Fun with TensorFlow

SOLUTIONS - do not distribute!

A 1. Coding Exercise

No coding solutions for part (a1)

In the cell below, construct a computational graph that accepts three inputs and computes both:

1. `x1 * x2`
2. `(x1 * x2) + x3`

Use as few nodes in the graph as possible (i.e. use the fact the 1 is a sub-graph of 2).

Hint: Use the "Both" approach from the previous cell. For performance reasons, when called this way (multiple outputs at once), TensorFlow is "smart" enough to compute `x1 * x2` only once and use it in both 1 and 2.

Evaluate them for each of the following:

- `{x1: 2.0, x2: 3.0, x3: 4.0}`
- `{x1: 5.0, x2: 3.0, x3: 4.0}`
- `{x1: 6.0, x2: 3.0, x3: 4.0}`

In [6]:
```
#### YOUR CODE HERE ####
# Construct the computational graph.
tf.reset_default_graph()

# Execute it.


#### END(YOUR CODE) ####
```

```
[6.0, 10.0]
[15.0, 19.0]
[18.0, 22.0]
```

If you did the exercise correctly, you should see:

```
[6.0, 10.0]
[15.0, 19.0]
[18.0, 22.0]
```

TensorFlow can perform operations on more than just scalars - we'll commonly use it to manipulate vectors, matricies, vectors, matrices, and occasionally higher-order tensors.

A 2. Play with More APIs

No coding solutions for part (a2)

Implement the missing parts of the subsequent code fragments.

Hint: see the TensorFlow Documentation (https://www.tensorflow.org/api_docs/python/tf/) for the APIs in question!

In [11]:
```
tf.reset_default_graph()
W = np.array([[1, 2], [4, 5], [7, 8], [9, 10]])
# Uncomment to help debug:  print('W:\n', W)
x = np.array([[8, 8, 8, 8], [7, 7, 7, 7]])
# Uncomment to help debug:  print('x:\n', x)
b = np.array([4, 3])
# Uncomment to help debug:  print('b:\n', b)

numpy_value = x.dot(W) + b
# Uncomment to help debug:  print('np result:\n', numpy_value)

# Construct placeholders for all the inputs.
W_ph = tf.placeholder(tf.float32)
x_ph = tf.placeholder(tf.float32)
b_ph = tf.placeholder(tf.float32)

# YOUR CODE HERE
# Build an affine transform calculation, xW + b, using tf.matmul and tf.add be
low.
# Be sure to understand the broadcasting discussion above.
# affine_matmul_add =


# This affine
# Create another part of the graph that computes it with tf.nn.xw_plus_b.
# affine_xw_plus_b =

# END YOUR CODE HERE

# Execute the graph.
sess = tf.Session()
affine_matmul_add_val, affine_xw_plus_b_val = sess.run(
    [affine_matmul_add, affine_xw_plus_b],  # Desired nodes to evaluate.
    feed_dict={W_ph: W, x_ph: x, b_ph: b})  # Desired input.

assert np.all(numpy_value == affine_matmul_add_val)
assert np.all(numpy_value == affine_xw_plus_b_val)
```

In [12]:
```
tf.reset_default_graph()

# Use tf.nn.embedding_lookup to extract:
# a) the first row of W.
# b) the second and third rows of W.

W_ph = tf.placeholder(tf.float32)

# YOUR CODE HERE

# first_row =

# second_and_third_rows =

# END YOUR CODE HERE

W = np.array([[1, 1, 1], [2, 2, 2], [3, 3, 3]])

sess = tf.Session()
first_row_val, second_and_third_rows_val = sess.run(
    [first_row, second_and_third_rows],  # Desired nodes to evaluate.
    feed_dict={W_ph: W})  # Desired input.

assert np.all(first_row_val == [[1, 1, 1]])
assert np.all(second_and_third_rows_val == [[2, 2, 2], [3, 3, 3]])
```

