BRIAN DOLHANSKY

ml • code • photography

home
blog
research
cv
photography

Artificial Neural Networks: Linear Regression (Part 1)

July 10, 2013 in ml primers, neural networks

Artificial neural networks (ANNs) were originally devised in the mid-20th century as a computational model of the human brain. Their used waned because of the limited computational power available at the time, and some theoretical issues that weren't solved for several decades (which I will detail at the end of this post). However, they have experienced a resurgence with the recent interest and hype surrounding Deep Learning. One of the more famous examples of Deep Learning is the "Youtube Cat" paper by Andrew Ng et al.

It is theorized that because of their biological inspiration, ANN-based learners will be able to emulate how a human learns to recognize concepts or objects without the time-consuming feature engineering step. Whether or not this is true (or even provides an advantage in terms of development time) remains to be seen, but currently it's important that we machine learning researchers and enthusiasts have a familiarity with the basic concepts of neural networks.

This post covers the basics of ANNs, namely single-layer networks. We will cover three applications: linear regression, two-class classification using the perceptron algorithm and multi-class classification.

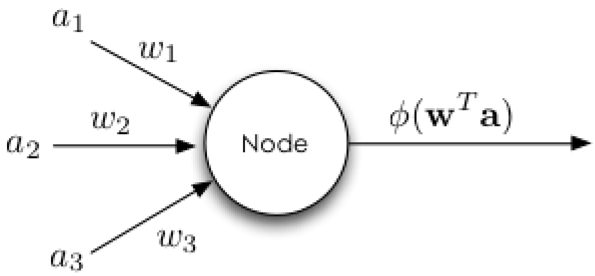
Theory

Neural network terminology is inspired by the biological operations of specialized cells called neurons. A neuron is a cell that has several inputs that can be activated by some outside process. Depending on the amount of activation, the neuron produces its own activity and sends this along its outputs. In addition, specific input or output paths may be "strengthened" or weighted higher than other paths. The hypothesis is that since the human brain is nothing but a network of neurons, we can emulate the brain by modeling a neuron and connecting them via a weighted graph.

The artificial equivalent of a neuron is a *node* (also sometimes called neurons, but I will refer to them as nodes to avoid ambiguity) that receives a set of weighted inputs, processes their sum with its *activation function* ϕ, and passes the result of the activation function to nodes further down the graph. Note that it is simpler to represent the input to our activation function as a dot product:

$$\phi\left(\sum_i w_i a_i\right) = \phi(\mathbf{w}^T\mathbf{a})$$

Visually this looks like the following:



There are several canonical activation functions. For instance, we can use a *linear* activation function:

$$\phi(\mathbf{w}^T\mathbf{a}) = \mathbf{w}^T\mathbf{a}$$

This is also called the *identity* activation function. Another example is the *sigmoid* activation function:

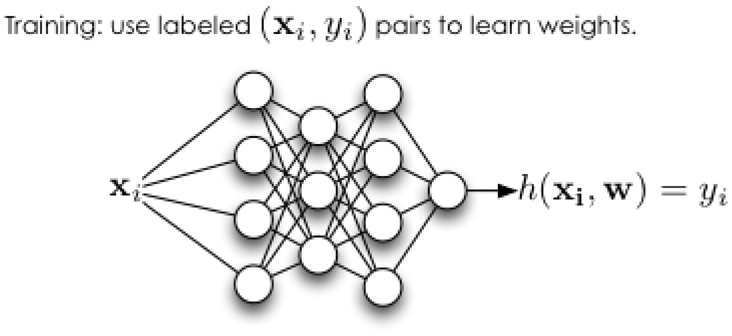
$$\phi(\mathbf{w}^T\mathbf{a}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{a})}$$

