

p6_classification_tutorial_tensorflow

[Bookmark this page](#)

TUTORIAL (0 POINTS): TensorFlow

For the next questions, you will be using [TensorFlow](#) to train neural net models and testing them. In addition to the [previously released tutorial](#), we have put together a short tutorial to go over the basic concepts of TensorFlow. You can find the code used in this tutorial in the script `tutorial.py`. We also strongly recommend looking at the [TensorFlow tutorial](#) if you want to learn more about it. Even if you've done some of the tutorials, doing all three tutorials will only help!

If you don't have TensorFlow installed already, you should follow [TensorFlow's installation instructions](#). If your installation was successful, you should be able to import tensorflow without getting any errors. We will also be using the `numpy` library.

```
>>> import tensorflow as tf
>>> import numpy as np
```

We have installed TensorFlow with GPU support in all of the [hive machines](#) (except for `hive3`, which seems to have a broken installation), so you may optionally use your `cs188` instructional account to do the project. You may want to use these machines if, for example, you were unable to successfully install TensorFlow in your own machine or you want to use a GPU but you don't have one. You can get an account [here](#) if you don't have one already.

Ops and Tensors

TensorFlow is a programming system in which you represent computations as graphs. Nodes in the graph are called ops (short for operations). An op takes Tensors, performs some computation, and produces Tensors. A Tensor is a typed multi-dimensional array whose value may depend on other Tensors. For example, you can represent a mini-batch of images as a 4D array of floating point numbers with dimensions `[batch, height, width, channels]`.

Computation Graph and Session

A TensorFlow graph is a description of computations. To compute anything, a graph must be launched in a `Session`. A session places the graph ops onto `Devices`, such as CPUs or GPUs, and provides methods to execute them. TensorFlow programs are usually structured into a construction phase that assembles a graph, and an execution phase that uses a session to execute ops in the graph. For example, it is common to create a graph to represent the training of a neural network in the construction phase, and then repeatedly execute a set of training ops in the graph in the execution phase.

Placeholders

We start building the computation graph by creating nodes for the input images.

```
>>> x_ph = tf.placeholder(tf.float32, shape=(2, 1))
```

Here `x_ph` does not have a specific value. It is a `placeholder`—an object that will be replaced by a tensor when we run a computation involving that placeholder. Even though a placeholder does not have an specific tensor value, its type must be specified and its dimensions (also known as `shape`) can optionally be specified. It's typical in deep learning to use 32-bit floating point numbers since the additional precision of 64-bit numbers is unnecessary. In addition, some of the dimensions of the shape can remain unknown by specifying `None` instead of a size. For instance, a shape of `(2, None)` indicates that this placeholder could take the value of a 2-dimensional tensor where its first dimension has to be 2 and its second dimension could be any size.

Variables

We now define the weight matrices `W_var` and biases `b_var`, which are the parameters of our model. We could imagine treating these like additional inputs, but TensorFlow has an even better way to handle them: `Variable`. A `Variable` is a value that lives in TensorFlow's computation graph. It can be used and even modified by the computation. In machine learning applications, one generally has the model parameters be `Variables`. We pass the initial value for each parameter when instantiating each variable. These values are `numpy.array`s, which are the standard way of representing tensors *values*.

```
>>> W_var = tf.Variable(np.array([[1, 0], [0, 1], [1, 0], [0, 1]]), dtype=tf.float32)
>>> b_var = tf.Variable(np.array([[1], [1], [-1], [-1]]), dtype=tf.float32)
```

Tensor

Unlike `numpy.array`s, which represent actual tensor values, TensorFlow's `Tensor` may depend on placeholders and variables instead of having a particular value. That is, it could be evaluated to a `numpy.array` when the session is run.

```
>>> y_tensor = tf.matmul(W_var, x_ph) + b_var
```

Variable Initialization

We initialize both `W_var` and `b_var` as tensors from some `numpy.array`. Before variables can be used within a session, they must be initialized using that session. This step takes the initial values that have already been specified, and assigns them to each variable. This can be done for all variables at once.

```
>>> session = tf.Session()
>>> session.run(tf.initialize_all_variables())
```

You don't need to initialize any variables in your project since all the variables are already initialized for you.

