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

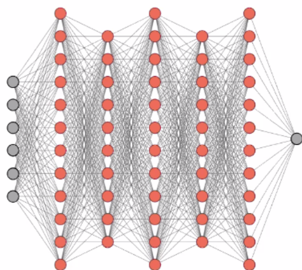
Mladen Kolar and Rob McCulloch

1. Introduction and a Single Layer Fit
2. Understanding the Basic model, what are units?
3. More than one x
4. Deep Neural Networks
5. Activation Functions
6. Regularization and Dropout and Early Stopping
7. Optimization
8. Cars Example with Deep Learning
9. Binary classification, IMDB example
10. Simple MNIST: multinoulli classification
11. Simple Gradient Example
12. How Does it Work Again, XOR
13. XOR Deep
14. More on Digit Recognition
15. Recurrent Neural Networks
16. Don't worry, man vs machine

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*.



How do all the layers of neurons give us $y = f(x)$??????!!!!!

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, dropout, and early stopping.

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 just a single layer.

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

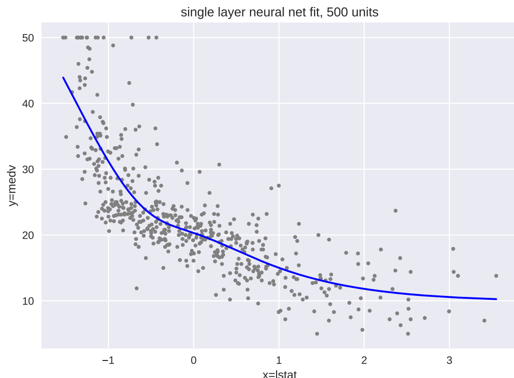
2. Understanding the Basic model, what are units?

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

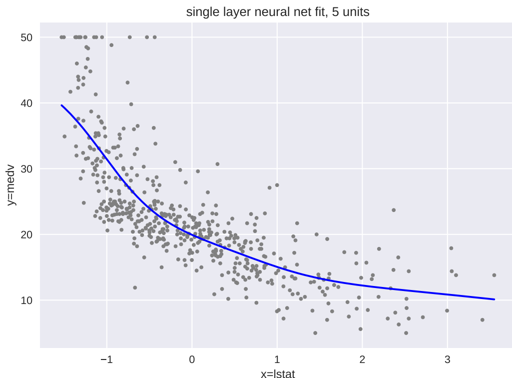
Here is the fit with "500 units".

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”.



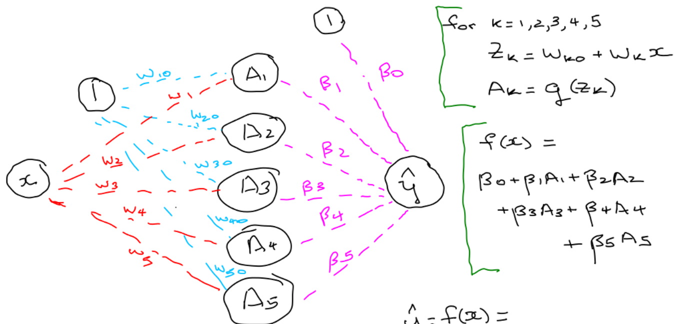
Does not look quite as good as with 500 units, but let’s pull this simpler fit apart to see how it actually works.

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 $\beta_0, k = 1, 2, 3, 4, 5$.



ŷ = f(x) =

$$\beta_0 + \sum_{k=1}^{5} \beta_k A_k$$

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

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

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*.

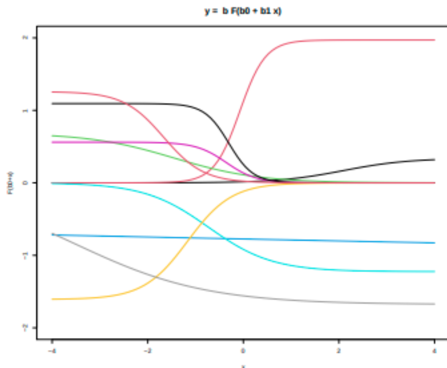
Model:

- 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.

This gives us a nonlinear function of x.
Clearly, the non-linear activation function g is crucial since if we just linearly combined linear we would just get linear.

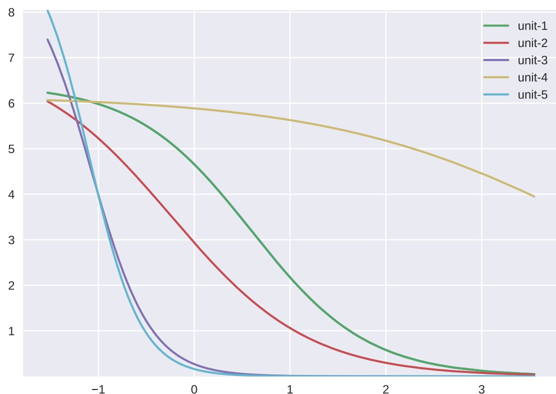
Why is this such a great idea ????

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



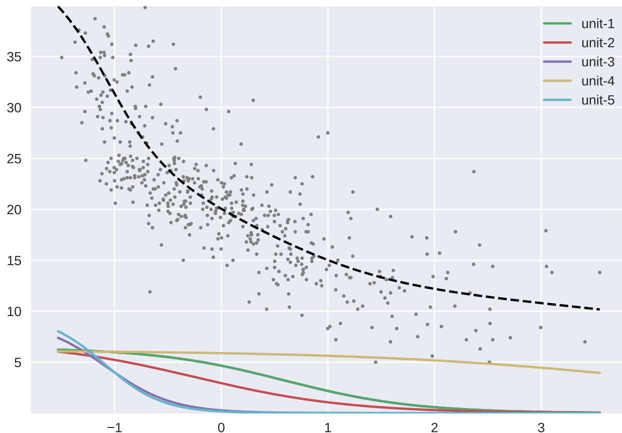
We can get get just about any function we want just by adding up these kinds of functions !!

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



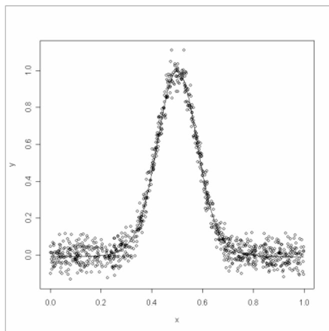
For example unit 1 gives us the function $6.42 * g(.973 - 1.64x)$.

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.

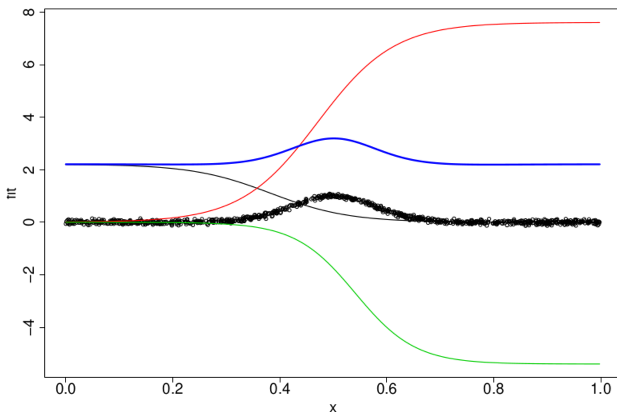
The line drawn through the data is a neural net fit with just three units.



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??

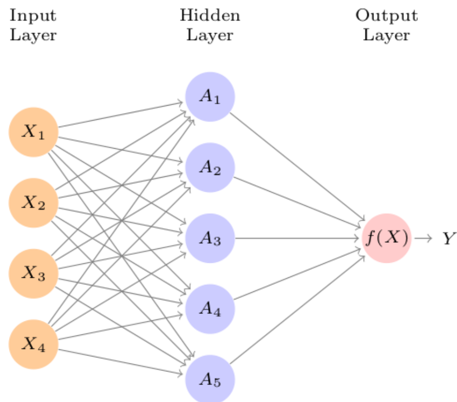
I did this in R with the nnet package.

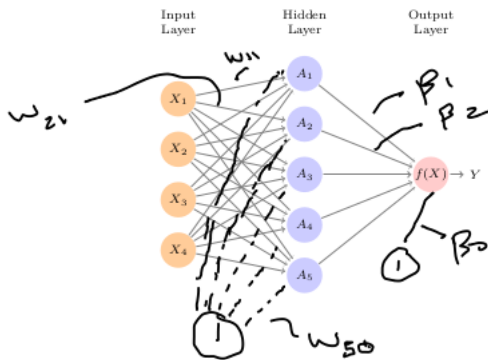
```
F = function(x) {return(exp(x)/(1+exp(x)))}

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.





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

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

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

All in one line:

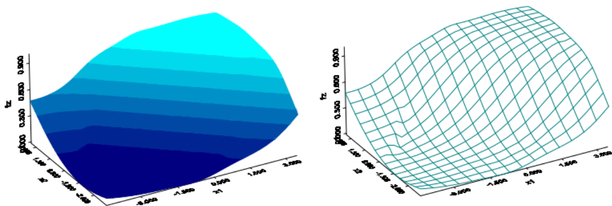
$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g(w_{k0} + \sum_{j=1}^{p} w_{kj} \, X_j)$$

- X is our input layer with each unit corresponding to a component X_i.
- K units in a single hidden layer. Each unit corresponds to a linear function of the units in the previous layer.
- a single unit in our output layer.

A function of the form

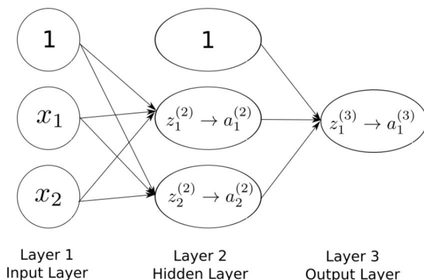
$$\beta\, g(w_0 + \sum_{j=1}^{p} w_j X_j)$$

(we dropped k)



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

Different notation, very simple model.



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

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*.

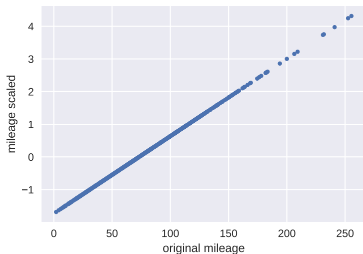
Example: Used Cars with mileage and year

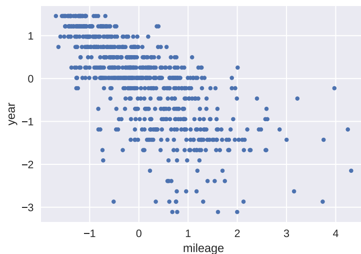
75% - 25% train-test split.

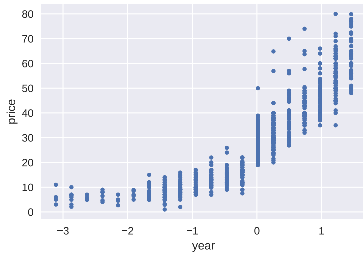
```
In [5]: Xtrs.shape
Out[5]: (750, 2)

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

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

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

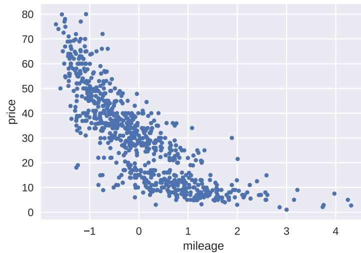
We use the "standard scaling" for both mileage and year.

Here are some plots of the training data.

