3.2.1 Introduction to Deep Learning

In this course, we're going to study a revolutionary and exciting new approach called deep learning. In contrast to the approaches such as Linear Regression, Logistic Regression, and some other algorithms in Machine Learning which are well understood, no one really knows when or why deep learning works so well. The **interpretability** of deep learning models is **very low compared to Machine learning**.

In the last decade, the application of deep neural networks to long-standing problems has brought about a breakthrough in performance and prediction power. However, high accuracy, deriving from the increased model complexity, has often come at the price of loss of interpretability, i.e., many of these models behave as black-boxes and fail to provide explanations on their predictions. While in certain application fields this issue may play a secondary role, in high-risk domains, e.g., health care, it is crucial to build trust in a model and be able to understand its behavior.

So why is everyone so excited about Deep Learning?
Let us look at some examples of applications where deep neural networks have revolutionized things:

1. Speech recognition
2. Google's AI algorithm masters the ancient game of Go
3. Computer vision systems are now outperforming humans at various tasks
4. Self-driving cars

Speech recognition has given computers the ability to understand natural language.
It's not that speech recognition didn't have good solutions before deep learning came around, it's just that the solutions that came from deep learning are markedly more accurate and are fast enough that they can even be done in real-time so that you don't have to wait seconds to have your voice transcribed. One such example is the Amazon voice assistant **Alexa**.

Another major achievement of deep learning was mastering the game of Go. Go was long thought to be a much harder game for computers to play competitively than other games like chess, checkers or backgammon. In each of those games, we've had terrific

computer programs for many years that could beat or at least compete with the world's best players, but until 2016, the best computer programs for Go were nowhere near the top Go players in the world. But now the crown belongs to the machines.

Read more about AlphaGo - the first Go program to defeat professional Go players.

Some of the most exciting applications of deep learning that have grabbed everyone's attention have been in computer vision. Computer vision is a field of study which enables computers to replicate the human visual system. It collects information from digital images or videos and processes them to define attributes. Some applications:

1. **Cancer Detection:** Image recognition allows scientists to detect slight differences between cancerous and non-cancerous images and diagnose data from magnetic resonance imaging (MRI) scans and inputted photos as malignant or benign
2. **Mask Detection:** Detect the use of masks and protective equipment to limit the spread of coronavirus

Most current computer vision applications such as cancer detection, self-driving cars, and facial recognition make use of deep learning.

For self-driving cars, computer vision enables them to make sense of their surroundings. Cameras can capture video from different angles around the car and feed it to the computer vision software, which then processes the images in real-time to find the extremities of roads, read traffic signs, detect other cars, objects, and pedestrians. The self-driving car can then steer its way on streets and highways, avoid hitting obstacles, and (hopefully) safely drive its passengers to their destination.

There are a lot of techniques in deep learning-based computer vision and it's an active research area, so a lot of new algorithms are rapidly coming up. But how good are these approaches when compared to each other? Within the computer vision community, there's a famous data set called **ImageNet**, which is associated with a yearly competition for the best image recognition performance on that dataset as well.

ImageNet is a large dataset of annotated photographs intended for computer vision research. The goal of developing the dataset was to provide a resource to promote the research and development of improved methods for computer vision. ImageNet consists of 1 million labeled images for training, 50,000 validation images, and 100,000 test images. Each image is labeled with one of 1000 different possible categories. Check out the 1000 classes of images here.

Now this is a massive collection of images. The goal is to train a classifier that can correctly label images with the smallest possible error rate. There's one technicality, in that the way we measure error rates is to let the classifier choose five different labels which it thinks are the most likely labels for the image, and we count how often the true label of the image is one of these five predicted labels. Every time it is, we call it a success, and every time it isn't, we call it an error.

What makes image classification so challenging is that it's something that humans are quite good at, but machines are relatively bad at.

There are some tasks that machines are good at, like calculating products of eight-digit numbers, and other such computational tasks. Surely your computer or smartphone is much better at that than we are.

But there are still some things that humans are very good at which machines simply haven't caught up to. For those of you who have kids, remember back the time when your kid was first learning to recognize animals. At first, he/she may not know the difference between an elephant and a dog. Maybe they'd seen many dogs in person but never an elephant. So they might point to a picture of an elephant in a storybook and say "dog", but it takes very few examples to teach your child the difference between elephants and dogs. You could show them a few examples of each and point out the long trunk, and you'd be able to extrapolate all sorts of new pictures that they've never seen before. This is because the **human mind is amazing** at understanding patterns from a very small number of examples**.** This phenomenon is often called **one-shot learning** because it takes just a few examples to learn an entirely new category. Nowadays we are also using one-shot learning in deep learning for image classification.

Check out a research paper on one-shot learning to understand it here.

We're quite far away from having machines that can learn nearly as fast. There have been decades of work on computer vision, developing a lot of techniques, but until recently, the state-of-the-art algorithms could only achieve error rates between 25 and 30% on ImageNet. This is still quite good, but it's quite far away from what humans can achieve, which is somewhere in the range of 5-6%. Some of the categories are so specific that even humans get them wrong. All of this changed in 2012 when a team of researchers used deep learning to achieve error rates of about 15%. In fact, even these bounds were quickly improved upon. Every year since 2012, the competition has been won by deep learning, and the entire field of computer vision has shifted its center of mass towards these new techniques. The current best algorithms achieve error rates of less than 5% and amazingly perform about as well, if not a little bit better than humans themselves.

This is something that many researchers thought would take decades to accomplish, but this problem and many others like it are areas where deep learning is rapidly reshaping the boundaries of what we think of as possible. It's enabling machine learning in a way that's both awesome and exciting, but it's also a technology, unlike most of what we've covered earlier, that we don't fully understand the algorithms behind.

Additional Content:
Why is deep learning taking off?

Deep learning has been around for decades. So why is it only now taking off?

1. **Data:** There is a large amount of data available through digitalization, and data is generated from a lot of devices. Deep neural networks always work better with more data. So you can get better performance as long as you collect more and more data, without changing the algorithm itself.

2. **Computation:** The computational power available has grown exponentially in recent years, thanks to cloud computing and GPU development. We can now train neural networks faster, and that has been a big contributor to developing high-performing deep learning models.

3. **Algorithms:** The process of improving a neural network's architecture is also iterative. So the faster computation described in the previous point also helps to iterate, improve and create new algorithms faster. This improvement in algorithms has also led to incremental and sometimes significant gains in the performance of deep learning algorithms.

These factors have helped make deep neural networks the dominant algorithm in machine learning these days.

3.2.2 Classification Using a Single Perceptron

Let's understand the term "deep learning". The word deep refers to the layers. So deep learning is about building complex hierarchical representations from simple building blocks.

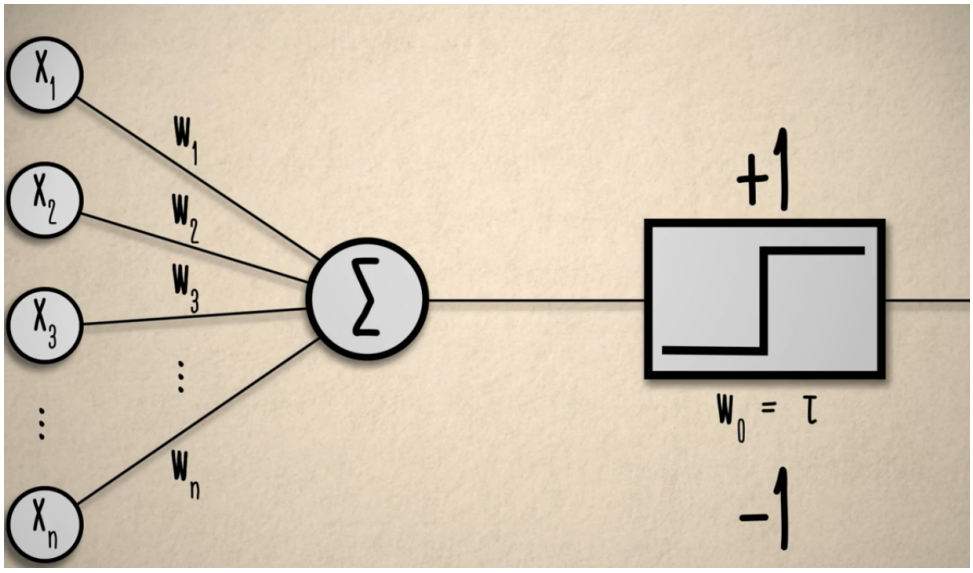
"*The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI deep learning.*" - Ian Goodfellow

Deep learning models have the ability to perform automatic feature extraction from raw data, also called feature learning.

We now understand that deep learning has been built using simple building blocks. Let's dive deep in and understand these building blocks.

The perceptron is the basic building block of deep learning. Sometimes we also call it a neuron, so the neural network is nothing but a network or layer of neurons or perceptrons.

Let's recall what the perceptron is,
It's a function that has several inputs and one output. Let's say that it has n inputs {X1,....., Xn}. Then the output of a perceptron is computed in two steps:



Step 1: we compute a linear function of the inputs; the coefficients of this linear function are called weights. We define these weights with random values.

$$W_1X_1 + W_2X_2 + W_3X_3 + \ldots + W_nX_n$$

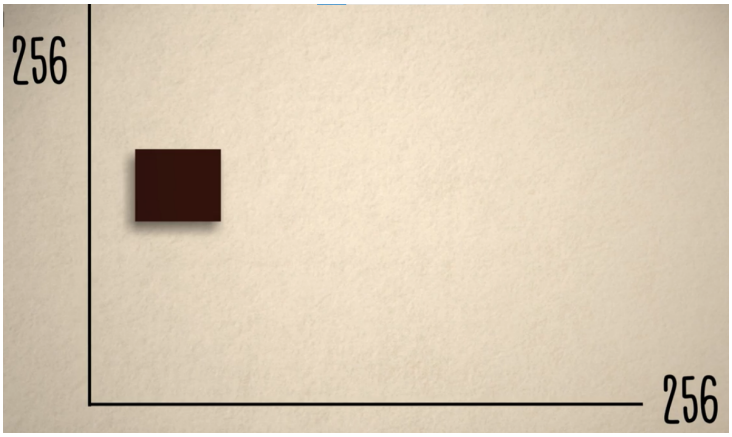
Step 2: we take the output of the first step and compute a threshold. This threshold takes any value above some cutoff tau and maps it to the value +1, and maps everything below tau to -1. The second step is the only non-linearity in the network.

So the above perceptron required n weights corresponding to n inputs and one value of tau. So in total, we need n+1 parameters for each perceptron.

What's the connection to support vector machines?
The perceptron only recognizes linear patterns, and to combat this, we introduce the idea of mapping the input space to a new space using a kernel in deep learning, we will take a very different approach where we layer perceptrons on top of each other to get our non-linear.

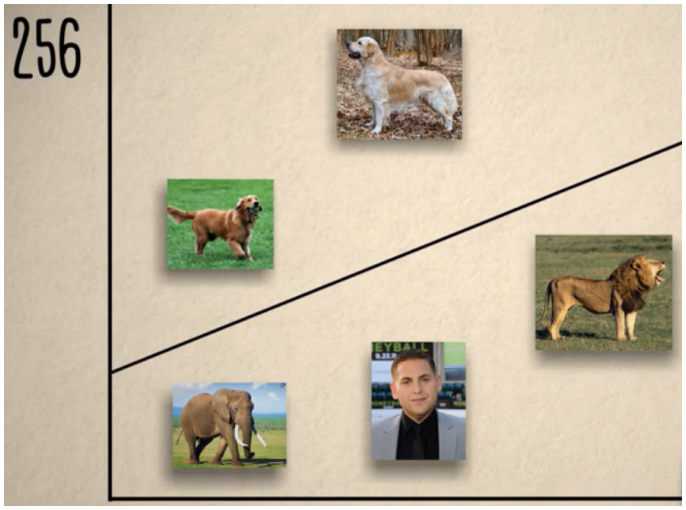
So we'll start with a concrete example. Let's return to the problem of image classification. Here we need to classify which image contains the dog and which doesn't?

These are colored images so each pixel is associated with three values Red, Green, and Blue which we call RGB. we can represent our input, say, 256 by 256 size then the image dimension will be (256,256,3) where 256, 256 are height and width of the image and 3 is the RGB values for each pixel.



