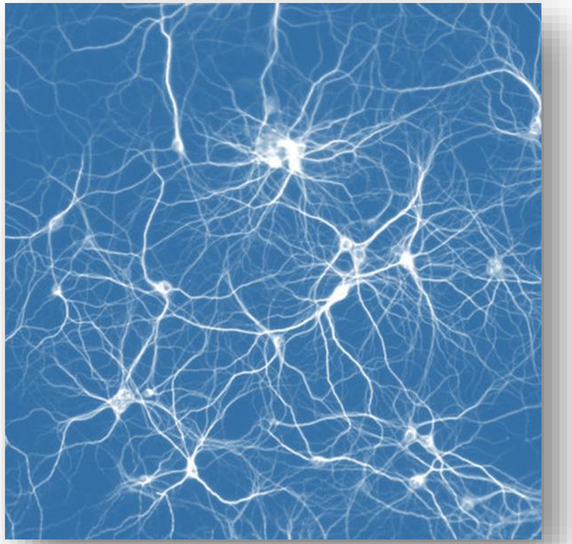
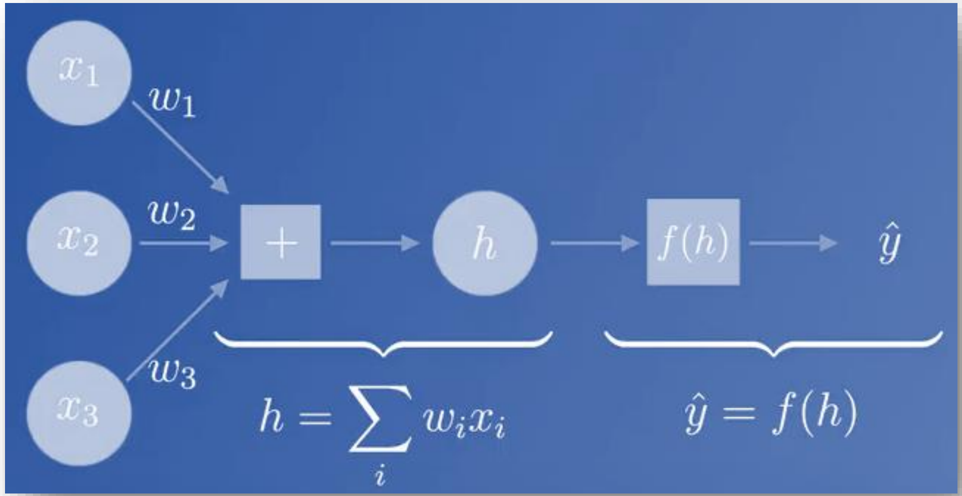
Introduction to Neural Networks-2

