Neural Networks

Some of the figures in this presentation are taken from "An Introduction to Statistical Learning, with applications in R, second edition" (Springer, 2021) with permission from the authors:

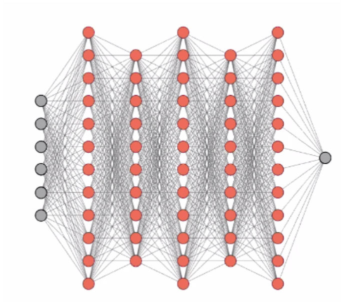
G. James, D. Witten, T. Hastie and R. Tibshirani

Mladen Kolar and Rob McCulloch

1. Introduction and a Single Layer Fit

There is a lot to take in when learning neural nets.

In general, neural net models are composed of *layers* where each layer consists of a set of *units also called neurons*.



Lots of issues to address:

- how do the units in a layer help us to build up interesting functions?
- how do the layers help us to build up interesting functions?
- how do you use neural nets with (i) numeric outcomes, (ii) binary outcomes, (iii) multinoulli outomes
- optimization issues in learning neural nets.
- regularization with L1 and L2 penalties and dropout.

Let's start by understanding a neural net model with a single layer first.
This will help us get a feeling for the first issue above.

After we understand a single layer we can move on to multiple layers.

There will be a lot to learn with just a single layer !!!

In general, you can actually fit just about anything with a single layer.

But we will see that having multiple layers can be are good way to build complex models.

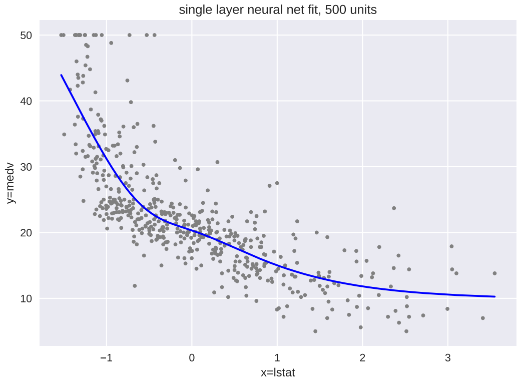
2. Understanding the Basic model, what are units?

Let's use a single layer neural net model to fit y=medv x=lstat.

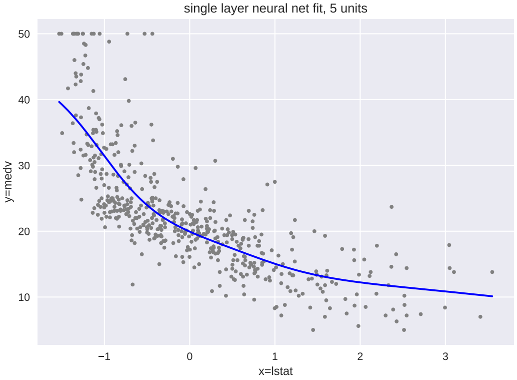
Here is the fit with "500 units".

We use all the data as training data.

The x=lstat was scaled to have mean 0 and variance 1.



Here is the fit with “5 units”.



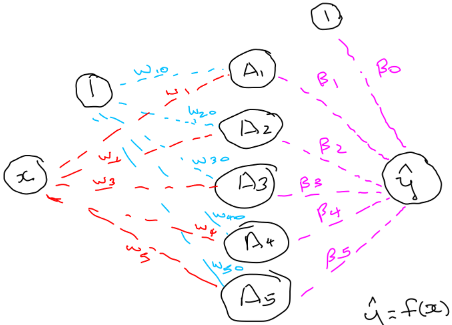
Does not look quite as good as with 500 units, but let’s pull this simpler fit apart to see how it actually works.

Here are the learned parameters.

```
x weights
(1, 5)
[[-1.6415458 -1.3937181 -3.2372243 -0.6942747 -3.719547 ]]


x bias
(5,)
[ 0.9728854  -0.33562386 -3.4881692   3.0414855  -4.071572  ]


output weights
(5, 1)
[[6.4191284]
 [7.04142  ]
 [9.101682 ]
 [6.1671214]
 [9.624342 ]]


output bias
(1,)
[6.1394553]
```

So how do these numbers give us the function on the previous slide ???

x weights are the $w_k, k = 1, 2, 3, 4, 5$.
x bias are the $w_{k0}, k = 1, 2, 3, 4, 5$.
output weights are the $\beta_k, k = 1, 2, 3, 4, 5$.
output bias is $\beta_0, k = 1, 2, 3, 4, 5$.



$$A_k = g(\underbrace{w_{k0} + w_k x}_{a})$$

$$g(a) = \frac{e}{1+e^{a}}$$

$$\hat{y} = f(x) = \beta_0 + \sum_{k=1}^{5} \beta_k A_k$$

K is the number of units.

In our example, $K = 5$.

$$z_k = w_{k0} + w_k\, x, \;\; A_k = g(z_k), \;\; k = 1, 2, \ldots K.$$

$$f(x) = \beta_0 + \sum_{k=1}^{K} \beta_k\, A_k.$$

Or, all in one fell swoop,

$$f(x) = \beta_0 + \sum_{k=1}^{K} \beta_k\, g(w_{k0} + w_k\, x)$$

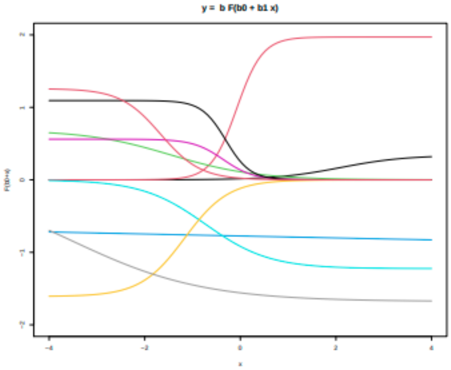
In neural net world the intercepts (β_0, w_{k0}) are called the biases.
The coefficients (β_k, w_k) are called the weights.
g is the *activation function*.

Model:

- make K different linear functions of x, one for each unit.
- put the results of each linear function into a nonlinear activation function giving the *activations*, one for each unit.
- return a linear function of the K activations.

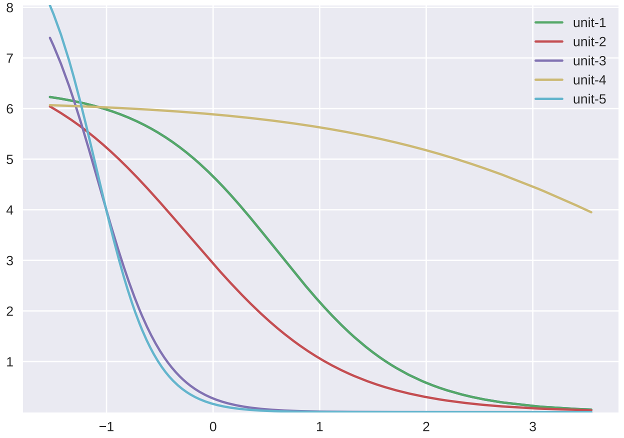
Why is this such a great idea ????

Various functions of the form $\beta\, g(w_0 + w_1\, x)$ for different values of β, w_0, and w_1.

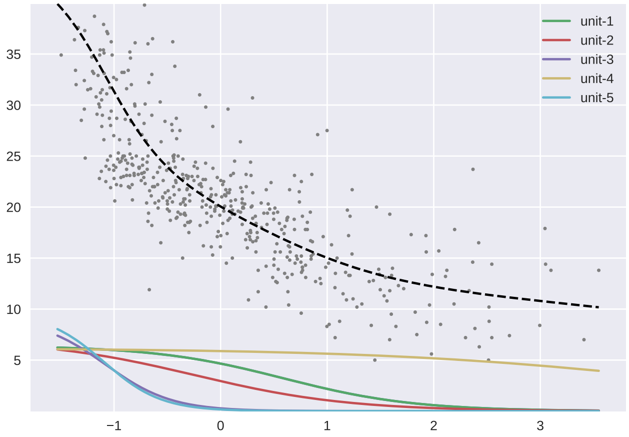


We can get get just about any function we want just by adding up these kinds of functions !!

