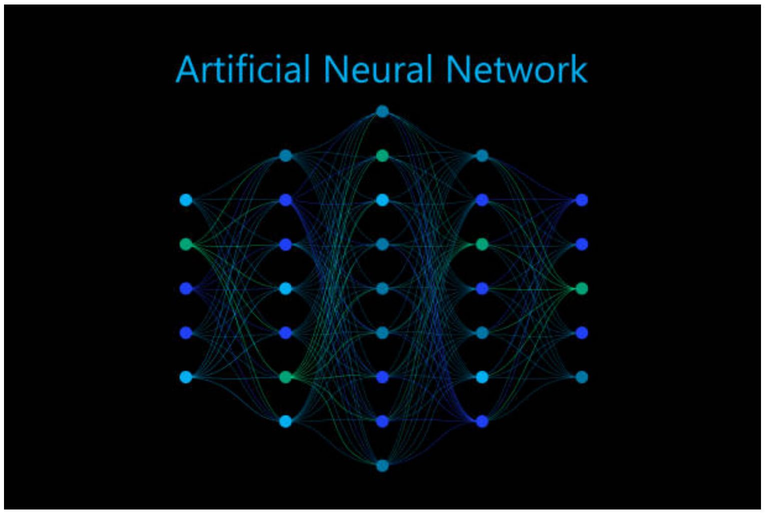
Neural Networks

Comparison between Multilayer perception, k-nearest Neighbour and Nearest Class Centroid

Iasonas Kakandris



A report presented for the course of
Neural Networks: Deep learning

Aristotle University of Thessaloniki
Faculty of Electrical and Computer Engineering
20/11/2023

Contents

1 Introduction

The goal of this project is the creation of a multilayer perceptron, able to identify correctly different classes of the CIFAR-10 dataset. More specifically the dataset consists of 60,000 32x32 color images (50,000 training and 10,000 test) from 10 different classes which are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The neural network must be able to identify correctly the images, have a lower loss (better accuracy) than the k-nearest Neighbour and Nearest Class Centroid algorithms, and do it in a faster time.

2 Code description

For the creation of the neural network, the code was divided into multiple classes to be easily readable and understandable. In this section, there will be a description of each class that was used to create the multilayer perceptron

2.1 Layer_Dense class

The Layer_Dense class represents a basic building block for the neural network. It performs a linear transformation on the input data during the forward pass and computes the gradients during the backward pass for updating the parameters during training. This class will be used in conjunction with activation functions and other layers to create the complete neural network architecture.

```python
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

2.2 Layer_Input class

The Layer_Input class represents the input layer, which is responsible for passing the input data through the network. It doesn't perform any computation or transformation on the input data during the forward pass. The output attribute is set to the input data, and this value is then used as the input for subsequent layers in the network.

```python
class Layer_Input:

    # Forward pass
    def forward(self, inputs, training):
        self.output = inputs
```

2.3 Activation_ReLU class

The Activation_ReLU class implements the ReLU activation function for forward and backward passes. The ReLU activation is widely used in neural networks to introduce non-linearity and sparsity in the network, helping in learning complex patterns. During the forward pass, it replaces negative values with zero, and during the backward pass, it backpropagates gradients through non-negative inputs.

```
# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

2.4 Activation_Softmax class

The Activation_Softmax class implements the Softmax activation function for forward and backward passes. It is commonly used in the output layer of neural networks for multiclass classification problems, providing normalized probability distributions over classes. During the backward pass, it computes the gradient of the loss concerning its inputs, allowing for effective training in multiclass scenarios.

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))

        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)
        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)
```

2.5 Optimizer_SGD class

The Optimizer_SGD class implements the Stochastic Gradient Descent optimization algorithm with optional learning rate decay and momentum. It is responsible for updating the weights and biases of a neural network layer during the training process. The momentum helps accelerate convergence in certain cases, and the learning rate can be decayed over time to fine-tune the optimization process.

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)

                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates

        # Vanilla SGD updates (as before momentum update)
        else:
            weight_updates = -self.current_learning_rate * \
                layer.dweights
            bias_updates = -self.current_learning_rate * \
                layer.dbiases

        # Update weights and biases using either
        # vanilla or momentum updates
        layer.weights += weight_updates
        layer.biases += bias_updates

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

2.6 Optimizer_Adam class

The Optimizer_Adam class implements the Adam optimization algorithm, which is a popular variant of stochastic gradient descent. It utilizes momentum-like updates for the first moment and RMSprop-like updates for the second moment. The algorithm also incorporates bias correction to adjust for the initialization bias of the first and second-moment estimates. This combination of techniques makes Adam effective in a wide range of optimization scenarios and is widely used in practice.

Note: The Adam Optimizer was taken directly from this source: [Kuk] due to lack of time in creating him from scratch I am not familiar with some aspects of this code

```
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum with current gradients
        layer.weight_momentums = self.beta_1 * \
            layer.weight_momentums + \
            (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
            layer.bias_momentums + \
            (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
            weight_momentums_corrected / \
            (np.sqrt(weight_cache_corrected) +
             self.epsilon)

        layer.biases += -self.current_learning_rate * \
            bias_momentums_corrected / \
            (np.sqrt(bias_cache_corrected) +
             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

2.7 Loss class

The Loss class provides a common interface for various loss functions. It includes methods for calculating losses for individual samples and accumulated losses over multiple samples. The remember_trainable_layers method allows storing references to trainable layers, which can be useful for regularization. Due to lack of time, the regularization functionality has not been added to the current code but could be included in future additions.

```
class Loss:
    #maybe this could be removed
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss # , self.regularization_loss()

    # Calculates accumulated loss
    def calculate_accumulated(self, *, include_regularization=False):

        # Calculate mean loss
        data_loss = self.accumulated_sum / self.accumulated_count

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss # , self.regularization_loss()

    # Reset variables for accumulated loss
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

2.8 Loss_CategoricalCrossentropy class

The Loss_CategoricalCrossentropy class provides the forward and backward computations for the categorical cross-entropy loss. It handles both sparse and one-hot encoded label formats and is designed to be used in neural network training for classification tasks with multiple classes. The forward pass calculates the negative log-likelihood, and the backward pass computes the gradient needed for updating model parameters during training.

```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

      #  print('y true shape is ', y_true.shape)
        # if len(y_true.shape) == 2:
        #    y_true = np.ravel(y_true)
        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
#        print(negative_log_likelihoods)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, y_pred, y_true):

        # Number of samples
        samples = len(y_pred)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(y_pred[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / y_pred
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

2.9 Activation_Softmax_Loss_CategoricalCrossentropy class

The Activation_Softmax_Loss_CategoricalCrossentropy class provides a convenient way to combine the Softmax activation and categorical cross-entropy loss in a single layer. The code is a simpler version of the previous 2 classes so it will not be added to be more concise.

2.10 Accuracy class

The Accuracy class provides a common interface for calculating accuracy in classification problems. It includes methods for calculating accuracy for individual batches, calculating the accumulated accuracy, and resetting variables for a new pass through the data. The compare method is responsible for comparing predictions with true labels and generating binary values indicating whether predictions are correct. This method is being created for each Accuracy class individually.

```python
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Add accumulated sum of matching values and sample count
        self.accumulated_sum += np.sum(comparisons)
        self.accumulated_count += len(comparisons)

        # Return accuracy
        return accuracy

    # Calculates accumulated accuracy
    def calculate_accumulated(self):

        # Calculate an accuracy
        accuracy = self.accumulated_sum / self.accumulated_count

        # Return accuracy
        return accuracy

    # Reset variables for accumulated accuracy
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

2.11 Accuracy_Categorical class

The Accuracy_Categorical class specializes in calculating accuracy for categorical classification problems. It extends the more general Accuracy class and introduces specific handling for one-hot encoded labels in non-binary classification tasks. The compare method is responsible for generating binary values based on the correctness of predictions.

