

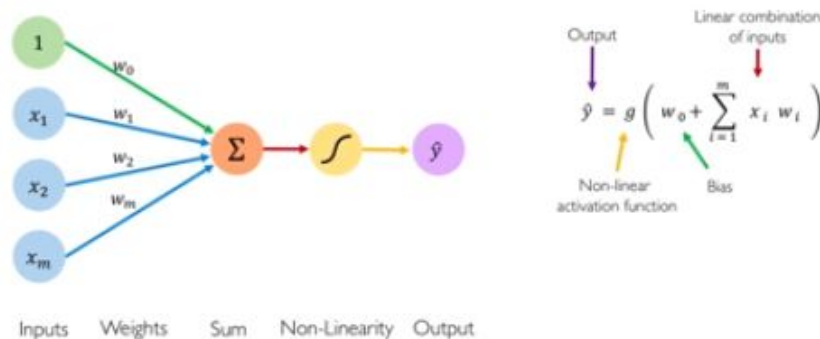
INTRO TO DEEP LEARNING

Learning from the things are working later ~ learning features from underlying features.

Big Data -> Hardware -> Software

Perceptron :

The Perceptron: Forward Propagation



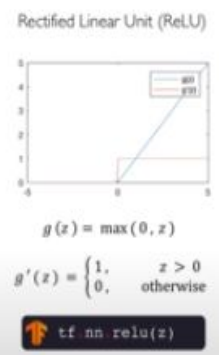
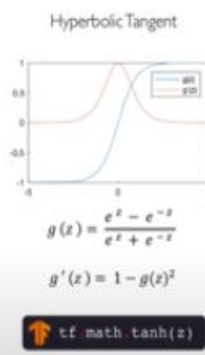
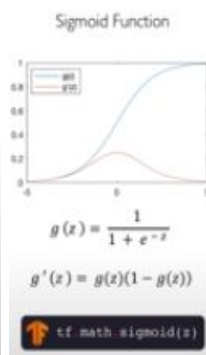
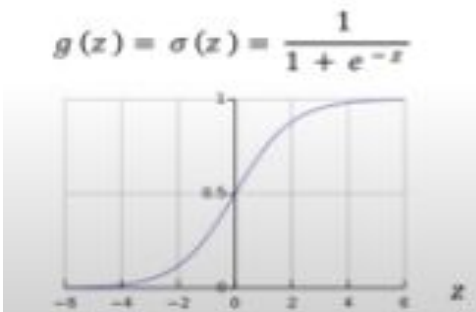
Activation Function :

Takes a real number in x axis and converts it into a number in 0 and 1.

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

Common Activation Functions

- Example: sigmoid function



Purpose of AF is - to introduce non-linearities in functions.

Hidden layers , dense layer

Quantifying Loss : measures the cost incurred from incorrect predictions.

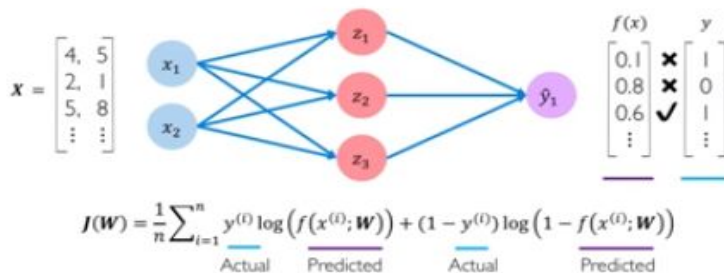
$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss : total loss over the entire dataset.

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Binary Cross Entropy Loss

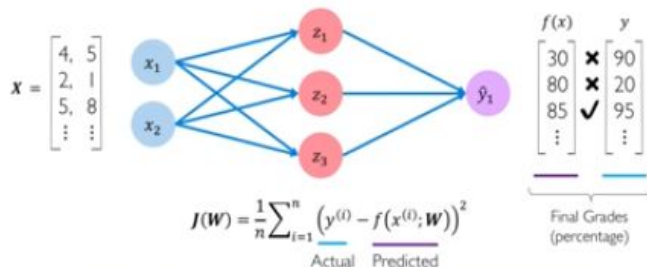
Cross entropy loss can be used with models that output a probability between 0 and 1



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

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
```

Loss Optimisation : finding the network weights that achieve the lowest loss through maybe 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

```
import tensorflow as tf

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

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

    weights = weights - lr * gradient
```

Backpropagation : chain rule ~

Optimization through gradient descent

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

How can we set the learning rate?

Adaptive Learning Rates ~

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
 - how large gradient is
 - how fast learning is happening
 - size of particular weights
 - etc...

Gradient Descent Algorithms

Algorithm	TF Implementation	Refe
• SGD	 <code>tf.keras.optimizers.SGD</code>	Kiefer & Wolfowitz, "Sto Maximum of a Regressi
• Adam	 <code>tf.keras.optimizers.Adam</code>	Kingma et al. "Adam: A l Optimization," 2014.
• Adadelta	 <code>tf.keras.optimizers.Adadelta</code>	Zeiler et al. "ADADELTA Method," 2012.
• Adagrad	 <code>tf.keras.optimizers.Adagrad</code>	Duchi et al. "Adaptive S Learning and Stochastic
• RMSProp	 <code>tf.keras.optimizers.RMSProp</code>	

```

import tensorflow as tf
model = tf.keras.Sequential([...])
# pick your favorite optimizer
optimizer = tf.keras.optimizers.SGD()
while True: # loop forever
    # forward pass through the network
    prediction = model(x)
    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)
    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Can replace TensorFlow

Stochastic gradient descent ~ means using a single point.

Using a single point for gradient descent is very noisy and using all the data points is very computationally expensive, therefore the solution is ~

Using mini batches of points ~ and true is calculated by taking the average, therefore smoother convergence and allows greater learning rates, enables parallelism therefore faster.

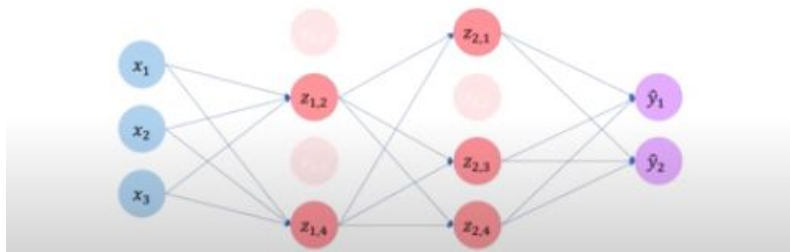
Overfitting exists so to deal with it ~

Regularisation ~ technique that constraints our optimisation problem to discourage complex models ~ improve generalisation.

Regularization I: Dropout

- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node

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



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



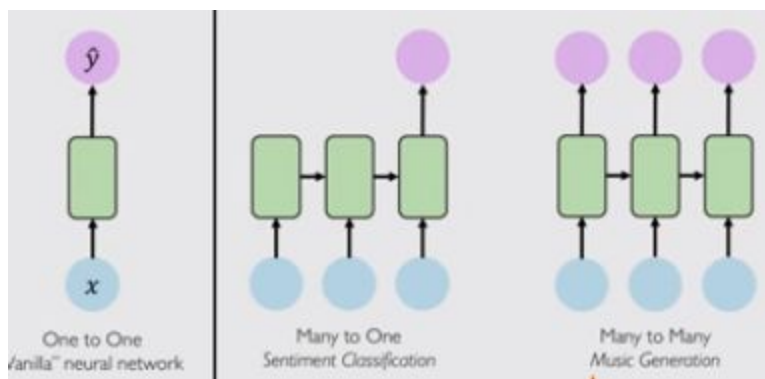
RECURRENT NEURAL NETWORKS

Sequential Modelling : predict the next thing that will happen after a previous states are given

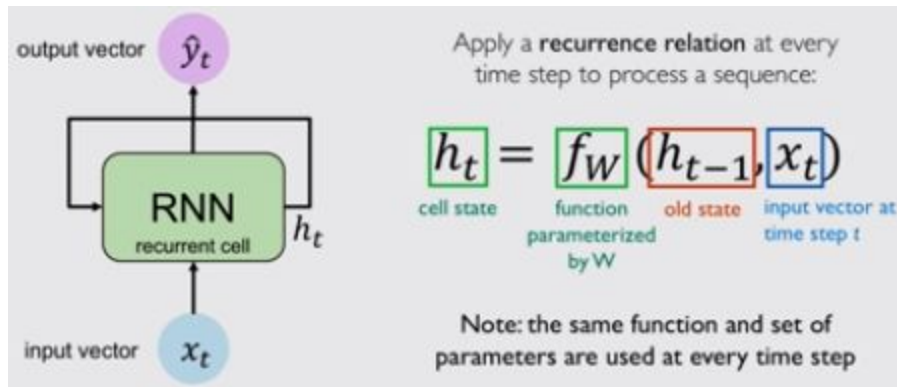
If word prediction then if using BagOfWords then the order is not reserved therefore Separate parameters should be used to encode the sentence.

In standard Feed Forward -> it goes one to one passing of data

RNN for Sequence Modeling :



RNNs have a loop in them that enables them to maintain an internal state , not like a vanilla NN where the final output is only present.



Output Vector

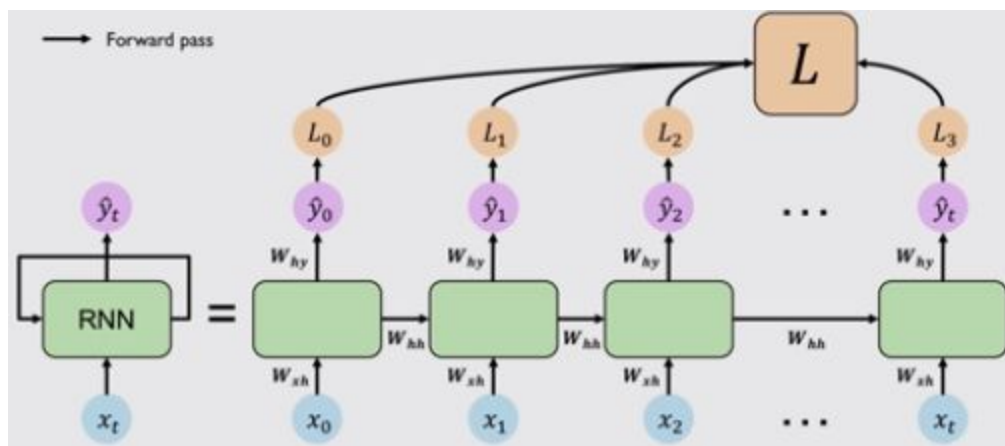
$$\hat{y}_t = W_{hy}^T h_t$$

Update Hidden State


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

Input Vector

$$x_t$$

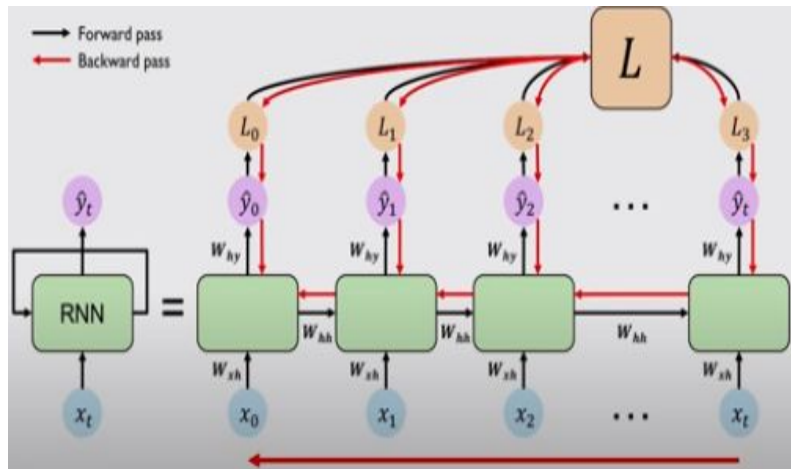


```
tf.keras.layers.SimpleRNN(rnn_units)
```

