Deep Learning for Regression

By: The Lazy Programmer
https://lazyprogrammer.me
https://twitter.com/lazy_scientist
https://facebook.com/lazyprogrammer.me

Welcome to this exclusive special report on deep learning for regression. Why did I make this?

I've gotten quite a few requests recently for (a) examples using neural networks for regression rather than classification, and (b) examples using time series.

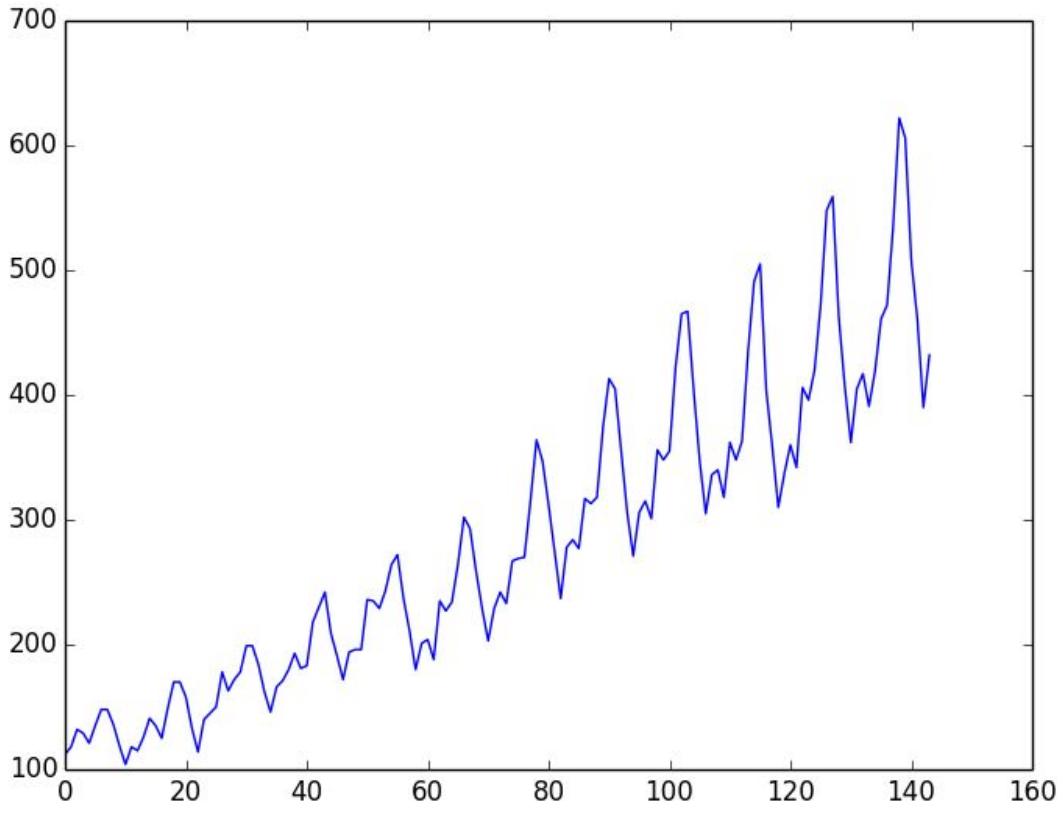
This tutorial includes both!

We will examine the dataset, and then attempt 3 different approaches to the problem: linear regression, feedforward neural networks, and recurrent neural networks.

Data Description

The data is from a popular dataset called the airline passengers dataset. The dataset consists of monthly totals of airline passengers from January 1949 to December 1960. There are 144 data points in total. The number in the dataset is given in thousands, although we will normalize our data anyway.

This is a plot of the data (using plain integers on the x-axis):



As you can see, there are multiple trends here.

The first is that there is an overall increase in number of passengers over time.

The second is that there is a periodic pattern, most likely corresponding to summer vacations.

Note that the amplitude of the cycle increases over time.

Because these patterns are obvious, one could model the series as:

$$\hat{y}(t) = b + at + A(t)cos(\alpha + \omega t)$$
$$A(t) = \gamma t + \delta$$

And that would be another example of “feature engineering”.

