Building Neural Networks from Scratch

Joyce Mai

July, 2025

Contents

1 Introduction

This note documents the step-by-step process of understanding and building neural networks from scratch using Python and mathematical intuition.

Topics include:

- Neuron model and activation functions
- Forward propagation
- Backpropagation and gradient descent
- Loss functions
- Building and training a neural network without libraries

What is a Neural Network

Neural networks are computational models inspired by the human brain. They consist of layers of interconnected nodes (also called neurons), where each connection has an associated weight. Neural networks learn patterns from data by adjusting these weights to minimize prediction error.

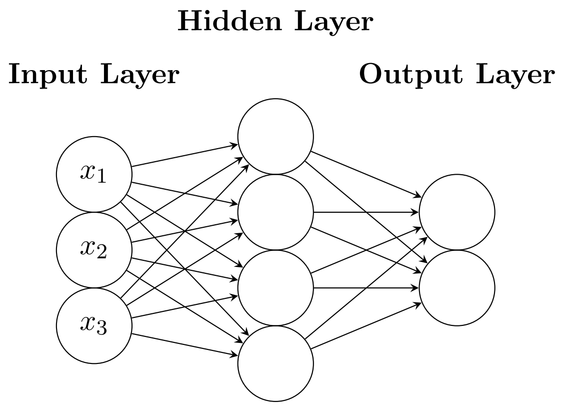
At a high level, the goal of a neural network is to approximate a function:

$$f^* : \mathbb{R}^n \to \mathbb{R}^m$$

given input-output pairs (x, y), the network learns a mapping $f(x; \theta) \approx y$ by optimizing parameters θ (weights and biases) using training data.

Basic Structure of a Feedforward Neural Network

Below is a simplified diagram of a neural network with one hidden layer:



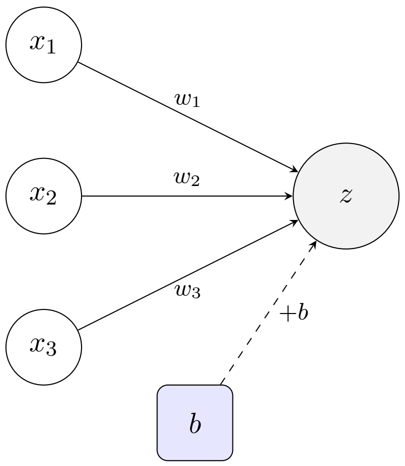
2 Artificial Neuron and Activation Functions

2.1 Perceptron Model

A single artificial neuron computes a weighted sum of its inputs and adds a bias to produce an output. The basic formula is:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Below is a diagram illustrating how this computation works:



Here is the corresponding code in Python:

Listing 1: Single Neuron

```python
def single_neuron(x,w,b):
    z=0
    for i in range(len(x)):
        z += x[i] * w[i]
    z += b
    return z
```

2.2 Model a Layer

A single neuron computes a weighted sum of its inputs plus a bias and optionally applies an activation function. To build a layer, we stack multiple such neurons in parallel. Each neuron in the layer receives the same input vector but has its own set of weights and bias. Mathematically, if we have an input vector $\mathbf{x} \in \mathbb{R}^n$ and a layer with m neurons, we can represent the weights of the layer as a matrix $W \in \mathbb{R}^{m \times n}$, the biases as a vector $\mathbf{b} \in \mathbb{R}^m$, and compute the layer output as:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}$$

Here is the corresponding code in Python:

Listing 2: Single Layer

```python
import numpy as np

def single_layer(w,x,b):
    return np.dot(w,x) + b
```

2.3 Model a Batch

When training neural networks, we usually process multiple input examples at once in what is called a **batch**. Instead of feeding one data point at a time, we use a batch of B samples, each with n features. This allows us to leverage fast matrix operations and improve training efficiency and stability.

For a layer with weight matrix $W \in \mathbb{R}^{m \times n}$, and a batch input $X \in \mathbb{R}^{B \times n}$, the output of the layer is:

$$Z = XW^\top + \mathbf{b}, \quad A = \sigma(Z)$$

Here is the corresponding code in Python:

Listing 3: Model Batch

```python
import numpy as np

inputs = [[1,1,1],[2,2,2]]
weights = [[1,2,3],[2,3,4]]
bias = np.array([0,0])

def single_batch(w,x,b):
    return np.dot(w, np.array(x).T) + b

print(single_batch(weights,inputs,bias))
#output:
#[[ 6 12]
# [ 9 18]]
```

2.4 Object Layers

To better organize our code and prepare for building deeper networks, we can define a Python class to represent a dense (fully connected) layer. This class stores the layer's parameters (weights and biases) and provides a method for forward propagation.

