Fashion-MNIST Final Classification Project

DONGHEE LEE, VICTOR SUCIU, and LINH HA

1 INTRODUCTION

The following report explores the usage of artificial neural networks to classify Fashion-MNIST. Fashion-MNIST is a data set of 60,000 grey-scaled clothing images. For our first set of experiments, we embarked on a grand journey of trial-and-error and tested various configurations of a multilayered perceptron that utilized backpropagation. We modified hyper-parameters, changed the number of layers and neurons, and adjusted the transfer functions. Next, we augmented the training data by adding multiple slightly altered copies of each image. Finally, we created an Ensemble model that aggregated multiple trained MLPs into a single, more intelligent model. We tracked the accuracy of each model and produced a csv submission file for the *UWB CSS 485: Winter 2021 Project* Kaggle competition.

2 EXPERIMENTS - METHODS

First, we started training a wide variety of MLPs in an attempt to increase classification accuracy. We experimented with the number of layers, neuron counts, and hyperparameters for gradient descent. After we achieved reasonably high performance, we implemented some data augmentation techniques to increase the size of the training set and experimented further with MLP configurations. *Appendix 7.1 Multilayer Perceptron Class* and *Appendix 7.2 Perceptron Layer Class* are the classes used to build our network. *Appendix 7.6 Transfer Functions and their Derivatives* are the different functions we used for our network.

2.1 Preprocessing, Training, and Prediction

Before training, we performed some minor preprocessing to the data. First, the numeric labels were converted to one-hot format using *Appendix 7.11 One-Hot Encoding and Decoding* and the images were reshaped into 784-dimensional column vectors. Next, each image was normalized by dividing each element by 255 which reduced their range to $[0, 1]$. Then, we divided our data set into a training set with 50,000 data points and a validation set of 10,000 data points.

Training was done with a batch size ranging from 24 to 64 and a variety of epoch size. For epochs, we did not use a fixed number automatic early stopping. Instead, MultilayerPerceptron's training function checked the validation accuracy after every epoch, and saved a checkpoint if the accuracy was a new maximum. Each checkpoint was an entire MultilayerPerceptron object that was serialized as a file. This meant that any checkpoint could be loaded back into MatLab for prediction or further training. This strategy allowed us to continue training a model well into overfitting territory while leaving the best version untarnished. *Appendix 7.12 Main Function - Standard* demonstrates the driver code that builds and trains the model.

Once a model was trained, we loaded in its best checkpoint and used it to predict the submission data. Final prediction was done by passing data through the model's forward function and applying the hardmax function to the output. Hardmax, also known as argmax, takes a vector as input and sets maximum element to 1 and the rest to 0. This results in the same one-hot format as the labels where the maximum vector element is the model's prediction. *Appendix 7.8 Hardmax Function* shows the implementation of hardmax.

Authors' address: Donghee Lee; Victor Suciu; Linh Ha.

Table 1. All Unique MLP Configurations with Best Accuracy Per-Configuration

Layer 1		**Layer 2**		**Layer 3**		**Layer 4**		**Params**			
Trans	Neur	Trans	Neur	Trans	Neur	Trans	Neur	Lr	Mtm	Dcy	**Acc %**
50,000 Train — 10,000 Validation											
tanh	256	tanh	128	tanh	48	sigmoid	10	0.1	0	0	68.00
linear	256	linear	128	linear	48	sigmoid	10	0.05	0	0	84.00
sigmoid	400	sigmoid	250	sigmoid	100	relu	10	0.1	0	0	86.86
sigmoid	400	sigmoid	250	sigmoid	100	softmax	10	0.07	0.8	0	88.40
sigmoid	400	sigmoid	250	sigmoid	100	relu	10	0.08	0.5	0	88.40
sigmoid	400	sigmoid	250	sigmoid	100	relu	10	0.05	0.8	0	89.10
sigmoid	400	sigmoid	250	sigmoid	100	relu	10	0.03	1	0	89.24
250,000 Augmented Train — 10,000 Validation											
sigmoid	400	sigmoid	250	sigmoid	100	relu	10	0.03	1	0	89.40
sigmoid	400	sigmoid	250	sigmoid	100	relu	10	0.07	1	0	89.63
tanh	400	sigmoid	250	sigmoid	100	softmax	10	0.05	0.9	0	89.26
sigmoid	400	sigmoid	150	softmax	10	—	—	0.08	1	0	90.14
sigmoid	400	sigmoid	150	softmax	10	—	—	0.08	1	-0.01	90.40
300,000 Augmented Train — 10,000 Validation											
sigmoid	500	sigmoid	250	relu	10	—	—	0.08	1	-0.01	89.92

Every unique multilayer perceptron configuration we tried and the best validation accuracy for each one. Many of these were trained multiple times to account for random weight initialization, but only the best accuracy is shown for each configuration.
Key:
Trans - Transfer Function   Neur - Number of Neurons   Lr - Learning Rate   Mtm - Momentum   Dcy - Weight Decay
Acc - Validation Accuracy

2.2 Overview of Trials

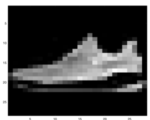
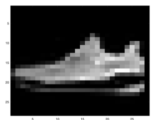
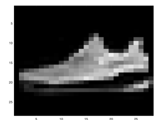
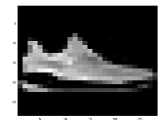
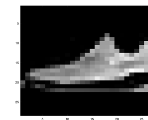
There are far too many combinations of layers, neurons, transfer functions, and training parameters to search for an optimal configuration methodically. Because of the time limitation, we decided to try a few reasonable combinations to find the one that works the best. We constrained the architectures in a few general ways. First, we limited the depth of our networks to 3-4 layers. This is slightly deeper than our best MNIST architecture, and a harder task like Fashion-MNIST would warrant more parameters in total. Second, each layer in the network has fewer neurons than the previous layer. The motivation for this funnel-like architecture is to help the layers learn incrementally more general features as they decrease in size. Most of our experimentation was with the transfer functions and the hyperparameters of learning rate, momentum, and weight decay. Cross-entropy loss was used when the output layer had softmax while MSE loss was used in all other cases. *Appendix 7.7 Loss Functions and their Derivatives* contains aforementioned functions.

