ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

AUEB M.S.c in Data Science (part-time)

Deep Learning
MLP - CNN
fashion MNIST dataset

Manolis Proimakis (P3351815)

Introduction

Fashion MNIST is a dataset containing **60000** images for the training set and **10000** images for the test set, all of which represent a fashion item belonging to one of the 10 categories seen below.

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle Boot

Each image is represented by a **28 x 28 grayscale** pixel matrix. After regularizing the values of each pixel to fit in the **[0, 1]** scale we can see below a sample of the representation of the first **50** images along with their categories.

The goal of this project is to find a good **MLP** and **CNN** neural network architecture. We are gonna fit the model on the train set and find the categorical accuracy on the unseen data. The problem is a multi-class classification of images into 10 categories and in many other similar problems, MLPs and CNNs have good results.

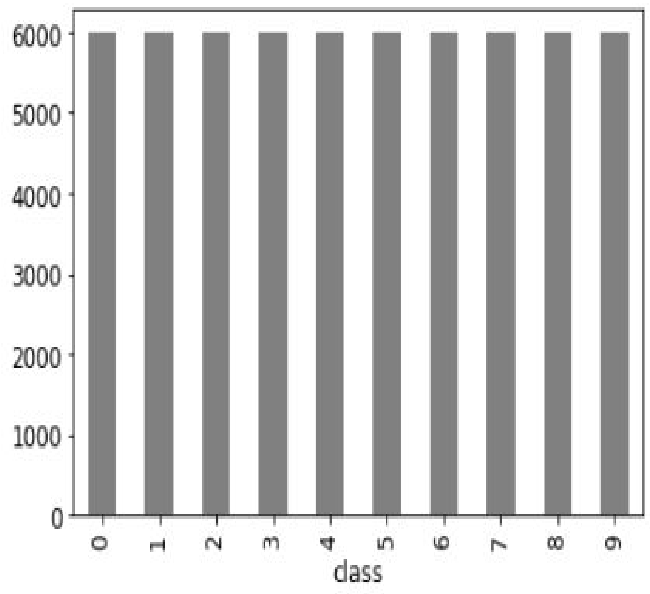
The data do not need any preprocessing steps except normalizing the pixel values to the [0, 1] scale. This will help to make computations faster since neural networks process inputs with small weights and high-value inputs will slow down fitting the model. Since the image representation stays the same in both bands [0-255] and [0-1] we don't lose any information.

Experiments

In my experiments, I am going to use two neural network types that achieve better results with images, simple MLP, and CNN networks. Each architecture will be described in different sections since the general thoughts behind each architecture change significantly.

Datasets

The dataset is very well balanced, as we can see each class has exactly the same samples.



I am gonna stratify split my train set to

- ➔ X_train, Y_train - 48000 samples

Which I 'll use for fitting the models.

- ➔ X_val, Y_val - 12000 samples

Which I 'll use for validating the model on each epoch.

The final evaluation of each model will be done on the X_test, Y_test that contains 10000 samples and the model has never seen before.

Loss Function and Metrics

The task we have is a multi-class classification problem and the loss functions that have proved to be the best for these types of problems are **categorical_crossentropy** and **sparse_categorical_crossentropy**. In my case, I ll use **sparse_categorical_crossentropy** as I have represented my categories as **simple integers from [0] to [9]** and not 1-hot encoded vectors like **[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]** to save time and space, moreover, the underline loss calculation is the same so there is no issue by using either of them. For my metric, again I will use **sparse_categorical_accuracy**, as my classes are simple integers.

- Keras-losses

Optimizers

There are many optimizers that can be used for our task, and this is one of the most important parts of a good model. There are algorithms that follow the generic GD algorithm where many parameters should be tuned like SGD and adaptive algorithms that optimize their parameters e.g. learning rate, momentum. [1]

- **SGD:** This is the defo optimizer with mini-batches (the batch size should be defined cause we can't use the whole dataset for every step due to memory limits), this optimizer without tuning has problems.
  - Finding a proper learning rate requires multiple experiments.
  - Same learning rate applies to all parameters
  - Finding the optimum is easy to be lost when we come across local minima or saddle points,

  Improvements:
  1. **Momentum** is a method that helps accelerate SGD in the relevant direction and dampens oscillations
  2. **NAG** While Momentum first computes the current gradient and then takes a big jump in the direction of the updated accumulated gradient, NAG first makes a big jump in the direction of the previously accumulated gradient, measures the gradient and then makes a correction

- **Adaptive Algorithms** can optimize the learning rate for each parameter. As these adaptive algorithms were introduced in the history they were solving problems of previous algorithms. A brief summary below:
  - **Adagrad** was introduced to automatically adapt the learning rate to the parameters.
  - **Adadelta, RMSprop** were introduced to solve Adagrad's radically diminishing learning rates.
  - **Adam** does what Adalta and RMSprop do but it also takes into account past gradients, something similar to momentum for SGD
  - **Nadam** combines Adam with NAG which was observed that it has better results than simple momentum.
  - **AMSGrad** even though adaptive algorithms are mostly used, it was observed that there are cases e.g. for object recognition that hey fail to converge to an optimal solution and has worse generalization than a simple SGD with momentum(0.9). This algorithm uses the maximum of past squared gradients rather than the exponential average to update the parameters.
  - + more optimizations algorithms that were proposed over time.

Essentially the optimizer to be used is a hyperparameter for our neural network, moreover, the parameters of each optimizer should also be tuned for better performance, but making experiments for all of these parameters is very costly in time so the path I am gonna follow is make baseline experiments with **Adam** and **Nadam** and then try to make some tests using **SGD** using different learning rates with momentum and give a try on AMSGrand which is not very widely used yet.

