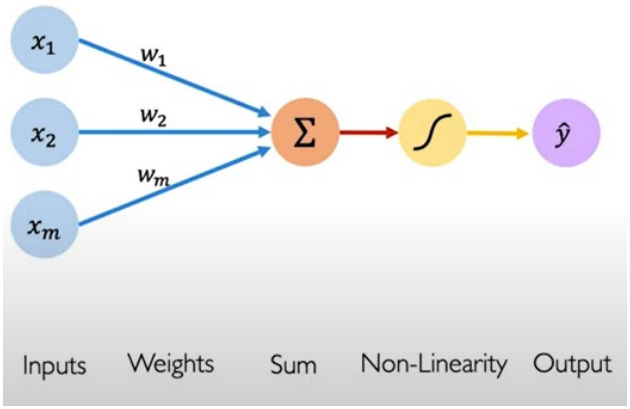
Deep Learning

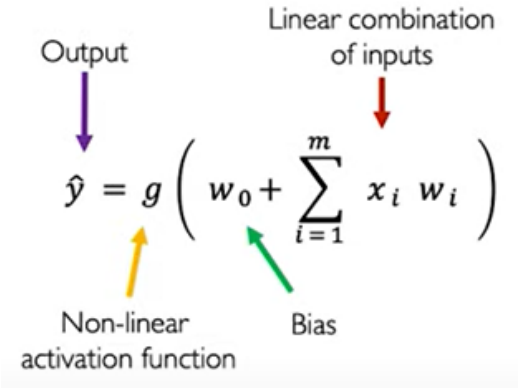
Why Deep Learning and Why Now?

Traditionally, machine learning algorithms define a set of features in their data. Usually, these are features that are handcrafted or engineered and as a result they tend to be very brittle in practise when they're deployed.

The key idea of deep learning is to learn these features directly from the data in a hierarchical manner. For learning a face, we can start with the edges, composing them together, detect mid-level features such as eyes, nose or mouth and go deeper to compose them into structural facial features to ultimately recognising each face. This hierarchical learning is core to deep learning.

Fundamental building block – Neuron or Perceptron





We have a set of inputs {Xi, i = 1, 2, …n}. each of these inputs are multiplied respectively by weights. And then added together. We also have something called the 'bias' term which allows us to shift the activation function.

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

$$\text{where: } \boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

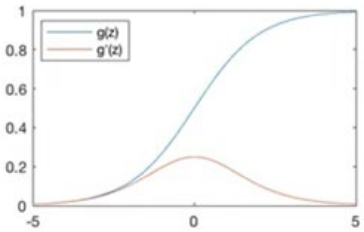
matrix representation.

'g' is a non-linearity function. There are many types of non-linearity functions.

The sigmoid function takes any number, and outputs it between 0 and 1 making it highly suitable for problems in probability.

Under modern Deep Learning problems, ReLU is popular function because of its simplicity. It is a piece-wise linear function, it is zero when in the negative regime and it is strictly the identity function in the positive regime.

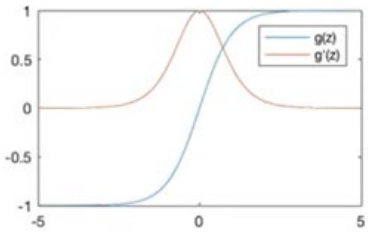
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

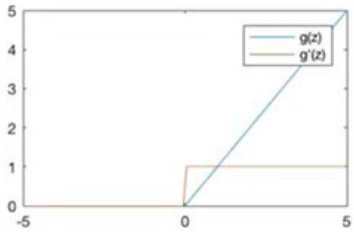
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

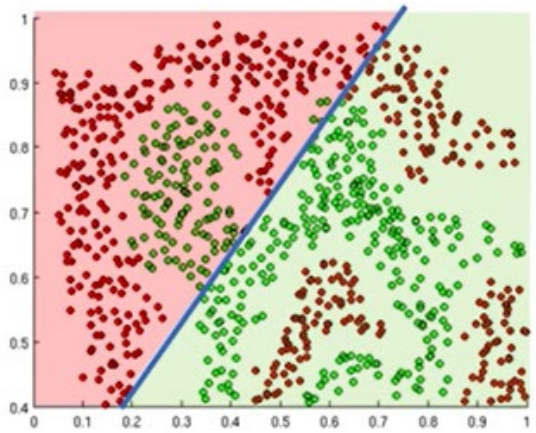


$$g(z) = \max(0, z)$$

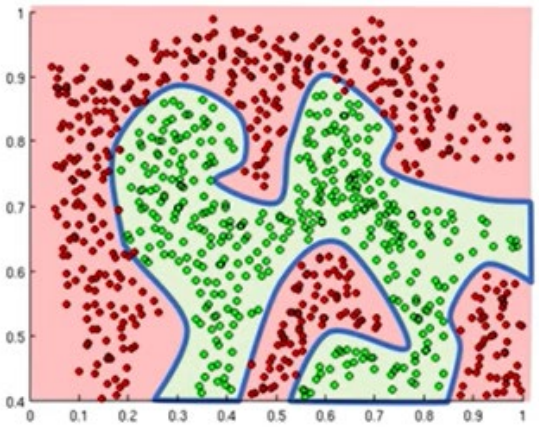
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Why do we need activation functions?

The purpose of activation functions is to introduce non-linearities into the network.



Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

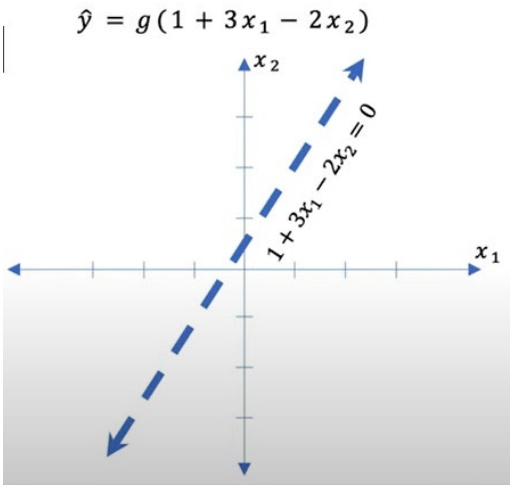
If we use linear activation functions, the separation outcome is poor. But, using non-linear activation functions allows us to approximate arbitrarily complex functions and produce better results.

We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

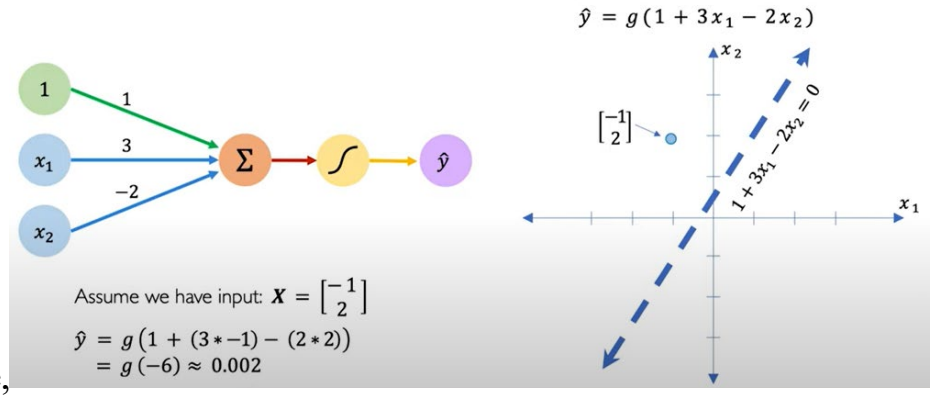
This is just a line in 2D!

Suppose, we have . We can plot this 2D line

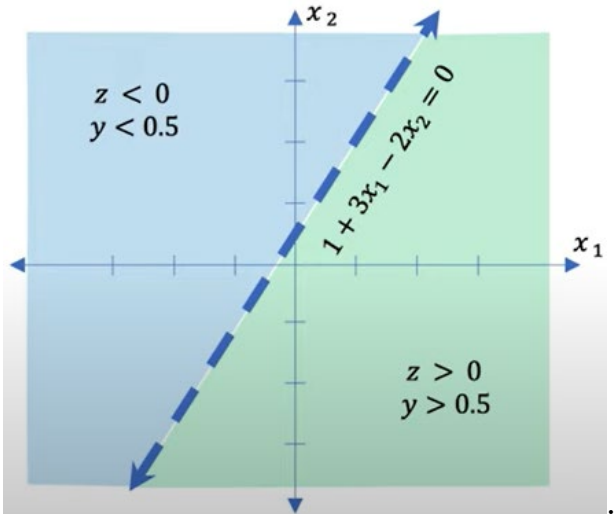


this 2D space is called the Input-space with x1 and x2 on respective axis. This line corresponds to all the decisions the neural network can make.

For a new points we have,



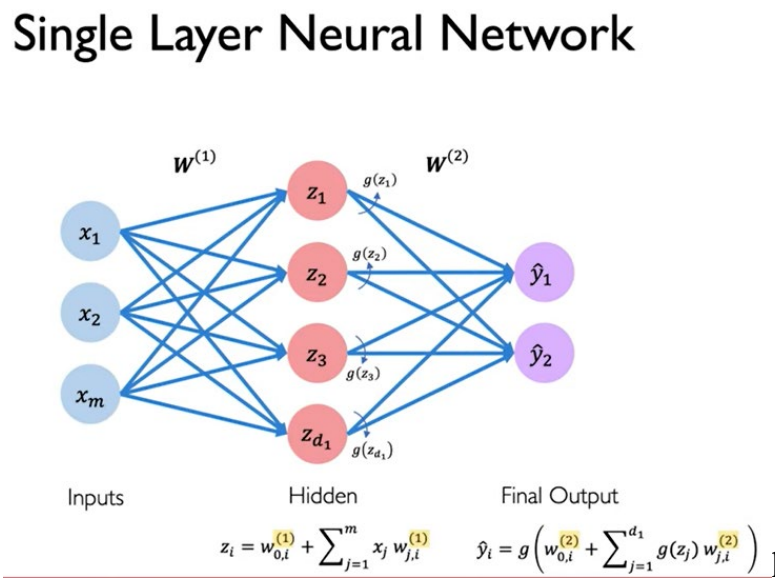
the sigmoid function, divide the input-space into 2 regimes shown by this plot



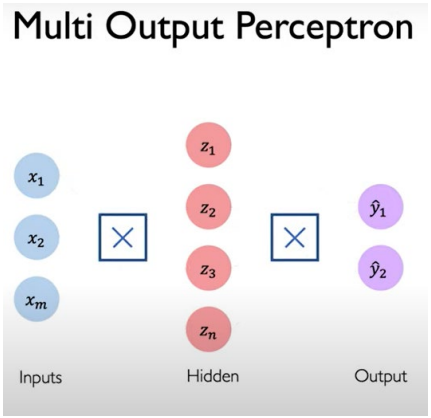
. Since we have only a few inputs and weights, we

can visualise this here. But for millions of data inputs and weights, it becomes difficult to make such plots.

Single Layer Neural Network

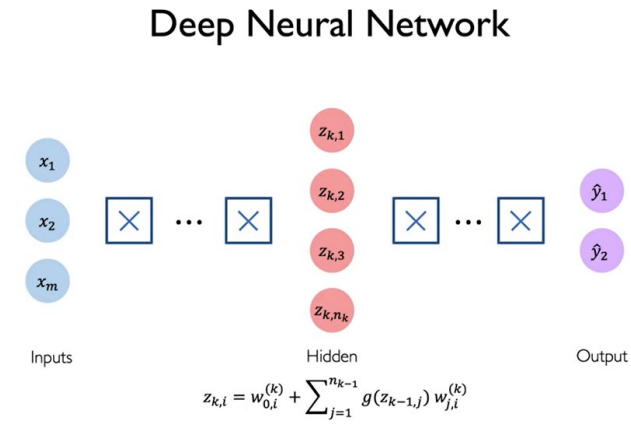


here, we have a single hidden layer between inputs and outputs. This is called hidden layers because unlike the input and output layers, the states of this layer is typically unobserved, hidden to some extent. Each layer has a specific weight matrix $W^{(1)}$ and $W^{(2)}$.



To clean things up a little, we depict the network as the symbol represents a dense layer or a fully connect layer.

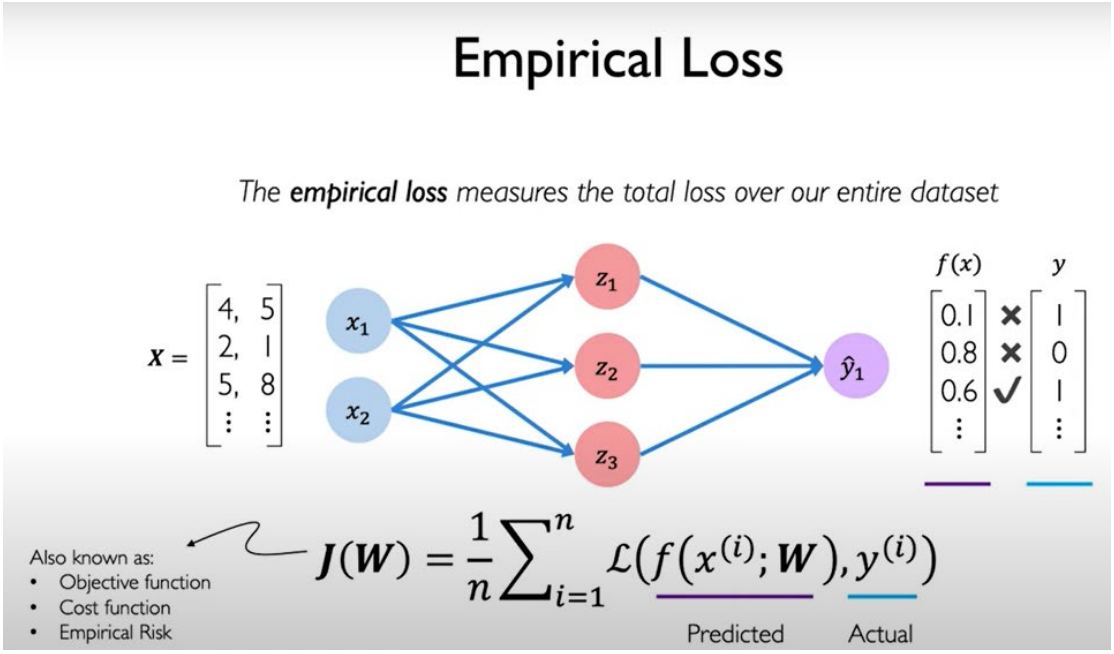
To create a deep neural network, we need to keep stacking these dense layers together



.

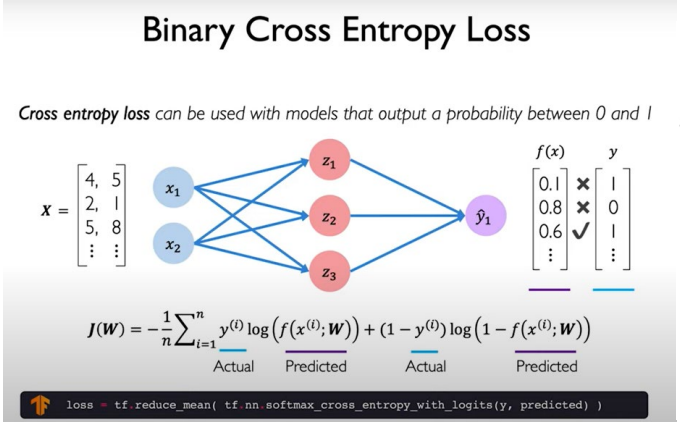
Empirical Loss

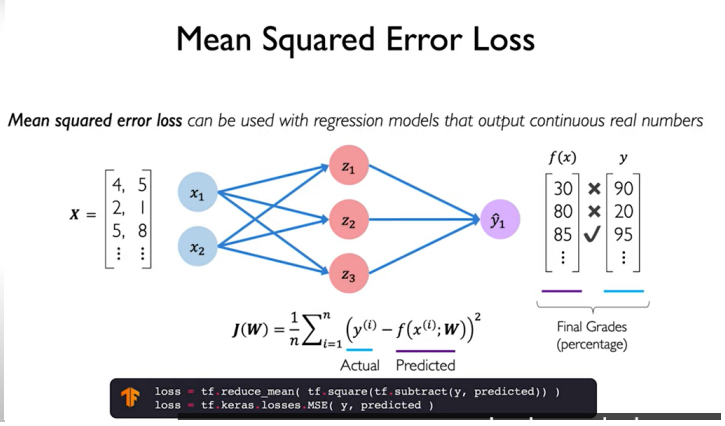
Empirical loss is the average of individual losses over the entire database. We want to minimise the empirical loss between our predictions and the true outputs.



Binary Cross Entropy Loss

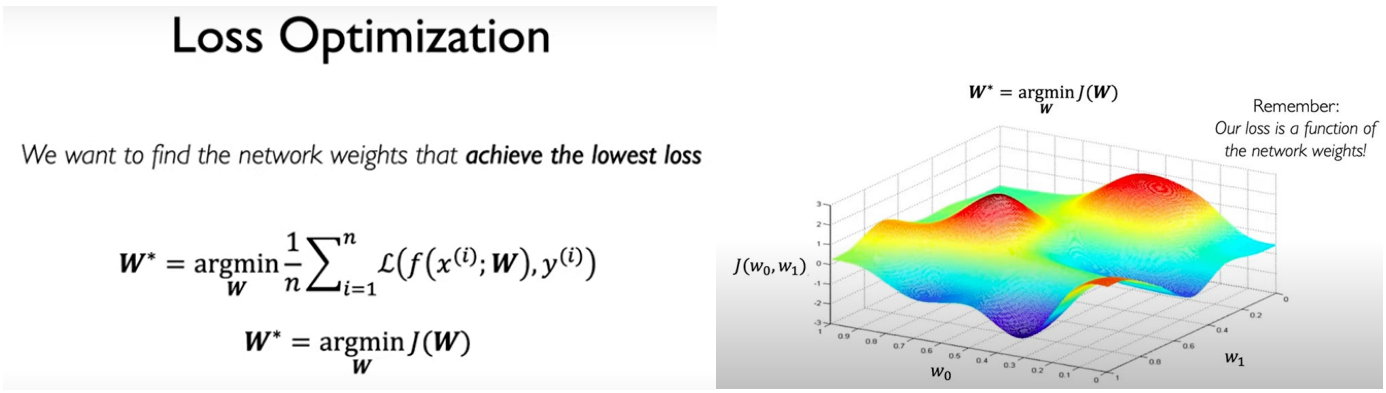
Here we look at the problem of binary classification where the neural network has to answer yes-no or 1-0. We can use, what is called, a soft mac cross entropy loss. It is actually defined by cross entropy between 2 probability distributions, it measures how far apart the ground truth probability distribution is from the predicted probability distribution.





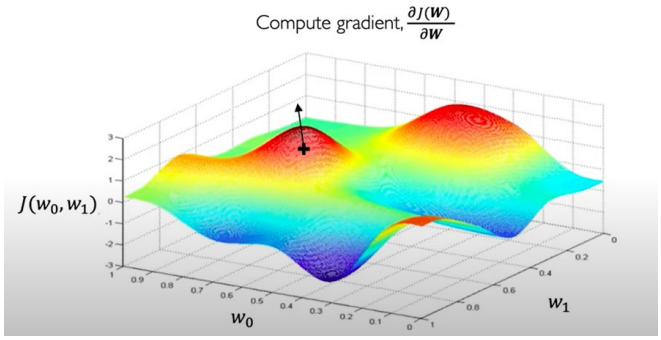
In the other case, if we want to predict a final score instead of pass/fail, because the nature of the output is different (continuous instead of categorical), we need a different loss function. Here, we can use Mean Squared Error.

Training neural network – loss optimization



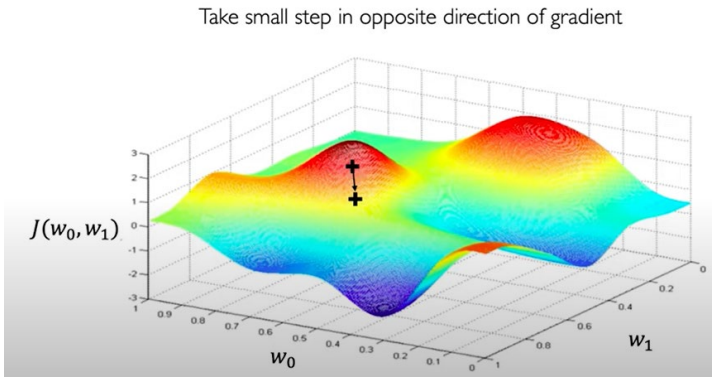
Now because the loss function is a function of the weights and we have only 2 weights, we can make this plot. The aim of training a neural network is to find the lowest point on this landscape which tell us the optimal W0 and W1 values.

1. Randomly pick an initial (w0, w1)
2. Compute gradient of the landscape at that point. This gradient tells us the direction of the highest or steepest ascent i.e. which way is up/ down.

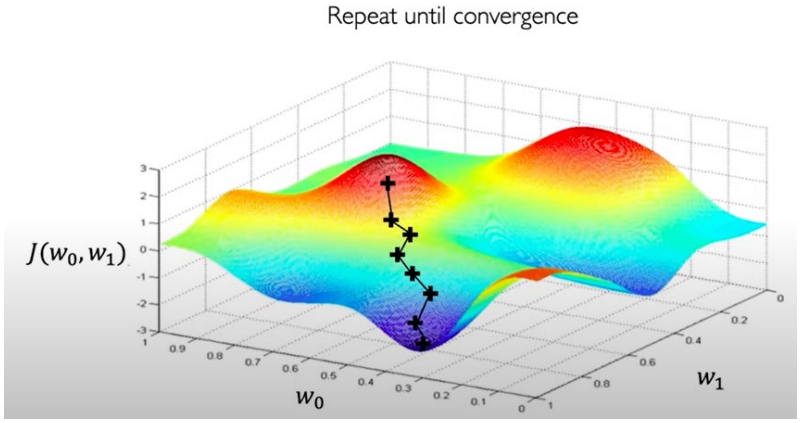


instead of going up, we want to go down, so we take the negative.

3. Take a small step in opposite direction of gradient.



4. Repeat until converge - We repeat the above steps to reach the local minimum.



This process is called the Gradient Descent.

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

to take a small step, we multiply the derivative with a factor 'η' which is called the Learning Rate.

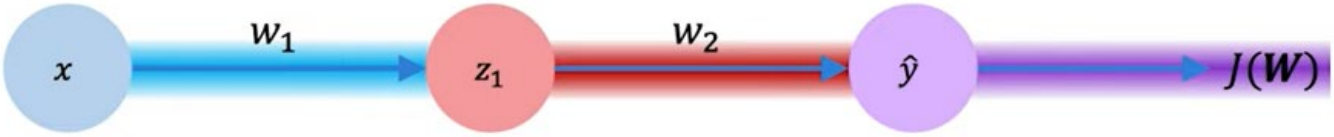
Backpropagation

How do we compute Gradient? Let's look at a simple neural network with one input, one hidden layer and one output.

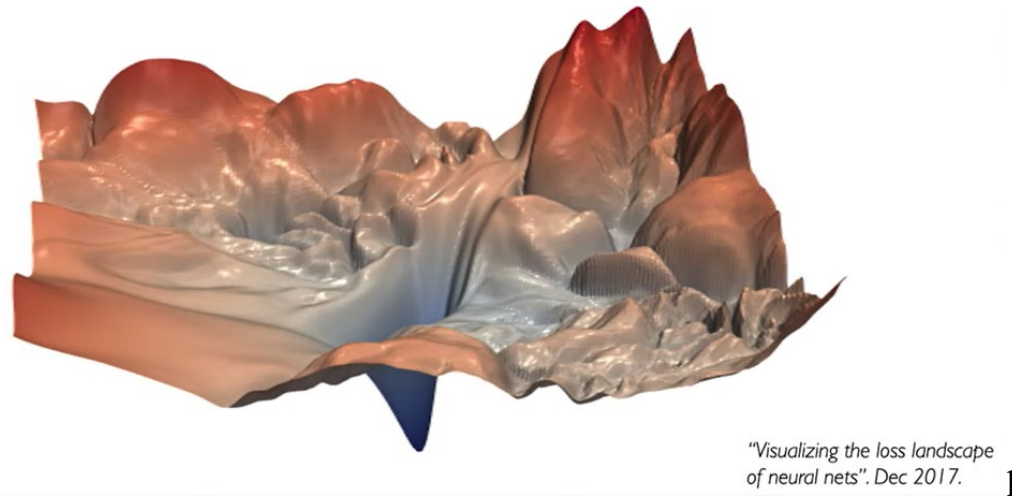
How does a small change in one weight (ex. w_2) affect the final loss $J(W)$?

to know how a small change in the value of w2 affect J(W) can take the derivative $\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$ and break it using the chain rule.

We can do this with respect to w1 as well, $\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$ and expand it further $\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$



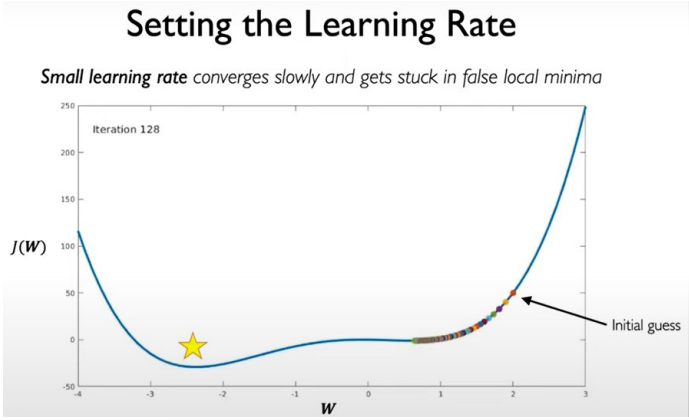
In theory this backpropagation algorithm sounds simple, but in practise this process is much more complicated.

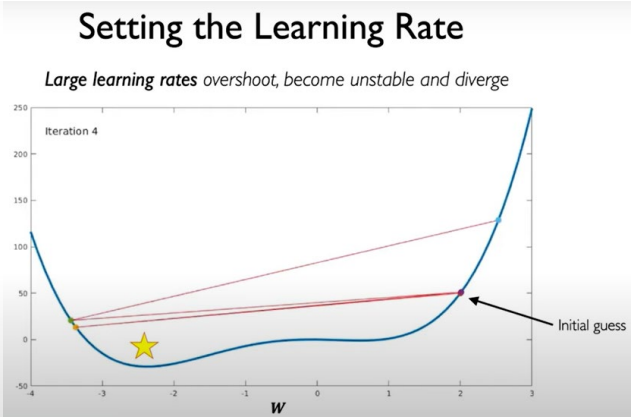


here, we can see that there are a lot of local minimums (non-convex) making gradient descent ineffective. If we start of on a random point on the landscape, chances are we get stuck at the local minima and not reach the global minimum.

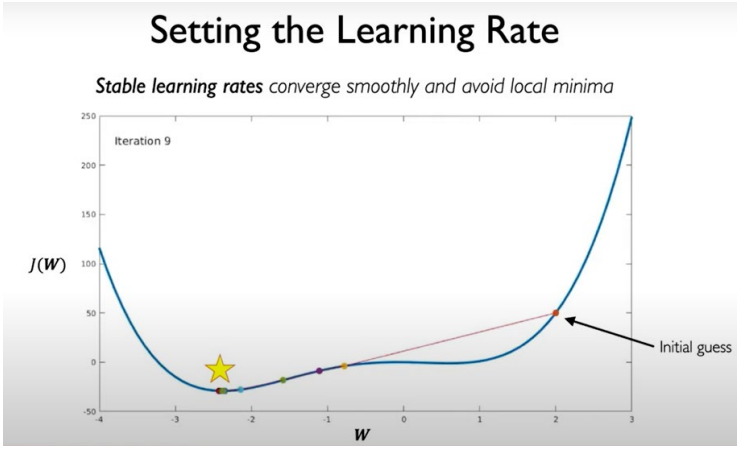
The gradient tells us the direction but not the magnitude of the direction. Learning rate η tells us how big a step we need to take.

If the learning rate is too low, we can get stuck on the local minima given a non-convex loss landscape and it will take time to converge to that minimum.





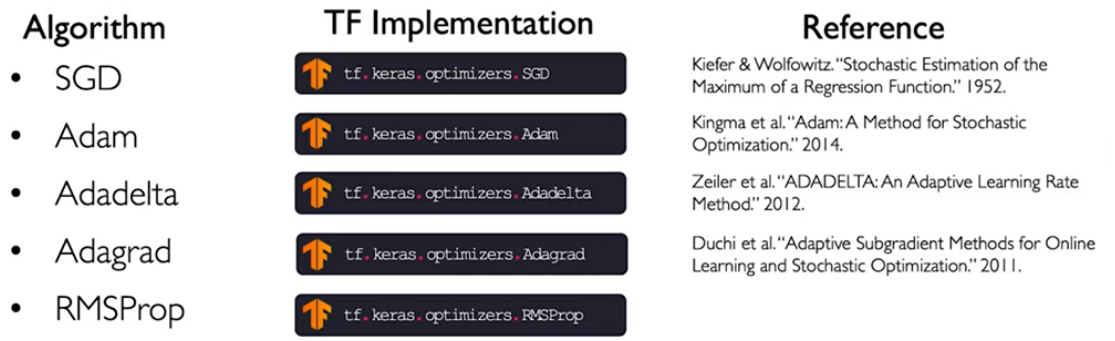
If the learning rate is too high, we can overshoot our minima and diverge.



How to find the best learning rate?

- Try different learning rates
- Design an adaptive learning rate that 'adapts' to the landscape.

Adaptive Learning Rates can be made larger/ smaller depending on how large the gradient is, how fast learning is happening, size of particular weight and so on.



Neural Networks in practise – mini batches

Under gradient descent, it becomes computationally intensive to compute the gradient using backpropagation especially if it is computed over the entire training set. Is essentially is a summation of every data over the entire data set.

Stochastic Gradient Descent

The idea here is to pick a single point and calculate gradient with respect to the weights and then update all the weights based on the gradient.

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights

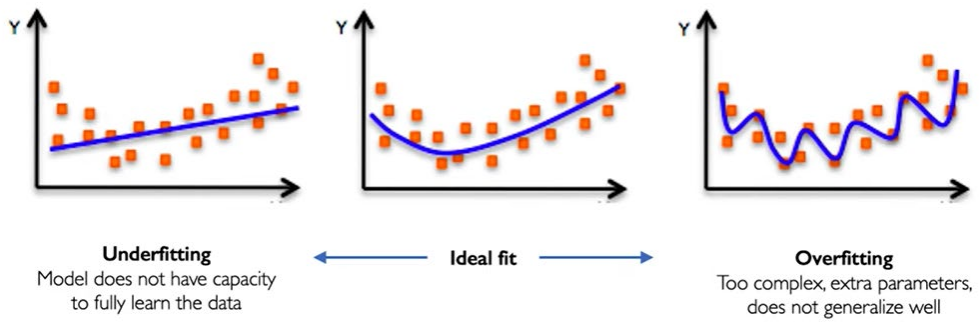
This is very easy to compute because it is only using one-point but it also very noisy because it is from only one-data point.

Instead of a single-point, we can use a 'batch' of B points and we compute the gradient estimate simply as the average over this batch. It is a faster and much more accurate way compared to purely stochastic descent.

