COMP 551 Machine Learning, MiniProject 3

Sebastian Arenas, Lia Formenti, Trevor Shillington

April 19, 2020

Abstract

Neural networks have become an incredibly powerful tool in machine learning for many applications, such as image recognition. In particular, multilayer perceptrons (MLPs) (feedforward neural networks) and the more recent and more powerful convolutional neural networks can be well suited for image recognition. In this report we will use a multilayer perceptron and various implementations of a convolutional neural network to classify images from the CIFAR-10 dataset. We code from scratch an MLP and show that it scores modestly in classifying the images, with a test set accuracy of 39.8%. We then utilize PyTorch to implement three CNNs, each progressively more complex than the last. We show that the more complex CNNs perform better at the cost of computational resources. Our final and best performing CNN, CNN-III, attains a test set accuracy of 70.6%. Our goal in this report is to demonstrate the power of neural networks in the context of multi-class image recognition.

1 Introduction

Neural networks can be difficult to train but few models can surpass their power and flexibility. The layered structure results in many parameters and hence a non-convex optimization problem with many local optima; however, since they can be universal function approximators if allowed to be wide or deep enough, they can model the most complicated distributions [1]. To gain an understanding of the process of training and tuning a neural net, a multilayer perceptron (MLP) class (SimpleMLP) was written to classify images from the CIFAR-10 image dataset [2].

In reality, convolutional Neural Networks (CNN) are used instead of MLPs for image classification tasks since the initial work presented in [3]. CNNs use several learnable filters (kernels) that convolve the input signals at specified strides to model complicated distributions. As such, CNN models are better able to use the spatial information of an image than an MLP whose inputs unravel the grid structure of the image [4]. Interestingly, CNNs also extract higher level representation of image features without performing feature engineering like in previous studies, which involved a manual and expensive process that used domain knowledge to create features for training in machine learning algorithms [5][6]. Generally, CNNs require less parameters than MLPs, which reduces the computational complexity [5]. However, the convolution operation is expensive so deep networks do require longer computation time [7]. Krizhevsky [2] obtained record performance on on the CIFAR dataset with a CNN. In this work, three different CNN network architectures are compared. Overall, wisdom about training networks gained by these exercises is recorded.

2 Dataset

CIFAR-10 is a collection of 60,000 32 × 32 pixel 3-channel colour images, each belonging to one of ten possible classes describing the image (e.g. dog, horse, truck, car, etc.). The goal of our algorithms will be to predict which object is in the image. The dataset is pre-split into 50,000 training images and 10,000 test images [2]. For the MLP, the training set was further partitioned to reserve 10,000 images for a validation set, using code from [8]. The pixel intensities in each of the channels were normalized from lying between 0 and 1 to between -1 and 1, necessary for Xavier initialization [9], discussed later. Accuracy was deemed an appropriate evaluation metric for the following models since the cost of misclassification was the same for all ten classes.

3 Models

3.1 Multilayer Perceptron (MLP)

A SimpleMLP class and its methods were written by the authors for this work. The MLP written here is a feedforward network with a single hidden layer with an adjustable number of hidden units. The inputs are the 32 × 32 × 3 = 3072 pixel intensities that define the image. Parameters used in the model are defined in Figure 1, with the dimensions defined in Table 1. The output layer has ten units, acted upon by a softmax function to get an estimate of the probability of each class assuming a categorical distribution [10], and the class predicted corresponds to the unit with the maximum probability. Note that the true labels used in training had to be one-hot encoded to match the MLP architecture.

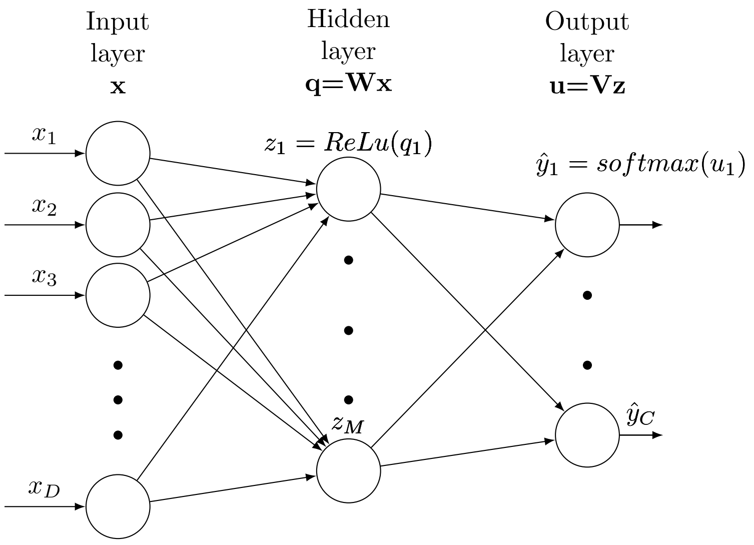


Figure 1: Single layer MLP architecture, defining the variables used in this report.

