



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی برق

گزارش کار آزمایش ششم
آزمایشگاه مقدمه ای بر هوش محاسباتی

پیاده سازی CNN

نگارش
ارشیا اسمعیل طهرانی
علی بابالو
پویا ابراهیمی

استاد راهنما
سرکار خانم موسوی

آذر ماه 1401

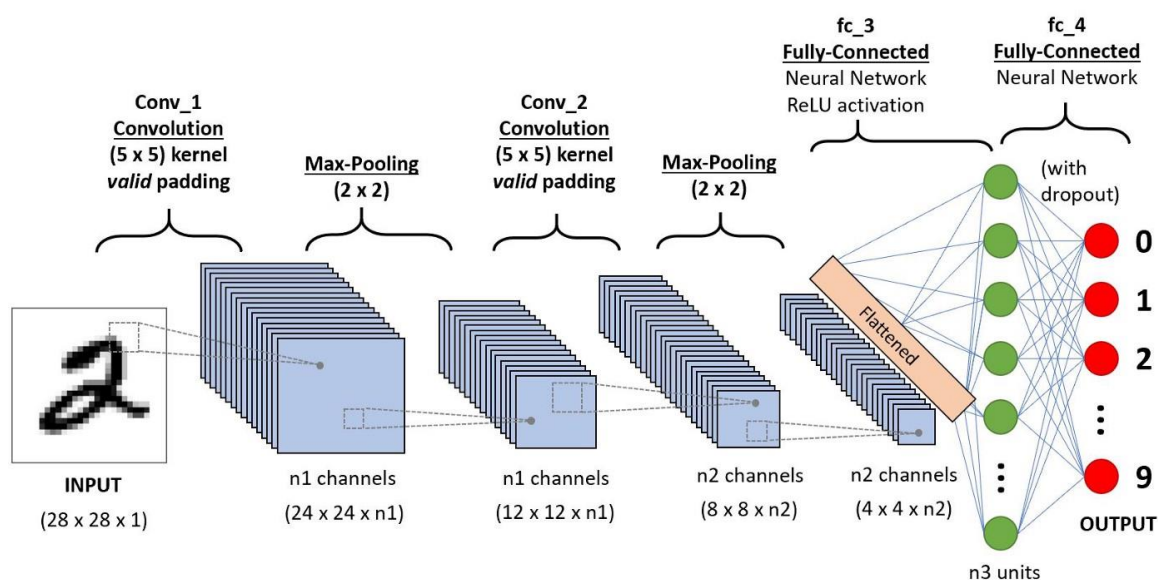
چکیده

در این آزمایش هدف پیاده سازی یک شبکه کانولوشنی روی (سی فار 10) است. شبکه عصبی کانولوشنی از پرکاربردترین شبکه های عصبی برای پردازش تصویر می باشد. عملکرد بسیاری از حملات نیز بر روی این شبکه عصبی بررسی می شوند .

صفحه	فهرست مطالب
1	پیش گزارش.....
1	ساختار یک شبکه عصبی CNN.....
2	لایه کانوولوشنی.....
2	لایه ادغام.....
3	لایه کاملاً متصل.....
3	شرح آزمایش.....
4	معماری VGG.....
4	Dropout Regularization.....
5	تنوع Dropout Regularization.....
5	Batch Normalization.....
5	Adam Learning Rate.....
6	محیط Python.....
6	دیتاست mnist.....
11	تمرین.....
11	دیتاست CIFAR-10.....
21	تمرین ۲.....

2-1- ساختار یک شبکه عصبی CNN.

شبکه کانولوشنی از لایه های اصلی زیر تشکیل شده است:

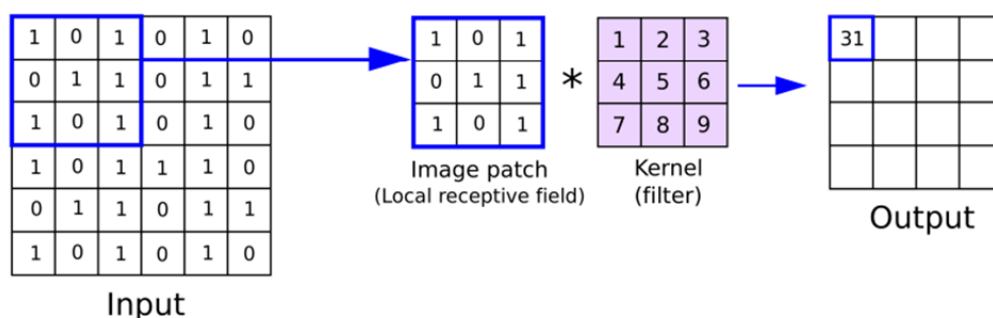


شکل 1) نمونه ای شبکه عصبی کانولوشنی

- لایه کانولوشنی
- لایه ادغام
- لایه کاملاً متصل

2-3-1- لایه کانولوشنی:

در این لایه عمل کانولوشن بر روی داده ورودی و با استفاده از تعدادی فیلتر انجام می شود. برای محاسبه هر درایه خروجی، ماتریس فیلتر بر روی ماتریس ورودی لغزانده می شود. عمل کانولوشن به این صورت تعریف می شود که ابتدا اولین عنصر فیلتر بر روی اولین عنصر ماتریس ورودی قرار می گیرد. سپس مجموع ضرب درایه های متناظر فیلتر با درایه های متناظر ماتریس ورودی محاسبه می شود. در نهایت فیلتر بر روی تصویر ورودی به اندازه پارامتر از پیش تعیین شده s به جلو برده می شود. با تکرار این مراحل ماتریس خروجی محاسبه می شود.



شکل 2) عملگر کانولوشن

2-3-2- لایه ادغام:

یکی دیگر از لایه های شبکه کانولوشنی لایه ادغام است. این لایه هیچ پارامتر آموزشی ندارد. هدف این لایه کاهش ابعاد ماتریس ورودی و همزمان حفظ اطلاعات ارزشمند ورودی است. در این لایه ابتدا ماتریس با ابعاد از پیش تعیین شده k در نظر گرفته می شود. این پارامتر معمولاً برابر با ۲ در نظر گرفته می شود. سپس با لغزاندن فیلتر بر روی ورودی اندازه ماتریس ورودی کاهش می یابد. یک نمونه از عملگرهایی که برای نمونه برداری در این لایه استفاده می شود، عملگر بیشینه است. در این حالت ماتریس از پیش تعیین شده بر روی داده ورودی لغزانده می شود و تنها بیشینه عناصری که در هر بخش قرار می گیرند را به عنوان خروجی در نظر می گیرد.

2-3-3- لایه کاملاً متصل:

در این لایه یک شبکه عصبی کاملاً متصل قرار گرفته است. در این لایه هدف مرتبط کردن ماتریس نهایی با خروجی های نهایی شبکه است. وزن های شبکه کاملاً متصل از طریق پس انتشار خطا بدست می آید.

شرح آزمایش

CIFAR مخفف عبارت Canadian Institute for Advanced Research است و مجموعه داده CIFAR-10 همراه با مجموعه داده CIFAR-100 توسط محققان موسسه CIFAR توسعه یافته است.

مجموعه داده شامل 60000 عکس رنگی 32×32 پیکسل از اشیاء از 10 کلاس، مانند قورباغه، پرندگان، گربه ها، کشتی ها و غیره است. برچسب های کلاس و مقادیر عدد صحیح مرتبط با آنها در زیر فهرست شده اند.

