МОСКОВСКИЙ ИНСТИТУТ ЭЛЕКТРОННОЙ ТЕХНИКИ
Институт системной и программной инженерии
и информационных технологий (Институт СПИНТех)

Лабораторный практикум по курсу
"Свёрточные нейронные сети в компьютерном зрении"
(02/20 – 06/20)

Лабораторная работа 4

Многослойный перцептрон и распознавание рукописных цифр на TensorFlow.

На этом занятии компьютерного практикума вы изучите главную составляющую любой нейронной сети – перцептрон, а также то, как перцептроны соединяются в сети.

Прежде чем приступить собственно к программированию, настоятельно рекомендуется ознакомиться с материалом лекций, дополнительными материалами, имеющими отношение к работе перцептронов, функциям активации нейронов, глубоким сетям, а также с языком программирования Python.

Файлы, включенные в задание:

lab 4 multilayer_perceptron.ipynb – программа, реализующая пошаговое обучение нейронной модели на примере классической задачи распознавания рукописных цифр.

1. Когда появились первые нейронные сети

По меркам истории искусственного интеллекта нейронные сети – это одна из старейших, самых первых конструкций, появившаяся ещё до знаменитого эссе Тьюринга *Computing Machinery and Intelligence*, до Дартмутского семинара, до того, как появился собственно термин «искусственный интеллект».

По-видимому, первой работой, предлагающей математическую модель нейрона, была статья Уоррена Маккаллоха (Warrem McCulloch) и Уолтера Питтса (Walter Pitts) «Логическое исчисление идей, относящихся к нервной активности» (A Logical Calculus of the Ideas Immanent in Nervous Activity), опубликованная в 1943 году. Авторы отмечают, что из-за бинарной природы нейронной активности (нейрон либо «включен», либо «выключен», практически без промежуточных состояний) нейроны удобно описывать в терминах пропозициональной логики, а для нейронных сетей разрабатывают целый логический аппарат, формализующий ациклические графы. Сама конструкция искусственного нейрона, который у Маккаллоха и Питтса называется Threshold Logic Unit (TLU), или Linear Threshold Unit, получилась очень

современной: линейная комбинация входов, которая затем поступает на вход нелинейности в виде «ступеньки», сравнивающей результат с некоторым порогом (threshold).

Статья Маккаллоха и Питтса прошла практически незамеченной среди нейробиологов, но создатель кибернетики Норберт Винер сразу понял перспективы искусственных нейронных сетей и идеи о том, как мышление может самопроизвольно возникать из таких простых логических элементов. Винер познакомил Маккаллоха и Питтса с фон Нейманом, и они впятером, вместе с когнитивистом Джеромом Леттвином, стали работать над тем, как адаптировать статистическую механику для моделирования мышления, а потом и построить работающий компьютер на основе моделирования нейронов.

Завершая этот краткий исторический экскурс, упомянем один подробный обзор современной истории нейронных сетей, работу Юргена Шмидхубера (Juergen Schmidhuber: Deep Learning in Neural Networks). Несмотря на то, что это очень плотный и довольно сухой текст, он выгодно отличается от многих других обзоров и исторических очерков тем, что анализирует именно *идеи*: как развивались не просто нейронные сети в целом, а конкретные конструкции и алгоритмы оптимизации, чего они достигали на каждом этапе развития, и где сейчас все эти идеи применяются. Кроме того, автор старается проследить историю каждой идеи до самых её истоков, до первого появления в литературе, зачастую с неожиданными результатами.

2. **Как работает перцептрон**

Итак, мы наконец-то приступаем к рассмотрению математики нейронных сетей и начнём мы с первой конструкции – *линейного перцептрона*, описанного ещё Фрэнком Розенблаттом в 1950-х годах.

По сути своей перцептрон Розенблатта – это линейная модель *бинарной* классификации, когда все объекты в тренировочной выборке помечены одной из двух меток (скажем, +1 или -1), и задача состоит в том, чтобы научиться расставлять эти метки и у новых, ранее не виденных объектов. А «линейная модель» означает, что в результате обучения модель разделит всё пространство входов на две части гиперплоскостью: правило принятия решения о том, какую метку ставить, будет линейной функцией от входящих признаков.

