МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Искусственные нейронные сети»

Тема: Распознавание объектов на фотографиях

Студент гр.8382 _____ Терехов А.Е.

Преподаватель _____ Жангриров Т.Р.

Санкт-Петербург

2021

Цель работы

Распознавание объектов на фотографиях (Object Recognition in Photographs) CIFAR-10 (классификация небольших изображений по десяти классам: самолет, автомобиль, птица, кошка, олень, собака, лягушка, лошадь, корабль и грузовик).

Задачи

- Ознакомиться со сверточными нейронными сетями.
- Изучить построение модели в Keras в функциональном виде.
- Изучить работу слоя разреживания (Dropout).

Требования

1. Построить и обучить сверточную нейронную сеть.
2. Исследовать работу сеть без слоя Dropout.
3. Исследовать работу сети при разных размерах ядра свертки.

Основные теоретические положения

В лабораторной 4 архитектуру “многослойный перцептрон” (MLP) применили к MNIST. Но все же полносвязный перцептрон обычно не выбирают для задач, связанных с распознаванием изображений — в этом случае намного чаще пользуются преимуществами сверточных нейронных сетей (Convolutional Neural Networks, CNN).

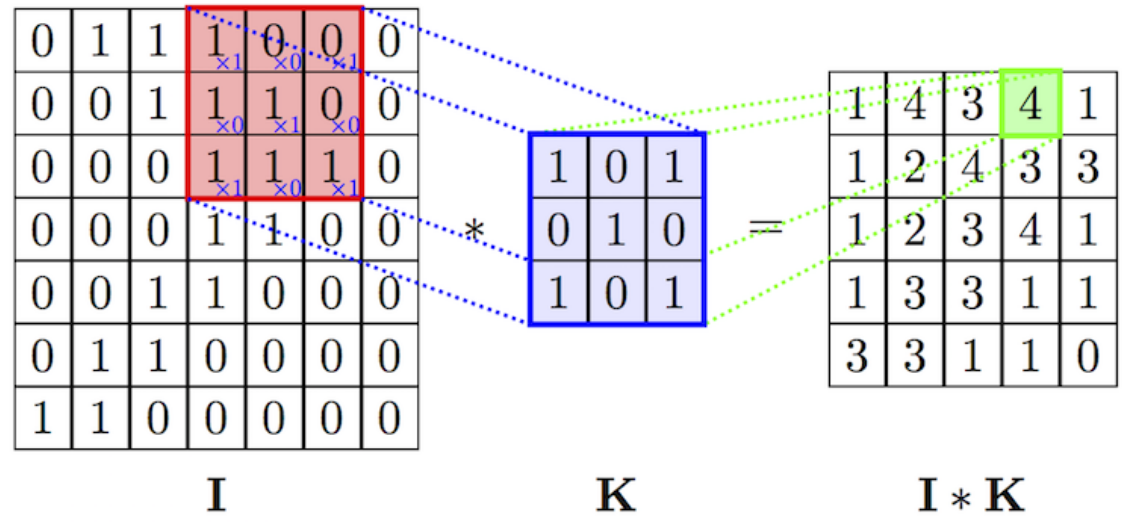
Посмотрим, что происходит с количеством параметров (весов) в модели MLP, когда ей на вход поступают необработанные данные. Например, CIFAR-10 содержит 32 x 32 x 3 пикселей, и если мы будем считать каждый канал каждого пикселя независимым входным параметром для MLP, каждый нейрон в

первом скрытом слое добавляет к модели около 3000 новых параметров. И с ростом размера изображений ситуация быстро выходит из-под контроля, причем происходит это намного раньше, чем изображения достигают того размера, с которыми обычно работают пользователи реальных приложений.

Одно из популярных решений — понижать разрешение изображений до той степени, когда MLP становится применим. Тем не менее, когда мы просто понижаем разрешение, мы рискуем потерять большое количество информации, и было бы здорово, если бы можно было осуществлять полезную первичную обработку информации еще до применения понижения качества, не вызывая при этом взрывного роста количества параметров модели.

Существует весьма эффективный способ решения этой задачи, который обращает в нашу пользу саму структуру изображения: предполагается, что пиксели, находящиеся близко друг к другу, теснее “взаимодействуют” при формировании интересующего нас признака, чем пиксели, расположенные в противоположных углах. Кроме того, если в процессе классификации изображения небольшая черта считается очень важной, не будет иметь значения, на каком участке изображения эта черта обнаружена.

Введем понятие оператора свертки. Имея двумерное изображение I и небольшую матрицу K размерности $h \times w$ (так называемое ядро свертки), построенная таким образом, что графически кодирует какой-либо признак, мы вычисляем свернутое изображение $I * K$, накладывая ядро на изображение всеми возможными способами и записывая сумму произведений элементов исходного изображения и ядра. Результат применения операции свертки (с двумя разными ядрами) к изображению с целью выделить контуры объекта:



Сверточные и субдискретизирующие слои

Оператор свертки составляет основу сверточного слоя (convolutional layer) в CNN. Слой состоит из определенного количества ядер $\vec{K}$ (с аддитивными составляющими смещения $\vec{b}$ для каждого ядра) и вычисляет свертку выходного изображения предыдущего слоя с помощью каждого из ядер, каждый раз прибавляя составляющую смещения. В конце концов ко всему выходному изображению может быть применена функция активации σ. Обычно входной поток для сверточного слоя состоит из d каналов, например, red/green/blue для входного слоя, и в этом случае ядра тоже расширяют таким образом, чтобы

они также состояли из d каналов; получается следующая формула для одного канала выходного изображения сверточного слоя, где K — ядро, а b — составляющая смещения:

$$\mathrm{conv}(I,K)_{x,y} = \sigma(b + \sum_{i=1}^{h}\sum_{j=1}^{w}\sum_{k=1}^{d} K_{ijk} \times I_{x+i-1,y+j-1,k})$$

Обратите внимание, что так как все, что мы здесь делаем — это сложение и масштабирование входных пикселей, ядра можно получить из имеющейся обучающей выборки методом градиентного спуска, аналогично вычислению весов в многослойном перцептроне (MLP). На самом деле MLP мог бы в совершенстве справиться с функциями сверточного слоя, но времени на обучение (как и обучающих данных) потребовалось бы намного больше.

