Types of Learning

- Supervised
  - *Structured/labeled data*
  - Ex: picture → label
  - Ex: picture → digit number
- Unsupervised
  - *Unstructured data*
  - Dataset of pictures(not labeled)
- **m** denotes # of training examples

Binary Classification

- Classifies whether the input (is/has) or (isn't/doesnt have) a particular thing.
- Ex: Given an image, tell whether it has a cat in it or not 1(cat) vs 0(non-cat).
- In general, computers store images in 3 matrices(red, green, and blue) each of size width x height
- Images are stretched into a single column vector, usually starting with all the red pixels then all the green ones, then all the blue ones.
- n_x = the dimension of the input features
  - Ex: 64 x 64 x 3 = n_x = 12288
- A single training example is denoted as:
  - (x, y) where $x \in \mathbb{R}^{n_x}$ and $y \in \{0, 1\}$
- m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
  - Other common notation:
    - $m = m_{train}$
    - m_{test} = # test examples
- X is used to denote a matrix of all the inputs
  - $X \in \mathbb{R}^{n_x \times m}$
  - X.shape = (n_x, m)
- Y is used to denote a row vector of all the labels
  - $Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$
  - $Y \in \mathbb{R}^{1 \times m}$
  - Y.shape = $(1, m)$

Logistic Regression

- Given x, we want $\hat{y} = P(y = 1 \mid x)$
- Parameters:
  - $x \in \mathbb{R}^{n_x}$
  - $b \in \mathbb{R}$
- Output:
  - $\sigma(w^T x + b)$

- We use the *sigmoid* function to make sure that the output is between 0 and 1, and centered on 0.5

Logistic Regression Cost Function

- Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$, we want $\hat{y}^{(i)} \approx y^{(i)}$
- Loss(error) function:
  - Computes the error for a single training example
  - $L(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log (1 - \hat{y}))$
  - If $y = 1 : \mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ wants $\hat{y}$ to be large
  - If $y = 0 : \mathcal{L}(\hat{y}, y) = -\log (1 - y) \leftarrow$ wants $\hat{y}$ to be small
- Cost function:
  - Averages the loss of the entire training set
  - $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$i_

Gradient Descent

- $w := w - \alpha \frac{dJ(w,b)}{dw}$
- $b := b - \alpha \frac{dJ(w,b)}{db}$
- α is called the **Learning Rate**

Neural Network Representation

- Each layer is denoted by $a^{[n]}$
  - a is a column vector representing activations of neurons
  - The superscript n represents the layer that we are looking at
- The input layer is denoted $a^{[0]}$

Activation Functions

- Activation functions are denoted $g^{[n]}$
  - g represents the function
  - n represents the layer
- Previously, we used Sigmoid(σ)
  - Output is always between 0 and 1
  - Centered on 0.5
- Tanh is another alternative
  - Output is always between -1 and 1
  - Centered on 0
- Tanh is usually much better except for the output layer for binary activation
- ReLU = $\max(0, z)$
  - Rectified Linear Unit
- Leaky ReLU is another option
- No activation function is called a **linear Activation** function.
  - Its very uncommon to use a linear activation function.

- Sometimes used for the output layer when $y \in \mathbb{R}$

L-layer Deep Neural Network

- Forward Pass
  - $Z^{[\ell]} = W^{[\ell]} \cdot A^{[\ell-1]} + b^{[\ell]}$
  - $A^{[\ell]} = g^{[\ell]}(Z^{[\ell]})$
- Backward Pass
  - $dZ^{[\ell]} = dA^{[\ell]} * g^{[\ell]}{}^{\backprime}(Z^{[\ell]})$
  - $dW^{[\ell]} = \frac{1}{m} dZ^{[\ell]} \cdot A^{[\ell-1]T}$
  - $db^{[\ell]} = \frac{1}{m}\text{np.sum}(dZ^{[\ell]}, \text{axis=1})$
  - $dA^{[\ell-1]} = W^{[\ell]T} \cdot dZ^{[\ell]}$

Train/Dev/Test Sets

- For small datasets, you have to allocate a larger % of examples for testing and validation.
  - Ex: 70/30 train/test split
- For larger datasets, a much larger % of examples are used for testing.
  - Ex: 99/1/1 train/dev/test split
- Always make sure the dev and test sets come from the same distribution.

