A brief summary on Machine Learning

Nicholas Funari Voltani *

August 4, 2020

Contents

1 What is Machine Learning? **2**

2 Supervised Learning **2**

3 Regression Problem: Predicting Continuous Outputs **3**
3.1 Linear Regression . . . 3
3.2 Gradient Descent Method . . . 3
3.3 Normal Equation . . . 5
3.4 Polynomial Fits and Normalization of Features . . . 5

4 Classification Problem: Predicting Discrete Outputs **5**
4.1 Logistic Regression . . . 5
4.2 Decision Boundary . . . 6
4.3 Cost Function for Logistic Regression . . . 6
4.4 Multiclass Classfication: One vs. All . . . 8

5 Neural Networks **8**
5.1 Intuition/Motivation . . . 8
5.2 Forward propagation . . . 8
5.3 Cost Function . . . 10
5.4 Backpropagation . . . 11

6 Bias and Variance **13**

7 Cross-Validation **15**

8 Support Vector Machines **15**
8.1 The original problem . . . 16
8.2 Solving for the optimal margin . . . 17

9 Unsupervised Learning **18**

10 K-Means Clustering Algorithm **18**

11 Principal Component Analysis; Dimensionality Reduction **19**

12 Anomaly Detection: Probability Distributions **21**

*nicholas.voltani@usp.br

13 Alternative Methods in Machine Learning **22**
13.1 Stochastic Gradient Descent 22
13.2 Mini-batch Gradient Descent 22
13.3 Mathematical Interlude: Exponentially Weighted [Moving] Averages 23
13.4 Adam Algorithm [16] 24
13.5 Batch Normalization [17] 24

The following text was written as a summary of the course I'm taking on Machine Learning, by Andrew Ng [1], as well as other interesting things I pick up along the way.

1 What is Machine Learning?

Arthur Samuel defined, in 1959, machine learning as

> "*the field of study that gives computers the ability to learn without being explicitly programmed.*"

A more modern definition was given by Tom Mitchell:

> "*A computer is said to learn from experience E with respect to some task T and some performance measure P, if its performance on said task T (as measured by given P) improves with experience E.*"

Example 1.1. Suppose we need an algorithm which learns to recognize spam. We then have:

- Experience E: watching as the user labels emails as spam or not;
- Task T: labeling emails as spam without the user's command;
- Measure P: ratio of the number of emails autosent to spam with respect to the number of emails sent to spam by the user.

What's desired is that the algorithm improves its efficiency (with respect to the measure P) on its given task T, and that happens as it analyzes and interprets the data (emails) which is provided (experience E).

2 Supervised Learning

Supervised Learning presupposes the "right" answer to a problem, and tries to adjust to it. It essentially boils down to trying to adjust datasets to some hypothesis function, be it linear or non-linear.

The *modus operandi* in supervised learning is straight-forward (in theory):

1. there is a training set, determined by a matrix X;
2. determine which hypothesis model $h_\theta(\vec{x})$ will be used to fit the dataset X;
3. define a cost function $J(\vec{\theta})$ (where $\vec{\theta}$ is determined by the hypothesis), which determines the error between the hypothesis fit and the dataset;
4. minimize $J(\vec{\theta})$ with respect to $\vec{\theta}$;
5. use $\vec{\theta}$ in $h_\theta(\vec{x})$ as the prediction model.

Some caution is in order, as $J(\theta)$ may have multiple local minima, for instance. However, in the following sections, $J(\theta)$ will be a convex function (thus, will have a global maximum).

The problems concerning supervised learning are divided in two groups: regression problems and classification problems.

3 Regression Problem: Predicting Continuous Outputs

3.1 Linear Regression

Definition 3.1 (Cost Function - Linear Regression)**.** *Let $\{x^{(i)}\}_{i=1}^m$ be inputs, and $\{y^{(i)}\}_{i=1}^m$ be their respective values. Then define the cost function by the quadratic error between the hypothesis $h_\theta(x)$ and the values $y^{(i)}$ associated with each dataset point $x^{(i)}$:*

$$J(\theta) \equiv \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

Example 3.1.
For simplicity, let's consider, for starters, a dataset where $\{x^{(i)}\}_{i=1}^m$ are the inputs, and $\{y^{(i)}\}_{i=1}^m$ are their respective values ($x^{(i)}, y^{(i)}$ are just numbers). So we have m pairs $(x^{(i)}, y^{(i)})$.

Assume we want to fit a linear function to the dataset; this means

$$h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}\,,\ \theta_0, \theta_1 \in \mathbb{R}$$

Just for convenience, we can attach to each $x^{(i)}$ another term, in such a way that

$$x^{(i)} \mapsto \vec{x}^{(i)} \equiv \begin{pmatrix} 1 & x^{(i)} \end{pmatrix}$$

By making this change, we can then write

$$h_\theta(\vec{x}^{(i)}) = \vec{x}^{(i)} \cdot \vec{\theta}$$

We can have a dataset with multiple features for each training instance. Let there be n features for each m training input. We then write the dataset matrix as

$$X = \begin{pmatrix} \text{—} & \vec{x}^{(1)} & \text{—} \\ & \vdots & \\ \text{—} & \vec{x}^{(m)} & \text{—} \end{pmatrix} \in M_{m\times(n+1)}(\mathbb{R})$$

where[1] $\vec{x}^{(i)} = \begin{pmatrix} 1 & x_1^{(i)} & \dots & x_n^{(i)} \end{pmatrix}$. With this dataset matrix, we have

$$h_\theta(\vec{x}^{(i)}) = \vec{x}^{(i)} \cdot \vec{\theta} = (X\vec{\theta})^{(i)}$$

This is called a linear regression.

With this form of the hypothesis, the cost function is given by

$$J(\vec{\theta}) = \frac{1}{2m}(X\vec{\theta} - \vec{y})^T(X\vec{\theta} - \vec{y}) = \frac{1}{2m}||X\vec{\theta} - \vec{y}||^2$$

3.2 Gradient Descent Method

Ok, that's fine and all, but how do we find the optimal $\vec{\theta}$, to minimize the cost function $J(\vec{\theta})$? We seek some $\vec{\theta}$ which minimizes $J(\vec{\theta})$. In the present case, there is only one such $\vec{\theta}$, since it's a quadratic function of the θ_j's, and it's a (global) minimum.

The gradient descent method is an algorithm which seeks to make the gradient of the cost function vanish. That is, we have an algorithm which takes the form

[1] The term $x_0^{(i)} \equiv 1$ is often called the "bias term", but it's not agreed upon: some call the terms $b_0 \equiv \theta_0 x_0$ the "bias term", adding it after the dot products (not as aesthetic, but anyway...). Needless to say, the end results are the same, nonetheless.

$$\theta_j \leftarrow \theta_j - \alpha \nabla J(\vec{\theta}) = \vec{\theta} - \frac{\alpha}{m}(X\vec{\theta} - \vec{y})^T X_j$$
$$\iff \vec{\theta} \leftarrow \vec{\theta} - \frac{\alpha}{m} X^T (X\vec{\theta} - \vec{y})$$

for each step, where X_j is the j-th column of X. The number $\alpha > 0$ is the "learning rate", which determines how fast or slow the algorithm converges. Some care is due, since if α is too small, the steps are finer and the algorithm will take longer to converge; if α is too large, there may be some overshooting, and the final result will diverge. One needs to find some α between these two cases.

This method is also called "batch" gradient descent, since every step of the algorithm has a sum over the entire training set (represented by the terms of the gradient ∇J). Also, good to note, all θ_j's have to be updated simultaneously, for every step of the algorithm. Better safe than sorry.

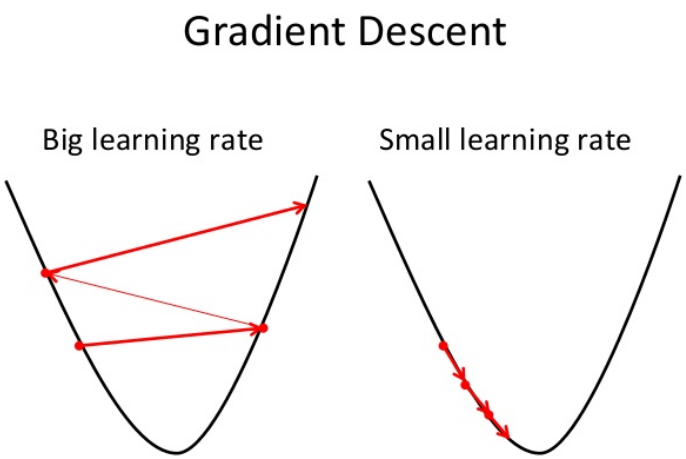

Figure 1: A graphical illustration of the gradient descent for different learning rates.

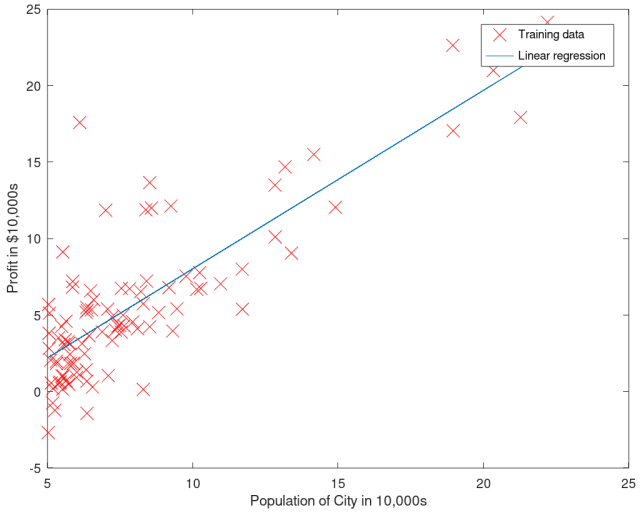

Figure 2: Linear regression for a dataset given in the first assignment of the Machine Learning course in Coursera (obtained via gradient descent).