The ReLu activation on the hidden layer was chosen because of its well behaved gradient and proven robustness. More traditional choices of activation function such as a sigmoid have gradients that saturate at large activations, which impedes a gradient descent based optimization algorithm from working efficiently [1][11].

3.1.1 Optimization

A stochastic gradient descent (SGD) algorithm was written to perform the weight adjustments. Each epoch,

$$W \to W - lr \times dW \quad (1)$$

$$V \to V - lr \times dV \quad (2)$$

Input, x	D features + bias	3073
Input weights, W	M × (D+1)	M × 3073
Hidden units, z	M units + bias	M+1
Hidden weights, V	C × (M+1)	10 × (M+1)
Output, $\hat{y}$	C classes	10

Table 1: Table defining the dimensions of parameters of the MLP. Individual parameters are referenced by the dimension letter in lowercase as a subscript.

where lr is the learning rate and dW dV are the gradients of the chosen cost function calculated by the back propagation algorithm. Since the task is categorical classification, cross entropy loss was used, and the cost was the average loss over a batch of 100 sampled training instances. The weights were adjusted until the accuracy on the validation set did not change by more than 0.01 for three iterations, or until the maximum number of iterations was reached. The early stopping helped to prevent overfitting and wasteful use of computation time. An example of the evolution of the cost function is shown in Figure 2.

The inputs, weights and gradients were all implemented as Pytorch tensors to take advantage of the GPUs available through Google Colab for computational efficiency. In addition, the batches for SGD were selected using a Pytorch Dataloader, and the predict function took as input a Dataloader. The Dataloaders create an iterable over batches of input data automatically, which is memory and speed efficient. Even when calculating the training accuracy on all 40,000 examples in the training set, runs never took more than 6 minutes.

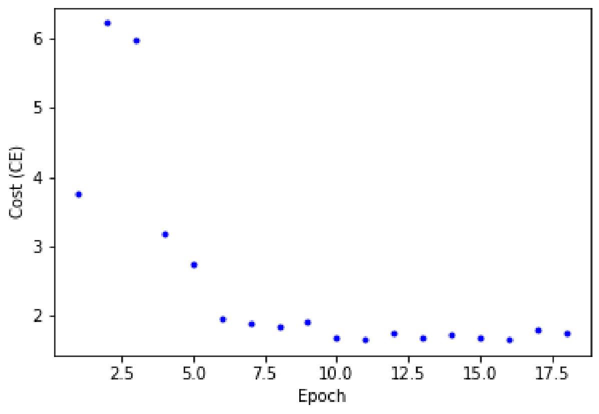


Figure 2: Example of evolution of cost over epochs for the MLP, with tuned hyperparameters (detailed in section 3.1.3)

3.1.2 Back-Propagation

Back-propagation is the algorithm for computing the gradient of the loss with respect to both layers' sets of parameters for a single instance (which can be averaged over a batch to get an approximate gradient of the cost). The gradient of the loss, L, with respect to the parameter matrix V is given by:

$$\frac{\partial L}{\partial V_{C\times M}} = \frac{\partial L}{\partial \hat{y}_C}\frac{\partial \hat{y}_C}{\partial u_C}\frac{\partial u_C}{\partial V_{C\times M}} \tag{3}$$

The individual partials can be calculated by first evaluating the values of each unit in the network at the current epoch in a "forward pass", then substituting these values into the analytic forms for the partials. The analytic form of the partials depends on the activations chosen at each layer and the loss. With the softmax activation on the output layer, the partial derivative of the loss function L with respect to $V_{C\times M}$ is:

$$\frac{\partial L}{\partial V_{CxM}} = (\hat{y}_C - y_C)z_M \tag{4}$$

For numerical stability, the maximum value of the softmax input u was subtracted from all u_c, which does not change the output but prevents overflows.

The partial of the loss with respect to W can be derived similarly.

$$\frac{\partial L}{\partial W_{M\times D}} = \sum_c (\hat{y}_c - y_c) W_{M\times D} \frac{\partial ReLu(q_M)}{\partial q_M} x_d, \tag{5}$$

where the derivative of the ReLu activation is zero for inputs less than or equal to zero and one for inputs greater than zero. Note that the partials of the loss with respect to the parameters V are part of this equation. This is what makes back propagation efficient: the required values for each step of the calculation are pre-stored. The partials of the loss of a batch of training examples were averaged to get an approximation of the gradient of the cost function.

