Homework 13: Building and Training Models in TensorFlow

Due May 7, 11:59 pm

Worth 20 points

Instructions on writing and submitting your homework can be found on the course webpage at `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2021/STAT679/hw_instructions.html`. *Failure to follow these instructions will result in lost points.* Please direct any questions the instructor.

Note: this is the last assignment of the semester, and it doubles as your final exam. As such, this homework requires a bit more of you compared to previous assignments in terms of reading documentation and learning things (almost) from scratch. This is designed to be a challenging assignment. As such, I do not expect every student to complete it. Don't worry about that. Start early, do your best, and post to the discussion board or come to office hours if you run into trouble.

Another note: this assignment has you training several neural nets. This procedure can be time-consuming, and if we were to run your notebook start to finish, like the grading script usually does, this could take a very long time. For the sake of our sanity (and the happiness of the computer running the grading script), please make sure that you follow the instructions below and turn in Problem 3 in a separate notebook file. Failure to follow these instructions will result in a penalty.

Important: Owing to the university grade submission deadline, you must turn in this assignment in by 11:59pm on May 7th. You **may not** use late days to extend the due date of this assignment.

1 Warmup: Constructing a 3-tensor (2 points)

You may have noticed that the TensorFlow logo, seen in Figure 1 below, is a 2-dimensional depiction of a 3-dimensional orange structure, which casts shadows shaped like a "T" and an "F", depending on the direction of the light. The structure is five "cells" tall, four wide and three deep.

Create a TensorFlow constant tensor `tflogo` with shape 5-by-4-by-3. This tensor will represent the 5-by-4-by-3 volume that contains the orange structure depicted in the logo (said another way, the orange structure is inscribed in this 5-by-4-by-3 volume). Each cell of your tensor should correspond to one cell in this volume. Each entry of your tensor should be 1 if and only if the corresponding cell is part of the orange structure, and should be 0 otherwise. Looking at the logo, we see that the orange structure can be broken into 11 cubic cells, so your tensor `tflogo` should have precisely 11 non-zero entries. For the sake of consistency, the $(0, 3, 2)$-entry of your tensor (using 0-indexing) should correspond to the top rear corner of the structure where the cross of the "T" meets the top of the "F". **Note:** if you look carefully, the shadows in the logo do not correctly reflect the orange structure—the shadow of the "T" is incorrectly drawn. Do not let this fool you!

Figure 1: The TensorFlow logo.

Hint: you may find it easier to create a `numpy` array representing the structure first, then turn that `numpy` array into a TensorFlow constant. **Second hint:** as a sanity check, try printing your tensor. You should see a series of 4-by-3 matrices, as though you were looking at one horizontal slice of the tensor at a time, working your way from top to bottom.

2 Ordinary Least Squares Linear Regression (6 points)

This problem will walk you through building and fitting a simple linear regression model, expanding on the example from lecture. In particular, we will adapt the code from lecture to define a class `LinearModel` that encodes linear regression for p-dimensional data, instead of the 1-dimensional case from lecture. You are free to start from the lecture code, which is available in the demo notebook on the course webpage or canvas, or just copy-paste this (be mindful of weird indentation caused by copying from a pdf!):

```
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([1.0], dtype=tf.float32, name="slope")
        self.b = tf.Variable([1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b
```

1. Override the initialization method to take an optional argument `p`, and initialize `W` and `b` to be tensors with shapes `(p,)` and `(1,)`, respectively, with entries initialized to be independent standard normals. The argument `p` should default to `p=1` and your initialization method should raise an appropriate error in the event that this argument is not a positive integer. **Note:** in the Keras tutorials below, we'll see a different way to do this using the `tf.keras.layers` submodule, which is expressly designed for specifying multilayer perceptrons and neural nets. For this problem, please just use TensorFlow Variable Tensor objects.

2. Implement an additional method `get_dimension` that takes no arguments (other than `self`) and returns the data dimension (i.e., the initialization argument `p`). You are free to store `p` as a class attribute or read `p` from `self.W.shape`.

3. Implement the `__call__` method so that the input `x` is a `tf.Tensor` of shape `(n,p)`, where

   - `n` is a number of observations (not known ahead of time!) and
   - `p` is the dimension specified upon initialization.