3.3 Normal Equation

There is an analytical way of finding $\vec{\theta}$. Since we seek to minimize $J(\vec{\theta})$, then we can make $\nabla J = \vec{0}$. It'll be a minimum, since $J \propto \theta_j^2$ as $\vec{\theta}$ grows arbitrarily large; if there is a critical vector, it's bound to be a minimum.

Thus

$$\nabla J(\vec{\theta}) = \frac{1}{m} X^T (X\vec{\theta} - \vec{y}) = \vec{0}$$
$$\therefore \vec{\theta} = (X^T X)^{-1} X^T \vec{y}$$

This is the so-called "normal equation". However, one needs to take care about the inversibility of $X^T X$. In particular, by avoiding using the same characteristic with different units as distinct features (some feature measuring the area as ft^2 and another as m^2, which are redundant, for instance); there would be linearly dependent columns in X, and thus $\det(X^T X) = \det(X)^2 = 0$, so not invertible.

3.4 Polynomial Fits and Normalization of Features

We can also try a polynomial fit for our dataset. We do it the following way:

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x^i \mapsto \sum_{i=0}^{n} \theta_i x_i$$

that is, for each power x^i, we call it a new input x_i. This creates a new problem, because this greatly increases (or shrinks) the magnitude of the higher features, which may be problematic for the algorithm's operations.

So we do the following: for each feature $j \neq 0$ (or for each column in the matrix X, aside from the bias column of 1's), we make

$$x_j \mapsto \frac{x_j - \mu_j}{\sigma_j}$$

where μ_j is the average of the x_j's over the column j, and σ_j is the standard deviation of the x_j's over the column j; it makes sense to be over the columns, since each line is a dataset input, we are *averaging over our different training examples*. This way, the magnitude of each feature is lessened ("normalized") and won't interfere on the computations.

4 Classification Problem: Predicting Discrete Outputs

4.1 Logistic Regression

Now the problem is different: for each input $\vec{x}^{(i)}$, $y^{(i)} \in \mathbb{N}$, that is, the outputs are discrete.

In logistic regression, we assume the hypothesis to be

$$h_\theta(\vec{x}) = g(\vec{\theta} \cdot \vec{x})$$

where

$$g(z) = \frac{1}{1 + \exp(-z)}$$

is the sigmoid function.[2]

Note that $g(z) \in (0, 1)$, but it still has continuous values. What we do is the following: suppose we have a dataset which only takes the values $\{0, 1\}$. Then we predict

- $y = 1 \iff \vec{\theta} \cdot \vec{x} \geq 0 \iff round(g(\vec{\theta} \cdot \vec{x})) = 1 \iff g(\vec{\theta} \cdot \vec{x}) \geq 0.5$;

[2] It's called a logistic regression because the sigmoid function is a solution to the logistic differential equation $\frac{dN}{dt} = kN(1 - \frac{N}{L})$ (in our case, $k = L = 1$).

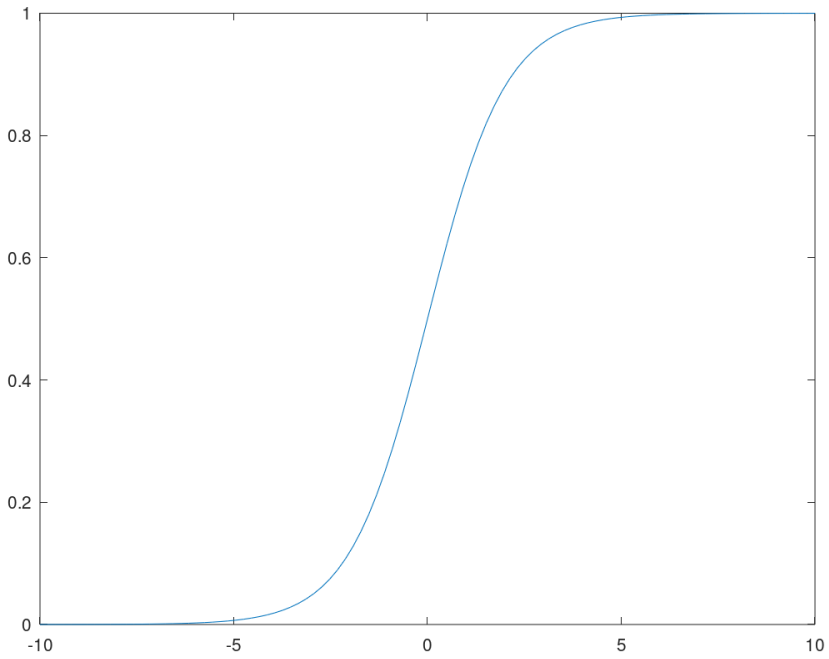

Figure 3: The graph of a sigmoid function $g(z) = \frac{1}{1+\exp(-z)}$.

- $y = 0 \iff \vec{\theta} \cdot \vec{x} < 0 \iff round(g(\vec{\theta} \cdot \vec{x})) = 0 \iff g(\vec{\theta} \cdot \vec{x}) < 0.5$.

It may seem too farfetched, but we base ourselves on the premise that the sigmoid function grows too fast to 1 or falls too quickly to 0. But in any case, now the hypothesis function (the sigmoid function) assigns some sort of probability that an input x assumes the value $y = 1$, i.e., $h_\theta(x) = P(y = 1|x_i; \theta)$.

4.2 Decision Boundary

When we have classification problems, it's usual to see clusters of data points with same values, as in figure 4. It's useful to plot a "decision boundary", that is, the geometrical space defined by $\vec{\theta} \cdot \vec{x} = 0$, "inbetween" the prediction of $y = 1$ and $y = 0$.

4.3 Cost Function for Logistic Regression

Note that, since $h_\theta(x)$ is essentially exponential, the usual cost function will most likely not be convex, i.e., probably won't have a single, minimum point.

Definition 4.1 (Cost Function, Logistic Regression)**.** *Define a new cost function as*

$$J(\vec{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log(h_\theta(\vec{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(\vec{x}^{(i)})) \right)$$

Remark. *For those who are more mathematically inclined, the function $J(\vec{\theta})$ is explicitly convex. By the way, this cost function is also called the "cross-entropy" cost function.*

This seemingly arbitrary choice of J may make more sense with figure 6. Actually, this cost function is obtained via the maximum likelihood estimation; a derivation of it can be found here.

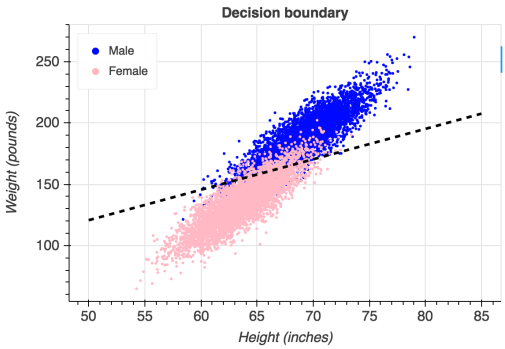

Figure 4: A classification problem: the outputs are either "male" or "female"; a (linear) decision boundary was plotted as well. (Note that a more quadratic hypothesis may yield a better decision boundary.)

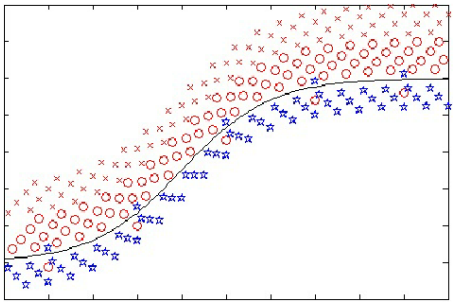

Figure 5: A classification problem with 3 possible outputs, with a partial decision boundary. Note that it "lumps" the red types of points together, as opposed to the blue star-shaped points.

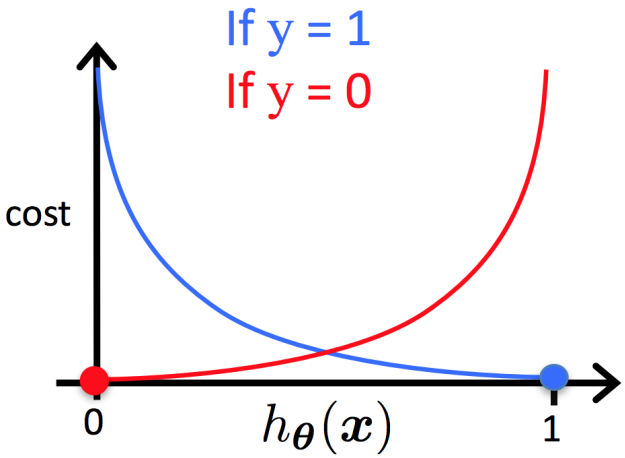

Figure 6: The possible graphs of the logistic regression cost function. Note that it diverges if $h_\theta(x) = 1$ and $y = 0$ and vice-versa; the idea is that it gets "costlier" to make these wrong predictions, pushing the hypothesis toward a better prediction.