2.3 Data Augmentation

The increase in performance of our model became very minimal over time. Rather than exploring further with different parameters with the same training data set, we decided to perform augmentation on the data set. Our model could benefit from having more data because it prevents over-fitting and would cover more examples from the entire distribution.

Our training data was augmented in 4 different ways as shown in Table 2. Each sample was rotated clock-wise and counter clock-wise by 2 degrees, flipped horizontally, and distorted with a set of x and y scale. Applying these augmentation on 50,000 samples, we gained 200,000 more training data.

Table 2. Example of Data Augmentations

Original	2° Clockwise	2° Counterclock.	Flipped	Distorted
				

An original image from Fashion-MNIST (left) compared with our augmentations of it. The differences between the augmentations and the original are barely perceivable by the human eye. However, they did result in a notable increase in accuracy after being added to the training set.

As the size of the training data quadrupled, the time spent in training a model significantly increased, taking as long as 8 hours. However, the increase in validation accuracy due to augmentation was undeniable. We were convinced that the augmentation of data will contribute a lot in improving the performance of our models. Hence, we started training all of our models with augmented data, sometimes even adding one more augmentation. Table 1 shows our experiment with data augmentation and its results in terms of accuracy. *Appendix 7.9 Augmented Datasets* demonstrates how the data was augmented using Python. *Appendix 7.13 Main Function - Augmented Data* demonstrates the driver code that builds and trains the model using the augmented data.

3 EXPERIMENTS - RESULTS

According to these experiments, using sigmoid for all hidden layers yields the best performance. Using Tanh worked extremely bad even though it looks a lot like sigmoid, and cursory reading states that these two functions should perform similarly. We do not know the reason for this poor performance. Using Relu worked surprisingly well as an output function, only slightly surpassed by softmax. We stuck to using momentum of 1 after attempting several different values of range $[0, 1]$. However, we acknowledge that momentum of 1 is equivalent to momentum of 0 with the weight update being offset by one batch. A lower learning rate seems to work best for 4-layer models as shown by the last line of the "50,000 train" section. Conversely, a higher learning rate seems to work better for 3-layer models as shown by the last layer of the "250,000 train" section. Small weight decay also improves performance slightly, as shown by the final two rows of the table. Data augmentation was effective at improving performance, increasing accuracy by 1.14% over models trained with original data. This was expected because more data gives the model more information with which to fit a function.

4 ENSEMBLE - METHODS

Our final attempt at improving performance was to aggregate our pile of trained models into a single, more intelligent model. The idea is to generate a prediction from many models, and weigh each model's prediction by how reliable it is at predicting a certain class. The reasoning is very intuitive. If model-1 predicts that the input is a shoe, but its classification performance on shoes is poor, its prediction is not to be trusted. Conversely, if model-2 predicts that the input is a dress, and its dress performance is very good, its prediction is deemed trustworthy. In practice, we used a class-wise performance vector over the validation set. This vector is the same shape as the prediction vectors, and each index holds the validation performance of its respective class. Mathematically, the ensemble is the weighted sum of each model's prediction multiplied by its respective performance vector. The ideal case is where each model has specialized in a unique subset of the classes so that they cover each other's shortcomings in the weighted sum.

In an ensemble E containing n models, the feedforward operation given an input vector x is defined as

$$E(x) = \text{hardmax}\left(\sum_{i=1}^{n} p_i \odot m_i(x)\right)$$

where m_i is the i-th model in the ensemble, p_i is the i-th model's class-wise performance vector over the validation set, and $\odot$ is element-wise multiplication. *Appendix 7.3 Ensemble Class* demonstrates the ensemble model class.

4.1 Choosing a Performance Metric

Table 3. Ensemble Accuracies Using Different Prediction Weights

	Validation Accuracy %		
Ensemble Config	Using Precision	Using Recall	Using (Precision $\odot$ Recall)
1	89.92	89.78	89.79
2	89.80	89.57	89.51
3	90.00	89.83	89.79
4	90.32	90.32	90.24
5	89.90	89.71	89.64
6	89.72	89.49	89.49

Validation accuracies for six random ensemble configurations, each tested with three different performance metrics. For every ensemble configuration, using precision results in the same or better accuracy than the other metrics.

There are several performance metrics to choose from, and it is not clear in theory which one would work the best. We can at least eliminate accuracy, because the vast majority of predictions and labels over any one class will be 0, i.e. negative. Since the class-wise prediction and label vectors are so sparse, the model will guess 0 most of the time and be correct, causing class-wise accuracy to be severely inflated. This leaves precision and recall, which solve this sparsity problem by only caring about positive predictions and labels.

We tested six different ensemble models, each containing a random and unique assortment of models. For each ensemble, we tested three different performance metrics: precision, recall, and the element-wise product of them. As shown by Table 3, ensembles using precision consistently outperform the same ensembles using the other two metrics. As a result, we exclusively used precision as the performance metric in every subsequent ensemble.

4.2 Selecting Which Models to Ensemble

To find the best subset of models for the ensemble, we looked at each model's precision vector and chose the models with the highest precision in at least one class. Table 4 shows which models were chosen and the classes for which they had the highest precision.

Appendix 7.4 Selecting Models for Ensemble demonstrates the code used to analyse the precisions of each model and select an optimal set of models to ensemble. *Appendix 7.14 Main Function - Ensemble* shows the driver code used to assemble an ensemble of models and use it to generate a submission.

