Machine Learning - Coursera

Andrew Ng

April 25, 2022

Contents

Chapter 1

Introduction

1.1 Introduction

- **Machine Learning**: field of study that gives computers the ability to learn without being explicitly programmed.
  - A computer program is said to *learn* from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.
- Machine learning algorithms can be roughly split into two different categories:
  1. **Supervised Learning**: learning a function that maps an input to an output based on the given input-output pairs.
     - Examples of supervised learning tasks are regression (mapping inputs to continuous outputs) and classification (mapping inputs to discrete outputs).
  2. **Unsupervised Learning**: learns patterns from unlabeled data.
     - Examples of unsupervised learning tasks are clustering and dimensionality reduction.

1.2 Linear Regression with One Variable

- We call the data that is given to the model to train on the **training set**. For supervised learning tasks, the inputs are labeled as x and the outputs as y.
  - The inputs are also sometimes referred to as **features** and there may be multiple features which the model needs to keep track of. This case will be dealt with in the next section.
  - One training example: $(x^{(i)}, y^{(i)})$ where $i \in \{1, 2, \ldots, M\}$ labels the i-th training example. M is the number of training examples on which the model will train on.
- The goal of any supervised learning algorithm is to learning the relationship between the input and output variables. This relationship is referred to as the **hypothesis**. This can be seen in Fig. 1.1.
- **Univariate Linear Regression**: linear regression with one variable. The hypothesis function is given in Eq. (1.1) where $\theta = (\theta_0, \theta_1)^{\mathrm{T}}$ are the parameters the learning algorithm needs to learn.

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1.1}$$

  - We want the model to learn the θ such that the hypothesis function outputs are "close to" the expected outputs. In essence, the model is trying to find a line that best fits the data.
  - We measure this closeness by computing how different the predicted output is from the expected output. Since it is expected that an equal number of the expected outputs will be on either side of the line. Thus, we will square the results and average. Eq. (1.2) will be our cost function for this problem, and is sometimes referred to as the mean-squared error.

$$J(\theta) = \frac{1}{2M} \sum_{i=1}^{M} [h_\theta(x^{(i)}) - y^{(i)}]^2 \tag{1.2}$$

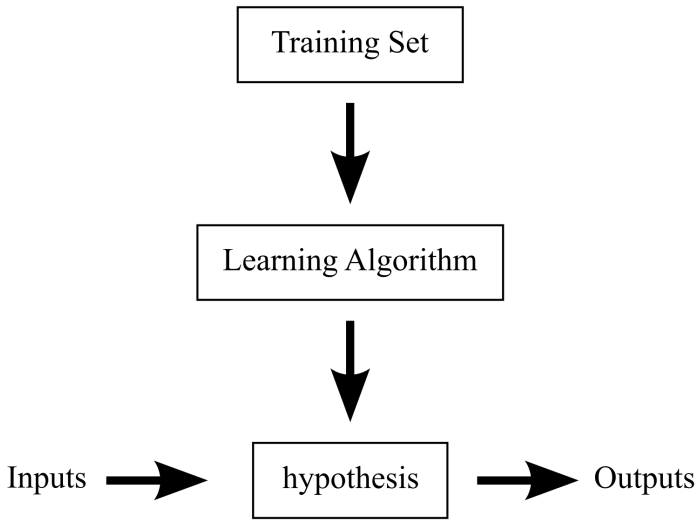

Figure 1.1: The following shows a typical flow chart for training a supervised learning model. The goal of any such model is to learning the hypothesis function $h : x \to y$.

  * **Cost Function**: function used to model the performance of a supervised learning algorithm.
  * Goal: we want to minimize the cost function, Eq. (1.2), with respect to the parameters θ. This is laid out in detail in Fig. 1.2.

1.3 Gradient Descent

- **Gradient Descent**: a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient at the current point, which corresponds to the direction of steepest descent.
  - Start with a some θ_0 and θ_1.
  - Update these parameters using gradient descent which will (hopefully) reduce the cost function. This step is repeated until convergence.
    * Gradient descent formula for updating these parameters is given by Eq. (1.3). Here, α is the **learning rate** which is a tuning parameter that determines the step size at each iteration while moving toward a minimum cost function.
      · Note that these updates are simultaneous. You must not update on parameter first then the other. These updates need to be at the same time.
      · Note that too small of a learning rate will result in slow convergence; too large of a learning rate can cause the gradient descent to overshoot the minimum and possibly diverge.

$$\theta := \theta - \alpha \nabla_\theta J(\theta) \tag{1.3}$$

1.4 Gradient Descent for Linear Regression

- Our univariate linear regression hypothesis function is given by Eq. (1.1) and the corresponding cost function is given by Eq. (1.2). In order to use gradient descent to learn the hypothesis function, we will need to take the gradient of the cost function with respect to the parameters:

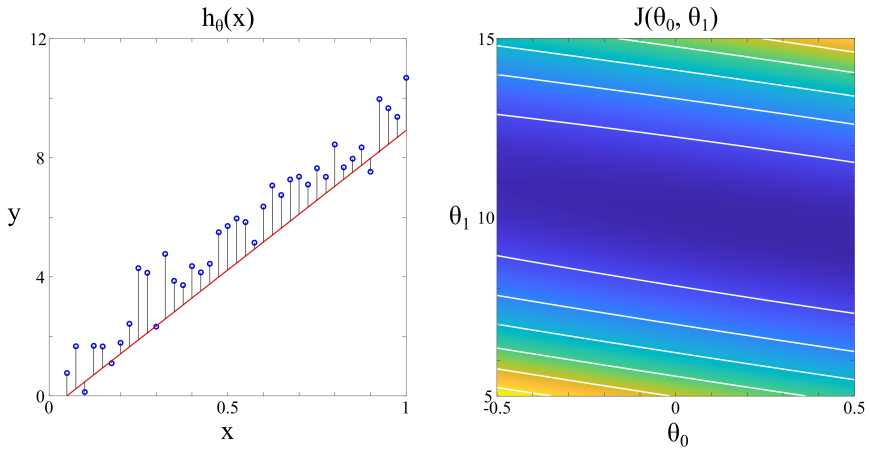

Figure 1.2: (left figure) Shows an example predicted best-fit line. The cost function (which can be imagined as the "cost" for using this best-fit line to model the data) measures the average deviation of this prediction from the expected output. These deviations are shown as black lines that are projected from each of the data points to the predicted best-fit line. (right figure) Show how the cost function varies as a function of the two parameters $\theta_{0,1}$. The theoretical best-fit line is that which minimizes the cost function (the region in dark blue in the right figure).

$$\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_0} = \frac{1}{M}\sum_{i=1}^{M}[h_\theta(x^{(i)}) - y^{(i)}]$$
$$\frac{\partial J(\theta_0,\theta_1)}{\partial \theta_1} = \frac{1}{M}\sum_{i=1}^{M}[h_\theta(x^{(i)}) - y^{(i)}] \cdot x^{(i)}$$

- Fig. 1.3 shows the results of training a linear regression model on a basic linear resistor dataset.
  - This dataset consists of $M = 39$ different measurements of the voltage across a simple linear resistor as the current is increase from 0.050 A to 1.000 A in 0.025 A increments.
- Fig. 1.4 demonstrates what happens when the learning rate is allowed to vary.

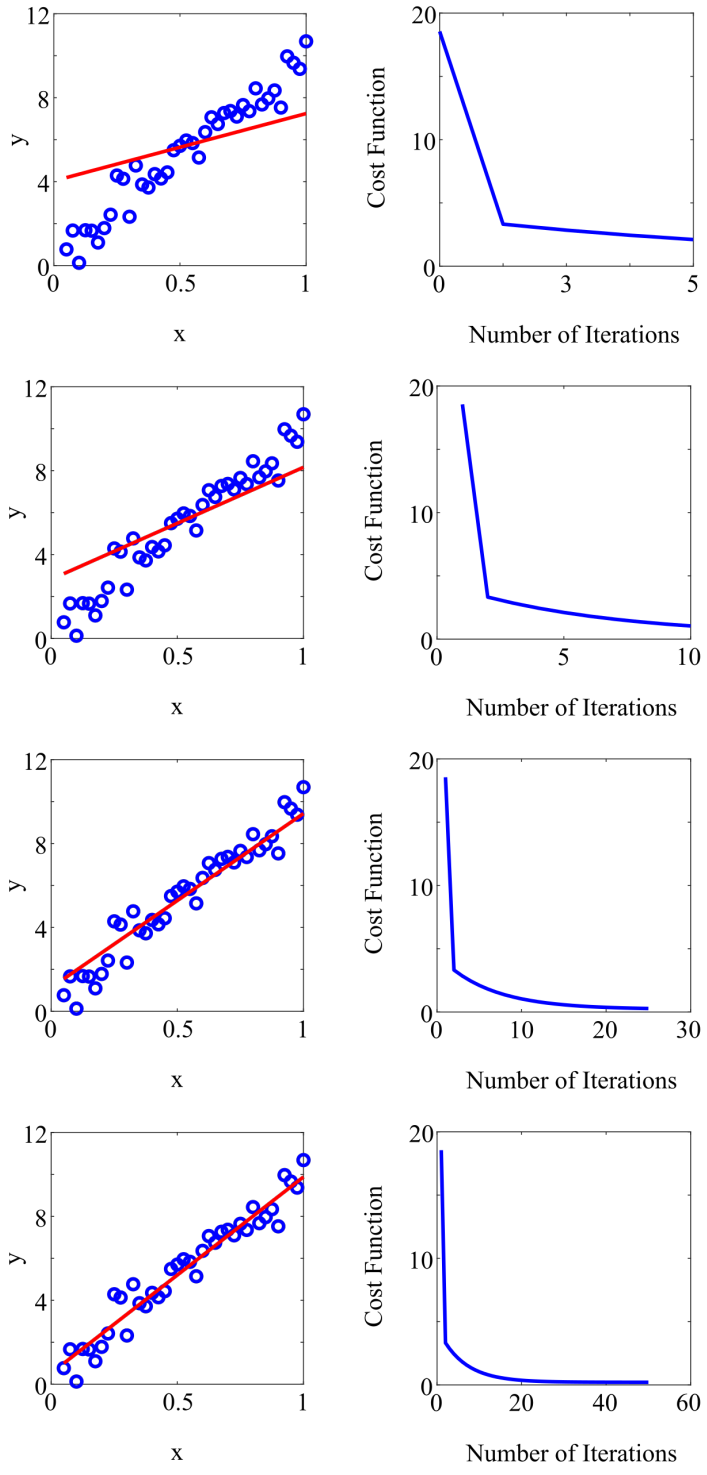

Figure 1.3: The following shows the training process of the linear regression model on a toy dataset. The learning rate is set at $\alpha = 1.0$ and the model is trained for (first row) 5, (second row) 10, (third row) 25, and (fourth row) 50 iterations.

