

Revolutions

Daily news about using open source R for big data analysis, predictive modeling, data science, and visualization since 2008

July 18, 2017

Neural Networks from Scratch, in R

By Ilia Karmanov, Data Scientist at Microsoft

This post is for those of you with a statistics/econometrics background but not necessarily a machine-learning one and for those of you who want some guidance in building a neural-network from scratch in R to better understand how everything fits (and how it doesn't).

Andrej Karpathy wrote that when CS231n (Deep Learning at Stanford) was offered:

"we intentionally designed the programming assignments to include explicit calculations involved in backpropagation on the lowest level. The students had to implement the forward and the backward pass of each layer in raw numpy. Inevitably, some students complained on the class message boards".

Why bother with backpropagation when all frameworks do it for you automatically and there are more interesting deep-learning problems to consider?

Nowadays we can literally train a full neural-network (on a GPU) in 5 lines.

```
import keras
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer=RMSprop())
model.fit()
```

Karpathy, abstracts away from the "intellectual curiosity" or "you might want to improve on the core algorithm later" argument. His argument is that the calculations are a leaky abstraction:

"it is easy to fall into the trap of abstracting away the learning process—believing that you can simply stack arbitrary layers together and backprop will 'magically make them work' on your data"

Hence, my motivation for this post is two-fold:

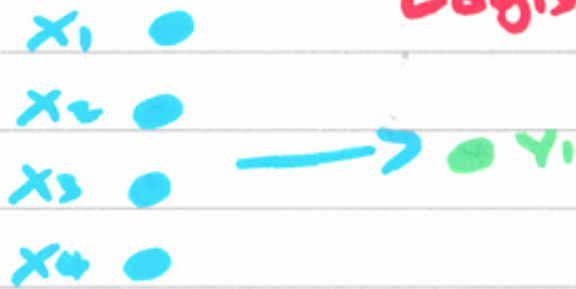
1. Understanding (by writing from scratch) the leaky abstractions behind neural-networks dramatically shifted my focus to elements whose importance I initially overlooked. If my model is not learning I have a better idea of what to address rather than blindly wasting time switching optimisers (or even frameworks).
2. A deep-neural-network (DNN), once taken apart into lego blocks, is no longer a black-box that is inaccessible to other disciplines outside of AI. It's a combination of many topics that are very familiar to most people with a basic knowledge of statistics. I believe they need to cover very little (just the glue that holds the blocks together) to get an insight into a whole new realm.

Starting from a linear regression we will work through the maths and the code all the way to a deep-neural-network (DNN) in the accompanying R-notebooks. Hopefully to show that very little is actually new information.



Linear Reg & ②

Logistic Reg



③

Softmax Reg



④

Neural Net

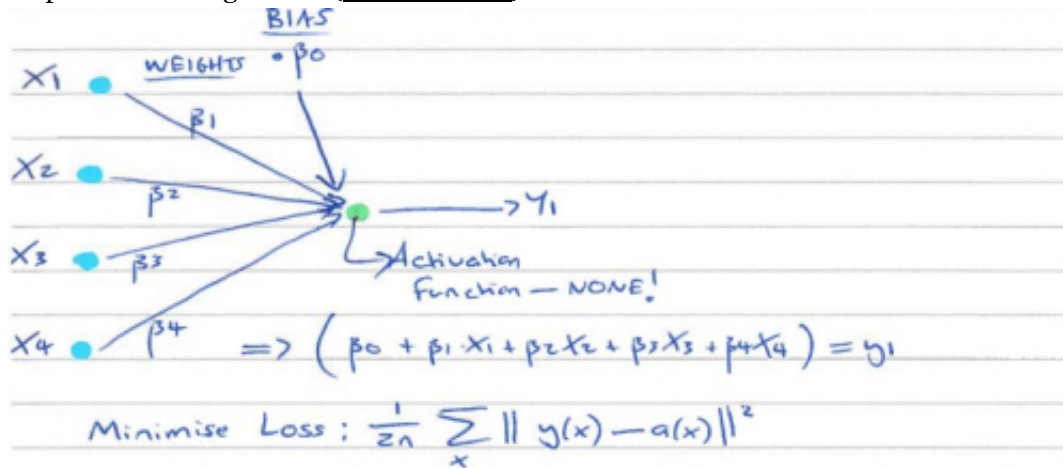


⑤

Convolutional Neural Net



Step 1 - Linear Regression (See Notebook)



Implementing the closed-form solution for the Ordinary Least Squares estimator in R requires just a few lines:

```
# Matrix of explanatory variables
X <- as.matrix(X)
# Add column of 1s for intercept coefficient
intcpt <- rep(1, length(y))
# Combine predictors with intercept
X <- cbind(intcpt, X)
# OLS (closed-form solution)
beta_hat <- solve(t(X) %*% X) %*% t(X) %*% y
```

The vector of values in the variable `beta_hat` define our "machine-learning model". A linear regression is used to predict a continuous variable (e.g. how many minutes will this plane be delayed by). In the case of predicting a category (e.g. will this plane be delayed - yes/no) we want our prediction to fall between 0 and 1 so that we can interpret it as the probability of observing the respective category (given the data).

