Лабораторная работа 7

Введение в фреймворки и настройки сети

Очевидно было бы логичным исследовать, приведёт ли добавление дополнительных слоев в наших спроектированных ранее нейронных сетях к увеличению их точности. Оказывается, что получение более глубоких сетей для хорошего обучения является серьёзным препятствием. Чтобы преодолеть эти препятствия и обеспечить глубокое обучение (ГО), потребовался ряд инноваций в теории и методах машинного обучения. Преимущество использования платформы и фреймворков ГО, с которыми мы будем знакомиться далее, заключается в том, что нам не нужно реализовывать все эти новые методы с нуля в нашей нейронной сети. Недостатком же является то, что вы не будете разбираться в деталях так глубоко, как в предыдущих примерах.

Пример программы: переход к фреймворку ГО

В этом примере мы показываем, как реализовать классификацию рукописных цифр используя фреймворк ГО на примере TensorFlow и (в некоторых случаях) PyTorch. Оба этих фреймворка популярны и гибки.

TensorFlow предоставляет ряд различных конструкций и позволяет работать на разных уровнях абстракции, используя разные интерфейсы прикладного программирования (API). В общем, для простоты эффективнее выполнять свою работу на максимально возможном уровне абстракции, потому что это означает, что вам не нужно реализовывать низкоуровневые детали модели ГО. Для примеров, которые мы будем изучать, подходящим уровнем абстракции является Keras API. Keras начинался как отдельная библиотека. Он не был привязан к TensorFlow и мог использоваться с несколькими фреймворками ГО. Однако на данный момент Keras полностью поддерживается внутри самого TensorFlow.

Фреймворки реализованы в виде библиотек Python. То есть мы по-прежнему пишем нашу программу как программу на Python и просто импортируем выбранный фреймворк как библиотеку. Затем мы можем использовать функции ГО из работы в нашей программе. Код инициализации для нашего примера TensorFlow показан во фрагменте кода 7.1.

Фрагмент кода 7.1 — Импортирование для нашего примера TensorFlow/Keras

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
import numpy as np
import logging

tf.get_logger().setLevel(logging.ERROR)

tf.random.set_seed(7)
```

```
EPOCHS = 20
BATCH_SIZE = 1
```

Как вы можете видеть в коде, у TensorFlow есть собственная функция для генерации случайного начального числа, которое необходимо установить, если мы хотим, как и ранее, воспроизводимых результатов. Однако это по-прежнему не гарантирует, что повторные прогоны модели дадут идентичные результаты для всех типов сетей, поэтому далее мы не будем беспокоиться о задании случайных начальных значений. Предыдущий фрагмент кода также задает уровень ведения журнала — будут выводиться только ошибки, без вывода предупреждений.

Затем мы загружаем и подготавливаем наш набор данных MNIST. Поскольку MNIST — это общий набор данных, он включен в Keras. Мы можем получить к нему доступ, вызвав keras.datasets.mnist и load_data. Переменные train_images и test_images будут содержать входные значения, а переменные train_labels и test_labels будет содержать истинные значения (фрагмент кода 7.2).

Фрагмент кода 7.2 — Загрузка и подготовка обучающих и тестовых наборов данных

```
#     Загрузить обучающие и тестовые наборы данных.
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images,
test_labels) = mnist.load_data()

#     Стандартизируем данные
mean = np.mean(train_images)
stddev = np.std(train_images)
train_images = (train_images - mean) / stddev
test_images = (test_images - mean) / stddev

#     Метки быстрого кодирования
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)
```

Как и раньше, нам нужно стандартизировать входные данные и сразу закодировать метки. Мы используем функцию to_categorical для быстрого кодирования наших меток вместо того, чтобы делать это вручную, как мы делали в нашем предыдущем примере. Это служит примером того, как фреймворк предоставляет функциональные возможности для упрощения нашей реализации общих задач.

> *Если вы не так хорошо знакомы с Python, стоит отметить, что функции могут быть определены с необязательными аргументами, и чтобы избежать необходимости передавать аргументы в определенном порядке, необязательные аргументы можно передавать, сначала указав, какой аргумент мы пытаемся передать. Примером может служить аргумент* ***num_classes*** *в функции* ***to_categorical****.*

Теперь мы готовы создать нашу сеть. Нет необходимости определять переменные для отдельных нейронов, потому что фреймворк предоставляет функциональные возможности для одновременного создания экземпляров целых слоев нейронов. Нам нужно решить, как инициализировать веса, что мы и делаем, создавая объект инициализатора, как показано во фрагменте кода 7.3. Это может показаться несколько запутанным, но пригодится, когда мы захотим поэкспериментировать с различными значениями инициализации.

Фрагмент кода 7.3 — Создание сети

