Deep learning applications in classification problems (2025)

First A. Spyros Kougias, ECE AUTH

***Abstract*—This report aims to explain the processes, methods and hindrances used and encountered during the application of deep learning methods in classification problems, specifically in its potential to classify images. The nature of the project is exploratory with the aim of acquiring practical experience.**

I. INTRODUCTION

THE dataset chosen is cifar-10 which contains 60 thousand images, each belonging to one of ten classes. The data has already been subdivided into a training set (50k) and a testing set (10k).

This paper will firstly use classical machine learning methods and then apply more complex deep learning models, to provide a frame of reference.

More information about the dataset can be found in the Reference [6].

The machine learning techniques used will be:

- 1-NN (nearest neighbor)
- 3-NN (nearest neighbor)
- Nearest Centroid

II. CLASSICAL MACHINE LEARNING

A. Implementation

Firstly, the data was preprocessed by dividing with its mean and scaling down (values from 0 to 1).
The whole image was used as input in the script in the form of a one-dimensional vector. Each test sample's distance was separately calculated from each training sample with the use of the following formula:

$$||x-y||^2 = ||x||^2 + ||y||^2 - 2 \cdot x \cdot y$$

Libraries such as "pytorch" and "sklearn" were used to hasten the calculation of the distances and accuracy scores. The use of PCA was also considered and included in the experiments.

B. Experiments

Each experiment run for a hundred times, the mean of the accuracy score is used to determine the quality of the method in this specific classification problem.

Table 1

pca_dim	chunk_size	use_pca	1NN_acc	3NN_acc	centroid_acc
50	1	TRUE	0.29283	0.31696	0.261985
50	200	TRUE	0.29342	0.31688	0.262015
50	400	TRUE	0.29313	0.31761	0.261995
200	1	TRUE	0.27891	0.28945	0.262355
200	200	TRUE	0.27924	0.29004	0.26237
200	400	TRUE	0.27899	0.28922	0.262355
300	1	TRUE	0.26884	0.28493	0.263
300	200	TRUE	0.26875	0.28475	0.263
300	400	TRUE	0.26886	0.28467	0.263
500	1	TRUE	0.26271	0.27593	0.26279
500	200	TRUE	0.26255	0.27544	0.262835
500	400	TRUE	0.26294	0.27562	0.262845

In table 1 we see the tests run for a variety of pca dimensions and chunks sizes: PCA {50, 200, 300, 500} CHUNKS {1, 200, 400}

Table 2

1NN_std	3NN_std	centroid_std
0.002314011	0.002428451	0.000544624
0.002689974	0.002375294	0.000646226
0.002427348	0.002604949	0.000707062
0.00214191	0.002001893	0.000480895
0.002216194	0.002155191	0.000485125
0.002271897	0.002057654	0.000480207
0.001178254	0.001478841	0
0.001217507	0.001431076	0
0.001154875	0.001511271	0
0.001208514	0.001416034	0.00040636
0.001358661	0.001451714	0.000372989
0.001277782	0.001369067	0.000363662

In table 2 we see the corresponding stds of the previous experiments.

Table 3

use_pca	chunk_size	use_pca	1NN_acc	3NN_acc	centroid_acc
FALSE	1	FALSE	0.282	0.285	0.256
FALSE	200	FALSE	0.282	0.285	0.256
FALSE	400	FALSE	0.282	0.285	0.256

In table 3 we see the tests run for different numbers of chunks, without the use of pca: CHUNKS {1, 200, 400}

As expected, choosing a different chunk size does not significantly affect the accuracy of the classification process, as its intention is only to improve the speed of the script, at the cost of using more memory. Importantly, using PCA improves the speed of the algorithm, but also seems to partially increase its accuracy. This can probably be attributed to the denoising effect it the data.

The conclusion we can draw from these experiments is that we have better results with **3-NN (using PCA)** which shows an accuracy of **31.7%** and **3-NN (without PCA)** with **28.5%.**

We also observe that PCA dimensions, for 1-NN and 3NN specifically, give better results at the smaller end of the spectrum (close to 50). While with the centroid classifier we observe improvements closer to 300 PCA dimensions and not 50.

III. Deep Learning

Now we will look at how using a deep learning model can potentially improve our accuracy and estimation time (only forward, not training time). We will begin by describing the structure of the project and general approach to finding a solution

A. Introduction

The project consists of, mainly, four scripts:

- utils.py
- models.py
- train_evaluation.py
- main.py

While, for the earlier experiments this structure had not yet been established, it was found to be more convenient and much more readable.

We begin with the "first" script, utils. It was originally conceived as a script that simply loaded the dataset, mostly for readability, but was later expanded to also include data transformations (augmentation). The decision to split the dataset into training, validation and testing batches implemented early on, to ensure objective evaluation and combat overfitting (using an optimizer).

