Building a Simple Image Classification Neural Network

MTE 203 Project 2
prepared for Professor Matthew Harris

Lucas Di Pietro

July 2023

Introduction

The field of computer vision has long since been one of the driving forces of the AI revolution; classification of images is one of its most difficult and rewarding goals. In this report we define, construct, and test a multilayer perceptron neural network. The network is built using Python, specifically using the NumPy library to leverage its linear algebra tools, and trained on the CIFAR-10 dataset [1], a set of 60,000 32x32 pixel RGB images, labelled with one of 10 labels. The training dataset consists of 50,000 of these images, and the testing dataset consists of the remaining 10,000. The labels and an image from each are provided in Figure 1:

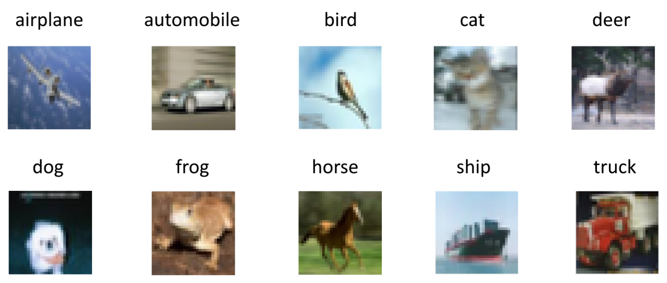


Figure 1: The categories of images in the CIFAR-10 dataset. Source: Adapted from [2]

Our goal is to build a neural network, which, through iterative training, can be given another image in one of these categories, and label it accordingly, outputting which category the image falls into. The success rate of the network in doing so, or the *accuracy* of its predictions, quantify how "good" the network is at its job. Training, in the context of our network, refers to the following cycle:

1. Provide the network with input data (for CIFAR-10, a training image) and record its response.
2. Measure how much error was present in the response, compared to the expected output.
3. Determine which parameters of the network were responsible for the error, and tune them accordingly.

By repeating this cycle many times, we try to minimize the error in the response and thus increase the accuracy of the network's responses. By quantifying the error in the network as a function of the parameters, this becomes a very large optimization problem, however, shortcuts exist to drastically simplify the derivation.

The CIFAR-10 dataset was published in 2008; in 2010, its creator Krizhevsky reported achieving 77-79% accuracy on the dataset using contemporary methods [3]. Modern networks can achieve about 99% accuracy using state-of-the-art methods [4]. Our network will only use a basic neural network structure; we hope to achieve respectable accuracy despite said constraint.

Background

The *perceptron*, a network of artificial neurons, was first devised in 1943 [5], but it was not until 15 years later that it was implemented by Frank Rosenblatt [6]. The structure of Rosenblatt's perceptron resembles modern neural networks, including our own. The basic structure of a multilayer perceptron neural network is a set of L layers, each containing N_L neurons. A neuron is the basic unit of a network, and can take on any value (its 'activation') between 0 and 1. We represent the activation of the N$^{\text{th}}$ neuron in the L$^{\text{th}}$ layer with $a_N^{(L)}$. There are three types of layers in a neural network like ours:

1. **Input Layers -** Encode the data that will be fed into the network as activation values. The number of neurons in this layer is always application-specific.
2. **Hidden Layers -** Transitory neurons; can be considered a "black box" system.
3. **Output Layers -** Represent the network's result. The number of output layer neurons required is application-specific, and often represents some choice or prediction.

Figure 2 shows how these layers of neurons make connections between layers to form a network.

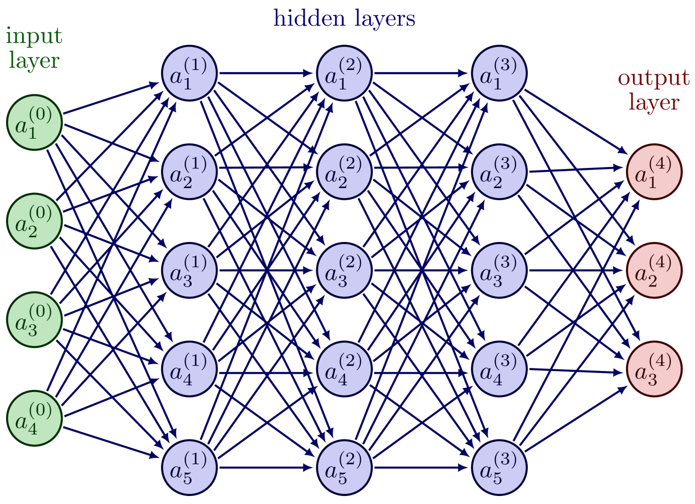


Figure 2: The basic structure of a multilayer perception neural network. Source: [7]

We will build a classification network, where each output neuron corresponds to one of the CIFAR-10 labels. The highest activation output neuron represents the network's classification of the input image.

