BERT можно использовать для классификации пар предложений или просто предложений, ответов на вопросы и простановку заголовков. Использование BERT для разных задач:





Последним пунктом в этой части пройдём Bidirectional and Auto-Regressive Transformer (**BART**). Мы уже говорили, что GPT использует декодер, а BERT – кодер, поэтому GPT хорошо генерирует текст, а BERT хорошо классифицирует. BART можно понимать как обобщение двух этих моделей, она хорошо работает на обеих этих задачах. Причина в том, что в BART используется двухсторонний кодер и декодер авторегрессии слева направо. Возникает вопрос, а в чём же разница между BART и оригинальным transformer. Есть несколько маленьких изменений, например, размер модели и функции активации. Более интересное изменение – BART работает с разными входными данными. В оригинальном transformer было два входа, исходная последовательность в кодер и ответ в декодер. BART же нужен только первый вход, он может всё ещё переводить текст, ему также нужен ответ, но ему не обязательно давать ответ прямо в декодер. Немного про структуру BART. Обычный текст сначала разрушается, а потом кодируется кодером. Это кодирование вместе с токенами идёт дальше в декодер. Будет посчитан cross-entropy loss между выходом кодера и оригинальным текстом, а потом всё это будет оптимизировано во время обучения. Эти два текста идут дальше в декодер. Чтобы пояснить шаг разрушения, вспомним, что BERT и GPT претренируются с помощью восстановления масок слов, BERT заполняет пробелы, а GPT предсказывает следующее слово. Это всё своего рода разрушения. В BART исходный текст разрушают, а второй дают какой он есть. Есть 5 способов разрушения в BART: token masking, token deletion, text infilling, sentence permutation, document rotation. В общем случае можно применить одну или несколько техник к одному предложению. Генерация последовательности в BART немного отличается по сравнению с GPT из-за кодера. BART последовательности более сопоставимы с обобщением, где модели представляются набором контекстов и предлагается сгенерировать резюме на определенные вопросы. BART – это лучший выбор для ответов на вопросы, диалогов, выводов.

## 16.5 Fine-tuning a BERT model in PyTorch

Конечно есть много моделей, но мы выберем для практики BERT, т.к. она хорошо балансирует между скоростью тренировки на одном GPU и качеством модели. Претренировать модель с нуля довольно сложно, поэтому мы пропустим этот этап. Первым шагом загрузим IMDb movie review датасет. Мы опять будем использовать библиотеку `transformers`. Мы будем использовать **DistilBERT** модель – это легковесная transformer модель, основанная на претренированной BERT модели. Эта модель содержит на 40% меньше параметров, работает на 60% быстрее, при этом сохраняет 95% качества исходной модели. Сначала сделаем все необходимые нам импорты:

```
import gzip
import shutil
import time

import pandas as pd
import requests
import torch
import torch.nn.functional as F
import torchtext

import transformers
from transformers import DistilBertTokenizerFast
from transformers import DistilBertForSequenceClassification
```

Зададим необходимые параметры и устройство:

```
# if True, causes cuDNN to only use deterministic
# convolution algorithms.
torch.backends.cudnn.deterministic = True
RANDOM_SEED = 123
torch.manual_seed(RANDOM_SEED)
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
NUM_EPOCHS = 3
```

Загрузим датасет, мы его уже видели в главах 8 и 15:

```

url = ("https://github.com/rasbt/"
       "machine-learning-book/raw/"
       "main/ch08/movie_data.csv.gz")
filename = url.split("/")[-1]

with open(filename, "wb") as f:
    r = requests.get(url)
    f.write(r.content)

with gzip.open('movie_data.csv.gz', 'rb') as f_in:
    with open('movie_data.csv', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)

df = pd.read_csv('movie_data.csv')
print(df.head(3))

```

Разобъём наш датасет на части:

```

train_texts = df.iloc[:35000]['review'].values
train_labels = df.iloc[:35000]['sentiment'].values

valid_texts = df.iloc[35000:40000]['review'].values
valid_labels = df.iloc[35000:40000]['sentiment'].values

test_texts = df.iloc[40000:]['review'].values
test_labels = df.iloc[40000:]['sentiment'].values

```

Теперь надо токенизировать датасет. См. код:

```

tokenizer = DistilBertTokenizerFast.from_pretrained(
    'distilbert-base-uncased')

# truncation — обрезка длинных последовательностей до
# максимальной длины, по умолчанию 512
# padding — дополнение последовательностей в батче до
# одинаковой длины
train_encodings = tokenizer(list(train_texts), truncation=True, padding=True)
valid_encodings = tokenizer(list(valid_texts), truncation=True, padding=True)
test_encodings = tokenizer(list(test_texts), truncation=True, padding=True)

```

Наследованные токенайзеры поддерживают консистентность между претренированной моделью и датасетом, поэтому они рекомендованы, если мы лишь хотим настроить модель. Теперь упакуем всё в датасет и dataloader:

```

class IMDbDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        # key = input_ids & attention_mask
        # len(val) = 35000, len(val[0]) = 512
        item = {key: torch.tensor(val[idx])
                for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

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

train_dataset = IMDbDataset(train_encodings, train_labels)
valid_dataset = IMDbDataset(valid_encodings, valid_labels)
test_dataset = IMDbDataset(test_encodings, test_labels)

train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=16, shuffle=True)
valid_loader = torch.utils.data.DataLoader(
    valid_dataset, batch_size=16, shuffle=False)
test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=16, shuffle=False)

```

Маска `attention_mask` в методе `__getitem__` содержит 0 и 1, они показывают на какие токены модель должна обращать внимание, а на какие нет. Здесь 0 соответствуют padding элементам, которые дополняют все 35000 примеров до одной длины. Пришло время загрузить и настроить саму претренированную BERT модель. Вот код для загрузки модели:

```

model = DistilBertForSequenceClassification.from_pretrained(
    'distilbert-base-uncased')
model.to(DEVICE)
model.train()

```

```
optim = torch.optim.Adam(model.parameters(), lr=5e-5)
```

Слово uncased означает, что модель не различает большие и маленькие буквы. Пришло время тренировать модель. Объявим функцию для подсчёта качества модели. Заметим, что эта функция считает conventional classification accuracy. Почему так сложно? Мы грузим батч за батчем в RAM или VRAM (GPU память), поэтому есть ограничения для работы с большими моделями. См. код:

```
def compute_accuracy(model, data_loader, device):
    with torch.no_grad():
        correct_pred, num_examples = 0, 0
        for batch_idx, batch in enumerate(data_loader):
            input_ids = batch['input_ids'].to(device)
            attention_mask = \
                batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(input_ids,
                            attention_mask=attention_mask)
            logits = outputs['logits']
            predicted_labels = torch.argmax(logits, 1)
            num_examples += labels.size(0)
            correct_pred += \
                (predicted_labels == labels).sum()
    return correct_pred.float() / num_examples * 100
```

Наконец-то напишем fine-tuning циклы:

```
start_time = time.time()
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, batch in enumerate(train_loader):
        input_ids = batch['input_ids'].to(DEVICE)
        attention_mask = batch['attention_mask'].to(DEVICE)
        labels = batch['labels'].to(DEVICE)

        outputs = model(input_ids,
                        attention_mask=attention_mask,
                        labels=labels)
        loss, logits = outputs['loss'], outputs['logits']

        optim.zero_grad()
        loss.backward()
        optim.step()

        if not batch_idx % 250:
            print(f'Epoch: {epoch+1:04d}/{NUM_EPOCHS:04d} | '
                  f'Batch: {batch_idx:04d}/'
                  f'{len(train_loader):04d} | '
                  f'Loss: {loss:.4f}')

    model.eval()
    # Context-manager that sets gradient calculation to on or off.
    with torch.set_grad_enabled(False):
        print(f'Training accuracy: {compute_accuracy(model, train_loader, DEVICE):.2f}%')
        f'\nValid accuracy: {compute_accuracy(model, valid_loader, DEVICE):.2f}%')
    print(f'Time elapsed: {(time.time() - start_time)/60:.2f} min')

print(f'Total Training Time: {(time.time() - start_time)/60:.2f} min')
print(f'Test accuracy: {compute_accuracy(model, test_loader, DEVICE):.2f}%')
```

Вот примерный вывод во время тренировки, он недетерминированный, поэтому у вас может немного отличаться. Здесь только 1 эпоха, вторую я не ждал, т.к. обучение проходит довольно долго:

```
# Epoch: 0001/0002 | Batch 0000/2188 | Loss: 0.6649
# Epoch: 0001/0002 | Batch 0250/2188 | Loss: 0.3023
# Epoch: 0001/0002 | Batch 0500/2188 | Loss: 0.3368
# Epoch: 0001/0002 | Batch 0750/2188 | Loss: 0.0985
# Epoch: 0001/0002 | Batch 1000/2188 | Loss: 0.3865
# Epoch: 0001/0002 | Batch 1250/2188 | Loss: 0.3441
# Epoch: 0001/0002 | Batch 1500/2188 | Loss: 0.3495
# Epoch: 0001/0002 | Batch 1750/2188 | Loss: 0.4035
# Epoch: 0001/0002 | Batch 2000/2188 | Loss: 0.2204
# Training accuracy: 95.80%
# Valid accuracy: 91.62%
# Time elapsed: 38.22 min
```

В главе 15 RNN достигла точности в 85%, а тут уже после первой эпохи на valid датасете 92%, виден значительный прирост. Посмотрим как можно использовать Trainer API для более удобной настройки модели. Мы писали цикл тренировки вручную, чтобы показать, что нет большой разницы по сравнению с CNN или RNN. Однако в `transformers` библиотеке есть удобный Trainer API. Напишем код:

```
model = DistilBertForSequenceClassification.from_pretrained(
    'distilbert-base-uncased')
model.to(DEVICE)
model.train()
optim = torch.optim.Adam(model.parameters(), lr=5e-5)

from transformers import Trainer, TrainingArguments
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    logging_dir='./logs',
    logging_steps=10,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    # optim and learning rate scheduler
    optimizers=(optim, None)
)
```

