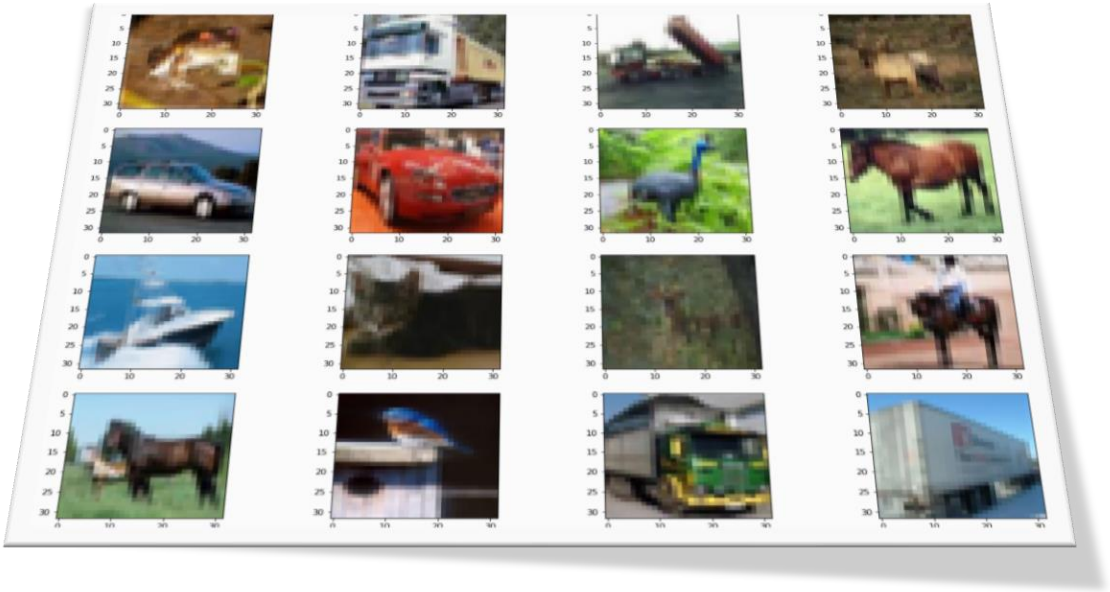
MSc IN DATA SCIENCE
UNIVERSITY OF PELOPONNESE –
NCSR "DEMOKRITOS"

Deep Learning

Assignment 1: A Simple Classification Network



A report by:
Georgios Touros (dsc18023)

November 2019

Experimental Setup and Data Pre-processing

Our assignment is a multi-class image classification task on the cifar-10 dataset, utilising only fully connected feed-forward neural networks. The dataset has 10 classes and 60 thousand samples, split into train and test with a 5:1 ratio.

By plotting the distribution of the labels both in the train and the test set, we see that they are uniformly distributed, therefore there are no class imbalances we need to address in the pre-processing stage. The only pre-processing that we apply is transforming the labels to a bag-of-labels, i.e. to 10 binary columns, so that we will be able to work with categorical cross-entropy as loss function. We also re-scale all the attributes to the [0,1] range, because the weights of the model are initialized to small random values and updated via an optimization algorithm in response to estimates of error on the training dataset, therefore the scale of the input is important[1]. We initially abhor from augmenting the dataset using random flips or rotation, because the images (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks) seem to be usually aligned to the horizontal axis, therefore we initially felt this would add more noise.

We choose to keep the categorical cross-entropy loss function in the initial experiments, as it seems to be a community standard for this dataset. We found a paper[2] that challenges this, wherein the writers were applying hinge loss on convolutional networks with good results. We experiment briefly with it in the final section.

According to a benchmarking website[3] the worst accepted performance of a deep neural network on the cifar-10 is dataset is 75.86% using Shallow Convolutional Networks. The state of the art is Giant Neural Networks using Pipeline Parallelism (with 99% accuracy). We will also use accuracy as an evaluation metric, in order to be comparable, even though precision and recall would be preferable in such a classification task. Since we will not be using convolutional networks, and given the scope of the exercise, we will set a goal of **55% accuracy** on the test set, just marginally better than chance, but quite better than the initial code we were given in the assignment (more on that later). We will also be mindful of execution time, given that we do not have access to GPUs. Therefore we set a maximum running time of **20 minutes** for each experiment. Nevertheless, we do not maximise efficiency in this assignment, therefore, we will accept improvements on accuracy, even if this costs considerably more in execution time.

For a summary of model results, please refer to Appendix A. The visualisations on the results of each experiment are gathered in Appendix B.

Baseline Model & Depth-Width experimentation

We run the first test using the model as was given to us in the assignment. It's a simple feed-forward with 2 dense layers with ReLu activations and a final SoftMax activation layer, that runs on batches of 128 samples, for 40 epochs. It achieves a test loss of 1.409 and accuracy on the test set 49%, which is not that bad, given that it's quite fast (just over 1.5 minutes) and very simple. Even though it generalises well (accuracy on the train set is 54%), it underfits noticeably.

We increase the depth, initially by one layer (model01) and then increased the width of each layer from 100 to 200 (model02). Each time we saw some slight improvement, both on test and train accuracy. We decided to further increase the capacity, and we reached a total of 5 dense layers of 400 units each (model03). The test accuracy increases to 52.3%, but we started noticing an **increasing divergence** between train and test errors. Given our limited computational resources, we decide not to further increase the size, opting for some experimentation on the optimizer.

[1] Goodfellow, Bengio, Courville: Deep Learning, MIT Press 2016, chapter 12.2.1
[2] Janocha, K., & Czarnecki, W. M. (2017). On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*.
[3] https://benchmarks.ai/cifar-10

Optimizer selection and Regularization

The models so far seem to underperform when it comes to learning. We had been using a Stochastic Gradient Descent optimizer for the back propagation, with a static learning rate of 0.0099. We decide to start (model04) with a larger initial learning rate of 0.03 and introduce a decay of 1×10^{-9}. We get some slightly better results (test accuracy of 52%) but the problems persist. We also introduced a Nestorov momentum of 0.01 (model05) which further improved the accuracy to 53%.

We decided to experiment with adaptive learning rates, starting with rmsprop in its default setting (model06), which resulted to a highly unstable model and worse performance. We then tried out the Adam optimizer (model07), which resulted to extreme overfitting (train accuracy almost 90% and test accuracy of 46%). Nevertheless, this optimizer seems to increase the learning capacity of the model considerably. Therefore, we decided to keep this and experiment with various regularization techniques in order limit this capacity and improve generalization.

Our first attempt to regularization was using the 'kernel regularizer' function on each layer, which applies an L2 regularization on the weights of each layer (model08). This "fixed" the generalization problem, but there still wasn't much improvement in the accuracy overall (51%). We still needed a more effective regularization, and some more learning capacity. We decide to increase the depth of the network by one Dense layer and add a Dropout layer for every two fully connected layers (model09). This seemed to work, as we achieved 53% accuracy on the test set, and an acceptable generalization behavior. This made us curious as to whether we could further improve by removing all L2 regularizers and add a Dropout layer after each fully connected layer (model10). Obviously, this was a bad idea, as the accuracy of the test set was now higher than the one of the train set (47% for the test set and 45% for the train set). It seems that we have handicapped the learning capacity of the model too much, so we revert to the setting of model09.

