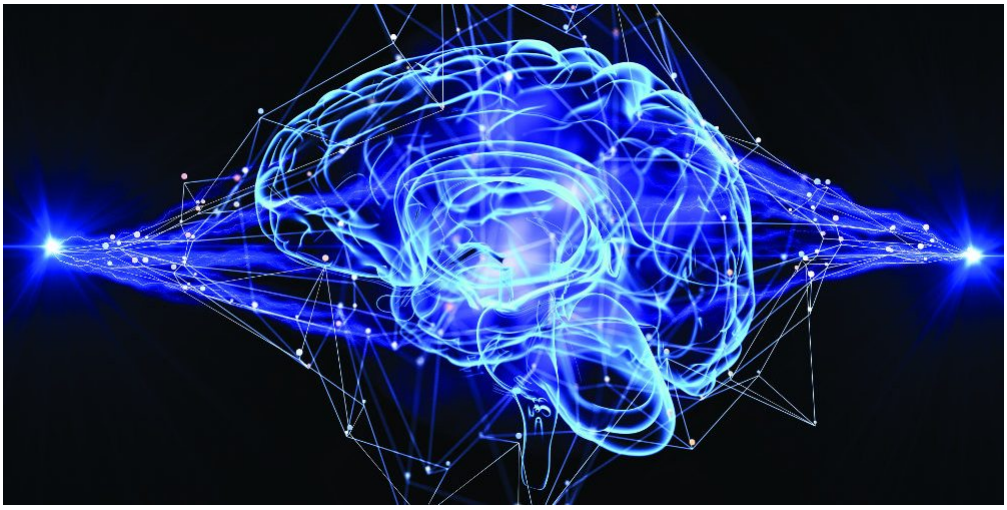
Neural Networks and Deep Learning Summer Report

June 7, 2020

P. Veda Pranav
190050094
Computer Science and Engineering
MENTOR : M. K. IPSIT



Contents

1 Introduction

What is machine learning? We probably use it dozens of times a day without even knowing it. When we do a web search in Google or Bing, that's machine learning. When Facebook or Apple's photo application recognizes our friends in our pictures, that's also machine learning. Each time we read our email and a spam filter saves us from having to wade through tons of spam, again, that's because our computer has learned to distinguish spam from non-spam email and that's machine learning. Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. **Machine learning focuses on the development of computer programs that can access data and use it learn for themselves.** One of the reasons I'm excited about this is the **AI**, or artificial intelligence problem i.e., building truly intelligent machines that can do just about anything that humans can do. Many scientists think the best way to make progress on this is through learning algorithms called neural networks, which mimic how the human brain works.

1.1 Definitions

The introduction given above gives us a broad view on machine learning. Even among machine learning practitioners, there isn't a well accepted definition of what is and what isn't machine learning. People tried to define it in different ways. Here are some:

- **Arthur Samuel(1959):**
  Field of study that gives computers the ability to learn without being explicitly programmed.This is an older, informal definition.

- **Tom Mitchell(1998):**
  A computer program is said to learn from experience **E** with respect to some task **T** and some performance **P**, if its performance on **T**, as measured by **P**, improves with experience **E**.

1.2 Types of Machine Learning

In general, any machine learning problem can be assigned to one of two broad classifications:

- **Supervised Learning**

- **Unsupervised Learning**

Some other types are **Reinforcement Learning** and **Recommender Systems**. We will look into **Reinforcement Learning** later.

Supervised Learning

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output. Supervised learning problems are categorized into **"regression"** and **"classification"** problems. In a **regression** problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a **classification** problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Unsupervised Learning

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results. Example:

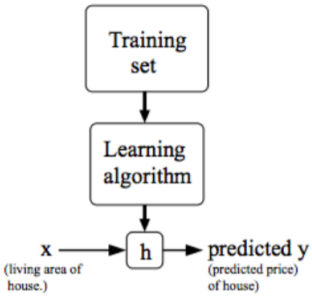
Clustering :
Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.

Non-clustering :
The **Cocktail Party Algorithm**, allows you to find structure in a chaotic environment(i.e identifying individual voices and music from a mesh of sounds at a cocktail party).

2 Linear Regression

2.1 Model Representation

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables, also called input features, and $y^{(i)}$ to denote the "output" or target variable that we are trying to predict. A pair $(x^{(i)}, y^{(i)})$ is called a training example, and the dataset that we will be using to learn - a list of m training examples is called a "training set". The superscript "(i)" has nothing to do with exponentiation. We will also use X to denote the space of input values, and Y to denote the space of output values. To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function h : X $\leftarrow$ Y so that h(x) is a "good" predictor for the corresponding values of y. This function h is called a **hypothesis**.

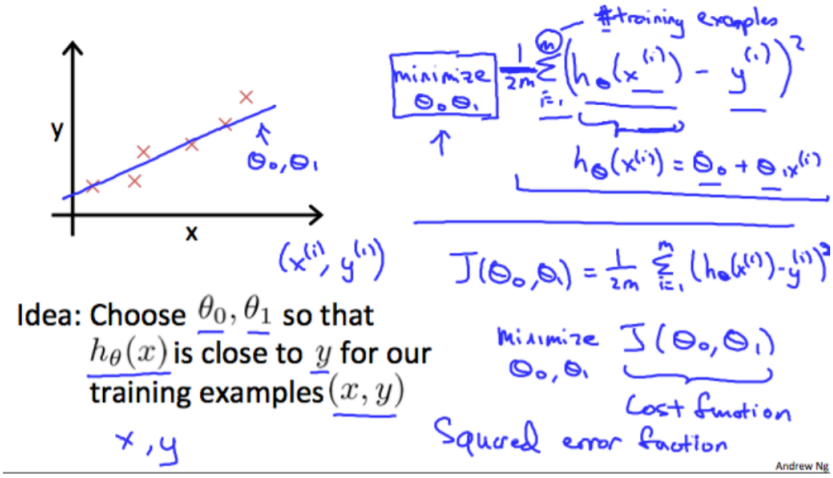


2.2 Cost Function

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.

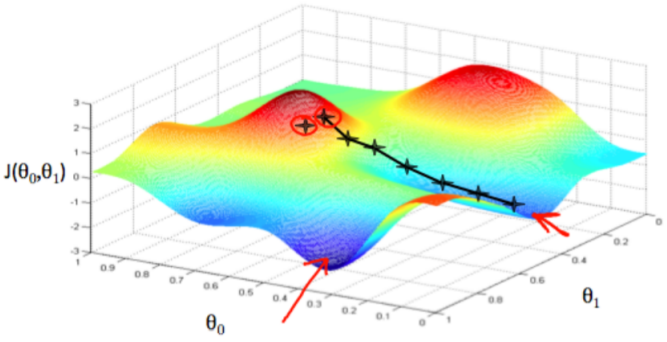
$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\overline{y_i} - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$$

To break it apart, it is $\frac{1}{2}\overline{x}$ where $\overline{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value. This function is otherwise called the "Squared error function". The mean is halved as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.



2.3 Gradient Descent

We have a hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function.That's where gradient descent comes in. Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

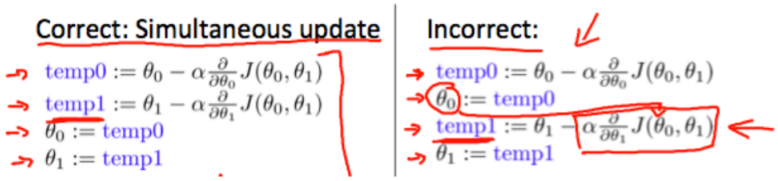


We are successful when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph. We do this by taking the derivative of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step

is determined by the parameter α, whiich is called the **learning rate**.The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$. Depending on where one starts on the graph, one could end up at different points.
The gradient descent algorithm is: repeat until convergence:

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

where j = 0,1 represents the feature index number.
At each iteration j, one should simultaneously update the parameters $\theta_0, \theta_1, ...\theta_n$. Updating a specific parameter prior to calculating another one on the j^{th} iteration would yield to a wrong implementation.

