WILDML

Artificial Intelligence, Deep Learning, and NLP

SEPTEMBER 3, 2015 BY DENNY BRITZ

Get the code: To follow along, all the code is also available as an iPython notebook on Github.

In this post we will implement a simple 3-layer neural network from scratch. We won't derive all the math that's required, but I will try to give an intuitive explanation of what we are doing. I will also point to resources for you read up on the details.

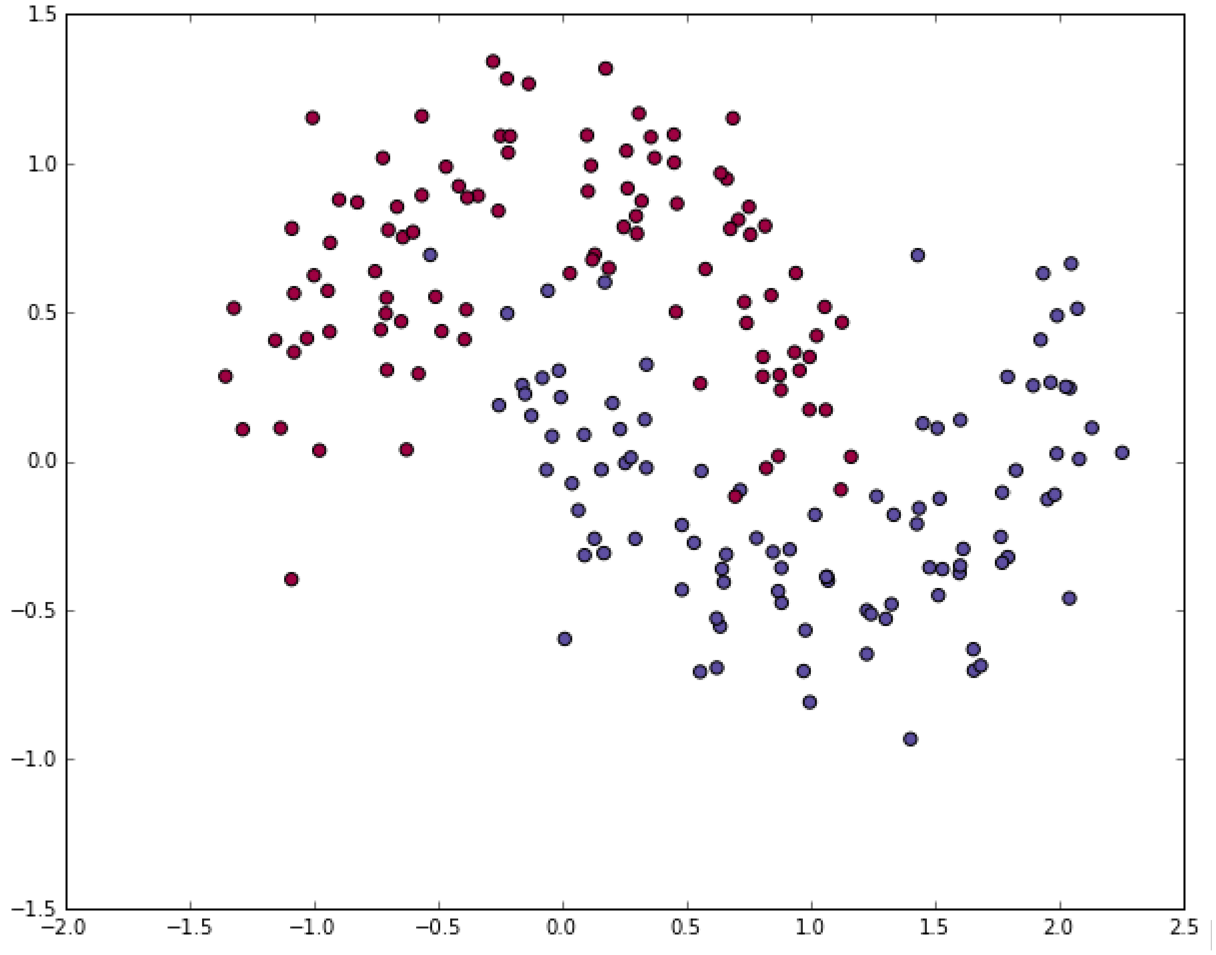
Here I'm assuming that you are familiar with basic Calculus and Machine Learning concepts, e.g. you know what classification and regularization is. Ideally you also know a bit about how optimization techniques like gradient descent work. But even if you're not familiar with any of the above this post could still turn out to be interesting ;)

But why implement a Neural Network from scratch at all? Even if you plan on using Neural Network libraries like PyBrain in the future, implementing a network from scratch at least once is an extremely valuable exercise. It helps you gain an understanding of how neural networks work, and that is essential for designing effective models.