1. هواپیما
2. خودرو
3. پرنده
4. گربه
5. آهو
6. سگ
7. قورباغه
8. اسب
9. کشتی
10. کامیون

این تصاویر بسیار کوچک هستند، بسیار کوچکتر از یک عکس معمولی، و مجموعه داده برای تحقیقات بینایی کامپیوتری در نظر گرفته شده است.

CIFAR-10 یک مجموعه داده کاملاً درک شده است که به طور گسترده برای محک زدن الگوریتم‌های بینایی رایانه در یادگیری ماشین استفاده می‌شود. مشکل حل شده است. "دستیابی به دقت طبقه بندی 80 درصد نسبتاً ساده است. عملکرد برتر در مورد مشکل با یادگیری عمیق شبکه های عصبی کانولوشن با دقت طبقه بندی بالای 90٪ در مجموعه داده آزمایشی به دست می آید. اکنون می‌توانیم یک مدل پایه برای مجموعه داده CIFAR-10 بررسی کنیم.

2-2 معماری VGG

یک مدل پایه حداقل عملکرد مدل را ایجاد می کند که همه مدل های دیگر ما را می توان با آن مقایسه کرد و یک معماری مدلی که می توانیم به عنوان مبنای مطالعه و بهبود استفاده کنیم. یک نقطه شروع خوب، اصول کلی معماری مدل های VGG است. اینها نقاط شروع خوبی هستند زیرا در رقابت ILSVRC 2014 به عملکرد برتر دست یافتند و به دلیل اینکه ساختار مدولار معماری قابل درک و پیاده سازی آسان است.

این معماری شامل انباشته شدن لایه‌های کانولوشن با فیلترهای کوچک 3×3 و به دنبال آن یک لایه ترکیبی حداکثر است. این لایه‌ها با هم یک بلوک را تشکیل می‌دهند و این بلوک‌ها می‌توانند در جایی که تعداد فیلترها در هر بلوک با عمق شبکه افزایش می‌یابد، تکرار شوند، مانند 32، 64، 128 و 256 برای چهار بلوک اول مدل. بالشتک روی لایه های کانولوشن برای اطمینان از مطابقت ارتفاع و عرض نقشه های ویژگی خروجی با ورودی ها استفاده می شود.

ما این معماری را در مسئله CIFAR-10 بررسی می کنیم و مدلی را با این معماری با بلوک های 1، 2 و 3 مقایسه می کنیم.

2-3 Dropout regularization از جمله راه های بهبود مدل

Dropout یک تکنیک ساده است که به طور تصادفی گره ها را از شبکه خارج می کند. این یک اثر منظم کننده دارد زیرا گره های باقی مانده باید برای برداشتن سستی گره های حذف شده سازگار شوند.

Dropout را می توان با افزودن لایه های Dropout جدید به مدل اضافه کرد، جایی که مقدار گرهِ های حذف شده به عنوان یک پارامتر مشخص می شود. الگوهای زیادی برای افزودن Dropout به یک مدل وجود دارد، از نظر اینکه در کجای مدل باید لایه ها را اضافه کرد و از چه تعداد حذفی استفاده کرد. در این حالت، لایه های Dropout را بعد از هر لایه جمع آوری حداکثر و بعد از لایه کاملاً متصل اضافه می کنیم و از نرخ خروج ثابت 20٪ استفاده می کنیم (به عنوان مثال، 80٪ گرهِ ها را حفظ می کنیم).

2-4- تنوع dropout regularization

یک تغییر این است که میزان dropout از 20٪ به 25٪ یا 30٪ افزایش پیدا کند. یکی دیگر از تغییراتی که ممکن است جالب باشد، استفاده از الگوی افزایش dropout از 20٪ برای بلوک اول، 30٪ برای بلوک دوم و به همین ترتیب به 50٪ در لایه کاملاً متصل در بخش طبقه بندی کننده مدل است. این نوع افزایش dropout با عمق مدل یک الگوی معمولی است. موثر است زیرا لایه های عمیق مدل را مجبور می کند تا بیش از لایه های نزدیک به ورودی را منظم کنند.

2-5- استفاده از Batch Normalization

نرمال سازی دسته ای در تلاش برای تثبیت یادگیری و شاید تسریع روند یادگیری اضافه شده است. برای جبران این شتاب، از الگوی dropout فزاینده استفاده می شود. در ادامه به دو راهکار دیگر که برای تقویت این مدل استفاده شده است نیز پرداخته شده است؛

2-6- استفاده از Adam learning rate

بهینه سازی آدام یک روش stochastic gradient descent است که مبتنی بر تخمین تطبیقی همان های مرتبه اول و مرتبه دوم است. به خاطر ذات تطبیقی یا adaptive بسیار مورد استفاده قرار می گیرد به گونه ای که به گفته کینگما و همکاران، 2014، این روش "از نظر محاسباتی کارآمد است، نیاز به حافظه کمی دارد، نسبت به مقیاس مجدد مورب گرادیان ها ثابت است، و برای مسائلی که از نظر داده/پارامترها بزرگ هستند، مناسب است."

2-1- Python محیط

به طور کلی مراحل زیر در پیاده سازی این پروژه طی شده است؛

2-1-2 دیتاست mnist

در این قسمت آزمایش با استفاده از کتابخانه tensorflow و با استفاده از دیتاست mnist که شامل ۷۰۰۰۰ تصویر از اعداد ۰-۹ به صورت دست نوشت است قرار است اعداد دست نوشته شده را با استفاده از یک شبکه CNN پیش‌بینی کنیم. برای اینکار ابتدا کتابخانه ها و دیتاست را import می‌کنیم.

Programming Assignment

CNN classifier for the MNIST dataset


Type *Markdown* and *LaTeX*: α^2

```
In [6]: ##### PACKAGE IMPORTS #####

# Run this cell first to import all required packages. Do not make any imports elsewhere in the notebook

import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# If you would like to make further imports from TensorFlow, add them here
```

 MNIST overview image

The MNIST dataset

In this assignment, you will use the [MNIST dataset](#). It consists of a training set of 60,000 handwritten digits with corresponding labels, and a test set of 10,000 images. The images have been normalised and centred. The dataset is frequently used in machine learning research, and has become a standard benchmark for image classification models.

- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, 86(11):2278-2324, November 1998.

Your goal is to construct a neural network that classifies images of handwritten digits into one of 10 classes.

شکل ۳) ایمپورت کردن کتابخانه ها

Load and preprocess the data

In [7]: # Run this cell to load the MNIST data

```
mnist_data = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist_data.load_data()
```

First, preprocess the data by scaling the training and test images so their values lie in the range from 0 to 1.

