CLASSIFICATION WITH NEURAL NETWORKS FC VS CNN

SEQUENTIAL API VS FUNCTIONAL API MODEL IN KERAS

Siamak Farjami
dept. Computer Science
(University of Greenwich)

London, UK
sf6447b@gre.ac.uk

Abstract— **In this portfolio we want to create two different *classifier Deep Learning models, Fully Connected (FC)* and *Convolutional Neural Network (CNN)*. We will use *Sequential method* to create *FC* model and Functional *API* for the *CNN model*. This model will be used to recognise hand written digits in images. To train our models we want to use *MNIST* database. Then we want to compare them and finding out that using Convolutional Layers instead of Fully Connected in our model and changing some parameters like optimization how much can be effective for creating a better model.**

CHOOSING THE MODEL IDEA

Choosing which type of neural network, we are going to use is normally depends on database we have access to: After finding out some information about our database, then it is time to choose which type of neural network and what architecture is suitable for that database to use.

In this portfolio we are using MNIST dataset to train ours. The MNIST database of handwritten digits has been a benchmark in Computer Vision for many years. Although it is considered "solved" by many today, new algorithms are still tested on it first, and it still serves as a good learning tool. More information on MNIST: [1]

Below we can see some type Neural Network Architecture and their usages:

- Fully Connected (FC): Normally we use them for one-dimension dataset, like Signals and Vectors.
- Auto Encoder (AE): Normally we use them for one-dimension dataset, like Signals and Vectors.
- Generative adversarial networks (GAN)
- Recurrent neural network (RNN): Normally they used for datasets which depend on time, like Video, Voice, Text.
- Convolutional neural network (CNN): Normally they used for more than one dimensions datasets, like Images and Tensors.

This is just normally usage of Neural Network Architecture, it might not be right. For example, for images we can also use RNN or RNN + CNN. It depends on how much creativity we have.

Here we want to train our model using MNIST database.

As we know MNIST database is 60000 of handwritten digits 2D images(grayscale) as it is 2D we can use CNN model; CNN model input can be 2D. We can also change it to vector, which is 1D, and use fully connected model.

MNIST dataset consists of four part, train and test images and labels. So, as we have labels for our dataset we can use supervised learning models.

One of the models which we normally use for supervised learning is CNN.

We can use many different types of neural network models in MNIST dataset. Here I want to use FC and CNN models.

LOAD THE DATASET

First, we need to load the dataset. The dataset we are using is in **keras.datasets** package. We can load the data using **load_data ()** function.

As the MNIST dataset has also labels, we need to make sure to load them as well.

The code below illustrates the output.

```
In[1]:

from keras.datasets import mnist
import numpy as np

# Load data
(train_images, train_labels), (test_images,
test_labels) = mnist.load_data()
```

Looking at the data attribute will give us a good start point, and we can see if we need to apply any changes in the dataset to be compatible with the model we want to create.

- Using **ndim** command, we can see the data dimensions.
- Using **shape** command, we can see the data shape.
- Using **dtype** command, we can see the data type.
- Using **max** command, we can see the max value.
- Using **min** command, we can see the min value.

```
In[2]:

# Data attributes
print("train_images:")
print("dimensions: ", train_images.ndim)
print("shape: ", train_images.shape)
print("type: ", train_images.dtype)
print("train_labels shape: ", train_labels.shape )
print()
print("test_images:")
print("dimensions: ", test_images.ndim)
print("shape: ", test_images.shape)
print("type: ", test_images.dtype)
print("test_labels shape: ", test_labels.shape)
print()
print('Image values (min to max):',
np.min(train_images), 'to', np.max(train_images))
print('Label values (min to max):',
np.min(train_labels), 'to', np.max(train_labels))
```

```
Out[2]:

train_images:
dimensions:  3
shape:  (60000, 28, 28)
type:  uint8
train_labels shape:  (60000,)

test_images:
dimensions:  3
shape:  (10000, 28, 28)
type:  uint8
test_labels shape:  (10000,)

Image values (min to max): 0 to 255
Label values (min to max): 0 to 9
```

I. FC NEURAL NETWORK

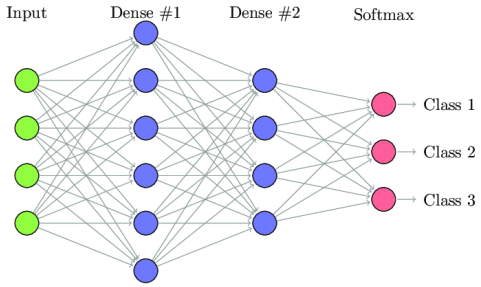

Fig. 1. *Fully Connected Neural Network*

[2]

IMPLEMENTATION STEPS

After Loading the data and finding out which model we are going to create, we need to follow the steps below for creating our model.

Way of creating the model and layers in keras has normally six steps:

A. ***Data preparation (Train/Validation/Test)***

B. ***Creating layers and model***

C. ***Setting training parameters (Loss & optimization functions ,...)***

D. ***Train the model (using fit())***

E. ***Visualization (Plot training behaviour)***

F. ***Evaluation (Evaluate the model on the test set)***

A. *Data preparation (Train/Validation/Test)*

To prepare the data for **Fully Connected model (FC)**

- First: We need to change the shape of data:
  The MNIST dataset is collection of handwriting grayscale images with 28×28 pixels dimensions. We need to change them to the vectors, as the pixel values are not suitable for learning with a FC Neural Network model inputs and needs to be vectors.
  So 28×28 images will be 28×28=784 vectors.
  Flatten the images into 1D vectors by using NumPy's **reshape()** function: [3]
- Second: The data needs to be normalised, it means all of them need to be between 0 and 1:
  For a grayscale images, the pixel value is a single number that represents the brightness of the pixel. The range of this values from 0 t0 255. Typically, zero is taken to be black, and 255 is taken to be white.
  So to normalise them we need to divide them to their max value which is 255. Before doing this, we need to make sure first to change their type to float as dividing the integers occurs error. one can do this using **astype('float32')** function.

Then, the data will be ready to input in FC model.

The code below illustrates the output.

```
# In[3]:

# Reshape
X_train1D = train_images.reshape(60000, 784) #
len(X_train1D) = 60000
X_test1D = test_images.reshape(10000, 784)

print("X_train1D shape: ", X_train1D.shape)
print("X_test1D shape: ", X_test1D.shape)
```

```
# Out[3]:

X_train1D shape:  (60000, 784)
X_test1D shape:  (10000, 784)
```

```
# In[4]:

# Normalise
X_train1D = X_train1D.astype('float32') # changing
the types
X_test1D = X_test1D.astype('float32')

X_train1D /= 255
X_test1D /= 255
```

Now we also need to change our labels and make it compatible with our trained model output.

