NEURAL NETWORKS - 1ST PROJECT REPORT

Victoria Galanopoulou 10630

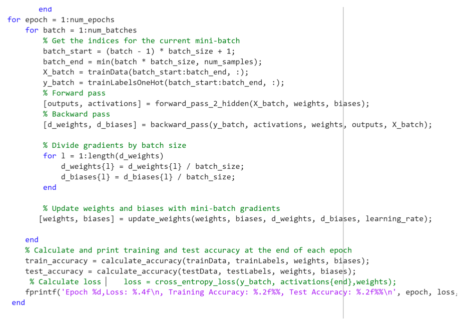
November 2024

For this project I constructed from scratch an MLP Neural Network applying the back propagation algorithm using the CIFAR-10 library.

In this report, first of all, I will analyze the code, then I will explain some errors I conducted during the training, I will explain the differences in accuracies depending on changes in the parameters of the model and some techniques that I tried to improve my model's performance. Last but no least, I will compare my MLP NN with Nearest Neighbor and Nearest Centroid algorithms.

1 Explanation of the code

For the implementation of the NN I used the programming language MATLAB. Below I will explain each function used for the network to work separately. For each epoch, the training data is divided into mini-batches, and for each batch, the input data (X batch) and corresponding labels (y batch) are extracted. A forward pass is performed to calculate the activations and outputs of the network using the forward pass function. After that, the backward pass is executed using the backward pass function to compute the gradients for the weights and biases. These gradients are then used to update the network's weights and biases. Once the training for all batches in an epoch is complete, the forward pass is performed on the test data to calculate the test accuracy, and the training accuracy is also computed. Finally, the loss, training accuracy, and test accuracy are displayed for each epoch to monitor the network's performance.



```
        end
    for epoch = 1:num_epochs
        for batch = 1:num_batches
            % Get the indices for the current mini-batch
            batch_start = (batch - 1) * batch_size + 1;
            batch_end = min(batch * batch_size, num_samples);
            X_batch = trainData(batch_start:batch_end, :);
            y_batch = trainLabelsOneHot(batch_start:batch_end, :);
            % Forward pass
            [outputs, activations] = forward_pass_2_hidden(X_batch, weights, biases);
            % Backward pass
            [d_weights, d_biases] = backward_pass(y_batch, activations, weights, outputs, X_batch);

            % Divide gradients by batch size
            for l = 1:length(d_weights)
                d_weights{l} = d_weights{l} / batch_size;
                d_biases{l} = d_biases{l} / batch_size;
            end

            % Update weights and biases with mini-batch gradients
           [weights, biases] = update_weights(weights, biases, d_weights, d_biases, learning_rate);

        end
        % Calculate and print training and test accuracy at the end of each epoch
        train_accuracy = calculate_accuracy(trainData, trainLabels, weights, biases);
        test_accuracy = calculate_accuracy(testData, testLabels, weights, biases);
         % Calculate loss |    loss = cross_entropy_loss(y_batch, activations{end},weights);
        fprintf('Epoch %d,Loss: %.4f\n, Training Accuracy: %.2f%%, Test Accuracy: %.2f%%\n', epoch, loss,
     end
```

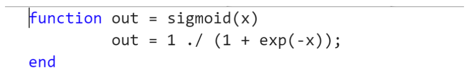
Figure 1: Basic algorithm of the network

To enhance the performance of the algorithms, we employed double precision and normalized the data by dividing by 255. This step was crucial to mitigate rounding errors during calculations and to ensure higher accuracy, particularly in high-dimensional data scenarios, such as this case with 3072 features per image. Normalization also helps scale the input features, enabling better convergence and stability in numerical computations.

In my code weights is a cell array containing the weight matrices for each layer. Each weight matrix has dimensions [Ni, No], where Ni is the number of neurons in the previous layer and No is the number of neurons in the current layer. Biases is a cell array containing the bias vectors for each layer. Each bias vector has dimensions [1, No], where No is the number of neurons in the current layer.

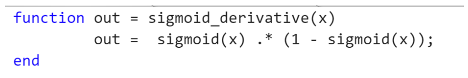
1.1 Activation functions

First of all, as an activation function for the internal layers I used sigmoid function, so in the figures below sigmoid and its derivative are presented.



```
function out = sigmoid(x)
        out = 1 ./ (1 + exp(-x));
end
```

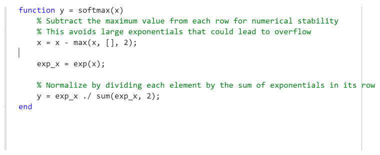
Figure 2: Sigmoid function



```
function out = sigmoid_derivative(x)
        out =  sigmoid(x) .* (1 - sigmoid(x));
end
```

Figure 3: Sigmoid Derivative

For the output layer, I used `softmax` because it converts the row values in the output layer into probabilities that can be interpreted as confidence levels for each class. By transforming the values in the output layer's neurons into probabilities, softmax allows the model to make probabilistic predictions, making it useful for tasks like image classification. It also highlights the largest scores, amplifying differences between the most likely classes and less likely ones, aiding in more confident predictions.



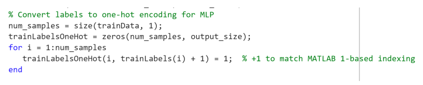
```
function y = softmax(x)
    % Subtract the maximum value from each row for numerical stability
    % This avoids large exponentials that could lead to overflow
    x = x - max(x, [], 2);

    exp_x = exp(x);

    % Normalize by dividing each element by the sum of exponentials in its row
    y = exp_x ./ sum(exp_x, 2);
end
```

Figure 4: Softmax function

1.2 Converting labels - One Hot Encooding

`trainLabels`: This is an array containing the class labels for each sample in the training set which range from 0 to 9, with each number representing one of the 10 classes.

`trainLabelsOneHot`: This is a matrix initialized to hold the one-hot encoded labels for all samples. Each row represents one sample, and each column represents a class. It has dimensions [num samples, 10].



```
% Convert labels to one-hot encoding for MLP
num_samples = size(trainData, 1);
trainLabelsOneHot = zeros(num_samples, output_size);
for i = 1:num_samples
   trainLabelsOneHot(i, trainLabels(i) + 1) = 1;  % +1 to match MATLAB 1-based indexing
end
```

Figure 5: One hot encoding

The for loop iterates over all samples (`i = 1:num samples`), so each sample's label can be converted to a one-hot encoded format. `trainLabels(i)` gives the class label for the i-th sample. For CIFAR-10, these labels would be integers between 0 and 9. `trainLabels(i) + 1` adjusts the label from 0-based to 1-based indexing, as in MATLAB array indices start at 1. `trainLabelsOneHot(i, trainLabels(i) + 1) = 1` sets the element in the one-hot encoded matrix for the current sample to 1 at the position representing its class. All other positions in the row remain 0.

