

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

2020-06-06

APPLICATIONS



OF DATA SCIENCE

Logistic Regression as *we* know it

APPLICATIONS



OF DATA SCIENCE

LR as GLM

- We observe y_1, \dots, y_n binary outcomes, therefore we say $Y_i \sim \text{Bernoulli}(p_i)$ and $P(Y_i) = p_i^{y_i} (1 - p_i)^{1-y_i}$
- We have $X_{n \times (q+1)}$ 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 $\beta_{(q+1) \times 1}$, 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:
$$\text{logit}(p_i) = \log\left(\frac{p_i}{1-p_i}\right) = x_i \beta$$

- 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:

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

- 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:

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

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

$$\begin{aligned} &\sum_{i=1}^n \ln[1 - g^{-1}(x_i\beta)] + y_i \ln\left[\frac{g^{-1}(x_i\beta)}{1-g^{-1}(x_i\beta)}\right] = \\ &\sum_{i=1}^n -\ln[1 + e^{x_i\beta}] + y_i x_i \beta \end{aligned}$$

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

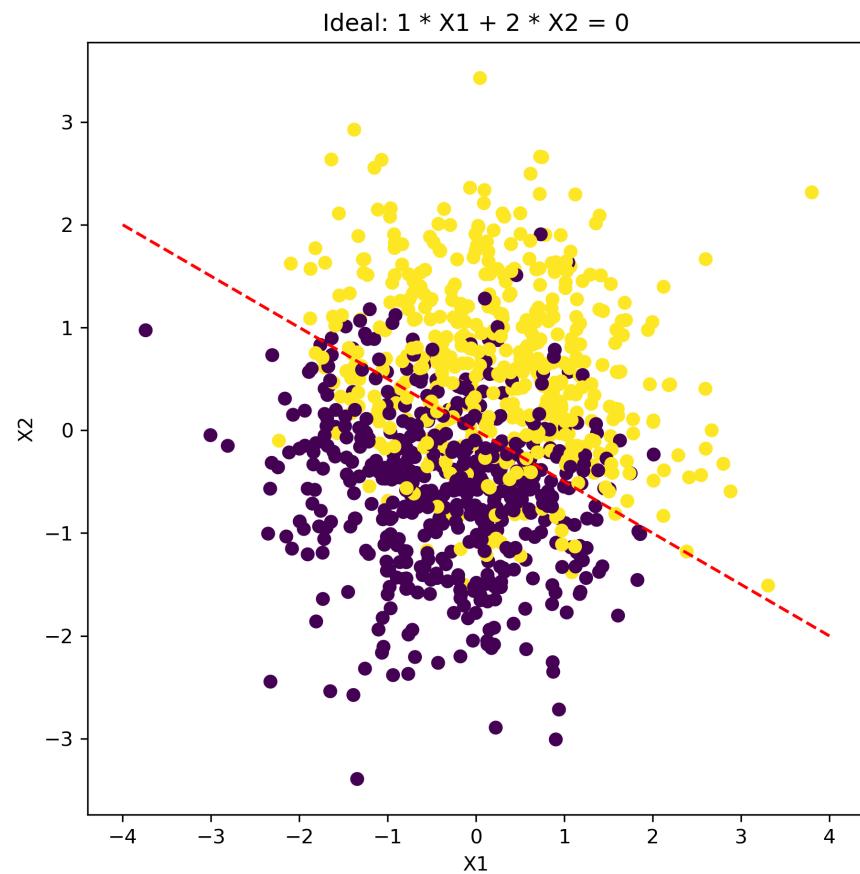
Show me that it's working

```
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:
```



sklearn should solve this easily:

```
from sklearn.linear_model import LogisticRegression

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

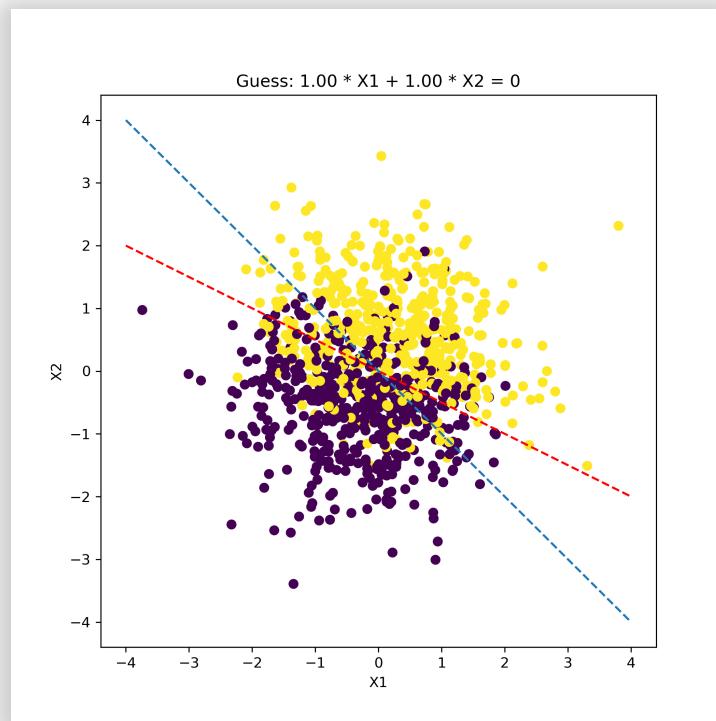
## LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=F
##                      intercept_scaling=1, l1_ratio=None, max_iter=100,
##                      multi_class='auto', n_jobs=None, penalty='none',
##                      random_state=None, solver='lbfgs', tol=0.0001, verbose=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

lr.coef_

## array([[0.94792812, 2.16330134]])
```