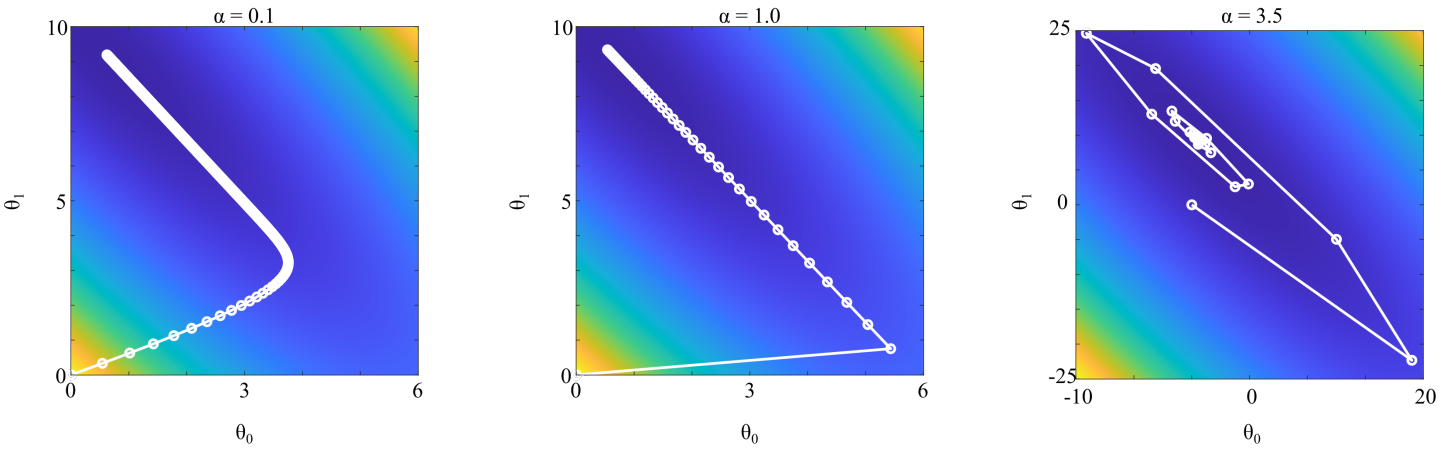

Figure 1.4: The following show what happens when the learning rate is allowed to vary. When α is small (the left figure) then gradient descent takes a long time to find the minimum value. When α is large (the right figure) then the solution will possibly overshoot the minimum value and oscillate as shown. In the worst case scenario, this could even result in the solution diverging. Finally, for a well-tuned learning rate (the middle figure), gradient descent finds the minimum value efficiently.

Chapter 2

Multivariate Linear Regression

2.1 Multiple Features

- **Feature (variable)**: an individual measurable property or characteristic of a phenomena. Choosing informative, discriminating, and independent features is a crucial element of effective algorithms.
  - Label the j-th feature of the i-th training example as $x_j^{(i)}$ where $i \in \{1, 2, \ldots, M\}$ and $j \in \{1, 2, \ldots, N\}$. We let N be the number of features in the dataset.
  - For each training example, we can arrange these features into a vector $\mathbf{x}^{(i)}$ of size $N \times 1$.
- Before, for univariate linear regression, the hypothesis function was given by Eq. (1.1). Now, for the case of multiple features (multivariate linear regression), the hypothesis function is given by

$$h_\theta(x_1, x_2, \ldots, x_N) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_N x_N \tag{2.1}$$

  - As with each training example input being arranged into a vector, we can arrange the $N + 1$ parameters into an $(N + 1) \times 1$ vector as well: $\theta = (\theta_0, \theta_1, \theta_2, \ldots, \theta_N)^{\mathrm{T}}$.
  - If we insert $x_0 = 1$ into the first entry of $\mathbf{x}^{(i)}$ making it an $(N + 1) \times 1$ vector. Then we can replace Eq. (2.1) with a dot product.

$$h_\theta(\mathbf{x}) = \theta^{\mathrm{T}}\mathbf{x} \tag{2.2}$$

2.2 Gradient Descent for Multiple Variables

- The cost function for multivariate linear regression is similar to the cost function for univariate linear regression given in Eq. (1.2). Now, we replace each training example $x^{(i)}$ with $\mathbf{x}^{(i)}$, and θ is understood to be an $(N + 1) \times 1$ vector of parameters.

$$J(\theta) = \frac{1}{2M} \sum_{i=1}^{M} [h_\theta(\mathbf{x}^{(i)}) - y^{(i)}]^2 \tag{2.3}$$

  - Note that Eq. (1.2) is a special case (when $N = 1$) of Eq. (2.2).
- Now there are $N + 1$ parameters we need to update simultaneously (as explained in section 1.3). The partial derivative of the cost function Eq. (2.2) for the j-th parameter is

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{M} \sum_{i=1}^{M} [h_\theta(\mathbf{x}^{(i)}) - y^{(i)}] x_j^{(i)}$$

  - Note that $x_0^{(i)} = 1$ for all $i \in \{1, 2, \ldots, M\}$.

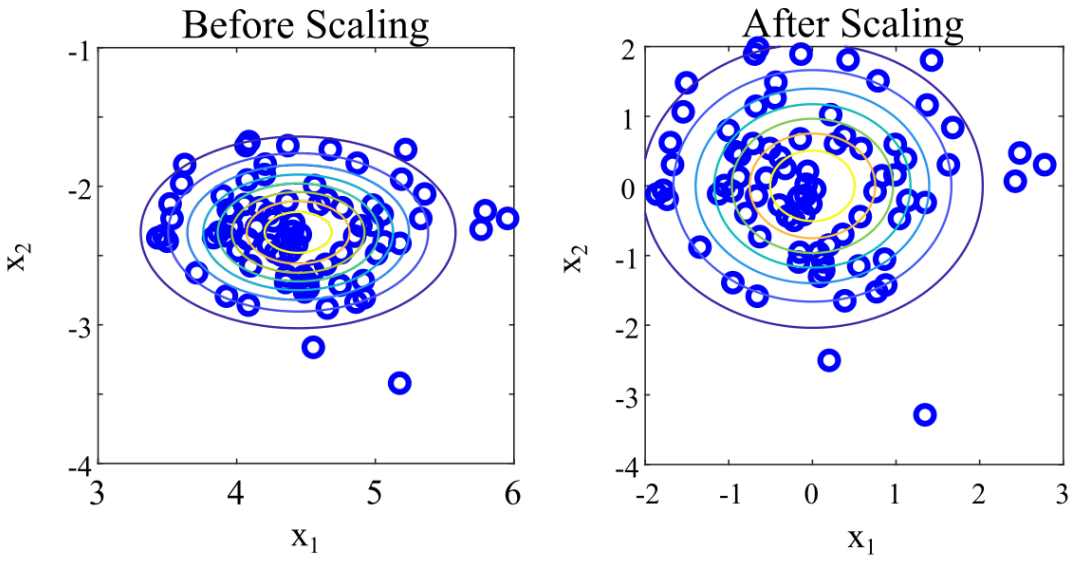

Figure 2.1: This demonstrates the process of featuring scaling. The data consists of randomly sampled data points from a normal distribution. Initially, the mean is $(4.4433, -2.3329)$ with a standard deviation of $(0.5439, 0.3310)$. This data, along the contour plot of the underlying probability distribution is outlined in the left plot. The right plot shows the results of feature scaling. Now the mean for both features is zero and the standard deviation for both is one. The contours of this scaled data now are more circular, as seen in the right plot. This has the effect of making gradient descent converge quicker.

2.3 Gradient Descent in Practice: Feature Scaling

- **Feature scaling**: a method used to normalize the range of features of a dataset. This will result in all of the features being on the same scale which will result in a faster convergence of gradient descent.
  - We want to roughly have each feature be on the interval $[-1, 1]$.
  - This can be done through a process of **rescaling**. Rescaling can be performed by either dividing by the range of the feature, or by dividing by the standard deviation of the feature over the dataset.
  - Note: in what follows, we will not be modifying the zeroth feature since it is (by design) supposed to be all ones.
- **Mean normalization**: the process of shifting all of the features such that they have zero mean. This can be done by simply subtracting each feature by the average of the j-th feature taken over the entire dataset.
  - Note that mean normalization should be done first, followed by the rescaling step. This is illustrated in Fig. 2.1.

$$x_j := \frac{x_j - \mu_j}{\sigma_j} \tag{2.4}$$

$$x_j := \frac{x_j - \mu_j}{\max_i x_j - \min_i x_j} \tag{2.5}$$

2.4 Features and Polynomial Regression

- **Polynomial regression**: a form of multivariate linear regression where the hypothesis function assumes there is a polynomial relationship of n-th degree between the input(s) and outputs.
  - It is considered a special case of multivariate linear regression. One is allowed to create new features from the original features provided in the dataset. For example, if one has features x_1 and x_2, then one could in theory create feature $x_1^2 x_2^4$ or $x_1 x_2^9$.
  - These new features do not need to be necessarily polynomial, they could be defined in terms of other elementary functions, so long as they don't diverge. For example, one could define $x_1\sqrt{x_2}$.

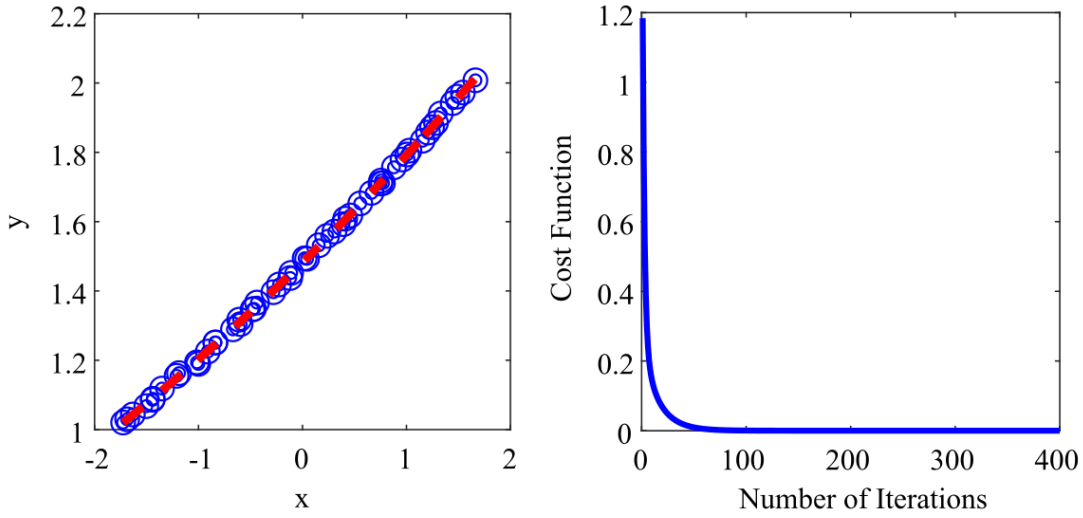

Figure 2.2: (left figure) shows the input data (blue dots) along with the learned best-fit line (red dashed line). (right figure) shows the cost function as the model learns from the data.

