# Multi-Layer Perceptron for Multi-Class Classification

TopiasDeepLearning

December 2022

**Abstract**

A multi-layer perceptron is a group of connected logistic regression units (MLP). During training, MLP employs the supervised learning technique known as backpropagation. This study will discuss the creation of the complete method for training an MLP for multi-class classification. The development of the multi-layer perceptron method will also be presented. The backpropagation algorithm will be covered in order to comprehend how a neural network can be taught. The whole theory presented in these sections will be converted into Python code as part of this paper's main goal. The theory-based development of a multi-layer perceptron in code is demonstrated in this study. The application of concepts like adaptive learning rates, momentum, mini-batch gradient descent, and bias correction improves training effectiveness.

## 1 Introduction

A collection of interconnected logistic regression units is identified as a multi-layer perceptron [19] (MLP). A minimum of three different nodes arranged in layers compensate an MLP: the input layer, the hidden layer, and the output layer. Each node, with the exception of the input nodes, is a neuron that employs a non-linear activation function.

Backpropagation is a supervised learning method that is used by MLP during training [21]. MLP differs from a linear perceptron due to its numerous layers and non-linear activation. It can discriminate between data that cannot be separated linearly. When multilayer perceptrons have just one hidden layer, they are sometimes made reference herein as ”vanilla” neural network models.

The development of the full procedure for training an MLP for multi-class classi-

fication will be covered in this paper. Both the algorithm's theoretical underpinnings and its Python code implementation will be discussed. I have developed a program that will improve comprehension of the theories discussed in this paper. In order to discriminate between three provided classes in an artificial dataset, the project is a supervised learning classification problem that makes use of a multi-layer perceptron.

# 2 Methodologies

The specifics of the project will be included in this section. It includes an explanation of the research, the math, and the Python libraries used. Also highlighted will be the construction of the multi-layer perceptron algorithm.

## 2.1 Cost Function

The categorical cross entropy loss function is the cost/objective/error function used in multi-class classification [24]. A deep learning model seeks to minimize the discrepancy between predicted and actual values of the target variable. It makes sense to reduce this inaccuracy in order to train the model because the higher the difference, the larger the mistake will be.

It is for this reason that deep learning programmers begin by learning how to make predictions. It is because obtaining the forecasts is necessary before determining the error, which is the discrepancy between the target and the predictions.

### 2.1.1 Multi-Class Classification

Classification into multiple categories [5] is the premise for this paper. The observed random variable derives from a categorical distribution when there are several potential outcomes. A die roll can be used as an analogue for this type of classification issue in real life. The outcome of a die roll must be a number between 1 and 6 (inclusive).

### 2.1.2 Maximum Likelihood Estimation for a Die Roll

The following is the format of the maximum likelihood estimation [15] for a potentially biased die: Let us say a dice is rolled several times, and the observed results are: $\{6, 2, 2, 5, 3\}$

It is practical to depict these findings using the indicator matrix $t(n, k)$. This can be viewed as a matrix of binarized values that has been one-hot encoded. $t(n, k) = 1$ if $k$ is rolled on the $n^{th}$ roll, otherwise $t(n, k) = 0$.

**Example.** *For the above series of observations the indicator matrix for the first few rolls are shown as:*

$$t(1,6) = 1,$$
$$t(2,2) = 1,$$
$$t(3,2) = 1,$$
$$etc.$$

There are no model predictions in this example because this is a die, hence there are no $Y$'s. The probabilities of rolling each of the $K$ values are presented as an alternative. They are referred to as $w_1, w_2, \cdots, w_6$.

The likelihood is then expressed as follows:

$$L = \prod_{n=1}^{N} \prod_{k=1}^{6} w_k^{t_{nk}} \tag{1}$$

### 2.1.3 Categorical Cross Entropy Loss

It is important to keep in mind that the likelihood [12] is simply the product of the probability mass functions (PMF)'s or probability density functions (PDF)'s of the $N$ datapoints. A categorical distribution's PMF is:

$$f(x = i \mid p) = p_i \tag{2}$$

where $(p = p_1, \cdots, p_k)$, $p_i$ represents the probability of seeing element $i$

and $\sum_{i=1}^{k} p_i = 1$

It is now clear how this could be used with a neural network. The predictions for each of the targets $t(n,k)$ are provided by the output probabilities $y(n,k)$. $t(n,k)$ is more likely to be $1$ if $y(n,k)$ is large.

**Example.** *If $y(n,k)$ is large, then $t(n,k)$ is more probable.*

The only difference between this and the die roll example is that a more generic $K$ is used in place of the number $6$. The probability is not merely a fixed $w_k$, but $y_{nk}$ for each category.

$$L = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}} \tag{3}$$

The loss is the negative log likelihood [25]. Therfore the categorical cross entropy loss function is simply the negative log likelihood, given that the distribution of

the targets is categorical. It is crucial to understand that decreasing the negative log likelihood is the same as increasing the log likelihood. As the negative sign is tedious and essentially superfluous, it is crucial to remove it entirely so that it does not continue on each line of the derivation that will be calculated in the following section of this paper.

$$\text{Categorical Cross Entropy Loss} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \log y_{nk} \tag{4}$$

**Example.** *If $y(n,k)$ is more wrong, the loss should be larger.*

*If $y(n,k)$ is more right, the loss should be smaller.*

*Consider just $1$ sample, so there is no index for "$n$". Then the loss will just be:*

$$Loss = -\sum_{k=1}^{K} t_k \log y_k \tag{5}$$

- *If it is exactly right: $t_k = 1$ and $y_k = 1$. Substituting those values into the expression yields: $-1 \times \log 1 = 0$, therefore the minimum loss is $0$.*
- *If there is only $50\%$ probability on the correct target: $-1 \times \log 0.5 = 0.693$.*
- *If there is only $20\%$ probability on the correct target: $-1 \times \log 0.2 = 1.609$.*
- *If there is only $0\%$ probability on the correct target: $-1 \times \log 0 = \infty$.*

*This demonstrates that the loss function performs as intended.*

## 2.2 Logistic Regression with Softmax

The training of a multi-class logistic regression [17] classifier to comprehend how to use these methods in order to create an MLP is covered in this subsection. The following elements are necessary for this derivation:

- The prediction is:
$$y = \textbf{softmax}\left(W^T x\right) \tag{6}$$
- The loss function is the same as Equation 4:
$$-\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \log y_{nk}$$

The objective is to reduce the loss $L$ relative to the weight matrix $W$.

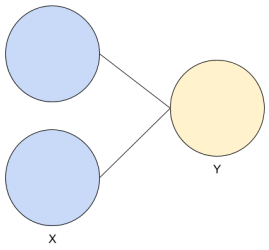


Figure 1: Logistic Regression

### 2.2.1 Remove the negative sign

The next step is to remove the negative sign in order to complete the derivation, as was described in section 2.1.3. This suggests an objective of maximization rather than minimization. Instead of the negative log likelihood, this is simply the model's log likelihood relative to the data. As a result, gradient ascent can be executed rather than descent [11].

Because of the explanations given in section 2.1.3, gradient ascent is the same as gradient ascent. The goal of maximization is to get the cost to converge to the local maximum.

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_{nk} \tag{7}$$

### 2.2.2 Chain Rule in Logistic Regression

To solve for $W$, the approach is to apply gradient ascent [11]. To accomplish this, we must determine $J$'s gradient with respect to $W$. The gradient ascent algorithm can then be used to move $W$ in the direction of the gradient until it converges to the solution once we have the answer. The fact that there are additional functions makes this difficult. $J$'s expression does not explicitly reference $W$. It is reliant on $y$, $y$ is reliant on $a$, and $a$ is reliant on $W$.

$$a = W^T x \tag{8}$$

$$y = \textbf{softmax}\left(W^T x\right)$$

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_{nk}$$

The chain rule of differentiation [14] can be used to determine the derivative of $J$ with respect to $W$. Finding the derivative of composite functions can be especially helpful using this method.

$$\begin{aligned} J = f(z), z = g(y), y = h(x) \\ \frac{\mathrm{d}J}{\mathrm{d}x} = \frac{\mathrm{d}J}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} = f'(z)g'(y)h'(x) \end{aligned} \tag{9}$$

Multi-Class logistic regression follows the same logic in the following way:

$$\frac{\partial J}{\partial W_{ik}} = \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial W_{ik}} \tag{10}$$

Take note that while there is a $k$ on the left, there is a sum over $k'$ on the right. The next section addresses the reason why $k$ and $k'$ both appear on the right.

