Deep Learning

Art of extracting patterns from data using NN

Notes are compiled by keeping the lecture slides as priority.

There are additional material from over the internet that will reduce the browsing costs. I have mentioned important links for further reading.

Compiled by:

Sandeep Kumar Kushwaha

LinkedIn

Exam pattern:

3-5 topics: explain in detail everything that you know.

One topic: self-choice, explain.

Professor can ask in between, about processes & tricky questions.

Experience: Good. Easy. Can score obtain a score of 26-28 within preparation of ~ 25 hours. You need to get out of the slides to add extra information to get that.

Note: I have used repetitive topics like Neural Network as NN, Backpropagation as BP, etc to make notes more compact.

All the best.

Image Processing

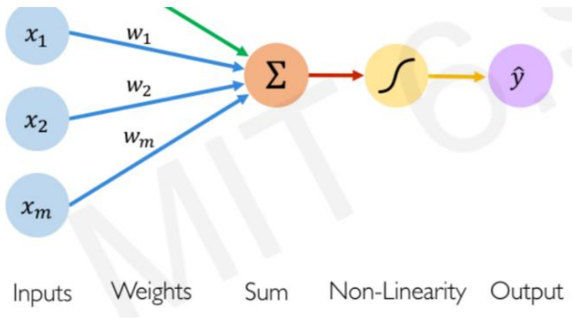
- MNIST
    - Grayscale images – 28*28 pixels
    - 60k train + 10k test images
- ImageNet
    - 15 million 22k objects high resolution images
- Image Detection

Expressiveness

THE PERCEPTRON:

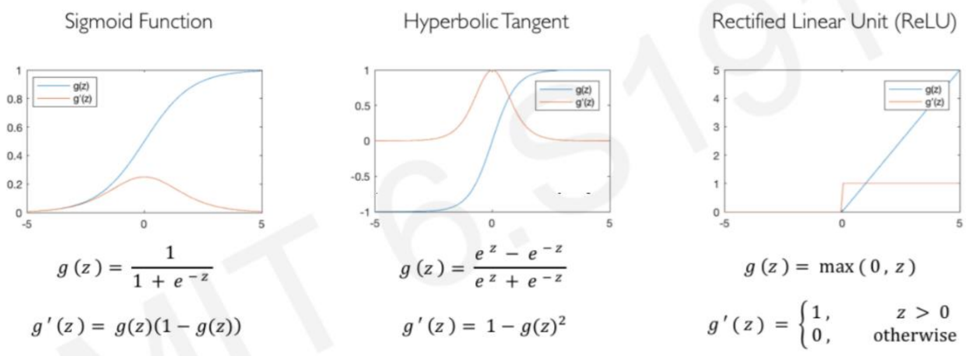
It is an algorithm that is used for supervised learning of the binary classifiers. It enables the neurons to learn and process the training set elements with time. There are two type of P

- **Single Layer** – it can only learn linear functions.
- **Multilayer** that contains more than 1 hidden layer apart form input and output layer. It can also learn non-linear functions.



Why need activation functions?

It works as an intermediate gateway between the neuron input and output being passed to the next layer. To introduce non linearities into network.



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

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

```
tf.math.sigmoid(z)
```

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

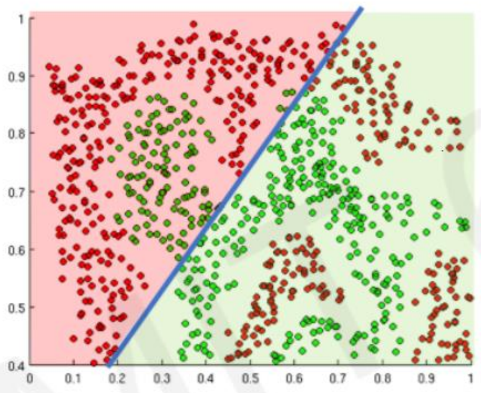
$$g'(z) = 1-g(z)^2$$

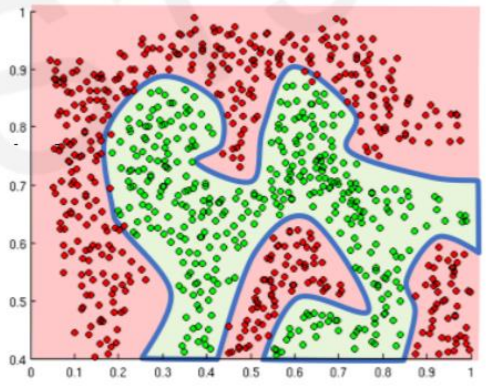
```
tf.math.tanh(z)
```

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

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

```
tf.nn.relu(z)
```


Linear activation functions produce linear decisions no matter the network size


Non-linearities allow us to approximate arbitrarily complex functions

- <u>Hyperplanes:</u>
  - These are decision boundaries that help classify the data points.
  - The dimension of hyperplane depends upon the number of features.
  - If the number of input features is 2, then the hyperplane is just a line.

- Shallow nets are complete, why choose Deep nets?
  - Because it computes the same function with less neural units.
  - Activation functions play important role. They are the source of non-linearity.

- Every continuous function can be approximated by NN.

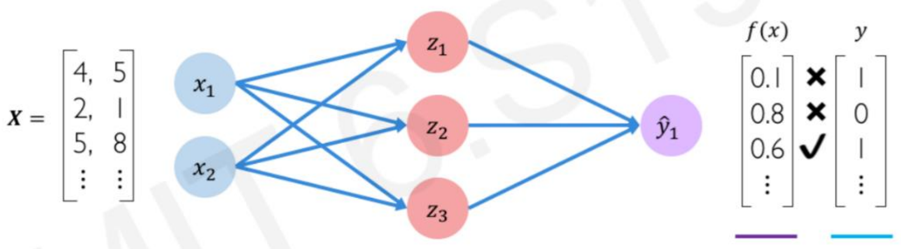
- Expressive Power of Deep Learning

  - Text and images can be more efficiently represented by deep hierarchical networks than the shallow ones.

<u>Loss</u>

- <u>Loss:</u> its measure of the cost we had to bear because of incorrect predictions. If predictions deviates too much from actual results, loss function would cough up a very large number.

- <u>Empirical loss:</u> total loss over entire dataset-



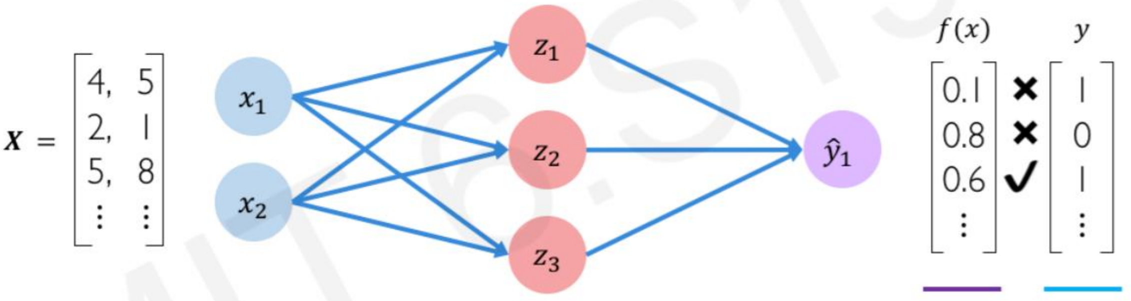
$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$f(x)$
y
$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}$ ✗ ✗ ✓ $\begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$

Also known as:
- Objective function
- Cost function
- Empirical Risk

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

Binary Cross Entropy Loss:

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

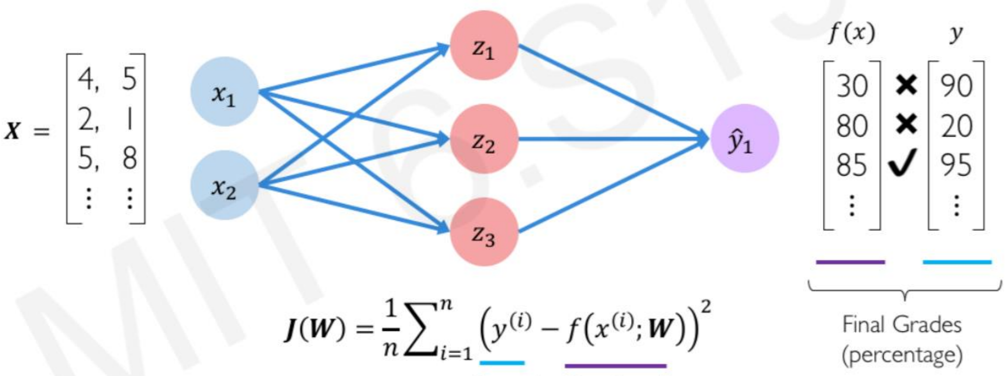


$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$f(x)$
y
$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}$ ✗ ✗ ✓ $\begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + \big(1-\underbrace{y^{(i)}}_{\text{Actual}}\big)\log\Big(1-\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

Mean squared error loss:

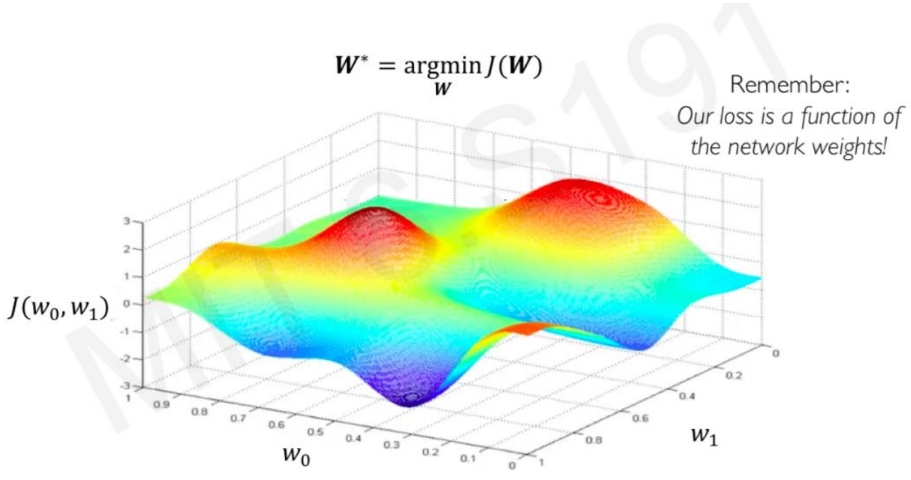
Mean squared error loss *can be used with regression models that output continuous real numbers*



$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
$f(x)$
y
$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix}$ ✗ ✗ ✓ $\begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$
Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \Big(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)^2$$

Training: process to find the best weights that achieve the lowest loss

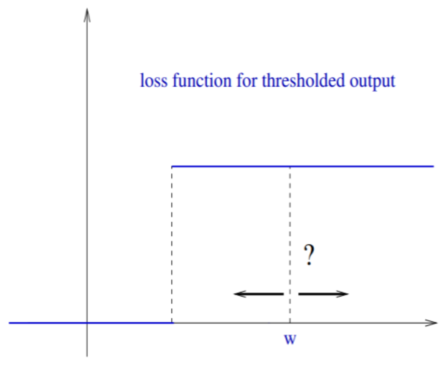


Loss optimization: Gradient Descent

- Optimization algorithm to find a local minimum of a differentiable function.
- We perform GD by taking repeated steps in the opposite direction of the gradient of the function at the current point, because that will be the direction of steepest descent.

minimize Loss. How can we do it?

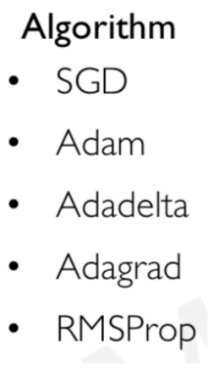
- By changing the parameters of the model.
- So, we can change parameters using a Naive approach- Learning by trials
  - Similar to Reinforcement Learning that has a reward based action model.
  - Inefficient.
- So instead of randomly adjusting the parameters, lets predict them,
  - We use derivatives to know where we are going.
  - If angle is <90 then derivative is positive; we must decrease the parameter.
  - Magnitude of derivative is related to the steepness of tangent.
-
- Why binary thresholding is not good for learning? Coz the derivative is always 0.



THE GRADIENT DESCENT

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

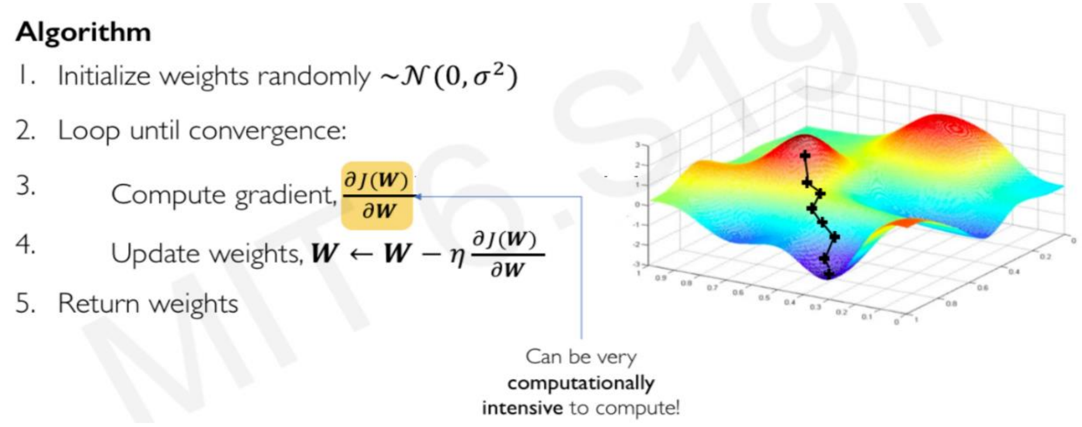
- Gradient descent is a general-purpose algorithm that numerically finds minima of multivariable functions.
- it finds local minima, but not by setting \nabla f = 0∇f=0del, f, equals, 0 like we've seen before. Instead of finding minima by manipulating symbols, gradient descent approximates the solution with numbers.
- When we have many parameters, we have different derivatives for each – partial derivative.
- The vector of all partition derivatives is called gradient of the function.
-
- $$\nabla_w[L(w)] = [\frac{\partial L(w)}{\partial w_1}, \ldots, \frac{\partial L(w)}{\partial w_n}]$$
-
-
- The gradient points in the direction of steepest ascent. Magnitude of partial derivatives are relevant to govern the orientation of the gradient.
-
- The Gradient Descent Technique
  - Start with random parameters.
  - Compute gradient of loss function.
  - Make a "small step" in the direction of gradient (maximise); OR
  - Make a "small step" against the direction of gradient (minimize)



Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

Algorithms: GD model



Algorithm

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

Can be very **computationally intensive** to compute!

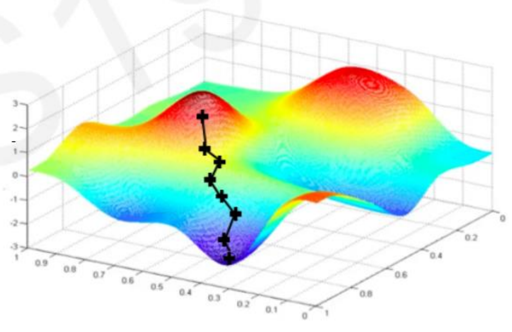
Stochastic Gradient Descent: we compute the Gradient on single data point i

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

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
```

Easy to compute but
very noisy (stochastic)!

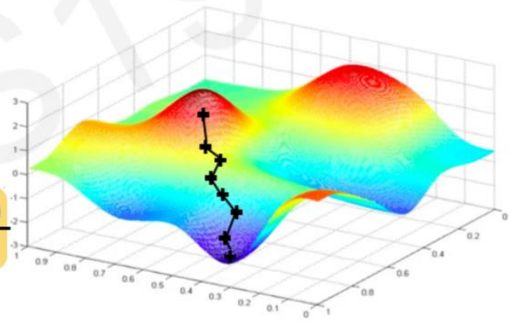


Instead of picking a point, we choose a batch of points

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
```

Fast to compute and a much better
estimate of the true gradient!



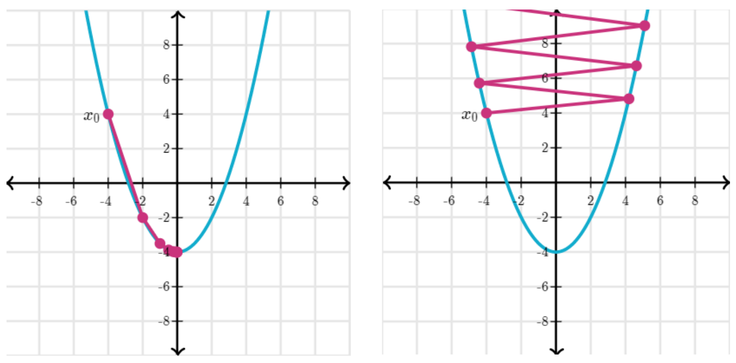
Mini batches while training:

More accurate estimation of gradients
Smoother convergence
Can be used for large Learning Rates
Leads to fast training
Can perform parallel computation – increased speed performance on GPU.

- How to avoid getting stuck at local minima?
  - We can use momentum gradient descent
  - Limitations of GD:
    - It only finds a local minima (rather than global).
    - each valley, it would trap the GD algorithm.
    - We can never distinguish the global and local minima in a go.
    - The function should be differentiable everywhere. No breaks in the graph.

