MLPs and CNN for image classification

Koutsomarkos Alexandros[1] and Lakkas Ioannis[2]

[1]p3352106 , [2]p3352110
Emails: akoutsomarkos@aueb.gr , ilakkas@aueb.gr

April 18, 2023

Google Colab MLP
Google Colab CNN

1 Fashion item recognition

Given an image of a fashion item, build a deep learning model that recognizes the fashion item. You must use at least 2 different architectures, one with MLPs and one with CNNs. Use the Fashion-MNIST dataset to train and evaluate your models. More information about the task and the dataset can be found at **github**. The dataset is also available from Tensorflow.

1.1 Dataset

For the purpose of the exercise we imported the Fashion-MNIST dataset from tensorflow. The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The labels are an array of integers, ranging from 0 to 9. These correspond to the class of clothing the image represents. Each image is mapped to a single label:

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

There are 60,000 images in the training set, with each image represented as 28 x 28 pixels. There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels.

1.2 Exploratory Data Analysis (EDA)

As we observe in Figure (1) all classes have equal number of data points

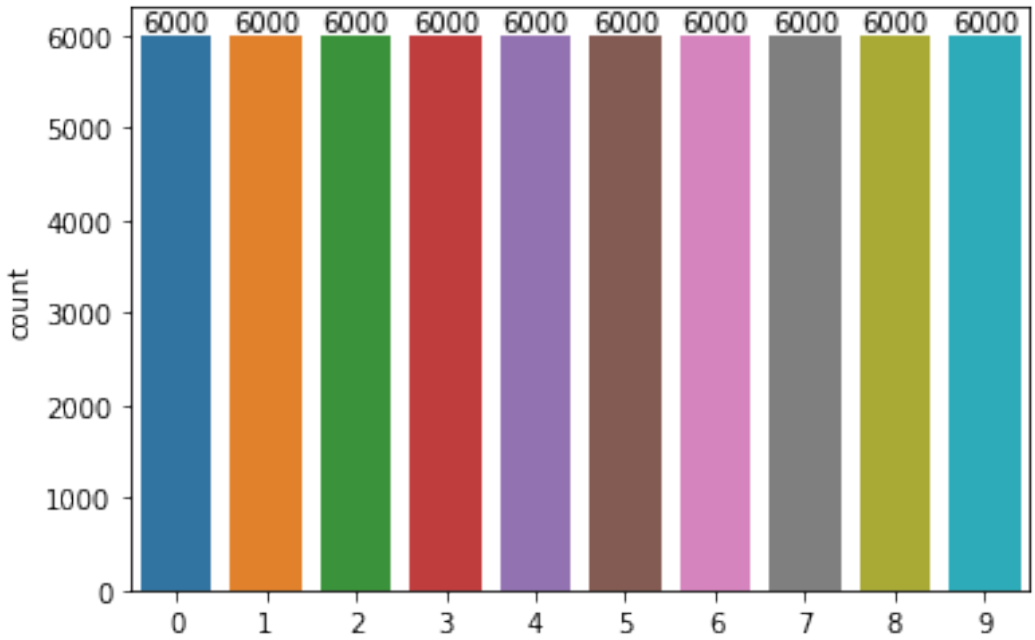

Figure 1: Class distribution for the train dataset.

1.3 Pre-Processing

Before proceeding with the development of the models, we had to do some data-prepossessing. If we inspect the training set Figure (2), we see that the pixel values fall in the range of 0 to 255. So we normalize them to [0, 1].

To verify that the data is in the correct format and that you're ready to build and train the network, let's display the first 25 images from the training set and display the class name below each image Figure (3).

We then reshape the data so as to flatten the two dimensions of the images and get them ready to be used as input in our models.

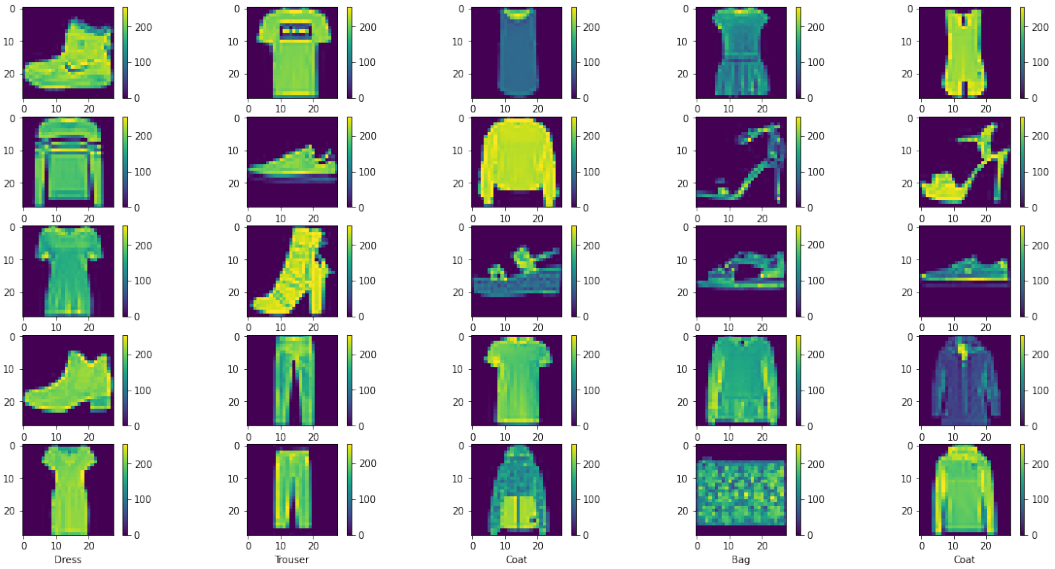

Figure 2: First 25 examples of the dataset.

Figure 3: First 25 examples of the dataset after normalization.

1.4 Train/Dev/Test split

To ensure the creation of a robust model for inference, we will reserve a portion of the training data as a development set. This will enable us to tune the hyperparameters of the model architectures effectively. Since our dataset has already been split to

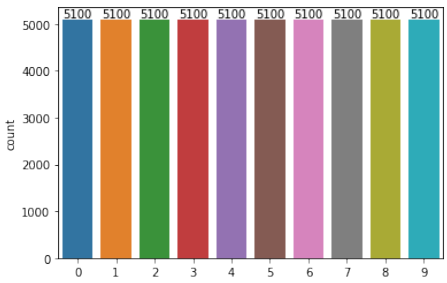

Figure 4: Class distribution for the train dataset.

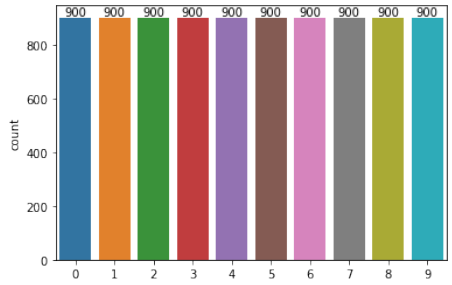

Figure 5: Class distribution for the dev dataset.

train/test sets we will reserve a portion of the training data (15%) for the development set. We shall also be extra careful to retain the uniform nature of the target class distribution so as not to introduce bias. To do this, we shall split the sets in a stratified fashion (Figure (4) and Figure (5))

1.5 MLP Models

We start by building our MLP Models which have three variations in terms of the hidden layers used. For each model size, we have variations in terms of the number of units used within the hidden layers (64 and 128) as well as the initializer (He uniform and Glorot uniform). In total we created 12 models as shown in Table (1)

size	hidden layers	hl units	initializer
small	1	64	He uniform
small	1	128	He uniform
small	1	64	Glorot uniform
small	1	128	Glorot uniform
medium	2	64	He uniform
medium	2	128	He uniform
medium	2	64	Glorot uniform
medium	2	128	Glorot uniform
large	3	64	He uniform
large	3	128	He uniform
large	3	64	Glorot uniform
large	3	128	Glorot uniform

Table 1: Model Variations

1.5.1 Model Training

For the training part we chose to use two different optimizers, namely Stochastic Gradient Descent and Adam. Both optimizers were imported form **keras** library. For all training variations we chose to configure 100 epochs and 32 as batch size, while the validation split is 10%.

SGD

Stochastic gradient descent method. For this optimizer we used the following configuration.

learning rate	momentum	nesterov
0.01	0.8	True

Table 2: SGD Configuration

Adam

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. For this optimizer we used a LearningRateSchedule that uses an inverse time decay schedule, with the following configuration.

initial learning rate	decay steps	decay rate
0.001	1593750 ((train records / batch size) * 1000)	1

Table 3: Adam Configuration

1.5.2 Training Results

In the below Figures we can see how each model performed, using the SGD and the Adam optimizer. Available are graphs for the training loss, and validation loss, the training accuracy and validation accuracy. Also there are tables for test loss and test accuracy as well as confusion matrices which were produced during the evalution of the models on the test dataset.

SGD

We observe that the initializations with the Glorot uniform initializer at least in this task seem to be better than the ones produced by the He uniform initializer. Therefore, we will continue only with this initializer during the training with the Adam optimizer.

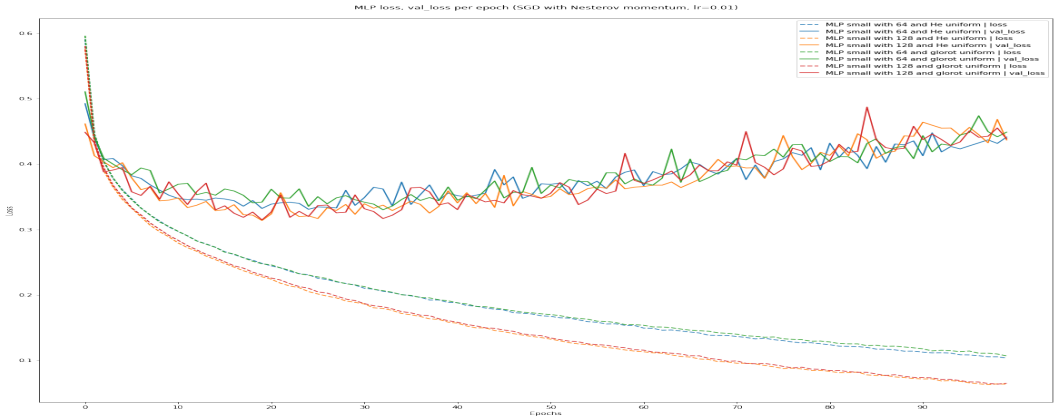

Figure 6: Loss for small MLPs with SGD.