Заметим также, что оператор свертки вовсе не ограничен двухмерными данными: большинство фреймворков глубокого обучения (включая Keras) предоставляют слои для одномерной или трехмерной свертки прямо “из коробки”.

Стоит также отметить, что хотя сверточный слой сокращает количество параметров по сравнению с полносвязным слоем, он использует больше гиперпараметров — параметров, выбираемых до начала обучения.

В частности, выбираются следующие гиперпараметры:

Глубина (depth) — сколько ядер и коэффициентов смещения будет задействовано в одном слое;

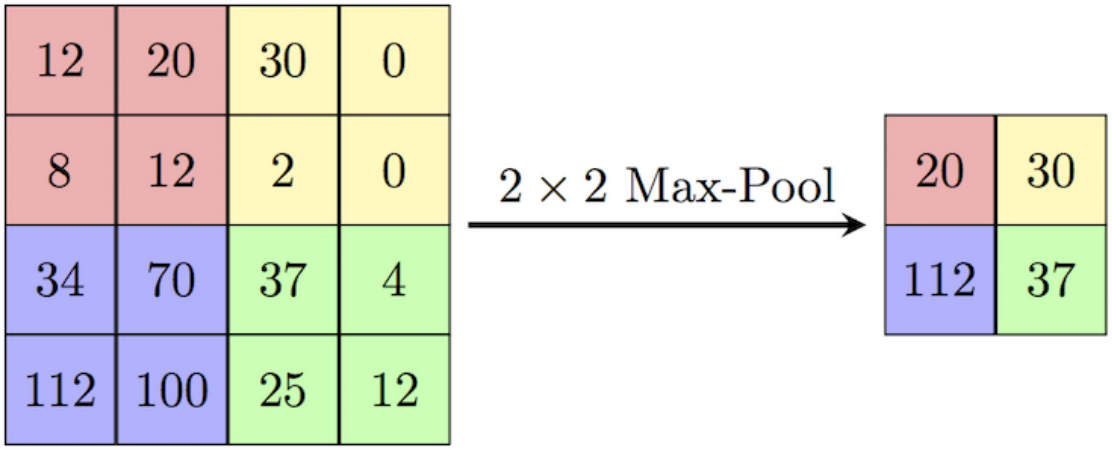
Высота (height) и ширина (width) каждого ядра;

Шаг (stride) — на сколько смещается ядро на каждом шаге при вычислении следующего пикселя результирующего изображения. Обычно его принимают равным 1, и чем больше его значение, тем меньше размер выходного изображения;

Отступ (padding): заметим, что свертка любым ядром размерности более, чем 1x1 уменьшит размер выходного изображения. Так как в общем случае желательно сохранять размер исходного изображения, рисунок дополняется нулями по краям.

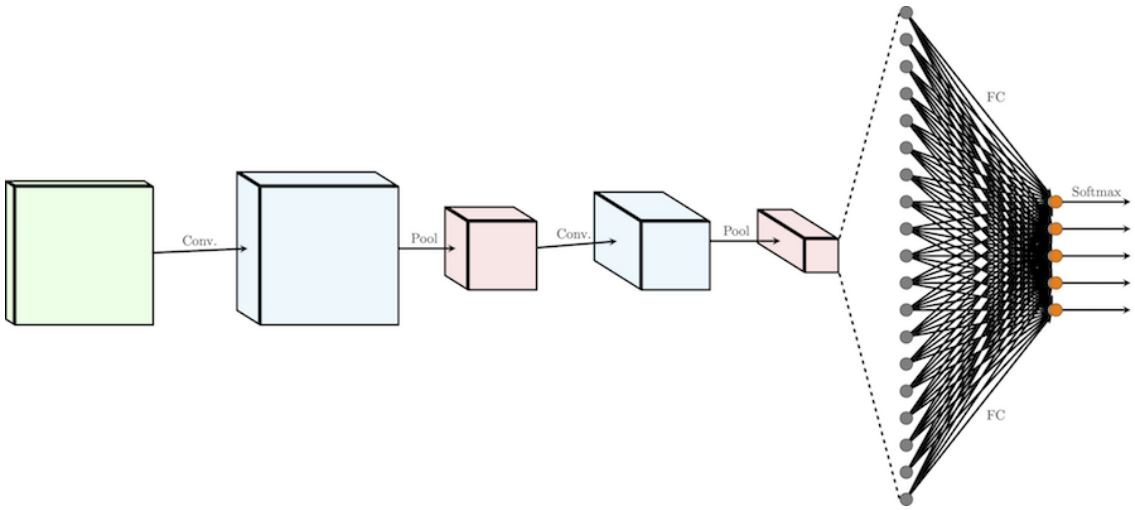
Операции свертки — не единственные операции в CNN (хотя существуют многообещающие исследования на тему “чисто-сверхточных” сетей); они чаще применяются для выделения наиболее полезных признаков перед субдискретизацией (downsampling) и последующей обработкой с помощью MLP.

