
Victor Zhou

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.

MARCH 3, 2019 | UPDATED JULY 24, 2019

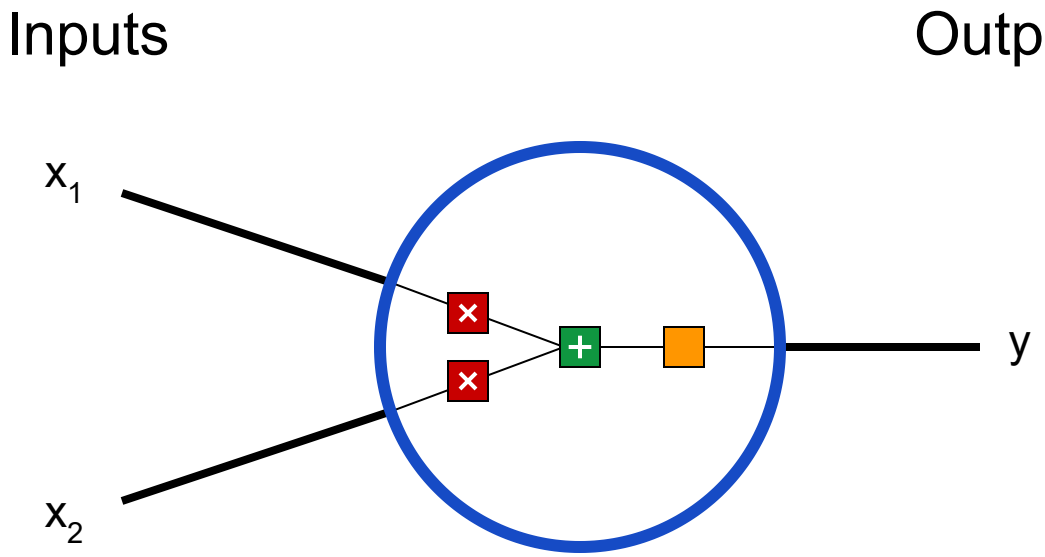
Here's something that might surprise you: **neural networks aren't that complicated!** The term "neural network" gets used as a buzzword a lot, but in reality they're often much simpler than people imagine.

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!

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:

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

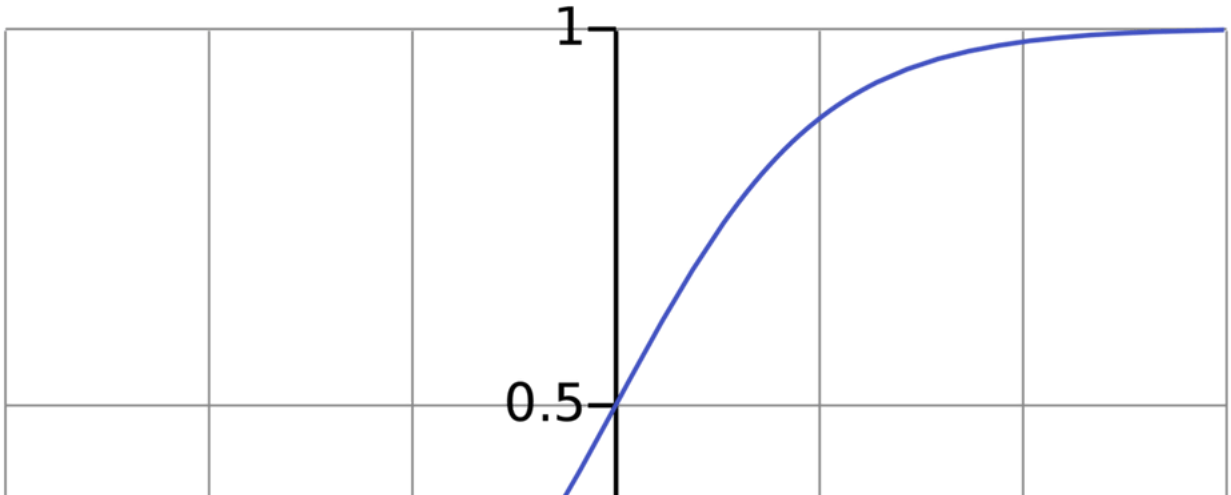
Next, all the weighted inputs are added together with a bias b:

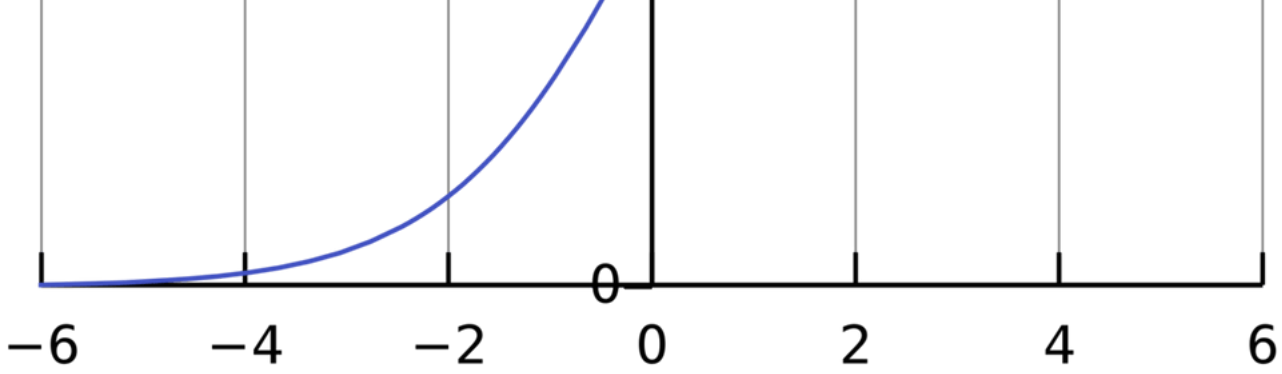
$$(x_1 * w_1) + (x_2 * w_2) + b$$

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 $(-\infty, +\infty)$ to $(0, 1)$ - big negative numbers become ~0, and big positive numbers become ~1.

A Simple Example

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 $w_1 = 0, w_2 = 1$ 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

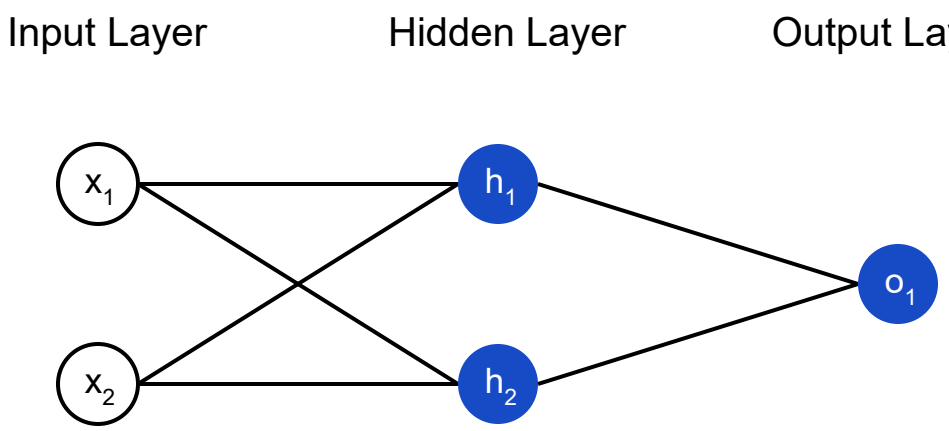
Time to implement a neuron! We'll use NumPy, a popular and powerful computing library for Python, to help us do math:

```
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 = 4
n = Neuron(weights, bias)

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

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 (h_1 and h_2), and an output layer with 1 neuron (o_1). Notice that the inputs for o_1 are the outputs from h_1 and h_2 - 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 h_1, h_2, o_1 denote the *outputs* of the neurons they represent.

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

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

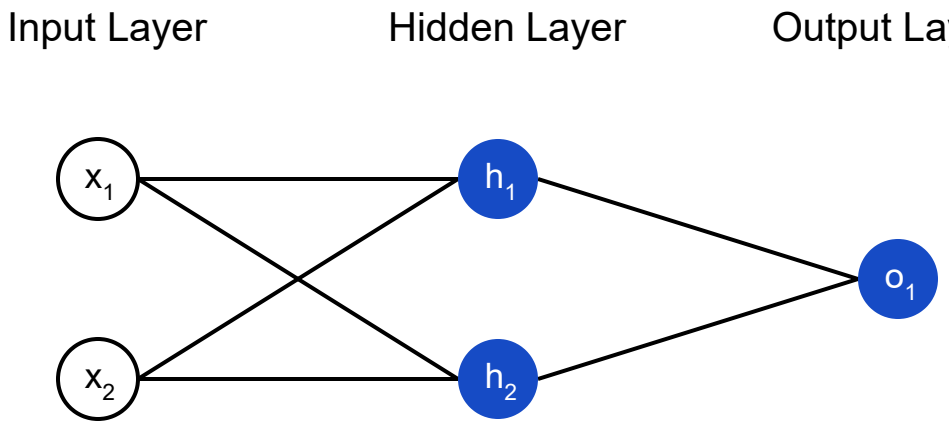
$$\begin{aligned} o_1 &= f(w \cdot [h_1, h_2] + b) \\ &= f((0 * h_1) + (1 * h_2) + 0) \\ &= f(0.9526) \\ &= \boxed{0.7216} \end{aligned}$$

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:




```
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
```

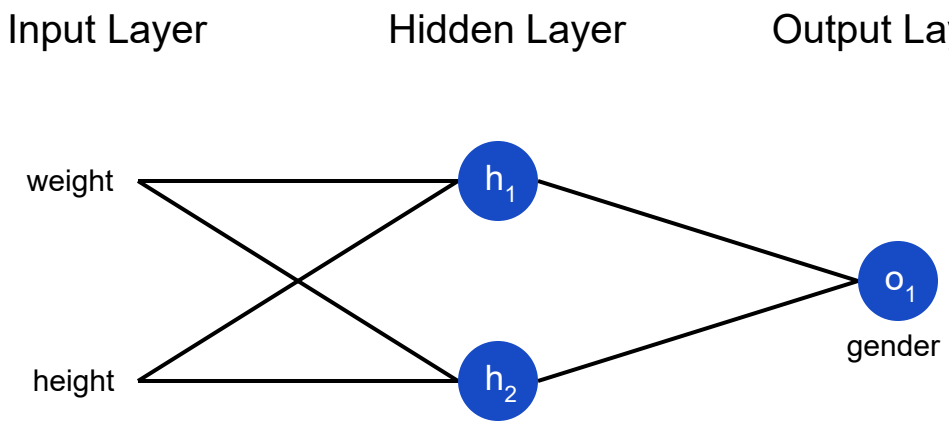
We got 0.7216 again! Looks like it works.

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

> *I arbitrarily chose the shift amounts (135 and 66) to make the numbers look nice. Normally, you'd shift by the mean.*

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 0 - 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

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
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
```

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

Nice. Onwards!

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.*

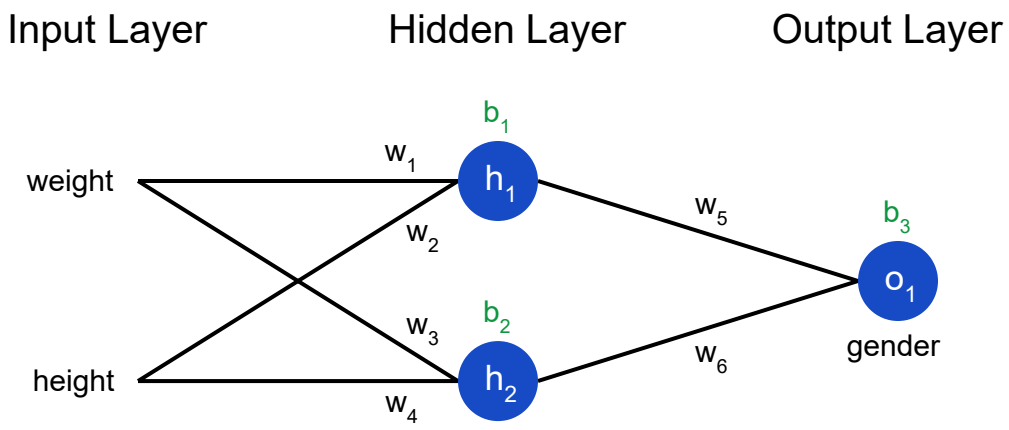
For simplicity, let's pretend we only have Alice in our dataset:

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 $\frac{\partial L}{\partial w_1}$ 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.*

To start, let's rewrite the partial derivative in terms of $\frac{\partial y_{pred}}{\partial w_1}$ 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 $\frac{\partial L}{\partial y_{pred}}$ 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 $\frac{\partial y_{pred}}{\partial w_1}$. Just like before, let h_1, h_2, o_1 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 w_1 only affects h_1 (not h_2), 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 $\frac{\partial h_1}{\partial w_1}$:

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

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

You guessed it, Chain Rule.

x_1 here is weight, and x_2 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 $\frac{\partial L}{\partial w_1}$ 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 $\frac{\partial L}{\partial w_1}$:

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

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

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

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

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

> *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 w_1, 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 $\eta \frac{\partial L}{\partial w_1}$ from w_1:

- If $\frac{\partial L}{\partial w_1}$ is positive, w_1 will decrease, which makes L decrease.
- If $\frac{\partial L}{\partial w_1}$ is negative, w_1 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:

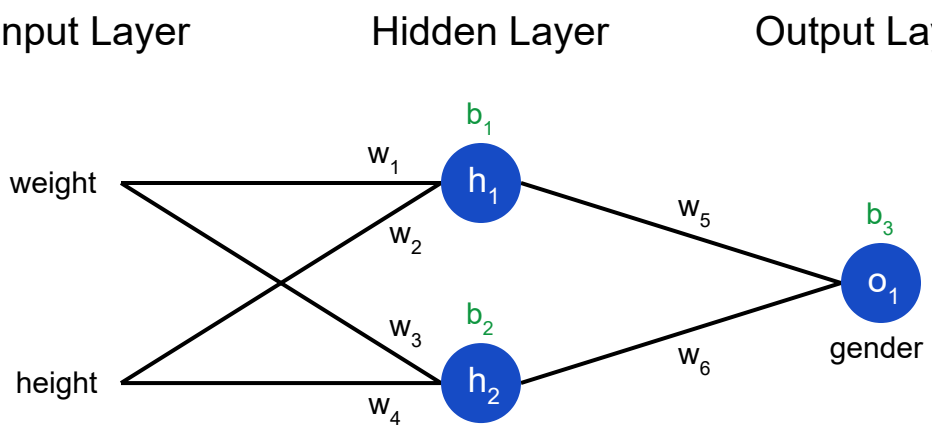
1. Choose **one** sample from our dataset. This is what makes it *stochastic* gradient descent - we only operate on one sample at a time.
2. Calculate all the partial derivatives of loss with respect to weights or biases (e.g. $\frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_2}$, etc).
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))
```

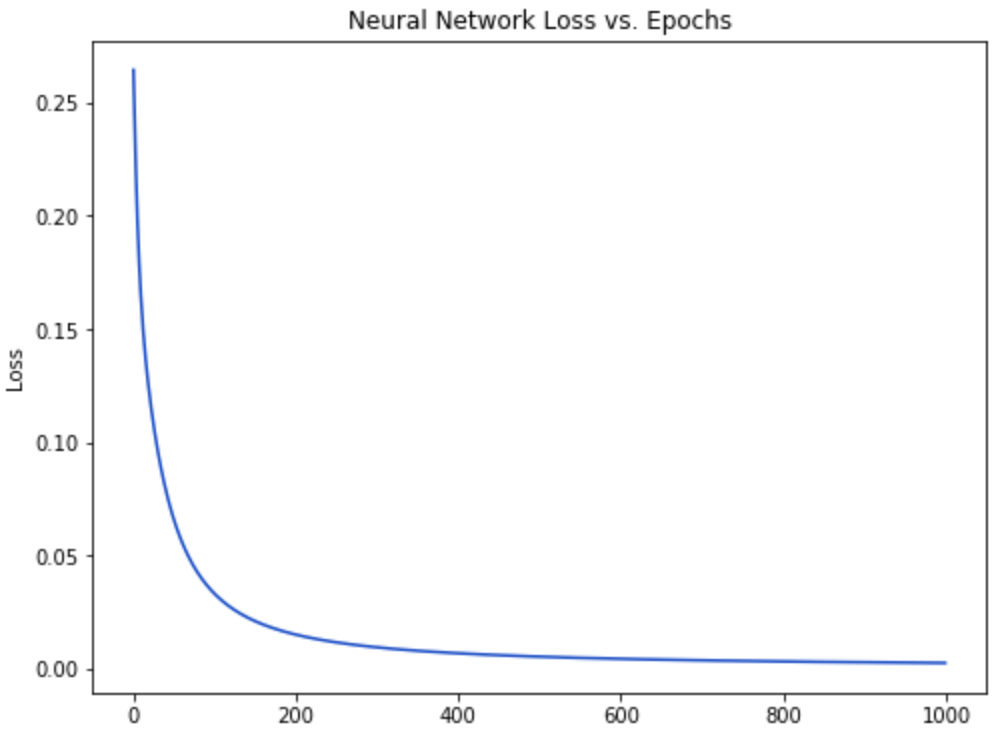
```
def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)

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.
      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
    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)
```

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

Our loss steadily decreases as the network learns:



Epochs

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
```

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 revolutionized the field of Computer Vision and can be extremely powerful.
- Read my introduction to Recurrent Neural Networks (RNNs), which are often used for Natural Language Processing (NLP).

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

Thanks for reading!

I write about ML, Web Dev, and more. **Subscribe to get new posts by email!**

example@domain.com

Send me *only* ML posts

SUBMIT

This site is protected by reCAPTCHA and the Google Privacy Policy and Terms of Service apply.

