Fashion MNIST Classification

Yash Khanchandani, Dylan Thibault, Guiragos Guiragossian
CSS 485: Artificial Neural Networks

1. Introduction

This report will document the culmination of our artificial neural-network architecture exploration. Our primary focus is the implementation of an efficient and accurate classifier of the fashion-MNIST database through an expanded multilayer perceptron neural network with backpropagation. We wish to demonstrate our thorough understanding of our learning algorithm as it is expanded by a variety of optimization techniques. Lastly, this project will serve as a synthesis of our ANN course knowledge and current academic literature, showing our ability to understand and apply network improvements outside of the course content.

We have broken the report into two overarching sections featuring: first, a brief description of our base layer sans any advanced techniques, second, a dive into some of the optimization techniques which we explored. Each section is further broken down into a methodology (methods) section and a results section describing our MATLAB implementation and the subsequent accuracy (MSE over time) of various elements in our architecture. Lastly, within our optimization discussion, we will specifically explore the effects of varying layer and hidden unit counts, varying hyperparameters, data preprocessing and augmentation, momentum, variable learning rate, and other smaller features we utilized.

2. Multi-Layer Perceptron Base Layer

Even though our base layers were thoroughly explained in our previous report on backpropagation, we felt a brief description to be needed to be able to place and explain our optimizations. Further, we will also discuss the brief changes our base layers saw that were not strictly related to network optimization, such as changes to the data import functionality for example. Lastly, a brief results section focusing on our initial forays with changing hyperparameters and a dead-end optimization (mini-batching).

2.1 Methods

Our fundamental goal was to appropriately expand on the structure which we used for the significantly easier MNIST digit recognition problem. To do this, we first thoroughly verified the highest accuracy each of our respective applications saw when confronted with the MNIST digit database. Unfortunately, each implementation proved to be too different to fully merge into a singular version. Instead, we merged the features which we thought to be effective in our solutions and decided to work in parallel. We collaborated to craft a final singular version which was a culmination of all our optimizations by applying the most successful optimizations from each project to our final combined one. Some optimizations were specifically chosen and isolated into a separate project entirely so that their effect could be studied more effectively. This approach allowed us to better understand individual changes as well as work and run our networks in parallel. The main difference between these versions was the role of the NN layer in

execution and where components were located within it. In the version crafted by Guiragos' the neural net's functions were compartmentalized and represented by distinct methods (see ***Appendix A.2***). The execution of the layer was not centralized in a method local to it but rather within a separate MATLAB file responsible for execution**.** Thus, each step of the backpropagation algorithm was executed sequentially in the execution file.

Comparatively, Yash's version had a more universal training method, requiring only a single call in the execution file to run. These differences did not affect our network performances; they simply came down to preference. Hereafter, we will not describe the version which the code snippet lies in (unless that is crucial to understanding its function). Instead, we will refer to concepts and modifications generally as in both versions they boiled down to changing the value of a given variable. This section will mainly focus on the code generated by Guiragos' version simply because its execution order is easier to visually understand (though more frustrating to modify).

Within the execution file, layers are individually declared with the activation function and neuron count as parameters. Further, before beginning network functions load the data and convert the labels into their respective one-hot encoded variant. Note, instead of targeting binary (0 and 1) values, we target 0.1 and .9 respectively. This was done as per the suggestion of Jason Brownlee in (Brownlee, 2019) as it would provide easier values for our network to converge on (given sigmoid activation). This change did not noticeably hurt or improve performance but was kept as we trusted the literature on it. Finally, note that data was given in the CSV format so we have to convert CSV to table and table to array before finally receiving a matrix of inputs (p) which can be trained on. We ensured that we were properly loading images by generating them with our input matrix as can be seen in ***Figure 1*** and ***Figure 2***.

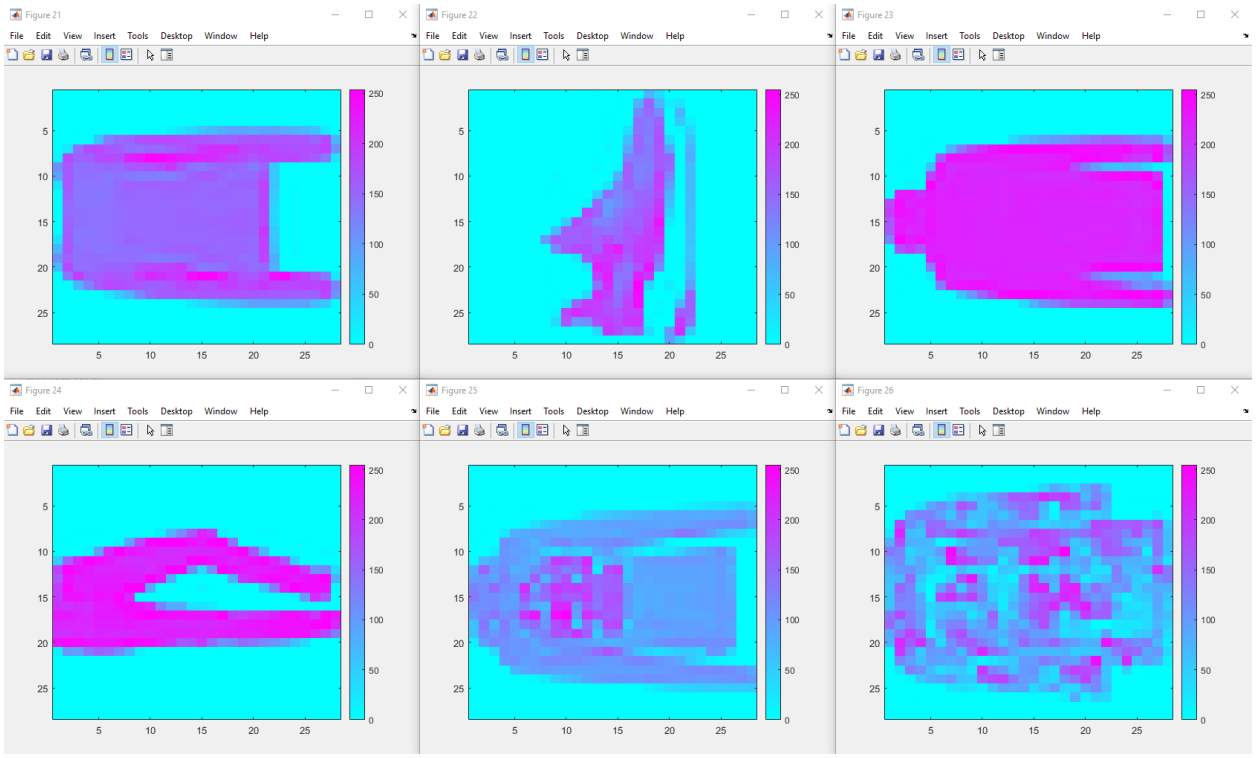


Figure 1: The first 6 images without rotation from the fashion training set - sideways

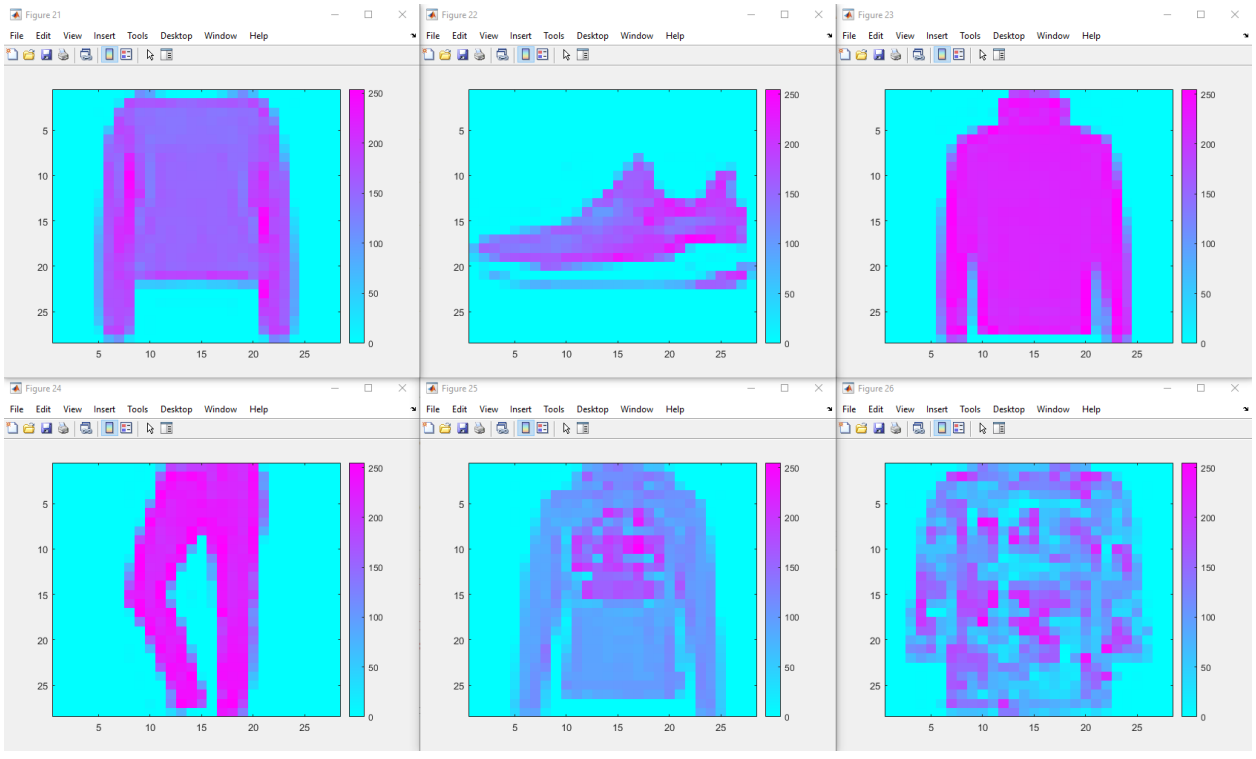


Figure 2: The first 6 images with rotation from the fashion training set - upright

