

WILDML

AI, DEEP LEARNING, NLP

≡ MENU

IMPLEMENTING A NEURAL NETWORK FROM SCRATCH IN PYTHON – AN INTRODUCTION

September 3, 2015

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

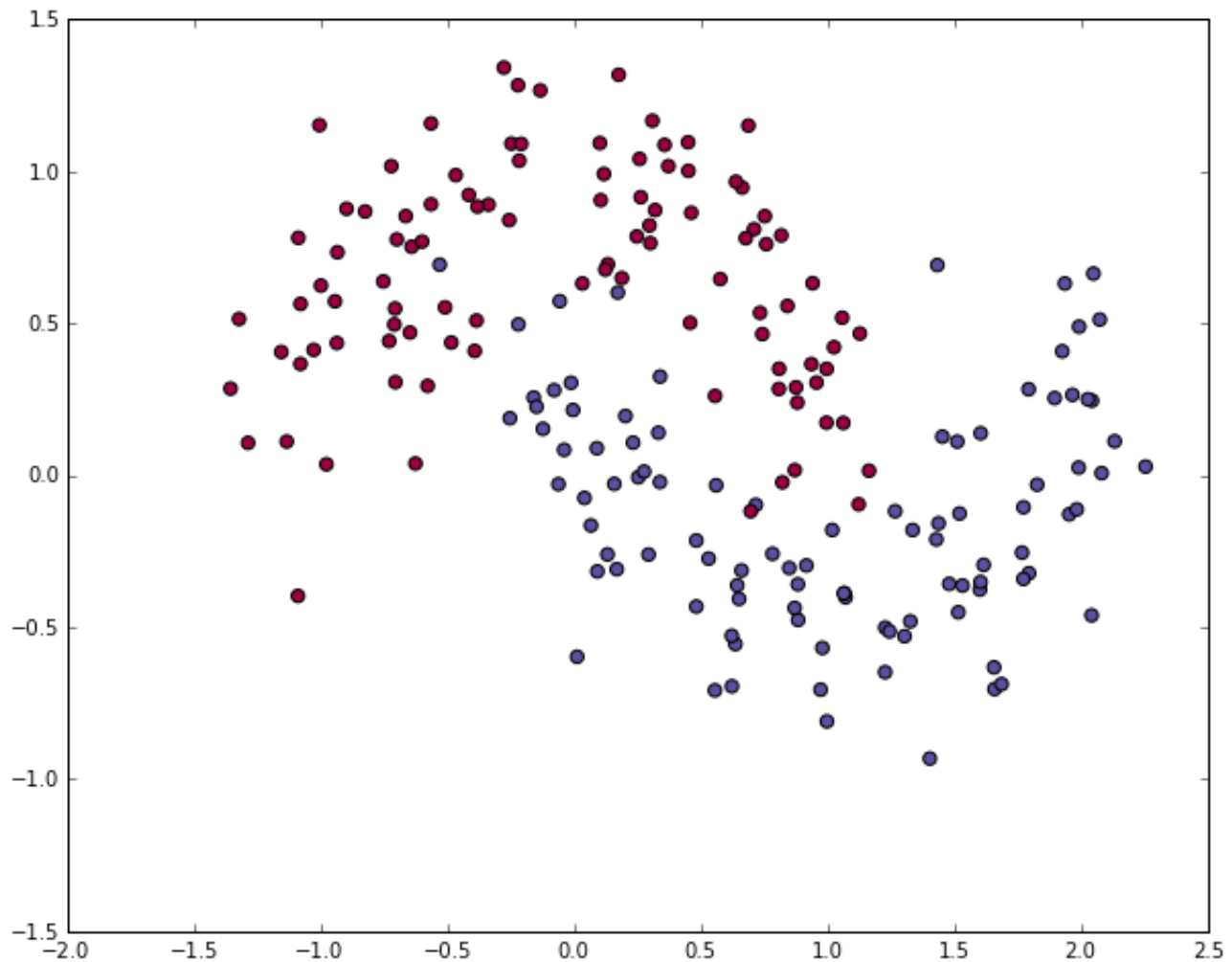
understanding of how neural networks work, and that is essential for designing effective models.

