

Pattern Recognition 2021-2022

Final Project , Papadopoulos Aristeidis A.M.:57576

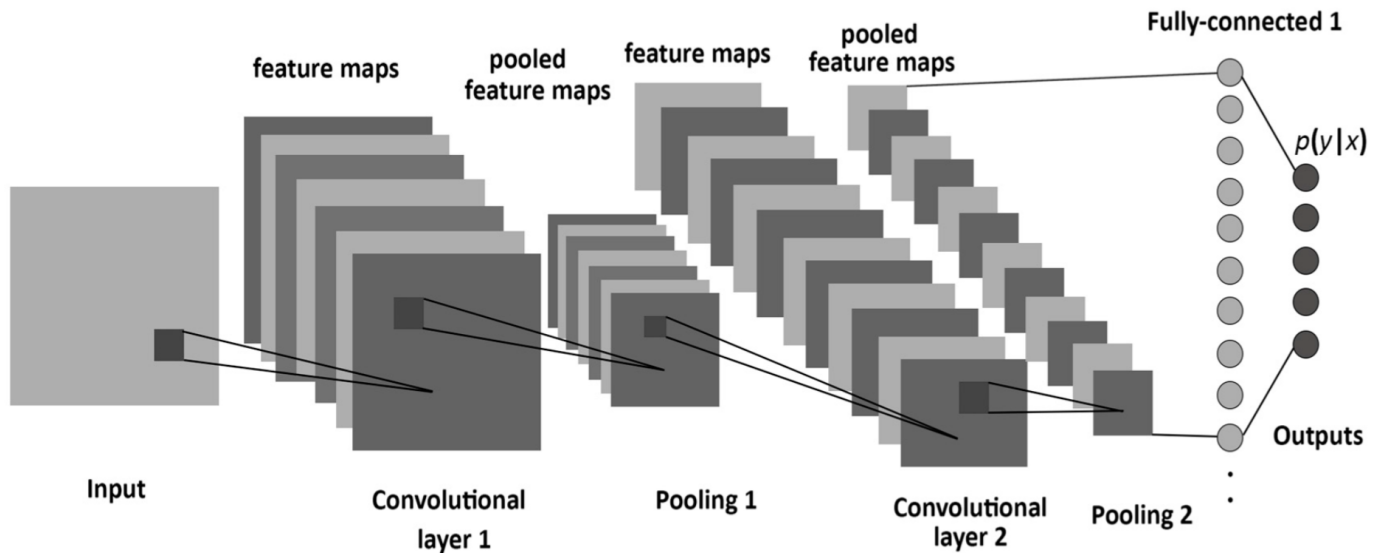


Image 1 Typical Layout of a Convolutional Neural Network (CNN)

Dataset synopsis

In this project, the following three datasets are being used:

- *Fashion_MNIST:*

This dataset contains 70.000 images, sized 28x28 in grayscale (only gray color variations) from 10 different clothing objects (T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle Boot), which are being divided in 60.000 images for training and 10.000 images for testing. The following image contains examples from the dataset:



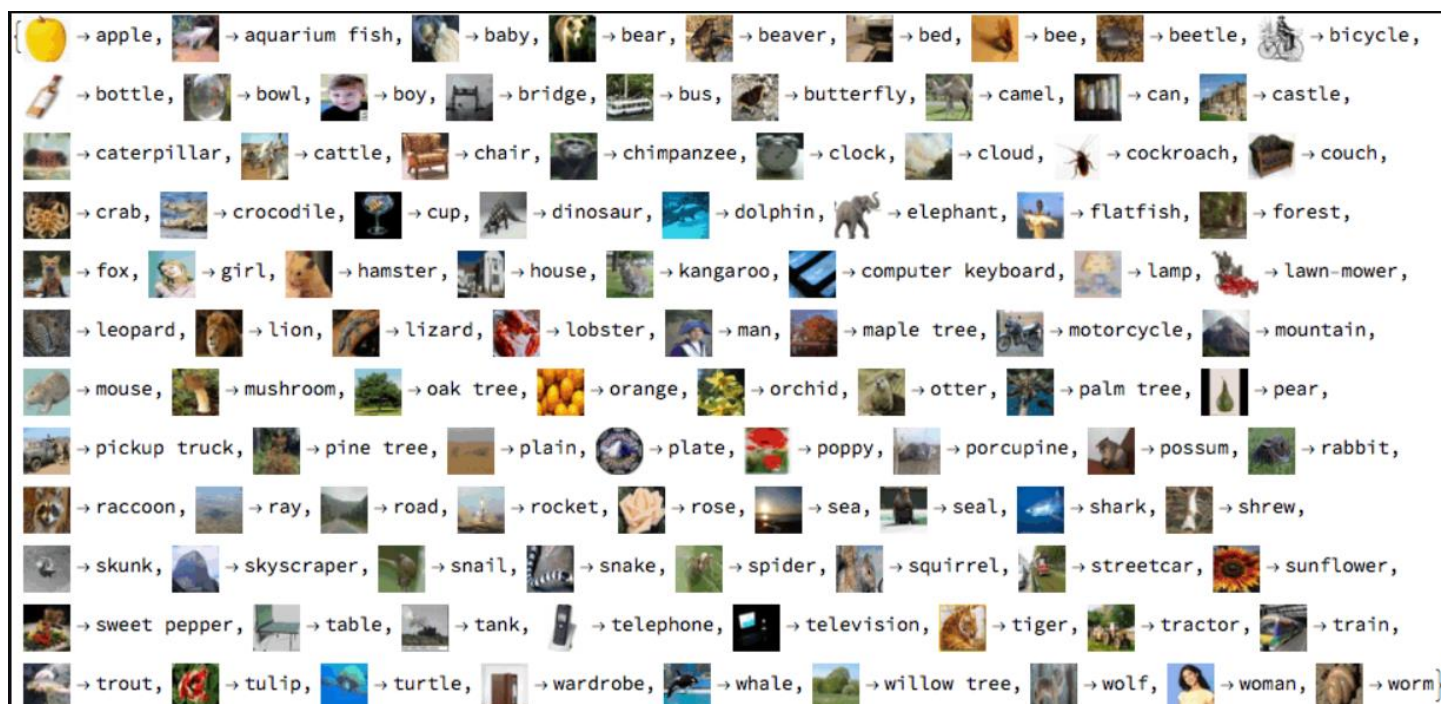
- *Cifar-10:*

The Cifar-10 dataset is consisted of 60.000 images, dimensions of 32x32 in RGB format, divided in 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). They are divided in 50.000 training images and 10.000 testing images, with each image being like the following:



- *Cifar-100:*

Finally, Cifar-100 dataset contains 60.000 images of dimensions 32x32 in RGB format, with 100 classes being merged into 20 superclasses. They are being split in 50.000 training images (500 for each class) and in 10.000 testing images (100 for each class), with the images having the following format:



Data preprocessing

Data preprocessing is a widely used technique, being used to transform the training images, ultimately to optimize the network's accuracy. The following techniques are used in every dataset.

- **Transform image dimensions:**

Whenever a Multilayer Perceptron Network is used, the images are being transformed so that they may be used properly by the input layer.

- **Divide images for training, validation, testing sets:**

By default, the dataset is split in training and test datasets, as mentioned before. Furthermore, to create a validation dataset, the usage of *validation_split* equal to 0.1 is defined, where the 10% of the training dataset is excluded and used as validation dataset.

- **Normalization:**

Instead of using the original values of each pixel, i.e., 0 to 255, each pixel is being normalized between 0 to 1. Using this method, the training is faster, and the probability of the network being trapped in local maxima is reduced.

- **One-Hot encoding:**

The vectors that represent the classes that are to be classified, are encoded using one-hot encoding, in order to use categorical cross for the classification. For example, the 1st class is being represented as 00...01, the 2nd as 00...10 and so on.

- **Image Augmentation:**

To help the network generalize, usually training data are transformed with rotation, zoom, shearing, displacements in each axis and more . In that way, more data are created, which give the model flexibility in its classification ability. However, because the datasets used are meticulously created, the Image Augmentation techniques don't increase the accuracy by lot, so they are not being used in this project.