Для простоты будем считать, что каждый вход представляет собой вектор вещественных чисел $\boldsymbol{x} = (x_1, x_2, \dots, x_d) \in R^d$, и входы в тренировочном множестве снабжены известными выходами $y(\boldsymbol{x}) \in \{-1,1\}$. Тогда «линейная модель» означает, что мы будем искать веса $w_0, w_1, \dots, w_d \in R^d$, чтобы знак линейной функции

$$sign(w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d)$$

как можно чаще совпадал с правильным ответом $y(x)$. Чтобы не тащить везде за собой свободный член w_0, введём в вектор $\boldsymbol{x}$ лишнюю размерность: $x = (1, x_1, x_2, \dots, x_d)$, и $\boldsymbol{x} \in R^{d+1}$; тогда $w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d$ можно считать скалярным произведением $\boldsymbol{w}^T \boldsymbol{x}$ вектора весов $\boldsymbol{w} = (w_0, w_1, \dots, w_d)$ на входной вектор $\boldsymbol{x}$.

Прежде, чем обучать такую функцию, необходимо выбрать функцию ошибки. В перцептроне Розенблатта используется функция ошибки, называемая *критерий перцептрона*:

$$E_p(\boldsymbol{w}) = -\sum_{\boldsymbol{x} \in M} y(\boldsymbol{x})(\boldsymbol{w}^T \boldsymbol{x}),$$

где M обозначает множество тех примеров, которые перцептрон с весами классифицирует неверно.

Иначе говоря, мы минимизируем суммарное отклонение наших ответов от правильных, но только в неправильную сторону: верный ответ ничего не вносит в функцию ошибки. Умножение на $y(\boldsymbol{x})$ здесь нужно для того, чтобы знак -1, значит, перцептрон выдал положительное число (иначе бы ответ был верным), и наоборот.

Теперь мы можем оптимизировать её градиентным спуском. На очередном шаге получаем:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{\tau} - \eta \nabla_{\boldsymbol{w}} E_p(\boldsymbol{w}) = \boldsymbol{w}^{\tau} + \eta t_n \boldsymbol{x}_n.$$

Это правило обновления весов так и называется – *правило обучения перцептрона*, и это было основной математической идеей Розенблатта.

Но и это ещё не всё. Чтобы двигаться дальше, нам нужно добавить в перцептрон ещё один компонент – *функцию активации*. Дело в том, что в реальности перцептроны, не могут быть линейными, как мы их сейчас определили: если они останутся линейными, то из них невозможно будет составить содержательную сеть. На выходе перцептрона обязательно присутствует нелинейная *функция активации*, которая принимает на вход всё ту же линейную комбинацию. Самая классическая и наиболее популярная функция активации – *логистический сигмоид*:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

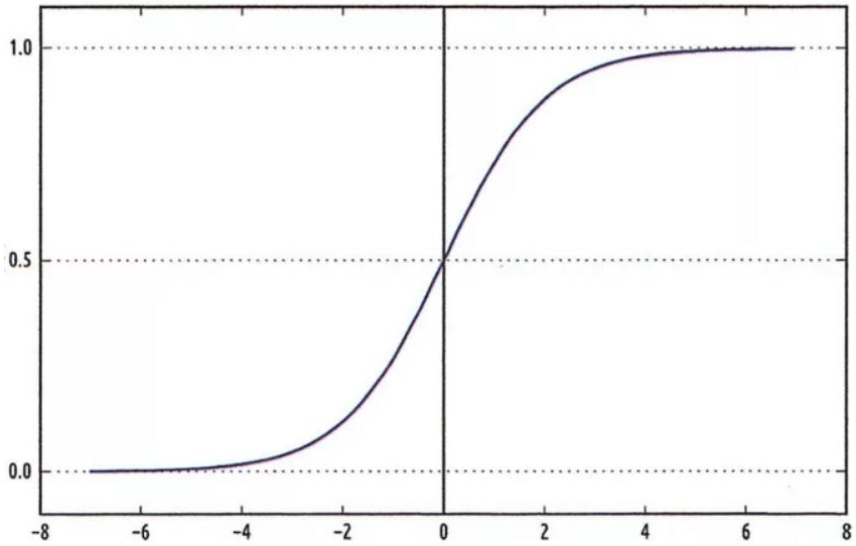


Рис. 1. Логистический сигмоид.

График его показан на рис. 1. Неформально говоря, если на вход подают большое отрицательное число, то нейрон совсем не активируется, а если большое положительное, то активируется почти наверняка. Эта функция называется сигмоидом потому, что е форма похожа на букву S.

Обучать такой перцептрон не сложно: можно использовать всё тот же градиентный спуск. Разница только в том, что теперь мы рассматриваем задачу бинарной классификации, и данные $y(\boldsymbol{x})$ представляют собой метки 0 и 1, а функция ошибки выглядит как перекрёстная энтропия:

$$E(\boldsymbol{w}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i log_\sigma(\boldsymbol{w}^T\boldsymbol{x}) + (1-y_i)\log(1-\sigma(\boldsymbol{w}^T\boldsymbol{x}))).$$

В итоге перцептрон с логистическим сигмоидом в качестве функции активации реализует логистическую регрессию (которую вы изучили в пятой лабораторной работе) и строит при этом линейные разделяющие поверхности.