Moving on to the model script. It simply houses the architectures that were used for our experiments. The model's structures themselves did not remain rigid throughout the lifetime of this project but constantly changed based on the need of each experiment. When later the need for more formal long running experiments was made obvious, the models were semi-formalized, and arguments were used to more easily switch the activation, criterion and optimization functions (as well as batch size, dropout etc.). These last two can be seen at the "training and evaluation" script.

One of the scripts that observed the most changes was naturally the "training and evaluation" script. It defines the criterion and optimizer that are used during training, implements the training and evaluation loops (and plots the results).

Finally, "main.py" is the control panel, where the arguments are defined and models are loaded. Each part of the project is controlled by this script.

Lastly there are two scripts of lesser importance. The "get stats" script, which calculated the mean and std of the training data (used for normalization). And the experiment script, which runs multiple versions of the same experiment in an attempt to determine the mean accuracy and loss of each method.

B. Models and Experiments

Basic MLP

We begin with a basic Neural Network of 3 layers, input, hidden and output. We started with a simple architecture to establish a baseline. We begin with 3072 inputs for the first layer (which is the flattened three-color image) then 512 inputs for the second layer (3072/6) and of course the final layer has an input of 512 and an output of 10.

Techniques such as dropout and data augmentation were not used (and had not yet been implemented) at this stage of the project. Specifically with data augmentation, we assume that it is almost always beneficial and thus does not contribute towards exploring the other parameters (this assumption is proven wrong at the end of the Basic MLP section). It will only be applied towards the end of each model's exploration.

The settings for this model started off as such:

- Epochs: 100
- Learning rate: 0.001
- Batch Size: 64
- Criterion: cross entropy
- Activation: sigmoid

Since it was the first attempt, suggestions from the web were used. It needs to be stated that the model settings will change as we conduct more experiments.

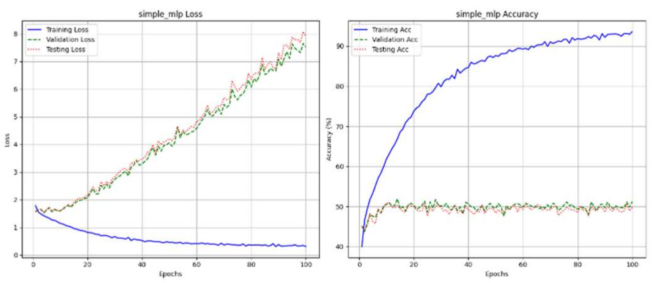
After the first experiment was finished (at 100 epochs) the result was:

Table 4

Metric	Train	Validation	Test
Loss	0.3122	7.5229	7.8962
Accuracy	93.57%	51.22%	49.76%

Although this is a good first attempt it leaves a lot to be desired. The validation batch was not yet used in any meaningful way, but was plotted, nonetheless.

Figure 1



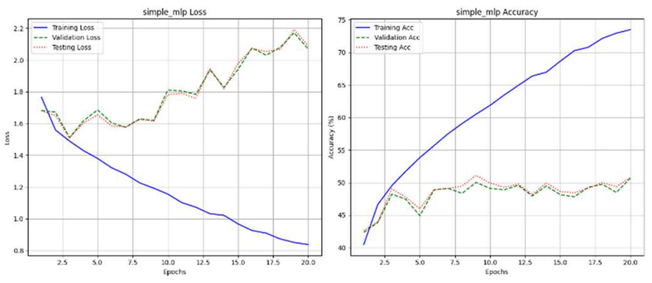
As we can clearly observe, the testing loss keeps ascending with each epoch (after ~15 epochs), while the accuracy seems to improve at the start and finally plateau at 49.76% (again after ~15 epochs).

Later, when the experiment script was established. The same experiment runs ten times resulting in a mean accuracy of 49.27% with a standard deviation of 0.95%.

It is clear from this data that we have a case of severe overfitting. Training accuracy is very high, which implies that the model has successfully memorized the training data. In the plot depicting loss we can observe that while training loss is approaching zero, testing loss rapidly increases after epoch 15. This implies that more than 15 epochs in simple mlp models can be dismissed as unnecessary in this kind of experiment.

The next logical step was to run the same experiment for around **20 epochs** after which we observe more reasonable the testing loss of 2.0880 and a final accuracy of 50.80% (Figure 2).

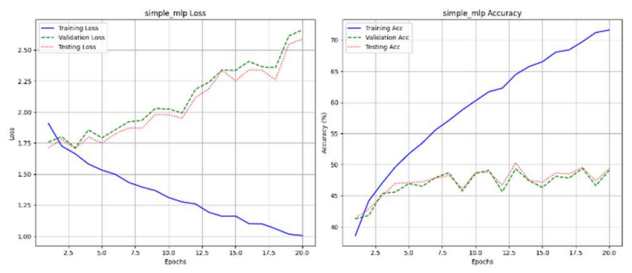
Figure 2



To improve overfitting and accuracy we investigate the number of neurons in the hidden layer. Starting off with **512** we got the above-mentioned results.