But we won’t.

You can download the data yourself from https://datamarket.com/data/set/22u3/international-airline-passengers-monthly-totals-in-thousands-jan-49-dec-60.

I’ve also included it in the repo:

https://github.com/lazyprogrammer/machine_learning_examples

Folder: airline

To load the data, we will use Pandas:

```
import pandas as pd
```

If you look at the CSV, you’ll notice that there are 3 lines at the bottom that are irrelevant. You could delete these manually, but Pandas’ read_csv function includes parameters that allow us to skip footer rows. It is only supported by the “Python” engine, so we will need to specify that as well (the default engine is “C”, which is faster).

```
df = pd.read_csv('international-airline-passengers.csv',
engine='python', skipfooter=3)
```

The column names are a little crazy so I’ve renamed them:

```
df.columns = ['month', 'num_passengers']
```

And then you can plot the data like so:

```
plt.plot(df.num_passengers)
plt.show()
```

To ensure that we train and test our model in a fair way, we are going to split the data down the middle in time into train and test sets.

Typically, we want our models to be trained on all the possible inputs it could see, so that it has a target to learn from in every “area” of the input space.

Ex. If we trained on X=1..10 and then tried to make a prediction for X=100, that would be a major extrapolation. We would most likely be wrong.

On the other hand, if we had training data for X=1,2,3,4,5, and then tried to make a prediction for X=2.5, we could probably be more confident in the answer, since it is close to our training data.

With the airline passenger data, this could potentially be problematic.

Notice how at the halfway point, things start to really pick up. The amplitude of the periodic wave increases by a lot, as does the average count.

However, splitting the data like this is the most “fair” because in real life, we want to predict the future. If it’s currently October, we can’t get the results for December and create a model that accurately predicts November.

Linear Regression

Our first attempt at modeling the data will make use of linear regression.

Let us be clear about what the inputs and outputs (targets) are.

I want to be able to use past passenger counts to predict future passenger counts.

In particular, I want to predict the passenger count x(t) using x(t-1), x(t-2), etc.

I will not use the month or year, as that would allow me to learn the trends I described in the previous section.

Using linear regression, this model is:

$$x(t) = w_0 + w_1 x(t-1) + w_2 x(t-2) + w_3 x(t-3)$$

For predicting x(t) with 3 past data points.

We have a special name for such a model. It is called the “autoregressive” (AR) model.

It’s “regressive” because we are doing regression, and it’s “auto” because we are using the series to predict itself.

As I always try to teach my students, it doesn’t matter much “what” the data is. We just want to mold it into our usual problem:

An NxD matrix of inputs called X and and N-length vector called Y.

Suppose we are given the data c(1), c(2), …, c(10). I’m using the letter “c” here to represent the “count”, to differentiate between X, which is my data matrix into the linear regression model.

My training data would then become:

X1	X2	X3	Y
c(1)	c(2)	c(3)	c(4)
c(2)	c(3)	c(4)	c(5)

c(3)	c(4)	c(5)	c(6)
c(4)	c(5)	c(6)	c(7)
c(5)	c(6)	c(7)	c(8)
c(6)	c(7)	c(8)	c(9)
c(7)	c(8)	c(9)	c(10)

Notice that X is of size 7x3. There can only be 7 data points because the first one we can predict that makes use of 3 inputs is c(4), and the last one we can predict that exists is c(10).

We can put this into code as follows:

```
series = df.num_passengers.as_matrix()
N = len(series)
n = N - D
X = np.zeros((n, D))
for d in xrange(D):
  X[:,d] = series[d:d+n]
Y = series[D:D+n]
```

In the above code, D is the number of past data points we want to use to make the prediction. In the final code, we will loop through various settings of D.

Split the data into train and test sets:

```
Xtrain = X[:n/2]
Ytrain = Y[:n/2]
Xtest = X[n/2:]
Ytest = Y[n/2:]
```

Train a model and print the train and test scores (the R^2, since this is regression):

```
model = LinearRegression()
model.fit(Xtrain, Ytrain)
print "train score:", model.score(Xtrain, Ytrain)
print "test score:", model.score(Xtest, Ytest)
```

Note that we could have implemented linear regression ourselves - both the fit and predict functions would only be 1 line each. We are just saving ourselves a little trouble by using Sci-Kit Learn.

