# Оптимизация нейронных сетей

Как было рассмотрено на предыдущем занятии, каждый слой нейронной сети из нашего первого примера преобразует данные следующим образом:

**output = relu(dot(W, input) + b)**

В этом выражении W и b — тензоры, являющиеся атрибутами слоя. Они называются весами, или обучаемыми параметрами слоя (атрибуты kernel и bias соответственно). Эти веса содержат информацию, извлеченную сетью из обучающих данных.

Первоначально эти весовые матрицы заполняются небольшими случайными значениями (этот шаг называется случайной инициализацией). Начальные представления не несут никакого смысла, но они служат начальной точкой. Далее, на основе сигнала обратной связи, происходит постепенная корректировка весов. Эта постепенная корректировка, которая также называется обучением, составляет суть машинного обучения.

Ниже перечислены шаги, выполняемые в так называемом цикле обучения, который повторяется столько раз, сколько потребуется:

1. Извлекается пакет обучающих экземпляров x и соответствующих целей y .
2. Сеть обрабатывает пакет x (этот шаг называется прямым проходом) и получает пакет предсказаний y_pred.
3. Вычисляются потери сети на пакете, дающие оценку несовпадения между y_pred и y.
4. Корректируются веса сети так, чтобы немного уменьшить потери на этом пакете.

В конечном итоге получается сеть, имеющая очень низкие потери на тренировочном наборе данных: малое несовпадение предсказаний y_pred с ожидаемыми целями y . Сеть «научилась» отображать входные данные в правильные конечные значения. В процессе обучения шаг 1 по сути это операция ввода/вывода. Шаг 2 и 3 это набор операций с тензорами. Наиболее сложный шаг – это шаг 4, корректировка весов сети. Разберемся, как по весам сети понять какой и насколько вес должен изменяться, чтобы модель обучалась.

Градиент — это производная операции с тензором, обобщение понятия производной на функции с многомерными входными данными, то есть на функции, принимающие на входе тензоры.

Рассмотрим входной вектор x , матрицу W , цель y и функцию потерь loss . Вы можете с помощью W вычислить приближение к цели y_pred и определить потери или несоответствие, между кандидатом y_pred и целью y:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

Если входные данные x и y зафиксированы, тогда это можно интерпретировать как функцию, отображающую значения W в значения потерь:

```
loss_value = f(W)
```

Допустим, что W0 — текущее значение W . Тогда производной функции f в точке W0 будет тензор gradient(f)(W0) с той же формой, что и W , в котором каждый элемент gradient(f) (W0)[i, j] определяет направление и величину изменения в loss_value , наблюдаемого при изменении W0[i, j] . Тензор gradient(f)(W0) — это градиент функции f(W) = loss_value в W0.

Производную некоторой функции g(x) от единственного аргумента можноинтерпретировать как наклон кривой этой функции. Аналогично gradient(f)(W0) можно интерпретировать как тензор, описывающий кривизну f(W) в окрестностях W0. Соответственно, как и в случае с функцией g(x), значение которой можно уменьшить, немного сместив x в направлении, противоположном производной, функцию f(W) тензора также можно уменьшить, сместив W в направлении, противоположном градиенту: например, W1 = W0 - step * gradient(f)(W0) (где step — небольшой по величине множитель). Это означает, что для снижения нужно идти против кривизны. Обратите внимание: множитель step необходим, потому что gradient(f)(W0) лишь аппроксимирует кривизну в окрестностях W0 , поэтому очень нежелательно уходить слишком далеко от W0.

## Стохастический градиентный спуск

Как известно, минимум функции — это точка, где производная равна 0. То есть остается только найти все точки, где производная обращается в 0, и выяснить, в какой из этих точек функция имеет наименьшее значение.

Применительно к нейронным сетям это означает аналитический поиск комбинации значений весов, при которых функция потерь будет иметь наименьшее значение. Этого можно добиться, решив уравнение gradient(f)(W) = 0 для W . Это полиномиальное уравнение с N переменными, где N — количество весов в сети. Решить уравнение для случая N = 2 или N = 3 не составляет труда, но превращается в практически неразрешимую задачу для нейронных сетей, в которых количество параметров редко бывает меньше нескольких тысяч и часто достигает нескольких десятков миллионов.