Then we run the same experiment as above (learning rate = 0.001, cross entropy etc.) but with a hidden layer of size **1024**. Of course, the number of epochs was chosen to be 20 from the previous experiment's conclusion.

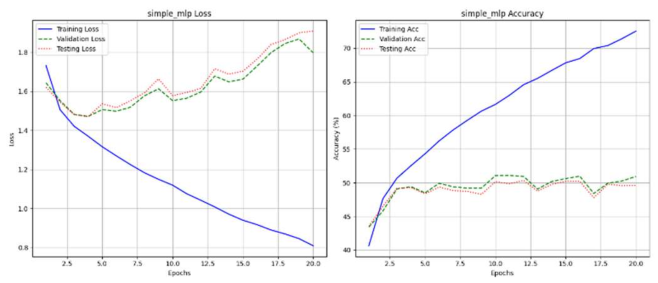
Figure 3



From comparing the two figures (Figures 2 and 3) we observe the main problem with increasing the size of the hidden layer, faster overfitting. We see that testing loss in figure 3 is climbing faster for the same 20 epochs of the experiments when compared to figure 2.

Similarly, we try a size of **256** for the hidden layer. We should observe slower training speed and less overfitting. Because of this expectation we will run an experiment identical to the others, with the exception of the hidden layer size (256) and the number of epochs (30 because we assume the training will take longer).

Figure 4

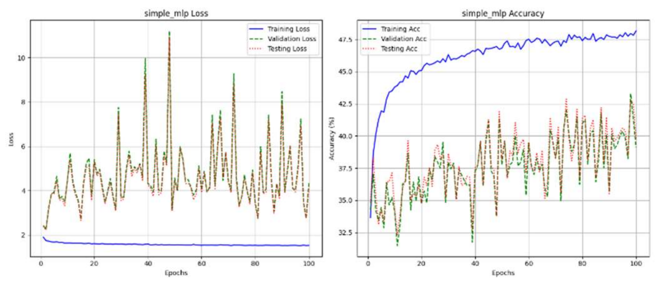


As we had assumed, overfitting is less severe yet still present. The problem though with using such a small number of neurons is that you could bottle neck training without. Thus, we will steer clear from smaller numbers in the following experiments and stick with 512 as the current number of neurons.

An experiment was run for 100 epochs using a hidden layer size 256 and even in a simple architecture like this we observed a small difference in training accuracy (90.2 compared to figure one which is 93). The effects of overfitting were a lot less severe (5.53 instead of 7.52 testing loss).

Lastly, we will run an experiment with augmentation to see how it impacts the results:

Figure 5



Contrary to our assumption when the model cannot handle the extra data (through augmentation) it seems to give highly variable results and poor accuracy.

Complex MLP

We continue with a more complex version of the previous model. Where we will experiment with the number of hidden layers, activation functions, optimizers and criterion functions.

We begin with 3 hidden layers with sizes of 1024, 512, 256 from input to output. We choose those sizes to "funnel" the data from the input layer, towards the output layer. The above decision was made after researching the size of hidden layers as shown in references [1]-[3]. We will be inquiring into the number of hidden layers as our first experiment. Its also worth noting that after every layer we normalize the inputs with batch norm.

We will run experiments with Adam, Adamw and Sgd optimizers after advising Reference [4] and based on Reference [5] will also experiment with relu, leaky relu and sigmoid activation functions. Lastly, we will check how the choice of criterion influences the results and will check: cross entropy, label smoothing and hinge loss.

We begin with the hidden layer experiments and test three variations:

1. [2048, 1024, 512, 256] (4 hidden start)
2. [1024, 512, 256] (3 hidden)
3. [1024, 512, 256, 128] (4 hidden end)

The experiments will have these settings:

- Epochs: 20
- Learning rate: 0.001
- Batch Size: 64
- Optimizer: Adam
- Criterion: cross entropy
- Activation: relu