Finally, we want to plot the target data along with our model predictions.

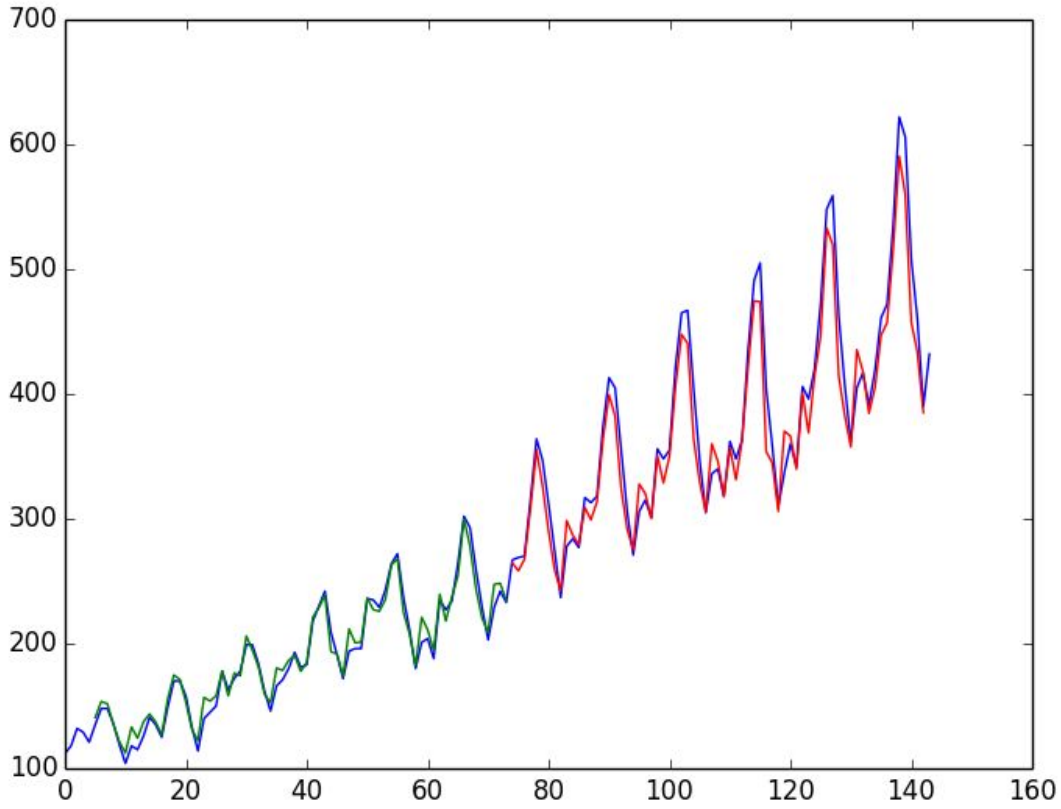
```
plt.plot(series)

train_series = np.empty(n)
train_series[:n/2] = model.predict(Xtrain)
train_series[n/2:] = np.nan
# prepend d nan's since the train series is only of size N - D
plt.plot(np.concatenate([np.full(d, np.nan), train_series]))

test_series = np.empty(n)
test_series[:n/2] = np.nan
test_series[n/2:] = model.predict(Xtest)
plt.plot(np.concatenate([np.full(d, np.nan), test_series]))

plt.show()
```

Lining up the predictions is a little complicated. The full series is of size N, where N = n + D.

Using np.nan means nothing shows up in the plot for that point.

The first d points are nan's since they don't have predictions. The next n/2 points are train predictions. For the test series these should all be nan's. The final n/2 points are test predictions. For the train series these should all be nan's. This ensures that the train and test predictions will show up in different colors.

