Lab 8: TensorFlow (in Pyret!)

Fall 2018

You are **encouraged to work with a partner** on this lab, taking turns driving and navigating.

Contents

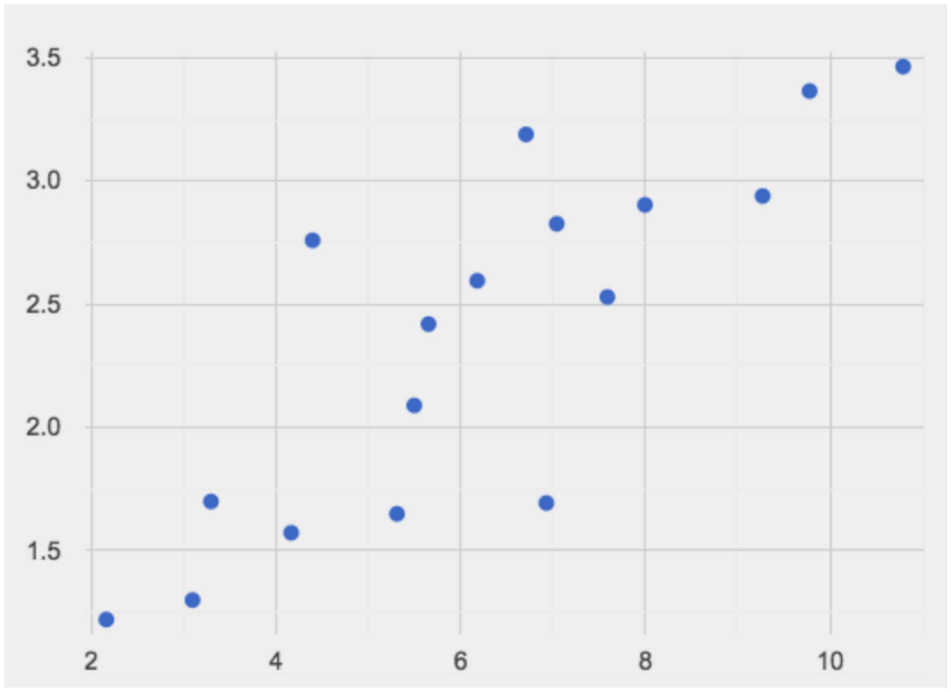
Objectives

By the end of this lab, you will:

- Learn about TensorFlow, a Pyret library for math and machine learning applications
- Use the `tensorflow` library to develop programs that can infer information from data sets and predict future points in the data set

1 Overview

Consider a scenario where you are given some data and want to make predictions based on it. For instance, we might have some data that, when plotted on a graph, looks like this:



For example, this graph might represent *the sizes of test inputs to some function* and the function's corresponding *runtimes on those inputs, in seconds.*

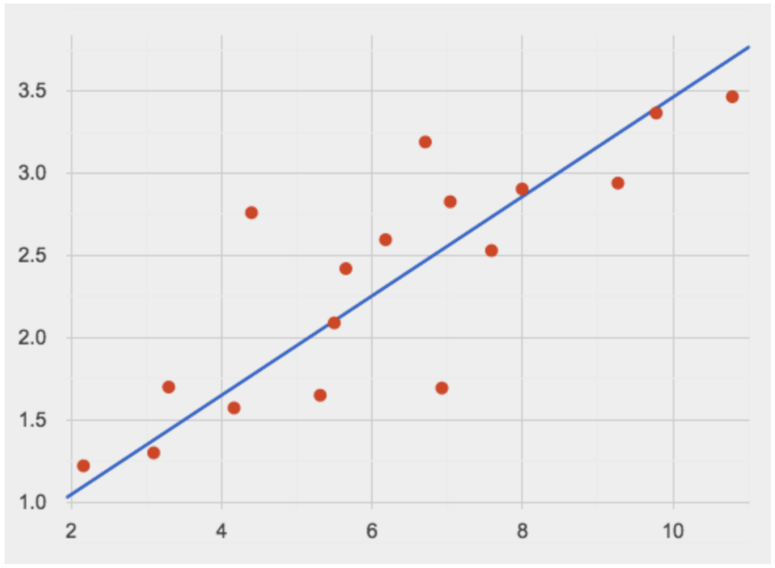
From this data, we may want to predict future performance of our function in some way; for instance, we might want to estimate how long our code will run on a given input size before we start running our function on this input. We assume that there is some relationship between the inputs and the outputs, and, for the purposes of this lab, that the relationship is (initially) "linear". In **machine learning**, given such a data set and the assumption that there is a relationship, we want to discover that relationship so that, for inputs on which we don't already know the answer, we can guess what the answer might be.