With Gradient Descent let's start with initial guess

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

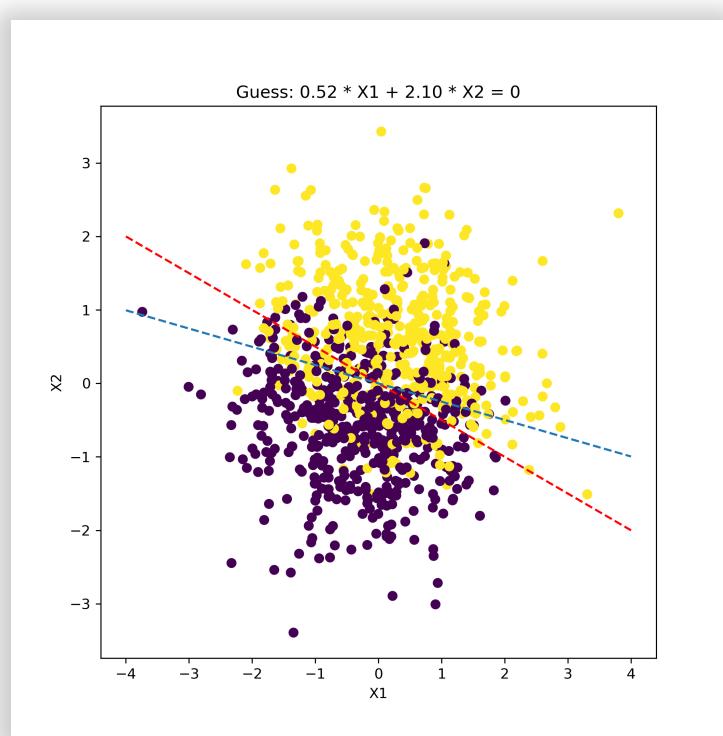


Let's do 1 iteration:

```
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

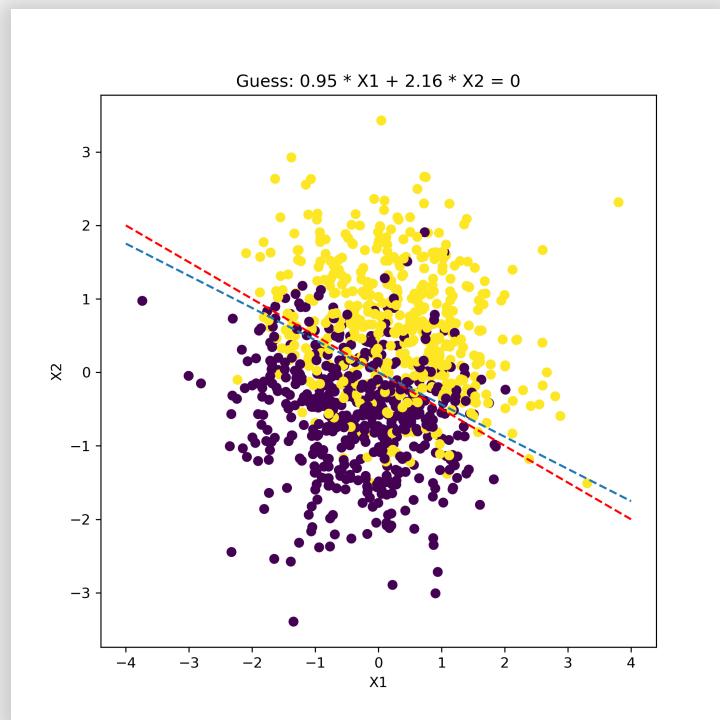
plot_sim()
```



