*Ioannis Daskalopoulos*

*F3351805*

**Deep Learning Final Project: Fashion Mnist using CNN**

The goal of this project is to explore various architectures and techniques available in the field of CNNs. I will do so, by developing them and applying them to the fashion mnist dataset. The project is divided into 4 sections, each one building up from the previous.

In the first section we will explore various architectures from old ones to new ones and analyze their building blocks and capabilities. Later, we will compare them and pick the best one before we move over to the next section.

The second section, is the tuning phase. We will try to pick the most important hyperparameters for our previously chosen model, as well as their most promising search spaces. Then we will train the resulting model for many epochs complimenting it with data augmentation.

The third section, will enhance the model derived from the second section with state-of-the-art techniques. In this stage, we will use advanced ways to pick the hyperparameters we previously derived from tuning as well as replace some conventional concepts with improved ones.

The fourth section, will list techniques that I could not use due to time or resource constraints but I would like to do so in the future. I will also briefly explain the concepts and make suggestions on how I would implement them.

We will start by exploring popular architectures historically and applying them to the fashion mnist dataset. We will not tune them, as we will use the recommended values just to measure their general performance. Also, there won’t be any data augmentation yet.

**For all the architectures we use:**

**Activation function**: ReLU

**Number of epochs**: 50

**Input size:** = 28x28x1 (We might need to adjust some architectures for this reason)

**Optimizer:** Adam

**Learning rate:** default in Neural Networks 0.001

**Loss function:** Categorical Cross Entropy

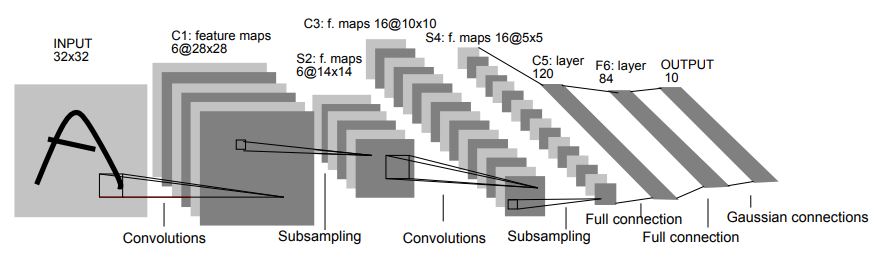
**Metrics:** Accuracy

**No augmentation**

**Batch size:** 32

**Section 1: CNN Architecture**

**LENET-5**



The LeNet-5 is a very classic architecture, which consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier.

input\_shape = (28,28,1)

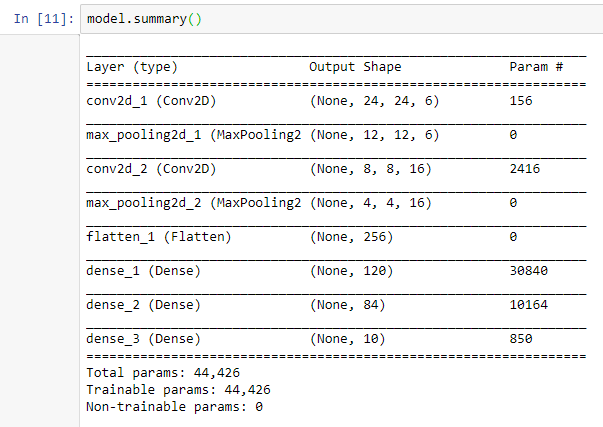
Layer 1: Conv2D (filters = 6, kernel\_size = 5), MaxPool2D (pool\_size = 2, strides = 2)

Layer 2: Conv2D (filters = 16, kernel\_size = 5), MaxPool2D (pool\_size = 2, strides = 2)

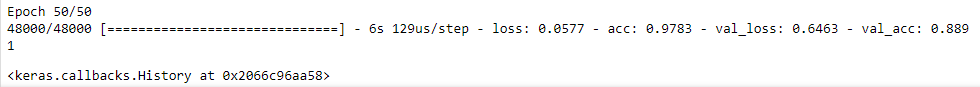
Layer 5: Dense (units = 120)

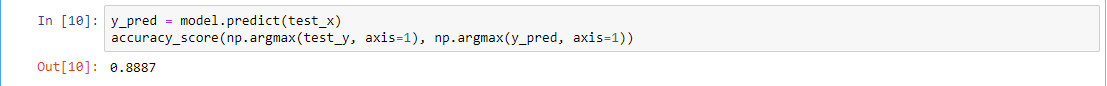
Layer 6: Dense (units = 84)

Layer 7: Dense (units = 10, activation = 'softmax')



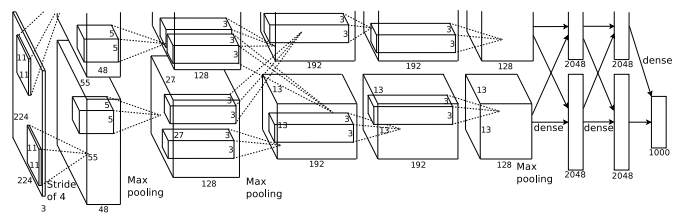
The model is very simple with a low number of parameters and trains very fast.





We start with a 0.8887 accuracy score on the test set which will be our baseline when evaluating the rest of the models.

**AlexNet**



AlexNet is very similar to LeNet, but it is deeper, something which was considered very computationally expensive at its time. It starts with smaller channel of filters which gradually increase and large kernel sizes that end up being 3x3 as the layers progress.

It uses padding=”same” in all conv2d, batch normalization in the end of each layer, dropout at the end of the dense blocks

input\_shape = (28,28,1)

Layer 1: Conv2D (filters=96, kernel\_size = 11), MaxPool2D (pool\_size = 2, strides = 2)

Layer 2: Conv2D (filters=256, kernel\_size = 5), MaxPool2D (pool\_size = 2, strides = 2)

Layer 3: Conv2D (filters=512, kernel\_size = 3), MaxPool2D (pool\_size = 2, strides = 2)

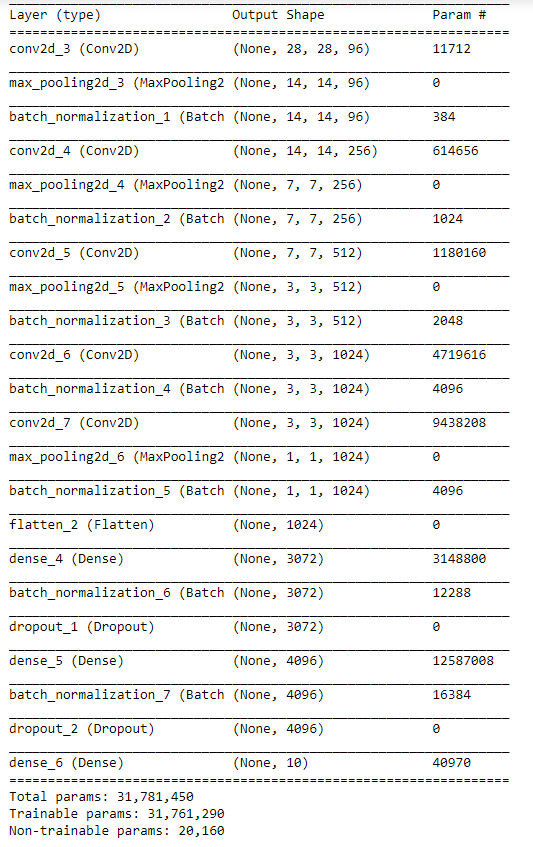
Layer 4: Conv2D (filters=1024, kernel\_size = 3), MaxPool2D (pool\_size = 2, strides = 2)

Layer 5: Conv2D (filters=1024, kernel\_size = 3), MaxPool2D (pool\_size = 2, strides = 2)

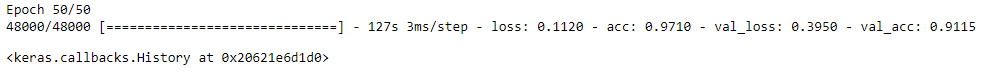
Layer 6: Dense (units = 3072)

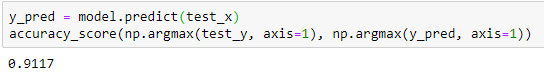
Layer 7: Dense (units = 4096)

Layer 8: Dense (units = 10, activation = 'softmax')



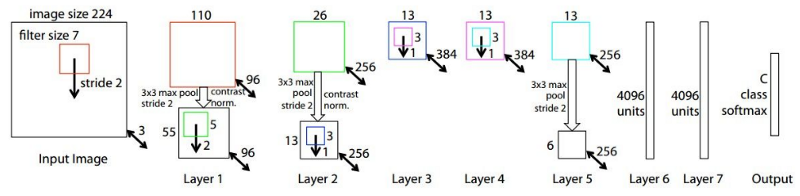
The number of trainable parameters has exploded, increasing the computational cost by a huge factor. This is due to the large amount of feature maps in each layer as well as its depth.





As we can see, by adding depth to the Network the performance on the test set has dramatically increased. Also, the computational cost increased by orders of magnitude.

**ZFNet**



ZFNet is essentially the same as AlexNet when it comes to architecture. Only notable changes are found in the hyperparameters. We can see the dense Layers and the Conv2Ds are using different kernel sizes and number of neurons respectively. Also, we do not have max pooling in every layer. Finally, we start seeing weight initialization (uniform).