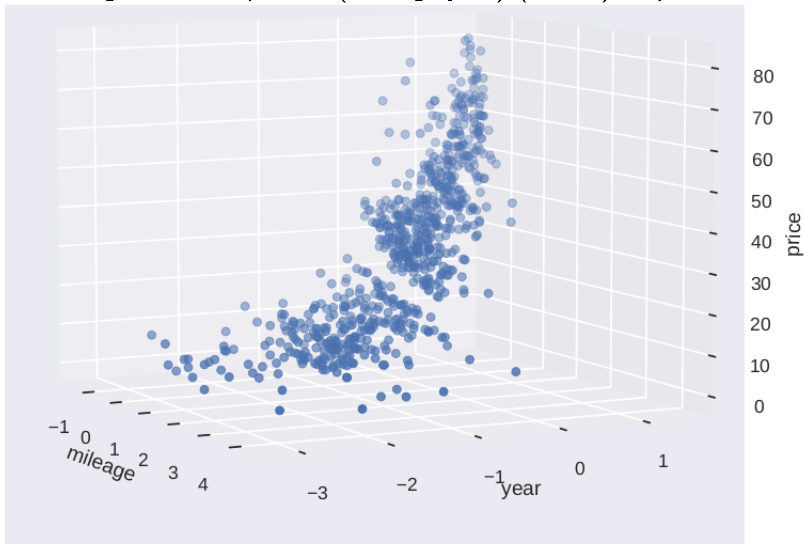






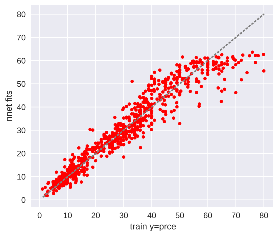


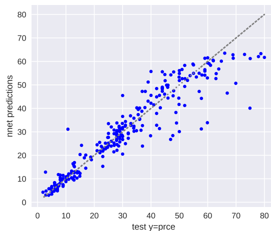
Training data. 3D plot of (mileage,year) (scaled) vs price.

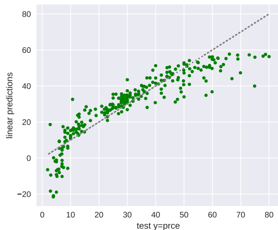


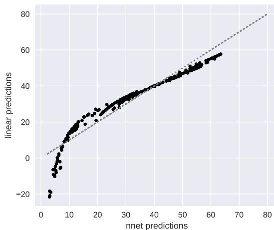
Neural net result with 2 input features (mileage,year) and 50 units in a single hidden layer.

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







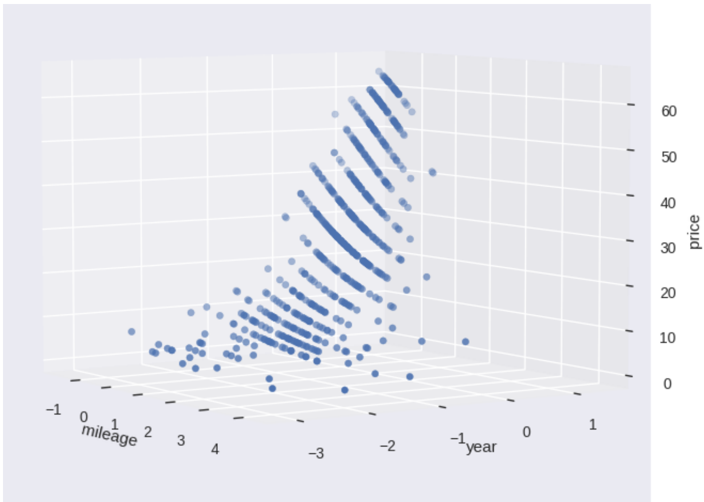


Correlations and rmse on test data.

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.

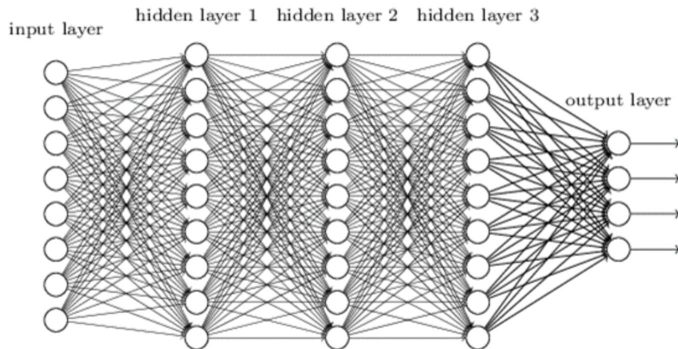


What is neural about neural networks??

ISLR:

> *The name neural network originally derived from thinking of these hidden units as analogous to neurons in the brain – values of the activation* $A_k = h_k(X)$ *close to one are "firing" while those close to zeror are "silent" (using the sigmoid activation function).*

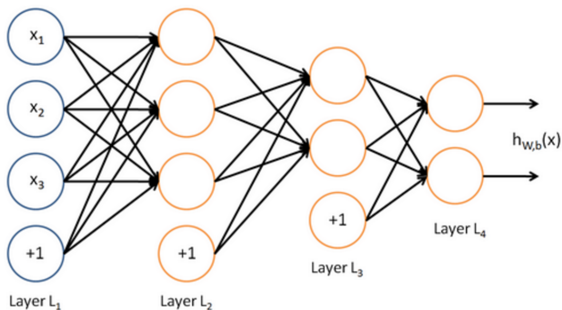
4. Deep Neural Networks



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”.

> *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”.

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.

ISLR, section 10.2.

> *Modern neural networks typically have more than one hidden layer, and often many units per layer. In theory a single hidden layer with a large number of units has the ability to approximate most functions. However the learning task of discovering a good solution is made much easier with multiple layers each of modest size.*

We can think of deep networks as building a complicated function using a sequence of *compositions*.

The activations in the units of each layer represent of nonlinear function of the activations in the previous layers and the network is the composition of all these functions.

recall: $(h \, o \, f)(x) = h(f(x))$, is the function obtained by the compostion of h and f.

Reality check, section 10.6 of ISLR, “When to use Deep Learning”.

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 basic deep neural net we are considering is called a *fully connected* or *dense* network.

Each unit in each layer is "connected" to each unit of the previous and subsequent layer.

Some of the big successes in neural nets design very special units and sets of connections tailored to the structure of the problem. We have convolution networks for images and recurrent neural networks for sequences.

The notation for the general dense network 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 kth 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:

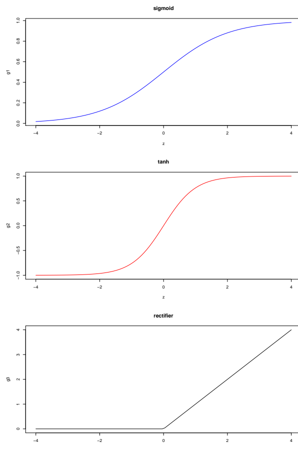
$$g(z) = \frac{1}{(1 + e^{-z})}$$

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 and Early Stopping

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

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

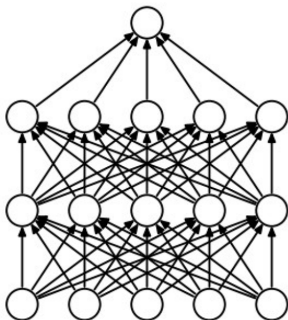
Here is a snippet of keras/python code illustrating the building of a model with two layers and regularization at each layer.

