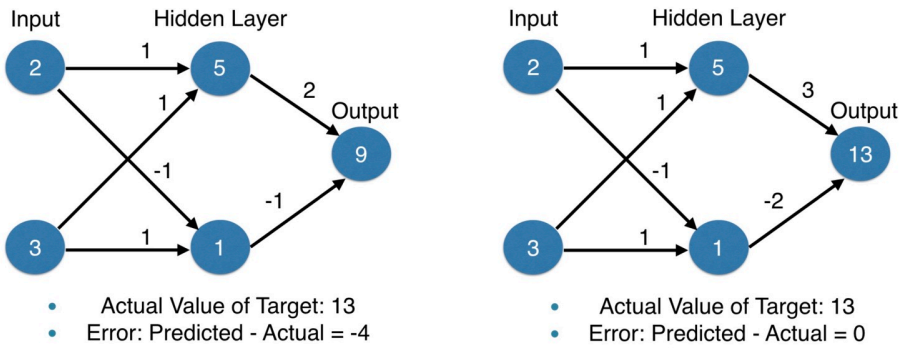
Optimizing a Neural Network with Backward Propagation

LaTeX, Puteaux, 2020, 2021

Table of Contents

1 The need for optimization

1.1 [note-1] A baseline neural network

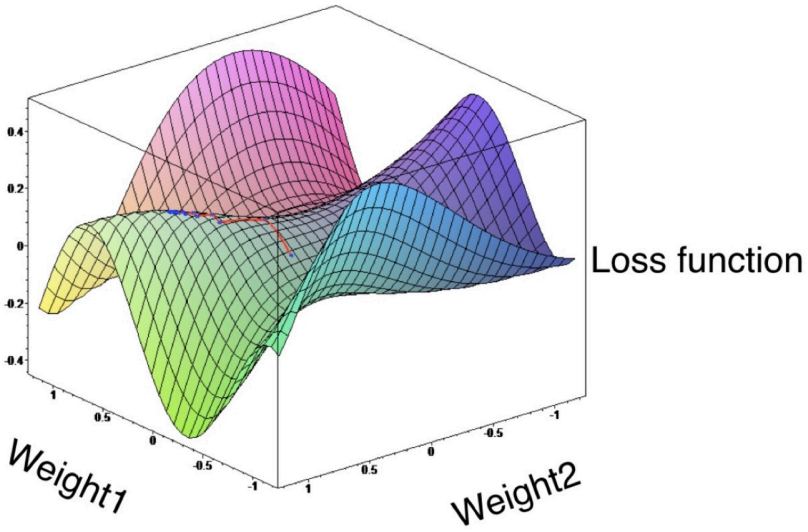


1.2 [note-2] Predictions with multiple points

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

1.3 [note-3] Loss function

- Aggregates errors in predictions from many data points into a single number.
- The loss function is a measure of the model's predictive performance.
- A lower loss function value means a better model.
- Goal: find the weights that give the lowest value for the loss function.
- Gradient descent.



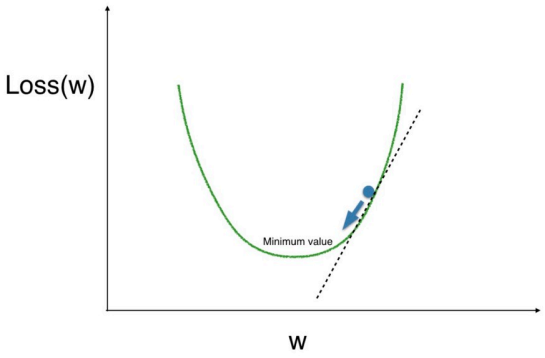
1.4 [note-4] Gradient descent

- Imagine starting in a pitch dark field.
- Want to find the lowest point.
- Feel the ground to see how it slopes.
- Take a small step downhill.
- Repeat until it is uphill in every direction.

1.5 [note-5] Gradient descent steps

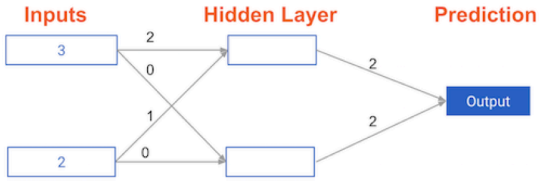
- Start at a random point.
- Until arrived at somewhere flat:
  - find the slope
  - take a step downhill

1.6 [note-6] Optimizing a model with a single weight



1.7 [quiz-1] 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.