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

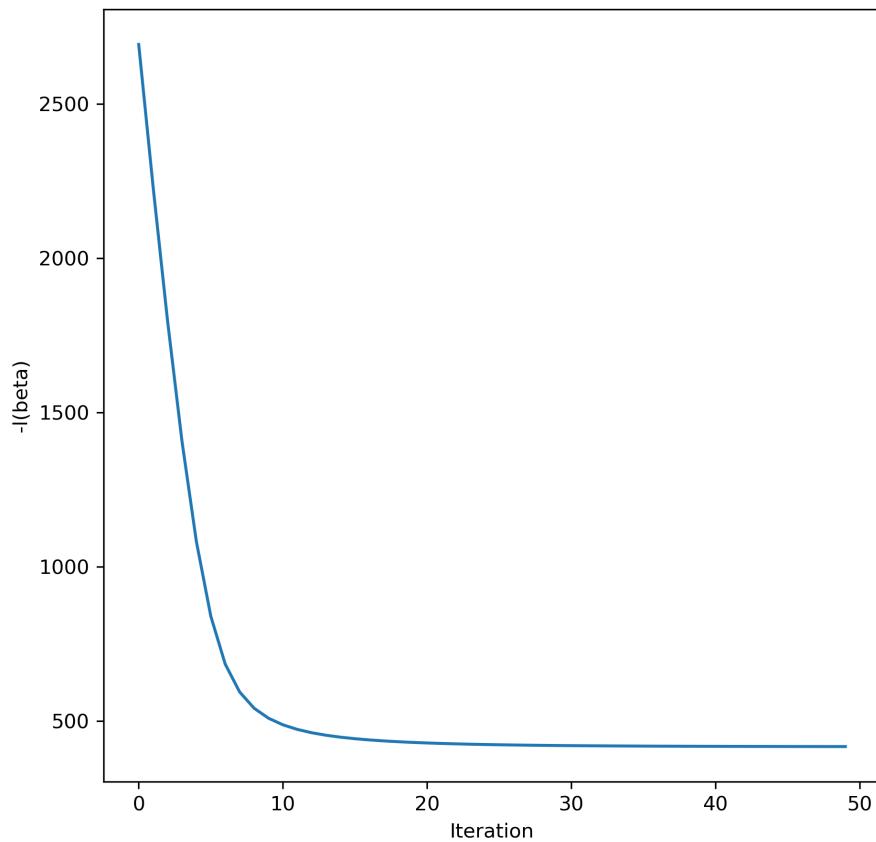
plot_sim()
```



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

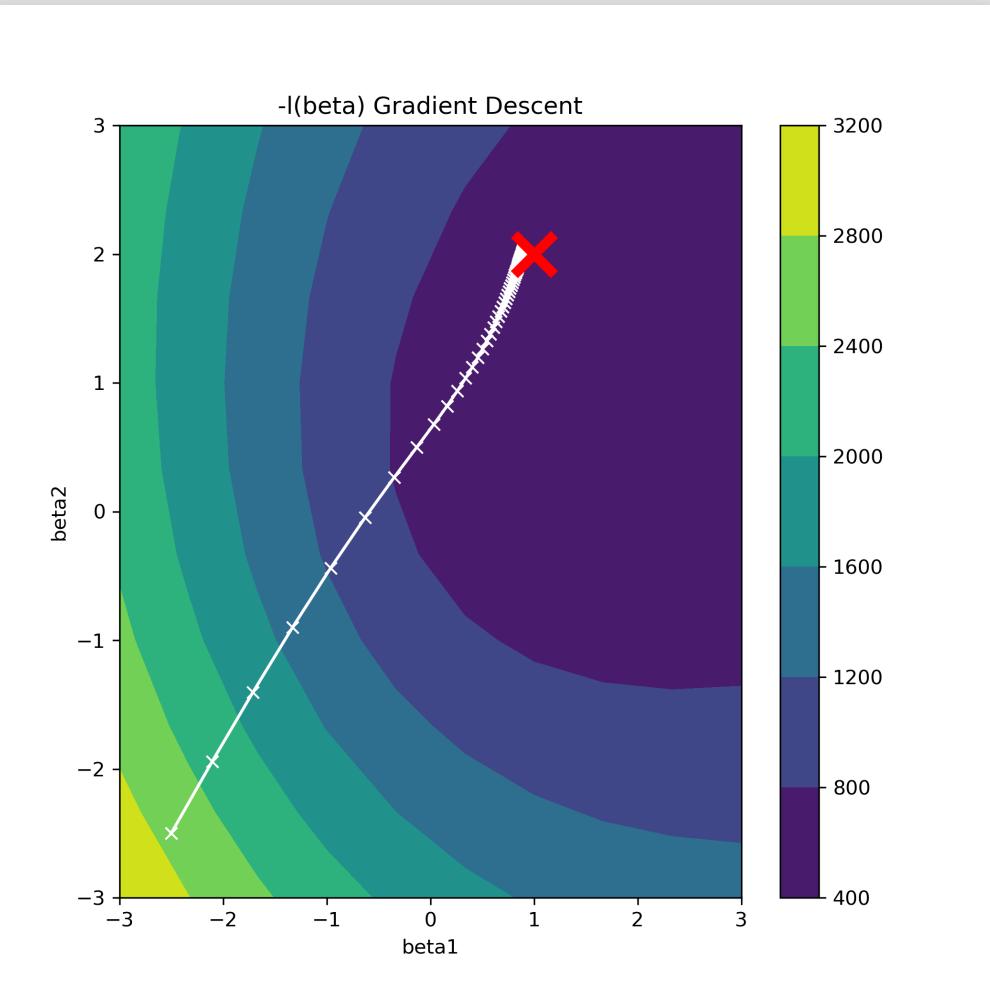
```
alpha = 0.001
beta_hat = np.array([-2.5, -2.5])
betas = [beta_hat]
ls = []
for i in range(50):
    p_hat = 1 / (1 + np.exp(-np.dot(X, beta_hat)))
    l_minus = -np.sum(y * np.log(p_hat) + (1 - y) * np.log(1 - p_hat))
    ls.append(l_minus)
    grad = -np.dot(X.T, (y - p_hat))
    beta_hat = beta_hat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

