Пишем простую нейронную сеть с использованием математики и Numpy

А. Алексеев

30 мая 2019 г.

Аннотация

Зачем очередная статья про то, как писать нейронные сети с нуля? Увы, я не смог найти статьи, где были бы описаны теория и код с нуля до полностью работающей модели. Сразу предупреждаю, что тут будет много математики. Я предполагаю, что читатель знаком с основами линейной алгебры, частными производными и хотя бы частично, с теорией вероятностей, а также Python и Numpy. Будем разбираться с полносвязной нейронной сетью и MNIST.

1 Математика. Часть 1 (простая). Forward propagation

Что такое полносвязный слой? Обычно говорят, что-то в духе «Полносвязный слой – это слой, каждый нейрон которого соединён со всеми нейронами предыдущего слоя». Вот только непонятно что такое нейроны, как они соединены, особенно в коде. Сейчас я попробую разобрать это на примере. Вот пусть есть слой из 100 нейронов. Я знаю, что пока не объяснил, что это, но давайте просто представим, что есть 100 нейронов и у них есть вход, куда подаются данные, и выход, откуда они выдают данные. И на вход им подаётся чёрно-белая картинка 28x28 пикселей – всего 784 значения, если растянуть её в вектор. Картинку можно назвать входным слоем. Тогда чтобы каждый из 100 нейронов соединился с каждым «нейроном» предыдущего слоя (то есть картинкой) нужно, чтобы каждый из 100

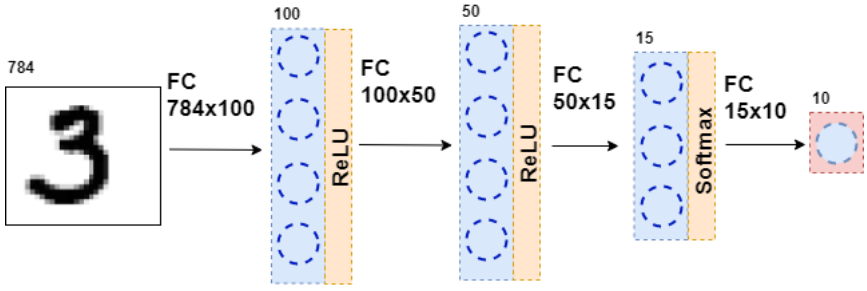
нейронов принял 784 значения исходной картинки. Например, для каждого из 100 нейронов достаточно будет умножить 784 значения картинки на какие-то 784 цифры и сложить между собой, в результате выходит одно число. То есть это и есть нейрон:

$$\begin{aligned} \text{Выход нейрона} = \text{какая-то цифра}_1 \cdot \text{значение картинки}_1 + \\ + \ldots + \text{какая-то цифра}_{784} \cdot \text{значение картинки}_{784} \end{aligned} \tag{1}$$

Тогда получится, что у каждого нейрона есть 784 цифры, а всего этих цифр: (количество нейронов на этом слое) x (количество нейронов на предыдущем слое) = 100x784 = 78,400 цифр. Эти цифры обычно называют весами слоя. Каждый нейрон выдаст свою цифру и в итоге получится 100-мерный вектор, и на самом деле можно записать, что этот 100-мерный вектор получается умножением 784-мерного вектора (нашей исходной картинки) на матрицу весов размером 100x784:

$$\boldsymbol{x}^{100} = W_{100x784} \cdot \boldsymbol{x}^{784} \tag{2}$$

Дальше получившиеся 100 цифр передаются дальше, на функцию активации – некоторую нелинейную функцию. Например, сигмоида, гиперболический тангенс, ReLU и другие. Функция активации обязательно нелинейная, иначе нейронная сеть научится только простым преобразованиям. Затем получившиеся данные вновь подаются на полносвязный слой, но уже с другим количеством нейронов, и вновь на функцию активации. Так происходит несколько раз. Последний слой сети – это слой, который выдаёт ответ. В данном случае, ответ – это информация о том, какая цифра на картинке.



Во время обучения сети необходимо чтобы мы знали, какая цифра изображена на картинке. То есть чтобы датасет был размечен. Тогда можно

использовать ещё один элемент – функцию ошибки. Она смотрит на ответ нейронной сети и сравнивает с настоящим ответом. Благодаря этому нейронная сеть и учится.