Early Stopping - Setting Number of epochs and Batch Sizes

According Goodfellow et al.[4] in the cases where the train error decreases consistently over time, while the validation error forms some asymmetric U-shaped curve, it is often the case that early stopping helps. The idea is that we return to the parameter setting that had the lowest validation set error and iterate on those until we manage to improve upon them. If not, we output those best recorded parameters. We set the Callback function to monitor validation set accuracy (as this is the metric we're optimizing on) and set the patience parameter to 10 epochs, so that we don't stop due to some slight fluctuations. We also increase the number of epochs, so that there is enough padding for the patience parameter to take effect (model11). The model stops in epoch 31, with a test accuracy of 52.6%, which was an improvement in terms of accuracy and efficiency.

As we still experience a small learning capacity, we decide to increase the batch size. Smaller batches can offer a regularizing effect, due to the noise they add to the learning process[5]. Therefore, by increasing the batch size to 256 (model 12) we hope to achieve a higher accuracy rate. The experiment is successful, as the model stops in epoch 31 and achieves a test accuracy of 53.3%, the highest so far.

Loss Functions, Batch Normalization and Data Augmentation

Before wrapping up, we did 3 final experiments. First, we tried hinge loss on model12, as it has been proposed as a good alternative (model13). The results were unacceptable (34.6% test accuracy) for 60 epochs. Nevertheless, early stopping didn't take place, which might indicate that this is a slower learner that needs more epochs to converge. Also, this seemed to be the most consistent in terms of generalization. As we will not be using convolution, we decide to leave it at that.

As model12 needed to learn faster, we decided instead to add a Batch Normalization Layer in front of the network. As this could lead to high overfitting, we follow that with a 50% Dropout layer, and

[4] Goodfellow, Bengio, Courville: Deep Learning, MIT Press 2016, chapter 7.8
[5] Goodfellow, Bengio, Courville: Deep Learning, MIT Press 2016, chapter 8.1.3

continue with the network in the usual fashion of 10% Dropouts after 2 fully connected layers. After some tweaking, in order to eradicate the fluctuations of the test accuracy, we increase the rate of the Dropout layers. The final model (model14) also has some regularized layers, using L2 regularization. The result is comparable to model12 but took more time to train.

As a final experiment, given that the algorithm doesn't seem to be able to learn and generalize adequately, we decide to add more data to the training model, using keras' ImageDataGeneration class for data augmentation. We initiate it with random flips on the horizontal axis, as well as random rotations of up to 90 degrees, with random shifts of width and height of 10%. This class returns a generator, therefore we have to alter the fit command of the model to a fit_generator command. This also affects the input shape, so we also add a Flatten layer which will allow us to maintain the correct output shape Unfortunately, the model would require approximately one hour for each epoch. This is not acceptable according to the rules we have set; therefore, we abort the experiment.

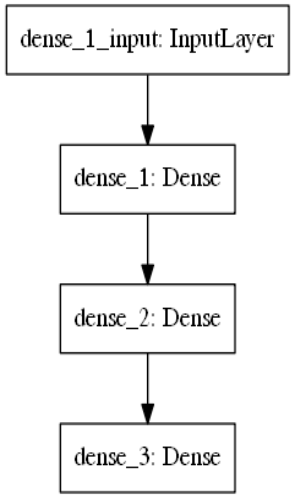
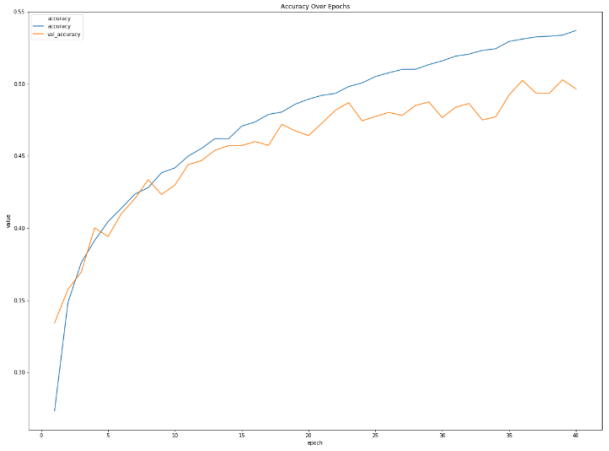
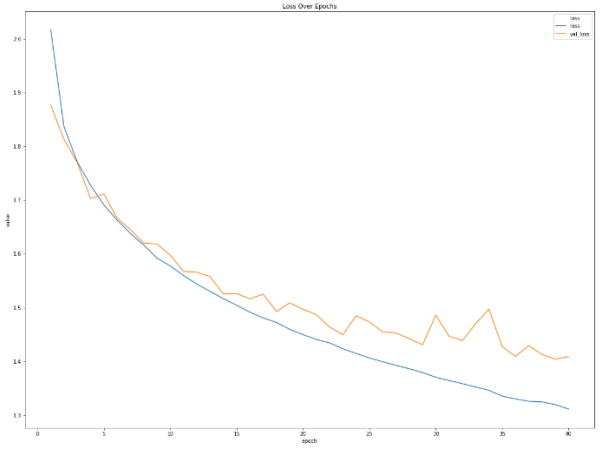
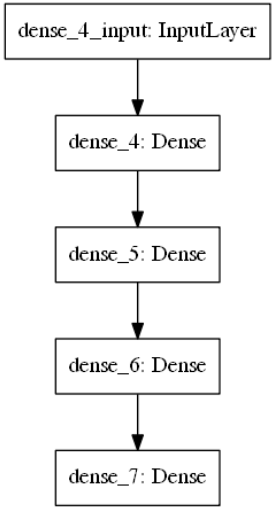
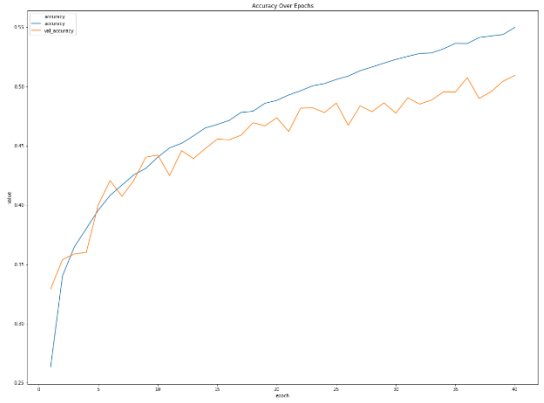
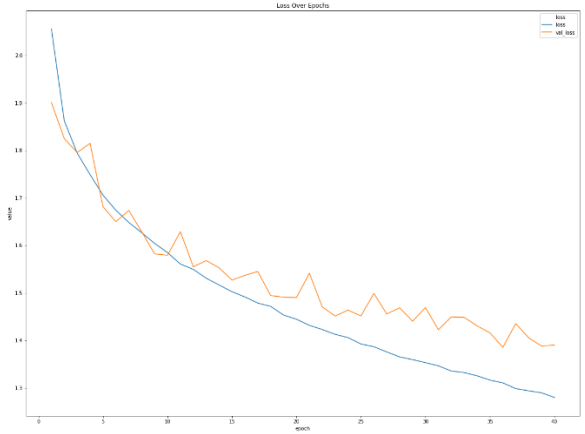
Conclusion

It seems that without convolution, this image recognition task will not go much further. We haven't reached our goal of 55% accuracy on the test set, but we do have a model (model12) that performs slightly better than chance, and trains adequately fast (under 8 minutes).