1.3 Forward function

FORWARD PASS THROUGH THE FIRST LAYER

`X * weights1` computes the weighted sum of the inputs and the weights for each neuron in the first hidden layer. This results in a matrix of dimensions `[batch size, hidden size1]`, where hidden size1 is the number of neurons in the first hidden layer. Biases1 adds the bias term to each neuron in the first hidden layer. Biases are broadcasted across the batch (sincebiases1 has dimensions `[1, hidden Size1]`). The sigmoid function is applied to the row outputs producing the activated outputs for the first hidden layer, stored in activations1. The dimensions remain `[batch size, hidden size1]`.

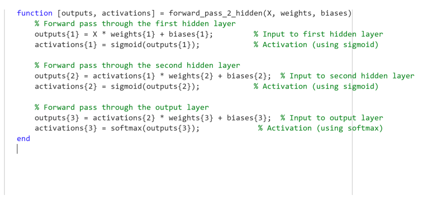
FORWARD PASS THROUGH THE SECOND LAYER

Having the same steps as before, `activations1 * weights2` computes the weighted sum of the activations from the first hidden layer and the weights for the second hidden layer. This results in a matrix of dimensions `[batch size, hidden size2]`, where hidden size2 is the number of neurons in the second hidden layer. Biases2 adds the bias term for the second hidden layer, broadcasting across the batch. The sigmoid function is again applied to introduce non-linearity, resulting in activated outputs for the second hidden layer stored in `activations2`. Dimensions remain `[batch size, hidden size2]`.

FORWARD PASS THROUGH THE THIRD LAYER

`Activations2 * weights3` computes the weighted sum of the activations from the second hidden layer and the weights for the output layer. This results in a matrix of dimensions [batch size, output size], where output size is the number of neurons in the output layer (e.g., the number of classes for classification). `Biases3` adds the bias term for the output layer, broadcasting across the batch. The softmax function is applied to the raw outputs of the output layer to produce a probability distribution over the output classes, ensuring that the output activations (stored in `activations3`) are probabilities that sum to 1 for each sample in the batch. Dimensions remain `[batch size, output size]`.

In the case of one hidden layer, the procedure would be the same but without the second step and in the third pass, the multiplication would be activations1 * weights2 and biases2.



```
function [outputs, activations] = forward_pass_2_hidden(X, weights, biases)
    % Forward pass through the first hidden layer
    outputs{1} = X * weights{1} + biases{1};         % Input to first hidden layer
    activations{1} = sigmoid(outputs{1});            % Activation (using sigmoid)

    % Forward pass through the second hidden layer
    outputs{2} = activations{1} * weights{2} + biases{2};  % Input to second hidden layer
    activations{2} = sigmoid(outputs{2});            % Activation (using sigmoid)

    % Forward pass through the output layer
    outputs{3} = activations{2} * weights{3} + biases{3};  % Input to output layer
    activations{3} = softmax(outputs{3});             % Activation (using softmax)
end
```

Figure 6: Forward Pass for 2 hidden layers

1.4 Backward function

The backward pass propagates the error from the output layer back to the input layer, layer by layer. Two cell arrays, d weights and d biases, are initialized to store the gradients of the weights and biases for each layer. The backward pass starts from the output layer, where the error can be directly measured as the difference between the predicted outputs and the true labels, because we used softmax in the output layer, otherwise it would be multiplied with the sigmoid derivative. This error is propagated backward through the network,

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \qquad (4.13)$$

where the *local gradient* $\delta_j(n)$ is defined by

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \qquad (4.14) \\ &= e_j(n) \varphi_j'(v_j(n))\end{aligned}$$

Figure 7: delta calculation-output layer, Haykin's book

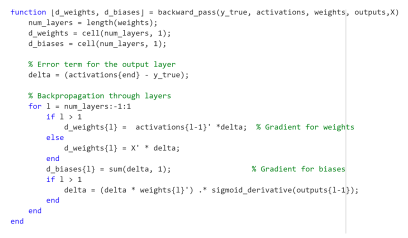
layer by layer, adjusting for the influence of each layer on the final output. For each internal layer, the weight gradients are computed as the product of the activations from the previous layer/the input data and the deltas of the current layer. Bias gradients are computed by summing the deltas across the mini-batch. Furthermore, in the end we compute the delta for using it in the next internal layer, following the formula from Haykin's book:

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}$$

Figure 8: delta calculation for internal layer, Haykin's book

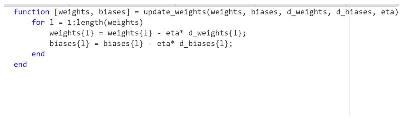
1.5 Updating weights

After calculating the d-weights from back propagation function and divide them with the sample size, we call this function to update the initial weights, subtracting from them the d-weights multiplied with the learning rate.



```
function [d_weights, d_biases] = backward_pass(y_true, activations, weights, outputs,X)
    num_layers = length(weights);
    d_weights = cell(num_layers, 1);
    d_biases = cell(num_layers, 1);

    % Error term for the output layer
    delta = (activations{end} - y_true);

    % Backpropagation through layers
    for l = num_layers:-1:1
        if l > 1
            d_weights{l} =  activations{l-1}' *delta;  % Gradient for weights
        else
            d_weights{l} = X' * delta;
        end
        d_biases{l} = sum(delta, 1);                  % Gradient for biases
        if l > 1
            delta = (delta * weights{l}') .* sigmoid_derivative(outputs{l-1});
        end
    end
end
```

Figure 9: Backward function

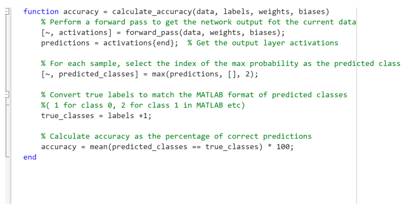


```
function [weights, biases] = update_weights(weights, biases, d_weights, d_biases, eta)
    for l = 1:length(weights)
        weights{l} = weights{l} - eta* d_weights{l};
        biases{l} = biases{l} - eta* d_biases{l};
    end
end
```