One thing to note is that the code examples here aren't terribly efficient. They are meant to be easy to understand. In an upcoming post I will explore how to write an efficient Neural Network implementation using Theano. (Update: now available)

Let's start by generating a dataset we can play with. Fortunately, scikit-learn has some useful dataset generators, so we don't need to write the code ourselves. We will go with the make_moons function.

```
# 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)
```



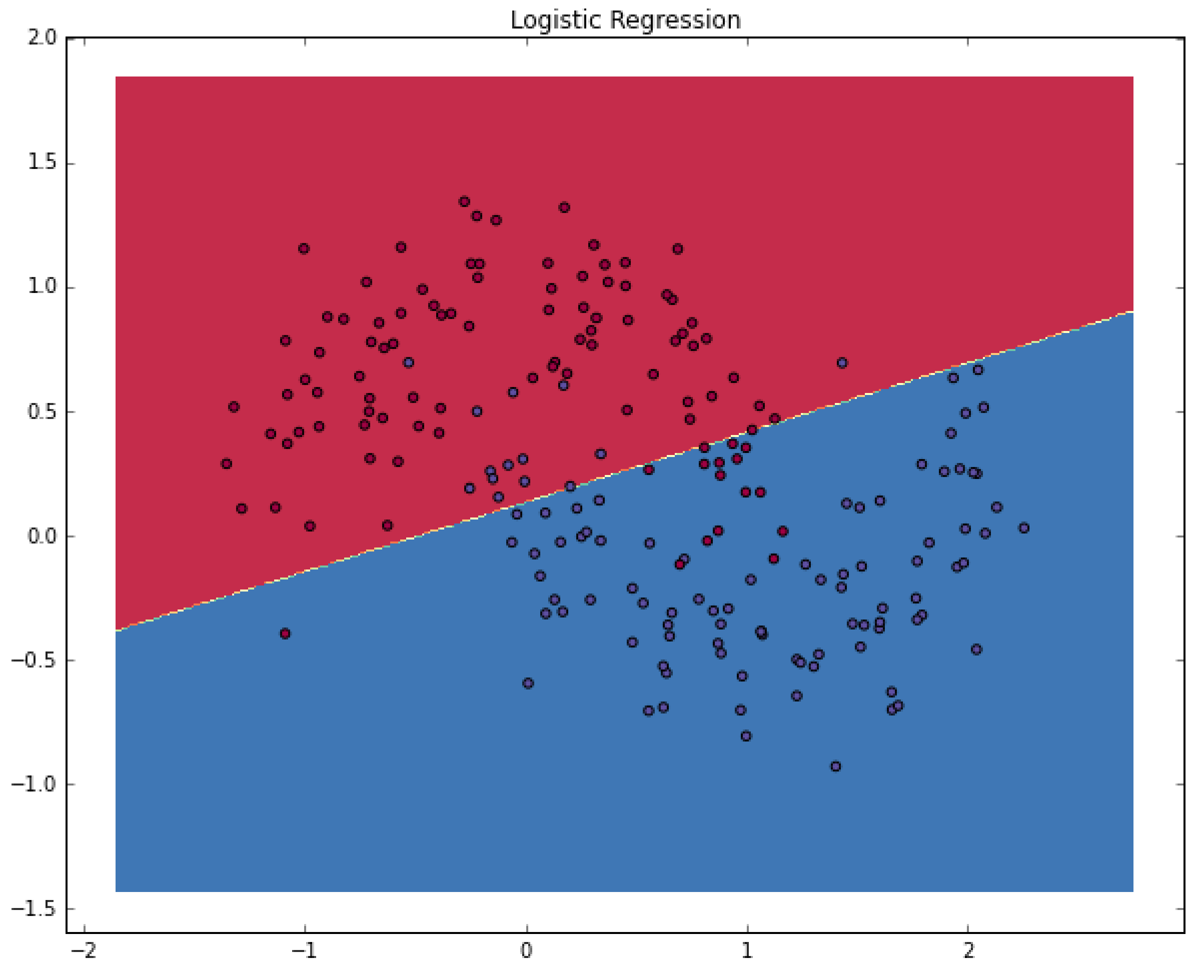
The dataset we generated has two classes, plotted as red and blue points. You can think of the blue dots as male patients and the red dots as female patients, with the x- and y- axis being medical measurements.

Our goal is to train a Machine Learning classifier that predicts the correct class (male of female) given the x- and y- coordinates. Note that the data is not linearly separable, we can't draw a straight line that separates the two classes. This means that linear classifiers, such as Logistic Regression, won't be able to fit the data unless you hand-engineer non-linear features (such as polynomials) that work well for the given dataset.

In fact, that's one of the major advantages of Neural Networks. You don't need to worry about feature engineering. The hidden layer of a neural network will learn features for you.