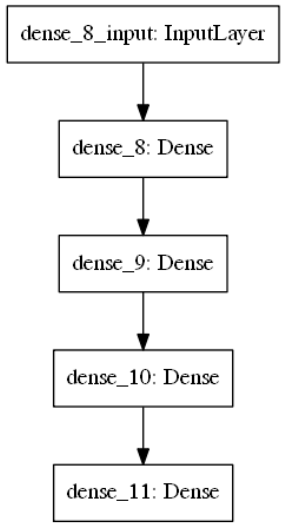
Appendix A – Attempt Results Summary

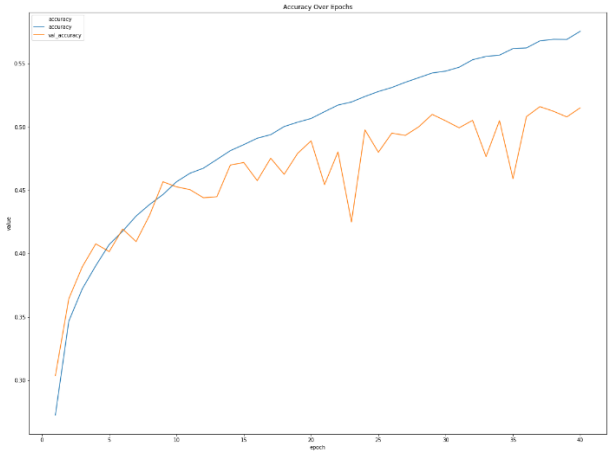
Model Name	**Train Accuracy**	**Test Accuracy**	**Execution Time**
Baseline Model	53.71%	49.67%	1min 32sec
Model01	54.98%	50.93%	1min 39sec
Model02	57.56%	51.52%	2min 26sec
Model03	61.46%	52.31%	4min 53sec
Model04	61.08%	52.03%	4min 50sec
Model05	61.58%	53.68%	6min 18sec
Model06	60.26%	50.32%	6min 55sec
Model07	88.13%	46.91%	8min 52sec
Model08	53.99%	50.85%	10min 37sec
Model09	64.46%	53.25%	15min 30sec
Model10	45.72%	47.02%	23min 10sec
Model11	61.06%	52.45%	10min 59sec
Model12	53.35%	53.35%	7min 26sec
Model13	36.21%	34.66%	13min 4sec
Model14	69.64%	45.92%	10min 20sec

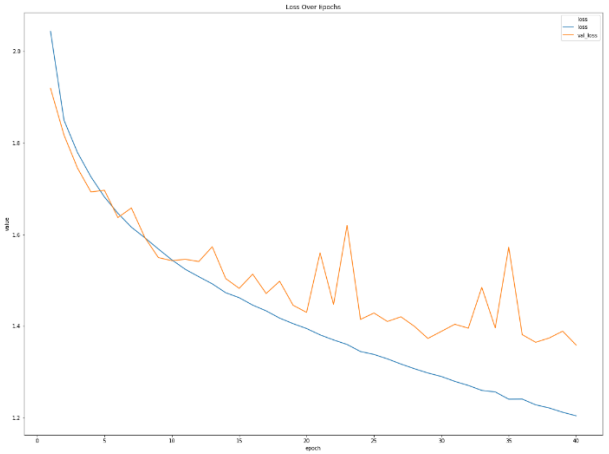
Appendix B – Model Summaries, Accuracy and Loss over Epochs