Популярный способ субдискретизации изображения — слой подвыборки (также называемый слоем субдискретизации, по-английски downsampling или pooling layer), который получает на вход маленькие отдельные фрагменты изображения (обычно 2x2) и объединяет каждый фрагмент в одно значение. Существует несколько возможных способов агрегации, наиболее часто из четырех пикселей выбирается максимальный. Этот способ схематически изображен ниже.



Итого: обычная CNN

Теперь, когда у нас есть все строительные блоки, давайте рассмотрим, как выглядит обычная CNN целиком.



Обычную архитектуру CNN для распределения изображений по k классам можно разделить на две части: цепочка чередующихся слоев свертки/-подвыборки Conv -> Pool (иногда с несколькими слоями свертки подряд) и несколько полносвязных слоев (принимающих каждый пиксель как независимое значение) с слоем softmax в качестве завершающего. Я не говорю здесь о функциях активации, чтобы наша схема стала проще, но не забывайте, что обычно после каждого сверточного или полносвязного слоя ко всем выходным значениям применяется функция активации, например, ReLU.

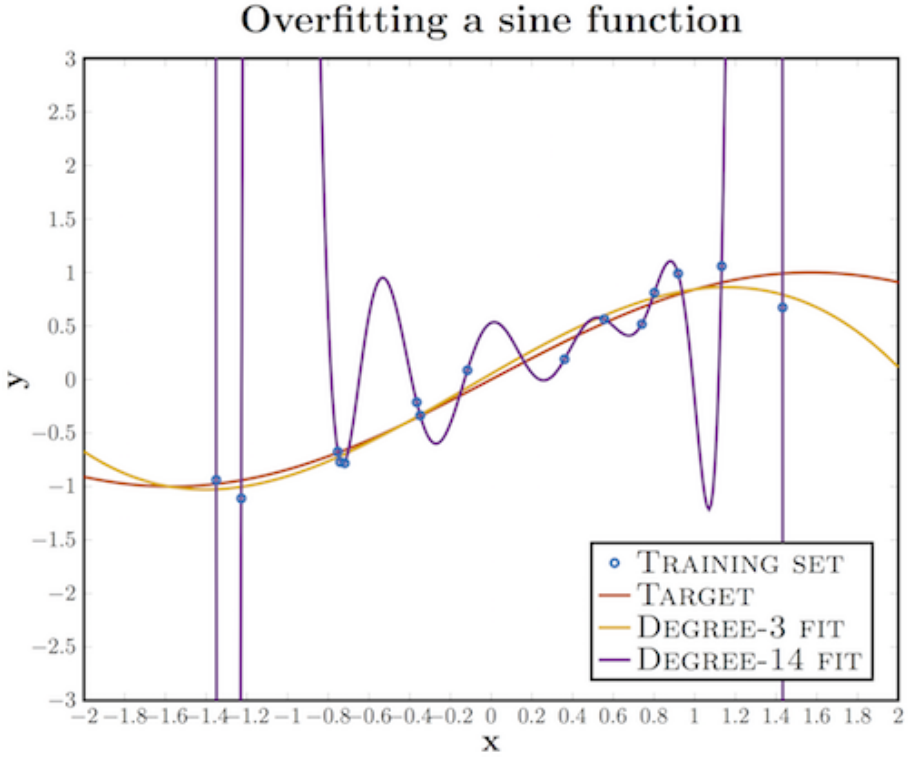
Один проход Conv -> Pool влияет на изображение следующим образом: он сокращает длину и ширину определенного канала, но увеличивает его значение (глубину).

Softmax и перекрестная энтропия более подробно рассмотрены на предыдущем уроке. Напомним, что функция softmax превращает вектор действительных чисел в вектор вероятностей (неотрицательные действительные числа, не превышающие 1). В нашем контексте выходные значения являются вероятностями попадания изображения в определённый класс. Минимизация потерь перекрестной энтропии обеспечивает уверенность в определении принадлежности изображения определенному классу, не принимая во внима-

ние вероятность остальных классов, таким образом, для вероятностных задач softmax предпочтительней, чем, например, метод квадратичной ошибки.

Переобучение, регуляризация и dropout:

Переобучение — это излишне точное соответствие нейронной сети конкретному набору обучающих примеров, при котором сеть теряет способность к обобщению. Другими словами, наша модель могла выучить обучающее множество (вместе с шумом, который в нем присутствует), но она не смогла распознать скрытые процессы, которые это множество породили. В качестве примера рассмотрим задачу аппроксимации синусоиды с аддитивным шумом.

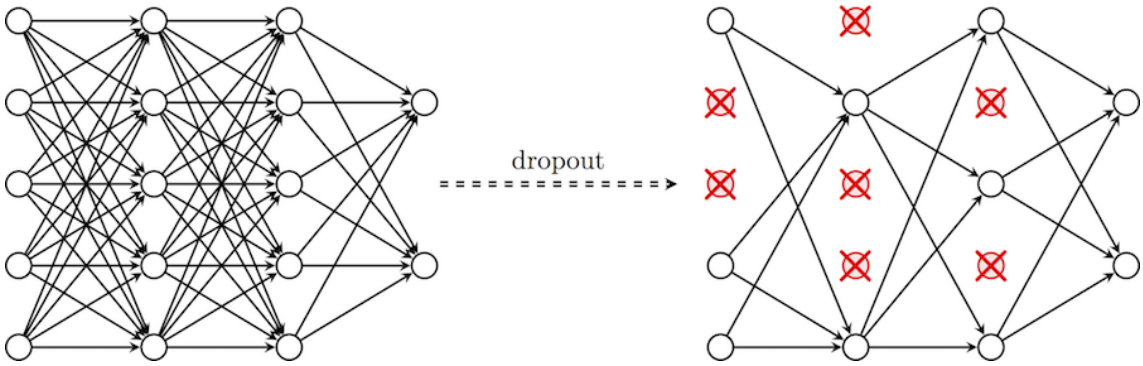


У нас есть обучающее множество (синие кружки), полученное из исходной кривой синуса, с некоторым количеством шума. Если мы приложим к этим данным график многочлена третьей степени, мы получим хорошую аппроксимацию исходной кривой. Кто-то возразит, что многочлен 14-й степени подошел бы лучше; действительно, так как у нас есть 15 точек, такая аппроксимация идеально описала бы обучающую выборку. Тем не менее, в

