UC Berkeley Extension

Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Machine Learning With TensorFlow®**
- Introduction, Python - pros and cons
- Python modules, DL packages and scientific blocks
- Working with the shell, IPyton and the editor
- Installing the environment with core packages
- Writing “Hello World” HW1 (10pts)

- **Tensorflow and TensorBoard basics**
- Linear algebra recap
- Data types in Numpy and Tensorflow
- Basic operations in Tensorflow
- Graph models and structures with Tensorboard

- **TensorFlow operations**
- Overloaded operators
- Using Aliases
- Sessions, graphs, variables, placeholders
- Name scopes

- **Data Mining and Machine Learning concepts**
- Basic Deep Learning Models, k-Means
- Linear and Logistic Regression
- Softmax classification HW2 (10pts)

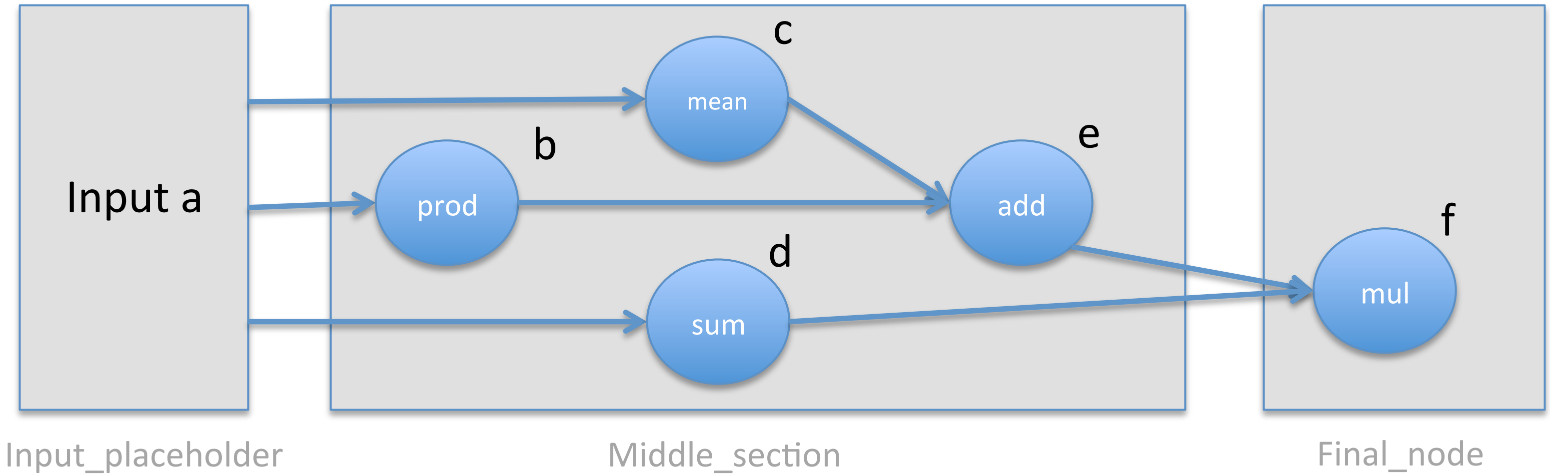
- **Neural Networks**
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions

TensorFlow

- HW2:

Recreate the graph and visualize it in Tensorboard using:

1. *Placeholder for an input array with dtype float32 and shape None*
2. *Scopes for the input, middle section and final node f*
3. *Feed the placeholder with an array A consisting of 100 normally distributed random numbers with Mean = 1 and Standard deviation = 2*
4. *Save your graph and show it in TensorBoard*
5. *Plot you input array on a separate figure*
6. *Make sure you comment your code well and provide your name on top of your work*
7. *Email your Github link (or code directly) to me including your .py file + screenshots of TensorBoard*



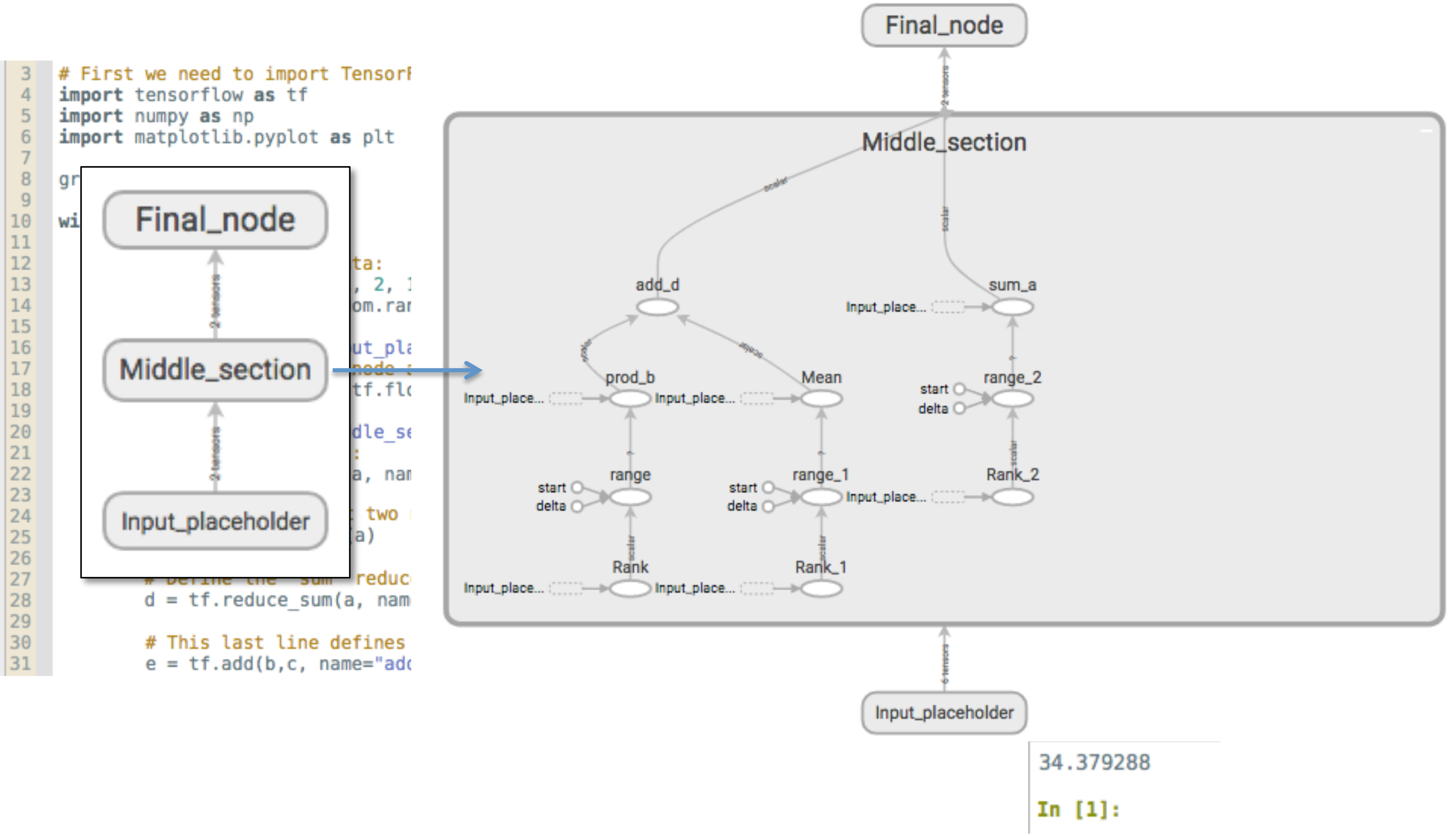
HW2 Discussion

```python
# First we need to import TensorFlow and NumPy:
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

graph = tf.Graph()

with graph.as_default():

    # Create some random data:
    xs = np.random.normal(1, 2, 100)
    ys = np.asarray(np.random.randn(len(xs)) * xs)

    with tf.name_scope("Input_placeholder"):
        # Define our input node as placeholder:
        a = tf.placeholder(tf.float32, None, name="a")

    with tf.name_scope("Middle_section"):
        # Defining nobe 'b':
        b = tf.reduce_prod(a, name="prod_b")

        # Defining the next two nodes in our graph:
        c = tf.reduce_mean(a)

        # Define the 'sum' reducer node for a:
        d = tf.reduce_sum(a, name="sum_a")

        # This last line defines the final node in our graph:
        e = tf.add(b,c, name="add_d")

    with tf.name_scope("Final_node"):
        # Create our final 'multiply' node:
        f = tf.multiply(e,d, name="final_multiply_node")

# To run we have to add the two extra lines or run them in the shell:
sess = tf.InteractiveSession(graph=graph)
init = sess.run(f, feed_dict={a: [xs,ys]})

# Plotting section:
plt.figure(1)
plt.scatter(xs,ys)
plt.pause(1)

plt.figure(2)
plt.plot(xs), plt.plot(ys)
plt.pause(2)

# Display the final result:
print(init)

# To create the graph:
sess.graph.as_graph_def()
file_writer = tf.summary.FileWriter('./', sess.graph)

# We clean up before we exit:
file_writer.close()
sess.close()
```