<table>
<tr>
<td>Baseline Model
 
</td>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>Model01
 
</td>
<td> 
</td>
<td> 
</td>
</tr>
</table>

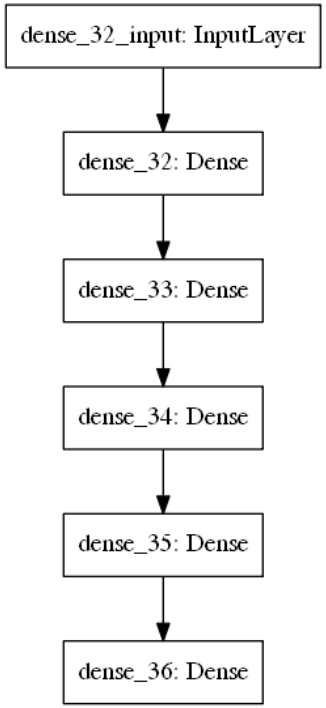
Model02

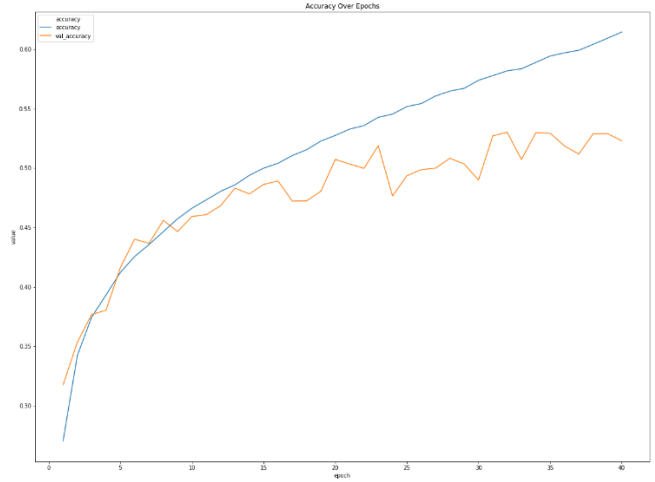


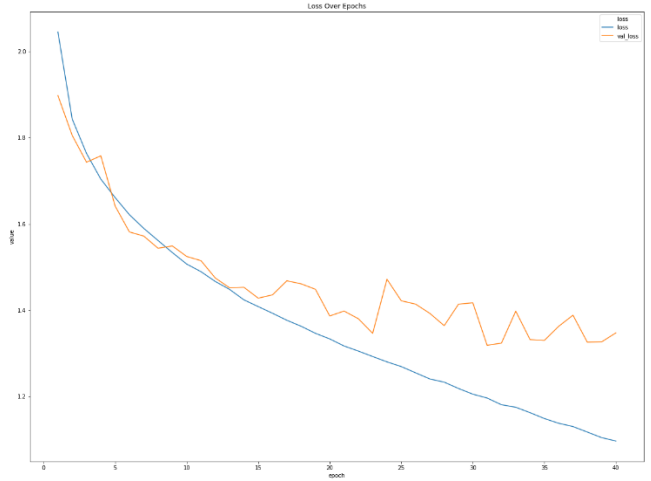




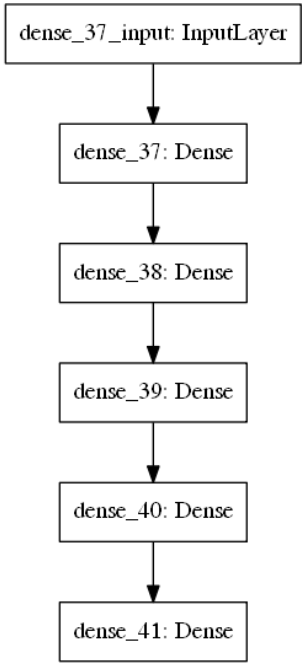
Model03

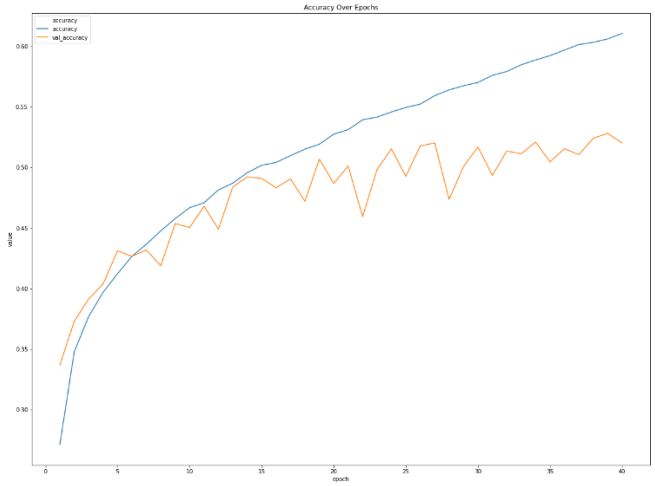


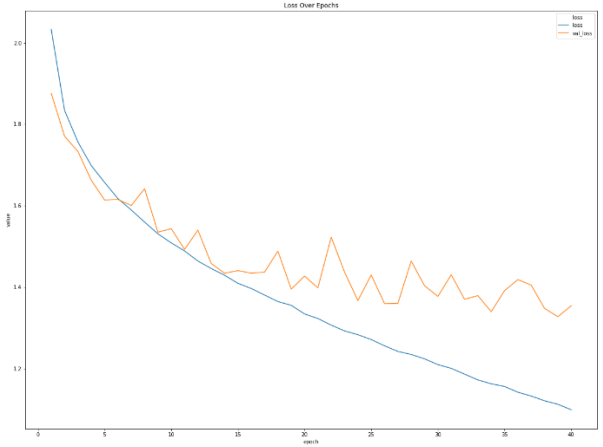




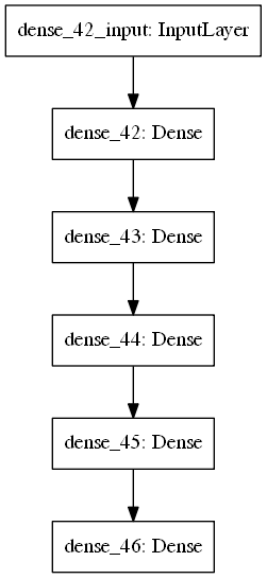
Model04

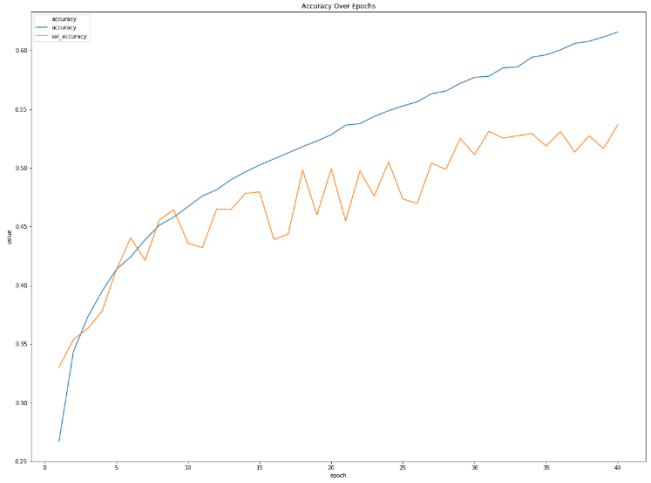


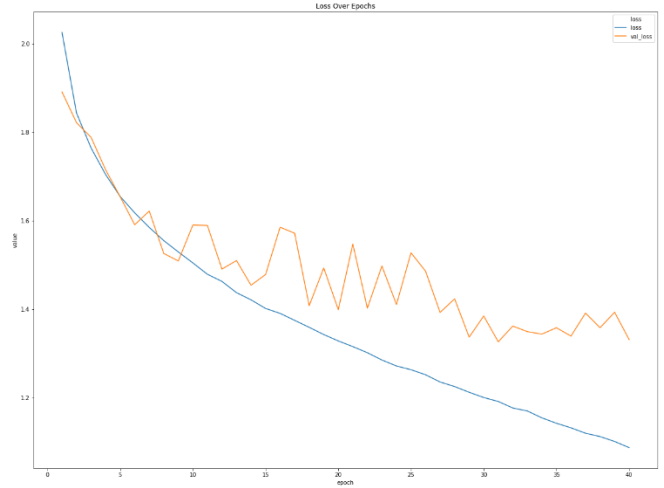




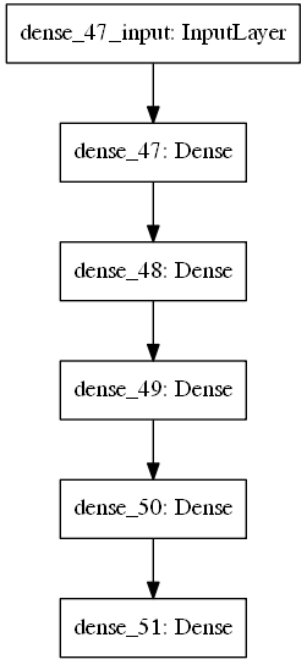