It uses padding=”same” in all conv2d, batch normalization in the end of each layer, dropout at the end of the dense blocks.

input\_shape = (28,28,1)

Layer 1: Conv2D (filters=96, kernel\_size = 7), MaxPool2D (pool\_size = (2, 2), strides= (2, 2))

Layer 2: Conv2D (filters=256, kernel\_size = 5), MaxPool2D (pool\_size = (2, 2), strides= (2, 2))

Layer 3: Conv2D (filters=512, kernel\_size = 3)

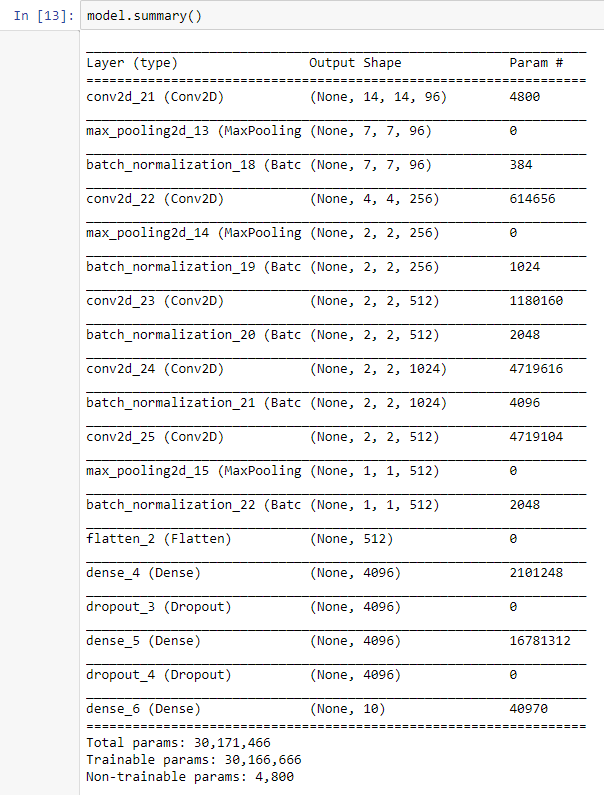
Layer 4: Conv2D (filters=1024, kernel\_size = 3)

Layer 5: Conv2D (filters=512, kernel\_size = 3), MaxPool2D (pool\_size=(2, 2) ,strides=(2, 2))

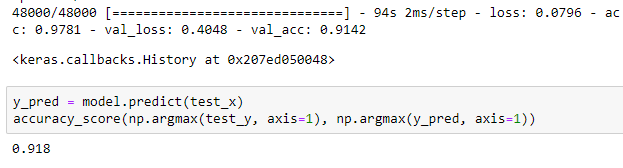
Layer 6: Dense (units = 4096)

Layer 7: Dense (units = 4096)

Layer 8: Dense (units = 10, activation = 'softmax')

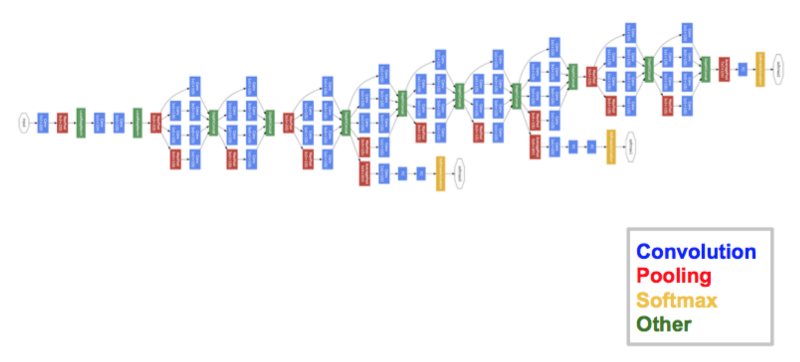


As expected, computational cost is close to that of AlexNet.



Like the paper suggests, it is an improvement in score to the AlexNet.

**Inception V3**



Inception is increasing the complexity of CNNs and introduces many new ideas. The first, is the main building block which is the inception module. Another, is the bottleneck, and finally, multiple exits.

Inception module: It is a mixture of convolution layers and pooling layers with different kernel sizes all outputting feature maps with the same width and height (padding=” same”) stacked together into a single output. The idea is, to not choose between layers, but to get the best of all worlds.

Bottleneck layer: A bottleneck layer, is a 1x1 convolutional layer that exists between an input and a convolutional block. It contains smaller number of channels than both, easing the computational cost.

Exit: We can perform the classification task and calculate the loss, in multiple depths inside the network.

We use padding=”same”, glorot uniform initialization and dropout in dense layers.

Layer 1: Conv2D (64, (7, 7), padding='same', strides= (2, 2)) =>MaxPool2D (pool\_size = 3, strides = 2)

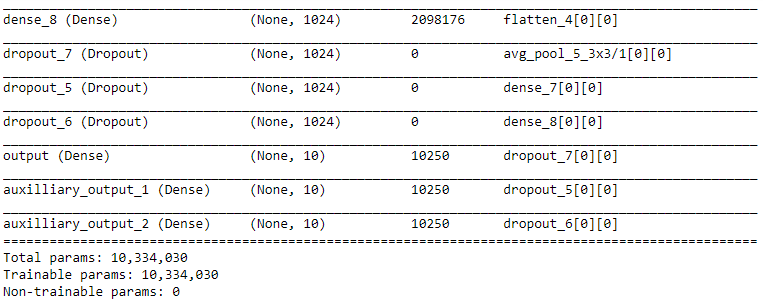
Layer 2: Conv2D (64, (1, 1)) (bottleneck)=>Conv2D (192, (3, 3)) =>MaxPool2D (pool\_size = 3, strides = 2)

Layer 4-6: 3 inception modules and an exit

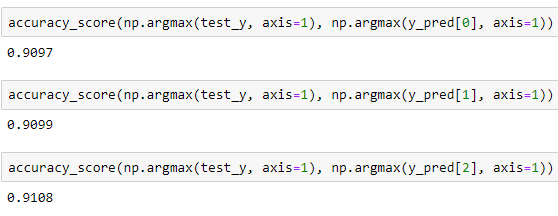
Layer 7-8: 3 inception modules and an exit

Layers 9-11: 3 inception modules and the main exit

Exits are usually: bottleneck layer=>dense=>dense(softmax)

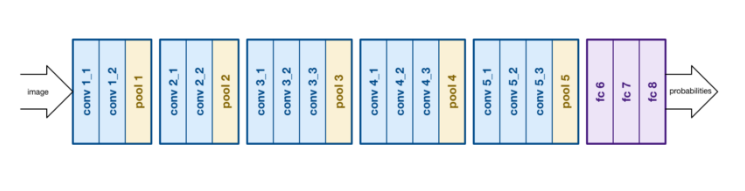


We can see that the number of parameters has been reduced significantly compared to ZFNet. This is partly thanks to the bottleneck layers.



Inception is difficult to train compared to the previous CNNs. However, it has got the capacity to surpass them given enough tuning. We can discern from the results, that each trio of inception modules somewhat increases the performance.

**VGGNet**



VGGNet’s appeal is that while it was developed the same time as inception, it was much simpler with a much more uniform architecture. It starts with 2 sets of conv\*2=>pool and 3 sets of conv\*3=>pool ending with some fully connected layers. With each set. The number of feature maps gradually increases.

Padding is “same” and at the end of the first 2 layers there are dropouts.

input\_shape= (28,28,1)

Layer 1: 2\*Conv2D (64, (3,3)) => MaxPool2D (pool\_size = 2, strides = 2)

Layer 2: 2\*Conv2D (128, (3,3)) => MaxPool2D (pool\_size = 2, strides = 2)

Layer 3: 2\*Conv2D (256, (3,3)) => MaxPool2D (pool\_size = 2, strides = 2)

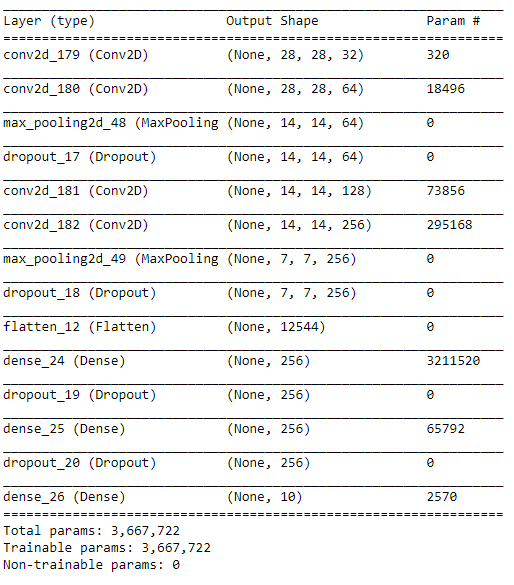
Layer 4: 2\*Conv2D (512, (3,3)) => MaxPool2D (pool\_size = 2, strides = 2)

Layer 5: 2\*Conv2D (512, (3,3)) => MaxPool2D (pool\_size = 2, strides = 2)

Layer 6: Dense (256)

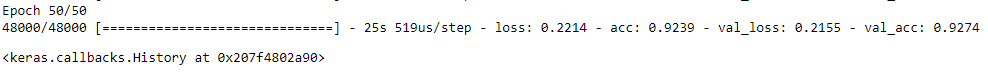
Layer 7: Dense (256)

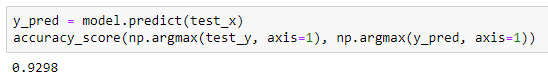
Layer 8: Dense (10, softmax)



The architecture that I described above was similar to the one cited in the paper. However, the number of parameters (which was much higher than that of the Inception) was simply too big for 28x28 images.

This resulted to extremely low performance for comparison. So, I removed layers 3-5 and tweaked a bit the first ones.





Still, the results were incredible, but I think that even with tuning and augmentation (Introduced later), it will be bound to cap at ~9.4.

**ResNet**

ResNet is a serious upgrade to the previous CNNs. They solve a long-standing problem in deep neural networks which is that with depth, the accuracy starts saturating for some epochs and then it sharply degrades. This is the problem I encountered for the first time in the VGGNet before I considered reducing the depth.

They do so by introducing the concept of the Residual block. It contains usually 2 or 3 convolutional layers, passing to the next block the feature map it produced as output and the feature map it received as input! Now the next block can choose either the previous output or if it seems redundant, it can choose the previous input. As a result, if some blocks do not produce quality feature maps, they will simply be discarded and not worsen the total performance no matter the depth. This gives them the ability to converge faster than non-residual networks and produce slightly better performance with each block added.

