Building Neural Networks: A Hands-On Journey from Scratch with Python

Long Nguyen · Follow
11 min read · Nov 18, 2023

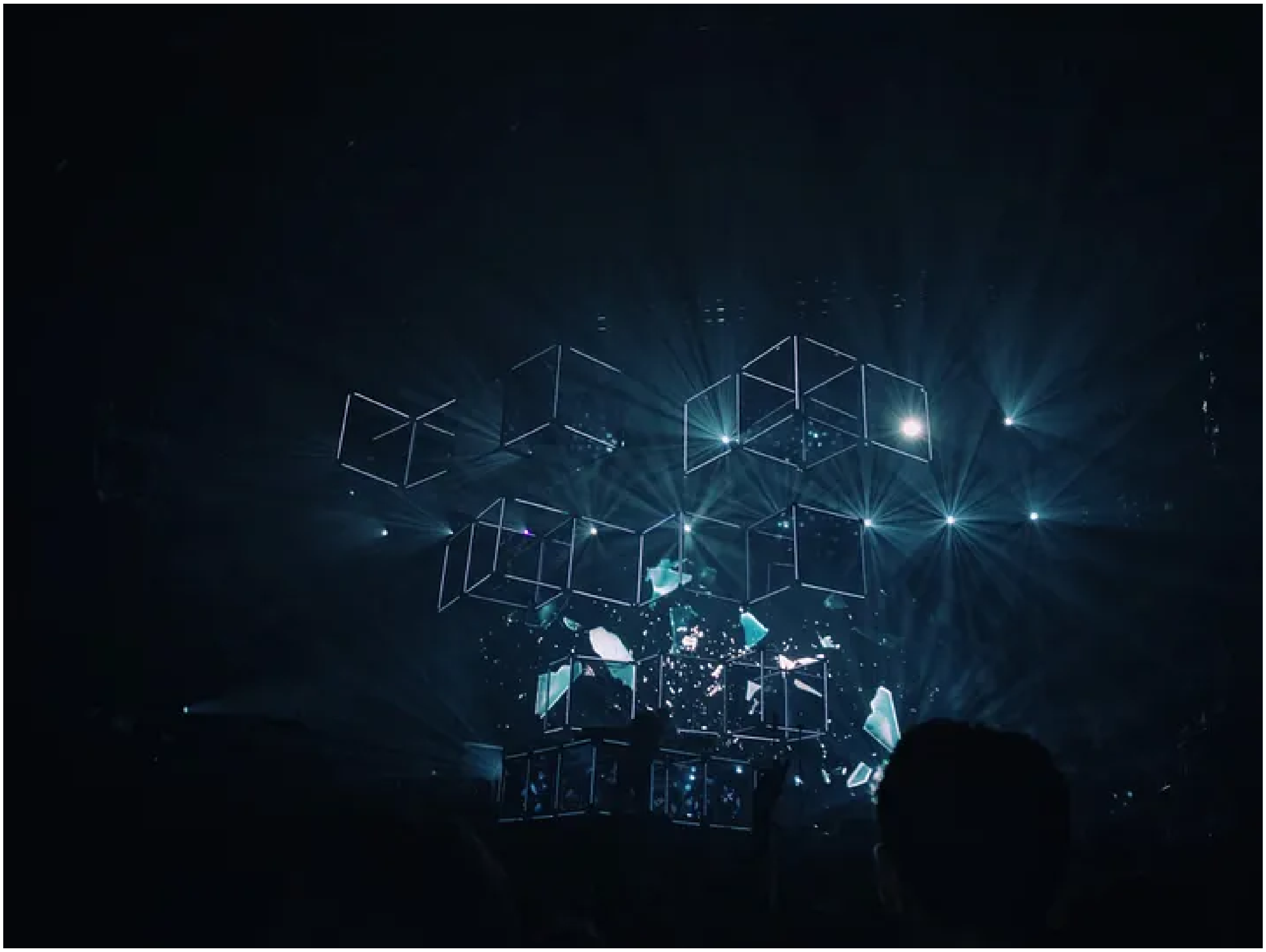


Photo by fabio on Unsplash

In this blog post, we will explore the fundamentals of neural networks, understand the intricacies of forward and backward propagation, and implement a neural network from the ground up with Python in 3 levels!

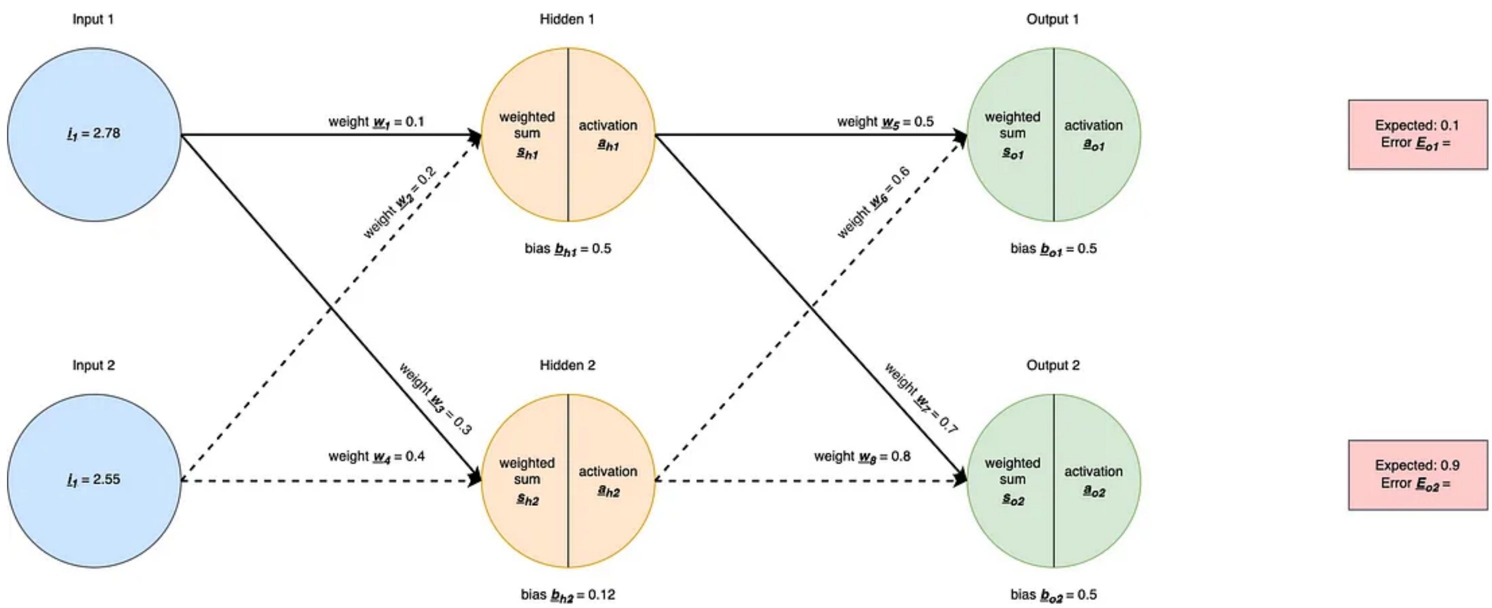
- Level 1: Without using external libraries
- Level 2: With numpy
- Level 3: With Tensorflow

If you are interested in learning about building Recurrent Neural Network from scratch as well, check out this post.