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

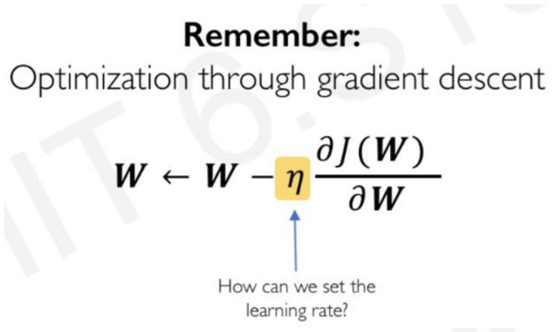
- What is step size?
  - It controls the rate of convergence
  - A step size too large, may never converge to a local minimum since it will overshoot.
  - Good step size moves towards minimum rapidly, and converges slowly



Learning Rate (μ)

The dimension of the step in the direction of the gradient is LR.

- A small positive value between 0 and 1.
- LR controls how quickly the model adopts to the problem.

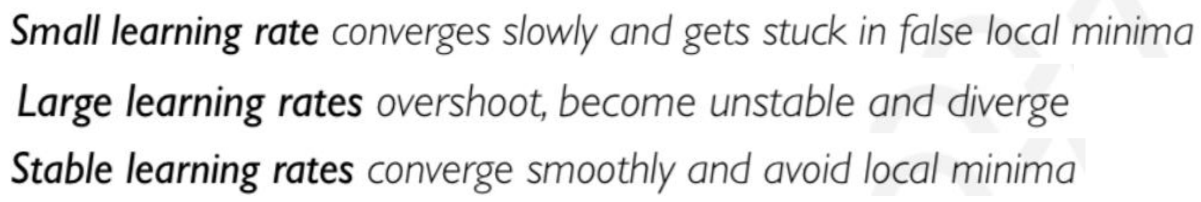


Remember:

Optimization through gradient descent

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

How can we set the learning rate?

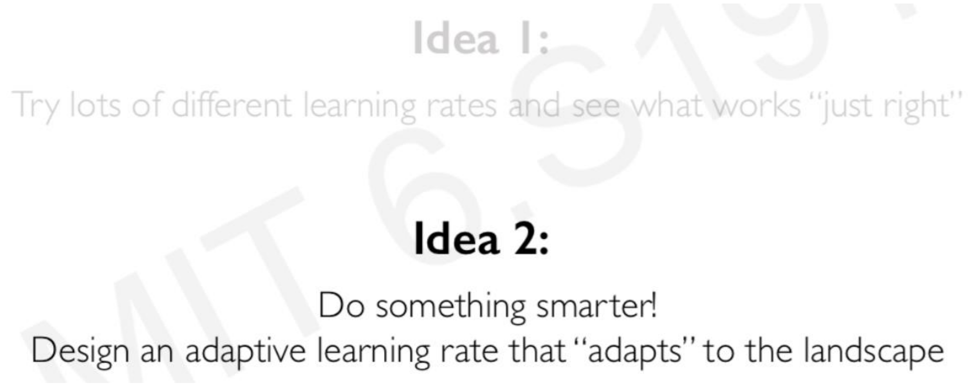


Small learning rate *converges slowly and gets stuck in false local minima*

Large learning rates *overshoot, become unstable and diverge*

Stable learning rates *converge smoothly and avoid local minima*

How to deal with this?



Idea 1:

Try lots of different learning rates and see what works "just right"

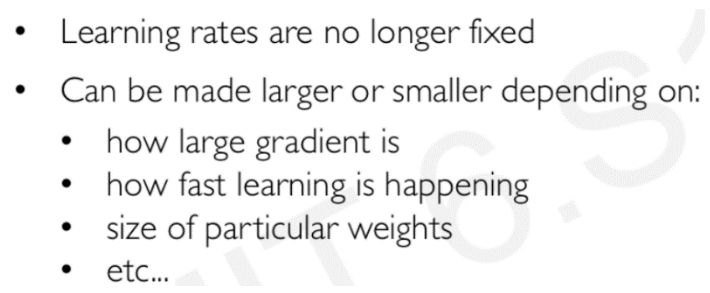
Idea 2:

Do something smarter!

Design an adaptive learning rate that "adapts" to the landscape

Adaptive Learning Rates:

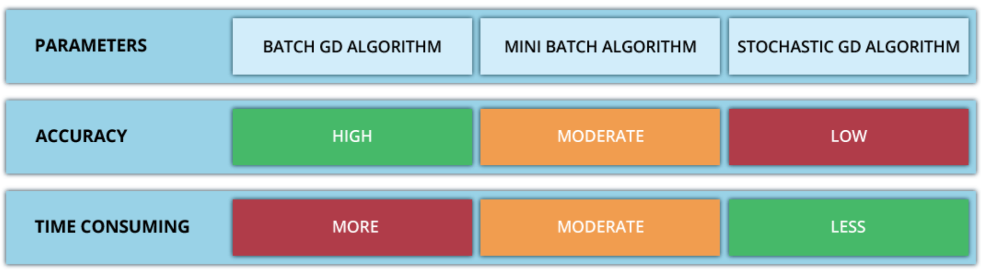
These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.



- 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...

Linear Optimization Problem

- Backpropagation Method
- Can be generalized to multilayer non-linear networks
-
- Optimizations
- How to update the weights:
  - Online -
    - after each training sample.
    - GD zigzags around the direction of steepest descent.
  - Full batch –
    - full sweep through training data
    - GD points to direction of steepest descent on the error surface
  - Mini batch –
    - for a small random set of training cases
    - good compromise for GD
-
- There are three variations of Gradient Descent:



PARAMETERS	BATCH GD ALGORITHM	MINI BATCH ALGORITHM	STOCHASTIC GD ALGORITHM
ACCURACY	HIGH	MODERATE	LOW
TIME CONSUMING	MORE	MODERATE	LESS

  -
-

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

- Stochastic Gradient Descent
  - SGD performs parameter update for each example
  - Performs frequent updates with high variance that causes objective function to fluctuate heavily.
  - The fluctuation can helps to find better local minima, also causes overshooting.
  - When learning rate is decreased, it behaves like a batch GD with good convergence.
  - The gradient of loss function should be computed over all training samples in case of full batch and small set in case of mini batch.
  - Can be expensive, if dataset is huge
  -
-
- Momentum-
  - If GD moves in a stable direction, we can improve its magnitude.
  - We try to reduce the risk of getting stuck at a local minima, just like a ball rolling down.
  - The momentum can be updated wrt time here:
    - *v^t is vector of updates at time t.*
  -

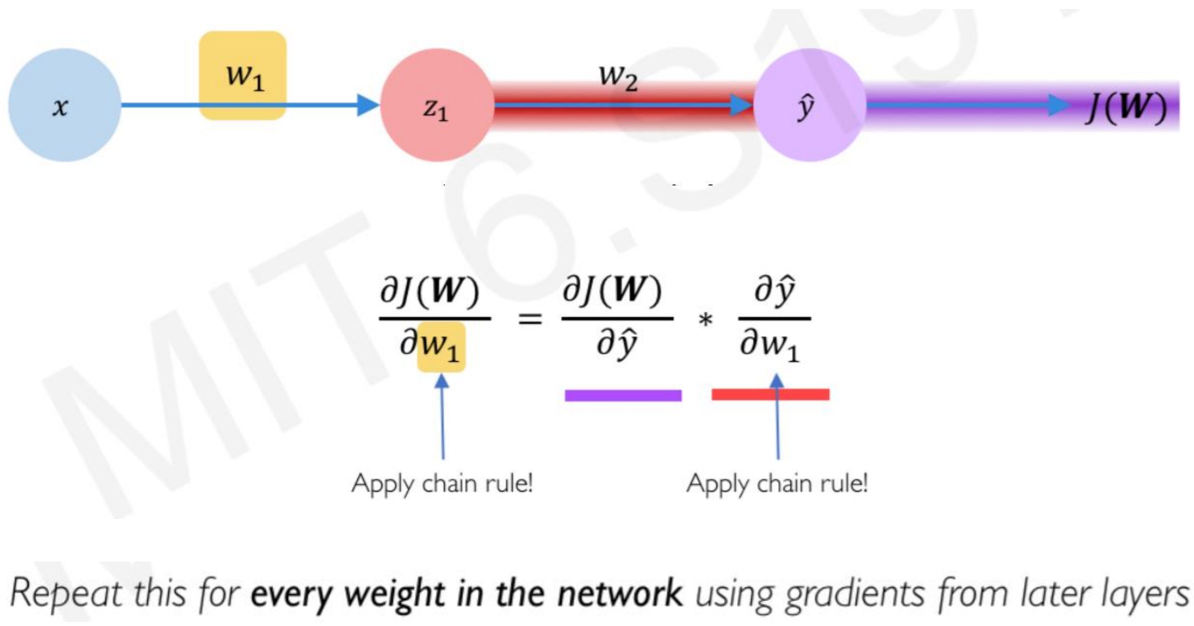
$$v^t = \underbrace{\mu * \nabla L(w)}_{\text{gradient step}} + \underbrace{\alpha * v^{t-1}}_{\text{momentum}}$$

- Nesterov momentum
  -
  - Similar to previous method
  - Only difference is the position at which gradient is computed
- Here the gradient term is not computed from the current position $\theta_t \theta t$ in parameter space but instead from a position $\theta_{intermediate}=\theta_t+\mu v_t \theta intermediate=\theta t+\mu vt$. This helps because while the gradient term always points in the right direction, the momentum term may not. If the momentum term points in the wrong direction or overshoots, the gradient can still "go back" and correct it in the same update step.

==

THE BACKPROPAGATION THEOREM

- It can be expressed as backward propagation of errors.
- In an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights.
- It propagates the total loss back into the neural network to know how much of the loss every node is responsible for, and subsequently updating the weights in such a way that minimizes the loss. This is done by giving the nodes with higher error rates lower weights and vice versa.
- Iteratively computes partial derivatives of loss function to each parameter of the n/w.
- To terminate the backpropagation on the network we can use multiple methods.
    - Training the network with fixed number of epochs – iteration
    - Setting a threshold for error, if the error goes below that level we can stop training the NN.
    - Creating a validation of sample data. After each iteration we can validate our model with this validation set & choose the highest accuracy model as final.



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

*Repeat this for **every weight in the network** using gradients from later layers*

❖ **Computing the Gradient:**

- In NN we compute gradient of complex function due to numerous layers of NN.
- How can we compute gradient wrt specific parameter?
  - Use chain rule

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{derivative of composition}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

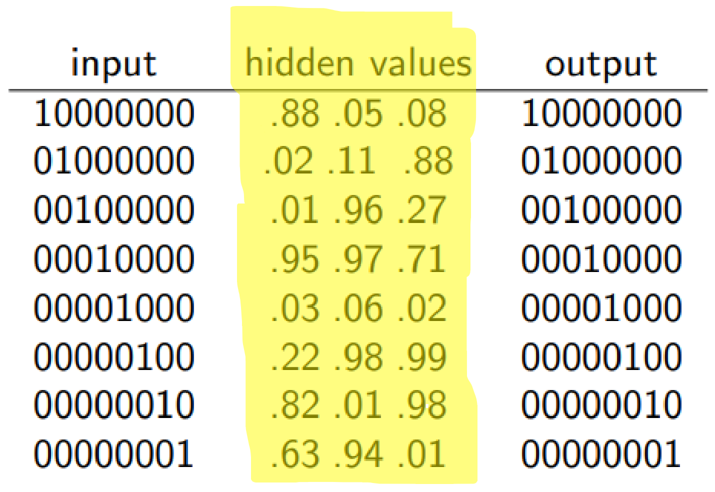
- Forward propagation calculates & stores the intermediate variables in the NN. The direction of flow if from the left and towards the right.
- When training the model, fwd propagation & bP are inter dependent.

- BP algorithm explained.
- Derivatives of the common activation function

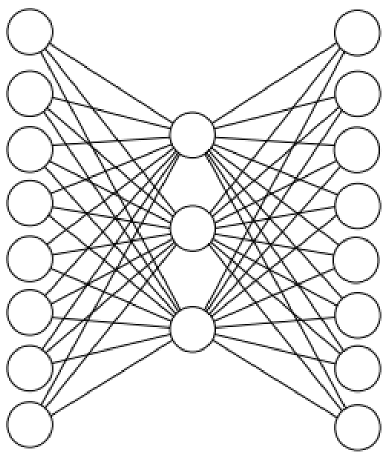
activation function	**derivative**
logistic function	
$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$
hyperbolic tangent	
$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$tanh'(x) = sech^2(x)$
rectified linear	
$relu(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$	$relu'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

-
-

- NN from scratch
  - Using BP in NN



input	hidden values	output
10000000	.88 .05 .08	10000000
01000000	.02 .11 .88	01000000
00100000	.01 .96 .27	00100000
00010000	.95 .97 .71	00010000
00001000	.03 .06 .02	00001000
00000100	.22 .98 .99	00000100
00000010	.82 .01 .98	00000010
00000001	.63 .94 .01	00000001



-
- The data in yellow was obtained after training, we started with only input and O/P values. Hidden values are new representation of data.
- ➢ Shannon theory

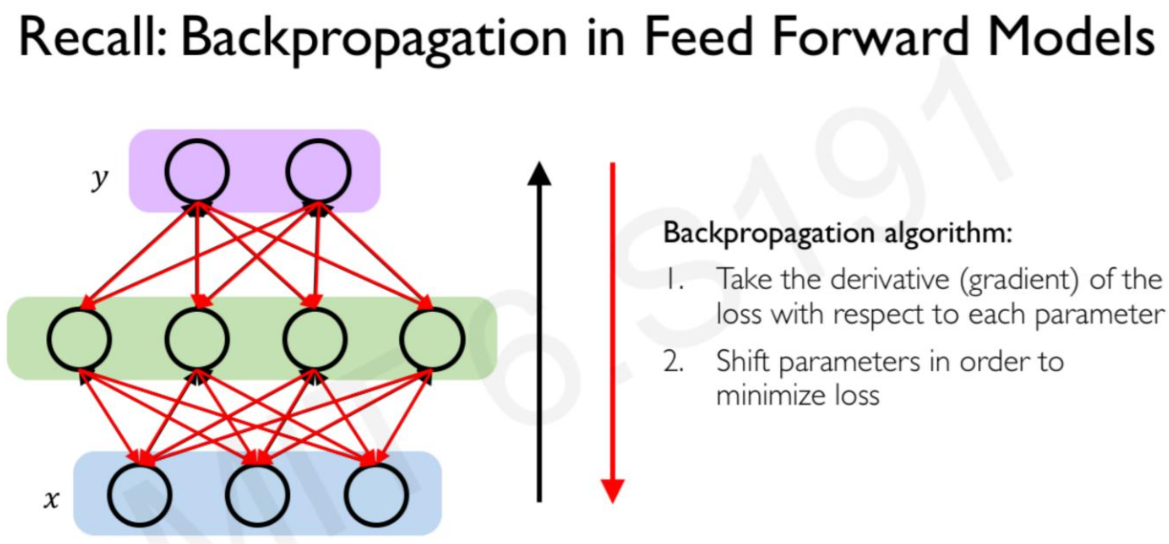
-

❖ Learning Issues
- ➢ Why is learning slow?
  - ▪ If either its i/p is low, or o/p is saturated (close of 1/0)
  - ▪ The GD can be slow, and may point to the local minimum
  - ▪ In BP, when activations are low, weights learn/change slowly.
  - ▪ In BP, we can get saturated neurons as well.

-
  - ➢ Vanishing gradient problem

    Solution: The simplest solution is to use other activation functions, such as ReLU, which doesn't cause a small derivative. Residual networks are another solution, as they provide residual connections straight to earlier layers.
  - ➢ Optimization rules

Backpropagation Through Time (BPTT)



Recall: Backpropagation in Feed Forward Models

Backpropagation algorithm:

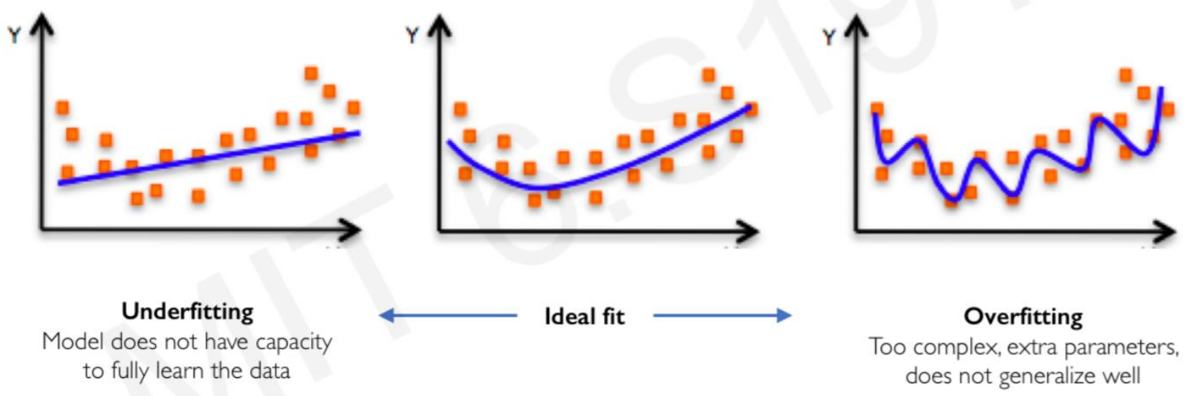
1. Take the derivative (gradient) of the loss with respect to each parameter
2. Shift parameters in order to minimize loss

- ==

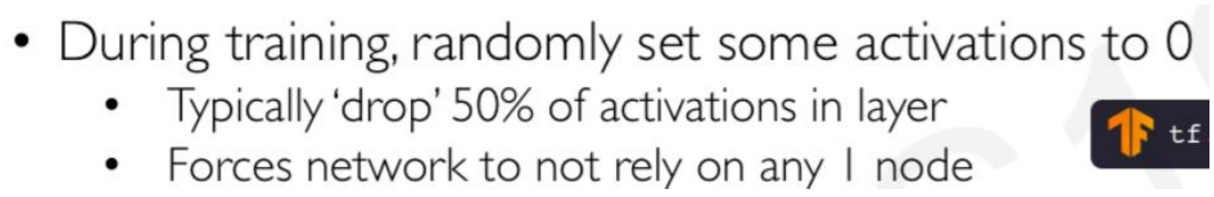
- SLIDE 7
- **Overfitting**
  - ➢ Here the model fits exactly according to the training data. It is too complex and has many extra parameters. Such kind of model is not generalised to our data instead it is more specified towards the training set.
  - ➢ Even the noise and fluctuations are picked up & learned as concepts by the model.

