# Problem 1: CNN Training on LeNet-5

## Motivation

Convolution Neural Network (CNN), is really just an empirically verified combination of constituent networks and procedures as layers. In our case, LeNet-5 consists of 2 convolutional layers, 2 max-pooling layers, and 3 fully-connected dense layers, as shown: The purpose of each of these layers will be discussed in (a).

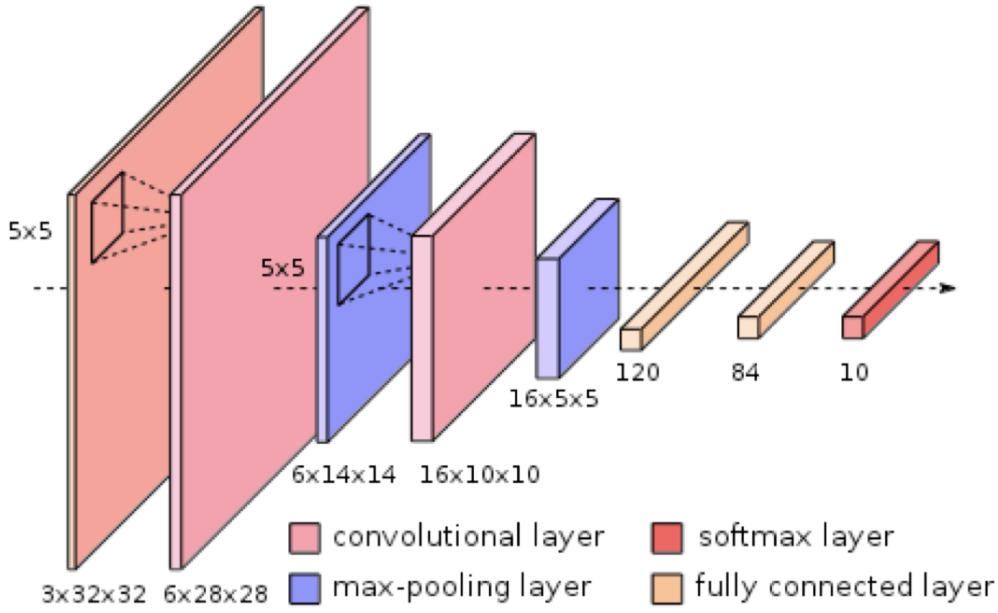


Figure 1: LeNet-5

## Methods

*tensorflow*2 *keras* is used as the driving package in this assignment. Datasets *mnist*, $fashion\_mnist$, and $cifar10$ are all built into the $tensorflow.keras.datasets$ module. All images are normalized to [0,1] by dividing each pixel by 255 to stabilized training. The network implementation is exactly the same as instructed, with the same network dimensions as given in the figure above, $softmax$ activation for the final output dense layer, and *ReLU* activation for all other layers (except for the pooling layers which don't need activation). The loss function used is categorical cross-entropy loss, which is standard for image classification task and corresponds mathematically with softmax function.

## Results and Discussion

**(a)** (1) The role played by convolutional layers in LeNet-5 is feature extraction. Convolution, being a local operation, encapsulates only local information within a small window and therefore makes the features sensitive to spatial variations. This is somewhat reminiscent to how locally-averaged Laws filter features were used to cluster pixels in texture segmentation. The max-pooling layers help convolutional layers in feature extraction by down-sampling convolutional layer outputs,

which not only reduces feature dimension and makes learning easier, but also makes the extracted features more robust to small local changes. The fully-connected dense layers learn to combine these extracted features. The output of the last of these layers is compared to the label of the input data, and through some kind of loss function (in our case cross-entropy loss), calculates a loss which is propagated all the way back to the first convolutional layer, where if the loss is high, larger weights which contrinute to this high loss will be reduced and vice versa. The activation function for convolutional and dense layers are needed because it is the ingredient for non-linearity which enables more complex combinations of features. Finally, softmax function is needed at the last dense layer because the last layer needs to output a one-hot vector to represent its predicted classification. In order to circumvent local optima and prevent diminishing gradient, all categories are given some probability to be predicted, with bias towards the largest element in the output vector.