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**)

GENERATING A DATASET

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 or 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

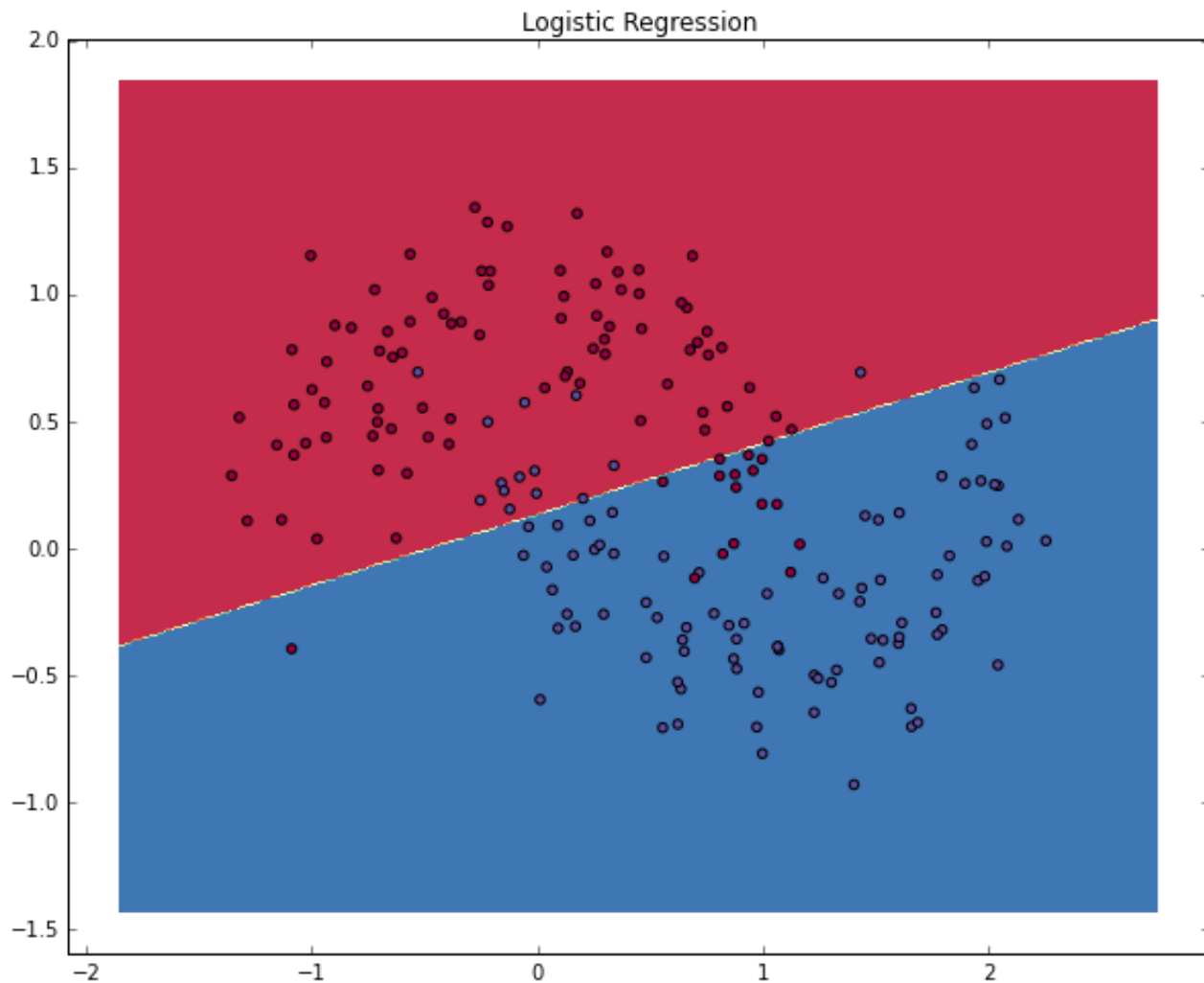
worry about **feature engineering**. The hidden layer of a neural network will learn features for you.

LOGISTIC REGRESSION

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 *scikit-learn*.

```
# Train the logistic regression 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")
```

