Applications

Of Data Science

Demystifying Deep Neural Networks

Applications of Data Science - Class 16

Giora Simchoni

gsimchoni@gmail.com and add #dsapps in subject

Stat. and OR Department, TAU

2020-03-01

Logistic Regression as *we* know it

LR as GLM

- We observe $y_1, \ldots, y_n$ binary outcomes, therefore we say $Y_i \sim Bernoulli(p_i)$ and $P(Y_i) = p_i^{y_i}(1-p_i)^{1-y_i}$
- We have $X_{n\text{x}(q+1)}$ matrix of q predictors for each observation + a $\vec{1}$ column for the intercept, let each row be x_i
- We wish to estimate a vector of weights for each of the $q+1$ predictors $\beta_{(q+1)\text{x}1}$, such that some function of $x_i\beta$ explains $E(Y_i) = P(Y_i = 1) = p_i$
- We choose some *link function* g and model *this* transformation of $E(Y_i)$
- Typically for this case g is the logit function: $logit(p_i) = log(\frac{p_i}{1-p_i}) = x_i\beta$

- And so we can write:

$$E(Y_i) = P(Y_i = 1|x_i; \beta) = p_i = g^{-1}(x_i\beta) = \frac{1}{1+e^{-x_i\beta}}$$

- Also note that now we can write:

$$P(Y_i|X; \beta) = [g^{-1}(x_i\beta)]^{y_i}[1 - g^{-1}(x_i\beta)]^{1-y_i} = (\frac{1}{1+e^{-x_i\beta}})^{y_i}(\frac{e^{-x_i\beta}}{1+e^{-x_i\beta}})^{1-y_i}$$

- Once we get our estimate $\hat{\beta}$:

1. We could "explain" Y_i, the size and direction of each component of $\hat{\beta}$ indicating the contribution of that predictor to the *log-odds* of Y_i being 1
2. We could "predict" probability of new observation x_i having $Y_i = 1$ by fitting a probability $\hat{p}_i = \frac{1}{1+e^{-x_i\hat{\beta}}}$, where typically if $\hat{p}_i > 0.5$, or $x_i\hat{\beta} > 0$, we predict $Y_i = 1$

How to fit the model? MLE

Under the standard Maximum Likelihood approach we assume Y_i are also *independent* and so their joint "likelihood" is:

$$L(\beta|X,y) = \prod_{i=1}^{n} P(Y_i|X;\beta) = \prod_{i=1}^{n} [g^{-1}(x_i\beta)]^{y_i}[1-g^{-1}(x_i\beta)]^{1-y_i}$$

The $\hat{\beta}$ we choose is the vector maximizing $L(\beta|X,y)$, only we take the log-likelihood which is easier to differentiate:

$$l(\beta|X,y) = \sum_{i=1}^{n} \ln P(Y_i|X;\beta) =$$
$$\sum_{i=1}^{n} y_i \ln[g^{-1}(x_i\beta)] + (1-y_i)\ln[1-g^{-1}(x_i\beta)] =$$

This looks Ok but let us improve a bit just for easier differentiation:

$$\sum_{i=1}^{n} \ln[1-g^{-1}(x_i\beta)] + y_i \ln[\frac{g^{-1}(x_i\beta)}{1-g^{-1}(x_i\beta)}] =$$
$$\sum_{i=1}^{n} -\ln[1+e^{x_i\beta}] + y_i x_i \beta$$

Life is like a box of chocolates

Differentiate:

$$\frac{\partial l(\beta|X,y)}{\partial \beta_j} = \sum_{i=1}^{n} -\frac{1}{1+e^{x_i\beta}} e^{x_i\beta} x_{ij} + y_i x_{ij} = \sum_{i=1}^{n} x_{ij}(y_i - g^{-1}(x_i\beta))$$

Or in matrix notation:

$$\frac{\partial l(\beta|X,y)}{\partial \beta} = X^T(y - g^{-1}(X\beta))$$

We would like to equate this with $\vec{0}$ and get $\hat{\beta}$ but there's no closed solution...

At which point usually the Newton-Raphson method comes to the rescue.

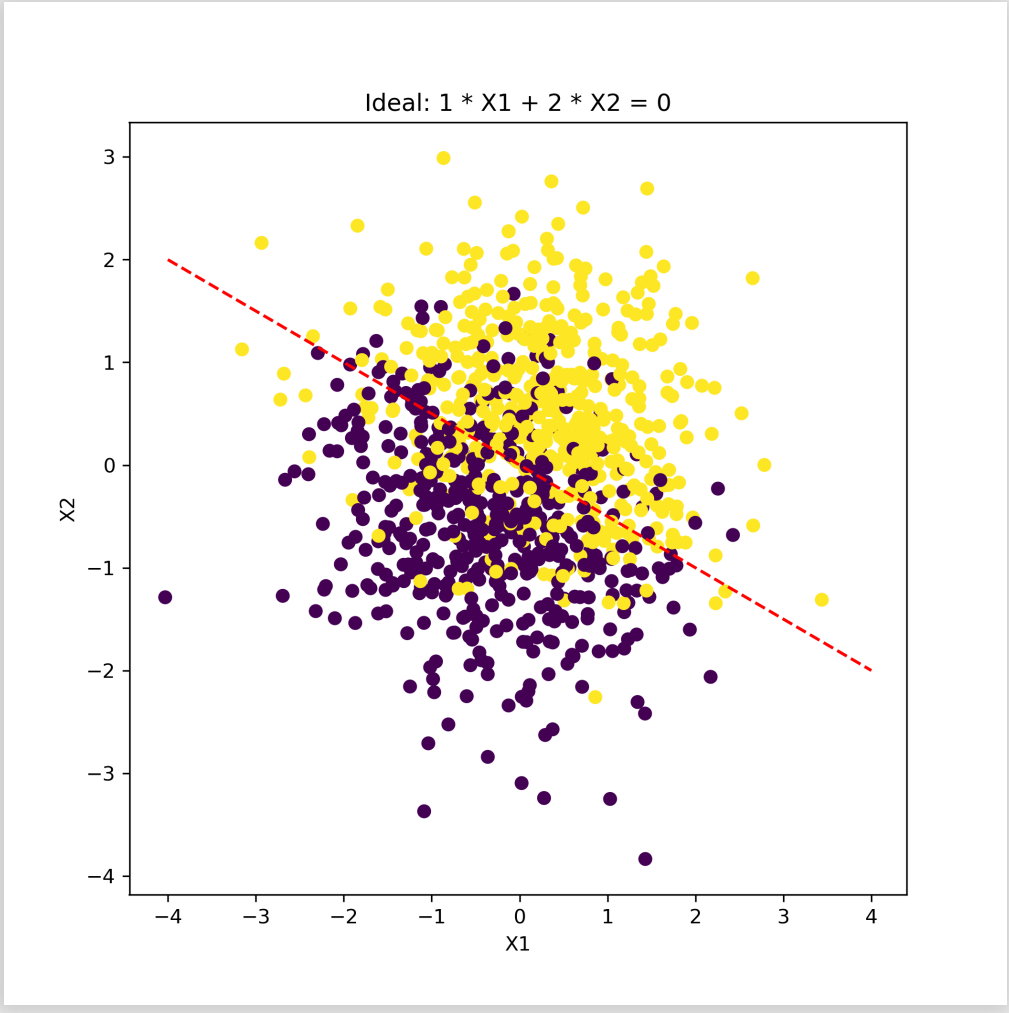
But let's look at simple gradient descent:

Gradient De(A)scent

- Instead of maximizing log-likelihood, let's minimize minus log-likelihood $-l(\beta)$
- We'll start with an initial guess $\hat{\beta}_{t=0}$
- The partial derivatives vector of $-l(\beta)$ at point $\hat{\beta}_t$ (a.k.a the *gradient* $-\nabla l(\hat{\beta}_t)$) points to the direction of where $-l(\beta)$ has its steepest descent
- We'll go a small $alpha$ step down that direction: $\hat{\beta}_{t+1} = \hat{\beta}_t - \alpha \cdot [-\nabla l(\hat{\beta}_t)]$
- We do this for I iterations or until some stopping rule indicating $\hat{\beta}$ has converged

Show me that it's working

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

n = 1000
q = 2
X = np.random.normal(size = n * q).reshape((n, q))
# X_with_intercept = np.hstack((np.ones(n).reshape((n, 1)), X))
beta = np.arange(1, q + 1) # [1, 2]
p = 1 / (1 + np.exp(-np.dot(X, beta)))
y = np.random.binomial(1, p, size = n)


X1 = np.linspace(-4, 4) # for plotting

def plot_sim(plot_beta_hat=True):
  plt.clf()
  plt.scatter(X[:, 0], X[:, 1], c = y)
  plt.plot(X1, -X1 * beta[0]/beta[1], linestyle = '--', color = 'r
  if plot_beta_hat:
    plt.plot(X1, -X1 * beta_hat[0]/beta_hat[1], linestyle = '--')
  plt.xlabel('X1')
  plt.ylabel('X2')
  if plot_beta_hat:
    title = 'Guess: %.2f * X1 + %.2f * X2 = 0' % (beta_hat[0], bet
  else:
```


Ideal: 1 * X1 + 2 * X2 = 0
X2
X1
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

`sklearn` should solve this easily:

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='none', fit_intercept=False, max_i
lr.fit(X, y)
```

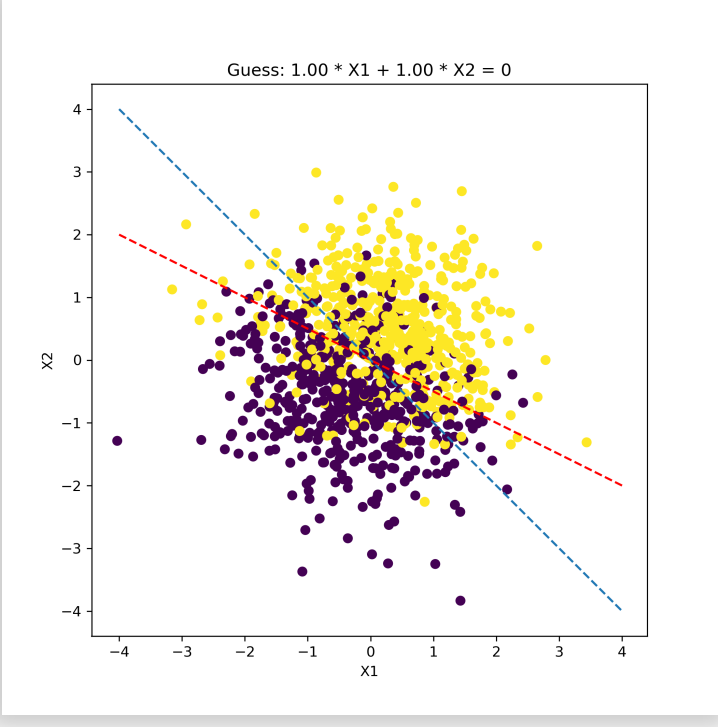
```
## LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Fa
##                    intercept_scaling=1, l1_ratio=None, max_iter=100,
##                    multi_class='auto', n_jobs=None, penalty='none',
##                    random_state=None, solver='lbfgs', tol=0.0001, verbos
##                    warm_start=False)
```

```
lr.coef_
```

```
## array([[1.05142777, 1.88704076]])
```

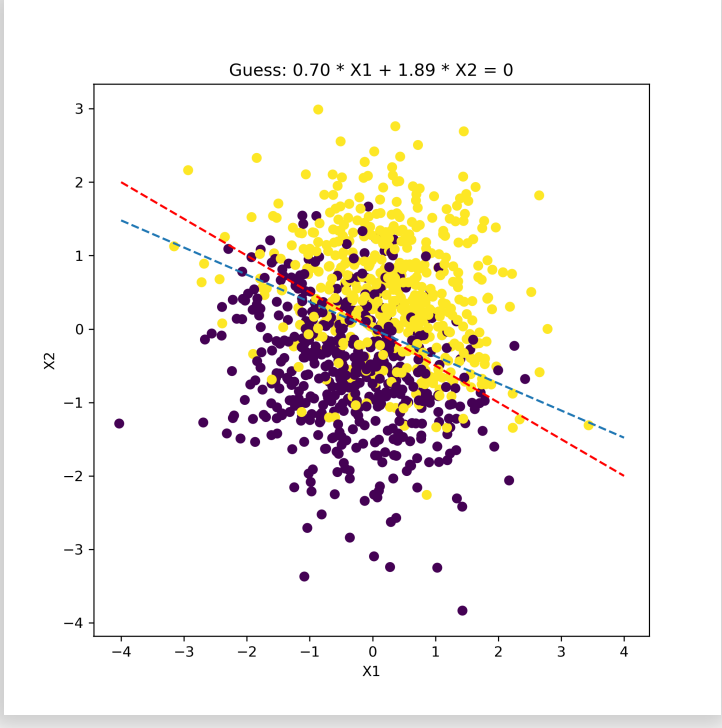
With Gradient Descent let's start with initial guess

```
beta_hat = np.ones(q) # [1, 1]

plot_sim()
```



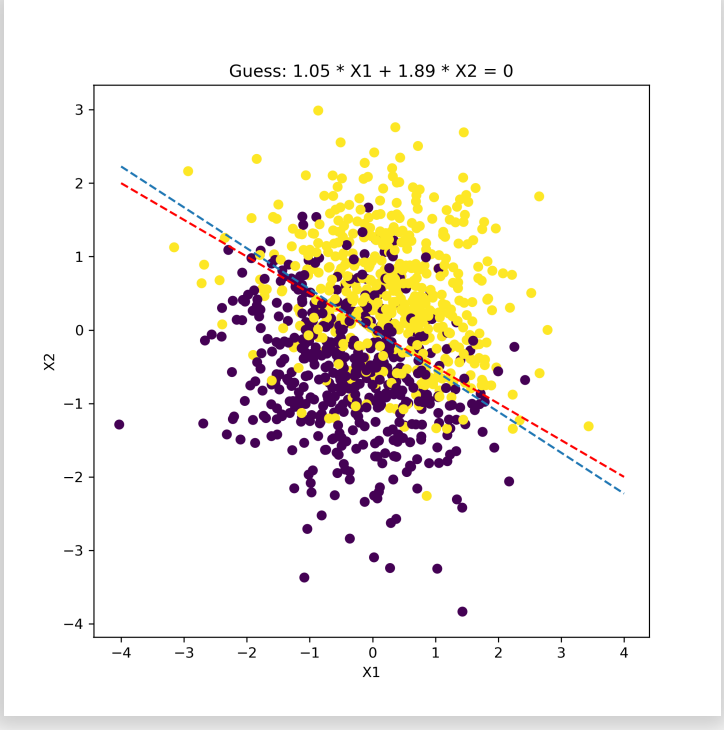
Let's do 1 iteration:

```
alpha = 0.01

p_hat = 1 / (1 + np.exp(-np.dot(X, beta_hat)))
grad = -np.dot(X.T, (y - p_hat))
beta_hat = beta_hat - alpha * grad

plot_sim()
```