The Neuron Activation Equation

To calculate the activations of neurons in the next layer, we start by taking some linear combination of all neurons in the previous layer. Each neuron's activation is multiplied by some 'weight' value $w_{i,N}^{(L)}$ - the weight that neuron i in the previous layer contributes to the neuron N in layer L, which can be negative or positive. As well, we add an additional bias value $b_N^{(L)}$ to introduce an extra degree of freedom to tune. The raw activation therefore of a neuron can be expressed as:

$$\left[\sum_{i=1}^{R} w_{i,N}^{(L-1)} a_i^{(L-1)}\right] + b_N^{(L)}$$

However, recall that neuron values are bounded from 0 to 1. To ensure activation values stay within this range, we put the raw activation value into an activation function, in our case, the sigmoid function. This smoothly maps the set of real numbers to the interval $(0, 1)$, like an analog version of the unit step, as shown in Figure 3 below:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

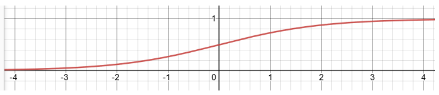


Figure 3: The sigmoid function, plotted in Desmos.

Therefore the value of any neuron can be expressed as

$$a_N^{(L)} = \sigma\left(\left[\sum_{i=1}^{R} w_{i,N}^{(L-1)} a_i^{(L-1)}\right] + b_N^{(L)}\right).$$

To reduce the complexity for the computer, we can combine the weights of all neurons in a layer into a matrix. Since all activations in layer L are constructed as a linear combination of all activations of the previous layer, we can simply "stack" these below the last. Therefore, to find the activations of all N neurons in layer L, we can write the result in terms of a matrix-vector product:

$$\begin{bmatrix} a_1^{(L)} \\ a_2^{(L)} \\ \vdots \\ a_N^{(L)} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1,1}^{(L-1)} & w_{2,1}^{(L-1)} & \cdots & w_{R,1}^{(L-1)} \\ w_{1,2}^{(L-1)} & w_{2,2}^{(L-1)} & \cdots & w_{R,2}^{(L-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,N}^{(L-1)} & w_{2,N}^{(L-1)} & \cdots & w_{R,N}^{(L-1)} \end{bmatrix}\begin{bmatrix} a_1^{(L-1)} \\ a_2^{(L-1)} \\ \vdots \\ a_R^{(L-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(L)} \\ b_2^{(L)} \\ \vdots \\ b_N^{(L)} \end{bmatrix}\right)$$

where the sigmoid function is applied elementwise to the resultant vector inside its brackets.

The Cost Function

To train the network, we then feed it small sets of training data with known answers. Then we compare these answers to the network's output, quantifying how close these input-output pairs are using a cost function. The cost function selected for a particular network is usually selected based on its objectives. For classification networks like ours, it is typical to use a cross-entropy cost function [8]. Suppose we have N output neurons in the output layer L, with activations denoted with a and ideal activations T. The cost of the entire network is then the sum of the cross entropy of each output neuron:

$$C = -\sum_{i=1}^{N}\left[T_i^{(L)}\ln(a_i^{(L)}) + (1-T_i^{(L)})\ln(1-a_i^{(L)})\right]$$

Note that we consider $\ln(0) = -\infty$.
Our ideal activations will only be either 0 or 1, and we can plot the resultant cost against the neuron activation:

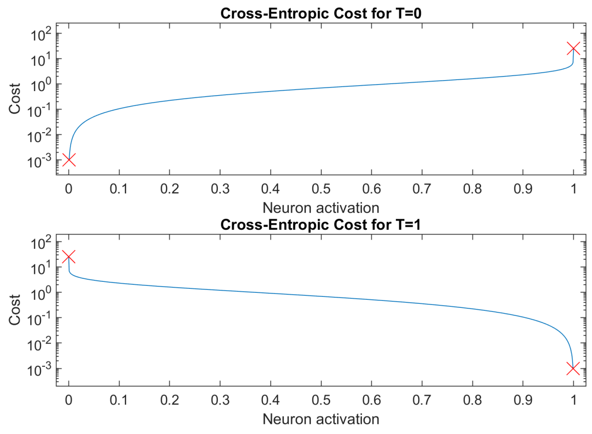


Figure 4: The cross-entropy value of a neuron on a log scale. Note the drastic increase as the expected and observed activation diverge.

The cost function becomes useful when we take the average cost over many examples in a dataset, and compare trends in the average cost as the model is trained. A decreasing cost generally indicates that a neural network is properly learning and becoming more accurate in its outputs. Thus the objective of training a neural network can be abstracted down to a single goal: minimizing (optimizing) the cost function of any valid input to the system.

