Lecture 11

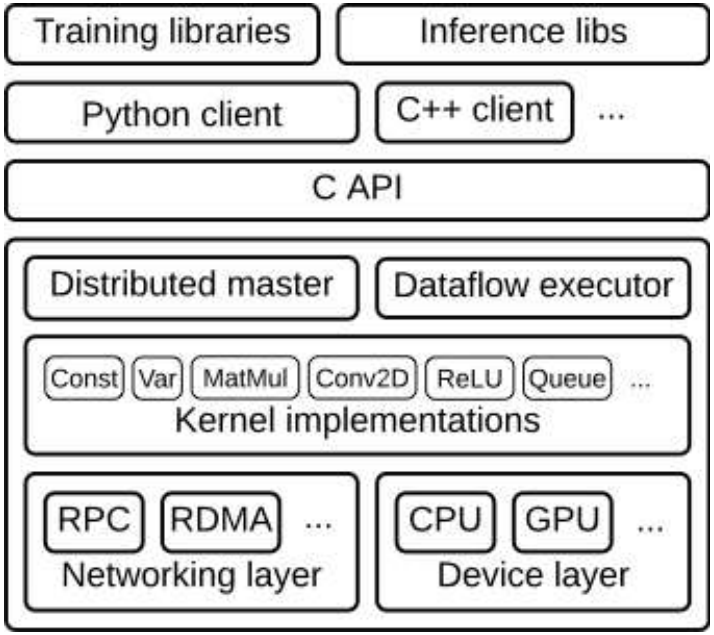
Tensor Flow, continued

Zoran B. Djordjević

TensorFlow Architecture

- A very legitimate question was raised about utility of TensorFlow. It is quite apparent that we could do all of our current homework calculations with Python, and sometime in a much more straightforward way. So, why bother.
- What we did was learning the syntax and usage patterns. TensorFlow was designed with bigger goals in mind.
- TensorFlow is designed for large-scale distributed training and inference, but it is also flexible enough to support experimentation with new machine learning models and system-level optimizations.

- The TensorFlow runtime is a cross-platform library.
- Figure on the right illustrates its general architecture.
- A C API separates user level code in different languages from the core runtime.



Key Components of Distributed TF

Client:

- Defines the computation as a dataflow graph.
- Initiates graph execution using a **session**

Distributed Master:

- Prunes a specific subgraph from the graph, as defined by the arguments to `Session.run()`.
- Partitions the subgraph into multiple pieces that run in different processes and devices.
- Distributes the graph pieces to worker services.
- Initiates graph piece execution by worker services.

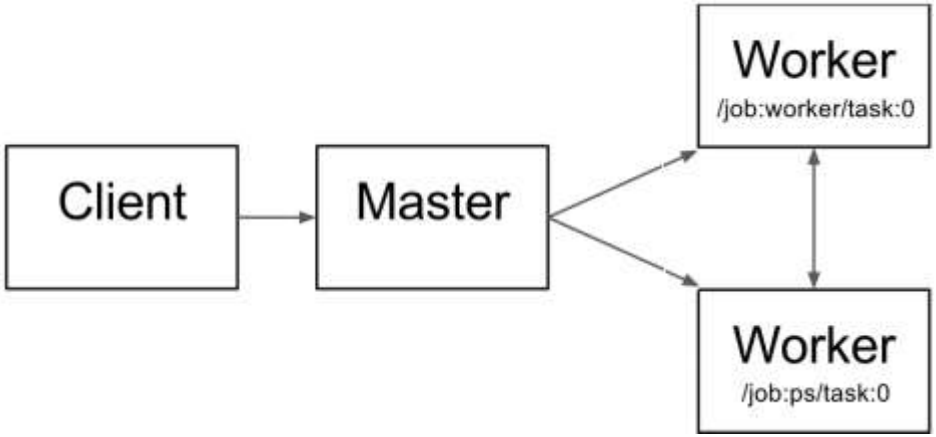
Worker Services (one for each task)

- Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, etc).
- Send and receive operation results to and from other worker services.

Kernel Implementations

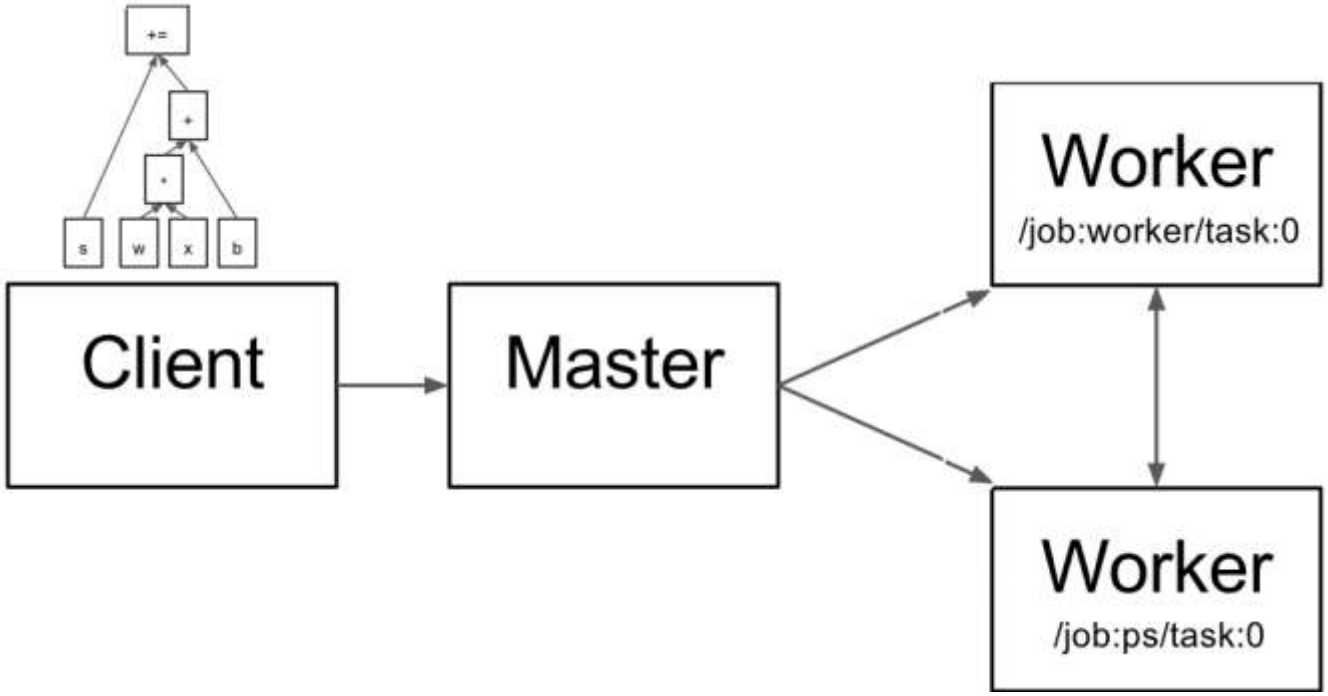
- Perform the computation for individual graph operations.

- Figure bellow illustrates the interaction of these components. "/job:worker/task:0" and "/job:ps/task:0" are both tasks with worker services.
- "PS" stands for "parameter server": a task responsible for storing and updating the model's parameters.
- Other tasks send updates to these parameters as they work on optimizing the parameters. This particular division of labor between tasks is not required, but it is common for distributed training.
- the Distributed Master and Worker Service only exist in distributed TensorFlow. The single-process version of TensorFlow includes a special Session implementation that does everything the distributed master does but only communicates with devices in the local process.



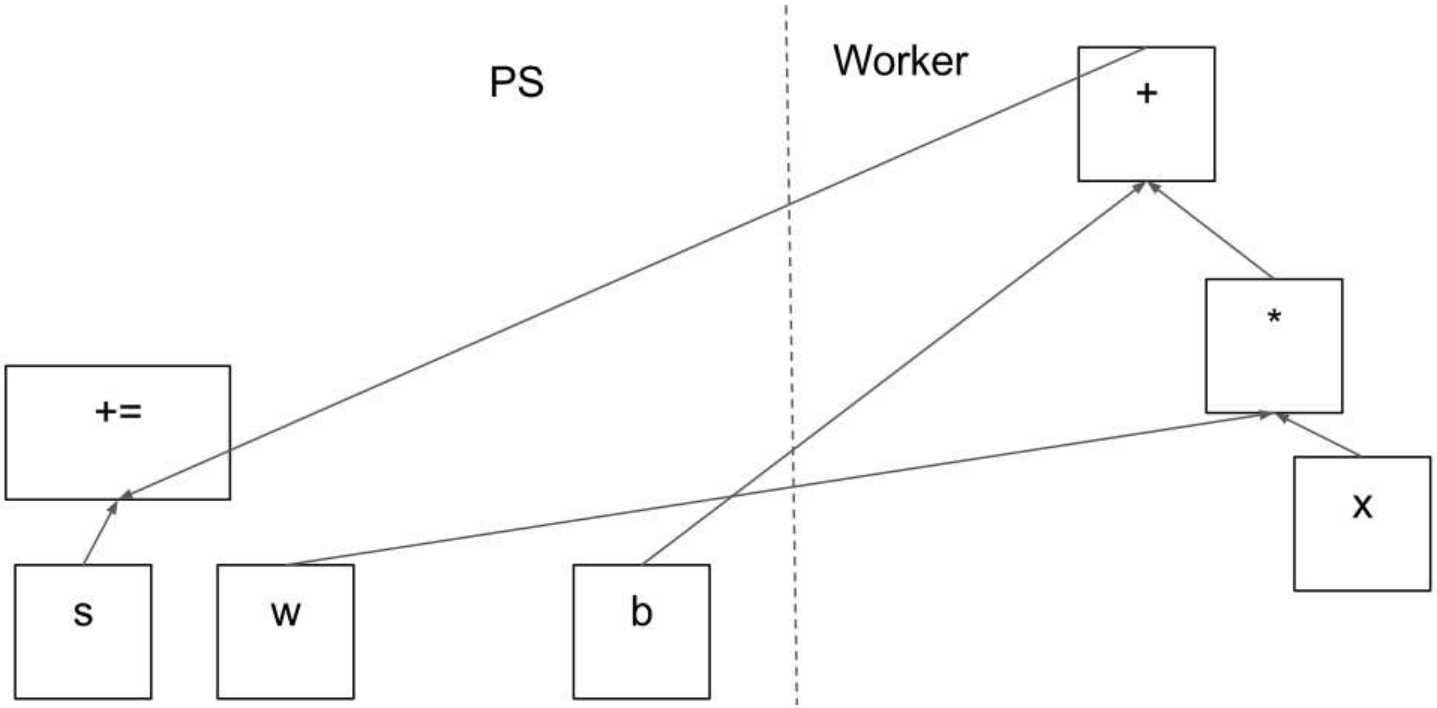
Client

- Users write the client TensorFlow program that builds the computation graph.
- The client creates a session, which sends the graph definition to the distributed master as a `tf.GraphDef` protocol buffer.
- The client has built a graph that applies weights (w) to a feature vector (x), adds a bias term (b) and saves the result in a variable (s).
- When the client evaluates a node or nodes in the graph, `RunStep()`, the evaluation triggers a call to the distributed master to initiate computation.