❖ **How to know when we are overfitting?**

- ➢ By checking validation metrics such as accuracy and loss.
- ➢ Overfitting when model's error on training set is very low but large on the test set.



- **Ways to handle overfitting:**
  - Using Dropout layers - randomly remove certain features by setting them to zero.



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

  - Reduce model capacity – remove the layers or else we can reduce the number of elements in the hidden layers.
  - *Regularization/weight decay*– we can add cost to the loss functions with large weights. Regularization helps in improving the generalization of our model when new data arrives. **Types of Regularization**: **L1, L2, dropout, early stopping, and data augmentation.**
  - Model averaging - using several models at once for making predictions
  - Data augmentation - technique of increasing the size of data used for training a model. Can be done two ways – position augmentation [scaling, cropping, flipping, rotation] OR Colour augmentation[Brightness, Contrast, Hue]

- **Regularization 1:**

  L1 & L2 regularization

  We add regularization term with the loss to obtain total cost function.

  Cost function = Loss + Regularization term

  - L2 [**ridge regularization**]- it forces the weight to reduce but never makes them zero. in L2 the square of the weights are penalised.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

  - L1 [Lasso regularization] - where weights are never reduced to zero, in L1 the absolute value of the weights are penalised.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

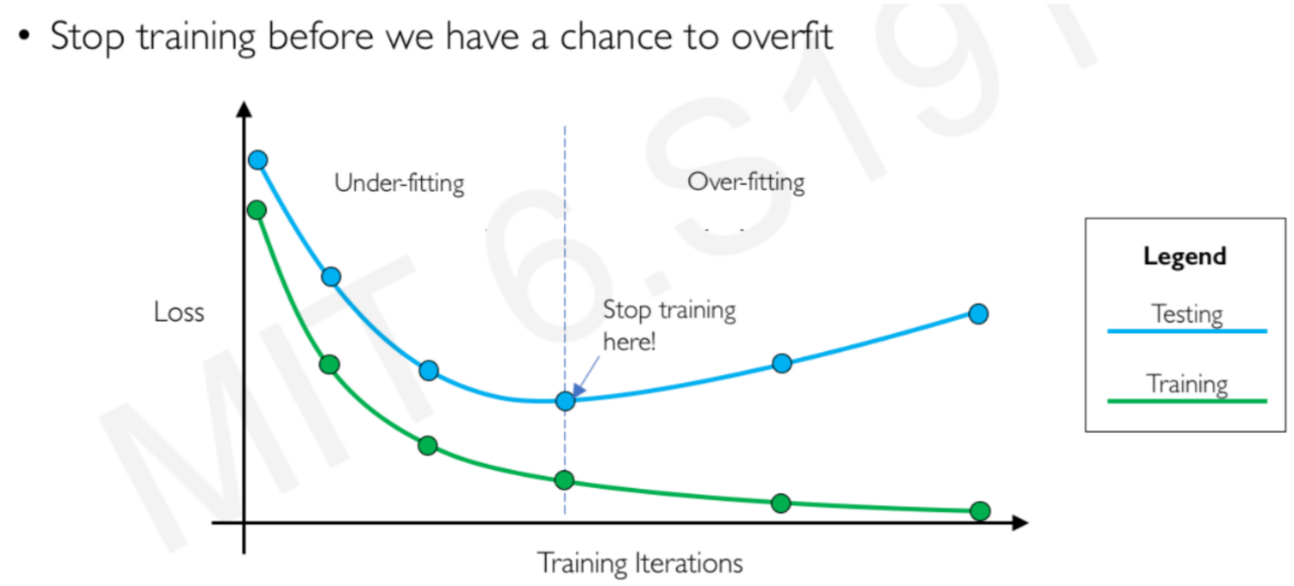
  Dropout

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

- Cripple the NN by randomly removing hidden units. It means that they are temporarily obstructed from influencing or activating the downward neuron in a forward pass, and none of the weights updates is applied on the backward pass.
- Hidden units cannot co-adapt with other and they become self reliant.
- So neurons are randomly dropped out of the network during training, the other neurons step in and make the predictions for the missing neurons. Due to this, there is the network becomes less sensitive to the specific weight of the neurons.

Early Stopping

- It is a kind of cross-validation strategy where one part of the training set is used as a validation set, and the performance of the model is tested against this set. So, as soon as the performance on validation set gets worse, the training is immediately stopped.


• Stop training before we have a chance to overfit

Underfitting

- The model is too simple and is unable to capture the relationship between the input and output variables accurately.
- It generates a high error rate on both the training set and unseen data.
- Good fitting is not the goal instead generalization.
- Handling Underfitting:
    - By decreasing Regularization – Regularization reduces the variance of the model by applying penalty on the input parameters with large coeff.

When we decrease the regu, the complexity and variation in the model increases.

- By increasing the duration of training.
- Feature selection – We should focus of features of more importance. The features we are choosing are not sufficient to portray the relations between the data, so we need to consider important features.

- Activation of loss functions for Classification

❖ Sigmoid

- When the n/w is a value b/w and 1 (binary classification problem) we use sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \quad 0$$

❖ Softmax

- Used when result of network is a probability distribution over k different categories.
- Sum equals upto 1

$$\text{softmax}(j, x_1, \ldots, x_k) = \frac{e^{x_j}}{\sum_{j=1}^{k} e^{x_j}}$$

❖ Softmax vs Sogmoid

- Softmax is multiclass/Sigmoid is two-class logistic regression
- Softmax is extension of Sigmoid.

- Cross Entropy

❖ Loss Functions: maps decisions to their associated costs. [link]

- Gradient Descent is optimization strategy that reduces the loss
- Difference between Loss function and Cost Function: Loss function is error function calculated over single training instance, whereas cost function is average loss over the entire TrainSet. Optimization deals with minimizing the costFunc.
- Regression loss functions:
  - Squared Error Loss
  - Absolute EL

  - Huber
- Binary Classification Loss Function
  - Binary cross Entropy loss: entropy ~ disorder/uncertainty
  - Hinge Loss: used with SVM

- Multi Class Classification Loss F
  - Multi Class Cross Entropy Loss
  - KL Divergence – measure of how a probability distribution differs from another distribution. KL divergence of 0 means, distributions are equal.

$$\begin{aligned} DKL(P\|Q) &= \sum_i P(i) log \frac{P(i)}{Q(i)} \\ &= \sum_i P(i)(logP(i) - logQ(i)) \\ &= -\underbrace{\mathcal{H}(P)}_{entropy} - \sum_i P(i) logQ(i) \end{aligned}$$

  - -It's a measure of information loss due to approximation P and Q.
  - - This function is not symmetric
  - - the cross entropy b/w P and Q:

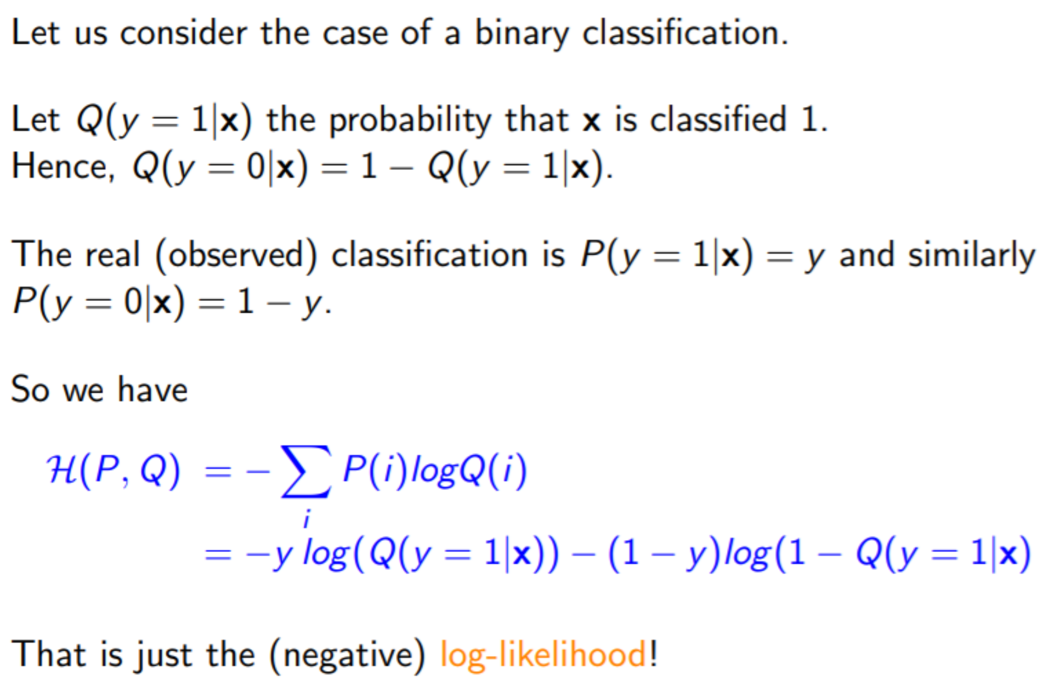
$$\mathcal{H}(P, Q) = -\sum_i P(i) logQ(i) = \mathcal{H}(P) + DKL(P\|Q)$$

  -
-

❖ Minimizing the cross entropy
- We can consider learning objective to be - minimizing KL divergence:
- Since the entropy H(P) is constant – minimizing DKL(P||Q) is equivalent to minimizing cross entropy H(P,Q) between P & Q.
-

❖ Cross Entropy & log likelihood

- Log likelihood measures the goodness of fit of statistical model to sample of data for given values of the unknown parameters.



Let us consider the case of a binary classification.

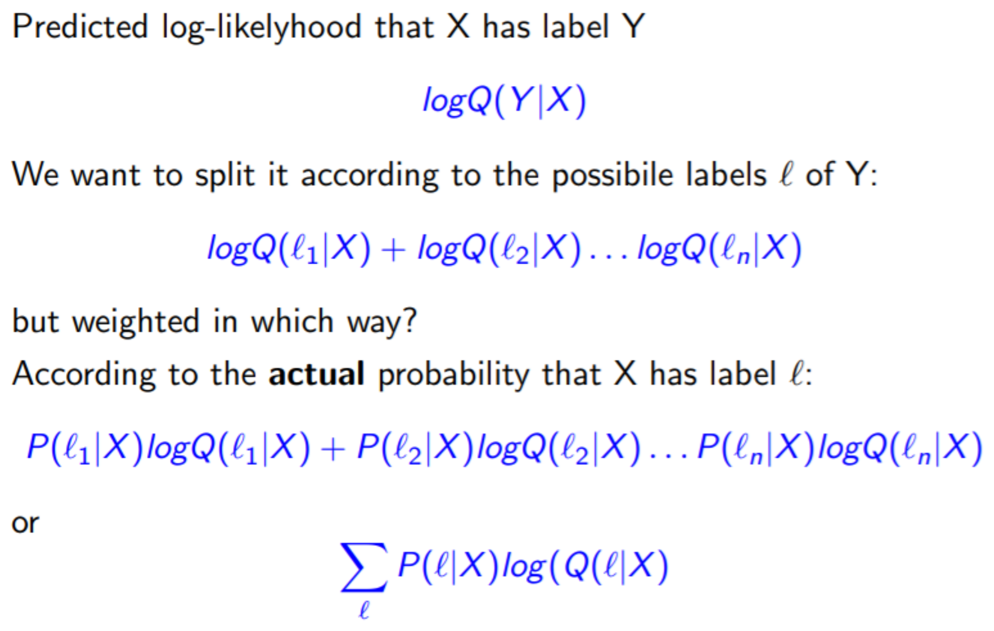
Let $Q(y=1|\mathbf{x})$ the probability that $\mathbf{x}$ is classified 1.
Hence, $Q(y=0|\mathbf{x}) = 1 - Q(y=1|\mathbf{x})$.

The real (observed) classification is $P(y=1|\mathbf{x}) = y$ and similarly $P(y=0|\mathbf{x}) = 1 - y$.

So we have

$$\begin{aligned}\mathcal{H}(P,Q) &= -\sum_i P(i) log Q(i) \\ &= -y\, log(Q(y=1|\mathbf{x})) - (1-y) log(1 - Q(y=1|\mathbf{x})\end{aligned}$$

That is just the (negative) log-likelihood!



Predicted log-likelyhood that X has label Y

$$log Q(Y|X)$$

We want to split it according to the possibile labels ℓ of Y:

$$log Q(\ell_1|X) + log Q(\ell_2|X) \dots log Q(\ell_n|X)$$

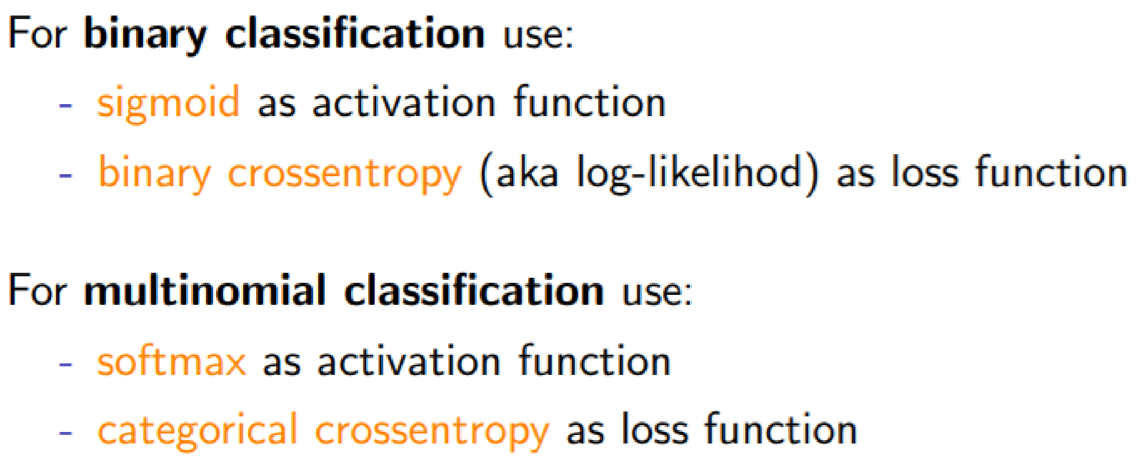
but weighted in which way?
According to the **actual** probability that X has label ℓ:

$$P(\ell_1|X) log Q(\ell_1|X) + P(\ell_2|X) log Q(\ell_2|X) \dots P(\ell_n|X) log Q(\ell_n|X)$$

or

$$\sum_\ell P(\ell|X) log(Q(\ell|X)$$

-


For **binary classification** use:

- sigmoid as activation function
- binary crossentropy (aka log-likelihod) as loss function

For **multinomial classification** use:

- softmax as activation function
- categorical crossentropy as loss function

-
- ===

CNN

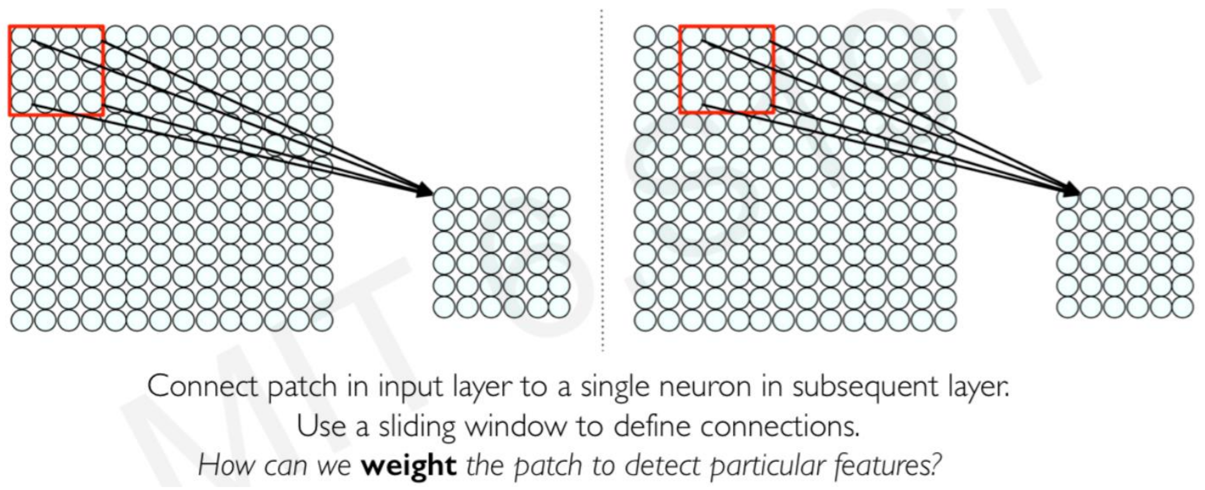
❖ A class of deep neural networks, most applied to analyse visual images. The conv nets reduce the images into a form that is easier to process, without losing features that are critical for getting a good prediction.