Here are the plots of x vs $\beta_k A_k$ for each $k = 1, 2, 3, 4, 5$ from our example.

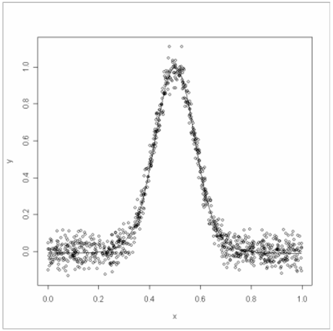


Here are the plots of x vs $\beta_k A_k$ for each $k = 1, 2, 3, 4, 5$ *and* the sum of the pieces with β_0 added on.



Let's try this function.

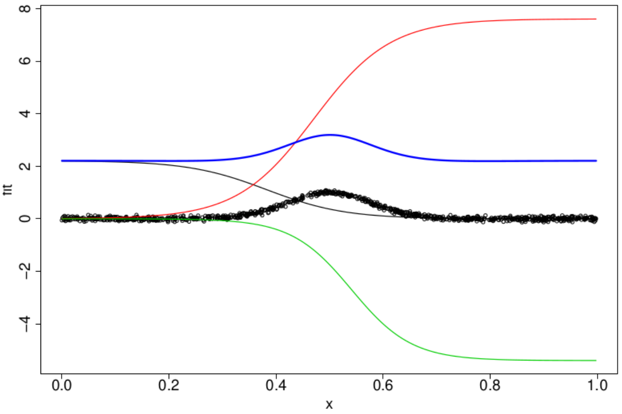
The line drawn through the data is a neural net fit with just three units.



Here are the three pieces of the form $\beta\, g(w_0 + w_1\, x)$.

The blue is the sum of the black, red, and green.

Then we add the constant β_0 to move it down to fit the data.



See how they add up to the bump??

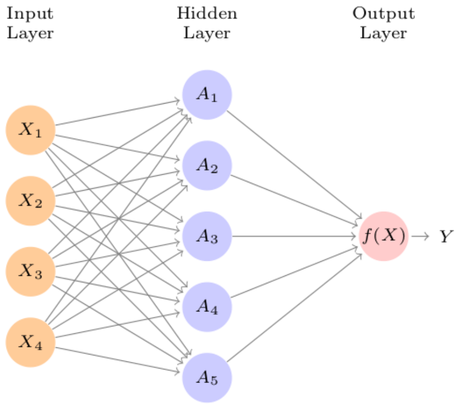
I did this in R with the nnet package.

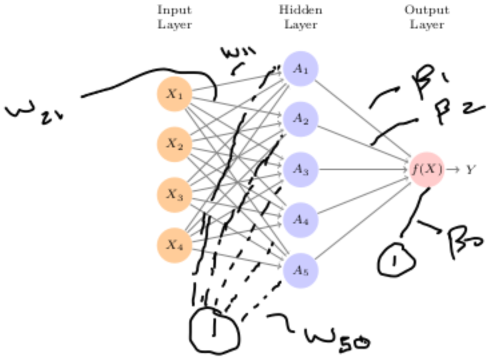
```
F = function(x) {return(exp(x)/(1+exp(x)))}

z1 =  5.26 - 13.74*x
z2 = -6.58 + 13.98*x
z3 = -9.67 + 17.87

f1 = 2.21*F(z1)
f2 = 7.61*F(z2)
f3 = -5.40*F(z3)
```

3. More than one x

How does it work with more than one variable in x?

Just make each unit a linear function of the the vector x.





$$X = (X_1, X_2, \ldots, X_p).$$

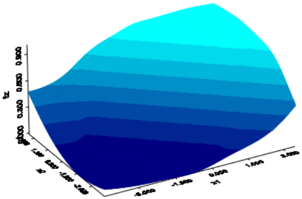
$$z_k = w_{k0} + \sum_{j=1}^{p} w_{kj} X_j, \quad A_k = g(z_k), \quad k = 1, 2, \ldots, K.$$

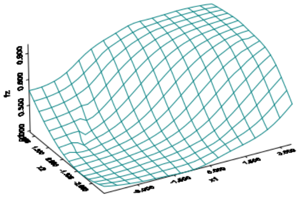
$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k.$$

A function of the form

$$\beta\, g(w_0 + \sum_{j=1}^{p} w_j X_j)$$

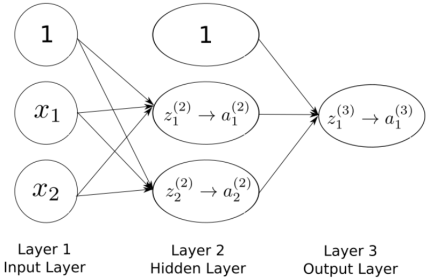
(we dropped k)





Obviously, we can we any function we like by summing up functions like this !!!

Different notation, very simple model.


1
x_1
x_2
1
$z_1^{(2)} \to a_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$
$z_1^{(3)} \to a_1^{(3)}$
Layer 1
Input Layer
Layer 2
Hidden Layer
Layer 3
Output Layer

$$\begin{aligned}
z_1^{(2)} &= b_{10}^{(1)} + b_{11}^{(1)} x_1 + b_{12}^{(1)} x_2 & \longrightarrow \quad & a_1^{(2)} = g(z_1^{(2)}) \\
z_2^{(2)} &= b_{20}^{(1)} + b_{21}^{(1)} x_1 + b_{22}^{(1)} x_2 & \longrightarrow \quad & a_2^{(2)} = g(z_2^{(2)}) \\
z_1^{(3)} &= b_{10}^{(2)} a_0^{(2)} + b_{11}^{(2)} a_1^{(2)} + b_{12}^{(2)} a_2^{(2)} & \longrightarrow \quad & a_1^{(3)} = g(z_1^{(3)})
\end{aligned}$$

g is the *activation function*.

Note

We will be using regularization.

Neural net models have *many* linear functions !!!

As with our basic linear regression model, if helps a lot if we first standardize our features.

Note

The features (X) are the *input layer*.

The final layer is the *output layer*.

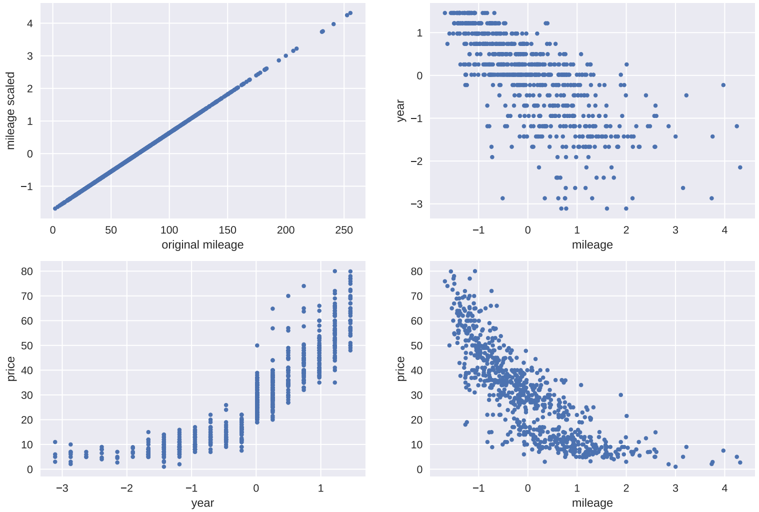
Example: Used Cars with mileage and year

75% - 25% train-test split.