Algorithm

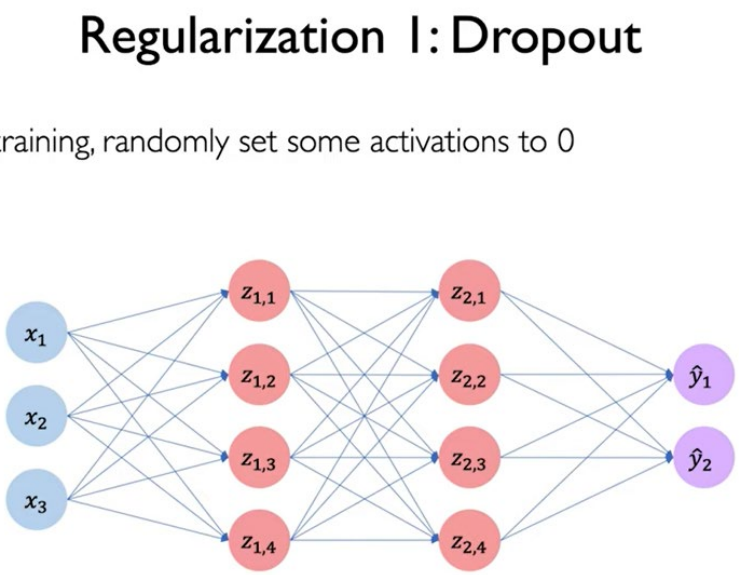
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights

Overfitting – problem of generalisation

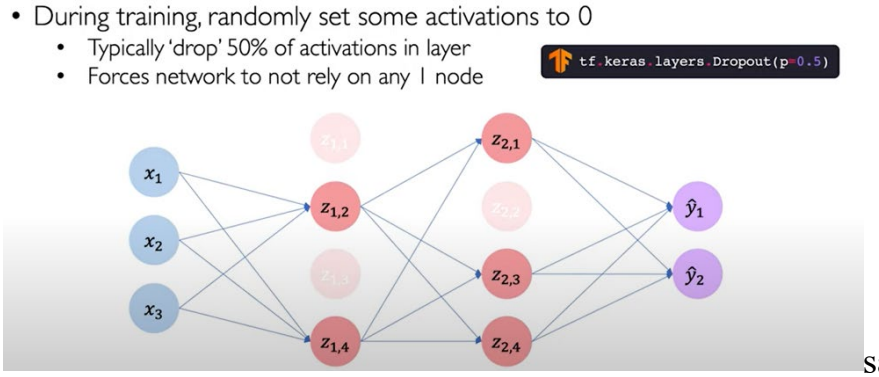


Solution = Regularisation. It is a technique that constrains our optimization problem to discourage complex models.

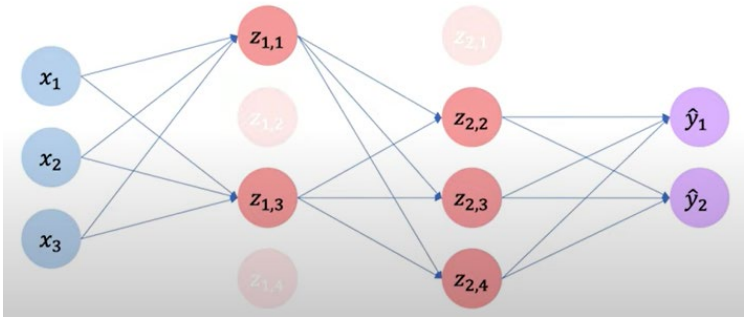
Dropout



during training, we randomly set some of the activations to zero with some probability.



say we take a probability of 0.5, so we randomly drop 50% of the neurons – it forces the algorithm to perform better on test sets. This not only reduces the reliance of network on a particular node making training stronger but it also cuts down the computational time.
On every iteration, we dropout different sets of 50% neurons.



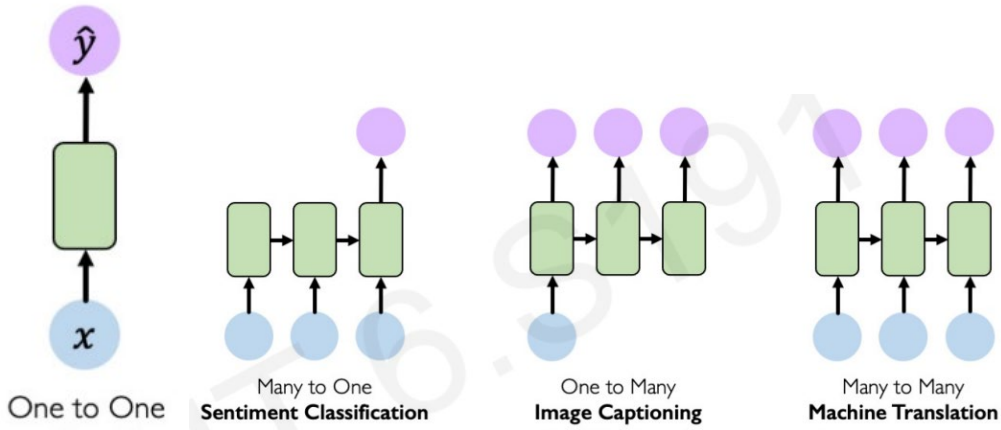
Early Stopping

Idea is to stop training when we realise that the loss is increasing. Increasing of loss is indicative of overfitting.

The loss on training set will always go down as long as the network can memories (i.e. overfit) the training data and simultaneously, the loss on test will increase.

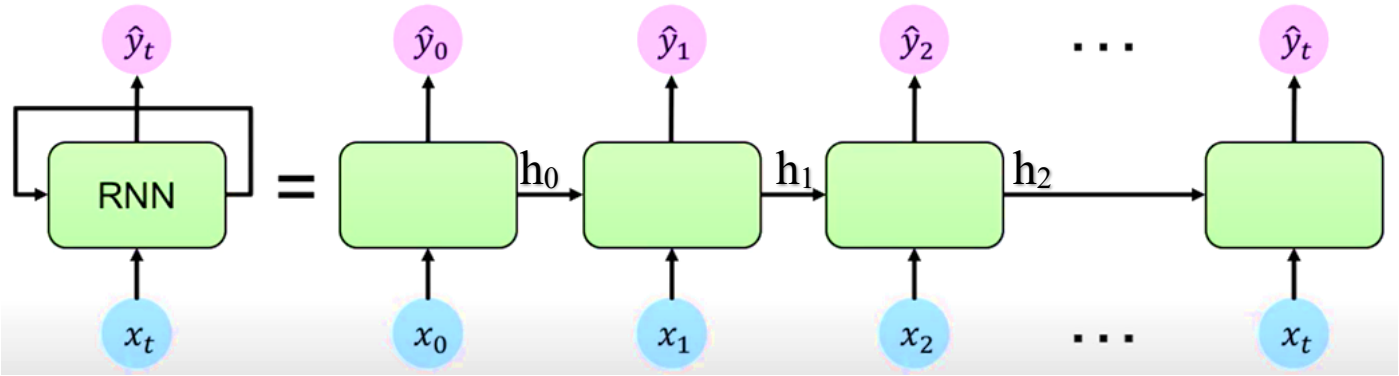
Recurrent Neural Network

Standard Feed Neural Network



here, data is propagated in 1-direction from Input to output. This cannot handle sequential information about previous events.

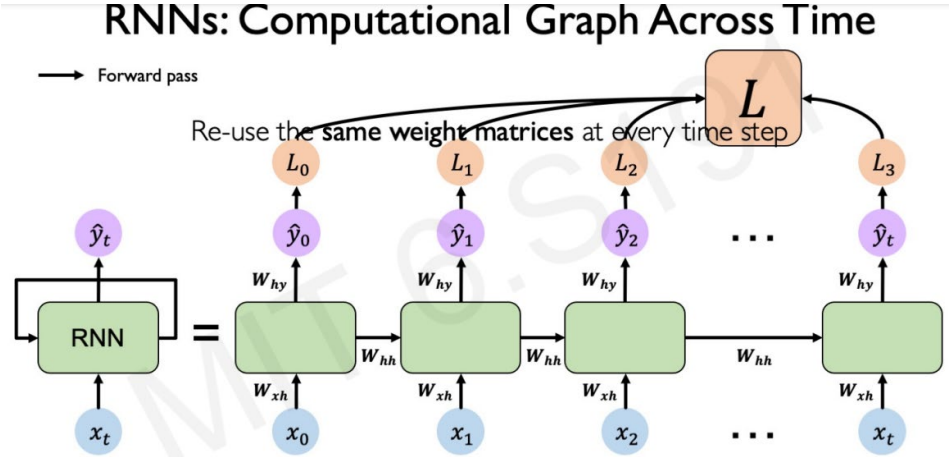
Networks like these require sequential modelling/ networks.



RNN networks have a 'loop' inside the network which allows information to persist over time. Input Xt at time step 't' and output Yt. Additionally, it generates internal state update 'ht' which contains information about the internal state, which is passed on to the next time step.
Such networks with loops in them are called RECURRENT because information is passed from one time step to another, internally within the network.
How?
By using a simple recurrence relation to process sequential data. Specifically, RNNs maintain this internal state $h_t = f_w (h_{t-1}, x_t)$ and at each time step they apply a function 'f' which is parameterised by weights 'w' to update the state 'h'
This update is based on the 'h_{t-1} *and* x_t' (old state and current input).
Note - the same function and the same set of parameters are used at every time step.
Updating hidden state → $h_t = tanh (W_{hh}^T h_{t-1} + W_{xh}^T X_t)$
Output vector → $\hat{Y}_t = W_{hy}^T h_t$
This is how RNNs update the hidden state and produce an output.

Computational Graph Across Time

Here we unroll the loop over time. This can be thought of having multiple copies itself where each passes a message to its descendent.



At every time step, we can compute a Loss and this computation of loss will then complete out forward propagation through the network.

Total Loss $L = L_0 + L_1 \ldots L_t$.

This total loss contains individual contributions over time. This means that during training, one will have to somehow involve this time component.

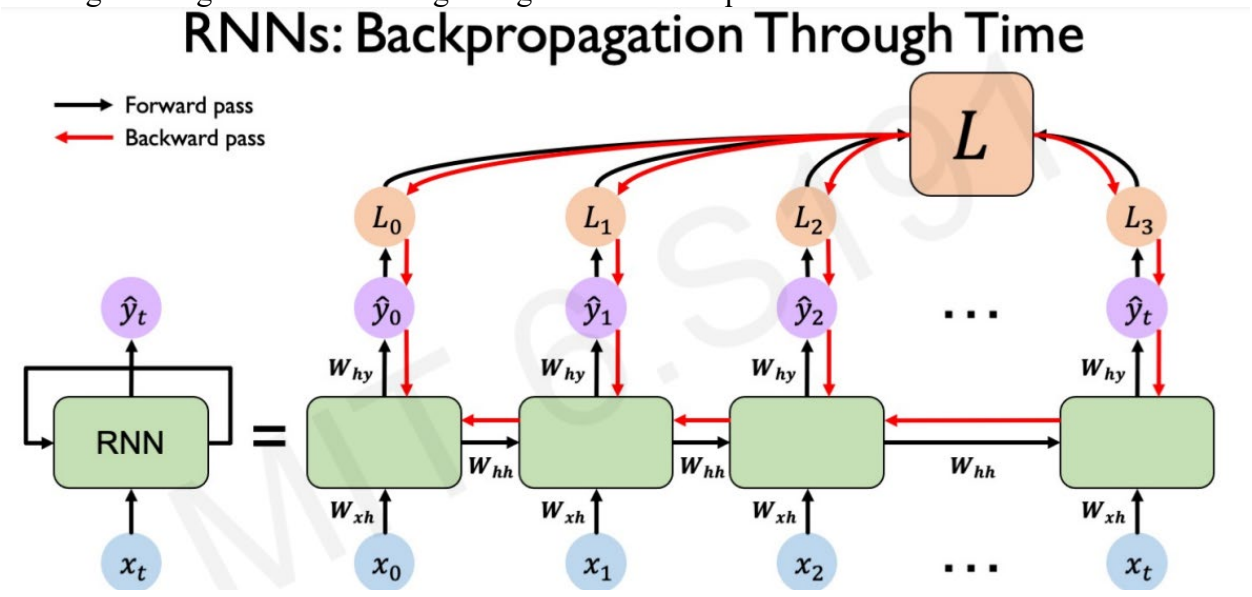
Backpropagation Through Time (BPTT)

How do we develop an algorithm to train RNN?

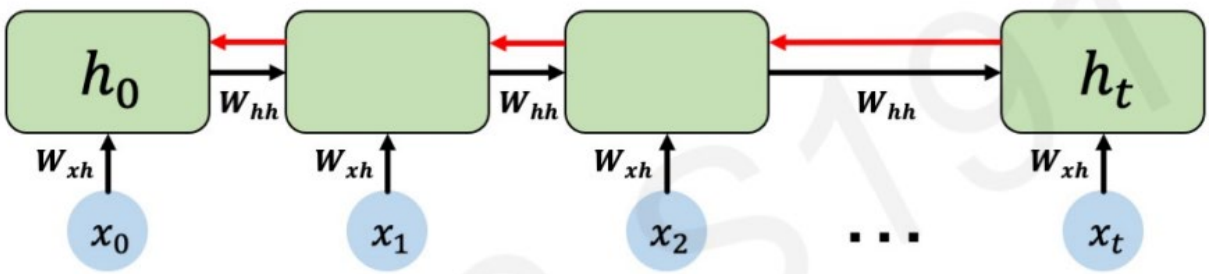
In the Feed forward models, first we make a forward pass i.e. generate an output. Then in order to train the model, we back propagate the gradients through the network, taking the derivative of loss with respect to each weight parameter and then adjust the parameter value to minimise the loss.

But in RNN, our forward pass also consists of going forward across time updating the cell state based on the input and the previous state generating an output Yt and loss Lt. then we can sum these individual losses to get out total loss L.

Thus , instead of back propagating errors through single feed forward network at a single time step, In RNNs, errors are back propagated at each individual time step and finally across all time steps all the way back from where we are currently to the beginning of the sequence. This is called BPTT as all errors are flowing backing in time to the beginning of our data sequence.



Standard RNN Gradient Flow



Computing gradient with respect to h0 involves many factors W_{hh} + repeated gradient computation! This can be problematic.

1. **Exploding Gradient Problem**

If values involved in this repeated matrix multiplication are >1 (too large) we have a problem called exploding gradient problem. Here, gradient values become extremely large and we can't optimise them.
Solution- gradient clipping ei scaling back large gradients.

2. **Vanishing Gradient Problem**

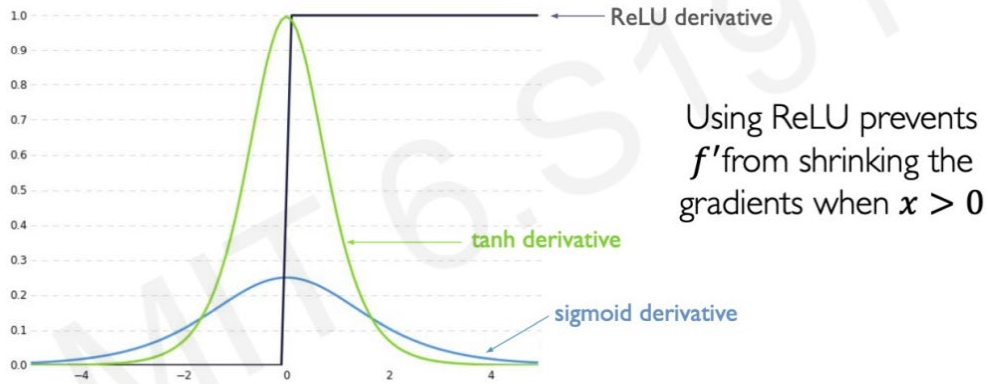
Here, the values < 1 (are too small). The gradients become increasingly smaller as we make these repeated multiplications and we can no longer train the network.
Solution - (1) choosing proper activation function, (2)initializing weights cleverly, (3) designing network architecture to handle it effectively.

Why Are Vanishing Gradients A Problem?

When we keep multiplying small numbers (0,1 range) together, they keep shrinking in size and eventually 'vanish'.
This makes it harder to propagate errors back into the past because the gradients are becoming smaller and smaller. Thus, we bias out network to capture more of short-term dependencies.
E.g. "the clouds are in the ___." Here, we can predict the next word easily with short term dependencies. But for a sentence like, "I grew up in France … and i speak fluent ___."; here we need to remember the context of 'France' to predict the language French. If there is short term dependency, we know the predicted word can be any language.
As the gap between important linguistic clues increase, the standard RNNs become increasingly unable to "connect to dots" i.e. link these relevant pieces of information together. This occurs due to the vanishing gradient problem.
Solution?

(1) Activation Function Choice

Using ReLU prevents the derivate f' from shrinking the gradients when $x > 0$. ReLU function has a gradient of 1 for x >0.



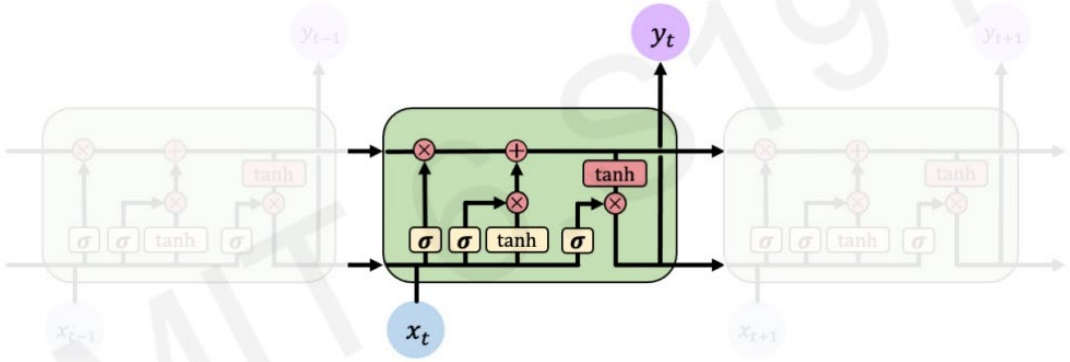
(2) Parameter Initialisation

Initialise weights to Identity matrix helps prevent the weights from shrinking to ZERO too rapidly during back propagation.

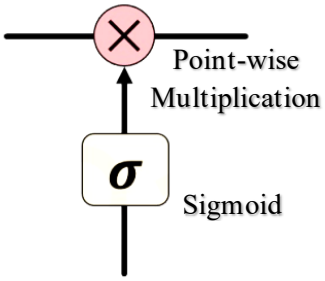
(3) Gated Cells

- Use a slightly more complex recurrent unit that can more effectively track long-term dependencies in the data by controlling what information is passed through and what is used in controlling the internal state.
- Many types of gated cell structure exist like LSTM, GRU …

Representation of Inner Workings of LSTM Cells

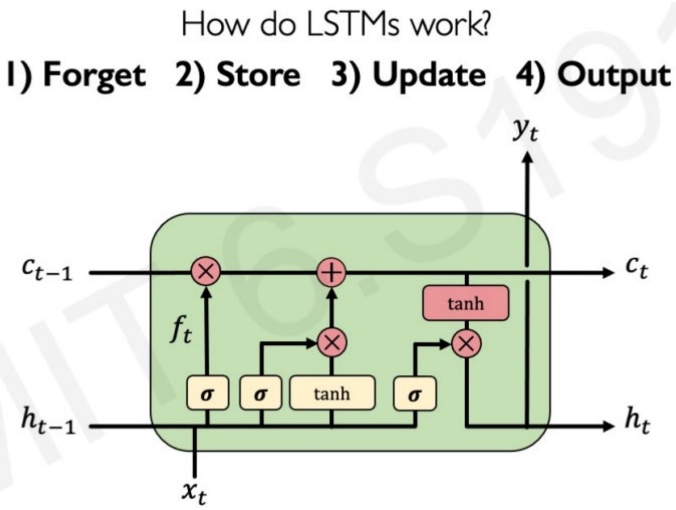


LSTM modules contain computational blocks that control information flow.
The key building block of LSTM is called Gate Cells.



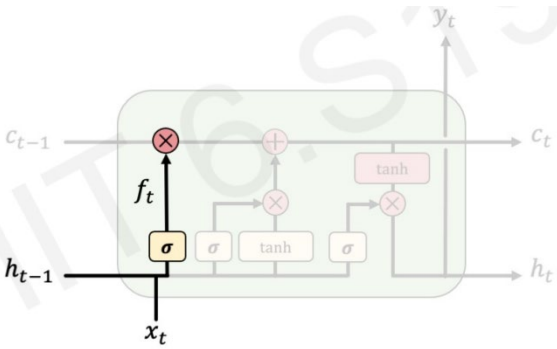
These are functions which selectively add/ remove information to its cell state and gates consist of a neural net layer like a sigmoid and a point-wise multiplication (red).
Sigmoid translated the input over '0 - 1 range'. This translation can be thought of as "how much information passing through the gate should be retained?". It's between nothing (zero) and everything (one). Thus we're sort of gating the flow of information.

How do LSTMs work?



4 simple steps –

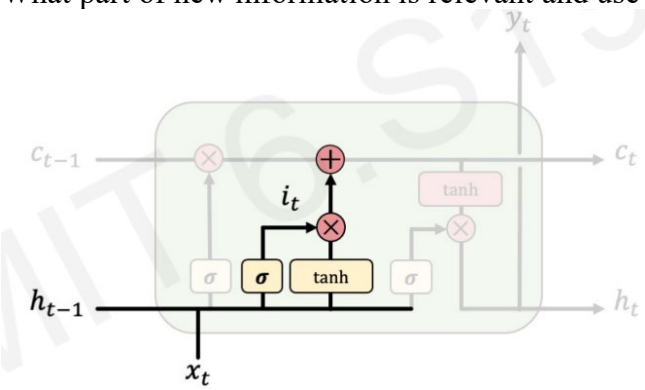
1. Forget – irrelevant history



Here we decide what part of information is thrown away.

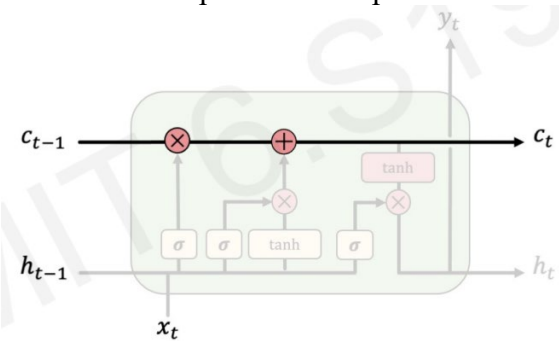
2. Store – perform computation to store relevant parts of new information

What part of new information is relevant and use this to store information into cell state.



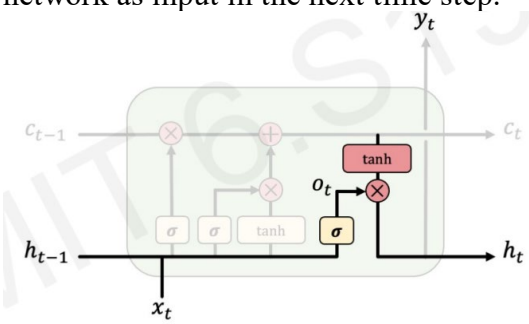
3. Update – use above two steps to update the internal state

Takes relevant parts of both prior and current information and uses this to selectively update its cell state.

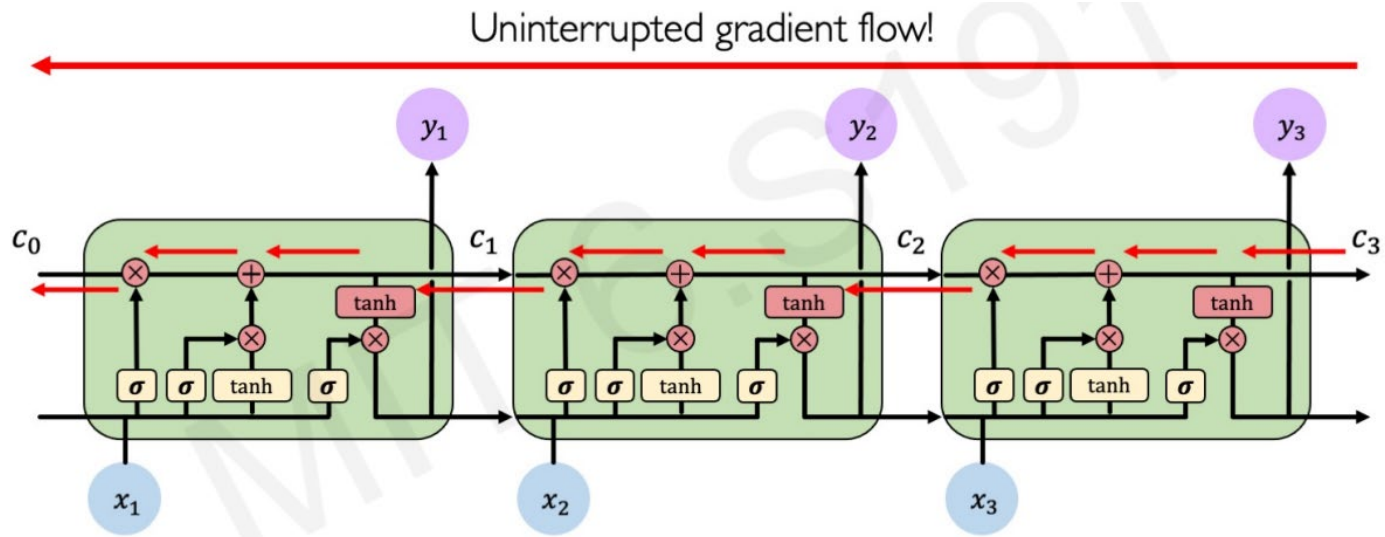


4. Output generation

This is known as ***output gate*** which controls what information encoded in the cell state is sent to the network as input in the next time step.



How does it help us train the network?



An important property of LSTM is that all of these different gating and update mechanisms actually work to create this internal cell state 'C' which allows for the uninterrupted flow of gradients through time.

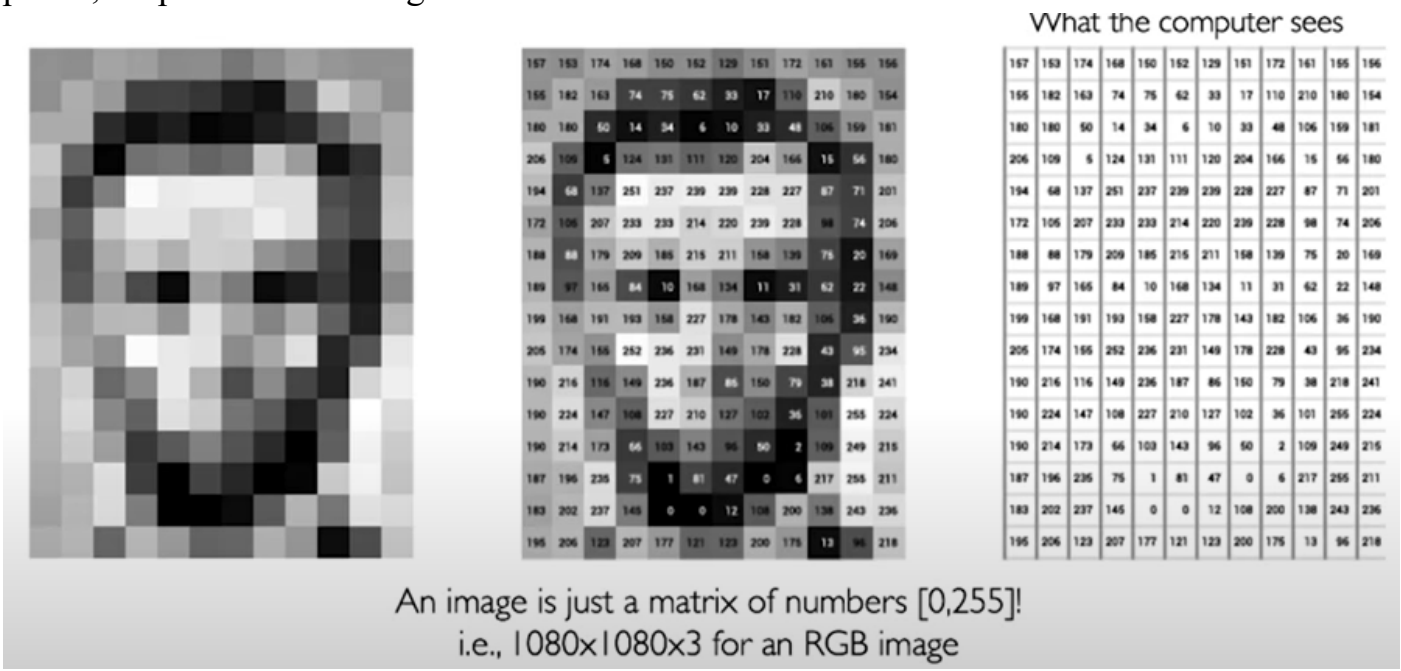
It is like a highway of cell state where gradients can flow uninterrupted shown here in red. This enables us to alleviate and mitigate the vanishing gradient problem.

LSTMs Key concepts

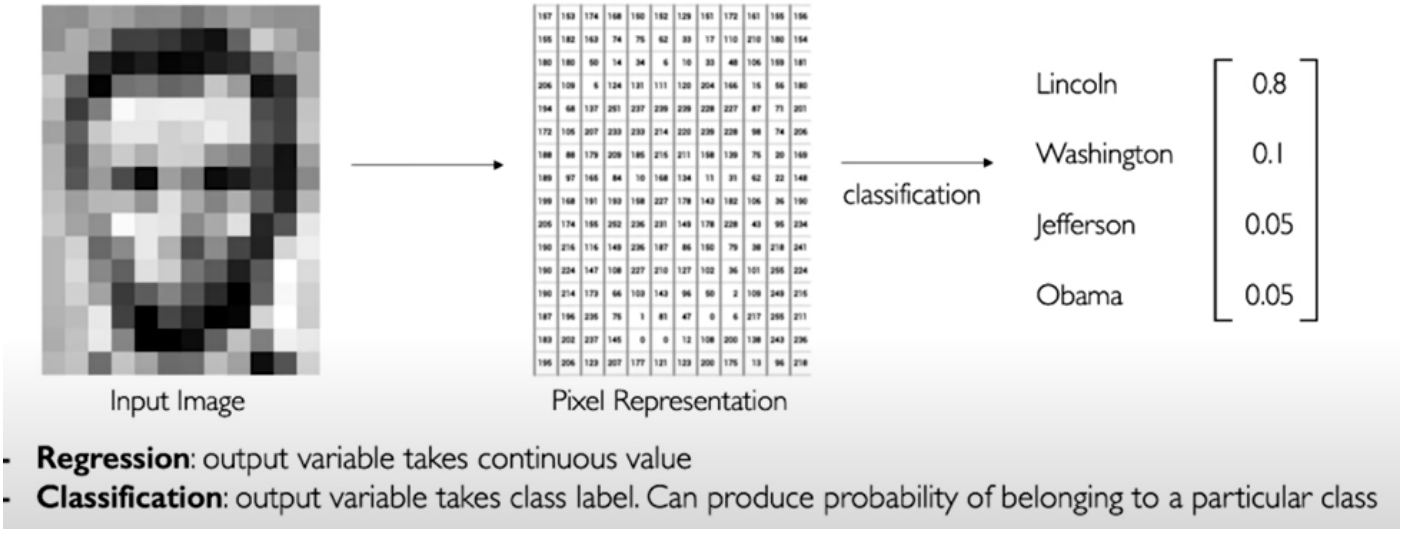
- They are able to maintain this separate cell state independent of what is outputted
- They use gates to control the flow of information
    - Forgetting irrelevant history
    - Storing relevant new information
    - Selectively updating cell state
    - Output gate returns a filtered version of the cell state
- Maintaining this separate cell state allows for the efficient training of the neural network via backpropagation through time.

Convolution Neural Network

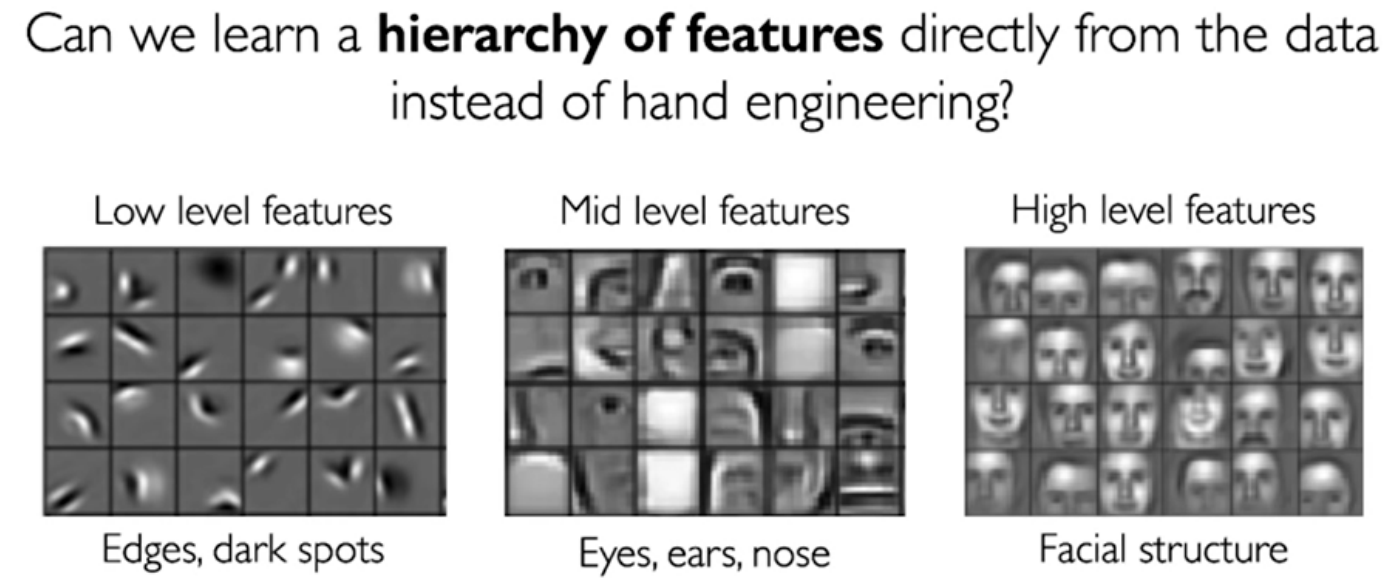
To a computer, images are just a list of 2-dimensional numbers. Every image is made up of pixels, all pixels are nothing but numbers.