```
#     Объект, используемый для инициализации весов
initializer = keras.initializers.RandomUniform(
      minval=-0.1, maxval=0.1)

#     Создайте последовательную модель
#     784 входов
#     Два плотных (полностью связанных) слоя с 25 и 10
#     нейронами.
#     tanh функция активации для скрытого слоя.
#     Лог. (сигмоидная) функция активации выходного слоя.
model = keras.Sequential([
      keras.layers.Flatten(input_shape=(28, 28)),
      keras.layers.Dense(25, activation='tanh',
            kernel_initializer=initializer,
            bias_initializer='zeros'),
      keras.layers.Dense(10, activation='sigmoid',
            kernel_initializer=initializer,
            bias_initializer='zeros')])
```

Сеть создается путем создания экземпляра объекта `keras.Sequential`, что означает, что мы используем Keras Sequential API. (Это самый простой API, и мы будем использовать его в следующих нескольких примерах, пока не начнем создавать сети, требующие более сложного API.) Мы передаем список слоев в качестве аргумента классу Sequential. Первый слой — это слой Flatten, который не выполняет вычислений, а только изменяет организацию ввода. В нашем случае входные данные меняются с массива 28×28 на массив из 784 элементов. Если бы данные уже были организованы в одномерный массив, мы могли бы пропустить слой Flatten и просто объявить два слоя Dense. Если бы мы сделали это таким образом, нам нужно было бы передать параметр input_shape первому слою Dense, потому что мы всегда должны объявлять размер входных данных для первого слоя в сети.

Второй и третий слои являются плотными слоями, что означает, что они полностью связаны. Первый аргумент сообщает, сколько нейронов должен иметь каждый слой, а аргумент активации сообщает тип функции активации; мы выбираем tanh и сигмоид, где сигмоид означает логистическую сигмовидную функцию. Мы передаем наш объект инициализатора для инициализации обычных весов с помощью аргумента kernel_initializer. Веса смещения инициализируются равными 0 с помощью аргумента bias_initializer.

Одна вещь, которая может показаться странной, заключается в том, что мы ничего не говорим о количестве входов и выходов для второго и третьего слоев. Если подумать, количество входов полностью определяется тем, что оба слоя полностью связаны, и тем фактом, что мы указали количество нейронов в каждом слое вместе с количеством входов для первого слоя сети. Это подчеркивает, что использование фреймворка ГО позволяет нам работать на более высоком уровне абстракции. В частности, мы используем слои вместо отдельных нейронов в качестве строительных блоков, и нам не нужно беспокоиться о деталях того, как отдельные нейроны связаны друг с другом. Это часто отражается и в том, как мы изображаем модели на рисунках, где мы работаем с отдельными нейронами только тогда, когда нам нужно объяснить альтернативные топологии сети. В связи с этим на рис. 7.1 показана наша сеть распознавания цифр на этом более высоком уровне абстракции. Мы используем прямоугольные блоки с закругленными углами, чтобы изобразить слой нейронов, в отличие от кругов, которые представляют отдельные нейроны.

Теперь мы готовы обучить сеть, что делается с помощью фрагмента кода 7.4. Сначала мы создаем объект keras.optimizer.SGD. Это означает, что мы хотим использовать стохастический градиентный спуск (SGD) при обучении сети. Как и в случае с инициализатором, это может показаться несколько запутанным, но обеспечивает гибкость настройки параметров для процесса обучения, который мы вскоре рассмотрим. На данный момент мы просто установили скорость обучения на 0,01, чтобы соответствовать тому, что мы сделали в нашем простом примере Python. Затем мы подготавливаем модель к обучению, вызывая функцию компиляции.

Рисунок 7.1 — Сеть классификации цифр, использующая слои в качестве строительных блоков

Мы предоставляем параметры, чтобы указать, какую функцию потерь использовать (где мы используем mean_squared_error, как и раньше), оптимизатор, который мы только что создали и который нам интересно посмотреть на показатель точности во время обучения.

Фрагмент кода 7.4 — Обучение сети

```
#     Используйте стохастический градиентный спуск (SGD)
#     со скоростью обучения 0,01 без других настроек.
#     MSE как функция потерь и показатель точности
#     во время обучения.
opt = keras.optimizers.SGD(learning_rate=0.01)
model.compile(loss='MSE', optimizer = opt,
      metrics =['accuracy'])