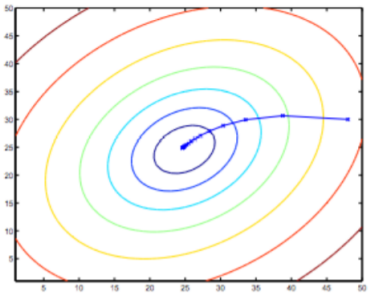


When specifically applied to the case of linear regression, gradient descent looks like :
repeat until convergence:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$$

with usual notations.This method looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, which was initialized at (48, 30). The x's in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through as it converged to its minimum.

3 Multivariate Linear Regression

Linear regression with multiple variables is also known as "multivariate linear regression".

3.1 Multiple Variables

We follow the following notation :

$$\begin{aligned} x_j^{(i)} &= \text{value of j in the } i^{th} \text{ training example} \\ x^{(i)} &= \text{the input (features) of the } i^{th} \text{ training example} \\ m &= \text{the number of training examples} \\ n &= \text{the number of features} \end{aligned}$$

The multivariate form of the hypothesis function accommodating these multiple features is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_n x_n$$

The above equation can be vectorized for one training example, as follows:

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 & ... & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

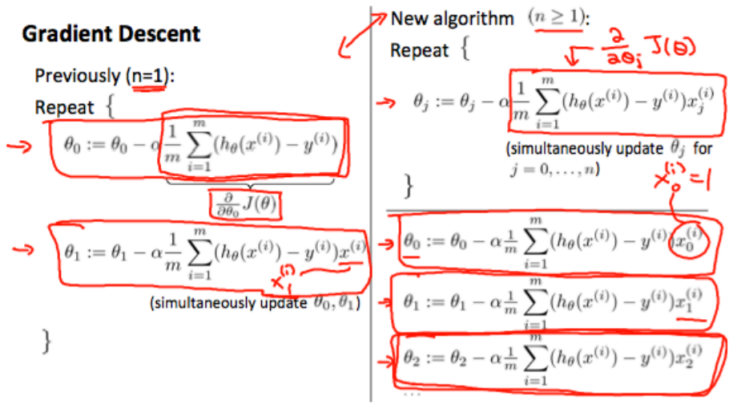
3.2 Gradient Decent For Multiple Variables

The gradient descent equation itself is generally the same form, we just have to repeat it for our 'n' features:

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j := 0...n}$$

}



3.3 Features Scaling and Mean Normalization

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate ineffectively down to the optimum when the variables are very uneven.
Feature scaling involves dividing the input values by the range(i.e. the maximum value minus the minimum value)of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero.

$$x_i := \frac{x_i - \mu_i}{s_i}$$

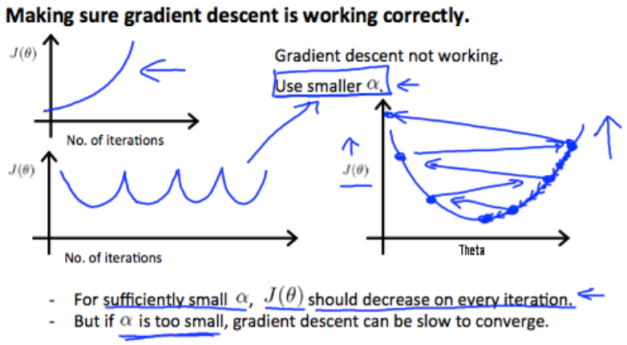
where μ_i is the "average" of all the values for the feature (i) and s_i is the range of values or the standard deviation.

3.4 Learning Rates

Debugging gradient descent : Plot $J(\theta)$ vs number of iterations of gradient descent. If $J(\theta)$ ever increases, we have to decrease α.

Automatic Convergence Test : Declare convergence if $J(\theta)$ decreases by less than ϵ in one iteration, where ϵ is some small value such as 10^{-3}. However in practice it's difficult to choose this threshold value.

It has been proved that if learning rate α is sufficiently small, then $J(\theta)$ will decrease on every iteration.



To summarize:

- if α is too slow : slow convergence
- If α is too large : may not decrease on every iteration and thus may not converge.

3.5 Modification of Features

We can improve our features and the form of our hypothesis function in a couple different ways. We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 x_2$.

3.6 Polynomial Regression

We can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function(or any other form) by creating additional features. Feature scaling becomes important in this case.

4 Computation of Parameters using Analytical Methods

This is another method to minimize our cost function $J(\theta)$. In the Normal Equation method, we will minimize J by explicitly taking its derivatives with respect to the θ_j 's, and setting them to zero. The normal equation formula is given as follows:

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

There is no need to do feature scaling for normal equation.The following is a comparision of gradient descent and the normal equation:

Gradient Descent	**Normal Equation**
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$, need to calculate inverse of $X^T X$
Works well when n is large	Slow if n is very large

Normal Equation Noninvertibility: If $X^T X$ is **noninvertible**, the common causes might be having :

- Redundant features, where two features are very closely related(i.e. they are linearly dependent)
- Too many features(e.g. $m \leq n$). In this case, delete some features or use "regularization".

5 Classification

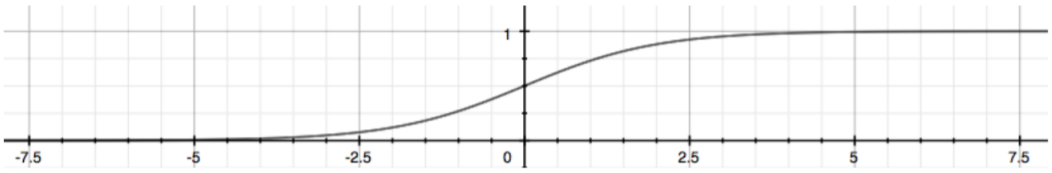
5.1 Introduction

Classification is also called **Logistic Regression** even if the term regression is misleading. The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. First we will focus on the **binary classification** problem in which y can take on only two values, 0 and 1. This will be generalized later to multi-class classification problem. Hence, $y \in \{ 0, 1 \}$. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols "-" and "+". Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the label for the training example.

5.2 Hypothesis Representation

Intuitively, it doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{ 0, 1 \}$. This is accomplished by plugging $\theta^T x$ into the Logistic Function. Our new form uses the "Sigmoid Function," also called the "Logistic Function":

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



$h_\theta(x)$ will the **probability** that our output is 1.

$$h_\theta(x) = P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta)$$
$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1$$

5.3 Descision Boundary

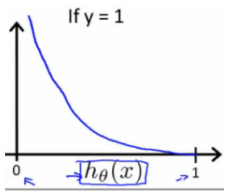
The **descision boundary** is the line or the curve that seperates the area where $y = 0$ and where $y = 1$.It is created by our hypothesis function.

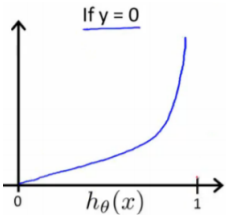
$$\theta^T x \geq 0 \rightarrow h_\theta(x) \geq 0.5 \rightarrow y = 1$$
$$\theta^T x < 0 \rightarrow h_\theta(x) < 0.5 \rightarrow y = 0$$

5.4 Cost Function and Gradient Descent

We cannot use the same cost function that we use for linear regression because the **Logistic Function** will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.Instead, our cost function for logistic regression looks like:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \mathbf{Cost}(h_\theta(x^{(i)}), y^{(i)})$$
$$\mathbf{Cost}(h_\theta(x), y) = -y log(h_\theta(x)) - (1 - y) log(1 - h_\theta(x))$$





Writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression.A vectorized implementation is:

$$h = g(X\theta)$$
$$J(\theta) = \frac{1}{m}(-y^T log(h) - (1-y)^T log(1-h))$$

For gradient descent, a vectorized implementation is:
repeat $\{$

$$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \overline{y})$$

$\}$
Notice that this algorithm is identical to the one we used in linear regression. But the hypothesis is different.