All the plots should look something like this:



For the final setting of D=7, we achieve:

train score: 0.850075979734
test score: 0.769876100967

Not bad! The simple linear regression model manages to successfully extrapolate the trend in the latter half of the data.

The full code can be found in lr.py.

Feedforward Neural Network

Remember, anything we can plug into a linear regression or logistic regression model, we can also plug into a feedforward neural network.

The training data remains the same: X (NxD) and Y (N)

The only thing that changes when we're doing regression is that we use the squared error instead of the cross-entropy error.

We also don't use the sigmoid at the end since that would constrain our output to be between 0 and 1.

Recall that one way of viewing the feedforward neural network is that it's just a bunch of logistic units stacked together (if you used a sigmoid as your nonlinearity).

Another way to think of it is:

tanh layer → tanh layer → … → logistic regression layer

This is because the last layer is sigmoid(previous_layer.dot(W) + b), which is exactly logistic regression.

For regression problems, we just make the last layer a linear regression layer.

tanh layer → tanh layer → … → linear regression layer

Note that we don't need to use tanh, we can still use ReLU or sigmoid in the hidden layers.

Assuming you already know how a neural network is implemented (otherwise, you would probably not be reading this), the main changes are:

```
thY = T.vector('Y')
```

The targets are no longer integer indexes but rather floats.

```
Yhat = self.forward(thX)
```

We no longer take the argmax to get the prediction because we are not predicting classes.

```
cost = T.mean((thY - Yhat).dot(thY - Yhat))
```

The cost is the squared error.

Note that because the dynamic range of tanh() is small, I've scaled the training data so that the maximum value is 1 and the minimum value is 0.

```
series = series.astype(np.float32)
series = series - series.min()
series = series / series.max()
```

The neural network is very sensitive to hyperparameters, and I would encourage you to try to find better ones.

You may find that the neural network manages to learn the training data just fine, but thinks the maximum value it saw during training is the maximum value forever, and so there is a plateau in the test predictions.

You may also find that in the test predictions, the trend goes downward!

I stopped trying to optimize the hyperparameters when they were comparable to the linear regression results.

The full code can be found in ann.py. I've included it here for reference:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import theano
import theano.tensor as T
from sklearn.utils import shuffle

def init_weight(M1, M2):
    return np.random.randn(M1, M2) / np.sqrt(M1 + M2)

def myr2(T, Y):
    Ym = T.mean()
    sse = (T - Y).dot(T - Y)
    sst = (T - Ym).dot(T - Ym)
    return 1 - sse / sst

class HiddenLayer(object):
    def __init__(self, M1, M2, f, an_id):
        self.id = an_id
```

```
        self.M1 = M1
        self.M2 = M2
        self.f = f
        W = init_weight(M1, M2)
        b = np.zeros(M2)
        self.W = theano.shared(W, 'W_%s' % self.id)
        self.b = theano.shared(b, 'b_%s' % self.id)
        self.params = [self.W, self.b]

    def forward(self, X):
        return self.f(X.dot(self.W) + self.b)


class ANN(object):
    def __init__(self, hidden_layer_sizes):
        self.hidden_layer_sizes = hidden_layer_sizes

    def fit(self, X, Y, activation=T.tanh, learning_rate=10e-4,
mu=0.5, reg=0, epochs=5000, batch_sz=None, print_period=100,
show_fig=True):
        X = X.astype(np.float32)
        Y = Y.astype(np.float32)

        # initialize hidden layers
        N, D = X.shape
        self.hidden_layers = []
        M1 = D
        count = 0
        for M2 in self.hidden_layer_sizes:
            h = HiddenLayer(M1, M2, activation, count)
            self.hidden_layers.append(h)
            M1 = M2
            count += 1
        W = np.random.randn(M1) / np.sqrt(M1)
        b = 0.0
        self.W = theano.shared(W, 'W_last')
        self.b = theano.shared(b, 'b_last')