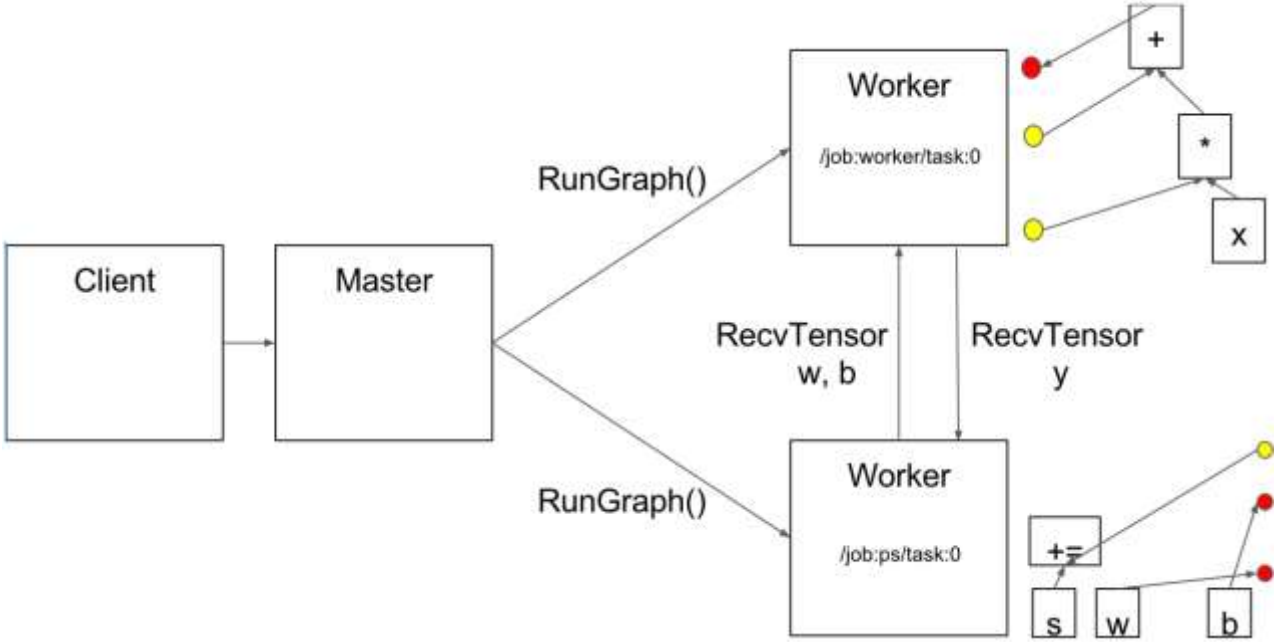
Distributed Master

- The distributed master:
  - prunes the graph to obtain the subgraph required to evaluate the nodes requested by the client,
  - partitions the graph to obtain graph pieces for each participating device, and
  - caches these pieces so that they may be re-used in subsequent steps.
- The distributed master groups the model parameters in order to place them together on the parameter server. Master establishes partitions



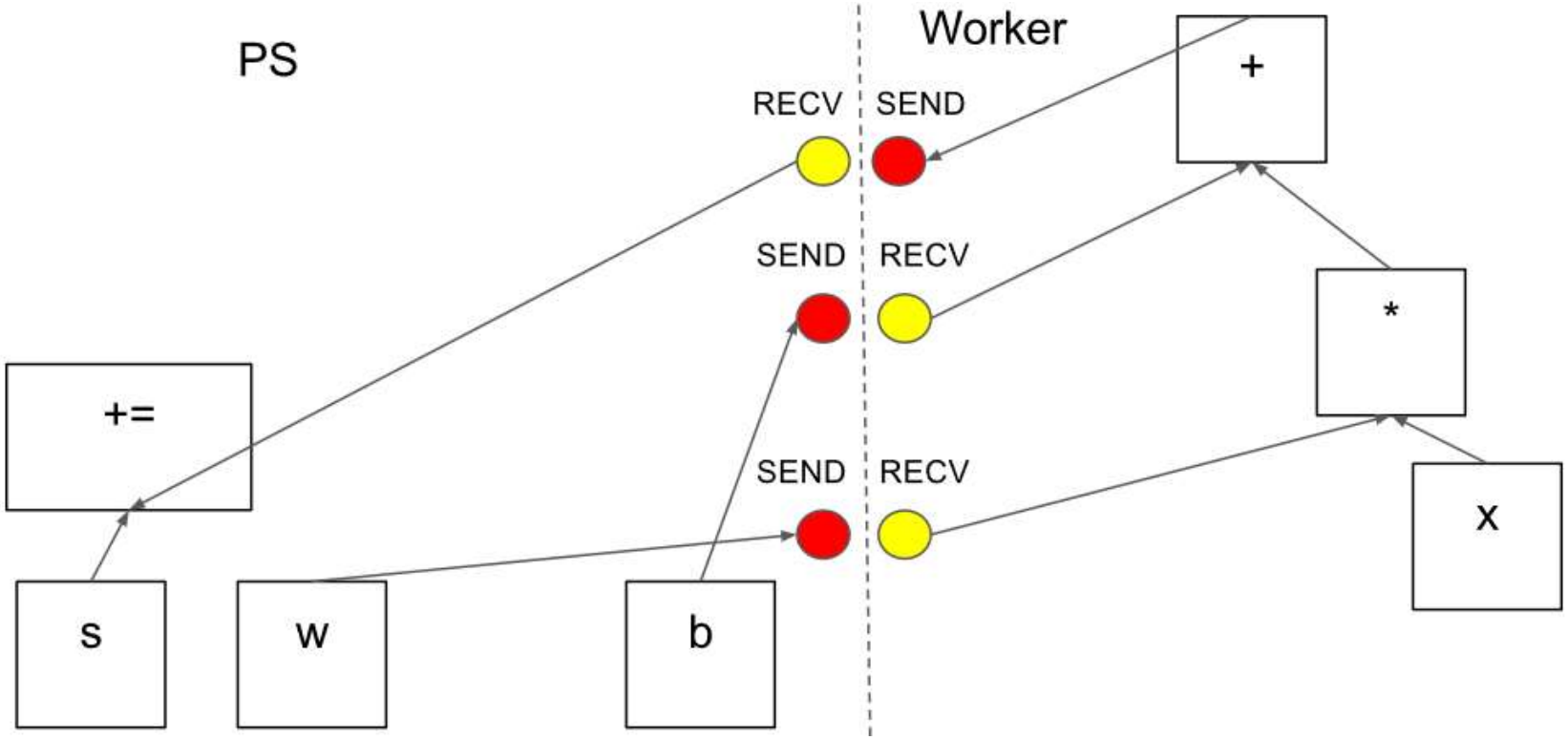
Distributed Master

- The distributed master:
  - prunes the graph to obtain the subgraph required to evaluate the nodes requested by the client,
  - partitions the graph to obtain graph pieces for each participating device, and
  - caches these pieces so that they may be re-used in subsequent steps.
- Since the master sees the overall computation for a step, it applies standard optimizations such as common subexpression elimination and constant folding. It then coordinates execution of the optimized subgraphs across a set of tasks.



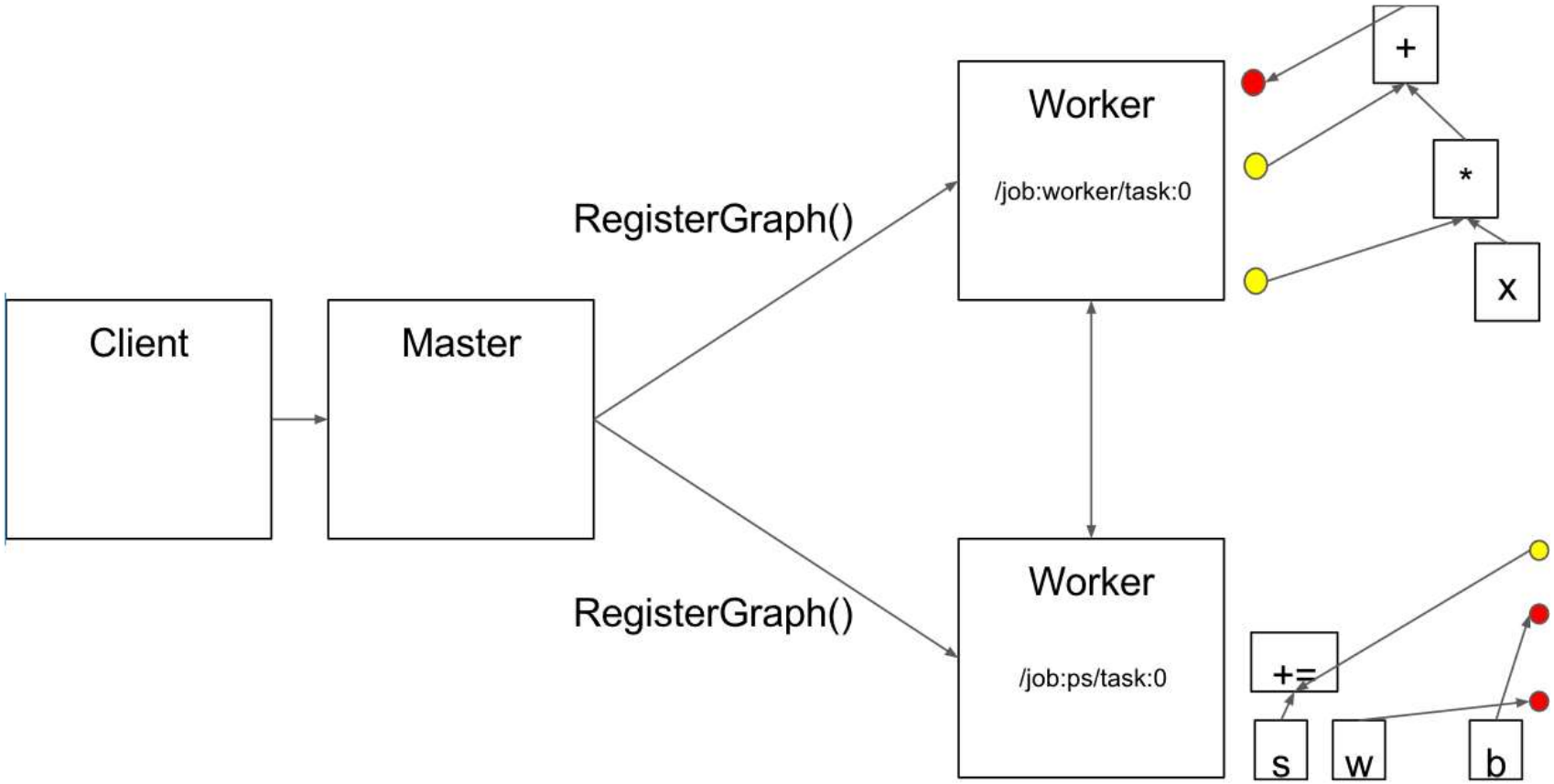
Graph Edges

- Where graph edges are cut by the partition, the distributed master inserts send and receive nodes to pass information between the distributed tasks



Execution on Different Nodes

- The distributed master then ships the graph pieces to the distributed tasks.



Worker Services

- The worker service in each task:
  - handles requests from the master,
  - schedules the execution of the kernels for the operations that comprise a local subgraph, and
  - mediates direct communication between tasks.
- Worker service is optimized for running large graphs with low overhead.
- Current implementation of TensorFlow can execute tens of thousands of subgraphs per second, which enables a large number of replicas to make rapid, fine-grained training steps. The worker service dispatches kernels to local devices and runs kernels in parallel when possible, for example by using multiple CPU cores or GPU streams.
- TensorFlow has build in specialized Send and Recv operations for each pair of source and destination device types:
  - Transfers between local CPU and GPU devices use the cudaMemcpyAsync() API to overlap computation and data transfer.
  - Transfers between two local GPUs use peer-to-peer DMA, to avoid an expensive copy via the host CPU.
- For transfers between tasks, TensorFlow uses multiple protocols, including:
  - gRPC over TCP.
  - RDMA over Converged Ethernet.

Distributed TensorFlow

- TensorFlow architecture allows efficient calculations over large neural networks both in training and in inference.
- TensorFlow efficiently uses both GPU machines and clusters of CPU and GPU machines.
- The rumor has it that big vendors like Amazon, Microsoft, even Google, internally use other, potentially even more powerful frameworks.
- TensorFlow is very popular, many algorithms implemented in TensorFlow are publicly available.
- Many commercial vendors and companies doing internal NN development use TensorFlow.
- For starters, if you have a CUDA card in your machine, please try to activate it.

- Our colleague: Sri Chaitanya Somanchi, submitted Piazza post @791 with detailed instructions on Installing TensorFlow GPU version on Windows10.
- If you have similar experience on a Mac or Linux machine, please share it with the class.

Summary Operations, `tf.summary`-s

- Summaries provide a way to export condensed information about a model, which is then accessible in tools such as TensorBoard.