### 2.2.3 Dummy Variables

Outside of the sum, nothing involving the summation index can exist. It is impossible to remove the variables from the summing. They are referred to as dummy variables [6]. Dummy variables can be given any name; $i$ $j$, $k$, etc. what matters is that they cannot exist independently of the summation.

Returning to the previously defined math equation, the dummy index or variable utilized for the summing is $k'$. The left-side $k$ does not stand for a dummy index. Instead, it represents the weight $W$'s output index. Consequently, $i$ stands for the input index and $k$ for the output index. Notice that $n$ is a dummy variable as well.

### 2.2.4 Switching from k' to k

$\frac{\partial y_{nk'}}{\partial a_{nk}}$ is the most significant derivative in this case. The boundary is the point at which the notation changes from $k'$ to $k$. In this section, the significance of changing from form $k'$ to $k$ is discussed.

Simply said, $y$ is a softmax [8] of $a$, as referenced in Equation 8. This softmax function has a non-trivial characteristic that explains why the notation is switched.

If actual numbers are used, it is simpler to determine this. Consider a system with three output activations, $a1$, $a2$, and $a3$. Three equivalent output probabilities are

then generated from this: $y1$, $y2$, and $y3$. The softmax function can be used to compute the probability.

$y1$ depends on $a1$, $a2$, as well as $a3$. Similar to $y1$; $y2$, and $y3$ also depends on each of the $a$'s.

$$y_1 = \frac{exp(a_1)}{exp(a_1) + exp(a_2) + exp(a_3)}$$
$$y_2 = \frac{exp(a_2)}{exp(a_1) + exp(a_2) + exp(a_3)}$$
$$y_3 = \frac{exp(a_3)}{exp(a_1) + exp(a_2) + exp(a_3)}$$

This indicates that there are actually nine separate derivatives to take into account. More crucially, it has been noted that there are instances where the index for $y$ and the index for $a$ coincide and instances where they do not.

$$\frac{\partial y_1}{\partial a_1}, \frac{\partial y_1}{\partial a_2}, \frac{\partial y_1}{\partial a_3}$$
$$\frac{\partial y_2}{\partial a_1}, \frac{\partial y_2}{\partial a_2}, \frac{\partial y_2}{\partial a_3}$$
$$\frac{\partial y_3}{\partial a_1}, \frac{\partial y_3}{\partial a_2}, \frac{\partial y_3}{\partial a_3}$$

As a result, it is illogical to only consider $\frac{\partial y_k}{\partial a_k}$. The instances where $\frac{\partial y_{k'}}{\partial a_k}$ where $k' \neq k$ also needs to be considered.

This essentially means that, in the context of neural networks and logistic regression, the probability at one output node depends on the activations at the other output nodes and, consequently, depends on the weights (each with their own different indices) that are applied to the other output nodes. That makes sense because, in order to obtain a probability, it must first be divided by the sum of all the activations combined.

### 2.2.5 Derivative of the Softmax function

This will be calculated first because it is the only non-trivial derivative that needs to be solved.

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = \frac{\partial}{\partial a_{nk}} exp(a_{nk'}) \left\{ \sum_j exp(a_{nj}) \right\}^{-1}$$

In order to prevent it from interfering with the variables $k'$ and $k$ outside of the total, another another dummy variable, $j$, is utilized.

Both the case where $k' = k$ and the case where $k'$ is not equal to $k$ must be taken into account when calculating this expression's derivative. The first term of this equation will be regarded as a constant if $k'$ is not equal to $k$. Both terms in the equation must be differentiated if $k' = k$.

Consider the case where k' is not equal to k first. The following is the derivative of this expression:

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = exp(a_{nk'})(-1)\left\{\sum_j exp(a_{nj})\right\}^{-2} exp(a_{nk}) \tag{11}$$

It is fortunate that derivatives of neural networks can be discussed in terms of outputs rather than inputs. This equation can be seen as two softmaxes multiplied by one another by rearranging the derivative.

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = (-1)\frac{exp(a_{nk'})}{\sum_j exp(a_{nj})}\frac{exp(a_{nk})}{\sum_j exp(a_{nj})} = -y_{nk'}y_{nk} \tag{12}$$

Next, consider the scenario where $k' = k$. Simply substitute $k'$ with $k$ in this situation because they are identical.

$$\frac{\partial y_{nk}}{\partial a_{nk}} = \frac{\partial}{\partial a_{nk}} exp(a_{nk})\left\{\sum_j exp(a_{nj})\right\}^{-1}$$

Now that both terms depend on $a_{nk}$, differentiating the two terms results in:

$$\frac{\partial y_{nk}}{\partial a_{nk}} = exp(a_{nk})\left\{\sum_j exp(a_{nj})\right\}^{-1} - exp(a_{nk})^2\left\{\sum_j exp(a_{nj})\right\}^{-2} \tag{13}$$

As before we can express this in terms of outputs rather than inputs:

$$\frac{\partial y_{nk}}{\partial a_{nk}} = \frac{exp(a_{nk})}{\sum_j exp(a_{nj})} - \frac{exp(a_{nk})^2}{\left(\sum_j exp(a_{nj})\right)^2} = y_{nk} - y_{nk}^2 \tag{14}$$

$$\frac{\partial y_{nk}}{\partial a_{nk}} = y_{nk}(1 - y_{nk}) \tag{15}$$

Now we have both Equation 12 and Equation 14 as follows:

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = -y_{nk'} y_{nk} \ \textbf{if } k' \neq k$$
$$\frac{\partial y_{nk}}{\partial a_{nk}} = y_{nk}(1 - y_{nk}) \ \textbf{if } k' = k$$

It would be convenient to merge these into a single formula and plug these equations into the loss derivative. A highly specific approach is used to express the second derivative. The first occurrence of $y$ is represented as $y'_{nk}$, while the second occurrence is represented by $y_{nk}$. Since $k' = k$, they both provide the same outcome technically. However, the remainder of the derivation is straightforward using this form.

The Kronecker delta function is the necessary tool for this. If $k' = k$, it yields $1$, otherwise it returns $0$.

$$\begin{aligned} \delta_{kk'} &= 1 \ \ if k' = k \\ \delta_{kk'} &= 0 \ \ if k' \neq k \\ \frac{\partial y_{nk'}}{\partial a_{nk}} &= y_{nk'} \left(\delta_{kk'} - y_{nk}\right)) \end{aligned} \tag{16}$$

### 2.2.6 Back to the Original Expression

Now the derivative of the cost function: Equation 10 can be found by differentiating the three functions Equation 8, Equation 6, and Equation 7. The derivative of the softmax has been identified in Equation 16.

$$\begin{aligned} J_{nk'} &= t_{nk'} \log y_{nk'} \\ y_n &= \textbf{softmax}(a_n) \\ a_{nk} &= W^T_{:,k} x_n \end{aligned}$$

$$\frac{\partial J_{nk'}}{\partial y_n k'} = \frac{t_{nk'}}{y_{nk'}} \tag{17}$$

$$\frac{\partial a_{nk}}{\partial W_i k} = x_{ni} \tag{18}$$

Then, simplify after plugging these into the gradient expression:

$$\frac{\partial J}{\partial W_{ik}} = \sum_{n=1}^{N}\sum_{k'=1}^{K}\frac{t_{nk'}}{y_{nk'}}y_{nk'}(\delta_{kk'-y_{nk}})x_{ni}$$

$$\frac{\partial J}{\partial W_{ik}} = \sum_{n=1}^{N}\sum_{k'=1}^{K}t_{nk'}(\delta_{kk'-y_{nk}})x_{ni}$$