When we have just two mutually-exclusive outcomes we would use a binomial logistic regression. With more than two outcomes (or "classes"), which are mutually-exclusive (e.g. this plane will be delayed by less than 5 minutes, 5-10 minutes, or more than 10 minutes), we would use a multinomial logistic regression (or "softmax"). In the case of many (n) classes that are not mutually-exclusive (e.g. this post references "R" and "neural-networks" and "statistics"), we can fit n-binomial logistic regressions.

An alternative approach to the closed-form solution we found above is to use an iterative method, called Gradient Descent (GD). The procedure may look like so:

- Start with a random guess for the weights
- Plug guess into loss function
- Move guess in the opposite direction of the gradient at that point by a small amount (something we call the 'learning-rate')
- Repeat above for N steps

GD only uses the Jacobian matrix (not the Hessian), however we know that when we have a convex loss, all local minima are global minima and thus GD is guaranteed to converge to the global minimum.

The loss-function used for a linear-regression is the Mean Squared Error:

$$C = \frac{1}{2n} \sum_x (y(x) - a(x))^2$$

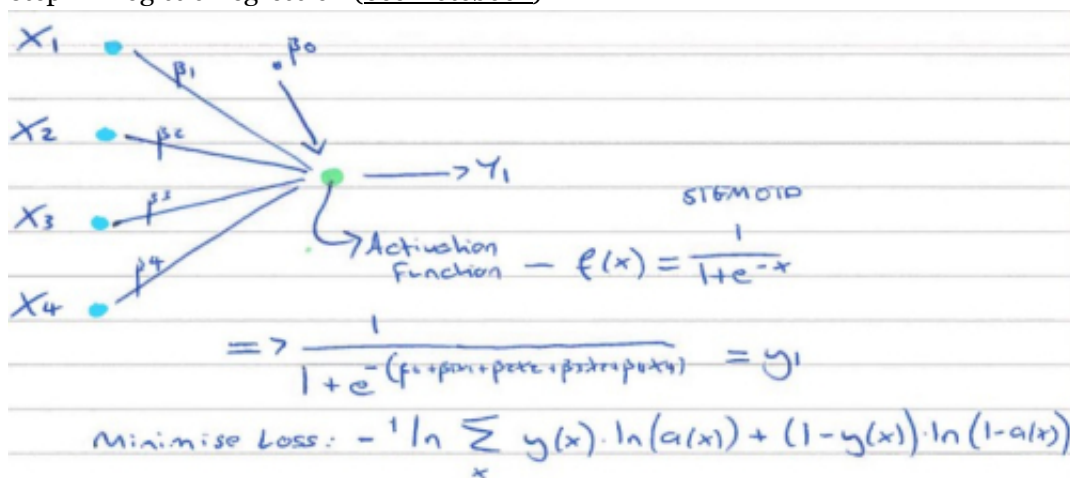
To use GD we only need to find the partial derivative of this with respect to `beta_hat` (the 'delta'/gradient).

This can be implemented in R, like so:

```
# Start with a random guess
beta_hat <- matrix(0.1, nrow=ncol(X_mat))
# Repeat below for N-iterations
for (j in 1:N)
{
  # Calculate the cost/error (y_guess - y_truth)
  residual <- (X_mat %*% beta_hat) - y
  # Calculate the gradient at that point
  delta <- (t(X_mat) %*% residual) * (1/nrow(X_mat))
  # Move guess in opposite direction of gradient
  beta_hat <- beta_hat - (lr*delta)
}
```

Running this for 200 iterations gets us to same gradient and coefficient as the closed-form solution. Aside from being a stepping stone to a neural-network (where we use GD), this iterative method can be useful in practice when the the closed-form solution cannot be calculated because the matrix is too big to invert (to fit into memory).

Step 2 - Logistic Regression (See Notebook)



A logistic regression is a linear regression for binary classification problems. The two main differences to a standard linear regression are:

1. We use an 'activation'/link function called the logistic-sigmoid to squash the output to a probability bounded by 0 and 1
2. Instead of minimising the quadratic loss we minimise the negative log-likelihood of the Bernoulli distribution

Everything else remains the same.

We can calculate our activation function like so:

```
| sigmoid <- function(z){1.0/(1.0+exp(-z))}
```

We can create our log-likelihood function in R:

```
log_likelihood <- function(X_mat, y, beta_hat)
{
  scores <- X_mat %*% beta_hat
  ll <- (y * scores) - log(1+exp(scores))
  sum(ll)
}
```

This loss function (the logistic loss or the log-loss) is also called the cross-entropy loss. The cross-entropy loss is basically a measure of 'surprise' and will be the foundation for all the following models, so it is worth examining a bit more.