I. Forward and Backward Propagation Walkthrough

But first, let's use an example neural network and work out the mathematical calculation one neuron at a time to understand what's happening behind the scene!

Our sample neural network will consist of: 2 input neurons, 1 hidden layer with 2 neurons and an output layer with 2 neurons. Some initial weights and bias values have been provided to help with the calculation. Assume the expected output is 0.1 and 0.9:



1. Forward Propagation

Note: I'm using sigmoid as the activation function

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

Hidden Layer

Hidden Neuron 1:

$$WeightedSum(s_{h1}) = w_1 \times i_1 + w_2 \times i_2 + b_{h1}$$

$$WeightedSum(s_{h1}) = (2.78 \times 0.1) + (2.55 \times 0.2) + 0.5 = 0.278 + 0.51 + 0.5 = 1.288$$

$$Activation(a_{h1}) = \frac{1}{1 + e^{-s_{h1}}}$$

$$Activation(a_{h1}) = \frac{1}{1 + e^{-1.288}} \approx \frac{1}{1 + 0.275} \approx \frac{1}{1.275} \approx 0.784$$

Hidden Neuron 2:

$$WeightedSum(s_{h2}) = w_3 \times i_1 + w_4 \times i_2 + b_{h2}$$

$$WeightedSum(s_{h2}) = (2.78 \times 0.3) + (2.55 \times 0.4) + 0.12 = 0.834 + 1.02 + 0.12 = 1.974$$

$$Activation(a_{h2}) = \frac{1}{1 + e^{-s_{h2}}}$$

$$Activation(a_{h2}) = \frac{1}{1 + e^{-1.974}} \approx \frac{1}{1 + 0.138} \approx \frac{1}{1.138} \approx 0.878$$

Output Layer

Output Neuron 1:

$$WeightedSum(s_{o1}) = w_5 \times a_{h1} + w_6 \times a_{h2} + b_{o1}$$

$$WeightedSum(s_{o1}) = (0.783 \times 0.5) + (0.878 \times 0.6) + 0.2 = 0.391 + 0.527 + 0.2 = 1.118$$

$$Activation(a_{o1}) = \frac{1}{1 + e^{-s_{o1}}}$$

$$Activation(a_{o1}) = \frac{1}{1 + e^{-1.118}} \approx \frac{1}{1 + 0.327} \approx \frac{1}{1.327} \approx 0.753$$

Output Neuron 2:

$$WeightedSum(s_{o2}) = w_7 \times a_{h1} + w_8 \times a_{h2} + b_{o2}$$

$$WeightedSum(s_{o2}) = (0.783 \times 0.7) + (0.878 \times 0.8) + 0.4 = 0.548 + 0.702 + 0.4 = 1.65$$

$$Activation(a_{o2}) = \frac{1}{1 + e^{-s_{o2}}}$$

$$Activation(a_{o2}) = \frac{1}{1 + e^{-1.65}} \approx \frac{1}{1 + 0.192} \approx \frac{1}{1.192} \approx 0.839$$

Mean Squared Error (MSE) Calculation

Given Expected Outputs: 0.1 (for a_{o1}) and 0.9 (for a_{o2})

$$MSE = \frac{1}{2} \sum (Expected - Actual)^2$$

$$MSE = \frac{1}{2} \left((0.1 - 0.753)^2 + (0.9 - 0.839)^2 \right)$$

$$MSE = \frac{1}{2} (0.414 + 0.003)$$

$$MSE = 0.2085$$

So, the Mean Squared Error (MSE) is approximately 0.2085. This is a measure of the difference between the expected and actual outputs. A lower MSE indicates a better fit of the model to the given data.

2. Backward Propagation

Once predictions are obtained, we need to train the network by adjusting weights and biases based on prediction errors. This is achieved through backward propagation.

Assuming we use the learning rate of 0.5

$$\alpha = 0.5$$

With backpropagation, we want to understand the sensitivity of the error function, which represents the disparity between actual and expected values, to a small adjustment ("nudge") in a particular weight, such as *w5*. I found a lot of value in revising the basics of calculus and derivatives (especially the chain rule) which has helped me grasp how backpropagation works easier. This video does a great job of explaining the intuition https://www.youtube.com/watch?v=tIeHLnjs5U8.

Then, the objective is to diminish the error function by reducing its gradient, thereby facilitating a 'descent' along the gradient — "gradient descent".

Let's start from the output layer and work backwards.

Output Layer

Applying the chain rule to get the formula to calculate the change in error function with respect to a small change in weight *w5*

$$\frac{\partial E_{o1}}{\partial w_5} = \frac{\partial E_{o1}}{\partial a_{o1}} \cdot \frac{\partial a_{o1}}{\partial s_{o1}} \cdot \frac{\partial s_{o1}}{\partial w_5}$$

Let's work out what each component maps to.

First — We've got the error function and its derivative with respect to *ao1*

$$E = \frac{1}{2}\sum_{i=1}^{n}(a_{oi} - expected_i)^2$$

$$\frac{\partial E}{\partial a_{o1}} = 2 \cdot \frac{1}{2} \cdot (a_{o1} - expected_1)$$

$$\frac{\partial E_{o1}}{\partial a_{o1}} = a_{o1} - expected_1$$

Second — the derivative of the activation over the weighted sum, aka the derivative of sigmoid function

$$\frac{\partial a_{o1}}{\partial s_{o1}} = \frac{\partial}{\partial s_{o1}}(Sigmoid(s_{o1})) = a_{o1} \cdot (1 - a_{o1})$$

Lastly — the derivative of the weighted sum with respect to *w5* gives you *ah1* which is the output of the *h1* neuron in the previous layer

$$\frac{\partial s_{o1}}{\partial w_5} = \frac{\partial(w_5 \cdot a_{h1} + w_6 \cdot a_{h2} + b_{o1})}{\partial w_5} = a_{h1}$$

Putting them together

$$\frac{\partial E_{o1}}{\partial w_5} = (a_{o1} - expected_1) \cdot a_{o1} \cdot (1 - a_{o1}) \cdot a_{h1}$$

Usually, we can define a delta as

$$\delta_{o1} = (a_{o1} - expected_1) \cdot a_{o1} \cdot (1 - a_{o1})$$

Then the formula can be shortened to

$$\frac{\partial E_{o1}}{\partial w_5} = \delta_{o1} \cdot a_{h1}$$

This is the gradient of the error function — applying gradient descent to get a new value of weight *w5* by reducing the weight it by the learning rate times gradient

$$w_5' = w_5 - \alpha \cdot \frac{\partial E_{o1}}{\partial w_5} = w_5 - \alpha \cdot \delta_{o1} \cdot a_{h1}$$

Let’s generalise the formulas for the delta of an output neuron, and the formula to update the weight in an output layer.

$$\delta_o = (a_o - expected) \cdot a_o \cdot (1 - a_o)$$

$$w_{new} = w - \alpha \cdot \delta_o \cdot a_h$$

where:
- a_o is the output of the neuron,
- $expected$ is the expected output,
- α is the learning rate,
- δ_o is the delta of the output neuron, and
- a_h is the output of the neuron in the hidden layer connected to this output neuron (aka output of the neuron in previous layer)

From this exercise, you should be able to then derive the formula for updating bias on your own — it's very similar to updating weights. Hint: the final result doesn't involve previous layer neuron's output.

