# Neural Networks

Perceptrons

A perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data

For a point with coordinates (p,q), label y, and prediction given by the equation

y^​=step(w1​x1​+w2​x2​+b):

* If the point is correctly classified, do nothing.
* If the point is classified positive, but it has a negative label, subtract  αp,αq, and α from w1​,w2​, and b respectively.
* If the point is classified negative, but it has a positive label, add  αp,αq, and α to  w1​,w2 , and b respectively.

Error Function

Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function

To apply gradient descent the error function should be continuous and differentiable.

Log-loss Error

-y\*np.log(output) - (1 - y) \* np.log(1-output)

Backpropagation

Output = sigmoid(np.dot(x, weights)

Error formula= -y\*np.log(output) - (1 - y) \* np.log(1-output)

error\_term = (y-output)\*sigmoid\_prime(x) = (y-output)\*( sigmoid(x) \* (1-sigmoid(x))

Gradient descent step

del\_w += error\_term \* x

Update weights

weights += learnrate \* del\_w / n\_records

Mean Squared Error

The mean of the squares of the differences between the predictions and the labels.

A close up of a clock

Description automatically generated

First, the inside sum over j. This variable j represents the output units of the network. So this inside sum is saying for each output unit, find the difference between the true value y and the predicted value from the network y^​, then square the difference, then sum up all those squares.

Then the other sum over μ is a sum over all the data points. So, for each data point you calculate the inner sum of the squared differences for each output unit. Then you sum up those squared differences for each data point. That gives you the overall error for all the output predictions for all the data points.

Multilayer Perceptrons

A screenshot of a cell phone

Description automatically generated

A close up of a clock

Description automatically generated

A forward pass through a 4x3x2 network, with sigmoid activation functions for both layers

hidden\_layer\_in = np.dot(X,weights\_input\_to\_hidden)

hidden\_layer\_out = sigmoid(hidden\_layer\_in)

output\_layer\_in = np.dot(hidden\_layer\_out,weights\_hidden\_to\_output)

output\_layer\_out = sigmoid(output\_layer\_in)

Backpropagation

Now we've seen that the error term for the output layer is

*δk*​=(*yk*​−*y*^​*k*​)*f*′(*ak*​)

and the error term for the hidden layer is

[[A picture containing drawing

Description automatically generated](https://classroom.udacity.com/nanodegrees/nd101/parts/94643112-2cab-46f8-a5be-1b6e4fa7a211/modules/89a1ec1d-4c22-4a77-b230-b0da99240c89/lessons/07f472eb-0210-446f-8ec2-d297b06c86d0/concepts/b2bbdc9a-9f48-4735-b408-71cf67f5b000)](https://classroom.udacity.com/nanodegrees/nd101/parts/94643112-2cab-46f8-a5be-1b6e4fa7a211/modules/89a1ec1d-4c22-4a77-b230-b0da99240c89/lessons/07f472eb-0210-446f-8ec2-d297b06c86d0/concepts/b2bbdc9a-9f48-4735-b408-71cf67f5b000)

For now we'll only consider a simple network with one hidden layer and one output unit. Here's the general algorithm for updating the weights with backpropagation:

* Set the weight steps for each layer to zero
  + The input to hidden weights Δ*wij*​=0
  + The hidden to output weights Δ*Wj*​=0
* For each record in the training data:
  + Make a forward pass through the network, calculating the output *y*^​
  + Calculate the error gradient in the output unit,  *δo*=(*y*−*y*^​)*f*′(*z*) where *z*=∑*j*​*Wj*​*aj*​, the input to the output unit.
  + Propagate the errors to the hidden layer *δjh*​=*δoWj*​*f*′(*hj*​)
  + Update the weight steps:
    - Δ*wij*​=Δ*wij*​+*δjh*​*ai*​
    - Δ*Wj*​=Δ*Wj*​+*δoaj*​
* Update the weights, where *η* is the learning rate and *m* is the number of records:
  + *Wj*​=*Wj*​+*η*Δ*Wj*​/*m*
  + *wij*​=*wij*​+*η*Δ*wij*​/*m*
* Repeat for *e* epochs.

Example-

* Assume we're trying to fit some binary data and the target is  =1. We'll start with the forward pass, first calculating the input to the hidden unit

h =∑*i*​*wi*​*xi*​ = 0.1×0.4−0.2×0.3=−0.02

and the output of the hidden unit

*a*=*f*(*h*)=sigmoid(−0.02)=0.495.