  * Polynomial regression specifically requires all the terms be polynomial in nature.

- An example of this is shown in Fig. 2.2 where we are trying to fit a best-fit line to free fall data taken in the lab. This consists of $M = 50$ measurements consisting of the initial height from which the ball was dropped and the time it took the ball to hit the ground.
  - In the lab, we assumed that air resistance was negligible. So the model we used for fitting the data was $y_0 = \frac{1}{2}gt^2$, where y_0 is the initial height from which the ball was dropped, $g = 9.81$ m/s^2 is the acceleration due to gravity on Earth, and t is the time of flight.
  - We added quadratic polynomial features to the feature scaled input data. We trained the model on the data for 400 iterations with a learning rate of $\alpha = 0.1$. The results we found were that $\theta = (1.4823, 0.2944, 0.0153)^{\mathrm{T}}$ with a root mean-squared error of 0.0038.

Chapter 3

Logistic Regression

3.1 Classification

- **Classification**: a predictive modeling problem where a class label is predicted for a given example from a problem dataset. Classification can be subdivided into two different types:

  1. *Binary* Classification - predicting whether an element belongs to one class or another class. Sometimes this is framed as deciding whether an element belongs to a given class (usually a true or positive) or not (false or negative).
     - The outputs are labels which we can convert to integers. For this case, we use zero (the negative class) and one (the positive class) (i.e., $y \in \{0, 1\}$).
  2. *Multi-Class* Classification - predicting which class from a number of different classes an element belongs from.

- Our hypothesis function now corresponds to a probability of an element belonging to the positive class or not: $h_\theta(\mathbf{x}) = \Pr(\mathbf{x} \in P|I)$ where P is the positive class and I is any relevant background information. We can assign $\mathbf{x}$ to P if the probability exceeds 0.5. If it is below 0.5, then it is more likely that $\mathbf{x}$ belongs to P^{C}.
  - Recall however that $h_\theta(\mathbf{x})$ as defined in Eq. (2.2). There is no constraints on this function. So it is perfectly possible for h to exceed one or go below zero. In order to perform this classification task, we need to somehow constrain h to the interval $[0, 1]$.

3.2 Hypothesis Representation

- From Bayesian data analysis, when we want to look at the probability of an element belonging to a set or not, one can show that **sigmoid function** can be utilized. In our case then, we can insert Eq. (2.2) into the sigmoid to get the probability of an element belonging to P or not. This is graphed in Fig. 3.1.

$$h_\theta(\mathbf{x}) = \frac{1}{1 + \exp(-\theta^{\mathrm{T}}\mathbf{x})} \tag{3.1}$$

- For example, if $h_\theta(\mathbf{x}) = 0.75$, then there is a 75% chance that $\mathbf{x}$ belongs to P and a 25% chance it doesn't.

3.3 Decision Boundary

- Based on Fig. 3.1 and Eq. (3.1), if $\theta^{\mathrm{T}}\mathbf{x} \geq 0$ then $h_\theta(\mathbf{x}) \geq 0.5$ and if $\theta^{\mathrm{T}}\mathbf{x} < 0$ then $h_\theta(\mathbf{x}) < 0.5$. Therefore, it follows that the line (or in higher dimensions, the **hyperplane**) $z = \theta^{\mathrm{T}}\mathbf{x}$ defines the boundary between elements that are more likely to belong to P and those elements that don't. This boundary is referred to as the **decision boundary**. An example of this is depicted in Fig. 3.2.

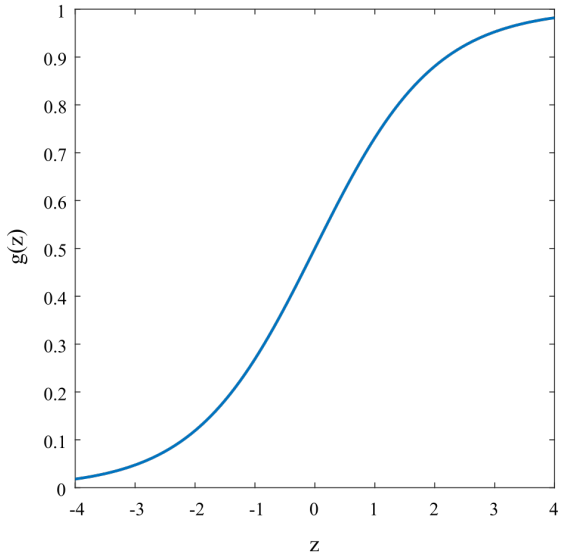

Figure 3.1: The following shows the plot of the sigmoid function for values of z in the interval $[-4, 4]$.

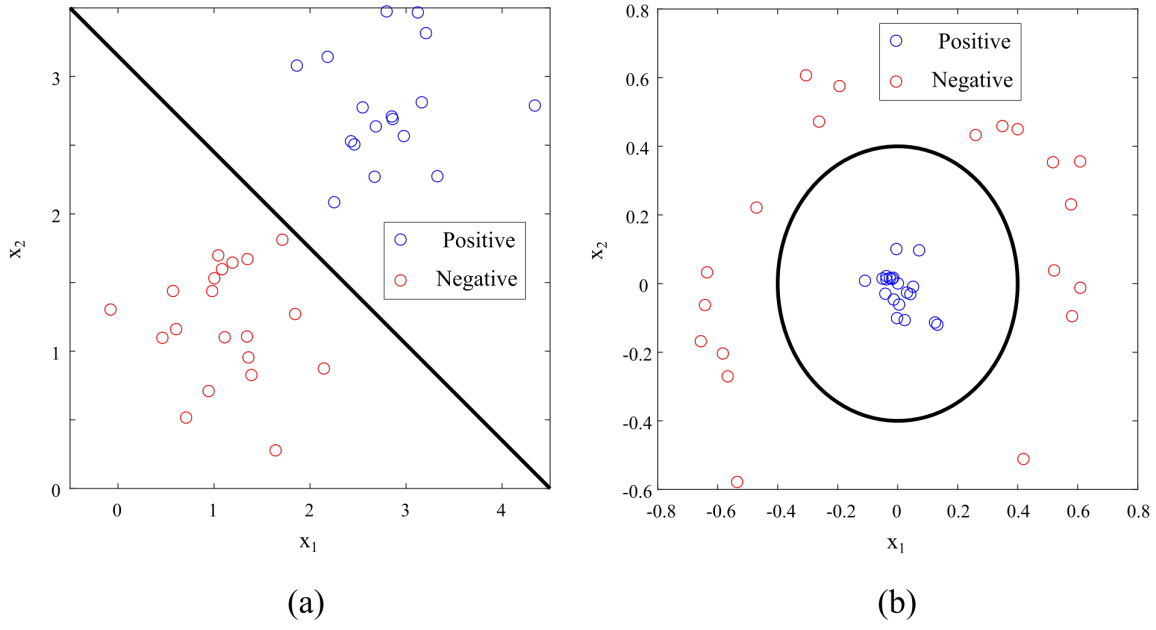

Figure 3.2: (a) shows an example of a linear decision boundary. (b) shows an example of a non-linear decision boundary. In either case, one can distinguish which elements belong to which class based on which side of the decision boundary they lie on.

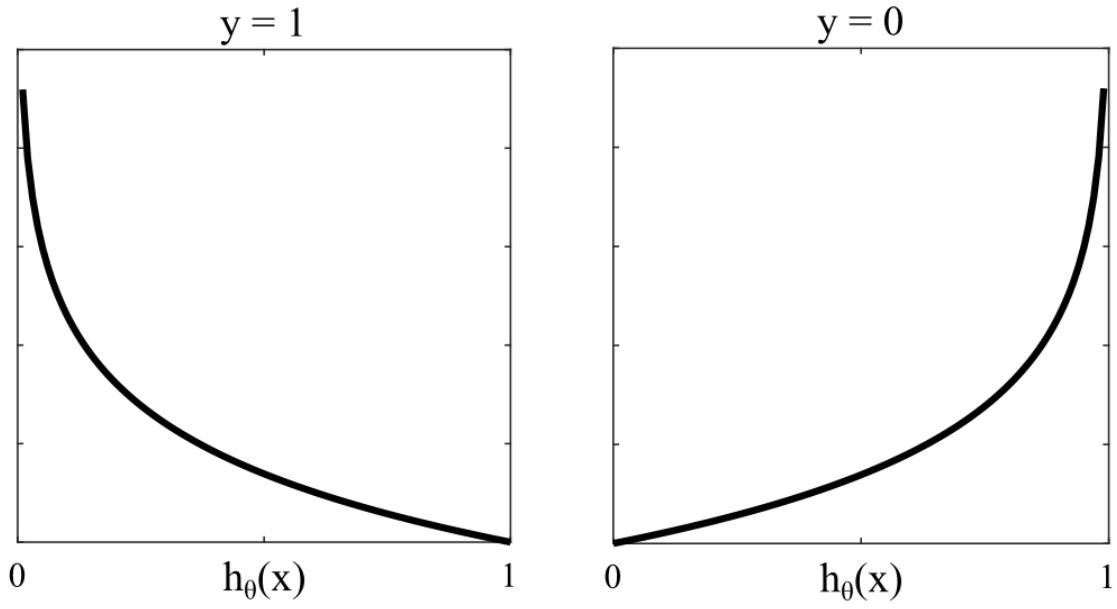

Figure 3.3: (left) shows the cost function when the given label is $y = 1$ (i.e., the positive class). Here, if we predict that the label should be $y = 1$, then our prediction matches the given label, so there should be no cost. If we predict the probability of being in the positive class is small, then our prediction should be penalized more. This is the opposite for the right figure. (right) shows the cost function when the given label is $y = 0$ (i.e., the negative class). Here, if we predict that the label should be $y = 0$, then our prediction matches the given label, so there should be no cost. If we predict the probability of being in the positive class is large, then our prediction should be penalized more.