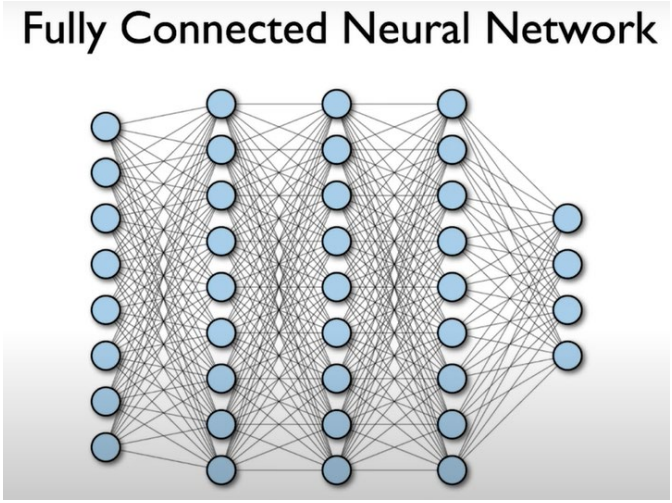
We can represent the image as a 2-dimensional matrix of these numbers. One number for each pixel in the image and this is how a computer sees an image.
For a RGB image, we can represent it by a 3-dimensional array. Now we have 3 2-dimensional arrays stacked on top of each other corresponding to red, green and blue respectively.



The image can be dealt either as a Regression output or a classification output. For regression., we need output in the form of probability of the image being a certain US President (for e.g.).
For this we need to be able to tell what is actually unique about a picture of Lincoln versus pictures of other presidents like Washington or Jefferson or Obama. This difference in pictures is with respect to features/ characteristics of that particular class.
We can manually define different characteristics of the categories but there can be deformation, occlusion (background light), illumination conditions, viewpoint variation, background clutter and intra class variation.
We need our algorithm to be able to be invariant to all of these types of variation and yet be sensitive enough to pick out different intra-class variation i.e. being able to distinguish a feature that is unique to this class in comparison to features present within the class.
So we want to be able to extract features and detect their presence in images automatically in a ***hierarchical fashion***.

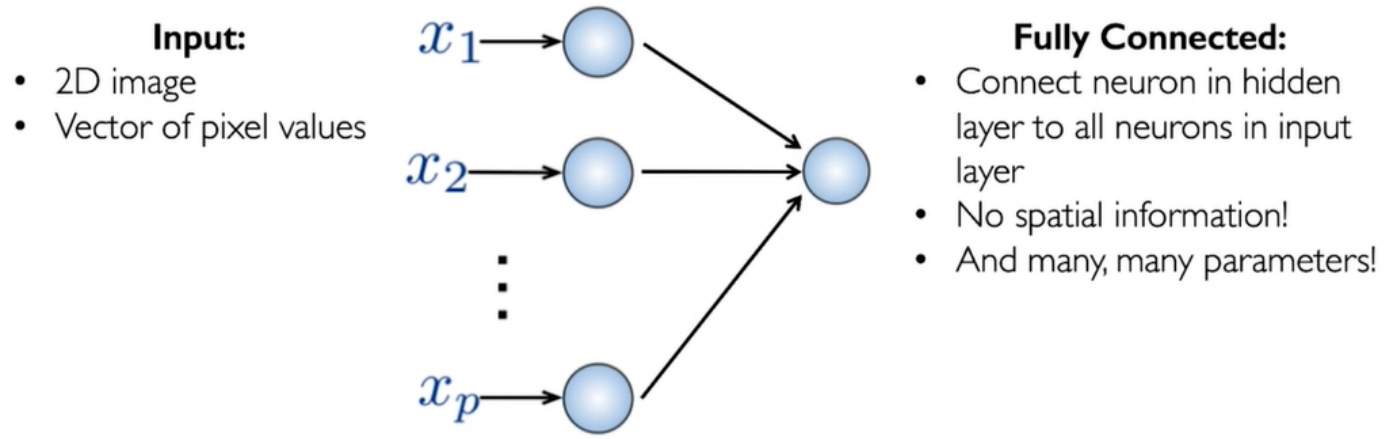


Learning Visual Features
To learn hierarchical visual features, we need a different structure from feed-forward dense layers and recurrent layers for handling sequential data.

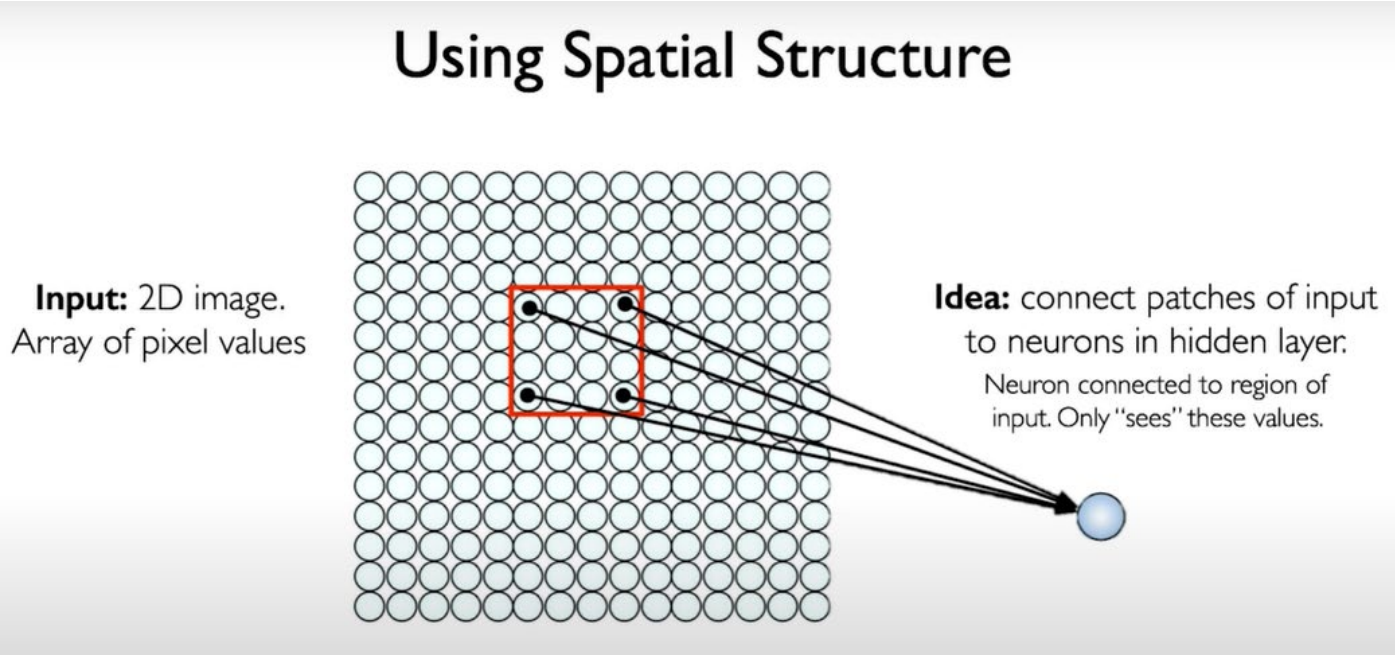


To summarise what we've covered, this is a fully connected neural network which is also called dense neural networks where there are

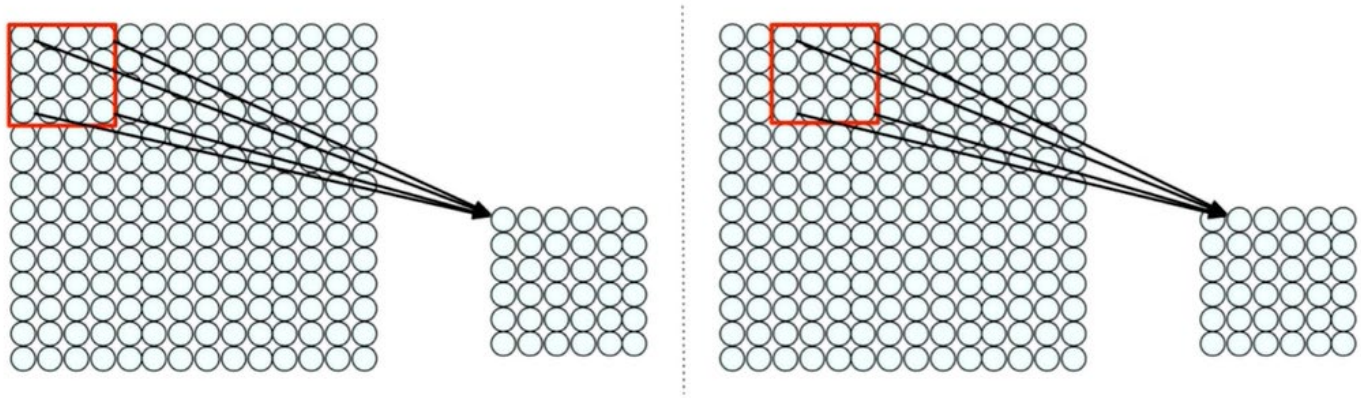
multiple hidden layers stacked on top of one another. Each neuron in each hidden layer is connected to every other neuron in the previous layer.



So for image classification problem, we have a 2D image which can be read in as a 1 dimensional vector of pixel values which is the input for the network.
Now because we have flatted the 2D image into 1 dimension, we has lost any and all spatial information. Thus the network has to learn the spatial information for e.g. that one pixel is closer to its neighbouring pixel. This important information is lost in the fully connected network.
So, how can we use spatial structure in the input to inform the architecture of the network? For this we will retain the image in 2 dimensional array and not collapse it into 1 dimension.



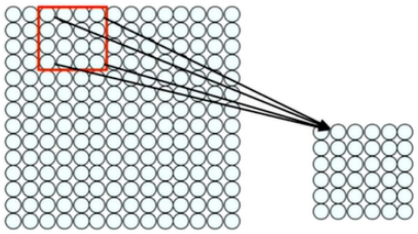
One way we can use spatial information, is to connect patches of the input (not the whole) to the neurons in the hidden layer. So each output neuron only sees the input coming from a particular patch which precedes it. This not only reduces the weights in the model, but also allows us to leverage the fact that in an image spatially close pixels are likely to be somewhat related and correlated to each other.



Connect patch in input layer to a single neuron in subsequent layer.
Use a sliding window to define connections.
How can we **weight** *the patch to detect particular features?*

Now to define connections across the whole network, we can simply slide the patch across the input image. *Each time we slide it, we have a new output neuron in the subsequent layer*. This way we can take into account the spatial structure.
To learn the visual features, we weight the connections between the patches and the neurons so that we can detect particular features. By doing so, we enable the patch to detect particular features. So, now the question arises about weighting the patch?

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This "patchy" operation is **convolution**

1) Apply a set of weights – a filter – to extract **local features**

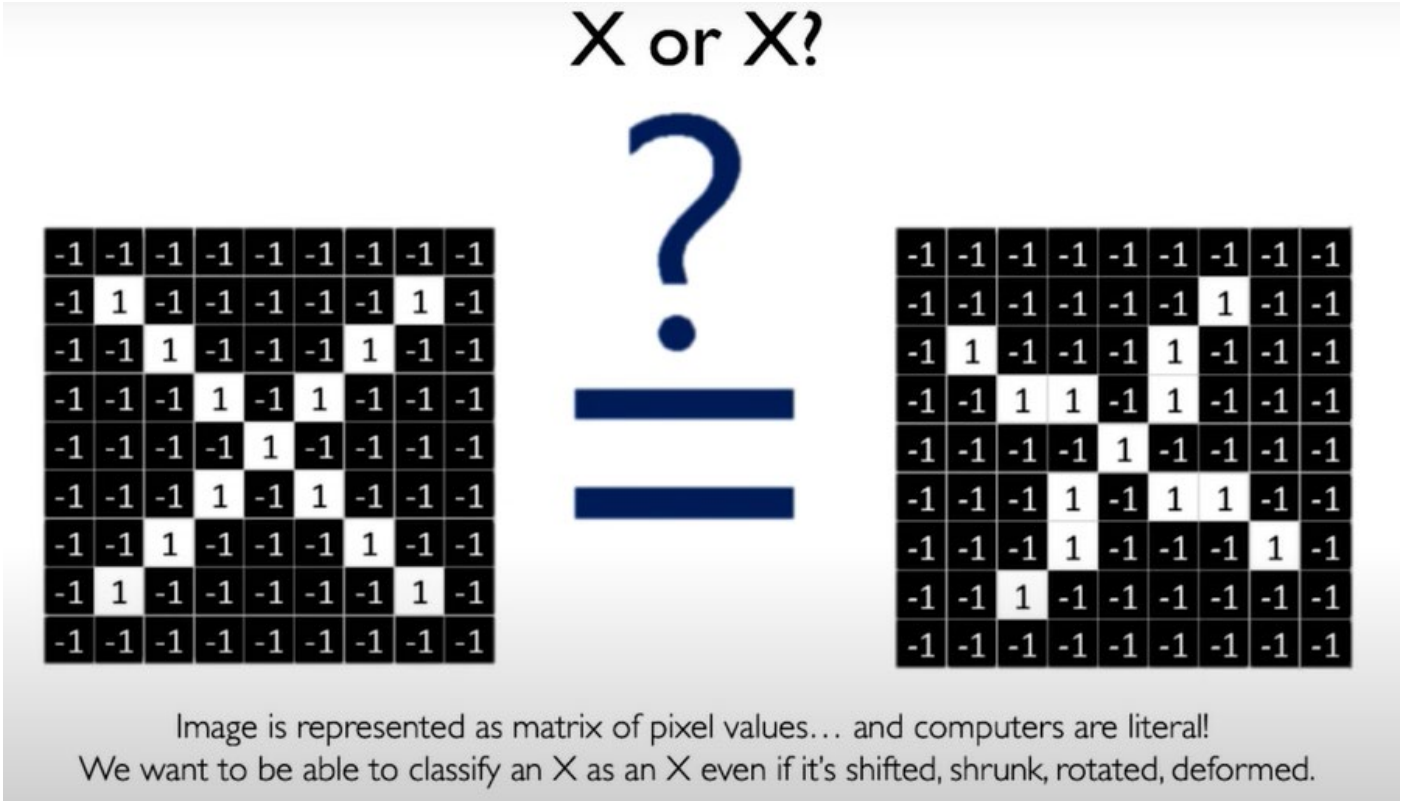
2) Use **multiple filters** to extract different features

3) **Spatially share** parameters of each filter

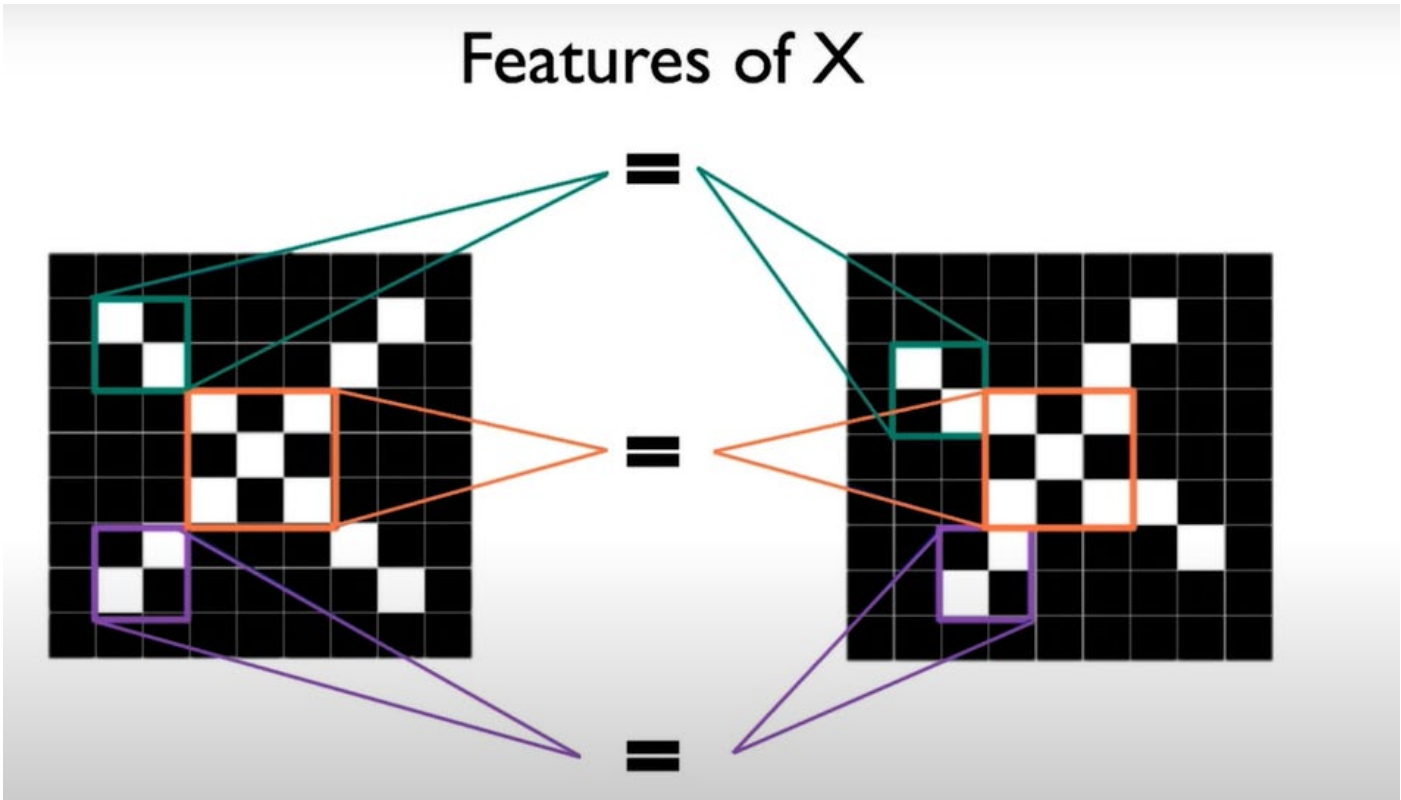
In practise, there is an operation called Convolution.

- The image can be thought of to be divided into overlapping 4x4 patch and we can imagine a filter (red square) consisting of 16 weights (because 4 by 4).
- This 4 by 4 filter is applied on the entire image. As a result, the information we have (about the pixels of the various patches) will be used to define the "internal state" of the output neuron in the next layer. So this 4-by-4 filter had learnt certain weights.
- We then shift the patch by say 2 pixels to grab the next patch and compute the next output neuron.

How does the Convolution filter help us extract features?

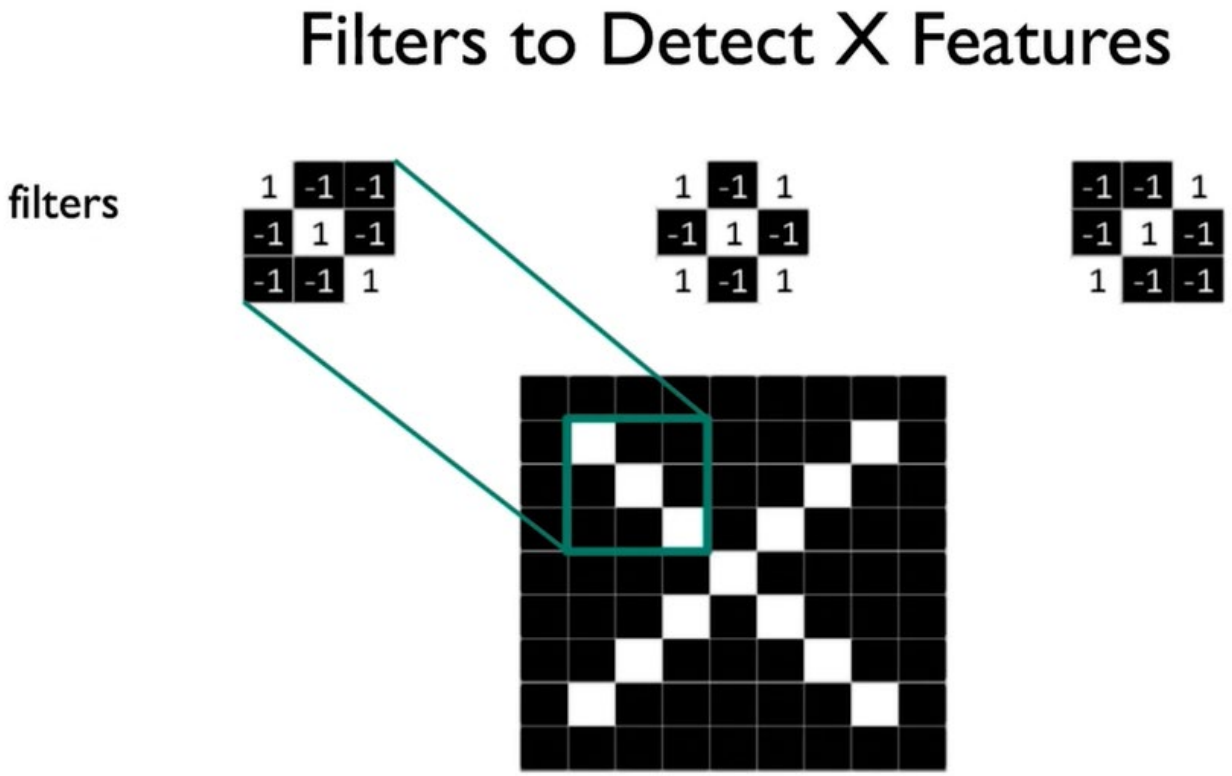


- Suppose we want to classify the letter ‘X’ in a set of black-and-white images of letters.
- Where, black = -1 and white = 1.
- Now we cannot simply compare the two as matrices of -1 and 1 because it will not allow for slight deformations/ shifts/ tilts/ enlargement/ rotation and so on. We need something more robust.
- We want the feature which define what ‘X’ looks like.

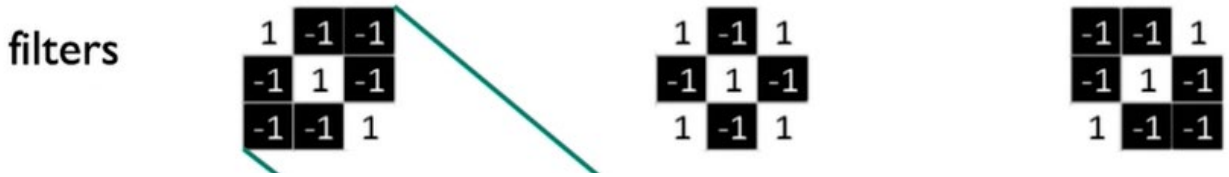


- We want the model to compare pieces of ‘X’ instead of the image as a whole. The important pieces of ‘X’ is what we call ‘features’.

- If the model can find these defining features which define X, in roughly same positions then it can better understand the similarities between different pictures of letter X even in the presence of deformities.

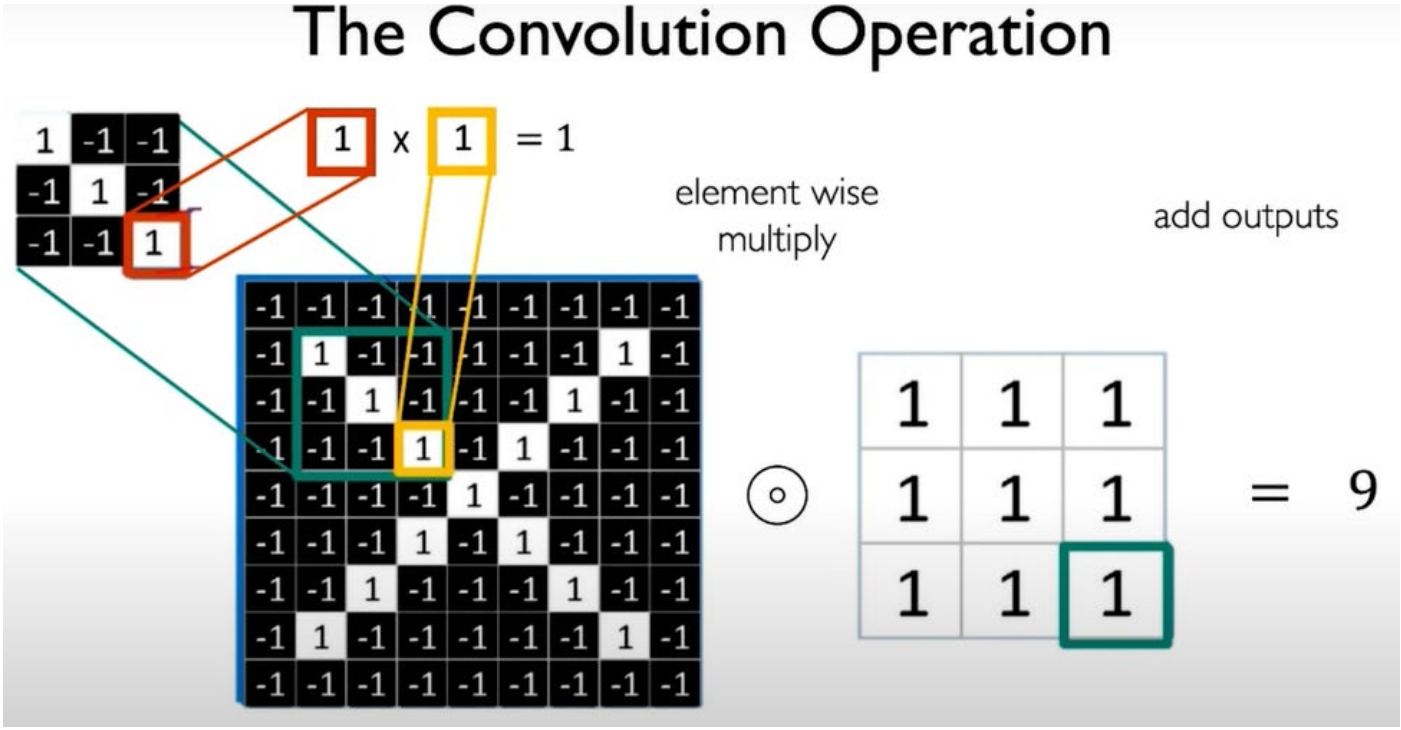


- We can imagine each feature to be a mini-image/patch. At the same time, we remember that it is a small 2-dimensional array of values.
- So we can have filters to detect these features for e.g. edge of diagonals and center where lines cross.
- So with the help of the filters, we can detect different variations of 'X'
- The top of row filters



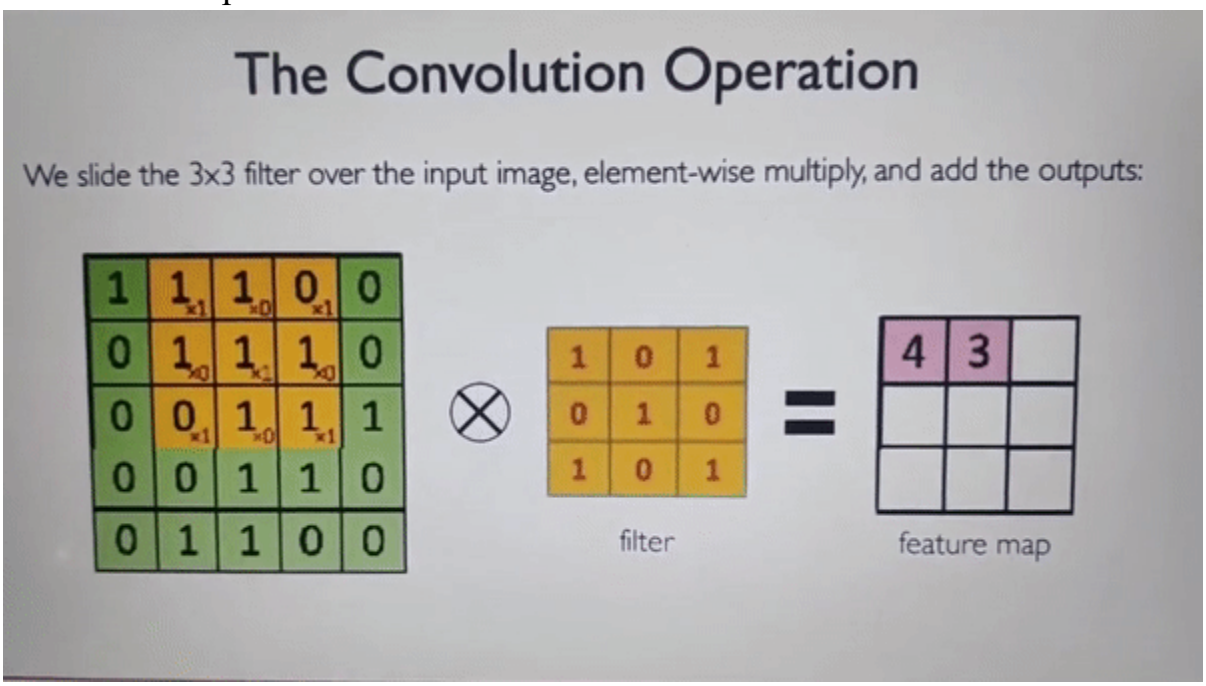
are nothing but matrices of weights. The task of weights is to help us understand and detect the corresponding features of the input image.

Given the filters, how can a convolution detect features? (where the feature is occurring in the image)



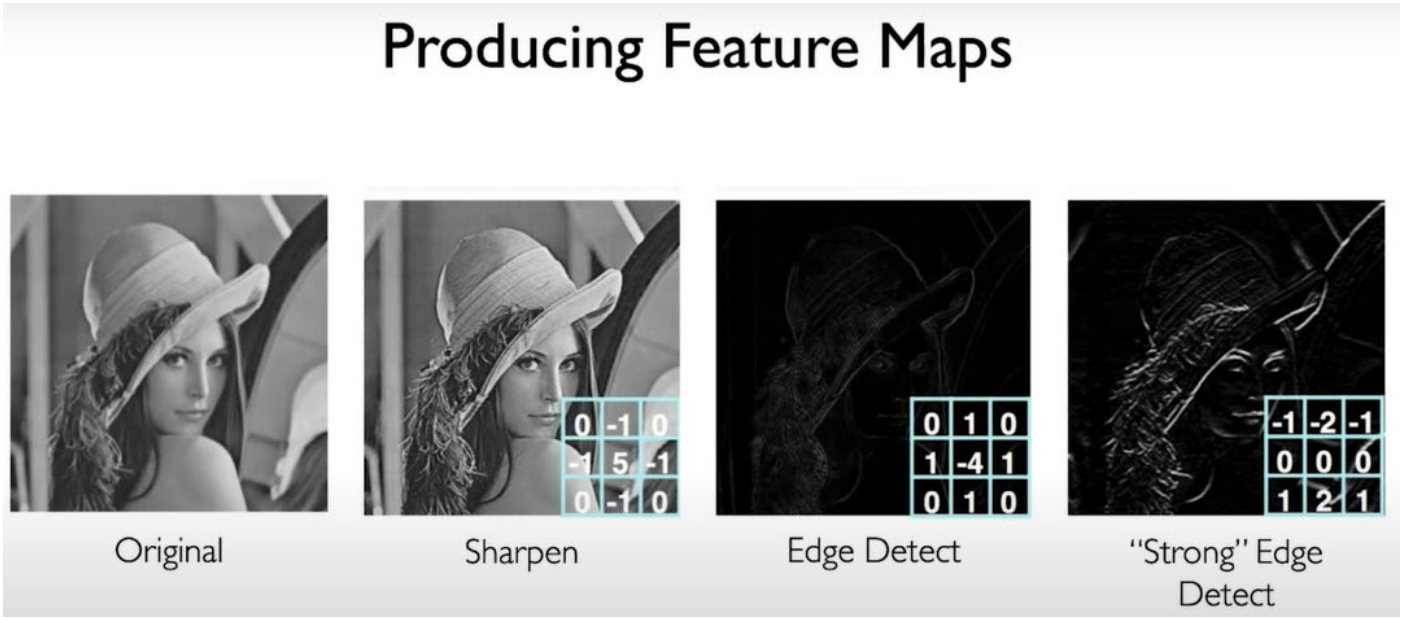
- The idea of convolution is to preserve spatial relationship between pixels by learning image features in small little patches of image data
- To do this, we need to perform an element wise multiplication between the filter matrix (learned weights) and the patch of input image. (obvious the dimension of matrices are the same).
- Result of element-wise multiplication was $\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$. Adding all elements we get 9.