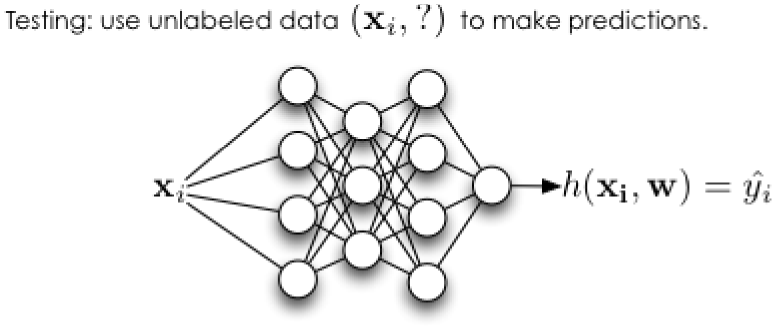
One more example is the *tanh* activation function:

$$\phi(\mathbf{w}^T\mathbf{a}) = \tanh(\mathbf{w}^T\mathbf{a})$$

We can then form a network by chaining these nodes together. Usually this is done in layers - one node layer's outputs are connected to the next layer's inputs (we must take care not to introduce cycles in our network, for reasons that will become clear in the section on backpropagation).

Our goal is to train a network using labelled data so that we can then feed it a set of inputs and it produces the appropriate outputs for unlabeled data. We can do this because we have both the input $\mathbf{x}_i$ and the desired target output y_i in the form of data pairs. Training in this case involves learning the correct edge weights to produce the target output given the input. The network and its trained weights form a function (denoted h) that operates on input data. With the trained network, we can make predictions given any unlabeled test input.


Training: use labeled $(\mathbf{x}_i, y_i)$ pairs to learn weights.
$\mathbf{x}_i$
$h(\mathbf{x_i}, \mathbf{w}) = y_i$


Testing: use unlabeled data $(\mathbf{x}_i, ?)$ to make predictions.
$\mathbf{x}_i$
$h(\mathbf{x_i}, \mathbf{w}) = \hat{y}_i$

Training and testing in the neural network context. Note that a multilayer network is shown here. Training a multilayer network is covered in Parts 3-6 of this primer.

We can train a neural network to perform regression *or* classification. In this part, I will cover linear regression with a single-layer network. Classification and multilayer networks are covered in later parts.

Linear Regression

Linear regression is the simplest form of regression. We model our system with a linear combination of features to produce one output. That is,

$$y_i = h(\mathbf{x}_i, \mathbf{w}) = \mathbf{w}^T \mathbf{x}_i$$

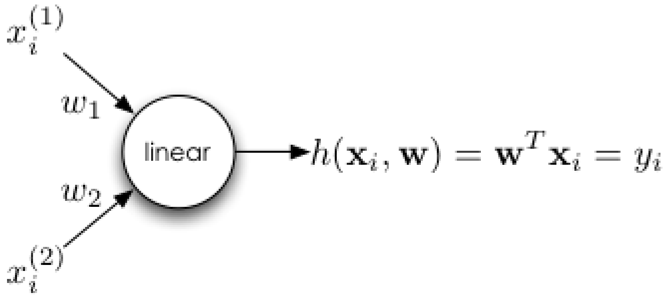
Our task is then to find the weights the provide the best fit to our training data. One way to measure our fit is to calculate the leasts squares error (or *loss*) over our dataset:

$$L(\mathbf{w}) = \sum_i \left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2$$

In order to find the line of best fit, we must minimize $L(\mathbf{w})$. This has a closed-form solution for ordinary least squares, but in general we can minimize loss using gradient descent.

Training a neural network to perform linear regression

So what does this have to do with neural networks? In fact, the simplest neural network performs least squares regression. Consider the following single-layer neural network, with a single node that uses a linear activation function:


$x_i^{(1)}$
w_1
linear
$h(\mathbf{x}_i, \mathbf{w}) = \mathbf{w}^T \mathbf{x}_i = y_i$
w_2
$x_i^{(2)}$

This network takes as input a data point with two features $x_i^{(1)}, x_i^{(2)}$, weights the features with w_1, w_2 and sums them, and outputs a prediction . We could define a network that takes data with more features, but we would have to keep track of more weights, e.g. $w_1, \dots, w_j$ if there are j features.

If we use quadratic loss to measure how well our network performs, (quadratic loss is a common choice for neural networks), it would be identical to the loss defined for least squares regression above:

$$L(\mathbf{w}) = \sum_i \left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2$$

This is the sum squared error of our network's predictions over our entire training set.