Before changing the label if we open the train label we can see it has a number for each sample, which it shows the number of that image, but the output in our model is a vector with 10 neurons, and if the model trains good all of these 10 neurons will be closed to zero except that neuron is depends of input index, which is close to one. For example, if the input is image 5 then in output all the neurons are close to zero except 5th neuron which is close to one. So now we need to change the labels to the format of the output of our model. We need to change them to vectors with 10 elements which all the elements are equal to 0, except *nth* element which shows the number *n* is equal to 1.

This type of vectors called **One hot vector (Categorical)**, which all the elements are equal to zero except one is equal to one. For example, if the input is image 5 then 5th element of the vector should be 1 and rest 0.

For doing this we can use **np.utils** library in keras. [4]

We need to do this for both train and test labels.

The code below illustrates the output.

```
# In[5]:

import pandas as pd
pd.DataFrame(train_labels)
```

```
# Out[5]:
```

	0
0	5
1	0
2	4
3	1
4	9
...	...
59995	8
59996	3
59997	5
59998	6
59999	8

60000 rows × 1 columns

```
# In[6]:

from keras.utils import np_utils
Y_train1D = np_utils.to_categorical(train_labels)
# One hot vector (Categorical)
Y_test1D = np_utils.to_categorical(test_labels)
```

```
# In[7]:

pd.DataFrame(Y_train1D)

# Out[7]:
```

	0	1	2	3	4	5	6	7	8	9
0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...
59995	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
59996	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
59997	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
59998	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
59999	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0

60000 rows × 10 columns

B. Creating layers and model

Now it is time to create our model. We want to create a fully connected (FC) model that input is X_train1D and compare the output with Y_train1D in train time. Then we can test the result with X_test1D and Y_test1D.

So, our model will be like:

- Input Layer is 784 neurons.
- First Hidden Layer I want to have 500 neurons.
- Second Hidden Layer with 100 neurons
- Output Layer would be 10 neurons.

For creating the model in Keras we can use two different way:

- First: Using Sequential model. In this type of model, we create a **Sequential model** and then add the layer to the model respectively. This type of models is using from the first and are conventional from the first generation of keras.
- Second: Using **Functional API** in kears. In this type of model, we make our layer separately in our choice, and then we can choose the direction of the connection between the layers.
  In this type of creating the models we have more flexibility and we can also create the nonconventional models like the models has more than one input and outputs, or the models has the branches in their paths, like GoogLeNet inception model or U-Net for segmentation.

In this part we want to use simple Sequential model.

For doing this we need to import Sequential model from **keras. Model** package. Then we need to create our model before adding the layers. To do this we define a variable which is name of our model equal to Sequential ().

Here we have **FCModel = Sequential ().**

After creating the model using Sequential model we can add the layers in to the model.

Layers: We have different type of layer which keras supports most of them. For instance, some of these layers are Fully Connected, Convolutional, Recurrent and etc. We can also make our own layer and use it in our model.

We want to use simple *Fully Connected* layer in here, which is called **Dense** in keras.

We can import the layers from keras. Layers library in keras.

We can add to our model the layers using **add ()** function.

Here we have **add(Dense ())**

To create Dense Layers, we need to define the parameters needed:

First, we need to define the number of neurons we want to use in the Dense layers. We want to use **500** for the First Hidden Layer and **100** for the Second one, also the Output Layer will have **10** neurons.

Then we need to define the *activation function*. We want to use **relu** function in both Hidden Layers. For Output Layer normally **Softmax** activation is being used.
Also we need to make sure in Fully connected models (FC) we mention the Input size in creating the **First Hidden Layer**. Which here would be input **shape= (784,).**

We can also add **Dropout Layer** to our model.

Dropout layer is the layer which will be used to drop some of the neurons by random during training the model.

Dropping the neurons means make them zero or make their weights zero so they don't participate in training the model. It can prevent *over-fitting* and it is also cause of training the models independently as they don't learn from other neuron attitude which it might happen in FC models.

Deciding whether using the Dropout layer or not in our model it depends on how well our model trained and can it be optimized or is our model is over-fit or not, how many epochs we are using in our model and etc. We can find this out after the step of fitting our model and comparing it with validation set. So, I also explained briefly in that step.

Over-fitting normally happens when we don't have enough data or when the model is so complicated.

Drop out layer normally used in FC model to reduce the percentage of over-fitting and we can instead train the model with more epochs.

Using Drop out layer will also increase the loss, but sometime this tiny increase in loss and error is worthy to have no over-fitting.

We can see the summary of the model information using the **summary ()** function.

The code below illustrates the output.

```
# In[8]:

# Creating our model
# Load Library
from keras.models import Sequential
from keras.layers import Dense, Dropout


# In[9]:

# creating the model
FCModel = Sequential()
FCModel.add(Dense(500, activation='relu',
input_shape=(784,))) # Hidden Layer 1
FCModel.add(Dropout(20)) # Dropout Layer
FCModel.add(Dense(100, activation='relu')) #
Hidden Layer 2
FCModel.add(Dropout(20)) # Dropout Layer
FCModel.add(Dense(10, activation='softmax')) #
Output Layer


# In[10]:

FCModel.summary()


# Out[10]:

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param
=================================================================
dense_1 (Dense)              (None, 500)               39250
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 100)               50100
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1010
=================================================================
Total params: 443,610
Trainable params: 443,610
Non-trainable params: 0
_________________________________________________________________
```

C. *Setting training parameters (Loss & optimization functions ,...)*

After creating the model, we need to compile the model: In this step we need to define the other parameters such as **Loss Function** and **Optimization** for the model created. We can do this using **compile()** function, which here will be **FCModel.compile().**

We use ***Loss** Function* for computing the error that our model may have, then we use *Optimization* algorithm for optimizing the model by modifying the weights towards Global minimum that error and making it as minimum as possible.

We have different types of *Loss Function* such as **Mean squared, cross- entropy (categorical and binary), Hinge and Softmax** that we can use. Each of these Loss Function use different algorithm for computing the error.

Optimization has also different algorithm that we can use such as **SGD (Stochastic Gradient Descent) which is the most famous one, Momentum, Adam and etc.**

Here we define *SGD* as the optimizer of the model and *categorical_crossentropy* for the loss function in the ***compile()*** function.

We also define *metrics* to accuracy(acc). By doing this except the error, it also shows us the accuracy of the model during training. So, error should go toward zero and accuracy need to go to one or %100.