3.4 Cost Function

- In order to determine the parameters that will define the decision boundary, we will implement gradient descent again assuming we can find a cost function to minimize which is convex (or, simply put, one who only has one local minimum). If the cost function is non-convex, then there could exist multiple local minima and gradient descent might find an erroneous result. The mean-squared error used in Chapter 3 in Eq. (2.3) applied in this case is non-convex and therefore should be discarded.
  - If $h_\theta(\mathbf{x})$ is above 0.5, then we want to strongly penalize a negative class prediction, and vice versa. If $h_\theta(\mathbf{x})$ is below 0.5, we want to strongly penalize a positive class prediction. One way to do this is through logarithms: $J(\theta) = -\log[h_\theta(\mathbf{x})]$ if the associated label is one, and $J(\theta) = -\log[1 - h_\theta(\mathbf{x})]$ if the associated label is zero. The intuition is laid out in Fig. 3.3.
    - We can combine together these two parts to build our new cost function for logistic regression. This cost function is known as the **cross-entropy**.

$$J(\theta) = -\frac{1}{M}\sum_{i=1}^{M}\{y^{(i)}\log[h_\theta(\mathbf{x}^{(i)})] + (1 - y^{(i)})\log[1 - h_\theta(\mathbf{x}^{(i)})]\} \tag{3.2}$$

    - In order to use this in gradient descent, we need to compute its gradient with respect to the parameters θ. Oddly enough, the gradient is the same as the gradient of the cost function Eq. (2.3).

- Two examples of binary classification are shown in Fig. 3.4 using the data used to generate Fig.3.2.
  - Both the linear and non-linear decision boundary datasets have $M = 40$ training examples, each consisting of two features each $N = 2$, labeled as x_1 and x_2.
  - The logistic regression model was trained for 50 iterations with a learning rate of $\alpha = 1$.
  - The **classification accuracy** is also shown. This is the average number of times the model is able to accurately predict the output label.

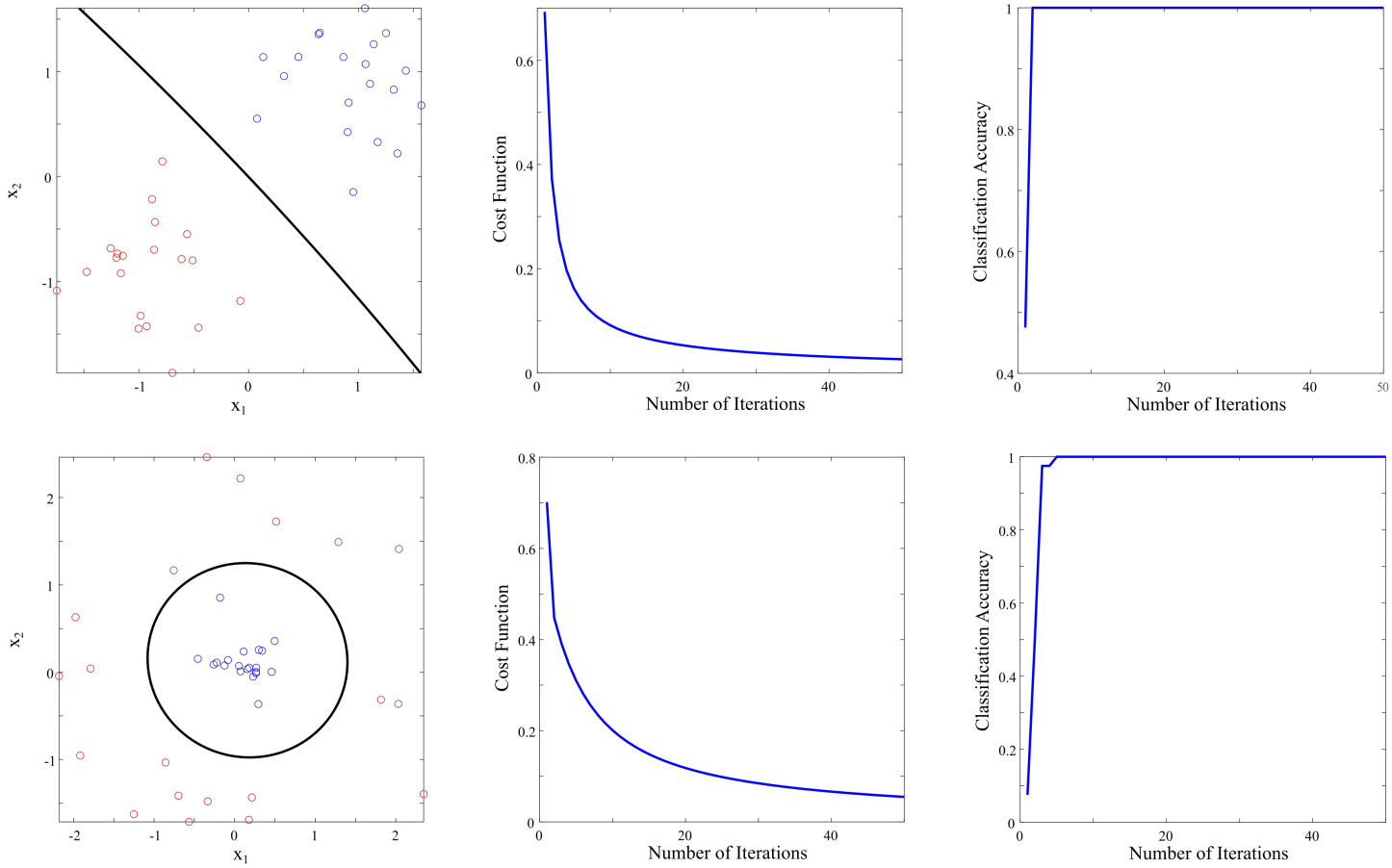

Figure 3.4: (top row) shows an example of binary classification where the decision boundary is linear. (bottom row) shows an example of binary classification where the decision boundary is non-linear. Even though we added quadratic features to the hypothesis function, this is still technically linear, at least in the higher dimensional space where we mapped our original features onto. When the decision boundary is projected back into the original space, it appears to be non-linear.

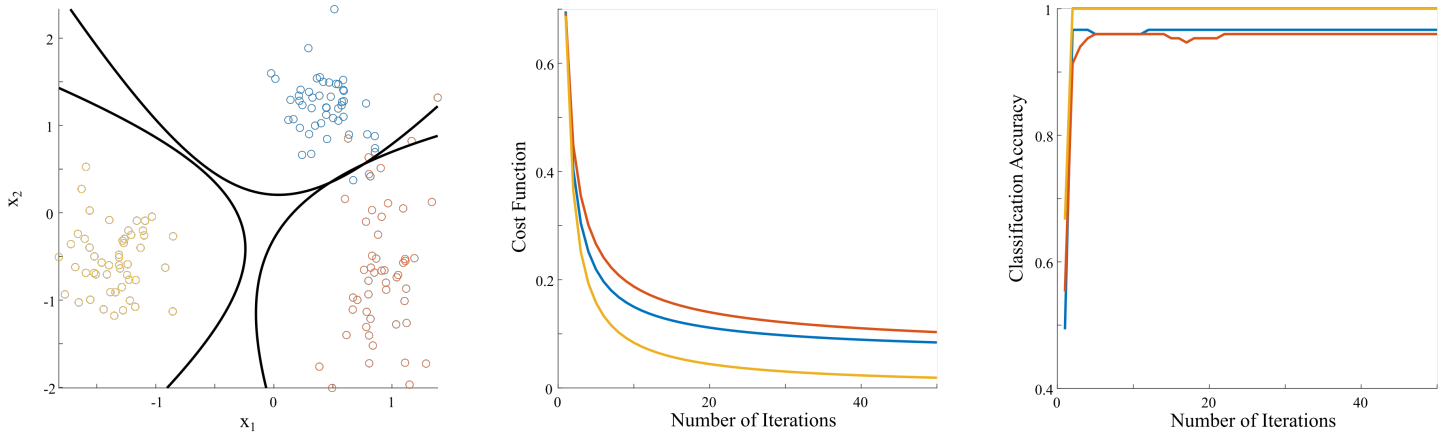

Figure 3.5: This shows the results of running the one-vs-all algorithm on a toy dataset. The three classes are labeled with blue, red, and orange in each of the three plots. Note that since classes blue and red are slightly overlapping, the model struggles to discern between the two at that point.

3.5 Multi-Class Classification: One-vs-All

- The data provided could be grouped into more than just two classes. In this case, we need to be careful with how we perform logistic regression, since the model was developed only with binary classification in mind.
  - We can transform the problem of trying to classify data into multiple classes into a series of C binary classification problems. Instead of trying to compute the probabilities that a data is in each of the classes, we can ask whether a data belongs to one class or not. We then perform this C times. This is referred to as the **one-vs-all** algorithm.
    - For each class $c \in \{1, 2, \ldots, C\}$ predict the probability that $y^{(i)} = c$.
    - For each training example, we will have C probabilities (which should add to one). The one with largest probability will correspond the class the data belongs to.
    - This is demonstrated in Fig. 3.5 where we used a toy model to train a one-vs-all algorithm. Here there were $M = 150$ training examples with three classes. Each iteration of the algorithm was run 50 times with a learning rate of $\alpha = 1$.

3.6 Regularization - The Problem of Overfitting

- **Overfitting**: a modeling error in statistics that occurs when a function is too closely aligned to a limited set of data points. As a result, the model is useful in reference only to its initial dataset, and not any other datasets. If we have too many features, the learned hypothesis may fit the training set very well, but fail to generalize to new examples.
  - Overfitting is associated with a high **variance** situation - referring to changes in the model when using different portions of the training data. It is the variability in the model, which is a measure of how well the model can adjust to new data.
  - Characteristics of high variance are: (1) the model tries to fit the noise in the model, (2) too many features.
- **Underfitting**: a scenario where a data model is unable to capture the relationship between the input and output variables accurately, generating a high error rate on both the training set and any unseen data.
  - Underfitting is associated with a high **bias** situation - a phenomena that skews the result of an algorithm in favor or against an idea. Bias is considered to be a systematic error that occurs in the model itself due to incorrect assumptions. Bias is also defined as the error between the average model prediction and the ground truth.
  - Characteristics of high bias are: (1) failure to capture proper data trends, (2) overly simplified, (3) high error rate.
- Examples of overfitting, underfitting, and a properly trained model are shown in Fig. 3.6.