❖ **Working of CNN**:-

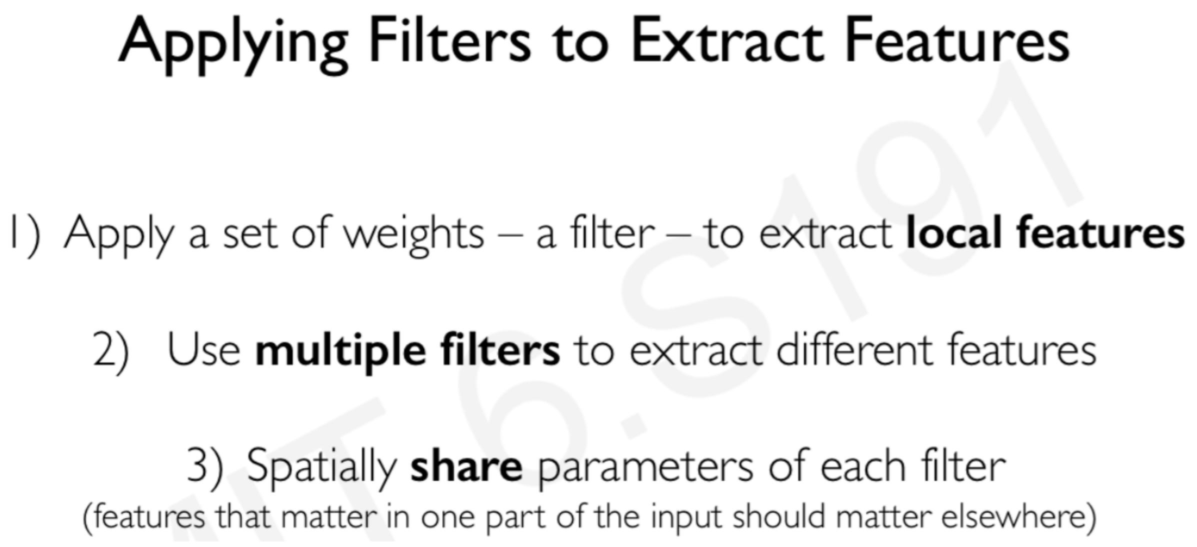
- ➢ Let's talk about images first. An RGB image is nothing but a matrix of pixel values having three planes whereas a grayscale image is the same but it has a single plane. Images can be represented as arrays. Interesting points in the images- where there is sharp change on intensity, calculated by derivatives.
- ➢ The activation of neuron is not influenced form all neurons of the previous layer BUT only from subset of adjacent neurons called as filter - the receptive field
- ➢ Every neuron works as convolution filter.
- ➢ Weights are shared – evey N perform the same transformation on different areas of i/p
- ➢ Conv is symmetric, associative and distributive.
- ➢ In the end we are left with assembling local featues of image into global structure.
- ➢ To perform the Convolution, we take a filter also known as kernel and process the operation on batch of these pixels and in result we a new convolved image with desired features.

- In ConvNet, each layer generates several activation functions that are passed on to the next layer.
- The first layer usually extracts basic features such as horizontal or diagonal edges. This output is passed on to the next layer which detects more complex features such as corners or combinational edges. As we move deeper into the network it can identify even more complex features such as objects, faces, etc.
- **Pooling Layer**: is responsible for reducing the spatial size of the Convolved Feature. Once the dimensions are reduced, there is decrease in the computational power required to process the data. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction.
  - **Max pooling**: find the maximum value of a pixel from a portion of the image covered by the kernel.
  - **Average Pooling**: average of all the values from the portion of the image covered by the Kernel. Max Pooling performs a lot better than Average Pooling.

•



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?*

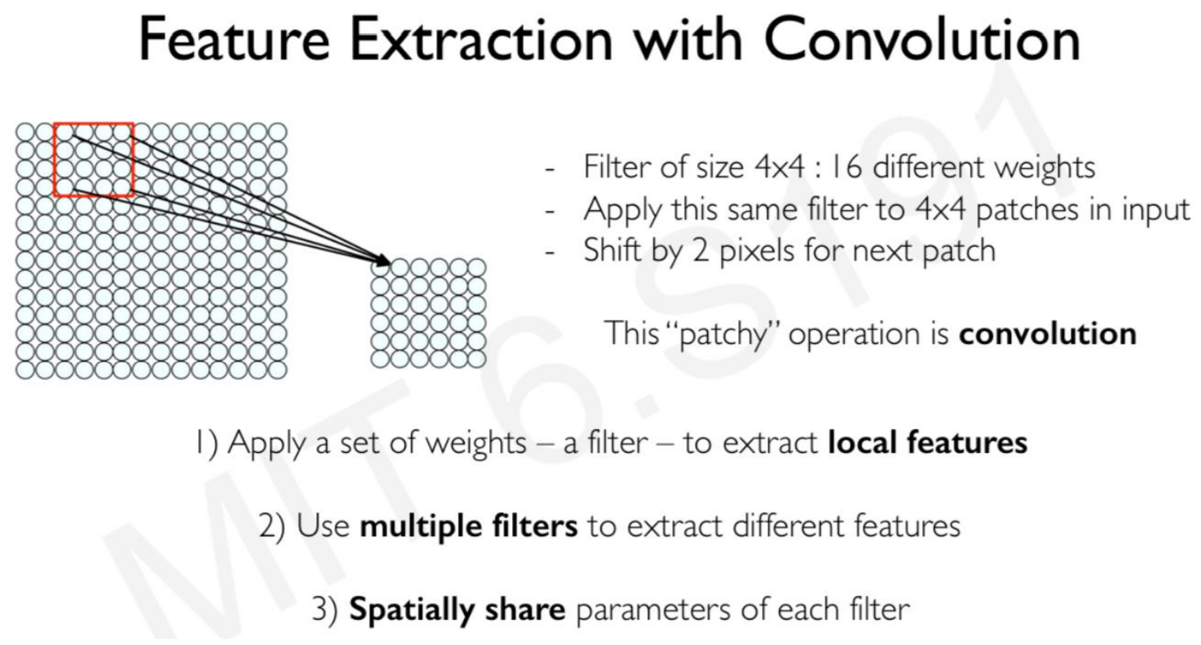
- 

Applying Filters to Extract Features

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
(features that matter in one part of the input should matter elsewhere)



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

-

❖ Linear filters

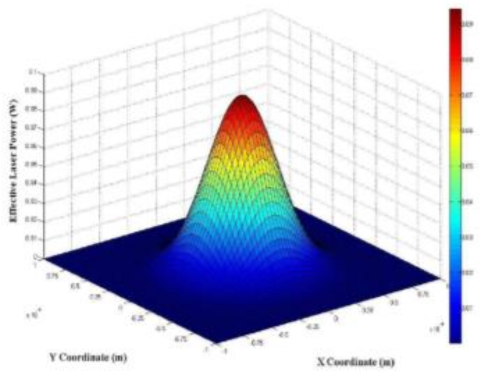
These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

- Derivatives are example of linear filters
- Blurring (mean), gaussian smoothing, edge detection, sharpening, embossing ++
- Properties-
  - O/P is a linear transformation
  - Shift of input results in shifts of O/P
  - Linear filter can be combined

- **Convolution and Correlation**
  - Correlation - Correlation is measurement of the similarity between two signals/sequences.
  - The kernel is flipped before taking products ~ cross correlation.
  - if kernel is symmetric – Conv and correlation are same.

- Gaussian smoothening
  - Filters for CNN that is used for removing Noise via smoothening. The weightages of the kernels are distributed in such a way that the central pixels get maximum weightage while the pixels are the corner receive minimum weightage.
  - Good in removing noise, sharper than the box filter (average filter)

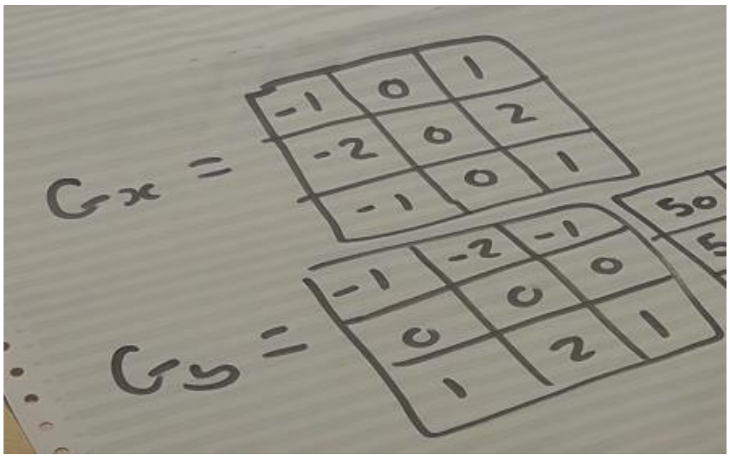
$$K = \begin{bmatrix} .02 & .08 & .14 & .08 & .02 \\ .08 & .37 & .61 & .37 & .08 \\ .14 & .61 & 1 & .61 & .14 \\ .08 & .37 & .61 & .37 & .08 \\ .02 & .08 & .14 & .08 & .02 \end{bmatrix}$$

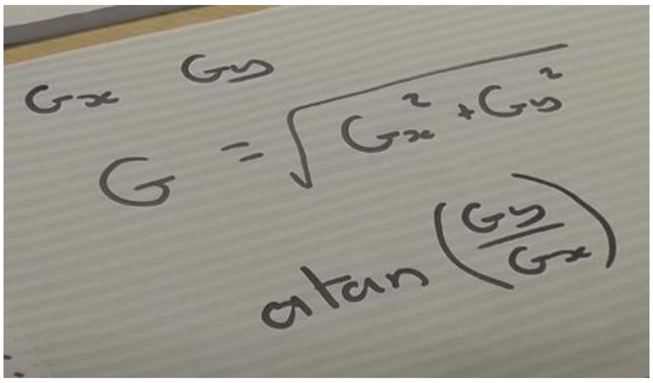


- Kernel Convolution to detect Edges

  Sharp changes in color indicate the edges. We use Sobel to detect the edges
  - **Sobel operator**
    - The pixels are in X and Y directions. So we have special filters for the X directions and Y directions. Left is in -, middle in 0, right in +ve for X direction filter. It can detect edges horizontally.
    - Now similarly we can calculate for Y axis.

- To get information about the edges as hypotenuse, we can calculate the square root of sum of squares of values obtained from the X and Y filters. We can even get the direction of the edge in terms of angle if we divide the gradient result of Gy over Gx.


$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$


G_x G_y
$G = \sqrt{G_x^2 + G_y^2}$
$\operatorname{atan}\left(\frac{G_y}{G_x}\right)$

- But Sobel produces edges for the noisy points also. It is very sensitive to such points, so it is better to take gaussian first and then perform the Sobel operator.

➢ **Smooth derivatives:**

- At the very least, this implies that the function is continuously differentiable (i.e. the first derivative exists everywhere and is continuous).
  - Start smoothing (e.g. with a Gaussian filter G_σ) and then derive:

    $$\frac{\partial(G_\sigma * f)}{\partial x}$$

  - properties of differentiation of convolutions

    $$\frac{\partial(g * f)}{\partial x} = \frac{\partial g}{\partial x} * f$$

  - so we can directly convolve with the derivative of the gaussian

    $$\frac{\partial(G_\sigma * f)}{\partial x} = G'_\sigma * f$$

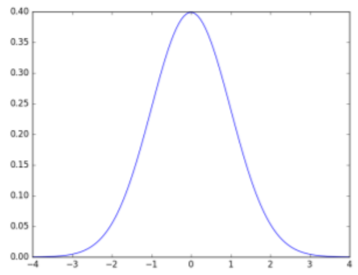
-

❖ **First order derivatives:**

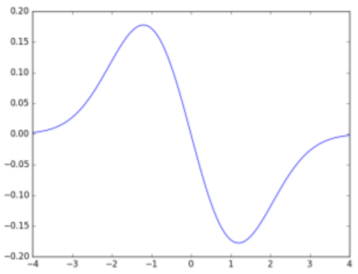
used as optimization method in deep learning is the first-order algo- rithm that based on gradient descent (GD). It is about finding a local minimum of a differentiable function.

❖ **Second order Derivatives:**
Second-order optimization technique is the next step of first-order optimization in neural networks which gives information about the direction. By using second order methods on the entire training data will give us a fairly accurate estimate of the descent direction and magnitude. If the secord order dervi is +ve, it means we are moving on convex surface.

$$G_\sigma = e^{-\frac{x^2}{2\sigma^2}}$$



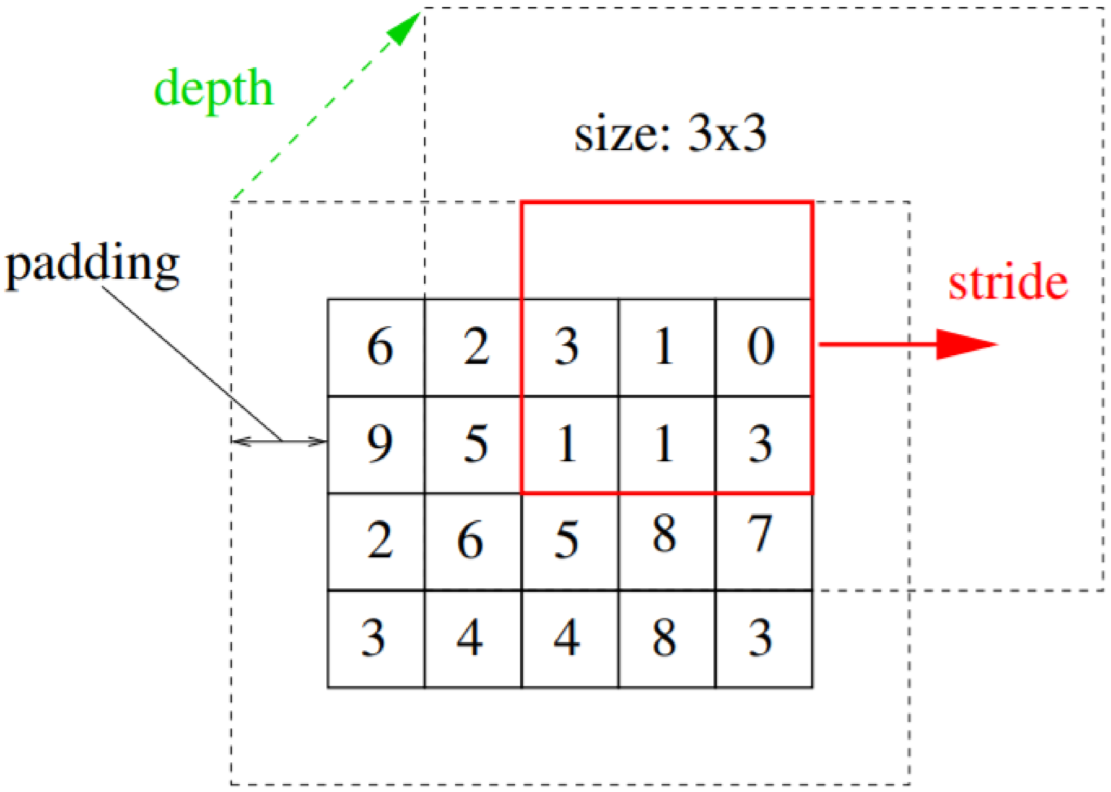
$$G'_\sigma = -\frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}$$



▪

❖ In CNN:
- ➢ Instead of using pre defined filters,, let them learn it's own filters.
- ➢ Important for deep architectures.

