Backpropagation Algorithm

1

The backpropagation algorithm consists of two phases:

1. The forward pass where our inputs are passed through the network and output predictions obtained (also known as the propagation phase).

2. The backward pass where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule to update the weights in our network (also known as the weight update phase).

HUYTRAN

ARTIFICIAL INTELLIGENCE

1

Backpropagation Algorithm

2

Search for the related information of the following concepts:

1. Forward pass
2. Backward pass
3. Gradient
4. Loss function
5. Chain rule

HUYTRAN

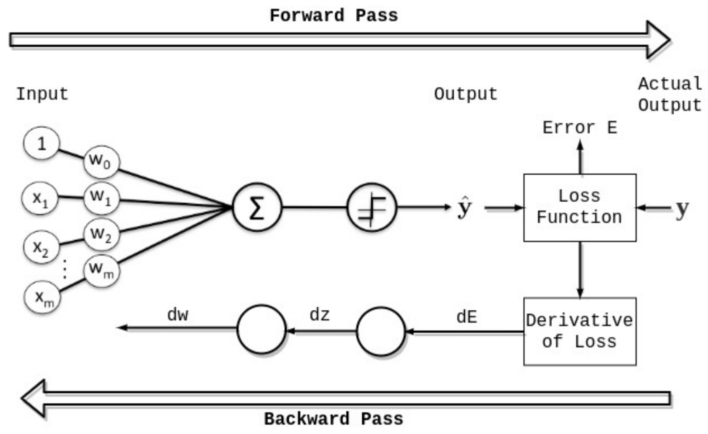
ARTIFICIAL INTELLIGENCE

2

The Forward Pass

3

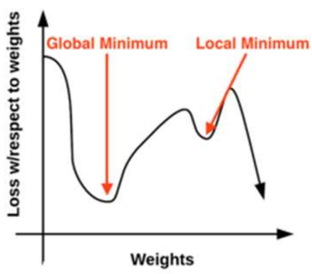
- The purpose of the forward pass is to propagate our inputs through the network by applying a series of dot products and activations until we reach the output layer of the network (i.e., our predictions).



Gradient

4

- The gradient descent method is an iterative optimization algorithm that operates over a loss landscape
- As we can see, our loss landscape has many peaks and valleys based on which values our parameters take on. Each peak is a local maximum that represents very high regions of loss –the local maximum with the largest loss across the entire loss landscape is the global maximum.
- Similarly, we also have local minimum which represents many small regions of loss



Gradient

5

- The surface of our bowl is the loss landscape, which is a plot of the loss function. The difference between our loss landscape and your cereal bowl is that your cereal bowl only exists in three dimensions, while your loss landscape exists in many dimensions, perhaps tens, hundreds, or thousands of dimensions.
- Each position along the surface of the bowl corresponds to a particular loss value given a set of parameters W (weight matrix) and b (bias vector). Our goal is to try different values of W and b, evaluate their loss, and then take a step towards more optimal values that (ideally) have lower loss.

5

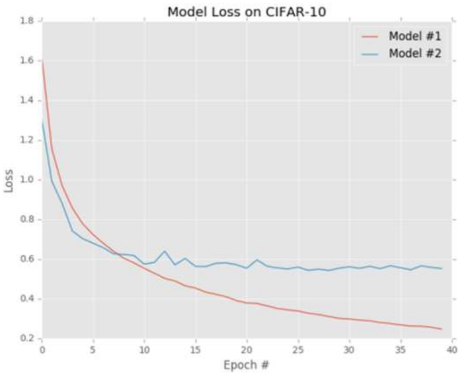
Loss function

6

The loss function quantifies how “good” or “bad” of a job a given model is doing classifying data points from the dataset. Model #1 achieves considerably lower loss than Model #2.

The smaller the loss, the better a job the classifier is at modeling the relationship between the input data and output class labels.

To improve our classification accuracy, we need to tune the parameters of our weight matrix W or bias vector b. Exactly how we go about updating these parameters is an optimization problem.