Session

You can evaluate placeholders, variables, tensors and ops by passing them to `session.run`. For example, we can evaluate `W_var`.

```
>>> W_value = session.run(W_var)
>>> print('W_value:\n%r' % W_value)
W_value:
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 1.,  0.],
       [ 0.,  1.]], dtype=float32)
```

However, if you try to evaluate `y_tensor` without passing any other arguments, an error is raised since you need to specify a `numpy.array` to be used to replace the placeholder `x_ph`.

```
>>> y_value = session.run(y_tensor)
...
tensorflow.python.framework.errors.InvalidArgumentError: You must feed a value for placeholder tensor 'Placeholder' with dtype float and shape [2,1]
...
```

To do it correctly, you should pass in a dictionary mapping placeholders to the `numpy.array`s they should replace. You should use the keyword argument `feed_dict` to specify such dictionary.

```
>>> x_datum = np.array([[3], [7]])
>>> y_value = session.run(y_tensor, feed_dict={x_ph: x_datum})
>>> print('x_datum:\n%r\ny_value:\n%r' % (x_datum, y_value))
x_datum:
array([[3],
       [7]])
y_value:
array([[ 4.],
       [ 8.],
       [ 2.],
       [ 6.]], dtype=float32)
```

More on session

You can also specify ops to be performed when running the session. For instance, `W2_op` is an op that updates `W_var` to be twice its value.

```
>>> W2_op = tf.assign(W_var, W_var*2)
```

You can pass a list to `session.run` to do multiple evaluations at a time. So, if you pass in `[y_tensor, W2_op]`, `y_tensor` is evaluated again and `W_var` is updated to be twice as much. The return values of `session.run` are the evaluated `numpy.array`s of the corresponding elements in the list that was passed in.

```
>>> y_value, W2_value = session.run([y_tensor, W2_op], feed_dict={x_ph: x_datum})
>>> print('y_value:\n%r\nW2_value:\n%r' % (y_value, W2_value))
y_value:
array([[ 4.],
       [ 8.],
       [ 2.],
       [ 6.]], dtype=float32)
W2_value:
array([[ 2.,  0.],
       [ 0.,  2.],
       [ 2.,  0.],
       [ 0.,  2.]], dtype=float32)
```

When `y_tensor` is evaluated again, it is using the new value of `W_var`.

```
>>> y2_value = session.run(y_tensor, feed_dict={x_ph: x_datum})
>>> print('y2_value:\n%r' % y2_value)
y2_value:
array([[ 7.],
       [15.],
       [ 5.],
       [13.]], dtype=float32)
```

TensorFlow - Computer Analogy

It can help to think of the following computer analogy when learning TensorFlow: when writing TensorFlow code, you're effectively writing Python code that describes how to build a computer and that describes how to write code to run on that computer. With this analogy, you can have the following mental mapping:

- ops <--> instruction set of a computer (e.g. ADD, MULT)
- Tensors <--> data types (e.g. float, boolean, etc.)
- Graph <--> Your code that applies instructions (ops) to your data types (Tensors)
- Session <--> Actual computer/program interpreter that runs your code
- Placeholders <--> Input to your code (e.g. command line arguments)
- Variables <--> Variables in your code
- Variable initialization <--> Allocating memory for your variables

Running a program requires all of these components. Similarly, writing TensorFlow code requires you to think about all of these components. This isn't a perfect analogy, but this analogy will hopefully help you understand how TensorFlow works.

p6_classification_q4_updates

[Bookmark this page](#)

QUESTION 4 (2 POINTS): Gradient Descent Updates

In this question and the following ones, you will be implementing a gradient descent optimizer, and variants of it, to train neural networks. Neural networks are implemented as `Models` and the optimizers are implemented as `Solvers`.

Models

A neural net is represented by a `Model`, which contains a TensorFlow's computation graph to compute the output of the network. This output depends on the inputs and on the parameters w of the network, which are represented as TensorFlow's `placeholders` and `variables`, respectively. A few models (linear regression, softmax regression and convolutional neural net models) have already been provided for you in `models.py`, so you shouldn't need to modify anything in that file. The parameters of these model will be optimized using a `Solver`. You will be implementing a few solvers in `solvers.py`.