(2) Overfitting is when the network fits the training data very well but fails on the testing data. Imaging a case where the target function to be predicted is simply a straight line and 3 points on the line are given as training data. Ideally, the model should learn the straight line passing all 3 points, but if it is trained for too long, it may learn, for example, a wiggly sinusoidal curve that passes all 3 points, but deviates from the correct function. There are many ways to combat this. For example, a simple method is to split the training data and use only some of it to train, while the rest are reserved for evaluation. Note this doesn't mean having less available training data, since this can be run for many epochs and each epoch can select a different subset for training, such that in the end, the model would have seen all training instances. This is technique is called cross-validation. More complex methods include restricting weight values and dropping out some weights each epoch. Both combat overfitting by preventing the model becoming overly reliant on some weights.

(3) ReLU can be defined as $ReLU(x) = max(0, x)$.[1] As shown in Figure 2, it consists of a linear

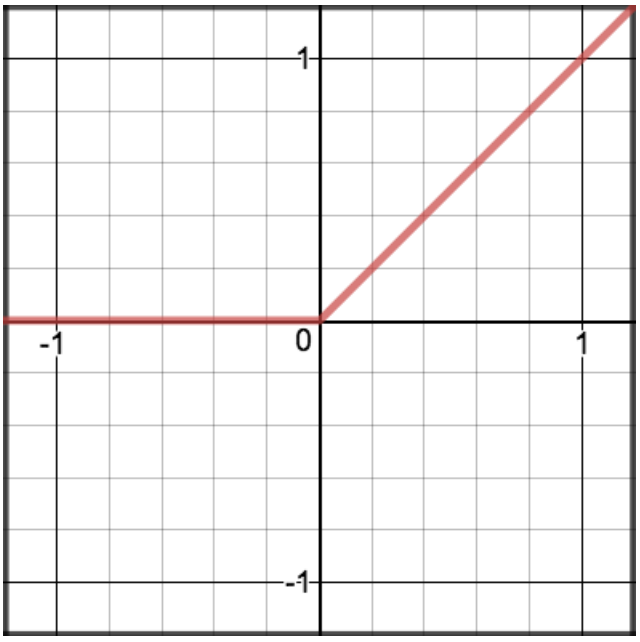


Figure 2: ReLU

positive part and zero negative part, which means nodes that output negative values pre-activation

[1]Images are taken from https://ml-cheatsheet.readthedocs.io/en/latest/activation$_f$*unctions.html*

will be ignored. This has the advantage of preventing cases such as when two negative values cancel each other and confuse the training, but this would also mean nodes that produce negative values pre-activation will not receive gradients in back propagation and essentially become dead. This is known as the dead ReLU problem.
Leaky ReLU addresses dead ReLU by giving the negative part a very small gradient, defined as $max(0, x) + min(0.01x, 0)$. As shown in Figure 3, since the gradient for the negative part is small,

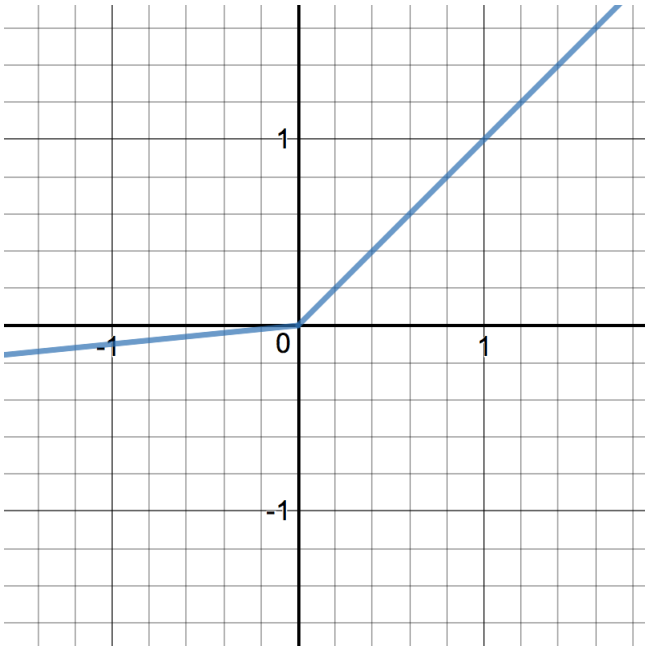


Figure 3: Leaky ReLU

hopefully the network will be able to distinguish positive from negative nodes (though this often turns out to be false in practise). And since there's now some gradient for the negative part, dead nodes can recover.
ELU follows this trend of allowing some gradient for the negative part to combat dead ReLU problem, and can be defined as $max(0, x) + min(e^x - 1, 0)$ As shown in Figure 4, it improves over leaky ReLU because the curve for the negative part flattens out towards more extreme negative values, so dead nodes will still be able to recover, while those that have already become very negative don't affect the network too badly.

