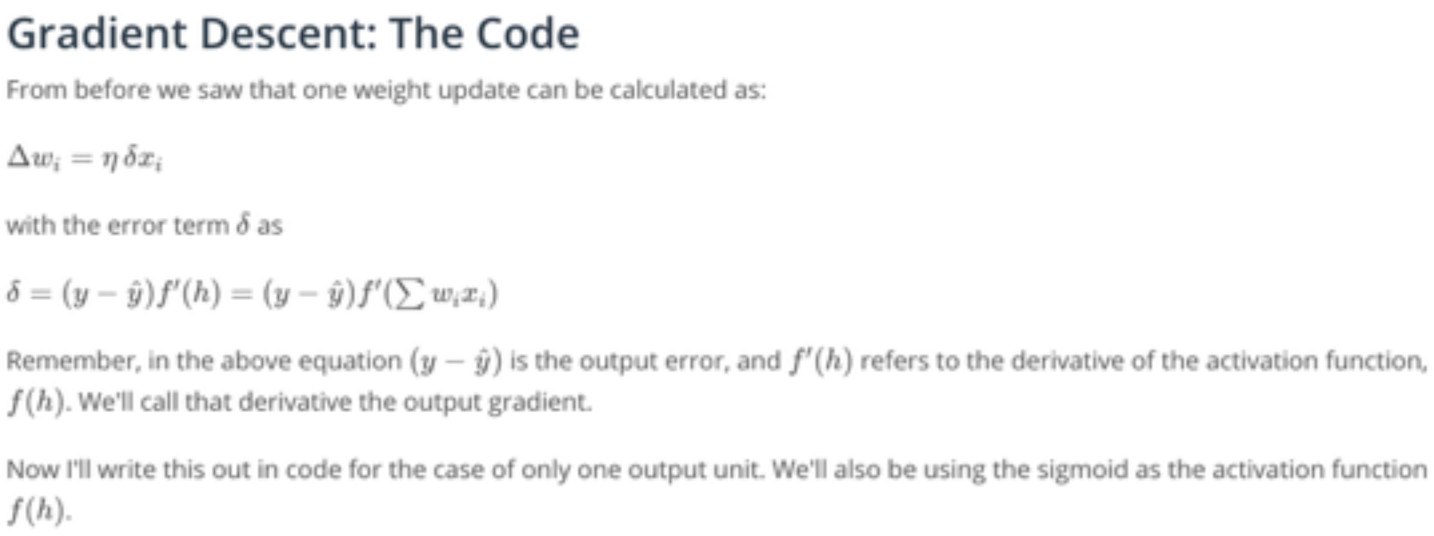
Udacity
Adam McMurchie
11 June 2017

Week 1 Neural Networks

Code



Gradient Descent: The Code

From before we saw that one weight update can be calculated as:

$\Delta w_i = \eta \, \delta x_i$

with the error term δ as

$\delta = (y - \hat{y}) f'(h) = (y - \hat{y}) f'(\sum w_i x_i)$

Remember, in the above equation $(y - \hat{y})$ is the output error, and $f'(h)$ refers to the derivative of the activation function, $f(h)$. We'll call that derivative the output gradient.

Now I'll write this out in code for the case of only one output unit. We'll also be using the sigmoid as the activation function $f(h)$.

```
# Defining the sigmoid function for activations
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Derivative of the sigmoid function
def sigmoid_prime(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Input data
x = np.array([0.1, 0.3])
# Target
y = 0.2
```

```
# Input to output weights
weights = np.array([-0.8, 0.5])

# The learning rate, eta in the weight step equation
learnrate = 0.5

# the linear combination performed by the node (h in f(h) and f'(h))
h = x[0]*weights[0] + x[1]*weights[1]
# or h = np.dot(x, weights)

# The neural network output (y-hat)
nn_output = sigmoid(h)

# output error (y - y-hat)
error = y - nn_output

# output gradient (f'(h))
output_grad = sigmoid_prime(h)

# error term (lowercase delta)
error_term = error * output_grad

# Gradient descent step
del_w = [ learnrate * error_term * x[0],
          learnrate * error_term * x[1]]
# or del_w = learnrate * error_term * x
```

Quiz answers

```
import numpy as np

def sigmoid(x):
    """
    Calculate sigmoid
    """
    return 1/(1+np.exp(-x))

def sigmoid_prime(x):
    """
    # Derivative of the sigmoid function
    """
    return sigmoid(x) * (1 - sigmoid(x))

learnrate = 0.5
x = np.array([1, 2. 3, 4])
y = np.array(0.5)

# Initial weights
w = np.array([0.5, -0.5, 0.3, 0.1])

### Calculate one gradient descent step for each weight
### Note: Some steps have been consilated, so there are
###       fewer variable names than in the above sample code

# TODO: Calculate the node's linear combination of inputs and weights
h = np.dot(x, w)
```

```
# TODO: Calculate output of neural network
nn_output = sigmoid(h)

# TODO: Calculate error of neural network
error = y - nn_output

# TODO: Calculate the error term
#       Remember, this requires the output gradient, which we haven't
#       specifically added a variable for.
error_term = error * sigmoid_prime(h)
# Note: The sigmoid_prime function calculates sigmoid(h) twice,
#       but you've already calculated it once. You can make this
#       code more efficient by calculating the derivative directly
#       rather than calling sigmoid_prime, like this:
# error_term = error * nn_output * (1 - nn_output)

# TODO: Calculate change in weights
del_w = learnrate * error_term * x

print('Neural Network output:')
print(nn_output)
print('Amount of Error:')
print(error)
print('Change in Weights:')
print(del_w)
```

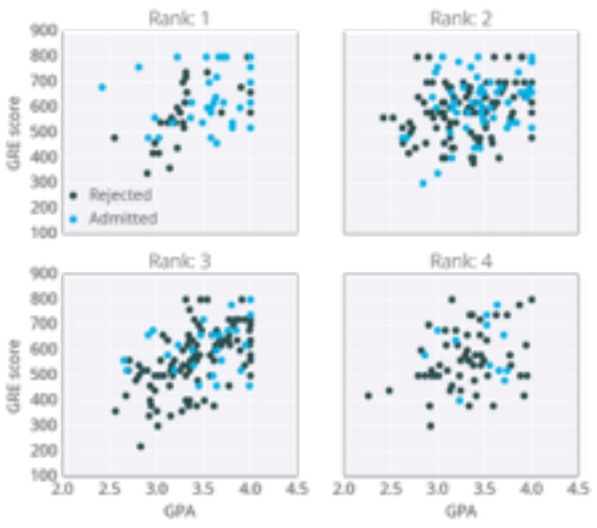
```
Neural Network output:
0.689974481128
Amount of Error:
-0.189974481128
Change in Weights:
[-0.02031869 -0.04063738 -0.06095608 -0.08127477]

Nice job!  That's right!
```

Implementing gradient descent

Okay, now we know how to update our weights:

$$\Delta w_{ij} = \eta \delta_j x_i,$$

You've seen how to implement that for a single update, but how do we translate that code to calculate many weight updates so our network will learn?



As an example, I'm going to have you use gradient descent to train a network on graduate school admissions data (found at http://www.ats.ucla.edu/stat/data/binary.csv. This dataset has three input features: GRE score, GPA, and the rank of the undergraduate school (numbered 1 through 4). Institutions with rank 1 have the highest prestige, those with rank 4 have the lowest.