Another Example

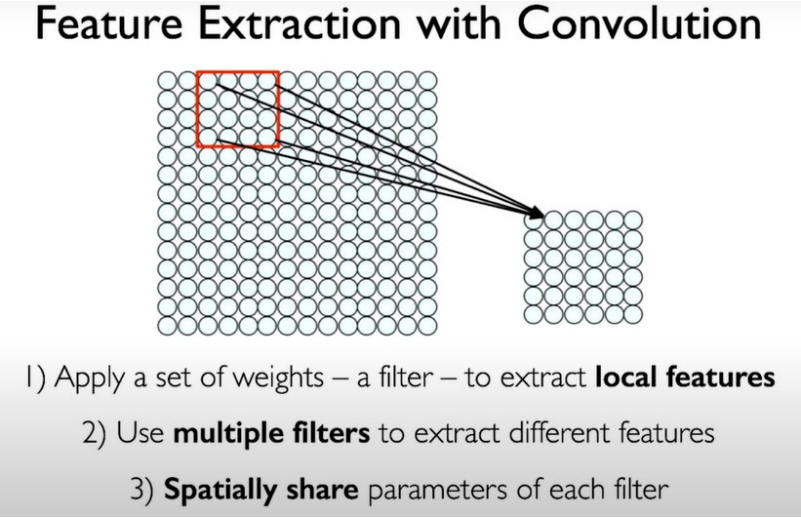


- Suppose we have a 5-5 image alongside a 3-3 filter
- We start by placing the filter on top left corner and the multiply the filter matrix and input image matrix and add the entries to get a result of 4.

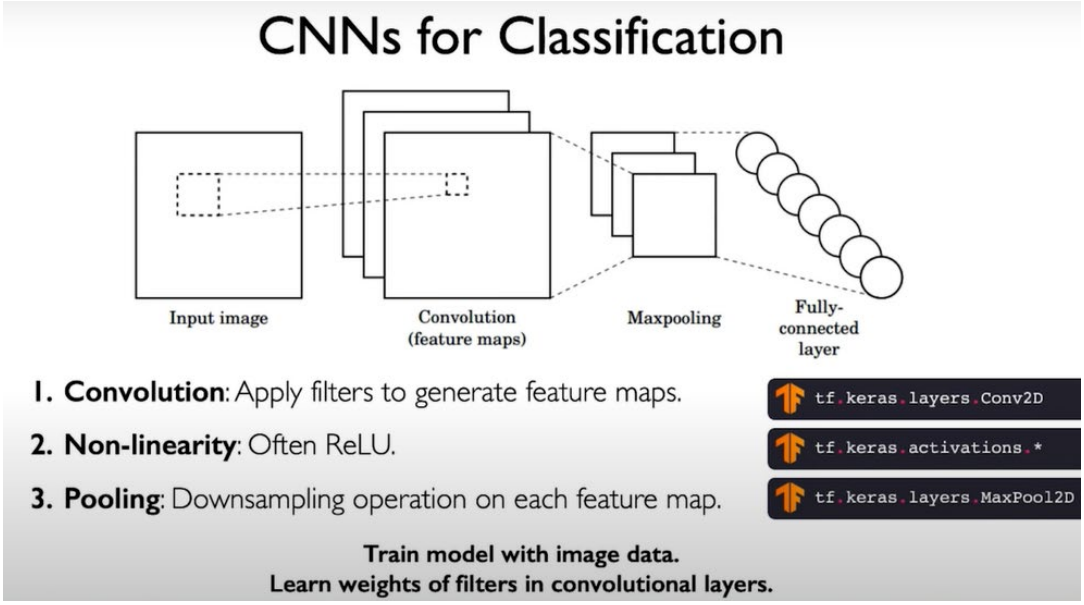
- We can then slide the filter and compute the entry 3.
- This continues until we reach the end. Here the resulting matrix $\begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$ is called Feature Map (because it tells us where in the image the features occur) or Activation.
- Wherever "the pattern inside the filter" occurs in the input image, the value in the Feature Map will be the highest. This is the place where we can say that we have "maximum activation".
- So now we know "where the features occur" OR "where did the filter activate most, on the image".



- Let's consider the image of this women.
- Bottom right corner we have 3 different filters (different because they have different weights)
- By simply changing the weights in the filters, we can learn to detect very different features in that image.
- We can learn to sharpen the image by applying this specific type of filter called sharpening filter
- We can learn to detect edges OR learn to detect very strong edges.
- Here, the filters shown are "not learnt" but "constructed" filters (hand-engineered).
- So instead of learning some specific features, the networks learns every feature necessary to identify the image. ("necessary" is decided by weights).



Convolutional Neural Network (CNNs)



- Let's consider a simple CNN designed to learn features directly from the image data. Then we use these learned features to map these onto a classification task for these images.
- There are 3 core components to CNN
    - Convolution operation – as seen earlier, it helps to generate Feature Maps (to detect where features are in our image).
    - Applying a non-linearity = ReLU (because the features are highly non-linear).
    - Pooling operation – another word of down-sampling operation on each feature map – allows us to scale down the size of each feature map
- The output of the 3 components are then sent to a dense (fully) neural network. As a part of training, the network finds the optimal value of the weights used in the convolutional layers and the fully-connected layers.

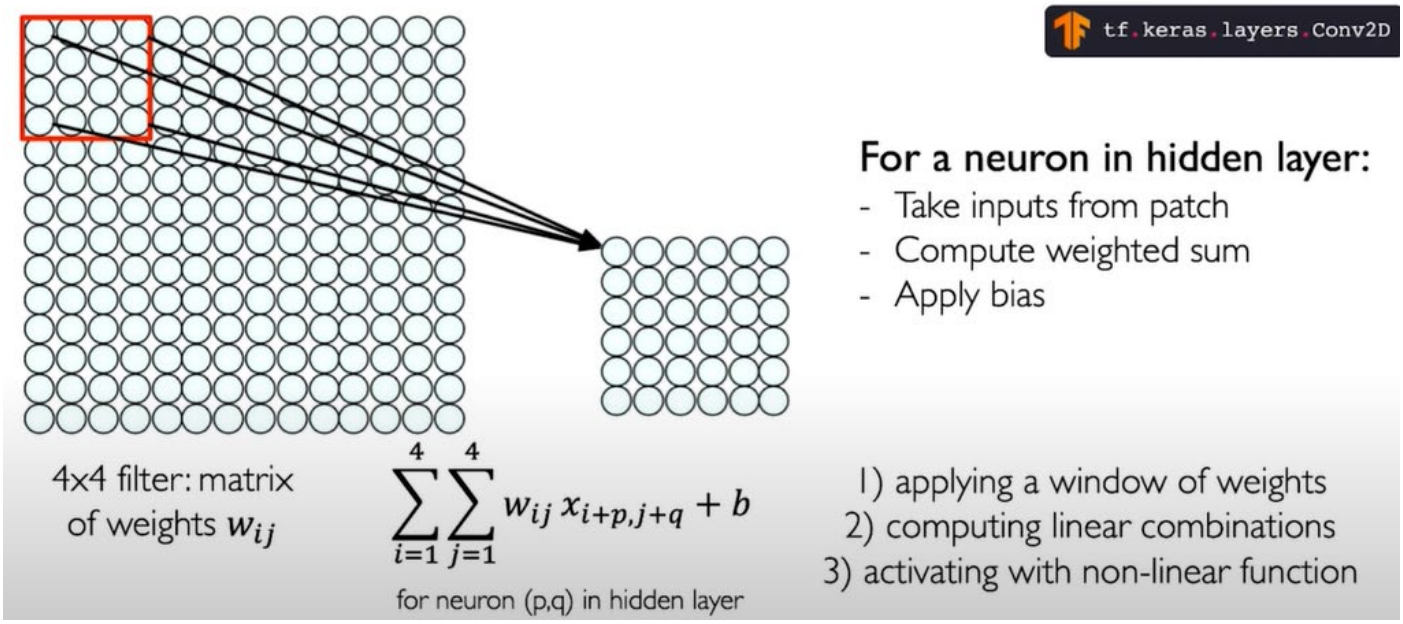
Let's look at the components.

1. Convolutional Layer

Each neuron in the hidden layer will compute a weighted sum of each of its inputs like we saw in the dense layers we'll also need to add on a bias to allow us to shift the activation

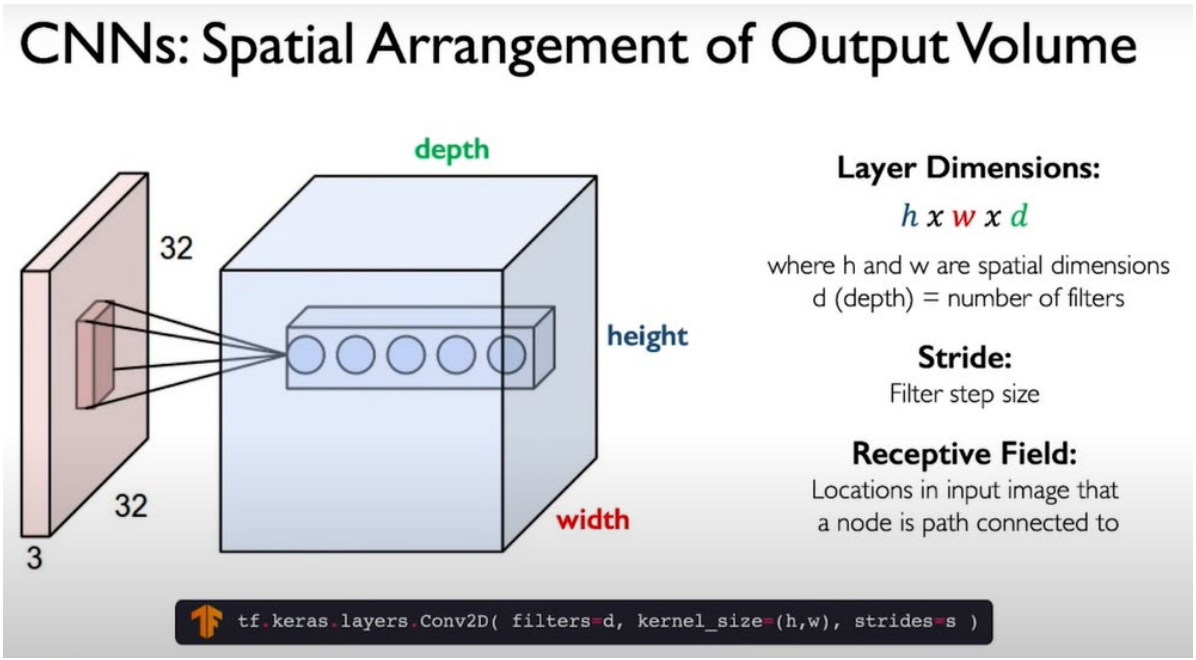
function and apply and activate it with some non-linearity so that we can handle non-linear data relationships.

As we know now, each neuron only sees a very specific patch (locally connected patch) of the entire input image. We take a weighted sum of those patches, apply a bias and activate it with a non-linear activation function. At the end of all of this, we are left with a feature map.



- Here we have a 4x4 filter matrix.
- As we known, each neuron in the hidden layer receives input only from a patch of input image.
- We multiply weight 'Wij' with input 'x' and then add like we did above to get 4, then we add a bias 'b' and activating the non-linearity.

How can we have multiple filters – earlier we saw how we can apply one filter and have one feature map – in reality there are a lot of features to learn. How can we do multiple-feature extraction?

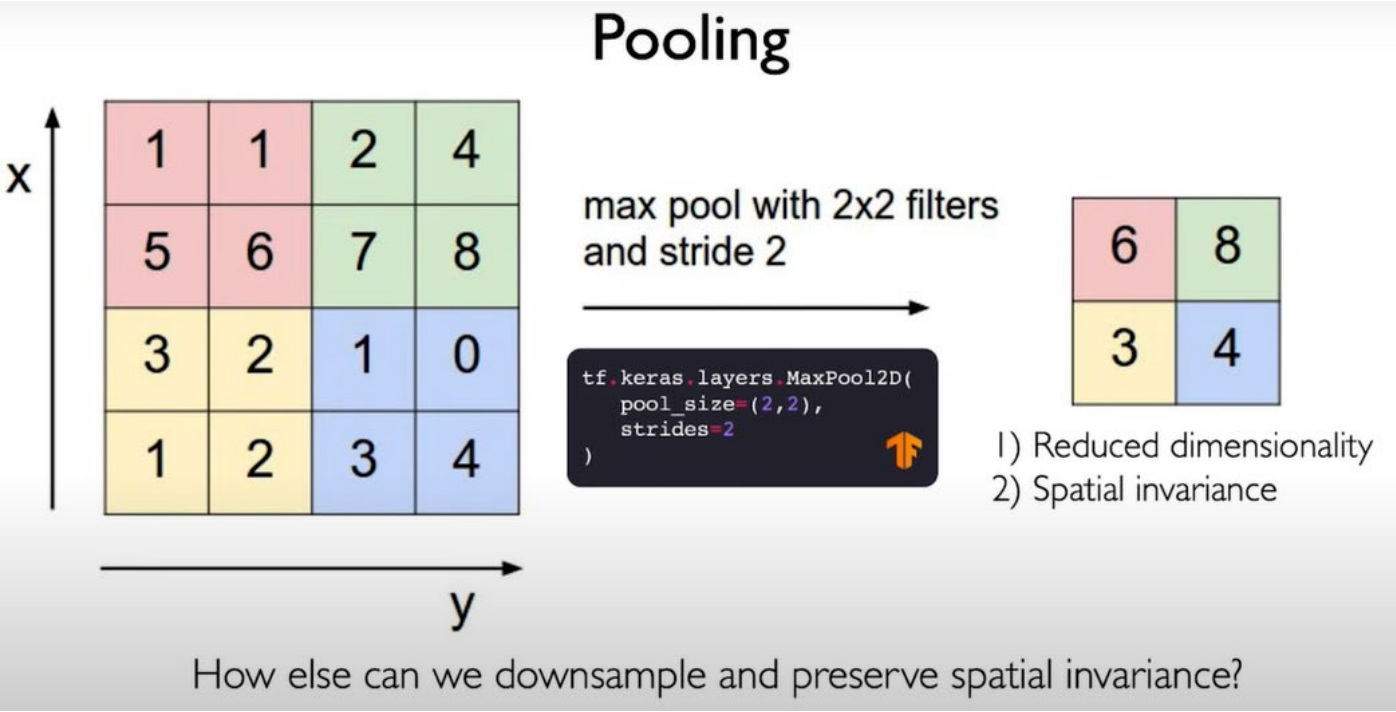


- The output layer, still a convolution, now has a volume (height*width*depth). Where depth = number of filters
- Where, height and width are spatial dimensions based on the input image and filter applied. This also depends on how many pixels we shift the filter.
- But we also need to think about the connections of the neurons in these layers in terms of their "receptive field" which defines the spatial arrangement of how neurons are connected in the convolutional layers and how those connections are defined.
- So the output of a convolution layer in this case will have this volume dimension. So instead of having this one filter one feature map, now we're going to have a volume of filters.

2. Introducing Non-Linearity

- After each convolution operation, we need to apply this non-linear activation function to the output volume of that layer.
- We do this because image data is highly non-linear.
- Common activation function – ReLU i.e. Rectified Linear Unit. This is a pixel-wise operation that replaces all negative values with zero and keeps all positive values as is.
- We can think of this as a thresholding operation, anything less than zero becomes zero/ gets thresholded to zero.
- Negative values detect negative detection of a convolution but this non-linearity clamps that.

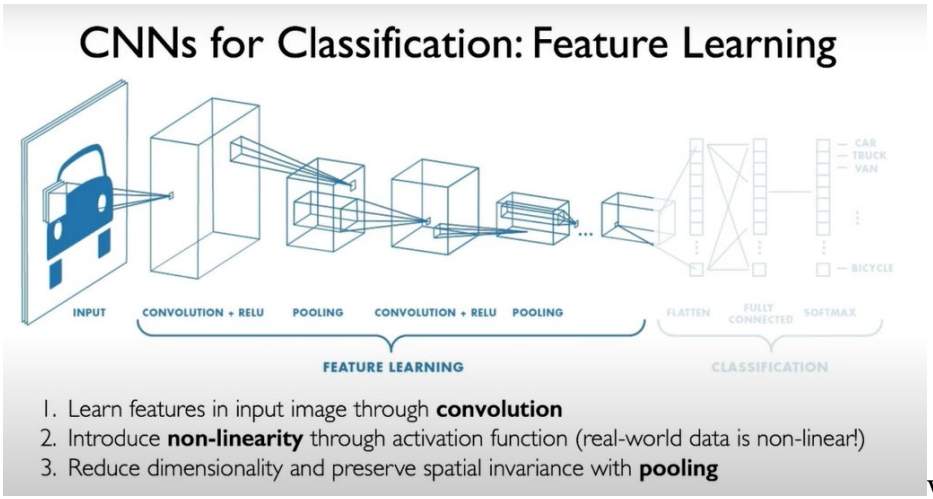
3. Pooling



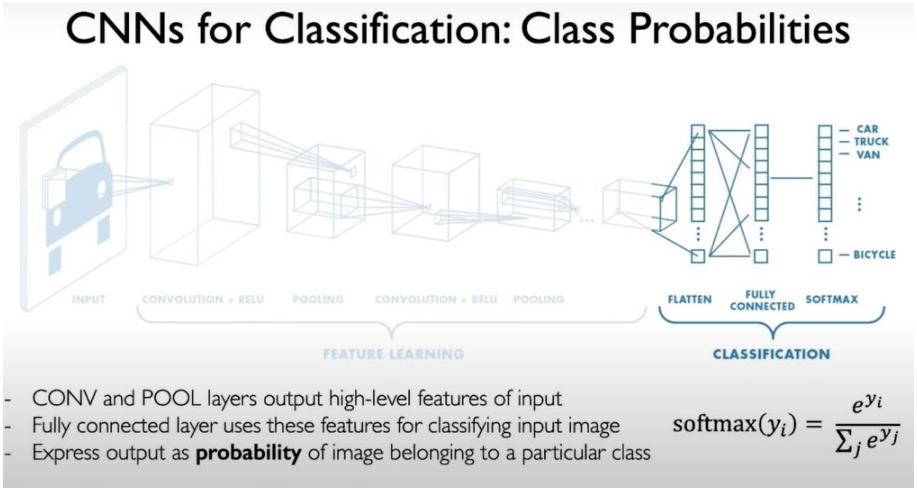
- Pooling is an operation that is used to reduce the dimensionality of our inputs and our feature maps while still preserving spatial invariants.
- Common type of pooling = Max Pooling.
- Here, we simply take the maximum over these 2-by-2 filters in our patches and sliding the patch over our input image. This is very similar to convolutions but instead of element-wise multiplication and addition, we take the maximum.

- Another way = Mean pooling – take average over the patch. It is a very smooth way of performing the operation (smooth because it is less subject to outliers).

So the CNN operation can be broken down into 2 parts – (1) Feature Learning and (2) Class probabilities.



we have seen this play out above.

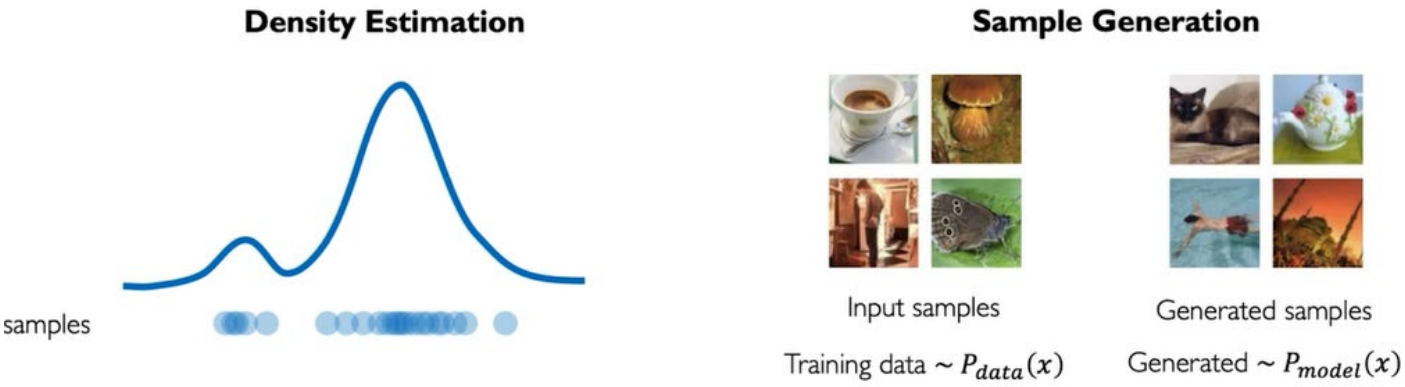


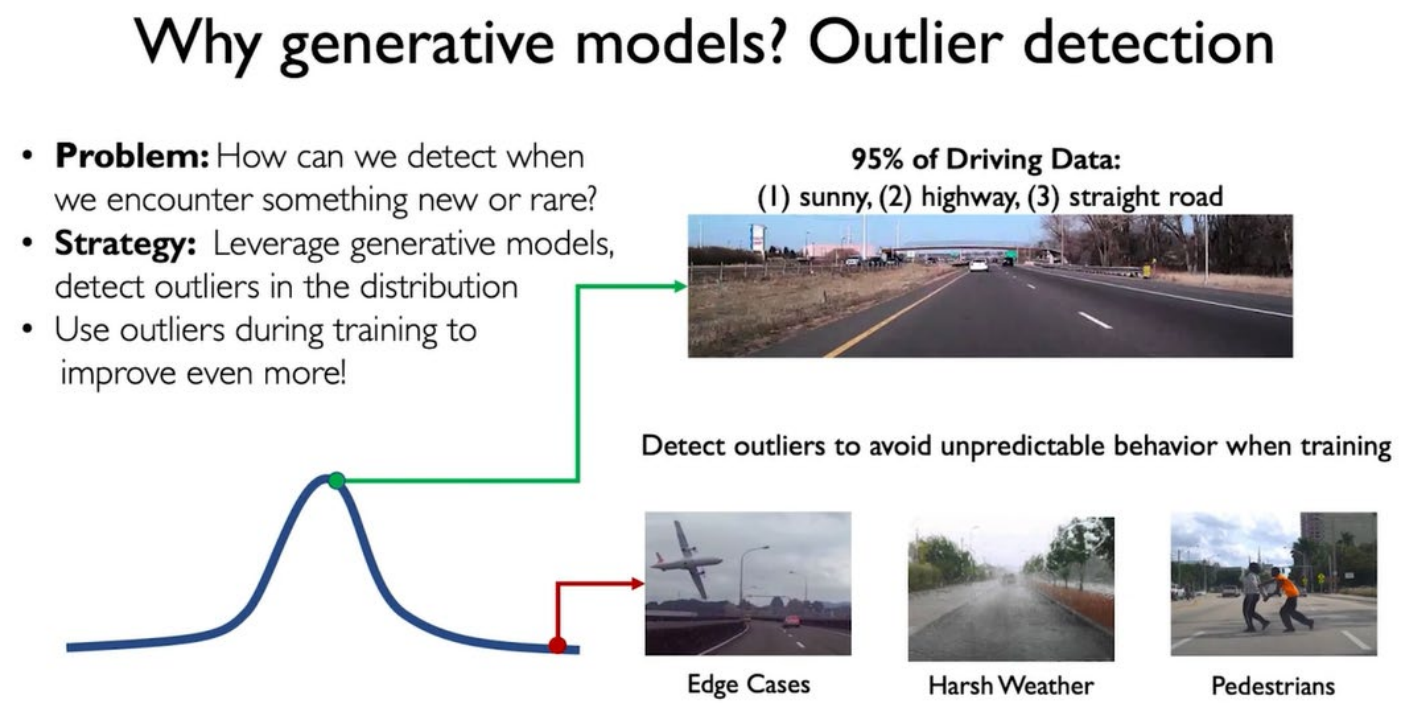
now with the outputs of the first part, our task is to perform classification/ calculation of class probabilities present in the input image. This classification is done by a dense neural network layer. We can output class probabilities using a function called 'softmax' whose output actually represents a categorical probability distribution.

Deep Generative Modelling

Generative Modelling is one example of unsupervised learning. The goal is to take as input training samples from some distribution and learn a model that represents that distribution.

This can be achieved in 2 principle ways – (1) Density Estimation and (2) Sample Generation

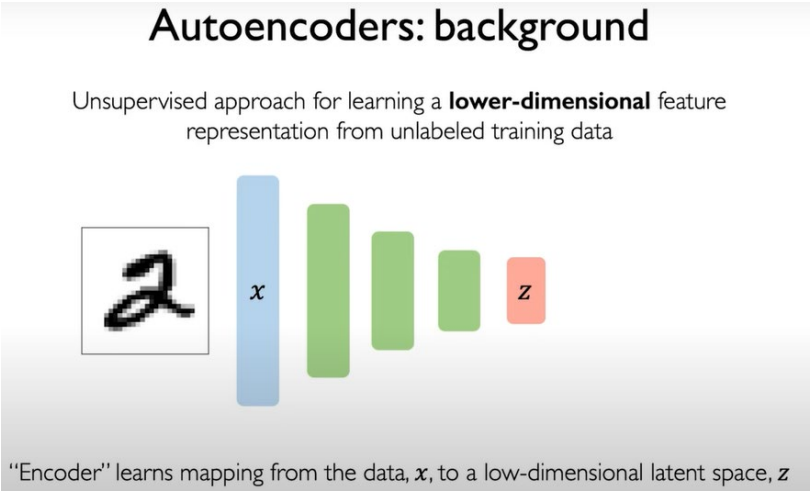




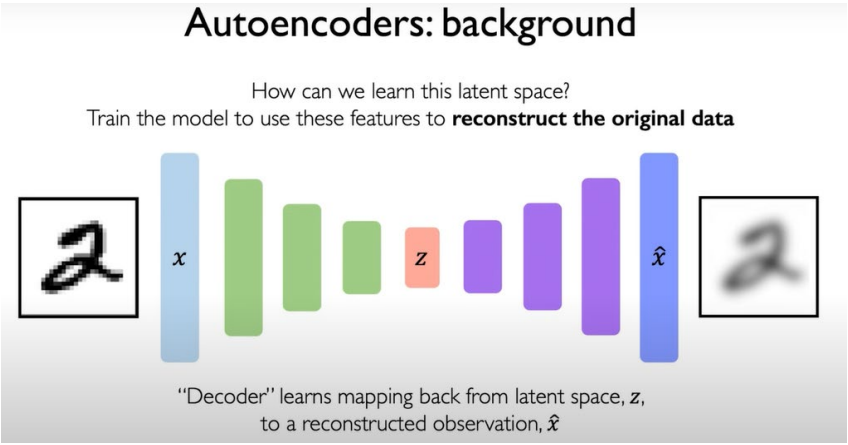
Latent Variable Models → True explanatory factors that are creating the observable variable are called Latent Variables (latent because they cannot be observed only felt). The aim of generative modelling is to learn these underlying latent variables even when we are only given the observations that are seen.

Autoencoders

Simple and foundation generative model which tries to build up these latent variables representation by self-encoding the input.



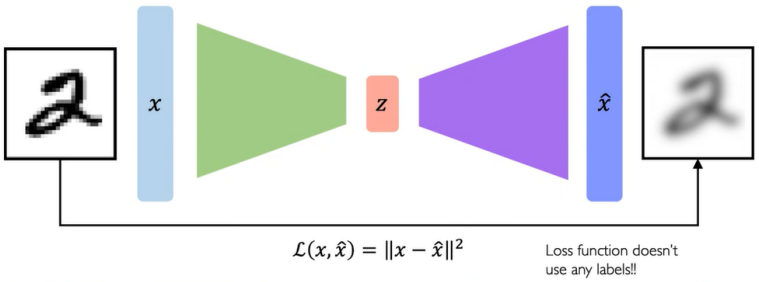
- It is an approach to learn lower-dimensional latent space from raw-data.
- Image of 2 is the raw-data inputted into a series of deep neural networks and the output generated is a low-dimensional latent space 'z'
- We called it an encoder because it is mapping raw data 'x' into an encoded vector of latent variables 'z'.
- Why is important to ensure low-dimensionality of z?
  Low dimension = compress data so that we can learn compact and rich features.



- Now because this is an unsupervised algorithm, there are no labels for 'z'. So to train such a model we can build up a decoder network which will reconstruct the original image starting from this lower dimensional latent space.
- Again, this decoder portion is a series of neural layers like convolutional layers.
- The output is called x-hat because it is our prediction – an imperfect reconstruction of input 'x'
- We can compute the difference between the input x and output x-hat and work to minimize the difference between them. $L(x, \hat{x}) = \left||x - \hat{x}|\right|^2$. We can take the mean squared error (for images, it is the pixel-wise difference)

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**

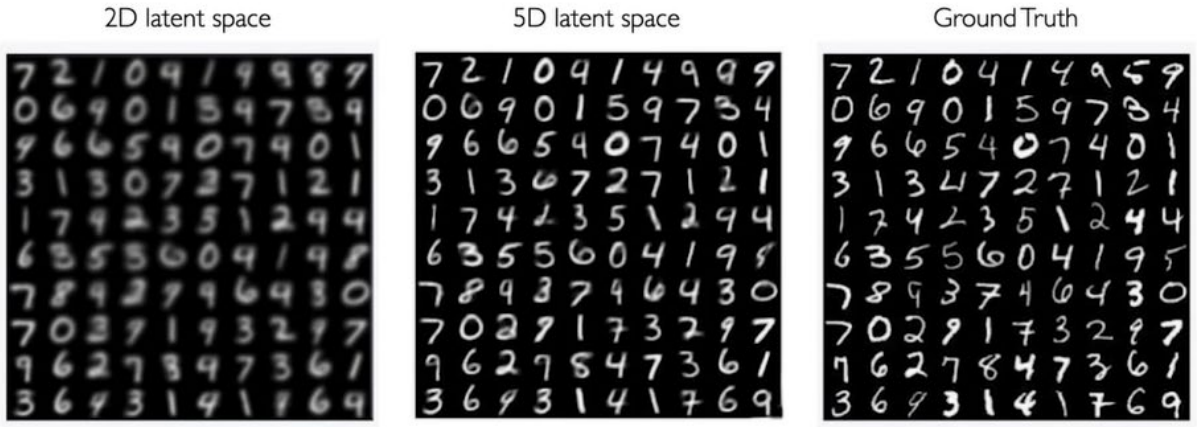


simplify the diagram

- Note: the loss function is not using any labels – it is simply using the raw data to supervise itself on the output.

Dimensionality of latent space → reconstruction quality

Autoencoding is a form of compression!
Smaller latent space will force a larger training bottleneck



- When we constraint this latent space to a lower-dimensionality that affects the degree to which we can actually reconstruct the input. We can imagine that we are bottlenecking (narrow tunnel end of bottle) the information .In practise, the lower the dimensionality of the latent space, the poorer/ worse quality reconstruction we are going to get.

