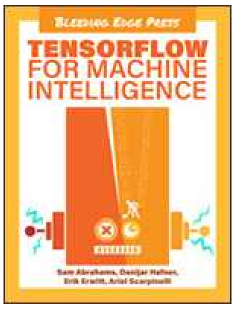


TensorFlow for Machine Intelligence: A Hands-On Introduction to Learning Algorithms

by Sam Abrahams, Danijar Hafner, Erik Erwitt and Ariel Scarpinelli

Bleeding Edge Press. (c) 2016. Copying Prohibited.

Chapter 4: Machine Learning Basics

Overview

Now that we covered the basics about how Tensorflow works, we are ready to talk about its main usage: machine learning.

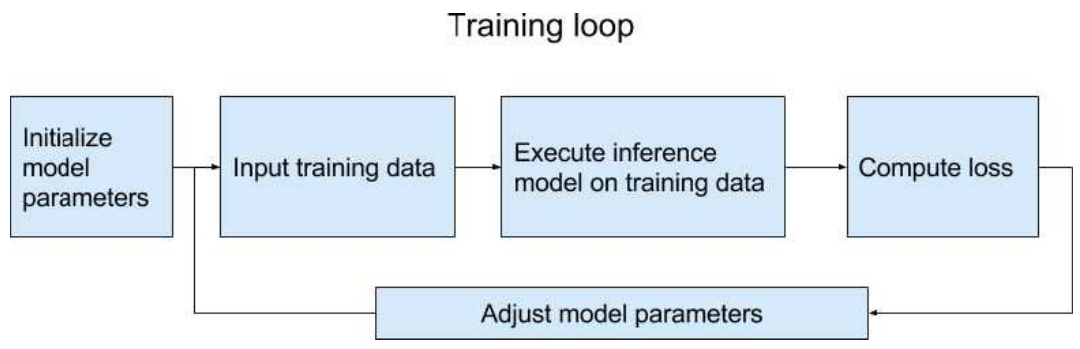
We are going to present high level notions of basic machine learning topics along with code snippets, showing how we work with them in Tensorflow.

Supervised learning introduction

In this book 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. 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 values, like "multiply by 3" or "add 2." These values are the **parameters** of the model, and 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 structure for training them:


Training loop
Initialize model parameters
Input training data
Execute inference model on training data
Compute loss
Adjust model parameters

We create a **training loop** that:

- 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 you can use and are present in the book.
- 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, you can use a gradient descent algorithm for this, which we will explain in the following section.

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.

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 predeicts 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 scaffolding for the model code.

```python
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 total loss

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

# 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=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 decre-
mented 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()
```

This is the basic shape for an inference model code. First it initializes the model parameters. Then it 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. In the following sections we will fill these template methods with required code for each type of inference model.

Once you are happy with how the model responds, you can focus on exporting it and serving it to run inference against the data you need to work with.

Saving training checkpoints

As we stated above, training models implies updating their parameters, or variables in Tensorflow lingo, 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. Luckily, there is the `tf.train.Saver` class to save the graph variables in propietary 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 slighly 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()
```

In the code above we instatiate a saver before opening the session, inserting code inside the training loop to call the `tf.train.Saver.save` method for each 1000 training steps, along with the final step when the training loop finishes. 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. By default the saver will keep only the most recent 5 files and delete the rest.

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

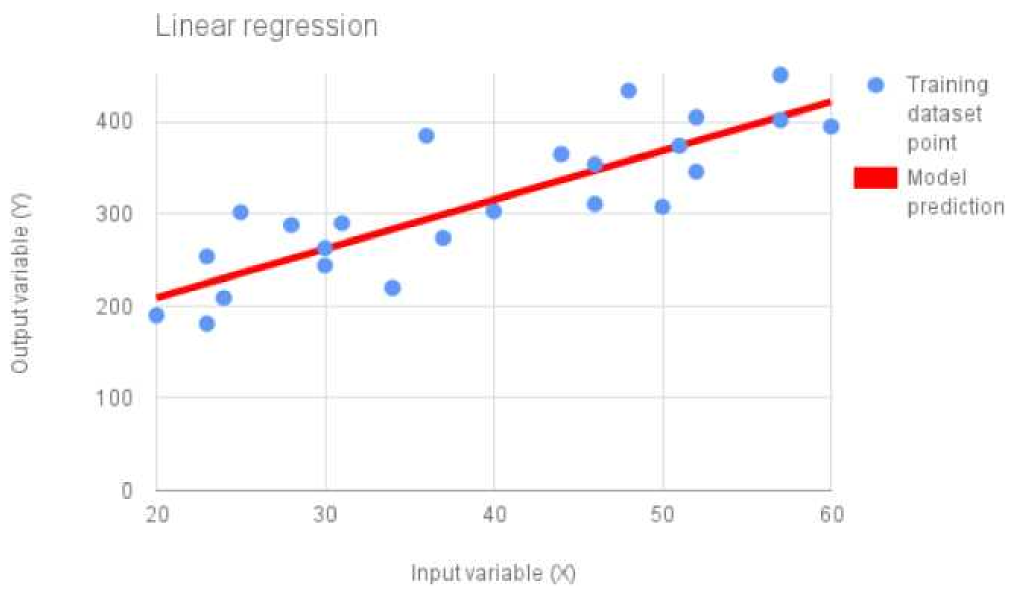
```
with tf.Session() as sess:

    # model setup....

    initial_step = 0

    # verify if we don't have a checkpoint saved already
    ckpt = tf.train.get_checkpoint_state(os.path.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.

Now that we know how supervised learning works in general, as well as how to store our training progress, let's move on to explain some inference models.

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 the charting 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.

Let's begin with a bit of math to explain how the model will work. The general formula 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:

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

- ***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.

