Optimizing a neural network with backward propagation

Puteaux, Fall/Winter 2020-2021

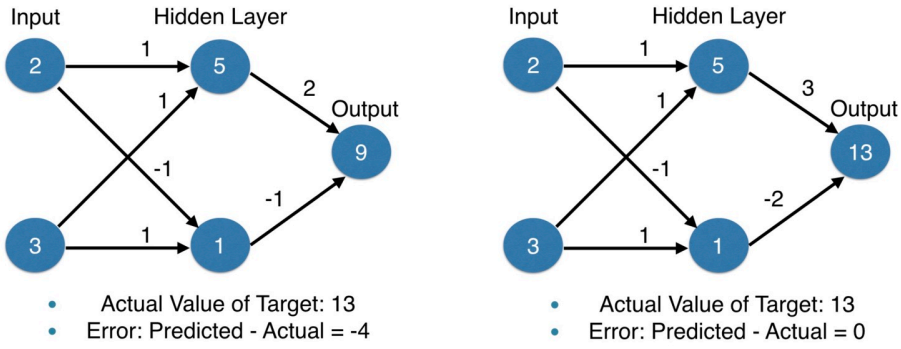
```
################################
##                            ##
##  Deep Learning in Python   ##
##                            ##
################################
```

§1 Introduction to Deep Learning in Python

§1.2 Optimizing a neural network with backward propagation

1 The need for optimization

1.1 How to measure the baseline for the neural network?



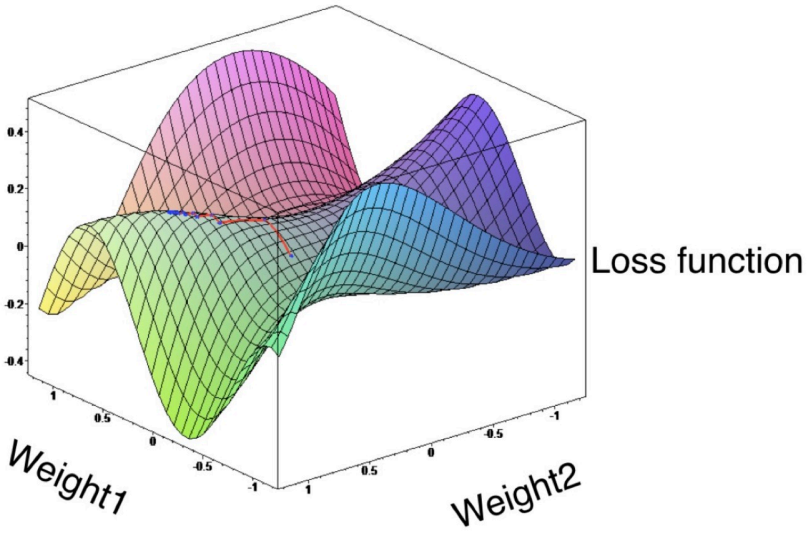
1.2 What are the challenges for the predictions with multiple points?

- Making accurate predictions gets more challenging with more points.
- At any set of weights, there are many values of the error corresponding to the many points for making predictions.

1.3 What is the importance of the loss function?

- Aggregates errors in predictions from many data points into a single number for measuring the model's predictive performance.
- A lower loss function value means a better model.

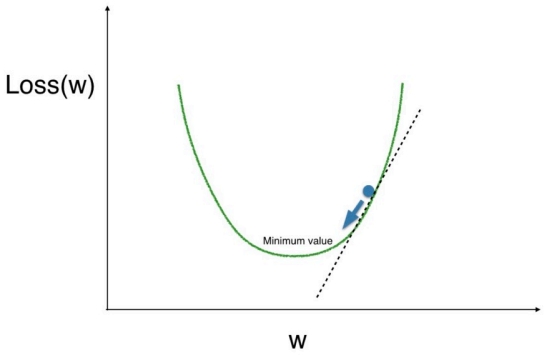
- The loss function's goal is to find the weights that give the lowest value for the loss function by gradient descent.