Now let's apply real numbers from the example to those equations to calculate new weights *w5, w6, w7, w8*

Output Neuron 1:

$$\text{Delta } (\delta_{o1}) = (Activation(a_{o1}) - Expected(e_{o1})) \times SigmoidDerivative(a_{o1})$$

$$\text{Delta } (\delta_{o1}) : (0.753 - 0.1) \times (0.753 \times (1 - 0.753)) \approx 0.121$$

$$\text{New Weight } (\mathrm{w}_5) : Weight(w_5) - LearningRate \times Delta(\delta_{o1}) \times Activation(a_{h1})$$

$$\text{New Weight } (\mathrm{w}_5) : 0.5 - 0.5 \times 0.121 \times 0.784 \approx 0.453$$

$$\text{New Weight } (\mathrm{w}_6) : Weight(w_6) - LearningRate \times Delta(\delta_{o1}) \times Activation(a_{h2})$$

$$\text{New Weight } (\mathrm{w}_6) : 0.6 - 0.5 \times 0.121 \times 0.878 \approx 0.547$$

Output Neuron 2:

$$\text{Delta } (\delta_{o2}) = (Activation(a_{o2}) - Expected(e_{o2})) \times SigmoidDerivative(a_{o2})$$

$$\text{Delta } (\delta_{o2}) : (0.839 - 0.9) \times (0.839 \times (1 - 0.839)) \approx -0.008$$

$$\text{New Weight } (\mathrm{w}_7) : Weight(w_7) - LearningRate \times Delta(\delta_{o2}) \times Activation(a_{h1})$$

$$\text{New Weight } (\mathrm{w}_7) : 0.7 - 0.5 \times -0.008 \times 0.784 \approx 0.703$$

$$\text{New Weight } (\mathrm{w}_8) : Weight(w_8) - LearningRate \times Delta(\delta_{o2}) \times Activation(a_{h2})$$

$$\text{New Weight } (\mathrm{w}_8) : 0.8 - 0.5 \times -0.008 \times 0.878 \approx 0.804$$

Hidden Layer

Applying the chain rule again to get the formula to calculate the change in error function with respect to a small change in weight *w1*

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial a_{h1}} \cdot \frac{\partial a_{h1}}{\partial s_{h1}} \cdot \frac{\partial s_{h1}}{\partial w_1}$$

This formula will be a little bit more complicated as we're further away from the output, so a lot more "chaining" of functions will happen so take your time to go through this.

Looking at the first derivative — derivative of total errors with respect to *ah1*. Because total error equals sum of *Eo1* and *Eo2*, using the sum rule we've got

$$\frac{\partial E_{total}}{\partial a_{h1}} = \frac{\partial E_{o1}}{\partial a_{h1}} + \frac{\partial E_{o2}}{\partial a_{h1}}$$

Applying the chain rule to each element

$$\frac{\partial E_{o1}}{\partial a_{h1}} = \frac{\partial E_{o1}}{\partial a_{o1}} \cdot \frac{\partial a_{o1}}{\partial s_{o1}} \cdot \frac{\partial s_{o1}}{\partial a_{h1}}$$

$$\frac{\partial E_{o2}}{\partial a_{h1}} = \frac{\partial E_{o2}}{\partial a_{o2}} \cdot \frac{\partial a_{o2}}{\partial s_{o2}} \cdot \frac{\partial s_{o2}}{\partial a_{h1}}$$

Since we calculated Delta(Eo1) and Delta (Eo2) previously

$$\delta_{o1} = \frac{\partial E_{o1}}{\partial a_{o1}} \cdot \frac{\partial a_{o1}}{\partial s_{o1}} = (a_{o1} - expected_1) \cdot a_{o1} \cdot (1 - a_{o1})$$

$$\delta_{o2} = \frac{\partial E_{o2}}{\partial a_{o2}} \cdot \frac{\partial a_{o2}}{\partial s_{o2}} = (a_{o2} - expected_2) \cdot a_{o2} \cdot (1 - a_{o2})$$

We can substitute those in

$$\frac{\partial E_{o1}}{\partial a_{h1}} = \delta_{o1} \cdot \frac{\partial s_{o1}}{\partial a_{h1}}$$

$$\frac{\partial E_{o2}}{\partial a_{h1}} = \delta_{o2} \cdot \frac{\partial s_{o2}}{\partial a_{h1}}$$

Derivative of the weighted sum with respect to previous layer neuron's output is basically just the corresponding weight

$$\frac{\partial s_{o1}}{\partial a_{h1}} = \frac{\partial (w_5 \cdot a_{h1} + w_6 \cdot a_{h2} + b_{o1})}{\partial a_{h1}} = w_5$$

$$\frac{\partial s_{o2}}{\partial a_{h1}} = \frac{\partial (w_7 \cdot a_{h1} + w_8 \cdot a_{h2} + b_{o2})}{\partial a_{h1}} = w_7$$

The derivative of total error with respect to weight *ah1* now looks like

$$\frac{\partial E_{total}}{\partial a_{h1}} = \frac{\partial E_{o1}}{\partial a_{h1}} + \frac{\partial E_{o2}}{\partial a_{h1}} = \delta_{o1} \cdot w_5 + \delta_{o2} \cdot w_7$$

Substituting this back to the initial formula

$$\frac{\partial E_{total}}{\partial w_1} = (\delta_{o1} \cdot w_5 + \delta_{o2} \cdot w_7) \cdot \frac{\partial a_{h1}}{\partial s_{h1}} \cdot \frac{\partial s_{h1}}{\partial w_1}$$

Derivative of *ah1* over *sh1* is the derivative of the sigmoid function, and the derivative of *sh1* over *w1* is the output of the previous layer neuron (which is the input layer neuron as we only have 1 hidden layer in this example)

$$\frac{\partial a_{h1}}{\partial s_{h1}} = \frac{\partial}{\partial s_{h1}}(Sigmoid(s_{h1})) = a_{h1} \cdot (1 - a_{h1})$$

$$\frac{\partial s_{h1}}{\partial w_1} = \frac{\partial(w_1 \cdot i_1 + w_2 \cdot i_2 + b_{h1})}{\partial w_1} = i_1$$