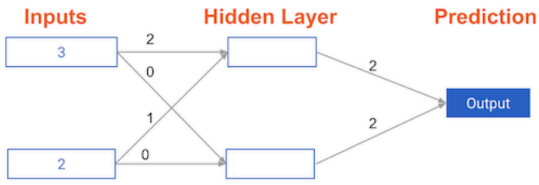
☐ 6.

☒ 11.

☐ 16.

1.8 [quiz-2] 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?



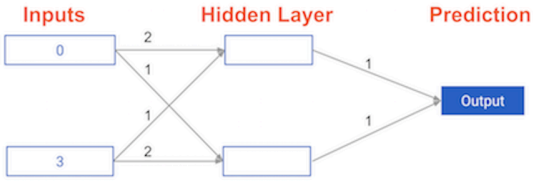
☐ More accurate.

☒ Less accurate.

☐ Stay the same.

1.9 [task-1] Coding how weight changes affect the accuracy

▶ Task diagram



▶ 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))
```

▶ Task 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
```

1.10 [task-2] Scaling up to multiple data points

▶ **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]
```

▶ **Task 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))
```

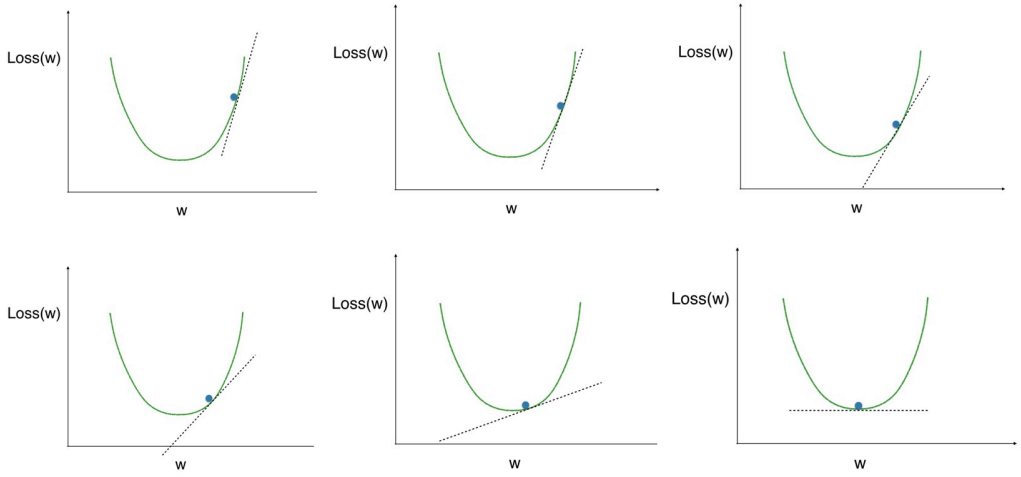
```
    # Append prediction to model_output_1
    model_output_1.append(predict_with_network(row, weights_1))

# 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 [note-1] Gradient descent

- 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
- Solution: *learning rate*
  - update each weight by subtracting *learning rate* × *slope*



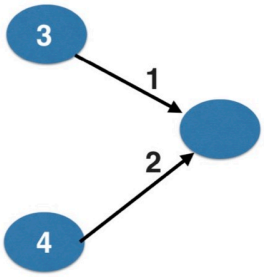
2.2 [note-2] Slope calculation example

- To calculate the slope for a weight, need to multiply:
  - the slope of the loss function with respect to value at the node which has been feed into
  - the value of the node that feeds into the weight
  - the slope of the activation function with respect to the value which has been feed into
- In this example, the slope of the mean-squared loss function with respect to prediction will be considered:

$$\begin{aligned} MSE' &= \Big((Predicted\ Value\ -\ Actual\ Value)^2\Big)' \\ &= 2\ \cdot\ (Predicted\ Value\ -\ Actual\ Value) \\ &= 2\ \cdot\ Error \end{aligned}$$

- In this example, the slope of the mean-squared loss function will be $2 \cdot (6 - 10) = 2 \cdot (-4)$, the value of the node that feeds into the weight is 3, and no activation function needs to be considered. So the final multiplied slope for the weight will be $2 \cdot (-4) \cdot 3 = -24$.
- In this example, if the learning rate is 0.01, the new weight would be $2 - 0.01 \cdot (-24) = 2.24$.

2.3 [note-3] Network with two inputs affecting prediction



2.4 [code-1] Calculate slopes and update weights

[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
```

