Implementing a perceptron and backpropagation

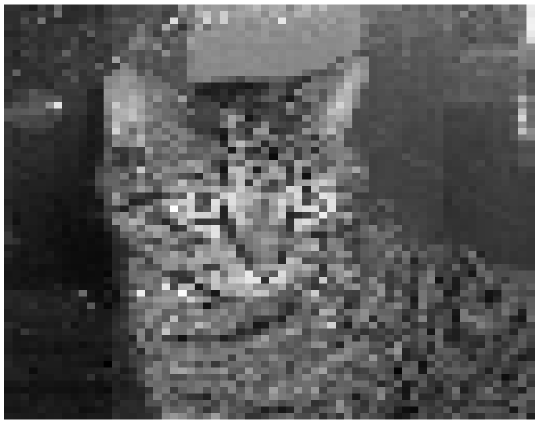
In this notebook we are going to walk-through the nuts and bolts of training a simple perceptron with backprop. The first thing we need to do is to load the data and some helper functions:

```
source("https://raw.githubusercontent.com/beamandrew/BMI705/master/helper.R")
load(url("https://www.dropbox.com/s/unmfitketiba9i7/cat_dog_data_greyscale.RData?dl=1"))
options(repr.plot.width=4, repr.plot.height=4)
```

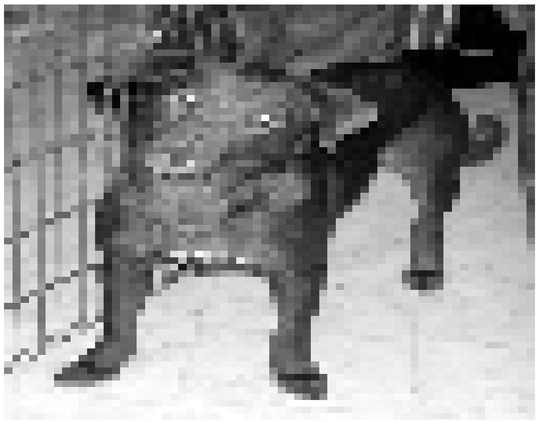
Data Structure

The data we will be working with are images of cats and dogs, obtained from this kaggle contest that have been resized to 64x64 images. For the moment, we will be working with black and white images to keep things simple. The `x_train` matrix is a matrix with dimensions of `(2000,64,64)` and `y_train` is a binary vector where `1` means the corresponding entry in `x_train` is a cat and 0 means it is a dog. Let's look at a few images:

```
display_image(x_train[2,,])
```



```
display_image(x_train[1002,,])
```



Our perceptron will expect each example to be a vector, so we will need to flatten the imsages so that each row represents one image. This means that we need to reshape our input to be a 2000 x 4096 (64*64) matrix.

```
## Flatten all of our images into row-vectors ##
x_train <- matrix(x_train,nrow=nrow(x_train))
```

```
x_test <- matrix(x_test,nrow=nrow(x_test))
print(paste0("Number of observations: " , nrow(x_train)))
```

```
## [1] "Number of observations: 2000"
```

```
print(paste0("Number of features: " , dim(x_train)[2]))
```

```
## [1] "Number of features: 4096"
```

Definining the forward pass

This first thing we need to define is the *forward pass*. During the forward pass, the network computes the output probabilites for each input example.

Let's setup a vector `w` that will serve as our network's weights. We will start by giving all of the 4096 weights small random values:

```
## Let's set up our weight vector with random values initially ##
w <- t(rnorm(ncol(x_train), mean = 0, sd=0.01))
```

Next, given our input example x_i and a weight vector w, we first compute the weighted sum for example `i` like so:

$$h_i = \sum_{j=1}^{4096} x_{ij} * w_j$$

Next, we transfrom h_i into a probability using the *sigmoid* transformation:

$$\frac{1}{1 + exp(-h_i)}$$

Now we have all the components for a full forward pass in our perceptron. Below is a template of the forward pass, try to fill out the missing parts. In `utils.R` (which should already be loaded), there is a `sigmoid(h)` function that is vectorized and will take the entire h vector and apply the transform each element. Feel free to use it if you wish.

```
# This should return a vector representing the
# probability that y = 1 for each of the 2000 training samples
forward_pass <- function(X,w) {
    ## Compute the matrix-vector multiplication between the inputs and weights ##
    h <- 0 # Fill this in
    ## Transform into probabilities using the provided sigmoid() function ##
    probs <- 0 # Fill this in
    return(probs)
}
## See if it works ##
print("Dog probabilities for the first five samples:")
p <- forward_pass(x_train,w)
print(dim(p))
```