Below is a simple implementation:

Listing 4: Layer class for forward pass

```
import numpy as np

np.random.seed(0)

class DenseLayer:
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.10 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases

# example
X = [[1, 2, 3, 4],
     [2.4, 1.5, -4, 2],
     [6, -5.3, 4.8, 1.1]]

layer1 = DenseLayer(4,5)
layer2 = DenseLayer(5,2)

layer1.forward(X)
print(layer1.output)
'''
[[ 0.15763246  1.26394709  0.21385037  0.36517514  0.06039617]
[ 0.28589832 -0.0443426  -0.13325321  0.53627507  0.16143925]
[ 1.68223377  0.59894745  1.01019281  1.49208337  1.02202141]]
'''

layer2.forward(layer1.output)
print(layer2.output)
'''
[[ 0.12848393 -0.11256537]
 [-0.0798689   0.0550407 ]
 [ 0.01507247  0.04082539]]
'''
```

2.5 Activation Functions

An **activation function** introduces non-linearity into a neural network. Without activation functions, a network composed of only linear layers would be equivalent to a single linear transformation, regardless of depth.

After computing the weighted sum $z = \mathbf{w} \cdot \mathbf{x} + b$, we apply an activation function $a = \sigma(z)$, where $\sigma(\cdot)$ is a non-linear function. This enables neural networks to learn complex patterns and decision boundaries.

Common Activation Functions

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Output range: $(0, 1)$
  - Historically used in binary classification
  - *Limitation:* suffers from vanishing gradients for large/small inputs
- **Tanh:** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Output range: $(-1, 1)$
  - Zero-centered output
  - *Limitation:* suffers from vanishing gradients for large/small inputs
- **ReLU (Rectified Linear Unit):** $\mathrm{ReLU}(x) = \max(0, x)$
  - Very efficient and widely used in modern networks
  - Encourages sparse activation
  - *Limitation:* can suffer from "dying ReLU" where neurons stop activating (i.e., always output 0)
- **Leaky ReLU:** $\mathrm{LeakyReLU}(x) = \max(0.01x, x)$
  - Variant of ReLU that allows a small gradient when $x < 0$
  - Helps prevent dying ReLU problem

Choice of Activation in This Note

For the remainder of this study note, we will primarily use the **ReLU activation function** due to its simplicity, fast convergence, and strong empirical performance in deep networks.

Here is the corresponding code in Python:

Listing 5: ReLU Activation Function

```python
import numpy as np

class Activation_ReLU:
    def forward(self, inputs):
        self.output = np.maximum(0, inputs)


print(layer1.output)
'''
[[ 0.15763246  1.26394709  0.21385037  0.36517514  0.06039617]
[ 0.28589832 -0.0443426  -0.13325321  0.53627507  0.16143925]
[ 1.68223377  0.59894745  1.01019281  1.49208337  1.02202141]]
'''


```

```python
activation1 = Activation_ReLU()
activation1.forward(layer1.output)
print(activation1.output)
'''
[[0.15763246 1.26394709 0.21385037 0.36517514 0.06039617]
 [0.28589832 0.         0.         0.53627507 0.16143925]
 [1.68223377 0.59894745 1.01019281 1.49208337 1.02202141]]
'''
```

2.6 Softmax Activation

The **softmax activation** function is commonly used in the output layer of a neural network for **multi-class classification**. It converts a vector of raw scores (logits) into a probability distribution, where each value lies in $(0, 1)$, and the total sums to 1.

Given a vector of logits $\mathbf{z} = [z_1, z_2, \ldots, z_k]$, the softmax output for class i is computed as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad \text{for } i = 1, \ldots, k$$

This function highlights the largest values and suppresses values that are significantly lower, making it ideal for interpreting the outputs as class probabilities.

Exponential can grow very large, which may cause numerical overflow or precision issues. To avoid this, we subtract the maximum value in the input vector from each element before applying the exponential. This does not affect the result because:

$$\frac{e^{z_i}}{\sum_j e^{z_j}} = \frac{e^{z_i - \max(z)}}{\sum_j e^{z_j - \max(z)}}$$

This common trick ensures the softmax computation remains stable even when z_i values are large.

Below is a simple implementation:

Listing 6: Softmax Activation Class

```python
import numpy as np

class Activation_Softmax:
    def forward(self, inputs):
        exp_values = np.exp(inputs - np.max(inputs, axis = 1, keepdims =
            ↪ True))
        probabilities = exp_values / np.sum(exp_values, axis = 1, keepdims
            ↪ = True)
        self.output = probabilities
```

2.7 Calculating Loss with Categorical Cross-Entropy

After computing the network's output probabilities, we need a way to measure how well the model is performing. This is done using a **loss function**, which quantifies the difference between the predicted output and the true label.