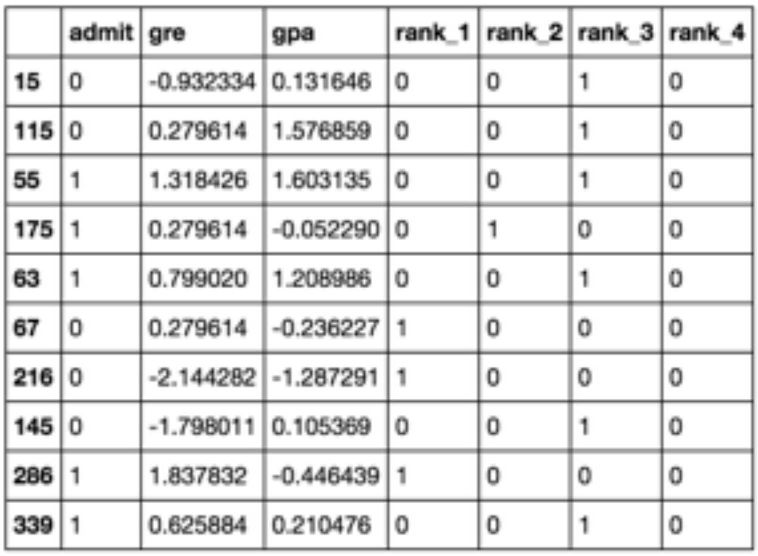
The goal here is to predict if a student will be admitted to a graduate program based on these features. For this, we'll use a network with one output layer with one unit. We'll use a sigmoid function for the output unit activation.

Data cleanup

You might think there will be three input units, but we actually need to transform the data first. The rank feature is categorical, the numbers don't encode any sort of relative values. Rank 2 is not twice as much as rank 1, rank 3 is not 1.5 more than rank 2. Instead, we need to use dummy variables to encode rank, splitting the data into four new columns encoded with ones or zeros. Rows with rank 1 have one in the rank 1 dummy column, and zeros in all other columns. Rows with rank 2 have one in the rank 2 dummy column, and zeros in all other columns. And so on.

We'll also need to standardize the GRE and GPA data, which means to scale the values such they have zero mean and a standard deviation of 1. This is necessary because the sigmoid function squashes really small and really large inputs. The gradient of really small and large inputs is zero, which means that the gradient descent step will go to zero too. Since the GRE and GPA values are fairly large, we have to be really careful about how we initialize the weights or the gradient descent steps will die off and the network won't train. Instead, if we standardize the data, we can initialize the weights easily and everyone is happy.

This is just a brief run-through, you'll learn more about preparing data later. If you're interested in how I did this, check out the data_prep.py file in the programming exercise below.



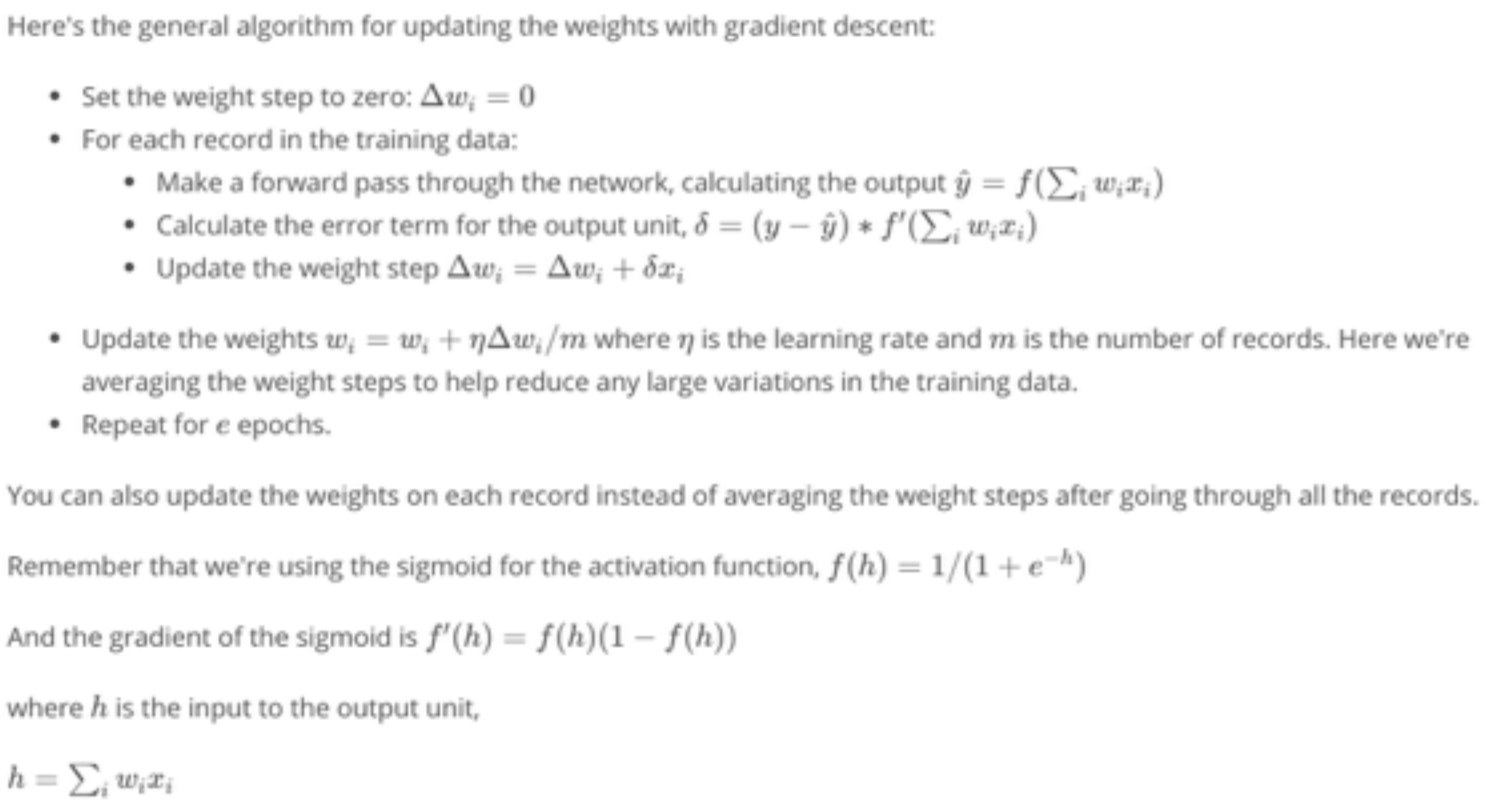
	admit	gre	gpa	rank_1	rank_2	rank_3	rank_4
15	0	-0.932334	0.131646	0	0	1	0
115	0	0.279614	1.576859	0	0	1	0
55	1	1.318426	1.603135	0	0	1	0
175	1	0.279614	-0.052290	0	1	0	0
63	1	0.799020	1.208986	0	0	1	0
67	0	0.279614	-0.236227	1	0	0	0
216	0	-2.144282	-1.287291	1	0	0	0
145	0	-1.798011	0.105369	0	0	1	0
286	1	1.837832	-0.446439	1	0	0	0
339	1	0.625884	0.210476	0	0	1	0

Now that the data is ready, we see that there are six input features: gre, gpa, and the four rank dummy variables.

Mean Square Error

We're going to make a small change to how we calculate the error here. Instead of the SSE, we're going to use the mean of the square errors (MSE). Now that we're using a lot of data, summing up all the weight steps can lead to really large updates that make the gradient descent diverge. To compensate for this, you'd need to use a quite small learning rate. Instead, we can just divide by the number of records in our data, m to take the average. This way, no matter how much data we use, our learning rates will typically be in the range of 0.01 to 0.001. Then, we can use the MSE (shown below) to calculate the gradient and the result is the same as before, just averaged instead of summed.

$$E = \frac{1}{2m}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2$$



Here's the general algorithm for updating the weights with gradient descent:

- Set the weight step to zero: $\Delta w_i = 0$
- For each record in the training data:
  - Make a forward pass through the network, calculating the output $\hat{y} = f(\sum_i w_i x_i)$
  - Calculate the error term for the output unit, $\delta = (y - \hat{y}) * f'(\sum_i w_i x_i)$
  - Update the weight step $\Delta w_i = \Delta w_i + \delta x_i$
- Update the weights $w_i = w_i + \eta \Delta w_i / m$ where η is the learning rate and m is the number of records. Here we're averaging the weight steps to help reduce any large variations in the training data.
- Repeat for e epochs.