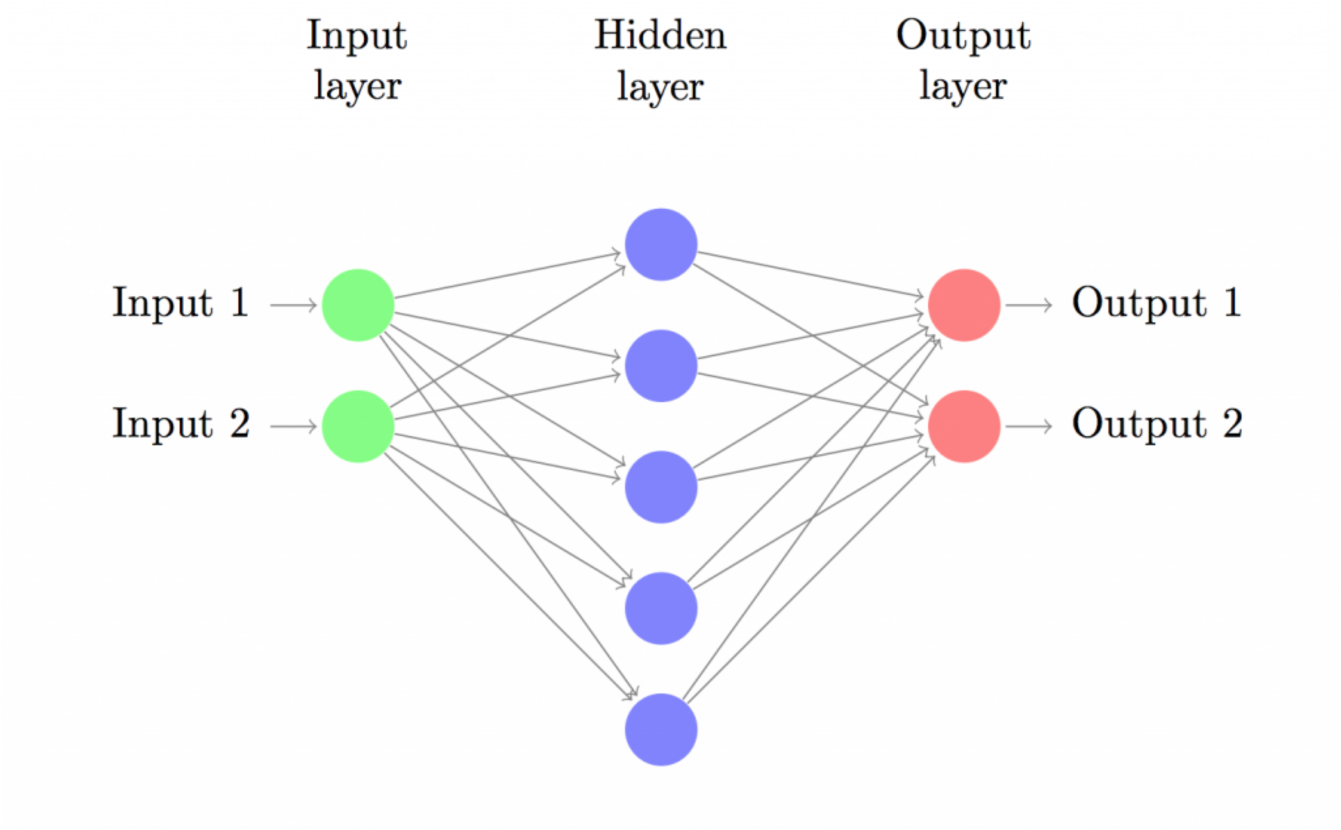
To demonstrate the point let's train a Logistic Regression classifier. It's input will be the x- and y-values and the output the predicted class (0 or 1). To make our life easy we use the Logistic Regression class from .

```
# Train the logistic rgeression classifier
clf = sklearn.linear_model.LogisticRegressionCV()
clf.fit(X, y)

# Plot the decision boundary
plot_decision_boundary(lambda x: clf.predict(x))
plt.title("Logistic Regression")
```



The graph shows the decision boundary learned by our Logistic Regression classifier. It separates the data as good as it can using a straight line, but it's unable to capture the "moon shape" of our data.

Let's now build a 3-layer neural network with one input layer, one hidden layer, and one output layer. The number of nodes in the input layer is determined by the dimensionality of our data, 2. Similarly, the number of nodes in the output layer is determined by the number of classes we have, also 2. (Because we only have 2 classes we could actually get away with only one output node predicting 0 or 1, but having 2 makes it easier to extend the network to more classes later on). The input to the network will be x- and y- coordinates and its output will be two probabilities, one for class 0 ("female") and one for class 1 ("male"). It looks something like this:



We can choose the dimensionality (the number of nodes) of the hidden layer. The more nodes we put into the hidden layer the more complex functions we will be able fit. But higher dimensionality comes at a cost. First, more computation is required to make predictions and learn the network parameters. A bigger number of parameters also means we become more prone to overfitting our data.

How to choose the size of the hidden layer? While there are some general guidelines and recommendations, it always depends on your specific problem and is more of an art than a science. We will play with the number of nodes in the hidden later later on and see how it affects our output.

We also need to pick an for our hidden layer. The activation function transforms the inputs of the layer into its outputs. A nonlinear activation function is what allows us to fit nonlinear hypotheses. Common chocies for activation functions are tanh, the sigmoid function, or ReLUs. We will use , which performs quite well in many scenarios. A nice property of these functions is that their derivate can be computed using the original function value. For example, the derivative of $\tanh x$ is $1 - \tanh^2 x$. This is useful because it allows us to compute $\tanh x$ once and re-use its value later on to get the derivative.

Because we want our network to output probabilities the activation function for the output layer will be the softmax, which is simply a way to convert raw scores to probabilities. If you're familiar with the logistic function you can think of softmax as its generalization to multiple classes.

How our network makes predictions

Our network makes predictions using forward propagation, which is just a bunch of matrix multiplications and the application of the activation function(s) we defined above. If x is the 2-dimensional input to our network then we calculate our prediction $\hat{y}$ (also two-dimensional) as follows:

$$
\begin{aligned}
z_1 &= xW_1 + b_1 \\
a_1 &= \tanh(z_1) \\
z_2 &= a_1 W_2 + b_2 \\
a_2 &= \hat{y} = \mathrm{softmax}(z_2)
\end{aligned}
$$

z_i is the input of layer i and 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. You can think of them as matrices transforming data between layers of the network.

