UC Berkeley Extension

Machine Learning With TensorFlow

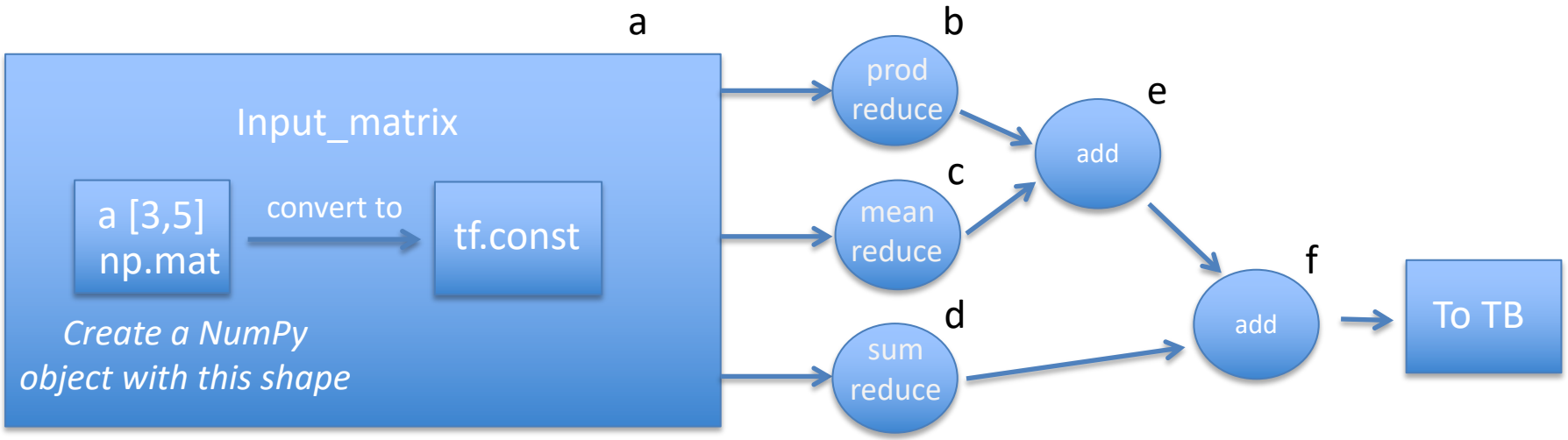
X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

TensorFlow

HW2:

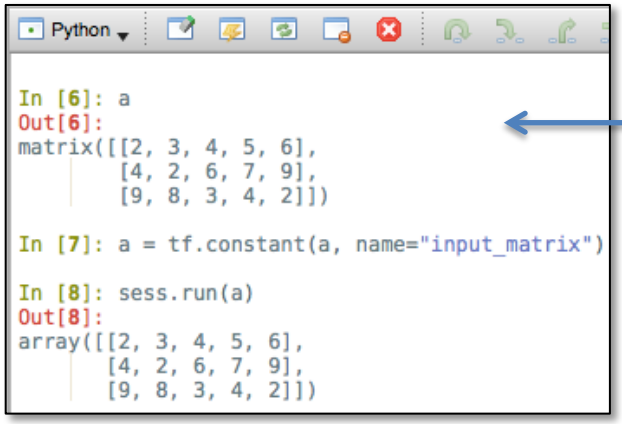
recreate the graph and visualize it in Tensorboard

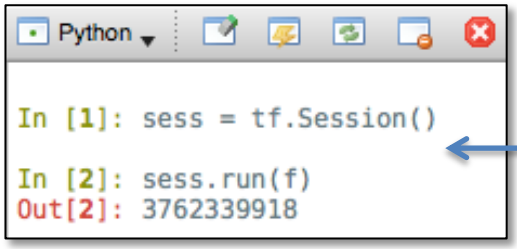


TensorFlow

- HW2 *solution*: *recreate the graph and visualize it in Tensorboard*

solution



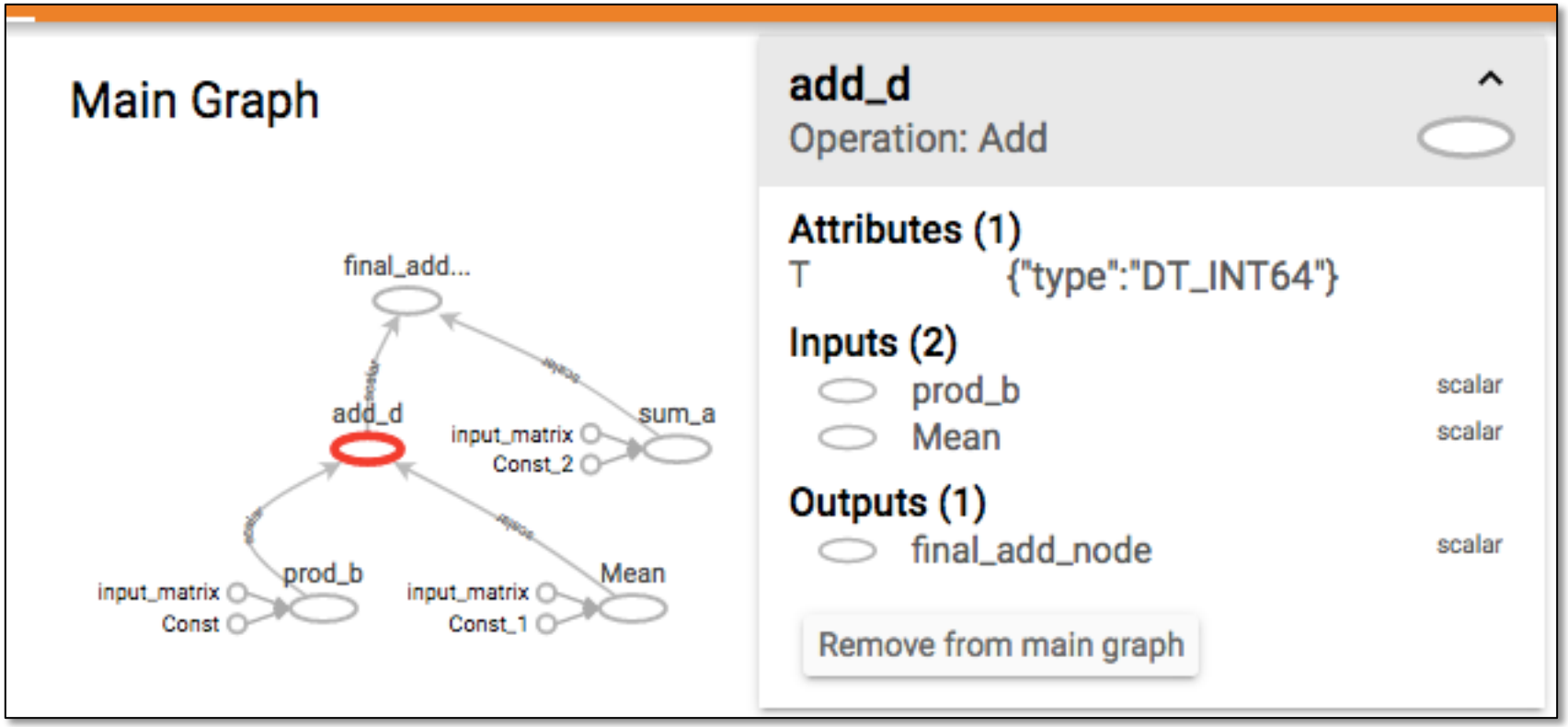



```
1  ## Building a matrix input then create a TensorFlow graph:
2
3  # First we need to import TensorFlow and NumPy:
4  import tensorflow as tf
5  import numpy as np
6
7  # Create a Numpy matrix:
8  a = np.matrix([[2,3,4,5,6],[4,2,6,7,9],[9,8,3,4,2]])
9
10 # Defining our single input node:
11 a = tf.constant(a, name="input_matrix")
12
13 # Defining nobe 'b':
14 b = tf.reduce_prod(a, name="prod_b")
15
16 # Defining the next two nodes in our graph:
17 c = tf.reduce_mean(a)
18
19 # Define the 'sum' reducer node for a:
20 d = tf.reduce_sum(a, name="sum_a")
21
22 # This last line defines the final node in our graph:
23 e = tf.add(b,c, name="add_d")
24
25 # Create our final 'add' node:
26 f = tf.add(e,d, name="final_add_node")
27
28 # To run we have to add the two extra lines or run them in the shell:
29 sess = tf.Session()
30 sess.run(f)
31
32 # To create the graph:
33 sess.graph.as_graph_def()
34 file_writer = tf.summary.FileWriter('./', sess.graph)
35
36 # We clean up before we exit:
37 file_writer.close()
38 sess.close()
```