**Isolate the delta**

$$\sum_{k'=1}^{K}t_{k'}(\delta_{kk'} - y_k) = \sum_{k'=1}^{K}t_{k'}\delta_{kk'} - \sum_{k'=1}^{K}t_{k'}y_k$$

**The first term**

$$\delta = 1 \text{ only when } k' = k$$

$$\sum_{k'=1}^{K}t_{k'}\delta_{kk'} = 1 \times t_k + 0 \times \text{other terms } = t_k$$

**The second term** Since k is not a dummy variable, it can be used separately from the summation. The target variable for a neural network is represented by t. All but one of the values in t are zeroes. Since they will all be totaled together, it does not matter which of the values is one. Thus, the result will be:

$$\sum_{k'=1}^{K}t_{k'}y_k = y_k\sum_{k'=1}^{K}t_{k'} = y_k \times 1 = y_k$$

**Simplify the result** Plug this back into the gradient [1] equation:

$$\frac{\partial J}{\partial W_{ik}} = \sum_{n=1}^{N}(t_{nk} - y_{nk})x_{ni} \tag{19}$$

### 2.2.7 Vectorize the equations

Use linear algebra to vectorize [4] Equation 19 in order to convert it into numpy code.

$$a = \sum_{i=1}^{D}w_i x_i = w_x^T$$

$$A_{kj} = \sum_{i=1}^{D}W_{kiX_{ij}}$$

### 2.2.8 Matrix form of the vectorized softmax derivative

$$\nabla J = X^T(T - Y) \tag{20}$$

## 2.3 Backpropagation

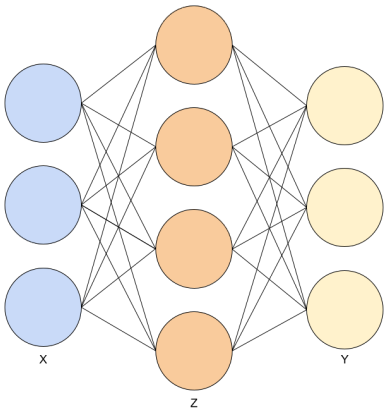


Figure 2: Feedforward Neural Network

To understand how a neural network can be trained, the backpropagation algorithm [21] will be discussed in this subsection. The objective remains the same: construct the cost function, identify the gradients [1], and utilize gradient descent [11] to locate the local minimum. Some of the logic in subsection 2.2 will be applied here.

$$\text{Input} \to \text{Hidden} \quad z = \sigma(W^T x + b) \tag{21}$$

$$\text{Hidden} \to \text{Output} \quad y = \textbf{softmax}(V^T z + c) \tag{22}$$

$$\text{Output} \to \text{Loss} \quad J = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_{nk} \tag{23}$$

### 2.3.1 Chain rule in Neural Networks (Backpropagation)

As before, these equations can be divided into three distinct functions.

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_{nk}$$

$$y_n = \textbf{softmax}(a_n)$$

$$a_{nk} = V_{:,k}^T z_n + c_k$$

The gradients of $V$ and $c$ can also be determined here using the chain rule. To accomplish this, these functions are first divided into three distinct derivatives.

$$\frac{\partial J}{\partial V_{mk}} = \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial V_{mk}} \tag{24}$$

$$\frac{\partial J}{\partial c_{k}} = \sum_{n=1}^{N} \sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial y_{nk'}} \frac{\partial y_{nk'}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial c_{k}} \tag{25}$$

There is no need to re-derive the gradients of the first two equations since they were previously computed in Equation 19. The third derivative is easy to derive.

$$\frac{\partial a_{nk}}{\partial V_{mk}} = z_{nm} \tag{26}$$

$$\frac{\partial a_{nk}}{\partial c_{k}} = 1 \tag{27}$$

**Put it all together** The following derivatives are now obtained for $V_{mk}$ and $c_k$:

$$\frac{\partial a_{nk}}{\partial V_{mk}} = \sum_{n=1}^{N} (t_{nk} - y_{nk}) z_{nm} \tag{28}$$

$$\frac{\partial a_{nk}}{\partial c_{k}} = \sum_{n=1}^{N} (t_{nk} - y_{nk}) \tag{29}$$

### 2.3.2 Vectorize the Gradients of V and c

The gradient of $V$ is given as:

$$\nabla_V J = Z^T (T - Y) \tag{30}$$

The gradient of $c$ has no mathematical expression. Python is used as an alternative to writing the gradient of $c$ as:

```
grad_c = np.sum(T - Y, axis=0)
```

Code 1: Gradient of c

### 2.3.3 Calculating the Gradients of W and b

**Distinguish the activations** It is practical to have a variable that holds the value prior to applying the activation function and represents the activations at each layer. The hidden layer will be represented by $\alpha$ while the output will be represented by $a$.

$$\alpha = W_x^T + b \tag{31}$$

$$z = \sigma(\alpha) \tag{32}$$

$$a = V_z^T + c \tag{33}$$

$$y = softmax(a) \tag{34}$$

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_{nk} \tag{35}$$

Now there are five functions. Six derivatives are to be calculated. The calculation should continue as usual despite this, not stop. These five derivatives are all simply the derivatives of the functions that was previously examined in Equation 24, and Equation 25. Therefore, taking derivatives mechanically is not too difficult. But the conceptual work that has to be done is challenging. In this case, the "Law of Total Derivatives" must be used.

The dummy index $k$ no longer appears outside the sum in the derivatives of $W$ and $b$. The gradients for $V$ and $c$ have $k$ on the outside since those variables were actually indexed by $k$. But $W$ and $b$ are not indexed by $k$.

### 2.3.4 The Law of Total Derivatives

Consider tracking a particle's location over time in a three-dimensional space. Three arbitrary functions are $x(t)$, $y(t)$, and $z(t)$. Assume that $f(x, y, z)$ is a positional function. The following is the derivative of $f$ with respect to time:

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}t}$$

This formula can be made more broad such that $t$ can parameterize any number of variables. The individual terms are summed together in this situation.

$$x_k(t) \text{ for } k = 1 \cdots k$$

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \sum_{k=1}^{K} \frac{\partial f}{\partial x_k}\frac{\mathrm{d}x_k}{\mathrm{d}t} \tag{36}$$

**How do neural networks fit into this?** The objective function is $J$. For all $n$'s and $k$'s, $J$ is a function of $a(n,k)$. $W(d,m)$ parametersizes $a(n,k)$.

**How come?** The weight that moves one input node to one hidden node is $W(d,m)$. All of the output nodes, or more specifically, all of the $K$ outputs, are influenced by this one hidden node.

**Why does this matter?** It implies that all $k$'s potential values must be added together in order to get rid of the annoying dummy variable. You will see that the output nodes now have a double summation, one over $k'$ and one over $k$.

$$\frac{\partial J}{\partial W_{dm}} = \sum_{n=1}^{N}\sum_{k'=1}^{K}\sum_{k'=1}^{K}\frac{\partial J_{nk'}}{\partial y_{nk'}}\frac{\partial y_{nk'}}{\partial a_{nk}}\frac{\partial a_{nk}}{\partial z_{nm}}\frac{\partial z_{nm}}{\partial \alpha_{nm}}\frac{\partial \alpha_{nm}}{\partial W_{dm}} \tag{37}$$

$$\frac{\partial J}{\partial b_{m}} = \sum_{n=1}^{N}\sum_{k'=1}^{K}\sum_{k'=1}^{K}\frac{\partial J_{nk'}}{\partial y_{nk'}}\frac{\partial y_{nk'}}{\partial a_{nk}}\frac{\partial a_{nk}}{\partial z_{nm}}\frac{\partial z_{nm}}{\partial \alpha_{nm}}\frac{\partial \alpha_{nm}}{\partial b_{m}} \tag{38}$$

Plug in the derivatives that were solved in Equation 28, Equation 29 in the gradient for $W$ and $b$. The other three derivatives can be solved as follows:

$$\frac{\partial a_{nk}}{\partial a_{nm}} = V_{mk} \tag{39}$$

$$\frac{\partial \alpha_{nm}}{\partial W_{dm}} = x_{nd} \tag{40}$$

The derivative of $\sigma$ will be different depending on what $\sigma$ represents. whether the sigmoid, tanh or relu functions are used.