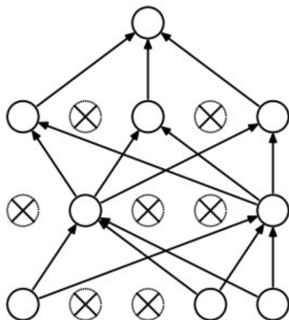
```
#make model
lp1pen = .0500 #l1 penalty
#nunit =  500
nunit =  100
nx = Xtrs.shape[1] # number of x's
nn2 = tf.keras.models.Sequential()
## add one hidden layer
nn2.add(tf.keras.layers.Dense(units=nunit,activation='tanh',
   kernel_regularizer = tf.keras.regularizers.l1(lp1pen),input_shape=(nx,)))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

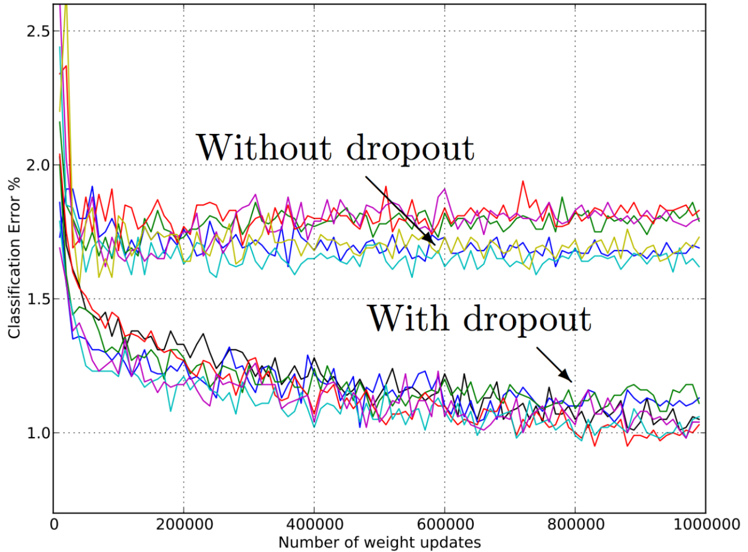
Dropout

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

Early Stopping

Neural nets are fit with an iterative optimization.

As you iterate, the fit on the training data just gets better and better (hopefully).

As simple and time honored way to “regularize” is just to stop the iterations before the fit on the training data gets *too good*.

This is called *early stopping*.

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.

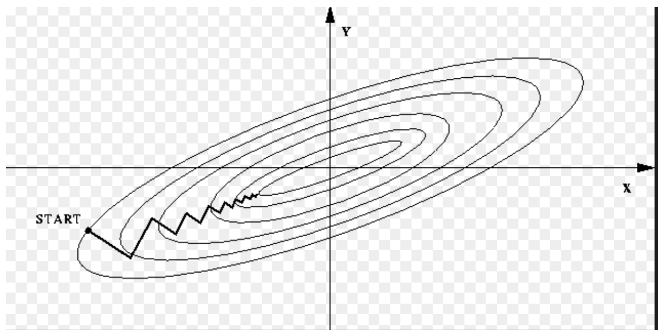
Gradient descent just uses the update

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

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

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

We always move downhill, perpendicular to the contours.



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*.

Let (x_i^b, y_i^b) be the observations in the bth minibatch, $i = 1, 2, \ldots, m_b$, $b = 1, 2, \ldots, B$.

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

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

Usually the minibatches are a disjoint partition of the training data, all having the same size.
An *epoch* is one pass through the entire data set (all the batches).

Note:

How do we compute the gradient?

It is just the chain rule.

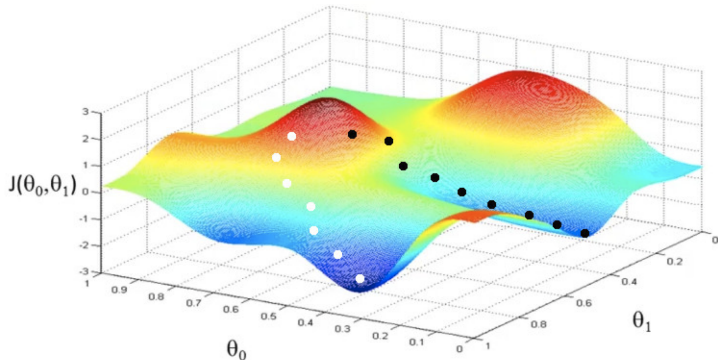
However, a lot of work has gone into organizing the the computations so they can be done efficiently and the method for computing the gradient is called *back propogation*.

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.

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

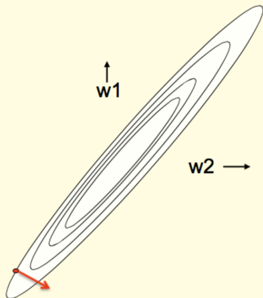
This picture illustrates going to different local minimums depending on the starting value.



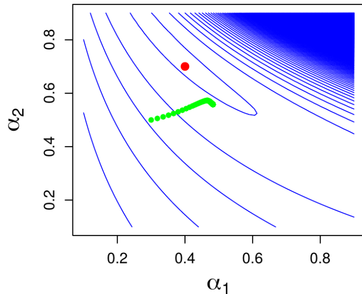
This picture gives the basic idea of how gradient descent could be much worse than Newton's method.

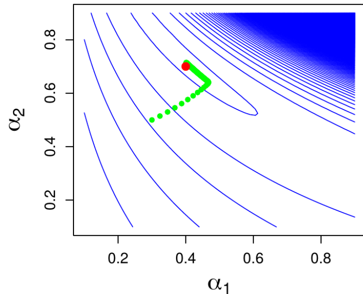
Why learning can be slow

- 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.



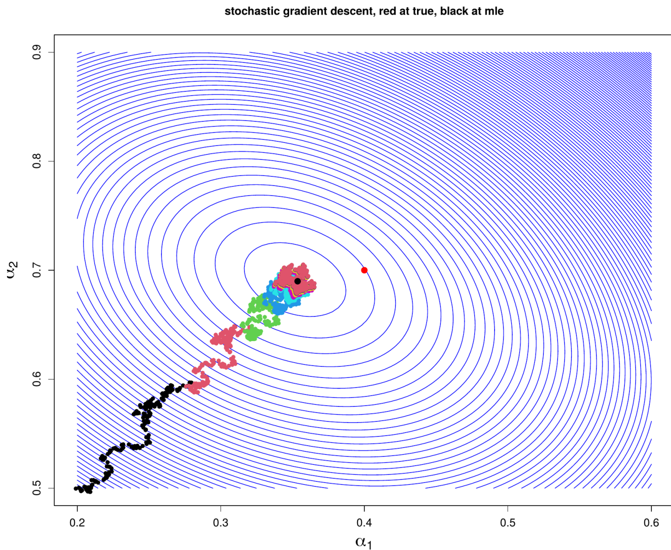
Gradient descent.





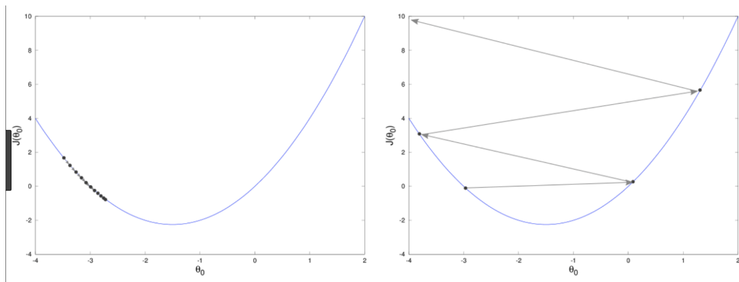
Path at left was l2 regularized.
Path at right was not.

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 example, in these schemes the learning rate is decreased as you iterate:

$$\epsilon_t = (1 - \frac{t}{\tau})\epsilon_o + \frac{t}{\tau}\,\epsilon_\tau, t = 1, 2, \ldots, \tau, \;\; \epsilon_t = \epsilon, t > \tau.$$

$$\epsilon_t = \epsilon_o \; \exp(-kt).$$

$$\epsilon_t = \frac{\epsilon_o}{1 + kt}$$

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.

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

where t indexes iteration.

RMSprop

$$A_{it} = \rho \, A_{i,t-1} + (1-\rho) \, (\frac{\partial L(\theta_{t-1})}{\partial \theta_i})^2$$

$$\theta_{it} = \theta_{i,t-1} - \frac{\alpha}{\sqrt{A_{it}}} \frac{\partial L(\theta_{t-1})}{\partial \theta_i}.$$

- Initialize A at 0.
- $\rho \in (0,1)$.
- Use $\sqrt{A_{it} + \varepsilon}$ instead of $\sqrt{A_{it}}$.

Note:

- learning rate is adjusted parameter by parameter (the i in θ_{it}).
- the A_{it} keep track of how big the partial derivative as been for an individual parameter and adjusts the learning rate down when they have been big.

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.

Network Tuning, ISLR 10.7.4

Has some general comments on "network tuning".

Have to think about:

- *The number of hidden layers, and the number of units per layer*.
  "Modern thinking is that the number of units per layer can be large, and overfitting can be controlled via the various forms of regularization."

- *Regularization tuning parameters.* Dropout, L1, L2, typically set separately at each layers (also early stopping).

- *Details of stochastic gradient descent*. batch size, number of epochs, learning rate.

If my number of layers/ number of unit architecture is too simple, I just won't be able to get a function complicated enough to capture the patterns in the data.

If I "overdo it" with a too complicated model, I can "reign it it" with regularization.

This is analogous to putting a lot of predictors in a linear model so that if you just ran the straight regression it would be ridiculous.

But, I can make it sensible with L1 or L2 regularization.

8. Cars Example with Deep Learning

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

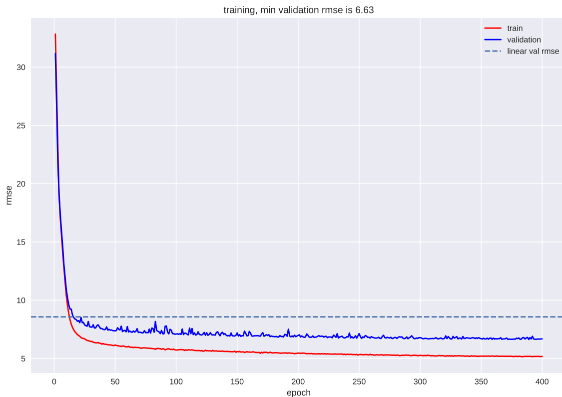
Note all the choices we have to make about model architecture, optimization, and regularization.

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 numeric 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.

Out of sample predictions.
ypednn 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

Let's do an example with a binary target y.

Two fundamental things we have to change from the way we did things with a numeric y are:

- transform the final output to (0,1) so it is a probability (like we did in logistic regression).
- use cross-entropy loss.

For example with a single layer we would transform the final ouput to a probability giving:

$$P(Y = 1 \,|, X, w, \beta) = F(\beta_0 + \sum_{k=1}^{K} \beta_k \, g(w_{k0} + \sum_{j=1}^{p} w_{kj} \, X_j))$$

where F is the sigmoid function and g is any activation function.

In the deep case, Z^L is the single output from the final layer and

$$P(Y = 1 \,|\, X, w, b) = F(Z^L).$$

IMDB Example

We want to classify movie reviews as positive or negative based on the text of the reviews.

We have 25,000 train and 25,000 test observations.

We use bag of words to convert the reviews to features

The data is processed so that there are 10,000 possible words.
Each review is then a list of integers giving the word ids in the list.

What is x?

For example the first train observation has 218 words in the bag. `train_data` contains the train "x".

```
In [27]: type(train_data)
Out[27]: numpy.ndarray

In [28]: type(train_data[0])
Out[28]: list

In [29]: len(train_data)
Out[29]: 25000

In [23]: len(train_data[0])
Out[23]: 218

#first 5 words in list for bag of words in first train x
In [24]: train_data[0][:5]
Out[24]: [1, 14, 22, 16, 43]

In [25]: max(train_data[0])
Out[25]: 7486

In [26]: min(train_data[0])
Out[26]: 1
```

```
In [62]: ## just print out the first nw words from the review
    ...: rid = 10 # review id to translate to words
    ...: for i in range(1,15):
    ...:     wid = train_data[rid][i]
    ...:     print(wid,reverse_word_index[wid-3]) #offset of 3
    ...:
785 french
189 horror
438 cinema
47 has
110 seen
142 something
7 of
6 a
7475 revival
120 over
4 the
236 last
378 couple
7 of
```

To use neural nets we need our x to be a numeric feature vector !!

We one-hot encode the bag of words.
Each term is a variable and the bags of words is captured by dummies for each term indicating whether that term is in the bag (document).

```
In [69]: def vseq(seq,dim=10000):
    ...:    res = np.zeros((len(seq),dim))
    ...:    for i, s in enumerate(seq):
    ...:       res[i,s] = 1
    ...:    return res
    ...:
    ...: xtr = vseq(train_data)
    ...: xte = vseq(test_data)
    ...:

In [70]: xtr.shape
Out[70]: (25000, 10000)

In [71]: xte.shape
Out[71]: (25000, 10000)

In [72]: xtr[:5,:4]
Out[72]:
array([[0., 1., 1., 0.],
       [0., 1., 1., 0.],
       [0., 1., 1., 0.],
       [0., 1., 1., 0.],
       [0., 1., 1., 0.]])
```

What is y?

The y labels are all 0 or 1 where 1 means a good review and 0 means a bad one.

The data has been chosen so that it is balanced in the sense that we have just as many good ones as bad ones in both the train and test data.

```
In [32]: print(pd.Series(train_labels).value_counts())
    ...: print(pd.Series(test_labels).value_counts())
    ...:
1    12500
0    12500
dtype: int64
0    12500
1    12500
dtype: int64
```

This does not change the structure of y = (train,test)_labels, but we convert to double.

```
## labels as 32 arrays
ytr = np.asarray(train_labels).astype('float32')
yte = np.asarray(test_labels).astype('float32')
print(pd.Series(ytr).value_counts())
print(pd.Series(yte).value_counts())
```

Our Model

We know have train/test x/y we can use.

We can see how we do with a neural net model.

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(16,activation='relu',input_shape=(xtr.shape[1],)))
model.add(keras.layers.Dense(16,activation='relu'))
model.add(keras.layers.Dense(1,activation='sigmoid'))
```

- our first layer is just the features with input dimension = number of columns in x.
- we add two hidden layers, each having 16 units.
- our output layer just has one output with sigmoid activation

We have to chose our loss function and some details about the stochastic gradient descent.

- our loss function to optimize is cross entropy.
- we use rsmprop to adaptively adjust the learning rate.
- we also choose an additional measure of loss to monitor which accuracy (% correct).

```
model.compile(optimizer='rmsprop',loss='binary_crossentropy',
   metrics=['accuracy'])
```

Let's split our train data into a train/val subset.
So we are using the three set approach.

```
## train val

nval = 10000

xval = xtr[:nval]
yval = ytr[:nval]

pxtr = xtr[nval:]
pytr = ytr[nval:]
```

So our 25,000 train is now 15,000 train and 10,000 validation.
Remember, we still have 25,000 test.

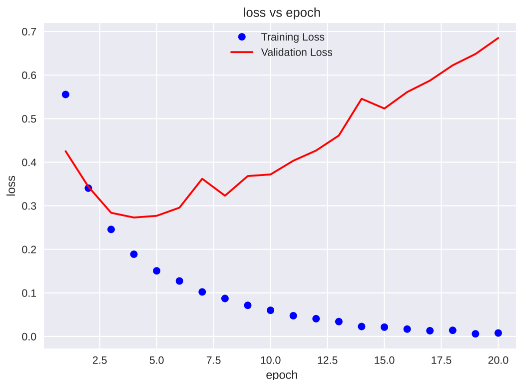
To actually do the optimization, I still have to choose:

- number of epochs.
- batch size within an epoch.

Keras will keep track of the loss on the train and validation data as the SGD iterates.

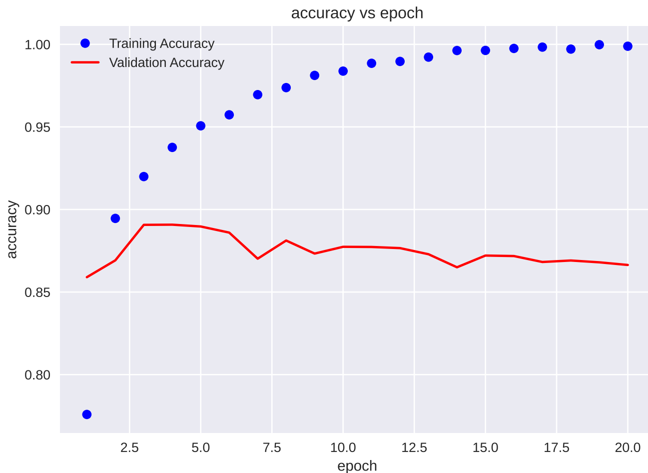
```
trh = model.fit(pxtr,pytr,epochs=20,batch_size=512,validation_data=(xval,yval))
```

In and out of sample loss (cross entropy) plotted versus epoch number.