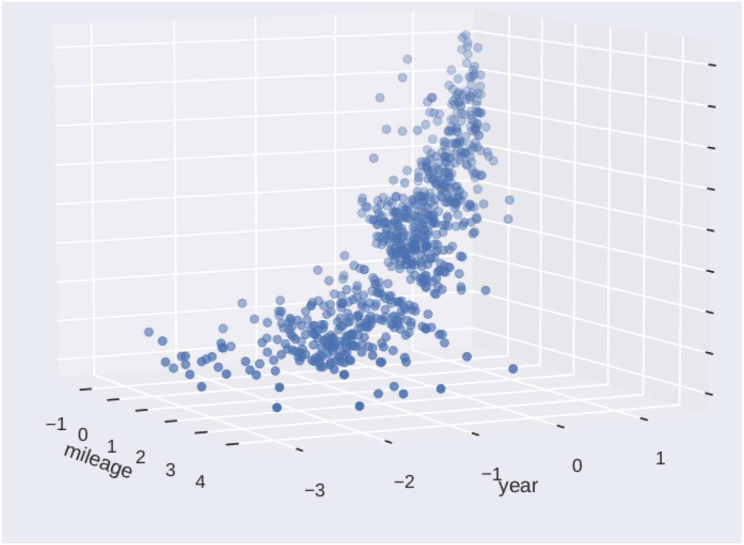
```
In [5]: Xtrs.shape
Out[5]: (750, 2)

In [6]: ytr.shape
Out[6]: (750,)

In [7]: Xtes.shape
Out[7]: (250, 2)

In [8]: yte.shape
Out[8]: (250,)
```

We use the "standard scaling" for both mileage and price.

Here are some plots of the training data.

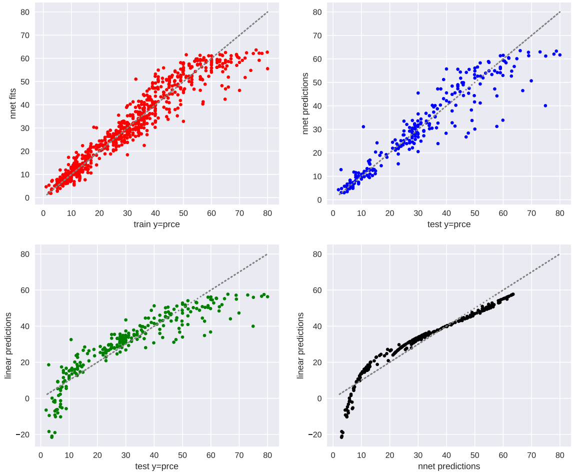


Training data. 3D plot of (mileage,year) (scaled) vs price.



Neural net result with 2 input features (mileage,year) and 50 units in a single hidden layer.

top row: In and out of sample fits and predictions.
second row: comparison to linear predictions and fit.

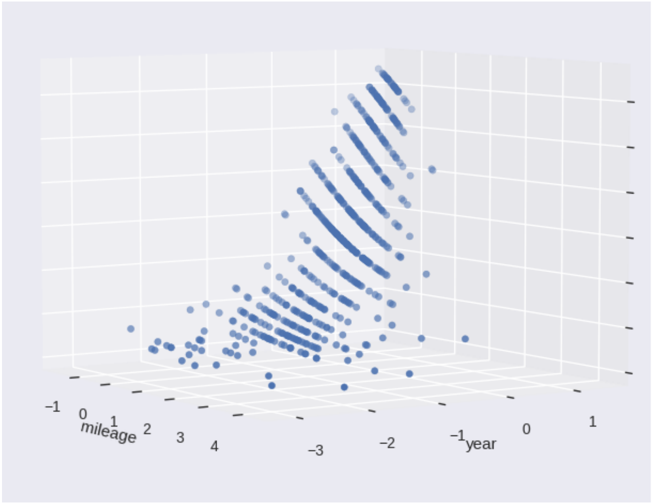


Correlations and rmse on test data.

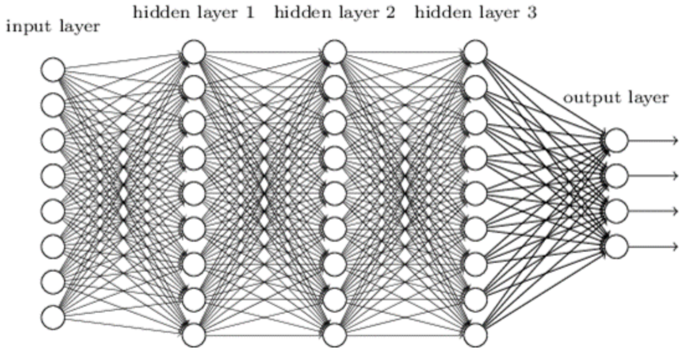
Compare yte: test y=price, yprednn: neural net prediction, ypredlin: linear regression prediction.

```
               yte   yprednn  ypredlin
yte       1.000000  0.938916  0.896020
yprednn   0.938916  1.000000  0.950466
ypredlin  0.896020  0.950466  1.000000

In [16]: f'{minrmse:0.2f}'
Out[16]: '6.60'

In [17]: f'{rmselin:0.2f}'
Out[17]: '8.57'
```

3D plot of (mileage,year) vs neural net fit on train data.



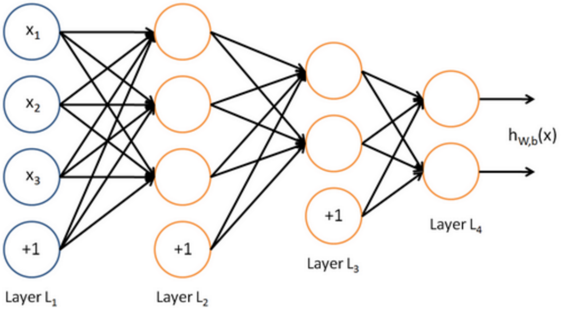
4. Deep Neural Networks



A deep neural network is a neural network with more than one hidden layer.

The activations at each unit are a linear function of the activations from the all the units in the previous layer (plus an intercept) put into a nonlinear activation function.

A simple version with 2 layers.
Input layer is X.
First hidden layer has 3 units.
Second hidden layer has 2 units.
Output layer has 2 units.



`http://ufldl.stanford.edu/wiki/index.php/Neural_Networks`

The input layer is the feature vector.

Note that the output layer can have more than on unit.

This will be useful when we consider multinoulli outcomes (categorical outcomes with more than two categories) but for our basic cases of a single numeric outcome and a binary outcome, there will be just one unit in the output layer.

In theory, any function can be approximated arbitrarily well with just a single layer.

But, in some problems in turns out to be effective to incrementally understand what transformation of X helps us understand Y.

Chollet, "Deep Learning with Python", section 2.3.6.

> *Deep learning .. takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones. ... Each layer in a deep network applies a transformation that disentangles the data a little - and a deep stack of layers makes tractable an extremely complicated disentanglement process.*

Chollet, “Deep Learning with Python”, section 1.2.6.

Two essential essential characteristics of how deep learning learns from data:

> *...the incremental, layer-by-layer way in which increasingly complex representations are developed ...*

and

> *... these intermediate incremental representations are learned jointly.*

Rather than have to do a lot of “feature engineering” the deep neural net can figure out the potentially complex high dimensional transformation of the features which is best for predicting y.

Reality check, section 10.6 of ISLR.

Tried Hitters data with neural nets and lasso and *with much less effort* got a better out of sample mse with lasso than neural nets.

In addition, the linear model is much simpler and more interpetable.

Deep learning folks might scoff at this example, but the bulk of applied statistics is more like the Hitters data than like digit recognition.

The notation for the general case gets a bit intense.
You can skip this if you like.

Let's start by letting ℓ index the layers.

ℓ goes from 1 to L where $\ell = 1$ is the input layer (x) and L is the final output layer.

To keep things simple, we will have just one outcome with associated activation function g^L. For a single numeric outcome, g^L would typically be the identity function $I(x) = x$.

We will use the same activation function g at all the interior units (neurons), but it would be a minor change to have activation function $g^{(\ell)}$ at layer ℓ.

Let p_ℓ be the number of neurons at layer ℓ.
Note that $p_1 = p$ where p is the dimension of x since that is the input layer.