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} = z_{nm}(1-z_{nm}) \quad \textbf{if } \sigma(x) = \frac{1}{1+exp(-x)} \tag{41}$$

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} = 1 - z_{nm^2} \quad \textbf{if } \sigma(x) = \textbf{tanh}(x) \tag{42}$$

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} = u(z_{nm}) \quad \textbf{if } \sigma(x) = \textbf{relu}(x)\textbf{, where } \mathbf{u}(\cdot) = \textbf{step function} \tag{43}$$

The gradients of $W$ and $b$ are:

$$\frac{\partial J}{\partial W_{dm}} = \sum_{k=1}^{K}\sum_{n=1}^{N}(t_{nk}-y_{nk})V_{mk}z_{nm}(1-z_{nm})x_{nd} \tag{44}$$

$$\frac{\partial J}{\partial b_{bm}} = \sum_{k=1}^{K}\sum_{n=1}^{N}(t_{nk}-y_{nk})V_{mk}z_{nm}(1-z_{nm}) \tag{45}$$

### 2.3.5 Vectorize the Gradients of W and b

Given that there are many more terms involved, this phase is more difficult than the previous vectorization stage seen in subsubsection 2.3.2. It also involves some element-wise multiplication. By observing the shapes of the matrices involved, one can discover the reasoning behind how it functions.

First, keep in mind that the $z$ terms are just multiplication of elements. As a result, whenever $z_{nm}$ exists, $1 - z_{nm}$ will likewise exist. The indices are therefore the same. This turns into $Z \odot (1 - Z)$, where Z is the full $(N \times M)$ matrix of hidden layer values.

$$Z' = Z \odot (1 - Z)$$

Next, take into account the shapes of all the objects. $t - y$ is of shape $(N \times K)$. $V$ is of shape $(M \times K)$, $z$ is of shape $(N \times M)$, and $x$ is of shape $(N \times D)$. Because none of the inner dimensions match, it is impossible to perform a simple matrix multiply.

Here is the solution. The inner dimension will change to K if V is transposed, and this is a legitimate matrix multiply. It produces a matrix of size (N x M).

$$(T - Y)_{N \times K} V^T_{K \times M} \rightarrow N \times M$$

Z' also has a size of (N x M), allowing for another element-by-element matrix multiplication.

$$\left[(T - Y)V^T\right]_{N \times M} \odot Z'_{N \times M}$$

The final matrix has dimensions (N x M) and x has dimensions (N x D). Since the outcome must be (D x M), a sum over N must be performed, which necessitates that N be the inner dimension.

$$\underset{(D \times M)}{\nabla_W J} = X^T \left\{ \left[(T - Y)V^T\right] \odot Z \odot (1 - Z) \right\}$$

The more general form of this can be written as just $Z'$, which can stand in for any activation function mentioned in Equation 41, Equation 42, and Equation 43

Using ReLU activation:

$$\nabla_W J = X^T \left\{ \left[(T - Y)V^T\right] \odot Z \odot (1 - Z) \right\} \tag{46}$$

Generic form:

$$\nabla_W J = X^T \left\{ \left[ (T - Y) V^T \right] \odot Z > 0 \right\} \tag{47}$$

Again, the bias term's vectorized form cannot be described mathematically since there is no symbol in linear algebra to represent the sum over an index. Thus, the following must be expressed in numpy form:

```
grad_b = np.sum((T - Y).dot(V.t) * Z * (1 - Z), axis=0)
```

Code 2: Gradient of b

This is the same form as section 2.3.2 where there is no $x$ term. this is equivalent as $x = 1$.

## 2.4 Mini-Batch Gradient Descent

In a variant of the gradient descent process known as mini-batch gradient descent [20], the training dataset is divided into smaller batches that are then utilized to compute model error and update model coefficients. Mini-batch gradient descent aims to strike a balance between batch gradient descent's efficiency and stochastic gradient descent's [9] robustness. It is the deep learning application of gradient descent that is utilized the most frequently.

The following is how Mini-Batch Gradient Decent can be done in code:

```
for i in range(num_epochs):
    shuffle(X, Y)
    for Xb, Yb in get_batches(X, Y):
        gradient = grad(Xb, Yb, params)
        params = params - learning_rate * gradient
```

Code 3: mini-batch gd code

### 2.4.1 Implementing Batching

The given batching algorithm will divide the inputs and targets into batches that will be fed to the model once during each iteration of training. Every iteration will yield a decrease in cost over time.

```
# Data is of shape (X, Y), batch size is of size B
# N = len(X) = len(Y)
# Suppose N % B != 0
# Number of batches when no remainder = N / B, with
   remainder = ceiling(N / B)
```

```
num_batches = int(np.ceil(N / B))
for j in range(num_batches):
  Xb = X[j * B : (j + 1) * B]
  Yb = Y[j * B : (j + 1) * B]
  # calculate gradient descent using Xb and Yb
```

Code 4: Batching in code

## 2.5 Adam (Adaptive Moment Estimation)

Adam [10] can be viewed as an RMSprop [13] with momentum [2] combination. It scales the learning rate using squared gradients, similar to RMSprop, and leverages momentum by using the gradient's moving average rather than the gradient itself, similar to SGD with momentum. As an adaptive learning rate method, Adam calculates individual learning rates for various parameters. Adam employs estimates of the first and second moments of the gradient to change the learning rate for each weight of the neural network, which is how it gets its name, adaptive moment estimation.

Two helpful ways to improve "vanilla" gradient descent are:

- Momentum
- RMSprop

Adam and these strategies shall be discussed in this section.

### 2.5.1 Momentum and RMSprop

To assist in the set up for the notation required to develop the Adam optimization method, the notation used to express momentum and RMSprop will change slightly from the standard means.

Momentum parameter updating happens in two steps. $m$ is determined by multiplying the current gradient by a fraction of the previous momentum subtracted by learning rate. The following step, known as $\theta$, involves updating the parameters themselves. The old $\theta$ value plus the new momentum constitutes the update [2].

$$m_t = \mu m_{t-1} - \eta g_t$$
$$\theta_t = \theta_{t-1} + m_t$$

This paper will present momentum in a distinctive manner. It will be viewed as somewhat the negative of what it was previously. The weighted sum of $m_{t-1}$ and

the current gradient $g_t$ represents the update for $m_t$.

$$\begin{aligned} m_t &= \beta m_{t-1} + (1-\beta) g_t \\ \theta_t &= \theta_{t-1} - \eta m_t \end{aligned} \tag{48}$$

RMSprop also moves forward in two stages. The cache is refreshed first. Take note of how this is similar to the modified version of momentum. It is the gradient's squared weighted sum. The second stage involves updating the parameter theta as normal, with the exception that the learning rate is split by the square root of the cache rather than remaining constant over time [13].

"Vanilla" RMSprop:

$$\begin{aligned} \textbf{cache}(t) &= \beta\ \textbf{cache}\ (t-1) + (1-\beta) g_t^2 \\ \theta_t &= \theta_{t-1} - \eta \frac{g_t}{\sqrt{\textbf{cache}(t)} + \epsilon} \end{aligned}$$

Modified RMSprop:

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta) g_t^2 \\ \theta_t &= \theta_{t-1} - \eta \frac{g_t}{\sqrt{v_t} + \epsilon} \end{aligned} \tag{49}$$

### 2.5.2 Moving Averages

Moving averages [7] must be explored in order to comprehend why the expressions for momentum and RMSprop must be changed. First, The regular average can be written as:

Suppose there is a set of data points: $\{X_1, X_2, X_3, \cdots, X_t\}$

$$\bar{X}_T = \frac{1}{T} \sum_{t=1}^{T} X_t \tag{50}$$

What happens if there is so much data (say, 1 TB) that the computer is unable to store it all in memory? A fresh measurement $X_t$ is gathered at every time step $T$. The $X$'s up to this point must be added, and the sum must then be divided by $T$ to determine the average of $X$.