Pyret provides us with a `tensorflow` library, which is a library for developing programs with machine learning capabilities. In this lab, we'll cover various applications of TensorFlow and walk through how we can write programs that take advantage of TensorFlow features in Pyret.

2 Linear Regression

In statistics, linear regression is a linear approach to modelling the relationship between a scalar **dependent variable** and one or more **independent variables**. Given a data set of n coordinates (x_i, y_i) where $0 \leq i < n$, our goal is to find some m and b such that the equation $y = mx + b$ most accurately reflects the data points.

For example, the following diagram gives an example of output from a linear regression program, where a line is fitted to points from an original data set (the red dots):



2.1 Designing a Program

We can write a Pyret program that solves linear regression problems using TensorFlow. A TensorFlow program for linear regression is comprised of three main parts:

- A set of **variables**, which are mutable data that our TensorFlow program will attempt to change over time.

  In this particular program, since we are trying to find some m and b for the equation $y = mx + b$, we'll have two variables `m` and `b` whose values we will continually adjust and iterate over until their values result in a line equation that comes close to the data points.

- An **objective function**, which returns a single scalar value. This objective function provides an empirical way of determining the "closeness" of our predicted line ($y = mx + b$) to the actual data points (x_i, y_i).

  For example, one possible way we could calculate a "measure of difference" between the predicted y-values along the regression line and the actual y-values is by computing the *sum of the absolute differences* between our predicted y-values along the line and the actual y-values for each point (x_i, y_i):

  $$\sum_{\text{for all } i} |\text{actual } y\text{-value} - \text{predicted } y\text{-value}| = \sum_{\text{for all } i} |y_i - (mx_i + b)|$$

  This would work as an objective function for linear regression because as the predicted line comes closer to the actual points, the output of the objective function will become smaller.

In this lab, we'll use an objective function that calculates a measure of difference between the predicted y-values along the regression line and the actual y-values by computing the *mean of the squares of the differences* between the predicted and the actual y-values.

- An **optimizer**, which is a built-in TensorFlow algorithm that uses heuristics to change the value of variables such that the objective function is minimized. Our TensorFlow program will take our objective function, pass it to the optimizer, and modify our `m` and `b` variables in an attempt to find coefficients to our line equation $y = mx + b$ that are close to the actual data points.

2.2 Setting up the Program

Open up this link in your browser: https://pyret-tensorflow.herokuapp.com/editor. This is a specialized instance of Pyret that includes the `tensorflow` library.

Start by opening the Pyret editor. **Copy the following `import` statements and `type` declarations into your code:**

```
import tensorflow as TF
import chart as C
import image as I
import lists as L

type Tensor = TF.Tensor
type Optimizer = TF.Optimizer
type ChartWindow = C.ChartWindow
type Image = I.Image
```

We'll also need some points to perform linear regression against. **Copy the following data set into your code:**

```
train-x = [list:
  3.3, 4.4, 5.5, 6.71, 6.93, 4.168, 9.779, 6.182, 7.59, 2.167,
  7.042, 10.791, 5.313, 7.997, 5.654, 9.27, 3.1]

train-y = [list:
  1.7, 2.76, 2.09, 3.19, 1.694, 1.573, 3.366, 2.596, 2.53, 1.221,
  2.827, 3.465, 1.65, 2.904, 2.42, 2.94, 1.3]
```

To visualize this data set, we can generate a scatter plot. **Copy the following line to your code file:**

```
scatter-plot = C.from-list.scatter-plot(train-x, train-y)
```

You can then type `scatter-plot` in the REPL to visualize the data as a scatter plot.

Task: Before continuing, double check that you copied everything on the previous pages into your code file exactly as shown. There is no checkoff required for this task.

2.3 Tensors and Variables

At this point, we need to set up our variables.

TensorFlow uses data structures called `Tensor`s. A `Tensor` is essentially an `Array` of `Number`s specifically designed for use in a TensorFlow program. `Tensor`s are initially immutable, but we can create **variable**, mutable `Tensor`s by calling `.to-variable` on an already existing `Tensor`. Calling `.to-variable` on a `Tensor` allows a TensorFlow `Optimizer` to change the value of the `Tensor`, the reasons for which were discussed previously.

There's also another particular type of `Tensor` which is called a *scalar* `Tensor`—these `Tensor`s have a single value. For this program, we'll want to create two scalar, variable `Tensor`s, where one represents the m and the other represents the b in the equation $y = mx + b$.

