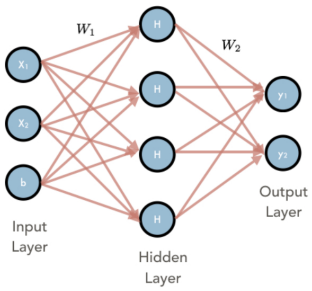
THE NEURAL PERSPECTIVE
DEEP LEARNING SIMPLIFIED

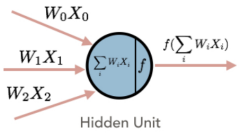
VANILLA NEURAL NETWORK

Edit

I. OBJECTIVE:

Simple 1-layer multi layer perceptron (MLP). f is the activation function and introduces the non-linearity to our system.





Forward Pass:

$$z_2 = XW_1$$
$$a_2 = f(z_2)$$
$$z_3 = a_2W_2$$
$$\hat{y} = f(z_3)$$

II. LINEAR MODEL:

A simple linear model with a softmax layer on top. The main difference here is the lack of a non-linear activation function (ReLU, tanh, etc.). Thanks to Karpathy for the data and code structure, but we will break down the math behind the lines for better understanding. You can check out the code for loading the data on the Github repo but here we will focus on the main model operations.

$$XW$$

```
# Class scores [NXC]
logits = np.dot(X, W)
```

$$\frac{e^{XW_y}}{\sum e^{XW}}$$

```
# Class probabilities
exp_logits = np.exp(logits)
probs = exp_logits / np.sum(exp_logits, axis=1, keepdims=Tr
```

$$-log(p) + \frac{\lambda}{2}\sum_k\sum_l W_{k,l}^2$$

```
# Loss
correct_class_logprobs = -np.log(probs[range(len(probs)), y
loss = np.sum(correct_class_logprobs) / config.DATA_SIZE
loss += 0.5 * config.REG * np.sum(W*W)
```

$$\frac{\partial J}{\partial W_j} = XP, \frac{\partial J}{\partial W_y} = X(P-1)$$

```
# Backpropagation
dscores = probs
dscores[range(len(probs)), y] -= 1
dscores /= config.DATA_SIZE
```

$$\frac{\partial J}{\partial W_i} = \frac{\partial J}{\partial W_i} + \lambda W_i$$

```
dW = np.dot(X.T, dscores)
dW += config.REG*W
```

$$W_i = W_i - \alpha\frac{\partial J}{}$$

- SEARCH -

Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…

Embed View on Twitter

- RECENT POSTS -

Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

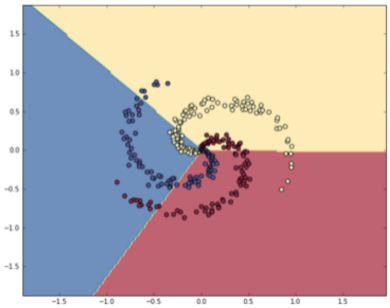
Opening the Black Box of Deep Neural Networks via Information

$$W_i = W_i - \alpha \frac{\partial}{\partial W_i}$$

```
W += -config.LEARNING_RATE * dW
```

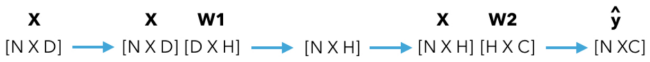
Results:

We can see that the decision boundary of our classifier is linear and cannot adapt to the non-linear contortions of the data.



III. NEURAL NETWORK:

Now we introduce a neural net with a softmax on the last layer for class probabilities. We use a ReLU unit to introduce non-linearity. Our network will have two layers, where the shape of the input will be manipulated as follows:



Once again, let's break down the code.

