Implementing a Neural Network in Python

WildML

March 28, 2016

Outline

Outline

Python and IDEs

Install Python and pip

- Linux (Ubuntu):

```
$ sudo apt-get install python
$ sudo apt-get install python-pip
$ sudo pip install --upgrade pip
```

- Windows: Read this article.
- MacOS: Read this article.

- PyCharm
- WingIDE
- PyDev
- Vim and spf-13

Outline

numpy and scipy

http://www.scipy.org

- Download and install: Read this article.

scikit-learn

http://scikit-learn.org/stable/

- Download and install: Read this article.

Outline

Anaconda Platform
https://www.continuum.io/

- Download and install: Read this article.

Outline

Recap: Neural Networks

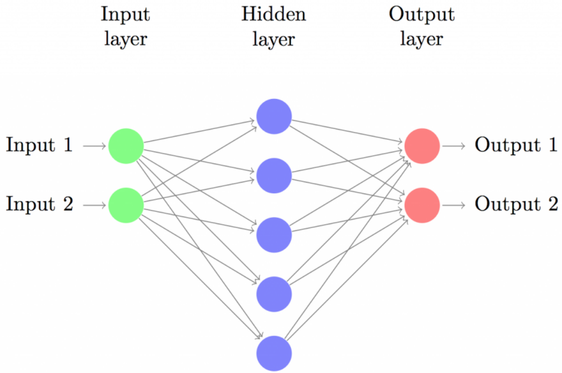

Figure : A 3-layer neural network with one input layer, one hidden layer, and one output layer.

Recap: Neural Networks
Cont.

- The number of nodes in the input layer is determined by the dimensionality of input data.
- The number of nodes in the output layer is determined by the number of classes.
- The number of nodes in the hidden layer is **hyper-parameter**, we can choose in the experiments.

Outline

How our network makes predictions

Our network makes predictions using forward propagation:

$$
\begin{aligned}
z_1 &= xW_1 + b_1 \\
a_1 &= tanh(z_1) \\
z_2 &= a_1 W_1 + b_2 \\
a_2 &= \hat{y} = softmax(z_2)
\end{aligned}
$$

where

- z_i is the input of layer i
- a_i is the output of layer i after applying the activation function
- W_1, b_1, W_2, b_2 are parameters of our network, which we need to learn from our training data

How our network makes predictions

Cont.

- Some activation functions often use:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$
$$relu(x) = max(0, x)$$

- *softmax* function is simply a way to convert raw scores to probabilities.

$$softmax(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

Outline

Learning the Parameters

- Learning the parameters for our network means finding parameters (W_1, b_1, W_2, b_2) that minimize the error on our training data.
- A common choice with the softmax output is the **categorical cross-entropy loss**

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log \hat{y}_{n,i}$$

Learning the Parameters
Cont.

- We can use **gradient descent** to find the minimum.
- Gradient descent needs the gradients (vector of derivatives) of the loss function with respect to our parameters: $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_2}$
- To calculate these gradients we use the famous **back-propagation algorithm**.

Learning the Parameters
Cont.

Applying the back-propagation formula we find the following:

$$\delta_3 = \hat{y} - y$$
$$\delta_2 = (1 - \tanh^2 z_1) \circ \delta_3 W_2^T$$
$$\frac{\partial L}{\partial W_2} = a_1^T \delta_3$$
$$\frac{\partial L}{\partial b_2} = \delta_3$$
$$\frac{\partial L}{\partial W_1} = x^T \delta 2$$
$$\frac{\partial L}{\partial b_1} = \delta 2$$

Outline

Implement

Generating a dataset

Example

`scikit-learn` has some useful dataset generators, so we don't need to write the code ourselves:

```python
import matplotlib.pyplot as plt
import numpy as np
import sklearn
import sklearn.datasets
import sklearn.linear_model
import matplotlib

# Generate a dataset and plot it
np.random.seed(0)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.
    Spectral)
```

Outline

Implement
Defining variables

We start by defining some useful variables and parameters for gradient descent:

```
num_examples = len(X) # training set size
nn_input_dim = 2 # input layer dimensionality
nn_output_dim = 2 # output layer dimensionality

# Gradient descent parameters (I picked these by hand)
epsilon = 0.01 # learning rate for gradient descent
reg_lambda = 0.01 # regularization strength
```

Outline

Implement

Loss function

First let's implement the loss function we defined above:

```python
def calculate_loss(model):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation to calculate our predictions
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # Calculating the loss
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)
    # Add regulatization term to loss (optional)
    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./num_examples * data_loss
```

Outline

Implement
Predict function

We also implement a helper function to calculate the output of the network

```
# Helper function to predict an output (0 or 1)
def predict(model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return np.argmax(probs, axis=1)
```

Outline

Implement
Batch gradient descent

Finally, here comes the function to train our Neural Network.

```
def build_model(nn_hdim, num_passes=20000, print_loss=False):
    # Initialize the parameters to random values. We need to learn these.
    np.random.seed(0)
    W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
    b1 = np.zeros((1, nn_hdim))
    W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
    b2 = np.zeros((1, nn_output_dim))
    # This is what we return at the end
    model = {}
```

Implement

Batch gradient descent (Cont.)

Finally, here comes the function to train our Neural Network.

```
# Gradient descent. For each batch...
for i in xrange(0, num_passes):
    # Forward propagation
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # Backpropagation
    delta3 = probs
    delta3[range(num_examples), y] -= 1
    dW2 = (a1.T).dot(delta3)
    db2 = np.sum(delta3, axis=0, keepdims=True)
    delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
    dW1 = np.dot(X.T, delta2)
    db1 = np.sum(delta2, axis=0)
```

Implement

Batch gradient descent (Cont.)

Finally, here comes the function to train our Neural Network.

```
# Add regularization terms (b1 and b2 don't have regularization terms)
dW2 += reg_lambda * W2
dW1 += reg_lambda * W1
# Gradient descent parameter update
W1 += -epsilon * dW1
b1 += -epsilon * db1
W2 += -epsilon * dW2
b2 += -epsilon * db2
```

Implement

Batch gradient descent (Cont.)

Finally, here comes the function to train our Neural Network.

```
        # Assign new parameters to the model
        model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}

        # Optionally print the loss.
        # This is expensive because it uses the whole dataset, so we don't want to do it too often.
        if print_loss and i % 1000 == 0:
          print "Loss after iteration %i: %f" %(i, calculate_loss(model))

return model
```

A network with a hidden layer of size 3

Let's see what happens if we train a network with a hidden layer size of 3.

```
# Build a model with a 3-dimensional hidden layer
model = build_model(3, print_loss=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(model, x))
plt.title("Decision Boundary for hidden layer size 3")
```

Varying the hidden layer size

Let's now get a sense of how varying the hidden layer size affects the result.

```
plt.figure(figsize=(16, 32))
hidden_layer_dimensions = [1, 2, 3, 4, 5, 20, 50]
for i, nn_hdim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer size %d' % nn_hdim)
    model = build_model(nn_hdim)
    plot_decision_boundary(lambda x: predict(model, x))
plt.show()
```