Summer of Science 2020

Final Report

An Overview of Machine Learning, Neural Networks, and Deep Learning

Ankit Kumar Misra
Roll No: 190050020
Computer Science and Engineering
IIT Bombay

Mentor: Soumya Chatterjee

June 14, 2020

Contents

Part I

Machine Learning

1 What is Machine Learning?

The term *machine learning* was coined by Arthur Samuel in 1959. He defined it as the field of study that gives computers the ability to learn without being explicitly programmed.

In 1998, Tom Mitchell came up with a much more formal definition of a machine learning algorithm:
A computer program is said to learn from an 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.

The main categories of machine learning algorithms are:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

1.1 Supervised Learning

Supervised learning algorithms are used when each training example in the training data consists of input feature values as well as the corresponding desired outputs. Thus, the algorithm is given the 'right answers' for the training inputs. Using a predefined cost function, the algorithm tries to generate a function that best fits this training data, and later uses this function to predict outputs for unseen inputs, that were not part of the training data. Supervised learning problems are of two types:

1. *Regression*: These problems require the machine learning model to predict a continuous valued output; e.g., a housing price prediction problem is a regression task, since prices can take continuous values.
2. *Classification*: These problems require the machine learning model to predict a discrete valued output; e.g., a problem which requires us to predict whether certain cancer tumors are benign or malignant is a classification task, since the output can take only two values; a tumor is either benign or malignant.

1.2 Unsupervised Learning

Unsupervised learning algorithms are used to find structure in unlabeled and unclassified datasets that contains only inputs. The algorithm clusters data points; i.e., it assigns a set of observations into subsets (clusters) so that observations within the same cluster is similar, while observations from different clusters are dissimilar. Unsupervised learning algorithms are used in applications such as grouping similar news articles from different sources, social media analysis, market segmentation, and even astronomical data analysis.

1.3 Reinforcement Learning

Reinforcement learning algorithms are concerned with what actions a software agent should take in an environment in order to maximize a notion of cumulative reward. Owing to this generality, reinforcement learning algorithms are used in widely varying fields. The environment is represented as a Markov Decision Process, and the algorithm tries to take decisions based on its previous experiences in similar environments, thus attempting to maximize its reward.

2 Linear Regression

Linear regression is a supervised machine learning regression algorithm which, given a training dataset along with the corresponding continuous valued labels, determines the straight line function which best fits the training data. Thus, it assumes a linear function for the task and optimizes the function's parameters to fit the training data.

NOTATION

m	the number of instances in the dataset
$\boldsymbol{x^{(i)}}$	the vector of all feature values (excluding the label) of the i^{th} instance in the dataset
$y^{(i)}$	the label (desired/expected output value) of the i^{th} instance in the dataset
$\boldsymbol{x_j}$	the vector of values from all data instances corresponding to the j^{th} feature in the dataset
$x_j^{(i)}$	the value of the j^{th} feature in the i^{th} instance of the dataset
$\boldsymbol{X}$	the entire dataset (excluding labels)
$\boldsymbol{y}$	the vector of all labels for the dataset
θ_j	the parameter of the regression model which corresponds to the j^{th} feature in the dataset
$\boldsymbol{\theta}$	the vector of all parameters of the linear regression model
$h_\theta\left(\boldsymbol{x^{(i)}}\right)$	the predicted label (output value) for the i^{th} training instance

---

2.1 Hypothesis Function

In linear regression tasks, the hypothesis function for an input data instance $\boldsymbol{x}$ having n features and parameter vector $\boldsymbol{\theta}$ is given by:

$$h_\theta(\boldsymbol{x}) = \boldsymbol{\theta}^T\boldsymbol{x} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$$

Here, x_0 is always equal to 1, and θ_0 is called the *bias term*. It can be thought of as the intercept parameter. Therefore, any linear regression task is effectively a problem of analyzing the given training dataset and figuring out the best possible parameter vector which achieves the maximum accuracy among all possible parameter vectors. This is where the cost function comes in.

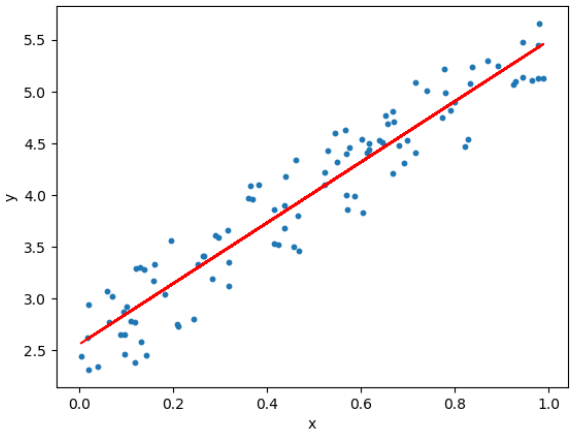

Figure 1: Linear regression

2.2 Cost Function

In a machine learning algorithm, the cost function is used to determine the most accurate set of parameters which fits the training dataset. It provides a measure of the inaccuracy of the machine learning model, and the best possible parameters are determined by minimizing the cost function, using various methods. In linear regression tasks, the most commonly used cost function is:

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

The linear regression model attempts to optimize this cost function, i.e., it calculates the parameters that minimize it. Several methods are available for this optimization, such as batch gradient descent, stochastic gradient descent, mini-batch gradient descent, and the method of normal equations.

2.3 Gradient Descent

The method of gradient descent first assumes a totally random parameter vector, and then moves towards the optimal parameters step by step. During each such step, the gradient of the cost function at the current parameter vector is computed, and a multiple of the gradient is subtracted from the parameter vector, so as to always keep moving along the direction of fastest descent. Note that we have to be careful with this method, since gradient descent moves towards a local minimum, and may never reach the global minimum. The mathematical formulation for each step of gradient descent is as follows:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \alpha\begin{pmatrix}\frac{\partial}{\partial\theta_0}J\left(\boldsymbol{\theta}\right)\\ \frac{\partial}{\partial\theta_1}J\left(\boldsymbol{\theta}\right)\\ \vdots\\ \frac{\partial}{\partial\theta_n}J\left(\boldsymbol{\theta}\right)\end{pmatrix} = \boldsymbol{\theta} - \frac{\alpha}{m}\begin{pmatrix}\sum_{i=1}^{m}\left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)x_0^{(i)}\\ \sum_{i=1}^{m}\left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)x_1^{(i)}\\ \vdots\\ \sum_{i=1}^{m}\left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)x_n^{(i)}\end{pmatrix}$$

$$\implies \boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) \boldsymbol{x^{(i)}} = \boldsymbol{\theta} - \frac{\alpha}{m} \boldsymbol{X}^T \left(\boldsymbol{X\theta} - \boldsymbol{y} \right)$$

This formula is used for a method called batch gradient descent, where the entire dataset is considered in every single iteration, thus slowing down the training. To speed up the process, we can use stochastic gradient descent (one randomly selected instance in every iteration) or mini-batch gradient descent (a randomly selected subset of the dataset in every iteration).

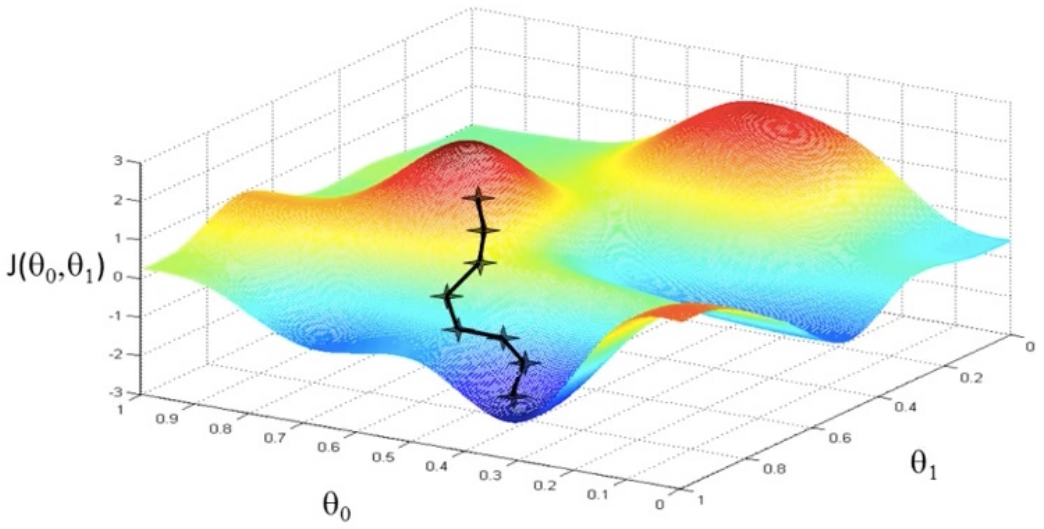

Figure 2: Movement towards lower cost function values using gradient descent. This figure shows a dataset consisting of only two features, with corresponding parameters θ_0 and θ_1.