   That is, each row of `x` is a vector of `p` predictors. The output of the call should be a `tf.Tensor` of shape `(n,)`, and should encode the prediction of the model applied to each row of `x`. There is no need to perform error checking for this method. We will leave it to TensorFlow to yell at us if `x` is of the wrong type or the wrong shape when we try to multiply it. **Hint:** you will be tempted to write something like `self.W * x + self.b`, but this will not work, owing to the shapes of `W` and `x`. Instead, look at the `tf.tensordot` function.[1]

   As a sanity check, try running the following code snippet (this assumes that you have kept the names `W` and `b` for the slope and intercept; you'll have to update it if you chose something different):

   ```
   linear_model = LinearModel(p=3)
   linear_model.W.assign([1.0,1.0,1.0])
   linear_model.b.assign([1.0])
   x = tf.constant([[0,1,2],[3,4,5]], dtype=tf.float32)
   linear_model( x )
   ```

   The result should be a tensor that looks something like

   ```
   <tf.Tensor: shape=(2,), dtype=float32, numpy=array([ 4., 13.], dtype=float32)>
   ```

4. Following the example from lecture, define a loss function `loss` that takes two arguments, `y_obsd` and `y_pred`, which are two `tf.Tensor` objects of the same shape `(n,)` for some positive integer `n`, and returns the *mean* squared error between the two vectors. That is, it computes

$$\frac{1}{n}\sum_{i=1}^{n}\left(y_i^{\text{obsd}} - y_i^{\text{pred}}\right)^2.$$

   Note that the example from lecture uses `tf.reduce_sum`, so you'll need to read the documentation to find out how to get the mean reduce operation. There is no need to perform any error checking in this function.

5. Continuing to follow the example from lecture, define a function `train` that takes a `LinearModel` object, a `tf.Tensor` `X` with shape `(n,p)`, a `tf.Tensor` `y` with shape `(n,)` and a float `step_size` (i.e., the learning rate parameter in the lecture code) as its arguments.

   ```
   train( model, x, y, step_size )
   ```

[1] `https://www.tensorflow.org/api_docs/python/tf/tensordot`

should compute the gradient of the loss with respect to the parameters and update the model parameters accordingly, scaling the gradient vector by `step_size`.

6. The zip file at

   `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2021/STAT679/HW13_lm.zip`

   contains two Numpy `.npy` files:

   - `lm_X.npy` : contains a 400-by-6 matrix whose rows are the predictor variables in a data set.
   - `lm_y.npy` : contains a 400-dimensional vector whose entries are the responses. The i-th entry of this vector is the response for the i-th row of the matrix in `lm_X.npy`.

   Please include these files in your submission so that we can run your code without downloading them again. **Note:** we didn't discuss reading numpy data from files. To load the files, you can simply call

   ```
   xtrain = np.load('lm_X.npy')}
   ```

   to read the data into the variable `xtrain`, which will be a `numpy` array. You'll need to read the documentation to see how to then turn this into `tf.Tensor` objects, but you should have a good guess already based on how we constructed them in lecture.

7. Use the code written in the previous subproblems to write a training loop to fit your model to the data in `lm_X.npy` and `lm_y.npy` as training data. In the lecture code, we kept the step size parameter fixed for all of training. As mentioned in lecture, this is not necessarily a good idea, because it can lead to weird behavior if we choose the learning rate poorly.[2] There are myriad rules of thumb for choosing the learning rate or choosing ways to make it change over time. A thorough discussion of these is well outside the scope of this course, but I encourage you to play around with the learning rate, including writing a loop in which it changes over time (most likely you'll want it to shrink toward zero as the step size increases). For the sake of consistency, your training loop should proceed for one thousand steps, though it should not take nearly this many steps to drive the loss to near-zero. The easiest way to check that training has worked correctly is to look at the loss and verify that it has stopped decreasing. In the current case, the problem is easy enough that you should see the loss decrease to a fairly small number (say, less than 10). Create a plot of the loss over the course of training and save it in a file called `lm_training.pdf`. Please include this file in your submission.

8. The data was, in reality, generated with

   $$W = (1, 1, 2, 3, 5, 8), \qquad b = -1.$$

   Your estimated parameters should be quite close to these numbers, say, within an additive error of 0.1. If they aren't, that's a good indication that there is a problem in your training loop. Once you are confident that you have trained the model to a near-optimal solution, save your model's learned parameters `W` and `b` (i.e., attributes of your model object) in global variables `W_lm_trained` and `b_lm_trained`, respectively.

[2]This article includes a thorough discussion of this point: `https://towardsdatascience.com/gradient-descent-the-learning-rate-and-the-importance-of-feature-scaling-6c0b416596e1`

3 Training a Basic Neural Net in `tf.keras` (4 points)

In this problem, you'll get your first exposure to training a neural net in TensorFlow Keras. Keras a submodule of TensorFlow, newly incorporated in version 2, which makes it shockingly easy to build fairly complicated models. We will have an overview of some of the basics of neural nets in our last lecture of the semester, Lecture 19, to be released April 26, so you may want to wait until that lecture to tackle this, but it's up to you.