The code below illustrates the output.

```
# In[11]:

# Load Library
from keras.optimizers import SGD  # We can also
define like 'SGD' without importing
from keras.losses import categorical_crossentropy
```

```
# In[12]:

#compile the model
FCModel.compile(optimizer=SGD(lr=0.001),
loss=categorical_crossentropy, metrics=['acc'])
# Lr: learning rate
```

D. Train the model (using fit())

In this step we want to train our created model.

For training the model we use **fit()** function. In here would be **FCModel.fit().**

fit() function has also the parameters which need to be defined.

We need to define *input, output, batch size, epoch, ...* *Input* will be our *X_train1D* which is 60000 of 784 elements vectors, and *output* would be *Y_train1D* which is 10 elements one hot vector.

we define *batch size* to 128 here for now as we have quite a lot data. Normally, as much as the batch size is more our model will have better accuracy.

epochs are the number of times that that model will see or use all the data to train itself. So obviously with more epochs our model will train more, which it doesn't mean our model would be a good model necessarily.

Too much training the model can cause some problems like **overfitting** which it means that your model does not learn the data, it memorizes the data.

Using more than enough epochs can also be computational expensive. So, using enough epochs is important.

We add epochs until we see the minimum loss function, error, or it goes until we don't see significant reduction in loss function.

Finding the suitable epochs number to use to train the model is completely empirical and according to trial and error.

Another thing we can do in this part is to split our train dataset to two part of train and validation, which we can evaluate the model accuracy by validation data and change the parameters as long as we optimize the model. We can do this with different ways.

The data might already split from the first to train set, test set and validation set, which we can evaluate the validation set using **evaluate()**function after training the train set by using **fit().**

Another way and better way to do is when training the model using **fit()** function we can also define the parameter **validation_split** by giving the percentage of validation set we need to split from train set. Here the *validation_split = 0.2*.

If we already have validation set and we don't need to take from the train set we need to define **validation_Data** instead of **validation_split**, and we will give the data to **validation_Data**.

We can place the brief or history of our trained model in a variable by assigning to that variable.

In here **network_history** is the history of our trained model in 20 epochs. So, we can analyse the model in the next steps.

To see how much it takes to train our model we can use **datetime()** function in *datetime* library. We need to set the start and end before and after fitting our model and subtracting end time from start time will show us the time of the train.

The code below illustrates the output.

```
# In[13]:

# Train our model
import datetime
start = datetime.datetime.now()

network_historyFC = FCModel.fit(X_train1D,
Y_train1D, batch_size=128, epochs=20,
validation_split=0.2)

end = datetime.datetime.now()
elapsed = end - start
print('Total training time: ', str(elapsed))
```

```
# In[13]:

Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 5s 110us/step -
 loss: 2.1978 - acc: 0.2771 - val_loss: 2.0440 - val_acc: 0.45
10
Epoch 2/20
48000/48000 [==============================] - 4s 94us/step -
loss: 1.9166 - acc: 0.5434 - val_loss: 1.7627 - val_acc: 0.634
7
Epoch 3/20
48000/48000 [==============================] - 6s 127us/step -
 loss: 1.6390 - acc: 0.6735 - val_loss: 1.4782 - val_acc: 0.73
23
Epoch 4/20
48000/48000 [==============================] - 6s 131us/step -
 loss: 1.3733 - acc: 0.7412 - val_loss: 1.2234 - val_acc: 0.78
24
Epoch 5/20
48000/48000 [==============================] - 6s 125us/step -
 loss: 1.1512 - acc: 0.7791 - val_loss: 1.0241 - val_acc: 0.81
34
Epoch 6/20
48000/48000 [==============================] - 7s 150us/step -
 loss: 0.9840 - acc: 0.8015 - val_loss: 0.8790 - val_acc: 0.82
94
Epoch 7/20
48000/48000 [==============================] - 6s 126us/step -
 loss: 0.8625 - acc: 0.8180 - val_loss: 0.7739 - val_acc: 0.84
23
Epoch 8/20
48000/48000 [==============================] - 6s 128us/step -
 loss: 0.7727 - acc: 0.8316 - val_loss: 0.6961 - val_acc: 0.85
15
Epoch 9/20
48000/48000 [==============================] - 5s 102us/step -
 loss: 0.7048 - acc: 0.8411 - val_loss: 0.6366 - val_acc: 0.85
93
Epoch 10/20
48000/48000 [==============================] - 5s 101us/step -
 loss: 0.6519 - acc: 0.8484 - val_loss: 0.5903 - val_acc: 0.86
58
Epoch 11/20
48000/48000 [==============================] - 6s 119us/step -
 loss: 0.6098 - acc: 0.8554 - val_loss: 0.5530 - val_acc: 0.87
17
Epoch 12/20
```

```
48000/48000 [==============================] - 6s 134us/step -
 loss: 0.5754 - acc: 0.8613 - val_loss: 0.5230 - val_acc: 0.87
55
Epoch 13/20
48000/48000 [==============================] - 6s 132us/step -
 loss: 0.5468 - acc: 0.8657 - val_loss: 0.4974 - val_acc: 0.87
96
Epoch 14/20
48000/48000 [==============================] - 5s 101us/step -
 loss: 0.5226 - acc: 0.8702 - val_loss: 0.4760 - val_acc: 0.88
38
Epoch 15/20
48000/48000 [==============================] - 6s 116us/step -
 loss: 0.5018 - acc: 0.8739 - val_loss: 0.4578 - val_acc: 0.88
78
Epoch 16/20
48000/48000 [==============================] - 5s 99us/step -
loss: 0.4839 - acc: 0.8778 - val_loss: 0.4421 - val_acc: 0.890
3
Epoch 17/20
48000/48000 [==============================] - 5s 98us/step -
loss: 0.4681 - acc: 0.8803 - val_loss: 0.4282 - val_acc: 0.892
5
Epoch 18/20
48000/48000 [==============================] - 7s 138us/step -
 loss: 0.4542 - acc: 0.8827 - val_loss: 0.4160 - val_acc: 0.89
50
Epoch 19/20
48000/48000 [==============================] - 8s 157us/step -
 loss: 0.4418 - acc: 0.8847 - val_loss: 0.4053 - val_acc: 0.89
67
Epoch 20/20
48000/48000 [==============================] - 6s 128us/step -
 loss: 0.4307 - acc: 0.8869 - val_loss: 0.3956 - val_acc: 0.89
88
Total training time:  0:01:56.290826
```

Comments

As we can see in the top table, not only we have loss(loss) and accuracy(acc) for the train set in each epoch, we also have loss(val_loss) and accuracy(val_acc) for the validation set in each epoch. It means in each epoch the data is will train with train set and they have their own loss and accuracy, and then this trained model which is split from validation set by itself will use the validation set to test the accuracy and loss of the validation set.