Solvers

The `Solver` addresses the general optimization problem of loss minimization. The typical loss (also known as objective) is the average loss over all m data points throughout the dataset,

$$l(w) = \frac{1}{m} \sum_{i=1}^m l_i(w) + \lambda r(w),$$

where $l_i(w)$ is the loss on data point $x^{(i)}$ and $r(w)$ is a regularization term with weight λ (also known as weight decay). Typical losses include the squared error loss for regression, *[Math Processing Error]* $l_i(w) = ||y^{(i)} - f(x^{(i)}; w)||^2$, and the categorical crossentropy for classification, $l_i(w) = - \sum_c y_c^{(i)} \log(f_c(x^{(i)}; w))$, where f is the neural net function and c are the categories. The regularization term on the weights (e.g. the ℓ_2 norm *[Math Processing Error]* $r(w) = ||w||_2^2$) is typically included in the loss to help prevent overfitting to the training data.

The size of the dataset, m , can be very large, so in practice, we use a stochastic approximation of the objective in each solver iteration by drawing a mini-batch of $k \ll m$ instances,

$$l(w) \approx \frac{1}{k} \sum_{i=1}^k l_i(w) + \lambda r(w).$$

The loss $l(w)$ can be optimized using the gradient descent optimization algorithm. When the stochastic approximation of the loss is used, the algorithm is called stochastic gradient descent.

Gradient Descent Updates

Gradient descent (and its stochastic variations) updates the model parameters w by subtracting it by the gradient of the loss, $\nabla l(w)$, weighted by the learning rate α .

Formally, we have the following formula to compute the updated parameters w_{t+1} at iteration $t + 1$, given the current parameters w_t :

$$w_{t+1} = w_t - \alpha \nabla l(w_t)$$

The learning hyperparameter α has to be chosen and it might require a bit of tuning.

Question

Fill in the `get_updates_without_momentum` method in `solvers.py`, which implements the parameter updates above. This method should return a list of tuples, each tuple being the parameter variable to be updated and the updated parameter tensor, (w_t, w_{t+1}) . To run the autograder for this question:

```
python autograder.py -q q4
```

Hints and observations:

- You should use the member variable `self.learning_rate` for the learning rate α .
- Applying an operation to a TensorFlow `Variable` results in a TensorFlow `Tensor`.

p6_classification_q5_updates_with_momentum

[Bookmark this page](#)

QUESTION 5 (1 POINT): Gradient Descent Updates with Momentum

In this question, you will add momentum to the gradient descent update. In this setting, the model parameters w are updated with a linear combination of the negative gradient $\nabla H(w_t)$ and the previous weight update v_t . The learning rate α is the weight of the negative gradient and the momentum μ is the weight of the previous update.

Formally, we have the following formulas to compute the updated value v_{t+1} and the updated weights w_{t+1} at iteration $t + 1$, given the previous weight update v_t and current weights w_t :

$$v_{t+1} = \mu v_t - \alpha \nabla H(w_t)$$

$$w_{t+1} = w_t + v_{t+1}$$

The learning hyperparameters α and μ have to be chosen and it might require a bit of tuning. Notice that in the extreme case when $\mu = 0$, the update becomes the update that you implemented in question 4.

Question

Fill in the `get_updates_with_momentum` method in `solvers.py`, which implements the updates above. As before, this method should return a list of tuples, each tuple being the variable to be updated and the updated variable tensor. To run the autograder for this question:

```
python autograder.py -q q5
```

Hints and observations:

- You should use the member variables `self.learning_rate` and `self.momentum` for the learning rate α and the momentum μ , respectively.
- Applying an operation to a TensorFlow `Variable` results in a TensorFlow `Tensor`.
- The TensorFlow's variables `vel_vars` correspond to the variables v_t of each parameter of the model.
- The returned updates should include updates of the the variables v_t .

p6_classification_q6_gradient_descent

[Bookmark this page](#)

QUESTION 6 (4 POINTS): Gradient Descent

In this question you will implement gradient descent and its stochastic variants.