20210330



Gradient Descent[Single Unit]: The Math



$E = (y - \hat{y})$

loss/residual

$E = (y - \hat{y})^2$

$E = \sum_u (y^u - \hat{y}^u)^2$

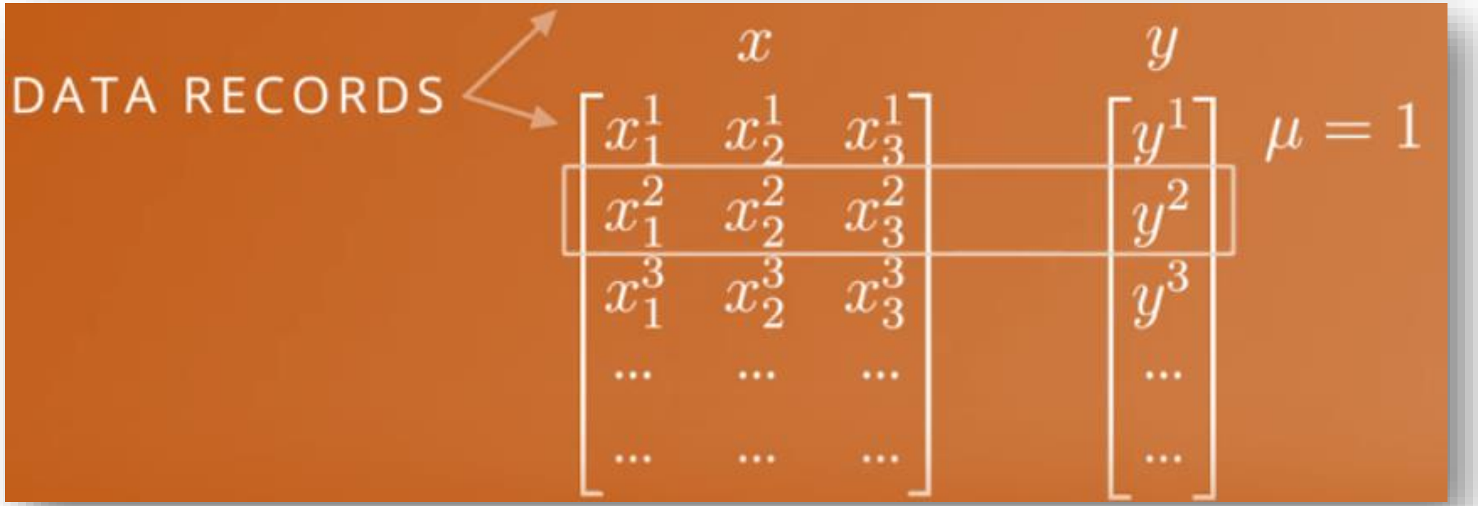
The error depends the weights w_i & input value x_i

$E = \frac{1}{2} \sum_u (y^u - \hat{y}^u)^2$

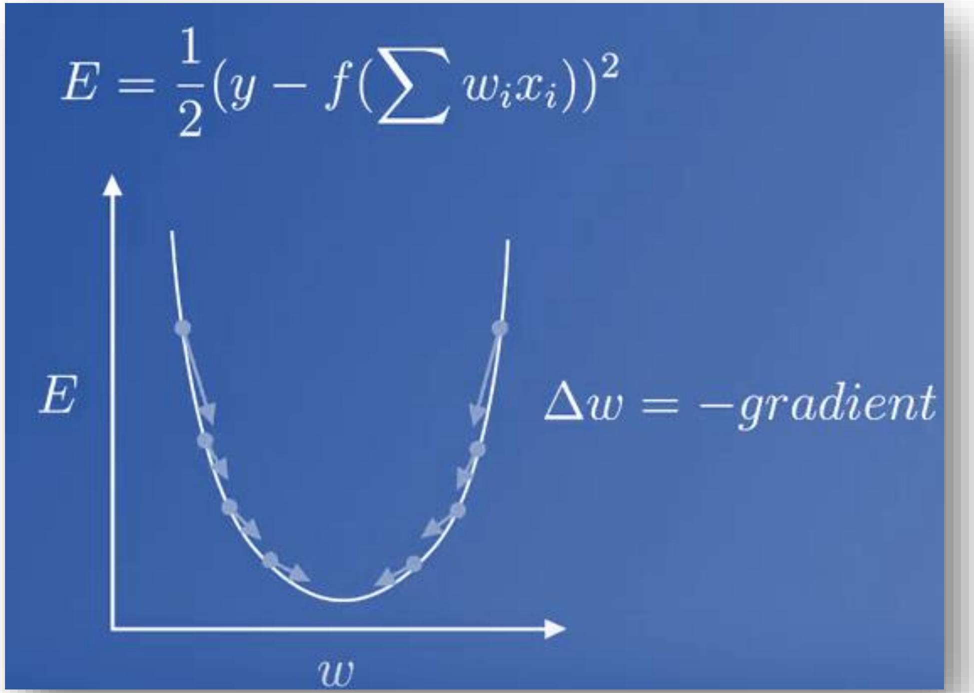
The SUM of The SQUARED ERRORS(SSE)