```python
class Accuracy_Categorical(Accuracy):

    def __init__(self, *, binary=False):
        # Binary mode?
        self.binary = binary

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if not self.binary and len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y
```

2.12 Model class

The Model class serves as the central entity, managing the neural network architecture and training process. Key functionalities include:

- Initialization: The model initializes with an empty list of layers and a placeholder for the softmax classifier's output.
- Adding Layers: Layers can be added to the model using the add method, allowing for flexible model architecture design.
- Setting Loss, Optimizer, and Accuracy: The model can be configured with a specific loss function, optimizer, and accuracy metric using the set method.
- Finalizing the Model: The finalize method completes the model setup, establishing connections between layers, identifying trainable layers, and updating the loss object with trainable layers.
- Training the Model: The training method orchestrates the training process, including forward and backward passes, optimization, and printing summaries. It supports both batch and epoch-based training and optionally includes validation during training.
- Forward and Backward Passes: The forward and backward methods execute the forward and backward passes through the layers, respectively.
- Confusion Matrix and Metrics: The code generates confusion matrices for both the training and validation datasets, providing insights into model performance. Additionally, it computes accuracy, loss, and other metrics during training and validation.

Due to the amount of code that is needed to generate the model class, it will not be added to this report for conciseness

3 Problems

During the planning and creation of the neural network, there were issues regarding the way of the implementation of the code. Some of them are described below along with their solutions

3.1 Categorial Cross entropy loss

The formula of the categorial cross-entropy loss is given below:

$$CE = -\sum_{i=1}^{N} y_{true_i} * \log y_{pred_i} \tag{1}$$

This function creates an issue when $y_{pred} = 0$ because $\log y_{pred} = \infty$. Furthermore if $y_{pred} = 1$ then the result would be $\log y_{pred} = 0$ In order to work around this issue we need to put a max and a min value on the predicted values. This way we avoid extreme values and numerical instability during the computation of the cross-entropy loss

The 'np.clip' function from numpy is used to clip the values between 10^{-7} and $1 - 10^{-7}$

```
y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

3.2 Layer Input

During the creation of the Model class, the is that was encountered was the construction of the input layer. This layer did not have any formulas and its only objective was to pass the data to the next layer. So the typical Layer dense class could not be used. For this reason, a simple Input layer class was created which passes the input data to the next layer. This way we can incorporate the input layer into our class model

```
# Input "layer"
class Layer_Input:

   # Forward pass
   def forward(self, inputs, training):
      self.output = inputs
```

3.3 Activation Softmax for output

The rectified linear unit is unbounded, not normalized with other units, and exclusive. "Not normalized" implies the values can be anything, an output of [12, 99, 318] is without context, and "exclusive" means each output is independent of the others. To address this lack of context, the softmax activation on the output data can take in non-normalized, or uncalibrated, inputs and produce a normalized distribution of probabilities for our classes. In the case of classification, what we want to see is a prediction of which class the network "thinks" the input represents. This distribution returned by the softmax activation function represents confidence scores for each class and will add up to 1. The predicted class is associated with the output neuron that returned the largest confidence score

3.4 Activation Softmax derivative

One of the most difficult parts both with respect of understanding and coding the backward method of the activation softmax function. To have a better explanation of the final code, there will also be a theoretical explanation based on the [Kuk] book and the 3 blow 1 brown videos about neural networks.

First of all the activation softmax formula is as follows:

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,j}}} \tag{2}$$

So the derivative would be

$$\frac{\partial S_{i,j}}{z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,j}}}}{\partial z_{i,k}} \tag{3}$$

$$\frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{l,i}}]^2} \tag{4}$$

After some operations the final derivative is

$$S_{i,j}\delta_{j,K} - S_{i,j}S_{i,k} \tag{5}$$

This $\delta_{j,k}$ is the Kronecker delta whose equation is:

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} (6)$$

From (5), we can now create our backward function.
The first part of the derivative can be obtained by multiplying the Softmax output with a unity matrix. To do so the ‘diagflat‘ function of numpy was used. Our next part of the derivative is the multiplication of softmax outputs iterating over the j and k respectively. This is the dot product of the softmax output and its Transpose matrix. In conclusion the the derivative will be

```
np.diagflat(single_output) - \
              np.dot(single_output, single_output.T)
```

This is the Jacobian matrix. Jacobian matrix is an array of partial derivatives. We are calculating the partial derivatives of every output of the Softmax output concerning each input respectively. We need to sum the values from these vectors so that each of the inputs for each of the samples will return a single partial derivative value instead. Because each input influences all of the outputs, the returned vector of the partial derivatives has to be summed up for the final partial derivative with respect to this input. This is happening with the dot product So the final code will be:

```
for index, (single_output, single_dvalues) in \
          enumerate(zip(self.output, dvalues)):
      # Flatten output array
      single_output = single_output.reshape(-1, 1)
      # Calculate Jacobian matrix of the output
      jacobian_matrix = np.diagflat(single_output) - \
          np.dot(single_output, single_output.T)
      # Calculate sample-wise gradient
      # and add it to the array of sample gradients
      self.dinputs[index] = np.dot(jacobian_matrix,
                                single_dvalues
```

The loop iterates over each pair of the output of the softmax activation (single_output) and the corresponding gradient (single_dvalues). enumerate is used to get both the values and their index.

3.5 Combination of Softmax and loss categorial cross entropy

Due to the fact that the Softmax function is before the final output, it could be argued that a combination of the Softmax and the categorial cross-entropy functions for the forwards and backward methods would be beneficial and easier to code. To do so we need to combine the 2 derivatives. First of all, we need to understand which is the derivative by using the chain rule:

$$\frac{\partial L_i}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y_{i,j}}} \frac{\partial S_{i,j}}{z_{i,k}} \tag{7}$$

As we mentioned, the loss function is after the activation softmax function, so the input of the loss will be the output of the softmax, which means $S_{i,j} = \hat{y_{i,j}}$ Now we need to combine the 2 derivatives. The derivative of the loss function is

$$-\sum_j \frac{y_{i,j}}{\hat{y_{i,j}}} \tag{8}$$

and the softmax as was mentioned previously is

$$S_{i,j}\delta_{j,K} - S_{i,j}S_{i,k} \tag{9}$$

or for clarification before the Kronecker delta

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j}(1 - S_{i,k}) & j = k \\ S_{i,j}(0 - S_{i,k}) & j \neq k \end{cases} \tag{10}$$

So now we have the below function

$$-\sum_j \frac{y_{i,j}}{\hat{y_{i,j}}} \frac{\partial S_{i,j}}{\partial z_{i,k}} = -\frac{y_{i,j}}{\hat{y_{i,j}}}\hat{y_{i,j}}(1 - \hat{y_{i,k}}) + \sum_{j \neq k} \frac{y_{i,j}}{\hat{y_{i,j}}}(\hat{y_{i,j}}\hat{y_{i,k}}) \tag{11}$$

By simplifying the equation we have

$$-y_{i,k} + \sum_j y_{i,j}\hat{y_{i,k}} \tag{12}$$

and because we have 1 hot encoded vector it means that the sum will be different from 0 only if $j = k$ which will be $\hat{y_{i,k}}$ So the final formula will be

$$\hat{y_{i,k}} - y_{i,k} \tag{13}$$

This formula is much easier to be coded and escape possible mistakes during the development.

3.6 Model's layers order

In order to use our model and make the UX better, we need to find a way to know each layer's previous and next output. To achieve this an array of layers is created inside the model so that we can connect the layers, without tinkering with the layer dense and activation functions. To achieve this we iterate over the layers that are inside our list of layers and we define their previous and next layer. If it is the first hidden layer of our neural network then as previous there is the initial layer with the data, and if it is the last hidden layer which is the softmax activation function then its next layer will be the loss function

3.7 Data for each epoch

As each epoch comes to an end, we need to know the neural network performance, as it is desired on the report. To do so we need to add accumulated accuracy and loss variables to our accuracy and Loss classes. Furthermore, it is needed to reset these values, each time we start a new epoch. In order to achieve these, we need to add a function whose job will be to reset the accumulated values. This will also be added to the Activation and Loss classes, as it is in line with the separation of concerns philosophy. This is happening with the accumulated variables in each aforementioned class, as well as a forward pass that resets to zero the accumulated values

For the loss class the accumulated loss and the erase of the previous accumulated loss can be seen below:

```
# Calculates accumulated loss
   def calculate_accumulated(self, *, include_regularization=False):

       # Calculate mean loss
       data_loss = self.accumulated_sum / self.accumulated_count

       # If just data loss - return it
       if not include_regularization:
           return data_loss
```

```
        # Return the data and regularization losses
        return data_loss # , self.regularization_loss()

    # Reset variables for accumulated loss
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

4 Results

The tests were made with a Lenovo ideapad flex 5 with a Ryzen 3 4000 series CPU Radeon graphics and 4 GB of RAM. The tests showed similar results (with less time to be concluded) on a PC with an Nvidia GTX 1050ti and an Intel Core i5 7500 series 16 GB RAM.

The results are divided into 4 sections depending on the method we used to classify the images. In each section there will be a short description of the method, the time it took to make the classifications, a confusion matrix that shows the predicted and the actual classifications of the images, the accuracy and the loss (in the neural network method).

4.1 Pre-process data

To use the CIFAR-10 dataset there was a need for pre-processing. The data need to have values between 0 and 1 (normalization). This happens because we do not want higher values to affect greater results in comparison to lower values. The images are colored and the values range from 0 to 255. To normalize the values we divide the inputs with the maximum value which is 255.

```
# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# make the rgb value from 0 - 255 --> 0 - 1 ==> scaling
X_train, X_test = X_train / 255.0, X_test / 255.0

# CIFAR-10 class names
class_names = ['airplane', 'automobile', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

After the normalization of the data, we need to lessen the dimensions of our data. When we are working with colored images we have 3 dimensions.

- The width of the image
- The height of the image
- The three color channels (Red, Green, Blue) for each pixel.