Generation of Summaries

Classes for writing Summaries

- `tf.summary.FileWriter`
- `tf.summary.FileWriterCache`

Summary Ops

- `tf.summary.tensor_summary`
- `tf.summary.scalar`
- `tf.summary.histogram`
- `tf.summary.audio`
- `tf.summary.image`
- `tf.summary.merge`
- `tf.summary.merge_all`

•max_queue: Maximum number of summaries or events pending to be written to disk before one of the 'add' calls block.

Class `tf.summary.FileWriter`

- The FileWriter class provides a mechanism to create an event file in a given directory and add summaries and events to it. The class updates the file contents asynchronously. This allows a training program to call methods to add data to the file directly from the training loop, without slowing down training.
- On construction the summary writer creates a new event file in logdir. This event file will contain Event protocol buffers constructed when you call one of the following functions: add_summary(), add_session_log(), add_event(), or add_graph().
- If you pass a Graph to the constructor it is added to the event file. (Equivalent to calling add_graph() later).
- TensorBoard will pick the graph from the file and display it graphically so you can interactively explore the graph you built. You will usually pass the graph from the session in which you launched it:

Create a graph...

- # Launch the graph in a session.

sess = tf.Session()

- # Create a summary writer, add the 'graph' to the event file.

writer = tf.summary.FileWriter(<some-directory>, sess.graph)

logdir: A string. Directory where event file will be written.
graph: A Graph object, such as sess.graph.
max_queue: Integer. Size of the queue for pending events and summaries.
flush_secs: Number. How often, in seconds, to flush the pending events and summaries to disk.

tf.summary.scalar()

```
scalar(
    name,
    tensor,
    collections=None,
    family=None
)
```

- Defined in tensorflow/python/summary/summary.py.
- Outputs a Summary protocol buffer containing a single scalar value.
- The generated Summary has a Tensor.proto containing the input Tensor.

Args:

- `name`: A name for the generated node. Will also serve as the series name in TensorBoard.
- `tensor`: A real numeric Tensor containing a single value.
- `collections`: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to [GraphKeys.SUMMARIES].
- `family`: Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns:

- A scalar Tensor of type string. Which contains a Summary protobuf.

Raises:

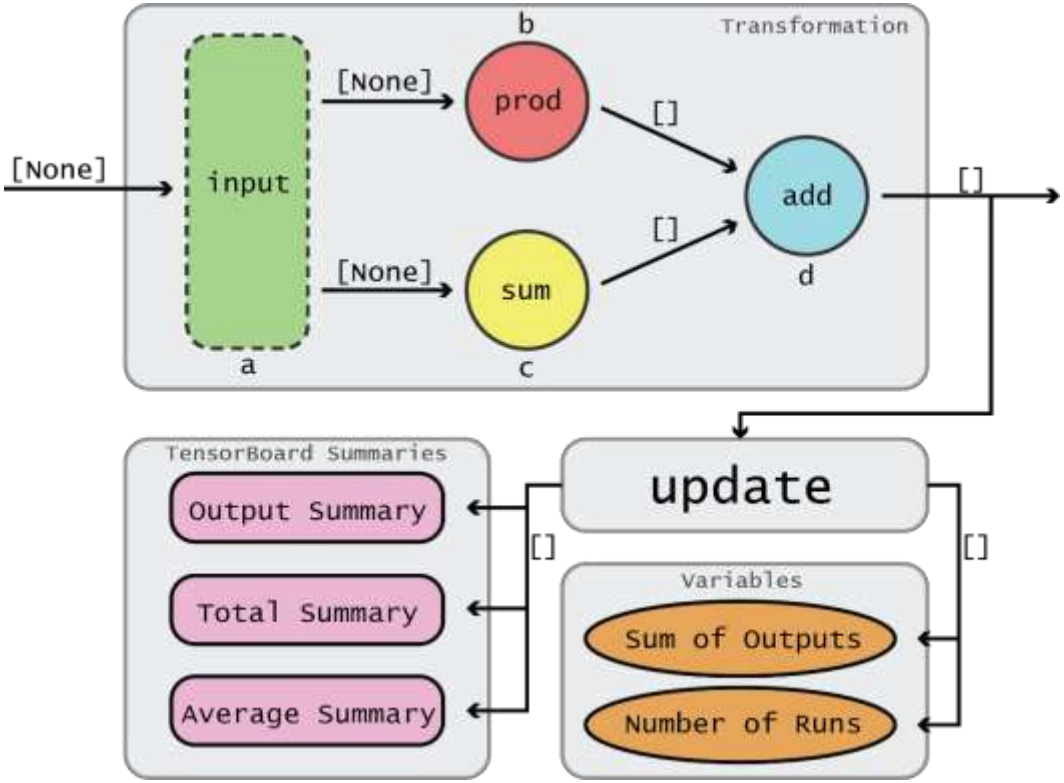
- ValueError: If tensor has the wrong shape or type.

Illustration of `summary` functions

- He following code will illustrate use of Tensors, Graphs, Operations, Variables, placeholders, Sessions, and name scopes. We'll also include some TensorFlow summaries so we can keep track of the graph as it runs.
- We will crate a TensorFlow graph and run it several times with different inputs and track how various summaries are changing with each run.
- Our inputs will be placeholders
- Instead of taking two discrete scalar inputs, our model will take in a single vector of any length
- We're going to accumulate the total value of all outputs over time as we use the graph
- The graph will be segmented nicely with name scopes
- After each run, we are going to save the output of the graph, the accumulated total of all outputs, and the average value of all outputs to disk for use in TensorBoard.

Graph and Process

- Here's a rough outline of what we'd like our graph to look like:



Program Objectives

- Notice how each edge now has either a [None] or [] icon next to it. This represents the TensorFlow shape of the tensor flowing through that edge, with None representing a vector of any length, and []representing a scalar.
- The output of node `d` now flows into an “update” section, which contains Operations necessary to update Variables as well as pass data through to the TensorBoard summaries.
- We have a separate name scope to contain our two Variables- one to store the accumulated sum of our outputs, the other to keep track of how many times we’ve run the graph.
- These two Variables operate outside of the flow of our main transformation, and it makes sense to put them in a separate space.
- There is a name scope dedicated to our TensorBoard summaries which will hold ourtf.scalar_summary Operations. We place them after the “update” section to ensure that the summaries are added *after* we update our Variables, otherwise things could run out of order.

graph_and_summaries.py

```
import tensorflow as tf
import numpy as np
# Explicitly create a Graph object
graph = tf.Graph()
with graph.as_default():

    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, name="global_step")

        # Variable that keeps track of the sum of all output values over time:
        total_output = tf.Variable(0.0, dtype=tf.float32, name="total_output")

    # Primary transformation Operations
    with tf.name_scope("transformation"):

        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32, shape=[None], name="input_placeholder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")
```

graph_and_summaries.py

```
# Separate output layer
    with tf.name_scope("output"):
        output = tf.add(b, c, name="output")

with tf.name_scope("update"):
    # Increments the total_output Variable by the latest output
    update_total = total_output.assign_add(output)

  # Increments the above `global_step` Variable, run whenever the graph is run
    increment_step = global_step.assign_add(1)

# Summary Operations
with tf.name_scope("summaries"):
    avg = tf.div(update_total, tf.cast(increment_step, tf.float32), name="average")

    # Creates summaries for output node
    tf.summary.scalar('Output', output)
    tf.summary.scalar('Sum_of_outputs_over_time', update_total)
    tf.summary.scalar('Average_of_outputs_over_time', avg)

# Global Variables and Operations
with tf.name_scope("global_ops"):
    # Initialization Op
    init = tf.global_variables_initializer()
    # Merge all summaries into one Operation
    merged_summaries = tf.summary.merge_all()
```

graph_and_summaries.py

```
# Start a Session, using the explicitly created Graph
sess = tf.Session(graph=graph)

# Open a SummaryWriter to save summaries
writer = tf.summary.FileWriter('summaries', graph)

# Initialize Variables
sess.run(init)

def run_graph(input_tensor):
    """
    Helper function; runs the graph with given input tensor and saves summaries
    """
    feed_dict = {a: input_tensor}
    out, step, summary = sess.run([output, increment_step, merged_summaries],
feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)

# Run the graph with various inputs
run_graph([2,8])
run_graph([3,1,3,3])
run_graph([8])
run_graph([1,2,3])
run_graph([11,4])
run_graph([4,1])
run_graph([7,3,1])
run_graph([6,3])
run_graph([0,2])
run_graph([4,5,6])
```

graph_and_summaries.py, close() everything

```
# Write the summaries to disk
writer.flush()

# Close the tf.summary.FileWriter
writer.close()

# Close the session
sess.close()
```

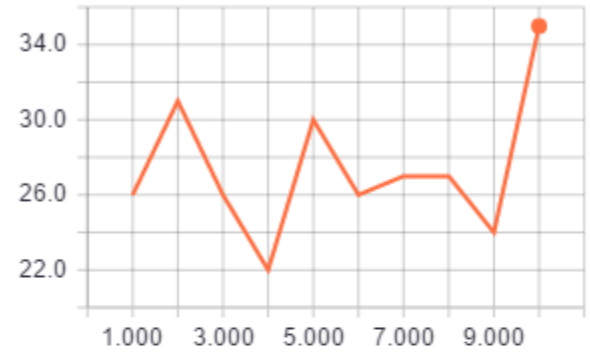
- To start TensorBoard after running this code, run the following command:

```
$ tensorboard --logdir 'summaries'
```