        if batch_sz is None:
            batch_sz = N

        # collect params for later use
        self.params = [self.W, self.b]
```

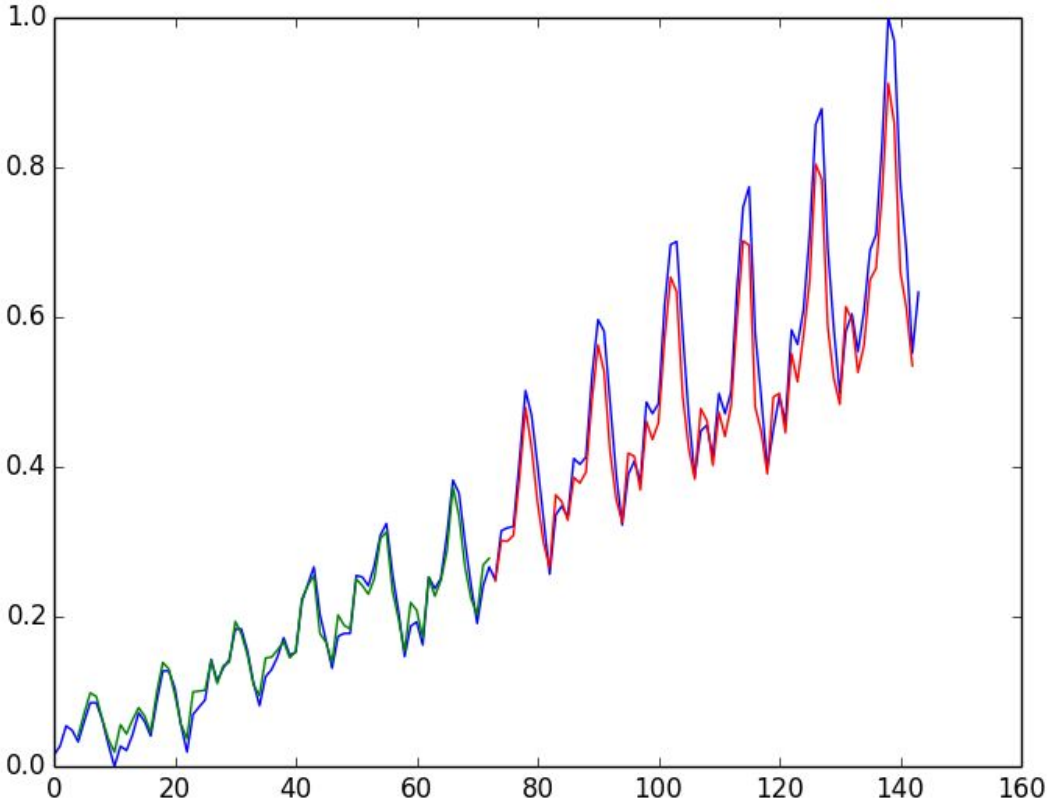
```
        for h in self.hidden_layers:
            self.params += h.params

        # for momentum
        dparams = [theano.shared(np.zeros(p.get_value().shape)) for p
in self.params]

        # set up theano functions and variables
        thX = T.matrix('X')
        thY = T.vector('Y')
        Yhat = self.forward(thX)

        rcost = reg*T.mean([(p*p).sum() for p in self.params])
        cost = T.mean((thY - Yhat).dot(thY - Yhat)) + rcost
        prediction = self.forward(thX)
        grads = T.grad(cost, self.params)

        # momentum only
        updates = [
            (p, p + mu*dp - learning_rate*g) for p, dp, g in
zip(self.params, dparams, grads)
        ] + [
            (dp, mu*dp - learning_rate*g) for dp, g in zip(dparams,
grads)
        ]

        train_op = theano.function(
            inputs=[thX, thY],
            outputs=[cost, prediction],
            updates=updates,
        )

        self.predict_op = theano.function(
            inputs=[thX],
            outputs=prediction,
        )

        n_batches = N / batch_sz
        # print "N:", N, "batch_sz:", batch_sz
        # print "n_batches:", n_batches
        costs = []
        for i in xrange(epochs):
            X, Y = shuffle(X, Y)
```