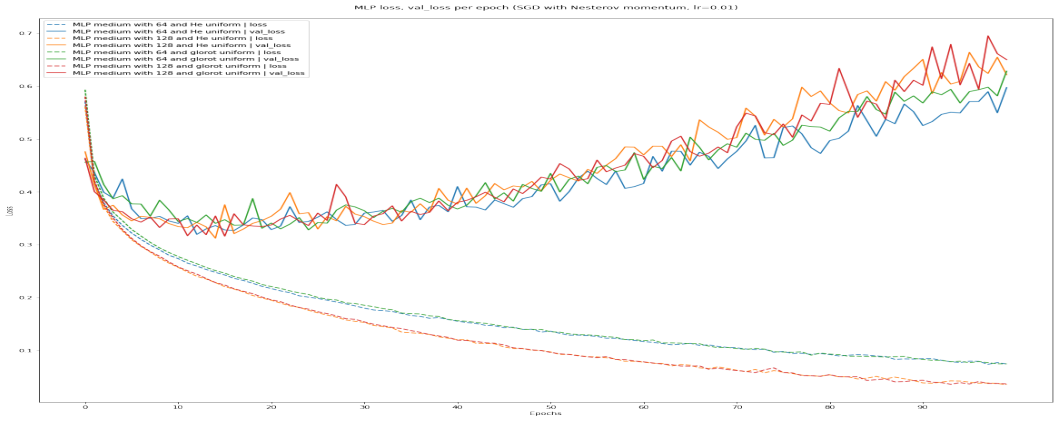

Figure 7: Loss for medium MLPs with SGD.

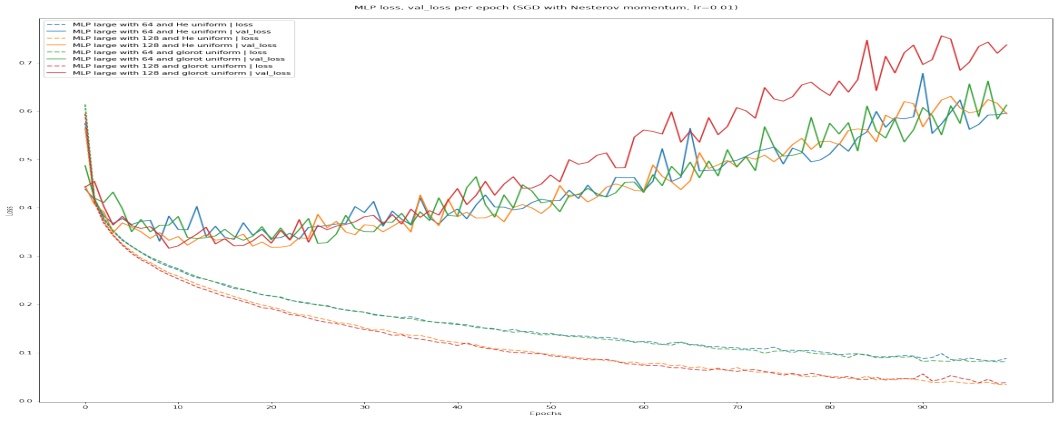

Figure 8: Loss for large MLPs with SGD.

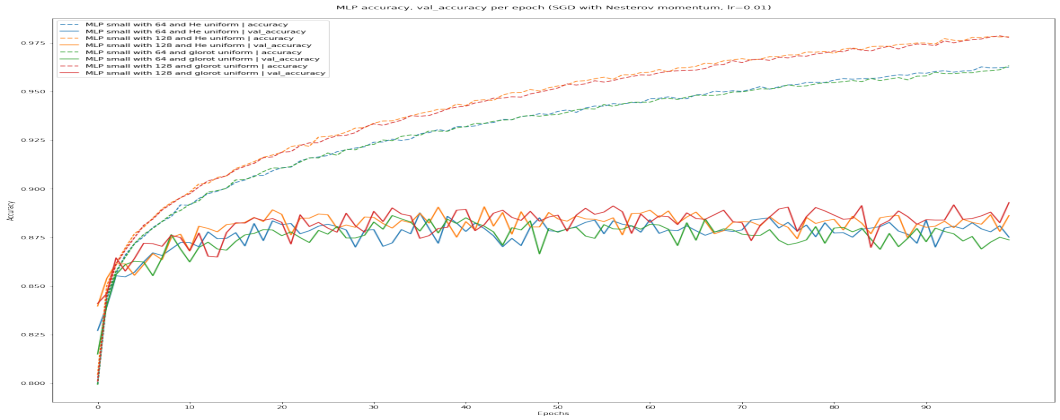

Figure 9: Accuracy for small MLPs with SGD.

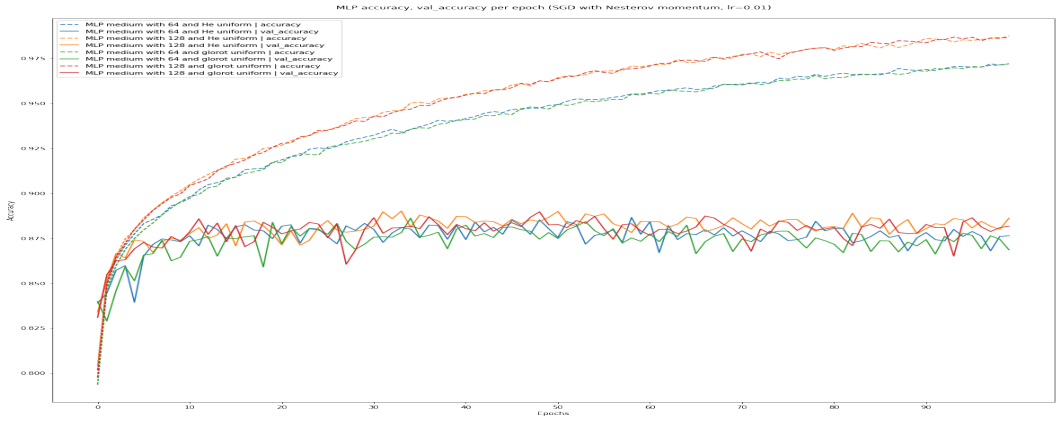

Figure 10: Accuracy for medium MLPs with SGD.

Figure 11: Accuracy for large MLPs with SGD.

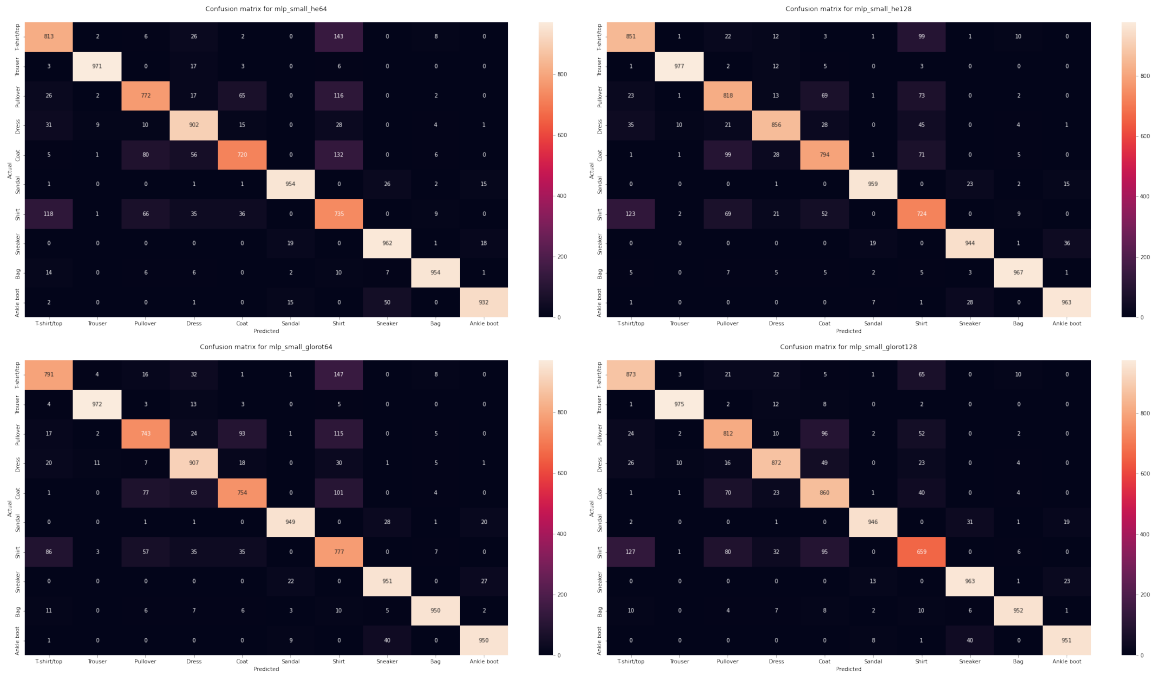

Figure 12: Confusion Matrix for small MLPs with SGD.

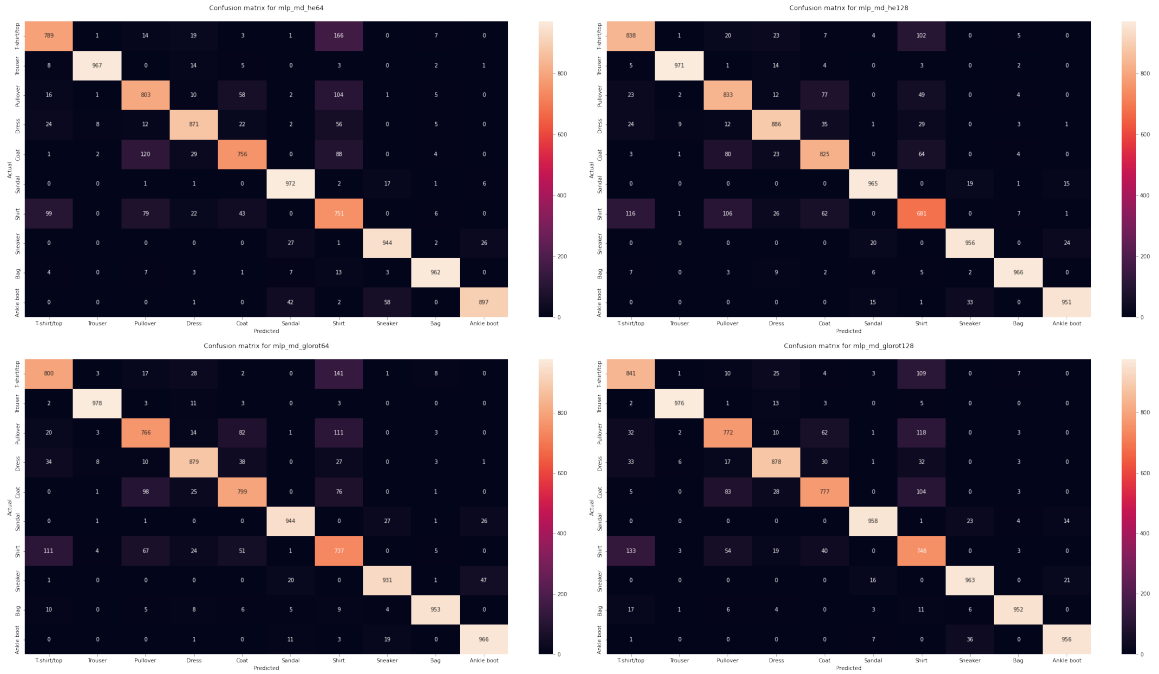

Figure 13: Confusion Matrix for medium MLPs with SGD.