Python

```
34.379288

In [1]:
```

HW2 Discussion



TensorFlow

- Linear regression – *recall our model with data*:

```
def inputs():
    weight_age = [[84, 46], [73, 20], [65, 52], [70, 30], [76,
57], [69, 25], [63, 28], [72, 36], [79, 57], [75, 44], [27,
24], [89, 31], [65, 52], [57, 23], [59, 60], [69, 48], [60,
34], [79, 51], [75, 50], [82, 34], [59, 46], [67, 23], [85,
37], [55, 40], [63, 30]]
    blood_fat_content = [354, 190, 405, 263, 451, 302, 288,
385, 402, 365, 209, 290, 346, 254, 395, 434, 220, 374, 308,
220, 311, 181, 274, 303, 244]

    return tf.to_float(weight_age), tf.to_float(blood_fat_con-
tent)
```

Data source: http://people.sc.fsu.edu/~jburkardt/datasets/regression/x09.txt

TensorFlow

- *...vith da...*

Index of /~jburkardt/datasets/regression

Name	Last modified	Size	Description
Parent Directory		-	
regression.html	11-Mar-2015 08:19	20K	
x01.txt	03-Oct-2011 16:52	2.0K	
x02.txt	03-Oct-2011 16:52	1.0K	
x03.txt	03-Oct-2011 16:52	1.2K	
x04.txt	03-Oct-2011 16:52	1.7K	
x05.txt	03-Oct-2011 16:52	1.9K	
x06.txt	03-Oct-2011 16:52	1.7K	
x07.txt	03-Oct-2011 16:52	2.0K	
x08.txt	03-Oct-2011 16:52	1.4K	
x09.txt	03-Oct-2011 16:52	1.2K	
x10.txt	03-Oct-2011 16:52	1.2K	
x11.txt	03-Oct-2011 16:52	2.9K	
x12.txt	03-Oct-2011 16:52	1.4K	

```
de
                   65, 52]
57                 57],
24                 60],
34                 46],
37

                   263, 4
38                 434, 2
22

                   to_floa
te
```

	A	B	C
1	Weight (kg)	Age (Years)	Blood fat
2	84	46	354
3	73	20	190
4	65	52	405
5	70	30	263
6	76	57	451
7	69	25	302
8	63	28	288
9	72	36	385
10	79	57	402
11	75	44	365
12	27	24	209
13	89	31	290
14	65	52	346
15	57	23	254
16	59	60	395
17	69	48	434
18	60	34	220
19	79	51	374
20	75	50	308
21	82	34	220
22	59	46	311
23	67	23	181
24	85	37	274
25	55	40	303
26	63	30	244

Data source: http://people.sc.fsu.edu/~jburkardt/datasets/reg

```python
# Linear regression:
import tensorflow as tf
import pandas as pd

W = tf.Variable(tf.zeros([2, 1]), name="weights")
b = tf.Variable(0., name="bias")

# Computing our model in a series of mathematical operations that we apply to our data:
def inference(X):
    return tf.matmul(X, W) + b

# Calculate loss over expected output:
def loss(X, Y):
    Y_predicted = tf.transpose(inference(X)) # make it a row vector
    return tf.reduce_sum(tf.squared_difference(Y, Y_predicted))

# Read input training data:
def inputs():
    weight_age = []
    blood_fat  = []
    data = pd.read_csv('blood_fat_data.csv')
    data.head(1)                      # reads the first line
    rows = len(data)                  # counts the number of rows in the file
    shape = data.shape                # shows the shape
    columns = (data.columns)          # shows the column titles
    weight = data[columns[0]]         # write entire column
    age = data[columns[1]]            # write entire column
    blood_fat_content = data[columns[2]]    # write entire column
    for k in range(rows):             # use loop to put it in the expected format
        weight_age.append([weight[k], age[k]])
        blood_fat.append(blood_fat_content[k],)

    return tf.to_float(weight_age), tf.to_float(blood_fat)
```

```python
# Linear regression:
import tensorflow as tf
import pandas as pd

W = tf.Variable(tf.zeros([2, 1]), name="weights")
b = tf.Variable(0., name="bias")

# Computing our model in a series of mathematical operations that we apply to our data:
def inference(X):
    return tf.matmul(X, W) + b

# Calculate loss over expected
def loss(X, Y):
    Y_predicted = tf.transpose(inference(X)) # make it a row vector
    return tf.reduce_sum(tf.squared_difference(Y, Y_predicted))

# Read input training data:
def inputs():
    weight_age = []
    blood_fat  = []
    data = pd.read_csv('blood_fat_data.csv')
    data.head(1)                    # reads the first line
    rows = len(data)                # counts the number of rows in the file
    shape = data.shape              # shows the shape
    columns = (data.columns)        # shows the column titles
    weight = data[columns[0]]       # write entire column
    age = data[columns[1]]          # write entire column
    blood_fat_content = data[columns[2]]     # write entire column
    for k in range(rows):           # use loop to put it in the expected format
        weight_age.append([weight[k], age[k]])
        blood_fat.append(blood_fat_content[k],)

    return tf.to_float(weight_age), tf.to_float(blood_fat)
```

```
In [2]: tf.matmul?
Signature: tf.matmul(a, b, transpose_a=False, transpose_b=False,
False, name=None)
Docstring:
Multiplies matrix `a` by matrix `b`, producing `a` * `b`.
```

all Pandas objects

```python
# Using training, we adjust the model parameters:
def train(total_loss):
    learning_rate = 0.000001
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)

# We evaluate the resulting model:
def evaluate(sess, X, Y):
    print(sess.run(inference([[55., 40.]]))) # ~ 295 (but it is 303)
    print(sess.run(inference([[50., 70.]]))) # ~ 256 (other values not in table)
    print(sess.run(inference([[90., 20.]]))) # ~ 303 ( ... )
    print(sess.run(inference([[90., 70.]]))) # ~ 256 ( ... )

# Launch the graph in a session and run the training loop:
with tf.Session() as sess:

    tf.initialize_all_variables().run()

    X, Y = inputs()
    total_loss = loss(X, Y)
    train_op = train(total_loss)

    # Actual training loop:
    training_steps = 10000
    for step in range(training_steps):
        sess.run([train_op])
        # See how the loss gets decremented thru training steps:
        if step % 1000 == 0:
            print("Epoch:", step, " loss: ", sess.run(total_loss))

    print("Final model W=", sess.run(W), "b=", sess.run(b))
    evaluate(sess, X, Y)

    sess.close()
```

```python
# Using training, we adjust the model parameters:
def train(total_loss):
    learning_rate = 0.000001
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)

# We evaluate the resulting model:
def evaluate(sess, X, Y):
    print(sess.run(inference([[55., 40.]])))
    print(sess.run(inference([[50., 70.]])))
    print(sess.run(inference([[90., 20.]])))
    print(sess.run(inference([[90., 70.]])))

# Launch the graph in a session and run the
with tf.Session() as sess:

    tf.initialize_all_variables().run()

    X, Y = inputs()
    total_loss = loss(X, Y)
    train_op = train(total_loss)

    # Actual training loop:
    training_steps = 10000
    for step in range(training_steps):
        sess.run([train_op])
        # See how the loss gets decremented thru training steps:
        if step % 1000 == 0:
            print("Epoch:", step, " loss: ", sess.run(total_loss))

    print("Final model W=", sess.run(W), "b=", sess.run(b))
    evaluate(sess, X, Y)

    sess.close()
```

Python

```
ated and will be removed after 2017-03-02.
Instructions for updating:
Use `tf.global_variables_initializer` instead.
Epoch: 0  loss:  1230281.8
Epoch: 1000  loss:  47094.402
Epoch: 2000  loss:  47081.83
Epoch: 3000  loss:  47069.746
Epoch: 4000  loss:  47057.695
Epoch: 5000  loss:  47045.66
Epoch: 6000  loss:  47033.69
Epoch: 7000  loss:  47021.734
Epoch: 8000  loss:  47009.824
Epoch: 9000  loss:  46997.938
Final model W= [[1.2922349]
 [5.5893784]] b= 1.1374356
[[177.53676]]
[[457.00565]]
[[229.22615]]
[[508.69504]]
```

TensorFlow

- Logistic regression: *example*

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

learning_rate = 0.01
training_epochs = 500

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

# Create our data points on the x and y axis:
x1 = np.random.normal(5, 3, 100)
x2 = np.random.normal(-5, 3, 100)
xs = np.append(x1, x2)
ys = np.asarray([0.] * len(x1) + [1.] * len(x2))
plt.scatter(xs, ys)