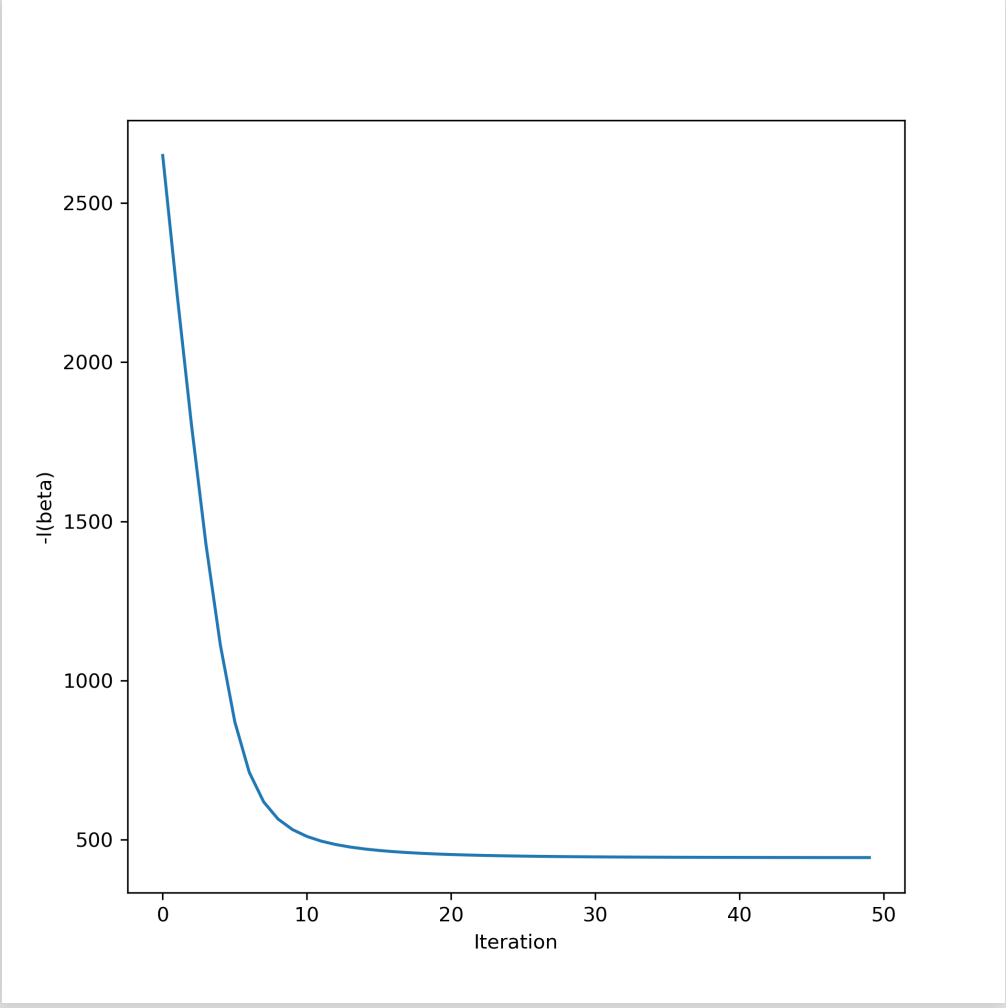
Let's do 50 more:

```
for i in range(10):
  p_hat = 1 / (1 + np.exp(-np.dot(X, beta_hat)))
  grad = -np.dot(X.T, (y - p_hat))
  beta_hat = beta_hat - alpha * grad

plot_sim()
```



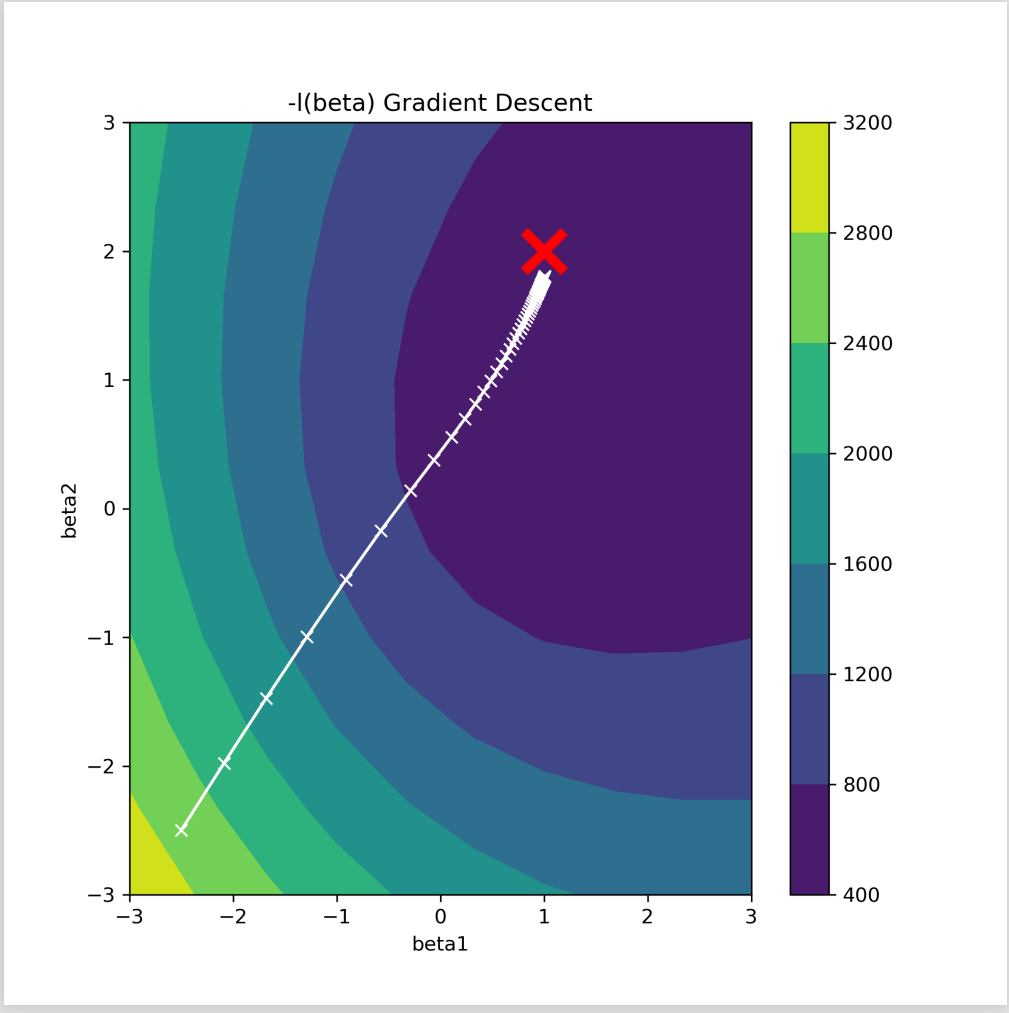
We didn't need to compute $-l(\beta)$ but let's:

```
alpha = 0.001
beta_hat = np.array([-2.5, -2.5])
betas = [beta_hat]
ls = []
for i in range(50):
  p_hat = 1 / (1 + np.exp(-np.dot(X, beta_hat)))
  l_minus = -np.sum(y * np.log(p_hat) + (1 - y) * np.log(1 - p_hat
  ls.append(l_minus)
  grad = -np.dot(X.T, (y - p_hat))
  beta_hat = beta_hat - alpha * grad
  betas.append(beta_hat)

plt.plot(range(50), ls)
plt.xlabel("Iteration")
plt.ylabel("-l(beta)")
plt.show()
```