TensorFlow

- HW2 *solution*: *recreate the graph and visualize it in Tensorboard*

solution



Course Content Outline

- **Machne Learning**
- Linear and Logistic Regression
- Softmax classification
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation

- **Neural Networks**
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions
- Common layers: Conv and Pooling Layers
- CNN Overview

Midterm / Project proposal due (30pts)

- **Working with images**
- Normlization
- Loading Images
- Image formats and manipulation

- **CNN Implementation**
- Training
- Recurrent Neural Network (RNN)
- Project Presentations 1/2

- **Project Presentations**
- Project Presentation 2/2

Final Project (40pts)

Machine Learning

- For the rest of the course we will focus on supervised learning problems, where we train an inference model with an input dataset, along with the real or expected output for each example.

- The model will cover a dataset and then be able to predict the output for new inputs that don't exist in the original training dataset.

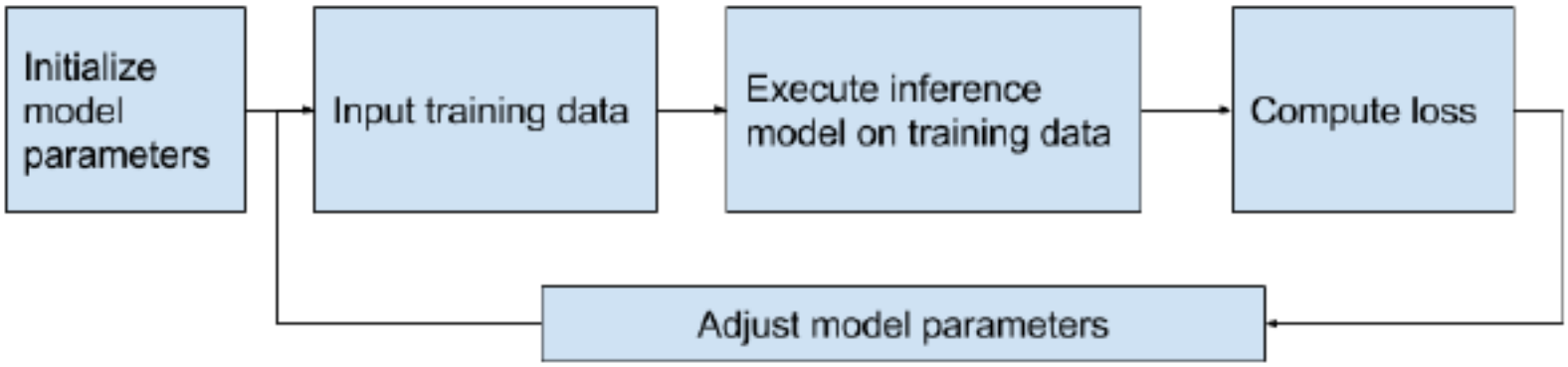
- An inference model is a series of mathematical operations that we apply to our data.

Machine Learning

- We create a training loop that:
  - Initializes the model parameters for the first time
  - Reads the training data along with the expected output data for each data example
  - Executes the inference model on the training data, so it calculates for each training input example
  - Computes the loss
  - Adjusts the model parameters

Machine Learning

- The models may vary significantly in:
  - the number of operations they use (multiply, add, etc.)
  - the way they combine
  - the number of parameters they have
- Regardless, we always apply the same general structure for training them:



Machine Learning

- The loop repeats this process through several cycles, according to:
  - the learning rate that we need to apply, and
  - depending on the model and data we input to it.
- After training, we apply an evaluation phase:
  - we execute the inference against a different set data to which we also have the expected output and evaluate the loss for it.

Machine Learning

- Given how this dataset contains examples unknown for the model, the evaluation tells us how well the model predicts beyond its training.

- A very common practice is to take the original dataset and randomly split it in 70% of the examples for training, and 30% for evaluation

- Let's use this structure to define some generic frame for the model code:

Machine Learning

```
import tensorflow as tf

# initialize variables/model parameters

# define the training loop operations
def inference(X):
    # compute inference model over data X and return the result

def loss(X, Y):
    # compute loss over training data X and expected outputs Y

def inputs():
    # read/generate input training data X and expected outputs
Y

def train(total_loss):
    # train / adjust model parameters according to computed to-
tal loss

def evaluate(sess, X, Y):
    # evaluate the resulting trained model
```

Machine Learning

```
# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:

    tf.initialize_all_variables().run()

    X, Y = inputs()

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

    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=co-
ord)

    # actual training loop
    training_steps = 1000
    for step in range(training_steps):
        sess.run([train_op])
        # for debugging and learning purposes, see how the
loss gets decremented thru training steps
```

Machine Learning

```
# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:

    tf.initialize_all_variables().run()

    X, Y = inputs()

    tc
    tr

    co
    th
ord)

    # actual training loop
    training_steps = 1000
    for step in range(training_steps):
        sess.run([train_op])
        # for debugging and learning purposes, see how the
loss gets decremented thru training steps
```

```
In [10]: tf.train.start_queue_runners?
Signature: tf.train.start_queue_runners(sess=None, coord=None, daemon=True,
Docstring:
Starts all queue runners collected in the graph.

This is a companion method to `add_queue_runner()`.  It just starts
threads for all queue runners collected in the graph.  It returns
the list of all threads.
```

Machine Learning

```
        if step % 10 == 0:
            print "loss: ", sess.run([total_loss])

    evaluate(sess, X, Y)

    coord.request_stop()
    coord.join(threads)
    sess.close()
```