6

Loss function

7

Regression Losses

Mean Bias Error	Captures average bias in prediction. But is rarely used for training.	$\mathcal{L}_{\mathcal{MBE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i))$
Mean Absolute Error	Measures absolute average bias in prediction. Also called L1 Loss.	$\mathcal{L}_{\mathcal{MAE}} = \frac{1}{N}\sum_{i=1}^{N}\lvert y_i - f(x_i)\rvert$
Mean Squared Error	Average squared distance between actual and predicted. Also called L2 Loss.	$\mathcal{L}_{\mathcal{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i))^2$
Root Mean Squared Error	Square root of MSE. Loss and dependent variable have same units.	$\mathcal{L}_{\mathcal{RMSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i))^2}$
Huber Loss	A combination of MSE and MAE. It is parametric loss function.	$\mathcal{L}_{\text{Huberloss}} = \begin{cases} \frac{1}{2}(y_i - f(x_i))^2 & : \lvert y_i - f(x_i)\rvert \le \delta \\ \delta(\lvert y_i - f(x_i)\rvert - \frac{1}{2}\delta) & : otherwise \end{cases}$
Log Cosh Loss	Similar to Huber Loss + non-parametric. But computationally expensive.	$\mathcal{L}_{\text{LogCosh}} = \frac{1}{N}\sum_{i=1}^{N} log(cosh(f(x_i) - y_i))$

Classification Losses (Binary + Multi-class)

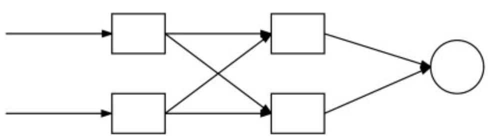
Binary Cross Entropy (BCE)	Loss function for binary classification tasks.	$\mathcal{L}_{\mathcal{BCE}} = \frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(x_i)) + (1 - y_i)\cdot log(1 - p(x_i))$
Hinge Loss	Penalizes wrong and right (but less confident) predictions. Commonly used in SVMs.	$\mathcal{L}_{\text{Hinge}} = max(0, 1 - (f(x)\cdot y))$
Cross Entropy Loss	Extension of BCE loss to multi-class classification.	$\mathcal{L}_{\mathcal{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}.log(f(x_{ij}))$ $N : samples; M : classes$
KL Divergence	Minimizes the divergence between predicted and true probability distribution	$\mathcal{L}_{\mathcal{KL}} = \sum_{i=1}^{N} y_i \cdot log(\frac{y_i}{f(x_i)})$