Looking at the matrix multiplications above we can figure out the dimensionality of these matrices. If we use 500 nodes for our hidden layer then $W_1 \in \mathbb{R}^{2\times500}$, $b_1 \in \mathbb{R}^{500}$, $W_2 \in \mathbb{R}^{500\times2}$, $b_2 \in \mathbb{R}^2$. Now you see why we have more parameters if we increase the size of the hidden layer.

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. But how do we define the error? We call the function that measures our error the . A common choice with the softmax output is the categorical cross-entropy loss (also known as negative log likelihood). If we have N training examples and C classes then the loss for our prediction $\hat{y}$ with respect to the true labels y is given by:

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

The formula looks complicated, but all it really does is sum over our training examples and add to the loss if we predicted the incorrect class. The further away the two probability distributions y (the correct labels) and $\hat{y}$ (our predictions) are, the greater our loss will be. By finding parameters that minimize the loss we maximize the likelihood of our training data.

We can use gradient descent to find the minimum and I will implement the most vanilla version of gradient descent, also called batch gradient descent with a fixed learning rate. Variations such as SGD (stochastic gradient descent) or minibatch gradient descent typically perform better in practice. So if you are serious you'll want to use one of these, and ideally you would also decay the learning rate over time.

As an input, 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 , which is a way to efficiently calculate the gradients starting from the output. I won't go into detail how backpropagation works, but there are many excellent explanations (here or here) floating around the web.

Applying the backpropagation formula we find the following (trust me on this):

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

Implementation

Now we are ready for our implementation. 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
```

First let's implement the loss function we defined above. We use this to evaluate how well our model is doing:

```
# Helper function to evaluate the total loss on the dataset
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.squ
    return 1./num_examples * data_loss
```

We also implement a helper function to calculate the output of the network. It does forward propagation as defined above and returns the class with the highest probability.

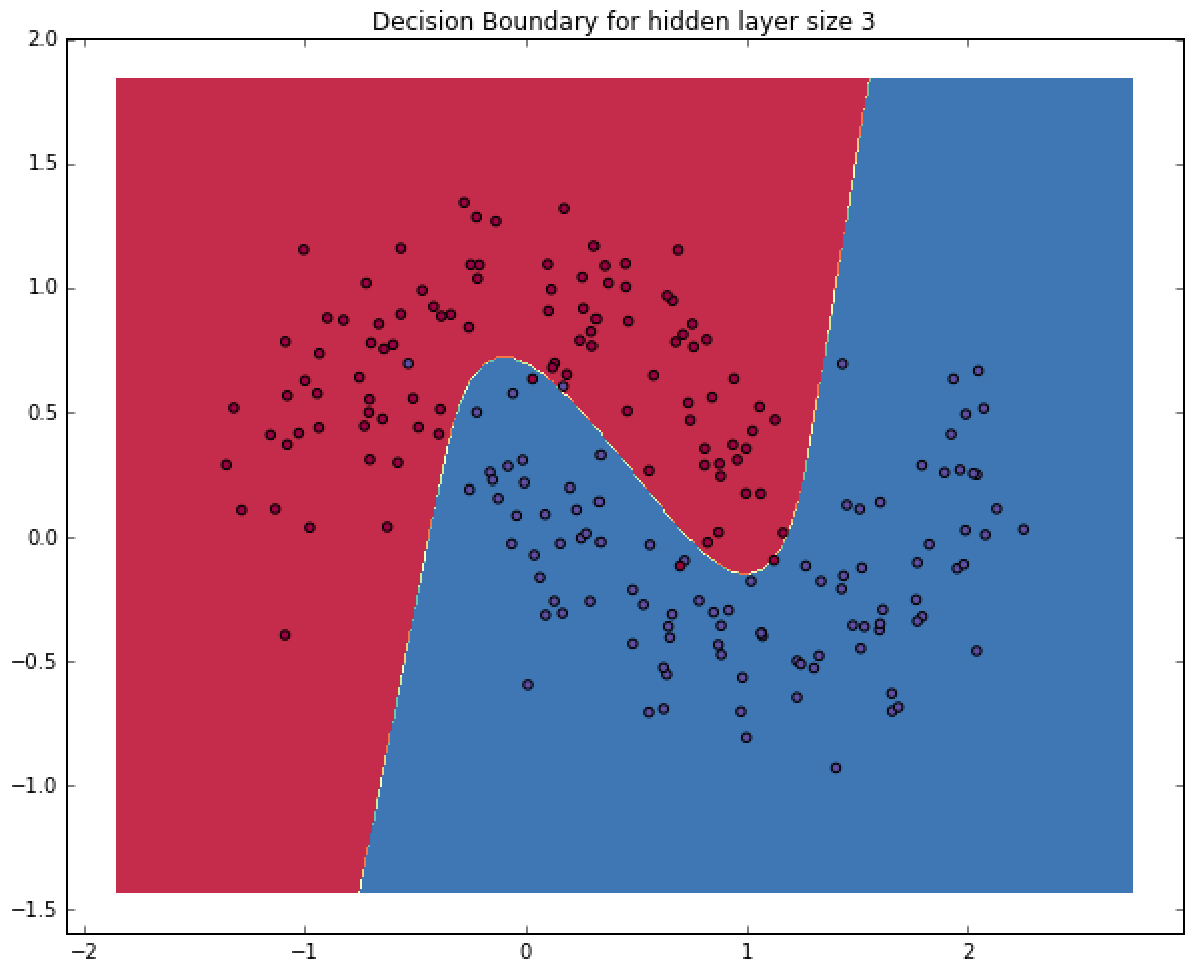
```
# Helper function to predict an output (0 or 1)
def predict(model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2'
    # Forward propagation
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores,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