What's really interesting is that the gradient of J is

$$\nabla J = \frac{1}{m} X(h_\theta(\vec{x}) - \vec{y})$$

where we make

$$h_\theta(\vec{x}) = \begin{pmatrix} h_\theta(x^{(1)}) \\ \vdots \\ h_\theta(x^{(m)}) \end{pmatrix}$$

The gradient of the logistic regression cost function has the same "shape" as the one from the linear regression cost function! The only difference is, of course, the form of the hypothesis function $h_\theta(x)$.

4.4 Multiclass Classfication: One vs. All

We can also have classification problems with more than 2 possible outcomes (also called classes). Then we use the "one vs. all" method[3], in which we obtain $h_\theta^{(i)}(x)$ for each class i, as represented in figure 7. So, if we have K classes, we divide our classification problem into K cost-minimization problems, obtaining hypotheses $h_\theta^{(i)}(x)$, for each class $i \in \{1, \ldots, K\}$, referring to the probabilities "input is in i-th class" (as opposed to "input is not in i-th class"). Then we take the index (say, j) which yields $h_\theta^{(j)}(x) = \max_{i \in \{1,\ldots,K\}} h_\theta^{(i)}(x)$, and predict that our input is (most likely) in the j-th class.

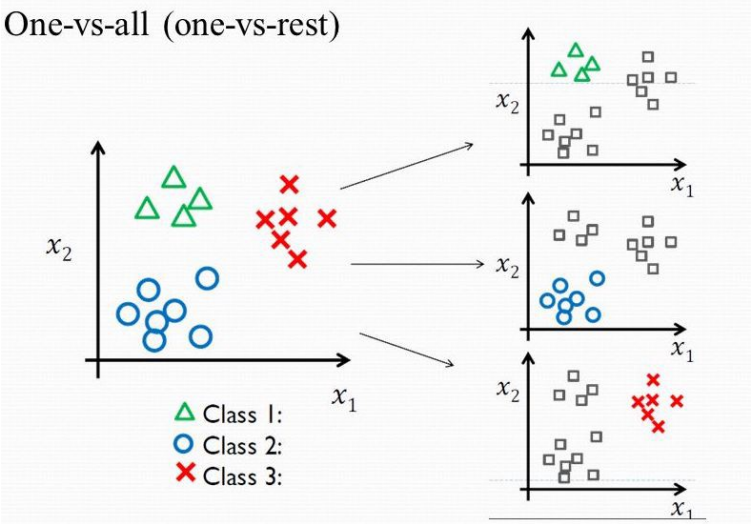

Figure 7: Example of the One vs. All method.

5 Neural Networks

5.1 Intuition/Motivation

Neural networks are a more biologically-inspired approach to programming (biomimics rejoice), attempting to take advantage of the idea of, as the name suggests, neural networks in our brains, as in figure 8: we have inputs, which are interpreted by so-called "hidden" layers, and then, finally, an output, which will predict future results. All of these layers are represented by units, also called neurons. The way these inbetween layers interpret the neurons is by analyzing a weighted sum of their values, and seeing whether it "activates/fires" or not, just like a neuron has an electrochemical threshold which determines whether it'll fire or not (which is where the bias comes in!); see figure 9.

A practical result of this is the possibility of tackling difficult problems by breaking it down in a larger number of simpler problems. The problem then becomes: what "architecture" do we use for this problem? In other words, what arrangement of layers will we use for our given problem?

There are many good references about neural networks, some of which are referenced in the bibliography ([3], [6]).

5.2 Forward propagation

Let there be L-many layers (including the input and output layers). We define $a^{(1)} \equiv x$ the first layer by the input (x naturally a vector; let m be its length).

[3]Should really be called "one vs. all *the rest*" method, but I reckon it doesn't have the same conciseness.

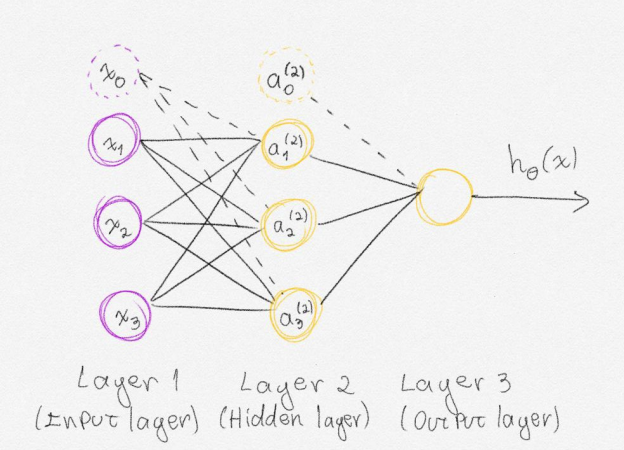

Figure 8: Depiction of layers in neural networks; the dashed circles are bias cells. Source: Andrew Ng, Coursera [1].

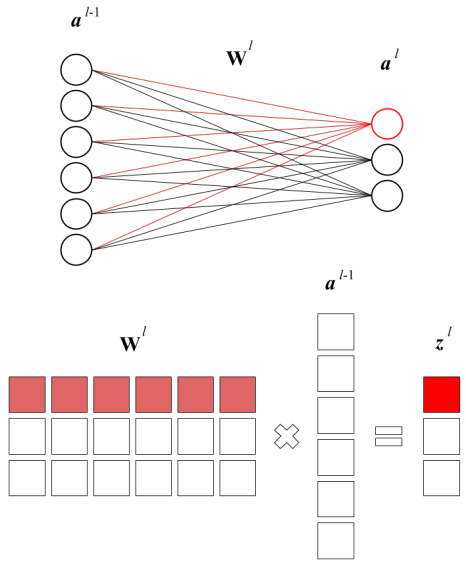

Figure 9: A depiction of the action of the weight matrices (here denoted $\mathbf{w^l}$) on a layer $l-1$ ($\mathbf{a}^l = g(\mathbf{z}^l)$, with g the activation function). Source: Erik Hallström [5].

Then, we advance to the next layer via a matrix $\Theta^{(1)}$, by

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

and pass it to the activation function g[4]

$$a^{(2)} = g(z^{(2)})$$

We do the same for all layers, up to the output layer

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$
$$a^{(l+1)} = g(z^{(l)})$$

[4]For simplicity, we'll assume $g^{(1)} = \cdots = g^{(L)} = g$, the sigmoid function.

In components[5]: $z_j^{(l+1)} = \sum_k \Theta_{jk}^{(l)} a_k^{(l)}$, all the while never forgetting to augment each $a^{(l)}$ with the bias terms, except for the L-th layer/output layer, which we denote

$$h_\Theta(x) \equiv a^{(L)}$$

Thus, the last layer yields the prediction for the neural network. This whole process is called "forward propagation".

Neural networks can naturally be used for multiclass classification problems; we determine the predicted output by the highest value of the output cells. We also denote each possible output with a unit vector.

The canonical example is by creating a neural network for identifying handwritten numbers between 0 and 9. We can, for instance, identify 0 with the unit vector

$$\hat{e}_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^{10}$$

1 with $\hat{e}_2$, etc.

5.3 Cost Function

We want to optimize this prediction with neural networks, so naturally we need a cost function.

Definition 5.1 (Cost Function, Neural Networks)**.** *Let a neural network with m training examples and K possible outputs (also called "classes"), in a multiclass classification problem. Then we define the cost function as*

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left(y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - h_\Theta(x^{(i)}))_k \right)$$

The definition seems like a lot to unpack, but it's essentially the logistic regression cost function, for each training example. Note that the $y^{(i)}$'s and the $h_\Theta(x^{(i)})$'s are K-dimensional; the $y^{(i)}$'s are the actual value of $x^{(i)}$'s, which are unit vectors depending on the label $k = 1, \ldots, K$ they represent, and $h_\Theta(x^{(i)})$'s are the value obtained by forward-propagating the training example $x^{(i)}$. Naturally, we have vectorized the log, i.e.,

$$\log(\begin{pmatrix} x \\ y \\ z \end{pmatrix}) = \begin{pmatrix} \log(x) \\ \log(y) \\ \log(z) \end{pmatrix}$$

We can also write the cost function as

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \cdot \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\Theta(x^{(i)})) \right)$$

where $\cdot$ denotes a dot product of vectors in $\mathbb{R}^K$. It becomes clearer that we are summing over the partial costs for forward propagating each training example $x^{(i)}$.

There is another, more aesthetically-pleasing (maybe?) way to write the cost function. First, we create a matrix for the outputs, in such a way that

$$Y = \begin{pmatrix} — & \vec{y}^{(1)} & — \\ & \vdots & \\ — & \vec{y}^{(m)} & — \end{pmatrix} \in \mathbb{R}^{m \times K}$$

[5]It may seem that the index jk is flipped, but it's to prevent multiplications by $\Theta^{(l)}$'s transpose further down the line. A more mathematically inclined justification would be that k gets contracted.

(m number of training examples, K classes).

Noting that $h_\Theta(x^{(i)})$ is the i-th line of $a^{(L)} \in \mathbb{R}^{m\times K}$, we can then write

$$J(\Theta) = -\frac{1}{m}\left(Tr\left(Y\log(a^{(L)})^T\right) + Tr\left((1-Y)\log(1-a^{(L)})^T\right)\right)$$

where Tr denotes the trace of a matrix, and $1-Y$ is the matrix with 1's on all indices, subtracted by Y.