For multi-class classification, the most common choice is the **categorical cross-entropy** loss. It compares the predicted probability distribution $\hat{\mathbf{y}}$ (from softmax) to the true distribution $\mathbf{y}$ (a one-hot encoded vector).

One-Hot Encoding?

In classification, labels are typically integers (e.g., 0, 1, 2 for 3 classes). To compute loss correctly, we convert each label into a one-hot vector—an array where the correct class index is 1 and all others are 0.

Cross-Entropy Formula

Given a true label vector $\mathbf{y}$ and a predicted probability vector $\hat{\mathbf{y}}$, the categorical cross-entropy loss is:

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Since only one element of y_i is 1 (the correct class), this simplifies to:

$$L = -\log(\hat{y}_{\text{correct class}})$$

Below is a simple implementation:

Listing 7: Loss Calculation

```python
class Loss_CategoticalCrossentropy(Loss):
    def forward(self, y_pred, y_true):
        sample = len(y_pred)
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[range(sample), y_true]

        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(y_pred_clipped * y_true, axis =
            ↪ 1)

        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
```

3 Forward Propagation

Forward propagation is the process by which input data passes through the layers of a neural network to produce an output. It is a critical step during both training and inference, as it determines the predictions made by the model before loss calculation.

3.1 Overview of the Process

A typical forward pass in a neural network includes the following steps:

1. **Input Data:** The raw input features are fed into the network (e.g., a vector of length n per sample).
2. **Dense (Fully Connected) Layer:** Each input sample is passed through a dense layer which performs a linear transformation:

$$\text{output} = \text{inputs} \cdot \text{weights} + \text{biases}$$

3. **Activation Function:** The result is passed through an activation function (e.g., ReLU or Softmax) to introduce non-linearity and/or output probabilities.
4. **Loss Calculation (During Training):** If training, the final output is compared with the true labels to compute the loss using a loss function (e.g., categorical cross-entropy). This loss quantifies how far the predictions are from the ground truth.
5. **Prediction:** If evaluating or testing, the predicted class can be taken as the one with the highest probability (in the case of classification).

3.2 Code Implementation

Below is a Python implementation of forward propagation using NumPy. It includes dense layers, activation functions (ReLU and Softmax), and categorical cross-entropy loss:

```python
import numpy as np
from forward import DenseLayer, Activation_ReLU, Activation_Softmax,
    ↪ Loss_CategoticalCrossentropy

np.random.seed(0)

# Dummy input data: 3 samples, 2 features
X = np.array([[1.0, 2.0],
              [0.5, -1.5],
              [-1.0, 2.5]])

# Ground truth labels (class indices for classification)
y = np.array([0, 1, 1])

```

```python
# Create layers
# 2 inputs to 3 neurons
dense1 = DenseLayer(2, 3)
activation1 = Activation_ReLU()
# 3 inputs to 3 outputs (for 3 classes)
dense2 = DenseLayer(3, 3)
activation2 = Activation_Softmax()

# Forward pass through first dense layer + ReLU
dense1.forward(X)
print(dense1.output)
activation1.forward(dense1.output)
print(activation1.output)

# Forward pass through second dense layer + Softmax
dense2.forward(activation1.output)
print(dense2.output)
activation2.forward(dense2.output)

# Loss calculation
loss_function = Loss_CategoticalCrossentropy()
loss = loss_function.calculate(activation2.output, y)

# Output everything
print("Predicted␣probabilities:\n", activation2.output)
print("Loss:", loss)
```

3.3 Conclusion

This completes the full forward pass of a neural network. In training, this is typically followed by backpropagation to update the weights.

4 Backpropagation

Training a neural network involves minimizing a loss function by adjusting the weights of each layer. To achieve this, we need to calculate how much each weight contributes to the overall error. This is where backpropagation comes into play.

Backpropagation is an algorithm that efficiently computes the gradient of the loss function with respect to every weight in the network. It applies the chain rule of calculus layer by layer, propagating errors from the output layer backward through the network.

4.1 Gradient of Loss with Respect to Output Logits

In backpropagation, our ultimate goal is to update the weights of the network so that the loss L is minimized. The weights affect the loss indirectly: they influence the linear combinations z_i (logits) of the last layer, which then pass through the activation function (Softmax) to produce predictions $\hat{y}_i$, and finally determine the loss.

Therefore, we must first compute the gradient of the loss with respect to the logits z_i, written as:

$$\frac{\partial L}{\partial z_i}$$

These gradients act as the starting point for propagating error signals back through the network using the chain rule.