2 Общая постановка задачи

Весь датасет – это большой тензор[1] $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]$, где $\boldsymbol{x}_i$ – i-ый объект, например, картинка, которая тоже является тензором. Для каждого объекта есть y_i - правильный ответ на i-м объекте. В этом случае нейронную сеть можно представить как некоторую функцию, которая принимает на вход объект и выдает на нем какой-то ответ:

$$F(\boldsymbol{x}_i) = \hat{y}_i \tag{3}$$

Теперь давайте подробнее рассмотрим функцию $F(\boldsymbol{x}_i)$. Так как нейронная сеть состоит из слоев, то каждый отдельный слой – функция. А значит

$$F(\boldsymbol{x}_i) = f_k(f_{k-1}(\ldots(f_1(\boldsymbol{x}_i)))) = \hat{y}_i \tag{4}$$

То есть, в самую первую функцию – первый слой – подается картинка в виде некоторого тензора. Функция f_1 выдает какой-то ответ – тоже тензор, но уже другой размерности. Этот тензор будет называться внутреннем представлением. Теперь это внутреннее представление подается на вход функции f_2, которая выдает свое внутреннее представление. И так далее, пока функция f_k – последний слой – не выдаст ответ $\hat{y}_i$.

Теперь, задача – обучить сеть – сделать так, чтобы ответ сети совпадал с правильным ответом. Для начала нужно измерить насколько нейронная сеть ошиблась. Измерять это будет функция ошибки $L(\hat{y}_i, y_i)$. Причем накладываем ограничения:

1. $\hat{y}_i \to y_i \Rightarrow L(\hat{y}_i, y_i) \to 0$
2. $\exists\ dL(\hat{y}_i, y_i)$
3. $L(\hat{y}_i, y_i) \geq 0$

Ограничение 2 наложим на все функции слоев f_j – пусть они все будут дифференцируемы. Причем, на самом деле (об этом я умолчал) часть этих

[1] Например скаляр - тензор ранга 0, а вектор - тензор ранга 1

функции зависят от параметров – весов нейронной сети – $f_j(\boldsymbol{x}_i|\boldsymbol{\omega}_j)$. И вся идея заключается в том, чтобы подобрать такие веса, чтобы $\hat{y}_i$ совпадал с y_i на всех объектах датасета. Отмечу, что веса есть не во всех функциях.

Итак, на чем мы остановились? Все функции нейронной сети дифференцируемы, функция ошибки тоже дифференцируема. Вспомним одно из свойств градиента – показывать направление роста функции. Воспользуемся этим, ограничениями 1 и 3, фактом, что

$$L(F(\boldsymbol{x}_i)) = L(f_k(f_{k-1}(\dots(f_1(\boldsymbol{x}_i))))) = L(\hat{y}_i) \tag{5}$$

и тем, что я умею считать частные производные и производные сложной функции. Теперь есть все что нужно, для того чтобы посчитать

$$\frac{\partial L(F(\boldsymbol{x}_i))}{\partial \boldsymbol{\omega_j}} \tag{6}$$

для любого i и j. Эта частная производная показывает направление, в котором нужно изменить $\boldsymbol{\omega_j}$, чтобы увеличить L. Чтобы уменьшить нужно сделать шаг в сторону $-\frac{\partial L(F(\boldsymbol{x}_i))}{\partial \boldsymbol{\omega_j}}$, ничего сложного. Значит процесс обучения сети строится так: несколько раз в цикле проходим по всему датасету (это называется эпоха), для каждого объекта датасета считаем $L(\hat{y}_i, y_i)$ (это называется forward pass) и считаем частную производную ∂L для всех весов $\boldsymbol{\omega_j}$, затем обновляем веса (это называется backward pass).

Замечу, что я еще не ввел никаких конкретных функций и слоев. Если на данном этапе не совсем ясно, что со всем этим делать, предлагаю продолжить чтение – математики станет больше, но теперь она будет идти с примерами.