Note that a single epoch has $15000/512 = 30$ moves.

In and out of sample accuracy plotted versus epoch number.



We refit using the full 25,000 train and just 4 epochs.

```
mod = keras.models.Sequential()

mod.add(keras.layers.Dense(16,activation='relu',input_shape=(xtr.shape[1],)))
mod.add(keras.layers.Dense(16,activation='relu'))
mod.add(keras.layers.Dense(1,activation='sigmoid'))

mod.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=['acc'])


# train on all the train
nepoch = 4
mod.fit(xtr,ytr,epochs=nepoch,batch_size=512)
# not evaluate on test data
res = mod.evaluate(xte,yte)
print(res)
```

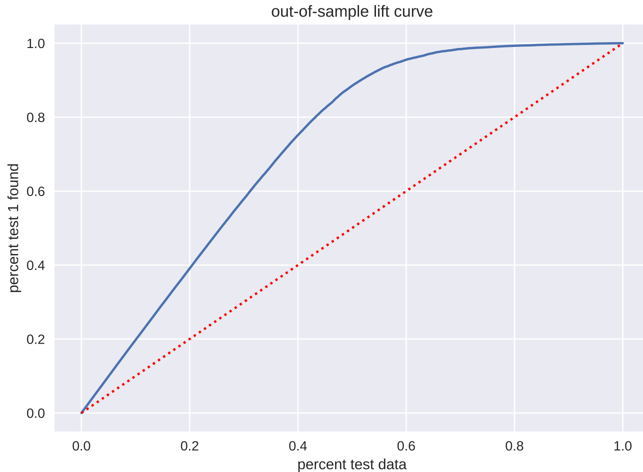
```
[0.29436346888542175, 0.8843600153923035]
```

88% accuracy on the test data.

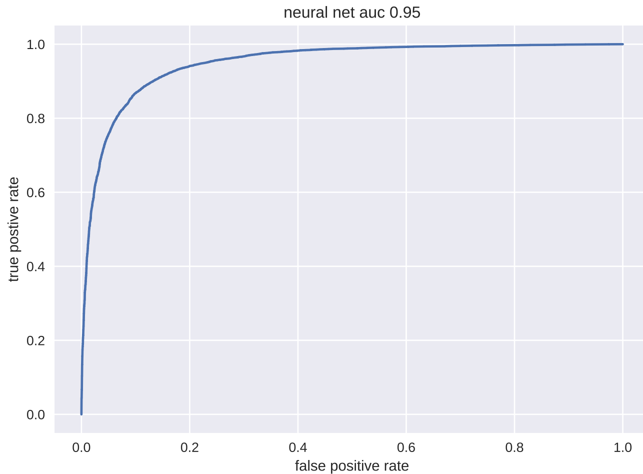
Out of sample (test) confusion matrix.

```
nte = len(yte)
ypred = np.zeros((nte,))
ypred[phat>.5]=1
cTab = pd.crosstab(yte,ypred)

cTab:

col_0    0.0    1.0
row_0
0.0    10942   1558
1.0     1333  11167

In [9]: np.diag(cTab.to_numpy()).sum()/nte
Out[9]: 0.88436
```

out of sample lift.

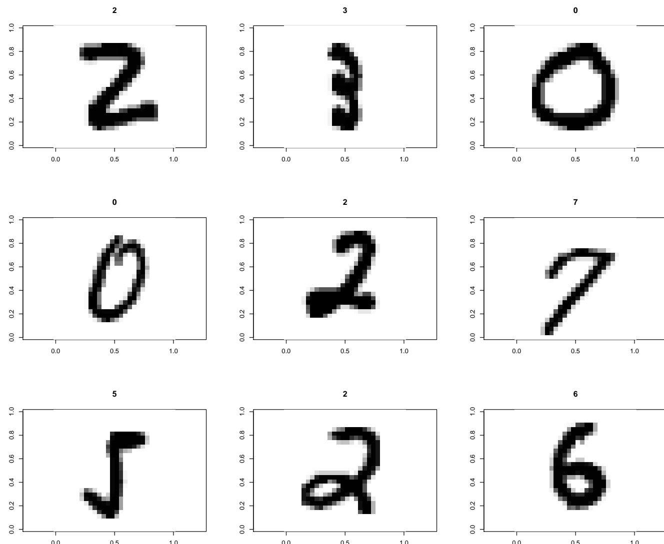


out of sample roc/auc.



10. Simple MNIST: multinoulli classification

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



Digit recognition:

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

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

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

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.

Remember, *tensors* in machine learning are just arrays. Your data could be an array with lots of dimensions !!

Another common array dimension in data is time.

In this section of notes we will just do a very simple neural net fit to the MNIST data.

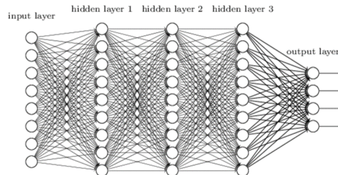
This is what is done in chapter 2 of “Deep Learning with Python” by Chollet (keras).

We are mostly using it as a nice example of a classification problem that is more than binary.

Computer vision is a huge area of application for neural net based models.
Many layers are used with very specific designs rather than just the basic dense networks we are looking at. For example, convolutional neural networks use very special kinds of layers tailored for the image recognition problem.

What do we have to change for a non-binary Multinoulli Outcome?

- If we have K possible categories, then we need a final output layer with K outcomes.
- We use the softmax function as our activation function for the output layer. This will map the K numeric outcomes to a probability vector.
- cross entropy loss.



softmax

$x \in R^p$. $x = (x_1, x_2, \ldots, x_p)$, $x_j \in R$.
$f : R^p \to R^p$.
$f(x) = (f_1(x), f_2(x), \ldots, f_p(x))$

$$f_j(x) = \frac{e^{x_j}}{\sum_{j=1}^{p} e^{x_j}}$$

```
In [1]: import numpy as np

In [2]: x = np.array([-1,0,2])

In [3]: pv = np.exp(x)

In [4]: pv
Out[4]: array([0.36787944, 1.        , 7.3890561 ])

In [5]: pv = pv/pv.sum()

In [6]: pv.sum()
Out[6]: 1.0

In [7]: pv
Out[7]: array([0.04201007, 0.1141952 , 0.84379473])
```

In the classic MNIST data we have 60,000 train and 10,000 test.

```
In [8]: print(pd.Series(train_labels).value_counts()/len(train_labels))
   ...: print(pd.Series(test_labels).value_counts()/len(test_labels))
   ...:
1    0.112367
7    0.104417
3    0.102183
2    0.099300
9    0.099150
0    0.098717
6    0.098633
8    0.097517
4    0.097367
5    0.090350
dtype: float64
1    0.1135
2    0.1032
7    0.1028
3    0.1010
9    0.1009
4    0.0982
0    0.0980
8    0.0974
6    0.0958
5    0.0892
dtype: float64

In [9]: len(train_labels)
Out[9]: 60000

In [10]: len(test_labels)
Out[10]: 10000
```

process x and y

We turn the 28x28 images into vectors of length 28^2.
We scale the [0,255] grayscale to [0,1].
We one-hot encode the targets.

```
In [12]: ###################################################
    ...: ## preparing the image data
    ...:
    ...: train_images = train_images.reshape((60000,28*28))
    ...: train_images = train_images.astype('float32')/255
    ...:
    ...: test_images = test_images.reshape((10000,28*28))
    ...: test_images = test_images.astype('float32')/255
    ...:
    ...: ###################################################
    ...: ## preparing the labels
    ...: print(train_labels.dtype)
    ...: train_labels = keras.utils.to_categorical(train_labels)
    ...: print(train_labels.dtype)
    ...: test_labels = keras.utils.to_categorical(test_labels)
    ...:
    ...: print(train_labels[:5])
uint8
float32
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

Given how we have setup the data, this is how we can make a very simple model in keras.

```
## the network architecture

network = keras.models.Sequential()
network.add(keras.layers.Dense(512,activation='relu',input_shape=(28*28,)))
network.add(keras.layers.Dense(10,activation='softmax'))
```

Just one hidden layer with 512 units.
relu activation.

Final layer has 10 units (one for each digit) and softmax activation.

- rmsprop for our adaptive learning rate.
- cross entropy for our loss.
- also monitor the accuracy.

```
##################################################
## the compilation step
network.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])
```

- 5 epochs.
- batch size 128.

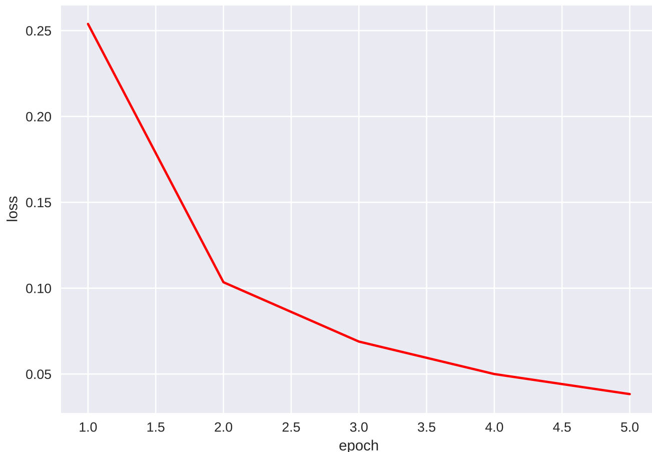
```
fhist = network.fit(train_images,train_labels, epochs = 5, batch_size = 128)
```

Let' check our performance on the test data.

```
In [16]: #################################################
    ...: ## loss
    ...:
    ...: test_loss, test_acc = network.evaluate(test_images,test_labels)
    ...: print('test_acc: ',test_acc)
    ...: print('test_loss: ',test_loss)
    ...:
313/313 [==============================] - 0s 863us/step - loss: 0.0665 - accur
test_acc:  0.9799000024795532
test_loss:  0.06653758883476257
```

Pretty amazing and this is a ridiculously simple model compared to the state of the art !!!

We could run it for more epochs and improve the in-sample fit (and possibly worse on test!!).



The ouf-of-sample confusion matrix.

```
In [43]: cTab
Out[43]:
col_0  0.0   1.0   2.0  3.0  4.0  5.0  6.0   7.0  8.0  9.0
row_0
0.0    974     0     0    1    0    1    1     1    2    0
1.0      0  1129     3    1    0    1    1     0    0    0
2.0      6     3  1004    3    1    0    2     6    7    0
3.0      0     0     4  993    0    4    0     5    2    2
4.0      2     0     2    1  960    1    2     3    1   10
5.0      2     0     0    9    1  874    2     1    1    2
6.0      6     3     0    1    4   10  933     1    0    0
7.0      1     5     6    2    0    0    0  1008    2    4
8.0      4     1     4    4    2   11    1     4  940    3
9.0      4     3     0    5    5    2    0     4    2  984

In [44]: np.diag(cTab.to_numpy()).sum()/nte
Out[44]: 0.9799

In [45]: print('test_acc: ',test_acc)
    ...: print('test_loss: ',test_loss)
    ...:
test_acc:  0.9799000024795532
test_loss:  0.06653758883476257
```

Simple MNIST Grid Search

I tried a simple grid search for the MNIST data in the software h2o.

Loosely following the advice from h2o and the book on h2o by Darren Cook, but not wanting to run for too long, I tried the following $2^5 = 32$ settings.

We'll try $2^5 = 32$ different deep neural net settings in our grid search.
Several hours on my portable workstation laptop.