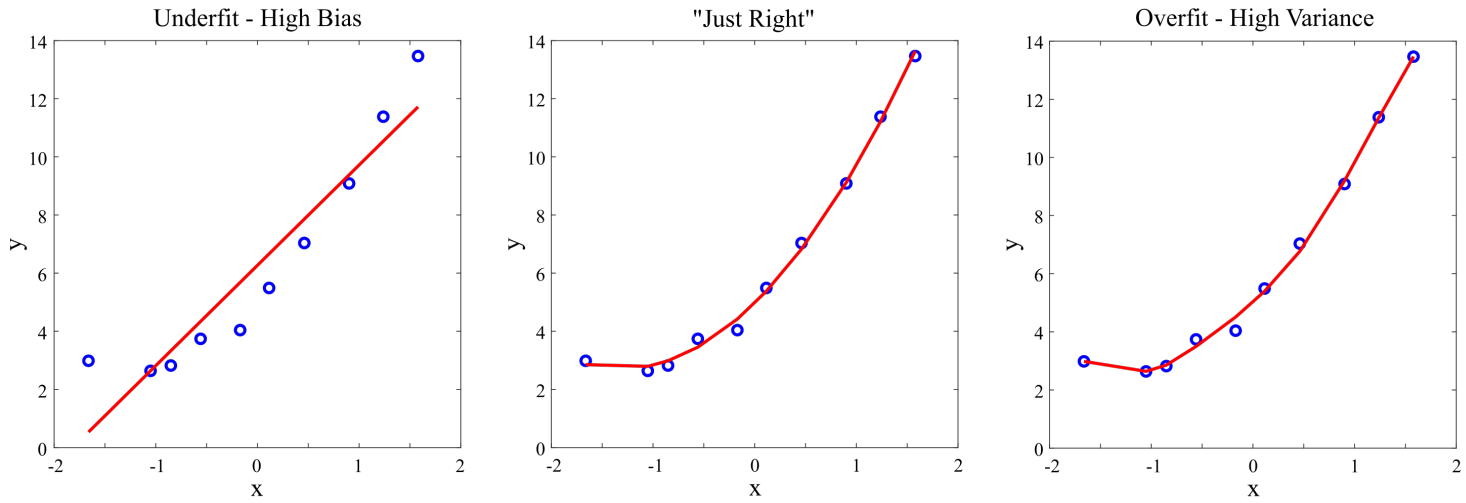

Figure 3.6: (left) shows a scenario of an underfit model. Here, the model cannot account for some of the structure found in the data, and so misses some potentially important details. (center) shows a scenario of a well-fit model. Note that one can never fully rid a model of bias or variance, only minimize it due to noise. (right) shows an overfit model. If we try to apply this model to a new dataset, then it will most likely fail.

- One can try to mitigate a model that is overfitting the data by either reducing the number of features, using a better model via some model selection technique, or use regularization.
- **Regularization**: the process of adding more information in order to solve an ill-posed problem or to prevent overfitting. Regularization adds a term to the cost function in order to penalize the model in order to reduce the generalization error.

3.7 Regularization - Cost Function

- With regularization, we want to constrain the learned parameters in such a way as to make our hypothesis function "simpler." This will result in our model being less likely to become overfit.
  - Regularization reduces the variance in a model by increasing the bias in the model. This is the so-called **bias-variance trade-off**. You will never be able to fully remove either of them.
  - By allowing the regularization parameter to become very large (as seen as the λ term in Eq. (3.3)), the model will no longer be overfit, but instead will be underfit and gradient descent will fail to converge.
- For linear regression, the cost function given in Eq. (2.3) can be modified to allow for L_2-regularization to give

$$J(\theta) = \frac{1}{2M}\left(\sum_{i=0}^{M}[h_\theta(\mathbf{x}^{(i)}) - y^{(i)}]^2 + \lambda\sum_{j=1}^{N}\theta_j^2\right) \tag{3.3}$$

  - Eq. (3.3) has a gradient given by Eq. (3.4) where $\delta_{j,0}$ is the Kronecker delta, and is used to filter out the zeroth term from the gradient.

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{M}\sum_{i=1}^{M}[h_\theta(\mathbf{x}^{(i)}) - y^{(i)}]x_j^{(i)} + \frac{\lambda}{M}\theta_j(1-\delta_{j,0}) \tag{3.4}$$

- For logistic regression, we can apply the same modification to Eq. (3.2) to allow for L_2-regularization, resulting in

$$J(\theta) = -\frac{1}{M}\sum_{i=1}^{M}\{y^{(i)}\log[h_\theta(\mathbf{x}^{(i)})] + (1-y^{(i)})\log[1-h_\theta(\mathbf{x}^{(i)})]\} + \frac{\lambda}{2M}\sum_{j=1}^{N}\theta_j^2 \tag{3.5}$$

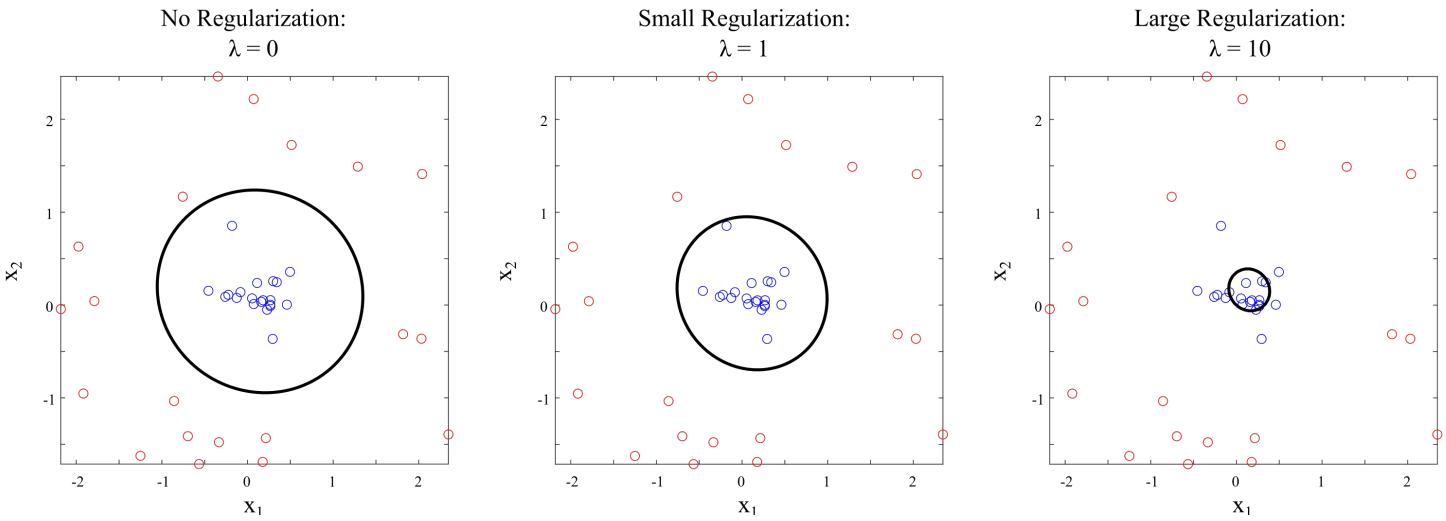

Figure 3.7: (left) shows the results of training of a binary classification model with no regularization. (center) shows a more refined model after using a regularization parameter of $\lambda = 1$. (right) shows when the regularization parameter is too high, in which case the decision boundary fits poorly.

– Eq. (3.5) has a gradient given by Eq. (3.6) where again $\delta_{j,0}$ is the Kronecker delta, and is used to filter out the zeroth term from the gradient.

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{M}\sum_{i=1}^{M}[h_\theta(\mathbf{x}^{(i)}) - y^{(i)}]x_j^{(i)} - \frac{\lambda}{M}\theta_j(1-\delta_{j,0}) \tag{3.6}$$

- Note that in Eq. (3.3) the second summation excludes θ_0 which corresponds to the constant-term in our model. If it were to be included, all it would do is shift the origin of the outputs. This also explains why allowing λ to become large results in underfitting - all of the $\theta \in \{\theta_2, \ldots, \theta_N\}$ would be shrunk to near-zero, whereas the constant term would be left alone.
- An example of this is laid out in Fig. 3.7 where we apply L_2-regularization to a binary classification problem.

Chapter 4

Neural Networks

4.1 Non-Linear Hypotheses

- Logistic regression (for binary classification or the one-vs-all algorithm) is decent when applied to low dimensional scenarios as shown in Fig. 3.7.
  - If we want to model non-linear decision boundaries, then we can simply add more multinomial terms to the hypothesis function.
  - As we add more multinomial terms, what once was a low dimensional problem becomes a higher dimensional one. As the decision boundary becomes more non-linear, the more complexity we need in our model and thus the more of these terms we will need, resulting in a high dimensional problem.
    - To see this, say we try to apply logistic regression to the problem of digit classification, where the training images each are of size 10×10 (or in other words, each image has 100 pixels). Therefore there are 100 features already. If we simply want to add quadratic features, then this number will blow up to 5151 features altogether.
    - Using logistic regression here is highly impractical, albeit still possible.

4.2 Neurons and the Brain

- **Neural Network**: a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain works. Neural networks are capable of learning highly complex hypothesis functions that other machine learning models cannot or struggle with.
  - Each neuron in the brain receives input from other neurons (or even external stimuli) through its dendrites. This data is processed in the nucleus of the neuron, and is then fed to the next set of neurons its connected to via its axon. This is illustrated in Fig. 4.1.
- Neural networks are comprised of neurons. These neurons will function similar to how biological neurons function and can be modeled as single logistic regression modules as shown in Fig. 4.2. Neural networks consist of multiple neurons, each of which act as an individual logistic regression unit. These neurons are arranged into layers as shown in Fig. 4.3.
  - There can be several layers to a neural network, each one adding more complexity to the model. The first layer is always the **input layer**, whereas the last layer is always the **output layer**.
    - The output layer can have multiple output neurons, which correspond to different classes. The outputs of these neurons corresponds to how likely the neural network thinks the input data belongs to each class.
    - Allows for multi-class classification.
  - All the layers in between the input and output layers are called **hidden layers**. They are named such because we don't know precisely what they are doing or why they do it.