3.1.3 Performance and Hyperparameter Tuning

The model originally was predicting only one or two outputs for every image, which was solved by implementing Xavier initialization [9] of the initial weights. They were initialized from a normal distribution with mean zero and standard deviation of $1/\sqrt{D+1}$. This roughly conserved the variance of the inputs as they propagated through the layers for better performance, and helped prevent numerical issues. Hyperparameter tuning brought the accuracies above random (10 %).

During the initial stage of tuning, a fast version of the fit method was run (small batch sizes, few epochs, etc.) to gauge the order of the parameters. The hyperparameters were: the learning rate, the batch size, and the number of hidden units. The learning rate was tuned first because the model will not converge at all with a poor learning rate. $lr = 0.5$ was chosen to balance the speed at which the parameters could be learnt while preventing large fluctuations in the cost. For good convergence, it was necessary to schedule the learning rates to decrease, according to the schedule:

$$lr_{epoch} = lr\ (epoch)^{-0.51} \tag{6}$$

where lr is the initial learning rate and *epoch* is the iteration number. This schedule satisfies the Robbins-Monroe conditions [10].

Larger batch sizes yielded better accuracies, potentially because they were using more information by studying more examples. However, the larger the batch size the longer the SGD algorithm would take. The results are summarized in Table 2. Note that the number of epochs varied because of the stochastic nature of the optimizer. The wall time, while useful for showing trends, also varied and included the time it took to calcuate per-epoch

Batch size	Train Acc	Val Acc	Test Acc	No. Epochs	Wall Time(s)
10	0.089	0.103	0.111	6	104
100	0.271	0.266	0.270	9	163
1000	0.380	0.378	0.378	12	221
1500	**0.395**	**0.385**	**0.392**	**15**	**299**
2000	0.394	0.383	0.386	14	310
2500	0.391	0.379	0.390	12	256

Table 2: The effect of batch size on the train, validation and test accuracy and run time, with lr=0.5 (with scheduled decreases) and M=2000. The value in bold gives the highest test accuracy.

accuracies.

With the batch size at 1500, the number of hidden units was tuned, with the results shown in Table 3. The validation accuracy increased with more hidden units (a more expressive model). The danger with expressivity is that the model will overfit; however, implementing early stopping helped to prevent this eventuality. Since the train, validation and test accuracies are close, there is no evidence of overfitting. The cost of more units was increased wall time. Moreover, it is common wisdom in machine learning that increasing the depth is more beneficial than increasing the width, so we were wary of creating more hidden units than input features [1]. We opted therefore for a fully connected model.

With the tuned hyperparameters, the evolution of the train, validation and test accuracy is shown in Figure 3. Note that the train accuracy is over the entire 40,000 training instances. Since the optimization is stochastic, not every epoch sees every example, thus the training accuracy reflects the test and validation accuracy.

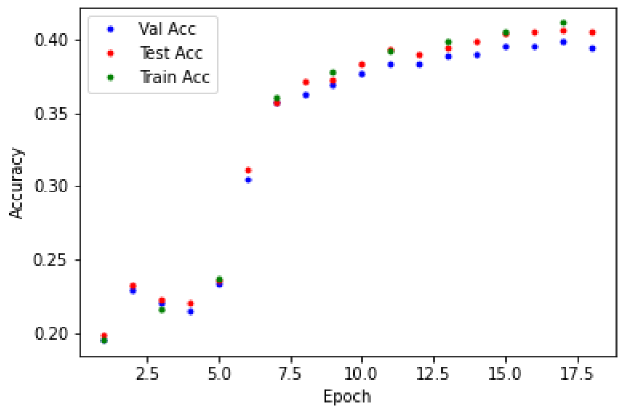


Figure 3: Train, validation and test accuracy of MLP per epoch with the tuned hyperparameters: $lr = 0.5$ (with scheduled decreases), $B = 1500$, $M = 3072$

3.2 Convolutional Neural Networks (CNN)

The convolutional neural network (CNN) approach is used extensively to create models that are invariant to certain transformations of the inputs. CNNs can do this by building the invariance properties into the structure of a neural network as was revisited in the previous section [3][12]. This is the basic idea of CNNs and it has been commonly applied to image recognition in data, as is done in this report.

As the name convolutional neural network indicates, the network uses a convolution operation which is a specialized kind of linear operation in place of general matrix multiplication in at least one of the layers [1]. The convolution operation for continuous inputs can be denoted as follows:

$$s(t) = \int x(a)w(t-a)da = (x * w)(t) \tag{7}$$

and can be typically expressed with an asterisk. The first argument is known as the input (function x) and the second argument refers to the kernel, or filter (function w). The output is sometimes referred to as the feature map and is expected to be a weighted average, therefore w is required to be a valid probability density function (PDF) [1]. Generally, convolution is defined for any functions for which the integral in equation 7 is defined. For this project, the convolutional operation is performed over discrete inputs, where x is the input image and w is the convolutional filter.