Autoencoders for representation learning

Bottleneck hidden layer forces network to learn a compressed latent representation

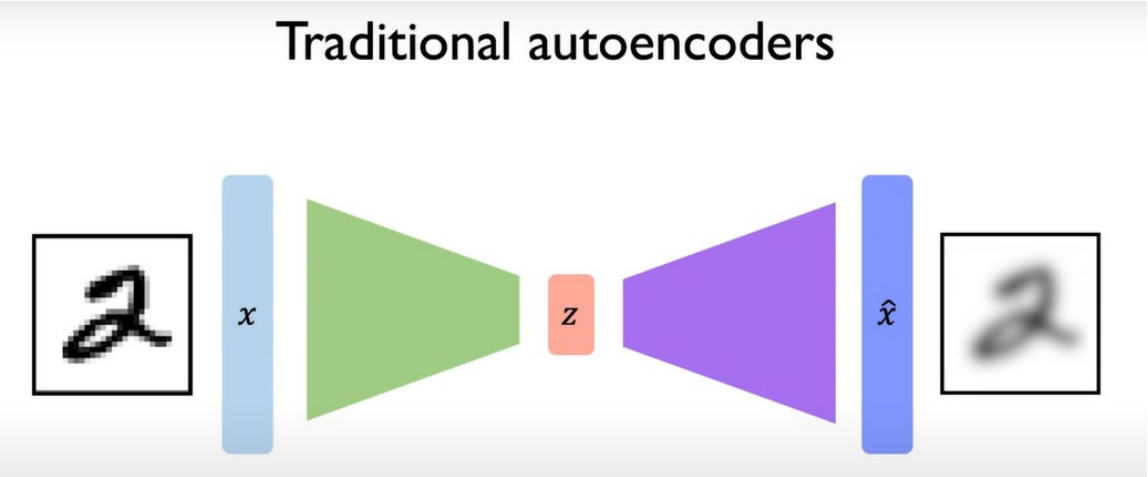
Reconstruction loss forces the latent representation to capture (or encode) as much "information" about the data as possible

Autoencoding = **Auto**matically **encoding** data

- In summary these autoencoder structures use this sort of bottlenecking hidden layers to learn a compressed latent representation of the data and we can self-supervise the training of this network by using what we call a reconstruction loss that forces the autoencoder network to encode as much information about the data as possible as into a lower dimensional latent space while still being able to build up faithful reconstructions.

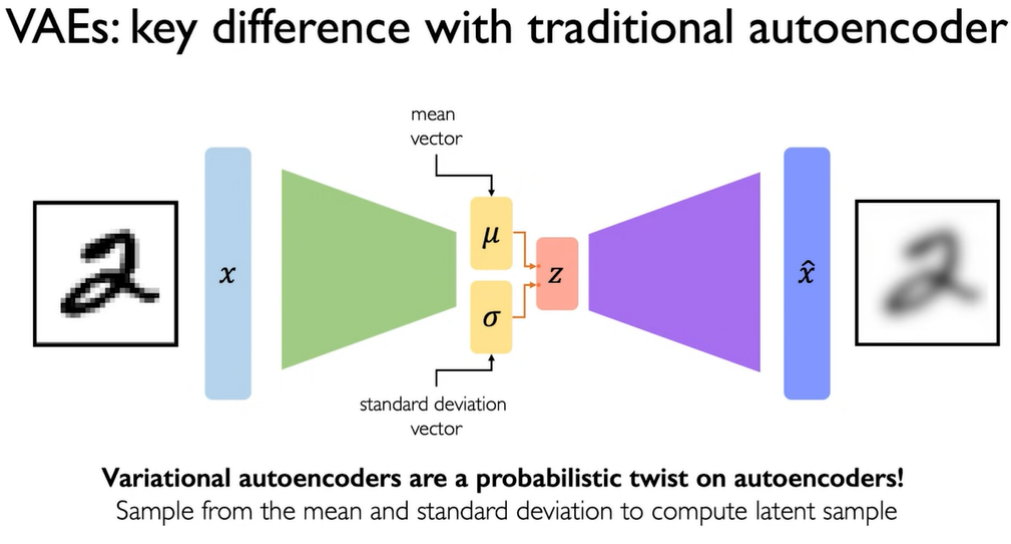
Variational Autoencoders (VAEs)

- As we saw traditional autoencoders go from input to reconstructive output.



if we pay closer attention to the latent layer 'z', it is just like any other layer in the neural network. It is ***deterministic*** i.e. if you feed in a particular input in the network, you're going to get the same output provided that the weights are the same. So, a traditional autoencoder learns this deterministic encoding which allows for reconstruction of the input.

- In contrast, VAEs impose a ***stochastic*** variation on this architecture.



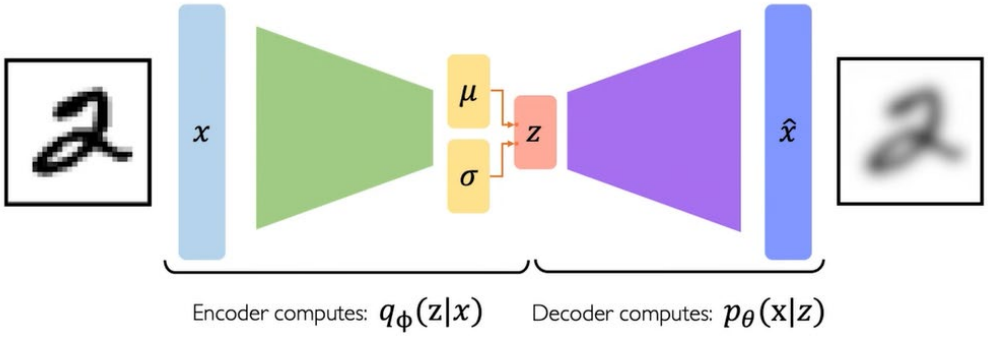
the idea behind doing this is to generate smoother representation of the input data and improve the quality of reconstruction and also generate new images that are simialr to the input data set but not direct reconstructions of the input data.

- This is done by replacing the determinisitic layer 'z' with a stochastic sampling operation. Thus instead of learning the latent features 'z' directly, for each variable the VAE learns a mean "μ" and Variance "σ^2" associated with the latent variable. With the

help of this mean and Variance we can parameterise a probability distribution for that latent variable.

- We can generate new data instances by sampling from the probability distribution defined by this μ and σ^2.

VAE optimization

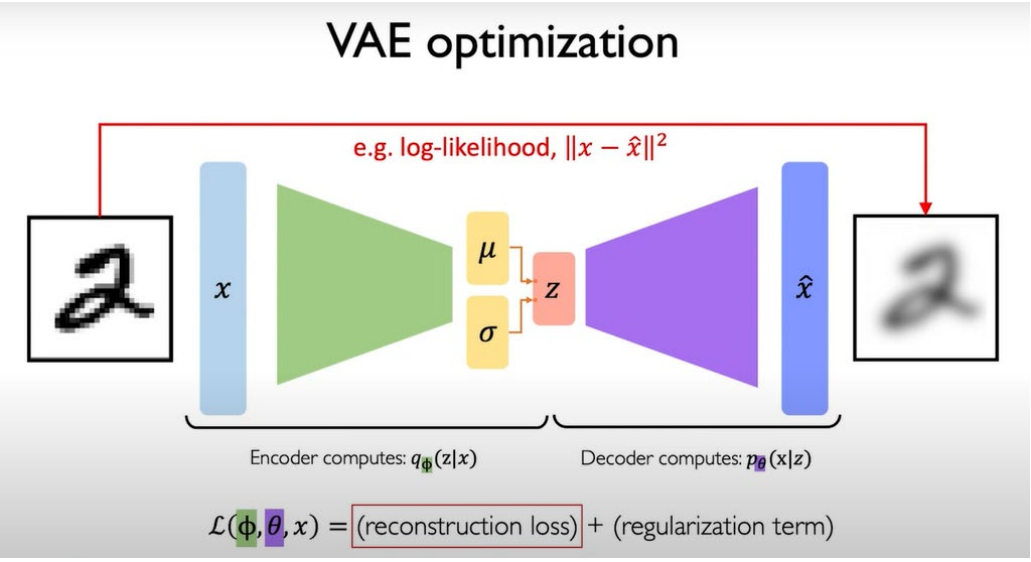


- because we have now introduced this stochastic component, the encoder and decoder are now probabilistic. The encoder tries to learn the probability distribution of latent space 'z' ***given*** the input data 'x' While the decoder taken in the learnt latent-representation and computes a new probability distribution of the of the input 'x' ***given*** the latent distribution 'z'. The encoder-decoder are define by separated sets of weights → φ and theta.
- To train the VAE, we define a loss function which is a function of the input 'x' and set of weights → φ and theta. Now the loss function has two components → reconstruction loss + regularization term.

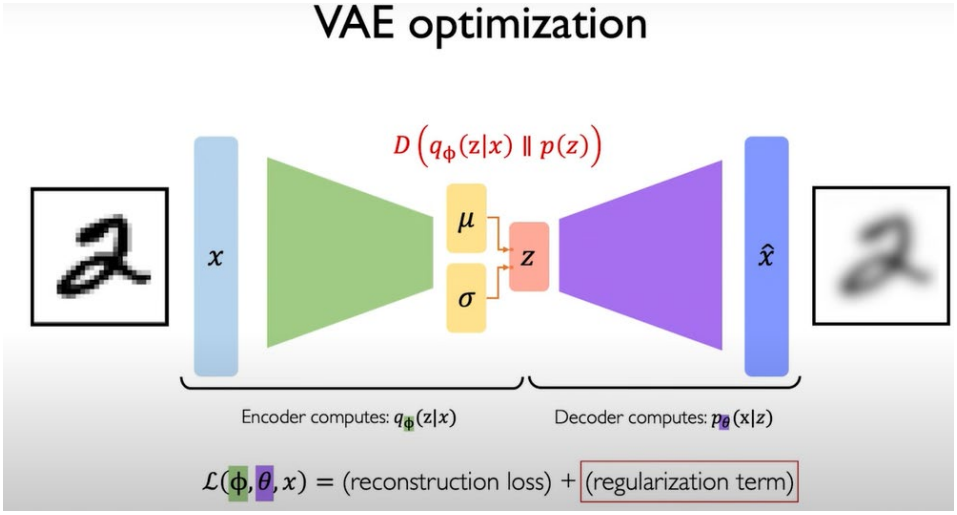
Encoder computes: $q_\phi(z|x)$ Decoder computes: $p_\theta(x|z)$

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

- The ***reconstruction loss*** as before captures the difference between input and reconstructed output. This can be the MSE. As seen before, the network self-supervises the reconstruction loss.



- The regularization loss (also called the VAE loss).



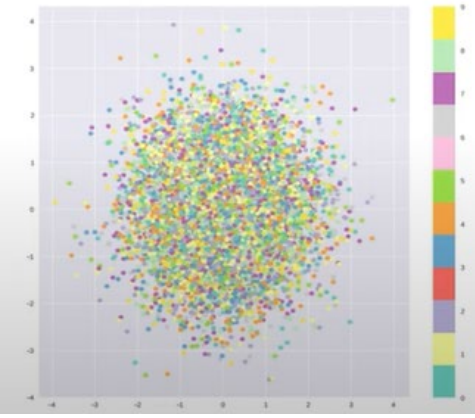
The encoder computes a probability distributed denoted by $q_\phi(z|x)$ is a distribution on the latent space 'z' given the data 'x'. here, regularisation enforces → as a part of the learning process, we are going to place a 'prior' on the latent space 'z' which is some prior belief about how the distribution of z is going to look like.

Basically, we want the difference between Prior Belief p(z) and posterior distribution $q_\phi(z|x)$ to be minimum. This is denoted in entirety by → $D(\ q_\phi(z|x)\ ||p(z)\ )$. 'D' stands for *Divergence*.

This prevents the network from overfitting on certain parts of the latent space by enforcing the fact that we want to encourage the latent variable to adopt a distribution that's similar to our prior.

#Good choice of prior?

- Common choice → Standard Normal distribution



Common choice of prior – Normal Gaussian:

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

- Encourages encodings to distribute encodings evenly around the center of the latent space
- Penalize the network when it tries to "cheat" by clustering points in specific regions (i.e., by memorizing the data)

So if the network tries to memorise outliers, it will be penalised because the loss function will have a huge value.

⇨ $D\left(q_\phi(z|x) \,||\, p(z)\right) = -\frac{1}{2}\sum_{j=0}^{k-1}(\sigma_j + \mu_j^2 - 1 - \log \sigma_j)$

This term is called the Kullback-Leibler Divergence (also known as relative entropy – measure of how one probability distribution is different from another).

Intuition on regularization and the Normal prior

What properties do we want to achieve from regularization? 🤔

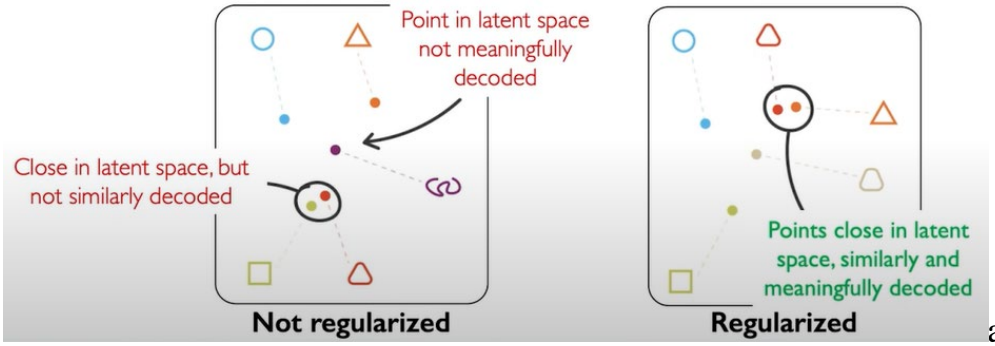
#

1. Continuity

Points which are close in the latent space – they should result in similar reconstruction after decoding. Thus regions in the latent space should have some sense of distance/ similarity to each other.

2. Completeness

When we sample from the latent space to decode it to generate an output, that should result in a meaningful reconstruction – something which resembles the original data distribution.



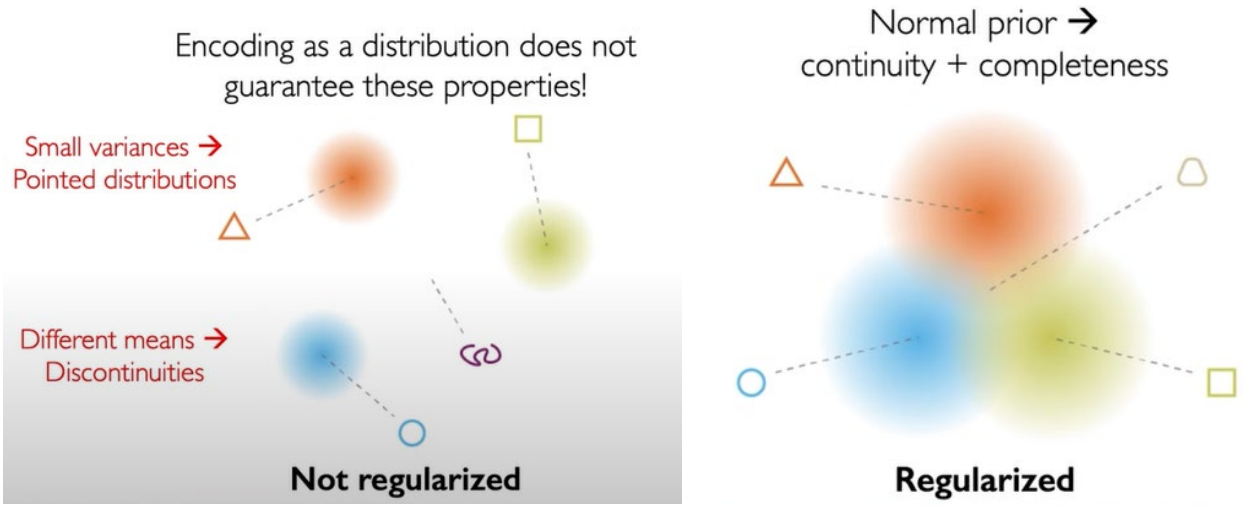
abstract understanding

- Input is a triangle image → latent space variable is represented by dots → if the decoded reconstruction is scribble (purple) or a circle – we do not have continuity or completeness.

Is learning mean and variance sufficient – can this guarantee continuity and completeness? ----- NO!

If there is no regularisation – the loss function has only the reconstruction term – thus the model will try to minimize the reconstruction error even though we are encoding the latent distribution via mean and variance. This would have 2 consequences –

1. We can have very small variances – resulting in point-ed distributions
2. We can have very different means – resulting in discontinuous latent space.



In order to overcome this, we need to regularise mean and Variance of the distribution returned by the encoder. Using the Std. Normal Prior helps us achieve this.
The std. normal prior encourages the learnt-latent-distributions to overlap in the latent space. It will drive all means towards 0 (a centred mean) and Variances towards 1. ***Centring the mean and regularising the variances***. Thus ensuring smoothness and regularity thus achieving continuity and completeness.

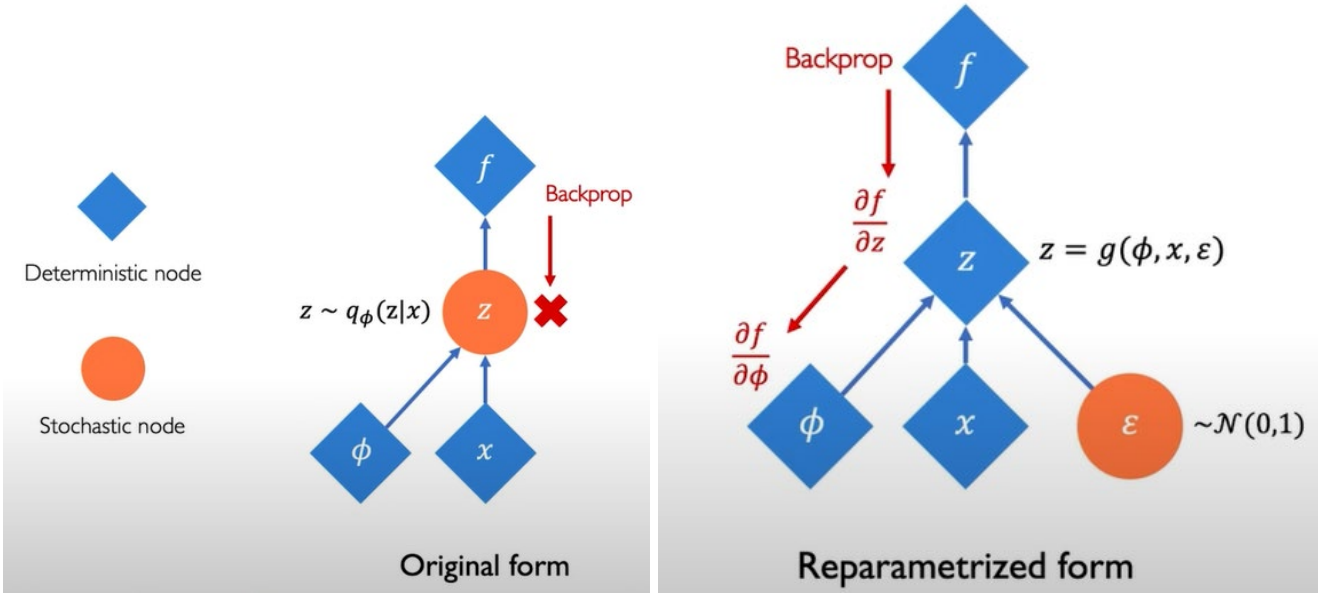
There is a trade-off – regularisation -vs- reconstruction. The more we regularise, the quality of reconstruction goes down.
Now we have understood how we can regularise learning and achieve continuity and completeness via the normal prior. These were all the components which define a forward-pass through the network going from input → encoding → decoding →reconstruction. The only missing part is ***Backpropagation***.
Now because we have this stochastic sampling layer – we now have a problem – we cannot backpropagate gradients because backpropagation requires deterministic nodes/layers for which we can iteratively apply the chain rule to optimize the loss via gradient descent.
VAEs introduced sort of a breakthrough idea that solved this issue of not being able to backpropagate through a stochastic sampling layer. The key idea was to subtly re-parameterize the sampling operation such that the network can be trained end-to-end.
#Re-parameterizing the sampling layer

Here we will consider the sampled latent vector ‘z’ as a sum of

- A fixed vector μ
- A fixed vector σ : scaled by random constants drawn from the prior distribution

⇨ $Z = \mu + \sigma \odot \epsilon$ where $\epsilon \sim N(0,1)$.

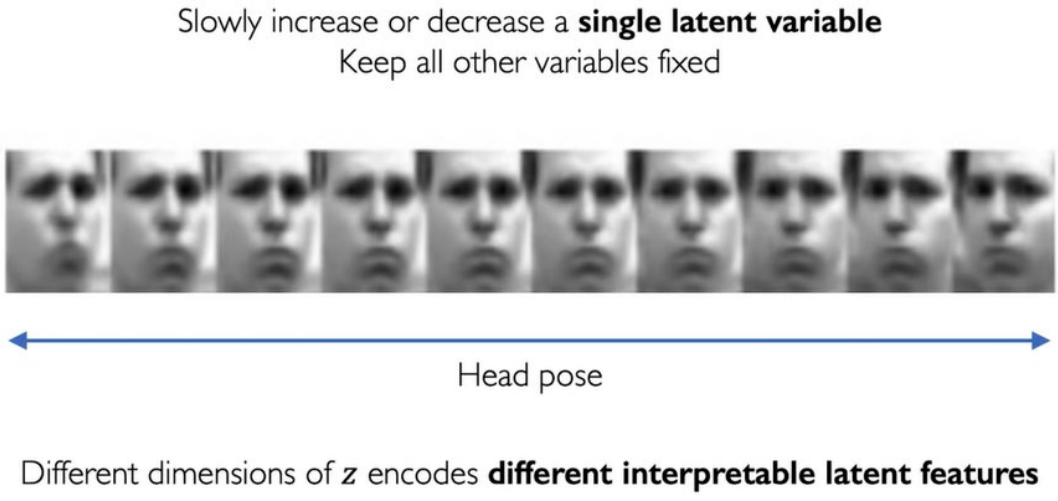
By reparametrizing the sampling layer this way, we still have this element of stochasticity but that stochasticity is introduced by this random constant epsilon which is not occurring within the bottleneck latent layer itself. we have reparametrized and distributed it elsewhere.



- Original form → we had deterministic node φ (weights) and input 'x' and we were stuck trying to backpropagate through stochastic sampling node 'z'.
- Reparameterization form → now latent layer 'z' is a function of/ define by μ, σ2 and ϵ (noise). Now when we backpropagate, we can assume ϵ value to be constant and easily backpropagate via μ and σ^2.

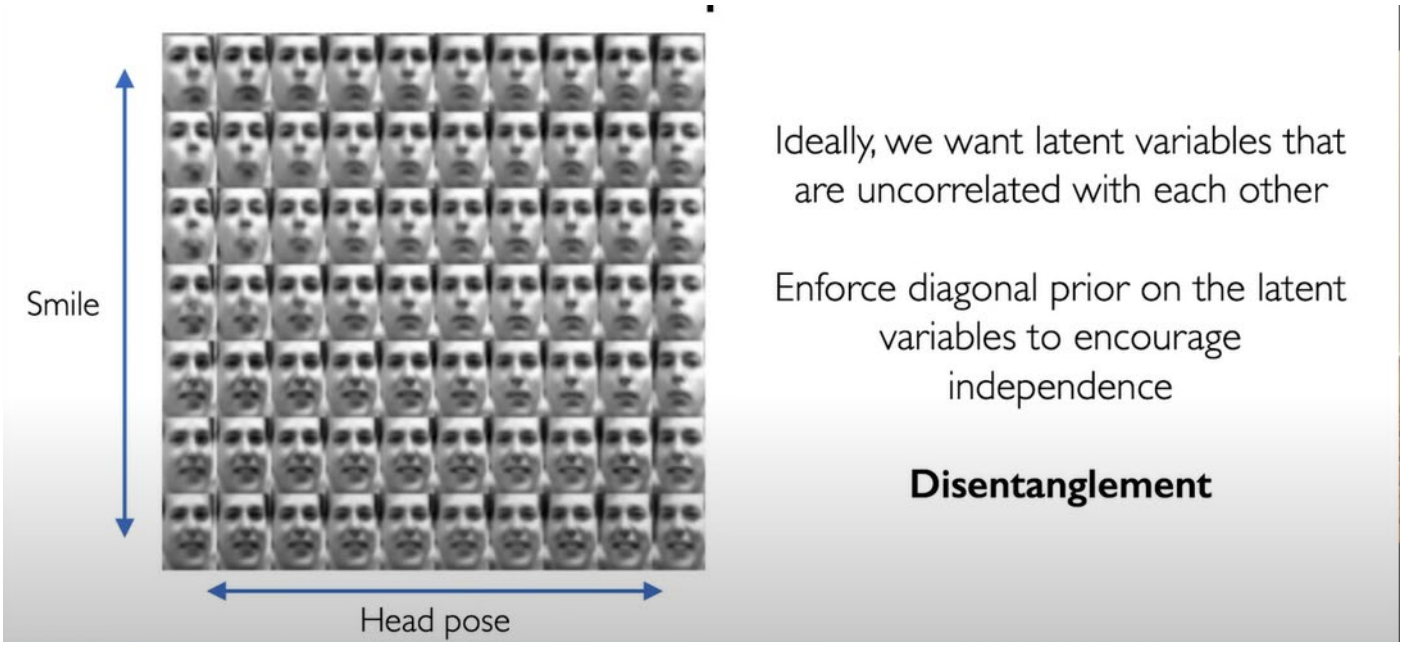
#VAEs – latent perturbation

Side effect of imposing distribution priors on the latent variable → we can actually sample from these latent variables and individually tune them while keeping others fixed. We can tune the value of a particular latent variable and run the decoder each time that variable is changed/ perturbed to generate a new reconstructed output.



Suppose we keep all other facial features fixed, change only a single latent variable which translates into different head pose.

Ideally, to optimize VAE i.e. maximise information that they encode – we want these latent variables to be uncorrelated with each other i.e. effectively ***disentangled***.



- Here we have head pose on X-axis and Smile on Y-axis
- We want these to be as uncorrelated with each other as possible.

#Latent Space disentanglement with β-VAEs

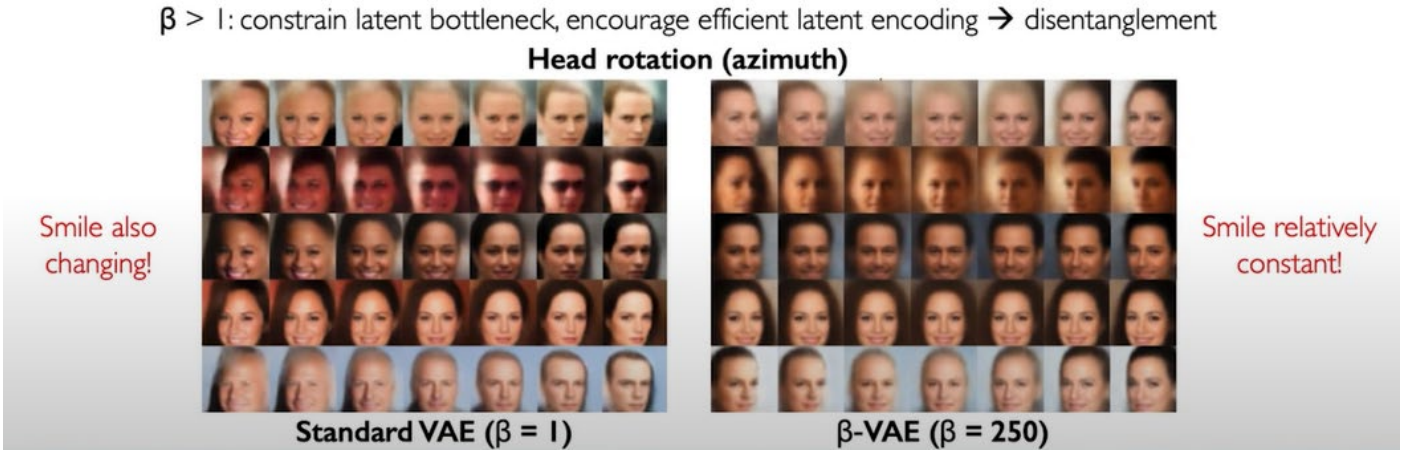
Standard VAE loss:

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction term}} - \underbrace{D_{KL}\left(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z})\right)}_{\text{Regularization term}}$$

β-VAE loss:

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction term}} - \underbrace{\beta D_{KL}\left(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z})\right)}_{\text{Regularization term}}$$

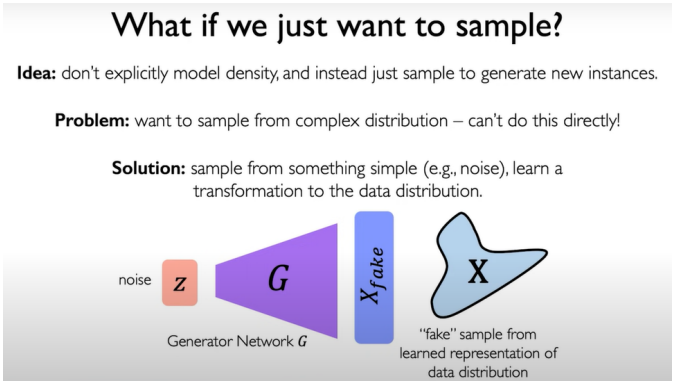
Under β-VAEs, we introduce a hyper-parameter β which controls the strength of this regularization term. It has been proved that by increasing β the effect is to place constraints on latent-encoding thus encouraging disentanglement.



- Left image → β = 1 i.e. we used a standard VAE. So if we consider the latent variable which translates to head-pose, for β=1, as the head-pose is changing the smile of some of the faces is also changing (because they are correlated i.e. entangled).

- Right image → β = 250 i.e. we used a β-VAE. Here, the smile remains relatively constant while we can perturb/ change the single latent variable of head-pose alone.

#Generative Adversarial Networks GANs

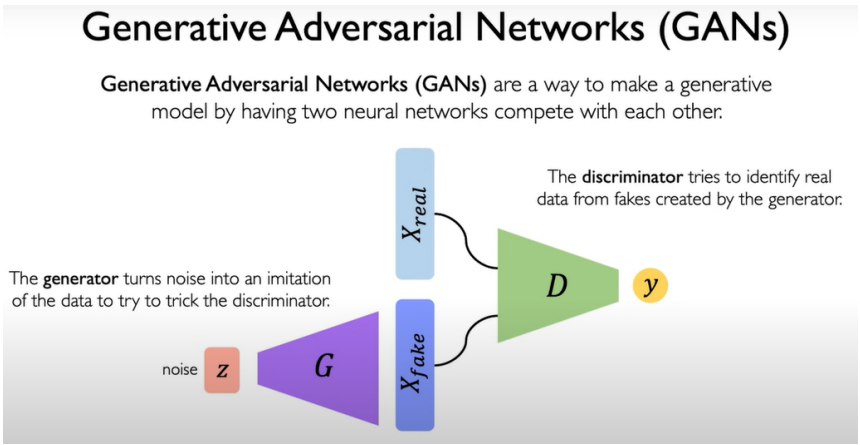


Idea – we don't want to model some density underlying some data, instead learn a representation that can be successful in generating new instances that are similar to the data. We want to optimize 'to sample from a very complex distribution' which cannot be learned and modelled directly – instead we are going to have to build up some approximation of this distribution.

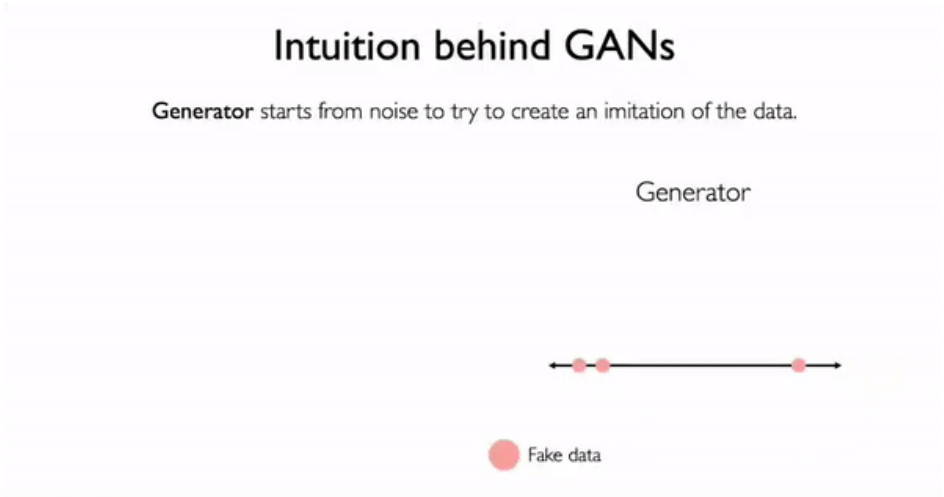
Here we start from Random Noise – try to build a generative neural network which will learn a functional transformation that goes from noise → data distribution. By learning this functional generative mapping we can then sample in order to generate "fake/synthetic" instances that are going to be as close to the real data distribution as possible.