$$z_2 = XW_1$$

```
z_2 = np.dot(X, W_1)
```

$$a_2 = max(0, XW_1)$$

```
a_2 = np.maximum(0, z_2) # ReLU
```

$$logits = a_2W_2$$

```
logits = np.dot(a_2, W_2)
```

$$P = \frac{e^{logits_y}}{\sum(e^{logits})}$$

```
# Class probabilities
exp_logits = np.exp(logits)
probs = exp_logits / np.sum(exp_logits, axis=1, keepdims=Tr
```

$$loss = -ln(P) + \frac{\lambda}{2}\sum\sum W_1 + \frac{\lambda}{2}\sum\sum W_2$$

```
# Loss
correct_class_logprobs = -np.log(probs[range(len(probs)), y
loss = np.sum(correct_class_logprobs) / config.DATA_SIZE
loss += 0.5 * config.REG * np.sum(W_1*W_1)
loss += 0.5 * config.REG * np.sum(W_2*W_2)
```

$$\frac{\partial J}{\partial W_{2j}} = a_2P, \frac{\partial J}{\partial W_{2y}} = a_2(P-1)$$

```
# Backpropagation
dscores = probs
dscores[range(len(probs)), y] -= 1
dscores /= config.DATA_SIZE
dW2 = np.dot(a_2.T, dscores)
```

$$\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial P}\frac{\partial P}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{W_1} = W_2(dscores)\partial(\text{ReLU})X$$

```
dhidden = np.dot(dscores, W_2.T)
dhidden[a_2 &amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;am
dW1 = np.dot(X.T, dhidden)
```

$$\frac{\partial J}{\partial W_i} = \frac{\partial J}{\partial W_i} + \lambda W_i$$

$\partial W_i \quad \partial W_i$

```
dW2 += config.REG * W_2
dW1 += config.REG * W_1
```

$$W_i = W_i - \alpha \frac{\partial J}{\partial W_i}$$

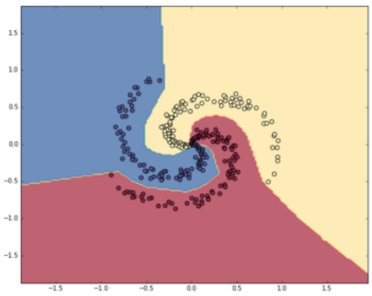
```
W_1 += -config.LEARNING_RATE * dW1
W_2 += -config.LEARNING_RATE * dW2
```

Our accuracy is very simple and involves doing the forward pass and then comparing the predicted class with the target class.

```
def accuracy(X, y, W_1, W_2=None):
    logits = np.dot(X, W_1)
    if W_2 is None:
        predicted_class = np.argmax(logits, axis=1)
        print "Accuracy: %.3f" % (np.mean(predicted_class
    else:
        z_2 = np.dot(X, W_1)
        a_2 = np.maximum(0, z_2)
        logits = np.dot(a_2, W_2)
        predicted_class = np.argmax(logits, axis=1)
        print "Accuracy: %.3f" % (np.mean(predicted_class
```

Results:

The resulting decision boundary is able to classify the non-linear data really well.



```
Epoch: 0, loss: 3.296
Epoch: 100, loss: 1.158
Epoch: 200, loss: 1.133
Epoch: 300, loss: 0.808
Epoch: 400, loss: 0.826
Epoch: 500, loss: 0.572
Epoch: 600, loss: 0.674
Epoch: 700, loss: 0.568
Epoch: 800, loss: 0.682
Epoch: 900, loss: 0.542
Accuracy: 0.977
```