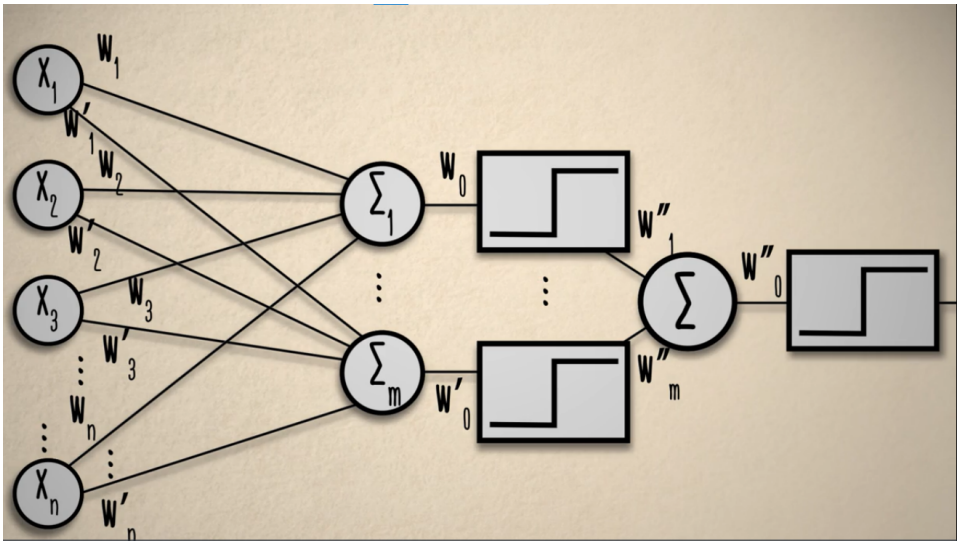
So it takes 256 X 256 times three-dimensional space.

Now we want the perceptron to solve our image classification problem. What we're really asking for is a linear function of pixels that returns positive if the image contains a dog and negative if it doesn't.



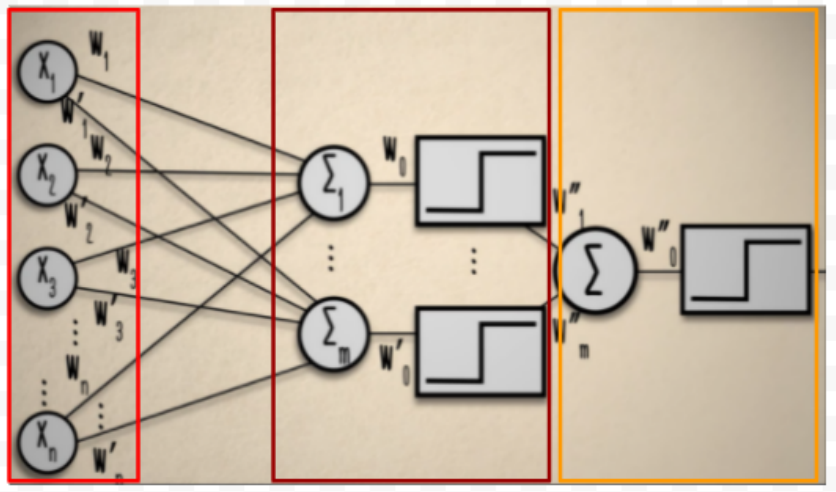
We might imagine a perceptron is just too simple a function to solve such a complex task efficiently. There probably isn't a nice linear function of the pixels that decides whether or not there is a dog in the picture. **So the key takeaway is that a simple linear function cannot identify complex patterns.** So we need more complex functions or non-linearity in the functions. Here non-linearity means it tries to fit a curve instead of a line as the boundary.

So how can we create more complex functions out of perceptrons as building blocks? As we have discussed we need to add more layers and the simplest setup imagines that there are just two layers of perceptrons.



Our input is an n-dimensional vector representing a picture. Each perceptron, in the first layer, has n inputs, each with its own weight. So each perceptron, in the first layer, computes its own threshold and returns the output. This output will be an input to the next layer perceptron and these inputs for the second layer perceptron have their own set of weights and threshold and return the output.

The above diagram can be divided into three layers,



The red one corresponds to the input layer and the brown one in the middle corresponds to the hidden layer which is responsible for calculating complex patterns. The orange one is the output layer.

So,
Total number of layers = Number of hidden layers + output layer

We don't count the input layer in the total layers.
As we know each perceptron has n+1 parameters. So in the above network, the first layer has m perceptrons, which take a total of **(n+1) x m parameters**. Similarly, for the next layer, we have one perceptron that takes n+1 parameters. So the total parameters this network takes is,

$$(n+1)m + (m+1) \text{ parameters}$$

Let's understand all these steps with a simple example:
Here we consider 2 perceptrons in the first layer and 1 perceptron in the second layer.

Suppose we have 6 input values,

x_1	10
x_2	24
x_3	15
x_4	18
x_5	5
x_6	20

We have two layers. Let's start with layer 1 with two perceptrons. Since we have 6 inputs, each perceptron takes 6 weights (takes random values), and let's say the threshold is 10. So:

Perceptron1_layer_1 = $x_1w_1 + x_2w_2 + x_3w_3 + x_4w_4 + x_5w_5 + x_6w_6$

= (10 x 0.2)+ (24 x 0.1) + (15 x 0.15) + (18 x 0.25) + (5 x 0.4) + (20 x 0.12)

= 15.55

Now if we compare with the threshold value which is 10, it's higher so it returns +1.

Now we need to calculate this for perceptron 2. It takes six weights which are different from the first layer and let's say the threshold is 15.

Perceptron2_layer_1 = $x_1w`_1 + x_2w`_2 + x_3w`_3 + x_4w`_4 + x_5w`_5 + x_6w`_6$

= (10 x 0.14)+ (24 x 0.23) + (15 x 0.2) + (18 x 0.3) + (5 x 0.28) + (20 x 0.1)

= 18.72

Now if we compare with the threshold value which is 15, it's higher so it returns +1.

So the outputs of the first layer are { +1, +1 }. These will be inputs for the second model. For the second layer which has one perceptron, we do a similar calculation. It has two inputs, so we need two weights here along with a threshold: let's say 0.5.

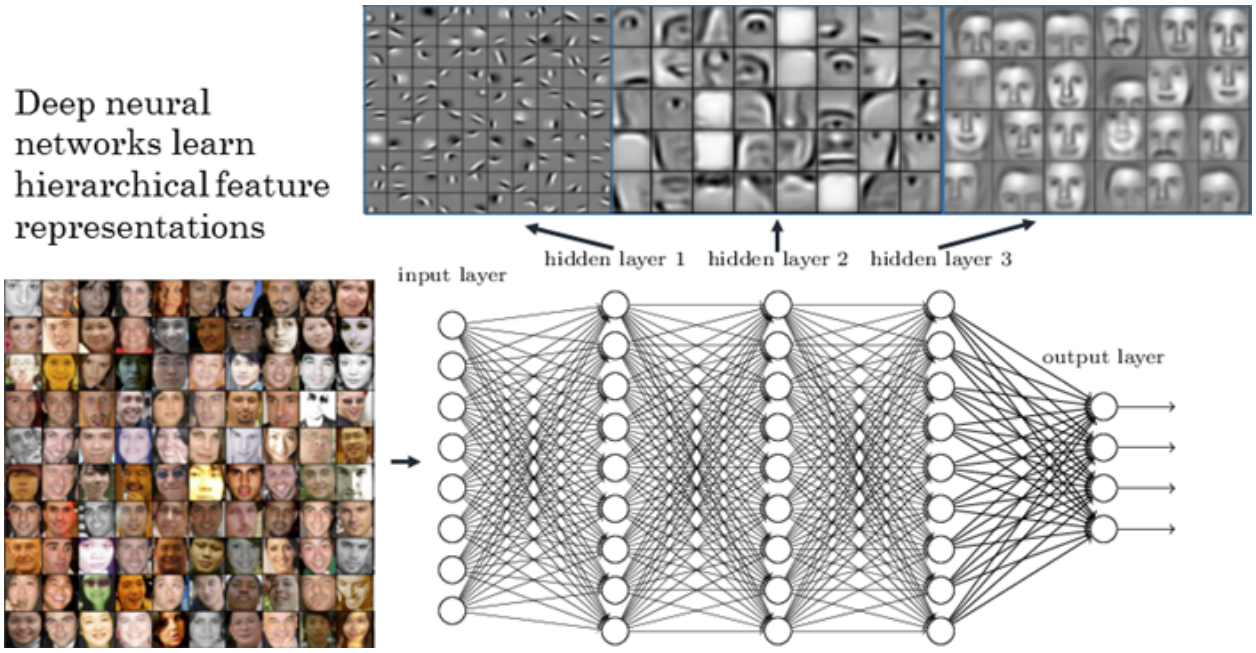
Perceptron1_layer_2 = $x_1 w_{21} + x_2 w_{22}$ = (1 x 0.5) + (1 x 0.2) = 0.7

Now if we compare with the threshold value which is 0.5, it returns +1.

Total number of parameters are = (n+1)m + (m+1) = (6+1)2 + (2+1) = 17.

This is how the values are calculated and can be extended to the multiple layers with multiple perceptrons.

We can take this idea much further and have more than two layers, the functions that we get are called **deep neural networks** and in practice, one usually sets them to have between 6 and 8 layers and millions of perceptrons in between. There are several fragments of the intuition behind these types of functions. The hope is that the lower-level layers of the network identify some base features like edges and that each layer on top of them, builds on the previous layer to create much more complex features.

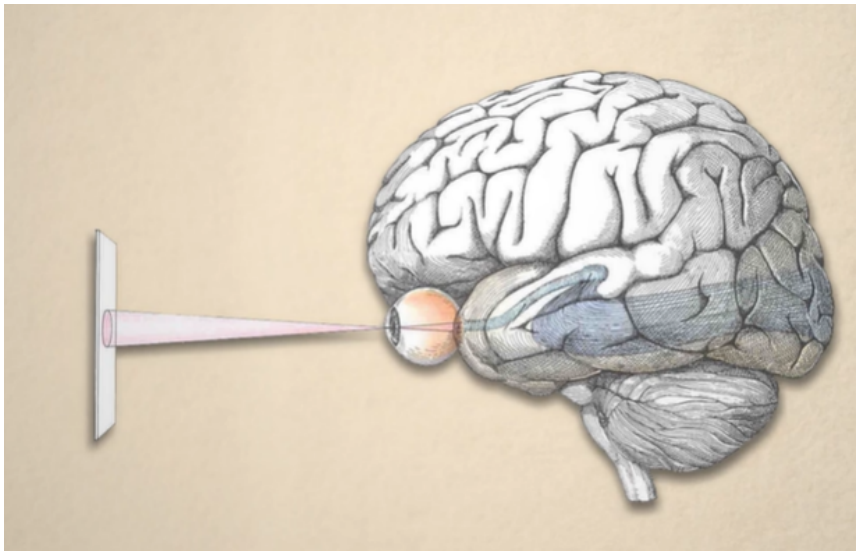


Source -Andrew NG

The intuition is to find whether or not a dog is present in an image. First, you would need to identify its edges, and then you'd identify which collections of arrangements of edges represent lights, and which represent the body of the head, and then which arrangements of these parts represent the dog.