5.4 Backpropagation

We seek to find the optimal $\Theta_j^{(l)}k$ for our neural network. We can apply gradient descent, and so the problem becomes: how do we find $\frac{\partial J}{\partial \Theta_{jk}^{(l)}}$? The canonical approach to it is by backpropagation.

Let there be a neural network with m inputs, K classes/labels, and L layers. Our plan of action is to analyze the quantities

$$\delta_j^{(l)} \equiv \frac{\partial J}{\partial z_j^{(l)}}$$

which tell how much the cost deviates as the signals (before activation) in each cell, $z_j^{(l)}$, are changed slightly.

For our demonstrations, we require that

1. $J = \frac{1}{n}\sum_x J_x$, that is, the cost function is an average of the individual costs for each input x;
2. $J = J(a^{(L)})$, that is, the cost function is a function of the output cells (last layer L).

Definition 5.2 (Hadamard product)**.**
Let $x, y \in \mathbb{R}^k$, with components x_i, y_i. Then

$$x \odot y = \begin{pmatrix} x_1y_1 \\ \vdots \\ x_ky_k \end{pmatrix}$$

is the vector of pointwise multiplication between x and y. It's also called the Hadamard product, or Schur product. It's trivial to see that it is associative and commutative.

For the sake of notation, we'll denote the cost for a single input x as J, instead of J_x. Just keep in mind that the following J's are not the total cost.

Proposition 5.1. *In the last layer L, we have*

$$\delta^{(L)} = \nabla_a J \odot g'(z^{(L)})$$

where $\nabla_a J$ is the gradient of J with respect to $a_i^{(L)} = h_\Theta(x^{(i)})$, and g' is the derivative of g.[6]

Proof.

$$\begin{aligned}\delta_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}} &= \sum_k \frac{\partial J}{\partial a_k^L}\frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} \\ &= \sum_k \frac{\partial J}{\partial a_k^L}\, g'(z^{(L)})\,\delta_{kj} \\ &= \frac{\partial J}{\partial a_j^L}\, g'(z_j^{(L)})\end{aligned}$$

[6] Remembering that the derivative of the sigmoid is the logistic equation, and $a^{(L)} = g(z^{(L)})$, we may write $\delta^{(L)} = \nabla_a J \odot \left(a_i^{(L)} \odot (1 - a_i^{(L)})\right)$

where we've used $J = J(a^{(L)})$, $a^{(L)} = g(z^{(L)})$ and $\frac{\partial z_k^{(L)}}{\partial z_j^{(L)}} = \delta_{jk}$ the Kronecker delta (yes, it is notationally abusive). A more compact way to write this is

$$\delta^{(L)} = \nabla_a J \odot g'(z^{(L)})$$

□

Remark. Do note that, for our cost function J for neural networks, we have that

$$\frac{\partial J}{\partial a_i^{(L)}} = \frac{(a_i^{(L)} - y^{(i)})}{g'(z_i^{(L)})} \implies \delta_i^{(L)} = a_i^{(L)} - y^{(i)}$$

Proposition 5.2. *For each layer* $2, \dots, L-1$,

$$\delta^{(l)} = (\Theta^{(l+1)})^T \delta^{l+1} \odot g'(z^{(l)})$$

Remark. Some caution is needed here: we've been using $\Theta^{(l)}$ as the matrix advancing the l-th layer toward the $(l+1)$-th layer. Some other texts may write $\Theta^{(l)}$ as the matrix advancing *toward* the l-th layer *from* the $(l-1)$-th layer.

Proof.

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \sum_k \frac{\partial J}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

Remember that[7]

$$z_k^{(l+1)} = \sum_j \Theta_{kj}^{(l)} a_j^{(l)} = \sum_j \Theta_{kj}^{(l)} g(z_j^{(l)}) \implies \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \Theta_{kj}^{(l)} g'(z_j^{(l)})$$

$$\therefore \delta_j^{(l)} = \left[\sum_k \Theta_{kj}^{(l+1)} \delta_k^{(l+1)} \right] g'(z_j^{(l)}) \iff \delta^{(l)} = (\Theta^{(l+1)})^T \delta^{(l)} \odot g'(z^{(l)})$$ □

One can really see the idea of *back*propagation in play here: we find $\delta^{(L)}$ first, and then go backwards in the network.

Proposition 5.3. $\frac{\partial J}{\partial \Theta_{jk}^{(l)}} = a_k^{(l)} \delta_j^{(l+1)}$

Proof.

$$\frac{\partial J}{\partial \Theta_{jk}^{(l)}} = \sum_p \frac{\partial J}{\partial z_p^{(l+1)}} \frac{\partial z_p^{(l+1)}}{\partial \Theta_{jk}^{(l)}}$$

Remembering that $z_p^{(l+1)} = \sum_k \Theta_{pk}^{(l)} a_k^{(l)}$, we have $\frac{\partial z_p^{(l+1)}}{\partial \Theta_{jk}^{(l)}} = \Theta_{pk}^{(l)} a_k^{(l)} \delta_{pj}$ (δ_{pj} the Kronecker delta).

$$\therefore \frac{\partial J}{\partial \Theta_{jk}^{(l)}} = \delta_j^{(l+1)} a_k^{(l)}$$

□

The intuition behind this is the following: the way the cost changes according to $\Theta_{jk}^{(l)}$ depends on the activation of the neuron you've left from, $a_k^{(l)}$ and the error on the neuron of arrival, $\delta_j^{(l+1)}$ associated to $a_j^{(l+1)}$.

In some texts, one finds the following equations:

[7] In our discussion, we've incorporated the bias weights inside the Θ matrices; other texts may deal with the biases $b_j^{(l)} = \Theta_{j0}^{(l)} a_0^{(l)} = \Theta_{j0}^{(l)}$ separately.

Corollary 5.1.
$$\begin{cases} \frac{\partial J}{\partial \Theta_{jk}^{(l)}} = \delta_j^{(l+1)} a_k^{(l)} \\ \frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l+1)} \end{cases}$$

Proof. We need only note that $b_j^{(l)} = \Theta_{j0}^{(l)} a_0^{(l)} = \Theta_{j0}^{(l)}$, since $a_0^{(l)} = 1$ per definition. □

Also, do note that this is all independent of the activation function g! All we required was that, aside from differentiability, the cost depended on the outputs and that it be "decomposable" as the average of the individual costs (which we've used implicitly throughout the entire demonstrations).

A more in-depth discussion on backpropagation can be found in [4]. The derivations above can also be found there, with more attention to details.

6 Bias and Variance

We need some quantitative way to tell whether our prediction models are fine or inacceptably off. We call attention to two extreme cases: under- and overfitting. Take a look at figure 10. For starters, suppose we have a training set (in which we adjust our hypothesis $h_\theta(x)$) and a test set, with which we judge our hypothesis afterwards.

Figure 10: Example of under- and overfitting.

Underfitting happened when our hypothesis was too simplistic for our data. Assume, for instance, we have a linear hypothesis which doesn't really fit our dataset, which is kind of curved in its graph. We picked the simplest model (explicitly dependent on our data), which demanded the lowest number of parameters. It would've paid off only if the dataset's complexity was compatible with it. It almost seemed as if we predisposed to believe, before even looking at the dataset, that it was a good idea. This is a case in which we have "high bias": our model is too "rigid", it isn't really up to par with the dataset's demanded complexity.

In contrast, overfitting happened when we essentially interpolated the entire training set with an appropriately sized polynomial. The result is that it fit our dataset perfectly (per definition, basically), but it had no business with adjusting our test data, afterwards. This is a case of "high variance": our model performs poorly when we look at different datasets with it.[8]

So we can see that there is some correlation between the complexity of our hypothesis and bias/variance. See figure 11. Let's only focus on the bias (red) and variance (blue) curves. Notice that, on the left of the graph, the bias is high and the variance is low: there's underfitting, the model is much simpler than our dataset demands, and so the bias is high; notice that the variance is low, and that is because the model is just not good enough, "not even trying" to adjust the data; it doesn't really see a difference between the training set and the test set.

[8] I almost feel kind of inclined to say that "high variance" examples feature some kind of "dataset bias" (in the common sense of the word "bias"), while "high bias" examples display some "model/person-behind-the-screen's bias". Yeah, explanations don't get much better than that, when it comes to this topic...

On the right, we have high variance but low bias: there's overfitting, our model is so complex that it fits the training data perfectly, but it fails to be of use in our test set: it's not flexible when it comes to a dataset which isn't its original training set.

And somewhere in between, there's a sweet spot which is the "ideal" model to be used with this dataset: it's the one which minimizes the total error, which is the sum of the bias (squared), the variance and the inherent error associated with the data (also called "irreducible error"). In it we have low bias (the model isn't more complex than it needs to be) and low variance (it adjusts well to different datasets).