этом случае введение дополнительных параметров в модель приводит к катастрофическим результатам: из-за того, что наша аппроксимация учитывает шумы, она не совпадает с исходной кривой нигде, кроме обучающих точек.

У глубоких сверточных нейронных сетей масса разнообразных параметров, особенно это касается полносвязных слоев. Переобучение может проявить себя в следующей форме: если у нас недостаточно обучающих примеров, маленькая группа нейронов может стать ответственной за большинство вычислений, а остальные нейроны станут избыточны; или наоборот, некоторые нейроны могут нанести ущерб производительности, при этом другие нейроны из их слоя не будут заниматься ничем, кроме исправления их ошибок.

Чтобы помочь нашей сети не утратить способности к обобщению в этих обстоятельствах, мы вводим приемы регуляризации: вместо сокращения количества параметров, мы накладываем ограничения на параметры модели во время обучения, не позволяя нейронам изучать шум обучающих данных. Здесь я опишу прием dropout, который сначала может показаться “черной магией”, но на деле помогает исключить ситуации, описанные выше. В частности, dropout с параметром p за одну итерацию обучения проходит по всем нейронам определенного слоя и с вероятностью p полностью исключает их из сети на время итерации. Это заставит сеть обрабатывать ошибки и не полагаться на существование определенного нейрона (или группы нейронов), а полагаться на “единое мнение” (consensus) нейронов внутри одного слоя. Это довольно простой метод, который эффективно борется с проблемой переобучения сам, без необходимости вводить другие регуляризаторы. Схема ниже иллюстрирует данный метод.



Реализация сети

В качестве практической части построим глубокую сверточную нейронную сеть и применим ее к классификации изображений из набора CIFAR-10.

Импорты:

```
from keras.datasets import cifar10
from keras.models import Model
from keras.layers import Input, Convolution2D, MaxPooling2D,
   Dense, Dropout, Flatten
from keras.utils import np_utils
import numpy as np
```

Как уже говорилось, обычно CNN использует больше гиперпараметров, чем MLP. В этом руководстве мы все еще будем использовать заранее известные “хорошие” значения, но не будем забывать, что в последующей лекции я расскажу, как их правильно выбирать.

Зададим следующие гиперпараметры:

- $batch\_size$ — количество обучающих образцов, обрабатываемых одновременно за одну итерацию алгоритма градиентного спуска;
- num_epochs — количество итераций обучающего алгоритма по всему обучающему множеству;
- $kernel\_size$ — размер ядра в сверточных слоях;
- $pool\_size$ — размер подвыборки в слоях подвыборки;

- $conv\_depth$ — количество ядер в сверточных слоях;
- $drop\_prob$ (dropout probability) — мы будем применять dropout после каждого слоя подвыборки, а также после полносвязного слоя;
- $hidden\_size$ — количество нейронов в полносвязном слое MLP.

```
batch_size = 32
num_epochs = 200
kernel_size = 3
pool_size = 2
conv_depth_1 = 32
conv_depth_2 = 64
drop_prob_1 = 0.25
drop_prob_2 = 0.5
hidden_size = 512
```

Загрузка и первичная обработка CIFAR-10 осуществляется ровно так же, как и загрузка и обработка MNIST, где Keras выполняет все автоматически. Единственное отличие состоит в том, что теперь мы не рассматриваем каждый пиксель как независимое входное значение, и поэтому мы не переносим изображение в одномерное пространство. Мы снова преобразуем интенсивность пикселей так, чтобы она попадала в отрезок [0,1] и используем прямое кодирование для выходных значений.

Тем не менее, в этот раз этот этап будет выполнен для более общего случая, что позволит проще приспосабливаться к новым наборам данных: размер будет не жестко задан, а вычислен из размера набора данных, количество классов будет определено по количеству уникальных меток в обучающем множестве, а нормализация будет выполнена путем деления всех элементов на максимальное значение обучающего множества.

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

```
num_train, depth, height, width = X_train.shape
num_test = X_test.shape[0]
num_classes = np.unique(y_train).shape[0]

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= np.max(X_train)
X_test /= np.max(X_train)

Y_train = np_utils.to_categorical(y_train, num_classes)
Y_test = np_utils.to_categorical(y_test, num_classes)
```

Настало время моделирования! Наша сеть будет состоять из четырех слоев $Convolution_2D$ и слоев $MaxPooling2D$ после второй и четвертой сверток. После первого слоя подвыборки мы удваиваем количество ядер (вместе с описанным выше принципом принесения высоты и ширины в жертву глубине). После этого выходное изображение слоя подвыборки трансформируется в одномерный вектор (слоем Flatten) и проходит два полносвязных слоя (Dense). На всех слоях, кроме выходного полносвязного слоя, используется функция активации ReLU, последний же слой использует softmax.

