GEOPHYSICAL TUTORIAL, August 2018

Neural networks

by GRAM, Head of Data Science, Expero Inc

There's no magic here.

We've all heard a proselytizing hyperbolist make the AI-is-going-to-steal-my-job speech. If you subscribe, look at the code in the notebook accompanying this tutorial at https://github.com/seg/tutorials-2018. It demonstrates a small neural network. You'll find a simple system composed chiefly of multiply and add operations. That's really all that happens inside a neural network. Multiply and add.

A neural network is nothing but a nonlinear system of equations like $\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$. In this demonstration, the nonlinearity is introduced by the *sigmoid*, aka *logistic*, function and its derivative:

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}}, \text{and } \frac{\partial \sigma(z)}{\partial z} = z(1 - z)$$

We need the derivative for the *backpropagation* process that enables neural networks to learn efficiently. Backpropagation adjusts the parameters of the neural network by injecting an error signal backwards through the network's layers, from the last to the first.

The sigmoid function looks like this in Python:

```python
def sigma(z, forward=True):
    if forward:
        return 1 / (1 + np.exp(-z))
    else:
        return z * (1 - z)
```

The function transforms, or 'squashes', numbers into the range [0, 1] and looks like this:

Define the network

We are now ready to implement the neural network itself. Neural networks consist of three or more *layers*: an input layer, one or more *hidden* layers, and an output layer.

Let's implement a network with one hidden layer. The layers are as follows:

$$\text{Input layer: } \mathbf{x}^{(i)}$$

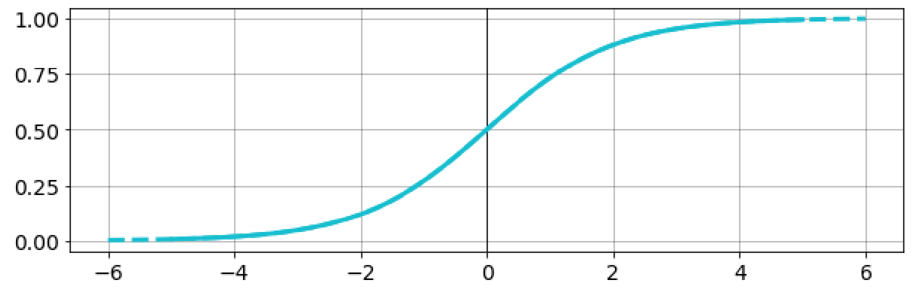

Figure 1: png

$$\text{Hidden layer: } \mathbf{a}_1^{(i)} = \sigma(\mathbf{W}_1\mathbf{x}^{(i)} + \mathbf{b}_1)$$

$$\text{Output layer: } \hat{\mathbf{y}}^{(i)} = \mathbf{W}_2\mathbf{a}_1^{(i)} + \mathbf{b}_2$$

where $\mathbf{x}^{(i)}$ is the i-th sample of the input data $\mathbf{X}$. $\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2$ are the weight matrices and bias vectors for layers 1 and 2 respectively, and σ is our nonlinear function. Applying the nonlinearity to $\mathbf{W}_1\mathbf{x}^{(i)} + \mathbf{b}_1$ in layer 1 results in the *activation* $\mathbf{a}_1$. The output layer yields $\hat{\mathbf{y}}^{(i)}$, the i-th estimate of the desired output. We're not going to apply the nonlinearity to the output, but people often do. The weights are randomly initialized and the biases start at zero; during training they will be iteratively updated to encourage the network to converge on an optimal approximation to the expected output.

We'll start by defining the forward pass, using NumPy's `@` operator for matrix multiplication:

```
def forward(xi, W1, b1, W2, b2):
    z1 = W1 @ xi + b1
    a1 = sigma(z1)
    z2 = W2 @ a1 + b2
    return z2, a1
```

Below is a picture of a neural network similar to the one we're building:

We see a simple neural network which takes 3 numbers as input (the green neurons) and outputs one number (the red neuron). In the middle (the orange neurons) we have a so-called *hidden layer*, which in this case has 5 neurons or *units*. Moving information from input layer, to hidden layer, to output layer is as simple as matrix multiplying and adding numbers. In the middle we apply the sigmoid function to each of the numbers.