Figure 10: Update weights

1.6 Calculate accuracy

This function calculates the accuracy that has the model in the input data. First, it calculates through a forward pass the output of the last layer, activations end. To convert this to a prediction, it calculates the index of the maximum value of the layer-10 neurons-, as we used softmax and it can be easily converted to probability. Then, we convert the input layer to MATLAB indexes, index of the start of the matrix = 1. And then we calculate how many times in the whole sample,we predicted well, and thus the accuracy is ready.



```
function accuracy = calculate_accuracy(data, labels, weights, biases)
    % Perform a forward pass to get the network output fot the current data
    [~, activations] = forward_pass(data, weights, biases);
    predictions = activations{end};  % Get the output layer activations

    % For each sample, select the index of the max probability as the predicted class
    [~, predicted_classes] = max(predictions, [], 2);

    % Convert true labels to match the MATLAB format of predicted classes
    %( 1 for class 0, 2 for class 1 in MATLAB etc)
    true_classes = labels +1;

    % Calculate accuracy as the percentage of correct predictions
    accuracy = mean(predicted_classes == true_classes) * 100;
end
```

Figure 11: Calculate accuracy function

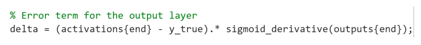
2 Errors during the training

First of all, in the start of the process, after writing all the functions I tried testing my code with 1000,5000 and 10000 samples from the first batch of the CIFAR-10 library as a training set, 1 hidden layer with 128/256 neurons and as an activation function sigmoid function in all layers. I tried running it with learning rate 0.01, 0.001 and 0.0001 and the model was not trained at all, as both training and validation accuracies were around 10. Having a learning rate

0.00001, the model started performing better reaching a training accuracy 16.10 and 14.98 at 40 epochs. I was so confused why the model didn't run at all at a more common learning rate, but I couldn't find my error. I got some advice from Chat-gpt and it suggested me changing the initialization of weights, for them to have the right variance and using Softmax as an activation function in the output layer.

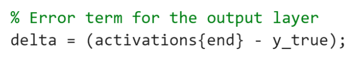
For the initialization of the weights I used He weight matrix where each layer is initialized with random values from a normal distribution, scaled by a factor that depends on the number of neurons in the layer. Scaling by sqrt(2 / (fan in + fan out)) helps keep the variance of the activations across layers under control, which prevents the values from getting too large or too small as they propagate through the network. In a neural network, weight initialization plays a crucial role in how quickly and effectively the network learns. Poor initialization can lead to problems such as vanishing or exploding gradients, which make training unstable or slow. This initialization is a modified form of He Initialization,as the scaling factor here is sqrt(2 / (fan in + fan out)),common in networks where activation functions are not strictly ReLU. Without this scaling, activations can grow or shrink rapidly, causing gradients to vanish (become very small) or explode (become very large) during back propagation. Properly scaled initialization helps the network learn faster by keeping the initial activations within a reasonable range, reducing the time needed for the model to reach effective learning.

Implementing all these, I observed a much better behavior while having 0.00001 as a learning rate . For example with 128 neurons in the hidden layer in 40 epochs I got 30 validation accuracy. However there were not significant changes for the bigger rates. Another mistake that I did is that I didn't use batches and I trained for example 1000 samples at once. Looking deeper in my equations in the back propagation algorithm , while doing the math following the function in the paper, I observed the way that $\Delta\omega$ is calculated. When updating the weights, I didn't divide with the batch size so, as a result I needed an hta-learning rate so small that when using it was like I am doing the division.



```
% Error term for the output layer
delta = (activations{end} - y_true).* sigmoid_derivative(outputs{end});
```

Figure 12: Wrong delta formula while using softmax in the output layer



```
% Error term for the output layer
delta = (activations{end} - y_true);
```

Figure 13: Right delta formula for softmax

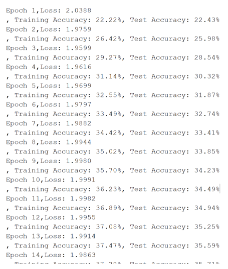
Lastly, after some training I found a mistake in the delta calculation while using softmax. If the output layer uses softmax, then using the sigmoid derivative in the output layer's backpropagation step is not appropriate. The gradient of the cross-entropy loss with respect to the softmax inputs is simply the difference between the predicted probability and the true label.

3 Experimenting with the parameters

3.1 Comparing Learning Rates

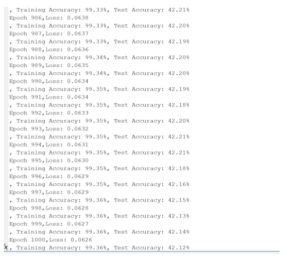
A high learning rate like 0.1 can help the model make large initial updates, which might speed up the learning a lot in the early stages of training. However, using a high learning rate can be hurtful as it hides the possibility of overpassing the correct point, as the weights are not updated slowly-overshoot optimal weights-.

To compare different rates, I used one hidden layer with 128 neurons and batch size 256. When I applied a 0.1 learning rate, I observed that in the start the accuracy had a good progress, but the cross entropy error is unstable. Then, the model overfits quickly, as the training accuracy increases a lot , the validation accuracy is staying almost constant.



```
Epoch 1,Loss: 2.0388
, Training Accuracy: 22.22%, Test Accuracy: 22.43%
Epoch 2,Loss: 1.9759
, Training Accuracy: 26.42%, Test Accuracy: 25.98%
Epoch 3,Loss: 1.9599
, Training Accuracy: 29.27%, Test Accuracy: 28.54%
Epoch 4,Loss: 1.9616
, Training Accuracy: 31.14%, Test Accuracy: 30.32%
Epoch 5,Loss: 1.9699
, Training Accuracy: 32.55%, Test Accuracy: 31.87%
Epoch 6,Loss: 1.9797
, Training Accuracy: 33.49%, Test Accuracy: 32.74%
Epoch 7,Loss: 1.9882
, Training Accuracy: 34.42%, Test Accuracy: 33.41%
Epoch 8,Loss: 1.9944
, Training Accuracy: 35.02%, Test Accuracy: 33.85%
Epoch 9,Loss: 1.9980
, Training Accuracy: 35.70%, Test Accuracy: 34.23%
Epoch 10,Loss: 1.9991
, Training Accuracy: 36.23%, Test Accuracy: 34.49%
Epoch 11,Loss: 1.9982
, Training Accuracy: 36.89%, Test Accuracy: 34.94%
Epoch 12,Loss: 1.9955
, Training Accuracy: 37.08%, Test Accuracy: 35.25%
Epoch 13,Loss: 1.9914
, Training Accuracy: 37.47%, Test Accuracy: 35.59%
Epoch 14,Loss: 1.9863
```