2.4 Normal Equations

The normal equations method gives us a direct formula to calculate the optimized vector of parameters:

$$\boldsymbol{\theta} = \left(\boldsymbol{X}^T \boldsymbol{X} \right)^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

Although this method gives us a direct method to obtain the answer in just one step, there is one major problem; the computation of the inverse of $\boldsymbol{X}^T \boldsymbol{X}$. This matrix is a square matrix having order equal to the number of instances in the dataset, and the inverse calculation becomes computationally very heavy for larger datasets. The complexity of matrix inversion is $O(n^3)$, and this is a huge drawback for this method. Thus, gradient descent is preferred for datasets with more than $10,000$ data instances, while the normal equations method is used for smaller datasets.

2.5 Polynomial Regression

Although polynomial regression is different from linear regression, a polynomial regression task can be reduced to a linear regression task by simply adding more features, formed by multiplication of powers of the existing features (according to the degree of polynomial regression), and then applying ordinary linear regression. Some datasets are better fit by polynomial functions than linear functions, and thus, a higher degree polynomial is often a better solution for the problem.

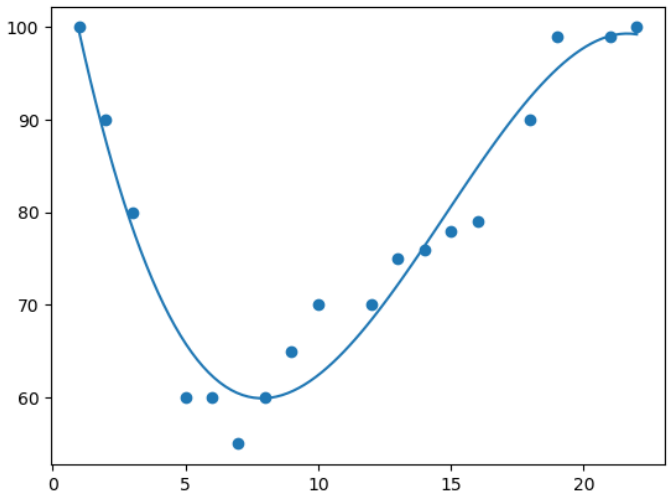

Figure 3: Polynomial regression

3 Logistic Regression

Logistic regression is (quite confusingly) a classification algorithm, under supervised learning. Given a training dataset along with the corresponding discrete valued labels, it determines a function to fit the dataset. The functioning of this algorithm is similar to that of linear regression. In a binary classification task, there is a scalar label which is either 0 or 1, whereas in multilabel classification, the label can be expressed as a vector where the index equal to the correct class number has value 1 and all others have value 0. We discuss binary classification first.

3.1 Hypothesis Function

The hypothesis function in this case forces the weighted sum of feature values into the interval $[0, 1]$. This number represents the predicted probability of that instance having a label of 1. The hypothesis function is usually written as:

$$h_\theta(\boldsymbol{x}) = g\left(\boldsymbol{\theta}^T \boldsymbol{x}\right)$$
$$\text{where} \quad g(z) = \frac{1}{1+e^{-z}}$$
$$\therefore \quad h_\theta(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{\theta}^T \boldsymbol{x}}}$$

Using this probability vector, we predict the label as:

$$y = \begin{cases} 1 & h_\theta(\boldsymbol{x}) \geq 0.5 \\ 0 & h_\theta(\boldsymbol{x}) < 0.5 \end{cases}$$

The above function $g(z)$ is called the sigmoid function. Also, the threshold here is 0.5, but in some special cases, where one out of precision or recall (later) is more important than the other, the threshold value can be changed to obtain better results.

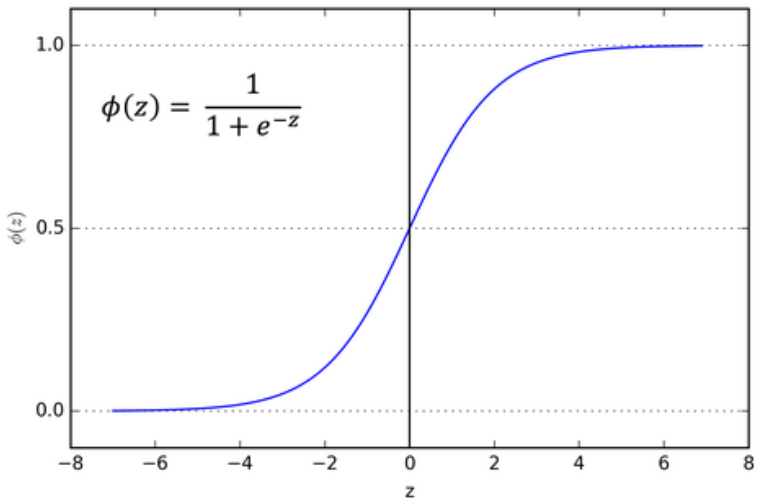

Figure 4: Sigmoid function

3.2 Decision Boundary

It is clear from the graph of the sigmoid function, that it is an increasing function, and it takes a value of 0.5 at $z = 0$. Thus, we predict label:

$$y = \begin{cases} 1 \text{ if } \boldsymbol{\theta}^T \boldsymbol{x} \geq 0 \\ 0 \text{ if } \boldsymbol{\theta}^T \boldsymbol{x} < 0 \end{cases}$$

This allows us to think of the classification in terms of a decision boundary. This boundary has the equation:

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n = 0$$

The region where the above quantity is positive is the label 1 region, whereas the remaining region is the label 0 region.

3.3 Cost Function

In logistic regression, the cross-entropy cost function is used:

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

It is clear that the above function adds larger errors for larger deviations in the probability from the true labels. As in linear regression, the cost function is minimized using gradient descent (batch/stochastic/mini-batch). There is no normal equations method here.

3.4 Gradient Descent

As in linear regression, gradient descent is implemented by subtracting a multiple of the gradient of the cost function from the parameter vector at each step:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \alpha \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} J(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\boldsymbol{\theta}) \end{pmatrix} = \boldsymbol{\theta} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta \left(\boldsymbol{x^{(i)}} \right) - y^{(i)} \right) \boldsymbol{x^{(i)}}$$

As can be seen, the gradient expression turns out to be somewhat similar to that in linear regression.

3.5 Multiclass Classification

For multiclass classification, the one-vs-rest algorithm is used. In this method, the probability that an instance belongs to class i is calculated for each class i, by creating several classifiers (number of classifiers equals the number of classes) and then training each classifier separately. For label prediction, the class label with the maximum probability is returned as the final prediction.

3.6 Skewed Classes

There occurs a somewhat funny problem when one of the two classification labels in the data is much less frequent than the other. Suppose we are trying to diagnose cancer in patients, and we achieve 99% classification accuracy (i.e., 1% error) with our model. However, suppose only 0.5% of patients actually have cancer. It is quite disheartening, then, to realize that a model which always predicts absence of cancer would have an accuracy of 99.5%, and would beat our model for this accuracy metric.

This emphasizes the need for a better error metric for skewed classes. This is where the concepts of precision and recall come in. Let $y = 1$ denote the rare class and $y = 0$ denote the common class. Consider the following table:

Actual Class / Predicted Class	$y = 1$	$y = 0$
$y = 1$	True Positives (TP)	False Positives (FP)
$y = 0$	False Negatives (FN)	True Negatives (TN)

We define:

$$\begin{aligned} \text{Precision} &= \text{Fraction of rare class predictions which are actually correct} \\ &= \frac{TP}{TP + FP} \\ \text{Recall} &= \text{Fraction of actual rare class instances which are classified correctly} \\ &= \frac{TP}{TP + FN} \end{aligned}$$

We can now evaluate our model using both the above values. To obtain a single evaluation metric, however, the F1 score, which is the harmonic mean of precision and recall, is often used:

$$\text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score is a much better evaluation metric than classification accuracy in case of skewed classes. Its range is 0 to 1. The higher the F1 score, the better is the model.

4 Overfitting and Underfitting

A machine learning model learns from patterns observed in the training dataset, and uses the training set observations to make predictions about other data which it hasn't seen before. We have to be careful of this distinction between the training and test datasets. The model only has access to patterns in the training data, and it has to make sure that it only learns patterns which will be present in new datasets as well, and not those patterns which are unique to the training dataset only. However, it cannot be made to learn too less from the training set, since that would make it incapable of accurate predictions.

The two problems described above are called overfitting and underfitting respectively:

1. **Overfitting**: The model becomes too closely acquainted with observations from the training dataset (often due to too many features), and is unable to generalize to new examples. Such a model is said to have high variance. Overfitting is characterized by very high prediction accuracy on the training set but much worse accuracy on test sets. Overfitting can often be due to very high degree polynomial regression, since this creates a lot of new features.

2. **Underfitting**: The model is unable to fit well to even the training data itself, and is thus incapable of making accurate predictions on the training set as well as test sets. Such a model is said to have high bias. Underfitting is characterized by low accuracies on training as well as test sets. Underfitting can often be due to very low degree polynomial regression, which does not represent the dataset well.