We can "teach" this simple system to model a mapping between one set of numbers and another set. For example, we can train this system to output a two

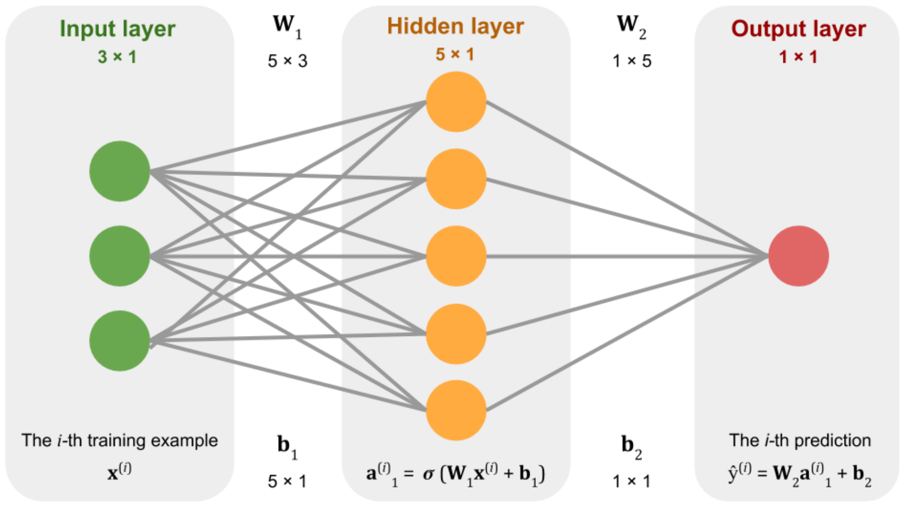

Figure 2: image

when we input a one, a four when we input a two, and $2N$ when we input an N. This is equivalent to building a linear model. More interestingly, we could teach it to output a nonlinear model: one maps to one, two maps to four, and N maps to N^2. More interestingly still, we could teach it to combine multiple inputs into a single output.

In this tutorial, we'll train a model like this to learn the reflectivity for P–P reflections at an interface. (Normally we'd use the Zoeppritz equation to do this — our only purpose here is to show that even a simple neural network can learn a nonlinear function. We wouldn't really want to compute the reflectivity this way.)

Instead of 3 inputs, we'll use 7: V_P, V_S and ρ for the upper and lower layer properties at each interface, plus the angle of incidence, θ at each interface. And instead of 5 units in the hidden layer, we'll use 300.

How does the network "learn?" The short version is that we show the system a bunch of corresponding input/output pairs we want it to learn, and we show it these pairs thousands of times. Every time we do so, we move the **W**'s and **b**'s in whatever direction makes the outputs of the network more similar to the known output we're trying to teach it.

This iterative adjustment of weights and biases relies on a process called *back-propagation of errors*.

Backpropagation is the critical piece of thinking which enabled the deep learning revolution. It is the reason Google can find images of flowers, or translate from Hindi to English. It is the reason we can predict the failure of drilling equipment days in advance of failure (see my video at http://bit.ly/2Ks5tQf for more on this).

Here is the backpropagation algorithm we'll employ:

```
For each training example:
    For each layer:
      - Calculate the error.
      - Calculate weight gradient.
      - Update weights.
      - Calculate the bias gradient.
      - Update biases.
```

This is straightforward for the output layer. However, to calculate the gradient at the hidden layer, we need to compute the gradient of the error with respect to the weights and biases of the hidden layer. That's why we needed the derivative in the `forward()` function.

Let's implement this as a Python function:

```python
def backward(xi, yi,
             a1, z2,
             params,
             learning_rate):

    err_output = z2 - yi
    grad_W2 = err_output * a1
    params['W2'] -= learning_rate * grad_W2

    grad_b2 = err_output
    params['b2'] -= learning_rate * grad_b2

    derivative = sigma(a1, forward=False)
    err_hidden = err_output * derivative * params['W2']
    grad_W1 = err_hidden[:, None] @ xi[None, :]
    params['W1'] -= learning_rate * grad_W1

    grad_b1 = err_hidden
    params['b1'] -= learning_rate * grad_b1

    return params
```

To demonstrate this backpropagation workflow, and thus that our system can learn, let's try to get the above neural network to learn the Zoeppritz equation. We're going to need some data.

Training data

We could make up some data, but it's more fun to use real logs. We'll use the R-39 well from offshore Nova Scotia.