```
> hyper_params
$hidden
$hidden[[1]]
[1] 200 200

$hidden[[2]]
[1] 300 300


$activation
[1] "TanhWithDropout"      "RectifierWithDropout"

$hidden_dropout_ratios
$hidden_dropout_ratios[[1]]
[1] 0.1 0.1

$hidden_dropout_ratios[[2]]
[1] 0.5 0.5


$l1
[1] 1e-04 1e-02

$max_w2
[1] 3.402823e+38 5.000000e+01
```

All 32 settings.

```
> expand.grid(hyper_params)
     hidden            activation hidden_dropout_ratios    l1       max_w2
1  200, 200       TanhWithDropout              0.1, 0.1 1e-04 3.402823e+38
2  300, 300       TanhWithDropout              0.1, 0.1 1e-04 3.402823e+38
3  200, 200 RectifierWithDropout               0.1, 0.1 1e-04 3.402823e+38
4  300, 300 RectifierWithDropout               0.1, 0.1 1e-04 3.402823e+38
5  200, 200       TanhWithDropout              0.5, 0.5 1e-04 3.402823e+38
6  300, 300       TanhWithDropout              0.5, 0.5 1e-04 3.402823e+38
7  200, 200 RectifierWithDropout               0.5, 0.5 1e-04 3.402823e+38
8  300, 300 RectifierWithDropout               0.5, 0.5 1e-04 3.402823e+38
9  200, 200       TanhWithDropout              0.1, 0.1 1e-02 3.402823e+38
10 300, 300       TanhWithDropout              0.1, 0.1 1e-02 3.402823e+38
11 200, 200 RectifierWithDropout               0.1, 0.1 1e-02 3.402823e+38
12 300, 300 RectifierWithDropout               0.1, 0.1 1e-02 3.402823e+38
13 200, 200       TanhWithDropout              0.5, 0.5 1e-02 3.402823e+38
14 300, 300       TanhWithDropout              0.5, 0.5 1e-02 3.402823e+38
15 200, 200 RectifierWithDropout               0.5, 0.5 1e-02 3.402823e+38
16 300, 300 RectifierWithDropout               0.5, 0.5 1e-02 3.402823e+38
17 200, 200       TanhWithDropout              0.1, 0.1 1e-04 5.000000e+01
18 300, 300       TanhWithDropout              0.1, 0.1 1e-04 5.000000e+01
19 200, 200 RectifierWithDropout               0.1, 0.1 1e-04 5.000000e+01
20 300, 300 RectifierWithDropout               0.1, 0.1 1e-04 5.000000e+01
21 200, 200       TanhWithDropout              0.5, 0.5 1e-04 5.000000e+01
22 300, 300       TanhWithDropout              0.5, 0.5 1e-04 5.000000e+01
23 200, 200 RectifierWithDropout               0.5, 0.5 1e-04 5.000000e+01
24 300, 300 RectifierWithDropout               0.5, 0.5 1e-04 5.000000e+01
25 200, 200       TanhWithDropout              0.1, 0.1 1e-02 5.000000e+01
26 300, 300       TanhWithDropout              0.1, 0.1 1e-02 5.000000e+01
27 200, 200 RectifierWithDropout               0.1, 0.1 1e-02 5.000000e+01
28 300, 300 RectifierWithDropout               0.1, 0.1 1e-02 5.000000e+01
29 200, 200       TanhWithDropout              0.5, 0.5 1e-02 5.000000e+01
30 300, 300       TanhWithDropout              0.5, 0.5 1e-02 5.000000e+01
31 200, 200 RectifierWithDropout               0.5, 0.5 1e-02 5.000000e+01
32 300, 300 RectifierWithDropout               0.5, 0.5 1e-02 5.000000e+01
```

Run the 32 settings in h2o:

```
gDNN = h2o.grid("deeplearning",
                 hyper_params=hyper_params,
                 x=x,y=y,training_frame=train,validation_frame=valid,
                 epochs=200)
```

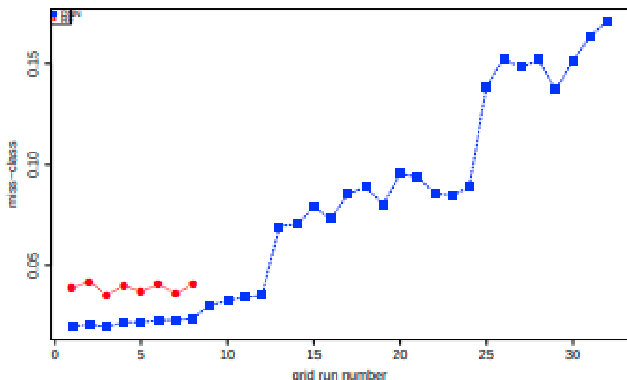
I also tried 8 random forests settings.

The best setting was:

```
ntrees,mtries,min_rows:  100 50 2

where we tried:

ntrees=c(100,500),
          mtries=c(28,50),
          min_rows=c(2,5))
```

Red loss for the 8 random forest setttings.
Blue is the loss for the 32 neural net settings.

The best nnet beats the best Random Forest.



The first 8 runs are the ones with l1 shrinkage $= 1e-04$, and $max_w2 = infinity$.

```
> expand.grid(hyper_params)
    hidden           activation hidden_dropout_ratios    l1       max_w2
1 200, 200      TanhWithDropout              0.1, 0.1 1e-04 3.402823e+38
2 300, 300      TanhWithDropout              0.1, 0.1 1e-04 3.402823e+38
3 200, 200 RectifierWithDropout              0.1, 0.1 1e-04 3.402823e+38
4 300, 300 RectifierWithDropout              0.1, 0.1 1e-04 3.402823e+38
5 200, 200      TanhWithDropout              0.5, 0.5 1e-04 3.402823e+38
6 300, 300      TanhWithDropout              0.5, 0.5 1e-04 3.402823e+38
7 200, 200 RectifierWithDropout              0.5, 0.5 1e-04 3.402823e+38
8 300, 300 RectifierWithDropout              0.5, 0.5 1e-04 3.402823e+38
```

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 numeric outcome
- squared error loss

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

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

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

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

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

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

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

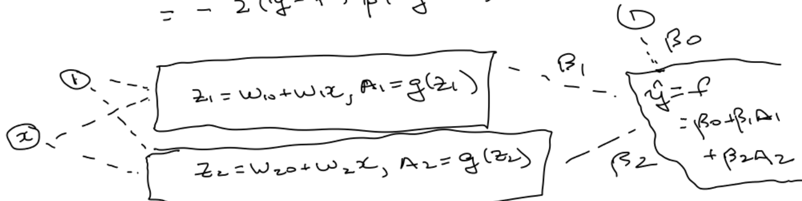
Drop: $L(y, f) = (y - f)^2$

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

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

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

$w_1 \downarrow z_1 \downarrow A_1 \downarrow f \downarrow L$



But how does this work in a general deep network?

The back propagation algorithm 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 it is composed of many basic linear (often called tensor) operations and the evaluations of the univariate activation function. Automatic differentiation uses the chain rule to compute the derivative given a chain of operations with known analytic derivatives.

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

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.

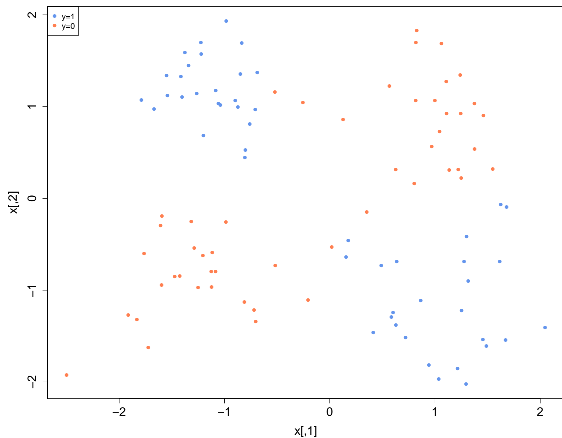
This is example is famous because it is a simple example where linear classification:

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

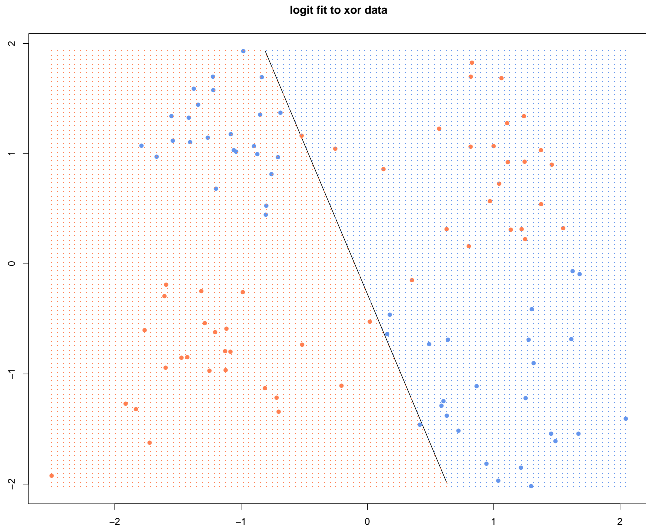
cannot work.

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

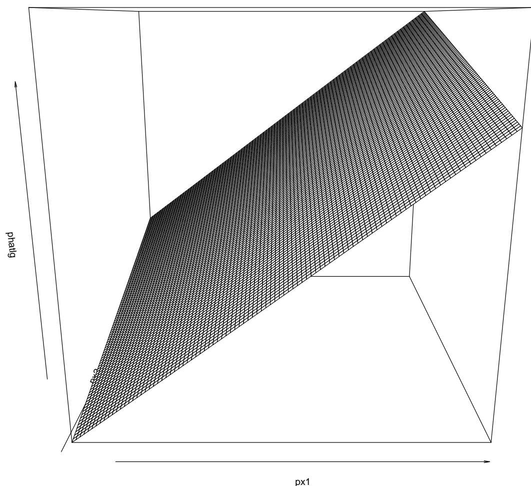
Here is a plot of the (simulated) data.



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.



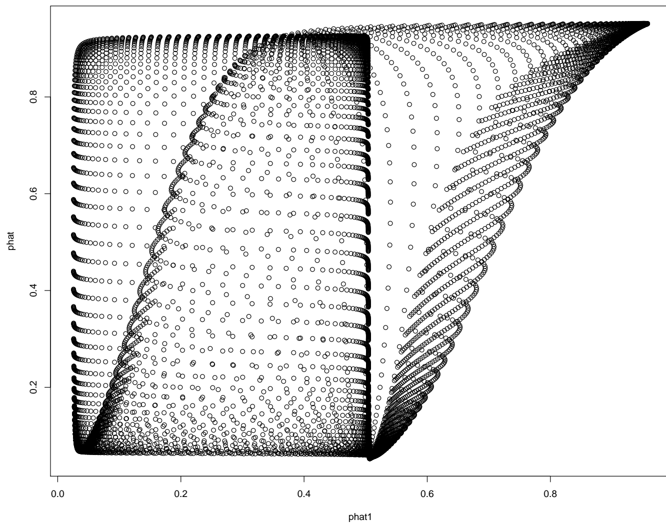
Really all the $\hat{p}$ are close to .5 !!

```
> 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
```

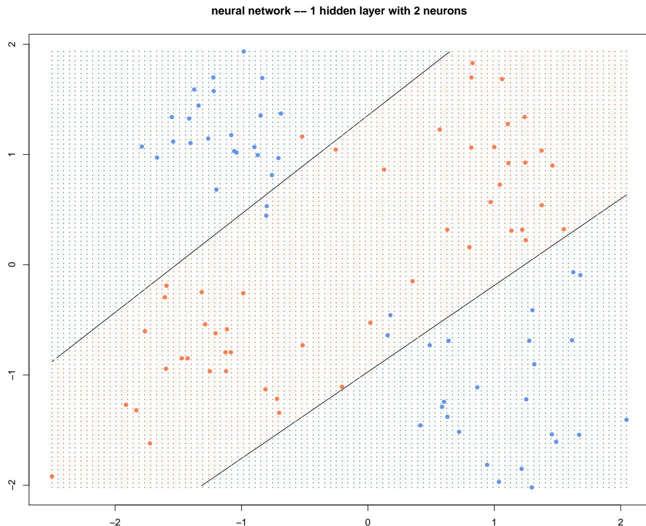
Let's try a nn fit.