```
            for j in xrange(n_batches):
                Xbatch = X[j*batch_sz:(j*batch_sz+batch_sz)]
                Ybatch = Y[j*batch_sz:(j*batch_sz+batch_sz)]

                c, p = train_op(Xbatch, Ybatch)
                costs.append(c)
                if (j+1) % print_period == 0:
                    print "i:", i, "j:", j, "nb:", n_batches,
"cost:", c

        if show_fig:
            plt.plot(costs)
            plt.show()

    def forward(self, X):
        Z = X
        for h in self.hidden_layers:
            Z = h.forward(Z)
        return Z.dot(self.W) + self.b

    def score(self, X, Y):
        Yhat = self.predict_op(X)
        return myr2(Y, Yhat)

    def predict(self, X):
        return self.predict_op(X)

# we need to skip the 3 footer rows
# skipfooter does not work with the default engine, 'c'
# so we need to explicitly set it to 'python'
df = pd.read_csv('international-airline-passengers.csv',
engine='python', skipfooter=3)

# rename the columns because they are ridiculous
df.columns = ['month', 'num_passengers']

# plot the data so we know what it looks like
# plt.plot(df.num_passengers)
# plt.show()

# let's try with only the time series itself
series = df.num_passengers.as_matrix()
```

```
# series = (series - series.mean()) / series.std() # normalize the
values so they have mean 0 and variance 1
series = series.astype(np.float32)
series = series - series.min()
series = series / series.max()

# let's see if we can use D past values to predict the next value
N = len(series)
for D in (2,3,4,5,6,7):
    n = N - D
    X = np.zeros((n, D))
    for d in xrange(D):
        X[:,d] = series[d:d+n]
    Y = series[D:D+n]

    print "series length:", n
    Xtrain = X[:n/2]
    Ytrain = Y[:n/2]
    Xtest = X[n/2:]
    Ytest = Y[n/2:]

    model = ANN([200])
    model.fit(Xtrain, Ytrain, activation=T.tanh)
    print "train score:", model.score(Xtrain, Ytrain)
    print "test score:", model.score(Xtest, Ytest)

    # plot the prediction with true values
    plt.plot(series)

    train_series = np.empty(n)
    train_series[:n/2] = model.predict(Xtrain)
    train_series[n/2:] = np.nan
    # prepend d nan's since the train series is only of size N - D
    plt.plot(np.concatenate([np.full(d, np.nan), train_series]))

    test_series = np.empty(n)
    test_series[:n/2] = np.nan
    test_series[n/2:] = model.predict(Xtest)
    plt.plot(np.concatenate([np.full(d, np.nan), test_series]))

    plt.show()
```

The results:

train score: 0.845147040812
test score: 0.738297290069



Recurrent Neural Network (GRU/LSTM)

The last technique we'll try is the recurrent neural network. Recurrent neural networks are suited for modeling sequences, and our data is a sequence of passenger counts at monthly intervals. Perfect!

Some people have asked me questions about modifying the RNN class code, since a lot of it is specifically geared toward language modeling.

But if you think about it, both language modeling and the autoregressive models from earlier both attempt to calculate p(x(t) | x(t-1), x(t-2), …).

The main difference is that you don't have a word embedding matrix, because your inputs are not word indexes, and your outputs are also not word indexes, so you'll use the squared error once again (with a linear regression layer at the end of the RNN).

Recall that we already wrote plug-and-playable LSTM and GRU units in the RNN class, so we can make use of those (and switch them out of our new RNN seamlessly):