Finally, here comes the function to train our Neural Network. It implements batch gradient descent using the backpropagation derivates we found above.

```
# This function learns parameters for the neural network and returns t
# - nn_hdim: Number of nodes in the hidden layer
# - num_passes: Number of passes through the training data for gradier
# - print_loss: If True, print the loss every 1000 iterations
def build_model(nn_hdim, num_passes=20000, print_loss=False):

    # Initialize the parameters to random values. We need to learn the
    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 = {}

    # 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)

        # Add regularization terms (b1 and b2 don't have regularizatic
        dW2 += reg_lambda * W2
        dW1 += reg_lambda * W1

        # Gradient descent parameter update
        W1 += -epsilon * dW1
        b1 += -epsilon * db1
        W2 += -epsilon * dW2
        b2 += -epsilon * db2
        # 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 c
        if print_loss and i % 1000 == 0:
          print "Loss after iteration %i: %f" %(i, calculate_loss(mode

    return model
```

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

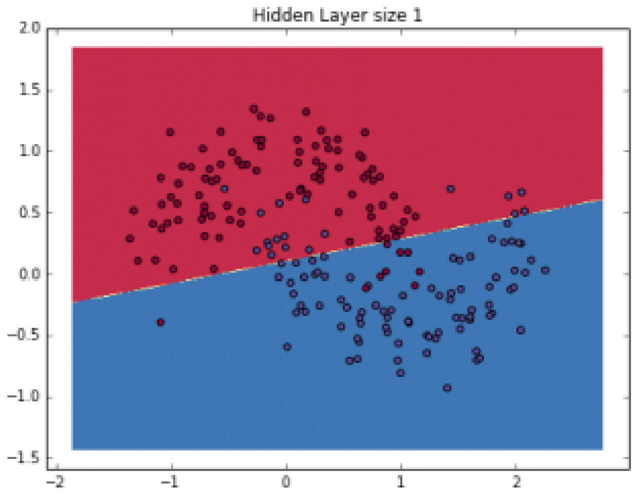
```
# Build a model with a 3-dimensional hidden layer
model = build_model(3, print_loss=True)

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

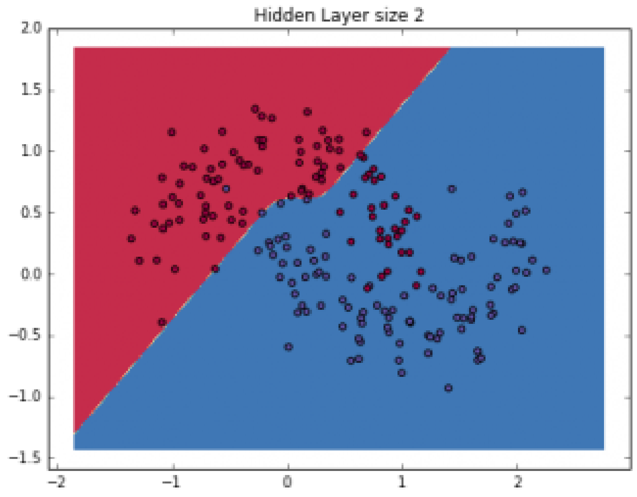


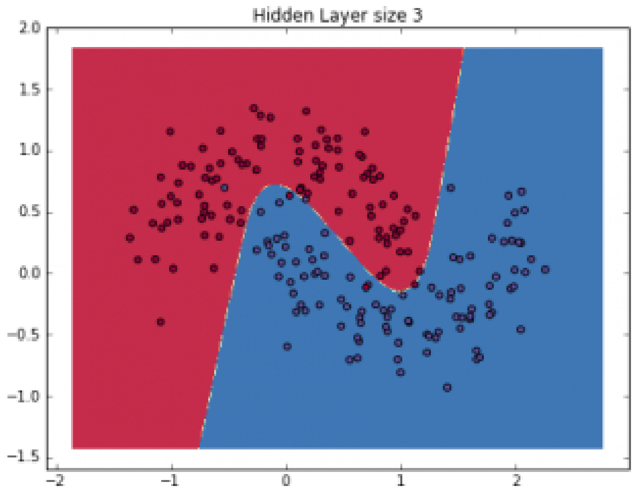
Yay! This looks pretty good. Our neural networks was able to find a decision boundary that successfully separates the classes.

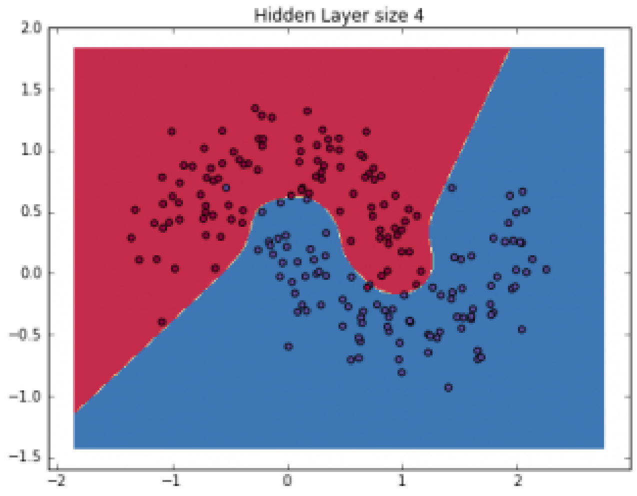
In the example above we picked a hidden layer size of 3. Let's now get a sense of how varying the hidden layer size affects the result.

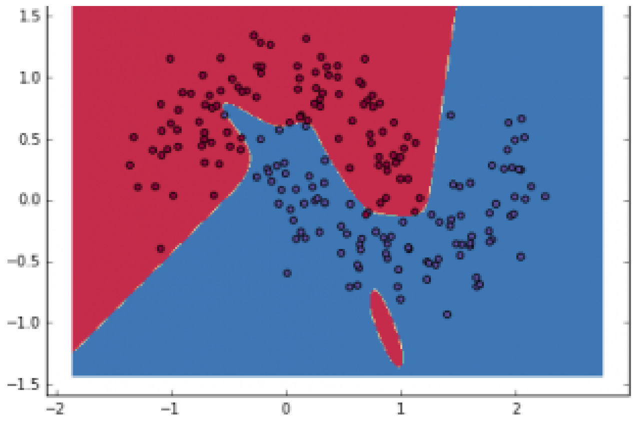
```
plt.figure(figsize=(16, 32))
hidden_layer_dimensions = [1, 2, 3, 4, 5, 20, 50]
for i, nn_hdim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer size %d' % nn_hdim)
    model = build_model(nn_hdim)
    plot_decision_boundary(lambda x: predict(model, x))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