# Create our parameters and placeholders for X and Y to feed them with the data above:
X = tf.placeholder(tf.float32, shape=(None,), name="x")
Y = tf.placeholder(tf.float32, shape=(None,), name="y")
w = tf.Variable([0., 0.], name="parameter", trainable=True)
y_model = tf.sigmoid(-(w[1] * X + w[0]))
```

TensorFlow

- Logistic regression: *example*

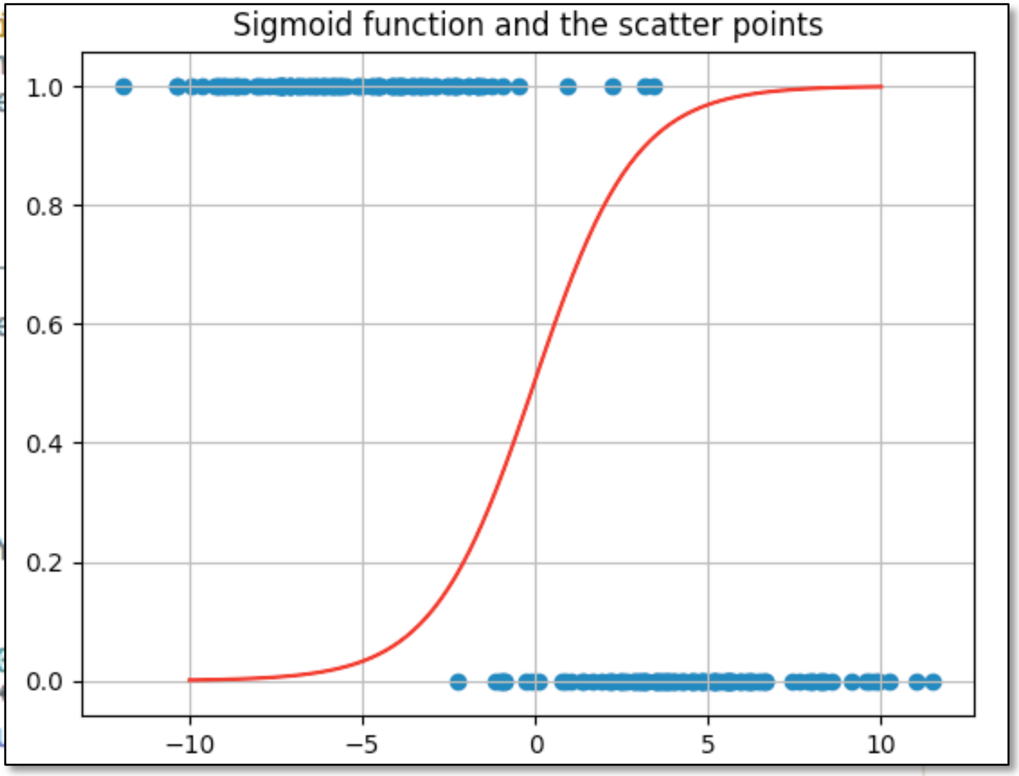
```
# Calculate the cost and adaptation (learning):
cost = tf.reduce_mean(-tf.log(y_model * Y + (1 - y_model) * (1 - Y)))
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Run the model:
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    prev_err = 0
    for epoch in range(training_epochs):
        err, _ = sess.run([cost, train_op], {X: xs, Y: ys}) # err = cost
        print(epoch, err)
        if abs(prev_err - err) < 0.0001: # adjust to see curve change with epochs
            break   # Check when the error is small enough to quit
        prev_err = err
    w_val = sess.run(w, {X: xs, Y: ys})

# Plot the resulting sigmoid:
all_xs = np.linspace(-10, 10, 100)
plt.plot(all_xs, sigmoid(all_xs * w_val[1] + w_val[0]), 'r') # calculate the sigmoid
plt.grid(), plt.title("Sigmoid function and the scatter points")
plt.pause(1)
```

TensorFlow

- Logistic regression: *example*

```
# Calculate the cost and adaptati
cost = tf.reduce_mean(-tf.log(y_m
train_op = tf.train.GradientDesce

# Run the model:
with tf.Session() as sess:
    sess.run(tf.global_variables_
    prev_err = 0
    for epoch in range(training_e
        err, _ = sess.run([cost,
        print(epoch, err)
                        err) <
                        ck when

    w_va                X: xs, Y

# Plot t                oid:
all_xs =                10, 100
plt.plot                all_xs *
plt.grid                gmoid fu
plt.paus
```