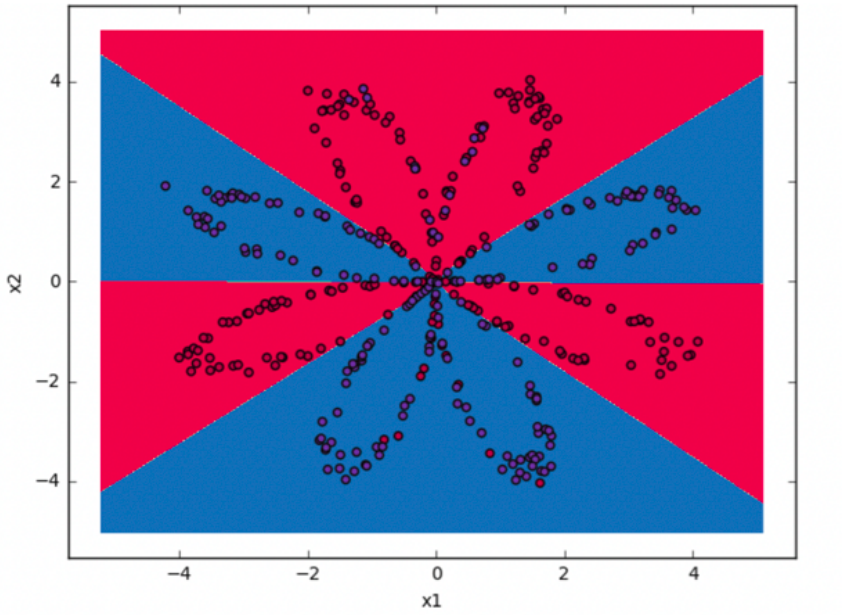
Recall that in ImageNet we want to correctly classify according to 1000 different labels at once. Even though there are a million total images, that's not actually that many examples for labels. So what's important is that the features that are useful for identifying one breed of dog can be useful in identifying other breeds of dog as well. In this sense, a deep network can have 1000 outputs, one for each label, built on top of a common deep network underneath it, one which is hopefully identifying useful high-level representations that are needed to understand images.

Another rationalization for deep neural networks is that they parallel what happens in the visual cortex. There's still a lot about the brain, okay, most of the brain that we don't understand. But it does seem that the visual cortex has a similar type of hierarchical structure with neurons in the lower layers recognizing lower level features like edges.



Moreover, we can measure how quickly a human can recognize an object, and how quickly a neuron fires. These tell us that even though the visual cortex is performing some hierarchical computation. It requires at most six to eight layers in order to solve high-level recognition problems.

So let's conclude, First, the perceptron has a linear and nonlinear, threshold function. If it wasn't for the threshold, creating deeper networks would not buy us anything. We would still be composing linear functions and no matter how deep we make the computation, the function we get would still be linear. It's the nonlinearity that we added that makes deep networks so functionally expressive. We have understood using the threshold function we are introducing some kind of non-linearity into the network but **why do we need non-linearity?** It is hard to find any physical world phenomenon which follows linearity straightforwardly. We need a non-linear function that can approximate the non-linear phenomenon. For example as shown in the image,



To introduce non-linearity we use some kind of function for example the threshold function. The functions are called **activation functions**. The purpose of the activation function is to **introduce non-linearity** into the output of a neuron.

In fact, there are many other nonlinear functions or activation functions that we could have chosen instead of a threshold, we could have chosen a logistic sigmoid or hyperbolic tangent or any other smooth approximation to a step function. This and many other aspects of the architecture of deep neural networks are all valid design choices that have their own merits. There are many research papers that grapple with issues like, which nonlinear functions work the best, and how we should structure the

internal layers, for example using convolution. We will talk more about these issues but it's good to know that they're very important.

Second, we've said nothing about actually finding the parameters of the deep network. So while calculating the perceptron we have used the weights which we have defined randomly. These weights need to be learned by the network. Modern deep networks have millions of parameters, which is a very large space to search for. When we talked about the support vector machine we had the perceptron algorithm that told us if there is a linear classifier, we can find it algorithmically. But even if there is a setting of the parameters of a deep network that really can classify images accurately into say different breeds of dog, how can we find them? There is no simple answer to this question. There are approaches that seem to work in practice, but why they do is still very much a mystery, and perhaps a phenomenon that has to do with some of the strange idioms of searching in such a high dimensional space.

Additional content :

Types of Activation functions:

1) **Sigmoid**

   The main reason why we use the sigmoid function is that it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output since the probability of anything exists only between the range of 0 and 1, sigmoid is the right choice. The logistic sigmoid function can cause a neural network to get stuck at the training time. Therefore, it is mostly used in the output layer. (in the case of binary classification).

2) **Tanh**

   Tanh is a shifted version from the sigmoid function where its range is between -1 and 1. The mean of the activations that come out of the hidden layer is closer to having a zero mean therefore data is more centered which makes the learning for the next layer easier and faster. One of the downsides of both sigmoid and tanh is if our weighted sum input is either very large or very small, then the gradient (also called derivative or slope) of this function becomes very small and ends up being very close to zero. This can slow down learning.

3) **Relu**

Relu is increasingly the default choice of activation functions. If you are not sure what to use in the hidden layers, just use the Relu activation function or one of its variants. It is a bit faster to compute than other activation functions, and gradient descent does not get stuck as much on plateaus thanks to the fact that it does not saturate for the large input values as opposed to the logistic function or the hyperbolic tangent function which saturate at 1. One disadvantage of Relu is that the derivative is equal to zero when (z: weighted some input) is negative. The problem is known as the dying Relu. If the weights in the network always lead to negative inputs into a Relu neuron, that neuron won't be effectively contributing to the network training. There is another version of the Relu activation function called the leaky Relu that solves the dying Relu problem. It usually works better than the Relu activation function.

4) **LeakyReLU**

   The LeakyRelu activation function usually works better than relu. But it is not that much used in practice.

5) **Softmax**

   The softmax activation function is used in neural networks when we want to build a multi-class classifier which solves the problem of assigning an instance to one class when the number of possible classes is larger than two(otherwise we can simply use sigmoid if possible classes=2).

In the output layer, the activation functions are selected based on the problem statement, for example, if it's

- Regression: linear (because values are unbounded)
- Classification:
  - Binary classification - Sigmoid
  - Multiclass classification - softmax

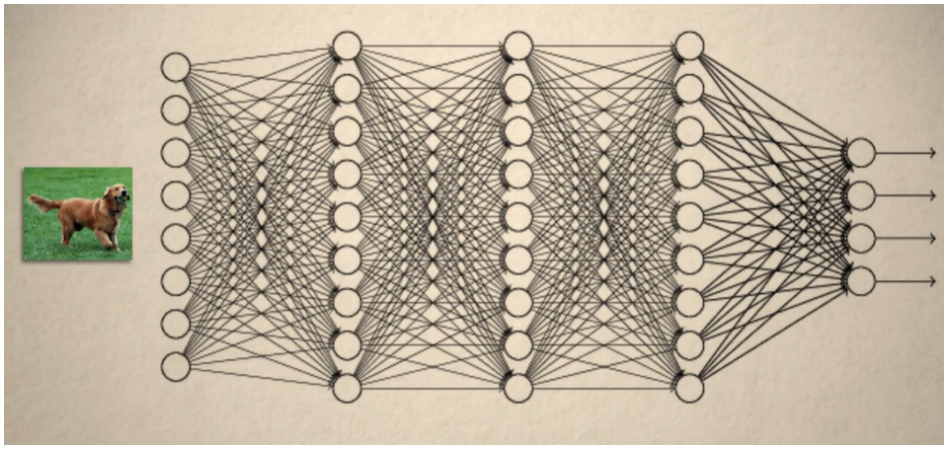
Above we have used the weights randomly but we should be very careful while defining the weights. To get a better idea, read from here Weight Initialization.

3.2.3 Setting parameters of a Deep Neural Network - Hierarchical Representations

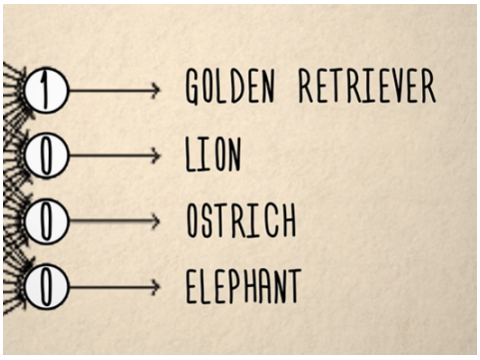
In previous videos we get to know that we need to set the parameters or weights but how do we set the parameters of a deep neural network? So our first goal is to cast the problem of **finding good parameters** as an optimization problem.

The catch is that not all optimization problems are created equal. There's an important class of optimization problems called **convex optimization problems** for which we have many good algorithms and we understand quite well, when and why they work. But life is not so easy in the case of deep neural networks, the optimization problems that come out will be **nonconvex and unwieldy**.

Let’s first define the optimization problem we'll be interested in, returning to the image net example, our deep neural networks have n inputs and 1000 outputs, The image will be assigned to one of the 1000 categories.

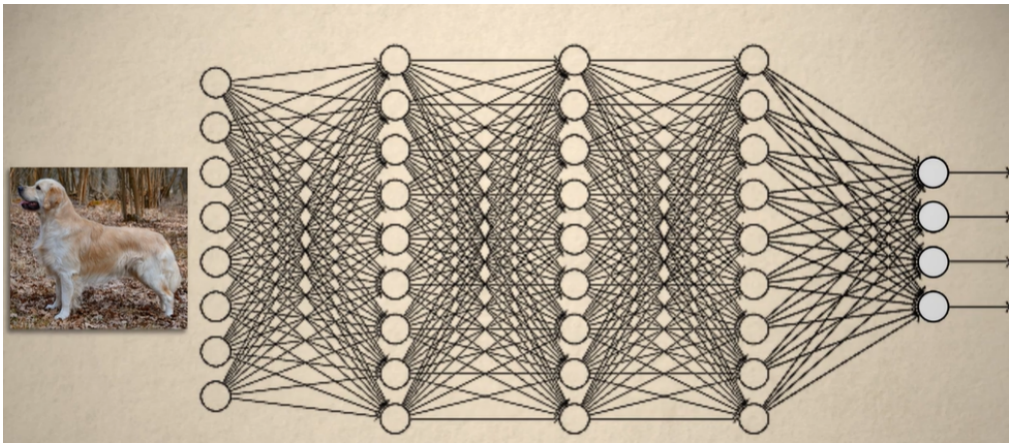


What we're hoping for is that when we feed in a picture as an n-dimensional vector, for example the image of a dog, then the output in the last layer contains a one in the correct category of zeros and all the other categories.



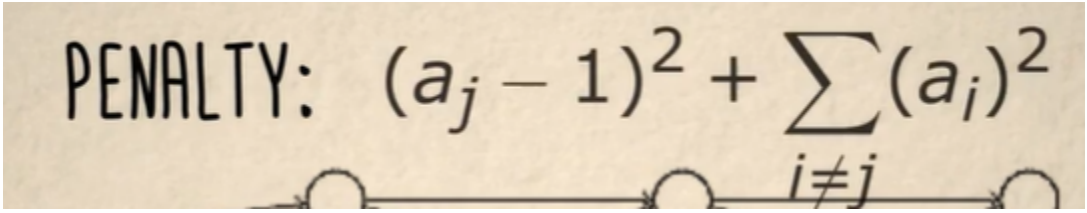
Now let's say we have all parameters of a deep network, then how could we evaluate how good these parameters are? We could take each one of the 1 million examples of the ImageNet then feed each picture in the network and check whether you got the correct label.

To be concrete let's discuss what's called the quadratic cost function, let's say the input is some vector x, and it's true label is category j, then the output that we would like to get is all zeros, except for a 1 in the J of outputs.



For example, in the above image, we pass the input as the dog(it's some vector x). So the true label of the image is the dog. Then as an output from the network, we need to get Zero for all other categories and 1 for the dog category.

Now we could penalize by how far off the output is from this idealized output. Let's say we if on input x we output $a_1, a_2, a_3 \ldots a_m$ where a refers to one of the categories. Then the penalty is,



This function evaluates to zero if you get exactly the correct output and evaluates to something nonzero if you get the wrong category. Here a_j is the correct category the image belongs to.

Let's understand the function with an example,
Suppose we have passed the image x and the true label for the image is the cat. Now the network has predicted cats. In this case, a_j will be 1 because it has been predicted correctly and the rest of the a_i's are zero.

So the penalty = $(1 - 1)^2 + (0 + 0 + ...) = 0$. Since it's predicted perfectly so there is no penalty.