The basic execution of the network can be seen in the overarching while loop which increments the epoch count until it hits a given max. Within that there were two different versions which differed slightly in their function, one which batch-calculated gradient descent and one which was purely stochastic (explored in **Section 2.1**). The only difference between those aforementioned versions was the use of an extra (nested) for loop which slowed down execution time considerably (and the associated indexing in the stochastic version). The update method for the weights also differed and may be seen in ***Figure 3***.

```
a = sum(a,2);
s = sum(s,2);
dW = s*a';
c = (alph/width(s))*dW;
cb = (alph/width(s))*s;

obj.W = obj.W - c;
obj.b = obj.b - cb;
```

Figure 3: Batch-update method in backprop layer

First was forward propagation through the layer.forward method, giving us each respective layer output. Next we calculated mean square error using that output and the one-hot encoded label. After that, each layer sensitivity is computed starting with the output layer and working backwards to the first hidden layer. Finally, the sensitivities are backpropagated to update each layer. The code which implements the mentioned formula can be seen in the appendix (***A.3***) in the order that it was mentioned. ***Figure 4*** illustrates how these steps were executed in the execution file.

```
while epoch<max_loops+1
    mbStart = randi([1 600000-mini_batch_size]);
    for i = mbStart:mbStart+mini_batch_size-1

        a = layerH.forward(aRaw(:,i));
        a2 = layerH2.forward(a);
        a3 = layerO.forward(a2);

        MSE(1,epoch) = MSE(1,epoch) + sum((targets(:,i) - a3)'*(targets(:,i) - a3));

        s1 = layerO.compOutSens(targets(:,i),a3);
        s2 = layerH2.compBackSens(layerO.W,s1,a2);
        layerH.compBackSens(layerH2.W,s2,a);

        if (epoch > 1)
            if (MSE (1,epoch) / MSE(1,epoch-1) > 1.02)
                lr = lr*1.0005;
                moment = momentOr;
                layerO.update(a2,moment,lr);
                layerH2.update(a,moment,lr);
                layerH.update(aRaw(:, i),moment,lr);
            elseif (MSE (1,epoch) / MSE(1,epoch-1) < .98)
                lr = lr*.9995;
                moment = 0;
            else
```

```
                layer0.update(a2,moment,lr);
                layerH2.update(a,moment,lr);
                layerH.update(aRaw(:, i),moment,lr);
            end
        end
        layerH.a0 = aRaw(:, i);
        layerH2.a0 = a;
        layer0.a0 = a2;

    end
      MSE(1,epoch) = MSE(1,epoch)/mini_batch_size;
    epoch = epoch + 1
end
```

Figure 4: Backprop algorithm, stochastic gradient descent, as seen in execution file

In order to reduce overfitting, we implemented early stopping which meant we separated 12,000 images from our original training set to use for testing each epoch so we could make sure we were not overfitting our data. This allowed us to have more insightful statistics regarding our network accuracy which we could track as we improved the network. The rest of our code consists of visualization and final output production functions which did not matter for network performance.

2.2 Results

Within this result section we will briefly explore the values which our network initially generates.We will touch on batch learning, but since it was not used for a majority of the optimizations it will not be explored further. This was because batch learning proved to be frustrating to work with and would produce very poor accuracy at times (and we ultimately could not pinpoint why). We also explored varying our hyperparameters until reaching a range of values which we felt produced reliably good accuracy.

2.2.1 Stochastic-Gradient-Descent and Mini-Batching

Both of these approaches were trained before our data set was expanded (data augmentation 3.1.3), so we trained on 48,000 images.

In ***Figure 2,*** we can see the results of our initial foray with a 196-49-10 hidden unit network with a learning rate of 0.0005 (structure shown in ***Figure 1***). This plots MSE over epoch over 1000 epochs which does eventually converge onto a low value of 1.25 (0.675). Note that MSE here was initially doubled so that it is easier to see (this was later changed but we lost this initial version so were not able to generate this graph again). The graph shows that in stochastic gradient descent we get a slow but consistent conversion.This implementation was incredibly slow because we would train on the full 48,000 images each epoch stochastically, requiring inefficient nested looping. As per Sushant Patrikar's article on stochastic and batch algorithms**,** stochastic gradient descent does small,slow steps but it has a more consistent ability to converge than batch learning(Patrikar, 2019). This is shown in the minimal jitters experienced in our MSE

and consistent, downward trend after 1000 epochs (**Figure 6**).

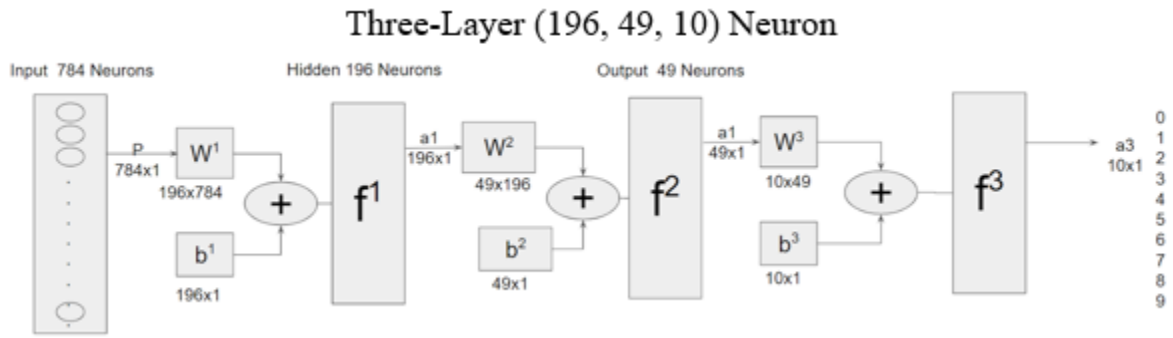


Figure 5: Three-layer (196,49,10 neuron) perceptron architecture

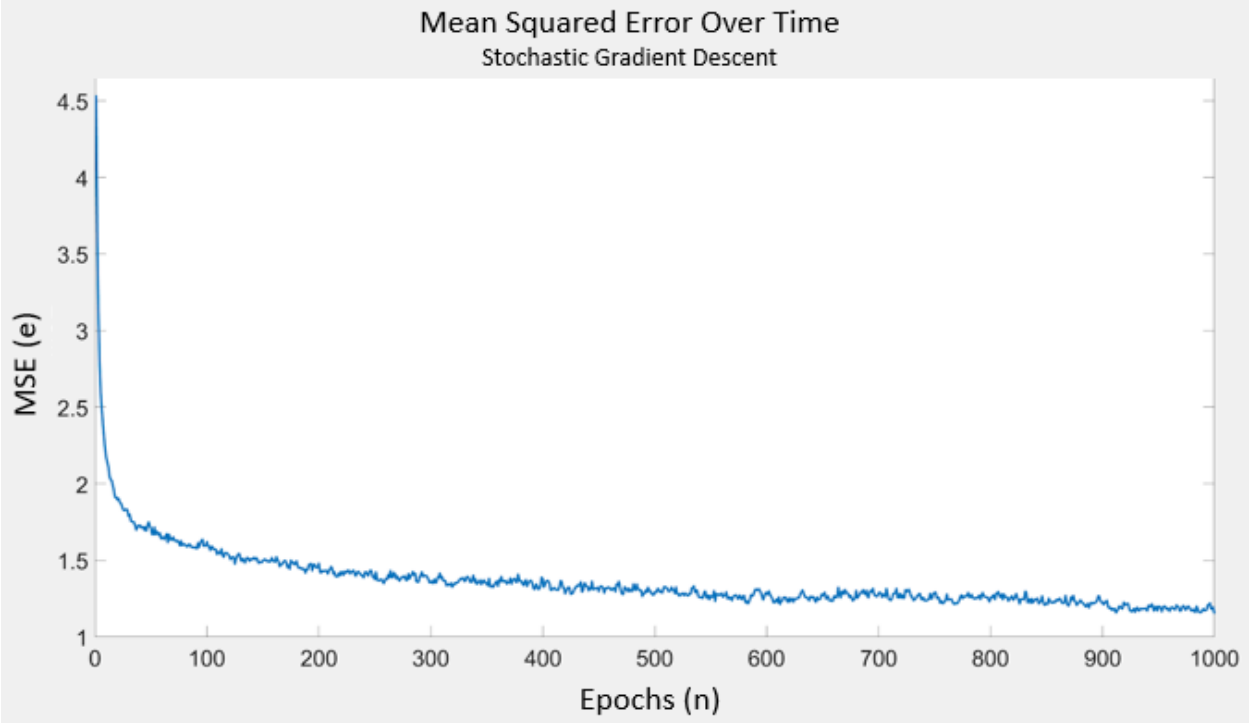


Figure 6: Mean-square-error over 1000 epochs for a stochastic-gradient descent trained 3-layer(196-49-10) network