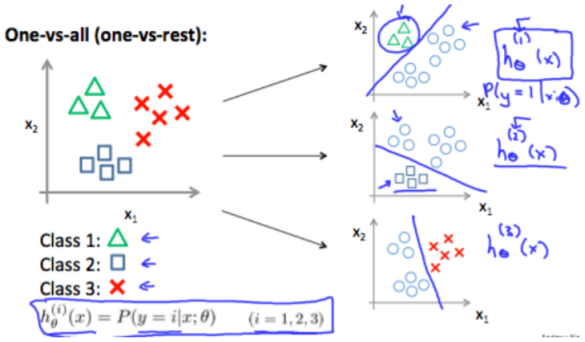
5.5 Advanced Optimisation

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. Octave provides them in libraries. We first need to provide a function that evaluates both cost function and it's partial derivatives. If we pass this function to the advanced optimization algorithms, we get the optimum values of θ directly.

5.6 Multiclass Classification: One-vs-all

Let's say we have n classes. We can deal this by dividing our problem into $n+1$ binary classification problems; in each one, we predict the probability of occurence of a particular class. We are basically choosing one class and

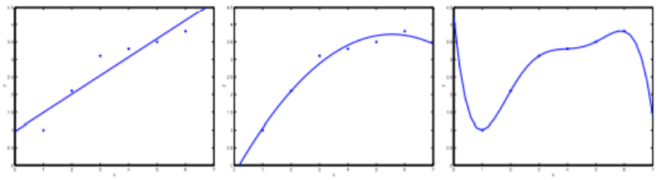
then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.



6 Regularization

6.1 The Problem of Overfitting

Consider the problem of predicting y from $x \in R$. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th} order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**—in which the data clearly

shows structure not captured by the model—and the figure on the right is an example of **overfitting**.
Underfitting or high bias, is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. At the other extreme, **overfitting**, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.
This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1. Reduce the number of features:
    - Manually select which features to keep.
    - Use a model selection algorithm.
2. Regularization:
    - Keep all the features, but reduce the magnitude of parameters θ_j .
    - Regularization works well when we have a lot of slightly useful features.

6.2 Regularized Linear Regression

To reduce the magnitude of parameters, we modify our cost function as follows:

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

Here, λ is called the regularization parameter. It determines how much the costs of our θ parameters are inflated. Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If λ is chosen to be too large, it may smooth out the function too much and cause underfitting.

Gradient Descent

We will modify our gradient descent function to seperate out θ_0 from the rest of the parameters because we don't want to penalize θ_0. repeat $\{$

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

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

Normal Equation

The formula for optimum θ will get modified as follows:

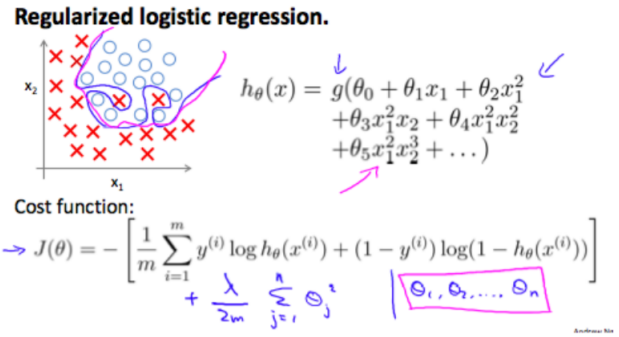
$$\theta = (X^T X + \lambda L)^{-1} X^T y$$

where $L = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$

L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension $(n + 1)*(n + 1)$. Recall that if $m < n$, then $X^T X$ is non-invertible. However, when we add the term λL, then $X^T X + \lambda L$ becomes invertible.

6.3 Regularized Logistic Expression

We can regularize logistic regression in a similar way that we regularize linear regression. As a result, we can avoid overfitting. The following image shows how the regularized function, displayed by the pink line, is less likely to overfit than the non-regularized function represented by the blue line:



6.3.1 Cost Function and Gradient Descent

Recall that our cost function for logistic regression was:

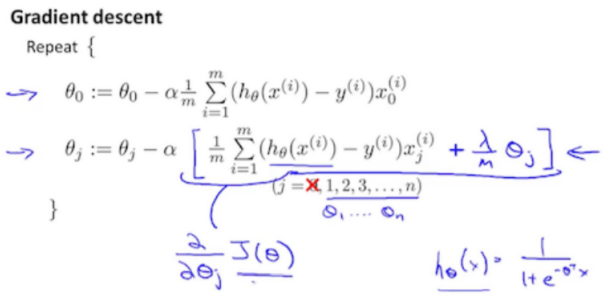
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \textbf{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\textbf{Cost}(h_\theta(x), y) = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$$

We can regularize this equation by adding a term to the end:

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

The second sum, $\sum_{j=1}^{n}\theta_j^2$ **means to explicitly exclude** the bias term, θ_0 i.e.,the θ vector is indexed from 0 to n (holding $n+1$ values, θ_0 through θ_n), and this sum explicitly skips θ_0, by running from 1 to n, skipping 0. Thus, when computing the equation, we should continuously update the two following equations:



7 Neural Networks

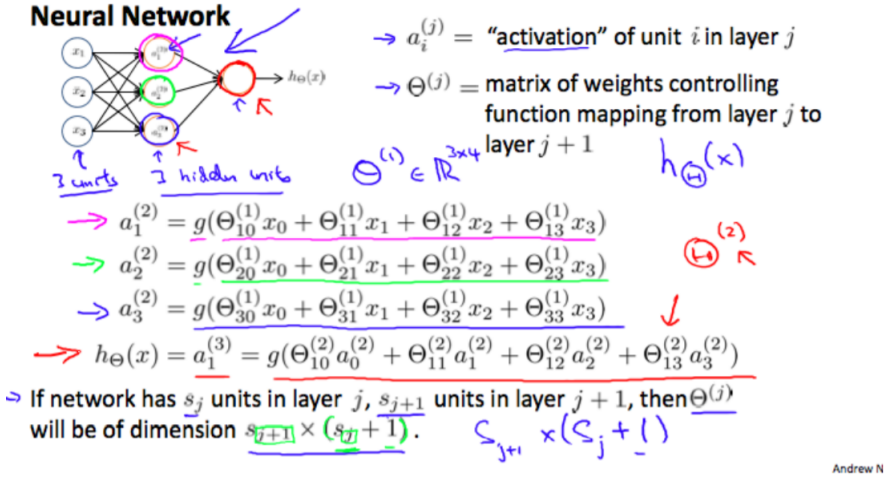
Neural networks are pretty old algorithms that was originally motivated by the goal of having machines that can mimic brain. This is generally used for non-linear hypotheses. A most common example where this is used is the problem of computer vision-object classification. These are computationally expensive. We will also be looking into the modern version of neural networks later.

7.1 Model Representation

Neural networks are developed as simulating the networks of neurons in the brain. At a very simple level, neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**). In our model, our dendrites are like the input features $x_1...x_n$, and the output is the result of our hypothesis function.In this model our x_0 input node is sometimes called the "bias unit", which is always equal to 1. We use the same logistic function as in classification,yet it is sometimes called a sigmoid **activation** function. Here we are using sigmoid to introduce non-linearity. Also the θ parameters are sometimes called "weights".
Our input nodes, also known as the **input layer**, go into a series of intermediate layers called the **hidden layers**, and finally the layer in which we get our output is called the **output layer**. This architecture is somewhat similar to the computational graphs. The following notations are used :

$a_i^{(j)} =$ **activation of unit** i **in layer j**
$\Theta^{(i)} =$ **matrix of weights controlling function mapping from layer** j **to layer** $j+1$

If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(i)}$ will be of dimension $s_{j+1} * (s_j + 1)$



For a vectorized implementation, we will define a new variable $z_k^{(j)}$ that encompasses the parameters going into the sigmoid function.

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

Then we will have :

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

We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This element will be $a_0^{(j)}$ and will be equal to 1. Notice that in this last step, between layer j and layer $j+1$, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.
To classify data into multiple classes, we let our hypothesis function return a vector of values, the size being wqual to the number of classes. The maximum value among the components is used to predict the class for a new test case.