3.2.1 Implementation of CNN-I and CNN-II

The first model implemented follows the configuration given by the suggested tutorial [13]. This implementation employs 2 convolutional layers with max pooling on each, followed by 3 linear layers. For the convolutional layers, 2D convolution is applied over an input signal composed of several input planes. The first layer includes 3 in-channels since the images are RGB, 6 output-channels (filters) and 5 filter size (kernel size). The process can be seen as 3 input channels that will be convolved by 6 different filters which will create 6 different output channels. The values chosen as output-channels are equivalent to the nodes in the following layer. The second convolutional layer employs 6 input-channels (the output of the previous layer) that will be convolved by 16 different filters which will create 16 different output-channels. For the stride, 1 by 1 steps (units) are being used to move the filter right and down across the input signal. No padding is added. Additionally, max-pooling is applied after each convolutional

Number of Hidden Units (M)	Train Acc	Validation Acc	Test Acc	No. Epochs	Wall Time (s)
10	0.189	0.193	0.189	6	115
100	0.295	0.294	0.300	15	302
1000	0.371	0.359	0.375	15	306
2000	0.395	0.385	0.392	15	299
3072	**0.407**	**0.392**	**0.398**	**15**	**413**
4000	0.419	0.399	0.410	18	368

Table 3: The effect of hidden layer width on train, validation and test accuracy and run time, with lr=0.5 and B=1500. The values in bold show the chosen number of hidden units.

layer. Max-pooling takes a 2x2 kernel with a stride of 2 and applies it to the given output, taking the maximum value from each zone. After the first two convolutional layers, 3 linear layers are applied with ReLu as the activation function in the first 2 linear layers. As was implemented in the MLP model, cross-entropy loss was used to compared the outputs of our neural network with the true labels.

For **CNN-I**, the optimization algorithm was stochastic gradient descent (SGD), which adjusted the weights at each iteration. A learning rate of 0.01 and a batch size of 64 were used for this model. A stopping criteria was implemented for the three CNN models to conserve computational resources: if, once the loss dropped below 1, the loss of epoch N was greater than the loss of epoch N-1, then we added to a count. Once that count hit 5 it terminated. This indicates an oscillation in the loss which shows that the training is complete. Figure 4 presents the evolution of the loss of the model over 30 epochs for both the training and testing sets. There is a notorious increase of the loss due to the high learning rate used that missed the minimum. Learning rate scheduling and gradient clipping are implemented in the following models to attack this. Moreover, figure 4 presents a big gap between training and testing losses which indicates overfitting. Regularization methods are considered in the following models in order to reduce overfitting. Finally, in Appendix A, a table of accuracies split into classes is presented. It is interesting, and not unexpected, to note that classes which could be easily mistaken for each other (e.g. cat, dog, horse) have a low score, while objects which have a unique shape score higher as they are more easily and confidently recognized by the CNN.

For **CNN-II**, the same architecture presented in **CNN-I** was used. However, for this second model, the Adam optimization algorithm, an extension to stochastic gradient descent, was used. Adam is an optimization algorithm that updates network weights iteratively like SGD, but a learning rate is maintained for *each* network parameter and is separately adapted as learning unfolds. Adam method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. According to [14], Adam combines the advantages of two other extensions of stochastic gradient descent, the Adaptive Gradient Algorithm (AdaGrad) and the Root Mean Square Propagation (RMSProp).

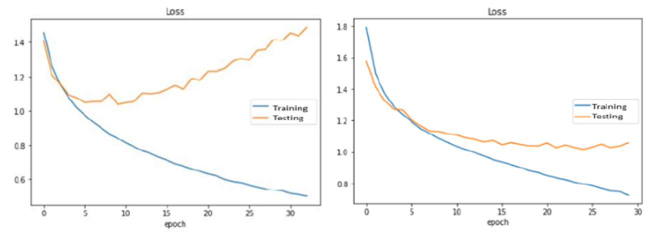


Figure 4: Cost vs epochs. Left: **(CNN-I)**;Right: **(CNN-II)**

A batch size study was performed for different learning rates and it is presented in table 4 and the corresponding plots can be found in appendices B, C and D.

