# IFT6135-H2019 - Assignment 1 Practical: Multilayer Perceptrons, Convolutional Neural Networks and Kaggle

Akilesh B. (20137625), Deepak Sharma (20143017),
Kanika Madan (20105657), Komal Teru(20148268)

February 17, 2019

Code available at: https://github.com/deepaks4077/IFT6135-HW1

## 1 Problem 1

### 1.a Initialization

In Figure 1 we observe that the loss and accuracy with weights initialized to zero do not change. In fact, the accuracy is essentially random for the training data (1/10 for 10 digits). This means that the network is not learning. This is most probably because the output activations are all zero and there aren't any gradients to propagate.

In contrast, both Normal and Glorot intializations seem to be doing fine. However, the normal initialization scheme is close to overfitting the data, as can be seen in the validation/test loss and accuracy curves that are plateauing after about 4 epochs. The Glorot initialization scheme is not close to overfitting the data and we later find that its performance continues to increase as we train the network over 60 epochs (question 1.b). This difference can be explained by two factors:

1. One the one hand, the MNIST dataset is a pretty simple dataset and both schemes perform well on the training, validation, and test data.
2. The Normal initialization scheme is causing the network to overfit the training data. We know from [1] that it causes the gradients to either explode or vanish as they propagate

| Hyperparameters | Best model | Attempt1 | Attempt2 | Attempt 3 | Attempt 4 |
|---|---|---|---|---|---|
| $h_1$ | 512 | 256 | 512 | 64 | 32 |
| $h_2$ | 512 | 256 | 256 | 64 | 32 |
| Non-linearity | ReLU | ReLU | ReLU | ReLU | ReLU |
| Learning rate | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 |
| Minibatch size | 100 | 100 | 100 | 100 | 100 |
| Number of epochs | 60 | 60 | 60 | 60 | 60 |

Table 1: Different set of hyperparameters tried in our experiments

| Model | # of parameters | Train accuracy | Val accuracy | Test accuracy |
|---|---|---|---|---|
| Best Model | 669706 | 98.76 | 97.67 | 97.33 |
| Attempt 1 | 269322 | 98.56 | 97.26 | 97.36 |
| Attempt 2 | 535818 | 98.73 | 97.58 | 97.26 |
| Attempt 3 | 55050 | 97.76 | 96.69 | 96.56 |
| Attempt 4 | 26506 | 97.10 | 96.34 | 95.93 |

Table 2: Train/Val/Test accuracies corresponding to different hyperparameters

backwards or forwards through the network. This may be causing the gradient descent to 'fall' into a local optimum. In contrast, the Glorot initialization scheme is designed to keep the variance of the activations of each layer on the same scale, smoothening the optimization process, helping the network bypass any local optimium and approach the true global optimum.

## 1.b Hyperparameter search

Finding out the best set of hyperparameters such that average accuracy on the validation set is at least 97%.

Total number of model parameters : 669706. (This falls in range 0.5M to 1M as given in question.
Train accuracy : **98.76**%
Validation accuracy : **97.67**%
Test accuracy : **97.33**%

The details of hyperparameters are listed in table 1 and the loss/accuracy as a function of epochs is plotted in figures 2 and 3.

2. We tried out other different combination of hyperparameters as described in the various columns of table 1 and the corresponding number of model parameters, train/test/valid accuracies are given in table 2.