Decomposition of the Cost Function

A neural network's cost function quantifies the error in the system, taking the results and 'accuracy' of an output given some input. Consider the term in the cost function that quantifies the error in a single output neuron at output index i:

$$C_i = -\left[T_i^{(L)}\ln(a_i^{(L)}) + (1-T_i^{(L)})\ln(1-a_i^{(L)})\right]$$

For simplicity, let the ideal activation for this neuron be represented by some constant T. To expand the activation term, recall that the activation of a neuron is given by the sigmoid of a certain un-normalized quantity, which we will denote $Q_i^{(L)}$, and this quantity is given by the weight-activation product plus some bias:

$$\begin{aligned}
C_i &= -\left[T_i \ln(a_i^{(L)}) + (1-T_i)\ln(1-a_i^{(L)})\right] \\
a_i^{(L)} &= \sigma(Q_i^{(L)}) \\
Q_i^{(L)} &= (w_{1,i}^{(L-1)} a_1^{(L-1)} + w_{2,i}^{(L-1)} a_2^{(L-1)} + \cdots + b_i^{(L)})
\end{aligned}$$

If we were to expand each of the activations in layer $L-1$ recursively, we would notice that all weights and biases of the system in the layers before the output layer are accounted for. Therefore, we can conclude that the cost function of a neural network is a function of all of its weights and biases[1], and ONLY its weights and biases - when the cost function is decomposed, only functions of the weights and biases remain. Additional parameters that do not directly affect the network such as training rate are called *hyperparameters.*

For posterity, suppose we have a system, with L layers, where layer $L-1$ has N neurons and layer L has R neurons. The neuron counts in other layers depend on the system, but are accounted for within the ellipses. Then we can define the cost function, specifically its input parameters, as $C(w_{1,1}^{(1)}, \ldots\, w_{N,R}^{(L)},\, b_1^{(1)}, \ldots\, b_R^{(L)})$

It follows that the task of minimizing the cost function is simply a high-dimensional optimization problem - the goal is to find an optimum sets of weights and biases to minimize the cost, such that the network reliably provides acceptable output.

Deriving Components of the Cost Gradient

Recall the cost function is, in the input-agnostic limit, dependent only on the weights and biases of the system. We can compute the gradient of the cost function to see how these parameters affect the cost, and the optimal changes to the parameters such that the cost approaches some local minimum. However, as we consider weights earlier in the system, calculating their partial derivative becomes extremely computationally difficult; we will derive a much simpler method for calculating these values.

Suppose we want to find the partial derivative of this cost function with respect to output neuron's activation $a_j^{(L)}$ - that is, neuron j in the output layer L. This is not a component of the gradient vector but rather an intermediate quantity from which we can derive the partial of a weight or bias in that neuron. We can un-nest the cost of this neuron by starting with the entire cost function $C = C_1 + C_2 + \cdots + C_j + \cdots + C_N$. However, notice that since the weight will only affect the contribution that neuron j makes to the cost, we can discard all other neurons. We can expand C_j further:

$$C_j = -[T\ln(a_j^{(L)}) + (1-T)\ln(1-a_j^{(L)})]$$
$$\text{where } a_j^{(L)} = \sigma(Q_j^{(L)}), \qquad T \text{ held constant}$$

The dependency tree in Figure 5 becomes clear:

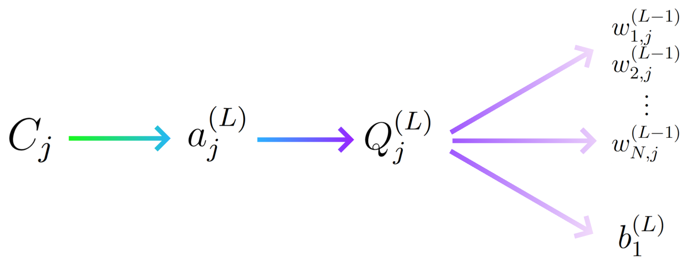

$$\begin{tikzcd}
 & & & w_{1,j}^{(L-1)} \\
 & & & w_{2,j}^{(L-1)} \\
 & & & \vdots \\
C_j \arrow[r] & a_j^{(L)} \arrow[r] & Q_j^{(L)} \arrow[ruuu] \arrow[r] \arrow[rd] & w_{N,j}^{(L-1)} \\
 & & & b_1^{(L)}
\end{tikzcd}$$

Figure 5: The chain tree of the cost function. The weights and biases are not relevant yet.

