Published in Towards Data Science

Victor Zhou Follow
Mar 6, 2019 · 9 min read · Member-only · Listen

Machine Learning for Beginners: An Introduction to Neural Networks

A simple explanation of how they work and how to implement one from scratch in Python.

Here's something that might surprise you: **neural networks aren't that complicated!** The term "neural network" gets used as a buzzword a lot, but in reality they're often much simpler than people imagine.

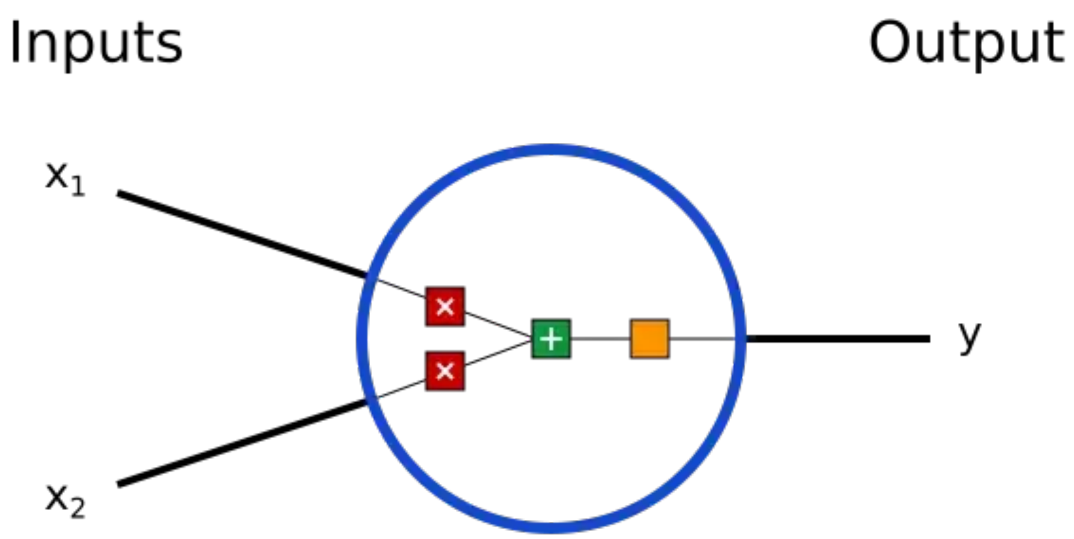
This post is intended for complete beginners and assumes ZERO prior knowledge of machine learning. We'll understand how neural networks work while implementing one from scratch in Python.

Let's get started!

> *Note: I recommend reading this post on victorzhou.com — much of the formatting in this post looks better there.*

1. Building Blocks: Neurons

First, we have to talk about neurons, the basic unit of a neural network. A **neuron takes inputs, does some math with them, and produces one output.** Here's what a 2-input neuron looks like:



3 things are happening here. First, each input is multiplied by a weight:

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

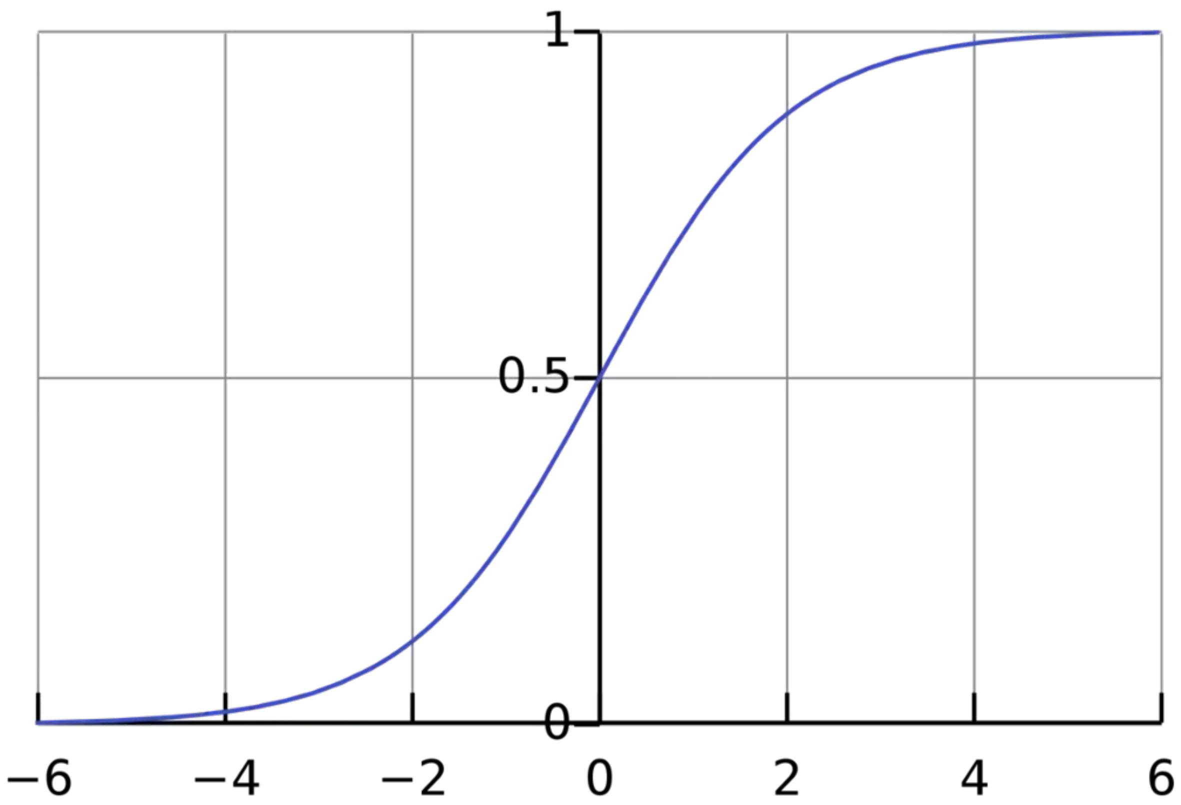
Next, all the weighted inputs are added together with a bias b:

$$(x_1 * w_1) + (x_2 * w_2) + b$$

Finally, the sum is passed through an activation function:

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

The activation function is used to turn an unbounded input into an output that has a nice, predictable form. A commonly used activation function is the sigmoid function:



The sigmoid function only outputs numbers in the range (0,1). You can think of it as compressing (−∞,+∞) to (0,1) — big negative numbers become ~0, and big positive numbers become ~1.

A Simple Example

> *Reminder: much of the formatting in this article looks better in the original post on victorzhou.com.*

Assume we have a 2-input neuron that uses the sigmoid activation function and has the following parameters:

$$w = [0, 1]$$
$$b = 4$$

w=[0, 1] is just a way of writing $w1$=0, $w2$=1 in vector form. Now, let's give the neuron an input of x=[2, 3]. We'll use the dot product to write things more concisely:

$$
\begin{aligned}
(w \cdot x) + b &= ((w_1 * x_1) + (w_2 * x_2)) + b \\
&= 0 * 2 + 1 * 3 + 4 \\
&= 7
\end{aligned}
$$

$$y = f(w \cdot x + b) = f(7) = \boxed{0.999}$$

The neuron outputs 0.999 given the inputs *x*=[2,3]. That's it! This process of passing inputs forward to get an output is known as **feedforward**.