In [8]: ##### GRADED CELL #####

```
# Complete the following function.
# Make sure to not change the function name or arguments.
```

```
def scale_mnist_data(train_images, test_images):
    """
    This function takes in the training and test images as loaded in the cell above, and scales them
    so that they have minimum and maximum values equal to 0 and 1 respectively.
    Your function should return a tuple (train_images, test_images) of scaled training and test images.
    """
    train_images = train_images/255
    test_images = test_images/255

    return (train_images, test_images)
```

In [9]: # Run your function on the input data

```
scaled_train_images, scaled_test_images = scale_mnist_data(train_images, test_images)
```

In [10]: # Add a dummy channel dimension

```
scaled_train_images = scaled_train_images[..., np.newaxis]
scaled_test_images = scaled_test_images[..., np.newaxis]
```

شکل ۴) لود کردن دیتاست و اسکیل کردن آن بین ۰-۱

بعد از جدا کردن دیتا ست به دو بخش `train` و `test` دیتای ترین را بین دو عدد ۰-۱ اسکیل می‌کنیم، برای اینکار نیاز است که کل دیتاست را تقسیم بر ۲۵۵ بکنیم به این علت که رنگ ها به ۰-۲۵۵ تقسیم شده اند و برای اسکیل کردن آن باید بر ۲۵۵ تقسیم کنیم.

سپس به ساختن مدل می‌پردازیم. برای بیلد کردن مدلمان با توجه به دستور کار از `sequential API` استفاده می‌کنیم که این شامل ۶ لایه (۱ لایه `conv2D` با ۸ فیلتر و سائز 3×3 با تابع فعال ساز `relu`، یک لایه `MaxPooling2D`، یک لایه `flatten` برای یک بعدی کردن داده ها و ۲ لایه `Dense` با ۶۴ نورون و تابع فعال ساز `relu` و لایه آخر با فعال ساز `softmax`) می‌شود.

بعد از تعریف مدلمان باید ان را `compile` کنیم که برای اینکار از تابع `compile` خود کتابخانه `tensorflow` استفاده می‌کنیم که در آن از اپتیمایز `adam`، `metrics = accuracy`

`loss = sparse_categorical_crossentropy` استفاده کردیم.

Build the convolutional neural network model

We are now ready to construct a model to fit to the data. Using the Sequential API, build your CNN model according to the following spec:

- The model should use the `input_shape` in the function argument to set the input size in the first layer.
- A 2D convolutional layer with a 3x3 kernel and 8 filters. Use 'SAME' zero padding and ReLU activation functions. Make sure to provide the `input_shape` keyword argument in this first layer.
- A max pooling layer, with a 2x2 window, and default strides.
- A flatten layer, which unrolls the input into a one-dimensional tensor.
- Two dense hidden layers, each with 64 units and ReLU activation functions.
- A dense output layer with 10 units and the softmax activation function.

In particular, your neural network should have six layers.

```
In [6]: ##### GRADED CELL #####

# Complete the following function.
# Make sure to not change the function name or arguments.

def get_model(input_shape):
    """
    This function should build a Sequential model according to the above specification. Ensure the
    weights are initialised by providing the input_shape argument in the first layer, given by the
    function argument.
    Your function should return the model.
    """
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(8, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
    model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2), strides=None))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dense(10, activation='softmax'))
    return model

In [7]: # Run your function to get the model

model = get_model(scaled_train_images[0].shape)
scaled_train_images.shape

Out[7]: (60000, 28, 28, 1)
```

شکل ۵) بیلد کردن مدل شبکه

Compile the model

You should now compile the model using the `compile` method. To do so, you need to specify an optimizer, a loss function and a metric to judge the performance of your model.

```
In [13]: ##### GRADED CELL #####

# Complete the following function.
# Make sure to not change the function name or arguments.

def compile_model(model):
    """
    This function takes in the model returned from your get_model function, and compiles it with an optimiser,
    loss function and metric.
    Compile the model using the Adam optimiser (with default settings), the cross-entropy loss function and
    accuracy as the only metric.
    Your function doesn't need to return anything; the model will be compiled in-place.
    """
    # model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    model.compile(optimizer = 'adam', loss = 'cross-entropy', metrics = ['accuracy'])

In [14]: # Run your function to compile the model

compile_model(model)
```

شکل ۶) کامپایل کردن شبکه

پس از آنکه شبکه با موفقیت ساخته شد نوبت به آموزش آن با استفاده از داده های آموزش است که برای اینکار از متد `fit` در `sequential API` استفاده می کنیم که آرگومان های ورودی آن علاوه بر داده های آموزش شامل `epoch ۵` و `batch_size` برابر با `۲۵۶` می باشد. پس از آموزش شبکه مشاهده می کنیم که به دقت `۹۸ درصدی` می رسیم

Fit the model to the training data

Now you should train the model on the MNIST dataset, using the model's `fit` method. Set the training to run for 5 epochs, and return the training history to be used for plotting the learning curves.

In [10]: ##### GRADED CELL #####

```
# Complete the following function.
# Make sure to not change the function name or arguments.

def train_model(model, scaled_train_images, train_labels):
    """
    This function should train the model for 5 epochs on the scaled_train_images and train_labels.
    Your function should return the training history, as returned by model.fit.
    """
    history = model.fit(x = scaled_train_images, y = train_labels, batch_size = 256, epochs = 5)
    return history
```

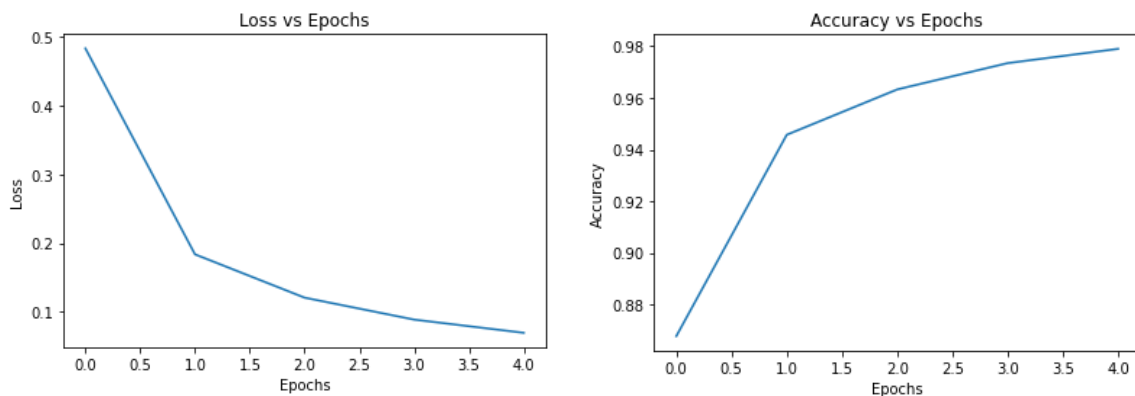
In [11]: # Run your function to train the model

```
history = train_model(model, scaled_train_images, train_labels)

Epoch 1/5
235/235 [=====] - 4s 13ms/step - loss: 0.4825 - accuracy: 0.8677
Epoch 2/5
235/235 [=====] - 3s 13ms/step - loss: 0.1837 - accuracy: 0.9457
Epoch 3/5
235/235 [=====] - 3s 13ms/step - loss: 0.1208 - accuracy: 0.9633
Epoch 4/5
235/235 [=====] - 3s 13ms/step - loss: 0.0890 - accuracy: 0.9735
Epoch 5/5
235/235 [=====] - 3s 13ms/step - loss: 0.0698 - accuracy: 0.9790
```

شکل ۷) آموزش شبکه با متد `fit`

در شکل زیر نیز میزان دقت و خطا بر حسب epoch را مشاهده می‌کنیم که شبکه به دقت ۹۸ درصدی بر روی داده‌های تست رسیده است.