Saving training checkpoints

- A word about queuing and threading in TensorFlow:
  - One of the best things about TensorFlow is the possibility to create multiple threads hence allow asynchronous operations
  - When we have large datasets the training process of our models will get much faster
  - In particular queuing and threading are very handy when processing our training data in mini-batches and doing the following operations:
    - *reading, extracting and pre-processing*

Saving training checkpoints

- A word about queuing and threading in TensorFlow:
  - Here are the important queuing operations of interest:
    - *FIFOQueue* – creates a First In First Out object
    - *RandomShuffleQueue* – creates randomized data-batches
    - *QueueRunner* – control the asynchronous execution of enqueue operations
    - *Coordinator* – making sure that all the threads we created will stop at the same time
    - *string_input_producer* – takes a list of filenames and creates a ***FIFOQueue***
    - *shuffle_batch* – creates a ***RandomShuffleQueue*** with batch-sized dequeuing
  - These packages are at the core of TensorFlow's efficient data consumption pipelines and supply data in more professional way as opposed using dictionaries as before:

```
56 with tf.Session() as session:
57   result = cross_validate(session)
58   print "Cross-validation result: %s" % result
59   print "Test accuracy: %f" % session.run(accuracy, feed_dict={x: test_x, y: test_y})
```

Saving training checkpoints

- A word about queuing and threading in TensorFlow:
  - FIFOQueue:


```
q = tf.FIFOQueue(3, "float")
init = q.enqueue_many(([0.,0.,0.],))

x = q.dequeue()
y = x+1
q_inc = q.enqueue([y])

init.run()
q_inc.run()
q_inc.run()
q_inc.run()
q_inc.run()
```

Saving training checkpoints

- A word about queuing and threading in TensorFlow:
  - The code:

```
## Queuing and Threading:
import tensorflow as tf

dummy_input = tf.random_normal([8], mean=0.5, stddev=2)
dummy_input = tf.Print(dummy_input, data=[dummy_input],
                       message='New dummy inputs have been created: ', summarize=6)
q = tf.FIFOQueue(capacity=3, dtypes=tf.float32)
enqueue_op = q.enqueue_many(dummy_input)
# now setup a queue runner to handle enqueue_op outside of the main thread asynchronously
qr = tf.train.QueueRunner(q, [enqueue_op] * 10)
tf.train.add_queue_runner(qr)

data = q.dequeue()
data = tf.Print(data, data=[q.size(), data], message='This is how many items are left in q: ')
# create a fake graph that we can call upon
fg = data + 1
```

Saving training checkpoints

- A word about queuing and threading in TensorFlow:
  - The code:

```
18 with tf.Session() as sess:
19     coord = tf.train.Coordinator()
20     threads = tf.train.start_queue_runners(coord=coord)
21     # now dequeue a few times, and we should see the number of items
22     # in the queue decrease
23     sess.run(fg)
24     sess.run(fg)
25     sess.run(fg)
26     # we have a queue runner on another thread making sure the queue is
27     # filled asynchronously
28     sess.run(fg)
29     sess.run(fg)
30     sess.run(fg)
31     # this will print, but not necessarily after the 6th call of sess.run(fg)
32     # due to the asynchronous operations
33     print("We're here!")
34     # we have to request all threads now stop, then we can join the queue runner
35     # thread back to the main thread and finish up
36     coord.request_stop()
37     coord.join(threads)
```

Saving training checkpoints

- A word about queuing and threading in TensorFlow:
  - The output:

…

```
New dummy inputs have been created: [-0.359175324 1.87123322 1.58081055 -3.31081557 2.14717 2.07823706...]
New dummy inputs have been created: [1.9641149 -1.37467635 3.45695853 0.471189022 -2.74250507 2.84581065...]
This is how many items are left in q: [3][3.31531882]
This is how many items are left in q: [3][2.10265017]
This is how many items are left in q: [3][-3.50813627]
This is how many items are left in q: [3][1.26989508]
This is how many items are left in q: [3][1.25680184]
New dummy inputs have been created: [-0.120787382 -0.794406772 0.913305461 1.24941087 -1.78667855 1.41696572...]
This is how many items are left in q: [3][-3.16560888]
We're here!
```

Saving training checkpoints

- As we already stated, training models implies updating their parameters, or variables in Tensorflow, through many training cycles
- Variables are stored in memory, so if the computer would lose power after many hours of training, we would lose all that work, so …
- We use tf.train.Saver class to save the graph variables in proprietary binary files
- We should periodically save the variables, create a checkpoint file, and eventually restore the training from the most recent checkpoint if needed.

Saving training checkpoints

- In order to use the Saver we need to slightly change the training loop scaffolding code:

```
# model definition code ...

# Create a saver.
saver = tf.train.Saver()

# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:

    # model setup....

    # actual training loop
    for step in range(training_steps):
        sess.run([train_op])

        if step % 1000 == 0:
            saver.save(sess, 'my-model', global_step=step)

    # evaluation...

    saver.save(sess, 'my-model', global_step=training_steps)

    sess.close()
```

Saving training checkpoints

- If we wish to recover the training from a certain point, we should use the tf.train.get_checkpoint_state() method
- It will verify if we already have a checkpoint saved, and the tf.train.Saver.restore() method to recover the variable values

```
with tf.Session() as sess:

    # model setup....

    initial_step = 0

    # verify if we don't have a checkpoint saved already
    ckpt = tf.train.get_checkpoint_state(os.path.dir-
name(__file__))
    if ckpt and ckpt.model_checkpoint_path:
        # Restores from checkpoint
        saver.restore(sess, ckpt.model_checkpoint_path)
        initial_step =
int(ckpt.model_checkpoint_path.rsplit('-', 1)[1])

    #actual training loop
    for step in range(initial_step, training_steps):
       ...
```

Linear models for classification

- Binary classification
- Line separates the two classes
  - Decision boundary - defines where the decision changes from one class value to the other
- Prediction is made by plugging in observed values of the attributes into the expression
  - Predict one class if output ≥ 0, and the other class if output < 0
- Boundary becomes a high-dimensional plane (*hyperplane*) when there are multiple attributes

Linear Models

- Linear regression is the simplest form of modeling for a supervised learning problem
- Primarily used for regression:
  - Inputs (attribute values) and output are all numeric
- Output is the sum of the weighted input attribute values
- The trick is to find good values for the weights
- There are different ways of doing this, which we will consider later:
  - the most famous one is to minimize the squared error

Example: Predicting CPU performance

- The table shows some data for which both the outcome and the attributes are numeric:

	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
...							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

Example: Predicting CPU performance

- It concerns the relative performance of computer processing power on the basis of a number of relevant attributes:
  - each row represents one of 209 different computer configurations.