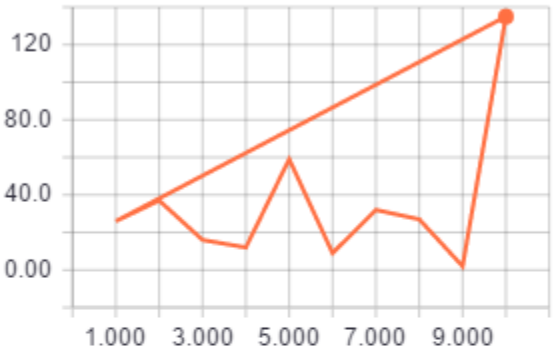
Summaries

summaries

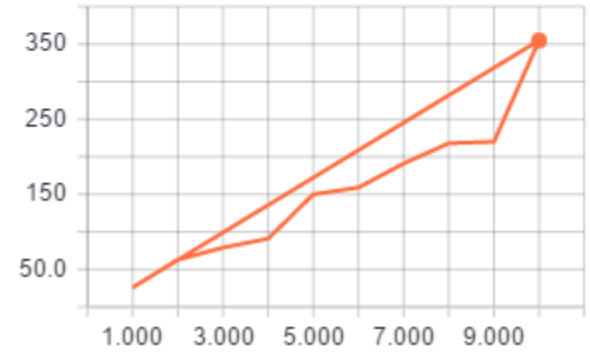
summaries/average_summary



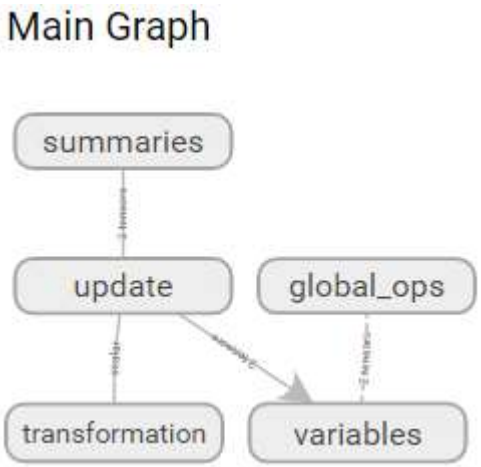
summaries/output

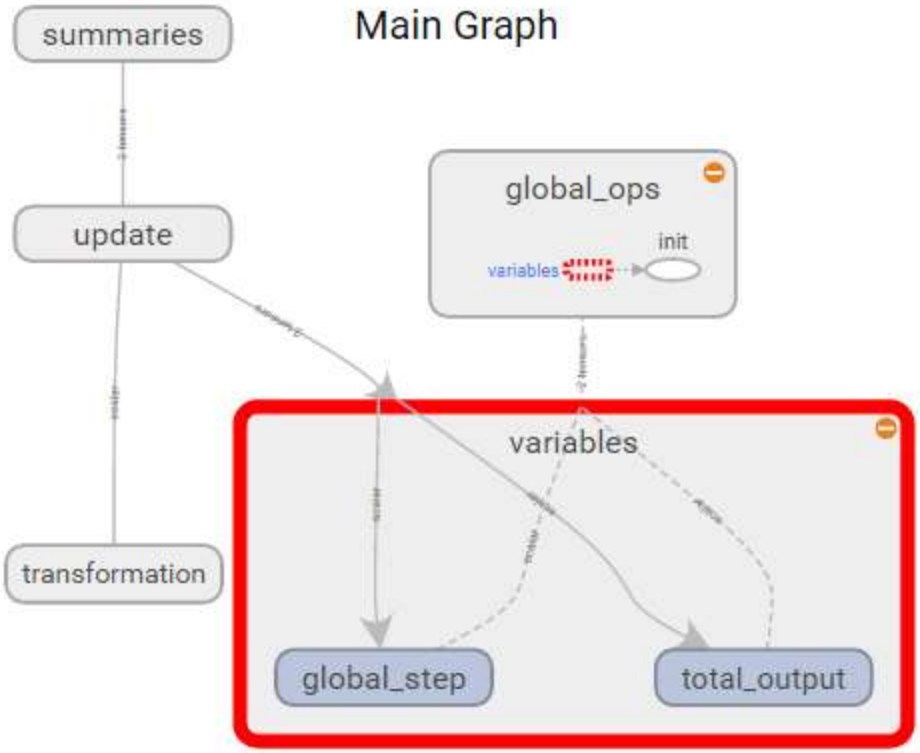


summaries/update_total

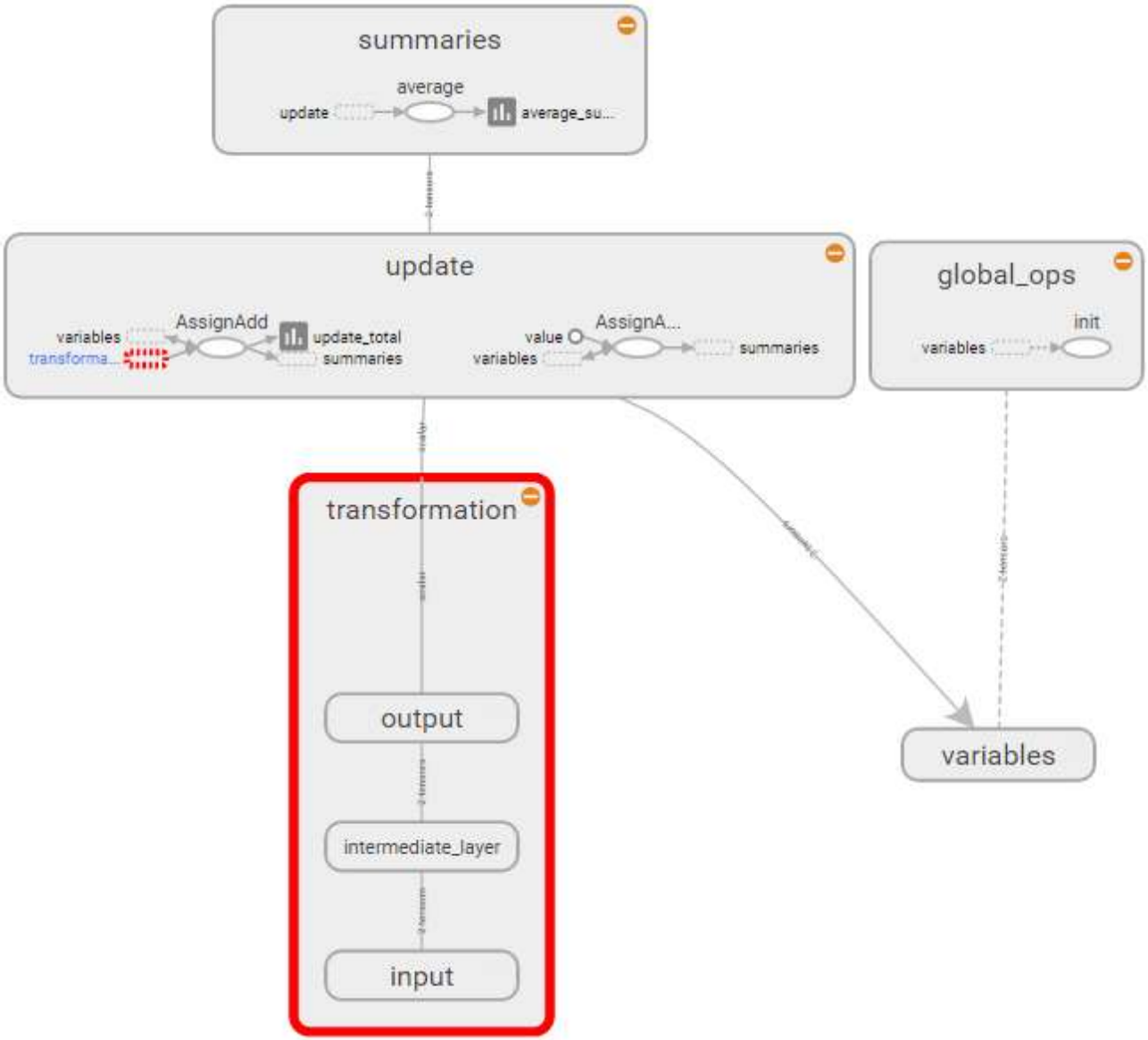


Graph

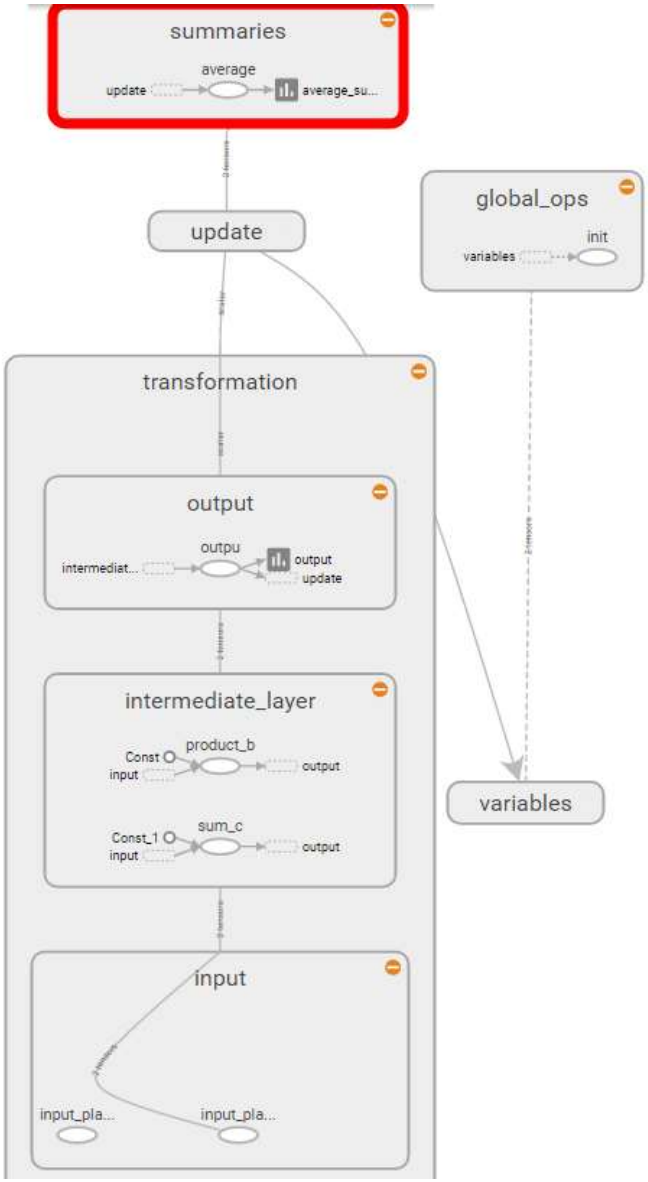

Main Graph
summaries
update
global_ops
transformation
variables


summaries
Main Graph
update
global_ops
init
variables
variables
transformation
global_step
total_output

Partially Expanded Graph

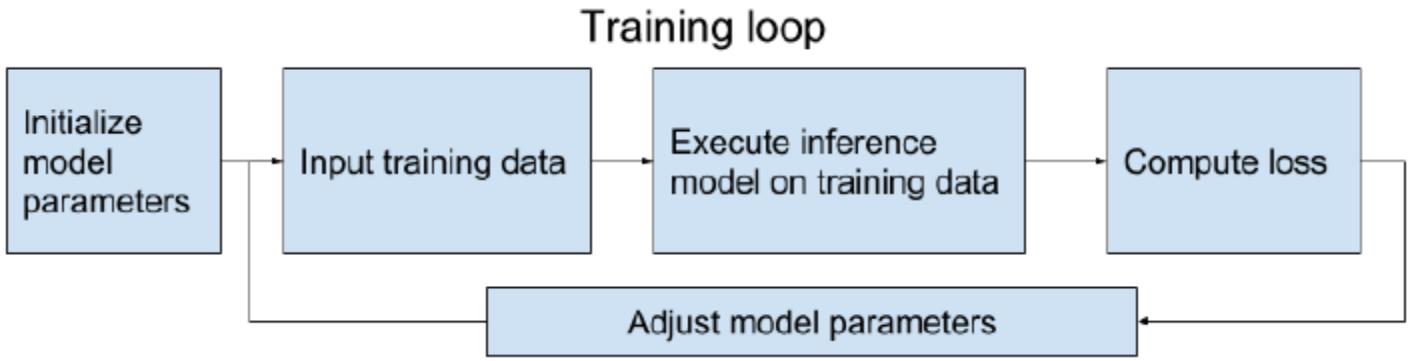


Partially Expanded Graph



Mechanism of Machine Learning Process

- We **train an inference model** with an input dataset, along with the real or expected output for each example. The model should 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. The steps are set by code and determined by the model we are using to solve a given problem.
- The operations composing our model are fixed. Inside the operations we have arbitrary **parameters** of the model, and those are the ones that change through training in order for the model to learn and adjust its output.
- Although the inference models may vary significantly in the number of operations they use, and in the way they combine and the number of parameters they have; we always apply the same general process, a training loop.



Steps in the Training Loop

- Initializes the model parameters for the first time. Usually we use random values for this, but in simple models we may just set zeros.
- Reads the training data along with the expected output data for each data example. Usual operations here may also imply randomizing the order of the data for always feeding it differently to the model.
- Executes the inference model on the training data, so it calculates for each training input example the output with the current model parameters.
- Computes the loss. The loss is a single value that will summarize and indicate to our model how far are the values that computed in the last step with the expected output from the training set. There are different loss functions that we can use.
- Adjusts the model parameters. This is where the learning actually takes place. Given the loss function, learning is just a matter of improving the values of the parameters in order to minimize the loss through a number of training steps. Most commonly, we can use a gradient descent algorithm for this.

- The loop repeats this process through a number of cycles, according to the learning rate that we need to apply, and depending on the model and data we input to it.

Evaluation Phase

- After training, we apply an *evaluation phase*; where we execute the inference against a different set data to which we also have the expected output, and evaluate the loss for it.
- Given how this dataset contains examples unknown for the model, the evaluation tells you 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.

- In the following we will define some generic scaffolding for the model code.
- First we will initializes the model parameters.
- Then we defines a method for each of the training loop operations: read input training data (inputs method), compute inference model (inference method), calculate loss over expected output (loss method), adjust model parameters (train method), evaluate the resulting model (evaluate method), and then the boilerplate code to start a session and run the training loop.