To help us later generalize these calculations to deeper layers, we can define a quantity $\delta_k^{(l)}$, which represents the error in neuron k of layer l. We will leverage this quantity to make finding partials significantly easier. This error is the partial derivative of the cost function with respect to said neuron's pre-sigmoid activation:

$$\delta_k^{(l)} = \frac{\partial C}{\partial Q_k^{(l)}}$$

Given the dependency tree above, it is clear that the error $\delta_j^{(L)}$ in the neuron we want to find, neuron j in output layer L, can be found by simply going down the chain:

$$\begin{aligned}
\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial (Q_j^{(L)})} &\Longrightarrow \frac{\partial C}{\partial \sigma(Q_j^{(L)})} \frac{\partial \sigma(Q_j^{(L)})}{\partial (Q_j^{(L)})} \\
&\Longrightarrow \frac{\partial C}{\partial \sigma(Q_j^{(L)})} \cdot \sigma'(Q_j^{(L)})
\end{aligned}$$

Now we just need to find the partial of C with respect to the activation $a_j^{(L)}$. For simplicity, we can represent $Q_j^{(L)}$ as Q and $a_j^{(L)} = \sigma(Q_j^{(L)})$ as $\sigma(Q)$. The derivation follows:

$$\frac{\partial C}{\partial \sigma(Q)} = \frac{T}{\sigma(Q)} - \frac{1-T}{1-\sigma(Q)}, \qquad \frac{\partial \sigma(Q)}{\partial Q} = \sigma'(Q)$$

[1]The cost function of a particular output is also dependent on what specific input is used, but we generalize this to the average.

$$\frac{\partial C}{\partial Q} = \frac{\partial C}{\partial \sigma(Q)} \cdot \frac{\partial \sigma(Q)}{\partial Q}$$
$$\frac{\partial C}{\partial Q} = \left[\frac{T(1-\sigma(Q)) - \sigma(Q)(1-T)}{\sigma(Q)(1-\sigma(Q))}\right] \sigma'(Q)$$
$$\frac{\partial C}{\partial Q} = \left[\frac{T}{\sigma(Q)} - \frac{1-T}{1-\sigma(Q)}\right] \sigma'(Q)$$
$$\frac{\partial C}{\partial Q} = \left[\frac{T - \cancel{T\sigma(Q)} - \sigma(Q) + \cancel{T\sigma(Q)}}{\sigma(Q)(1-\sigma(Q))}\right] \sigma'(Q)$$
$$\frac{\partial C}{\partial Q} = \frac{(T-\sigma(Q))(\sigma'(Q))}{\sigma(Q)(1-\sigma(Q))}$$
$$= \frac{(T-\sigma(Q))(\sigma'(Q))}{\sigma'(Q)}$$
$$\frac{\partial C}{\partial Q} = (T-\sigma(Q)) \cdot \frac{\cancel{\sigma'(Q)}}{\cancel{\sigma'(Q)}}$$
$$\frac{\partial C}{\partial Q} = (T-\sigma(Q)) = (T-a)$$

Recall this partial is equal to the error in any neuron in the outer layer, if we apply it to that specific neuron. This is a beautiful and powerful result[2] - to summarize our findings here, we can express the error in an output-layer neuron j with the expression $\delta_j^{(L)} = (T - a_j^{(L)})$.

We can expand this into a vector, for neurons 1 to j in the output layer:

$$\begin{bmatrix} \delta_1^L \\ \delta_2^L \\ \vdots \\ \delta_j^L \end{bmatrix} = \begin{bmatrix} (T - a_1^{(L)}) \\ (T - a_2^{(L)}) \\ \vdots \\ (T - a_j^{(L)}) \end{bmatrix}$$

To extend this to layers further inside the network, we can apply the method of backpropagation - solving for the error of a layer given quantities from its successor. In [9], Nielsen shows that the error vector of a layer l can be calculated based on the transpose of the weight matrix of the next layer $l+1$, the error vector of the next layer, and the derivatives of the sigmoid of Q_i, as shown below:

$$\begin{bmatrix} \delta_1^l \\ \delta_2^l \\ \vdots \\ \delta_k^l \end{bmatrix} = \left[\begin{bmatrix} w_{1,1}^{(l+1)} & w_{2,1}^{(l+1)} & \dots & w_{k,1}^{(l+1)} \\ w_{1,2}^{(l+1)} & w_{2,2}^{(l+1)} & \dots & w_{k,2}^{(l+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,j}^{(l+1)} & w_{2,j}^{(l+1)} & \dots & w_{k,j}^{(l+1)} \end{bmatrix}^T \begin{bmatrix} \delta_1^{l+1} \\ \delta_2^{l+1} \\ \vdots \\ \delta_j^{l+1} \end{bmatrix} \right] \odot \begin{bmatrix} \sigma'(Q_1^{(l)}) \\ \sigma'(Q_2^{(l)}) \\ \vdots \\ \sigma'(Q_k^{(l)}) \end{bmatrix}$$

This equation is quite simple in practice. Taking the transpose of the weight matrix essentially "reverses" the connections between layers - we use this to weight the error of each term in the next layer, to calculate how they contribute to the error in the current layer. This matrix-vector product results in a vector with k elements. Then, we multiply elementwise with the derivative of the sigmoid function at the pre-sigmoid activation Q_i.