	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
…							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

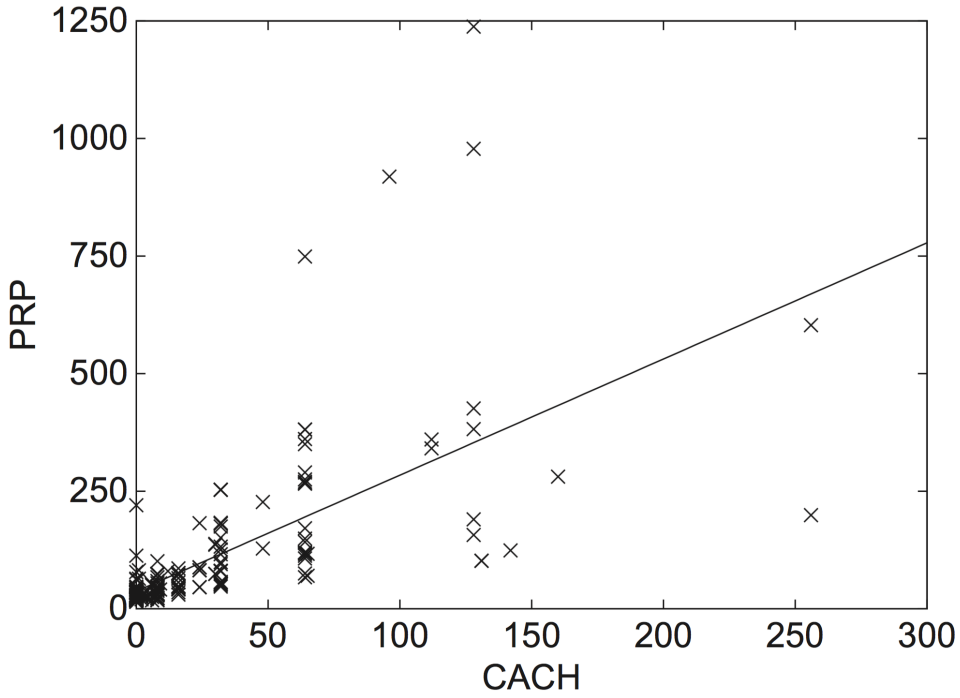
Example: Predicting CPU performance

- The classic way of dealing with continuous prediction is to write the outcome as a linear sum of the attribute values with appropriate weights, in this *linear regression function*:

```
PRP = -55.9 + 0.0489 MYCT + 0.0153 MMIN + 0.0056 MMAX
      + 0.6410 CACH - 0.2700 CHMIN + 1.480 CHMAX
```

	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
…							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

A linear regression function for the CPU performance data



PRP = 37.06 + 2.47CACH

- Given a set of data points as training data, you are going to find the linear function that best fits them.
- In a 2-dimensional dataset, this type of function represents a straight line.
- The x-marks are the training data points and the line is the what the model will infer (predict).

TensorFlow

- Working with Linear regression
  - What is Linear regression?
    - It is a standard technique for numeric prediction
    - It is the simplest form of modeling for a supervised learning problem
    - The outcome is linear combination of attributes

$$x = w_0 + w_1 a_1 + w_2 a_2 + \cdots + w_k a_k$$

    - Weights are calculated from the training data
    - Predicted value for first training instance $\mathbf{a}^{(1)}$

$$w_0 a_0^{(1)} + w_1 a_1^{(1)} + w_2 a_2^{(1)} + \cdots + w_k a_k^{(1)} = \sum_{j=0}^{k} w_j a_j^{(1)}$$

(assuming each instance is extended with a constant attribute with value 1)

TensorFlow

- Working with Linear regression
  - Another general representation of a linear function is:

$$y(x_1, x_2, , x_k) = w_1x_1 + w_2x_2 + ... + w_kx_k + b$$

  - And its matrix (or tensor) form is:

$$Y = XW^T + b \text{ where } X = (x_1, ..., x_k)\ W = (w_1, ..., w_k)$$

    - Y is the value we are trying to predict.
    - $x_1, ..., x_k$ independent or predictor variables are the values that we provide when using our model for predicting new values. In matrix form, you can provide multiple examples at once- one per row.
    - $w_1, ..., w_k$ are the parameters the model will learn from the training data, or the "weights" given to each variable.
    - b is also a learned parameter- the constant of the linear function that is also known as the bias of the model.

TensorFlow

- Instead of transposing weights, we can simply define them as a single column vector

- Here is a sample model:

```
# initialize variables/model parameters
W = tf.Variable(tf.zeros([2, 1]), name="weights")
b = tf.Variable(0., name="bias")


def inference(X):
    return tf.matmul(X, W) + b
```

TensorFlow

- Now we must define how to compute the loss

- For a simple model we can use the squared error:
  - It sums the squared difference of all the predicted values for each training example with their corresponding expected values.
  - Algebraically it is the squared Euclidean distance between the predicted output vector and the expected one.

- Graphically in a 2d dataset the error is the length of the vertical line that you can trace from the expected data point to the predicted regression line.

TensorFlow

- It is also known as L2 norm or L2 loss function.
- We use it squared to avoid computing the square root, since it makes no difference for trying to minimize the loss and saves a computing step:

$$loss = \sum_i (y_i - y\_predicted_i)^2$$

- We sum over *i*, where *i* is each data example. In code:

```
def loss(X, Y):
    Y_predicted = inference(X)
    return tf.reduce_sum(tf.squared_difference(Y, Y_predicted))
```

TensorFlow

- Next, we define the model training operation.

- We will use the gradient descent algorithm for optimizing the model parameters:

```
def train(total_loss):
    learning_rate = 0.0000001
    return tf.train.GradientDescentOptimizer(learn-
ing_rate).minimize(total_loss)
```

- When you run it, you will see printed how the loss gets smaller on each training step.

TensorFlow

- Now that we trained the model, it's time to evaluate it:

```
def evaluate(sess, X, Y):
    print sess.run(inference([[80., 25.]])) # ~ 303
    print sess.run(inference([[65., 25.]])) # ~ 256
```

- As a quick evaluation, you can check that the model learned how the blood fat decays with weight

- Also, the output values are in-between the boundaries of the original trained values.

TensorFlow

- Let's train 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

-

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	

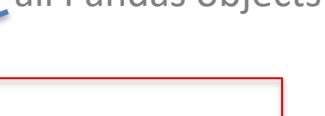
Data source: http://people.sc.fsu.edu/~jburkardt/datasets/reg

	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

```
# 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`.
```



```
38  # Using training, we adjust the model parameters:
39  def train(total_loss):
40      learning_rate = 0.000001
41      return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)
42
43  # We evaluate the resulting model:
44  def evaluate(sess, X, Y):
45      print(sess.run(inference([[55., 40.]]))) # ~ 295 (but it is 303)
46      print(sess.run(inference([[50., 70.]]))) # ~ 256 (other values not in table)
47      print(sess.run(inference([[90., 20.]]))) # ~ 303 ( ... )
48      print(sess.run(inference([[90., 70.]]))) # ~ 256 ( ... )
49
50  # Launch the graph in a session and run the training loop:
51  with tf.Session() as sess:
52
53      tf.initialize_all_variables().run()
54
55      X, Y = inputs()
56      total_loss = loss(X, Y)
57      train_op = train(total_loss)
58
59      # Actual training loop:
60      training_steps = 10000
61      for step in range(training_steps):
62          sess.run([train_op])
63          # See how the loss gets decremented thru training steps:
64          if step % 1000 == 0:
65              print("Epoch:", step, " loss: ", sess.run(total_loss))
66
67      print("Final model W=", sess.run(W), "b=", sess.run(b))
68      evaluate(sess, X, Y)
69
70      sess.close()
```