Early stopping

A good practice when training neural networks is to use EarlyStopping, the goal of this is to define a limit to stop the epochs for a model and keep the best model. Generally, a neural network of enough capacity has the ability to overfit on the training data if enough epochs are allowed, even up to a point of 99.9% accuracy on the train set, and all of the times overfitting on training data limits the generalization of our prediction on unknown sets. Earlystoppoing will allow us to fit the model for a quite large number of epochs, and monitor for drops in validation accuracy across time and stop fitting up to the point where overfit starts i.e. when val_loss starts to drop. Moreover, because a drop in val_loss on an epoch does not essentially mean overfitting, as it may increase on consecutive epochs, I will also allow patience of **20 epochs** before stopping fit.

Reduce Learning Rate on Plateau

Another good practice is to use ReduceLROnPlateue which helps us detecting when learning rate stagnates for a number of **10 epochs**, then we will reduce the learning rate by a **factor of 0.1,** cooldown 2 for up to a min learning rate of **0**

I will use **early stopping** and **learning rate reduction** on all my experiments to save some computational time, I ‘ll try to fit for a large number of **epochs (300).** I don’t expect that this will be reached since I have early stopping but if it does, then I ‘ll need to increase the number of epochs more, either way, will find what needs to be done from the history plots.

Multilayer Perceptron (MLP)

Baseline

We will start with a very simple MLP with no hidden layers at all, to have as a baseline. Our input is samples of images with (28, 28) pixel dimensions each, which will be used to fit the MLP.

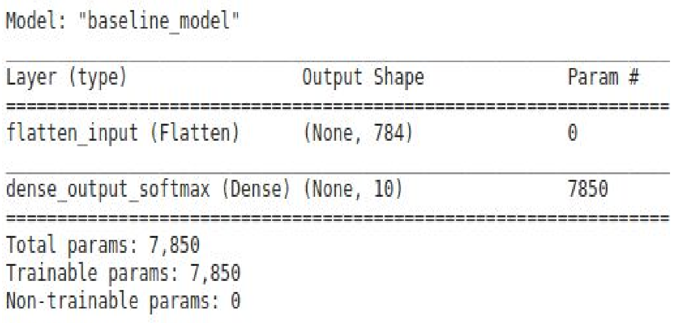
At the very least we ‘ll need an **Input layer of shape (28, 28)** and a **Flatten layer** that will get each row of **(1, 28)** pixels of the image and concatenate them to create a **(1, 768) vector**. (This could have been done as a preprocessing step to prepare the vectors for each sample but essentially it’s the same). Furthermore, unlike more sophisticated layers like the one used in CNNs, a simple MLP with Dense layers and Dropouts would not care about the actual image structure i.e. each row of pixels to be in the correct order, so this is the best that can be done.

As an output layer, we will have a **Dense output** with **softmax** activation, since essentially we want the output to be the probability (summed to 1) of each sample to belong to one of the 10 categories that I have.

The structure of the model can be seen on the right. It’s a very simple model with only

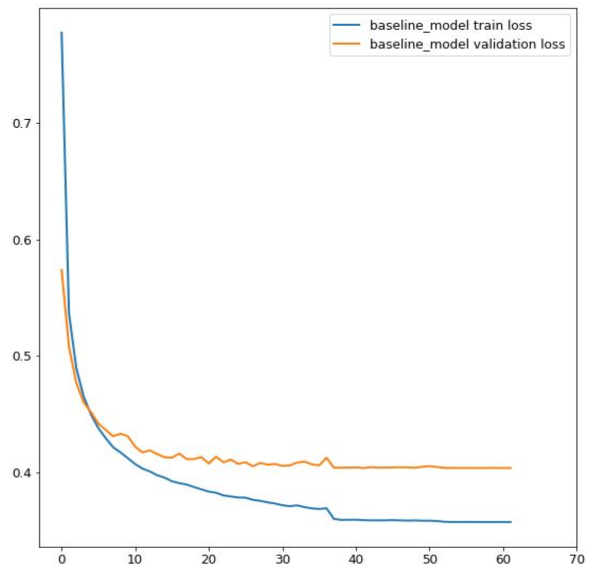
- [10, 784] for the weight matrix of input to output
- [1, 10] for the output layer

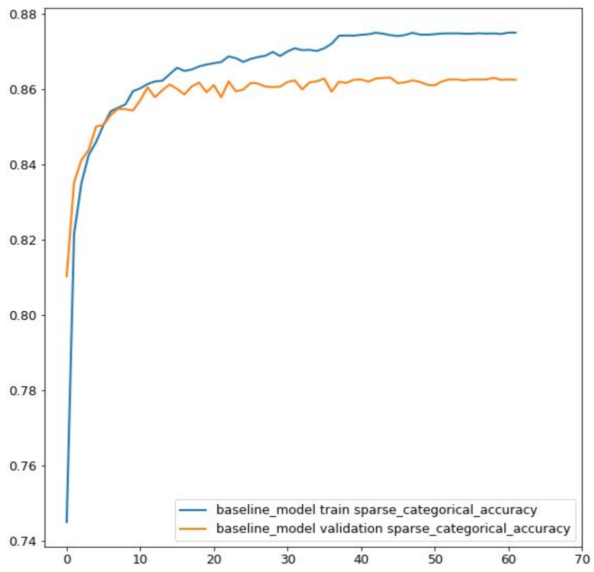
Totaling to 7850 trainable params.



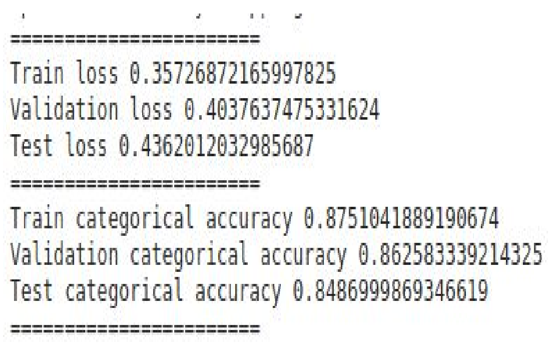
```
Model: "baseline_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_input (Flatten)      (None, 784)               0
_________________________________________________________________
dense_output_softmax (Dense) (None, 10)                7850
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
```

