Building a Neural Network from Scratch: Part 1

Mar 5, 2018

Machine Learning (http://jonathanweisberg.org/tags/machine-learning) • Neural Networks (http://jonathanweisberg.org/tags/neural-networks) • Python (http://jonathanweisberg.org/tags/python)

In this post we're going to build a neural network from scratch. We'll train it to recognize hand-written digits, using the famous MNIST data set.

We'll use just basic Python with NumPy to build our network (no high-level stuff like Keras or TensorFlow). We will dip into scikit-learn, but only to get the MNIST data and to assess our model once its built.

We'll start with the simplest possible "network": a single node that recognizes just the digit 0. This is actually just an implementation of logistic regression, which may seem kind of silly. But it'll help us get some key components working before things get more complicated.

Then we'll extend that into a network with one hidden layer, still recognizing just 0. Then we'll add a softmax for recognizing all the digits 0 through 9. That'll give us a 92% accurate digit-recognizer, bringing us up to the cutting edge of 1985 technology.

In a followup post we'll bring that up into the high nineties by making sundry improvements: better optimization, more hidden layers, and smarter initialization.

1. Hello, MNIST

MNIST (https://en.wikipedia.org/wiki/MNIST_database) contains 70,000 images of hand-written digits, each 28 x 28 pixels, in greyscale with pixel-values from 0 to 255. We could download (http://yann.lecun.com/exdb/mnist/) and preprocess the data ourselves. But the makers of scikit-learn already did that for us. Since it would be rude to neglect their efforts, we'll just import it:

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X, y = mnist["data"], mnist["target"]
```

We'll normalize the data to keep our gradients manageable:

```
X = X / 255
```

The default MNIST labels record `7` for an image of a seven, `4` for an image of a four, etc. But we're just building a zero-classifier for now. So we want our labels to say `1` when we have a zero, and `0` otherwise (intuitive, I know). So we'll overwrite the labels to make that happen:

```
import numpy as np

y_new = np.zeros(y.shape)
y_new[np.where(y == 0.0)[0]] = 1
y = y_new
```

Now we can make our train/test split. The MNIST images are pre-arranged so that the first 60,000 can be used for training, and the last 10,000 for testing. We'll also transform the data into the shape we want, with each example in a column (instead of a row):

```
m = 60000
m_test = X.shape[0] - m

X_train, X_test = X[:m].T, X[m:].T
y_train, y_test = y[:m].reshape(1,m), y[m:].reshape(1,m_test)
```

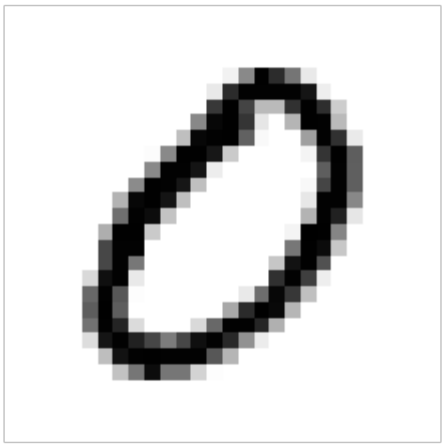
Finally we'll shuffle the training set for good measure:

```
np.random.seed(138)
shuffle_index = np.random.permutation(m)
X_train, y_train = X_train[:,shuffle_index], y_train[:,shuffle_index]
```

Let's have a look at a random image and label just to make sure we didn't throw anything out of wack:

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

i = 3
plt.imshow(X_train[:,i].reshape(28,28), cmap = matplotlib.cm.binary)
plt.axis("off")
plt.show()
print(y_train[:,i])
```



```
[1.]
```

That's a zero, so we want the label to be `1`, which it is. Looks good, so let's build our first network.

2. A Single Neuron (aka Logistic Regression)

We want to build a simple, feed-forward network with 784 inputs (=28 x 28), and a single sigmoid unit generating the output.

2.1 Forward Propogation

The forward pass on a single example x executes the following computation:

$$\hat{y} = \sigma(w^T x + b).$$

Here σ is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

So let's define:

```
def sigmoid(z):
    s = 1 / (1 + np.exp(-z))
    return s
```

We'll vectorize by stacking examples side-by-side, so that our input matrix X has an example in each column. The vectorized form of the forward pass is then:

$$\hat{y} = \sigma(w^T X + b).$$

Note that $\hat{y}$ is now a vector, not a scalar as it was in the previous equation.

In our code we'll compute this in two stages: `Z = np.matmul(W.T, X) + b` and then `A = sigmoid(Z)`. (`A` for Activation.) Breaking things up into stages like this is just for tidiness—it'll make our forward propagation computations mirror the steps in our backward propagation computations.