In [13]:
```
tf.reset_default_graph()

# In TensorFlow, a vector of dimension n is different than a matrix of dimensi
on n x 1.
# Sometimes, APIs require one or the other.  There are two useful functions...

v = tf.placeholder(tf.float32)

# Assume v is a matrix with a "1" sized dimension at the end (e.g. a 5 x 1 mat
rix).
# Use tf.squeeze to remove the last dimension (e.g. turn that 5 x 1 matrix int
o a vector of length 5).

# YOUR CODE HERE

# squeezed =

# END YOUR CODE HERE

sess = tf.Session()

squeezed_val = sess.run(squeezed, feed_dict={v: [[1], [2], [3]]})
assert squeezed_val.shape == (3,)

squeezed_val = sess.run(squeezed, feed_dict={v: [[[1], [2], [3]]]})
#  Hint: when calling tf.squeeze, always provide "axis", explicitly stating wh
ich dimension to squeeze out.
#  or you may have unintended side effects!
assert squeezed_val.shape == (1, 3)
```

In [14]:
```
tf.reset_default_graph()

# In TensorFlow, a vector of dimension n is different than a matrix of dimensi
on n x 1.
# Sometimes, APIs require one or the other.  There are two useful functions...

v = tf.placeholder(tf.float32)

# Assume v is a vector (e.g. length 5).
# Use tf.expand_dims to add a size 1 dimension (e.g. turn that 5 vector into a
 5 x 1 matrix).

# YOUR CODE HERE

# expanded =

# END YOUR CODE HERE

sess = tf.Session()

expanded_val = sess.run(expanded, feed_dict={v: [1, 2, 3]})
assert expanded_val.shape == (3, 1)
```

In this course, you'll realize that one of the most important debugging techniques is to draw pictures of all your tensors on a scrap piece of paper and make sure that it matches your code.

B. Short Answer Questions

Imagine you want to implement logistic regression:

- `z = xW + b`
- `y_hat = sigmoid(z)`

Where:

1. `x` is a 10-dimensional feature vector
2. `W` is the weight vector
3. `b` is the bias term

What are the dimensions of `W` and `b`? Recall that in logistic regression, `z` is just a scalar (commonly referred to as the "logit").

Draw a picture of the whole equation using rectangles to illustrate the dimensions of `x`, `W`, and `b`. See examples below for inspiration (though please label each dimension). It's fine to do this part on paper and take a photo.

B. Your Answers

SOLUTIONS - do not distribute!

1. [W] = 10 x 1
2. b is a scalar.

Name your image b_answer.png

Batching

Let's say we want to perform inference using your model (parameters `W` and `b`) above on 10 examples intsead of just 1. On modern hardware (especially GPUs), we can do this efficiently by *batching*.

To do this, we stack up the feature vectors in `x` like in the diagram below. Note that changing the number of examples you run on (i.e. your batch size) *does not* affect the number of parameters in your model. You're just running the same thing in parallel (instead of running the above one feature vector at a time at a time).

The red (# features) and blue (batch size) lines represent dimensions that are the same.

C. Short Answer Questions

If we have 10 features and running the model in parallel with 20 examples, what are the dimensions of:

1. `W` ?
2. b ?
3. x ?
4. `z` ?

Hint: remember that your model parameters stay fixed!

C. Your Answers

SOLUTIONS - do not distribute!

1. dim(W) = 10 x 1
2. dim(b) = scalar
3. dim(x) = 20 x 10
4. dim(z) = 20 x 1

D. Short Answer Questions

Recall `y_hat = sigmoid(z)`.

If you were to run your model one example of the time and `z = 0.8` for the first example and `z = -0.3` for the second example, that would result in `y_hat = 0.689` and `y_hat = 0.426`, respectively.

If you run them in parallel/as a batch, the answers shouldn't change.

Answer these questions:

1. What is the shape of `y_hat` when running in a batch, in terms of other dimensions.
2. What is the value of `y_hat` (as a vector) when running on the batch described earlier in this question?
3. Why do you think `sigmoid(vector)` is sometimes referred to as "the sigmoid is applied element-wise"?

D. Your answers

SOLUTIONS - do not distribute!

1. $\dim(\hat{y}) = 20 \times 1$
2. $\hat{y} = [\sigma\left(x[0,] \times W + b\right), \sigma\left(x[1,] \times W + b\right), \ldots, \sigma\left(x[batch - 1,] \times W + b\right)]$
3. Because the non-linearity $\sigma(\mathrm{vector})$ is applied to each element of vector to output the same shape as vector.