Этим кодом мы полностью заменили тренировочный цикл, который написали выше. Заметим, что мы ещё не использовали test датасет и не указали никаких метрик. Это потому что по умолчанию Trainer API показывает только training loss и не производит вычисление модели. Есть два способа вычислить финальную точность. Первый способ – написать функцию для подсчёта качества и создать новый Trainer объект. В аргументах ей передаются logits (по умолчанию для всех моделей) и метки. Надо установить дополнительную библиотеку `datasets`. См. код:

```
! pip install datasets

from datasets import load_metric
import numpy as np

metric = load_metric("accuracy")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(
        predictions=predictions, references=labels
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    compute_metrics=compute_metrics,
    optimizers=(optim, None)
)
```

Теперь запустим тренировку:

```
start_time = time.time()
trainer.train()
print(f'Total Training Time: {((time.time() - start_time)/60:.2f} min')
print(trainer.evaluate())
```

Второй метод для подсчёта финальной оценки – это переиспользовать прошлую функцию, которую мы написали руками:

```
model.eval()
model.to(DEVICE)
print(f'Test accuracy: {compute_accuracy(model, test_loader, DEVICE):.2f}%')
```

Если мы хотим проверять качество модели регулярно во время тренировки, мы можем попросить trainer выводить качество модели после каждой эпохи. Вот код:

```

from transformers import TrainingArguments
training_args = TrainingArguments("test_trainer",
                                  evaluation_strategy="epoch",
                                  ...)

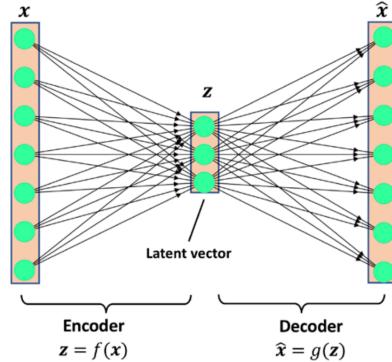
```

Однако в таком случае лучше использовать valid датасет, поменяем параметр `eval_dataset` на `valid_dataset`. В следующей главе мы обсудим generative adversarial networks. Как следует из названия, эти сети используют для генерации новых данных. Однако это область computer vision и генерации новых изображений.

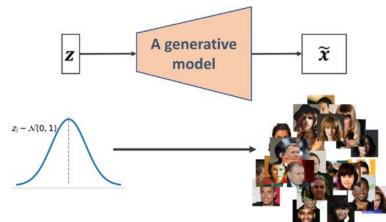
## 17 Generative Adversarial Networks (GANs)

### 17.1 Introducing GANs

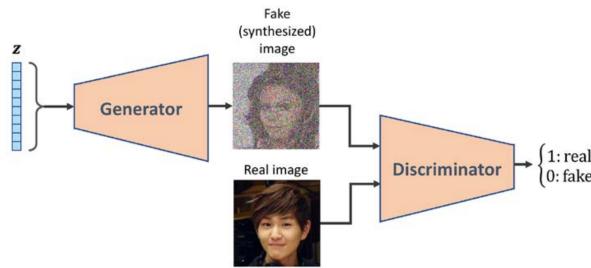
В computer vision GANs используют для перевода изображений (найти способ промаппить одно изображение другому), повышения разрешения изображений, дорисовывания отсутствующих частей изображения. Начнём обсуждение с autoencoders. Автокодеры могут сжимать и разжимать тренировочные данные. Автокодеры не умеют генерировать изображения, однако они нужны для понимания GANs. Автокодеры состоят из двух нейросетей: encoder сеть и decoder сеть. Первая сеть получает вектор фич  $x \in \mathbb{R}^d$  и кодирует его в вектор  $z \in \mathbb{R}^p$  ( $p < d$ ), то есть мы моделируем функцию  $z = f(x)$ .  $z$  называют latent vector. По сути происходит сжатие данных. Дальше декодер разжимает этот вектор в  $\hat{x}$ , имеем функцию  $\hat{x} = g(z)$ . Простая структура автокодера, где кодер и декодер состоят из одного fc слоя показана на рисунке:



В главе 5 мы прошли техники сжатия размерностей LDA и PCA. Автокодеры также можно использовать с этой целью. Если обе подсети линейны, то автокодеры практически идентичны PCA. Пусть веса одного слоя кодера находятся в матрице  $U$  (нет скрытых слоёв и нелинейных функций активации), тогда кодер выдаёт  $z = U^T x$ , аналогично декодер выдаёт  $\hat{x} = U z$ . Тогда в итоге мы получаем  $\hat{x} = UU^T x$ . Это ровно то, что делает PCA, за исключением условия  $UU^T = I_{n \times n}$ . Конечно мы можем добавить несколько скрытых нелинейных слоёв, чтобы автокодер лучше сжимал и разжимал. Также заметим, что мы использовали fc слои, но никто не мешает их заменить на свёрточные слои (см. главу 14). Обычно размер скрытого пространства меньше входного, поэтому его часто называют “bottleneck”, а такая конфигурация автокодеров называется undercomplete. Однако есть и другая категория, где  $p > d$ , их называют overcomplete. Во время тренировки overcomplete автокодеров есть тривиальное решение, где кодер и декодер просто копируют входные данные в выходной слой. Конечно такое решение не очень хорошее, однако с некоторыми модификациями процедуры тренировки, их можно использовать для уменьшения шума изображения. Для этого во время тренировки добавляют случайный шум  $\epsilon$  к входным данным, т.о. модель учится восстанавливать чистое изображение  $x$  из шумного сигнала  $x + \epsilon$ . Далее, когда модели дают реальное шумное изображение, она делает своё дело. Такие модели называют *denoising autoencoder*. Теперь уже поговорим про генеративные модели. Автокодеры – это детерминированные модели, они лишь разжимают данные, поэтому они не могут генерировать новые картинки. Генеративная модель может генерировать новый пример  $\tilde{x}$  из случайного вектора  $z$  (скрытый вектор). Мы берём  $z_i \sim Uniform(-1, 1)$  или  $z_i \sim Normal(\mu = 0, \sigma^2 = 1)$  и получаем  $\tilde{x}$ . См. картинку:



Видно, что генеративная модель имеет сходство с декодером в автокодере. Однако главное отличие в том, что мы не знаем распределение  $z$  в автокодере, в то время как в генеративной модели распределение  $z$  полностью характеризуемо. Есть подход **variational autoencoder (VAE)**, который обобщает автокодер до генеративной модели. В VAE кодер, получая  $x$  на вход, считает распределение скрытого вектора, т.е.  $\mu$  и  $\sigma^2$ . Во время обучения модель заставляют попадать в std, т.е.  $\mu = 0$  и  $\sigma^2 = 1$ . После обучения VAE, мы откодываем кодер и используем декодер для генерации примеров  $\tilde{x}$ , передавая случайное распределение вектора  $z$  из std. Кроме VAE есть другие типы генеративных моделей, например, autoregressive models и normalizing flow models. Теперь поговорим про генерацию новых примеров с помощью GANs. Чтобы понять, что делают GANs, предположим, что у нас есть сеть, которая получает случайный вектор  $z$  из известного распределения и генерирует выходное изображение  $\tilde{x} = G(z)$ , эту сеть мы назовём **generator** ( $G$ ). Наша цель – это генерировать изображения, например, животных. Как всегда мы сначала инициализируем сеть случайными числами, поэтому первые изображения будут похожи на шум. Пусть у нас есть функция качества изображения, назовём её *assessor* функция. Если у нас есть такая функция, то мы можем тренировать генератор. Давайте сделаем NN модель, которая будет оценивать качество новых изображений. Вот и вся идея GAN. Т.е. в GAN есть ещё одна NN, которая называется **discriminator** ( $D$ ), это классификатор, который учится отличать реальное изображение от фейкового. См. картинку:



В GAN генератор и дискриминатор учатся вместе. В итоге каждая сеть становится всё лучше и лучше. По сути получается игра, генератор пытается обмануть дискриминатор, в то же время дискриминатор пытается отличить подделку. Остался последний шаг – определить функции потерь для обеих сетей. Целевая функция в GANs выглядит так:

$$V(\theta^{(D)}, \theta^{(G)}) = E_{x \sim p_{data}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

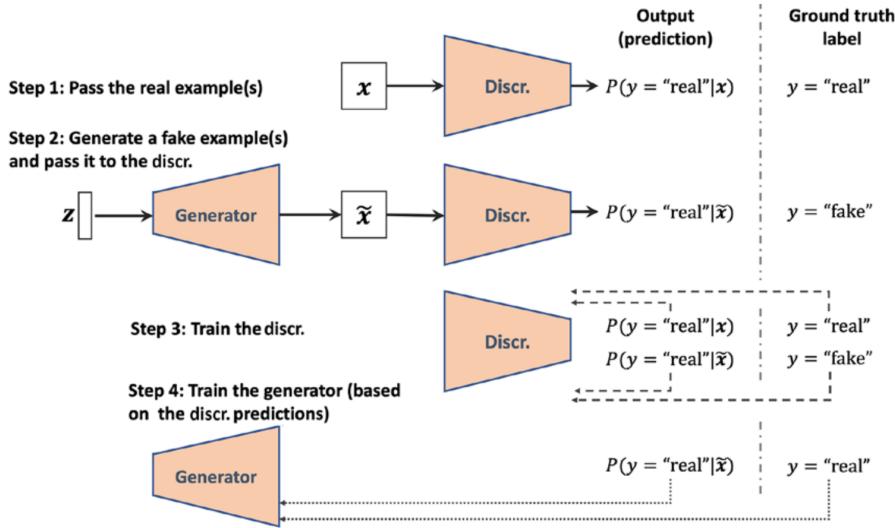
Здесь  $V(\theta^{(D)}, \theta^{(G)})$  – это **value function**, её можно интерпретировать как выигрыш, мы хотим максимизировать это значение по отношению к  $D$  и минимизировать по отношению к  $G$ , т.е.