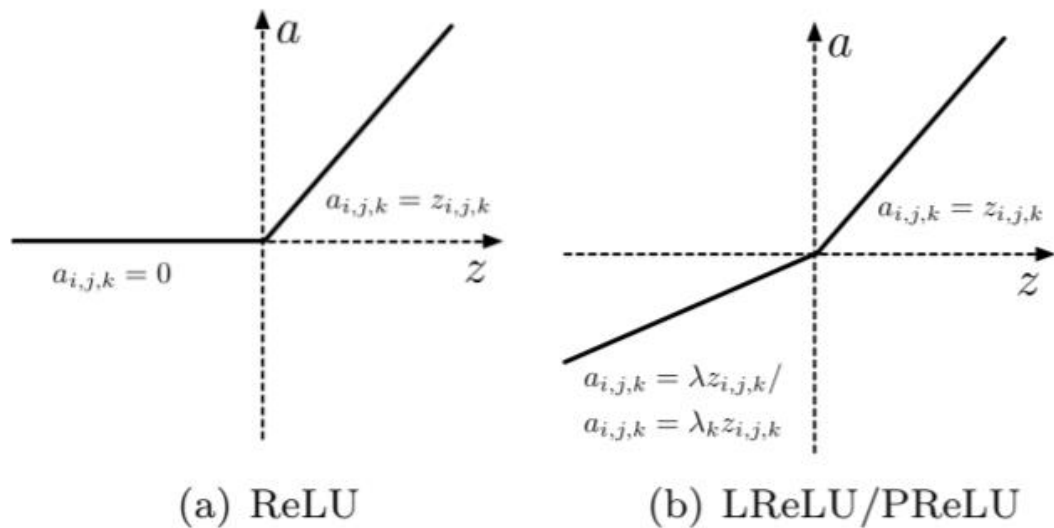
Layers used

In each network, the following layers are used:

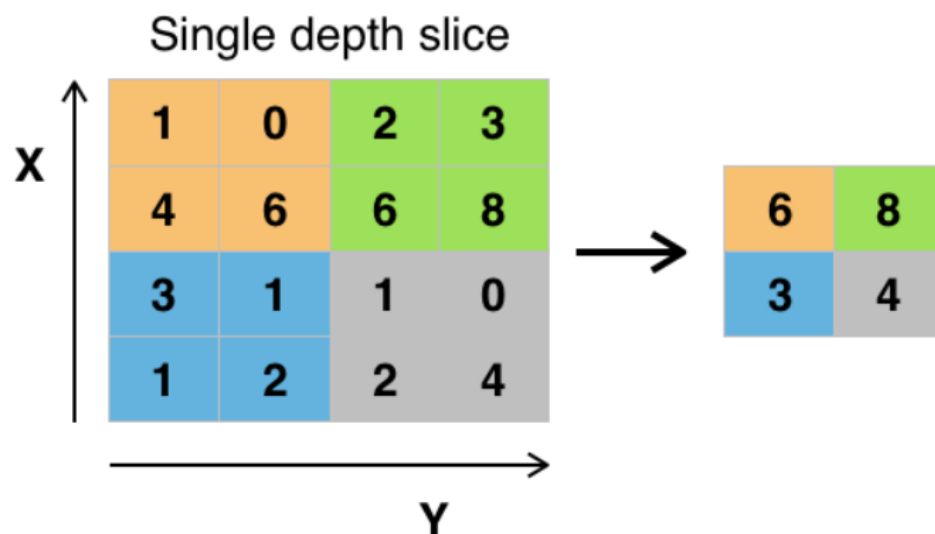
- **Conv2D:**

This layer implements convolution between its input and a kernel of fixed size, to detect features that will help classify the data used. One parameter of this layer is the kernel size, in other words the filter used for the convolution. παράμετρος του στρώματος αυτού, είναι το μέγεθος του πυρήνα, δηλαδή του φίλτρου που χρησιμοποιείται. The smaller its dimensions, the more local features are being detected, while the bigger its dimensions are, the more representative the detected

features are. Furthermore, padding is one argument, and it defines whether padding with zeros or not will be used. Here, it's used to pad with zeros evenly to the left/right or up/down of the input. Then, stride is another argument, which defines the number of pixels to be skipped between two consecutive convolutions. Lastly, an activation function is used, which is either *relu* or *leaky_relu*, while the number of filters/units of the layer represents the dimensionality of the layer output.



- **MaxPool:**



Example of max-pooling, sized 2x2 with strides=2

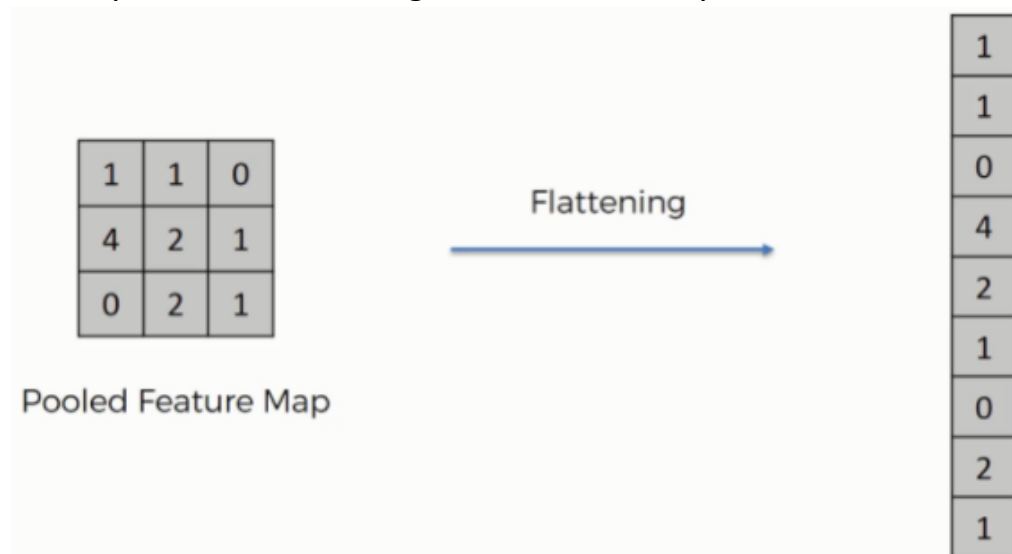
The max-pool layer is a non-linear downsampling method. Essentially, the convolution result (from the Conv2D layer), is divided into sub-squares (their size being one of this layer's parameters), from which the largest value is selected. This way the complexity of training is reduced, with the downside being the possible loss of important information. For example, if the size of the square is defined as 2x2, the largest value out of the four values of the square is selected, while if the size of the square is 3x3, then the largest out of nine values is selected. Finally, the parameter of stride is available, which works the same way as mentioned in the Conv2D layer.

- **Dense:**

It is the most typical layer used, where each neuron of this layer is connected with every neuron of the previous layer. Almost every time, this layer is the last layer of machine learning models, i.e., the model's output where the classification takes place. Main parameters of this kind of layer are the number of neurons, the activation function and more. For multi-classification, the last dense layer is usually used with the softmax activation function, which is a probabilistic representation of the model's prediction.

- **Flatten:**

This layer transforms the feature map into one vector, in order to use the feature map in dense layers, at the final stages of the model layout.



Example of flattening operation

- **Dropout:**

Using this layer, the effect of overfitting is reduced. This is because this layer deactivates randomly connections between neurons during training. Its argument is the probability with which the deactivation happens, and its values are between zero and one.

- **BatchNormalization :**

Finally, this layer normalizes the output of the aforementioned layers. Using batch normalization, the training time is reduced and its results are more stable.

- **Epochs & Callbacks:**

The number of epochs indicates how many times the samples are going to be passed through the network. One epoch equals the passage of all the training samples one time each. High value could risk overfitting the network, while small value could risk low accuracy of the network. In this project, its value is usually equal to 50 ħ 100, but using callbacks, it's not necessary that all the epochs are going to be used.

Furthermore, with callbacks is possible to save the weights of the network's neuron's that result to optimal accuracy and if the accuracy doesn't improve over a few epochs, the training stops, and the saved weights are restored. Its limit is 10 or 15, and the variable tracked is validation loss.

- **Batch Size:**

This variable's value defines the number of samples, after which the renewal of the network's weights is going to happen. Having small value means that the weights are going to be renew regularly, while high value leads to fast convergence of the weights, where furthermore training will not occur. Its value in this project is one of the following 100,250 ħ 500 , while when it is not defined, its value is calculated automatically at training.

- **Optimizer & Learning Rate:**

Essentially, optimizer is the weight renewal algorithm. Adam is widely used and is considered to be the best, so mostly Adam is used, but Nadam is also used, because this optimizer yields similar results to Adam and works in similar way. Learning rate contributes as to how fast the convergence will occur and to help avoid local optima. Learning rate with large value possibly will lead to fastest convergence of training but also possible avoidance of local optima. Contrary, low value will lead to slower convergence but possible trapping of the learning process to local optima. The default value is 0.001, but also the value 0.01 is used. Furthermore, the use of decaying learning rate is available, but because it's not compatible with Nadam optimizer and it's use with Adam optimizer did not yield better results.

- **Number of hidden layers and units number in each one:**

The number of neurons must be selected properly because high number could lead to overfitting and to the creation of an overly complicated network, while low number could lead to underfitting and inability to classify properly. Typically, their values are found after trial & error, and they depend on the problem. Usual values are 16,32,64,128,256 and so on.

- **Activation functions:**

As it has already been noted, relu or leaky_relu are used. However someone could also use tanh, sigmoid and more.

- **Shuffle:**

Using shuffle, each epoch, the samples are inserted randomly into the network, instead of them entering the network in the same order. This randomness has been

observed that sometimes it increases the accuracy of the model. Wherever I observed such an increase in accuracy, shuffle is used.