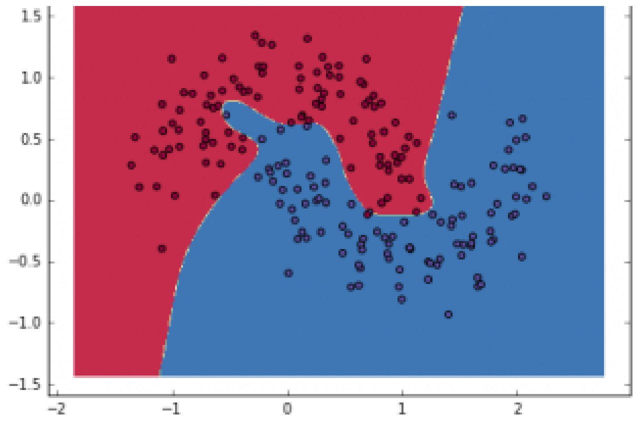


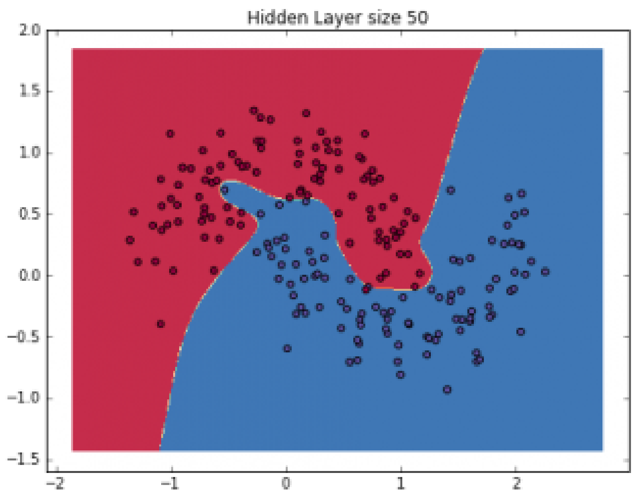












We can see that a hidden layer of low dimensionality nicely captures the general trend of our data. Higher dimensionalities are prone to overfitting. They are "memorizing" the data as opposed to fitting the general shape. If we were to evaluate our model on a separate test set (and you should!) the model with a smaller hidden layer size would likely perform better due to better generalization. We could counteract overfitting with stronger regularization, but picking the a correct size for hidden layer is a much more "economical" solution.

Here are some things you can try to become more familiar with the code:

1. Instead of batch gradient descent, use minibatch gradient descent (more info) to train the network. Minibatch gradient descent typically performs better in practice.
2. We used a fixed learning rate ϵ for gradient descent. Implement an annealing schedule for the gradient descent learning rate (more info).

3. We used a $\tanh$ activation function for our hidden layer. Experiment with other activation functions (some are mentioned above). Note that changing the activation function also means changing the backpropagation derivative.
4. Extend the network from two to three classes. You will need to generate an appropriate dataset for this.
5. Extend the network to four layers. Experiment with the layer size. Adding another hidden layer means you will need to adjust both the forward propagation as well as the backpropagation code.

All of the code is available as an iPython notebook on Github. Please leave questions or feedback in the comments!

DEEP LEARNING, NEURAL NETWORKS

129 Comments WildML 1 Login

Recommend 31 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Manuel Korfmann • 3 years ago
A pleasure to read, thanks for this piece!
19 • Reply • Share ›

Denny Britz Mod → Manuel Korfmann • 3 years ago

I'm happy if it was helpful!

1 ^ | v • Reply • Share ›

Emil Magerramov ➦ Denny Britz • a year ago

Hi, your article looks really nice.
Can I translate it to Russian and use as a reference on one of the classes I conduct?

^ | v • Reply • Share ›

electrocured • 2 years ago

Hi, great guide, thanks a lot.

For the not initiated, i would like to add the next lines. Took me a while to figure out the imports.

#required imports
import numpy as np
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt

# to show the plot at the end ;D
plt.show()

again thanks!