The ultimate goal is to make these 3 dimensions into 1. This way we will be able to use our data in our methods. To do so we are using the below code:

```
# we want to reshape the image from a 4D array to a 2D array
# This line extracts the number of samples, image height, image width, and number of
    channels (e.g., RGB channels) from the shape of the X_train array.
num_samples, img_height, img_width, num_channels = X_train.shape

# it flattens the image data, converting each image into a one-dimensional vector.
# The resulting shape is (num_samples, img_height * img_width * num_channels),
X_train = X_train.reshape(num_samples, -1)
num_samples, img_height, img_width, num_channels = X_test.shape
X_test = X_test.reshape(num_samples, -1)
```

Note: The other dimension that is mentioned is the number of samples that we have which are 50,000 for the train and 10,000 for the test samples

Note: The Null accuracy score: 0.1000. This is the score if we predicted each time the most common category. Because each class has the same amount of data, our prediction would be 10% of the time accurate. This is a baseline to compare the other methods.

4.2 Classification report

The classification report is a more detailed analysis of the predicted values of each class. It will be used in the knn and centroid algorithms. Below there is a description of each value on this report

4.2.1 Precision

Precision can be defined as the percentage of correctly predicted positive outcomes out of all the predicted positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true and false positives (TP + FP).

So, Precision identifies the proportion of correctly predicted positive outcomes. It is more concerned with the positive class than the negative class. Check the predicted airplanes, sum them up, and divide the actual airplanes (vertical lines on confusion matrices)

Mathematically, precision can be defined as the ratio of TP to (TP + FP).

4.2.2 Recall

Recall can be defined as the percentage of correctly predicted positive outcomes out of all the actual positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true positives and false negatives (TP + FN). Recall is also called Sensitivity.

Recall identifies the proportion of correctly predicted actual positives.

Mathematically, recall can be given as the ratio of TP to (TP + FN). Check the times the predicted airplane, sum them up, and divide the actual airplanes (horizontal line)

4.2.3 f1-score

f1-score is the weighted harmonic mean of precision and recall. The best possible f1-score would be 1.0 and the worst would be 0.0. f1-score is the harmonic mean of precision and recall. So, the f1-score is always lower than accuracy measures as they embed precision and recall into their computation. The weighted average of the f1-score should be used to compare classifier models, not global accuracy. not so useful in multi-class classification

4.2.4 Support

Support is the actual number of occurrences of the class in our dataset.

4.3 Cross Validation

Cross-validation is a technique used in machine learning to assess the performance of a model on multiple subsets of the dataset. The "k-fold" cross-validation is a specific type where the dataset is divided into k equally sized folds or subsets. The model is trained and evaluated k times, each time using a different fold as the test set and the remaining folds as the training set. This helps in obtaining a more robust performance estimate and reducing the impact of the dataset's partition on the evaluation. This will be done both in knn and centroid algorithms

4.4 Nearest Neighbour algorithm

The K-Nearest Neighbor (KNN) algorithm is a popular machine-learning technique used for classification and regression tasks. It relies on the idea that similar data points have similar labels or values.

During the training phase, the KNN algorithm stores the entire training dataset as a reference. When making predictions, it calculates the distance between the input data point and all the training examples, using a chosen distance metric such as Euclidean distance.

Next, the algorithm identifies the K nearest neighbors to the input data point based on their distances. In the case of classification, the algorithm assigns the most common class label among the K neighbors as the predicted label for the input data point.

4.4.1 1 Nearest Neighbour

The 1 Nearest Neighbour uses the neighbor closest to our input data to predict its class.

The time it took to make the predictions was **34.5 seconds**

The Model accuracy score for 1 neighbor was: **0.3539** This is an unacceptable prediction rate and as a result, it is not recommended to use this algorithm to classify our data

The Training-set accuracy score for 1 neighboor: **1.0000** We can see that even though the algorithm is 100% accurate on our training data, it cannot predict new data

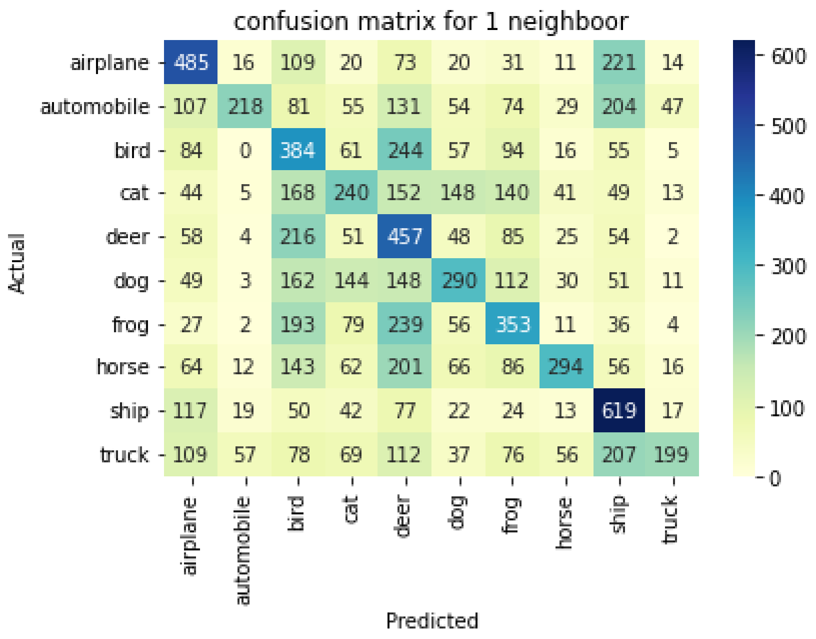
4.4.1.1 Classification report

The classification report is

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.42	0.48	0.45	1000
1	0.65	0.22	0.33	1000
2	0.24	0.38	0.30	1000
3	0.29	0.24	0.26	1000
4	0.25	0.46	0.32	1000
5	0.36	0.29	0.32	1000
6	0.33	0.35	0.34	1000
7	0.56	0.29	0.39	1000
8	0.40	0.62	0.49	1000
9	0.61	0.20	0.30	1000
Accuracy			0.35	10000
Macro Avg	0.41	0.35	0.35	10000
Weighted Avg	0.41	0.35	0.35	10000

4.4.1.2 Confusion matrix

The confusion matrix is given below



Actual \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	485	16	109	20	73	20	31	11	221	14
automobile	107	218	81	55	131	54	74	29	204	47
bird	84	0	384	61	244	57	94	16	55	5
cat	44	5	168	240	152	148	140	41	49	13
deer	58	4	216	51	457	48	85	25	54	2
dog	49	3	162	144	148	290	112	30	51	11
frog	27	2	193	79	239	56	353	11	36	4
horse	64	12	143	62	201	66	86	294	56	16
ship	117	19	50	42	77	22	24	13	619	17
truck	109	57	78	69	112	37	76	56	207	199

Figure 1: Confusion matrix of 1 nearest neighbor algorithm

4.4.1.3 Cross Validation

Cross-validation scores with 1nn:

0.3382	0.3406	0.3464	0.3382	0.3454	0.36	0.3424	0.3404	0.352	0.3496

We see that the scores are similar to each other which means that different sets of data don't influence the results Average cross-validation score with 1nn: **0.3453**

4.4.2 3 Nearest Neighbours

The 3 Nearest Neighbours uses the 3 neighbors closest to our input data to predict its class.

The time it took to make the predictions was **41.5 seconds**

The Model accuracy score for 3 neighbors: **0.3303** This accuracy is even lower than our 1 nearest neighbor which validates our conclusion that we cannot use the knn algorithm to predict the classes of the images

The Training-set accuracy score for 3 neighboors: **0.5790** We can see that even on the data that the algorithm is trained, it has a 58% accuracy which means that it cannot be used even for data that it has already seen

4.4.2.1 Classification report the classification report is

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.32	0.57	0.41	1000
1	0.58	0.24	0.34	1000
2	0.20	0.45	0.28	1000
3	0.26	0.23	0.24	1000
4	0.25	0.44	0.32	1000
5	0.43	0.21	0.28	1000
6	0.36	0.23	0.28	1000
7	0.73	0.20	0.31	1000
8	0.44	0.61	0.51	1000
9	0.73	0.12	0.21	1000
Accuracy			0.33	10000
Macro Avg	0.43	0.33	0.32	10000
Weighted Avg	0.43	0.33	0.32	10000

4.4.2.2 Confusion matrix The confusion matrix is given below

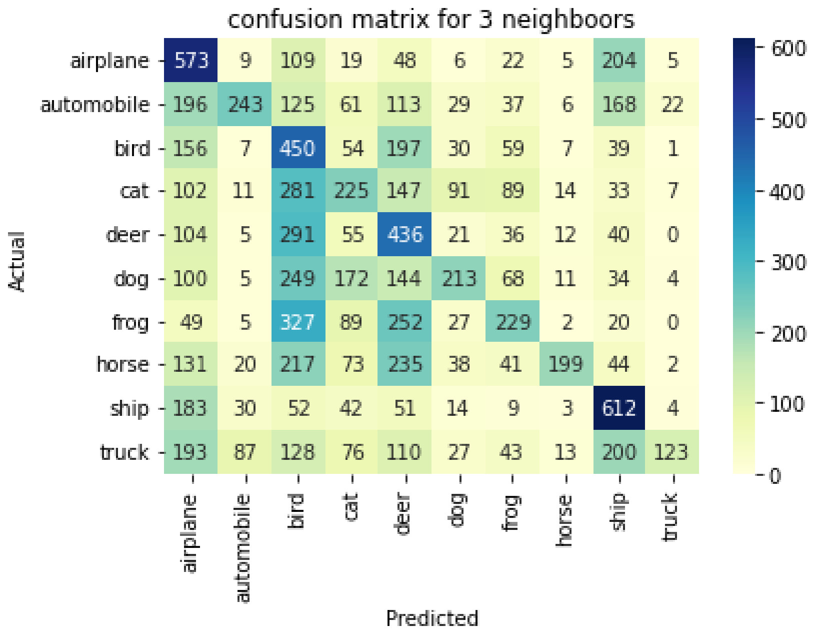


Figure 2: Confusion matrix of 3 nearest neighbors algorithm

We see that the ship category has a higher prediction rate than the 1 nearest neighbor but overall the loss was increased.