Ещё стоит отметить, что один перцептрон – это просто разновидность линейной регрессии, которая была подробно рассмотрена в четвёртой лабораторной работе.

1. Многослойный перцептрон

Теперь, когда мы разобрались с конструкцией одного перцептрона, мы можем собрать из них целую сеть. Она будет представлять собой граф вычислений. Один перцептрон (рис.2.) может служить одним узлом в этом графе, выступая в роли элементарной функции.

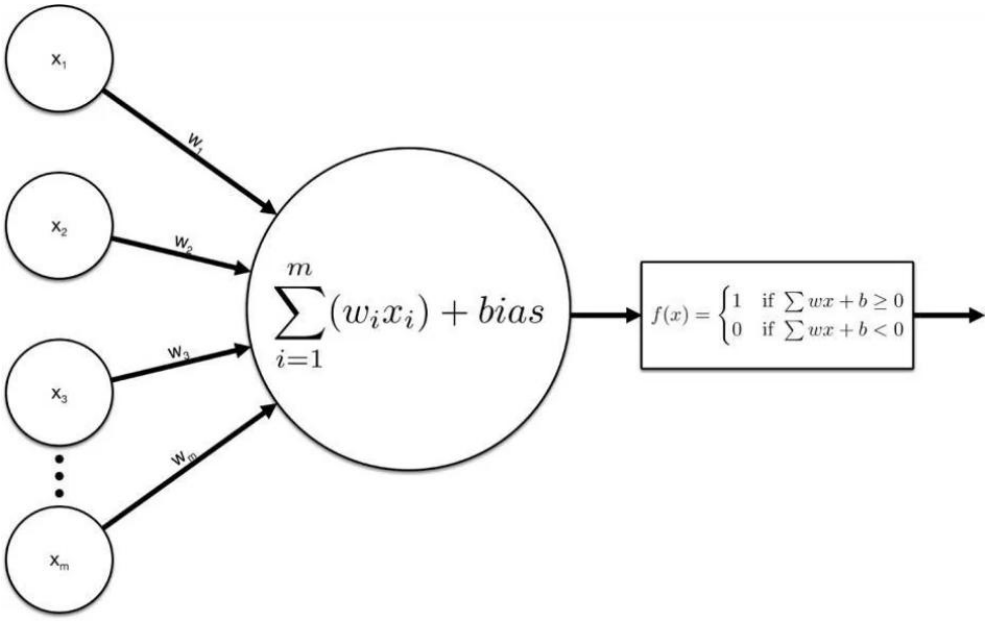


Рис.2. Граф вычислений для перцептрона.

Основная идея архитектуры нейронных сетей: рис. 2 показана структура одного перцептрона, на рис. 3 – полносвязная нейронная сеть с одним скрытым уровнем и одним выходным уровнем. Каждый вход этой сети подаётся на вход каждому перцептрону «первого уровня». Затем выходы перцептронов «первого уровня» подаются на вход перцептронам «второго уровня», а их выходы уже считаются выходами всей сети целиком.

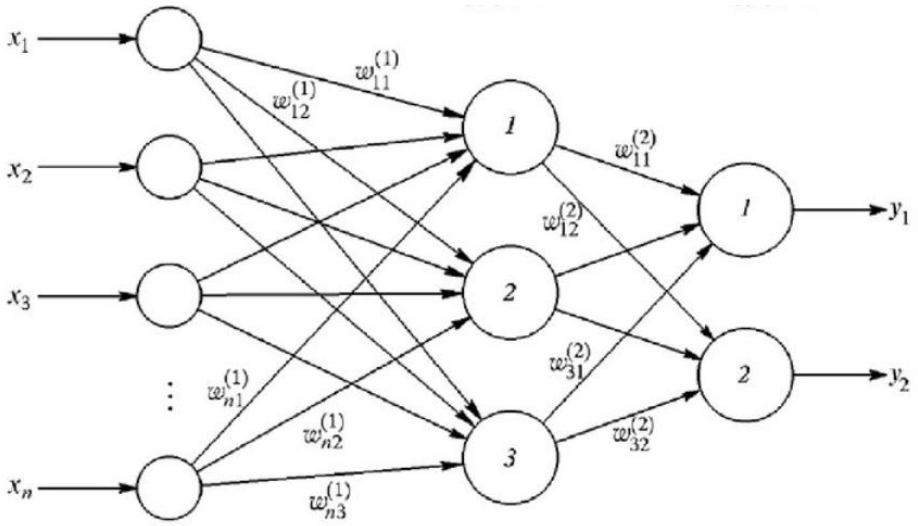


Рис. 3. Полносвязная нейронная сеть с одним скрытым уровнем
и одним выходным уровнем.