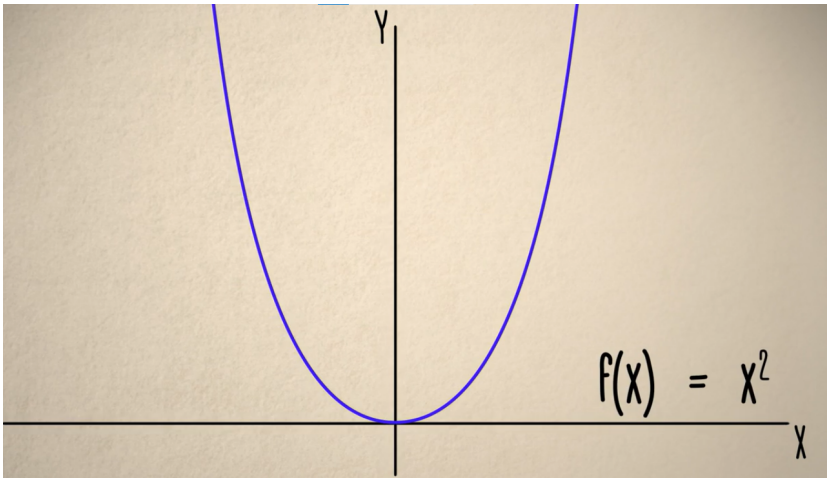
What we've done is we defined a cost function. The cost function is the technique of evaluating "the performance of our algorithm/model". Now if we have the parameters of a deep neural network we can evaluate how good it is by computing the average cost. We need the average cost to be minimum.

Now if we want to find a good setting of the parameters, we will look for the setting of the parameters that minimize this average cost. This is an optimization problem, we have an explicit function that depends on the parameters as well as the training data, that we'd like to minimize it turns out that if you find a set of the parameters that work well on the training examples, it has low average costs and achieves low error on a new set of examples. The key idea is to find the optimal parameters that have a low average cost and can predict the new set of examples with a minimal error rate.

We have an optimization problem that if we could solve would find good parameters. Is there just some off-the-shelf way that we can plug in any optimization problem we'd like, and get the best answer? Absolutely not. The difference between what optimization problems are easy to solve and which are hard is one of the foundational issues in theoretical computer science.

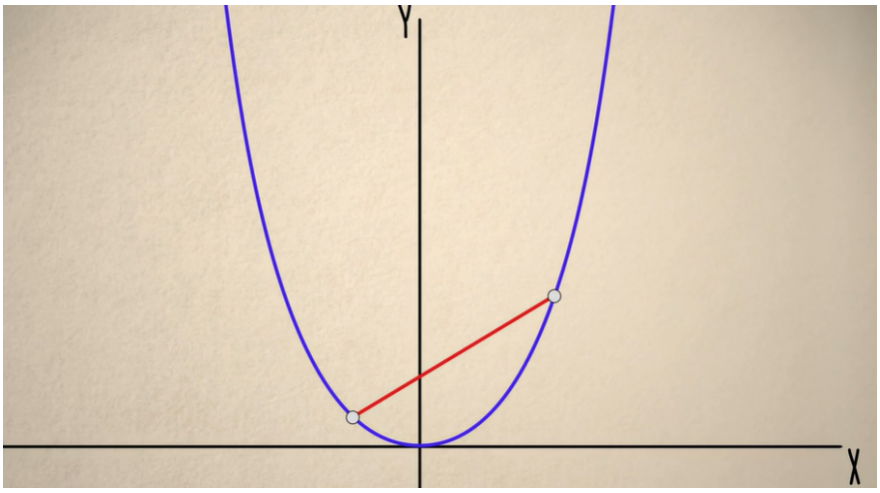
What types of optimization problems are easy?
Let's start with one-dimensional data to keep things as simple as possible. Let's say you have some function $f(x)$ and we want to find the x that minimizes the equation $f(x)$ where x is a real variable, so it can take on any real value.

There's an important class of functions called **convex function**. There are many ways to define convex function but let's start with an example, $f(x) = x^2$.



The above graph is convex it's convex because,

1. whenever you take any two points on the curve and draw a line between them. The line lies entirely above the curve, except that the endpoints.

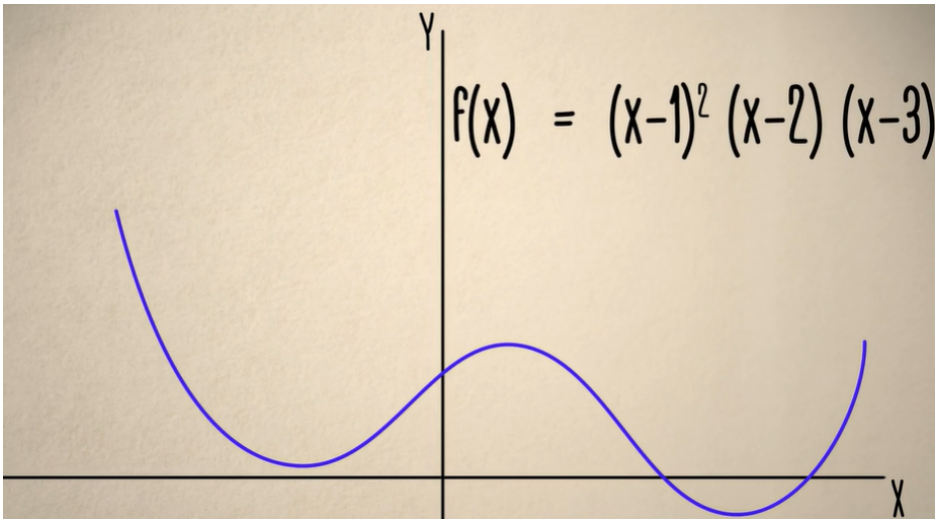


2. The more mathematical definition is “a function is convex if and only if its second derivative is always non-negative”. In the above example, the equation is $f(x) = x^2$ To calculate the second derivative we need to calculate the first-order derivative and then differentiate the first-order derivative to get the second-order derivative. According to the differentiation, the derivative of x^n is nx^{n-1}. $f'(x) = 2x$ which is the first-order derivative, For second-order we need to calculate derivative for the first order, So the second-order derivative is, $f'(f'(x)) = f'(2x) = 2(f'(x)) = 2 \times 1 = 2$. Which is a non-zero hence it’s a convex function.

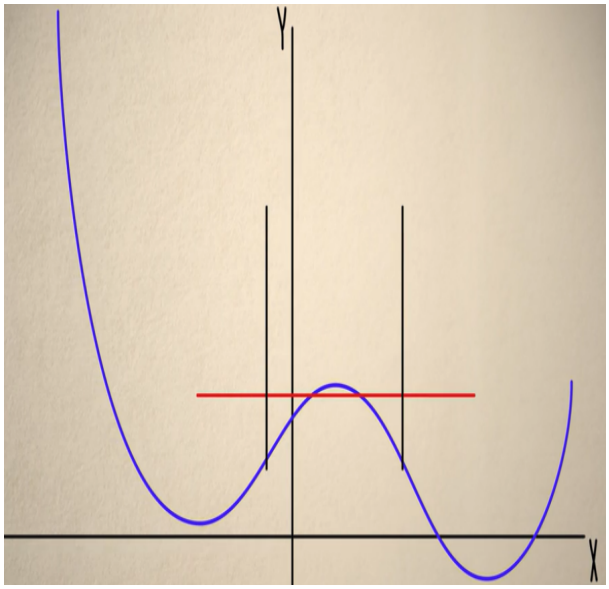
In the convex function, the global minima is equal to the local minima. The point at which a function takes the minimum value is called global minima. However, when the

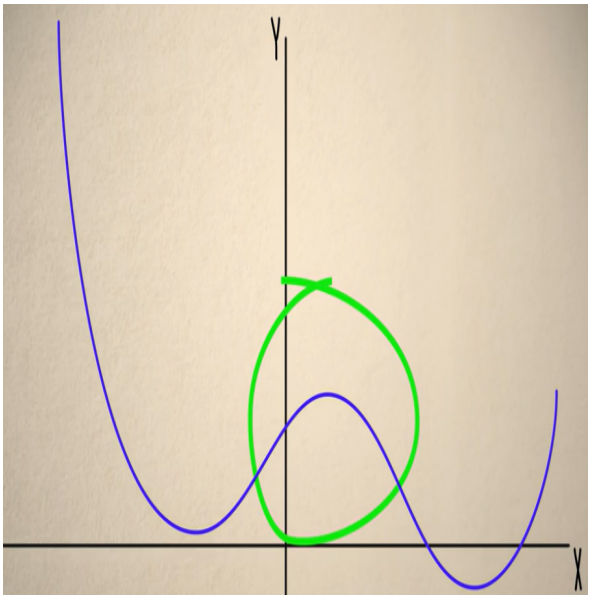
goal is to minimize the function and solve using optimization problems, it may so happen that the function may appear to have a minimum value at different points. Those several points which appear to be minima but are not the point where the function actually takes the minimum value are called **local minima**.

So what's an example of a non-convex function, let's say,



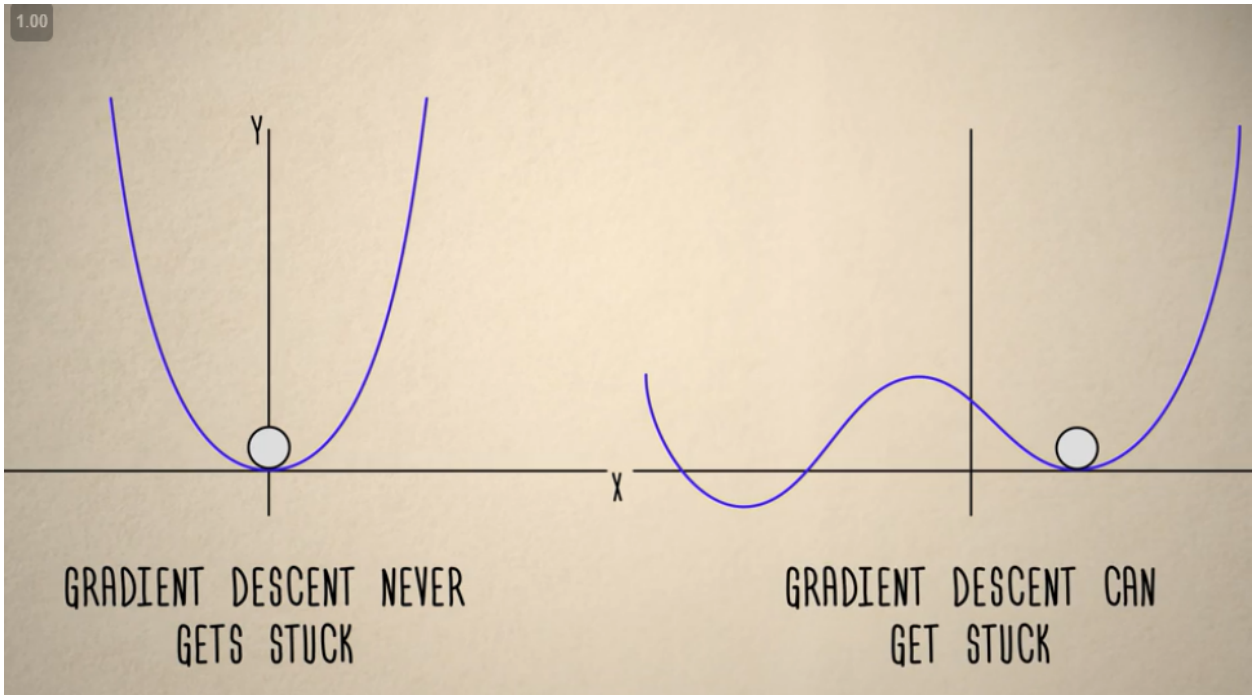
There are points on the curve, where the derivative is zero, and it's positive before and negative after.





This definitely means that its second derivative is negative around the green region. The real question is why is this bad. The important point is that convex functions have no local minima that are not also global minima. For convex, the local and global minima are at the same point. So there is nowhere that you can get stuck if you're

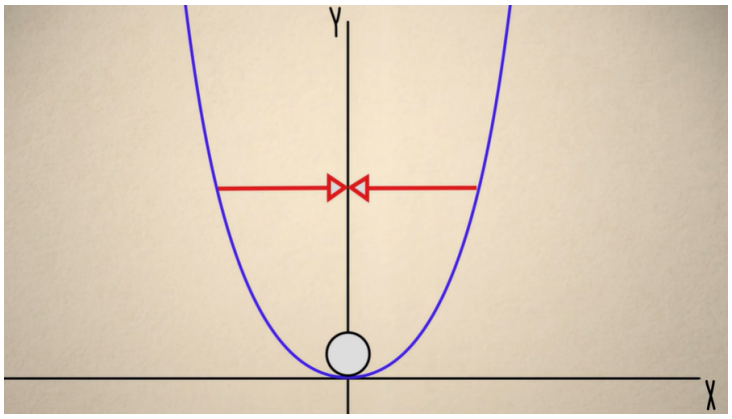
greedily following the path of steepest descent that isn't actually the globally optimal solution.



but as a non-convex case, you absolutely can get stuck. In our second example you could get stuck at x= 1, which achieves f(x) = 0, even though there are X's which make the function negative.