If we simply constructed the least-squares loss like before, because we now have a non-linear activation function (the sigmoid), the loss will no longer be convex which will make optimisation hard.

$$C = \frac{1}{2n} \sum_x (y(x) - a(x))^2$$

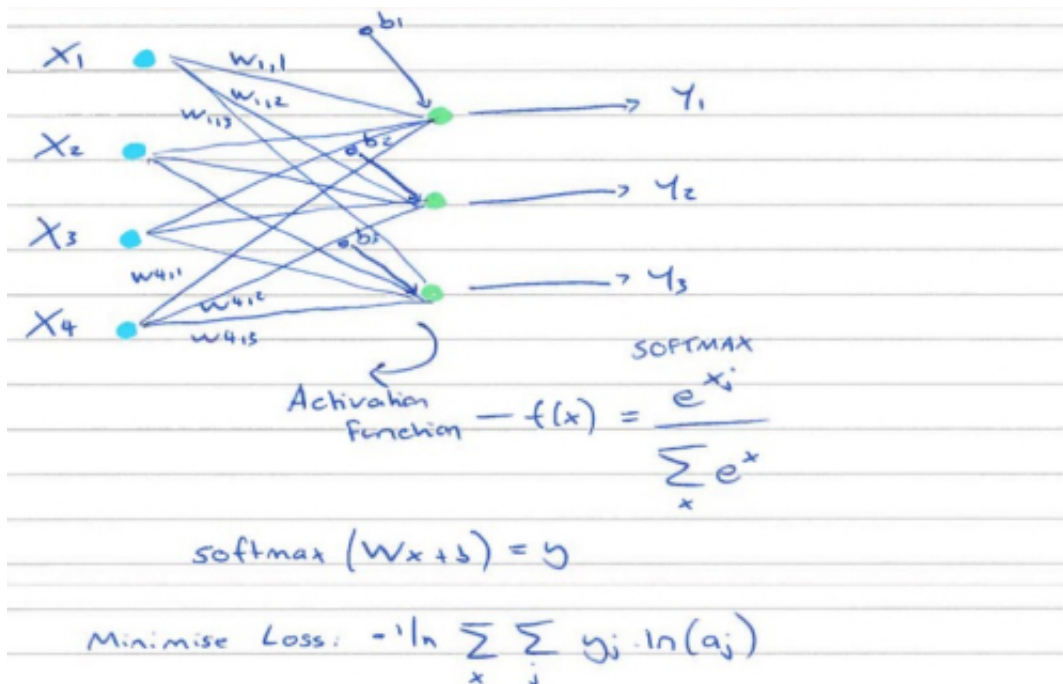
We could construct our own loss function for the two classes. When $y = 1$, we want our loss function to be very high if our prediction is close to 0, and very low when it is close to 1. When $y = 0$, we want our loss function to be very high if our prediction is close to 1, and very low when it is close to 0. This leads us to the following loss function:

$$C = -\frac{1}{n} \sum_x y(x) \ln(a(x)) + (1 - y(x)) \ln(1 - a(x))$$

The delta for this loss function is pretty much the same as the one we had earlier for a linear-regression. The only difference is that we apply our sigmoid function to the prediction. This means that the GD function for a logistic regression will also look very similar:

```
logistic_reg <- function(X, y, epochs, lr)
{
  X_mat <- cbind(1, X)
  beta_hat <- matrix(1, nrow=ncol(X_mat))
  for (j in 1:epochs)
  {
    # For a linear regression this was:
    # 1*(X_mat %*% beta_hat) - y
    residual <- sigmoid(X_mat %*% beta_hat) - y
    # Update weights with gradient descent
    delta <- t(X_mat) %*% as.matrix(residual, ncol=nrow(X_mat))*(1/nrow(X_mat))
    beta_hat <- beta_hat - (lr*delta)
  }
  # Print log-likelihood
  print(log_likelihood(X_mat, y, beta_hat))
  # Return
  beta_hat
}
```

Step 3 - Softmax Regression (No Notebook)



A generalisation of the logistic regression is the multinomial logistic regression (also called 'softmax'), which is used when there are more than two classes to predict. I haven't created this example in R, because the neural-network in the next step can reduce to something similar, however for completeness I wanted to highlight the main differences if you wanted to create it.

First, instead of using the sigmoid function to squash our (one) value between 0 and 1:

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

We use the softmax function to squash the sum of our n values (for n classes) to 1:

$$\phi(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

This means the value supplied for each class can be interpreted as the probability of that class, given the evidence. This also means that when we see the target class and increase the weights to increase the probability of observing it, the probability of the other classes will fall. The implicit assumption is that our classes are mutually exclusive.