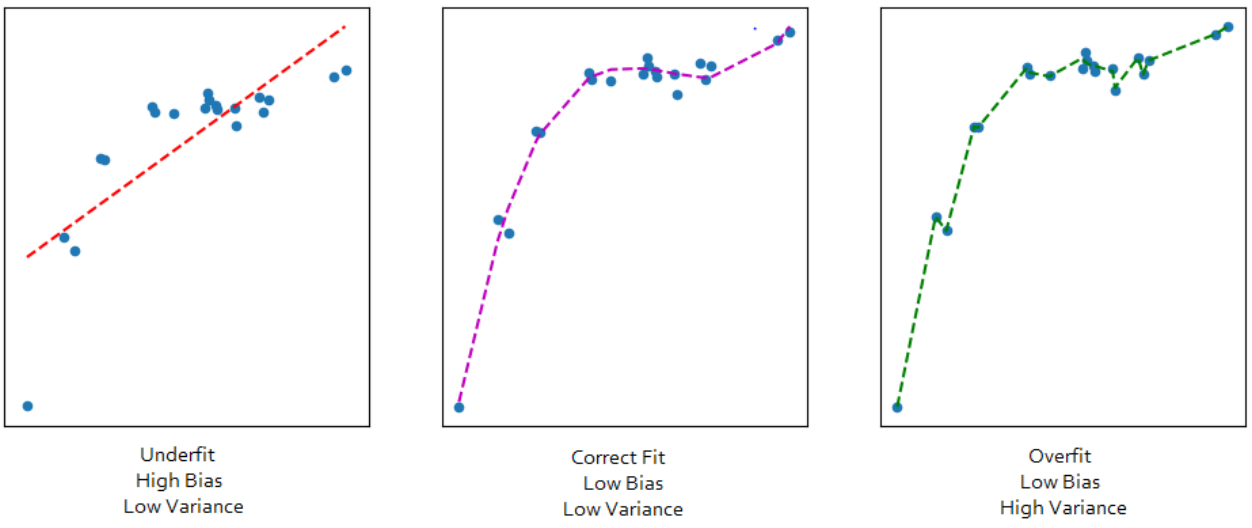

Figure 5: Overfitting and underfitting

4.1 Problem Diagnosis by Hypothesis Evaluation

The problems of overfitting and underfitting can be diagnosed by evaluating the hypothesis using a well defined measure of accuracy of the model.

First, we split the available labeled data into three sets; the training dataset, the cross-validation dataset, and the test dataset. The splitting ratio should be close to 60 : 20 : 20. The cross-validation set is used for tuning model parameters to generalize well to new data, and the test set is reserved for final evaluation of the model on new data.

Next, the model is trained using the training dataset, via gradient descent, normal equations, or any other method. The training set and cross-validation set prediction errors are then calculated using:

$$J_{train}\left(\boldsymbol{\theta}\right) = \frac{1}{2m_{train}} \sum_{i=1}^{m_{train}} \left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)^2$$

$$J_{cv}\left(\boldsymbol{\theta}\right) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} \left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)^2$$

A high bias (underfit) model is indicated by high values of both $J_{train}\left(\boldsymbol{\theta}\right)$ and $J_{cv}\left(\boldsymbol{\theta}\right)$ (and close to each other too), whereas a high variance (overfit) model is indicated by a low value of $J_{train}\left(\boldsymbol{\theta}\right)$ and a much higher value of $J_{cv}\left(\boldsymbol{\theta}\right)$.

4.2 Regularization: A Solution for Overfitting

One very commonly used solution for overfitting is regularization. Regularization works well when we have a large number of features.

In regularization, we keep all the features but we reduce the magnitudes of parameters θ_j. Mathematically, regularization is done by changing our cost function as follows:

$$J\left(\boldsymbol{\theta}\right) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

The value of λ has to be optimized as well, since it can cause underfitting if it is too high and overfitting if it is too low. Note that we do not penalize the bias parameter θ_0 during regularization.

Specifically, in the case of linear regression (and since polynomial regression can be implemented using linear regression, this applies to polynomial regression as well), the modifications caused by regularization are as follows:

The cost function becomes:

$$J\left(\boldsymbol{\theta}\right) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(\boldsymbol{\theta}^T\boldsymbol{x^{(i)}} - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

The gradient descent algorithm is changed as:

$$(\theta_0)_{new} = \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)$$

$$(\theta_j)_{new} = \theta_j - \frac{\alpha}{m}\left[\left(\sum_{i=1}^{m}\left(h_\theta\left(\boldsymbol{x^{(i)}}\right) - y^{(i)}\right)x_j^{(i)}\right) + \lambda\theta_j\right]$$

$$\forall\, j \in \{1, 2, \ldots n\}$$

The normal equations solution is modified as:

$$\boldsymbol{\theta} = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\begin{bmatrix}0 & 0 & \ldots & 0\\ 0 & 1 & \ldots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \ldots & 1\end{bmatrix}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Similarly, for logistic regression, the cost function becomes:

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h_\theta\left(\boldsymbol{x^{(i)}}\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_\theta\left(\boldsymbol{x^{(i)}}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

The gradient descent changes are similar to linear regression, and as before, there is no normal equations method for logistic regression.

An important point to note here is that the above changed cost function (in case of regularized learning) is used during the learning process only, and should not be used to evaluate the performance of the model. The model evaluation metric should be a measure of accuracy alone, and should thus be the same as before (i.e., the unregularized cost function).

4.3 Other Solutions

To get rid of overfitting or underfitting, the model has to have pretty good tuning of each of its parameters. These include the maximum degree of polynomial features d, the number of features n, the regularization parameter λ, the size of the training set m, etc.

To fine-tune any model parameter, we plot $J_{train}(\boldsymbol{\theta})$ and $J_{cv}(\boldsymbol{\theta})$ as functions of that parameter. The trend is as shown in the following plot:

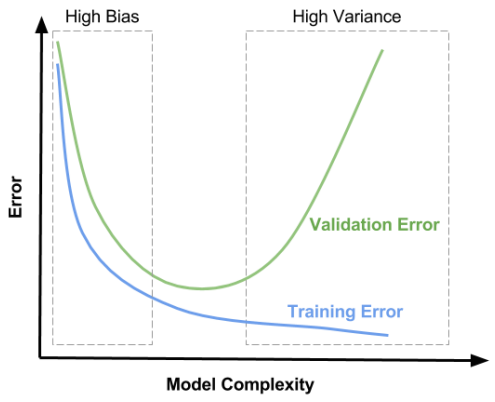

Figure 6: The bias-variance trade-off

We also know that:

- Model complexity increases with increase in d and n.
- Model complexity decreases with increase in λ.

Thus, by repeated training and evaluation of the model with changes in a single parameter, and by plotting the error versus parameter curve, we should find a minimum in the cross-validation error. The value of the parameter where this minimum is achieved is the optimum value of that parameter.

Another important factor to be considered is the training set size m. The curve for m is not the one shown above, since a larger training set is always better than a smaller one. However, we should still know whether a larger dataset would significantly improve our model's accuracy before we go searching for more data.

To figure this out, we plot the learning curve of the model. This is the graph of the training and cross-validation errors versus the training set size m. For any value of m, we train our model on a subset of our training data having size m, and we evaluate it on the smaller training set and the complete cross-validation set. We plot these errors, $J_{train}(\boldsymbol{\theta})$ and $J_{cv}(\boldsymbol{\theta})$, as functions of m. The learning curve can be of two types (if the model has a problem that needs fixing):

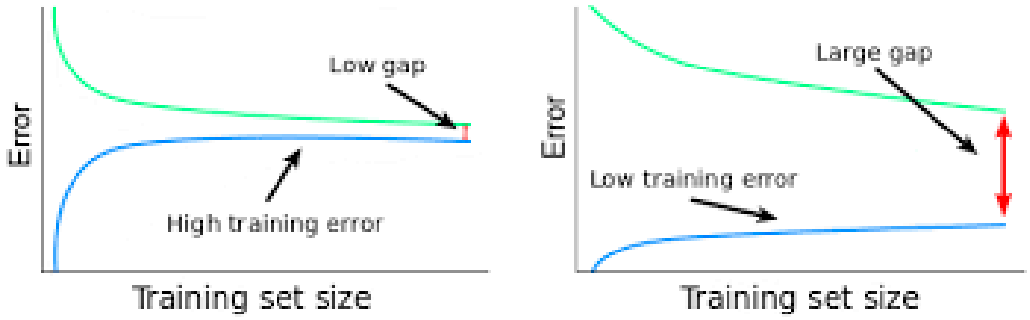

Figure 7: Learning curves for high bias and high variance models

In the first case (high bias, low variance), an increase in training set size wouldn't improve the accuracy much, whereas in the second case (low bias, high variance), an increase in training set size would improve accuracy quite significantly.

In a nutshell, debugging of learning algorithms should be performed only after diagnosing bias and variance issues. The steps to be taken to solve the problem are:

- For high bias (underfitting), try adding additional features and polynomial features, and decrease λ.
- For high variance (overfitting), get more training examples, try reducing features, and increase λ.