❖ **Parameters of Convolutional layers**
- ➢ Kernel size – dimension of linear filter
- ➢ Stride – movement of the linear filter | Stride ↑, overlap ↓
- ➢ Padding- at the borders to allow application of filters
- ➢ Depth – number of filters we wish to synthesize. Each neuron will be connected to same portion of I/P



-

- Dimension of O/P

$$\frac{W+P-K}{S}+1$$

  - width , padding, kernel , stride
  - Addition of Activation function does not change the dimensions of the O/P
- Padding
  - Valid – no padding
  - Same – apply minimal to enable calculations

-

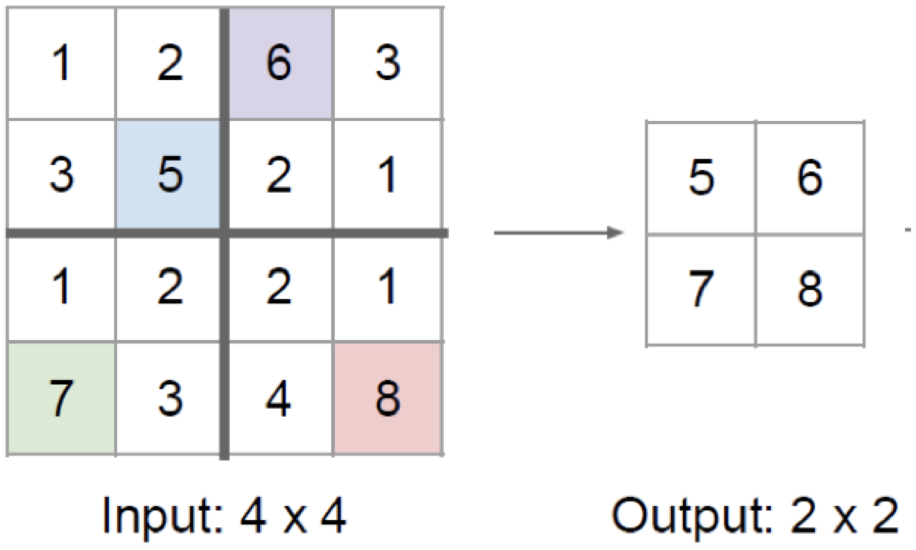
- Filter operates on all i/p channels in | |
- i/p depth – D, kernel size – NxM:
  - the dimension of filter is NxMxD
  - the convolution kernel maps cross-channel and spatial correlations

-

POOLING

- In deep cnn, we alternate convolution layers with the pooling layers, neach neuron takes the mean or maximal value in its receptive field
    - Double advantage- reduces the dimension of o/p + tolerance to translations

- Max Pooling example

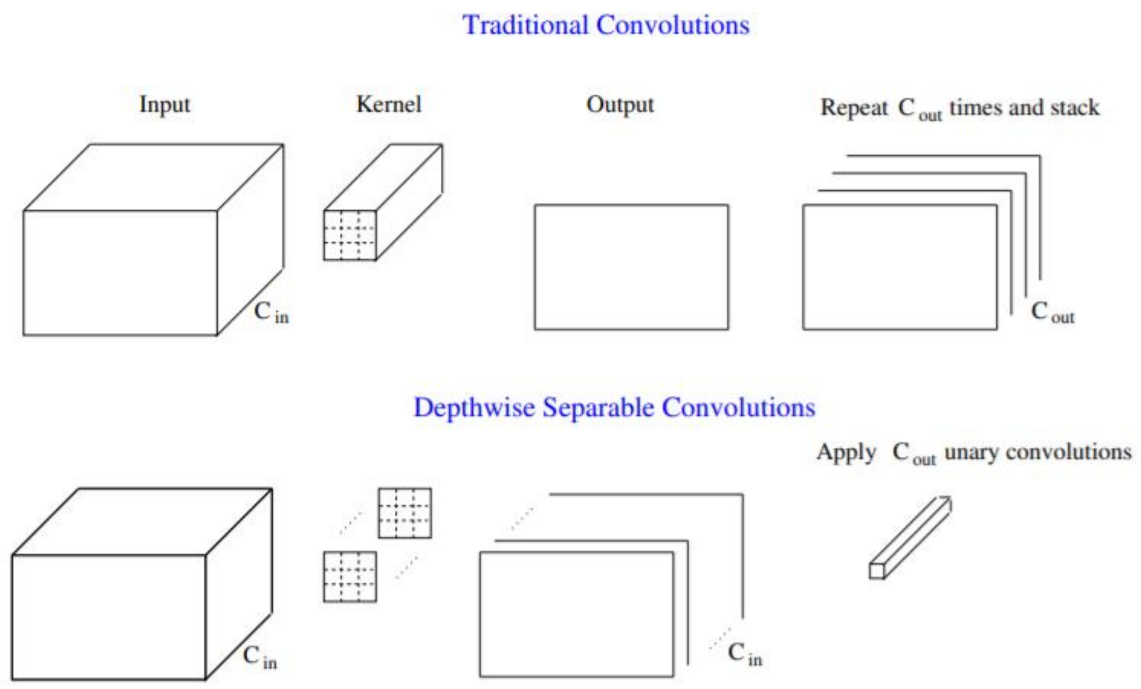


-
-
-
- Receptive Field
    - Dimension of the i/p region influencing it
    - A neuron cannot see anything outside itss receptive field
-
- Real examples of CNNs
    - AlexNet
    - VGG
    - Inception V3
        - Normal conv kernels are 3D simulateously mapping cross-channel correlations and spatial correlations.

- Inception modules – intermediate steps b/w a regular conv and depthwise separable convolution. [depthwise conv followed by pointwise conv]

•

❖ Depthwise seperable convolutions


Traditional Convolutions
Input
Kernel
Output
Repeat C_{out} times and stack
C_{in}
C_{out}
Depthwise Separable Convolutions
Apply C_{out} unary convolutions
C_{in}
C_{in}

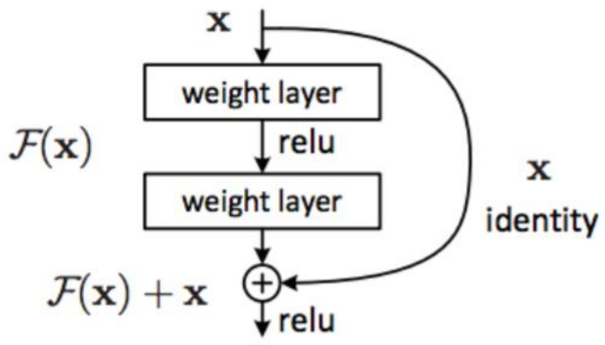
➢

➢ Traditional conv:
  - 3x3 kernel
  - 16 i/p channel
  - 32 o/p channel
  - So i/p is convolved 32 times with different kernels of dimension 3x3x16 = 144.
  - So total parameters = 32*144 = 4608.

➢ Depth wise Separable conv:
  - 3x3 kernel
  - 16 i/p channel
  - 32 o/p channel
  - We apply 32 different kernels with dimension 1x1x16 = 16
  - So total parameters = 16x3x3 + 32x1x1x16 = 656

-

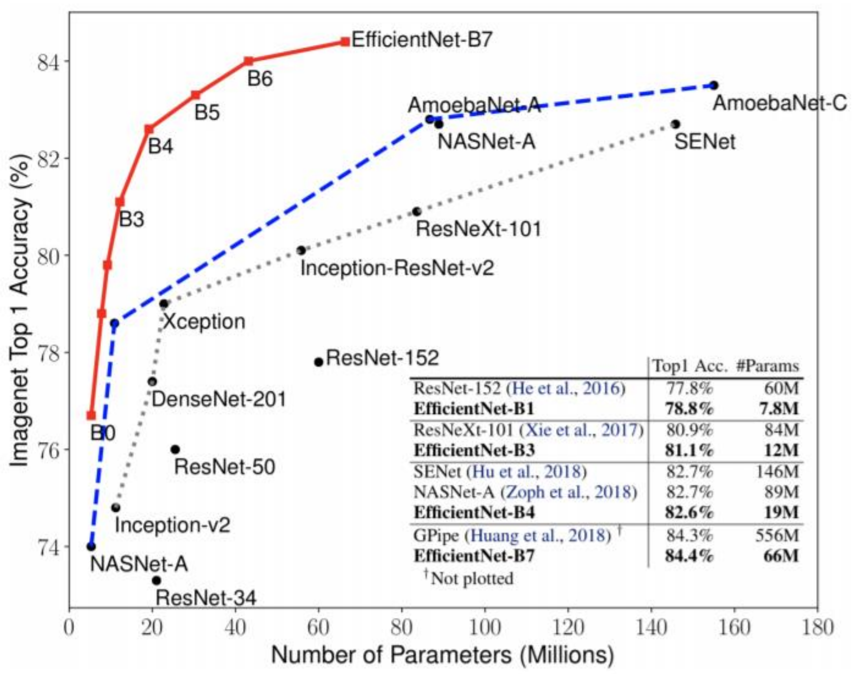
- Xception and MobileNet
- Residual Learning – instead of learning F(x) we learn F(x) + x
  - We add residual connection every 2-3 layers
  - Example: Inception Resnet
  - Working:
    - During BP, the gradient at higher layers can easily pass to lower layers without interacting wth the weight layers that might cause vanishing gradient/exploding gradient problem.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

- **Exploding gradients** are a problem where large error gradients add up and result in very large updates to neural network model weights during training. This causes the model to become unstable and unable to learn from your training data.
  - Common solution to exploding gradients is to change the error derivative before propagating it backward through the network and using it to update the weights. By rescaling the error derivative, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow.

-

- ❖ Efficient Net
  - ➢ ConvNets essentially grow in 3 directions
    - ▪ Layers
    - ▪ Channel
    - ▪ Resolution
  - ➢ Does scaling ConvNets improves accuracing and efficiency : https://arxiv.org/pdf/1905.11946.pdf



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**78.8%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.1%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.6%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

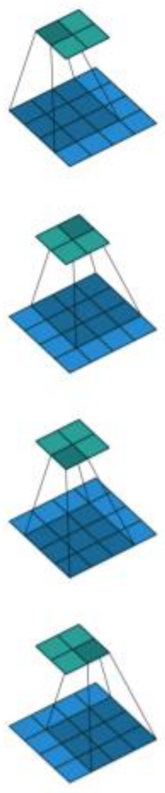
†Not plotted

- ===WEEK 9 - CNN + BP + Transposed conv + transfer learning

BP FOR CNN

- ➢ Study material : link

- I/p and O/p are unrolled into vectors from L to R.
- Operation performed by the conv can be seen as single dense n/w with 16 i/p and 4 outputs. Wi,j is kernel weight.
- In BP, we compute weights updates as usual. Updates relative to same kernels must be shared eg. Take mean among all updates [blue boxes]

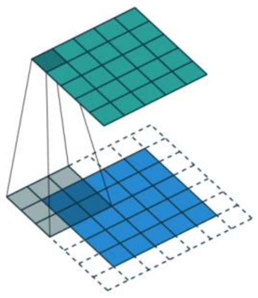


$$\begin{pmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{pmatrix}^T$$

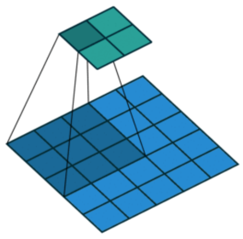
$$\begin{pmatrix} \boxed{w_{0,0}} & 0 & 0 & 0 \\ w_{0,1} & \boxed{w_{0,0}} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & \boxed{w_{0,0}} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & \boxed{w_{0,0}} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \end{pmatrix}^T$$

TRANSPOSED CONVOLUTIONS (DECONVOLUTION):

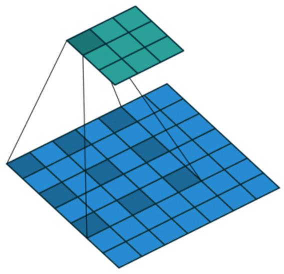
Before we talk about TC, lets know about Upsampling and Downsampling first
A downsampling convolutional neural attempts to compress the input, while an upsampling one tries to expand the input.
There are various waus to perform these actions: padding, strides & dilations.



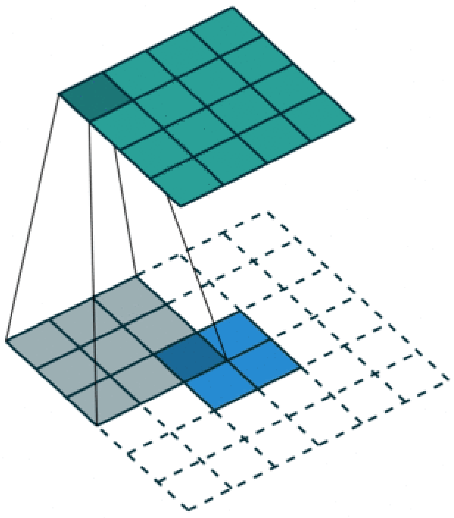
Generally the CNN are downsample the image when performed.



- ❖ To increase output dimensions, **padding** is usually used.
- ❖ **Strides** control how many units the kernel slides at a time. A high stride value can be used to further compress the output. The stride is usually and implicitly set to 1.
- ❖ **Dilations** can be used to control the output size, but their main purpose is to expand the range of what a kernel can see to capture larger patterns.



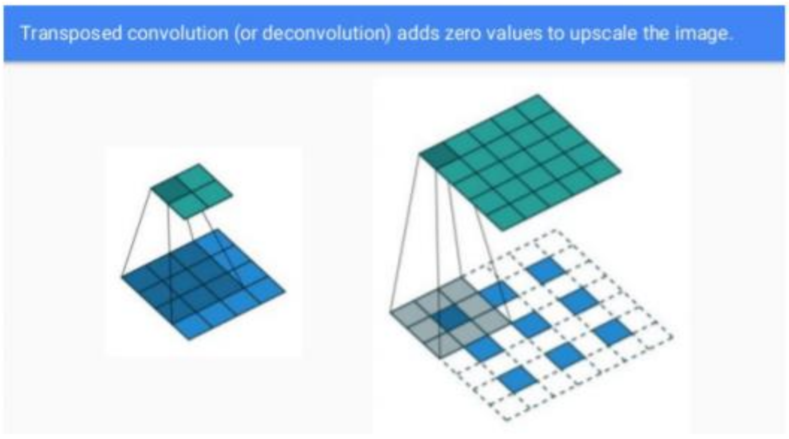
Lets talk about TC now, It is a method of unsampling. We use TC in auto encodes and GANs, or generally any network that deals with reconstructing images.



- ➢ Transpose- switching the places between each other and when we talk about convolution NN, the input and output dimensions are switched.
- ➢ Here, when the TC is performed the input of img is larger than the output.
- ➢ The input is padded in such a way that the corner kernel are just in touch with the corner of the input.

- ❖ TC as fractionally strided
  - ➢ TC (deconvolution) ~ normal convo with sub unitarian stride

To mimic subunitarian stride, we must first properly upsample the input (e.g. inserting empty rows and columns) and then apply a single strided convolution:


Transposed convolution (or deconvolution) adds zero values to upscale the image.

-
- By simple transposing the matrix we may convert the dimesionsion of i/p into the dimension of o/p.
- So kernel defines a conv matrix. the way matrix is applied decides whether direct convolution or transposed convolution.

❖ **Dilated Convolutions – Atrous Convolution**: link

- Atrous convolution is an alternative for the down sampling layer. It increases the receptive field whilst maintains the spatial dimension of feature maps. We deal with rate and pads here. When pad = 2, we pad 2 zeros at both L & R sides. When rate = 2 then we sample the input signal every 2 input for convolution.
- Useful in the first layers while on high resolution images

❖ **Temporal CN**

- It uses casual convolutions and dilations so that it is adaptive for sequential data. This way it can handle long ip sequences with its temporality and large receptive fields.

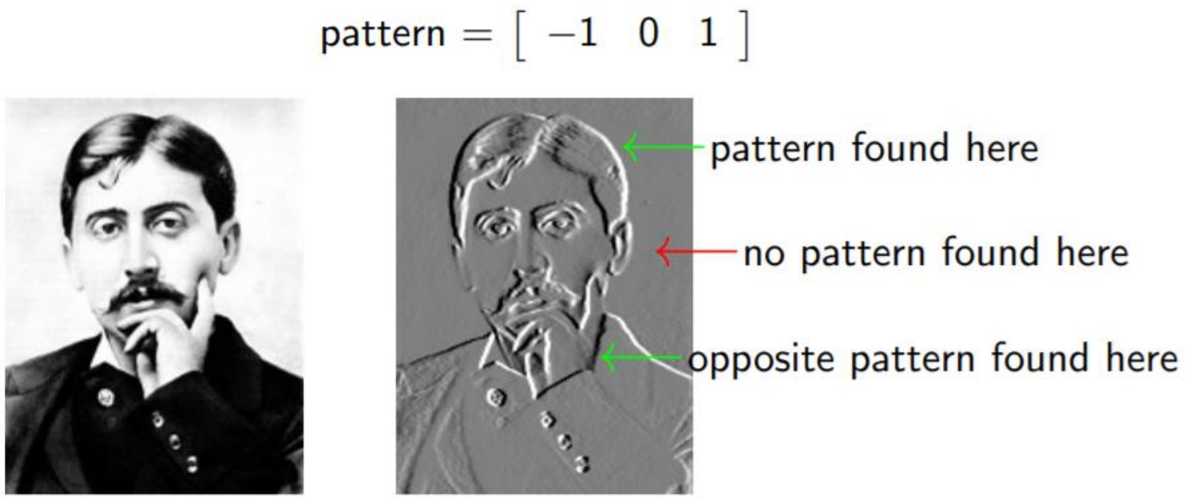
TRANSFER LEARNING

- ➢ It is a machine learning process used, that focuses on storing the knowledge gained while solving one problem and then applying it to a different problem. We should remember that both the problems should be related to each other. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks.
- ➢ Why do we do so? When we have less data of our specific problem, we use transfer learning. Since the model we will use is already trained on similar kind of data, it can move on with lesser amount of data.
- ➢ Transfer learning can be used to **accelerate the training of neural networks** as either a weight initialization scheme or feature extraction method.
- ➢ The weights in re-used layers can be used as the starting point for the training process for the new problem. This is also called as transfer learning as a type of weight initialization scheme. More useful when the first related problem has a lot more labelled data than the our problem which is similar to the first.

  •

- ==
- SLIDE 10
- How CNN see the world : link
-

❖ **Pattern:**



-

- **Complex (deep) patterns**
  - Neurons at the higher layers should recognize complex patters – combination of previous patterns – over large receptive field.
  - In the highest layer, neurons start recognizing patters similar to the features of objects in dataset ~ eyes, house, feathers.
- **Visualization of hidden layers**
  - Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data. Each hidden layer function is specialized to produce a defined output.
  - We need the hidden layers are required if and only if the data needs to be separated non-linearly.