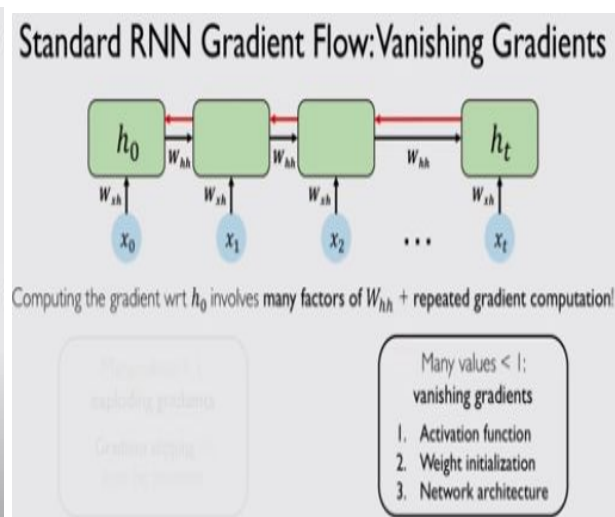
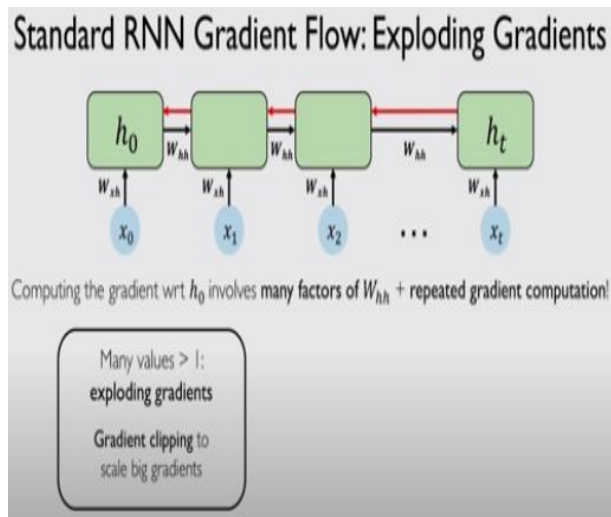


Backpropagation through time :

In RNNs at each time backprop is done also in total ~



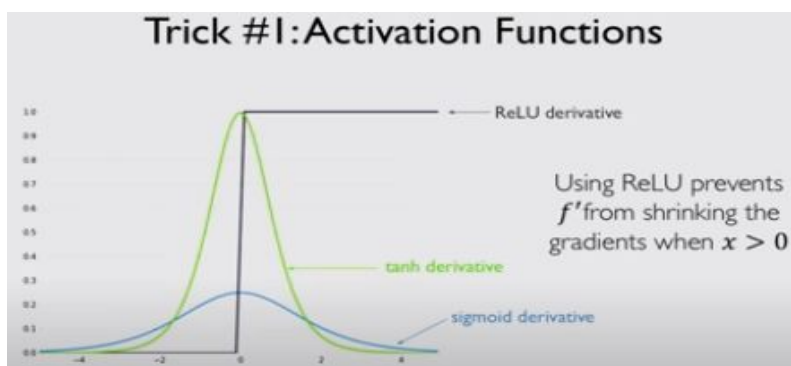
Standard RNN Gradient Flow can have 2 problems :



The problem of LongTermDependencies (vanishing gradient) ~

Multiply many small numbers together -> errors which are in further back time steps keep having smaller and smaller gradients -> this biases es parameters to capture dependencies in models , thus standard RNNs becoming less capable.

To solve the above problem :



Both tanh and sigmoid have derivatives less than 1.
Therefore use RELU but $x > 0$ only then.

Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

Solution #3: Gated Cells

Idea: use a more **complex recurrent unit with gates** to control what information is passed through

gated cell

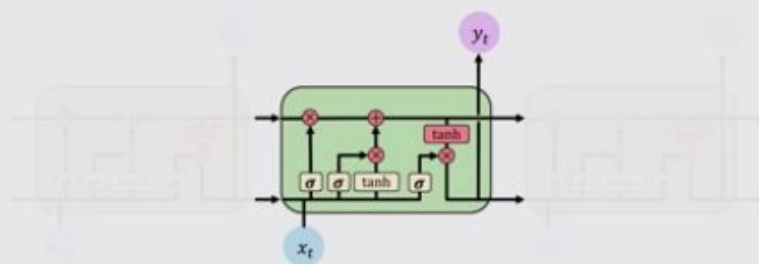
LSTM, GRU, etc.

Long Short Term Memory (LSTMs) networks rely on a gated cell to track information throughout many time steps.


The above is well suited for learning tasks.

Long Short Term Memory (LSTM) Networks :

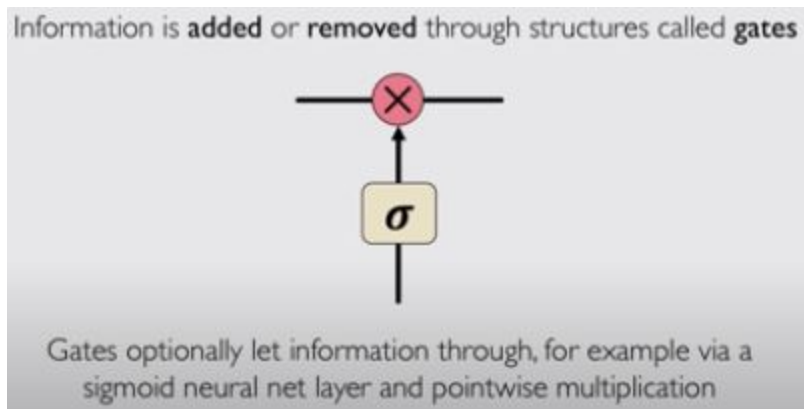
LSTM modules contain **computational blocks** that **control information flow**



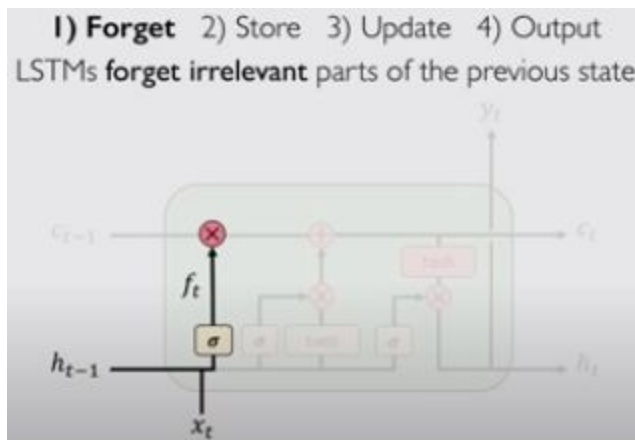
LSTM cells are able to track information throughout many timesteps

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

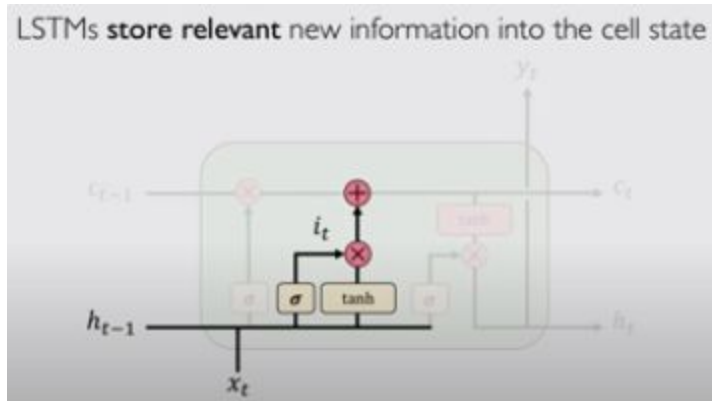
Gates are an important part in LSTMs



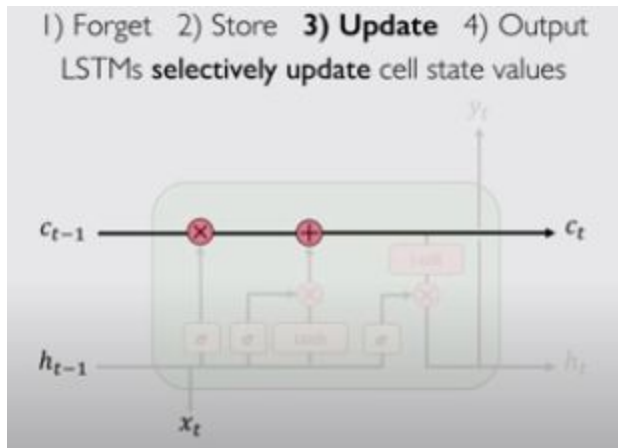
LSTMs work in 4 steps :



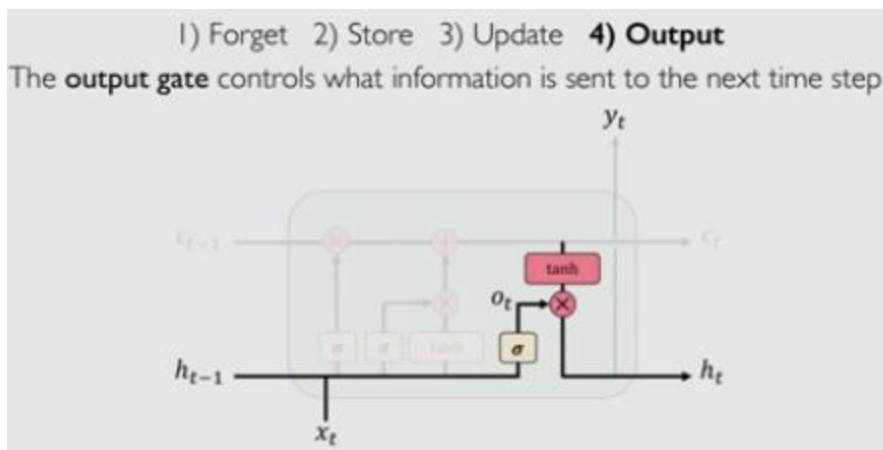
1.