$E = \frac{1}{2} \sum_u (y^u - \hat{y}^u)^2$

$= \frac{1}{2} \sum_u (y^u - f(\sum_i w_i x_i^u))^2$



Gradient Descent[Single Unit]:The Math



$w_i = w_i + \Delta w_i$ ← **Update the weight**

$\Delta w_i \propto -\frac{\partial E}{\partial w_i}$ ---→ ***The Gradient***

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Learning Rate

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}(y-\hat{y})^2$$

$$=\frac{1}{2}\frac{\partial}{\partial w_i}(y-\hat{y})^2$$

$$=(y-\hat{y})\frac{\partial}{\partial w_i}(y-\hat{y})$$

$$= -(y-\hat{y})\frac{\partial \hat{y}}{\partial w_i}$$

$$\hat{y} = f(x) \text{ where } h = \sum_i w_i x_i$$

$$= -(y-\hat{y})f'(h)\frac{\partial}{\partial w_i}\sum_i w_i x_i$$

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

$$\Delta w_i = -\eta(-(y-\hat{y})f'(h)x_i)$$

(Chain rule)

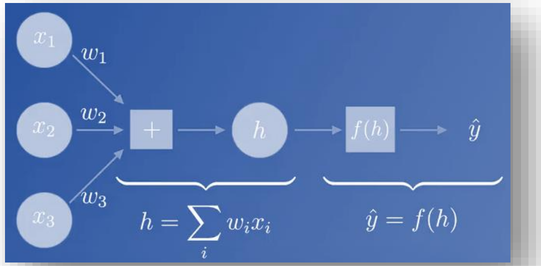
$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial (y-\hat{y})}\frac{\partial (y-\hat{y})}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial H}\frac{\partial H}{\partial w_i}$$

$\Delta w_i = \eta \delta x_i$ $(\eta \textit{ Learning Rate})$

with the error term δ as

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

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



Gradient Descent[Single Unit]: The Code

From before we saw that one weight update can be calculated as: $\Delta w_i = \eta \delta x_i$ $(\eta \ Learning \ Rate)$

with the error term δ as: $\delta = (y - \hat{y}) f'(h) = (y - \hat{y}) f'(\sum w_i x_i)$

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))
```

$$\frac{ds(x)}{dx} = \frac{1}{1+e^{-x}}$$

$$= \left(\frac{1}{1+e^{-x}}\right)^2 \frac{d}{dx}(1+e^{-x})$$

$$= \left(\frac{1}{1+e^{-x}}\right)^2 e^{-x}(-1)$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right)(-e^{-x})$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{-e^{-x}}{1+e^{-x}}\right)$$

$$= s(x)(1 - s(x))$$

```
# 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 neural network output (y-hat)
nn_output = sigmoid(x[0]*weights[0] +
x[1]*weights[1])
```

```
# or nn_output = sigmoid(np.dot(x, weights))
# output error (y - y-hat)
error = y - nn_output
# error term (lowercase delta)
error_term = error * sigmoid_prime(np.dot(x,weights))
# Gradient descent step
del_w = [ learnrate * error_term * x[0],
          learnrate * error_term * x[1]]