More specifically my implementation of a residual module has:

1 bottleneck layer to reduce number of channels to ¼ +bn

1 3x3 convolution layer + bn

1 1x1 convolution layer to increase the number of channels to a specified amount + bn

(The above 3 could be just a 3x3 block with the specified amount of channels but the computation cost would then skyrocket)

Their output is then added to the input and returned

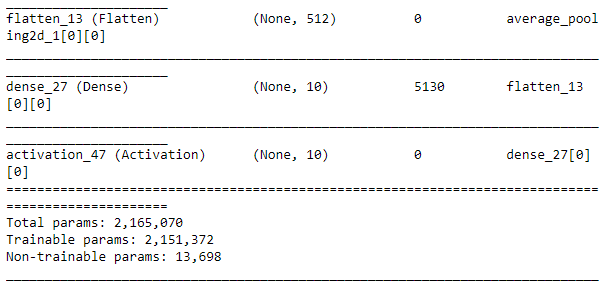
Layer 1: Conv2D (64, (5x5)) =>bn=>MaxPool2D ((3, 3), strides= (2, 2))

Layer 2-4: Residual block (128)

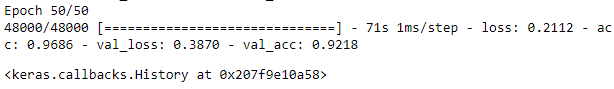
Layer 5-9: Residual block (256)

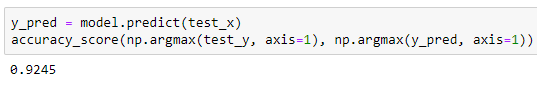
Layer 10-16: Residual block (512)

Layer 17: Average Pooling (4,4) =>dense (10, softmax)



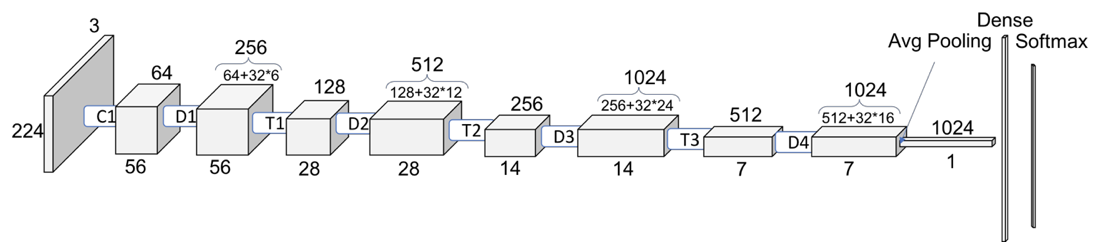
By using the bottleneck trick, even though we have created a relatively deep network, the number of parameters is fairly low.





In the results we can see that there is overfitting which we could address later with tuning. Nevertheless, the accuracy is very high for a network which could easily scale to 100+ blocks without problems.

**DenseNet**



DenseNets like ResNets solve the problem of vanishing gradient in deep networks by using connections between layers. This gives DenseNets the option to keep the previous input, if the current layer proves to be harmful for the network. The huge difference between a ResNet and a DenseNet is that the later uses “dense blocks” which are multiple layers that are interconnected with each other instead of just being serially connected.

The major advantage is that fewer parameters are required for the same or better level of performance in comparison to the older networks. This is because, there is no need to learn redundant feature maps, as they always keep the best and discard the others.

When it comes to the gradients from the loss function, DenseNets are very robust, as all the layers have direct access to the input image embedding

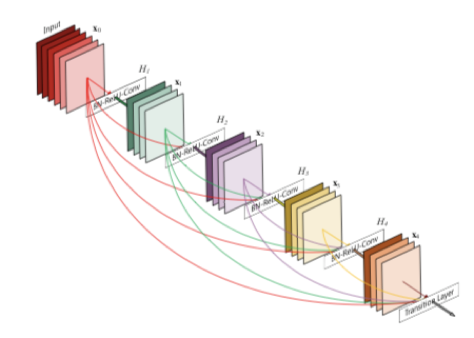
DenseNets are built from DenseBlocks and *Transition Layers.* In denseblocks, the dimensions of the feature maps remain the same. Between denseblocks there are Transition *Layers*which take care of the downsampling applying a batch normalization, a 1x1 convolution and a 2x2 pooling layer.

Another important concept introduced with the DenseNets is the growth rate. In essence what we are doing is concatenating the feature maps. This is why we need a rate with which to increase the number of feature maps with each layer. The formula states that the next layer will have as many layers as the previous plus the growth rate K. We could say that K is how much information is added to the network with each passing layer.

In overall, we could say that, because every layer inside the dense block has access to the output of all the other layers, the information is collective and additive.

The 3 subcategories are DenseNets-B, densenets-C and their combination: DenseNets-BC. For DenseNets-B, B stands for the bottleneck layers they have before the 3x3 convolution layers for computational efficiency. For DenseNets-C, C stands for the compression rate which hinders the growth of output feature maps in a rate between [0-1]

**Dense block**



It is built from convolution blocks which are actually bn=>bottleneck=>bn>3x3 conv.

To initialize a dense block, we create a list containing only the input. After each convolutional block is created, we add its feature map to the list and concatenate them together as output. This ensures that each next convolutional block will get all the information from the previous feature vectors.

The transition block has bn=>compression layer (1x1 conv2D like bottleneck) =>average pooling2d.

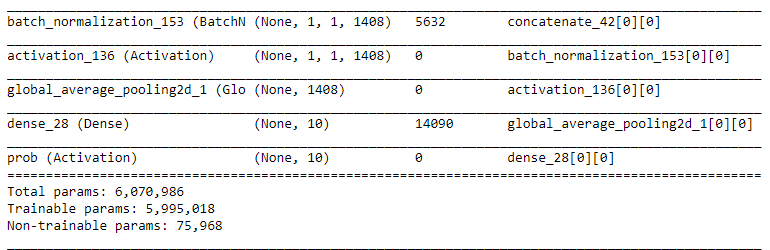
What I have implemented is a DenseNet-B with the capability to become a DenseNet-BC (But I have left the compression factor C at 1)

Layer 1: Conv2D=>bn>MaxPool2D

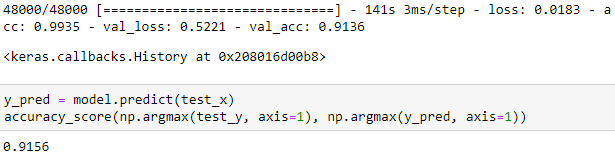
Layer 2: dense block out of 6 conv blocks

Layer 3: dense block out of 12 conv blocks

Layer 4: dense block out of 24 conv blocks

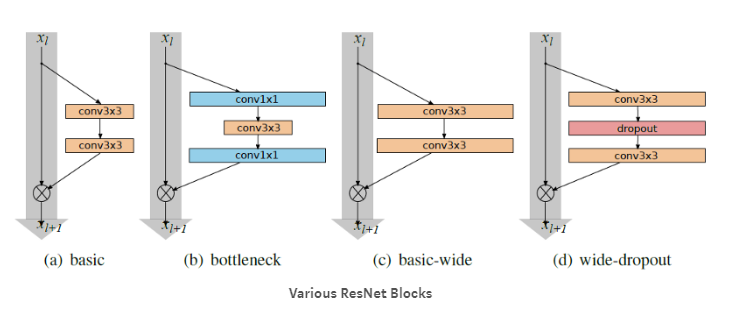


The resulting model has got 6m parameters which is very good considering we have almost 50 layers.



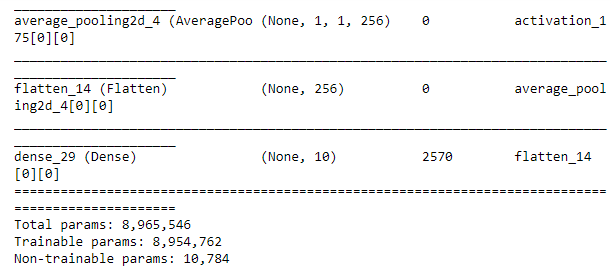
On the down side we overfitted. This is the reason the model’s later versions provide the compression and dropout hyperparameters which I kept to default just to be in line with the rest of the models.

**WRN**

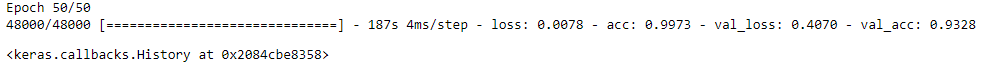


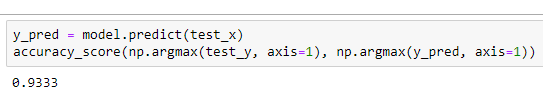
WRN was introduced in order to solve a previous ResNet problem. That is, although we could potentially create ResNets with thousands of layers, we could not force the gradients to go through them, as they in many cases preferred to take the shortcuts. This led to performance increase every time we doubled the size of the network with the computational costs increasing exponentially.

WRNs control the depth and width of each layer, motivating meaningful (more specialized) representations the deeper we go. That way, the gradients had an incentive to choose the actual blocks instead of the shortcuts all the time. Some popular models are the (28,10) and (40,4)



The number of parameters is in normal levels considering how deep the network is.





We can observe some extreme overfitting which is normal when we take into account that we did not use any regularization such as dropout throughout the model. The results it gave are still the best out of all the CNNs from the get go.

In overall, my first choice of architecture was a model very similar to VGGNet even before studying it. The model I used instead of the original one came from some experiments with it. This only shows how simple yet powerful an idea, a VGGNet is.

However, WRN looks more promising in terms of latent capabilities. It also looks very interesting albeit heavy. So, I would like to continue experimenting with it.

**Section 2: Tuning**

From all the architecture showcased above, the models that stand out are the Resnet and its variations as well as VGGNet. Despite the simplicity of VGGNet, tweaking its topology is simple and can give a major boost in accuracy. However, I already tweaked the topology a bit to accommodate for the dataset. So, I do not think that I will get a much better performance with tuning. What piqued my interest the most was the WRN. In spite of, overfitting horribly, it provided the best performance out of the base architectures. With this in mind, I am inclined to believe that I can get it to perform much better, as well as explore much more interesting hyperparameters and concepts.

In order to tune the WRN I tried 2 options:

1) Use my graphics card which is a Nvidia 1060 6GB

2) Use Google Collab which provides a 12GB GPU

The second option came with a limitation. I could run one session for 12 hours before it close. I tried calculating the time I needed to tune the network which was 20 epochs \* 30 evaluations \* ~3.5 minutes per epoch = 35 hours. The reason I calculated I needed that much time is described later in this section.

In this light, I chose to use my graphics card.

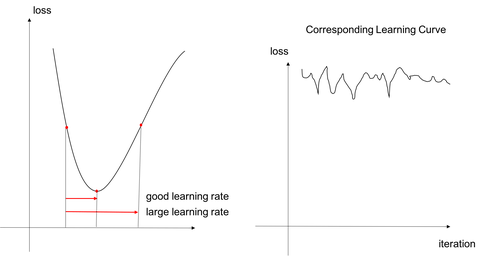
I have 3 phases during the tuning section:

1) A phase in which I tuned 20 models for 1 epoch each. I did that so that I could detect any potential bugs during the training and get some first insights about how the hyperparameters behave. I will be commenting on them throughout the section.