This blog is open-source on Github.

Tags: Machine Learning | Neural Networks | Python | For Beginners

YOU MIGHT ALSO LIKE

Neural Networks From Scratch
July 30, 2019
A 4-post series that provides a fundamentals-oriented approach towards understanding Neural Networks.

An Introduction to Recurrent Neural Networks for Beginners
July 24, 2019
A simple walkthrough of what RNNs are, how they work, and how to build one from scratch in Python.

Victor Zhou @victorczhou
SWE @ Facebook. CS '19 @ Princeton. I blog about web development, machine learning, and more topics.

SHARE THIS POST

Facebook Twitter LinkedIn Reddit

DISCUSS ON

Twitter Hacker News Reddit

Login

Add a comment

M↓ MARKDOWN

COMMENT ANONYMOUSLY ADD COMMENT

Upvotes Newest Oldest

K **Kory Mathewson**
2 points · 9 months ago

Great example! Nice work and a stellar write up. Curious why you chose to illustrate the basic example building blocks with MSE instead of cross-entropy loss?

> V **Victor Zhou**
> **1 point** · 9 months ago
>
> Thank you! I thought MSE was a bit simpler / less intimidating than cross-entropy.

> C **Cliparts**
> **0 points** · 2 months ago
>
> I always failed math courses when I was younger and I know that ML needs a lot of math but your explanation is understandable even by me. Thank you from Portland,OR.

D **Delphae**
1 point · 9 months ago

This tutorial explains very well the math behind a neural network and practical Python code. I have read a couple of times for a better understanding. It is very hard to explain this topic in a very clear way. You did a good job. Thanks.

A **aaron peacock**
1 point · 8 months ago

excellent post! I appreciate your breakdown of the buzzwords into concrete and bite-sized steps. Thank you!

Anonymous

1 point · 8 months ago

One of the best NN tutorial, big up Victor.

O **Océan Bao**
1 point · 8 months ago

Thank you for such clarity and grace! Please keep on the fantastic work and enlighten us!

? **Anonymous**
1 point · 6 months ago

I never comment on such websites, but this presentation of NN was so excellent that I just had to. Keep up the good work!

? **Anonymous**
1 point · 4 months ago

This is one of the best explanations of neural networks that I've seen so far. The examples were clearly explained and were of reasonable size so that the concepts could be easily understood without seeming overwhelming. The code was also very well written and easy to understand. Thank you for writing this tutorial, you have a great talent!

V **Victor Zhou** MODERATOR
0 points · 4 months ago

Thank you, glad it was helpful!

? **Anonymous**
1 point · 3 months ago

This tutorial was awesome! My maths isn't super strong so I did struggle to keep up with things a little after the mathematics went mental, but the code is good enough to follow along as a programmer.

For anybody interested, I took this tutorial and adapted it to TypeScipt (node.js) and got it running exactly the same with almost identical outputs. I also did away with numpy to make it easier to see what was going on for things like array subraction, array differencing and 'applyAlongAxis'.

https://github.com/megmut/basic-neural-net

V **Victor Zhou** MODERATOR
0 points · 3 months ago

Very cool! Thanks for sharing.

M **Mauro**
1 point · 3 months ago

I always failed math courses when I was younger and I know that ML needs a lot of math but your explanation is understandable even by me. Thank you from Italy.

J **John E Nowak**
1 point · 8 months ago

Thanks for this concise explanation. I'm an old fogey who, once upon a time, actually understood the math around all of this, but have gotten rusty while living a different life. Now, reading this, I cannot seem to make the leap from `y = f(7)` to `f(7) = .999` Why .999? Why not .873, .6, etc.? In other words, I do not understand how you determine what `f()` actually is. I am sure it is a simple thing, a given for you, but can you please indulge my ancient ignorance and help me cross this conceptual gap?

If I just postulate that `f()` is whatever you say it is, I believe I grasp the rest of this, but without that piece, how can I be sure of even that much understanding?

Thanks, John

V **Victor Zhou** MODERATOR
0 points · 8 months ago

Hey there, John!

`f()` is the **sigmoid** activation function, which is defined as f(x) = 1 / (1 + e^(-x)). Plugging in 7 for x gives us `f(7) = .999`.

D **Didgius**
0 points · 9 months ago

Hello Victor! Really great article. Can I ask you to create a video tutorials about the Neural Networks? I represent the education-ecosystem where users can upload their own tutorials

A **Arman Tavakoli**
0 points · 9 months ago

This is super clean - wish I understood the MV calc for the back-propagation part.