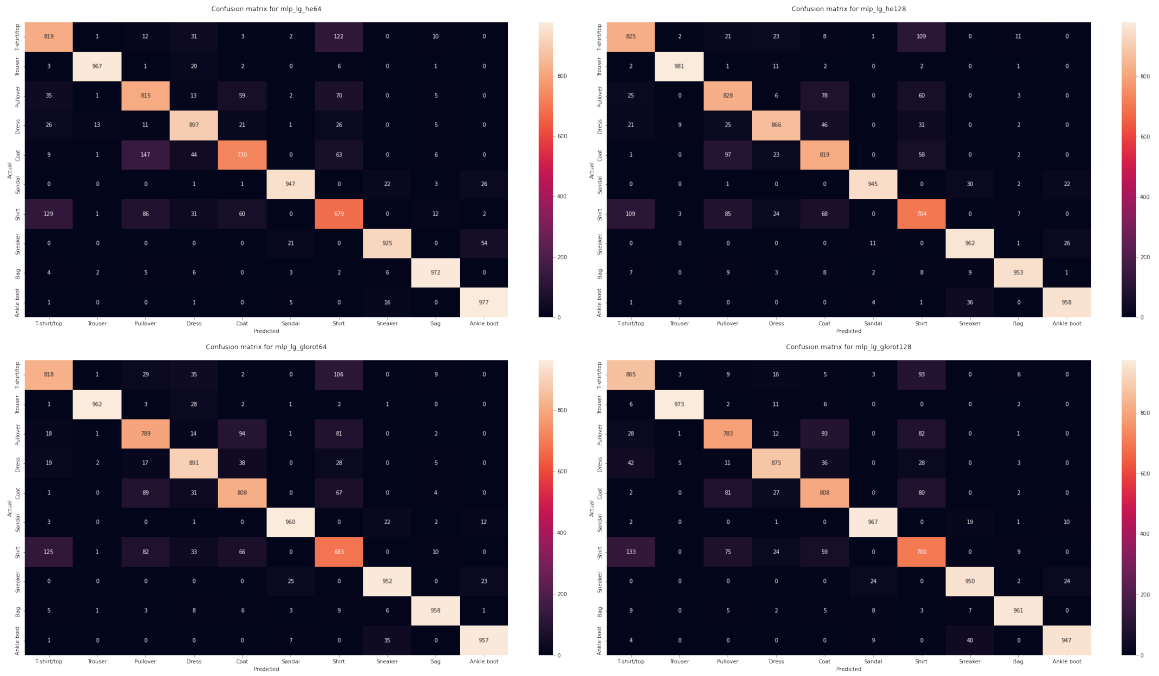

Figure 14: Confusion Matrix for large MLPs with SGD.

	test_loss	test_accuracy
MLP small with 64 and glorot uniform l	0.451170	0.8744
MLP small with 128 and glorot uniform l	0.482734	0.8863
MLP medium with 64 and glorot uniform l	0.629189	0.8753
MLP medium with 128 and glorot uniform l	0.700854	0.8821
MLP large with 64 and glorot uniform l	0.674234	0.8778
MLP large with 128 and glorot uniform l	0.810111	0.8829
MLP small with 64 and he uniform l	0.471518	0.8715
MLP small with 128 and he uniform l	0.472296	0.8853
MLP medium with 64 and he uniform l	0.658676	0.8712
MLP medium with 128 and he uniform l	0.664407	0.8872
MLP large with 64 and he uniform l	0.666049	0.8728
MLP large with 128 and he uniform l	0.665122	0.8841

Figure 15: Loss and accuracy for the MLPs with SGD within the test dataset.

Adam

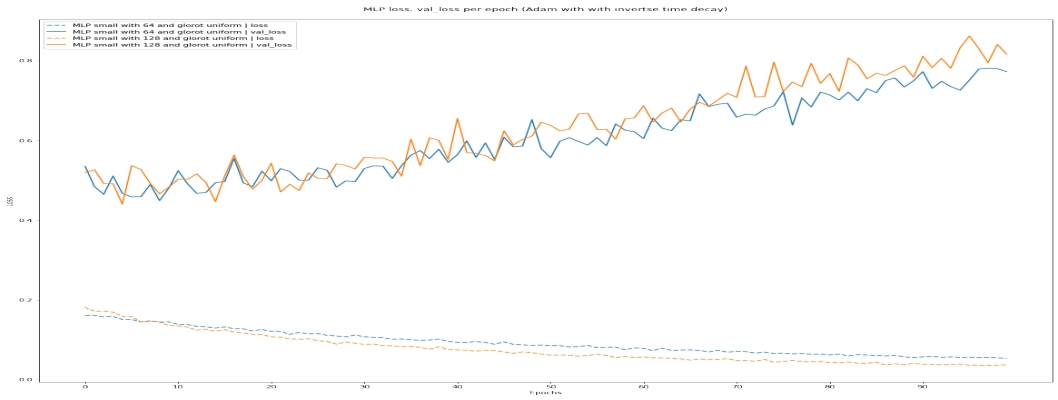

Figure 16: Loss for small MLPs with Adam.

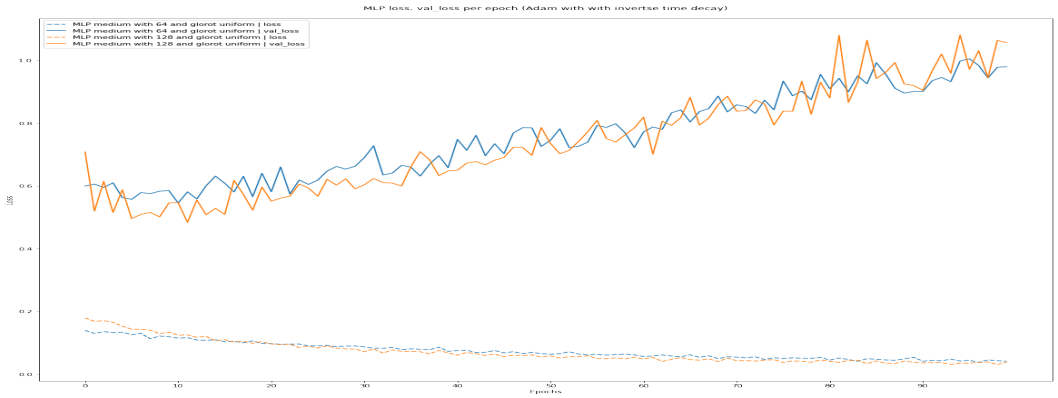

Figure 17: Loss for medium MLPs with Adam.

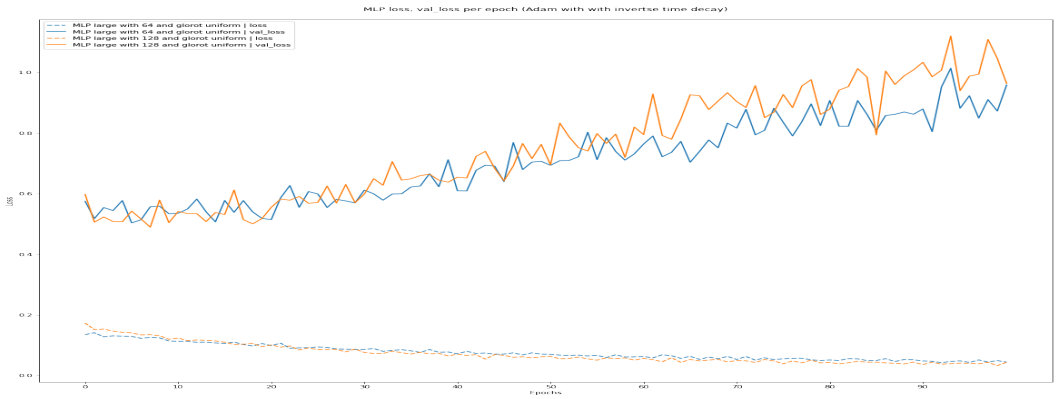

Figure 18: Loss for large MLPs with Adam.

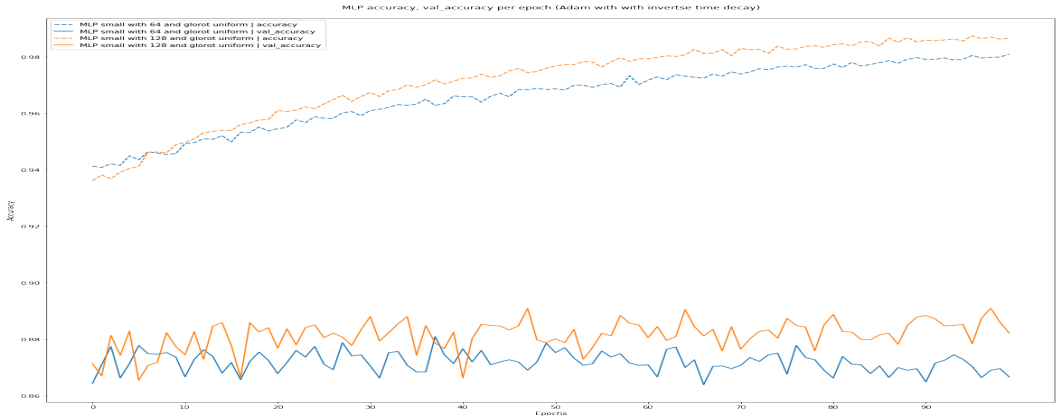

Figure 19: Accuracy for small MLPs with Adam.