Model05

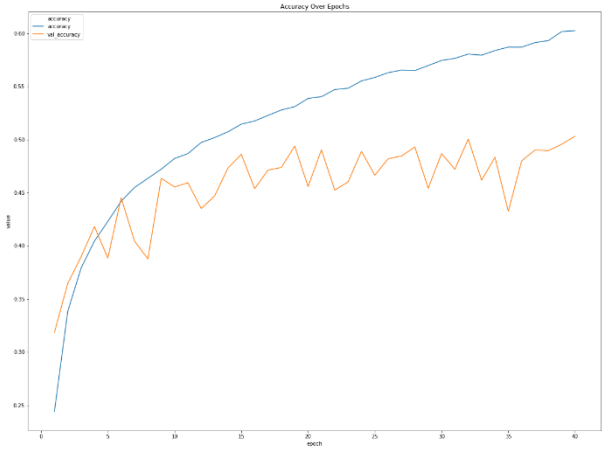


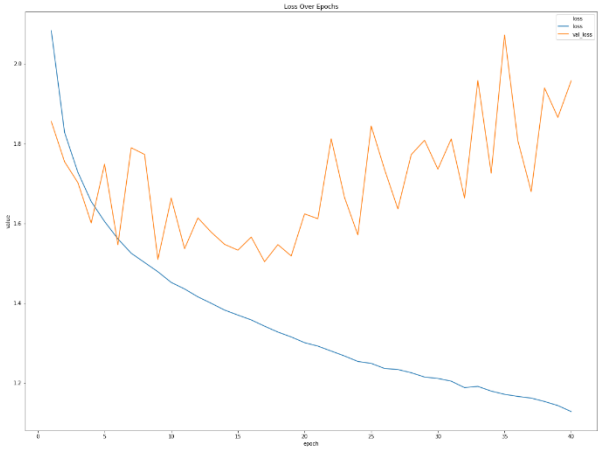




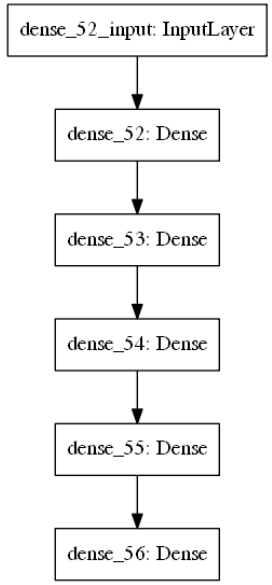
Model06

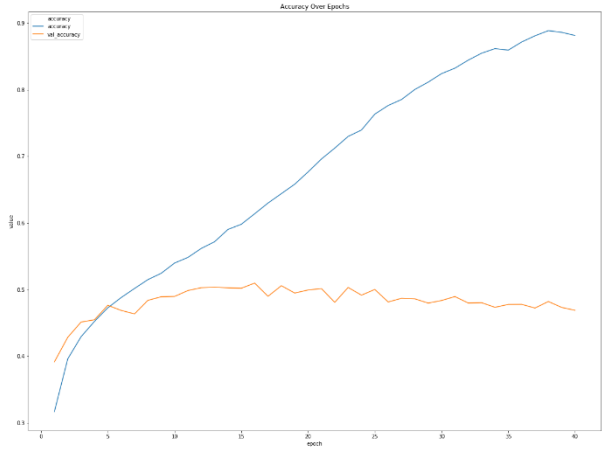


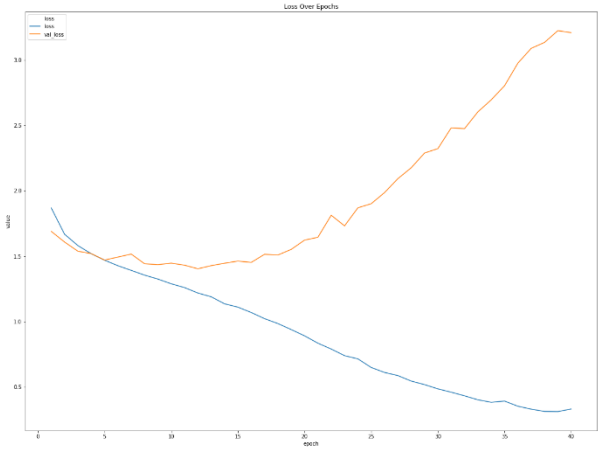


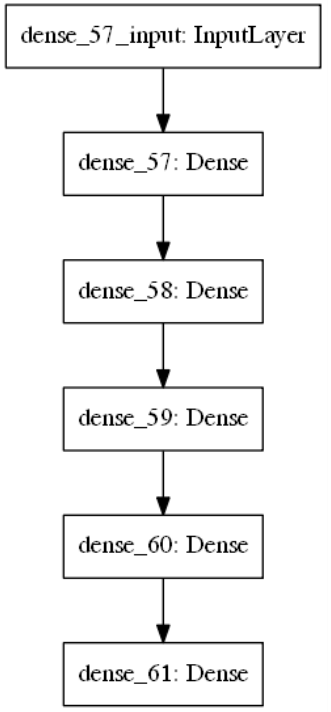


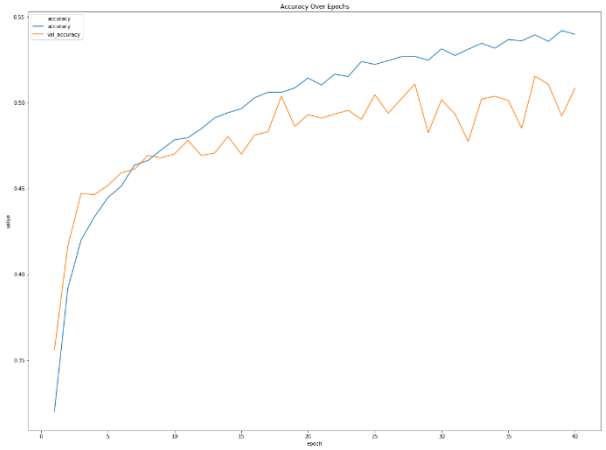
Model07

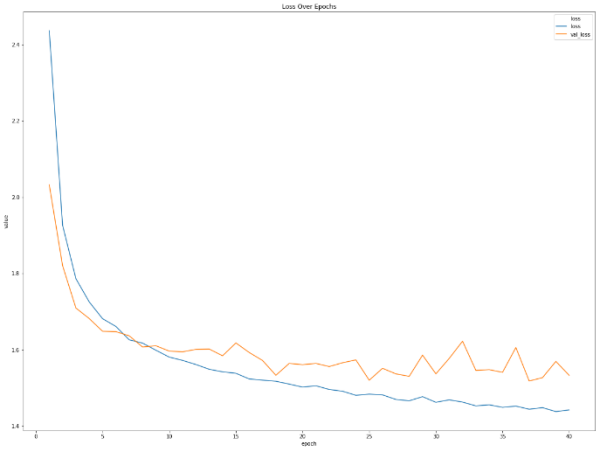




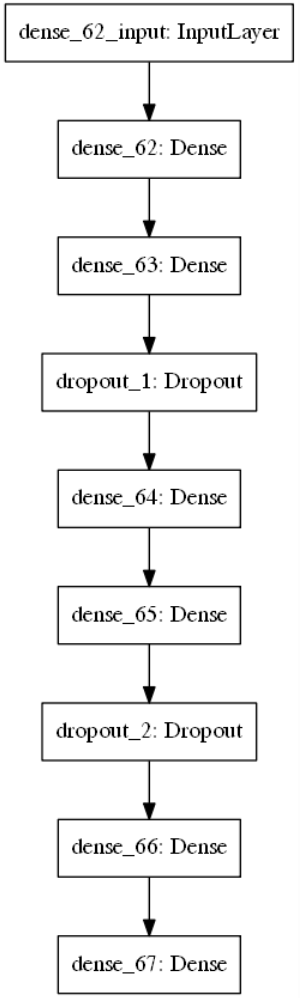


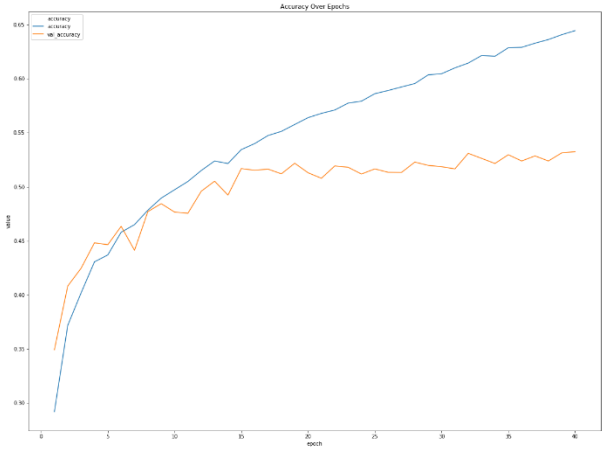

Model08
dense_57_input: InputLayer
dense_57: Dense
dense_58: Dense
dense_59: Dense
dense_60: Dense
dense_61: Dense