Supervised Learning Process, Scaffolding

```
import tensorflow as tf
# initialize variables/model parameters
# define the training loop operations
def inference(X)=YY: # compute inference model over data X and return result

def loss(YY, 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 total loss

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

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

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

- Training models implies updating their parameters, or variables, 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 of that work.
- TF API provides `tf.train.Saver` 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.
- 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()
```

- We call the `tf.train.Saver.save` method for each 1000 training steps. Each call will create a checkpoint file with the name template `my-model-{step}` like `my-model-1000, my-model-2000,` etc. The file stores the current values of each variable.

Recovery the Training from a Point

- If we wish to recover the training from a certain point, we use the method `tf.train.get_checkpoint_state`, which verify whether we already have a checkpoint saved, and the `tf.train.Saver.restore` method to recover the variable values.

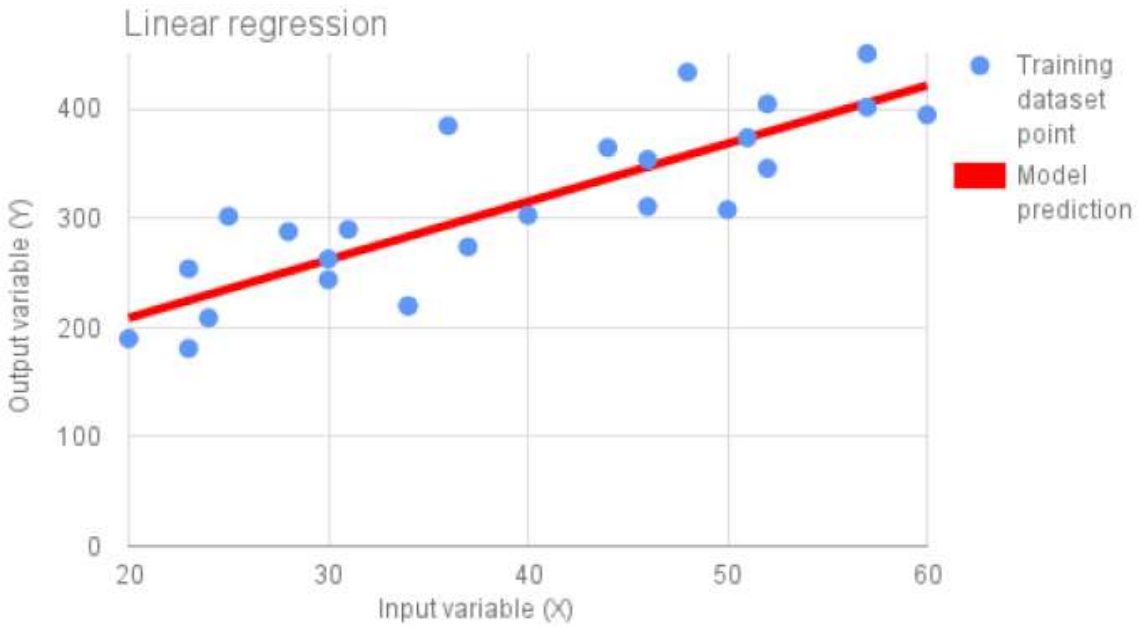
```
with tf.Session() as sess:
      # model setup....
      initial_step = 0
      # verify whether we have a checkpoint saved already
      ckpt = tf.train.get_checkpoint_state(os.path.dirname(__file__))
      if ckpt and ckpt.model_checkpoint_path:
            # Restores from checkpoint
            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):
…
```

- In the code above we check first if we have a checkpoint file, and then restore the variable values before staring the training loop. We also recover the global step number from the checkpoint file name.

Linear Regression

- Linear regression is the simplest form of modeling for a supervised learning problem. 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.



- Here is a chart of the lineal regression model in 2D. Blue dots are the training data points and the red line is the what the model will infer.

Linear Regression Model

- The general formula of a linear function used in linear regression model is:

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

- In matrix (tensor) form this reads:

$$Y = XW^T + b, \qquad \text{where } X = x_1, \ldots, x_k, \qquad \text{and } W = w_1, \ldots, w_k$$

- Y is the value we are trying to predict.
- $x_1, \ldots, x_k$ independent or predictor variables are the values that we provide when using our model for predicting new values.
- $w_1, \ldots, w_k$ are the parameters the model will learn from the training data, or the "weights" given to each variable.
- *b* is also parameter that we have to learn - the constant of the linear function that is also known as the bias of the model.
- In code, this model reads:

```
# 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
```

L2 Loss

- We have to define how to compute the loss. For this simple model we will use a squared error, which sums the squared difference of all the predicted values for each training example with their corresponding expected values.
- Algebraically, squared loss is the squared Euclidean distance between the predicted output vector and the expected one. Graphically, in a 2d dataset, the loss is the length of the vertical line that you can trace from the expected data point to the predicted regression line.
- Square loss 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 us a computing step.

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

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

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

Training and Evaluating the Model

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

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

- As we run the training, we will see how the loss (usually) gets smaller on each training step.
- Once we trained the model, it's time to evaluate it. We take a set of data (experiments) not in the training set and calculate the loss function (inference):

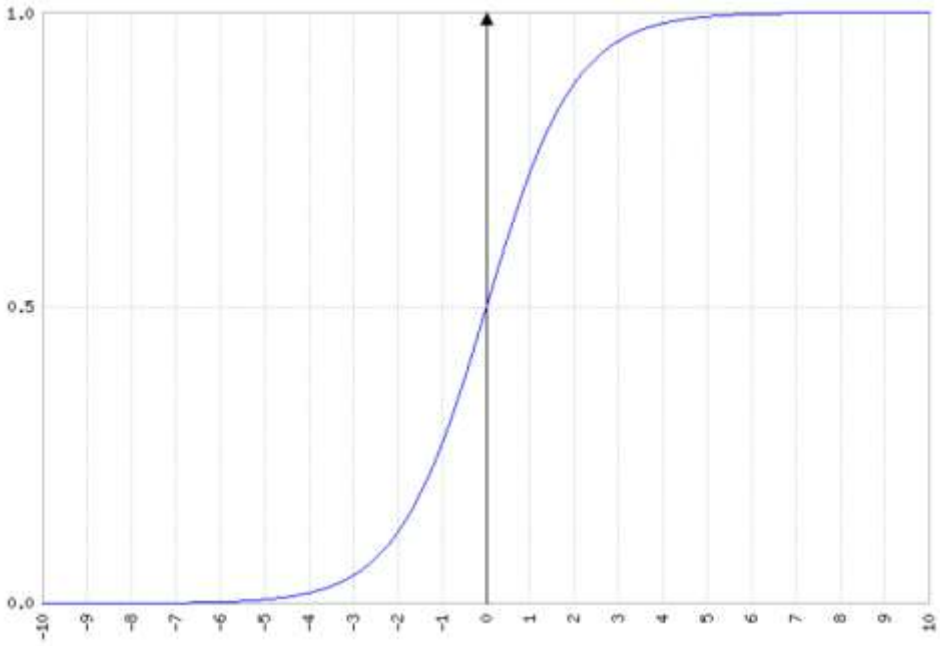
```
def evaluate(sess, X, Y):
     print sess.run(inference([[80., 25.]])) # ~ 303
     print sess.run(inference([[65., 25.]])) # ~ 256
```

- As a quick evaluation, we compare results with the original Y values and verify whether the model learned anything.
- If the output values are in between the boundaries of the original trained values, we declare the victory

Logical Regression

- The linear regression model predicts a *continuous* value, or any real number. We need models that can answer a yes-no type of question, like “Is this email spam?”. We need those models for classification problems.
- In machine learning we most frequently us a function called the **logistic function**. It is also known as the *sigmoid* function, because its shape is an S (and sigma is the Greek letter Σ equivalent to s).

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



Logistic Regression in Code

- The logistic function is a probability distribution function that, given a specific input value, computes the probability of the output being a *success*, and thus the probability for the answer to the question to be “yes.”
- This function takes a single input value. 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. We can use the linear regression model expression for doing this. When coding, we can reuse all of the elements of the linear model. We just slightly change the prediction to use the sigmoid.

```
# 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))
```

Loss Function for Classifications

- The logistic function computes the probability of the answer being “yes.” In the training set, a “yes” answer should represent 100% of probability, or simply the output value to be 1.
- Then the loss should be how much probability our model assigned less than 1 for that particular example, squared. Consequently, a “no” answer will represent 0 probability, hence the loss is any probability the model assigned for that example, and again squared.
- 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% sure that 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. Tensorflow method for calculating cross entropy directly for a sigmoid output in a single, optimized step:

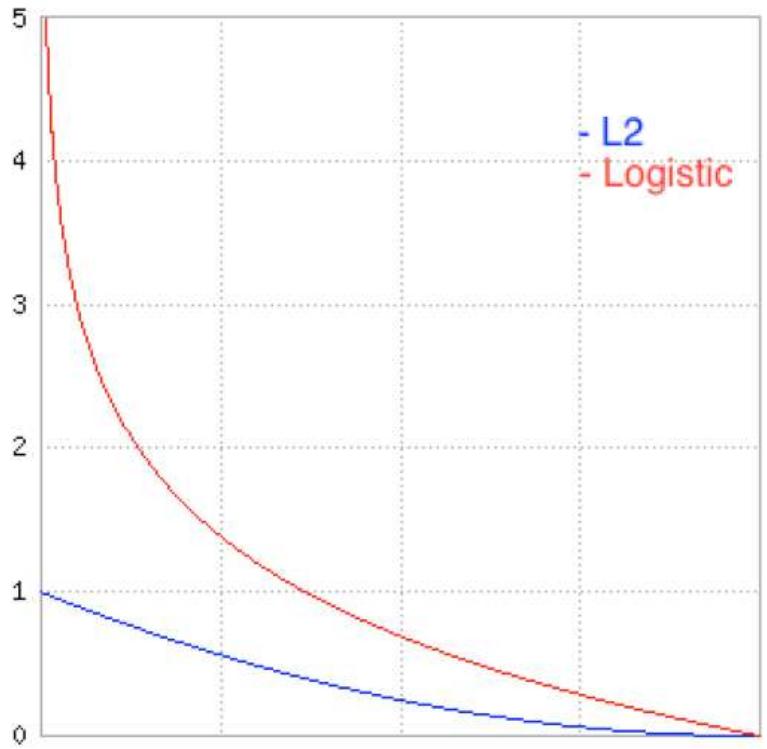
```
def loss(X, Y):
    return
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(combine_inputs(X), Y))
```

Cross Entropy Loss Function

- Cross entropy loss function is defined as:

$$loss = -\Sigma_i \left(y_i \,.\log(y\_predicted_i) + 1 - y_i \,.\log(1 - y\_predicted_i)\right)$$

- We can visualize this function according to the predicted output for a “yes” as:



- Cross entropy outputs a much greater value (“penalty”), because the output is farther from what is expected.
- With cross entropy, as the predicted probability comes closer to 0 for the “yes” example, the penalty grows towards infinity.
- This makes it very expensive for the model to make that mis-prediction.
- Cross entropy is better suited as a loss function for classification models.

Cross Entropy

- In the Information theory, Shannon entropy 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.

$$H = -\Sigma_i p_i \,.\log_2 (p_i)$$

- You can link this entropy with the thermodynamic concept of entropy.
- For instance, let's calculate the entropy for the word "HELLO", assuming that you need to express by bits only four letters: H, E, L and O.

$$p("H") = p("E") = p("O") = 1/5 = 0.2$$
$$p("L") = 2/5 = 0.4$$
$$H = -3 * 0.2 * log_2(0.2) - 0.4 * log_2(0.4) = 1.92193$$

- So, we need 2 bits per symbol to express word Hello (`00` for H, `01` for E, `10` for L and `11` for O)

Cross Entropy

- If you encode the symbols assuming any other probability for the q_i than the real p_i, *you will* need more bits for encoding of each symbol. That's where cross-entropy comes to play. It allows you to calculate the average minimum number of bits needed to encode the same string in another suboptimal encoding.

$$H = -\Sigma_i p_i \cdot \log_2 (q_i)$$

- For example, ASCII assigns the uniform probability $q_i = 1/256$ for all its symbols.
- Let's calculate the cross-entropy for the word "HELLO" in ASCII encoding.
- p_i = 0.2 for H,E and O and p_i = 0.4 for L.

$$q("H") = q("E") = q("L") = q("O") = 1/256$$
$$H = -3 * 0.2 * log_2(1/256) - 0.4 * log_2(1/256) = 8$$

- So, you need 8 bits per symbol to encode "HELLO" in ASCII, as you would have expected.
- One can show that cross entropy is at its minimum when *p* = *q*. Thus, we can use cross entropy to compare how one distribution "fits" another. The closer the cross entropy is to the entropy, the better *q* is an approximation of *p*.
- Effectively, cross-entropy reduces as the model better resembles the expected output, like you need in a loss function.
- We can freely exchange *log*2 with *log* for minimizing the entropy as you switch one to another by multiplying by the change of the base constant.