(4) L1Loss is defined as $\Sigma_{i=1}^{n}|y_{true} - y_{pred}|$. MSELoss is defined as $\frac{1}{n}\Sigma_{i=1}^{n}(y_{true} - y_{pred})^2$. These two loss functions are typically used in regression. Since in regression, the goal is to make all $y_{pred}$ as close to all $y_{true}$ as possible, both loss functions use some kind of difference between $y_{pred}$ and $y_{true}$, then either sum or average them across all points. BCELoss is defined as $-\frac{1}{n}\Sigma_{i=1}^{n} y_i * log(p(y_i)) + (1 - y_i) * log(1 - p(y_i))$. It is a specific case of cross-entropy loss and is typically used for classification. We know entropy is a measure that minimizes when the all its items have equal probability of occurrence (i.e. completely random). Cross-entropy gets inspiration from entropy, and it minimizes when the predicted probability distribution perfectly matches the true probability distribution.

**(b)** (1) I tried manipulating 3 parameters: kernel(weight) initialization, starting learning rate, and learning rate decay. I first tried having uniform initialization with a relatively high learning rate and decay. As shown in Figure 5, this results in large accuracy variances, especially for the first epoch,

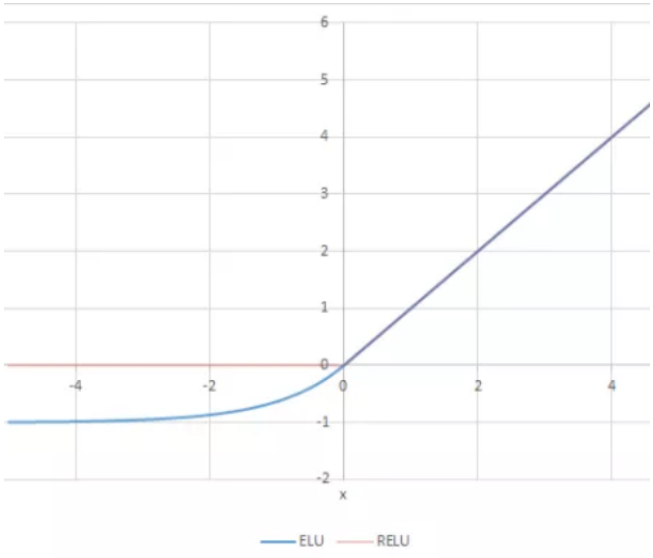


Figure 4: ELU

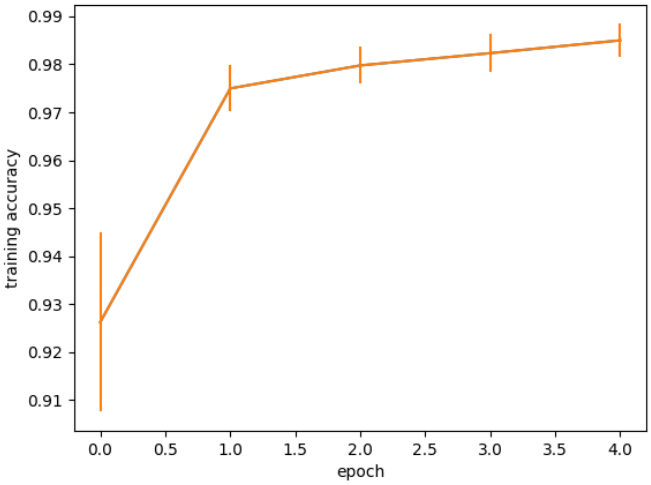


Figure 5: Uniform initialization; 0.01 starting learning rate; 0.9 decay; best 0.9879833459854126; means $[0.92627001, 0.97497667, 0.97977, 0.98232999, 0.98499666]$; stds $[0.0186567, 0.0048434, 0.00386093, 0.00393309, 0.00343504]$

since the initial weights are random and need some time to settle. The accuracy mean also increases more rapidly at the start because of the high learning rate, but it also peaks early and stops improving after some epochs. Then I tried glorot normal initialization, with small starting learning rate and decay, as shown in Figure 6. Glorot is the default initializer and gives a better