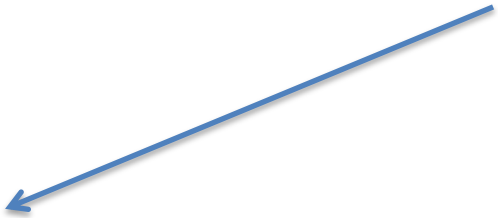
```python
# Using training, we adjust the model parameters:
def train(total_loss):
    learning_rate = 0.000001
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)

# 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

- Data file .csv:

```
In [48]: sess=tf.Session()

In [49]: sess.run(X)
Out[49]:
array([[84., 46.],
       [73., 20.],
       [65., 52.],
       [70., 30.],
       [76., 57.],
          ...
       [85., 37.],
       [55., 40.],
       [63., 30.]], dtype=float32)

In [50]: sess.run(Y)
Out[50]:
array([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.], dtype=float32)
```



	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
27			

LINEAR REGRESSION IN TF 1.X

WITH CONSTANTS:

eagerexecution.py | lin_reg.py | lin_reg2.0.py | lin_reg2.0constants.py | lin_regconstants.py

```
"""
@author: ailiev
"""
import tensorflow as tf
basic_addition_graph= tf.Graph()
with basic_addition_graph.as_default():
    a = tf.constant(value=[2, 5, 7],
                    name="a",
                    dtype=tf.int8,
                    shape=(3, ),
                    verify_shape=True)
    b = tf.constant(value=[1, -2, -4],
                    name="b",
                    dtype=tf.int8,
                    verify_shape=True)
    c = tf.constant(value=[[ -4, -1, 2], [ -4, -1, 2]],
                    name="c",
                    dtype=tf.int8,
                    shape=(2,3),
                    verify_shape=True)

    x = tf.add(a, b)
    y= tf.multiply(x , c) # BROADCASTING, (3, ) X (2, 3)

# CREATE A SESSION TO RUN THE GRAPH
with tf.Session(graph=basic_addition_graph) as sess:
    print('the tensorflow version is:',tf.__version__)
    print(sess. run([x, y])) # Providing No feed dict, No Placeholder in the graph
```

Output in TF 1.X:

```
the tensorflow version is: 1.14.0
[array([3, 3, 3], dtype=int8), array([[-12,  -3,   6],
       [-12,  -3,   6]], dtype=int8)]

In [3]:
```

CODE EXAMPLE FOR LINEAR REGRESSION IN TF 2.0

The complicated code which was written in TF 1.x is simplified below in TF 2.0:

eagerexecution.py | lin_reg.py | lin_reg2.0.py | lin_reg2.0constants.py

```
"""
@author: ailiev
"""
import tensorflow as tf

x = tf.random.normal(shape=(3,3))

y = tf.random.normal(shape=(3,3))

z = tf.ones(shape=(1))

result = z + x * y
print('the tensorflow version is:',tf.__version__)

print(result)
```

The output fromTF 1.14 can also be obtained in TF 2.0, but in the form of matrix by using the simplified code:

```
the tensorflow version is: 2.0.0-rc1
tf.Tensor(
[[0.56481594 0.948835   0.9111127 ]
 [2.5992794  0.6788218  0.8626047 ]
 [1.3522041  0.20513034 0.43107367]], shape=(3, 3), dtype=float32)

In [4]:
```

LINEAR REGRESSION IN TF 1.X

WITH VARIABLES:

- Variable is a class which has lot of functionality in itself.

eagerexecution.py | lin_reg.py | lin_reg2.0.py | lin_reg2.0c

```
"""
@author: ailiev
"""
import tensorflow as tf
import numpy as np
#CREATE GRAPH
basic_graph = tf.Graph()
with basic_graph.as_default ():
    a= tf.placeholder(name="a",
                      dtype=tf.float32,
                      shape= (None, 5))
    W=tf.Variable(tf.random_normal( (5, 3)), name= "W") # INITIALIZE IT WITH RANDOM
    b=tf.Variable(tf.random_normal( (3,)), name="b") # INITIALIZE IT WITH RANDOM
    y= tf.add(tf.matmul (a, W), b)
#CREATE A SESSION TO RUN THE GRAPH
with tf.Session(graph=basic_graph) as sess:
    #wri ter = tf.s ummary.F1lewri ter( 'basiC addi tion graph, sess.graph)
    # tensorboard-logdira"basiC addition graph"-port 8800
    init = tf.global_variables_initializer()
    sess.run(init)
    print('the tensorflow version is', tf.__version__)
    print(sess.run(y, feed_dict={a: np.random.uniform (size= (500, 5)) } ))
    print('-' * 20)
    print(b.eval())
```

Output in TF 1.X:

```
the tensorflow version is 1.14.0
[[-1.638287    3.7896645  -3.4496527 ]
 [-0.14133322  0.34866196 -1.270206  ]
 [-1.6203686   1.9089545  -1.4133935 ]
 ...
 [-2.400795    3.7269175  -3.084959  ]
 [-2.0975566   3.7196305  -2.9466667 ]
 [ 0.21348056  1.1293043  -2.56114   ]]
--------------------
[ 0.5004295  -0.08231466 -1.4022843 ]

In [6]:
```

LINEAR REGRESSION IN TF 2.0

- As the 2.0 has the Eager Execution, we can skip the graph, session code

Output in TF 2.0:

```
the tensorflow version is 2.0.0-rc1
tf.Tensor(
[[-2.4430084   4.3219256   3.667462  ]
 [ 3.9929757  -3.1744885   2.0444593 ]
 [ 6.417946   -0.07052445 -3.2881093 ]
 ...
 [ 3.3592832   0.348719    0.7960553 ]
 [-4.073365   -0.01392078  5.558447  ]
 [ 2.6047485  -1.4717494   1.2911493 ]], shape=(500, 3), dtype=float32)