```
#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.

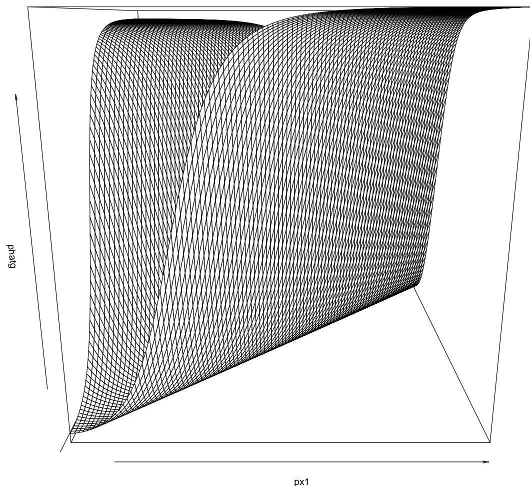


Here is the nn decision boundary (from the one that worked).



Beautiful !!!

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



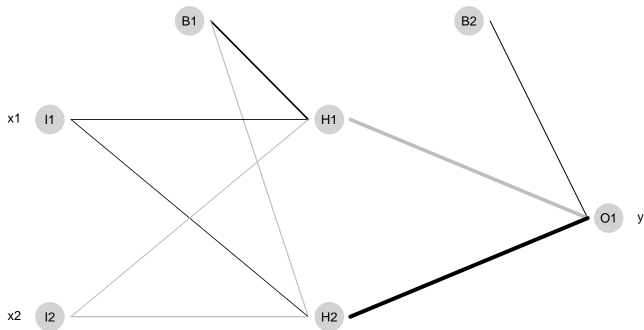
Obvious !!!!????

(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
```

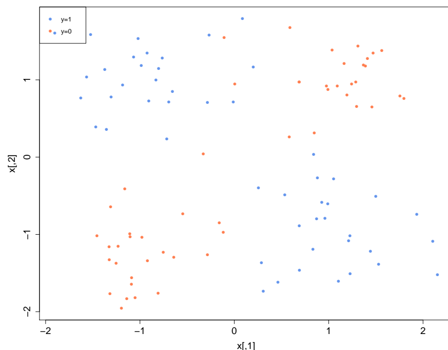
Basically uses $x_1 - x_2$!!!!.

A plot of `xnn`:



13. XOR Deep

Let's try some deep neural nets on the XOR example.

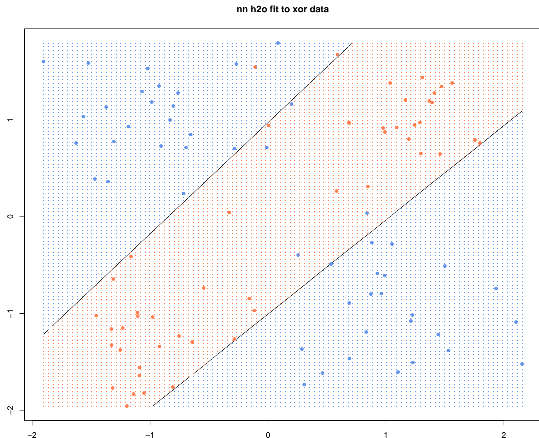


I did these with the h2o software.

A nn with one hidden layer having 2 neurons.
tanh activation.

```
#  1 hidden 2 neurons
model2 = h2o.deeplearning(x=1:2, y=3,
                          training_frame = dfh2o,
                          hidden = c(2),
                          activation = "Tanh",
                          epochs = 100000,
                          export_weights_and_biases = TRUE,
                          model_id = "xor.model2"
                          #use this if you want to get the same results
                          #seed=99,reproducible=TRUE
                          )

#phat on test
phat = h2o.predict(model2, dftest)
```

Here is a picture of the fit, looks good.



Note: *if I run it again I could get a solution with very little fit !!!!!!!!!!!!*

Let's look at the estimates:

```
> h2o.biases(model2, vector_id = 1)
         C1
1 -11.971459
2   6.656402

> h2o.weights(model2, matrix_id = 1)
        x1         x2
1 13.546680 -14.352386
2  7.834642  -7.051126
```

These are the coefficients going from the input x to the 2 neurons in the hidden layer.

These are the coefficents going from the hidden layer to the to the output.

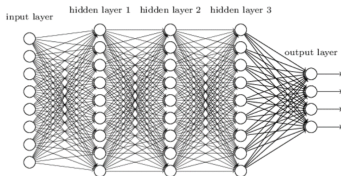
```
> h2o.biases(model2, vector_id = 2)
         C1
1 -3.094724
2  2.822581

[2 rows x 1 column]
> h2o.weights(model2, matrix_id = 2)
         C1        C2
1 -2.588578 -2.885383
2  3.261906 -5.987208
```

So, $O_1 = -3.094724 - 2.588578g(z_1) - 2.885383g(z_2)$.

Recall, we have a binary output so we get two outputs which are then softmaxed to get a probability vector.

Deep Features

Remember, in Machine Learning world the x's are called *the features*.



The last layer has the form

$$O_j = \beta_0 + \beta_{j1}\tilde{x}_1 + \beta_{j2}\,\tilde{x}_2 + \ldots + \beta_{jm}\tilde{x}_m$$

where m is the number of units in the final layer.

The $\tilde{x}_i$ are called the *deep features*.
They are nonlinear tranformations of the original x such that y is linearly predicted from them.

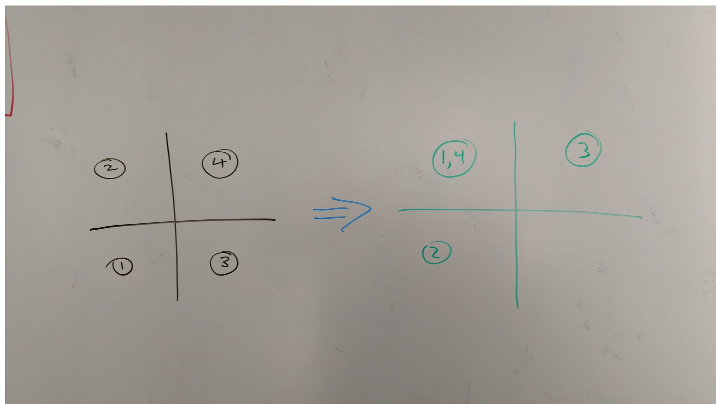
```
> tmp.df = as.h2o(data.frame(x1=c(-1, -1, 1, 1), x2=c(-1, 1, -1, 1)),
+                 destination_frame = "xor.4points")
  |======================================================================| 100%
> trans.features = h2o.deepfeatures(model2, tmp.df, layer = 1)
  |======================================================================| 100%
> as.matrix( h2o.cbind(tmp.df, trans.features) )
     x1 x2 DF.L1.C1   DF.L1.C2
[1,] -1 -1       -1  0.9999284
[2,] -1  1       -1 -0.9999988
[3,]  1 -1        1  1.0000000
[4,]  1  1       -1  0.9999980
```

```
> tanh(-11.971459 + 13.54668 -14.352386)
[1] -1
```

I evaluated the output of the first unit of the hidden layer inputing $(x_1, x_2) = (1, 1)$.

"trans.features" *t̃ransformed features*.

With the original features I can't linearly separate the 1's from the 0s'.



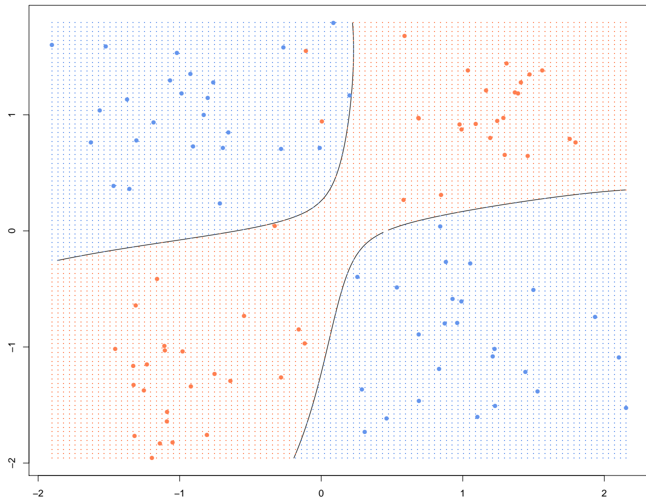
With the transformed features, I can.

Awesome.

This kind of example does make neural nets look magical.

Let's try one hidden layer with 10 units and L1 regularization:

```
model10R = h2o.deeplearning(x=1:2, y=3,
                            dfh2o,
                            hidden = c(10),
                            activation = "Tanh",
                            epochs = 100000,
                            export_weights_and_biases = TRUE,
                            l1 = 1e-2,
                            model_id = "xor.model10R"
)
```


nn h2o fit to xor data
−2
−1
0
1
2
−2
−1
0
1

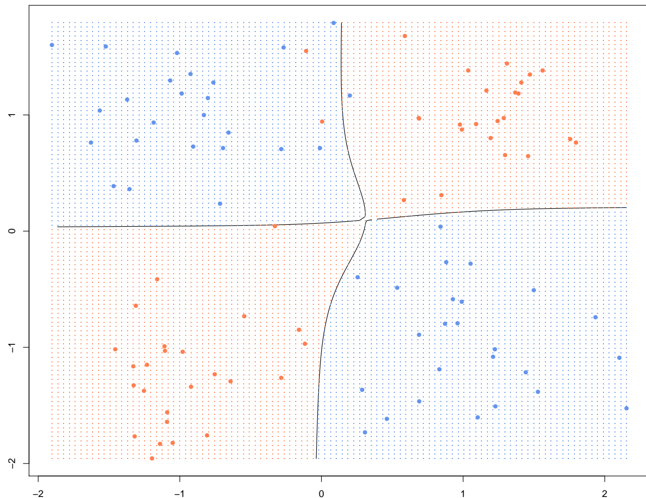
```
> h2o.weights(model10R, matrix_id = 1)
             x1            x2
1  0.0003904525 -0.0004619349
2  0.8172686100  0.9237694740
3 -0.0004394429  0.0000323276
4 -0.0005785795 -0.0005293362
5 -0.0004584224  0.0002672540
6  0.0004523931  0.0003855705

[10 rows x 2 columns]
> h2o.weights(model10R, matrix_id = 2)
            C1        C2            C3            C4            C5
1 0.0003701166 -1.297764 -0.0005743245 -2.533599e-04 -4.597708e-07
2 0.0001853947  1.382249  0.0002720330 -2.158213e-05 -7.892725e-05
             C6         C7         C8         C9           C10
1  0.0003624223  2.1420848  0.1726678 -2.6662343 -0.0001776762
2 -0.0005718899 -0.9607086 -2.9622021  0.5610269 -0.0002848183

[2 rows x 10 columns]
```

Ok, enough fooling around, let's get deep.