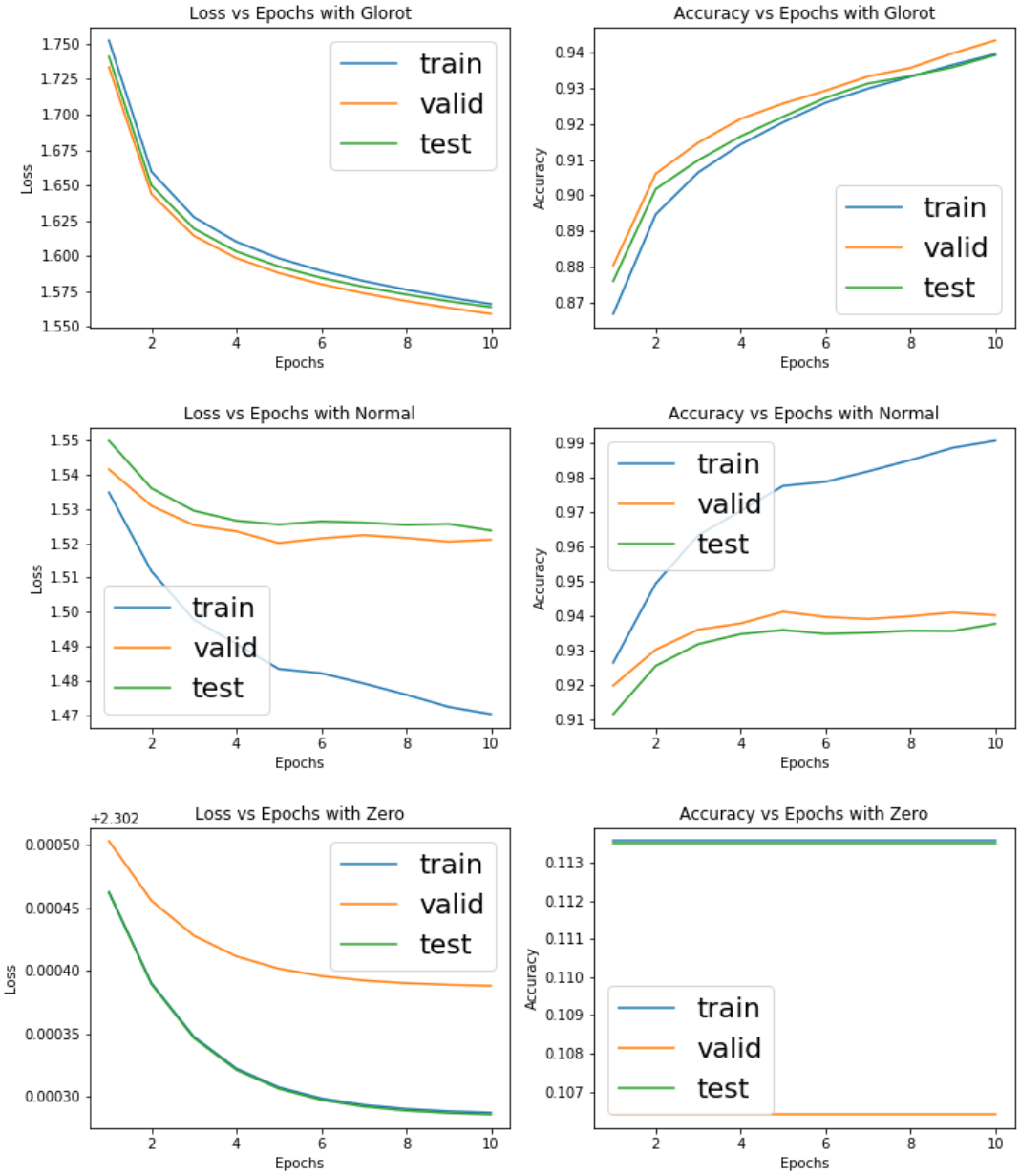



Figure 1: Loss and Accuracy over training time (epochs)