You can create a scalar `Tensor` using the following function:

```
TF.make-scalar :: (Number -> Tensor)
```

where the argument to `make-scalar` is the initial value of the `Tensor`. For example, to create a variable, scalar `Tensor` with initial value `3`, we could use the following line of code:

```
some-tensor = TF.make-scalar(3).to-variable()
```

For this particular program, you should set the variables to a random positive number between `0` and `1`.[1] (You can generate a random integer from `0` to `n` by using the built-in `num-random` function—think about how you can use this to generate a random value chosen between `0` and `1`.)

Task: Create two `Tensor` variables, `m` and `b`, that are scalar `Tensor`s with an initial value randomly chosen between `0`, inclusive, and `1`, exclusive. These variables should be defined at the top-level of your program (that is, not nested within a block or a function).

Before continuing, call over a TA to check that your answer is right.

2.4 The Objective Function

At this point, we now need to write our **objective function**.

Once we write our objective function, we'll pass it to a TensorFlow `Optimizer`, which will use the objective function to determine how close our current line (determined by

[1]We do this because we don't really know what the coefficients of the line should be to begin with, but by choosing random coefficients, we *might* be able to start optimizing close to where the coefficients of the line should be. Note that we could have initialized the value of the variables to a different value (for example, we could initialize them all to zero) and we would still see close-to-optimal results. On the other hand, there are some initial variable value states that could make it very difficult for our `Optimizer` to minimize our objective function. For the purposes of this lab, you don't have to worry about this—though if you want to learn more about these "hyperparameters" and how they can affect the performance of your optimization programs, take CS142 or CS157.

the current value of the `m` and `b` variables) is to our data set. Then, the `Optimizer` will modify the value of the `m` and `b` variables by randomly pushing them in a slightly more positive or negative direction, then evaluate the objective function with the new values of the `m` and `b` variables to check whether or not its new `m` and `b` choices moved the line in a more fitting position.

How do we know what objective function outputs correspond to "more fitting positions"? In TensorFlow, we correlate *better* results with *smaller* objective function outputs. This means we want to create an objective function that produces larger values when our predicted line is far away from the data set, but produces smaller values when our predicted line is close to the data set.

In this lab, we'll use an objective function that calculates a measure of difference between the predicted y-values along the regression line and the actual y-values by computing the *mean of the squares of the differences* between the predicted and the actual y-values.

Task: Write a function with the following signature:

```
objective :: ( -> Tensor)
```

which consumes no arguments and returns a *scalar* `Tensor` that is the *mean of the squares of the differences* between the predicted y-values at each x-coordinate and the actual y-values. (When we pass our objective function to a TensorFlow `Optimizer` later in the lab, our objective function will not be able to consume any arguments directly, which is why the function we're writing here consumes zero arguments. This is also why we asked you to define the `m` and `b` variables at the top-level of your program, so you can refer to them inside of your objective function without having to pass them in as arguments.)

Below are some `tensorflow` functions that you might find relevant when writing your objective function (read more about them at the `tensorflow` documentation here: `https://cs0190.github.io/tensorflow.html`):

- **Arithmetic Operations**
  - `TF.add-tensors` or `Tensor.add`
  - `TF.subtract-tensors` or `Tensor.subtract`
  - `TF.multiply-tensors` or `Tensor.multiply`
  - `TF.divide-tensors` or `Tensor.divide`
- **Math Operations**
  - `TF.tensor-square`
- **Reduction Operations**
  - `TF.reduce-mean` (Note that this function consumes two arguments, a `Tensor` and a `axis`. For the purposes of this lab, you can ignore the `axis` argument—to get the mean of all of the values in a `Tensor x`, you can call `TF.reduce-mean(x, none)`.)

- **Conversion Operators**
  - `TF.list-to-tensor`

You should start by using the `TF.list-to-tensor` function to convert the `train-x` coordinates to a `Tensor`. Then, use the current values of `m` and `b` to predict what y-values will be generated for the `train-x` coordinates along the regression line. Finally, using these "prediction" y-values, use some combination of `TF.reduce-mean`, `TF.tensor-square`, and `TF.subtract-tensors` to compute the mean of the squares of the differences between the predicted y-values and the actual y-values in `train-y`.

Before continuing, call over a TA to check that your answer is right.

2.5 Optimizers

Now that we have our variables and our objective function, we need to use a TensorFlow `Optimizer` to change the value of our `m` and `b` variables. Recall that we correlate *smaller* objective function outputs with *better* results—our `Optimizer` will iteratively modify the values of our `m` and `b` variables in an attempt to *minimize* the output of the objective function.