4.4.2.3 Cross validation The cross-validation report is

0.3308	0.327	0.3326	0.3236	0.3306	0.3402	0.3294	0.3258	0.326	0.331

Average cross-validation score with 3nn: 0.3297

Similarly to the 1nn neighbor, we see that the results are not correlated to the data that we get, so they don't change significantly

4.5 Nearest Centroid algorithm

In machine learning, a nearest centroid classifier or nearest prototype classifier is a classification model that assigns to observations the label of the class of training samples whose mean (centroid) is closest to the observation.

What the nearest centroid classifier does can be explained in three steps:

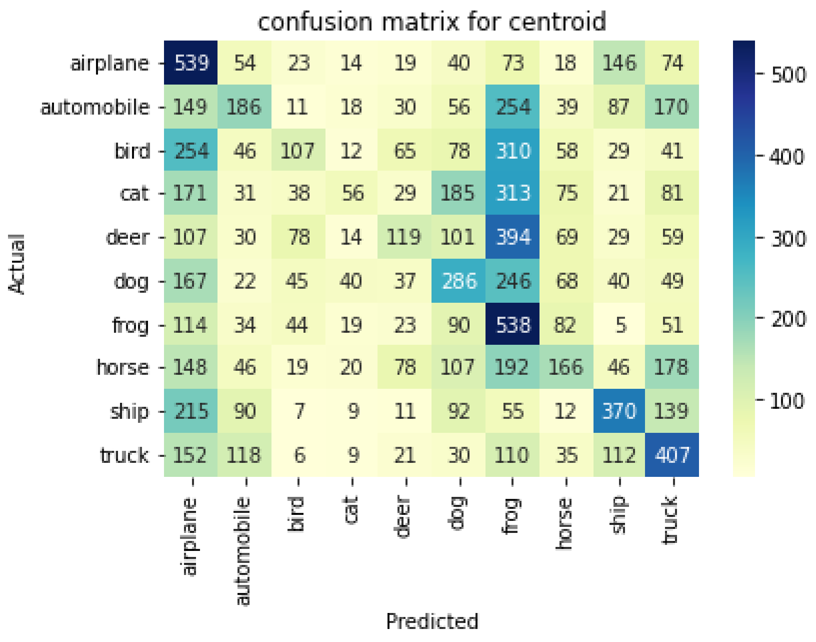
- The centroid for each target class is computed while training.
- After training, given any point, say 'X'. The distances between point X and each class' centroid are calculated.
- Out of all the calculated distances, the minimum distance is picked. The centroid to which the given point's distance is minimum, its class is assigned to the given point.

The time to conclude the nearest centroid classification was:

Model accuracy score for centroid classifier: 0.2774 We see that the accuracy is even lower than the 3nn prediction which makes the algorithm unusable for for such classifications

Training-set accuracy score for centroid classifier: 0.2697 The accuracy score of the training data set is even lower than the test dataset which is strange and shows the algorithm's inability to predict even the data it was trained with.

4.5.1 Confusion matrix



Actual \ Predicted	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	539	54	23	14	19	40	73	18	146	74
automobile	149	186	11	18	30	56	254	39	87	170
bird	254	46	107	12	65	78	310	58	29	41
cat	171	31	38	56	29	185	313	75	21	81
deer	107	30	78	14	119	101	394	69	29	59
dog	167	22	45	40	37	286	246	68	40	49
frog	114	34	44	19	23	90	538	82	5	51
horse	148	46	19	20	78	107	192	166	46	178
ship	215	90	7	9	11	92	55	12	370	139
truck	152	118	6	9	21	30	110	35	112	407

Figure 3: Enter Caption

We can see that the algorithm predicted falsely frog for 5 categories more than 200 times which may indicate something about our data

4.5.2 Classification report

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.27	0.54	0.36	1000
1	0.28	0.19	0.22	1000
2	0.28	0.11	0.16	1000
3	0.27	0.06	0.09	1000
4	0.28	0.12	0.17	1000
5	0.27	0.29	0.28	1000
6	0.22	0.54	0.31	1000
7	0.27	0.17	0.20	1000
8	0.42	0.37	0.39	1000
9	0.33	0.41	0.36	1000
Accuracy			0.28	10000
Macro Avg	0.29	0.28	0.25	10000
Weighted Avg	0.29	0.28	0.25	10000

4.5.3 Cross validation

0.2696	0.2768	0.2656	0.256	0.2858	0.2708	0.26	0.2704	0.2672	0.2724

Average cross-validation score with the nearest centroid: 0.2695 Again we see that the result is not correlated with the data that we pick which means that our initial result is unbiased

4.6 Neural Network

Our Neural Network is a multilayer perception.

We will use 2 optimization methods and different numbers of layers with different numbers of neurons to check their performance

We will keep the neurons between 128 and 256. These values were picked because they are a power of 2 Choosing several neurons that have a power of 2 is not a strict rule or requirement in neural network design. It's more of a convention or heuristic that is sometimes followed for practical reasons. The reason for that is Hardware Optimization. Some hardware architectures, especially GPUs, are designed to work more efficiently with sizes that are powers of 2. The hardware might have optimizations that work better with these sizes, leading to faster computation.

4.6.1 SGD optimizer

4.6.1.1 Learning Rate 1 We will first try the SGD optimizer without decay nor momentum with a learning rate = 1.

Results: validation, acc: 0.100, loss: 2.305 **Elapsed time**: 92.8s

We conclude that the neural network is not learning and we do not have any progress during our epochs. It predicts the same category each time so the accuracy remains at 10%.

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.106	2.304	0.102	2.305	1.0
5	0.102	2.304	0.100	2.305	1.0
10	0.102	2.306	0.100	2.305	1.0

Table 1: Training and Validation Data SGD

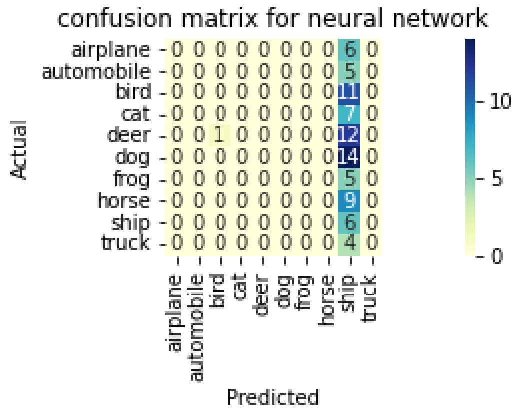
We will try to increase the learning rate and add a decay

4.6.1.2 learning rate and decay Decay is a way of reducing the learning rate as we move forward with the epochs. We have decided to use: **learning_rate=2.5, decay=5e-5**

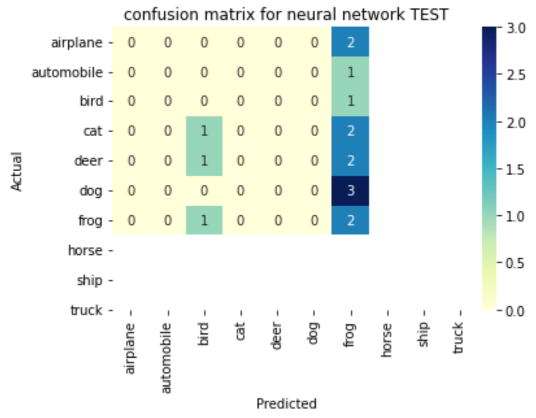
Results: validation, acc: 0.100, loss: 2.307 Elapsed time: 108.1 s

The loss was increased a bit and we still do not see any major improvements during the epochs, which means that this optimizer is still not usable for our categorization

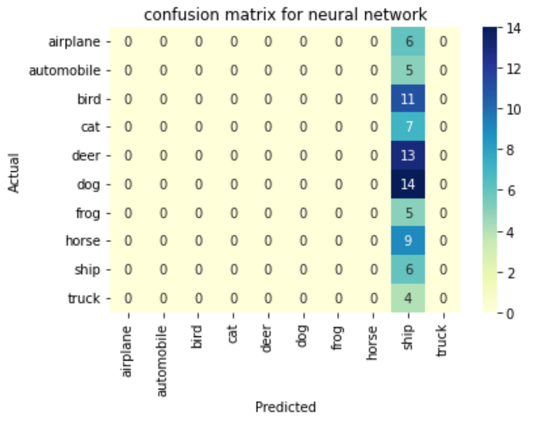
our next try is to use momentum to decrease our loss