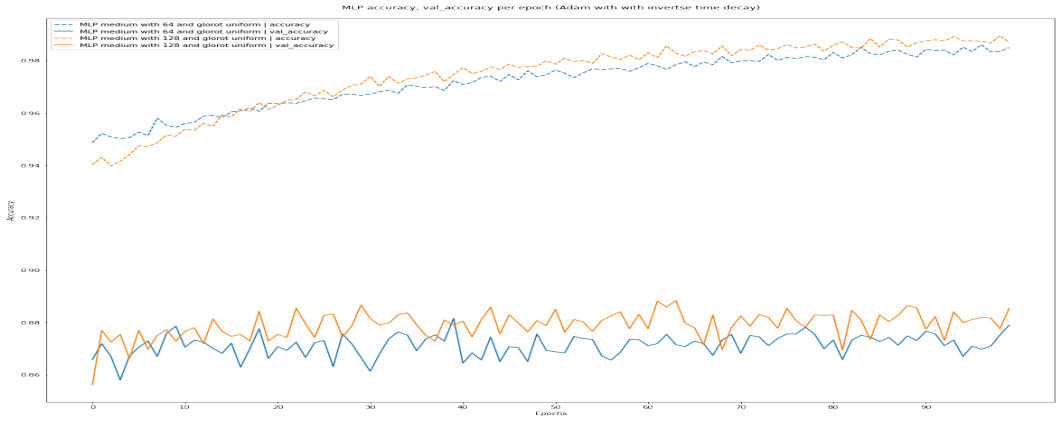

Figure 20: Accuracy for medium MLPs with Adam.

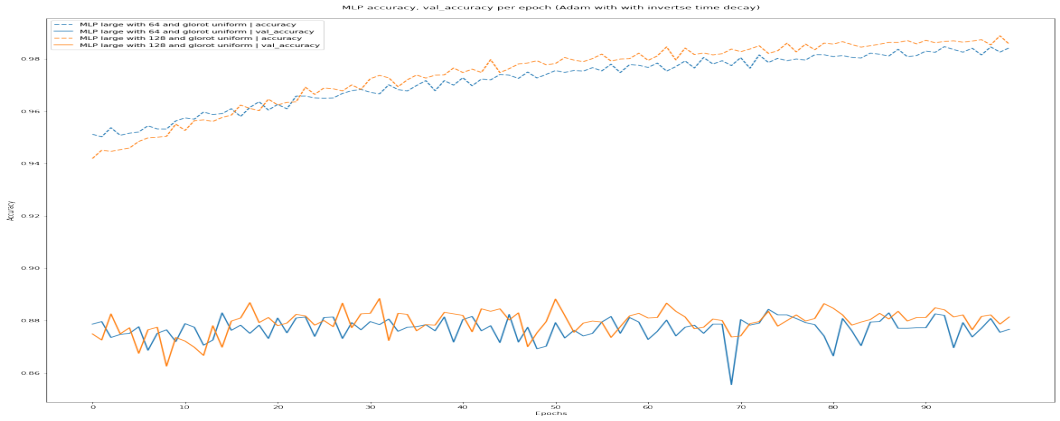

Figure 21: Accuracy for large MLPs with Adam.

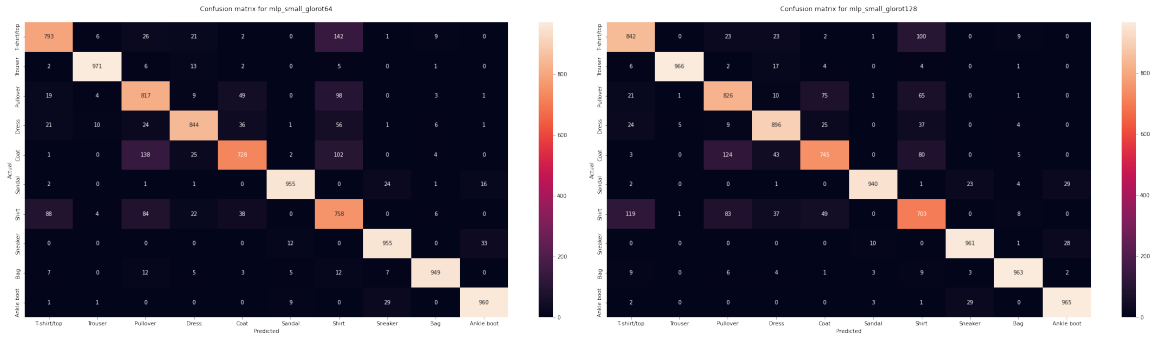

Figure 22: Confusion Matrix for small MLPs with Adam.

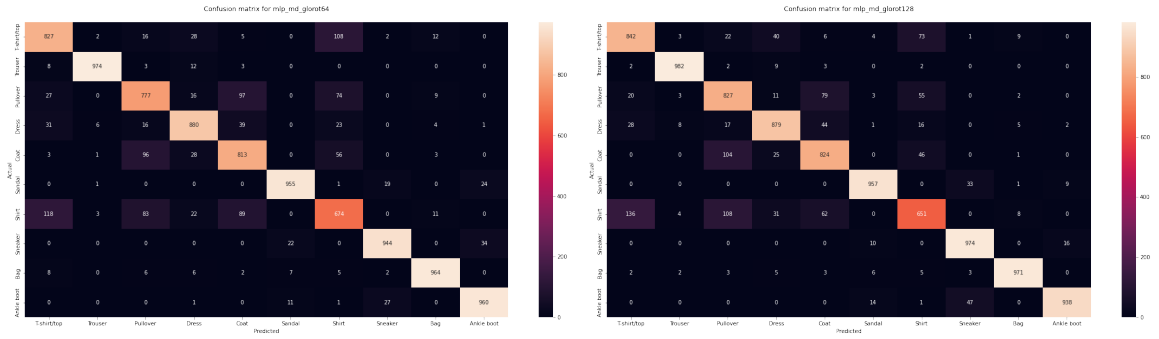

Figure 23: Confusion Matrix for medium MLPs with Adam.

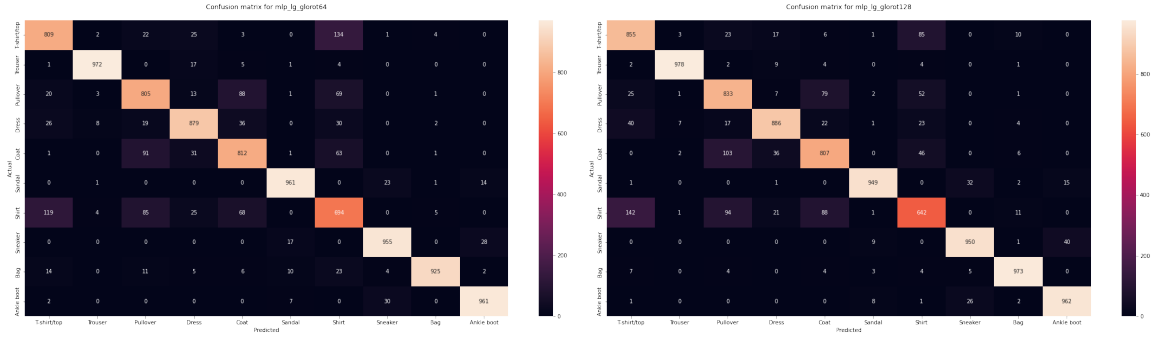

Figure 24: Confusion Matrix for large MLPs with Adam.

	test_loss	test_accuracy
MLP small with 64 and glorot uniform l	0.798712	0.8730
MLP small with 128 and glorot uniform l	0.870116	0.8807
MLP medium with 64 and glorot uniform l	1.041234	0.8768
MLP medium with 128 and glorot uniform l	1.126850	0.8845
MLP large with 64 and glorot uniform l	1.104637	0.8773
MLP large with 128 and glorot uniform l	1.003273	0.8835

Figure 25: Loss and accuracy for the MLPs with Adam within the test dataset.

1.5.3 Training Analysis

We can observe that all the above models are overfitting. So we should use techniques for mitigating overfitting and impoving the generalization performance. Two such techniques are the dropout and the early stopping.

Dropout is a regularization technique that randomly drops out (i.e., sets to zero) some of the neurons in a neural network during training. By doing so, dropout forces the network to learn more robust features that are not dependent on the presence of specific neurons. This helps to prevent overfitting by reducing the network's reliance on any particular set of neurons, making it more generalizable to new data.

Early stopping, on the other hand, is a technique where the training of a model is stopped early based on some criteria. The criteria can be a fixed number of training iterations, or it can be based on the performance of the model on a validation set. Early stopping works by monitoring the performance of the model on a separate validation set during training and stopping the training when the performance on the validation set stops improving. This prevents the model from overfitting to the training data by halting the training before the model starts to learn noise or spurious patterns in the data.

1.5.4 Model Training with dropout and early stopping

Here again we use only the glorot uniform initializer. As described already above, we use Stochastic Gradient Descent and Adam as optimizers. For all training variations we chose to configure 100 epochs and 32 as batch size, while the validation split is 10%.

In this training session we use multiple dropout layers for each model. Specifically we have one dropout layer before each hidden layer and before the output layer. Each dropout layer has a dropout rate of 20%. For the early stopping we monitor the validation loss and we configure a patience of 10 epochs.

1.5.5 Training Results with dropout and early stopping

In the below Figures we can see how each model performed, using the SGD and the Adam optimizer. Available are graphs for the training loss, and validation loss, the training accuracy and validation accuracy. Also there are tables for test loss and test accuracy as well as confusion matrices which were produced during the evalution of the models on the test dataset.

SGD

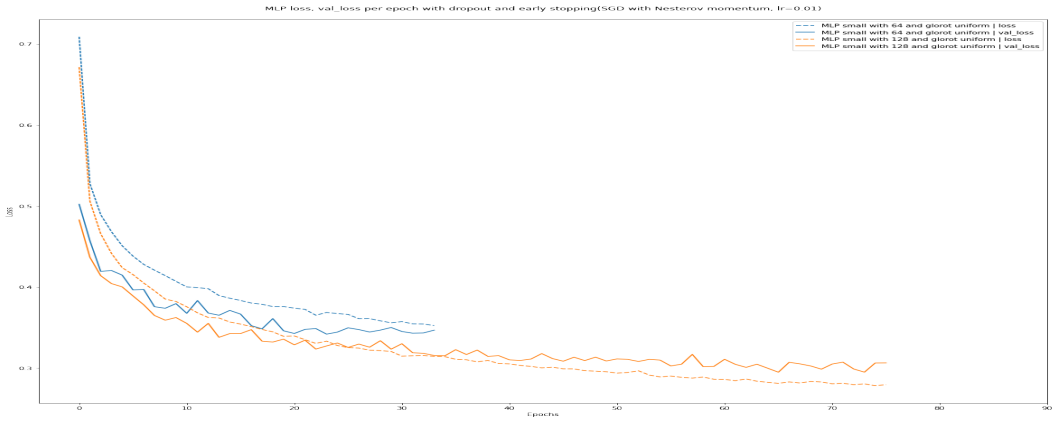

Figure 26: Loss for small MLPs with SGD.