Networks Implemented

General:

- In my networks implementations, i.e., the ones not defined by the project description, I tried to compromise between the number of training parameters and the model accuracy.
- Almost exclusively the filters dimensions are 3x3, because filters with even number of dimensions divide symmetrically the previous level's pixels around the current output, while 1x1 develop great locality to the features that they detect.
- The logic that was used, was that while the network deepens, the number of neurons in each layer increases, and after each Conv2d layer, usually a maxpool layer and a dropout layer follows.
- Loss as a metric refers to the value of the loss function, which is categorical crossentropy.

- **Fashion_MNIST Dataset** ([file fashion_mnist_57576.ipynb](#)):

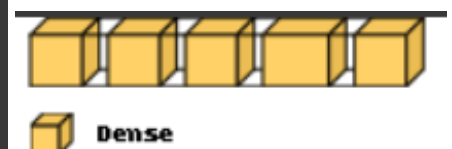
1. Multi-Layer Perceptron Network(FC NN):

Following the description about this network, a Multilayer Perceptron network is implemented, with 3 hidden layers of 64, 128 and 256 units. Furthermore, the output layer has 10 units, because the classes in this problem are 10, while the input layer has an input shape equal to $28*28*1 = 784$, a number which derives from the images height and width.αριθμός. In the following image the number of parameters is displayed using the *model.summary* command and then the network is visualized using *VisualKeras* module:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 32)	25120
dense_1 (Dense)	(None, 64)	2112
dense_2 (Dense)	(None, 128)	8320
dense_3 (Dense)	(None, 256)	33024
dense_4 (Dense)	(None, 10)	2570

```
=====
Total params: 71,146
Trainable params: 71,146
Non-trainable params: 0
```



Trainable parameters are defined as the number of units that are able to change their weight values through the use of backpropagation algorithm or other algorithm. The non-trainable refer to the units that don't change their values, e.g. the Dropout and Batch normalization values.

2. Convolutional Network (CNN):

Same as before, the requested network consists of two hidden layers of 32 and 64 filters, each followed by a maxpooling layer. The input layer has 32 filters(arbitrary value), with input shape equal to the dimensions of the images, i.e., 28x28x1(last channel is one because the images are in grayscale format).The output layer once again has 10 units, for the same reason as previously. The next image displays the model summary and the visualization of the network:

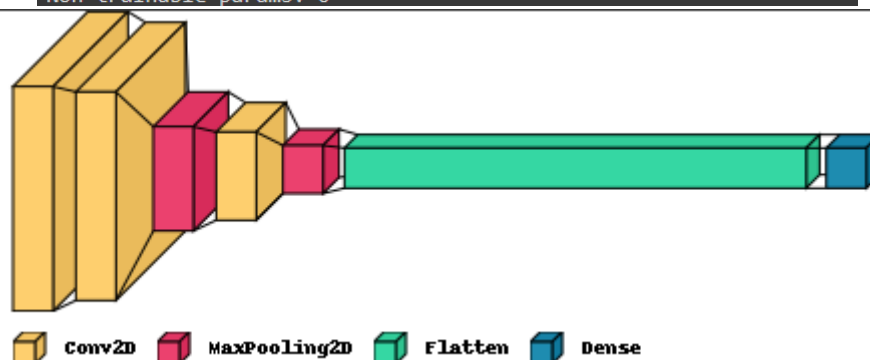
```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
conv2d_1 (Conv2D)	(None, 26, 26, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten (Flatten)	(None, 2304)	0
dense_5 (Dense)	(None, 10)	23050

```

=====
Total params: 51,114
Trainable params: 51,114
Non-trainable params: 0

```



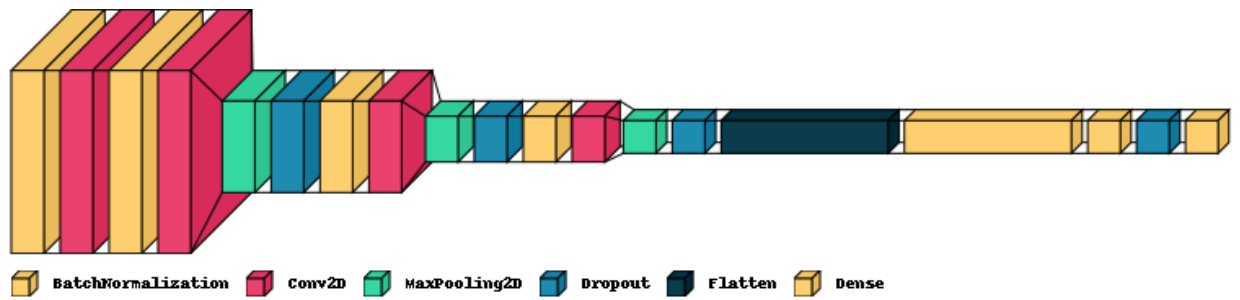
The flatten layer is used to convert the feature maps to one vector, so that the classification may happen in the output layer.

3. Personal model implementation:

I chose to create a convolutional neural network, having in mind to increase the accuracy as much as it was possible, without increasing the a lot the number

parameters. The more the parameters are, the more resources are used by the model. After trial and error, the model I implemented was the following:

Model: "sequential_2"		
Layer (type)	Output Shape	Param #
batch_normalization (Batch Normalization)	(100, 28, 28, 1)	112
conv2d_3 (Conv2D)	(100, 28, 28, 32)	320
batch_normalization_1 (Batch Normalization)	(100, 28, 28, 32)	112
conv2d_4 (Conv2D)	(100, 28, 28, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(100, 14, 14, 64)	0
dropout (Dropout)	(100, 14, 14, 64)	0
batch_normalization_2 (Batch Normalization)	(100, 14, 14, 64)	56
conv2d_5 (Conv2D)	(100, 14, 14, 64)	16448
max_pooling2d_3 (MaxPooling2D)	(100, 7, 7, 64)	0
dropout_1 (Dropout)	(100, 7, 7, 64)	0
batch_normalization_3 (Batch Normalization)	(100, 7, 7, 64)	28
conv2d_6 (Conv2D)	(100, 7, 7, 64)	36928
max_pooling2d_4 (MaxPooling2D)	(100, 4, 4, 64)	0
dropout_2 (Dropout)	(100, 4, 4, 64)	0
flatten_1 (Flatten)	(100, 1024)	0
batch_normalization_4 (Batch Normalization)	(100, 1024)	4096
dense_6 (Dense)	(100, 32)	32800
dropout_3 (Dropout)	(100, 32)	0
dense_7 (Dense)	(100, 10)	330
=====		
Total params: 109,726		
Trainable params: 107,524		
Non-trainable params: 2,202		



Essentially it is an evolution of the previous network.

4. Results Comparison:

It must be noted that the results are indicative, because an important parameter is the weight initialization and the values that the random parameters have, where those change every time, the network is retrained. In that case, the following results are a snapshot of each network's function.

Firstly, training the aforementioned models and using the test set, through *model.evaluate* the following results are produced:

```
Test loss: 0.3622902035713196
Test accuracy: 0.873199999332428
```

Image 4.1 Results using the fully connected network

```
Test loss: 0.241521418094635
Test accuracy: 0.9165999889373779
```

Image 4.2 Results using the convolutional network