$$\min_G \max_D V(\theta^{(D)}, \theta^{(G)})$$

$D(x)$  – это вероятность, которая показывает,  $x$  – это реальное изображение или фейк. Первое слагаемое относится к ожидаемому значению величины в скобках по отношению к примерам из распределения данных (распределение реальных примеров). Второе слагаемое относится к ожидаемому значению величины относительно распределения  $z$  векторов. Один шаг тренировки GAN требует два шага оптимизации: максимизация выигрыша для  $D$  и минимизация для  $G$ . То есть мы замораживаем одни параметры и в это время обновляем другие. Пусть мы обновили генератор, а теперь хотим обновить дискриминатор. Оба слагаемых в value функции участвуют в его оптимизации, первое слагаемое соответствует потерям на реальных примерах, второе – на фейковых. Поэтому, когда  $G$  зафиксирован, наша цель – максимизировать value функцию, чтобы  $D$  всё лучше и лучше отличал реальные изображения от фейковых. Теперь меняем  $G$  и  $D$  местами. В этом случае только второе слагаемое имеет значение. Т.е., когда  $D$  зафиксирован, наша цель – минимизировать value функцию. Функция  $\log (1 - D(G(z)))$  страдает от проблемы исчезающих градиентов на первых шагах обучения. Причина в том, что на первых шагах  $G(z)$  совсем не похожи на реальные примеры, поэтому с высокой вероятностью  $D(G(z))$  будет близка к 0. Этот феномен называется **saturation**. Чтобы решить эту проблему, можно переформулировать минимизацию второй части так:

$$\max_G E_{z \sim p_z(z)} [\log (D(G(z)))]$$

Эта замена означает, что для тренировки генератора, мы можем свопнуть метки для реальных и фейковых примеров и сделать обычную минимизацию функции. Другими словами, несмотря на то, что примеры генератора фейковые и помечены 0, мы можем свопнуть метки и присвоить 1 этим примерам. А дальше можно минимизировать BCE с этими новыми метками вместо максимизации формулы выше. Мы обозначаем 0 фейковые примеры, а 1 – реальные. Посмотрим на изображение, чтобы лучше понять все шаги простой GAN модели:



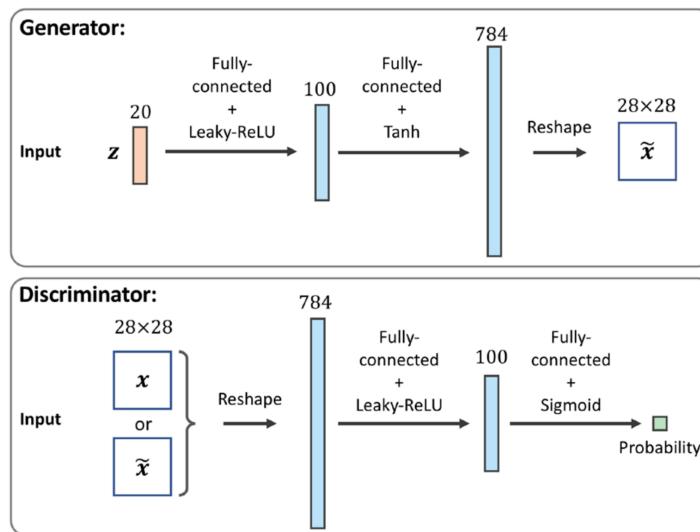
В следующей части мы напишем GAN с нуля для генерации новых рукописных цифр.

## 17.2 Implementing a GAN from scratch

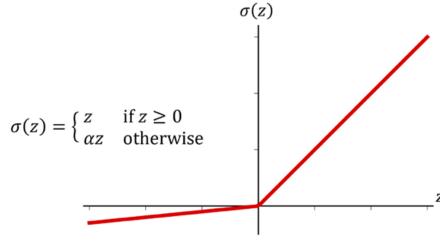
В этой части для ускорения вычислений нам понадобятся GPUs, поэтому писать код мы будем в Google Colab. Google Colab предоставляет CPUs, GPUs и даже tensor processing units (TPUs), однако время исполнения на них ограничено 12 часами. В этой среде мы можем создавать Jupyter Notebooks – это удобный графический интерфейс GUI для работы с кодом. Настроим среду и устройство:

```
import torch
print(torch.__version__) # 1.12.1+cu113
print("GPU Available:", torch.cuda.is_available()) # True
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = "cpu"
print(device) # cuda:0
```

Теперь реализуем генератор и дискриминатор. В генераторе и дискриминаторе будут две fc сети с одним и более скрытых слоёв. См. изображение:



На картинке изображена оригинальная GAN модель, мы будем её называть *vanilla GAN*. В этой модели для каждого скрытого слоя мы применяем leaky ReLU функцию. Использование просто ReLU приводит к разреженным градиентам, это может нам не подходит, если мы хотим иметь градиенты для всего спектра входных значений. В дискриминаторе после каждого скрытого слоя также идёт dropout слой.  $\tanh$  функция в генераторе рекомендована, т.к. она помогает обучаться. Скажем пару слов про leaky rectified linear unit (ReLU) функцию активации. Просто ReLU приводит к разреженным градиентам, это не всегда плохо, а часто даже хорошо, но конкретно в GANs мы хотим иметь градиенты для всего спектра значений, поэтому мы будем использовать leaky ReLU. Это небольшая модификация, которая отрицательным значениям сопоставляет маленькие числа вместо нулей. График этой функции вот такой:



Мы объявим две вспомогательные функции для создания двух сетей ( $D$  и  $G$ ):

```
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

def make_generator_network(
    input_size=20,
    num_hidden_layers=1,
    num_hidden_units=100,
    num_output_units=784):
    model = nn.Sequential()
    for i in range(num_hidden_layers):
        model.add_module(f'fc_g{i}', nn.Linear(input_size, num_hidden_units, bias=False))
        model.add_module(f'relu_g{i}', nn.LeakyReLU())
        input_size = num_hidden_units
    model.add_module(f'fc_g{num_hidden_layers}', nn.Linear(input_size, num_output_units))
    model.add_module('tanh_g', nn.Tanh())
    return model

def make_discriminator_network(
    input_size,
    num_hidden_layers=1,
    num_hidden_units=100,
    num_output_units=1):
    model = nn.Sequential()
    for i in range(num_hidden_layers):
        model.add_module(f'fc_d{i}', nn.Linear(input_size, num_hidden_units, bias=False))
        model.add_module(f'relu_d{i}', nn.LeakyReLU())
        model.add_module('dropout', nn.Dropout(p=0.5))
        input_size = num_hidden_units
    model.add_module(f'fc_d{num_hidden_layers}', nn.Linear(input_size, num_output_units))
    model.add_module('sigmoid', nn.Sigmoid())
    return model
```

Теперь надо задать параметры для тренировки модели. Как мы помним изображения в MNIST размером  $28 \times 28$  (grayscale images). Размер скрытого пространства будет равен 20. См. код:

```
image_size = (28, 28)
z_size = 20
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100
torch.manual_seed(1)
gen_model = make_generator_network(
    input_size=z_size,
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size))
print(gen_model)
# Sequential(
#   (fc_g0): Linear(in_features=20, out_features=100, bias=False)
#   (relu_g0): LeakyReLU(negative_slope=0.01)
#   (fc_g1): Linear(in_features=100, out_features=784, bias=True)
#   (tanh_g): Tanh()
# )

disc_model = make_discriminator_network(
    input_size=np.prod(image_size),
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)
```

```

print(disc_model)
# Sequential(
#   (fc_d0): Linear(in_features=784, out_features=100, bias=False)
#   (relu_d0): LeakyReLU(negative_slope=0.01)
#   (dropout): Dropout(p=0.5, inplace=False)
#   (fc_d1): Linear(in_features=100, out_features=1, bias=True)
#   (sigmoid): Sigmoid()
# )

```

Теперь нужно создать тренировочный датасет. Так как генератор на выходе использует `tanh`, то пиксели новых изображений лежат в  $(-1, 1)$ . Однако MNIST изображения в  $[0, 255]$  (`PIL.Image.Image`), поэтому мы применим функцию `torchvision.transforms.ToTensor`. После этого, помимо смены типа, пиксели будут в  $[0, 1]$ . Теперь осталось вычесть 0.5 и поделить на 0.5. Эти сдвиги улучшат gradient descent-based обучение. См. код:

```

import torchvision
from torchvision import transforms
image_path = './'
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5), std=(0.5)),
])
mnist_dataset = torchvision.datasets.MNIST(
    root=image_path, train=True,
    transform=transform, download=True
)
example, label = next(iter(mnist_dataset))
print(f'Min: {example.min()} Max: {example.max()}'')
# Min: -1.0 Max: 1.0
print(example.shape)
# torch.Size([1, 28, 28])

```

Теперь нужно создать случайный вектор  $z$ , основанный на известном распределении:

```

def create_noise(batch_size, z_size, mode_z):
    if mode_z == 'uniform':
        input_z = torch.rand(batch_size, z_size)*2 - 1
    elif mode_z == 'normal':
        input_z = torch.randn(batch_size, z_size)
    return input_z

```

Напишем пример кода для лучшего понимания GANs:

```

from torch.utils.data import DataLoader
batch_size = 32
dataloader = DataLoader(mnist_dataset, batch_size, shuffle=False)
input_real, label = next(iter(dataloader))
input_real = input_real.view(batch_size, -1)
# torch.Size([32, 1, 28, 28]) > torch.Size([32, 784])
torch.manual_seed(1)
mode_z = 'uniform'
input_z = create_noise(batch_size, z_size, mode_z)
print('input-z — shape:', input_z.shape) # [32, 20]
print('input-real — shape:', input_real.shape) # [32, 784]

g_output = gen_model(input_z)
print('Output of G — shape:', g_output.shape) # [32, 784]

d_proba_real = disc_model(input_real)
d_proba_fake = disc_model(g_output)
print('Disc. (real) — shape:', d_proba_real.shape) # [32, 1]
print('Disc. (fake) — shape:', d_proba_fake.shape) # [32, 1]

```

Теперь будем тренировать нашу модель. Заведём функцию потерь:

```

loss_fn = nn.BCELoss()
# for generator:
g_labels_real = torch.ones_like(d_proba_fake)
g_loss = loss_fn(d_proba_fake, g_labels_real)
print(f'Generator Loss: {g_loss:.4f}') # 0.6863

# for discriminator:
d_labels_real = torch.ones_like(d_proba_real)
d_labels_fake = torch.zeros_like(d_proba_fake)
d_loss_real = loss_fn(d_proba_real, d_labels_real)
d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
print(f'Discriminator Losses: Real {d_loss_real:.4f} Fake {d_loss_fake:.4f}')
# Real 0.6226 Fake 0.7007