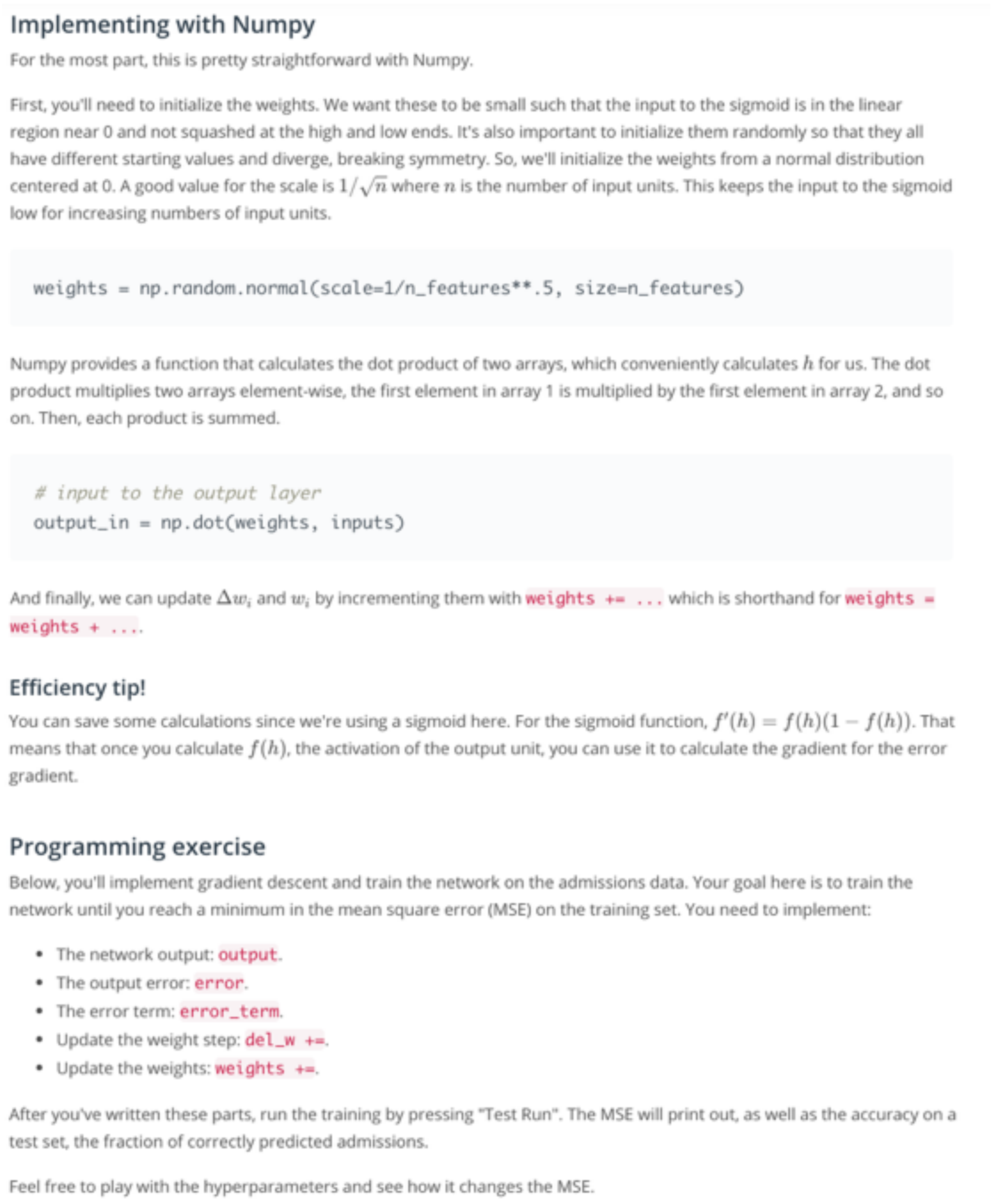
You can also update the weights on each record instead of averaging the weight steps after going through all the records.

Remember that we're using the sigmoid for the activation function, $f(h) = 1/(1 + e^{-h})$

And the gradient of the sigmoid is $f'(h) = f(h)(1 - f(h))$

where h is the input to the output unit,

$h = \sum_i w_i x_i$



Implementing with Numpy

For the most part, this is pretty straightforward with Numpy.

First, you'll need to initialize the weights. We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is $1/\sqrt{n}$ where n is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

```
weights = np.random.normal(scale=1/n_features**.5, size=n_features)
```

Numpy provides a function that calculates the dot product of two arrays, which conveniently calculates h for us. The dot product multiplies two arrays element-wise, the first element in array 1 is multiplied by the first element in array 2, and so on. Then, each product is summed.

```
# input to the output layer
output_in = np.dot(weights, inputs)
```

And finally, we can update Δw_i and w_i by incrementing them with `weights += ...` which is shorthand for `weights = weights + ...`.

Efficiency tip!

You can save some calculations since we're using a sigmoid here. For the sigmoid function, $f'(h) = f(h)(1 - f(h))$. That means that once you calculate $f(h)$, the activation of the output unit, you can use it to calculate the gradient for the error gradient.

Programming exercise

Below, you'll implement gradient descent and train the network on the admissions data. Your goal here is to train the network until you reach a minimum in the mean square error (MSE) on the training set. You need to implement:

- The network output: `output`.
- The output error: `error`.
- The error term: `error_term`.
- Update the weight step: `del_w +=`.
- Update the weights: `weights +=`.

After you've written these parts, run the training by pressing "Test Run". The MSE will print out, as well as the accuracy on a test set, the fraction of correctly predicted admissions.

Feel free to play with the hyperparameters and see how it changes the MSE.

Summary Section required

1. Set the weight step to zero: $\mathbf{\Delta w_i=0}$
2. For each record in the training data:
    1. forward pass through the network, calculating the output $\hat{y}=f(\sum_i w_i x_i)$
    2. Calculate the error term for the output unit, $\delta=(y-\hat{y})*f'(\sum_i w_i x_i)$
    3. Update the weight step $\Delta w_i=\Delta w_i+\delta x_i$
3. Update the weights $w_i=w_i+\eta\Delta w_i/m$ where η is the learning rate and m is the number of records. Here we're averaging the weight steps to help reduce any large variations in the training data.
4. Repeat for **e** epochs.

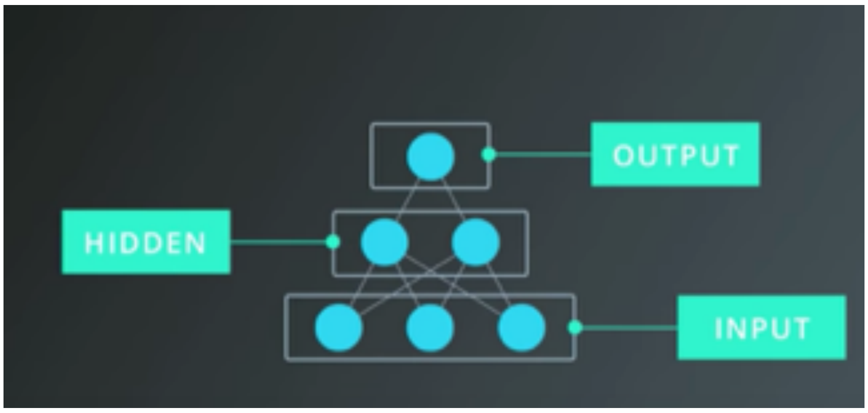
GRADIENT.PY (Full code located in deep learning neural network)

```
import numpy as np
from data_prep import features, targets, features_test, targets_test

def sigmoid(x):
    """
    Calculate sigmoid
    """
    return 1 / (1 + np.exp(-x))

# TODO: We haven't provided the sigmoid_prime function like we did in
#       the previous lesson to encourage you to come up with a more
#       efficient solution. If you need a hint, check out the comments
#       in solution.py from the previous lecture.

# Use to same seed to make debugging easier
```