**Problem:** As more $X$s are gathered, the computation grows longer. Because the steps required to complete the computation are proportional to $T$, this calculation is known as the time complexity $O(T)$.

**Claim:** It is possible to prove that this time complexity O(T) is constant by applying earlier calculations. This approach can be used to calculate both the average and the preceding average.

Manipulating the equation for the average/sample mean:

$$\begin{aligned}
\bar{X}_T &= \frac{1}{T}\sum_{t=1}^{T} X_t \\
&= \frac{1}{T}\left(\sum_{t=1}^{T-1} X_t + X_t\right) \\
&= \frac{1}{T}\sum_{t=1}^{T-1} X_t + \frac{1}{T}X_t \\
&= \frac{T-1}{T}\bar{X}_{T-1} + \frac{1}{T}X_t \quad \textbf{since } \bar{X}_{T-1} = \frac{1}{T-1}\sum_{t=1}^{T-1} X_t \\
&= \left(1 - \frac{1}{T}\right)\bar{X}_{T-1} + \frac{1}{T}X_T
\end{aligned}$$

Now replace $\frac{1}{T}$ with constant $\alpha$. This will be helpful later on. The sample mean is given by $\frac{1}{T}$. As $T$ rises, $\frac{1}{T}$ drops. Each $X$ taken into consideration thus far is weighed by $\frac{1}{T}$. It will not result in the sample mean if $\frac{1}{T}$ is changed to a constant. In fact, the exponentially weighted moving average [18] will be produced if that occurs.

$$\bar{X}_T = (1-\alpha)\,\bar{X}_{T-1} + \alpha X_T$$

By convention, $\beta$ might be used in place of $\alpha$. The "decay" [23] is the name for $\beta$. Take note of the analogies between the momentum update and the cache update for RMSprop (subsubsection 2.5.1) in this equation.

$$\bar{X}_T = \beta X_{T-1} + (1-\beta)\,\bar{X}_T \quad \textbf{where } \beta = 1-\alpha \tag{51}$$

### 2.5.3 Moments

In RMSProp, the average of the squared gradient is calculated using an exponentially weighted moving average. To calculate expected values, the sample mean is employed. A random variable's expected square value and variance are connected. The second moment is when this happens.

$$\textbf{mean } (X) = E(X) = \textbf{"1st moment of } X\textbf{"} \tag{52}$$

$$\textbf{var } (X) = E(X^2) - \mu^2 = \textbf{"2nd moment of } X\textbf{" - } \mu^2 \tag{53}$$

The first moment [16] of the gradient is estimated by momentum, while the second moment [16] is estimated by RMSProp. Each of these values is given a potential $\beta$ value in order to be tuned separately. $\beta_2$ for RMSProp and $\beta_1$ for momentum.

**Momentum**

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ \theta_t &= \theta_{t-1} - \eta m_t \end{aligned} \tag{54}$$

**RMSProp**

$$\begin{aligned} v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \theta_t &= \theta_{t-1} - \eta \frac{g_t}{\sqrt{v_t} + \epsilon} \end{aligned} \tag{55}$$

**Combining these into Adam** The parameters are subtracted by the learning rate in order to understand how to combine these two concepts. Then, $m$ is used to multiply them. The square root of $v$ is then used to divide them. Then, to prevent division by $0$, it is increased by the minuscule amount $\epsilon$ [10].

$$\theta_t = \theta_{t-1} - \eta \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{56}$$

### 2.5.4 The usage of the exponentially weighted moving average

Technical analysis frequently makes use of the EWMA [18]. Although it might not be applied directly, it is combined with other indicators to produce trading indications. The Negative Volume Index, which is combined with its EWMA, is a well-known example. When the Negative Volume Index goes over its 250 day EWMA, a buy signal is generated.

The EWMA can also be utilized in a straightforward crossover technique, in which the price crosses the EWMA from above and below, respectively, to provide buy and sell signals. The EWMA can be utilized as support or resistance levels, which is another way in which it is put to use in technical analysis.

In place of the simple average, the EWMA is employed. When the data is non-stationary, or when it fluctuates over time, it is valuable. More recent values are therefore more helpful than earlier values. Here is another way to consider the EWMA:

**Example.** *Consider a time series signal like the price of a stock. This is a nice example of when to use this type of average. The input is designated as $x(t)$, while the output is designated as $y(t)$.*

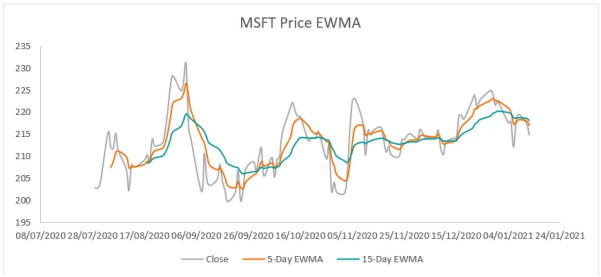


Figure 3: An example of EWMA, from corporatefinanceinstitute.com. A figure that shows how the exponentially weighted moving average can be used to match the trend of the price of a stock [3].

*A smoothed-out version of the time series input is what is produced.*

A low pass filter is what this is known as in signal processing. It is important to pay attention to the low frequency movements because they are usually of greater size. The high frequency movements are typically quite minute and do not have much of an impact. An algorithm that needs to eliminate the high frequencies is fed the input signals.

The last value is always a factor in $y(t)$. The first value, $y(0)$, is not defined though. The output has a bias towards $0$ at the beginning, so when it must start from $0$, it will take longer to try to match the trend. Setting $y(1) = x(1)$ is one option, but there is a superior one that will be explored.

### 2.5.5 Bias Corrrection

To obtain $\hat{y}(t)$, modify this number by a certain amount rather than using $y(t)$. Since $\beta$ is smaller than $1$, $\beta^t$ will eventually approach zero as $t$ increases [22]. Consequently, $y(t)$ will be closer to $\hat{y}(t)$ when $t$ is big since the value is divided by a number close to $1$. This is due to the fact that bias correction is only necessary when $t$ is small.

$$y(t) = \beta y(t-1) + (1-\beta)x(t) \tag{57}$$

$$\hat{y}(t) = \frac{y(t)}{1-\beta^t} \quad \textbf{As } t \to \infty, \hat{y}(t) \to y(t) \tag{58}$$

**Example.**

$$\begin{aligned}
\textit{Suppose } \beta = 0.999\textbf{,} \\
y(0) = 0 \\
y(1) = \beta y(0) + (1-\beta)x(1) = 0.001x(1) \\
\hat{y}(1) = \frac{y(1)}{1-\beta^1} = \frac{0.001x(1)}{0.001} = x(1) \\
y(2) = 0.999y(1) + 0.001x(2) = 0.000999x(1) + 0.001x(2) \\
\hat{y}(2) = \frac{y(2)}{1-\beta^2} = 0.49974x(1) + 0.50025x(2)
\end{aligned}$$

**Incorporating bias correction into Adam** Substitute the values of $m$ and $v$ with their bias corrected terms $\hat{m}$ and $\hat{v}$:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{59}$$

Adam works best with these default hyperparameters:

```
learning_rate = 0.001
beta_1 = 0.9
epsilon = 1e-7 # or 1e-8
```

Code 5: Default Hyperparameters for Adam

**Is Adam ever the right decision?** You will never know the true answer unless you test it yourself because ”machine learning is experimentation, not philosophy.” Trying to predict what the program will do is foolish and ineffective. As the Lazy Programmer [1] says; ”Run the program if you wish to see the output of the program.”

# 3 Constructing the Algorithm

This section will go over the Pseudocode and the Class that are utilized to run the program in order to construct an MLP in Python.

[1]The Lazy Programmer is a machine learning engineer and data scientist. On numerous data science and machine learning-related initiatives, he collaborates with a small number of clients. He creates courses in data science, machine learning, deep learning, and artificial intelligence (AI).