5 Support Vector Machines

A support vector machines (SVM) is a supervised learning model used for classification. It differs from logistic regression in that it exploits the geometrical properties of the data; unlike the

statistical approach of logistic regression, it tries to draw the best possible decision boundary for classification, and then uses this boundary for predictions.

5.1 Hypothesis Function

The hypothesis function for an SVM is the same as that for logistic regression:

$$h_\theta(\boldsymbol{x}) = g\left(\boldsymbol{\theta}^T\boldsymbol{x}\right) = \frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x}}}$$

5.2 Cost Function

The cost function is modified for an SVM. Here, we want a decision boundary which can distinguish between the two classes quite well, and we wouldn't like to accept a boundary that barely distinguishes data points. For $y=1$, we want $\boldsymbol{\theta}^T\boldsymbol{x} \gg 0$, and for $y=0$, we want $\boldsymbol{\theta}^T\boldsymbol{x} \ll 0$.

For regularized logistic regression, we had the following cost function:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\theta\left(\boldsymbol{x^{(i)}}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(\boldsymbol{x^{(i)}}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\left(-\log\frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}}}\right) + \left(1-y^{(i)}\right)\left(-\log\left(1-\frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}}}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \end{aligned}$$

For an SVM, we modify the cost function as follows:

$$J(\boldsymbol{\theta}) = C\sum_{i=1}^{m}\left[y^{(i)}cost_1\left(\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}\right) + \left(1-y^{(i)}\right)cost_0\left(\boldsymbol{\theta}^T\boldsymbol{x^{(i)}}\right)\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

Here, C represents the inverse of the regularization parameter λ. The two component cost functions above are:

$$cost_1(z) = \max\left(0, 1-\boldsymbol{\theta}^T\boldsymbol{x}\right)$$
$$cost_0(z) = \max\left(0, -1+\boldsymbol{\theta}^T\boldsymbol{x}\right)$$

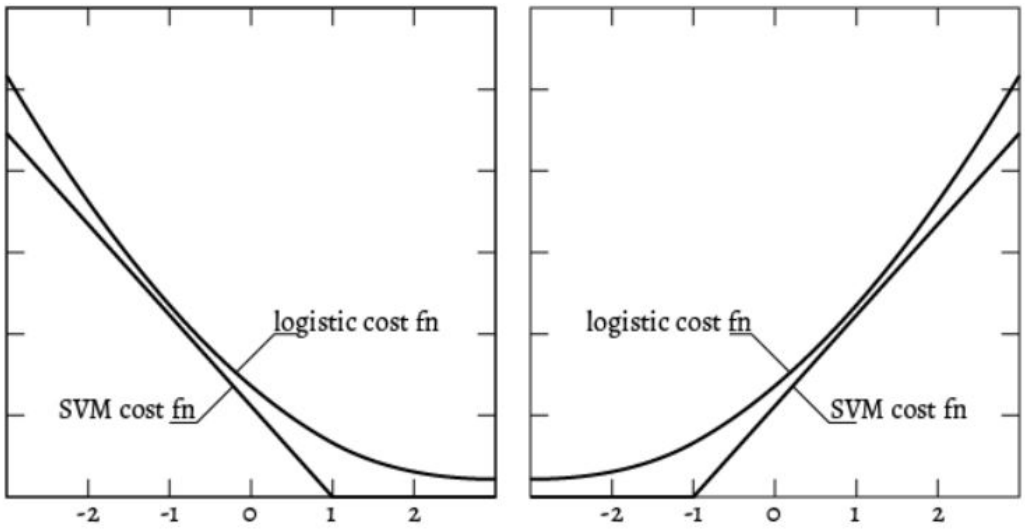

Figure 8: The two component cost functions of an SVM

5.3 Working of an SVM

If we ignore θ_0 and assume we are using a large value of C, the objective of an SVM can be expressed as follows:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & \frac{1}{2}\|\boldsymbol{\theta}\| \\ \text{s.t.} \quad & \boldsymbol{\theta}^T \boldsymbol{x}^{(i)} \geq 1 \qquad \text{if } y^{(i)} = 1 \\ \text{and} \quad & \boldsymbol{\theta}^T \boldsymbol{x}^{(i)} \leq -1 \qquad \text{if } y^{(i)} = 0 \end{aligned}$$

Now, $\boldsymbol{\theta}^T \boldsymbol{x}^{(i)}$ is the scalar product of the vectors $\boldsymbol{\theta}$ and $\boldsymbol{x}^{(i)}$, and is thus also the projection of $\boldsymbol{x}^{(i)}$ along $\boldsymbol{\theta}$, multiplied by $\|\boldsymbol{\theta}\|$. Thus, the optimization of the above expression would mean maximizing the projection magnitudes of $\boldsymbol{x}^{(i)}$ along $\boldsymbol{\theta}$, so that $\|\boldsymbol{\theta}\|$ can be minimized. Thus, the SVM algorithm finds a hyperplane in the dataset which can clearly distinguish between data point classes, by maximizing the distance of the hyperplane from the data points of both classes.

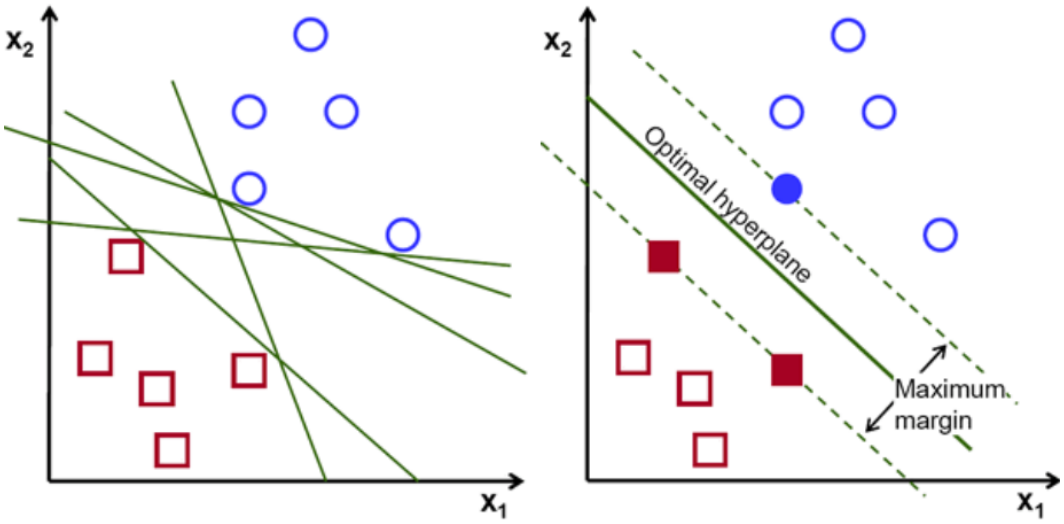

Figure 9: Hyperplane optimization by an SVM

5.4 SVMs with Kernels

Kernel functions are used with SVMs to make non-linear decision boundary formation much more effective. A kernel function takes a data point and a landmark as input and returns a new data point. Thus, a dataset can be transformed into another using a kernel function. The most commonly used kernel is the Gaussian kernel, which we will denote by $sim\,(\boldsymbol{x}, \boldsymbol{y})$, since it is a measure of similarity between the two points $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$sim\,(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{2\sigma^2}\right)$$

Given a dataset, let $\boldsymbol{l^{(1)} = x^{(1)}}$, $\boldsymbol{l^{(2)} = x^{(2)}}$, $\ldots$, $\boldsymbol{l^{(m)} = x^{(m)}}$ be landmarks.
For each training example $\left(\boldsymbol{x^{(i)}}, y^{(i)}\right)$:

$$\begin{aligned} f_0^{(i)} &= 1 \\ f_1^{(i)} &= sim\left(\boldsymbol{x^{(i)}}, \boldsymbol{l^{(1)}}\right) \\ f_2^{(i)} &= sim\left(\boldsymbol{x^{(i)}}, \boldsymbol{l^{(2)}}\right) \\ &\vdots \\ f_m^{(i)} &= sim\left(\boldsymbol{x^{(i)}}, \boldsymbol{l^{(m)}}\right) \end{aligned}$$

Thus, we obtain a new dataset F of size $m \times (m+1)$, and we use this to train the model.
Our new hypothesis predicts $y^{(i)} = 1$ if $\boldsymbol{\theta^T f^{(i)}} \geq 0$, and $y^{(i)} = 0$ otherwise.
During training too, the cost function should be calculated by a modified SVM cost function, with $\boldsymbol{\theta^T x^{(i)}}$ replaced by $\boldsymbol{\theta^T f^{(i)}}$.

Note that:

- Large C and small σ can cause low bias and high variance.
- Small C and large σ can cause high bias and low variance.

6 K-Means Clustering

K-means clustering is a clustering algorithm under unsupervised learning. It is an iterative algorithm used to group a dataset into K clusters.

