4. Training the Network

Forward propagation is simply the summation of the previous layer's output multiplied by the weight of each wire, while back-propagation works by computing the partial derivatives of the cost function with respect to **every** weight or bias in the network. In back propagation, the network gets better at minimizing the error and predicting the output of the data being used for training by incrementally updating their weights and biases using stochastic gradient descent.

We are trying to estimate a continuous-valued function, thus we will use squared loss as our cost function and an identity function as the output activation function. $f(x)$ is the activation function that is called on the input to our final layer output node, and $\hat{a}$ is the predicted value, while y is the actual value of the input.

$$C = \frac{1}{2}(y - \hat{a})^2 \tag{5.1}$$

$$f(x) = x \tag{5.2}$$

When you're done implementing the function `train` (below and in your local repository), run the script and see if the errors are decreasing. If your errors are all under 0.15 after the last training iteration then you have implemented the neural network training correctly.

You'll notice that the `train` functin inherits from `NeuralNetworkBase` in the codebox below; this is done for grading purposes. In your local code, you implement the function directly in your Neural Network class all in one file. The rest of the code in `NeuralNetworkBase` is the same as in the original `NeuralNetwork` class you have locally.

In this problem, you will see the network weights are initialized to 1. This is a bad setting in practice, but we do so for simplicity and grading here.

You will be working in the file `part2-nn/neural_nets.py` in this problem

Implementing Train

5/5 points (graded)

Available Functions: You have access to the NumPy python library as `np` , `rectified_linear_unit` , `output_layer_activation` , `rectified_linear_unit_derivative` , and `output_layer_activation_derivative`

Note: Functions `rectified_linear_unit_derivative` , and `output_layer_activation_derivative` can only handle scalar input. You will need to use `np.vectorize` to use them

```
class NeuralNetwork(NeuralNetworkBase):

    def train(self, x1, x2, y):

        ### Forward propagation ###
        input_values = np.matrix([[x1],[x2]]) # 2 by 1

        # Calculate the input and activation of the hidden layer
        hidden_layer_weighted_input = self.input_to_hidden_weights @ input_values + self.biases # TODO (3 by 1 matrix)
        hidden_layer_activation = np.zeros(hidden_layer_weighted_input.shape) #hidden_layer_weighted_input.copy()
        for i in range(len(hidden_layer_activation)):
            hidden_layer_activation[i] = rectified_linear_unit(hidden_layer_weighted_input[i].item()) # TODO (3 by 1 matrix)

        output =  self.hidden_to_output_weights @ hidden_layer_activation # TODO
        activated_output = output_layer_activation(output) # TODO
```

Press ESC then TAB or click outside of the code editor to exit

Correct

Test results

Hide output

CORRECT

Test: train random linear combination

Testing random datapoints to emulate function x1 + x2

Output:

```
Training pairs:  [((3, -9), -6), ((-10, 5), -5), ((6, 8), 14), ((7, -8), -1), ((0, 8), 8), ((1, 9), 10), ((-7,
-10), -17), ((9, -6), 3), ((6, 8), 14), ((-3, -8), -11)]
Starting params:

(Input --> Hidden Layer) Weights:  [[1. 1.]
 [1. 1.]
 [1. 1.]]
(Hidden --> Output Layer) Weights:  [[1. 1. 1.]]
Biases:  [[0.]
 [0.]
 [0.]]

Epoch  0
(Input --> Hidden Layer) Weights:  [[0.68912517 0.62494639]
 [0.68912517 0.62494639]
 [0.68912517 0.62494639]]
(Hidden --> Output Layer) Weights:  [[0.42226674 0.42226674 0.42226674]]
Biases:  [[-0.04736219]
 [-0.04736219]
 [-0.04736219]]
Epoch  1
(Input --> Hidden Layer) Weights:  [[0.70076089 0.65966499]
 [0.70076089 0.65966499]
 [0.70076089 0.65966499]]
(Hidden --> Output Layer) Weights:  [[0.4903666 0.4903666 0.4903666]]
Biases:  [[-0.04383336]
 [-0.04383336]
 [-0.04383336]]
Epoch  2
(Input --> Hidden Layer) Weights:  [[0.69740075 0.66582162]
 [0.69740075 0.66582162]
 [0.69740075 0.66582162]]
(Hidden --> Output Layer) Weights:  [[0.49387628 0.49387628 0.49387628]]
Biases:  [[-0.04380186]
 [-0.04380186]
 [-0.04380186]]
Epoch  3
(Input --> Hidden Layer) Weights:  [[0.69398941 0.66917519]
 [0.69398941 0.66917519]
 [0.69398941 0.66917519]]
(Hidden --> Output Layer) Weights:  [[0.49361143 0.49361143 0.49361143]]
Biases:  [[-0.04394368]
 [-0.04394368]
 [-0.04394368]]
Epoch  4
(Input --> Hidden Layer) Weights:  [[0.69132551 0.671671  ]
 [0.69132551 0.671671  ]
 [0.69132551 0.671671  ]]
(Hidden --> Output Layer) Weights:  [[0.49327095 0.49327095 0.49327095]]
Biases:  [[-0.04405298]
```

```
 [-0.04405298]
 [-0.04405298]]
Epoch  5
(Input --> Hidden Layer) Weights:  [[0.68927151 0.67357832]
 [0.68927151 0.67357832]
 [0.68927151 0.67357832]]
(Hidden --> Output Layer) Weights:  [[0.49300363 0.49300363 0.49300363]]
Biases:  [[-0.0441292]
 [-0.0441292]
 [-0.0441292]]
Epoch  6
(Input --> Hidden Layer) Weights:  [[0.68768759 0.67504159]
 [0.68768759 0.67504159]
 [0.68768759 0.67504159]]
(Hidden --> Output Layer) Weights:  [[0.49279779 0.49279779 0.49279779]]
Biases:  [[-0.04417942]
 [-0.04417942]
 [-0.04417942]]
Epoch  7
(Input --> Hidden Layer) Weights:  [[0.68646535 0.67616644]
 [0.68646535 0.67616644]
 [0.68646535 0.67616644]]
(Hidden --> Output Layer) Weights:  [[0.49263906 0.49263906 0.49263906]]
Biases:  [[-0.04420948]
 [-0.04420948]
 [-0.04420948]]
Epoch  8
(Input --> Hidden Layer) Weights:  [[0.68552167 0.67703244]
 [0.68552167 0.67703244]
 [0.68552167 0.67703244]]
(Hidden --> Output Layer) Weights:  [[0.49251635 0.49251635 0.49251635]]
Biases:  [[-0.04422394]
 [-0.04422394]
 [-0.04422394]]
Epoch  9
(Input --> Hidden Layer) Weights:  [[0.68479273 0.67769995]
 [0.68479273 0.67769995]
 [0.68479273 0.67769995]]
(Hidden --> Output Layer) Weights:  [[0.49242123 0.49242123 0.49242123]]
Biases:  [[-0.04422633]
 [-0.04422633]
 [-0.04422633]]
```