7

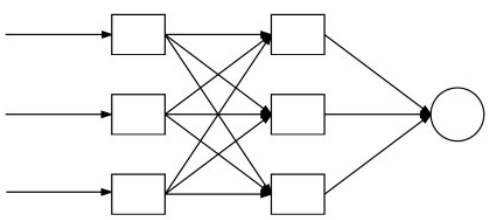
Backpropagation Algorithm

8

2-2-1



3-3-1



HUYTRAN

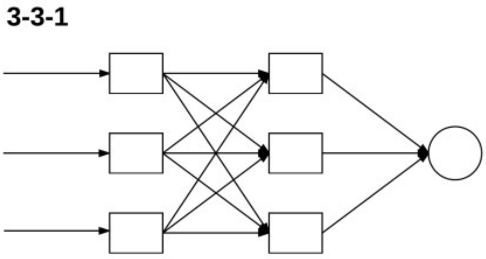
ARTIFICIAL INTELLIGENCE

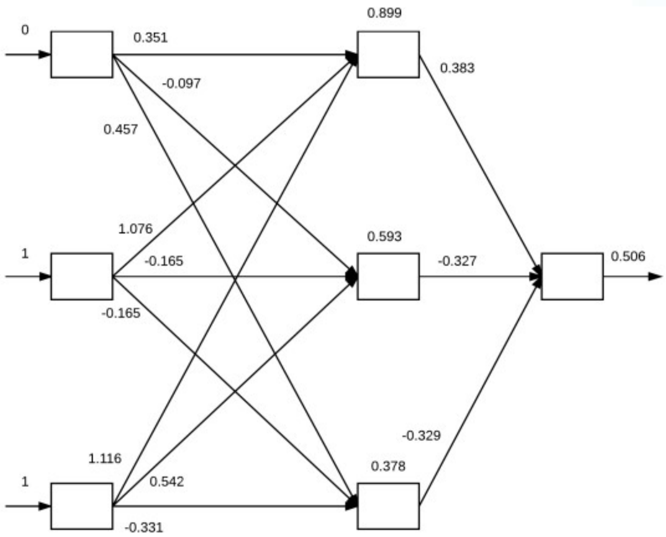
8

Backpropagation Algorithm

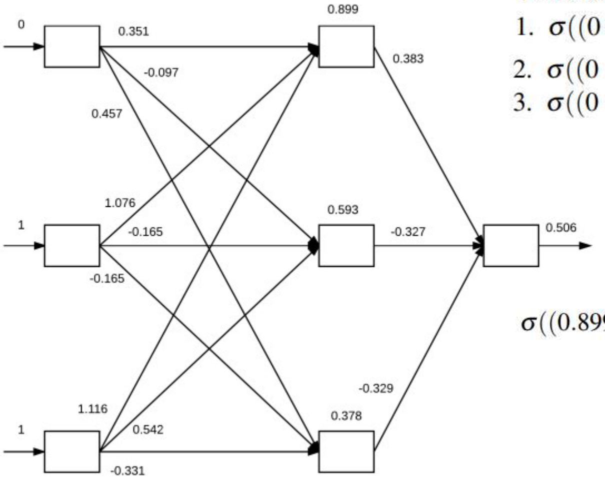
we present the feature vector (0,1,1) (and target output value 1 to the network). Here we can see that 0, 1, and 1 have been assigned to the three input nodes in the network.

To propagate the values through the network and obtain the final classification, we need to take the dot product between the inputs and the weight values, followed by applying an activation function (in this case, the sigmoid function, σ)





11



Let's compute the inputs to the three nodes in the hidden layers:

1. $\sigma((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$
2. $\sigma((0 \times -0.097) + (1 \times -0.165) + (1 \times 0.542)) = 0.593$
3. $\sigma((0 \times 0.457) + (1 \times -0.165) + (1 \times -0.331)) = 0.378$

$$\sigma((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

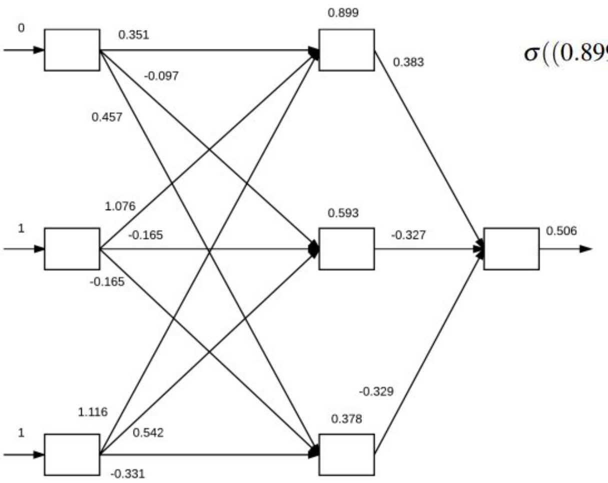
$$f(net) = \begin{cases} 1 & if net > 0 \\ 0 & otherwise \end{cases}$$

HUYTRAN

ARTIFICIAL INTELLIGENCE

11

12



$$\sigma((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

The output of the network is thus 0.506. We can apply a step function to determine if this output is the correct classification or not:

$$f(net) = \begin{cases} 1 & if net > 0 \\ 0 & otherwise \end{cases}$$

HUYTRAN

ARTIFICIAL INTELLIGENCE

12

13

$$f(net) = \begin{cases} 1 & if net > 0 \\ 0 & otherwise \end{cases}$$

Applying the step function with net = 0.506 we see that our network predicts 1 which is, in fact, the correct class label. However, our network is not very confident in this class label – the predicted value 0.506 is very close to the threshold of the step. Ideally, this prediction should be closer to 0.98–0.99, implying that our network has truly learned the underlying pattern in the dataset. In order for our network to actually "learn", we need to apply the backward pass.

HUYTRAN

ARTIFICIAL INTELLIGENCE

13

14

In order to apply the backpropagation algorithm, our activation function must be *differentiable* so that we can compute the *partial derivative* of the error with respect to a given weight $w_{i,j}$, loss (E), node output o_j, and network output net_j.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{i,j}} \tag{10.5}$$

HUYTRAN

ARTIFICIAL INTELLIGENCE

14

Implementing Backpropagation with Python

15

Open up a new file, name it neuralnetwork.py, store it in the nn submodule of pyimagesearch, and let's get to work:

```
1  # import the necessary packages
2  import numpy as np
3
4  class NeuralNetwork:
5      def __init__(self, layers, alpha=0.1):
6          # initialize the list of weights matrices, then store the
7          # network architecture and learning rate
8          self.W = []
9          self.layers = layers
10         self.alpha = alpha
```