6.1 Algorithm

The algorithm works as follows:

1. Randomly initialize K cluster centroids, $\boldsymbol{\mu_1}$, $\boldsymbol{\mu_2}$, $\ldots$, $\boldsymbol{\mu_K} \in \mathbb{R}^n$.
2. $\forall\, i \in \{1, \ldots, m\}$, $c^{(i)}$:= index (from 1 to K) of the cluster centroid closest to $\boldsymbol{x^{(i)}}$.
3. $\forall\, k \in \{1, \ldots, K\}$, $\boldsymbol{\mu_k}$:= centroid (mean) of points assigned to cluster k.
4. Repeat steps 2 and 3 above until the cluster centroids stop moving.

6.2 Optimization Objective

The optimization objective for K-means clustering can be expressed as follows:

$$\min_{c^{(1)}, \ldots, c^{(m)}, \boldsymbol{\mu_1}, \ldots, \boldsymbol{\mu_K}} J\left(c^{(1)}, \ldots, c^{(m)}, \boldsymbol{\mu_1}, \ldots, \boldsymbol{\mu_K}\right)$$

$$\text{where } J\left(c^{(1)}, \ldots, c^{(m)}, \boldsymbol{\mu_1}, \ldots, \boldsymbol{\mu_K}\right) = \frac{1}{m} \sum_{i=1}^{m} \|\boldsymbol{x^{(i)}} - \mu_{c^{(i)}}\|^2$$

The above cost function is also called the *distortion* of the dataset.

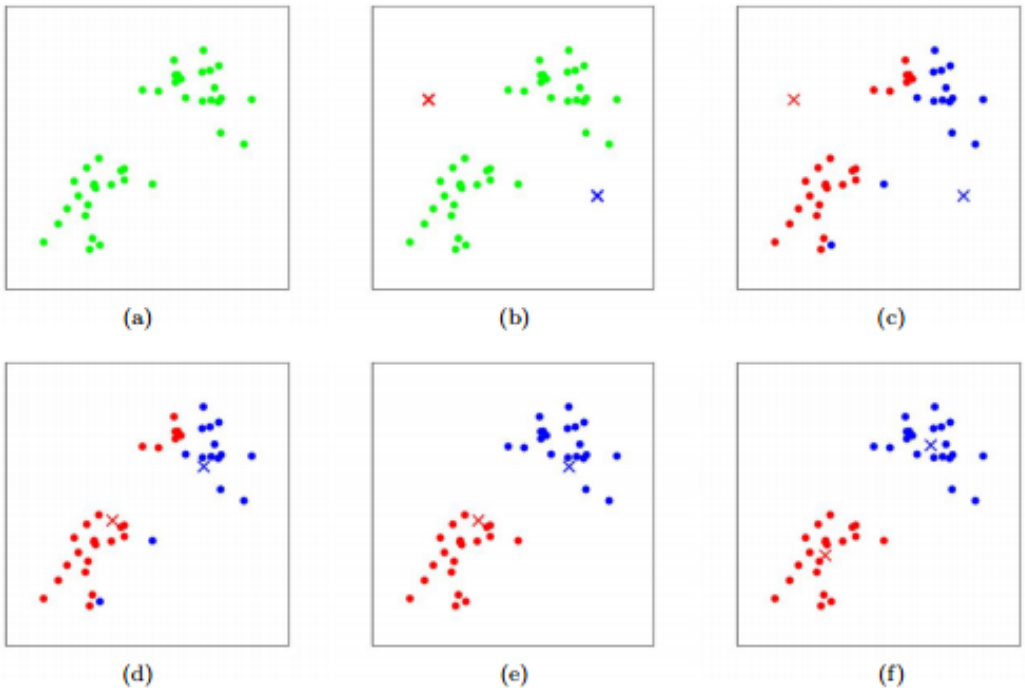

Figure 10: K-means clustering algorithm

6.3 Random Initialization and Selecting K

The initial cluster centroids should be well spread-out throughout the dataset. To ensure this, the clustering algorithm is performed several times, each time randomly initializing the cluster centroids to be K points from the dataset. The iteration which returns the minimum value of $J\left(c^{(1)}, \ldots, c^{(m)}, \boldsymbol{\mu_1}, \ldots, \boldsymbol{\mu_K}\right)$ is considered to have the right clusters, and these clusters are finally chosen by the algorithm.

As for selecting the number of clusters K for grouping the dataset, this is most often done manually, according to the programmer's requirements, or the nature of the dataset. If unsure about the number of clusters in the data, one can try using the elbow method, although this method does not always work. In this method, the curve of the distortion function (cost function) $J\left(c^{(1)}, \ldots, c^{(m)}, \boldsymbol{\mu_1}, \ldots, \boldsymbol{\mu_K}\right)$ is plotted against the number of clusters K. This graph sometimes has an elbow, a point where the slope of the curve suddenly decreases. The value of K at this elbow is an ideal selection for K.

7 Anomaly Detection

Unsupervised learning can be used for anomaly detection in data. Anomalies are data points for which one or more feature values lie far away from those of the majority of data instances. Thus, for anomaly detection, we have to come up with a probability function for data points, and we predict an anomaly whenever the probability falls below a threshold value ϵ.

7.1 Algorithm

The following algorithm works on the basis that the feature values distribution should be approximately Gaussian in nature. The algorithm works as follows:

1. Choose features $\boldsymbol{x_i}$ that could be indicative of anomalous instances.
2. Calculate parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$.

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} \left(x_j^{(i)} \right)^2$$

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} \left(x_j^{(i)} - \mu_j \right)^2$$

3. For anomaly detection on a new example $\boldsymbol{x}$, calculate:

$$p\left(\boldsymbol{x}\right) = \prod_{j=1}^{n} p\left(x_j; \mu_j, \sigma_j^2\right) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{\left(x_j - \mu_j\right)^2}{2\sigma_j^2} \right)$$

   Predict an anomaly if $p(\boldsymbol{x}) < \epsilon$ for some predefined threshold ϵ.

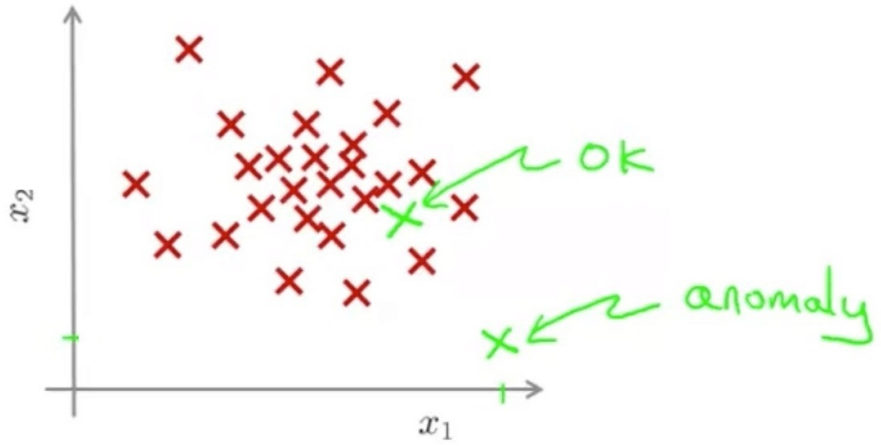

Figure 11: An anomaly in a dataset

7.2 Model Evaluation

An anomaly detection model should not be evaluated using cross-entropy error or prediction accuracy, because it consists of skewed classes. If we have a test dataset with the anomalies already labelled, then we can use it to evaluate our model through metrics like precision, recall, and the F1-score. This is because the anomalous examples are much rarer than the non-anomalous ones, i.e., the problem has skewed classes.

7.3 Multivariate Gaussian Distribution

Instead of expressing the probability function as a product of individual feature value probabilities, we could express it as a single multivariate Gaussian probability distribution representing the

entire dataset:

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

where $\boldsymbol{\mu}$ is, as before, the vector of mean feature values (calculated and stored during training), and $\boldsymbol{\Sigma}$ is the $(n \times n)$ covariance matrix of the mean normalized training dataset:

$$\boldsymbol{\Sigma} = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{x^{(i)}}-\boldsymbol{\mu}\right)\left(\boldsymbol{x^{(i)}}-\boldsymbol{\mu}\right)^T$$

Note that the previous probability distribution function (product form) is a special case of the multivariate Gaussian function, with off-diagonal elements of the covariance matrix equal to zero.

The multivariate function is advantageous in that it can determine correlations between features by itself for anomaly detection, but it is computationally expensive due to the inverse calculation. Also, to use the multivariate function, we must have $m > n$ to ensure invertibility of $\boldsymbol{\Sigma}$.

8 Recommender Systems

Recommender systems are programs that explore a user's product ratings, and learn to recommend new products to that user in accordance with his/her preferences. Without loss of generality, we consider the problem of movie recommendations.

Suppose we have n_u users and n_m movies. Users watch movies, and rate them after watching. Suppose we have a matrix $\boldsymbol{R} = [r_{ij}]$, where $r_{ij} = 1$ if user j has rated movie i and 0 otherwise, and a ratings matrix $\boldsymbol{Y} = [y_{ij}]$, where y_{ij} is the rating (from 1 to 5) given by user j to movie i, defined only if $r_{ij} = 1$. The way we go about recommending movies to users is by predicting the undefined ratings in matrix $\boldsymbol{Y}$, i.e., we guess how users would have rated the movies they haven't watched yet, and then we recommend the movies which get the highest predicted ratings.