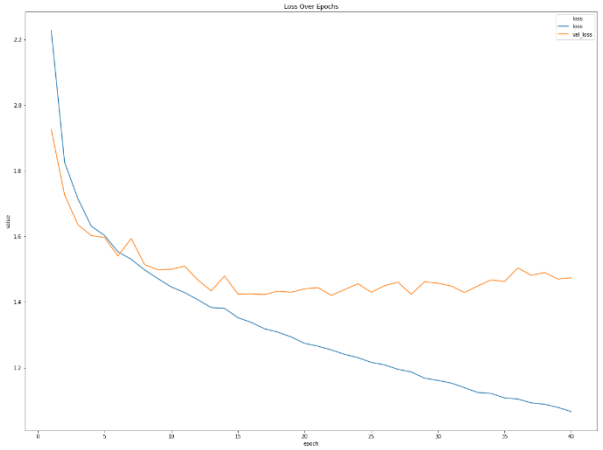

Accuracy Over Epochs
accuracy
val_accuracy
epoch


Loss Over Epochs
loss
val_loss
epoch

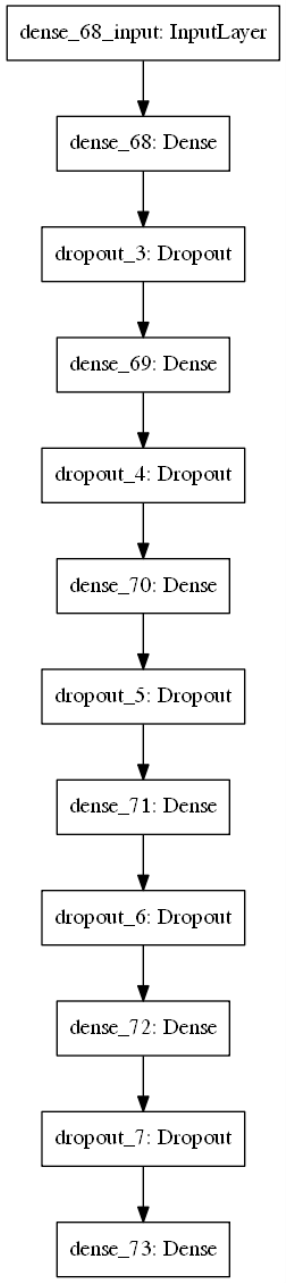
Model09

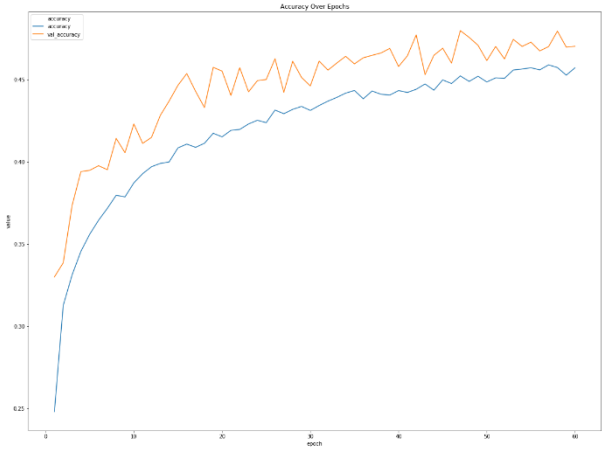

dense_62_input: InputLayer
dense_62: Dense
dense_63: Dense
dropout_1: Dropout
dense_64: Dense
dense_65: Dense
dropout_2: Dropout
dense_66: Dense
dense_67: Dense

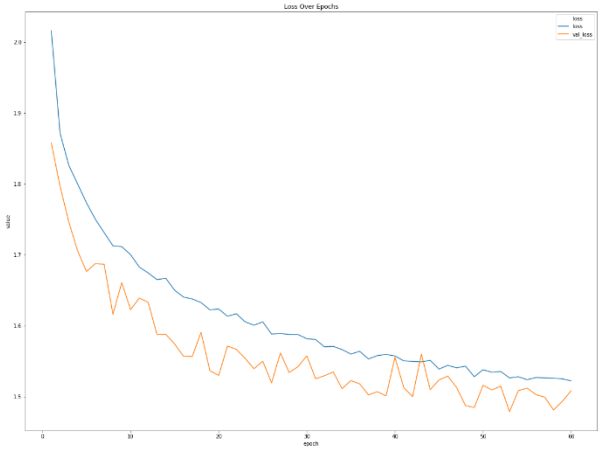

Accuracy Over Epochs
accuracy
val_accuracy
value
epoch

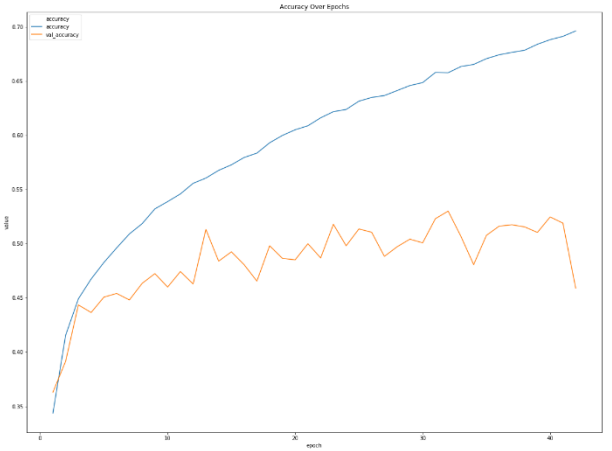

Loss Over Epochs
loss
val_loss
value
epoch

Model10

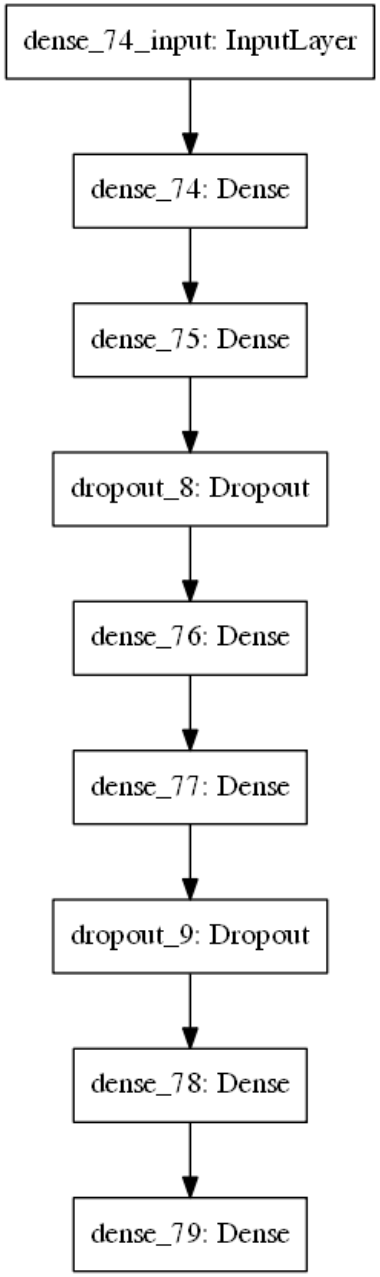

dense_68_input: InputLayer
dense_68: Dense
dropout_3: Dropout
dense_69: Dense
dropout_4: Dropout
dense_70: Dense
dropout_5: Dropout
dense_71: Dense
dropout_6: Dropout
dense_72: Dense
dropout_7: Dropout
dense_73: Dense