#     Обучить модель на 20 эпохах.
#     Перемешать (сделать случайным) порядок.
#     Обновлять веса после каждого примера (batch_size=1).
history = model.fit(train_images, train_labels,
                    validation_data=(test_images,
                    test_labels), epochs=EPOCHS,
                    batch_size=BATCH_SIZE, verbose=2,
                    shuffle=True)
```

Наконец, мы вызываем функцию подгонки `fit` для модели, которая запускает процесс обучения. Как видно из названия функции, она подгоняет модель к данным. Первые два аргумента определяют обучающий набор данных. Параметр validation_data это тестовый набор данных. Наши переменные EPOCHS и BATCH_SIZE из кода инициализации определяют, сколько эпох нужно обучать и какой размер пакета мы используем. Мы установили для BATCH_SIZE значение 1, что означает, что мы обновляем вес после одного обучающего примера, как мы сделали в нашем простом примере на Python. Мы устанавливаем verbose=2, чтобы получить разумный объем информации, распечатываемой в процессе обучения, и устанавливаем в случайном порядке значение True, чтобы указать, что мы хотим, чтобы порядок данных обучения был рандомизированным в процессе обучения. В целом, эти параметры соответствуют тому, что мы сделали в нашем простом примере на Python.

В зависимости от того, какую версию TensorFlow вы используете, вы можете получить достаточное количество выводов об открытии библиотек, обнаружении графического процессора (GPU) и других проблемах при запуске программы. Если вы хотите, чтобы они были менее подробным, вы можете установить для переменной среды TF_CPP_MIN_LOG_LEVEL значение 2. Если вы используете bash, вы можете сделать это с помощью следующей командной строки:

```
export TF_CPP_MIN_LOG_LEVEL=2
```

Другой вариант — добавить следующий фрагмент кода вверху вашей программы.

```
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

Здесь показаны выводы для первых нескольких периодов обучения. Некоторые временные метки убраны, чтобы сделать вывод более читаемым.

```
Эпоха 1/20
loss: 0.0535 - acc: 0.6624 - val_loss: 0.0276 - val_acc: 0.8893

Эпоха 2/20
loss: 0.0216 - acc: 0.8997 - val_loss: 0.0172 - val_acc: 0.9132

Эпоха 3/20
loss: 0.0162 - acc: 0.9155 - val_loss: 0.0145 - val_acc: 0.9249

Эпоха 4/20
loss: 0.0142 - acc: 0.9227 - val_loss: 0.0131 - val_acc: 0.9307

Эпоха 5/20
loss: 0.0131 - acc: 0.9274 - val_loss: 0.0125 - val_acc: 0.9309

Эпоха 6/20
loss: 0.0123 - acc: 0.9313 - val_loss: 0.0121 - val_acc: 0.9329
```

В выводе loss представляет собой среднеквадратичную ошибку (СКО) обучающих данных, acc представляет точность прогноза обучающих данных, val_loss представляет СКО тестовых данных, а val_acc представляет точность предсказания тестовых данных.

Стоит отметить, что мы не получаем точно такое же поведение при обучении, которое наблюдалось в нашей простой модели на Python. Трудно понять почему, не углубляясь в детали реализации TensorFlow. Скорее всего, это могут быть тонкие проблемы, связанные с рандомизацией исходных параметров и случайным порядком выбора обучающих примеров.

Ещё один факт, на который стоит обратить внимание, это то, насколько просто было реализовать наше приложение для классификации цифр с помощью TensorFlow. Использование фреймворка TensorFlow позволяет нам изучать более продвинутые методы, сохраняя при этом код простым и читаемым.

Теперь мы переходим к описанию некоторых методов, необходимых для обучения в более глубоких сетях. После этого мы, наконец, можем провести наш первый эксперимент по глубокому обучению.

Проблема насыщенных нейронов и Исчезающие градиенты

В наших экспериментах мы внесли некоторые, казалось бы, произвольные изменения в параметр скорости обучения, а также в диапазон, с которым мы инициализировали веса. Для нашего примера обучения персептрона и сети XOR мы использовали скорость обучения 0,1, а для классификации цифр мы использовали 0,01. Точно так же для весов мы использовали диапазон от -1,0 до +1,0 для примера XOR, тогда как мы использовали от -0,1 до +0,1 для примера с цифрами. Резонный вопрос заключается в том, есть ли здесь какая-то система. Секрет заключается в том, что мы изменили значения просто потому, что без этих изменений наши сети плохо обучались. Далее мы обсудим причины этого и рассмотрим некоторые рекомендации, которые можно использовать при выборе этих, казалось бы, случайных параметров.