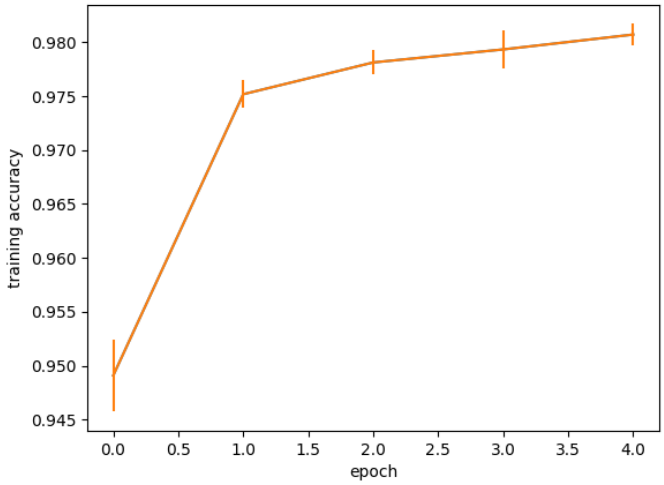


Figure 6: Glorot normal initialization; 0.003 starting learning rate; 0.99 decay; best 0.9818833470344543; means $[0.94908667, 0.97517999, 0.97812666, 0.97933666, 0.98071667]$; stds $[0.00335396, 0.00127772, 0.00113434, 0.00173598, 0.00099034]$

accuracy at the end of first epoch. Lower learning rate actually doesn't harm the early accuracy mean increase in this case, possibly because of the better initialization, and the low decay means accuracy peaks later and model continues to improve for more epochs. Finally I tried truncated normal initialization, with moderate starting learning rate and decay, as shown in Figure 7. This has the best performance of the three, possibly because the starting weights kept within a smaller range and doesn't bias the training too much, while the learning rate and decay achieves a good balance between changing the weights just enough to let them improve, and not too much such that existing progress gets destroyed.

(2) This plot is obtained by training for 20 epochs with the best parameters in (1), i.e. truncated normal initialization; 0.005 starting learning rate; 0.9 decay.

(3) The parameters used here are glorot normal initialization, 0.005 starting learning rate, and 0.95 decay. I trained for 30 epochs in this case, though by hindsight, 20 would have been enough, since a trend of overfitting can be observed after 20 epochs.[2]

(4) The parameters used here are glorot normal initialization, 0.002 starting training rate, and 0.9 decay. Batch size was also increased to 128.

(5) Aside from the obvious fact that CIFAR-10 generally has lower accuracies than fashion-MNIST, which in turn has lower accuracies than MNIST, we may also observe that fashion-MNIST and CIFAR-10 are significantly more likely to overfit than MNIST, which is exhibited in the larger gap between training and testing accuracies. This is possibly because of the truth models are more complex for these two datasets. In my parameter selection, I tried to tackle this by using smaller

[2]You may observe that the testing accuracy at epoch 30 is below 0.9; however, I used keep best, so best accuracy is actually the one at 16th epoch 0.90060.

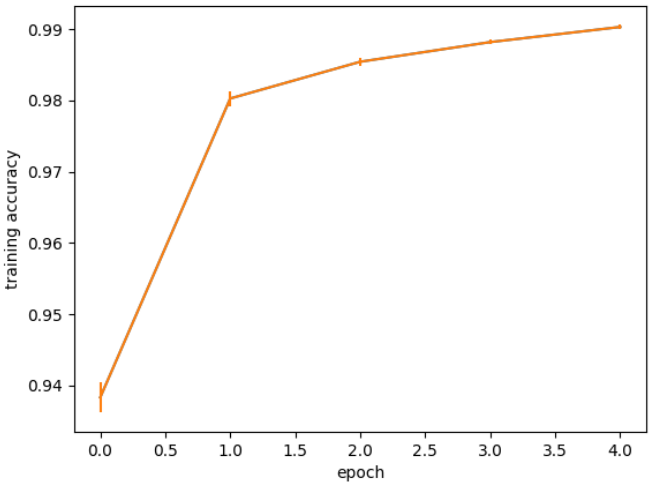


Figure 7: Truncated normal initialization; 0.005 starting learning rate; 0.9 decay; best 0.9906666874885559; means $[0.93829999, 0.98027667, 0.98543668, 0.988196660.99032]$; stds $[0.00213996, 0.00106305, 0.00057865, 0.00034212, 0.00029503]$