1.4 What are the steps of gradient descent?

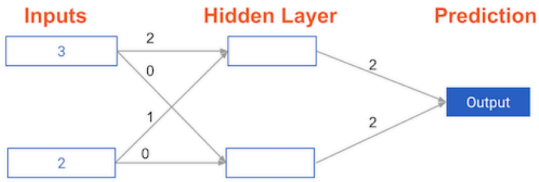
Start at a random point until got somewhere flat, find the slope, take a step downhill.

1.5 How to optimize a model with a single weight?



1.6 Practice question for calculating model errors:

- Continue working with the network to predict transactions for a bank.
- What is the error ($error = predicted - actual$) for the following network using the ReLU activation function when the input data is $[3, 2]$, and the actual value of the target, which tries to predict, is 5? It may be helpful to get out a pen and piece of paper to calculate these values.



☐ 5.

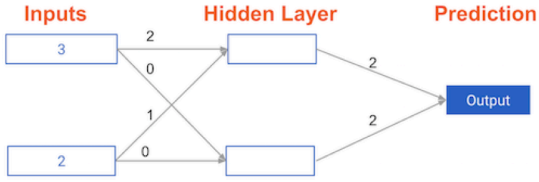
☐ 6.

☒ 11.

☐ 16.

1.7 Practice question for understanding how weights change model accuracy:

- Imagine making a prediction for a single data point. The actual value of the target is 7. The weight going from `node_0` to the output is 2, as shown below. If it is increased slightly, changing it to 2.01, would the predictions become more accurate, less accurate, or stay the same?



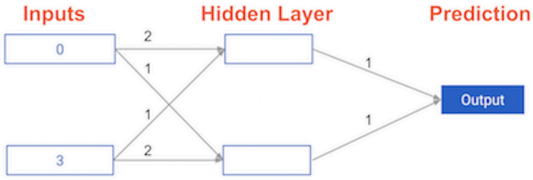
☐ More accurate.

☒ Less accurate.

☐ Stay the same.

1.8 Practice exercises for the need for optimization:

▶ Diagram of the forward propagation:



▶ Package pre-loading:

[1]:
```
import numpy as np
```

▶ Code pre-loading:

[2]:
```
def predict_with_network(input_data_point, weights):
    node_0_input = (input_data_point * weights['node_0']).sum()
    node_0_output = relu(node_0_input)

    node_1_input = (input_data_point * weights['node_1']).sum()
    node_1_output = relu(node_1_input)

    hidden_layer_values = np.array([node_0_output, node_1_output])
    input_to_final_layer = (hidden_layer_values * weights['output']).sum()
    model_output = relu(input_to_final_layer)

    return (model_output)


def relu(my_input):
    return (max(0, my_input))
```

▶ **Weight changes affect accuracy practice:**

[3]:
```
# The data point you will make a prediction for
input_data = np.array([0, 3])

# Sample weights
weights_0 = {'node_0': [2, 1], 'node_1': [1, 2], 'output': [1, 1]}

# The actual target value, used to calculate the error
target_actual = 3

# Make prediction using original weights
model_output_0 = predict_with_network(input_data, weights_0)

# Calculate error: error_0
error_0 = model_output_0 - target_actual

# Create weights that cause the network to make perfect prediction (3):␣
 ↪weights_1
weights_1 = {'node_0': [2, 1], 'node_1': [1, 0], 'output': [1, 1]}

# Make prediction using new weights: model_output_1
model_output_1 = predict_with_network(input_data, weights_1)

# Calculate error: error_1
error_1 = model_output_1 - target_actual

# Print error_0 and error_1
print(error_0)
print(error_1)
```

```
6
0
```

▶ Data pre-loading:

[4]:
```
input_data = [
    np.array([0, 3]),
    np.array([1, 2]),
    np.array([-1, -2]),
    np.array([4, 0])
]

weights_0 = {
    'node_0': np.array([2, 1]),
    'node_1': np.array([1, 2]),
    'output': np.array([1, 1])
}

weights_1 = {
    'node_0': np.array([2, 1]),
    'node_1': np.array([1., 1.5]),
    'output': np.array([1., 1.5])
}

target_actuals = [1, 3, 5, 7]
```

▶ Scaling up to multiple data points practice:

[5]:
```
from sklearn.metrics import mean_squared_error

# Create model_output_0
model_output_0 = []
# Create model_output_1
model_output_1 = []

# Loop over input_data
for row in input_data:
    # Append prediction to model_output_0
    model_output_0.append(predict_with_network(row, weights_0))

    # Append prediction to model_output_1
    model_output_1.append(predict_with_network(row, weights_1))

# Calculate the mean squared error for model_output_0: mse_0
mse_0 = mean_squared_error(target_actuals, model_output_0)

# Calculate the mean squared error for model_output_1: mse_1
mse_1 = mean_squared_error(target_actuals, model_output_1)
```

```
# Print mse_0 and mse_1
print("Mean squared error with weights_0: %f" % mse_0)
print("Mean squared error with weights_1: %f" % mse_1)
```

```
Mean squared error with weights_0: 37.500000
Mean squared error with weights_1: 49.890625
```

2 Gradient descent

2.1 How does gradient descent work?

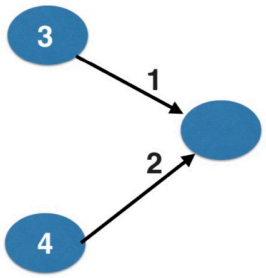
- If the slope is positive:
  - going opposite the slope means moving to lower numbers
  - subtract the slope from the current value
  - too big a step might lead astray
- The solution to avoid leading astray is using the learning rate:
  - update each weight by subtracting $learning\ rate \times slope$

2.2 What are the steps of the slope calculation?

- To calculate the slope for a weight, need to multiply:
  - the slope of the loss function with respect to value at the node which fed into
    - e.g., *the slope of the mean-squared loss function*, in this case,

      $2 \cdot (Predicted\ Value - Actual\ Value) = 2 \cdot Error$
  - the value of the node that feeds into the weight
  - the slope of the activation function with respect to the value fed into

2.3 Code of the slope calculation and weights updating:



[6]:
```
import numpy as np

weights = np.array([1, 2])
```

```
input_data = np.array([3, 4])
target = 6
learning_rate = 0.01

preds = (weights * input_data).sum()
error = preds - target
print(error)
```

```
5
```

[7]:
```
gradient = 2 * input_data * error
gradient
```

[7]:
```
array([30, 40])
```

[8]:
```
weights_updated = weights - learning_rate * gradient

preds_updated = (weights_updated * input_data).sum()
error_updated = preds_updated - target
print(error_updated)
```

```
2.5
```

2.4 Practice exercises for gradient descent:

▶ **Package pre-loading:**

[9]:
```
import numpy as np
```

▶ **Data pre-loading:**

[10]:
```
input_data = np.array([1, 2, 3])

weights = np.array([0, 2, 1])

target = 0
```

▶ **The slope calculation practice:**

[11]:
```
# Calculate the predictions: preds
preds = (weights * input_data).sum()

# Calculate the error: error
error = preds - target

# Calculate the slope: slope
slope = 2 * error * input_data
```

```
# Print the slope
print(slope)
```

```
[14 28 42]
```

▶ The model weights improving practice:

[12]:
```
# Set the learning rate: learning_rate
learning_rate = 0.01

# Calculate the predictions: preds
preds = (weights * input_data).sum()

# Calculate the error: error
error = preds - target

# Calculate the slope: slope
slope = 2 * input_data * error

# Update the weights: weights_updated
weights_updated = weights - learning_rate * slope

# Get updated predictions: preds_updated
preds_updated = (weights_updated * input_data).sum()

# Calculate updated error: error_updated
error_updated = preds_updated - target

# Print the original error
print(error)

# Print the updated error
print(error_updated)
```

```
7
5.04
```