In the same publication, Nielsen shows two more rather intuitive equations required to derive the entire gradient. Once we obtain all neuron error values δ through backpropagation, we can calculate the partial derivatives of every weights and bias in the network in a single step. The partial derivative of cost with respect to a neuron's bias is equal to the error in that neuron:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

The partial derivative of cost with respect to any weight is given by the following equation:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

Intuitively, this partial is simply equal to the activation of the neuron that the weight serves as a multiplier for, times the error of the neuron that the weight affects.

Approach

Introducing Gradient Descent

To optimize the cost function, we want to find a local minimum. While in typical optimization problems, the goal is to find a global minimum, Choromanska et. al showed that networks of sufficient size (of which our network will be) exhibit local minima of roughly equal quality; thus optimizing to any local minimum will tend to result in a well-performing network [10].

To optimize a multivariate problem - recall that the cost function is an extremely high-dimensional function - the gradient operator is used. The gradient of a function gives the direction and relative magnitude of 'steps' in which moving the input variables would give the greatest positive increase in the function value. It follows that we can iteratively calculate the cost of an input, then adjust the weights and biases of the network based on the *negative* gradient of the cost function (scaled by some constant training rate η), in order to approach a local minimum. This is known as the method of *gradient descent*.

Consider a dataset of S training samples. Instead of calculating the gradient for each of S samples then applying the mean gradient, we randomly subdivide S into B subsets called mini-batches. Each mini-batch contains S/B samples, which we can compute the average cost function across, then apply gradient descent using this average to the network. We can repeat this process with a new batch up to B times. This heuristic method is known

[2]It is not coincidental that these terms cancel - the cross-entropy cost function is specifically designed to cancel when chained with the derivative of the sigmoid.

as the method of the method of *mini-batch* stochastic gradient descent, and has been shown to yield results of similar (if not better) quality to that of standard gradient descent [11]. The average cost over B input vectors b in a mini-batch can be found by taking the sum of each input's cost as:

$$C = -\frac{1}{B}\sum_{B}\sum_{i=1}^{N}\left[T_i^{(L)}\ln(a_i^{(L)}) + (1 - T_i^{(L)})\ln(1 - a_i^{(L)})\right]$$

The Backpropagation Algorithm

By putting the equations derived in the previous section together, we can formulate an algorithm for backpropagation in order to implement it into our network. Backpropagation is executed as follows:

1. **Input phase:** We set up the vector of input neurons for the network to operate on.
2. **Forward propagation phase:** We calculate the activation values for each neuron, given by the weights and biases. It helps to also store the pre-sigmoid activations Q for use later.
3. **Output error phase:** We calculate the error vector for the output layer, $\left[\delta_j^L\right]$.
4. **Backpropagation phase:** We iteratively calculate the error vector for each predecessor layer, and stop at the first hidden layer, as the the input layer cannot be modified.
5. **Extraction phase:** We independently calculate the partial derivative of the cost function with respect to each weight and bias in the system, but do not apply it yet.
   Repeat steps 1-5 over an entire mini-batch, storing each partial of each parameter.
6. **Descent phase:** We take the mean of all partial derivatives for each weight and bias, and apply these changes to the parameters, scaled by some training factor η.
   Repeat steps 1-6 for each mini-batch, over the entire dataset.

Executing steps 1-6 for the entire dataset is called one *epoch*. We then randomize the training data into new mini-batches and repeat the process, completing several hundred epochs to train the network.

Implementation

To implement the network, we use the NumPy Python library, specifically for its linear algebra capabilities. Note that only critical methods are described in this report; the full written source code is linked after the References section. We start by defining a constructor that defines a 4-layer neural network and initializes its weight matrices and bias vectors as Numpy arrays with normally-distributed initial values, with appropriate sizes:

```
class NeuralNetwork4:
 def __init__(self, inputNeurons,
  h1Neurons, (etc...))
  self.inputNeurons = inputNeurons
  self.h1Neurons = h1Neurons
  self.h1w = rng.normal(MEAN, STDDEV,
   (h1Neurons, inputNeurons))
  self.h1b = rng.normal(MEAN, STDDEV,
   (h1Neurons,))
  self.inputA = np.zeros((inputNeurons,))
  self.h1A = np.zeros((h1Neurons,))
   (etc...)
```

Create a feedforward method that accepts an input vector, then calculates each layer's activations (note the @ operator performs matrix multiplication):

```
def feedforward(self, inputVec):
 self.inputA = inputVec
 self.h1Raw = (self.h1W @
  self.inputA) + self.h1B
 self.h1A = sigmoid(self.h1Raw)
 (etc...)
 return self.outputA
```

Create a backpropagation method that calculates the error in each layer:

```
def backpropagateError(self)
 self.outputE = (self.outputT-self.outputA)
 self.h1E = (self.outputW.transpose()
  @ self.outputE) * dSigmoid(self.h2Raw)
 (etc...)
```