Coding a Neuron

Time to implement a neuron! We'll use NumPy, a popular and powerful computing library for Python, to help us do math:

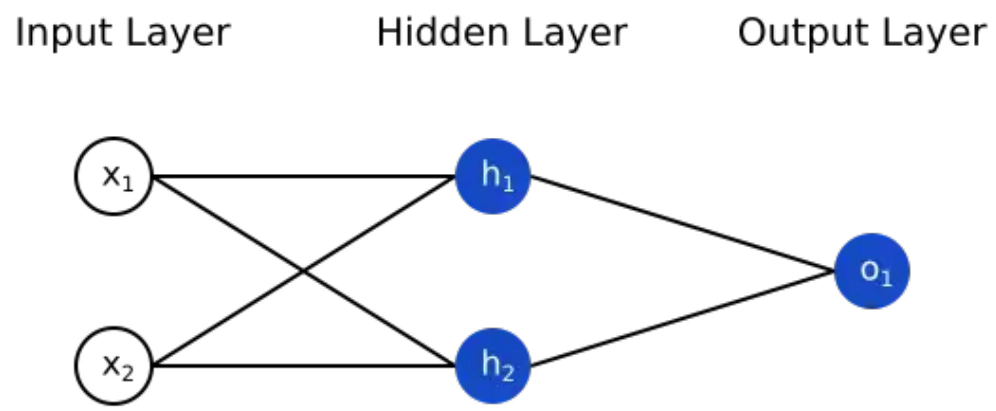
```
import numpy as np

def sigmoid(x):
  # Our activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    # Weight inputs, add bias, then use the activation function
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                   # b = 0
n = Neuron(weights, bias)

x = np.array([2, 3])       # x1 = 2, x2 = 3
print(n.feedforward(x))    # 0.9990889488055994
```

neuron.py hosted with ❤ by GitHub view raw

Recognize those numbers? That's the example we just did! We get the same answer of 0.999.

2. Combining Neurons into a Neural Network

A neural network is nothing more than a bunch of neurons connected together. Here's what a simple neural network might look like:



This network has 2 inputs, a hidden layer with 2 neurons ($h1$ and $h2$), and an output layer with 1 neuron ($o1$). Notice that the inputs for $o1$ are the outputs from $h1$ and $h2$ — that's what makes this a network.

> *A hidden layer is any layer between the input (first) layer and output (last) layer. There can be multiple hidden layers!*

An Example: Feedforward

Let's use the network pictured above and assume all neurons have the same weights w=[0,1], the same bias b=0, and the same sigmoid activation function. Let $h1$, $h2$, $o1$ denote the *outputs* of the neurons they represent.

What happens if we pass in the input x=[2, 3]?

$$
\begin{aligned}
h_1 = h_2 &= f(w \cdot x + b) \\
&= f((0 * 2) + (1 * 3) + 0) \\
&= f(3) \\
&= 0.9526
\end{aligned}
$$

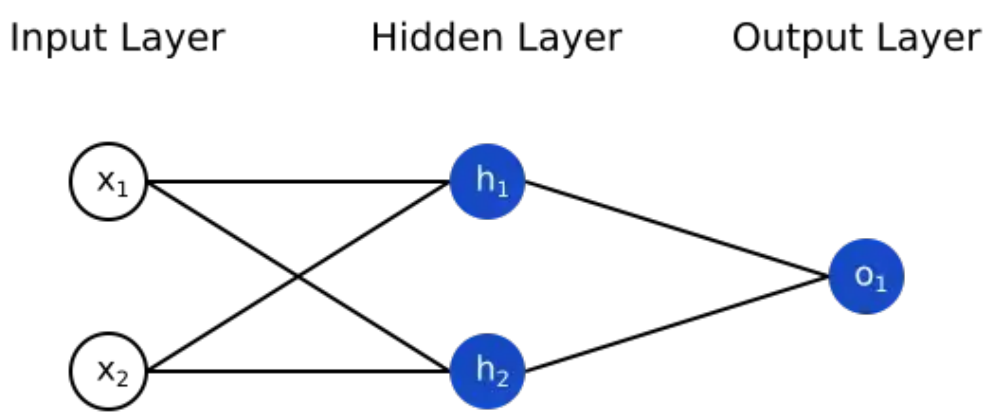
$$
\begin{aligned}
o_1 &= f(w \cdot [h_1, h_2] + b) \\
&= f((0 * h_1) + (1 * h_2) + 0) \\
&= f(0.9526) \\
&= \boxed{0.7216}
\end{aligned}
$$

The output of the neural network for input $x=[2,3]$ is 0.7216. Pretty simple, right?