Copy the following function into your program:

```
fun train():
  doc: ```Trains the model by using an Optimizer. The optimizer
       will change any variable Tensors used in the function passed
       into it in an attempt the minimize the value of the
       objective function.```
  learning-rate = 0.005
  optimizer = TF.train-sgd(learning-rate)

  optimizer.minimize(objective, empty)
end
```

The above `train` function runs your program through a "stochastic gradient descent" algorithm (hence the `train-sgd` function naming).[2] You don't need to know exactly how this optimization algorithm works, though you should pay special attention to the `learning-rate` constant in the `train` function. This specifies the **learning rate** at which the `Optimizer` attempts to minimize the function.

Here, we've set the rate to `0.005`. Setting this parameter too high can cause the algorithm to never approach smaller values, but setting it too low makes the optimization

[2] We won't cover *exactly* how this algorithm works, but you can read more about it in more depth here: https://en.wikipedia.org/wiki/Stochastic_gradient_descent As a very high-level overview, the `train-sgd` optimizer will modify the value of the `m` and `b` variables by randomly pushing them in a slightly more positive or negative direction, then evaluate the objective function with the new values of the `m` and `b` variables to check whether or not its new `m` and `b` choices moved the line in a more fitting position. After repeating this process several times, it picks final, updated values for `m` and `b` by picking some of the randomly selected values that worked the best. Take CS142 if you want to learn more about how this works.

algorithm very slow as it approaches smaller values. We've done our own experimentation to determine that `0.005` is a good balance between speed and numerical stability, though you are welcome to experiment with other learning rates to see how they affect the training process.

If you call the `train` function in the REPL, then type `m` and `b` into the REPL to see their values before and after calling the `train` function, you should notice that they're slightly different from what they were before.

Task: It's important to check that our `Optimizer` is *actually minimizing* the output of `objective` function over time (given some `m` and `b`). How can we check this? You should be able to show to a TA exactly what you can do to verify this property.

2.6 Visualizing the Regression

Now, let's plot our regression line on our original `scatter-plot` so we have a visual sense of the performance of our `Optimizer`.

Task: Write a function with the following signature:

```
plot :: ( -> ChartWindow)
```

which consumes no arguments and returns a `ChartWindow` containing the original `scatter-plot` and a line in the form $y = mx + b$ where m and b are determined from the current values of the `Tensor`s `m` and `b`, respectively. (You may wish to look up the Pyret `chart` library to learn more about `ChartWindow`s and how they work.)

You can get the current values of `m` and `b` as `Roughnum`s using the following lines of code:

```
current-m = m.data-now().first
current-b = b.data-now().first
```

Before continuing, call over a TA to check that your answer is right.

2.7 Running the Regression Model

Now that we've defined all of the necessary parts to run and visualize our linear regression program, all that's left to do is to run our optimizer several times until we find values for `m` and `b` that are close to minimizing our objective function.

Task: Write a function with the following signature:

```
train-steps :: (Number -> Image)
```

which consumes a `Number` that represents the number of times to call the `train` function you defined earlier. After calling `train` the required number of times, it then returns an `Image` of the scatter-plot / chart overlay, which can be found by calling `plot().get-image()`.

Once you're done, you will have defined a function that consumes some number of "training iterations"—that is, how many times to run the `Optimizer` on our objective function.

You may also wish to print out the current $y = mx + b$ equation after calling `train`, which you can do with the following line of code:

```
print("y = " + num-to-string(m.data-now().first)
    + "x + " + num-to-string(b.data-now().first))
```

Task: Once you've defined your `train-steps` function, determine the number of training iterations that are necessary to get your linear regression program to find a line that looks mostly accurate to the data.

Before continuing, call over a TA to check that your answer is right.

3 General Curve Fitting

Pyret already has a linear regression library (see the `statistics` library), but the true power of TensorFlow comes with its ability to optimize non-linear functions. In this section of the lab, you'll develop a new program that expands on your linear regression program to fit a *cubic function* (which can easily be changed to any arbitrary function).

3.1 Setting Up the Program

Open a new Pyret program and **copy the following import statements and type declarations into the file**:

```
import tensorflow as TF
import chart as C
import image as I
import lists as L

type Tensor = TF.Tensor
type Optimizer = TF.Optimizer
type ChartWindow = C.ChartWindow
type Image = I.Image
```