شکل ۸) دقت و خطا بر حسب epoch ها

در انتها نیز با استفاده از متد `evaluate` میزان دقت شبکه آموزش داده شده بر روی داده‌های تست را امتحان می‌کنیم.

Evaluate the model

Finally, you should evaluate the performance of your model on the test set, by calling the model's `evaluate` method.

```
In [15]: ##### GRADED CELL #####

# Complete the following function.
# Make sure to not change the function name or arguments.

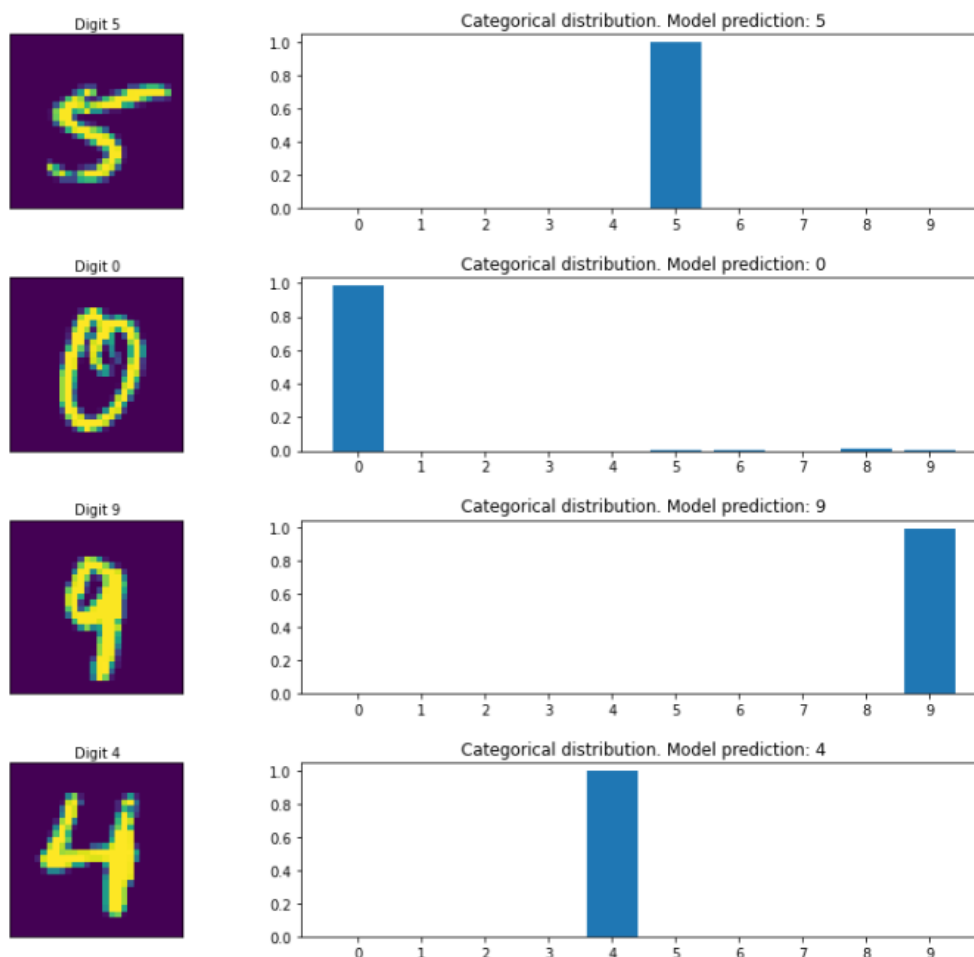
def evaluate_model(model, scaled_test_images, test_labels):
    """
    This function should evaluate the model on the scaled_test_images and test_labels.
    Your function should return a tuple (test_loss, test_accuracy).
    """
    history = model.evaluate(scaled_test_images, test_labels)
    return history

In [16]: # Run your function to evaluate the model

test_loss, test_accuracy = evaluate_model(model, scaled_test_images, test_labels)
print(f"Test loss: {test_loss}")
print(f"Test accuracy: {test_accuracy}")

313/313 [=====] - 1s 2ms/step - loss: 0.0703 - accuracy: 0.9781
Test loss: 0.07029616087675095
Test accuracy: 0.978100018119812
```

شکل ۹) تست شبکه آموزش داده شده با دیتا تست که به دقت ۹۷.۸ درصدی منجر شد.



شکل ۱۰) مشاهده مقدار خروجی داده شده.

3-1 تمرین

3-1-1 دیتاست CIFAR-10

توضیحات مربوط به دیتاست cifar-10 در پیش گزارش داده شده است پس مستقیم به توضیح کد می‌پردازیم.
در ابتدا مانند دیتاست mnist کتابخانه هارا ایمپورت می‌کنیم و دیتاست را ادد می‌کنیم.

▼ CNN classifier for the cifar10 dataset ¶

Importing libraries

```
[1]: import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Dataset preparation (Preprocessing)

```
[2]: cifar10_data = tf.keras.datasets.cifar10
(x_train, y_train), (x_test, y_test) = cifar10_data.load_data()
assert x_train.shape == (50000, 32, 32, 3)
assert x_test.shape == (10000, 32, 32, 3)
assert y_train.shape == (50000, 1)
assert y_test.shape == (10000, 1)

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [=====] - 13s 0us/step
```

Dataset scaling

```
[3]: def scale_data(train_images, test_images):
    """
    This function takes in the training and test images as loaded in the cell above, and scales them
    so that they have minimum and maximum values equal to 0 and 1 respectively.
    Your function should return a tuple (train_images, test_images) of scaled training and test images.
    """
    train_images = train_images/255.
    test_images = test_images/255.

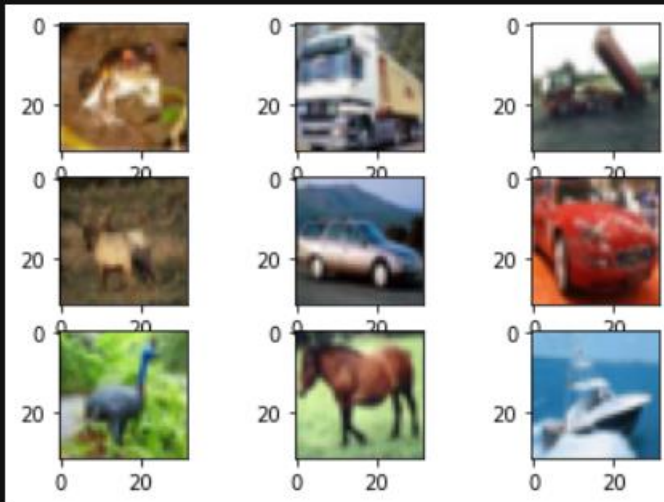
    return (train_images, test_images)

[4]: x_train_s, x_test_s = scale_data(x_train, x_test)
print(f"x_train_s shape:\n{x_train_s.shape}")
print(f"x_test_s shape:\n{x_test_s.shape}")

x_train_s shape:
(50000, 32, 32, 3)
x_test_s shape:
(10000, 32, 32, 3)
```

شکل ۱۱) ایمپورت کردن دیتاست و کتابخانه ها