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



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_hat))
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='white')
ax.plot([beta[0]], [beta[1]], marker='x', color='red', markersize=10)
plt.show()
```



Logistic Regression as Neural Network

APPLICATIONS



OF DATA SCIENCE

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 $W_{(q+1) \times 1}$, 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:

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

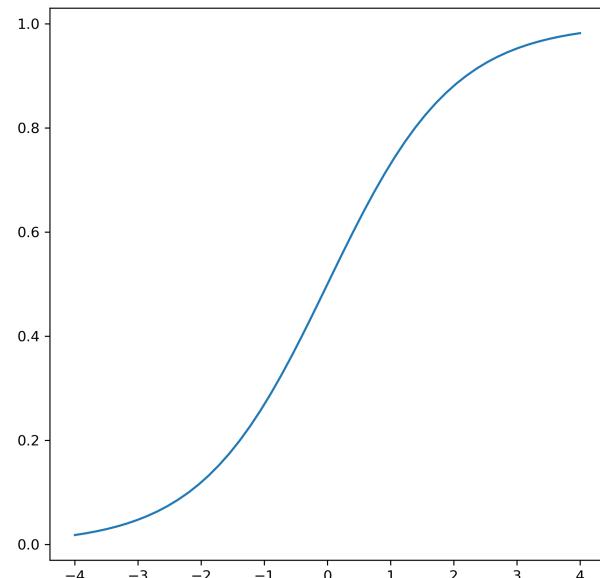
Which is exactly the contribution of the i th observation to $-l(\hat{\beta})$.

Sigmoid Function

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

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

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



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 exactly 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

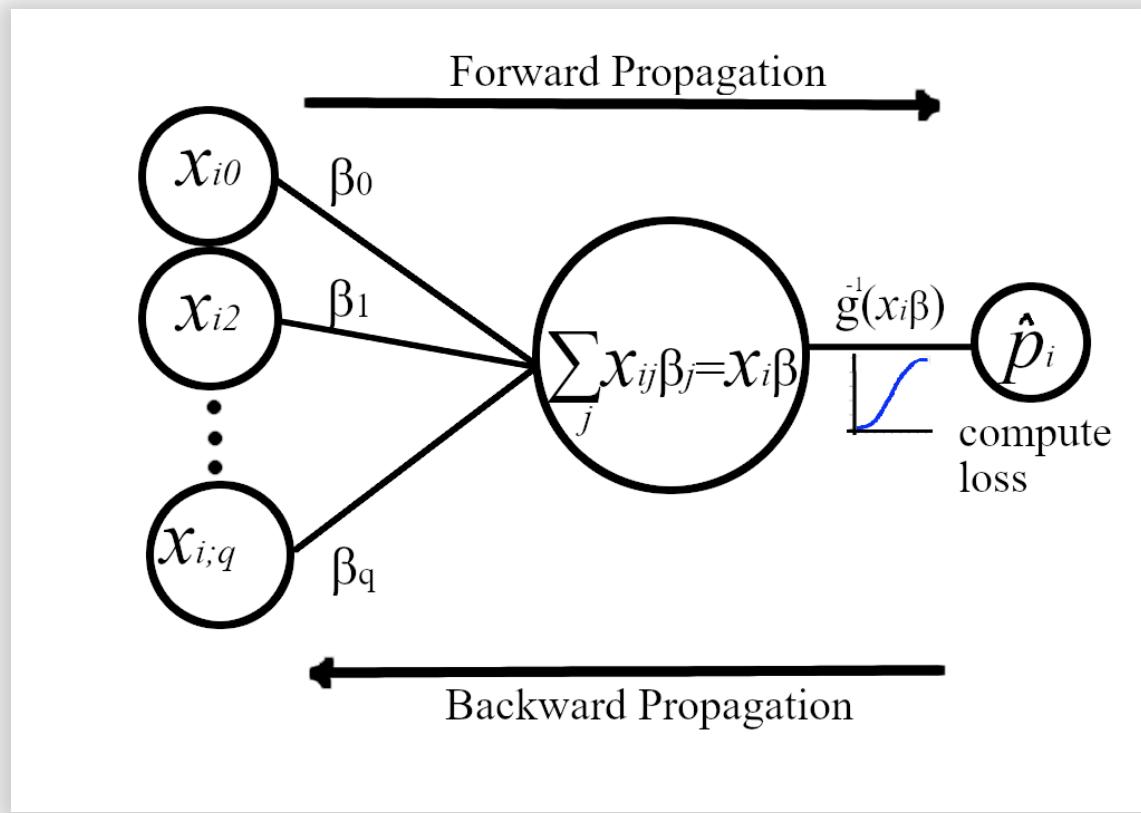
```
def lr_nn(X, y, epochs):
    beta_hat = np.random.rand(X.shape[1])
    alpha = 0.001
    batch_size = 100
    n = X.shape[0]
    steps = int(n / batch_size)
    for i in range(epochs):
        print('epoch %d:' % i)
        permute = np.random.permutation(n)
        X_perm = X[permute, :]
        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 1/50:  
## 100/1000, loss = 59  
## 200/1000, loss = 57  
## 300/1000, loss = 59  
## 400/1000, loss = 53  
## 500/1000, loss = 57  
## 600/1000, loss = 53  
## 700/1000, loss = 55  
## 800/1000, loss = 52  
## 900/1000, loss = 51  
## 1000/1000, loss = 57  
## epoch 2/50:  
## 100/1000, loss = 51  
## 200/1000, loss = 54  
## 300/1000, loss = 50  
## 400/1000, loss = 57  
## 500/1000, loss = 49  
## 600/1000, loss = 47  
## 700/1000, loss = 53  
## 800/1000, loss = 50  
## 900/1000, loss = 51  
## 1000/1000, loss = 55  
## epoch 3/50:  
## 100/1000, loss = 54  
## 200/1000, loss = 54  
## 300/1000, loss = 48  
## 400/1000, loss = 49  
## 500/1000, loss = 50  
## 600/1000, loss = 50  
## 700/1000, loss = 47
```

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 tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.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)

## Epoch 1/50
##
## 1/10 [==>.....] - ETA: 0s - loss: 1.2191
## 10/10 [=====] - 0s 797us/step - loss: 1.0212
## Epoch 2/50
##
## 1/10 [==>.....] - ETA: 0s - loss: 0.8968
## 10/10 [=====] - 0s 697us/step - loss: 0.8112
## Epoch 3/50
##
## 1/10 [==>.....] - ETA: 0s - loss: 0.7309
## 10/10 [=====] - 0s 704us/step - loss: 0.6790
## Epoch 4/50
##
```

See that it makes sense:

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

## [array([[0.9027474],
##          [2.0467637]], dtype=float32)

pred = model.predict(X)
pred[:3]

## array([[0.3520471 ],
##          [0.4607477 ],
##          [0.48886845]], dtype=float32)

pred_manual = 1/(1+np.exp(-np.dot(X, beta_hat[0])))
pred_manual[:3]

## array([[0.35204704],
##          [0.46074767],
##          [0.48886842]])
```

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

APPLICATIONS



OF DATA SCIENCE

C Neurons for C Classes

Alternatively, we could:

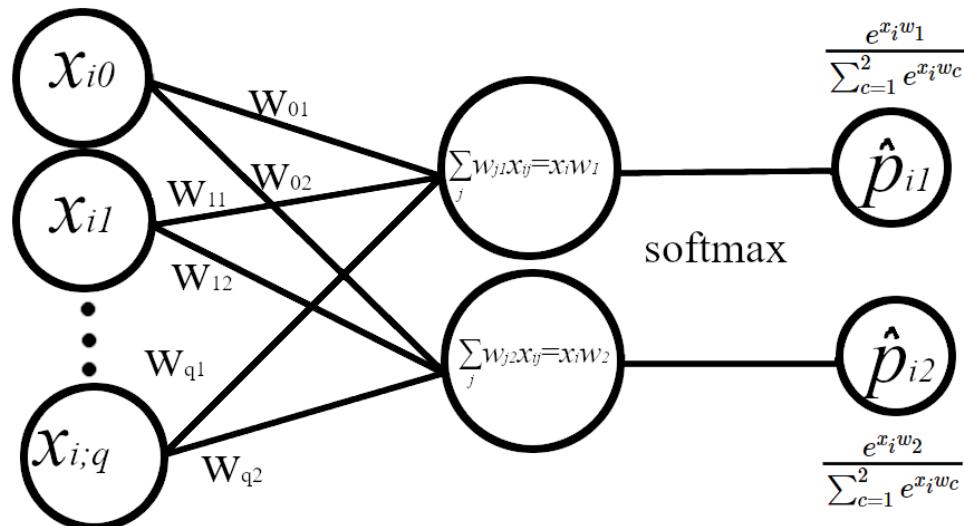
- fit a β vector for each class (or let's start talking about W)
- have C neurons for C classes
- where the output layer is the *Softmax Function*, to make sure the fitted \hat{p} sum up to 1:

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

Where x_i is the i th row of X as before and w_c is the c th row of W^T (or c th 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 tensorflow.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
##
## 1/10 [==>.....] - ETA: 0s - loss: 0.9543
## 10/10 [=====] - 0s 631us/step - loss: 0.7961
## Epoch 2/50
##
## 1/10 [==>.....] - ETA: 0s - loss: 0.6128
## 10/10 [=====] - 0s 604us/step - loss: 0.5933
## Epoch 3/50
##
## 1/10 [==>.....] - ETA: 0s - loss: 0.5704
## 10/10 [=====] - 0s 617us/step - loss: 0.5139
## Epoch 4/50
##
## 1/10 [==>.....] - ETA: 0s - loss: 0.4564
## 10/10 [=====] - 0s 600us/step - loss: 0.4771
```