When training for 300 epochs





Remarks



```
=======================
Train loss 0.35726872165997825
Validation loss 0.4037637475331624
Test loss 0.4362012032985687
=======================
Train categorical accuracy 0.8751041889190674
Validation categorical accuracy 0.862583339214325
Test categorical accuracy 0.8486999869346619
=======================
```

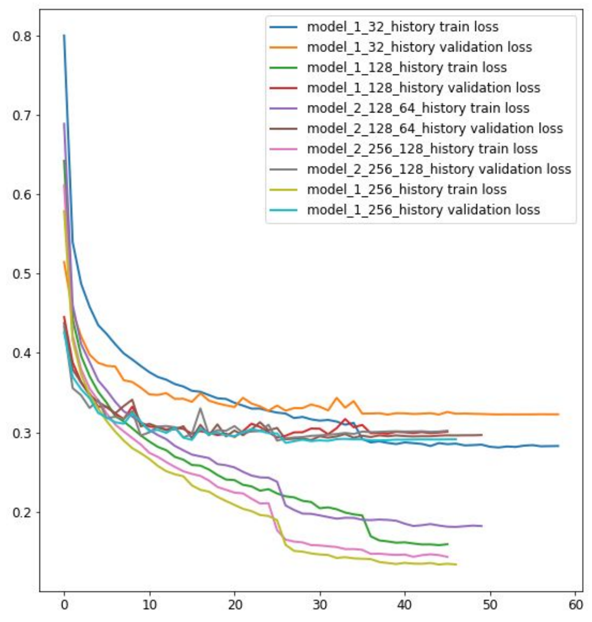
- max accuracy on train set is around 0.90, this could mean that to model the data we would need a bigger model,
- Fitting stops after 60 epochs, this essentially means that the model is overfitting on the train data

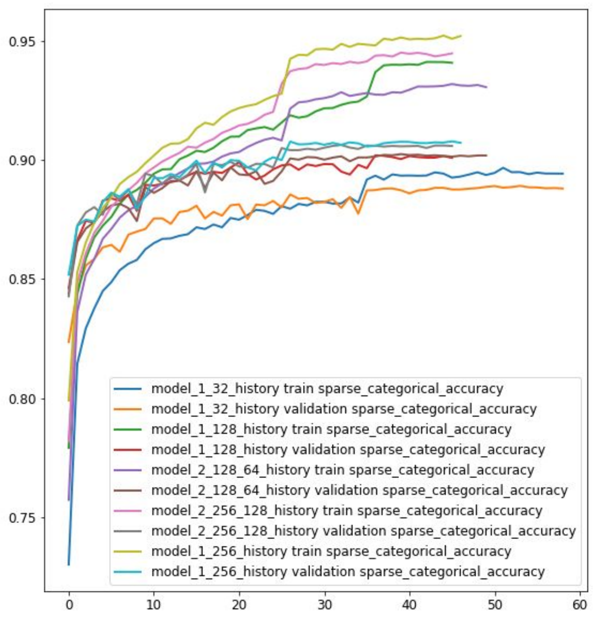
and generalization is dropping so fitting will stop restoring the best weights.
- For such a simple model 0.8625 categorical accuracy is not so bad, but this also means that we can achieve better prediction with a better model.

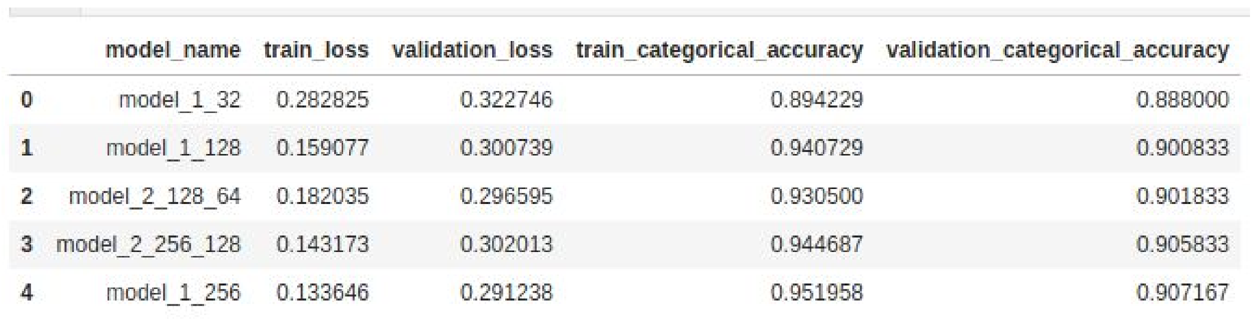
MLP with more capacity

So now I would like to try with bigger models to see how the graph changes, for each dense layer I ‘ll also add a Dropout of 0.2. I ‘ll also try other dropout values in later steps with Talos.

1) 1 Dense layer with 32 units
2) 1 Dense layer with 128 units
3) 1 Dense layer with 256 units
4) 2 Dense layers with 128 units and 64 respectively
5) 2 Dense layers with 256 units and 128 respectively







	model_name	train_loss	validation_loss	train_categorical_accuracy	validation_categorical_accuracy
0	model_1_32	0.282825	0.322746	0.894229	0.888000
1	model_1_128	0.159077	0.300739	0.940729	0.900833
2	model_2_128_64	0.182035	0.296595	0.930500	0.901833
3	model_2_256_128	0.143173	0.302013	0.944687	0.905833
4	model_1_256	0.133646	0.291238	0.951958	0.907167

After many manual experiments, with some architectures the results of some can be seen above, I can see that the models with 1 dense layer in most cases outperform models with more layers in respect to **validation loss**. At the moment the highest validation accuracy that we achieved is with the 1 layer and 256 neurons model **0.9071** which also has the smallest loss of **0.2912**. There are some important things to consider here, there is a general rule of thumb when selecting the number of neurons for each layer in MLP, we can start with 1 layer and the number of neurons should be a number between the 768 input size and the 10 output size. Since the tuning grid for these parameters is infinite we need an automated way to perform a grid search of the tunable parameters of an MLP. Since now we have an overview of the loss per number of layers and units.