2.



3.



4.

All these allow LSTMs to have an uninterrupted gradient flow.

RNN Applications :

Example :

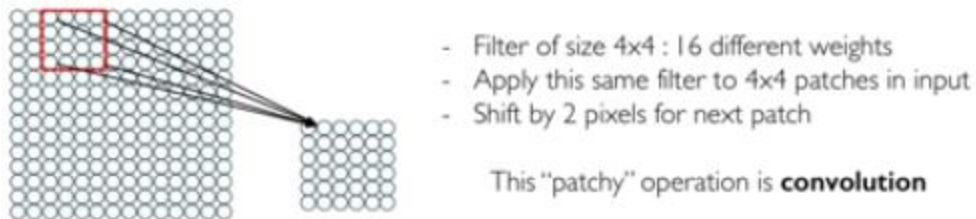
1. Music Generation ~ at each level the next tune is predicted.
2. Sentiment Classification ~
3. Machine Translation ~ encoding bottleneck is a problem so attention mechanism is used ~ all states of time steps are accessed and trained upon individually.
4. Trajectory Prediction ~ Self Driving Car
5. Environmental Modelling

CONVOLUTION NEURAL NETWORKS

Spatial features need to be created in case of IMAGE data.

Using Spatial features :

Feature Extraction with Convolution



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

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

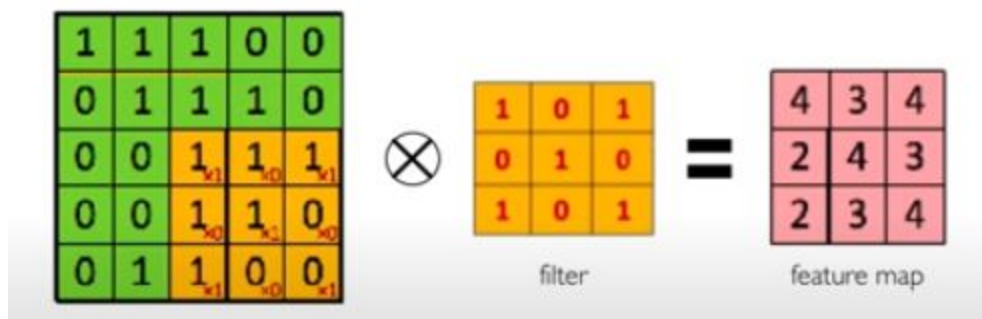
3) **Spatially share** parameters of each filter

Deformation should be handled.

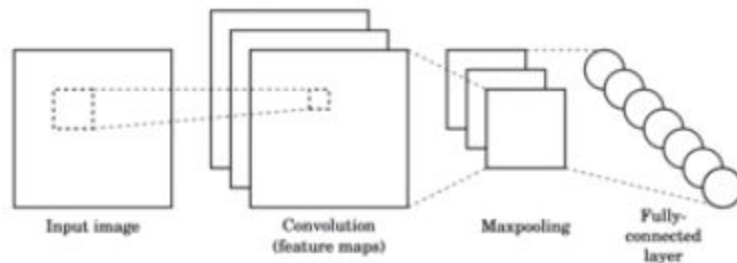
Convolution preserves the features , multiply the features of the patch and the same size patch its being compared to -> then if the whole thing is outcoming 1 then -> its an exact match.

The Convolution Operation


We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



CNNs for Classification




1. **Convolution:** Apply filters to generate feature maps.

 `tf.keras.layers.Conv2D`

2. **Non-linearity:** Often ReLU.

 `tf.keras.activations.*`

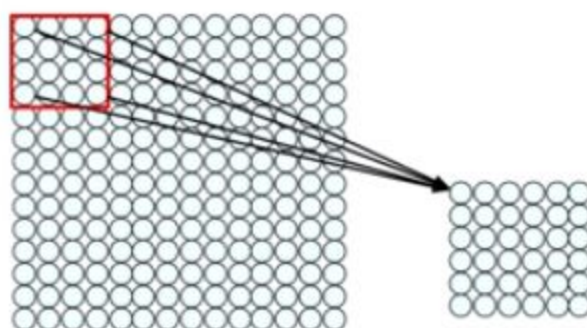
3. **Pooling:** Downsampling operation on each feature map.

 `tf.keras.layers.MaxPool2D`

Train model with image data.
Learn weights of filters in convolutional layers.

Computation of class scores can be outputted as a dense layer representing the probability of each class.

Convolutional Layers: Local Connectivity



 `tf.keras.layers.Conv2D`

For a neuron in hidden layer:

- Take inputs from patch
- Compute weighted sum
- Apply bias

4x4 filter: matrix
of weights w_{ij}

$$\sum_{i=1}^4 \sum_{j=1}^4 w_{ij} x_{i+p,j+q} + b$$

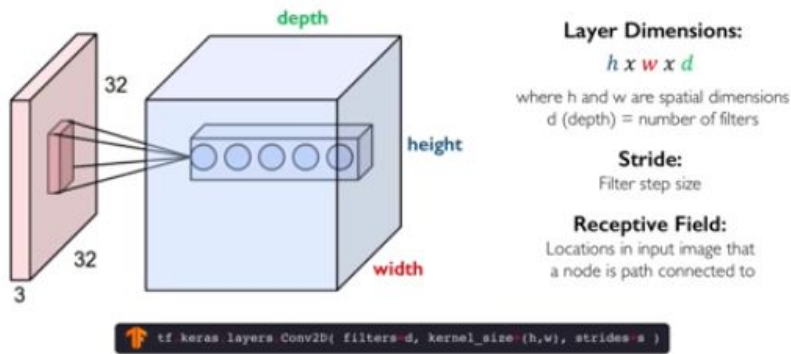
for neuron (p,q) in hidden layer

- 1) applying a window of weights
- 2) computing linear combinations
- 3) activating with non-linear function

1.

We have to define how many features to detect.

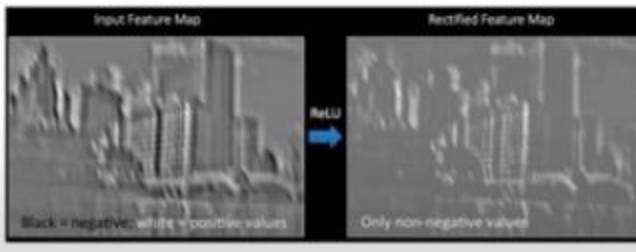
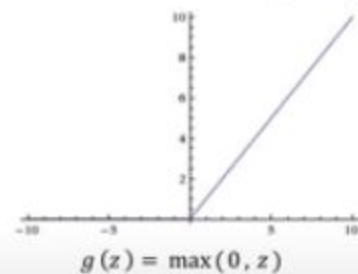
CNNs: Spatial Arrangement of Output Volume



Introducing Non-Linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**

Rectified Linear Unit (ReLU)



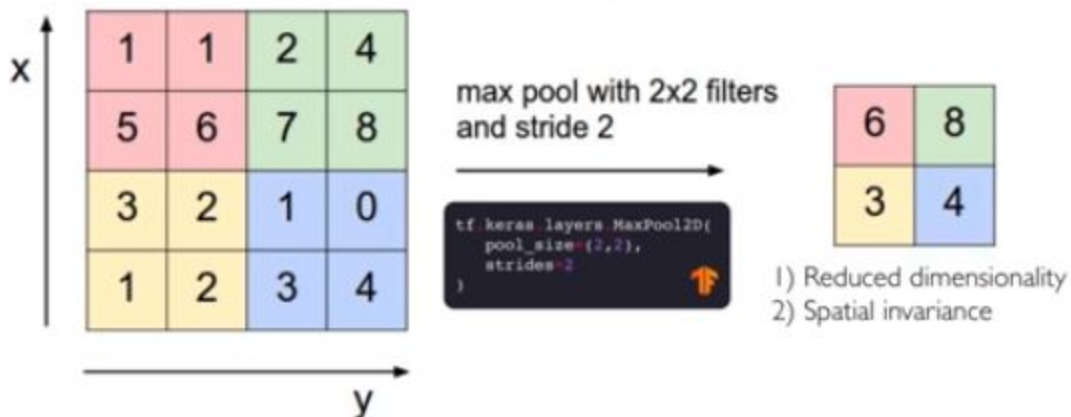
```
tf.keras.layers.ReLU
```

2.

ReLU takes input any real number and it shifts any number less than zero to zero and greater than zero it keeps the same. (min of everything is 0)

3. Downsampling using Pooling :

Pooling

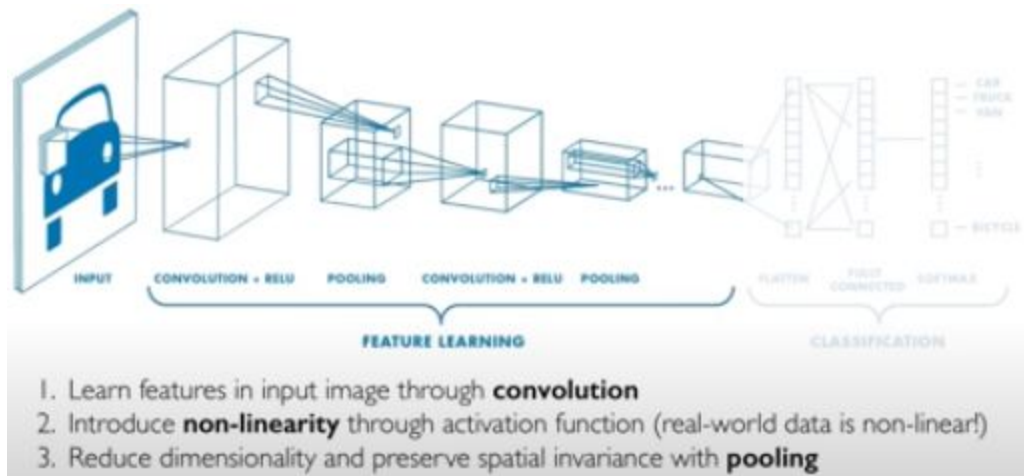


How else can we downsample and preserve spatial invariance?

There are other ways too, to downsample other than MAX-pooling.

In DEEP CNNs we can stack layers to bring out the features, low -> mid -> high

Part 1: Feature Learning -> extract and learn the features.