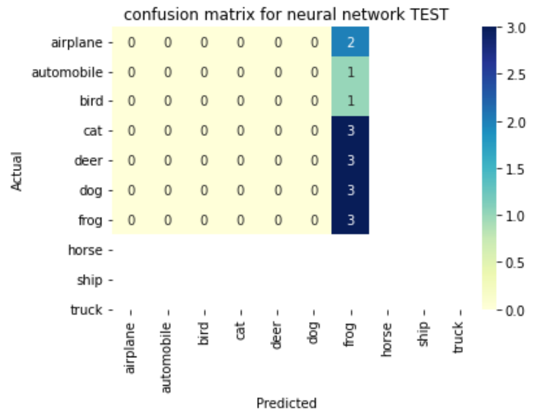
(a) 1 epoch train



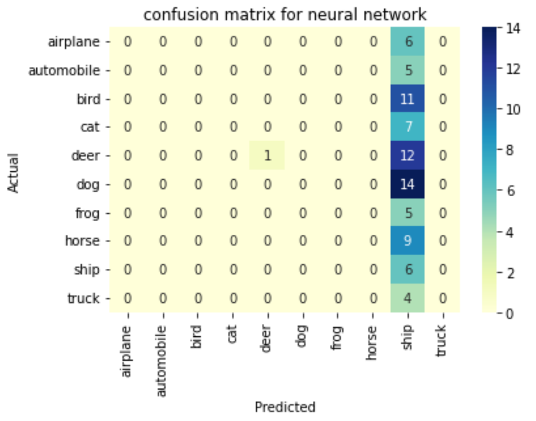
(b) 1 epoch test



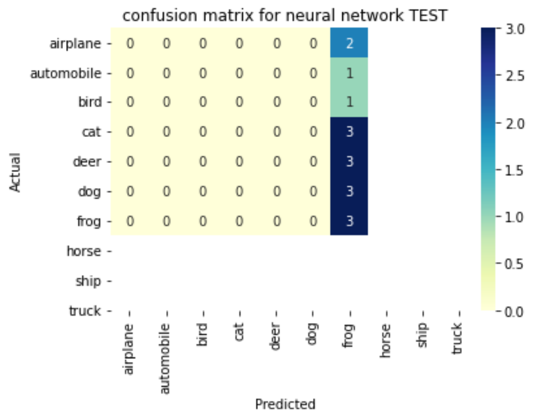
(c) 5 epoch train



(d) 5 epoch test



(e) 10 epoch train

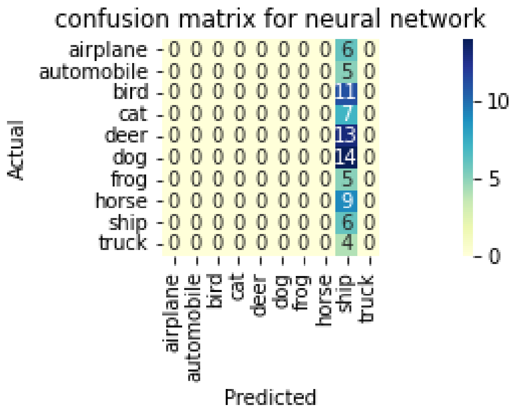


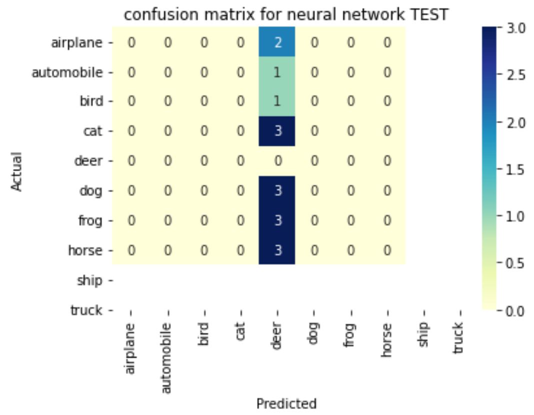
(f) 10 epoch test

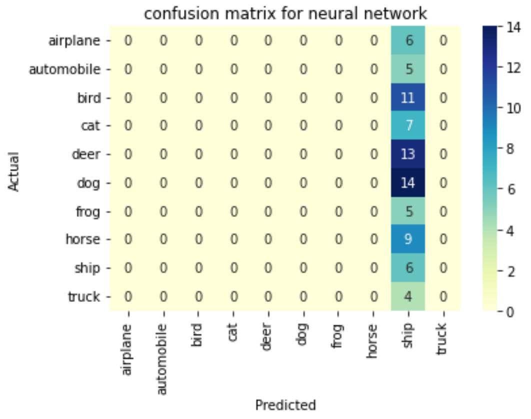
Figure 4: SGD with learning rate = 1

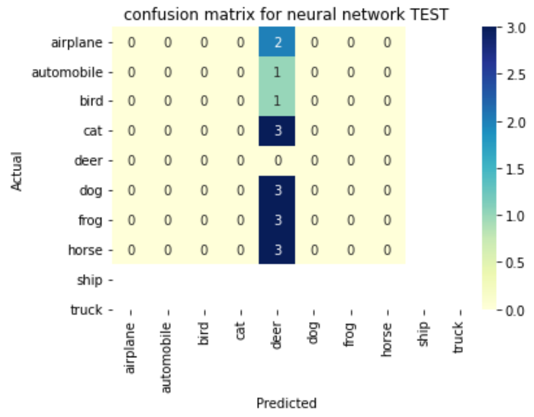
Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.100	2.343	0.100	2.309	2.4521824423737124
2	0.100	2.307	0.100	2.308	2.2774892958003097
3	0.100	2.307	0.100	2.307	2.091262704420929

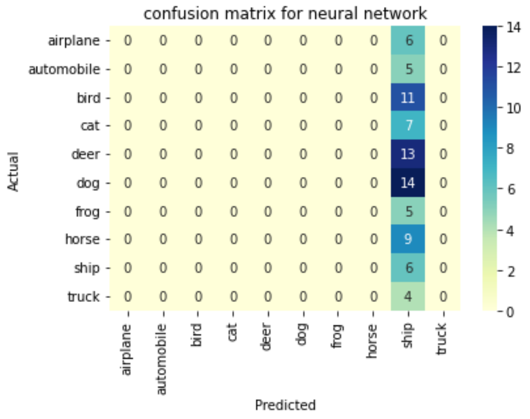
Table 2: Training and Validation Data with decay


(a) 1 epoch train


(b) 1 epoch test


(c) 5 epoch train


(d) 5 epoch test


(e) 10 epoch train

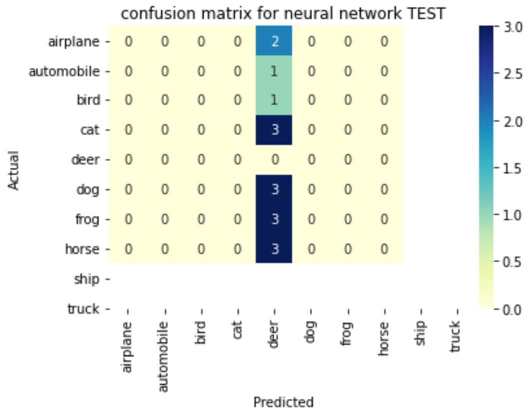

(f) 10 epoch test

Figure 5: SGD with learning rate and decay

4.6.2 SGD with momentum

Momentum creates a rolling average of gradients over some number of updates and uses this average with the unique gradient at each step. Another way of understanding this is to imagine a ball going down a hill — even if it finds a small hole or hill, momentum will let it go straight through it towards a lower minimum — the bottom of this hill. This can help in cases where you're stuck in some local minimum (a hole), bouncing back and forth. With momentum, a model is more likely to pass through local minimums, further decreasing loss **Results**: validation, ACC: 0.100, loss: 2.309 Elapsed time: 200.3 s

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.100	2.644	0.100	2.310	2.4521824423737124
2	0.099	2.310	0.100	2.309	2.2774892958003097
3	0.099	2.309	0.100	2.309	2.091262704420929

Table 3: Training and Validation Data with momentum

We still see no difference in the optimizing method which means that 3 layers with 128 neurons might be too low to have any major improvements to our classification efforts. There is a slight possibility there is a problem with the code that needs to be confirmed, but otherwise, the optimizer is not doing its job so we will try the Adam optimizer

4.6.3 Adam

we will use the Adam optimizer with 3 layers and 128 neurons each Elapsed time: 168.06

We can see that the train images as well as the test images are getting better over each epoch. the loss is getting lower and the accuracy is higher. It is still understandable that a 48% accuracy and a loss of 1.464 is not good enough for a neural network but it is better than the SGD optimizer and for this reason, we will use Adam instead of SGD. Below we have some values of the loss and the accuracy in the epochs we showcase the confusion matrices.