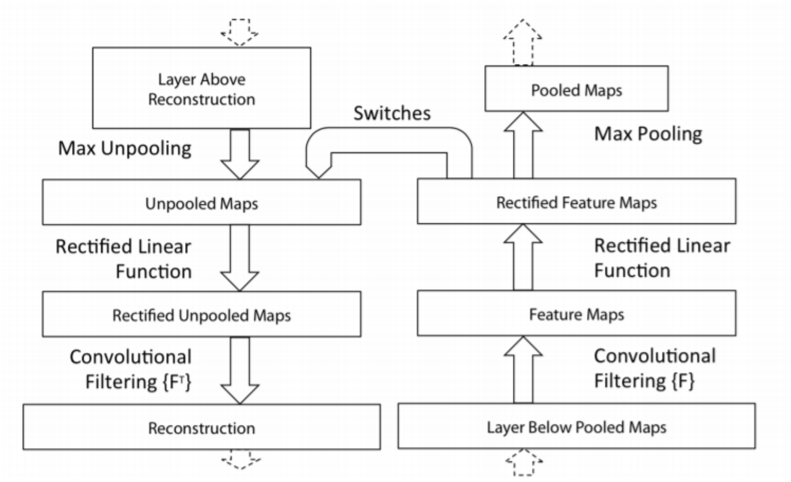
❖ **The Gradient Ascent technique:**

- If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent. Since in gradient Descent we had an aim to minimize the loss function. Here we look forward to maximising a particular function.
- Why do we need this? When we are treading the surface of function, we can face a convex or concave surface. If it is convex we use Gradient Descent and if it is concave we use we use Gradient Ascent. When we use the convex one we use gradient descent and

when we use the concave one we use gradient ascent. One interesting point is that, if I add a minus before a convex function it becomes concave and other way around is also true.

❖ **VGG**
- The network has 41 layers. There are 16 layers with learnable weights: 13 convolutional layers, and 3 fully connected layers.
- The number of filter we use is roughly doubling on every step

❖ Emphasization though deconvolution
➢ Instead of synthesizing an img maximizing the activation of the neuron, emphasize in real img what caused the activation.
➢ LINK
➢ Deconvolution- convolution with sub-unitarian stride
➢ Unpooing via switches – Boolean maps
•



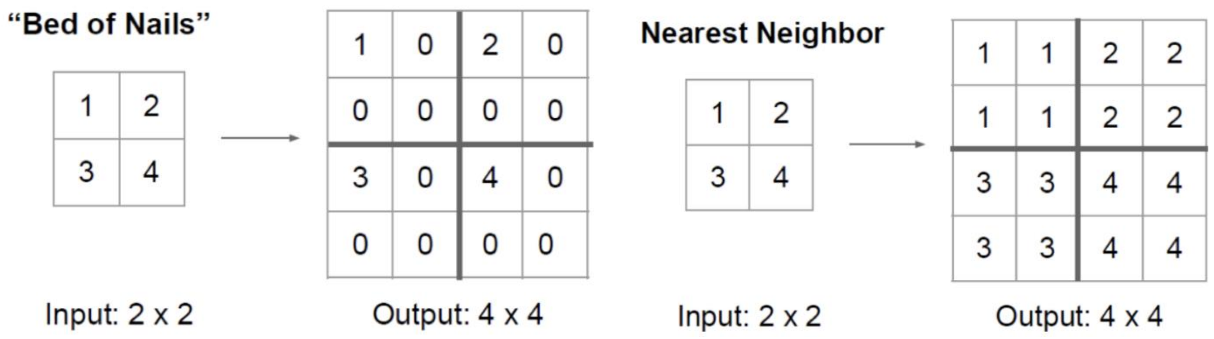
Deconvolution: In the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus.

❖ **Deconv architecture**
➢ Select img with strong activation for given neuron
➢ Zero out all activations for different neurons
➢ Go back to the img space, via deconv net.
➢ Conv kernel must be TRANSPOSED,
• Flipped horizontally and vertically.

❖ **Unpooling**: In the convnet, the max pooling operation is non-invertible, however we can obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of switch variables.

➢ Nearest neighbour method à

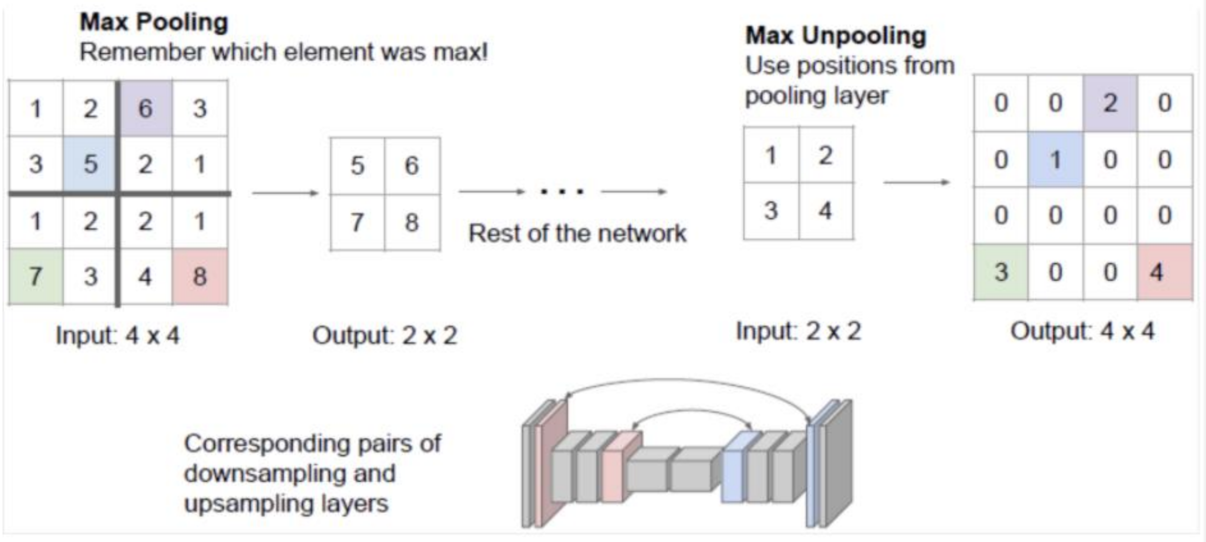
➢ Bed of nails:



•

➢ Max unpooling:

• Remember the positions while pooling:



Max Pooling
Remember which element was max!
Input: 4 x 4
Output: 2 x 2
Rest of the network
Max Unpooling
Use positions from pooling layer
Input: 2 x 2
Output: 4 x 4
Corresponding pairs of downsampling and upsampling layers

Figure 5. Illustration of Max Unpooling, from [11]

❖ **Matched filter:**

➢ Applying transposed kernel we emphasize the portion of img that caused activation.

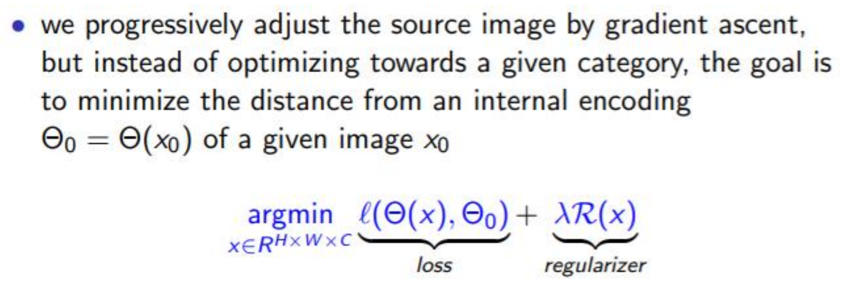
➢ Transpose of kernel -> *matched filter*.

-
- As we move towards higher levels we observe
- -Growing structural complexity
- -More semantic grouping
- -Greater invariance to scale and rotation

  ===========

❖ Comparison to other works:

- ➢ Objective is similar to encoding but
  - It works with a arbitrary n/w (classifier) not an encoder
  - We don't train NN, BP is used to reconstruct the image
  - Loss is measured on internal representation.



- we progressively adjust the source image by gradient ascent, but instead of optimizing towards a given category, the goal is to minimize the distance from an internal encoding $\Theta_0 = \Theta(x_0)$ of a given image x_0

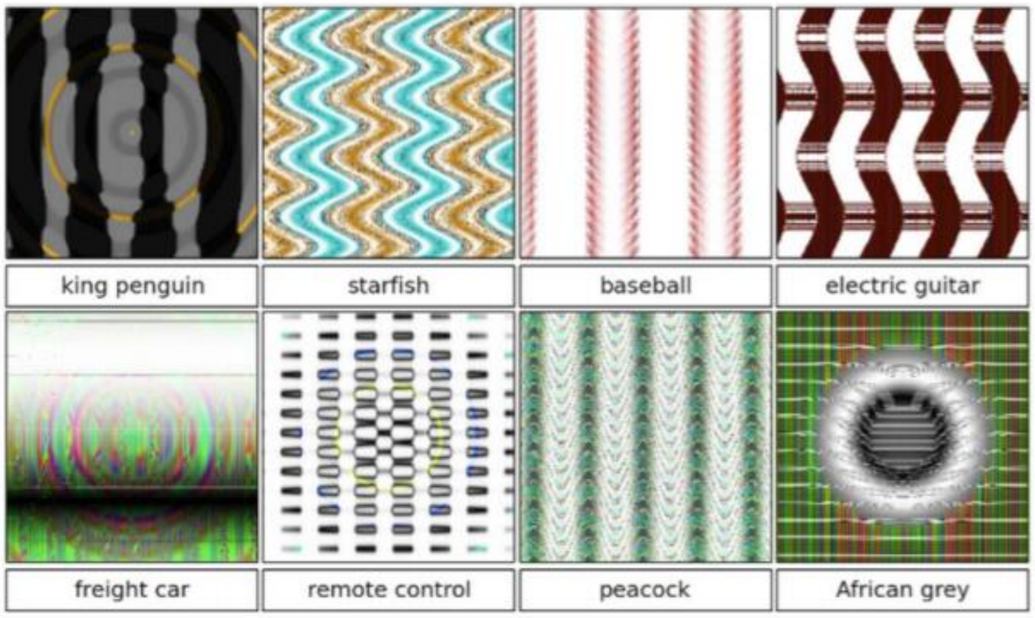
$$\underset{x \in R^{H \times W \times C}}{\operatorname{argmin}} \underbrace{\ell(\Theta(x), \Theta_0)}_{loss} + \underbrace{\lambda \mathcal{R}(x)}_{regularizer}$$

- ➢
- ➢ Results:
- ➢ The layers are invertible code, progressively become fuzzier.
  - Last layers invert back to multiple copies of object parts at different positions and scales.

❖ **Fooling NN**

- ➢ We can use gradient ascent to generate weird-looking images to maximize activation of a given unit
- ➢ This technique increase the score of whatever class we want

Related finding: it is easy to generate perceptually meaningless images that will be classified as any given class with high confidence



- ➢ We should be able to automatically sunthesize img of any kinds under consideration.
- ➢ Out of the many pixels, a tiny deviation is enough to fool the classifier; link

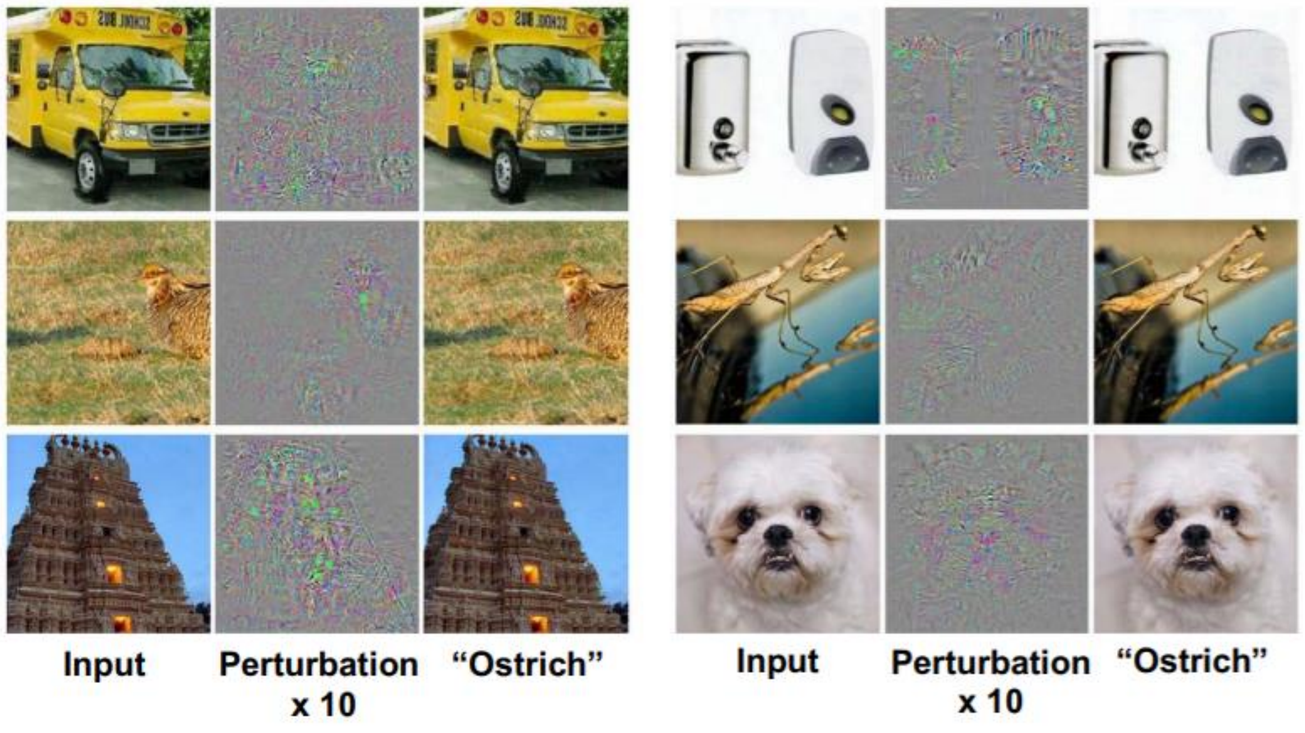
•

- ➢ Rather than searching for the smallest possible perturbation, it is easier to take small gradient steps in desired direction
- ➢ Is it possible to obtain similar result using network as black box

❖ DNN are easily fooled
- ➢ Evolutionary approach
  - ▪ Start with random population of images
  - ▪ Alternately apply selection (keep best) and mutation (random perturbation/crossover)

- When tested the results can be astonishing since the results are no where near the expectations.

- Sample results:



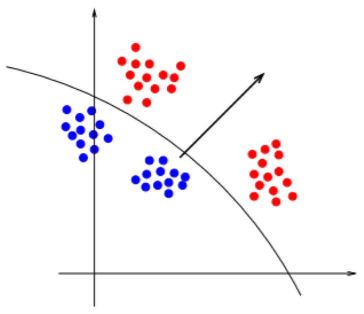
❖ Imgage encoding:
- ➢ Direct encoding- explicit representation of img as array of pixel
- ➢ Indirect encoding – implicit representation of img as composition of regular function – sine, gaussian, sigmoid, linear etc

MANIFOLD

- ➢ manifold is an object of dimensionality d that is embedded in some higher dimensional space. Imagine a set of points on a sheet of paper. If we crinkle up the paper, the points are now in 3 dimensions. Many manifold learning algorithms seek to "uncrinkle" the sheet of paper to put the data back into 2 dimensions.

- Natural imgs have many manifold with low dimensionality.
- In high dimensional space – easy to move away from the manifold of a given category.
- Once we have a manifold to describe your data, you can make predictions about the remaining space

➢ **Manifold issue**



- Due to correlation bw features, datapoints occupy small and specific areas of space.
- Moving away from a class the probability to end up in a meaning point can be very low.

❖ **Inceptionism**

➢ Deep dream & Inceptionism

- Intention to see what dNN is seeing when given a image
- It evolved into a new form of art

➢ Deep dreams

- Train a nw for image classification
- Revert the nw to slightly adjust (via BP) the original img to improve the activation of specific neuron
- After enough reiteratins, img will be incepted by the desired features, creating psychedelic and surreal effects.
- Generated imgs take advantage by strong regularizers privileging ip that have statistics similar to natural images ~ correlation b/w neighbouring pixels

➢ Enhancing the content

- Instead of prescribing the feature we want to amplify, we can also fix a layer and enhance whatever is detected.
- Each level deals with diff level of abstraction

➢ Style Transfer à

- define two distance functions, one that describes how different the content of two images are, Lcontent, and one that describes the difference between the two images in terms of their style, Lstyle. Then, given three images, a desired style image, a desired content image, and the input image (initialized with the content image), we try to transform the input image to minimize the content distance with the content image and its style distance with the style image.

- The gradient ascent technique can also be used to mimic artistic style

---

❖ Capturing Style

We know that at layer ℓ an image is encoded with D^ℓ distinct feature maps F^ℓ_d, each of size M^ℓ (width times height).

$F^\ell_{d,x}$ is thus the activation of the filter d at position x at layer l.