Figure 6

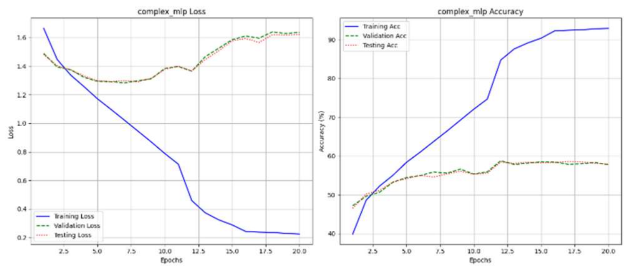


Figure 7

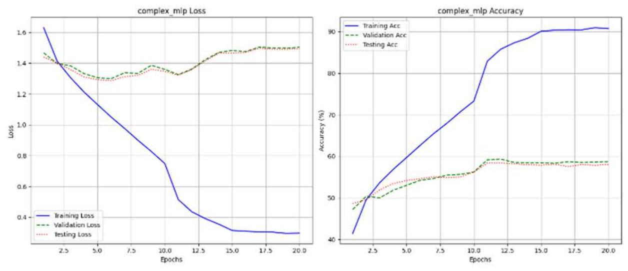
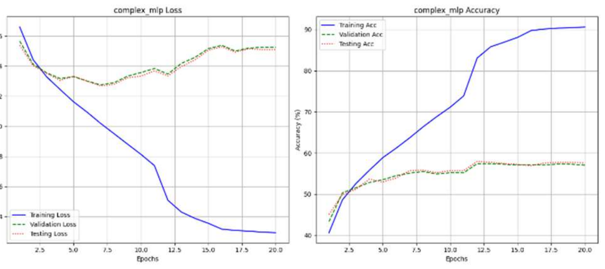
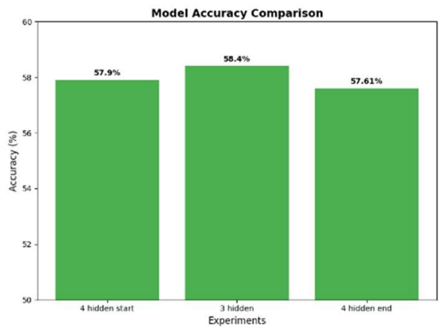


Figure 8



Figures 6-8 are [4 hidden start, 3 hidden, 4 hidden end] respectively. The results show negligible performance difference.

The first configuration (4 hidden start) although having the most neurons did not outperform the smaller models. This indicates that lack of capacity is not the problem. The model seems to have enough neurons to learn, but the MLP architecture cannot extract better features from the input.



The third configuration (4 hidden end) also does not outperform our baseline.

The second configuration is the clear winner since it outperforms the others (albeit slightly) and is less complex.

All configurations exhibited severe overfitting after epoch 6, indicating that the fully connected architecture may have reached its ceiling.

We will use the MLP for testing:

- Optimizers
  - Adam
  - Adamw
  - Sgd
- Activation functions
  - Relu
  - Leaky Relu
  - Sigmoid
- Criterion
  - Cross Entropy
  - Label Smoothing
  - Hinge Loss

And then move on to CNN architecture using the information from the MLP experiments.

The optimizers

The experiments will have these settings:

- Epochs: 20
- Learning rate: 0.001
- Batch Size: 64

- Optimizer: Adam, Adamw, Sgd
- Criterion: cross entropy
- Activation: relu

Figure 9

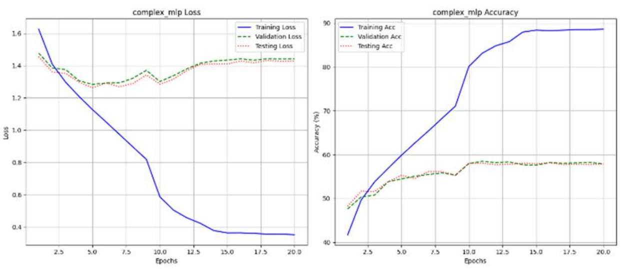


Figure 10

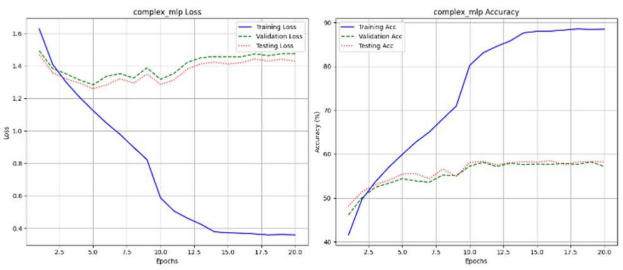
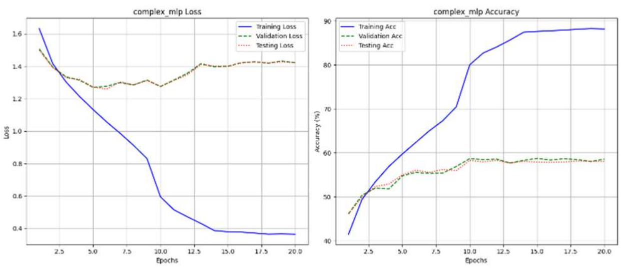
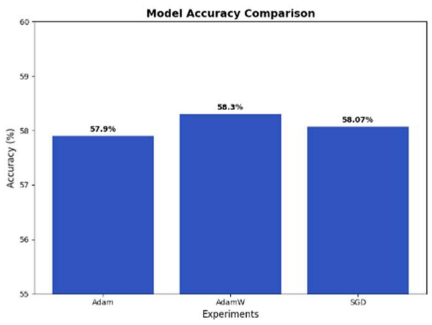


Figure 11



Figures 9-11 are [Adam, AdamW, Sgd] respectively. The results show small differences. The winner seems to be AdamW by a small margin. Although sources seem to consider Adam to be a better fit for this data set [9].