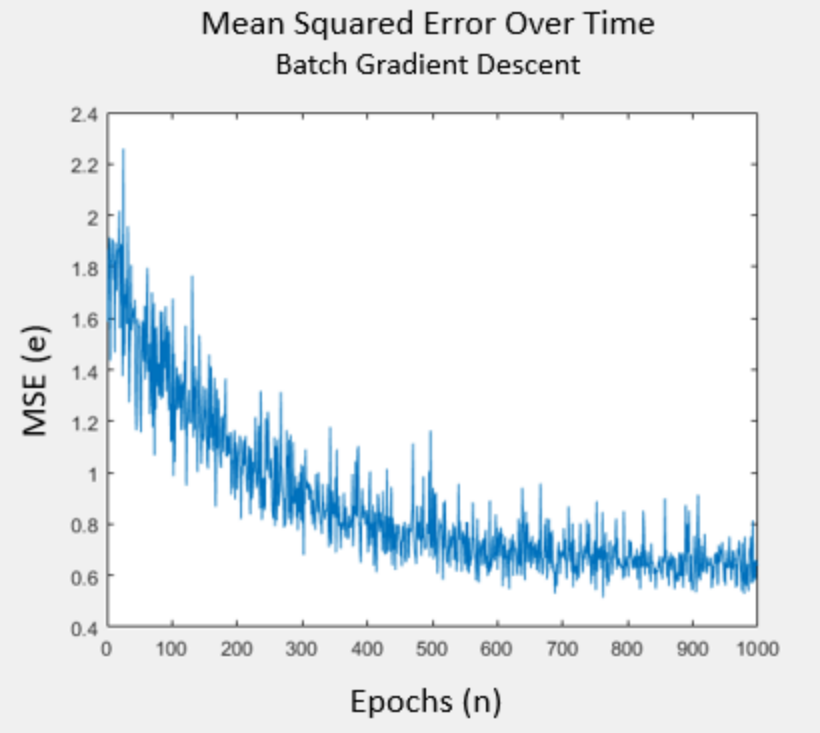


Figure 7: Mean-square-error over 1000 epochs for a batch-gradient descent 3-layer(196-49-10) network

Comparatively, our mini-batch-descent results (seen in ***Figure 7***) with the same hyperparameters (excluding mini-batch size) equaling to at best a 0.6 MSE. Mini-batch size was set to 100, which resulted in quick MSE convergence but not consistently. We theorize that our mini-batch-descent often failed to converge to a low value (with the best result in the ***Figure 7***) because it would occasionally get stuck in valleys in the parameter space that it would struggle to leave. This would result in it stopping to decrease MSE after a set number of epochs. Note mini-batch execution speed (time-wise) was a lot faster but with the other parameters we were using we did not feel fully comfortable continuing to use it.

2.2.2 Hyperparamater variance

From this point on we exclusively used hidden-unit sizes in multiples of 8. This is because neural net vector structures function most efficiently when they match the bit size of the system architecture (Anderson et al., 2019).Thus, we increased our hidden neuron count to 256-128-10, seeing a modest increase in performance but not substantial enough to justify a visualization. This hidden neuron count remained for the rest of our runs. We continued varying hyperparameters until we found decent ranges for the learning rate, momentum, and normalized weights.

Further, as can be noticed in the figures seen in the next section, we would frequently change the amount of epochs (along with the batch size we stochastically updated). Thanks to us keeping a batch-size variable, we were ultimately able to keep the amount of inputs that we trained on

about the same. Thus, while the epochs might vary, the relative iteration count remains static through each of our runs.

3. Optimization Techniques

Our optimization techniques involved using the age-old maxim of "throw everything at the wall and see what sticks". In that, some techniques were major improvements, some were minor, and some proved to be not worth the effort we put into them. We found success by using mostly heuristic optimization techniques as they were easy to implement and modify. Note, the following sections are not ordered in any specific order, though they do resemble the order in which we implemented them.

3.1 Methods

3.1.1 Data Preprocessing

When looking for optimizations, we thought it would be fitting to begin with the data set we received. One of the most basic upgrades we could do was data normalization, reducing the scale of the values in each *p* input vector (Alam, 2020) . First, we simply divided each element of the input space by 255 (maximum pixel value) to reduce them to a value between 0 and 1. This clustered the data and brought it closer to our desired one-hot encoded output. We believed we could push normalization further by centering each element to the mean and doing component-wise variance on each element (Brownlee, 2019). Doing this allowed us to better distribute our input starting values which in turn made it easier for the network to converge as it allowed us to keep our weights (and their updates) small.

3.1.2 Momentum and Variable Learning Rate

Next, we tackled heuristic optimizations as suggested by Hagan in *Neural Network Design.* Namely, we explored momentum for MSE-change minimization and a variable learning rate for faster and more accurate convergence.

Momentum allows us to introduce a scaling variable which scales a change with respect to the previous epoch's parameters. This in turn minimizes the significance of each change, smoothing out our movement in the parameter space. Its implementation was done by having a deltaW (and deltaB) function which measured the change (epoch-to-epoch) between weights. It then took those changes and applied the momentum. Implementation can be seen in ***Figure 8.***

```
function deltW = wUpdate(obj,alph)
    arguments
        obj
        alph
    end
    deltW = -1*(alph*obj.sO(:, 1)*obj.aO(:, 1)');
end
```

```
        function deltB = bUpdate(obj,alph)
            arguments
                obj
                alph
            end
            deltB = -1*(alph*obj.sO(:, 1));
        end
```

Variable learning rate allowed us to better control the change in our weights epoch-to-epoch. This in turn helps us accurately gauge the best size of each step in our parameter space when searching for minima. The implementation can be seen in ***Figure 9***.

```
if (epoch > 1)
    if (MSE (1,epoch) / MSE(1,epoch-1) > 1.02)
        lr = lr*1.005;
        moment = momentOr;
        layerO.update(a2,moment,lr);
        layerH2.update(a,moment,lr);
        layerH.update(aRaw(:, i),moment,lr);
    elseif (MSE (1,epoch) / MSE(1,epoch-1) < .98)
        lr = lr*.9995;
        moment = 0;
    else
        layerO.update(a2,moment,lr);
        layerH2.update(a,moment,lr);
        layerH.update(aRaw(:, i),moment,lr);
    end
end
```

Figure 9: Variable learning rate implementation

3.1.3 Data Augmentation

One of the most effective optimizations we incorporated into our network was data augmentation. We realized that the training set we had of 60,000 images would not be enough to reach the performance we were looking for. In order to better generalize our network for new test data, we ran all of our training data through various filters and skewed, shifted or added noise to each image in various amounts such that we ended up with a new training set that included the original training set plus an additional 540,000 tweaked images.

To accomplish this, we used a separate Matlab script which loaded in the training data and made the various modifications to the training data while preserving the labels for each modified image. Since we effectively created more unique examples for our network to train on, our network was able to better generalize and therefore was able to make predictions more accurately on new data.

Here is one example of our augmented data. On the left is the original image and on the right is our augmented image. This particular image was modified first with a slight x axis shear (so the top and bottom of the image are shifted opposite each other), as well as a gaussian noise filter to add some randomness to the image. While this is one example, we programmatically applied

varying filters and varying intensities of filtering to each image as we created our augmented training set.

Figure 10: Example of before and after image augmentation

Implementation can be seen in the code snippet featured in **Figure 11**. This was one of many augmentations performed, and the full augmentation script can be found in the appendix.

```
imAugmented = aT(:,:,i);

transformation = randomAffine2d('XShear',[-j/2,j/2]);
transOutput = affineOutputView(size(imAugmented),transformation);
imAugmented = imwarp(imAugmented,transformation,'OutputView',transOutput);
imAugmented = imAugmented .* imnoise(imAugmented, 'gaussian', 0 + ((j-5)*0.05));

montage({im2double(aT(:,:,i))./255,im2double(imAugmented)./255});
```

Figure 11: Sample code for image augmentation

3.2 Results

In this section we can see a selection of MSE over time visualizations following a given change. Note these graphs are not necessarily the best run we got with an optimization. Our starting values were consistently 0.005 for learning rate, 0.6 for momentum, 256-128-10 hidden unit count, random (normally distributed) weights between -.1 and .1., variable epoch count (~25000 iterations)

3.2.1 Data Preprocessing

After several normalization techniques, we found the component-size mean centering with standard-deviation variance (as mentioned above) to be most effective. This allowed for a large

initial jump towards the minimum as our weights. This effect can be seen in **Figure 12** at epoch 1. Note here we also have momentum enabled (with values of 0.6 for each layer) in this version (discussed in **Section 3.2.2**).