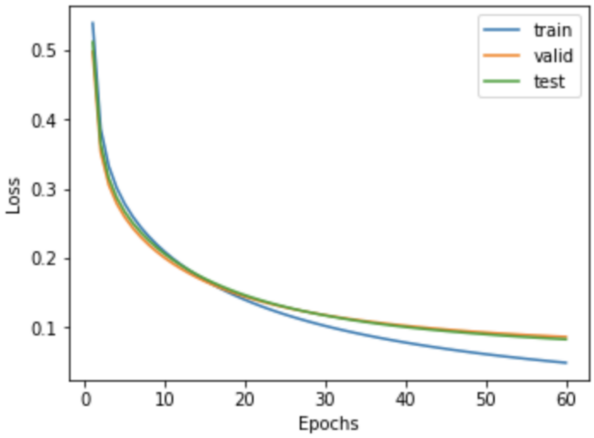



Figure 2: Loss vs Epochs

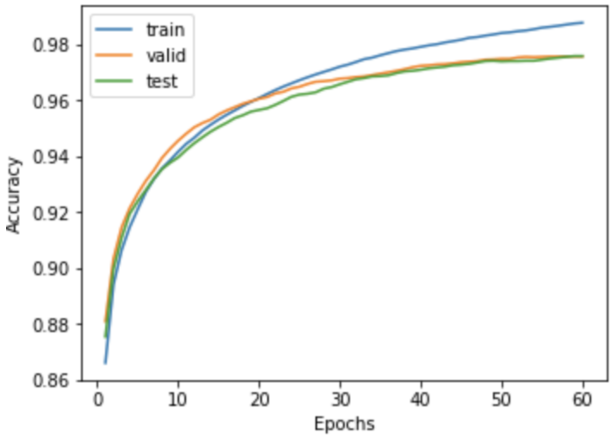



Figure 3: Accuracy vs Epochs

Figure 4: Best set of hyperparameters

As we can see from these experiments, even with less number of parameters we are able to achieve $> 95\%$ accuracy when the model is trained for 60 epochs and with other set of hyperparameters mentioned in the corresponding columns of table 1. This can be explained as the MNIST dataset is reasonably simple and it can be learnt well with fewer parameters.

### 1.c Validation of gradients

The finite gradient technique is trying to approximate the gradient of the loss function w.r.t the first 10 parameters of the second layer. This gradient will be inaccurate when the limit is taking with a large $\epsilon$ vs. when it is taken with a very small $\epsilon$ value. Thus, we see in figures 5 and 6 that the approximate gradient approaches the true gradient as N gets bigger (implying that $\epsilon$ is getting smaller).

## 2 Problem 2: CNN on MNIST

### 2.a Architecture

The CNN architecture used is as shown in the figure below. In the first convolutional layer, we blow up the channels from 1 to 18. In the two subsequent convulutional layers, we double the channels while halving the spatial dimensions using max pooling. These convolutional layers are followed by two fully connected layers, the first bringing down the dimension to 750 and the last giving the scores for all ten classes.