PART 2: Classification

CNNs for Classification: Class Probabilities



- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Putting it all together

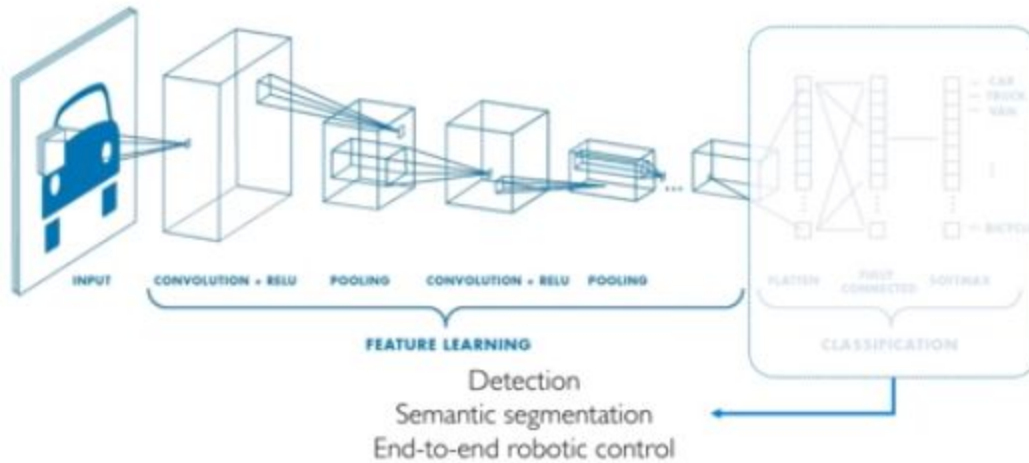
```
import tensorflow as tf

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

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

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

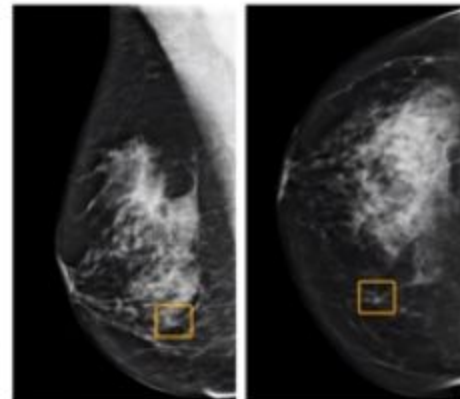
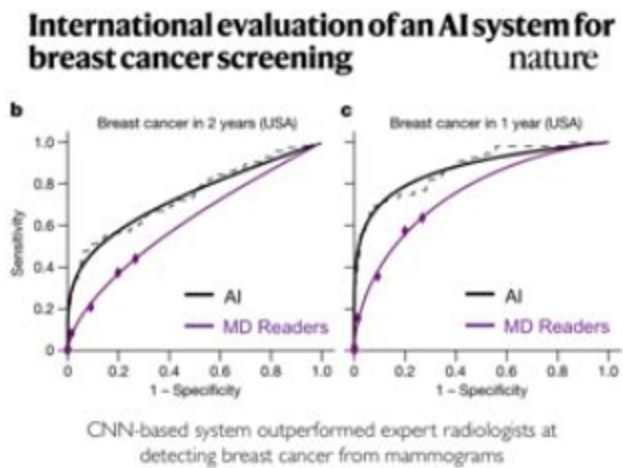

An Architecture for Many Applications



The first half can remain the same, but the end can be altered to fit.

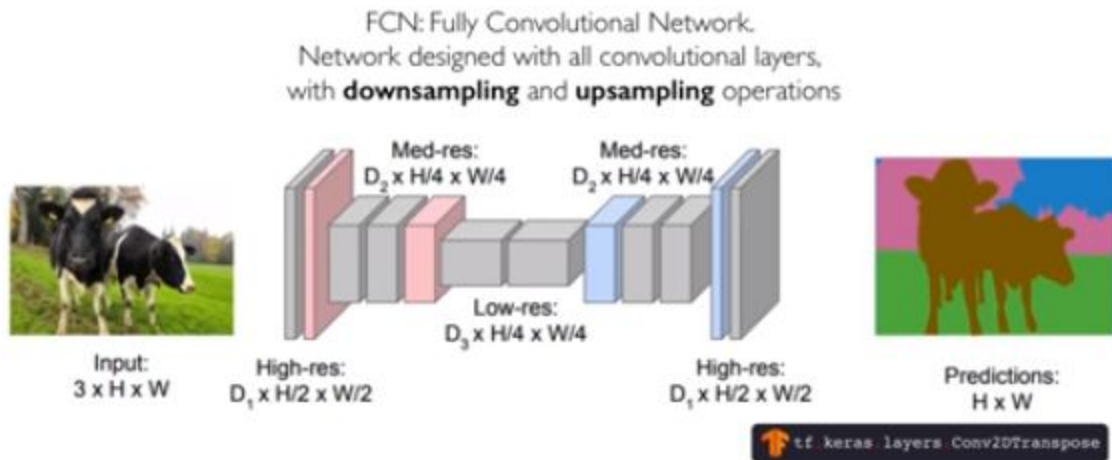
EXAMPLES :

Detection: Breast Cancer Screening



Breast cancer case missed by radiologist but detected by AI

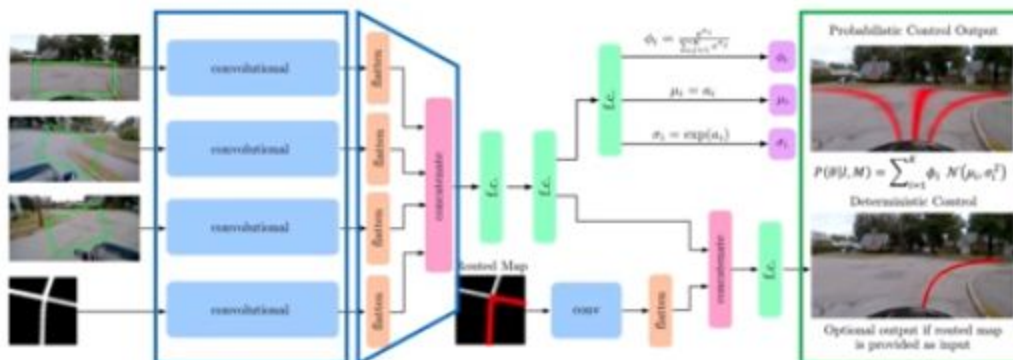
Semantic Segmentation: Fully Convolutional Networks



1. The picture is fed into a convolution feature extractor then its decoded / upscaled.
2. Class of the features are found out.
3. Upsampling is performed using transpose convolution

End-to-End Framework for Autonomous Navigation

Entire model is trained end-to-end **without any human labelling or annotations**



This whole thing is end to end , nothing was told explicitly.

DEEP GENERATIVE MODELING

Supervised Learning

Data: (x, y)
 x is data, y is label

Goal: Learn function to map
 $x \rightarrow y$

Examples: Classification,
regression, object detection,
semantic segmentation, etc.

Unsupervised Learning

Data: x
 x is data, no labels!

Goal: Learn the *hidden* or
underlying structure of the data

Examples: Clustering, feature or
dimensionality reduction, etc.

Generative Modeling:

Goal: Take as input training samples from some distribution
and learn a model that represents that distribution

Density Estimation



Sample Generation



Input samples

Generated samples

Training data $\sim P_{data}(x)$

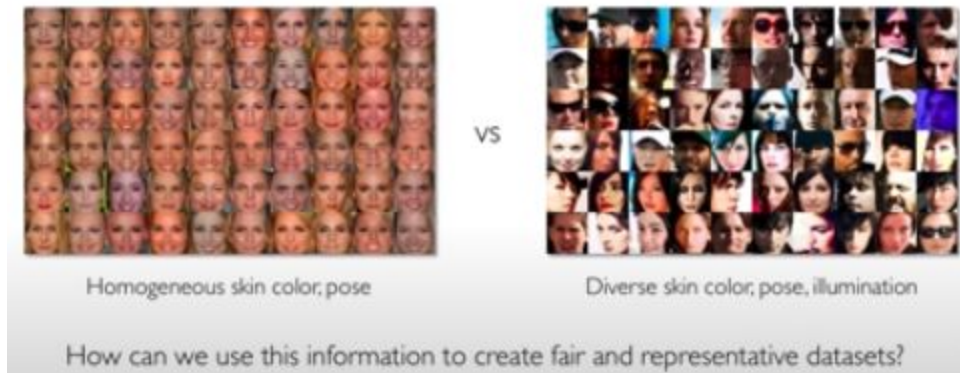
Generated $\sim P_{model}(x)$

How can we model some probability distribution that's similar to the true distribution.

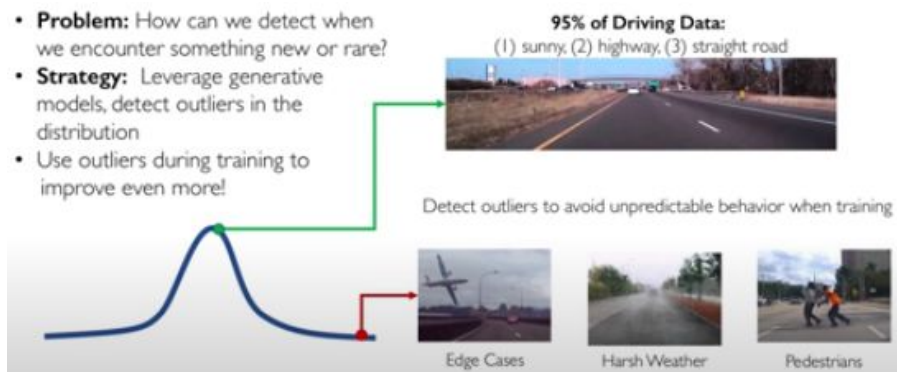
Generative models :

Debiasing

Capability to uncover the underlying features.



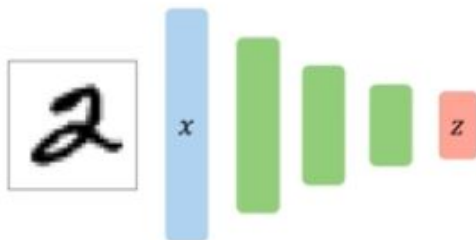
Outlier Detection



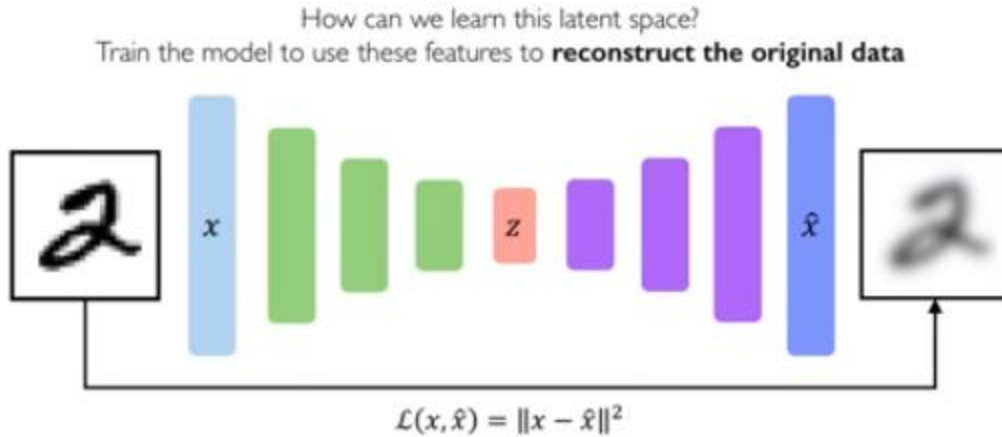
Latent Variable : they are not directly visible but are the things that actual matter.

Autoencoders :

Unsupervised approach for learning a **lower-dimensional** feature representation from unlabeled training data



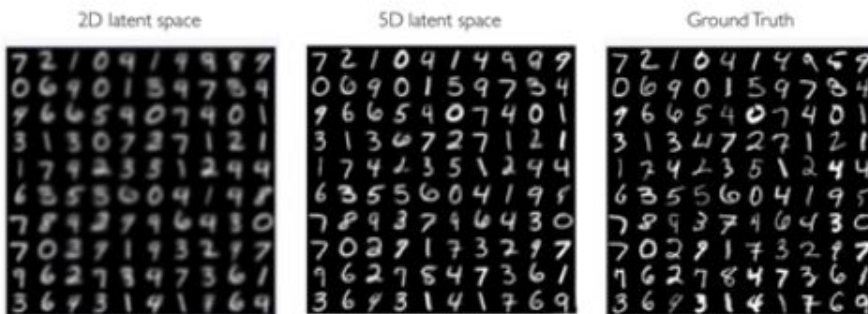
"Encoder" learns mapping from the data, x , to a low-dimensional latent space, z



This loss function doesn't have any labels.

Dimensionality of latent space → reconstruction quality

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