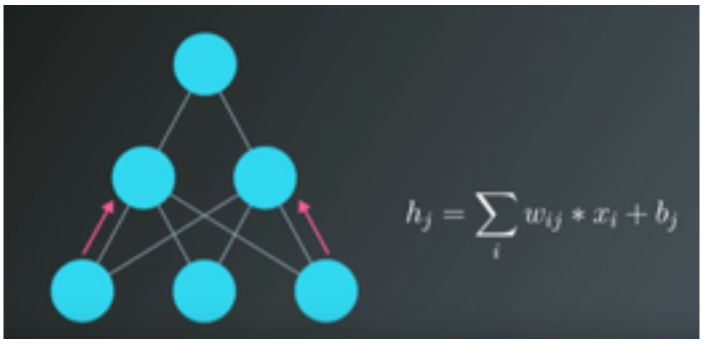
```
np.random.seed(42)

n_records, n_features = features.shape
last_loss = None

# Initialize weights (1/sqrt(n_features), size = n_features)
weights = np.random.normal(scale=1 / n_features**.5, size=n_features)

# Neural Network hyperparameters
epochs = 1000
learnrate = 0.5

for e in range(epochs):
    del_w = np.zeros(weights.shape)
    for x, y in zip(features.values, targets):
        # Loop through all records, x is the input, y is the target
        # Activation of the output unit
        #   Notice we multiply the inputs and the weights here
        #   rather than storing h as a separate variable
        output = sigmoid(np.dot(x, weights))

        # The error, the target minus the network output
        error = y - output

        # The error term
        #   Notice we calulate f'(h) here instead of defining a separate
        #   sigmoid_prime function. This just makes it faster because we
        #   can re-use the result of the sigmoid function stored in
        #   the output variable
        error_term = error * output * (1 - output)

        # The gradient descent step, the error times the gradient times the
inputs
        del_w += error_term * x

    # Update the weights here. The learning rate times the
    # change in weights, divided by the number of records to average
    weights += learnrate * del_w / n_records

    # Printing out the mean square error on the training set
    if e % (epochs / 10) == 0:
        out = sigmoid(np.dot(features, weights))
        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
tes_out = sigmoid(np.dot(features_test, weights))
predictions = tes_out > 0.5
accuracy = np.mean(predictions == targets_test)
print("Prediction accuracy: {:.3f}”.format(accuracy))
```

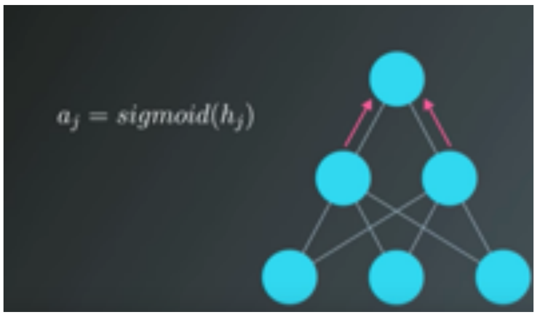
Implementing the hidden layer

We saw that adding a second layer of units allows the model to find solutions to linearly inseparable problems.



Calculating the output is the same as before, but now the activations of the hidden layer is the input to the output layer.





Input remains the same , again activation function like a sigmoid is used to calculate the output of the hidden layer. The hidden layer activations are passed to the output layer through the second set of weights and again use an activation function to get the output of the network

$$o_k = sigmoid(\sum_j w_j k * a_j + b_j)$$

Stacking more layers lets the network learn more complex patterns .

This is deep learning (more layers)

Implementing the hidden layer

Prerequisites

Below, we are going to walk through the math of neural networks in a multilayer perceptron. With multiple perceptrons, we are going to move to using vectors and matrices. To brush up, be sure to view the following:

Khan Academy's introduction to vectors.

Khan Academy's introduction to matrices.

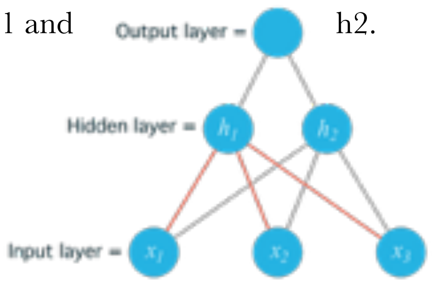
https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/vectors/v/vector-introduction-linear-algebra

https://www.khanacademy.org/math/precalculus/precalc-matrices

Derivation

Before, we were dealing with only one output node which made the code straightforward. However now that we have multiple input units and multiple hidden units, the weights between them will require two indices: wij where i denotes input units and j are the hidden units.

For example, the following image shows our network, with its input units labeled x1,x2, and x3, and its hidden nodes labeled h1 and h2.



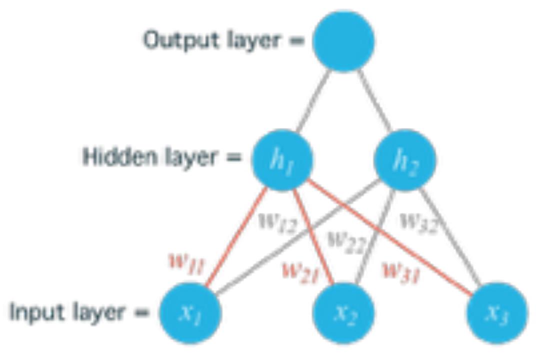
The lines indicating the weights leading to h1 have been colored differently from those leading to h2 just to make it easier to read.

Now to index the weights, we take the input unit number for the i and the hidden unit number for the j That gives us

w11 for the weight leading from x1 to h1

w12 for the weight leading from x1 to h2

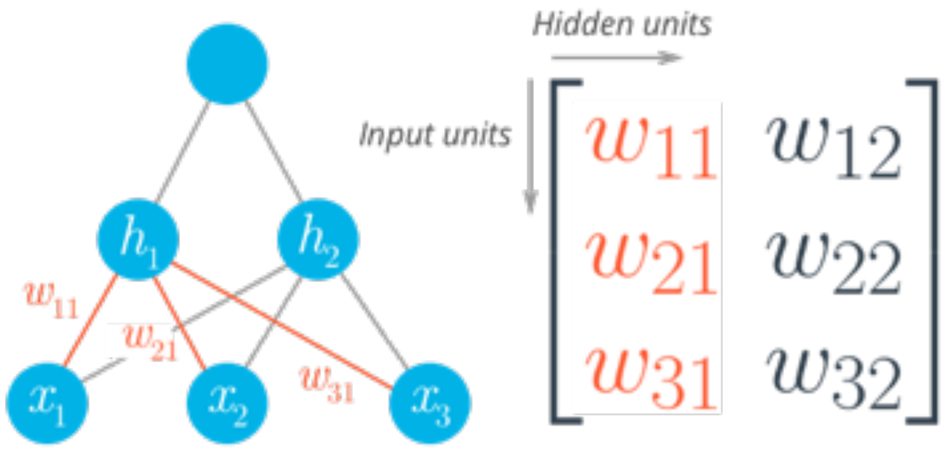
The following image includes all of the weights between the input layer and the hidden layer, labeled with their appropriate wij indices:



Before, we were able to write the weights as an array, indexed as wi.

But now, the weights need to be stored in a matrix, indexed as wij.

Each row in the matrix will correspond to the **weights** leading out of a **single input** unit, and each column will correspond to the weights **leading in** to a single **hidden unit**. For our three input units and two hidden units, the weights matrix looks like this:


Hidden units
Input units
h_1
h_2
x_1
x_2
x_3
w_{11}
w_{21}
w_{31}
$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$

Be sure to compare the matrix above with the diagram shown before it so you can see where the different weights in the network end up in the matrix.