Test: training run

Testing datapoints from local tests

Output:

```
Training pairs:  [((2, 1), 10), ((3, 3), 21), ((4, 5), 32), ((6, 6), 42)]
Starting params:

(Input --> Hidden Layer) Weights:  [[1. 1.]
 [1. 1.]
 [1. 1.]]
(Hidden --> Output Layer) Weights:  [[1. 1. 1.]]
Biases:  [[0.]
 [0.]
 [0.]]

Epoch  0
(Input --> Hidden Layer) Weights:  [[1.002 1.001]
 [1.002 1.001]
 [1.002 1.001]]
(Hidden --> Output Layer) Weights:  [[1.003 1.003 1.003]]
Biases:  [[0.001]
 [0.001]
 [0.001]]
(Input --> Hidden Layer) Weights:  [[1.01077397 1.00977397]
 [1.01077397 1.00977397]
 [1.01077397 1.00977397]]
(Hidden --> Output Layer) Weights:  [[1.02052462 1.02052462 1.02052462]]
Biases:  [[0.00392466]
 [0.00392466]
 [0.00392466]]
(Input --> Hidden Layer) Weights:  [[1.02772391 1.03096139]
 [1.02772391 1.03096139]
 [1.02772391 1.03096139]]
(Hidden --> Output Layer) Weights:  [[1.05829312 1.05829312 1.05829312]]
Biases:  [[0.00816214]
 [0.00816214]
 [0.00816214]]
(Input --> Hidden Layer) Weights:  [[1.04523414 1.04847162]
 [1.04523414 1.04847162]
 [1.04523414 1.04847162]]
(Hidden --> Output Layer) Weights:  [[1.09237808 1.09237808 1.09237808]]
Biases:  [[0.01108051]
 [0.01108051]
 [0.01108051]]
```

Hide output

Submit You have used 4 of 20 attempts

✔ Correct (5/5 points)

Discussion

Topic: Unit 3 Neural networks (2.5 weeks):Project 3: Digit recognition (Part 2) / 4. Training the Network

Hide Discussion

Add a Post

All Posts

[Staff] Stuck with backpropagation part

question posted about 21 hours ago by **disguiser**

I'm doing something wrong, but don't understand what. So we have

$$C = \frac{1}{2}(y - a_2)^2$$

$$\frac{\partial C}{\partial a_2} = (a_2 - y)$$

Then

$$a_2 = z_2 = W_2 a_1$$

$$\frac{\partial a_2}{\partial z_2} = 1$$

$$\frac{\partial z_2}{\partial W_2} = a_1$$

$$\frac{\partial C}{\partial W_2} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_2}$$

Then

$$a_1 = ReLU\,(z_1)$$

$$\frac{\partial a_1}{\partial z_1} = rectified\_linear\_unit\_derivative\,(z_1)$$

Then

$$z_1 = W_1 x + b$$

$$\frac{\partial z_1}{\partial W_1} = x$$

$$\frac{\partial z_1}{\partial b} = 1$$

Then

$$\frac{\partial C}{\partial W_1} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial b}$$

And finally

$$biases- = \frac{\partial C}{\partial b}$$

$$input\_to\_hidden\_weights- = \frac{\partial C}{\partial W_1}$$

$$hidden\_to\_output\_weights- = \frac{\partial C}{\partial W_2}$$

This network does not converge and loss only grows. Please take a look and provide any hints what may be wrong here. If I revealed too much, please advise how can I get help not compromising solution.