6 ^ | v • Reply • Share ›

Alex Lementuev • a year ago

Thumbs up if you came here from Coursera! :D

5 ^ | v • Reply • Share ›

u777 ➦ Alex Lementuev • a year ago

Here we go again!

^ | v • Reply • Share ›

Zhu Xiaohu • 3 years ago

Hi, thanks for this gentle introduction to NN. I find a small typo in formula for $$\delta_2$$. Should it be $$\delta = (1-\tanh^2 z_1) $$?

2 ^ | v • Reply • Share ›

Denny Britz Mod ➦ Zhu Xiaohu • 3 years ago

Yes, you're right! Thanks for catching that! Really appreciate it.

1 ^ | v • Reply • Share ›

Samuel Bedard ➦ Denny Britz • 2 years ago

its also in the code : delta2 = delta3.dot(W2.T) *

its also in the code : delta2 = delta3.dot(W2.T)
(1 - np.power(a1, 2)) should be :
delta2 = delta3.dot(W2.T) * (1 - np.power(z1, 2))
Reply • Share ›

E Unum Pluribus ☆ capitalist → Denny Britz
• 3 years ago
if a1 = tanh(z1), then does a1^2 = tanh^2(z1)?
(and 1-a1^2 = 1 - tanh^2(z1))?
Reply • Share ›

Zhu Xiaohu → Denny Britz • 3 years ago
You're so kind. :) Look forward for your new great articles!
Reply • Share ›

Michael A. Alcorn • 3 years ago
Found my way here from /r/MachineLearning... really good stuff! I always think it's helpful to take the time to really step through these things.

Just one comment, I think your description of the cross-entropy function is a little off. The cross-entropy function, when used with multi-class classification, is really just a different way of calculating the likelihood of the model. The typical likelihood function is calculated by multiplying together p(y | model) for all y, while the log-likelihood is calculated by adding log(p(y | model)) for all y. The log of a small probability is a very negative number, so a model with a low likelihood for all outputs (meaning the model assigned a low probability to all of the true classes) will have a very negative log-likelihood, while a model with a high likelihood for all outputs will have a log-likelihood that is close to zero, but still negative. Because most machine learning optimization procedures think of the cost in positive terms, a negative sign is added to the front of the log-likelihood... so now minimizing the cost function is equivalent to maximizing the likelihood of the model!
2 • Reply • Share ›

Denny Britz Mod → Michael A. Alcorn • 3 years ago
Hi Michael. Thanks for for clarifying that. I'll update the post to make it clearer, really appreciate it!
1 • Reply • Share ›

Praveen Kumar • 3 years ago
Thanks for the article. It cleared many of my doubts regarding neural networks.

2 ^ | ˅ • Reply • Share ›

Denny Britz Mod ➦ Praveen Kumar • 3 years ago

Glad that you found it useful!

^ | ˅ • Reply • Share ›

Jaydeep Kulkarni • 9 months ago

Hi
I have seen example of softmax + cross entropy loss where y is a "one hot encoded".
I see that its not the case here as you have retained y as pure labels.
Is there any particular advantage/disadvantage of having y as "one hot encoded"?
Btw .. great tutorial ..thanks a lot.

1 ^ | ˅ • Reply • Share ›

Dushyant Chauhan • a year ago

will anyone tell that why we used "Set min and max values and give it some padding"?? and how does clf.fit(X, y) function work?? I have no idea for this so please help to understand this.

1 ^ | ˅ • Reply • Share ›

Jere Kabi ➦ Dushyant Chauhan • 8 months ago

"Set min and max values and give it some padding"?? It is a good way to generate more data when using np.linspace func also it makes the plots have a good range in the x1 and x2 dimensions

clf.fit(X,y) -- here the code is doing logistic regression which tries to separate the data using a straight line..read up on details of how to do logistic regression to undetstand clearly..coursera machine learning course by Andrew Ng is recommended.

^ | ˅ • Reply • Share ›

Yasser Souri • 3 years ago

Hi, thanks for the blog post. Just some ideas.
1. It would be nice to have a function called forward like this:

```
def forward(W1, b1, W2, b2, x):
z1 = x.dot(W1) + b1
a1 = np.tanh(z1)
z2 = a1.dot(W2) + b2
exp_scores = np.exp(z2)
```

y_hat = exp_scores / np.sum(exp_scores, axis=1,
keepdims=True)
return y_hat, z1, a1, z2

This way you could reuse the same code in both calculate_loss, predict and build_model. e.g. predic would be something like this:

def predict(model, x):
W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'],
model['b2']

see more

1 ^ | v • Reply • Share ›

Denny Britz Mod ➦ Yasser Souri • 3 years ago

Hey Yasser! Thanks for the feedback. You're absolutely right, these would be some good improvements to the code. I'll see if I get around to implementing them. For the biases, there's no particular reason I initialized them to zero, I think their initialization doesn't matter as much as that of the other parameters, which is quite important (I could be wrong though).

^ | v • Reply • Share ›

Phil Glau ➦ Denny Britz • 2 years ago

I believe that the regularization loss should not be normalized by the number of example

Rather than:
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

I think you would want:

data_loss = np.sum(corect_logprobs) /
float(num_examples)
# Add regulatization term to loss (optional)
data_loss += reg_lambda/2 *
(np.sum(np.square(W1)) +
np.sum(np.square(W2)))
return data_loss

^ | v • Reply • Share ›

Dan Marthaler • 3 years ago

Dan Marthaler • 3 years ago

Thanks! This is really nice. Note, you can next apply drop out to the moon data. From that you should (hopefully) see a static boundary after adding more hidden units. Might be a good way to segue into more advanced NN theory.