## 3.1 The Dataset

This project uses an artificial classification problem that was created using the sklearn library [2] as the dataset.

```
# make a classification problem
from sklearn.datasets import make_classification

df = make_classification(
    n_samples=500,
    n_features=5,
    n_informative=5,
    n_redundant=0,
    n_classes=3,
    random_state=7
    )
```

Code 6: Classification Dataset

## 3.2 Train and Test Splits

The formation of the data into seperate train and test splits will be done to help the model generalize to new data it sees after training.

```
# split data into train and test sets
from sklearn.model_selection import train_test_split

Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y,
   test_size=0.2, random_state=7)
```

Code 7: Train and Test Splits

## 3.3 Indicator Matrix

The targets that will be fed into the algorithm in order to determine the predictions are represented in the indicator matrix as a one-hot encoded form.

```
# binarize the targets to create the output matrices to go
   in the algorithm
```

[2] Scikit-learn is a free machine learning package for the Python programming language (formerly known as scikits.learn and also referred to as sklearn). Support-vector machines, random forests, gradient boosting, k-means, and DBSCAN are just a few of the classification, regression, and clustering algorithms it offers. It is also built to work with Python's NumPy and SciPy scientific and numerical libraries.

```
from sklearn.preprocessing import LabelBinarizer

label_binarizer = LabelBinarizer()
Ytrain_ind = label_binarizer.fit_transform(Ytrain)
Ytest_ind = label_binarizer.fit_transform(Ytest)
```

Code 8: Indicator Matrix

## 3.4 Cost Function

```
# define cost function: categorical cross entropy
def cost(p_y, t):
    tot = t * np.log(p_y)
    return -tot.sum()
```

Code 9: Cost Function

## 3.5 Forward Function

The forward function will calculate the values of the hidden layer using the ReLU activation function and the outputs using the Softmax function.

```
# define forward function
def forward(X, W1, b1, W2, b2):
    # relu
    Z = X.dot(W1) + b1 # values in the hidden layer
    Z[Z < 0] = 0 # apply relu activation

    A = Z.dot(W2) + b2 # calculate the values from the
   hidden layer to the output layer
    expA = np.exp(A) # exponentiate those values

    Y = expA / expA.sum(axis=1, keepdims=True) # calculate
   the output using the softmax function

    return Y, Z
```

Code 10: Forward Function

## 3.6 Gradients

The gradients are calculated in code as follows:

```
# define derivatives for the neural network parameters
def derivative_w2(Z, T, Y):
    return Z.T.dot(Y - T)

def derivative_b2(T, Y):
    return (Y - T).sum(axis=0)

def derivative_w1(X, Z, T, Y, W2):
    # for relu activation
    return X.T.dot((Y - T).dot(W2.T) * (Z > 0))

def derivative_b1(Z, T, Y, W2):
    # for relu activation
    return ((Y - T).dot(W2.T) * (Z > 0)).sum(axis=0)
```

Code 11: Gradients

## 3.7 Prediction and Error Rate

The indicator matrix's output probabilities are computed using the prediction function. The error function will determine the degree of error and the discrepancy between predictions and targets.

```
# define the function to calculate the prediction
def predict(p_y):
    return np.argmax(p_y, axis=1)

# define the function to calculate the error rate
def error_rate(p_y, t):
    prediction = predict(p_y)
    return np.mean(prediction != t)
```

Code 12: Prediction and Error Rate

## 3.8 The Algorithm

The objective of this paper is to transfer all of the theory from the earlier sections into Python code. Make the dataset first. Then, using the formulas found in equations Equation 30, section 2.3.2, section 2.3.5, and Equation 47, loop over the maximum number of iterations and the number of batches to divide the data into batches for mini-batch gradient descent. The changed values for the first and second moments of the gradient are then updated while applying bias correction.

The model parameters—which are just the weights and biases—should then be updated. Lastly, print the findings for each batch to determine if the training has improved. The pseudo-code for the algorithm can be seen below as follows:

**Algorithm 1** Multi-Layer Perceptron

**Require:** Classification Dataset

**for** i in range(max-iter) **do**:

**for** j in range(num-batches) **do**:

Sort the data into batches,

```
Xbatch = Xtrain[(j * batch-sz) : (j * batch-sz + batch-sz), ]
Ybatch = Ytrain-ind[(j * batch-sz) : (j * batch-sz + batch-sz), ]
```

Calculate the gradients,

$$\nabla_V J = Z^T(T - Y) \quad 30$$

```
grad(c) = np.sum(T - Y, axis=0)   2.3.2
```

$$\nabla_W J = X^T \left\{ \left[ (T - Y)V^T \right] \odot Z > 0 \right\} \quad 47$$

```
grad(b) = np.sum((T - Y).dot(V.t) * Z * (1 - Z), axis=0)   2.3.5
```

Calculate the first moment of the gradient,

$$\textbf{mean}\ (X) = E(X) = \textbf{"1st moment of } X\textbf{"} \quad 52$$

Calculate the second moment of the gradient,

$$\textbf{var}\ (X) = E(X^2) - \mu^2 = \textbf{"2nd moment of } X\textbf{" - } \mu^2 \quad 53$$

Handle bias correction for the first and second moments,

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad 59$$

```
t += 1
```

Update the parameters.

**if** j % print period == 0 **then**:

Print the cost and error rate for every batch per iteration.

**end if**

**end for**

**end for**

Print the final error rate.