We can also observe a phenomenon that we did not experience with the simple mlp. The sharp vertical jumps at epochs 9-10. These jumps can also be seen in the "number of hidden layers" experiments. This effect is caused by the scheduler [8] that reduces the learning rate based on the optimizer (with validation batch).

Sadly, with all three optimizers the overfitting problem did not subside.

The Activation Functions

The experiments will have these settings:

- Epochs: 20
- Learning rate: 0.001
- Batch Size: 64
- Optimizer: Adamw
- Criterion: cross entropy
- Activation: relu, leaky relu, sigmoid

Figure 12

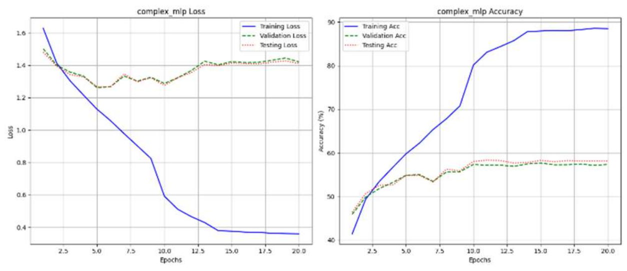
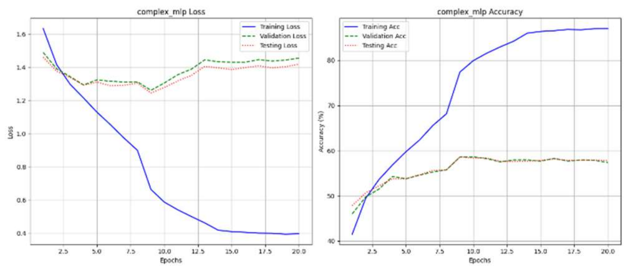
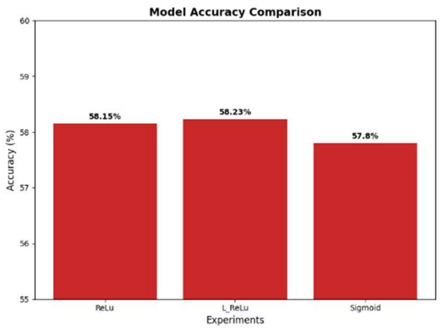


Figure 13





Figures 11-13 are [ReLu, Leaky ReLu, Sigmoid] respectively. Contrary to expectations, the Sigmoid function did not suffer from vanishing gradients and remained competitive with the others. This can be attributed to the use of Batch Normalization in the MLP architecture, which kept neuron activations in the linear part of the Sigmoid curve (close to zero). However, ReLU and Leaky ReLU still converged faster in the early epochs.

The "dying ReLu" problem did not affect our model and so it achieved similar and even slightly better results than Leaky ReLu. Once again, we still have severe overfitting.

Criterion Functions

The experiments will have these settings:

- Epochs: 20
- Learning rate: 0.001
- Batch Size: 64
- Optimizer: Adamw
- Criterion: Cross Entropy, Label Smoothing, Hinge

Loss
- Activation: Relu

Figure 14

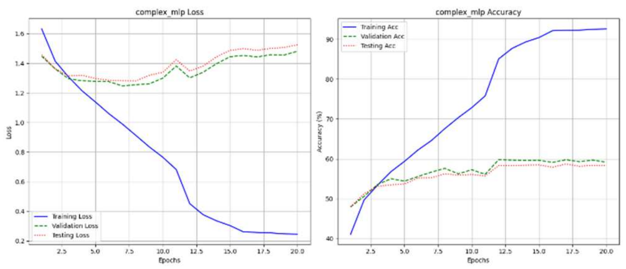
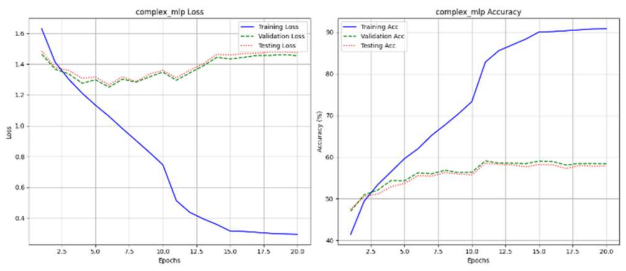


