Introduction to Machine Learning

Dan Sehayek

November 25, 2018

Contents

1 Introduction

Welcome to my introduction to machine learning! Over the past year I've been taking the time to learn about machine learning and how to apply it. This introduction is essentially a summary of my notes on machine learning from a few different amazing resources that I was very fortunate to come across.

- Andrew Ng's Coursera Course on Machine Learning
- Kevin Markham's YouTube Playlist on Machine Learning with Python
- Grant Sanderson's YouTube Playlist on Neural Networks

All of the people above did an incredible job of explaining some of the challenging concepts of machine learning and I would like to thank these people for giving me the confidence and motivation to study the subject further. I would also encourage anybody reading this to go check them out!

As far as these notes go, they were originally meant to help summarize and review the content that I've learned. But if you also have acquired a sudden interest in the subject and would like to learn more about machine learning, please feel to read this!

2 Learning Algorithms

2.1 Linear Regression

Consider a scenario in which we have an input variable x and an output variable y. We also have m training examples that each contain the y value corresponding to a given x value. We would like to develop a learning algorithm that uses this data set to generate a function that predicts the y values for all x values. This function h is called the **hypothesis**. If we assume that this function is linear then we may write:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

We call θ_i the **parameters**. This type of function is called a **linear regression** with one variable or a **univariate linear regression**. We want to solve a minimization problem in that we want to minimize $(h_\theta(x) - y)^2$ for every example. We use a power instead of an absolute value because this will make our lives easier when we need to start taking derivatives. Here we define the **cost function**.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

We want to find the values of θ_0 and θ_1 for which the cost function is minimized. At this point you may recall that $\vec{\nabla} J$ gives the direction in which J decreases the most at a given point.

$$\vec{\nabla} J = \left(\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1} \right)$$

Thus it would make sense to update our parameters as follows:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

Where α is a parameter that controls the step size and j can be equal to 0 and 1. α is called the **learning rate**. If α is too small then the steps take too long. If α is too large then it will overshoot the minimum and fail to converge. The process of continuously updating the parameters using the gradient is called the **gradient descent algorithm** and can be written as follows.

1. Start with an initial guess
2. Modify θ_0 and θ_1 by some multiple of the gradient
3. Repeat this process until J converges to a local minimum.

If we had multiple input variables x then we could write our hypothesis as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 = \theta^T X$$

This is an example of **multivariate linear regression**. Of course the gradient descent algorithm would still be applied in the same manner. One detail to note when applying this algorithm is the concept of **feature scaling**. You should generally make sure that the given features have a similar scale. This will allow the gradient descent to converge more quickly. For example if we have two input variables x_1 and x_2 and x_1 varies from 1 to 5 while x_2 varies from 0 to 2000 then this will lead to a θ_1 vs θ_2 plot that is very tall and thin in shape. If you define each value x_1 and x_2 by dividing by the maximum of each feature then the contour plots become more like circles. We typically want to establish a range between -1 and 1. We can also do a **mean normalization** in which we take each feature x_i and replace it with $(x_i - \text{mean})/\text{max}$. max can be replaced by max $-$ min or by the standard deviation.

2.2 Polynomial Regression

Recall our previous example with one input variable x and one output variable y. We initially assumed that the hypothesis function takes the form $h_\theta(x) = \theta 0 + \theta_1 x$. Rather than choosing a linear form we can choose a polynomial form by including higher powers of x. For example we can include two more powers by writing:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

This form of regression is called **polynomial regression**. Using this may be better as it may provide a better fit to the data.

2.3 Normal Equation

Another method to determining the parameters for a hypothesis function uses the **normal equation**. The normal equation consists of a **design matrix** X and an output vector $\vec{y}$. The design matrix X contains all of the training data featuresand the output vector $\vec{y}$ contains all of the corresponding output values. The parameter vector $\vec{theta}$ can then be computed as follows.

$$\vec{\theta} = (X^T X)^{-1} X^T \vec{y}$$

Computing this gives the value of $\vec{\theta}$ that minimizes the cost funtion. The benefit of using this method is that there is no need to choose a learning rate and no need to iterate or check for convergence. Gradient descent is generally used when n is massive (typically when $n \geq 10000$).

Notice that there may be cases when $X^T X$ is non invertible. This is commonly caused by having two many features in the learning model. The first thing to do in this case is to check for any redundant features. Another method that we can apply is **regularization** which will be discussed later.

2.4 Logistic Regression

Another type of learning problem that we commonly deal with is the classification problem in which y is a discrete value. Let us consider the case in which y can be either 0 or 1. In this case we would like to have a way of ensuring that the value returned by the hypothesis function is always between 0 and 1. This job is done by the **sigmoid function/logisitic curve**.

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

Applying this to the hypothesis function gives

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

When the $h_\theta(x) \geq 0.5$ we predict that $y = 1$. Otherwise we predict that $y = 0$. We essentially treat the output as the probability that that the given example falls under the class corresponding to $y = 1$.

We notice from the sigmoid function that $h_\theta \geq 0.5$ when $\theta^T x \geq 0$. Say for example that we have two features. Then this translates to $\theta_1 x_1 + \theta_2 x_2 \geq \theta_0$. Thus we predict $y = 1$ if $\theta_1 x_1 + \theta_2 x_2 \geq \theta_0$. Plotting $\theta_1 x_1 + \theta_2 x_2 = \theta_0$ gives the **decision boundary**. Figure 1 shows a specific example in which $\theta_0 = -3$ and $\theta_1 = \theta_2 = 1$. In this case the decision boundary is represented by $x_1 + x_2 = 3$. All points to the left of the decision boundary are classified as 0 while all points to the right of the decision boundary are classified as 1.

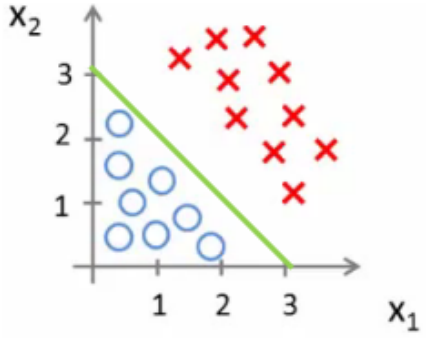


Figure 1: The decision boundary for a learning problem involving two features. The decision boundary is given by $x_1 + x_2 = 3$ and is shown in green. It is the line where $h_\theta(x) = 0.5$.

How do we write a cost function for this method? Recall that our previous cost function was as follows

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

If we define $\text{Cost}\left(h_\theta(x^i), y^i\right) = \frac{1}{2}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$ then we have

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}\left(h_\theta(x^i), y^i\right)$$

Applying gradient descent on this cost function may yield a local minimum instead of a global minimum. For this reason $J(\theta)$ is considered to be a **non convex function**. Ideally we would like to have a **convex function** so that we converge to a global minimum when we run gradient descent. This can be done by modifying the cost function as follows

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

First recall that $0 \leq h_\theta(x) \leq 1$ since we are using the sigmoid function. If $y = 1$ and $h_\theta(x)$ is close to 0 or vice versa then the cost function will yield a very large positive value. However if both y and $h_\theta(x)$ are very close to each other then the cost function will yield a value very close to 0. This type of cost function is called a **logistic regression cost function**.

How do we apply logistic regression when there are more than two classes? Well we basically apply logistic regression once for each class while considering all of the other possible classes as simply an opposing class. This is illustrated in Figure 2 for the case of 3 different classes. We essentially split the training set into three separate binary classification problems:

- Triangles vs Crosses and Squares: $h_\theta^{(1)}(x)$
- Crosses vs Triangles and Squares: $h_\theta^{(2)}(x)$
- Squares vs Crosses and Triangles: $h_\theta^{(3)}(x)$

When given a new input x we pick the class i that maximizes the value of $h_\theta^{(i)}$.

2.5 Regularization

One thing we have to consider when determining an appropriate hypothesis to our data is to avoid underfitting and overfitting. **Underfitting** refers to scenarios where the hypothesis does not provide a good enough fit to the training data. If this is the case then the hypothesis function will likely perform poorly when testing on new data. An example of underfitting or **high bias** is shown in the left graph in Figure 3. For this case a linear regression was used such that $h_\theta(x) = \theta_0 + \theta_1 x$. **Overfitting** refers to scenarios where the hypothesis fits the data so well that it does not generalize well to new data. An example of overfitting or **high variance** is shown in the right graph in Figure 3. For this case a polynomial regression of order 4 was used such that $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

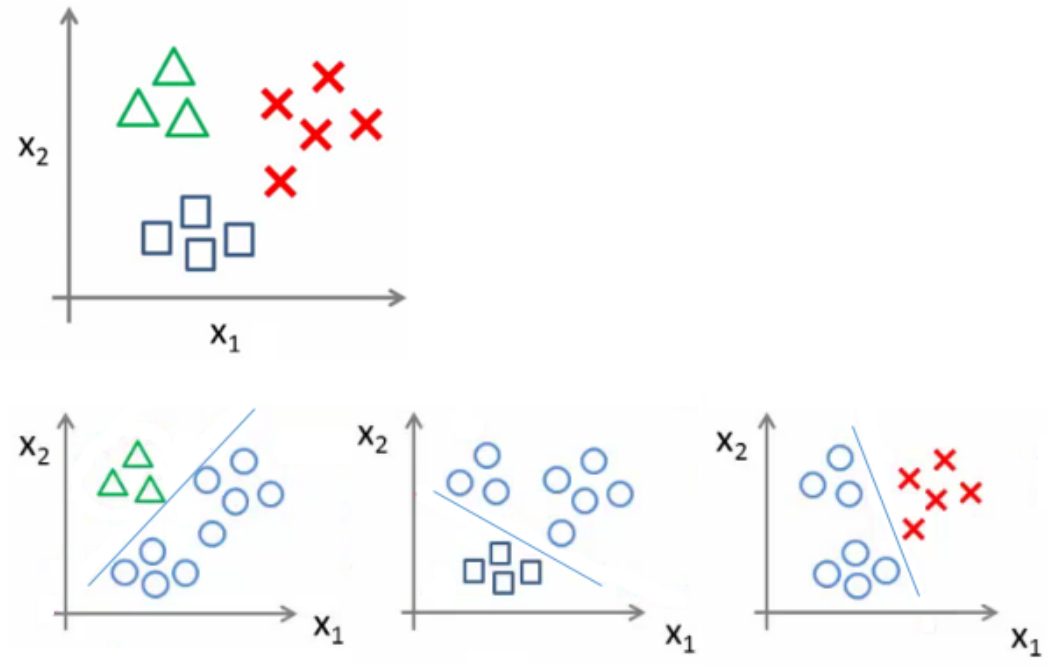


Figure 2: Logistic regression being used for multiclass classification as opposed to binary classification. We apply logistic regression once for each class while considering all other classes as simply an opposing class.

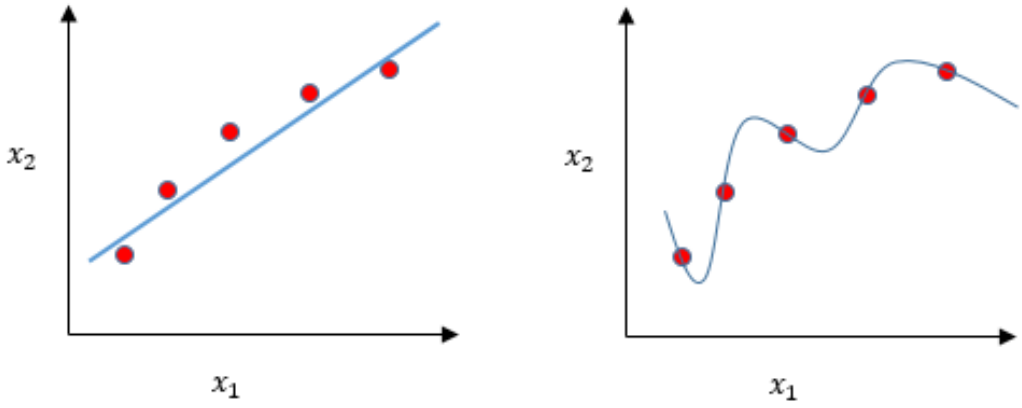


Figure 3: The graph to the left demonstrates an example of underfitting or high bias while the graph to the right demonstrates an example of overfitting or high variance.