5 ENSEMBLE - RESULTS

We did see some gains in accuracy from the ensemble but they were minimal. The largest increase we saw was +0.2%, both in the validation accuracy as shown in Table 5 and the public Kaggle leaderboard. We came to a conclusion that the ensembling caused such small improvements because

Table 4. Classification Precision per Clothing Class

	Class Precisions									
Model Name	0	1	2	3	4	5	6	7	8	9
HansModal	**0.853**	0.986	0.827	**0.905**	**0.831**	0.965	**0.783**	0.956	0.964	0.962
AdalynModelina	0.84	0.986	**0.837**	0.891	0.825	0.969	0.77	0.951	0.97	0.966
KateModelle	0.84	**0.99**	0.789	0.886	0.79	0.961	0.783	**0.961**	0.966	0.951
aug-model-3-4-20-35	0.841	0.984	0.827	0.893	0.795	0.963	0.773	0.951	0.961	0.965
aug-model-3-4-5-9	0.824	0.983	0.793	0.893	0.81	**0.973**	0.776	0.959	**0.97**	0.95
model-3-3-5-14	0.845	0.978	0.813	0.878	0.808	0.97	0.743	0.954	0.955	0.964
aug-model-3-7-3-11	0.838	0.989	0.817	0.89	0.785	0.962	0.772	0.949	0.962	**0.967**
model-3-3-5-14	0.845	0.978	0.813	0.878	0.808	0.97	0.743	0.954	0.955	0.964

Classification precision for our eight best models per-class. The highest precision in each class is bolded, and the name of any model that achieved at least one highest precision is also bolded. Only these models that have at least one highest precision are used in the ensemble.

Table 5. Ensemble Accuracies After Selecting Models by Best Precision

Ensemble Config	Validation Accuracy %)
1	90.49
2	90.60

Validation accuracies for two ensemble configurations. For these ensembles, their model sets were chosen using the "best precision" strategy. Ensemble 1 was the first attempt using this strategy. Ensemble 2 was created using the same strategy after new models were introduced.

all of our models have an extremely similar distributions of competency. Table 4 shows that all the models have almost the same precisions to each other for every class, meaning they are smart and dumb in all the same areas. This is far from the ideal case for ensembling as described above, where each class would be covered by a model that excels at classifying it. There was not much to be gained from ensembling our particular set of models.

6 CONCLUSION

We were surprised how far the multilayer perceptron could be pushed on the Fashion-MNIST dataset. By far, the most improvement came from fine-tuning the MLP architecture and hyperparameters using the validation accuracy as a metric. We found that 3-4 layers worked well, along with a learning rate of 0.03-0.07, momentum of 1 (functionally equivalent to momentum of 0), and weight decay of -0.01. The best-performing function set proved to be sigmoid for all hidden layers, softmax for the output layer, and cross-entropy for the loss function, although relu on the output layer and MSE loss also worked extremely well together.

The next most significant technique was data augmentation, which resulted in a 1.14% accuracy increase from the best model trained on non-augmented data. This is an expected result of data augmentation because it helps diversify our data set while gaining more data.

The final and smallest increase in performance was achieved by carefully selecting a set of of fully trained models and combining them into an ensemble. This technique caused a 0.2% increase in accuracy, both on the validation set and the Kaggle leaderboard when compared to the best individual model. Since all of our models had very similar distributions of class-wise performance, they could not complement each other very well in the ensemble's weighted sum. Nonetheless, their small differences were sufficient to achieve a notable increase in performance.

7 APPENDIX

7.1 Multilayer Perceptron Class

Listing 1. Multilayer Perceptron

```
classdef MultilayerPerceptron < handle

    properties
        layers
        cost_func
        d_cost_func
    end

    methods
        function obj = MultilayerPerceptron(cost_func, d_cost_func)
            obj.layers = [];
            obj.cost_func = cost_func;
            obj.d_cost_func = d_cost_func;
        end

        function add_layer(obj, layer)
            obj.layers = [obj.layers layer];
        end

        function [losses, best_metric, all_metrics] = fit(obj, examples, labels,
             epochs, batch_size, stop_buff, stop_thresh, test_data, test_labels,
             metric_func, checkpoint_name)
            for i = 1:size(obj.layers, 2)
                disp(size(obj.layers(i).W));
            end

            avg_losses = [];

            batch_losses = zeros(batch_size);
            epoch_losses = [];
            temp_avg_losses = [];
            metric = [];
            best_metric = 0
            disp([0.0000 metric_func(hardmax(obj.frozen_forward(test_data)),
                 test_labels)]);

            for e = 1:epochs
                perm = randperm(size(examples, 2));
                examples = examples(:, perm);
                labels = labels(:, perm);

                for b = 1:int32(ceil(size(examples, 2) / batch_size) + 1)
                    % fprintf('batch %d / %d\n', b, int32(ceil(size(examples, 2) /
                         batch_size) + 1));

                    start_i = (b - 1) * (batch_size) + 1;
                    end_i = min(b * batch_size, size(examples, 2));
                    if start_i > end_i
                        break
                    end
```

```
                    for i = start_i:end_i
                        a = obj.forward(examples(1:size(examples, 1), i));
                        batch_losses(i - start_i + 1) = obj.backward(a, labels(1:size(labels, 1), i));
                    end
                    obj.update((e-1)/epochs); % pass current progress
                    avg_losses = [avg_losses mean(batch_losses(1:(end_i - start_i + 1)))];
                end

                metric = [metric metric_func(hardmax(obj.frozen_forward(test_data)), test_labels)];
                disp([e metric(size(metric, 2))]);

                if(metric(length(metric)) > best_metric)
                    best_metric = metric(length(metric));
                    obj.try_save_checkpoint(checkpoint_name, best_metric);
                end

                temp_avg_losses = avg_losses(:,:);
                epoch_losses = [epoch_losses mean(temp_avg_losses)];
                if(doEarlyStop(e, metric, stop_buff, stop_thresh))
                    break
                end
            end

            losses = epoch_losses;
            all_metrics = metric;
        end

        function try_save_checkpoint(obj, filename, metric)

            % save current model
            metric_str = num2str(metric);
            model_timestamp = filename + "_METRIC_" + metric_str(3:size(metric_str, 2)) + '.mat';
            mlp = obj;
            save(model_timestamp, "mlp");

            % find and delete the other model
            files = ls('models/*.mat');

            filename = extractAfter(filename, "models/") + "_METRIC_";

            for n = 1:height(files)

                if contains(files(n,:), filename) & (strtrim(files(n,:)) ~= extractAfter(model_timestamp, "models/")) % to avoid deleting the current model
                   delete("models/" + files(n,:));
                   break
                end
            end
        end
```

```

        function out = forward(obj, vec)
            for layer = obj.layers
                vec = layer.forward(vec);
            end
            out = vec;
        end

        function out = frozen_forward(obj, vec)
            for layer = obj.layers
                vec = layer.frozen_forward(vec);
            end
            out = vec;
        end

        function loss = backward(obj, last_a, target)
            last_n = obj.layers(size(obj.layers, 2)).n;
            last_s = obj.layers(size(obj.layers, 2)).d_trans_func(last_n) .* obj.d_cost_func(last_a, target);

            obj.layers(size(obj.layers, 2)).add_to_s(last_s);

            for i = (size(obj.layers, 2) - 1):-1:1
                obj.layers(i).backward(obj.layers(i+1).W, obj.layers(i+1).s)
            end
            loss = mean(obj.cost_func(last_a, target));
        end

        function update(obj, epoch_progress)
            for i = 1:size(obj.layers)
                obj.layers(i).update(epoch_progress);
            end
        end
    end
end
```