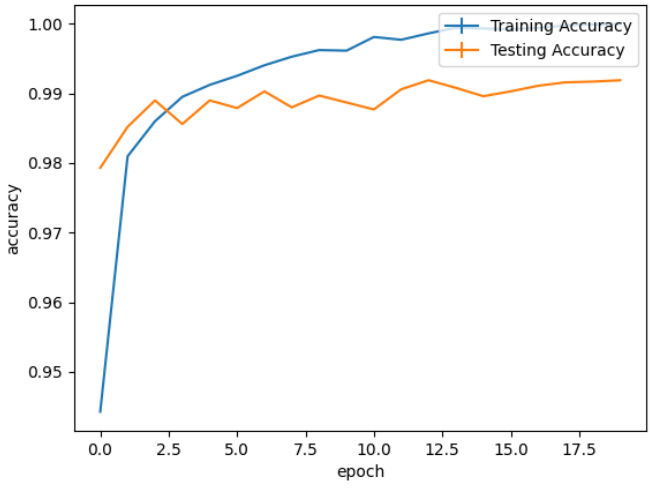


Figure 8: MNIST accuracies

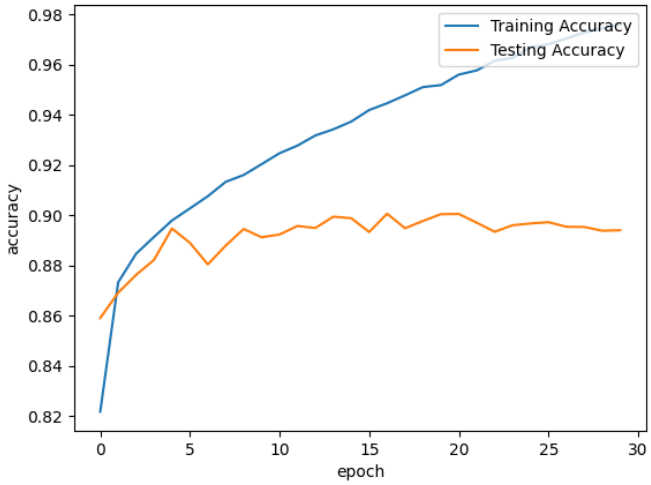


Figure 9: fashion-MNIST accuracies

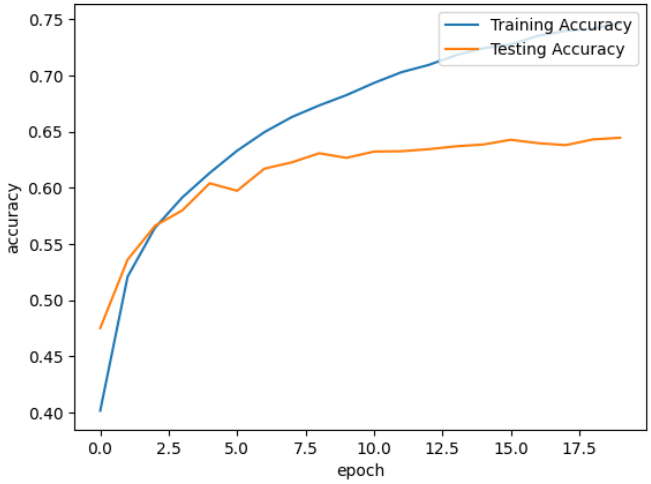


Figure 10: CIFAR-10 accuracies

learning rates. This is especially obvious in the CIFAR-10 plot, where the testing accuracy rises smoothly due to the small learning rate. The other two used higher learning rate, which caused accuracies to fluctuate.