3 Математика. Часть 2 (сложная с повторениями). Backpropagation

3.1 Функция ошибки

Я начну с конца и выведу функцию ошибки для задачи классификации. Для задачи регрессии вывод функции ошибки хорошо описан в книге «Глубокое обучение. Погружение в мир нейронных сетей».

Для простоты, есть нейронная сеть (NN), которая отделяет фотографии котиков от фотографий собак, и есть набор фотографий кошек и собак, для которых есть правильный ответ y_{true}.

$$NN(picture|\Omega) = y_{pred} \tag{7}$$

Все что я буду делать дальше, очень похоже на метод максимального правдоподобия. Поэтому основная задача – найти функцию правдоподобия. Если опустить подробности, то такую функцию, которая сопоставляет предсказание нейронной сети и правильный ответ, и, если они совпадают, выдает большое значение, если же нет, наоборот. На ум приходит вероятность правильного ответа при заданных параметрах:

$$p(y_{pred} = y_{true}|\Omega) \tag{8}$$

А теперь сделаем некоторый финт, который вроде бы ни от куда не следует. Пусть нейронная сеть выдает ответ в виде двумерного вектора, сумма значений которого равна 1. Первый элемент этого вектора можно назвать мерой уверенности, что на фотографии кот, а второй элемент – мерой уверенности, что на фотографии пёс. Да это же почти вероятности!

$$NN(picture|\Omega) = \begin{bmatrix} p_0 \\ p_1 \end{bmatrix} \tag{9}$$

Теперь функцию правдоподобия можно переписать в виде:

$$p(y_{pred} = y_{true}|\Omega) = p_\Omega(y_{pred})_0^t * (1 - p_\Omega(y_{pred}))_1^t = p_0^{t_0} * p_1^{t_1} \tag{10}$$

Где t_0, t_1 метки верного класса, например, если $y_{true} = cat$, то $t_0 == 1, t_1 == 0$, если $y_{true} = dog$, то $t_0 == 0, t_1 == 1$. Таким образом всегда рассматривается вероятность класса, который должен был быть предсказан нейронной сетью (но не обязательно был предсказан ею). Теперь это можно обобщить на любое количество классов (например, m классов):

$$p(y_{pred} = y_{true}|\Omega) = \prod_0^m p_i^{t_i} \tag{11}$$

Однако, в любом датасете есть много объектов (например, N объектов). Хочется, чтобы на каждом или большинстве объектов нейронная сеть выдавала верный ответ. И для этого нужно перемножить результаты формулы

выше для каждого объекта из датасета.

$$MaximumLikelyhood = \prod_{j=0}^{N} \prod_{i=0}^{m} p_{i,j}^{t_{i,j}} \tag{12}$$

Чтобы получить хорошие результаты, эту функцию нужно максимизировать. Но, во-первых, минимизировать круче, потому что у нас есть стохастический градиентный спуск и все плюшки для него – просто припишем минус, а, во-вторых, с огромным произведением работать затруднительно – логарифмируем.

$$CrossEntropyLoss = -\sum_{j=0}^{N} \sum_{i=0}^{m} t_{i,j} \cdot \log(p_{i,j}) \tag{13}$$

Замечательно! Получилась перекрестная энтропия или, в бинарном случае, logloss. Эту функцию легко считать и еще легче дифференцировать:

$$\frac{\partial CrossEntropyLoss}{\partial p_j} = -\frac{\boldsymbol{t_j}}{\boldsymbol{p_j}} \tag{14}$$

Дифференцировать нужно для алгоритма обратного распространения ошибки. Замечу, что функция ошибки не изменяет размерность вектора. Если, как в случае MNIST, на выходе получается 10-мерный вектор ответов, то и при вычислении производной получится 10-мерный вектор производных. Ещё одна интересная вещь то, что только один элемент производной не будет равен нулю, при котором $t_{i,j} \neq 0$, то есть при правильном ответе. И чем меньше вероятность верного ответа предсказала нейронная сеть на данном объекте, тем больше на нем будет функция ошибки.