Для регуляризации нашей модели после каждого слоя подвыборки и первого полносвязного слоя применяется слой Dropout. Здесь Keras также выделяется на фоне остальных фреймворков: в нем есть внутренний флаг, который автоматически включает и выключает dropout, в зависимости от того, находится модель в фазе обучения или тестирования.

```
inp = Input(shape=(depth, height, width))
conv_1 = Convolution2D(conv_depth_1, kernel_size,
                       kernel_size, padding='same',
                       activation='relu')(inp)
conv_2 = Convolution2D(conv_depth_1, kernel_size,
```

```
                        kernel_size, padding='same',
                        activation='relu')(conv_1)
pool_1 = MaxPooling2D(pool_size=(pool_size, pool_size))(conv_2)
drop_1 = Dropout(drop_prob_1)(pool_1)
conv_3 = Convolution2D(conv_depth_2, kernel_size,
                        kernel_size, padding='same',
                        activation='relu')(drop_1)
conv_4 = Convolution2D(conv_depth_2, kernel_size,
                        kernel_size, padding='same',
                        activation='relu')(conv_3)
pool_2 = MaxPooling2D(pool_size=(pool_size, pool_size), padding='
   same')(conv_4)
drop_2 = Dropout(drop_prob_1)(pool_2)
flat = Flatten()(drop_2)
hidden = Dense(hidden_size, activation='relu')(flat)
drop_3 = Dropout(drop_prob_2)(hidden)
out = Dense(num_classes, activation='softmax')(drop_3)

model = Model(inputs=inp, outputs=out)
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(X_train, Y_train,
          batch_size=batch_size, epochs=num_epochs,
          verbose=1, validation_split=0.1)
model.evaluate(X_test, Y_test, verbose=1)
```

Ход работы

В соответствии с теоретическими положениями построим и обучим модель. Схема модели представлена на рисунке 1.

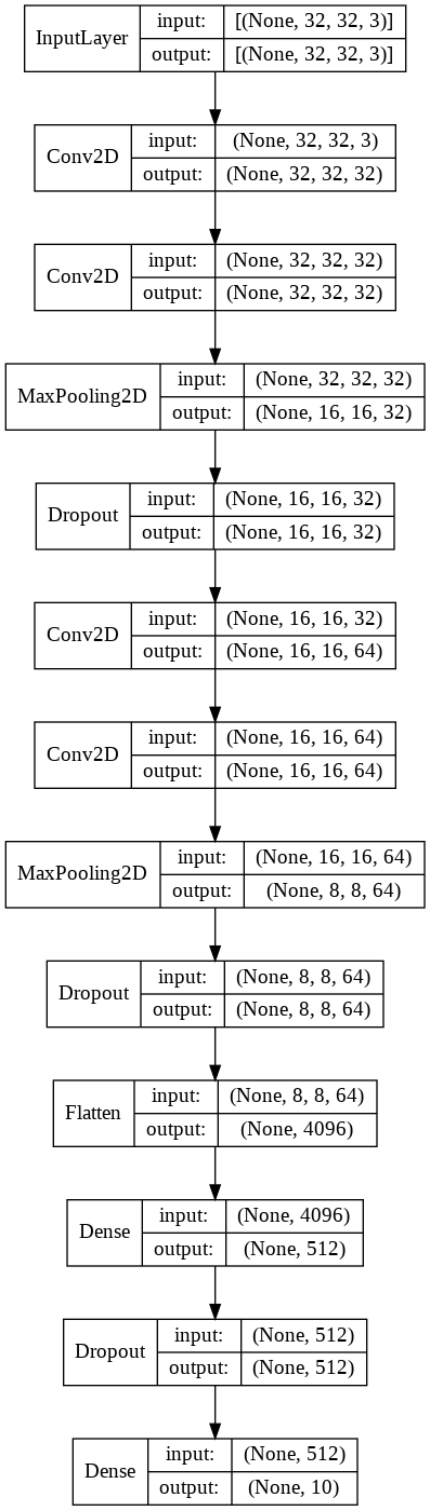


Рис. 1 – Схема модели

Полученные результаты при обучении на 200 эпохах представлены в таблице 1 и на рисунках 2-3.