2.2 Cost Function

We'll use cross-entropy for our cost function. The formula for a single training example is:

$$L(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}).$$

Averaging over a training set of m examples we then have:

$$L(Y, \hat{Y}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right).$$

So let's define:

```
def compute_loss(Y, Y_hat):

    m = Y.shape[1]
    L = -(1./m) * ( np.sum( np.multiply(np.log(Y_hat),Y) ) + np.sum( np.

    return L
```

2.3 Backward Propagation

For backpropagation, we'll need to know how L changes with respect to each component w_j of w. That is, we must compute each $\partial L / \partial w_j$.

Focusing on a single example will make it easier to derive the formulas we need. Holding all values except w_j fixed, we can think of L as being computed in three steps: $w_j \longrightarrow z \longrightarrow \hat{y} \longrightarrow L$. The formulas for these steps are:

$$\begin{aligned} z &= w^T x + b, \\ \hat{y} &= \sigma(z), \\ L(y, \hat{y}) &= -y \log(\hat{y}) - (1-y)\log(1-\hat{y}). \end{aligned}$$

And the chain rule tells us:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j}.$$

Looking at $\partial L / \partial \hat{y}$ first:

$$\begin{aligned} \frac{\partial L}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} \left(-y \log(\hat{y}) - (1-y)\log(1-\hat{y}) \right) \\ &= -y \frac{\partial}{\partial \hat{y}} \log(\hat{y}) - (1-y) \frac{\partial}{\partial \hat{y}} \log(1-\hat{y}) \\ &= \frac{-y}{\hat{y}} + \frac{(1-y)}{1-\hat{y}} \\ &= \frac{\hat{y} - y}{\hat{y}(1-\hat{y})}. \end{aligned}$$

Next we want $\partial \hat{y} / \partial z$:

$$\begin{aligned} \frac{\partial}{\partial z} \sigma(z) &= \frac{\partial}{\partial z} \left(\frac{1}{1+e^{-z}} \right) \\ &= -\frac{1}{(1+e^{-z})^2} \frac{\partial}{\partial z} (1+e^{-z}) \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \frac{e^{-z}}{1+e^{-z}} \\ &= \sigma(z) \frac{e^{-z}}{1+e^{-z}} \\ &= \sigma(z) \left(1 - \frac{1}{1+e^{-z}} \right) \\ &= \sigma(z) \left(1 - \sigma(z)\right) \\ &= \hat{y}(1-\hat{y}). \end{aligned}$$

Lastly we tackle $\partial z/\partial w_j$:

$$\begin{aligned}\frac{\partial}{\partial w_j}(w^T x + b) &= \frac{\partial}{\partial w_j}(w_0 x_0 + \ldots + w_n x_n + b)\\ &= w_j.\end{aligned}$$

Finally we can substitute into the chain rule to find:

$$\begin{aligned}\frac{\partial L}{\partial w_j} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_j}\\ &= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\hat{y}(1-\hat{y})w_j\\ &= (\hat{y}-y)w_j.\end{aligned}$$

In vectorized form with m training examples this gives us:

$$\frac{\partial L}{\partial w} = \frac{1}{m}X(\hat{y}-y)^T.$$

What about $\partial L/\partial b$? A very similar derivation yields, for a single example:

$$\frac{\partial L}{\partial b} = (\hat{y}-y).$$

Which in vectorized form amounts to:

$$\frac{\partial L}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}).$$

In our code we'll label these gradients according to their denominators, as `dW` and `db`. So for backpropagation we'll compute `dW = (1/m) * np.matmul(X, (A-Y).T)` and `db = (1/m) * np.sum(A-Y, axis=1, keepdims=True)`.

2.4 Build & Train

Ok we're ready to build and train our network!

```
learning_rate = 1

X = X_train
Y = y_train

n_x = X.shape[0]
m = X.shape[1]

W = np.random.randn(n_x, 1) * 0.01
b = np.zeros((1, 1))

for i in range(2000):
    Z = np.matmul(W.T, X) + b
    A = sigmoid(Z)

    cost = compute_loss(Y, A)

    dW = (1/m) * np.matmul(X, (A-Y).T)
    db = (1/m) * np.sum(A-Y, axis=1, keepdims=True)

    W = W - learning_rate * dW
    b = b - learning_rate * db

    if (i % 100 == 0):
        print("Epoch", i, "cost: ", cost)

print("Final cost:", cost)
```

```
Epoch 0 cost:  0.6840801595436431
Epoch 100 cost:  0.041305162058342754
... *snip* ...
Final cost: 0.02514156608481825
```