```
Test loss: 0.18925587832927704
Test accuracy: 0.9359999895095825
```

Image 4.3 Results using my personal implementation

The test loss value corresponds to the value that the loss function returns, with the test accuracy value representing the percentage of correct classifications by each network.

Subsequently, observing the evolution of training and validation accuracy and loss, one could notice whether the training is stable or not, where it is observed that the more complex the model is, the more epochs are needed to achieve optimal results(considering that the model is deeper). Also, it is observed that there is variance in the validation set metrics, while the training set metrics have a smoother progress:

Accuracy & Loss Plots in FashionMnist Dataset

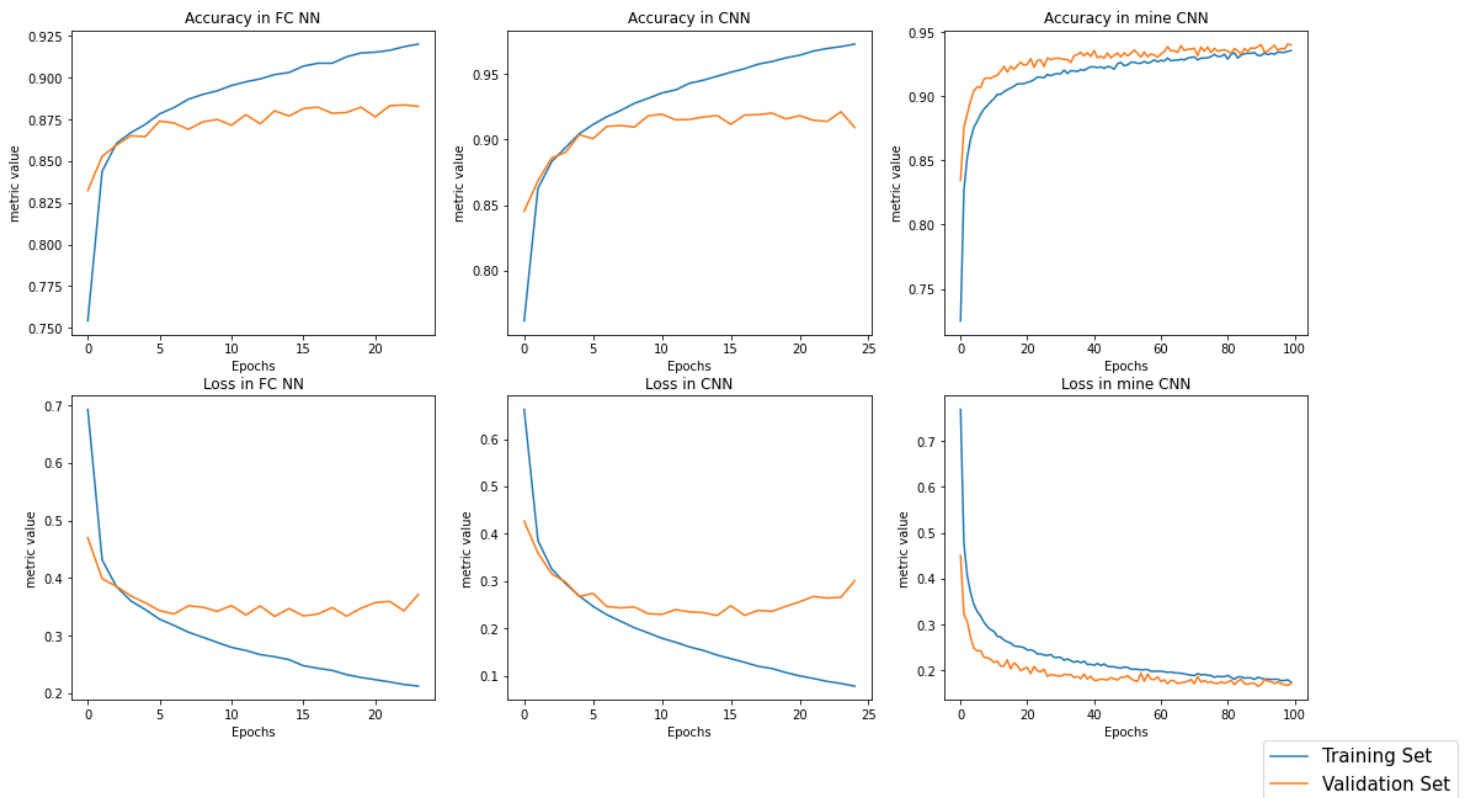


Image 4.4 Plots of loss and accuracy for every network

Then, using and plotting a confusion matrix, it can be easily understood how the classification of the test set samples is being made, since it is visible how the classification is for each class:

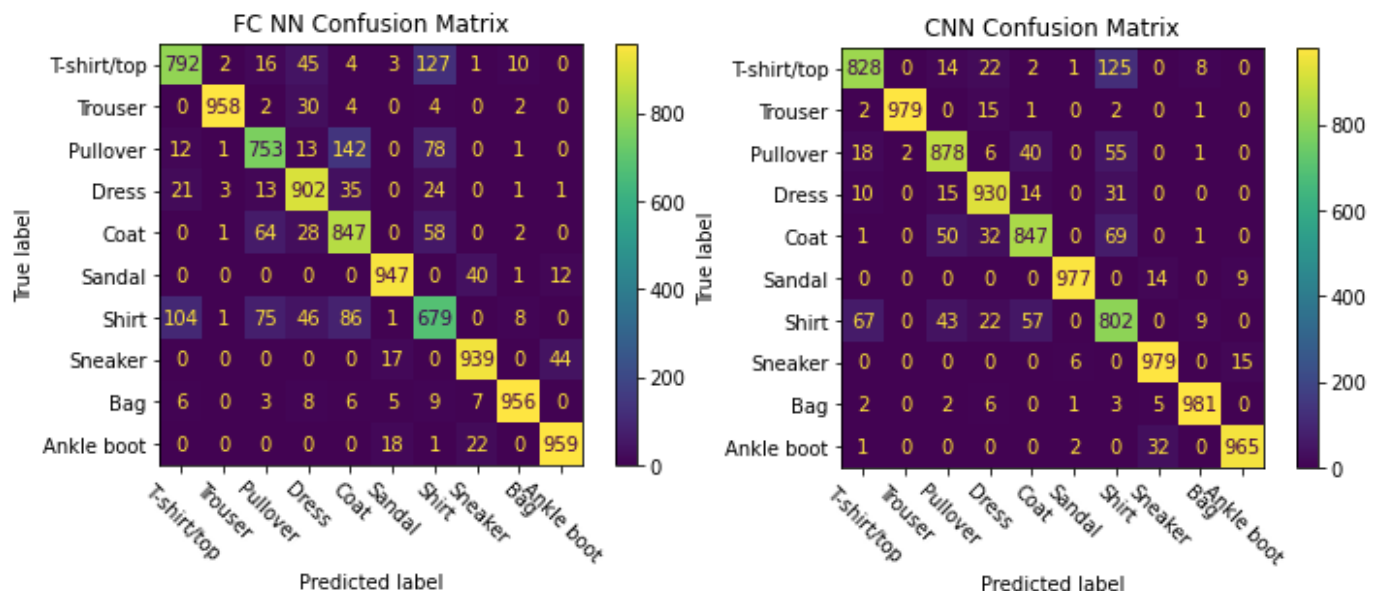


Image 4.5a,b Confusion matrices for Multilayer Perceptron Network(left) & Convolutional NN (right)

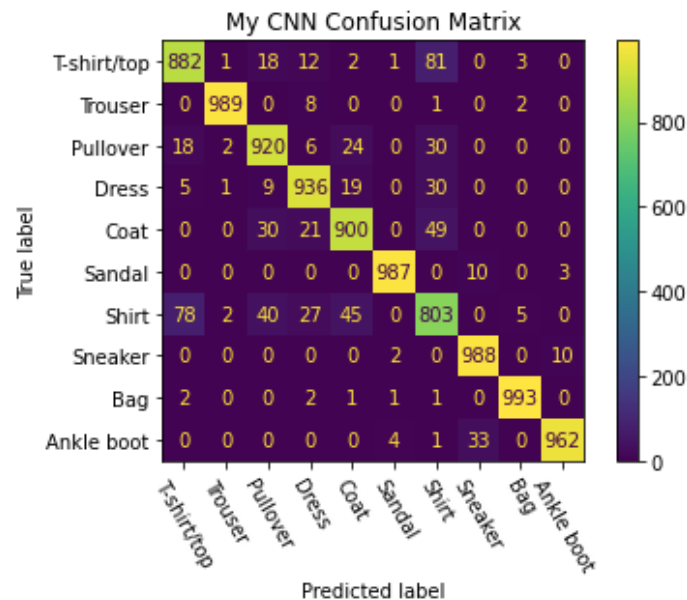


Image 4.5c Confusion Matrix for my personal implementation

Based on the above, one could observe that misclassification happens for the classes T-shirt/Top and Shirt, as one could expect, because those two look alike. It looks like the sleeves as a feature are the most difficult features to learn, because wrong classifications for the most part happen on classes that have sleeves.

Finally, the following images displays 15 predictions from each model to understand more easily the performance of each model (even though the images are randomly selected, one could reach a conclusion):

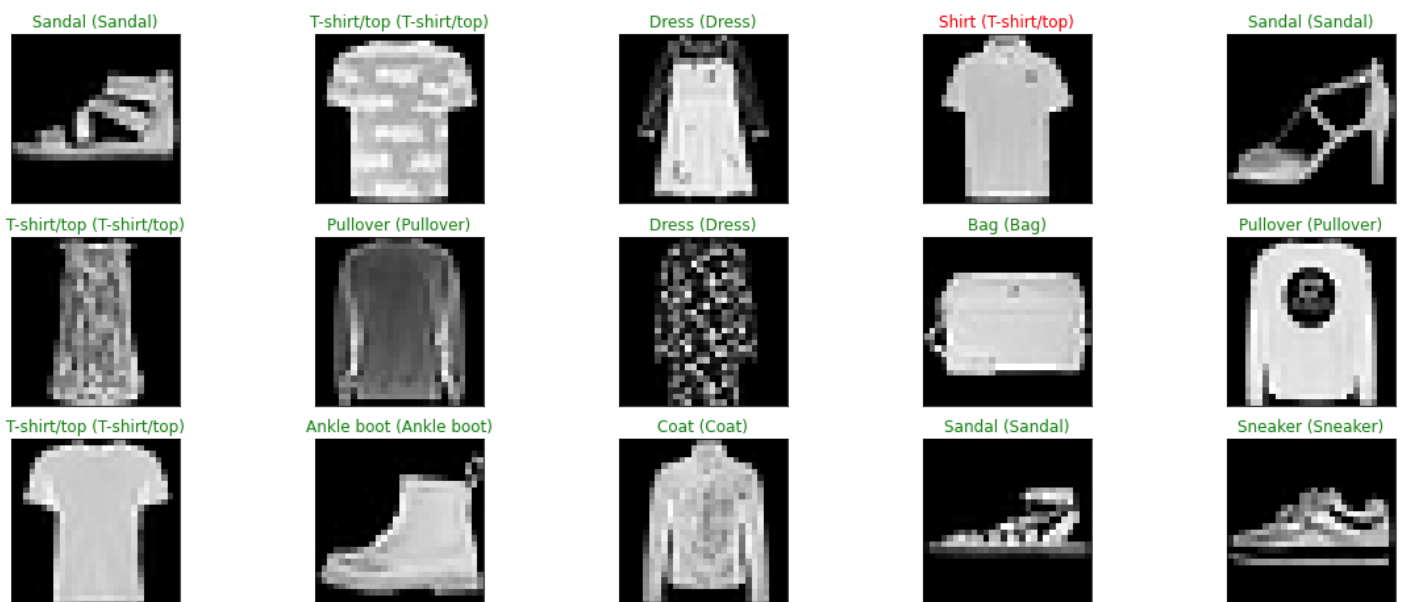


Image 4.6 Prediction results using FC NN

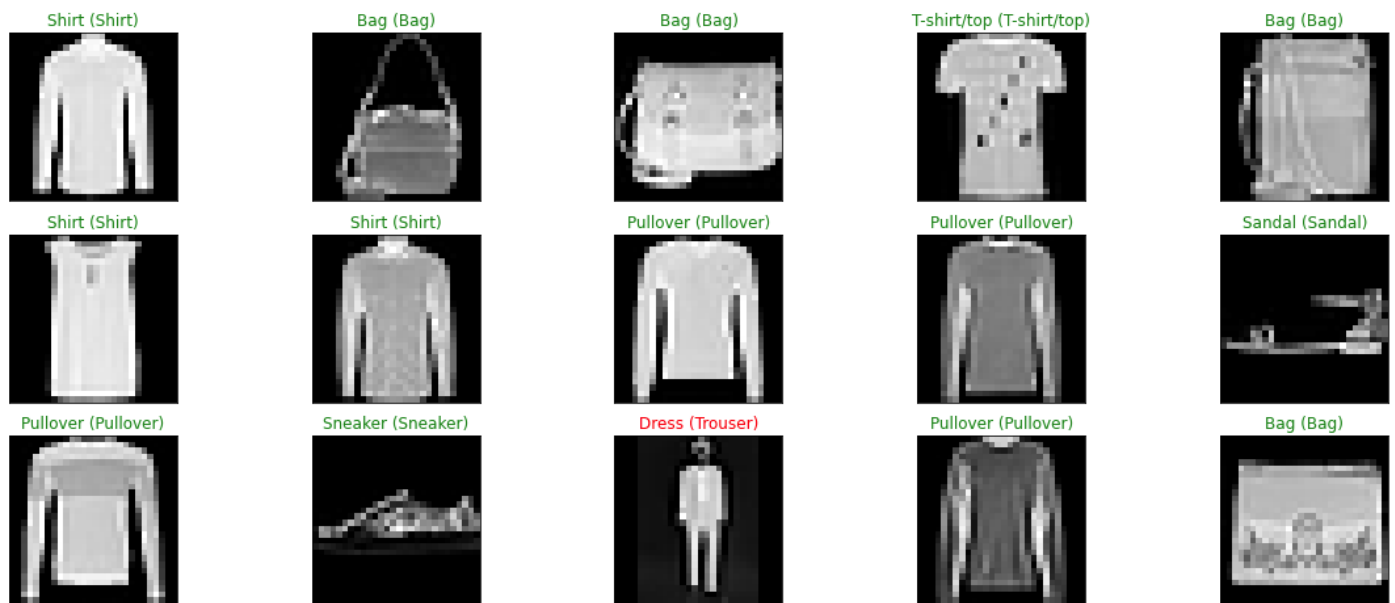


Image 4.7 Prediction results using CNN



Image 4.8 Prediction results using my CNN

In conclusion, it is observed that whenever the complexity of the model is increased, the accuracy of that model is increased, while the loss is reduced. Furthermore, this dataset is relatively simple, so the model accuracies are similar, a statement that does not apply for the following datasets.

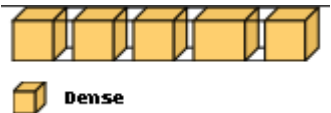
- **Cifar10 Dataset** ([file cifar10_57576.ipynb](#)):

1. Multi-Layer Perceptron Network (FC NN):

