The need for optimization

Puteaux, Fall/Winter 2020-2021

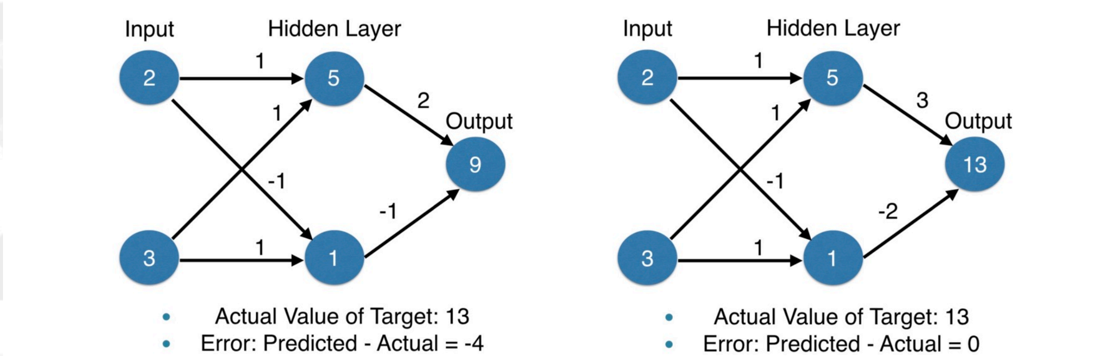
```
################################
##                            ##
##  Deep Learning in Python   ##
##                            ##
################################
```

§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?



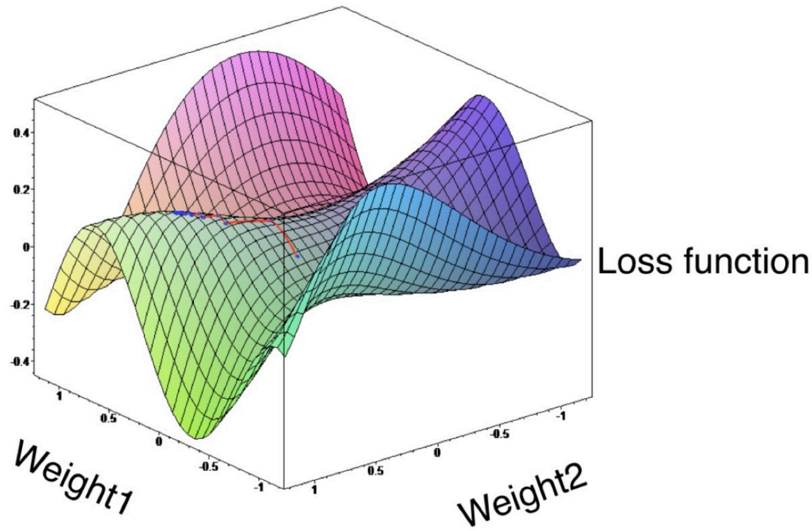
1.2 What are the challenges for the predictions with multiple points?

- 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.

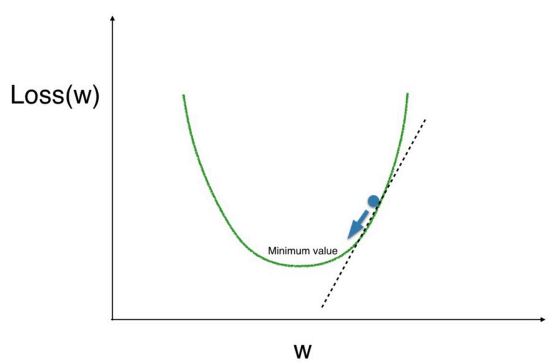
- The loss function's goal is to find the weights that give the lowest value for the loss function by gradient descent.



1.4 What are the steps of gradient descent?

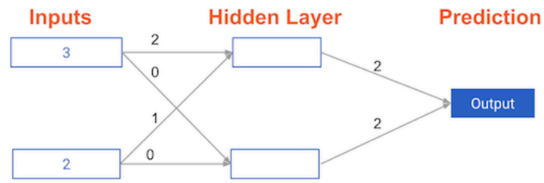
Start at a random point until got somewhere flat, find the slope, take a step downhill.

1.5 How to optimize a model with a single weight?



1.6 Practice question for calculating model errors:

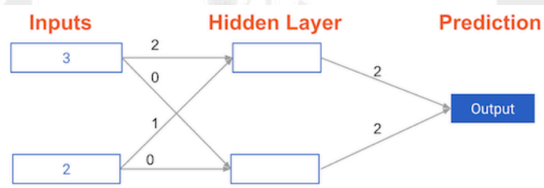
- Continue working with the network to predict transactions for a bank.
- What is the error ($error = predicted - actual$) for the following network using the ReLU activation function when the input data is $[3, 2]$, and the actual value of the target, which tries to predict, is 5? It may be helpful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

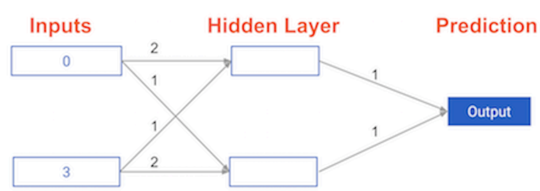
- Imagine making a prediction for a single data point. The actual value of the target is 7. The weight going from `node_0` to the output is 2, as shown below. If it is increased slightly, changing it to 2.01, would the predictions become more accurate, less accurate, or stay the same?



- ☐ 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]:
```python
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]:
```python
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
```