4.1.1 Relation Between z_i, $\hat{y}_i$, and L

The flow of computations at the output neuron can be summarized as:

$$\begin{tikzcd} z_i \arrow[r, "\text{Softmax}"] & \hat{y}_i \arrow[r, "\text{Cross-Entropy}"] & L \end{tikzcd}$$

4.1.2 Chain Rule Expression

Since L depends on z_i indirectly through $\hat{y}_i$, we use the chain rule of calculus to decompose the derivative:

$$\frac{\partial L}{\partial z_i} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_i}$$

Here:

- $\frac{\partial L}{\partial \hat{y}_k}$ is the gradient of the loss with respect to the predicted probabilities.
- $\frac{\partial \hat{y}_k}{\partial z_i}$ is the gradient of the softmax output with respect to the logits.

This expression shows how backpropagation leverages the chain rule to efficiently propagate gradients backward through the network.

4.2 Deriving the Gradient $\frac{\partial L}{\partial z_i}$

We now derive the expression for the gradient of the loss L with respect to the logits z_i at the output layer. This is the first gradient computed in backpropagation and serves as the starting point for propagating errors backward.

Step 1. Definitions

The softmax output for class i is:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

The categorical cross-entropy loss is given by:

$$L = -\sum_k y_k \log(\hat{y}_k)$$

where y_k is the one-hot encoded true label.

Step 2. Apply the Chain Rule

Since L depends on z_i through $\hat{y}_k$, we use:

$$\frac{\partial L}{\partial z_i} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_i}$$

Step 3. Derivative of L w.r.t. $\hat{y}_k$

From the loss expression:

$$\frac{\partial L}{\partial \hat{y}_k} = -\frac{y_k}{\hat{y}_k}$$

Step 4. Derivative of $\hat{y}_k$ w.r.t. z_i

The derivative of the softmax function is:

$$\frac{\partial \hat{y}_k}{\partial z_i} = \begin{cases} \hat{y}_i(1-\hat{y}_i), & \text{if } k = i \\ -\hat{y}_k\hat{y}_i, & \text{if } k \neq i \end{cases}$$

Step 5. Combine Terms

Substitute the expressions for both derivatives into the chain rule:

$$\frac{\partial L}{\partial z_i} = \sum_k \left(-\frac{y_k}{\hat{y}_k}\right) \cdot \frac{\partial \hat{y}_k}{\partial z_i}$$

Split the summation into two cases:

$$\frac{\partial L}{\partial z_i} = \left(-\frac{y_i}{\hat{y}_i}\right) \hat{y}_i(1-\hat{y}_i) + \sum_{k\neq i} \left(-\frac{y_k}{\hat{y}_k}\right) (-\hat{y}_k\hat{y}_i)$$

Simplify the first term:

$$= -y_i(1-\hat{y}_i) + \hat{y}_i \sum_{k\neq i} y_k$$

Since $\sum_k y_k = 1$ (because y_k is one-hot), we have $\sum_{k\neq i} y_k = 1 - y_i$. Therefore:

$$\frac{\partial L}{\partial z_i} = -y_i(1-\hat{y}_i) + \hat{y}_i(1-y_i)$$

Combine terms:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

Final Result

The derivative simplifies to:

$$\boxed{\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i}$$

This is a remarkably simple expression: the gradient at the output layer is just the difference between the predicted probability $\hat{y}_i$ and the true label y_i. This simplicity is one reason why softmax activation is almost always paired with categorical cross-entropy loss.

Code Implementation

Here is a simple code implementation for the class *Activation_ Softmax_ Loss_ CategoricalCrossentropy*

```python
class Activation_Softmax_Loss_CategoricalCrossentropy:

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

        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        self.dinputs = y_pred.copy()
        self.dinputs[range(samples), y_true] -= 1
        self.dinputs = self.dinputs / samples
```

4.3 Gradients with Respect to Weights and Biases

Once we have the gradient of the loss with respect to the logits z_i, we can compute the gradients for the parameters W and b in the dense (fully connected) layer.

Step 1. Dense Layer Equation

For a single dense layer, the logits are computed as:

$$z = Wa + b$$

where:

- W is the weight matrix of shape $(n_{\text{inputs}}, n_{\text{neurons}})$,
- a is the input vector (activations from the previous layer),
- b is the bias vector of shape $(1, n_{\text{neurons}})$,

- z is the output (logits) before activation.