Чтобы понять, почему иногда сложно заставить сети обучаться, нам нужно

более подробно рассмотреть нашу функцию активации. На рис. 7.2 показаны две наши S- образные функции.

Следует отметить, что обе функции не представляют интереса за пределами показанного z-интервала (именно поэтому мы в первую очередь показали только этот z-интервал). Обе функции представляют собой более или менее прямые горизонтальные линии за пределами этого диапазона.

Теперь рассмотрим, как работает наш процесс обучения. Мы вычисляем производную функции ошибки и используем ее, чтобы определить, какие веса корректировать и в каком направлении. Интуитивно, что мы делаем, так это слегка подправляем входные данные для функции активации (z на графике на рис. 7.2) и смотрим, повлияет ли это на выходные данные.

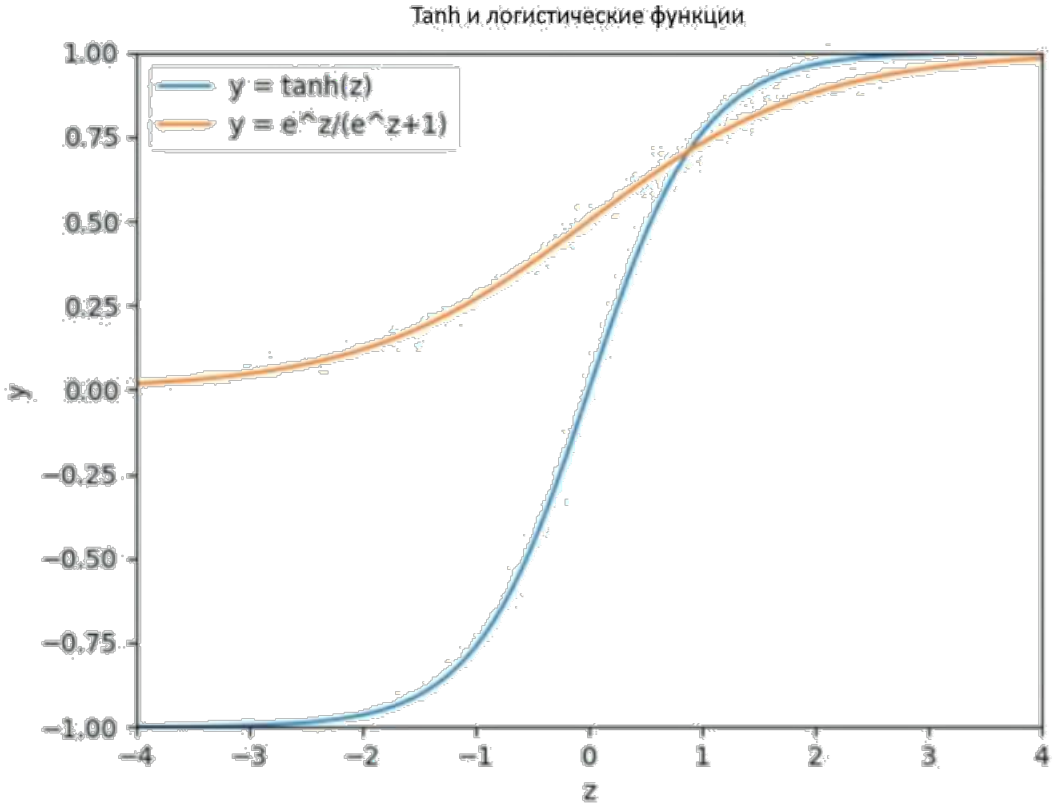

Рисунок 7.2 — Две S-образные функции tanh и логистическая сигмоида

Если значение z находится в пределах небольшого диапазона, показанного на диаграмме, это изменит вывод (значение y на диаграмме). Теперь рассмотрим случай, когда значение z является большим положительным или отрицательным числом. Изменение ввода на небольшую величину (или даже на большую величину) не повлияет на вывод, поскольку вывод представляет собой горизонтальную линию в этих областях. Мы говорим, что нейрон насыщен.

Насыщенные нейроны могут привести к полной остановке обучения. Как вы помните, когда мы вычисляем градиент с помощью алгоритма обратного распространения, мы распространяем ошибку обратно по сети, и частью этого

процесса является умножение производной функции потерь на производную функции активации.