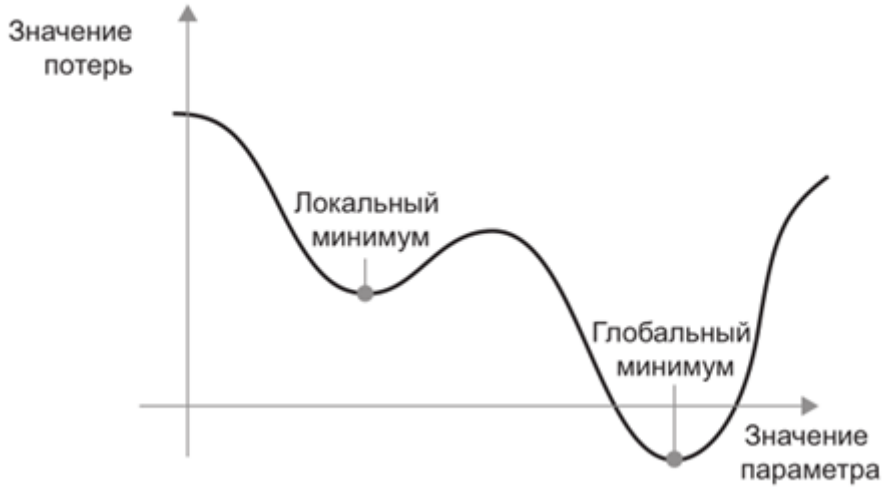
Поэтому на практике используется алгоритм из четырех шагов, представленный в начале этого занятия: вы можете понемногу изменять параметры, опираясь на текущие значения потерь в случайном пакете данных. Поскольку функция дифференцируема, можно вычислить ее градиент, который позволяет эффективно реализовать шаг 4, который называется обратный проход. И в итоге в алгоритме 4 шаг разбивается на 2:

1. Вычисляется градиент потерь для параметров сети (обратный проход).
2. Параметры корректируются на небольшую величину в направлении, противоположном градиенту, например W -= step * gradient , и тем самым снижаются потери.

Выбор разумной величины шага step имеет большое значение. Если выбрать его слишком маленьким, спуск потребует большого количества итераций и может застрять в локальном минимуме. Если шаг будет слишком большим, ваши корректировки могут приобретать нецеленаправленный характер и приводить в случайные точки на кривой.

По сути, описанный выше алгоритм является стохастическим градиентным спуском на небольших пакетах (mini-batch stochastic gradient descent, minibatch SGD). Термин «стохастический» отражает тот факт, что каждый пакет данных выбирается случайно (в науке слово «стохастический» считается синонимом слова «случайный»). На самом деле истинный SGD это когда в каждой итерации используется единственный образец и цель, а не весь пакет данных.

Существует также множество вариантов стохастического градиентного спуска, которые отличаются тем, что при вычислении следующих приращений весов принимают в учет не только текущие значения градиентов, но и предыдущие приращения. Примерами могут служить такие алгоритмы, как SGD с импульсом, Adagrad, RMSProp и некоторые другие. Эти варианты известны как методы оптимизации, или оптимизаторы. В частности, внимания заслуживает идея импульса, которая используется во многих этих вариантах. Импульс вводится для решения двух проблем SGD: невысокой скорости сходимости и попадания в локальный минимум. Рассмотрим функцию, которая имеет форму, представленную на рисунке




Как видите, для значения данного параметра имеется локальный минимум: движение из этой точки влево или вправо повлечет увеличение потери. Если корректировка рассматриваемого параметра осуществляется методом градиентного спуска с маленьким шагом обучения, тогда процесс оптимизации может застрять в локальном минимуме, не найдя пути к глобальному минимуму. Таких проблем можно избежать, если использовать идею импульса, заимствованную из физики. Вообразите, что процесс оптимизации — это маленький шарик, катящийся вниз по кривой потерь. Если шарик имеет достаточно высокий импульс, он не застрянет в мелком овраге и окажется в глобальном минимуме. Импульс реализуется путем перемещения шарика на каждом шаге, исходя не только из текущей величины наклона (текущего ускорения), но также из текущей скорости (набранной в результате действия силы ускорения на предыдущем шаге). На практике это означает, что приращение параметра w определяется не только по текущему значению градиента, но также по величине предыдущего приращения параметра:

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum + learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

## Объединение производных: алгоритм обратного распространения ошибки

В предыдущем алгоритме мы произвольно предположили, что, если функция дифференцируема, мы можем явно вычислить ее производную. На практике функция нейронной сети состоит из множества последовательных операций с тензорами, объединенных в одну цепочку, каждая из которых имеет простую, известную производную. Например, пусть есть сеть f , состоящая из трех операций с тензорами a , b и c и весовыми матрицами W1 , W2 и W3 :

**f(W1, W2, W3) = a(W1, b(W2, c(W3)))**

Формула сообщает нам, что такую цепочку функций можно получить с использованием следующего тождества, которое называется цепным правилом: f(g(x)) = f'(g(x)) * g'(x) . Применение цепного правила к вычислению значений градиента нейронной сети приводит к алгоритму, который называется обратным распространением ошибки (Backpropagation), или обратным дифференцированием. Обратное распространение начинается с конечного значения потери и

движется в обратном направлении, от верхних слоев к нижним, применяя цепное правило для вычисления вклада каждого параметра в значение потери.