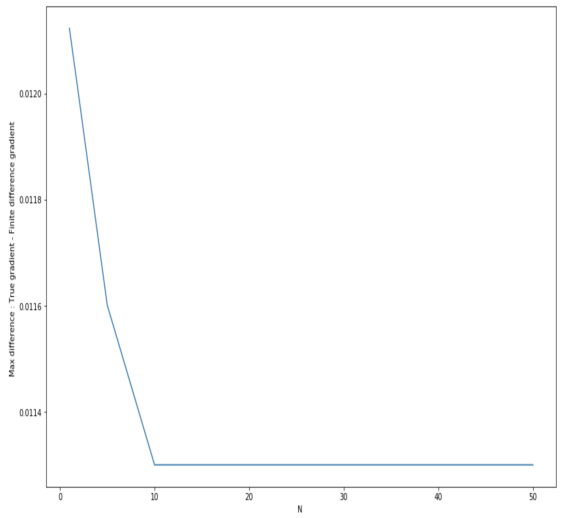



Figure 5: Finite Gradient vs N

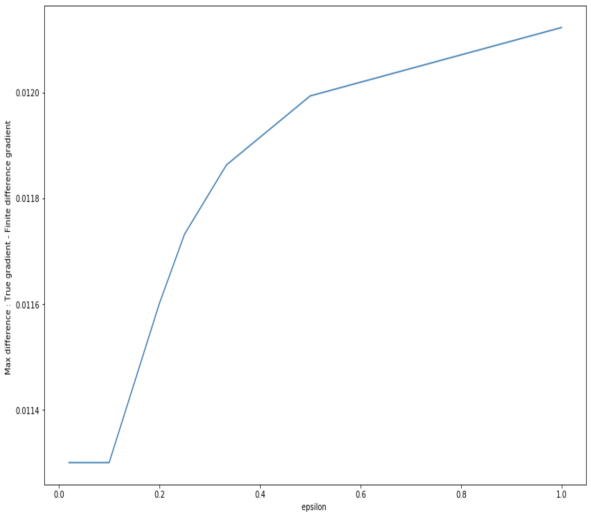



Figure 6: Finite Gradient vs $\epsilon$

Figure 7: Finite Gradients

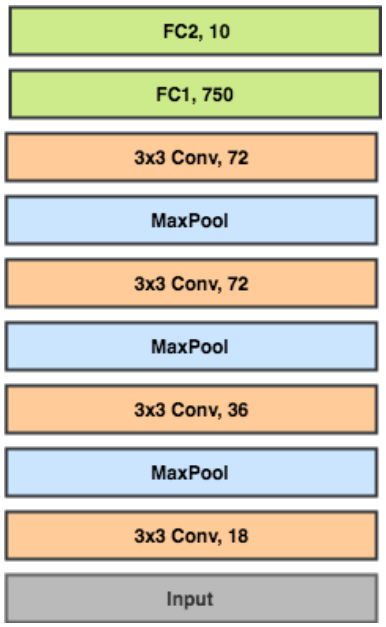



Figure 8: CNN model architecture

## 2.b Parameters

The total number of parameters are 570436. The break up is given in the Table 3. Notably, 86.6% of the parameters are in the final fully connected layers.

| Layer | Weight | Bias | Total |
|---|---|---|---|
| Conv1 | $(3 \times 3 \times 1) \times 18$ | 18 | 180 |
| Conv2 | $(3 \times 3 \times 18) \times 36$ | 36 | 5868 |
| Conv3 | $(3 \times 3 \times 36) \times 72$ | 72 | 23400 |
| Conv4 | $(3 \times 3 \times 72) \times 72$ | 72 | 46728 |
| FC1 | $72 \times 3 \times 3 \times 750$ | 750 | 486750 |
| FC2 | $750 \times 10$ | 10 | 7510 |
| **Total** | | | 570436 |

Table 3: Parameters distribution in CNN model

## 2.c Comparison with MLP

With the above architecture, the test accuracy obtained was 98.59%. This is better than the best performance, 97.33%, of MLP with 669706 parameters. Also note that the presented CNN performance is after only 10 epochs which is much better than the MLP which was at around 94% after 10 epochs.

The learning curves for the above model for training of 10 epochs was as follows.

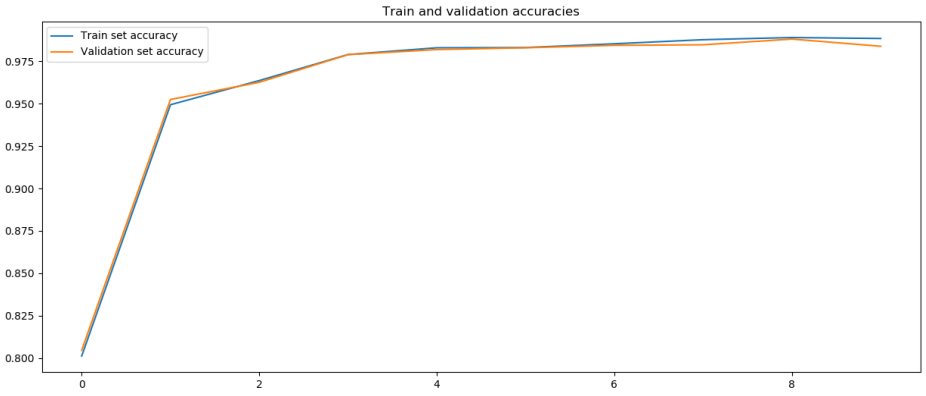



Figure 9: CNN model learning curve for 10 epochs

## 2.d Insights

To better understand the representational power of CNNs we trained a CNN model with just 2066 parameters. That model could achieve 97.1% test accuracy which is better than 95.93% achieved by MLP model with 26505 parameters (more that 10 times the CNN model).

Interestingly, we noticed that the effects of overfitting were not very apparent with in the increase in number of parameters of the model. To investigate this further we trained the

two CNN models, one with 2066 parameters and the other with 570436 parameters, for 100 epochs. Shown below are the learning curves for both.