We could probably eek out a bit more accuracy with some more training. But the gains have slowed considerably. So let's just see how we did, by looking at the confusion matrix:

```
from sklearn.metrics import classification_report, confusion_matrix

Z = np.matmul(W.T, X_test) + b
A = sigmoid(Z)

predictions = (A>.5)[0,:]
labels = (y_test == 1)[0,:]

print(confusion_matrix(predictions, labels))
```

```
[[8980   33]
 [  40  947]]
```

Hey, that's actually pretty good! We got 947 of the zeros and missed only 33, while getting nearly all the negative cases right. In terms of f1-score that's 0.99:

```
print(classification_report(predictions, labels))
```

```
             precision    recall  f1-score   support

      False       1.00      1.00      1.00      9013
       True       0.97      0.96      0.96       987

avg / total       0.99      0.99      0.99     10000
```

So, now that we've got a working model and optimization algorithm, let's enrich it.

3. One Hidden Layer

Let's add a hidden layer now, with 64 units (a mostly arbitrary choice). I won't go through the derivations of all the formulas for the forward and backward passes this time; they're a pretty direct extension of the work we did earlier. Instead let's just dive right in and build the model:

```
X = X_train
Y = y_train

n_x = X.shape[0]
n_h = 64
learning_rate = 1

W1 = np.random.randn(n_h, n_x)
b1 = np.zeros((n_h, 1))
W2 = np.random.randn(1, n_h)
b2 = np.zeros((1, 1))

for i in range(2000):

    Z1 = np.matmul(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(W2, A1) + b2
    A2 = sigmoid(Z2)

    cost = compute_loss(Y, A2)

    dZ2 = A2-Y
    dW2 = (1./m) * np.matmul(dZ2, A1.T)
    db2 = (1./m) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(W2.T, dZ2)
    dZ1 = dA1 * sigmoid(Z1) * (1 - sigmoid(Z1))
    dW1 = (1./m) * np.matmul(dZ1, X.T)
    db1 = (1./m) * np.sum(dZ1, axis=1, keepdims=True)

    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1

    if i % 100 == 0:
        print("Epoch", i, "cost: ", cost)

print("Final cost:", cost)
```

```
Epoch 0 cost:  0.9144384083567224
Epoch 100 cost:  0.08856953026938433
... *snip* ...
Final cost: 0.024249298861903648
```

How'd we do?

```
Z1 = np.matmul(W1, X_test) + b1
A1 = sigmoid(Z1)
Z2 = np.matmul(W2, A1) + b2
A2 = sigmoid(Z2)

predictions = (A2>.5)[0,:]
labels = (y_test == 1)[0,:]

print(confusion_matrix(predictions, labels))
print(classification_report(predictions, labels))
```

```
[[8984   36]
 [  36  944]]
             precision    recall  f1-score   support

      False       1.00      1.00      1.00      9020
       True       0.96      0.96      0.96       980

avg / total       0.99      0.99      0.99     10000
```

Hmm, not bad, but about the same as our one-neuron model did. We could do more training and add more nodes/layers. But it'll be slow going until we improve our optimization algorithm, which we'll do in a followup post.

So for now let's turn to recognizing all ten digits.

4. Upgrading to Multiclass

4.1 Labels

First we need to redo our labels. We'll re-import everything, so that we don't have to go back and coordinate with our earlier shuffling:

```
mnist = fetch_mldata('MNIST original')
X, y = mnist["data"], mnist["target"]

X = X / 255
```

Then we'll one-hot encode MNIST's labels, to get a 10 x 70,000 array.

```
digits = 10
examples = y.shape[0]

y = y.reshape(1, examples)

Y_new = np.eye(digits)[y.astype('int32')]
Y_new = Y_new.T.reshape(digits, examples)
```

Then we re-split, re-shape, and re-shuffle our training set:

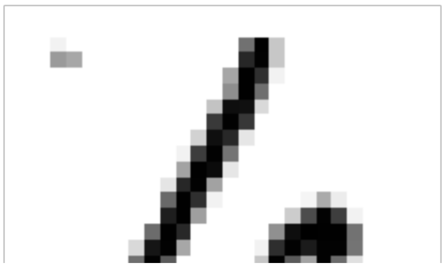
```
m = 60000
m_test = X.shape[0] - m

X_train, X_test = X[:m].T, X[m:].T
Y_train, Y_test = Y_new[:,:m], Y_new[:,m:]

shuffle_index = np.random.permutation(m)
X_train, Y_train = X_train[:, shuffle_index], Y_train[:, shuffle_index]
```

A quick check that things are as they should be:

```
i = 12
plt.imshow(X_train[:,i].reshape(28,28), cmap = matplotlib.cm.binary)
plt.axis("off")
plt.show()
Y_train[:,i]
```



```
array([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.])
```

Looks good, so let's consider what changes we need to make to the model itself.