One thing we can do in order to avoid overfitting is reduce the number of features. However this is not ideal as data is lost. Another method we can apply is **regularization**. With regularization we keep all of the features but reduce the magnitude of some of the parameters. In order to reduce the size of certain parameters, we add terms with those parameters and massive constants.

Let us consider the hypothesis function that resulted in the overfitting case shown in Figure 3. This function was $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$. Currently we are trying to minimize the cost function as shown below.

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

We recognize that the overfit would be reduced if we minimize the parameters θ_3 and θ_4. This can be done by adding two terms for each parameter. These terms are highlighted in blue.

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\theta_3^2 + 1000\theta_4^2$$

Consider 100 features $x_1 ... x_{100}$. Unlike the polynomial example we do not know what the higher order terms are. How do we pick which ones to shrink? With regularization we take the cost function and modify it to shrink all of the parameters as shown below.

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

Notice that we do not penalize θ_0. In practice including θ_0 has little impact. λ is called the **regularization parameter**. Applying this method normally results in a much smoother curve. Note that if λ is very large then all of the parameters will end up being close to 0. This will result in underfitting. When applying regularization while using the normal equation we rewrite the equation as follows.

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

For linear and logistic regression the update rule is modified as follows.

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

2.6 KNN classification

KNN classification or **K nearest neighbours classification** is another algorithm used for dealing with classification problems. Consider a collection of data points where the color of each point indicates its response class. This is shown in Figure 4.

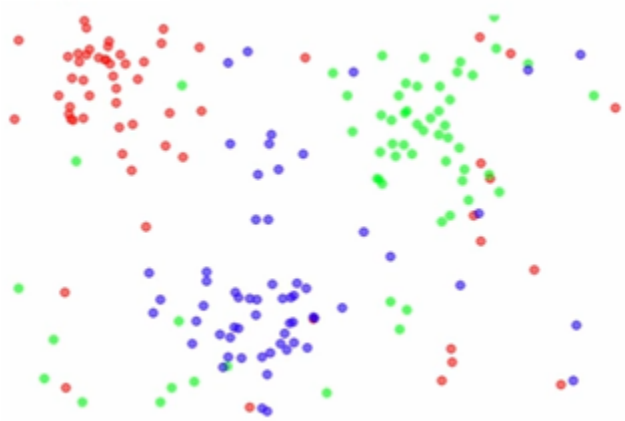


Figure 4: Several data points in a feature space. The colour of each data point indicates its response class.

Consider placing a new data point at a random location in the feature space. The idea is that this new object is assigned to the class most common among its k nearest neighbours. If $k = 1$ then the object is simply assigned to the class of its single nearest neighbour. KNN classification for $k = 1$ and $k = 5$ is shown in Figures 5 and 6 where the colour gives the predicted response class. Note that the white areas indicate regions in which KNN cannot make a clear decision because there is a tie between two classes.

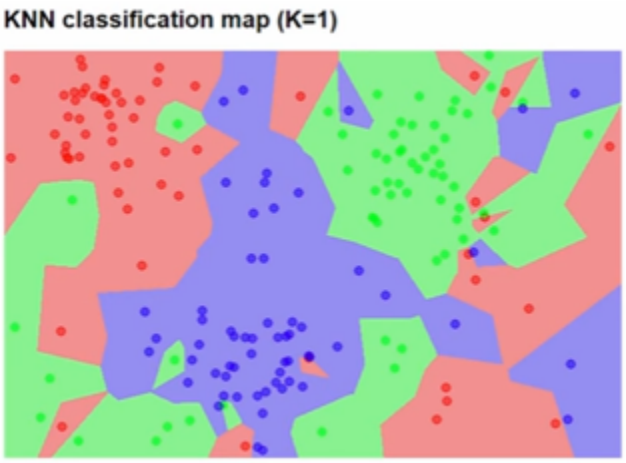


Figure 5: KNN classification for $k = 1$.

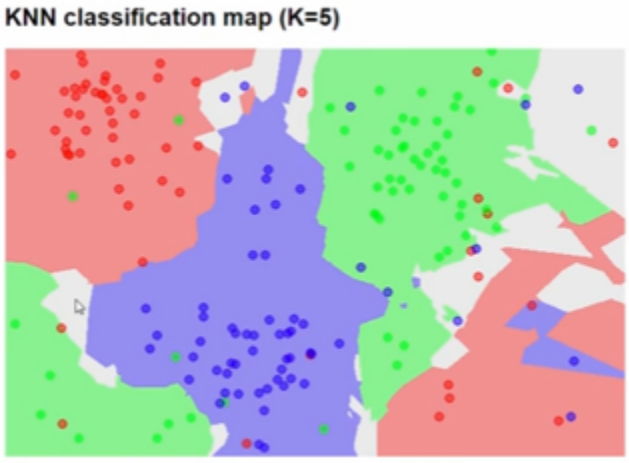


Figure 6: KNN classification for $k = 5$.

2.7 K Fold Cross Validation

Cross validation is a model validation technique used for assessing how well our results will generalize to an independent data set. It steps can be stated as follows.

1. Split the data into K equal partitions/folds
2. Use fold 1 as the testing set and the union of the other folds as the training set
3. Calculate the testing accuracy
4. Repeat steps 2 and 3 K times using a different fold as the testing set each time
5. Use the average testing accuracy as the estimate of out of sample accuracy

Note that the testing accuracy for a given fold can be calculated using a few different **evaluation metrics**. One evaluation metric is the **mean absolute error** metric (MAE) in which we calculate the mean of the absolute value of the errors.

$$\frac{1}{n}\sum_{i=1}^{n}|h_\theta(x_i) - y_i|$$

Another evaluation metric is the **mean squared error** metric (MSE) in which we calculate the mean of the squared errors. This metric is used more frequently because it punishes larger errors.

$$\frac{1}{n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)^2$$

The last evaluation metric that we will review is the **root mean squared error** metric (RMSE) in which we calculate the square root of the mean of the squared errors. This is even more popular because it is interpretable in y units.

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)^2}$$

This cross validation method is illustrated in 7 for the case of $K = 5$. Additional things that can be done to improve the application of K fold cross validation are as follows.

- Repeat cross validation multiple times (with different numbers of folds) and average the results. This provides a more reliable estimate of out of sample data performance by reducing the variance associated with a single trial of cross validation
- Create a **hold out set** that contains a portion of the data while conducting the model building process. Then locate the best model using cross validation on the remaining data and test it using the hold out set. This is a more reliable estimate of out of sample performance since the hold out is truly out of sample.

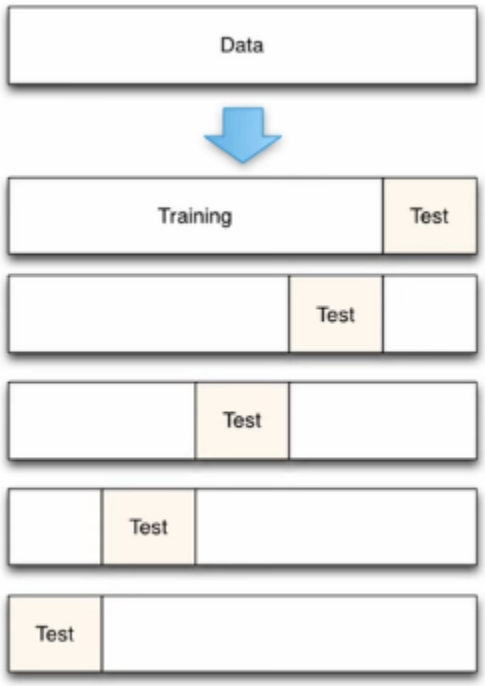


Figure 7: Illustration of 5 fold cross validation.

Another common construct used in assessing the accuracy of a model is the **confusion matrix**. The confusion matrix is a 2 x 2 matrix and each entry contains one of the following values. For ease of explanation we will pretend our learning model is attempting to predict whether or not a given patient has diabetes.

- **True Positives**: We correctly predicted that they do have diabetes
- **True Negatives**: We correctly predicted that they do not have diabetes
- **False Positives**: We incorrectly predicted that they do have diabetes.
- **False Negatives**: We incorrectly predicted that they do not have diabetes.

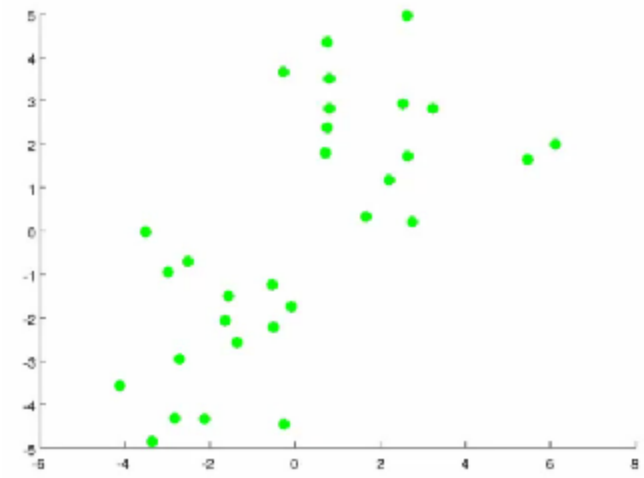
An example of a confusion matrix is shown below.

n=192	**Predicted: 0**	**Predicted: 1**
Actual: 0	118	12
Actual: 1	47	15

2.8 K Means Clustering

The KNN classification algorithm that was previously described is a form of **supervised learning** since the machine is given a set of labels and is asked to fit a hypothesis. Sometimes problems will require **unsupervised learning** in which the data is not labeled and the structure of the data needs to determined. **Clustering** refers to learning from unlabeled data. Clustering algorithms group data together based on data features. One example of a clustering algorithm is the **K means clustering algorithm**. Consider the set of unlabeled data shown below. We want to group this data into two clusters. The algorithm completes the following steps:

1. Randomly allocated K points as the cluster centroids.
2. Go through each data point and depending on whether it is closer to the rend centroid or the blue centroid, assign each point to one of the two clusters.
3. Take each centroid and move it to the average of the correspondingly assigned data points.
4. Repeat steps 2 and 3 until convergence.



In this example we will use $K = 2$. The left diagram in Figure 8 represents the placement of the cluster centroids and the corresponding classifications. The right diagram in Figure 8 represents movement of the cluster centroids to the average of their assigned data points.

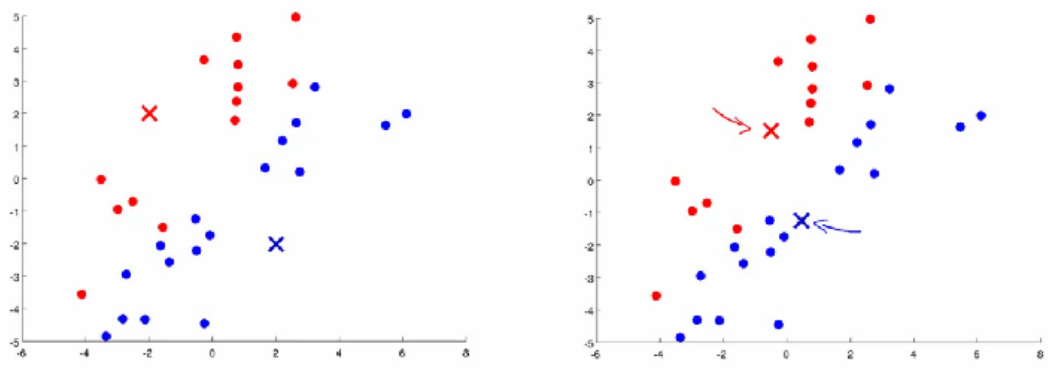


Figure 8: An example of the K means clustering algorithm for the case of $K = 2$. Each X represents a cluster centroid.