In [5]:
```

eagerexecution.py | lin_reg.py | lin_

```
"""
@author: ailiev

"""
import tensorflow as tf
#RANDOm TO SOMETHING USEFUL
x = tf.random.normal(shape= (500,8)) # 508 Examples, 8 Features.
w = tf.Variable(tf.random.normal(shape=(8, 3)))
#3classes
b = tf.Variable(tf.random.normal(shape= (3,)))

y_pred = tf.matmul (x, w) + b

print('the tensorflow version is',tf.__version__)
print(y_pred)
```

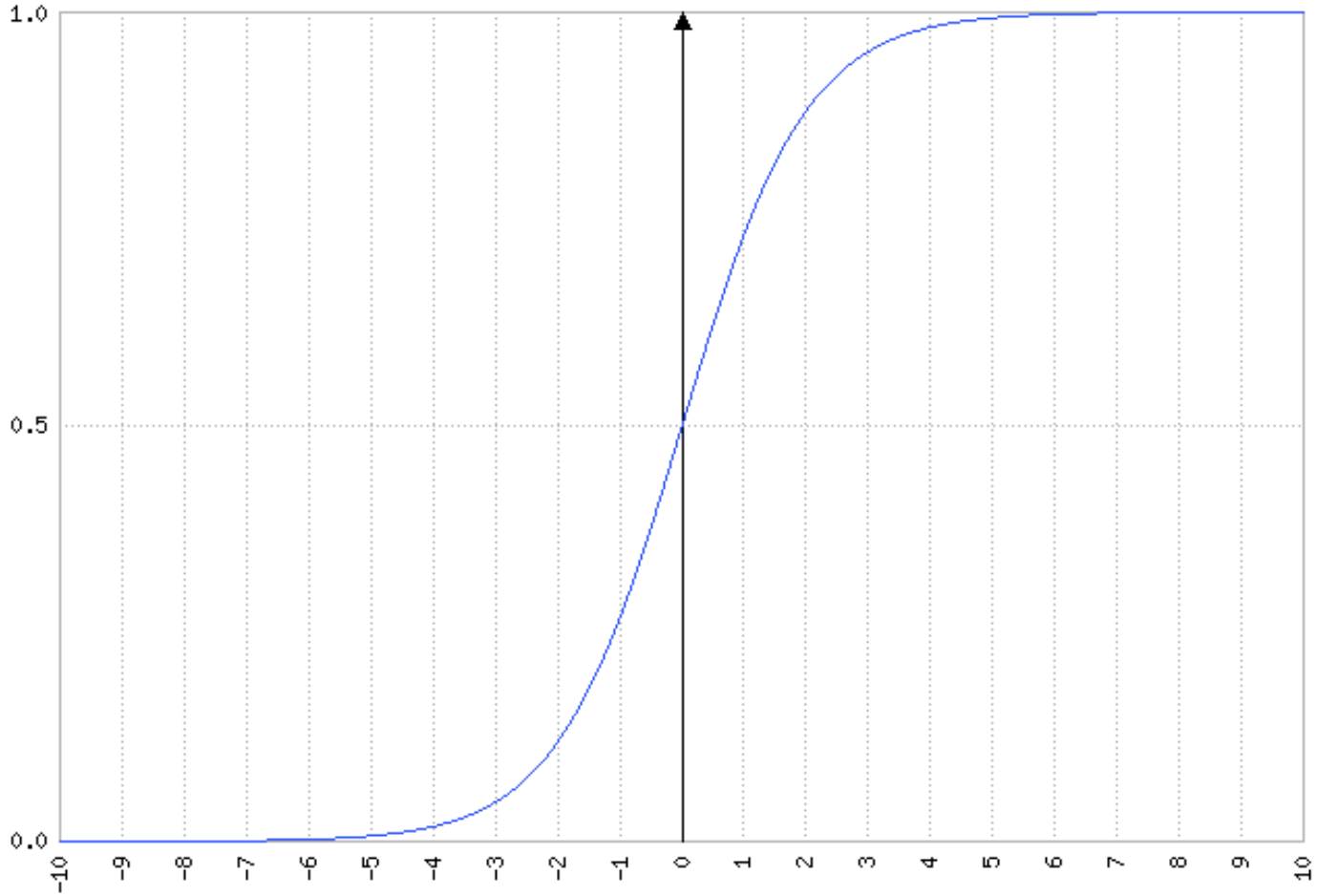
TensorFlow

- Working with Logistic regression
  - The linear regression model predicts a continuous value, or any real number.
  - Now we are going to present a model that can answer a yes-no type of question, like “Is this email spam?”

- There is a function used commonly in machine learning called the logistic function. It is also known as the *sigmoid function*, because its shape is an S (and sigma is the Greek letter equivalent to s).

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

TensorFlow



TensorFlow

- In order to feed the function with the multiple dimensions, or features from the examples of our training datasets, we need to combine them into a single value

```
# same params and variables initialization as log reg.
W = tf.Variable(tf.zeros([5, 1]), name="weights")
b = tf.Variable(0., name="bias")

# former inference is now used for combining inputs
def combine_inputs(X):
    return tf.matmul(X, W) + b

# new inferred value is the sigmoid applied to the former
def inference(X):
    return tf.sigmoid(combine_inputs(X))
```

TensorFlow

- Logistic regression: *example*

```
1  import numpy as np
2  import tensorflow as tf
3  import matplotlib.pyplot as plt
4
5  learning_rate = 0.01
6  training_epochs = 500
7
8  # Defining the sigmoid function:
9  def sigmoid(x):
10     return 1. / (1. + np.exp(-x))
11
12 # Create our data points on the x and y axis:
13 x1 = np.random.normal(5, 3, 100)
14 x2 = np.random.normal(-5, 3, 100)
15 xs = np.append(x1, x2)
16 ys = np.asarray([0.] * len(x1) + [1.] * len(x2))
17 plt.scatter(xs, ys)
18
19 # Create our parameters and placeholders for X and Y to feed them with the data above:
20 X = tf.placeholder(tf.float32, shape=(None,), name="x")
21 Y = tf.placeholder(tf.float32, shape=(None,), name="y")
22 w = tf.Variable([0., 0.], name="parameter", trainable=True)
23 y_model = tf.sigmoid(-(w[1] * X + w[0]))
```

TensorFlow

- Logistic regression: *example*

```python
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
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]))
```