E. Short Answer Questions

In deep neural networks, there are often intermediate "hidden layers", a vector per example.

Building on the batching in the previous example, the formulation looks like this:

Assuming we want a hidden layer size of 75 (continuing with 10 features and a batch size of 20 examples), what are the shapes of:

1. W?
2. b?
3. x?
4. z?

E. Your Answers

SOLUTIONS - do not distribute!

1. dim(W) = 10 x 75
2. dim(b) = 1 x 75
3. dim(x) = 20 x 10
4. dim(z) = 20 x 75

Putting it all together

In all of these cases, the optimizer needs:

- A batch of examples of x and y from the training set
- Variables to maintain the current values of the parameters of the model
- A loss function *(such as cross-entropy)*
- An optimization strategy *(such as stochastic gradient descent (SGD))*

F. Coding exercise

No coding solutions for part (a2)

In this section, you don't need to create the graph and session (we've done it for you). Instead, you will simply implement functions (in `graph.py`) that construct parts of a larger graph.

You will first build an affine layer:

$$z = xW + b$$

and then then a stack of fully connected layers (described in more detail below), each implementing

$$h^{(i)} = f(h^{(i-1)}W + b)$$

You'll use the former as a building block for the latter.

F.1 Affine Layer

In particular, your function will accept a TensorFlow Op that represents the value of x and should return value z of desired dimension. You must construct whatever variables you need.

In `graph.py`, implement `affine_layer(...)`

Remember to take a photo of the sketch we ask you to make in the function's comments.

Name your image f1_answer.png and rerun this cell

Hints:

- use `tf.get_variable()` to create variables to store the current values of parameters.
- `W` should be randomly initialized using Xavier initialization (https://www.tensorflow.org/versions/master/api_docs/python/contrib.layers/initializers)
- `b` should be initialized to a vector of zeros
- `a * b` is a element-wise product, but what you'll want here is proper matrix multiplication (`tf.matmul`).

Run the little fragment below until you get your code up and running, then run the more comprehensive unit tests in the cell below that.

In [15]:
```
reload(graph)
with tf.Graph().as_default():
    tf.set_random_seed(0)
    sess = tf.Session()
    x_ph = tf.placeholder(tf.float32, shape=(None, 3))
    y = graph.affine_layer(1, x_ph)  #### <---- Your code called here.
    sess.run(tf.global_variables_initializer())

    print('You should have two trainable variables, one for each of parameters
 W and b: ',
          len(tf.trainable_variables()))
    assert len(tf.trainable_variables()) == 2

    print('These should be a (3, 1) W weight matrix and a (1,) offset.')
    variables = sess.run(tf.trainable_variables())
    print(variables[0].shape)
    print(variables[1].shape)
    assert set([variables[0].shape, variables[1].shape]) == set([(3, 1), (1
,)])

    print('This should be [[-2.36877394]].')
    y_val = sess.run(y, feed_dict={x_ph: np.array([[1, 2, 3]])})
    print(y_val)
    assert y_val.shape == (1, 1)
```

```
You should have two trainable variables, one for each of parameters W and b:
2
These should be a (3, 1) W weight matrix and a (1,) offset.
(3, 1)
(1,)
This should be [[-2.36877394]].
[[-2.36877394]]
```

In [16]:
```
reload(graph)
reload(graph_test)
unittest.TextTestRunner(verbosity=2).run(
    unittest.TestLoader().loadTestsFromName(
        'TestLayer.test_affine', graph_test))
```

```
test_affine (graph_test.TestLayer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.026s

OK
```

Out[16]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

F.2: Fully-Connected Layers

A fully connected layer has the following form (you'll notice this is very similar to logistic regression!):

1. An affine transform $z^{(i)} = h^{(i-1)}W_i + b_i$
2. An elementwise nonlinearity $h^{(i)} = f(z^{(i)})$

Logistic regression can be thought of as a single fully-connected layer where $f = \sigma$ is a sigmoid. We'll use ReLU (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) here instead, but the structure is otherwise the same.