▶ Package re-pre-loading:

[13]:
```
import matplotlib.pyplot as plt
```

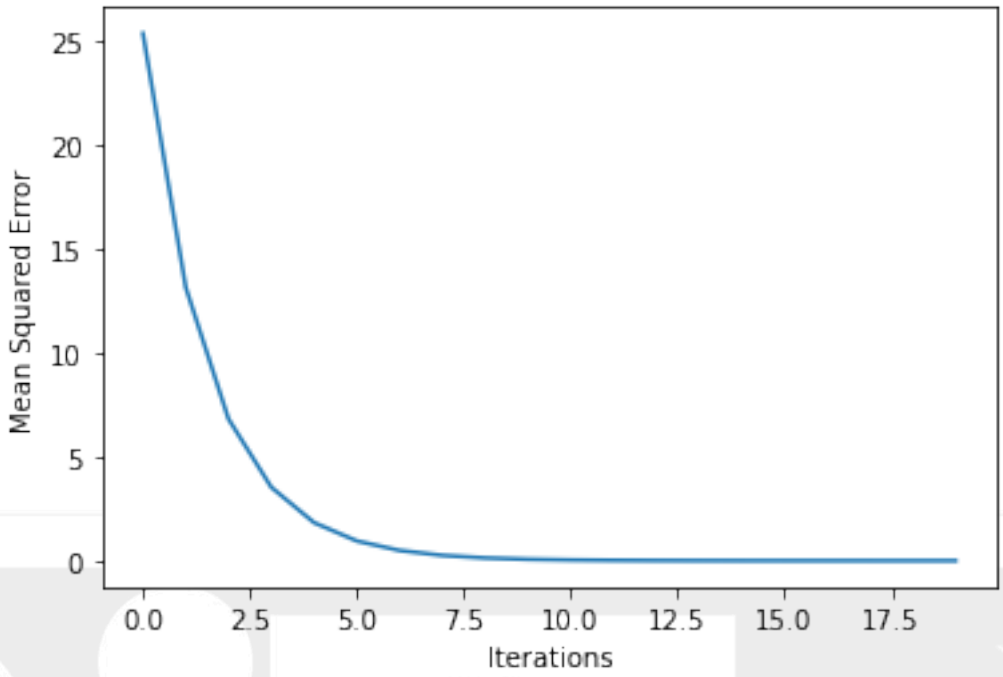
▶ Code pre-loading:

[14]:
```
def get_error(input_data, target, weights):
    preds = (weights * input_data).sum()
    error = preds - target
    return (error)
```

```
def get_slope(input_data, target, weights):
    error = get_error(input_data, target, weights)
    slope = 2 * input_data * error
    return (slope)


def get_mse(input_data, target, weights):
    errors = get_error(input_data, target, weights)
    mse = np.mean(errors**2)
    return (mse)
```

▶ **Weights multiple updates practice:**

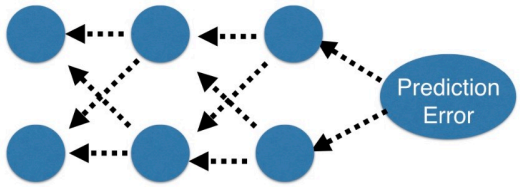
[15]:
```
n_updates = 20
mse_hist = []

# Iterate over the number of updates
for i in range(n_updates):
    # Calculate the slope: slope
    slope = get_slope(input_data, target, weights)

    # Update the weights: weights
    weights = weights - 0.01 * slope

    # Calculate mse with new weights: mse
    mse = get_mse(input_data, target, weights)

    # Append the mse to mse_hist
    mse_hist.append(mse)

# Plot the mse history
plt.plot(mse_hist)
plt.xlabel('Iterations')
plt.ylabel('Mean Squared Error')
plt.show()
```



3 Backpropagation

3.1 What is the aim of backpropagation?

Backpropagation aims to allow gradient descent to update all weights in the neural network by getting gradients for all weights. This idea comes from the chain rule of calculus. It is important to understand the process, but generally, a library is available that implements this.