Рассмотрим, каковы производные двух приведенных выше функций активации для значений z значительной величины (положительных или отрицательных). Производная равна 0! Другими словами, никакая ошибка не будет распространяться назад, и веса не будут корректироваться. Точно так же, даже если нейрон не полностью насыщен, производная меньше 0. Выполнение серии умножений (по одному на слой), где каждое число меньше 0, приводит к приближению градиента к 0. Эта проблема известна как проблема исчезающего градиента. Насыщенные нейроны — не единственная причина исчезновения градиентов, как мы увидим позже.

> ***Насыщенные*** *нейроны нечувствительны к входным изменениям, потому что их производная равна 0 в насыщенной области. Это одна из причин проблемы* ***исчезающего градиента****, когда ошибка обратного распространения равна 0, а веса не корректируются.*

Инициализация и нормализация Методы предотвращения насыщения нейронов

Теперь мы исследуем, как мы можем предотвратить или решить проблему насыщения нейронов. Обычно используются и часто комбинируются три метода: инициализация веса, стандартизация ввода и нормализация пакета.

Инициализация веса

Первый шаг в предотвращении насыщения нейронов — убедиться, что наши нейроны не насыщены с самого начала, и именно здесь важна инициализация веса. Стоит отметить, что, хотя мы используем один и тот же тип нейронов в наших разных примерах, фактические параметры нейронов, которые мы показали, сильно различаются. В примере XOR нейроны в скрытом слое имели три входа, включая смещение, тогда как в примере с классификацией цифр нейроны в скрытом слое имели 785 входов. С таким количеством входных данных нетрудно представить, что взвешенная сумма может сильно колебаться как в отрицательном, так и в положительном направлении, если есть лишь небольшой дисбаланс в количестве отрицательных и положительных входных данных, если веса велики. С этой точки зрения имеет смысл, что если нейрон имеет большое количество входных данных, то мы хотим инициализировать веса меньшим значением, чтобы иметь разумную вероятность сохранения входных данных для функции активации, близких к 0, чтобы избежать насыщенность. Двумя популярными стратегиями инициализации веса являются инициализация Глорота и инициализация Хе. Инициализация Glorot рекомендуется для нейронов на основе tanh и сигмоидных нейронов, а инициализация Хе рекомендуется для нейронов на основе ReLU (описанных ниже). Оба они учитывают количество входов, а инициализация Глорота также учитывает количество выходов. Инициализация Глорота и Хе существует в двух вариантах:

один основан на равномерном случайном распределении, а другой основан на нормальном случайном распределении.

> *Мы не будем вдаваться в формулы инициализации **Glorot** и **He**, но это хорошие темы, которые стоит рассмотреть для дальнейшего чтения (Glorot and Bengio, 2010; He et al., 2015b).*

Ранее мы видели, как мы можем инициализировать веса из равномерного случайного распределения в TensorFlow с помощью инициализатора, как это было сделано во фрагменте кода 7.4. Мы можем выбрать другой инициализатор, объявив любой из поддерживаемых инициализаторов в Keras. В частности, мы можем объявить инициализаторы Glorot и He следующим образом:

initializer = keras.initializers.glorot_uniform()

initializer = keras.initializers.he_normal()

Параметры для управления этими инициализаторами можно передать конструктору инициализатора. Кроме того, оба инициализатора Glorot и He представлены в двух вариантах: однородном и обычном. Мы выбрали униформу для Глорота и обычную форму для Хе.

Если вы не чувствуете необходимости настраивать какой-либо из параметров, то вообще не нужно объявлять объект инициализатора, но вы можете просто передать имя инициализатора в виде строки в функцию, в которой вы создаете слой. Это показано во фрагменте кода 7.5, где для аргумента kernel_initializer установлено значение «glorot_uniform».

Фрагмент кода 7.5 — Установка инициализатора путем передачи его имени в виде строки

```
model = keras.Sequential([

        keras.layers.Flatten(input_shape=(28, 28)),

        keras.layers.Dense(25, activation='tanh',

              kernel_initializer='glorot_uniform',
              bias_initializer='zeros'),

        keras.layers.Dense(10, activation='sigmoid',

              kernel_initializer='glorot_uniform',
              bias_initializer='zeros')])
```

Мы можем отдельно установить для bias_initializer любой подходящий инициализатор, но, как было сказано ранее, хорошей начальной рекомендацией будет просто инициализировать веса смещения равными 0, что и делает инициализатор 'zeros'.