Figure 10: CNN model with 570436 parameters

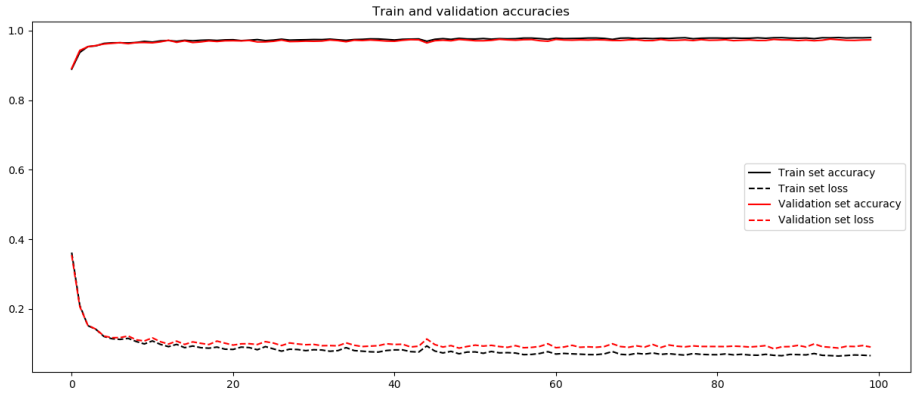



Figure 11: CNN model with 2066 parameters

As we can see, even though there is non-negligible difference in the loss values of training and validation set the accuracy values are surprisingly close. This applies to both over-parameterized and under-parameterized models. Our pre-matured deduction is that the MNIST dataset split distributions are very close that learning one is easily transferred to the other. In other words, MNIST dataset is too simple to do any rigorous study on deep neural network architectures.

# 3 Problem 3: Cats vs Dogs Kaggle challenge

This question was based on an in-class Kaggle challenge on Cats vs. Dogs image classification. A few things with respect to the data:

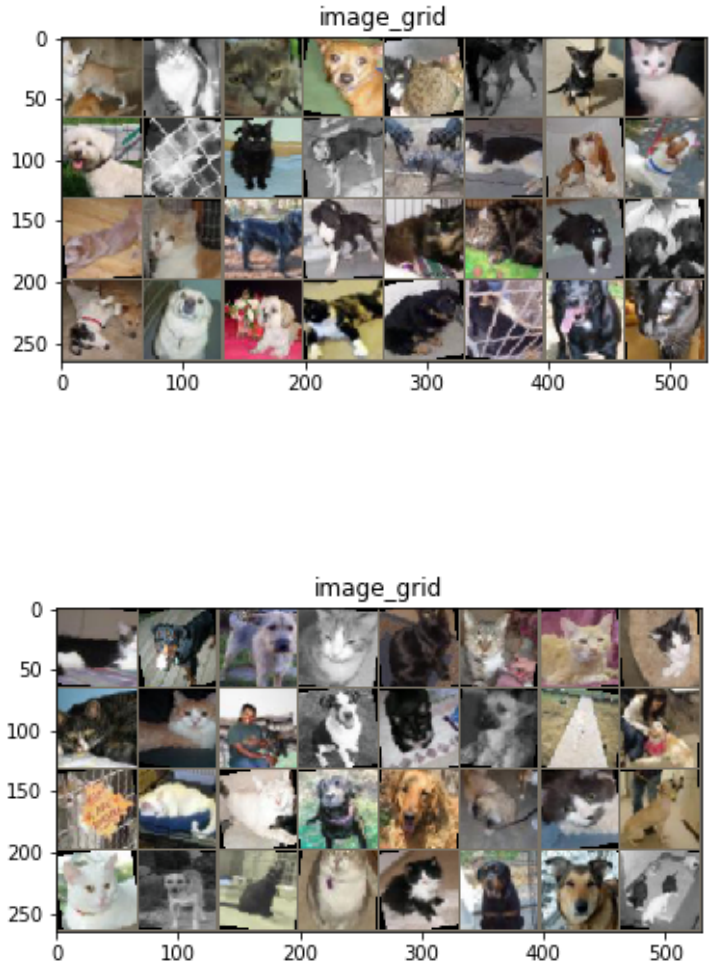



Figure 12: Visualizing images

- The data consisted of a training set of 19998 images and a test set of 4999 images of cats and dogs.
- The training data was a balanced dataset containing equal number of images for both classes of cats and dogs. Since this was not an imbalanced dataset, the accuracy metric can be trusted here.
- The images were of dimension 64 x 64 x 3.