2) Main tuning phase. I will train 30 models for 20 epochs each. There will not be any tuning for augmentation as the process is quite similar and I do not have the resources (time) for this.

3) Run the final model for 100 epochs with augmentation and comment on the results from the experiment.

**Learning rate**



Learning rate is considered to be the most important hyperparameter for any Neural Network.

Generally, a large learning rate allows the model to learn faster, at the cost of it arriving on a sub-optimal final set of weights. A smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train.

The rule of the thumb is to initialize at 0.001 which is also the default value for most Keras optimizers. However, it is not wise to expect that this value will work for every neural network.

The paper I linked bellow states, that a more sensible search space for the learning rate is [10^-6,10^0].

**I decided to omit slower learning rates**

Due to restrictions in time and hardware, I will only use 20 epochs per model, so the higher learning rates will be favored. To further enhance this point, low learning rates are considered to slow down the model, reduce its generalization ability and thus overfit. During our first run of WRN, I found out that it had overfitting problems and I would not like to make worse. Furthermore, I will make use of reduce learning rate on plateau which allows for smaller learning rates if needed.

**The first tuning phase made clear that very high learning rates do not work for this network.**

My first experiment “Hyperas Optimization” used [10^-3,10^0]. The first results have shown me that the space [10^-1,10^0] steadily produced bad results as the model failed to converge.

So, I decided to change the search space for the main tuning to [10^-1,10^-4].

***Practical recommendations for gradient-based training of deep architectures*** <https://arxiv.org/abs/1206.5533>

**Number of epochs**

The number of epochs as stated above will be 20. In WRNs there are many hyperparameters which I would like to tweak at least a little and thus I would like to try 30 models. Each epoch takes from 2 to 5 minutes with an average close to 3.5. **[(28,10) takes ~5 and the rest take ~3]**

Thus, even for those tries I would need around **20\*30\*3.5=2100m=35h**. I cannot distribute any more time in the tuning phase.

The practical reason I chose 20, is that in the previous experiments, all models had already converged before this number. Also, the reason I opted for as many evaluations as possible, is that Hyperopt (and consequently Hyperas) use the tpe. suggest strategy. This strategy, narrows down the search space as the number of models evaluated increases.

TPE (Tree Structured Parzen Estimator)

It belongs to the category of Sequential model-based optimization (SMBO) methods which are a formalization of Bayesian optimization. They are sequential because the models are run one after the other. Each time, we try to find which set of values is the most probable of producing the best results, given the loss we got from the previous set of values. More specifically, TPE use the Bayes rule in order to find the next set of hyperparameters. For this reason, it is essential to fit as many evaluations as possible in this schedule.

Early stopping

To further tackle the problem of overfitting and potentially save some time in the process, we will use a soft early stopping. It will have a patience of 5, which means that if the loss does not improve for 5 runs in a row, the model stops.

Reduce learning rate on plateau

Sometimes the model may get a very big initial learning rate and converge very fast. This means that in the best-case scenario it will stop early in around the same accuracy. However, the accuracy might also start to decrease so that we end up with a worse accuracy than its true potential. The solution is to cut the learning rate by half every 3 runs without increase in accuracy, with the hope that we will see further improvement in score, before the early stopping forces it to stop.

**Number of hidden layers and units**

Now, in WRNs we could theoretically increase the number of hidden layers and units infinitely and not have a problem like in Resnet. However, increasing the growth factor and depth sharply increases the number of computations. There are a lot of common topologies for WRNs.

For this reason, I tried to find the best representations of a balanced, a wide and a deep topology. (22,8), (28,10) and (40,4) fit the bill respectively.

**Batch size**

The paper linked bellow, suggests that the batch size must be between 32-512. The batch size should always be a power of 2 since the number of the physical processors in the GPU is always a power of 2 and this allows for a better allocation of resources. Large batch sizes lose in generalization ability and small batch sizes increase the computational cost. This is due to how computationally efficient the vector computations are in neural networks.

After experimenting in tuning phase 1 with different WRN topologies, I found that my graphics card memory can only support 32, 64 for (28,10) and 32, 64,128 for (22,8) and (40,4).

**On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima**

<https://arxiv.org/abs/1609.04836>

**DROPOUT**

Nowadays, all variants of Resnet make heavy use of batch normalization.

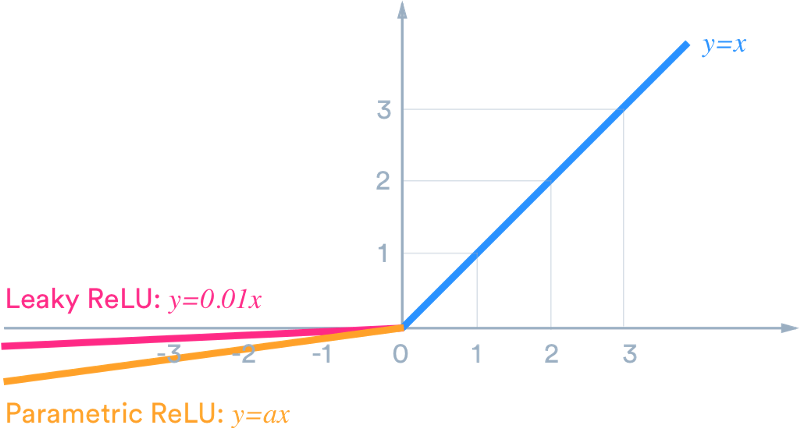
There are many papers like the one linked bellow, which suggest that a combination of batch normalization and dropout leads to bad performance, or outright state that dropout is redundant. However, we can still have our tuning algorithm to decide its importance. The fact that our model overfitted previously means that there is space for regularization anyways.

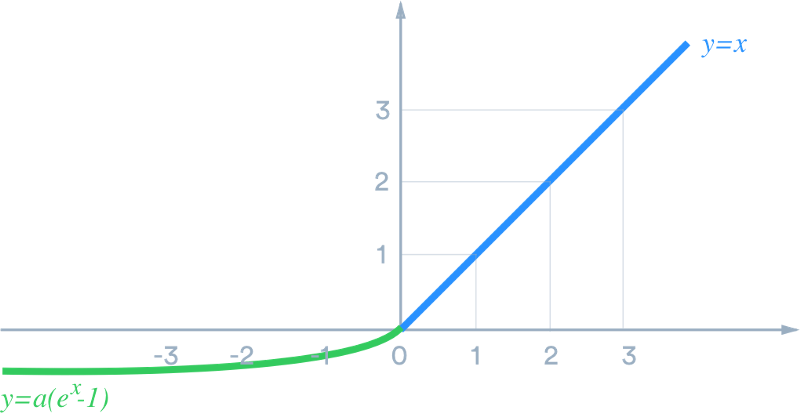
I will try the typical values, which are between 0 and 0.5.

**Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift**

<https://arxiv.org/abs/1801.05134>

**Activation function**





All variations of ReLU have their pros and cons and I did not find anything convincing to use one over the other. Nevertheless, they seem to consistently outperform more traditional activation functions when used in Resnet variations. Thus, I tried most ReLU functions, Keras offers.

**Weight initialization**

Another very important hyperparameter is the weight initialization.

We would like the starting weights in the neural network to not be too small as they would degrade to numbers close to 0 during training in deep networks, becoming useless.

We would also like the starting weights in the neural network to not be huge either, as they would be amplified during training in deep networks, becoming useless as well.

There are 2 popular techniques that Keras offers.

**All you need is a good init**

<https://arxiv.org/abs/1511.06422> (Glorot/Xavier)

This paper was written and tested at sigmoid functions which could take values in [-1,1].

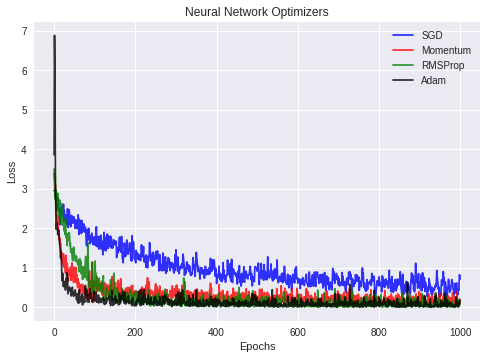
**Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification**

<https://arxiv.org/abs/1502.01852> (He)

This paper was published to accommodate for the ReLU which became the dominant activation function later, which could take values in [0,1].

The differences in the formulas are minimal. So, I decided to use both and see for myself which initialization is better for the model. One more note is that in the original papers Glorot was uniform and He normal, but here I used both distributions for each.

**Optimizer**



There is no single best optimizer. Adam is the one most widely used optimizer. However, other optimizers can work just as well or better, depending on the problem. Some of them converge faster, while others opt for precision.

What usually, sets apart one optimizer from another, is the use of momentum and adaptive learning rate.

Momentum is the notion taking all the previous steps with significance from newest to oldest into account when calculating the next step’s direction.

Adaptive learning rate is the notion of taking all the previous steps with significance from newest to oldest into account when calculating how fast the next step will be. Thus, it will go faster in plateaus and slower in minimas.

Adam may be this popular because it combines both properties, providing more hyperparameters for tuning.

For our purposes we will use most optimizers that Keras offers as we have no lead in what works best for this particular problem.

**Hyperas**

Hyperas is a wrapper for Hyperopt, allowing for faster prototyping of the code required for tuning. It is a string parser and this imposes some limitations to its capabilities. For our purposes, it is more than enough and actually very helpful after we surpass some of its initial caveats.

1) The use of comments is discouraged (typical of string parsers)

2) We need to explicitly specify the name of the notebook it will read (Whole process becomes much easier if we simply run the code in a .py).

3) In the first tuning phase the program was always running out of GPU memory. The reason was that it kept the models in its memory, in order keep the best model and some history data in the end. The only solution was to clean memory and delete the models in each evaluation. The drawback was that I had to search for the best model manually. Also, I could not provide any plots for information.

4) I could not pass the activation functions and optimizers as strings as they never get activated by the parser. Instead, I passed the classes and let the model initialize them during running.

For hyperparameters I used uniform for continuous values and choice for discrete. (pseudo log uniform for learning rate).

A described above, the tpe suggest algorithm was used for 20 epochs and 30 evaluations.

**Data Augmentation**

In the final model I used data augmentation with the generator not being tuned. So, I tried to explore the best parameters that would likely not confuse images from different categories.