```
[6]: # plotting first few images
for i in range(9):
    # define subplot
    plt.subplot(330 + 1 + i)
    # plot raw pixel data
    plt.imshow(x_train_s[i])
# show the figure
plt.show()
```



شکل ۱۲) مشاهده چند نمونه از دیتاهای cifar-10

در مرحله بعدی به بیلد کردن شبکه مشغول می‌شویم. در اینجا نیز از API sequential استفاده می‌کنیم و لایه‌های مختلف را به شبکه اضافه می‌کنیم. برای اینکه تاثیر افزایش لایه‌ها را متوجه بشویم ۳ مدل استفاده می‌کنیم که هر کدام به اندازه شمارشان VGG دارند. در مدل ۱ علاوه بر نرمالایز کردن بچ‌ها و لایه maxpooling از ۲ لایه conv2D استفاده کردیم که به تعداد ۳۲ فیلتر دارد و سایز آن 3×3 می‌باشد. در لایه مخفی اول نیز از ۱۲۸ نورون با تابع فعال ساز relu و لایه آخر نیز از تابع فعال ساز softmax استفاده کردیم. در مدل ۲ علاوه بر VGG استفاده شد مدل ۱ از ۲ لایه conv2D با ۶۴ فیلتر و تابع فعال ساز relu استفاده شده است و در مدل سوم نیز علاوه بر استفاده از مدل اول از ۲ لایه conv2D با ۱۲۸ فیلتر و با همان تابع فعال ساز relu استفاده کردیم که در ادامه تاثیر این افزایش لایه‌ها را مشاهده می‌کنیم.

Building the convolutional neural network models

Model_1 (1 VGG layer)

```
[7]: def cnn_get_model_1(input_shape):
    """
    This function should build a Sequential model according to the above specification. Ensure the
    weights are initialised by providing the input_shape argument in the first layer, given by the
    function argument.
    Your function should return the model.
    """
    model = tf.keras.Sequential()

    # VGG_1
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # Output
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(10, activation='softmax'))

    return model
```

شکل ۱۳) تابع برای مدل اول شبکه

Model_2 (2 VGG layers)

```
[8]: def cnn_get_model_2(input_shape):
    """
    This function should build a Sequential model according to the above specification. Ensure the
    weights are initialised by providing the input_shape argument in the first layer, given by the
    function argument.
    Your function should return the model.
    """
    model = tf.keras.Sequential()

    # VGG_1
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # VGG_2
    model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.3))

    # Output
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(10, activation='softmax'))

    return model
```

شکل ۱۴) تابع برای بیلد کردن شبکه دوم

Model_3 (3 VGG layers)

```
[9]: def cnn_get_model_3(input_shape):
    """
    This function should build a Sequential model according to the above specification. Ensure the
    weights are initialised by providing the input_shape argument in the first layer, given by the
    function argument.
    Your function should return the model.
    """
    model = tf.keras.Sequential()

    # VGG_1
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # VGG_2
    model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.3))

    # VGG_3
    model.add(tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=input_shape))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.4))

    # Output
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(10, activation='softmax'))
    return model
```

شکل ۱۵) تابع برای بیلد کردن شبکه سوم

```
[10]: model_1 = cnn_get_model_1(x_train_s[0].shape)
      model_2 = cnn_get_model_2(x_train_s[0].shape)
      model_3 = cnn_get_model_3(x_train_s[0].shape)
```

شکل ۱۶) مدل ۳ را با استفاده از توابعشان تعریف می‌کنیم.

در ادامه نیز می‌توانید سامری شبکه هارا در کد مشاهده کنید که در شکل پایین لایه های شبکه مدل ۲ را مشاهده می‌کنید.

```
In [12]: model_2.summary()
```

```
Model: "sequential_1"
Layer (type)                 Output Shape              Param #
-----
conv2d_2 (Conv2D)            (None, 32, 32, 32)        896
batch_normalization_3 (Batch Normalization) (None, 32, 32, 32)        128
conv2d_3 (Conv2D)            (None, 32, 32, 32)        9248
batch_normalization_4 (Batch Normalization) (None, 32, 32, 32)        128
max_pooling2d_1 (MaxPooling2D) (None, 16, 16, 32)         0
dropout_2 (Dropout)          (None, 16, 16, 32)         0
conv2d_4 (Conv2D)            (None, 16, 16, 64)       18496
batch_normalization_5 (Batch Normalization) (None, 16, 16, 64)        256
conv2d_5 (Conv2D)            (None, 16, 16, 64)       36928
batch_normalization_6 (Batch Normalization) (None, 16, 16, 64)        256
max_pooling2d_2 (MaxPooling2D) (None, 8, 8, 64)          0
dropout_3 (Dropout)          (None, 8, 8, 64)          0
flatten_1 (Flatten)          (None, 4096)               0
dense_2 (Dense)              (None, 128)                524416
batch_normalization_7 (Batch Normalization) (None, 128)                512
dropout_4 (Dropout)          (None, 128)                0
dense_3 (Dense)              (None, 10)                 1290
-----
Total params: 592,554
Trainable params: 591,914
Non-trainable params: 640
```

شکل ۱۷) شبکه تشکیل شده برای مدل ۲

بعد از تشکیل دادن شبکه هایمان نیاز به کامپایل کردن آنها داریم که برای اینکار از متد compile با اپتیمایزر adam استفاده می کنیم.

Compiling the model

```
[14]: def compile_model(model):
      """
      This function takes in the model returned from your get_model function, and compiles it with an optimiser,
      loss function and metric.
      Compile the model using the Adam optimiser (with default settings), the cross-entropy loss function and
      accuracy as the only metric.
      Your function doesn't need to return anything; the model will be compiled in-place.
      """
      model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])

[15]: compile_model(model_1)
      compile_model(model_2)
      compile_model(model_3)
```

شکل ۱۸) کامپایل کردن شبکه ها

بعد از کامپایل کردن شبکه به آموزش شبکه با استفاده از داده های آموزش می پردازیم که برای اینکار از متد fit استفاده می کنیم.

Training

```
[16]: def train_model(model, scaled_train_images, train_labels, batch_size, epochs):
    """
    This function should train the model for # epochs on the scaled_train_images and train_labels.
    Your function should return the training history, as returned by model.fit.
    """
    history = model.fit(x = scaled_train_images, y = train_labels, batch_size = batch_size, epochs = epochs)
    return history
```

Model 1 (1 VGG layer) ⓘ

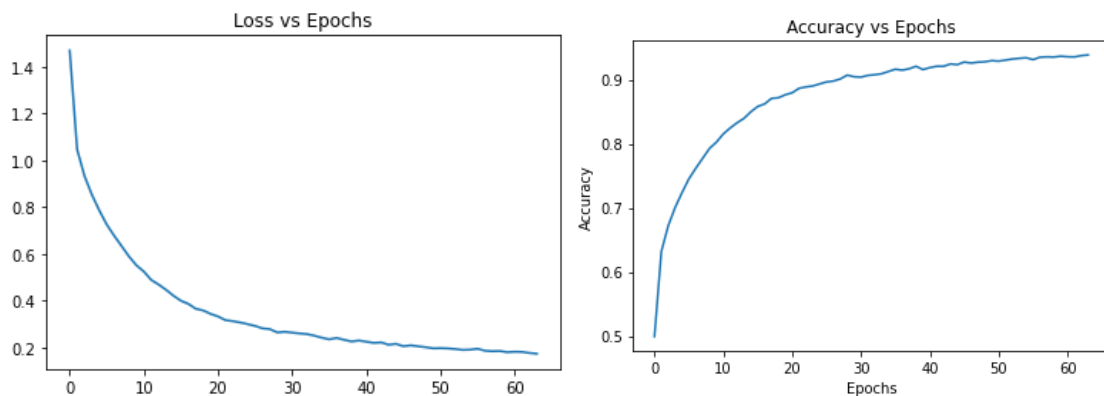
```
[17]: history_model_1 = train_model(model_1, x_train_s, y_train, batch_size=64, epochs=64)
```

شکل ۱۹) آموزش شبکه ها

برای آموزش شبکه از بچ سایز های ۶۴ تایی استفاده کردیم و به این علت که دوست عزیزمان! آقای طهرانی علاقه زیاده به دقت های بالا دارند از ۶۴ epoch استفاده کردند که ان شاء الله لپتاپشان بعد از ران گرفتن این شبکه ها سالم مانده باشد اما ما برای مقایسه دقت این ۳ مدل به همان ۱۰ epoch اکتفا می کنیم.