# or del_w = learnrate * error_term * x
```

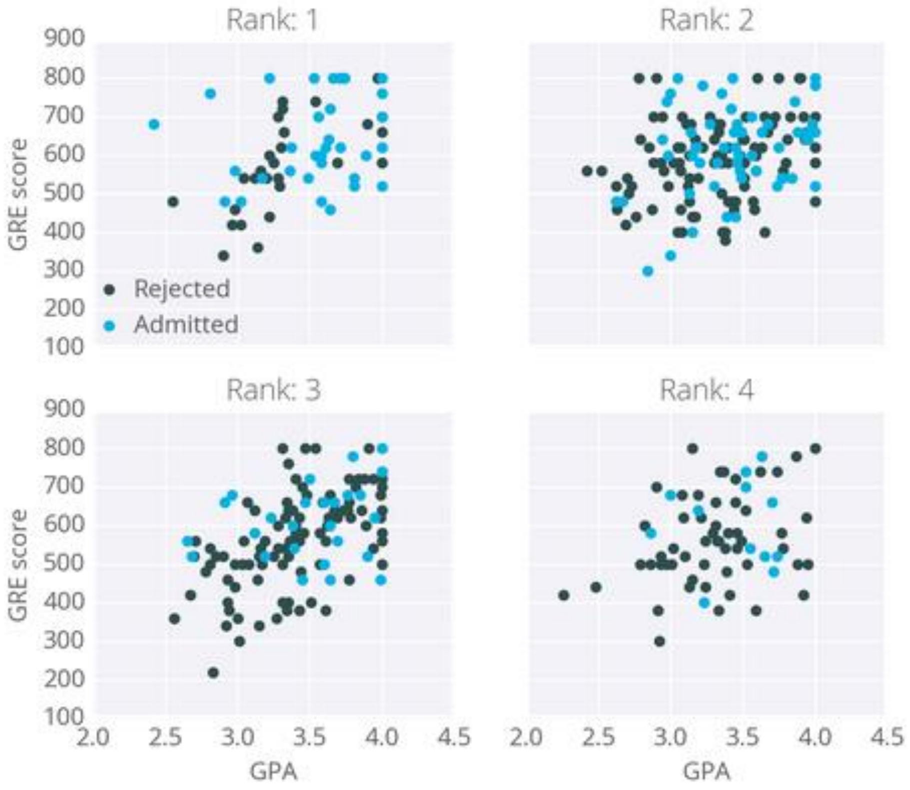
Code: 04Gradient_Descent_the_code.txt

ANSWER: 04Gradient_Descent_the_code_ANSWER.txt

Implementing Gradient Descent

Okay, now we know how to update our weights $\Delta w_{ij} = \eta \delta_j x_i$

How do we translate this into code? As an example, I'm going to have you use gradient descent to train a network on graduate school admissions data (found at https://stats.idre.ucla.edu/stat/data/binary.csv).



This dataset has three input features: GRE score, GPA, and the rank of the undergraduate school (1~4).

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.

Implementing Gradient Descent

05binary.txt - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