Таблица 1 – Точность сети

Train	Validation	Test
0.9324	0.7886	0.4497

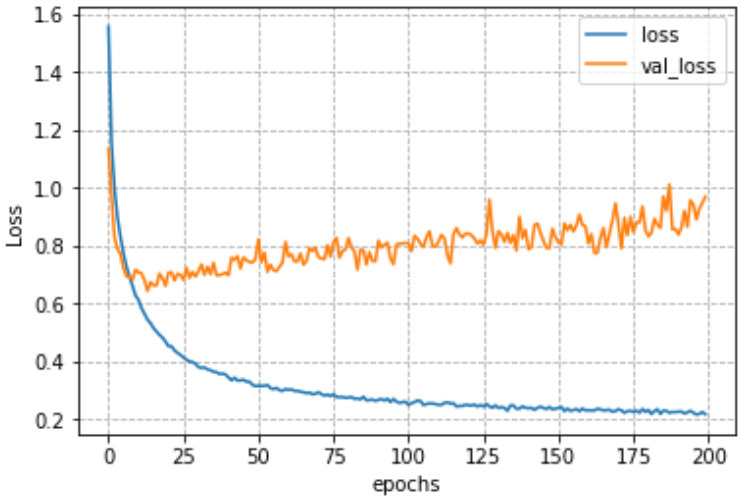


Рис. 2 – График ошибок

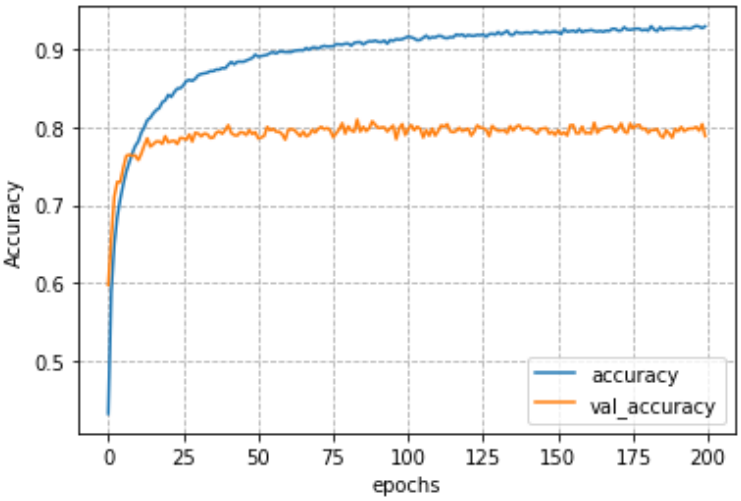


Рис. 3 – График точности

Очевидно, что 200 эпох это слишком много и модель попросту переобучалась. Поэтому сократим число эпох до 16 и еще раз обучим модель. Полученные результаты представлены в таблице 2 и на рисунках 4-5.

Таблица 2 – Точность сети

Train	Validation	Test
0.8387	0.7972	0.6239

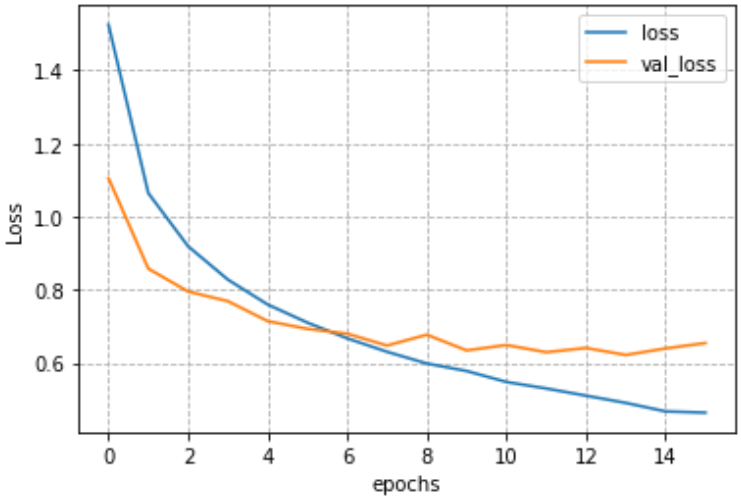


Рис. 4 – График ошибок

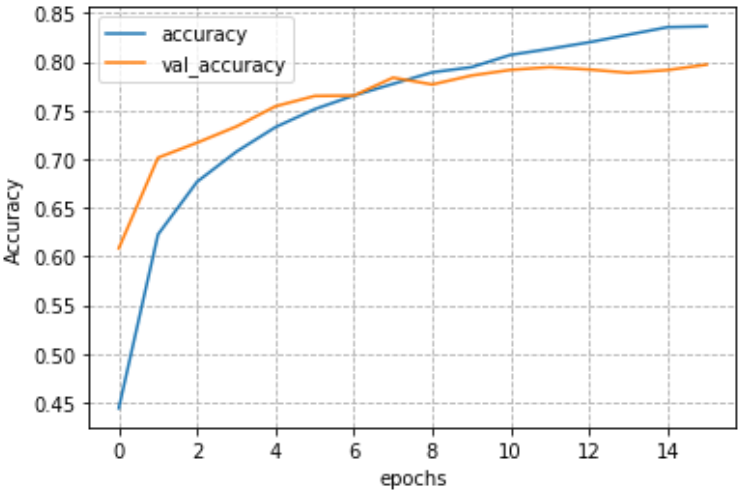


Рис. 5 – График точности

Удалим слои Dropout и еще раз обучим сеть. Полученные результаты представлены в таблице 3 и на рисунках 6-7.