Figure 15

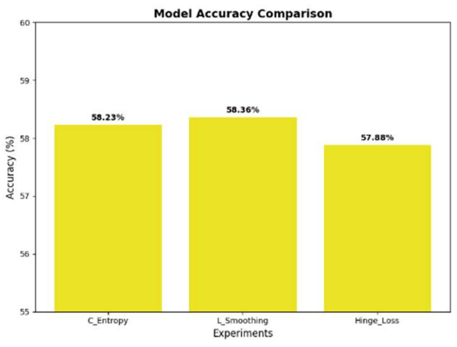


Figures 11,14 and 15 are [Cross Entropy, Label Smoothing, Hinge Loss] respectively.

While Label Smoothing achieved the highest test accuracy, it did not reduce the generalization gap. Instead, the training accuracy increased to 92% (compared to 89% with Cross Entropy), widening the gap.

This tells us that while Label Smoothing helped the model to learn the training data more thoroughly, it did not act as a strong regularizer for this specific architecture.

We observe that Hinge Loss performed slightly worse compared to Cross Entropy. Cross Entropy optimizes the probability of the correct class (encouraging confident answers). While Hinge Loss focuses on creating a margin between correct and incorrect answers [11]. Our results indicate that for cifar-10, cross entropy helps our model learn better features.



Lastly, we will check the effects of data augmentation in the MLP architecture:

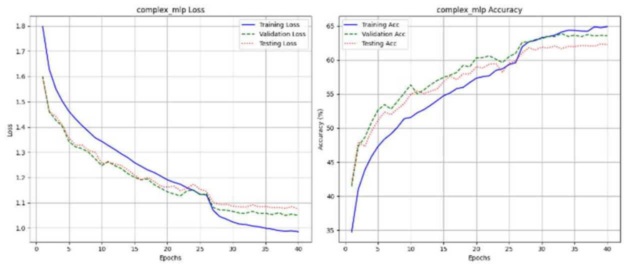


Figure 16

We can clearly see a big improvement in both overfitting and accuracy. This makes sense since we have made our dataset much bigger.

These experiments demonstrate that an MLP architecture, which treats images as flattened vectors and ignores spatial relationships, has a tendency for overfitting. No amount of hyperparameter tuning or capacity increase can overcome this. Even though data augmentation helped it cannot get us past the fundamental flaw of using an MLP in an image classification problem. So, we are forced to move on to Convolutional NNs.

CNN

We continue with a different approach, a model which considers the spatial relationships between the pixels.

Our specific architecture includes Double Convolutions.

Instead of a single layer, the model uses two blocks to scan the image:

- Block 1, we take the 32x32 input and scan it with 32 filters and then 64 and finally, a Max Pooling layer shrinks the image size by half
- Block 2, we now take the 16x16 input and increase the number of filters to 128 and again, a Max Pooling layer to shrink it to 8x8

Lastly the fully connected section flattens the input (128x8x8) and decides the classes (like an MLP). Its worth noting that we use batch normalization in each block and the final “fully connected” section.

We decided on a 2-block structure because of the comparison with the original 1 block design:

The experiments for both will have these settings:

- Epochs: 30
- Learning rate: 0.001
- Batch Size: 64
- Optimizer: Adamw
- Criterion: Cross Entropy
- Activation: Relu

Figure 17

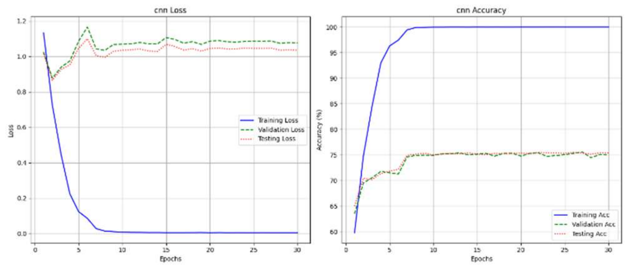
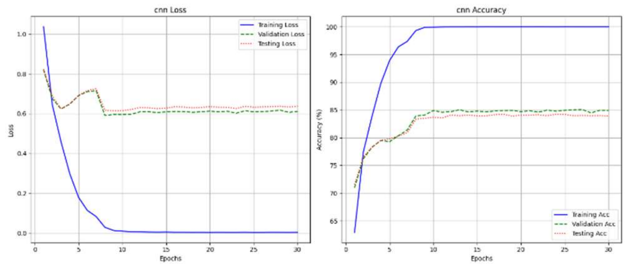


Figure 18



As we can clearly see we get much better results with the 2-block model (Figure 18). It is better with both overfitting and overall accuracy.

Both models (even without augmentation) outperform every MLP model (even with augmentation).