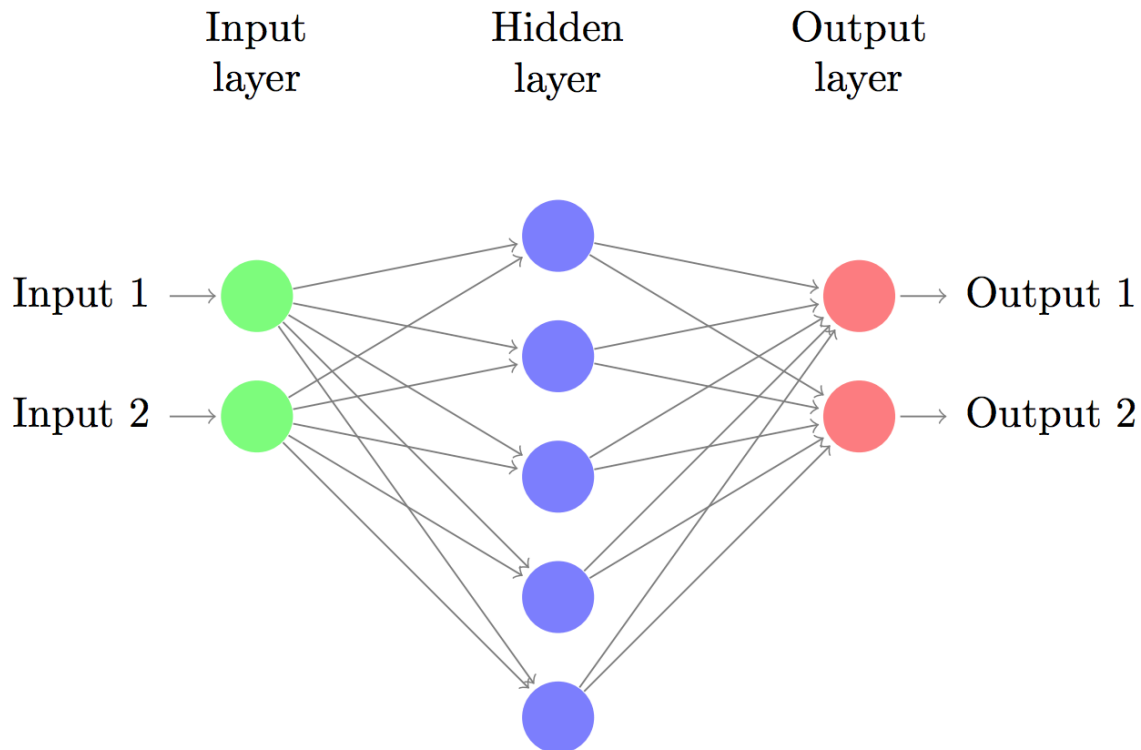


The graph shows the decision boundary learned by our Logistic Regression classifier. It separates the data as good as it can using a straight line, but it's unable to capture the "moon shape" of our data.

TRAINING A NEURAL NETWORK

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 to 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 layer later on and see how it affects our output.

We also need to pick an *activation function* 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 choices for activation functions are **tanh**, the **sigmoid function**, or **ReLU**s. We will use *tanh*, which performs quite well in many scenarios. A nice property of these functions is that their derivative 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:

$$\begin{aligned} z_1 &= xW_1 + b_1 \\ a_1 &= \tanh(z_1) \\ z_2 &= a_1W_2 + b_2 \\ a_2 &= \hat{y} = \text{softmax}(z_2) \end{aligned}$$

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 $W_1 \in \mathbb{R}^{2 \times 500}$, $b_1 \in \mathbb{R}^{500}$, $W_2 \in \mathbb{R}^{500 \times 2}$, $b_2 \in \mathbb{R}^2$. 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 *loss function*. 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 *backpropagation algorithm*, 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.square(W2)))
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 derivatives we found above.

```

# This function learns parameters for the neural network and returns the model.
# - nn_hdim: Number of nodes in the hidden layer
# - num_passes: Number of passes through the training data for gradient descent
# - 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 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 = {}

# 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 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

    # 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
    if print_loss and i % 1000 == 0:
        print "Loss after iteration %i: %f" %(i, calculate_loss(model))

return model

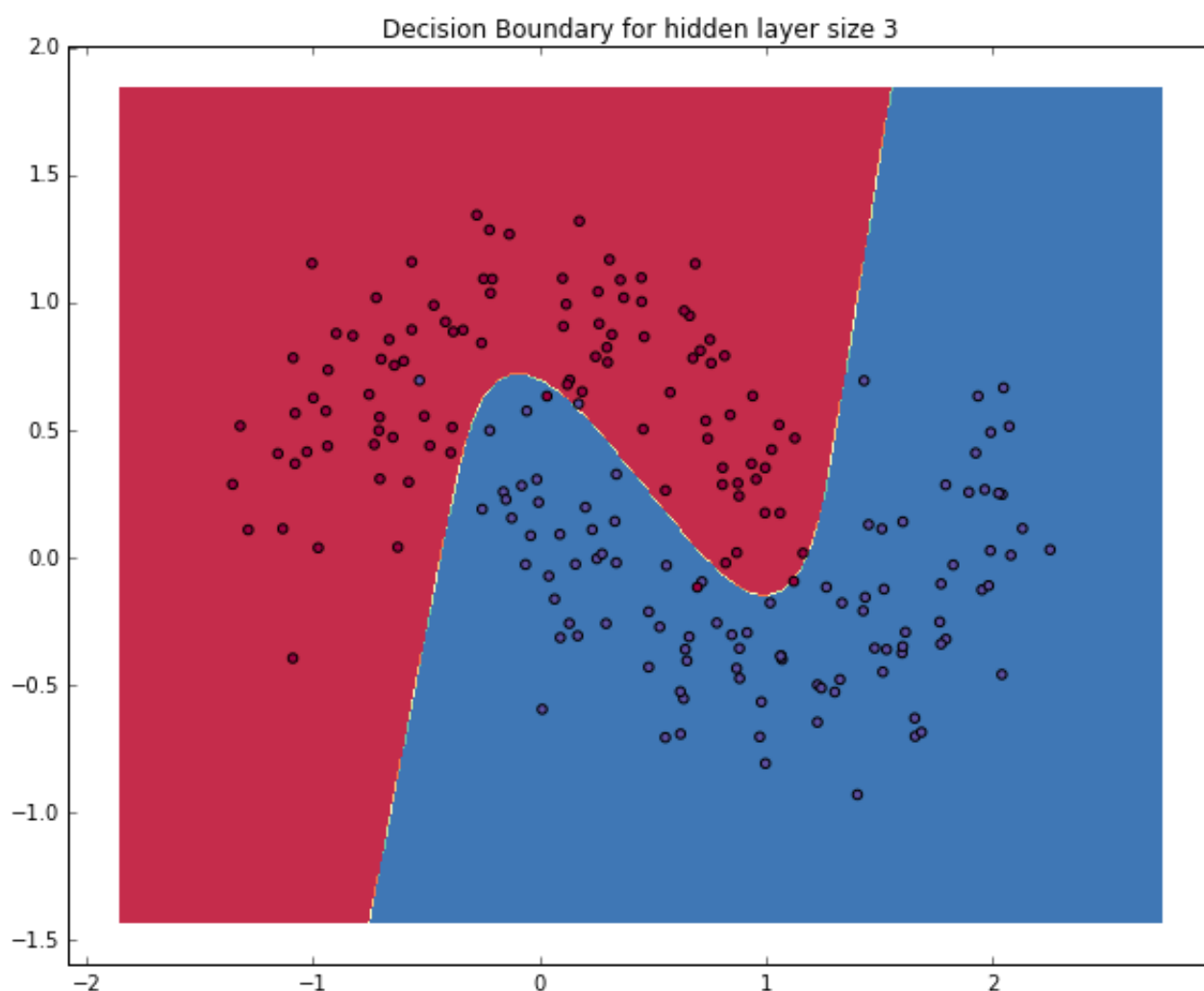
```