```

Подготовим всё к обучению, а затем уже напишем цикл обучения:

```

batch_size = 64
torch.manual_seed(1)
np.random.seed(1)
mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
                      shuffle=True, drop_last=True)

gen_model = make_generator_network(
    input_size=z_size,
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size)
).to(device)

disc_model = make_discriminator_network(
    input_size=np.prod(image_size),
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size
).to(device)

loss_fn = nn.BCELoss()
g_optimizer = torch.optim.Adam(gen_model.parameters())
d_optimizer = torch.optim.Adam(disc_model.parameters())

```

Напишем две функции тренировки для обеих сетей:

```

def d_train(x):
    disc_model.zero_grad()
    # real batch
    batch_size = x.size(0)
    x = x.view(batch_size, -1).to(device)
    d_labels_real = torch.ones(batch_size, 1, device=device)
    d_proba_real = disc_model(x)
    d_loss_real = loss_fn(d_proba_real, d_labels_real)
    # fake batch
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)
    d_proba_fake = disc_model(g_output)
    d_labels_fake = torch.zeros(batch_size, 1, device=device)
    d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
    # optimize
    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    d_optimizer.step()
    return d_loss.data.item(), d_proba_real.detach(), d_proba_fake.detach()

def g_train(x):
    gen_model.zero_grad()
    batch_size = x.size(0)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_labels_real = torch.ones(batch_size, 1, device=device)

    g_output = gen_model(input_z)
    d_proba_fake = disc_model(g_output)
    g_loss = loss_fn(d_proba_fake, g_labels_real)

    g_loss.backward()
    g_optimizer.step()
    return g_loss.data.item()

```

Теперь уже приступим к тренировке, мы будем тренировать по очереди обе сети, сохраняя потери, а так же мы будем генерировать изображения после каждой эпохи. См. код:

```

fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
def create_samples(g_model, input_z):
    g_output = g_model(input_z)
    images = torch.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0

epoch_samples = []
all_d_losses = []
all_g_losses = []
all_d_real = []
all_d_fake = []
num_epochs = 30

for epoch in range(1, num_epochs+1):
    d_losses, g_losses = [], []
    d_vals_real, d_vals_fake = [], []
    for i, (x, _) in enumerate(mnist_dl):

```

```

d_loss, d_proba_real, d_proba_fake = d_train(x)
d_losses.append(d_loss)
g_losses.append(g_train(x))
d_vals_real.append(d_proba_real.mean().cpu())
d_vals_fake.append(d_proba_fake.mean().cpu())
all_d_losses.append(torch.tensor(d_losses).mean())
all_g_losses.append(torch.tensor(g_losses).mean())
all_d_real.append(torch.tensor(d_vals_real).mean())
all_d_fake.append(torch.tensor(d_vals_fake).mean())
print(f'Epoch {epoch:03d} | Avg Losses >>')
    f' G/D {all_g_losses[-1]:.4f}/{all_d_losses[-1]:.4f},'
    f' [D-Real: {all_d_real[-1]:.4f},'
    f' D-Fake: {all_d_fake[-1]:.4f}]')
epoch_samples.append(
    create_samples(gen_model, fixed_z).detach().cpu().numpy())
)
# Epoch 001 | Avg Losses >> G/D 0.9441/0.9336 [D-Real: 0.7775 D-Fake: 0.4667]
# Epoch 002 | Avg Losses >> G/D 0.7610/1.2846 [D-Real: 0.5560 D-Fake: 0.4788]
# Epoch 003 | Avg Losses >> G/D 1.1606/1.1124 [D-Real: 0.6091 D-Fake: 0.3976]
# ...
# Epoch 029 | Avg Losses >> G/D 0.9490/1.2386 [D-Real: 0.5707 D-Fake: 0.4323]
# Epoch 030 | Avg Losses >> G/D 0.9914/1.2116 [D-Real: 0.5807 D-Fake: 0.4206]

```

Я тренирую на 30 эпохах, чтобы долго не ждать, но для хорошего результата нужно около 100 эпох. После тренировки полезно построить потери обеих сетей, чтобы проанализировать их поведение и понять, где они сошлись. Очень полезно выводить средние значения для real и fake примеров после каждой итерации. Мы ожидаем, что эти вероятности будут около 0.5, это значит, что дискриминатор не может точно отличить реальные изображения от фейковых. См. код:

```

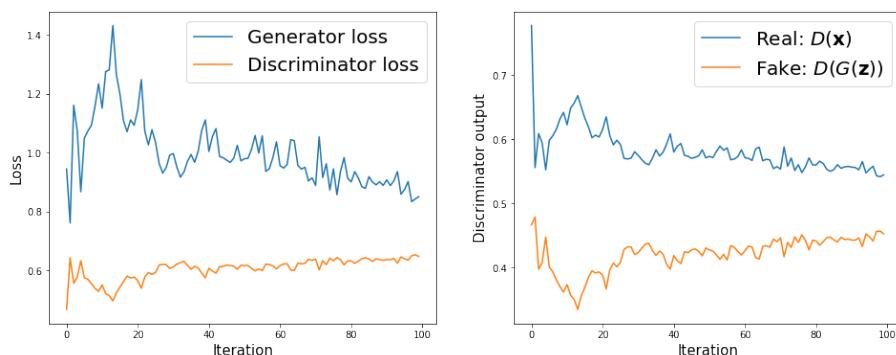
import itertools
fig = plt.figure(figsize=(16, 6))

ax = fig.add_subplot(1, 2, 1)
plt.plot(all_g_losses, label='Generator loss')
half_d_losses = [all_d_loss/2 for all_d_loss in all_d_losses]
plt.plot(half_d_losses, label='Discriminator loss')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

ax = fig.add_subplot(1, 2, 2)
plt.plot(all_d_real, label='Real: $D(\mathbf{x})$')
plt.plot(all_d_fake, label='Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)
plt.show()

```

В итоге, чтобы построить нормальные графики, мне пришлось дотренировать модель до 100 эпох. См. картинку:



Как мы видим, по началу D хорошо отличал примеры, но чем дальше шло обучение, тем больше он сомневался. В начале G имел большие потери, но со временем они становились всё меньше, а вот у D они наоборот становились всё больше. Посмотрим на цифры, которые мы получили:

```

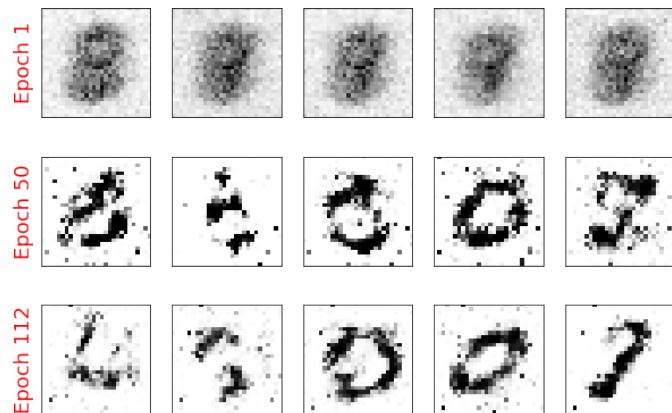
selected_epochs = [1, 50, 112]
fig = plt.figure(figsize=(10, 14))
for i, e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])

```

```

if j == 0:
    ax.text(
        -0.06, 0.5, f'Epoch {e}',
        rotation=90, size=18, color='red',
        horizontalalignment='right',
        verticalalignment='center',
        transform=ax.transAxes
    )
image = epoch_samples[e-1][j]
ax.imshow(image, cmap='gray_r')
plt.show()

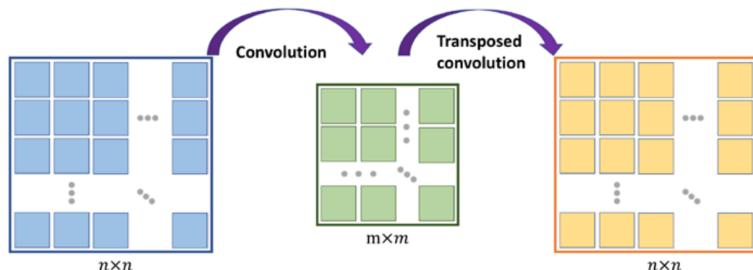
```



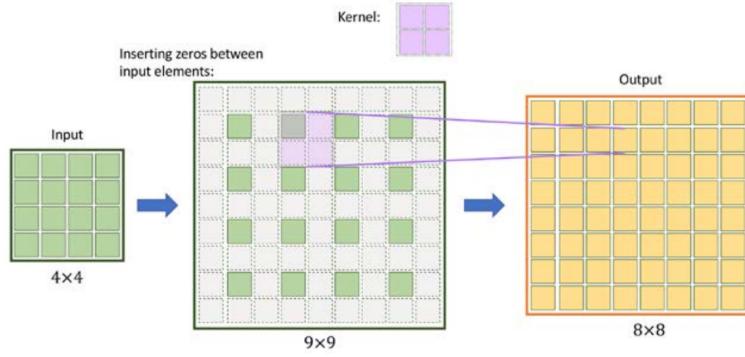
Мы видим, что качество цифр стало сильно лучше, чем на 0 эпохе, однако цифры всё ещё далеки от хороших. Для улучшения качества модели будем использовать свёрточные слои. В следующей части мы реализуем deep convolutional GAN (DCGAN).