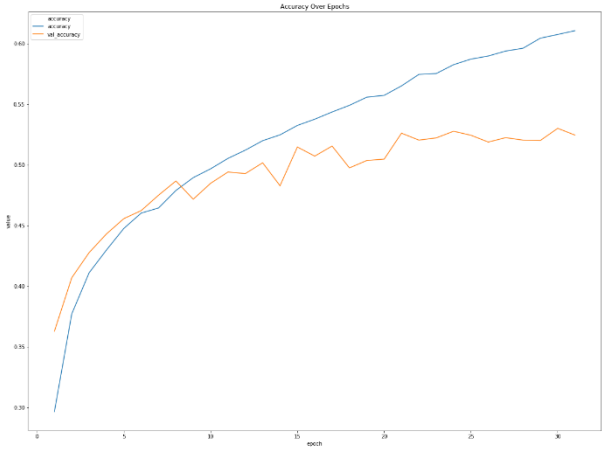

Accuracy Over Epochs
accuracy
val_accuracy
epoch

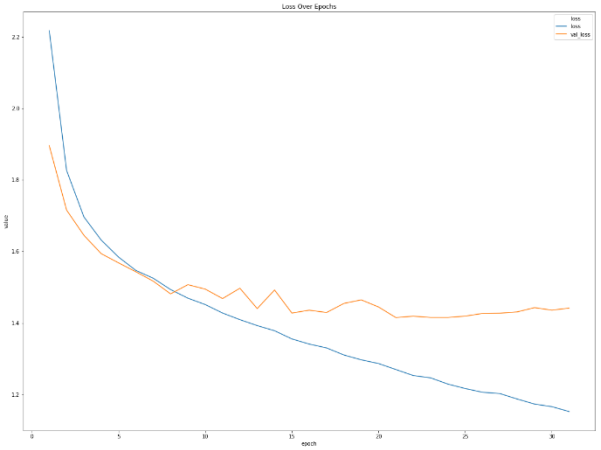

Loss Over Epochs
loss
val_loss
epoch


Accuracy Over Epochs
accuracy
val_accuracy
epoch

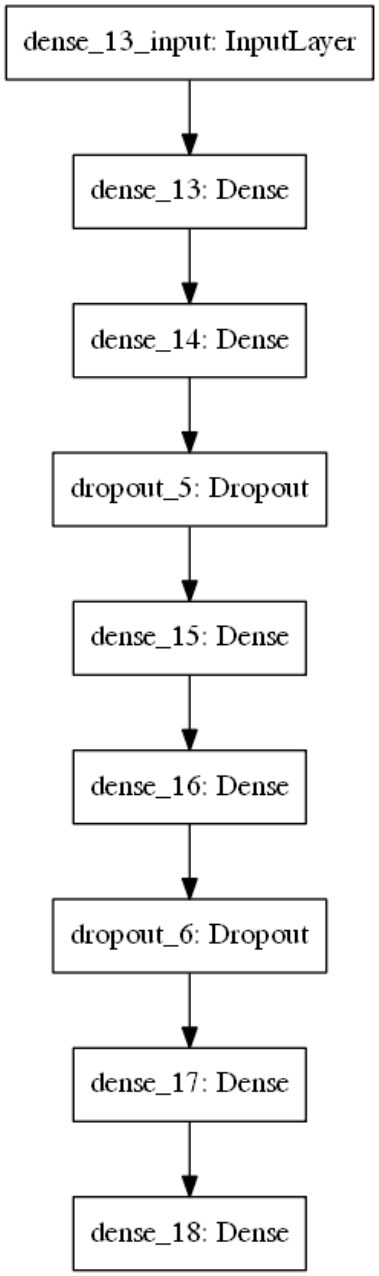
Model11

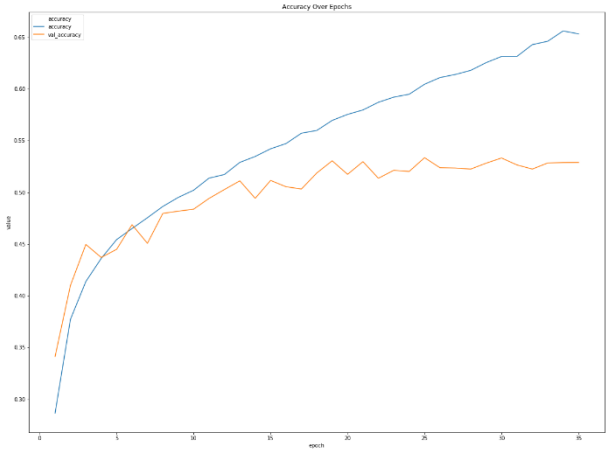

dense_74_input: InputLayer
dense_74: Dense
dense_75: Dense
dropout_8: Dropout
dense_76: Dense
dense_77: Dense
dropout_9: Dropout
dense_78: Dense
dense_79: Dense