Lower the dimensionality lower is the quality of output.

Autoencoders for representation learning

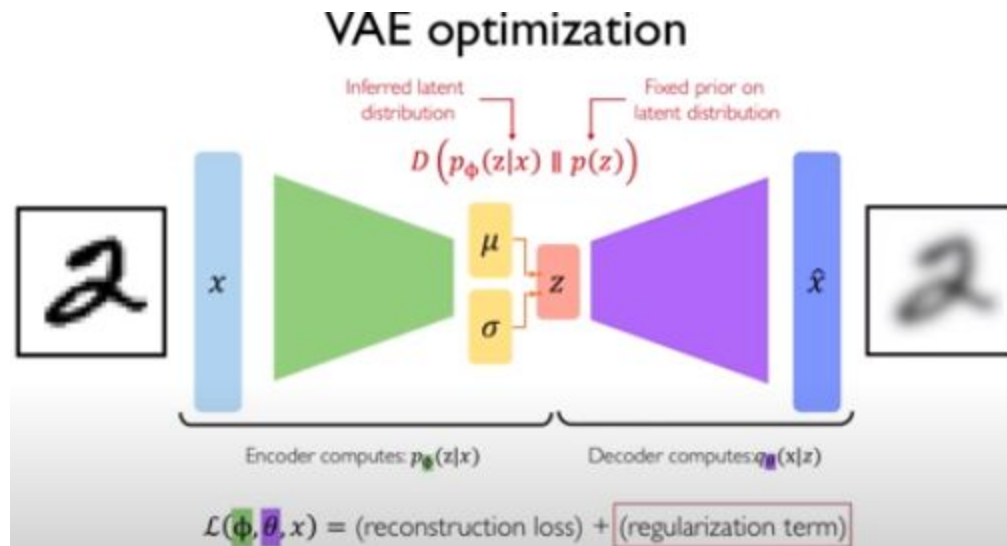
Bottleneck hidden layer forces network to learn a compressed latent representation

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

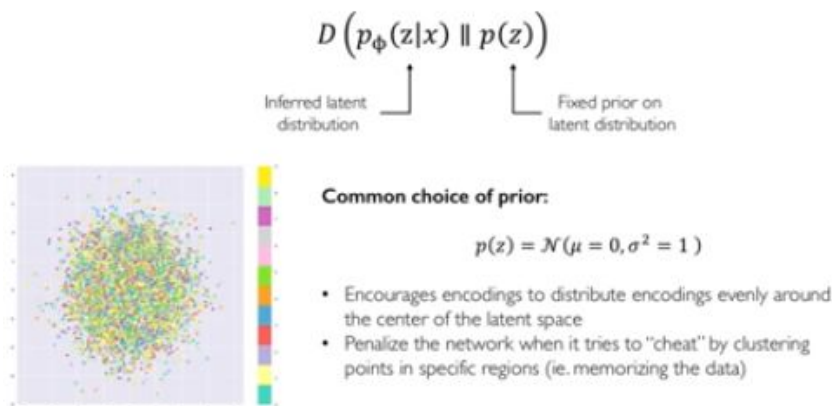
Autoencoding = **Automatically encoding** data

Variational Auto Encoders :

Instead of a deterministic layer there is a stochastic sampling operation , i.e for each learn a mean and sigma (deviations) that represent the probability distribution.



Priors on the latent distribution



Regularisation term that is used to formulate the total loss. The Gaussian term.

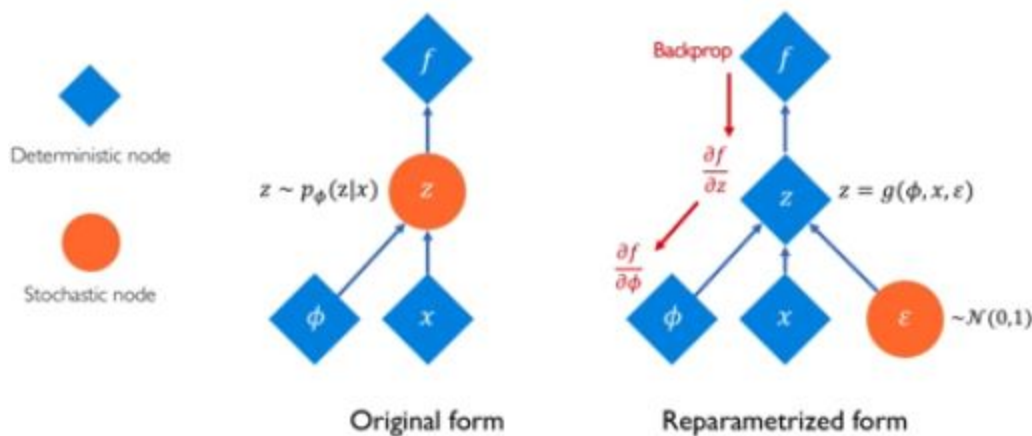
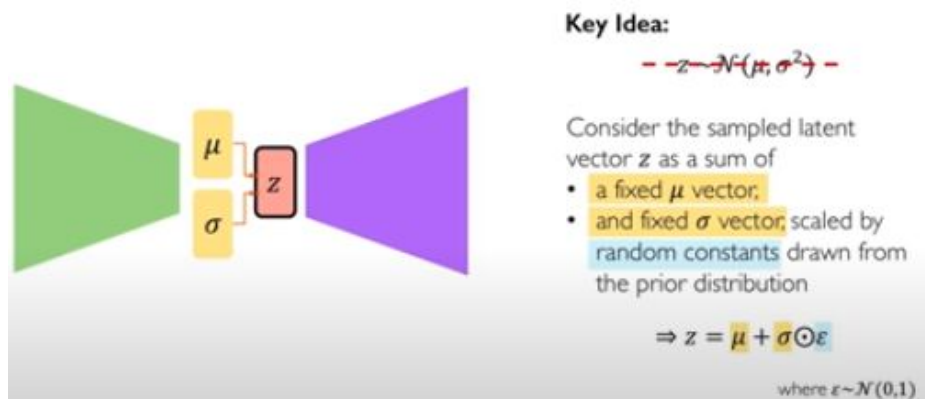
$$D(p_\phi(z|x) \parallel p(z)) = -\frac{1}{2} \sum_{j=0}^{k-1} (\sigma_j + \mu_j^2 - 1 - \log \sigma_j)$$

KL-divergence between the two distributions

We cannot backpropagate gradients through a sampling layer, due to their stochastic nature as bp requires chain rule.

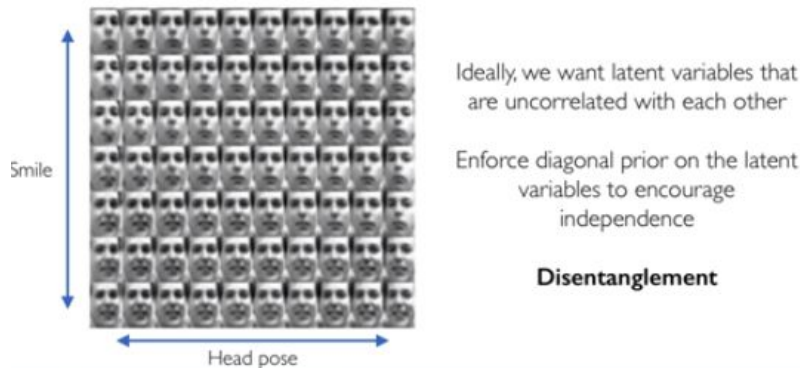
So we can Reparameterize the sampling layer.

Reparameterizing the sampling layer



Slow tuning of the latent variable (increase or decrease), then run the decoder to get the output. Therefore it forms a semantic meaning.

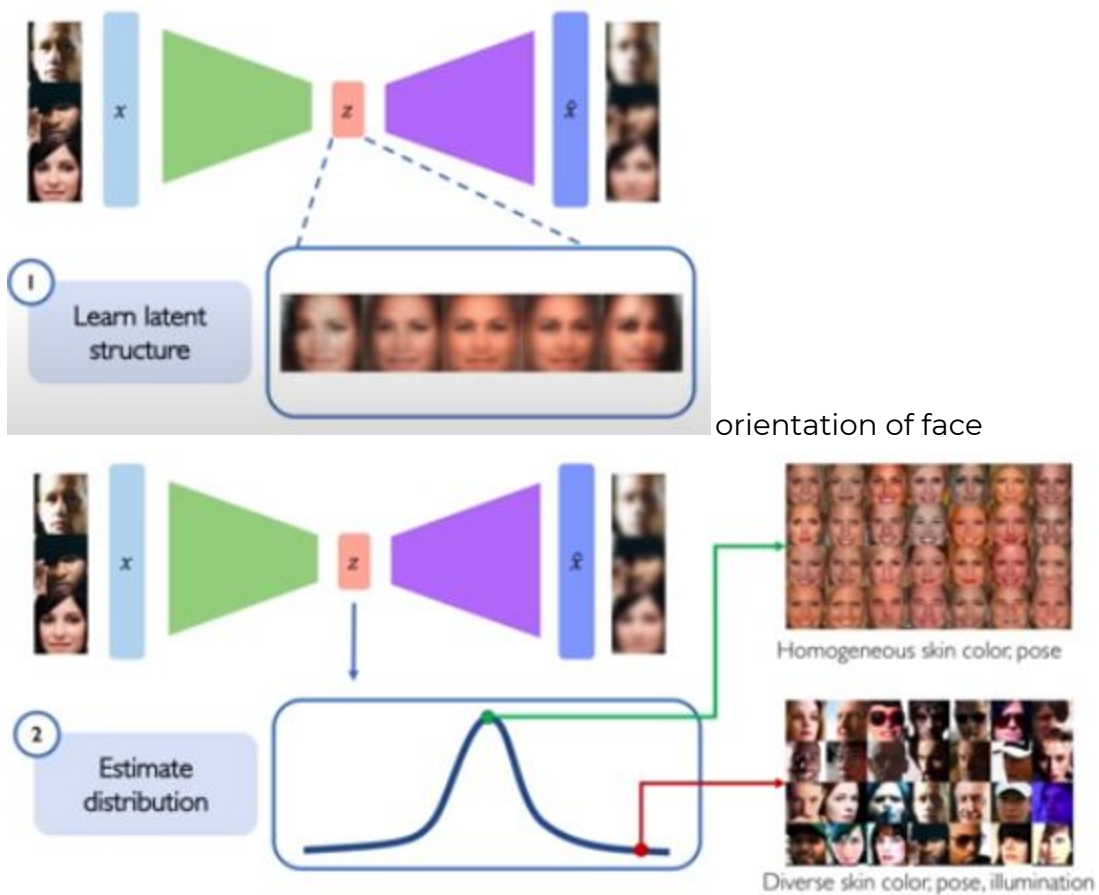
By perturbing the value of a single latent variable we actually get to know what they mean and represent.



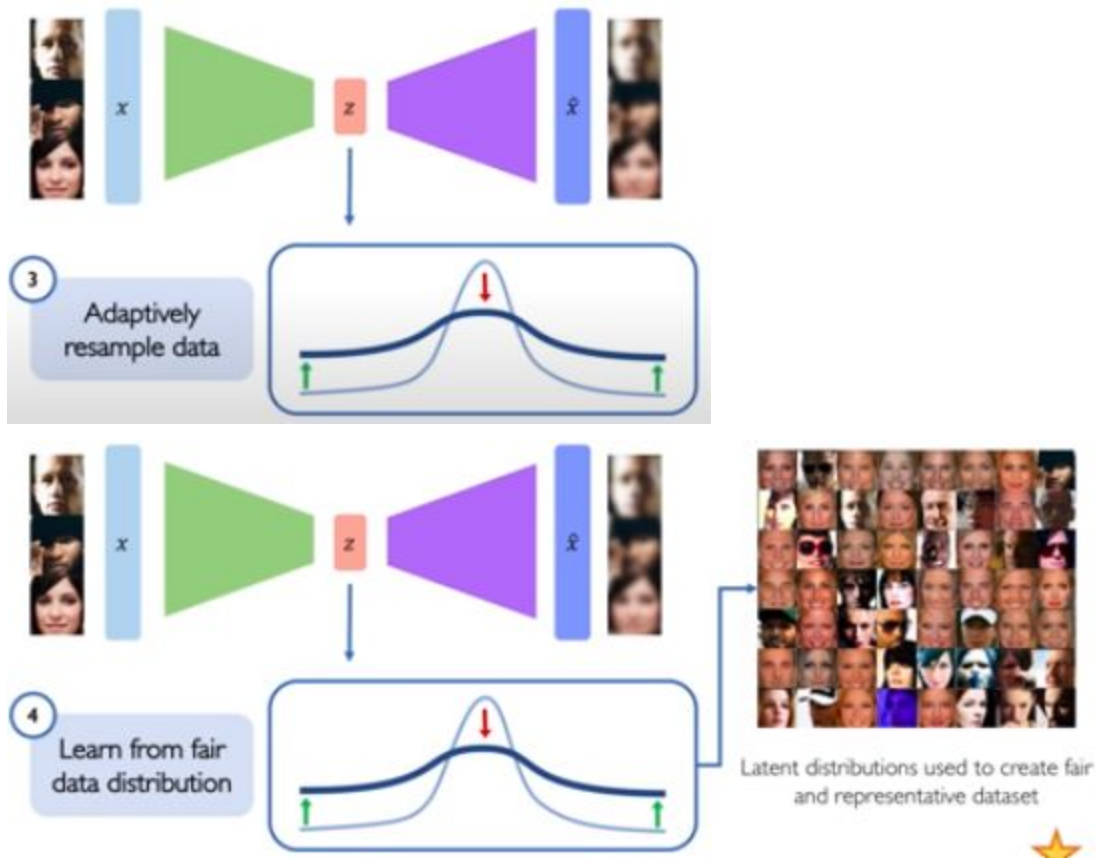
All other variables are fixed other and then these are fine tuned accordingly , Disentanglement lets the model learn features not being correlated to one another.

We can use biases to get these.

Mitigating bias through learnt latent structures :



These may be under-represented.



VAEs:

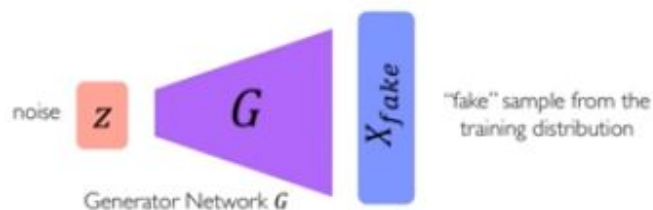
1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end
4. Interpret hidden latent variables using perturbation
5. Generating new examples

Generative Adversarial Networks : (GANs)

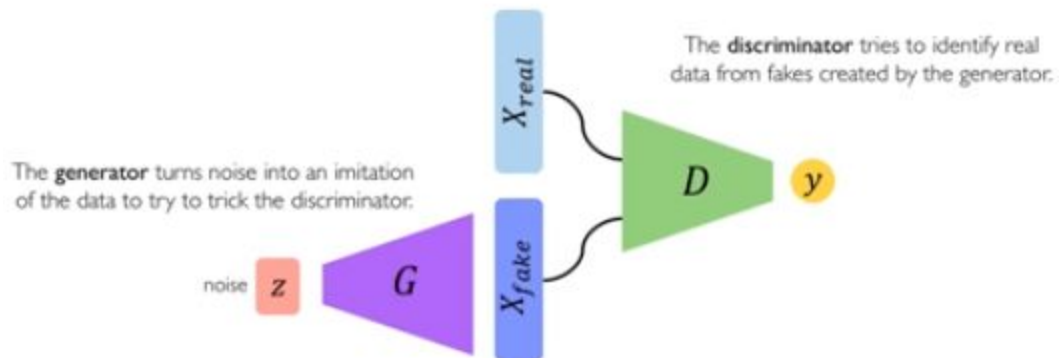
Idea: don't explicitly model density, and instead just sample to generate new instances.

Problem: want to sample from complex distribution – can't do this directly!

Solution: sample from something simple (noise), learn a transformation to the training distribution.



Generative Adversarial Networks (GANs) are a way to make a generative model by having two neural networks compete with each other.



Generator and Discriminator ~ both these are adversaries , so they are fighting with each other.

This forces the discriminator to become as better as possible to differentiate the real from the created one.

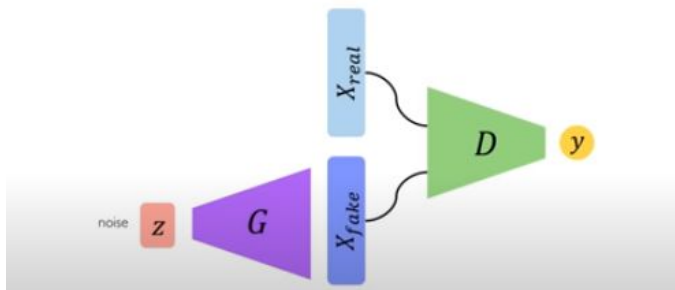
Intuition behind the GANs :