TOTAL TRAINING TIME FOR FC MODEL

0:01:56.290826 for 20 epochs with 128 batch size

E. Visualization (Plot training behaviour)

In this step we want to analyse the model to see how the model trained, also check the accuracy and loss function, error, of the model by visualization.

We can do this by using our *network_history* variable defined in previous step and is the history of our trained model in 20 epochs.

network_history is a keras object and has a dictionary inside of it which is accessible by .**history** command.

We also create the **plot_history ()** for easier visualisation in our next model.

The code below illustrates the output.

```
# In[14]:

print(type(network_historyFC))
historyFC = network_historyFC.history
print(type(historyFC))
print(historyFC.keys())
```

```
# Out[14]:

<class 'keras.callbacks.callbacks.History'>
<class 'dict'>
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```

```
# In[15]:

# plot
def plot_history(net_history):
    history = net_history.history
    import matplotlib.pyplot as plt
    losses = history['loss']
    val_losses = history['val_loss']
    accuracies = history['acc']
    val_accuracies = history['val_acc']

    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.plot(losses)
    plt.plot(val_losses)
    plt.legend(['loss', 'val_loss'])

    plt.figure() # creating new figure for second
plot
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(accuracies)
    plt.plot(val_accuracies)
    plt.legend(['acc', 'val_acc'])

plot_history(network_historyFC)
```

```
# Out[15]:
```

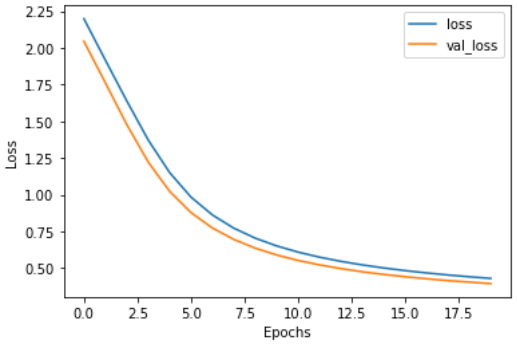

Fig. 2. *FC Model Loss and Loss Validation*

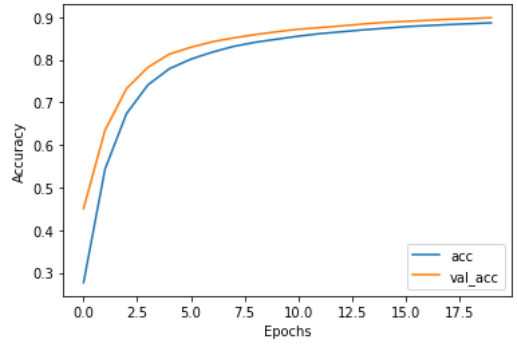

Fig. 3. *FC Model Accuracy and Accuracy Validation*

Comments

As we can see in the top plots, the accuracy and loss for train set and validation set are moving according to each other's and in a similar way, it means the model trained well until this epoch, 20 in here, and we can add the epochs.

In the Loss plot when the train set plot starts to reduce but **val_loss** shows the different attitude, that point or intersection is possibly shows that our model is **over-fit**. To solve the problem, we need to stop the epochs in that part or using the technique will prevent overfitting, like using Dropout layer. I added also Dropout layer to my model just for sake of my portfolio however this model with 20 epochs was fine even without **Dropout layer**. after adding Drop out layer to the model we can change the epochs to more than 20 until we see the better accuracy and less loss.

F. Evaluation (Evaluate the model on the test set)

In this step we want to use the trained model with the test set which our model hasn't seen this data yet. We are doing this to evaluate our model. We can do this using the **evaluate()** and **predict()** function. In here will me ***FCModel.evaluate()*** and ***FCModel.predict*().**

In **evaluate()** function we need to define input and output. Input is the test set(X_test1D) and the output is the test label(Y_test1D).

evaluate() function will test set to evaluate the trained model and in output will give us the loss and accuracy of the test data. So, we assign it into two variable ***test_loss*** and ***test_acc*.** which *mean of the loss* will place in the **test_loss** and *mean of the accuracy* in **test_acc**.

Using **predict()** function we can give to our trained model the test set (X_test1D) and see the prediction result for the model and how much our trained model is accurate.

The code below illustrates the output.

```
# In[16]:

# Evaluation
test_loss, test_acc = FCModel.evaluate(X_test1D,
Y_test1D)
test_labels_p = FCModel.predict(X_test1D)
```

```
# Out[16]:

10000/10000 [==============================] - 1s
78us/step
```

```
# In[17]:

print("test_loss: ", test_loss)
print("test_acc: ", test_acc)
```

```
# Out[17]:

test_loss:  0.40140881116390226
test_acc:  0.8966000080108643
```

FC MODEL ACCURACY AND LOSS

Total training time: *0:01:56.290826 for 20 epochs with 128 batch size*
test_loss: *0.40140881116390226*
test_acc*: 0.8966000080108643*

```
# In[18]:

pd.DataFrame(test_labels_p)
```

Out[18]:

	0	1	2	3	4	5	6	7	8	9
0	0.001265	2.27445 1e-05	0.00149 6	0.00267 5	1.49022 2e-03	0.00218 9	0.00006 7	9.68214 3e-01	0.00211 8	0.00204 62
1	0.03557 2	6.14200 9e-03	0.54460 6	0.03913 4	1.14244 2e-04	0.04458 7	0.29839 2	2.15226 5e-04	0.00315 2	0.00008 5
2	0.001788	9.13333 5e-01	0.00173	0.00143 4	5.01606 0e-03	0.00874 2	0.01273 1	5.06865 0e-03	0.00184 58	0.00034 15
3	0.990868	6.63117 2e-07	0.00004 33	0.00002 06	5.22130 5e-06	0.00564 5	0.00207 2	3.51381 7e-04	0.00003 63	0.00000 57
4	0.000489 1	4.38365 6e-04	0.00268 42	0.00034 33	7.31256 4e-01	0.00501 3	0.01374 8	4.65942 8e-02	0.00183 87	0.14939 95
...	...	...	...	...	...	...	...	...	...	...
9995	0.000001 8	1.12031 9e-05	0.99698 5	0.00017 67	1.22670 6e-06	0.00001 6	0.00007 7	5.12194 6e-05	0.00107 0	0.00000 4
9996	0.000041 9	6.10315 9e-06	0.00204 9	0.99327 4	7.31721 1e-07	0.00342 4	0.00002 4	3.34262 9e-04	0.00043 3	0.00003 6
9997	0.000023	5.04561 9e-04	0.00000 42 9	0.00004 25	7.93525 9e-01	0.00053 96	0.00048 7	9.95537 0e-03	0.00234 24	0.16583 0
9998	0.001963 4	1.06371 4e-01	0.00271 23	0.00176 22	3.95084 6e-02	0.33921	0.04388 2	3.55868 7e-02	0.36956	0.00639 5
9999	0.000216 2	4.34504 5e-06	0.00029 79	0.00000 3	3.64673 9e-05	0.00016 4	0.99459 4	3.09454 9e-07	0.00000 53	0.00000 3

10000 rows × 10 columns

Comments

As we can see **above,This test_labels_p** is the prediction number of test set with our FC model. It is a 10×1000010×10000 matrix which each output has 10 elements and the maximum of each row is the the number of that input. Now if we want to change this to the vector which has only one number for each data and that number is the index of the maximum of that 10 elements, we can use **NumPy's** library and **argmax()** function.

We need to also define axis, here we need to define axis equal to one which is rows.

The code below illustrates the output.