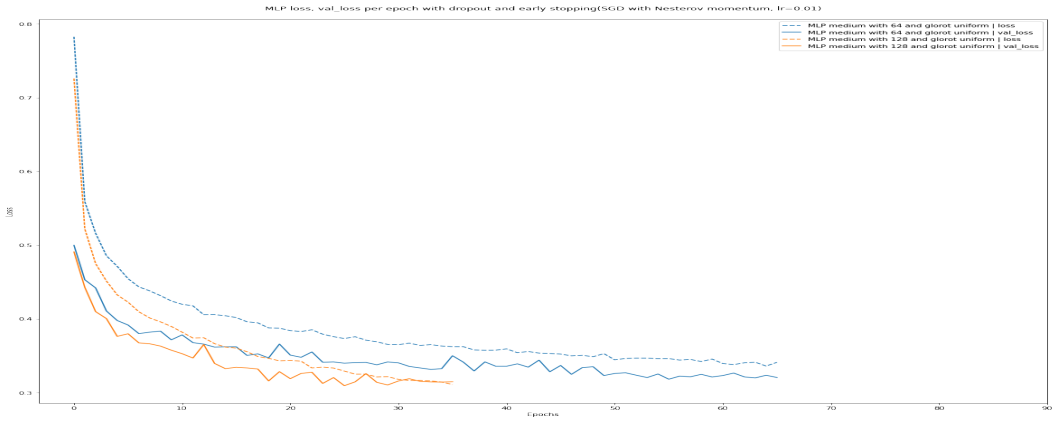

Figure 27: Loss for medium MLPs with SGD.

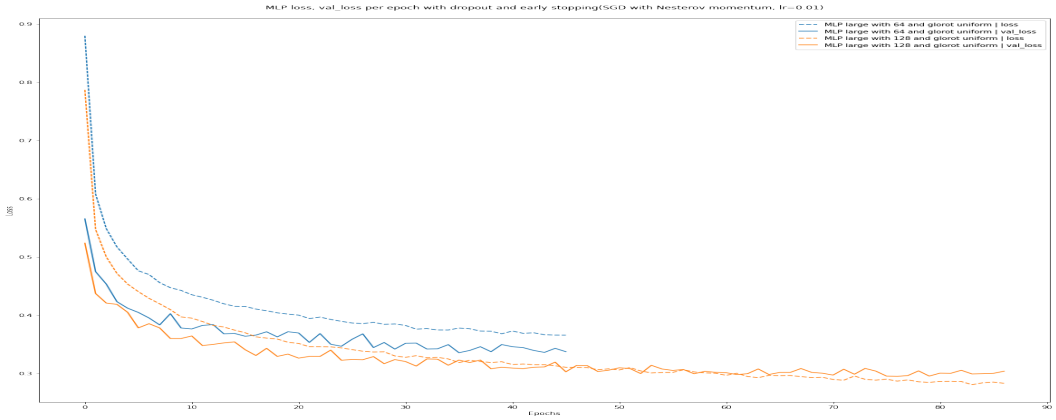

Figure 28: Loss for large MLPs with SGD.

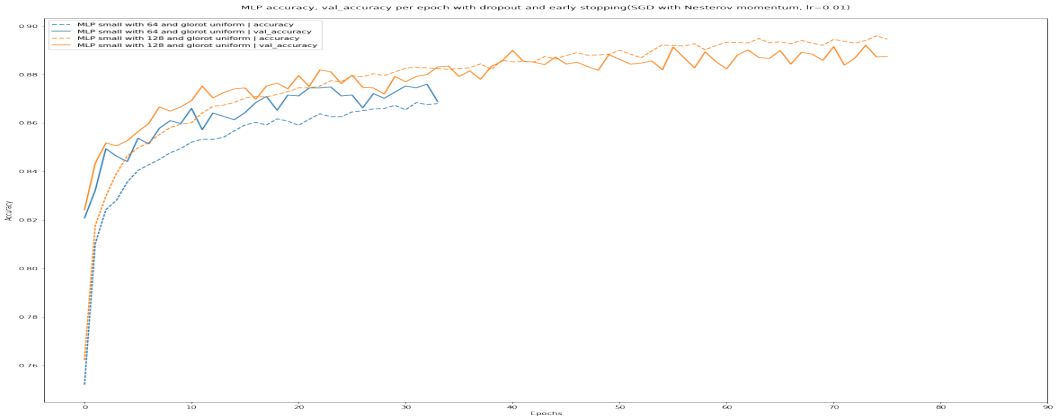

Figure 29: Accuracy for small MLPs with SGD.

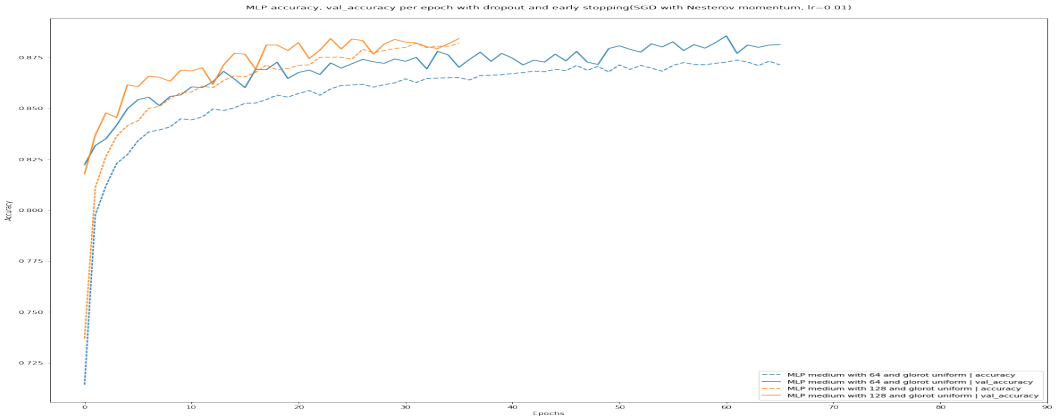

Figure 30: Accuracy for medium MLPs with SGD.

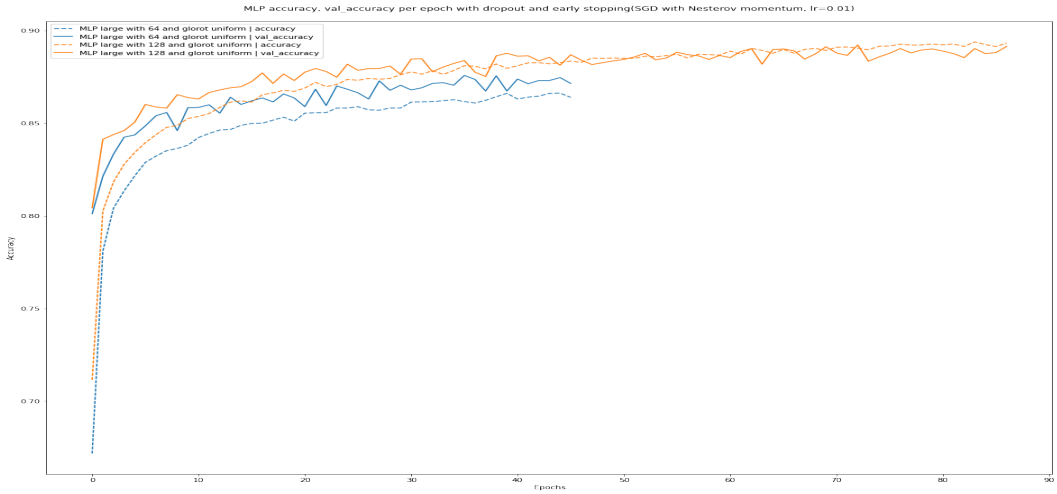

Figure 31: Accuracy for large MLPs with SGD.

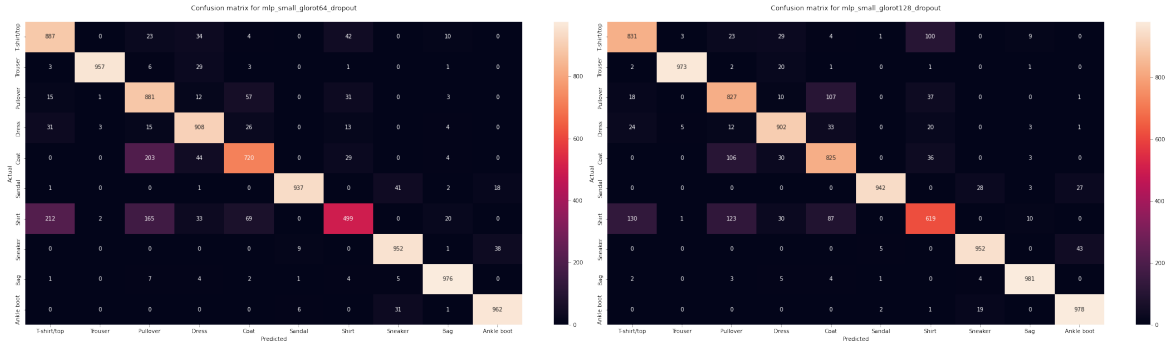

Figure 32: Confusion Matrix for small MLPs with SGD.

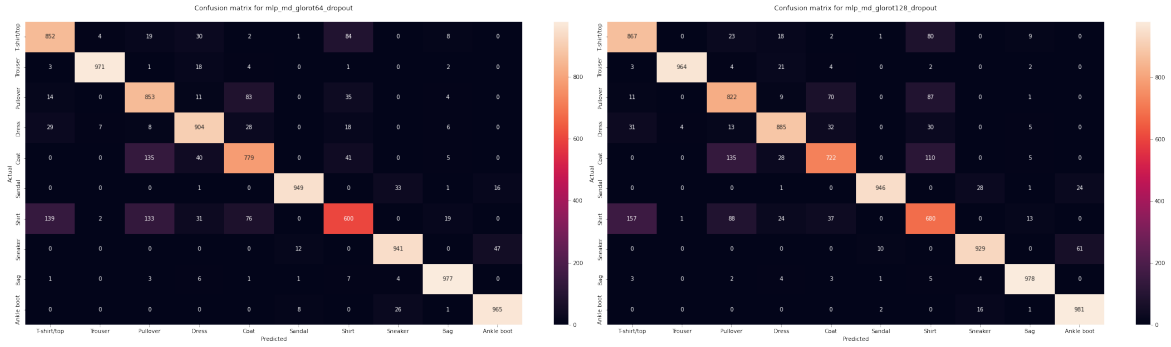

Figure 33: Confusion Matrix for medium MLPs with SGD.