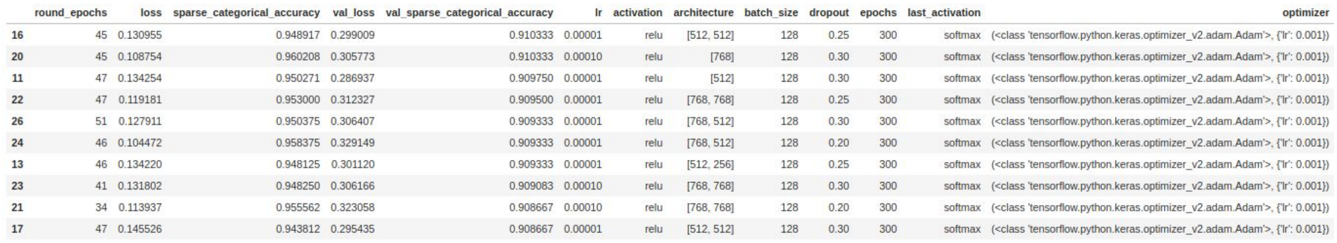
MLP Talos

Model Architecture

After having an overview of the effect, of the number of layers and the size of each layer on the loss with the above manual cases I will use Talos to perform a grid search for the architecture of the MLP. This is more like a brute force method to verify the best architecture.

```
{
    'architecture':[
        [256, 256], [384], [384, 256, 128], [512], [512, 256], [512, 512], [768], [768, 768], [768, 512], [768, 512, 256]
    ],
    'batch_size': [128],
    'epochs': [300],
    'dropout': [0.2, 0.25, 0.3],
    'optimizer': [
        (tf.keras.optimizers.Adam, {"lr": 0.001}),
    ],
    'activation':['relu'],
    'last_activation': ['softmax']
}
```

From a total of 30 experiments below, we can see the top 10 performing ones, as we have seen from manual experiments the results are expected, always 1 layer outperform 2 layers because MLP overfits the train data quickly and cannot generalize on the validation set.



	round_epochs	loss	sparse_categorical_accuracy	val_loss	val_sparse_categorical_accuracy	lr	activation	architecture	batch_size	dropout	epochs	last_activation	optimizer
16	45	0.130955	0.948917	0.299009	0.910333	0.00001	relu	[512, 512]	128	0.25	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
20	45	0.108754	0.960208	0.305773	0.910333	0.00010	relu	[768]	128	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
11	47	0.134254	0.950271	0.286937	0.909750	0.00001	relu	[512]	128	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
22	47	0.119181	0.953000	0.312327	0.909500	0.00001	relu	[768, 768]	128	0.25	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
26	51	0.127911	0.950375	0.306407	0.909333	0.00001	relu	[768, 512]	128	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
24	46	0.104472	0.958375	0.329149	0.909333	0.00001	relu	[768, 512]	128	0.20	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
13	46	0.134220	0.948125	0.301120	0.909333	0.00001	relu	[512, 256]	128	0.25	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
23	41	0.131802	0.948250	0.306166	0.909083	0.00010	relu	[768, 768]	128	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
21	34	0.113937	0.955562	0.323058	0.908667	0.00010	relu	[768, 768]	128	0.20	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
17	47	0.145526	0.943812	0.295435	0.908667	0.00001	relu	[512, 512]	128	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})

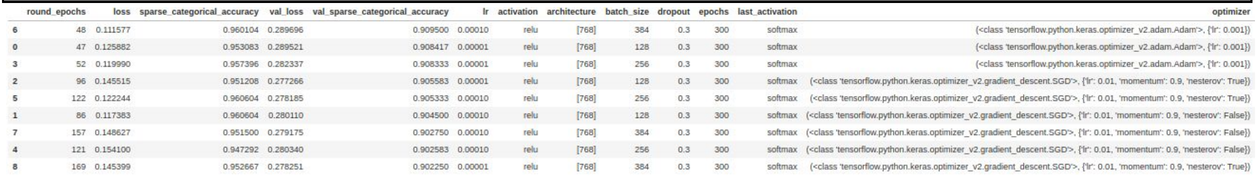
[the complete experiment set can be found on the repo]

So the best performing architecture is a **1 layer of 768** units and a **Dropout** of **0.3** that achieved **0.91033** categorical accuracy on the validation set and has less complexity than the [512, 512] that achieve the accuracy.

Optimizers

After having a good architecture with Adam I would like to test other optimizers with the same architecture. Moreover, I ‘ll use a batch size of 128, 256, and 512 since it may have an effect on the learning rate of the model, the memory of the GPU can handle higher batch sizes too, but higher batch sizes can have a detrimental effect on the generalization of the model.

```
{
    'architecture':[[768]],
    'batch_size': [128, 256, 384, 512],
    'epochs': [300],
    'dropout': [0.3],
    'optimizer': [
        (tf.keras.optimizers.Adam, {"lr": 0.001}),
        (tf.keras.optimizers.Nadam, {"lr": 0.001}),
        (tf.keras.optimizers.SGD, {"lr": 0.01, "momentum": 0.9, "nesterov": False}),
        (tf.keras.optimizers.SGD, {"lr": 0.01, "momentum": 0.9, "nesterov": True})
    ],
    'activation':['relu'],
    'last_activation': ['softmax']
}
```