We will then use gradient descent on the loss's gradient $\nabla_{\mathbf{w}} L(\mathbf{w})$ in order to minimize the overall error on the training data. We first derive the gradient of the loss with respect to a particular weight $w_{j \to k}$ (which is just the weight of the edge connecting node j to node k [note that we treat inputs as "nodes," so there is a weight $w_{j \to k}$ for each connection from the input to a first-layer node]) in the general case:

$$\begin{aligned}\frac{\partial}{\partial w_{j \to k}} L(\mathbf{w}) &= \frac{\partial}{\partial w_{j \to k}} \sum_i \left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2 \\ &= \sum_i \frac{\partial}{\partial w_{j \to k}} \left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2 \\ &= \sum_i 2\left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right) \frac{\partial}{\partial w_{j \to k}} h(\mathbf{x}_i, \mathbf{w})\end{aligned}$$

At this point, we must compute the gradient of our network function with respect to the weight in question ($\frac{\partial}{\partial w_{j \to k}} h(\mathbf{x}_i, \mathbf{w})$). In the case of a single layer network, this turns out to be simple.

Recall our simple two input network above. The network function is $h(\mathbf{x}_i, \mathbf{w}) = w_1 x_i^{(1)} + w_2 x_i^{(2)}$. The gradient with respect to w_1 is just x_1, and the gradient with respect to w_2 is just x_2. We usually store all the weights of our network in a vector or a matrix, so the full gradient is:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \left(\frac{\partial L(\mathbf{w})}{\partial w_1}, \frac{\partial L(\mathbf{w})}{\partial w_2} \right) = \left(\sum_i 2x_i^{(1)} h(\mathbf{x}_i, \mathbf{w}), \sum_i 2x_i^{(2)} h(\mathbf{x}_i, \mathbf{w}) \right)$$

Using this, we then update our weights using standard gradient descent:

$$\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w})$$

As with all gradient descent methods, care must be taken to select the "right" step size η, with "right" being application-dependent. After a set amount of epochs, the weights we end up with define a line of best-fit.

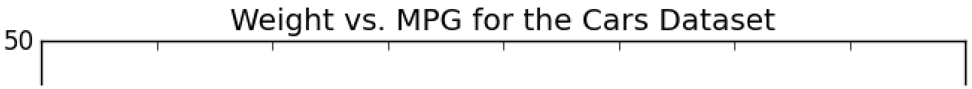
Testing

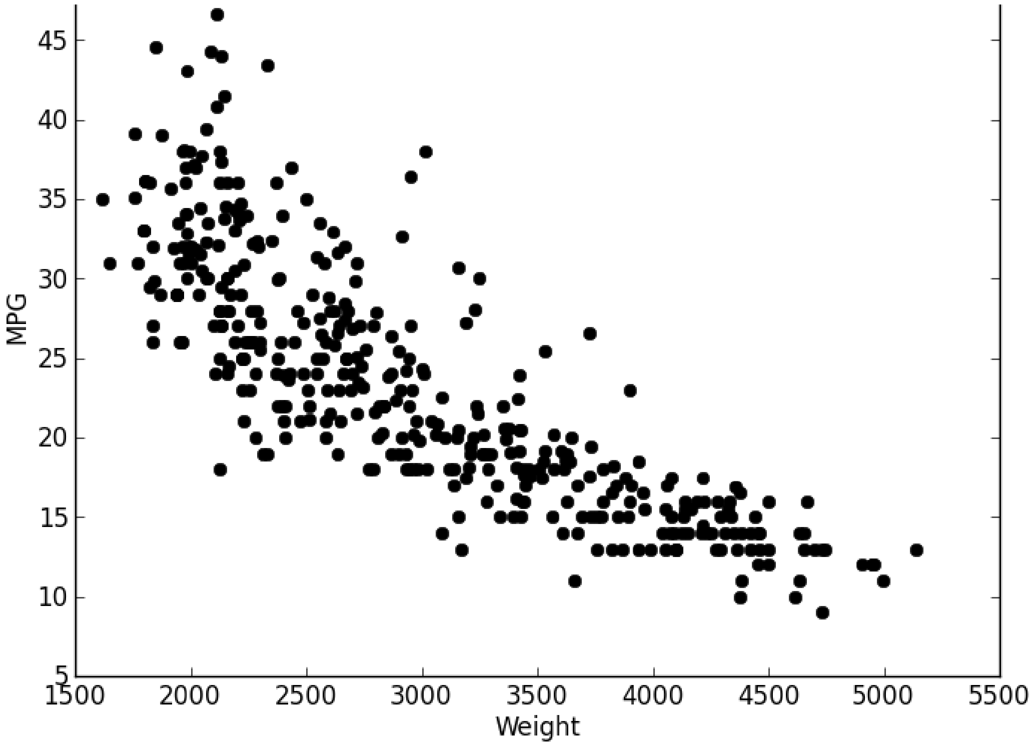
With our trained network, testing consists of obtaining a prediction for each test point x_i using $h(\mathbf{x}_i, \mathbf{w})$. The test error is computed with the quadratic loss, exactly as in training:

$$L(\mathbf{w}) = \sum_i \left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2 = \sum_i \left(\hat{y}_i - y_i\right)^2$$

Implementation

For this implementation, we will use the weight of a car to predict its MPG. The data looks something like this:





Note that this relationship does not appear to be linear - linear regression will probably not find the underlying relationship between weight and MPG. However, it *will* find a line that models the data "pretty well."

As with my other tutorials, I will be using Python with numpy (for matrix math operations) and matplotlib (for plotting). (All the code listed here is located in the file `ann_linear_1D_regression.py`). To begin, let's first load the MPG data from `mpg.csv`:

```
X_file = np.genfromtxt('mpg.csv', delimiter=',', skip_header=1)
N = np.shape(X_file)[0]
X = np.hstack((np.ones(N).reshape(N, 1), X_file[:, 4].reshape(N, 1)))
Y = X_file[:, 0]
```

This loads our data into two matrices, $ X $ (containing the features, the weight) and Y (containing the labels). You may haven noticed something odd - we are also appending a column of ones to X. It is important to have bias weights in our neural network - otherwise, we could only fit functions that pass through 0.

Next, we standardize the input. This is another implementation-specific detail. Although it is not theoretically necessary, it helps provide stability to our gradient descent routine and prevents our weights from quickly "blowing up." We standardize the weight features by subtracting their mean and normalizing by their standard deviation:

```
14 X[:, 1] = (X[:, 1]-np.mean(X[:, 1]))/np.std(X[:, 1])
```