Create methods that calculate and return the weight and bias error vectors, respectively:

```
def getWeightGradient(self)
 weightGradientH1 =
  self.inputA * self.h1E[:, np.newaxis][3]
 (etc...)
def getBiasGradient(self)
 biasGradientH1 = self.h1E * self.h1B
 (etc...)
```

And methods that apply minibatch-mean error vectors to tune the parameters of the network, where **x**Grad is a list of the layer gradients:

[3]This syntax transposes a vector in Numpy. We use it to take advantage of array broadcasting and create a meshgrid-like sum matrix over two differently-sized vectors.

```
def adjustBiases(self, bGrad, eta)
 self.h1B -= (bGrad[0])*eta (etc...)
def adjustWeights(self, wGrad, eta)
 self.h1W -= (wGrad[0])*eta (etc...)
```

These are the only numerical functions required to implement the network. We then write wrapper code that runs the backpropagation algorithm, across multiple layers of nested loops:

1. Execute feedforward and backpropagation for each input in a minibatch, calculate and store the gradients, then apply the mean gradient to the network at some training rate.

2. Perform the above for each minibatch in the training dataset, recording the average cost and accuracy over the epoch.

3. Perform the above for each epoch (usually 100-300). Log the system parameters (weights and biases) to file and randomize the mini-batches after each epoch.

When done training, we finally test the network using the test dataset - 10,000 images the network has not been exposed to yet - and measure its accuracy using a confusion matrix. Each result increments an entry in the matrix, with the row corresponding to the ground-truth label of the test data and the column corresponding to the label the network assigned it.

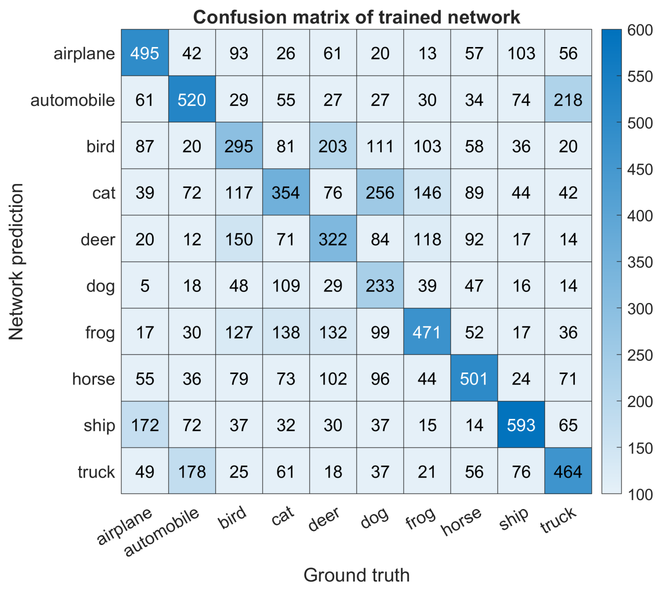
Results

To train the network, we must select sets of hyperparameters to define the network's structure and training process. For our network, the hyperparameters include the number of neurons in hidden layers H_1 and H_2, the training rate η, the mini-batch size M, and the number of epochs E to train for. Unfortunately, selection of hyperparameters is usually trial-and-error [12], so a solution cannot be known to be optimal - however, we can compare the results of each trained network relative to the others. Empirically, the mini-batch size M was found to have no effect on training speed/performance, so it was held constant at 100 for all trials. Results of some selected configurations are compared in Table 1.

Table 1: Hyperparameter selection and results.

Index	H_1	H_2	η	E	Elapsed	Test accuracy
1	2	2	0.01	1	17s	10.17%
2	120	120	0.01	100	2h48m	29.42%
3	120	120	0.1	100	3h14m	39.09%
4	80	80	0.01	200	4h09m	33.94%
5	60	60	0.2	300	5h19m	42.04%
6	150	40	0.25	150	6h24m[4]	42.48%

As it produced the best test accuracy, we will analyze the results of Trial 6. Its confusion matrix is shown in heatmap format in Figure 6:



Network prediction \ Ground truth	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	495	42	93	26	61	20	13	57	103	56
automobile	61	520	29	55	27	27	30	34	74	218
bird	87	20	295	81	203	111	103	58	36	20
cat	39	72	117	354	76	256	146	89	44	42
deer	20	12	150	71	322	84	118	92	17	14
dog	5	18	48	109	29	233	39	47	16	14
frog	17	30	127	138	132	99	471	52	17	36
horse	55	36	79	73	102	96	44	501	24	71
ship	172	72	37	32	30	37	15	14	593	65
truck	49	178	25	61	18	37	21	56	76	464

Figure 6: The confusion matrix of the Trial 6 network, evaluated over the CIFAR-10 test dataset.