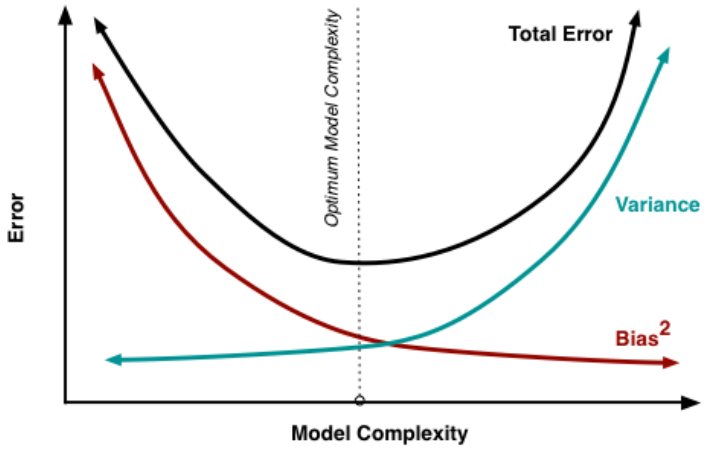

Figure 11: A graph of bias and variance over the complexity of our hypothesis.

A similar graph appears when we analyze the training/test errors with respect to the regularization constant λ; see figure 12. Remember that, if we initially had a model which overfit our data, then with different regularization constants λ, some results would occur: $\lambda \to 0$ would not do much, and the model would still be overfitting; and $\lambda \to \infty$ would cause $\theta_j \to 0\,(j \neq 0)$, which would yield a constant model, which would obviously underfit our data. The same analysis as above holds, but it's in reverse: on the left side of the graph, our model fits the training data well (training error is small) but doesn't adjust the test set well-enough; on the right side, it's not adjusting either the training nor the test sets.

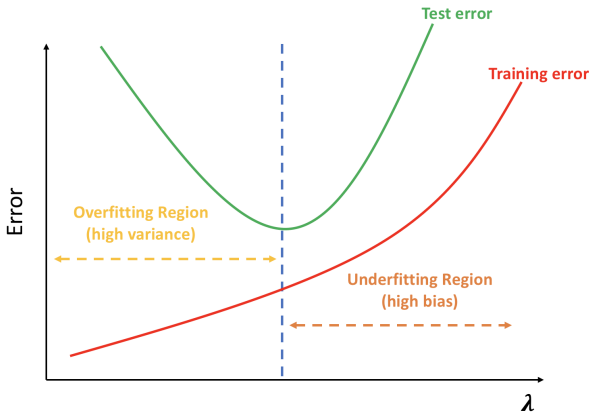

Figure 12: A graph of the training and test errors with respect to the regularization constant.

Sometimes (most times) one may think: more data points might do the trick. Well, that's not really the case: see figures 13 and 14. They are called learning curves: they're plotting errors by the number of training examples.

Note that, if we have a high bias problem, our model doesn't really have any business to do with our training set, and so having a larger one would most likely increase the training error; it does decrease the test error, since our model is learning with more and more examples, so there is some bang for your buck; the downside is that the errors might be too large to make any meaningful predictions.

If we have a high variance problem, increasing the size of the training set might actually show that our at first "too complex" model might actually do the trick just fine. It may not be perfectly fitting the training set as before, but it may start fitting the test set better and better.

The main takeaway is: increasing the size of the training set is not a one-size-fits-all; it is highly case-dependent (in particular, for high variance problems).

Figure 13: A learning curve with high bias: note that increasing N doesn't really solve the fact that it's not well-adjusting the dataset. Source: Andrew Ng, Coursera [1].

Figure 14: A learning curve with high variance: note that increasing N may help the errors to converge to some desired quantity. Source: Andrew Ng, Coursera [1].

7 Cross-Validation

When training different machine learning models, we'd like to pick the best ones for our purposes. It's common practice to divide our entire dataset into *training*, *cross-validation* and *test* sets. So a summary of the whole process of choosing a best model would be:

1. For each model, train it in the training set;
2. For each trained model, evaluate its efficacy on the cross-validation set (based on a given evaluation metric);
3. Choose the best model out of the previous ones (based on a given evaluation metric);
4. Test the best model in the test set.

It's just like when we try out different things before our presentations, and then only present the best thing we've got: the cross-validation does what its name suggests, it "cross-evaluates" models, and then picks out the fittest to our needs, which will be subject to the final, challenging, fear-inducing test set.

When machine learning began to become famous, a common rule of thumb was a $60\% - 20\% - 20\%$ divide of the entire dataset into training-(cross-val)-test sets. But with the advent of Big Data and Deep Learning, our dataset can be much bigger than before, and so we can actually spend a gigantic amount of data such as to minimize overfitting of our models. A partition of $98\% - 1\% - 1\%$ may not be as petty as it seems, as 1% of a billion datapoints is still in the order of millions of datapoints.

8 Support Vector Machines

There can be a lot of ambiguity on which decision boundary we choose for our classification datasets, as can be seen in figure 15. So how can we measure which ones are good and which aren't? Is there an optimal one?

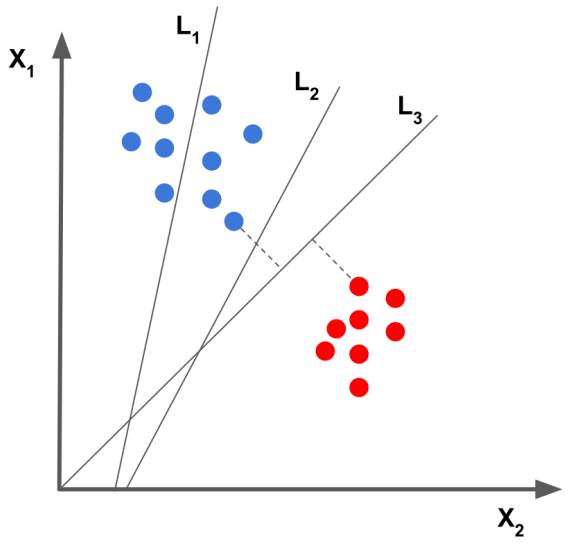

Figure 15: Some decision boundaries for a given classification problem.

Turns out that there is, if our dataset is linearly separable (via some transformation on the feature space), as in the previous image. For now, let's assume we have a linearly separable dataset $\{x^{(i)}, y^{(i)}\}_{i=1}^{m}, x^{(i)} \in \mathbb{R}^n$, but we'll assume something different from previous sections: $y^{(i)} \in \{-1, 1\}$; this will help with describing upcoming mathematical inequalities.

We'll let out weight parameters be given by $w \in \mathbb{R}^n$ instead of θ, following SVM convention. Thus, our predictions will be

$$h_w(x) = g(w^T x + b) = \begin{cases} 1, & \text{if } w^T x \geq 0 \\ -1, & \text{if } w^T x < 0 \end{cases}$$

with g the sigmoid function, for our purposes, and b the bias term[9].

8.1 The original problem

The following derivations follow [9], [11] and [12].

Definition 8.1 (Functional Margin)**.**
Let there be a dataset example $(x^{(i)}, y^{(i)})$; define the *functional margin* as

$$\hat{\gamma}^{(i)} \equiv y^{(i)}(w^T x^{(i)} + b)$$

If $\hat{\gamma}^{(i)} > 0$, then the prediction is in accord with the actual $y^{(i)} = \pm 1$ value; if negative, its opposite to the actual value. Its absolute value $|\hat{\gamma}^{(i)}|$ shows how confident we can be on the prediction.

We then define the functional margin of the dataset as

$$\hat{\gamma} \equiv \min_i \hat{\gamma}^{(i)}$$

Remark. *Note that, while our prediction with g only depends on the sign of $(w^T x^{(i)} + b)$ (and not on the magnitude of it), we have that the functional margins depend explicitly on both the magnitude and sign of $(w^T x^{(i)} + b)$.*

Definition 8.2 (Geometric Margin)**.**
Let there be a dataset example $(x^{(i)}, y^{(i)})$; define the *geometric margin* as

$$\gamma^{(i)} \equiv y^{(i)} \frac{(w^T x^{(i)} + b)}{||w||} = \frac{\hat{\gamma}^{(i)}}{||w||}$$

[9] We choose to separate $b\,(\equiv w_0)$ from w since we're seeking a "real" n-dimensional vector, which will be the normal vector to the hyperplane separating the classes.

We then define the geometrical margin of the dataset as

$$\gamma \equiv \min_i \gamma^{(i)}$$

Remark. *There is a (no pun intended) geometrical interpretation to this definition: let $x^{(i)}$ be a vector in the feature space, and $\gamma^{(i)} \frac{w}{||w||}$ a normal vector from a point in the decision boundary to $x^{(i)}$ (for now, $\gamma^{(i)} \in \mathbb{R}$ is unknown). Then, by addition of vectors, we have that the vector from the origin to that point in the decision boundary is given by*

$$x^{(i)} - \gamma^{(i)} \frac{w}{||w||}$$

which satisfies, by definition of decision boundary

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0$$

which yields our definition of geometrical margin. Note that our definition of $\gamma^{(i)}$ depends on the sign of $(w^T x^{(i)} + b)$ to make a prediction, and independs on a change of scale $(w, b) \mapsto (\alpha w, \alpha b)$.

So now our problem becomes maximizing our geometrical margin γ with respect w, b, γ with the constraints

$$\begin{cases} y^{(i)}(w^T x^{(i)} + b) \geq \gamma, & i = 1, \cdots, m \text{ (min. distance from decision boundary)} \\ ||w|| = 1, & \text{(implies Geo. Margin = Func. Margin)} \end{cases}$$

We can change the problem to make it more optimizable:

$$\max_{\hat{\gamma}, w, b} \frac{\hat{\gamma}}{||w||} \;\; s.t. \;\; y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, i = 1, \cdots, m$$

We abdicate the condition of $||w|| = 1$ since it's non-convex and may be of trouble, but the problem is still the same: maximize the (geometrical) margin.