```
0 0.6931474
1 0.63596
2 0.58760583
3 0.5465827
4 0.5115978
5 0.4815739
  ...
188 0.15198787
189 0.15188722
190 0.15178768
```



Dealing with the data

- We need to be able to split the data in training and testing
- There are many ways to do this
- In the next slides we will see 4 of the most common methods

Cross-validation

- Estimates the prediction error in production
- Helps find the best fit model (out of many)
- Helps ensure avoiding overfitting
- In cross-validation, you decide on a fixed number of folds, or partitions, of the data
- Four main types:
  - Holdout method
  - K-Fold cross validation (CV)
  - Leave one out CV
  - Bootstrap method

Holdout estimation

- What should we do if we only have a single dataset?
- The *holdout* method reserves a certain amount for testing and uses the remainder for training, after shuffling
  - Usually: one third for testing, the rest for training
- Problem: the samples might not be representative
  - Example: class might be missing in the test data, so this method can be biased
- Advanced version uses *stratification*
  - Ensures that each class is represented with approximately equal proportions in both subsets

Repeated holdout method

- Holdout estimate can be made more reliable by repeating the process with different subsamples
  - In each iteration, a certain proportion is randomly selected for training (possibly with *stratificiation*)
  - The error rates on the different iterations are averaged to yield an overall error rate
- This is called the *repeated holdout* method
- Still not optimum: the different test sets overlap
  - Can we prevent overlapping?

Cross-validation

- *K-fold cross-validation* avoids overlapping test sets
  - First step: split data into *k* subsets of equal size
  - Second step: use each subset for testing, the remainder for training
  - This means the learning algorithm is applied to *k* different training sets
- Often the subsets are *stratified* before the cross-validation is performed to yield stratified *k*-fold cross-validation
- The error estimates are averaged to yield an overall error estimate; also, standard deviation is often computed
- Alternatively, predictions and actual target values from the *k* folds are pooled to compute one estimate
  - Does not yield an estimate of standard deviation

More on cross-validation

- Standard method for evaluation is:
  stratified ten-fold cross-validation
- Why ten?
  - Extensive experiments have shown that this is the best choice to get an accurate estimate
  - There is also some theoretical evidence for this
- Stratification reduces the estimate's variance
- Even better: repeated stratified cross-validation
  - E.g., ten-fold cross-validation is repeated ten times and results are averaged (reduces the variance)

Leave-one-out Cross-Validation

- Leave-one-out:
  is a particular form of *k*-fold cross-validation (CV):
  - Set number of folds to = number of training instances