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")
```

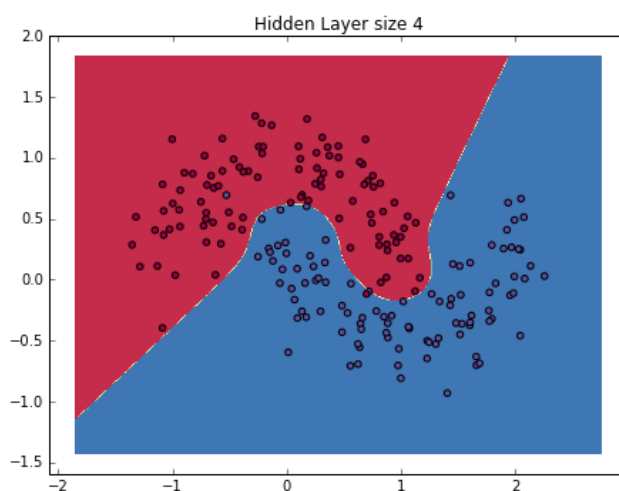
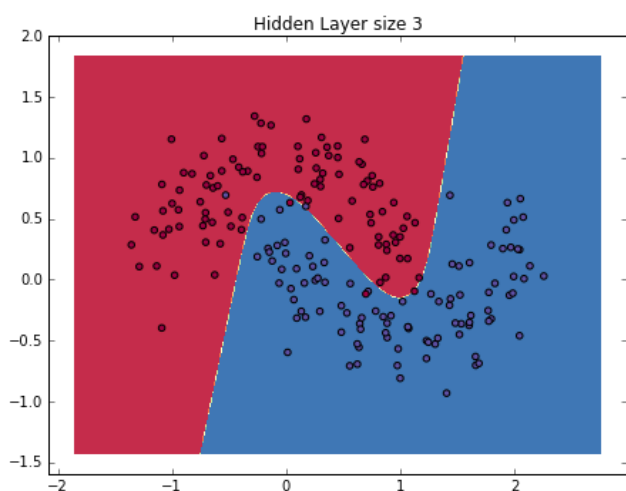
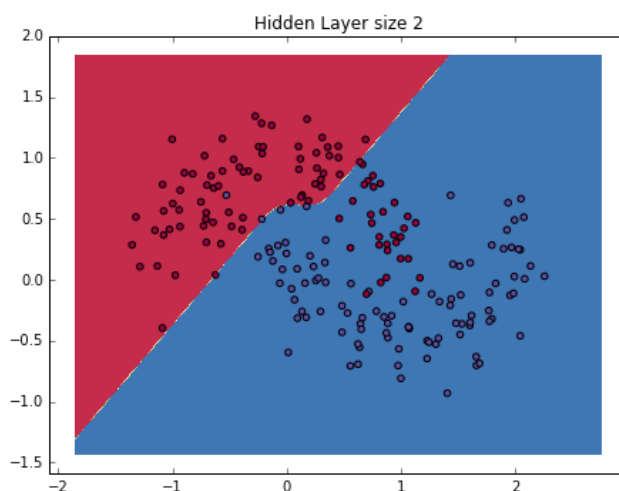
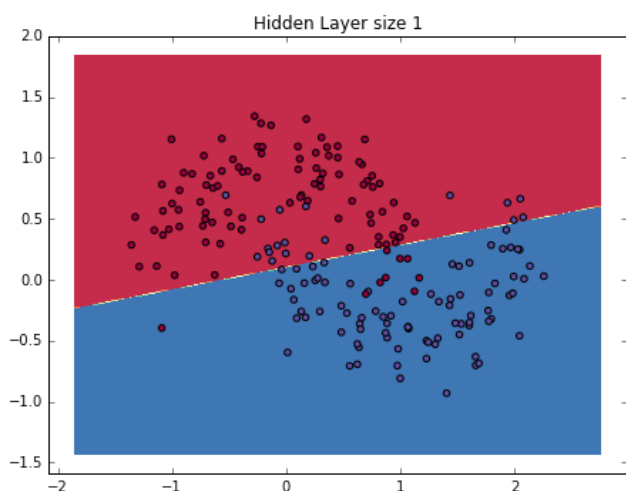


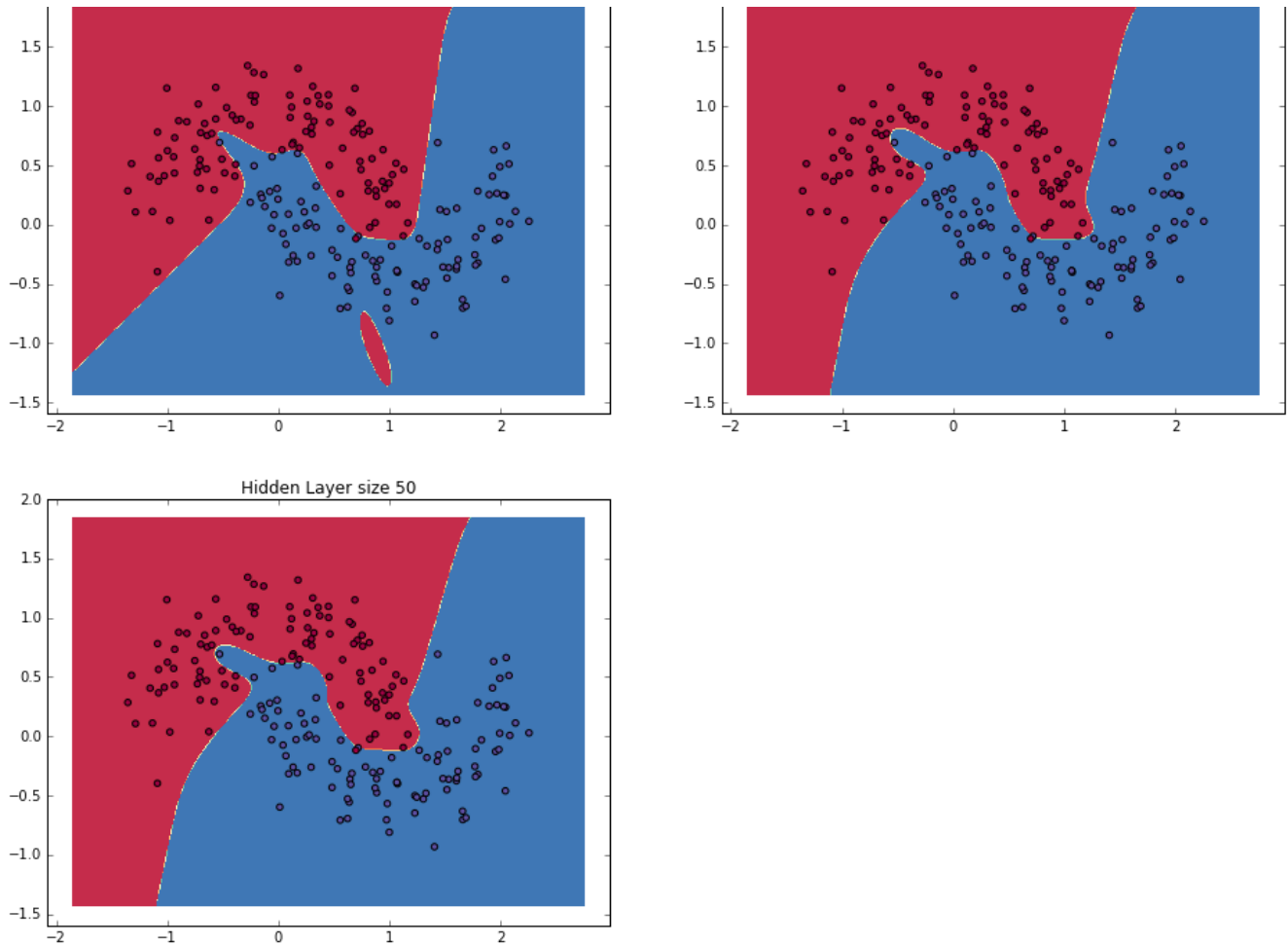
Yay! This looks pretty good. Our neural networks was able to find a decision boundary that successfully separates the classes.

VARYING THE HIDDEN LAYER SIZE

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.

EXERCISES

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!

Posted in: Deep Learning, Neural Networks

Speeding up your Neural Network with Theano and the GPU →

101 Comments

WildML

 Login ▾

 Recommend 18

 Share

Sort by Best ▾



Join the discussion...



electrocured • / months 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!

2 ^ | v • Reply • Share ›



Zhu Xiaohu • a year ago

Hi, thanks for this gentle introduction to NN. I find a small typo in formula for δ_2 . Should it be $\delta = (1 - \tanh^2 z_1)$?

1 ^ | v • Reply • Share ›



Denny Britz Mod → Zhu Xiaohu • a year ago

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

^ | v • Reply • Share ›



Samuel Bedard → Denny Britz • 5 months ago

its also in the code : $\delta_2 = \delta_3 \cdot (1 - \text{np.power}(a_1, 2))$
should be :