**(c)** (1) The confusion matrix is shown in Figure 11.[3] The top 3 confusion pairs are $4-9$, $5-3$, and $9-4$

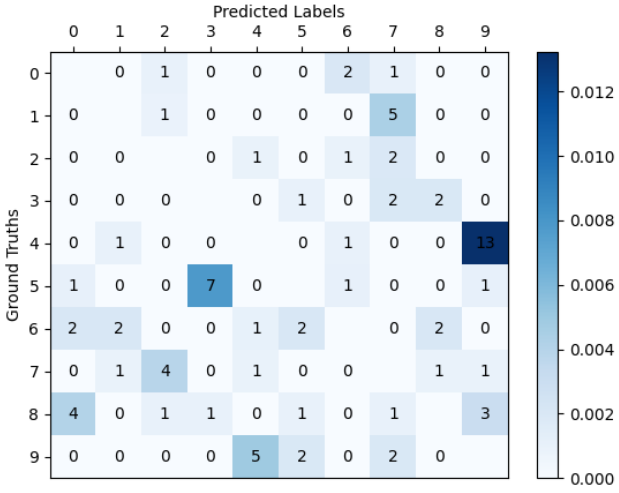


Figure 11: MNIST confusion

(presented in the format $truth - prediction$; the rest of this writeup will follow this convention), as shown in Figure 12. These are some messy examples, so it's not hard to imagine why the


Figure 12: MNIST confusion pairs

model failed on these cases. It's noteworthy that if a $truth - prediction$ pair is confused often,

[3]The plot shows a non-scaled confusion matrix with diagonal cells omitted. This is for cleaner display since the scaled values are riddled with scientific notation. Please see **mnist_conf.txt** if you want to see the scaled confusion matrix.

its opposite pair with the truth and prediction classes reversed also tends to be confused often.

(2) The confusion matrix is shown in Figure 13.[4] The top 3 confusion pairs are $Tshirt - shirt$,

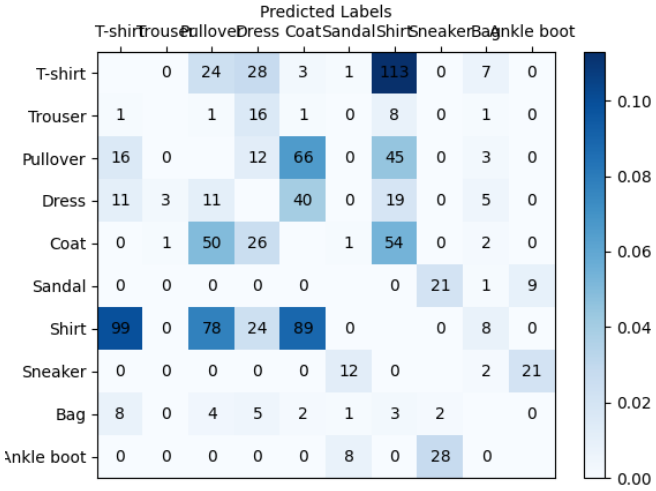


Figure 13: fashion-MNIST confusion

$shirt - Tshirt$, $shirt - coat$, as shown in Figure 14. This confirms our observations in (1).


Figure 14: fashion-MNIST confusion pairs

(3) The confusion matrix is shown in Figure 15.[5] The top 3 confusion pairs are $cat - dog$, $dog - cat$, $automobile - truck$, as shown in Figure 16.

**(d)** (1) I implemented noisy training data by going through all rows in the training data, which consists of one-hot vectors, and generating a probability distribution for each row where the correct label

[4]Likewise, please see **fashion_conf.txt** for the scaled version.
[5]Likewise, please see **cifar_conf.txt** for the scaled version.

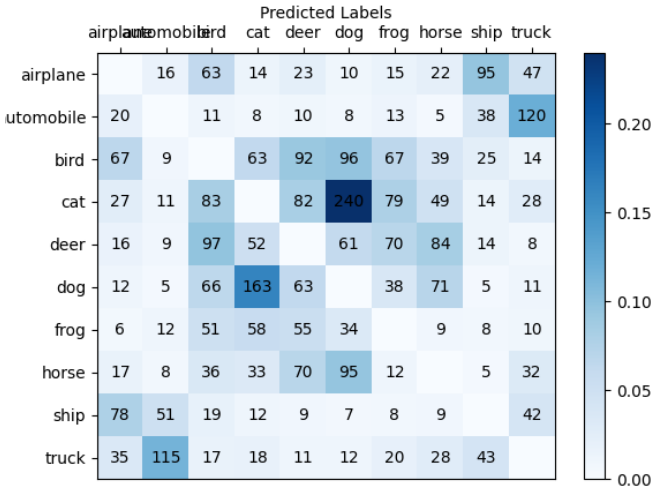


Figure 15: CIFAR-10 confusion


Figure 16: CIFAR-10 confusion pairs

has probability $1-\epsilon$ and all other labels have uniform probabilities $\frac{\epsilon}{9}$. Then I simply sampled from this distribution to get the new label, and the corresponding one-hot vector for the row. The scaled confusion matrix is shown below. I'm guessing the confusion matrix here will be used to affirm whether I implemented the noise correctly, so the confusion matrix shown here was generated for the training set, not the testing set. As shown, the probability distribution for each row approximately follows the definition.