```
modelDR = h2o.deeplearning(x=1:2, y=3,
                           dfh2o,
                           hidden = c(3,4),
                           activation = "Tanh",
                           epochs = 100000,
                           export_weights_and_biases = TRUE,
                           l1 = 1e-2,
                           model_id = "xor.modelDR"
)
```

Two hidden layers.
First layer has 3 units, second layer has 4 units.

L1 regularization.
tanh activation.


nn h2o fit to xor data
1
0
−1
−2
−2
−1
0
1
2

$2 \rightarrow 3$

$3 \rightarrow 4$

$4 \rightarrow 2$

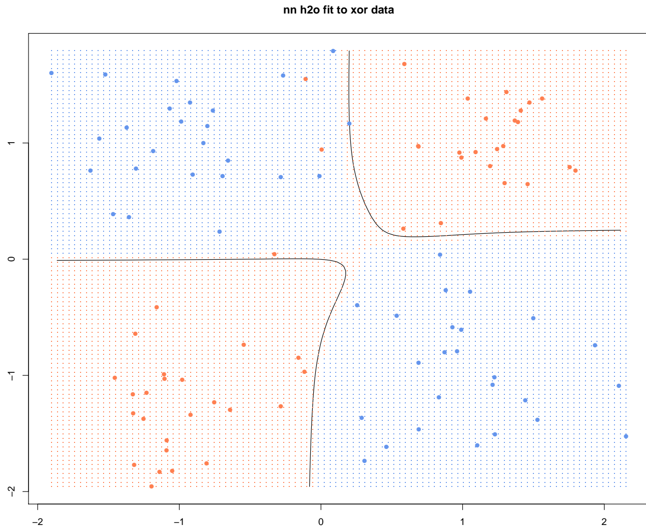
```
> h2o.weights(modelDR, matrix_id = 1)
             x1           x2
1  1.5895231962 -0.0305938125
2 -0.0004432469 -0.0001800873
3  0.0122436108 -2.0536758900

[3 rows x 2 columns]
> h2o.weights(modelDR, matrix_id = 2)
            C1            C2            C3
1 0.0100201899 -4.842104e-04  1.3224930763
2 1.7325805426  4.302524e-04  1.4567693472
3 0.0002096088 -5.461264e-05  0.0001562708
4 1.3153511286 -6.162645e-04 -1.0473971367

[4 rows x 3 columns]
> h2o.weights(modelDR, matrix_id = 3)
          C1        C2            C3         C4
1  0.9344926 -1.814222  0.0004014346  0.2660763
2 -3.0249763  2.523128 -0.0001543377 -3.8591626

[2 rows x 4 columns]
```

```
> h2o.performance(modelDR)
H2OBinomialMetrics: deeplearning
** Reported on training data. **
** Metrics reported on full training frame **

MSE:  0.02630829
RMSE:  0.1621983
LogLoss:  0.1133708
Mean Per-Class Error:  0.03
AUC:  0.9964
Gini:  0.9928

Confusion Matrix for F1-optimal threshold:
        0  1    Error     Rate
0      47  3 0.060000   =3/50
1       0 50 0.000000   =0/50
Totals 47 53 0.030000  =3/100

Maximum Metrics: Maximum metrics at their respective thresholds
                        metric threshold    value idx
1                       max f1  0.394910 0.970874  52
2                       max f2  0.394910 0.988142  52
3                 max f0point5  0.742596 0.975610  48
4                 max accuracy  0.742596 0.970000  48
5                max precision  0.981877 1.000000   0
6                   max recall  0.394910 1.000000  52
7              max specificity  0.981877 1.000000   0
8             max absolute_mcc  0.394910 0.941697  52
```

Let's try the "max f1" threshold.

```
contour(px1, px2, phatg, levels=0.3949, labels="", xlab="", ylab="",
        main= "nn h2o fit to xor data")
points(x, col=ifelse(g==1, "cornflowerblue","coral"),pch=16)
points(gd, pch=".", cex=1.5, col=ifelse(phatv>0.5, "cornflowerblue","coral"))
```

Get's 3 of the 0's (red) wrong.