At this point it would be useful to define some notation.

- c^i is the index of clusters $1...K$ to which x^i is currently assigned
- μ_k is the cluster associated with centroid k.
- μ_{c^i} is the cluster centroid of the cluster to which x^i has been assigned.

Our cost function or the function we are trying to minimize can then be written as follows.

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

This represents the sum of the squared distances between the training example x^i and the cluster centroid to which x^i has been assigned to. How do we initialize the centroids? One method is as follows.

1. Set the number of centroids to be less than number of examples ($K < m$)
2. Randomly select K training examples
3. Set μ_1 to μ_K to be equal to these example values

Note that K means can converge to different solutions depending on the initialization setup. In other words there is a risk of obtaining a local minimum. We can do multipole random initializations and check to see if we get the same result. Many same results are likely to indicate a global minimum.

2.9 Dimensionality Reduction

Another type of unsupervised learning problem is **dimensionality reduction**. Consider having collected many features. Is there a way to remove the redundant features while speeding up the algorithm? Figure 9 illustrates an example of a dimensionality reduction from 2D to 1D and Figure 10 illustrates an example of a dimensionality reduction from 3D to 2D. One can see that this is reasonable if all of the points exist in a shallow area. Thus we can basically ignore one dimension.

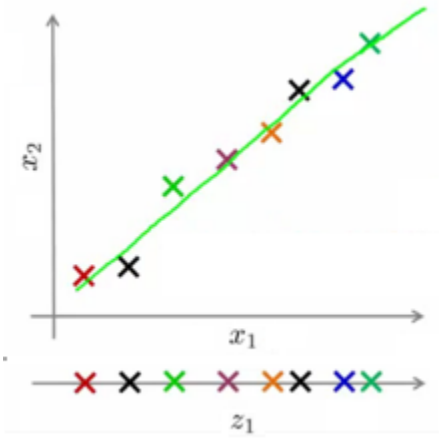


Figure 9: Dimensionality reduction from 2D to 1D.

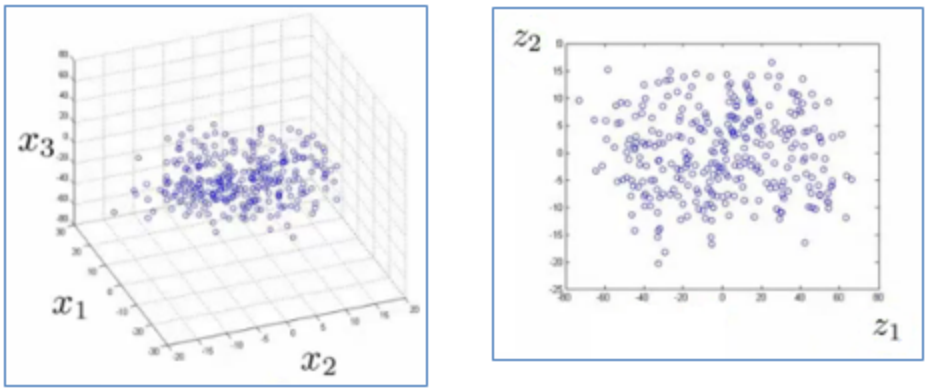


Figure 10: Dimensionality reduction from 3D to 2D.

The most common algorithm for applying dimensionality reduction is **PCA** or **Principle Component Analysis**. Consider again the 2D data set shown in Figure 9. We want to find a single line to project this data on. The distance between each point and the projected version should be small. PCA tries to find a lower dimensional surface such that the sum of the squared distances from that surface is minimized. These distances are sometimes called the **projection error**. Thus PCA tries to find the surface that has the minimum projection error. Note that you should normally do mean normalization and feature scaling on your data before applying PCA.

- Mean normalization: Replace x_j^i with $x_j^i - \mu_j$. This translates to determining the mean of each feature set and then for each feature subtracting the mean from that value. This scales the mean to be 0.
- Feature scaling: Replace each x_j^i with $(x_j^i - \mu_j)/s_j$ where s_j is some measure of the range. This could be max – min but the standard deviation is more common.

To reduce the dimensions of the data we need to compute the **covariance matrix** σ.

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

We then compute the eigenvectors of Σ by applying singular value decomposition on Σ to obtain $U\Sigma V$. It turns out that the columns of U are the u vectors that we want. To reduce a system from n dimensions to k dimensions we simply take the first k column vectors of U. In order to change x (which is n dimensional) to z (which is k dimensional) we take the first k columns of U to form a new matrix U_{reduce}. z is then given by $z = U_{\text{reduce}}^T x$

The steps for PCA can be summarized as follows:

1. Preprocessing (Mean Normalization and Feature Scaling)
2. Calculate the covariance matrix Σ
3. Calculate the eigenvalues using singular value decomposition
4. Take the first k column vectors from U to form U_{reduce}.
5. Calculate $z = U^T_{\text{reduce}} x$

How do we choose the number of principle components k? We know that PCA tries to minimize the averaged squared projection error.

$$\frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - x^{(i)}_{\text{approx}} \right\|^2$$

We choose k if it satisfies the following inequality.

$$\frac{\frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - x^{(i)}_{\text{approx}} \right\|^2}{\frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} \right\|^2} \leq 0.01$$

It is small if x^i is very close to x^i_{approx}. We therefore lose very little information in the dimensionality reduction.

2.10 Support Vector Machine

Support Vector Machine or **SVM** is a classification algorithm that works well for dealing with extreme cases. Consider the data shown in Figure 11. Many different lines/decision boundaries can be drawn in order to properly separate the data. But an unoptimized decision boundary could result in greater misclassification on new data.

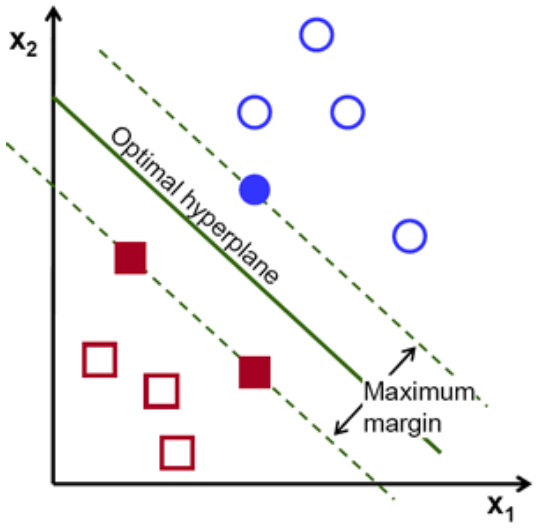


Figure 11: SVM works to determine the optimal decision boundary between two classes.

Ideally we would like a decision boundary that is a maximum orthogonal distance away from the closest point of each class. In the context of SVMS, this distance is normally called the **margin**. The decision boundary is normally called the **hyperplane** and the two lines whose distance from each other gives the margin are called the **support vectors**.

The situation described in Figure 11 is an example of a **linear SVM**. Sometimes simply determining a flat line or plane for a decision boundary is not enough. We may need to add an additional dimension to our data first. Figure 12 illustrates an example of transitioning from 1D to 2D and Figure 13 illustrates an example of transitioning from 2D to 3D. This application of SVM is called **non linear SVM**.

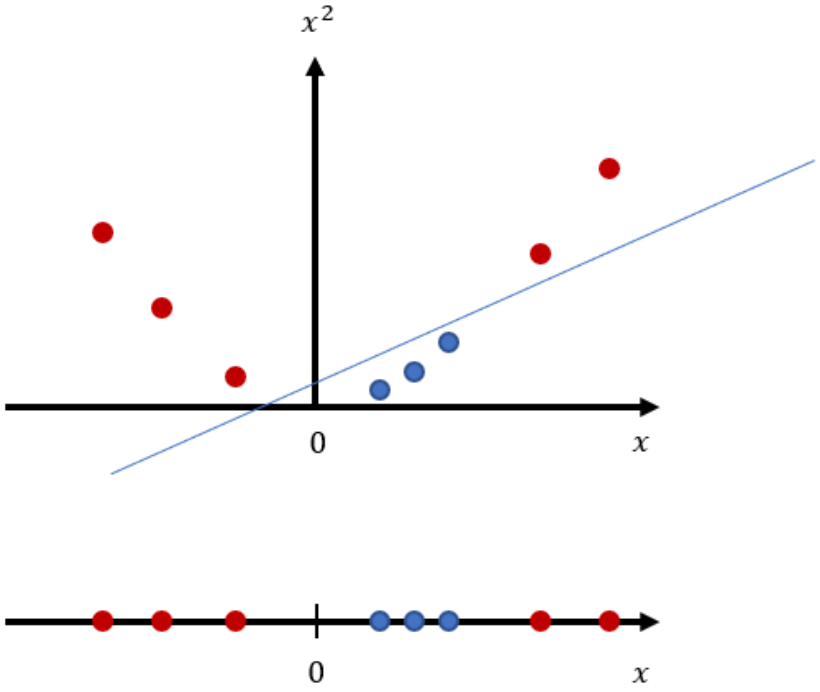


Figure 12: Applying a non linear SVM in which we transition from 1D to 2D. This transition now allows us to separate the data using a simple line.

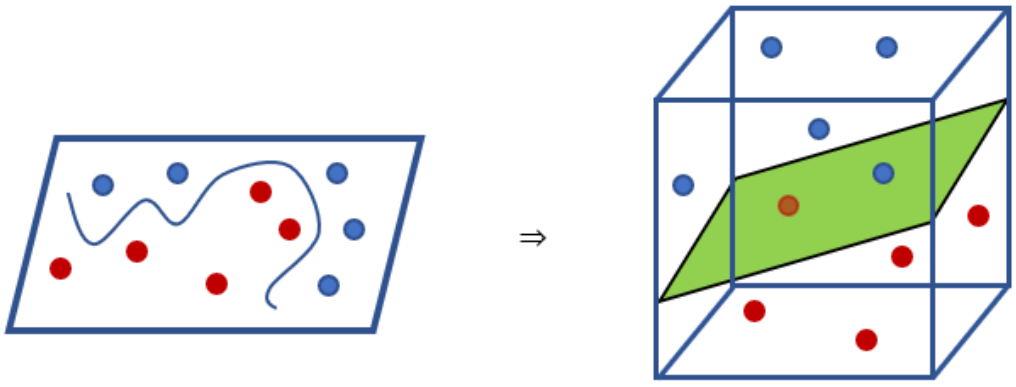


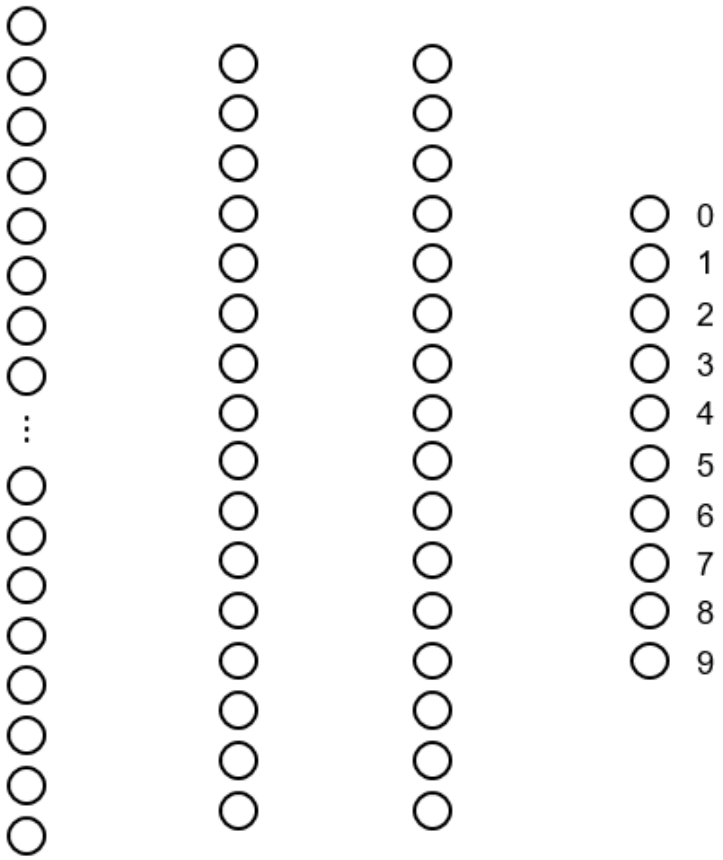
Figure 13: Applying a non linear SVM in which we transition from 2D to 3D. This transition now allows us to separate the data using a 2D plane.