```
Epoch 60/64
782/782 [=====] - 4s 6ms/step - loss: 0.1786 - accuracy: 0.9367
Epoch 61/64
782/782 [=====] - 5s 6ms/step - loss: 0.1805 - accuracy: 0.9358
Epoch 62/64
782/782 [=====] - 5s 6ms/step - loss: 0.1796 - accuracy: 0.9355
Epoch 63/64
782/782 [=====] - 5s 6ms/step - loss: 0.1753 - accuracy: 0.9376
Epoch 64/64
782/782 [=====] - 4s 6ms/step - loss: 0.1721 - accuracy: 0.9387
```

شکل ۲۰) پایان آموزش شبکه ۱ با دقت ۹۳.۹ درصدی در ۶۴ epochs و دقت ۸۰.۲۵ درصدی در ۱۰ epochs



شکل ۲۱) دقت و خطا شبکه ۱ بر حسب epochs

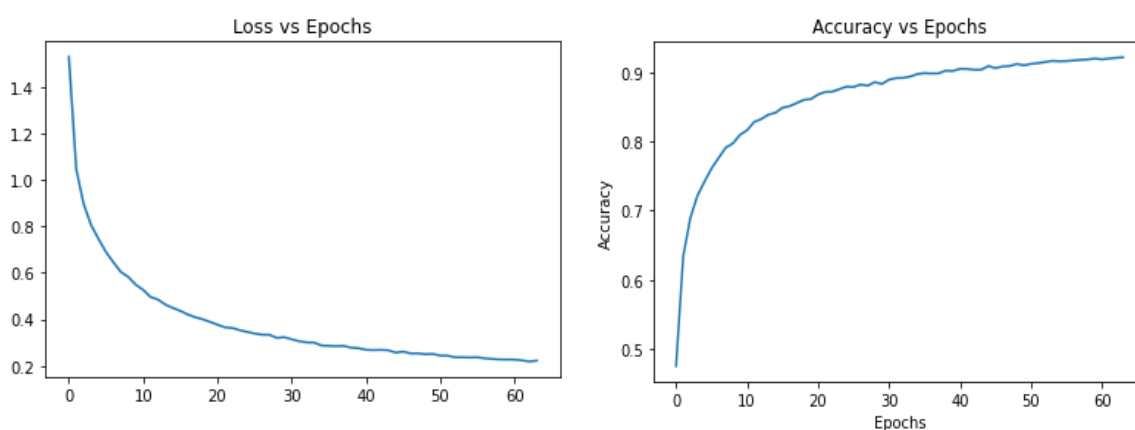
Model 2 (2 VGG layers)

```
[21]: history_model_2 = train_model(model_2, x_train_s, y_train, batch_size=64, epochs=64)
```

شکل (۲۲) آموزش شبکه ۲

```
Epoch 60/64
782/782 [=====] - 6s 8ms/step - loss: 0.2275 - accuracy: 0.9198
Epoch 61/64
782/782 [=====] - 6s 8ms/step - loss: 0.2270 - accuracy: 0.9187
Epoch 62/64
782/782 [=====] - 6s 8ms/step - loss: 0.2236 - accuracy: 0.9198
Epoch 63/64
782/782 [=====] - 6s 8ms/step - loss: 0.2191 - accuracy: 0.9208
Epoch 64/64
782/782 [=====] - 6s 8ms/step - loss: 0.2224 - accuracy: 0.9215
```

شکل (۲۳) پایان آموزش شبکه ۲ با دقت نهایی ۹۲.۱۵ درصدی و دقت ۸۰.۹۶ درصدی در ۱۰ epochs



شکل (۲۴) دقت و خطا مدل ۲ بر حسب epochs

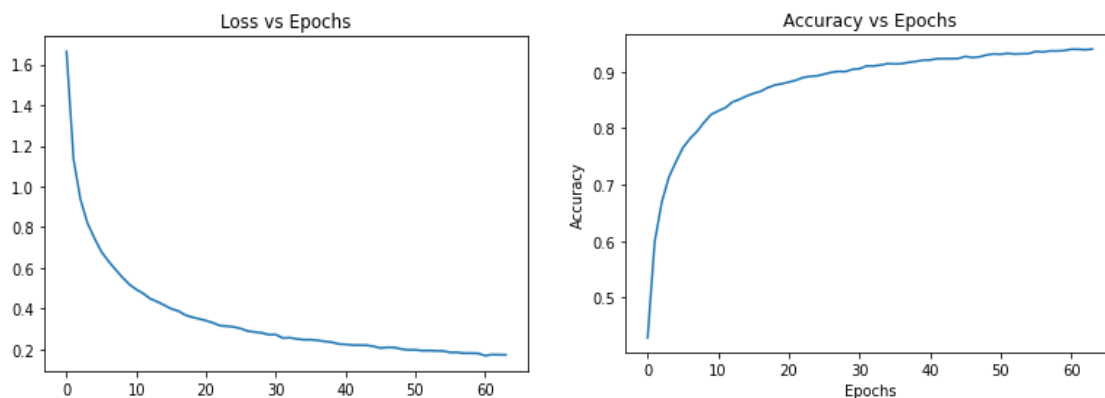
Model 3 (3 VGG layers)

```
[25]: history_model_3 = train_model(model_3, x_train_s, y_train, batch_size=64, epochs=64)
```

شکل (۲۵) آموزش شبکه ۳

```
Epoch 60/64
782/782 [=====] - 7s 9ms/step - loss: 0.1806 - accuracy: 0.9377
Epoch 61/64
782/782 [=====] - 7s 9ms/step - loss: 0.1699 - accuracy: 0.9401
Epoch 62/64
782/782 [=====] - 7s 9ms/step - loss: 0.1746 - accuracy: 0.9400
Epoch 63/64
782/782 [=====] - 7s 9ms/step - loss: 0.1738 - accuracy: 0.9393
Epoch 64/64
782/782 [=====] - 7s 9ms/step - loss: 0.1734 - accuracy: 0.9404
```

شکل (۲۶) پایان آموزش مدل ۳ با دقت ۹۴ درصدی و دقت ۸۲.۴۵ درصدی در ۱۰ epochs



شکل ۲۷) دقت و خطا مدل ۳ برحسب epochs

بعد از آموزش شبکه نوبت به تست دیتای تست بر روی هر ۳ مدل است که برای اینکار از متد `evaluate` استفاده می‌کنیم که در پایین نتیجه آن برای هر ۳ مدل را مشاهده می‌کنید.