Putting it together

$$\frac{\partial E_{total}}{\partial w_1} = (\delta_{o1} \cdot w_5 + \delta_{o2} \cdot w_7) \cdot a_{h1} \cdot (1 - a_{h1}) \cdot i_1$$

Let’s group the weighted sum of the deltas in the next layer (output layer) with the sigmoid derivative and call it the Delta(*h1*)

$$\delta_{h1} = (\delta_{o1} \cdot w_5 + \delta_{o2} \cdot w_7) \cdot a_{h1} \cdot (1 - a_{h1})$$

Rewrite the formula of the gradient of error function with respect to *w1*

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} \cdot i_1$$

Applying gradient descent and updating *w1* with learning rate *alpha*

$$w_1' = w_1 - \alpha \cdot \frac{\partial E_{total}}{\partial w_1} = w_1 - \alpha \cdot \delta_{h1} \cdot i_1$$

Let’s generalise the formulas for the delta of a neuron in a hidden layer and the formula to update the weight in hidden layer

$$\delta_h = \left(\sum_o \delta_o \cdot w_o\right) \cdot a_h \cdot (1 - a_h)$$

$$w_{new} = w - \alpha \cdot \delta_h \cdot i$$

where:
- δ_o is the delta of an output neuron,
- w_o is the weight connecting the hidden layer neuron to the output neuron
- a_h is the output of the hidden layer neuron
- $a_h \cdot (1 - a_{h1})$ is its sigmoid derivative
- α is the learning rate,
- δ_h is the delta of the hidden layer neuron, and
- i is the input to the hidden layer neuron (in this case where there's only 1 hidden layer, it is the input neuron, however if you have more than 1 hidden layer, it will be the output of the previous hidden layer neuron)

Now let's apply real numbers from the example to those equations to calculate new weights *w1, w2, w3, w4*

Hidden Neuron 1:

$$\text{Delta } (\delta_{h1}) (Weight(w_5) \times Delta(\delta_{o1}) + Weight(w_7) \times Delta(\delta_{o2})) \times SigmoidDerivative(a_{h1})$$

$$\text{Delta } (\delta_{h1}) : ((0.5 \times 0.121) + (0.7 \times -0.008)) \times (0.784 \times (1 - 0.784)) \approx 0.009$$

$$\text{New Weight } (\mathrm{w}_1) : Weight(w_1) - LearningRate \times Delta(\delta_{h1}) \times Input(i_1)$$

$$\text{New Weight } (\mathrm{w}_1) : 0.1 - 0.5 \times 0.009 \times 2.78 \approx 0.087$$

$$\text{New Weight } (\mathrm{w}_2) : Weight(w_2) - LearningRate \times Delta(\delta_{h1}) \times Input(i_2)$$

$$\text{New Weight } (\mathrm{w}_2) : 0.2 - 0.5 \times 0.009 \times 2.55 \approx 0.189$$

Hidden Neuron 2:

$$\text{Delta } (\delta_{h2}) : (Weight(w_6) \times Delta(\delta_{o1}) + Weight(w_8) \times Delta(\delta_{o2})) \times SigmoidDerivative(a_{h2})$$

$$\text{Delta } (\delta_{h2}) : ((0.6 \times 0.121) + (0.8 \times -0.008)) \times (0.878 \times (1 - 0.878)) \approx 0.007$$

$$\text{New Weight } (\mathrm{w}_3) : Weight(w_3) - LearningRate \times Delta(\delta_{h2}) \times Input(i_1)$$

$$\text{New Weight } (\mathrm{w}_3) : 0.3 - 0.5 \times 0.007 \times 2.78 \approx 0.290$$

$$\text{New Weight } (\mathrm{w}_4) : Weight(w_4) - LearningRate \times Delta(\delta_{h2}) \times Input(i_2)$$

$$\text{New Weight } (\mathrm{w}_4) : 0.4 - 0.5 \times 0.007 \times 2.55 \approx 0.391$$

That's it! All of our weights have been updated — and that was just 1 iteration (epoch). Imagine if we run it thousands or millions of times, the error will become smaller and smaller, hence increasing the accuracy of the network's prediction.

There were a lot of math formulas and calculations and variables so errors are quite likely to occur. If you notice something that is incorrect, please let me know!

II. Level 1: Building A Neural Network Without Using External Libraries

Now that we've covered the math, let's dive into the first level of building a neural network: Without using external libraries (like numpy or PyTorch or Tensorflow)

First, let's define the 2 functions for the sigmoid activation function and its derivative. These 2 will be reused throughout the exercise.

```python
from math import exp

def sigmoid(x: float) -> float:
    return 1.0 / (1.0 + exp(-x))


def sigmoid_derivative(z: float) -> float:
    return z * (1.0 - z)
```

math_utils.py hosted with ❤ by GitHub view raw

Now let's build a class for our Neuron:

- Weights (`weights`): Neurons receive input signals, each associated with a weight. These weights determine the importance of each input.
- Bias (`bias`): Similar to the intercept in a linear equation, the bias allows the neuron to adjust its output independently of the input.
- Delta (`delta`): This is used during the backpropagation process for adjusting weights (you see the knowledge from the walkthrough we've done earlier is coming into our code). It represents the error derivative with respect to the weighted sum.
- Output (`output`): The result of the neuron's activation function.

The sigmoid function introduces non-linearity to the model, enabling it to learn complex patterns.

```python
from dataclasses import dataclass
from typing import List, Optional

from math_utils import sigmoid, sigmoid_derivative


@dataclass
class Neuron:
    weights: List[float]
    bias: float
    delta: Optional[float] = 0.0
    output: Optional[float] = 0.0

    def _set_output(self, output: float) -> None:
        self.output = output

    def set_delta(self, error: float) -> None:
        self.delta = error * sigmoid_derivative(self.output)

    def weighted_sum(self, inputs: List[float]) -> float:
        """
        Usually results in a big number, but we tend to use a value [0, 1] for activation
        Hence, after calculating this, we use the sigmoid function to normalize the result
        """
        ws = self.bias
        for i in range(len(self.weights)):
            ws += self.weights[i] * inputs[i]
        return ws

    def activate(self, inputs: List[float]) -> float:
        """
        Calculates the output of the neuron using a non-linear activation function
        In this case we use the sigmoid function
        """
        output = sigmoid(self.weighted_sum(inputs))
        self._set_output(output)
        return output
```

neuron.py hosted with ❤ by GitHub view raw

Neurons are then organised into layers — here's the Layer class. Layers organise neurons into meaningful groups. Neurons in the same layer share the same input and output dimensions.

```python
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class Layer:
    neurons: List[Neuron]

    @property
    def all_outputs(self) -> List[float]:
        return [neuron.output for neuron in self.neurons]

    def activate_neurons(self, inputs: List[float]) -> List[float]:
        return [neuron.activate(inputs) for neuron in self.neurons]

    def total_delta(self, previous_layer_neuron_idx: int) -> float:
        return sum(
            neuron.weights[previous_layer_neuron_idx] * neuron.delta
            for neuron in self.neurons
        )
```

layer.py hosted with ❤ by GitHub view raw

Now, the bulk of the logic is in the Network class. It represents the neural network itself and orchestrates its training and prediction processes.