Second, we use a more general version of the cross-entropy loss function:

$$C = -\frac{1}{n} \sum_x \sum_j y_j \ln(a_j)$$

To see why, remember that for binary classifications (previous example) we had two classes: $j = 2$, under the condition that the categories are mutually-exclusive $\sum_j a_j = 1$ and that y is one-hot so that $y_1 + y_2 = 1$, we can re-write the general formula as:

$$C = -\frac{1}{n} \sum_x y_1 \ln(a_1) + (1 - y_1) \ln(1 - a_1)$$

Which is the same equation we first started with. However, now we relax the constraint that $j = 2$. It can be

shown that the cross-entropy loss here has the same gradient as for the case of the binary/two-class cross-entropy on logistic outputs.

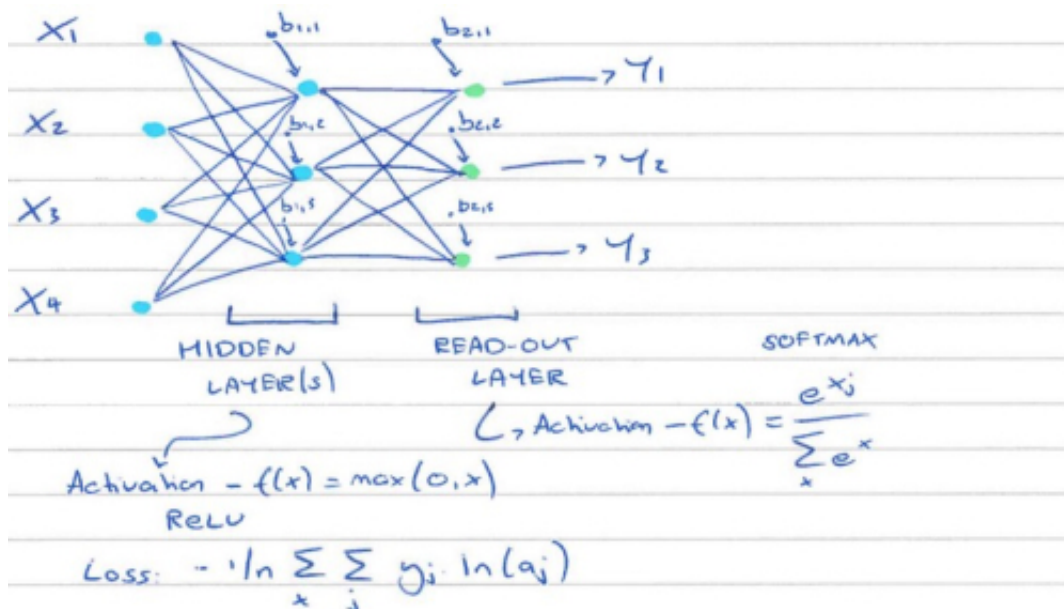
$$\frac{\partial C}{\partial \beta_i} = \frac{1}{n} \sum_x x_i (a(x) - y)$$

However, although the gradient has the same formula it will be different because the activation here takes on a different value (softmax instead of logistic-sigmoid).

In most deep-learning frameworks you have the choice of 'binary-crossentropy' or 'categorical-crossentropy' loss. Depending on whether your last layer contains sigmoid or softmax activation you would want to choose binary or categorical cross-entropy (respectively). The training of the network should not be affected, since the gradient is the same, however the reported loss (for evaluation) would be wrong if these are mixed up.

The motivation to go through softmax is that most neural-networks will use a softmax layer as the final/'read-out' layer, with a multinomial/categorical cross-entropy loss instead of using sigmoids with a binary cross-entropy loss — when the categories are mutually exclusive. Although multiple sigmoids for multiple classes can also be used (and will be used in the next example), this is generally only used for the case of non-mutually-exclusive labels (i.e. we can have multiple labels). With a softmax output, since the sum of the outputs is constrained to equal 1, we have the advantage of interpreting the outputs as class probabilities.

Step 4 - Neural Network (See Notebook)



A neural network can be thought of as a series of logistic regressions stacked on top of each other. This means we could say that a logistic regression is a neural-network (with sigmoid activations) with no hidden-layer.

This hidden-layer lets a neural-network generate non-linearities and leads to the Universal approximation theorem, which states that a network with just one hidden layer can approximate any linear or non-linear function. The number of hidden-layers can go into the hundreds.

It can be useful to think of a neural-network as a combination of two things: 1) many logistic regressions stacked on top of each other that are 'feature-generators' and 2) one read-out-layer which is just a softmax regression. The recent successes in deep-learning can arguable be attributed to the 'feature-generators'. For example; previously with computer vision, we had to painfully state that we wanted to find triangles, circles, colours, and in what combination (similar to how economists decide which interaction-terms they need in a linear regression). Now,

the hidden-layers are basically an optimisation to decide which features (which 'interaction-terms') to extract. A lot of deep-learning (transfer learning) is actually done by generating features using a trained-model with the head (read-out layer) cut-off, and then training a logistic regression (or boosted decision-trees) using those features as inputs.