## 3.9 Implementing the Algorithm in Code

This is how the algorithm code is put into practice. To make the code more reuseable, a python class containing all the functions described in the earlier subsections is built. The classes offered on Sklearn served as the inspiration for this concept.

```
class MLP:
    def __init__(self, M, X, Y):
        self.X = X
        self.Y = Y
        self.M = M
        self.N, self.D = self.X.shape # the number of
    samples and the dimensionality of the training set
        self.K = len(set(Y)) # number of classes in Y
        self.batch_sz = 50 # the batch size
        self.n_batches = self.N // self.batch_sz # the
    number of batches

        # initialize the weights randomly
        self.W1 = np.random.randn(self.D, self.M) / np.sqrt
    (self.D)
        self.b1 = np.zeros(self.M)
        self.W2 = np.random.randn(self.M, self.K) / np.sqrt
    (self.M)
        self.b2 = np.zeros(self.K)

        self.max_iter = 500 # maximum number of iterations/
    epochs
        self.print_period = 7 # iteration period to print
    out the results

        # values for the first moment
        self.mW1 = 0
        self.mb1 = 0
        self.mW2 = 0
        self.mb2 = 0

        # values for the second moment
        self.vW1 = 0
        self.vb1 = 0
        self.vW2 = 0
        self.vb2 = 0

        self.learning_rate = 0.0001
```

```
        self.beta_1 = 0.99 # decay rate for the first
moment (momentum)
        self.beta_2 = 0.999 # decay rate for second moment
(sum of the squared gradient in RMSProp)
        self.eps = 1e-8 # small parameter added to the
cache to avoid dividing by zero
        self.reg = 0 # reularization value


    def split_data(self, X, Y):

        Xtrain, Xtest, Ytrain, Ytest = train_test_split(
self.X, self.Y, test_size=0.2, random_state=7)

        return Xtrain, Xtest, Ytrain, Ytest

    def binarize(self, Ytrain, Ytest):
        label_binarizer = LabelBinarizer()
        Ytrain_ind = label_binarizer.fit_transform(Ytrain)
        Ytest_ind = label_binarizer.fit_transform(Ytest)

        return Ytrain_ind, Ytest_ind

    def cost(self, p_y, t):
        tot = t * np.log(p_y)

        return -tot.sum()

    def forward(self, Xtrain, W1, b1, W2, b2):
        # relu
        Z = Xtrain.dot(W1) + b1 # values in the hidden
layer
        Z[Z < 0] = 0 # apply relu activation

        A = Z.dot(W2) + b2 # calculate the values from the
hidden layer to the output layer
        expA = np.exp(A) # exponentiate those values

        Y = expA / expA.sum(axis=1, keepdims=True) #
calculate the output using the softmax function

        return Y, Z
```

```
 def derivative_w2(self, Z, T, Y):
     return Z.T.dot(Y - T)

 def derivative_b2(self, T, Y):
     return (Y - T).sum(axis=0)

 def derivative_w1(self, X, Z, T, Y, W2):
     # for relu activation
     return X.T.dot((Y - T).dot(W2.T) * (Z > 0))

 def derivative_b1(self, Z, T, Y, W2):
     # for relu activation
     return ((Y - T).dot(W2.T) * (Z > 0)).sum(axis=0)

 def predict(self, p_y):
     return np.argmax(p_y, axis=1)

 def error_rate(self, p_y, t):
     prediction = self.predict(p_y)
     return np.mean(prediction != t)

 def fit(self, Xtrain, Ytrain_ind, Xtest, Ytest_ind,
Ytest, plot=False):
     self.Xtrain = Xtrain
     self.Ytrain_ind = Ytrain_ind
     self.Xtest = Xtest
     self.Ytest_ind = Ytest_ind
     self.Ytest = Ytest

     loss = [] # list to store loss
     errors = [] # list to store error values
     t = 1 # time index: t

     for i in range(self.max_iter):
         for j in range(self.n_batches):

             # sort the data into batches
             Xbatch = self.Xtrain[(j * self.batch_sz) :
(j * self.batch_sz + self.batch_sz), ]
             Ybatch = self.Ytrain_ind[(j * self.batch_sz
) : (j * self.batch_sz + self.batch_sz), ]
             pYbatch, Z = self.forward(Xbatch, self.W1,
self.b1, self.W2, self.b2)
```

```
            # calculate the gradients
            gW2 = self.derivative_w2(Z, Ybatch, pYbatch
) + self.reg * self.W2
            gb2 = self.derivative_b2(Ybatch, pYbatch) +
 self.reg * self.b2
            gW1 = self.derivative_w1(Xbatch, Z, Ybatch,
 pYbatch, self.W2) + self.reg * self.W1
            gb1 = self.derivative_b1(Z, Ybatch, pYbatch
, self.W2) + self.reg * self.b1

            # new values for the 1st moment
            self.mW1 = self.beta_1 * self.mW1 + (1 -
self.beta_1) * gW1
            self.mb1 = self.beta_1 * self.mb1 + (1 -
self.beta_1) * gb1
            self.mW2 = self.beta_1 * self.mW2 + (1 -
self.beta_1) * gW2
            self.mb2 = self.beta_1 * self.mb2 + (1 -
self.beta_1) * gb2

            # new values for the 2nd moment
            self.vW1 = self.beta_2 * self.vW1 + (1 -
self.beta_2) * gW1 * gW1
            self.vb1 = self.beta_2 * self.vb1 + (1 -
self.beta_2) * gb1 * gb1
            self.vW2 = self.beta_2 * self.vW2 + (1 -
self.beta_2) * gW2 * gW2
            self.vb2 = self.beta_2 * self.vb2 + (1 -
self.beta_2) * gb2 * gb2

            # bias correction
            correction1 = 1 - self.beta_1 ** t
            hat_mW1 = self.mW1 / correction1
            hat_mb1 = self.mb1 / correction1
            hat_mW2 = self.mW2 / correction1
            hat_mb2 = self.mb2 / correction1

            correction2 = 1 - self.beta_2 ** t
            hat_vW1 = self.vW1 / correction2
            hat_vb1 = self.vb1 / correction2
            hat_vW2 = self.vW2 / correction2
            hat_vb2 = self.vb2 / correction2
```

```
                # update t
                t += 1

                # update the parameters
                self.W1 = self.W1 - self.learning_rate *
    hat_mW1 / np.sqrt((hat_vW1) + self.eps)
                self.b1 = self.b1 - self.learning_rate *
    hat_mb1 / np.sqrt((hat_vb1) + self.eps)
                self.W2 = self.W2 - self.learning_rate *
    hat_mW2 / np.sqrt((hat_vW2) + self.eps)
                self.b2 = self.b2 - self.learning_rate *
    hat_mb2 / np.sqrt((hat_vb2) + self.eps)

                # print results at every print period
                if j % self.print_period == 0:
                    pY, _ = self.forward(self.Xtest, self.
    W1, self.b1, self.W2, self.b2)
                    l = self.cost(pY, self.Ytest_ind)
                    loss.append(l)
                    print(f"Cost at iteration i = {i:d}, j
    = {j:d}: {l:.6f}")

                    err = self.error_rate(pY, self.Ytest)
                    errors.append(err)
                    print(f'Error rate: {err}')

        if plot == True:
            plt.plot(loss, label='Loss curve')
            plt.show()
```

Code 13: The Multi-Layer Perceptron Class

# 4 Results

The loss curve shows how well the model can cut down on overall loss. The list of modifications made to the hyperparameters during training, along with the loss curve is shown in this section:

```
# Final Training Run
  max_iter = 500
  print_period = 7
  batch_sz = 50
```

```
  M = 50
  learning_rate = 0.0001
  beta_1 = 0.99
  beta_2 = 0.999
  eps = 1e-8
  reg = 0
Cost at iteration i = 499, j = 7: 49.135031
Error rate: 0.18
```

Code 14: Hyperparameter Modifications

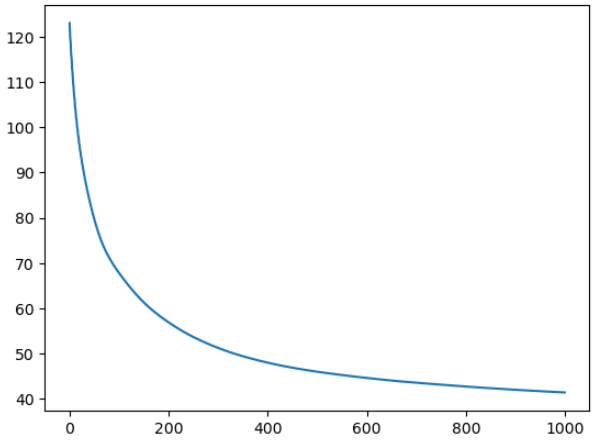


Figure 4: Loss curve for the MLP

The loss converges to 0.18.

# 5 Conclusion

This paper demonstrates the theory-based creation of a multi-layer perceptron in code. MLPs are employed for multi-class classification issues and have a proven track record of producing effective outcomes. The effectiveness of training is enhanced by the use of ideas like adaptive learning rates, momentum, mini-batch gradient descent, and bias correction.

# Acknowledgement

I wish to express my gratitude to the Lazy Programmer on Udemy for, in the words of his lecture on Modern Deep Learning in Python, Adam Optimization (Part 2), ”being the source of many surprising and unusual facts”. I also want to thank him for giving me the information and abilities I needed to create such an algorithm. For all of my projects, I have attempted to adhere to his approach of first discussing the theory and then putting it into practice via code.


# References


[1] Atılım Güneş Baydin, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without backpropagation, 2022.

[2] Aleksandar Botev, Guy Lever, and David Barber. Nesterov’s accelerated gradient and momentum as approximations to regularised update descent, 2016.

[3] CFI. Exponentially weighted moving average (ewma), 2022.

[4] Charles Davi. Vectorized deep learning, 11 2020.

[5] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview, 2020.

[6] Manfred te Grotenhuis and Paula Thijs. Dummy variables and their interactions in regression analysis: examples from research on body mass index, 2015.

[7] Seng Hansun. A novel research of new moving average method in time series analysis. *International Journal of New Media Technology (IJNMT)*, 1:22, 08 2014.

[8] Geoffrey E Hinton and Russ R Salakhutdinov. Replicated softmax: an undirected topic model. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[9] Jie Hu, Vishwaraj Doshi, and Do Young Eun. Efficiency ordering of stochastic gradient descent, 2022.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[11] Haochuan Li, Farzan Farnia, Subhro Das, and Ali Jadbabaie. On convergence of gradient descent ascent: A tight local analysis, 2022.