2. Как объединять нейроны в сеть

Обычно в любой сети отдельные нейроны объединены в *слои*. Вектор входов подаётся сразу в несколько параллельных нейронов, у каждого из которых свои

собственные веса, а затем выходы этих параллельных нейронов опять рассматриваются как единое целое, новый вектор выходов.

Казалось бы, для графа вычислений теоретически нет разницы, какую именно архитектуру выбрать, всё равно это всего лишь набор независимых нейронов. Однако концепция слоя очень важна с вычислительной точки зрения. Дело в том, что вычисления в целом слое нейронов можно *векторизовать*, то есть представить в виде умножения матрицы на вектор и применении вектор-функций активации. Если в слое k нейронов, и веса у них:

$$\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_k, \qquad \boldsymbol{w}_i = (w_{i1}, w_{i2}, \dots, w_{in})^T,$$

а на вход подаётся вектор

$$\boldsymbol{x} = (x_1, x_2, \dots, x_n)^T,$$

то в результате мы получим у нейрона с весами $\boldsymbol{w}_i$ выход

$$y_i = f(\boldsymbol{w}_i{}^T \boldsymbol{x}),$$

где f – его функция активации. Тогда вычисление, которое делают все нейроны сразу, можно представить в векторной форме так:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix} = \boldsymbol{y} = f(W, \boldsymbol{x}) = \begin{pmatrix} f(\boldsymbol{w}_1{}^T \boldsymbol{x}) \\ \vdots \\ f(\boldsymbol{w}_k{}^T \boldsymbol{x}) \end{pmatrix},$$

где

$$W = \begin{pmatrix} w_1 \\ \vdots \\ w_k \end{pmatrix} = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & & \vdots \\ w_{k1} & \cdots & w_{kn} \end{pmatrix},$$

и вычисление всего слоя сведётся к умножению матрицы весов на вектор входов, а затем покомпонентному применению одной и той же функции активации. При этом оказывается, что матричные вычисления можно реализовать гораздо эффективнее, в частности на графических процессорах (видеокартах), чем те же вычисления, но представленные в виде обычных циклов. Поэтому такое векторизованное представление – это из главных инструментов для того, чтобы перенести обучение и применение нейронных сетей на видеокарты, а это ускоряет все процессы буквально в десятки раз.

3. Современные перцептроны: функции активации

Все современные нейронные сети состоят из комбинации перцептронов. Разница есть только в том, какова собственно, конструкция нелинейности. И вот на этом вопросе уже стоит остановиться подробнее, он представляется чрезвычайно интересным.

Мы уже рассматривали выше одну из самых популярных функций активации – логистический сигмоид (рис.1): $\sigma(x) = \frac{1}{1+e^{-x}}$. Эта функция обладает всеми свойствами, необходимыми для нелинейности в нейронной сети: она ограничена, везде дифференцируема, и её производную легко подсчитать как $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Но конечно же она не одна такая. Есть много разных функций активации, познакомимся с ними поближе.

Гиперболический тангенс:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

очень похож по свойствам на логистический сигмоид: он тоже непрерывен, тоже ограничен, и производную тоже легко подсчитать через него самого:

$$\tanh'(x) = 1 - \tanh^2(x).$$

По сравнению с логистическим сигмоидом гиперболический тангенс (рис.4.) значительно «круче» растёт и убывает, быстрее приближается к своим пределам.

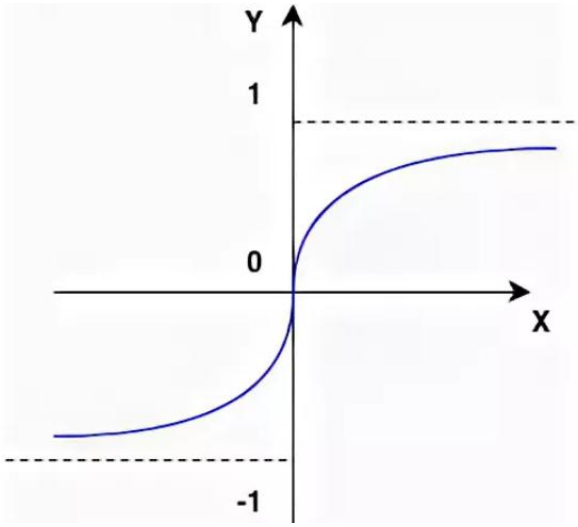


Рис.4. Гиперболический тангенс.