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.275	1.951	0.350	1.791	0.0009808729769494849
5	0.444	1.561	0.454	1.527	0.0009109957183201238
10	0.493	1.420	0.480	1.464	0.0008365050817683717

Table 4: Training and Validation Data Adam

4.6.4 2 Layers 128 neurons

Elapsed time: 289.39s

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.213	2.045	0.271	1.919	0.0009808729769494849
2	0.422	1.608	0.432	1.578	0.0009109957183201238
3	0.489	1.428	0.482	1.465	0.0008365050817683717

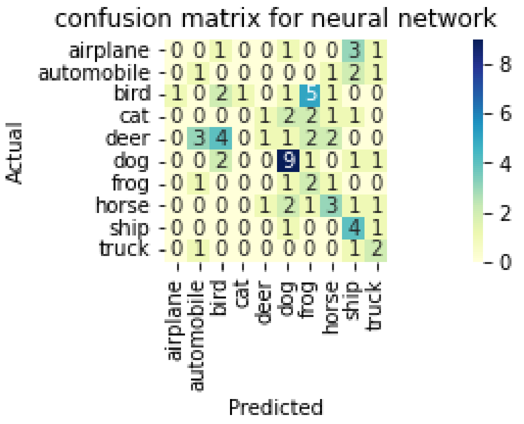
Table 5: Training and Validation Data 2 layers 128 neurons

We see improvements with 2 layers also but the elapsed time is longer than the 3 layers

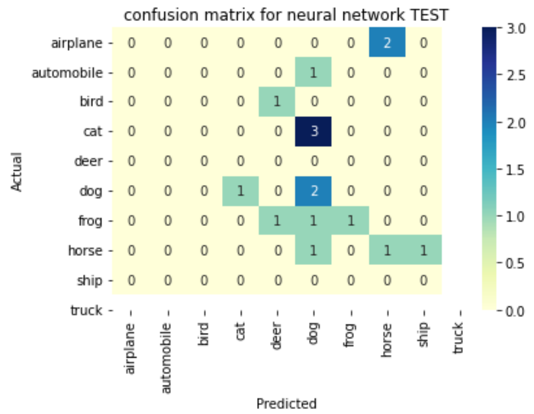
4.6.5 2 Layers 256 neurons

Elapsed time: 463.74s

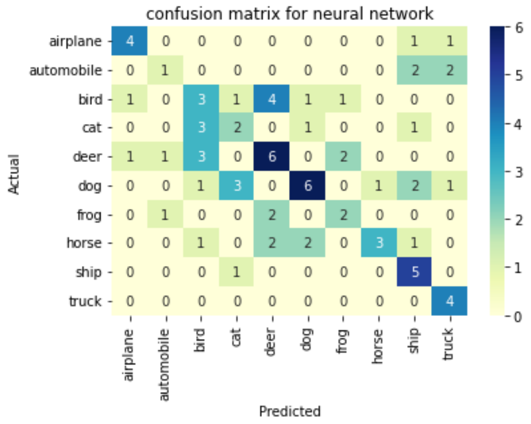
The same can be said here. We are close to the loss of 3 neurons but the time it took is almost 3 times longer



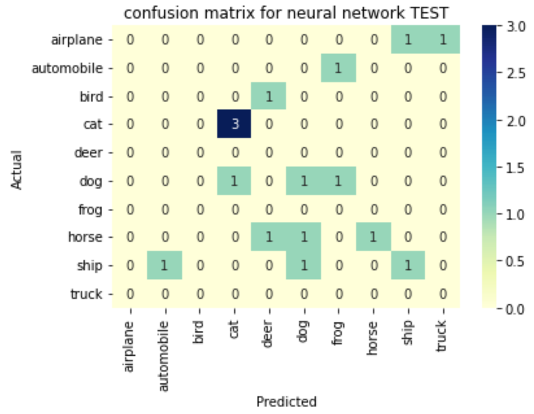
(a) 1 epoch train



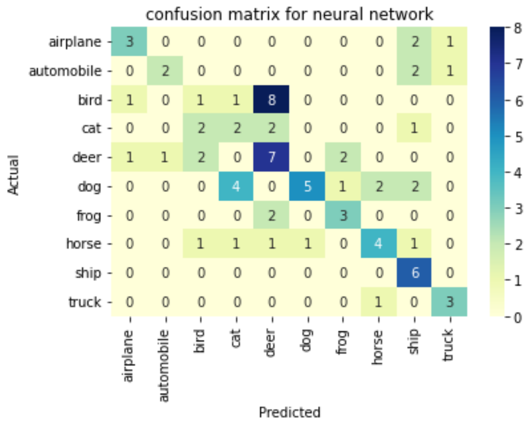
(b) 1 epoch test



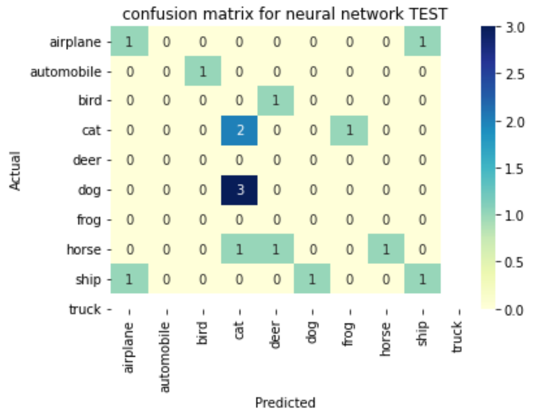
(c) 5 epoch train



(d) 5 epoch test



(e) 10 epoch train

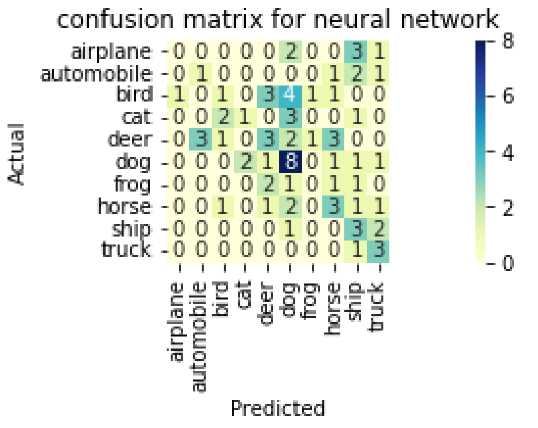


(f) 10 epoch test

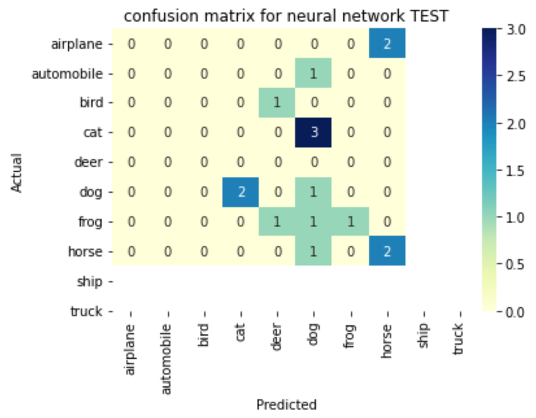
Figure 6: Adam

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.322	1.888	0.368	1.750	0.0009808729769494849
2	0.454	1.543	0.447	1.557	0.0009109957183201238
3	0.496	1.432	0.471	1.479	0.0008365050817683717