This post is visible to everyone.

Add a Response

2 responses

disguiser
about 20 hours ago

I assume that the problem is in those chain multiplications:

$$\frac{\partial C}{\partial W_1} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial W_1}$$

Some partial derivatives are numbers, some are 1x1 matrices, some are vectors of different dimentions. Should I use matrix multiplication there? Is so, order of multiplication must matter, and some matrices probably need to be transposed. I'm confused...

Stuck on the same point as well. Got the program all written up, calculated all the partial derivatives on paper and yet I cannot get matrices to multiply like they should. It's so annoying :P

posted about 2 hours ago by **Sarcus**

Add a comment

Mark_B2 (Community TA)
about 10 hours ago

You should use memberwise multiplication *and* matrix multiplication. To get sense of which, why and when, write down partial derivatives and only then convert them in the matrix form.
I mean write $\frac{\partial L}{\partial w_{21}}$, $\frac{\partial L}{\partial w_{22}}$ and so on. You'll see a pattern.

Why is a_2 = softmax(z2)? According to the instructions we are supposed to use f(x) = x as the output activation function.

posted about 9 hours ago by **hatalo**

My fault. I'll fix it.

posted about 8 hours ago by **Mark_B2** (Community TA)

On the second thought, if you were instructed to use activation function and gradient activation function, I think to be on the safe side one should use them. Thus it's much safer to assume `a2=output_activation(z2)` and `dz2/da2 = output_activation_gradient(z2)`.

posted about 5 hours ago by **Mark_B2** (Community TA)

Thank you for response, I'll try to do what you've suggested.

posted about 4 hours ago by **disguiser**

Add a comment

sandipan_dey
9 minutes ago

The following equations (and the similar ones) should be sufficient, struggled a lot with completely non-vectorized code to obtain the correct answer finally.

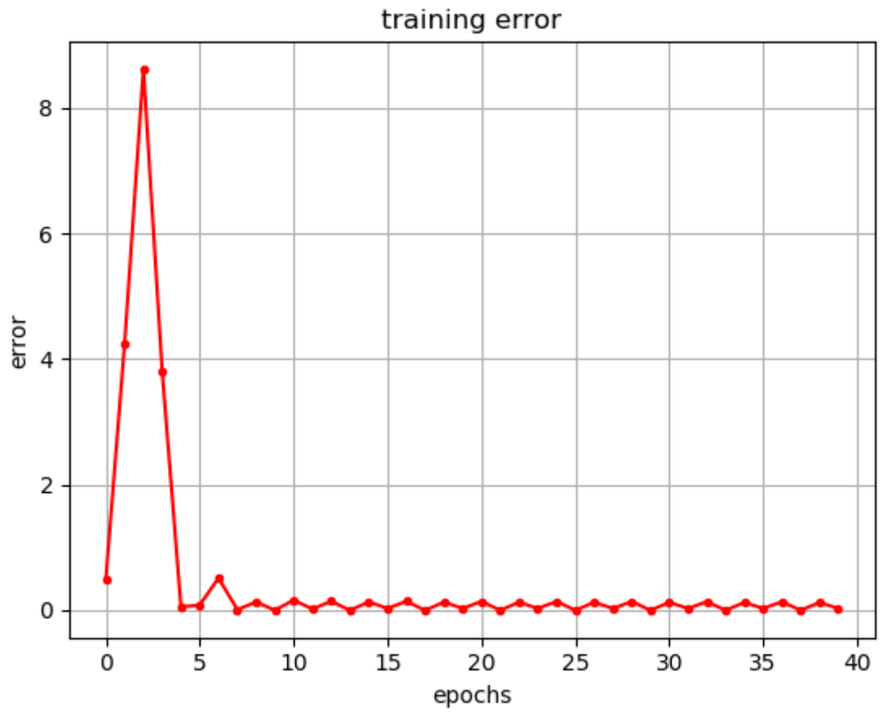
$\frac{\partial C}{\partial a_1} = (o_1 - y) \,.\, f'(u_1) \,.\, V_{11}$

$\frac{\partial C}{\partial b} = (o_1 - y) \,.\, f'(u_1)$

$$\frac{\partial C}{\partial V_{11}} = (o_1 - y) \,.\, f'(u_1) \,.\, a_1$$

$$\frac{\partial C}{\partial W_{11}} = \frac{\partial C}{\partial a_1} \,.\, \frac{\partial a_1}{\partial z_1} \,.\, x_1$$

The following figure shows how the training error converges.



Add a comment

Preview

Submit

Showing all responses

Add a response:

Preview

Submit

Learn About Verified Certificates