Key properties:

- Hidden Layers (`hidden_layers`): A list containing hidden layers, each represented by the `Layer` class.
- Output Layer (`output_layer`): The output layer of the network, also represented by the `Layer` class.
- Learning Rate (`learning_rate`): A hyperparameter determining the step size at each iteration during the training process.

Key functions:

- The `feed_forward` method conducts the forward pass, activating each neuron in sequence, starting from receiving the inputs and progressing through hidden layers to the output layer.
- The `back_propagate` method performs the backpropagation algorithm, calculating and updating the deltas of neurons in each layer. Then it calls `update_weights_for_all_layers` to update the weights after delta calculation is done.
- The `train` method trains the neural network for a specified number of epochs using the provided training set and expected outputs. The `expected` list uses one-hot encoding to indicate the expected output.

```
from dataclasses import dataclass
from typing import List, Optional


@dataclass
class Network:
    hidden_layers: List[Layer]
    output_layer: Layer
    learning_rate: float

    @property
    def layers(self) -> List[Layer]:
        return self.hidden_layers + [self.output_layer]

    def feed_forward(self, inputs: List[float]) -> List[float]:
        for layer in self.hidden_layers:
            # update inputs as outputs of previous layers as we go
            inputs = layer.activate_neurons(inputs)
        return self.output_layer.activate_neurons(inputs)

    def derivative_error_to_output(
        self, actual: List[float], expected: List[float]
    ) -> List[float]:
        """
        Derivative of error function with respect to the output
        """
        return [actual[i] - expected[i] for i in range(len(actual))]

    def back_propagate(self, inputs: List[float], errors: List[float]) -> None:
        """
        Compute the gradient and then update the weights
        """

        # Delta of output layer = derivative of the error functions times the derivative of outp
        # We calculate deltas of output layer first
        # So when we get to hidden layers, the output deltas are ready to be used in calculatior
        for index, neuron in enumerate(self.output_layer.neurons):
            neuron.set_delta(errors[index])

        # Calculate deltas of hidden layer
        for layer_idx in reversed(range(len(self.hidden_layers))):
            layer = self.hidden_layers[layer_idx]
            next_layer = (
                self.output_layer
                if layer_idx == len(self.hidden_layers) - 1
```

```
                if layer_idx == len(self.hidden_layers) - 1
                else self.hidden_layers[layer_idx + 1]
            )
            for neuron_idx, neuron in enumerate(layer.neurons):
                error_from_next_layer = next_layer.total_delta(neuron_idx)
                neuron.set_delta(error_from_next_layer)

        # Only update the weights after you've calculated the deltas
        # If you update the weights as you move through the network, it will affect the deltas (
        self.update_weights_for_all_layers(inputs)

    def update_weights_for_all_layers(self, inputs: List[float]):
        """
        Update weights for all layers
        """
        # Update weights for hidden layers
        for layer_idx in range(len(self.hidden_layers)):
            layer = self.hidden_layers[layer_idx]
            previous_layer_outputs: List[float] = (
                inputs
                if layer_idx == 0
                else self.hidden_layers[layer_idx - 1].all_outputs
            )
            for neuron in layer.neurons:
                self.update_weights_in_a_layer(previous_layer_outputs, neuron)

        # Update weights for output layer
        for index, neuron in enumerate(self.output_layer.neurons):
            self.update_weights_in_a_layer(self.hidden_layers[-1].all_outputs, neuron)

    def update_weights_in_a_layer(
        self, previous_layer_outputs: List[float], neuron: Neuron
    ) -> None:
        """
        Update weights in all neurons in a layer
        """
        for idx in range(len(previous_layer_outputs)):
            neuron.weights[idx] -= (
                self.learning_rate * neuron.delta * previous_layer_outputs[idx]
            )
            neuron.bias -= self.learning_rate * neuron.delta

    def train(
        self,
        num_epoch: int,
```

```python
        num_outputs: int,
        training_set: List[List[float]],
        training_output: List[float],
    ) -> None:
        for epoch in range(num_epoch):
            sum_error = 0.0
            for idx, row in enumerate(training_set):
                expected = [0 for _ in range(num_outputs)]
                expected[training_output[idx]] = 1  # one-hot encoding
                actual = self.feed_forward(row)
                errors = self.derivative_error_to_output(actual, expected)
                self.back_propagate(row, errors)
                sum_error += self.mse(actual, training_output)
            print(f"Mean squared error: {sum_error}")
            print(f"epoch={epoch}")

    def predict(self, inputs: List[float]) -> int:
        outputs = self.feed_forward(inputs)
        return outputs.index(max(outputs))

    def mse(self, actual: List[float], expected: List[float]) -> float:
        """
        Mean Squared Error formula
        """
        return sum((actual[i] - expected[i]) ** 2 for i in range(len(actual))) / len(
            actual
        )
```

network.py hosted with ❤ by GitHub view raw

Now it's time to try to run this code on some sample data. I've reused the data from this tutorial.

This function creates a sample dataset and initialise the network with 1 hidden layer (with 2 neurons) and 1 output layer (with 2 neurons). Then, training is run for 40 epochs with learning rate of 0.5.

The neurons' weights are randomised initially and updated as training goes on.

```
def test_make_prediction_with_network():
    # Test making predictions with the network
    # Mock data is from https://machinelearningmastery.com/implement-backpropagation-algorithm-s
    dataset = [
        [2.7810836, 2.550537003],
        [1.465489372, 2.362125076],
        [3.396561688, 4.400293529],
        [1.38807019, 1.850220317],
        [3.06407232, 3.005305973],
        [7.627531214, 2.759262235],
        [5.332441248, 2.088626775],
        [6.922596716, 1.77106367],
        [8.675418651, -0.242068655],
        [7.673756466, 3.508563011],
    ]
    expected = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
    n_inputs = len(dataset[0])
    n_outputs = len(set(expected))
    hidden_layers = [
        Layer(
            neurons=[
                Neuron(weights=[random() for _ in range(n_inputs)], bias=random()),
                Neuron(weights=[random() for _ in range(n_inputs)], bias=random()),
            ],
        )
    ]
    output_layer = Layer(
        neurons=[
            Neuron(weights=[random() for _ in range(n_outputs)], bias=random()),
            Neuron(weights=[random() for _ in range(n_outputs)], bias=random()),
        ],
    )
    network = Network(
        hidden_layers=hidden_layers, output_layer=output_layer, learning_rate=0.5
    )
    network.train(40, n_outputs, dataset, expected)
    print(f"Hidden layer: {network.layers[0].neurons}")
    print(f"Output layer: {network.layers[1].neurons}")

    # This is just for demonstration only
    for i in range(len(dataset)):
        prediction = network.predict(dataset[i])
        print("Expected=%d, Got=%d" % (expected[i], prediction))