```
admit,gre,gpa,rank
0,380,3.61,3
1,660,3.67,3
1,800,4,1
1,640,3.19,4
0,520,2.93,4
1,760,3,2
1,560,2.98,1
0,400,3.08,2
1,540,3.39,3
0,700,3.92,2
0,800,4,4
```

Data cleanup

	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

Ten rows of the data after transformations.

We need to use dummy variables to encode rank.

We'll also need to standardize the GRE and GPA data.

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

Implementing Gradient Descent

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 gradient in 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,

$$f(h) = 1/(1 + e^{-h})$$

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 Gradient Descent

Implementing 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.

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

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.

Implementing Gradient Descent

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 error gradient: error.

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.

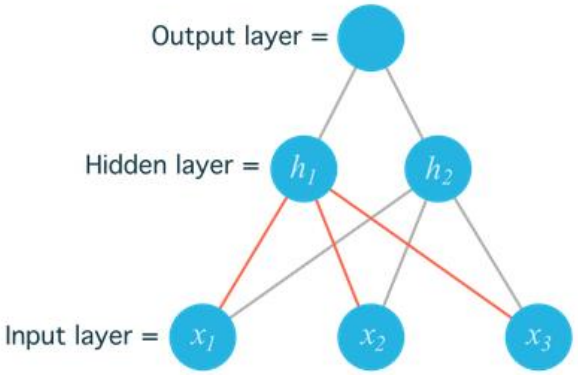
Code: [05Implementing Gradient Descent(gradient).txt]

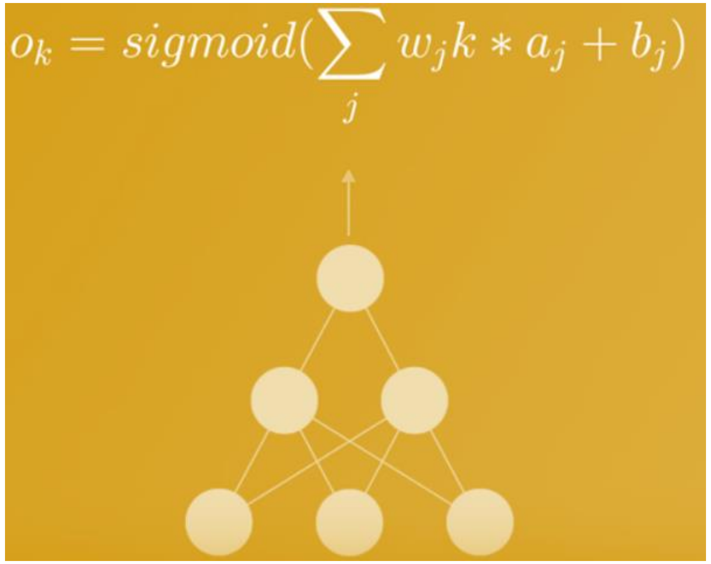
Code: [05Implementing Gradient Descent(data_prep).txt]

Code: [05binary.txt]

Code: [05solution.txt]

Multilayer Perceptrons





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.

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*:

Multilayer Perceptrons

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

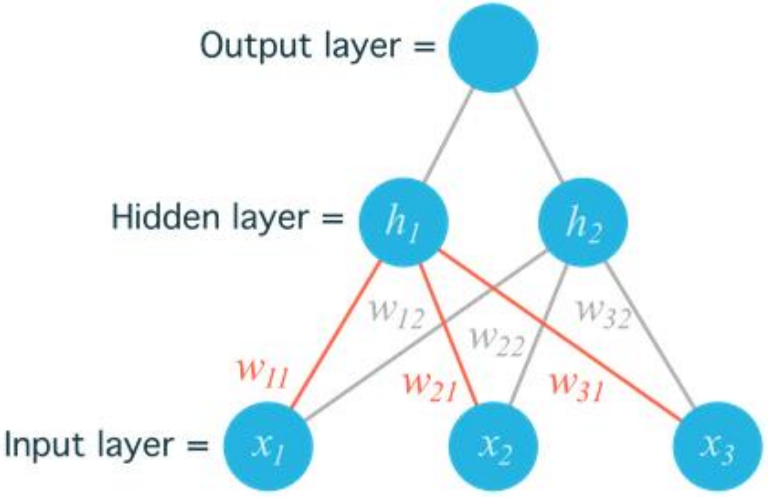
w_{11}

for the weight leading from *x1* to *h1*, and

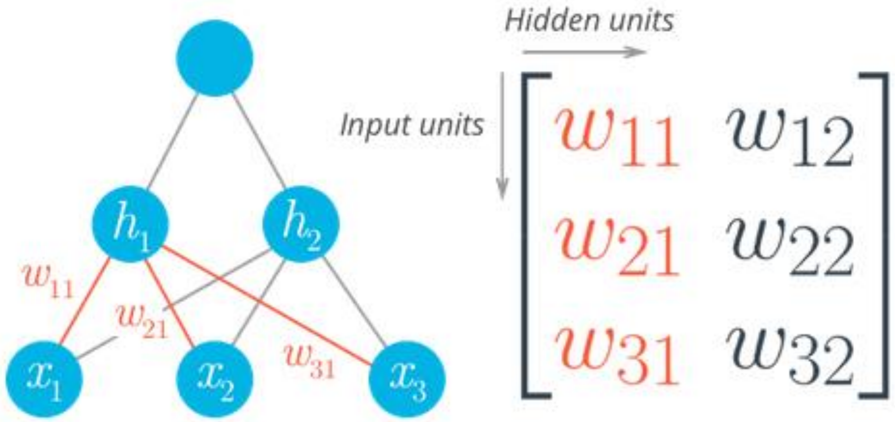
w_{12}

for the weight leading from x_1 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:



Multilayer Perceptrons

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))
```

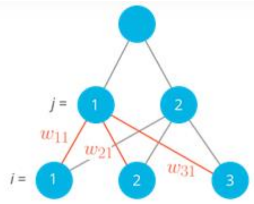
This creates a 2D array (i.e. a matrix) named weights_input_to_hidden with dimensions n_inputs by n_hidden. Remember how the input to a hidden unit is the sum of all the inputs multiplied by the hidden unit's weights. So for each hidden layer unit, hj, we need to calculate the following:

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

To do that, we now need to use matrix multiplication.

Multilayer Perceptrons

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:



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

Calculating the input to the first hidden unit with the first column of the weights 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.