Lots of Notation !!!!:

$Z_k^{(\ell)}$: the value of the linear function of the activation from the previous layer at the k^{th} unit of layer (ℓ), $k = 1, 2, \ldots, p_\ell$.

We have $Z_{\text{unit}}^{(\text{layer})}$. Similary, we have activations $a_k^{(\ell)}$ with,

$a_k^{(\ell)} = g(Z_k^{(\ell)})$.

$w_{kj}^{(\ell)}$ = weight from $a_j^{(\ell)}$ (at layer ℓ) to $Z_k^{(\ell+1)}$ (at layer $(\ell+1)$).

Think of w as $w_{kj}^{(\ell)} = w_{k \leftarrow j}^{(\ell)}$.

$b_k^{(\ell)}$ = intercept for $Z_k^{(\ell+1)}$ (at layer $(\ell+1)$).

$$Z_k^{(\ell)} = b_k^{(\ell-1)} + \sum_{j=1}^{p_{(\ell-1)}} w_{kj}^{(\ell-1)} \, a_j^{(\ell-1)}, \quad k = 1, 2, \ldots, p_\ell.$$

$$Z_k^{(\ell)} = b_k^{(\ell-1)} + \sum_{j=1}^{p_{(\ell-1)}} w_{kj}^{(\ell-1)} \, a_j^{(\ell-1)}, \quad k = 1, 2, \ldots, p_\ell.$$

Matrix/Vector version:

$$Z^{(\ell)} = (Z_1^{(\ell)}, Z_2^{(\ell)}, \ldots, Z_{p_\ell}^{(\ell)})'$$

$$a^{(\ell)} = g(Z^{(\ell)})$$

$$b^{(\ell)} = (b_1^{(\ell)}, b_2^{(\ell)}, \ldots, b_{p_{(\ell+1)}}^{(\ell)})'$$

$$W^{(\ell)} = \left[w_{kj}^{(\ell)} \right], \quad p_{(\ell+1)} \times p_\ell$$

Then,

$$Z^{(\ell)} = b^{(\ell-1)} + W^{(\ell-1)} a^{(\ell-1)}$$

5. Activation Functions

Up until now we have used the sigmoid (or logistic) activation function:

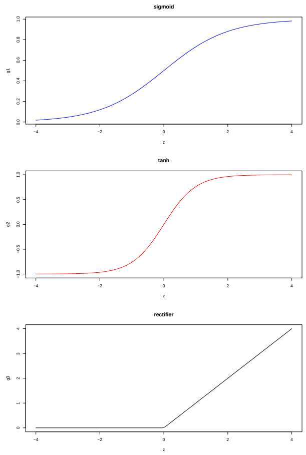
$$g(z) = \frac{1}{(1 + e^{-z})}$$

Other commonly used activation functions are tanh (hyperbolic tangent):

$$g(z) = \frac{e^{z} - e^{-z}}{(e^{z} + e^{-z})}$$

and the rectified linear unit, or RELU:

$$g(z) = z \text{ for } z > 0, \text{ and 0 else.}$$



Intuitively, it does not seem like there should be much of a difference between sigmoid and tanh, but it turns out tanh works better for gradient computations and seems to be favoured in the deep world.

RELU is very popular, especially for images.

6. Regularization and Dropout

We can choose L1 and L2 penalties to regularize the parameter estimation.

Typically, the regularization is applied to the weights but not the biases.

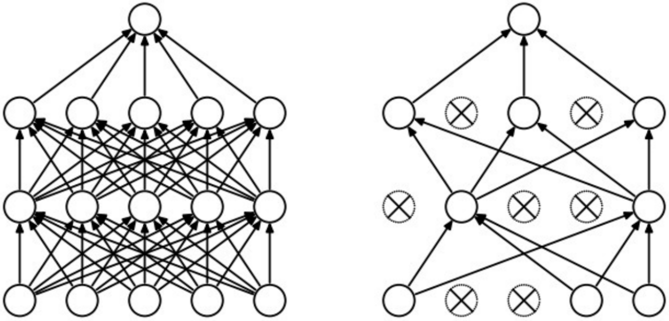
Here is a snippet of keras/python code illustrating the building of a model with two layers and regularization at each layer.

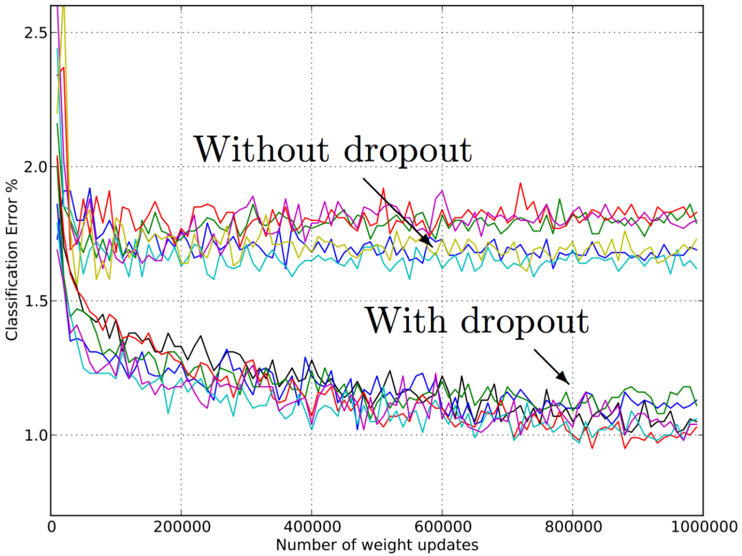
```
#make model
lp1pen = .0500 #l1 penalty
#nunit =  500
nunit =  100
nx = Xtrs.shape[1] # number of x’s
nn2 = tf.keras.models.Sequential()
## add one hidden layer
nn2.add(tf.keras.layers.Dense(units=nunit,activation=’tanh’,
   kernel_regularizer = tf.keras.regularizers.l1(lp1pen),input_shape=(nx,)))
## add second hidden layer
nn2.add(tf.keras.layers.Dense(units=nunit,activation=’tanh’,
   kernel_regularizer = tf.keras.regularizers.l1(lp1pen)))
## one numeric output
nn2.add(tf.keras.layers.Dense(units=1))
```

Dropout

Dropout is another popular way to regularize a neural net fit.
Eliminate some of the connections.
You simply randomly pick some of the connections to eliminate.

Dropout:


(a) Standard Neural Net (b) After applying dropout.


Without dropout
With dropout
Classification Error %
2.5
2.0
1.5
1.0
0
200000
400000
600000
800000
1000000
Number of weight updates

7. Optimization

How do we learn all the weights and biases !!!!

There could be a lot of them !!!!

Suppose we have 2 numeric inputs, two hidden layers with 100 units each and 1 numeric output.

Then we have

$(2*100) + (100)*(100)+100*1 = 10{,}300$

weights to estimate!!

One thing that makes working with neural nets different is that you have to have a little understanding of the optimization to run the software.

You even have to make choices about the optimization !!

Gradient Descent

As usual we have training data and a loss function $L(x, y, \theta)$ where θ denotes all the weights and biases.

For example with a numeric outcome we have

$$L(x, y, \theta) = (y - \hat{y}(x, \theta))^2$$

We seek to minimize:

$$\sum_{i=1}^{n} L(x_i, y_i, \theta).$$

where θ is all the biases and weights.

Computing the Hessian matrix is not practical, so the methods are based on the gradient.

Gradient descent just uses the update

$$\theta \to \theta - \epsilon \, \tfrac{1}{n} \sum_{i=1}^{n} \nabla L(x_i, y_i, \theta).$$

where the gradient is with respect to the elements of θ (all the biases and weights) and ϵ is called the “learning rate”.

Stochastic Gradient Descent

If n is big, each update will take a long time to compute.

Stochastic gradient descent computes the gradient using subsets of the data called *minibatches*.

At iteration k of the algorithm we select a set of minibatch subsets of data $\{x_i^b, y_i^b\}, i = 1, 2, \ldots, m$.