$$\begin{bmatrix}
0.597 & 0.051 & 0.045 & 0.047 & 0.045 & 0.041 & 0.040 & 0.045 & 0.043 & 0.046 \\
0.037 & 0.634 & 0.039 & 0.043 & 0.041 & 0.038 & 0.043 & 0.048 & 0.041 & 0.037 \\
0.037 & 0.049 & 0.588 & 0.048 & 0.044 & 0.041 & 0.046 & 0.054 & 0.047 & 0.045 \\
0.047 & 0.043 & 0.043 & 0.602 & 0.039 & 0.043 & 0.041 & 0.055 & 0.043 & 0.044 \\
0.048 & 0.054 & 0.044 & 0.047 & 0.583 & 0.038 & 0.046 & 0.047 & 0.043 & 0.049 \\
0.047 & 0.053 & 0.043 & 0.049 & 0.045 & 0.565 & 0.044 & 0.052 & 0.051 & 0.050 \\
0.040 & 0.052 & 0.046 & 0.049 & 0.043 & 0.038 & 0.601 & 0.043 & 0.045 & 0.043 \\
0.044 & 0.048 & 0.043 & 0.047 & 0.038 & 0.040 & 0.045 & 0.609 & 0.045 & 0.042 \\
0.045 & 0.049 & 0.045 & 0.045 & 0.049 & 0.041 & 0.049 & 0.045 & 0.585 & 0.047 \\
0.048 & 0.050 & 0.040 & 0.044 & 0.048 & 0.043 & 0.041 & 0.049 & 0.043 & 0.593
\end{bmatrix}$$

(2)

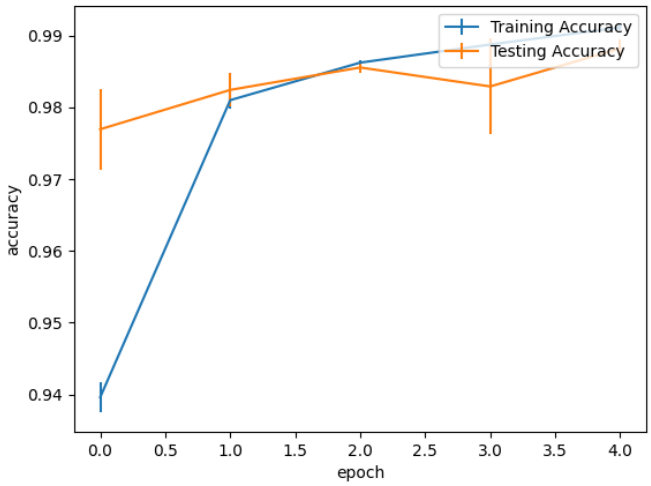


Figure 17: Accuracies $\epsilon = 0$

(3) Noises to the training data damage training accuracies, as expected, but it is interesting to observe that smaller noises don't destroy testing accuracies as one might expect. Only when $\epsilon = 0.8$ did the testing accuracy drop catastrophically. This shows the algorithm's robustness with outliers. I think this robustness can be attributed to the ReLU activation function, where if a node gets a very negative gradient at an iteration, likely because of the outlier, it stays zero and will henceforth be ignored by the network. $\epsilon = 0.8$ broke the network because there were just too many dead nodes. It can also be observed that the variances also increased as $\epsilon$ increased, though again, it was a graceful process until $\epsilon = 0.8$, possibly for the same reason.