```
import os
import sys
sys.path.append(os.path.abspath('..'))
from rnn_class.lstm import LSTM
from rnn_class.gru import GRU
```

The architecture is:

X → (Wx) → [GRU] → (Wo) → Y

Where Wx is of size DxM, the GRU's state is a vector of size M, and Wo is also a vector of size M, since Y is a scalar.

Quiz question: What is the size of D?

In our linear regression and feedforward neural network examples, D was the number of past inputs to use to calculate the output.

But now that we are working with sequences, what do we have? We have a sequence of length T, and each element of the sequence is a count (scalar).

Therefore, our data matrix (as with all other recurrent neural networks) is of size NxTxD, where D = 1.

This is similar to how we worked with images when we used logistic regression and feedforward neural networks. Images are matrices (2-D objects) because they have width and height. Because the input must be a vector, we flattened the matrix so that D = width x height.

Another way of interpreting that is you are ignoring the structure of the data and treating all the inputs equally. This is different from convolutional neural networks where pixels being near each other has meaning.

Similarly, each past count is treated the same way (multiplied by a scalar weight) in the linear regression model.

With recurrent nets, order is taken into account. x(t-3) affects h(t-3). x(t-2) and h(t-3) affect h(t-2).

You should attempt to play with the hyperparameters (hidden state size, number of hidden layers, learning rate, momentum, etc.) to see if you can get better results than me. I stopped when I reached results comparable to linear regression.

You will notice that the RNN can fail in the same way as the ANN (plateauing or even decreasing during the test stage).

For reference, the code is here (I've bolded the important changes):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import theano
import theano.tensor as T
from sklearn.utils import shuffle
from datetime import datetime

import os
import sys
sys.path.append(os.path.abspath('..'))
from rnn_class.lstm import LSTM
from rnn_class.gru import GRU

def init_weight(M1, M2):
    return np.random.randn(M1, M2) / np.sqrt(M1 + M2)
```

```
def myr2(T, Y):
    Ym = T.mean()
    sse = (T - Y).dot(T - Y)
    sst = (T - Ym).dot(T - Ym)
    return 1 - sse / sst

class RNN(object):
    def __init__(self, hidden_layer_sizes):
        self.hidden_layer_sizes = hidden_layer_sizes

       def  fit(self,  X,  Y,  activation=T.tanh,  learning_rate=10e-2,
mu=0.5, reg=0, epochs=2000, show_fig=False):
        N, t, D = X.shape

        self.hidden_layers = []
        Mi = D
        for Mo in self.hidden_layer_sizes:
            ru = GRU(Mi, Mo, activation)
            self.hidden_layers.append(ru)
            Mi = Mo

        Wo = np.random.randn(Mi) / np.sqrt(Mi)
        bo = 0.0
        self.Wo = theano.shared(Wo)
        self.bo = theano.shared(bo)
        self.params = [self.Wo, self.bo]
        for ru in self.hidden_layers:
            self.params += ru.params

        lr = T.scalar('lr')
        thX = T.matrix('X')
        thY = T.scalar('Y')
        Yhat = self.forward(thX)[-1]

        # let's return py_x too so we can draw a sample instead
        self.predict_op = theano.function(
            inputs=[thX],
            outputs=Yhat,
            allow_input_downcast=True,
        )

        cost = T.mean((thY - Yhat)*(thY - Yhat))
        grads = T.grad(cost, self.params)
```