1. Generator starts from noise to try to create an imitation of the data.
2. Discriminator looks at both the real data and fake data created by the generator.
3. Discriminator tries to predict what's real and what's fake.



4. As the Discriminator becomes perfect the Generator tries to improve its imitation of the data.

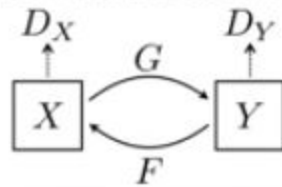
After training, use generator network to create **new data** that's never been seen before.



EXAMPLEs:

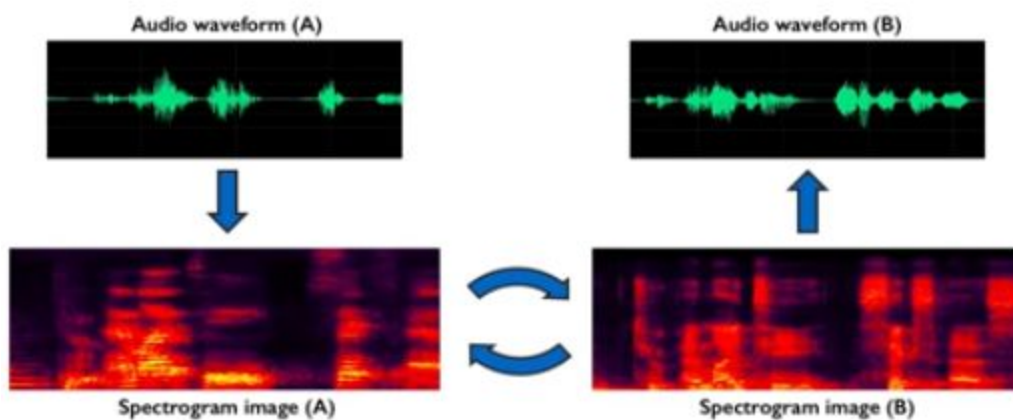
CycleGAN: domain transformation

CycleGAN learns transformations across domains with unpaired data.



1.

CycleGAN: Transforming speech



2.

DEEP REINFORCEMENT LEARNING

Reinforcement Learning :

Data given in state action pairs , the goal is to maximise future rewards over many time steps.

Agent :

The thing that takes the actions.

Environment :

The place where the agent acts

Actions :

The move that the agent can take in the environment.

State :

A situation where the agent perceives.

Goal of RL is to maximise the reward i.e the feedback or success/failure of the agent.
Total Reward exists.

Discounted Rewards :

Multiplying the discounting factor with reward and take the total sum , the discounting factor decreases over time (its between 0 and 1).

Defining the Q-function

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

Total reward, R_t , is the discounted sum of all rewards obtained from time t

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

The Q-function captures the **expected total future reward** an agent in **state, s** , can receive by executing a certain **action, a**

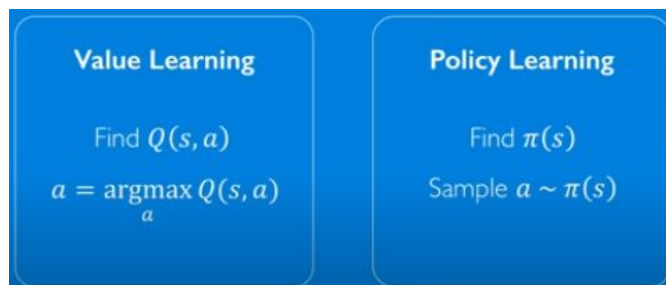
Higher the Q value tells the action is more desirable in this state.

Ultimately, the agent needs a **policy** $\pi(s)$, to infer the **best action to take** at its state, s

Strategy: the policy should choose an action that maximizes future reward

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

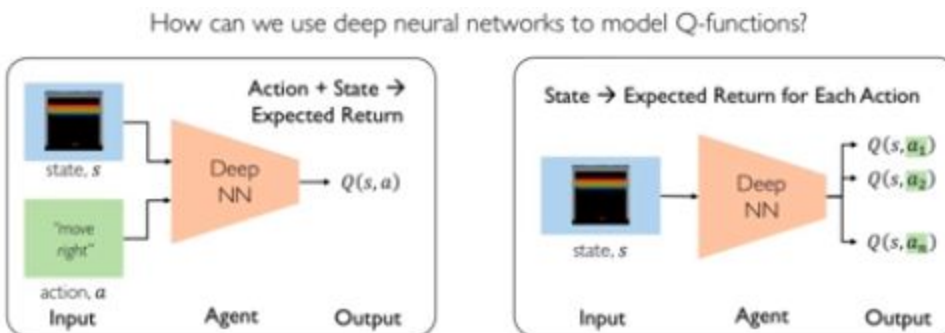
Two types of DRL algorithms exists :



Value Learning :

The Q Function is learnt then use it to determine the policy

Its difficult to estimate the Q function , choosing the (s,a) pair is difficult.

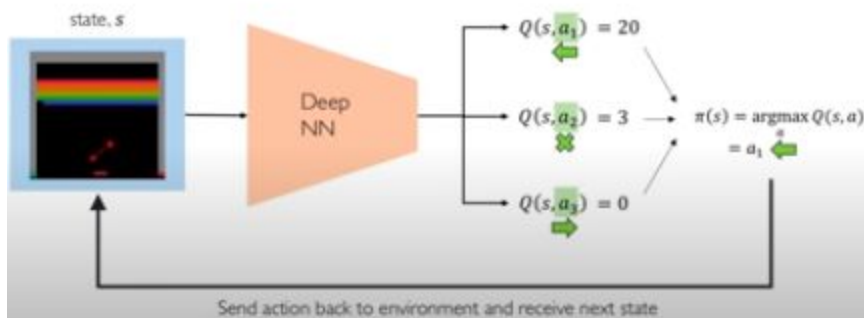


Maximise the target return -> that will train the agent.

$$\mathcal{L} = \mathbb{E} \left[\left\| \overbrace{\left(r + \gamma \max_{a'} Q(s', a') \right)}^{\text{target}} - \overbrace{Q(s, a)}^{\text{predicted}} \right\|^2 \right] \quad \text{Q-Loss}$$

Deep Q network summary :

Use NN to learn Q-function and then use to infer the optimal policy, $\pi(s)$



Downsides of Q-learning

Complexity:

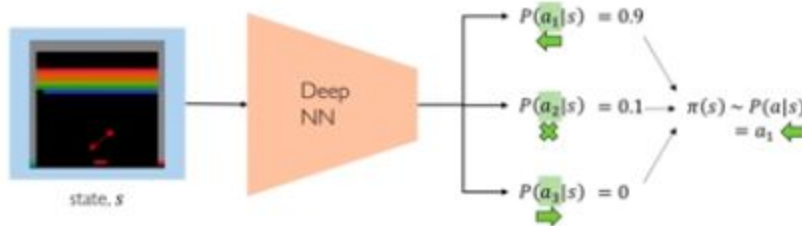
- Can model scenarios where the action space is discrete and small