Table 4 and appendices B, C and D show that the larger the batch size, the faster the learning. However, this learning shows a volatile behaviour with higher variance in the classification accuracy. On the other hand, the smaller the batch size, the slower the learning process. Furthermore, with smaller batch sizes the final stages of learning present a more stable convergence with lower variance in classification accuracy [15]. According to [15] and what is shown table 4 and appendices B, C and D, increasing the batch size progressively decreases the amount of learning rates that provide stable convergence and high performances on the test set. In contrast, small batch sizes provide more timely gradient calculations, which produce more stable behaviours regarding to the training [15][5]. Table 4 and appendix B, C and D show an unstable behaviour with the chosen learning rate of 0.1 with poor performance and violent changes, confirming that the learning rate used to update the weights is too large. Decreasing the learning rate showed more stable behaviour in the training of the network [15] [3]. From table 4, a batch size of 256 and a learning rate of 0.001 were chosen for building **CNN-II**. The learning rate can also be adjusted during the learning process, via the decay weight parameter of the Adam optimizer. Additionally, it can be updated per epoch or per N epochs. Updating the learning rate can allow for faster convergence as in [16]. The presented results suggest that small batch sizes allow to improve the convergence and accuracy as well as the effect of reducing the computational parallelism available. This can be seen as a motivation for future work to consider the trade-off between computational cost and test performance.

Learning Rate	Batch Size	No. Epochs	Train Acc	Val Acc
0.01	8	30	0.122	0.135
0.01	32	30	0.388	0.397
0.01	64	30	0.525	0.589
0.01	128	30	0.513	0.546
0.01	256	30	0.687	0.592
0.01	500	30	0.681	0.586
0.01	1000	30	0.759	0.599
0.001	8	30	0.798	0.622
0.001	32	30	0.826	0.625
0.001	64	30	0.836	0.616
0.001	128	30	0.841	0.630
0.001	**256**	**30**	**0.749**	**0.667**
0.001	500	30	0.699	0.643
0.001	1000	30	0.647	0.613
0.0001	8	30	0.728	0.643
0.0001	32	30	0.648	0.614
0.0001	64	30	0.626	0.589
0.0001	128	30	0.655	0.614
0.0001	256	30	0.557	0.528
0.0001	500	30	50.8	49.9
0.0001	1000	30	0.489	0.489

Table 4: CNN Experiment. The value in bold gives the highest test accuracy.

Figure 4 shows an improvement in **CNN-II** regarding the overfitting presented in **CNN-I** as a result of the new optimization method, batch size and learning rate chosen.

3.2.2 Implementation of CNN-III

For **CNN-III**, a third convolutional layer was added. This network used the same parameters as **CN-II**, with the only difference being the implementation of layers. In this architecture we applied max pooling after the second and third convolutional layer. The architecture is CONV1 → CONV2 → MAX POOL → CONV3 → MAX POOL → FC1 → FC2 → FC3. The output channels were also modified from **CNN-II** to incorporate the additional layer. Additional layers, in general, increase the accuracy of the CNN. As can be seen in [5] and [16], which also implemented CNNs on the CIFAR-10 dataset, increasing to as many as 14 layers can lead to accuracies on the test set of more than 90%. Due to restrictions in computational power we were unable to produce a CNN of more than three layers, but nevertheless **CNN-III** is able to show improvement on **CNN-II**. The choices for output channels were pseudo-random, taking some inspiration from [5] and [16], but keeping in mind our limited resources. The accuracy and loss vs epoch plots can be found in Figure 5. These not only show a slight increase in accuracy over **CNN-II**, but also less overfitting. These plots were created without using a stopping criteria in order to maintain consistency with other plots, although it is implemented in the code. The final accuracy is 70.6%.

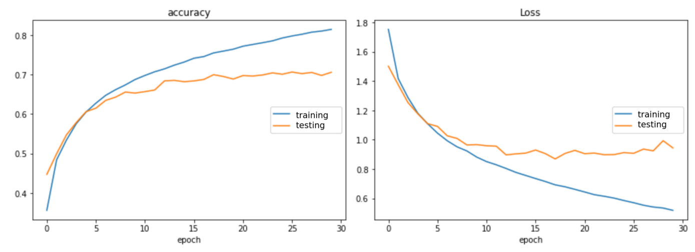


Figure 5: Left: accuracy vs epoch for **CNN-III**. Right: loss vs epoch for **CNN-III**.

4 Results and Conclusions

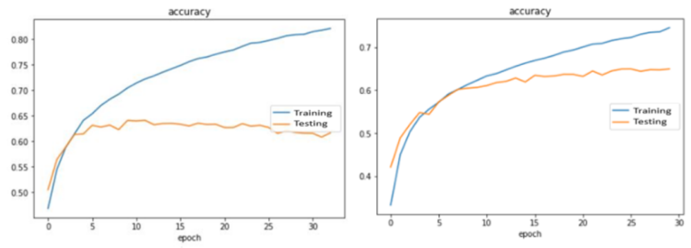


Figure 6: Accuracy vs epochs. Left:**CNN-I**. Right:**CNN-II**.