To initialize these weights in Numpy, we have to provide the shape of the matrix. If features is a 2D array containing the input data:

```
# Number of records and input units
n_records, n_inputs = features.shape
# Number of hidden units
n_hidden = 2
weights_input_to_hidden = np.random.normal(0, n_inputs**-0.5, size=(n_inputs, n_hidden))
```

This creates a 2D array (i.e. a matrix) named weights_input_to_hidden with dimensions

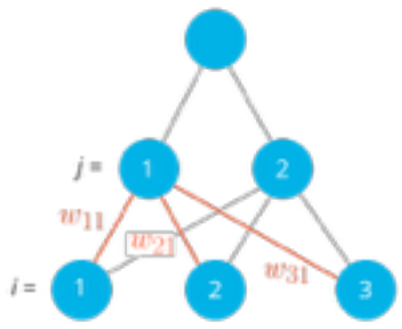
$$h_j = \sum_i w_{ij} x_i$$

n_inputs by n_hidden. Remember how the input to a hidden unit is the sum of all the inputs multiplied by the hidden unit's weights. So for each hidden layer unit, hj, we need to calculate the following:

To do that, we now need to use matrix multiplication. If your linear algebra is rusty, I suggest taking a look at the suggested resources in the prerequisites section. For this part though, you'll only need to know how to multiply a matrix with a vector.

https://en.wikipedia.org/wiki/Matrix_multiplication

In this case, we're multiplying the inputs (a row vector here) by the weights. To do this, you take the dot (inner) product of the inputs with each column in the weights matrix. For example, to calculate the input to the first hidden unit, j=1, you'd take the dot product of the inputs with the first column of the weights matrix, like so:



$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} \boxed{w_{11}} & w_{12} \\ \boxed{w_{21}} & w_{22} \\ \boxed{w_{31}} & w_{32} \end{bmatrix}$$

Calculating the input of the first hidden unit with the first column of the weight matrix

$$h_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31}$$

And for the second hidden layer input, you calculate the dot product of the inputs with the second column. And so on and so forth.

In Numpy, you can do this for all the inputs and all the outputs at once using np.dot

```
hidden_inputs = np.dot(inputs, weights_input_to_hidden)
```

You could also define your weights matrix such that it has dimensions n_hidden by n_inputs then multiply like so where the inputs form a column vector:

$$h_j = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Note: The weight indices have changed in the above image and no longer match up with the labels used in the earlier diagrams. That's because, in matrix notation, the row index always precedes the column index, so it would be misleading to label them the way we did in the neural net diagram. Just keep in mind that this is the same weight matrix as before, but rotated so the first column is now the first row, and the second column is now the second row. If we were to use the labels from the earlier diagram, the weights would fit into the matrix in the following locations:

$$\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}$$

Weight matrix shown with labels matching earlier diagrams.

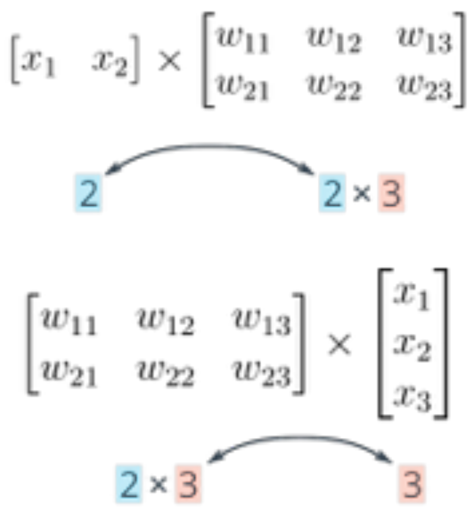
Remember, the above is not a correct view of the indices, but it uses the labels from the earlier neural net diagrams to show you where each weight ends up in the matrix.

The important thing with matrix multiplication is that the dimensions match. For matrix multiplication to work, there has to be the same number of elements in the dot

```
# Same weights and features as above, but swapped the order
hidden_inputs = np.dot(weights_input_to_hidden, features)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-1bfa0f615c45> in <module>()
----> 1 hidden_in = np.dot(weights_input_to_hidden, X)

ValueError: shapes (3,2) and (3,) not aligned: 2 (dim 1) != 3 (dim 0)
```

products. In the first example, there are three columns in the input vector, and three rows in the weights matrix. In the second example, there are three columns in the weights matrix and three rows in the input vector. If the dimensions don't match, you'll get this above.

The dot product can't be computed for a 3x2 matrix and 3-element array. That's because the 2 columns in the matrix don't match the number of elements in the array. Some of the dimensions that could work would be the following:

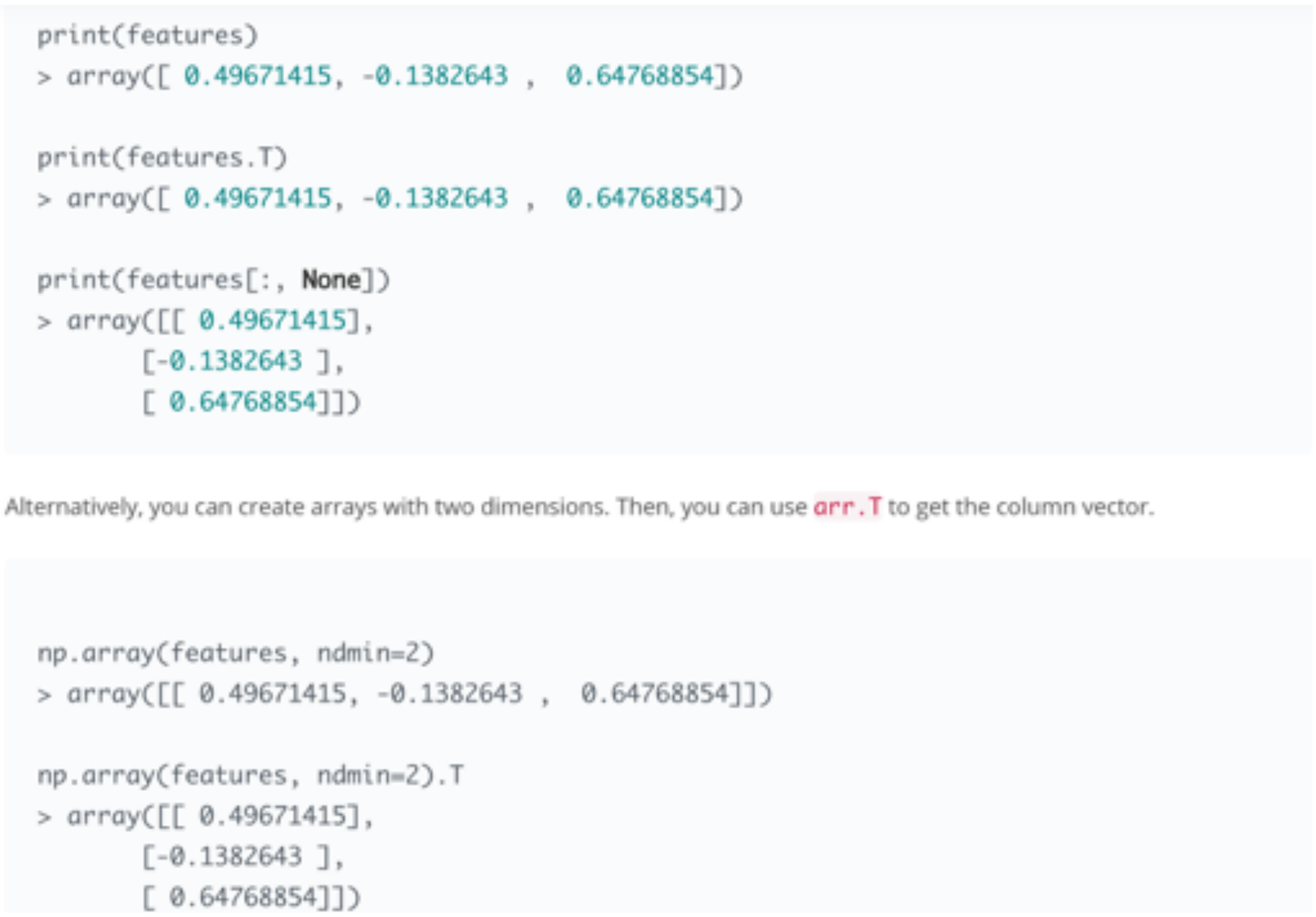


$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$
2
2 × 3
$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$
2 × 3
3

The rule is that if you're multiplying an array from the left, the array must have the same number of elements as there are rows in the matrix. And if you're multiplying the matrix from the left, the number of columns in the matrix must equal the number of elements in the array on the right.

Making a column vector

You see above that sometimes you'll want a column vector, even though by default Numpy arrays work like row vectors. It's possible to get the transpose of an array like so arr.T, but for a 1D array, the transpose will return a row vector. Instead, use arr[:,None] to create a column vector:



```
print(features)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features.T)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features[:, None])
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

Alternatively, you can create arrays with two dimensions. Then, you can use `arr.T` to get the column vector.