```

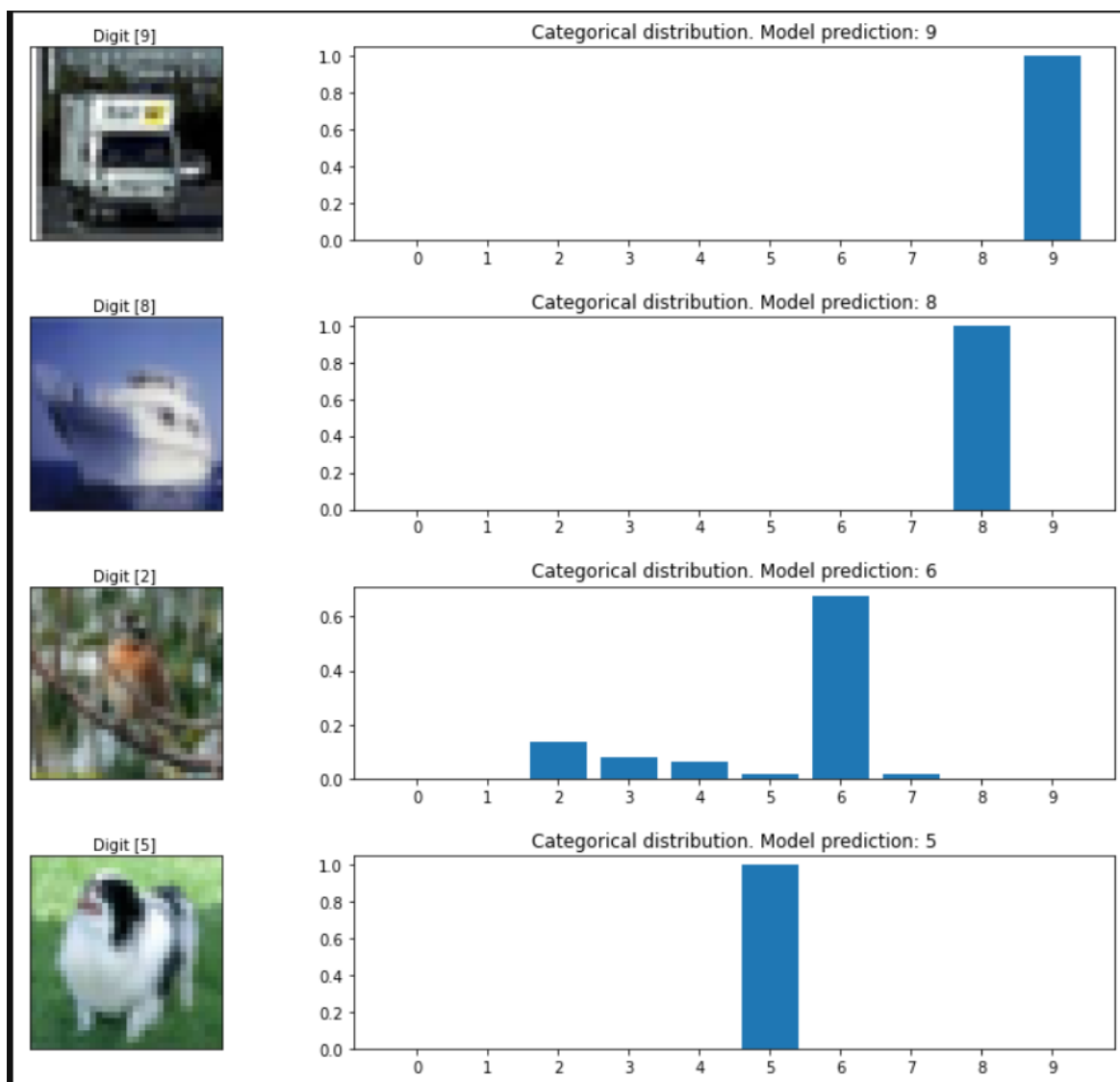
Model Evaluation
[29]: def evaluate_model(model, scaled_test_images, test_labels):
      """
      This function should evaluate the model on the scaled_test_images and test_labels.
      Your function should return a tuple (test_loss, test_accuracy).
      """
      history = model.evaluate(scaled_test_images, test_labels)
      return history

Model_1 (1 VGG layer)
[30]: test_loss, test_accuracy = evaluate_model(model_1, x_test_s, y_test)
      print(f"Test loss: {test_loss}")
      print(f"Test accuracy: {test_accuracy}")

313/313 [=====] - 1s 3ms/step - loss: 1.1251 - accuracy: 0.7252
Test loss: 1.1250661611557007
Test accuracy: 0.7251999974250793

```

شکل ۲۸) تست شبکه که به دقت ۷۲.۵ درصدی ختم شد



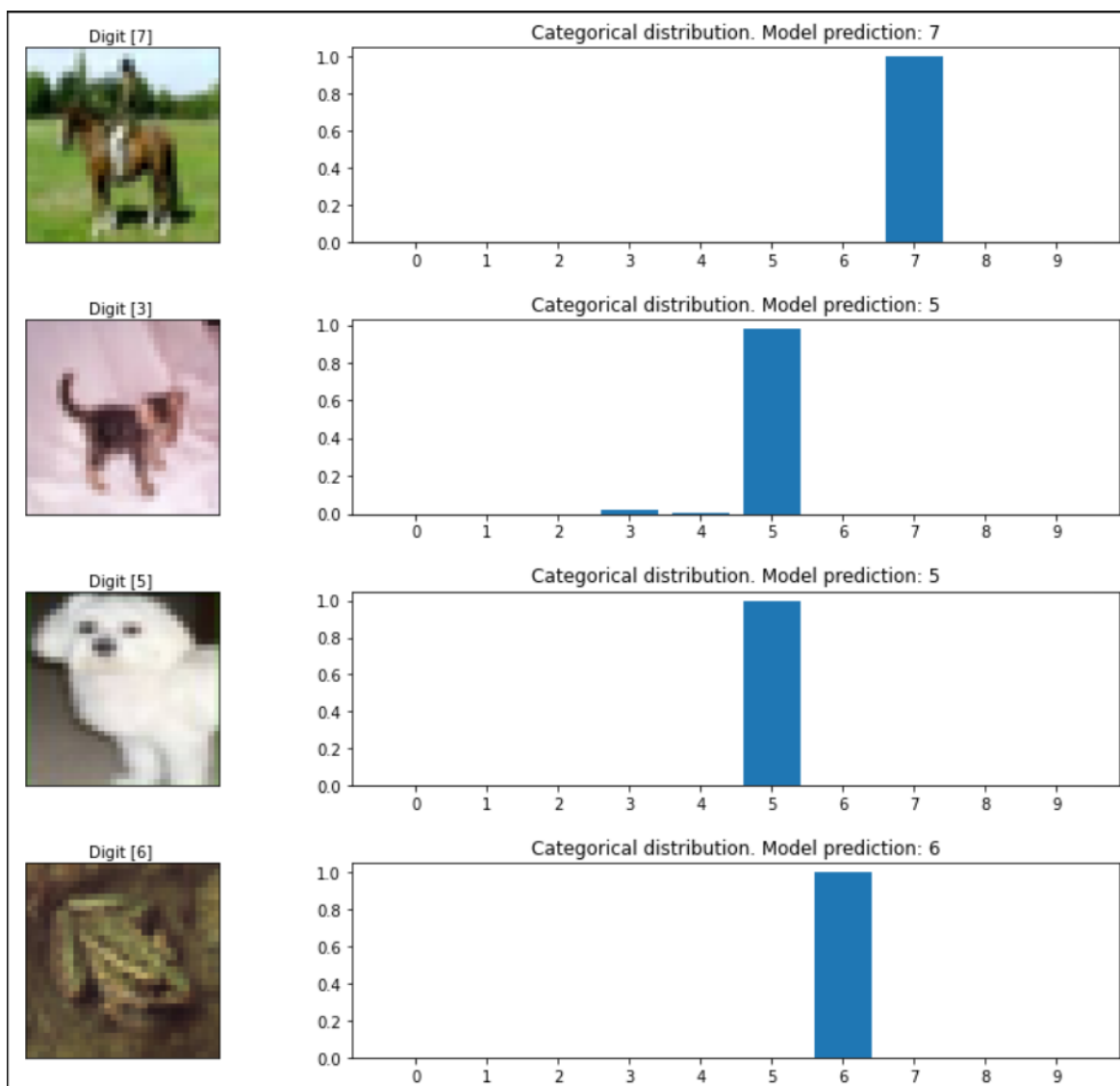
شکل ۲۹) تست خروجی شبکه ۱

Model_2 (2 VGG layers)