Question 6a (2 points): Gradient Descent

Fill in the `solve` method of the `GradientDescentSolver` class in `solvers.py`, which implements the gradient descent algorithm. That is, in every iteration of the algorithm, the parameters should be updated using the gradient of the loss of all the training data. Read the method's docstring for specific details of what you should do for this part. To run the autograder for this part:

```
python autograder.py -q q6a
```

Question 6b (1 point): Stochastic Gradient Descent

Fill in the `solve` method of the `StochasticGradientDescentSolver` class in `solvers.py`, which implements the stochastic gradient descent algorithm. That is, in every iteration of the algorithm, the parameters should be updated using the gradient of the loss of only one data point of the training data. Read the method's docstring for specific details of what you should do for this part. To run the autograder for this part:

```
python autograder.py -q q6b
```

Question 6c (1 point): Minibatch Stochastic Gradient Descent

Fill in the `solve` method of the `MinibatchStochasticGradientDescentSolver` class in `solvers.py`, which implements the gradient descent algorithm. That is, in every iteration of the algorithm, the parameters should be updated using the gradient of the loss of a batch of the training data. Read the method's docstring for specific details of what you should do for this part. To run the autograder for this part:

```
python autograder.py -q q6c
```

Hints and observations:

- Unlike the perceptron algorithm where a single iteration corresponds to iterating over the whole dataset once, a single iteration in `StochasticGradientDescentSolver` and `MinibatchStochasticGradientDescentSolver` corresponds to a single update of the parameters.
- You must use the `MinibatchIndefinitelyGenerator` class provided for you to iterate over the data. This class shuffles the data for you and indefinitely iterates over the data (possibly iterating through the dataset multiple times).
- For every iteration, the training loss should be evaluated with the weights before updating them, but the validation loss should be evaluated with the weights after updating them.
- You should call `run.session(...)` exactly two times per iteration.

p6_classification_q7_softmax_convergence

[Bookmark this page](#)

QUESTION 7 (1 POINT): Softmax Regression Convergence

Instead of using the perceptron algorithm, we are now minimizing a loss function with optimization techniques. Does this approach improve convergence?

Question

Visualize the Softmax Regression's training process on two 2D data sets by running the following commands:

```
python dataClassifier.py --model SoftmaxRegressionModel --data datasetA --iter [iterations]
```

```
python dataClassifier.py --model SoftmaxRegressionModel --data datasetB --iter [iterations]
```

You should replace `[iterations]` with the number of iterations Softmax Regression will run for.

For which datasets does Softmax Regression converge?

Answer the question this question in the `q7` method of `answers.py` returning either 'none', 'a', 'b', or 'both'.

p6_classification_q8_hyperparameters

[Bookmark this page](#)

QUESTION 8 (2 POINTS): Hyperparameter Search

Finding a good set of hyperparameters allows your model to train faster and perform better. Hyperparameters differ from model parameters in that hyperparameters (e.g. learning rate, momentum and batch size) are not part of your model. The training procedure learns the model parameters (e.g. the weights in the Perceptron models) but not the hyperparameters. A validation data set (not the training data set) is used to find the best hyperparameter values.

Fill in the `search_hyperparams` function in `search_hyperparams.py` to find the best hyperparameters values. From a given set of values for the learning rate, momentum and batchsize, the `search_hyperparams` function will return the combination of values that results in the highest validation set accuracy. Run your code with:

```
python autograder -q q8
```

Hints and observations:

- You must use the minibatch SGD solver `MinibatchStochasticGradientDescentSolver`.
- Train the model on training set and evaluate the model on validation set.
- You can train the model using the `solver.solve` method of the solver.
- You can evaluate the accuracy of the model using its method `model.accuracy`.

p6_classification_q9_feature_design

[Bookmark this page](#)

QUESTION 9 (5 POINTS): Digit Feature Design

Building classifiers is only a small part of getting a good system working for a task. Indeed, the main difference between a good classification system and a bad one is usually not the classifier itself (e.g. perceptron vs. softmax regression), but rather the quality of the features used. So far, we have used the simplest possible features: the identity of each pixel (being on/off). You can see the classifier run with this set of features by running the following command:

```
python dataClassifier.py --data largeMnistDataset --feature_extractor basicFeatureExtractor --iter 1000 --learning_rate 0.01 --momentum 0.9
```

To increase your classifier's accuracy further, you will need to extract more useful features from the data. The `enhancedFeatureExtractor` in `features.py` is your new playground. When analyzing your classifiers' results, you should look at some of your errors and look for characteristics of the input that would give the classifier useful information about the label. You can add code to the `analysis` function in `features.py` to inspect what your classifier is doing. For instance in the digit data, consider the number of separate, connected regions of white pixels, which varies by digit type. 1, 2, 3, 5, 7 tend to have one contiguous region of white space while the loops in 6, 8, 9 create more. The number of white regions in a 4 depends on the writer. This is an example of a feature that is not directly available to the classifier from the per-pixel information. If your feature extractor adds new features that encode these properties, the classifier will be able exploit them. Note that some features may require non-trivial computation to extract, so write efficient and correct code.