С другими методами обучения можно ознакомиться по данной ссылке

# Построение модели в функциональном виде

Ранее, построение моделей происходило в последовательным виде, и выглядело следующим образом:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

То есть создавался объект model класса Sequential, а затем происходило последовательное добавление слоев функцией add.

При таком подходе мы не можем создать модель с несколькими входными или выходными слоями. Поэтому, в библиотеке Keras есть возможность создавать модели в функциональном виде. И созданная модель выше будет создаваться следующим образом:

```
from keras.layers import Input, Dense
from keras.models import Model

# Данный слой просто возвращает тензор
inputs = Input(shape=(10000,))

# Вызов слоя принимает тезор и возвращает тензор
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(46, activation='softmax')(output_2)

# Создание модели, которая включает
# Один слой типа Input и 3 слоя типа Dense
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Все модели могут вызываться, как слои. С функциональным API легко использовать обученные модели: можно обрабатывать любую модель, как если бы она была слоем, вызывая ее для тензора. Обратите внимание, что, вызывая модель, вы не просто повторно используете архитектуру модели, вы также повторно используете ее веса.

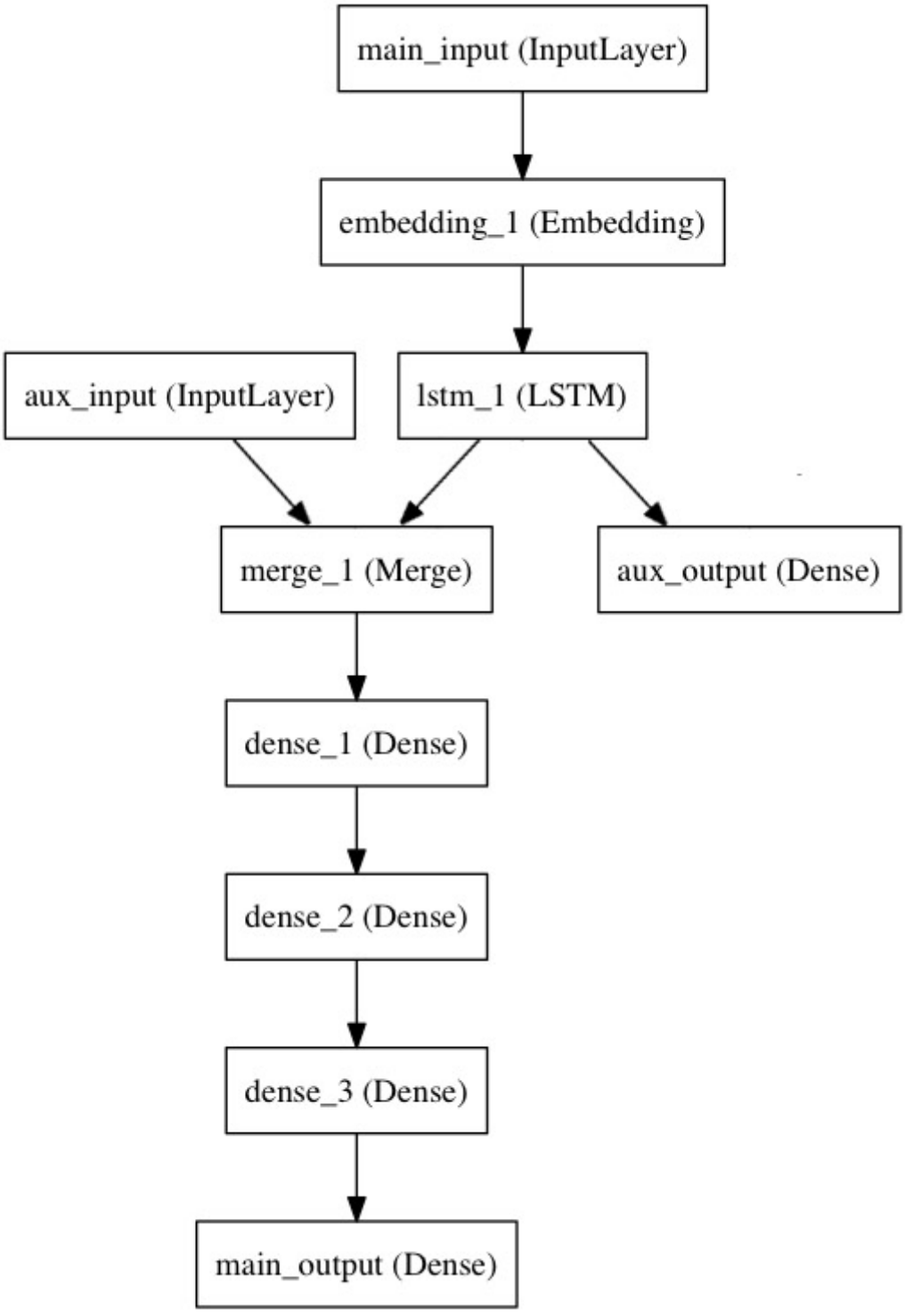
```
x = Input(shape=(10000,))
# Это будет работать, и возвращать вектор из 46 элементов пройденный через модель описанную выше
y = model(x)
```

## Пример модели с несколькими входами и выходами

Вот хороший пример использования функционального API: модели с несколькими входами и выходами. Функциональный API позволяет легко манипулировать большим количеством переплетенных потоков данных.

Давайте рассмотрим следующую модель. Мы стремимся предсказать, сколько ретвитов и лайков будут получать заголовки новостей в Twitter. Основным входом в модель будет сам заголовок в виде последовательности слов, но, чтобы оживить ситуацию, наша модель также будет иметь вспомогательный вход, получая дополнительные данные, такие как время суток, когда заголовок был опубликован, и т. Д. Модель также будет контролироваться с помощью двух функций потерь. Использование главной функции потерь ранее в модели является хорошим механизмом регуляризации для глубоких моделей.

Вот так будет выглядеть наша модель:



Давайте реализуем это с помощью функционального API.

Основной вход получит заголовок в виде последовательности целых чисел (каждое целое число кодирует слово). Целые числа будут между 1 и 10000 (словарь из 10000 слов), а последовательности будут длиной 100 слов.

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model
import numpy as np
np.random.seed(0)  # Зададим ядро случайной генерации, чтобы результат повторялся

# Основной вход: будем получать последовательность из 100 целых чисел из диапазона от
1 до 10000.
# Обратите внимание, что мы можем называть любой слой через параметр name.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# Это слой встраивания, который будет кодировать нашу последовательность
# в последовательность вектором размерностью 512.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# LSTM будет преобразовывать последовательность векторов в один вектор
# содержащий информацию о всей последовательности
lstm_out = LSTM(32)(x)
```