Softmax Classification

- With logistic regression we were able to model the answer to the yes-no question.
- We want to be able to answer a multiple choice type of question like: “Was a person born in Boston, London, or Sydney?”
- For that case we use the **softmax** function, which is a generalization of the logistic regression for K possible different value*s*.
- Softmax function calculates K components of a K-dimensional vector representing corresponding probabilities for each output class. Using softmax function, j-th component of that vector is represented as:

$$f(x)_j = \frac{e^{-x_j}}{\sum_{i=0}^{K-1} e^{-x_i}}, \qquad \sum_{j=0}^{K-1} f(x)_j = 1$$

- As it is a probability, the sum of the K vector components always equal to 1. That is because the formula is composed such that every possible input data example must belong to one output class, covering the 100% of possible examples.
- If the sum would be less than 1, it would imply that there could be some hidden class alternative. If it would be more than 1, it would mean that each example could belong to more than one class.
- You can show that if the number of classes is 2, the resulting output probability is the same as a logistic regression model.

Coding Multiple Categories Model

- Now, to code a model with several categories, you will have one slight change from the previous models in the variable initialization.
- Given that our model computes K outputs instead of just one, we need to have K different weight groups, one for each possible output.
- So, you 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.
- For our homework we will use the classical Iris flower dataset to try softmax. You can download the dataset from ***https://archive.ics.uci.edu/ml/datasets/Iris .*** It 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.
- The variable initialization code should look like:

```
# this time weights form a matrix, not a vector, with one "feature 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))
```

Loss Function in Multiple Categories, `softmax`

- In loss computation, the same considerations for logistic regression apply for fitting a candidate loss function, as the output here is also a probability. We will use the cross-entropy again, adapted for multiple classes in the computed probability.
- For a single training example *i*, cross entropy now reads:

$$loss_i = -\Sigma_j y_j \cdot \log(y\_predicted_j)$$

- We are summing the loss for each output class on the training example. Note that y_j would equal 1 for the expected class of the training example and 0 for the rest, so only one loss value is actually summed, the one measuring how far the model predicted the probability for the true class.
- To calculate the total loss among the training set, we sum the loss for each training example:

$$loss = -\Sigma_i \Sigma_j y_{ji} \cdot \log(y\_predicted_{ji})$$

- There are two versions of `softmax` cross entropy function. One is specially optimized for training sets with a single class value per example.
- For example, our training data may have a class value that could be either "dog","person" or "tree". That function is `tf.nn.sparse_softmax_cross_entropy_with_logits`.

```
def loss(X, Y):
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
        combine_inputs(X), Y))
```

TensorFlow `softmax` Implementations

- The other version of sofmax function lets you work with training sets containing the probabilities of each example to belong to every class.
- For instance, you could use training data like “60% of the asked people consider that this picture is about dogs, 25% about trees, and the rest about a person”.
- That function is `tf.nn.softmax_cross_entropy_with_logits`.
- You may need such a function with some real world usages, but we won’t need it for our first homework examples. The sparse version is preferred when possible because it is faster to compute.
- Note that the final output of the model will always be one single class value, and this version is just to support a more flexible training data.

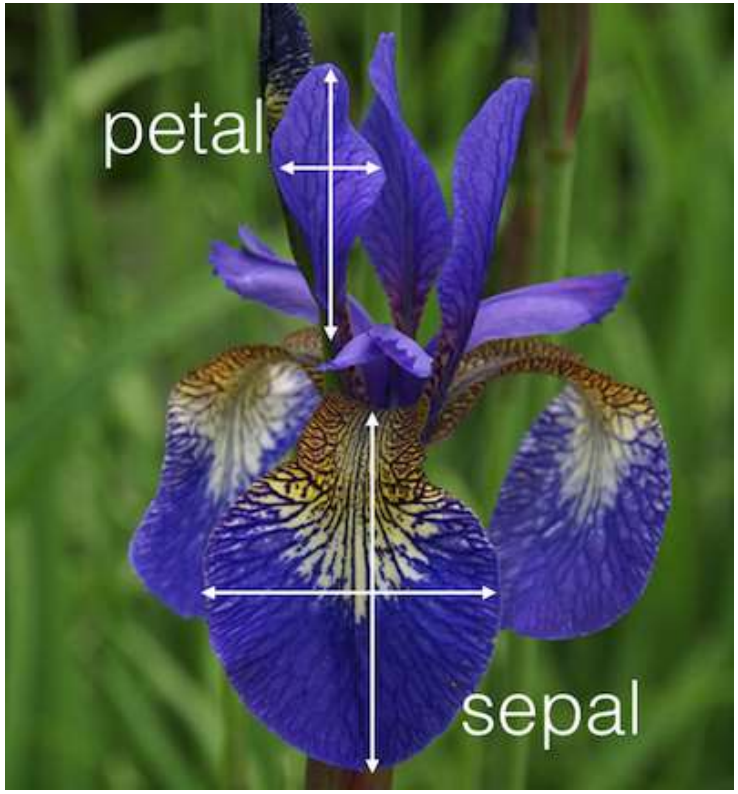
Analyzing Iris Dataset (from Wikipedia)

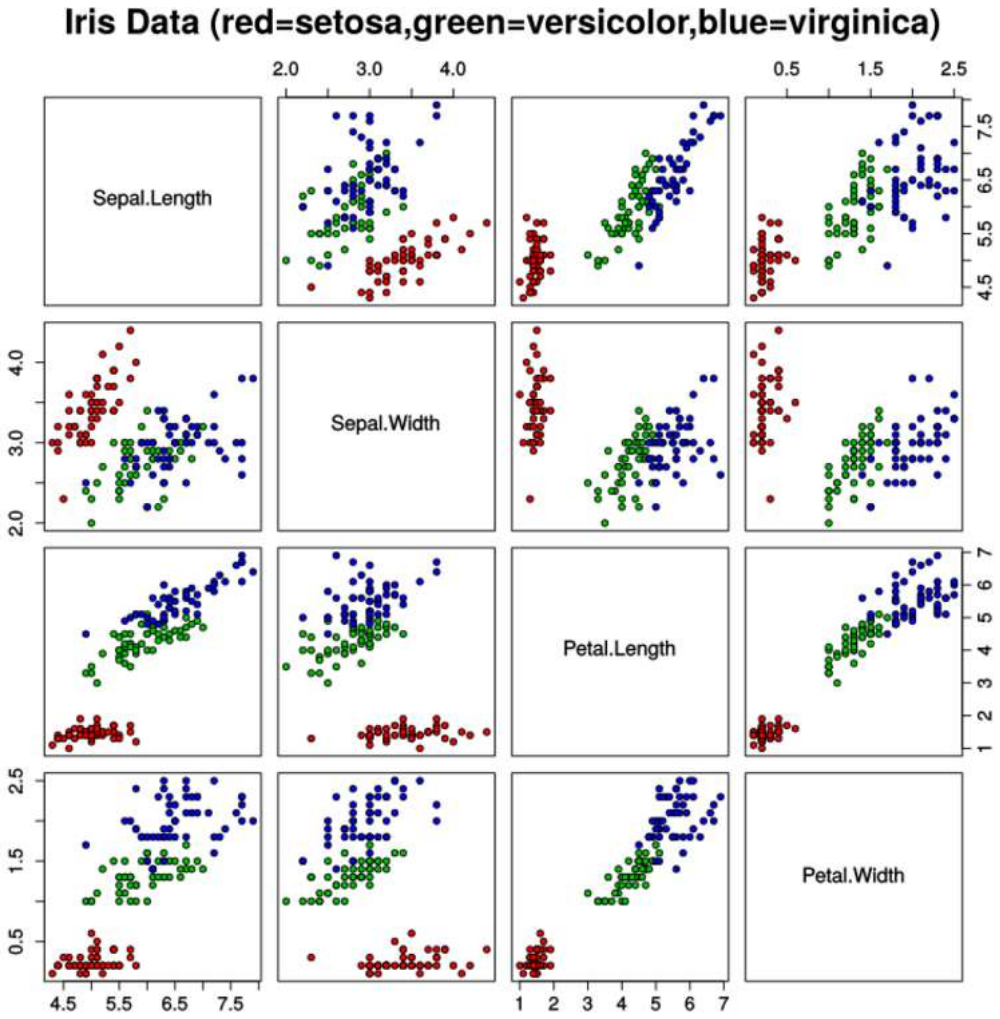
- The ***Iris* flower data set** or **Fisher's *Iris* data set** is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper *The use of multiple measurements in taxonomic problems* as an example of linear discriminant analysis.
- The dataset is sometimes called **Anderson's *Iris* data set** because Edgar Anderson collected the data to quantify the morphologic variation of *Iris* flowers of three related species.
- Two of the three species were collected in the Gaspé Peninsula "all from the same pasture, and picked on the same day and measured at the same time by the same person with the same apparatus".
- The data set consists of 50 samples from each of three species of *Iris* (*Iris setosa*, *Iris virginica* and *Iris versicolor*).
- Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres.
- Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

Scatterplot of Correlation Matrix

- By Nicoguaro

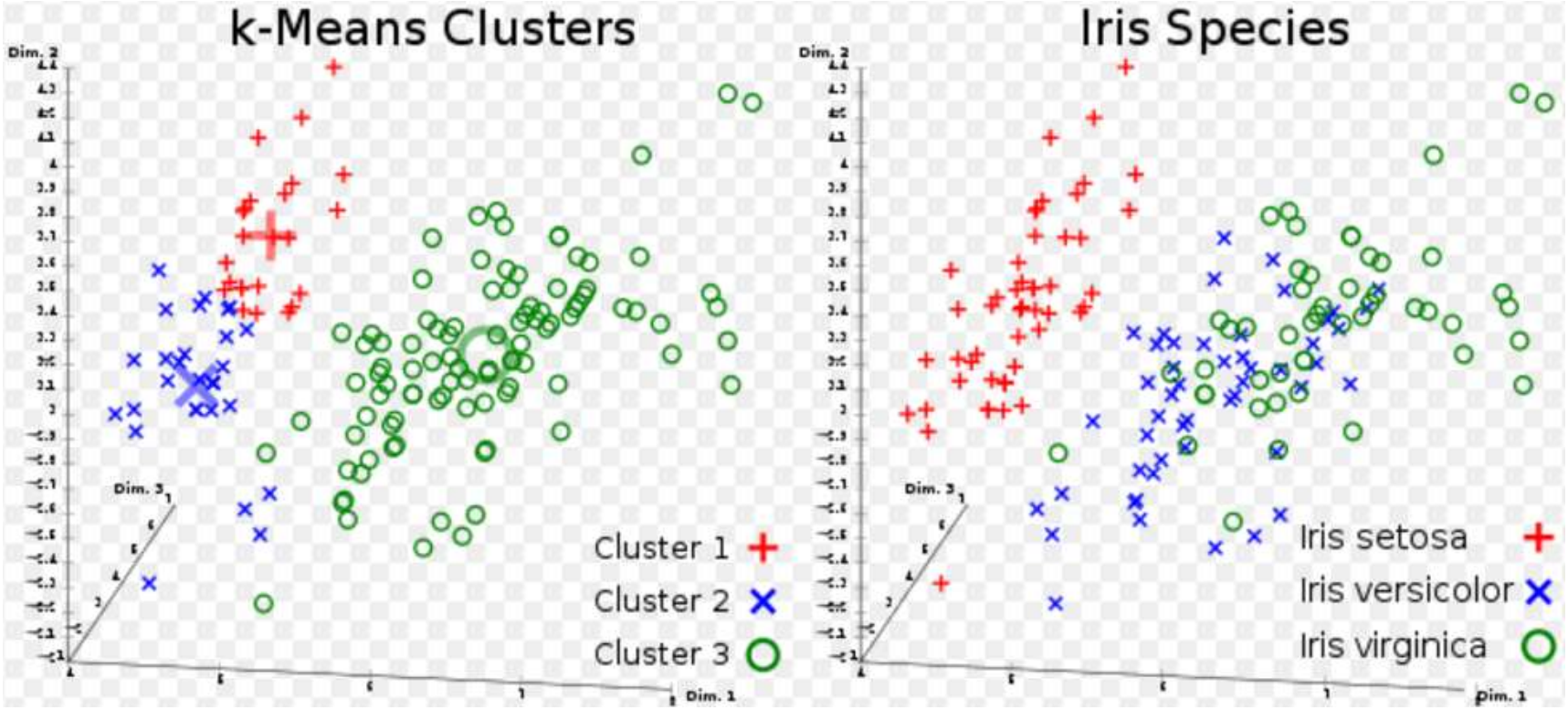
https://commons.wikimedia.org/w/index.php?curid=46257808