8.1 Content Based Recommendations

The problem is formulated as follows:

$r(i,j) = 1$ if user j has rated movie i (otherwise 0)

$y^{(i,j)}$ = rating by user j on movie i, if defined

$\boldsymbol{\theta^{(j)}}$ = parameter vector for user j

$\boldsymbol{x^{(i)}}$ = feature vector for movie i

Predicted rate for user j, movie $i = \left(\boldsymbol{\theta^{(j)}}\right)^T \boldsymbol{x^{(i)}}$

Thus, the overall optimization objective (for all users) can be expressed as:

$$\min_{\boldsymbol{\theta^{(1)}},\boldsymbol{\theta^{(2)}},\ldots,\boldsymbol{\theta^{(n_u)}}} \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left(\left(\boldsymbol{\theta^{(j)}}\right)^T \boldsymbol{x^{(i)}} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\theta_k^{(j)}\right)^2$$

We can then calculate each user's own parameter vector by minimizing this loss function, and then use these parameters along with the movie features to give predictions.

There is one disadvantage with this recommendation method, though. To implement it, we need to set the features (usually the genre distribution) of all the movies manually beforehand. To overcome this problem, the method of collaborative filtering is used.

8.2 Collaborative Filtering

Given all $\boldsymbol{x^{(i)}}$, we can learn all $\boldsymbol{\theta^{(j)}}$ as before:

$$\min_{\boldsymbol{\theta^{(1)}},\boldsymbol{\theta^{(2)}},\ldots,\boldsymbol{\theta^{(n_u)}}} \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left(\left(\boldsymbol{\theta^{(j)}}\right)^T\boldsymbol{x^{(i)}}-y^{(i,j)}\right)^2+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\theta_k^{(j)}\right)^2$$

Similarly, given all $\boldsymbol{\theta^{(j)}}$, we can learn all $\boldsymbol{x^{(i)}}$ as follows:

$$\min_{\boldsymbol{x^{(1)}},\boldsymbol{x^{(2)}},\ldots,\boldsymbol{x^{(n_m)}}} \frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}\left(\left(\boldsymbol{\theta^{(j)}}\right)^T\boldsymbol{x^{(i)}}-y^{(i,j)}\right)^2+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$

In collaborative filtering, we combine both the above into a single optimization objective:

$$\min_{\boldsymbol{\theta^{(1)}},\ldots,\boldsymbol{\theta^{(n_u)}},\boldsymbol{x^{(1)}},\ldots,\boldsymbol{x^{(n_m)}}} J\left(\boldsymbol{\theta^{(1)}},\ldots,\boldsymbol{\theta^{(n_u)}},\boldsymbol{x^{(1)}},\ldots,\boldsymbol{x^{(n_m)}}\right)$$

where:

$$\begin{aligned}J\left(\boldsymbol{\theta^{(1)}},\ldots,\boldsymbol{\theta^{(n_u)}},\boldsymbol{x^{(1)}},\ldots,\boldsymbol{x^{(n_m)}}\right) = \frac{1}{2}\sum_{(i,j):r(i,j)=1}&\left(\left(\boldsymbol{\theta^{(j)}}\right)^T\boldsymbol{x^{(i)}}-y^{(i,j)}\right)^2\\ &+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\theta_k^{(j)}\right)^2\end{aligned}$$

As always, we can now initialize all $\boldsymbol{x^{(i)}}$s and $\boldsymbol{\theta^{(j)}}$s to random values and optimize them through gradient descent.

There is one problem with this approach. Suppose a user hasn't watched or rated any movies yet. Then, the cost function's first term becomes zero for that user, and the minimization of the second regularization term means that all the user's parameters will be predicted as zero. So, we wouldn't be able to make recommendations. Rather than this, we would practically prefer to recommend movies with high average ratings to that user. To do this, we perform mean normalization of the ratings for each individual movie before running our algorithm; for each movie, we make the mean of our user ratings zero (and store the actual mean). That way, after predicting the rating of a user for a movie, we add back the mean rating before reporting the predicted rating. This solves the new user problem.

Part II

Neural Networks and Deep Learning

9 Neural Networks

Although we could use linear and logistic regression algorithms and their variants, or SVMs, to solve machine learning problems quite efficiently, there exist some other learning methods which are much more effective for generating non-linear hypotheses. One such method is neural networks.

Neural networks originated when computer scientists tried to mimic the learning processes of the human brain, hence the name. In fact, there is not much in common between the neural networks discussed here and those that constitute our brain. However, the essential notion of neurons receiving several inputs and passing on a single output to the next neuron is preserved in the neural networks which we discuss here.

9.1 Model Representation

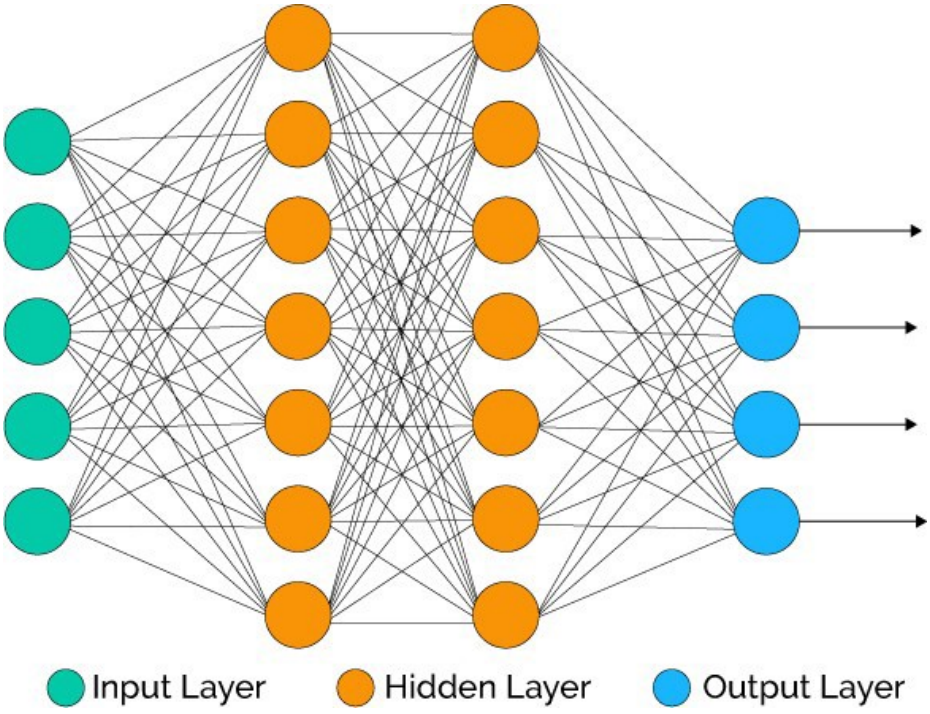

Figure 12: A deep neural network

The above diagram shows a 3-layer neural network (input layer is not really a layer). It has one input layer, two hidden layers, and one output layer. A general neural network has one input layer (with number of input units equal to the number of features available in the dataset), any number of hidden layers (with any number of units each), and an output layer (with number of output units equal to number of values to be predicted; e.g., for a 4-class classification, this number would be 4).

We represent the number of units by L. Here, $L = 3$. We number the layers from 0 to L; the input layer is called layer 0, hidden layers are numbered from 1 to $L - 1$ (left to right), and the output layer is called layer L.

For the layer numbered l, we denote the number of units in that layer by $n^{[l]}$. In the above diagram, we have $n^{[0]} = 5$, $n^{[1]} = 7$, $n^{[2]} = 7$, and $n^{[3]} = 4$.

Each neuron takes input from the previous layer and produces an output, called its activation. We represent the activation of the k^{th} neuron in layer l by $a_k^{[l]}$. The activations of layer l, when stacked in a column vector, form its activations vector, denoted by $\boldsymbol{a^{[l]}}$. Thus, $\boldsymbol{a^{[0]}}$ is the input features vector (for the data instance being considered), and $\boldsymbol{a^{[L]}}$ is the output of the neural network.

9.2 Forward Propagation

So, how does the above network take our input and generate an output? Well, each layer in the neural network essentially performs two functions:

1. Every unit (neuron) in layer l of the neural network first calculates a linear combination of all the values (outputs) from the previous layer $l-1$. For each neuron in layer l, this linear combination is calculated by using a different vector of weights for the values from layer $l-1$. When these vectors are stacked as rows of a matrix, we get a weight matrix from layer $l-1$ to layer l. This weights matrix is denoted by $\boldsymbol{W^{[l]}}$, and the shape of this matrix is $n^{[l]} \times n^{[l-1]}$. The k^{th} row of the matrix $\boldsymbol{W^{[l]}}$ is the weights vector for calculations for the neuron $a_k^{[l]}$. In addition to these weights, each neuron in each layer also has a bias unit $b_k^{[l]}$ associated with it. Thus, the $n^{[l]} \times 1$ column vector $\boldsymbol{b^{[l]}}$ is the bias vector for layer l calculation. Mathematically, the first calculation for each layer is:

$$\boldsymbol{z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}}$$

2. Secondly, each neuron in the layer applies an activation function (same function for all neurons in the same layer) to the previously computed linear combination $z_k^{[l]}$. This is necessary for the network to be able to generate non-linear hypotheses, and an extremely effective way of doing it too. The activation function is generally taken to be the ReLU (Rectified Linear Unit) function for hidden layers, the sigmoid function for the output layer if it is a classification task (to be able to generate a probability output between 0 and 1), and the identity function ($g(z) = z$) for the output layer if it is a regression task (to be able to generate any real number as an output).