See that it makes sense:

```
W = model.get_weights()  
W  
  
## [array([[-0.74589944,  0.19894522],  
##          [-0.28890684,  1.8619905 ]], dtype=float32)
```

```
pred = model.predict(X)  
pred[:3]
```

```
## array([[0.6550382 ,  0.34496185],  
##          [0.5403821 ,  0.45961788],  
##          [0.51036483,  0.48963526]], dtype=float32)
```

```
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  
pred_manual[:3]
```

```
## array([[0.65503816,  0.34496184],  
##          [0.5403821 ,  0.4596179 ],  
##          [0.51036482,  0.48963518]])
```

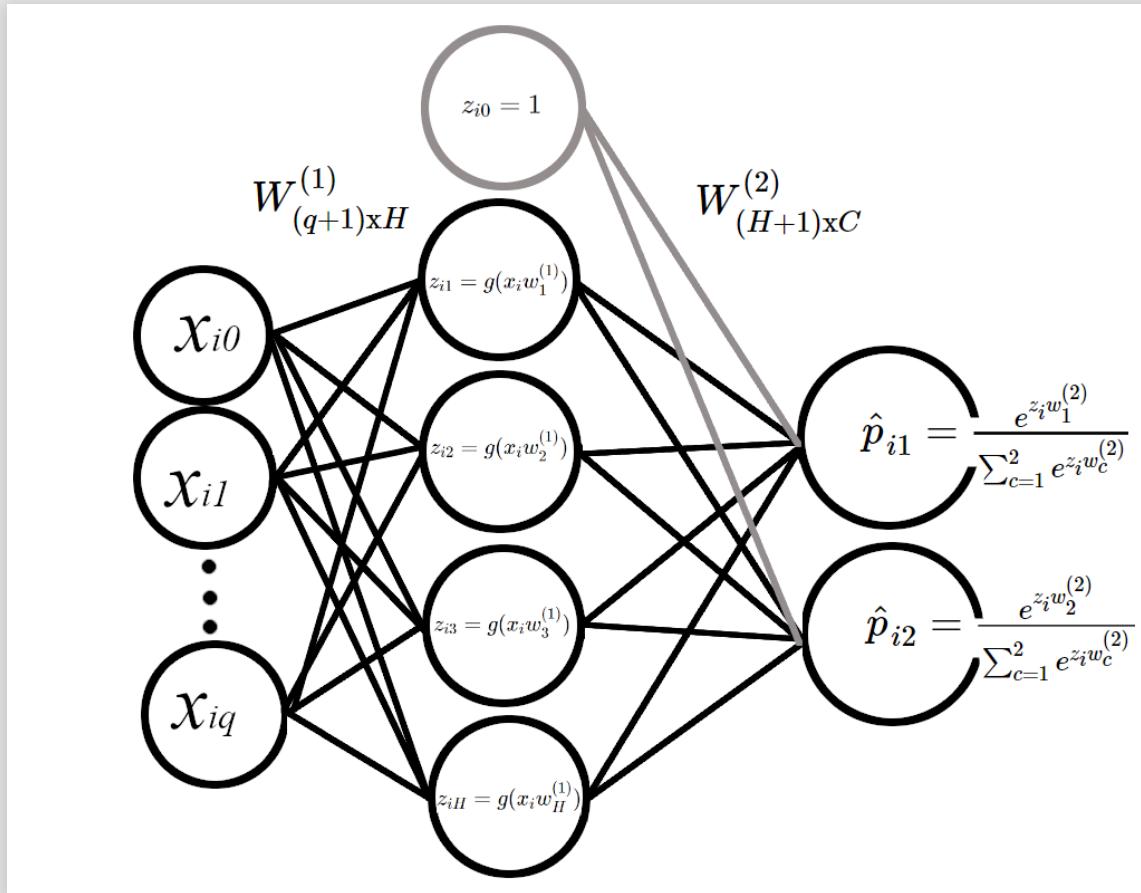
Add Hidden Layers

APPLICATIONS



OF DATA SCIENCE

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='sigmoid')
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()
```

```
## <tensorflow.python.keras.callbacks.History object at 0x00000000667B0D60>
```

```
## Model: "sequential_2"
##
##          Layer (type)           Output Shape        Param #
##          ======  ======  ======
##          dense_2 (Dense)        (None, 4)           12
##          dense_3 (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(X)

pred[:3]

## array([[0.68068177, 0.31931818],
##        [0.55274254, 0.44725746],
##        [0.5142706 , 0.48572943]], dtype=float32)
```