Attention also should be drawn to the drawbacks of using a CNN. The training time is a lot longer and requires more processing power than that of an MLP. Example: The above experiment's 2-block time compared to the previous MLP is [352.03 seconds vs 137.26 seconds]

Dropout

We will move on with dropout and data augmentation. Since these techniques are complimentary

The experiments will have these settings:

- Epochs: 40, 60
- Learning rate: 0.001
- Batch Size: 128
- Optimizer: Adamw
- Criterion: Cross Entropy
- Activation: Relu
- Dropout: 0.0, 0.5, 0.8
- Augmentation: True

Figure 19

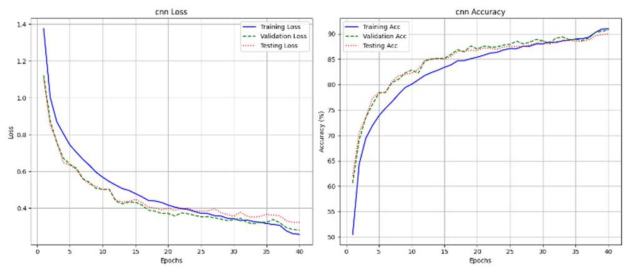
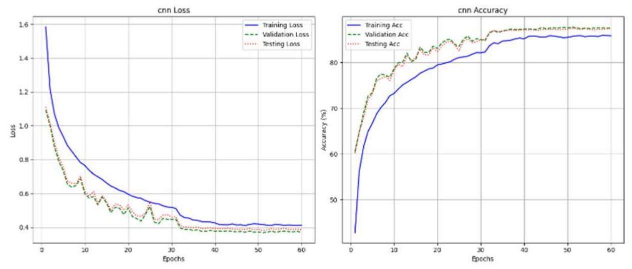


Figure 20



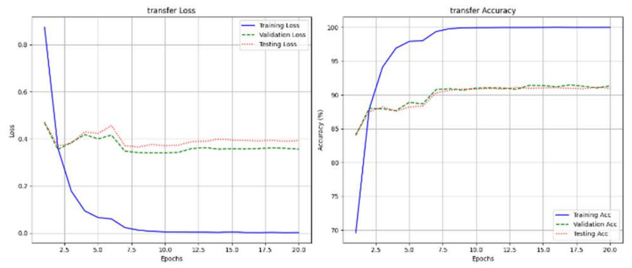
The application of Dropout + Augmentation yielded significant performance gains. A dropout rate of 0.5 proved to be the optimal setting, achieving an accuracy of 89.96% with converging loss curves that showed no signs of overfitting.

When the dropout rate was increased to 0.8, we observed a bizarre phenomenon where the test accuracy consistently exceeded the training accuracy. This occurs because the network is heavily handicapped during the training phase, with 80% of its neurons deactivated at any given step, severely limiting its capacity to memorize the data. However, during the testing phase, the dropout is disabled, allowing the model to function normally. While this prevented overfitting, it ultimately led to underfitting, resulting in a lower final accuracy of ~87% compared to the ~90% achieved with the 0.5 rate.

<u>Transfer Learning</u>

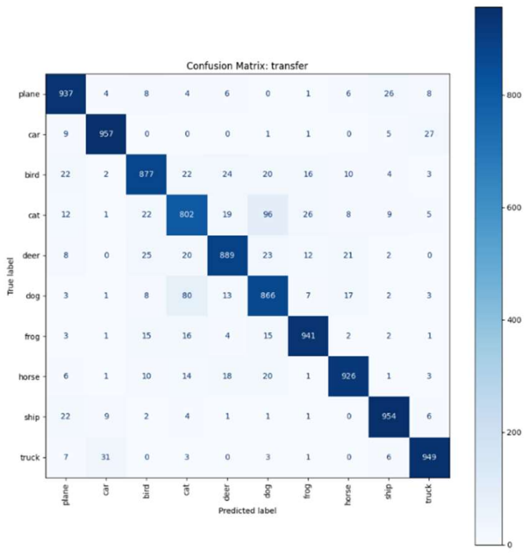
Lastly, we will try to re-train a pre-trained model, specifically ResNet18. The main issue was re-shaping the input to be compatible with our 32x32 images. After some small-scale tests (smaller batches of training and testing data), we concluded that the best option is to unfreeze all the weights and add dropout to the replaced clarifier head.

Figure 21

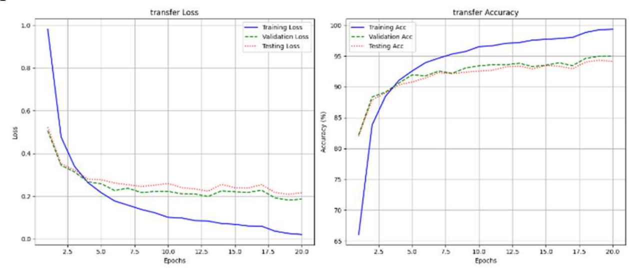


These are the best results we have seen so far in terms of accuracy and loss curves. By modifying the first convolutional layer to accommodate the 32x32 input size and unfreezing the pre-trained weights, the model used ImageNet feature extractors to achieve convergence, reaching over 95% training accuracy by the third epoch.