Important: please complete these tutorials in a separate Jupyter notebook from the one in which you complete the rest of the assignment. This notebook should be in a file called `NetID.tutorials.ipynb`, where `NetID` is your NetID, in all lower-case letters.

1. The best way to gain a broad familiarity with TensorFlow Keras is to walk through some of the tutorials. The tutorial at

   `https://www.tensorflow.org/tutorials/quickstart/beginner`

   will walk you through the process of building a simple neural net with dropout and cross-entropy loss, and training the CNN on the famed MNIST data set. Please run the code from this tutorial in your Jupyter notebook for submission. The result of running the code in this tutorial is a pair of `tf.keras.models.Sequential` objects, named `model` and `probability_model`, if you were copy-pasting the code exactly. Save these models in the `SavedModel` format[3] in directories named `digits_prediction` and `digits_probability`, respectively, and include these directories in your submission. **Note:** as a sanity check, you can try loading your models back into memory from these directories using `tf.keras.models.load_model`.[4]

2. The tutorial at

   `https://www.tensorflow.org/tutorials/keras/classification`

   Builds a broadly similar neural net, this time for classifying black and white images of clothing, but gives a more detailed explanation of some of the preprocessing steps and components of the model. Once again, if you're copy-pasting the code exactly, you will end up with two Keras models named `model` and `probability_model`. Save these models in the `SavedModel` format in directories named `fashion_prediction` and `fashion_probability`, respectively, and include these directories in your submission. **Note:** some of the variable names collide with those from the previous tutorial. This should not be a problem, so long as if you are going to run the first tutorial again, you do so from its beginning so that all the variables (e.g., `model` and `probability_model`) get initialized properly.

4 Building and Training a Logistic Regression (6 points)

In this problem, you'll use TensorFlow to build and train a logistic regression model, which we will discuss briefly in Lecture 19. In this model, the binary response Y is distributed as a Bernoulli random variable with success parameter $\sigma(W^T X + b)$, where

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

[3]See `https://www.tensorflow.org/tutorials/keras/save_and_load#savedmodel_format` and `https://www.tensorflow.org/guide/saved_model`

[4]`https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model`

is the logistic function, X and W are vectors of the same dimension, and b is a scalar.

We will use the `tf.keras` optimization framework to fit this model. Note that this is a different approach from the one we took in the linear regression problem earlier in this assignment. There, we simply computed the gradient and implemented gradient descent ourselves. That is fine for small models, but once we are trying to train larger, more complicated models, it's better to make use of the machinery provided by TensorFlow. Of course, logistic regression isn't actually such a complicated model, but its use of a nonlinearity applied to the output of a linear transformation makes it, in a sense, the simplest neural network we can build, and it suffices to illustrate the basic challenges that arise when we go from simple linear regression to a nonlinear model with a more interesting loss function.

1. Define a class `LogisticModel` that encodes logistic regression for p-dimensional data. You will likely find it easiest to start from the code you wrote above for the `LinearModel` class above. Indeed, you might want to have `LogisticModel` inherit from `LinearModel`, though that is not required. This model should have `__init__` and `__call__` methods, as well as a method `get_dimension`, just like `LinearModel`.

2. Override the initialization method to take an optional argument `p`, and initialize `W` and `b` to be tensors with shapes `(p,)` and `(1,)`, respectively, with entries drawn independently from a standard normal. This optional argument should default to `p=1` and your initialization method should raise an appropriate error in the event that this argument is not a positive integer. You are free to follow an approach similar to that used in `LinearModel` or, if you prefer, you may use the Keras layers framework, which you saw lecture and in the tutorial.

3. Implement an additional method `get_dimension` that takes no arguments (other than `self`) and returns the data dimension (i.e., the initialization argument `p`). You are free to store `p` as a class attribute or read `p` from `self.W.shape`.

4. Implement the `__call__` method so that the input `x` is a `tf.Tensor` of shape `(n,p)`, where

   - `n` is a number of observations (not known ahead of time!) and
   - `p` is the dimension specified upon initialization.

   That is, each row of `x` is a vector of `p` predictors. The output of the call should be a `tf.Tensor` of shape `(n,)`, corresponding to the output of the logistic regression model, evaluated on the rows of `x`. There is no need to perform error checking for this method. We will leave it to TensorFlow to yell at us if `x` is of the wrong type or the wrong shape when we try to multiply it.

5. The zip file at

   `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2021/STAT679/HW13_logistic.zip`

   contains four Numpy `.npy` files that contain train and test data generated from a logistic model:

   - `logistic_xtest.npy` : contains a 500-by-3 matrix whose rows are the independent variables (predictors) from the test set.