Figure 14: Learning Rate = 0.1 start of the training



```
, Training Accuracy: 99.33%, Test Accuracy: 42.21%
Epoch 986,Loss: 0.0638
, Training Accuracy: 99.33%, Test Accuracy: 42.20%
Epoch 987,Loss: 0.0637
, Training Accuracy: 99.33%, Test Accuracy: 42.19%
Epoch 988,Loss: 0.0636
, Training Accuracy: 99.34%, Test Accuracy: 42.20%
Epoch 989,Loss: 0.0635
, Training Accuracy: 99.34%, Test Accuracy: 42.20%
Epoch 990,Loss: 0.0634
, Training Accuracy: 99.35%, Test Accuracy: 42.19%
Epoch 991,Loss: 0.0634
, Training Accuracy: 99.35%, Test Accuracy: 42.18%
Epoch 992,Loss: 0.0633
, Training Accuracy: 99.35%, Test Accuracy: 42.20%
Epoch 993,Loss: 0.0632
, Training Accuracy: 99.35%, Test Accuracy: 42.21%
Epoch 994,Loss: 0.0631
, Training Accuracy: 99.35%, Test Accuracy: 42.21%
Epoch 995,Loss: 0.0630
, Training Accuracy: 99.35%, Test Accuracy: 42.18%
Epoch 996,Loss: 0.0629
, Training Accuracy: 99.35%, Test Accuracy: 42.16%
Epoch 997,Loss: 0.0629
, Training Accuracy: 99.36%, Test Accuracy: 42.15%
Epoch 998,Loss: 0.0628
, Training Accuracy: 99.36%, Test Accuracy: 42.13%
Epoch 999,Loss: 0.0627
, Training Accuracy: 99.36%, Test Accuracy: 42.14%
Epoch 1000,Loss: 0.0626
, Training Accuracy: 99.36%, Test Accuracy: 42.12%
```

Figure 15: Clear Overfit

Learning rate=0.01

Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	1.95	35.60	34.60
100	1.97	38.67	37.15
200	1.89	41.25	39.10
300	1.79	43.30	40.21
400	1.70	45.37	40.62
500	1.63	47.22	41.53
600	1.56	49.00	42.18
700	1.50	50.79	42.88
800	1.43	52.40	42.97
900	1.37	54.14	43.24
1000	1.30	55.79	43.57
1200	1.18	58.22	44.21

Reducing the learning rate to 0.01 allowed for more gradual and stable learning, reflected in smoother convergence and better final test accuracy. For a learning rate of 0.01, the training accuracy steadily increased over epochs, while the test accuracy plateaued. This indicates overfitting is occurring after certain epochs.

LEARNING RATE=0.001

Epoch	Training Accuracy (%)	Test Accuracy (%)
40	23.71	23.90
100	29.24	28.86
200	32.31	31.78
300	34.26	33.42
500	36.32	35.32
700	37.40	36.22
800	38.08	36.52
900	38.45	36.79
1000	38.87	37.13
1200	39.50	37.70

We observe that with the lower learning rate the NN converges slower, needs more epochs to reach the desirable validation accuracy.

3.2 Comparing Batch sizes

LEARNING WITH 8000 TRAINING AND 2000 TEST SET

Learning Rate=0.01, 1 hidden layer = 128 neurons

BATCH SIZE 32

Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	1.55	42.23	37.75
100	1.43	50.11	40.50
200	1.22	60.00	43.30
300	1.00	70.16	42.70

It reaches 43% earlier and overfits earlier, as it logical
BATCH SIZE 64

Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	1.72	39.06	34.60
100	1.61	43.92	38.85
200	1.46	50.66	40.80
300	1.35	56.23	42.70
500	1.15	66.85	43.05
650	0.99	73.98	42.90

I stopped the training at the start of the overfitting.
BATCH SIZE 128

Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	1.79	37.06	35.80
100	1.70	40.58	37.70
200	1.61	44.35	39.60
300	1.53	47.85	40.55
500	1.38	53.94	41.70
600	1.32	56.54	42.80
700	1.27	59.30	43.50
800	1.22	62.25	43.05

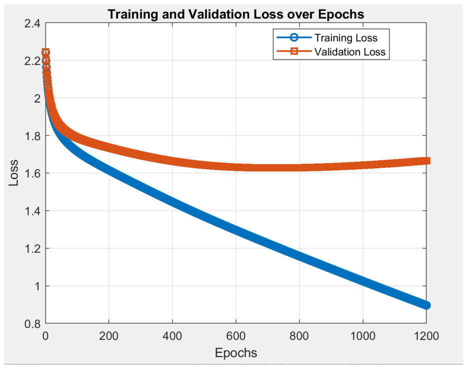
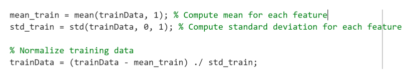


Figure 16: Training vs Validation Loss, 1 hidden Layer, 128 neurons, Batch size=128



```
mean_train = mean(trainData, 1); % Compute mean for each feature
std_train = std(trainData, 0, 1); % Compute standard deviation for each feature

% Normalize training data
trainData = (trainData - mean_train) ./ std_train;
```

Figure 17: Normalising the data

BATCH SIZE 256

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.88	34.54	33.55
100	1.76	37.80	36.50
200	1.70	40.81	37.60
300	1.66	42.29	38.80
500	1.58	46.38	40.40
700	1.49	49.75	41.10
1000	1.38	54.14	41.65
1200	1.32	56.95	42.85

CONCLUSIONS-OBSERVATIONS

When I decreased the batch size, each epoch needed more time to be executed-as it divides the dataset in smaller group and the iterations become more, but the needed number of epochs for reaching the maximum accuracy was reduced, so the whole execution time was similar. Also, the overfitting was happening in different number of epochs and it was more intense when the batch sizes were small. Comparing the results, a batch size of 128 achieved the best balance, yielding steady improvements in both training and test accuracy before and after overfitting began.

3.3 Comparing number of layers

I experimented with one and two hidden layers and I noticed that I do not get so different accuracies. When I put the whole sample, with two hidden layers I reach 2-3% more.

3.4 Normalization

Normalization involves rescaling the input data so that each feature has a mean of 0 and a standard deviation of 1 (standard normalization) or scales the data to a range such as [0, 1].

In the code I computed the mean (mean train) and standard deviation (std train) for each column (feature) of trainData, and then subtract the mean and divide by the standard deviation for each element in the corresponding column to standardize the values, ensuring each feature has a mean of 0 and standard deviation of 1. I handled the test data with mean train and std train as well,to evaluate the model's performance and ensure that it lies in the same range and scale as the training data.