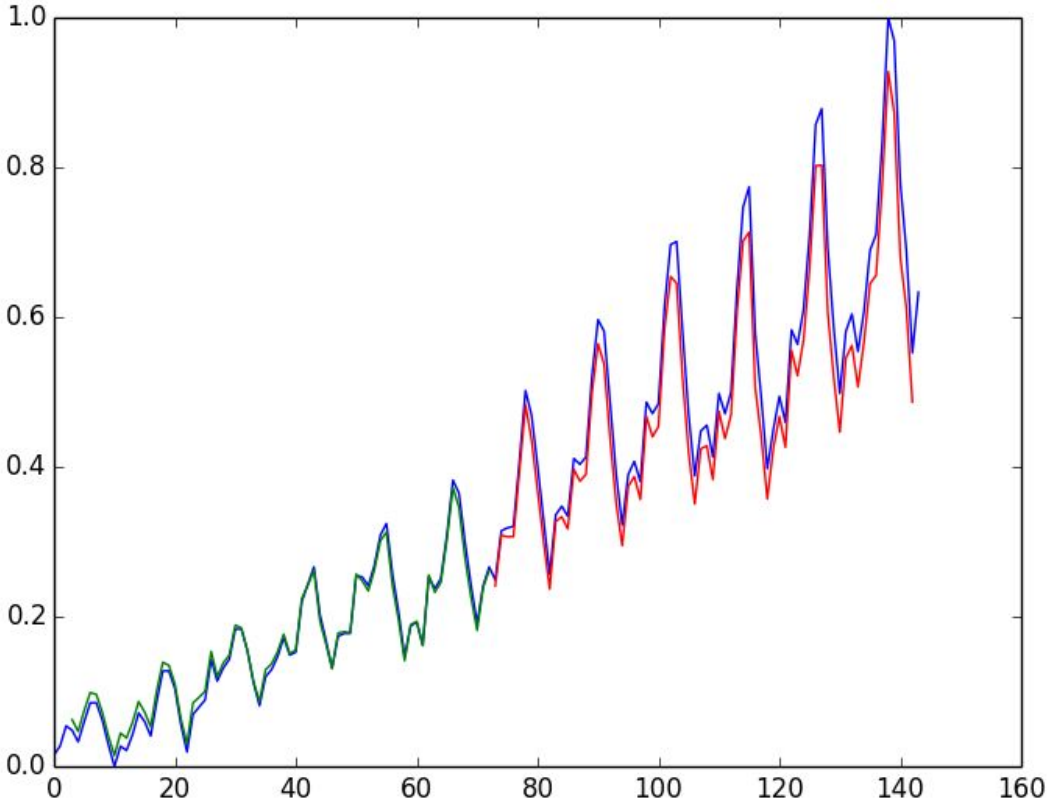
```
                dparams = [theano.shared(p.get_value()*0) for p in
self.params]

        updates = [
              (p, p + mu*dp - lr*g) for p, dp, g in zip(self.params,
dparams, grads)
        ] + [
            (dp, mu*dp - lr*g) for dp, g in zip(dparams, grads)
        ]

        self.train_op = theano.function(
            inputs=[lr, thX, thY],
            outputs=cost,
            updates=updates
        )

        costs = []
        for i in xrange(epochs):
            t0 = datetime.now()
            X, Y = shuffle(X, Y)
            n_correct = 0
            n_total = 0
            cost = 0
            for j in xrange(N):

                c = self.train_op(learning_rate, X[j], Y[j])
                cost += c
            if i % 10 == 0:
                    print "i:", i, "cost:", cost, "time for epoch:",
(datetime.now() - t0)
            if (i+1) % 500 == 0:
                learning_rate /= 10
            costs.append(cost)

        if show_fig:
            plt.plot(costs)
            plt.show()

    def forward(self, X):
        Z = X
        for h in self.hidden_layers:
            Z = h.output(Z)
        return Z.dot(self.Wo) + self.bo
```

```
    def score(self, X, Y):
        Yhat = self.predict(X)
        return myr2(Y, Yhat)

    def predict(self, X):
        N = len(X)
        Yhat = np.empty(N)
        for i in xrange(N):
            Yhat[i] = self.predict_op(X[i])
        return Yhat
```

The loading of the data and creation of the model are exactly the same as with the ANN by design.

Here are the results from the RNN:

train score: 0.841706904334
test score: 0.697168543555


1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
120
140
160

Discussion

While everyone is excited about RNNs these days, here are a few important things to consider.

Which of the models led to the best results?

Linear regression!

How many parameters did each of the models use?

Linear regression: 3
ANN: 3x200 (input weight) + 200 (input bias) + 200 (output weight) + 1 (output bias)
RNN: As an exercise, calculate this yourself.

Which model has the most hyperparameters?

The RNN/ANN have similar numbers of hyperparameters, but the linear regression model contains only 1 - we just need to specify the number of past counts to use.

And we already saw that the number of past counts doesn't have a huge effect on the score, linear regression does pretty well regardless.

Compare that to the RNN/ANN, which, if you tried to modify the hyperparameters as I recommended, you would have seen that it is very easy to obtain very poor results.

Which model takes the longest to train?

In order: RNN with LSTM > RNN with GRU > ANN > Linear Regression.

So why do I always say "try the simplest model first"? Now you know!

I hope you enjoyed this tutorial.

I will be releasing more in the future, and I always do new releases first by newsletter, so make sure you sign up at https://lazyprogrammer.me