Let's understand what's going on here.

If you have a convex function, and you're on some value x, and you're searching for the value that minimizes the function f, what you would do is take the derivative at x, if it's negative you would take a step to the right increase x and if it's positive, you would take a step to the left.

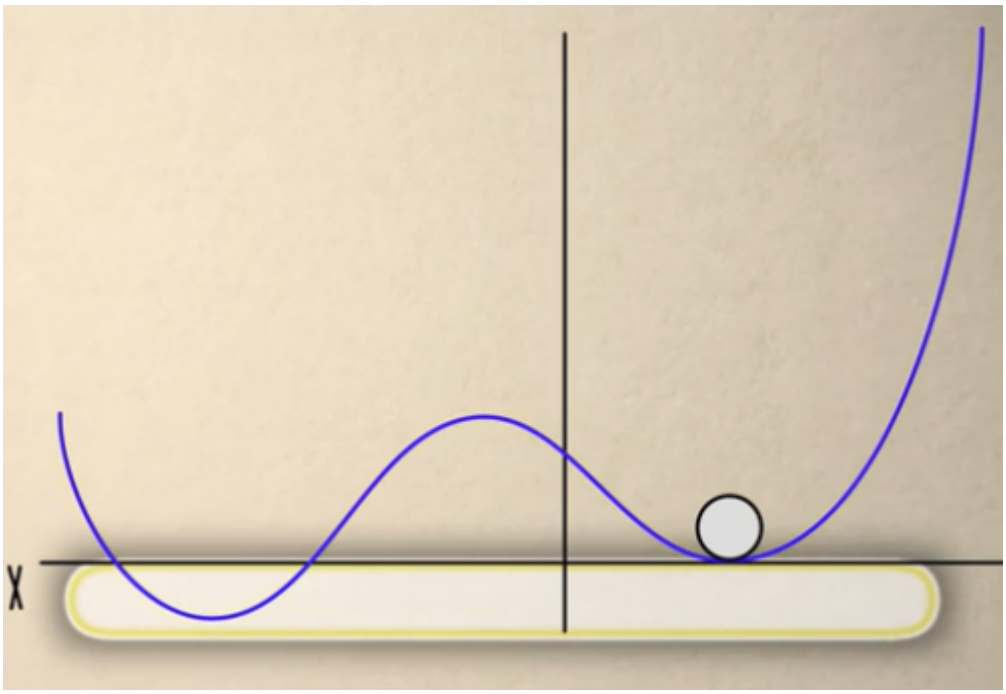


So in convex function to reach global minima the steps carried out are,

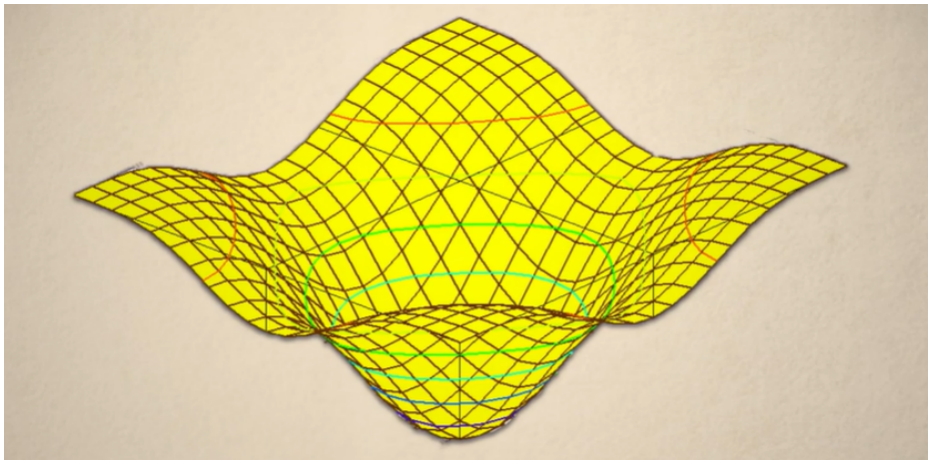
1. Calculate derivative at x.
2. If the derivative is negative you would take a step right or increase the value of x.
3. If the derivative is positive you would take a step left or decrease the value of x.
4. Repeat the process until the derivative is zero.

If you choose the step size correctly this is guaranteed to converge to the global minimizer of the function. As you may notice, you need to decrease the magnitude of the step based on how large the derivatives are.

Now, what happens if you try the same strategy on a non-convex function, all you can say is that you reach a local minimum. But in general, it will not be a global minimum.

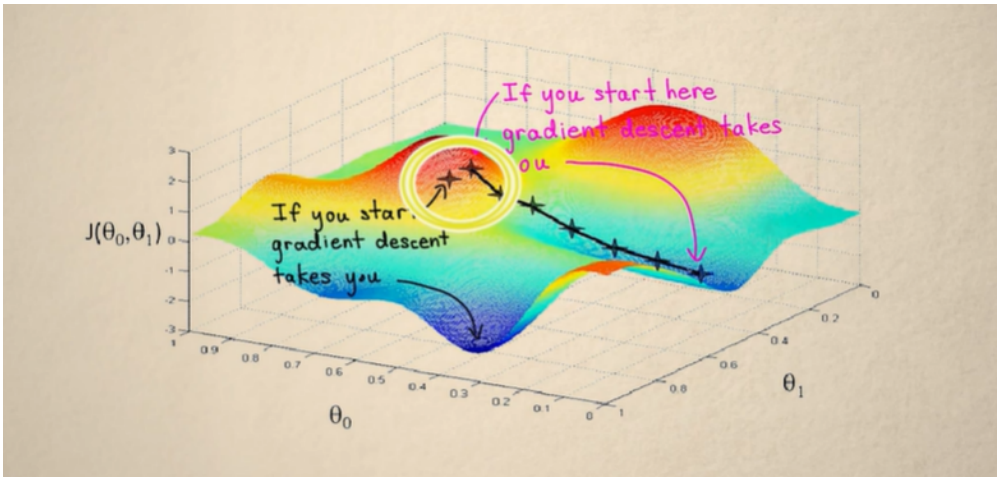


All of these ideas can be extended in a straightforward manner, the higher dimensional space.



Instead of taking the derivative, you would take the gradient. The gradient points in the direction of the largest increase for the local linear approximation and wherever you currently are, you would take a step in the direction opposite to the gradient. When you have a convex function and a high dimension. This is again, guaranteed to converge to the globally optimal solution.

When you have a non-convex function, it might only converge to a locally optimal solution, or even worse it could get stuck in a saddle point, which is not a local optimum but for which the gradient is zero.



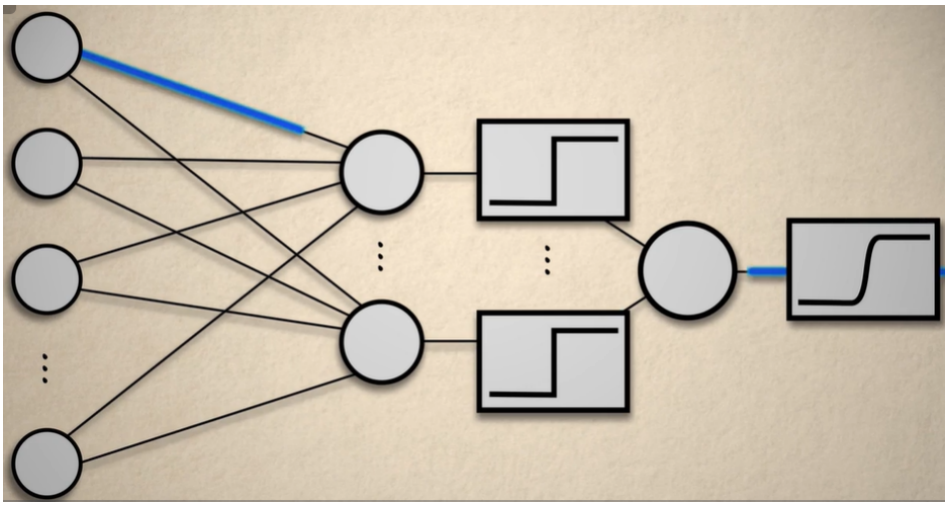
So what we're going to do is use gradient descent to find the best parameters for our deep neural network, even though the function we're trying to minimize is non-convex, we're taking an algorithm that's guaranteed to work in the convex case that we know does not always work in the non-convex case and using it anyways. One of the greatest mysteries is that it still seems to work, what it finds is not necessarily the

globally optimal solution, but even the locally optimal solution, It finds, is seemingly still good enough.

3.2.4 Computing Gradient using Backpropagation

Now let's understand how to apply gradient descent to a deep neural network. We have a cost function that we'd like to minimize using gradient descent.

So far when we talked about perceptrons we described them as linear plus nonlinear. We took the threshold function to be our non-linearity, but there are many other reasonable choices like a sigmoid function. The calculation is similar in the case of sigmoid, the perceptron computes a linear function of their inputs using their weights add in a constant term called the bias and feed this value into a sigmoid function to get the output.

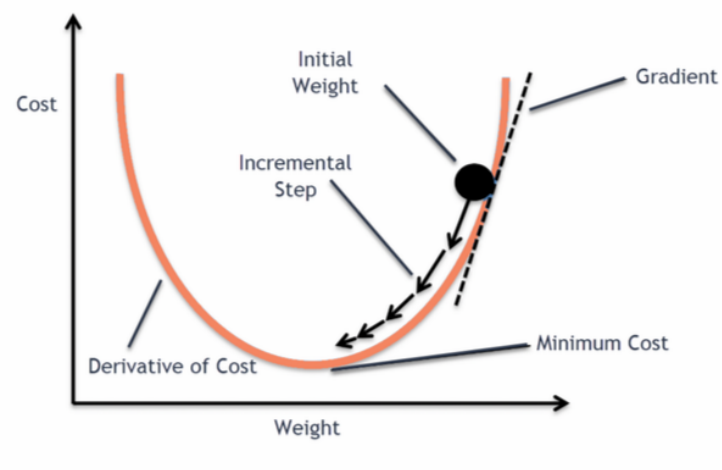


One of the reasons it will be important to work with sigmoids is that these functions are smooth and differentiable, whereas the derivative of the threshold is everywhere, either zero or undefined. So the value of sigmoid ranges from 0 to 1 and the derivative of sigmoid lies in the range of 0 to 0.5. This is one of the reasons for not using sigmoids in the hidden layers.

We need to minimize or approximate the non-convex function which comes from computing the quadratic cost function on the output using gradient descent. This quadratic cost function depends on all of the weights and biases. Inside the network. The gradient is a measure of how changes to the weights and biases change the value of the cost function, and we know we want to move in the direction opposite to the gradient because we want to minimize the cost function.

Let's quickly discuss **what is Gradient descent?**

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost). You can think of it as a large bowl. This bowl is a plot of the cost function (f). A random position on the surface of the bowl is the cost of the current values of the coefficients (cost). The bottom of the bowl is the cost of the best set of coefficients, the minimum of the function. The goal is to continue to try different values for the coefficients, evaluate their cost, and select new coefficients that have a slightly better (lower) cost. Repeating this process enough times will lead to the bottom of the bowl and you will know the values of the coefficients that result in the minimum cost. Read more about gradient descent Here.