Similarly with the previous dataset, a Fully Connected NN is implemented, with the difference being that this model has one more hidden layer, sized 256. In addition, the image dimensions change, as now the images are 32x32 RGB (i.e. 3 channels, red, green, blue), so now the input dimensions are $32 \times 32 \times 3 = 3072$. Since the classes are again 10, the output layer is a dense one, with 10 neurons,

one for each class. So, *model.summary* has the following output, with the second image being the network visualization:

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 32)	98336
dense_6 (Dense)	(None, 64)	2112
dense_7 (Dense)	(None, 128)	8320
dense_8 (Dense)	(None, 256)	33024
dense_9 (Dense)	(None, 10)	2570
=====		
Total params: 144,362		
Trainable params: 144,362		
Non-trainable params: 0		

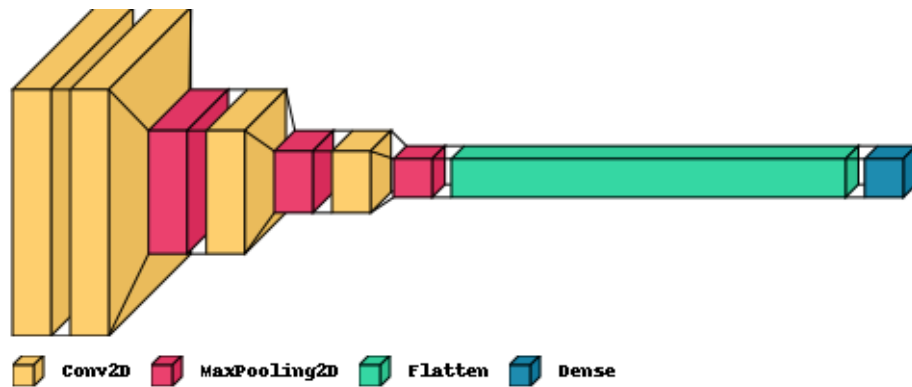


As is expected, with the increase of hidden layers, the trainable parameters also are increased.

2. Convolutional Network (CNN):

As in the previous network, now the model consists of 3 hidden layers with 64,128,256 units, followed by max-pooling layers, while the output layer has 10 units and lastly the input layer has 32 units with input shape equal to 32x32x3:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_3 (Conv2D)	(None, 8, 8, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense_5 (Dense)	(None, 10)	20490
=====		
Total params: 122,986		
Trainable params: 122,986		
Non-trainable params: 0		



3. Personally Implemented Network:

Due to the complexity of this dataset, and in order to achieve as better results as possible, a model was made with 4 hidden layers, each with 32,64,64,128 units respectively, thus incrementing the total model parameters:

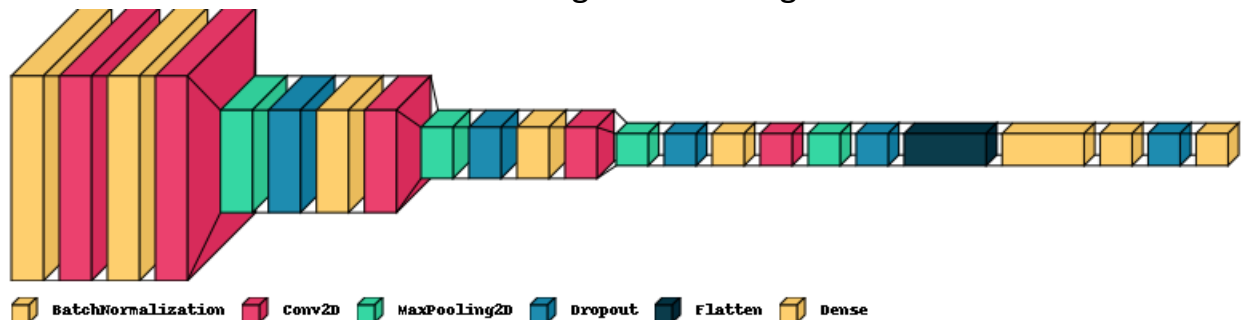
Layer (type)	Output Shape	Param #
batch_normalization (Batch Normalization)	(100, 32, 32, 3)	128
conv2d_4 (Conv2D)	(100, 32, 32, 32)	896
batch_normalization_1 (Batch Normalization)	(100, 32, 32, 32)	128
conv2d_5 (Conv2D)	(100, 32, 32, 32)	9248
max_pooling2d_3 (MaxPooling2D)	(100, 16, 16, 32)	0
dropout (Dropout)	(100, 16, 16, 32)	0
batch_normalization_2 (Batch Normalization)	(100, 16, 16, 32)	64
conv2d_6 (Conv2D)	(100, 16, 16, 64)	18496
max_pooling2d_4 (MaxPooling2D)	(100, 8, 8, 64)	0
dropout_1 (Dropout)	(100, 8, 8, 64)	0
batch_normalization_3 (Batch Normalization)	(100, 8, 8, 64)	32
conv2d_7 (Conv2D)	(100, 8, 8, 64)	36928
max_pooling2d_5 (MaxPooling2D)	(100, 4, 4, 64)	0
dropout_2 (Dropout)	(100, 4, 4, 64)	0
batch_normalization_4 (Batch Normalization)	(100, 4, 4, 64)	16
conv2d_8 (Conv2D)	(100, 4, 4, 128)	73856
max_pooling2d_6 (MaxPooling2D)	(100, 2, 2, 128)	0
dropout_3 (Dropout)	(100, 2, 2, 128)	0
flatten_1 (Flatten)	(100, 512)	0