	round_epochs	loss	sparse_categorical_accuracy	val_loss	val_sparse_categorical_accuracy	lr	activation	architecture	batch_size	dropout	epochs	last_activation	optimizer
6	48	0.111577	0.960104	0.289696	0.909500	0.00010	relu	[768]	384	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
0	47	0.125882	0.953083	0.289521	0.908417	0.00001	relu	[768]	128	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
3	52	0.119990	0.957396	0.282337	0.908333	0.00001	relu	[768]	256	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
2	96	0.145515	0.951208	0.277266	0.905583	0.00001	relu	[768]	128	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.gradient_descent.SGD'>, {'lr': 0.01, 'momentum': 0.9, 'nesterov': True})
5	122	0.122244	0.960604	0.278185	0.905333	0.00010	relu	[768]	256	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.gradient_descent.SGD'>, {'lr': 0.01, 'momentum': 0.9, 'nesterov': True})
1	86	0.117383	0.960604	0.280110	0.904500	0.00010	relu	[768]	128	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.gradient_descent.SGD'>, {'lr': 0.01, 'momentum': 0.9, 'nesterov': False})
7	157	0.148627	0.951500	0.279175	0.902750	0.00010	relu	[768]	384	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.gradient_descent.SGD'>, {'lr': 0.01, 'momentum': 0.9, 'nesterov': False})
4	121	0.154100	0.947292	0.280340	0.902583	0.00010	relu	[768]	256	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.gradient_descent.SGD'>, {'lr': 0.01, 'momentum': 0.9, 'nesterov': False})
8	169	0.145399	0.952667	0.278251	0.902250	0.00001	relu	[768]	384	0.3	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.gradient_descent.SGD'>, {'lr': 0.01, 'momentum': 0.9, 'nesterov': True})

[the complete experiment set can be found on the repo]

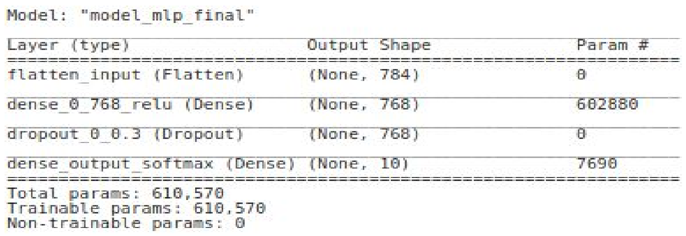
From a total of 31 experiments, Adam had the best performance with batches of 256. In all cases, SGD with momentum either with Nesterov or not performed worse than Adam and Nadam.

Summary

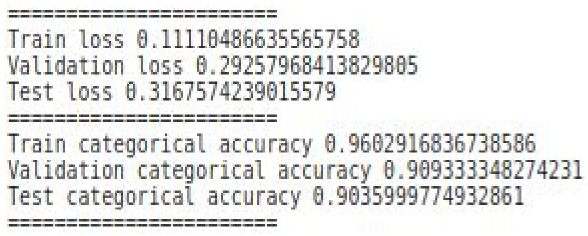
After adding more layers or more units per layer does not improve the validation loss since the model overfits quickly on the train data and validation loss starts to drops fast. So the harder task is to find an MLP deep enough to have a good train on more epochs without losing the generalization ability of our model.

Evaluation

After finding the best parameters for the MLP I am going to train the final model and get the evaluation results on the test set.



```
Model: "model_mlp_final"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_input (Flatten)      (None, 784)               0
_________________________________________________________________
dense_0_768_relu (Dense)     (None, 768)               602880
_________________________________________________________________
dropout_0_0.3 (Dropout)      (None, 768)               0
_________________________________________________________________
dense_output_softmax (Dense) (None, 10)                7690
=================================================================
Total params: 610,570
Trainable params: 610,570
Non-trainable params: 0
```



```
=======================
Train loss 0.11110486635565758
Validation loss 0.29257968413829805
Test loss 0.3167574239015579
=======================
Train categorical accuracy 0.9602916836738586
Validation categorical accuracy 0.909333348274231
Test categorical accuracy 0.9035999774932861
=======================
```

The final model got a categorical accuracy of **0.9035** on the test set which is very good for this multiclass classification problem. The saved model is attached in tf format on the repository under **models/model_mlp_final** for further use.

Convolutional neural network (CNN)

Adding convolutional layers has good results on pattern detection and can enhance our model to detect objects in an image. The main feature is the addition of filter masks of a size e.g 3x3 that pan across the pixels of input and subsample the image. Each convolution layer will detect from simpler to more complex combinations.

Again we will need an **Input layer of shape (28, 28, 1)** the last dimension is the channel which is 1 since we have to process grayscale images and there will be no Flatten layer since in convolution layers we care about keeping the structure of the input. And a Dense output with softmax to predict the 10 categories.

For the convolutional layers, the parameter grid here is even bigger

- filters: The number of filters that will be trained in each layer to detect a pattern, i.e features maps
- kernel_size (filter_size) (3x3) will be used as it has become a standard since it performs better, also my image is not big enough. (Probably I will try a (5x5) filter too for validation
- Stride: I don't want to downsample the image so (1,1)
- Dilation Rate: I don't want to increase the rate since the images are pretty small, also I can afford to use a 5x5 kernel size instead.
- Padding: I want to pad pixels in any case with zeros instead of dropping so 'same'

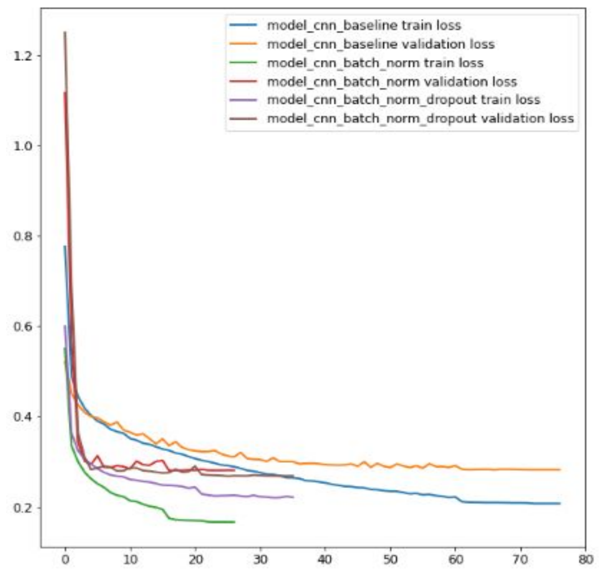
In general training, CNN takes long even on GPU, so I 'll follow the best practice on similar tasks and build up on top of them. For example, in most cases,