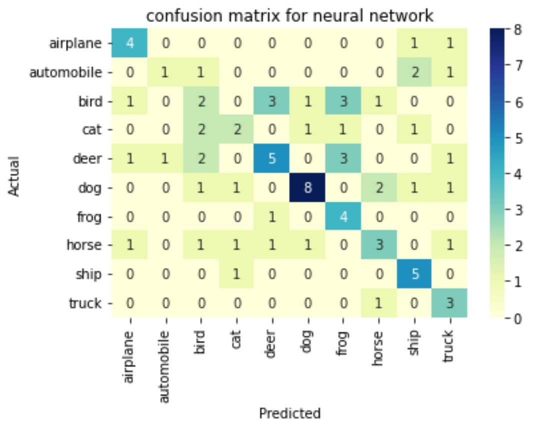
Table 6: Training and Validation Data



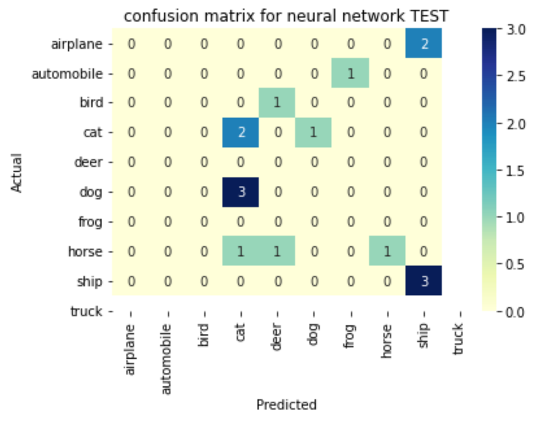
(a) 1 epoch train



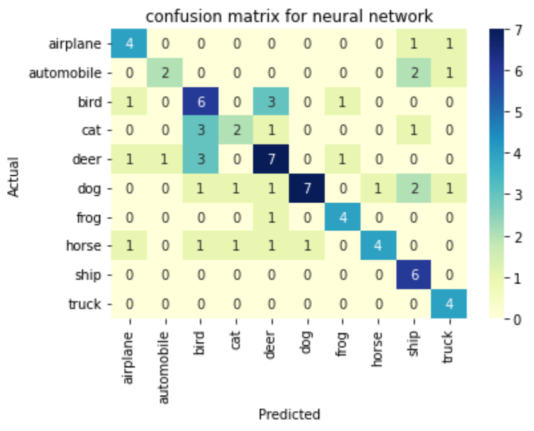
(b) 1 epoch test



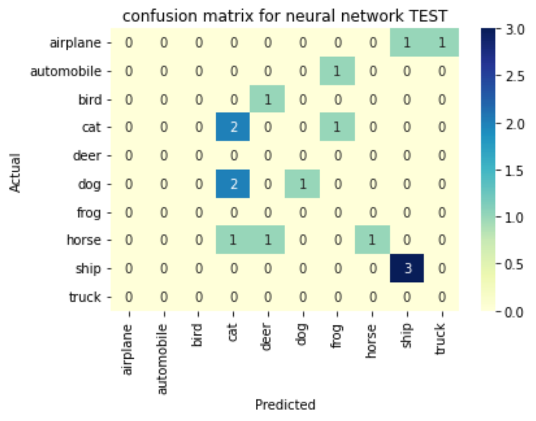
(c) 5 epoch train



(d) 5 epoch test

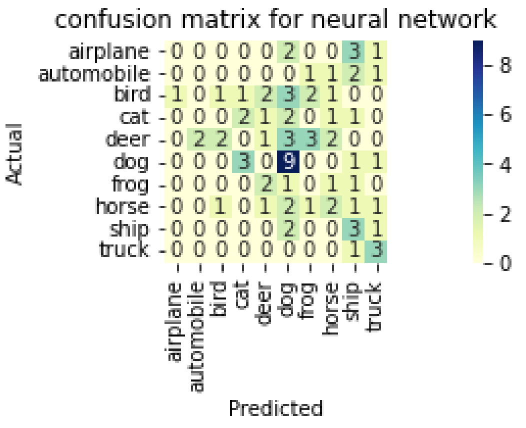


(e) 10 epoch train

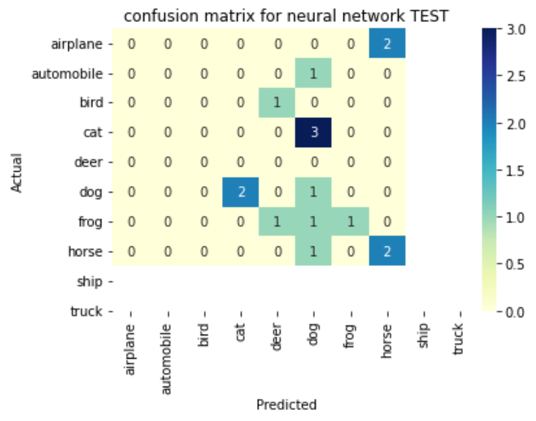


(f) 10 epoch test

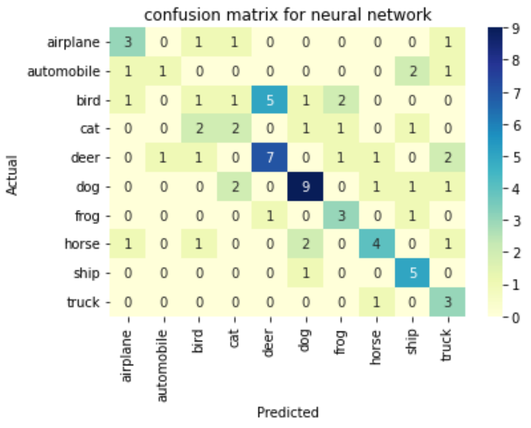
Figure 7: 2 Layers 128 neurons



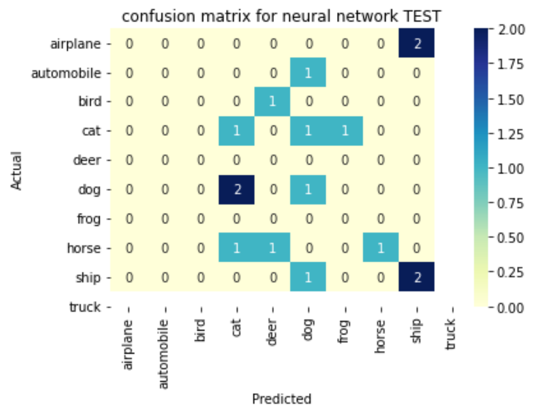
(a) 1 epoch train



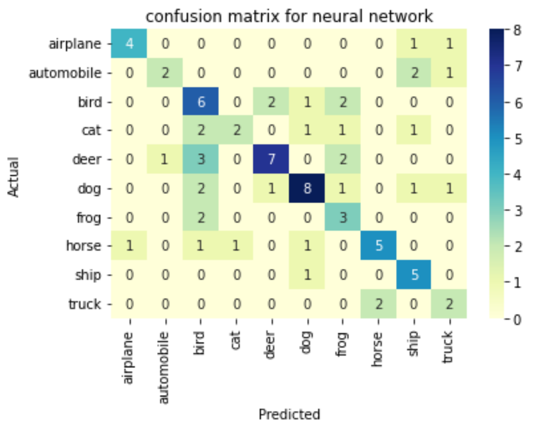
(b) 1 epoch test



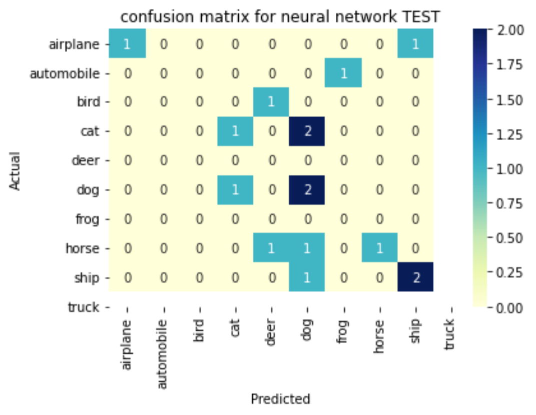
(c) 5 epoch train



(d) 5 epoch test



(e) 10 epoch train



(f) 10 epoch test

Figure 8: 2 layers 256 neurons

4.6.6 3 Layers 256 neurons

Elapsed time: 412.73
The time is evidently longer than 128 neurons with marginal improvement

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.302	1.902	0.365	1.738	0.0009808729769494849
2	0.461	1.500	0.460	1.511	0.0009109957183201238
3	0.516	1.357	0.490	1.447	0.0008365050817683717

Table 7: Training and Validation Data

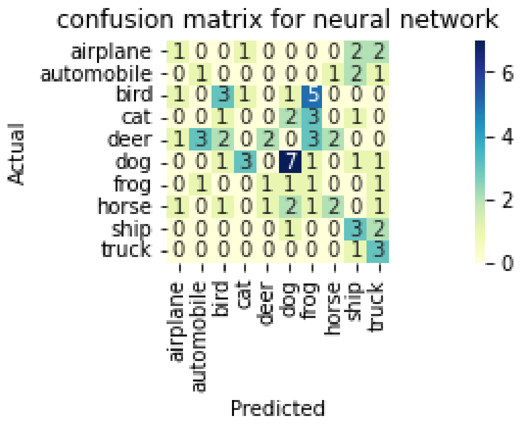
We will try to use 4 layers as a last resort for improvement

4.6.7 4 layers 128 neurons

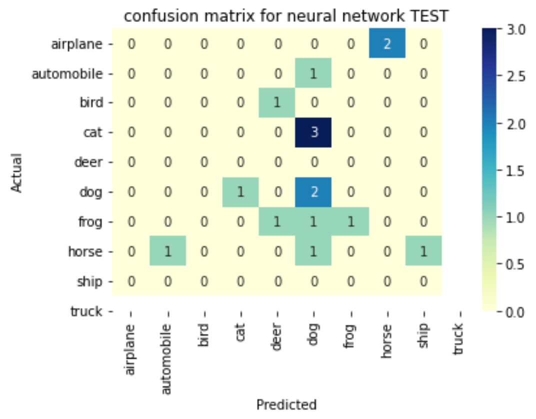
Elapsed time: 283.95s
We see that the time is lower than the 3 layers with 256 neurons but greater than 3 layers with 128 neurons without any improvements

Epoch	Training Acc	Training Loss	Validation Acc	Validation Loss	Learning Rate
1	0.213	2.045	0.271	1.919	0.0009808729769494849
2	0.422	1.608	0.432	1.578	0.0009109957183201238
3	0.489	1.428	0.482	1.465	0.0008365050817683717