Here are the logs and the reflectivity series we're training on:

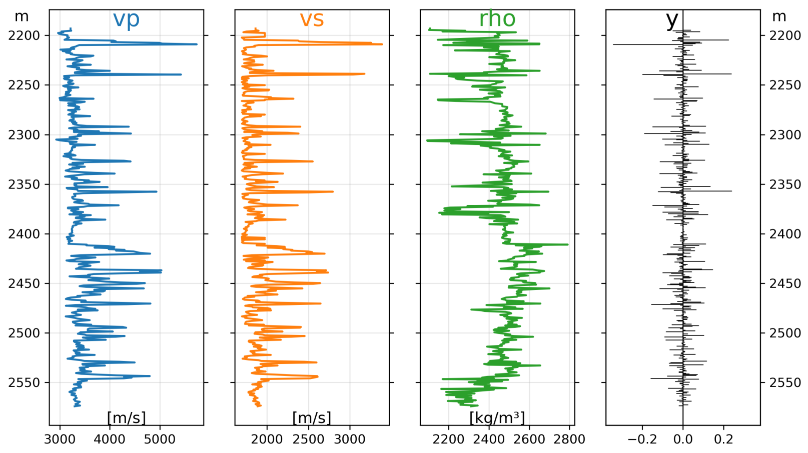

Figure 3: image

After processing the data and reserving 20% of it for for validation testing, we have a feature matrix `X` with shape 400 × 7, and a label vector `y` with 400 elements. The feature matrix has one row for each data sample, and one column for each feature. The features are V_P, V_S, and ρ for the upper and lower layer at each sample, plus the angle of incidence — 7 features in all. The labels in `y` are the reflectivities calculated from each set of features.

Train the network

Now we can initialize the weights and biases for our network. A common approach is to initialize the weights with small random numbers (with NumPy's `randn()` function) and the biases with zeros:

```
def initialize_params(units, features):
    np.random.seed(42)
    params = {
        "W1": 0.1 * randn(units, features),
        "b1": np.zeros(shape=units),
        "W2": 0.1 * randn(units),
        "b2": np.zeros(shape=1)
    }
    return params

units = 300
features = X_train.shape[-1]

params = initialize_params(units, features)
```

During training, we expose the network to the input/output pairs one at a time. These pairs are called `xi` and `yi` respectively in the code. According to our diagram above, the input goes into the green slots and we adjust the orange neurons to make the red slot output from the network a tiny bit closer to the true Zoeppritz result.

We do this many times. Every time we do, we calculate the mean squared error between the network's prediction and the ground-truth output. After many iterations, or *epochs*, we draw a plot which shows the total error, or loss, at each step. If the network is learning anything, we expect the loss to decrease, as the predictions are getting closer to the ground truth.

```
num_epochs = 100
learning_rate = 0.001
loss_history = []

data = list(zip(X_train, y_train))

for i in tnrange(num_epochs):

    np.random.shuffle(data)
    loss = 0

    for xi, yi in data:
        z2, a1 = forward(xi, **params)

        params = backward(xi, yi,
                          a1, z2,
                          params,
                          learning_rate)

        loss += np.square(z2 - yi)

    loss_history.append(loss/y_train.size)
```

In practice, we also predict a result from the validation set, capturing the loss on it too. This tells us how well the network generalizes to data it did not see during training, and whether the network is overtraining. See the complete code in the notebook; it produces the following loss curves:

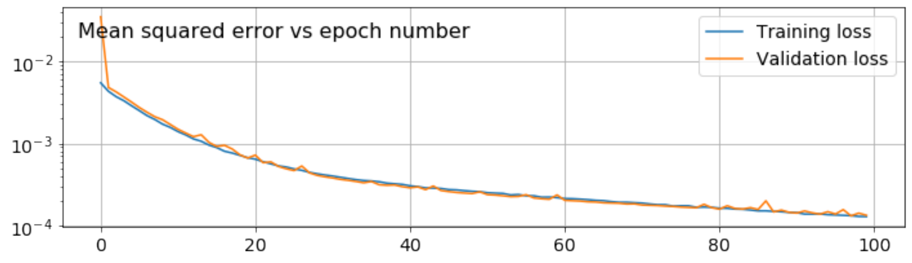

Figure 4: png

The loss decreased dramatically over the course of 100 epochs, so presumably the network has learned something. To test this theory, let's plot the first 100 network outputs before (green) and after (orange) training and compare them to the expected result (blue):

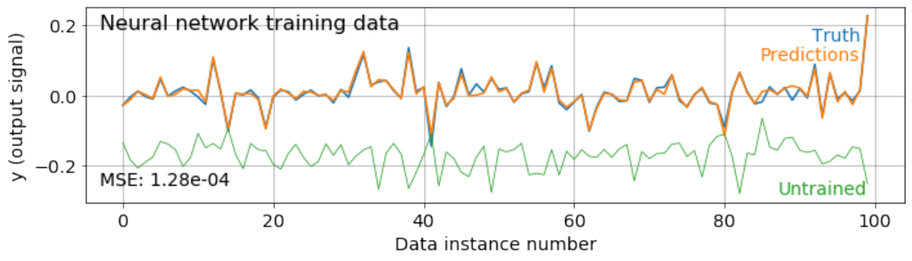

Figure 5: png

To see how well the network predicts data it was not trained on, we can compare the output from the validation set with the ground truth:

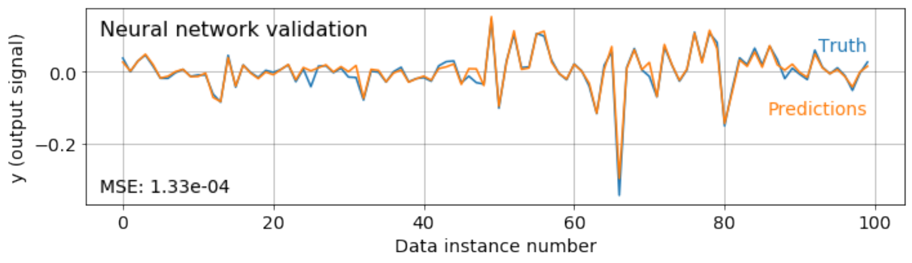

Figure 6: png

Blind test: new rocks

The chart above shows the result of applying the neural network to data that it was not directly trained on, but is from the same rocks that we trained on. Let's test the network on more different data to what it has seen before. We'll use the higher-impedance rocks from near the bottom of the same well — we say that the data are outside the *span of the input domain*.

In the plot below, the blue line is the true Zoeppritz result. The green line is the output from the network before training (i.e. with random weights). The orange line is the output from the network after training (i.e. after the weights have been learned).

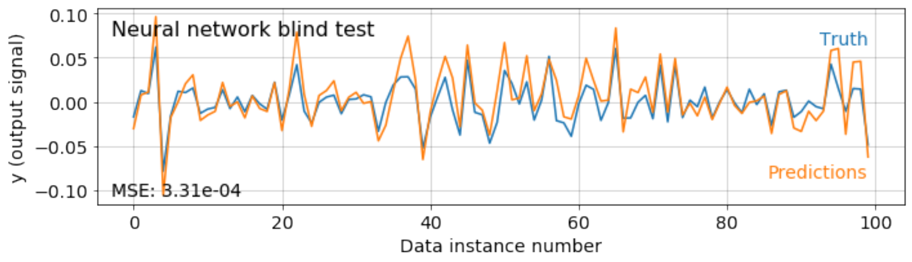

Figure 7: png

So, indeed, our neural network has learned to approximate the Zoeppritz equation, and that it generalizes to rocks it did not see during training.

Blind test: new angles

As well as predicting the reflectivity for rocks we did not train on, we can try predicting reflectivity at angles we did not train on. We'll look at the AVO response at a single interface and see how it compares to the exact solution.

The network has maybe not generalized as well as we hoped. It is at least in the right ballpark inside the range of angles over which it trained. But it doesn't do all that well outside that range. We probably need to make more training data — this is left as an exercise for the reader. You can find all the code to run this network yourself at https://github.com/seg/tutorials-2018.

Summary

That's it. That's all of deep learning. Multiply, add, apply nonlinearity. There's really no magic. It's just simple arithmetic.

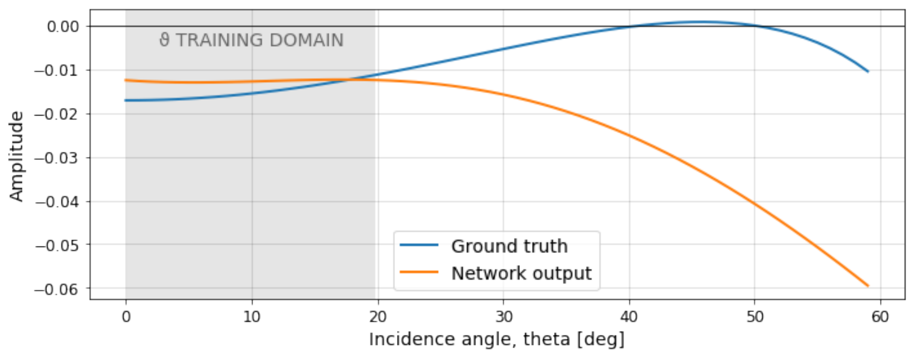

Figure 8: png

Acknowledgments

Thanks to Expero for encouraging me to continue to publish. Thanks to Matt Hall for continuously pushing our professional community to modernize with efforts like the one you're reading now. Thanks to Lukas Mosser and Jesper Dramsch for helpful feedback on the draft of this manuscript.