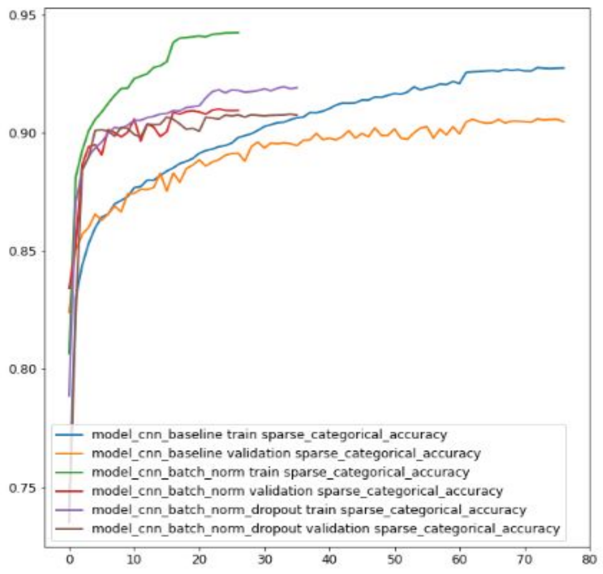
1. smaller kernels are better than larger ones.
2. the filter size of the first layer is tunable but there is a relation with the type of images that we expect to handle. In our case, the images are not much complex so I'll start with **8 filters** and then try with more filters. If I was to start with more filters like 32 or 64 it would make fitting slower and probably will not be needed.
3. The filter number in subsequent convolutional layers is the double of the previous so that it can detect more complex features.
4. After every convolution layer, I will use **MaxPolling with size 2** with **(2,2) stride** and padding **`same`** to reduce the model parameters so that the next layers take more attention to the details.
5. After each Convolution layer, I will use BatchNormalization before feeding to the next output since having normalized input between (0,1) provides faster and probably better results on neural networks.

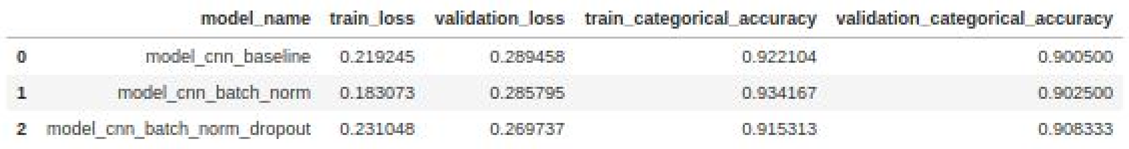
Baseline

I have trained 3 models

1. Simple CNN with 8 filters
2. Simple CNN with 8 filters and BatchNormalization after each Convolutional layer
3. Simple CNN with 8 filters and BatchNormalization and Dropout after each Conv Layer





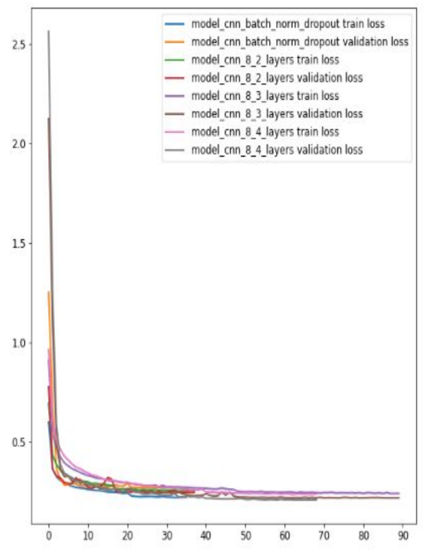


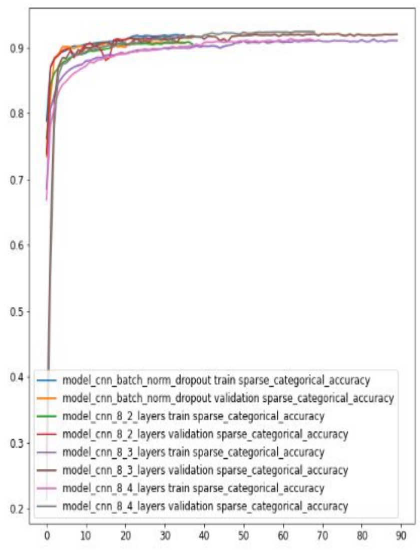
	model_name	train_loss	validation_loss	train_categorical_accuracy	validation_categorical_accuracy
0	model_cnn_baseline	0.219245	0.289458	0.922104	0.900500
1	model_cnn_batch_norm	0.183073	0.285795	0.934167	0.902500
2	model_cnn_batch_norm_dropout	0.231048	0.269737	0.915313	0.908333

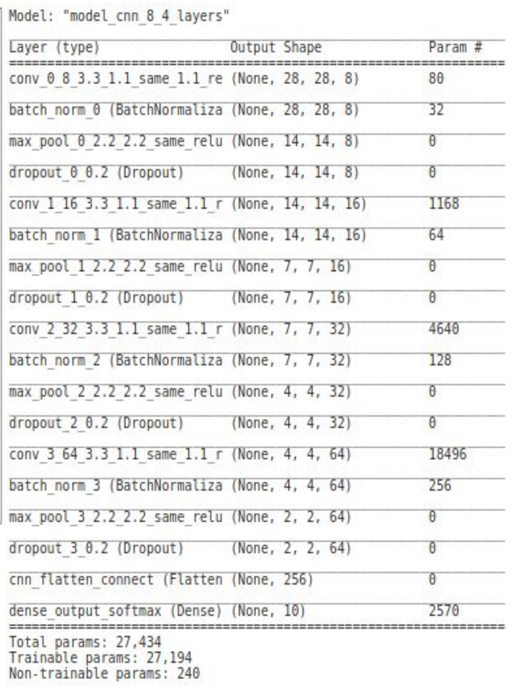
I see that the simple baseline CNN achieves 0.905 accuracy on the validation set in 76 epochs, around the same as our best MLP which is really good and shows that there is much room for improvement. The train and validation curves are not very close which is an indication of overfitting so we will need to add a dropout after each convolutional layer. BatchNormalization helped us increase the validation accuracy and **the addition of dropout** increased it to 0.9083 but also on the curves I see that there is less overfitting as train and validation curves follow the same path. So the addition of BatchNormalization helped us fit the model faster and Dropout increased the generalization.