IV. TENSORFLOW IMPLEMENTATION:

We will start by setting up our tensorflow model but we will have an extra function called **summarize()** which will store the progress as we training through the epochs. We will decide which values to store with **tf.scalar_summary()** so we can see the changes later.

```
def create_model(sess, FLAGS):

    model = mlp(FLAGS.DIMENSIONS,
                FLAGS.NUM_HIDDEN_UNITS,
                FLAGS.NUM_CLASSES,
                FLAGS.REG,
                FLAGS.LEARNING_RATE)
    sess.run(tf.initialize_all_variables())
    return model

class mlp(object):

    def __init__(self,
                 input_dimensions,
                 num_hidden_units,
                 num_classes,
                 regularization,
                 learning_rate):

        # Placeholders
        self.X = tf.placeholder("float", [None, None])
        self.y = tf.placeholder("float", [None, None])

        # Weights
        W1 = tf.Variable(tf.random_normal(
            [input_dimensions, num_hidden_units], stddev=0
        W2 = tf.Variable(tf.random_normal(
            [num_hidden_units, num_classes], stddev=0.01),

        with tf.name_scope('forward_pass') as scope:
            z_2 = tf.matmul(self.X, W1)
            a_2 = tf.nn.relu(z_2)
            self.logits = tf.matmul(a_2, W2)

        # Add summary ops to collect data
        W_1 = tf.histogram_summary("W1", W1)
        W_2 = tf.histogram_summary("W2", W2)

        with tf.name_scope('cost') as scope:
            self.cost = tf.reduce_mean(
                tf.nn.softmax_cross_entropy_with_logits(se
                    + 0.5 * regularization * tf.reduce_sum
                    + 0.5 * regularization * tf.reduce_sum

            tf.scalar_summary("cost", self.cost)

        with tf.name_scope('train') as scope:
            self.optimizer = tf.train.AdamOptimizer(
```

```
                learning_rate=learning_rate).minimize(self

    def step(self, sess, batch_X, batch_y):

        input_feed = {self.X: batch_X, self.y: batch_y}
        output_feed = [self.logits, self.cost, self.optimi

        outputs = sess.run(output_feed, input_feed)
        return outputs[0], outputs[1], outputs[2]

    def summarize(self, sess, batch_X, batch_y):
        # Merge all summaries into a single operator
        merged_summary_op = tf.merge_all_summaries()
        return sess.run(merged_summary_op,
            feed_dict={self.X:batch_X, self.y:batch_y})
```

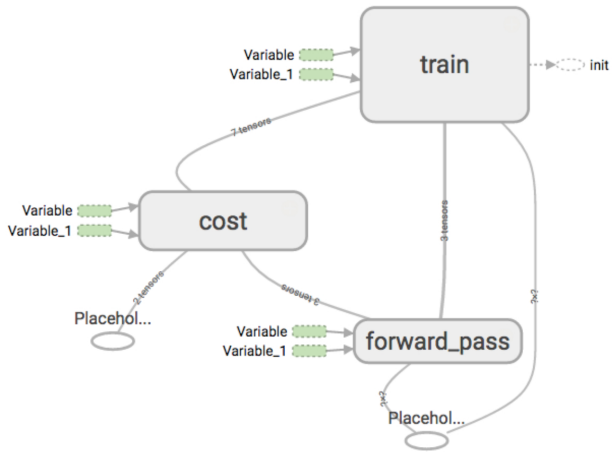
Then we will train for several epochs and save the summary each time.

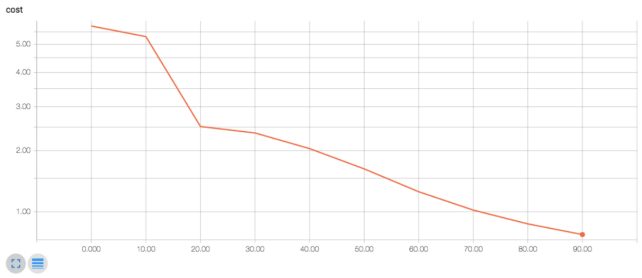
```
def train(FLAGS):

    # Load the data
    FLAGS, X, y = load_data(FLAGS)

    with tf.Session() as sess:

        model = create_model(sess, FLAGS)
        summary_writer = tf.train.SummaryWriter(
          FLAGS.TENSORBOARD_DIR, graph=sess.graph)

        # y to categorical
        Y = tf.one_hot(y, FLAGS.NUM_CLASSES).eval()

        for epoch_num in range(FLAGS.NUM_EPOCHS):
            logits, training_loss, _ = model.step(sess, X,
            # Display
            if epoch_num%FLAGS.DISPLAY_STEP == 0:
                print "EPOCH %i: \n Training loss: %.3f, A
                  % (epoch_num,
                     training_loss,
                     np.mean(np.argmax(logits, 1) == y))

                # Write logs for each epoch_num
                summary_str = model.summarize(sess, X, Y)
                summary_writer.add_summary(summary_str, ep

if __name__ == '__main__':
    FLAGS = parameters()
    train(FLAGS)
```