The hidden-layer also means that our loss function is not convex in parameters and we can't roll down a smooth-hill to get to the bottom. Instead of using Gradient Descent (which we did for the case of a logistic-regression) we will use Stochastic Gradient Descent (SGD), which basically shuffles the observations (random/stochastic) and updates the gradient after each mini-batch (generally much less than total number of observations) has been propagated through the network. There are many alternatives to SGD that Sebastian Ruder does a great job of summarising [here](#). I think this is a fascinating topic to go through, but outside the scope of this blog-post. Briefly, however, the vast majority of the optimisation methods are first-order (including SGD, Adam, RMSprop, and Adagrad) because calculating the second-order is too computationally difficult. However, some of these first-order methods have a fixed learning-rate (SGD) and some have an adaptive learning-rate (Adam), which means that the 'amount' we update our weights by becomes a function of the loss - we may make big jumps in the beginning but then take smaller steps as we get closer to the target.

It should be clear, however that minimising the loss on training data is not the main goal - in theory we want to minimise the loss on 'unseen'/test data; hence all the optimisation methods proxy for that under the assumption that a low loss on training data will generalise to 'new' data from the same distribution. This means we may prefer a neural-network with a higher training-loss; because it has a lower validation-loss (on data it hasn't been trained on) - we would typically say that the network has 'overfit' in this case. There have been some [recent papers](#) that claim that adaptive optimisation methods do not generalise as well as SGD because they find very sharp minima points.

Previously we only had to back-propagate the gradient one layer, now we also have to back-propagate it through all the hidden-layers. Explaining the back-propagation algorithm is beyond the scope of this post, however it is crucial to understand. Many good [resources](#) exist online to help.

We can now create a neural-network from scratch in R using four functions.

First, we initialise our weights:

```
neuralnetwork <- function(sizes, training_data, epochs,  
  mini_batch_size, lr, C, verbose=FALSE,  
  validation_data=training_data)
```

Since we now have a complex combination of parameters we can't just initialise them to be 1 or 0, like before - the network may get stuck. To help, we use the gaussian distribution (however, just like with the optimisation, there are many other methods):

```
1 biases <- lapply(seq_along(listb), function(idx){  
2   r <- listb[[idx]]  
3   matrix(rnorm(n=r), nrow=r, ncol=1)  
4   })  
5  
6 weights <- lapply(seq_along(listb), function(idx){  
7   c <- listw[[idx]]  
8   r <- listb[[idx]]  
9   matrix(rnorm(n=r*c), nrow=r, ncol=c)
```



```
10      })
```

weights.R hosted with ❤ by GitHub

[view raw](#)

Second, we use stochastic gradient descent as our optimisation method:

```
1  SGD <- function(training_data, epochs, mini_batch_size, lr, C, sizes, num_layers, b
2      verbose=FALSE, validation_data)
3  {
4      # Every epoch
5      for (j in 1:epochs){
6          # Stochastic mini-batch (shuffle data)
7          training_data <- sample(training_data)
8          # Partition set into mini-batches
9          mini_batches <- split(training_data,
10                                ceiling(seq_along(training_data)/mini_batch_size))
11         # Feed forward (and back) all mini-batches
12         for (k in 1:length(mini_batches)) {
13             # Update biases and weights
14             res <- update_mini_batch(mini_batches[[k]], lr, C, sizes, num_layers, biases,
15             biases <- res[[1]]
16             weights <- res[[-1]]
17         }
18     }
19     # Return trained biases and weights
20     list(biases, weights)
21 }
```

sgd.R hosted with ❤ by GitHub

[view raw](#)


Third, as part of the SGD method, we update the weights after each mini-batch has been forward and backwards-propagated:

```
1  update_mini_batch <- function(mini_batch, lr, C, sizes, num_layers, biases, weights
2  {
3      nmb <- length(mini_batch)
4      listw <- sizes[1:length(sizes)-1]
5      listb <- sizes[-1]
6
7      # Initialise updates with zero vectors (for EACH mini-batch)
8      nabla_b <- lapply(seq_along(listb), function(idx){
9          r <- listb[[idx]]
10         matrix(0, nrow=r, ncol=1)
```

```

11     })
12     nabla_w <- lapply(seq_along(listb), function(idx){
13         c <- listw[[idx]]
14         r <- listb[[idx]]
15         matrix(0, nrow=r, ncol=c)
16     })
17
18     # Go through mini_batch
19     for (i in 1:nmb){
20         x <- mini_batch[[i]][[1]]
21         y <- mini_batch[[i]][[-1]]
22         # Back propogation will return delta
23         # Backprop for each observation in mini-batch
24         delta_nablas <- backprop(x, y, C, sizes, num_layers, biases, weights)
25         delta_nabla_b <- delta_nablas[[1]]
26         delta_nabla_w <- delta_nablas[[-1]]
27         # Add on deltas to nabla
28         nabla_b <- lapply(seq_along(biases), function(j)
29             unlist(nabla_b[[j]])+unlist(delta_nabla_b[[j]]))
30         nabla_w <- lapply(seq_along(weights), function(j)
31             unlist(nabla_w[[j]])+unlist(delta_nabla_w[[j]]))
32     }
33     # After mini-batch has finished update biases and weights:
34     # i.e. weights = weights - (learning-rate/numbr in batch)*nabla_weights
35     # Opposite direction of gradient
36     weights <- lapply(seq_along(weights), function(j)
37         unlist(weights[[j]])-(lr/nmb)*unlist(nabla_w[[j]]))
38     biases <- lapply(seq_along(biases), function(j)
39         unlist(biases[[j]])-(lr/nmb)*unlist(nabla_b[[j]]))
40     # Return
41     list(biases, weights)
42 }