  - I.e., for *n* training instances, build classifier *n* times
- Makes best use of the data (especially when small set)
- Involves no random subsampling
- Very computationally expensive (exception: using lazy classifiers such as the nearest-neighbor classifier)

Leave-one-out CV and Stratification

- Disadvantage of Leave-one-out CV:

  stratification is not possible

  - In fact, it *guarantees* a non-stratified sample because there is only one instance in the test set!

- Extreme example:

  random dataset split equally into two classes

  - Best is 50% accuracy on fresh data (when 2 classes presented)
  - Leave-one-out CV estimate can give 100% error in some instances!

The bootstrap

- CV uses sampling *without replacement*
    - The same instance, once selected, can not be selected again for a particular training/test set

- The *bootstrap* uses sampling *with replacement* to form the training set, also known as *bagging*
    - Sample a dataset of *n* instances *n* times *with replacement* to form a new dataset of *n* instances
    - Use this data as the training set
    - Use the instances from the original dataset that do not occur in the new training set for testing

The 0.632 bootstrap

- Also called the *0.632 bootstrap*
- A particular instance has a probability of 1–1/*n* of *not* being picked
- Thus its probability of ending up in the test data is:

$$\left(1-\frac{1}{n}\right)^{n} \approx e^{-1} = 0.368$$

- This means the training data will contain approximately 63.2% of the instances

Estimating error with the 0.632 bootstrap

- The error estimate on the test data will be quite pessimistic
  - Trained on just ~63% of the instances
- Idea: combine it with the resubstitution error:

$$e = 0.632 \cdot e_{\text{test instances}} + 0.368 \cdot e_{\text{training instances}}$$

- The resubstitution error gets less weight than the error on the test data
- Repeat process several times with different samples; average the results

More on the bootstrap

- Probably the best way of estimating performance for very small datasets

- However, it has some problems
  - Consider the random dataset from above
  - A perfect memorizer will achieve
    0% resubstitution error and ~50% error on test data
  - Bootstrap estimate for this classifier:

$$e = 0.632 \times e_{\text{test instances}} + 0.368 \times e_{\text{training instances}}$$

  - True expected error: 50%

$$(0.632 \times 50\% + 0.368 \times 0\%) = 31.6\%$$

Data Science

Class 5 …

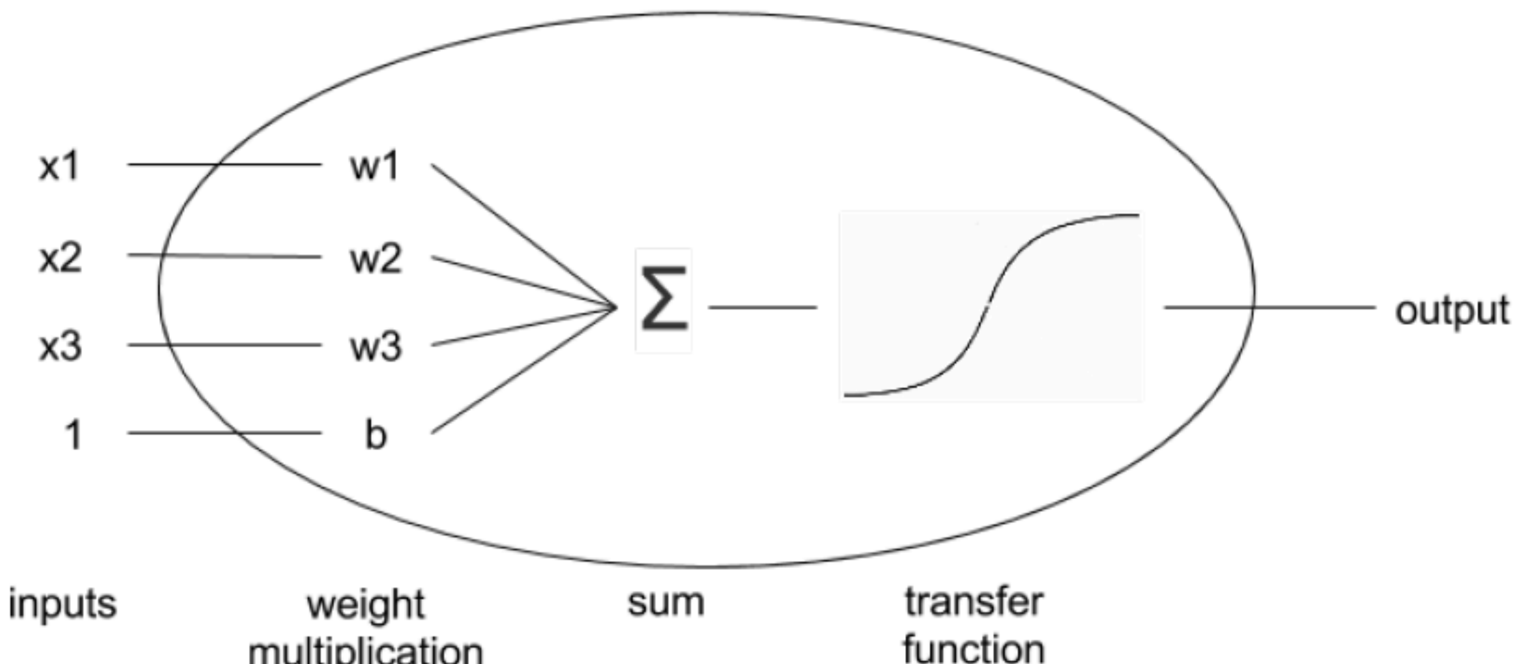
Neural networks …

Multi-layer neural networks

- So far we have been using simple neural networks.
- Both linear and logistic regression models are single neurons that:
  - Do a weighted sum of the input features. Bias can be thought of as the weight of an input feature that equals 1 for every example. We call that a *linear combination* of the features
  - Then apply an activation or transfer function to calculate the output. In the case of the lineal regression, the transfer function is the identity (i.e. same value), while the logistic uses the sigmoid as the transfer.