Figure 6 shows a deviation between the training and test accuracies for **CNN-I**. It shows the onset of overfitting after the fifth epoch and by following the trend one can deduce that the overfitting will not improve with more epochs. The overfitting was improved with regularization and more depth [5]. Figure 6 presents how the overfitting by **CNN-I** was improved in **CNN-II** by applying Adam optimization with the corresponding regularization and by tuning the batch size and learning rate. **CNN-III** presents a modified CNN architecture, adding in an additional convolutional layer in order to show that more layers improve accuracy. **CNN-III** gave the best accuracy of 70.6%, although research on more advanced CNNs on the CIFAR-10 dataset show that accuracies greater than 90% can be achieved with many more layers [5][16]. However, each additional layer comes with the cost of increasing computational power required [7]. Also considering the 30% increase in accuracy over the MLP, the results demonstrates the power and usefulness of using convolutional neural networks in image recognition.

MODEL	Train Acc	Test Acc
MLP-I	0.407	0.398
CNN-I	0.838	0.612
CNN-II	0.749	0.667
CNN-III	0.814	0.706

Table 5: Performance of the models

4.1 Contributions

Lia implemented the MLP from scratch. Seb implemented and tested CNN-I and CNN-II, while Trevor implemented and tested CNN-III. All three contributed to the writing of the paper, each member focusing on the sections that they coded.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images, April 2009.

[3] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[4] Ce Zhang, Xin Pan, Huapeng Li, Andy Gardiner, Isabel Sargent, Jonathon Hare, and Peter M Atkinson. A hybrid MLP-CNN classifier for very fine resolution remotely sensed image classification. *ISPRS Journal of Photogrammetry and Remote Sensing*, 140:133–144, 2018.

[5] Akwasi Darkwah Akwaboah. Convolutional Neural Network for CIFAR-10 Dataset Image Classification. (November), 2019.

[6] Edgar Medina, Mariane R Petraglia, Gabriel R C Gomes, and Antonio Petraglia. Comparison of CNN and MLP Classifiers for Algae Detection in Underwater Pipelines.

[7] Abdelaziz Botalb. Contrasting Convolutional Neural Network (CNN) with Multi-Layer Perceptron (MLP) for Big Data Analysis. *2018 International Conference on Intelligent and Advanced System (ICIAS)*, pages 1–5, 2018.

[8] rishab96. All datasets don't have an option to get validation set split.168, 2017.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. page 8.

[10] Christopher M Bishop. *Pattern Recognition and Machine Learning*, volume 4 of *Information science and statistics*. Springer, 2006.

[11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[12] Yann LeCun, Léon Bottou, Patrick Haffner, and Paul G. Howard. Djvu: a compression method for distributing scanned documents in color over the internet. In *6th Color and Imaging Conference, CIC 1998, Scottsdale, Arizona, USA, November 17-20, 1998*, pages 220–223. IS&T - The Society for Imaging Science and Technology, 1998.

[13] Training a Classifier — PyTorch Tutorials 1.4.0 documentation.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[15] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018. cite arxiv:1804.07612.

[16] John Olafenwa. Basics of image classification with pytorch, 2018.

[17] Aurelien Geron. *Hands-On Machine Learing With Scikit-Learn & Tensor Flow*. O'Reilly Media, 2017.

[18] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009.

[19] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, Cambridge, Mass. [u.a.], 2013.

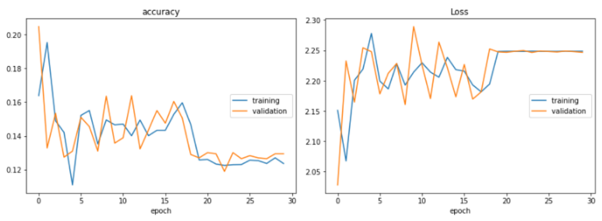
[20] S Ben Driss, M Soua, R Kachouri, M Akil, Esiee Paris, and Bd Blaise Pascal. A comparison study between MLP and Convolutional Neural Network models for character recognition. 10223:1–11, 2017.

Appendix A

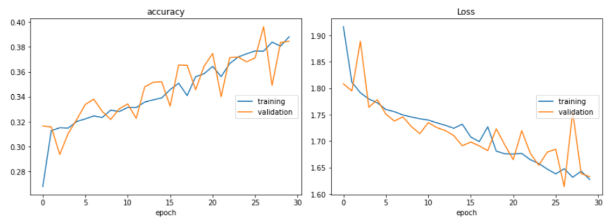
Class	**Testing Accuracy**
Plane	0.78
Car	0.74
Bird	0.51
Cat	0.38
Deer	0.47
Dog	0.47
Frog	0.66
Horse	0.54
Ship	0.79
Truck	0.60

Table 6: Testing accuracy by class for **CNN-I**.