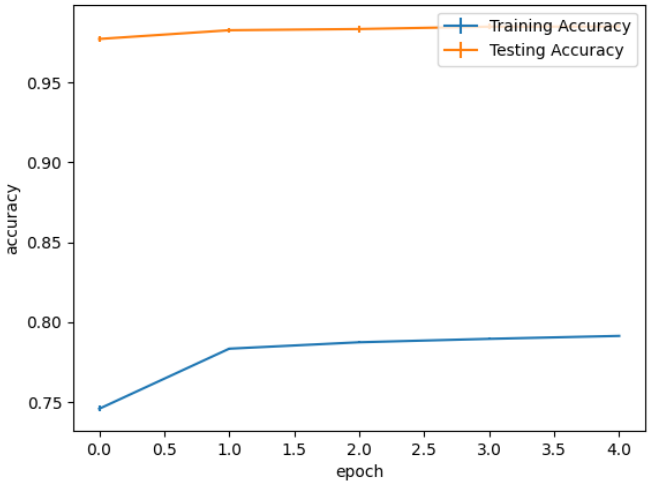


Figure 18: Accuracies $\epsilon = 0.2$

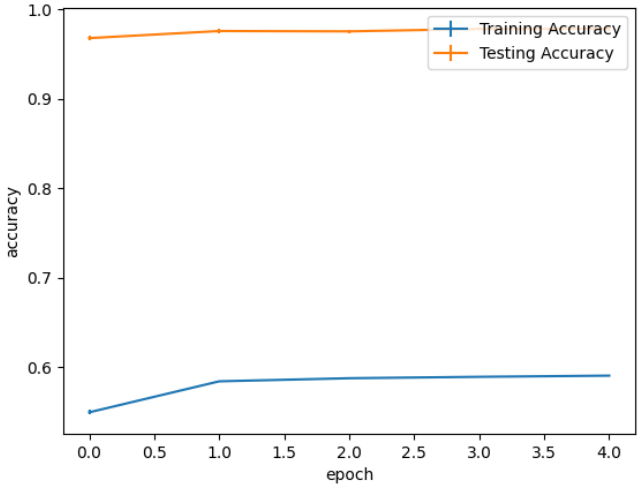


Figure 19: Accuracies $\epsilon = 0.4$

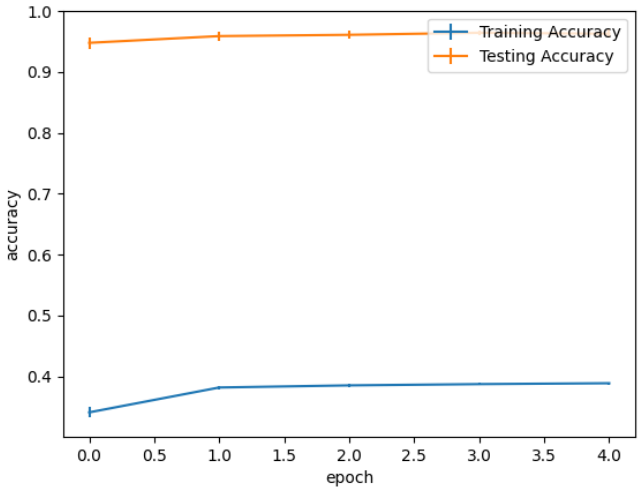


Figure 20: Accuracies $\epsilon = 0.6$

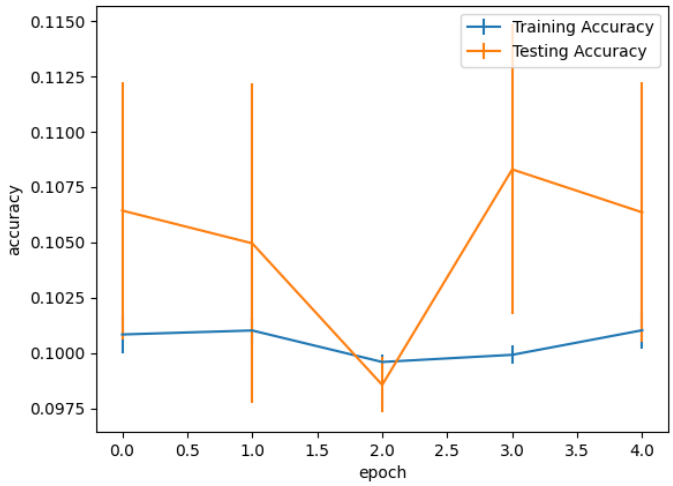


Figure 21: Accuracies $\epsilon = 0.8$