2.5 [task-1] Calculating slopes

▶ **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
```

▶ **Task 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]
```

2.6 [task-2] Improving model weights

▶ **Task 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
```

2.7 [task-3] Making multiple updates to weights

▶ **Package pre-loading**

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

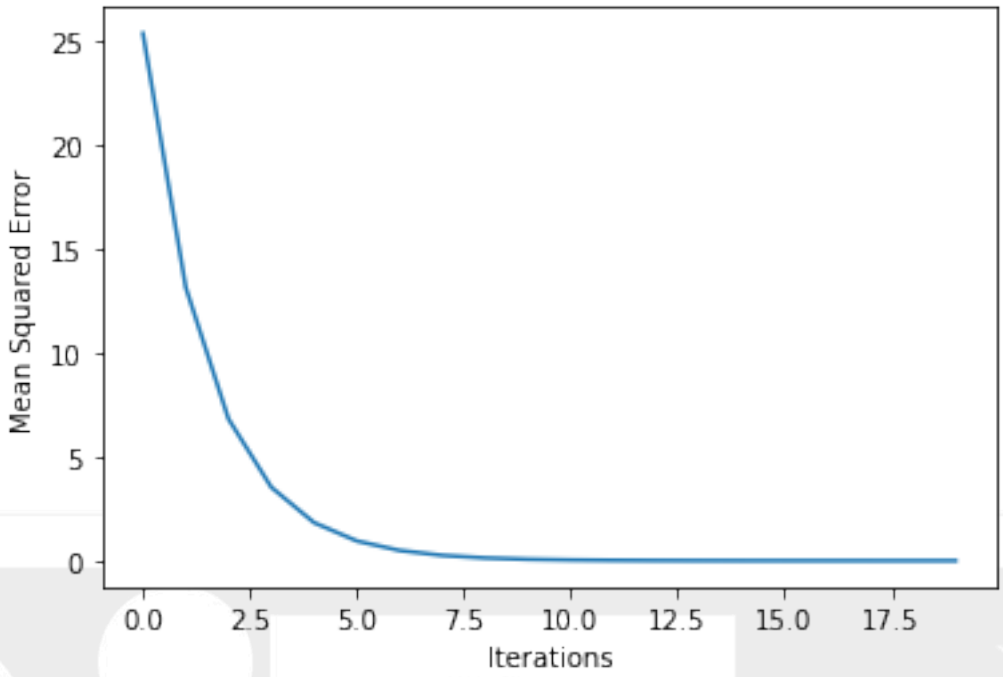
▶ **Code pre-loading**

[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)
```