2500
2000
1500
1000
500
-l(beta)
0
10
20
30
40
50
Iteration

Even fancier, visualize the actual Gradient Descent in the β space:

```
betas_arr = np.array(betas)
m = 10
beta1 = np.linspace(-3.0, 3.0, m)
beta2 = np.linspace(-3.0, 3.0, m)
B1, B2 = np.meshgrid(beta1, beta2)
L = np.zeros((m, m))
for i in range(m):
  for j in range(m):
    beta_hat = np.array([beta1[i], beta2[j]])
    p_hat = 1 / (1 + np.exp(-np.dot(X, beta_hat)))
    L[i, j] = -np.sum(y * np.log(p_hat) + (1 - y) * np.log(1 - p_h
fig, ax = plt.subplots(1,1)
cp = ax.contourf(B1, B2, L)
cb = fig.colorbar(cp)
ax.set_title('-l(beta) Gradient Descent')
ax.set_xlabel('beta1')
ax.set_ylabel('beta2')
ax.plot(betas_arr[:, 0], betas_arr[:, 1], marker='x', color ='whit
ax.plot([beta[0]], [beta[1]], marker='x', color='red', markersize=
plt.show()
```


-l(beta) Gradient Descent
beta1
beta2
3200
2800
2400
2000
1600
1200
800
400

Logistic Regression as Neural Network

Call me by your name

1. Call our $-l(\beta)$ "Cross Entropy"
2. Call $g^{-1}(X\beta)$ the "Sigmoid Function"
3. Call computing $\hat{p}_i$ and $-l(\hat{\beta})$ a "Forward Propagation" or "Feed Forward" step
4. Call the differentiation of $-l(\hat{\beta})$ a "Backward Propagation" step
5. Call our β vector $W_{(q+1)\text{x}1}$, a weight matrix
6. Add *Stochastic* Gradient Descent
7. Draw a diagram with circles and arrows, call these "Neurons", say something about the brain

And you have a Neural Network*.

*Ok, We'll add some stuff later

Cross Entropy

For discrete probability distributions $P(X)$ and $Q(X)$ with the same support $x \in \mathcal{X}$ Cross Entropy could be seen as a metric of the "distance" between distributions:

$$H(P, Q) = -E_P[\log(Q)] = -\sum_{x \in \mathcal{X}} P(X = x) \log[Q(X = x)]$$

In case X has two categories, and $p_1 = P(X = x_1), p_2 = 1 - p_1$ and same for q_1, q_2:

$$H(P, Q) = -[p_1 \log(q_1) + (1 - p_1) \log(1 - q_1)]$$

If we let $p_1 = y_i$ and $q_1 = \hat{p}_i = g^{-1}(x_i\hat{\beta})$ we get:

$$H(y_i, \hat{p}_i) = -[y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)] = \\ -[y_i \ln[g^{-1}(x_i\hat{\beta})] + (1 - y_i) \ln[1 - g^{-1}(x_i\hat{\beta})]]$$

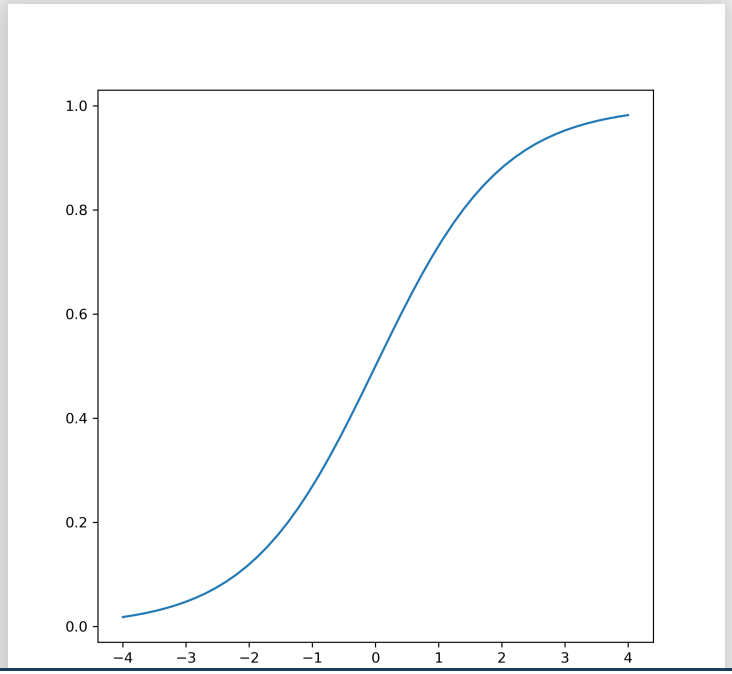
Which is exactly the contribution of the $i\text{th}$ observation to $-l(\hat{\beta})$.

Sigmoid Function

If $g(p)$ is the logit function, its inverse would be the sigmoid function:

$$g(p) = logit(p) = \log(\frac{p}{1-p}); \;\; g^{-1}(z) = sigmoid(z) = \frac{1}{1+e^{-z}}$$

So: $g^{-1}(g(p)) = sigmoid(logit(p)) = p$



Forward/Backward Propagation

Recall that each iteration of Gradient Descent included:

1. Forward step: Calculating the loss $-l(\hat{\beta})$
2. Backward step: Calculate the gradient $-\nabla l(\hat{\beta}_t)$
3. Gradient Descent: $\hat{\beta}_{t+1} = \hat{\beta}_t - \alpha \cdot [-\nabla l(\hat{\beta}_t)]$

```
# forward step
p_hat = 1 / (1 + np.exp(-np.dot(X, beta_hat)))
l_minus = -np.sum(y * np.log(p_hat) + (1 - y) * np.log(1 - p_hat))
# backward step
grad = -np.dot(X.T, (y - p_hat))
# descent
beta_hat = beta_hat - alpha * grad
```

Why "Forward", why "Backward"?...

Reminder: Chain Rule

In our case differentiating $l(\beta)$ analytically was... manageable.

As the NN architecture becomes more complex there is need to generalize this, and break down the derivative into (backward) steps.