Steps in Gradient descent :

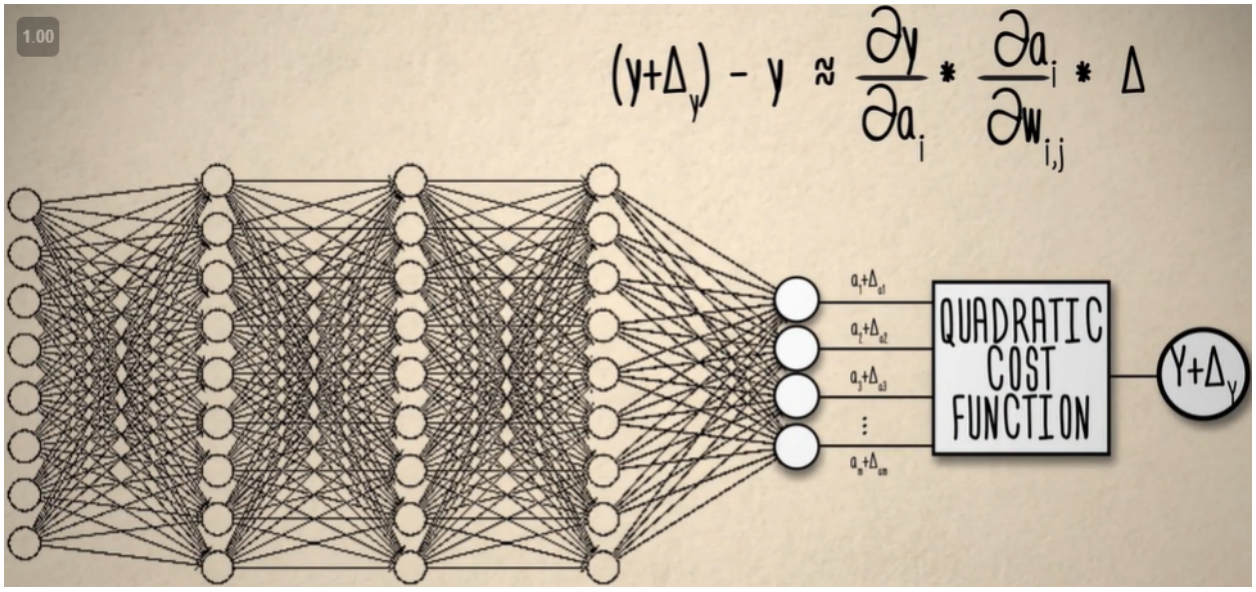
1. Initialize the weights with random values and calculate the cost function.
2. Calculate the gradient. We need to know the gradient so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.
3. Adjust the weights with the gradients to reach the optimal values where cost is minimized.

$$coefficient = coefficient - (learning\_rate * gradient)$$

4. Use the new weights for prediction and to calculate the new cost.
5. Repeat steps 2 and 3 till further adjustments to weights don't significantly reduce the Error.

How do we compute the gradient?

In the context of neural networks, we can compute the gradient using what's called **backpropagation**. This method helps calculate the gradient of a loss function with respect to all the weights in the network. The intuition is much simpler than what backpropagation is really doing is using the chain rule to compute the gradient layer by layer.



Let's get into the intuition, imagine in the m outputs of our neural network are $a_{1,}\, a_{2....}\, a_m$ and these are parameters that we feed into the quadratic cost function, then it's straightforward to compute the partial derivative of the cost function with respect to any of these parameters. The partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant (as opposed to the total derivative, in which all variables are allowed to vary). Here we are going to find a gradient for each of the parameters. This gradient says how much the parameter needs to be changed to get into global minima.

Now the idea is that these M outputs are themselves based on the weights in the previous layer. If we fixed the ith output. It's a function of the weights coming into ith the perceptron, in the last layer, and also its bias, we can compute again how changes in these weights and biases affect the ith output, this is exactly where we need the nonlinear function. In our case, the sigmoid is differentiable. Backpropagation continues

in this manner computing the partial derivatives of how the cost function changes as we vary the weights and the element layer based on the partial derivatives that we've computed for the weights and biases in the layer above. That's it. That's backpropagation.

Now we have all the tools we need to apply deep learning. Let's understand what all steps involved in creating neural networks,

1. Pick a network architecture. Decide the number of layers, number of perceptrons and activation functions, and number of output perceptrons.
2. Random Initialization of Weights: The weights are randomly initialized to a value between 0 and 1, or rather, very close to zero.
3. Implementation of the forward propagation to calculate the cost for a set on input vector for any of the hidden layers. The cost function would help determine how well the neural network fits the training data.
4. Implementation of backpropagation algorithm to compute the gradient.
5. Use gradient descent technique with backpropagation to try and minimize the cost function as a function of parameters or weights.
6. Repeat steps 3 to 5 until the model finds the optimal parameters.

Why does gradient descent on a non-convex function work at all? In low dimensions, it seems obvious that it really does get stuck. But the truth is that no one knows why it seems to work in high dimensions. There are several possible explanations

1. Maybe these functions are closer to convex than we think and are at least convex on a large region of states.
2. Another factor is that what it means for a point to be a local minimum is much more stringent than higher dimensions, a point as a local minimum if every direction you try to move in the function starts to increase. Intuitively it seems much harder to be a local minimum in higher dimensions because there are so many directions where you escape from
3. Yet another takeaway is that when you apply backpropagation to fit the parameters, you might get very different answers depending on where you start. This just doesn't happen with convex functions, because wherever you start from you'll always end up in the same place, like a globally optimal solution.

Also, neural networks first became popular in the 1980s but computers just weren't powerful enough back then, so it was only possible to work with fairly small neural networks. The truth is that if you're stuck with a small neural network, and you want to solve some classification task, there are much better approaches such as support vector machines, but the vast advances in computing power have made it feasible to work with, truly huge neural networks. And this is a major driving force behind their research.

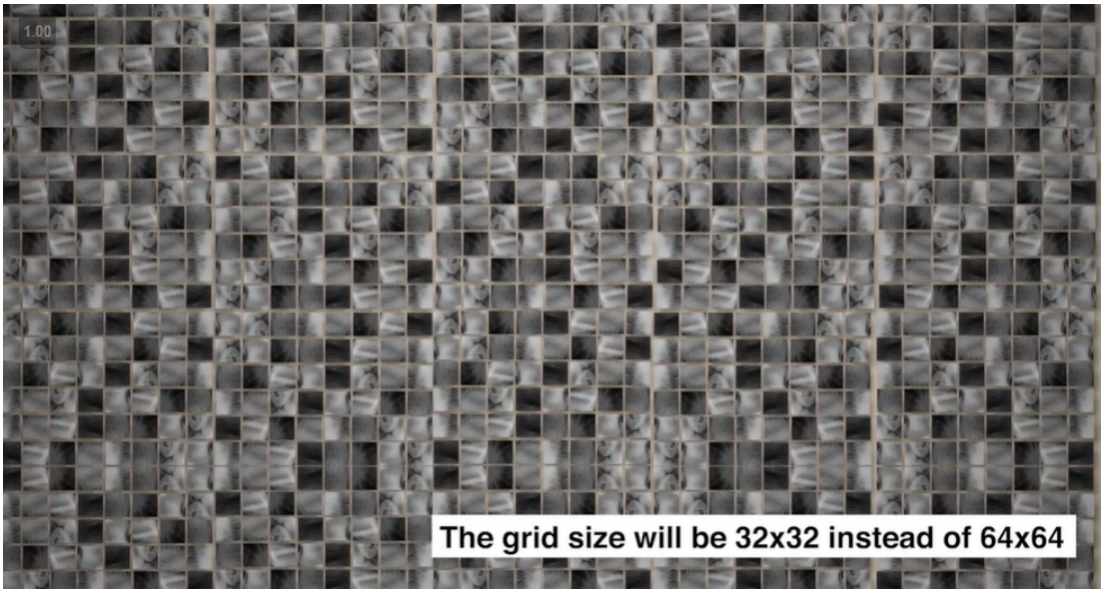
Lastly, about hierarchical representations, We talked about some of the philosophy and connections to neuroscience, there is, at best, a very loose parallel between these two assumptions, and you're advised to not take this too literally. In the early days, there was so much focus on doing exactly what Neuroscience tells us happens in the visual cortex that researchers actually stayed away from gradient descent, because it wasn't, and still isn't clear that the visual cortex can implement these types of algorithms.

3.2.5 Convolutional Neural Networks

Let's understand the notion of a convolutional neural network and how it is different from the neural network we learned.

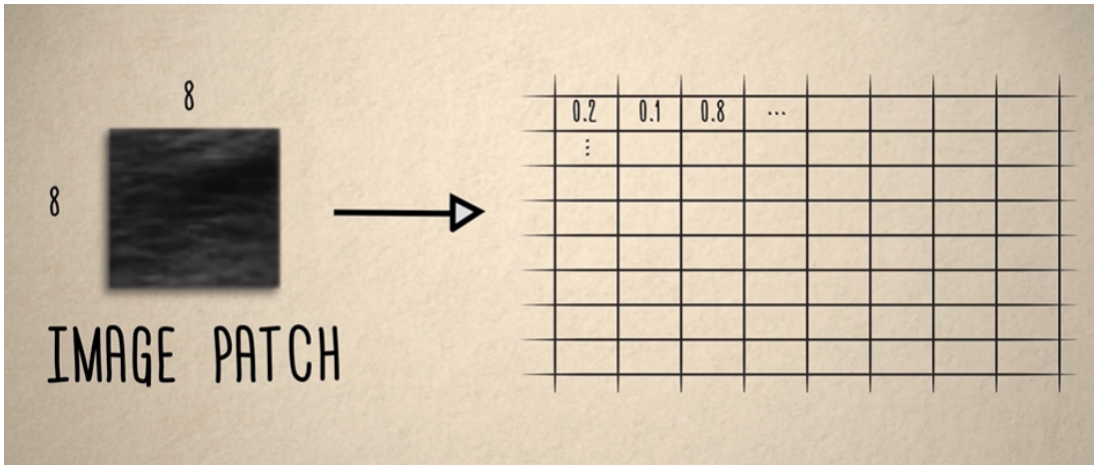
Let’s understand the first layer once. Let's say that our input is a grayscale image that's 256 x 256. In grayscale images, we have only one channel so the image size will be 256 x 256 x 1.

We can break it up into 8 x 8 image patches. The patch is nothing but a small portion of an image.

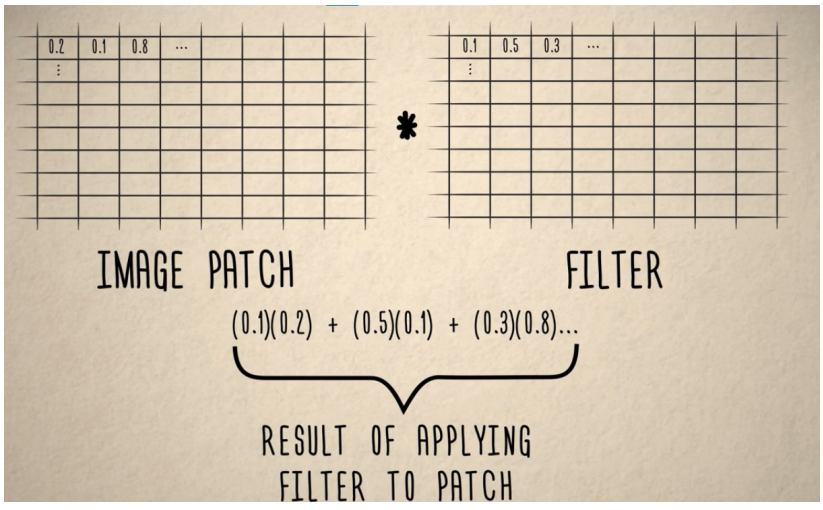


In total, we have a 32 x 32 grid of these patches that comprise the image.

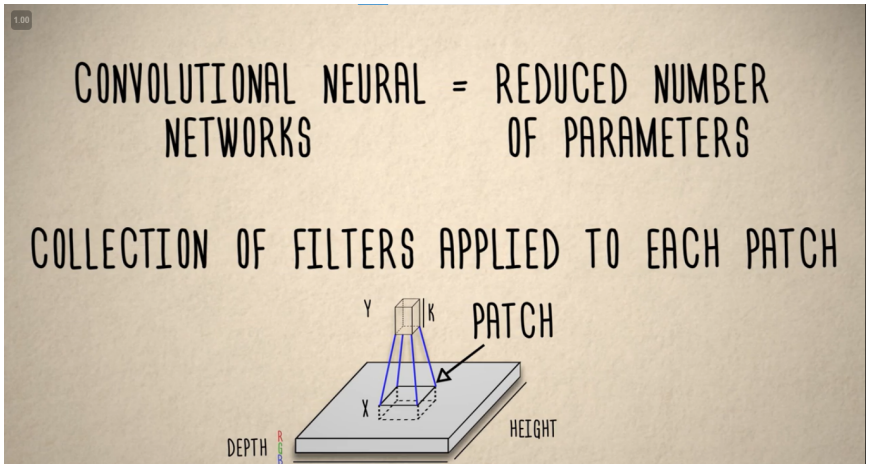
Now let's consider a linear function on just the 8 x 8 image patch. Sometimes this is called a filter.