- Cannot handle continuous action spaces

Flexibility:

- Policy is deterministically computed from the Q function by maximizing the reward \rightarrow cannot learn stochastic policies

To address these, consider a new class of RL training algorithms:
Policy gradient methods

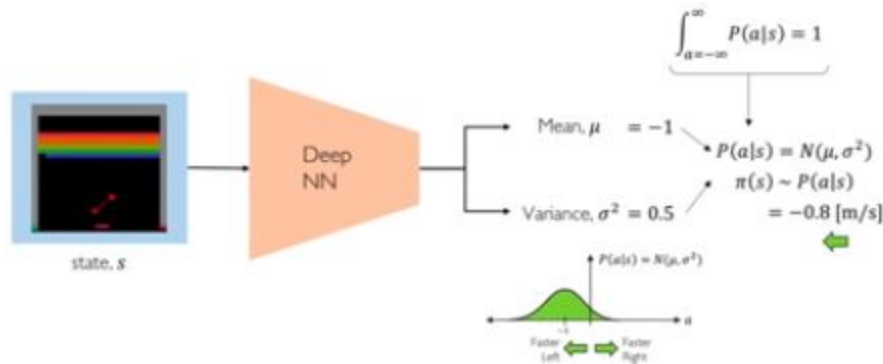
Policy Gradient: Directly optimize the policy $\pi(s)$



its total mass must add up to one.

Thus we are not restrained to categorical distribution.

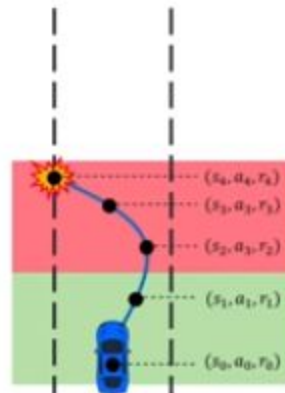
Leading to a Continuous Action Space rather than a discrete action space.



Training Policy Gradients

Training Algorithm

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Then keep repeating over and over until it becomes best.

Run until termination is not actually feasible in real life, photo realistic simulators can be used.

Deploying End-to-End RL for Autonomous Vehicles



Policy Gradient RL agent trained entirely within VISTA simulator



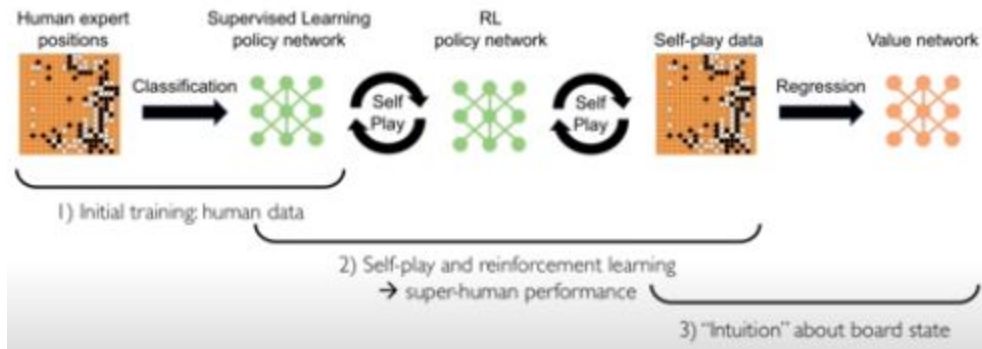
End-to-end agent directly deployed into the real-world



First full-scale autonomous vehicle trained using RL entirely in simulation and deployed in real life!

EXAMPLES :

AlphaGo Beats Top Human Player at Go (2016)

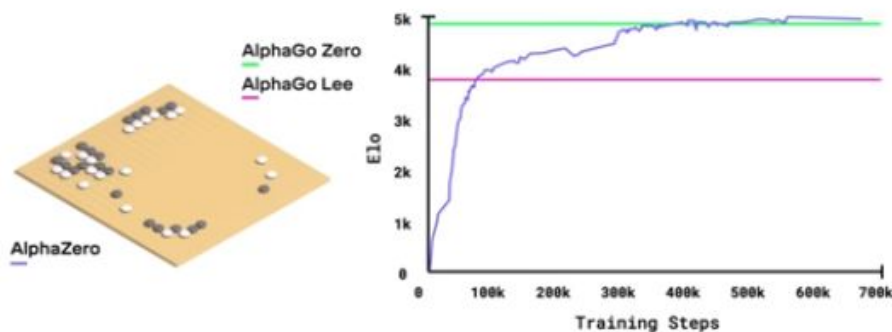


1.

This uses a build up of user data.

AlphaZero: RL from Self-Play (2018)

Go



2.

DEEP LEARNING : NEW FRONTIERS

Universal Approximation Theorem : A feedforward network with a layer is sufficient to approximate an arbitrary precision , any continuous function.

Limitations :

1. Number of hidden layers may be unfeasibly large.
2. The resulting model may not generalize

Deep neural networks are very good at perfectly fitting random function.

Therefore NN can be thought of as “Excellent” function approximators.

Adversarial Attacks on NN :

1. Take some data instance then perform some perturbation then it comes out as a nonsensical label for the original object.

Adversarial Image:

Modify image to increase error

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

“How does a small change in the input increase our loss”

Synthesizing Robust Adversarial Examples



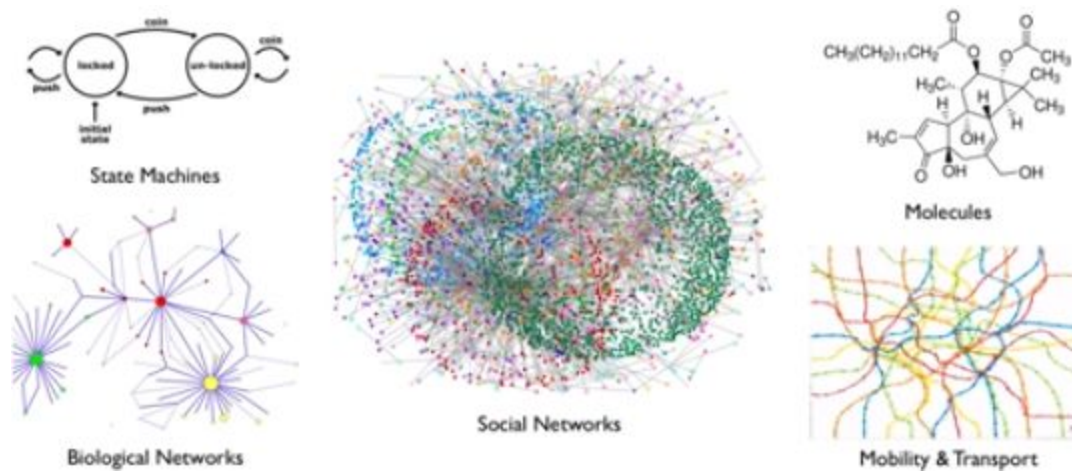
■ classified as turtle ■ classified as rifle
■ classified as other

- 2.

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**
- Difficult to **encode structure** and prior knowledge during learning
- Poor at **representing uncertainty** (how do you know what the model knows?)
- Uninterpretable **black boxes**, difficult to trust
- **Finicky to optimize**: non-convex, choice of architecture, learning parameters
- Often require **expert knowledge** to design, fine tune architectures

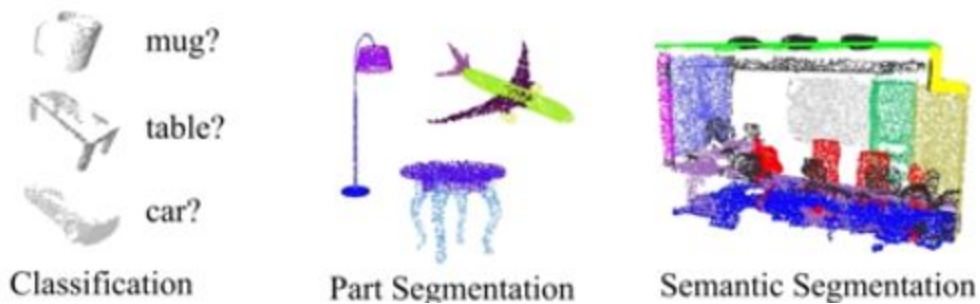