```
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

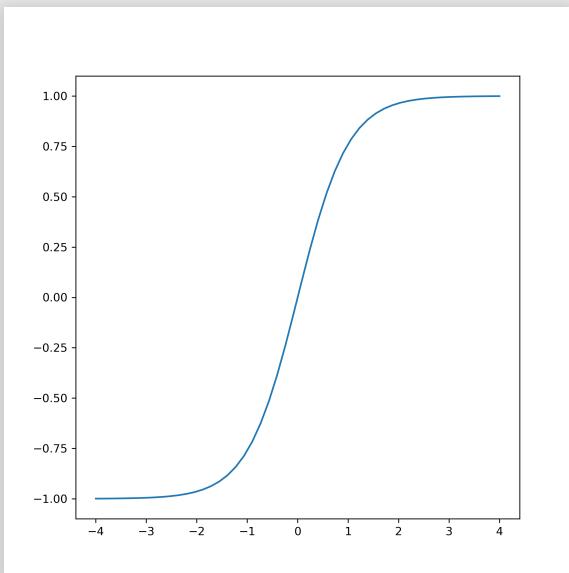
pred_manual[:3]
```

```
## array([[0.68068184, 0.31931816],
##        [0.55274257, 0.44725743],
##        [0.5142706 , 0.4857294 ]])
```

Activation Functions: Tanh

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

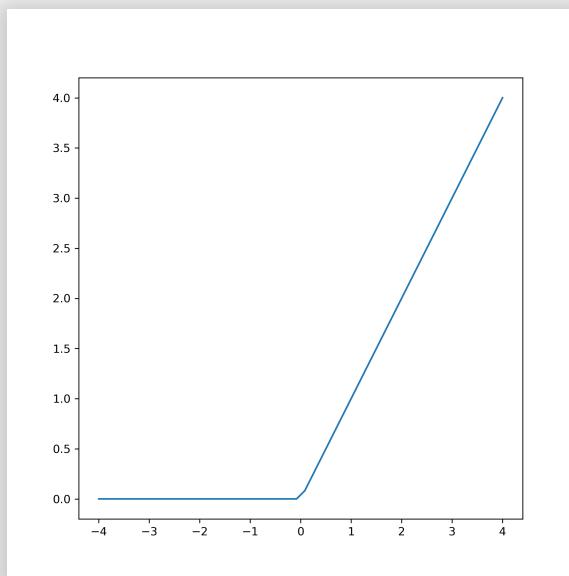
```
plt.clf()
plt.plot(X1, (np.exp(X1) - np.exp(-X1)) / (np.exp(X1) + np.exp(-X1)))
plt.show()
```



Activation Functions: ReLU

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

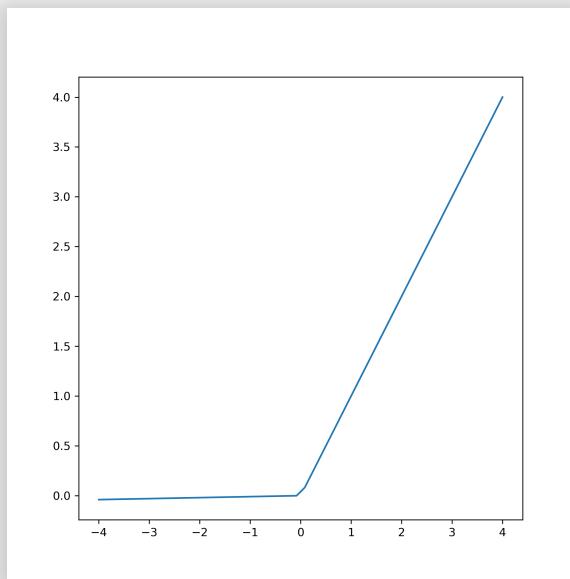
```
plt.clf()  
plt.plot(X1, np.maximum(X1, 0))  
plt.show()
```



Activation Functions: Leaky ReLU

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

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



Add Regularization

APPLICATIONS



OF DATA SCIENCE

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 regularization 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:

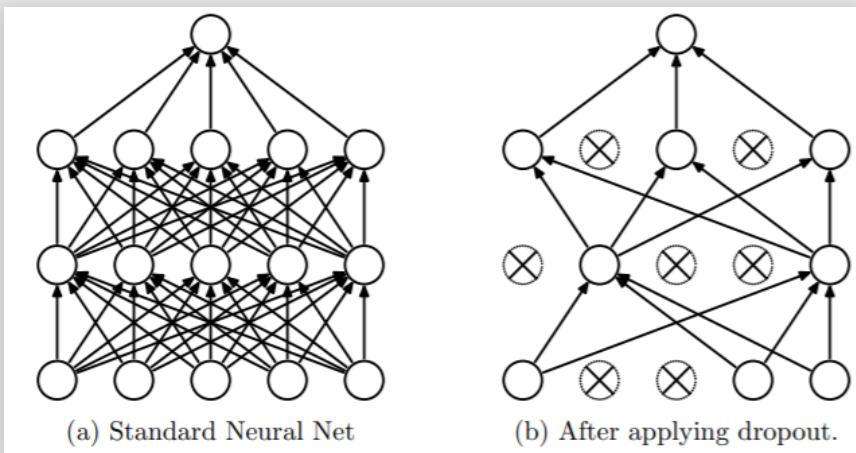
```
from tensorflow.keras import regularizers

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.

Why does it work? You could look at Dropout as an ensemble of neural networks! Each neuron can either count or not at each training step, so after 1K training steps you have virtually trained $1K$ slightly different models out of 2^N possible (where N is no. of neurons).

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

```
from tensorflow.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 tensorflow.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)
```

Batch Normalization

Keras (The Beloved)

APPLICATIONS



OF DATA SCIENCE

Keras is an API

- Keras is a high-level API "designed for human beings, not machines" developed by François Chollet
- It bridges to some popular DL backends such as Tensorflow, Theano, Apache MXNet
- But it is best integrated with Tensorflow, by Google. In fact, the formal docs of Keras now say:

```
import tensorflow as tf
from tensorflow import keras
```

- "ease of use does not come at the cost of reduced flexibility"
- Seamless integration with the Pandasverse
- Keras itself has a surprisingly great API in R, by RStudio
- You should know the competition by Facebook: PyTorch

Malaria!