Then we cycle through the minibatches using the update (at each minibatch):

$$\theta \to \theta - \epsilon_k \, \tfrac{1}{m} \sum_{i=1}^{m} \nabla L(x_i^b, y_i^b, \theta).$$

An *epoch* is one pass through the entire data set.

Note:

How do we compute the gradient?

It is just the chain rule.

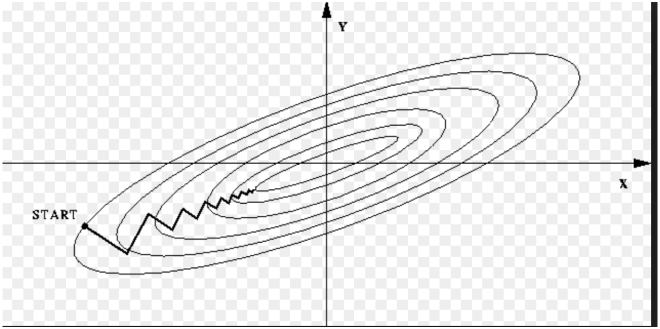
However, a lot of work has gone into organizing the the computations so they can be done efficiently and the method for computing the gradient is called *back propogation*.

To evaluate the model, you move "forward" through the layers from inputs to output layer.

To evaluate the gradient you move backward from the output layer.

Here is a (stolen) picture showing basic gradient descent.

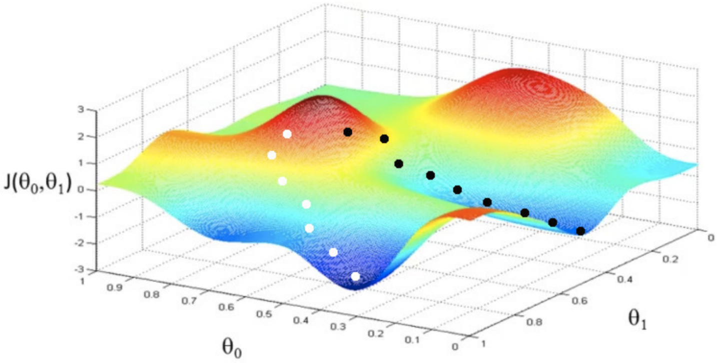
We always move downhill, perpendicular to the contours.



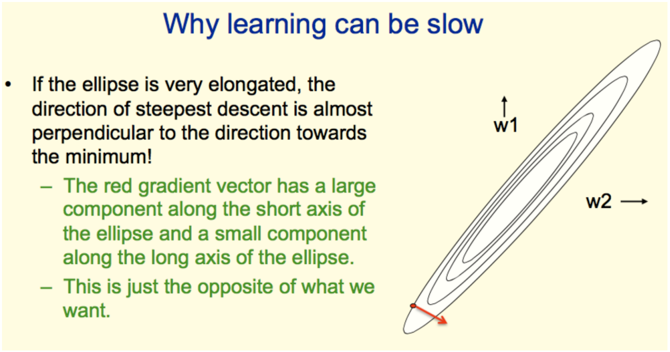
Note, *stochastic* gradient descent will tend to move downhill but not with the full gradient information at each move.

https://www.internalpointers.com/post/gradient-descent-function

This picture illustrates going to different local minimums depending on the starting value.



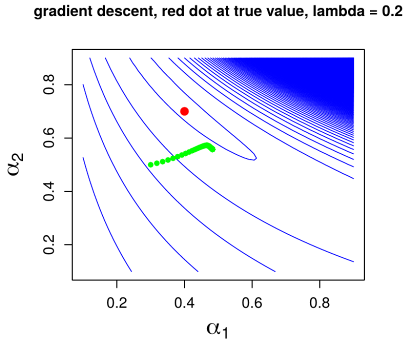
This picture gives the basic idea of how gradient descent could be much worse than Newton's method.

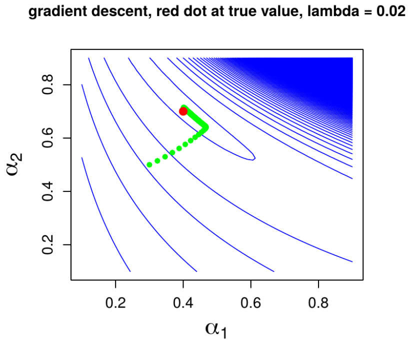


Why learning can be slow

- If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum!
  - The red gradient vector has a large component along the short axis of the ellipse and a small component along the long axis of the ellipse.
  - This is just the opposite of what we want.

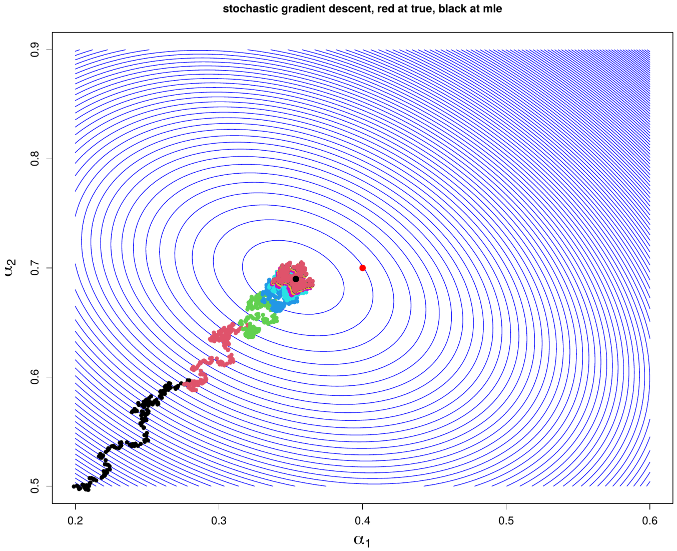
Gradient descent.





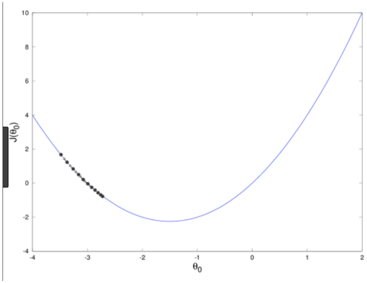
Path at left was l2 regularized.
Path at right was not.

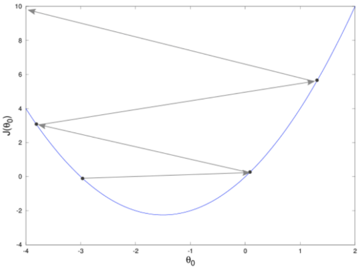
Stochastic Gradient descent. Epochs color coded.



This picture shows “gradient” descent in 1-d and illustrates the role of the learning rate.

$$x \to x - \epsilon_k f'(x)$$





At left we have a small fixed ϵ_k.

At right we have a big fixed ϵ_k.

Adaptive Learning rates

Clearly, the learning rate is a key part of procedure.

The are a variety of schemes for adaptively adjusting the learning rates for individual weights.

Momentum:
Momentum based methods address the problems with local optima, flat spots, and zigzagging by incorporating the overall direction of past moves.

Our next step is the a weighted combination of the previous step and the current gradient information.

$$\theta_t = \theta_{t-1} - \epsilon_k \nabla L + \gamma(\theta_{t-1} - \theta_{t-2})$$

where t indexes iteration.

RMSprop

Weights that have varied a lot in past interations have downweighted learning rates.

Adam:

Combines momentum and RMSprop idea.

Clearly fitting a neural net model is no joke !!

The stochastic gradient descent algorithm is typically intiallized with random values for the parameters.

Since there are local minimum,

you can run it twice and get different answers !!!!

In practice it can be important to "run it" several times and play with basic parameters like the number of epochs. This can all end up being very labour intensive.

Fitting neural networks: Tips from h2o

- more layers for more complex functions (more nonlinearity).
- more neurons per layer to fit finer structure in data.
- add regularization (max_w2=50 or L1 = 1e-5).
- do a grid search to get a feel for parameters.
- try "Tanh", the "Rectifier".
- try dropout (input 20%, hidden 50%).