```
 $\delta_2 = \delta_3 \cdot (1 - \text{np.power}(z_1, 2))$ 
```

^ | v • Reply • Share ›



Deploribus Unum ☆ TRUMP → Denny Britz • a year ago

if $a_1 = \tanh(z_1)$, then does $a_1^2 = \tanh^2(z_1)$? (and $1 - a_1^2 = 1 - \tanh^2(z_1)$)?

^ | v • Reply • Share ›



Zhu Xiaohu → Denny Britz • a year ago

You're so kind. :) Look forward for your new great articles!

^ | v • Reply • Share ›



Michael A. Alcorn • a year 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 $p(y \mid \text{model})$ for all y , while the log-likelihood is calculated by adding $\log(p(y \mid \text{model}))$ 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!

1 ^ | v • Reply • Share ›



Denny Britz Mod → **Michael A. Alcorn** • a year ago

Hi Michael. Thanks for for clarifying that. I'll update the post to make it clearer, really appreciate it!

1 ^ | v • Reply • Share ›



Yasser Souri • a year 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']
    y_hat, _, _, _ = forward(W1, b1, W2, b2, x)
```

```
return np.argmax(y_hat, axis=1)
```

[see more](#)

1 [^](#) | [v](#) • [Reply](#) • [Share](#) ›



Denny Britz Mod [→ Yasser Sourì](#) • a year 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](#) • 6 months 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 • a year 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 [^](#) | [v](#) • [Reply](#) • [Share](#) ›



Denny Britz Mod [→ Dan Marthaler](#) • a year ago

Thanks Dan! That's a great idea for a follow-up post. I will definitely add that

— Thanks Denny! 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.

^ | v • Reply • Share ›



Aron Bordin • a year ago

Nice post! Thx

1 ^ | v • Reply • Share ›



Praveen Kumar • a year ago

Thanks for the article. It cleared many of my doubts regarding neural networks.

1 ^ | v • Reply • Share ›



Denny Britz Mod → **Praveen Kumar** • a year ago

Glad that you found it useful!

^ | v • Reply • Share ›



Manuel Korfmann • a year ago

A pleasure to read, thanks for this piece!

1 ^ | v • Reply • Share ›



Denny Britz Mod → **Manuel Korfmann** • a year ago

I'm happy if it was helpful!

^ | v • Reply • Share ›



Ehsan • a year ago

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 ^ | v • Reply • Share ›



Denny Britz Mod → **Ehsan** • a year 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 ^ | v • Reply • Share ›