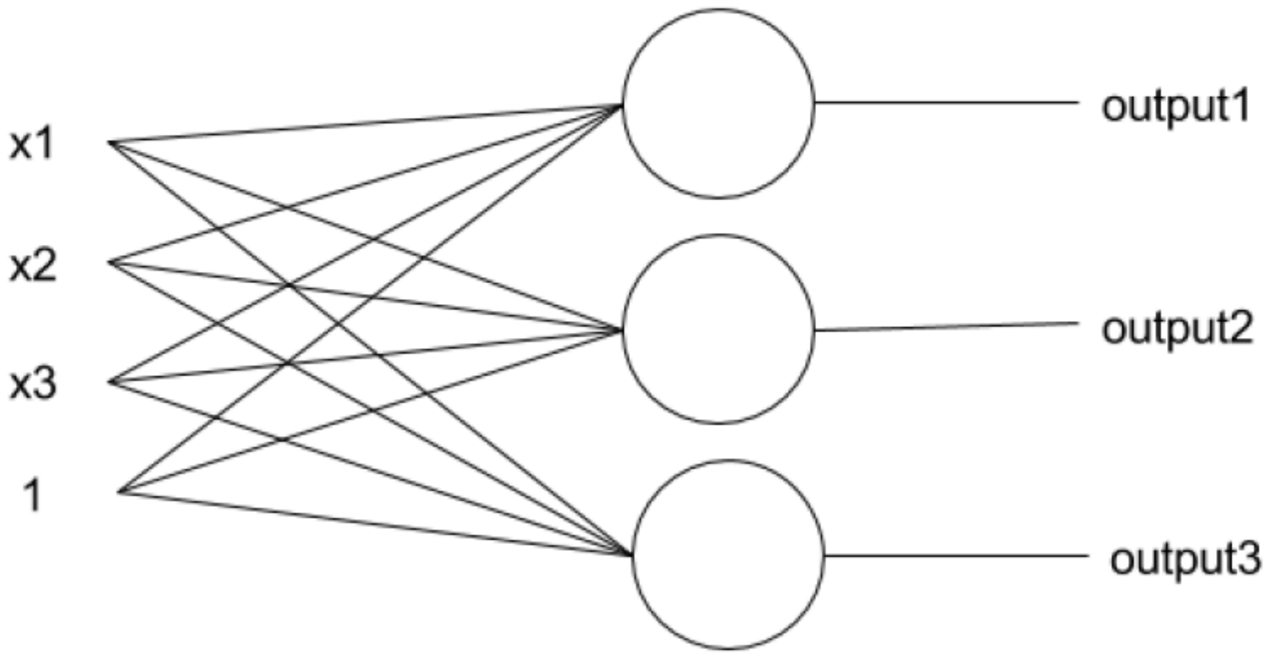
Multi-layer neural networks

- The following diagram represents each neuron inputs, processing and output:



Multi-layer neural networks

- In the case of softmax classification, we used a network with C neurons- one for each possible output class:



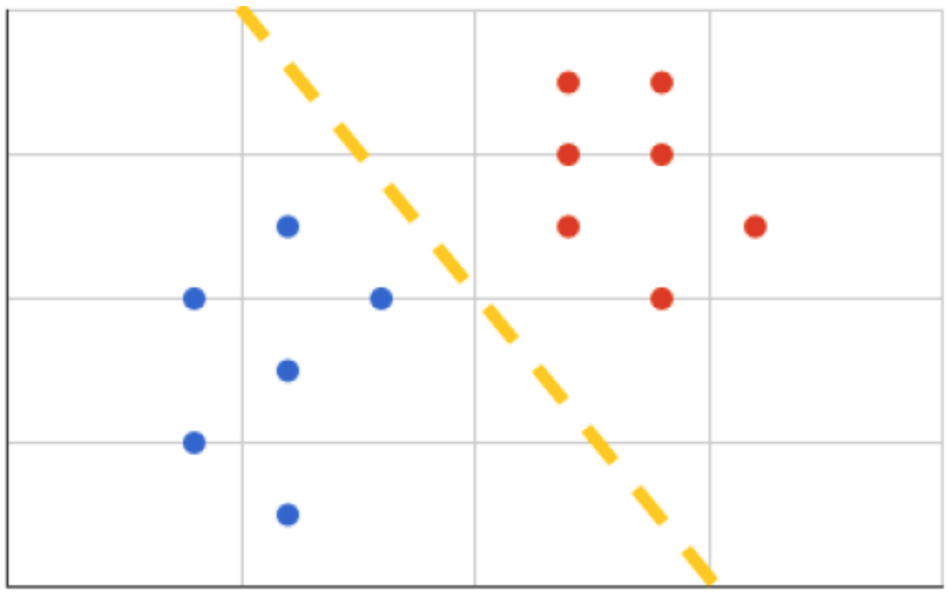
Multi-layer neural networks

- Now, in order to resolve more difficult tasks, like reading handwritten digits, or identifying cats and dogs on images, we are going to need a more developed model.
- Lets start with a simple example:
  - Suppose we want to build a network that learns how to fit the XOR (eXclusive OR) Boolean operation:

XOR operation truth table

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

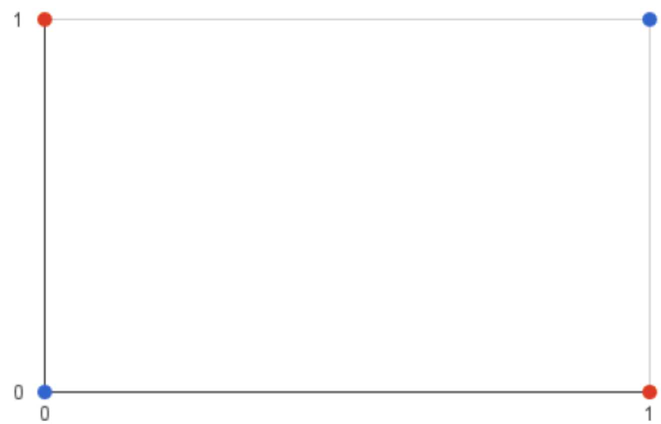
Multi-layer neural networks



- In the chart we can see example data samples as dots, with their associated class as the color.
- As long as we can find that yellow line completely separating the red and the blue dots in the chart, the sigmoid neuron will work fine for that dataset.

Multi-layer neural networks

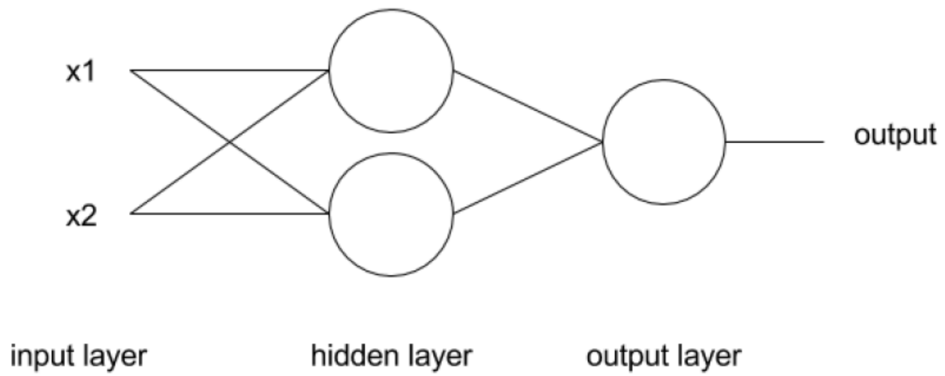
- Let's look at the XOR gate function chart:



- We can't find a single straight line that would split the chart, leaving all of the 1s (red dots) in one side and 0s (blue dots) in the other.
- That's because the XOR function output is not linearly separable.

Multi-layer neural networks

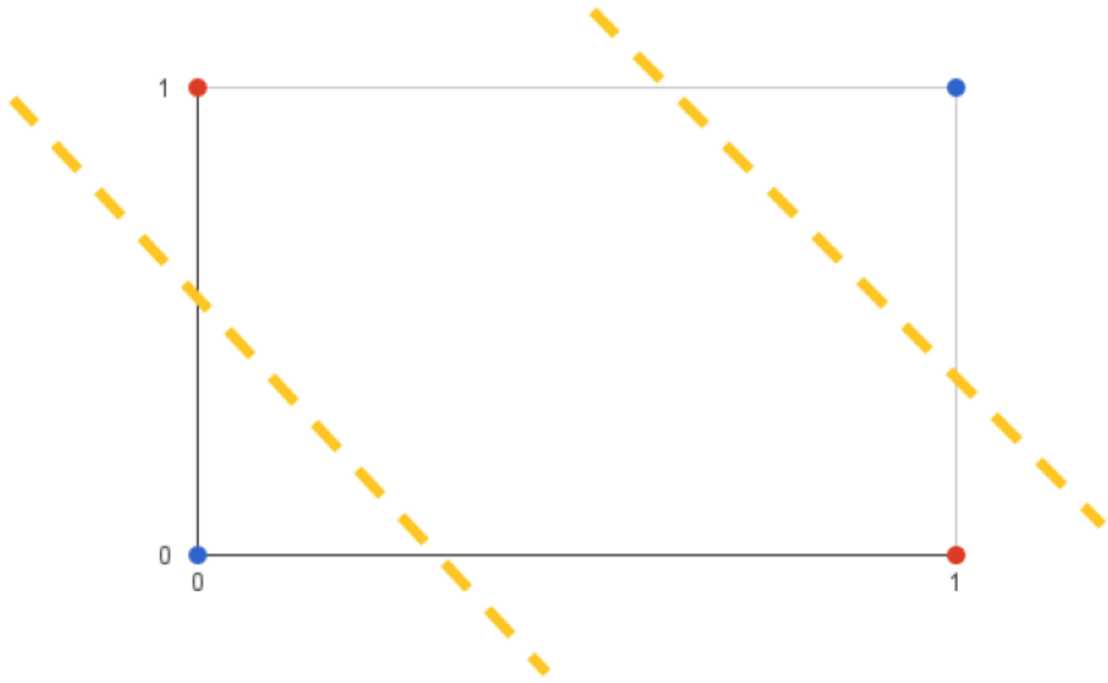
- Using more neurons between the input and the output of the network, introducing the *hidden layer*:



- You can think of it as allowing our network to ask multiple questions to the input data, one question per neuron on the hidden layer
- Deciding the output based on the answers of those questions

Multi-layer neural networks

- Graphically, we are allowing the network to draw more than one single separation line:



Gradient descent and backpropagation

- Gradient descent is an algorithm to find the points where a function achieves its minimum value.
- Remember that we defined learning as improving the model parameters in order to minimize the loss through a number of training steps
- With that concept, applying gradient decent to find the minimum of the loss function will result in our model learning from our input data

Gradient descent and backpropagation

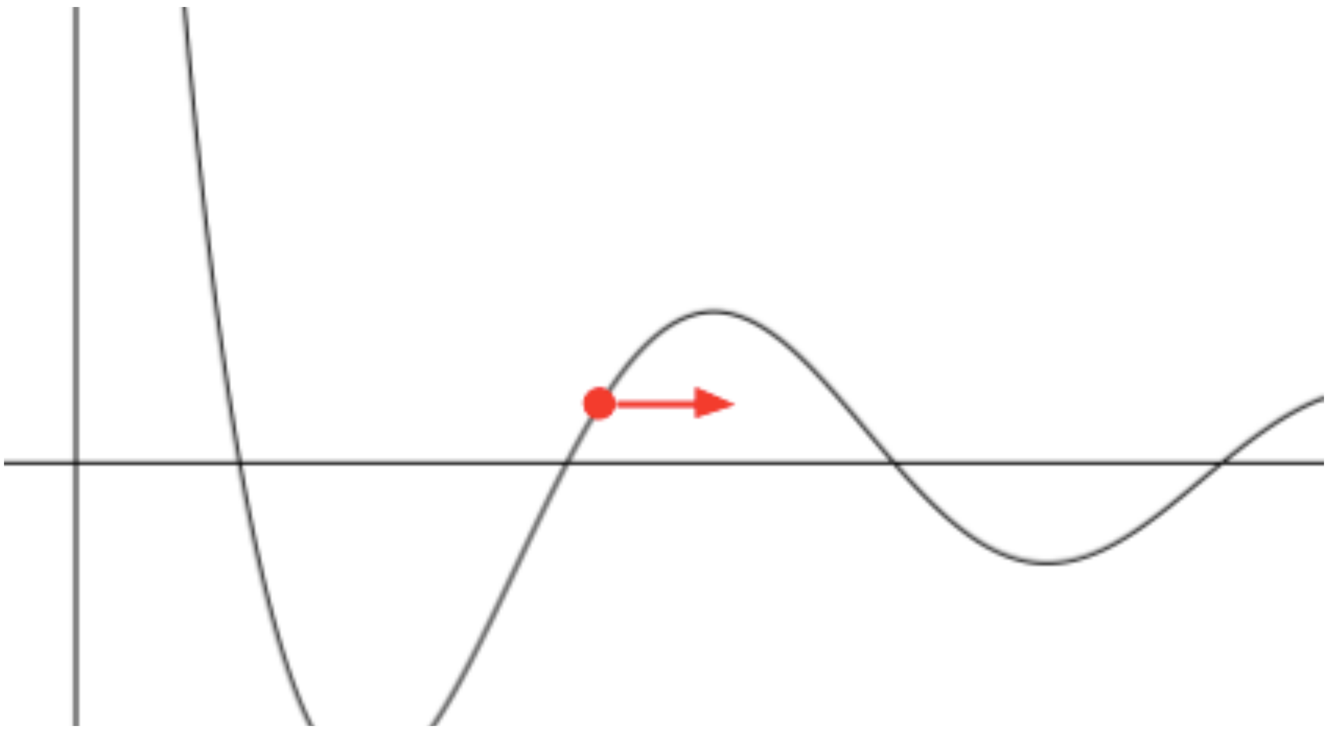
- What is a gradient?
- The gradient is a mathematical operation, generally represented with the ∇ symbol (nabla greek letter).
- It is analogous to a derivative, but applied to functions that input a vector and output a single value; like our loss functions do
- The output of the gradient is a vector of partial derivatives, one per position of the input vector of the function

$$\nabla \equiv \left(\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_N} \right)$$

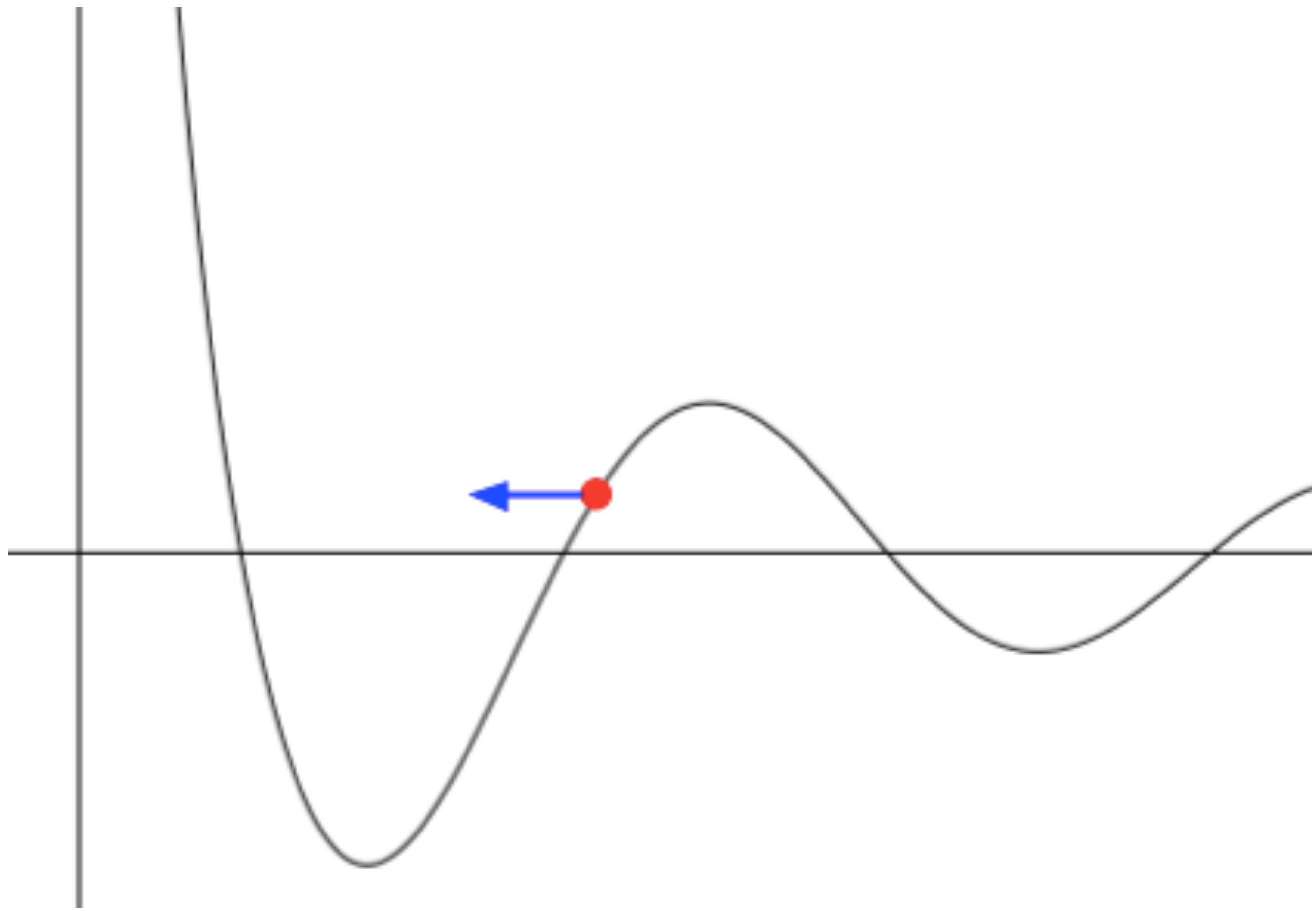