These fully connected layers can be stacked repeatedly to build a deep neural network:

$$\begin{aligned} h^{(0)} &= x \\ h^{(1)} &= f(z^{(1)}) &= f(h^{(0)}W_1 + b_1) \\ h^{(2)} &= f(z^{(2)}) &= f(h^{(1)}W_2 + b_2) \\ &\ldots \\ h^{(i)} &= f(z^{(i)}) &= f(h^{(i-1)}W_i + b_i) \end{aligned}$$

In `graph.py`, implement the `fully_connected_layers()` function.

```
In [17]: reload(graph)
         reload(graph_test)
         unittest.TextTestRunner(verbosity=2).run(
             unittest.TestLoader().loadTestsFromName(
                 'TestLayer.test_fully_connected_layers', graph_test))
```

```
test_fully_connected_layers (graph_test.TestLayer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.116s

OK
```

```
Out[17]: <unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

In [18]:
```
reload(graph)
reload(graph_test)
unittest.TextTestRunner(verbosity=2).run(
    unittest.TestLoader().loadTestsFromName(
        'TestLayer.test_fully_connected_doesnt_use_hidden_dim_as_layer_name',
graph_test))
```

```
test_fully_connected_doesnt_use_hidden_dim_as_layer_name (graph_test.TestLaye
r) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.197s

OK
```

Out[18]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

In [19]:
```
reload(graph)
reload(graph_test)
unittest.TextTestRunner(verbosity=2).run(
    unittest.TestLoader().loadTestsFromName(
        'TestLayer.test_no_fully_connected_layers', graph_test))
```

```
test_no_fully_connected_layers (graph_test.TestLayer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.006s

OK
```

Out[19]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

F.3 Compute logits

Use the functions you've already implemented to build the computational graph taking features x_ph through a forward pass of a fully connected neural network with dimensions `hidden_dims` (a list of integers, like `[50, 35, 10]`).

In `graph.py`, implement `MakeLogits()`

In [20]:
```
reload(graph)
reload(graph_test)
unittest.TextTestRunner(verbosity=2).run(
    unittest.TestLoader().loadTestsFromName(
        'TestLayer.test_make_logits', graph_test))
```

```
test_make_logits (graph_test.TestLayer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.143s

OK
```

Out[20]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

F.4 Compute loss

Given the logits and the labels, compute cross entropy loss.

In `graph.py`, implement `MakeLoss()`

```
In [21]: reload(graph)
         reload(graph_test)
         unittest.TextTestRunner(verbosity=2).run(
             unittest.TestLoader().loadTestsFromName(
                 'TestLayer.test_make_loss', graph_test))
```

```
test_make_loss (graph_test.TestLayer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.025s

OK
```

```
Out[21]: <unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

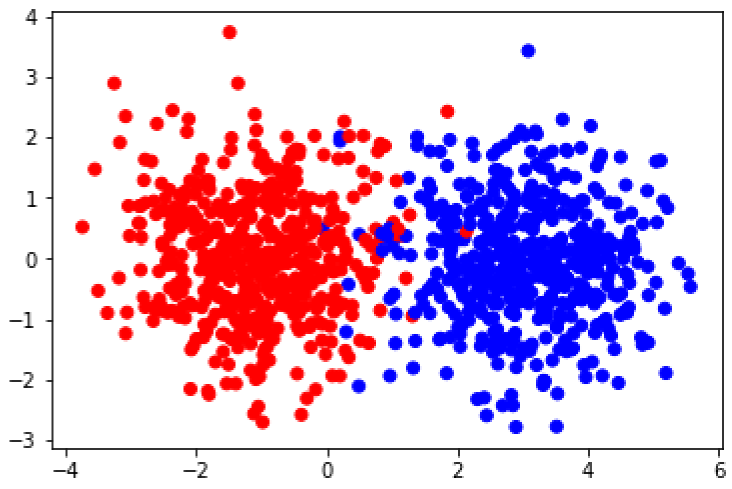
F.5 Training a Neural Network

Let's put it all together, and build a simple neural network that fits some training data.