Finally, we can view our training progress using:

```
$ tensorboard --logdir=logs
```

and then heading over to **http://localhost:6006/** on your browser to view the results. Here are a few:



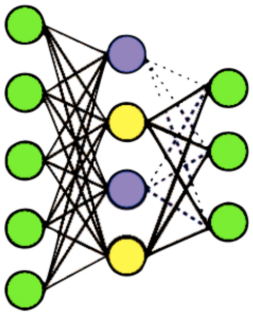


EXTRAS (DROPOUT AND DROPCONNECT):

There are many add on techniques to this vanilla neural network that works to increase optimization, robustness and overall performance. We will be convering many of them in future posts but I will briefly talk about a very common regularization technique: dropout.

What is it? Dropout is a regularization technique that allows us to nullify the outputs of certain neurons to zero. This will effectively be the same as the neuron not existing in the network. We will do this for p% of the total neurons in each layer and for each batch, a new p% of the neurons in each layer are "dropped".

Why do we do this? It works out to be a great regularization technique because for each input batch, we are sampling from a different neural net since a whole new set of neurons are dropped. By repeating this, we are preventing the units from co-adapting too much to the data. The original paper describes each iteration as a "thinned" network because p% of the neurons are dropped. **Note**: Dropout is only for training time. At test time, we will not be dropping any neurons.

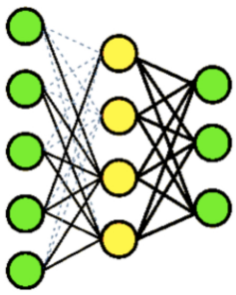


In the image above, the layer has p=0.5 which means half of it's units are dropped. In an other iteration, a different set of 1/2 of the neurons will be dropped. Let's take a look at masking code to really understand what's happening.

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$

We use a Bernoulli distribution to generation 0/1 with probability p for 0. We apply this mask to the outputs from our layer. The parts that are multiplied by zero are our "dropped" neurons since they will yield an output of 0 when multiplied by the next set of weights.

Another regularization method, which is an extension of dropout, is dropconnect. It also involves a similar mechanism but is applied to the weights instead.



Notice that here, a set of weights are dropped instead of the neurons.

$$
\begin{aligned}
mask_{ij}^{(l)} &\sim Bernoulli(p) \\
\tilde{W}^{(l)} &= W^{(l)} * mask^{(l)} \\
z_i^{(l+1)} &= \sum_{j=1}^{n} \tilde{W}_{ij}^{(l)} y_j^{(l)} + b_i^{(l)} \\
y^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

We apply a similar bernoulli mask to the weights and we use those weights for the layers. Any inputs that are dot producted with the zeroed weights will result in 0. You can see the similarity with dropout and so, empirically, both techniques offer similar results. Dropconnect was proposed because you always have more weights than neurons, so there are more ways to create "thinned" models thus results in more robust training. However, in more papers you will see mostly dropout being utilized and very rarely drop connect since results are similar.

You can read more about dropout **here** and dropconnect **here**.

V. RAW CODE:

GitHub Repo (Updating all repos, will be back up soon!)

More

Posted in: *Uncategorized*
Tagged: *Tutorials*

← *Logistic Regression* *Convolutional Neural Networks (CNN)* →

2 THOUGHTS ON "VANILLA NEURAL NETWORK"

OLGA August 17, 2017 at 9:30 pm EDIT REPLY →

One of the most succinct and clearest NN guides I've seen so far, great material

★ Liked by you

GOKUMOHANDAS August 17, 2017 at 10:15 pm EDIT REPLY →

Thanks 😉 made it for myself initially just to be able to look at it and recall everything quickly

★ Like

LEAVE A REPLY

Enter your comment here...