Однако есть и более важная разница: для функции σ ноль является точкой насыщения, то есть если пытаться обучить значение этой функции в ноль, вход будет стремиться к минус бесконечности, а производная к нулю. А для $tanh$ ноль – это как раз самая нестабильная промежуточная точка, от нуля легко оттолкнуться и начать менять аргумент в любую сторону.

В качестве функции активации можно рассмотреть и обычную ступенчатую функцию, она же *функция Хевисайда*:

$$step(x) = \begin{cases} 1, \text{если } x < 0, \\ 0, \text{если } x > 0. \end{cases}$$

Эта функция использовалась в ранних конструкциях перцептронов. То, что она не определена в нуле, не очень мешает вести обучение. Один перцептрон со ступенчатой функцией активации обучить вполне возможно. Для этого достаточно просто точно так же подсчитывать «мягкий» результат в виде комбинации входов и весов.

Но сеть с несколькими уровнями на ступенчатых функциях активации, к сожалению, не построишь, ведь производная от ступеньки просто всегда равна нулю.

Разобравшись с классическими функциями активации, давайте двигаться к современности. Главная идея, во многом изменившая архитектурные основы современных нейронных сетей, – это так называемые *rectified linear units (ReLU)*. Функция активации у них кусочно-линейная:

$$ReLU(x) = \begin{cases} 0, \text{если } x < 0, \\ x, \text{если } x \geq 0. \end{cases}$$

То же самое можно записать более кратко $ReLU(x) = \max(0, x)$.

Прежде всего, отметим, что ReLU-нейроны эффективнее основанных на логистическом сигмоиде и гиперболическом тангенсе. Например, чтобы подсчитать производную $\sigma'(x)$, нужно вычислить непростую функцию σ, а затем умножить $\sigma(x)$ на $(1 - \sigma(x))$; с тангенсом примерно та же история, только нужно возводить в квадрат. А чтобы вычислить производную $ReLU'(x)$, нужно ровно одно сравнение: если x меньше нуля, выдаём ноль, если больше нуля – единицу. На практике это означает, что основанные на ReLU-нейронах сети при одном и том же «вычислительном бюджете» на обучение, на одном и том же «железе» могут быть значительно больше (по размеру, то есть по числу нейронов), чем сети с более сложными функциями активации.

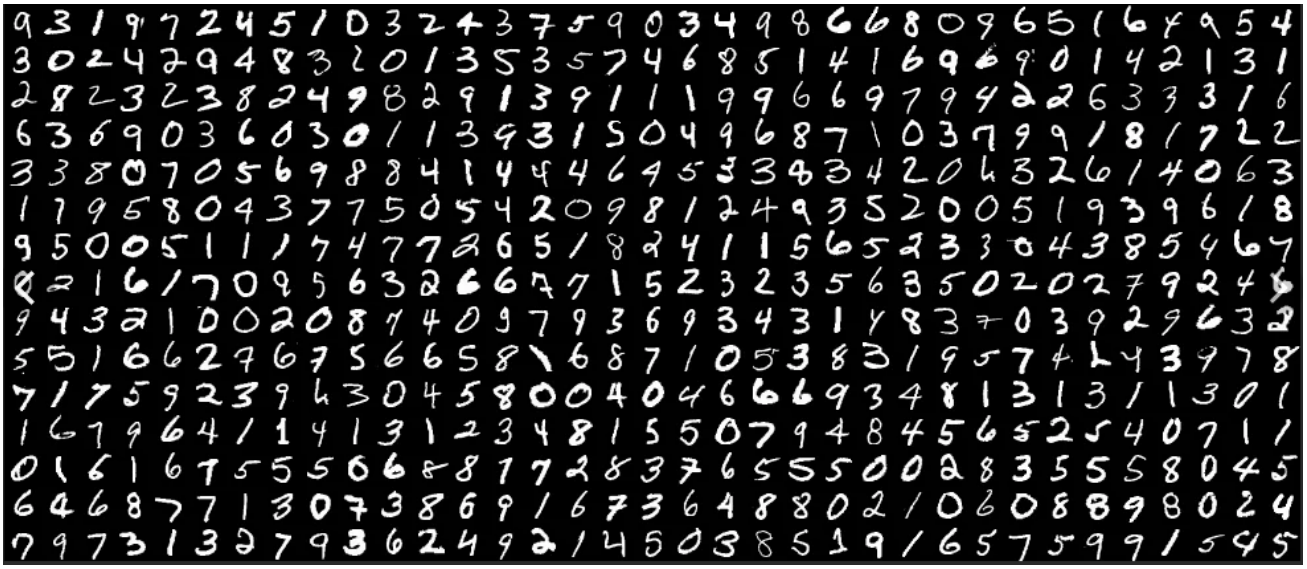
Прямые непосредственные исследования функции активации в реальных нейронах человеческого мозга и приближённые к биологической модели дают функцию, гораздо более похожую на ReLU, чем на сигмоид.