```
# In[19]:

import numpy as np
test_labels_p = np.argmax(test_labels_p, axis=1)
```

```
# In[20]:

pd.DataFrame(test_labels_p)
```

```
# Out[20]:
```

	0
0	7
1	2
2	1
3	0
4	4
...	...
9995	2
9996	3
9997	4
9998	8
9999	6

```
10000 rows × 1 columns
```

ACCESS TO LAYERS

After making the model we can also access to the layer separately and modify them. We can do this using layer parameter we can do this. We can change the name of the layers by adding the name parameter and assign it to another name if we want. We can also stop the layers to be trainable by assigning trainable parameter to False.

The code below illustrates the output.

```
# In[21]:

# Change layers config
FCModel.layers[0].name = 'Layer_0'
FCModel.layers[0].trainable = False
FCModel.layers[0].get_config()
```

```
# Out[21]:

{'name': 'Layer_0',
 'trainable': False,
 'batch_input_shape': (None, 784),
 'dtype': 'float32',
 'units': 500,
 'activation': 'relu',
 'use_bias': True,
 'kernel_initializer': {'class_name': 'VarianceSca
ling',
  'config': {'scale': 1.0,
   'mode': 'fan_avg',
   'distribution': 'uniform',
   'seed': None}},
 'bias_initializer': {'class_name': 'Zeros', 'conf
ig': {}},
 'kernel_regularizer': None,
 'bias_regularizer': None,
 'activity_regularizer': None,
 'kernel_constraint': None,
 'bias_constraint': None}
```

II. CNN NEURAL NETWORK

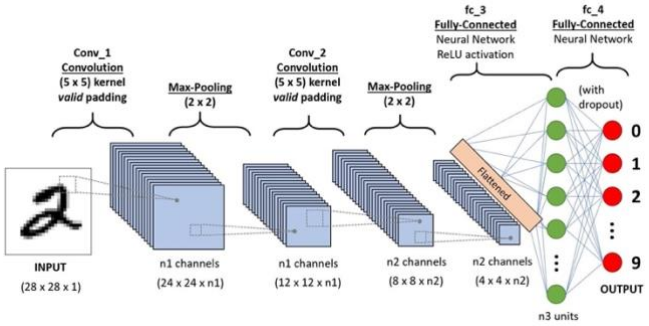

Fig. 4. *Convolutional Neural Network*

[5]

In previous part we have seen that how we can create a *Fully connected FC* model in **sequential manner**. Now I want to create the model using **functional API** and use **Convolutional** neural network **CNN** for the same database MNIST.

As we know after Loading the data and finding out which model we are going to create, we need to follow the steps below for creating our model.

Way of creating the model and layers in keras has normally six steps:

A. ***Data preparation (Train/Validation/Test)***

B. ***Creating layers and model***

C. ***Setting training parameters (Loss & optimization functions ,...)***

D. ***Train the model (using fit())***

E. ***Visualization (Plot training behaviour)***

F. ***Evaluation (Evaluate the model on the test set)***

A. *Data preparation (Train/Validation/Test)*

To prepare the data for **Convolutional neural network (CNN)**:

The difference in here with FC model is that we do not need to change the data to vector anymore and data need to be the images as it is in MNIST dataset. We also have one more dimension in image datasets which is the channel that we need to mention in CNN models, A grayscale image has just one channel, so we need to add 1 in here. Rest would be the same as FC model Date preparation.

The code below illustrates the output.

```
# In[22]:

from keras.datasets import mnist

# Load data
(train_images, train_labels), (test_images,
test_labels) = mnist.load_data()
```

```
# Data attributes
print("train_images dimentions: ",
train_images.ndim)
print("train_images shape: ", train_images.shape)
print("train_images type: ", train_images.dtype)

X_train2D = train_images.reshape(60000, 28, 28, 1)
X_test2D = test_images.reshape(10000, 28, 28, 1)

X_train2D = X_train2D.astype('float32')
X_test2D = X_test2D.astype('float32')

X_train2D /= 255
X_test2D /= 255

from keras.utils import np_utils
Y_train1D = np_utils.to_categorical(train_labels)
Y_test1D = np_utils.to_categorical(test_labels)