Also, **copy this data-generation function into your file** as well:

```
fun generate-data(
    num-points :: Number,
    coefficients :: {a :: Number, b :: Number, c :: Number, d :: Number},
    sigma :: Number)
  -> {xs :: Tensor, ys :: Tensor}:
  doc: "Generates random data points along a cubic curve."
  a = TF.make-scalar(coefficients.a)
  b = TF.make-scalar(coefficients.b)
  c = TF.make-scalar(coefficients.c)
  d = TF.make-scalar(coefficients.d)

  xs = TF.random-uniform([list: num-points], some(-1), some(1))

  # The below represents ax^3 + bx^2 + cx + d:
  ys = a.multiply(xs.expt(TF.make-scalar(3)))
    .add(b.multiply(TF.tensor-square(xs)))
    .add(c.multiply(xs))
    .add(d)
    .add(TF.random-normal([list: num-points], some(0), some(sigma)))

  # Normalize the y values to the range 0 to 1:
  y-min = TF.reduce-min(ys, none)
  y-max = TF.reduce-max(ys, none)
  y-range = TF.subtract-tensors(y-max, y-min)
  ys-normalized = TF.divide-tensors(TF.subtract-tensors(ys, y-min), y-range)

  {xs: xs, ys: ys-normalized}
end
```

You aren't expected to know exactly how the above `generate-data` function works, but the goal of the function is to provide us an easy way of generating test data that generally follows the path of a cubic function. As a general overview, it consumes three arguments:

- `num-points`, the number of random data points to generate
- `coefficients`, the coefficients of the cubic function to generate random data points around
- `sigma`, a number determining how "spread out" the generated random data should be

Then, **generate a set of data using the following lines of code**:

```
# Generate synthetic data based on a cubic function:
test-data = generate-data(100, {a: -0.8, b: -0.2, c: 0.9, d: 0.5}, 0.04)
train-x = test-data.xs.data-now()
train-y = test-data.ys.data-now()
```

The arguments passed to `generate-data` mean that we're looking to generate 100 data points that are *generally* along the path of the following cubic function:

$$y = -0.8x^3 - 0.2x^2 + 0.9x + 0.5$$

Finally, **plot the points on a scatter plot** using the following line of code:

```
# Plot the random points ahead of time for better perfomance:
scatter-plot = C.from-list.scatter-plot(train-x, train-y)
```

Task: Before continuing, double check that you copied everything on the previous pages into your code file exactly as shown. There is no TA checkoff required for this task.

3.2 Do-It-Yourself Curve Fitting

Using your knowledge of linear regression, it's now your turn to write a Pyret program that uses `tensorflow` to fit the cubic data we generated.

Final Task (challenge, time permitting): Write a Pyret program using `tensorflow` that uses techniques from our linear regression program to "curve fit" the `train-x` and `train-y` data we generated above. More precisely, your goal is to find some coefficients a, b, c, and d for the cubic equation $y = ax^3 + bx^2 + cx + d$ such that the coefficients found come close to the coefficients we used to generate our test data in the `generate-data` function.

You should approach this problem in the same way as we built the linear regression program. Just like in our linear regression program, your TensorFlow program for cubic curve fitting should have the following three main parts:

- A set of **variables**, which are mutable structures that are referred to in our objective function. Consider what variables are necessary given the problem specification above.
- An **objective function**, which returns a single scalar value that we are attempting to *minimize*. You should be able to reuse most of your `objective` function from the linear regression program—with the small change that your objective function handles data that follows a cubic pattern instead of a linear one.
- An **optimizer**, which is a built-in TensorFlow algorithm that uses heuristics to change the value of variables such that the objective function is minimized. You should be able to reuse the `Optimizer` from your linear regression program by **copying the `train` function** into your curve fitting program.

Once you've completed the above three steps, **you should also develop a `plot` function** that overlays the `scatter-plot` with the current state of the curve $y = ax^3 + bx^2 + cx + d$ where a, b, c, and d are determined from the current values of the variable `Tensor`s you defined. This is similar to the `plot` function from our linear regression program.

Finally, you should also **develop a `train-steps` function** with the following signature:

```
train-steps :: (Number -> Image)
```

which consumes a `Number` that represents the number of times to call the `train` function you copied from your linear regression program. After calling `train` the required number of times, it then returns an `Image` of the scatter-plot / chart overlay, which can be found by calling `plot().get-image()`.

When you're done writing the program, run your `train-steps` function to find your predicted coefficients for the cubic equation. Recall that the arguments passed to `generate-data` produced data points that were generally along the curve described by the equation $y = -0.8x^3 - 0.2x^2 + 0.9x + 0.5$. If you set up your program correctly, your program should find predicted a, b, c, and d coefficients that are close to the originals.

Once you've completed this task, you will have developed a Pyret program that can curve fit a cubic function, which can be expanded into a program that can curve fit any arbitrary function! Wow!

Call over a TA to check that your answer is right. Once a lab TA signs off on your work, youve finished the lab! Congratulations!