We introduce the constraint that $\hat{\gamma} = 1$; thus, we have the problem

$$\min_{w, b} \frac{1}{2} ||w||^2 \;\; s.t. \;\; y^{(i)}(w^T x^{(i)} + b) \geq 1, i = 1, \cdots, m$$

Solving this will give an optimal margin classifier for our problem.

8.2 Solving for the optimal margin

Our problem as stated above can be solved via Lagrange Multipliers. Let our lagrangian be

$$L(w, b, \alpha) = \frac{1}{2} ||w||^2 - \sum_{i=1}^{m} \alpha_i \left[(y^{(i)}(w^T x^{(i)} + b) - 1 \right]$$

We set the derivatives of L with respect to w and b to 0:

$$\begin{cases} \nabla_w L(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0 \implies w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} (\in \mathbb{R}^n) \\ \frac{\partial L}{\partial b}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \end{cases}$$

Plugging these results back in the lagrangia n, we obtain

$$\tilde{L}(\alpha) = \sum_{i=1}^{m} - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)} \cdot x^{(j)}) \;\; s.t. \;\; \begin{cases} \alpha_i \geq 0, i = 1, \ldots, m \\ \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \end{cases}$$

(The condition $\alpha_i \geq 0$ comes from the Kuhn-Tucker conditions; details in [9].) It's conventionally said that L is the primal form of our lagrangian, while $\tilde{L}$ is its dual form.

It turns out that the only non-zero α's belong to the so-called *support vectors*, that is, the closest points to the decision boundary, which serve as "pivots" to the margins.

When we have non-linear decision boundaries, we can find some transformation φ from the feature space onto itself such as to make the decision boundary linear. Then we'd have to calculate $\varphi(x^{(i)}) \cdot \varphi(x^{(j)})$, which can be very expensive. A more inexpensive way, it turns out, is to have a kernel $K(x^{(i)}, x^{(j)}) = \varphi(x^{(i)}) \cdot \varphi(x^{(j)})$. One of the most used kernels is the radial basis function (gaussian), $K(x, y) = \exp(-\frac{||x-y||^2}{2\sigma^2})$.

9 Unsupervised Learning

"Unsupervised" learning takes on datasets without any y-labels, that is, undistinguishable points (apart from their features). It consists mostly of "clumping" data points together in different *clusters* (based on their position in feature space $\mathbb{R}^n$) or trying to reduce their feature space by focusing only on their "preferred" direction (in the feature space $\mathbb{R}^n$). It can be useful in its own right, for instance, in analyzing market tendencies with data, or voting preferences based on factors such as voters' wealth, religiousness etc.

In unsupervised learning, we will only have training sets without a respective label vector, i.e., there will only be the training matrix $X \in \mathbb{R}^{m \times n}$ as usual.

10 K-Means Clustering Algorithm

There are some cases in which we can instantly see "distribution patterns" in scatter plots (as in figure 16) . In others, it may be more demanding on the mind's eye. But it feels so subjective... can we really create an algorithm for it?

Of course, it's also relevant how many *clusters* (the commonly used name) we are willing to search for. There are two extremes: we can say they're all in one cluster —"they're all undistinguishable data points, and that's it!"— which isn't very creative (and would be some sort of "underfitting"); or we can say all points are in a single-point cluster, that is, themselves, which doesn't really tell anything interesting (which would be some sort of "overfitting": you're looking too much into it!).

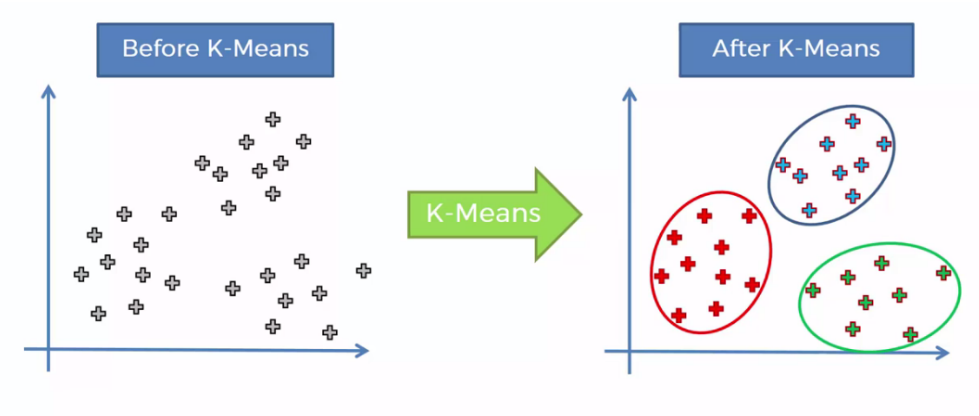

Figure 16: One can easily see 3 clusters of points!

So how can we go about it? One way is by using the K-Means algorithm, which proposes:

1. Let $X \in \mathbb{R}^{m \times n}$ be our usual training set;
2. choose how many clusters will be searched for (let it be denoted $K \leq m$);
3. randomly choose K different points (rows) in X to be the initial "cluster centroids" $\{\mu_k\}_{k=1}^{K}$;

4. assign to each training point $x^{(i)}$ an index $1 \leq c^{(i)} \leq K$ which denotes its closest cluster centroid $\mu_{c^{(i)}}$ (by Euclidean distance, usually);

5. update the cluster centroids by making $\mu_k \mapsto \frac{1}{m_k} \sum_{\alpha=1}^{m_k} x^{(\alpha)}$ (i.e. the average of the m_k points in each cluster k);

6. go back to item 4 until the centroids don't change anymore, or until a maximum number of iterations.

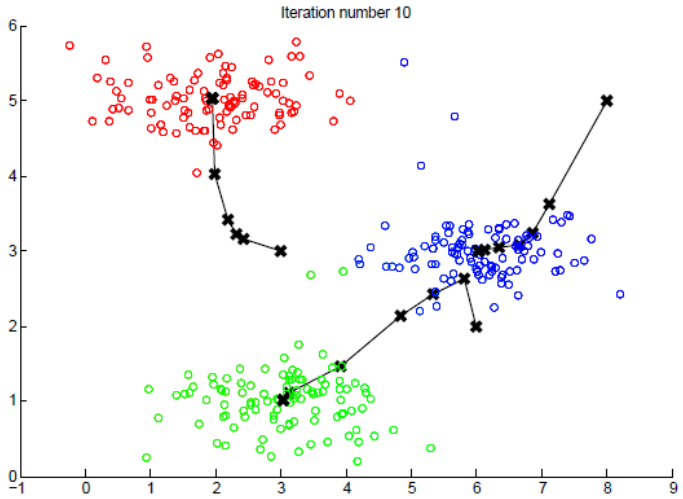

Figure 17: Example of K-Means clustering applied to Assignment 7 on the Machine Learning course on Coursera: initial points were randomly chosen from the training set; their trajectory is shown by the black lines.

It's a pretty interesting and powerful method, as can be seen in figure 17, but there are some problems. For instance, it depends entirely on the randomly selected initial centroids, so it may give pretty different results (it may happen when K is small). What one can do is define a cost function for it as

$$J(c^{(i)}, \mu_k) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

It turns out that minimizing it equals to steps 4 and 5 above[10]. What we can do, when confronted with different clustering results, is take the one which yields the smallest cost function J; it guarantees a good result (as far as the algorithm is concerned), since we're just minimizing the average quadratic distance between points and their centroids.

11 Principal Component Analysis; Dimensionality Reduction

One thing we can do with our training set is to decompose it into its most important "directions"; that is, if $x^{(i)} \in \mathbb{R}^n$, we project it onto a hyperplane in $\mathbb{R}^K, K \leq n$. The figure 18 show a projection of points in $\mathbb{R}^2$ onto a line defined by the dataset's *principal component* (its main "direction/tendency"). This can be helpful for visualization, by reducing the feature space from $n > 3$ to $K \leq 3$ dimensions, and may also speed up the code compilation.

So to formalize the idea of principal components, we need to talk about the covariance matrix. Let $X \in \mathbb{R}^{m \times n}$ be our training set. First of all, we make

$$Col_i \to Col_i - \mu_i$$

[10] Minimizing with respect to $c^{(i)}$ equals to step 4, and minimizing with respect to μ_k equals to step 5.

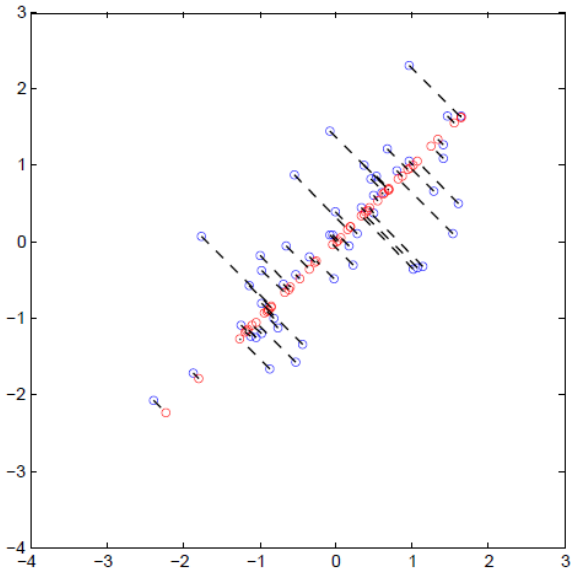

Figure 18: An example of dimensionality reduction via PCA: the blue original data points are projected onto a straight line, represented by the red points.

that is, each column gets subtracted (pointwise) by the column's average value over the training examples ($\mu_i = \frac{1}{m}\sum_{k=1}^{m} X_{ki}$). So we're, in a way, taking the averages of each of the dataset's features.