```
> ## deep features
> tmp.df = as.h2o(data.frame(x1=c(-1, -1, 1, 1), x2=c(-1, 1, -1, 1)),
+                 destination_frame = "xor.4points")
  |======================================================================| 100%
> trans.features = h2o.deepfeatures(modelDR, tmp.df, layer = 2)
  |======================================================================| 100%
> as.matrix( h2o.cbind(tmp.df, trans.features) )
     x1 x2   DF.L2.C1   DF.L2.C2     DF.L2.C3   DF.L2.C4
[1,] -1 -1  0.8399254 -0.9172577 3.285519e-04 -0.9980520
[2,] -1  1 -0.8563508 -0.9996605 3.132807e-05 -0.9047109
[3,]  1 -1  0.8458606  0.9073782 7.016522e-04 -0.8108996
[4,]  1  1 -0.8505694 -0.8524170 4.033999e-04  0.6813285
```

An input (x_1, x_2) is mapped *nonlinearly* to a new x vector with 4 components.

The Deep NN has created nonlinear "deep features" that can be used to predict y more powerfully than the orginal x features.

Better than throwing in x^2 ?? !!

14. 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.

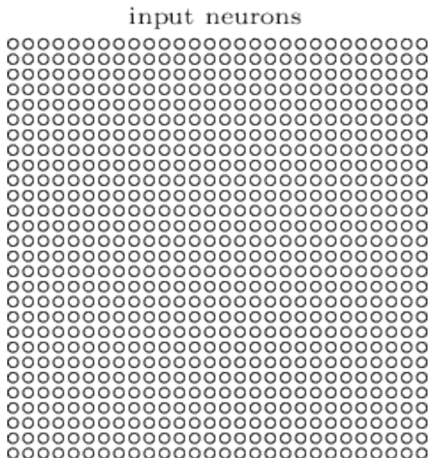
Besides the usual hidden layers we have looked at, different kinds of layers are used to take advantage of the spatial pixel structure.

Convolution layers replace a pixel value with a linear combination of nearby pixels.

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

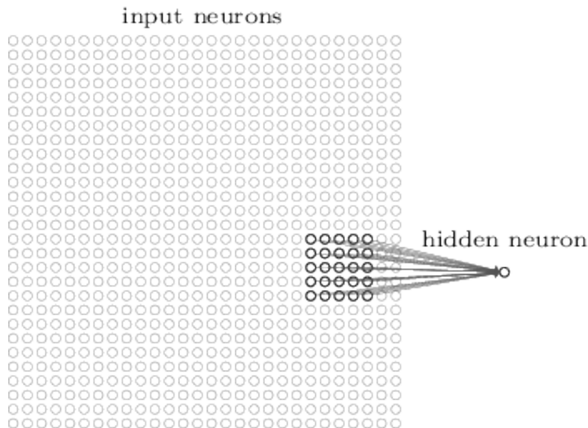
Convolution Layers:

Here is our 28^2 input layer:

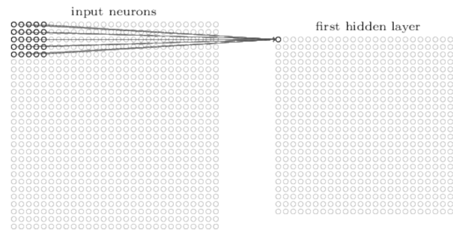


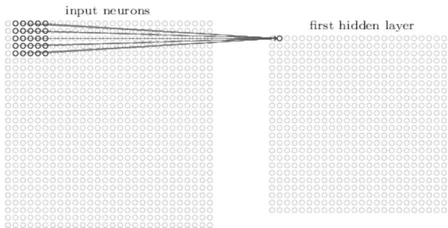
From: http://neuralnetworksanddeeplearning.com/chap6.html

To get single neuron for the next layer, take a linear combination 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 number of neighbors.
There are also “stride” parameters which determine how much you move the box around to get the “pixels” for the next layer.

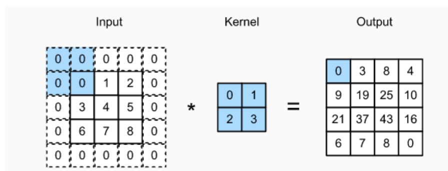




This will give an ouput layer a little smaller or about the same size depending on how you do it.

example

Start with 3x3.
2 x 2 filter.
pad edges with 0s.
end up with 4x 4.



$$\begin{bmatrix} 0 & 1 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 0\cdot 0 & 1\cdot 1 \\ 3\cdot 2 & 4\cdot 3 \end{bmatrix}$$

sum $\Rightarrow$ 19

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 1\cdot 0 & 2\cdot 1 \\ 4\cdot 2 & 5\cdot 3 \end{bmatrix}$$

sum $\Rightarrow$ 25

You use the same weights to combine neuron values in each subblock.

You learn the weights just like you learn the weights in a dense network.

You can use multiple convolutions (corresponding to different weights) associated with the same input layer.

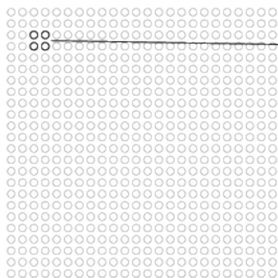
ISLR , pg 413.

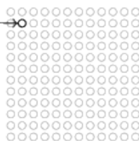
> *the convolved image highlights regions of the original image that resemble the convolution filter.*

By “convolution filter” the mean a choice of weights.

Pooling Layer:

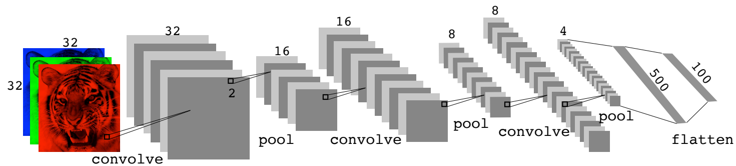
A pooling layer replaces the pixel values in non-overlapping regions with the maxiumum value.





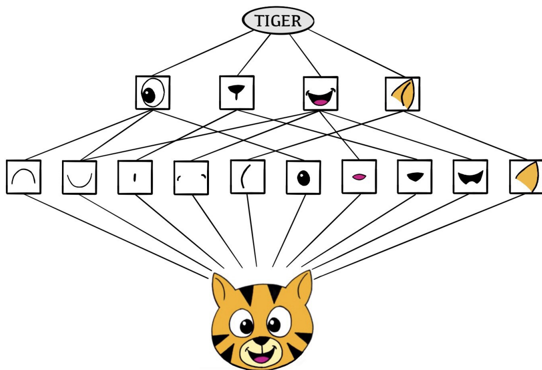
This will typically reduce the number or neurons in the next layer. The pooling layer “introduces an elmement of local translation invariance” (Efron and Hastie).

ISLR figure 10.8.



The dream of deep networks is that the layers build up an understanding of the feature vector x in a hierarchical manner in which each layer reinterprets the representation from the previous layers.

We can see this dramatically in convolutional networks where the pixel structure is (mostly) retained as we build the convolutional layers.



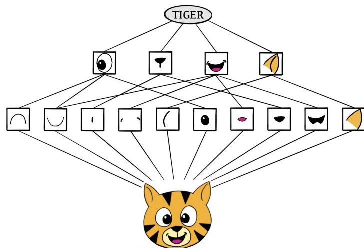
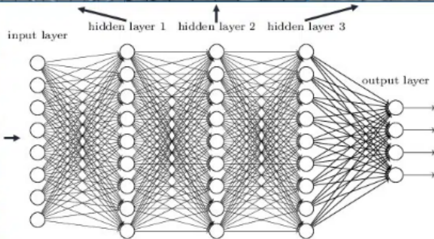


Figure 10.6 in ISLR

The network takes in the image and identifies local features. It then combines the local features in order to create compound features, which in this example includes eyes and ears. These compound features are used to ouput the label "tiger".

https://www.rsipvision.com/exploring-deep-learning/

Deep neural networks learn hierarchical feature representations



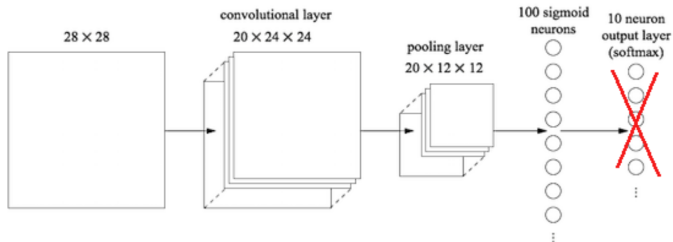
Another cool idea:

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.

A true story

A student did a thesis where
$x = (f(x_1), f(x_2), \dots, f(x_n))$ looks like an evaluation of a function of a grid of points x_j.

Y is disease status of a patient.

The student's idea was that the derivative $f'(x_j)$ should be what you use to predict Y and used this with machine learning techniques.

e.g KNN with the distance between patients a and b given by
$||f_a'(x) - f_b'(x)||$

where $f_a'(x)$ is the derivative computed from the observed function values for patient a.

I thought

“ a two layer deep neural net should be able to get this since the first layer can compute the derivative from the function values and then the second layer can figure out what to do with the derivative values.”

Tried it and it bombed.

Increased the L1 regularization and it beat anything the student had done.

15. Recurrent Neural Networks

This section follows 10.5 of ISLR closely.

Most of our modeling has assumed that the order of the observations is not meaninful as is the case with a random sample from some population.

In many problems our observations are *sequences* where the order in which things happen is a fundamental part of the process being modeled.

- weekly returns on a stock.
- a sequence of words in a document.
- daily temperatures.

We want our model to reflect the order in a sequence of observations.

Let $X = (X_1, X_2, \ldots, X_\ell, \ldots, X_L)$ be a sequence.
Hence X_ℓ is the ℓth observed X in the sequence.

X is our *input*.

For example X_ℓ could be the one hot encoding of the ℓth word in a document given a dictionary (list of possible words).

Our output is Y which may also be a sequence or just a scalar.

In our basic version of a *Recurrent Neural Network* (**RNN**) we will also have *activations* $(A_1, A_2, \ldots, A_\ell, \ldots, A_L)$.

Each X_ℓ and each A_ℓ can be a vector.

$$X_\ell^T = (X_{\ell 1}, X_{\ell 2}, \ldots, X_{\ell p}).$$
$$A_\ell^T = (A_{\ell 1}, A_{\ell 2}, \ldots, A_{\ell K}).$$

So, p is the dimension of X and K is the dimension of A.

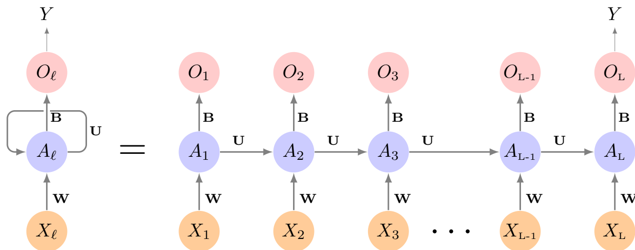
The X_ℓ are our *observed* sequences.

The A_ℓ will the a vector of units which will be nonlinear functions of the X designed to capture the X information in a way that helps us predict Y.

So the A our analogous to the the hidden units in a simpler set of dense layers but they have a special structure to capture the sequential nature of our system.

We have weights w_{kj} determining the dependence of $A_{\ell k}$ on $X_{\ell j}$ and weights u_{ks} determining the dependence of $A_{\ell k}$ and $A_{\ell-1s}$.

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right)$$



With output O_ℓ:

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}.$$

A_ℓ:

- information from past X_ℓ is passed along throught the past dependence of A_ℓ on $A_{\ell-1}$.
- information from current X_ℓ is put directly into A_ℓ.

Remember, in this simple version, an observation consists of:

$$(X_1, X_2, \ldots, X_\ell, \ldots, X_L), \quad Y.$$

- g is an activation function such as RELU.
- for a categorical Y an activation (e.g sigmoid for binary) can be applied to O.
- note that the weights W, B, U are the same throughout the sequence (*weight sharing*).

The key idea is that the A_ℓ capture all the information from past observed X.

X_1 affects O_3 through

$$X_1 \to A_1 \to A_2 \to A_3 \to O_3$$

pg 423. ... the activations A_ℓ accumulate a history of what has been seen before....

loss

For single observed $X = (X_1, X_2, \ldots, X_L)$ sequence and numeric Y we could use our usual loss

$$(Y - O_L)^2$$

Note that here Y is a single observed number and X is a $p \times L$ matrix.

For a data set of observed (x_i, y_i) we sum the loss as usual:

$$\sum_{i=1}^{n} (y_i - O_{iL})^2$$

where O_{iL} depends on x_i each of which is $p \times L$.

In some problems the loss may depend on the sequence of O_ℓ rather than the last one.
For example if we are translating a sentence.

time and space

The convolution layers for images moved about the the image computing a weighted combination of spatially close by pixels using the same weights.

The RNN uses a fixed weight U to see how the present A_ℓ is affected by the close by (previous) $A_{\ell-1}$.

The above is a basic RNN.
When used at "full strength" recurrent neural networks can be quite complex.

Example: Sentiment Analysis

Previously we looked a the IMDB example where we tried to predict the "sentiment of a review" (good or bad) from the text of the review.

We used a "bag of words" approach to try to capture the information in the review.

We had:

- x: vector of binary indicators, each element is 1 if the corresponding word is in the review and 0 otherwise.
- y: 1 if the review is good, 0 otherwise.

We had $p = 10,000$ dimension for x which is the number of terms (words).

Obviously, bag of words loses something from the text that a human could readily perceive.

How can we use the sequencing of the words in the review to make a prediction??

word2vec:

In our bag of words approach, we use binary indicators to tell us which of 10,000 possible words are in a review.

In our sequential RNN approach, we could use dummies to say what each word is in a sequence.

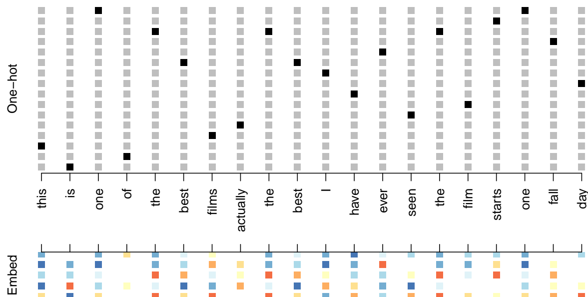
This would give us $p = 10,000$.

`word2vec` and `GloVe` map each word to a lower dimensional real vector in the "embedding space".

The goal is to do this in such a way that ".. words in the embedding space preserve semantic meaning; e.g. synonyms should appear near each other."

one hot word encoding and embedding

16 possible words.
document of 20 words.
embedding dimension $= 5$.



data representation

Each document (and the sentiment) is an observation.

Use last L words of the review.
Documents which are shorter than L, get padded with zeros upfront.

For each document $X = (X_1, X_2, \ldots, X_L)$ are the embedding vector reresentations of the corresponding word.

For each document Y is the sentiment.

ISLR tried:

Embedding dimension = 32.

K=32 hidden units.

Model trained with dropout regularization on the 25,000 train.

Got "a disappointing" 76% on test.

Network using GloVe embeddings did slightly worse.

Note

See page 426 of ISLR.

We are working with a simple RNN.

Long term and short term (LSTM) versions are more complex and keep track of more recent information *and* information from the more distant past rather than relying on the simple $A_{\ell-1} \rightarrow A_{\ell}$ transmission.

ISLR got 87% using LSTM.

State of the art is 95% !!!

Example: Time Series Prediction

We have three daily time series from December 3, 1962 to December 31, 1986.

Each trading day we measure:

- v: log trading volume:
  This is the fraction of all outstanding shares that are traded on that day, relative to a 100 day moving average of past turnover, on the log scale.

- r: Dow Jones return:
  difference between the log of the Dow Jones Industrial Index on consecutive trading days.

- z: Log volatility:
  This is based on the absolute value of daily price movements.

So, our data is:

$$(v_t, r_t, z_t), \; t = 1, 2, \ldots, T = 6,051.$$

Goal: predict v_t (log volume) from past observations of all three series.

Train/Test ??

We can't just do a random split of observations into train and test!!!!

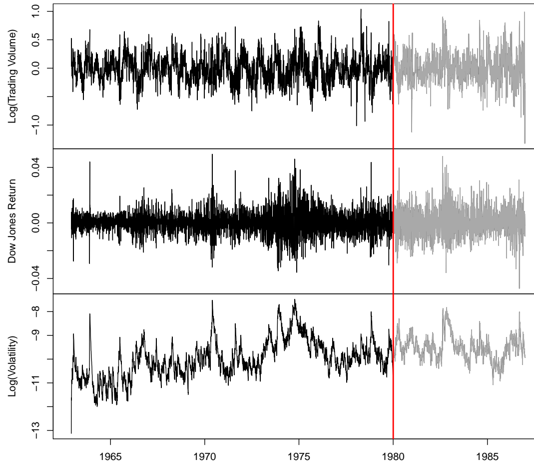
You have to predict the future from past information !!!

A simple approach:

Fit a model on the 4,281 observations up to January 2, 1980.

Use the model to predict v_t on all subsequent days.

Time series plots of our three series.



Before the red line is train, after is test.

How do we represent this problem in terms of our basic RNN structure?

For each $t \in L+1, L+2, \ldots, T$ we have on observation of the form:

$$X_1 = \begin{bmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{bmatrix}, \; X_2 = \begin{bmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{bmatrix}, \ldots, X_{L-1} = \begin{bmatrix} v_{t-2} \\ r_{t-2} \\ z_{t-2} \end{bmatrix} X_L = \begin{bmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{bmatrix}$$

$$Y = v_t$$

Note:

In time series, $(v_{t-l}, r_{t-l}, z_{t-l})$ are the value at *lag* l.

We use lagged values (*the past*) to predict the current value (*the future*).

$y = v_t$

O_ℓ

↑ B

$A_\ell = \begin{bmatrix} A_{\ell 1} \\ A_{\ell 2} \\ \vdots \\ A_{\ell k} \end{bmatrix}$

ℓ

↑ W

$X_\ell = \begin{bmatrix} v_j \\ r_j \\ z_j \end{bmatrix}$ $j^* = t - L + \ell - 1$

* $\ell = L$

$j = t - L + \ell - 1$

$= t - 1$

$\ell = 1$

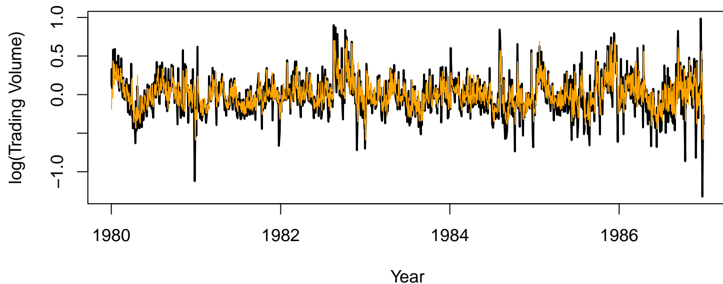
$j = t - L$

ISLR tried the model specification:

$L = 5$.
Use the values of the three series from the past 5 trading days.

$K = 12$ activations or hidden units (the dimension of each A_ℓ).

Note that with $L = 5$ we have 6,046 observations instead of 6,051 since we have to start with the first day for which we have the 5 lags.



Does not look bad!

Achieved an (out of sample) R^2 of .42 on the test data.

16. Don't worry, man vs machine

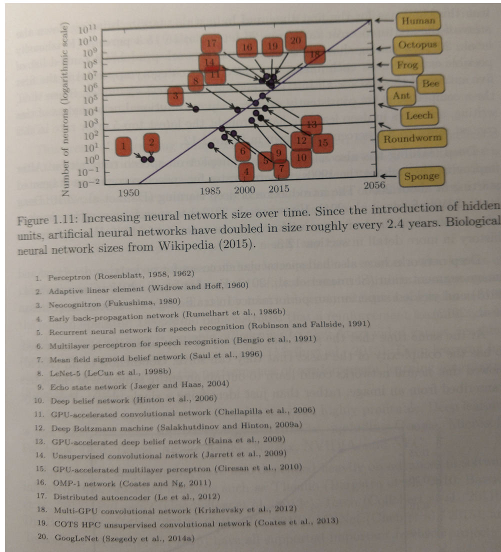


Figure 1.11: Increasing neural network size over time. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton et al., 2006)
11. GPU-accelerated convolutional network (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina et al., 2009)
14. Unsupervised convolutional network (Jarrett et al., 2009)
15. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU convolutional network (Krizhevsky et al., 2012)
19. COTS HPC unsupervised convolutional network (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)