Существуют очень популярные на сегодняшний день различные модификации и обобщения ReLU, которые пытаются сохранить вычислительную эффективность и при этом добавить немного гибкости в базовую конструкцию. Например, «протекающий ReLU» (Leaky ReLU, LReLU), «параметризованный ReLU» (Parametric ReLU, PReLU) или «экспоненциальный линейный нейрон» (Exponential Linear Unit, ELU), но их рассмотрение выходит за рамки данной лабораторной работы.

4. Набор данных MNIST

До начала выполнения любого задания было бы полезным изучить исходные. Здесь и в последующих лабораторных работах мы будем использовать ***набор данных MNIST***.

Итак, набор данных MNIST состоит из изображений размером 28x28, каждый пиксель которого представляет собой оттенок серого.



Набор данных содержит изображения десяти цифр от 0 до 9. Каждое изображение имеет свою метку класса:

МЕТКА	*КЛАСС*
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Каждому входному изображению соответствует одна из перечисленных выше меток. Набор данных MNIST содержит 70 000 изображений. Из этих 70 000 мы воспользуемся 60 000 для обучения логистической регрессии.

А оставшимися 10 000 элементами мы воспользуемся для того, чтобы проверить насколько хорошо наша модель обучилась классифицировать цифры.

5. Распознавание рукописных цифр

Поскольку MNIST – это своего рода Hello World для современной обработки изображений, в TensorFlow этот набор данных поддерживается «из коробки», и для импорта MNIST достаточно написать буквально 1 строчку кода:

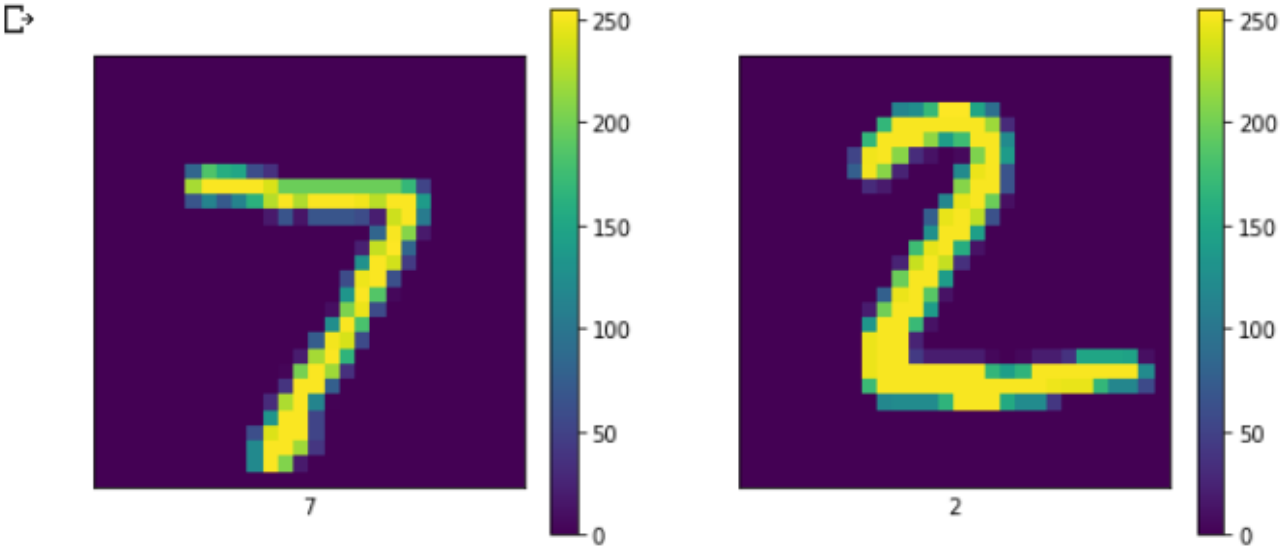
```
[ ]  # Import MINST data
     (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Полученные данные уже разбиты на тренировочную (x_train, y_train), тестовую (x_test, y_test) выборки, содержащие 60000 и 10000.

Упражнение

Запустите Jupyter Notebook и откройте блокнот *multilayer_perceptron.ipynb*. В первой ячейке осуществляется загрузка библиотек, затем происходит инициализация данных для обучения и тестирования модели: загрузка набора данных MNIST. В следующей ячейке вам необходимо дописать следующий код, для визуализации изображений из загруженного набора данных:

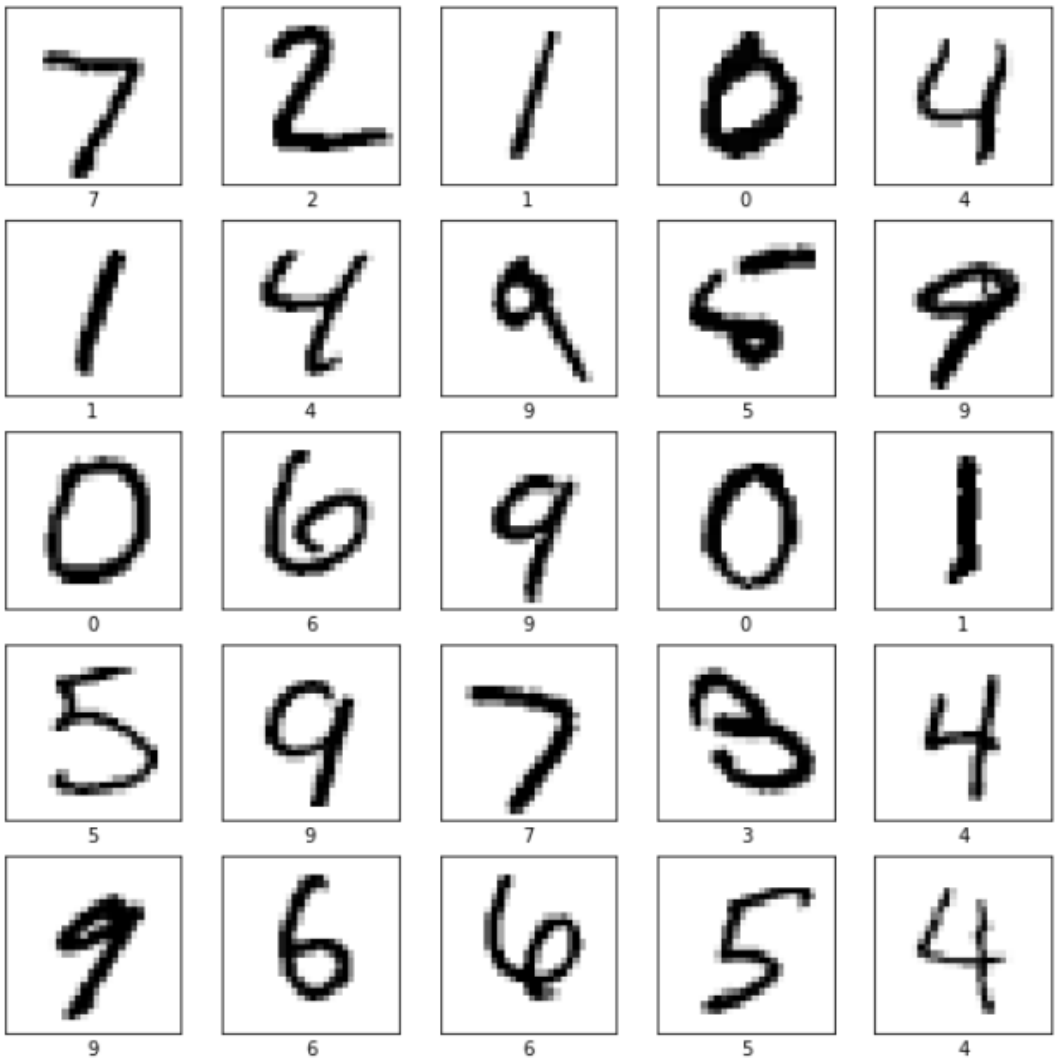
```
[3]  plt.figure(figsize=(10,10))
     for i in range(4):
         plt.subplot(2,2,i+1)
         plt.xticks([])
         plt.yticks([])
         plt.grid(False)
         plt.imshow(x_test[i])
         plt.xlabel(y_test[i])
         plt.colorbar()
```



Упражнение

Перед подготовкой модели данные должны быть предварительно обработаны. На предыдущем шаге мы отобразили изображения из тренировочного набора, заметим, что значения пикселей находятся в диапазоне от 0 до 255. Поэтому следующим шагом выполняется нормализация изображений, т.е. мы масштабируем эти значения до диапазона от 0 до 1. В ячейке после нормализации вам необходимо дописать следующий код, для визуализации первых 25 бинаризованных изображений из тестового набора:

```
[ ]  plt.figure(figsize=(10,10))
     for i in range(25):
         plt.subplot(5,5,i+1)
         plt.xticks([])
         plt.yticks([])
         plt.grid(False)
         plt.imshow(x_test[i], cmap=plt.cm.binary)
         plt.xlabel(y_test[i])
```



В этом примере мы не будем использовать двумерную структуру изображений, а представим их в виде обычных векторов размерности 784 (28*28).

6. Объявление необходимых для обучения переменных

Для начала установим начальные параметры: скорость обучения, количество эпох обучения и размер батча. По умолчанию они заданы 0.001, 15 и 100 соответственно.