Actually I wish there was just a more visual way to explain it - awesome stuff!

A **Adz**
0 points · 9 months ago

https://www.amazon.co.uk/Gr...

this is the best book on understanding Deep Learning

V **vcavallo**
0 points · 9 months ago

Excellent post, thanks!
Totally minor, but I think you've got a small error in the bias comment in the "Coding a neuron" code block. (bias=0 vs bias=4)

V **Victor Zhou**
0 points · 9 months ago

nice catch, will fix!

K **kiran**
0 points · 9 months ago

Nice article. Crisp and clear. Just one question[Probably a stupid one], While calculating the MSE, shouldn't that be divided by 2?

I **Ian Wright**
0 points · 9 months ago

Awesome! Thank you! Minor correction in "Training: Stochastic Gradient Descent" section --> If {partial L wrt w1} is negative, w1 will increase, which makes L **increase**.

V **Victor Zhou**
0 points · 9 months ago

I actually don't think that's wrong - if (partial L / partial w1) is negative, w1 increasing makes L decrease because the derivative was negative. The point was that the SGD update should make the loss decrease regardless.

I **Ian Wright**
0 points · 9 months ago

Oops, I read it wrong - thanks. Great post!

P **Putu Alfred Crosby**
0 points · 9 months ago

Great and clear explained article!
I'd like to write my article about neural network too. Wondering how do you write the math in your article?

G **guru prasaad**
0 points · 9 months ago

i guess we are using linearRegression with Gradient Descent ? am i right ?

S **Shubham Kumar**
0 points · 9 months ago

Sorry if this is a really basic question. Why are we assigning weights (hard coding them) in the class rather than letting them be tunable. I thought the entire point for the network is to adjust biases and weights to fit the training data. Thanks!

K **Kiran Joshi**
0 points · 9 months ago

Great article - thanks!

There's a small typo in your calculation of the derivative of the sigmoid: the numerator should be exp(-x), instead of exp(x).

V **Victor Zhou**
0 points · 9 months ago

nice catch, thank you!

A **Artemiy**
0 points · 9 months ago

There's something I don't quite understand.
Is there a simple way to calculate all the partial derivatives in an automatic way with more complex (say, 3- or 4-level) neural networks?
Or will other optimizers fit such a task better?
Thank you.

> V **Victor Zhou**
> **0 points** · 9 months ago
>
> There's definitely better ways to calculate the partial derivatives - how it's actually done is by calculating them one layer at a time, then propagating those gradients back to the previous layer (thus the name backpropagation). Weights are normally stored in matrices because matrix operations make this much easier. There are lots of state-of-the-art research papers that use SGD in networks with 20+ layers.

M **Mahmoud Elkafafy**
0 points · 9 months ago

Hello,

In section 3/Part 1, when you shift the data , for instance, you subtract the weight from 135. What does 135 represent? It is neither the mean nor the median.

Thank you in advance!

Mahmoud

> V **Victor Zhou**
> **0 points** · 9 months ago
>
> The 135 is just an arbitrary number here - I wanted to keep the numbers nice looking for simplicity. Usually you'd subtract the mean.

S **Sriram**
0 points · 9 months ago

Part 1 was very easy to understand. It took couple of readings to understand part 2. Executed python program and got the desired results. Thanks.

G **Grisselle Diaz Perez**
0 points · 9 months ago

Simple... Victor keep this topic going to more complex levels...I loved it.

T **Trust Patches**
0 points · 9 months ago

Awesome example, just a little confused about one thing. In the first part of the tutorial you showed us how to build a Neuron, and even created a network using them, but once you started talking about training you stopped using them in your network as far as I can tell. Could you explain that? Thanks!

V **Victor Zhou**
0 points · 9 months ago

They're still in the network, just less explicit. The network stores each weight and bias for all the neurons - for example, neuron h1 is stored via weights w1, w2 and bias b1. If you go through all 3 neurons (h1, h2, o1) you'll realize that each weight and bias needed for that neuron is indeed stored in the network!

? **Anonymous**
0 points · 8 months ago

Hi Victor,

Thanks for your post and it is simple and clear to understand.

I have below query which I'm unable to understand. Please help me.

In your post, you have mentioned 2 cases after deriving partial derivative for Loss(L) w.r.t (w1) = 0.0214. This means for a unit increase in w1, L is increased by 0.0214.

By SGD formula: w1 <- w1 - n(above derivative)

2 Cases: 1. If (above derivative) is positive, w1 will decrease and L will also decrease

1. If (above derivative) is negative, w1 will increase and L will also INCREASE., right?
   [**But, you have mentioned it as "decrease" I did not understand this 2nd case. Could you please help me to clarify this point?**]

Thanks for your post and it is simple and clear to understand.