```

```python
if __name__ == "__main__":
    test_make_prediction_with_network()
```

test_make_prediction_with_network.py hosted with ❤ by GitHub view raw

If you run the code, you should get something similar to this

```
Mean squared error: 3.694643563407745
epoch=35
Mean squared error: 3.715320273531503
epoch=36
Mean squared error: 3.7351519986813067
epoch=37
Mean squared error: 3.7541891932363085
epoch=38
Mean squared error: 3.7724787370507626
epoch=39
Hidden layer: [Neuron(weights=[-1.7955933584377923, 2.220092841757106], bias=1.8524769105456214,
Output layer: [Neuron(weights=[3.8411067437736643, -0.4041378519239978], bias=-1.509430106691685
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

sample_terminal_output.sh hosted with ❤ by GitHub view raw

That concludes level 1 — building a neural network without using external libraries. As you can see, most of the math formula we derived from the initial walkthrough is used extensively in the code, so it really helps to do all of the calculation manually before you start implementing the code.

Now, the code is obviously quite lengthy and somewhat complex — let's try to simplify that by using numpy!

III. Level 2: Building A Neural Network With Numpy

Since you are now familiar with the flow of the network, I'll give you all of the code at once:

```python
from __future__ import annotations
from dataclasses import dataclass
from typing import Optional, List

import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(z: float) -> float:
    return z * (1.0 - z)


@dataclass
class Layer:
    weights: np.array
    bias: np.array
    outputs: np.array
    deltas: np.array


@dataclass
class Network:
    layers: List[Layer]
    learning_rate: Optional[int] = 0.5

    @property
    def length(self) -> int:
        return len(self.layers)

    @property
    def outputs(self) -> np.array:
        return self.layers[-1].outputs

    @staticmethod
    def create(
        layers: List[int],
    ) -> Network:
        """
        Create a network with random weights and biases given a list of layers
        The "layers" is a list of the number of neurons in each layer
        """
        layers = [
```

```
        layers = [
            Layer(
                # layers[i] is the number of neurons in layer i (row), layers[i - 1] is the numb
                weights=np.random.rand(layers[i], layers[i - 1]),
                bias=np.random.rand(layers[i]),
                outputs=np.zeros(layers[i]),
                deltas=np.zeros(layers[i]),
            )
            for i in range(1, len(layers))
        ]

        return Network(layers=layers)

    def feed_forward(self, inputs: np.array) -> np.array:
        for layer in self.layers:
            # layer.outputs is a (3,1) - dimension we expect
            # layer.weights is a (3,2), inputs is a (2,1) - multiply to get (3,1)
            layer.outputs = sigmoid(layer.weights @ inputs + layer.bias)  # == np.matmul, https
            inputs = layer.outputs
        return self.layers[-1].outputs

    def back_propagate(self, inputs: np.array, expected: np.array) -> None:
        for idx in reversed(range(self.length)):
            layer = self.layers[idx]
            if idx == len(self.layers) - 1:  # if last layer (output layer)
                layer.deltas = (layer.outputs - expected) * sigmoid_derivative(
                    layer.outputs
                )
            else:
                next_layer = self.layers[idx + 1]
                # layer.deltas is a (3,1) - the dimension we expect
                # next_layer.weights is a (2,3), next_layer.deltas is a (2,1)
                # need to transpose next_layer.weights to get (3,2) then multiply by next_layer
                layer.deltas = (
                    next_layer.weights.T @ next_layer.deltas
                    * sigmoid_derivative(layer.outputs)
                ) * sigmoid_derivative(layer.outputs)

        self.update_weights(inputs)

    def update_weights(self, inputs: np.array) -> None:
        for idx in range(self.length):
            layer = self.layers[idx]
            previous_layer_outputs = self.layers[idx - 1].outputs if idx > 0 else inputs
            # deltas (3,) -> deltas[np.newaxis] (1, 3) -> .T (3, 1)
```

```python
            # previous_layer_outputs (2,) -> previous_layer_outputs[np.newaxis] (1, 2)
            # (3,1) @ (1,2) = (3,2) for weights
            layer.weights -= (
                layer.deltas[np.newaxis].T
                @ previous_layer_outputs[np.newaxis]
                * self.learning_rate
            )
            layer.bias -= layer.deltas * self.learning_rate

    def train(self, inputs: np.array, expected: np.array, epochs: int) -> None:
        for epoch in range(epochs):
            sum_error = 0.0
            for idx, row in enumerate(inputs):
                actual = self.feed_forward(row)
                self.back_propagate(row, expected[idx])
                sum_error += self.mse(actual, expected[idx])
            print(f"Mean squared error: {sum_error}")
            print(f"epoch={epoch}")

    def mse(self, actual: np.array, expected: np.array) -> float:
        return np.power(actual - expected, 2).mean()

    def predict(self, inputs: np.array) -> int:
        outputs = self.feed_forward(inputs)
        return np.where(outputs == outputs.max())[0][0]
```

nn_with_numpy.py hosted with ❤ by GitHub view raw

Using numpy helps us shorten the code a little bit — you can imagine that it's doing "bulk" calculation by utilising matrices instead of looping one neuron and one layer at a time as our previous implementation.

However, you'd need to have a pretty good mental model of the dimensions of the matrices in each step in order to understand and write the correct calculation, which can be a bit challenging. I've put comments in the code about the dimensions expected for most of the calculations (based on the test case).

The `Layer` class is a data class that encapsulates the parameters and attributes associated with a layer in the neural network. We don't need a `Neuron` class anymore since it will just be an element in the numpy array/matrix.

I've added a static method `create` which creates a network with random weights and biases based on the specified number of neurons in each layer. The rest of the functions are the same, except the calculation is done with matrix multiplications instead of manually multiplying each neuron's data.

There are obviously many ways of implementing this — one might simplify this further and remove the `Layer` class completely and represents the whole network with nested arrays. However I find that approach a bit hard to wrap my head around with all of the multiple dimensions so I went with this approach for now.

Let's try running the code with the same dataset