Note: max_w2:
An upper limit for the (squared) sum of the incoming weights to a neuron.
h2o default is to have no limit.

8. Cars Example with Deep Learning

Let's do cars with (mileage,year) and price with more than one layer.

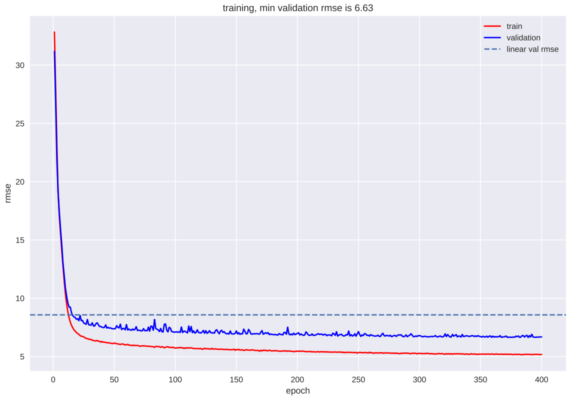
Note all the choices we have to make about model architecture, optimization, and regularization.

To make all this concrete, let's look at the python-keras code.

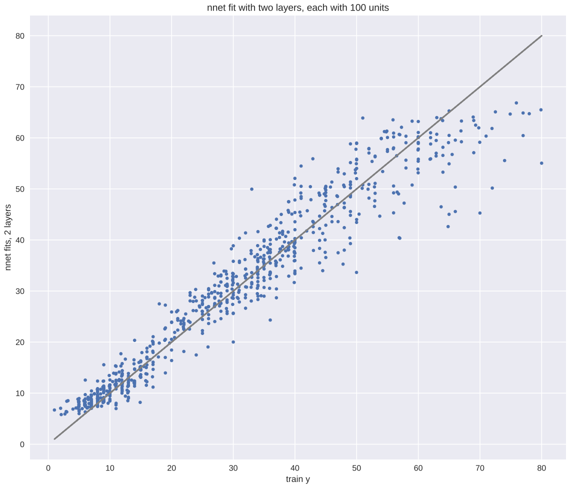
Two layers, each with 100 units.
L1 regularization at each layer.
tanh activation at each layer (except output layer).
rmsprop learning rate.
mse loss.

```
seed=34
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed) ## ? just need this one ??

#make model
lp1pen = .0500 #l1 penalty
nunit =  100
nx = Xtrs.shape[1] # number of x’s
nn2 = tf.keras.models.Sequential()
## add one hidden layer
nn2.add(tf.keras.layers.Dense(units=nunit,activation=’tanh’,
   kernel_regularizer = tf.keras.regularizers.l1(lp1pen),input_shape=(nx,)))
## add second hidden layer
nn2.add(tf.keras.layers.Dense(units=nunit,activation=’tanh’,
   kernel_regularizer = tf.keras.regularizers.l1(lp1pen)))
## one numberic output
nn2.add(tf.keras.layers.Dense(units=1))

#compile model
nn2.compile(loss=’mse’,optimizer=’rmsprop’,metrics=[’mse’])

# fit
nepoch = 400
nhist2 = nn2.fit(Xtrs,ytr,epochs=nepoch,verbose=1,batch_size=20,validation_data=(Xtes,yte))
```

Training.
In out out sample loss (rmse) by epoch.



Fit on train.



nnet fit with two layers, each with 100 units
nnet fits, 2 layers
train y
0
10
20
30
40
50
60
70
80

Out of sample predictions.
yprednn is from the single layer model and yprednn2l is the 2 layers of 100 units model.
ypredlin is the linear model.

```
               yte   yprednn  ypredlin  yprednn2l
yte       1.000000  0.938916  0.896020   0.942015
yprednn   0.938916  1.000000  0.950466   0.997156
ypredlin  0.896020  0.950466  1.000000   0.942825
yprednn2l 0.942015  0.997156  0.942825   1.000000

In [22]: f'{minrmse:0.2f}'
Out[22]: '6.63'
```

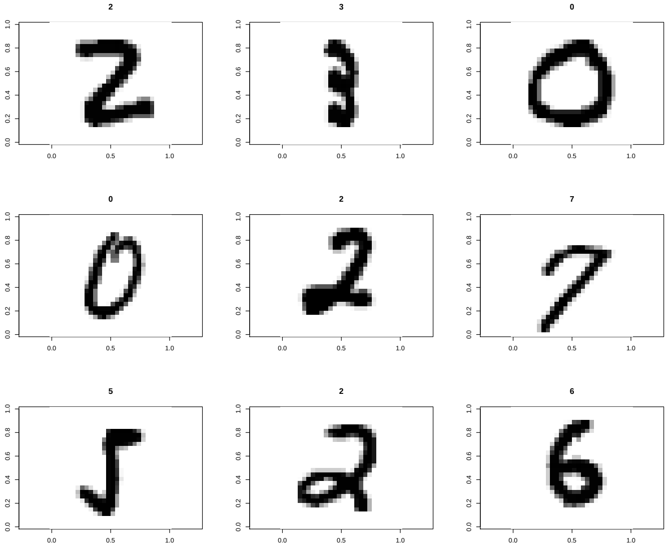
Correlation and out-of-sample rmse is about the same as the single layer model.

9. Binary classification, IMDB example

blah.

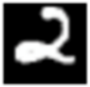
10. MNIST: classification with 10 outcomes

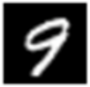
Handwritten digits captured as 0-255 grayscale values on a 28×28 grid.



Digit recognition:

Guess the digit, given the $28^2 = 784$ values:

$P(y = 2 \mid$  $, b)$

$P(y = 9 \mid$  $, b)$

where "b" is model parameters (e.g. weights).

Easy for a person, hard for a machine !!

Note:

Our black and white images are values in [0,255] on a 2 dimensional grid of pixels.

Color images are (r,g,b) values on a grid of pixels.

(r,g,b): red, green, blue.

For example: the input might be 32 x 32 x 3.

11. Simple Gradient Example

How do we compute the gradient vector?

Let’s explicitly compute the gradient for the simplest version of a neural net model: one x, one layer of 2 units, one numberic outcome, squared error loss.

$$f(x,\Theta) = \beta_0 + \beta_1 g(w_{10} + w_1 x) + \beta_2 g(w_{20} + w_2 x)$$

$$= \beta_0 + \beta_1 A_1 + \beta_2 A_2$$

$$A_1 = g(z_1) \quad A_2 = g(z_2) \quad z_1 = w_{10} + w_1 x \quad z_2 = w_{20} + w_2 x$$

$$L(y,\hat{y}) = (y - \hat{y})^2 \qquad \hat{y} = f(x,\Theta)$$

$$\Theta = (\beta_0, \beta_1, \beta_2, w_{10}, w_1, w_{20}, w_2)$$

$$L(\Theta) = \sum_{i=1}^{n} (y_i - \hat{y})^2 = \sum_{i=1}^{n} L(y_i, f(x_i,\Theta))$$

$$\nabla L(\Theta) = \sum_{i=1}^{n} \nabla L(y_i, f(x_i,\Theta))$$

Drop :

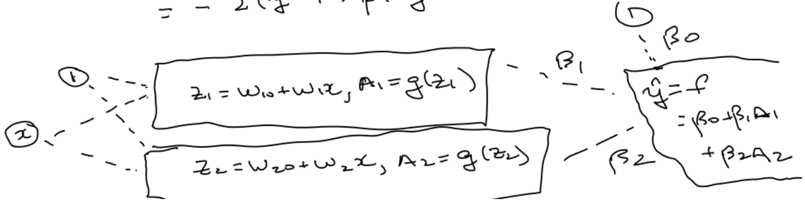
$$L(y,f) = (y-f)^2$$

$$\begin{tikzcd}[row sep=small] w_1 \arrow[d] \\ z_1 \arrow[d] \\ A_1 \arrow[d] \\ f \arrow[d] \\ L \end{tikzcd}$$

$$\frac{\partial L(y,f)}{\partial \beta_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial \beta_1} = -2(y-f)A_1$$

$$\frac{\partial L(y,f)}{\partial w_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial A_1}\frac{\partial A_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$

$$= -2(y-f)\,\beta_1\, g'(z_1)\, x$$



1
x
$z_1 = w_{10} + w_1 x,\ A_1 = g(z_1)$
$z_2 = w_{20} + w_2 x,\ A_2 = g(z_2)$
1
β_0
β_1
β_2
$\hat{y} = f$
$= \beta_0 + \beta_1 A_1$
$+ \beta_2 A_2$

But how does this work in a general deep network?