Bias/Variance

- High **Bias** means *underfitting*.
  - Ex: 15% training error and 16% dev error
- High **Variance** means *overfitting*.
  - Ex: 1% training error and 11% dev error.
- High bias and high variance can both be present.
  - Ex: 15% training error and 30 % dev error.
- In between high bias and high variance is **just right**
  - Ex: 0.5% training error and 1% dev error.

Basic Recipe for Machine Learning

- Issues with high bias?
  - Increase the size of your network
  - Train longer
- High variance?
  - Try to get more data
  - Regularization
  - More appropriate architecture

L2 Regularization

- **L2 Regularization**
  - *Sum of all the weights squared*

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

Dropout Regularization

- keep_prob = 0.8
- d3 = np.random.randn(a3.shape[0], a3.shape[1]) < keep_prob
- a3 = np.myltiply(a3, d3)
- a3 /= keep_prob
  - This is called *Inverted Dropout*
  - This line ensures that the values are bumped up for the next layer

Normalizing Inputs

- Normalizing Inputs allows us to use larger learning rates because we wont be oscilating back and forth in our loss function
- Subtract Mean
  - $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
  - $x := x - \mu$
- Normalize Variance
  - $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x * *2$
    - element-wise squaring
  - $x/ = \sigma^2$

Gradient Check

- Compute the gradient, $d/theta$, then compute the limit of the gradient and compare the two in euclidean distance
- 10^{-7} is usually considered a fine margin of difference
- 10^{-5} is questionable but may be fine
- 10^{-3} is very likely that something is wrong

Mini-batch Gradient Descent

- t denotes the mini-batch index we are using for our dataset
- $t = 1, 2, \ldots, 5000$ for a dataset split into 5000 batches
- $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
- $A^{[1]} = g^{[1]}(Z^{[1]})$
- $\ldots$ and so on$

- Cost is then computed by dividing by the size of the batch instead of $\frac{1}{m}$
- *epoch* is defined as a single pass through the training set
- Tends to make the cost-graph oscilate a bit
- Batch size is a hyper-parameter between 64 and 512
  - Usually a power of 2

Batch Gradient Descent

- Mini-batch Gradient Descent with a batch size of m
- Very large steps
- Takes too long per iteration

Stochastic Gradient Descent(SGD) - Mini-batch Gradient Descent with a batch size of 1 - Tends to be very noisy and wanders quite a bit - Never exactly converges - Lose speedup from vectorization**

Exponentially Weighted Average

- $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
- Averages your data over the last $\frac{1}{1-\beta}$ datapoints
- β is usually between 0 and 1
- Smooths out a pretty noisy graph(like a graph of temperature)
- The higher the value of β, the slower it is to react, so keep that in mind
- the lower the value of β, the faster it is to react, but the more noisy it gets

Gradient Descent with Momentum

- $V_{dW} = \beta V_{dW} + (1-\beta)dW$
- $V_{db} = \beta V_{db} + (1-\beta)db$
- $W = W - \alpha V_{dW}$
- $b = b - \alpha V_{db}$
- You can use momentum to make bigger weight updates at first then smaller updates later on

Learning Rate Decay

- Decreasing the learning rate as time goes on, so fewer and fewer steps are being made
- Helps convergence at the end while having big steps at the beginning
- $\alpha = \frac{1}{1+\text{decay-rate}} * \text{epoch-num}\alpha_0$
- The learning rate becomes a function of the epoch number

Hyperparameter Tuning

- Start with learning rate α
- Then try momentum β
- Then try # hidden units
- Then try mini batch size
- Lastly, try # of layers and the learning rate decay.
- Use random values. Do not use a *grid*
- Use logarithmic scale to make sure you are focusing resources where they are most likely best
- $r = -4 * $ np.random.rand() which makes $r \in [-4, 0]$
- $\alpha = 10^r$
- Try to train multiple models at once and cull them as they become clearly bad
  - The *caviar* approach
- Train one model at a time and fidget with settings till you get it right
  - The *panda* approach

Batch Normalization

- Normalizing Z to speed up learning

Softmax Regression for Multi-class Classificiation

- C is used to denote the number of classes
- $n^{[L]} = C$
  - The output layer has C nodes
- $n_i^{[L]} = P(\hat{y} = i|x)$