```

batch_normalization_5 (Batch Normalization)      (100, 512)      2048
dense_6 (Dense)                                   (100, 16)        8208
dropout_4 (Dropout)                              (100, 16)         0
dense_7 (Dense)                                   (100, 10)        170
=====
Total params: 150,218
Trainable params: 149,010
Non-trainable params: 1,208

```

With a visualization of the model being the following:



4. Results comparison:

Each network's results can be seen in the images below, where the loss and accuracy are calculated:

```

Test loss: 1.4195231199264526
Test accuracy: 0.5001000165939331

```

Image 4.1 Results using the fully connected network

```

Test loss: 0.7626890540122986
Test accuracy: 0.7437999844551086

```

Image 4.2 Results using the CNN

```

Test loss: 0.48939210176467896
Test accuracy: 0.8429999947547913

```

Image 4.3 Results using my personal implementation

Considering that my implementation has just a bit more parameters than the CNN network in the project description, the result is very satisfactory. Furthermore, by observing the plots of loss and accuracy, generally the same conclusions can be extracted as before, only now more epochs are needed. The fact that appear fluctuations on the loss and accuracy of the validation set might be a good sign that no overfitting has occurred.

Accuracy & Loss Plots in Cifar10 Dataset

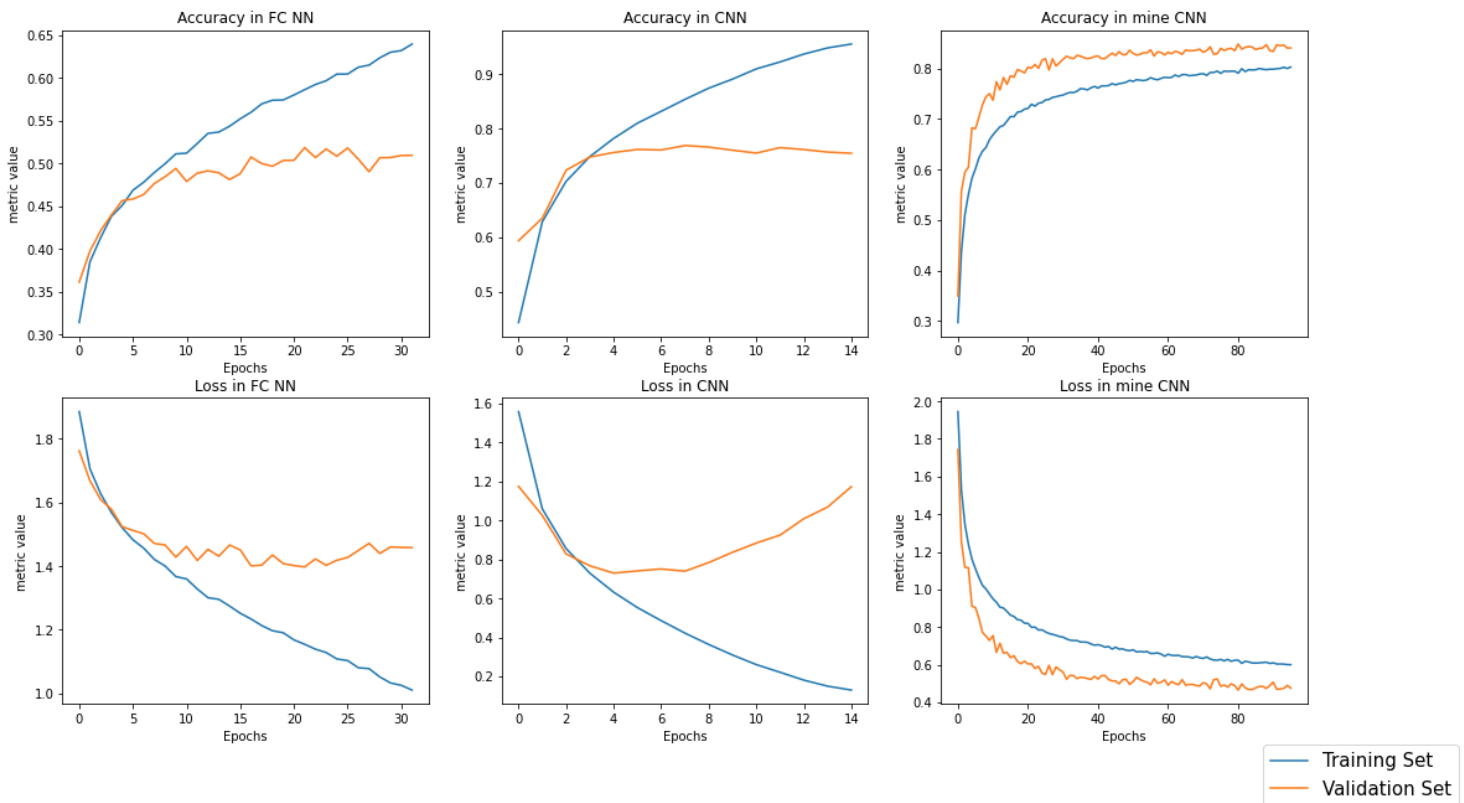


Image 4.4 Plots of loss and accuracy for each model

In the following images, the confusion matrices of each model are plotted. According to them, there seems to be misclassification between the car and dog classes, and to lesser extend between the classes car & truck and deer & horse, as in those classes appear the most misclassified samples.