Read the code for `train_nn()`, in `graph.py`

Note that much of this is boilerplate, but all the elements should be familiar from above: constructing the graph, feeding minibatches from NumPy arrays, and calling `session.run()`. On future assignments, we'll make use of the high-level `tf.Estimator` (https://www.tensorflow.org/programmers_guide/estimators) API to abstract away some of this and allow us to focus on the model structure.

```
In [22]: reload(graph_test)
         X_train, y_train, X_test, y_test = graph_test.generate_data(1000, 10)
         plt.scatter(X_train[:,0], X_train[:,1], c=y_train, cmap='bwr');
```



Hint: You should expect to see an initial loss here of 0.2 - 1.0. This is because a well-initialized random classifier tends to output a uniform distribution. For each example in the batch, we either compute the cross-entropy loss of the label (`[1, 0]` or `[0, 1]`) against the model's output (~`[0.5, 0.5]`). Both cases result in $-\ln(0.5) = ln(2) = 0.69$.

Of course, your random classifier won't output exactly uniform distributions (it's random after all), but you should anticipate it being pretty close. If it's not, your initialization may be broken and make it hard for your network to learn.

[Optional] Some technical details... if your randomly initialized network is outputting very confident predictions, the loss computed may be very large while at the same time the sigmoids in the network are likely in saturation, quickly shrinking gradients. The result is that you make tiny updates in the face of a huge loss.

In [23]:
```
reload(graph)
reload(graph_test)
unittest.TextTestRunner(verbosity=2).run(
    unittest.TestLoader().loadTestsFromName(
        'TestNN.test_train_nn', graph_test))
```

```
test_train_nn (graph_test.TestNN) ...

Initial loss: 0.479213
Step:  20 Loss: 0.467621

ok

----------------------------------------------------------------------
Ran 1 test in 0.177s

OK
```

Out[23]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

In [24]:
```
reload(graph)
reload(graph_test)
unittest.TextTestRunner(verbosity=2).run(
    unittest.TestLoader().loadTestsFromName(
        'TestNN.test_train_nn_with_fclayers', graph_test))
```

```
test_train_nn_with_fclayers (graph_test.TestNN) ...

Initial loss: 0.839931
Step:  20 Loss: 0.804735

ok

----------------------------------------------------------------------
Ran 1 test in 0.388s

OK
```