Classification Analysis

- We are looking for ways to identify type of Iris



Iris Data Set

Sepal Length	Sepal Width	Petal Length	Petal Width	Class
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa

Iris Classification using `sofmax`

```
# Softmax example in TF using the classical Iris dataset
# Download iris.data from https://archive.ics.uci.edu/ml/datasets/Iris
import tensorflow as tf
import os

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

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

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

def loss(X, Y):
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(combine_inputs(X), Y))

def read_csv(batch_size, file_name, record_defaults):
    filename_queue = tf.train.string_input_producer([os.path.dirname(__file__) + "/" + 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_defaults)
```

Iris Classification using sofmax

```
    # 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)

def inputs():
    sepal_length, sepal_width, petal_length, petal_width, label =\
        read_csv(100, "iris.data", [[0.0], [0.0], [0.0], [0.0], [""]])

    # 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 matrix;
    # We then transpose to have a matrix with one example per row and one feature per column.
    features = tf.transpose(tf.pack([sepal_length, sepal_width, petal_length, petal_width]))
    return features, label_number

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

Iris Classification using sofmax

```
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)))


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

    tf.global_variables_initializer().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=coord)

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

    evaluate(sess, X, Y)

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

Description of Code

- As the input method, we will reuse the read_csv function, 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, label =\
       read_csv(100, "iris.data", [[0.0], [0.0], [0.0], [0.0], [""]])

    # 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 matrix;
    # We then transpose to have a matrix with one example per row and
    # one feature per column.
       features = tf.transpose(tf.pack([sepal_length, sepal_width, petal_
  length, petal_width]))
          return features, label_number
```

Description of code

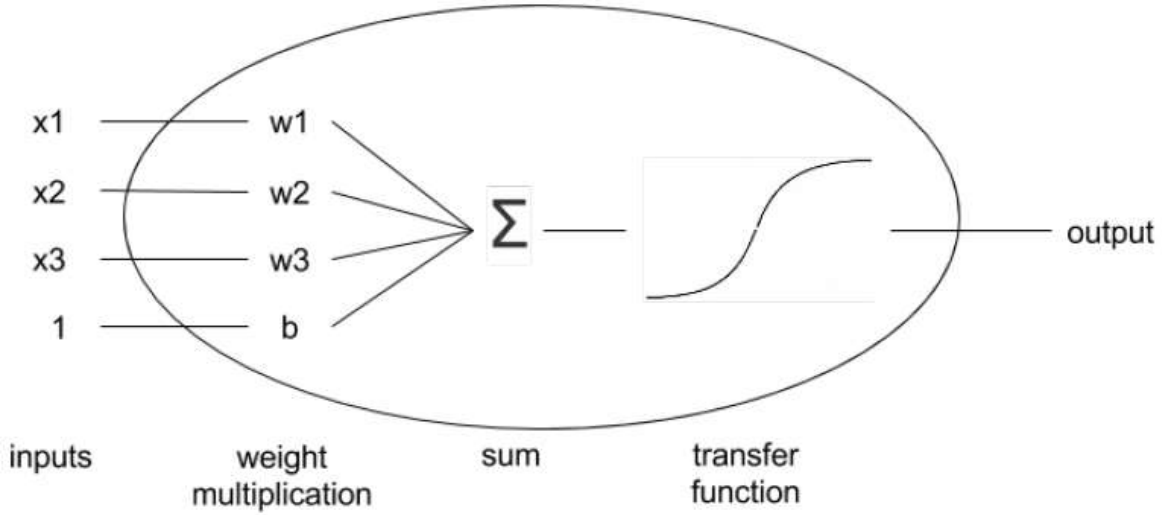
- We don't need to convert each class to its own variable to use with `sparse_softmax_cross_entropy_with_logits`, but we need the value to be a number in the range of 0..2, since we have 3 possible classes.
- In the dataset file the class is a string value from the possible "Iris-setosa", "Iris-versicolor", or "Iris-virginica". To convert it we create a tensor with `tf.pack`, comparing the file input with each possible value using `tf.equal`.
- Then we use `tf.argmax` to find the position on that tensor which is valued true, effectively converting the classes to a 0..2 integer.
- For evaluation of accuracy, we need the following 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)))
```

- The inference will compute the probabilities for each output class for our test examples. We use the `tf.argmax` function to choose the one with the highest probability as the predicted
- output value. Finally, we compare with the expected class with `tf.equal` and apply `tf.reduce_mean` just like with the sigmoid example.
- Running the code should print an accuracy of about 96%.

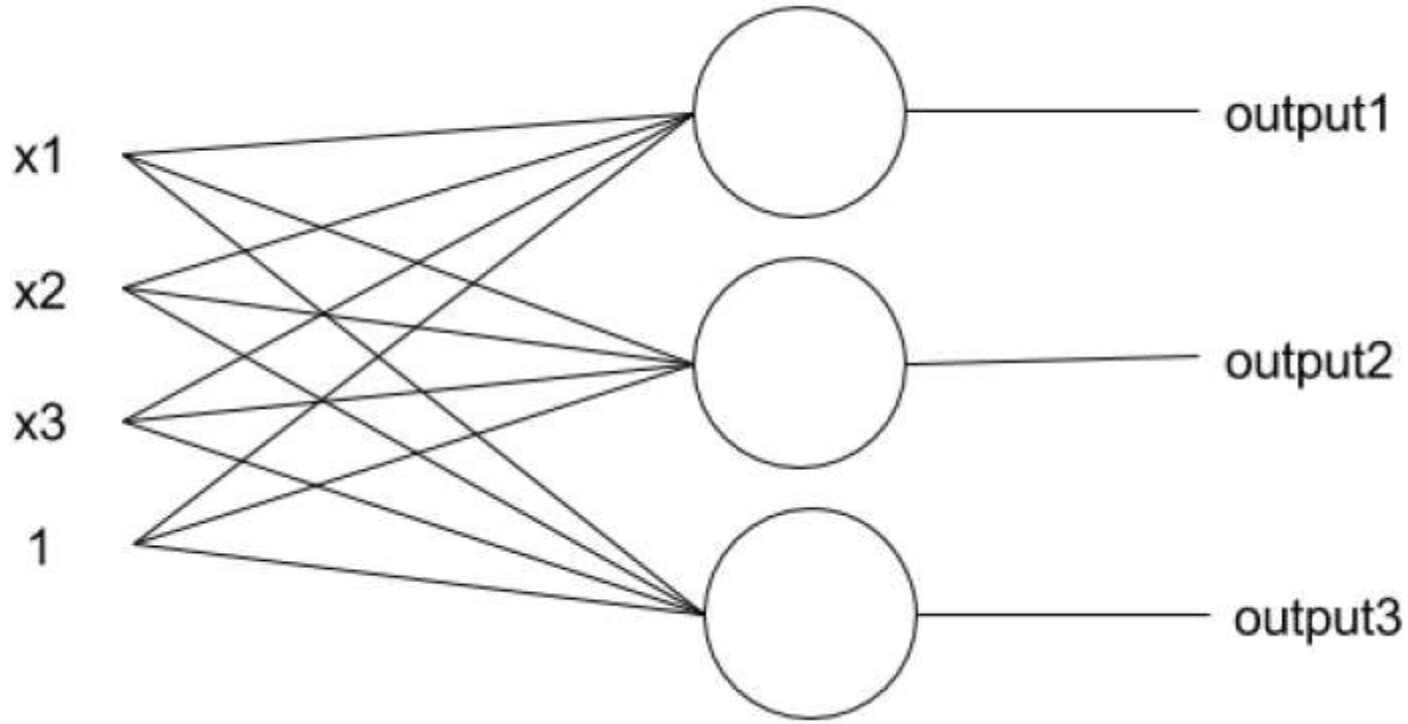
Simple Neural Networks

- 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 considered the weight of an input feature that equals to 1 for every example. We call that doing 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.
- The following diagram represents each neuron inputs, processing and output:



Simple Neural Networks

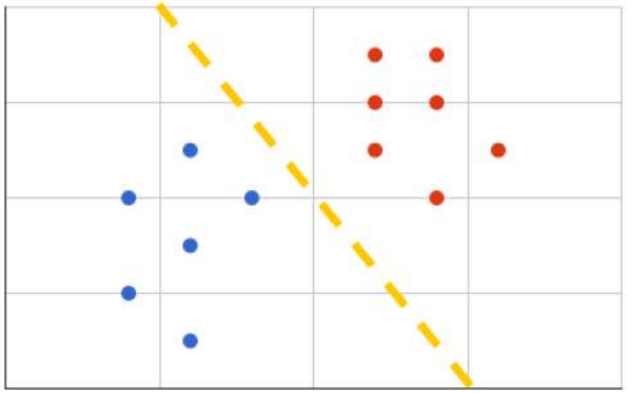
- In the case of softmax classification, we use a network with K (3) neurons, one for each possible output class:



- In order to resolve more difficult tasks, like reading handwritten digits, or actually identifying cats and dogs on images, we are going to need a more complex model.

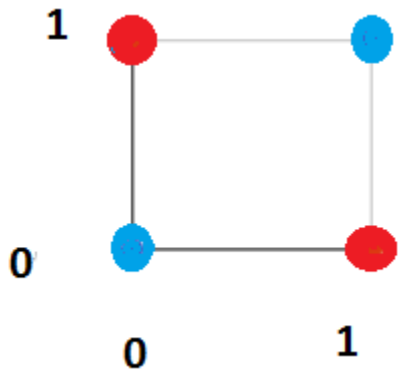
A straw that broke camels back

- Sigmoid type of neurons require our data to be *linearly separable* to do their job well. That means that there must exist a straight line in 2 dimensional data (or hyperplane in higher dimensional data) that separates all the data examples belonging to a class in the same side of the plane, which looks something like this:



- 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.

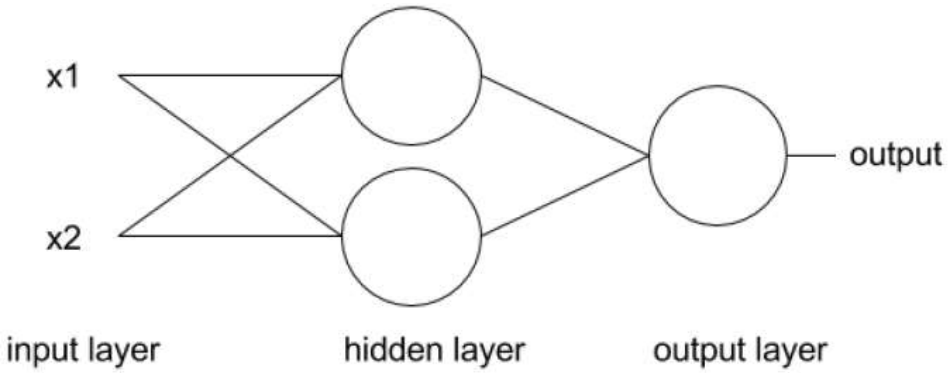
- If you recall our XOR problem.