```

update_mini_batch.R hosted with  by GitHub

[view raw](#)

Fourth, the algorithm we use to calculate the deltas is the back-propagation algorithm.

In this example we use the cross-entropy loss function, which produces the following gradient:

```

| cost_delta <- function(method, z, a, y) {
|   if (method=='ce'){return (a-y)}
| }

```

Also, to be consistent with our logistic regression example we use the sigmoid activation for the hidden layers and for the read-out layer:

```
# Calculate activation function
sigmoid <- function(z){1.0/(1.0+exp(-z))}
# Partial derivative of activation function
sigmoid_prime <- function(z){sigmoid(z)*(1-sigmoid(z))}
```

As mentioned previously, usually the softmax activation is used for the read-out layer. For the hidden layers, ReLU is more common, which is just the max function (negative weights get flattened to 0). The activation function for the hidden layers can be imagined as a race to carry a baton/flame (gradient) without it dying. The sigmoid function flattens out at 0 and at 1, resulting in a flat gradient which is equivalent to the flame dying out (we have lost our signal). The ReLU function helps preserve this gradient.

The back-propagation function is defined as:

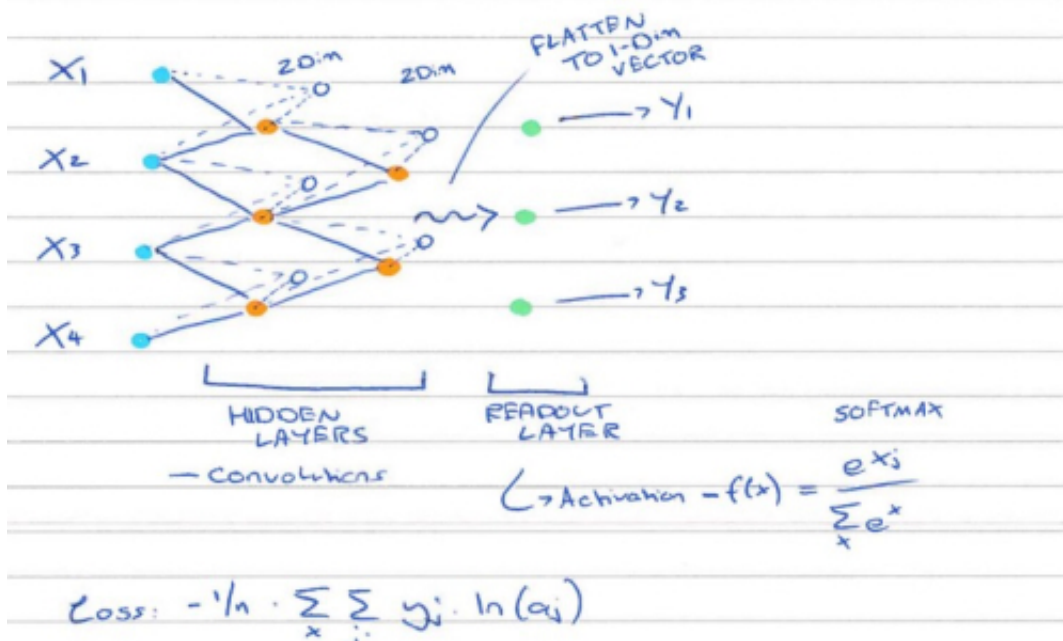
```
backprop <- function(x, y, C, sizes, num_layers, biases, weights)
```

Check out the [notebook](#) for the full code — however the principle remains the same: we have a forward-pass where we generate our prediction by propagating the weights through all the layers of the network. We then plug this into the cost gradient and update the weights through all of our layers.

This concludes the creation of a neural network (with as many hidden layers as you desire). It can be a good exercise to replace the hidden-layer activation with ReLU and read-out to be softmax, and also add L1 and L2 regularization. Running this on the [iris dataset](#) in the notebook (which contains 4 explanatory variables with 3 possible outcomes), with just one hidden-layer containing 40 neurons we get an accuracy of 96% after 30 rounds/epochs of training.