### 17.3 Convolutional and Wasserstein GANs

В этой части мы реализуем DCGAN, в добавок мы кратко обсудим дополнительную ключевую технику **Wasserstein GAN (WGAN)**. Мы обсудим ещё две техники: transposed convolution и batch normalization (BatchNorm). В DCGAN было предложено использовать свёрточные слои для обеих сетей. Начиная с случайного вектора  $z$ , мы сначала используем fc слой, чтобы перегнать  $z$  в вектор специального размера, который уже можно представить в виде  $h \times w \times c$ , этот размер меньше выходного изображения. Затем серия свёрточных слоёв (транспонированных свёрток) используется, чтобы перегнать feature maps в желаемый выходной размер. Начнём разговор с транспонированной свёртки. В главе 14 мы прошли свёртки в одно и двумерном пространствах, мы посмотрели как padding и strides меняют размер выходных feature maps. Свёртка обычно применяется для уменьшения размера (можно поставить  $stride = 2$  или добавить pooling слой после свёрточного), а вот *transposed convolution* применяют для увеличения размера. Пусть у нас есть feature map размера  $n \times n$ , дальше мы делаем свёртку и получаем размер  $m \times m$ . Вопрос в том, как применить ещё одну свёртку, чтобы обратно получить размер  $n \times n$ , сохраняя шаблоны между входом и выходом. См. картинку:



Transposed convolution так же называют fractionally strided convolution. В DL её так же называют deconvolution. Однако изначально deconvolution определяли подругому. Пусть  $f_w(x) = x'$ , тогда deconvolution – это  $f^{-1}$ ,  $f_w^{-1}(f(x)) = x$ . Transposed convolution фокусируется на восстановлении размера, а не исходных значений. Идея вставки нулей между исходными элементами. К входному размеру  $4 \times 4$  применим tr. свёртку с stride  $2 \times 2$  и kernel  $2 \times 2$ . Мы получим размер  $9 \times 9$ , далее применим обычную свёртку с ядром  $2 \times 2$  и stride 1 и получим на выходе размер  $8 \times 8$ . Для проверки можно сделать обычную свёртку на выходном значении с stride 2 и получить выход размером  $4 \times 4$ . См. пример на картинке:



Теперь поговорим про batch normalization. Главная идея BatchNorm в нормализации входного слоя и избежании изменений в их распределении во время тренировки, что приводит к быстрой и лучшей сходимости. BatchNorm преобразует mini-batch на основе посчитанной статистики. Пусть у нас есть  $4 \times$  мерный тензор  $Z$  после свёрточного слоя размером  $[m \times c \times h \times w]$ , где  $m$  – это размер батча. BatchNorm можно записать в виде 3х шагов:

- Посчитаем mean и std для net inputs для каждого mini-batch:

$$\mu_B = \frac{1}{m \times h \times w} \sum_{i,j,k} Z^{[i,j,k]}$$

$$\sigma_B^2 = \frac{1}{m \times h \times w} \sum_{i,j,k} (Z^{[i,j,k]} - \mu_B)^2$$

Где  $\mu_B$  и  $\sigma_B^2$  имеют размер  $c$ .

- Стандартизуем net inputs для всех примеров в батче:

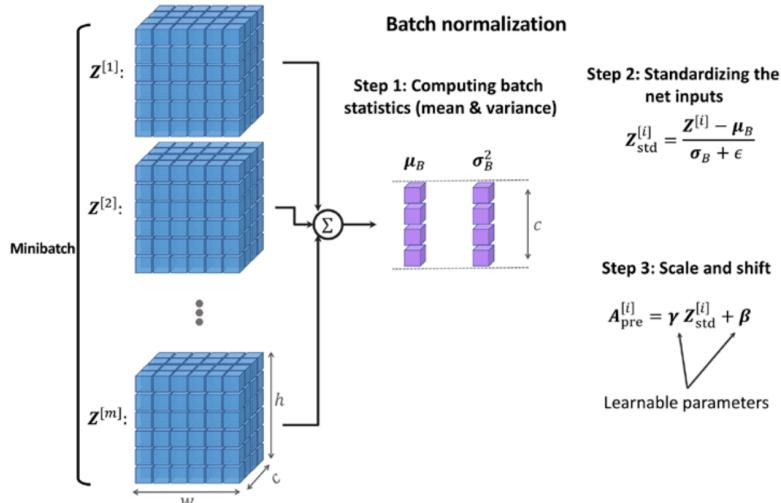
$$Z_{std}^{[i]} = \frac{Z^{[i]} - \mu_B}{\sigma_B + \epsilon}$$

Где  $\epsilon$  – это маленькое число, чтобы не было деления на 0.

- Масштабируем и сдвинем  $Z_{std}$ , используя два обучающихся вектора параметров  $\gamma$  и  $\beta$  размера  $c$ :

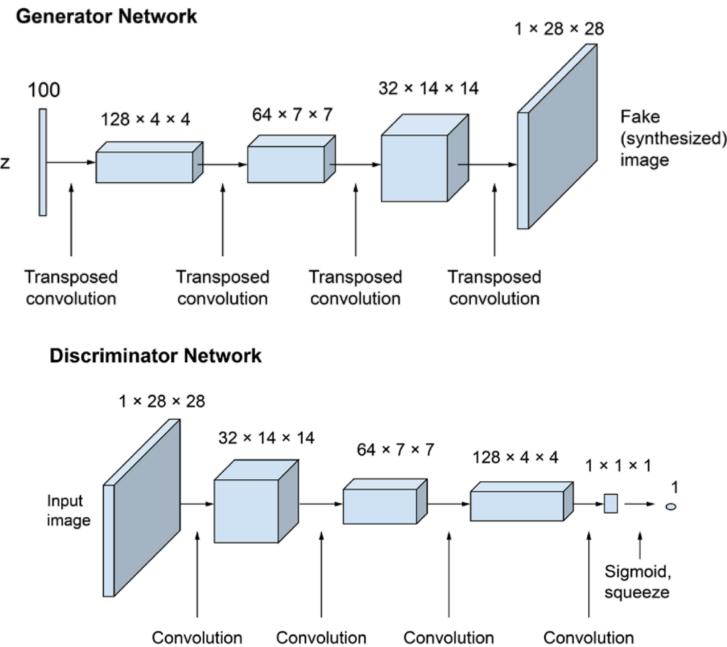
$$A_{pre}^{[i]} = \gamma Z_{std}^{[i]} + \beta$$

См. изображение для лучшего понимания:



С одной стороны центрирование и единичная дисперсия – это хорошо для алгоритмов градиентного спуска. С другой стороны постоянная нормализация так, чтобы разные mini-batch, которые могут отличаться, имели одинаковые свойства, может жёстко влиять на качество NN модели. Чтобы этого избежать, как раз и были введены параметры  $\gamma$  и  $\beta$ . Параметры  $\gamma$ ,  $\beta$ ,  $\mu_B$  и  $\sigma_B^2$  используются и для тестовых данных. Почему BatchNorm помогает оптимизациям? Изначально она была придумана, чтобы уменьшить internal covariance shift – это изменения, которые происходят в распределениях активаций в слоях из-за обновления параметров сети. Чтобы объяснить это на простом примере,

представим, что фиксированный batch проходит через сеть на 1ой эпохе. Мы запомним активации, далее начинается вторая эпоха и этот батч опять проходит через сеть. Мы заметим, что значения в этих двух эпохах сильно отличаются, т.к. поменялись значения параметров модели. Это называется internal covariance shift, что замедляет обучение NN модели. Однако потом выяснилось, что главная эффективность не в этом феномене, а в более гладкой поверхности функции потерь, что делает невыщуклые оптимизации более устойчивыми. PyTorch имеет класс `nn.BatchNorm2d()` (и `nn.BatchNorm1d()`), его можно добавить как обычный слой. Этот класс делает всё то, что мы описали, важно помнить, что  $\gamma$  и  $\beta$  обновляются только во время тренировки. Мы прошли основные компоненты DCGAN, поэтому теперь реализуем gen. и disc. Архитектуру сетей см. на картинках:



Генератор принимает вектор  $z$  размера 100, затем серия `nn.ConvTranspose2d()` приводит к итоговой картинке. Каждый tr. свёрточный слой дополняется BatchNorm слоем и leaky ReLU функцией, кроме последнего (в нём только `tanh`). В disc. каждый свёрточный слой так же дополняется BatchNorm и leaky ReLU слоями. Последний слой использует ядро размера  $7 \times 7$  и один фильтр, чтобы выдать одно число. Соображения по архитектурному проектированию для свёрточных GAN. Заметим, что число feature maps имеет разные тренды в disc. и gen. В gen. мы начинаем с большого числа f.m. и уменьшаем вплоть до последнего слоя. В disc. всё наоборот. Так же заметим, что spatial size идёт в обратном порядке по сравнению с числом feature maps. Это очень важный момент для CNNs. Так же не рекомендуется использовать bias unit в слоях после BatchNorm слоя. В нашем случае bias не имеет смысла, т.к. мы уже имеем сдвиг в  $\beta$ . А вот и код для создания сетей:

```
# Напомню, что есть формула o=floor((n+2p-m)/s)+1.
# Как раз на её основе и вычисляются параметры для
# nn.ConvTranspose2d слоёв.
# Например, если из 14x14 получается 28x28, то надо сделать
# такие параметры, чтобы обычная свёртка делала обратное
# преобразование. Начальный размер: 100x1x1.
def make_generator_network(input_size, n_filters):
    model = nn.Sequential(
        # in_channels, out_channels, kernel_size, stride, padding
        nn.ConvTranspose2d(input_size, n_filters*4, 4, 1, 0, bias=False),
        nn.BatchNorm2d(n_filters*4),
        nn.LeakyReLU(0.2),
        nn.ConvTranspose2d(n_filters*4, n_filters*2, 3, 2, 1, bias=False),
        nn.BatchNorm2d(n_filters*2),
        nn.LeakyReLU(0.2),
        nn.ConvTranspose2d(n_filters*2, n_filters, 4, 2, 1, bias=False),
        nn.BatchNorm2d(n_filters),
        nn.LeakyReLU(0.2),
        nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
        nn.Tanh()
    )
    return model

class Discriminator(nn.Module):
    def __init__(self, n_filters):
        super().__init__()

```

```

        self.network = nn.Sequential(
            nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2),
            nn.Conv2d(n_filters, n_filters*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_filters * 2),
            nn.LeakyReLU(0.2),
            nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1, bias=False),
            nn.BatchNorm2d(n_filters*4),
            nn.LeakyReLU(0.2),
            nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.network(input)
        return output.view(-1, 1).squeeze(0)

z_size = 100
image_size = (28, 28)
n_filters = 32
gen_model = make_generator_network(z_size, n_filters).to(device)
print(gen_model)
# Sequential(
#   (0): ConvTranspose2d(100, 128, kernel_size=(4, 4), stride=(1, 1), bias=False)
#   (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
#   (2): LeakyReLU(negative_slope=0.2)
#   (3): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
#   (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
#   (5): LeakyReLU(negative_slope=0.2)
#   (6): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
#   (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
#   (8): LeakyReLU(negative_slope=0.2)
#   (9): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
#   (10): Tanh()
# )

disc_model = Discriminator(n_filters).to(device)
print(disc_model)
# Discriminator(
#   (network): Sequential(
#     (0): Conv2d(1, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
#     (1): LeakyReLU(negative_slope=0.2)
#     (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
#     (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
#     (4): LeakyReLU(negative_slope=0.2)
#     (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
#     (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
#     (7): LeakyReLU(negative_slope=0.2)
#     (8): Conv2d(128, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
#     (9): Sigmoid()
#   )
# )