```
[ ]  # Parameters
     learning_rate = 0.001
     training_epochs = 15
     batch_size = 100
     display_step = 1
```

В ходе выполнения лабораторного задания вам необходимо будет подобрать эти параметры таким образом, чтобы получить наиболее точный результат классификации.

В четвёртой ячейке установлены параметры нейронной сети:

```
[ ]  # Network Parameters
     n_hidden_1 = 256 # 1st layer number of features
     n_hidden_2 = 256 # 2nd layer number of features
     n_input = 784 # MNIST data input (img shape: 28*28)
     n_classes = 10 # MNIST total classes (0-9 digits)
```

В TensorFlow требуемые операции выражаются с помощью символьных переменных, поэтому в пятой ячейке мы создаём переменные для тренировочных данных:

```
[ ]  # tf Graph input
     x = tf.placeholder("float", [None, n_input])
     y = tf.placeholder("float", [None, n_classes])
```

В данном случае x и y – это не какие-то заранее заданные тензоры, а так называемые ***заглушки (placeholder)***, которые будут заполнены, когда мы попросим TensorFlow произвести вычисления. Мы хотим иметь возможность использовать произвольное число 784-мерных векторов для обучения, поэтому в качестве одной из размерностей указываем None. Для TensorFlow это значит, что данная размерность может иметь произвольную длину.

Кроме заглушки для тренировочных данных, нам потребуются переменные, которые мы собственно и будем изменять при обучении нашей модели: weights и biases, они объявляются в шестой ячейке блокнота *multilayer_perceptron.ipynb*.

Здесь weights имеет размерность 784 * 10, так как мы собираемся умножать вектор размерности 784 на weights и получить предсказание для 10 возможных меток, а вектор biases размерности 10 – это свободный член, который мы добавляем к выходу.

7. Обучение

В качестве модели для обучения мы рассмотрим *softmax-регрессию*. Это обобщение логистической регрессии на случай нескольких классов: чтобы получить «вероятности» классов, которые нам хочется оценить, мы применяем так называемую softmax-функцию к вектору получившихся ненормализованных оценок:

$$softmax(x)_j = \frac{\exp(x_j)}{\sum_i \exp(x_j)}.$$

Идея softmax-функции состоит в том, чтобы несколько заострить, преувеличить разницу между полученными значениями: softmax будет выдавать значения очень близкие к нулю, для всех x_j, существенно меньших максимального.

В качестве функции потерь мы будем использовать стандартную для логистической регрессии перекрёстную энтропию (кросс-энтропию, cross-entropy):

$$H_t(y) = -\sum_i t_i \log(y_i),$$

где y – предсказанное нами значение, а t – исходная разметка (правильный ответ).

В седьмой ячейке объявляется функция *multilayer_perceptron*, которая задаёт простую нейронную сеть, состоящую из двух скрытых слоёв с ReLU-активациями и выходного слоя с линейной активацией.

После того как мы импортировали нужные модули и объявили все переменные, нашу модель на TensorFlow можно записать как представлено в ячейке 8.

В девятой ячейке описан непосредственно сам цикл обучения и теста.

Упражнение

В блокноте multilayer_perceptron.ipynb представлен уже реализованный пример распознавания рукописных цифр. Все используемые при его реализации операции уже известны вам из данного и предыдущих компьютерных практикумов. Ваша задача заключается в том, чтобы подробно разобраться в реализации метода и процессе его и настроить параметры обучения наиболее оптимальным образом (то есть так, чтобы функция стоимости cost оказалась как можно меньше, а точность классификации тестовой выборки Accuracy как можно выше). В качестве эксперимента, вы можете

изменять архитектуру сети, задаваемой в функции multilayer_perceptron, и оценить как это влияет на результаты обучения.

Задание к лабораторной работе

1) Выполните все упражнения из данного документа.

2) Представьте отчёт в виде отредактированного файла *lab4 multilayer_perceptron.ipynb* с результатами вашей работы.

3) Ответьте на контрольные вопросы:

1. Что такое линейный перцептрон и почему из них невозможно составить «хорошую» сеть?

2. Что представляет собой перцептрон Розенблатта, какая в нём используется функция ошибки и правило обучения перцептрона?

3. Что такое многослойный перцептрон?

4. Как правильно объединять нейроны в сеть?

5. Перечислите рассмотренные в данной работе функции активации, в чём их основное отличие, какая является наиболее эффективной и почему?