```python

def test_make_prediction_with_network():
    # Test making predictions with the network
    # Mock data is from https://machinelearningmastery.com/implement-backpropagation-algorithm-sc
    dataset = np.array(
        [
            [2.7810836, 2.550537003],
            [1.465489372, 2.362125076],
            [3.396561688, 4.400293529],
            [1.38807019, 1.850220317],
            [3.06407232, 3.005305973],
            [7.627531214, 2.759262235],
            [5.332441248, 2.088626775],
            [6.922596716, 1.77106367],
            [8.675418651, -0.242068655],
            [7.673756466, 3.508563011],
        ]
    )
    expected = np.array(
        [
            [1, 0],
            [1, 0],
            [1, 0],
            [1, 0],
            [1, 0],
            [0, 1],
            [0, 1],
            [0, 1],
            [0, 1],
            [0, 1],
        ]
    )
    # 2 input neurons, 3 hidden neurons, 2 output neurons
    network = Network.create([len(dataset[0]), 3, len(expected[0])])
    network.train(dataset, expected, 40)
    for i in range(len(dataset)):
        prediction = network.predict(dataset[i])
        print(
            f"{i} - Expected={np.where(expected[i] == expected[i].max())[0][0]}, Got={prediction
        )


if __name__ == "__main__":
    test_make_prediction_with_network()
```

test_make_prediction_with_network_numpy.py hosted with ❤ by GitHub view raw

The dimensions of input and expected output are obviously slightly changed to fit with numpy n-D arrays, but the data stay the same.

This is a sample output

```
Mean squared error: 1.1978125477947856
epoch=36
Mean squared error: 1.1020105642439684
epoch=37
Mean squared error: 1.013486082749448
epoch=38
Mean squared error: 0.9322033466403574
epoch=39
0 - Expected=0, Got=0
1 - Expected=0, Got=0
2 - Expected=0, Got=0
3 - Expected=0, Got=0
4 - Expected=0, Got=0
5 - Expected=1, Got=1
6 - Expected=1, Got=1
7 - Expected=1, Got=1
8 - Expected=1, Got=1
9 - Expected=1, Got=1
```

nump-sample-output.sh hosted with ❤ by GitHub view raw

III. Level 3: Building A Neural Network With Tensorflow

In this level, we've transitioned from a detailed, 200-line implementation of a neural network to just a few concise lines using TensorFlow. The power of TensorFlow allows us to express neural network architectures with ease.

However, I won't go into details about this code, and Tensorflow in general as our aim in this blog is not to delve into the intricacies of TensorFlow but to comprehend the fundamental workings of a neural network. TensorFlow abstracts away many of the underlying details, making it an efficient tool for practical applications but potentially not the best way to learn.

This is merely to demonstrate that neural network is complex and to fully understand it, it's recommended to attempt to build one from scratch. Starting from the basics lays a solid foundation, enabling a deeper understanding of the complexities involved. While libraries like TensorFlow offer convenience, diving into their usage without a fundamental understanding of neural networks can hinder comprehensive learning.

```python
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers


def build_model() -> tf.keras.Sequential:
    model = tf.keras.Sequential(
        [
            layers.Dense(units=3, activation="sigmoid", input_shape=(2,)),
            layers.Dense(units=2),
        ]
    )
    model.summary()
    loss_fn = keras.losses.MeanSquaredError()
    model.compile(optimizer="adam", loss=loss_fn, metrics=["accuracy"])
    return model


if __name__ == "__main__":
    dataset = np.array(
        [
            [2.7810836, 2.550537003],
            [1.465489372, 2.362125076],
            [3.396561688, 4.400293529],
            [1.38807019, 1.850220317],
            [3.06407232, 3.005305973],
            [7.627531214, 2.759262235],
            [5.332441248, 2.088626775],
            [6.922596716, 1.77106367],
            [8.675418651, -0.242068655],
            [7.673756466, 3.508563011],
        ]
    )
    expected = np.array(
        [
            [1, 0],
            [1, 0],
            [1, 0],
            [1, 0],
            [1, 0],
            [0, 1],
            [0, 1],
            [0, 1],
            [0, 1]
```

```
            [0, 1],
            [0, 1],
        ]
    )
    model = build_model()
    # Convert the data to TensorFlow format
    dataset_tf = tf.constant(dataset, dtype=tf.float32)
    expected_tf = tf.constant(expected, dtype=tf.float32)

    # Train the model
    model.fit(dataset_tf, expected_tf, epochs=200)
    predictions = model.predict(dataset_tf)
    print(predictions)

    # Convert continuous predictions to class labels (0 or 1)
    class_predictions = np.argmax(predictions, axis=1)
    print(class_predictions)

    # Print the comparison
    for i, (expected_row, prediction) in enumerate(zip(expected, class_predictions)):
        print(f"{i} - Expected={expected_row.argmax()}, Got={prediction}")
```

nn_with_tf.py hosted with ❤ by GitHub view raw

Sample output