The back propagation algorith works by going back through the network starting at the loss. At each step backwards all the partial derivatives are computed which enable us to keep track of the downstream effect on the loss due to a change in the parameters upstream.

While an overall deep network is complex is is composed of many basic linear (often called tensor) operations. Automatic differentiation used the chain rule to compute the derivative given a chain of operations with know derivatives.

Another key to making all this work is parallel computing with GPU has been used to speed things up.

See section 2.4.4 of Chollet.

12. How Does it Work Again, XOR

Let's look again at how a neural net works by playing around with the famous XOR example.

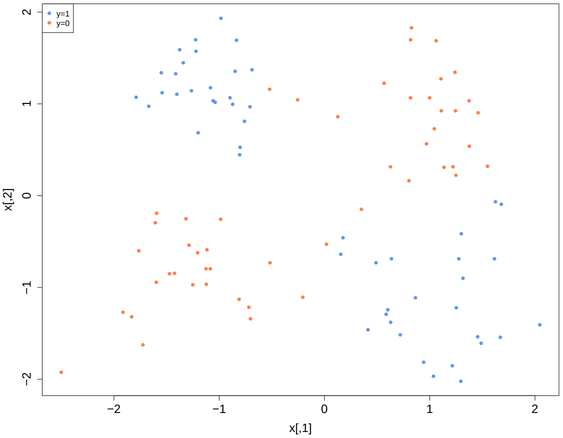
This is example is famous because it is a simple example where linear classification:

$$y = 1 \text{ if } a + b_1 x_1 + b_2 x_2 > 0$$

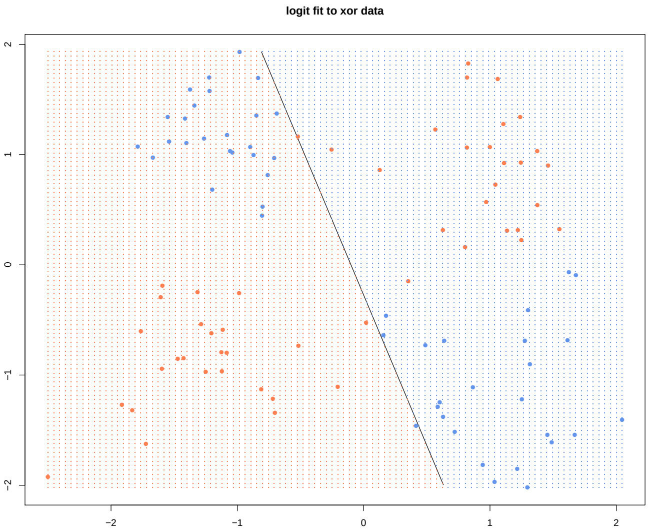
cannot work.

Basically, y is 1 if the $sign(x_1) \neq sign(x_2)$ but I added noise so a few points cross the boundaries.

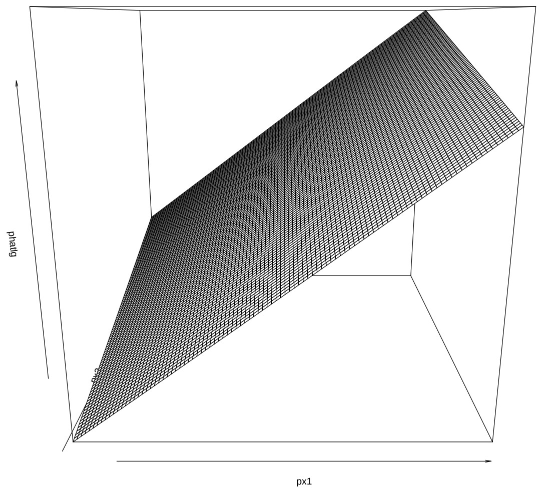
Here is a plot of the (simulated) data.



Here is the decision boundary ($\hat{y} = 1$ if $\hat{p} > .5$) for a linear logit fit.



Here is a plot of $\hat{p}(x_1, x_2)$ from the logit fit.



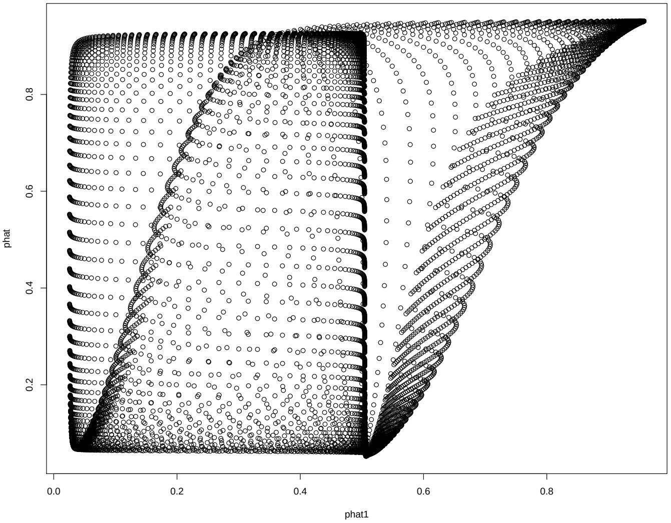
Really all the $\hat{p}$ are close to .5 !!

```
> print(summary(lgfit))

Call:
glm(formula = y ~ ., family = binomial, data = dfd)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.25921  -1.17512   0.02788   1.17894   1.23320

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.01013    0.20113   0.050    0.960
x1           0.10058    0.17129   0.587    0.557
x2           0.03688    0.18028   0.205    0.838

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 138.63  on 99  degrees of freedom
Residual deviance: 138.27  on 97  degrees of freedom
AIC: 144.27

Number of Fisher Scoring iterations: 3

> summary(phatl)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.4217  0.4676  0.4964  0.4964  0.5253  0.5713
```

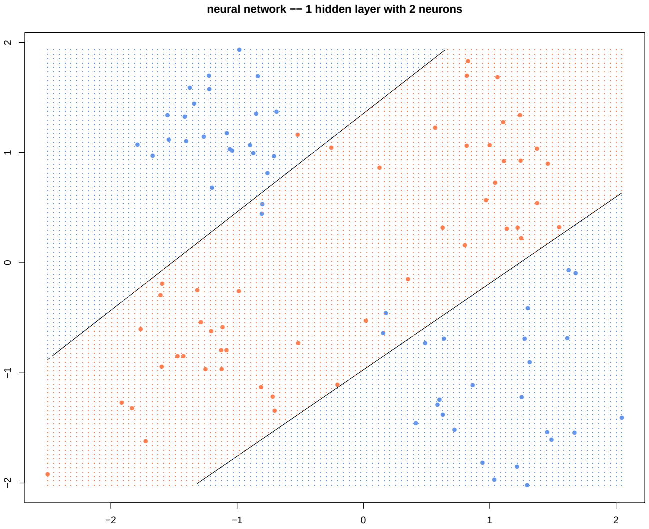
Let's try a nn fit.

```
#uses random starting values for iterative optimization
set.seed(99) #misses
xnn = nnet(y~.,dfd,size=2,decay=.1)
phat1 = predict(xnn,gd)[,1]

set.seed(14) #works
xnn = nnet(y~.,dfd,size=2,decay=.1)
phat = predict(xnn,gd)[,1]

#plot fits, far out!!
plot(phat1,phat)
```

Far out.