3.2 What is the process of backpropagation?

- Try to estimate the slope of the loss function with respect to each weight.
- Do forward propagation to calculate predictions and errors.
- Go back one layer at a time.
- Gradients for weight is the product of:
    1. node value feeding into that weight
    2. slope of the loss function with respect to node it feeds into

3. slope of activation function at the node it feeds into

- It is also necessary to keep track of the slopes of the loss function with respect to node values.
- The slope of node values is the sum of the slopes for all weights that come out of them.

3.3 Practice question for the relationship between the forward and backward propagation:

- If gone through 4 iterations of calculating slopes (using backward propagation) and then updated weights, how many times must be done forward propagation?

  ☐ 0.

  ☐ 1.

  ☒ 4.

  ☐ 8.

3.4 Practice question for thinking about backward propagation:

- If the predictions were all exactly right, and the errors were all exactly 0, the slope of the loss function with respect to the predictions would also be 0. In that circumstance, which of the following statements would be correct?

  ☒ The updates to all weights in the network would also be 0.

  ☐ The updates to all weights in the network would be dependent on the activation functions.

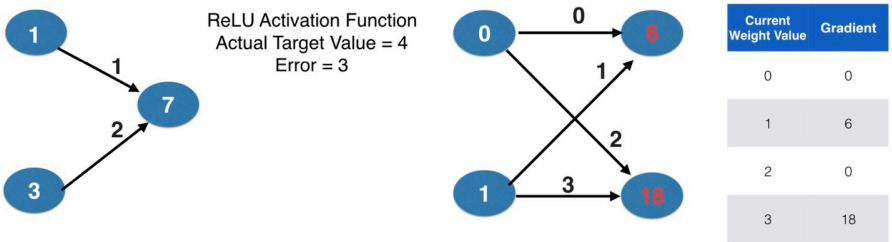
  ☐ The updates to all weights in the network would be proportional to values from the input data.

4 Backpropagation in practice

4.1 How to calculate slopes associated with any weight?

- Gradients for weight is the product of:

  1. node value feeding into that weight
  2. slope of activation function for the node being fed into
  3. slope of the loss function with respect to the output node



4.2 To recap backpropagation, what are the steps?

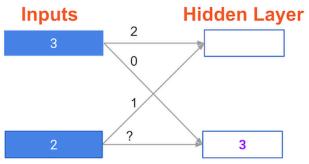
- Start at some random set of weights.
- Use forward propagation to make a prediction.
- Use backward propagation to calculate the slope of the loss function with respect to each weight.
- Multiply that slope by the learning rate, and subtract from the current weights.
- Keep going with that cycle until got a flat part.

4.3 What are the principles of stochastic gradient descent?

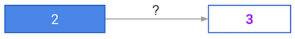
- What are the principles of stochastic gradient descent?
- It is common to calculate slopes on only a subset of the data (a batch).
- Use a different batch of data to calculate the next update.
- Start over from the beginning once all data is used.
- Each time through the training data is called an epoch.
- When slopes are calculated on one batch at a time, it is so-called stochastic gradient descent.

4.4 Practice question for a round of backpropagation:

- In the network shown below, forward propagation was done, and node values calculated as part of the forward propagation are shown in white. The weights are shown in black. Layers after the question mark show the slopes calculated as part of back-prop, rather than the forward-prop values. Those slope values are shown in purple.
- This network again uses the ReLU activation function, so the slope of the activation function is 1 for any node receiving a positive value as input. Assume the node being examined had a positive value (so the activation function's slope is 1).



- What is the slope needed to update the weight with the question mark?



☐ 0.

☐ 2.

☒ 6.

☐ Not enough information.