7.2 Cost Function

The following convention will be followed :

$$\begin{aligned} \text{L} &= \text{total number of layers in the network} \\ s_l &= \text{number of units(not counting bias unit) in layer } l \\ \text{K} &= \text{number of output units/classes} \\ h_\theta(x)_k &= \text{hypothesis that results in the } k^{th} \text{ output} \end{aligned}$$

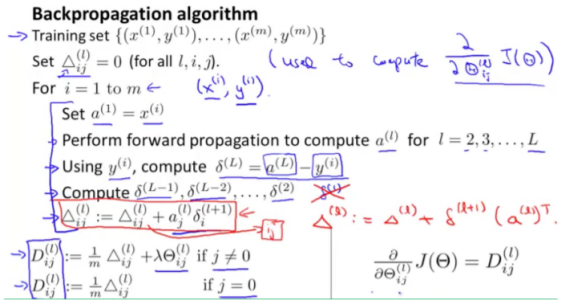
The cost function for neural networks will be a generalization of the one used for logistic regression. For neural networks, it will be slightly complicated:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

The double sum simply adds up the logistic regression costs calculated for each cell in the output layer. The triple sum simply adds up the squares of all the individual Θs in the entire network. The i in the triple sum does **not** refer to training example i.

7.3 Backpropagation Algorithm

"Backpropagation" is a neural-network terminology for minimizing our cost function, just like the gradient descent in logistic and linear regression. We will compute the partial derivatives of $J(\Theta)$ as follows :



For each training example, after performing forward propagation, we compute the errors $\delta^{(l)}$ at each layer by using the concept of chain rule from calculus. So our error for the last layer is simply the difference between the actual output and the value of our hypothesis. The errors of the remaining layers can be calculated as follows :

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \; .* \; a^{(l)} \; .* \; (1 - a^{(l)})$$

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with the derivative of the activation function g evaluated with the input values given by $z^{(l)}$.

$$g'(z^{(l)}) = a^{(l)}. * (1 - a^{(l)})$$

As shown in the figure, the derivative of the non regularized part at each layer is stored in the Δ matrix. After considering regularization, the partial derivatives are stored in the vector D where :

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

We then use advanced optimization techniques provided by the software to compute the optimum values of the weights.

7.4 Gradient Checking

Gradient checking is used to check whether backpropagation is working as intended. We can approximate the derivative as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

where ϵ is a small positive number(10^{-4}).
By comparing the approximate gradient with the gradient computed using backprop, we can verify. **Once** verification is done, approximate gradient is not needed anymore and hence not computed. Backprop is used for learning.

Note: Always initializing the weights to zero doesn't work. This may lead to gradient killing.So we generally initialize the weights to a random value between $[-\epsilon, \epsilon]$. This ϵ is unrelated to the one used in gradient checking.

7.5 Overall Procedure and Conclusion

1. Pick a suitable network architecture.
2. Randomly initialize weights.
3. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
4. Implement the cost function.
5. Implement the backpropagation to compute partial derivatives.
6. Use gradient checking to confirm that backprop works. Then disable gradient checking.
7. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in θ.

When we perform forward and back propagation, we loop on every training example. This is the content that was developed initially.

8 A Modern Approach on Neural Networks

Here, we try to explore new ideas in neural networks including implementation by **TensorFlow**.
TensorFlow is the most widely used language to represent machine learning today. Given below is a TensorFlow implementation of a dense layer (any hidden layer) in the neural network.

```
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, input_dim, output_dim):
    super(MyDenseLayer, self).__init__()

    # Initialize weights and bias
    self.W = self.add_weight([input_dim, output_dim])
    self.b = self.add_weight([1, output_dim])

  def call(self, inputs):
    # Forward propagate the inputs
    z = tf.matmul(inputs, self.W) + self.b

    # Feed through a non-linear activation
    output = tf.math.sigmoid(z)

    return output
```

If you observe the syntax is mostly python. We just have to import a package called tensorflow. Here we imported it as tf. If we have a hidden layer with two units or neurons, below is the implementation:

```
import tensorflow as tf

layer = tf.keras.layers.Dense(
   units=2)
```

Below is the implementation for a sequential neural network containing a hidden layer of n units and an output layer with two outputs:

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

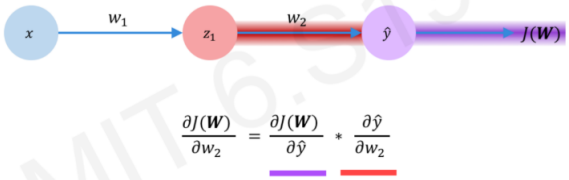
The cost function is also called the "*Loss*" of our network. The cost function we learned for logistic regression is also called **Binary Cross Entropy Loss**, and is feeded in the tensorflow package under this name.

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

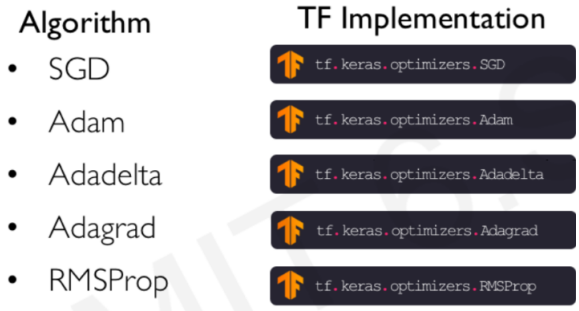
The tensorflow implementation for gradient descent is

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
   with tf.GradientTape() as g:
      loss = compute_loss(weights)
      gradient = g.gradient(loss, weights)

   weights = weights - lr * gradient
```

The Backpropagation Algorithm we learned, is a consequence of the chain rule.



Until now, we have dealt with the learning rate as a number which can be found after various trials and is fixed, once found. But now using tensorflow we can implement an "*Adaptive Learning Rate*". Here the learning rate changes according to the algorithm we used for finding the gradient descent.



Up until now, we have used a single set of values for the weights or θs (random initialization), we could also be picking a *batch* of points to find the gradient and average over the values as this will be more effective.
While doing regularization, we have used a λ term to regulate the values of the weights. We can also drop out some weights (making them zero) randomly every time. This also is very effective.

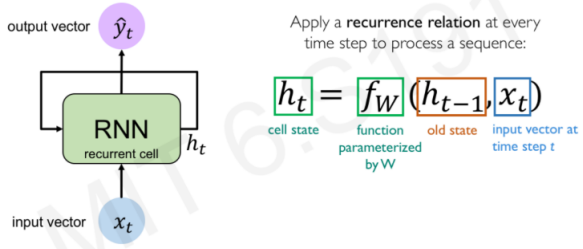
```
tf.keras.layers.Dropout(p=0.5)
```

This implementation drops out weights in each layer with a probability of 0.5. We could also use the concept of early stopping where we plot the cost function with a number of iterations using the training data and the testing data. There comes a point when this plot starts to diverge. This is the point where our network becomes too accurate for our training set and less accurate for the testing set (*Overfitting*). So we can stop the process of learning here.

8.1 RNN

The neural networks we learned so far, take a single input to give a single output. But that's generally not the case in various real life problems. We

may be given a sequence of inputs and asked to predict the next thing in the sequence. This problem requires our network to be able to generate outputs at every stage or take every new input and this also requires a prior knowledge of the previous inputs. The neural networks we learned won't be able to do that.
So, we introduce the **RNNs** or **Recurrent Neural Networks** which will precisely do this. Here is a representation of RNNs:

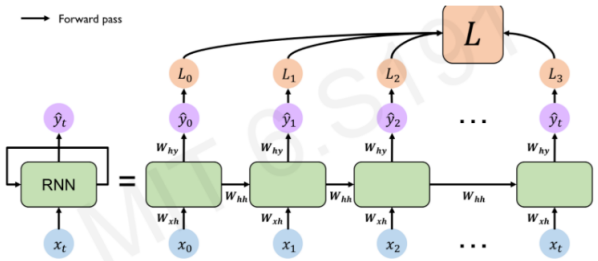


$$\textbf{Output Vector :}\ \hat{y}_t = W_{hy}^T h_t$$

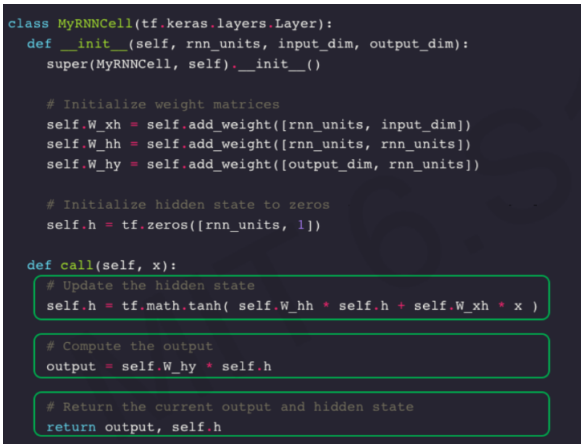
$$\textbf{Update Hidden State :}\ h_t = tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

$$\textbf{Input Vector :}\ x_t$$

Here, the weights are constant for every input. The hyperbolic tangent function introduces the *non-linearity*.



Here we are calculating the cost function for every output and summing them up to get the total cost function. Below is an implementation of RNNs.