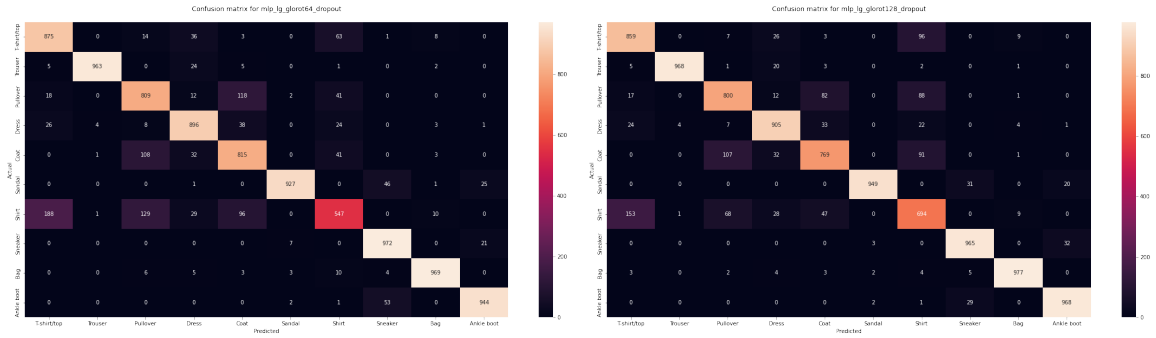

Figure 34: Confusion Matrix for large MLPs with SGD.

	test_loss	test_accuracy
MLP small with 64 and glorot uniform I	0.359710	0.8679
MLP small with 128 and glorot uniform I	0.322176	0.8830
MLP medium with 64 and glorot uniform I	0.339137	0.8791
MLP medium with 128 and glorot uniform I	0.338471	0.8774
MLP large with 64 and glorot uniform I	0.357432	0.8717
MLP large with 128 and glorot uniform I	0.321269	0.8854

Figure 35: Loss and accuracy for the MLPs with SGD within the test dataset.

Adam

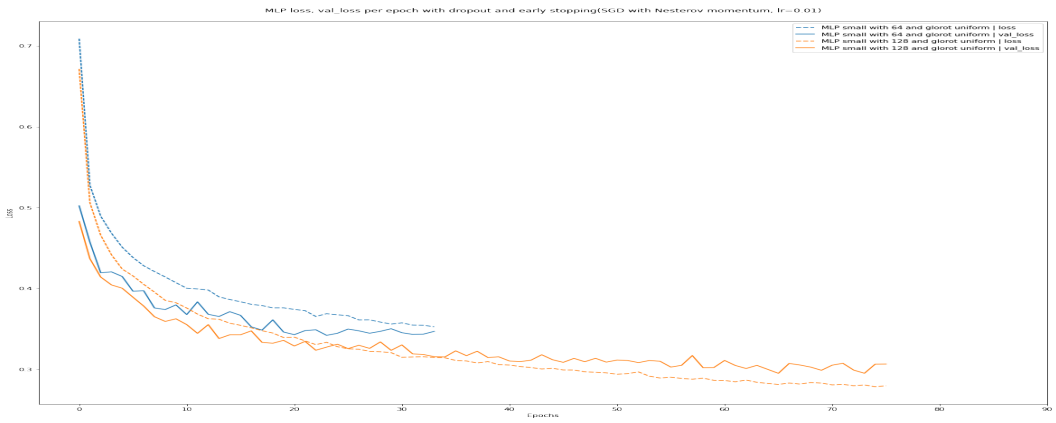

Figure 36: Loss for small MLPs with Adam.

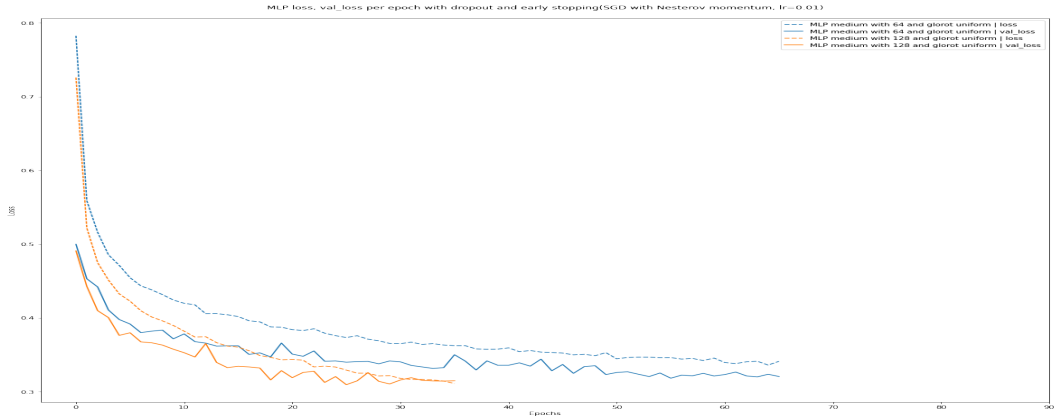

Figure 37: Loss for medium MLPs with Adam.

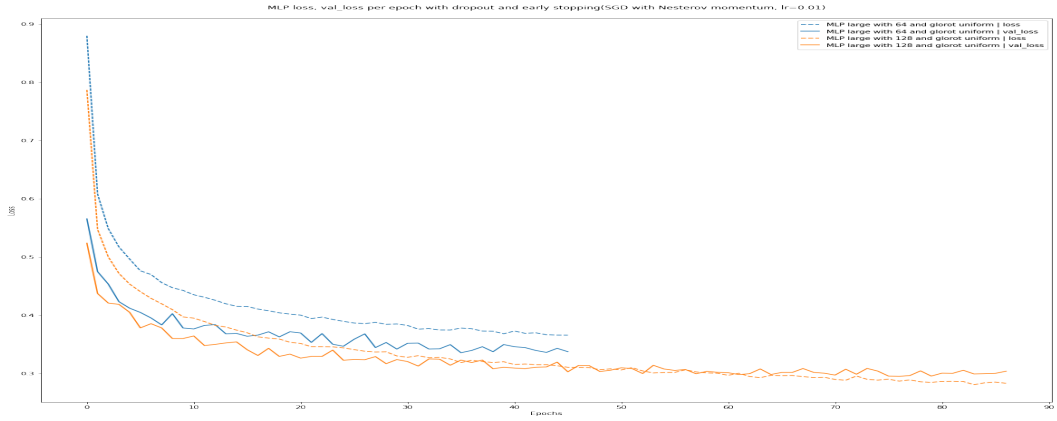

Figure 38: Loss for large MLPs with Adam.

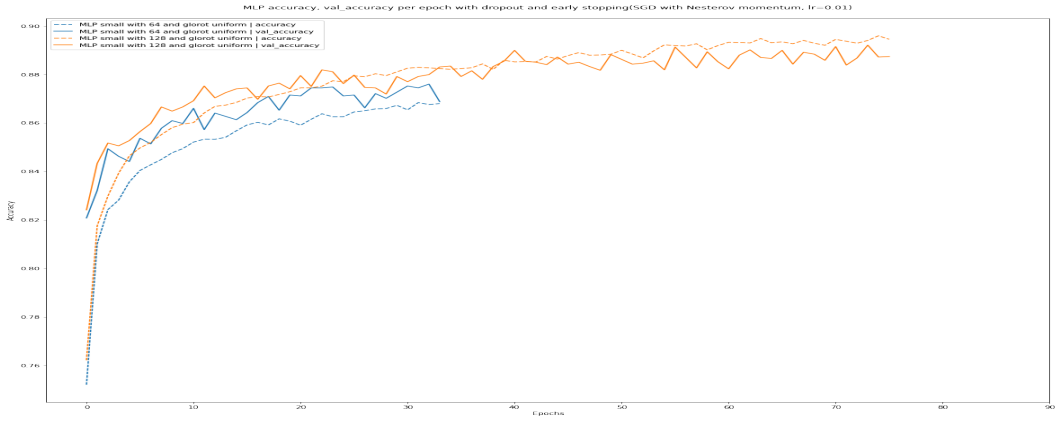

Figure 39: Accuracy for small MLPs with Adam.

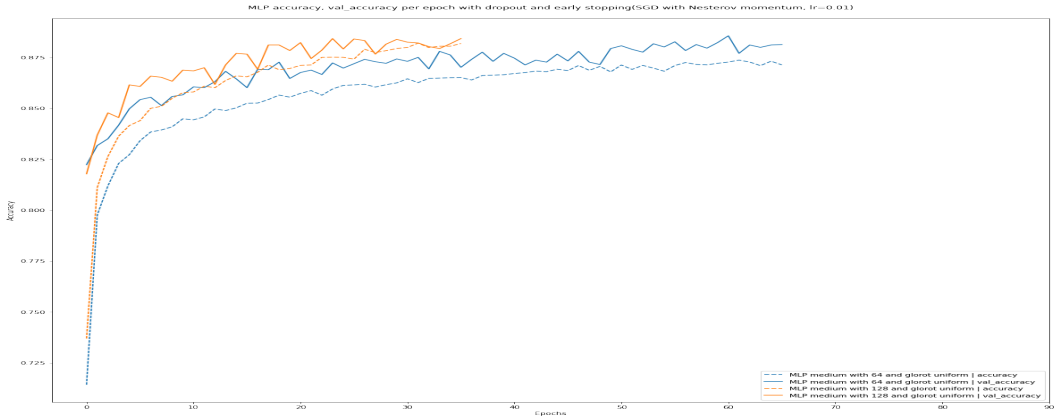

Figure 40: Accuracy for medium MLPs with Adam.

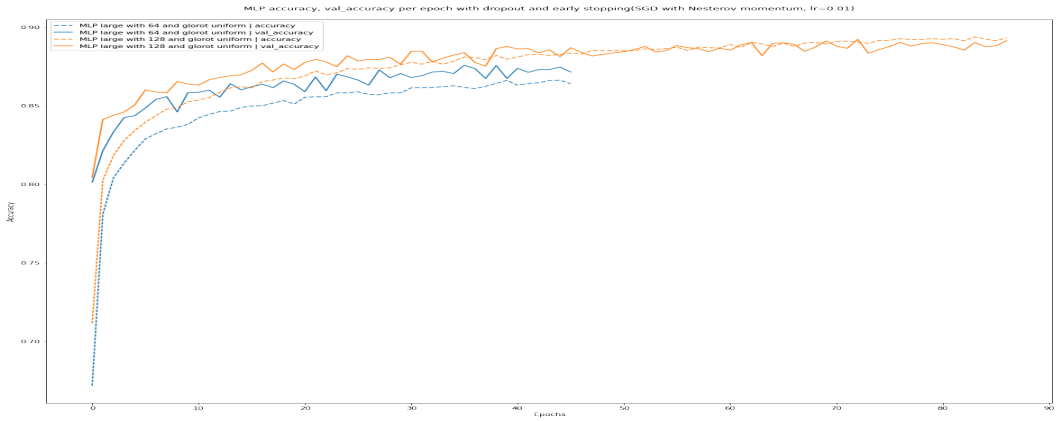

Figure 41: Accuracy for large MLPs with Adam.