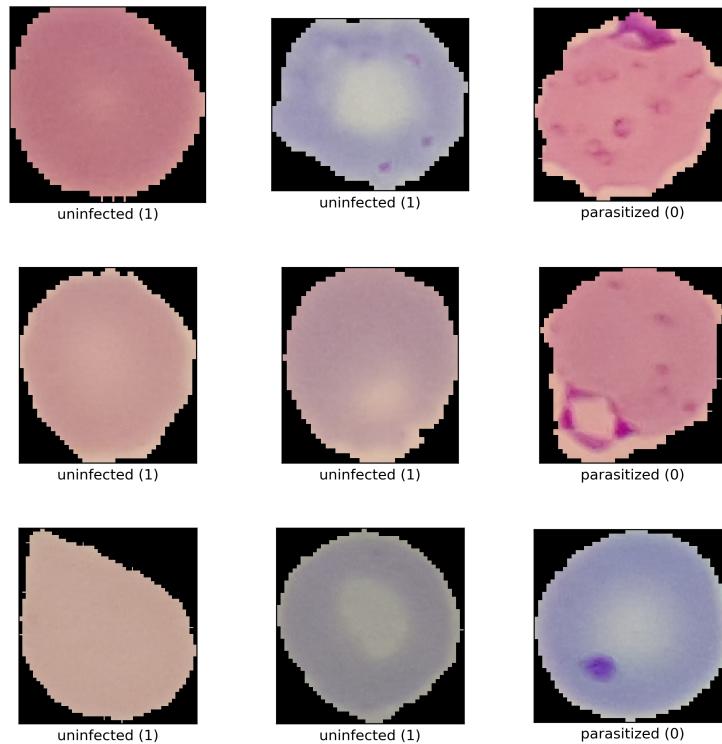
The [Malaria](#) dataset contains over 27K (processed and segmented) cell images with equal instances of parasitized and uninfected cells, from hundreds of patients in Bangladesh. The images were taken by a mobile application that runs on a standard Android smartphone attached to a conventional light microscope. The goal is "reduce the burden for microscopists in resource-constrained regions and improve diagnostic accuracy".

This dataset is part of the [tensorflow_dataset](#) library which gives you easy access to dozens of varied datasets.

Here I will take only ~10% of the images as a Numpy array and resize them all to 100x100 pixels, for the sake of speed.

```
import tensorflow_datasets as tfds
from skimage.transform import resize

malaria, info = tfds.load('malaria', split='train', with_info=True)
fig = tfds.show_examples(malaria, info)
```



```
from sklearn.model_selection import train_test_split

images = []
labels = []
for example in tfds.as_numpy(malaria):
    images.append(resize(example['image'], (100, 100)))
    labels.append(example['label'])
    if len(images) == 2500:
        break

X = np.array(images)
y = np.array(labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

X_train = X_train.flatten().reshape((X_train.shape[0], -1))
X_test = X_test.flatten().reshape((X_test.shape[0], -1))

print(X_train.shape)

## (2000, 30000)
```

```
print(X_test.shape)
```

```
## (500, 30000)
```

```

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='none', max_iter=100, random_state=42)
lr.fit(X_train, y_train)

## LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
##                     intercept_scaling=1, l1_ratio=None, max_iter=100,
##                     multi_class='auto', n_jobs=None, penalty='none',
##                     random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
##                     warm_start=False)
##
## C:\Users\gsimc\AppData\Local\Programs\Python\Python38\lib\site-packages\sklearn\linear_model\logistic.py:455: ConvergenceWarning: TOTAL NO. of ITERATIONS REACHED LIMIT.
## Increase the number of iterations (max_iter) or scale the data as shown in:
##     https://scikit-learn.org/stable/modules/preprocessing.html
## Please also refer to the documentation for alternative solver options:
##     https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
##     n_iter_i = _check_optimize_result()

lr.score(X_test, y_test)

## 0.63

```

The Sequential() API

```
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential()
model.add(Dense(300, input_shape=(30000,), activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Alternatively we could:

```
model = Sequential([
    Dense(300, input_shape=(30000,), activation='relu'),
    Dense(100, activation='relu'),
    Dense(50, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Make sure you get these numbers:

```
model.summary()
```

```
## Model: "sequential"
##
##             Layer (type)      Output Shape       Param #
## =====
##             dense (Dense)    (None, 300)        9000300
##             dense_1 (Dense)   (None, 100)         30100
##             dense_2 (Dense)   (None, 50)          5050
##             dense_3 (Dense)   (None, 1)           51
## =====
##             Total params: 9,035,501
##             Trainable params: 9,035,501
##             Non-trainable params: 0
##
```

 Are you at all worried?

Access layers and their weights:

```
model.layers
```

```
## [<tensorflow.python.keras.layers.core.Dense object at 0x00000000667C8AC0]
```

```
model.layers[0].name
```

```
## 'dense'
```

```
w1, b1 = model.get_layer('dense').get_weights()
```

```
print(w1.shape)
```

```
## (30000, 300)
```

```
w1
```

```
## array([[ 0.00080536, -0.01035658,  0.00614795, ...,  0.00070105,
##          0.01156601, -0.00157061],
##         [ 0.00391309,  0.00460913,  0.00586426, ..., -0.00503888,
##          -0.01196953, -0.00992271],
##         [ 0.00486556,  0.00683407,  0.0070433 , ...,  0.00927963,
##          -0.00292998, -0.00625381],
##         ...]]
```

Compiling your model:

```
model.compile(loss="binary_crossentropy",
  optimizer="adam",
  metrics=["accuracy"])
```

For more initialization schemes, losses, metrics and optimizers:

- <https://keras.io/api/layers/initializers/>
- <https://keras.io/api/losses/>
- <https://keras.io/api/optimizers/>
- <https://keras.io/api/metrics/>