Multilayer Perceptrons

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 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:

```
# 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)
```

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:

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

$2 \qquad 2 \times 3$

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

$2 \times 3 \qquad 3$

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.

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]])

Multilayer Perceptrons

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]])
```

I personally prefer keeping all vectors as 1D arrays, it just works better in my head.

Programming quiz

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

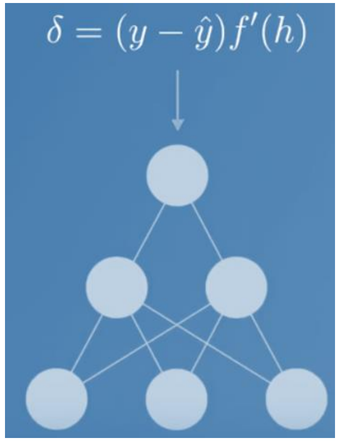
Things to do:

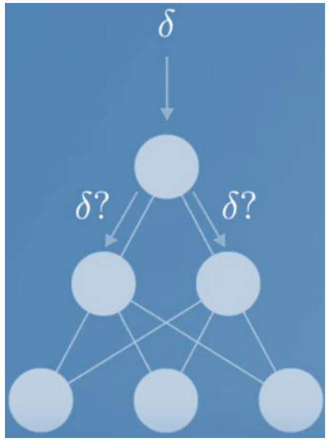
- Calculate the input to the hidden layer.
- Calculate the hidden layer output.
- Calculate the input to the output layer.
- Calculate the output of the network.

Code: 06Multilayer_Perceptrons.txt

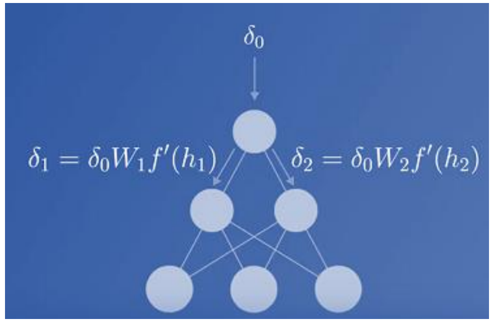
Code: 06solution.txt

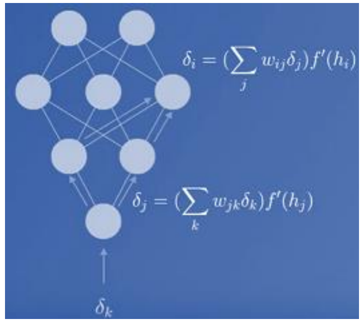
Backpropagation





How do we find these errors to use in the Gradient Descent step?





Backpropagation Is fundamental to how neural networks learn.

Backpropagation

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 δ_k^0 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

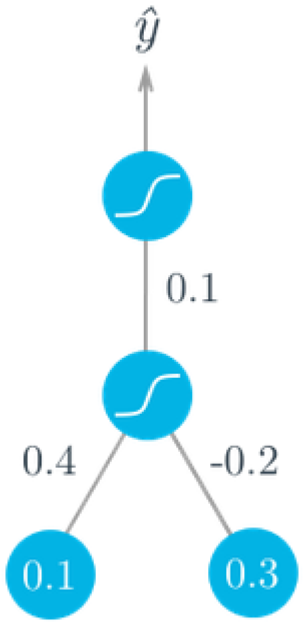
$$\Delta w_{pq} = \eta \delta_{output} V_{in}$$

- 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.

Backpropagation

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 $\hat{y}$ 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

$h=\sum_i W_i X_i = 0.4\times0.1-0.2\times0.3=-0.02$

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

$\hat{y}=f(W\cdot a)=\text{sigmoid}(0.1\times0.495)=0.512.$

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 for the output unit is

$\delta^0=(y-\hat{y})f'(W\cdot a)=(1-0.512)\times0.512\times(1-0.512)=0.122.$

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

For the hidden unit error, $\delta_j^h=\sum W_{jk}\delta_k^o f'(h_j)$, but since we have one hidden unit and one output unit, this is much simpler.

$\delta^h = W\delta^0 f'(h)= 0.1\times0.122\times0.495\times(1-0.495)=0.003$

Backpropagation

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.

$\Delta W = \eta \delta^0 a = 0.5 \times 0.122 \times 0.495 = 0.0302$

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

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

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,)
```

Backpropagation

Also, w_{ij} 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 error.
- Calculate the output layer error gradient.
- Use backpropagation to calculate the hidden layer error.
- Calculate the weight update steps.