The notebook also runs a 100-neuron [handwriting-recognition](#) example to predict the digit corresponding to a 28x28 pixel image.

Step 5 - Convolutional Neural Network (See Notebook)



Here, we will briefly examine only the forward-propagation in a convolutional neural-network (CNN). CNNs were first made popular in 1998 by [LeCun's seminal paper](#). Since then, they have proven to be the best method we have for recognising patterns in images, sounds, videos, and even text!

Image recognition was initially a manual process; researchers would have to specify which bits (features) of an image were useful to identify. For example, if we wanted to classify an image into 'cat' or 'basketball' we could have created code that extracts colours (basketballs are orange) and shapes (cats have triangular ears). Perhaps with a count of these features we could then run a linear regression to get the relationship between number of triangles and whether the image is a cat or a tree. This approach suffers from issues of image scale, angle, quality and light. [Scale Invariant Feature Transformation](#) (SIFT) largely improved upon this and was used to provide a 'feature description' of an object, which could then be fed into a linear regression (or any other relationship learner). However, this approach had set-in-stone rules that could not be optimally altered for a specific domain.

CNNs look at images (extract features) in an interesting way. To start, they look only at very small parts of an image (at a time), perhaps through a restricted window of 5 by 5 pixels (a filter). 2D convolutions are used for images, and these slide the window across until the whole image has been covered. This stage would typically extract colours and edges. However, the next layer of the network would look at a combination of the previous filters and thus 'zoom-out'. After a certain number of layers the network would be 'zoomed-out' enough to recognise shapes and larger structures.

These filters end up as the 'features' that the network has learned to identify. It can then pretty much count the presence of each feature to identify a relationship with the image label ('basketball' or 'cat'). This approach appears quite natural for images — since they can be broken down into small parts that describe it (colours, textures, etc.). CNNs appear to thrive on the fractal-like nature of images. This also means they may not be a great fit for other forms of data such as an excel worksheet where there is no inherent structure: we can change the column order and the data remains the same — try swapping pixels in an image (the image changes)!

In the previous example we looked at a standard neural-net classifying handwritten text. In that network each neuron from layer i , was connected to each neuron at layer j — our 'window' was the whole image. This means if we learn what the digit '2' looks like; we may not recognise it when it is written upside down by mistake, because we have only seen it upright. CNNs have the advantage of looking at small bits of the digit '2' and finding patterns between patterns. This means that a lot of the features it extracts may be immune to rotation, skew, etc. For more detail, Brandon Rohrer explains [here](#) what a CNN actually is in detail.

We can define a 2D convolution function in R:

```
convolution <- function(input_img, filter, show=TRUE, out=FALSE)
{
  conv_out <- outer(
    1:(nrow(input_img)-kernel_size[[1]]+1),
    1:(ncol(input_img)-kernel_size[[2]]+1),
    Vectorize(function(r,c) sum(input_img[r:(r+kernel_size[[1]]-1),
                                c:(c+kernel_size[[2]]-1)]*filter))
  )
}
```

And use it to apply a 3x3 filter to an image:

```
conv_emboss <- matrix(c(2,0,0,0,-1,0,0,0,-1), nrow = 3)
convolution(input_img = r_img, filter = conv_emboss)
```

You can check the notebook to see the result, however this seems to extract the edges from a picture. Other, convolutions can 'sharpen' an image, like this 3x3 filter:

```
conv_sharpen <- matrix(c(0,-1,0,-1,5,-1,0,-1,0), nrow = 3)
convolution(input_img = r_img, filter = conv_sharpen)
```

Typically we would randomly initialise a number of filters (e.g. 64):