Таблица 3 – Точность сети

Train	Validation	Test
0.9863	0.6746	0.5993

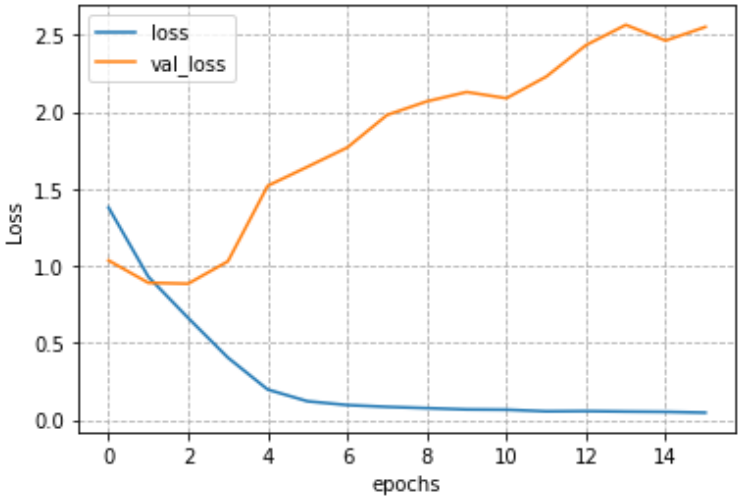


Рис. 6 – График ошибок

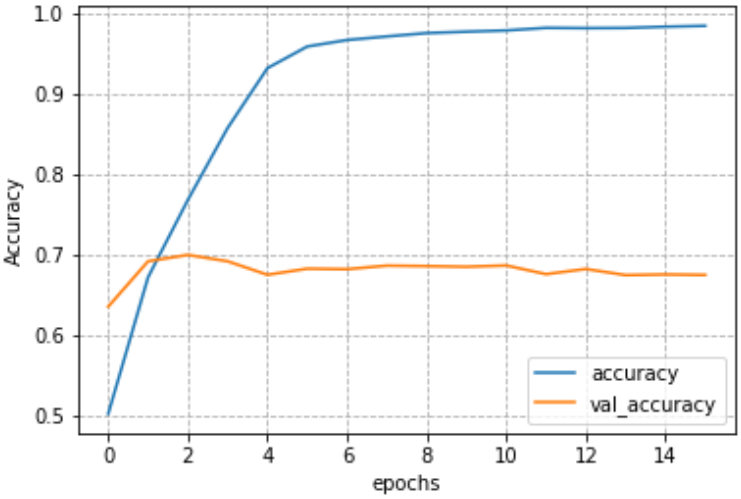


Рис. 7 – График точности

Очевидно, что без слоев прореживания сеть переобучилась, причем очень быстро. На тренировочных данных точность сильно возросла, но на тестовых упала, что и ожидалось.

Исследуем работу сети при размерах ядра свертки 2×2, 3×3 (исходный), 4×4. Полученные результаты представлены в таблицах 4-5 и на рисунках 8-11.