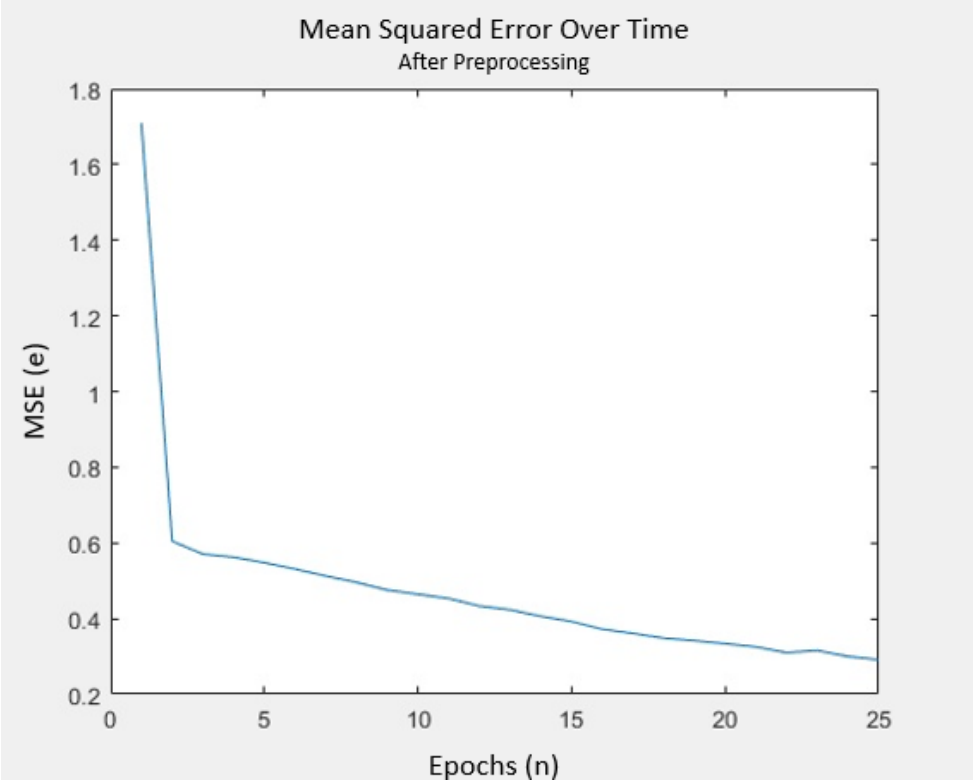


Figure 12: Mean-square-error over 1000 epochs after center to mean and component-wise scale to unit variance

3.2.2 Momentum and Variable Learning Rate

We found high momentum values to be successful in minimizing drastic increases in mean-squared error but also slowed down our convergence significantly. Instead, we opted for 0.6 momentum values (when not nullified by variable learning rate) to be successful in reducing jittering (large jumps) without hurting accuracy.

We settled on varying the learning rate by 0.05% if the MSE successfully decreases by more than 2%. Alternatively, if we saw an increase of more than 2% epoch to epoch, we would discard that weight update, nullify momentum, and scale the learning rate down by .05%. If it was between the aforementioned values, we would simply update the weights at that iteration. These values were arrived at by trial-and-error, and consulting Hagan (Hagan et al., 2014). The results after momentum and variable learning rate may be seen in ***Figure 13*** and ***Figure 14*** respectively. Notice, our learning rate (though sometimes moving MSE in the wrong direction) allowed for occasionally bigger steps to be taken. This allowed for faster and deeper network convergence.

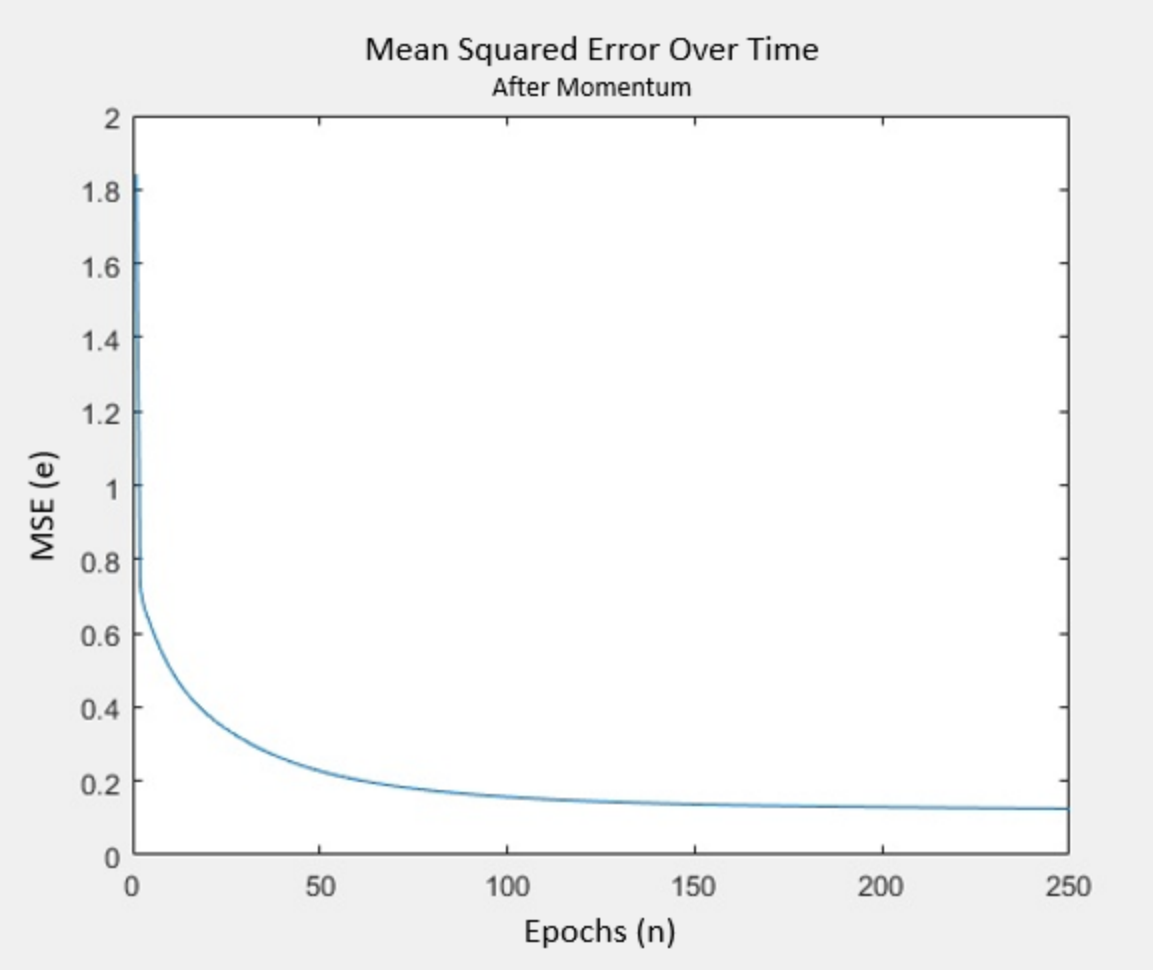


Figure 13: MSE over Epochs after high momentum values are added

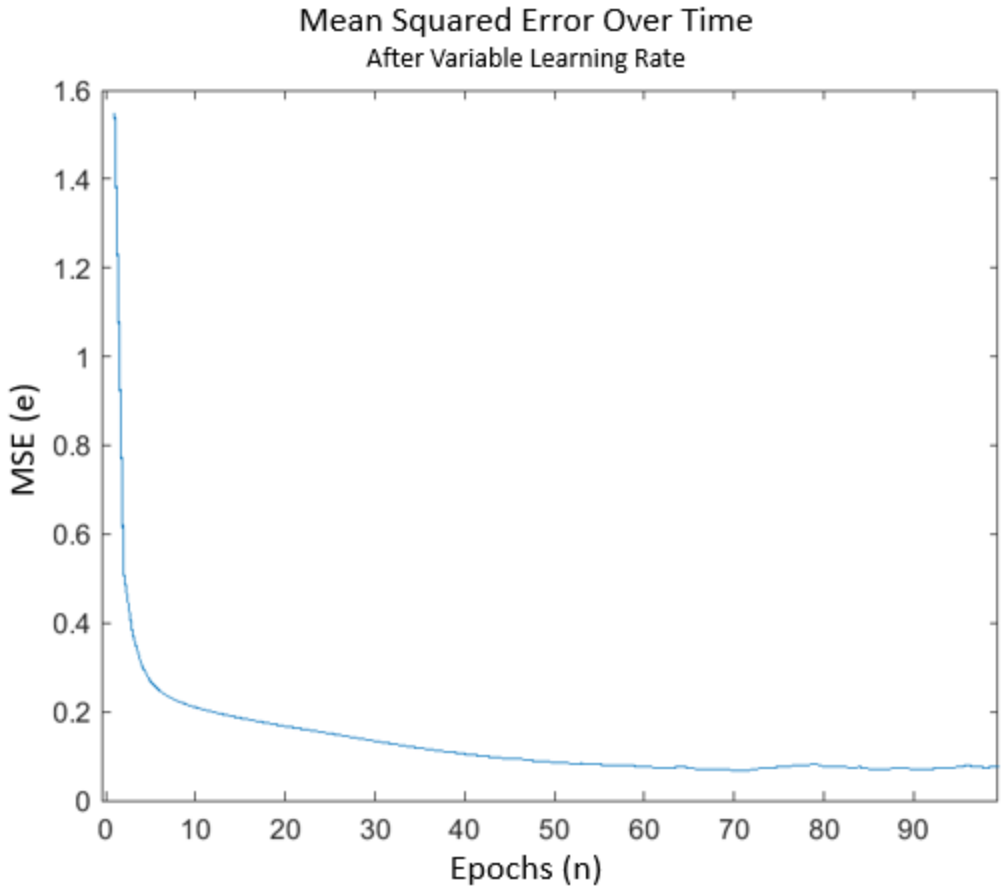


Figure 14: MSE over Epochs after VLR is added

3.2.3 Data Augmentation

Once we implemented data augmentation we saw significant improvements in our overall network performance. While our network took far longer to train than before (given the hugely increased dataset size), our error rate dropped significantly and we saw much less overfitting in our network than before when submitting on Kaggle. Our performance before data augmentation can be seen in ***Figure 2***, and our performance after can be seen in ***Figure 15*** **(**Final figure with all optimizations added**)**.