3 Neural Networks

3.1 Structure of a Neural Network

We will look at the **plain vanilla form / multilayer perceptron**. A **neuron** is a thing that holds a number between 0 and 1. This number is called its **activation**.

Let us consider the example of classifying handwritten digits. These digits will exist on a 28 by 28 pixel grid. Thus the network will start with several neurons that each correspond to one of the 28 x 28 = 784 pixels of the input image.The value on each neuron will represent the greyscale value of the corresponding pixel where 0 is black and 1 is white. These 784 neurons will make up the first layer of the network.



In the final layer of the neural network we notice that each neuron represents one of the ten digits. Activation in these neurons represents how much the system thinks that the given image corresponds with the given digit. The highest number represents the choice for what the network believes is the image.

The middle layers are called **hidden layers**. The idea is that activations in one layer determine the activations in the next layer. One hope we might have is that neurons in the second hidden layer each represent one of the subcomponents and that neurons in the first hidden layer each represent one of the edges.

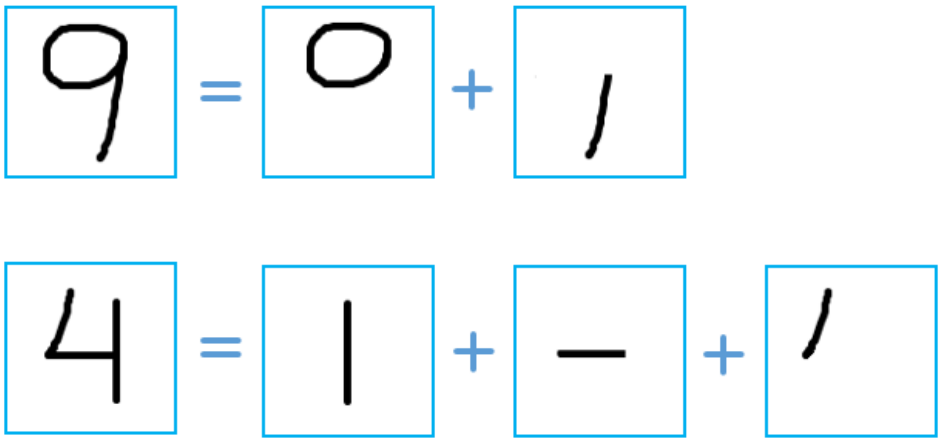


Figure 14: A few examples of how we might expect the digits to be decomposed into subcomponents. The idea we be to have each of the neurons in the second hidden layer represent one of these subcomponents.

How do the activations in one layer determine the activations in the next layer? Let us consider one very specific example. We will consider the connection between all of the neurons of the first layer and one of the neurons of the first hidden layer. This is shown in Figure 15. Notice that there is a **weight** w_i associated with each connection. Each weight determines how much the activation of the i^{th} neuron in the first layer influences the activation the particular neuron that we are considering in the first hidden layer. In fact the activation of this particular neuron is calculated using a weight sum of the activations of all of the neurons in the preceding layer. In this case the weighted sum is given by $a_1 w_1 + \cdots + a_{784} w_{784}$.

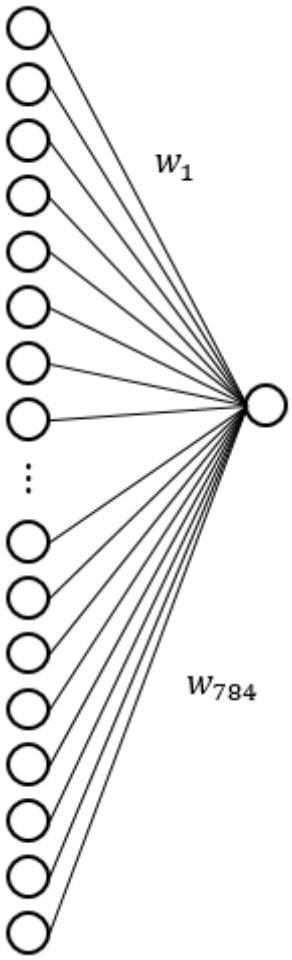


Figure 15: Each connection between one of the neurons from the first layer and the specific neuron from the first hidden layer is given a specific weight.

Let us consider that this particular neuron detects whether or not there is a horizontal edge located in the upper region as shown in Figure 16. Then for this particular case we would want really high positive weights in the region of the edge and really low negative weights in some of the regions surrounding the edge.

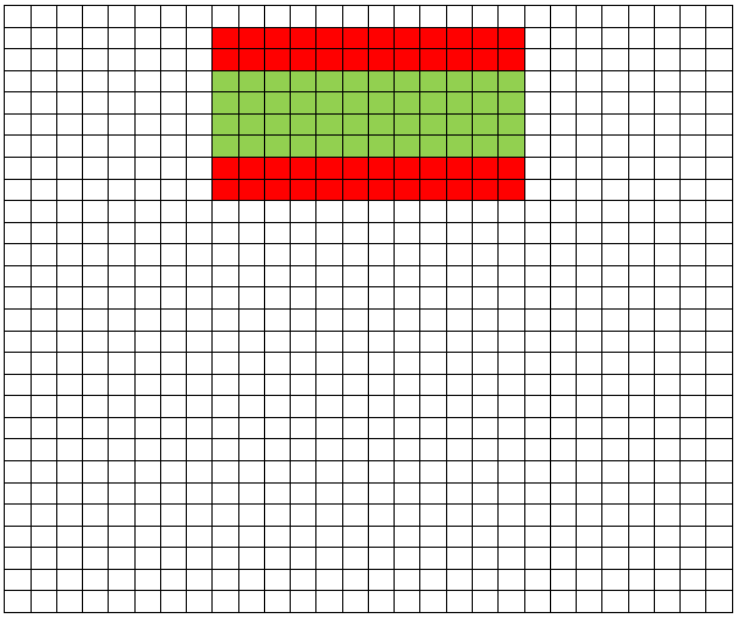


Figure 16: The purpose of this neuron could be to detect whether or not a horizontal edge is located in the upper region of the image grid. Here positive weights are indicated in green and negative weights are indicated in red.

When we compute the weight sum we may come out with any number. However we want the activations to be between 0 and 1. Thus we would like to be able to insert this weighted sum into a function that compresses the number line into a range between 0 and 1. One function that does this is the **sigmoid function/logistic curve**. This function is shown in Figure 17.

Using this sigmoid function we can then calculate the activation of this particular neuron using $\sigma(a_1 w_1 + \cdots + a_{784} w_{784})$. But perhaps you do not want the neuron to light up when its weighted sum is greater than 0. Maybe you want it to light up when its greater than 10. Then we can assign a **bias** as follows: $\sigma(a_1 w_1 + \cdots + a_{784} w_{784} - 10)$. The bias is generally denoted using the symbol b. So the general expression for calculating the activation of our particular neuron is:

$$a = \sigma(a_1 w_1 + \cdots + a_{784} w_{784} + b)$$

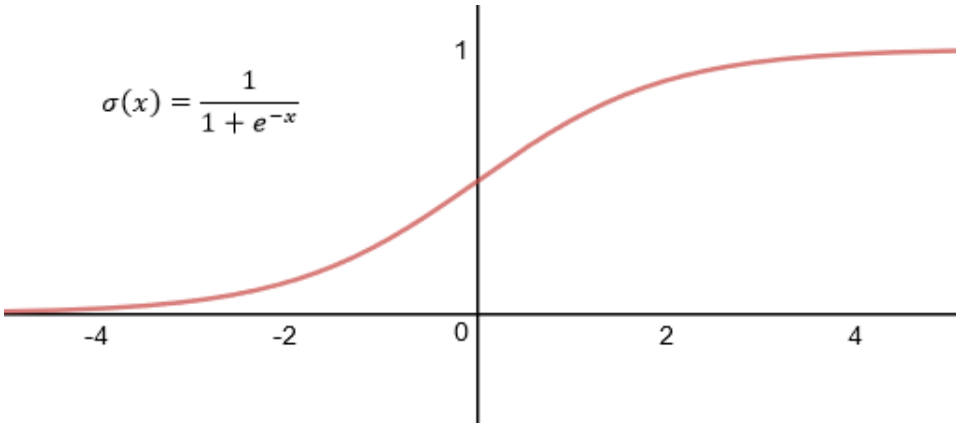


Figure 17: The sigmoid function transforms very negative inputs into values that are close to 0 and very positive inputs into values that are close to 1.

In summary the weights tell you what pixel pattern this neuron is picking up on and the bias tells you how high the weighted sum needs to be before the neuron starts getting meaningfully active. For this particular problem we have:

- 784×16 weights + 16 biases (first layer to first hidden layer)
- 16×16 weights + 16 biases (first hidden layer to second hidden layer)
- 16×10 weights + 10 biases (second hidden layer to final layer)

Thus we have a total of 13002 weights and biases. **Learning** refers to finding the right weights and biases. At this point it would be useful to invent some notation. The general symbol for an activation is $a_j^{(k)}$ where k refers to the refers to the layer and j refers to a neuron inside that layer. This is illustrated in Figure 18. Looking at our previous example, the activations of all of the neurons in the first hidden layer can be written as follows.

$$a_0^{(1)} = \sigma(w_{00}a_0^{(0)} + w_{01}a_1^{(0)} + \cdots + w_{0n}a_n^{(0)} + b_0)$$
$$\vdots$$
$$a_k^{(1)} = \sigma(w_{k0}a_0^{(0)} + w_{k1}a_1^{(0)} + \cdots + w_{kn}a_n^{(0)} + b_k)$$

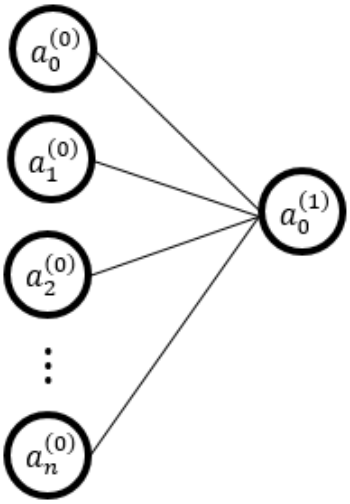


Figure 18: An example to illustrate the notation of activation values.