```
Epoch 196/200
1/1 [==============================] - 0s 784us/step - loss: 0.0737 - accuracy: 1.0000
Epoch 197/200
1/1 [==============================] - 0s 691us/step - loss: 0.0733 - accuracy: 1.0000
Epoch 198/200
1/1 [==============================] - 0s 643us/step - loss: 0.0729 - accuracy: 1.0000
Epoch 199/200
1/1 [==============================] - 0s 629us/step - loss: 0.0725 - accuracy: 1.0000
Epoch 200/200
1/1 [==============================] - 0s 640us/step - loss: 0.0721 - accuracy: 1.0000
1/1 [==============================] - 0s 27ms/step
[[0.66456985 0.39463678]
 [0.75454247 0.1885925 ]
 [0.92614865 0.29184788]
 [0.6771406  0.21432315]
 [0.7149838  0.39005262]
 [0.17372087 0.8059198 ]
 [0.2435018  0.72199327]
 [0.19464082 0.74787194]
 [0.2519498  0.5949259 ]
 [0.17731054 0.8268534 ]]
[0 0 0 0 0 1 1 1 1 1]
0 - Expected=0, Got=0
1 - Expected=0, Got=0
2 - Expected=0, Got=0
3 - Expected=0, Got=0
4 - Expected=0, Got=0
5 - Expected=1, Got=1
6 - Expected=1, Got=1
7 - Expected=1, Got=1
8 - Expected=1, Got=1
9 - Expected=1, Got=1
```

sample_terminal_output_tf.sh hosted with ❤ by GitHub view raw

In this blog journey, we took a dive into the behind the scene of neural networks, starting from the basic walkthrough with math calculation and then moving into code implementation with Python. We built these

As we wrap up, I invite you to try this out for yourself. Coding is a journey of discovery, and building a neural network from scratch is like a backstage tour. So, grab your coding gear, start tinkering, and enjoy the adventure of learning. Happy coding!

References and resources:

- Gradient descent, how neural networks learn
- What is backpropagation really doing?
- The essence of calculus
- Backpropagation calculus
- A Step by Step Backpropagation Example
- How to Code a Neural Network with Backpropagation In Python (from scratch)
- Difference between numpy dot() and Python 3.5+ matrix multiplication
- CHAPTER 2 — How the backpropagation algorithm works

Neural Networks | Backpropagation | Numpy | Artificial Neural Network | Deep Learning

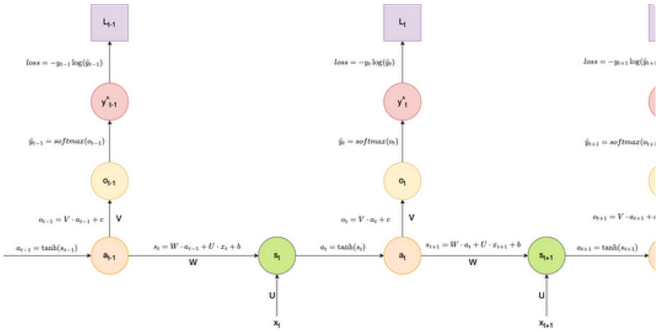
Written by Long Nguyen

Follow

202 Followers

Software Engineer @ Atlassian

More from Long Nguyen



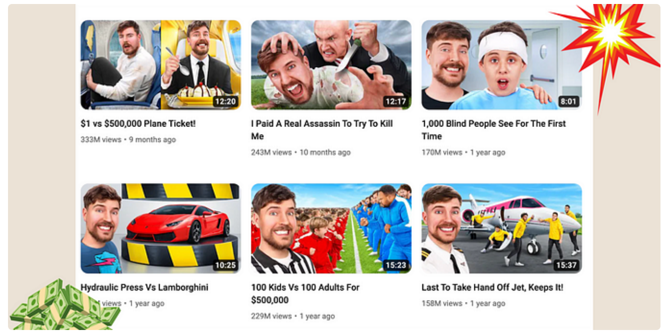
Long Nguyen

Building a Recurrent Neural Network From Scratch

In this blog post, we will explore Recurrent Neural Networks (RNNs) and the...

14 min read · Jan 28, 2024

77



Sinem Günel in The Startup

I Constantly Use MrBeast's Strategies to Grow My Business—...

3 psychological biases you won't be able to unsee anymore

· 11 min read · Feb 13, 2024

5.1K 86

Desiree Peralta in The Startup

One of The Most Profitable 1-Person Business Anyone Can Star...

Regardless of your knowledge, experience, and social media presence.

· 7 min read · Feb 2, 2024

2.6K 51

Long Nguyen in The Startup

Large Language Models & Transformer Architecture: The...

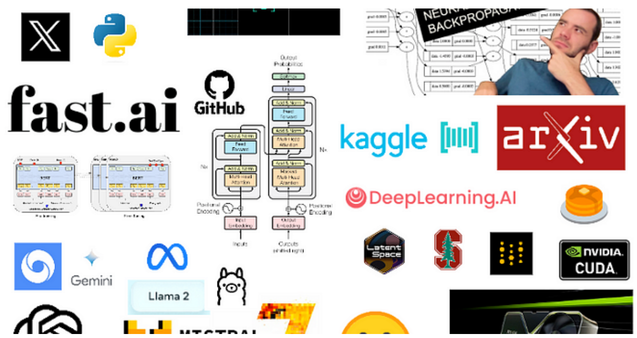
Explore LLMs and the Transformer architecture: from tokenisation, embeddings...

15 min read · Jul 25, 2023

16

See all from Long Nguyen

Recommended from Medium



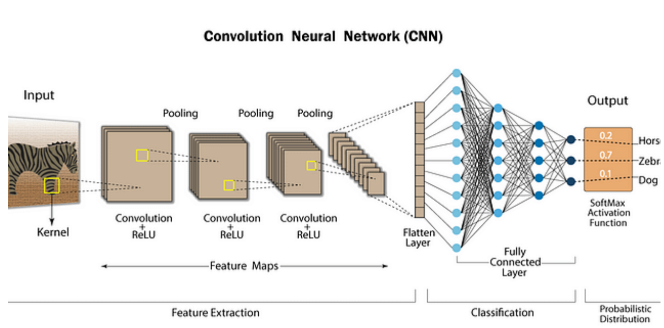
Benedict Neo in bitgrit Data Science Publication

Roadmap to Learn AI in 2024

A free curriculum for hackers and programmers to learn AI

11 min read · 3 days ago

2.2K 26



Koushik

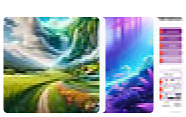
Understanding Convolutional Neural Networks (CNNs) in Depth

Convolutional Neural Networks skillfully capturing and extracting patterns from data,...

12 min read · Nov 28, 2023

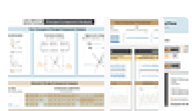
583 5

Lists



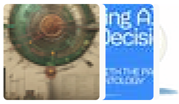
Natural Language Processing

1224 stories · 706 saves



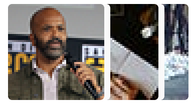
Practical Guides to Machine Learning

10 stories · 1096 saves



data science and AI

40 stories · 83 saves



Staff Picks

587 stories · 776 saves

Fareed Khan in Level Up Coding

Solving Transformer by Hand: A Step-by-Step Math Example

Performing numerous matrix multiplications to solve the encoder and decoder parts of th...

13 min read · Dec 18, 2023

1.7K 25

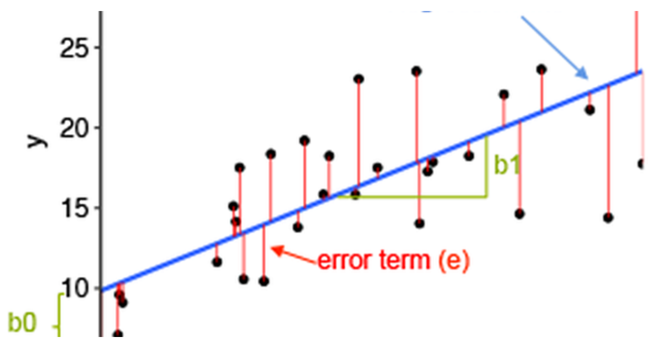
Jason Roell

Ultimate Python Cheat Sheet: Practical Python For Everyday...

This Cheat Sheet was born out of necessity. Recently, I was tasked with diving into a new...

33 min read · Jan 30, 2024

2.1K 27

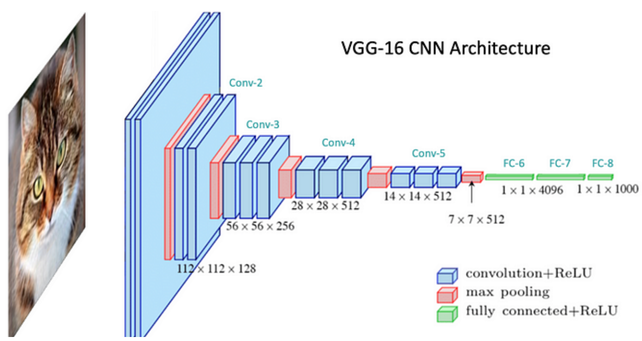


Asad iqbal

Demystifying Machine Learning: A Guided Tour of the Top 10...

1. Linear Regression

19 min read · Sep 16, 2023



Luís Fernando Torres in LatinXinAI

Convolutional Neural Network From Scratch

The most effective way of working with image data

21 min read · Oct 16, 2023