This helps in improving our network's ability to generalize as it allows the network to be more confident in identifying distinct input patterns.

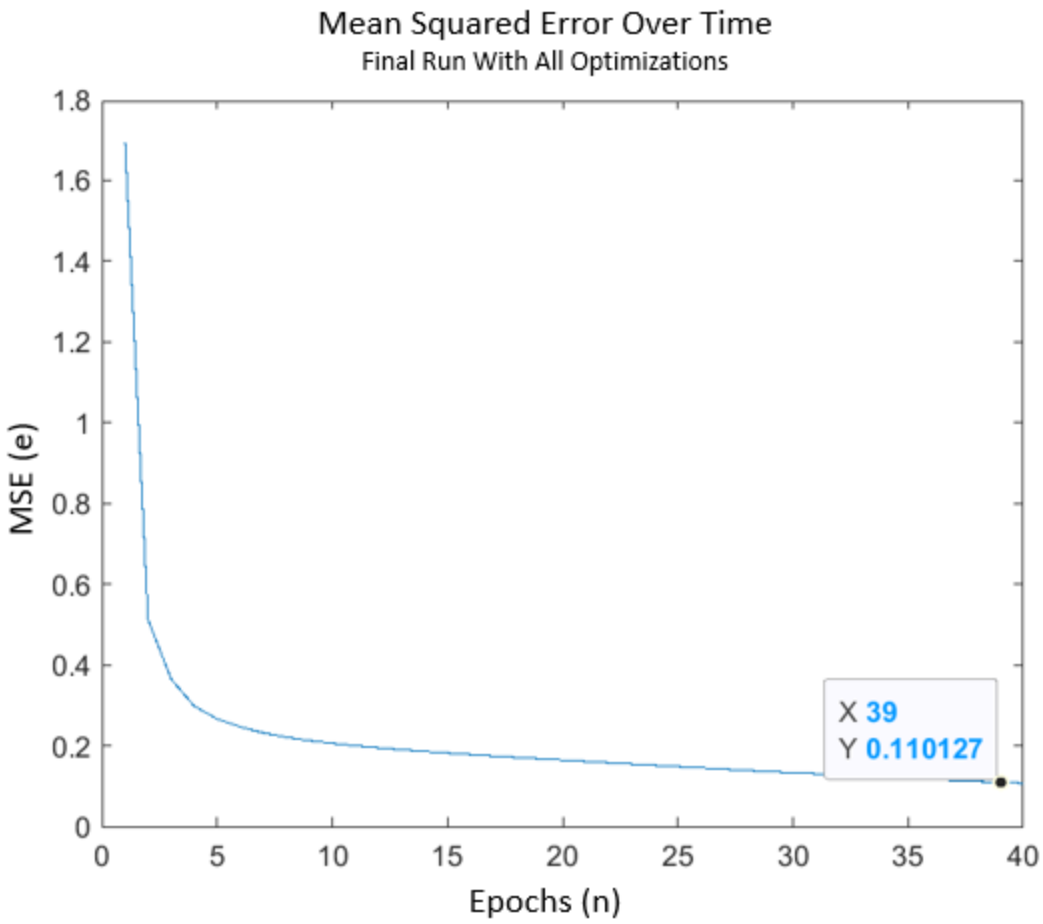


Figure 15: Final MSE chart with all optimizations added

4. Conclusion

Throughout this project we were able to take a shotgun approach at different methods for improving neural network performance and as a result we were able to combine our hands on experiences and gain a far better understanding of how neural networks function and how they can be improved. While some of the results we saw were expected, many optimizations we added surprised us with how effective or ineffective they were with improving our networks' performance.

One of the most surprising insights from this project was definitely how effective techniques like data augmentation and general data manipulation were in improving our overall network performance while still decreasing our network's overfitting by a significant amount. We were also very surprised to see that batching was incredibly finicky to work with and in our case never turned out to be worth the effort, offering minimal returns especially considering the amount of

effort it took to even get it working at all. With more time and more expertise we would likely have much more success with batching in our network, however for a shorter project such as this we found it surprisingly inaccessible and ended up using it sparingly.

Momentum and VLR were the two optimizations that worked almost exactly as we expected, and after implementing them properly we found they provided steady improvements to our performance and to how quickly and smoothly we were able to train our network.

The best results we were able to achieve were used data normalization, adding momentum, augmenting data with skewed, shifted, and filtered data (using code in Appendix A.2 - A.5). This allowed the network to converge much faster and with significantly higher accuracy. Our final network used 3 layers with (256, 128, 10) neurons, a much bigger and normalized training set, and variable learning rate with momentum (0.005 and 0.6 starting values respectively).This resulted in our lowest MSE runs (0.07-0.11 on the training set) with the lowest one whose graph visualization we outputted being 0.11.

Ultimately, this is a project whose results exceeded our expectations, allowing us to develop a nuanced understanding of backpropagation based MLP structures.

References

Alam, M. (2020, December 13). *Data normalization in machine learning*. Towards Data Science. https://towardsdatascience.com/data-normalization-in-machine-learning-395fdec69d02

Anderson, A., Doyle, M. J., & Gregg, D. (2019, December 12). Scalar Arithmetic Multiple Data:Customizable Precision for Deep Neural Networks. *School of Computer Science and Statistics*, *2*. https://arxiv.org/pdf/1809.10572.pdf

Brownlee, J. (2019, February 22). *8 Tricks for Configuring Backpropagation to Train Better Neural Networks*. Machine Learning Mastery. https://machinelearningmastery.com/best-advice-for-configuring-backpropagation-for-deep-learning-neural-networks/

Gandhi, A. (2021, March 6). *Data Augmentation | How to use Deep Learning when you have Limited Data — Part 2*. Nanonets. https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Hagan, M. T., Demuth, H. B., Beale, M. H., & Jesús, O. D. (2014). *Neural Network Design* (2nd ed.). http://hagan.okstate.edu/nnd.html

MathWorks. (2021). *Augment Images for Deep Learning Workflows Using Image Processing Toolbox*. MathWorks. https://www.mathworks.com/help/deeplearning/ug/image-augmentation-using-image-processing-toolbox.html#AugmentImagesForDeepLearningWorkflowsExample-1

Patrikar, S. (2019, September 30). *Batch, Mini Batch & Stochastic Gradient Descent*. Towards Data Science. https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a