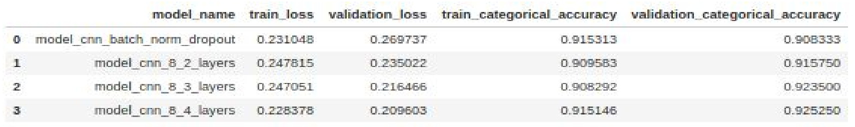
Convolutional Layers

Now that I have a baseline I will try to add more convolution layers which essentially is the power of CNNs to allow my model to detect more complex features and then try to optimize the number of filters. In general, for such tasks, the number of convolutional layers is not so easy to find. From preliminary results:






```
Model: "model_cnn_8_4_layers"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_0_8_3.3_1.1_same_1.1_re (None, 28, 28, 8)         80
_________________________________________________________________
batch_norm_0 (BatchNormaliza (None, 28, 28, 8)         32
_________________________________________________________________
max_pool_0_2.2_2.2_same_relu (None, 14, 14, 8)         0
_________________________________________________________________
dropout_0_0.2 (Dropout)      (None, 14, 14, 8)         0
_________________________________________________________________
conv_1_16_3.3_1.1_same_1.1_r (None, 14, 14, 16)        1168
_________________________________________________________________
batch_norm_1 (BatchNormaliza (None, 14, 14, 16)        64
_________________________________________________________________
max_pool_1_2.2_2.2_same_relu (None, 7, 7, 16)          0
_________________________________________________________________
dropout_1_0.2 (Dropout)      (None, 7, 7, 16)          0
_________________________________________________________________
conv_2_32_3.3_1.1_same_1.1_r (None, 7, 7, 32)          4640
_________________________________________________________________
batch_norm_2 (BatchNormaliza (None, 7, 7, 32)          128
_________________________________________________________________
max_pool_2_2.2_2.2_same_relu (None, 4, 4, 32)          0
_________________________________________________________________
dropout_2_0.2 (Dropout)      (None, 4, 4, 32)          0
_________________________________________________________________
conv_3_64_3.3_1.1_same_1.1_r (None, 4, 4, 64)          18496
_________________________________________________________________
batch_norm_3 (BatchNormaliza (None, 4, 4, 64)          256
_________________________________________________________________
max_pool_3_2.2_2.2_same_relu (None, 2, 2, 64)          0
_________________________________________________________________
dropout_3_0.2 (Dropout)      (None, 2, 2, 64)          0
_________________________________________________________________
cnn_flatten_connect (Flatten (None, 256)               0
_________________________________________________________________
dense_output_softmax (Dense) (None, 10)                2570
=================================================================
Total params: 27,434
Trainable params: 27,194
Non-trainable params: 240
```



	model_name	train_loss	validation_loss	train_categorical_accuracy	validation_categorical_accuracy
0	model_cnn_batch_norm_dropout	0.231048	0.269737	0.915313	0.908333
1	model_cnn_8_2_layers	0.247815	0.235022	0.909583	0.915750
2	model_cnn_8_3_layers	0.247051	0.216466	0.908292	0.923500
3	model_cnn_8_4_layers	0.228378	0.209603	0.915146	0.925250

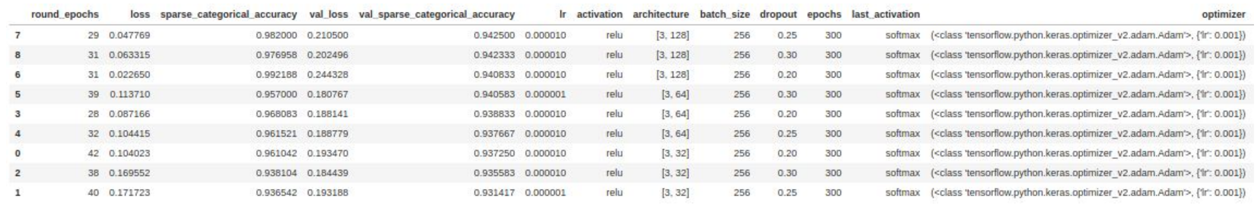
Each layer doubles the previous layers so on the above table the 4th CNN with 8 starting filters will have **4 layers** with **8, 16, 32, 64** filters respectively. Based on the results I see that even with 8 starting filters, adding more convolutional layers decreases validation loss and increases the accuracy. Moreover, the curves of the current structure look pretty good as the training and validation look the same more or less. The problem with adding more layers is that it adds more complexity and requires much more time to finish. So I will start with 3 layers and try to find a good number for the filters.

CNN Talos

After having an overview of the effect, of the number of layers on the loss with the above manual cases I will use Talos to perform a grid search for the number of filters of CNN.

```
{
    'architecture':[  [3, 32], [4, 32], [3, 64], [4, 64], [3, 128], [4, 64],  ], -> The format is [no of layers, first layer size]
    'batch_size': [256],
    'epochs': [300],
    'dropout': [0.2, 0.25, 0.3],
    'optimizer': [
        (tf.keras.optimizers.Adam, {"lr": 0.001}),
    ],
    'activation':['relu'],
    'last_activation': ['softmax']
}
```

From a total of 9 experiments below, we can see the performance of each one, as we can see the deeper the more filters we add the performance increases, but also the training time increases as well. Moreover, the performance gain is not so big.