4.2 Forward Propagation

Only the last layer of our network is changing. To add the softmax, we have to replace our lone, final node with a 10-unit layer. Its final activations are the exponentials of its z-values, normalized across all ten such exponentials. So instead of just computing $\sigma(z)$, we compute the activation for each unit i:

$$\frac{e^{z_i}}{\sum_{j=0}^{9} e^{z_j}}.$$

So, in our vectorized code, the last line of forward propagation will be `A2 = np.exp(Z2) / np.sum(np.exp(Z2), axis=0)`.

4.3 Cost Function

Our cost function now has to generalize to more than two classes. The general formula for n classes is:

$$L(y, \hat{y}) = -\sum_{i=0}^{n} y_i \log(\hat{y}_i).$$

Averaging over m training examples this becomes:

$$L(Y, \hat{Y}) = -\frac{1}{m} \sum_{j=0}^{m} \sum_{i=0}^{n} y_i^{(j)} \log(\hat{y}_i^{(j)}).$$

So let's define:

```
def compute_multiclass_loss(Y, Y_hat):

    L_sum = np.sum(np.multiply(Y, np.log(Y_hat)))
    m = Y.shape[1]
    L = -(1/m) * L_sum

    return L
```

4.4 Backprop

Luckily it turns out that backprop isn't really affected by the switch to a softmax. A softmax generalizes the sigmoid activiation we've been using, and in such a way that the code we wrote earlier still works. We could verify this by deriving:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i.$$

But I won't walk through the steps here. Let's just go ahead and build our final network.

4.5 Build & Train

```
n_x = X_train.shape[0]
n_h = 64
learning_rate = 1

W1 = np.random.randn(n_h, n_x)
b1 = np.zeros((n_h, 1))
W2 = np.random.randn(digits, n_h)
b2 = np.zeros((digits, 1))

X = X_train
Y = Y_train

for i in range(2000):

    Z1 = np.matmul(W1,X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(W2,A1) + b2
    A2 = np.exp(Z2) / np.sum(np.exp(Z2), axis=0)

    cost = compute_multiclass_loss(Y, A2)

    dZ2 = A2-Y
    dW2 = (1./m) * np.matmul(dZ2, A1.T)
    db2 = (1./m) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(W2.T, dZ2)
    dZ1 = dA1 * sigmoid(Z1) * (1 - sigmoid(Z1))
    dW1 = (1./m) * np.matmul(dZ1, X.T)
    db1 = (1./m) * np.sum(dZ1, axis=1, keepdims=True)

    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1

    if (i % 100 == 0):
        print("Epoch", i, "cost: ", cost)

print("Final cost:", cost)
```

```
Epoch 0 cost:  9.243960401572568
... *snip* ...
Epoch 1900 cost:  0.24585173887243117
Final cost: 0.24072776877870128
```

Let’s see how we did:

```
Z1 = np.matmul(W1, X_test) + b1
A1 = sigmoid(Z1)
Z2 = np.matmul(W2, A1) + b2
A2 = np.exp(Z2) / np.sum(np.exp(Z2), axis=0)

predictions = np.argmax(A2, axis=0)
labels = np.argmax(Y_test, axis=0)

print(confusion_matrix(predictions, labels))
print(classification_report(predictions, labels))
```

```
[[ 946    0   14    3    3   10   12    2    9    4]
 [   0 1112    3    2    1    1    2    8    3    4]
 [   3    4  937   24   10    7    8   18    8    3]
 [   4    2   17  924    1   39    4   13   26    9]
 [   0    1   10    0  905    9   11    9   10   40]
 [  12    5    2   26    3  786   15    3   24   14]
 [   8    1   19    2    9   10  902    1    9    1]
 [   2    1   13   14    3    5    1  946    9   25]
 [   5    9   16   11    5   18    3    5  868    9]
 [   0    0    1    4   42    7    0   23    8  900]]
             precision    recall  f1-score   support

          0       0.97      0.94      0.95      1003
          1       0.98      0.98      0.98      1136
          2       0.91      0.92      0.91      1022
          3       0.91      0.89      0.90      1039
          4       0.92      0.91      0.92       995
          5       0.88      0.88      0.88       890
          6       0.94      0.94      0.94       962
          7       0.92      0.93      0.92      1019
          8       0.89      0.91      0.90       949
          9       0.89      0.91      0.90       985

avg / total       0.92      0.92      0.92     10000
```

We’re at 92% accuracy across all digits, not bad! And it looks like we could still improve with more training.

But let’s work on speeding up our optimization alogirthm first. We’ll pick things up there in the next post.