The breakthrough in achieving this was the GAN structure – having these 2 neural networks – 1. Generator network and 2. Discriminator network. These are effectively competing against each other. They are Adversaries.

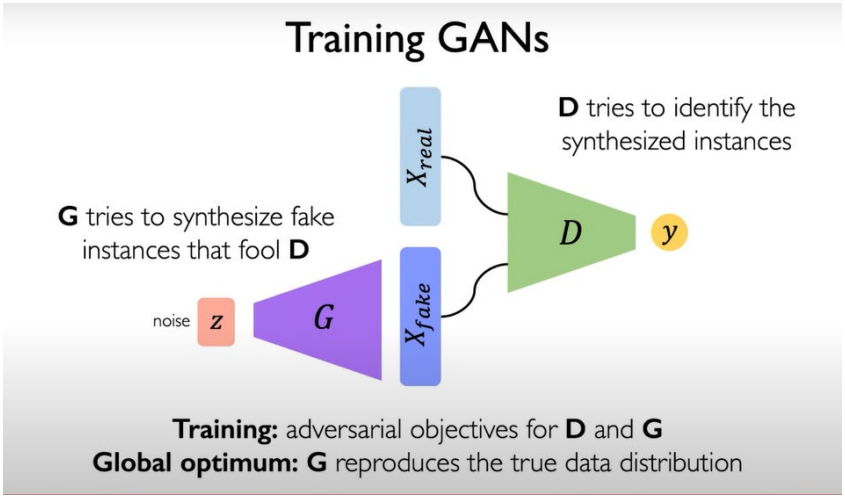
The generator network 'G' is trained to go from random noise to produce an imitation of the data and then the discriminator network 'D' is going to take that synthetic data and the real data and be trained to differentiate between the fake and real. Thus, in the training process, these two are going to be competing against each other – the overall effect is that the Discriminator is going to get better and better at learning how to classify real and fake and the better it becomes at doing that – it's going to force the Generator to produce better and better synthetic data to try to fool the Discriminator.



#Intuition behind GANs



- Generator starts from noise to try to create an imitation of the data.
- Discriminator looks at both real and fake data created by the Generator – it is then trained to output a probability that the data it sees are real/fake. The beginning of the training process, the discriminator will perform poorly, but with training iterations, it will get better.
- Now we are back to the Generator, it will take instance of where real data lie as inputs and then improve its imitation of the data – trying to move the synthetic data closer to the real one.
- The Discriminator now receives the new points – estimate probability of being real – train to estimate better.
- The Generator will now further move the fake points closer to the real ones such that fake data is almost following the distribution of the real data. At this point, it is really hard for the Generator to distinguish between real and fake.



To summarise the training process -
To actually train we define a loss function which depicts the competing and adversarial objectives of the Discriminator and Generator. The *Global Optimum* would be the point where Generator can perfect reproduce true data from noise, and the discriminator can no longer distinguish.
The loss function for GAN is based on cross-entropy loss defined between the true and generated distribution.

#Loss from the perspective of Discriminator

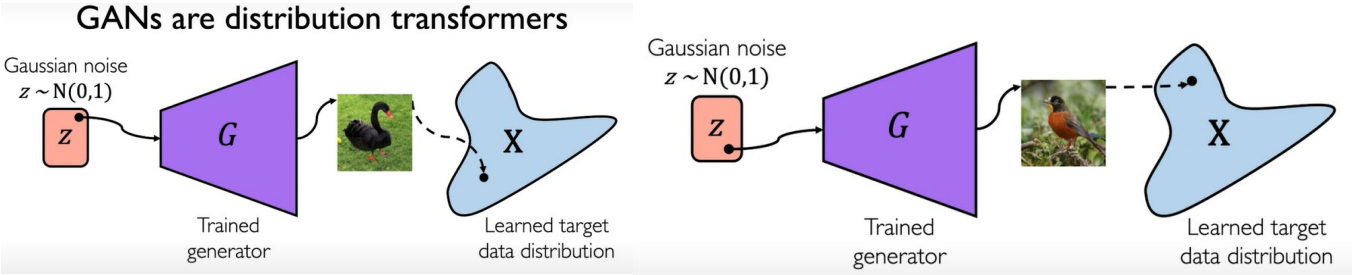
$$\arg\max_{D} \mathbb{E}_{\mathbf{z},\mathbf{x}}[\underbrace{\log D(G(\mathbf{z}))}_{\text{Fake}} + \underbrace{\log(1 - D(\mathbf{x}))}_{\text{Real}}]$$

- we want to maximise the probability that fake data is fake.
- For this we define the output of Generator as G(z).
- The discriminator's estimate of probability that fake is fake is D(G(z))
- The discriminator's estimate of probability that real is real is 1-D(x).
- So from the Discriminator perspective – we maximise both these probabilities.

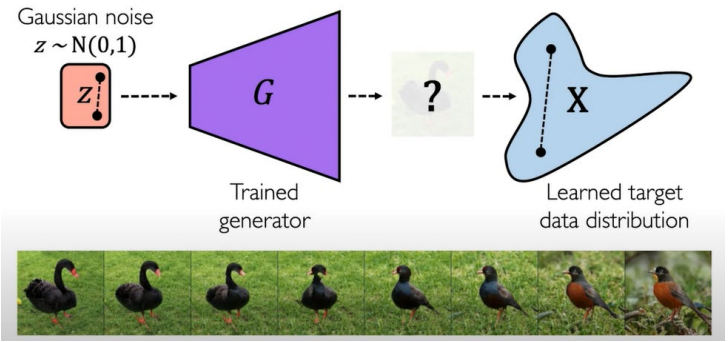
#Loss from the perspective of Generator

- The generator takes in noise and tries to generate an instance.
- So it cannot directly affect the D(x) term in the loss function because it is solely based on the operation of Discriminator on real data.
- The generator has an adversarial objective to the discriminator – minimize the probability D(G(z)).
- Thus the ultimate goal is captured by the min-max objective function which has the two components.

After training the Generator network is trained to produce new data instances that have never been seen before by the network, from noise. When the trained Generator synthesis new instances, it is effectively learning a transformation from noise → target data distribution. This transformation/mapping is learnt over the course of training.



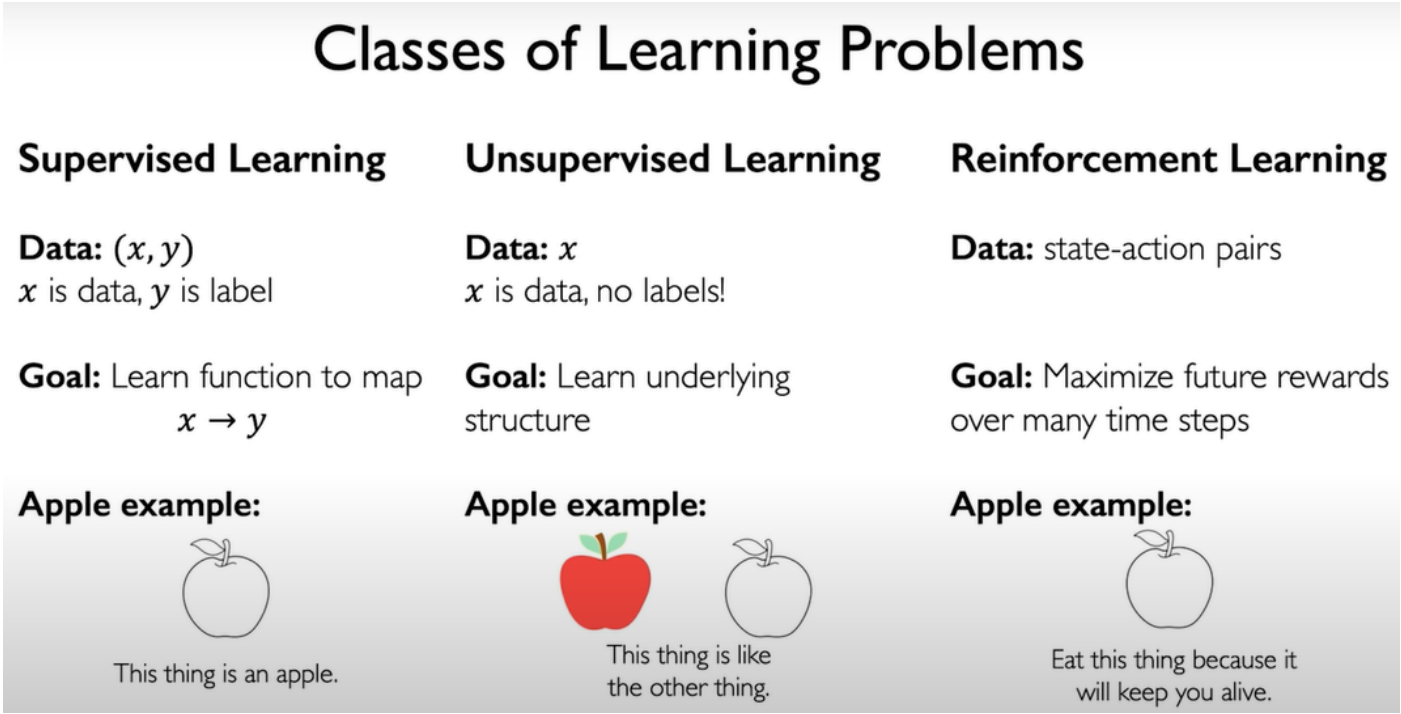
So if we consider one point from a latent noise distribution – it will result in a particular output in the target data space. Another point = new instance.



Thus we can actually interpolate and traverse in the space of gaussian noise to result in interpolation in target space as shown in the picture.

Deep Reinforcement Learning

So far, we have dealt with fixed data sets. In reinforcement learning, deep learning is placed in some environment where it can explore and interact with the environment, learn how to best accomplish its goal. Usually it does this without any human-supervision/guidance. E.g. robotics, game play and strategy.



- Under supervised learning, we are given both the data x and its label y. our task to predict labels for unseen data x.
- Under unsupervised learning, we are given only the data x and no labels our task is to learn the underlying structure. We don't know that it is an apple, we know the two are very similar.
- Under reinforcement learning, we are given with state-action pairs. States are the observations of the system and actions are the behaviors of the system or agents it takes when it sees those states. The goal is to maximize the reward/ future reward of that agent in that environment over many time steps in that environment. In the apple example, we don't know what it is (no labels) but we know consuming the apple is good i.e. it maximises some reward-function like survival chances in real life.

Key vocabulary

- Agent → is something that can take actions is the environment. E.g. drone making a delivery, super mario navigating in a game.
- Environment → the world in which the agent lives/exists and operates.
- Actions → Thus Agent and Environment is related/connected by "actions" which the Agent can take in the Environment.
- An action can be denoted by a_t i.e. an action at time 't', while the set of all possible actions is A i.e. the *Action Space* {discrete or continuous}.
- Observation → are ways in which the environment reacts to the actions of the agent. Through these reactions-of-the-environment OR observations, the agent can observe where it is in the environment and how it's actions affect its current *state*.
- State → is a situation which the agent perceives or finds itself.

E.g. A delivery-drone (agent) is hovering in the sky (environment). It's wing rotate (action) and it moves forward in the environment (observation). It is moving forward (state). The GPS coordinates (continuous Action space containing different possible directions) helps it know/perceive where in the environment it is.

- Reward → a way of feedback from the environment to the agent – to measure success or failure of agent's action. Rewards can be either immediate or delayed.

So for the drone, an immediate reward could be moving in the correct direction. A delayed reward could be completion of the delivery in time with efficiency.

- Total Reward – sum of all rewards collected by the agent from time 't' to '∞'.

$$R_t = \sum_{i=t}^{\infty} r_i = r_t + r_{t+1} + \cdots + r_{t+n} + \cdots$$

- Discounted Reward – often we consider discounted reward instead of the sum of all rewards. Basic time-value of money concept. Discounting factor → γ (range: 0 to 1) has a dampening impact on the agent's choice of action.

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^t r_t + \gamma^{t+1} r_{t+1} + \cdots + \gamma^{t+n} r_{t+n} + \cdots$$

Why? This makes future rewards less important than immediate rewards. It enforces a short-term learning in the agent.

So for the drone, flying through open windows or across people homes will enhance the final reward because delivery was made fast. But the short term rewards are neglected. By using discounted reward system, we can prioritise short-term rewards like not deviating from the safer flight path.

- Q-function

Total Discounted Reward $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$ is the discounted value of rewards obtained since time 't'. The Q-function is a function takes as input the current state of the agent and action the agent takes in that state and then returns the expected total future reward. $Q(s_t, a_t) = E(R_t | s_t, a_t)$.

Our task is to find the best policy which maximises this expected return or the Q-function given our current state. We can perform different set of actions (or policies) and pick the one with the highest Q-value. $\pi^*(s) = argmax\ Q(s, a)$ where π represents the chosen policy.

2 classes of reinforcement learning algorithm

1. Value Learning

Find Q(s, a) where a = argmax Q(s, a)

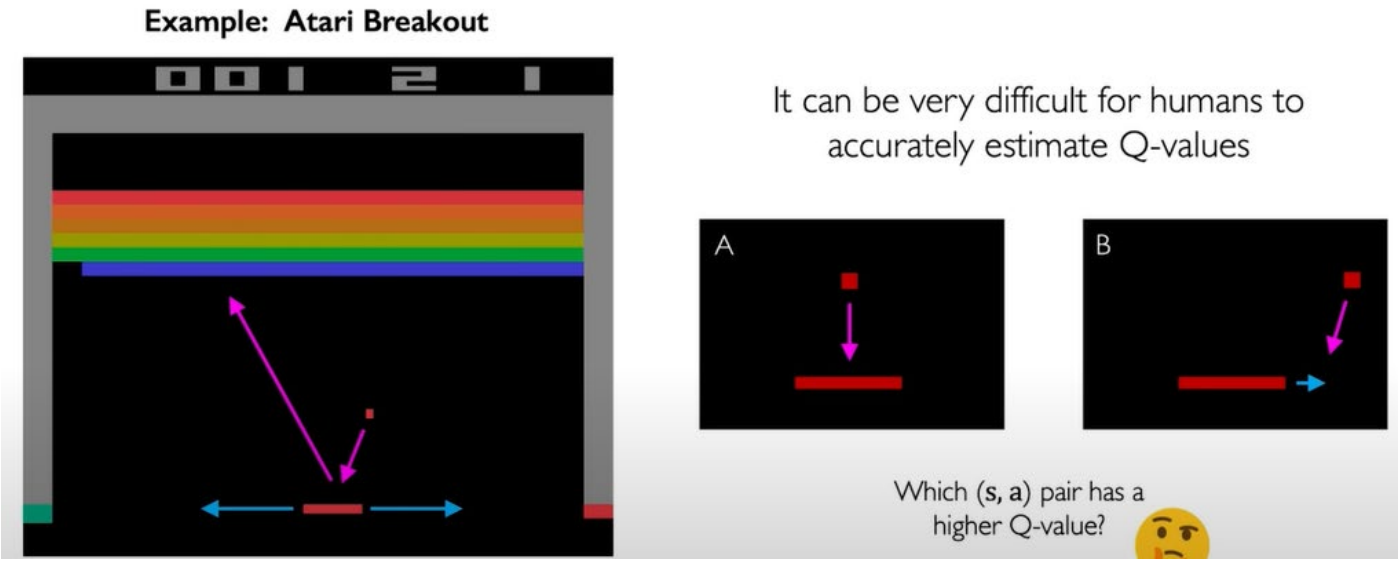
Here the aim is to learn the Q-function.

2. Policy learning

Find π (s) sample a ~ π (s)

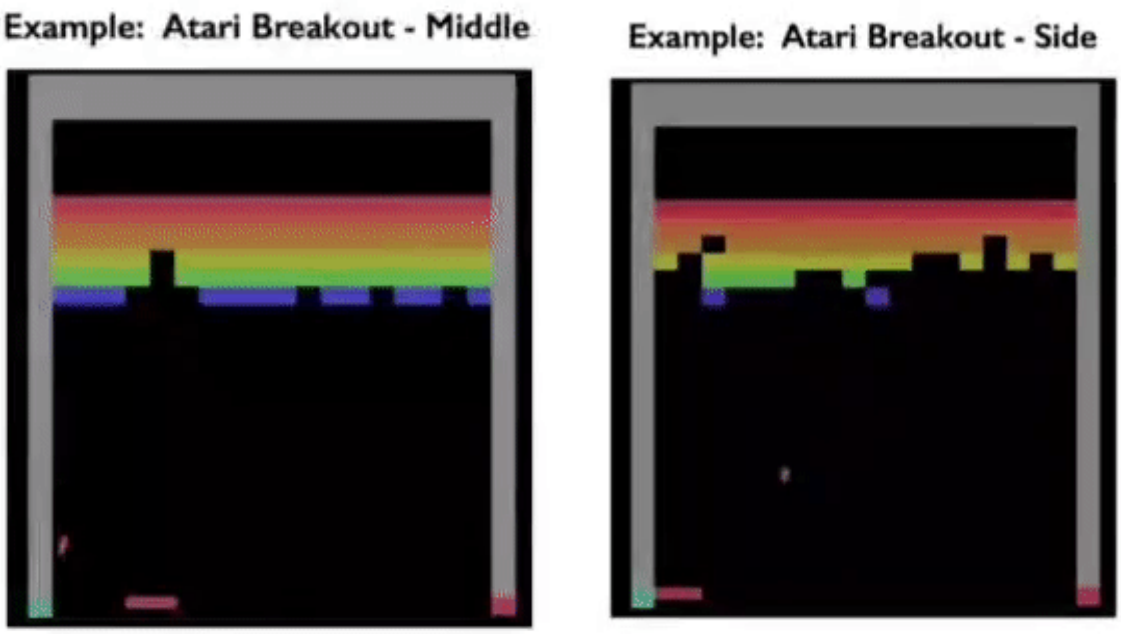
Here the aim is to directly learn the policy instead of using the Q-function to infer your policy.

#Digging deeper into the Q-function – Atari Breakout Game



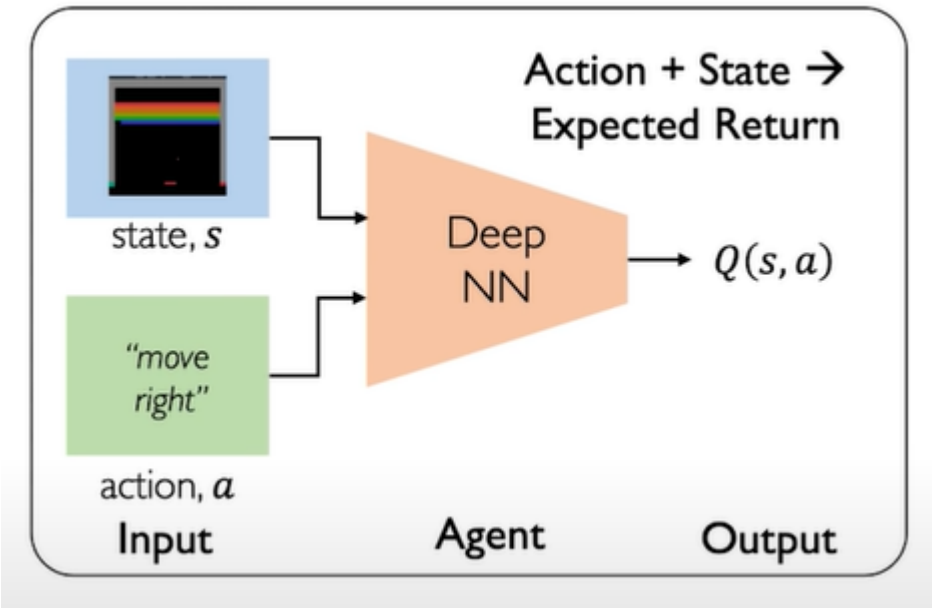
Among the two state-action pairs, we cannot tell for sure which yields the max-future-return. In the first, the ball drops perpendicularly, and on the other it falls at an angle – hitting at the corner of the paddle.

Option A – break out most of the center. Option B – break out from the corners.

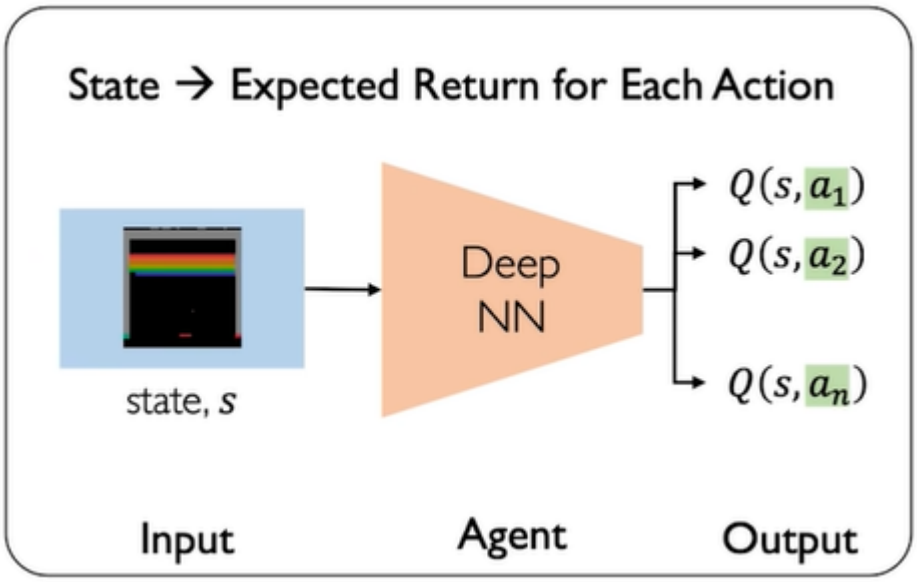


Under option B, the ball gets stuck over the block and keeps yielding points/ more reward – in much shorter time and less action/ movement of the paddle. Thus, option B has the higher Q-function value. But it's a relatively un-intuitive option. Thus making it hard to define the Q-function. Instead of defining a Q-value function we can use an deep neural network that gets input of state-action pairs and output is predict the Q-value.

Deep Q Network



But this is rather inefficient because for every time step, we need to run the model 'n-times' because there are n-possible-actions to take at a given time step. Then, we need to perform n-calculations for all t-time steps.

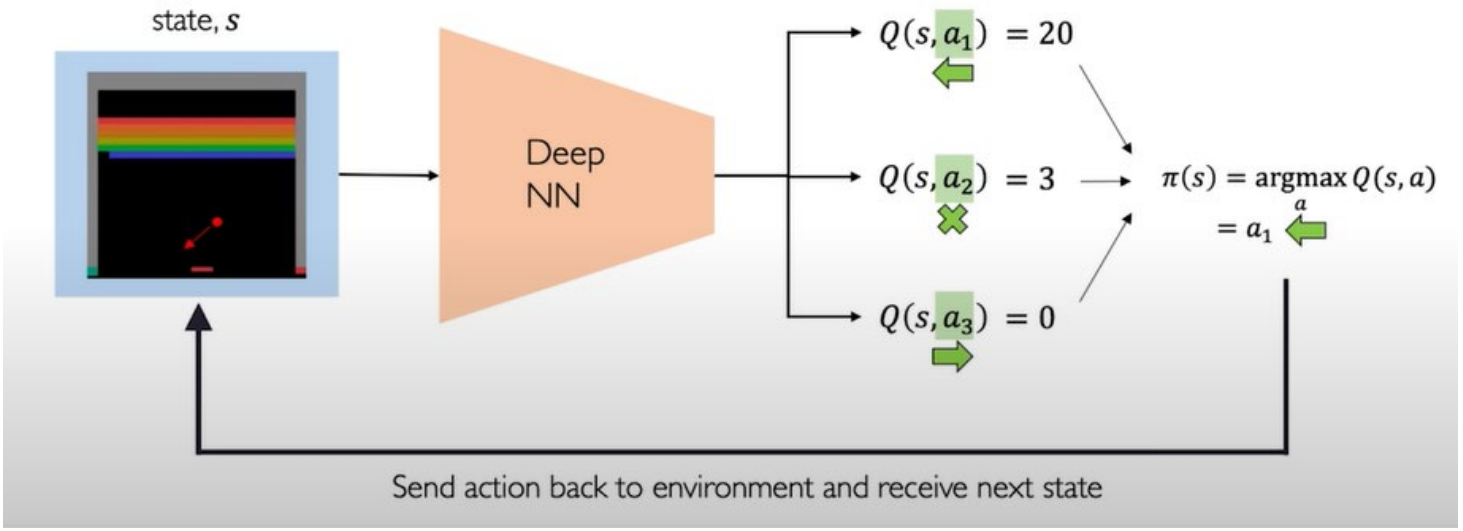


It is rather better to output all of the Q-value at once as a vector. Here we need to run the algorithm only t-times for t-time-steps and as output we get a n-dimensional vector of Q-values as output.

For training we use a Q-loss Function given by –

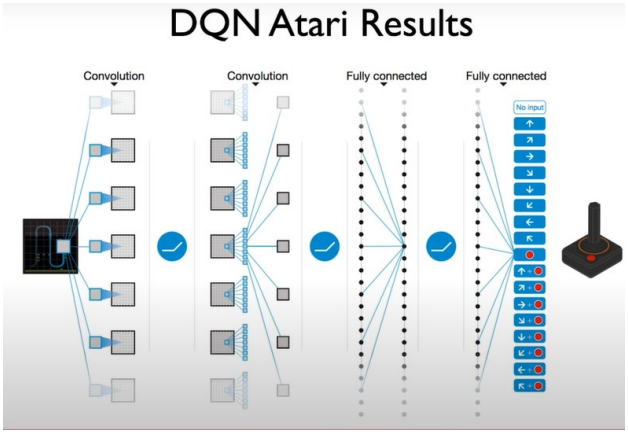
$L = E\left[\left|\left(r + \gamma^* maxQ\left(s', a'\right)\right) - Q(s, a)\right|^2\right]$. This is a kind of a mean-squared error function where we take expected value of squared difference of target function and prediction.

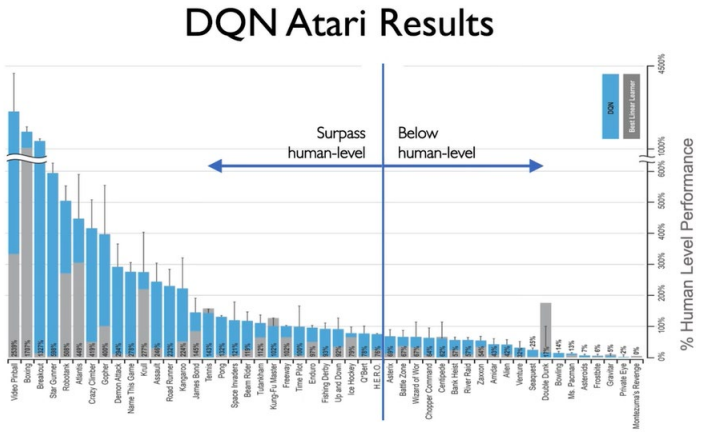
Target function → $\left(r + \gamma^* maxQ\left(s', a'\right)\right)$. We formulate the expected return provided we take the All-best-actions max Q(s', a'), discounted by γ and add initial reward 'r'.



- 3 possible actions – move to right, left or stay.

- Our deep neural network sees as input a state. We try to output the Q-value for each of the 3 possible actions.
- In order to infer the optimal policy, it has to look at each of the q-values, shown in the photo.
- Optimal policy = pick the action which maximises Q-value, here, a_1 i.e. move to left.
- We then send the choice of action to the game-engine and we receive our next state.
- Repeats.





- Atari Gaming Company – applied this method and was able to surpass human-level performance in over 50% games.

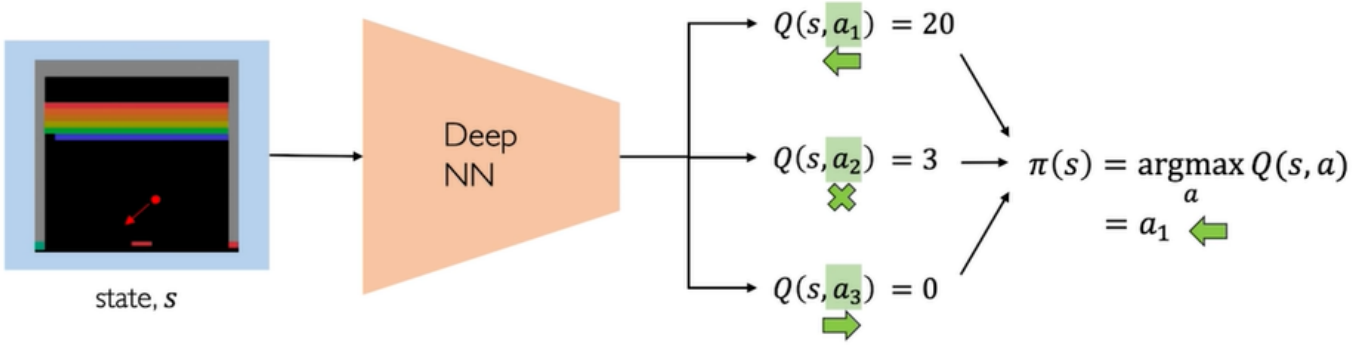
Downside

- Complexity
  - Performs well for small and discrete action space but cannot handle continuous ones
- Flexibility
  - Limited flexibility – unable to learn stochastic-policies i.e. policies which can change based on some unseen probability distribution.
  - It can only pick deterministic policies which yield the max-Q value or expected return.

To deal with this, we turn to ***Policy-Gradient methods***. The key difference is that:

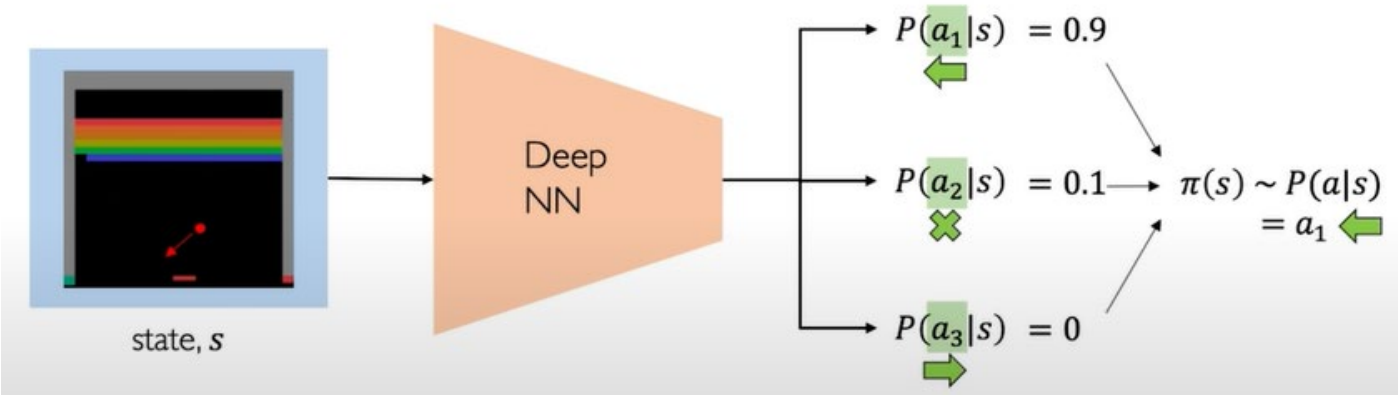
- Under value-learning, we have a neural network to learn our Q-value (expected return in a state given a particular action) which is used to find the best action to take.
- Under policy learning, directly learn the policy using the neural network via input a state.
  Find $\pi(s)$ i.e. a policy sample for state 's' and as a result we get an action 'a' $\sim \pi(s)$.