### Visualization of given data

We plotted the images to visualize them (without zero-centering for easy plotting) in Figure 12. We can see from this figure (with cells indexed from $(1, 1)$) that the given images are pretty diverse with things like **occlusion** (Grid 1, Image $(2, 2)$), **deformation** (Grid 1, Image $(3, 1)$), **background clutter** (Grid 1, Image $(4, 3)$), multiple objects (Grid 1, Image $(3, 8)$), and **viewpoint variation** (Grid 2, Image $(4, 8)$). We also noticed images with **missing objects** and would thus be very hard to classify. For example, Grid 2, Image $(2, 7)$, has a road, and in Grid 2, Image $(2, 3)$, the main object is a man and the labeled class is very small.

### Model Architecture


Figure 13: Model architecture based on VGG

We used a VGG-inspired architecture [2], with alternating layers of Convolutional layers and Pooling layers built using PyTorch. The architecture of our model is shown in figure 13. We used (3 x 3) convolutional layers with padding of 1, with alternating layers of max-pooling in between with a kernel size of (2 x 2) and a stride of 2. Near the end of the network, we used three fully connected layers with size $4096, 4096$, and 2.

**Data augmentation and Hyper-parameters**

We used the following data augmentation and pre-processing:

- A batch size of 32.
- Data augmentation for training data:
  - Random rotation with degree 10
  - Random horizontal flip
  - Random crop with scale=(0.80, 1.0)
  - Color-jitter
  - Random gray-scaling with probability 0.2
  - Zero-center the images converted to (0,1) scale
- For test data, we zero-centered the images converted to (0,1) scale.
- Random split of trainset folder data into train set and validation set with a train-to-valid ratio of 0.8 (i.e., 80% training and 20% validation)

For the model, we used the following hyper-parameters:

- Convolutional layers: Kernel size $(3, 3)$, Stride= 1, Padding= 1.
- Max pooling: Kernel size= $(2, 2)$, Stride= 2
- ReLU activation function
- Xavier initialization
- Dropout=0.5 in fully connected layers
- Momentum= 0.9, Learning rate= 0.01, L2 Fegularization= $1.5e-3$
- Scheduler reducing learning rate on plateau with reduce factor= 0.05 and patience of 10 epochs

We implemented the dropout, momentum and L2 regularization ourselves to help with overfitting and easier convergence.

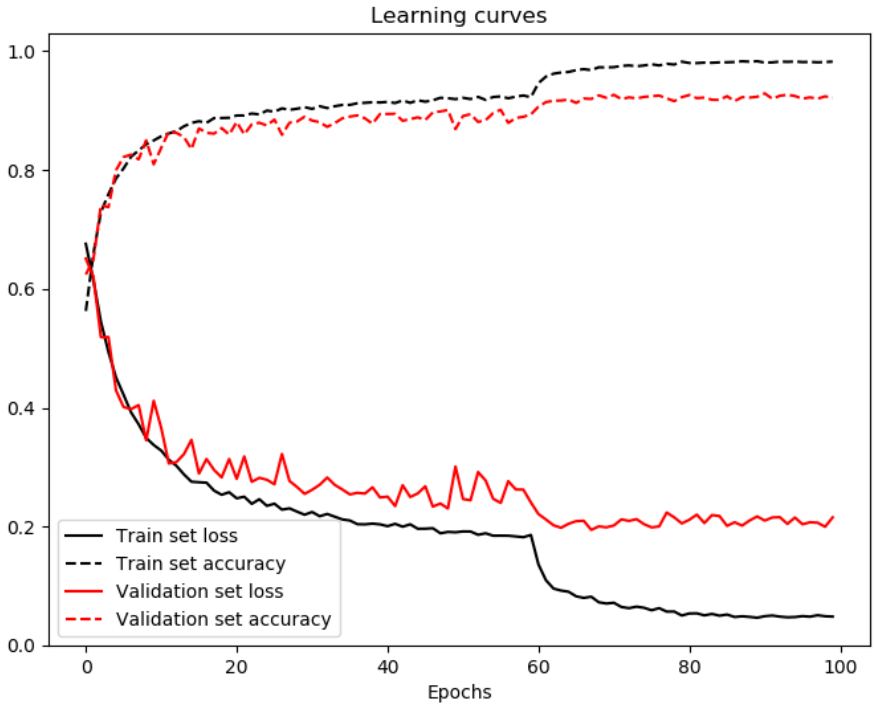



Figure 14: Training loss, Training accuracy, Validation loss, Validation accuracy for our model

### Hyper-parameter search

We adopted a random hyperparameter search and went from coarse to fine-tuning. We tweaked different knobs of the network and the most helpful were the regularization parameters.