```
np.array(features, ndmin=2)
> array([[ 0.49671415, -0.1382643 ,  0.64768854]])

np.array(features, ndmin=2).T
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

Programming quiz

Below, you'll implement a forward pass through a 4x3x2 network, with sigmoid activation functions for both layers.

Things to do:

Calculate the input to the hidden layer.

Calculate the hidden layer output.

Calculate the input to the output layer.

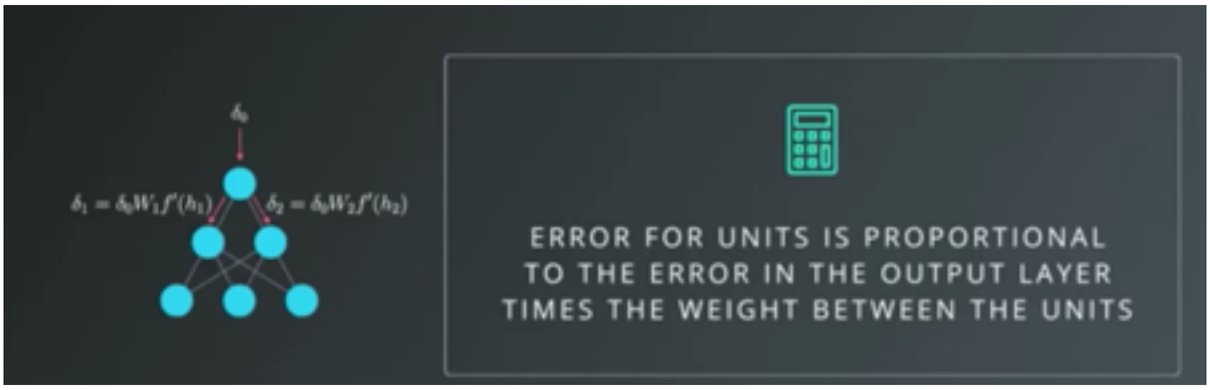
Calculate the output of the network.

```
import numpy as np

def sigmoid(x):
    """
    Calculate sigmoid
    """
    return 1/(1+np.exp(-x))

# Network size
N_input = 4
N_hidden = 3
N_output = 2

np.random.seed(42)
# Make some fake data
X = np.random.randn(4)

weights_input_to_hidden = np.random.normal(0, scale=0.1, size=(N_input, N_hidden))
weights_hidden_to_output = np.random.normal(0, scale=0.1, size=(N_hidden, N_output))


# TODO: Make a forward pass through the network

hidden_layer_in = np.dot(X, weights_input_to_hidden)
hidden_layer_out = sigmoid(hidden_layer_in)

print('Hidden-layer Output:')
print(hidden_layer_out)

output_layer_in = np.dot(hidden_layer_out, weights_hidden_to_output)
output_layer_out = sigmoid(output_layer_in)

print('Output-layer Output:')
print(output_layer_out)
```

```
Hidden-layer Output:
[ 0.41492192  0.42604313  0.5002434 ]
Output-layer Output:
[ 0.49815196  0.48539772]

Nice job!  That's right!
```

Backpropagation

We still want to use the errors to calculate the gradient


δ_o
$\delta_1 = \delta_o W_1 f'(h_1)$
$\delta_2 = \delta_o W_2 f'(h_2)$

ERROR FOR UNITS IS PROPORTIONAL TO THE ERROR IN THE OUTPUT LAYER TIMES THE WEIGHT BETWEEN THE UNITS

This makes sense, the unit with the stronger connection to the output node will contribute more error towards the output.

here you see the error times the weights - this is the same way you propagate inputs through the network, the inputs times the weights between the layers. Instead of propagating error forwards ,we propagate backwards thru the network. its like flipping it over and using error as input.

Back propagation

Now we've come to the problem of how to make a multilayer neural network learn. Before, we saw how to update weights with gradient descent. The backpropagation algorithm is just an extension of that, using the chain rule to find the error with the respect to the weights connecting the input layer to the hidden layer (for a two layer network).

To update the weights to hidden layers using gradient descent, you need to know how much error each of the hidden units contributed to the final output. Since the output of a layer is determined by the weights between layers, the error resulting from units is scaled by the weights going forward through the network. Since we know the error at the output, we can use the weights to work backwards to hidden layers.

For example, in the output layer, you have errors δko attributed to each output unit k. Then, the error attributed to hidden unit j is the output errors, scaled by the weights between the output and hidden layers (and the gradient):

$$\delta_j^h = \sum W_{jk} \delta_k^o f'(h_j)$$

Then, the gradient descent step is the same as before, just with the new errors:

$$\Delta w_{ij} = \eta \delta_j^h x_i$$

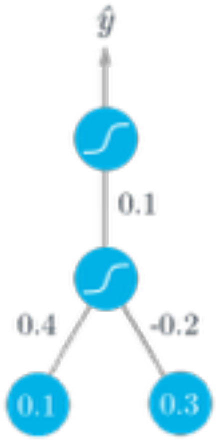
where w_{ij} are the weights between the inputs and hidden layer and x_i are input unit values. This form holds for however many layers there are. The weight steps are equal to the step size times the output error of the layer times the values of the inputs to that layer

$$\Delta w_{pq} = \eta \delta_{output} V_{in}$$

Here, you get the output error, δ_{output}, by propagating the errors backwards from higher layers. And the input values, V_{in} are the inputs to the layer, the hidden layer activations to the output unit for example.

Working through an example

Let's walk through the steps of calculating the weight updates for a simple two layer network. Suppose there are two input values, one hidden unit, and one output unit, with sigmoid activations on the hidden and output units. The following image depicts this network. (Note: the input values are shown as nodes at the bottom of the image, while the networks output value is shown as y^ at the top. The inputs themselves do not count as a layer, which is why this is considered a two layer network.)



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

$$h = \sum_i w_i x_i = 0.1 \times 0.4 - 0.2 \times 0.3 = -0.02$$

and the output of the hidden unit

$$a = f(h) = \text{sigmoid}(-0.02) = 0.495.$$

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

$$\hat{y} = f(W \cdot a) = \text{sigmoid}(0.1 \times 0.495) = 0.512.$$

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 $f'(W \cdot a) = f(W \cdot a)(1 - f(W \cdot a))$, the error term for the output unit is

$\delta^o = (y - \hat{y})f'(W \cdot a) = (1 - 0.512) \times 0.512 \times (1 - 0.512) = 0.122.$

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, $\delta^h_j = \sum_k W_{jk} \delta^o_k f'(h_j)$, but since we have one hidden unit and one output unit, this is much simpler.

$\delta^h = W \delta^o f'(h) = 0.1 \times 0.122 \times 0.495 \times (1 - 0.495) = 0.003$