Thanks, Shibhi

V **Victor Zhou** MODERATOR
0 points · 8 months ago

Hey, thanks for the question.

- If the derivative dL/dw1 is positive (e.g. 0.0214), that means increasing w1 increases L. Thus, decreasing w1 decreases L.
- If the derivative dL/dw1 is negative (e.g. -0.0214), then increasing w1 instead **decreases** L. In your words, "for a unit increase in w1, L is increased by -0.0214 (i.e. L is decreased by 0.0214".

Hope that clears things up.

? **Anonymous**
0 points · 8 months ago

Is shifting your data normalizing it?

V **Victor Zhou** MODERATOR
0 points · 8 months ago

Yup - usually you'd shift by the mean and then scale so the standard deviation is 1, but I just kept it simple for this post.

F **fan y**
0 points · 8 months ago

it's helpful.

? **Anonymous**
0 points · 6 months ago

You did a good job. Merci

? **Anonymous**
0 points · 8 months ago

good post! thanks, Victor

? **Anonymous**
0 points · 7 months ago

You have taken weight reference as 135 pounds and Height reference as 66 inches. Instead of taking the shifted amount in the dataset as [-2,5] etc, when I am taking a general value like [135,66], then Why is it giving me output as 0.05 and 0.05 always?

K **Karan Jadhav**
0 points · 6 months ago

Hi Victor Thank you for the tutorial very well explained :) , I am new to ML just had one doubt ,the above procedure for analysis can be used for Unsupervised Learning or is it only for Supervised Learning?. As we are predicting the output and calculating the loss based on true output.

? **Anonymous**
0 points · 6 months ago

hi, thank you for this awesome tutorial, I try to use this code for face detection but I can do that any help please ?

A **ANKIT YADAV**
0 points · 6 months ago

Awesome article among all of them I read .

? **Anonymous**
0 points · 5 months ago

How did you arrived to the below equation? can you clarify. It it a power rule?

--- Calculate partial derivatives.

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

F **Fabio Vitaliano**
0 points · 5 months ago

this is the best explanation about the math for back propagation ann i have read congratulations and thank you

? **Anonymous**
0 points · 5 months ago

Honestly an amazing introduction (as I was able to do it and I am a beginner). I did have a few doubts about the final implementation. I am not sure now NNs are coded normally but does it have variable numbers of hidden neurons and input variables? I tried coding this modified version(currently trying to calculate the derivatives and change weights and biases in a single loop). I would love pointers on where I can find examples of "optimal" NN code.

? **Anonymous**
0 points · 4 months ago

Thank you for the tutorial. I ported the final code to typescript:
https://gist.github.com/drets/db10b66bb30c89c48e882cb75e60c77d

? **Anonymous**
0 points · 4 months ago

How does the math change when you have more than 1 output

Specifically as it relates to the following code lines

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

If you had say two outputs instead of one would you just do the previous twice then take an average of the values?

V **Victor Zhou** MODERATOR
0 points · 4 months ago

You'd just sum the values.

? **Anonymous**
0 points · 4 months ago

Can you explain what this actually is a bit more in depth?

d*L*d*ypred* = -2 * (*y*true - y_pred)

I think I understand what you are doing - but the naming conventions may be throwing me off. Would really help if you explained it in plain English.

Thanks for the great work - really informative article BTW!!

V **Victor Zhou** MODERATOR
0 points · 4 months ago

We know `L = (y_true - y_pred)^2`, so the derivative of L with respect to `y_pred` is `-2 * (y_true - y_pred)` using Chain Rule.

Anonymous
0 points · 4 months ago

Can anyone please explain how the following result arrived: ∂(1−ypred)2/∂ypred = −2(1−ypred)

Anonymous
0 points · 4 months ago

Thanks a lot for this article, I have finally understood principle behind neural network!

Anonymous
0 points · 4 months ago

In the part where you are computing the partial derivative "Example: Calculating the Partial Derivative" f1 = f(w1x1 + w2x2+b1) = f(-2+-1+0) = f(-3) = 0.0474. if f(x) is the sigmoid function than f(-3) = 0.952 and not 0.0474.

can you please explain this value? Thank you

Anonymous
0 points · 3 months ago

In the implementation , the part where data is trained is confusing; in the comment, you say that we are looping on the entire dataset, but the zip function for x, y*true in zip(data, allytrues): will always return the same tuple ([-2,-1]"Alice' data",[1,0]"all*y_true")

Therefore, I think the training is iterating 1000 times over Alice's data only.