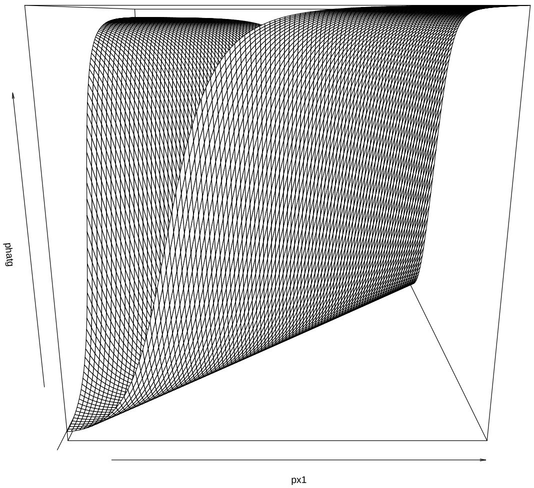


Here is the nn decision boundary (from the one that worked).



Beautiful !!!

Here is a plot of $\hat{p}(x_1, x_2)$ from the nn fit.



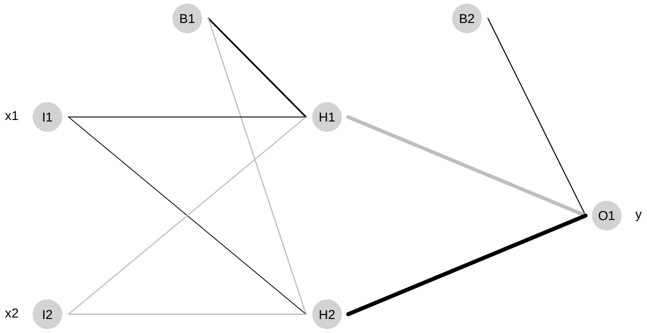
Obvious !!!!????

(see plot3d in xor.R).

```
> summary(xnn)
a 2-2-1 network with 9 weights
options were - entropy fitting  decay=0.1
 b->h1 i1->h1 i2->h1
  3.35   2.38  -2.66
 b->h2 i1->h2 i2->h2
 -2.73   2.28  -2.90
 b->o h1->o h2->o
 2.54 -5.84  6.30
```

Basically uses $x_1 - x_2$!!!!.

A plot of xnn:



13. More on Digit Recognition

The digit recognition problem is a famous problem of basic importance in Machine Learning/Statistics.

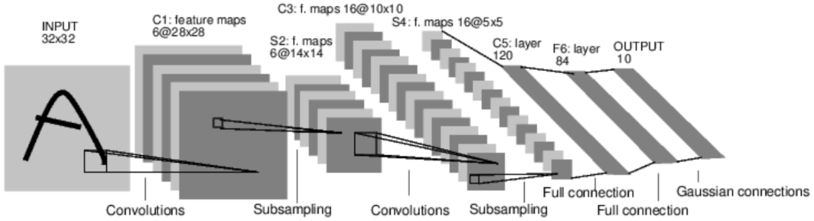
Deep neural nets have been very successful
with some special twists !!!

The pixel layout is a very special structure and some approaches have been developed to take advantage of it.

These approaches coupled with deep learning are the "state of the art".

Let's just get a rough idea of what is involved.

Besides the usual hidden layers we have looked at, different kinds of layers are used to take advantage of the pixel structure:



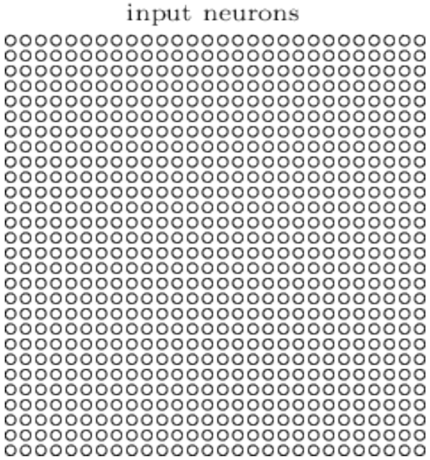
Convolution layers replace a pixel value with the average of nearby pixels.

Pooling layers replace of rectangular set of pixels with the maximum value.

See http://yann.lecun.com/exdb/lenet/

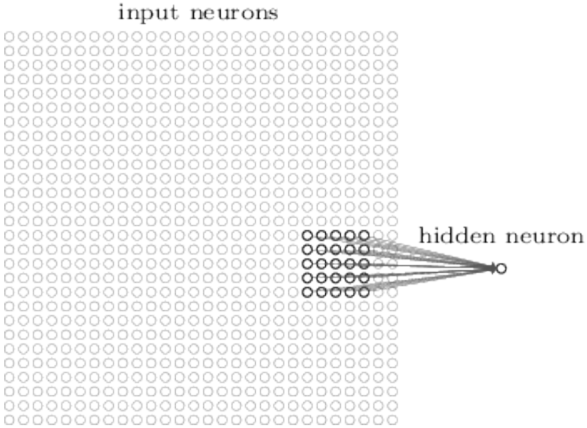
Convolution Layers:

Here is our 28^2 input layer:

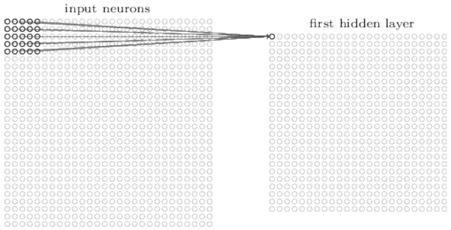


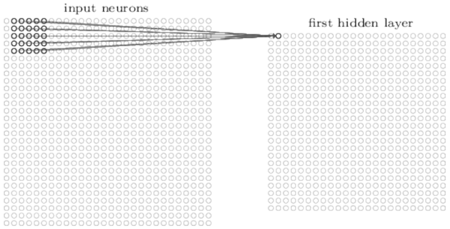
From: http://neuralnetworksanddeeplearning.com/chap6.html

To get single neuron for the next layer, take a weighted average of neurons in a box where the neuron is at the top left corner.
(in images you often make the origin the top left).



You have to pick the weights and number of neighbors.

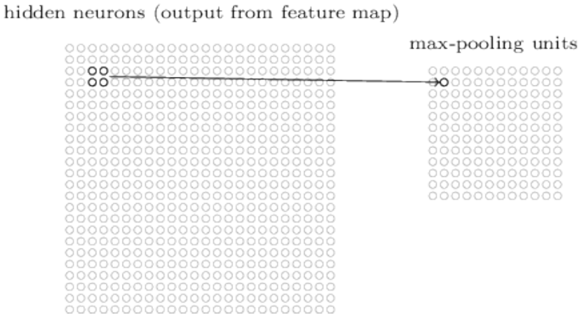




This will give an ouput layer a little smaller or about the same size depending on how you do it.

Pooling Layer:

A pooling layer replaces the pixel values in non-overlappying regions with the maxiumum value.



This will typically reduce the number or neurons in the next layer. The pooling layer "introduces an elmement of local translation invariance" (Efron and Hastie).

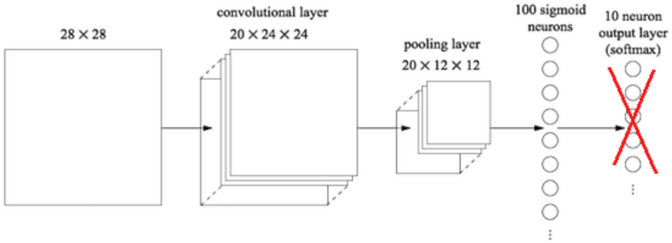
Another cool idea:

Expand the set of examples.

For each (x, y) pair produce a set a pairs
(x_s, y) where x_s is obtained from x by small distortions:
scaling, rotation, . . .

Then add all the generated (x_s, y) to your training data!!!

Another cool idea:



Use the output of the last layer as a representation of your data.

Fit a model with this representation.