Таблица 4 – Точность сети при размере ядра 2×2

Train	Validation	Test
0.8166	0.7806	0.4640

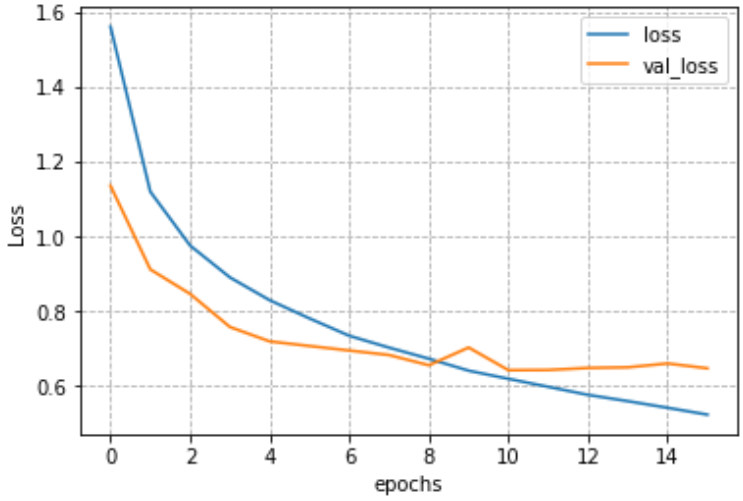


Рис. 8 – График ошибок при размере ядра 2×2

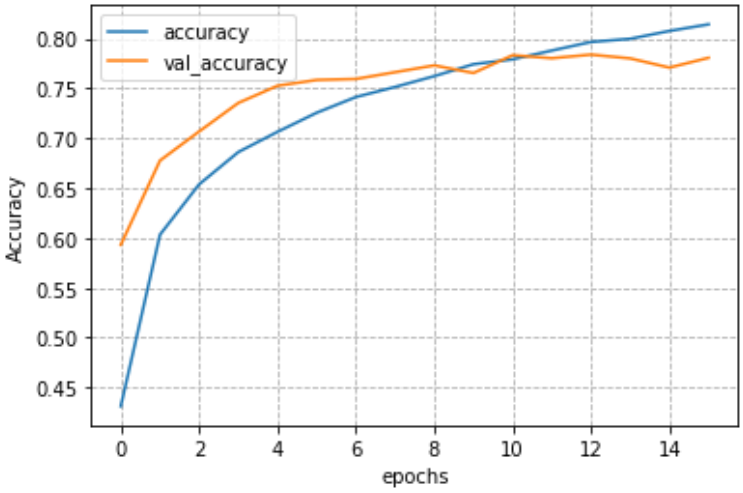


Рис. 9 – График точности при размере ядра 2×2

Таблица 5 – Точность сети при размере ядра 4×4

Train	Validation	Test
0.8036	0.7728	0.5514

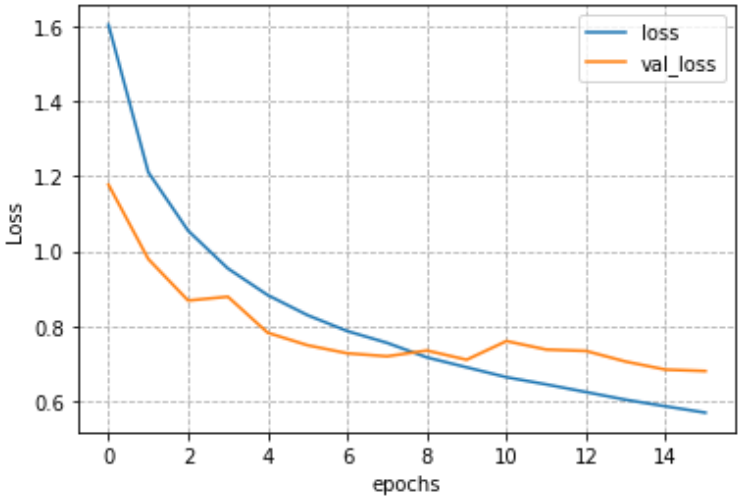


Рис. 10 – График ошибок при размере ядра 4×4

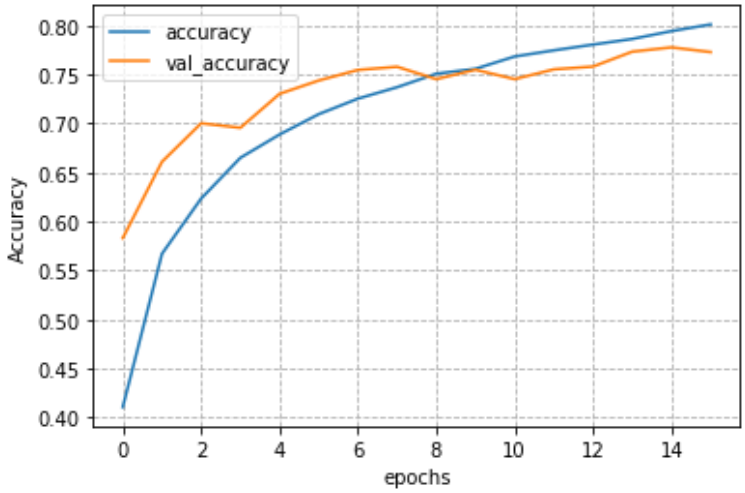


Рис. 11 – График точности при размере ядра 4×4

Исходя из полученных результатов можно сделать вывод, что оптимальным размером ядра является 3×3, так как при таком размере была достигнута максимальная точность 0.6239.

Вывод

В ходе работы были изучены сверточные нейронные сети. Модели строились в функциональном виде. Было исследовано влияние слоя разреживания (Dropout), без данного слоя сеть очень быстро переобучалась. При изменении размеров ядра свертки был определен оптимальный – 3×3.

ПРИЛОЖЕНИЕ А

```
from keras.utils import np_utils, plot_model
from keras.datasets import cifar10
from keras.models import Model
from keras.layers import Input, Convolution2D, MaxPooling2D,
   Dense, Dropout, Flatten

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np


def plot(data: pd.DataFrame, label: str):
    axis = sns.lineplot(data=data, dashes=False)
    axis.set(ylabel=label, xlabel='epochs')
    axis.grid(True, linestyle="--")
    plt.show()


if __name__ == '__main__':
    batch_size = 32  # in each iteration, we consider 32 training
        examples at once
    num_epochs = 50  # we iterate 200 times over the entire
       training set
    kernel_size = 3  # we will use 3x3 kernels throughout
    pool_size = 2  # we will use 2x2 pooling throughout
    conv_depth_1 = 32  # we will initially have 32 kernels per
       conv. layer...
    conv_depth_2 = 64  # ...switching to 64 after the first
       pooling layer
    drop_prob_1 = 0.25  # dropout after pooling with probability
```

```
    0.25
drop_prob_2 = 0.5  # dropout in the dense layer with
    probability 0.5
hidden_size = 512  # the dense layer will have 512 neurons
(X_train, y_train), (X_test, y_test) = cifar10.load_data()  #
    fetch CIFAR-10 data

num_train, depth, height, width = X_train.shape  # there are
    50000 training examples in CIFAR-10
num_test = X_test.shape[0]  # there are 10000 test examples
    in CIFAR-10
num_classes = np.unique(y_train).shape[0]  # there are 10
    image classes

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= np.max(X_train)  # Normalise data to [0, 1] range
X_test /= np.max(X_train)  # Normalise data to [0, 1] range

Y_train = np_utils.to_categorical(y_train, num_classes)  #
    One-hot encode the labels
Y_test = np_utils.to_categorical(y_test, num_classes)  # One-
    hot encode the labels
inp = Input(shape=(depth, height, width))  # N.B. depth goes
    first in Keras
# Conv [32] -> Conv [32] -> Pool (with dropout on the pooling
    layer)
conv_1 = Convolution2D(conv_depth_1, kernel_size, kernel_size
    , padding='same', activation='relu')(inp)
conv_2 = Convolution2D(conv_depth_1, kernel_size, kernel_size
    , padding='same', activation='relu')(conv_1)
pool_1 = MaxPooling2D(pool_size=(pool_size, pool_size))(
```

```
    conv_2)
drop_1 = Dropout(drop_prob_1)(pool_1)
# Conv [64] -> Conv [64] -> Pool (with dropout on the pooling
    layer)
conv_3 = Convolution2D(conv_depth_2, kernel_size, kernel_size
    , padding='same', activation='relu')(drop_1)
conv_4 = Convolution2D(conv_depth_2, kernel_size, kernel_size
    , padding='same', activation='relu')(conv_3)
pool_2 = MaxPooling2D(pool_size=(pool_size, pool_size),
    padding='same')(conv_4)
drop_2 = Dropout(drop_prob_1)(pool_2)
# Now flatten to 1D, apply Dense -> ReLU (with dropout) ->
    softmax
flat = Flatten()(drop_2)
hidden = Dense(hidden_size, activation='relu')(flat)
drop_3 = Dropout(drop_prob_2)(hidden)
out = Dense(num_classes, activation='softmax')(drop_3)

model = Model(inputs=inp, outputs=out)  # To define a model,
    just specify its input and output layers
model.compile(loss='categorical_crossentropy',  # using the
    cross-entropy loss function
              optimizer='adam',  # using the Adam optimiser
              metrics=['accuracy'])  # reporting the accuracy
H = model.fit(X_train, Y_train,  # Train the model using the
    training set...
              batch_size=batch_size, epochs=num_epochs,
              verbose=1, validation_split=0.1)  # ...holding
                 out 10% of the data for validation
model.evaluate(X_test, Y_test, verbose=1)  # Evaluate the
    trained model on the test set!
plot_model(model, show_shapes=True, rankdir="TB",
```

```
   expand_nested=True, show_layer_names=False)
df_history = pd.DataFrame(H.history)
plot(df_history[['loss', 'val_loss']], "Loss")
plot(df_history[['accuracy', 'val_accuracy']], "Accuracy")
```