Definining the backward pass

The next thing we need to do is to implement the *backward pass*. The backward pass computes the derivative of all our network's parameters with respect to the loss function. A typical loss function for binary classification is *log loss* and for a single example is:

$$\ell(y_i, p_i) = -y_i * log(p_i) - (1 - y_i) * log(1 - p_i)$$

The derivative of this function with respect to w_j is:

$$\frac{\partial w_j}{\partial \ell} = -(y_i - p_i) * x_{ij}$$

The full gradient is obtained by simply taking the average across all examples:

$$\frac{\partial w_j}{\partial \ell} = \frac{1}{2000} \sum_{i=1}^{2000} -(y_i - p_i) * x_{ij}$$

Note we take the average instead of summing so that later our learning rate will be independent of the number of samples.

```r
## This function should return a vector the same size as the weight vector w.
## Each element of the vector is the partial dervative
## evaluated at the current value of the loss function
backward_pass <- function(y, X, p) {
  grads <- 0 ## Fill this in
  return( grads )
}

## See if it works
grads <- backward_pass(y_train,x_train,p)
print(dim(grads))
```

In class assignment

Now it's time to run the gradient descent procedure. The update rule for each w_j at each iteration is given by:

$$w_j = w_j - \eta * \frac{\partial w_j}{\partial \ell}$$

where η is the *learning rate* and is typically some small number < 1.

Fill in the template below to train your perceptron. Try to see if you can get an accuracy ~ 65%. Note there will be some randomness in your loss and accruacy from run to run, even with the same parameters.

```r
## Choose how many iterations to run gradient descent ##
epochs = 1

# Choose a number > 0 and < 1 as your learning rate
learning_rate = 0

## Initialize w ##
w <- t(rnorm(ncol(x_train), mean = 0, sd=0.01))

for(i in 1:epochs) {
  probs <- forward_pass(x_train,w)
  grads <- backward_pass(y_train, x_train, probs)
  ## Update the weights using the gradient descent rule ##
  w <- 0 # Fill the gradient descent

  ## Print out some information every 100 epochs ##
  if(i %% 100 == 0) {
    message(paste0("\nReport at epoch: ", i))
    message(paste0("Loss: ", log_loss(y_train,probs)))
    message(paste0("Accuracy: ", binary_accuracy(y_train,probs)))
    message(paste0("Learning rate: ", learning_rate))
  }
}
```

Solution

Here are the matrix-versions of the forward and backward passes

```
# This should return a vector representing the
# probability that y = 1 for each of the 2000 training samples
forward_pass <- function(X,w) {
    ## Compute the matrix-vector multiplication between the inputs and weights ##
    h <- X %*% t(w)
    ## Transform into probabilities using the provided sigmoid() function ##
    probs <- sigmoid(h)
    return(probs)
}

backward_pass <- function(y, X, p) {
  grads <- -t(y-p)%*%X
  return( grads/nrow(X) )
}
```

Here is the full gradient descent procedure

```
## Choose how many iterations to run gradient descent ##
epochs = 1000

# Choose a number > 0 and < 1 as your learning rate
learning_rate = 0.01

## Initialize w ##
w <- t(rnorm(ncol(x_train), mean = 0, sd=0.01))

for(i in 1:epochs) {
  probs <- forward_pass(x_train,w)
  grads <- backward_pass(y_train, x_train, probs)
  ## Update the weights using the gradient descent rule ##
  learning_rate <- learning_rate*0.999
  w <- w - learning_rate*grads

  ## Print out some information every 100 epochs ##
  if(i %% 100 == 0) {
    message(paste0("\nReport at epoch: ", i))
    message(paste0("Loss: ", log_loss(y_train,probs)))
    message(paste0("Accuracy: ", binary_accuracy(y_train,probs)))
    message(paste0("Learning rate: ", learning_rate))
  }
}
```

```
##
## Report at epoch: 100

## Loss: 0.710740076482673

## Accuracy: 0.54

## Learning rate: 0.00904792147113708

##
## Report at epoch: 200

## Loss: 0.659479770461822

## Accuracy: 0.6195

## Learning rate: 0.00818648829478636

##
## Report at epoch: 300

## Loss: 0.652878830032204

## Accuracy: 0.634

## Learning rate: 0.00740707032156099

##
## Report at epoch: 400

## Loss: 0.647585541332747

## Accuracy: 0.64

## Learning rate: 0.0067018590600674

##
## Report at epoch: 500

## Loss: 0.643167406299669

## Accuracy: 0.651

## Learning rate: 0.00606378944861184

##
## Report at epoch: 600

## Loss: 0.639402735244797

## Accuracy: 0.6575

## Learning rate: 0.00548646907485496

##
## Report at epoch: 700

## Loss: 0.636154075217076

## Accuracy: 0.663

## Learning rate: 0.00496411413431098

##
## Report at epoch: 800
## Loss: 0.633326333163344

## Accuracy: 0.6695

## Learning rate: 0.00449149148610075

##
## Report at epoch: 900

## Loss: 0.630849247921382

## Accuracy: 0.673

## Learning rate: 0.00406386622545204

##
## Report at epoch: 1000

## Loss: 0.628668653509669

## Accuracy: 0.675

## Learning rate: 0.00367695424770963
```