3.2 Функции активации

На выходе каждого полносвязного слоя нейронной сети обязательно должна присутствовать нелинейная функция активации. Без неё невозможно обучить содержательную нейронную сеть. Если забежать вперед, то полносвязный слой нейронной сети – это просто умножение входных данных на матрицу весов. В линейной алгебре это называется линейным отображением – некоторая линейная функция. Комбинация линейных функций – тоже линейная функция. Но это значит, что такая функция может аппроксимировать только линейные функции. Увы, это не то, зачем нужны нейронные сети.

3.2.1 Softmax

Обычно эта функция используется на последнем слое сети, так как она превращает вектор с последнего слоя в вектор «вероятностей»: каждый элемент вектора лежит от 0 до 1 и их сумма равна 1. Она не меняет размерность вектора.

$$Softmax_i = \frac{e^{x_i}}{\sum\limits_j e^{x_j}} \tag{15}$$

Теперь перейдем к поиску производной. Так как $\boldsymbol{x}$ – вектор, и в знаменателе всегда присутствуют все его элементы, то при взятии производной получим якобиан:

$$J_{Softmax} = \begin{cases} x_i - x_i \cdot x_j, i = j \\ -x_i \cdot x_j, i \neq j \end{cases} \tag{16}$$

Теперь про backpropagation. От предыдущего слоя (обычно это функция ошибки) приходит вектор производных $\boldsymbol{dz}$. В случае если $\boldsymbol{dz}$ пришел от функции ошибки на MNIST, $\boldsymbol{dz}$ – 10-мерный вектор. Тогда якобиан имеет размерность 10x10. Чтобы получить $\boldsymbol{dz_{new}}$, который пойдет далее, на предыдущий слой (не забываем, что мы идем от конца к началу сети при обратном распространении ошибки), нужно умножить $\boldsymbol{dz}$ на $J_{Softmax}$ (строка на столбец):

$$dz_{new} = \boldsymbol{dz} \times J_{Softmax} \tag{17}$$

На выходе получаем 10-мерный вектор производных $\boldsymbol{dz_{new}}$.

3.2.2 ReLU

$$ReLU(x) = \begin{cases} x, x > 0 \\ 0, x < 0 \end{cases} \tag{18}$$

Массово использовать ReLU стали после 2011 года, когда вышла статья «Deep Sparse Rectifier Neural Networks». Однако, такая функция была известна и ранее. К ReLU применимо такое понятие, как «сила активации» (подробнее об этом можно почитать в книге «Глубокое обучение. Погружение в мир нейронных сетей»). Но главная особенность, которая делает ReLU привлекательнее других функций активации – простое вычисление

производной:

$$d(ReLU(x)) = \begin{cases} 1, x > 0 \\ 0, x < 0 \end{cases} \tag{19}$$

Таким образом ReLU вычислительно эффективнее других функций активации (сигмоида, гиперболический тангенс и др.).

3.3 Полносвязный слой

Теперь время обсудить полносвязный слой. Наиболее важный из всех остальных, потому что именно в этом слое находятся все веса, которые и нужно настроить для того, чтобы нейронная сеть работала хорошо. Полносвязный слой это просто-напросто матрица весов:

$$W = |w_{i,j}|$$

Новое внутреннее представление получается, когда матрица весов умножается на столбец входных данных:

$$\boldsymbol{x}_{new} = W \cdot \boldsymbol{x} \tag{20}$$

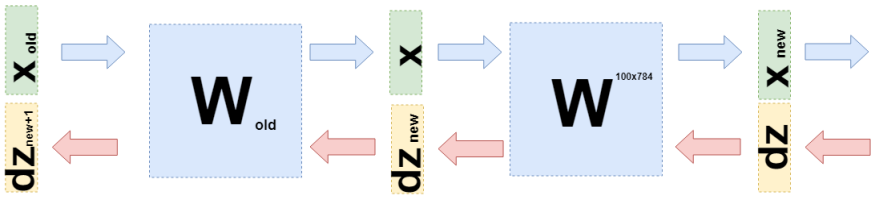
Где $\boldsymbol{x}$ имеет размер input_shape, а x_{new} – output_shape. Например, $\boldsymbol{x}$ – 784-мерный вектор, а $\boldsymbol{x}_{new}$ - 100-мерный вектор, тогда матрица W имеет размер 100x784. Получается, что на этом слое находится 100x784 = 78,400 весов.