Implementing Backpropagation with Python

16

```
1  # import the necessary packages
2  import numpy as np
3
4  class NeuralNetwork:
5      def __init__(self, layers, alpha=0.1):
6          # initialize the list of weights matrices, then store the
7          # network architecture and learning rate
8          self.W = []
9          self.layers = layers
10         self.alpha = alpha
```

Line 5 then defines the constructor to our NeuralNetwork class. The constructor requires a single argument, followed by a second optional one:

- layers: A list of integers which represents the actual architecture of the feedforward network. For example, a value of [2,2,1] would imply that our first input layer has two nodes, our hidden layer has two nodes, and our final output layer has one node.
- alpha: Here we can specify the learning rate of our neural network. This value is applied during the weight update phase.

Implementing Backpropagation with Python

17

```
1  # import the necessary packages
2  import numpy as np
3
4  class NeuralNetwork:
5      def __init__(self, layers, alpha=0.1):
6          # initialize the list of weights matrices, then store the
7          # network architecture and learning rate
8          self.W = []
9          self.layers = layers
10         self.alpha = alpha
```

Line 8 initializes our list of weights for each layer, W.
We then store layers and alpha on Lines 9 and 10.
Our weights list W is empty, so let's go ahead and initialize it

HUYTRAN

ARTIFICIAL INTELLIGENCE

17

Implementing Backpropagation with Python

18

```
12         # start looping from the index of the first layer but
13         # stop before we reach the last two layers
14         for i in np.arange(0, len(layers) - 2):
15             # randomly initialize a weight matrix connecting the
16             # number of nodes in each respective layer together,
17             # adding an extra node for the bias
18             w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
19             self.W.append(w / np.sqrt(layers[i]))
```

On Line 14 we start looping over the number of layers in the network (i.e., len(layers)), but we stop before the final two layer.
Each layer in the network is randomly initialized by constructing an MxN weight matrix by sampling values from a standard, normal distribution (Line 18). The matrix is MxN since we wish to connect every node in current layer to every node in the next layer.

HUYTRAN

ARTIFICIAL INTELLIGENCE

18

Implementing Backpropagation with Python

19

```
12        # start looping from the index of the first layer but
13        # stop before we reach the last two layers
14        for i in np.arange(0, len(layers) - 2):
15            # randomly initialize a weight matrix connecting the
16            # number of nodes in each respective layer together,
17            # adding an extra node for the bias
18            w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
19            self.W.append(w / np.sqrt(layers[i]))
```

We scale w by dividing by the square root of the number of nodes in the current layer, thereby normalizing the variance of each neuron's output (Line 19).

HUYTRAN

ARTIFICIAL INTELLIGENCE

19

Implementing Backpropagation with Python

20

The final code block of the constructor handles the special case where the input connections need a bias term, but the output does not:

```
        # the last two layers are a special case where the input
        # connections need a bias term but the output does not
        w = np.random.randn(layers[-2] + 1, layers[-1])
        self.W.append(w / np.sqrt(layers[-2]))
```

Again, these weight values are randomly sampled and then normalized.

HUYTRAN

ARTIFICIAL INTELLIGENCE

20

Implementing Backpropagation with Python

21

The next function we define is a Python “magic method” named __repr__ – this function is useful for debugging:

```
    def __repr__(self):
        # construct and return a string that represents the network
        # architecture
        return "NeuralNetwork: {}".format(
            "-".join(str(l) for l in self.layers))
```

In our case, we’ll format a string for our NeuralNetwork object by concatenating the integer value of the number of nodes in each layer.