Normalization ensures that the training process is stable by preventing features with large numerical ranges from dominating the gradient updates. It accelerates the convergence of optimization algorithms (e.g., gradient descent) by keeping the data values within a consistent range. As a result, it reduces the chances of exploding or vanishing gradients by ensuring inputs to layers are well-scaled.

1 layer 128 batch=128 learning rate=0.01 with normalization

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.61	46.74	39.55
100	1.43	55.38	41.00
150	1.39	61.94	41.15
200	1.17	67.71	40.70

It reaches a high validation accuracy earlier, however without normalization we saw before that for the same parameters we reached 45%.

64-16 without normalization

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	2.07	26.50	26.12
100	1.98	31.67	30.24
200	1.97	36.14	35.09
300	1.91	39.41	37.91
400	1.80	42.32	39.51
500	1.64	45.17	40.91
600	1.46	48.05	41.97
800	1.16	53.67	42.83
1000	0.92	59.62	43.14
1200	0.73	65.96	42.91

64-16 with normalization

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.93	33.90	33.00
100	1.80	40.10	37.36
200	1.50	47.85	40.13
300	1.16	55.45	40.14

It is clear that when we normalise the data the model converges a lot faster, it needs fewer epochs and at the same time its execution time was less than before. However, without regularization we reached bigger validation accuracies and the model overfitted slower.

512-256 with normalization

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.91	39.17	37.82
100	1.67	44.96	40.25
200	1.29	53.99	41.88
300	0.88	67.17	42.54
400	0.47	83.87	41.95

Normalizing the data could have amplified certain patterns or noise in the data. This makes the model more prone to overfitting, as it may focus excessively on the smaller variations that normalization introduced. If the model capacity (e.g., the number of parameters, complexity) is high, it may overfit to even these subtle patterns in the data.

3.5 Comparing number of neurons

1 HIDDEN LAYER

First of all, I compared the number of neurons in a NN with only one hidden layer. In the table below, the training and test accuracies of different models with a hidden layer of 32, 64, 128 and 512 are presented.

Epochs	Train 32	Test 32	Train 64	Test 64	Train 128	Test 128	Train 512	Test 512
40	35.15	33.80	37.54	35.60	37.06	35.8	37.55	36.30
100	38.41	36.65	41.11	38.25	40.58	37.70	40.99	37.55
300	45.35	39.95	47.79	40.70	47.85	40.55	48.09	39.40
400	48.24	40.25	50.69	41.90			51	39.75
600	57.65	41.25	56.32	42.25	56.54	42.80		
800	58.27	40.90	60.79	43.30	62.25	43.05		
1000			65.41	44.40				

From the figure above it is evident that models with more neurons, such as 512 neurons, take significantly longer to converge compared to those with fewer neurons and the algorithm gets a lot more slower in execution time/epoch too. I tested with 1024 neurons too and the algorithm was so slow , that I decided to end the execution even earlier than with the 512 neurons. Moreover, increasing the number of neurons leads to higher training accuracy across epochs, especially in models with 128 and 512 neurons. More neurons in the hidden layer allow the network to model more complex relationships in the data. This added capacity enables the model to fit the training data better, leading to higher training accuracy. For smaller rates, we observe better accuracies at earlier epochs, but they overfit and don't reach the desirable accuracy.

Models with 64 and 128 neurons achieve good accuracy early and in general while maintaining a manageable training time. These models strike a balance between capacity and generalization. They are large enough to learn useful features but not so large that they overfit or converge too slowly. All in all, the results suggest that the CIFAR-10 dataset, with its moderate complexity, does not require excessively large networks for good performance. Beyond a certain size, adding neurons only increases computational costs without significantly boosting accuracy.

2 HIDDEN LAYERS

512-128 , learning rate = 0.01 , training set=8000, test set=2000

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	2.10	26.77	40.25
100	2.02	32.41	32.16
200	1.87	36.82	36.41
300	1.73	39.89	38.40
400	1.63	42.42	39.79
500	1.52	45.22	40.78
600	1.38	47.56	41.56
700	0.97	50.25	42.43
1000	0.85	59.63	45.05
1100	0.79	83.87	45.34

128-64

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.90	37.21	35.59
100	1.63	43.67	39.57
200	1.27	51.39	41.52
300	0.97	60.53	42.20
400	0.69	71.87	41.99

1024-512

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.89	40.08	37.63
100	1.63	45.62	40.12
200	1.21	54.95	42.26
300	0.75	68.61	42.61

We observe that with more neurons in the layers the model becomes a lot more slower in the execution time as it is more complex and at the same time it doesn't offer a significant change in the accuracy, which is even slightly smaller than other combinations. When we used less number of neurons 128-64, the execution time was small, but the model didn't have the highest accuracy.

To conclude the best combination of neurons in the hidden layers that reached the biggest validation accuracy was 512 - 128.

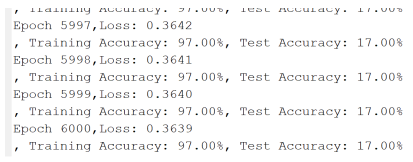
4 Overfitting

A model trained on CIFAR-10 images can easily memorize every detail in its training images, like specific lighting conditions or image noise. If you show it a new image, it may fail to recognize it correctly because it hasn't learned general characteristics that apply across similar images; instead, it learned specifics unique to the training set. Instead of learning useful general patterns, the model ”memorizes” training examples, making it less effective at generalizing. As a result, the model's performance on new data is often much worse than on the training data.

When a model is overfitting, it performs well on the training data but poorly on new, unseen data (such as the test or validation set), so, the model's training accuracy keeps improving, but test accuracy plateaus or even starts decreasing.

As overfitted models tend to make inconsistent predictions , they are not reliable. Furthermore, a model that learns unnecessary detail may require more memory and computational power without offering real benefits.