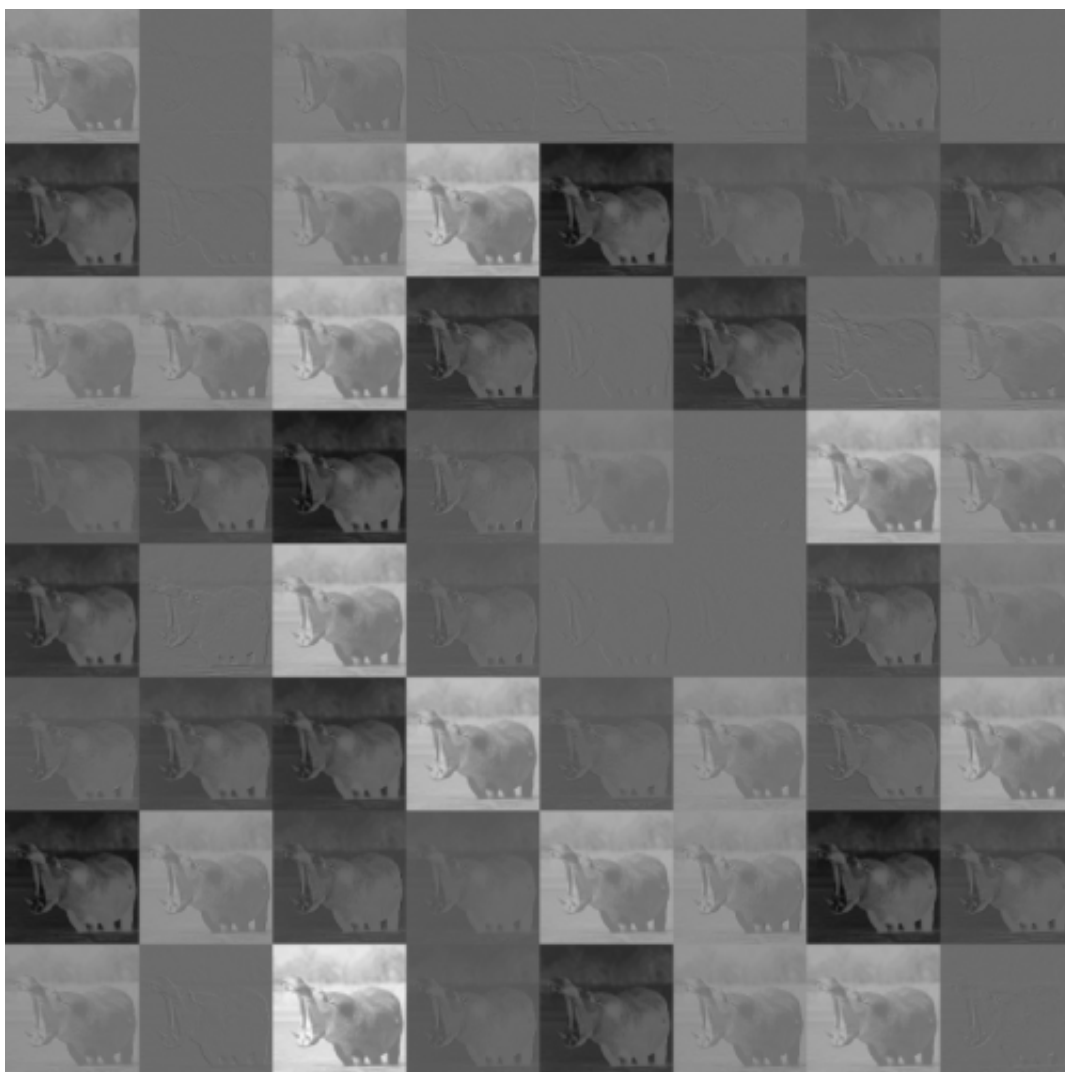
```
filter_map <- lapply(X=c(1:64), FUN=function(x){  
  # Random matrix of 0, 1, -1  
  conv_rand <- matrix(sample.int(3, size=9, replace = TRUE), ncol=3)-2  
  convolution(input_img = r_img, filter = conv_rand, show=FALSE, out=TRUE)  
})
```

We can visualise this map with the following function:

```
1 square_stack_lst_of_matrices <- function(lst)  
2 {  
3   sqr_size <- sqrt(length(lst))  
4   # Stack vertically  
5   cols <- do.call(cbind, lst)  
6   # Split to another dim  
7   dim(cols) <- c(dim(filter_map[[1]])[[1]],  
8                 dim(filter_map[[1]])[[1]]*sqr_size,  
9                 sqr_size)  
10  # Stack horizontally  
11  do.call(rbind, lapply(1:dim(cols)[3], function(i) cols[, , i]))  
12 }
```

featuremap.R hosted with ❤ by GitHub

[view raw](#)



Running this function we notice how computationally intensive the process is (compared to a standard fully-connected layer). If these feature maps are not useful 'features' (i.e. the loss is difficult to decrease when these are used) then back-propagation will mean we will get different weights which correspond to different feature-maps; which will become more useful to make the classification.

Typically we stack convolutions on top of other convolutions (and hence the need for a deep network) so that edges become shapes and shapes become noses and noses become faces. It can be interesting to examine some feature maps from trained networks to see what the network has actually learnt.

Download Notebooks

You can find notebooks implementing the code behind this post on Github by following the links in the section headings, or as Azure Notebooks at the link below:

Azure Notebooks: [NeuralNetR](#)

Posted by [Guest Blogger](#) at 09:30 | [Permalink](#)

Comments

 You can follow this conversation by subscribing to the [comment feed](#) for this post.

Wow... I understood a word or two. :)

Posted by: [Mike-EEE](#) | [July 18, 2017 at 10:59](#)

Great walk thru. Good bridge material between basic and advanced presentations.

Posted by: [Patrick Stroh](#) | [July 19, 2017 at 05:03](#)

The comments to this entry are closed.