```
(<module>)>>> sess.run(y_model, feed_dict = {X: xs})
array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
       0.5, 0.5, 0.5, 0.5, 0.5], dtype=float32)
```

TensorFlow

- Logistic regression: *example*

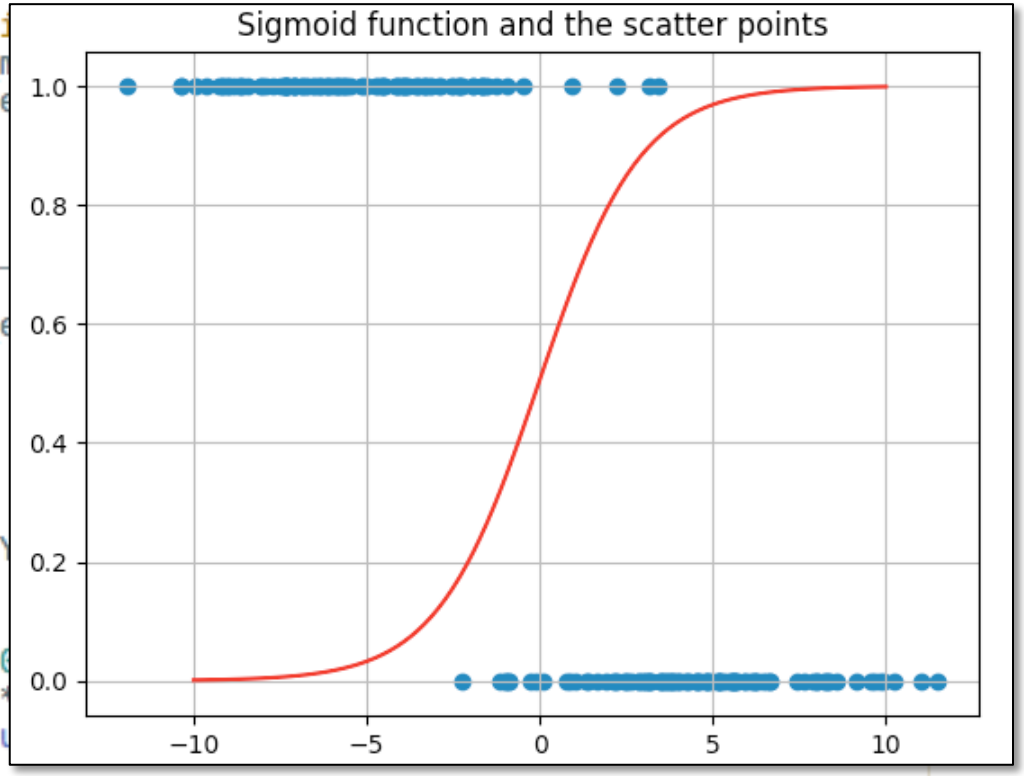
```
25  # Calculate the cost and adaptation (learning):
26  cost = tf.reduce_mean(-tf.log(y_model * Y + (1 - y_model) * (1 - Y)))
27  train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
28
29  # Run the model:
30  with tf.Session() as sess:
31      sess.run(tf.global_variables_initializer())
32      prev_err = 0
33      for epoch in range(training_epochs):
34          err, _ = sess.run([cost, train_op], {X: xs, Y: ys}) # err = cost
35          print(epoch, err)
36          if abs(prev_err - err) < 0.0001: # adjust to see curve change with epochs
37              break   # Check when the error is small enough to quit
38          prev_err = err
39      w_val = sess.run(w, {X: xs, Y: ys})
40
41  # Plot the resulting sigmoid:
42  all_xs = np.linspace(-10, 10, 100)
43  plt.plot(all_xs, sigmoid(all_xs * w_val[1] + w_val[0]), 'r') # calculate the sigmoid
44  plt.grid(), plt.title("Sigmoid function and the scatter points")
45  plt.pause(1)
```

TensorFlow

- Logistic regression: *example*

```
25  # Calculate the cost and adaptati
26  cost = tf.reduce_mean(-tf.log(y_m
27  train_op = tf.train.GradientDesce
28
29  # Run the model:
30  with tf.Session() as sess:
31      sess.run(tf.global_variables_
32      prev_err = 0
33      for epoch in range(training_e
34          err,   = sess.run([cost,
35          print(epoch, err)
36                                    err) <
37                                   ck when
38
39      w_va                        X: xs, Y
40
41  # Plot t                       oid:
42  all_xs =                        10, 100
43  plt.plot                       all_xs *
44  plt.grid                       gmoid fu
45  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
```



TensorFlow

- Consider an example where the expected answer is “yes” and the model is predicting a very low probability for it, close to 0.
- This means that it is close to 100% hence the answer is “no”
- The squared error penalizes such a case with the same order of magnitude for the loss as if the probability would have been predicted as 20, 30, or even 50% for the “no” output
- There is a loss function that works better for this type of problem, which is the cross entropy function

$$loss = -\sum_i (y_i \,.\, log(y\_predicted_i) + (1 - y_i) \,.\, log(1 - y\_predicted_i))$$

Criterion for attribute selection

- Which is the best attribute?
  - Want to get the smallest tree
  - Heuristic: choose the attribute that produces the "purest" nodes
- Popular selection criteria: *information gain*
  - Information gain increases with the average purity of the subsets
- Strategy: amongst attributes available for splitting, choose attribute that gives greatest information gain
- Information gain requires measure of *impurity*
- Impurity measure that it uses is the *entropy* of the class distribution, which is a measure from information theory

What is Entropy?

- *Various definitions same meaning:*
  - *Entropy*: lack of order or predictability; gradual decline into disorder
  - *Entropy*: (in information theory) a logarithmic measure of the rate of transfer of information in a particular message or language
  - *Entropy* may be understood as a measure of disorder within a macroscopic system
  - *Entropy* is the measure of the level of disorder in a closed but changing system, a system in which energy can only be transferred in one direction from an ordered state to a disordered state
  - The higher the *entropy*, the higher the disorder and the system's energy to do useful work is lower

Computing information

- We have a probability distribution: the class distribution in a subset of instances
- The expected information required to determine an outcome (i.e., class value), is the distribution's *entropy*
- Formula for computing the entropy:

$$\text{Entropy}(p_1, p_2, \ldots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \ldots - p_n \log p_n$$

- Using base-2 logarithms, entropy gives the information required in expected *bits*
- *Entropy is maximal when all classes are equally likely and minimal when one of the classes has probability 1*

Example: attribute *Outlook*

- *Outlook* = *Sunny* :

$$\text{Info}([2,3]) = 0.971 \text{ bits}$$

$$Entropy(S_{sunny}) = -\frac{2}{5}\log_2\left(\frac{2}{5}\right) - \frac{3}{5}\log_2\left(\frac{3}{5}\right)$$

- *Outlook* = *Overcast* :

$$\text{Info}([4,0]) = 0.0 \text{ bits}$$

$$Entropy(S_{overcast}) = -\frac{4}{4}\log_2\left(\frac{4}{4}\right) - 0\log_2(0)$$

- *Outlook* = *Rainy* :

$$\text{Info}([3,2]) = 0.971 \text{ bits}$$

$$Entropy(S_{rain}) = -\frac{3}{5}\log_2\left(\frac{3}{5}\right) - \frac{2}{5}\log_2\left(\frac{2}{5}\right)$$

- Expected information for attribute:

$$\text{Info}([2,3],[4,0],[3,2]) = (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971$$

$$= 0.693 \text{ bits}$$

Computing information gain

- Information gain: information before splitting – information after splitting

Gain (*Outlook*) = Info([9,5]) – info([2,3],[4,0],[3,2])
= 0.940 – 0.693
= 0.247 bits

- Information gain for attributes from weather data:

Gain (*Outlook*)	= 0.247 bits
Gain (*Temperature*)	= 0.029 bits
Gain (*Humidity*)	= 0.152 bits
Gain (*Windy*)	= 0.048 bits

TensorFlow

- There is a Tensorflow method for calculating cross entropy directly for a sigmoid output in a single, optimized step:

```
def loss(X, Y):
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_log-
its(combine_inputs(X), Y))
```

- Cross entropy meaning:
  - *In information theory, Shannon entropy allows to estimate the average minimum number of bits needed to encode a symbol s_i from a string of symbols, based on the probability p_i of each symbol to appear in that string.*

$$H = -\sum_i (p_i . log_2(p_i))$$

TensorFlow

- Cross entropy example:
  - *Task:* calculate the entropy for the word “HELLO.”
  - *Solution:*

$$p(H) = p(E) = p(O) = 1/5 = 0.2$$

$$p(L) = 2/5 = 0.4$$

$$H = -3 \cdot 0.2 \cdot \log_2(0.2) - 0.4 \cdot \log_2(0.4) = 1.92193$$

  - *Conclusion:* we need 2 bits per symbol to encode “HELLO” in the optimal encoding.

TensorFlow

- Let's apply the model to real data using the Titanic survivor contest dataset from:

https://www.kaggle.com/c/titanic/data

- We need to load the data first, so download the train.csv file

- You can load and parse it, and create a batch to read many rows packed in a single tensor for computing the inference efficiently: *… code on next slide*

TensorFlow