I wanted to observe the extreme over-fitting, checking as well that my back propagation algorithm performs well, so I tried my model with 100 samples as a training set ,100 as test set as well. It is clear that the model is learning the characteristics of the samples, so it performs perfectly on the known set, but when the input is an unknown photo it doesn't perform well,even worse than the other trials. The last thing happens because when you train a model with more representatives , it is trained better.



```
, Training Accuracy: 97.00%, Test Accuracy: 17.00%
Epoch 5997,Loss: 0.3642
, Training Accuracy: 97.00%, Test Accuracy: 17.00%
Epoch 5998,Loss: 0.3641
, Training Accuracy: 97.00%, Test Accuracy: 17.00%
Epoch 5999,Loss: 0.3640
, Training Accuracy: 97.00%, Test Accuracy: 17.00%
Epoch 6000,Loss: 0.3639
, Training Accuracy: 97.00%, Test Accuracy: 17.00%
```

Figure 18: Extreme over-fitting

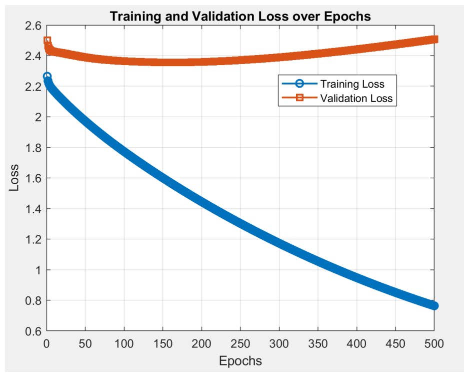
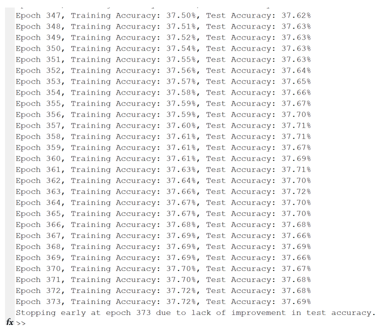


Figure 19: Test and train losses vs epochs

In the figure above, this phenomenon is demonstrated. In the start, both losses are getting reduced, but after some epochs the test loss starts to increase, while train loss decreases with larger rate.

In general, during the training of my model I didn't stop the execution when the overfitting started (10% difference in accuracies, but when we reached the maximum validation accuracy and saw that it starts to degrade. However, because as we train the model with different combination and techniques sometimes it is getting really slow and the accuracies increase slowly, we can stop the training when we see that after some epochs we don't have any difference in test performance. This technique is called early stopping.



```
Epoch 347, Training Accuracy: 37.50%, Test Accuracy: 37.62%
Epoch 348, Training Accuracy: 37.51%, Test Accuracy: 37.63%
Epoch 349, Training Accuracy: 37.52%, Test Accuracy: 37.63%
Epoch 350, Training Accuracy: 37.54%, Test Accuracy: 37.63%
Epoch 351, Training Accuracy: 37.55%, Test Accuracy: 37.63%
Epoch 352, Training Accuracy: 37.56%, Test Accuracy: 37.64%
Epoch 353, Training Accuracy: 37.57%, Test Accuracy: 37.65%
Epoch 354, Training Accuracy: 37.58%, Test Accuracy: 37.66%
Epoch 355, Training Accuracy: 37.59%, Test Accuracy: 37.67%
Epoch 356, Training Accuracy: 37.59%, Test Accuracy: 37.70%
Epoch 357, Training Accuracy: 37.60%, Test Accuracy: 37.71%
Epoch 358, Training Accuracy: 37.61%, Test Accuracy: 37.71%
Epoch 359, Training Accuracy: 37.61%, Test Accuracy: 37.67%
Epoch 360, Training Accuracy: 37.61%, Test Accuracy: 37.69%
Epoch 361, Training Accuracy: 37.63%, Test Accuracy: 37.71%
Epoch 362, Training Accuracy: 37.64%, Test Accuracy: 37.70%
Epoch 363, Training Accuracy: 37.66%, Test Accuracy: 37.72%
Epoch 364, Training Accuracy: 37.67%, Test Accuracy: 37.70%
Epoch 365, Training Accuracy: 37.67%, Test Accuracy: 37.70%
Epoch 366, Training Accuracy: 37.68%, Test Accuracy: 37.68%
Epoch 367, Training Accuracy: 37.69%, Test Accuracy: 37.66%
Epoch 368, Training Accuracy: 37.69%, Test Accuracy: 37.69%
Epoch 369, Training Accuracy: 37.69%, Test Accuracy: 37.66%
Epoch 370, Training Accuracy: 37.70%, Test Accuracy: 37.67%
Epoch 371, Training Accuracy: 37.70%, Test Accuracy: 37.68%
Epoch 372, Training Accuracy: 37.72%, Test Accuracy: 37.68%
Epoch 373, Training Accuracy: 37.72%, Test Accuracy: 37.69%
Stopping early at epoch 373 due to lack of improvement in test accuracy.
fx >>
```

Figure 20: early stopping

5 Prevent overfitting techniques

5.1 L2 Regulator

Method Description

L2 regularization is a technique used to prevent overfitting in neural networks by adding a penalty term to the loss function. This penalty discourages the network from assigning large weights to any particular feature, promoting generalization. Specifically, L2 regularization adds a term proportional to the sum of the squared weights to the loss function:

$$\text{Loss}_{\text{L2}} = \text{Loss}_{\text{original}} + \frac{\lambda}{2} \sum_i w_i^2$$

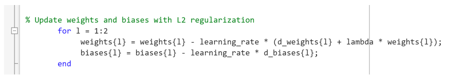
where λ is the regularization parameter that controls the strength of the penalty, and ω_i are the weights of the model. Larger values of λencourage smaller weights but can lead to underfitting, while smaller values allow more flexibility but can increase the risk of overfitting.

Code description

In the training code, L2 regularization was implemented by modifying the weight update rule. The gradients of the L2 penalty with respect to the weights were added to the gradients of the loss function, effectively penalizing large weights. We didn't apply this rule to the biases. The adjusted weight update rule is as follows:

$$w_i = w_i - \eta \left(\frac{\partial \text{Loss}}{\partial w_i} + \lambda w_i \right)$$

where η is the learning rate. This adjustment was incorporated into the code after backward pass function while updating the weights, and the regularization term was included in the total loss calculation.



```
% Update weights and biases with L2 regularization
    for l = 1:2
        weights{l} = weights{l} - learning_rate * (d_weights{l} + lambda * weights{l});
        biases{l} = biases{l} - learning_rate * d_biases{l};
    end
```

Figure 21: L2 update weights

BATCH SIZE 128 lambda = 0.01

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	2.69	32.05	32.45
100	2.25	35.80	34.95
200	1.99	37.54	37.15
500	1.89	39.67	38.35

With $\lambda = 0.01$, the effect of regularization was significant, causing a slow augmentation in training accuracy and a smaller gap between training and test accuracy. This indicates that overfitting was effectively mitigated, but at the cost of slower training progress. So, for a better accuracy, I decided to reduce the penalty

BATCH SIZE 128 lambda=0.001

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.98	33.44	33.60
100	1.88	37.69	37.05
200	1.79	41.51	38.10
300	1.73	44.52	40.25
400	1.67	47.20	40.95
500	1.62	49.56	41.20
800	1.54	53.67	42.50
900	1.48	57.46	43.80
1000	1.48	65.96	43.60

By reducing λ to 0.001, the model showed faster improvement in training accuracy and lower loss. At 500 epochs, the test accuracy reached 41.20, and training accuracy was 49.56. This setting achieved higher accuracy while still preventing significant overfitting, as evidenced by the relatively small gap between training and test accuracies. Beyond 800 epochs, the model began to overfit, with the training accuracy reaching 65.96 and test accuracy plateauing at 43.60. This suggests that, while smaller λ allowed better learning, prolonged training may still lead to overfitting.