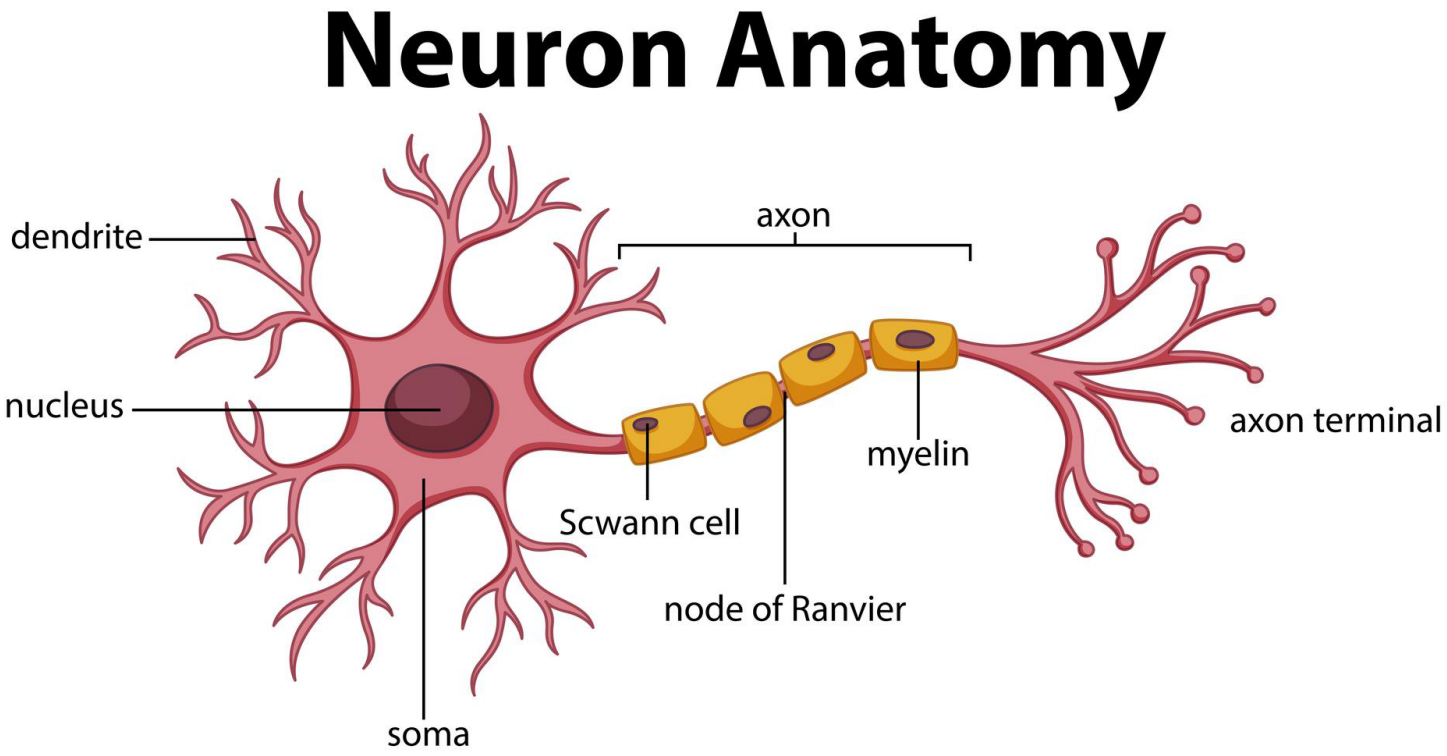

Figure 4.1: The following image illustrates the basic structures of a neuron in the human body. The neurons that comprise neural networks are based off this structure. Image courtesy of the website *vecteezy.com*

4.3 Model Representation

- **Neural Network Topology**: refers to the way the neurons are connected, and it is an important factor in network functioning and learning. We will represent the topology of a neural network with the vector $\mathbf{N} = (N_1, N_2, \ldots, N_L)$ where N_l is the number of neurons in the l-th layer (not including the bias neuron).
  - For example, in Fig. 4.3, the network topology is $\{2, 5, 2, 1\}$.
  - If there are N features in the data, then the input layer will need to have $N_1 = N$ neurons. Likewise, if there are C classes with which to classify the data, then the output layer will need to have $N_L = C$ neurons.
- As can be seen in Fig. 4.3, $a_j^{(l)}$ will represent the activation of the j-th neuron in the l-th layer. These will be arranged into a vector $\mathbf{a}^{(l)} = (a_1^{(l)}, a_2^{(l)}, \ldots, a_{N_l}^{(l)})$ of size $N_l \times 1$. For the input layer, we have $\mathbf{a}^{(1)} = \mathbf{x}^{(i)}$. Note that before these activations can be fed into the next layer we need to add the bias neuron $a_0^{(l)} = 1$. Thus, when $\mathbf{a}^{(l)}$ is fed to the next layer, it will be of size $(N_l + 1) \times 1$.
- As mentioned earlier, when the information gets fed from layer to layer, it gets weighted as it goes through the input connections. If there are N_l neurons in the l-th layer, and N_{l+1} neurons in the next layer, then we can arrange the weights of each of these input connections into an $N_{l+1} \times (N_1 + 1)$ matrix, labeled $\Theta^{(l)}$. Note that the $+1$ comes from the fact that we need to account for the bias neuron.
- We can then propagate the activations of the current layer to the next layer by multiplication.

$$\mathbf{z}^{(l+1)} = \mathbf{\Theta}^{(l)} \mathbf{a}^{(l)} \tag{4.1}$$

  - This quantity is sometimes referred to as the **preactivation** since it is what gets fed into the activation function, usually labeled as g, thus giving the activations of the next layer of neurons.
  - Note that the only requirement on the activation function g is that it be non-linear. If it were to be linear, then the neural network wouldn't be able to learn non-linear hypothesis functions. Also, it should be pointed out, that the activation function for each layer could be different.

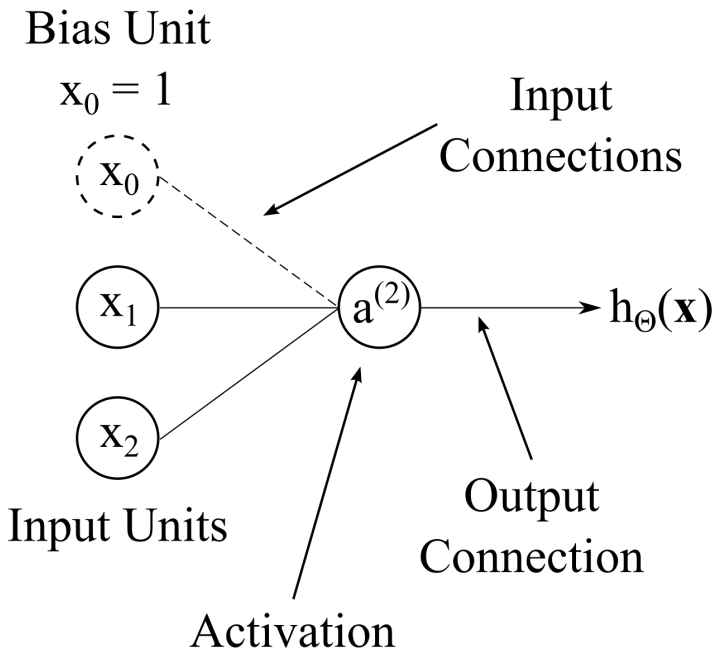

Figure 4.2: This shows a single neuron, which can be represented as a logistic regression unit. Here, the input neurons on the left receive information from the "outside" (i.e., the input matrix). The dashed neuron at the upper left represents the **bias unit** and always has an activation of one. This accounts for a constant shift in the predictions (it is here for the same reason we added a column of ones to the input matrix). Then this data will be feed to the neuron nucleus via the input connections (corresponding to the dendrites). Each of these connections will have a corresponding weight that will scale this information. These weights correspond to the parameters θ that the model is trying to learn. The scaled information is then processed by the nucleus, activating the neuron, so-to-say. Finally, the **activation** of the neuron is sent down the output connection (corresponding to the axon) and is outputted as our hypothesis function.

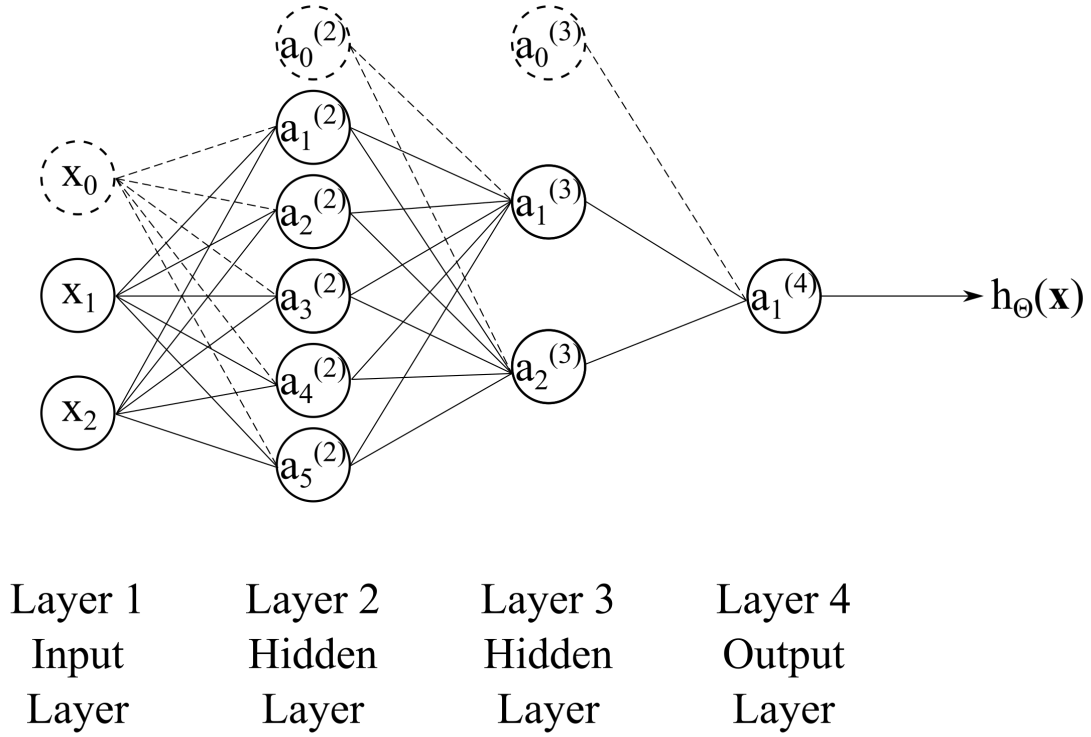

Figure 4.3: This shows the basic topology of a neural network with 4 layers consisting of $\{2, 5, 2, 1\}$ neurons each. Note that the input layer and the hidden layers each have a bias unit associated with them, just like in the case of the single neuron.

  - For now, we will assume that each layer uses the same activation function, namely the sigmoid function from Eq. (3.1).

- We continue this process of feeding the activations to the next layer until we get to the output layer, in which case we are done. This process is known as the **forward propagation** algorithm, and it is how one can use the neural network to make predictions on data.

4.4 Cost Function

- **Ones-Hot Encoding**: a process where integer categorical labels are mapped to vector quantities. If there are C classes, then these vectors will be of size $C \times 1$ and will consist of all zeros except at the index corresponding to the class the data belongs to.

  - For example: if there are three classes, then $1 \to (1, 0, 0)$ (a 1 in the index one and zeros everywhere else), $2 \to (0, 1, 0)$ (a 1 in the index two and zeros everywhere else), and $3 \to (0, 0, 1)$ (a 1 in the index three and zeros everywhere else).
  - This allows for a more efficient way to perform multi-class classification with neural networks and will allow us to compare the output of the neural network. Specifically, for multi-class classification, the cost function we can use cross-entropy as we did earlier.

$$J(\Theta) = -\frac{1}{M}\sum_{i=1}^{M}\sum_{c=1}^{C}\{y_c^{(i)}\log[h_\Theta(x^{(i)})]_c + (1-y_c^{(i)})\log[1-h_\Theta(x^{(i)})]_c\} + \frac{\lambda}{2M}\sum_{l=1}^{L-1}\sum_{j_1=1}^{N_l}\sum_{j_2=1}^{N_{l+1}}(\Theta_{j_1,j_2}^{(l)})^2 \tag{4.2}$$

4.5 Backpropagation Algorithm

- Given the cost function Eq. (4.2) we now want to minimize it with respect to the weights of the neural network $\Theta^{(l)}$. We can do this via gradient descent as outlined in section 1.3 and Eq. (1.3), but first we need to find the gradient of $J(\Theta)$.