Code: 07Backpropagation.txt

Code: 07solution.txt

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 Δw_{ij}=0
  - The hidden to output weights Δw_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, δ^0=(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 δ_j^h=$\delta^0 W_j f'(h_j)$
  - Update the weight steps,:
    - ΔW_j= ΔW_j +$\delta^0 a_j$
    - Δw_{ij}= Δw_{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.

Code: 08Implementing_Backpropagation.txt

Code: 08data_prep.txt

Code: 08binary.txt

Code: 08solution.txt

Gradient Descent and Backpropagation

The relationship between gradient descent and backpropagation

In machine learning, gradient descent and backpropagation often appear at the same time, and sometimes they can replace each other. So what is the relationship between them? In fact, we can consider backpropagation as a subset of gradient descent, which is the implementation of gradient descent in multi-layer neural networks. Since the same training rule recursively exists in each layer of the neural network, we can calculate the contribution of each weight to the total error inversely from the output layer to the input layer, which is so-called backpropagation.

Further Reading

Further reading

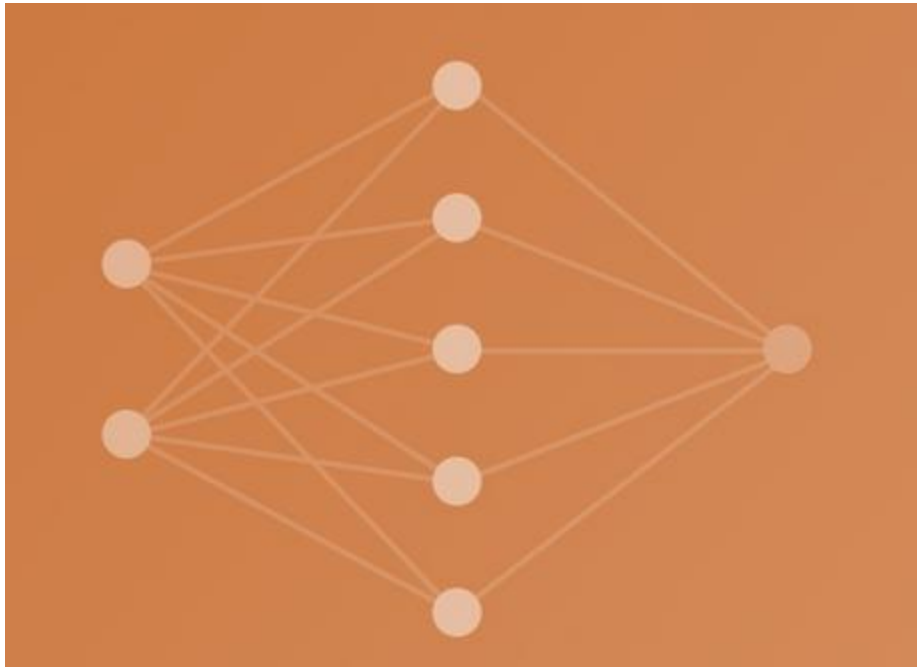
Backpropagation is fundamental to deep learning. TensorFlow and other libraries will perform the backprop for you, but you should really *really* understand the algorithm. We'll be going over backprop again, but here are some extra resources for you:

- From Andrej Karpathy: Yes, you should understand backprop
- Also from Andrej Karpathy, a lecture from Stanford's CS231n course

Create Your Own NN

In this lesson, you learned the power of perceptrons. How powerful one perceptron is and the power of a neural network using multiple perceptrons. Then you learned how each perceptron can learn from past samples to come up with a solution.

Now that you understand the basics of a neural network, the next step is to build a basic neural network. In the next lesson, you'll build your own neural network.



Summary

Next Step: Creating deep neural networks.