Implementing Backpropagation with Python

22

Given a layers value of (2, 2, 1), the output of calling this function will be:

```
>>> from pyimagesearch.nn import NeuralNetwork
>>> nn = NeuralNetwork([2, 2, 1])
>>> print(nn)
NeuralNetwork: 2-2-1
```

Implementing Backpropagation with Python

23

Next, we can define our sigmoid activation function:

```
    def sigmoid(self, x):
        # compute and return the sigmoid activation value for a
        # given input value
        return 1.0 / (1 + np.exp(-x))
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

23

Implementing Backpropagation with Python

24

As well as the derivative of the sigmoid which we'll use during the backward pass:

```
    def sigmoid_deriv(self, x):
        # compute the derivative of the sigmoid function ASSUMING
        # that `x` has already been passed through the `sigmoid`
        # function
        return x * (1 - x)
```

Again, note that whenever you perform backpropagation, you'll always want to choose an activation function that is differentiable

HUYTRAN

ARTIFICIAL INTELLIGENCE

24

Implementing Backpropagation with Python

25

```python
    def fit(self, X, y, epochs=1000, displayUpdate=100):
        # insert a column of 1's as the last entry in the feature
        # matrix -- this little trick allows us to treat the bias
        # as a trainable parameter within the weight matrix
        X = np.c_[X, np.ones((X.shape[0]))]

        # loop over the desired number of epochs
        for epoch in np.arange(0, epochs):
            # loop over each individual data point and train
            # our network on it
            for (x, target) in zip(X, y):
                self.fit_partial(x, target)

            # check to see if we should display a training update
            if epoch == 0 or (epoch + 1) % displayUpdate == 0:
                loss = self.calculate_loss(X, y)
                print("[INFO] epoch={}, loss={:.7f}".format(
                    epoch + 1, loss))
```

We'll draw inspiration from the scikit-learn library and define a function named fit which will be responsible for actually training our NeuralNetwork

HUYTRAN

ARTIFICIAL INTELLIGENCE

25

Implementing Backpropagation with Python

26

```python
    def fit(self, X, y, epochs=1000, displayUpdate=100):
        # insert a column of 1's as the last entry in the feature
        # matrix -- this little trick allows us to treat the bias
        # as a trainable parameter within the weight matrix
        X = np.c_[X, np.ones((X.shape[0]))]
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

26

Implementing Backpropagation with Python

27

```python
        # loop over the desired number of epochs
        for epoch in np.arange(0, epochs):
            # loop over each individual data point and train
            # our network on it
            for (x, target) in zip(X, y):
                self.fit_partial(x, target)

            # check to see if we should display a training update
            if epoch == 0 or (epoch + 1) % displayUpdate == 0:
                loss = self.calculate_loss(X, y)
                print("[INFO] epoch={}, loss={:.7f}".format(
                    epoch + 1, loss))
```


Implementing Backpropagation with Python

28

The actual heart of the backpropagation algorithm is found inside our fit_partial method below:

```python
    def fit_partial(self, x, y):
        # construct our list of output activations for each layer
        # as our data point flows through the network; the first
        # activation is a special case -- it's just the input
        # feature vector itself
        A = [np.atleast_2d(x)]
```

Implementing Backpropagation with Python

29

From here, we can start the forward propagation phase:

```
        # FEEDFORWARD:
        # loop over the layers in the network
        for layer in np.arange(0, len(self.W)):
            # feedforward the activation at the current layer by
            # taking the dot product between the activation and
            # the weight matrix -- this is called the "net input"
            # to the current layer
            net = A[layer].dot(self.W[layer])

            # computing the "net output" is simply applying our
            # nonlinear activation function to the net input
            out = self.sigmoid(net)

            # once we have the net output, add it to our list of
            # activations
            A.append(out)
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

29

Implementing Backpropagation with Python

30

```
        # FEEDFORWARD:
        # loop over the layers in the network
        for layer in np.arange(0, len(self.W)):
            # feedforward the activation at the current layer by
            # taking the dot product between the activation and
            # the weight matrix -- this is called the "net input"
            # to the current layer
            net = A[layer].dot(self.W[layer])

            # computing the "net output" is simply applying our
            # nonlinear activation function to the net input
            out = self.sigmoid(net)

            # once we have the net output, add it to our list of
            # activations
            A.append(out)
```

The final entry in A is thus the output of the last layer in our network

HUYTRAN

ARTIFICIAL INTELLIGENCE

30

Implementing Backpropagation with Python

31

Now that the forward pass is done, we can move on to the slightly more complicated backward pass:

```
86          # BACKPROPAGATION
87          # the first phase of backpropagation is to compute the
88          # difference between our *prediction* (the final output
89          # activation in the activations list) and the true target
90          # value
91          error = A[-1] - y
92
93          # from here, we need to apply the chain rule and build our
94          # list of deltas `D`; the first entry in the deltas is
95          # simply the error of the output layer times the derivative
96          # of our activation function for the output value
97          D = [error * self.sigmoid_deriv(A[-1])]
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