DQN: Approximate Q-function and use to infer the optimal policy, $\pi(s)$



- Under Value learning – we predict Q-value for each possible action given a state. Then we pick the "best" action which gives max Q-value i.e. the max expected return. Then we execute that action.

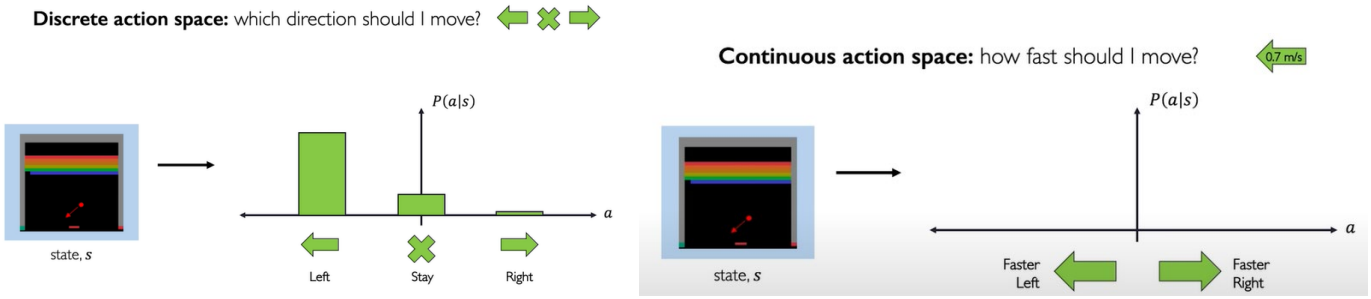
Policy Gradient: Directly optimize the policy $\pi(s)$



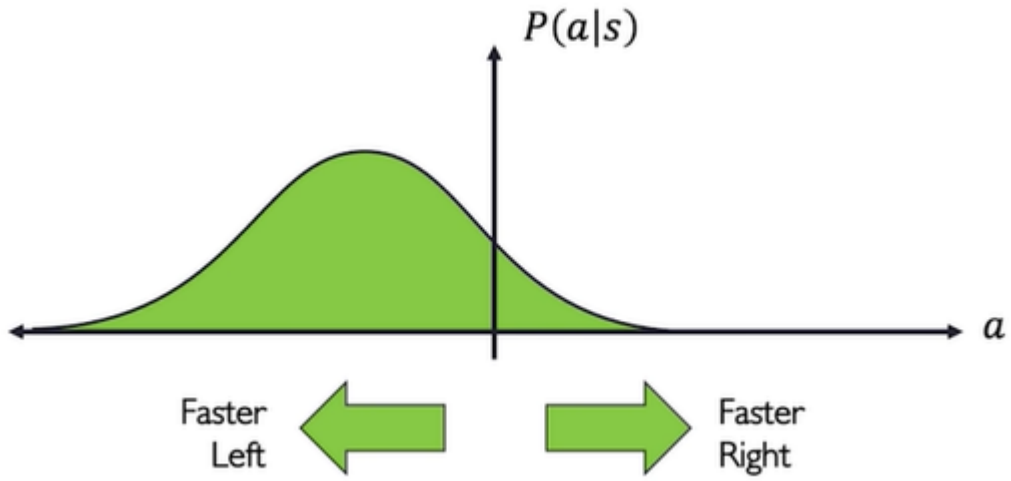
- Under policy learning – we do not predict q-value but directly optimise the policy π(s). here the policy-distribution is governing how we should act given a current state. Output – probability that an action is the "best" i.e. yields maximum reward given a current state. So now instead of framing a policy based on best-actions, we are framing a policy which is a probability-distribution of all possible actions from which we are choosing the action which has max-probability.
- Obviously because we are dealing with probabilities, the sum of all probabilities is 1. $\sum_{a_i \in A} P(a_i|s) = 1$.

Advantages of Policy-Learning?

- Much more direct way. Instead of optimising a Q-function we directly optimise the policy.
- ***This can handle continuous action-space. Value learning could only deal with discrete.***

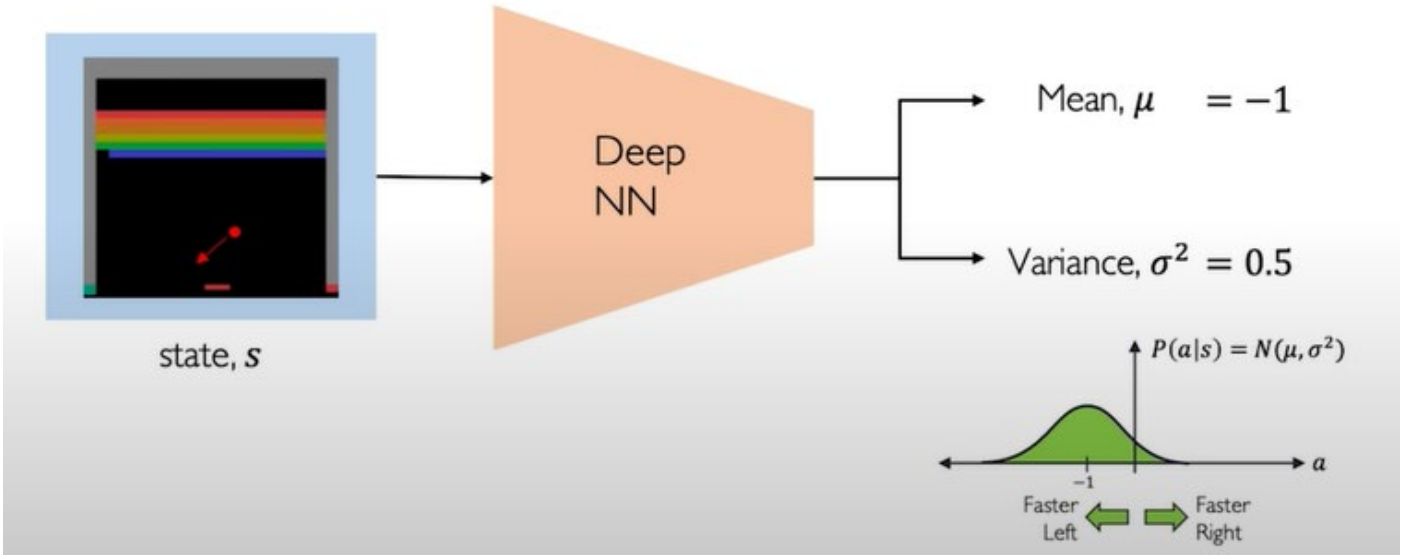


- Under continuous action-space, we seek answers not only to discrete questions (like which direction: right, left or none) but also continuous questions (like how fast) which can have infinite answers. +/- tells us the direction of movement.



- Using a probability distribution like Gaussian, we have say that the probability of moving left > probability of moving right. The mean of the probability distribution tells us how fast we should be moving left.

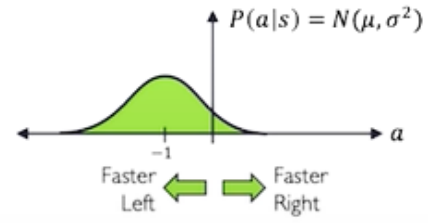
How to model continuous-action space using Policy Gradient (PG)



Here because we are working in continuous action-space we cannot predict which action to take because there are infinite possible actions thus infinite outputs. Instead of predicting

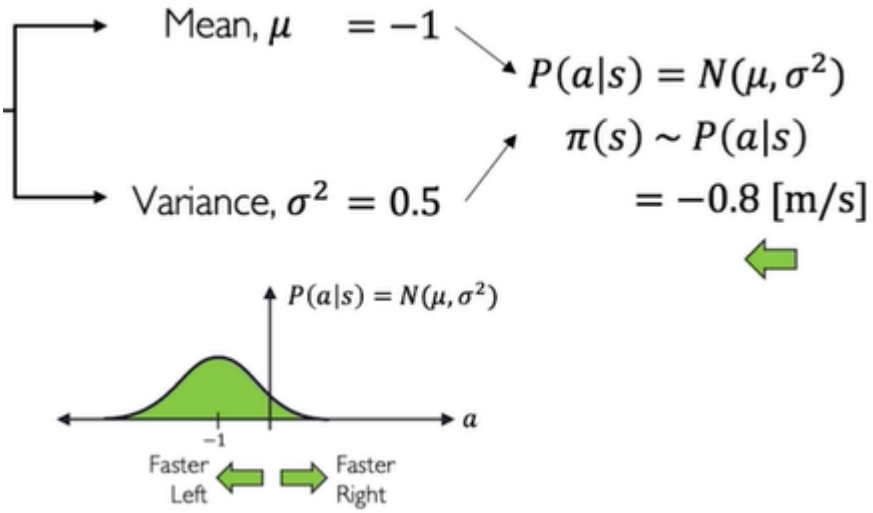
probabilities of every possible action, we want the mean μ and variance σ^2 which describe the probability-distribution.

If the output is μ = -1 and σ^2 = 0.5, the probability distribution looks like this - .



. it should on-average move to the left (indicated by negative sign) at a speed of 1 m/s with some variance.

If we sample an action from this, we get

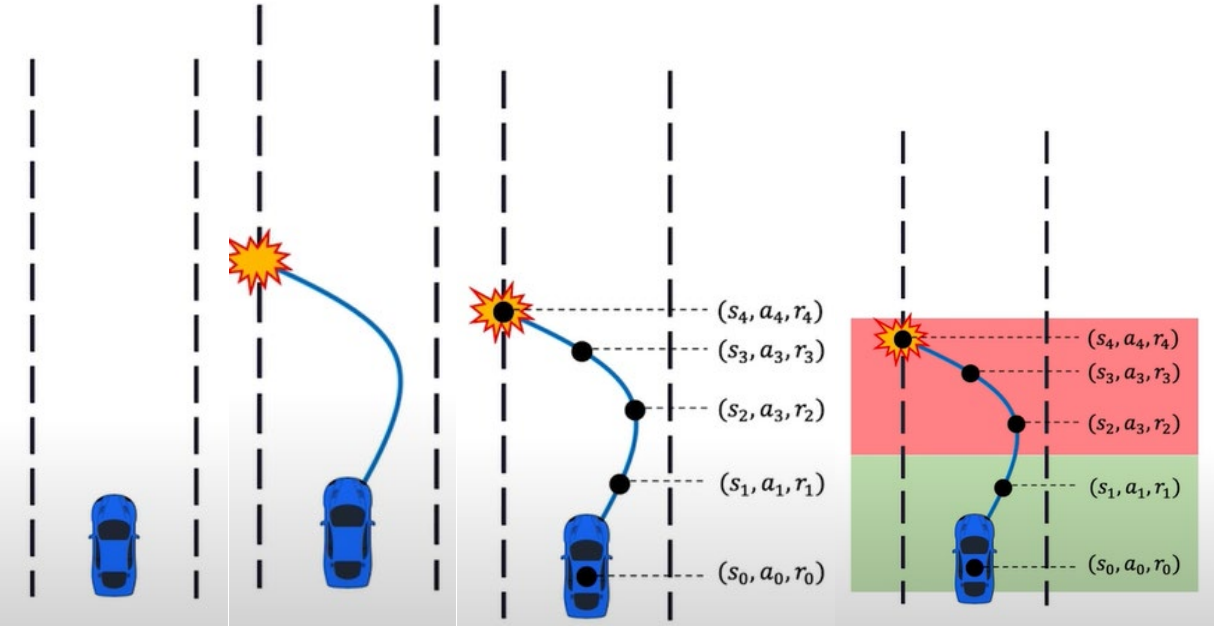


which indicates that we need to move left (negative) at a speed of 0.8 m/s.

Training policy gradients – case study of self-driving car

- Agent : vehicle (which is travelling in the environment, the world, the lane).
- State : camera, lidar etc. (state of the car is obtained via camera data, lidar data etc)
- Action : steering wheel angle (it can change the steering wheel angle – continuous action-space)
- Reward – distance travelled (how far it travelled before requiring human-intervention)

Training algorithm



1. Initialize the agent
2. Run a policy until termination

We have not yet trained the network in any way. We allow it to run any policy until the car crashes. We take records of these crashes. We call them 'roll-outs'

3. Record all states, actions, rewards

Along every roll-out, we record all states, actions and rewards. This tells us where the car was, its state, what action it took and resulting rewards.

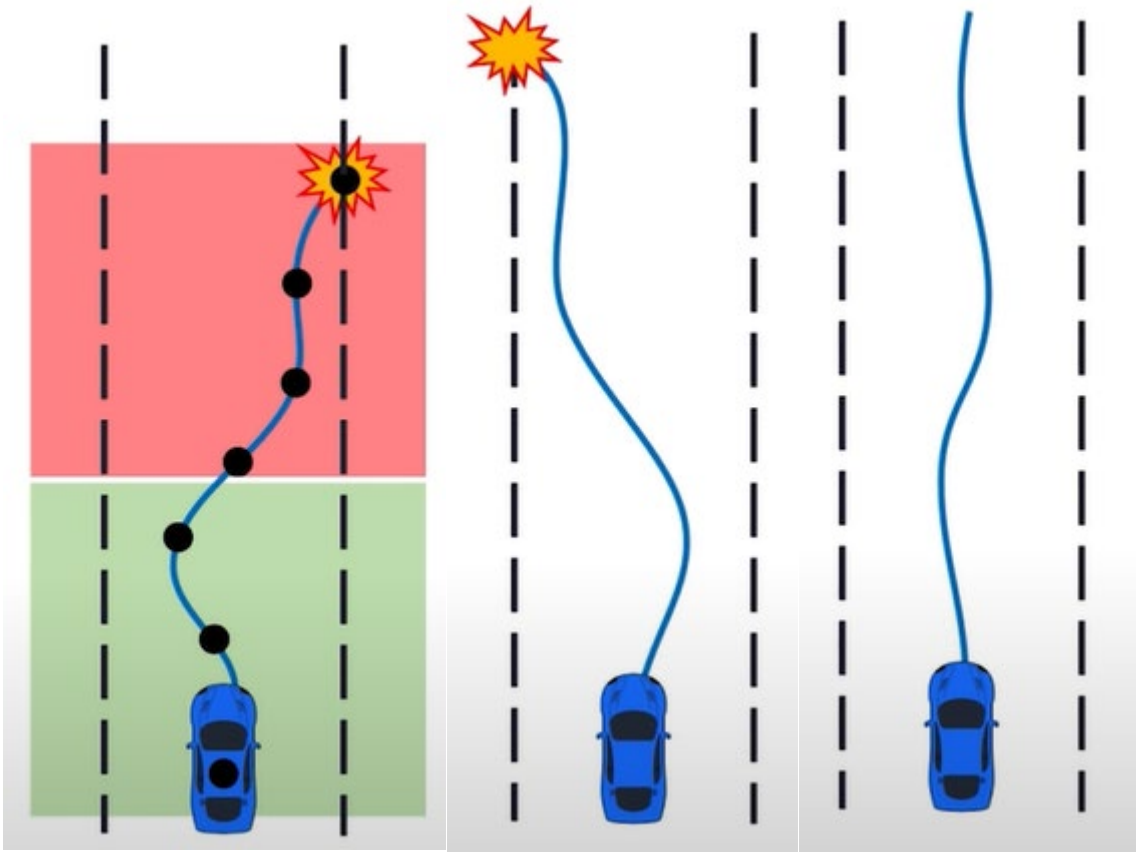
4. Decrease probability of action that resulting in low-reward.

We take those state-action-reward pairs and decrease the probability of taking those actions leading up to termination/crash.

5. Increase probability of actions that resulted in high-reward

Note – we do not necessarily know that there is anything good in the first part of the episode (green). We are just assuming that the termination/crash happened because of some action in the second episode (red). This is a very un-intelligent at the beginning because it does not have any feedback.

The next time →



So without any knowledge of what lanes are, what signals mean – the car learned to avoid crashes based on optimizing the reward function learnt based on its crashing-experience.

How do we update our policy on every training iteration? How can we increase and decrease probabilities of step 4 and 5?

- Loss function $-\log P(a_t|s_t)R_t$
- The first term is a log-likelihood term of our policy i.e. probability of an action given a state.
- The second term is the total-discounted reward R_t.
- Suppose an action has high reward and high log-likelihood i.e. low probability – this loss will be great – and get reinforced because they resulted in very good returns.
- Suppose an action has low reward and high probability – this loss will be small – will not get reinforced.
  -log(0.01) = 4.6 while -log(0.1) = 2.3

When we plug-in this loss function to the gradient descent algorithm to train our neural network we can see that the policy gradient term (blue)

log-likelihood of action

$$\textbf{loss} = -\log \mathrm{P}(a_t|s_t)\, R_t$$

reward

Gradient descent update:

$$w' = w - \nabla \textbf{loss}$$

$$w' = w + \nabla \log \mathrm{P}(a_t|s_t)\, R_t$$

Policy gradient!

Deep Learning – New Frontiers

Universal Approximation Theorem – presented in 1989 states that – A neural network with a single hidden layer is sufficient to approximate any arbitrary function to any arbitrary precision/ accuracy. All it requires if a ***single layer***.

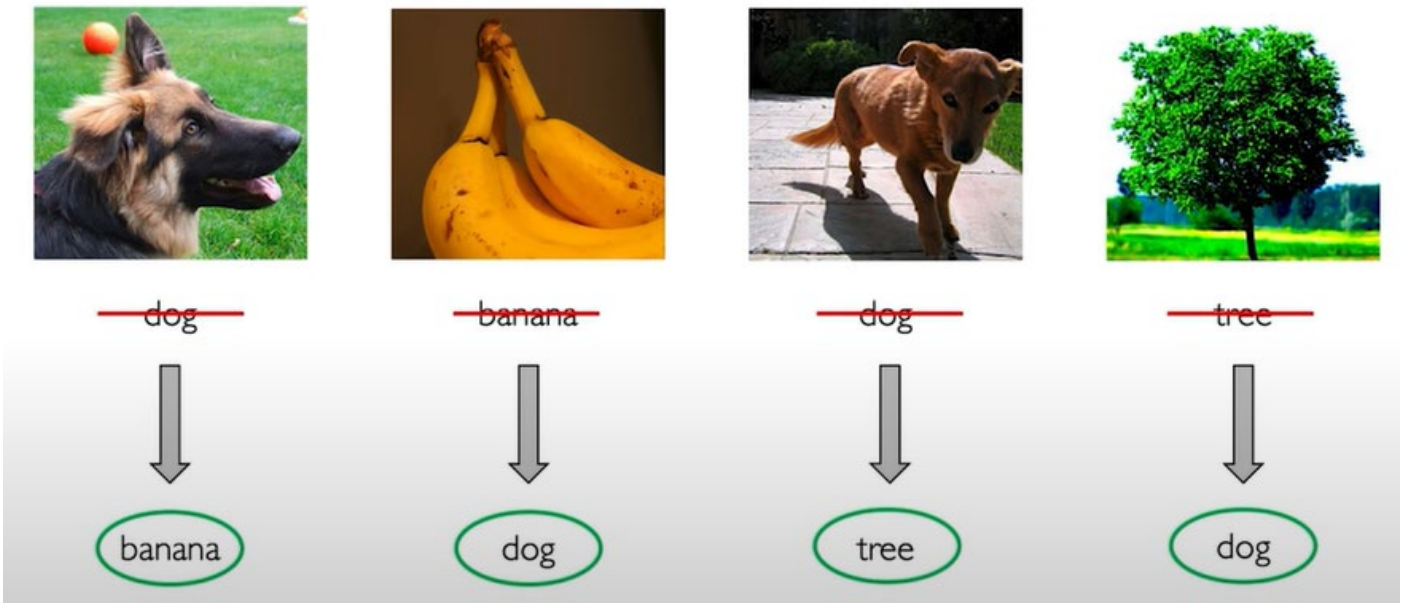
Limitations of the theorem

- The number of hidden units may be infeasibly large. Training such a model, finding appropriate weights can be computationally infeasible and is highly non-trivial.
- The resulting model may not generalize.

These limitations point towards a broader issue that relates to the possible effects of overhype in the AI community.

Limitations of Deep Neural Networks –

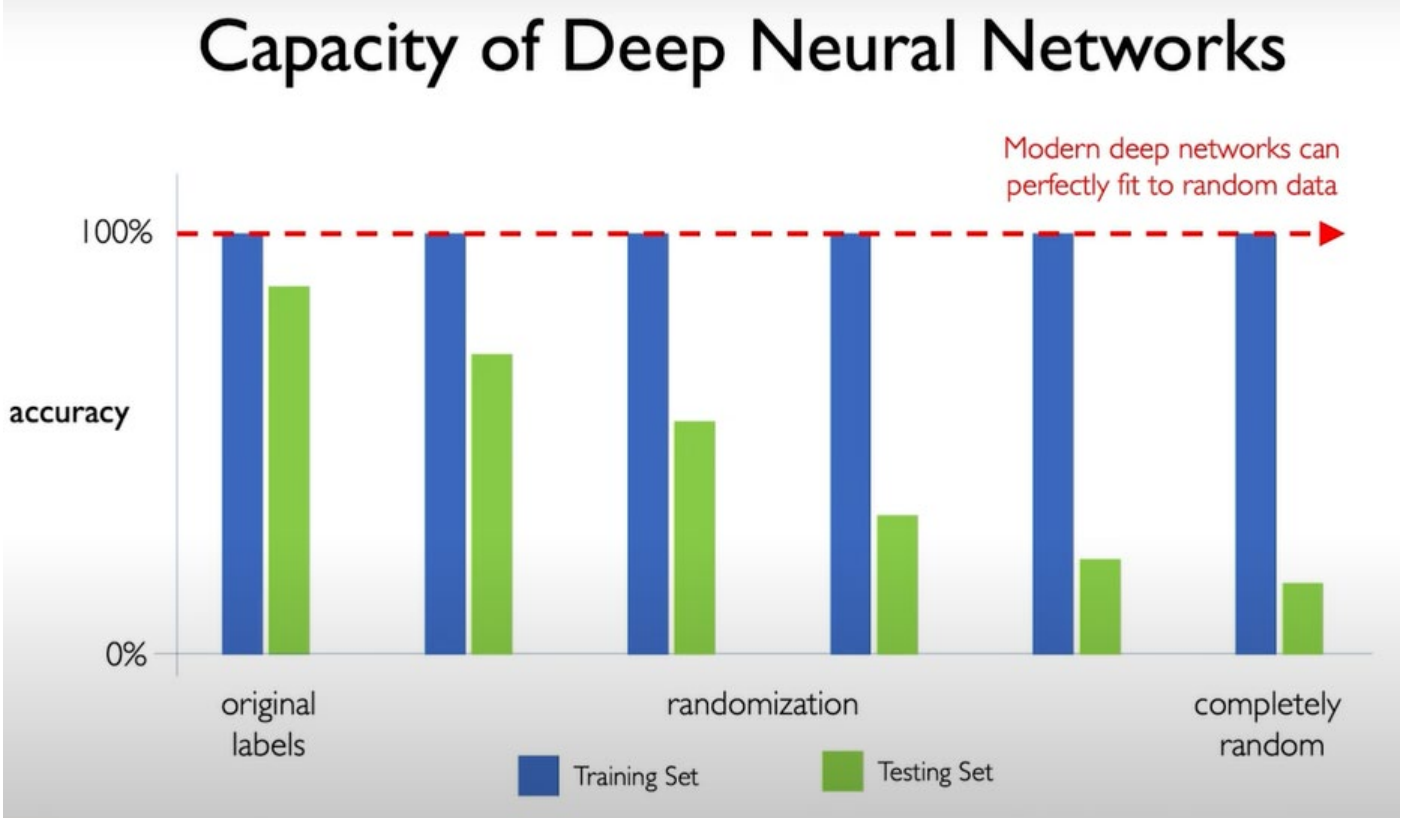
- Paper called "Understanding Deep Neural Networks Requires Rethinking Generalization".



They used pictures and labels to train a deep neural network to classify images. Then the researchers randomly shuffled the labels and re-trained the network.

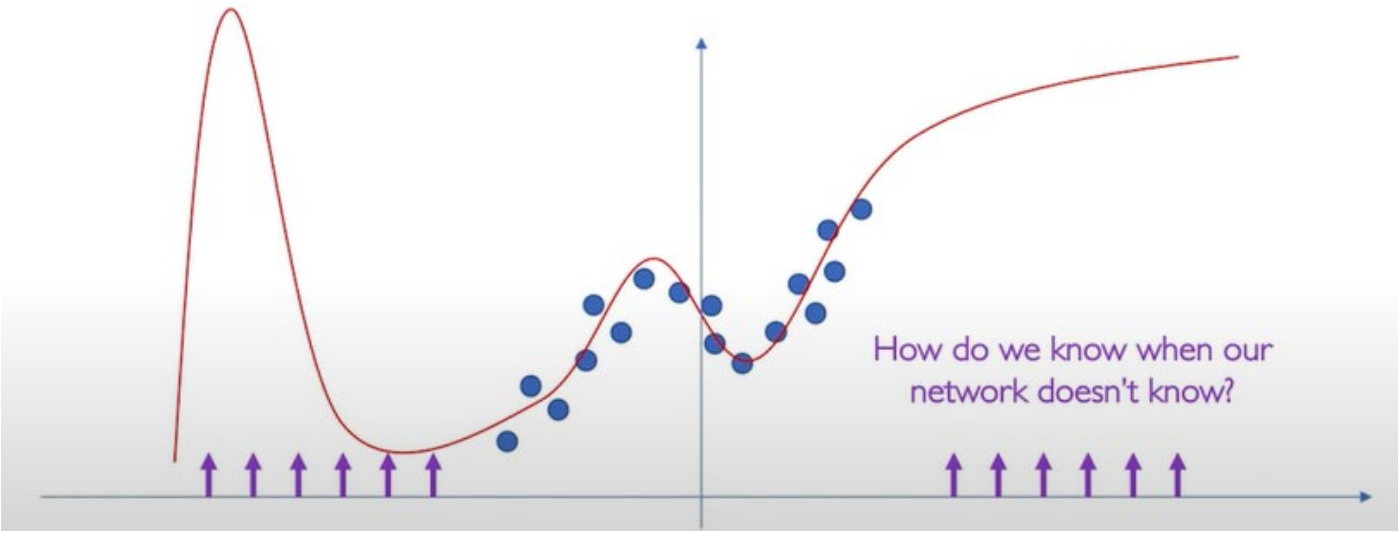
As expected the accuracy of the network on the test-set tends towards 0 as the randomness in the labels increased. But the researchers found that no matter how much the randomize the labels, the model was able to attain about 100% accuracy on the training set.

This tells us that Deep Neural Networks can perfectly fit to any function even if that function is associated with entirely random data driven by entirely random labelling.



Thus we can understand Neural Networks as excellent function approximators – even that's what the universal approximation theorem states.

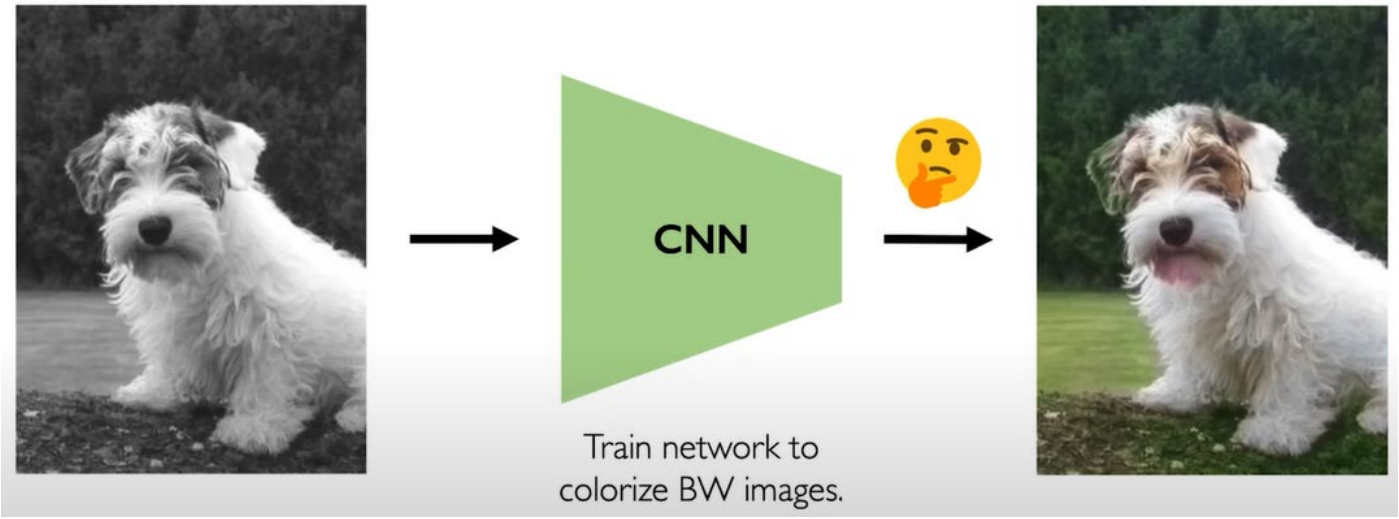
- Neural networks are excellent function approximators ONLY WHEN THEY HAVE TRAINING DATA.



Here, the network can work with training data (blue points) and end up with a good approximation. But it does not know where the function/approximation extends to when there is no training data. What happens in these regions where the network has not seen any training data? How do we know that the network is not confident in the predictions it is making?

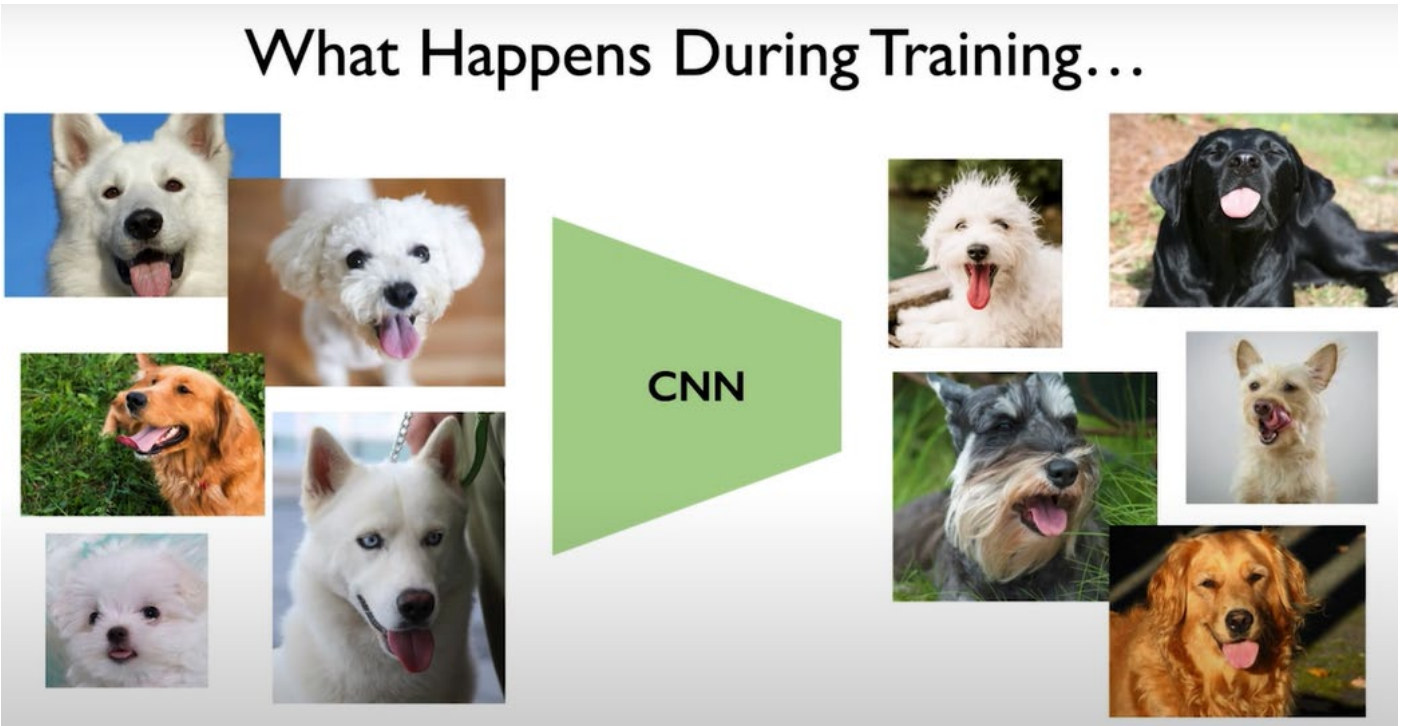
- Garbage In, Garbage Out

The neural network is only as good as your data. If you input correct data, you may end up with amazing results, but if the data is garbage, then the result will be the same.