BATCH SIZE 32 lambda=0.001 8000 training set - 2000 test set

Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	1.69	39.73	37.35
100	1.62	46.38	40.20
200	1.54	50.79	41.25
300	1.48	54.12	42.80
500	1.44	58.89	44.15
650	1.38	62.24	44.30

Without this technique the maximum that I could reach 43.25 , but with training accuracy 70.17, but earlier in 300 epoch. So,it is clear that we can reach a better accuracy but in more time, as the rate that the training accuracy is increased and the loss is decreased is much more smaller to prevent the overfitting. A value of λ like 0.001 strikes a good balance between training progress and regularization, leading to improved test accuracy compared to no regularization.

Now I tried implementing this method in all the sample.

2 LAYERS 512-128 BATCH SIZE lambda=0.001 50.000 training set - 10.000 test set

Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	2.36	39.53	39.87
100	2.17	42.64	41.99
200	2.14	44.42	43.60
300	2.14	45.31	45.60
500	2.14	46.27	46.50
1500	2.16	46.27	46.50

I decided to reduce lambda to 0.0001, because the penalty was too big as we barely had a progress in the training accuracy.

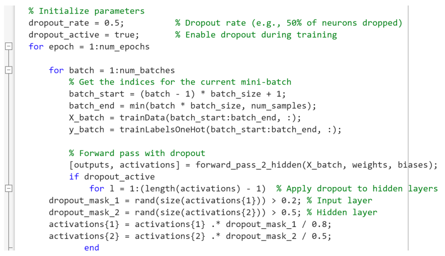
Epoch	Training Loss	Training Accuracy (%)	Test Accuracy (%)
40	2.15	41.12	40.85
100	1.99	46.27	45.09
200	1.88	51.65	49.19
300	1.78	55.76	51.62
400	1.71	59.22	53.08
500	1.64	62.18	54.17
600	1.59	65.15	54.76
800	1.55	67.72	55.14
1000	1.51	76.17	55.47
1100	1.50	76.68	55.74
1300	1.50	79.22	55.93

I stopped the simulation, before the overfitting as the simulation had a slow execution time.

5.2 Drop out

Dropout is a regularization technique to prevent overfitting. During training, dropout randomly ”drops” (sets to zero) a proportion of neurons in the network, which forces the model to learn more robust and generalized features by not relying heavily on any single neuron. By scaling the remaining activations (with division by the keep probability, e.g., 0.8 or 0.5), dropout maintains the expected value of the activations during training.

Dropout is applied conditionally (if dropout active is true). The `dropout mask 1` and `dropout mask 1` are binary masks created for the activations of two layers, with probabilities of keeping neurons at 80% and 50%, respectively. 512 -128 drop out + L2 ,learning rate=0.01 lambda=0.0001



```
% Initialize parameters
dropout_rate = 0.5;          % Dropout rate (e.g., 50% of neurons dropped)
dropout_active = true;       % Enable dropout during training
for epoch = 1:num_epochs

    for batch = 1:num_batches
        % Get the indices for the current mini-batch
        batch_start = (batch - 1) * batch_size + 1;
        batch_end = min(batch * batch_size, num_samples);
        X_batch = trainData(batch_start:batch_end, :);
        y_batch = trainLabelsOneHot(batch_start:batch_end, :);

        % Forward pass with dropout
        [outputs, activations] = forward_pass_2_hidden(X_batch, weights, biases);
        if dropout_active
            for l = 1:(length(activations) - 1)  % Apply dropout to hidden layers
    dropout_mask_1 = rand(size(activations{1})) > 0.2; % Input layer
    dropout_mask_2 = rand(size(activations{2})) > 0.5; % Hidden layer
    activations{1} = activations{1} .* dropout_mask_1 / 0.8;
    activations{2} = activations{2} .* dropout_mask_2 / 0.5;
           end
```

Figure 22: Drop out

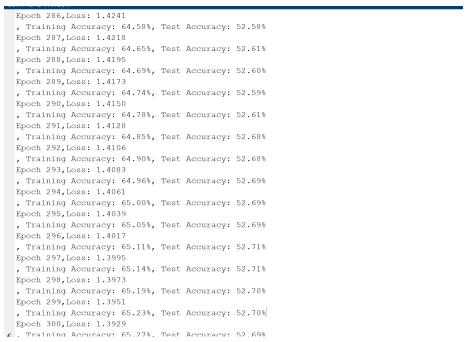
Epoch	Training Accuracy (%)	Test Accuracy (%)
40	41.14	40.88
100	46.32	45.20
200	51.71	49.10
300	55.82	51.60
500	62.33	54.36
600	65.15	54.74
700	67.78	55.15
900	72.31	55.51
1000	74.36	55.65
1100	76.17	55.81

When the L2 regulator is combined with dropout, the results are similar with the ones where drop out was not used. The model's capacity and dataset complexity might not have required the additional regularization provided by dropout. Otherwise, the dropout rates or L2 penalty might not have been optimized, leading to redundant or ineffective regularization.