The plots for our best model are plotted in the figure 14.

We found that changing the dropout and L2 regularization helped the most with reducing the generalization error, and finding the right values of momentum and learning rate helped with better convergence. Reducing learning rate on reaching a plateau helped in improving the accuracy: for example, we can see the loss dropping for both training and validation set at epoch 60 when the learning rate was reduced after reaching a plateau. We used **early stopping** when the model did not improve on the generalization loss any further with continued training. Our model scored **0.94397** on the public dataset (team name: skhwkk05). This was higher than the accuracy of 0.93 on our validation set.

Plots for some other experiments are contained in the Figure 15. We can see that using a dropout value of 0.5 and a higher L2 regularization penalizes the model too much, and both the training and validation error are relatively higher. As we reduce L2 regularization

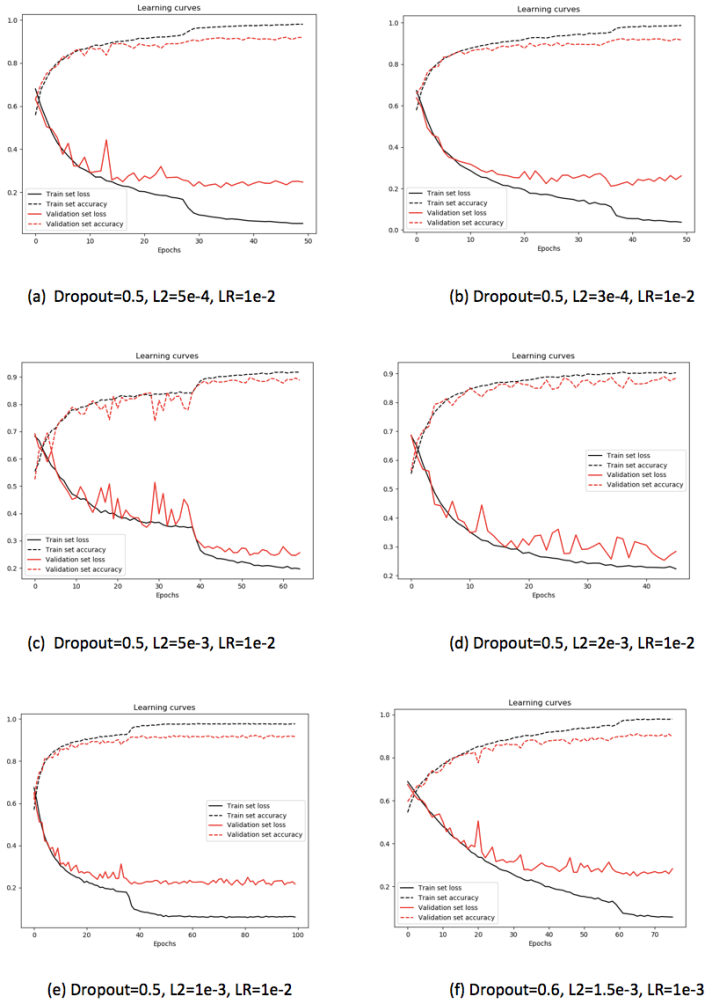



Figure 15: Training loss, Training accuracy, Validation loss, Validation accuracy for our other runs that were helpful in hyperparameter search

further to $2e-3$ and $1e-3$, both the training error and validation error reduce and we saw an increase in accuracy. We also tried the dropout value of 0.4 (and thus a keep-probability of 0.6), with a learning rate of $1e-3$, but the learning was very slow and the convergence of network was delayed. We then tried the same dropout with a bit higher learning rate of $1e-2$ and L2 regularization of $1.5e-3$, which gave us a good combination of easier convergence and lower generalization error. The plot for this is reported in Figure for our model 14.