Define the covariance matrix as

$$\Sigma \equiv \frac{1}{m} X^T X \in \mathbb{R}^{n \times n}$$

Note that, by construction, $\Sigma^T = \Sigma$. Each of its components is

$$\Sigma_{ij} = \sum_{k=1}^{n} \frac{1}{m} X_{ki} X_{kj} = \frac{1}{m} Col_i \cdot Col_j$$

As figure 19 shows, the covariance matrix (as the name suggests) captures the correlation between features: positive/negative off-diagonal elements Σ_{ij} show the linear relation between features i and j, while diagonal elements Σ_{ii} show the i-th element's spread over its corresponding axis (that is, their variance).

So what we do now is analyze Σ's eigenvectors, which will constitute the dataset's principal components. Let $U \in \mathbb{R}^{n \times n}$ be the matrix whose columns are eigenvectors of Σ, with decreasing eigenvalues (from left to right).

Let $K \leq n$, Then we can take $U_{reduce} \in \mathbb{R}^{n \times K}$ the first K columns of U, whose columns define a hyperplane of dimension K on $\mathbb{R}^n$. Then we project each training vector onto this hyperplane:

$$Z = X\, U_{reduce} \in \mathbb{R}^{m \times K}$$

Each component of Z is given by (denoting $U_{reduce} \equiv \tilde{U}$ just for a moment)

$$Z_j^{(i)} = \sum_{k=1}^{n} X_k^{(i)} \tilde{U}_{jk} = X^{(i)} \cdot U^{(j)}$$

that is, each component of the row vector $Z^{(i)}$ is given by the projection of $X^{(i)}$ onto the eigenvector $U^{(j)}$.

Thus, we have decomposed our dataset onto their "strongest" directions, called *principal components*.

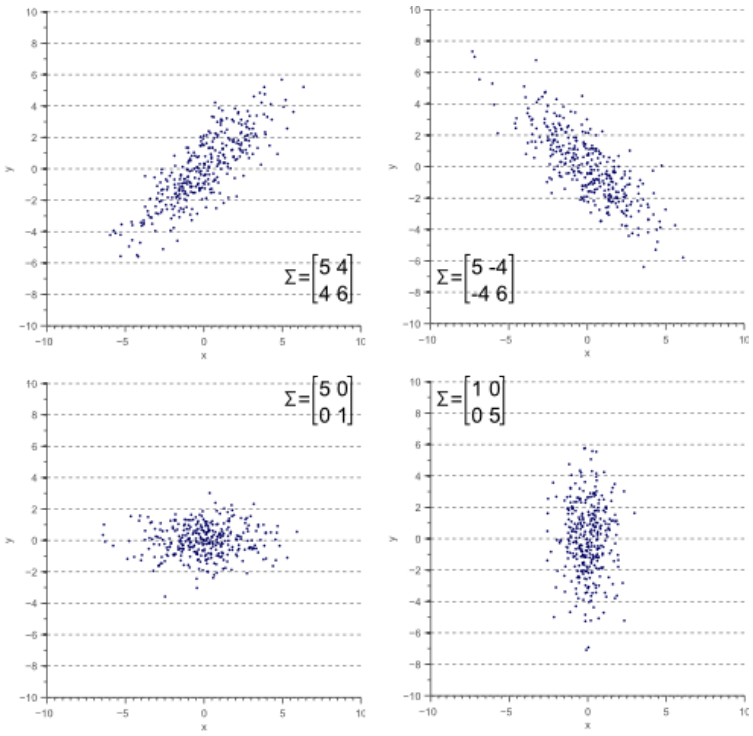

Figure 19: Examples of datasets and their respective covariance matrices.

12 Anomaly Detection: Probability Distributions

Suppose we have a dataset whose examples are either normal ($y = 0$) or anomalous ($y = 1$); suppose also that anomalous examples are often distinguished by having some features disproportionately bigger/smaller than other examples. Think about, for instance, the performance of computers in a data center, where an anomalous example could be a computer with too much memory use and too little users. How do we predict a new example will be anomalous or not?

A reasonable thought would be "logistic regression", since this is a classification problem. But there is another way to do this. We can create a probability distribution that will predict whether or not a new example will be anomalous or not.

So let $\{X^{(i)}, y^{(i)}\}$ be our dataset, with as many normal examples as possible. We then separate it into a training set with (if possible) only normal examples, a cross-validation set and a test set, both with normal and anomalous examples (naturally, many more normal than anomalous).

One thing we can do is make[11]

$$p(x) = \prod_{j=1}^{n} p(x_j : \mu_j, \sigma_j^2)$$

where μ_j, σ_j^2 are the average and variance of feature j over training examples. Usually we use the gaussian distribution, and so we analyze anomalous examples based on how far they are with respect to the center of the distribution, whose components are μ_j.

Another way to create the (gaussian) probability distribution from our training set would be by making

$$p(x : \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$

where Σ is the covariance matrix for X, $\mu = \begin{pmatrix} \mu_1 & \dots & \mu_n \end{pmatrix}^T$ the "center" of the training set, and $|\Sigma|$ the determinant of Σ. This yields a more concrete prediction for anomalies, since it's analyzing directly over

[11] A mathematical assumption would be that the features are *independent* (in the probabilistic sense), but computer scientists don't pay much mind to it...

the correlations between features. However, it may be much more computationally expensive, especially for more n features.[12]

13 Alternative Methods in Machine Learning

We may have a gigantic number of examples in our dataset, in the order of millions or even billions of points. In those cases, (batch) gradient descent (as we've used previously) can be a very slow and expensive procedure, since it'd be doing a sum over a million/billion points, *for each iteration*! Luckily, there are some alternatives.

13.1 Stochastic Gradient Descent

Suppose we have a (training) dataset $X \in \mathbb{R}^{m\times(n+1)}$ and their output $\vec{y}$, as usual. We then define the individual cost for each dataset point $x^{(i)}$ as

$$J_i \equiv \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Then, the cost of the dataset is the average over each example,

$$J_{train} = \frac{1}{m}\sum_{i=1}^{m} J_i$$

The *stochastic gradient descent* follows the algorithm:

1. Randomly shuffle the (examples in the) dataset;
2. Repeat until convergence:[13]

   for i=1:m{

   $$\begin{aligned}\vec{\theta} &:= \vec{\theta} - \alpha\nabla J_i \\ &:= \vec{\theta} - \alpha X^{(i)}\cdot(h_\theta(x^{(i)}) - \vec{y})\end{aligned}$$

   } (where $X^{(i)}$ is the i-th row).

That is, instead of analyzing the entire dataset for a single iteration of the gradient descent, we iterate for each example (we randomize it to make it less "biased"). The net result is that, instead of a "resolute" step toward the direction of steepest descent, we see some sort of "drunken walk", although its general direction is the same as the former (see figure 20).

13.2 Mini-batch Gradient Descent

Suppose that we have some dataset $X \in \mathbb{R}^{m\times(n+1)}$ and its output $\vec{y}$ as usual.

The *mini-batch gradient descent* is exactly what it seems: a (batch) gradient descent, but instead of using the entire dataset, we partition it, and do (batch) gradient descent in each partition:

1. Choose $b \ll m$ to be the size of the mini-batches;
2. Repeat until convergence:[14]

   for i = 1:b:m{

   $$\vec{\theta} := \vec{\theta} - \frac{\alpha}{b}X(i:i+b-1,:)^T(h_\theta(x^{(i)}) - \vec{y})(i:i+b-1)$$

   }

[12] Note that, if $\Sigma = \mathrm{diag}(\sigma_1^2,\ldots,\sigma_n^2)$, we have that both probability distributions above are equal.

[13] It's just pseudocode, from the first to the m-th row of X.

[14] Where i=1:b:m means "from 1 to m in b-sized steps", and v(i:i+j) means "from i-th to (i+j)-th element".

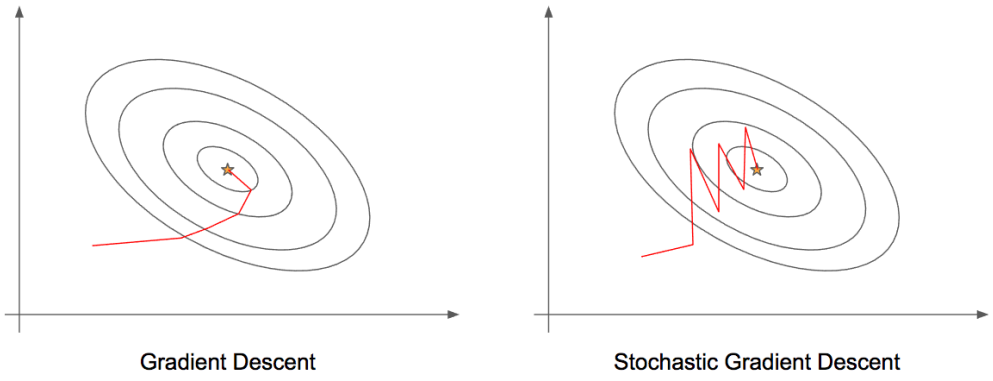

Figure 20: A graphical way of comparing (batch) versus stochastic gradient descent.