```
def read_csv(batch_size, file_name, record_defaults):
    filename_queue = tf.train.string_input_producer([
    os.path.join(os.getcwd(), file_name)])

    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)

    # decode_csv will convert a Tensor from type string (the
text line) in
    # a tuple of tensor columns with the specified defaults,
which also
    # sets the data type for each column
    decoded = tf.decode_csv(value, record_defaults=record_de-
faults)

    # batch actually reads the file and loads "batch_size"
rows in a single tensor
    return tf.train.shuffle_batch(decoded,
                                  batch_size=batch_size,
                                  capacity=batch_size * 50,
                                  min_after_dequeue=batch_size)
```

TensorFlow

- The model will have to infer, based on the passenger *age*, *sex* and *ticket* class if the passenger **survived** or **not**

- We must use categorical data from this dataset:
  - Ticket class and gender are string features with a predefined possible set of values that they can take
  - To use them in the inference model we need to convert them to numbers
  - A naive approach might be to assign a number for each possible value: For instance, you can use “1” for first ticket class, “2” for second, and “3” for third *

TensorFlow

- When working with categorical data, convert it to multiple boolean features, one for each possible value.
- This allows the model to weight each possible value separately. *
- That's because you can express a linear relationship between the values.
  - For instance if possible values are female = 1 and male = 0, then male = 1 - female , a single weight can learn to represent both possible states.

TensorFlow

```
def inputs():
    passenger_id, survived, pclass, name, sex, age, sibsp,
parch, ticket, fare, cabin, embarked = \
        read_csv(100, "train.csv", [[0.0], [0.0], [0], [""],
[""], [0.0], [0.0], [0.0], [""], [0.0], [""], [""]])

    # convert categorical data
    is_first_class = tf.to_float(tf.equal(pclass, [1]))
    is_second_class = tf.to_float(tf.equal(pclass, [2]))
    is_third_class = tf.to_float(tf.equal(pclass, [3]))

    gender = tf.to_float(tf.equal(sex, ["female"]))

    # Finally we pack all the features in a single matrix;
    # We then transpose to have a matrix with one example per
row and one feature per column.
    features = tf.transpose(tf.pack([is_first_class, is_sec-
ond_class, is_third_class, gender, age]))
    survived = tf.reshape(survived, [100, 1])

    return features, survived
```

TensorFlow

- Lets train our model now:

```
def train(total_loss):
    learning_rate = 0.01
    return tf.train.GradientDescentOptimizer(learn-
ing_rate).minimize(total_loss)
```

- To evaluate the results we are going to run the inference against a batch of the training set and count the number of examples that were correctly predicted. We call that measuring the accuracy

```
def evaluate(sess, X, Y):

    predicted = tf.cast(inference(X) > 0.5, tf.float32)

    print sess.run(tf.reduce_mean(tf.cast(tf.equal(predicted,
Y), tf.float32)))
```

Softmax classification

- With logistic regression we were able to model the answer to the yes-no question.
- Now we want to be able to answer a multiple-choice type of question like: “Were you born in Boston, London, or Sydney?”
- For that case there is the softmax function, which is a generalization of the logistic regression for C possible different values:

$$f(x)_c = \frac{e^{-x_c}}{\sum_{j=0}^{C-1} e^{-x_j}} \; for \; c = 0 \ldots C-1$$

Softmax classification

- Softmax function returns a probability vector of C components, filling the corresponding probability for each output class
- The sum of the C vector components always = 1 (*it is a probability*)
- To code this model, we will need to change from the previous models in the variable initialization
- Given that our model computes C outputs instead of just one, we need to have C different weight groups, one for each possible output
- So, we will use a weights matrix, instead of a weights vector
- That matrix will have one row for each input feature, and one column for each output class

Softmax classification

- Let's use the classical Iris flower dataset for trying softmax
- You can download it from here:

  https://archive.ics.uci.edu/ml/datasets/Iris

- The set contains:
  - 4 data features and
  - 3 possible output classes, one for each type of iris plant
  - so our weights matrix should have a 4x3 dimension

Softmax classification

- The variable initialization code should look like this:

```
# this time weights form a matrix, not a vector, with one "fea-
ture weights column" per output class.
W = tf.Variable(tf.zeros([4, 3]), name="weights")
# so do the biases, one per output class.
b = tf.Variable(tf.zeros([3], name="bias"))
```

- Tensorflow contains an embedded implementation of the softmax function:

```
def inference(X):
    return tf.nn.softmax(combine_inputs(X))
```

Softmax classification

- For a single training example *i*, cross entropy now becomes:

$$loss_i = -\sum_c y_c \cdot log(y\_predicted_c)$$

We are summing the loss for each output class on that training example

- Next, to calculate the total loss among the training set, we sum the loss for each training example:

$$loss = -\sum_i \sum_c y_{c_i} \cdot log\left(y\_predicted_{c_i}\right)$$

Softmax classification

- There are two versions implemented in Tensorflow for the softmax crossentropy function: *one specially optimized for training sets with a single class value per example **
- That function is:

tf.nn.sparse_softmax_cross_entropy_with_logits

```
def loss(X, Y):
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entro-
py_with_logits(combine_inputs(X), Y))
```

Softmax classification

- There are two versions implemented in Tensorflow for the softmax crossentropy function: *the other, lets you work with training sets containing the probabilities of each example to belong to every class **

- That function is:

  tf.nn.softmax_cross_entropy_with_logits

Softmax classification

- Let's define our input method. We will reuse the read_csv function from the logistic regression example, but will call it with the defaults for the values on our dataset, which are all numeric:

```
def inputs():

    sepal_length, sepal_width, petal_length, petal_width, la-
bel =\
        read_csv(100, "iris.data", [[0.0], [0.0], [0.0],
[0.0], [""]])
```

Softmax classification

```
    # convert class names to a 0 based class index.
    label_number = tf.to_int32(tf.argmax(tf.to_int32(tf.pack([
        tf.equal(label, ["Iris-setosa"]),
        tf.equal(label, ["Iris-versicolor"]),
        tf.equal(label, ["Iris-virginica"])
    ])), 0))

    # Pack all the features that we care about in a single ma-
trix;
    # We then transpose to have a matrix with one example per
row and one feature per column.
    features = tf.transpose(tf.pack([sepal_length, se-
pal_width, petal_length, petal_width]))

    return features, label_number
```

Softmax classification

- The training function is also the same

- For evaluation of accuracy, we need a slight change from the sigmoid version:

```
def evaluate(sess, X, Y):
    predicted = tf.cast(tf.arg_max(inference(X), 1), tf.int32)

    print sess.run(tf.reduce_mean(tf.cast(tf.equal(predicted,
Y), tf.float32)))
```

- Running the code should print an accuracy of about 96%.

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%

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

Project discussion

Show sample projects ...