This system of equations can then be written into matrix form.

$$\sigma\left(\begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0n} \\ w_{10} & w_{11} & \cdots & w_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k0} & w_{k1} & \cdots & w_{kn} \end{bmatrix}\begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_k \end{bmatrix}\right) = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_k^{(1)} \end{bmatrix}$$

Writing this into condensed form gives:

$$\vec{a}^{(1)} = \sigma(W\vec{a}^0 + \vec{b})$$

We will begin by setting all of the weights and biases to totally random values. Let us consider that we give the network a handwritten 3. Ideally we would like for the network to produce 1 for the neuron corresponding to 3 and 0 for all of the other neurons. Of course instead we will get complete nonsense since all of the weights and biases were set randomly. At this point we would like a function that tells us how far the network was off from the correct activation values. We will define the **cost function** which gives us the sum of the squared differences between the correct values and the actual values. If I_1 is the first training example then we may have something like:

$$C(I_1) = \begin{cases} (0.43 - 0.00)^2 + \\ (0.28 - 0.00)^2 + \\ (0.19 - 0.00)^2 + \\ (0.88 - 1.00)^2 + \\ (0.72 - 0.00)^2 + \\ (0.99 - 0.00)^2 + \\ \qquad \vdots \end{cases}$$

If $C(I_1) = 3.37$ then 3.37 is the cost of the first training example. In general the cost is small when the network confidently classifies the image correctly. We then consider the average cost over all of the training examples. This provides a measure for how lousy the network is. The cost function basically takes in 13002 parameters as inputs and produces 1 number. We want to find the weights and biases the minimize the value of this cost function.

3.2 Learning by Gradient Descent

Let us first consider a 1D case where we would have $C(x)$. For the 1D case, we would move in the direction that the slope decreases and make the step size proportional to the slope to prevent overshooting. Now let us consider the 2D case of $C(x, y)$. At this point it would nice to have a function that gives us the direction in which the function decreases the most quickly. This function is the **gradient operator** represented by the symbol $\vec{\nabla}$. For the 2D case we have that $\vec{\nabla} = (\partial_x, \partial_y)$. You may recall that $\vec{\nabla}C$ gives the direction of steepest ascent and thus $-\vec{\nabla}C$ gives the direction of steepest descent. You may also recall that $\left\|\vec{\nabla}C\right\|$ indicates how steep this slope is. The algorithm for minimizing the cost function is basically as follows:

1. Compute $\vec{\nabla}C$
2. Take small step in the $-\vec{\nabla}C$ direction
3. Repeat until local minimum is found

If $\vec{W}$ is the vector containing all of the current weights and biases and α is some scalar then one iteration of this procedure can be described mathematically as follows

$$\vec{W}_{New} = \vec{W} - \alpha\vec{\nabla}C(\vec{W})$$

Recall that the cost function is averaged over all of the training data. Thus if you minimize it the network should perform better on all of the training examples. When we say that the network is learning, we referring to the process of minimizing of the cost function. This process of repeatedly nudging the input of a function by some multiple of the negative gradient is called **gradient descent**. It is a way of converging to a local minimum of that function.

So we initialize the network with random weights and biases and adjust them many times based on the gradient descent process. Using this procedure with our current setup, our network classifies about 96% of the new images correctly. This is pretty good! But one thing to note is that when we look at the resulting weight grids for any of the connections between the first layer and the first hidden layer, we would expect them to trace out clear patterns such as edges. Yet they look almost random. This means that our network has basically found a local minimum. This is verified with the fact that the network classifies random images confidently.

	I_1	I_2	I_3	I_4	I_5	$\cdots$	Average
w_0	-0.08	+0.02	-0.02	+0.11	-0.05		-0.08
w_1	-0.11	+0.11	+0.07	+0.02	+0.09		+0.12
$\vdots$							$\vdots$
w_{13002}	+0.13	+0.08	-0.06	-0.09	-0.02		+0.04

Table 1: Table showing the desired weight and bias adjustments for several different training examples. We then apply the changes corresponding to the averaging of all of the desired changes from each of the training examples.

3.3 Back Propagation

Back Propagation is the process behind which weights and biases are adjusted based on the networks results for a given set of training examples. Let us consider that the case in which we given our network a handwritten two. Then ideally we would like the activation in the neuron corresponding to 2 to be one and the activations in the rest of neurons in the final layer to be zero. There are 3 different ways we could increase the activation of the neuron corresponding to 2.

- Increase the bias b
- Increase the weights w_i in proportion to a_i
- Increase the activations a_i in proportion to w_i

We cannot directly change the activations but we can directly change the weights and biases. In this case the desire of the digit 2 neuron would be added up with the desires of all of the other digit neurons. This process is then repeated for all of the subsequent layers. This is called back propagation. We can do this for every training example and then average together the desired changes as shown in the Table 1 below.

Of course it would take a long time to compute the desired nudges for every single training example at every single gradient descent step. Instead what we commonly do is randomly shuffle our training data and then divide the data into several subgroups. We then compute steps according to each of these subgroups. While the descent is less careful it is much faster. This process of dividing the training examples into subgroups is called **stochastic gradient descent**.

Let us recap what we have discussed so far:

- Back propagation is the algorithm for determining how a single training example would like to nudge the weights and biases to cause the most rapid decrease to the cost.
- A true gradient descent step would involve doing this for all of the 1000s of training examples and then averaging. Since this is slow, we divide the training examples into subgroups and repeatedly go through each of the subgroups while making these adjustments. Naturally we will converge to a local minimum of the cost function.

Consider the very simple network shown in Figure 19. Notice that we denote the final layer by L. We also use y to denote the desired output. Also notice that this network contains 3 weights and 3 biases. If we denote the cost of our first training example as C_0 then $C_0 = (a^{(L)} - y)^2$. We can also write the activation on our final neuron as $a^{(L)} = \sigma(w^{(L)}a^{(L-1)} + b^{(L)})$. If we define $z^{(L)} \equiv w^{(L)}a^{(L-1)} + b^{(L)}$ then $a^{(L)} = \sigma(z^{(L)})$.

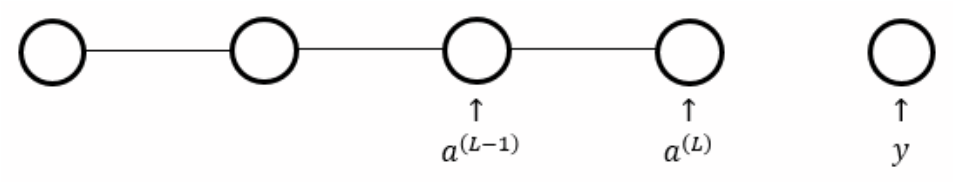


Figure 19: A very simple network to help explain back propagation.

In order to determine the weight $w^{(L)}$ that minimizes the cost C_0 it makes sense that we would want to know $\partial C_0/\partial w^{(L)}$. Using chain rule we can determine this as follows.

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \tag{1}$$

If we recall that $C_0 = (a^{(L)} - y)^2$ and that $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$ and that $a^{(L)} = \sigma(z^{(L)})$ then we can substitute these expressions into Equation 1 to get

$$\frac{\partial C_0}{\partial w^{(L)}} = 2a^{(L-1)}\sigma'\left(z^{(L)}\right)\left(a^{(L)} - y\right)$$

If C is the average cost over all of the training examples and there are n training examples then we can write

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

We can do a similar thing for the biases:

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = 2\sigma'\left(z^{(L)}\right)\left(a^{(L)} - y\right)$$

Using these expressions we can write out the gradient of our cost function for a given training example.

$$\nabla \vec{C} = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

We have now established how to calculate the gradient of the cost function! But we still need to consider networks consisting of layers with multiple neurons. Figure 20 illustrates the additional detail in notation in which we essentially add a subscript to indicate which neuron of the layer we are referring to. We should also note that the weight for the connection between the two labeled neurons in Figure 20 would be written as $w_{jk}^{(L)}$.

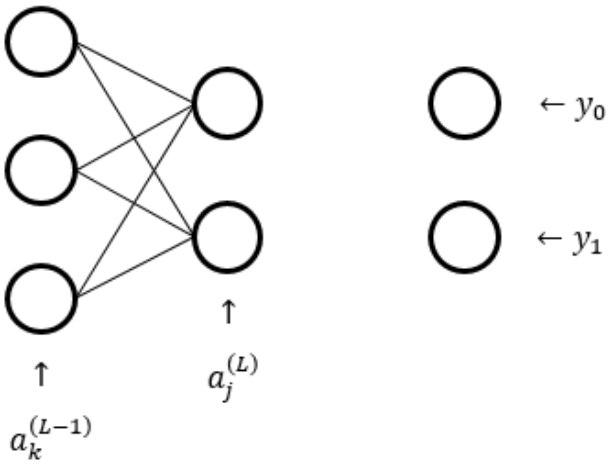


Figure 20: A subscript is added to our activation values to indicate which neuron of the layer we are referring to.

With this setup our previous equations are now written as follows:

$$C_0 = \sum_{j=0}^{n_L - 1} (a_j^{(L)} - y_j)^2 \tag{2}$$

$$a_j^{(L)} = \sigma(z_j^{(L)}) \tag{3}$$

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + b_j^{(L)} \tag{4}$$

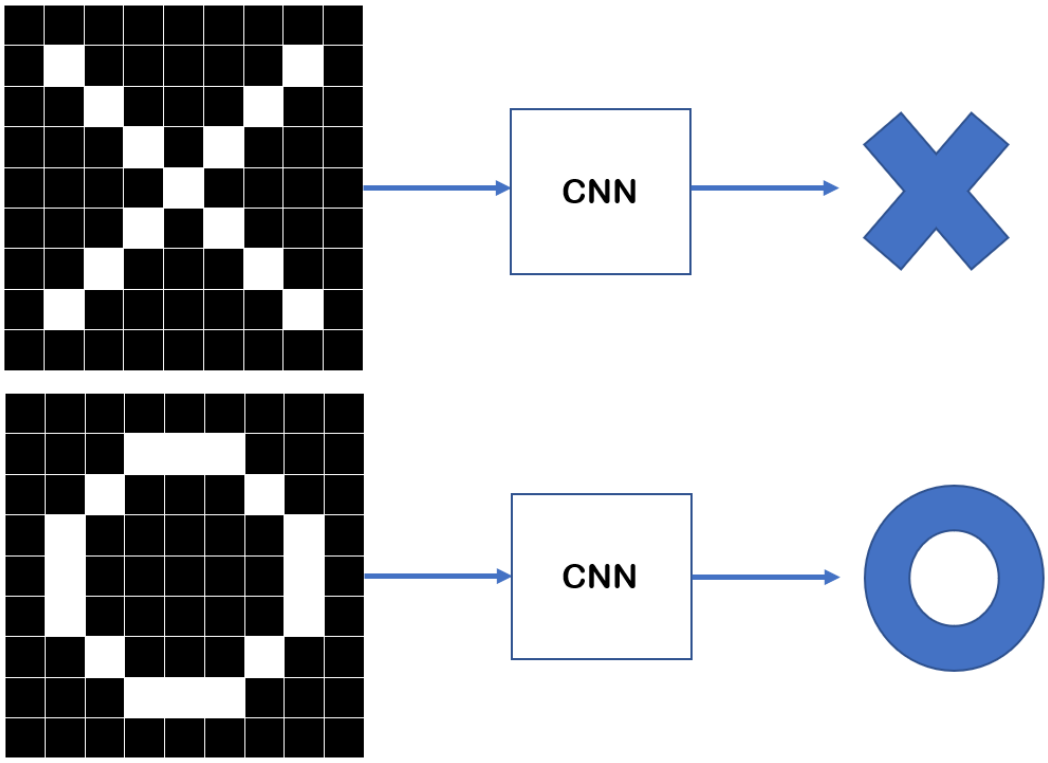
Finally our partial derivative is written as follows:

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}} \tag{5}$$

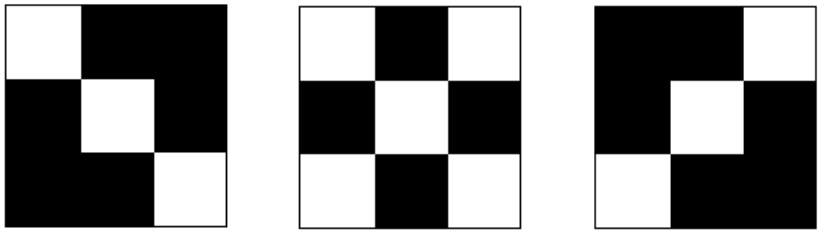
In the previous section we talked about using the gradient descent to minimize the cost function. But it may not have been clear as to how we take the gradient/partial derivatives since we had only expressed our cost function in terms of the final activations and the outputs. Now it should be clear how this gradient is computed based on the weight and biases and activations of our network.