# Out[22]:

train_images dimentions:  3
train_images shape:  (60000, 28, 28)
train_images type:  uint8
```

B. Creating layers and model "Fig. 5".

Now we want to create the layers.
Before creating the layers we need to import the layers. Here as we are creating the **CNN** model the layers we need to import would be *conv2D* and ***Max pooling*** and ***Input*** or alternatively we can only import layer from keras package and add the layer when we are creating the layer. This also make our code nicer, easier to choose and more readable.

The difference between *FC* model with sequential method which we have created earlier and *CNN* model with *API* method:

- In ***FC*** model with ***sequential*** method we need to make the model first like *FCModel = Sequential()* and then creating the layers, like we did earlier.
- In ***CNN*** model using ***API*** model first we need to create the layers and then crate the model. So, we can choose which layers we want them to be connected. For this reason, we have more flexibility and we can create our own model and use our creativity for creating the neural networks models. We need to make sure to mention the input layer in each layer after creating them in API model.

1) Creating the Conv2D layers

By using **layer.Conv2D()** we can create the convolutional layers for our models. We also need to define the parameters. Below we can see some of the parameters using in Conv2D layers:

- Filter: The number of filters we want to use and we should have in the output.
- Windows size: The size of the window we want to use for convolutional process.
- Activation: we need to define the activation function that we want to use.
- Padding: It means the output of that layer after convolving. If we want to padding "Fig. 10", the input data and then do the convolutional process which we will not have any data reduction and the output will be the same size of input we need to define padding parameter to *'same'*. If we don't want padding, the output will lose some data and will not have some of the margins, we can define the padding to *'valid'*.

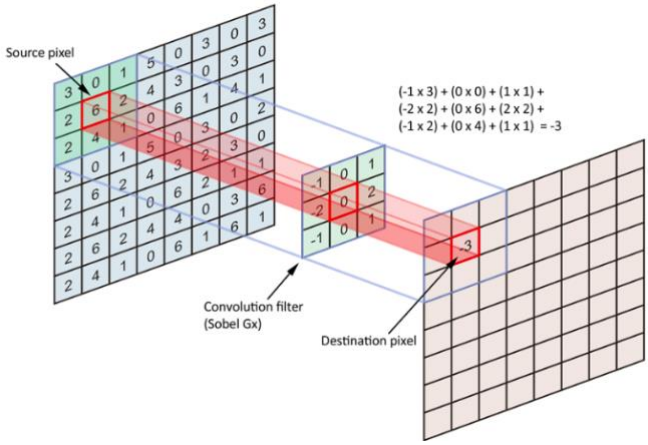

Fig. 5. *Convolutional Layer*

[6]

2) Creating the MaxPool2D layers

We use **Pooling(Max, Average, ..)** layer in order to reduction in data when training our model. We did the same thing for our FC model earlier by adding **Dropout layer**. but different process.

By using **layer.MaxPool2D()** we can create the MaxPool2D layers for our model "Fig. 6". We also need to define the **pool_size** . for example

layers.Maxpool2D(pool_size = 2)(Input layer name)

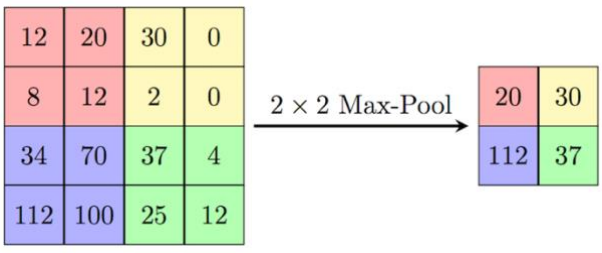

Fig. 6. *Max-Pool*

[7]

After creating the layers and choosing their input:
As we mentioned before we want to create the classifier model, so the output of the model needs to be the vector of 10 neuron to classify the numbers between 0 to 9. In CNN

model the output is also an image, so we need to change it to vector with 10 neurons.

To do this we can use **Flatten()** function.

Flatten() function makes whatever dimension input is to vector. We just need to mention the input layer.

After flattening the output of last CNN layer to a vector, we also need to input **Dense(Fully connected)** layer with 10 neuron in here for classification. We can doing this using the **layers.Dense()** as we explained earlier

Now it's time to create the models from the layers by using the **Model()** function in **kears.models** library. The parameters will be Input layer and output layer.
We can choose the input and output parameters in **Model()** function whatever we want. Here we define the input to **myInput** and output to **out_layer**. It could also be different, for example it could **myInput** be the input and conv2 the **output**, we just need to make sure our input layer meets the criteria.

So, we have a quite a lot flexibility to choose our layer, for example we could make the layer that all their inputs are the same, like the Inception Module in **GoogleNet** "Fig. 7", and "Fig. 8", Or we could even have many inputs or many outputs and create the different range of models with it.

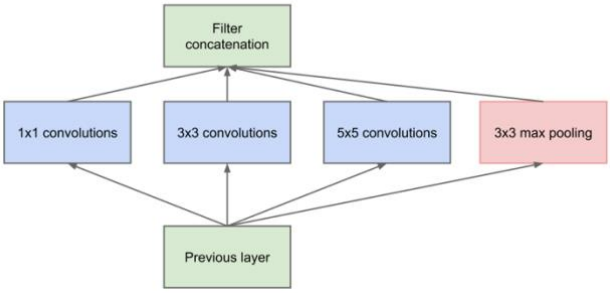

Fig. 7. *Inception Module, naive version*

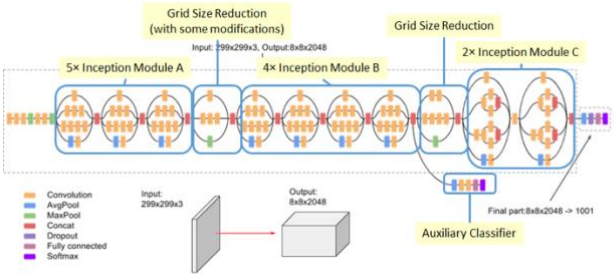

Fig. 8. *Inception Module*

[8]

The code below illustrates the output.

```
# In[23]:


# Creating our model
from keras.models import Model
from keras import layers
import keras

myInput = layers.Input(shape=(28,28,1))
conv1 = layers.Conv2D(16, 3, activation='relu',
padding='same')(myInput)
pool1 = layers.MaxPool2D(pool_size = 2)(conv1)
conv2 = layers.Conv2D(32, 3, activation='relu',
padding='same')(pool1)
pool2 = layers.MaxPool2D(pool_size = 2)(conv2)
flat = layers.Flatten()(pool2)
out_layer = layers.Dense(10,
activation='softmax')(flat)


CNNModel = Model(myInput, out_layer)

CNNModel.summary()


# Out[23]:

Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 32)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1568)              0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                15690
=================================================================
Total params: 20,490
Trainable params: 20,490
Non-trainable params: 0
_________________________________________________________________
```

Comments

As you can see above, the number of parameters is less than FC model, in fully connected model we had **443,610** parameters and here in CNN model we have only **20,490**.

Now we want to use one of the techniques using in deep learning and convolutional.

The first technique we want to implement is using stride instead of Max pooling layers. We can do this by defining the stride parameter in convolutional layer.

Conv2D layer except filter, window, activation and padding which we defined earlier has another parameter called strides. We can also define **stride** in the *convolutional layers instead of* adding pooling layer to reduce the data and the volume of computing in our model.

The code below illustrates the output.