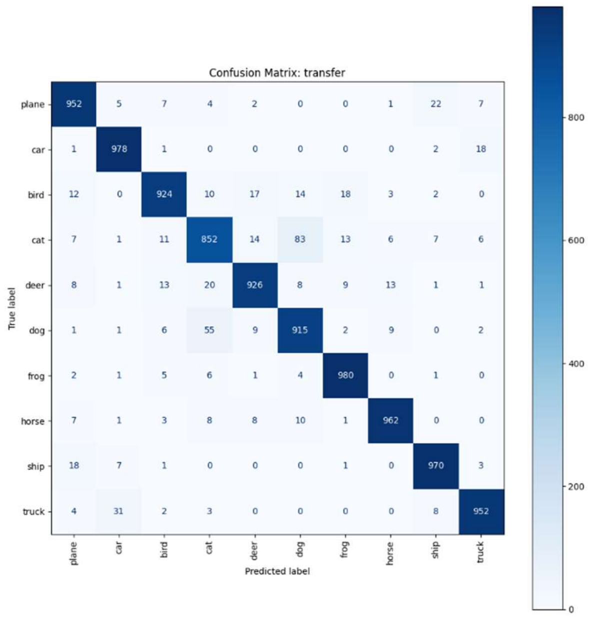
The confusion matrix shows that the model is almost perfect at recognizing distinct objects like Cars and Ships, but struggles slightly with similar-looking animals, confusing Cats and Dogs about 10% of the time. The model achieves a 100% training accuracy compared to 91% on the test data, which proves it has memorized the examples rather than fully learning the patterns; this suggests that adding data augmentation in the future would help it perform even better.



As expected, adding augmentation truly squeezed as much as possible from the model.



We reached a testing accuracy of 94% with a training time of 589.45 seconds way higher than any other experiment.



While the issue of the similar-looking animals did not much improve.

IV. Conclusion

In this project, we started with classical machine learning and simple Neural Networks to establish a baseline, but we quickly hit a wall. Our experiments with MLPs showed that no matter how many neurons we added or how much we tweaked the optimizers, we couldn't break past 58% accuracy. Thus, we moved to Convolutional Neural Networks (CNNs). Because they actually look at the spatial structure of the image, our 2-block model immediately outperformed the MLPs. We also learned that using Data Augmentation together with 50% Dropout was the sweet highly effective for CNN models pushing our accuracy up to 89.96%. Finally, Transfer Learning model (ResNet18), with augmented data, finally pushed us past 94% accuracy, proving that leveraging pre-existing knowledge is the most efficient way to solve complex tasks.

What we can learn from these experiments is that one must consider the problem they are trying to solve. We should never have expected that an MLP could categorize images with any comparable success to a CNN. This stems from the fact that CNNs can extract information from the input through the spatial relationships of the pixels. When an MLP flattens the image, it loses all that important spatial information that makes a CNN so successful in these types of tasks.

We can also observe the importance of balancing the amount of data with the complexity of a model. Augmentation, for example, produced horrendous results in the simple MLP but broke the perceived performance ceiling of our complex models.

References

[1] "Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks" ISBN: 1505714346.

[2] https://www.reddit.com/r/learnmachinelearning/comments/77vxsc/how_do_we_know_how_many_hidden_layers_and_how

[3] https://stackoverflow.com/questions/52485608/how-to-choose-the-number-of-hidden-layers-and-nodes *(Note: I am aware that I am citing internet forums)*

[4] https://neelesh609.github.io/cifar10

[5] https://sowjanyasadashiva.medium.com/cifar-10-dataset-864a87413e22

[6] https://www.cs.toronto.edu/~kriz/cifar.html (cifar 10 website)

[7] https://stackoverflow.com/questions/51017181/what-should-be-the-value-for-learning-rate-and-momentum ("Generally used values lr = 1e-4, momentum=0.9")

[8] https://wiki.cloudfactory.com/docs/mp-wiki/scheduler/reducelronplateau

[9] https://www.datacamp.com/tutorial/adamw-optimizer-in-pytorch

[10] https://github.com/adhishthite/cifar10-optimizers?tab=readme-ov-file (CNN ideas)

[11] Concepts of Loss Functions - What, Why and How at: https://www.topcoder.com/thrive/articles/Concepts%20of%20Loss%20Functions%20-%20What,%20Why%20and%20How