При обратном распространении ошибки нужно взять производную по каждому весу этой матрицы. Упростим задачу и возьмем только производную по $w_{1,1}$. При умножении матрицы и вектора первый элемент нового вектора $\boldsymbol{x}_{new}$ равен $x_{new\ 1} = w_{1,1} \cdot x_1 + ... + w_{1,784} \cdot x_{784}$, а производная $x_{new\ 1}$ по $w_{1,1}$ будет просто x_1, нужно всего лишь взять производную от суммы выше. Аналогично происходит для всех остальных весов. Но это еще на алгоритм обратного распространения ошибки, пока это просто матрица производных. Нужно вспомнить, что от следующего слоя на этот (ошибка идет от конца к началу) приходит 100-мерный вектор градиента $d\boldsymbol{z}$. Первый элемент этого вектора dz_1 будет умножаться на все элементы матрицы производных, которые «участвовали» в создании $x_{new\ 1}$, то есть на $x_1, x_2, ..., x_{784}$.

Аналогично и остальные элементы. Если перевести это на язык линейной алгебры, то это записывается так:

$$\frac{\partial L}{\partial W} = (d\boldsymbol{z},\ dW) = \begin{pmatrix} dz_1 \cdot \boldsymbol{x} \\ ... \\ dz_{100} \cdot \boldsymbol{x} \end{pmatrix}_{100} \tag{21}$$

На выходе получается матрица 100x784.



Теперь нужно понять, что же передавать на предыдущий слой. Для этого и для большего понимания что вообще сейчас произошло, я хочу записать то, что случилось при взятии производных на этом слое чуть-чуть другим языком, уйти от конкретики «что на что умножается» к функциям (опять). Когда я хотел настроить веса, то я хотел взять производную функции ошибки по этим весам: $\frac{\partial L}{\partial W}$. Выше было показано, как брать производные по функциям ошибки и функциям активации. Поэтому можно рассмотреть такой случай (в $d\boldsymbol{z}$ уже сидят все производные от функции ошибки и функций активации):

$$\frac{\partial L}{\partial W} = d\boldsymbol{z} \cdot \frac{\partial \boldsymbol{x}_{new}(W)}{\partial W} \tag{22}$$

Так можно сделать, потому что можно рассмотреть $\boldsymbol{x}_{new}$, как функцию от W: $\boldsymbol{x}_{new} = W \cdot \boldsymbol{x}$. Можно подставить это в формулу выше:

$$\frac{\partial L}{\partial W} = d\boldsymbol{z} \cdot \frac{\partial W \cdot \boldsymbol{x}}{\partial W} = d\boldsymbol{z} \cdot E \cdot \boldsymbol{x} \tag{23}$$

Где E матрица состоящая из единиц (НЕ единичная матрица). Теперь когда нужно взять производную от предыдущего слоя (пусть для простоты выкладок это тоже будет полносвязный слой, но в общем случае это ничего не меняет), то нужно рассмотреть $\boldsymbol{x}$, как функцию от предыдущего слоя

$\boldsymbol{x}(W_{old})$:

$$\begin{aligned}\frac{\partial L}{\partial W_{old}} = d\boldsymbol{z} \cdot \frac{\partial \boldsymbol{x}_{new}(W)}{\partial W_{old}} = d\boldsymbol{z} \cdot \frac{\partial W \cdot \boldsymbol{x}(W_{old})}{\partial W_{old}} = \\ = d\boldsymbol{z} \cdot \frac{\partial W \cdot W_{old} \cdot \boldsymbol{x}_{old}}{\partial W_{old}} = d\boldsymbol{z} \cdot W \cdot E \cdot \boldsymbol{x}_{old} = \\ = d\boldsymbol{z}_{new} \cdot E \cdot \boldsymbol{x}_{old}\end{aligned} \tag{24}$$

Именно $d\boldsymbol{z}_{new} = d\boldsymbol{z} \cdot W$ и нужно отправить на предыдущий слой.

4 Код: собираем все вместе

Чтобы не загромождать документ, я выложил код на гитхаб. Код для всех компонентов нейронной сети.