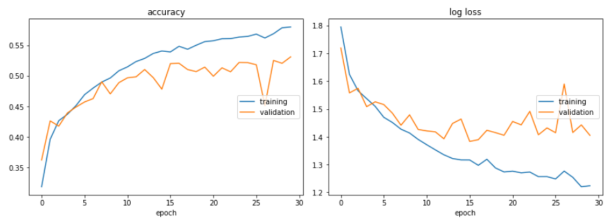
Appendix B



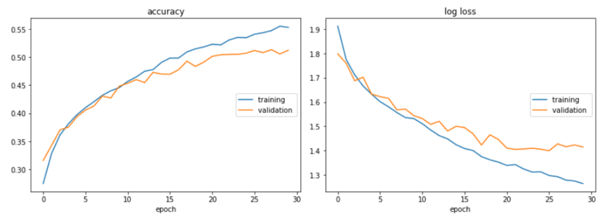
Batch Size = 8



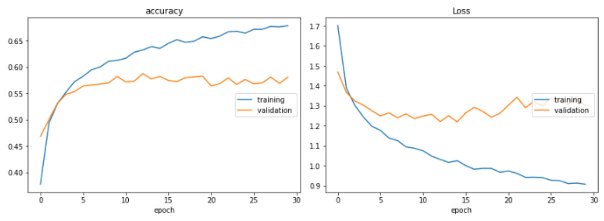
Batch Size = 32



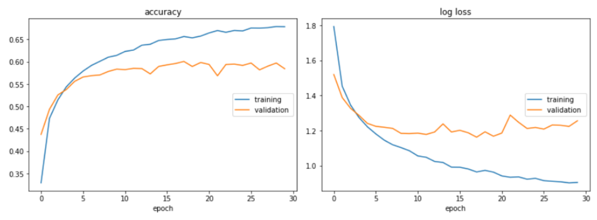
Batch Size = 64



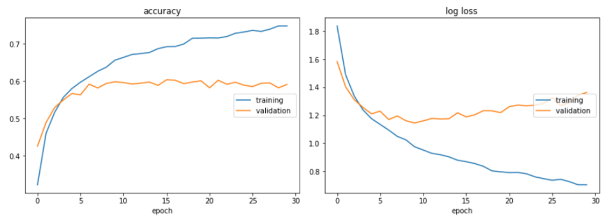
Batch Size = 128



Batch Size = 256



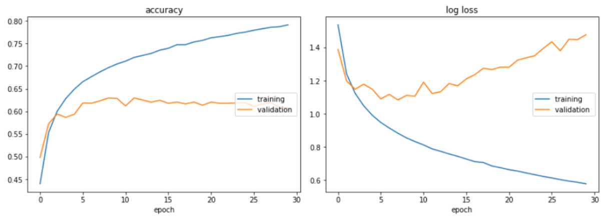
Batch Size = 500



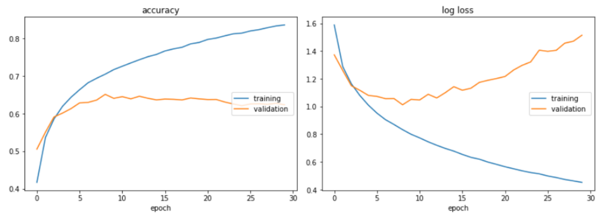
Batch Size = 1000

Figure 7: Accuracy and Loss for Learning Rate = 0.01 (Adam Optimization)