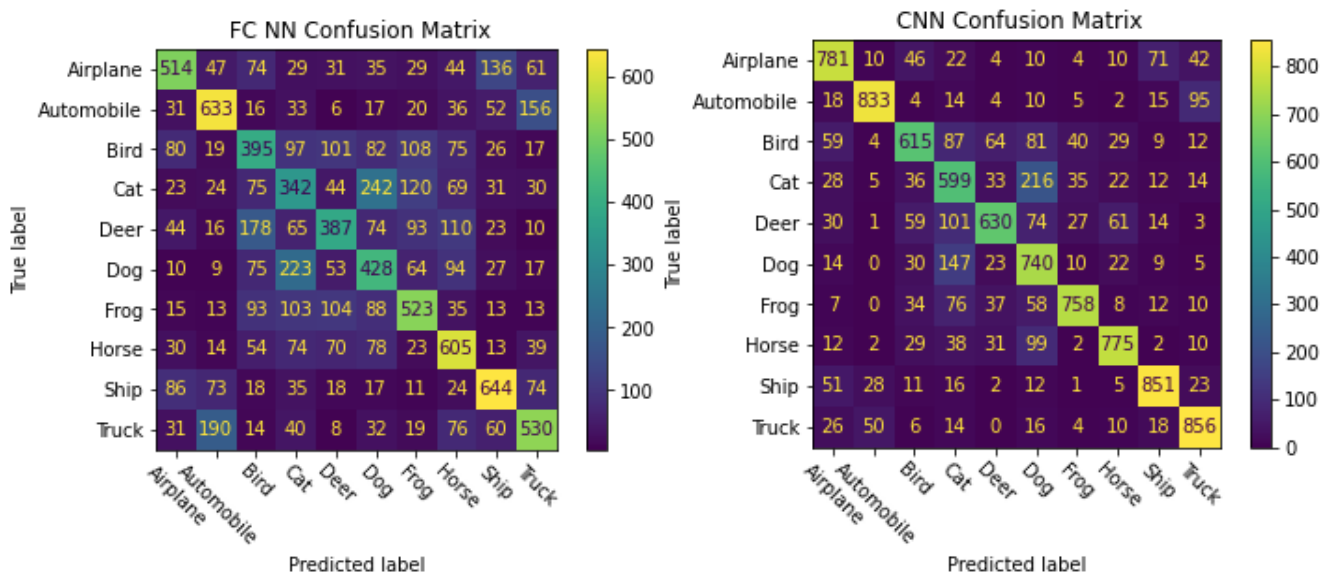


Image 4.5a,b Confusion matrices for Multilayer Perceptron Network(left) & Convolutional NN (right)

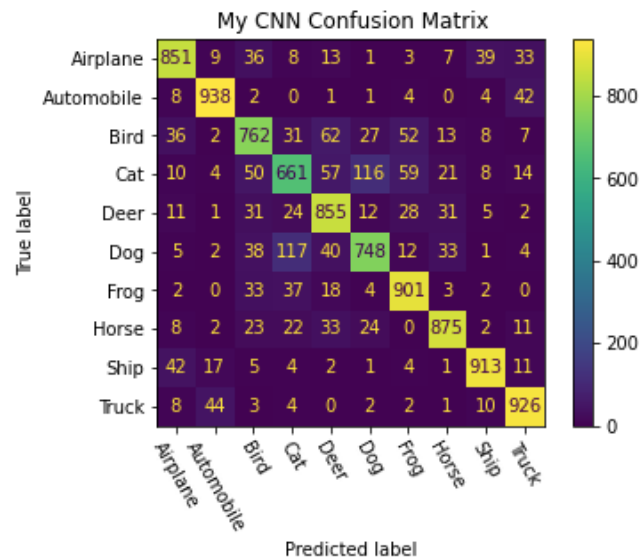


Image 4.5c Confusion Matrix for my personal implementation

Finally, using the three networks, predictions are made with the following results:



Image 4. 6 Prediction results using FC NN

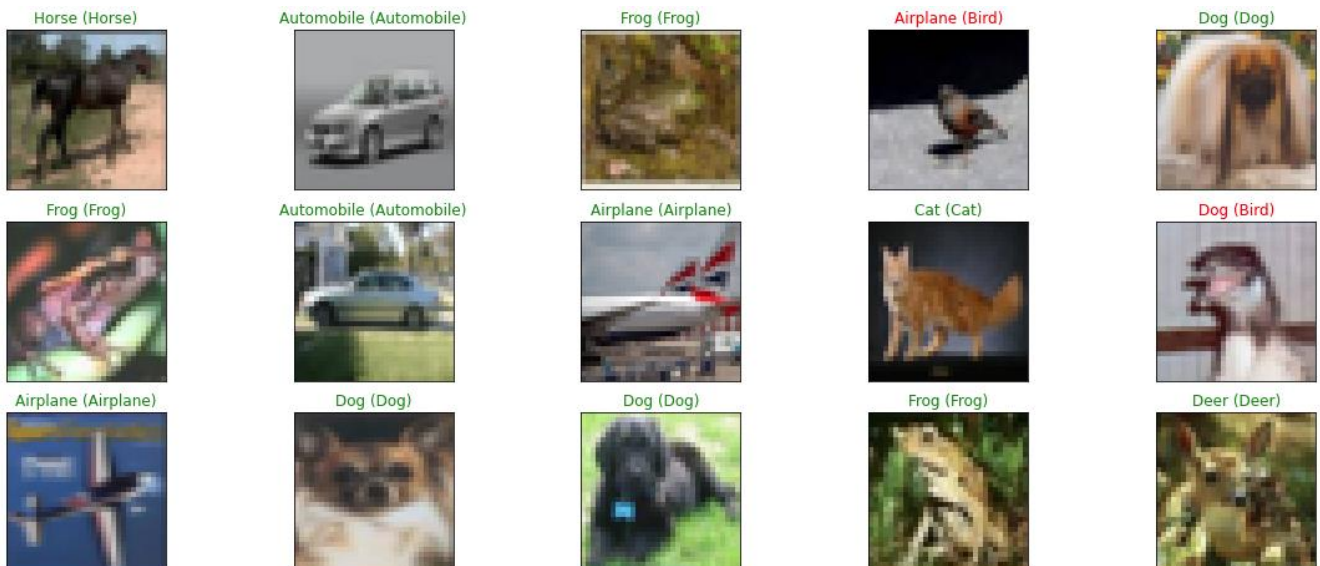


Image 4.7 Prediction results using CNN

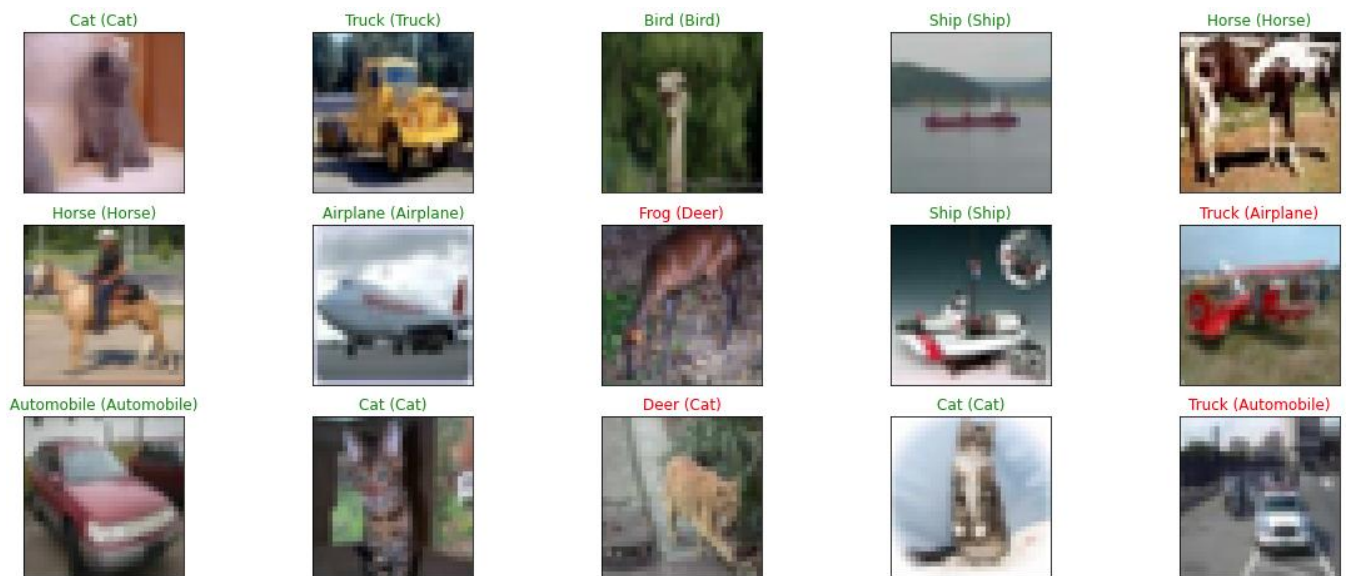
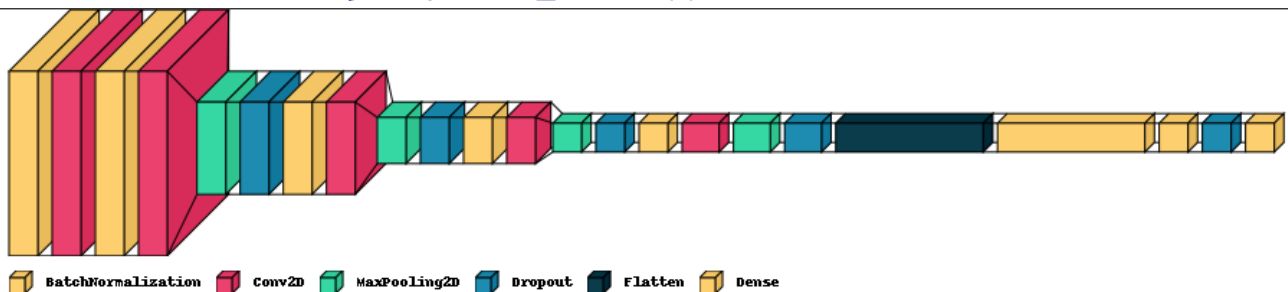


Image 4.8 Prediction results using CNN

- Cifar100 Dataset (file `cifar100_57576.ipynb`):



1. Implementation:

The network I ended up implementing is the following, as seen in *model.summary*:

Layer (type)	Output Shape	Param #
batch_normalization (Batch Normalization)	(100, 32, 32, 3)	128
conv2d (Conv2D)	(100, 32, 32, 32)	896
batch_normalization_1 (Batch Normalization)	(100, 32, 32, 32)	128
conv2d_1 (Conv2D)	(100, 32, 32, 64)	18496
max_pooling2d (MaxPooling2D)	(100, 16, 16, 64)	0
dropout (Dropout)	(100, 16, 16, 64)	0
batch_normalization_2 (Batch Normalization)	(100, 16, 16, 64)	64
conv2d_2 (Conv2D)	(100, 16, 16, 128)	73856
max_pooling2d_1 (MaxPooling2D)	(100, 8, 8, 128)	0
dropout_1 (Dropout)	(100, 8, 8, 128)	0
batch_normalization_3 (Batch Normalization)	(100, 8, 8, 128)	32
conv2d_3 (Conv2D)	(100, 8, 8, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(100, 4, 4, 128)	0
dropout_2 (Dropout)	(100, 4, 4, 128)	0
batch_normalization_4 (Batch Normalization)	(100, 4, 4, 128)	16