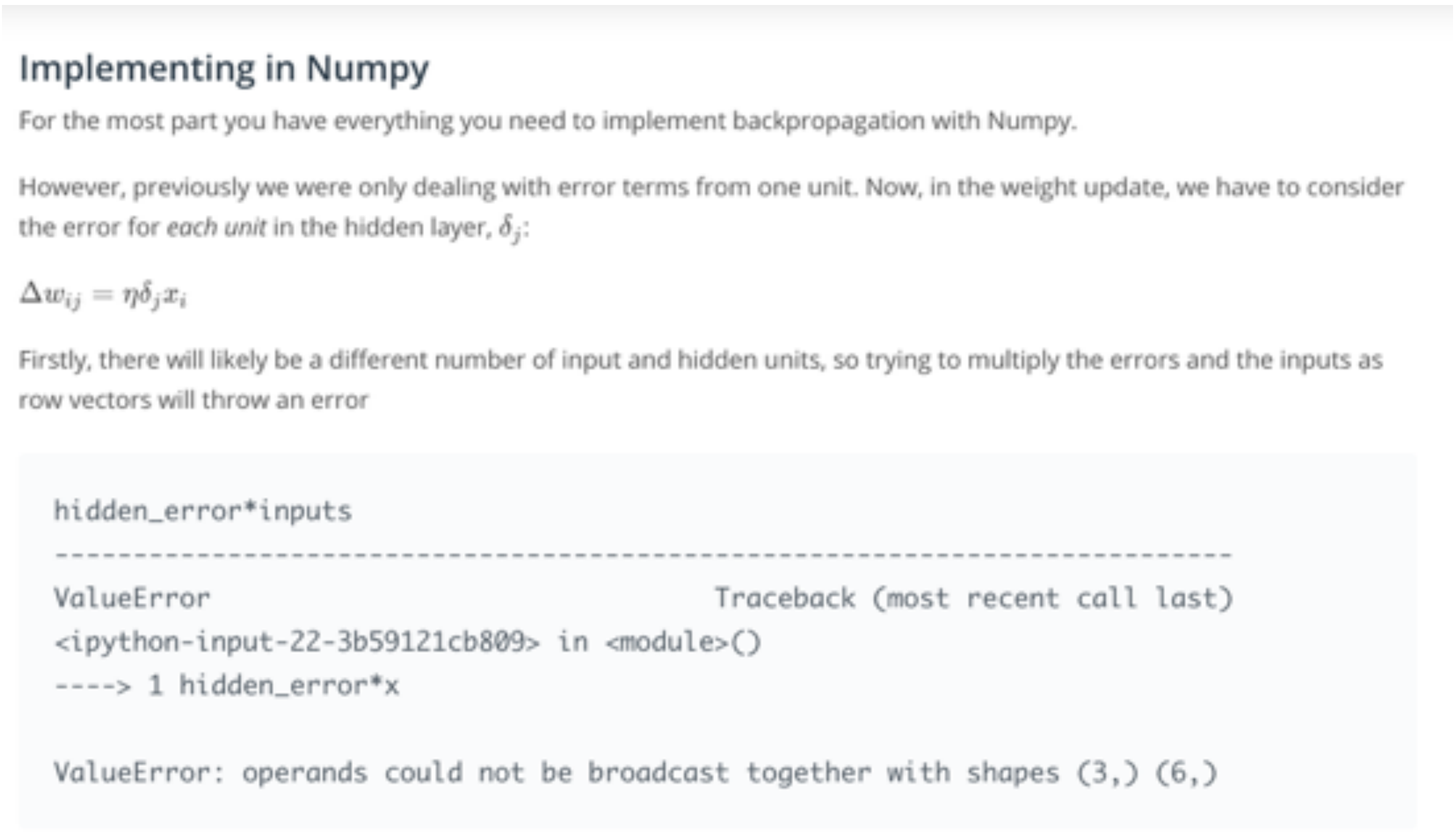
Now that we have the errors, we can calculate the gradient descent steps. The hidden to output weight step is the learning rate, times the output unit error, times the hidden unit activation value.

$\Delta W = \eta \delta^o a = 0.5 \times 0.122 \times 0.495 = 0.0302$

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

$\Delta w_i = \eta \delta^h x_i = (0.5 \times 0.003 \times 0.1, 0.5 \times 0.003 \times 0.3) = (0.00015, 0.00045)$

From this example, you can see one of the effects of using the sigmoid function for the activations. The maximum derivative of the sigmoid function is 0.25, so the errors in the output layer get reduced by at least 75%, and errors in the hidden layer are scaled down by at least 93.75%! You can see that if you have a lot of layers, using a sigmoid activation function will quickly reduce the weight steps to tiny values in layers near the input. This is known as the vanishing gradient problem. Later in the course you'll learn about other activation functions that perform better in this regard and are more commonly used in modern network architectures.



Implementing in Numpy

For the most part you have everything you need to implement backpropagation with Numpy.

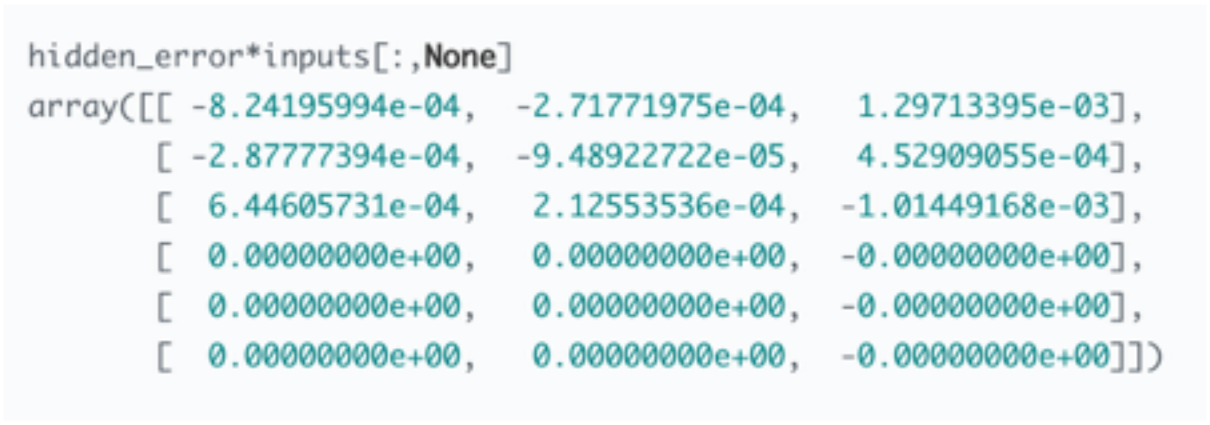
However, previously we were only dealing with error terms from one unit. Now, in the weight update, we have to consider the error for *each unit* in the hidden layer, δ_j:

$\Delta w_{ij} = \eta \delta_j x_i$

Firstly, there will likely be a different number of input and hidden units, so trying to multiply the errors and the inputs as row vectors will throw an error

```
hidden_error*inputs
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-3b59121cb809> in <module>()
----> 1 hidden_error*x

ValueError: operands could not be broadcast together with shapes (3,) (6,)
```

Also, wij is a matrix now, so the right side of the assignment must have the same shape as the left side. Luckily, Numpy takes care of this for us. If you multiply a row vector array with a column vector array, it will multiply the first element in the column by each element in the row vector and set that as the first row in a new 2D array. This continues for each element in the column vector, so you get a 2D array that has shape (len(column_vector), len(row_vector)).



```
hidden_error*inputs[:,None]
array([[ -8.24195994e-04,  -2.71771975e-04,   1.29713395e-03],
       [ -2.87777394e-04,  -9.48922722e-05,   4.52909055e-04],
       [  6.44605731e-04,   2.12553536e-04,  -1.01449168e-03],
       [  0.00000000e+00,   0.00000000e+00,  -0.00000000e+00],
       [  0.00000000e+00,   0.00000000e+00,  -0.00000000e+00],
       [  0.00000000e+00,   0.00000000e+00,  -0.00000000e+00]])
```

It turns out this is exactly how we want to calculate the weight update step. As before, if you have your inputs as a 2D array with one row, you can also do hidden_error*inputs.T, but that won't work if inputs is a 1D array.

Backpropagation exercise

Below, you'll implement the code to calculate one backpropagation update step for two sets of weights. I wrote the forward pass, your goal is to code the backward pass.

Things to do

Calculate the network's output error.

Calculate the output layer's error term.

Use backpropagation to calculate the hidden layer's error term.