Стандартизация входа

В дополнение к правильной инициализации весов важно предварительно обработать входные данные. В частности, стандартизация входных данных с центром вокруг 0 и с большинством значений, близких к 0, снизит риск

насыщения нейронов с самого начала. Мы уже использовали это в нашей реализации; давайте обсудим это немного подробнее. Как указывалось ранее, каждый пиксель в наборе данных MNIST представлен целым числом от 0 до 255, где 0 представляет собой чистый лист бумаги, а более высокое значение представляет собой пиксели, в которых была написана цифра.1 Большинство пикселей будут либо 0, либо значение близко к 255, где только края цифр находятся где-то посередине. Кроме того, большинство пикселей будут равны 0, потому что цифра разреженная и не покрывает все изображение 28×28. Если мы вычислим среднее значение пикселя для всего набора данных, то окажется, что оно составляет около 33. Понятно, что если бы мы использовали необработанные значения пикселей в качестве входных данных для наших нейронов, то был бы большой риск того, что нейроны будут далеко в область насыщения. Вычитая среднее значение и деля на стандартное отклонение, мы гарантируем, что нейронам будут представлены входные данные, которые находятся в области, не приводящей к насыщению.

Пакетная нормализация

Нормализация входных данных не обязательно предотвращает насыщение нейронов для скрытых слоев, и для решения этой проблемы Иоффе и Сегеди ввели пакетную нормализацию. Идея состоит в том, чтобы также нормализовать значения внутри сети и тем самым предотвратить насыщение скрытых нейронов. Это может показаться несколько нелогичным. Если мы нормализуем выход нейрона, не приведет ли это к отмене работы этого нейрона? Это было бы так, если бы это действительно была нормализация значений, но функция пакетной нормализации также содержит параметры для противодействия этому эффекту. Эти параметры настраиваются в процессе обучения. Примечательно, что после того, как первоначальная идея была опубликована, последующая работа показала, что причина, по которой пакетная нормализация работает, отличается от первоначального объяснения (Santurkar et al., 2018).

Это может показаться странным, поскольку значение 0 обычно соответствует черному цвету, а значение 255 — белому для изображения в градациях серого. Однако для этого набора данных это не так.

Существует два основных способа применения пакетной нормализации. В исходной статье предлагалось применить нормировку на входе к функции активации (после взвешенной суммы). Это показано слева на рис. 7.3.

Рисунок 7.3 — Слева: Нормализация пакета, представленная Иоффе и Сегеди (2015). Слой нейронов разбит на две части. Первая часть представляет собой взвешенные суммы для всех нейронов. К этим взвешенным суммам применяется пакетная нормализация. Функция активации (tanh) применяется к выходным данным операции нормализации пакета. Справа: пакетная нормализация применяется к выходным данным функций активации.

Это можно реализовать в Keras, создав экземпляр слоя без функции активации, за которым следует слой BatchNormalization, а затем применить функцию активации без каких-либо новых нейронов, используя слой активации. Это показано во фрагменте кода 7.6.

Фрагмент кода 7.6 — Пакетная нормализация перед функцией активации

```
keras.layers.Dense(64),
keras.layers.BatchNormalization(),

keras.layers.Activation('tanh'),
```

Однако оказывается, что пакетная нормализация также работает хорошо, если она выполняется после функции активации, как показано справа на рис. 7.3. Эта альтернативная реализация показана во фрагменте кода 7.7.

Фрагмент кода 7.7 — Пакетная нормализация после функции активации

```
keras.layers.Dense(64, activation='tanh'),
keras.layers.BatchNormalization(),
```

Задание

1. *Реализуйте полноценную программу по представленным фрагментам.*
2. *Согласно варианту N (по номеру в списке группы) модифицируйте модель следующим образом:*

   а) добавьте скрытые слои:

   Один слой для вариантов 1 – 5;

   Два слоя для вариантов 6 – 10;

   Три слоя для вариантов 11 – 15;

   Четыре слоя для вариантов 16 – 20.

   б) В скрытых слоях используйте функцию активации tanh, если N – нечётное и S-функцию, если вариант чётный;

   в) Добавьте инициализацию Глорота, если N – нечётное и Хе, если вариант чётный;

   г) Добавьте пакетную нормализацию перед функцией активации, если N – нечётное и после функции активации, если вариант чётный;

   д) реализуйте стандартизацию входных данных.

В отчёте приведите:

- *фрагменты модифицированного и дополнительно разработанного кода с комментариями,*
- *результаты текстовые и графические по всем моделям.*