Appendix

Appendix A.1 - Fashion Recognition Code - Yash

```
classdef FashionRecognition < handle
    properties
        Weight1 {mustBeNumeric}
        Weight2 {mustBeNumeric}
        Weight3 {mustBeNumeric}
        Weight4 {mustBeNumeric}
        Weight5 {mustBeNumeric}
        Bias1 {mustBeNumeric}
        Bias2 {mustBeNumeric}
```

```
    Bias3 {mustBeNumeric}
    Bias4 {mustBeNumeric}
    Bias5 {mustBeNumeric}
    LearningRate
    TransferType
    input
    A1
    A2
    A3
    A4
    A5
    MSE
    Threshold
    numEpoch
    exitEpoch
    momentum
end
methods
    function obj = FashionRecognition(input, output, layer1, layer2, learnrate, threshold, epochs, transtype)

        obj.Weight1 = -1 + (1+1)*rand(layer1,input);
        obj.Weight2 = -1 + (1+1)*rand(layer2,layer1);
        obj.Weight3 = -1 + (1+1)*rand(output,layer2);

        obj.Bias1 = -1 + (1+1)*rand(layer1,1);
        obj.Bias2 = -1 + (1+1)*rand(layer2,1);
        obj.Bias3 = -1 + (1+1)*rand(output,1);

        obj.TransferType = transtype;
        obj.LearningRate = learnrate;
        obj.Threshold = threshold;
        obj.numEpoch = epochs;
        obj.MSE = zeros(obj.numEpoch, 1);
        obj.momentum = 0.98;
    end

    function reinitalizing(obj, weights1, weights2, weights3, bias1, bias2, bias3, learnrate, threshold, epochs, transtype)

        obj.Weight1 = weights1;
        obj.Bias1 = bias1;
        obj.Weight2 = weights2;
       obj.Bias2 = bias2;
        obj.Weight3 = weights3;
        obj.Bias3 = bias3;
        obj.TransferType = transtype;
        obj.LearningRate = learnrate;
        obj.Threshold = threshold;
        obj.numEpoch = epochs;
        obj.MSE = zeros(obj.numEpoch, 1);
    end

    function outputArg1 = forward(obj, x)

        obj.input = x;
        z1 = obj.Weight1 * x.';
        z1 = z1 + obj.Bias1;
        a1 = obj.transferfunc(z1);
        obj.A1 = a1;
        z2 = obj.Weight2 * a1;
        z2 = z2 + obj.Bias2;
        a2 = obj.transferfunc(z2);
```

```
        obj.A2 = a2;
        z3 = obj.Weight3 * a2;
        z3 = z3 + obj.Bias3;
        a3 = obj.transferfunc(z3);
        obj.A3 = a3;

        outputArg1 = obj.A3;
    end

    function backward(obj, err)
        der = obj.derivative(obj.A1);
        der2 = obj.derivative(obj.A2);
        der3 = obj.derivative(obj.A3);

        s3 = -2 * der3 * err;
        s2 = (der2 * obj.Weight3.') * s3;
        s1 = (der * obj.Weight2.') * s2;
        inp = obj.input;

        obj.Weight3 = obj.Weight3 - (obj.LearningRate *(s3 * obj.A2.'));
        obj.Weight2 = obj.Weight2 - (obj.LearningRate *(s2 * obj.A1.'));
        obj.Weight1 = obj.Weight1 - (obj.LearningRate * (s1 * inp));
        obj.Bias3 = obj.Bias3 - obj.LearningRate * s3;
        obj.Bias2 = obj.Bias2 - obj.LearningRate * s2;
        obj.Bias1 = obj.Bias1 - obj.LearningRate * s1;
    end

    function err = errorLoss(obj, t, a)
        err = t.' - a;
    End
    function scoring(obj, p, t, epoch)
        obj.MSE(epoch,1) = 0;
        for q=1:size(p,1)
            a = obj.forward(p(q,:));
            e = obj.errorLoss(t(q, :),a);
            obj.MSE(epoch,1) = obj.MSE(epoch,1) + (e.' * e);
        end
    end
    function images = loadMNISTImages(obj, filename)
        fp = fopen(filename, 'rb');
        assert(fp ~= -1, ['Could not open ', filename, '']);

        magic = fread(fp, 1, 'int32', 0, 'ieee-be');
        assert(magic == 2051, ['Bad magic number in ', filename, '']);

        numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
        numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
        numCols = fread(fp, 1, 'int32', 0, 'ieee-be');

        images = fread(fp, inf, 'unsigned char');
        images = reshape(images, numCols, numRows, numImages);
        images = permute(images,[2 1 3]);

        fclose(fp);


        images = reshape(images, size(images, 1) * size(images, 2), size(images, 3));

        images = double(images) / 255;

    end
```

```
function labels = loadMNISTLabels(obj, filename)
    fp = fopen(filename, 'rb');
    assert(fp ~= -1, ['Could not open ', filename, '']);

    magic = fread(fp, 1, 'int32', 0, 'ieee-be');
    assert(magic == 2049, ['Bad magic number in ', filename, '']);

    numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');

    labels = fread(fp, inf, 'unsigned char');

    assert(size(labels,1) == numLabels, 'Mismatch in label count');

    fclose(fp);

end
function T = onehotencoding(obj, op)
    [tsize, ~] = size(op);
    T = zeros(tsize, 10);
    for i = 1:tsize
        T(i, op(i)+1) = 1;
    end
end

function training(obj, p, t)
    [pTrainRow, pTrainCol] = size(p);
    fprintf("***Traning Model with
    obj.MSE = zeros(obj.numEpoch, 1);
    for epoch=1:obj.numEpoch
        fprintf("\n ***Start of Epoch
        for q=1:pTrainRow
            a = obj.forward(p(q,:));
            e = obj.errorLoss(t(q, :),a);
            obj.backward(e);
        end
        if obj.checkDone(p,t, epoch)
            obj.exitEpoch = epoch;
            break
        elseif epoch == obj.numEpoch
            obj.exitEpoch = epoch;
        end
        fprintf("\n ***End of Epoch %d, MSE %6.4f ***\n", epoch, obj.MSE(epoch));
    end
end

function r = remodify(obj, inputs)
    [inputrow, inputcol] = size(inputs);
    r = [];
    for i = 1:inputrow
        temp = inputs(i, :);
        tempshape = reshape(temp, [28,28]);
        temptranpose = tempshape.';
        tempunshape = reshape(temptranpose, [1, 784]);
        r = [r;tempunshape];
    end
end

function acc = accuracy(obj, x, T)
    [Trow, Tcol] = size(T);
```

```
        counter = 0;
        for i = 1:Trow
            a = obj.forward(x(i, :));
            a = a.';
            [~, pos] = max(a(1, :));
            aZeroRow = zeros(1, Tcol);
            aZeroRow(1, pos) = 1;
            ifZero = T(i, :) - aZeroRow(1, :);
            if ifZero == 0
                counter = counter + 1;
            end
        end
        acc = counter/Trow;
    end

    function flag = checkDone(obj, p, t, epoch)
        flag = true;
        obj.scoring(p, t, epoch);
        if(obj.MSE(epoch) > obj.Threshold)
            flag = false;
        end
    end
    function graph(obj)

        x = 1:obj.exitEpoch;
        plot(x,obj.MSE(1:obj.exitEpoch));
        title('MSE versus training epoch');
        xlabel('number of epoch');
        ylabel('MSE');
    end

    function print(obj)
        fprintf("Weight1:\n")
        disp(obj.Weight1);
        fprintf("Bias1:\n")
        disp(obj.Bias1);
        fprintf("Weight2:\n")
        disp(obj.Weight2);
        fprintf("Bias2:\n")
        disp(obj.Bias2);
    end

end

methods (Access = private)
    function outputArg = transferfunc(obj, z)
        if strcmp(obj.TransferType, 'hardlim')
            a = hardlim(z);
        elseif strcmp(obj.TransferType, 'hardlims')
            a = hardlims(z);
        elseif strcmp(obj.TransferType, 'purelin')
            a = purelin(z);
        elseif strcmp(obj.TransferType, 'logsig')
            a = logsig(z);
        else
            a = 0;
       end
        outputArg = a;
    end
    function printSizeMat(obj, m, st)
        fprintf("%s:",st);
```

```
        disp(size(m));
    end

    function retderivative = derivative(obj, a)
        i = eye(length(a));
        if strcmp(obj.TransferType, 'hardlim')
            derv = 0;
        elseif strcmp(obj.TransferType, 'hardlims')
            derv = 0;
        elseif strcmp(obj.TransferType, 'purelin')
            derv = 1;
        elseif strcmp(obj.TransferType, 'logsig')
            derv = (1-a).*a;
        end
        retderivative = i .* derv;
    end
  end
end
```

Appendix A.2 - Fashion Recognition Layer Batch Gradient Descent- Guiragos BP4

```
function BP4
Test = readtable('train.csv', 'HeaderLines', 1);
aRawSheet = table2array(Test);
tRaw = aRawSheet (1:end, 1);
aRaw = aRawSheet(1:end, 3:end);
aRaw = aRaw';
aRaw = (aRaw-mean(aRaw, 1))./std(aRaw);

Test = readtable('test.csv', 'HeaderLines', 1);
tSheet = table2array(Test);
aTar = tSheet(1:end, 2:end);
aTar = aTar';

aTar = (aTar-mean(aTar, 1))./std(aTar);
targets = zeros(10,height (tRaw))+0.1;
for i = 1 : height (tRaw)
   targets(tRaw(i,:)+1,i) = .9;
end
layerH = BackpropLayer2(784,128,"logsig","hidden_layer");
layerO = BackpropLayer2(128,10,"logsig","output_layer");
mini_batch_size = 100;
max_loops = 1000;
epoch = 1;
lR = 0.001;
MSE = zeros (1,max_loops);
MSE_EarlyStop = zeros (max_loops);
resultsT = zeros(max_loops);
resultsV = zeros(max_loops);
stopWhen = 0;
while epoch<max_loops+1
mbStart = epoch;
   a = layerH.forward(aRaw(:,mbStart:mbStart+mini_batch_size-1));
   a2 = layerO.forward(a);
   sOut = layerO.compOutSens(targets(:,mbStart:mbStart+mini_batch_size-1),a2);
```

```
    sHid = layerH.compBackSens(layerO.W,sOut,a);
    for i = 1:mini_batch_size
        MSE(1,epoch) = MSE(1,epoch) + sum((targets(:,i) - a2(:,i))'*(targets(:,i) - a2(:,i)));
    end
       MSE(1,epoch) = MSE(1,epoch)/mini_batch_size;
    layerO.update(a,sOut,lR);
    layerH.update(aRaw(:,mbStart:mbStart+mini_batch_size-1),sHid,lR);
    epoch = epoch + 1
end
testindex = tSheet(1:end, 1);
temp = zeros(10000,2);
temp (:,1) = testindex;
for i = 1:10000
    a = layerH.forward(aRaw(:,i));
    a2 = layerO.forward(a);
    temp(i,2) = find(max(a2) == a2)-1;
end
tabout = array2table(temp);
writetable(tabout, 'output.csv');
end
```