Ehsan → Denny Britz • a year 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)
```

^ | v • Reply • Share ›



Denny Britz Mod → Ehsan • a year 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/2015/09/...>

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

1 ^ | v • Reply • Share ›

**Ehsan** → Denny Britz • a year ago

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

1 ^ | v • Reply • Share ›

**Denny Britz** Mod → Ehsan • a year ago

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

1 ^ | v • Reply • Share ›

**Ehsan** → Denny Britz • a year ago

Sure thing! will do ;)

^ | v • Reply • Share ›

**Weiguan Wang** → Ehsan • 18 days 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?

^ | v • Reply • Share ›

**Ling Ma** • a year ago

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

1 ^ | v • Reply • Share ›

**roachsina** • 5 days ago

Hi, thanks for your great post!

I've done mini_batch annealing learning rate, use sigmoid and ReLU function. I've some problems.

Mini_batch is well, but loss function is big than SGD. Loss 0.0716. Best one on my exercise.

When I done annealing as $\epsilon = a_0 / (1 + kt)$, where t is iteration, $a_0=1$ $k = 0.005$, I got a line, like just a perceptron. If I use other python even can't plot the contourf, the background is not blue and red, all like yellow(sample was all predict to 1), and the loss is 0.31. And the batch size is 40. Samples is 200, same with this great post, I mean other args.

When I do ReLU, epsilon is fixed 0.01. mini_batch size is 40. It also can't plot the contourf(sample was all predict to 1). Loss is 2.58. If I change SGD it works, but the boundary is not so good. loss is 0.6875.

When I do sigmoid, epsilon is fixed 0.01. And the boundary is good, loss is 0.18.

[see more](#)

^ | v • [Reply](#) • [Share](#) ›



Osama Salah • 17 days ago

I used a mini-batch of sizes 64 and 128 (out of 200 samples) but the loss increased! Shouldn't it decrease?

^ | v • [Reply](#) • [Share](#) ›



Harry • a month ago

Hi Denny, thanks for the great post ! It really helps me to understand the basis of NN. Just one thing I am still confused...

During the training: $\text{delta3}[\text{range}(\text{num_examples}), y] -= 1$. When the true y is 0, why it still subtract 1, instead of 0, according to the formula: $\text{delta3} = \text{predicted_y} - \text{true_y}$??

Also, still the same place: the value corresponding to opposite y should be 0, right? Since this is error, and the opposite value should keep unchanged (just ignore), so the error value should be 0, I think. But that equation still would keep the original value (for opposite true y), why is that ?

Thank you !!

^ | v • [Reply](#) • [Share](#) ›



Osama Salah → **Harry** • 16 days ago

Hi Harry,

The value of y (either 0 or 1) corresponds to the true probability, which is 1. For example, $y[50]=0$ means that the element $\text{probs}[0, 50]$ is the correct class and should equal 1. Similarly, $y[60]=1$ means that the element $\text{probs}[60, 1]$ should equal 1. So, $\text{probs}[\text{ANYTHING}, y]$ should equal 1, and consequently, we always subtract 1 (the true output) from the output that should equal 1.

Can you clarify the second part of the question?

^ | v • [Reply](#) • [Share](#) ›



Kevin Lu • 2 months ago

Hi Denny,

Would you care to show me how you got the values for the backpropagation equations. I've looked at the post that you have linked but I'm kind of stuck?

equations, I've looked at the post that you have linked but I'm kind of stuck?

Thanks :)

^ | v • Reply • Share ›



GP • 2 months ago

Hi Denny, thanks for the great post. Based on your code, I have modified to fit my own data format. However, I face some issues, which is - the values of dW_2 , db_2 , dW_1 , db_1 are too huge. I suspect it is due to too many input dimensions and too many rows of samples (number of rows of input is more than 16k, dimension of input is 9, dimension of output is 7). The code can only work if I divide those four parameters by "num_examples". Do you face such issue? Thanks.

^ | v • Reply • Share ›



Ner0 • 3 months ago

Hi Denny, I want to know if you have the answers(code) for the exercises

^ | v • Reply • Share ›



Ner0 • 3 months ago

Hi Denny, I want to know if you have code for the exercises~

^ | v • Reply • Share ›



Jeet Patel • 3 months ago

Hi,

Can you light me up on how to make a backward pass for CNN for sequence learning.

Also just a clarification, training a particular model and backpropogating the error for that model are two very different thing right?

^ | v • Reply • Share ›



Denny Britz Mod ➔ **Jeet Patel** • 3 months ago

They are related. Backpropagation is a technique to compute gradients, and you train the model by updating the weights in the direction of the gradients.

For an explanation of how to do backprop for CNNs check out

<https://www.youtube.com/watch?...> - That's a very in-depth explanation.





^ | v • Reply • Share ›



Mike Tamillow • 3 months ago

looks like the code came from a m.bashari. I would have hoped you'd have mentioned that...

^ | v • Reply • Share ›



Dang Tran • 4 months ago

Hi Denny, it is a amazing post!! However, I still don't understand how you apply the backpropagation algorithm to find the formulas. Can you explain for me clearly ?
Thanks

^ | v • Reply • Share ›



林建慶 • 5 months ago

Hey! It's a fantastic post! Really useful article.

Can I translate this to Chinese and share on my website ?

I will refer to here.

^ | v • Reply • Share ›



Denny Britz Mod ➔ 林建慶 • 5 months ago

Sure! Thanks for the translation :)

^ | v • Reply • Share ›



Anurag Kumar • 5 months ago

can u explain exp_score and how does it work. Having trouble in loss function.

^ | v • Reply • Share ›



Ahmed Zeidy • 5 months ago

Hi Denny, thanks for the great tutorial. would you give me some advice about using optimization technique such as bfgs with the network

^ | v • Reply • Share ›



TechUser2011 • 5 months ago

Thank you for the great article. At the end where you show the various decision boundaries, the plot for "Hidden Layer size 1" has an apparently linear boundary. Can you please explain why the boundary is linear in this case? I suspect the answer might have to do with the number of input dimensions (two dimensions in your data), but I am not sure.

Extending this question, if I have multiple hidden layers, and each layer is size 1, would I still get a linear boundary?

^ | v • Reply • Share ›



Yiqin Yu • 6 months ago

Great post. Really helpful for those who try to get to the bottom of nn.

^ | v • Reply • Share ›



saurabh sinha • 6 months ago

I am new to machine learning. Can you please explain this code