Let's represent this model in code. Instead of transposing weights, we can simply define them as a single column vector.

```
# 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
```

Now 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 it is the squared euclidean distance between the predicted output vector and the expected one. Graphically in a 2d dataset is the length of the vertical line that you can trace from the expected data point to the predicted regression line. 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 us 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))
```

It's now time to actually train our model with data. As an example, we are going to work with a dataset that relates age in years and weight in kilograms with blood fat content (http://people.sc.fsu.edu/~jburkardt/datasets/regression/ x09.txt).

As the dataset is short enough for our example, we are just going to embed it in our code. In the following section we will show how to deal with reading the training data from files, like you would in a real world scenario.

```
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_content)
```

And now we define the model training operation. We will use the *gradient descent* algorithm for optimizing the model parameters, which we describe in the following section.

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

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

Now that we trained the model, it's time to evaluate it. Let's compute the expected blood fat for a 25 year old person who weighs 80 kilograms. This is not originally in the source data, but we will compare it with another person with the same age who weighs 65 kilograms:

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

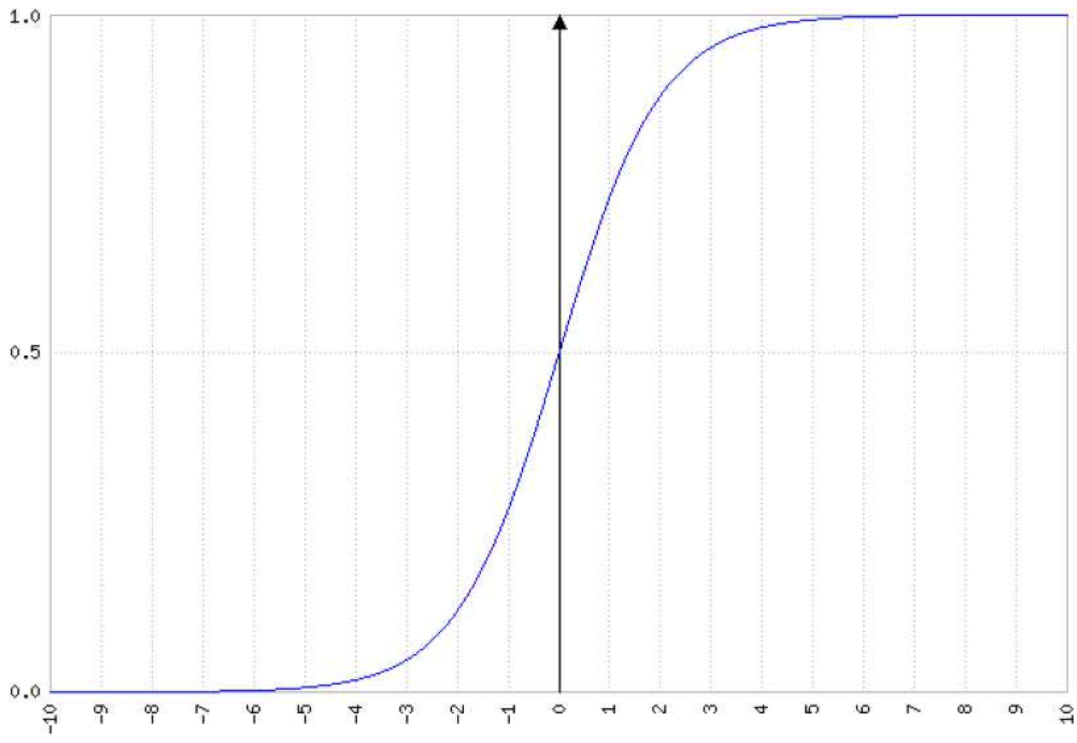
As a quick evaluation, you can check that the model learned how the blood fat decays with weight, and the output values are in between the boundaries of the original trained values.

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).

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



Here you see the charting of a logistic/sigmoid function, with its "S" shape.

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, like we did in the section above.

To express it in code, you can reuse all of the elements of the linear model, however, you just slighlty change the prediction to apply 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))
```

Now let's focus on the loss function for this model. We could use the squared error. 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. Consecuently, a "no" answer will represent 0 probability, hence the loss is any probability the model assigned for that example, and again squared.

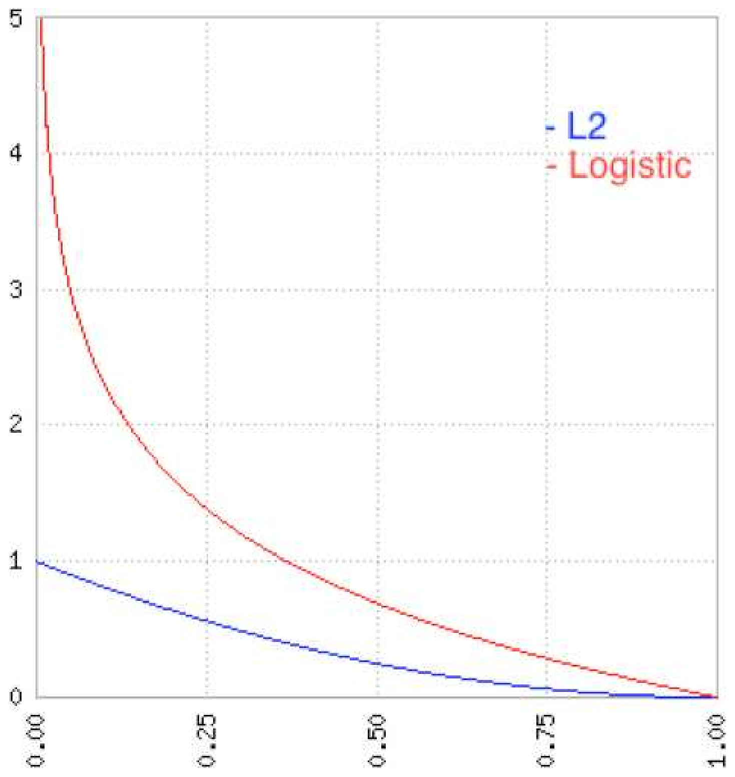
Consider an example where the expected answer is "yes" and the model is predicting a very low probability for it, close to 0. This means that it is close to 100% sure that the answer is "no."