Horizontal flip: It seems reasonable that the network should recognize a cloth from both sides

Vertical flip: I believe it is dangerous as it could confuse items with bulky areas on bottom (e.g. dress) with ones that have bulky areas on top (e.g. t-shirts).

Zoom: It will probably not give much accuracy, but in small values it is harmless to try. (I set it to a value that usually cuts a perimeter 1 pixel wide). It will usually just omit portrait area which holds no information.

Whitening/standardization: The pictures are simply black and white, so those techniques would just alter the contrast.

Rotation: Rotation might be good to try, but like Zoom, in very small doses. Maybe 1-2 degrees so that it does not alter the shape of items close to the bottom like shoes.

Shifting: Like rotation, shifting in small values should not drop a lot of useful pixels as most items have black background left and right.

**Tuning results**

Architecture: The best results were observed by (40,4) and (28,10)

Activation: We saw a lot of LeakyReLU but other functions were represented as well.

Initializer: Top models used different initializers so I cannot say they had much impact

Dropout: (28,10) seemed to have a sweet spot at around 0.2 in dropout whereas (40,4) preferred a range at [0.15,0.3]

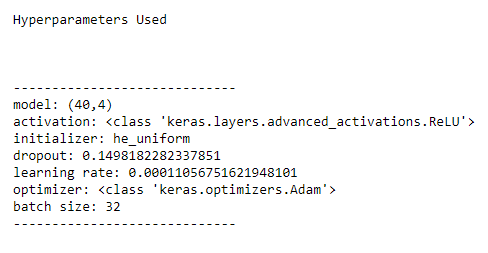
Learning rate: (28,10) preferred a learning rate of the magnitude [10^-2,10^-1], while (28,10) preferred a learning rate of the magnitude [10^-4,10^-3].

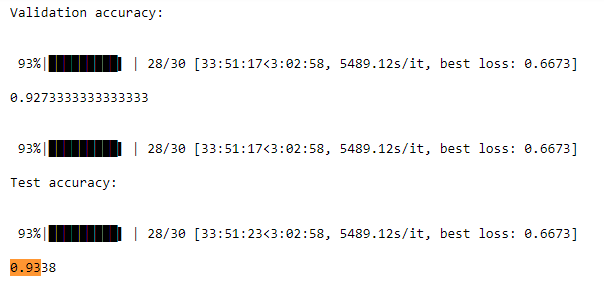
Optimizer: The dominant optimizers were variations of Adam

Batch-size: 32-64 were the most represented values in the top models.

**I chose the model which performed best in both validation and test sets.**

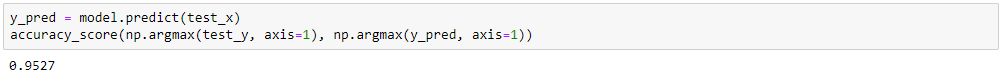
**It seems that (40,4) architectures have what it takes to provide a bit better generalization ability.**



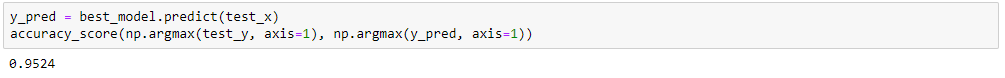


**Tuned model + augmentation**

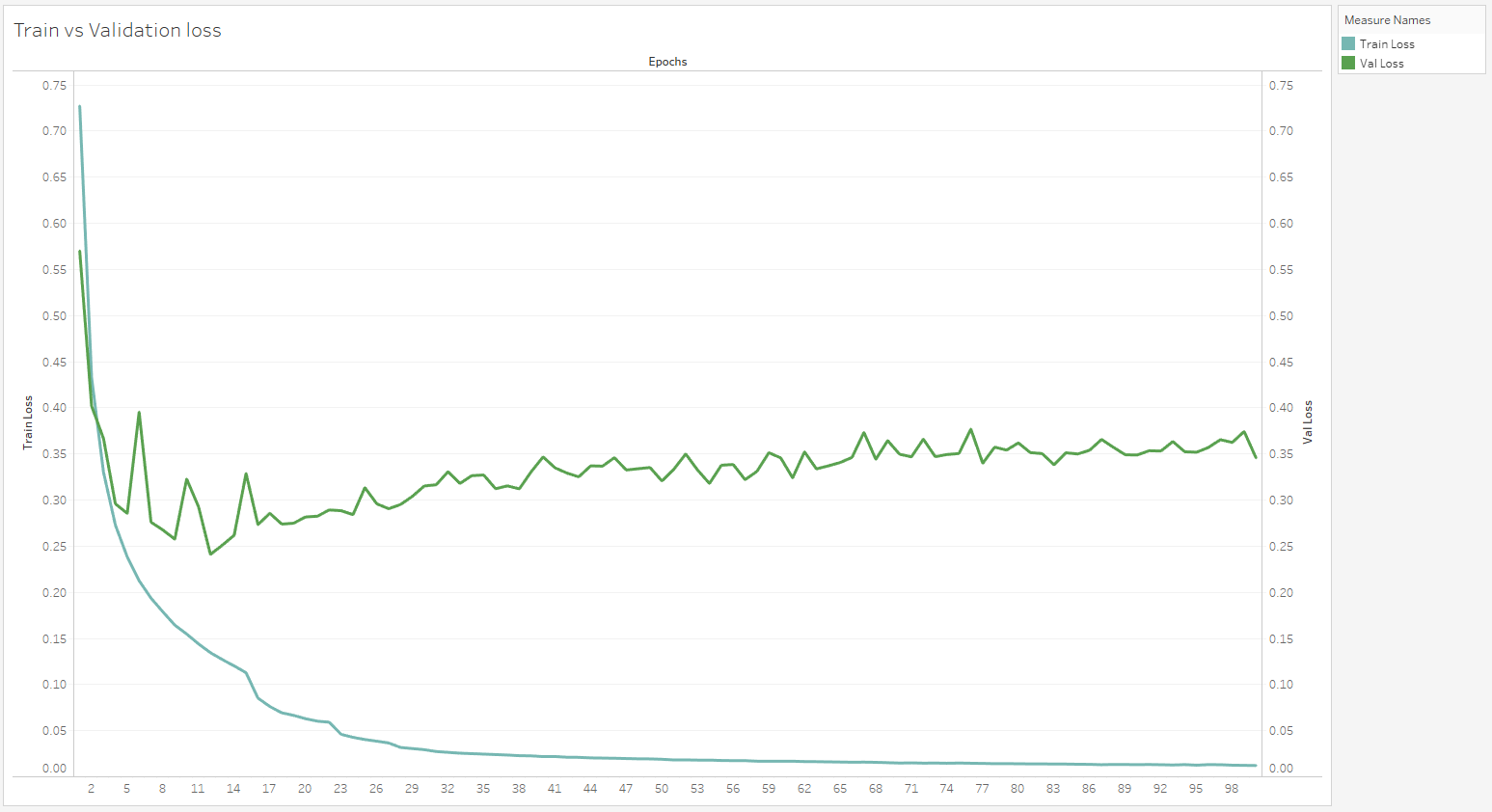
The performance of the augmented model is satisfying. We can already see how augmentation even without tuning can significantly boost the performance of a model.



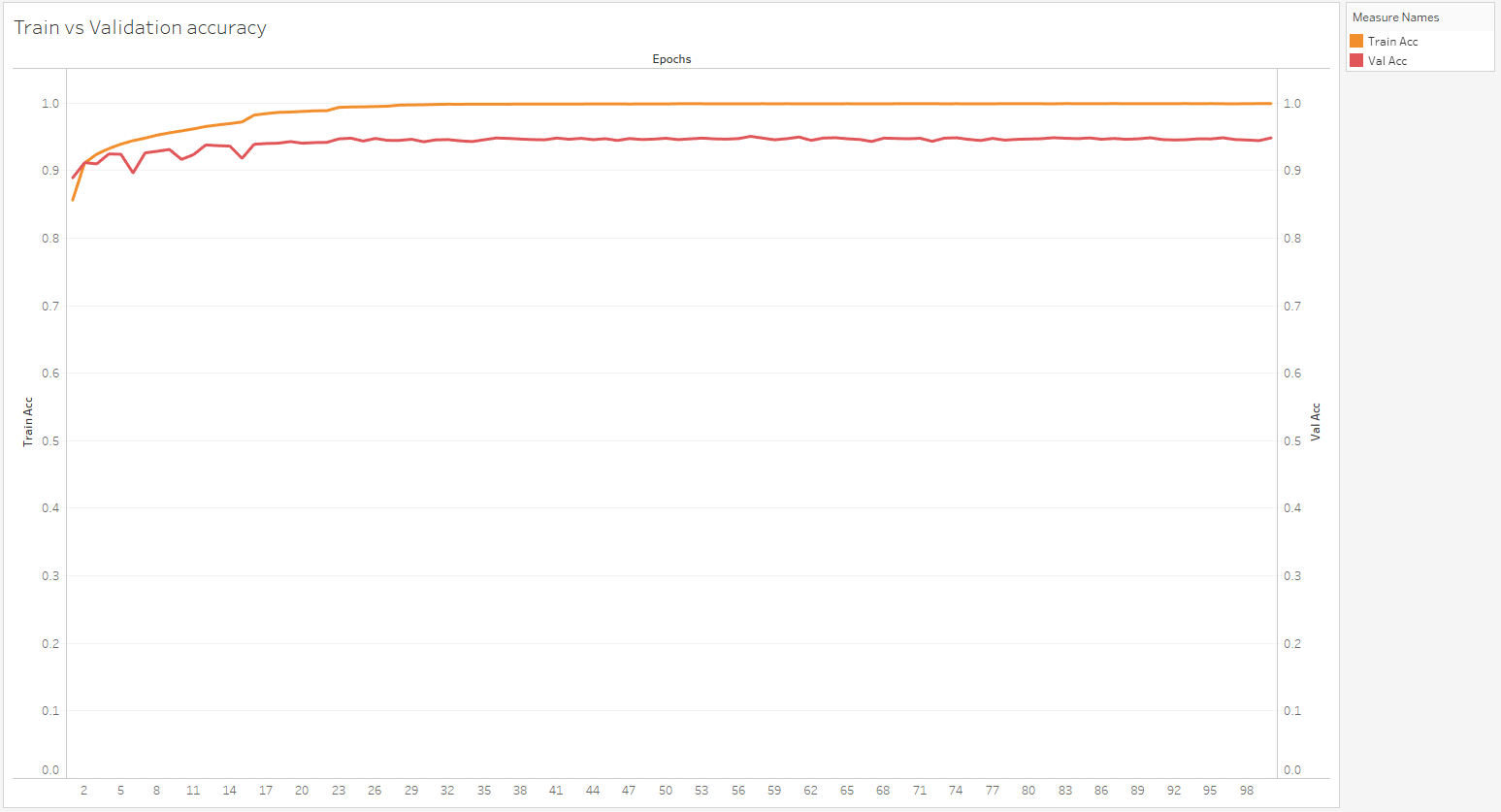
The best validation score was already achieved at epoch 58. After that, we can observe that the model gained next to none generalization ability from the next epochs.