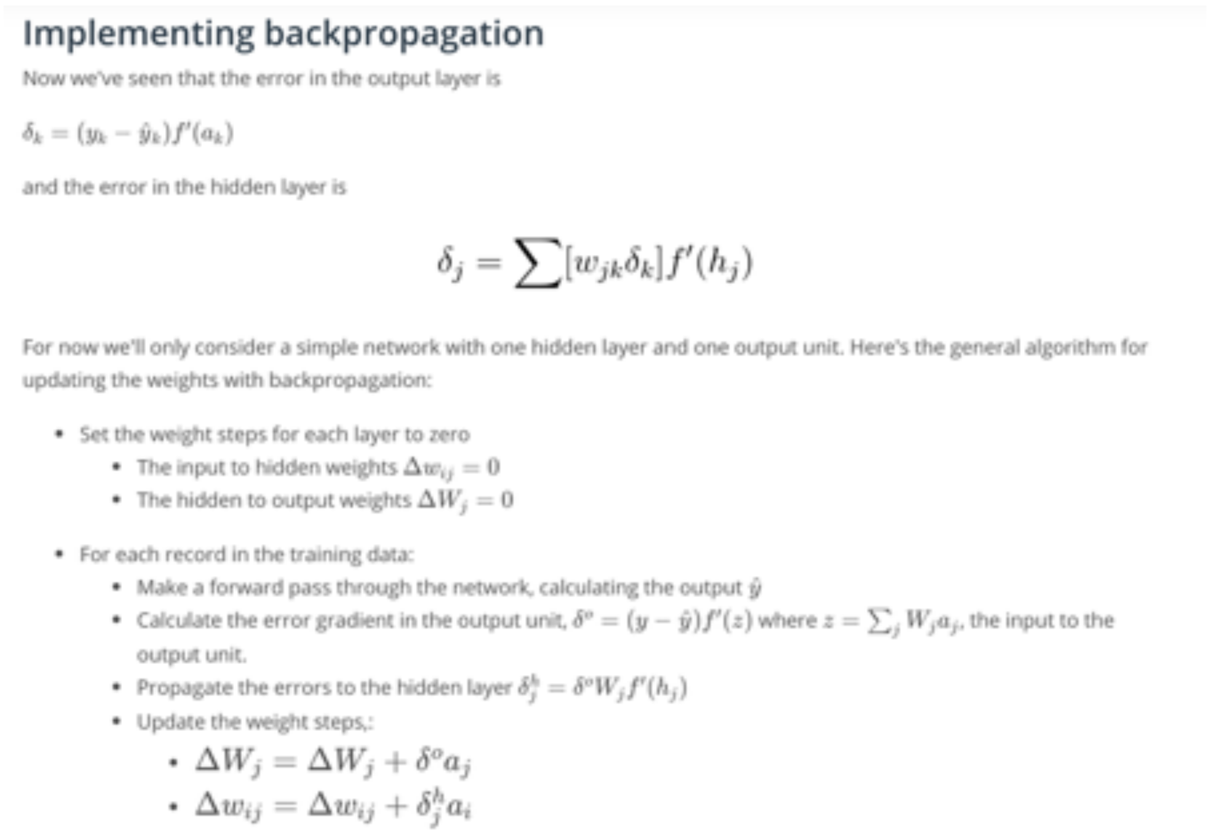
Calculate the change in weights (the delta weights) that result from propagating the errors back through the network.

fsdf

```
import numpy as np



def sigmoid(x):
    """
    Calculate sigmoid
    """
    return 1 / (1 + np.exp(-x))



x = np.array([0.5, 0.1, -0.2])
target = 0.6
learnrate = 0.5

weights_input_hidden = np.array([[0.5, -0.6],
                                 [0.1, -0.2],
                                 [0.1, 0.7]])

weights_hidden_output = np.array([0.1, -0.3])

## Forward pass
hidden_layer_input = np.dot(x, weights_input_hidden)
hidden_layer_output = sigmoid(hidden_layer_input)

output_layer_in = np.dot(hidden_layer_output, weights_hidden_output)
output = sigmoid(output_layer_in)

## Backwards pass
## TODO: Calculate output error
error = target - output

# TODO: Calculate error term for output layer
```

```
output_error_term = error * output * (1 - output)

# TODO: Calculate error term for hidden layer
hidden_error_term = np.dot(output_error_term, weights_hidden_output) * \
                    hidden_layer_output * (1 - hidden_layer_output)

# TODO: Calculate change in weights for hidden layer to output layer
delta_w_h_o = learnrate * output_error_term * hidden_layer_output

# TODO: Calculate change in weights for input layer to hidden layer
delta_w_i_h = learnrate * hidden_error_term * x[:, None]

print('Change in weights for hidden layer to output layer:')
print(delta_w_h_o)
print('Change in weights for input layer to hidden layer:')
print(delta_w_i_h)
```

Implementing Backpropagation



Implementing backpropagation

Now we've seen that the error in the output layer is

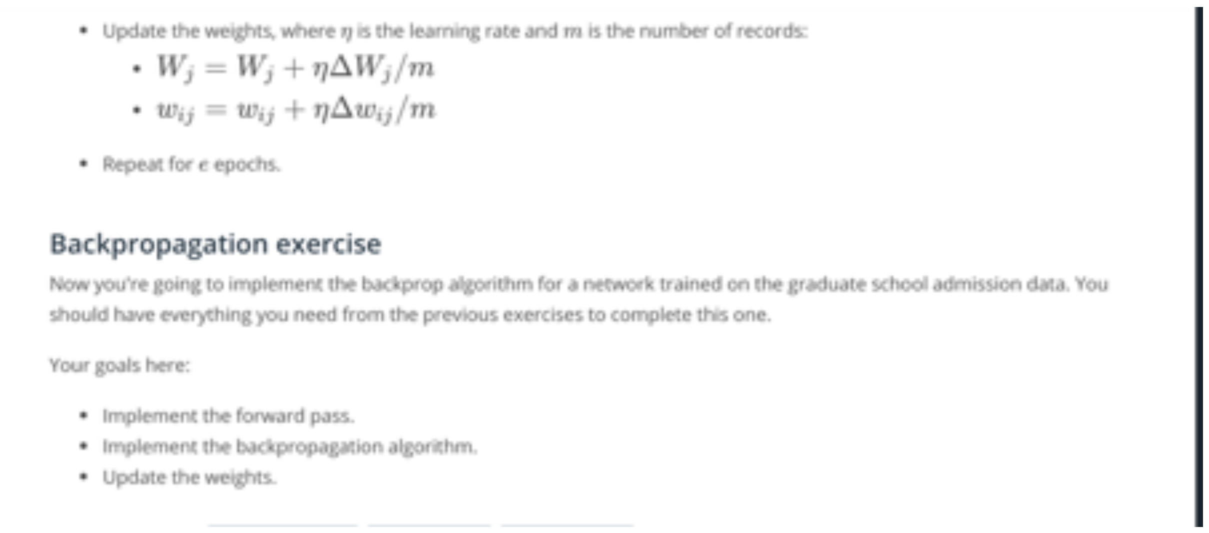
$\delta_k = (y_k - \hat{y}_k) f'(a_k)$

and the error in the hidden layer is

$$\delta_j = \sum [w_{jk} \delta_k] f'(h_j)$$

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 $\Delta w_{ij} = 0$
  - The hidden to output weights $\Delta W_j = 0$
- For each record in the training data:
  - Make a forward pass through the network, calculating the output $\hat{y}$
  - Calculate the error gradient in the output unit, $\delta^o = (y - \hat{y}) f'(z)$ where $z = \sum_j W_j a_j$, the input to the output unit.
  - Propagate the errors to the hidden layer $\delta_j^h = \delta^o W_j f'(h_j)$
  - Update the weight steps,:
    - $\Delta W_j = \Delta W_j + \delta^o a_j$
    - $\Delta w_{ij} = \Delta w_{ij} + \delta_j^h a_i$



- Update the weights, where η is the learning rate and m is the number of records:
  - $W_j = W_j + \eta \Delta W_j / m$
  - $w_{ij} = w_{ij} + \eta \Delta w_{ij} / m$
- Repeat for e epochs.

Backpropagation exercise

Now you're going to implement the backprop algorithm for a network trained on the graduate school admission data. You should have everything you need from the previous exercises to complete this one.

Your goals here:

- Implement the forward pass.
- Implement the backpropagation algorithm.
- Update the weights.

Challenge code in back prop folder, github neural networks (or at end)

Additional Reading

Backpropagation is fundamental to deep learning. TensorFlow and other libraries will perform the backprop for you, but you should really really understand the algorithm. We'll be going over backprop again, but here are some extra resources for you:

From Andrej Karpathy: Yes, you should understand backprop

Also from Andrej Karpathy, a lecture from Stanford's CS231n course

https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

https://www.youtube.com/watch?v=59Hbtz7XgjM

```
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
    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))
```