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.

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

We can visually compare the behavior of both loss functions according to the predicted output for a "yes."



The cross entropy and squarred error (L2) functions are charted together. Cross entropy outputs a much greater value ("penalizes"), 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 increases closer to infinity. This makes it impossible for the model to make that misprediction after training. That makes the cross entropy better suited as a loss function for this model.

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_logits(combine_in-
puts(X), Y))
```

What "Cross-entropy" Means

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.

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

You can actually link this entropy with the thermodynamics concept of entropy, in addition to their math expressions being analogous.

For instance, let's calculate the entropy for the word "HELLO."

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

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

$$H = -3 * 0.2 * log_2(0.2) - 0.4 * log_2(0.4) = 1.92193$$

So you need 2 bits per symbol to encode "HELLO" in the optimal encoding.

If you encode the symbols assuming any other probability for the q_i than the real p_i need more bits for encoding 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.

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

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.

$$q("H") = q("E") = q("L") = q("O") = 1/256$$

$$H = -3 * 0.2 * log_2(1/256) - 0.4 * log_2(1/256) = 8$$

So, you need 8 bits per symbol to encode "HELLO" in ASCII, as you would have expected.

As a loss function, consider *p* to be expected training output and distribution probability ("encoding"), where the actual value has 100% and any other 0. And use *q* as the model calculated output. Remember that the sigmoid function computes a probability.

It is a theorem that cross entropy is at its minimum when *p* = *q*. Thus, you can use cross entropy to compare how a distribution "fits" another. The closer the cross entropy is to the entropy, the better *q* is an approximation of *p*. Then effectively, cross-entropy reduces as the model better resambles 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.

Now let's apply the model to some data. We are going to use the Titanic survior Kaggle contest dataset from https://www.kaggle.com/c/titanic/data.

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

To make it a bit more interesting, let's use data from a file this time. Go ahead and download the `train.csv` file.

Here are the code basics for reading the file. This is a new method for our scaffolding. You can load and parse it, and create a batch to read many rows packed in a single tensor for computing the inference efficiently.

```
def read_csv(batch_size, file_name, record_defaults):
    filename_queue = tf.train.string_input_producer([os.path.dir-
name(__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)

    # 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)
```

You have to 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 assigning a number for each possible value. For instance, you can use "1" for first ticket class, "2" for second, and "3" for third. Yet that forces the values to have a lineal relationship between them that doesn't really exist. You can't say that "third class is 3 times first class". Instead you should expand each categorical feature to N boolean features, or one for each possible value of the original. This allows the model to learn the importance of each possible value independently. In our example data, "first class" should have greater probability of survival than others.

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.

In the case of categories with only two possible values, like the gender in the dataset, it is enough to have a single variable for it. 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.

```
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_second_class,
is_third_class, gender, age]))
    survived = tf.reshape(survived, [100, 1])

    return features, survived
```

In the code above we define our inputs as calling `read_csv` and converting the data. To convert to boolean, we use the `tf.equal` method to compare equality to a certain constant value. We also have to convert the boolean back to a number to apply inference with `tf.to_float`. We then pack all the booleans in a single tensor with `tf.pack`.

Finally, lets train our model.

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

As the model computes a probability of the answer being yes, we convert that to a positive answer if the output for an example is greater than 0.5. Then we compare equality with the actual value using `tf.equal`. Finally, we use `tf.reduce_mean`, which counts all of the correct answers (as each of them adds 1) and divides by the total number of samples in the batch, which calculates the percentage of right answers.

If you run the code above you should get around 80% of accuracy, which is a good number for the simplicity of this model.

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 regresion for C possible different values.

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

It returns a probability vector of C components, filling the corresponding probability for each output class. As it is a probability, the sum of the C 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 proof that if the number of classes is 2, the resulting output probability is the same as a logistic regression model.

Now, to code this model, you will have one slight 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, 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.

We are going to use the classical Iris flower dataset for trying softmax. You can download it 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"))
```

Also, as you would expect, Tensorflow contains an embedded implementation of the softmax function.

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

Regarding loss computation, the same considerations for logistic regression apply for fitting a candidate loss function, as the output here is also a probability. We are going to use then cross-entropy again, adapted for multiple classes in the computed probability.

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

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

Summing the loss for each output class on that training example. Note that y_c 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.

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

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

In code, there are two versions implemented in Tensorflow for the softmax cross-entropy function: one 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(com-
bine_inputs(X), Y))
```

The other version of it 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 simple 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.

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, 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, pet-
al_length, petal_width]))
    return features, label_number
```

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.