```
class MyRNNCell(tf.keras.layers.Layer):
  def __init__(self, rnn_units, input_dim, output_dim):
    super(MyRNNCell, self).__init__()

    # Initialize weight matrices
    self.W_xh = self.add_weight([rnn_units, input_dim])
    self.W_hh = self.add_weight([rnn_units, rnn_units])
    self.W_hy = self.add_weight([output_dim, rnn_units])

    # Initialize hidden state to zeros
    self.h = tf.zeros([rnn_units, 1])

  def call(self, x):
    # Update the hidden state
    self.h = tf.math.tanh( self.W_hh * self.h + self.W_xh * x )

    # Compute the output
    output = self.W_hy * self.h

    # Return the current output and hidden state
    return output, self.h
```

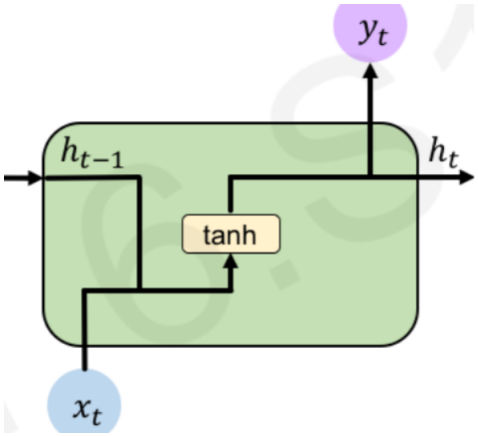
Here is inbuilt RNN in the tensorflow package.