[12] Keiji Miura. An introduction to maximum likelihood estimation and information geometry. *Interdisciplinary Information Sciences (IIS)*, 17, 11 2011.

[13] Mahesh Chandra Mukkamala and Matthias Hein. Variants of rmsprop and adagrad with logarithmic regret bounds, 2017.

[14] Uwe Naumann. On the computational complexity of the chain rule of differential calculus, 2021.

[15] R. H. Norden. A survey of maximum likelihood estimation. *International Statistical Review / Revue Internationale de Statistique*, 40(3):329–354, 1972.

[16] Daniel M. Packwood. Moments of sums of independent and identically distributed random variables, 2011.

[17] Joanne Peng, Kuk Lee, and Gary Ingersoll. An introduction to logistic regression analysis and reporting. *Journal of Educational Research - J EDUC RES*, 96:3–14, 09 2002.

[18] Marcus Perry. The exponentially weighted moving average, 06 2010.

[19] Marius-Constantin Popescu, Valentina Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8, 07 2009.

[20] Xin Qian and Diego Klabjan. The impact of the mini-batch size on the variance of gradients in stochastic gradient descent, 2020.

[21] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[22] Enzo Tartaglione, Carlo Alberto Barbano, and Marco Grangetto. End: Entangling and disentangling deep representations for bias correction, 2021.

[23] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. How does learning rate decay help modern neural networks?, 2019.

[24] Zhilu Zhang and Mert R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels, 2018.

[25] Donglai Zhu, Hengshuai Yao, Bei Jiang, and Peng Yu. Negative log likelihood ratio loss for deep neural network classification, 2018.

# A Appendix

Please take note that I have just labeled the crucial equations and outcomes. The steps/expressions/equations required to calculate those results are not labeled because they will be discussed in the proofs portion of this work, if an explanation for those results are required.

# B Proofs

## B.1 Proof for Equation 11

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = \frac{\partial}{\partial a_{nk}} exp(a_{nk'}) \left\{ \sum_j exp(a_{nj}) \right\}^{-1}$$

**Use the product rule,**

**Note: The derivative of 'exp' is just itself:** $\frac{\partial}{\partial a_{nk}} exp(a_{nj}) = exp(a_{nk})$

$$= \frac{\partial}{\partial a_{nk}} exp(a_{nk'}) \cdot \left\{ \sum_j exp(a_{nj}) \right\}^{-1} + exp(a_{nk'}) \cdot \frac{\partial}{\partial a_{nk}} \left\{ \sum_j exp(a_{nj}) \right\}^{-1}$$

$$\frac{\partial y_{nk'}}{\partial a_{nk}} = exp(a_{nk'})(-1) \left\{ \sum_j exp(a_{nj}) \right\}^{-2} exp(a_{nk})$$

## B.2 Proof for Equation 13

$$\frac{\partial y_{nk}}{\partial a_{nk}} = \frac{\partial}{\partial a_{nk}} exp(a_{nk}) \left\{ \sum_j exp(a_{nj}) \right\}^{-1}$$

**Use the product rule**

$$= \frac{\partial}{\partial a_{nk}} exp(a_{nk}) \cdot \left\{ \sum_j exp(a_{nj}) \right\}^{-1} + exp(a_{nk}) \cdot \frac{\partial}{\partial a_{nk}} \left\{ \sum_j exp(a_{nj}) \right\}^{-1}$$

$$= exp(a_{nk}) \left\{ \sum_j exp(a_{nj}) \right\}^{-1} + exp(a_{nk})(-1) \left\{ \sum_j exp(a_{nj}) \right\}^{-2} exp(a_{nk})$$

$$\frac{\partial y_{nk}}{\partial a_{nk}} = exp(a_{nk}) \left\{ \sum_j exp(a_{nj}) \right\}^{-1} - exp(a_{nk})^2 \left\{ \sum_j exp(a_{nj}) \right\}^{-2}$$

## B.3 Proof for Equation 17

$$J_{nk'} = t_{nk'} \log y_{nk'}$$

**Differentiation is linear**

**Differentiate summands seperately and pull out constant factors,**

$$\frac{\partial J_{nk'}}{\partial y_n k'} = t_{nk'} * \frac{1}{y_{nk'}}$$

$$\frac{\partial J_{nk'}}{\partial y_n k'} = \frac{t_{nk'}}{y_{nk'}}$$

## B.4 Proof for Equation 18

$$a_{nk} = W_{:,k}^T x_n$$

**From the Matrix Cookbook's eq:69,**

$$\frac{\partial x^T a}{\partial x} = \frac{\partial a^T x}{\partial x} = a$$

$$\frac{\partial a_{nk}}{\partial W_i k} = x_{ni}$$

## B.5 Proof for Equation 41

$$
\begin{aligned}
\frac{\partial z_{nm}}{\partial \alpha_{nm}} \quad & \textbf{if } \sigma(x) = \frac{1}{1+exp(-x)} \\
&= \frac{\partial}{\partial \alpha_{nm}} \left[ \frac{1}{1+exp(-z_{nm})} \right] \\
&= -\frac{\frac{\partial}{\partial \alpha_{nm}} \left[ 1+exp(-z_{nm}) \right]}{1+exp(-z_{nm})^2} \\
&= -\frac{\frac{\partial}{\partial \alpha_{nm}} [exp(-z_{nm})] + \frac{\partial}{\partial \alpha_{nm}} [1]}{1+exp(-z_{nm})^2} \\
&= -\frac{exp(-z_{nm}) \cdot \frac{\partial}{\partial \alpha_{nm}} [-z_{nm}] + 0}{1+exp(-z_{nm})^2} \\
&= -\frac{exp(-z_{nm})(-1)}{1+exp(-z_{nm})^2} \\
&= -\frac{-exp(-z_{nm})}{1+exp(-z_{nm})^2} \\
& \textbf{Remove the negative sign,} \\
&= \frac{exp(-z_{nm})}{1+exp(-z_{nm})^2} \\
&= \frac{1}{1+exp(-z_{nm})} \cdot \frac{exp(-z_{nm})}{1+exp(-z_{nm})} \\
\frac{\partial z_{nm}}{\partial \alpha_{nm}} &= z_{nm}(1-z_{nm})
\end{aligned}
$$

## B.6 Proof for Equation 42

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} \quad \textbf{if } \sigma(x) = \textbf{tanh}(x)$$

$$z_{nm} = \textbf{tanh}(\alpha_{nm})$$

$$= \textbf{tanh}(W_{:,m}^{T} x_n + b_m)$$

$$\textbf{tanh}(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)}$$

$$\textbf{let } x = W_{:,m}^{T} x_n + b_m$$

$$= \frac{\partial}{\partial \alpha_{nm}} \left[ \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \right]$$

**Since the equation is large; $e^x$ is now used,**

**Apply the quotient rule,**

$$= \frac{\frac{\partial}{\partial \alpha_{nm}}[e^x - e^{-x}] \cdot (e^{-x} + e^x) - (e^x - e^{-x}) \cdot \frac{\partial}{\partial \alpha_{nm}}[e^x + e^{-x}]}{(e^x + e^{-x})^2}$$

$$= \frac{\left(\frac{\partial}{\partial x}[e^x] - \frac{\partial}{\partial x}[e^{-x}]\right)(e^x + e^{-x}) - (e^x - e^{-x})\left(\frac{\partial}{\partial x}[e^x] - \frac{\partial}{\partial x}[e^{-x}]\right)}{(e^x + e^{-x})^2}$$

$$= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2}$$

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} = 1 - z_{nm}^2$$

## B.7 Proof for Equation 43

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} \quad \textbf{if } \sigma(x) = \textbf{relu}(x)\textbf{, where } \textbf{u}(\cdot) = \textbf{step function}$$

$$\frac{\partial z_{nm}}{\partial \alpha_{nm}} = \frac{\partial}{\partial \alpha_{nm}} \left[\textbf{max}\ (0, z_{nm})\right]$$

$$\textbf{Therefore } z_{nm}[z_{nm} < 0] = 0$$