```
# In[24]:


# Creating our model
from keras.models import Model
from keras import layers
import keras

myInput = layers.Input(shape=(28,28,1))
conv1 = layers.Conv2D(16, 3, activation='relu',
padding='same', strides=2)(myInput)
conv2 = layers.Conv2D(32, 3, activation='relu',
padding='same', strides=2)(conv1)
flat = layers.Flatten()(conv2)
out_layer = layers.Dense(10,
activation='softmax')(flat)


CNNModel = Model(myInput, out_layer)

CNNModel.summary()


# Out[24]:


Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape
 Param #
=================================================================
input_2 (InputLayer)         (None, 28, 28, 1)
 0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 16)
 160
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 7, 7, 32)
 4640
_________________________________________________________________
flatten_2 (Flatten)          (None, 1568)
 0
_________________________________________________________________
dense_5 (Dense)              (None, 10)
 15690
=================================================================
Total params: 20,490
Trainable params: 20,490
Non-trainable params: 0
_________________________________________________________________
```

Comments

As we can see in the table above, using ***strides = 2*** "Fig. 9", instead of Max pooling layers result in the same parameters, 20,490, by using less layer in the model. By doing this our model will be lighter, faster and less computing will be done. Now if we want we can even use more convolutional layer in our model.

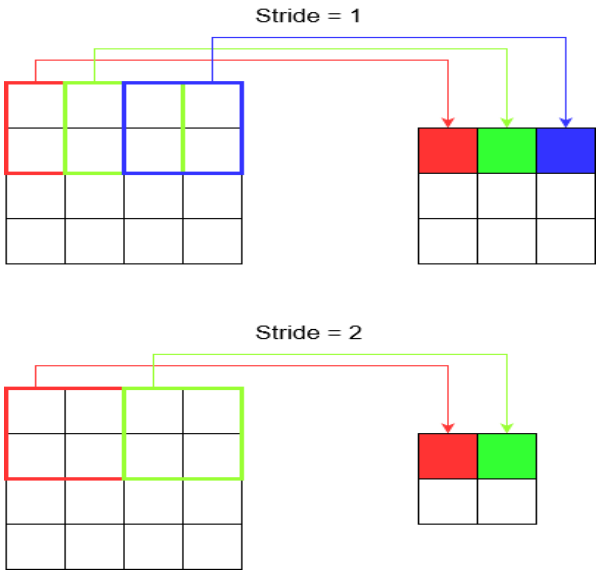

Fig. 9. *Stride 1 and 2*

[9]

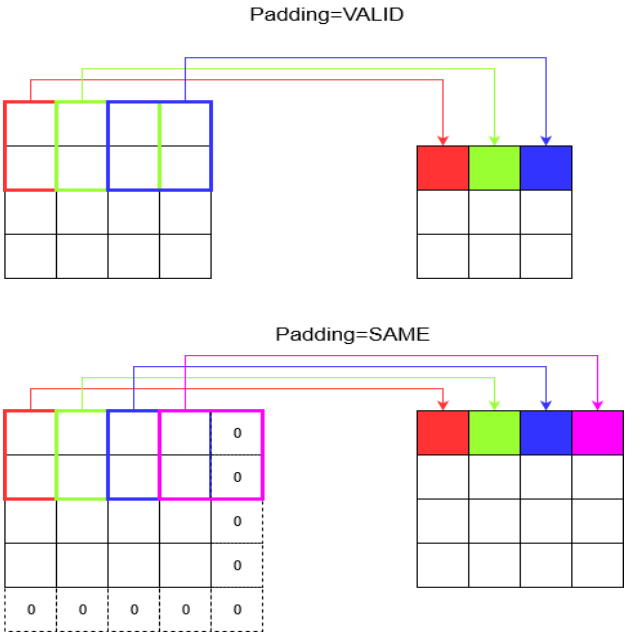

Fig. 10. *Padding Valid and Same differences*

[10]

C. Setting training parameters (Loss & optimization functions ,...)

The only difference in this step with our FC model is that we change our optimizer to **Adam()** instead of using **SGD().**

The code below illustrates the output.

```
# In[25]:

CNNModel.compile(optimizer=keras.optimizers.Adam()
, loss=keras.losses.categorical_crossentropy,
metrics=['acc'])
```

D. Train the model (using fit())

The code below illustrates the output.

```
# In[26]:

# Train our model
import datetime
start = datetime.datetime.now()

network_historyCNN = CNNModel.fit(X_train2D,
Y_train1D, batch_size=128, epochs=20,
validation_split=0.2)

end = datetime.datetime.now()
elapsed = end - start
print('Total training time: ', str(elapsed))


# Out[26]:

Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 13s 271
us/step - loss: 0.4893 - acc: 0.8653 - val_loss: 0.213
1 - val_acc: 0.9400
Epoch 2/20
48000/48000 [==============================] - 12s 254
us/step - loss: 0.1713 - acc: 0.9494 - val_loss: 0.124
5 - val_acc: 0.9656
Epoch 3/20
48000/48000 [==============================] - 17s 348
us/step - loss: 0.1062 - acc: 0.9685 - val_loss: 0.095
2 - val_acc: 0.9734
Epoch 4/20
48000/48000 [==============================] - 12s 254
us/step - loss: 0.0797 - acc: 0.9765 - val_loss: 0.087
4 - val_acc: 0.9754
Epoch 5/20
48000/48000 [==============================] - 17s 348
us/step - loss: 0.0671 - acc: 0.9792 - val_loss: 0.078
8 - val_acc: 0.9769
Epoch 6/20
48000/48000 [==============================] - 15s 303
us/step - loss: 0.0591 - acc: 0.9821 - val_loss: 0.071
0 - val_acc: 0.9796
Epoch 7/20
48000/48000 [==============================] - 19s 390
us/step - loss: 0.0522 - acc: 0.9838 - val_loss: 0.081
2 - val_acc: 0.9760
Epoch 8/20
48000/48000 [==============================] - 13s 275
us/step - loss: 0.0473 - acc: 0.9853 - val_loss: 0.067
5 - val_acc: 0.9803
Epoch 9/20
48000/48000 [==============================] - 13s 281
us/step - loss: 0.0419 - acc: 0.9870 - val_loss: 0.069
4 - val_acc: 0.9804
Epoch 10/20
48000/48000 [==============================] - 13s 262
us/step - loss: 0.0394 - acc: 0.9877 - val_loss: 0.065
9 - val_acc: 0.9812
Epoch 11/20
48000/48000 [==============================] - 13s 273
us/step - loss: 0.0354 - acc: 0.9892 - val_loss: 0.070
0 - val_acc: 0.9788
Epoch 12/20
48000/48000 [==============================] - 13s 279
us/step - loss: 0.0324 - acc: 0.9898 - val_loss: 0.070
9 - val_acc: 0.9803
Epoch 13/20
48000/48000 [==============================] - 14s 291
us/step - loss: 0.0287 - acc: 0.9911 - val_loss: 0.069
4 - val_acc: 0.9811
Epoch 14/20
48000/48000 [==============================] - 15s 320
us/step - loss: 0.0274 - acc: 0.9911 - val_loss: 0.069
4 - val_acc: 0.9819
Epoch 15/20
48000/48000 [==============================] - 16s 344
us/step - loss: 0.0249 - acc: 0.9921 - val_loss: 0.071
2 - val_acc: 0.9817
Epoch 16/20
48000/48000 [==============================] - 14s 302
us/step - loss: 0.0229 - acc: 0.9925 - val_loss: 0.068
8 - val_acc: 0.9822
Epoch 17/20
48000/48000 [==============================] - 15s 305
us/step - loss: 0.0211 - acc: 0.9933 - val_loss: 0.072
8 - val_acc: 0.9811
Epoch 18/20
48000/48000 [==============================] - 13s 270
us/step - loss: 0.0189 - acc: 0.9938 - val_loss: 0.066
4 - val_acc: 0.9843
Epoch 19/20
48000/48000 [==============================] - 13s 266
us/step - loss: 0.0177 - acc: 0.9943 - val_loss: 0.082
8 - val_acc: 0.9799
Epoch 20/20
48000/48000 [==============================] - 14s 295
us/step - loss: 0.0159 - acc: 0.9945 - val_loss: 0.076
3 - val_acc: 0.9808
Total training time:  0:04:45.057634
```

Comments

As we can see in the above table, by using convolutional layers instead of Fully Connected and also using *Adam()* function instead of *SGD()* are loss in much less than before. This is only because of the capability of Convolution layers and advantage of Adam() function.

TOTAL TRAINING TIME FOR FC MODEL

0:04:45.057634 for 20 epochs with 128 batch size.

E. Visualization (Plot training behaviour)

To visualize it we can use the **plot_history()** function that we have created earlier in our FC model.

The code below illustrates the output.