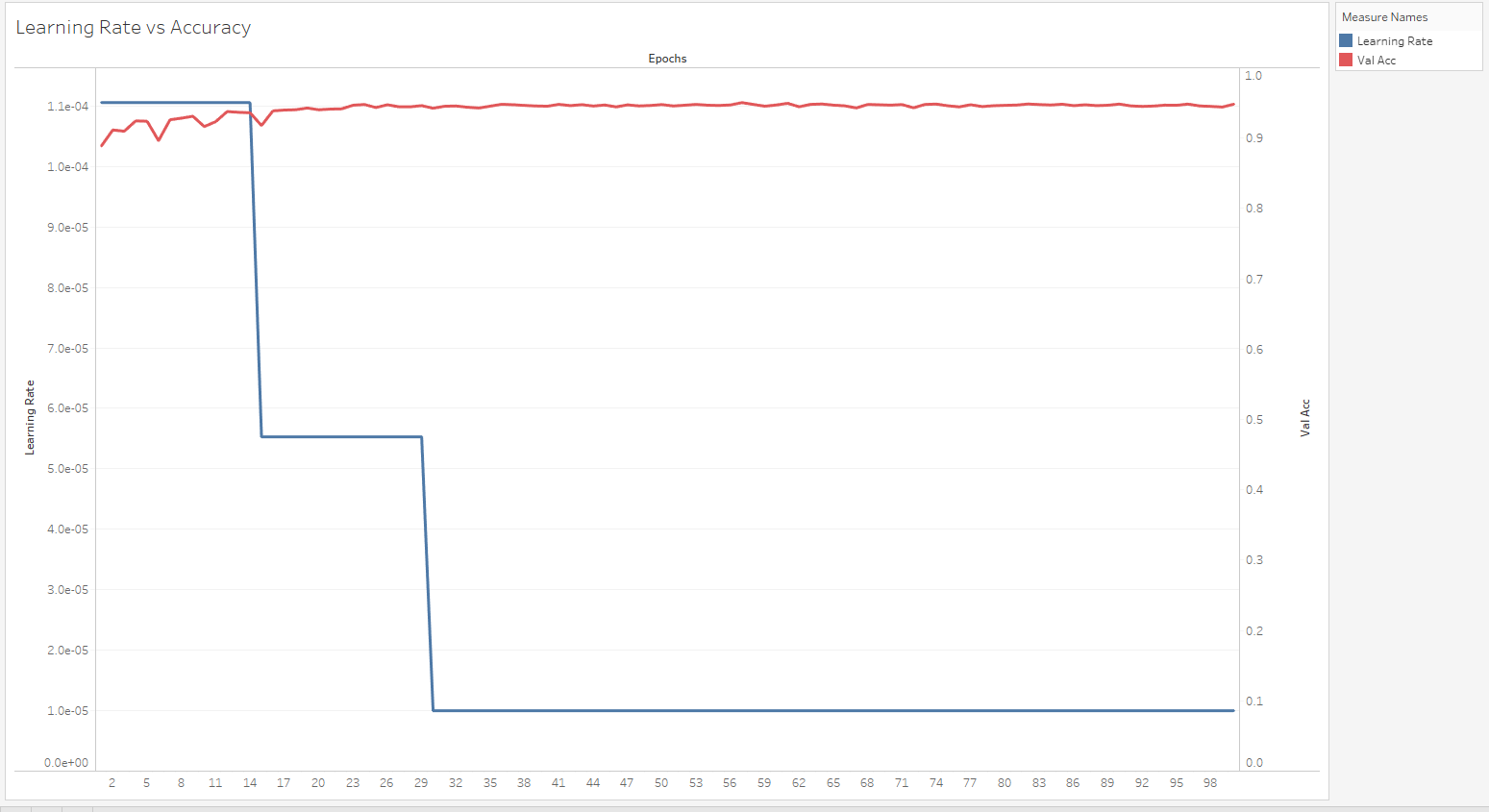
The difference in loss is big but much more interesting is how volatile it was as measure till the end of training. Maybe it would be better monitoring different measures such as f1 score for this model.

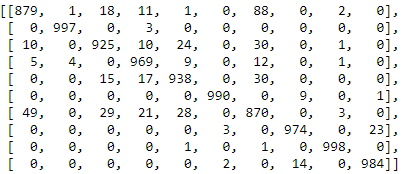


With regards to the accuracy, we can easily find out that there was overfitting again, but not as terrible as it was the first time, we used the model. It is also important to note, that validation accuracy was very volatile at start but in later epochs it started stabilizing.

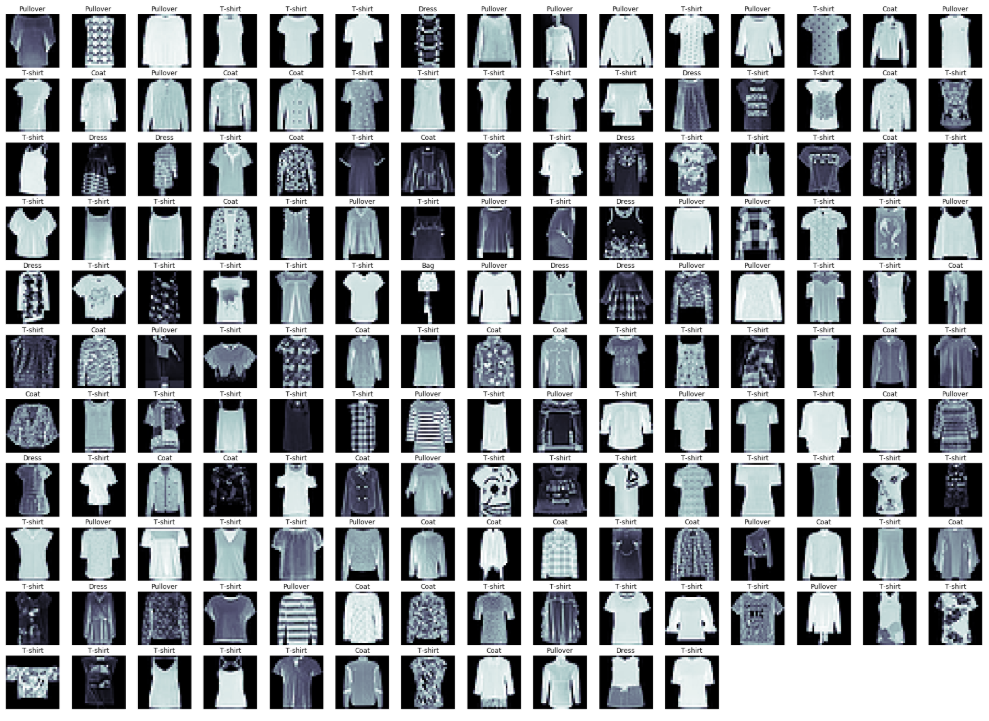


The reason the above phenomenon was caused piqued my interest. Later, I found out that learning rate had to do with it and so, I plotted them together. It is clear that with every reduction by half on plateau, the validation accuracy becomes less volatile.





Now it is time to analyze which individeal item proved the more troublesome to classify. It is clear that we have many items in categories t-shirt, pullover, dress and coat that were confused with each other. Most of them were other categories that got classified as t-shirts. This is expected as they share a lot of common characteristics.



Here it seems that the mistakes were clothes with no clear pictures, bulky dresses and t-shirts that look almost similar as shirts

**Section 3: State of the Art**

In the previous section, we tuned our basic model and added basic augmentation, in order to achieve a very good accuracy in test set. While the results were satisfying, our model still overfitted. This implies that there is still room for improvement.

In this session we will explore and implement some of the more advanced techniques in an attempt to reduce overfitting and increase the overall score. However, the objective here is not to achieve state of the art results, but to use and understand the techniques. As a result, there will not be a tuning session as we will just run one model that combines the techniques bellow.

All the hyperparameters for which I will not cover a better alternative, will be kept the same as in section 3.

Learning rate

Many tools have been developed in order to standardize the values of the ever-important learning rate.

**Learning Rate Warmup:**

As described earlier, weight initialization is very important in neural networks, as it helps us get good models who, converge fast. However, starting from maximum learning rate can significantly distort our initially good weights. Learning rate warmup solves this problem by starting with a small learning rate and gradually increasing it to the specified amount. This period as a rule of the thumb is usually for 1/20-1/10 of the original epochs.

Current epoch\* (Learning Rate)/(Num\_of\_Initial\_Epochs)

**Linear scaling learning rate:**

The idea behind this technique, is that the larger the batch size, the most likely it is to get a representative sample of the population in each batch. As a result, the variance between batches decreases and we can use a larger learning rate without affecting the CNN generalization capabilities. The opposite is in effect for small batch sizes.

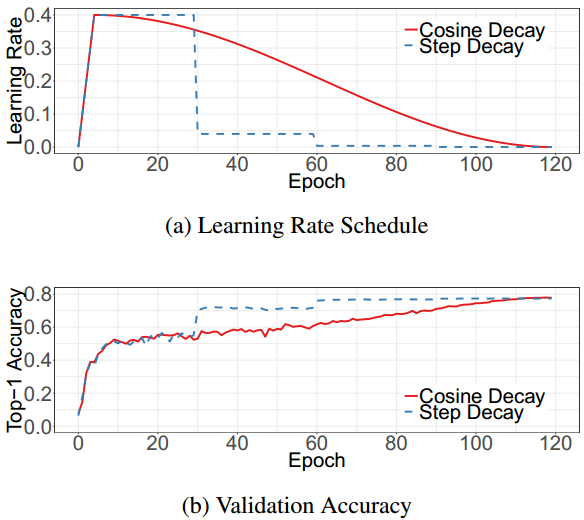
Learning rate is set using the formula: 0.1 x (Batch\_size)/256

**Cosine Learning Rate:**

Instead of using Step decay where the learning rate is decayed in “steps” at fixed batch numbers, the paper suggests using a cosine scheduled learning rate as follows:

Learning Rate (Cosine\_Decayed) = Initially set Learning rate \* 0.5 \* ( 1 + cos( Current\_Batch \* (Pi))/Total\_Num\_Batches))

This usually decreases the learning rate in each epoch by around 0.01% to 0.1% in the later epochs.