Appendix A.3 - Fashion Recognition Layer Stochastic Gradient Descent-Guiragos BP6

```
function BP6
Test = readtable('train.csv', 'HeaderLines', 1);
aRawSheet = table2array(Test);
tRaw = aRawSheet (1:end, 2);
aRaw = aRawSheet(1:end, 3:end);
aRaw = aRaw';
aRaw = (aRaw-mean(aRaw, 1))./std(aRaw);
Test = readtable('test.csv', 'HeaderLines', 1);
tSheet = table2array(Test);
aT = tSheet(1:end, 2:end);
aT = aT';
aT = (aT-mean(aT, 1))./std(aT);
targets = zeros(10,height (tRaw))+0.1;
for i = 1 : height (tRaw)
    targets(tRaw(i,:)+1,i) = .9;
end
layerH = BackpropLayer3(784,256,"logsig","hidden_layer");
layerH2 = BackpropLayer3(256,128,"logsig","output_layer");
layerO = BackpropLayer3(128,10,"logsig","output_layer");
mini_batch_size = 10000;
max_loops = 10;
epoch = 1;
lr = 0.01;
MSE = zeros (1,max_loops);
MSE_EarlyStop = zeros (1,max_loops);
resultsV = zeros(1,max_loops);
stopWhen = 0;
moment = .6;
momentOr = moment;
while epoch<max_loops+1

    mbStart = randi([1 600000-mini_batch_size]);
    for i = mbStart:mbStart+mini_batch_size-1
```

```
        a = layerH.forward(aRaw(:,i));
        a2 = layerH2.forward(a);
        a3 = layerO.forward(a2);

        MSE(1,epoch) = MSE(1,epoch) + sum((targets(:,i) - a3)'*(targets(:,i) - a3));

        s1 = layerO.compOutSens(targets(:,i),a3);
        s2 = layerH2.compBackSens(layerO.W,s1,a2);
        layerH.compBackSens(layerH2.W,s2,a);

        if (epoch > 1)
            if (MSE (1,epoch) / MSE(1,epoch-1) > 1.02)
                lr = lr*1.000005;
                moment = momentOr;
                layerO.update(a2,moment,lr);
                layerH2.update(a,moment,lr);
               layerH.update(aRaw(:, i),moment,lr);
            elseif (MSE (1,epoch) / MSE(1,epoch-1) < .98)
                lr = lr*.999995;
                moment = 0;
            else
                layerO.update(a2,moment,lr);
                layerH2.update(a,moment,lr);
                layerH.update(aRaw(:, i),moment,lr);
            end

        end
        layerH.aO = aRaw(:, i);
        layerH2.aO = a;
        layerO.aO = a2;

    end
      MSE(1,epoch) = MSE(1,epoch)/mini_batch_size;
    results = 0;
    for i = 50000:60000
        a = layerH.forward(aRaw(:,i));
        a2 = layerO.forward(a);
        MSE_EarlyStop(1,epoch) = MSE_EarlyStop(1,epoch) + sum((targets(:,i) -      a2)'*(targets(:,i) - a2));
    end
    if epoch > 1
        if (MSE_EarlyStop (1,epoch) - MSE_EarlyStop(1,epoch-1) > 0)
            stopWhen = stopWhen + 1;
        end
    end
    if (stopWhen == 15)
        break;
    end
    epoch = epoch + 1
end
figure
plot(1:epoch-1,MSE(1,1:epoch-1));
drawnow;
testindex = tSheet(1:end, 1);
temp = zeros(10000,2);
temp (:,1) = testindex;
for i = 1:10000
      a = layerH.forward(aT(:,i));
      a2 = layerH2.forward(a);
      a3 = layerO.forward(a2);
    temp(i,2) = find(max(a3) == a3)-1;
end
```

```
tabout = array2table(temp);
writetable(tabout, 'output.csv');

end
```

Appendix A.4 - Fashion Recognition Exec. Stochastic Gradient Descent-Guiragos BackpropLayer3

```
classdef BackpropLayer3 < handle
    properties
        W (:,:)
        b (:,:)
        s (:,:)
        sO (:,:)
        aO (:,:)
        f
        k
        layer_name string
    end
    properties (Constant=true)
        class_name = "BackpropLayer3";
    end
    methods (Access = private)
        function [w,b] = validate_weights_bias(~,weights,bias)

            if all(size(bias)==1)

                if all(size(weights)==1)
                    weights = normrnd(0, 0.1, bias, weights);
                    bias = normrnd(0, 0.1 , bias, 1);
                end
            elseif all(size(weights)==1)
                weights = rand(height(bias),weights)*.1-.1;
            end
            w = weights;
            b = bias;
        end
    end
    methods
        function obj = BackpropLayer3(weights, bias, transfer_func_handle, layer_name_optional)
            arguments
                weights (:,:){mustBeNumeric}
                bias (:,1){mustBeNumeric}
                transfer_func_handle string = "purelin";
                layer_name_optional string = "";
            end
            if isstring(transfer_func_handle)
                transfer_func_name = transfer_func_handle;
                transfer_func_handle = str_to_transfer_func(transfer_func_handle);
            else
                transfer_func_name = func2str(transfer_func_handle);
            end
            [obj.W,obj.b] = obj.validate_weights_bias(weights,bias);
```

```
    obj.f = transfer_func_handle;
    if ~strcmp(layer_name_optional,"") && ~contains(layer_name_optional,"__")
        layer_name_optional = "__"+layer_name_optional+"__";
    end


    obj.layer_name = obj.class_name+layer_name_optional+transfer_func_name;
    S = height (bias);
    obj.k = 0;
end
function a = forward(obj,p)
    arguments
        obj
        p (:,:)
    end
    if height(p)~=width(obj.W)
        if height(p)~=height(obj.b)
            err_msg = "\tDimension missmatch:\n\tobj.b has shape: %dx%d\n\tinput p has shape: %dx%d";
            error(err_msg,height(obj.b),width(obj.b),height(p),width(p));
        end
    end
    p = conv2(p, (1/16)*[1 2 1; 2 4 2; 1 2 1], 'same');
    n = obj.W*p+obj.b;
    a = obj.f(n);

end

function sens = compOutSens (obj,t,a)
    arguments
        obj
        t (:,:)
        a (:,:)
    end
    obj.sO = obj.s;
    dF = a.*(1-a);
    c = t-a;
    sens = -2*(dF.*c);
    obj.s = sens;
end

function sens = compBackSens (obj,W,s,a)
    arguments
        obj
        W (:,:)
        s (:,:)
        a (:,:)
    end
    obj.sO = obj.s;
    dF = a.*(1-a);
    sens  = dF.*(W'*s);
    obj.s = sens;
end
function deltW = wUpdate(obj,alph)
  arguments
        obj
        alph
    end
    deltW = -1*(alph*obj.sO(:, 1)*obj.aO(:, 1)');
end
function deltB = bUpdate(obj,alph)
```

```
        arguments
            obj
            alph
        end
        deltB = -1*(alph*obj.sO(:, 1));
    end
    function update(obj,a,gamm,alph)

        arguments
            obj
            a (:,:)

            gamm
            alph
        end

        c = gamm*obj.wUpdate(alph)-((1-gamm)*alph*obj.s(:, 1)*a(:, 1)');

        cb = gamm*obj.bUpdate(alph)-alph*obj.s(:, 1);
        obj.W = obj.W + c;
        obj.b = obj.b + cb;
    end
    function print(obj,reshaped)

        arguments
            obj
            reshaped = [];
        end

        if any(size(reshaped)>0)
            node_h = reshaped(1);
            node_w = reshaped(2);
        else
            node_h = 1;
            node_w = width(obj.W);
        end
        disp("Weights: ")
        for i = 1:height(obj.W)
            fprintf("node %d\n",i-1);
            w = reshape(obj.W(i,:),node_h,node_w);
            disp(w)
        end
        disp("Bias: ")
       disp(obj.b)
    end
  end
end

function fptr = str_to_transfer_func(transfer_func_name)
switch lower(transfer_func_name)
  case "hardlim"
    fptr = @hardlim;
  case "hardlims"
    fptr = @hardlims;
  case "purelin"
    fptr = @purelin;
  case "satlin"
    fptr = @satlin;
  case "satlins"
    fptr = @satlins;
  case "logsig"
```

```
        fptr = @logsig;
    case "tansig"
        fptr = @tansig;
    case "poslin"
        fptr = @poslin;
    case "test"
        fptr = @test;
    case "test2"
        fptr = @test2;
    otherwise
        msg = "The provided function name of: %s; was not recognized, defaulting back to the 'purelin' function";
        warning(msg,transfer_func_name);
        fptr = @purelin;
end
end

function result = hardlim(scalar_val)

result = ones([size(scalar_val)]);
result(scalar_val<0) = 0;
end

function result = hardlims(scalar_val)

result = ones(size(scalar_val));
result(scalar_val<0) = -1;
end

function result = purelin(scalar_val)

result = scalar_val;
end
function result = satlin(scalar_val)

result = scalar_val;
result(scalar_val>1) = 1;
result(scalar_val<0) = 0;
end

function result = satlins(scalar_val)

result = scalar_val;
result(scalar_val<-1) = -1;
result(scalar_val>1) = 1;
end

function result = logsig(scalar_val)

result = exp(-scalar_val);
result = result + 1;
result = result.^-1;
end

function result = tansig(n)

e_to_n = exp(n);
e_to_n_neg = exp(-n);
denom = e_to_n;
denom = denom + e_to_n_neg;
denom = 1 / denom;
result = e_to_n;
```