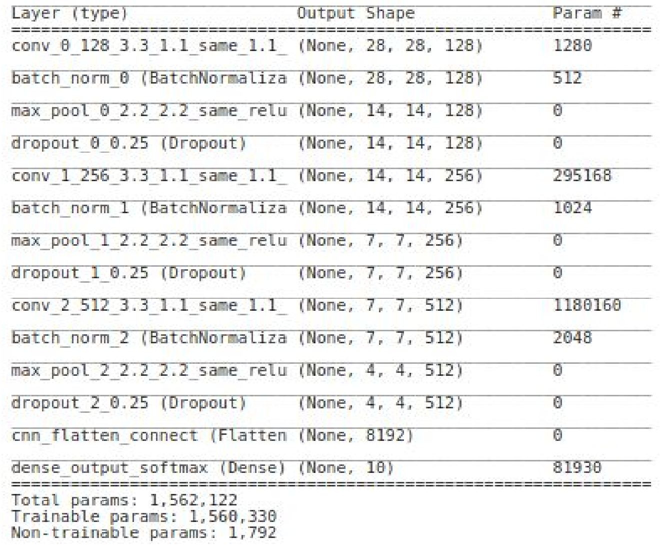
	round_epochs	loss	sparse_categorical_accuracy	val_loss	val_sparse_categorical_accuracy	lr	activation	architecture	batch_size	dropout	epochs	last_activation	optimizer
7	29	0.047769	0.982000	0.210500	0.942500	0.000010	relu	[3, 128]	256	0.25	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
8	31	0.063315	0.976958	0.202496	0.942333	0.000010	relu	[3, 128]	256	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
6	31	0.022650	0.992188	0.244328	0.940833	0.000010	relu	[3, 128]	256	0.20	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
5	39	0.113710	0.957000	0.180767	0.940583	0.000001	relu	[3, 64]	256	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
3	28	0.087166	0.968083	0.188141	0.938833	0.000010	relu	[3, 64]	256	0.20	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
4	32	0.104415	0.961521	0.188779	0.937667	0.000010	relu	[3, 64]	256	0.25	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
0	42	0.104023	0.961042	0.193470	0.937250	0.000010	relu	[3, 32]	256	0.20	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
2	38	0.169552	0.938104	0.184439	0.935583	0.000010	relu	[3, 32]	256	0.30	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})
1	40	0.171723	0.936542	0.193188	0.931417	0.000001	relu	[3, 32]	256	0.25	300	softmax	(<class 'tensorflow.python.keras.optimizer_v2.adam.Adam'>, {'lr': 0.001})

[the complete experiment set can be found on the repo]

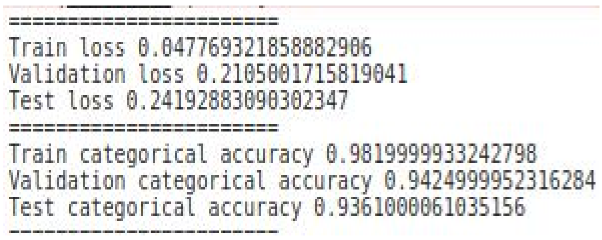
So the best performing architecture is a **3 layer CNN** with 128, 256, and 512 filter size respectively, and a **Dropout** of **0.25** that achieved **0.9425** categorical accuracy on the validation set.

Evaluation

After finding the best CNN architecture I am going to train the final model and get the evaluation results on the test set.



```
Layer (type)                 Output Shape              Param #
=================================================================
conv_0_128_3.3_1.1_same_1.1_ (None, 28, 28, 128)       1280
_________________________________________________________________
batch_norm_0 (BatchNormaliza (None, 28, 28, 128)       512
_________________________________________________________________
max_pool_0_2.2_2.2_same_relu (None, 14, 14, 128)       0
_________________________________________________________________
dropout_0_0.25 (Dropout)     (None, 14, 14, 128)       0
_________________________________________________________________
conv_1_256_3.3_1.1_same_1.1_ (None, 14, 14, 256)       295168
_________________________________________________________________
batch_norm_1 (BatchNormaliza (None, 14, 14, 256)       1024
_________________________________________________________________
max_pool_1_2.2_2.2_same_relu (None, 7, 7, 256)         0
_________________________________________________________________
dropout_1_0.25 (Dropout)     (None, 7, 7, 256)         0
_________________________________________________________________
conv_2_512_3.3_1.1_same_1.1_ (None, 7, 7, 512)         1180160
_________________________________________________________________
batch_norm_2 (BatchNormaliza (None, 7, 7, 512)         2048
_________________________________________________________________
max_pool_2_2.2_2.2_same_relu (None, 4, 4, 512)         0
_________________________________________________________________
dropout_2_0.25 (Dropout)     (None, 4, 4, 512)         0
_________________________________________________________________
cnn_flatten_connect (Flatten (None, 8192)              0
_________________________________________________________________
dense_output_softmax (Dense) (None, 10)                81930
=================================================================
Total params: 1,562,122
Trainable params: 1,560,330
Non-trainable params: 1,792
```



```
========================
Train loss 0.047769321858882906
Validation loss 0.2105001715819041
Test loss 0.24192883090302347
========================
Train categorical accuracy 0.9819999933242798
Validation categorical accuracy 0.9424999952316284
Test categorical accuracy 0.9361000061035156
========================
```

The final model got a mean categorical accuracy of **0.9384** with 5 KFold of the test set which is very good for this multiclass classification problem. The saved model is attached in tf format on the repository under **models/model_cnn_final** for further use.

Summary

For both tasks, the biggest challenge is to find an architecture that is big enough to have a good validation accuracy but not so big to overfit the train set early. This can be partly solved by using early stopping and Dropout on the model which helps but still adding making deeper models needs more computational power and will not always have better results. Moreover, **Talos** helped to automate some training tasks but the parameter grid is infinite for such tasks and especially in the case of CNN it takes much time to fit a model, so it’s almost impossible to tune the model perfectly. It helps to follow best practices at least for some parts like the layer and filter size. To finish with, CNN achieves better results on this task and could achieve some small improvements on the score if we use ImageAugmentation to train the model with more examples of the same image.

Additional Resources

The code used for the experiments along with the finalized models MLP and CNN can be found in the below resources for further usage.

- Github Repository: link
- Google Drive: link

- **./lib/DeepLearning** folder contains the code that I used to create the MLP and CNN models.
- **./notebooks/Deep Learning Assignment 1.ipynb** has all the experiments and visualizations that were used to create this report.
- **./model** has the trained tf models.