7.2 Perceptron Layer Class

Listing 2. Perceptron Layer

```
classdef PerceptronLayer < handle
    properties
        trans_func % layer transfer function
        d_trans_func % derivative of layer transfer function
        batch_count % current count of how many examples have been seen in the current batch
        lr_max
        lr_min
        W % layer weights
        b % layer biases
        n % most recent net input, needed for backprop (backward)
        a % most recent activation, needed for last layer sensitivity
        s % most recent sensitivity, needed for backprop
```

```matlab
        p % most recent input vector, needed to compute each p's sensitivity (add_to_s)
        avg_s % sum of sensitivities in the current batch, needed for gradient descent (update)
        avg_sp % sum of the sensitivity * p-input' matrices in the current batch. Averaged at update
        last_W_update
        last_b_update
        momentum
    end

    methods
        function obj = PerceptronLayer(arg1, arg2, trans_func, d_trans_func, learn_rate_max, learn_rate_min, momentum, std)
            obj.trans_func = trans_func;
            obj.d_trans_func = d_trans_func;
            obj.lr_max = learn_rate_max;
            obj.lr_min = learn_rate_min;
            obj.momentum = momentum;

            % If both arguments are scalers, create random weight matrix and
            % bias vector. Otherwise, use the provided weights and biases
            if size(arg1) == [1 1] & size(arg2) == [1 1]
                obj.W = normrnd(0, std, arg1, arg2);
                obj.b = normrnd(0, std, arg1, 1);
            else
                obj.W = arg1;
                obj.b = arg2;
            end

            obj.reset_for_next_batch();
            obj.last_W_update = zeros(size(obj.W));
            obj.last_b_update = zeros(size(obj.W, 1), 1);
        end

        function output = forward(obj, p)
            obj.batch_count = obj.batch_count + 1;
            obj.p = p;

            obj.n = obj.W * p + obj.b;
            obj.a = obj.trans_func(obj.n);
            output = obj.a;
        end

        function output = frozen_forward(obj, p)
            output = obj.trans_func(obj.W * p + obj.b);
        end

        function backward(obj, next_W, next_s)
            obj.add_to_s(obj.d_trans_func(obj.n) .* (next_W' * next_s));
        end

        function update(obj, epoch_progress)
            obj.avg_s = obj.avg_s / obj.batch_count;
```

```
            obj.avg_sp = obj.avg_sp / obj.batch_count;

            if obj.lr_min == obj.lr_max
               lr = obj.lr_min;
            else
               lr = obj.lr_max - (obj.lr_max - obj.lr_min) * epoch_progress;
            end

            obj.W = obj.W + ((lr * (1 - obj.momentum) * obj.avg_sp) + (obj.momentum *
                obj.last_W_update));
            obj.b = obj.b + ((lr * (1 - obj.momentum) * obj.avg_s) + (obj.momentum *
                obj.last_b_update));

            obj.reset_for_next_batch();
        end

        function add_to_s(obj, s_vec)
            obj.avg_s = obj.avg_s + s_vec;
            obj.avg_sp = obj.avg_sp + (s_vec * obj.p');
            obj.s = s_vec;
        end

        function reset_for_next_batch(obj)
            obj.last_W_update = obj.avg_sp;
            obj.last_b_update = obj.avg_s;

            obj.avg_sp = zeros(size(obj.W));
            obj.avg_s = zeros(size(obj.W, 1), 1);

            obj.batch_count = 0;
        end

        function print(obj)
            disp('Weights');
            disp(obj.W);
            disp('');

            disp('Biases');
            disp(obj.b);
            disp('');
        end
    end
end
```

7.3 Ensemble Class

Listing 3. Ensemble Class

```
classdef Ensemble < handle

    properties
        model_list
    end

```

```
    methods
        function obj = Ensemble()
            obj.model_list = [];
        end

        function add_model(obj, model)
            % add model to list
            obj.model_list = [obj.model_list model];
        end

        function [losses, best_metrics, all_metrics] = fit(obj, examples, labels, epochs, batch_size, stop_buff, stop_thresh, test_data, test_labels, metric_func)
            % fit all models in model_list one after another. Also, record
            % loss, best_metric, and all_metrics for each model in a
            % matrix, where each row is the result from one model.

            losses = [];
            best_metrics = [];
            all_metrics = [];

            for i = 1:size(obj.model_list, 2)
                [curr_losses, curr_best_metrics, curr_all_metrics] = obj.model_list(i).fit( ...
                    examples, ...
                    labels, ...
                    epochs, ...
                    batch_size, ...
                    stop_buff, ...
                    stop_thresh, ...
                    test_data, ...
                    test_labels, ...
                    metric_func ...
                );
                losses = [losses; curr_losses];
                best_metrics = [best_metrics; curr_best_metrics];
                all_metrics = [all_metrics; curr_all_metrics];
            end

            disp('FINAL INDIVIDUAL METRICS');
            disp(best_metrics);
            disp('FINAL ENSEMBLE METRIC');
            disp(metric_func(hardmax(obj.frozen_forward(test_data, test_data, test_labels)), test_labels));
        end

        function weighted_votes = frozen_forward(obj, vec, test_data, test_labels)
            % perform the voting algorithm. The final votes are the average
            % of the weighted sum of each model's predictions, where the weights are the
            % model's (precision * recall) score per-class.

            weighted_votes = zeros(size(test_labels));