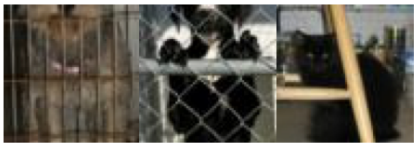
### Wrong predictions

We analyzed the images that our classifier had a hard time classifying and got them wrong. In addition to the challenges we found while visualizing the data in the beginning of the challenge (shown in figure 12, and we found some interesting patterns some of which are listed below in Figure 16.

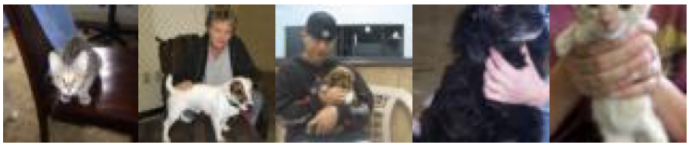
### Possible improvements and our learnings

While training our network on this dataset, we observed a few interesting things which we list here. We also mention some of things things which we think would have helped us generalize our network even better.

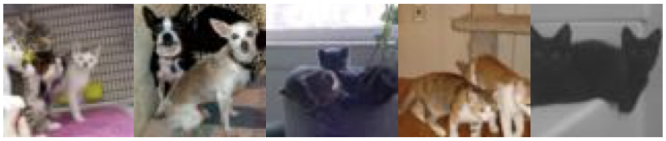
- Given limited data, data augmentation usually helps in the learning process and this is what helped our model as well. We feel that adding more data augmentation would have helped our network even more.
- Regularization techniques such as L2 and Dropout played a major role in our model. We believe this is mainly because a deep learning model can easily overfit such a small dataset and regularization techniques helped in keeping our weights in-check. Our intuition is that adding more advanced regularization techniques such as BatchNorm, LayerNorm would have helped here even further.
- We tried ResNet architecture [3], and found that they easily overfitted. This goes with our intuition that a good model does not necessarily need more depth, especially with a smaller dataset like this one. We found that our VGG-inspired model was doing well on the training data (i.e. it was learning), but needed some more regularization and data augmentation to help generalize better.
- The technique of random hyperparameter search helped us try out a wider range of hyperparameters. In next steps, however, we would have tried to fine-tune our model even more as we found that in this dataset, regularization and fine-tuning helped more than just the architecture.
- We also tried a deeper version of VGGNet, but we could easily see that it did not learn well. The training was very slow and the error rates did not get any better. Again, this goes with our intuition that an ever deeper model (without advanced regularization techniques) might not have helped much. The network in-fact needed more babysitting to improve on the test set.



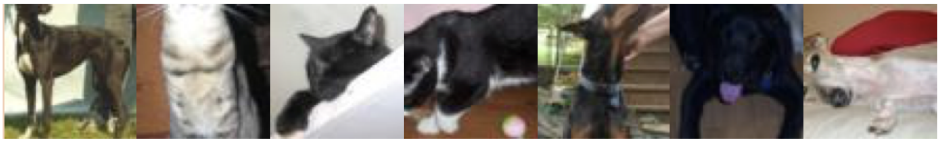
(a)



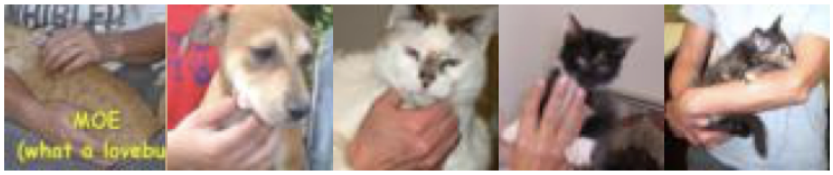
(b)



(c)



(d)




(e)

(f)

(g)

Figure 16: (a) Main class hidden behind some object; (b) Some other main class; (c) Multiple instances of the class; (d) Missing face or body; (e) Hand in the image; (f) Background clutter; and, (g) Distortion

# References

[1] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. AISTATS, 2010.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for Large scale image recognition. ICLR 2015.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. IEEE 2016.