```
tf.keras.layers.SimpleRNN(rnn_units)
```

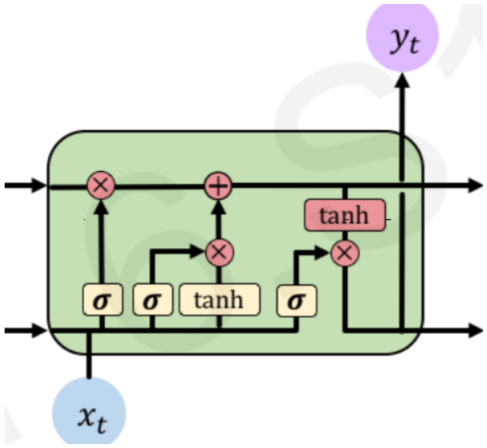
We have a similar algorithm as Backpropagation here, which is called *Back-propagation through Time*.
Here when we calculate gradient of w.r.t any parameter, we need to compute the gradients w.r.t all following parameters and we may face two problems:

- **Exploding Gradient** : This happens when many of the gradients are large values and the resulting gradient may explode.
- **Vanishing Gradient** : This happens when many of the gradients are small values and the resulting value may vanish.

Vanishing Gradient makes our network to be short-term dependent or doesn't care about inputs received way back.
We can avoid these problems by using the ReLU function as our activation function as it's derivative becomes 1 when the variable becomes greater than zero or by initializing the weights to the identity matrix. We can also use gated cells which is a more complex form of RNNs. One example is the **Long Short Term Memory**(LSTMs). This is a representation of a normal RNN cell :



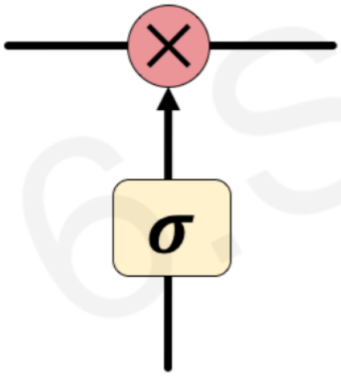
This is a representation of LSTM cell :



Implementation of LSTMs :

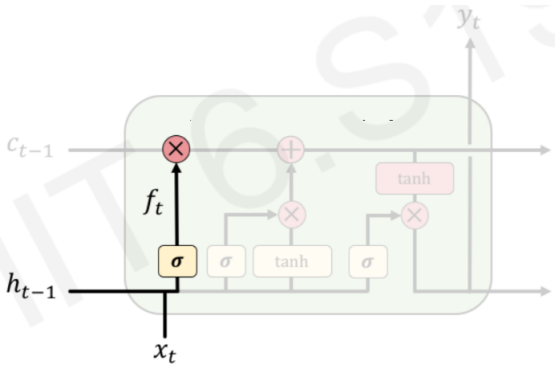
```
tf.keras.layers.LSTM(num_units)
```

LSTM modules contain computational blocks that control information flow. LSTM cells are able to track information throughout many timesteps. Information is added or removed through structures called gates.

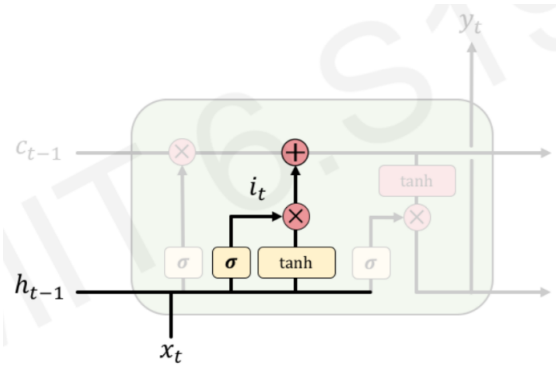


Gates optionally let information through, for example via a sigmoid neural net layer and pointwise multiplication. LSTMs work in four ways.

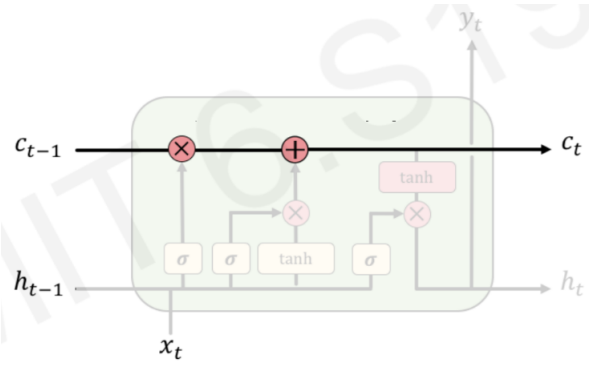
- **Forget** : LSTMs forget irrelevant parts of the previous state.Gates optionally let information through, for example via a sigmoid neural net layer and pointwise multiplication. LSTMs work in four ways.



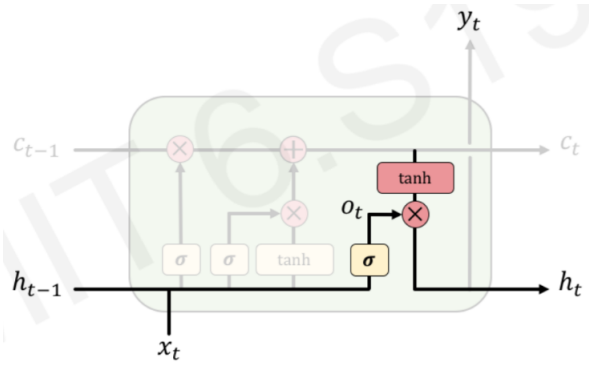
- **Store** : LSTMs store relevant new information into the cell state.



- **Update** : LSTMs selectively updates cell state values.



- **Output** : The output gate controls what information is sent into the next timestep.

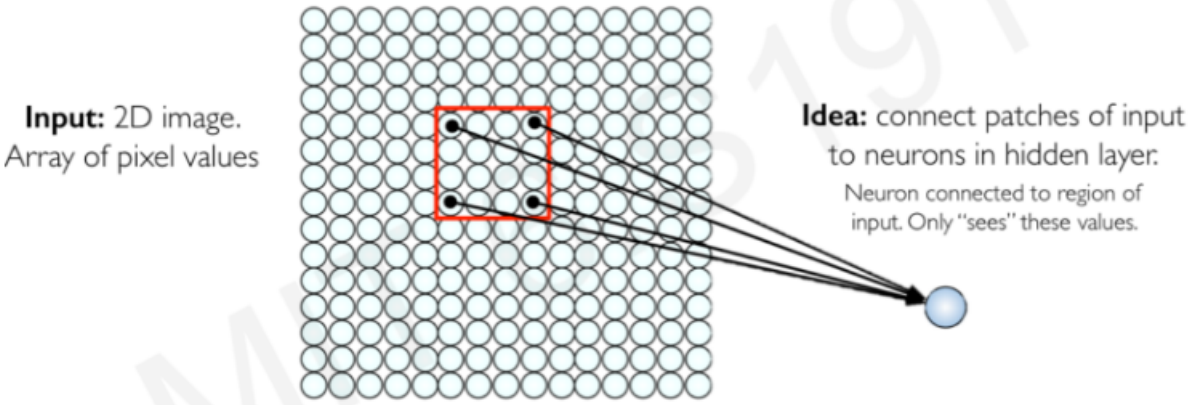


So, LSTMs maintain a separate cell state apart from what is outputted and Backpropagation through time happens with uninterrupted gradient flow.

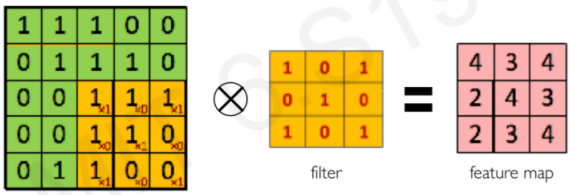
8.2 CNN

If we use normal neural networks for the purpose of vision, the computer won't be able recognize the image spatially i.e, the features present in the image. We can use normal neural networks or even simple logistic regression to identify preliminary images like digits etc, but can't be used as a tool to view any kind of image. So, we use **Convolutional Neural Networks** or **CNNs**.
Here, we consider the image as a *2D* matrix of pixels, and we pass filters spatially to it so that we can advance to the next layer in our neural network.

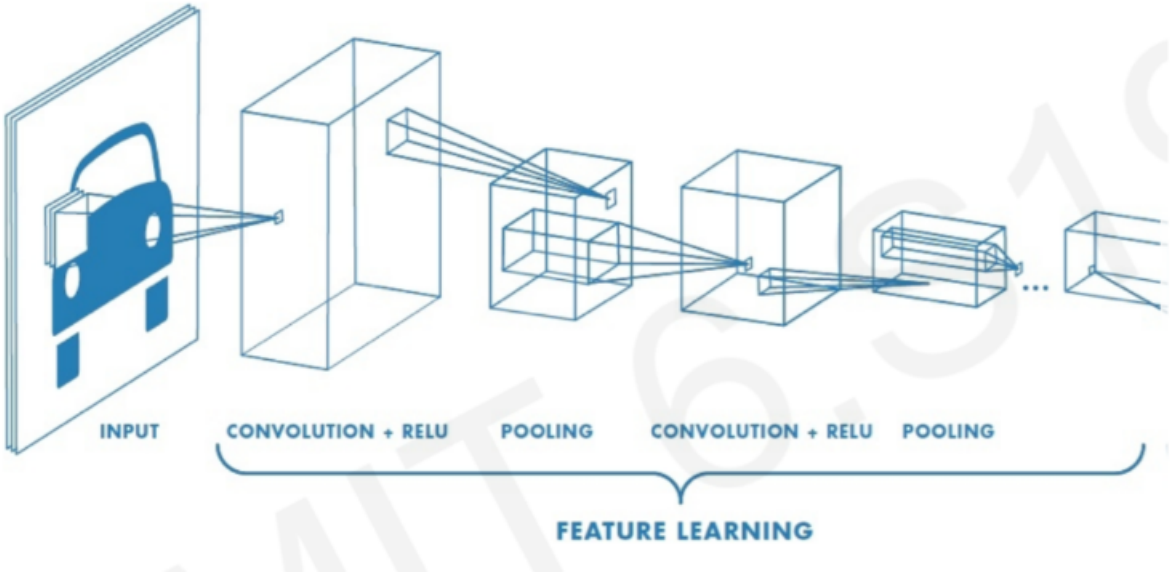
The filters represent features like eyes, nose, mouth, etc.



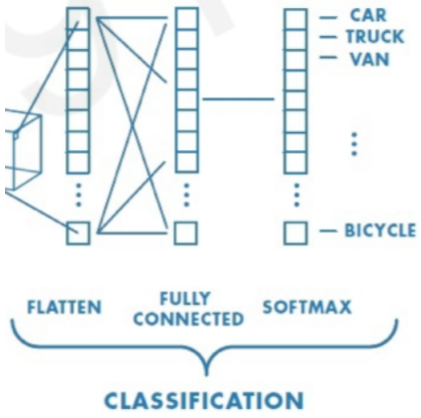
The pixels enclosed by the red box are collectively known as *patches*. We connect patches in the input layer to a single neuron in the subsequent layer. We use a sliding window (the red box) to define connections. The filters are the *weights* for these patches. This patchy operation is called convolution.
Example:
Here we slide the 3x3 filter over the input image, element- wise multiply and add the outputs.



We can use a bias here, if we want.
In CNNs, there are multiple features or filters and hence we visualize each layer of the network as a *3D* layer with each layer corresponding to one feature, stacked up on each other.
Dimensions = h x w x d
Where h and w are spatial dimensions and d = no. of features.
After every convolution operation, we apply non-linearity to it.
We also apply pooling which is compression of each layer into a smaller *3D* box by suitable operations. Putting it together:



And continuing this to our normal neural network :



Implementation:

```python
import tensorflow as tf

def generate_model():
    model = tf.keras.Sequential([
        # first convolutional layer
        tf.keras.layers.Conv2D(32, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # second convolutional layer
        tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # fully connected classifier
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1024, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')  # 10 outputs
    ])
    return model
```

8.3 Generative Modelling

The difference between supervised and unsupervised learning is that the output in supervised learning could be differentiated into different classes but there is no such differentiation in unsupervised learning. The training model of unsupervised learning learns the hidden or underlying features in the data to differentiate them.

Generative Modelling takes an input training sample from some distribution and learns a model that represents that distribution. We can then create new samples in that distribution.

We can use this model for debiasing purposes which includes better identification of faces, being able to identify extreme cases while self-driving etc.

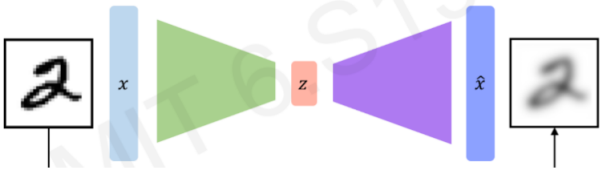
There are two models which is able to identify latent/hidden features in data :

- Autoencoders and Variational Autoencoders.
- Generative Adversarial Networks (GANs)

8.3.1 Autoencoders

It's an unsupervised approach for learning a *lower dimensional feature*(hidden feature) representation from unlabeled data.
It contains two parts :

- The Encoder
- The Decoder

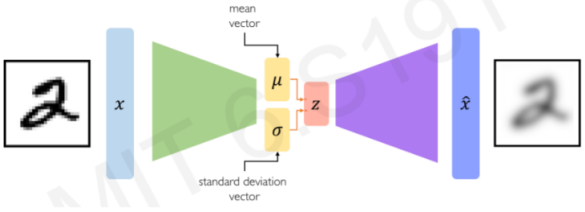


Encoder learns mapping from the data x into a lower dimensional latent space z. Decoder learns mapping back from latent z, to a reconstructed observation $\hat{x}$
The loss function here which doesn't use any labels:

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Higher dimensionality of z will give better results because different dimensions of z encodes different interpretable features. Bottleneck hidden layer forces the network to learn a compressed latent representation. Reconstruction loss forces the latent representation to capture as much information about the data as possible.

8.3.2 Variational Autoencoders



Variational autoencoders are a probabilistic twist on autoencoders. Sample from the mean and standard deviation to compute latent sample. **In a nutshell, a VAE is an autoencoder whose encodings distribution is regularised during the training in order to ensure that its latent space has good properties allowing us to generate some new data**.

Moreover, the term "variational" comes from the close relation there is between the regularisation and the variational inference method in statistics. Here the encoder computes the probability distribution $p_\phi(z|x)$. The decoder computes another probability distribution $q_\theta(x|z)$. A point from the latent space is sampled from the distribution of inputs encoded on the latent space. The sampled point is decoded and the reconstruction error can be computed. Finally, the reconstruction error is back propagated through the network.
The loss function:

$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term})$$

The reconstruction loss is the same as before. The regularization term is given by KL-divergence between the two distributions.

$$D(p_\phi(z|x) \parallel p(z)) = -\frac{1}{2}\sum_{j=0}^{k-1}(\sigma_j + \mu_j^2 - 1 - log\sigma_j)$$

Here, $p(z)$ is called a prior.

$$p(z) = \mathcal{N}(\mu = 0, \sigma^2 = 1)$$

This is a common choice of a prior as it encourages encodings to distribute encodings evenly around the center of the latent space and penalizes the network when it tries to cheat by clustering points in specific regions(i.e memorizing data).
But there is a problem in all this. We cannot backpropagate through sampling layers. Hence, we reparametrize the latent space, z, as $z = \sigma + \mu \odot \epsilon$. Where μ and σ are fixed vectors and epsilon is a random constant from the prior distribution.We enforce diagonal prior on the latent variables to encourage independence of features.

8.3.3 GAN

GANs are a way to make a generative model by having two neural networks compete with each other. The generator turns noise into an imitation of the to try to trick the discriminator.
The discriminator tries to identify real data from fakes created by the generator. Hence, by competing with each other, they both improve.
Then, we can create authentic fake data.

8.4 Reinforcement Learning

Here, we have state-action pairs and we will have rewards for every action if it is desirable. Our goal is to increase the total rewards which will mean that our

model is good. Here, we have an *agent* who takes actions in the environment and the set of all possible actions an agent can make in the *environment* is called the **action space**. We train the model based on observations of the environment after taking actions. A *state* is a situation which the agent perceives. The reward is the feedback that measures the success or failures of the agent's action. We use the discounted total reward to train the model:

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i$$

Here, the γ is less than one. We do this because we require immediate rewards rather than late rewards. Here, we also define a Q-function

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

This is the expected total future reward an agent in state s, can receive by executing a certain action a. Also we define a policy function $\pi(s)$ which keeps track of all possible Q-functions and tells us the max of them required for training.

$$\pi^*(s) = \underset{a}{\text{argmax}} Q(s, a)$$

Now, reinforcement learning is divided into two parts :

- **Value Learning**: We use the Q-function to train the model.
- **Policy Learning**: We use the policy function to train the model.

8.4.1 Value Learning

In value learning, our neural network will output the preferable action and it's reward as the Q-function. The loss function here is:

$$\mathcal{L} = \mathbb{E}\left[\left\|(r + \gamma \underset{a'}{\text{max}} Q(s', a')) - Q(s, a)\right\|^2\right]$$

There are some complexities for this type of learning. This model can model scenarios where the action space is discrete and small and also this model cannot handle continuous action spaces. Policy is deterministically computed from the Q-function by maximising the reward meaning it cannot learn stochastic policies.

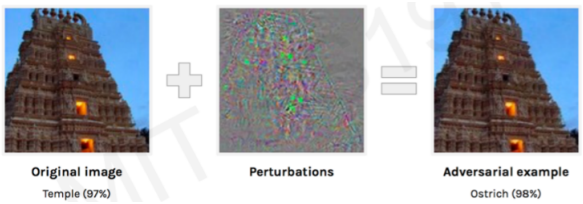
8.4.2 Policy Learning

In policy learning, we directly optimize the policy and outputting the policy. The advantage of this is that we can have a distribution of policies in case of continuous action spaces. The method of training this model is recording all states, actions and rewards for a particular agent and increasing the probability of actions that resulted in high reward and also decreasing probability of actions that resulted in low reward. The loss function here is:

$$\textbf{loss} = -logP(a_t|s_t)R_t$$

8.5 Limitations of Neural Networks

Neural networks are great function approximators. But if we somehow randomize the labels of the training data, the neural network will be trained in a wrong manner and there's no way it can identify that. Example:



Here, the first image is the original and the network identifies it correctly. The third image is made after adding perturbations which may be nothing for a human eye but, the model predicted as an ostrich which is very wrong. So it is poor at representing uncertainty (how do you know what the model knows?)

8.5.1 Bayesian Deep Learning

Network tries to learn output, y, directly from raw data, x. Bayesian neural networks aim to learn a posterior over weights.

$$P(W|X,Y) = \frac{P(Y|X,W)P(W)}{P(Y|X)}$$

This model evaluates T stochastic forward passes through the network W. Dropout as a form of stochastic sampling and we calculate mean and variance for the weights. This model can be used to uncertainty in the result of the model.

8.5.2 Evidential Deep Learning

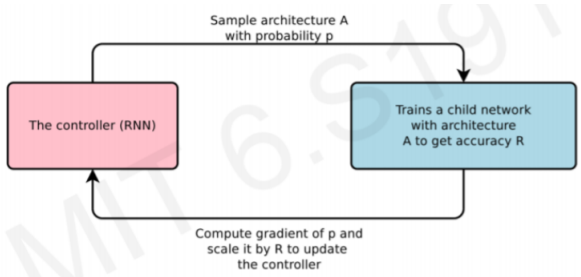
This directly learns the underlying uncertainties using evidential distributions. This also indicates the uncertainty in the result.

8.5.3 Multi-Task Learning

We can have different neural networks for different purposes and have loss which is the sum of all these losses.

8.5.4 Automated Machine Learning

Here, we build a learning algorithm that learns which model to use to solve a given problem.

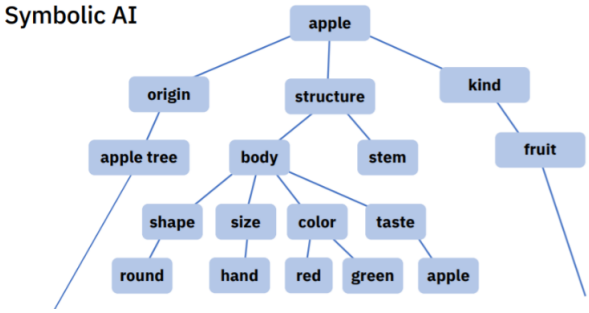


Here update *RNN* controller based on the accuracy of the child network after training.

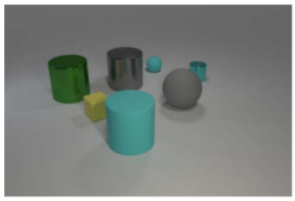
8.6 Symbolic AI

The neural networks we have been dealing with are able to identify an object given that we already trained it with similar objects, or generate new sample data from the training data. We also saw that these models are easily gullible to errors which are induced in the training data like adversarial attacks etc. These models also require a large amount of training samples to learn anything whereas we require only one training example to be able to fully identify it. We also can't ask any questions which involve reasoning hence we introduce **NeuroSymbolic AI**.

While viewing any item, a CNN uses features to identify it but a Neuro Symbolic AI goes deeper and analyses various things which distinguish it from other objects. This is how a NeuroSymbolic AI views an apple :



NeuroSymbolic AI is about disentangling reasoning from vision and language understanding. It involves an end-to-end neural network which analyses the question and answers it. This analyses the vision using a *CNN* and forms a structured representation of objects in the picture, and then analyses the question using *RNN* and answers it using a symbolic program. This method has a very high accuracy (99.8% generally). It also uses less sampling data to learn from (1% of usual amount generally). It also has a simple way to interpret things.



Question: Are there more yellow matte things that are right of the gray ball than cyan metallic objects?