Anonymous
0 points · 3 months ago

Thanks for the great post. I have a question regarding the code below for scaler operations. `python for x, y_true in zip(data, all_y_trues)`
I was curious what to do if I want to use the vector operation?

Anonymous
0 points · 3 months ago

Hello, I have a question. When instead of these shifted weight and height values I enter (in training examples) non-shifted values (for example 133 and 65 for Alice) I always get loss at 0.125 and gender at 0.500 what is the reason of that?

Anonymous
0 points · 2 months ago

非常有幸拜读到这篇文章，使我对神经网络有了更清楚的认识！

Anonymous
0 points · 2 months ago

hi ! i really enjoyed to read the article and it was an introcution for me. i want to know if you think about writing a complete book on AI ?.

> V **Victor Zhou** MODERATOR
> **0 points** · 2 months ago
>
> Glad you found it helpful. Not for now, but if you want to stay updated on future ML posts you can subscribe to my newsletter: https://victorzhou.com/subscribe/

t **trinhngoc quan**
0 points · 2 months ago

Hi, with the cost function, why not get the absolute value instead ?

M **Mohammed Qureshi**
0 points · 2 months ago

Came here from your Medium article. This is undoubtedly the best explanation I've seen of neural networks. Thanks a ton, Victor!

 **Godfrain Jacques**
0 points · 1 months ago

Hi there Victor! Nice tutorial. Do you mind if I do a c++ version?

> V **Victor Zhou** MODERATOR
> **0 points** · 1 months ago
>
> Hey there - should be fine, send me an email (vzhou842@gmail.com) or DM me on twitter (@victorczhou) to discuss more.
>
> >  **Godfrain Jacques**
> > **0 points** · 24 days ago
> >
> > awesome !

 **Yang Yang**
0 points · 2 months ago

Hey Victor, thank you for such a good post! I have an irrelevant but ML-related question, I'm wondering if you could help. I need to predict a bunch of variables from one single input. I have one single input and about 20 output. To make the problem simpler, I gave a simplified example:, one's hourly salary is determined by his total salary divided by hours of work. Given one's salary, predict his total salary and hours of work. Could you please help me what kind of ml algorithm I could be using to solve this problem? Could you just throw a huge number of data to neural network and solve this problem?

thank you very much in advance for your help! Yang

M **Memo son**
0 points · 1 months ago

my man.

Anonymous
0 points · 1 months ago

Thank you for this article. I tried to model this example using keras:

`import numpy as np import matplotlib.pyplot as pyplot from keras.models import Sequential from keras.layers import Dense from keras.utils import to_categorical

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])

model = Sequential([Dense(2, activation='sigmoid', input_shape=(2,)), Dense(1, activation='sigmoid'),])

model.compile(optimizer='sgd', loss='mean*squared*error', metrics=['accuracy'],)

history = model.fit(data, all*y*trues, epochs=1000, batch_size=1, verbose=0)

Plot metrics

pyplot.plot(history.history['accuracy']) pyplot.title('Model accuracy') pyplot.xlabel('Epochs') pyplot.ylabel('Accuracy')

pyplot.figure() pyplot.plot(history.history['loss']) pyplot.title('Model loss') pyplot.xlabel('Epochs') pyplot.ylabel('Loss') pyplot.show()`

But the accuracy is stepping from 0.5 to 1 in about 200 epochs. What am I doing wrong?

Syarifah kemala Putri
0 points · 1 months ago

hei victorzhou, nice example! but i want to ask about something that i dont understand. In your explaination, when all weight = 1 and dL/dw1 = 0.0214 , you say "if we were to increase w1, L would increase a tiiiny bit as a result". So that mean we have to increase all weight to 2 ?. Thank you and have good day:)

Anonymous
0 points · 15 days ago

Great explanation how to make a simple neural network in Python! Thank you, Victor!

Shafie Mukhre
0 points · 10 days ago

Great article! Thank you Victor! I think on under the subtopic - Training: Stochastic Gradient Descent, you meant If $\partial L/\partial w$ is negative, w1 will increase, which makes L increase (instead of decrease).

Victor Zhou MODERATOR

0 points · 10 days ago

Hey Shafie - actually, that's intentional. ∂L/∂w being negative literally means "L changes negatively when w changes positively" so an increase in w1 would make L decrease for that case. The main takeaway there is that no matter what, we should make L decrease!

Valeria Yura

0 points · 7 days ago

Thank you so much, you've helped me to understand Neural networks for my machine learning project.

Powered by **Commento**