1 ^ | ˅ • Reply • Share ›

Denny Britz Mod → Dan Marthaler • 3 years ago

Thanks Dan! That's a great idea for a follow-up post, I will definitely add that to my list of things I want to write about.

^ | ˅ • Reply • Share ›

Aron Bordin • 3 years ago

Nice post! Thx

1 ^ | ˅ • Reply • Share ›

Dushyant Chauhan • a year ago

please explain me this code, i am so confused what is the purpose of this code.

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
h = 0.01
# Generate a grid of points with distance h between them
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))
# Predict the function value for the whole gid
Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

I am not getting why all operation is performed on "xx and yy" instead of "X". And why are we using np.arrange concept here???? What do you really want to do using meshgrid function????

1 ^ | ˅ • Reply • Share ›

Stephen Nobles → Dushyant Chauhan • a year ago

The new variables are used as input to the contour plot. xx becomes a vector from x_min to x_max in increments of h; yy is similar. The function to be plotted (contoured) is evaluated at each xx, yy; the evaluations are assigned to Z, which is reshaped to preserve the x and y arrangement.

^ | ˅ • Reply • Share ›

Ehsan • 3 years ago

Hi Denny, thanks for the great post! I want to extend this code

Hi Denny, thanks for the great post! I want to extend this code to solve a regression problem rather than classification. Can you point me to the right direction? Which parts of the code need to be changed? thanks

1 ^ | ˅ • Reply • Share ›

Denny Britz Mod ➜ Ehsan • 3 years ago

There are a couple of things:

- You need to replace the output layer softmax with a matrix multiplication that produces a single number, which would be your regression prediction.
- We're using cross-entropy to learn the network weights, but that's a loss function for categorization. You need to change it to something suitable for regression, e.g. squared loss.
- Since you're changing the network and the loss function you need to recalculate the gradient equations using these changes.

Btw, I'm not sure how much sense it makes to use this for regression. The nonlinearities like tanh allow you to learn nonlinear decision boundaries, but I don't have intuitive interpretation for them for regression. If you take out the hidden layer and the nonlinearities, you're basically left with a simple Linear Regression model.

1 ^ | ˅ • Reply • Share ›

Ehsan ➜ Denny Britz • 3 years ago

Thank you very much for the reply Denny! I have done the changes above and tested it with my data set! good news is, I can learn. My squared error decreases and I get a good fitting after the last iteration. However, when I increase the number of neurons in my hidden layer (more than 6) the error rate grows instead and grows really fast (super large!). I suspect that I am making a mistake in my derivation calculations.
Assuming that I use the same non-linear function in my hidden layer, square error as my error function and a linear function in my final layer to determine the output (output = z2), do the following derivations look right to you?
delta3 = 2*(output-y)
dW2 = (a1.T).dot(delta3)
db2 = np.sum(delta3, axis=0)
delta2 = delta3.dot(nn['W2'].T) * (1 -

np.power(a1, 2))
dW1 = np.dot(X.T, delta2)
db1 = np.sum(delta2, axis=0)

• Reply • Share ›

Denny Britz Mod → Ehsan • 3 years ago

Hm, I don't have time to do the derivation in detail right now. Did you try lowering the learning rate? Maybe it's just too high.

If you want to make sure that you gradients are correct the best way is probably do implement a gradient check. I have some code for gradient checking here in this post: http://www.wildml.com/2015/...

But you can probably also find other code for gradient checking online.

1 • Reply • Share ›

Ehsan → Denny Britz • 3 years ago

Oh Great! will use that to check the derivations. Thanks a lot for your help. Really appreciate it :)

1 • Reply • Share ›

Denny Britz Mod → Ehsan • 3 years ago

Great, let me know if you figure it out what's wrong ;)

1 • Reply • Share ›

Ehsan → Denny Britz • 3 years ago

Sure thing! will do ;)

• Reply • Share ›

Weiguan Wang → Ehsan • 2 years ago

hello Ehsan, I got the same derivatives as you listed above, but I cannot train it. Can you tell me how you did it?

• Reply • Share ›

Ling Ma • 3 years ago

I do love the layout of this page, just like the zen of python. Thank you!

1 ^ | v • Reply • Share ›

Akash • 10 days ago

Greetings! I'm working on Intrusion Detection System based on ANN. Any suggestion for the implementation?

^ | v • Reply • Share ›

Zach Munro • 4 months ago

I am trying to implement this but instead of it being a [2,5,2] network I need it to be a [4,5,3] network. I am using it with the popular Iris Classification data set. Here is my code on github, https://github.com/zmunro/I...
If anyone could offer some advice it would be greatly appreciated. Right now it the loss stays consistent at 0.3657 so I am definitely doing some things wrong.

^ | v • Reply • Share ›

ömer • 4 months ago

it is a great works to understand NN mechanism

^ | v • Reply • Share ›

Cerebus • 6 months ago

Hey! Thanks for a nice read. I've got a questions concerning the initialization of the weights:

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))

Why do you divide by the square root of the dimension of the input to W1 and W2? Is that some kind of normalization? If so, what does that give us? What's the mathematical motivation for it?

Cheers!

^ | v • Reply • Share ›

Sameer Mahajan • 9 months ago

very good article! thanks for posting!