Gradient descent and backpropagation

- Few caveats:
  - When we talk about input variables of the loss function, we are referring to the model weights, not that actual dataset features inputs.
  - The latter are fixed by our dataset and cannot be optimized.
  - The partial derivatives we calculate are with respect of each individual weight in the inference model.
  - We care about the gradient because its output vector indicates the direction of maximum growth for the loss function.
  - You could think of it as a little arrow that will indicate in
  - every point of the function where you should move to increase its value: … *see next slide*

Gradient descent and backpropagation



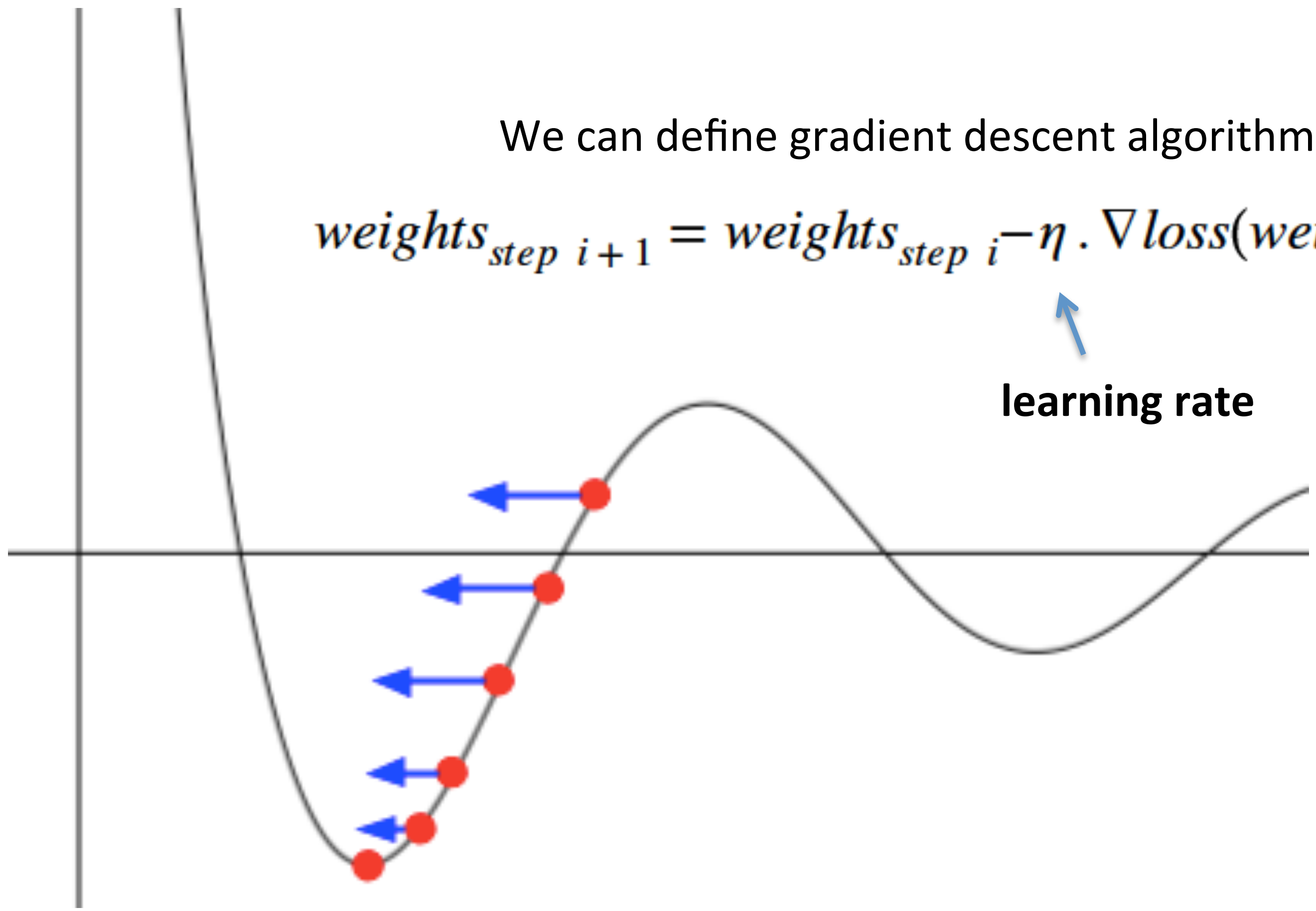
Gradient descent and backpropagation



Gradient descent and backpropagation

We can define gradient descent algorithm as:

$$weights_{step\ i+1} = weights_{step\ i} - \underbrace{\eta}_{\textbf{learning rate}} . \nabla loss(weights_{step\ i})$$

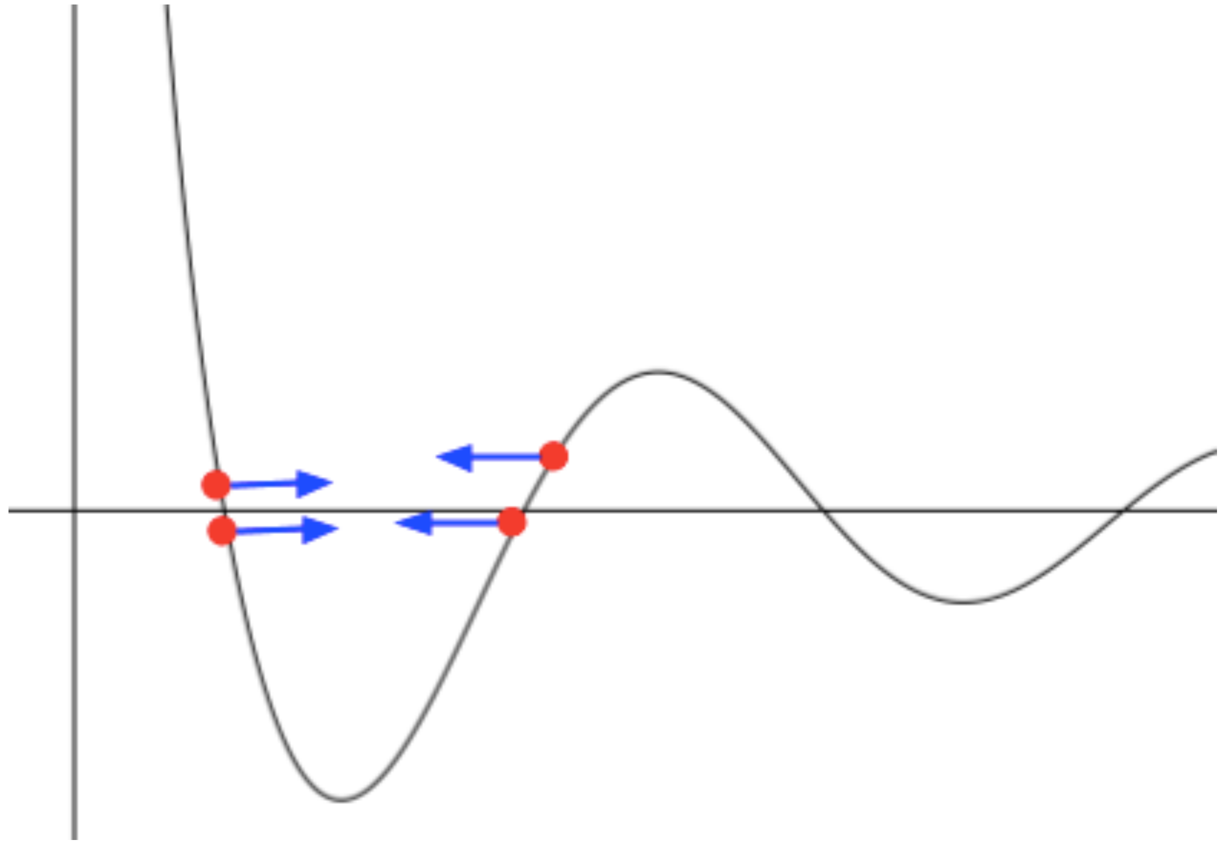


Gradient descent and backpropagation

- The learning rate is not a value that model will infer
- It is a hyperparameter, or a manually configurable setting for our model
- We need to figure out the right value for it:
  - If it is too small then it will take many learning cycles to find the loss minimum
  - If it is too large, the algorithm may simply "skip over" the minimum and never find it, jumping cyclically.
- That's known as overshooting.

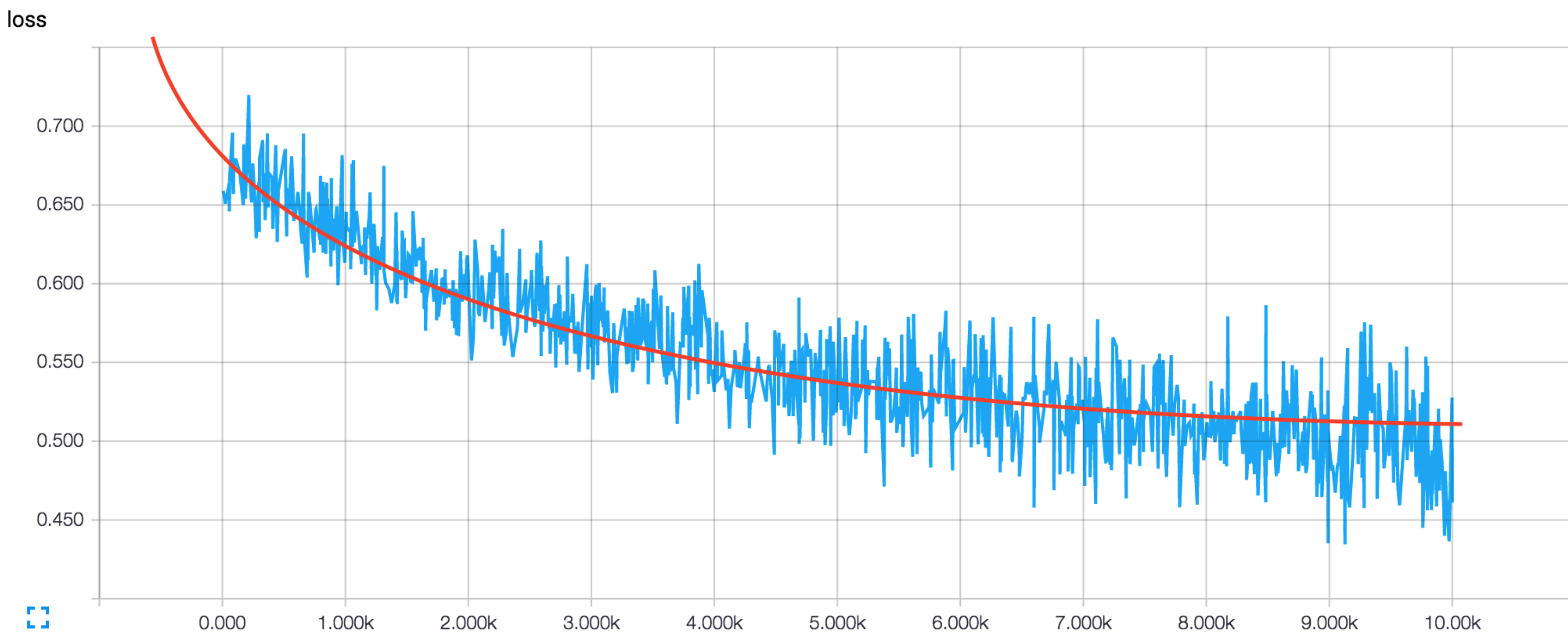
Gradient descent and backpropagation

- Here is what overshooting looks like:



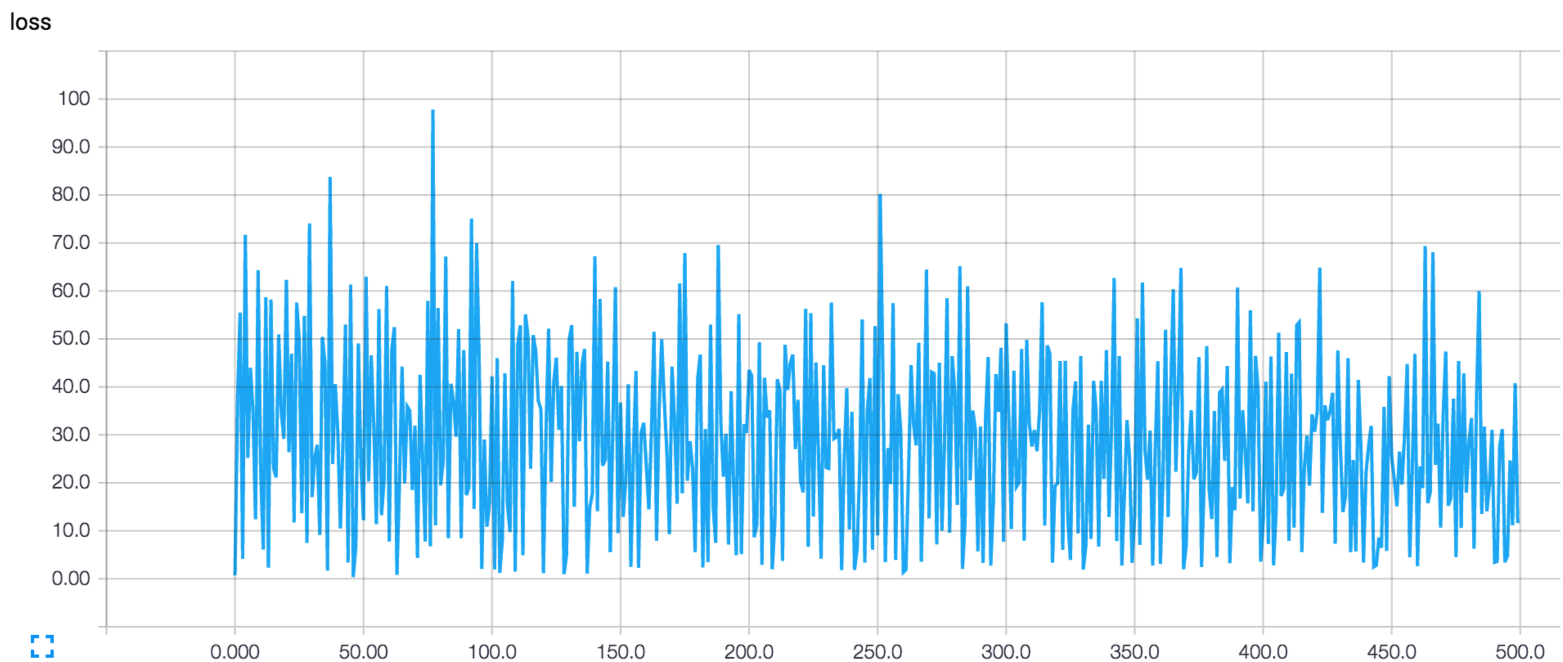
Gradient descent and backpropagation

- This is how a well behaving loss should diminish through time, indicating a good learning rate:

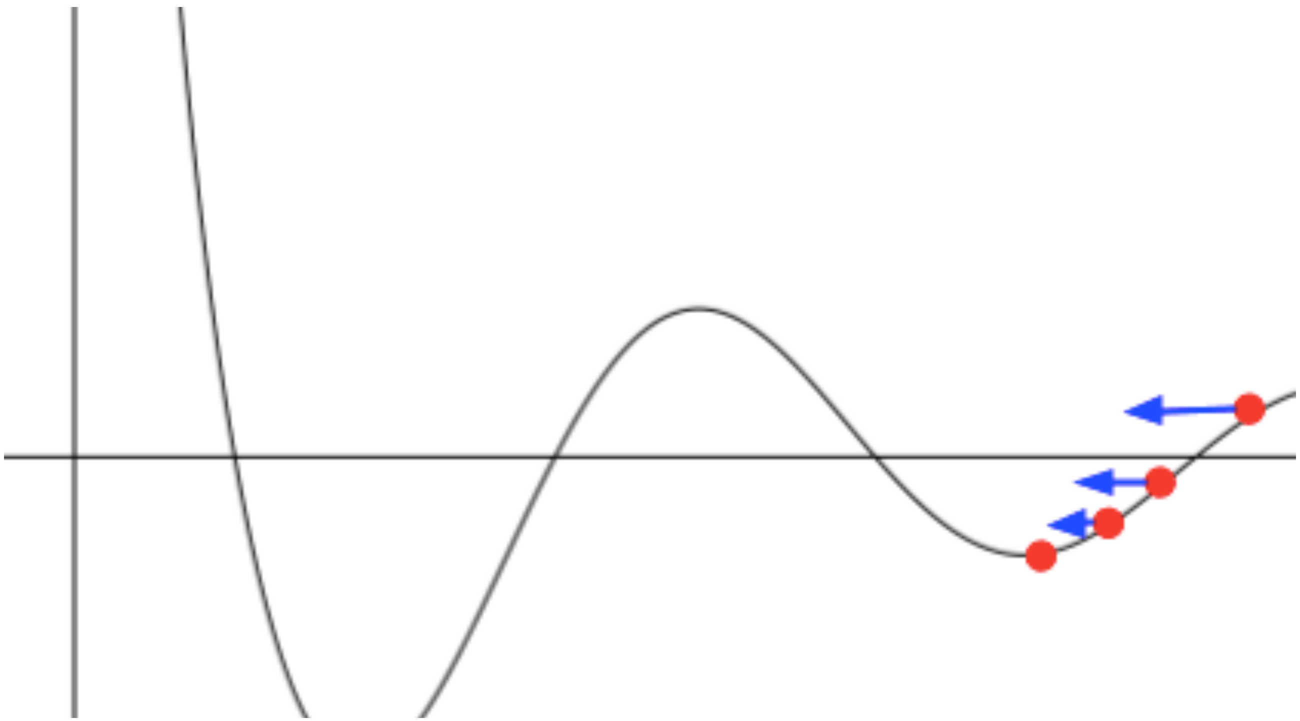


Gradient descent and backpropagation

- This is what it looks like when it is overshooting:



Gradient descent and backpropagation



Gradient descent and backpropagation

- Tensorflow includes the method tf.gradients to symbolically compute the gradients of the specified graph steps and output that as tensors

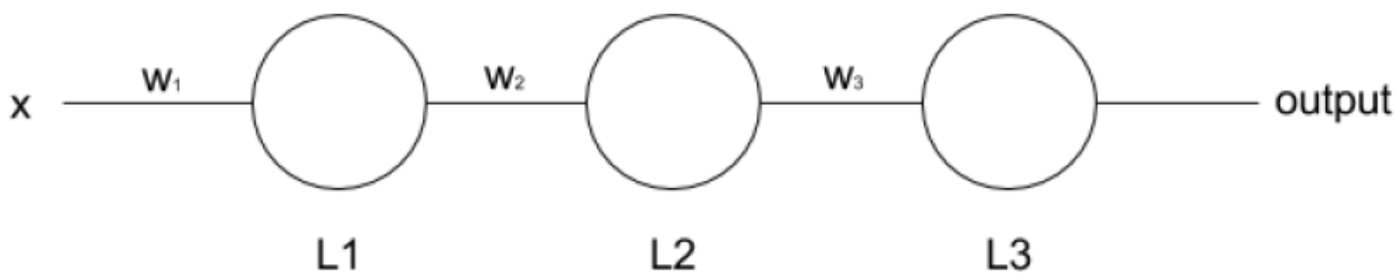
- We don't need to manually call, because it also includes implementations of the gradient descent algorithm, among others.

- We are going to present the backpropagation next

- It is a technique used for efficiently computing the gradient in a computational graph

Gradient descent and backpropagation

- Let's assume a really simple network, with one input, one output, and two hidden layers with a single neuron.
- Both hidden and output neurons will be sigmoids and the loss will be calculated using cross entropy.
- Such a network should look like this:



Gradient descent and backpropagation

- Let's define L_1 as the output of first hidden layer, L_2 the output of the second, and L_3 the final output of the network:

$$L1 = sigmoid(w_1 . x)$$

$$L2 = sigmoid(w_2 . L1)$$

$$L3 = sigmoid(w_3 . L2)$$