```
scene
filter_cyan
filter_metal
count
...(4 modules)
scene
filter_yellow
filter_rubber
count
greater_than
```

Answer: no

We can make it answer metaconcept questions like :

- Is red the same kind of concept as green?
- Is cube a synonym for block?

We can also make it learn using videos. This system is called *CLEVRER* and is developed by the IBM. We can ask descriptive, explanatory and counterfactual questions.

8.7 Generalizable Autonomy in Robot Manipulation

Vision: Build Intelligent Robotic Companions towards Human Enrichment and Augmentation. We can to do this by learning with structured inductive bias and priors.
Example: Learning to wipe a surface.

This needs an *Environment* model, a *Robot* model etc. This uses mainly reinforcement learning which involves a reward function. But we are going to use a model-free *RL* agent meaning there is not going to be any environment model, the state is the image of the enivronment.

$$\tau = f(\pi(o_t))$$

$$\pi(o_t) = a : [x_d, \dot{x}_d, K_p, K_v]$$

$$\tau = f(x_d, \dot{x}_d, K_p, K_v)$$

where f determines the position-velocity control, $x_d, \dot{x}_d$ represents the pose and velocity, K_p, K_v represents the impedance gains.
This is going to be our torque of the robotic arm which takes into account all the things like *pose*, *stiffness* of the arm etc. The reward function here is going to be

$$\lambda_1 \sum(dirt\_on\_table) + \lambda_2(distance\_to\_table) - \lambda_3 \parallel (F \geq 40N)$$

We are going to train these robots by partially teaching it. It means that we are going to code a few techniques like *gripping*, *pushing*, *pulling* etc directly and let the machine learn with these skills. This greatly decreases the learning time under normal circumstances. We are going to use an *off-policy RL* which involves a state, a neural network which determines the policy function and gives actions, a replay buffer which stores all the actions and rewards of that action. Here we form sample random mini-batches which are used for calculating the policy gradient and the target values. This along with the partial teaching provides very good results.

Sequential Skills and Multi-step Reasoning :

If we focus on the robot arm doing various tasks, we also need our model to contain a task-oriented grasping model, a manipulation model which accounts for the policy.

For the arm, to do the tasks correctly, it needs to have sense of organising it's tasks step-by-step. For this, the *CAVIN* planner is used. The fundamental challenge of planning for multi-step manipulation is to find effective and plausible action sequences that lead to the task goal. We present **Cascaded Variational Inference (CAVIN)** Planner, a model-based method that hierarchically generates plans by sampling from latent spaces. To facilitate

planning over long time horizons, this method learns latent representations that decouple the prediction of high-level effects from the generation of low-level motions through cascaded variational inference. This enables us to model dynamics at two different levels of temporal resolutions for hierarchical planning. We evaluate this approach in three multi-step robotic manipulation tasks in cluttered tabletop environments given high-dimensional observations. Empirical results demonstrate that the proposed method outperforms state-of-the-art model-based methods by strategically interacting with multiple objects.
This has two parts:

- Effect code
- Motion code

The effect code gives various checkpoints in the process. The motion code gives us various action sequences for an effect code.

8.7.1 Neural Task Programming

NTP takes as input a task specification (e.g., video demonstration of a task) and recursively decomposes it into finer sub-task specifications. These specifications are fed to a hierarchical neural program, where bottom-level programs are callable subroutines that interact with the environment. We validate our method in three robot manipulation tasks. *NTP* achieves strong generalization across sequential tasks that exhibit hierarchical and compositional structures. The experimental results show that *NTP* learns to generalize well towards unseen tasks with increasing lengths, variable topologies, and changing objectives.

8.7.2 Neural Task Graphs

Our goal is to generate a policy to complete an unseen task given just a single video demonstration of the task in a given domain. We hypothesize that to successfully generalize unseen complex tasks from a single video demonstration, it is necessary to explicitly incorporate the compositional structure of the tasks into the model. To this end, we propose **Neural Task Graph (NTG) Networks**, which use conjugate task graphs as the intermediate representation to modularize both the video demonstration and the derived policy. *NTG* improves data efficiency with visual input as well as achieving strong generalization without the need for dense hierarchical supervision. We

further show that similar performance trends hold when applied to real-world data.
Also we can find data for robotics using **RoboTurk**, where we operate the robot beforehand and feed this data into it for learning purposes.

8.8 Neural Rendering

Efficient rendering of photo-realistic virtual worlds is a long standing effort of computer graphics. Modern graphics techniques have succeeded in synthesizing photo-realistic images from hand-crafted scene representations. However, the automatic generation of shape, materials, lighting, and other aspects of scenes remains a challenging problem that, if solved, would make photo-realistic computer graphics more widely accessible. Concurrently, progress in computer vision and machine learning have given rise to a new approach to image synthesis and editing, namely *deep generative models*. **Neural rendering** is a new and rapidly emerging field that combines generative machine learning techniques with physical knowledge from computer graphics, e.g., by the integration of differentiable rendering into network training.

Traditional computer graphics rendering pipeline is designed for procedurally generating *2D* quality images from *3D* shapes with high performance. The non-differentiability due to discrete operations such as visibility computation makes it hard to explicitly correlate rendering parameters and the resulting image, posing a significant challenge for inverse rendering tasks. Recent work on **differentiable rendering** achieves differentiability either by designing surrogate gradients for non-differentiable operations or via an approximate but differentiable renderer. These methods, however, are still limited when it comes to handling occlusion, and restricted to particular rendering effects. We present **RenderNet**, a differentiable rendering convolutional network with a novel projection unit that can render *2D* images from *3D* shapes. Spatial occlusion and shading calculation are automatically encoded in the network. Experiments show that RenderNet can successfully learn to implement different shaders, and can be used in inverse rendering tasks to estimate shape, pose, lighting and texture from a single image.

We propose a novel generative adversarial network *(GAN)* for the task of unsupervised learning of *3D* representations from natural images. Most generative models rely on *2D* kernels to generate images and make few assumptions about the *3D* world. These models therefore tend to create blurry images or artefacts in tasks that require a strong *3D* understanding, such as novel-view synthesis. **HoloGAN** instead learns a *3D* representation of the

world, and to render this representation in a realistic manner. Unlike other GANs, *HoloGAN* provides explicit control over the pose of generated objects through rigid-body transformations of the learnt *3D* features. Experiments show that using explicit *3D* features enables *HoloGAN* to disentangle *3D* pose and identity, which is further decomposed into shape and appearance, while still being able to generate images with similar or higher visual quality than other generative models. *HoloGAN* can be trained end-to-end from unlabelled *2D* images only. Particularly, we do not require pose labels, *3D* shapes, or multiple views of the same objects. This shows that *HoloGAN* is the first generative model that learns *3D* representations from natural images in an entirely unsupervised manner.

8.9 Machine Learning for Scent

This is based on an early research model.
Here, we will try to *digitise* the sense of smell. Here, we also build a benchmark for the labels which is to be reached. We try to generalise the smell with existing rules but there are many exceptions to them.
We try to digitise the molecules not based on images and pixels but based on graphs. Hence, the final neural networks are also called **graph neural networks (GNNs)**. In a graph, there are *layers* of information such that each layer has all the information about the *previous* layers. Here, we begin with an atom, examine its surroundings and add that information back into the atom in each layer. Hence, finally every atom has information about every other atom in the molecule. After this, we create *embeddings* to use as inputs. In the context of neural networks, *embeddings* are low-dimensional, learned continuous vector representations of discrete variables. Neural network embeddings are useful because they can reduce the dimensionality of categorical variables and meaningfully represent categories in the transformed space. This is just the early stage of this research which has not been done conclusively.

9 Conclusion

This is everything that has been going on in this decade. We are still making progress towards AI. There are many questions we haven't found an answer to yet. But, we will in the future.

10 References

- Andrew Ng's Machine Learning course in Coursera.
- MIT 6.S191