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

$$\text{Sigmoid function: } g(z) = \frac{1}{1+e^{-z}}$$

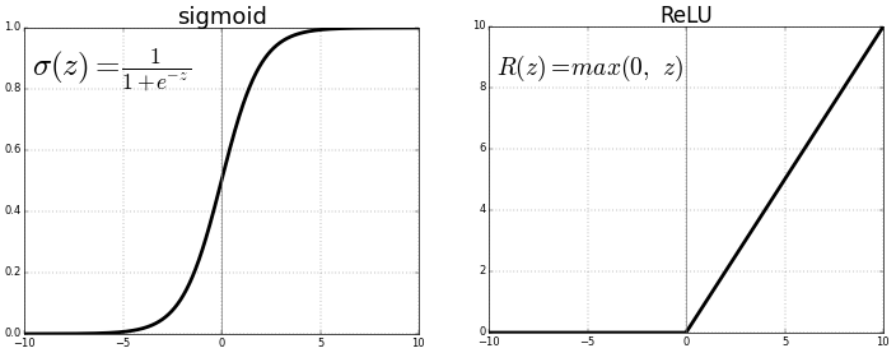

Figure 13: Sigmoid and ReLU functions

Thus, we obtain the activations of layer l as:

$$\boldsymbol{a^{[l]} = g\left(z^{[l]}\right)}$$

In the above discussion, we have till now considered manipulating only a single data instance. However, we can further vectorize the calculations to use the entire dataset together.

Let $\boldsymbol{A}^{[l]}$ be the matrix of activations for layer l. In this matrix, the i^{th} column contains the activations for the i^{th} data instance. In accordance with this convention, we now have to send in our input dataset with a shape of $n_x \times m$ (unlike we used to do in machine learning); i.e., each input data instance is contained in a column. Thus, the above two calculations are changed to:

$$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]}$$
$$\boldsymbol{A}^{[l]} = g\left(\boldsymbol{Z}^{[l]}\right)$$

It is very useful in debugging to keep checking shapes of matrices. Here, $\boldsymbol{Z}^{[l]}$ and $\boldsymbol{A}^{[l]}$ both have shape $n^{[l]} \times m$, $\boldsymbol{W}^{[l]}$ has shape $n^{[l]} \times n^{[l-1]}$, and $\boldsymbol{b}^{[l]}$ has shape $n^{[l]} \times 1$. Also note that the bias vector $\boldsymbol{b}^{[l]}$ is added separately to each column of the matrix $\boldsymbol{W}^{[l]}\boldsymbol{A}^{[l-1]}$; the above written equation assumes broadcasting to be understood.

We have now discussed how a layer calculates its outputs using outputs from the previous layer along with weights and biases. Looking at the big picture, we first assign our input dataset to the matrix $\boldsymbol{A}^{[0]}$, and using the above two equations, with the weights, biases, and activation function of each layer, the neural network propagates the data forward and finally generates an output in the fom of $\boldsymbol{A}^{[L]}$. This is called forward propagation. If this is the training phase, the output of forward propagation is checked for accuracy and then gradient descent is used to improve the weights and biases of the network. If training is over, and the parameters have been optimized, then this final output is our prediction.

9.3 Network Training and Backward Propagation

How do we train a neural network? We have a training dataset $\boldsymbol{X}$, and we have a network model like the figure shown previously, but how do we optimize the parameters for maximum prediction accuracy? The following steps are used:

1. We first initialize all the network parameters (weights and biases) to random values (but keep them small for faster gradient descent learning). It is important to know that we cannot initialize everything to zero, because this would generate symmetry between all neurons, and would not work at all. Generally, we initialize weights to small random values ($\sim 10^{-2}$) and biases to zero. We also set $\boldsymbol{A}^{[0]}$ to be the training dataset $\boldsymbol{X}$.

2. We then use forward propagation to move through the neural network and finally generate some outputs in layer L. During forward propagation, the outputs of each layer are stored in a cache, to be used during gradient descent. The final outputs are predictions, and using a cost function (such as mean squared error, or binary/categorical cross-entropy), we calculate the error in our predictions.

3. We use backward propagation to calculate the gradients of the cost function with respect to the model parameters. We use dp to denote the derivative of the cost function with respect

to a parameter p. The calculations are as follows:

$$\begin{aligned}
\boldsymbol{dZ^{[L]}} &= \frac{\partial J}{\partial \boldsymbol{Z^{[L]}}} && \text{(depends on cost function)} \\
&= \boldsymbol{A^{[L]}} - \boldsymbol{Y} && \text{(for the cross-entropy cost function)} \\
\boldsymbol{dW^{[L]}} &= \frac{1}{m}\boldsymbol{dZ^{[L]}} \left(\boldsymbol{A^{[L-1]}}\right)^T \\
\boldsymbol{db^{[L]}} &= \frac{1}{m}\boldsymbol{dZ^{[L]}} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} && \text{(avg of all columns of the matrix)} \\
&= np.sum(\boldsymbol{dZ^{[L]}}, axis = 1, keepdims = True)/m && \text{(in NumPy)} \\
\boldsymbol{dZ^{[L-1]}} &= \left(\boldsymbol{W^{[L]}}\right)^T \boldsymbol{dZ^{[L]}} * \left(g^{[L-1]}\right)' \left(\boldsymbol{Z^{[L-1]}}\right)
\end{aligned}$$

and so on until we reach $\boldsymbol{dW^{[1]}}$ and $\boldsymbol{db^{[1]}}$.

4. Now that we have the gradients, we perform gradient descent as before. Given a learning rate α, we reassign (for l from 1 to L):

$$\begin{aligned}
\boldsymbol{W^{[l]}} &:= \boldsymbol{W^{[l]}} - \alpha \boldsymbol{dW^{[l]}} \\
\boldsymbol{b^{[l]}} &:= \boldsymbol{b^{[l]}} - \alpha \boldsymbol{db^{[l]}}
\end{aligned}$$

5. Thus, we have achieved one iteration of gradient descent. We perform steps 2-4 as many times as the pre-decided number of iterations, and finally obtain the optimized weights and biases for the neural network.

Once we've trained the network, our model is ready for making predictions! Its accuracy can be checked by applying the same cost function (which was used for training) on its predictions.

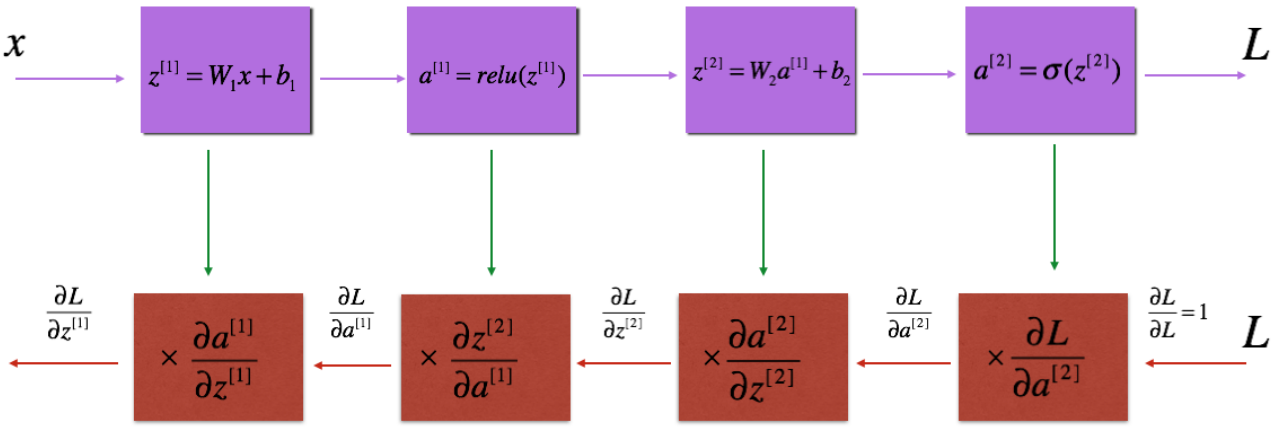

Figure 14: The process of training a neural network

9.4 Mathematics of the Backpropagation Algorithm

In the previous section, we saw some gradient formulae, for implementing backpropagation algorithm. In this section, I attempt to derive those formulae by using the formulae already defined for forward propagation.