A neural network can have **any number of layers** with **any number of neurons** in those layers. The basic idea stays the same: feed the input(s) forward through the neurons in the network to get the output(s) at the end. For simplicity, we'll keep using the network pictured above for the rest of this post.

Coding a Neural Network: Feedforward

Let's implement feedforward for our neural network. Here's the image of the network again for reference:



```python
import numpy as np

# ... code from previous section here

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)
  Each neuron has the same weights and bias:
    - w = [0, 1]
    - b = 0
  '''
  def __init__(self):
    weights = np.array([0, 1])
    bias = 0

    # The Neuron class here is from the previous section
    self.h1 = Neuron(weights, bias)
    self.h2 = Neuron(weights, bias)
    self.o1 = Neuron(weights, bias)

  def feedforward(self, x):
    out_h1 = self.h1.feedforward(x)
    out_h2 = self.h2.feedforward(x)

    # The inputs for o1 are the outputs from h1 and h2
    out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))

    return out_o1

network = OurNeuralNetwork()
x = np.array([2, 3])
print(network.feedforward(x)) # 0.7216325609518421
```

network.py hosted with ❤ by GitHub view raw

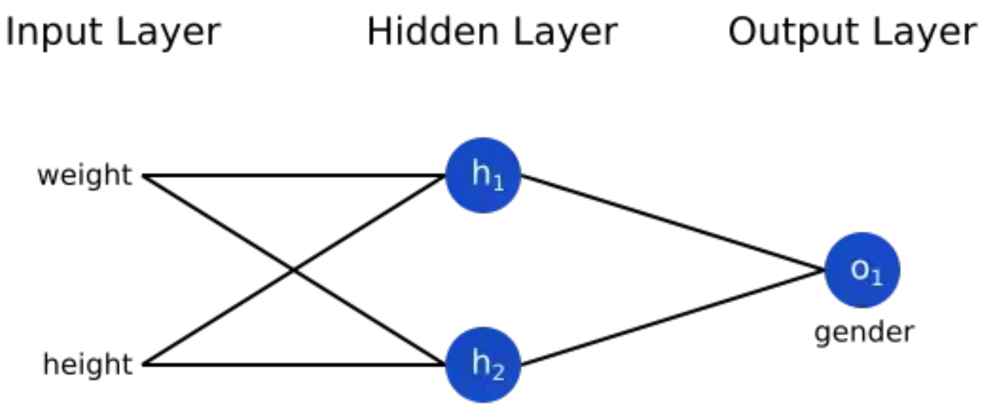
We got 0.7216 again! Looks like it works.

3. Training a Neural Network, Part 1

Say we have the following measurements:

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

Let's train our network to predict someone's gender given their weight and height:



We'll represent Male with a 0 and Female with a 1, and we'll also shift the data to make it easier to use:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

Loss

Before we train our network, we first need a way to quantify how "good" it's doing so that it can try to do "better". That's what the **loss** is.

We'll use the **mean squared error** (MSE) loss:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_{true} - y_{pred})^2$$

Let's break this down:

- n is the number of samples, which is 4 (Alice, Bob, Charlie, Diana).
- y represents the variable being predicted, which is Gender.
- *y_true* is the *true* value of the variable (the "correct answer"). For example, *y_true* for Alice would be 1 (Female).
- *y_pred* is the *predicted* value of the variable. It's whatever our network outputs.