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)))
```

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.

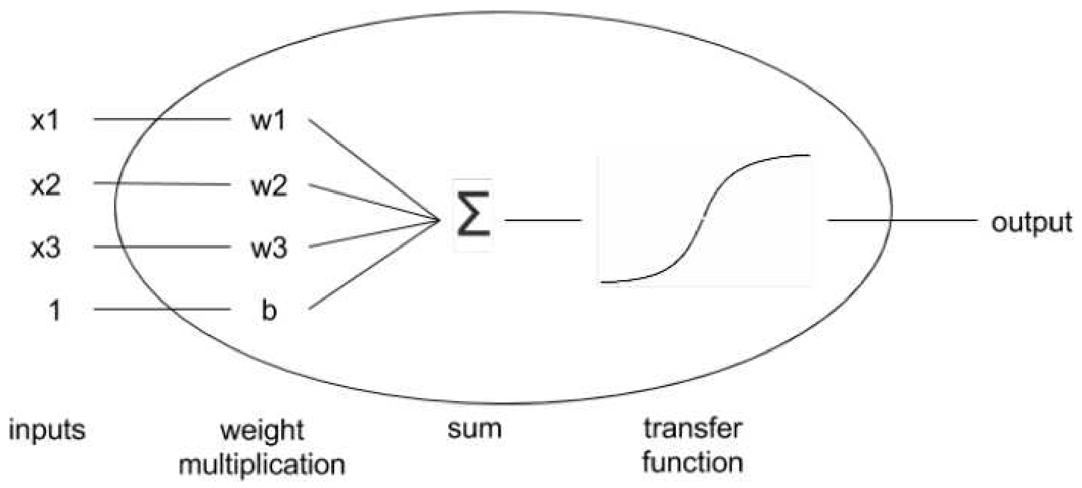
Running the code should print an accuracy of about 95%.

Multi-layer neural networks

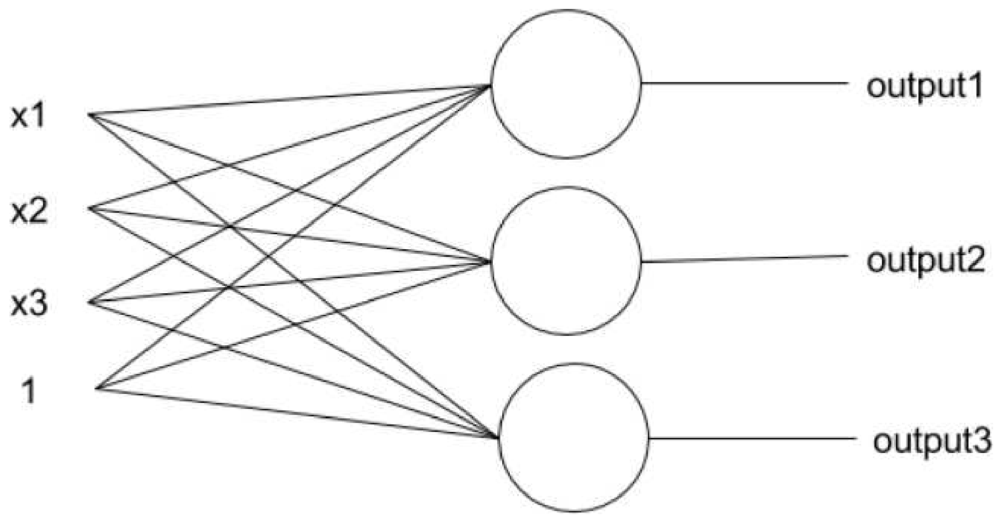
So far we have been using simple neural networks. Both linear and logistic regression models are single neurons that:

- Do a weighted sum of the input features. Bias can be 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:



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



Now, 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 developed model.

Lets start with a simple example. Suppose we want to build a network that learns how to fit the XOR (eXclusive OR) boolean operation:

Table 4-1: XOR

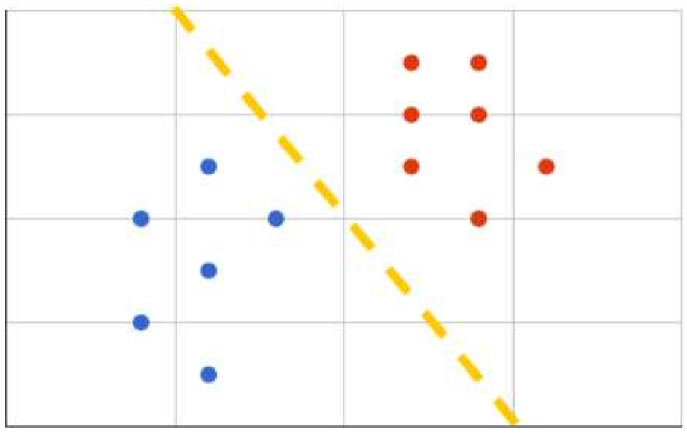
operation truth table

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

It should return 1 when either input equals to 1, but not when both do.

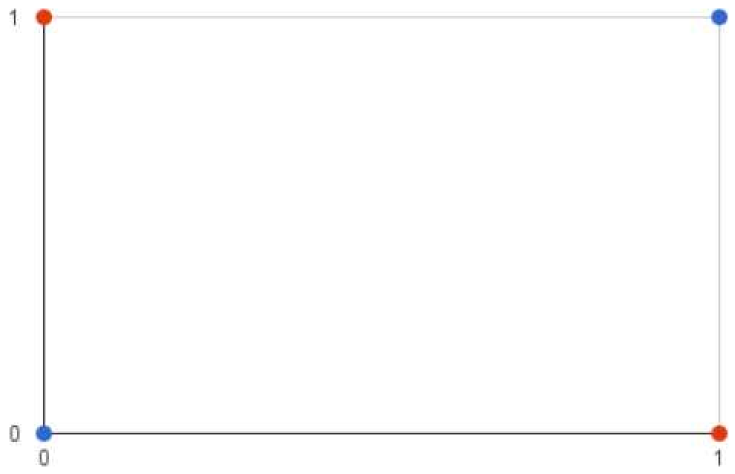
This seems to be a far more simpler problem that the ones we have tried so far, yet none of the models that we presented can solve it.

The reason is that 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:



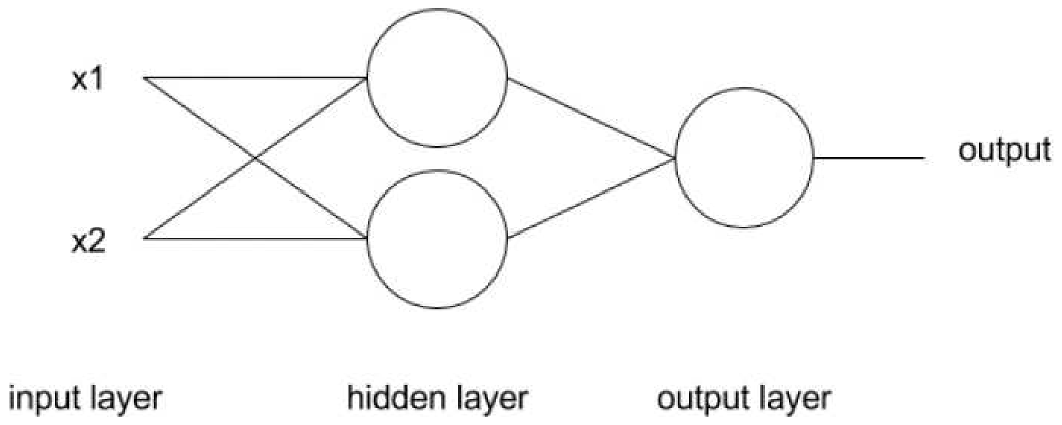
In the chart we can see example data samples as dots, with their associated class as the color. As long as we can find that yellow line completely separating the red and the blue dots in the chart, the sigmoid neuron will work fine for that dataset.

Let's look at the XOR gate function chart:



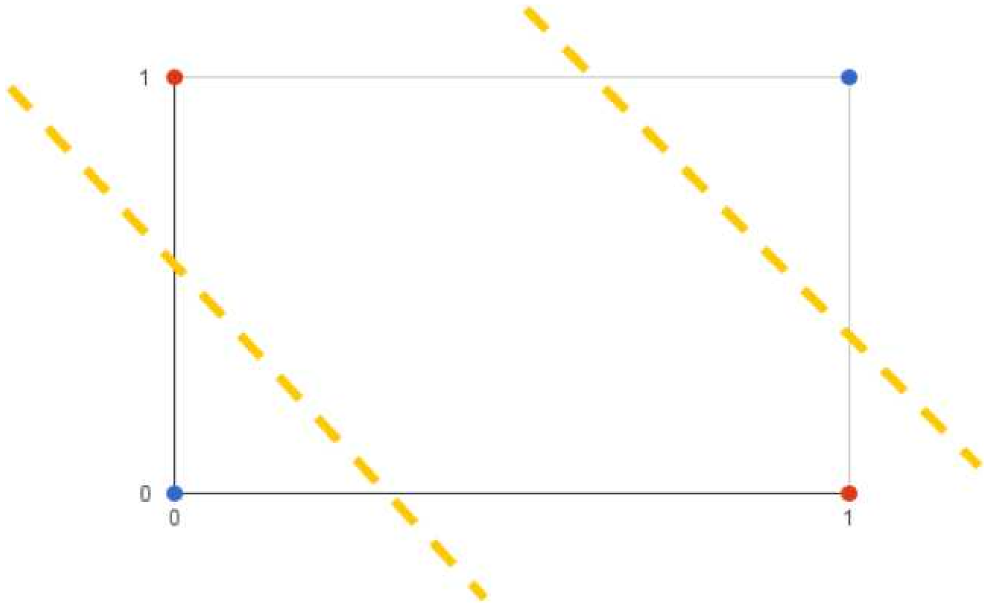
We can't find a single stright line that would split the chart, leaving all of the 1s (red dots) in one side and 0s (blue dots) in the other. That's because the XOR function output is not linearly separable.

This problem actually made neural network research to loss importance for about a decade around 1970's. So how did they fix the lack of linear separability to keep using networks? They did it by intercalating more neurons between the input and the output of the network, as you can see in the figure:



We say that 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 decide 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 toghether in a single area.

You can now guess what the *deep* means in deep learning. We make our networks deeper by adding more hidden layers on them. We may use different type of connections between them and even different activation functions in each layer.

Later in this book we present different types of deep neural networks for different usage scenarios.

Gradient descent and backpropagation

We cannot close the chapter about basic machine learning, without explaining how the learning algorithm we have been using works.

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

Let's define what a gradient is, in case you don't know. The gradient is a mathematical operation, generally represented with the ∇ symbol (nabla greek letter). It is analogous to a derivative, but applied to functions that input a vector and output a single value; like our loss functions do.

The output of the gradient is a vector of partial derivatives, one per position of the input vector of the function.

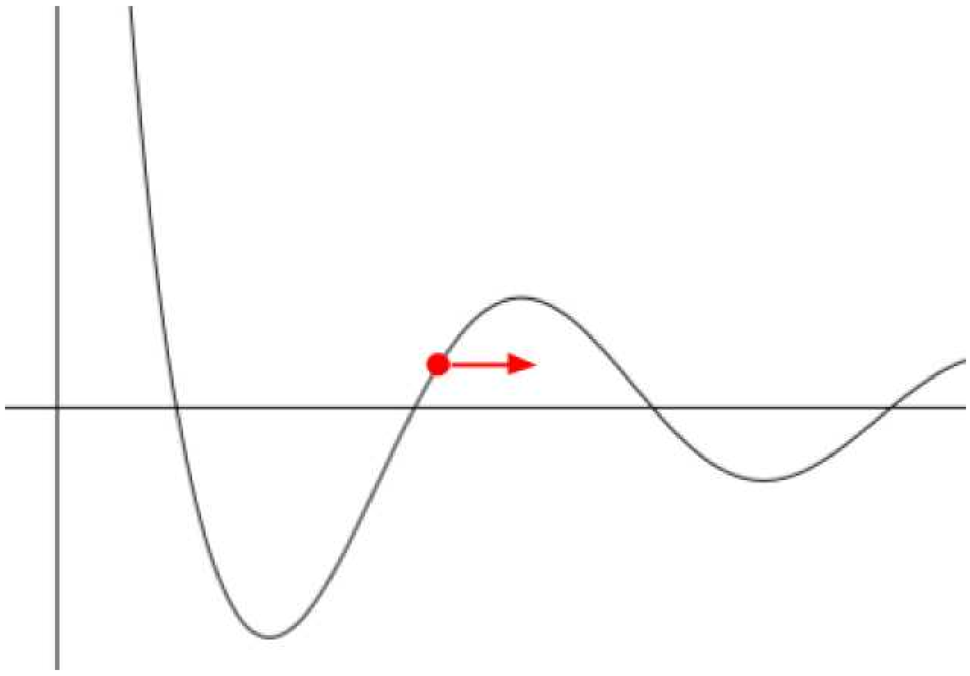
$$\nabla \equiv \left(\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_N}\right)$$

You should think about a partial derivative as if your function would receive only one single variable, replacing all of the others by constants, and then applying the usual single variable derivation procedure.

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

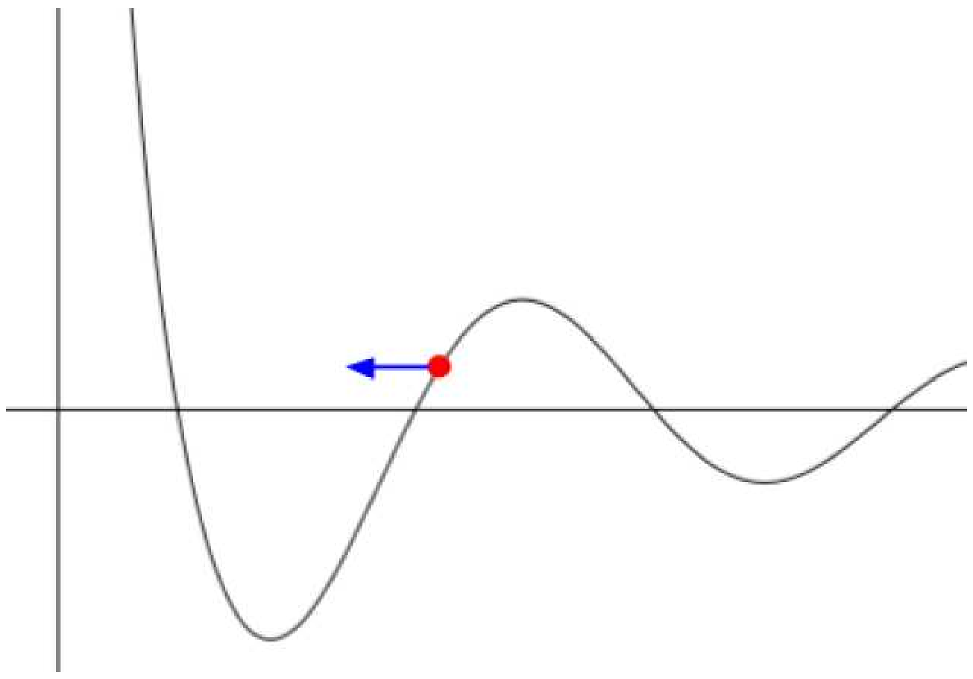
Here is a caveat before going on. When we talk about input variables of the loss function, we are referring to the model weights, not that actual dataset features inputs. Those are fixed by our dataset and cannot be optimized. The partial derivatives we calculate are with respect of each individual weight 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:



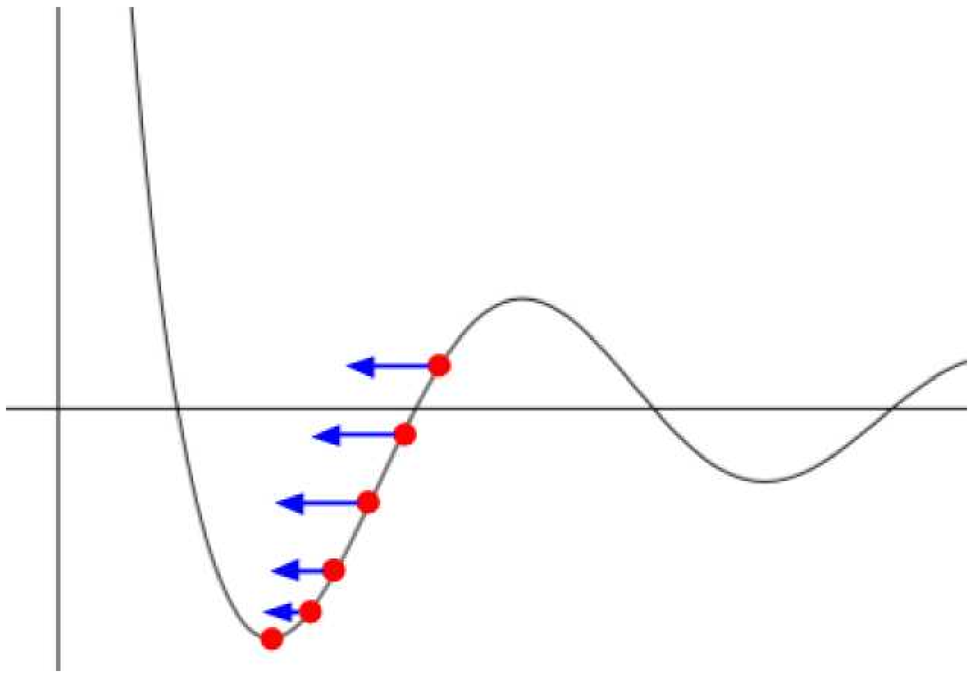
Suppose the chart above shows the loss function. The red dot represents the current weight values, where you are currently standing. The gradient represents the arrow, indicating that you should go right to increase the loss. More over, the length of the arrow indicates conceptually how much would you gain if you move in that direction.

Now, if we go the opposite direction of the gradient, the loss will also do the opposite: decrease.



In the chart, if we go in the opposite direction of the gradient (blue arrow) we will go in the direction of decreasing loss.

If we move in that direction and calculate the gradient again, and then repeat the process until the gradient length is 0, we will arrive at the loss minimum. That is our goal, and graphically should look like:

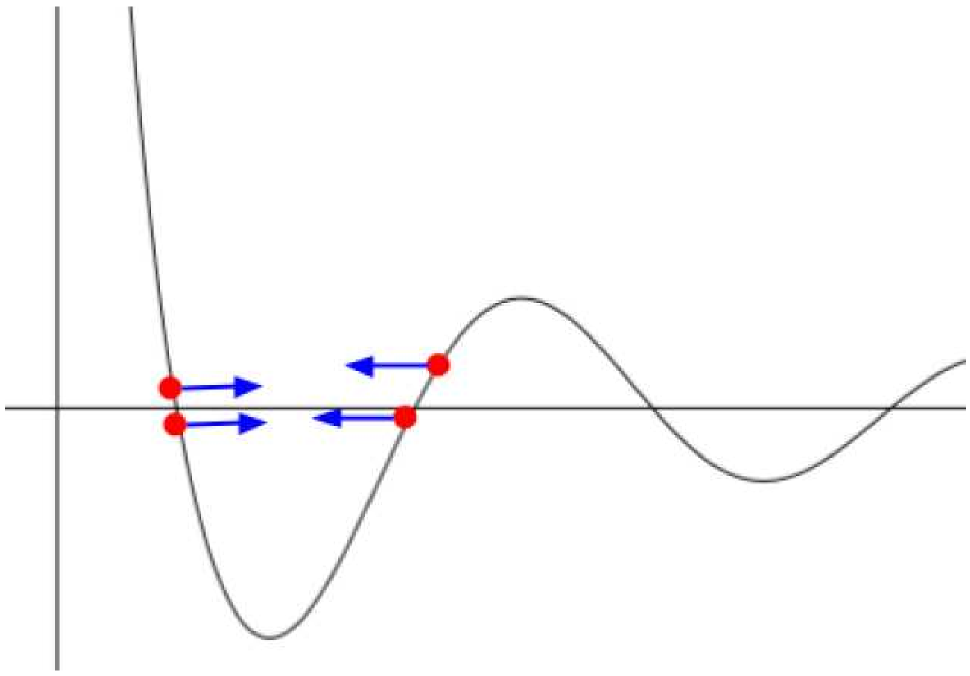


That's it. We can simply define gradient descent algorithm as:

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

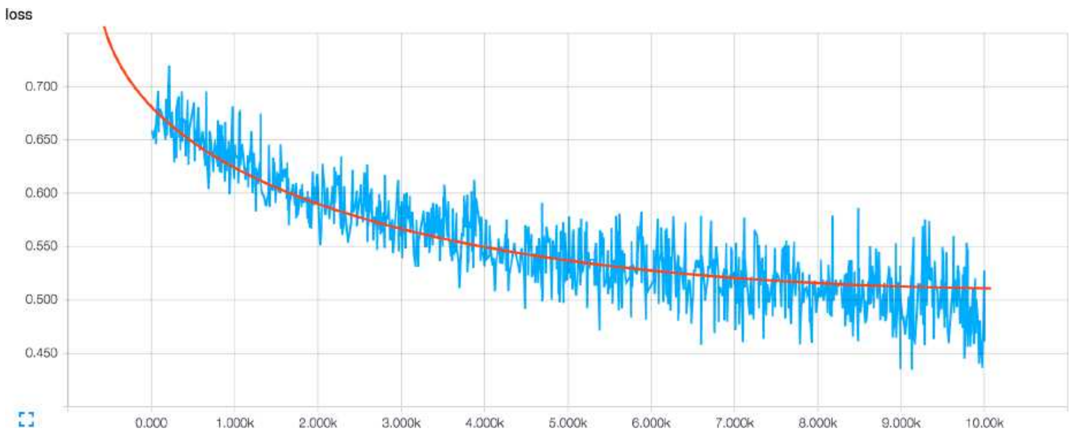
Notice how we added the η value to scale the gradient. We call it the learning rate. We need to add that because the length of the gradient vector is actually an amount measured in the "loss function units," not in "weight units," so we need to scale the gradient to be able to add it to our weights.

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. In our example loss function chart, it would look like:



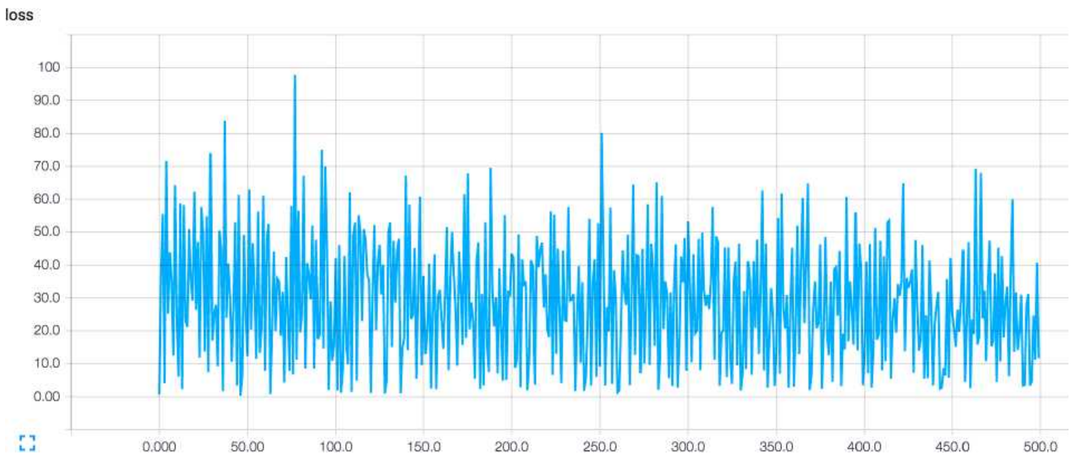
In practice, we can't really plot the loss function because it has many variables. So to know that we are trapped in overshooting, we have to look at the plot of the computed total loss thru time, which we can get in Tensorboard by using a `tf.scalar_summary` on the loss.

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



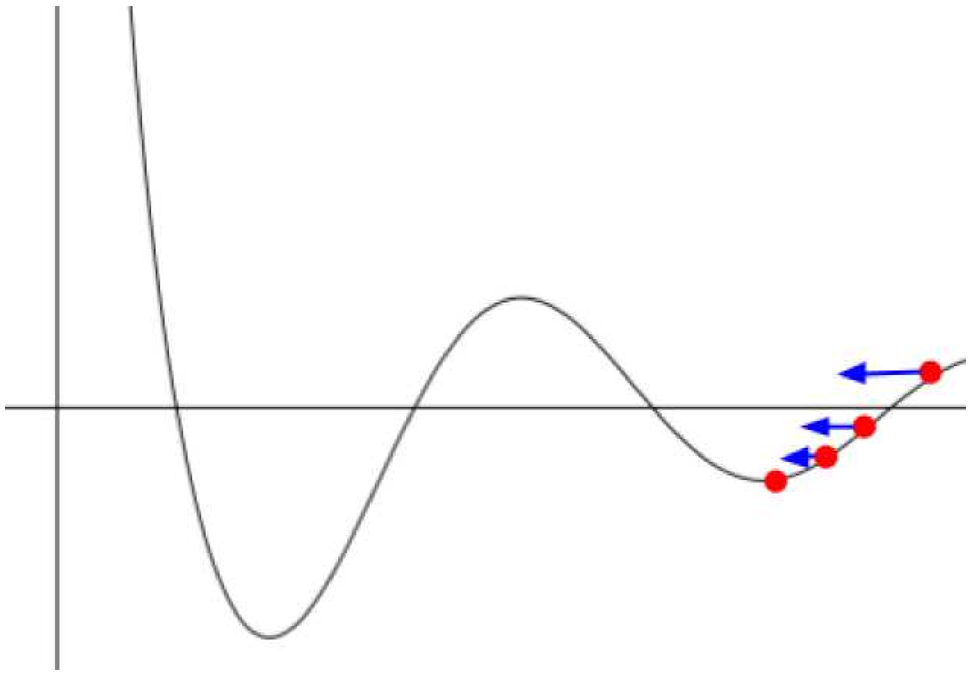
The blue line is the Tensorboard chart, and the red one represents the tendency line of the loss.

This is what it looks like when it is overshooting:



You should play with adjusting the learning rate so it is small enough that it doesn't overshoot, but is large enough to get it decaying quickly, so you can achieve learning faster using less cycles.

Besides the learning rate, other issues affect the gradient descent in the algorithm. The presence of local optima is in the loss function. Going back to the toy example loss function plot, this is how the algorithm would work if we had our initial weigths close to the right side "valley" of the loss function:



The algorithm will find the valley and then stop because it will think that it is where the best possible value is located. The gradient is valued at 0 in all minima. The algorithm can't distinguish if it stoped in the absolute minimum of the function, the global minimum, or a local minimum that is the best value only in the close neighborhood.

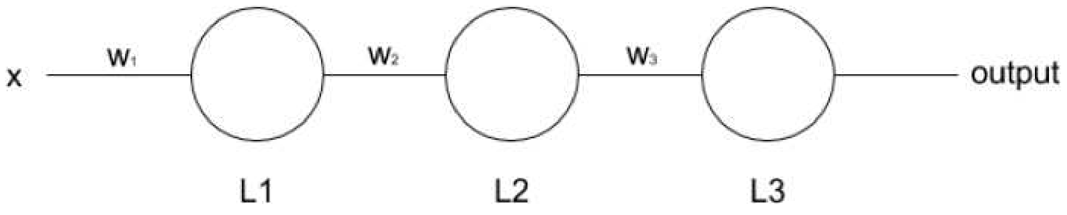
We try to fight against it by intializing the weights with random values. Remember that the first value for the weights is set manually. By using random values, we improve the chance to start descending closer from the global minimum.

In a deep network context like the ones we will see in later chapters, local minima are very frequent. A simple way to explain this is to think about how the same input can travel many different paths to the output, thus generating the same outcome. Luckily, there are papers showing that all of those minima are closely equivalent in terms of loss, and they are not really much worse than the global minimum.

So far we haven't been explicitly calculating any derivatives here, because we didn't have to. Tensorflow includes the method `tf.gradients` to simbolically computate the gradients of the specified graph steps and output that as tensors. We don't even need to manually call, 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 are going to present through 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 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:



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:

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

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

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

Finally, the loss of the network will be:

$$loss = cross_entropy\left(L3, y_{expected}\right)$$

To run one step of gradient decent, we need to calcuate 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 *w*$_3$

To simplify the expression we could define:

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

$$L3' = sigmoid'(w_3 \,.\, L2)$$

Then 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*$_1$:

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

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

You should notice a pattern. The derivative on each layer is the product of the derivatives of the layers after it by the output of the layer before. That's the magic of the chain rule and what the algorithm takes advantage of.

We go forward from the inputs calculating the outputs of each hidden layer up to the output layer. Then we start calcating 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.

Conclusion ~~~~~~~~~~~~

Notice how we have not used the definition of the sigmoid or cross entropy derivatives. We could have used a network with different activation functions or loss and the result would be the same.

This is a very simple example, but in a network with thousands of weights to calculate their derivatives, using this algorithm can save orders of magnitude in training time.

To close, there are a few different optimization algorithms included in Tensorflow, though all of them are based in this method of computing gradients. Which one works better depends upon the shape of your input data and the problem you are trying to solve.

Sigmoid hidden layers, softmax output layers, and gradient descent with backpropagation are the most fundamentals blocks that we are going to use to build on for the more complex models that will see in the next chapters.