Out[24]:
```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

That was fairly straightforward... the data is clearly linearly separable.

Tuning Parameters

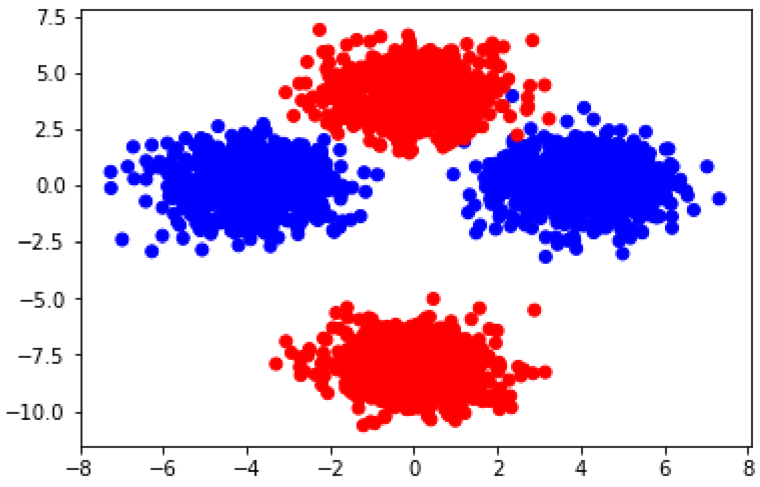
Let's try our network on a problem that's a bit harder! This is a version of the classic XOR problem, which is not linearly-separable. However, it's easy to solve with a deep network.

Here, we'll train a neural network with a couple of hidden layers before the final sigmoid. This lets the network learn non-linear decision boundaries.

Try playing around with the hyperparameters to get a feel for what happens if you set the learning rate too big (or too small), or if you don't give the network enough capacity (i.e. hidden layers and width).

In [25]:
```
reload(graph_test)
X_train, y_train, X_test, y_test = graph_test.generate_non_linear_data(1000, 1
0)
plt.scatter(X_test[:,0], X_test[:,1], c=y_test, cmap='bwr')
```

Out[25]:
```
<matplotlib.collections.PathCollection at 0x1a204cf390>
```

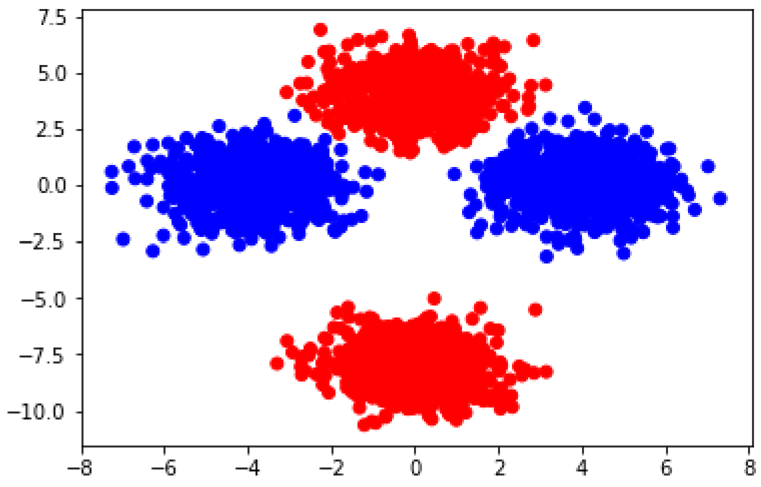


In [26]:
```
hidden_layers = [10, 10]
batch_size = 50
epochs = 2000
learning_rate = 0.001
tf.reset_default_graph()
predictions = graph.train_nn(X_train, y_train, X_test, hidden_layers, batch_si
ze, epochs, learning_rate)
```

```
Initial loss: 0.834448
Step:  20 Loss: 0.737587
Step:  6020 Loss: 0.0363514
Step:  12020 Loss: 0.0213854
Step:  18020 Loss: 0.0172886
Step:  24020 Loss: 0.0150452
Step:  30020 Loss: 0.0135012
Step:  36020 Loss: 0.0122878
```

In [27]:
```
plt.scatter(X_test[:,0], X_test[:,1], c=predictions, cmap='bwr')
```

Out[27]:
```
<matplotlib.collections.PathCollection at 0x1a2052e438>
```

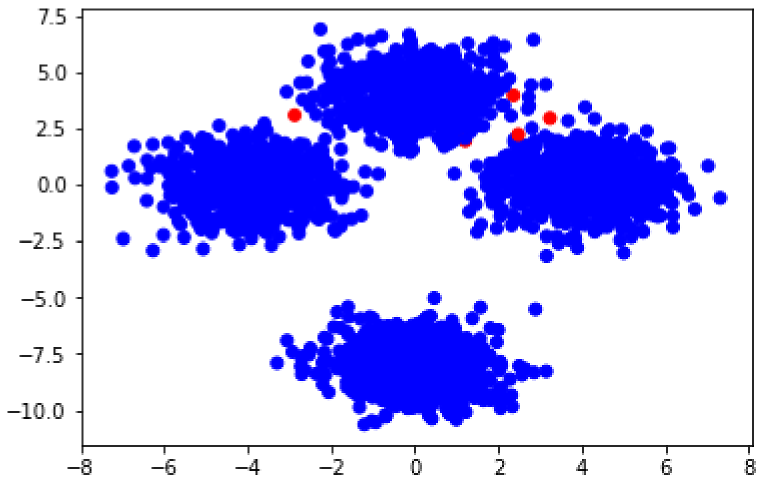


That looks pretty good!

Let's compare the predictions vs. the labels and see what we got wrong...

In [28]:
```
plt.scatter(X_test[:,0], X_test[:,1], c=(predictions!=y_test), cmap='bwr')
print("Accuracy: {:.02f}%".format(100*sum(predictions == y_test)/len(predictio
ns)))
```

```
Accuracy: 99.80%
```



Only a tiny number of errors (hopefully!). Good work!