```

conv2d_4 (Conv2D)          (100, 4, 4, 256)          295168
max_pooling2d_3 (MaxPooling (100, 2, 2, 256)          0
2D)
dropout_3 (Dropout)        (100, 2, 2, 256)          0
flatten (Flatten)          (100, 1024)               0
batch_normalization_5 (Batc (100, 1024)              4096
hNormalization)
dense (Dense)              (100, 128)               131200
dropout_4 (Dropout)        (100, 128)               0
dense_1 (Dense)            (100, 100)              12900
=====
Total params: 684,564
Trainable params: 682,332
Non-trainable params: 2,232

```

The number of parameters might alienate someone, as it is considerably higher than before, but considering the complexity of this dataset and looking for other networks applied the Cifar100 dataset, the number of parameters is considered acceptable for the accuracy that is achieved.

2. Results:

Firstly, the loss and accuracy metrics for the test set can be seen below:

```

Test loss: 1.4669256210327148
Test accuracy: 0.6014000177383423

```

Image 4.1 Classification results for test set for Cifar100 dataset

Having in mind the 100 classes that are to be classified, the test accuracy is acceptable. In the next plots, the accuracy and loss progress throughout training can be seen.

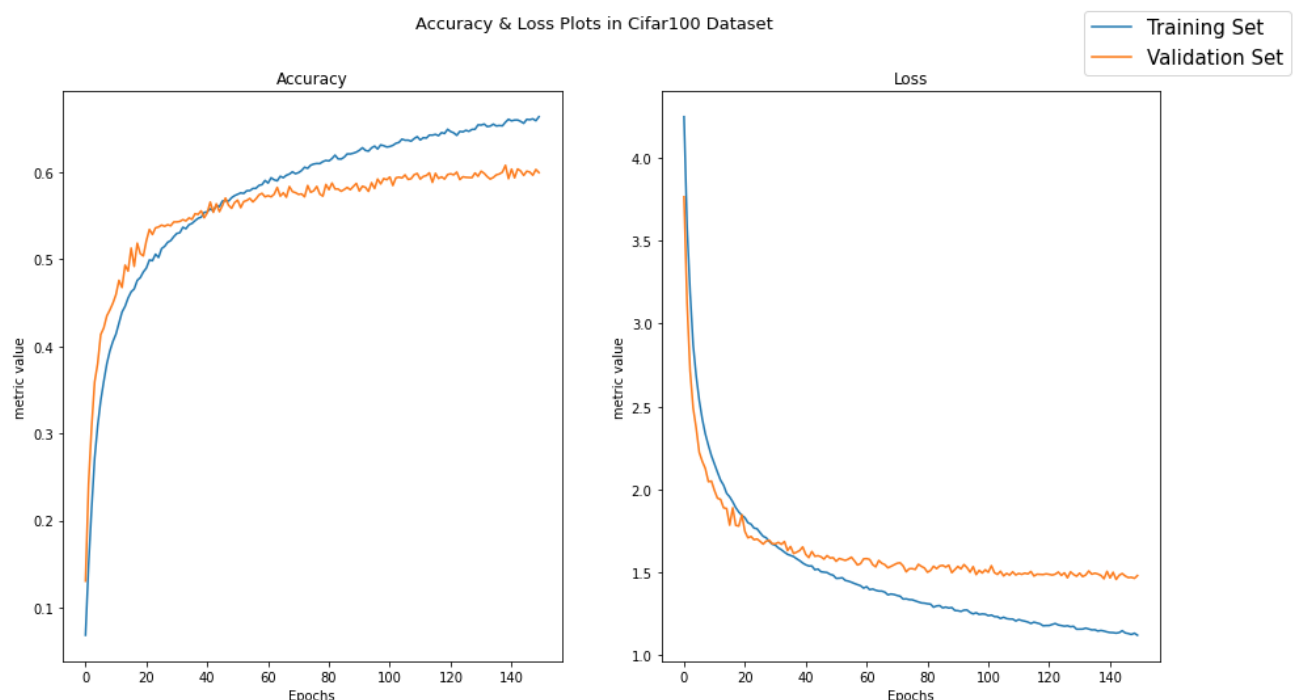


Image 4.2 Plots for loss and accuracy for training and validation set

Based on the above plots, there seems to be a deviation between the metrics for training and validation set. This is probably because the problem of overfitting might be starting to appear. Then, the confusion matrix is the following:

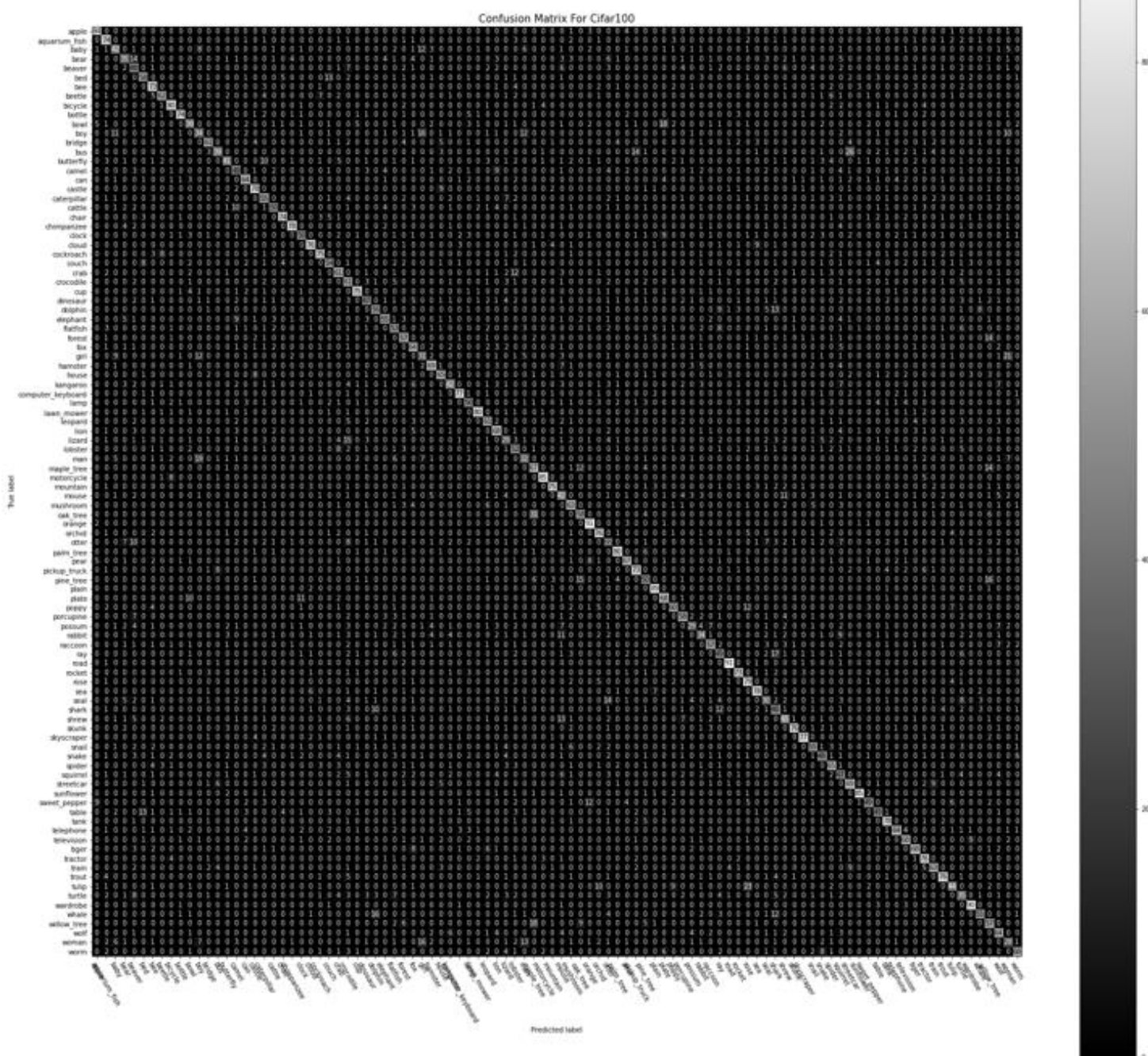


Image 4.3 Confusion Matrix for Cifar100 predictions

The darkest the color is, the less classifications have been made for the two corresponding classes. Finally, using the trained model to make predictions, the results are as follow:



Image 4.4 Some predictions using the trained model

References

- <https://medium.com/analytics-vidhya/how-to-choose-the-size-of-the-convolution-filter-or-kernel-size-for-cnn-86a55a1e2d15>
- https://en.wikipedia.org/wiki/Convolutional_neural_network
- <https://keras.io/api/>
- https://www.tensorflow.org/api_docs/python/tf
- <https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b>
- <https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7>
- <https://github.com/paulgavrikov/visualkeras/>
- Recent Advances in Convolutional Neural Networks by Jiuxiang Gao, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahrudiy, Bing Shuaib, Ting Liub, Xingxing Wang, Li Wang, Gang Wang, Jianfei Cai, Tsuhan Chen
- <https://github.com/zalandoresearch/fashion-mnist>
- <https://www.cs.toronto.edu/~kriz/cifar.html>