3.4 Convolutional Neural Networks

Convolutional neural networks or **CNNs** are commonly used for image classification and are a step up from the multilayer perceptron that we discussed previously. Consider the following example in which a CNN is used to classify images or two dimensional arrays of pixels.



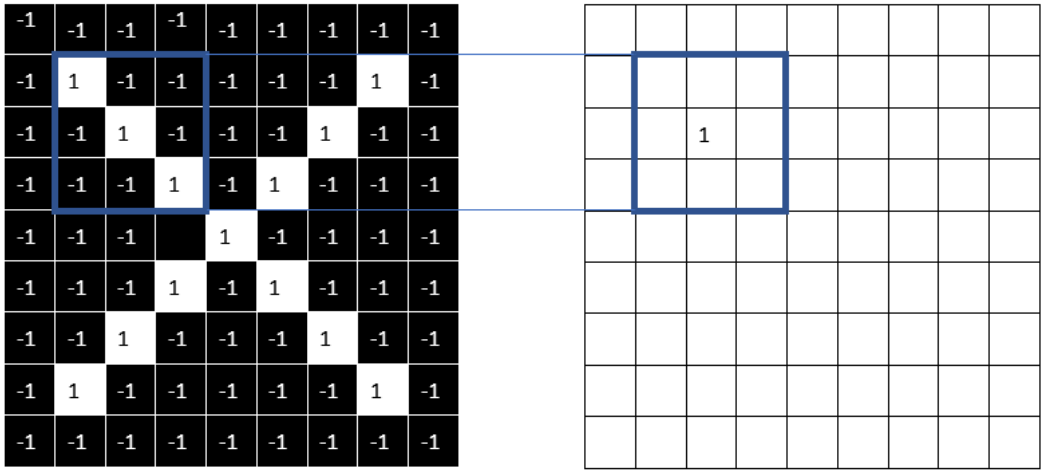
Of course not all arrays of pixels representing Xs and Os will come in this format. The arrangement of pixels is subject to many different types of transformations including translations, rotations and scaling. In order to correctly classify images with these types of transformations, CNNs match smaller parts or **features** of the image. The figure below shows three features that would seem useful for assessing whether or not an image is an X.



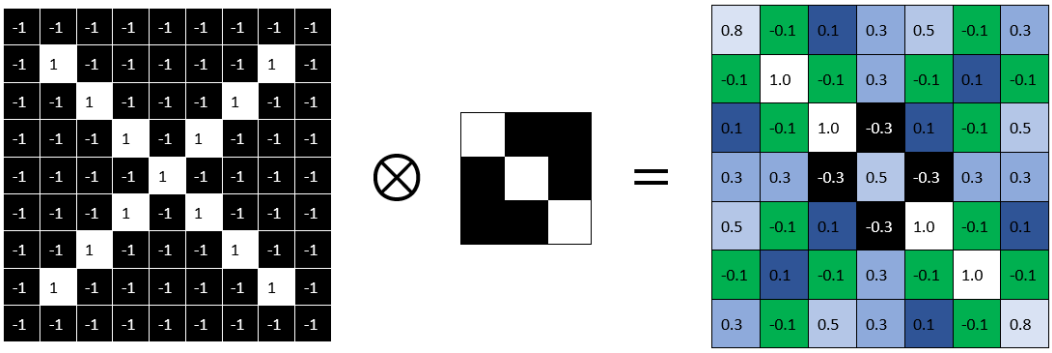
These features are compared with the input image in a process called **filtering**. This process can be stated as follows:

1. Line up the feature and the image patch.
2. Multiply each image pixel with the corresponding feature pixel.
3. Add them up and divide by the total number of pixels in the feature.

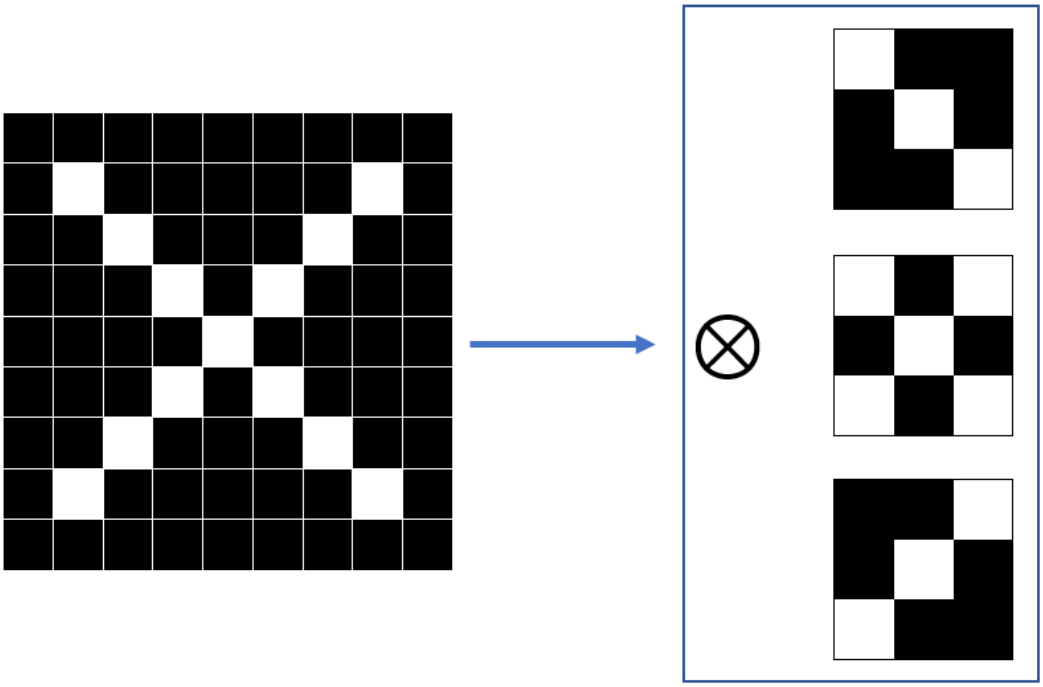
Note that for this case we could label all of the white squares as 1 and all of the black squares as -1. Let us consider applying the first feature above to the highlighted square below. Because our feature matches this square exactly we obtain a 1 in the filtered image in the location of the center of this square. All perfect matches will correspond to $9/9 = 1$ assuming that our feature is a 3 x 3 image.



Convolution refers to the repeated application of this feature to every pixel of the image. Applying convolution for this feature results in the following. Notice that we use the symbol $\otimes$ to denote **convolution**.



Of course this can do done for multiple features as shown below. The rectangle containing the desired features represents a **convolution layer**. In a convolution layer one image becomes a stack of filtered images. In this case we would obtain the filtered image shown above as well as the two more filtered images corresponding to the other two features.

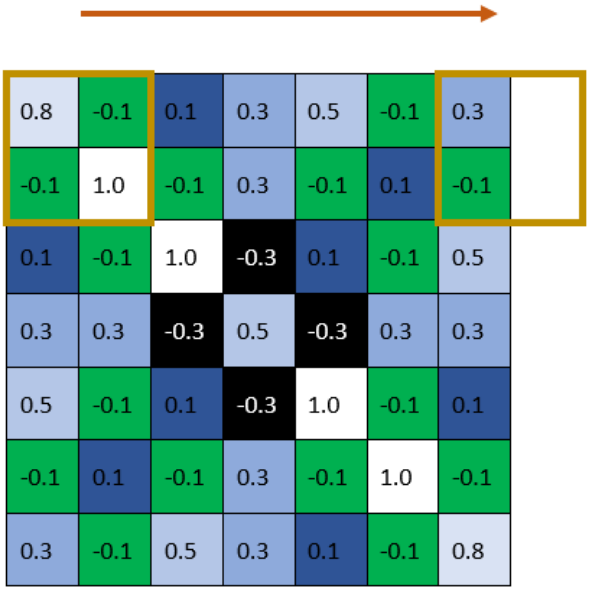


The next important layer to discuss is the **max pooling layer** or the **downsampling layer**. Pooling refers to shrinking the image stack. Only the locations on the image that show the strongest correlation to each feature are preserved and those maximum values combine to form a lower dimensional space. Pooling has the advantage of decreasing the amount of storage and processing required. However one must be careful to avoid removing too much information as this may hinder the performance of the CNN.

The process of pooling can be summarized as follows.

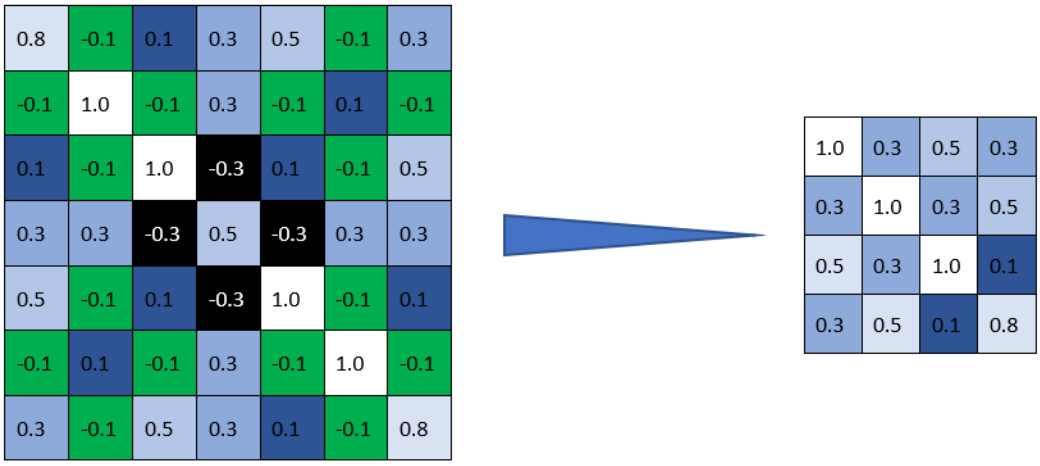
1. Pick a window size (usually 2 or 3)
2. Pick a stride (usually 2)
3. Walk your window across your filtered images.
4. From each window take the maximum value.

This image below illustrates this process for two different locations. For the second highlighted square we notice that half of this square is outside the image. In this case, we simply take the maximum value of the two available pixels.