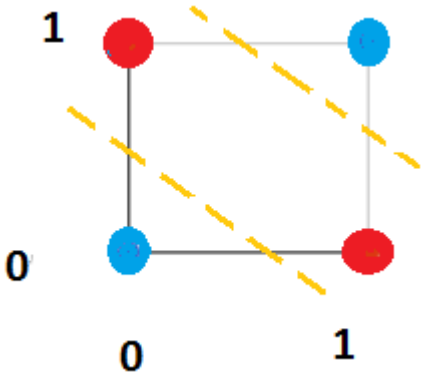


We can't find a single straight line that would split the chart, leaving all of the 1s (red dots) in one side and 0s (blue dots) in the other. That's because the XOR function output is not linearly separable. This problem actually resulted in neural network research losing importance for about a decade around 1970's. Problem got fixed intercalating more neurons between the input and the output of the network, as you know from HW 08

Multi-layer Network

- The lack of linear separability in a single layer network was solved by intercalating more neurons between the input and the output of the network, as you can see in the figure:





We added a *hidden layer* of neurons between the input and the output layers.
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, and finally deciding the output result based on the answers of those questions.
Graphically, we are allowing the network to draw more than one single separation line.
As you can see in the chart, each line divides the plane for the first questions asked to the input data. Then you can leave all of the equal outputs grouped together in a single
area.

Gradient Descent and Backpropagation

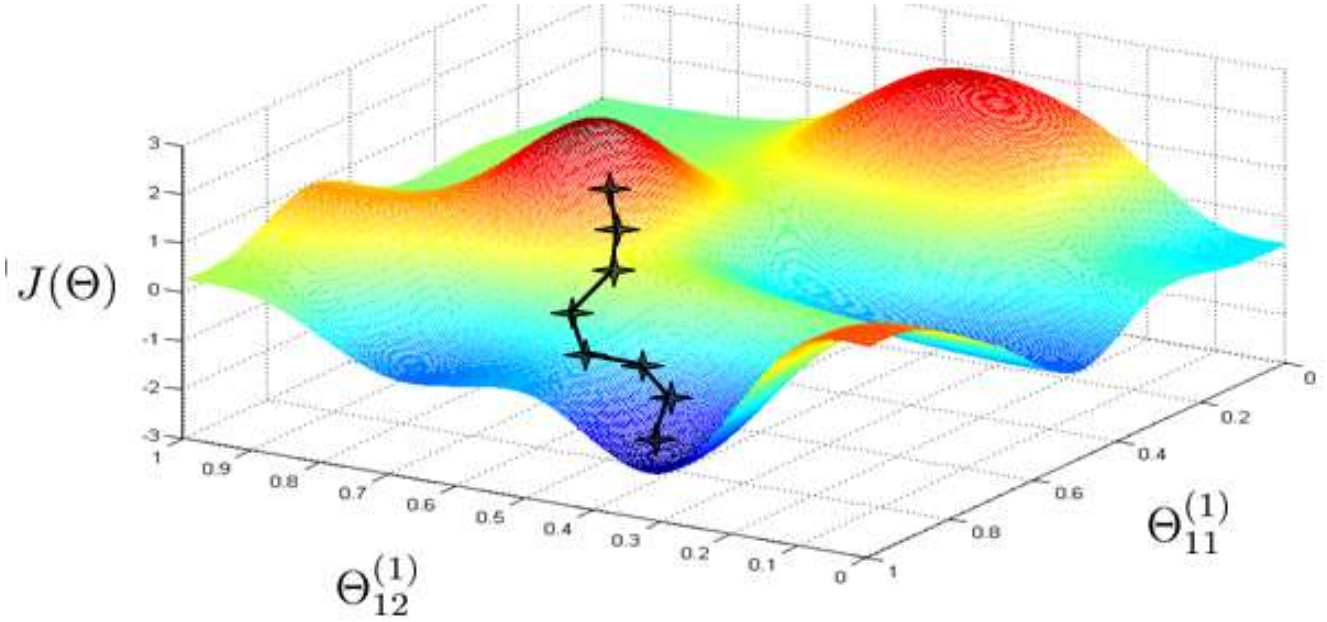
- We need to understand how gradient descent work in multi-layer networks.
- Gradient descent is an algorithm to find the points where a function achieves its minimum value. We defined learning as improving the model parameters in order to minimize the loss through a number of training steps.
- Applying gradient decent to find the minimum of the loss function will result in our model learning from our input data.
- The gradient is a mathematical operation, generally represented with the ∇ symbol (nabla greek letter). It is analogous to a derivative, but applied to functions in several dimensions, that take a vector as in input 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.

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

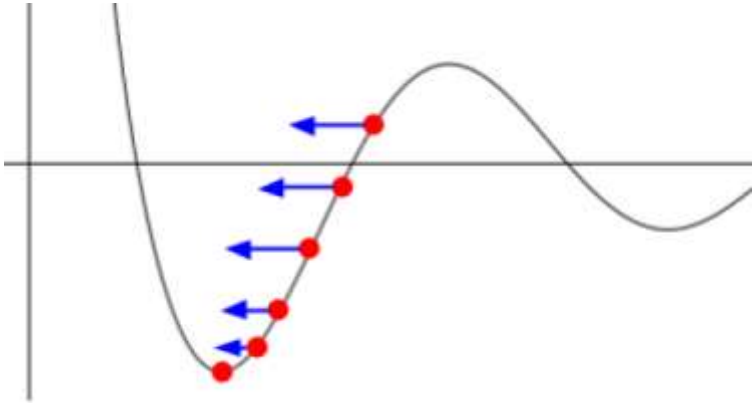
- The partial derivatives measure the rate of change of the function output with respect to a particular input variable. In other words, how much the output value will increase if we increase that perticular input variable value

Gradient Descent

- When we talk about input variables of the loss function, we are referring to the model weights, not that actual dataset features inputs. Those 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:



Gradient Descent



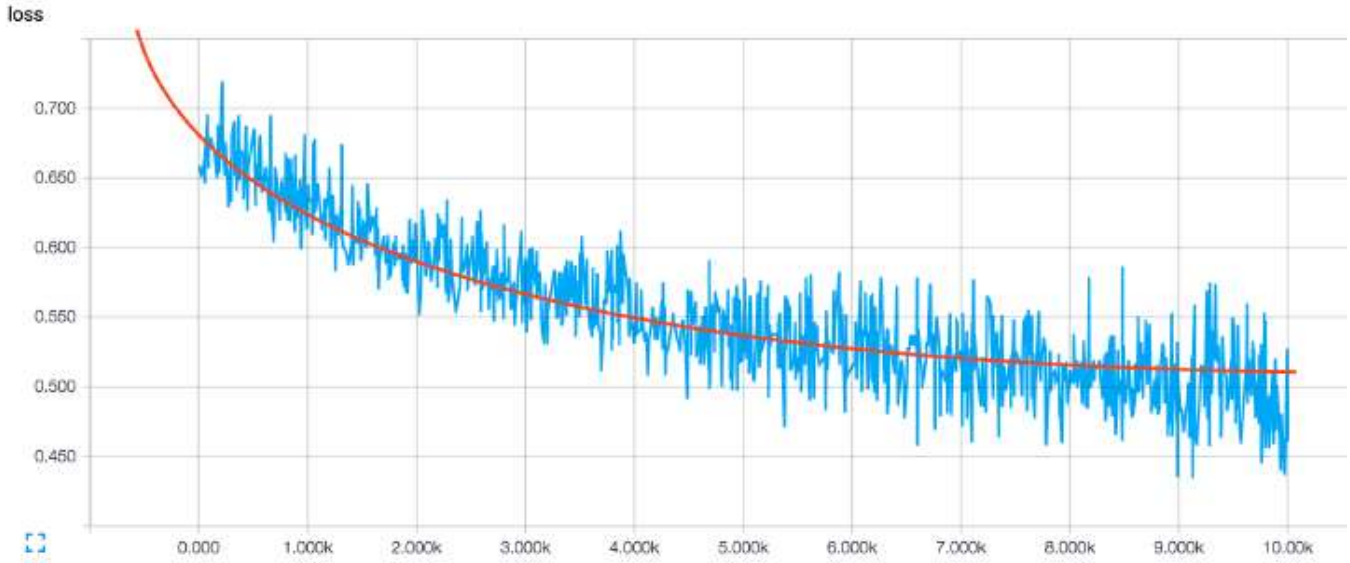
- We can simply define gradient descent algorithm as:

$$weights_{step\ i+1} = weights_{step\ i} - \eta \cdot \nabla loss(weights_{step\ i})$$

- We added the η value to scale the gradient. We call it the learning rate. The learning rate is not a value that model will infer. It is an *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.

Loss Function over time

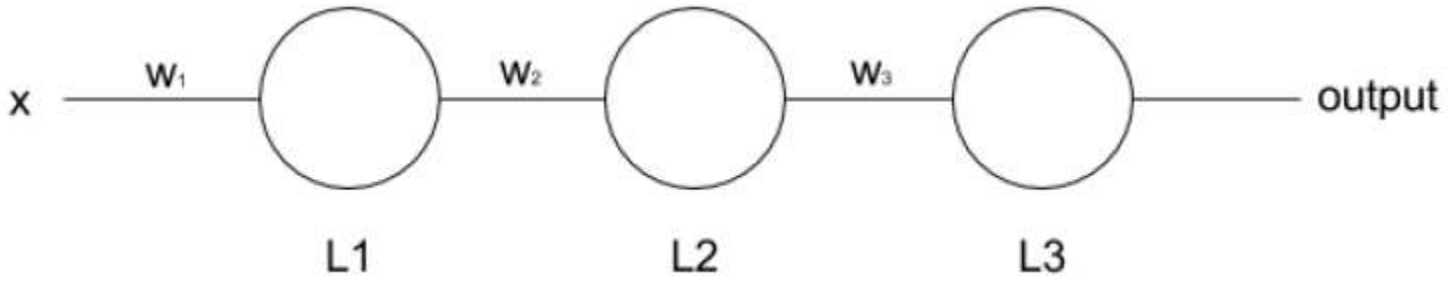
- In practice, we can't really plot the loss function because it has many variables. So to know that we are trapped in overshooting, we have to look at the plot of the computed total loss thru time, which we can get in TensorBoard by using a tf.summary.scalar() on the loss.
- This is how a well behaving loss should diminish through time, indicating a good learning rate:



- The blue line is the TensorBoard chart, and the red one represents the tendency line of the loss.

Calculation of Derivatives, Backpropagation

- So far we haven't been explicitly calculating any derivatives here, because we didn't have to. Tensorflow includes the method tf.gradients to symbolically computate the gradients of the specified graph steps and output that as tensors. We don't even need to make manual calls, because it also includes implementations of the gradient descent algorithm, among others. That is why we present high level formulas on how things should work without requiring us to go in-depth with implementation details and the math.
- We still need some understanding of backpropagation. It is a technique used for efficiently computing the gradient in a computational graph.
- Let's assume a really simply network, with one input, one output, and two hidden layers, each 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:



Backpropagation

- Let's define $L1$ as the output of first hidden layer, $L2$ the output of the second, and $L3$ the final output of the network:

$$L1 = sigmoid(w1 * X)$$
$$L2 = sigmoid(w2 * L1)$$
$$L3 = sigmoid(w3 * L2)$$

- Finally, the loss of the network will be:

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

- 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:

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

- $L2$ is just a constant for this case as it doesn't depend on $w3$. To simplify the expression we could define:

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

Backpropagation

- The resulting expression for the partial derivative would be:

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

- Now let's calculate the derivative for the second hidden layer weight, *w*2:

$$L2' = sigmoid'(w_2 \,.\, L1)$$

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

- And finally the derivative for *w*1:

$$L1' = sigmoid'(w_1 \,.\, x)$$

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

Pattern of BackPropagation

- You should notice a pattern. The derivative on each layer is the product of the derivatives of the layers after it and 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.