```

```
            for i = 1:size(obj.model_list, 2)
                num_layers = size(obj.model_list(i).layers, 2);

                % temporrarily swap out the last layer transfer function for softmax
                % temp_handle = obj.model_list(i).layers(num_layers).trans_func;
                % obj.model_list(i).layers(num_layers).trans_func = @softmax;

                % get preditioncs
                preds = obj.model_list(i).frozen_forward(vec);

                % revert last layer transfer function to original function
                % obj.model_list(i).layers(num_layers).trans_func = temp_handle;

                % compute model weights
                confidence_weights = repmat(compute_prec_rec_weight(hardmax(preds), test_labels), 1, size(vec, 2));

                % add to the weighted sum
                weighted_votes = weighted_votes + (confidence_weights .* preds);
            end

            % average the weighted sum
            weighted_votes = weighted_votes * (1 / size(vec, 2));
        end
    end
end
```

7.4 Selecting Models for Ensemble

Listing 4. Function that selects the models with the best performance in each class

```
function precision_testing()
    % make all folders fisible to matlab
    addpath('cost_functions');
    addpath('d_cost_functions');
    addpath('transfer_functions');
    addpath('d_transfer_functions');
    addpath('utility');
    addpath('data');
    addpath('nn_components');
    addpath('augmented');

    all_submission_data = readmatrix('test.csv'); % read all 10,000 submission datapoints into matrix
    submission_data = all_submission_data(:, 2:785)' * (1/255); % get rid of the useles "id" column in the submission file

    % split training and validation data
    all_examples = readmatrix('train.csv');
    all_labels = to_one_hot(all_examples(:, 2), 0, 9);
    all_examples = all_examples(:, 3:786)' * (1/255);

    % Training datapoints out of 60,000. The rest are used for validation
    TRAIN_SIZE = 50000;
```

```
    valid_data = all_examples(:, (TRAIN_SIZE + 1):60000);
    valid_labels = all_labels(:, (TRAIN_SIZE + 1):60000);

    % LOAD MODELS
    % load mlp models from .mat files
    model_files = [
        "models/HansModal_2021_3_12_22_17_METRIC_904.mat"
        "models/AdalynModelina_2021_3_12_1_15_METRIC_9014.mat"
        "models/KateModelle_aug_2021_3_11_14_36_METRIC_8926.mat"
        "models/aug_model_2021_3_4_20_35.mat"
        "models/aug_model_2021_3_4_5_9.mat"
        "models/model_2021_3_3_5_14.mat"
        "models/aug_model_2021_3_7_3_11.mat"
        "models/model_2021_3_2_14_27.mat"
        "models/model_2021_3_2_17_49.mat"
        "models/model_2021_3_3_5_14.mat"];

    precisions = [];
    best_model_per_class = [];
    unique_models = [""];

    for i = 1:length(model_files)
        load(model_files(i), "mlp");
        preds = hardmax(mlp.frozen_forward(valid_data));
        precisions = [precisions compute_prec_rec_weight(preds, valid_labels)];
    end
    disp(precisions);

    for i = 1:size(precisions, 1)
        class_precs = precisions(i, 1:size(precisions, 2));
        [maxval, maxi] = max(class_precs);
        class_best = model_files(maxi);

        best_model_per_class = [best_model_per_class; class_best];
        if ~ismember(class_best, unique_models)
            unique_models = [unique_models; class_best];
        end
    end
    disp(best_model_per_class);
    disp(unique_models);
end
```

Listing 5. Function that computes precision and recall

```
function weights = compute_prec_rec_weight(predictions, labels)
    true_pos = sum(predictions & labels, 2);
    precisions = true_pos ./ sum(predictions, 2); % TP / (TP + FP)
    recalls = true_pos ./ sum(labels, 2); % TP / (TP + FN)

    weights = precisions;
end
```

7.5 Accuracy

Listing 6. Accuracy Function

```
function a = accuracy(predictions, labels)
    count = 0;
    label_size = size(labels);

    for i = 1:label_size(2)
        if predictions(1:label_size(1), i) == labels(1:label_size(1), i)
            count = count + 1;
        end
    end
    a = count / label_size(2);
end
```

7.6 Transfer Functions and their Derivatives

Listing 7. Linear

```
function l = linear(x)
    l = x;
end
```

Listing 8. Linear Derivative

```
function l = d_linear(x)
    l = ones(size(x));
end
```

Listing 9. Tanh

```
function t = my_tanh(x)
    t = (exp(x) - exp(-x)) .* ((exp(x) + exp(-x)) .^ (-1));
end
```

Listing 10. Tanh Derivative

```
function t = d_my_tanh(x)
    t = 1 - (my_tanh(x) .^ 2);
end
```

Listing 11. ReLU

```
function r = relu(x)
    x(x < 0) = 0;
    r = x;
end
```

Listing 12. ReLU Derivative

```
function r = d_relu(x)
    x(x < 0) = 0;
    x(x >= 0) = 1;
    r = x;
```

```
end
```

Listing 13. Sigmoid

```
function s = sigmoid(x)
    s = (1 + exp(-x)) .^ (-1);
end
```

Listing 14. Sigmoid Derivative

```
function s = d_sigmoid(x)
    s = sigmoid(x) .* (1 - sigmoid(x));
end
```