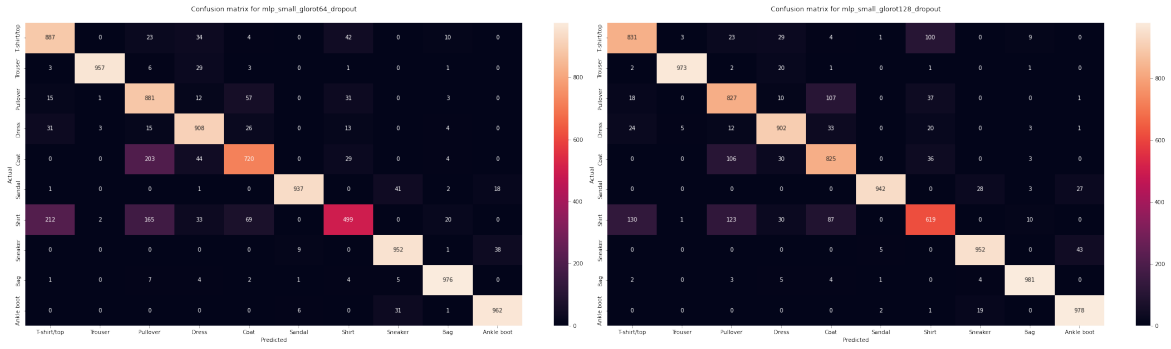

Figure 42: Confusion Matrix for small MLPs with Adam.

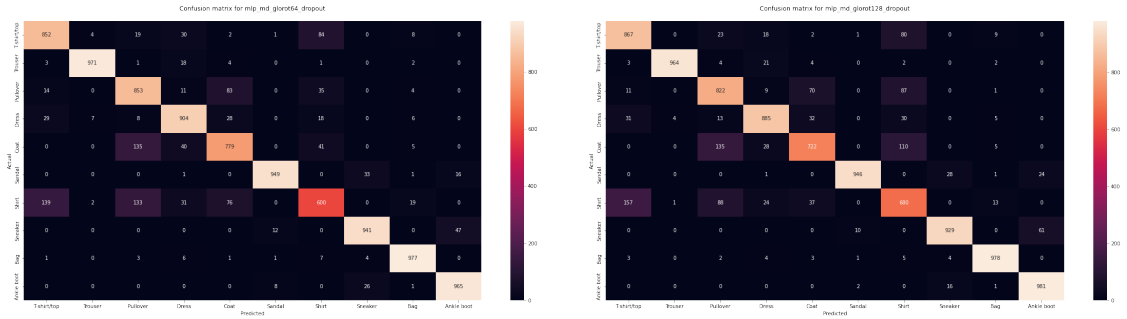

Figure 43: Confusion Matrix for medium MLPs with Adam.

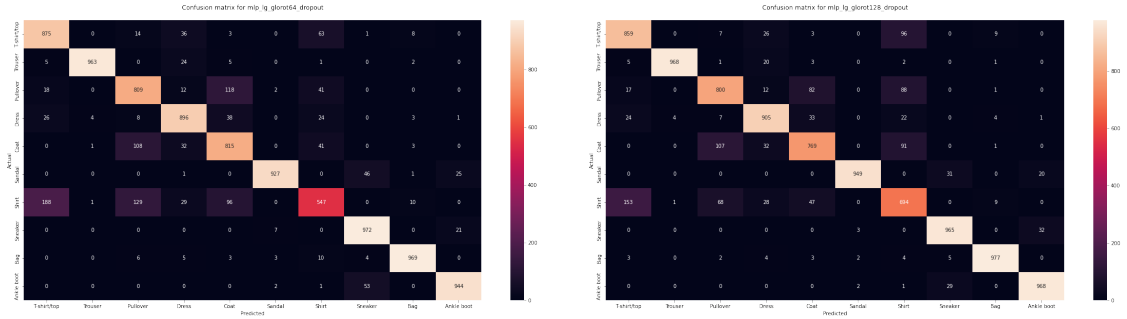

Figure 44: Confusion Matrix for large MLPs with Adam.

	test_loss	test_accuracy
MLP small with 64 and glorot uniform l	0.359710	0.8679
MLP small with 128 and glorot uniform l	0.322176	0.8830
MLP medium with 64 and glorot uniform l	0.339137	0.8791
MLP medium with 128 and glorot uniform l	0.338471	0.8774
MLP large with 64 and glorot uniform l	0.357432	0.8717
MLP large with 128 and glorot uniform l	0.321269	0.8854

Figure 45: Loss and accuracy for the MLPs with Adam within the test dataset.

1.5.6 Model Optimization with Talos

In the final part of the MLP models, we use **talos** library in order to configure, perform, and evaluate hyperparameter optimization experiments. The hyperparameter search space for our MLP models is as defined below:

batch size : (16, 32, 64)
learning rate : (0.01, 0.001)
dropout : (0.1, 0.2)
num of layers : (1, 2, 3)
layer units : (64, 128, 256)
activation : (tf.keras.activations.relu)
epochs : (100)

The top 5 models, regarding the validation accuracy are the following:

	layer_units	learning_rate	end	dropout	num_of_layers	val_loss	accuracy	epochs	batch_size	activation	start	loss	val_accuracy	duration	round_epochs
7	256	0.010	03/05/23-114456	0.2	3	0.292384	0.890118	100	64	<function relu at 0x7f387ef8c310>	03/05/23-114048	0.287496	0.898778	247.566100	26
1	256	0.001	03/05/23-112500	0.1	1	0.306748	0.913176	100	32	<function relu at 0x7f387ef8c310>	03/05/23-112056	0.228145	0.898556	243.781262	21
11	256	0.001	03/05/23-115925	0.1	3	0.291565	0.906588	100	64	<function relu at 0x7f387ef8c310>	03/05/23-115614	0.245752	0.898000	190.965945	20
8	128	0.001	03/05/23-114933	0.2	2	0.288715	0.884039	100	32	<function relu at 0x7f387ef8c310>	03/05/23-114513	0.308596	0.895556	260.589603	26
20	128	0.001	03/05/23-123322	0.2	1	0.291069	0.890980	100	64	<function relu at 0x7f387ef8c310>	03/05/23-123059	0.289758	0.894333	143.017563	27

Figure 46: Top 5 models using the validation accuracy.

Using the best model on the test dataset to evaluate the predictions we have an accuracy of 0.8858.

1.6 CNN Models

We start by building our CNN Models which have three variations in terms of the convolutional layers used. For each model size, we have variations in terms of the number of filters used within the convolutional layers (32, 64, 128, 256) as well as the existence of an extra dense layer. In total we created 10 models as shown in Table (4)

Every layer of filters is there to capture patterns. For example, the first layer of filters captures patterns like edges, corners, dots etc. Subsequent layers combine those patterns to make bigger patterns (like combining edges to make squares, circles, etc.).

Now as we move forward in the layers, the patterns get more complex; hence there are larger combinations of patterns to capture. That's why we increase the filter size in subsequent layers to capture as many combinations as possible.

In Figure (47) we can see the architecture of the model with the greatest capacity as a reference.

The rest configuration of the convolutional layers and the max pooling layers remain the same for all models.

size	convolutional layers	filters	dense layer units
small	1	32	0
small	1	64	0
medium	2	32, 64	0
medium	2	64, 128	0
large	3	32, 64, 128	0
large	3	64, 128, 256	0
medium	2	32, 64	64
medium	2	64, 128	128
large	3	32, 64, 128	128
large	3	64, 128, 256	256

Table 4: Model Variations

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Input (InputLayer)          [(None, 28, 28, 1)]       0

 conv_2d_layer_1 (Conv2D)    (None, 28, 28, 64)        640

 max_pool_2d_layer_1 (MaxPoo  (None, 14, 14, 64)       0
 ling2D)

 conv_2d_layer_2 (Conv2D)    (None, 14, 14, 128)       73856

 max_pool_2d_layer_2 (MaxPoo  (None, 7, 7, 128)        0
 ling2D)

 conv_2d_layer_3 (Conv2D)    (None, 7, 7, 256)         295168

 max_pool_2d_layer_3 (MaxPoo  (None, 4, 4, 256)        0
 ling2D)