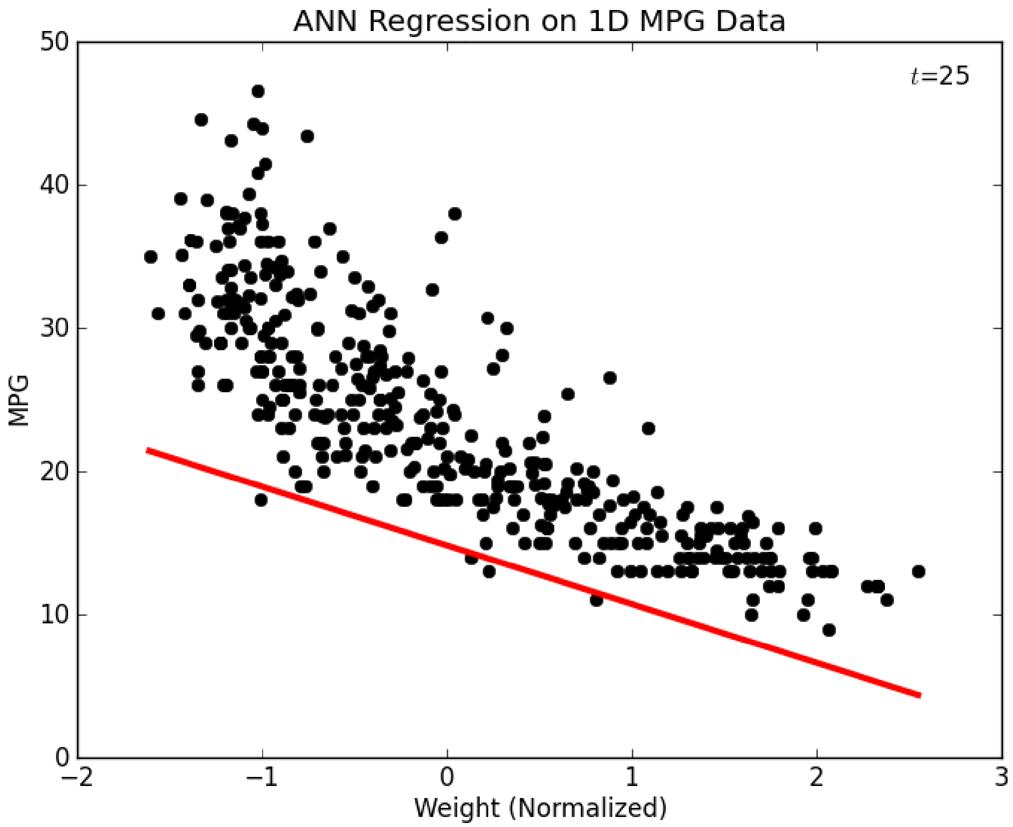
Then we begin gradient descent. We will run batch gradient descent for 100 epochs with a step size $\eta = 0.001$:

```
21 max_iter = 100
22 eta = 1E-3
23 for t in range(0, max_iter):
24     # We need to iterate over each data point for one epoch
25     grad_t = np.array([0., 0.])
26     for i in range(0, N):
27         x_i = X[i, :]
28         y_i = Y[i]
29         # Dot product, computes h(x_i, w)
30         h = np.dot(w, x_i)-y_i
31         grad_t += 2*x_i*h
32 
33     # Update the weights
34     w = w - eta*grad_t
```

In line 30, we compute the network function $h(\mathbf{x}_i, \mathbf{w}) = \mathbf{w}^T\mathbf{x}_i$. In line 31, we compute the actual gradient for both weights simultaneously and add them to the gradient of the current epoch. Then, in line 34 we perform the gradient descent update. Finally, to compute the line of best fit, we use the following:

```
tt = np.linspace(np.min(X[:, 1]), np.max(X[:, 1]), 10)
bf_line = w[0]+w[1]*tt
```

This uses the weights to compute the value of the line with the same domain spanned by our data. Here is an animation of the line of best fit at each epoch, which illustrates how the result of our weight update at each step:



You can download the code and data here. For posterity, here is the complete source file, complete with plotting functionality.

```
import matplotlib.pyplot as plt
import numpy as np

# Load the data and create the data matrices X and Y
# This creates a feature vector X with a column of ones (bias)
# and a column of car weights.
# The target vector Y is a column of MPG values for each car.
X_file = np.genfromtxt('mpg.csv', delimiter=',', skip_header=1)
N = np.shape(X_file)[0]
X = np.hstack((np.ones(N).reshape(N, 1), X_file[:, 4].reshape(N, 1)))
Y = X_file[:, 0]

# Standardize the input
X[:, 1] = (X[:, 1]-np.mean(X[:, 1]))/np.std(X[:, 1])

# There are two weights, the bias weight and the feature weight
w = np.array([0, 0])

# Start batch gradient descent, it will run for max_iter epochs and
have a step
# size eta
max_iter = 100
eta = 1E-3
for t in range(0, max_iter):
    # We need to iterate over each data point for one epoch
    grad_t = np.array([0., 0.])
    for i in range(0, N):
        x_i = X[i, :]
        y_i = Y[i]
        # Dot product, computes h(x_i, w)
        h = np.dot(w, x_i)-y_i
        grad_t += 2*x_i*h

    # Update the weights
    w = w - eta*grad_t
print "Weights found:",w

# Plot the data and best fit line
tt = np.linspace(np.min(X[:, 1]), np.max(X[:, 1]), 10)
bf_line = w[0]+w[1]*tt

plt.plot(X[:, 1], Y, 'kx', tt, bf_line, 'r-')
plt.xlabel('Weight (Normalized)')
plt.ylabel('MPG')
plt.title('ANN Regression on 1D MPG Data')

plt.savefig('mpg.png')

plt.show()
```

Conclusion

In this post, I detailed how to emulate linear regression using a simple neural network. Using a neural network for this task may seem useless, but the concepts covered in this post carry over to more complicated networks.