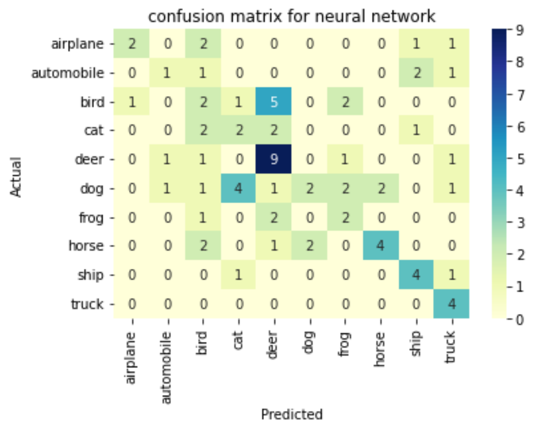
Table 8: Training and Validation Data



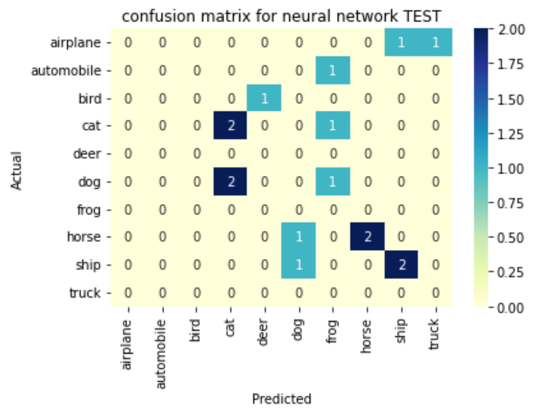
(a) 1 epoch train



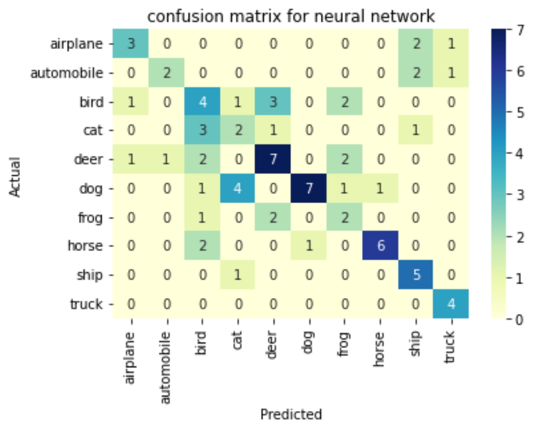
(b) 1 epoch test



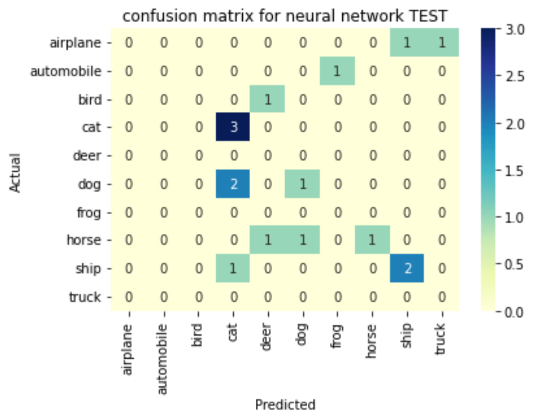
(c) 5 epoch train



(d) 5 epoch test

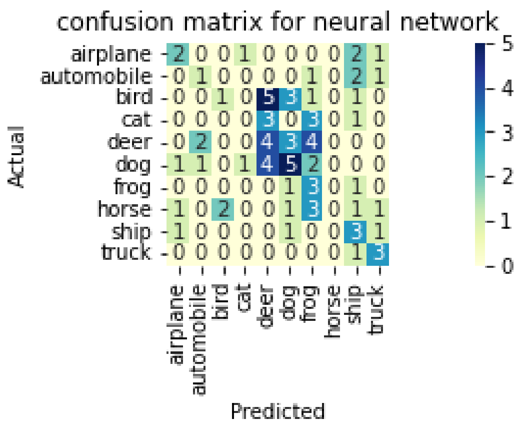


(e) 10 epoch train

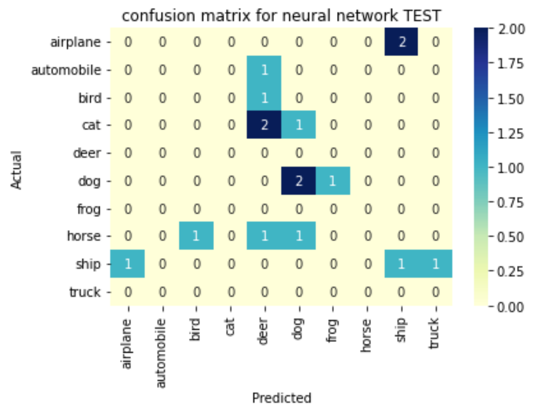


(f) 10 epoch test

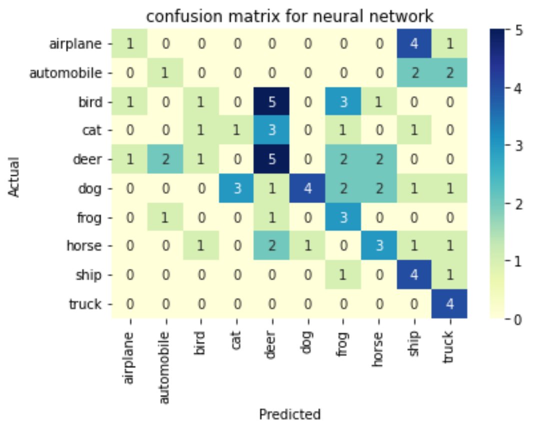
Figure 9: 3 Layers 256 neurons



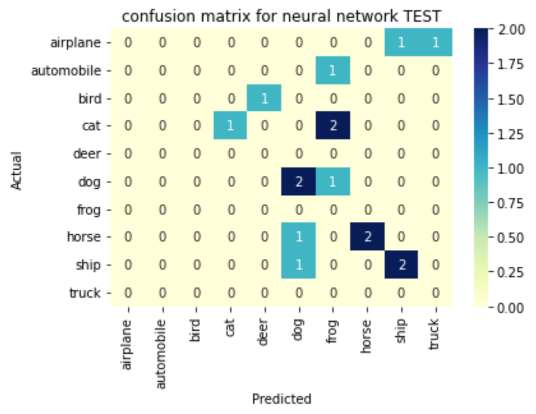
(a) 1 epoch train



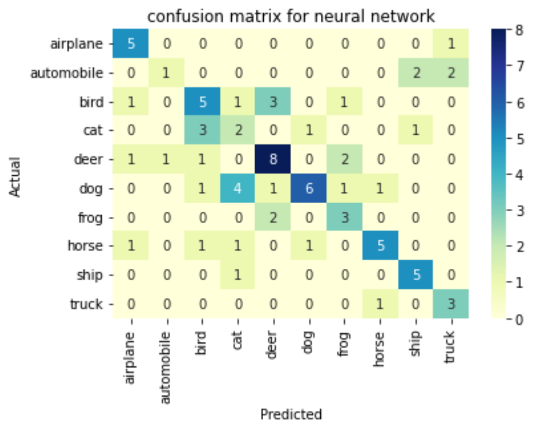
(b) 1 epoch test



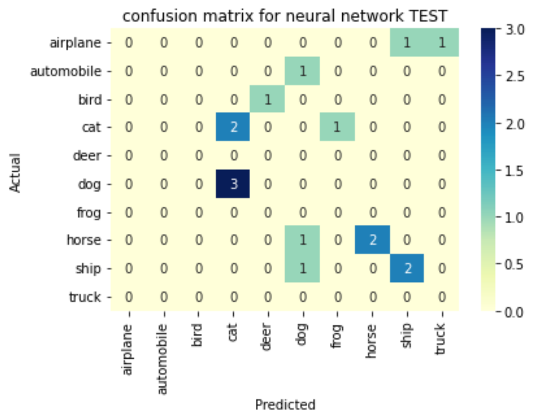
(c) 5 epoch train



(d) 5 epoch test



(e) 10 epoch train



(f) 10 epoch test

Figure 10: 4 Layers 128 neurons

5 Conclusions

From the results, we can conclude that:

- The nearest neighbor and nearest centroid algorithms cannot be used for our classification
- The SGD optimizer does not work for our case, even with momentum. The optimizer is not able to identify the images nor to learn during epochs. Further search must be done in case of a bug in the code.
- The Adam optimizer works well for our classification problem and can decrease the loss during each epoch
- The optimal number of layers and neurons are: **3 Layers with 128 neurons each**. This way we can achieve 48% accuracy and a loss of 1.464 with an elapsed time of 168 seconds. It is better than our previous algorithms and faster than using more or less layers.
- The best accuracy and loss were achieved with 3 layers and 256 neurons (49% and 1.447 loss) but it took 412 seconds which is roughly 2.5 times slower than the 128 neurons.
- Further studies should be conducted with more layers, neurons, and epochs to check for an optimal solution. Right now hardware limitations restrain our test capabilities.

References

[Kuk] Harrison Kinsley Daniel Kukieła. *Neural Networks from Scratch in Python.*