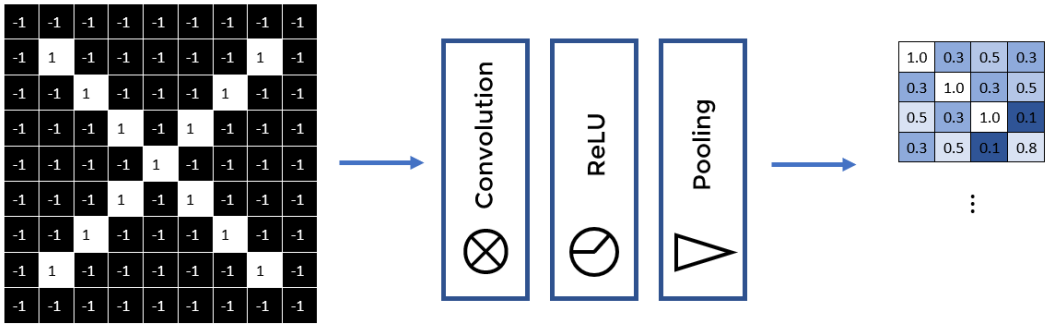


1.0	0.3	0.5	0.3

The final result is as follows. Once again this would be applied to each image in our image stack. Thus we would still have 3 images remaining. A pooling layer essentially turns a stack of images into a stack of smaller images.

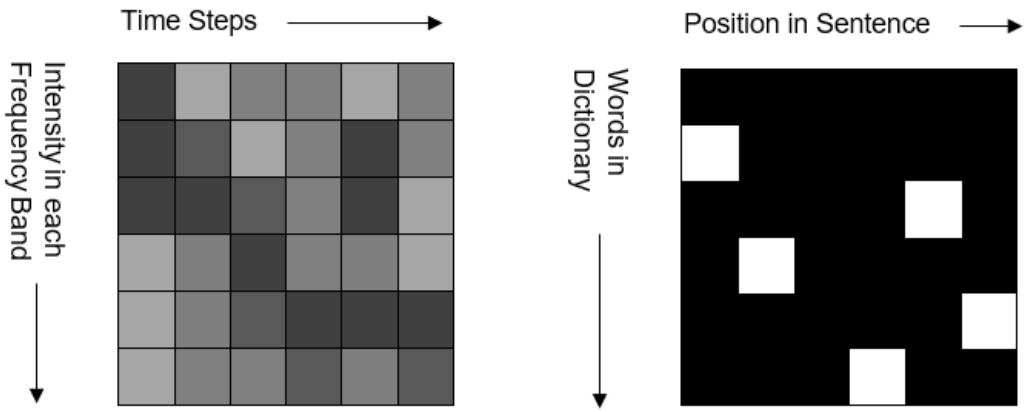


The final procedure that can be used in **normalization**. This keeps the math from blowing up or going to zero. All negative values are converted to 0 using the **ReLU function**. In a **ReLU layer** a stack of images becomes a stack of images with no negative values. If we stacked all three layers then we would have the following.



The term **deep stacking** refers to repeating several layers many times. The final layer is always a **fully connected layer**. After all of the convolutions and pooling procedures have been completed, every remaining value contributes to a neuron in this fully connected layer. The weight and biases are once again optimized through gradient descent. It is also possible to add several hidden layers between the fully connected layer and the output layer.

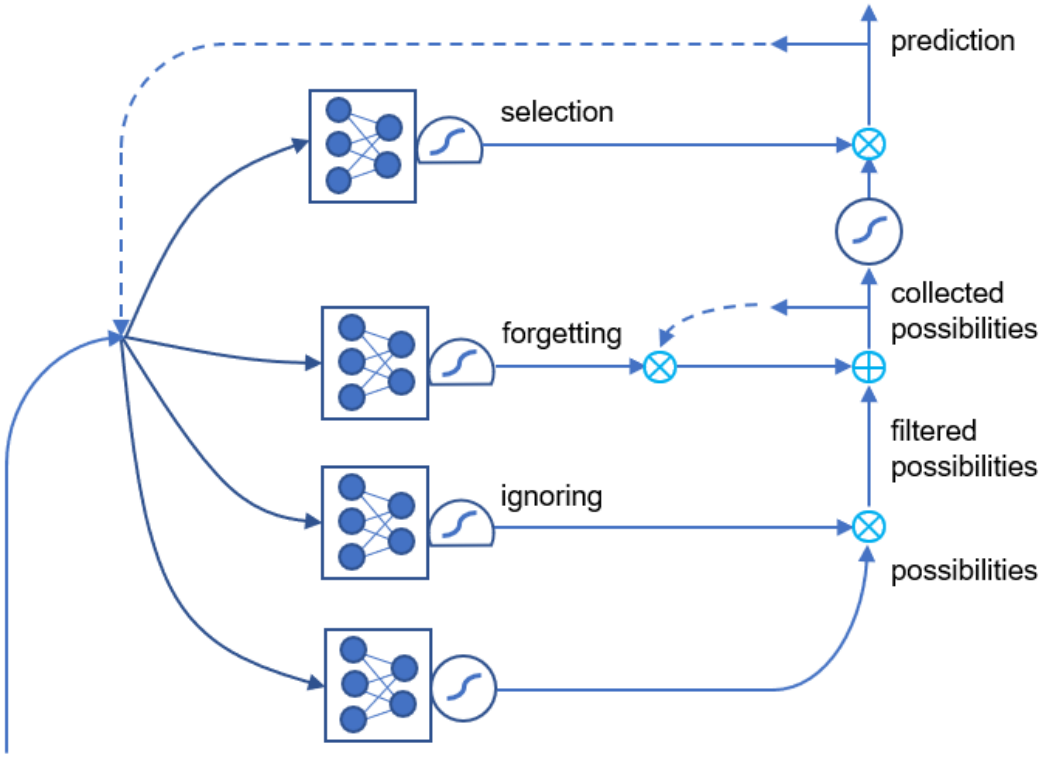
CNNs can be used for a lot more than image classification. They can be used on anything that exhibits some sort of spatial pattern. Examples are shown below for sound and words.



It is important to note that CNNs have the limitation of only being able to capture local spatial patterns in data. If the data cannot be made to look like an image then they are less useful. As a general rule of thumb: If your data is just as useful after swapping any of your columns with each other then you cannot use CNNs.

3.5 Recurrent Neural Networks

Recurrent neural networks or **RNNs** use previous predictions to help make new predictions. They are useful when there are temporal patterns or cycles in our data. Let us consider a children's book consisting of sentences of the format **A saw B**. We would like to be able to predict the next word in a sentence given a current word. To do this we will use an RNN that contains an **LSTM** or long short term memory unit. Our network will have the following structure.

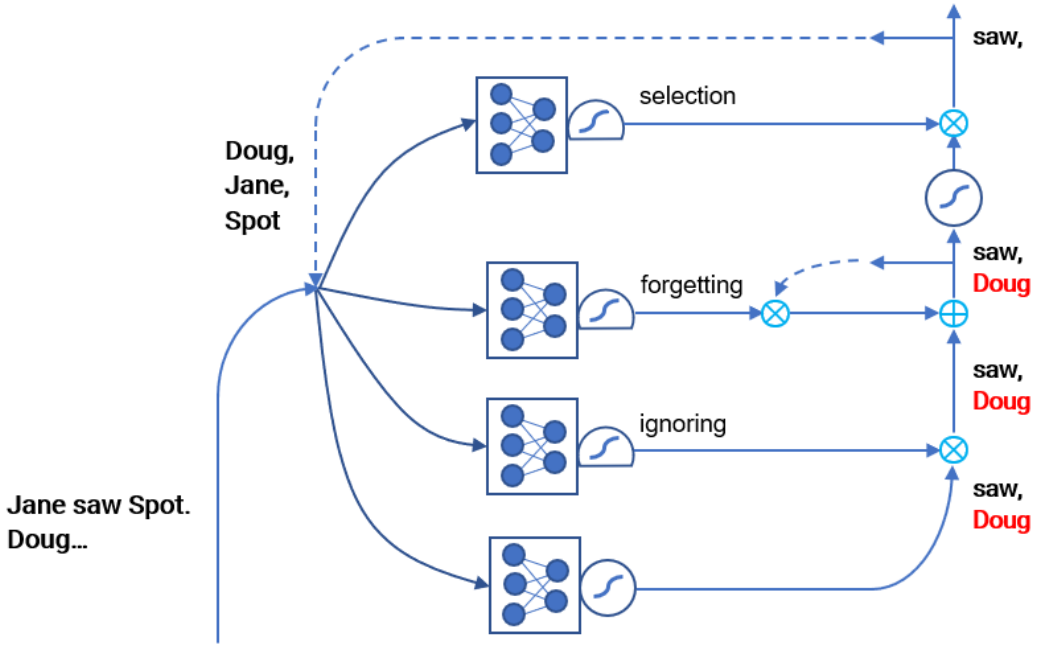


The circular and semicircular activation functions represent $\tanh x$ and the sigmoid function respectively. The idea of this setup is to allow the RNN to remember what happened many time steps ago. If it cannot do this then we could imagine it predicting things like **Doug saw Doug** and **Jane saw Spot saw Doug**. To do this we have introduced the **plus junction** represented by the symbol $\oplus$ and the **times junction** represented by the symbol $\otimes$. The times junction allows for **gating** as shown below. Here the first vector represents the signal and the second vector represents on/off gating.

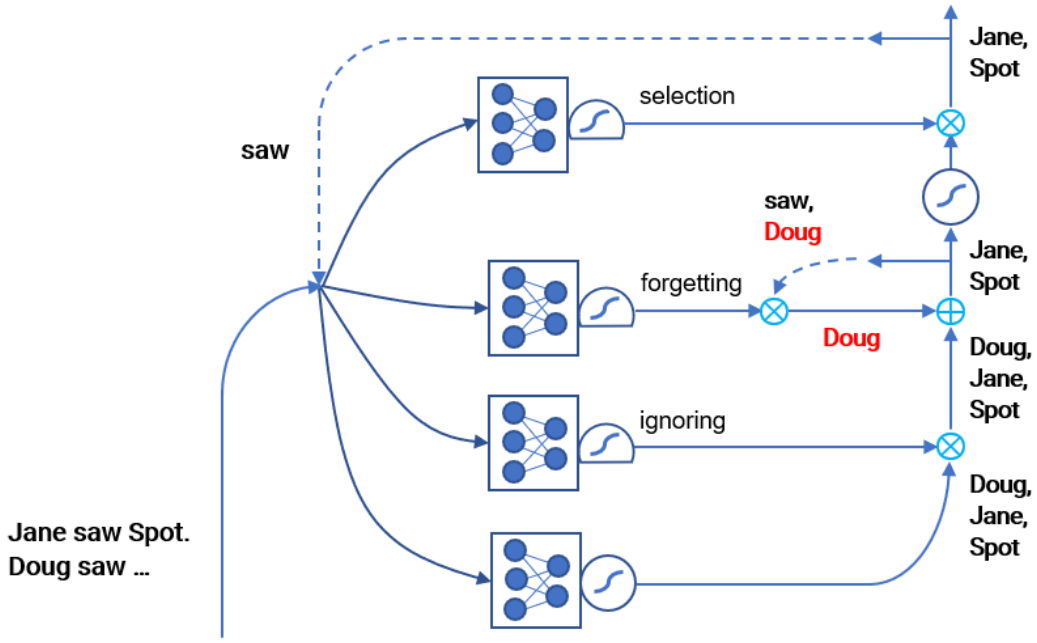
$$\begin{pmatrix} 0.8 \\ 0.8 \\ 0.8 \end{pmatrix} \otimes \begin{pmatrix} 1.0 \\ 0.5 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.8 \times 1.0 \\ 0.8 \times 0.5 \\ 0.8 \times 0 \end{pmatrix} = \begin{pmatrix} 0.8 \\ 0.4 \\ 0 \end{pmatrix}$$

The sigmoid function is useful for gating in that it ensures that our values will always be between 0 and 1. The **forgetting gate** chooses which memories to keep and which memories to forget. The **selection gate** keeps our memories inside and selects only our predictions. The **ignoring gate** removes possibilities that may not be immediately relevant.