$(y\_true - y\_pred)^2$ is known as the **squared error**. Our loss function is simply taking the average over all squared errors (hence the name *mean* squared error). The better our predictions are, the lower our loss will be!

Better predictions = Lower loss.

Training a network = trying to minimize its loss.

An Example Loss Calculation

Let's say our network always outputs 00 — in other words, it's confident all humans are Male 🤔. What would our loss be?

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

$$\text{MSE} = \frac{1}{4}(1 + 0 + 0 + 1) = \boxed{0.5}$$

Code: MSE Loss

Here's some code to calculate loss for us:

```
import numpy as np

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])

print(mse_loss(y_true, y_pred)) # 0.5
```

loss.py hosted with ❤ by GitHub   view raw

If you don't understand why this code works, read the NumPy quickstart on array operations.

Nice. Onwards!

> *Liking this post so far? I write a lot of beginner-friendly ML articles. Subscribe to my newsletter to get them in your inbox!*

4. Training a Neural Network, Part 2

We now have a clear goal: **minimize the loss** of the neural network. We know we can change the network's weights and biases to influence its predictions, but how do we do so in a way that decreases loss?

> *This section uses a bit of multivariable calculus. If you're not comfortable with calculus, feel free to skip over the math parts.*

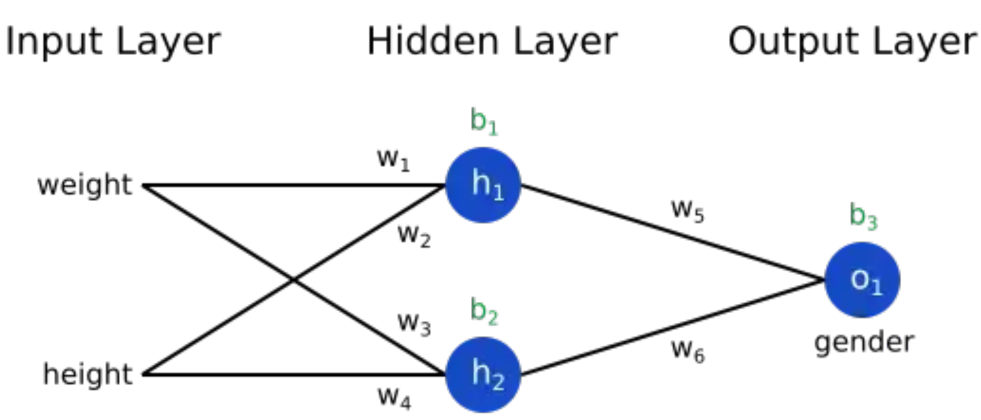
For simplicity, let's pretend we only have Alice in our dataset:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Then the mean squared error loss is just Alice's squared error:

$$\begin{aligned}\text{MSE} &= \frac{1}{1}\sum_{i=1}^{1}(y_{true} - y_{pred})^2 \\ &= (y_{true} - y_{pred})^2 \\ &= (1 - y_{pred})^2\end{aligned}$$

Another way to think about loss is as a function of weights and biases. Let's label each weight and bias in our network:



Then, we can write loss as a multivariable function:

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

Imagine we wanted to tweak *w*1. How would loss *L* change if we changed *w*1? That's a question the partial derivative can answer. How do we calculate it?

> *Here's where the math starts to get more complex.* ***Don't be discouraged!*** *I recommend getting a pen and paper to follow along — it'll help you understand.*
>
> *If you have trouble reading this: the formatting for the math below looks better in the original post on victorzhou.com.*

To start, let's rewrite the partial derivative in terms of ∂*y_pred*/∂*w*1 instead:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

This works because of the Chain Rule.

We can calculate $\partial L/\partial y\_pred$ because we computed L= $(1-y\_pred)^2$ above:

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1-y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1-y_{pred})}$$

Now, let's figure out what to do with $\partial y\_pred/\partial w1$. Just like before, let $h1$, $h2$, $o1$ be the outputs of the neurons they represent. Then

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

f is the sigmoid activation function, remember?