▶ Task practice

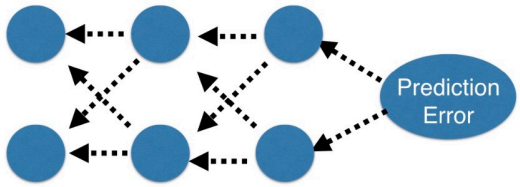
[15]:
```
n_updates = 20
mse_hist = []

# 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 [note-1] Backpropagation

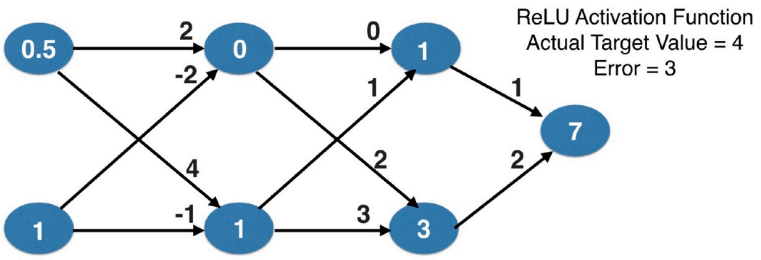
- Allows gradient descent to update all weights in the neural network (by getting gradients for all weights).
- This algorithm comes from the chain rule of calculus.
- Important to understand the process, but generally, there is a library available that implements this.



3.2 [note-2] Backpropagation process

- 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:

  - node value feeding into that weight
  - the slope of the loss function with respect to node it feeds into
  - the 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 [quiz-1] The relationship between the forward and the backward propagation

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

  ☐ 0.

  ☐ 1.

  ☒ 4.

  ☐ 8.

3.4 [quiz-2] 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 these 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.

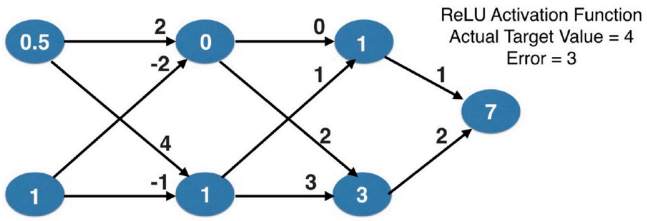
  ☐ The updates to all weights in the network would be proportional to values from the input data.

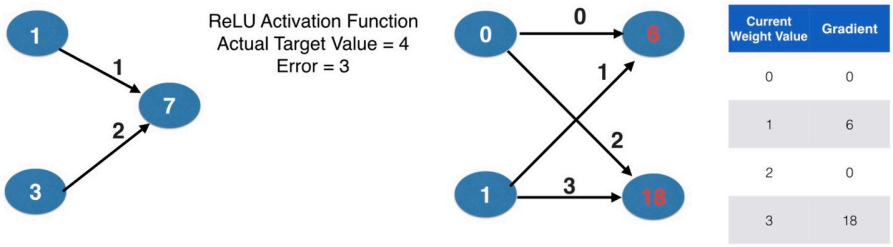
4 Backpropagation in practice

4.1 [note-1] Calculating slopes associated with any weight

- Gradients for weight is the product of:
  - node value feeding into that weight
  - the slope of activation function for the node being fed into

  - the slope of the loss function with respect to the output node





4.2 [note-2] Recapitulation of backpropagation

- 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 to a flat part.

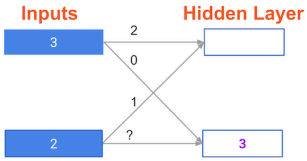
4.3 [note-3] Stochastic gradient descent

- 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:
  - stochastic gradient descent

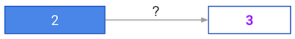
4.4 [quiz-1] A round of backpropagation

- In the network shown below, the forward propagation has been 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 the 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.

5 Requirements

[16]:
```
from platform import python_version
import sklearn
import matplotlib

python_version = ('python=={}'.format(python_version()))
numpy_version = ('numpy=={}'.format(np.__version__))
scikit_learn_version = ('scikit-learn=={}'.format(sklearn.__version__))
matplotlib_version = ('matplotlib=={}'.format(matplotlib.__version__))

writepath = '../../requirements.txt'
requirements = []
packages = [numpy_version, scikit_learn_version, matplotlib_version]

with open(writepath, 'w+') as file:
    for line in file:
        requirements.append(line.strip('\n'))
    for package in packages:
        if package not in requirements:
            file.write(package + '\n')

max_characters = len(python_version)
for package in packages:
    if max(max_characters, len(package)) > max_characters:
```

```
        max_characters = max(max_characters, len(package))

print('#' * (max_characters + 8))
print('#' * 2 + ' ' * (max_characters + 4) + '#' * 2)
print('#' * 2 + ' ' * 2 + python_version + ' ' *
      (max_characters - len(python_version) + 2) + '#' * 2)
print('#' * 2 + ' ' * 2 + numpy_version + ' ' *
      (max_characters - len(numpy_version) + 2) + '#' * 2)
print('#' * 2 + ' ' * 2 + scikit_learn_version + ' ' *
      (max_characters - len(scikit_learn_version) + 2) + '#' * 2)
print('#' * 2 + ' ' * 2 + matplotlib_version + ' ' *
      (max_characters - len(matplotlib_version) + 2) + '#' * 2)
print('#' * 2 + ' ' * (max_characters + 4) + '#' * 2)
print('#' * (max_characters + 8))
```

```
############################
##                        ##
##  python==3.7.9         ##
##  numpy==1.19.5         ##
##  scikit-learn==0.24.1  ##
##  matplotlib==3.3.4     ##
##                        ##
############################
```