At this point we will assume that all of the neural networks that are involved have been trained. To explain how our RNN works, we will begin with the following example. In the diagram below, the network predicted **Doug** or **Jane** or **Spot**. It also sees **Doug**. The network now predicts **saw** and also makes a negative prediction for **Doug**. The ignoring and memory gates cause no changes. However the selection gate selects **saw** and removes **not Doug** as it has learned that names are not relevant in this case.



Next we consider the following case. Following saw our network predicts one of the three possible names. From our previous case, **saw** and **not Doug** were passed to the forgetting gate. The forgetting gate forgets **saw** and remembers **not Doug** as it has learned that it wants to keep any predictions having to do with names. The positive vote for **Doug** + the negative vote for **Doug** causes them to cancel each other out. Thus the network predicts Jane or Spot. Notice that here we have avoided the Doug saw Doug problem. Overall RNNs can be applied to anything involving sequential patterns. Examples of applications include text and speech.



4 Generative Models

4.1 Generative vs Discriminative Algorithms

So far we have been focusing on discriminative algorithms. **Discriminative algorithms** aim to classify input data. Given the features of a data instance, they predict a label or category to which the data belongs. Let us call the label y and the features x. Our neural network attempts to determine $p(y|x)$ or the probability of a label y given a set of features x.

So discriminative algorithms map features to labels. **Generative algorithms** essentially do the opposite. Instead of predicting a label given certain features, they attempt to predict features given a certain label. They aim to determine $p(x|y)$ or the probability of a set of features x given a label y. Note that generative models can also be used as classifiers. The point here is that they can be used for more than simply categorizing input data. We can summarize this discussion with the following two points.

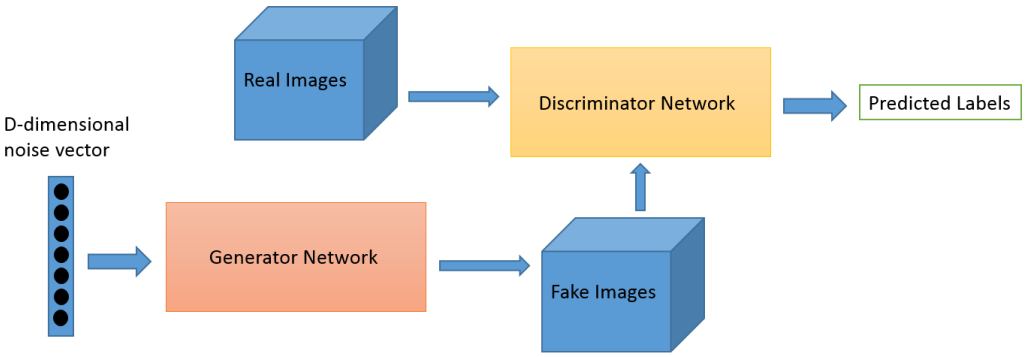
- Discriminative models learn the boundary between classes.
- Generative models model the distribution of individual classes.

4.2 Generative Adversarial Networks

GANs or **generative adversarial networks** are one example of a generative model. It consists of two neural networks: a generator and a discriminator. The **generator** generates new data instances while the **discriminator** evaluates them for authenticity. The discriminator essentially decides whether each instance of data it reviews belongs to the actual training set or not.

Let us consider the MNIST training set as an example. The goal of the discriminator when shown an instance from the true MNIST dataset is to recognize them as authentic. Meanwhile the generator is creating new images that it passes to the discriminator. It does so in the hopes that they too will be deemed authentic even though they are fake. The goal of the generator is to generate passable hand written digits or to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake. They steps completed by the GAN can be summarized as follows.

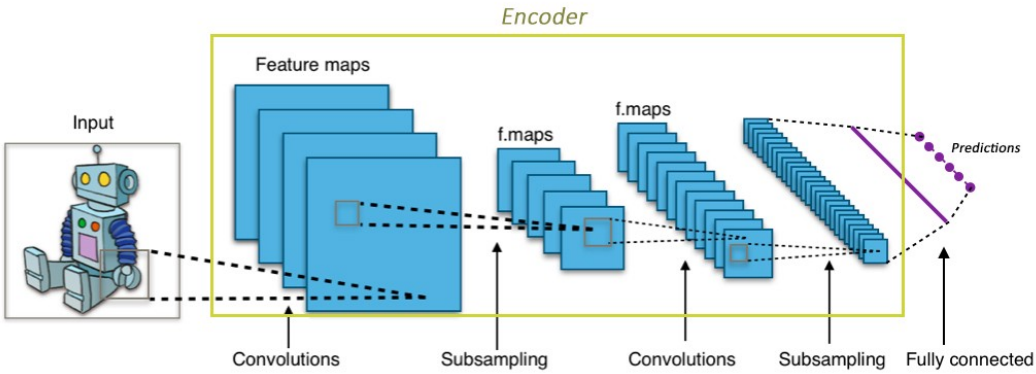
- The generator takes in random numbers and returns an image.
- This generated image is fed into the discriminator alongside a steam of images taken from the actual dataset.
- The discriminator takes in both real and fake images and returns probabilities with 1 representing a prediction of authenticity and 0 representing fake.



The discriminator network can be a standard feed forward neural network or convolutional network. Essentially it can be anything that serves as a binomial classifier. The generator network can be any generative model that will allow it to generate new images based on a set of input images. While the discriminator takes an image and downsamples to produce a probability, the generator takes a vector of random noise and upsamples it to create an image. Both neural networks are trying to optimize a different and opposing objective function or loss function. This is essentially an **actor critic model**.

4.3 Autoencoders

An autoencoder consists of two networks: an **encoder** and a **decoder**. An encoder network converts its input into a smaller dense representation. The decoder network is then responsible for converting this dense representation back into the original input. We have also seen encoders from our discussion on CNNs.



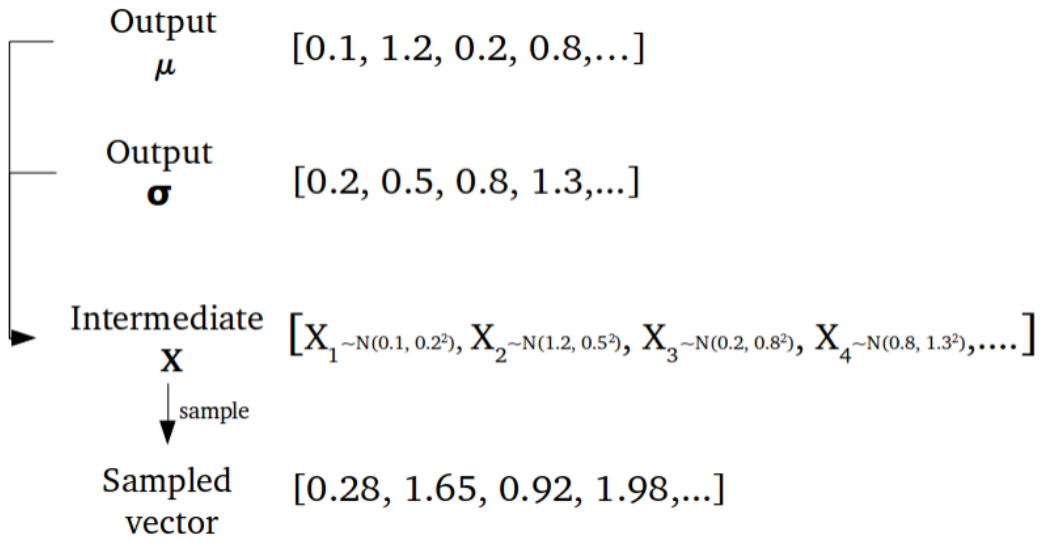
In the case of the CNN, this dense representation is then used by the fully connected classifier network to classify the image. The smaller dense representation produced by the encoder is called the **encoding**. In the case of an autoencoder, the decoder attempts to use the encodings generated by the encoder to reconstruct the original input. The network is usually trained as a whole. The loss function is usually either the **mean squared error** or **cross entropy** between the output and the input. Since it penalizes the network for creating outputs different from the input, it is commonly referred to as **reconstruction loss**. The encoder learns to preserve as much of the relevant information as possible while intelligently discarding irrelevant parts. The decoder learns to take the encoding and properly reconstruct it into a full image.

The fundamental problem with autoencoders is that the **latent space** that contains the encoded vectors may not be continuous or allow for easy interpolation. For example, consider training an autoencoder on the MNIST dataset. Visualizing the encodings from a 2D latent space reveals the formation of distinct clusters. This makes sense as distinct encodings for each image class makes it much easier for the decoder to decode them. This is fine if you're only replicating the same images. But when you're building a generative model, your goal is not to replicate the same image you put in. You want to randomly sample from the latent space or generate variations of an input image. If the space has discontinuities (gaps between clusters) and you sample/generate a variation from there, the decoder will simply generate an unrealistic output because the decoder has no idea how to deal with that region of the latent space.

4.4 Variational Autoencoders

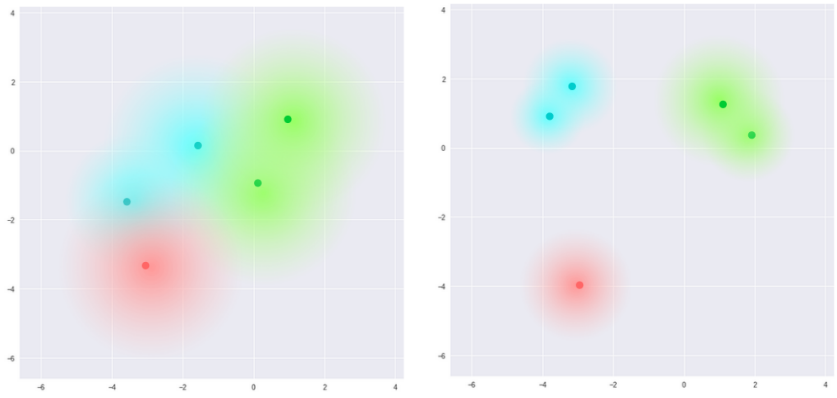
Variational autoencoders or **VAEs** have one fundamentally unique property that separates them from vanilla autoencoders. And it is this property that makes them so useful for generative modelling: Their latent spaces by design are continuous, allowing for easy random sampling and interpolation.

Rather than having the encoder output a single encoding vector of size n, the encoder now outputs two vectors of size n instead: a vector of means $\boldsymbol{\mu}$ and a vector of standard deviations $\boldsymbol{\sigma}$. They form the parameters of a vector of random variables of length n. Here the i^{th} element of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ represent the mean and standard deviation of the i^{th} random variable X_i from which we sample to obtain our sampled encoding.



This stochastic generation implies that equivalent inputs can have encodings that vary on every pass due to sampling. Our mean vector controls where the encoding of an input is centered while the standard deviation controls the area of the encoding. This ultimately allows the decoder to not just decode single encodings in the latent space but ones that slightly vary too as the decoder is exposed to a range of variations of the encoding of the same input during training.

The model's exposure to this local variation results in smooth latent spaces on a local scale. Ideally we want overlap between samples that are not very similar too. This will allow us to interpolate between classes. But since there are no limits on the possible values of μ and σ, the encoder can learn to generate very different μ for different classes while clustering them apart.



What we require

What we may inadvertently end up with

Our ideal scenario is to have encoding that are all as close as possible to each other while still being distinct. In order to force this, we introduce the KL divergence to the loss function. The KL divergence between two probability distributions simply measures how much they diverge from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters μ and σ to closely resemble that of the target distribution.