Здесь мы вставляем вспомогательные потери, что позволяет плавно обучать LSTM и слой Embedding, даже если основные потери в модели будут намного выше:

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

На этом этапе мы вводим в модель наши вспомогательные входные данные, объединяя их с выходом LSTM:

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# Добавляем набор полносвязных слоев
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# И добавляем слой логистической регрессии на выход
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

Определяем модель с двумя входами и двумя выходами:

```
model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output,
auxiliary_output])
```

Мы компилируем модель и присваиваем вес 0,2 для вспомогательных потерь. Чтобы указать разные потери или веса для каждого отдельного выхода, вы можете использовать список или словарь. Здесь мы передаем одну потерю в качестве аргумента потери, поэтому одна и та же функция будет использоваться на всех выходах:

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', loss_weights=[1., 0.2])
```

Мы можем обучить модель, передав ей списки входных массивов и целевых массивов:

```
headline_data = np.round(np.abs(np.random.rand(12, 100) * 100))
additional_data = np.random.randn(12, 5)
headline_labels = np.random.randn(12, 1)
additional_labels = np.random.randn(12, 1)
model.fit([headline_data, additional_data], [headline_labels, additional_labels],
          epochs=50, batch_size=32)
```

Поскольку наши входы и выходы имеют имена (мы передали им аргумент «name»), мы могли бы также скомпилировать модель с помощью:

```
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output':
'binary_crossentropy'},
              loss_weights={'main_output': 1., 'aux_output': 0.2})

# И обучить как
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': headline_labels, 'aux_output': additional_labels},
          epochs=50, batch_size=32)
```

Чтобы использовать модель для вывода, используйте:

```
model.predict({'main_input': headline_data, 'aux_input': additional_data})
```

Или

```
pred = model.predict([headline_data, additional_data])
```

Про большее количество примеров можно прочитать здесь: **https://keras.io/getting-started/functional-api-guide/**