31

Implementing Backpropagation with Python

32

The first phase of the backward pass is to compute our error, or simply the difference between our predicted label and the ground-truth label (Line 91).
Since the final entry in the activations list A contains the output of the network, we can access the output prediction via A[-1]. The value y is the target output for the input data point x.

```
91          error = A[-1] - y
92
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

32

Implementing Backpropagation with Python

33

Next, we need to start applying the chain rule to build our list of deltas, D. The deltas will be used to update our weight matrices, scaled by the learning rate alpha.
The first entry in the deltas list is the error of our output layer multiplied by the derivative of the sigmoid for the output value (Line 97)

```
97          D = [error * self.sigmoid_deriv(A[-1])]
```


Implementing Backpropagation with Python

34

Given the delta for the final layer in the network, we can now work backward using a for loop:

```
        # once you understand the chain rule it becomes super easy
        # to implement with a `for` loop -- simply loop over the
        # layers in reverse order (ignoring the last two since we
        # already have taken them into account)
        for layer in np.arange(len(A) - 2, 0, -1):
            # the delta for the current layer is equal to the delta
            # of the *previous layer* dotted with the weight matrix
            # of the current layer, followed by multiplying the delta
            # by the derivative of the nonlinear activation function
            # for the activations of the current layer
            delta = D[-1].dot(self.W[layer].T)
            delta = delta * self.sigmoid_deriv(A[layer])
            D.append(delta)
```

Implementing Backpropagation with Python


```
        for layer in np.arange(len(A) - 2, 0, -1):
            delta = D[-1].dot(self.W[layer].T)
            delta = delta * self.sigmoid_deriv(A[layer])
            D.append(delta)
```


Implementing Backpropagation with Python

Given our deltas list D, we can move on to the weight update phase:

```
        D = D[::-1]

        # WEIGHT UPDATE PHASE
        # loop over the layers
        for layer in np.arange(0, len(self.W)):
            # update our weights by taking the dot product of the layer
            # activations with their respective deltas, then multiplying
            # this value by some small learning rate and adding to our
            # weight matrix -- this is where the actual "learning" takes
            # place
            self.W[layer] += -self.alpha * A[layer].T.dot(D[layer])
```

Implementing Backpropagation with Python

37

```
        D = D[::-1]

        # WEIGHT UPDATE PHASE
        # loop over the layers
        for layer in np.arange(0, len(self.W)):
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

37

Implementing Backpropagation with Python

38

```
        for layer in np.arange(0, len(self.W)):
            # update our weights by taking the dot product of the layer
            # activations with their respective deltas, then multiplying
            # this value by some small learning rate and adding to our
            # weight matrix -- this is where the actual "learning" takes
            # place
            self.W[layer] += -self.alpha * A[layer].T.dot(D[layer])
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

38

Implementing Backpropagation with Python

39

Once our network is trained on a given dataset, we'll want to make predictions on the testing set, which can be accomplished via the predict method below:

```
def predict(self, X, addBias=True):
    # initialize the output prediction as the input features -- this
    # value will be (forward) propagated through the network to
    # obtain the final prediction
    p = np.atleast_2d(X)

    # check to see if the bias column should be added
    if addBias:
        # insert a column of 1's as the last entry in the feature
        # matrix (bias)
        p = np.c_[p, np.ones((p.shape[0]))]

    # loop over our layers in the network
    for layer in np.arange(0, len(self.W)):
        # computing the output prediction is as simple as taking
        # the dot product between the current activation value `p`
        # and the weight matrix associated with the current layer,
        # then passing this value through a nonlinear activation
        # function
        p = self.sigmoid(np.dot(p, self.W[layer]))

    # return the predicted value
    return p
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

39

Implementing Backpropagation with Python

40

```
def predict(self, X, addBias=True):
    # initialize the output prediction as the input features -- this
    # value will be (forward) propagated through the network to
    # obtain the final prediction
    p = np.atleast_2d(X)
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

40

Implementing Backpropagation with Python

41

```
        # check to see if the bias column should be added
        if addBias:
            # insert a column of 1's as the last entry in the feature
            # matrix (bias)
            p = np.c_[p, np.ones((p.shape[0]))]
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