Listing 15. Softmax

```
function s = softmax(x)
    exp_of_x = exp(x);
    s = exp_of_x ./ sum(exp_of_x, 1);
end
```

Listing 16. Softmax Derivative

```
function s = d_softmax(x)
    % softmax(x) is always used as softmax(sigmoid(x)), so we apply the chain rule
        here:
    % d_softmax(sigmoid(x)) * d_sigmoid(x)
    s = softmax(x) .* (1 - softmax(x));
end
```

7.7 Loss Functions and their Derivatives

Listing 17. Mean Squared Error

```
function error = squared_error(a, t)
    e = a - t;
    error = e .* e;
end
```

Listing 18. Mean Squared Error Derivative

```
function error = d_squared_error(a, t)
    error = t - a;
end
```

Listing 19. Cross-Entropy

```
function loss = cross_entropy(a, t)
    loss = -sum(t .* log(a));
end
```

Listing 20. Cross-Entropy Derivative (softmax derivative included)

```
function error = d_cross_entropy(a, t)
    % error = -(t ./ a) + ((1-t) ./ (1-a));
    error = t - a;
end
```

7.8 Hardmax Function

Listing 21. Hardmax

```
function out = hardmax(x)
    for i = 1:size(x, 2)
        [max_num, max_index] = max(x(1:size(x, 1), i));
        x(1:size(x, 1), i) = zeros(size(x, 1), 1);
        x(max_index, i) = 1;
    end
    out = x;
end
```

7.9 Generating Augmented Datasets

Listing 22. Dataset Augmentation Script (Python)

```
"""
Functions
- rotate, blur, add_salt_pepper_noise, distort, translate, scale, flip
- augment: performs some or all of those augmentation for each data sample in train set

Instructions
1) In 'uwb-css-485-winter-2021' directory, make duplicate of 'train.csv' and name it 'train_for_aug.csv'
2) Make following changes to 'train_for_aug.csv'
    - delete the first row ("id", "label", "pixel1", "pixel2", ...)
    - delete the validation data (rows from 50,001 - 60,000)
3) In the same depth as the 'utility' folder, make a directory called 'augmented'
4) Currently this code performs 4 different augmentation. if you want, make changes to augment() function
5) Run this code and the output will be saved in 'augmented' directory
6) When you train your model on MATLAB, train with main_agumented() instead of main()

"""
import pandas as pd
import numpy as np
from numpy import genfromtxt
from IPython.display import Image
from PIL import Image as PILimage
import scipy.ndimage
import cv2
import random
import csv
from csv import reader, writer
from datetime import datetime

```

```python
import gc

def blur(sample, sigma=0.58):
    return scipy.ndimage.filters.gaussian_filter(sample, sigma=sigma)

def add_salt_pepper_noise(sample, prob=0.05):
    output = np.zeros(sample.shape,np.uint8)
    thres = 1 - prob
    for i in range(sample.shape[0]):
        for j in range(sample.shape[1]):
            rdn = random.random()
            if rdn < prob:
                output[i][j] = 0
            elif rdn > thres:
                output[i][j] = 255
            else:
                output[i][j] = sample[i][j]
    return output

def distort(img, orientation='horizontal', func=np.sin, x_scale=0.05, y_scale=5):
    assert orientation[:3] in ['hor', 'ver'], "dist_orient should be 'horizontal'|'vertical'"
    assert func in [np.sin, np.cos], "supported functions are np.sin and np.cos"
    assert 0.00 <= x_scale <= 0.1, "x_scale should be in [0.0, 0.1]"
    assert 0 <= y_scale <= min(img.shape[0], img.shape[1]), "y_scale should be less then image size"
    img_dist = img.copy()

    def shift(x):
        return int(y_scale * func(np.pi * x * x_scale))

    for _ in range(3):
        for i in range(img.shape[orientation.startswith('ver')]):
            if orientation.startswith('ver'):
                img_dist[:, i] = np.roll(img[:, i], shift(i))
            else:
                img_dist[i, :] = np.roll(img[i, :], shift(i))

    return img_dist

def translate(sample, shift=1, direction = 1):
    # direction 1,2,3,4 correstponds to
    # left, right, upward, downward

    if direction < 3:
        shifted_area = np.zeros((28, shift))
        ax = 1
    else:
        shifted_area = np.zeros((shift, 28))
        ax = 0

    if direction == 1:
        sample = sample[:, shift:]
    elif direction == 2:
```

```
            sample = sample[:, :-shift]
        elif direction == 3:
            sample = sample[shift:, :]
        else:
            sample = sample[:-shift, :]

        if direction % 2 == 0:  # downard and right
            shifted = np.concatenate((shifted_area, sample),axis=ax)
        else:
            shifted = np.concatenate((sample, shifted_area),axis=ax)

        return shifted

def scale(sample, dsize=(28,28), resultsize=(28,28), interpolation=cv2.INTER_CUBIC):
    res = cv2.resize(sample, dsize=dsize, interpolation=cv2.INTER_CUBIC)

    if dsize[0] > resultsize[0]: # cut from each side almost equally
        diff = dsize[0] - resultsize[0]
        res = res[:,diff//2 : -(diff)//2]

    if dsize[1] > resultsize[1]:
        diff = dsize[1] - resultsize[1]
        res = res[diff//2:-(diff)//2,:]

    return res

def rotate(sample, angle=2):
    return scipy.ndimage.rotate(sample, angle, reshape=False)

def flip(sample):
    return np.fliplr(sample)

def augment(filename):
    t = datetime.now().strftime("%Y_%m_%d_%H_%M_%S")
    count = 0

    with open(f'../augmented/augmented_train_{t}.csv', 'a', newline='') as f1:
        csvwriter1 = writer(f1)
        with open(f'../augmented/augmented_label_{t}.csv', 'a', newline='') as f2:
            csvwriter2 = writer(f2)

            with open(filename, 'r') as read_obj:
                csv_reader = reader(read_obj)

                for row in csv_reader:
                    count += 1
                    if count % 1000 == 0: print(f'Progress: {count}/50000')
                    if count % 500 == 0: gc.collect()

                    arr = np.asarray(row).astype(float)
                    label = arr[1]
                    arr = arr[2:]

                    sample = arr.reshape(28,28)
```

```
                    """
                    !!! Make changes here if you want more or less augmentation !!!
                        Don't forget to .ravel()
                        Don't forget to change the range(n) in the for loop for writing labels
                    """
                    # writing origianl and augmented images
                    csvwriter1.writerows([sample.ravel(),\
                                          rotate(sample,-2).ravel(),\
                                          rotate(sample, 2).ravel(),\
                                          flip(sample).ravel(),\
                                          distort(sample, x_scale=0.03, y_scale=2).ravel()])

                    # writing labels
                    for _ in range(5):
                        csvwriter2.writerow([label])

    return t

def main():

    t = augment("../uwb-css-485-winter-2021/train_for_aug.csv") # this file should not contain the first row and the rows from 50,001 - 60,000

    print("\nFiles generated: ")
    print(f'augmented/augmented_train_{t}.csv')
    print(f'augmented/augmented_label_{t}.csv')

if __name__ == "__main__":
    main()
```