```
# In[27]:

plot_history(network_historyCNN)
```

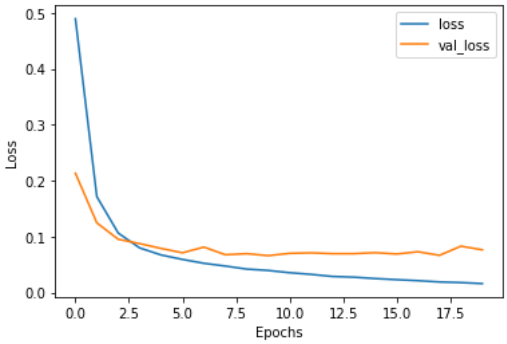

Fig. 11. *CNN Model Loss and Validation Loss*

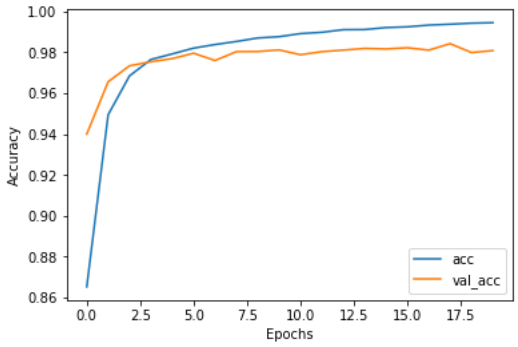

Fig. 12. *CNN Model Accuracy and Validation Accuracy*

F. Evaluation (Evaluate the model on the test set)

This step is would be exactly the same as FC model.

The code below illustrates the output.

```
# In[28]:

# Evaluation
test_loss, test_acc = CNNModel.evaluate(X_test2D,
Y_test1D)
test_labels_p = CNNModel.predict(X_test2D)

# Out[28]:

10000/10000 [==============================] - 1s
123us/step
```

```
# In[29]:

print("test_loss: ", test_loss)
print("test_acc: ", test_acc)


Out[29]:

test_loss:  0.06320389996870363
test_acc:  0.9828000068664551
```

CNN MODEL ACCURACY AND LOSS

Total training time: *0:04:45.057634 for 20 epochs with 128 batch size.*

batch size
test_loss: *0.06320389996870363*
test_acc*: 0.9828000068664551*

```
# In[30]:

import numpy as np
test_labels_p = np.argmax(test_labels_p, axis=1)
pd.DataFrame(test_labels_p)

# Out[30]:
```

	0
0	7
1	2
2	1
3	0
4	4
...	...
9995	2
9996	3
9997	4
9998	5
9999	6

10000 rows × 1 columns

Table above shows the predicted number for the test sets using CNN model.

III. COMPARISION BETWEEN FC AND CNN NEURAL NETWORK MODELS

Table below is the comparison between two different type of Neural Network models, **CNN** model and **FC** model, which we trained using **MNIST** dataset used to recognise hand written digits in images.
as we can see FC model is faster to train but not as accurate as CNN model.

CNN MODEL ACCURACY AND LOSS

Total training time: *0:04:45.057634 for 20 epochs with 128 batch size.*

batch size
test_loss: *0.06320389996870363*
test_acc*: 0.9828000068664551*

FC MODEL ACCURACY AND LOSS

Total training time: *0:01:56.290826 for 20 epochs with 128 batch size*
test_loss: *0.40140881116390226*
test***_acc****: 0.8966000080108643*

[1] yann.lecun, "exdb," [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[2] researchgate, "figure," [Online]. Available: https://www.researchgate.net/figure/Example-of-fully-connected-neural-network_fig2_331525817.

[3] docs.scipy., "doc/numpy," [Online]. Available: https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html.

[4] tensorflow, "api_docs/python," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/utils/to_categorical .

[5] TOWARDSDATASCIENCE, "A-COMPREHENSIVE-GUIDE-TO-CONVOLUTIONAL-NEURAL-NETWORKS-THE-ELI5-WAY-3BD2B1164A53," [Online]. Available: HTTPS://TOWARDSDATASCIENCE.COM/A-COMPREHENSIVE-GUIDE-TO-CONVOLUTIONAL-NEURAL-NETWORKS-THE-ELI5-WAY-3BD2B1164A53.

[6] freecodecamp, "news," [Online]. Available: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/.

[7] computersciencewiki, "index.php," [Online]. Available: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[8] deepai, "machine-learning-glossary-and-terms," [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/inception-module .

[9] stack.imgur, "XD2O4.png," [Online]. Available: https://i.stack.imgur.com/XD2O4.png.

[10] O01D7.png, "stack.imgur.," [Online]. Available: https://i.stack.imgur.com/O01D7.png.