Picture (a) is what we will create, giving as smoother and maybe better performance throughout all the epochs.

**Number of epochs**

This time they will be set to 100 as in the previous example.

We will not use early stopping this time as:

1. it may end abruptly in early stages the training if set wrong
2. cosine decay is in effect which will in theory produce better results the longer we train
3. more advanced regularization and augmentation methods will be used
4. We use ModelCheckpoint anyways, which will save the model which generalizes better in the validation set.

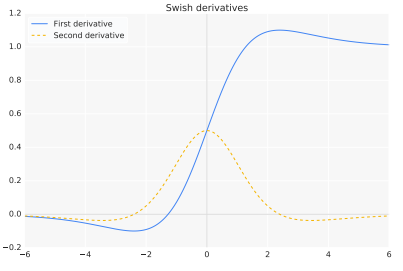
**Number of hidden layers and units**

The same architecture that we found in the previous tuning will be used, mostly for comparison purposes.

**Batch size**

Again, we do not have any better lead than what the previous tuning provided.

**Activation function**

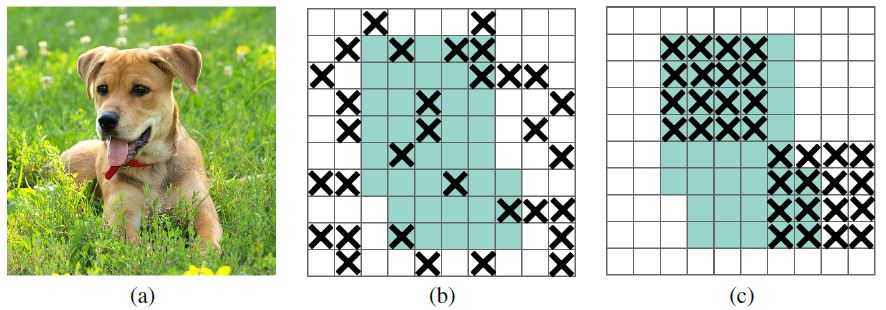


This time we will use the swish function which looks like a sigmoid, but is multiplied by the value of the input x.

Apparently, in many applications, it has shown an increase in performance over ReLU.

**DROPOUT**

A promising technique of 2018 is Dropblock which makes sense for CNNs.



In contrast to dropout (b), which is meant to hide features from the feature map, so that it can learn from a different source, Dropblock drops whole windows with the intent to hide the features most effectively.

Its parameters are:

block size: how many pixels will be the width and height of the dropped window.

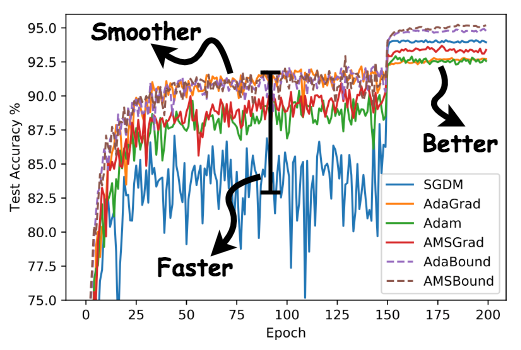
gamma: how many feature maps will be dropblocked.

Keep-prob: what ratio of the feature map will not be dropblocked.

**Weight initialization**

There is not any particular evidence that one initialization formula works better than the other for swish function. So, I chose He uniform as we need again to initialize in the space [0,1] and this is what we used in the previous model as well.

**Optimizer**



When it comes to optimizers, Adabound is a new one, that provides state of the art results in many applications. The concept is not entirely new as it embraces ideas of hybrid optimizers used in the past. More specifically, Adam is considered the go to optimizer for training neural nets allowing them to converge with good performance and relatively fast. However, experimental results have shown that variations of the much slower SGD usually beat Adam in performance after many epochs. This is why, many data scientists used to train the model first with Adam and after some runs continued to train it with the resulting weight with SGD. Adabound is a more sophisticated way to combine the best of both worlds. It starts with a low learning rate and Adam as the optimizer and later switches to higher learning rate and SGD optimizer.

**Augmentation**

The process of augmentation will be left the same, but this time we will use random cropping too.

Random cropping is as the name suggests, a technique that crops some pixels from the edges of the images and then proceeds to rescale them to the original size. Doing so makes the prediction of images that have information in the edges much harder, allowing for even more generalization power.



In order to implement random cropping, I had to extend the imagedatagenerator class and get a ready random cropping function (fortunately rescaling was provided by the class). Then, I had to reimplement the flow function in order to properly use random crop with batches, yielding the correct tuples in the process.

**Remarks**

Google Collab:

For this process I used google Collab as my model will take around 9-10 hours to train. The truth is I encountered some problems in the platform for which I needed to apply workarounds.

1) Reading other python files from google drive (e.g. Adabound, Swish etc.) was tricky, so I decided to toss all classes in the same notebook.

2) The time limit of 12 hours did not allow for the number of batches to be 3 times the training set. So I reduced it to 2 time the original training set.

3)There is apparently a limit in memory used for the output which I exceeded. Nevertheless, I had the model saved to run it in another session and show results.

Techniques:

Learning rate scheduler seemed to interfere with the way Adabound controlled the learning rate which resulted in very bad performance. So, I did not have a learning rate warmup phase or cosine decay, even though I implemented them.

Linear Learning rate was used however, even though it felt wrong. The reason is, the formula gave a 0.00125 learning rate (close to default values) while from the previous sessions we reached the conclusion that 10^-3 or less were better values for the (40,4) model.

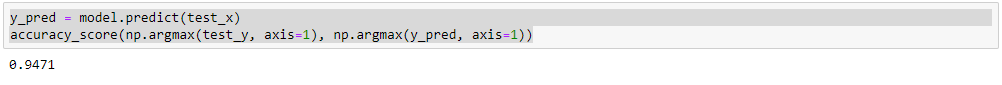
With regards to the number of epochs, early stopping was not used as we now have a whole array of regularization tools and model checkpoint.

In the first experiments, swish function seemed to outperform ReLU in convergence speed at least.

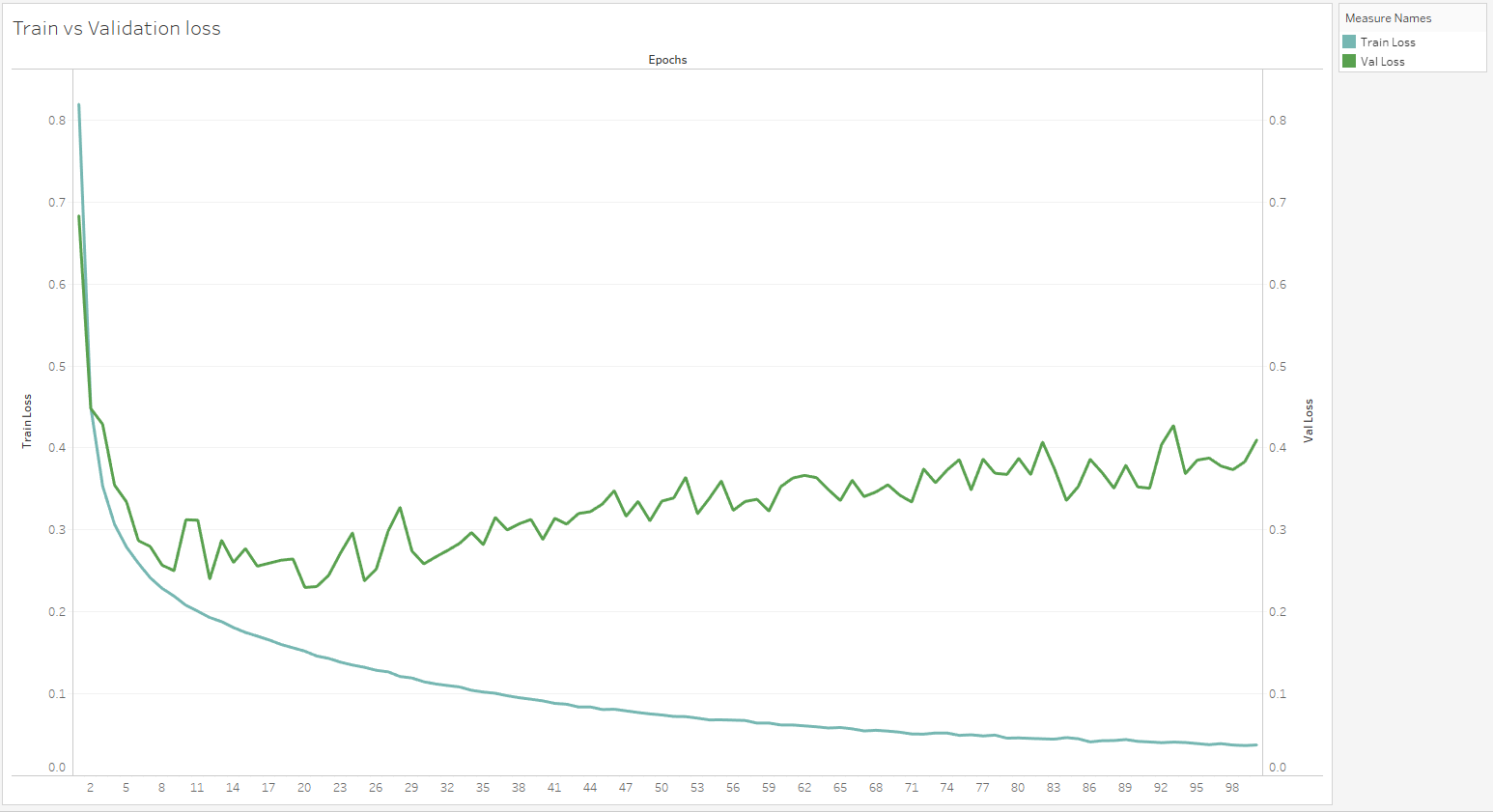
Dropblock dropped the performance by a last margin. I believe a logical explanation would be that there are not a lot of features in a 28x28 image of clothes. As such, it might block the only features of essence, allowing no useful information to be passed through the convolution layers.