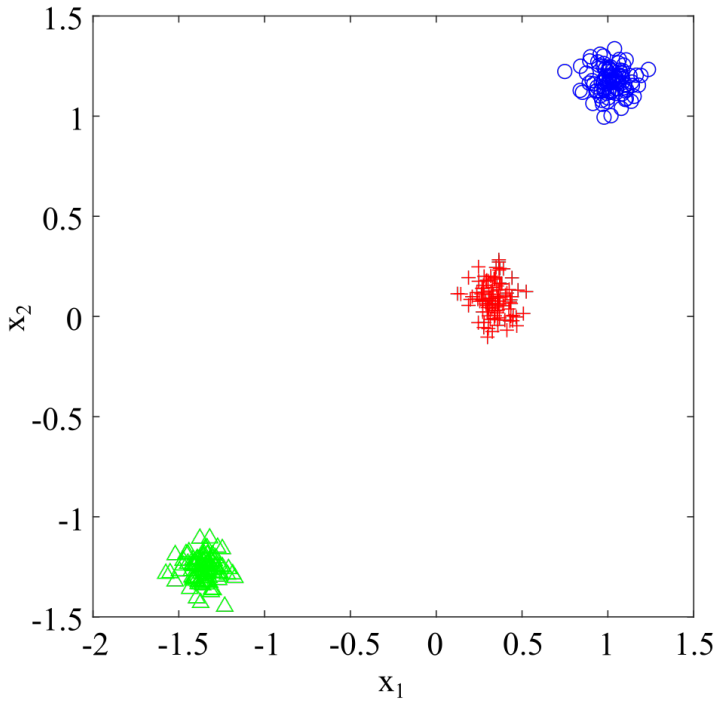

Figure 4.4: The following shows a toy dataset that will be used to train a neural network with topology $\mathbf{N} = (2, 5, 3)$. Here, blue is the first class, red is the second, and green is the third.

- **Backpropagation**: short for "backward propagation of errors," is a widely used algorithm for training neural networks using gradient descent. This algorithm calculates the gradient of the error function with respect to the neural network's weights.
  - Assume we have already performed forward propagation as outlined in Eq. (4.1) in section 4.3. This gives the predictions the neural network made with the current weightings.
  - The absolute error between these predictions and the known outputs is given by the difference $\delta^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}$. We will then trace this error backward through the network to see how the weights of the connections contributed to this error which can be found via differentiation and the chain rule.

$$\delta^{(l)} = (\Theta^{(l)})^{\mathrm{T}} \delta^{(l+1)} \circ g'(\mathbf{z}^{(l)}) \tag{4.3}$$

    - Note that $\circ$ represents element-wise multiplication and that since the input layer is assumed to be known (we feed information to the neural network) then there won't be any error in the $l = 1$ layer and so we ignore $\delta^{(1)}$.
  - The gradient can then be computed from this error in Eq. (4.3) to give

$$\frac{\partial}{\partial \Theta^{(l)}_{j_1,j_2}} J(\Theta) = \frac{1}{M} a^{(l)}_{j_2} \delta^{(l+1)}_{j_1} + \begin{cases} \lambda \Theta^{(l)}_{j_1,j_2} & \text{if } j_2 \neq 1 \\ 0 & \text{if } j_2 = 1 \end{cases} \tag{4.4}$$

- As an example, we will train a neural network to perform multi-class classification on the dataset shown in Fig. 4.4.
  - The results of the training are shown in Fig. 4.5, the decision boundaries are displayed in Fig. 4.6 and example network activations in Fig. 4.7.

4.6 Random Initialization: Symmetry Breaking

- When a neural network is trained, if all of the weights are initialized to some number (usually zero), then the network will have a hard time trying to update these parameters due to an inherent symmetry present. This is because, as can be seen in Eqs. (4.1), (4.3), and (4.4), the activations, errors, and gradients will all be the same for each layer.

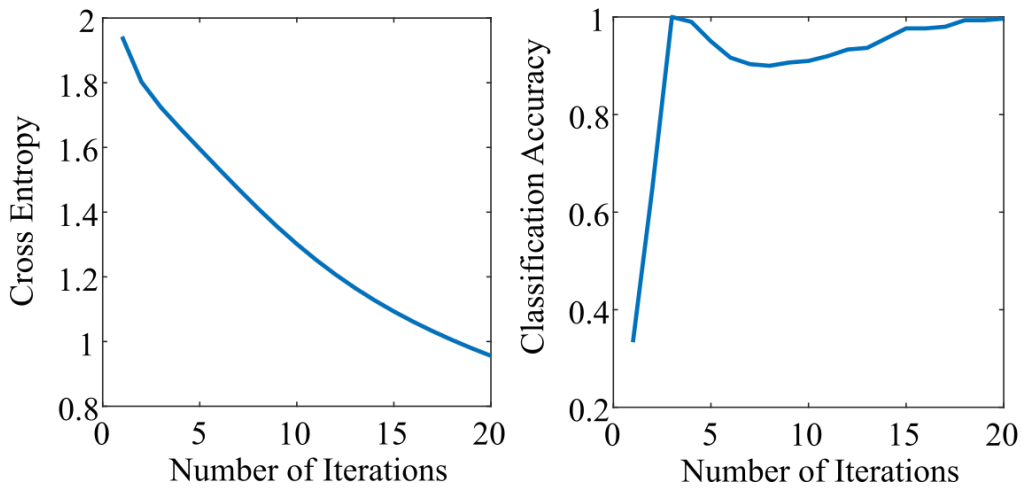

Figure 4.5: The following shows the results of training the neural network on the dataset shown in Fig. 4.4. On the left is the cross-entropy cost function given in Eq. (4.2) and on the right is the classification accuracy (i.e., the average number of times the classifier predicts the right class.)

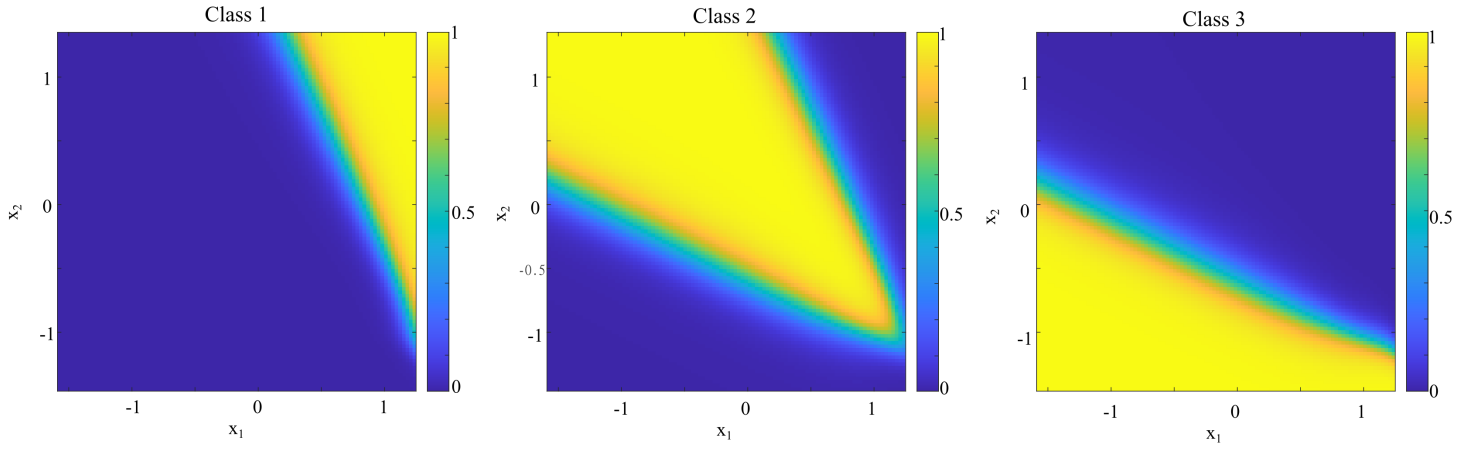

Figure 4.6: The following shows the decision boundaries for the trained neural network for each of the three classes, as labeled. For any point chosen in the yellow region, the neural network will believe it to be most likely part of that class. Points in the blue region, the neural network thinks they don't belong to that class. This is demonstrated in more detail in Fig. 4.7.

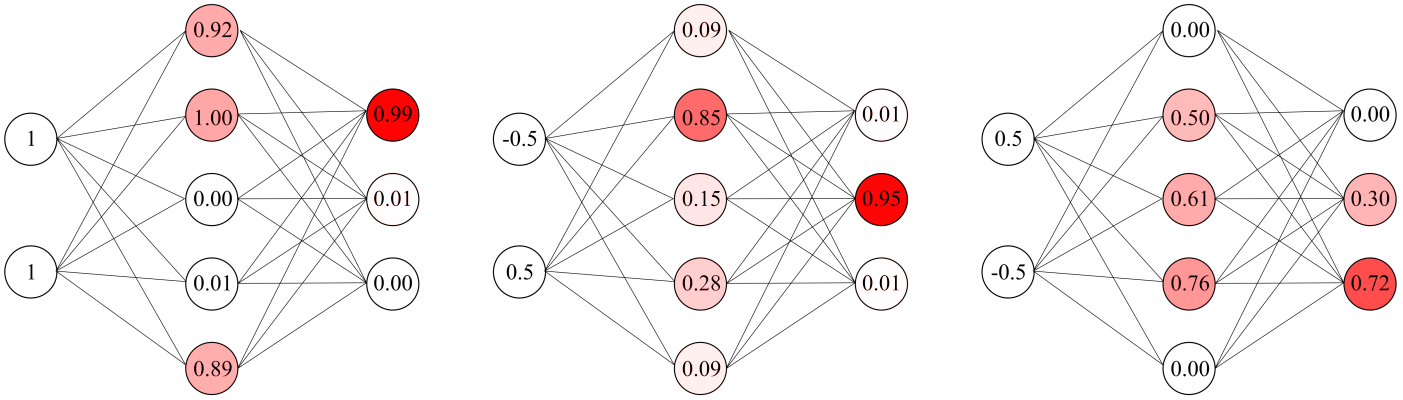

Figure 4.7: The following shows the activations of the trained neural network for three different data. Note that the class that the neural network thinks the data belongs to is the one that is most activated (indicated by a stronger shade of red). The activations of the hidden layers are also provided, albeit there is no way of truly knowing (at least currently) what they are doing or why they do it.