In our case, a filter is an 8 x 8 grid of weights learned during backpropagation to learn complex patterns in the image grid. And when we apply a filter to an image batch. What we do is we take an inner product between them as vectors.



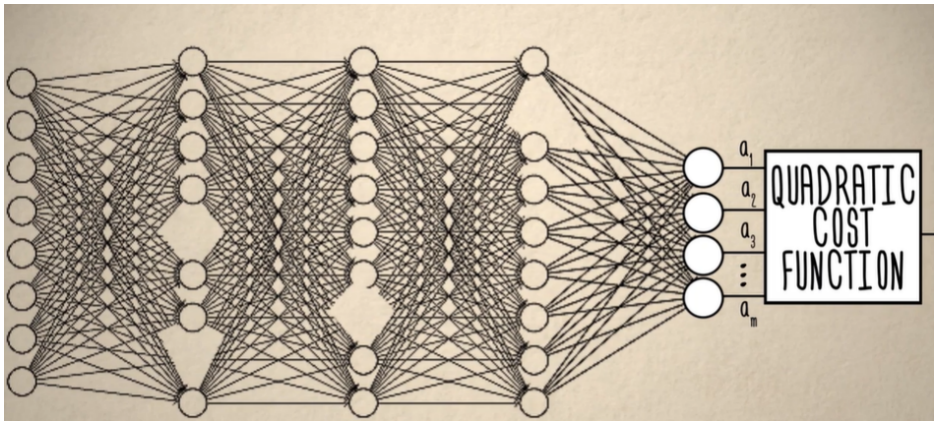
So if we apply a filter to each one of the patches, we get a new 32 x 32 grid of numbers. What good is this? remember the intuition is that the first few layers detect simple objects like edges.



We can think of a filter as a naive object detector, and instead of having different parameters when we apply it to each of the different image patches, why not have the same parameters throughout? This drastically reduces the number of parameters and consequently, there are many fewer things to learn. If we take this idea to its natural conclusion, We get convolutional neural networks.

In the first layer, we have a collection of filters, each one is applied to each of the image patches, and together they give the output of the first layer. After applying a non-linearity at the end. This is already a major innovation because it means we can work with much larger neural networks in practice. Just the first few layers are convolutional and the others are general and fully connected.

Another important idea is the notion of **dropout**.



Here when we compute how well a neural network classifies some image say through the quadratic cost function. We instead randomly delete some fraction of the network,

and then compute the new function from the input to the outputs. The idea is, if a neural network continues to work, even if we drop perceptrons from the intermediate layers then it must be spreading information out in a way that no node is a single point of failure. Training a neural network with dropout makes the function we learn more robust.

Additional content :

We have understood neural networks and how they can be used to build robust models using numerical data. Let's assume we have an image with height =6, width =6, and the number of channels =3 (Coloured image). So there are 6 x 6 x 3 =108 numbers. Consider we have the first hidden layer of a neural network with 10 units. The total number of parameters(Weights) are 108 x 10 =1080. So we need 1080 weights for only one layer and Generally, the size of the image will be equal to 224 x 224. In these kinds of cases, we get a lot of parameters to train which makes it computationally expensive and the model doesn't perform better. To deal with such problems, we have special neural networks called **Convolutional Neural Networks (CNNs).**

A convolutional neural network is a type of neural network that is used in image processing and image classification. This neural network takes the pixels of an image as input and generates the desired output.

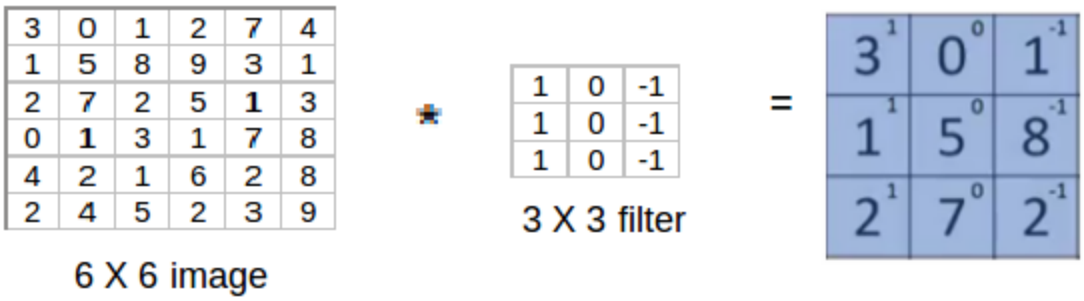
Let's understand the building blocks of CNN,

1. Convolution
2. Pooling
3. Padding
4. Stride
5. Fully Connected layer

Convolution :

The first step of a CNN is to **detect features** like edges, shapes e.c.t.This is done by applying a convolution to the image using filters (Filters are responsible for detecting some kind of shape).

Let's understand this using an example, We take an input image of 6 X 6 and convolve this 6 X 6 matrix with a 3 X 3 filter:



After the convolution, we will get a 4 X 4 image. The first element of the 4 X 4 matrix will be calculated as we take the first 3 X 3 matrix from the 6 X 6 image and multiply it with the filter. The first element of the 4 X 4 matrix, will be the sum of the element-wise product of these values which is

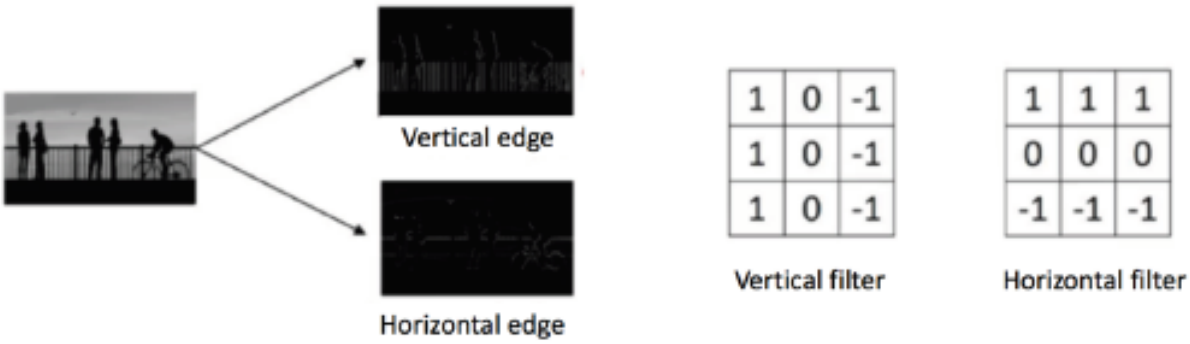
$= (3 \times 1) + (0) + (1 \times -1) + (1 \times 1) + (5 \times 0) + (8 \times -1) + (2 \times 1) + (7 \times 0) +$

Similarly, we will convolve over the entire image and get a 4 X 4 matrix.

In convolution the total parameters = 3 x 3 x 3 (numbers in filters) x 10(filters) +10(bias) =280. The number of weights is very less compared to the Neural Network .

Filters:

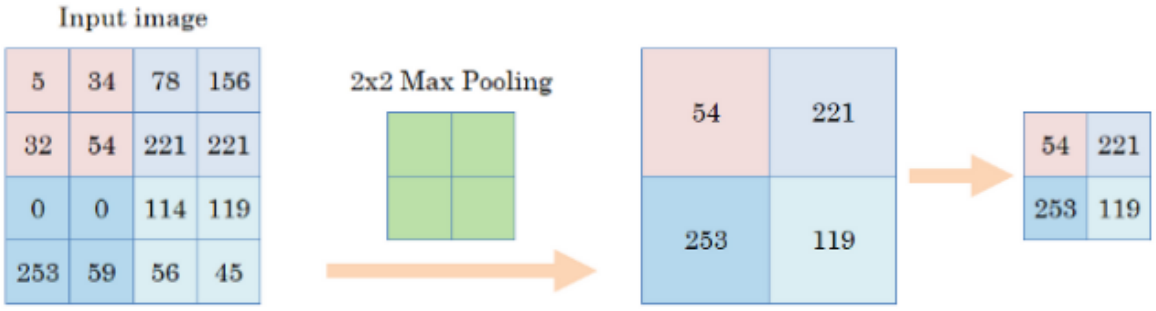
Filters are responsible for locating objects in an image by detecting the changes in intensity values of the images. Generally, we have an edge detector that detects edges in an image. For example



These filters are responsible for detecting vertical edges and horizontal edges.

So in images, we have a lot of complex features to get detected other than edges. For that purpose, we randomly choose filter values which the model can learn itself during backpropagation.

Pooling:
It's used to reduce the spatial size of the representation to reduce the number of parameters and computation in the network.



Strides

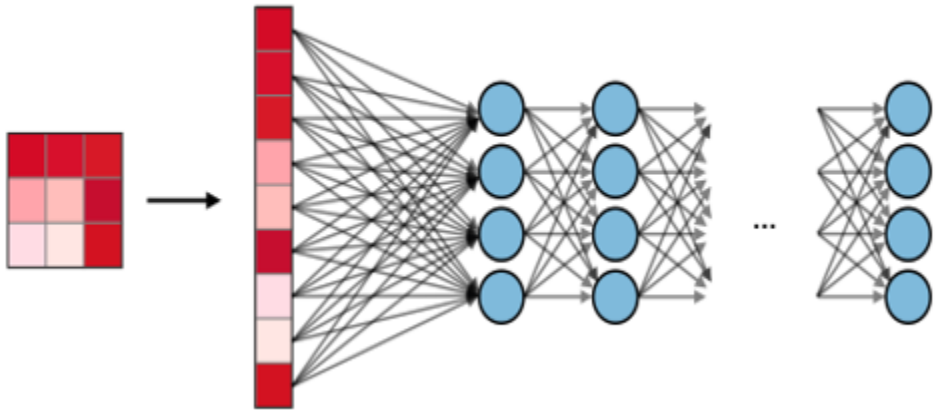
While performing convolution, we observe that we slide from the top-most corner to the bottom-most corner by a shift, this shift is called stride. Thus stride helps us in dimensionality reduction making the output dimension an integer rather than a fraction and we can also observe that as the number of strides increases the computational power required for computation decreases.

Padding :
The convolutional layers reduce the size of the output. So in cases where we want to increase the size of the output and save the information presented in the corners, in that case, we can use padding layers where padding helps by adding extra rows and columns on the outer dimension of the images. So the size of input data will remain similar to the output data. We mostly add zeros in the extra rows and columns.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Fully connected layer:

The result after applying a different filter is a matrix. We have to convert that matrix in the form of a vector to feed it into the neural network. For this, we make a fully connected layer. From the picture shown below, the 1st matrix is the result we get applying different filters and the second layer is the fully connected layer generated from the matrix.