Recall that according to the Chain Rule, if $y = y(x) = f(g(h(x)))$ then:

$$y'(x) = f'(g(h(x)) \cdot g'(h(x)) \cdot h'(x)$$

Or if you prefer, if $z = z(x);\ u = u(z);\ y = y(u)$ then:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dz} \cdot \frac{dz}{dx}$$

Let's re-write $-l(\beta)$ as a composite function:

- Multiplying β by x_i will be $z_i = z(\beta) = x_i\beta$
- Applying the sigmoid g^{-1} will be $p_i = g^{-1}(z_i) = \frac{1}{1+e^{-z_i}}$
- Calculating the (minus) Cross Entropy will be:
  $l_i = l(p_i) = y_i \ln(p_i) + (1 - p_i) \ln(1 - p_i)$
- So one element of $-l(\beta)$ will be: $l_i(p_i(z_i(\beta)))$

Hence, Forward.

Now $-l(\beta)$ is the sum of (minus) cross entropies:
$-l(\beta) = -\sum_i l_i(p_i(z_i(\beta)))$

And we could differentiate using the chain rule like so:

$$-\frac{\partial l(\beta)}{\partial \beta_j} = -\sum_i \frac{\partial l_i}{\partial p_i} \cdot \frac{\partial p_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial \beta_j}$$

Hence, Backward.

Each of these is simpler to calculate:

$$\frac{\partial l_i}{\partial p_i} = \frac{y_i - p_i}{p_i(1-p_i)}$$

$$\frac{\partial p_i}{\partial z_i} = p_i(1-p_i)$$

$$\frac{\partial z_i}{\partial \beta_j} = x_{ij}$$

And so:

$$-\frac{\partial l(\beta)}{\partial \beta_j} = -\sum_i \frac{y_i - p_i}{p_i(1-p_i)} \cdot p_i(1-p_i) \cdot x_{ij}$$

Which is excatly what we got analytically but now we can write our Gradient Descent iteration as a list of forward/backward steps:

```
def forward(X, y, beta_hat):
  z = np.dot(X, beta_hat)
  p_hat = 1 / (1 + np.exp(-z))
  l = y * np.log(p_hat) + (1 - y) * np.log(1 - p_hat)
  l_minus = -np.sum(l)
  return p_hat, l_minus

def backward(X, y, p_hat):
  dldz = y - p_hat
  dzdb = X.T
  grad = -np.dot(dzdb, dldz)
  return grad

def gradient_descent(alpha, beta_hat, grad):
  return beta_hat - alpha * grad

def optimize(X, y, alpha, beta_hat):
  p_hat, l = forward(X, y, beta_hat)
  grad = backward(X, y, p_hat)
  beta_hat = gradient_descent(alpha, beta_hat, grad)
  return l, beta_hat

def lr_nn(X, y, epochs):
  beta_hat = np.array([-2.5, -2.5])
  alpha = 0.001
  for i in range(epochs):
    l, beta_hat = optimize(X, y, alpha, beta_hat)
  return l, beta_hat
```

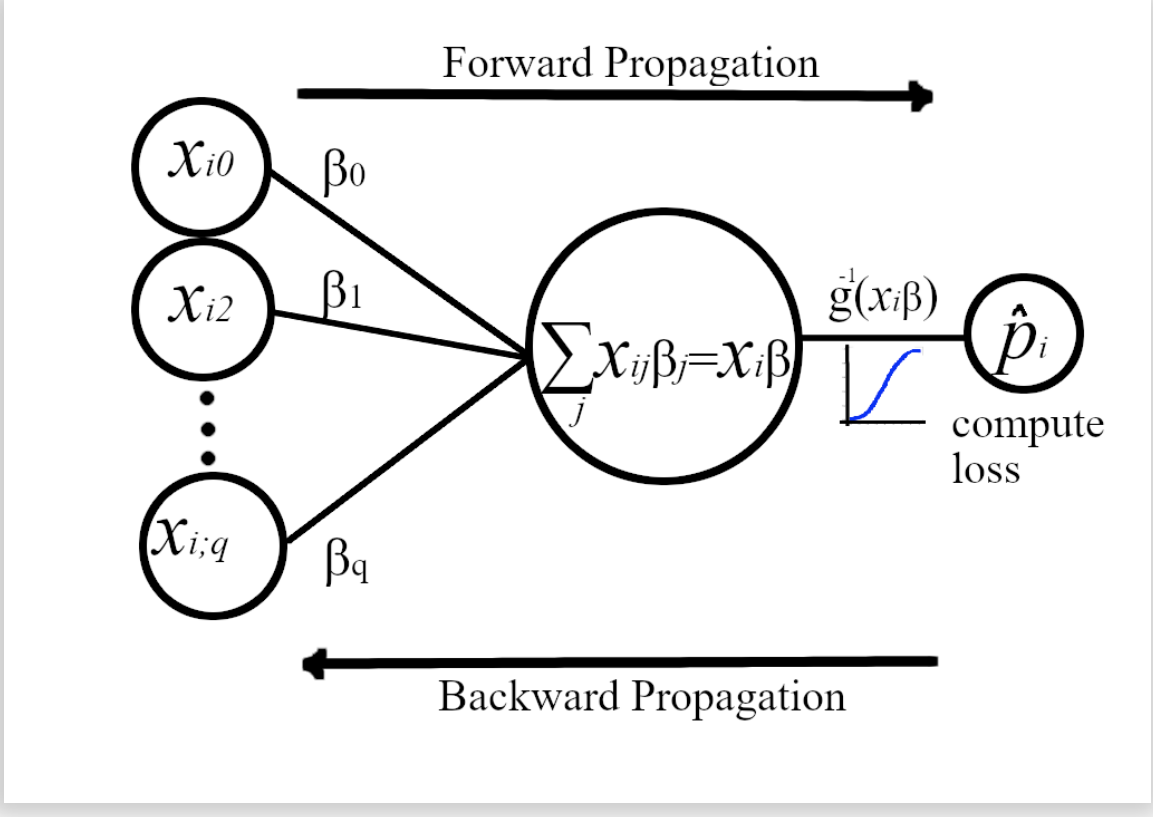
Stochastic Gradient Descent

```
def lr_nn(X, y, epochs):
  beta_hat = np.random.rand(X.shape[1])
  alpha = 0.001
  batch_size = 100
  n = X.shape[0]
  steps = int(n / batch_size)
  for i in range(epochs):
    print('epoch %d:' % i)
    permute = np.random.permutation(n)
    X_perm = X[permute, :]
    y_perm = y[permute]
    for j in range(steps):
      start = j * batch_size
      l, beta_hat = optimize(X_perm[start:start + batch_size, :],
                             y_perm[start:start + batch_size],
                             alpha, beta_hat)
      print('Trained on %d/%d, loss = %d' % (start + batch_size, r
  return l, beta_hat

l, beta_hat = lr_nn(X, y, 10)
```

```
## epoch 0/50:
##   100/1000, loss = 56
##   200/1000, loss = 64
##   300/1000, loss = 58
##   400/1000, loss = 57
##   500/1000, loss = 54
##   600/1000, loss = 58
##   700/1000, loss = 56
##   800/1000, loss = 56
##   900/1000, loss = 54
##   1000/1000, loss = 55
## epoch 1/50:
##   100/1000, loss = 54
##   200/1000, loss = 52
##   300/1000, loss = 52
##   400/1000, loss = 55
##   500/1000, loss = 56
##   600/1000, loss = 56
##   700/1000, loss = 56
##   800/1000, loss = 50
##   900/1000, loss = 52
##   1000/1000, loss = 51
## epoch 2/50:
##   100/1000, loss = 51
##   200/1000, loss = 54
##   300/1000, loss = 49
##   400/1000, loss = 58
##   500/1000, loss = 50
##   600/1000, loss = 50
##   700/1000, loss = 50
```