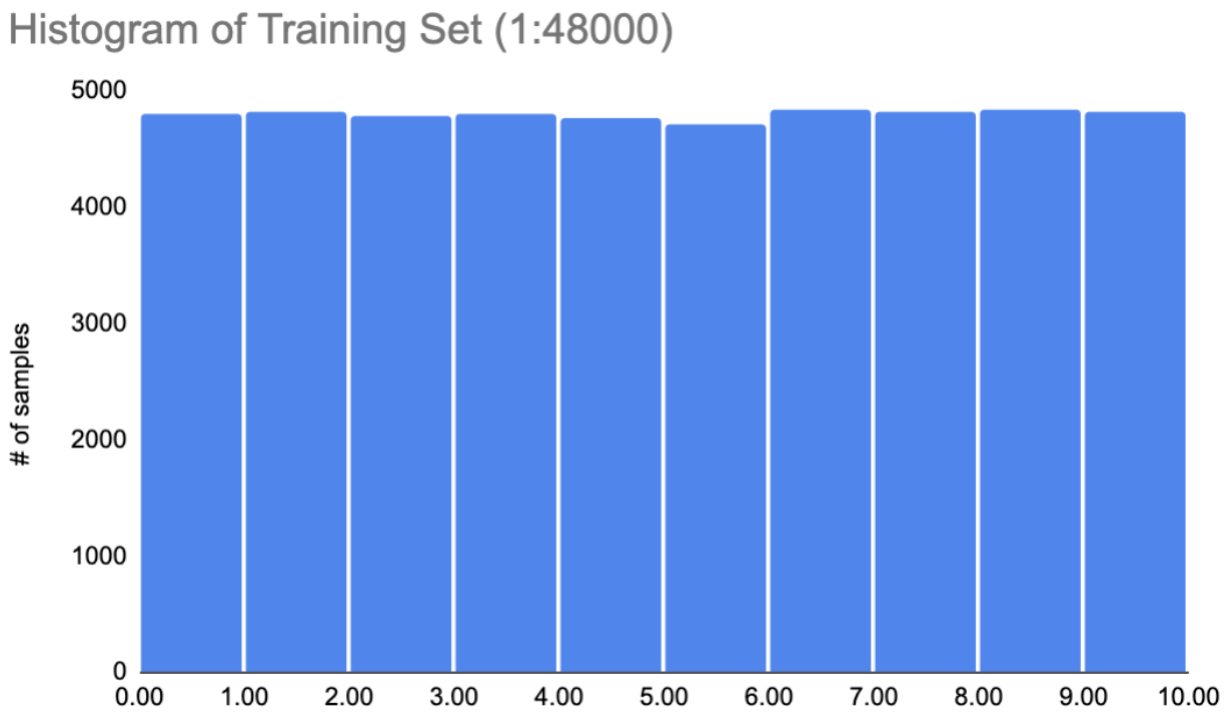
```
result = result - e_to_n_neg;
result = result * denom;
end

function result = poslin(scalar_val)

result = scalar_val;
result(scalar_val<0) = 0;
end
function result = test (scalar_val)
result = scalar_val;
result = result.^3;
end
function result = test2 (scalar_val)
result = scalar_val;
result = result.^2;
result = result.^-1;
end
```

Appendix A.5 - Data Augmentation Script Dylan

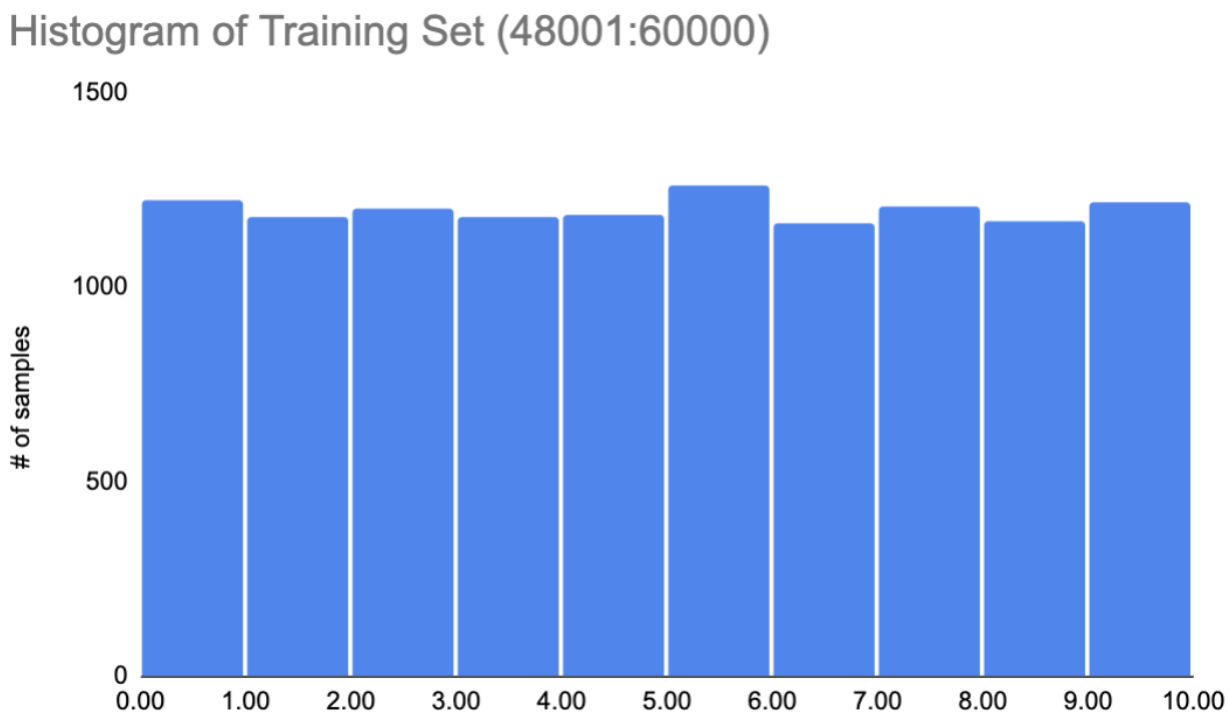
```
imgData = readtable('train.csv', 'HeaderLines', 1);
augments = 10;
rows = 60000;
imgOut = zeros(rows + (rows*augments),786);
aRawSheet = table2array(imgData);
labels = aRawSheet (1:end, 2);
images = aRawSheet(1:end, 3:end)';
aT = reshape(images,28,28,rows);

for i = 1:rows
    aT(:,:,i) =  aT(:,:,i)';
    imgOut(i,:) = aRawSheet(i,:);
end

for i = 1:rows
    for j = 1:augments
        switch j
            case 1
                imAugmented = aT(:,:,i);
                transformation = randomAffine2d('XShear',[-10,10]);
                transOutput = affineOutputView(size(imAugmented),transformation);
                imAugmented = imwarp(imAugmented,transformation,'OutputView',transOutput);

                currentRow = rows + (i*(j));
                reshaped = reshape(imAugmented.',1,784);
                imgOut(currentRow,3:end) = reshaped;
                imgOut(currentRow, 2) = imgOut(i, 2);
            case 2
                imAugmented = aT(:,:,i);
                imAugmented = imAugmented .* imnoise(imAugmented, 'gaussian');

                currentRow = rows + (i*(j));
                reshaped = reshape(imAugmented.',1,784);
                imgOut(currentRow,3:end) = reshaped;
                imgOut(currentRow, 2) = imgOut(i, 2);
            case 3
                imAugmented = aT(:,:,i);
                transformation = randomAffine2d('XReflection',true,'YReflection',false);
```

```
            transOutput = affineOutputView(size(imAugmented),transformation);
            imAugmented = imwarp(imAugmented,transformation,'OutputView',transOutput);

            currentRow = rows + (i*(j));
            reshaped = reshape(imAugmented.',1,784);
            imgOut(currentRow,3:end) = reshaped;
            imgOut(currentRow, 2) = imgOut(i, 2);
        case 4
            imAugmented = aT(:,:,i);
            imAugmented = imAugmented.*(1-0.2*rand);

            currentRow = rows + (i*(j));
            reshaped = reshape(imAugmented.',1,784);
            imgOut(currentRow,3:end) = reshaped;
            imgOut(currentRow, 2) = imgOut(i, 2);
         case 5
            imAugmented = aT(:,:,i);
            imAugmented = imAugmented + (0.5-rand);

            currentRow = rows + (i*(j));
            reshaped = reshape(imAugmented.',1,784);
            imgOut(currentRow,3:end) = reshaped;
            imgOut(currentRow, 2) = imgOut(i, 2);
         otherwise
            imAugmented = aT(:,:,i);
            transformation = randomAffine2d('XShear',[-j/2,j/2]);
            transOutput = affineOutputView(size(imAugmented),transformation);
            imAugmented = imwarp(imAugmented,transformation,'OutputView',transOutput);
            imAugmented = imAugmented .* imnoise(imAugmented, 'gaussian', 0 + ((j-5)*0.05));

            montage({im2double(aT(:,:,i))./255,im2double(imAugmented)./255});

            currentRow = rows + (i*(j));
            reshaped = reshape(imAugmented.',1,784);
            imgOut(currentRow,3:end) = reshaped;
            imgOut(currentRow, 2) = imgOut(i, 2);
      end
   end
end
writematrix(imgOut, "AugmentedData.csv");
```

Appendix A.6 - Histogram of the initial training dataset from 1 to 48000



Appendix A.7 - Histogram of the initial testing dataset from 48001 to 60000



Appendix A.8 - The weights after lowest MSE run (0.11)