Several questions remain. What if we want to perform classification? And how do we implement multilayer networks? Stay tuned for more parts in this series.

References

[1] http://www.willamette.edu/~gorr/classes/cs449/intro.html

[2] http://blog.zabarauskas.com/backpropagation-tutorial/

Tags: neural network, tutorial

♥ *18 Likes* *Share*

Prev / Next

Comments (13) Newest First Subscribe via e-mail

Preview Post Comment…

Mark 3 months ago

An excellent overview of ANNs.
The sigmoid function shown in the article is a special case from the family of sigmoid functions, known as the logistic function. This makes me wonder whether other sigmoid functions, apart from the logistic, are used in ANNs?

John 6 months ago

The best of all (among that I read)

Dev 11 months ago

This is by far the best overview that I have come across on ANN. The vector notation and the weight notation makes it very clear and succinct. Cannot wait to read all the parts.

Ravshan A year ago

Thanks for such an amazing overview! Coming from econometrics background it was a pleasure to see both ways of deriving OLS estimators mentioned. A great care for details!

Evgeny Yashin A year ago

Yes, in the formula for ∇w L(w) = .. the - yi part is missing.

Dmitry 2 years ago

Why does the gradient on ∇wL(w) is just h(xi, w)? it should be (h(xi, w) -y), shouldn't it?

Goutham 2 years ago

Really nice tutorial. Very simple to understand.

Jay 2 years ago

The equation that comes before the gradient descent update rule which is \nabla{\mathbf{w}}L(\mathbf{w}) = \left(\frac{\partial L(\mathbf{w})}{\partial w1}, \frac{\partial L(\mathbf{w})}{\partial w2}\right) = \left(\sumi 2xi^{(1)}h(\mathbf{x}i, \mathbf{w}), \sumi 2xi^{(2)}h(\mathbf{x}i, \mathbf{w})\right),
shouldn't it be like
\nabla{\mathbf{w}}L(\mathbf{w}) = \left(\frac{\partial L(\mathbf{w})}{\partial w1}, \frac{\partial L(\mathbf{w})}{\partial w2}\right) = \left(\sumi 2xi^{(1)}(h(\mathbf{x}i, \mathbf{w})-yi), \sumi 2xi^{(2)}(h(\mathbf{x}i, \mathbf{w})-yi)\right)?

JingleSting 4 months ago

I believe there is an error in this equation also. Expected to keep the -yi terms, but this has been missed.

Toyosi 2 years ago

Please, what makes artificial neural network better than simple linear regression?

Brian Dolhansky 2 years ago

A feed forward neural network with no hidden layers can do linear regression, but as soon as you start adding additional layers the network starts performing non-linear regression. Linear regression can be performed with a subset of deep architecture.

K. 3 years ago

This post was great in demonstrating how gradient descent functions minimize error functions until you arrive at the line of best fit. However, most of the examples on the internet involve simple linear regression where x is the predictor of y and you can visualise everything nicely. I'm having a hard time envisioning multiple linear regression where x, z, and w are predictors of y or neural networks where the activation function is non-linear like tanh or logistic (as you have listed in this post) and the explanatory variables are more than one. Should I give up trying to visualise these and really understand them? What do you suggest?

Brian Dolhansky 3 years ago

I can certainly envision the problem where there are 2 features and 1 target value - in this case, you're trying to fit a plane (instead of a line) to data distributed in 3 dimensional space (for instance, x and z are the features in a 3d coordinate system, and y is the target). For more than that, it's hard to visualize. I guess with three features and 1 target (a 4D problem) you could envision trying to fit a plane that moves through time to 3D data that also moves through time. Above that it's pretty tough!

Although it's intuitive, I generally try to avoid trying to spatially visualize how a model is learning from high-dimensional data. Instead I try to focus on what the important features are, and how the target changes in relation to those high-importance features.