Since $w1$ only affects $h1$ (not $h2$), we can write

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

More Chain Rule.

We do the same thing for $\partial h1/\partial w1$:

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

You guessed it, Chain Rule.

$x1$ here is weight, and $x2$ is height. This is the second time we've seen $f'(x)$ (the derivate of the sigmoid function) now! Let's derive it:

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

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

We'll use this nice form for $f'(x)$ later.

We're done! We've managed to break down $\partial L/\partial w1$ into several parts we can calculate:

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

This system of calculating partial derivatives by working backwards is known as **backpropagation**, or "backprop".

Phew. That was a lot of symbols — it's alright if you're still a bit confused. Let's do an example to see this in action!

Example: Calculating the Partial Derivative

We're going to continue pretending only Alice is in our dataset:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Let's initialize all the weights to 1 and all the biases to 0. If we do a feedforward pass through the network, we get:

$$\begin{aligned} h_1 &= f(w_1 x_1 + w_2 x_2 + b_1) \\ &= f(-2 + -1 + 0) \\ &= 0.0474 \end{aligned}$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$\begin{aligned} o_1 &= f(w_5 h_1 + w_6 h_2 + b_3) \\ &= f(0.0474 + 0.0474 + 0) \\ &= 0.524 \end{aligned}$$

The network outputs *y_pred*=0.524, which doesn't strongly favor Male (0) or Female (1). Let's calculate ∂*L*/∂*w*1:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\begin{aligned} \frac{\partial L}{\partial y_{pred}} &= -2(1 - y_{pred}) \\ &= -2(1 - 0.524) \\ &= -0.952 \end{aligned}$$

$$\begin{aligned} \frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \\ &= 1 * f'(0.0474 + 0.0474 + 0) \\ &= f(0.0948) * (1 - f(0.0948)) \\ &= 0.249 \end{aligned}$$

$$\begin{aligned} \frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1) \\ &= -2 * f'(-2 + -1 + 0) \\ &= -2 * f(-3) * (1 - f(-3)) \\ &= -0.0904 \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\ &= \boxed{0.0214} \end{aligned}$$

> *Reminder: we derived $f'(x) = f(x) * (1 - f(x))$ for our sigmoid activation function earlier.*

We did it! This tells us that if we were to increase $w1$, L would increase a *tiiiny* bit as a result.

Training: Stochastic Gradient Descent

We have all the tools we need to train a neural network now! We'll use an optimization algorithm called stochastic gradient descent (SGD) that tells us how to change our weights and biases to minimize loss. It's basically just this update equation:

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η is a constant called the **learning rate** that controls how fast we train. All we're doing is subtracting $\eta \, \partial w1/\partial L$ from $w1$:

- If $\partial L/\partial w1$ is positive, $w1$ will decrease, which makes L decrease.
- If $\partial L/\partial w1$ is negative, $w1$ will increase, which makes L decrease.

If we do this for every weight and bias in the network, the loss will slowly decrease and our network will improve.

Our training process will look like this:

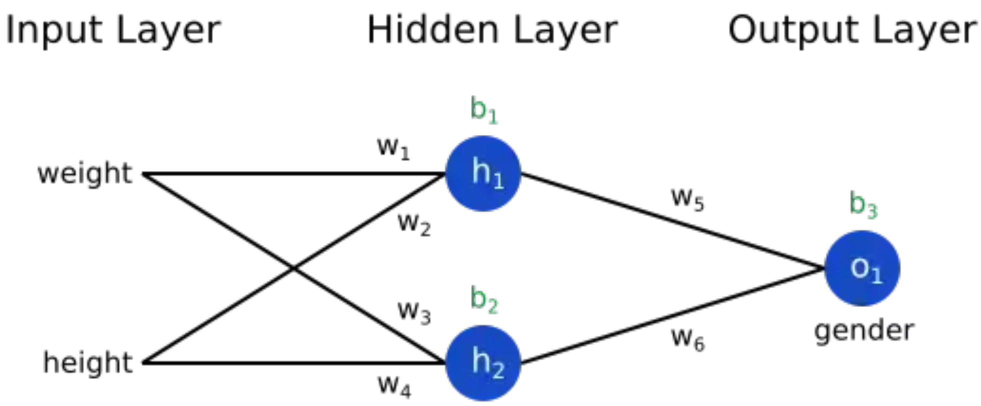
1. Choose **one** sample from our dataset. This is what makes it *stochastic* gradient descent — we only operate on one sample at a time.
2. Calculate all the partial derivatives of loss with respect to weights or biases (e.g. $\partial L/\partial w1$, $\partial L/\partial w2$, etc).
3. Use the update equation to update each weight and bias.
4. Go back to step 1.

Let's see it in action!

Code: A Complete Neural Network

It's *finally* time to implement a complete neural network:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1



```
import numpy as np

def sigmoid(x):
  # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)

  *** DISCLAIMER ***:
  The code below is intended to be simple and educational, NOT optimal.
  Real neural net code looks nothing like this. DO NOT use this code.
  Instead, read/run it to understand how this specific network works.
  '''
  def __init__(self):
    # Weights
    self.w1 = np.random.normal()
    self.w2 = np.random.normal()
    self.w3 = np.random.normal()
    self.w4 = np.random.normal()
    self.w5 = np.random.normal()
    self.w6 = np.random.normal()

    # Biases
    self.b1 = np.random.normal()
    self.b2 = np.random.normal()
    self.b3 = np.random.normal()

  def feedforward(self, x):
    # x is a numpy array with 2 elements.
    h1 = sigmoid(self.w1 * x[0] + self.w2 * x[1] + self.b1)
    h2 = sigmoid(self.w3 * x[0] + self.w4 * x[1] + self.b2)
    o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
    return o1

  def train(self, data, all_y_trues):
    '''
    - data is a (n x 2) numpy array, n = # of samples in the dataset.
    - all_y_trues is a numpy array with n elements.
```