```

Теперь напишем функции тренировки и зададим пару параметров, изменения будут, но они не слишком большие:

```

loss_fn = nn.BCELoss()
g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0003)
d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)

def create_noise(batch_size, z_size, mode_z):
    if mode_z == 'uniform':
        input_z = torch.rand(batch_size, z_size, 1, 1)*2 - 1
    elif mode_z == 'normal':
        input_z = torch.randn(batch_size, z_size, 1, 1)
    return input_z

def d_train(x):
    disc_model.zero_grad()
    # real batch
    batch_size = x.size(0)
    # менять не нужно view:
    x = x.to(device)
    d_labels_real = torch.ones(batch_size, 1, device=device)

```

```

d_proba_real = disc_model(x)
d_loss_real = loss_fn(d_proba_real, d_labels_real)
# fake batch
input_z = create_noise(batch_size, z_size, mode_z).to(device)
g_output = gen_model(input_z)
d_proba_fake = disc_model(g_output)
d_labels_fake = torch.zeros(batch_size, 1, device=device)
d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
# optimize
d_loss = d_loss_real + d_loss_fake
d_loss.backward()
d_optimizer.step()
return d_loss.data.item(), d_proba_real.detach(), d_proba_fake.detach()

```

Приступим к тренировке:

```

fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
epoch_samples = []
torch.manual_seed(1)
num_epochs = 100
for epoch in range(1, num_epochs+1):
    gen_model.train()
    d_losses = []
    g_losses = []
    for i, (x, _) in enumerate(mnist_dl):
        d_loss, d_proba_real, d_proba_fake = d_train(x)
        d_losses.append(d_loss)
        g_losses.append(g_train(x))
    print(f'Epoch {epoch:03d} | Avg Losses >>',
          f' G/D {torch.FloatTensor(g_losses).mean():.4f} ',
          f'/{torch.FloatTensor(d_losses).mean():.4f} ')
    gen_model.eval()
    epoch_samples.append(
        create_samples(gen_model, fixed_z).detach().cpu().numpy())
)
if epoch % 5 == 0:
    ask = input()
    if ask == "break":
        break
# Epoch 001 / Avg Losses >> G/D 5.1531/0.1089
# Epoch 002 / Avg Losses >> G/D 3.9652/0.2696
# ...

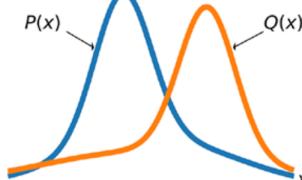
```

Теперь посмотрим на качество изображений (код для вывода картинок не меняется вообще):

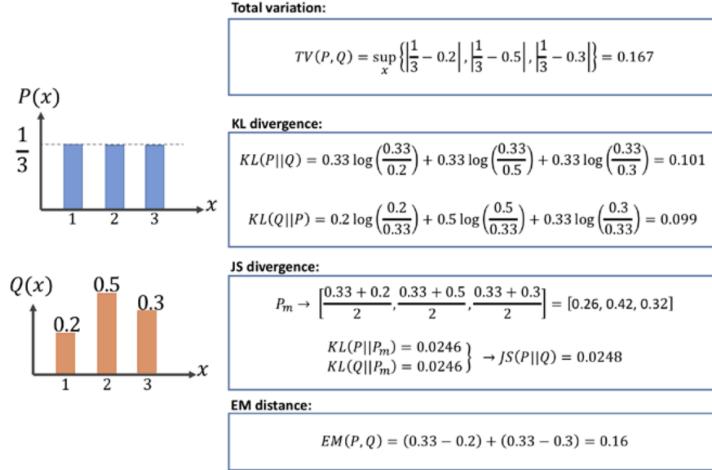


Сразу видно, что качество изображений значительно выросло. Возникает вопрос, как оценить качество GAN модели. Самый простой способ – просто посмотреть. Есть более сложные способы вычислений, которые оценивают изображения качественно и количественно. Есть теоретический аргумент, что генератор должен минимизировать непохожесть между распределениями в реальных и синтезированных данных. Поэтому, если использовать cross-entropy в качестве функции потерь, то модель будет вести себя не очень хорошо. Давайте обсудим WGAN, которые используют модифицированную функцию потерь, основанную на Wasserstein-1 расстоянии между распределением реальных и поддельных изображений. Для начала поговорим про меры различия между двумя распределениями. Сначала посмотрим на различные меры расхождения между двумя распределениями. Пусть  $P(x)$  и  $Q(x)$  – два распределения случайной величины  $x$ , тогда различие можно посчитать так:

Measures	Formulation
Total variation (TV)	$TV(P, Q) = \sup_x  P(x) - Q(x) $
Kullback-Leibler (KL) divergence	$KL(P  Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Jensen-Shannon (JS) divergence	$JS(P, Q) = \frac{1}{2} \left( KL\left(P  \frac{P+Q}{2}\right) + KL\left(Q  \frac{P+Q}{2}\right) \right)$
Earth mover's (EM) distance	$EM(P, Q) = \inf_{\gamma \in \Pi(P, Q)} E_{(u,v) \in Y} (\ u - v\ )$



TV мера считает самое большое расстояние между двумя точками при одном  $x$ . EM меру можно понимать как минимальную работу, которую нужно сделать, чтобы превратить одно распред. в другое. EM мера это сама по себе тоже задача оптимизации плана переноса  $\gamma(u, v)$ . KL и JS меры приходят к нам из теории информации. Заметим, что  $KL(P||Q) \neq KL(Q||P)$ , в отличие от JS меры. Эти формулы работают для непрерывных распред., но они могут быть расширены и на дискретный случай. А вот и пример их подсчёта:



Рассмотрим EM меру подробнее. Здесь у нас перевес на  $0.5 - \frac{1}{3} = 0.166$ , поэтому удобно перелить это значение в точки 1 и 3. Посмотрим на связь между KL divergence и cross-entropy.  $KL(P||Q)$  измеряет относительную энтропию распределения P по отношению к эталонному распределению Q. Формулу для KL можно расписать так:

$$KL(P||Q) = - \int P(x) \log(Q(x)) dx - \left( - \int P(x) \log(P(x)) dx \right)$$

Для дискретного случая можно написать так:

$$KL(P||Q) = - \sum_i P(x_i) \frac{P(x_i)}{Q(x_i)}$$

Тогда можно написать так:

$$KL(P||Q) = - \sum_i P(x_i) \log(Q(x_i)) - \left( - \sum_i P(x_i) \log(P(x_i)) \right)$$

Можно увидеть, что это cross-entropy между P и Q за вычетом self-entropy, т.е. можно написать так:  $KL(P||Q) = H(P, Q) - H(P)$ . Можно показать, что функция потерь, которую мы использовали, на самом деле минимизирует JS разницу, но эта мера имеет в данном контексте небольшие проблемы, поэтому лучше использовать EM меру. Какие плюсы у EM меры? Пусть у нас есть две параллельные линии  $P$  и  $Q$ . Одна линия – это  $x = 0$ , вторая линия может двигаться вдоль оси  $x$ , но изначально находится в точке  $x = \theta$ , где  $\theta > 0$ . Тогда  $KL(P||Q) = +\infty$ ,  $TV(P, Q) = 1$  и  $JS(P, Q) = \frac{1}{2} \log(2)$ . Ни одна из этих функций не есть функция от параметра  $\theta$ , поэтому её нельзя дифференцировать по  $\theta$ , т.е. мы не можем сделать  $P$  и  $Q$  более похожими. А вот  $EM(P, Q) = |\theta|$ , у этой функции есть градиент. Как использовать EM меру для тренировки модели? Пусть  $P_r$  - распред. реальных примеров, а  $P_g$  – распред. фейковых примеров. Мы уже говорили, что это сама по себе задача оптимизации, поэтому это очень затратно считать на каждой итерации цикла обучения. К счастью, есть теорема **Kantorovich-Rubinstein duality**, которая упрощает вычисление EM меры:

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} E_{u \in P_r}[f(u)] - E_{v \in P_g}[f(v)]$$

Здесь sup берётся по всем 1-Lipschitz непрерывным функциям, которые обозначают как  $\|f\|_L \leq 1$ . Пару слов про непрерывность Липшица. Это такая функция, которая удовлетворяет условию  $|f(x_1) - f(x_2)| \leq |x_1 - x_2|$ . Более того, вещ. функция  $f : \mathbb{R} \rightarrow \mathbb{R}$ , которая удовлетворяет условию  $|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$ , называется **K-Lipschitz continuous**. Теперь используем EM меру на практике для GANs. Вопрос в том, как найти 1-непрерывные функции Липшица, чтобы посчитать Wasserstein расстояние между  $P_r$  и  $P_g$ . Теория по этому вопросу нетривиальная, но ответ более чем простой. Вспомните, что мы рассматривали NNs как вещь для приближения функций. Это

значит, что можно просто тренировать ещё одну NN для вычисления Wasserstein расстояния. У нас был дискр. в виде классификатора, теперь пусть он возвращает просто число вместо вероятности (аналогия с оценкой жюри). Напишем формулы для функций потерь  $D$  и  $G$ , чтобы можно было использовать Wasserstein расст. Вот все компоненты:

$$L_{\text{real}}^D = -\frac{1}{N} \sum_i D(x_i)$$

$$L_{\text{fake}}^D = \frac{1}{N} \sum_i D(G(z_i))$$

$$L^G = -\frac{1}{N} \sum_i D(G(z_i))$$

Во время обучения функция так же должна быть Липшицевой, поэтому будем держать весовые параметры в  $[-0.01, 0.01]$ . Теперь поговорим про gradient penalty. Предложение по удержанию параметров в маленьком диапазоне может вызвать проблему маленьких или больших градиентов, к тому же критик ( $D$ ) не сможет учиться более сложным функциям. Поэтому было предложено другое решение – gradient penalty (GP). Результат – это **WGAN with gradient penalty (WGAN-GP)**. GP добавляют на каждой итерации, процедура состоит из 5 шагов:

1. Для каждой пары реальных и сген. изображений  $(x^{[i]}, \tilde{x}^{[i]})$  из данного батча выберем случайное число  $\alpha^{[i]} \in U(0, 1)$ .
2. Посчитаем интерполяцию между реальными и сген. изображ.  $\tilde{x}^{[i]} = \alpha x^{[i]} + (1 - \alpha) \tilde{x}^{[i]}$ , получим батч интерполированных примеров.
3. Посчитаем выход критика ( $D$ ) для всех интер. примеров  $D(\tilde{x}^{[i]})$ .
4. Посчитаем градиенты выхода критика  $\nabla_{\tilde{x}^{[i]}} D(\tilde{x}^{[i]})$ .
5. Вычислим GP так:

$$L_{gp}^D = \frac{1}{N} \sum_i \left( \|\nabla_{\tilde{x}^{[i]}} D(\tilde{x}^{[i]})\|_2 - 1 \right)^2$$

Тогда итоговые потери для  $D$  можно посчитать так:

$$L_{\text{total}}^D = L_{\text{real}}^D + L_{\text{fake}}^D + \lambda L_{gp}^D$$

Здесь  $\lambda$  – это очередной гиперпараметр. Теперь реализуем WGAN-GP, чтобы тренировать DCGAN модель. Мы уже написали функцию и класс для  $G$  и  $D$ . Здесь рекомендуется использовать layer normalization в WGAN вместо batch normalization (нормализация вдоль фич вместо норм. вдоль батча). А вот и код для WGAN модели:

```
def make_generator_wgan(input_size, n_filters):
    model = nn.Sequential(
        nn.ConvTranspose2d(input_size, n_filters * 4, 4, 1, 0, bias=False),
        nn.InstanceNorm2d(n_filters * 4),
        nn.LeakyReLU(0.2),
        nn.ConvTranspose2d(n_filters * 4, n_filters * 2, 3, 2, 1, bias=False),
        nn.InstanceNorm2d(n_filters * 2),
        nn.LeakyReLU(0.2),
        nn.ConvTranspose2d(n_filters * 2, n_filters, 4, 2, 1, bias=False),
        nn.InstanceNorm2d(n_filters),
        nn.LeakyReLU(0.2),
        nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
        nn.Tanh()
    )
    return model

class DiscriminatorWGAN(nn.Module):
    def __init__(self, n_filters):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2),
            nn.Conv2d(n_filters, n_filters * 2, 4, 2, 1, bias=False),
            nn.InstanceNorm2d(n_filters * 2),
            nn.LeakyReLU(0.2),
            nn.Conv2d(n_filters * 2, n_filters * 4, 3, 2, 1, bias=False),
            nn.InstanceNorm2d(n_filters * 4),
            nn.LeakyReLU(0.2),
            nn.Conv2d(n_filters * 4, 1, 4, 2, 1, bias=False),
            nn.Sigmoid()
        )
        return self
```

```

        nn.InstanceNorm2d(n_filters*4),
        nn.LeakyReLU(0.2),
        nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
        nn.Sigmoid()
    )

def forward(self, input):
    output = self.network(input)
    return output.view(-1, 1).squeeze(0)

gen_model = make_generator_network_wgan(
    z_size, n_filters
).to(device)
disc_model = DiscriminatorWGAN(n_filters).to(device)
g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0002)
d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)

```

Теперь напишем функцию для подсчёта GP компоненты:

```

lambda_gp = 10.0

from torch.autograd import grad as torch_grad
def gradient_penalty(real_data, generated_data):
    batch_size = real_data.size(0)

    # посчитаем интерполяцию:
    alpha = torch.rand(real_data.shape[0], 1, 1, 1,
                       requires_grad=True, device=device)
    interpolated = alpha * real_data + (1 - alpha) * generated_data

    # вероятности для этих примеров:
    proba_interpolated = disc_model(interpolated)

    # посчитаем градиенты вероятностей:
    gradients = torch_grad(
        outputs=proba_interpolated, inputs=interpolated,
        grad_outputs=torch.ones(proba_interpolated.size(), device=device),
        create_graph=True, retain_graph=True
    )[0]

    gradients = gradients.view(batch_size, -1)
    gradients_norm = gradients.norm(2, dim=1)
    return lambda_gp * ((gradients_norm - 1)**2).mean()

```

Теперь нужны новые версии функций тренировки для WGAN:

```

def d_train_wgan(x):
    disc_model.zero_grad()

    batch_size = x.size(0)
    x = x.to(device)

    d_real = disc_model(x)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)
    d_generated = disc_model(g_output)
    d_loss = d_generated.mean() - d_real.mean() +\
        gradient_penalty(x.data, g_output.data)
    d_loss.backward()
    d_optimizer.step()
    return d_loss.data.item()

def g_train_wgan(x):
    gen_model.zero_grad()

    batch_size = x.size(0)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)

    d_generated = disc_model(g_output)
    g_loss = -d_generated.mean()

    g_loss.backward()
    g_optimizer.step()
    return g_loss.data.item()

```

Теперь напишем цикл тренировки:

```

epoch_samples_wgan = []
lambda_gp = 10.0

```

```

num_epochs = 100
torch.manual_seed(1)
critic_iterations = 5
for epoch in range(1, num_epochs+1):
    gen_model.train()
    d_losses, g_losses = [], []
    for i, (x, _) in enumerate(mnist_dl):
        for _ in range(critic_iterations):
            d_loss = d_train_wgan(x)
            d_losses.append(d_loss)
            g_losses.append(g_train_wgan(x))
    print(f'Epoch {epoch:03d} | D Loss >> {torch.FloatTensor(d_losses).mean():.4f}')
    gen_model.eval()
    epoch_samples_wgan.append(
        create_samples(
            gen_model, fixed_z
        ).detach().cpu().numpy()
    )
    if epoch % 5 == 0:
        ask = input()
        if ask == "break":
            break
# Epoch 001 | D Loss >> -0.3610
# Epoch 002 | D Loss >> -0.5563
# ...

```

Посмотрим на качество изображений (код такой же):



Мы видим небольшое улучшение по сравнению с последними результатами. Из-за состязательной природы GAN моделей их довольно тяжело тренировать. Бывает, что генератор застревает в маленькой подобласти и учится генерировать похожие примеры, это называется **mode collapse**. Например, генератор выдаёт только единицы. Есть подход **mini-batch discrimination**, в нём мы позволяем дискриминатору сравнивать примеры в разных батчах. Есть ещё одна техника для стабилизации – *feature matching*. В feature matching мы вносим небольшую модификацию в целевую функцию генератора, добавляя дополнительный член, который минимизирует разницу между исходным и синтезированным изображениями на основе промежуточных представлений (feature maps) дискриминатора. Во время тренировки GAN может застрять в нескольких mode и прыгать между ними. Чтобы этого избежать, можно хранить старые примеры и отдавать их дискриминатору. Это техника называется *experience replay*. Интересный факт, что GANs можно расширить на semi-supervised и supervised задачи. Например, **conditional GAN (cGAN)** использует метки, чтобы генерировать изображения в зависимости от их значения, т.е.  $\tilde{x} = G(z|y)$ . Например, можно генерировать числа от 0 до 9 выборочно. С помощью cGANs можно делать image-to-image translation, т.е. учиться конвертировать изображения из одной области в другую. В этом контексте есть интересная работа Pix2Pix, в ней, кстати, дискриминатор выдаёт несколько предсказаний на разные области в изображении, в отличие от нашего случая, где на изображение выдавалась одна вероятность. CycleGAN – другая интересная работа, построенная поверх cGAN, она также нужна для перевода изображений. Однако здесь, в отличие от cGANs, картинки из разных областей тренируются отдельно, нет парного соответствия. С помощью CycleGAN можно, например, менять время года на картинке, можно сделать из лошади зебру. В следующей главе мы посмотрим на graph neural networks. Мы всё это время работали с табличными данными и картинками. Эти же сети работают с графиками, это полезно в биологии, инженерных и социальных задачах. Популярный пример – молекулы, состоящие из атомов, соединенных ковалентными связями.

## 18 GNNs in Graph Structured Data

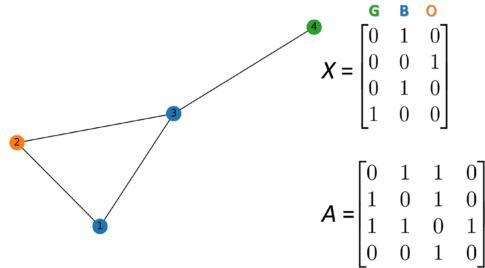
В этой главе мы будем обсуждать модели, которые работают с графиками, они называются **graph neural networks (GNNs)**. GNNs применяются во многих областях: text classification, recommender systems, traffic forecasting, drug discovery. Конечно пройти всё невозможно, поэтому мы обсудим общие идеи и напишем базовую реализацию. Так же мы представим **PyTorch Geometric** библиотеку, с помощью которой можно работать с графиками, в ней так же есть множество различных слоёв.

## 18.1 Introduction to graph data

В этой части обсуждались базовые понятия в теории графов, я их не буду здесь указывать в силу их простоты.

## 18.2 Understanding graph convolutions

Мы рассмотрим свёртку графа – ключевая компонента для GNNs. Плюс свёртки в CNNs был в её локальности, т.е. соседние пиксели к элементу важнее для него, чем отдалённые. Здесь тоже самое, вершины на расстоянии одного ребра важнее для нас, чем, например, на расстоянии пяти рёбер. Важное свойство графа в его инвариантности к перенумерованию вершин, т.е. не зависимо от номеров вершин и их порядка, структура графа остаётся такой же. Из этого следует, что любая свёртка также должна быть инвариантна. Свёрточный подход для нас очень важен, т.к. графы в большинстве датасетов имеют разные размеры. Вот для изображений можно найти датасеты с большим кол-вом картинок одинакового разрешения. Мы решали такую задачу с помощью fc слоёв. Давайте реализуем базовую графовую свёртку. Рассмотрим вот такой простой граф:



Одна из самых крутых библиотек для работы с графиками – это NetworkX, установим её (`pip install networkx`) и попробуем построить график с картинки:

```
! pip install networkx
import numpy as np
import networkx as nx

G = nx.Graph()
# hex коды для цветов
blue, orange, green = "#1f77b4", "#ff7f0e", "#2ca02c"
G.add_nodes_from([
    (1, {"color": blue}),
    (2, {"color": orange}),
    (3, {"color": blue}),
    (4, {"color": green})
])
G.add_edges_from([(1,2), (2,3), (1,3), (3,4)])
A = np.asarray(nx.adjacency_matrix(G).todense())
print(A)
# [[0 1 1 0]
#  [1 0 1 0]
#  [1 1 0 1]
#  [0 0 1 0]]

def build_graph_color_label_representation(G, mapping_dict):
    one_hot_ids = np.array([mapping_dict[v] for v in
                           nx.get_node_attributes(G, 'color').values()])
    one_hot_encoding = np.zeros(
        (one_hot_ids.size, len(mapping_dict)))
    one_hot_encoding[
        np.arange(one_hot_ids.size), one_hot_ids] = 1
    return one_hot_encoding

X = build_graph_color_label_representation(
    G, {green: 0, blue: 1, orange: 2})
print(X)
# [[0.  1.  0.]
#  [0.  0.  1.]
#  [0.  1.  0.]
#  [1.  0.  0.]]

color_map = nx.get_node_attributes(G, 'color').values()
nx.draw(G, with_labels=True, node_color=color_map)
```

Граф будет выглядеть ровно так, как это показано на предыдущей картинке. В контексте свёрток можно считать, что каждый ряд в  $X$  – это embedding информации, которая хранится в вершине.