Fitting the model:

```
from tensorflow.keras.callbacks import EarlyStopping

callbacks = [EarlyStopping(monitor='val_loss', patience=5,
                           restore_best_weights=True)]

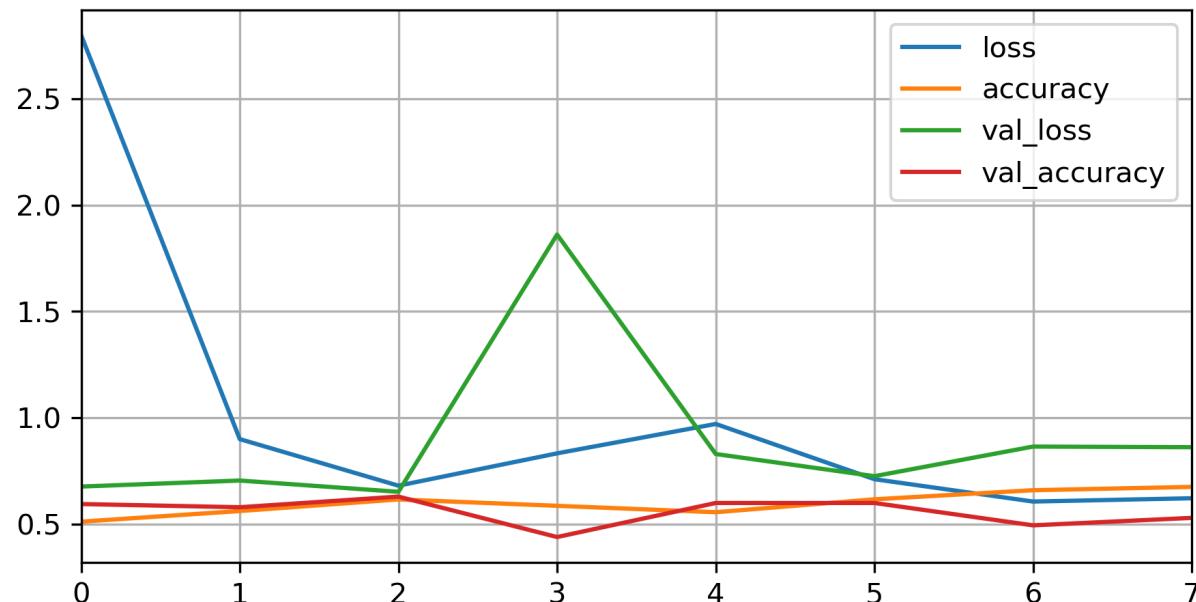
history = model.fit(X_train, y_train,
                      batch_size=100, epochs=50,
                      validation_split=0.1, callbacks=callbacks)
```

```
## Epoch 1/50
##
## 1/18 [>.....] - ETA: 0s - loss: 0.6941 - accuracy: 0.6941
## 2/18 [==>.....] - ETA: 2s - loss: 9.1782 - accuracy: 0.6941
## 3/18 [====>.....] - ETA: 2s - loss: 7.4989 - accuracy: 0.6941
## 4/18 [=====>.....] - ETA: 1s - loss: 6.1837 - accuracy: 0.6941
## 5/18 [=====>.....] - ETA: 1s - loss: 5.8417 - accuracy: 0.6941
## 6/18 [=====>.....] - ETA: 1s - loss: 5.1086 - accuracy: 0.6941
## 7/18 [=====>.....] - ETA: 1s - loss: 5.1150 - accuracy: 0.6941
## 8/18 [=====>.....] - ETA: 0s - loss: 5.0403 - accuracy: 0.6941
## 9/18 [=====>.....] - ETA: 0s - loss: 4.5974 - accuracy: 0.6941
## 10/18 [=====>.....] - ETA: 0s - loss: 4.2632 - accuracy: 0.6941
## 11/18 [=====>.....] - ETA: 0s - loss: 4.0184 - accuracy: 0.6941
## 12/18 [=====>.....] - ETA: 0s - loss: 3.7419 - accuracy: 0.6941
## 13/18 [=====>.....] - ETA: 0s - loss: 3.5272 - accuracy: 0.6941
## 14/18 [=====>.....] - ETA: 0s - loss: 3.3518 - accuracy: 0.6941
## 15/18 [=====>.....] - ETA: 0s - loss: 3.1799 - accuracy: 0.6941
## 16/18 [=====>.....] - ETA: 0s - loss: 3.0000 - accuracy: 0.6941
```

See later the history object's many fields.

```
import pandas as pd

pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```



Evaluate on test set:

```
model.evaluate(X_test, y_test)
```

```
##  
## 1/16 [>.....] - ETA: 0s - loss: 0.6902 - accuracy: 0.6597  
## 7/16 [=.....=>.....] - ETA: 0s - loss: 0.6390 - accuracy: 0.6621  
## 15/16 [=.....=>..] - ETA: 0s - loss: 0.6562 - accuracy: 0.6621  
## 16/16 [=.....=] - 0s 9ms/step - loss: 0.6597 - accuracy: 0.6621  
## [0.6597498655319214, 0.621999979019165]
```

```
from sklearn.metrics import confusion_matrix  
  
y_pred = (model.predict(X_test) > 0.5).astype(int).reshape(y_test.shape)
```

```
confusion_matrix(y_test, y_pred),  
index=['true:yes', 'true:no'],  
columns=['pred:yes', 'pred:no'])
```

```
##          pred:yes  pred:no  
## true:yes      147      115  
## true:no       74      164
```

Tuning params:

```
from tensorflow.keras.layers import InputLayer
from tensorflow.keras.optimizers import SGD

def malaria_model(n_hidden=1, n_neurons=30, lrt=3e-3):
    model = Sequential()
    model.add(InputLayer(input_shape=(30000, )))
    for layer in range(n_hidden):
        model.add(Dense(n_neurons, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss="binary_crossentropy",
                  optimizer=SGD(lr=lrt),
                  metrics=["accuracy"])
    return model

keras_clf = keras.wrappers.scikit_learn.KerasClassifier(malaria_m
```

```

from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

params = {
    'n_hidden': [0, 1, 2, 3],
    'n_neurons': np.arange(1, 100),
    'lrt': reciprocal(3e-4, 3e-2)
}

rnd_search_cv = RandomizedSearchCV(keras_clf, params, cv=5,
                                    n_iter=10)
rnd_search_cv.fit(X_train, y_train, epochs=50,
                   validation_split=0.1, callbacks=callbacks)

print(rnd_search_cv2.best_score_)
print(rnd_search_cv2.best_params_)

## 0.6630000114440918
## {'lrt': 0.01678485252717544, 'n_hidden': 0, 'n_neurons': 15}

```

See also sklearn's [GridSearchCV\(\)](#) and [KerasTuner](#) for a more robust solution.

Saving and restoring a model:

```
model.save('malaria.h5')
```

Then:

```
model = keras.models.load_model('malaria.h5')
model.predict(X_test[:3])
```

```
## array([[0.53044856],
##        [0.7757778 ],
##        [0.27708748]], dtype=float32)
```

The HDF5 model saves the model's architecture and hyperparameters, and all weights matrices and biases.

Also see the `ModelCheckpoint()` callback.

Few Excellent Books