After each epoch, the average epoch cost and accuracy (over the training dataset) and the test dataset accuracy[5] (at the current network state) were logged. These are plotted in Figure 7:

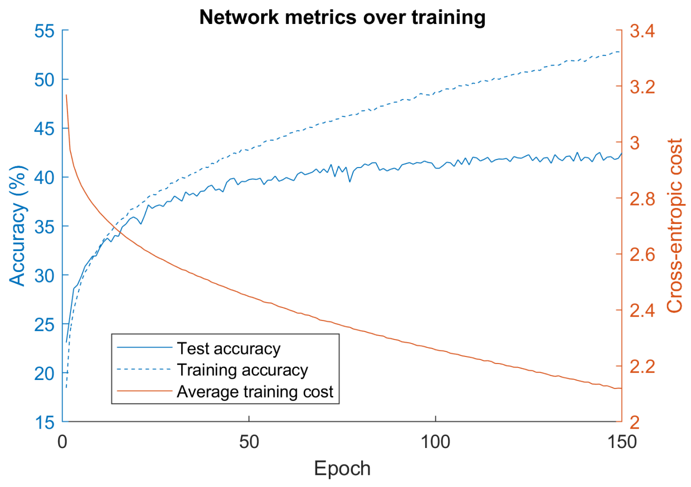


Figure 7: The training cost, training accuracy, and test accuracy over the training process.

[4]Note the machine was in use during this trial and a significant slowdown in training was experienced.

[5]The state of the network (its weights and biases) was logged to file after each epoch, then loaded later to get the test accuracy at each end-of-epoch state.

Discussion

Training neural networks is a difficult task - a significant amount of experimentation is required, and many runs must be completed with varying hyperparameters to find the best training configuration for a given problem.
In general, using the results from Table 1, several trends are visible. Primarily, using a more complex network (where the neuron counts H_1 and H_2 are larger) results in a more accurate network, but increases training time per epoch (not listed). Simplistic networks, such as that of trial 1, generally result in accuracy no better than random chance (for CIFAR-10, this is 10%). Naturally, increasing the number of epochs E linearly increases the training time, but also increases the test accuracy of the resultant network. The training rate η affects the rate of convergence of the network. It can be seen that at low values of η, the network may not fully converge to a local minimum within the specified number of epochs; trials 2 and 3 directly show this correlation. Given more epochs of runtime, it is likely these networks would converge to a similar accuracy.

Reading the confusion matrix (shown in Figure 6) of the network trained in Trial 6, we can make some immediate observations:

- The main diagonal of the confusion matrix, where the ground truth matches the network prediction, represents correct predictions.
  - The trace of this matrix (4,248) therefore gives the number of correct predictions by the network.
- Noise is present, concentrated in the centre of the matrix, representing false predictions. Interestingly, this often occurs for similar labels:
  - For the ground truth label of "automobile", the network made 520 correct predictions, but 178 erroneous predictions under the label "truck".
  - For the ground truth label of "dog", the network made 233 correct predictions, but 256 incorrect predictions under the label "cat".

This does indeed suggest that the network is "learning" - specifically, its weights and biases become tuned over the training process such that the output of the network is more likely to align with the expected category label! To better quantify the network during the training process, we can look at the results of Figure 7:

- The training accuracy increases rapidly over the first roughly 20 epochs, then increases about linearly.
- The training cost is inversely proportional to the training accuracy, decreasing rapidly at first, then at a decaying rate thereafter.
  - This makes sense - the network's only goal is to decrease the cost function, which quantifies the accuracy in the network (and ideally increases the test accuracy).
- The test accuracy is identical to the training accuracy during its rapid increase, but stagnates at about 40%. We can reasonably conclude the test accuracy has reached a local minimum, and thus the network is sufficiently trained[6].

Interestingly, the training accuracy continues to increase even as the test accuracy levels out. This is due to a phenomenon known as *overfitting*, where the network becomes tuned too tightly towards the specific training images, rather than the qualities they possess [13]. When a network exhibits overfitting, it struggles with generalizing what is has learned to new images. Overfitting is often caused by a network too complex for the problem it is solving (in our case, using unreasonably large hidden layers) or by training over too many epochs - however, this cannot always be solved by changing the hyperparameters.

While the implementation of advanced techniques is outside the scope of this project, it is relevant to discuss their mechanisms and use in the context of our results. One state-of-the-art method to prevent overfitting is known as *dropout* [14]. Recall that while networks with a high number of parameters can be extremely powerful, their complexity can exacerbate overfitting. Dropout applies the condition that each training case randomly selects some nodes in the network to ignore, as shown in Figure 8; during testing, all nodes are used.