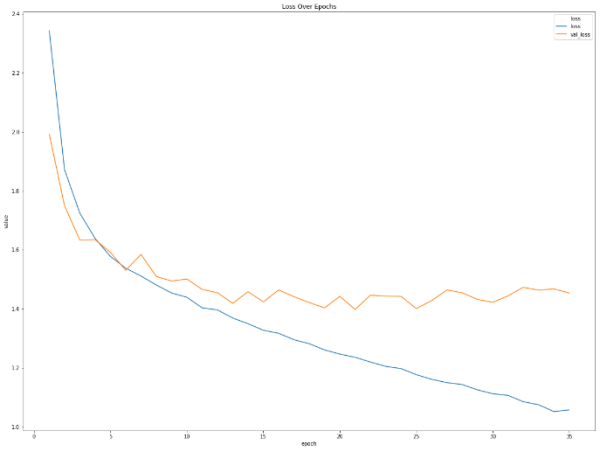

Accuracy Over Epochs
accuracy
val_accuracy
epoch


Loss Over Epochs
loss
val_loss
epoch

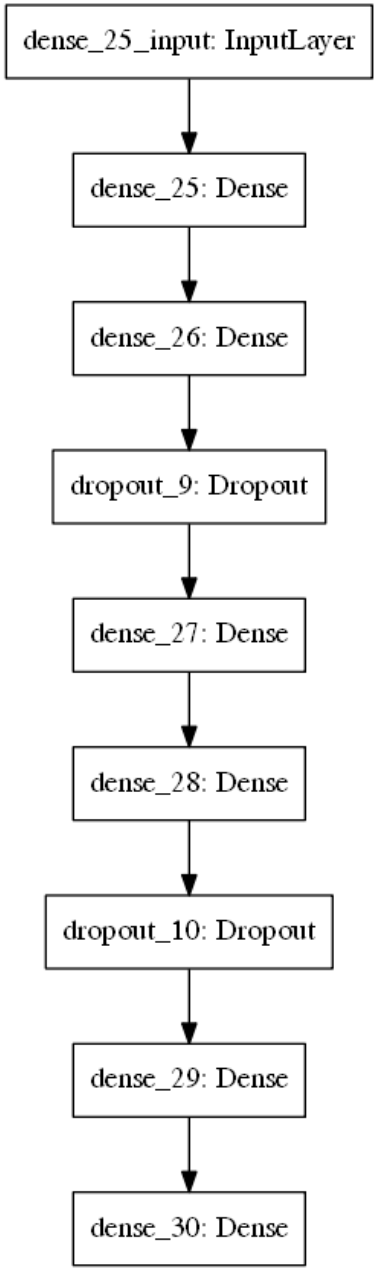
Model12

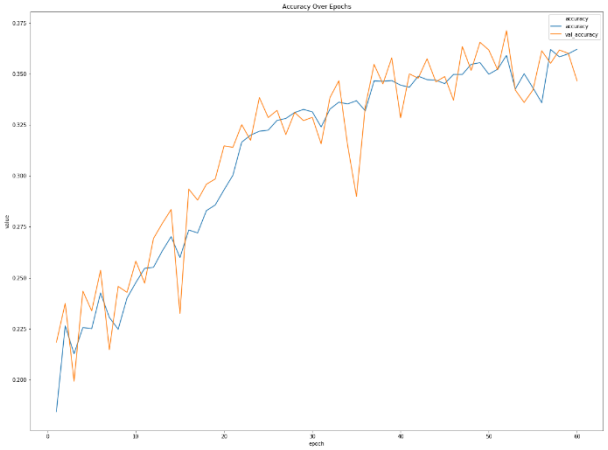

dense_13_input: InputLayer
dense_13: Dense
dense_14: Dense
dropout_5: Dropout
dense_15: Dense
dense_16: Dense
dropout_6: Dropout
dense_17: Dense
dense_18: Dense

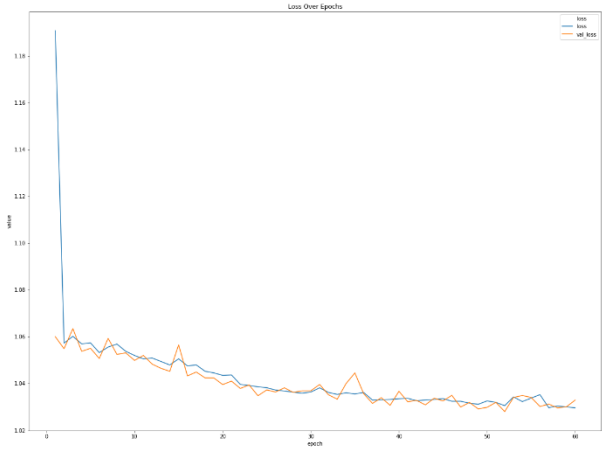

Accuracy Over Epochs
accuracy
accuracy
val_accuracy
value
epoch


Loss Over Epochs
loss
loss
val_loss
value
epoch

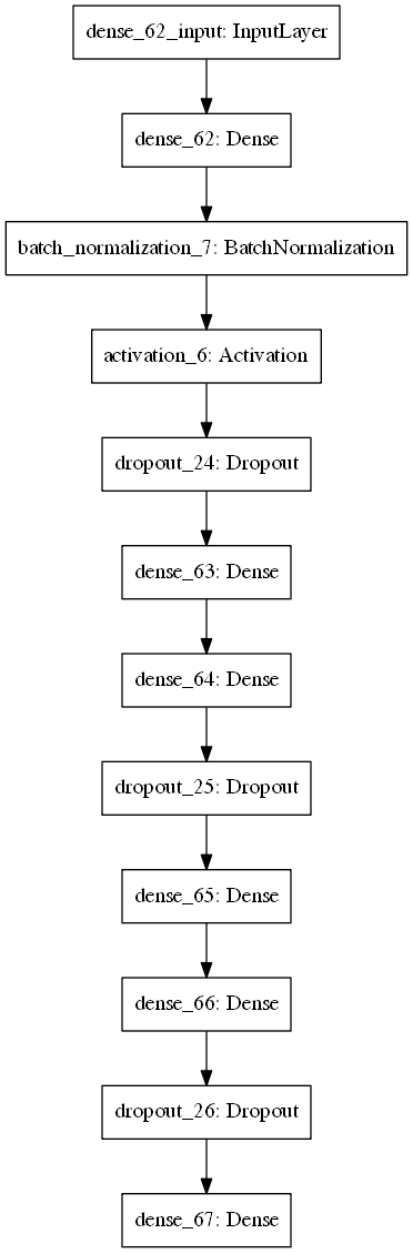
Model13

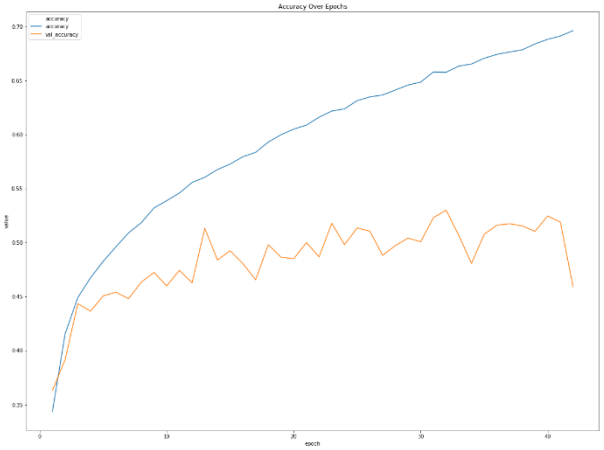

dense_25_input: InputLayer
dense_25: Dense
dense_26: Dense
dropout_9: Dropout
dense_27: Dense
dense_28: Dense
dropout_10: Dropout
dense_29: Dense
dense_30: Dense

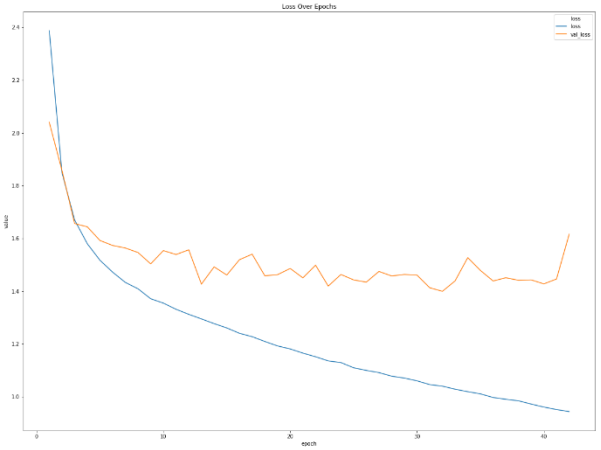

Accuracy Over Epochs
accuracy
val_accuracy
value
epoch


Loss Over Epochs
loss
val_loss
value
epoch

Model14


dense_62_input: InputLayer
dense_62: Dense
batch_normalization_7: BatchNormalization
activation_6: Activation
dropout_24: Dropout
dense_63: Dense
dense_64: Dense
dropout_25: Dropout
dense_65: Dense
dense_66: Dense
dropout_26: Dropout
dense_67: Dense


Accuracy Over Epochs
accuracy
val_accuracy
epoch


Loss Over Epochs
loss
val_loss
epoch