7.10 Early Stopping

Listing 23. Function that decides whether to stop early

```
function do_stop = doEarlyStop(ep, values, early_stop_buff_size, early_stop_threshold)
    % return true to do an early stop IF:
    % the current error is 0
    % OR BOTH the past <buff_size> errors have a standard deviation below
    %         a threshold AND there have been at least <buff_size> epochs
    if(ep >= early_stop_buff_size)
        disp(mean_trend(values( (size(values, 2) - early_stop_buff_size + 1):size(values, 2) )))
    end
    do_stop = (ep >= early_stop_buff_size ...
            && mean_trend(values( (size(values, 2) - early_stop_buff_size + 1):size(values, 2) )) <= early_stop_threshold);
end
```

Listing 24. Function that returns the average change in validation metric over N epochs

```
function t = mean_trend(vec)
    if(size(vec, 2) == 1)
```

```
        t = 0;
    else
        t = (vec(size(vec, 2)) - vec(1)) / ((size(vec, 2) - 1));
    end
end
```

7.11 One-Hot Encoding and Decoding

Listing 25. One-Hot Encoding - Converts a list of ints to a matrix of one-hot column vectors

```
function one_hot = to_one_hot(colwise_nums, min, max)
    one_hot = (colwise_nums == min:max)';
end
```

Listing 26. One-Hot Decoding - Converts a matrix of one-hot column vectors to a list of ints

```
function nums = one_hot_to_int(colwise_onehot, min, max)
    nums = colwise_onehot' * (min:max)';
end
```

7.12 Main Function - Standard

Listing 27. Driver code that initializes data builds and trains model and generates a submission file

```
function main()
    % make all folders fisible to matlab
    addpath('cost_functions');
    addpath('d_cost_functions');
    addpath('transfer_functions');
    addpath('d_transfer_functions');
    addpath('utility');
    addpath('data');
    addpath('nn_components');

    % Training datapoints out of 60,000. The rest are used for validation
    TRAIN_SIZE = 50000;

    % READ ALL DATA

    all_data = readmatrix('train.csv'); % read all 60,000 labeled datapoints and labels into matrix
    all_submission_data = readmatrix('test.csv'); % read all 10,000 submission datapoints into matrix
    submission_data = all_submission_data(:, 2:785)' * (1/255); % normalize, and get rid of the useles "id" column in the submission file

    all_examples = all_data(:, 3:786)' * (1/255); % normalize datapoints
    disp(all_examples(1:20, 1:10));
    all_labels = to_one_hot(all_data(:, 2), 0, 9); % convert labels (0-9) to one-hot vectors

    % split training and validation data
    train_data = all_examples(:, 1:TRAIN_SIZE);
    train_labels = all_labels(:, 1:TRAIN_SIZE);
    test_data = all_examples(:, (TRAIN_SIZE + 1):60000);
```

```
    test_labels = all_labels(:, (TRAIN_SIZE + 1):60000);

    % hyperparameters
    epochs = 1000;
    stop_buff = 1;
    stop_thresh = -1;
    std = 0.4;
    batch_size = 32;
    momentum = 0;
    lr = 0.05;

    % build model
    mlp = MultilayerPerceptron(@cross_entropy, @d_cross_entropy);

    mlp.add_layer(PerceptronLayer(256, 784, @my_tanh, @d_my_tanh, lr, momentum, std));
    mlp.add_layer(PerceptronLayer(10, 256, @softmax, @d_softmax, lr, momentum, std));

    % train the model
    [losses, acc, acc_list] = mlp.fit( ...
        train_data, ...
        train_labels, ...
        epochs, ...
        batch_size, ...
        stop_buff, ...
        stop_thresh, ...
        test_data, ...
        test_labels, ...
        @accuracy ...
    );
    disp(losses);
    disp(acc_list);

    % predict labels for submission data
    final_preds = one_hot_to_int(hardmax(mlp.frozen_forward(submission_data)), 0, 9);

    % format matrix for submission
    submission_matrix = [(60001:70000)' final_preds];

    % write submission matrix to data/SUBMISSION.csv
    writematrix(submission_matrix, 'data/SUBMISSION.csv');
end
```