For convenience, we denote the gradient from the next layer (or from the loss) as:

$$\delta = \frac{\partial L}{\partial z}$$

For the output layer with softmax and categorical cross-entropy, we have already derived:

$$\delta = \hat{y} - y$$

Step 2. Gradient w.r.t. Weights W

Using the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial W}$$

From $z = Wa + b$, we have:

$$\frac{\partial z}{\partial W} = a$$

Therefore, for a batch of samples:

$$\boxed{\frac{\partial L}{\partial W} = a^T \delta}$$

where a is the matrix of inputs to the layer, and δ is the error term propagated from the output.

Step 3. Gradient w.r.t. Biases b

The bias b is added directly to z, so:

$$\frac{\partial z}{\partial b} = 1$$

which gives:

$$\boxed{\frac{\partial L}{\partial b} = \sum_{\text{samples}} \delta}$$

We sum over the batch dimension because each sample contributes to the bias gradient.

Step 4. Gradient w.r.t. Inputs a (for Propagation)

Finally, we also compute the gradient with respect to the input activations a, which is required to propagate the error to the previous layer:

$$\boxed{\frac{\partial L}{\partial a} = \delta W^T}$$

Result

The gradients for a dense layer are:

$$\boxed{\frac{\partial L}{\partial W} = a^T \delta}, \qquad \boxed{\frac{\partial L}{\partial b} = \sum \delta}, \qquad \boxed{\frac{\partial L}{\partial a} = \delta W^T}$$

These equations will be used repeatedly for each layer during backpropagation. They allow us to efficiently compute weight updates and continue propagating gradients backward through the network.

Here is a simple code implementation

```python
class Layer_Dense:
    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

4.4 Backpropagation with ReLU Activation

For hidden layers, the forward pass consists of two steps:

1. A linear transformation:
$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$
2. An activation function (ReLU):
$$a^{(l)} = f(z^{(l)}) = \max(0, z^{(l)})$$

Here, $a^{(l-1)}$ is the activation from the previous layer, and $a^{(l)}$ is the output activation of the current layer l.

Step 1. Gradient w.r.t. Linear Output $z^{(l)}$

In backpropagation, we assume that we have already computed the gradient $\delta^{(l+1)} = \frac{\partial L}{\partial z^{(l+1)}}$ from the next layer. To propagate this backward, we first compute:

$$\frac{\partial L}{\partial a^{(l)}} = \delta^{(l+1)}(W^{(l+1)})^T$$

where $W^{(l+1)}$ is the weight matrix of the next layer.

Since the activation $a^{(l)}$ depends on $z^{(l)}$ through the ReLU function, we multiply by its derivative:

$$f'(z^{(l)}) = \begin{cases} 1, & \text{if } z^{(l)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Therefore, the gradient with respect to $z^{(l)}$ is:

$$\boxed{\delta^{(l)} = \frac{\partial L}{\partial z^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \odot f'(z^{(l)})}$$

where $\odot$ denotes element-wise multiplication.

Here is a simple code implementation

```python
class Activation_ReLU:
    # Backward pass
    def backward(self, dvalues):
        self.dinputs = dvalues.copy()
        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

Step 2. Gradients w.r.t. Parameters $W^{(l)}$ and $b^{(l)}$

Once $\delta^{(l)}$ is known, the gradients for the layer parameters are computed in the same way as for the output layer:

$$\boxed{\frac{\partial L}{\partial W^{(l)}} = (a^{(l-1)})^T \delta^{(l)}}$$

$$\boxed{\frac{\partial L}{\partial b^{(l)}} = \sum_{\text{samples}} \delta^{(l)}}$$

Step 3. Gradient w.r.t. Inputs $a^{(l-1)}$

Finally, we compute the gradient with respect to the inputs of this layer to propagate it further backward:

$$\boxed{\frac{\partial L}{\partial a^{(l-1)}} = \delta^{(l)} (W^{(l)})^T}$$

Result

The backpropagation for a hidden layer with ReLU consists of:

$$\delta^{(l)} = \left(\delta^{(l+1)} (W^{(l+1)})^T\right) \odot f'(z^{(l)})$$

$$\frac{\partial L}{\partial W^{(l)}} = (a^{(l-1)})^T \delta^{(l)}, \qquad \frac{\partial L}{\partial b^{(l)}} = \sum \delta^{(l)}$$

This process is repeated layer by layer until the gradients for all weights and biases in the network are computed.

5 Building a Neural Network from Scratch

1. Initialize weights and biases

2. Loop through epochs:
    - Forward pass
    - Compute loss
    - Backward pass
    - Update parameters

6 References

- Neural Networks from Scratch in Python
- Neural Networks from Scratch in Python (YouTube Playlist) by Sentdex
- Neural Networks (YouTube Playlist) by 3Blue1Brown