- The loss of the network will be:

$$loss = cross\_entropy(L3, y_{expected})$$

Gradient descent and backpropagation

- To run one step of gradient decent, we need to calculate the partial derivatives of the loss function with respect of the three weights in the network.

- We will start from the output layer weights, applying the chain rule:

$$\frac{\partial loss}{\partial w_3} = cross\_entropy'\left(L3, y_{expected}\right) . sigmoid'(w_3 . L2) . L2$$

- L_2 is just a constant for this case as it doesn't depend on w_3

Gradient descent and backpropagation

- To simplify the expression we could define:

$$loss' = cross\_entropy'\left(L3, y_{expected}\right)$$

$$L3' = sigmoid'\left(w_3 \cdot L2\right)$$

- The resulting expression for the partial derivative would be:

$$\frac{\partial loss}{\partial w_3} = loss' \cdot L3' \cdot L2$$

Gradient descent and backpropagation

- Now let's calculate the derivative for the second hidden layer weight, w_2:

$$L2' = sigmoid'(w_2 \cdot L1)$$

$$\frac{\partial loss}{\partial w_2} = loss' \cdot L3' \cdot L2' \cdot L1$$

- And finally the derivative for w_1:

$$L1' = sigmoid'(w_1 \cdot x)$$

$$\frac{\partial loss}{\partial w_1} = loss' \cdot L3' \cdot L2' \cdot L1' \cdot x$$

Gradient descent and backpropagation

- We notice a pattern:
  - The derivative on each layer is the product of the derivatives of the layers after it by the output of the layer before.
  - That's the magic of the chain rule and what the algorithm takes advantage of.
- We go forward from the inputs calculating the outputs of each hidden layer up to the output layer.
- Then we start calculating derivatives going backwards through the hidden layers and propagating the results in order to do less calculations by reusing all of the elements already calculated
- That's the origin of the name backpropagation.

Object Recognition and Classification

- At this point, we should have a basic understanding of **TensorFlow** and its best practices
- We can now build a model capable of object recognition and classification
- Building this model expands on the fundamentals that have been covered so far while adding terms, techniques and fundamentals of computer vision
- The technique used in training the model has become popular recently due to its accuracy across challenges

Object Recognition and Classification

- ImageNet, a database of labeled images, is where computer vision and deep learning saw a recent rise in popularity
- Convolutional Neural Networks (CNNs) primarily used for computer vision related tasks but are not limited to working with images
- For images, the values in the tensor are pixels ordered in a grid corresponding with the width and height of the image

Object Recognition and Classification

- The dataset used in training this CNN model is a subset of the images available in ImageNet named the Stanford's Dogs Dataset - http://vision.stanford.edu/aditya86/ImageNetDogs/

n02110185_712.jpg n02110185_4186.jpg n02110185_1532.jpg