Put it in a Neural Network Diagram

Binary Logistic Regression, is in fact a single neuron firing a sigmoid probability-like number between 0 and 1, for each sample:



LR as NN in `Keras`

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

model = Sequential()
model.add(Dense(1, input_shape=(X.shape[1], ),
  activation='sigmoid', use_bias=False))
sgd = SGD(lr=0.1)
model.compile(loss='binary_crossentropy', optimizer=sgd)
model.fit(X, y, batch_size=100, epochs=50)

# Once tensorflow will have a build for python 3.8...

# Epoch 1/50
# 1000/1000 [==============================] - 0s 43us/step - loss
# Epoch 2/50
# 1000/1000 [==============================] - 0s 9us/step - loss.
# Epoch 3/50
# 1000/1000 [==============================] - 0s 10us/step - loss
# Epoch 4/50
# 1000/1000 [==============================] - 0s 12us/step - loss
# Epoch 5/50
# 1000/1000 [==============================] - 0s 9us/step - loss.
# Epoch 6/50
# 1000/1000 [==============================] - 0s 11us/step - loss
```

See that it makes sense:

```
beta_hat = model.get_weights() # Note Keras gives a list of weight

# [array([[1.0378177],
#         [1.9981958]], dtype=float32)]

pred = model.predict_proba(X)

# array([[0.06305372],
#        [0.15950003],
#        [0.08662305],
#        [0.1727513 ],
#        [0.28716186]]...

# See that it makes sense
pred_manual = 1/(1+np.exp(-np.dot(X, beta_hat[0])))

# array([[0.06305372],
#        [0.15950003],
#        [0.08662305],
#        [0.1727513 ],
#        [0.28716186]]...
```

Is that it?

1. No 😆

2. > The knee-jerk response from statisticians was "What's the big deal? A neural network is just another nonlinear model, not too different from many other generalizations of linear models". While this may be true, neural networks brought a new energy to the field. They could be scaled up and generalized in a variety of ways... and innovative learning algorithms for massive data sets."

(*Computer Age Statistical Inference* by Bradley Efron & Trevor Hastie, p. 352)

Add Classes

C Neurons for C Classes

Alternatively, we could:

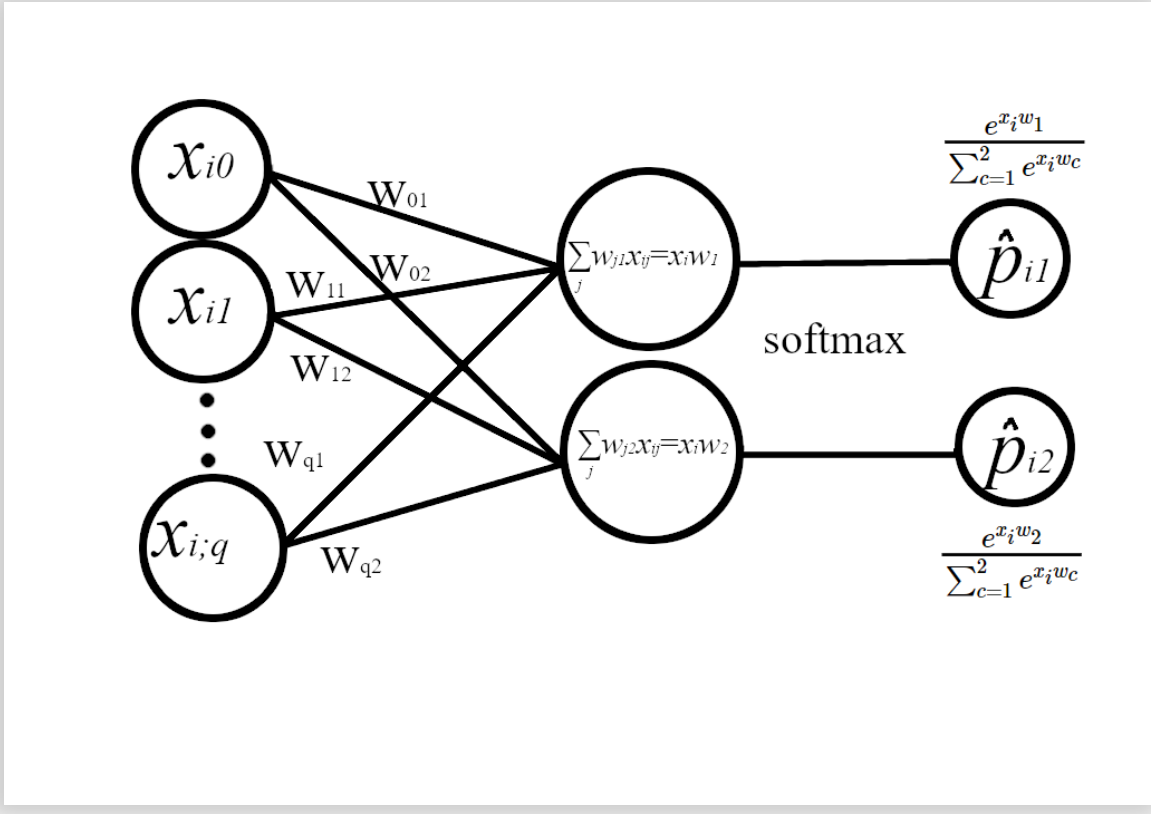
- fit a β vector for each class (or let's start talking about W)
- have C neurons for C classes
- where the output layer is the *Softmax Function*, to make sure the fitted $\hat{p}$ sum up to 1:

$$\hat{p}_{i;c} = \text{softmax}(c, W_{(q+1)\text{x}C}, x_i) = \frac{e^{x_i w_c}}{\sum_{c=1}^{C} e^{x_i w_c}}$$

Where x_i is the ith row of X as before and w_c is the cth row of W^T (or cth column of W)

> This would be equivalent to *multinomial logistic regression*!

So the architecture for 2 classes would be:



And in `Keras` we would do:

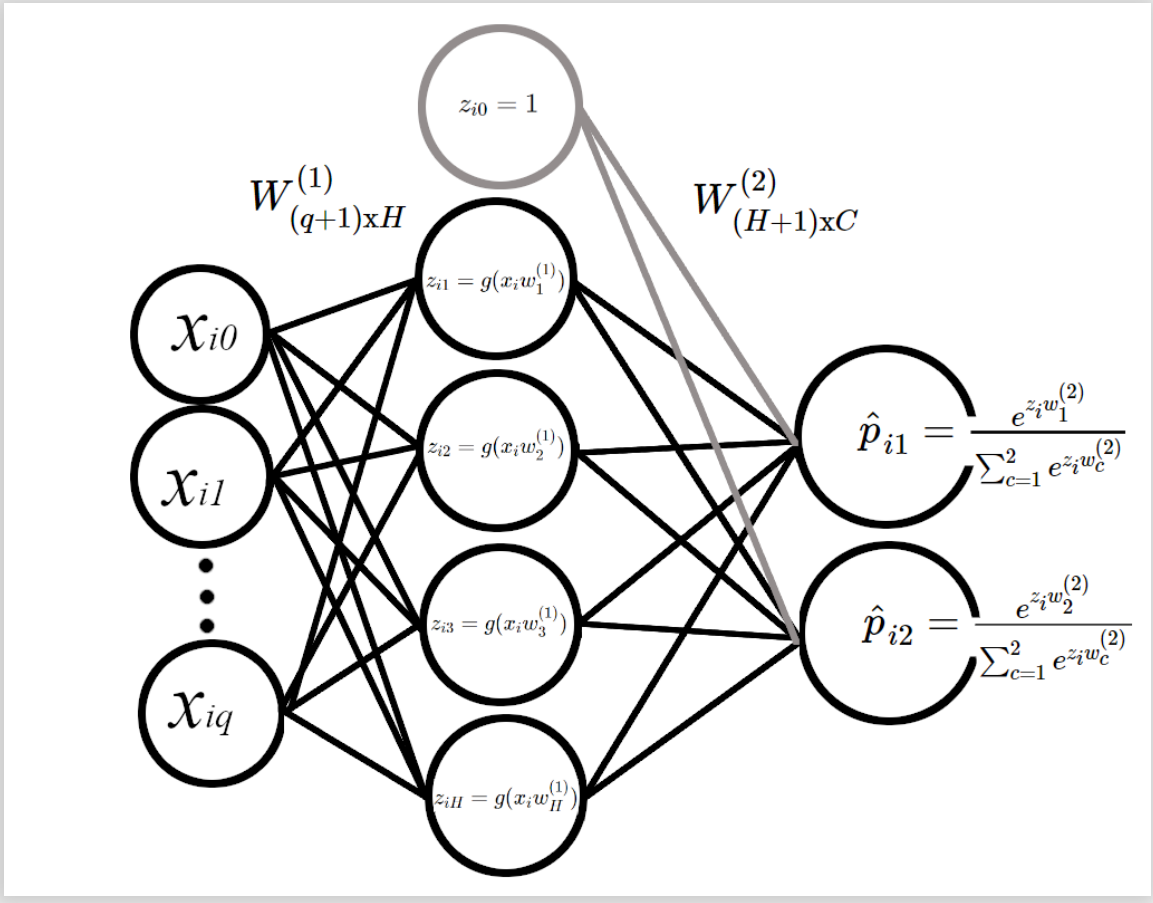
```
from keras.utils import to_categorical