 Flatten (Flatten)           (None, 4096)              0

 Output (Dense)              (None, 10)                40970

=================================================================
Total params: 410,634
Trainable params: 410,634
Non-trainable params: 0
_________________________________________________________________
```

Figure 47: Deepest CNN Model used.

kernel size	strides	padding	dilation rate
(3,3)	(1,1)	same	(1,1)

Table 5: Convolutional layers configuration

pool size	strides	padding
(2,2)	(2,2)	same

Table 6: Max pooling layers configuration

1.6.1 Model Training

For the training part we chose to use the Adam optimizer, imported form **keras** library. For all training variations we chose to configure 100 epochs and 32 as batch size, while the validation split is 10%. We also used early stopping with monitoring of the validation loss and a patience of 10 epochs.

Adam

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

1.6.2 Training Results

In the below Figures we can see how each model performed. Available are graphs for the training loss, and validation loss, the training accuracy and validation accuracy. Also there are tables for test loss and test accuracy as well as confusion matrices which were produced during the evalution of the models on the test dataset.

Feature Maps from the Convolutional Layer

Inspired by this **blogpost**

Basically what we do here is to copy a slice of our trained model - the Conv2D Layer. We then feed in a test image and visualise the output from the Conv2D - our feature maps. We plot the feauture maps of the first convolutional layer of the last model, which contains 3 convolutional layers with 64, 128 and 256 units. In Figure (54) we can see the feature maps for the first image of the training dataset and which patterns (edges, corners, lines, etc.) were recognized and used by the model for the decision.

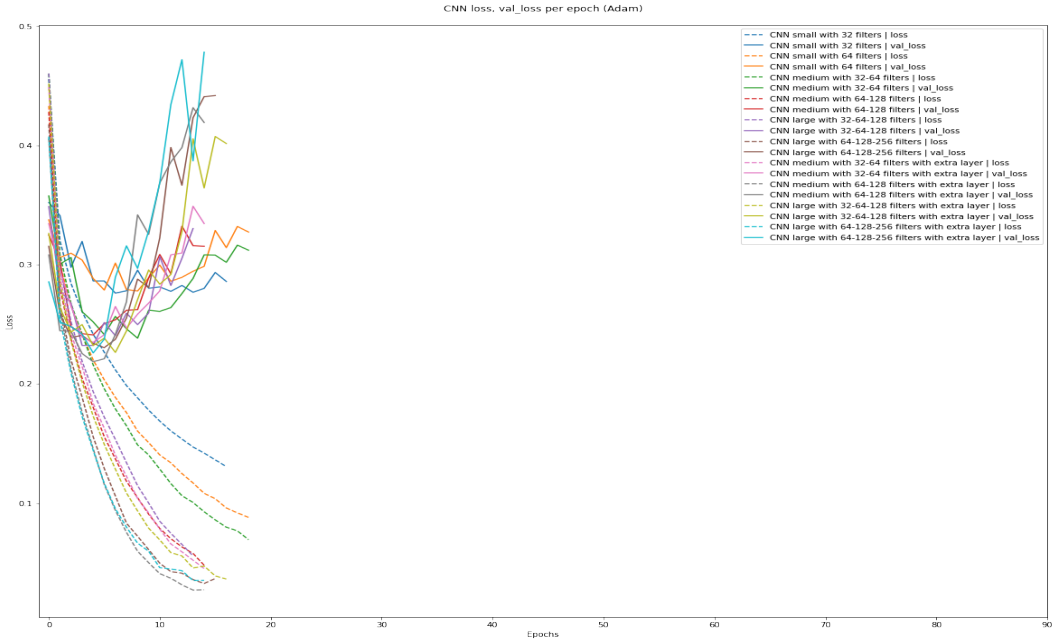

Figure 48: Loss for all CNNs.

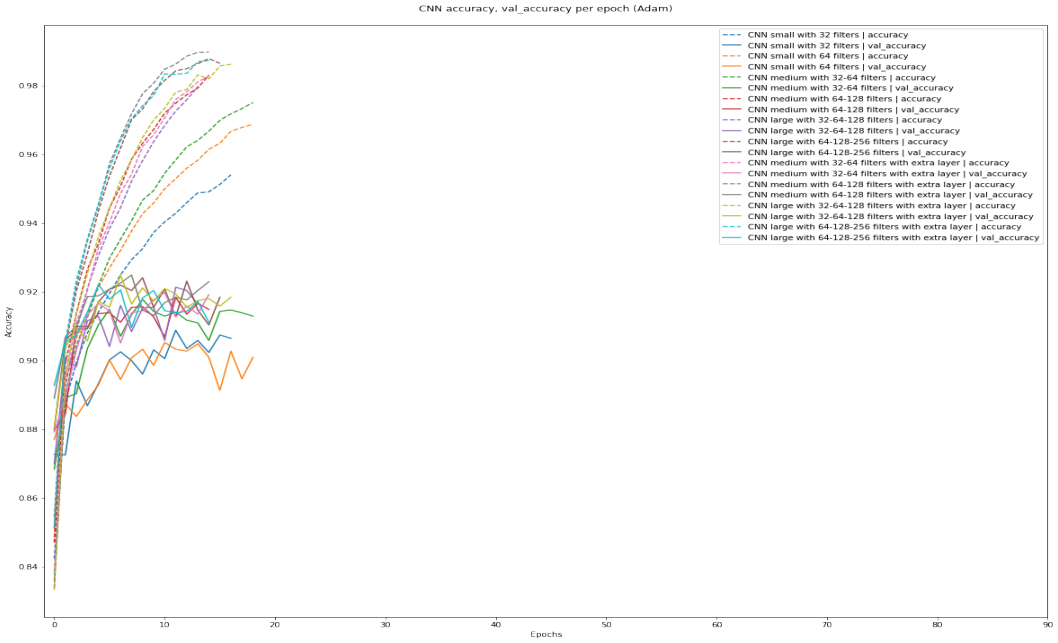

Figure 49: Accuracy for for all CNNs.

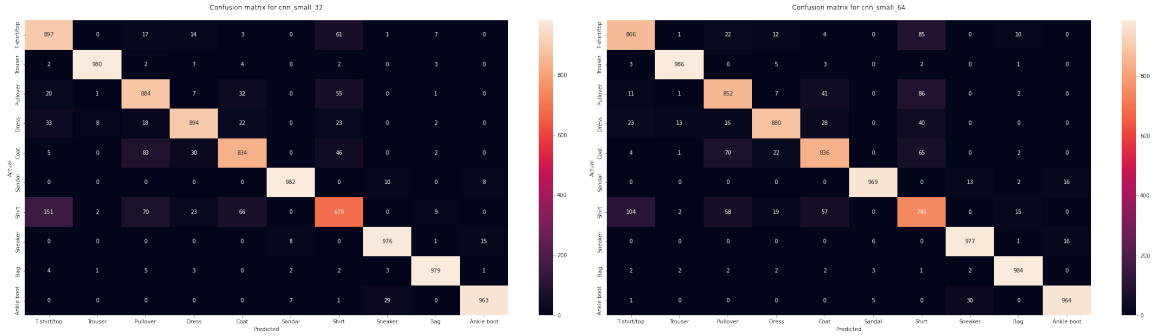

Figure 50: Confusion Matrix for small CNNs.

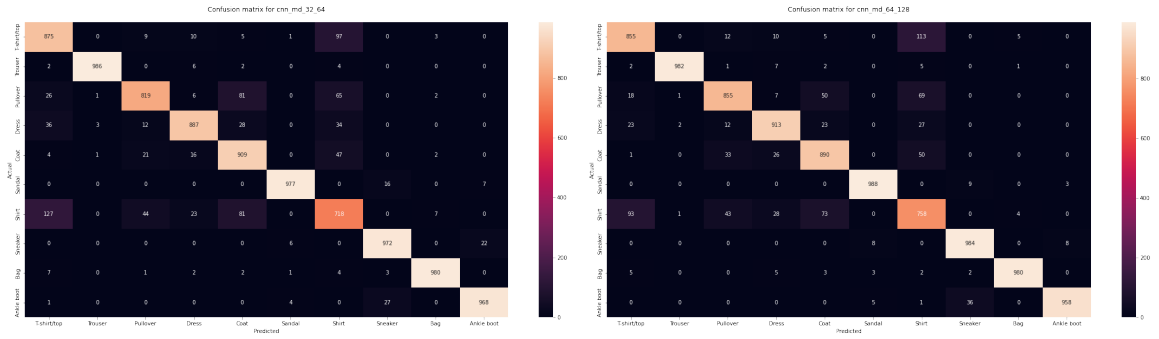

Figure 51: Confusion Matrix for medium CNNs.

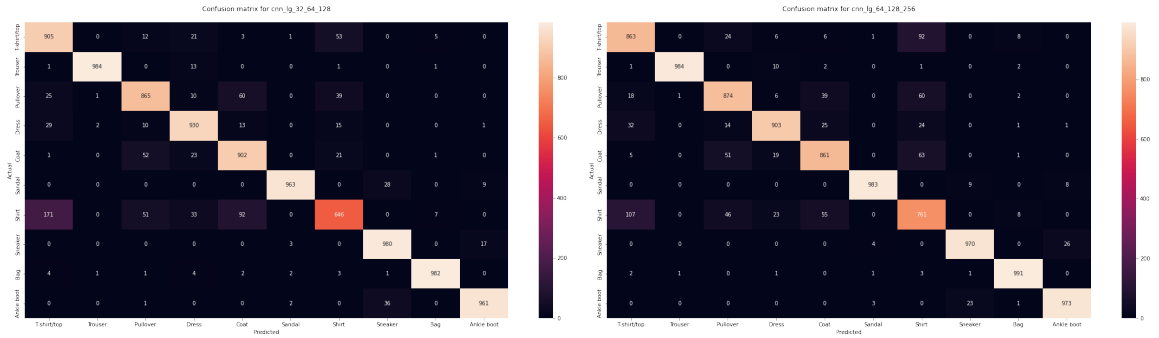

Figure 52: Confusion Matrix for large CNNs.

	test_loss	test_accuracy
CNN small with 32 filters I	0.291403	0.9068
CNN small with 64 filters I	0.329711	0.9059
CNN medium with 32-64 filters I	0.336929	0.9091
CNN medium with 64-128 filters I	0.344992	0.9163
CNN large with 32-64-128 filters I	0.357229	0.9118
CNN large with 64-128-256 filters I	0.471812	0.9163
CNN medium with 32-64 filters with extra layer I	0.379040	0.9107
CNN medium with 64-128 filters with extra layer I	0.477240	0.9168
CNN large with 32-64-128 filters with extra layer I	0.449519	0.9177
CNN large with 64-128-256 filters with extra layer I	0.493574	0.9138

Figure 53: Loss and accuracy for the CNNs within the test dataset.

Figure 54: Feature maps from first Convolutional layer.

1.6.3 Model Optimization with Talos

In the final part of the CNN models, we use **talos** library in order to configure, perform, and evaluate hyperparameter optimization experiments. The hyperparameter

search space for our CNN models is as defined below:

batch size : (32, 64)
learning rate : (0.01, 0.001)
dropout : (0.2)
first layer conv filters : (32, 64, 128)
first layer conv kernel size : ((3,3), (5,5))
first layer conv strides : ((1,1), (2,2))
second layer conv filters : (32, 64, 128)
second layer conv kernel size : ((3,3), (5,5))
second layer conv strides : ((2,2))
dense layer units : (32, 64, 128)
number of dense layers: (0)
epochs : (100)

The top 5 models, regarding the validation accuracy are the following:

	dropout	end	duration	batch_size	second_layer_conv_filters	start	second_layer_conv_strides	second_layer_conv_kernel_size	val_accuracy	epochs
21	0.2	03/05/23-165056	133.305948	64	64	03/05/23-164843	(2, 2)	(5, 5)	0.928889	100
47	0.2	03/05/23-173648	101.045109	64	128	03/05/23-173507	(2, 2)	(3, 3)	0.926778	100
4	0.2	03/05/23-162216	99.621469	32	64	03/05/23-162037	(2, 2)	(3, 3)	0.926667	100
84	0.2	03/05/23-184419	56.206182	64	128	03/05/23-184323	(2, 2)	(5, 5)	0.926111	100
62	0.2	03/05/23-180136	47.105011	64	128	03/05/23-180049	(2, 2)	(3, 3)	0.925444	100

Figure 55: Top 5 models using the validation accuracy 1/2.

learning_rate	dense_layers_num	loss	first_layer_conv_kernel_size	val_loss	first_layer_conv_filters	round_epochs	first_layer_conv_strides	dense_layers_units	accuracy
0.010	0	0.111588	(3, 3)	0.240421	128	12	(1, 1)	128	0.958235
0.001	0	0.119767	(3, 3)	0.229691	128	10	(1, 1)	128	0.955314
0.010	0	0.129789	(3, 3)	0.242846	64	13	(1, 1)	128	0.951627
0.001	0	0.118122	(3, 3)	0.225815	32	10	(1, 1)	64	0.955804
0.001	0	0.145265	(3, 3)	0.219551	32	10	(1, 1)	128	0.946510

Figure 56: Top 5 models using the validation accuracy 2/2.

Using the best model on the test dataset to evaluate the predictions we have an accuracy of 0.9207.