- `logistic_xtrain.npy` : contains a 2000-by-3 matrix whose rows are the independent variables (predictors) from the train set.
- `logistic_ytest.npy` : contains a binary 500-dimensional vector of dependent variables (responses) from the test set.
- `logistic_ytrain.npy` : contains a binary 2000-dimensional vector of dependent variables (responses) from the train set.

Download these four files and include them in your submission so that we can run your notebook without downloading the files again.

6. Initialize a `LogisticModel` object called `logreg`, with `p=3`. As a sanity check, if you have loaded `logistic_xtest.npy` into a variable called `logistic_xtest`, you should be able to call `logreg(logistic_xtest)` and get back a tensor with shape `TensorShape[500]`.

7. Following the demo code from Lecture 19 define a loss object[5] called `logreg_loss` that encodes the cross entropy between the output of the model and the observed class labels.

8. Define an optimizer object[6] called `logreg_optimizer`. You are free to choose an optimizer as you see fit, and I encourage you to try different optimizers and experiment with the learning rate (see the discussion of the training loop, below). You will likely find that some optimizers perform better than others on this problem.

9. Define loss and accuracy objects (from the `tf.keras.metrics` submodule) for both the train and test sets. That is, create variables `train_loss`, `train_acc`, `test_loss` and `test_acc`, that encode the loss on the train and test data (measured according to average cross entropy) and the accuracy on the train and test data.

10. Define functions `train_step` and `test_step`, both of which take arguments `x` and `y`, in that order, where `x` is a Tensor of shape `(n,p)` that encodes a collection of `n` predictors (one per row) and `y` is a Tensor of shape `(n,)` that encodes the responses for the `n` predictors. `train_step` should use the `tf.GradientTape` pattern that we saw in Lectures 18 and 19 to compute a gradient step to update the parameters of the instance `logreg` that you created previously. Make sure that you use the `@tf.function` decorator so that your code runs more quickly.

11. Write a training loop that optimizes the parameters of the model stored in `logreg` based on the training data. You can use the test data as a dev set, if you like, to verify that your model is actually learning something. You are encouraged to play around with different choices of the number of training steps, the optimizer, the learning rate... For that matter, if you want to try a different loss function altogether, you are welcome to do so (but if you do, please use a different set of variable names for your experimental model— `logreg` and the code designed to work with that should still be fit according to cross entropy loss). **Note:** this is comparatively noisy data. You are unlikely to be able to push the accuracy past about 0.75. That is fine.

[5] i.e., an object from the `tf.keras.losses` submodule: `https://www.tensorflow.org/api_docs/python/tf/keras/losses`

[6] i.e., an object from the `tf.keras.optimizers` submodule, `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers`

12. Having fit `logreg` to the training data, what is the loss of your model on the test data? Store the answer as a float in a variable `logreg_test_loss`. Similarly, store the training data loss in a variable `logreg_train_loss`. How do they compare?

13. The training and test sets were, in reality, generated with

$$W = (1, 0, -2), \qquad b = 1.$$

Write TensorFlow expressions to compute the squared error in recovering W and b separately. Store those errors in `logreg_W_sqerr` and `logreg_b_sqerr`. `logreg_W_sqerr` should be the squared Frobenius norm of the error between your estimate and the truth. **Note:** you need only evaluate the error of your final estimates, not at every step of training.

5 Freebie: What do we do next? (2 points)

Under normal circumstances, the natural next step would be to go over to Google Cloud Platform and use it to host a trained model to which we can submit instances (e.g., hand-drawn digits) for classification. Unfortunately, there is a gap between TensorFlow and Google Cloud Platform. While TensorFlow has moved on to version 2.X, much of GCP, including the documentation and tutorials, is still built on TensorFlow 1.14 or 1.15. This is further compounded by the fact that TensorFlow 1.15 is incompatible with the default setup of Google Cloud Console. Getting around these hurdles is quite involved, and while you would undoubtedly learn something by going through the process, it is not a good use of your time. We will have to be content with seeing an overview of the process as it would happen in TensorFlow version 1. I encourage you to keep an eye on Google Cloud's support for TensorFlow, which I imagine will extend to version 2 in the near future (this is based entirely on my own speculation and expectations, not on any insider knowledge).

Read over the tutorial at

`https://cloud.google.com/ai-platform/docs/getting-started-keras`,

which walks through the process of building and training a TensorFlow Keras model on Google's AI Platform (previously called ML-Engine), and then deploying that model so that we can submit instances (i.e., observations) for the model to classify.

You are welcome to try and complete the tutorial, but you **are not** required to do so. This problem is a free 2 points. Just create a variable in your Jupyter notebook file called `tutorial` with the Boolean value `True`.