7.13 Main Function - Augmented Data

Listing 28. Driver code that initializes the augmented data builds and trains model and generates a submission

```
% !!! IMPORTNAT !!!

% this file saves the model it trained to a directory called 'models'
% please make sure you have a directory 'models' before running this code

function main_augmented()
    % make all folders fisible to matlab
    addpath('cost_functions');
```

```
addpath('d_cost_functions');
addpath('transfer_functions');
addpath('d_transfer_functions');
addpath('utility');
addpath('data');
addpath('nn_components');
addpath('augmented');

% read data -- CHANGE THE FILE TO YOUR AUGMENTED DATASET

aug_data = readmatrix('augmented/small_augmented_train.csv');
aug_label = readmatrix('augmented/small_agumented_label.csv');

% aug_data = readmatrix('augmented/augmented_train_2021_03_12_15_15_00.csv');
% aug_label = readmatrix('augmented/augmented_label_2021_03_12_15_15_00.csv');
disp(size(aug_label));

all_submission_data = readmatrix('test.csv'); % read all 10,000 submission datapoints into matrix
submission_data = all_submission_data(:, 2:785)' * (1/255); % get rid of the useles "id" column in the submission file

train_data = aug_data' * (1/255); % normalize datapoints
train_labels = to_one_hot(aug_label, 0, 9); % convert labels (0-9) to one-hot vectors

% split training and validation data
all_examples = readmatrix('train.csv');
all_labels = to_one_hot(all_examples(:, 2), 0, 9);
all_examples = all_examples(:, 3:786)' * (1/255);


% Training datapoints out of 60,000. The rest are used for validation
TRAIN_SIZE = 50000;
valid_data = all_examples(:, (TRAIN_SIZE + 1):60000);
valid_labels = all_labels(:, (TRAIN_SIZE + 1):60000);

% hyperparameters
epochs = 20;
stop_buff = 1;
stop_thresh = -1;
std = 0.4;
batch_size = 24; %500;
momentum = 1.1;
lr_max = 1;
lr_min = 0.02;

% build model
mlp = MultilayerPerceptron(@cross_entropy, @d_cross_entropy);

mlp.add_layer(PerceptronLayer(350, 784, @sigmoid, @d_sigmoid, lr_max, lr_min, momentum, std));
mlp.add_layer(PerceptronLayer(170, 350, @sigmoid, @d_sigmoid, lr_max, lr_min, momentum, std));
```

```
    mlp.add_layer(PerceptronLayer(10, 170, @relu, @d_relu, lr_max, lr_min, momentum, std));


    % change each run to make identifying models easier
    model_name = "HansModal";

    t = datetime('now');
    model_timestamp = "models/" + model_name + "_" + year(t) + '_' + month(t) + '_' + day(t) + '_' + hour(t) + '_' + minute(t);

    % train the model
    [losses, acc, acc_list] = mlp.fit( ...
        train_data, ...
        train_labels, ...
        epochs, ...
        batch_size, ...
        stop_buff, ...
        stop_thresh, ...
        valid_data, ...
        valid_labels, ...
        @accuracy, ...
        model_timestamp ...
    );
    disp(" ==================== losses ====================" );
    disp(losses);
    disp(" ==================== acc_list ====================" );
    disp(acc_list);

    load("models/HansModal_2021_3_12_22_17_METRIC_904.mat", "mlp");
    % predict labels for submission data
    final_preds = one_hot_to_int(hardmax(mlp.frozen_forward(submission_data)), 0, 9);

    % format matrix for submission
    submission_matrix = [(60001:70000)' final_preds];

    % write submission matrix to data/SUBMISSION.csv
    submission_timestamp = "data/" + model_name + "_submission_" + year(t) + '_' + month(t) + '_' + day(t) + '_' + hour(t) + '_' + minute(t) + '.csv';
    writematrix(submission_matrix, submission_timestamp);

    % save the model
    % model_timestamp = "models/models/" + model_name + "_" + year(t) + '_' + month(t) + '_' + day(t) + '_' + hour(t) + '_' + minute(t) + '.mat';
    % save(model_timestamp, "mlp");
end
```

7.14 Main Function - Ensemble

Listing 29. Driver code that initializes data builds an Ensemble and uses it to generate a submission

```
function ensemble_main()
    % make all folders fisible to matlab
    addpath('cost_functions');
```

```
    addpath('d_cost_functions');
    addpath('transfer_functions');
    addpath('d_transfer_functions');
    addpath('utility');
    addpath('data');
    addpath('nn_components');
    addpath('augmented');

    all_submission_data = readmatrix('test.csv'); % read all 10,000 submission
        datapoints into matrix
    submission_data = all_submission_data(:, 2:785)' * (1/255); % get rid of the
        useles "id" column in the submission file

    % split training and validation data
    all_examples = readmatrix('train.csv');
    all_labels = to_one_hot(all_examples(:, 2), 0, 9);
    all_examples = all_examples(:, 3:786)' * (1/255);

    % Training datapoints out of 60,000. The rest are used for validation
    TRAIN_SIZE = 50000;
    valid_data = all_examples(:, (TRAIN_SIZE + 1):60000);
    valid_labels = all_labels(:, (TRAIN_SIZE + 1):60000);


    ensemble = Ensemble();

    % LOAD MODELS
    % load mlp models from .mat files
    load("models/aug_model_2021_3_7_3_11.mat", "mlp");
    ensemble.add_model(mlp);

    load("models/KateModelle_aug_2021_3_11_14_36_METRIC_8926.mat", "mlp");
    ensemble.add_model(mlp);

    load("models/HansModal_2021_3_12_22_17_METRIC_904.mat", "mlp");
    ensemble.add_model(mlp);

%     load("models/model_2021_3_3_5_14.mat", "mlp");
%     ensemble.add_model(mlp);

    load("models/aug_model_2021_3_4_5_9.mat", "mlp");
    ensemble.add_model(mlp);

    load("models/AdalynModelina_2021_3_12_1_15_METRIC_9014.mat", "mlp");
    ensemble.add_model(mlp);

    disp(accuracy(hardmax(ensemble.frozen_forward(valid_data, valid_data, valid_labels
        )), valid_labels));

    % predict labels for submission data
    final_preds = one_hot_to_int(hardmax(ensemble.frozen_forward(submission_data,
        valid_data, valid_labels)), 0, 9);

    % format matrix for submission
```

```
    submission_matrix = [(60001:70000)' final_preds];

    % write submission matrix
    t = datetime('now');
    submission_timestamp = "data/ensemble_submission_" + year(t) + '_' + month(t) + '_' + day(t) + '_' + hour(t) + '_' + minute(t) + '.csv';
    writematrix(submission_matrix, submission_timestamp);

end
```