41

Implementing Backpropagation with Python

42

```
        # loop over our layers in the network
        for layer in np.arange(0, len(self.W)):
            # computing the output prediction is as simple as taking
            # the dot product between the current activation value `p`
            # and the weight matrix associated with the current layer,
            # then passing this value through a nonlinear activation
            # function
            p = self.sigmoid(np.dot(p, self.W[layer]))

        # return the predicted value
        return p
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

42

Implementing Backpropagation with Python

43

The final function we'll define inside the NeuralNetwork class will be used to calculate the loss across our entire training set:

```
    def calculate_loss(self, X, targets):
        # make predictions for the input data points then compute
        # the loss
        targets = np.atleast_2d(targets)
        predictions = self.predict(X, addBias=False)
        loss = 0.5 * np.sum((predictions - targets) ** 2)

        # return the loss
        return loss
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

43

Backpropagation with Python Example #1: Bitwise XOR

44

Go ahead and open up a new file, name it nn_xor.py, and insert the following code:

```
# import the necessary packages
from pyimagesearch.nn import NeuralNetwork
import numpy as np

# construct the XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

44

Backpropagation with Python Example #1: Bitwise XOR

45

We can now define our network architecture and train it:

```python
# define our 2-2-1 neural network and train it
nn = NeuralNetwork([2, 2, 1], alpha=0.5)
nn.fit(X, y, epochs=20000)
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

45

Backpropagation with Python Example #1: Bitwise XOR

46

Once our network is trained, we'll loop over our XOR datasets, allow the network to predict the output for each one, and display the prediction to our screen:

```python
# now that our network is trained, loop over the XOR data points
for (x, target) in zip(X, y):
    # make a prediction on the data point and display the result
    # to our console
    pred = nn.predict(x)[0][0]
    step = 1 if pred > 0.5 else 0
    print("[INFO] data={}, ground-truth={}, pred={:.4f}, step={}".format(
        x, target[0], pred, step))
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

46

Backpropagation with Python Example #1: Bitwise XOR

47

To train our neural network using backpropagation with Python, simply execute the following command:

```
$ python nn_xor.py
[INFO] epoch=1, loss=0.5092796
[INFO] epoch=100, loss=0.4923591
[INFO] epoch=200, loss=0.4677865
...
[INFO] epoch=19800, loss=0.0002478
[INFO] epoch=19900, loss=0.0002465
[INFO] epoch=20000, loss=0.0002452
```

HUYTRAN

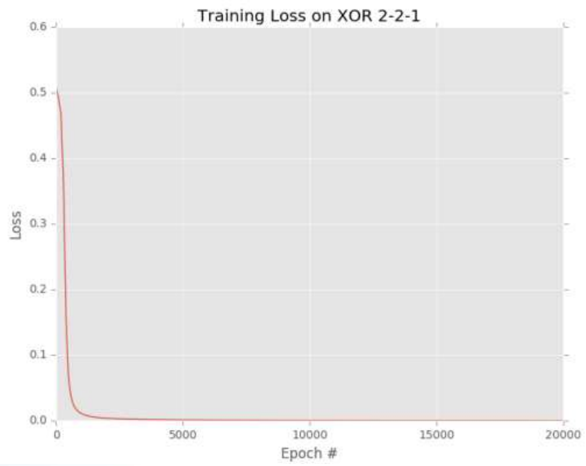
ARTIFICIAL INTELLIGENCE

47

Backpropagation with Python Example #1: Bitwise XOR

48

A plot of the squared loss is displayed below (Figure 10.11). As we can see, loss slowly decreases to approximately zero over the course of training.



HUYTRAN

ARTIFICIAL INTELLIGENCE

48

Backpropagation with Python Example #1: Bitwise XOR

49

Furthermore, looking at the final four lines of the output we can see our predictions:

```
[INFO] data=[0 0], ground-truth=0, pred=0.0054, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.9894, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.9876, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.0140, step=0
```

HUYTRAN

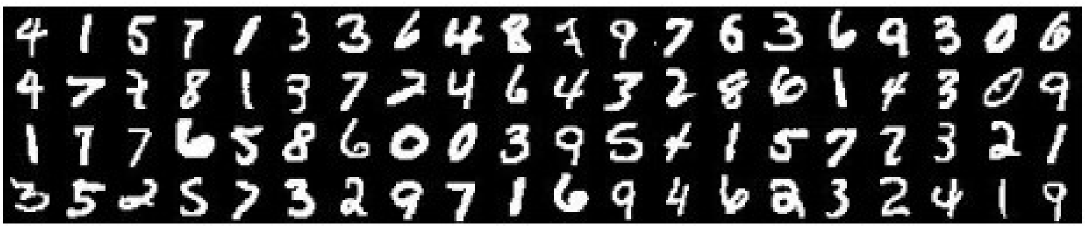
ARTIFICIAL INTELLIGENCE

49

Backpropagation with Python Example: MNIST Sample

50

Let's examine a subset of the MNIST dataset for handwritten digit recognition. This subset of the MNIST dataset is built-into the scikit-learn library and includes 1,797 example digits, each of which are 8×8 grayscale images (the original images are 28×28). When flattened, these images are represented by an 8×8 = 64-dim vector.



HUYTRAN

ARTIFICIAL INTELLIGENCE

50

51

```python
# import the necessary packages
from pyimagesearch.nn import NeuralNetwork
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn import datasets
```

```python
# load the MNIST dataset and apply min/max scaling to scale the
# pixel intensity values to the range [0, 1] (each image is
# represented by an 8 x 8 = 64-dim feature vector)
print("[INFO] loading MNIST (sample) dataset...")
digits = datasets.load_digits()
data = digits.data.astype("float")
data = (data - data.min()) / (data.max() - data.min())
print("[INFO] samples: {}, dim: {}".format(data.shape[0],
    data.shape[1]))
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

51

52

```python
# construct the training and testing splits
(trainX, testX, trainY, testY) = train_test_split(data,
    digits.target, test_size=0.25)

# convert the labels from integers to vectors
trainY = LabelBinarizer().fit_transform(trainY)
testY = LabelBinarizer().fit_transform(testY)
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

52

```
# train the network
print("[INFO] training network...")
nn = NeuralNetwork([trainX.shape[1], 32, 16, 10])
print("[INFO] {}".format(nn))
nn.fit(trainX, trainY, epochs=1000)
```

Here we can see that we are training a NeuralNetwork with a 64−32−16−10 architecture. The output layer has ten nodes due to the fact that there are ten possible output classes for the digits 0-9. We then allow our network to train for 1,000 epochs.

HUYTRAN

ARTIFICIAL INTELLIGENCE

53

54

Once our network has been trained, we can evaluate it on the testing set:

```
# evaluate the network
print("[INFO] evaluating network...")
predictions = nn.predict(testX)
predictions = predictions.argmax(axis=1)
print(classification_report(testY.argmax(axis=1), predictions))
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

54

55

```
$ python nn_mnist.py
[INFO] loading MNIST (sample) dataset...
[INFO] samples: 1797, dim: 64
[INFO] training network...
[INFO] NeuralNetwork: 64-32-16-10
[INFO] epoch=1, loss=604.5868589
[INFO] epoch=100, loss=9.1163376
[INFO] epoch=200, loss=3.7157723
[INFO] epoch=300, loss=2.6078803
[INFO] epoch=400, loss=2.3823153
[INFO] epoch=500, loss=1.8420944
[INFO] epoch=600, loss=1.3214138
[INFO] epoch=700, loss=1.2095033
[INFO] epoch=800, loss=1.1663942
[INFO] epoch=900, loss=1.1394731
[INFO] epoch=1000, loss=1.1203779
[INFO] evaluating network...
```

HUYTRAN

ARTIFICIAL INTELLIGENCE

55

56

	precision	recall	f1-score	support
0	1.00	1.00	1.00	45
1	0.98	1.00	0.99	51
2	0.98	1.00	0.99	47
3	0.98	0.93	0.95	43
4	0.95	1.00	0.97	39
5	0.94	0.97	0.96	35
6	1.00	1.00	1.00	53
7	1.00	1.00	1.00	49
8	0.97	0.95	0.96	41
9	1.00	0.96	0.98	47
avg / total	0.98	0.98	0.98	450

HUYTRAN

ARTIFICIAL INTELLIGENCE

56

57

Notice how our loss starts off very high, but quickly drops during the training process. Our classification report demonstrates that we are obtaining ≈ 98% classification accuracy on our testing set; however, we are having some trouble classifying digits 4 and 5 (95% and 94% accuracy, respectively).

HUYTRAN

ARTIFICIAL INTELLIGENCE

57