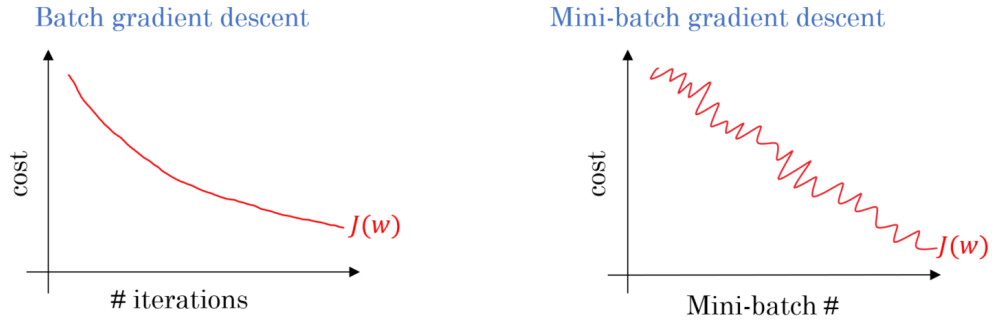

Figure 21: Graphical comparison between batch and mini-batch gradient descent.

13.3 Mathematical Interlude: Exponentially Weighted [Moving] Averages

Given a set of points $\{\theta_i\}$, we wish to compute the *exponentially weighted (moving) averages* as follows:

$$\begin{cases} S_0 & \equiv const \\ S_t & \equiv \alpha S_{t-1} + (1-\alpha)\theta_t \, (t > 0) \end{cases}$$

with $\alpha \in [0, 1]$.

Note that, given an iteration t we can recursively write

$$S_t = (1-\alpha) \sum_{k=1}^{t} \alpha^k \, \theta_{(t-k)} + \alpha^t S_0$$

which shows explicitly a form of averaging over the dataset $\{\theta_i\}$. Usually $S_0 \equiv 0$[15], and so one can also add a correction of the form

$$v_t \to \frac{v_t}{1-\alpha^t}$$

as in [15], such as to correct the earlier iterations of S_t.

We can think of it as averaging mostly over

$$\alpha^d = \frac{1}{e} \iff d = -\log_\alpha(e)$$

iterations. For instance, if $\alpha = 0.9$, we have an average over mostly $d = 10$ iterations.

[15] This is because S_0 acts as a sort of "phantom data" on earlier t's, since there'd be averaging "over some $t' < 0$ iterations"; e.g. for a dataset $\{\theta_i\}_{i=1}^K$, S_1, would be "averaging" over the last 3 iterations, then it'd be "taking into account" $t' = 0, -1$. So setting it to 0 reduces the bias that a non-zero value would induce into the averages $\{S_t\}$.

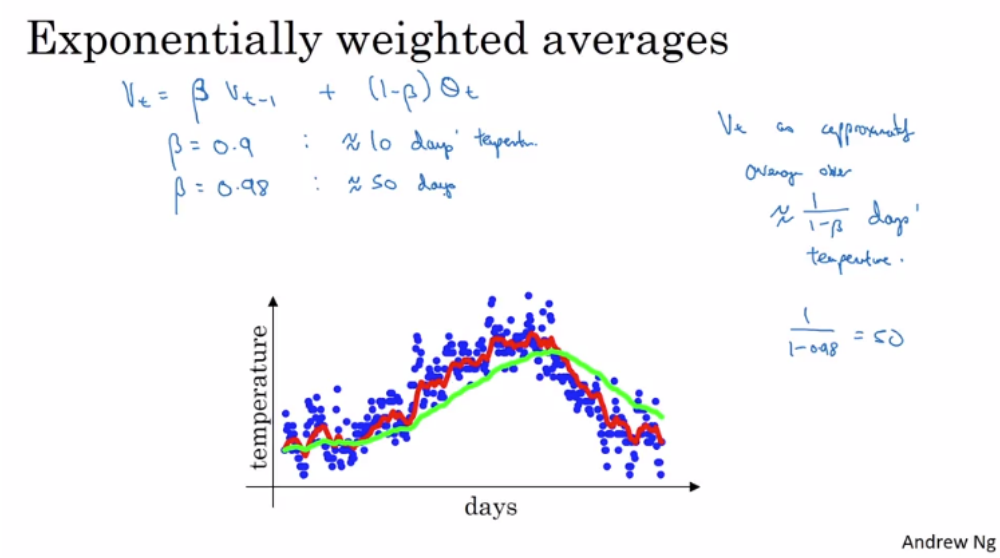

Figure 22: Graph showing e.w.a. over a dataset: red line shows $\beta = 0.9$, green shows $\beta = 0.98$ [14].

13.4 Adam Algorithm [16]

The most ubiquitous optimization algorithm used in Machine Learning nowadays is the Adam (acronym for "adaptive moment estimation") algorithm. Denoting $d\theta \equiv \frac{\partial J}{\partial \theta}$, the algorithm computes

$$\begin{cases} \begin{cases} V_{d\theta} = \beta_1 V_{d\theta} + (1-\beta_1) d\theta \\ V_{d\theta} := \frac{V_{d\theta}}{1-\beta_1^t} \end{cases} & \text{(Gradient Descent with Momentum, with bias correction)} \\ \begin{cases} S_{d\theta} = \beta_2 S_{d\theta} + (1-\beta_2) d\theta^2 \\ S_{d\theta} := \frac{S_{d\theta}}{1-\beta_2^t} \end{cases} & \text{(RMSProp, with bias correction)} \\ \theta := \theta - \alpha \frac{V_{d\omega}}{\sqrt{S_{d\theta}} + \epsilon} \end{cases}$$

with hyperparameters usually being $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

13.5 Batch Normalization [17]

It is known that normalizing the training data yields a faster training time for our ML algorithms. We can do the same for each layer in a Neural Network, for the values that the neurons take on.

So let there be an $L-$ layered Neural Network. In a layer l, we can perform a batch normalization of its units:

$$Z^{[l]}_{\text{norm}} \equiv \frac{Z^{[l]} - \mu^{[l]}}{\sqrt{\sigma^{[l]\,2} + \epsilon}}$$
$$\tilde{Z^{[l]}} \equiv \beta^{[l]} Z^{[l]}_{\text{norm}} + \gamma^{[l]}$$

where $\sigma^{[l]}(\mu^{[l]})$ is the standard deviation (mean) of the $Z^{[l]}$ layer (averaged over the training examples).

Note that $\gamma^{[l]}$ can take on the role of $b^{[l]}$, and so we only need to find values for $\gamma^{[l]}$ and $\mu^{[l]}$ (aside from the non-bias weights $\{\Theta^{[k]}\}$). The process can naturally be performed with mini-batches.

References

[1] Andrew Ng's Course on Machine Learning, Coursera:
https://bit.ly/2ZsEybi

[2] "Deep Learning Specialization", Andrew Ng & deeplearning.ai:
https://www.coursera.org/specializations/deep-learning

Neural networks:

[3] "Neural networks and deep learning", Michael Nielsen's online book (highly recommended)
http://neuralnetworksanddeeplearning.com/index.html

[4] "How the backpropagation algorithm works", Michael Nielsen
http://neuralnetworksanddeeplearning.com/chap2.html

[5] "Backpropagation from the beginning", Eric Hallström
https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d

[6] 3Blue1Brown's Youtube series on Neural Networks (highly recommended)
https://bit.ly/2sOZvBo

[7] "Neural Networks Demystified" playlist by WelchLabs (for a neat sanity check, with Python code)
https://bit.ly/2HnbuJT

Bias/Variance:

[8] "Understanding the Bias-Variance Tradeoff", Scott Fortmann-Roe
http://scott.fortmann-roe.com/docs/BiasVariance.html

Support Vector Machines:

[9] "CS229 Lecture notes, part V: Support vector machines", Andrew Ng
http://cs229.stanford.edu/notes/cs229-notes3.pdf

[10] "Support Vector Machine (SVM): A Simple Visual Explanation — Part 1", Dhilip Subramanian
https://bit.ly/2TVFPXy

[11] "An idiot's guide to Support vector machines (SVMs)", R. Berwick
http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf

[12] Support Vector Machine — Formulation and Derivation", Atul Agarwal
https://towardsdatascience.com/support-vector-machine-formulation-and-derivation-b146ce89f28

Principal Component Analysis:

[13] "A geometric interpretation of the covariance matrix", Vincent Spruyt
https://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/

Exponentially Weighted Averages:

[14] "Understanding Exponentially Weighted Averages", Andrew Ng, deeplearning.ai
https://www.youtube.com/watch?v=NxTFlzBjS-4

[15] "Bias Correction of Exponentially Weighted Averages", Andrew Ng, deeplearning.ai
https://www.youtube.com/watch?v=lWzo8CajF5s

[16] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).
https://arxiv.org/pdf/1412.6980.pdf

[17] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).
https://arxiv.org/pdf/1502.03167.pdf%20http://arxiv.org/abs/1502.03167.pdf