y_categorical = to_categorical(y)
model = Sequential()
model.add(Dense(2, input_shape=(X.shape[1], ),
  activation='softmax', use_bias=False))
sgd = SGD(lr=0.1)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(X, y_categorical, batch_size=100, epochs=50)

# Epoch 1/50
# 1000/1000 [==============================] - 0s 34us/step - loss
# Epoch 2/50
# 1000/1000 [==============================] - 0s 9us/step - loss.
# Epoch 3/50
# 1000/1000 [==============================] - 0s 12us/step - loss
# Epoch 4/50
# 1000/1000 [==============================] - 0s 10us/step - loss
# Epoch 5/50
# 1000/1000 [==============================] - 0s 13us/step - loss
# Epoch 6/50
# 1000/1000 [==============================] - 0s 13us/step - loss
# Epoch 7/50
# 1000/1000 [==============================] - 0s 10us/step - loss
# Epoch 8/50
# 1000/1000 [==============================] - 0s 13us/step - loss
```

See that it makes sense:

```
W = model.get_weights()

# [array([[-0.14954337,  0.9368226 ],
#         [-1.6922047 ,  0.40227336]], dtype=float32)]

pred = model.predict_proba(X)

# array([[0.9441268 , 0.05587323],
#        [0.8506812 , 0.14931884],
#        [0.9219005 , 0.07809943],
#        [0.83749175, 0.16250822],
#        [0.7218378 , 0.27816215]...

Z = np.dot(X, W[0])
Z_exp = np.exp(Z)
Z_exp_sum = Z_exp.sum(axis=1)[:, None]
pred_manual = Z_exp / Z_exp_sum

# array([[0.9441268 , 0.05587323],
#        [0.8506812 , 0.14931884],
#        [0.9219005 , 0.07809943],
#        [0.83749175, 0.16250822],
#        [0.7218378 , 0.27816215]...
```

Add Hidden Layers

Don't Panic.



Where $g()$ is some non-linear *activation function*, e.g. sigmoid (but not often used).

- Notice we are not in Logistic Regression land anymore
- I have re-instated the bias terms
- I'm calling `model.summary()` to see no. of params

```
model = Sequential()
model.add(Dense(4, input_shape=(X.shape[1], ), activation='sigmoi
model.add(Dense(2, activation='softmax'))
sgd = SGD(lr=0.1)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(X, y_categorical, batch_size=100, epochs=50, verbose=0)

model.summary()
# Model: "sequential_1"
# _________________________________________________________________
# Layer (type)                 Output Shape              Param #
# =================================================================
# dense_1 (Dense)              (None, 4)                 12
# _________________________________________________________________
# dense_2 (Dense)              (None, 2)                 10
# =================================================================
# Total params: 22
# Trainable params: 22
# Non-trainable params: 0
# _________________________________________________________________
```

See that it makes sense:

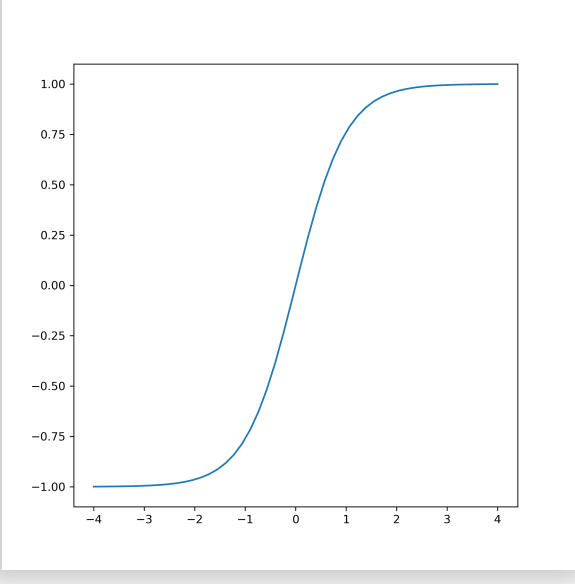
```
W1, b1, W2, b2 = model.get_weights()

W1.shape # (2, 4)
b1.shape # (4,)
W2.shape # (4, 2)
b2.shape # (2,)

W1 = np.vstack([b1, W1])
W2 = np.vstack([b2, W2])

W1.shape # (3, 4)
W2.shape # (5, 2)

# Get X ready with an intercept column
Xb = np.hstack((np.ones(n).reshape((n, 1)), X))
Xb.shape # (1000, 3)

pred = model.predict_proba(X)

# array([[0.90093774, 0.09906219],
#        [0.8220466 , 0.17795345],
#        [0.8905323 , 0.10946769],
#        [0.8115638 , 0.18843625],
#        [0.7119013 , 0.2880987 ]]...
```

```
Z = 1/(1+np.exp(-np.dot(Xb, W1)))
Zb = np.hstack((np.ones(n).reshape((n, 1)), Z))
Z2_exp = np.exp(np.dot(Zb, W2))
Z2_exp_sum = Z2_exp.sum(axis=1)[:, None]
pred_manual = Z2_exp / Z2_exp_sum