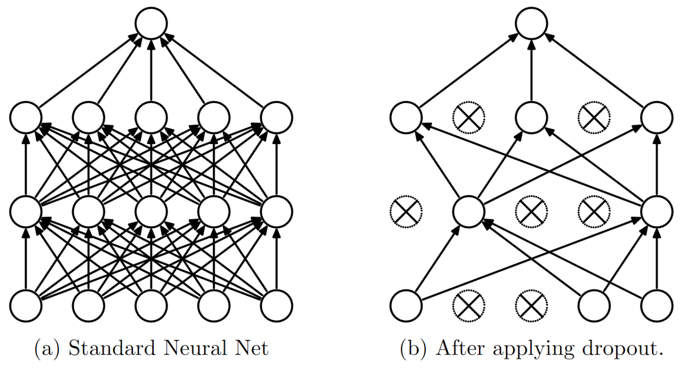


Figure 8: Network flow compared to dropout network flow. Source: [14]

[6]Training should be stopped once the test accuracy stops increasing, as further training on the same data can lead to overfitting, described above.

Using dropout has several major benefits:

- The computational complexity is drastically reduced.
- The network is simplified in the context of training.
- The network remains complex in the context of testing.
- The network must train a random subset of its nodes for each training example - the probability of "saturated" neurons whose pre-sigmoid activation values are very large or very small is greatly reduced, as no neuron can rely on others.
- The resultant network is equivalent to training very many simple networks, and taking the average result over all output layers, which greatly increases accuracy.

Dropout has been shown to increase the performance of neural networks almost universally. While this paper focuses on the implementation of a very simple network, dropout is a logical next step in improving the performance of our network.

Nonetheless, the results of the network trained under Trial 6's parameters are significant considering the relative simplicity of our approach. The results obtained after 150 epochs are likely very close to the maximum performance that this network could be trained to; this can be seen in Figure 7 in the asymptotic behaviour of the test accuracy in the latter half of training. In fact, the network should not be trained much more, in order to avoid overfitting and the test accuracy collapse that follows.

Conclusion

In this work we have explored the structure, mechanisms, and applications of perceptron neural networks. We have derived the equations that drive their feedforward and backpropagation calculations. We have discussed standard training processes used to apply backpropagation to the network. Using bespoke Python and NumPy code, we have trained several networks using different combinations of hyperparameters, and we have used specific metrics to compare and contrast their results. Finally, we have explored possible next steps to improve our networks.

The results we obtained, despite using relatively simple methods, were convincing. At most, about 42-43 percent accuracy was achieved on novel images - the network was able to generalize what it learned to new examples.Although additional techniques and/or computational power are required to achieve near-human or super-human performance on this dataset, this extension of our work is by no means out of reach. $\square$

References

[1] A. Krizhevsky, “Learning multiple layers of features from tiny images,” 2009.

[2] *Cifar-10 and cifar-100 datasets.* [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[3] A. Krizhevsky, “Convolutional deep belief networks on cifar-10,” 2010.

[4] H. M. D. Kabir, “Reduction of class activation uncertainty with background information,” 2023. arXiv: 2305.03238 `[cs.CV]`.

[5] W. S. McCulloch and W. H. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bulletin of Mathematical Biology*, vol. 52, pp. 99–115, 2021.

[6] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para* (Report: Cornell Aeronautical Laboratory). Cornell Aeronautical Laboratory, 1957. [Online]. Available: https://books.google.ca/books?id=P%5C_XGPgAACAAJ.

[7] I. Neutelings, *Neural network diagrams*, Apr. 2023. [Online]. Available: https://tikz.net/neural_networks/.

[8] L. Li, M. Doroslovački, and M. H. Loew, “Approximating the gradient of cross-entropy loss function,” *IEEE Access*, vol. 8, pp. 111 626–111 635, 2020. DOI: 10.1109/ACCESS.2020.3001531.

[9] M. A. Nielsen, Dec. 2019. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html.

[10] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, “The loss surface of multilayer networks,” *CoRR*, vol. abs/1412.0233, 2014. arXiv: 1412.0233. [Online]. Available: http://arxiv.org/abs/1412.0233.

[11] M. Li, T. Zhang, Y. Chen, and A. J. Smola, “Efficient mini-batch training for stochastic optimization,” in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’14, New York, New York, USA: Association for Computing Machinery, 2014, pp. 661–670, ISBN: 9781450329569. DOI: 10.1145/2623330.2623612. [Online]. Available: https://doi.org/10.1145/2623330.2623612.

[12] M. Shen, J. Yang, S. Li, A. Zhang, and Q. Bai, “Nonlinear hyperparameter optimization of a neural network in image processing for micromachines,” *Micromachines*, vol. 12, no. 12, 2021, ISSN: 2072-666X. DOI: 10.3390/mi12121504. [Online]. Available: https://www.mdpi.com/2072-666X/12/12/1504.

[13] Jun. 2023. [Online]. Available: https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/.

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

A repository containing all code required to reproduce results is available at https://github.com/lucs100/soplenet.