6 Training with the whole dataset 50.000

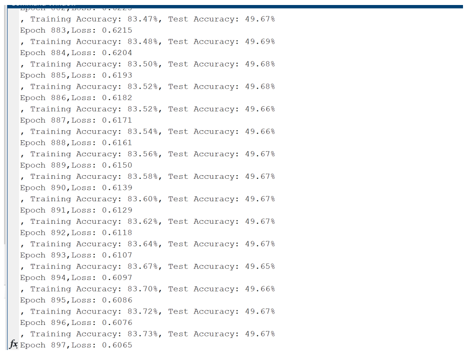
1 LAYER

Learning rate=0.01, batch size=64, 1 layer, 128 neurons



```
Epoch 286,Loss: 1.4241
, Training Accuracy: 64.58%, Test Accuracy: 52.58%
Epoch 287,Loss: 1.4218
, Training Accuracy: 64.65%, Test Accuracy: 52.61%
Epoch 288,Loss: 1.4195
, Training Accuracy: 64.69%, Test Accuracy: 52.60%
Epoch 289,Loss: 1.4173
, Training Accuracy: 64.74%, Test Accuracy: 52.59%
Epoch 290,Loss: 1.4150
, Training Accuracy: 64.78%, Test Accuracy: 52.61%
Epoch 291,Loss: 1.4128
, Training Accuracy: 64.85%, Test Accuracy: 52.68%
Epoch 292,Loss: 1.4106
, Training Accuracy: 64.90%, Test Accuracy: 52.68%
Epoch 293,Loss: 1.4083
, Training Accuracy: 64.96%, Test Accuracy: 52.69%
Epoch 294,Loss: 1.4061
, Training Accuracy: 65.00%, Test Accuracy: 52.69%
Epoch 295,Loss: 1.4039
, Training Accuracy: 65.05%, Test Accuracy: 52.69%
Epoch 296,Loss: 1.4017
, Training Accuracy: 65.11%, Test Accuracy: 52.71%
Epoch 297,Loss: 1.3995
, Training Accuracy: 65.14%, Test Accuracy: 52.71%
Epoch 298,Loss: 1.3973
, Training Accuracy: 65.19%, Test Accuracy: 52.70%
Epoch 299,Loss: 1.3951
, Training Accuracy: 65.23%, Test Accuracy: 52.70%
Epoch 300,Loss: 1.3929
, Training Accuracy: 65.27%, Test Accuracy: 52.69%
```

Figure 23: 300 epochs accuracy = 52.70%



```
, Training Accuracy: 83.47%, Test Accuracy: 49.67%
Epoch 883,Loss: 0.6215
, Training Accuracy: 83.48%, Test Accuracy: 49.69%
Epoch 884,Loss: 0.6204
, Training Accuracy: 83.50%, Test Accuracy: 49.68%
Epoch 885,Loss: 0.6193
, Training Accuracy: 83.52%, Test Accuracy: 49.68%
Epoch 886,Loss: 0.6182
, Training Accuracy: 83.52%, Test Accuracy: 49.66%
Epoch 887,Loss: 0.6171
, Training Accuracy: 83.54%, Test Accuracy: 49.66%
Epoch 888,Loss: 0.6161
, Training Accuracy: 83.56%, Test Accuracy: 49.67%
Epoch 889,Loss: 0.6150
, Training Accuracy: 83.58%, Test Accuracy: 49.67%
Epoch 890,Loss: 0.6139
, Training Accuracy: 83.60%, Test Accuracy: 49.67%
Epoch 891,Loss: 0.6129
, Training Accuracy: 83.62%, Test Accuracy: 49.67%
Epoch 892,Loss: 0.6118
, Training Accuracy: 83.64%, Test Accuracy: 49.67%
Epoch 893,Loss: 0.6107
, Training Accuracy: 83.67%, Test Accuracy: 49.65%
Epoch 894,Loss: 0.6097
, Training Accuracy: 83.70%, Test Accuracy: 49.66%
Epoch 895,Loss: 0.6086
, Training Accuracy: 83.72%, Test Accuracy: 49.67%
Epoch 896,Loss: 0.6076
, Training Accuracy: 83.73%, Test Accuracy: 49.67%
Epoch 897,Loss: 0.6065
```

Figure 24: showing the overfit

2 LAYERS LEARNING RATE=0.01 NUMBER OF NEURONS IN EACH LAYER 516 AND 128 , BATCH SIZE = 128

Epoch	**Training Loss**	**Training Accuracy (%)**	**Test Accuracy (%)**
40	1.97	35.69	35.48
100	1.81	41.71	41.70
200	1.64	48.63	47.28
300	1.46	53.91	50.80
400	1.27	59.40	52.90
500	1.09	64.42	53.82
600	0.92	69.30	54.50
700	0.76	74.20	54.21
750	0.70	76.37	53.81

overfits after 600

With the whole dataset I reach a higher accuracy but the code runs much more slower, as expected. To reach 53 without regulator , 3 hours were needed

All in all, for the whole dataset when using a single hidden layer model can reach up to almost 53% validation accuracy after 300 epochs and then it overfits. When I used two layers without any extra techniques we reached 51% at 300 epochs and the maximum was 54.50% at 600 epochs and then it overfitted. Then, after implementing a L2 regulator I was able to increase it at 55.93 %and maybe even more, but I stopped the training because it was so slow, 6 hours were necessary for the 1300 epochs. The drop out technique combined with L2 regulator had great results too, reaching 55.81 at 1100 epochs.(really slow training too)

7 Comparison NN with other ML techniques

Some comments about the other techniques:

Initially, the methods of one and three nearest neighbors are significantly more time-consuming than that of the nearest class center. This is because, in the first two methods, the distance of the test sample is calculated against all the elements of the training set, amounting to 50,000 computations. However, the latter method requires significantly fewer computations since it only compares the sample to the ten class centers. As the number of neighbors increases, the execution time also increases due to the sorting function and the array that is generated.

For this reason, when I was testing the code for the K-Nearest-Neighbors, I initially ran it with only one batch. It was observed, as expected, that the performance improved when the total 50,000 comparisons were made and not batch-wise (when the training set consisted of only the first batch, the performance was 28.77% for 1-nearest-neighbor and 27.85% for 3-nearest-neighbors).

The most efficient algorithm proved to be the 1-Nearest-Neighbor method with 35.39% accuracy, followed by the 3-Nearest-Neighbors with 33.03%, while the Nearest Centroid achieved 27.79%. (Although it excels in execution time, it lags in accuracy).

Comparing our neural network with the Nearest Neighbor Approximation and Nearest Centroid, the gain in the accuracy is evident, approximately 20% difference. This result is logical as neural networks learn abstract features through hidden layers, which often allow for better generalization across diverse datasets compared to nearest neighbor methods that rely purely on raw distance metrics. Also, it is important to note that although the model needs more time to finally converge in its best accuracy than the other techniques, if we wanted it to reach only 27-35% it would be much faster than them, as it can reach this accuracy in early epochs, in some minutes/ seconds depending on the sample. Last but not least, while nearest neighbors might work efficiently with small datasets, the computational cost becomes prohibitive as the dataset size increases. Neural networks, on the other hand, can handle larger datasets more efficiently once trained.