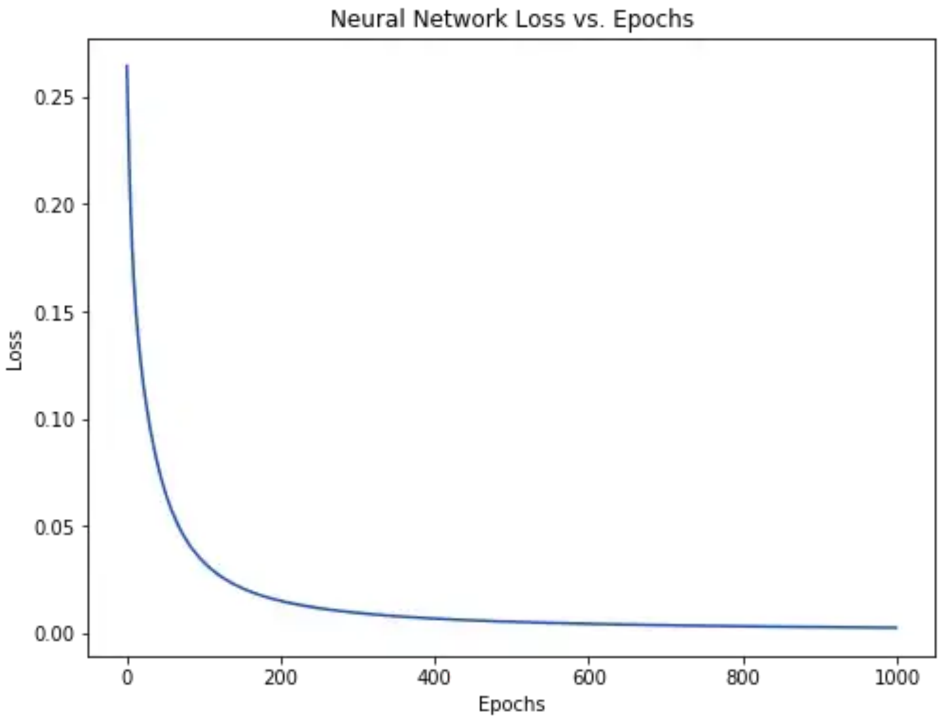
```python
      Elements in all_y_trues correspond to those in data.
    '''
    learn_rate = 0.1
    epochs = 1000 # number of times to loop through the entire dataset

    for epoch in range(epochs):
      for x, y_true in zip(data, all_y_trues):
        # --- Do a feedforward (we'll need these values later)
        sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
        h1 = sigmoid(sum_h1)

        sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
        h2 = sigmoid(sum_h2)

        sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
        o1 = sigmoid(sum_o1)
        y_pred = o1

        # --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial w1"
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Neuron o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)

        d_ypred_d_h1 = self.w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = self.w6 * deriv_sigmoid(sum_o1)

        # Neuron h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)

        # Neuron h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)

        # --- Update weights and biases
        # Neuron h1
        self.w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
        self.w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
        self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

        # Neuron h2
        self.w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
        self.w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
        self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2

        # Neuron o1
```

```python
        self.w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        self.w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3

      # --- Calculate total loss at the end of each epoch
      if epoch % 10 == 0:
        y_preds = np.apply_along_axis(self.feedforward, 1, data)
        loss = mse_loss(all_y_trues, y_preds)
        print("Epoch %d loss: %.3f" % (epoch, loss))

# Define dataset
data = np.array([
  [-2, -1],  # Alice
  [25, 6],   # Bob
  [17, 4],   # Charlie
  [-15, -6], # Diana
])
all_y_trues = np.array([
  1, # Alice
  0, # Bob
  0, # Charlie
  1, # Diana
])

# Train our neural network!
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```

fullnetwork.py hosted with ❤ by **GitHub** **view raw**

> *You can run / play with this code yourself. It's also available on Github.*

Our loss steadily decreases as the network learns:



We can now use the network to predict genders:

```
# Make some predictions
emily = np.array([-7, -3]) # 128 pounds, 63 inches
frank = np.array([20, 2])  # 155 pounds, 68 inches
print("Emily: %.3f" % network.feedforward(emily)) # 0.951 - F
print("Frank: %.3f" % network.feedforward(frank)) # 0.039 - M
```

predict.py hosted with ❤ by **GitHub**

Now What?

You made it! A quick recap of what we did:

- Introduced **neurons**, the building blocks of neural networks.
- Used the **sigmoid activation function** in our neurons.
- Saw that neural networks are just neurons connected together.
- Created a dataset with Weight and Height as inputs (or **features**) and Gender as the output (or **label**).
- Learned about **loss functions** and the **mean squared error** (MSE) loss.
- Realized that training a network is just minimizing its loss.

- Used **backpropagation** to calculate partial derivatives.

- Used **stochastic gradient descent** (SGD) to train our network.

There's still much more to do:

- Experiment with bigger / better neural networks using proper machine learning libraries like Tensorflow, Keras, and PyTorch.
- Build your first neural network with Keras.
- Tinker with a neural network in your browser.
- Discover other activation functions besides sigmoid, like Softmax.
- Discover other optimizers besides SGD.
- Read my introduction to Convolutional Neural Networks (CNNs). CNNs revolutio the field of Computer Vision and can be extremely powerful.
- Read my introduction to Recurrent Neural Networks (RNNs), which are often use Natural Language Processing (NLP).

I may write about these topics or similar ones in the future, so subscribe if you want notified about new posts.

Thanks for reading!

Originally posted on victorzhou.com.