Свёртка обновляет эти кодировки на основе кодировок соседей. Для нашего примера реализации свёртки свёртка будет считаться так:

$$x'_i = x_i W_1 + \sum_{j \in N(i)} x_j W_2 + b$$

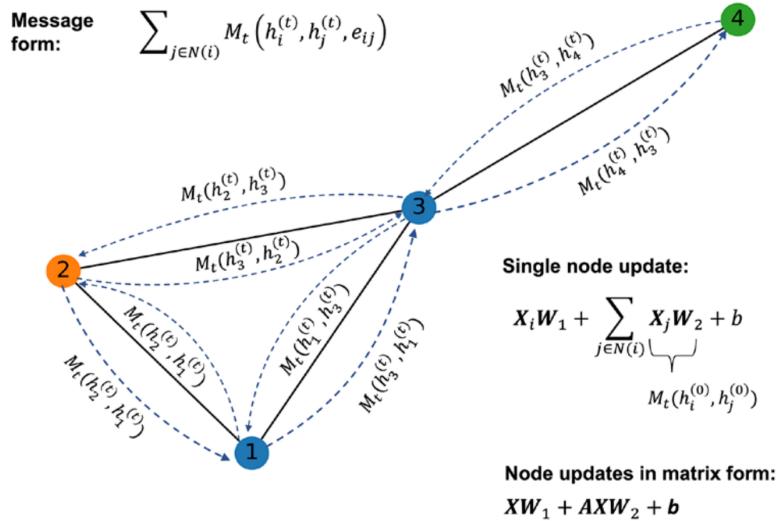
Здесь  $x'_i$  – это новое значение вершины  $i$ ,  $W_1$  и  $W_2$  – это весовые матрицы размера  $f_{in} \times f_{out}$ ,  $b$  – это bias вектор размера  $f_{out}$ . Эти две матрицы можно рассматривать как фильтры, где каждая колонка – это отдельный фильтр. Одна свёртка сможет уловить все зависимости, если они локальны. Если же в графе далёкие зависимости, то нужно использовать stacking свёртки (т.е. просто применим несколько раз). Может быть непонятно, как реализовывать сумму по соседям в графе? Для этого мы будем использовать матрицу смежности, формулу можно переписать так:  $XW_1 + AXW_2$ . Напишем код для прямого хода:

```
f_in, f_out = X.shape[1], 6
W_1 = np.random.rand(f_in, f_out)
W_2 = np.random.rand(f_in, f_out)
h = np.dot(X, W_1) + np.dot(np.dot(A, X), W_2)
```

В конце концов, мы хотим, чтобы свёртка обновляла значения в вершинах на основе структурной информации из матрицы  $A$ . Есть много способов это сделать, они применяются во многих типах свёрток, которые придумали. Есть **message-passing** framework для работы с разными свёртками. В нём каждая вершина имеет своё скрытое состояние  $h_i^{(t)}$ , где  $i$  – номер вершины, а  $t$  – это момент времени,  $h_i^{(0)} = X_i$ . Любую свёртку можно разбить на *message-passing* фазу и *node update* фазу.  $N(i)$  – соседи вершины  $i$  (для ор. графов – это рёбра, которые имеют конец в  $i$ ). Первая фаза считается так:

$$m_i = \sum_{j \in N(i)} M_t(h_i^{(t)}, h_j^{(t)}, e_{ij})$$

Здесь  $M_t$  – это *message function*, в нашем примере  $M_t = h_i^{(t)} W_2$ . Вторая фаза с функцией обновления  $U_t$  – это  $h_i^{(t+1)} = U_t(h_i^{(t)}, m_i)$ , в нашем примере – это  $h_i^{(t+1)} = h_i^{(t)} W_1 + m_i + b$ . Посмотрим на картинку, чтобы лучше понять *message-passing* идею и свёртку в принципе:



В следующей части давайте встроим эту свёртку в GNN модель, реализованную в PyTorch.

### 18.3 Implementing a GNN in PyTorch from scratch

Начнём с того, что объявим `NodeNetwork` модель, потом постепенно будем дополнять её деталями. См. код:

```
import networkx as nx
import torch
from torch.nn.parameter import Parameter
import numpy as np
import math
import torch.nn.functional as F

class NodeNetwork(torch.nn.Module):
    def __init__(self, input_features):
```

```

super().__init__()
self.conv_1 = BasicGraphConvolutionLayer(
    input_features, 32)
self.conv_2 = BasicGraphConvolutionLayer(32, 32)
self.fc_1 = torch.nn.Linear(32, 16)
self.out_layer = torch.nn.Linear(16, 2)

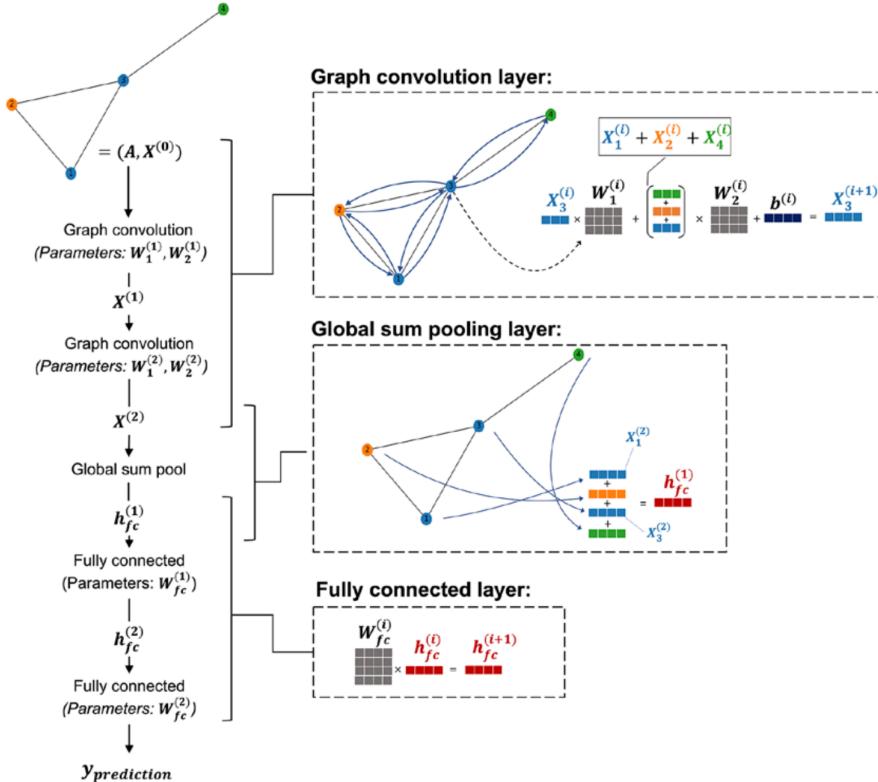
def forward(self, X, A, batch_mat):
    x = F.relu(self.conv_1(X, A))
    x = F.relu(self.conv_2(x, A))
    output = global_sum_pool(x, batch_mat)
    output = self.fc_1(output)
    output = self.out_layer(output)
    return F.softmax(output, dim=1)

```

Модель `NodeNetwork` можно записать в виде 4 шагов:

1. Сделаем две свёртки (`self.conv_1` и `self.conv_2`).
2. Объединим все кодировки вершин через `global_sum_pool`.
3. Прогоним всё это через fc слои.
4. Выведем class-membership вероятности, используя `softmax`.

См. картинку, на ней показаны все шаги:



Теперь надо обсудить саму свёртку и global pooling слой. Начнём с слоя для свёртки, см. код:

```

class BasicGraphConvolutionLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.W2 = Parameter(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))
        self.W1 = Parameter(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))

        self.bias = Parameter(torch.zeros(
            out_channels, dtype=torch.float32))

    def forward(self, X, A):
        potential_msgs = torch.mm(X, self.W2)
        propagated_msgs = torch.mm(A, potential_msgs)

```

```
root_update = torch.mm(X, self.W1)
output = propagated_msgs + root_update + self.bias
return output
```

Для примера применим этот слой для графа из первой части этой главы:

```
print('X.shape:', X.shape) # (4, 3)
print('A.shape:', A.shape) # (4, 4)

basiclayer = BasicGraphConvolutionLayer(3, 8)
out = basiclayer(
    X=torch.tensor(X, dtype=torch.float32),
    A=torch.tensor(A, dtype=torch.float32)
)

print('Output shape:', out.shape) # ([4, 8])
```

---

Краткое содержание.

16. Attention, self-attention и scaled dot-product attention механизмы, transformer архитектура, (masked) multi-head attention механизм, positional encoding, layer normalization, GPT, BERT, BART, пример использования DistilBERT модели на IMDb датасете, Trainer API
17. GANs, autoencoders, variational autoencoders (VAEs), leaky ReLU, deep convolutional GANs (DCGANs), Wasserstein GANs (WGANs), transposed convolution, batch normalization (или просто BatchNorm), Wasserstein-1 расстояние, KL, JS, TV и EM меры, WGAN-GP, mode collapse, другие виды GANs
18. GNNs, PyTorch Geometric библиотека, NetworkX, свёртка, message-passing framework