Question

Add new features for the digit dataset in the `enhancedFeatureExtractor` function. Note that you can encode a feature which takes 3 values [1, 2, 3] by using 3 binary features, of which only one is on at the time, to indicate which of the three possibilities you have. You can prototype your features and check the accuracy that you get with the following command:

```
python dataClassifier.py --data largeMnistDataset --feature_extractor enhancedFeatureExtractor --iter 1000 --learning_rate 0.01 --momentum 0.9
```

Running the softmax regression classifier with the basic features should yield 86.1% on the validation set and 87.0% on the test set. You must achieve an accuracy of at least 88% on the test set (of `largeMnistDataset`) in order to pass this question. For your reference, our implementation yields 88.0% on the validation set and 89.1% on the test set when using the features descibed above. However, you may use any features as long as you achieve an accuracy of at least 88% on the test set.

Hints and observations:

- You may want to use your new features in *addition* to the basic features by concatenating them all.
- The reference implementation uses the basic features (i.e. pixels being on/off) and the 3 enhanced features mentioned above (i.e. one-hot encoding representing the number of white connected components).
- Our reference implementation takes 28s in a Ubuntu 14.04 desktop (using a Titan X GPU), 900s in one of the hive machines (using a GeForce GT 740 GPU), and 82s in a MacBook Pro (using the CPU only).

p6_classification_q10_convnet

[Bookmark this page](#)

QUESTION 10 (2 POINTS): Convnet

As you may have experienced in question 9, hand specifying effective features is difficult so we'll have our model learn them instead! Neural networks, and in particular convolutional neural networks for images, are an effective model for learning features. The resulting learned features from these networks are oftentimes more expressive which results in substantial improvements in accuracy. However, finding a good set of hyperparameters to effectively train a large network can be difficult. Due to the limit of computation resources, the network in this question only has 2 convolution layers and 2 fully-connected layers. In practice, the networks are much bigger and requires GPUs to train in a reasonable amount of time. More information on what a convolutional network is can be found [here](#).

In this question, a full implementation of a convolutional network is provided for you in `models.py`. The code to train and evaluate the model is also provided for you in `dataClassifier.py`. The various commands to run the file can be found by

```
python dataClassifier.py --help
```

Just as in question 8, you should find a set of hyperparameters that give you the best accuracy after the model is trained. For this question, you should choose the learning rate and the momentum so that you achieve at least an accuracy of 96.5% on the MNIST test set when using a batch size of 32 and 500 iterations. You may use the provided script to try out your choice of hyperparameters:

```
python dataClassifier.py -m ConvNetModel -s MinibatchStochasticGradientDescentSolver --batch_size 32 --learning_rate [learning_rate] --momentum [momentum] --iterations 500 --data mnistDataset
```

You can also try multiple hyperparameter combinations using the hyperparameter search code that you wrote for the last question:

```
python dataClassifier.py -m ConvNetModel -s MinibatchStochasticGradientDescentSolver --batch_size 32 --learning_rate [learning_rate0] [learning_rate1] --momentum [momentum0] [momentum1] --iterations 500 --data mnistDataset
```

After you find the set of hyperparameters, write them in the `q10` method of `answers.py`.

Hints and observations:

- Before you train the net to finish, you should see whether the loss decreases over the number of iterations. If not, then you should probably change your hyperparameters.
- It's best to modify one parameter at a time to understand its effect, i.e. use controlled experiments.
- Our reference implementation takes 12s in a Ubuntu 14.04 desktop (using a Titan X GPU), 200s in one of the hive machines (using a GeForce GT 740 GPU), and 153s in a MacBook Pro (using the CPU only).