**Final model**

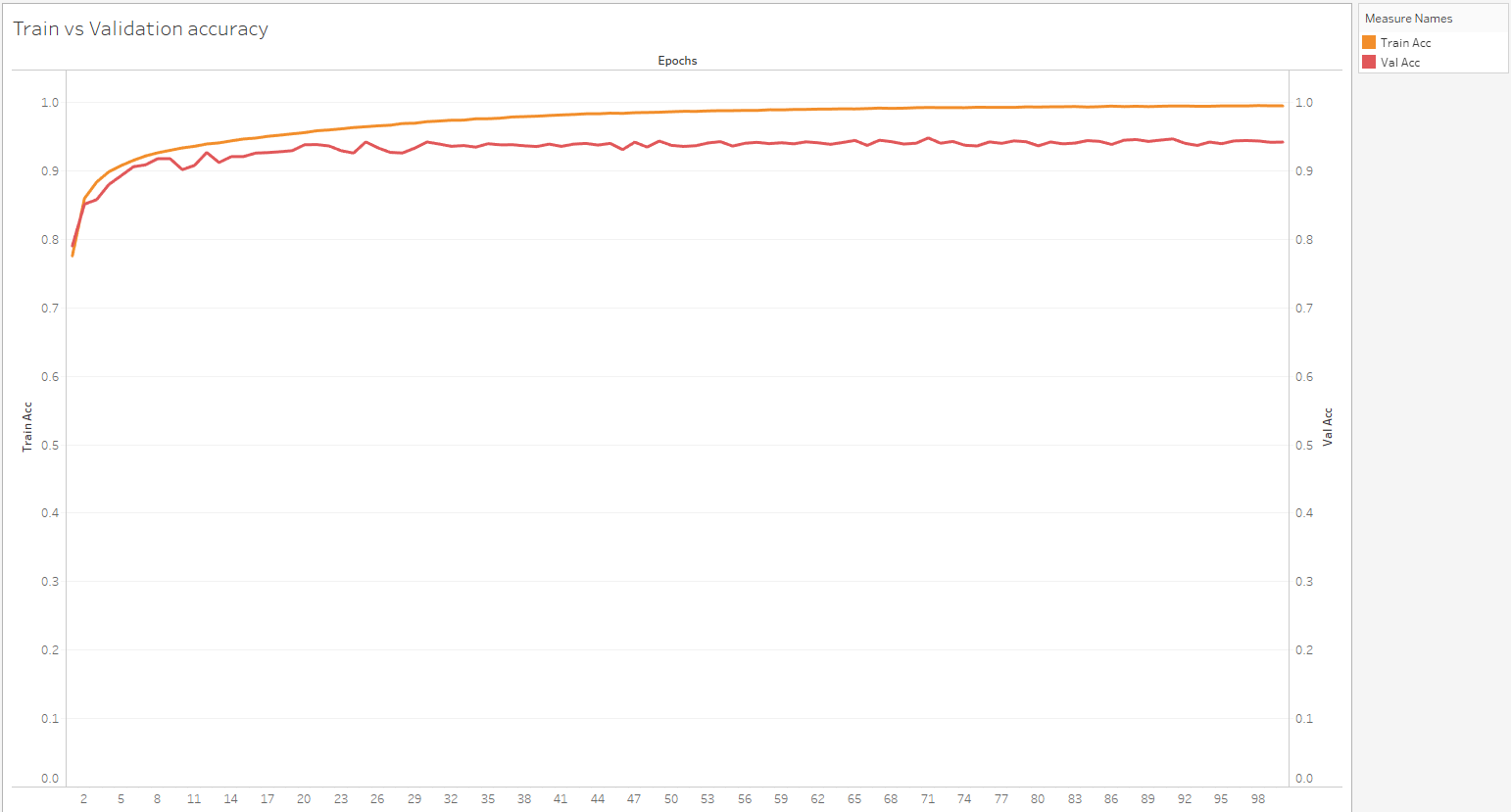
The final model performed well but not exactly as good as the tuned model. This is to be expected as part of it were new features patched together with some values derived from the previous tuning. If I wanted a better performing model, I should had tuned from scratch together with the new features. However, I did not have enough time or resources as it was even heavier than the previous.

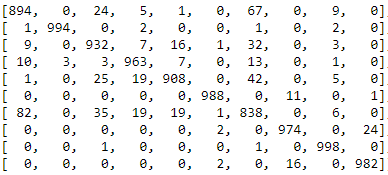


Here we witness something interesting. The validation and training accuracy begin to diverge from each other after epoch 32. My hypothesis is that after epoch 30, the optimization algorithm started shifting to SGD which started fitting very well the training set, but caused overfitting issues. This is why the hyperparameters of Adabound should be tuned as well.

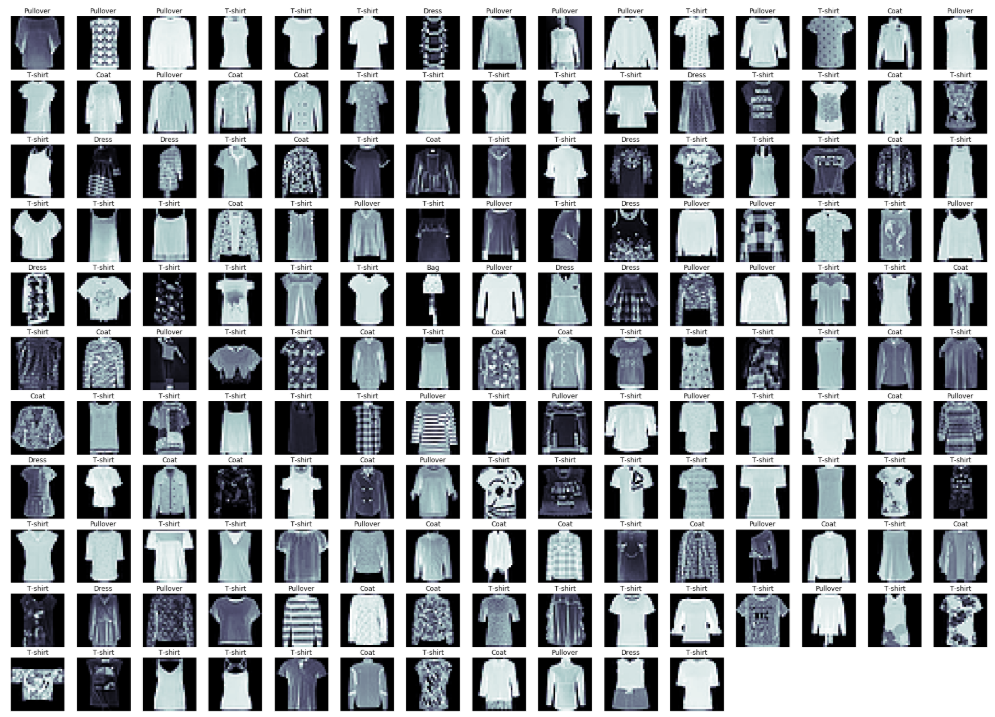


The same is depicted in the accuracy pair. We can see that the volatility faintly exists throughout the whole set now.





It is not surprising that the problem lies in the shirts again but magnified.



My conclusion is that we need to micromanage the model, especially through augmentation in order to target specifically the entanglement of the upper body categories. Reducing this confusion seems to be the key in achieving higher accuracy

**Section 4: Further actions**

There are a lot more techniques to experiment with in the future. Below are some techniques that I would like to use but the resources were prohibitive as well as how I would implement them.

**Ensembles:**

In neural networks, it is the concept of having multiple models that perform almost the same, decide together which label to give in each example. This makes the overall decision more robust and increase generalization. However, my model was already too high in precision and it would be difficult to make a different model, with the same performance. This is the committee of models approach.

What I would like to do instead, is train the model with a lot of different seeds and then keep the versions that performed better. Then, do model averaging to get a model that is more robust to randomness. (Which apparently was an issue observed throughout the whole project).

Unfortunately, the model became too heavy and I could not allocate any more time in the project.

**Pretrained models:**

Use another model not covered in the first section like MobileNet or XCeption. They could also be used to create an ensemble together with the model we developed. Again, they would need tweaking in order to perform as the WRN and thus they are not of much use now.

**Random erasing:**

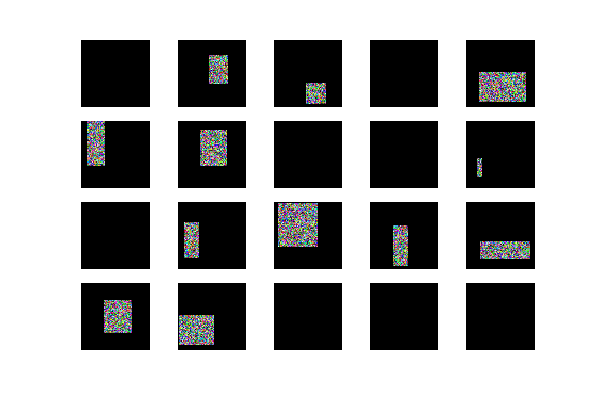
A technique complimentary to random cropping. Instead of cropping the edges, we mask an area with random values. I can see why it can work together with random crop but I imagine it causing the same issues as Dropblock. This is mainly due to the fact that we have 28x28 images with limited features to set them apart from each other. Thus, I was not motivated to use it in this dataset.

**Pyramid Net:**

I did not see any reason on implementing pyramid net in this model, as the only thing I need to do is tweak the WRN, which did not provide any educational value. However, it is considered a powerful tool for classification combined with some other advanced techniques I described.

**Auto-augment:**

I provided no tuning in the image generator which is considered a bad practice. It would be nice to automate the process in the future using auto-augment or even just Hyperas.



**Conclusions**

In this project we started from the basics of CNNs, progressed through various concepts and techniques of the field, finally scratching the state of the art. It seems that nowadays ResNets and variations rule the scene of CNNs. However, one problem I kept encountering in my project is overfitting. The WRN kept learning almost perfectly the training set almost perfectly no matter what. Using techniques to reduce overfitting seemed to not be able to change that. Nevertheless, they helped the model reach better generalization capabilities. Maybe this is why regularization and augmentation are now more prevalent than ever.