– We need to break this symmetry in order for the neural network to properly learn. This can be done by randomly choosing the weights, usually being uniformly sampled on the interval $[-\epsilon, \epsilon]$ for some small ϵ.

Chapter 5

Advice for Applying Machine Learning

5.1 Deciding What to Try Next

- Suppose you have some trained machine learning model. You apply this model to new data, which makes sense since you want to actually use the model in the real world, but it ends up doing very poorly. What should you try next?
  - Get more training examples: addresses high variance
  - Try smaller sets of features: addresses high variance
  - Try getting additional features (including possibly polynomial ones): addresses high bias
  - Try increasing or decreasing λ (if applicable): increasing addresses high variance and decreasing addresses high bias
  - (For neural networks): networks with a simpler topology are computationally cheap to train, but are prone to underfitting since there is less complexity. Networks with a more complex topology are more computationally expensive to train, and are prone to overfitting since there are more parameters to fit.

5.2 Evaluating a Hypothesis

- Suppose your hypothesis function fails to generalize to new data not in your training set. One way to see if your hypothesis generalizes well is to split your training set into two:
  1. **Training Set**: a dataset of examples used during the learning process and is used to fit the parameters. The model in question will initially be fit to the training set. This will consist of roughly 70% to 90% of the initial training examples.
  2. **Test Set**: a dataset used to provide an unbiased evaluation of a *final* model fit on the training dataset. The test set should be sampled from the same probability distribution as the training set (i.e., it should be gotten through similar means as the training set). This should be roughly 10% to 30% of the initial training examples.
     - If a model fit to the training data set also fits the test data set well, minimal overfitting has taken place. A better fitting of the training data set as opposed to the test data set usually points to over-fitting.
     - Train the model on the training set, then evaluate how the model does on the test set. This will give a measure of how well the model generalizes to new data.

5.3 Model Selection and Cross-Validation Sets

- After we trained our model on the training set, the training set error will tend to be lower than the generalization error. Given a number of different models each trying to perform the same task, how do we determine the best model to use?
  - We could train each of these models on the training set, then evaluate their test set errors and keep that model which has a minimal error. However, that model is likely to be an optimistic estimate of the generalization error.
    - The parameter(s) used to differentiate each model are likely to be fit to the test set.
    - We can alleviate this by introducing a new set, the **cross-validation set**. Then we can train each of the models on the training set. We can compare each of the models' performance on the cross-validation set. Finally, we can determine the generalization error using the test set.

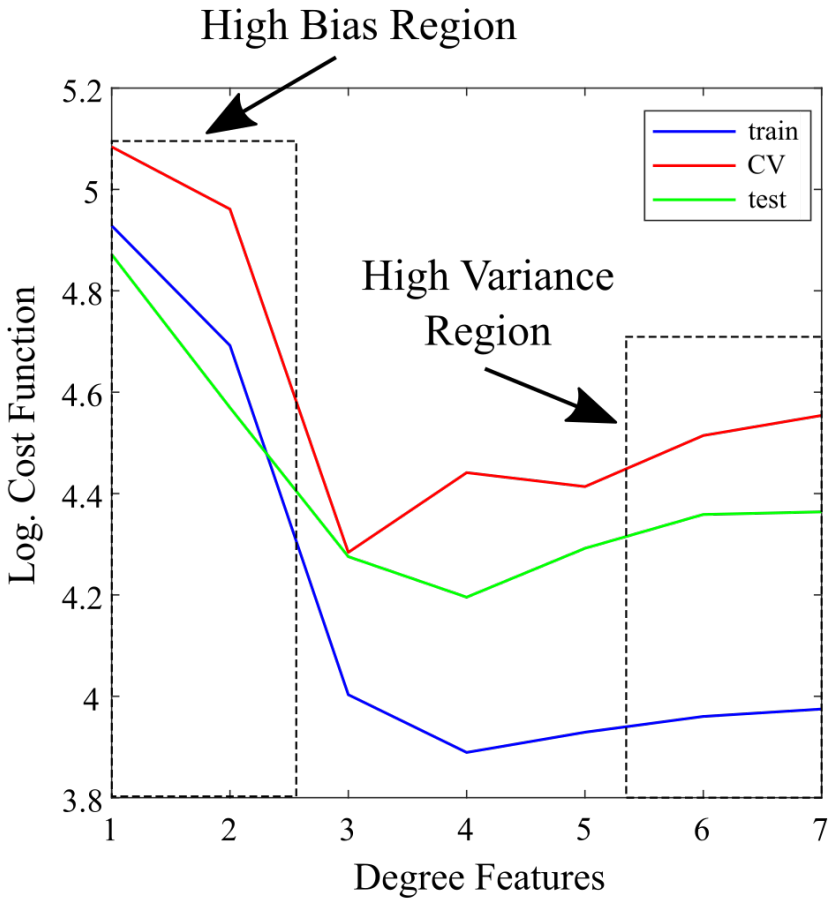

Figure 5.1: The following demonstrates the training set error, cross-validation set error, and test set error curves for varying linear regression models with differing polynomial features added.

5.4 Diagnosing Bias vs Variance

- Suppose we have, for example, several linear regression models in which we propose adding differing amounts of polynomial features. We can diagnose which models suffer from underfitting vs overfitting by plotting the cross-validation set error vs polynomial degree as shown in Fig. 5.1. This also will allow us to pick the model which performs the best.
  - Note, as can be seen, the high bias region corresponds to when J_{train} is high and $J_{\text{cv}} \approx J_{\text{train}}$. The region of high variance corresponds to when J_{train} is low and $J_{\text{cv}} \gg J_{\text{train}}$.
  - This same procedure can be applied to situations such as choosing the best L_2-regularization parameter or the best network topology.

5.5 Learning Curves

- **Learning Curve**: A learning curve is a plot of model learning performance over experience or time. Learning curves are a widely used diagnostic tool in machine learning for algorithms that learn from a training dataset incrementally. The model can be evaluated on the training dataset and on a hold out validation dataset after each update during training and plots of the measured performance can created to show learning curves.
  - As can be seen in Fig. 5.2(a), in the case of a high bias situation, getting more data will not suffice. For a high variance scenario, shown in Fig. 5.2(b), getting more data may help.

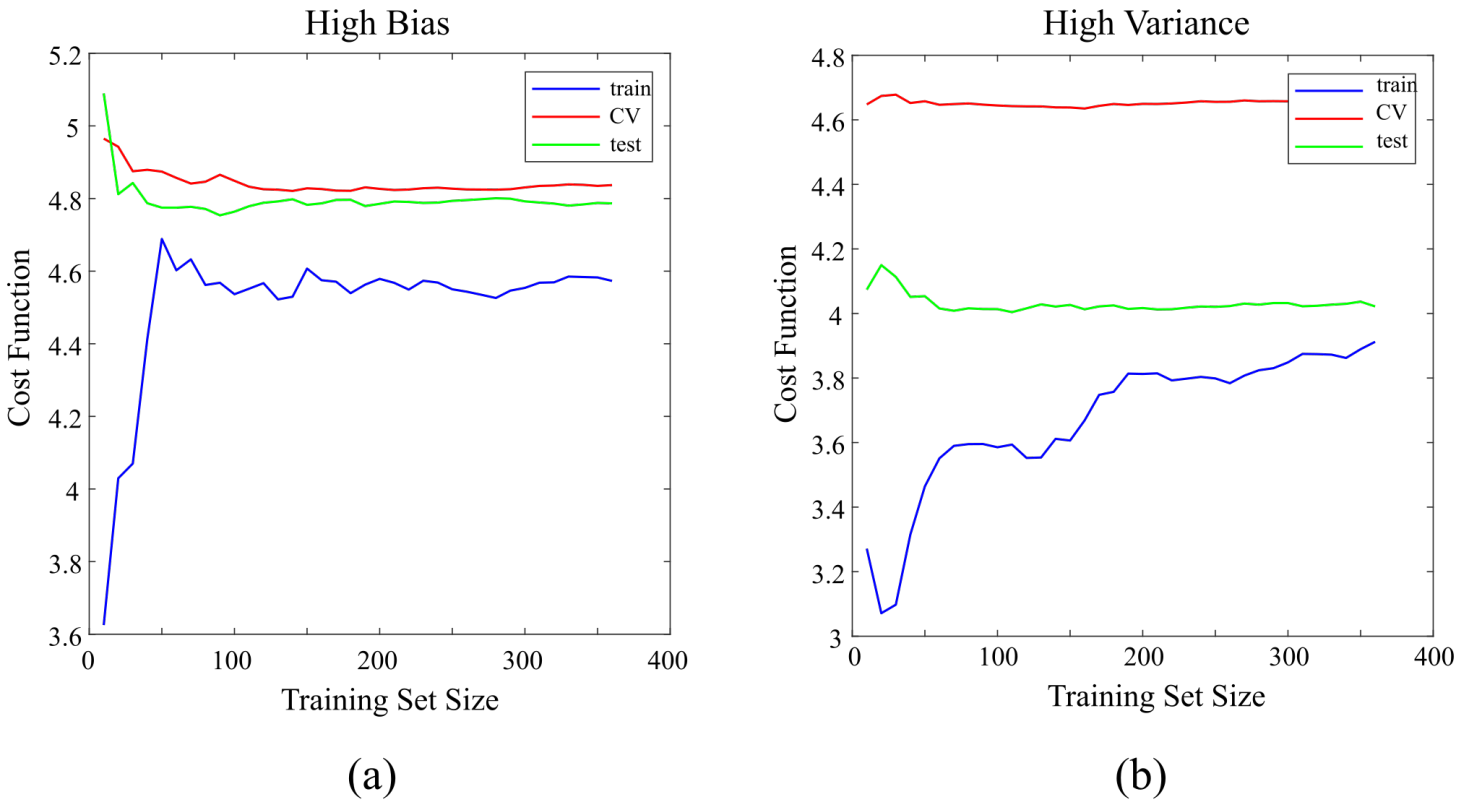

Figure 5.2: (a) Shows an example of a high bias situation. According to the learning curve, getting more data will not really help much. (b) Shows an example of a high variance situation. Here, getting more data may help with the training of the model.

5.6 Error Metrics for Skewed Classes

- Consider the case where we have the records of 100 patients and we are trying to classify them as whether or not they have type I diabetes. Of these records, let's say only 5 of them actually have diabetes. If we define a model that classifies everyone as not having diabetes ($y = 0$), then we would get a classification error of only 5% which would appear to be a reasonable result. But the model itself didn't learn anything and thus will not generalize well if applied to a larger dataset.