Here, we consider only one training instance. The derivation can be easily extended to cover m instances; weights and biases gradients just have to be averaged over all the training examples. Also, to make the neural network general, we won't impose restrictions on $n^{[L]}$; it may be 1 or greater. However, we will assume that the last activation is sigmoid. From forward propagation, we have for $l \in \{1, 2, \ldots, L\}$:

$$\begin{aligned}
\boldsymbol{a^{[0]}} &= \boldsymbol{x} \\
\boldsymbol{z^{[l]}} &= \boldsymbol{W^{[l]}} \boldsymbol{a^{[l-1]}} + \boldsymbol{b^{[l]}} \\
\boldsymbol{a^{[l]}} &= g^{[l]} \left(\boldsymbol{z^{[l]}} \right)
\end{aligned}$$

At the end of forward propagation, we have our loss function:

$$\boldsymbol{J} = -\sum_{k=1}^{n^{[L]}} \left[y_k \log \left(a_k^{[L]} \right) + (1 - y_k) \log \left(1 - a_k^{[L]} \right) \right]$$

We first calculate the derivative of $\boldsymbol{J}$ w.r.t. $\boldsymbol{z^{[L]}}$, denoted by $\boldsymbol{dz^{[L]}}$:

$$\boldsymbol{dz^{[L]}} = \frac{\partial \boldsymbol{J}}{\partial \boldsymbol{z^{[L]}}} = \begin{bmatrix} \dfrac{\partial \boldsymbol{J}}{\partial z_1^{[L]}} & \dfrac{\partial \boldsymbol{J}}{\partial z_2^{[L]}} & \cdots & \dfrac{\partial \boldsymbol{J}}{\partial z_{n^{[L]}}^{[L]}} \end{bmatrix}^T$$

Also:

$$\begin{aligned}
\frac{\partial \boldsymbol{J}}{\partial z_k^{[L]}} &= \frac{\partial \boldsymbol{J}}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_k^{[L]}} \\
&= \left(\frac{1 - y_k}{1 - a_k^{[L]}} - \frac{y_k}{a_k^{[L]}} \right) \sigma' \left(z_k^{[L]} \right) \\
&= \frac{a_k^{[L]} - y_k}{a_k^{[L]} \left(1 - a_k^{[L]} \right)} \cdot a_k^{[L]} \left(1 - a_k^{[L]} \right) \\
&= a_k^{[L]} - y_k
\end{aligned}$$

Therefore:

$$\boldsymbol{dz^{[L]}} = \boldsymbol{a^{[L]}} - \boldsymbol{y} \tag{1}$$

Thus, we now have $\boldsymbol{dz^{[L]}}$, the gradient for the last layer. Next, we will see how, given $\boldsymbol{dz^{[l]}}$ and parameters, we can compute $\boldsymbol{dW^{[l]}}$, $\boldsymbol{db^{[l]}}$, and $\boldsymbol{dz^{[l-1]}}$, for any general $l \in \{1, 2, \ldots, L\}$. We now calculate the derivative of $\boldsymbol{J}$ w.r.t. $\boldsymbol{W^{[l]}}$, denoted by $\boldsymbol{dW^{[l]}}$:

$$\boldsymbol{dW^{[l]}} = \frac{\partial \boldsymbol{J}}{\partial \boldsymbol{W^{[l]}}} = \begin{bmatrix} \dfrac{\partial \boldsymbol{J}}{\partial W_{11}^{[l]}} & \dfrac{\partial \boldsymbol{J}}{\partial W_{12}^{[l]}} & \cdots & \dfrac{\partial \boldsymbol{J}}{\partial W_{1n^{[l-1]}}^{[l]}} \\ \dfrac{\partial \boldsymbol{J}}{\partial W_{21}^{[l]}} & \dfrac{\partial \boldsymbol{J}}{\partial W_{22}^{[l]}} & \cdots & \dfrac{\partial \boldsymbol{J}}{\partial W_{2n^{[l-1]}}^{[l]}} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial \boldsymbol{J}}{\partial W_{n^{[l]}1}^{[l]}} & \dfrac{\partial \boldsymbol{J}}{\partial W_{n^{[l]}2}^{[l]}} & \cdots & \dfrac{\partial \boldsymbol{J}}{\partial W_{n^{[l]}n^{[l-1]}}^{[l]}} \end{bmatrix}$$

Also:

$$\begin{aligned}\frac{\partial \boldsymbol{J}}{\partial W_{jk}^{[l]}} &= \frac{\partial \boldsymbol{J}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial W_{jk}^{[l]}} \\ &= \left(dz_j^{[l]}\right) \cdot \left(a_k^{[l-1]}\right)\end{aligned}$$

Therefore:

$$\boldsymbol{dW^{[l]}} = \boldsymbol{dz^{[l]}}(\boldsymbol{a^{[l-1]}})^T \tag{2}$$

Next, we compute the derivative of $\boldsymbol{J}$ w.r.t. $\boldsymbol{b^{[l]}}$, denoted by $\boldsymbol{db^{[l]}}$:

$$\boldsymbol{db^{[l]}} = \frac{\partial \boldsymbol{J}}{\partial \boldsymbol{b^{[l]}}} = \begin{bmatrix} \frac{\partial \boldsymbol{J}}{\partial b_1^{[l]}} & \frac{\partial \boldsymbol{J}}{\partial b_2^{[l]}} & \cdots & \frac{\partial \boldsymbol{J}}{\partial b_{n^{[l]}}^{[l]}} \end{bmatrix}^T$$

Also:

$$\begin{aligned}\frac{\partial \boldsymbol{J}}{\partial b_k^{[l]}} &= \frac{\partial \boldsymbol{J}}{\partial z_k^{[l]}} \cdot \frac{\partial z_k^{[l]}}{\partial b_k^{[l]}} \\ &= \left(dz_j^{[l]}\right) \cdot 1\end{aligned}$$

Therefore:

$$\boldsymbol{db^{[l]}} = \boldsymbol{dz^{[l]}} \tag{3}$$

Next, we compute the derivative of $\boldsymbol{J}$ w.r.t. $\boldsymbol{z^{[l-1]}}$, denoted by $\boldsymbol{dz^{[l-1]}}$:

$$\boldsymbol{dz^{[l-1]}} = \frac{\partial \boldsymbol{J}}{\partial \boldsymbol{z^{[l-1]}}} = \begin{bmatrix} \frac{\partial \boldsymbol{J}}{\partial z_1^{[l-1]}} & \frac{\partial \boldsymbol{J}}{\partial z_2^{[l-1]}} & \cdots & \frac{\partial \boldsymbol{J}}{\partial z_{n^{[l-1]}}^{[l-1]}} \end{bmatrix}^T$$

Also:

$$\begin{aligned}\frac{\partial \boldsymbol{J}}{\partial z_k^{[l-1]}} &= \sum_{j=1}^{n^{[L]}} \frac{\partial \boldsymbol{J}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial a_k^{[l-1]}} \cdot \frac{\partial a_k^{[l-1]}}{\partial z_k^{[l-1]}} \\ &= \sum_{j=1}^{n^{[L]}} \left(dz_j^{[l]}\right) \cdot \left(W_{jk}^{[l]}\right) \cdot \left(\left(g^{[l-1]}\right)' \left(z_k^{[l-1]}\right)\right)\end{aligned}$$

Therefore:

$$\boldsymbol{dz^{[l-1]}} = \left(\boldsymbol{W^{[l]}}\right)^T \boldsymbol{dz^{[l]}} * \left(g^{[l-1]}\right)' \left(\boldsymbol{z^{[l-1]}}\right) \tag{4}$$

Here, $*$ stands for element-wise multiplication of matrices.
Hence, equation (1) acts as an induction base, and with equations (2), (3), and (4), we can say by induction that we can calculate the gradient of $\boldsymbol{J}$ w.r.t. all weights and biases of the entire neural network. We can then subtract these gradients (multiplied by a learning rate) from the respective parameters to make $\boldsymbol{J}$ smaller and smaller, which is essentially what we discussed previously.

Part III

Project Details

10 Updated Plan of Action

Following is my updated plan of action for SoS 2020:

Week 1 (May 2 - May 8)	Hyperparameter tuning, regularization, optimization of neural networks, structuring machine learning projects.
Week 2 (May 9 - May 15)	Convolutional neural networks, sequence models, image classification, loss functions and optimization, training neural networks for CV.
Week 3 (May 16 - May 22)	Deep learning software, CNN architectures, recurrent neural networks, detection and segmentation, visualizing and understanding features.
Week 4 (May 23 - May 29)	Generative models, reinforcement learning, deep reinforcement learning.
Week 5 (May 30 - Jun 5)	Efficient methods and hardware for deep learning, adversarial examples and adversarial training.
Week 6 (Jun 6 - Jun 10)	Endterm report writing and final submission.

11 Resources

1. Andrew Ng's Machine Learning Course on Coursera
2. Andrew Ng's Deep Learning Specialization on Coursera
3. Stanford University's CS231n Course Material
4. 'Deep Learning' by MIT Press
5. 'Pattern Recognition and Machine Learning' by Christopher Bishop