```
[32]: test_loss, test_accuracy = evaluate_model(model_2, x_test_s, y_test)
      print(f"Test loss: {test_loss}")
      print(f"Test accuracy: {test_accuracy}")

313/313 [=====] - 1s 3ms/step - loss: 0.6241 - accuracy: 0.8233
Test loss: 0.6240770220756531
Test accuracy: 0.8233000040054321
```

شکل ۳۰) تست شبکه ۲ بر روی دیتا تست که دقت ۸۲.۳ درصدی منجر شد



شکل (۳۱) خروجی شبکه ۲

```

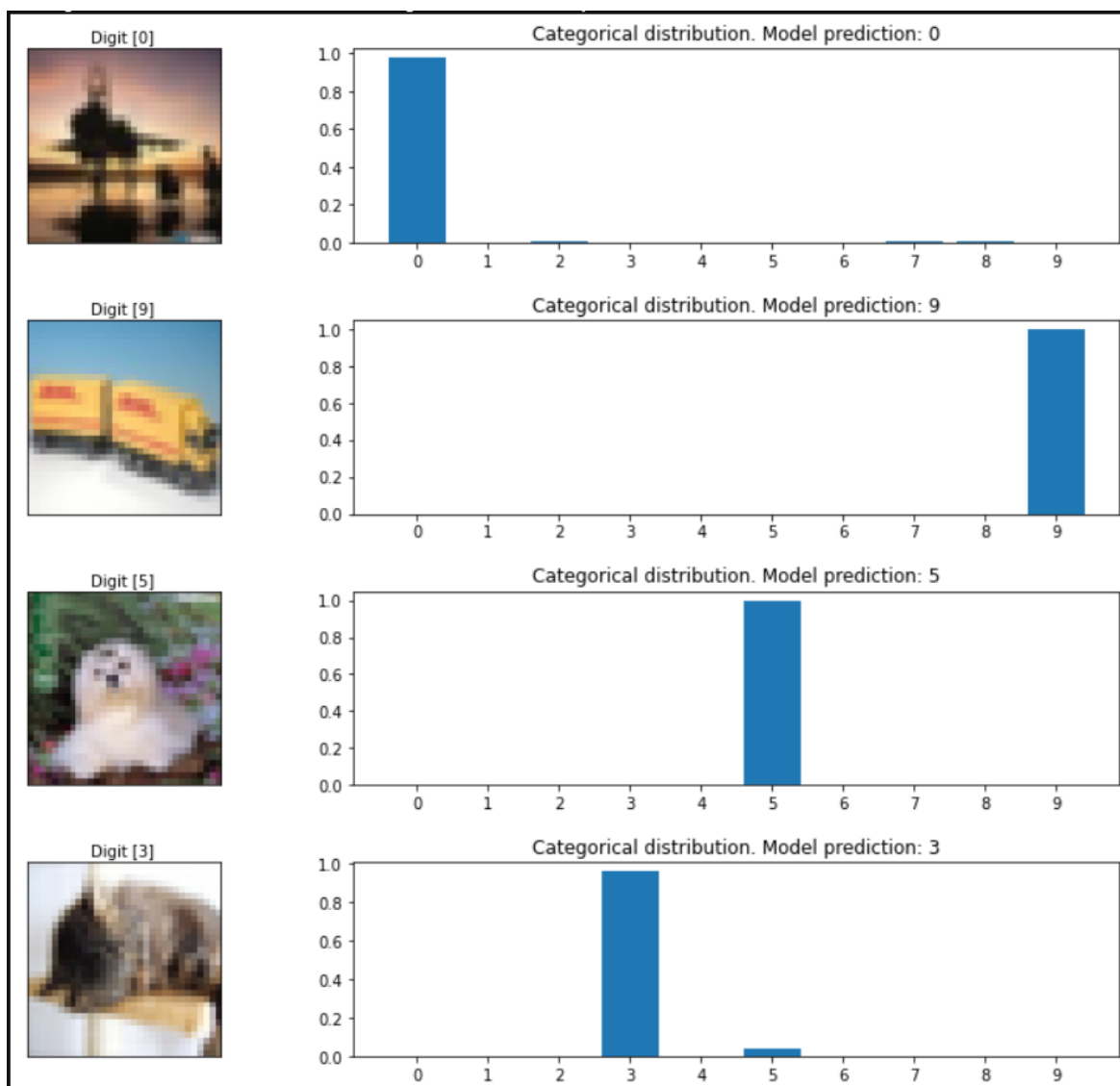
Model_3 (3 VGG layers)

[34]: test_loss, test_accuracy = evaluate_model(model_3, x_test_s, y_test)
      print(f"Test loss: {test_loss}")
      print(f"Test accuracy: {test_accuracy}")

313/313 [=====] - 1s 4ms/step - loss: 0.4617 - accuracy: 0.8688
Test loss: 0.4617026746273041
Test accuracy: 0.8687999844551086

```

شکل (۳۲) تست شبکه ۳ که دقت ۸۶.۸ درصدی منجر شد



شکل ۳۳) خروجی شبکه ۳

همانطور که مشاهده شد با افزایش تعداد لایه ها دقت شبکه ها بیشتر شد که این مقدار تفاوت در شبکه اول و سوم تقریباً برابر با ۱۵ درصد است که مقدار قابل توجهی است و با توجه به اینکه در این ۳ مدل اختلاف میان دقت در داده های آموزش و تست قابل توجه نیست پس داده آموزش بر روی شبکه overfit نکرده است پس بهتر است از همان شبکه سوم استفاده کنیم.

3-1-2 تمرین دوم

در صورتی که نیاز به چند خروجی در شبکه داشته باشیم یا قرار باشد دو شبکه عصبی با یکدیگر ارتباط داشته باشند یا لایه بعدی تعداد ورودی بیشتری از لایه قبلی داشته باشد بهتر است از functional-API به جای sequential-API استفاده بکنیم.

پایان