Graphs Convolutional Networks can solve :



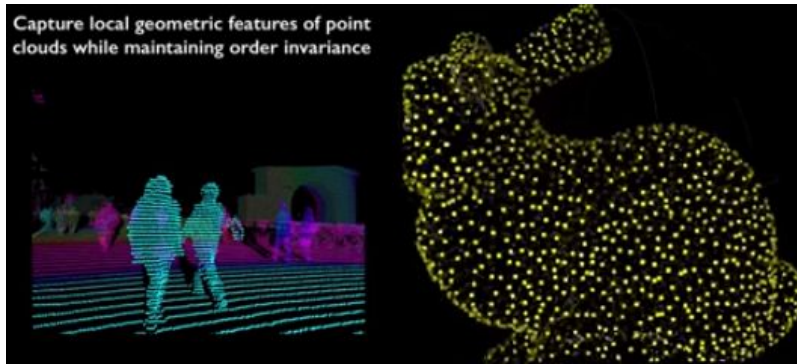
GCNs : instead of images graphs are present , the weights are moved across the nodes.

Learning from 3D data :

Point clouds are **unordered sets** with **spatial dependence** between points



Also PointCloud analysis can be implemented : the data points are distributed such that enables capturing the spacial information.



Probability is not a measure of confidence.
 Along with the probability, confidence should also be outputted.

Bayesian Deep Learning for Uncertainty enables that :

Approximates the posterior weights over the normal weights.

Network tries to learn output, \mathbf{Y} , directly from raw data, \mathbf{X}

Find mapping, f , parameterized by weights \mathbf{W} such that

$$\min \mathcal{L}(\mathbf{Y}, f(\mathbf{X}; \mathbf{W}))$$

Bayesian neural networks aim to learn a posterior over weights
 $P(\mathbf{W}|\mathbf{X}, \mathbf{Y})$:

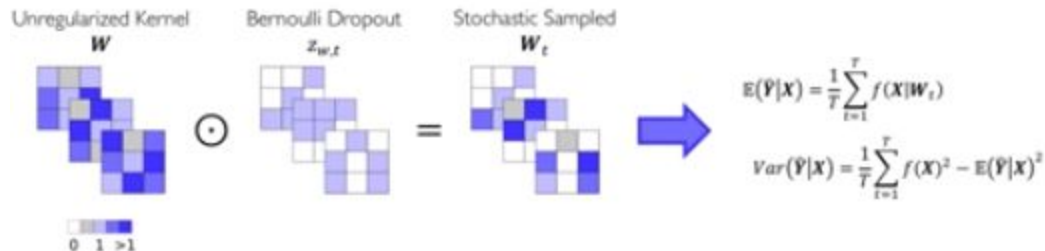
Intractable! $P(\mathbf{W}|\mathbf{X}, \mathbf{Y}) = \frac{P(\mathbf{Y}|\mathbf{X}, \mathbf{W})P(\mathbf{W})}{P(\mathbf{Y}|\mathbf{X})}$

Dropouts can be used ~

Dropout for Uncertainty

Evaluate T stochastic forward passes through the network $\{W_t\}_{t=1}^T$

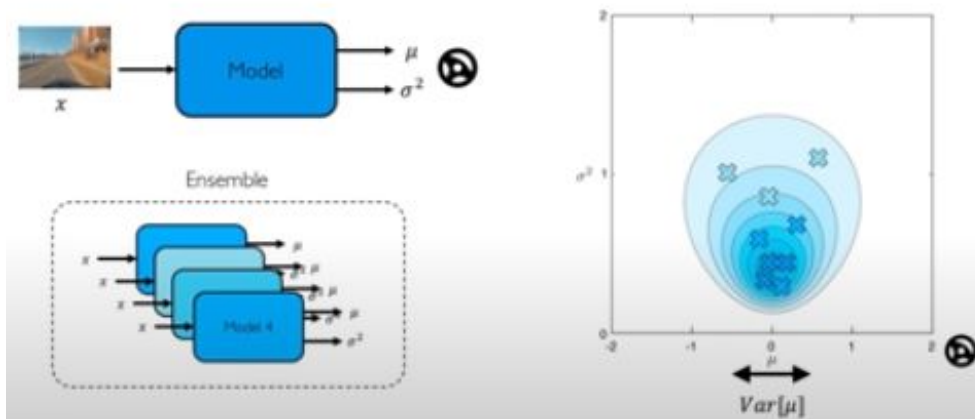
Dropout as a form of stochastic sampling $z_{w,t} \sim \text{Bernoulli}(p) \quad \forall w \in W$



By looking at the expected value and its variance gives us the uncertainty of the prediction.

By using different dropout gives us many estimates and then the variances can be observed, lower the spread out of the uncertainty better the prediction.

Model ensembling for estimating uncertainty

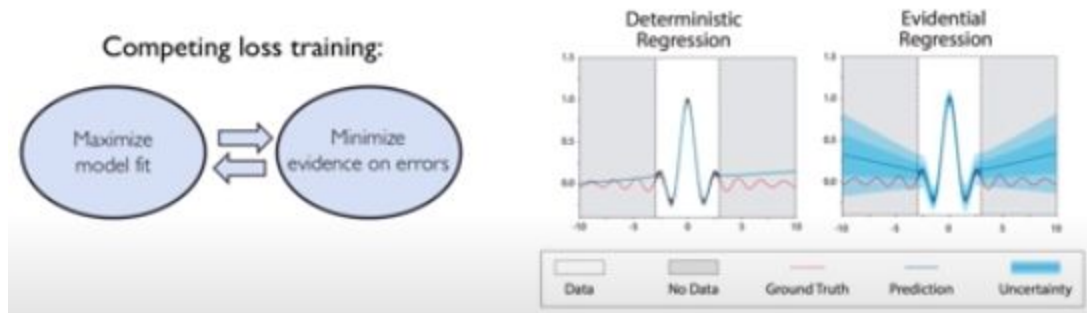


Thus we are actually learning the Evidential Distribution, effectively it says how much evidence the model has in support of a prediction.

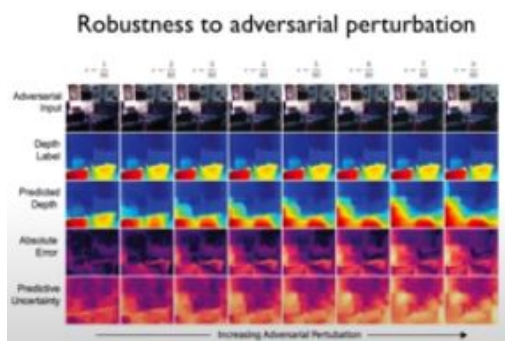
Telling us the uncertainty would be the best.

Evidential Deep Learning

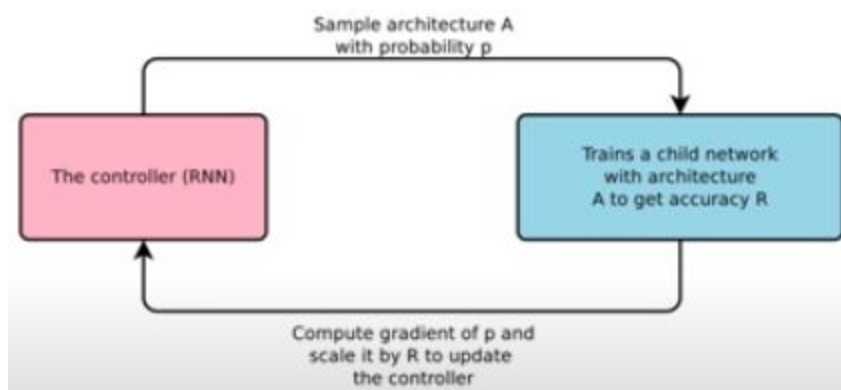
Directly learn the underlying uncertainties using **evidential distributions**



Thus it makes it robust to Adversarial Attacks : thus greater the extent of adversary or pampering greater the uncertainty.



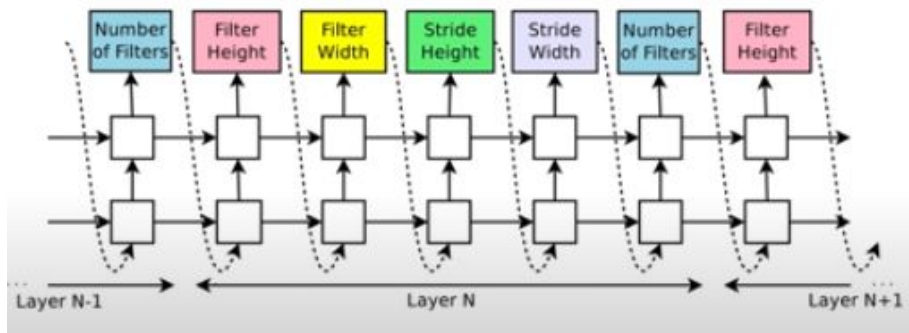
Auto ML : Automated Machine Learning (made by Google)



The RNN estimates the hyperparameters and using these the child NN is trained and the accuracy is measured and rewards are then put back into the RNN for better approximation.

AutoML: Model Controller

At each step, the model samples a brand new network



Then =>

AutoML: The Child Network



Compute final accuracy on this dataset.

Update RNN controller based on the accuracy of the child network after training.