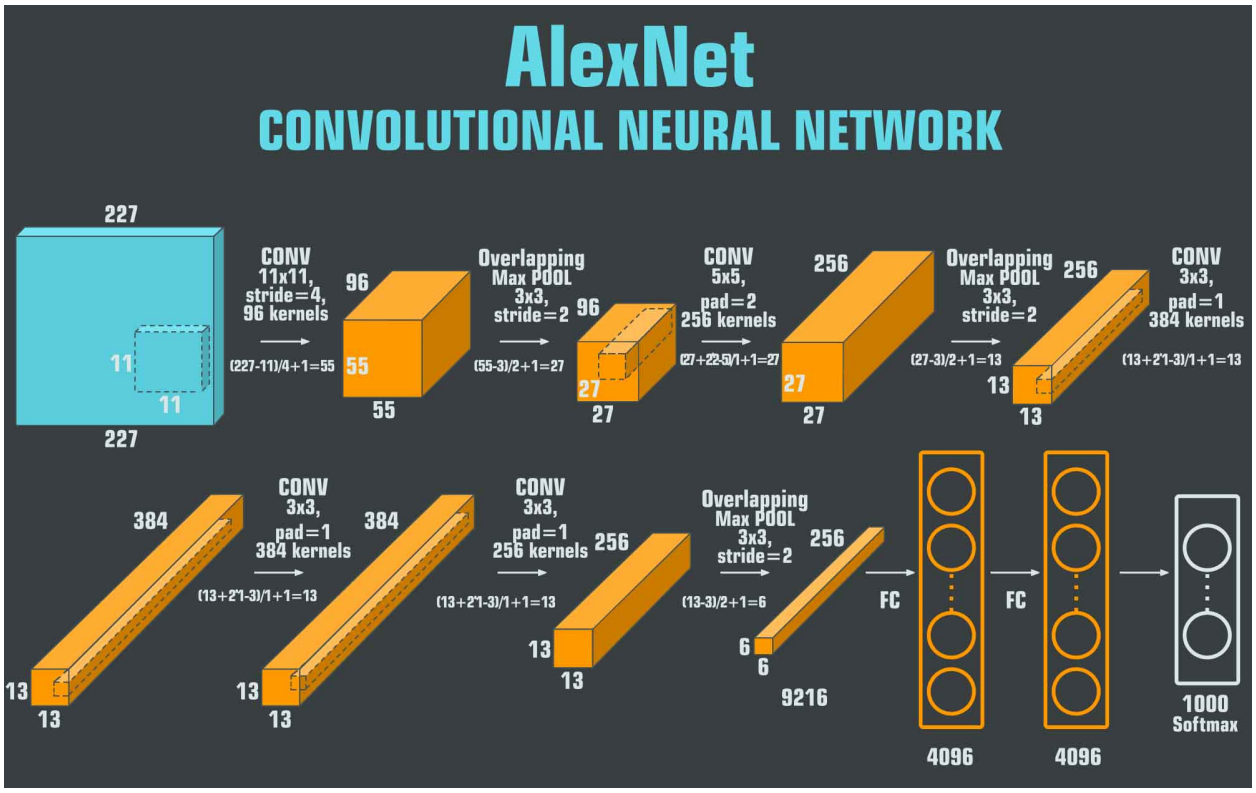
After getting a fully connected layer, we pass this layer as an input layer to the neural network in order to get the results.

We now have an understanding of the building blocks. CNN's are nothing but arranging these building blocks properly. Usually, the order will be a convolution layer followed by a pooling layer multiple times and finally a fully connected layer.

Let's see one of the architectures of CNN,

AlexNet :

- AlexNet is a masterpiece created by the SuperVision group, which included the masterminds Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever from the University of Toronto.
- The winner of ImageNet-2012, AlexNet showed that deep learning was the way towards achieving the least error rates



What is the architectural structure of AlexNet?

- The major feature of AlexNet is that it overlaps the pooling operation to reduce the size of the network.
- With five convolution layers and three fully connected layers, and the ReLu function applied after every Convolutional layer and fully connected layer, AlexNet showed us the way towards achieving state-of-the-art results in image classification.
- It uses the ReLu as its activation function, which speeds the rate of training and increases the accuracy. The regularization technique it uses is a dropout.

Explore the other kinds of architectures to get a better understanding of which building blocks are to be used in the applications of CNN.

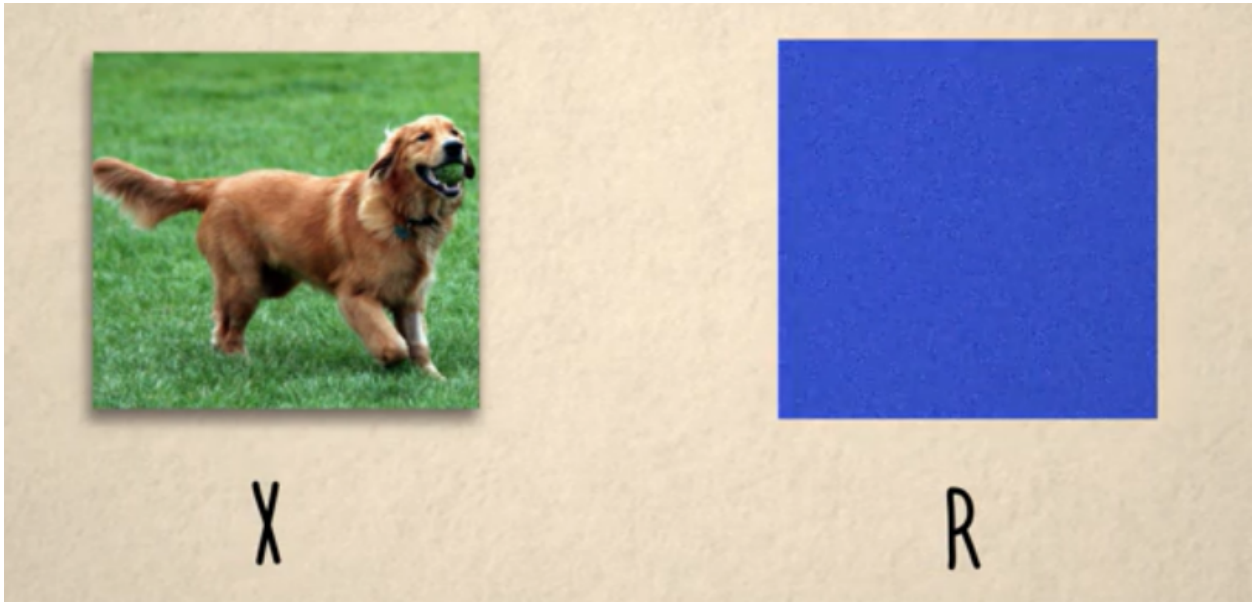
.

3.2.6 Limitations of Deep Neural Network

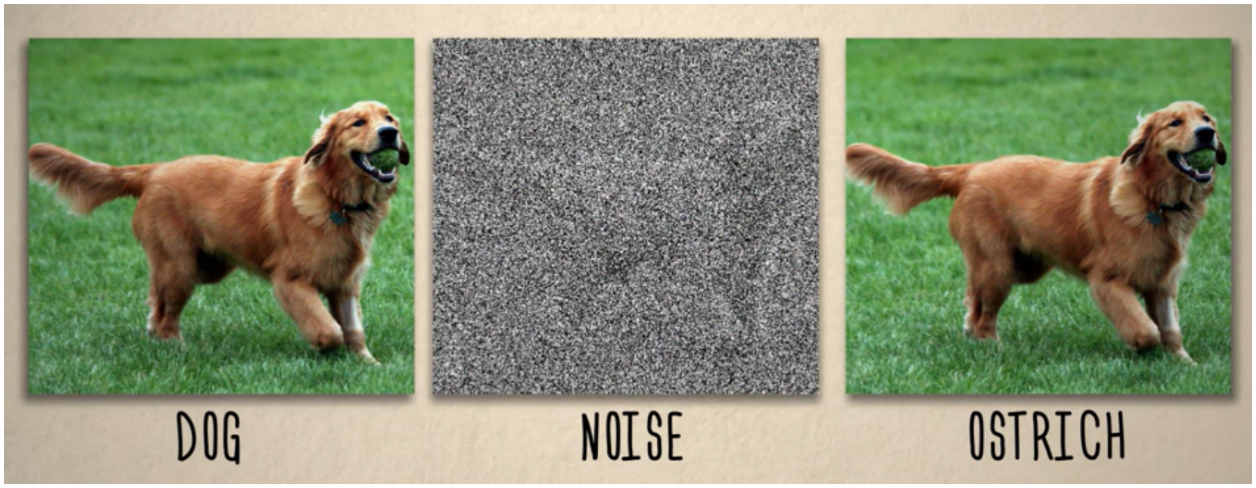
You got the sense of the excitement surrounding deep networks, but now it's time to take a sobering look at a more frightening aspect. In the examples, we've seen deep learning has been able to conquer some very hard challenge problems. And that's a great reason for being optimistic that they'll lead to practical self-driving cars very soon, and all sorts of other technological advances. But there's an important word when we talked about fitting deep networks parameters. We discussed the difference between convex and nonconvex functions, but the truth is that we don't really understand why things like gradient descent work so well. And even then, the actual features that it finds are difficult to interpret. We can't readily take a deep neural network that we've learned and see why it's working so well.

This leaves a lot of room for manipulation. What if we start with an image that the network successfully classifies, how much do we have to change the image to change the label that the network assigns ? Let's say I give you an image and a target label, say ostrich.

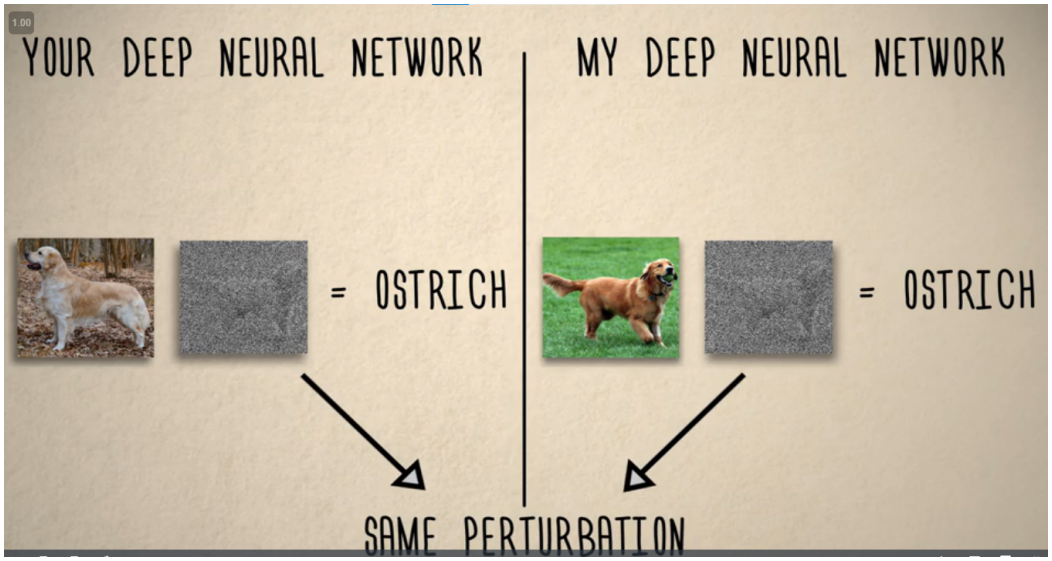
I want you to fool the deep network into thinking that the image is an ostrich, even though it's very clearly not. As usual, we represent the image as a high dimensional vector x. Now we can look for the smallest vector r, which we think of as a perturbation to the image.



So that when the deep network is applied to x plus r, the output has a one in the ostrich category and zeros in all the items. Finding such an R is a non-convex optimization problem. But as usual, there are approaches for approximately solving it nonetheless. Obviously, if I'm allowed to change the image entirely, then I can make the deep network think it's an ostrich, by making the picture. The question is **how small can the perturbation be?** It turns out that you can get away with a perturbation that's imperceptible to humans. Now let's put this in perspective, much of the initial excitement of deep learning came from the fact that we're now able to classify images automatically with error rates that are even slightly better than what humans can achieve. But the classification function that they've learned is so discontinuous and unwieldy, that changes ought not to change the category. In fact, you wouldn't even see them as a change in the image at all, and can be used to fool the deep network into thinking the images, whatever category, you'd like.



In fact, what's even more frightening is that the same perturbation works across many neural networks. So you could think that if I don't know what neural network you've learned, and don't have access to the millions of weights and biases that define your model, I won't be able to fool it, but it turns out that I could learn my own neural networks, even using a subset of the training data that you've used, find a perturbation for a given image that makes my network think that it's an ostrich. And the same perturbation would work for yours too.



So all this time we've been talking about deep neural networks being able to compete with or even outperform humans, if they can find new moves and the ancient game of Go, then why not solve other sorts of decision-making problems for us? We can imagine relying on them to keep us safe on our daily commute in self-driving cars or diagnosing us based on our symptoms of medical history or even estimating the credit

risk of an individual by incorporating more information than currently goes into your FICO score. But all of these wonderful possibilities come with the caveat that if we don't understand why deep learning works, the classifiers we learn might be subject to all sorts of nefarious manipulations, or might learn unfair and uncontrollable biases and how they make decisions.

To understand more you can refer to the links given below.
https://heartbeat.fritz.ai/how-to-trick-computer-vision-models-a78e081c2326
https://arxiv.org/pdf/1412.6572.pdf