Feature correlations for the given image are given by the Gram matrix $G^\ell \in \mathcal{R}^{D^\ell \times D^\ell}$, where $G^\ell_{d_1,d_2}$ is the dot product between the feature maps $F^\ell_{d_1}$ and $F^\ell_{d_2}$ at layer ℓ:

$$G^\ell_{d_1,d_2} = F^\ell_{d_1} \cdot F^\ell_{d_2} = \sum_k F^\ell_{d_1,k} \cdot F^\ell_{d_2,k}$$

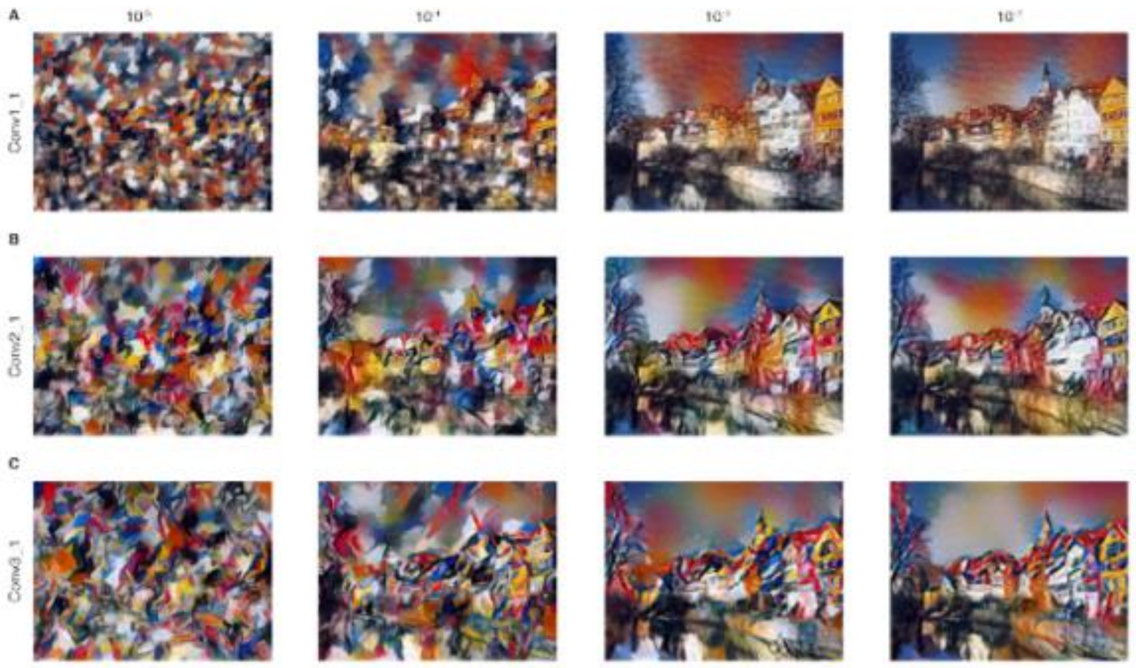
This is just yet another internal representation of the image, useful to encode style.

-
-

❖ Combine style and content

➢ Content and style are separable.

- Different combinations varying the reconstruction layer (Rows) and relevance ratio b/w styles.



-

❖ Other approaches to style transfer
  - Technique based on cycle gans
  - Universal transform via Feature transforms [link]
    - Takes i/p img, a style to mimic, and adapts content to given style.

- ==

L11. SEGMENTATION

- Classify each pixel of an image into a category. A group where it belongs to
- For example, in an image that has many cars, segmentation will label all the objects as car objects. However, a separate class of models known as instance segmentation.
- Instance segmentation is the task of detecting and delineating each distinct object of interest appearing in an image.
- Slightly more fine-grained task of segmenting the object

-

❖ **Semantic Segmentation**
  - the process of linking each pixel in an image to a class label.
  - we label specific regions of an image according to what's being shown.. Different categories have different colours. Used on Autonomous driving. Industrial inspection.
  - It is different from object detection as it does not predict any bounding boxes around the objects. We do not distinguish between different instances of the same object. For example, there could be multiple cars in the scene and all of them would have the same label

- **Convolutionalization:**

❖ Composition of conv is conv

❖ Stride of compound conv is product of strides of components.

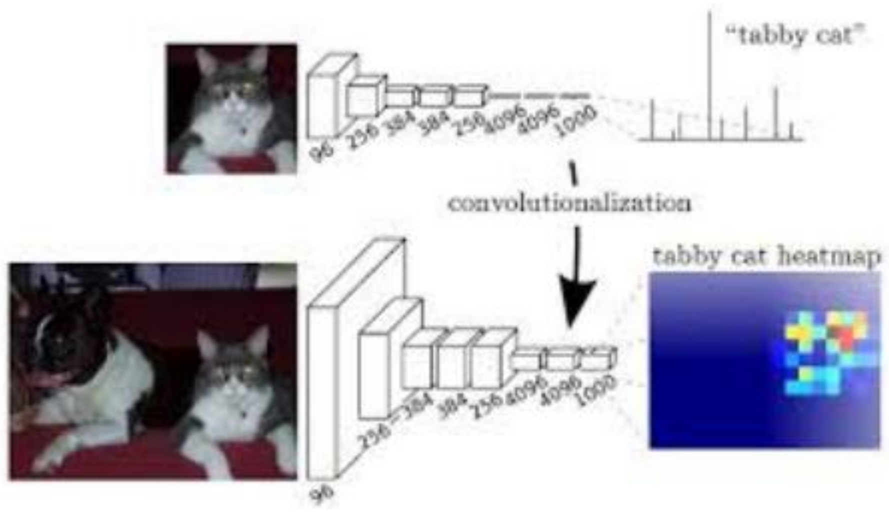
❖ Dimension of compound kernel:
  - Kernel dimension: $D_{in} = S * (D_{out} - 1) + K$
    - Intermediate dimension = Stride * (dimension out - 1) + K
  - Example:
    - ??

❖ *What breaks convo?*
  - If there are dense layers at the end of networks [if maxpooling has a fixed pooling dimension]

❖ Idea
  - Neurons in dense/ conv layers
  - HAVE THE same functionalities.
  - compute weighted sum of inputs

  - Dense layers are convo with large filters (= input size)

- So, each DenseLayer -> turn into -> conv layer (reuse same weights) -> conv network.

- If -> img classification -> on fixed size imgs, we get DCN that:
  - Takes input img of arbitrary dimension.
  - Produces output heatmap of activation of different obj categories.
    - This heatmap is relative to different locations of input img upon which DCN was convolved.
    - 

- InceptionV3 as single conv
  - Whole convnet ~ single convo
  - Stride? Kernel size?
    - Stride is product of all strides $2^5 = 32$
    - Kernel size D_{in} = (((((Dout -1)*2+2)*2+2)*2+4)*2+4)*2+3 = 32 Dout + 43

- Big stride we will have rough localization
  - Kernel = 75, stride = 32
  - Suppose we have ip dimension Din -> dimension of heatmap = ((Din -75)/32) +1
  - stride of 32 limits the scale of details:

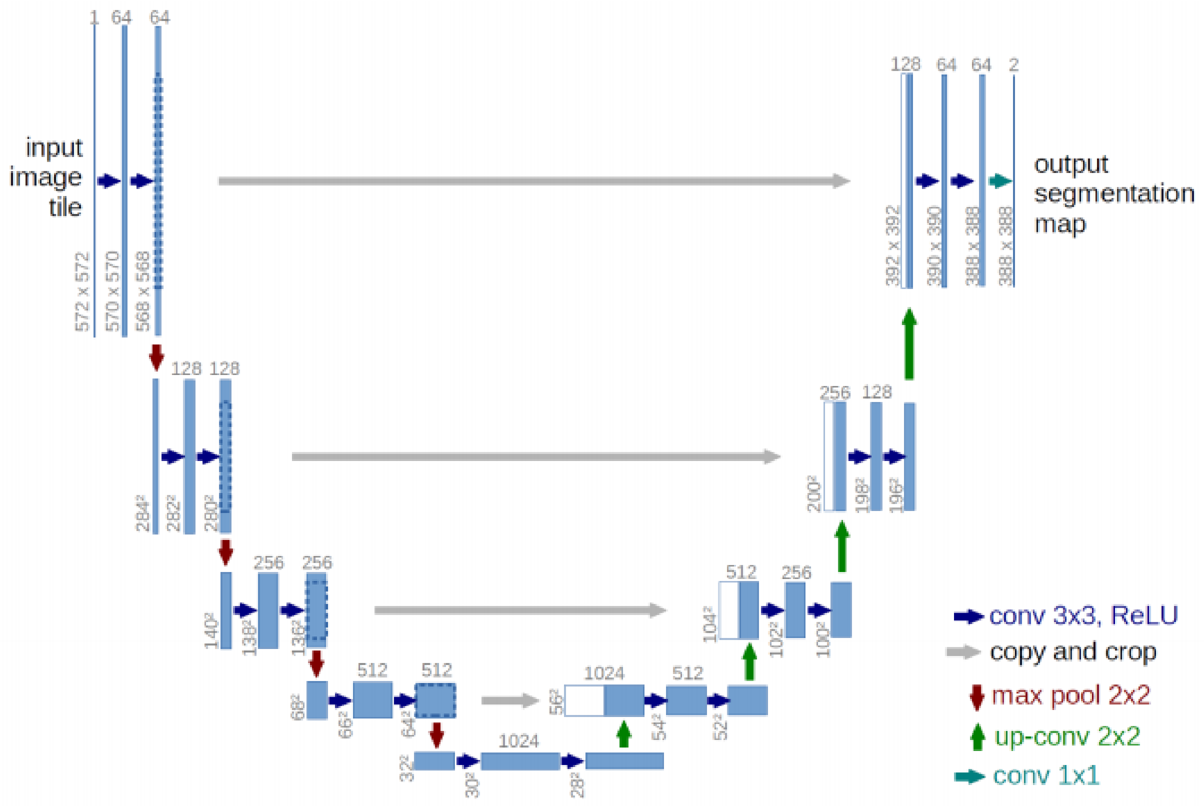
- ➢ Shelhamer added the skipped pixels, it combines the final prediction layer with lower layers of finer strides.

•

U-net:

architecture had two main parts that were encoder and decoder. The encoder is all about the covenant layers followed by pooling operation. It is used to extract the factors in the image. The second part decoder uses transposed convolution to permit localization. It is again an F.C connected layers network.

- ❖ Learns segmentation in end to end configuration
- ❖ Required less training examples. [data augmentation]
- ❖ Precise results
- ❖ No need of classification network



- Here you try to downsample the image first i.e from 572*572 then you reduce the final to 32*32 here you have the max pool layer and conv layer 3*3
  Again you try to upsample the image with the help of auto encoders to the original image size .
  The gray lines are the horizontal connections.

it only contains Convolutional layers and does not contain any Dense layer because of which it can accept image of any size.

Encoder (left side): It consists of the repeated application of two 3x3 convolutions. Each conv is followed by a ReLU and batch normalization. Then a 2x2 max pooling operation is applied to reduce the spatial dimensions. Again, at each downsampling step, we double the number of feature channels, while we cut in half the spatial dimensions.

Decoder path (right side): Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 transpose convolution, which halves the number of feature channels. We also have a concatenation with the corresponding feature map from the contracting path, and usually a 3x3 convolutional (each followed by a ReLU). At the final layer, a 1x1 convolution is used to map the channels to the desired number of classes.

===

L12: OBJECT DETECTION [OD]

❖ Object detection is a computer vision technique that allows us to identify and locate objects in an image.

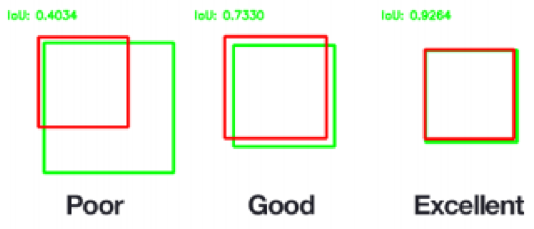
YOLO is an example of OD where an image feed it to the deep neural network(CNN) and the architecture is suppose to predict eight values .It gives you the coordinates of the box and what the box contains.

- Bounding box – which network predicts –predicted value
- Golden box – true value

<u>**Anchor boxes**</u> are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes we want to detect and are typically chosen based on object sizes in our training datasets.

- ❖ Similar to segmentation, but OD returns a bounding box containing the object.
- ❖ Training an end-to-end model -> difficult.
- ❖ No loss function is there.

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}$$



Intersection over union [Jaccard index]:

- ❖ A method of evaluating the bounding boxes: comparison between ground truth & calculated bounding box.

- Dataset for OD:
- ❖ PASCAL visual object classes
- ❖ Coco

Deep OD:

- ❖ Region proposal methods:
  - ➢ Derived via Selective Search algo
  - ➢ Selective Search algo
    - ▪ identify possible locations of interests

        - exploit texture and structure of img, object independent
    - RCNN, Fast RCNN, Faster RCNN
- Single Shot methods:
    - Yolo, SSD, Retina-net, FPN
-

YOLO – YOU ONLY LOOK ONCE

- In region proposal classification networks (fast RCNN) sometimes it perform detection on various region proposals and thus end up performing prediction multiple times for various regions in a image.
- Yolo architecture is more like FCNN (fully convolutional neural network) and passes the image (nxn) once through the FCNN and output is (mxm) prediction.
- This is an algorithm that detects and recognizes various objects in a picture (in real-time). Object detection in YOLO is done as a regression problem and provides the class probabilities of the detected images.

Why the YOLO algorithm is important

YOLO algorithm is important because of the following reasons:

- **Speed:** This algorithm improves the speed of detection because it can predict objects in real-time.
- **High accuracy:** YOLO is a predictive technique that provides accurate results with minimal background errors.
- **Learning capabilities:** The algorithm has excellent learning capabilities that enable it to learn the representations of objects and apply them in object detection.

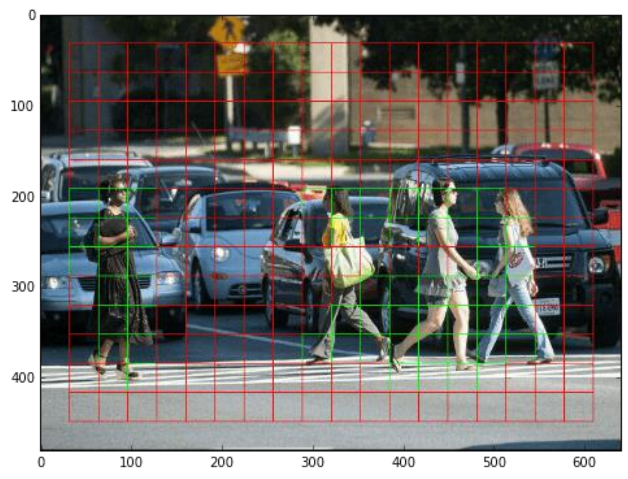
HOW THE YOLO ALGORITHM WORKS

YOLO algorithm works using the following three techniques:

- Residual blocks
- Bounding box regression
- Intersection Over Union (IOU)

RESIDUAL BLOCKS

First, the image is divided into various grids. Each grid has a dimension of S x S. The following image shows how an input image is divided into grids.



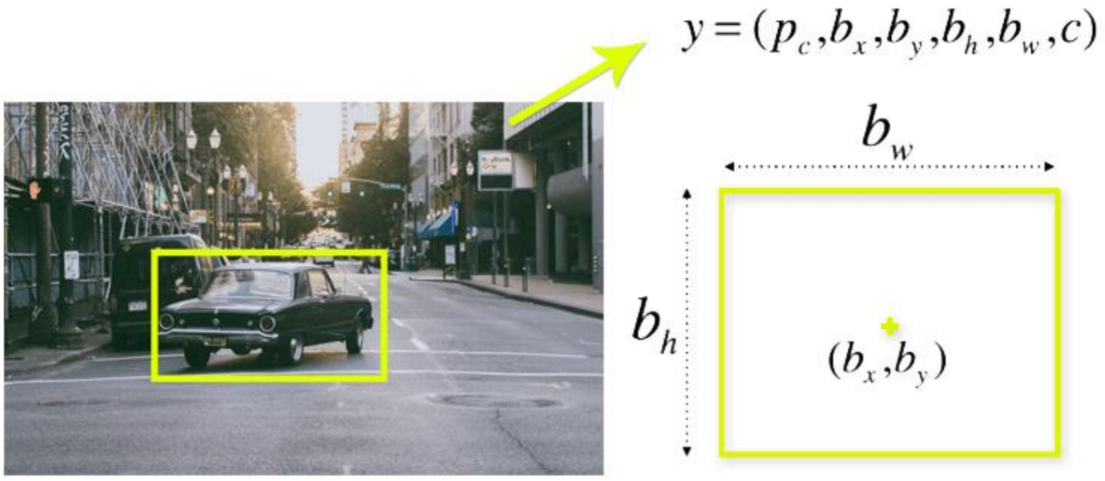
BOUNDING BOX REGRESSION

A bounding box is an outline that highlights an object in an image.Every bounding box in the image consists of the following attributes:

- Width (bw)
- Height (bh)

- Class (for example, person, car, traffic light, etc.)- This is represented by the letter c.
- Bounding box center (bx,by)

The bounding box is generally been represented by a coloured outline.