# array([[0.90093774, 0.09906219],
#        [0.8220466 , 0.17795345],
#        [0.8905323 , 0.10946769],
#        [0.8115638 , 0.18843625],
#        [0.7119013 , 0.2880987 ]]...
```

Activation Functions: Tanh

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

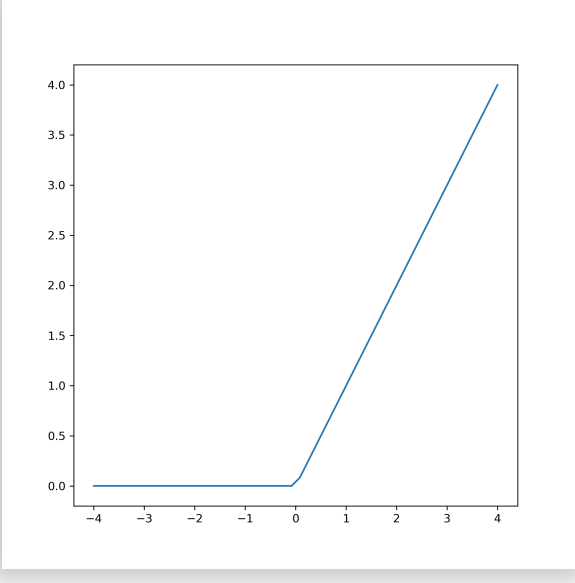
```
plt.clf()
plt.plot(X1, (np.exp(X1) - np.exp(-X1)) / (np.exp(X1) + np.exp(-X1
plt.show()
```



Activation Functions: ReLU

$$g(z) = \text{ReLU}(z) = max(z, 0)$$

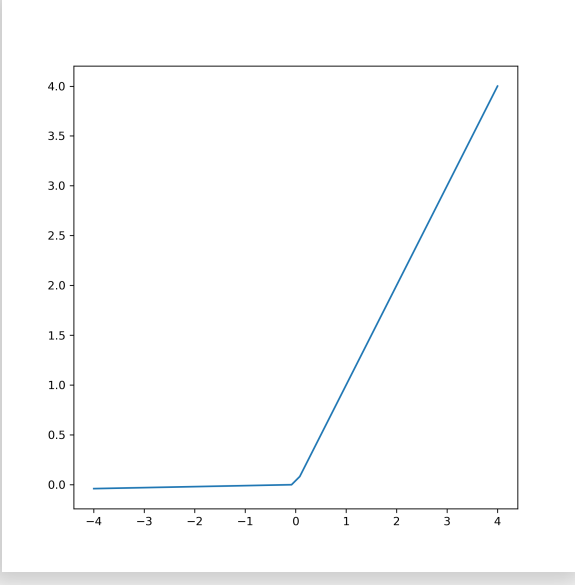
```
plt.clf()
plt.plot(X1, np.maximum(X1, 0))
plt.show()
```



Activation Functions: Leaky ReLU

$$g(z) = \text{LReLU}(z) = \begin{cases} z \geq 0 & z \\ z < 0 & \alpha z \end{cases}$$

```
plt.clf()
plt.plot(X1, np.where(X1 > 0, X1, X1 * 0.01))
plt.show()
```



Add Regularization

L1/L2 Regularization

You might have noticed neural networks intice you to add more and more params.

Therefore, NN are infamous for overfitting the training data, and some kind of regulariztion is a must.

Instead of minimizing some loss L (e.g. Cross Entropy) we add a penalty to the weights: $\min_W L(y, f(X; W)] + P(W)$

Where $P(W)$ would typically be:

- $P_{L_2}(W) = \lambda \sum_{ijk} (W_{ij}^{(k)})^2$
- $P_{L_1}(W) = \lambda \sum_{ijk} |W_{ij}^{(k)}|$
- or both (a.k.a Elastic Net, but not quite):
  $P_{L1L2}(W) = \lambda_1 \sum_{ijk} (W_{ij}^{(k)})^2 + \lambda_2 \sum_{ijk} |W_{ij}^{(k)}|$

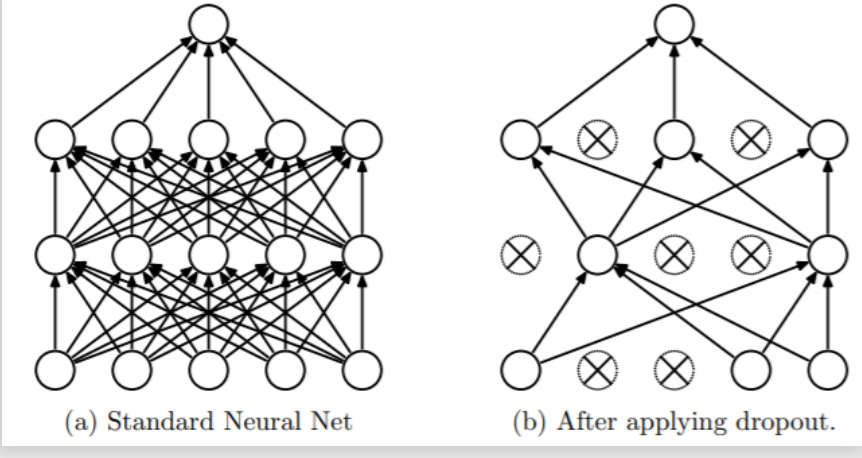
L1/L2 Regularization in `Keras`:

```
from keras import regularizers

model = Sequential()
model.add(Dense(4, input_shape=(X.shape[1], ), activation='relu',
  kernel_regularizer=regularizers.l1(0.01),
  bias_regularizer=regularizers.l2(0.01)))
model.add(Dense(2, activation='softmax',
  kernel_regularizer=regularizers.l1_l2(l1=0.01, l2=0.01)))
sgd = SGD(lr=0.1)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(X, y_categorical, batch_size=100, epochs=50, verbose=0)
```

Dropout

How to take neurons with a grain of salt?

During each epoch, individual neurons are either "dropped out" of the net with probability $1 - p$ (i.e. their weight is zero) or kept with probability p, so that a reduced network is left.



⚠ During prediction no Dropout is performed, but neurons output is scaled by p to make it identical to their expected outputs at training time.

Dropout in `Keras` (the `rate` parameter is the "fraction of the input units to drop"):

```
from keras.layers import Dropout

model = Sequential()
model.add(Dense(4, input_shape=(X.shape[1], ), activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(2, activation='softmax'))
sgd = SGD(lr=0.1)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(X, y_categorical, batch_size=100, epochs=50, verbose=0)
```

Early Stopping

Since NN are trained iteratively and are particularly useful on large datasets it is common to monitor the model performance using an additional validation set, or some of the training set. If you see no improvement in the model's performance (e.g. decrease in loss) for a few epochs - stop training.

```
from keras.callbacks import EarlyStopping

model = Sequential()
model.add(Dense(4, input_shape=(X.shape[1], ), activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(2, activation='softmax'))
sgd = SGD(lr=0.1)
callbacks = [EarlyStopping(monitor='val_loss', patience=5)]
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(X, y_categorical, batch_size=100, epochs=50,
  validation_split=0.2, callbacks=callbacks)
```

Deep Neural Networks