Using this as the input to the output unit, the output of the network is

*y*^​=*f*(*W*⋅*a*)=sigmoid(0.1×0.495)=0.512.

A close up of a sign

Description automatically generated

With the network output, we can start the backwards pass to calculate the weight updates for both layers. Using the fact that for the sigmoid function

*f*′(*W*⋅*a*)=*f*(*W*⋅*a*)(1−*f*(*W*⋅*a*)),

The error term for the output unit is

*δo*=(*y*−*y*^​)*f*′(*W*⋅*a*)=(1−0.512)×0.512×(1−0.512)=0.122.

Now we need to calculate the error term for the hidden unit with backpropagation. Here we'll scale the error term from the output unit by the weight *W* connecting it to the hidden unit. For the hidden unit error term,

*δjh* ​ = ∑*k*​*Wjk*​*δko*​*f*′(*hj*​),

But since we have one hidden unit and one output unit, this is much simpler.

*δh*=*Wδof*′(*h*)=0.1×0.122×0.495×(1−0.495)=0.003

Now that we have the errors, we can calculate the gradient descent steps. The hidden to output weight step is the learning rate, times the output unit error, times the hidden unit activation value.

Δ*W*=*ηδoa*=0.5×0.122×0.495=0.0302

Then, for the input to hidden weights *wi*​, it's the learning rate times the hidden unit error, times the input values.

Δ*wi*​=*ηδhxi*​=(0.5×0.003×0.1,0.5×0.003×0.3)=(0.00015,0.00045)

* Example 2

import numpy as np

from data\_prep import features, targets, features\_test, targets\_test

np.random.seed(21)

def sigmoid(x):

"""

Calculate sigmoid

"""

return 1 / (1 + np.exp(-x))

# Hyperparameters

n\_hidden = 2 # number of hidden units

epochs = 900

learnrate = 0.005

n\_records, n\_features = features.shape

last\_loss = None

# Initialize weights

weights\_input\_hidden = np.random.normal(scale=1 / n\_features \*\* .5,

size=(n\_features, n\_hidden))

weights\_hidden\_output = np.random.normal(scale=1 / n\_features \*\* .5,

size=n\_hidden)

for e in range(epochs):

del\_w\_input\_hidden = np.zeros(weights\_input\_hidden.shape)

del\_w\_hidden\_output = np.zeros(weights\_hidden\_output.shape)

for x, y in zip(features.values, targets):

## Forward pass ##

# TODO: Calculate the output

hidden\_input = np.dot(x, weights\_input\_hidden)

hidden\_output = sigmoid(hidden\_input)

output = sigmoid(np.dot(hidden\_output,weights\_hidden\_output))

## Backward pass ##

# TODO: Calculate the network's prediction error

error = y - output

# TODO: Calculate error term for the output unit

output\_error\_term = error \* output \*(1-output)

## propagate errors to hidden layer

# TODO: Calculate the hidden layer's contribution to the error

hidden\_error = np.dot(output\_error\_term,weights\_hidden\_output)

# TODO: Calculate the error term for the hidden layer

hidden\_error\_term = hidden\_error \* hidden\_output \*(1-hidden\_output)

# TODO: Update the change in weights

del\_w\_hidden\_output += output\_error\_term\*hidden\_output

del\_w\_input\_hidden += hidden\_error\_term\*x[:,None]

# TODO: Update weights (don't forget to division by n\_records or number of samples)

weights\_input\_hidden += learnrate\*del\_w\_input\_hidden/n\_records

weights\_hidden\_output += learnrate\*del\_w\_hidden\_output/n\_records

# Printing out the mean square error on the training set

if e % (epochs / 10) == 0:

hidden\_output = sigmoid(np.dot(x, weights\_input\_hidden))

out = sigmoid(np.dot(hidden\_output,

weights\_hidden\_output))

loss = np.mean((out - targets) \*\* 2)

if last\_loss and last\_loss < loss:

print("Train loss: ", loss, " WARNING - Loss Increasing")

else:

print("Train loss: ", loss)

last\_loss = loss

# Calculate accuracy on test data

hidden = sigmoid(np.dot(features\_test, weights\_input\_hidden))

out = sigmoid(np.dot(hidden, weights\_hidden\_output))

predictions = out > 0.5

accuracy = np.mean(predictions == targets\_test)

print("Prediction accuracy: {:.3f}".format(accuracy))