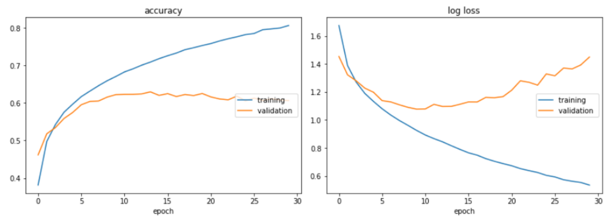
Appendix C



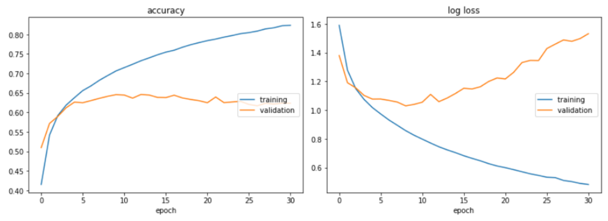
Batch Size = 8



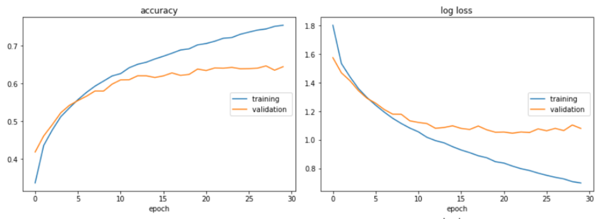
Batch Size = 32



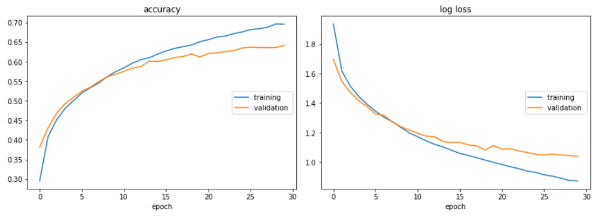
Batch Size = 64



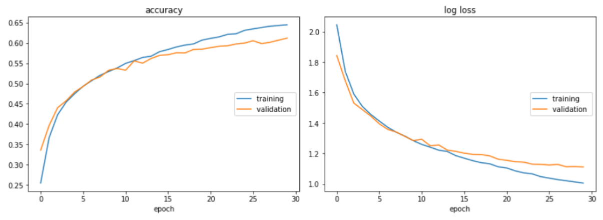
Batch Size = 128



Batch Size = 256



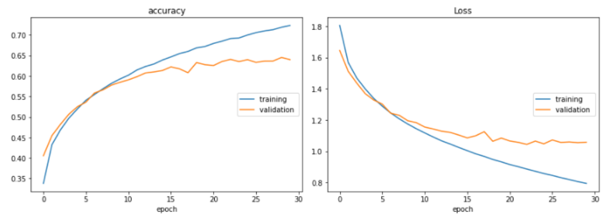
Batch Size = 500



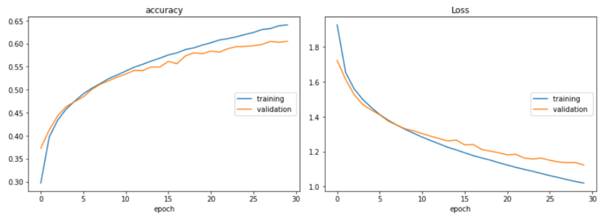
Batch Size = 1000

Figure 8: Accuracy and Loss for Learning Rate = 0.001 (Adam Optimization)

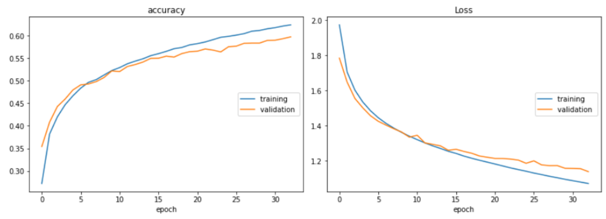
Appendix D



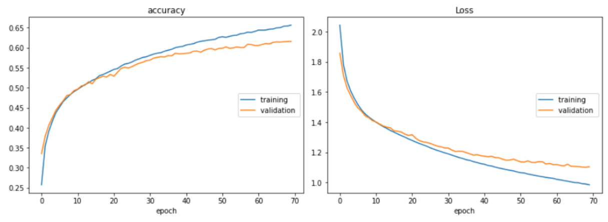
Batch Size = 8



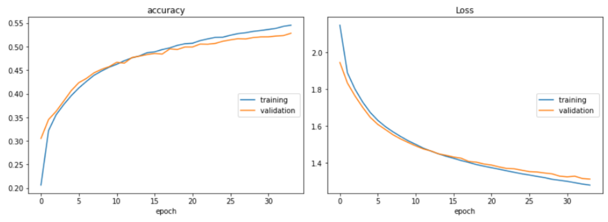
Batch Size = 32



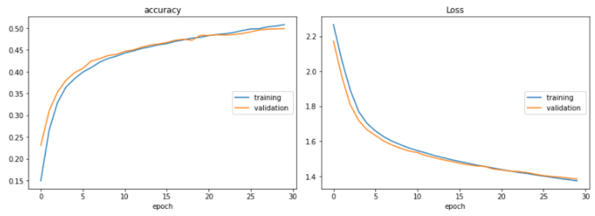
Batch Size = 64



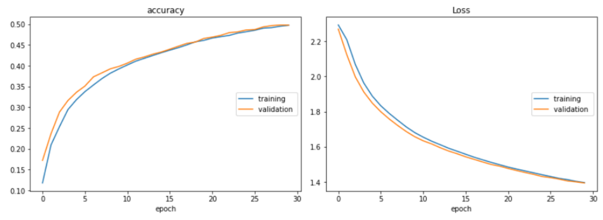
Batch Size = 128



Batch Size = 256



Batch Size = 500



Batch Size = 1000

Figure 9: Accuracy and Loss for Learning Rate = 0.0001 (Adam Optimization)