The task of this Convolutional Neural Network was to colorize the black-and-white picture. If we look carefully, the area under the colorized dog-picture is pinkish, which does not make sense.

If we look under the hood at the training set, the input pictures it learns from might have a lot of dogs with the tongues out - which is quite natural for dogs. So the CNN might have mapped that region under the nose to being pink. So, when it saw a dog with closed mouth, it assumes in a way and maps that region to the color pink.



The neural networks build up representations based on the data they have seen. This is a critical point. The model will always be as good as the data.

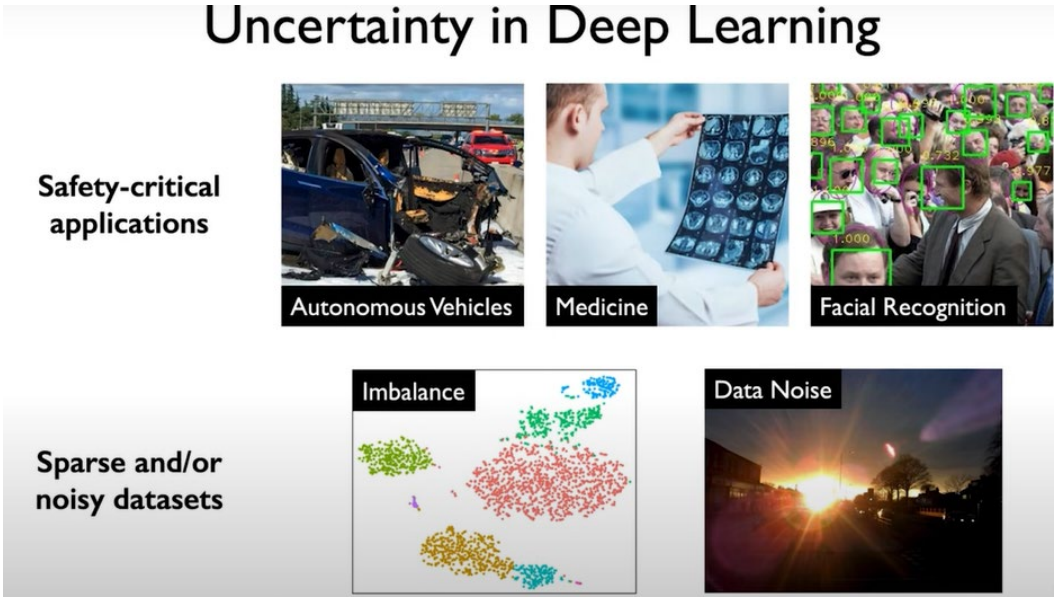
- How do neural networks handle data instances that they haven't encountered before?

This is the infamous Tesla crash case where the failure of neural networks lead to loss of human life. There was a new construction (modification of a barrier) which was not part of the training data on which the car's neural network was built upon. As a result, the car crashed into the barrier.

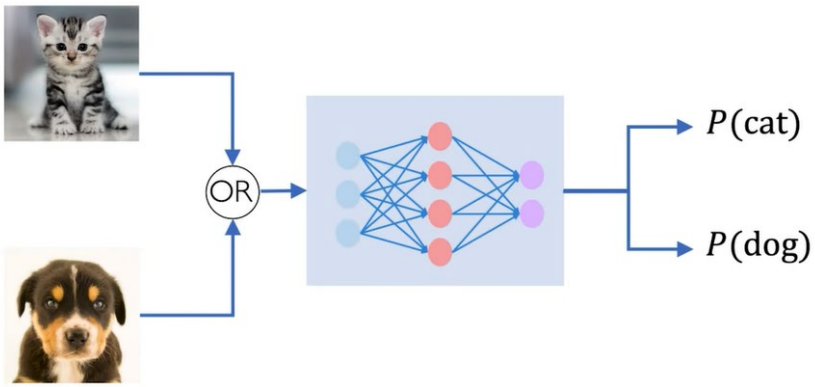
- Uncertainty in Deep Learning – when can we not trust the predictions of Deep Neural Networks?

This uncertainty is very crucial in deciding the fields of deployment of these techniques. Fields also called as safety-critical applications such as Autonomous vehicles, Medicine, Facial Recognition. As these algorithms are interacting more and more with human life, there needs to be a principled way to ensure there robustness.

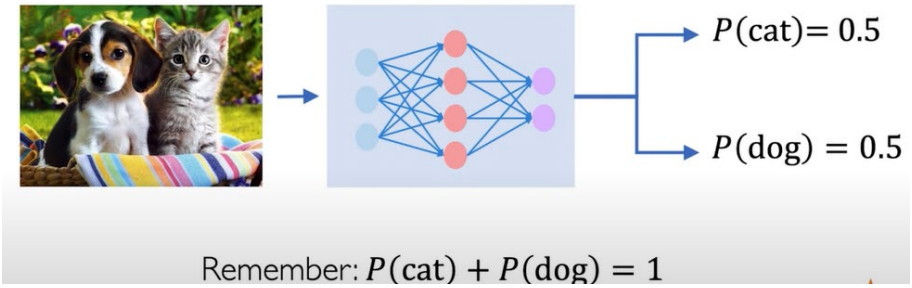
Uncertainty is almost important in cases where we have imbalanced or noisy data sets.



What uncertainties do we need?



- Let's consider this classification problem – build a NN to model probability over fixed set of classes {images of cats and dogs}.
- Remember, P(cat) + P(dog) = 1.
- What happens when we have an input image having both a cat and a dog.

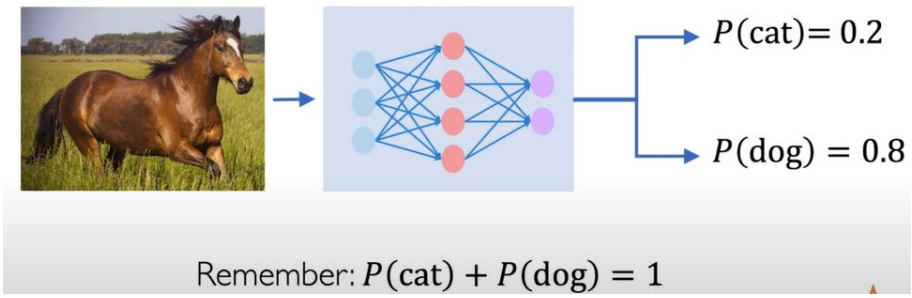


The network still has to output probabilities which sum to 1. But it has both a cat and a dog.

This problem can be referred to as there being Noise or Stochasticity present in the data. If the network is trained on images of cats and dogs alone, then a new instance of both is noisy with respect to what the model has seen before.

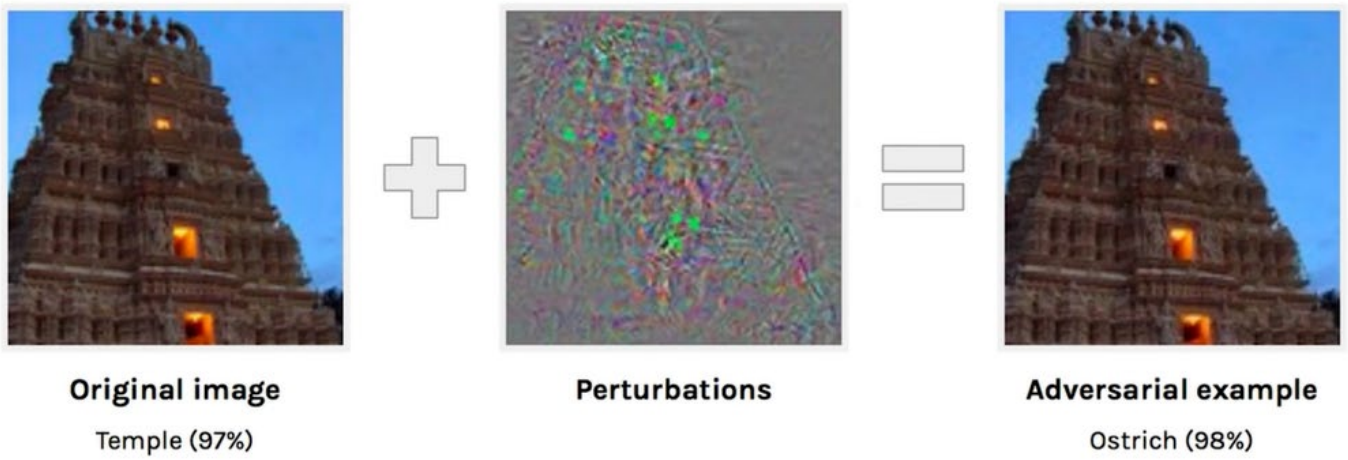
Uncertainty Metrics can help assess the statistical-noise that's inherent to the data. This is called Data-Uncertainty or ***Aleatoric Uncertainty***.

- Let's input the image of a horse in the network.



Again, the output probabilities have to sum to 1. But it is neither a cat or a dog. Here we are testing the model on an image that is absolutely out of distribution, an image of a horse. This is a different type of uncertainty – Model or ***Epistemic Uncertainty***.

- Adversarial Example



Suppose we input this image of a temple. A standard CNN may classify it as a temple with an accuracy of 97%. We then take the input an apply certain Perturbations to the image to generate an "adversarial example". Now this image fed to the same CNN – it no longer sees the image as a temple but classifies it as an ostrich! **What is this perturbation doing?**

- Remember when we train the neural networks using Gradient Descent – our task is to optimise some objective J (function of input X, prediction Y and weights W). In doing so, we are asking the question – ***how does a small change in the weights decrease our loss function?***

Remember:

We train our networks with gradient descent

$$W \leftarrow W - \eta \frac{\partial J(W, x, y)}{\partial W}$$

Fix your image x, and true label y

"How does a small change in weights decrease our loss"

-
- In order to do so, we train the network with a fixed image 'x' and a true label 'y' and perturbed/changed only the weights to minimise the loss.
- Under an Adversarial attack, we are asking – how can we modify the input image to increase the error.

Adversarial Image:

Modify image to increase error

$$x \leftarrow x + \eta \frac{\partial J(W, x, y)}{\partial x}$$

Fix your weights θ, and true label y

"How does a small change in the input increase our loss"

-

Non-exhaustive list of limitations of neural networks.

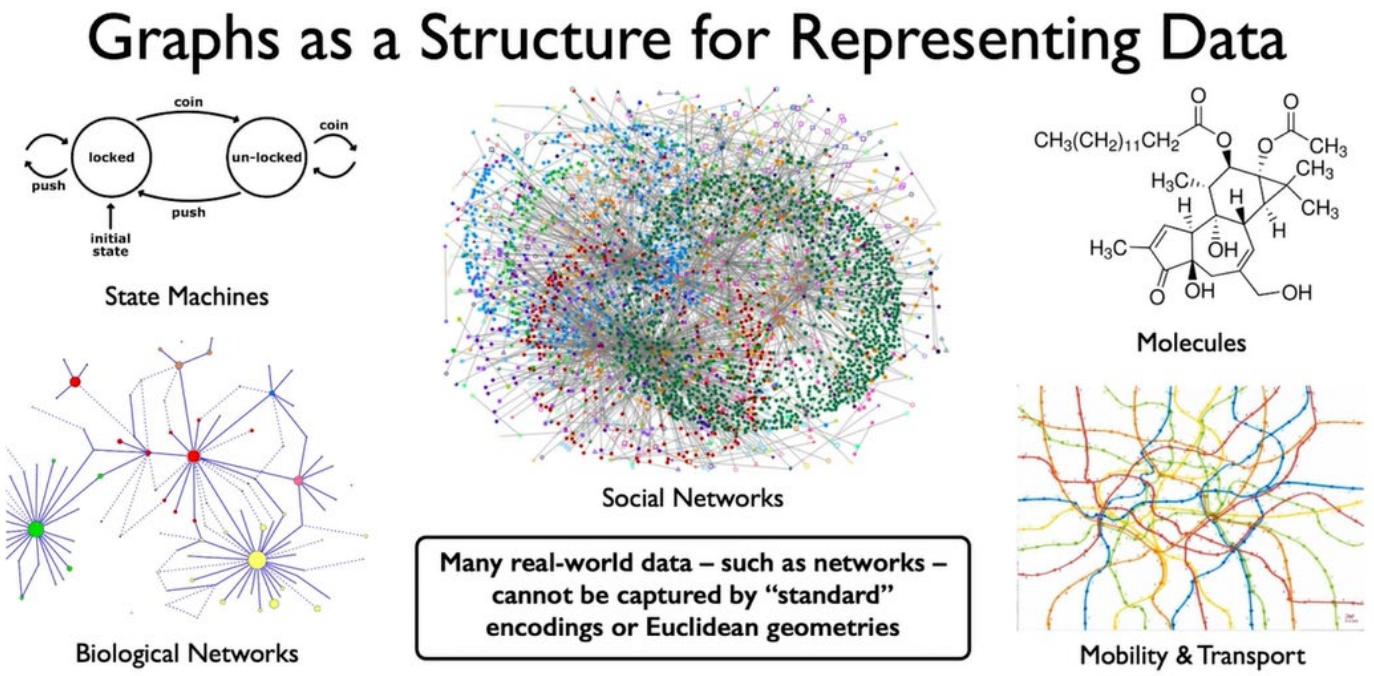
Neural Network Limitations…

- Very **data hungry** (eg. often millions of examples)
- **Computationally intensive** to train and deploy (tractably requires GPUs)
- Easily fooled by **adversarial examples**
- Can be subject to **algorithmic bias**
- Poor at **representing uncertainty** (how do you know what the model knows?)
- Uninterpretable **black boxes**, difficult to trust
- Difficult to **encode structure** and prior knowledge during learning
- **Finicky to optimize**: non-convex, choice of architecture, learning parameters
- Often require **expert knowledge** to design, fine tune architectures

New Frontiers of DNN

1. Encoding Structure And Domain Knowledge Into Deep Learning Algorithms

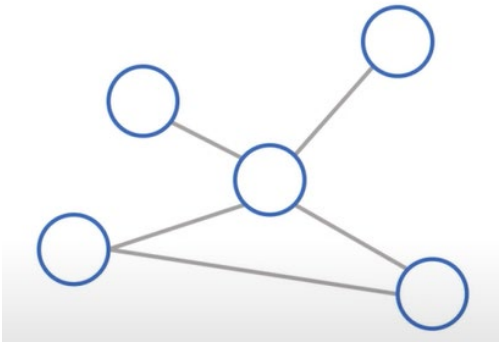
We see an example of this under CNN – inspired by human vision and brain – trying to capture spatial dependencies in the data.

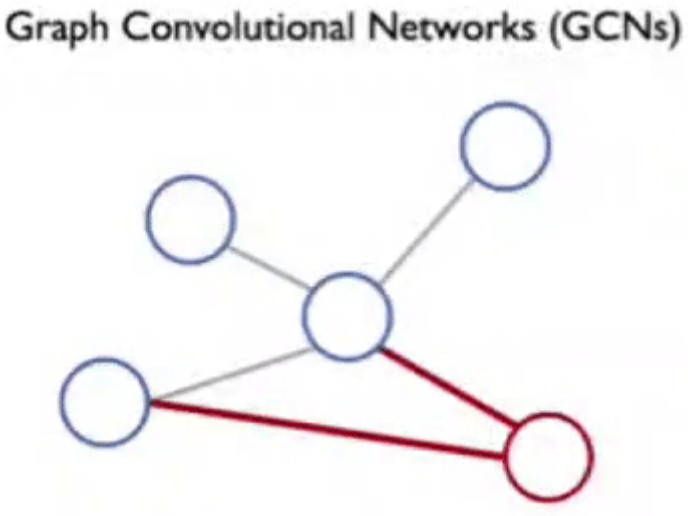


The graphs represent and contain a high-degree of rich and structural information which can be encoded in our networks. But it is unclear on how to include them. We can think of social

network, state machines, biological networks, molecule structures and transport-maps and so on have one commonality – their data cannot be captured by either an image or a temporal sequence. Thus graphs can provide a new and non-standard encoding for a series of problems.

Graph Convolution Networks (GCNs)

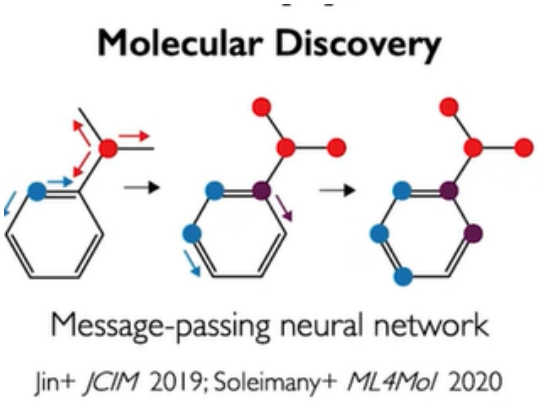
- They operate not on a 2D image but on a graph.
-  Nodes and Lines
- Lines represent relationship between nodes.
- We take a kernel/ weight matrix patch as under CNN. But instead of sliding the patch around, the patch will visit the different nodes of the graph. While it is on a node, it looks at the local-neighbourhood of the node and picks up on features relevant to the local connectivity of the node with the graph. This is the graph convolution operation.
- Here the network learns to define the weights associated with the nodes and connecting lines.

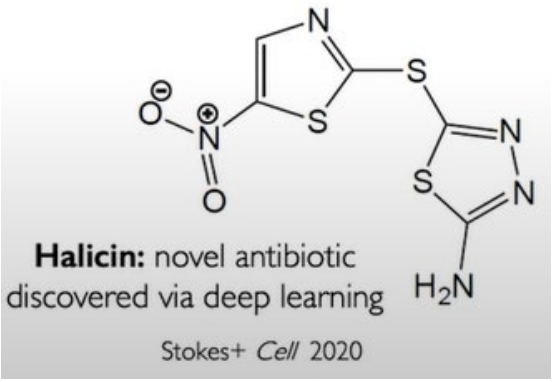


- Weight patch goes to different nodes. Looks at its emergent-neighbours. It associates weights for these connecting lines and applies these weights across the graph.
- Then it moves to next node in the graph – extracting information about its local connectivity. This continues to cover all nodes.
- The local information is going to be aggregated and the NN will learn a function which will encode the local information into a higher-level representation.

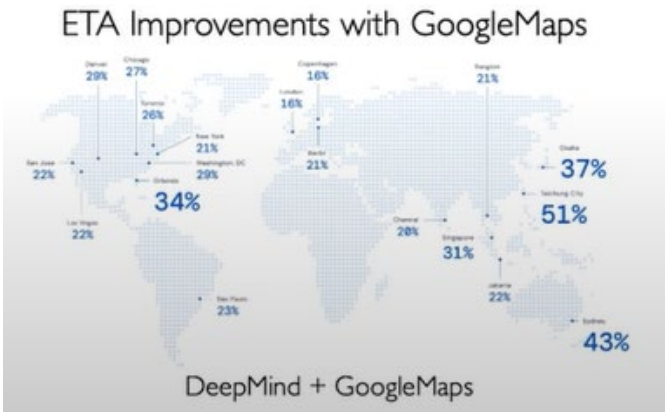
Applications

- Molecular Discovery → Message-passing networks build up a learnt representation of bonds present in chemical structures.

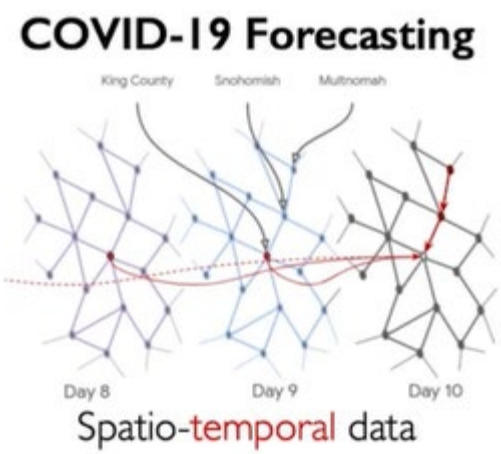


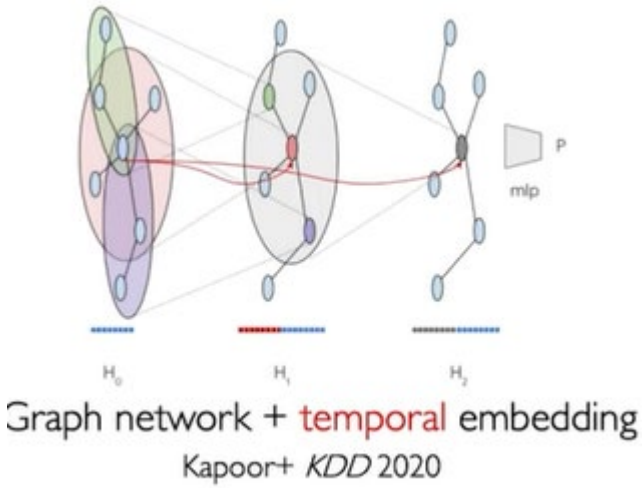


- Traffic Prediction → here we can take streets and represent them as nodes and model the intersections as a graph defining the connectivity. This helps in predicting traffic patterns resulting in improving ETA.

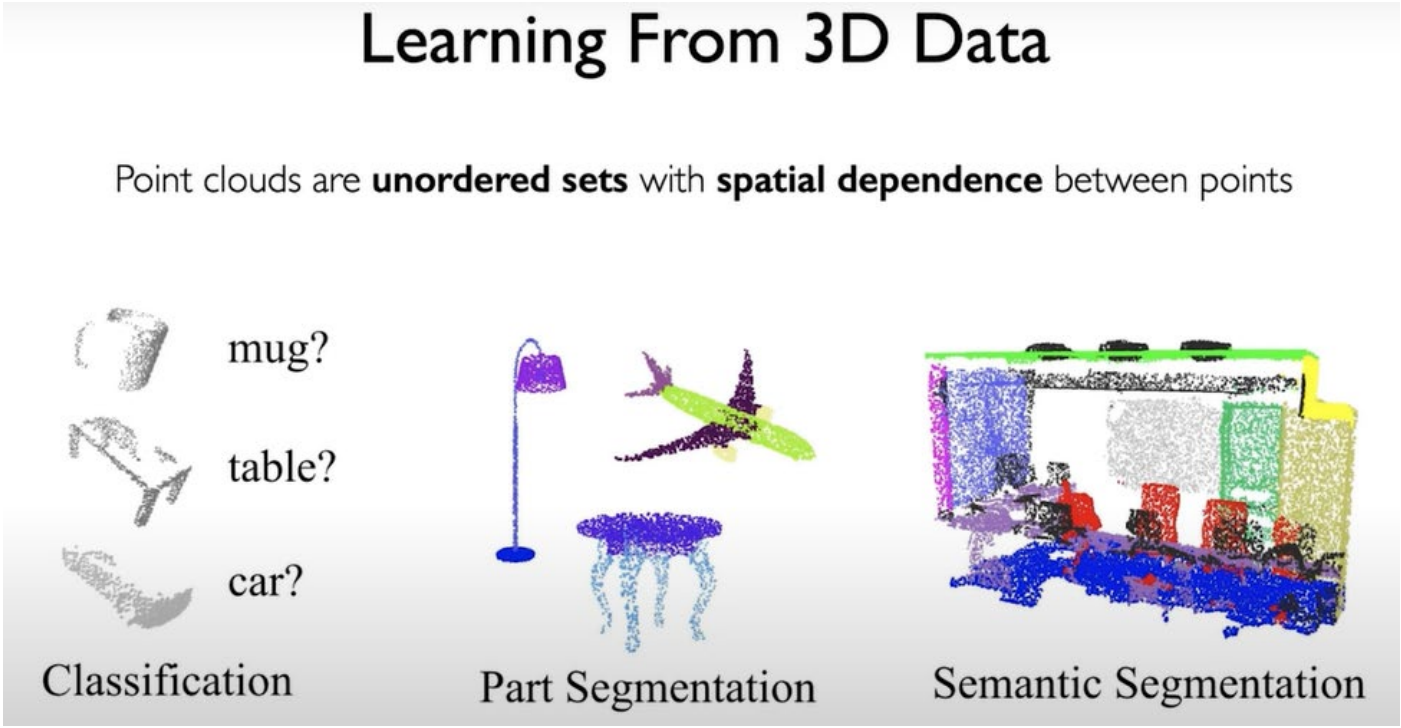


- Covid 19 forecasting – by incorporating geographic data (where a person is living, who they are connected to…) and temporal data (person's movement and trajectory over time) as inputs for GCNs.



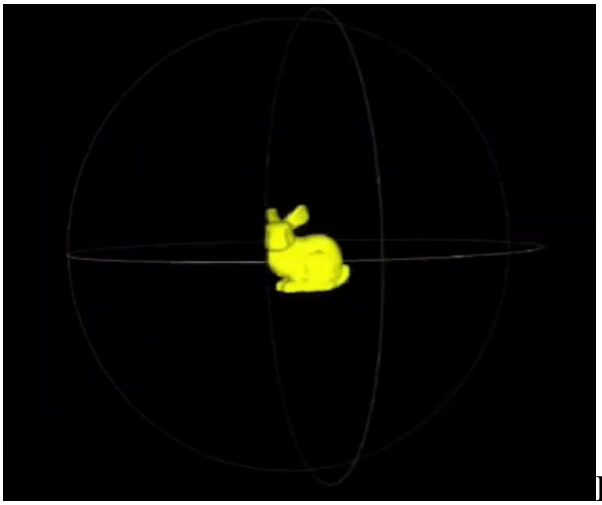


#3D Data Referred To As Point-Clouds



These cloud points are essentially unordered points in a 3D space where there is some underlying spatial dependence between them. We can thus take a point cloud for a classification problem or part/ semantic segmentation.

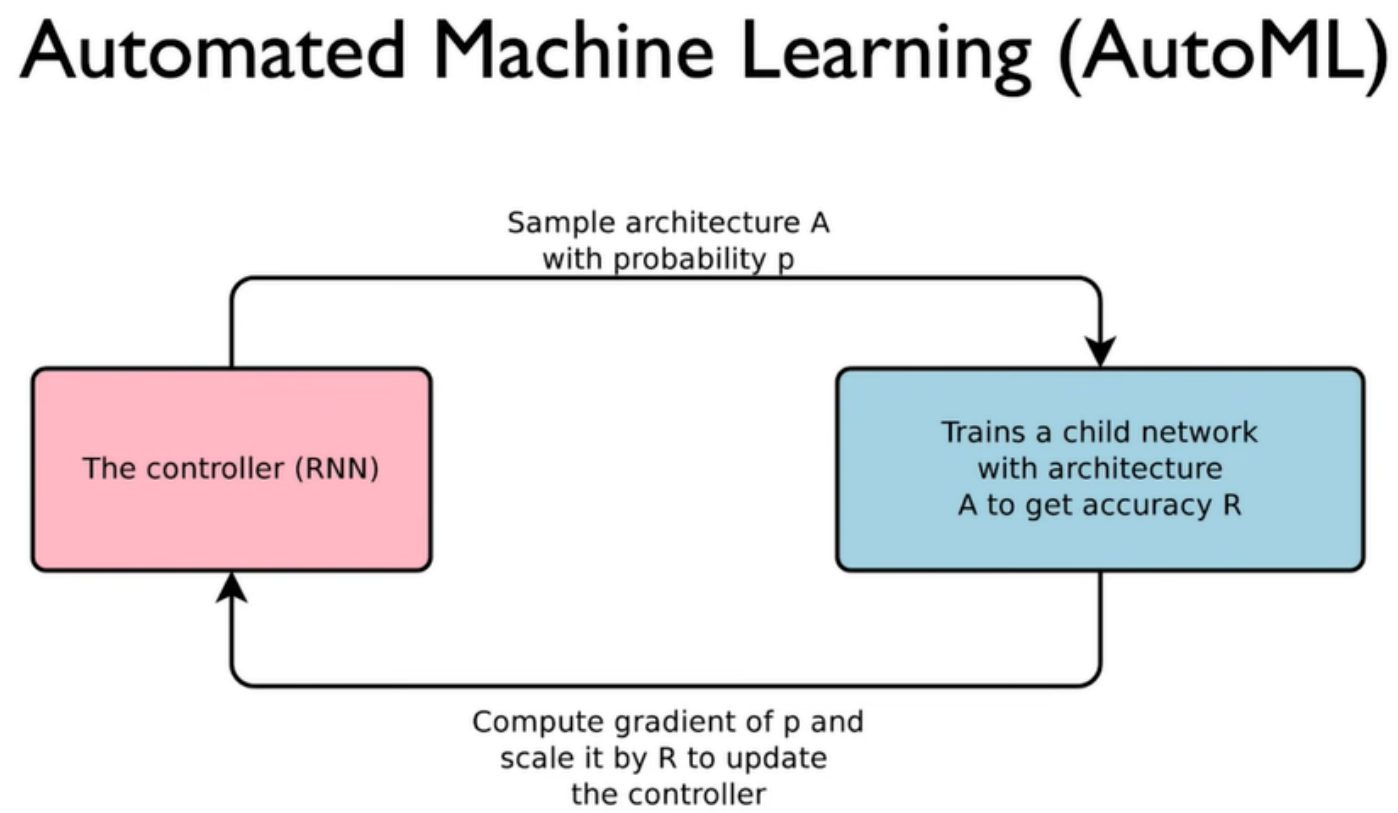
#Extend Graph CNNs To Point Clouds


How is it done – take a point cloud. Expand it out and dynamically compute a graph using the meshes (local connectivity) inherent in the point cloud. Then apply GCNs to maintain invariances about order of points in 3d space and simultaneously capture the local geometry of the data system.

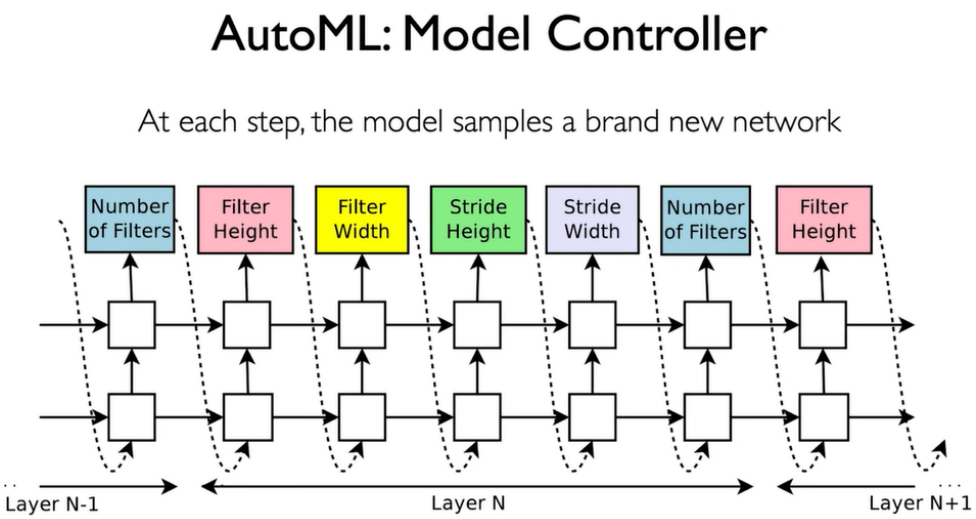
Automated Machine Learning

Standard deep neural network architectures require domain-expertise i.e. expert knowledge to build a neural network for a given task. The idea behind Automated ML is that building a learning algorithm that learns which model (which neural network architecture) is optimal/best for a given problem. Not only

the structure or type of neural network is optimised but also the hyperparameter values and number of hidden layers and nodes too.



- The idea is based on Reinforcement Learning
- There is a 'controller network' which is a Recurrent NN. It proposes a sample model architecture – called the child architecture. And this architecture is defined by a set of hyperparameters
- The accuracy of the child architecture is used as a REWARD in this reinforcement learning framework to promote the controller to improve the network proposals in the next round of optimisation.
- This cycle repeats 1000s of times generating new architecture – testing them – giving feedback to controller. Eventually, architectures with higher accuracy will be assigned higher probability and vice-versa in the search-space.



How does this controller work?

- At each iteration, the controller models a brand new architecture. This controller network is optimised to predict the hyper-parameters associated with the child-architecture.