```
delta3 = probs  
delta3[range(num_examples), y] -= 1
```

^ | v • Reply • Share ›



Cornelis • 7 months ago

Hi Denny, excellent tutorial! Gave me lots of insight in how a neural network works. I'm trying to extend the model to 4 layers (having two hidden layers), but I'm stuck at finding the right derivatives for the backpropagation algorithm. I'm not sure how the chain rule is applied for vectors/matrices. Would you be so kind to help me figuring these out? How for example do you determine δ_2 ? How would the derivatives change if you add an extra hidden layer? Thanks for your help!

^ | v • Reply • Share ›



Denny Britz Mod ➔ Cornelis • 7 months ago

Check out this tutorial on backpropagation, it's pretty good:

<http://colah.github.io/posts/2...>

In practice, people don't usually compute the derivatives by hand. Instead they use a library like Theano or Tensorflow that automatically calculated the derivatives for them. Still, computing them by hand can be a valuable exercise.

^ | v • Reply • Share ›



Ahmed Zeidy • 8 months ago

Hi Denny, thanks for the great tutorial.

^ | v • Reply • Share ›



Lucas David • 8 months ago

I didn't get why you initialized $W_$ values like this: ``np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)`` (by dividing by the sqrt of the input dimension)? It kinda makes me think you want the standard deviation of $W_$ to be ``nn_input_dim^{-1/2}``, but why would you want that?

^ | v • Reply • Share ›

Load more comments

ALSO ON WILDML

Deep Learning for Chatbots, Part 2 – Implementing a Retrieval-Based ...

57 comments • 6 months ago •



Kannan Ravindran — You mentioned “You could imagine feeding in 100 potential responses to a context and ...

Learning Reinforcement Learning (with Code, Exercises and Solutions)

16 comments • 3 months ago •



luf0816 — It's amazing!

Recurrent Neural Networks Tutorial, Part 2 – Implementing a RNN with ...

197 comments • a year ago •



Nitin Agarwal — Thanks for such great post. I was facing problem in understanding the working of ...

Deep Learning for Chatbots, Part 1 – Introduction

39 comments • 9 months ago •



Emilio Srougo — Thanks for the info! Yes, it seems that api.ai and wit are quite similar, their power lies in making the ...

✉ Subscribe Add Disqus to your site Add Disqus Add Privacy

- CONNECT -

- RECENT POSTS -

[Learning Reinforcement Learning \(with Code, Exercises and Solutions\)](#)

[RNNs in Tensorflow, a Practical Guide and Undocumented Features](#)

[Deep Learning for Chatbots, Part 2 – Implementing a Retrieval-Based Model in Tensorflow](#)

[Deep Learning for Chatbots, Part 1 – Introduction](#)

[Attention and Memory in Deep Learning and NLP](#)

[Implementing a CNN for Text Classification in TensorFlow](#)

[Understanding Convolutional Neural Networks for NLP](#)

[Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano](#)

- LINKS -

[Home](#)

[About](#)

[NLP Consulting](#)

- ARCHIVES -

October 2016

August 2016

July 2016

April 2016

January 2016

December 2015

November 2015

October 2015

September 2015

- CATEGORIES -

Conversational Agents

Convolutional Neural Networks

Deep Learning

GPU

Language Modeling

Memory

Neural Networks

[NLP](#)

[Recurrent Neural Networks](#)

[Reinforcement Learning](#)

[RNNs](#)

[Tensorflow](#)

- SUBSCRIBE TO BLOG VIA EMAIL -

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

SUBSCRIBE

- META -

[Log in](#)

[Entries RSS](#)

[Comments RSS](#)

[WordPress.org](#)

PROUDLY POWERED BY WORDPRESS | THEME: SELA BY WORDPRESS.COM.