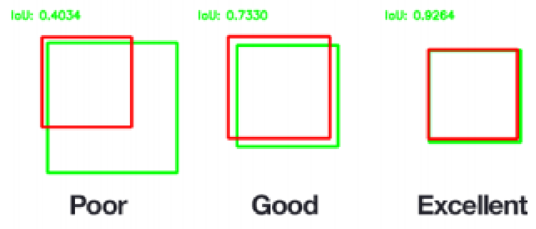
YOLO uses a single bounding box regression to predict the height, width, center, and class of objects. It also represents the probability of an object appearing in the bounding box.

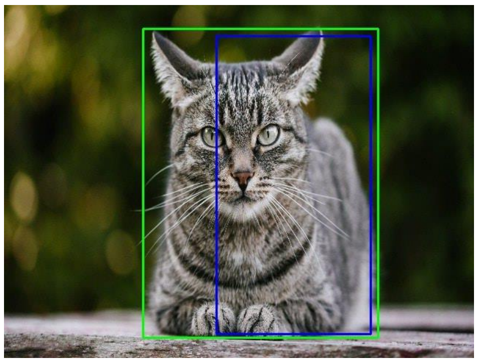
INTERSECTION OVER UNION (IOU)

Intersection over union (IOU) is a phenomenon in object detection that describes how boxes overlap. YOLO uses IOU to provide an output box that surrounds the objects perfectly.

Each grid cell is responsible for predicting the bounding boxes and their confidence scores. The IOU is equal to 1 if the predicted bounding box is the same as the real box. This mechanism eliminates bounding boxes that are not equal to the real box.

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

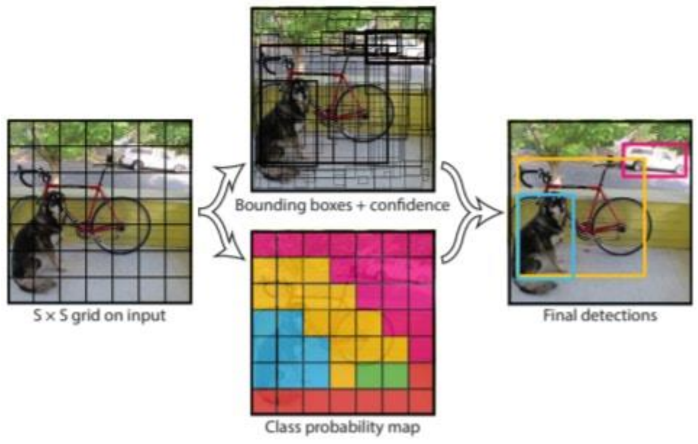




YOLO ensures that the two bounding boxes are equal

COMBINATION OF THE THREE TECHNIQUES

The following image shows how the three techniques are applied to produce the final detection results.



First, the image is divided into grid cells. Each grid cell forecasts B bounding boxes and provides their confidence scores. The cells predict the class probabilities to establish the class of each object.

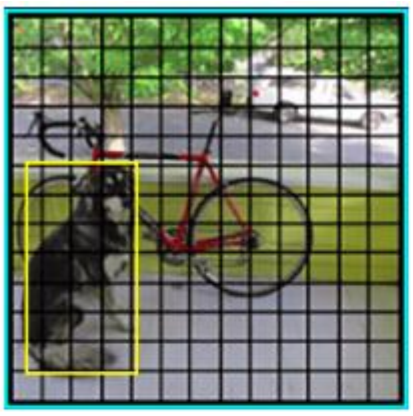
For example, if our image have three classes of objects: a car, a dog, and a bicycle. All the predictions are made simultaneously using a single convolutional neural network.

Intersection over union ensures that the predicted bounding boxes are equal to the real boxes of the objects. This phenomenon eliminates unnecessary bounding boxes that do not meet the characteristics of the objects (like height and width). The **final detection will consist of unique bounding boxes that fit the objects perfectly.**

For example, the car is surrounded by the pink bounding box while the bicycle is surrounded by the yellow bounding box. The dog has been highlighted using the blue bounding box.

1 convolution layer --> 2 FC layers

- OD that uses features learned by DCN to detect objects.
- Input img is processed a single time
- Fully conv network.
- Ip is down sampled by 2^5 (32 times smaller parts -> Feature maps)
- This downgrade gives positions of objects as well.



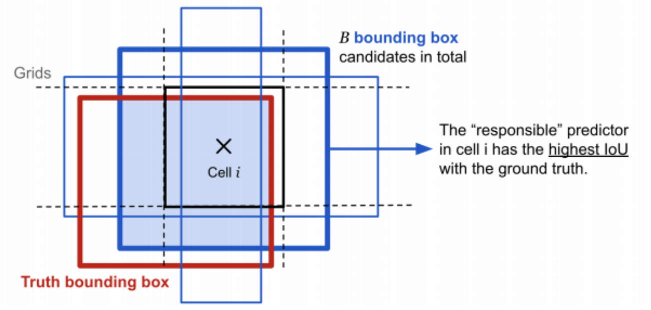
- Dimension of convo kernel -> Receptive Field of each neuron -> depends on n/w.
- Detecting obj. generally involves neurons outside the building box.
- Credit for detection goes to:
  - Single neuron is responsible for detection, it is located at the centre of bounding box.
  - Neuron makes finite no of predictions.
- Predictions:

- 13*13 neurons in feature map.
- Depth wise entries ->
    - bounding boxes * (5 + num. of categories)
- Each bounding box has 5 + C attributes
    - 2 for centre coord.
    - 2 for dimension.
    - 1 for objectness score.
    - C for confidence score.

➢ Anchor boxes:

- We can't directly predict the width and height of bounding boxes -> causes unstable gradients during training.

- Modern OD predict log space affine transforms for pre defined default bounding boxes -> anchors
- These transformations are applied to anchor boxes to get pred.


Grids
B bounding box candidates in total
Cell i
The "responsible" predictor in cell i has the highest IoU with the ground truth.
Truth bounding box

- YOLO v3 has 3 anchors -> prediction of 3 bounding boxes per cell.
- Anchors as selected by K means clustering on training set, reflecting the most likely shapes of bounding boxes.

➢ Making predictions:

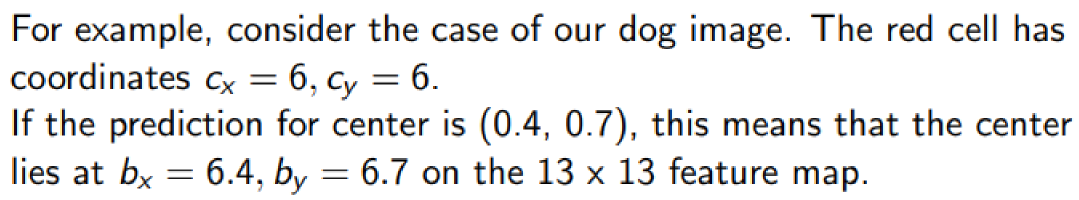
- The output gives following details:

$$b_x = c_x + \sigma(t_x)$$
$$b_y = c_y + \sigma(t_y)$$
$$b_w = p_w * e^{t_w}$$
$$b_h = p_h * e^{t_h}$$

- Tx, Ty, Tw, Th (cell coord)
  Cx, Cy (anchor dimensions)
-

➢ Centre cords:

- YOLO does not print exact bounding box centre dimensions. Instead it gives,
  - Relative to top left corner of grid cell which is predicting the objects.
  - Normalised by the dimension of cell from the feature map ~ 1


For example, consider the case of our dog image. The red cell has coordinates $c_x = 6, c_y = 6$.
If the prediction for center is (0.4, 0.7), this means that the center lies at $b_x = 6.4, b_y = 6.7$ on the 13 x 13 feature map.

LIMITATIONS OF YOLO

YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that our model can predict. Our model struggles with small objects that appear in groups, such as flocks of birds. Since our model learns to predict bounding boxes from data, it struggles to generalize to objects in new or unusual aspect ratios or configurations.

- *Our model also uses relatively coarse features for predicting bounding boxes since our architecture has multiple down sampling layers from the input image. Finally, while we train on a loss function that approximates detection performance, our loss function treats errors the same in small bounding boxes versus large bounding boxes. A small error in a large box is generally okay but a small error in a small box has a much greater effect on IOU.* ***Our main source of error is incorrect localizations.***

- Dimension of BoundingBoxes;
  - Predicted by applying log space transform to output & multiplying with anchor dimension.
  - Results: bw, bh, [normalised by height and width of img]
  - Training labels are chosen this way.
-
- Objectness Score
  - Prob of object contained inside the bouding box.
  - Obj score is passed through sigmoid -> to be interpreted as a probability.
-
- Class confidences
  - Probability of detected obj belonging to particular class.
  - Before v3, YOLO class scores were computed via Softmax.
  - Since, YOLOv3 multiple sigmoid function are used (considering possibility of objects belonging to multiple hierarchical categories).
-

❖ Yolo Loss Function:

- Two parts-
  - **Localization loss ->** BB offset prediction.
    v -> true value ; v^ -> predicted one -

$$\mathcal{L}_{loc} = \sum_{i=0}^{S^2}\sum_{j=0}^{B} 1_{ij}^{obj}[(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2+(\sqrt{w_i}-\sqrt{\hat{w}_i})^2+(\sqrt{h_i}-\sqrt{\hat{h}_i})^2]$$

    i -> traverses cells
    j -> traverses bb

  - **Classification loss ->** conditional class prob.
    Relative to object confidence & actual classification

- **Whole loss :** $\mathcal{L} = \lambda_{coord}\mathcal{L}_{loc} + \mathcal{L}_{cls}$

  Lambda coord – additional parameter that balances the contribution between localization loss & classification loss.
  In YOLO lambda coord = 5, lambda noObj = 0.5

MULTI SCALE PROCESSING

In this process we convolve the image with a Gaussian kernel at different σ sigma values, and in return we obtain different scale representation of the image. We need to do this detect the information from the image at different scales.

❖ **Image Pyramids:**

- used to build feature pyramids -> slow
- YOLOv1 moved from high scale to low -> detection of small object is bad ☹

❖ **Feature Pyramids Network:**

- feature extractor that takes a single-scale image of an arbitrary size as input, and outputs proportionally sized feature maps at multiple levels.
- The construction of FPN involves bottom-up pathway and top-down pathway.
- **Bottom-up pathway** is feed forward computation of backbone of convnet.
    - It computes the feature hierarchy that contains the feature maps at multiple scales. Here the step size of the scales is 2.

    - The output of last layer of each stage is used as reference set of feature maps.
    - The bottom up feature maps is lower level semantics but its activations are more accurately localized as it is subsampled lesser times.
- **Top-down pathway** has higher resolution features. It performs un-sampling of the features.
    - These features are later enhancement with a features from the bottom-up pathway via lateral connections.
    - Each connection merges the feature maps of same size from bottom up path and top-down path.
    - Steps**:**
        - Higher level features are unsampled.
        - Feature map direction bottom up -> pathway undergoes pathway reduction via 1*1 conv layer
        - Finally, these two feature maps are merged

- Single Shot Detector (SSD) reuse pyramid features hierarchy as ConvNet -> featurised image pyramids.
- FPN, RetinaNet, YOLOv3 recombines features via backward pathway.
    - Fast & Accurate
    - Feature maps have thin lines, semantically stronger features have thick outlines.

Non Maximum Suppression:

- Non-max suppression is the final step of these object detection algorithms and is used to select the most appropriate bounding box for the object. It solves the problem of multiple detection of same image, corresponding to different anchor & adjacent cells.
- ~~[YOLO v3 predicts feature maps at 13, 26, 52 scale.~~
- ~~At end we have $13^2 + 26^2 + 52^2 = 10647$ BB of dimension 85 [4 coord, 1 conf, 80 class]~~
- NMS takes 2 things into account-
  - **Thresholding by Object Confidence**:
    -We filter boxes based on confidence score, below threshold are ignored
  - **Overlap of bounding boxes** – IOU – [intersection over union]

  The non-max suppression will first select the bounding box with the highest objectiveness score. And then remove all the other boxes with high overlap.

  - Outline:

```
♦ Divide BB a/c predicted classes C.
♦ Each list BB is processed separately.
♦ Order BBc a/c to the confidence
♦ Initialize truePredictions to an empty list.
♦ While BBc is not empty:
      Pop the first element p from BBc
      Add p to truePredictions
      Remove from BBc all pred with IoU s.t p>th
  Return truePrediction
```

Ablation

- is the removal of a component of an AI system. An ablation study studies the performance of an AI system by removing certain components, to understand the contribution of the component to the overall system.

==

AUTOENCODERS: PCA

❖ Autoencoder – used to reconstruct input data of learned internal representation. An autoencoder is a special type of neural network that is trained to copy its input to its output. For example, given an image of a handwritten digit, an autoencoder first encodes the image into a lower dimensional latent representation, then decodes the latent representation back to an image.

Autoencoder is a type of neural network that can be used to learn a compressed representation of raw data. An autoencoder is composed of an encoder and a decoder sub-models. An autoencoder is a neural network model that can be used to learn a compressed representation of raw data.

➢ Internal representation has lower dimensionality. So it is good for compression.

Architecture of AE:

Consists of Encoder, Code & Decoder**.** There are 3 components.

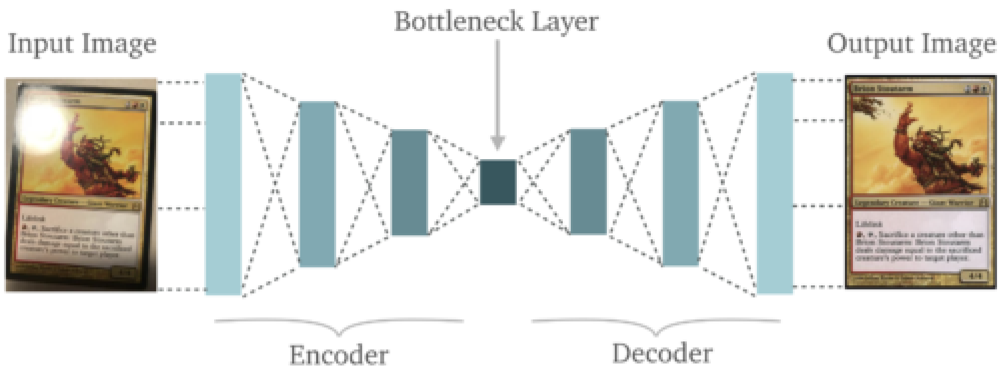
- **Encoder:** layer encodes the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
- **Code:** this part of n/w represent the compressed input which is later used as input to the decoder. Also known as bottleneck. It is designed in such a way so that it can decide which aspects of observed data are relevant information and what aspects can be discarded.
- **Decoder:** decodes the encoded image back to the original dimension. The decoded image is a lossy reconstruction of the original image and it is reconstructed from the latent space representation.

Properties of Autoencoders:

- Data-specific: Autoencoders are only able to compress data similar to what they have been trained on.
- Lossy: The decompressed outputs will be degraded compared to the original inputs.

Learned automatically from examples: It is easy to train specialized instances of the algorithm that will perform well on a specific type of input.

AutE have multiple uses:
AEncodders Can be used for multiple purpose.



❖ **Compression**

- Input data has regularities, we learnt the pattern between them and then perform the compression.
- Irregular data -> [high random] no compression is possible.

- Form of Data Compression (DC) – internal layer has fewer units of ip, AutoEncoders are a form Data Compresssion.
- **Compression types-**
  - Data specific: works only on data with strong correlation. Different from traditional DC algo
  - Lossy: o/p is degraded wrt i/p. Different from textual compression algo, s.a. gZip

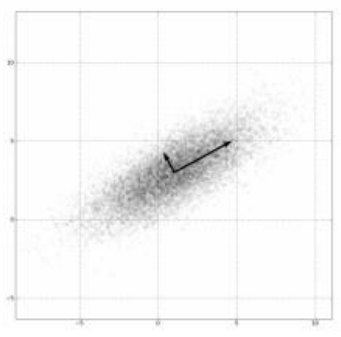
- **Autoencoders – pros:**
  - Data denoising
  - Dimensionality reduction
- **Cons:** They are Data-specific: Autoencoders are only able to compress data similar to what they have been trained on. Lossy: The decompressed outputs will be degraded compared to the original inputs.

PCA:

- Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis [transformation along certain axes] on the data, sometimes using only the first few principal components and ignoring the rest.
- PCA is used in exploratory data analysis and for making predictive models.
- commonly used for dimensionality reduction by projecting each data point onto only the first few principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible.
- n-dimensional space linearly projects the min lower dimensional space, minimizing the quadratic error of their reconstruction.

➢ We focus on instead of minimizing the error, it is equivalent to maximize the variance.

Maximizing Variance



- Since the distance of each point from centre remains constant, we can maximise the variance ~ minimizing the error.
- For the points, we have eigen vectors as axes.
- The eigenVector v, for a matrix A is such that there is scalar λ

$$Av = \lambda v$$

Co-variance

- With bi-dimensional data, the variance of individual componects x & y don't capture the variance of data. Why?
  Since, exchanging x1,y1 x2,y2 with x1y2, x2y1 has no change in variance of y & x.

Co-variance matrix:

- is a square matrix giving the covariance between each pair of elements of a given random vector. Any covariance matrix is symmetric and positive semi-definite and its main diagonal contains variances (i.e., the covariance of each element with itself).
- Expresses deformation of data. Way of distribution of data
- X dataset having n data * D dimensions (features)
- Covariance Matrix(X) – matrix (d*d)
  $var(X) = E[(X - E(X))^T (X - E(X))]$
  $var(X) = E[X^T X] = 1/n\, X^T X$ when data is centered $E(X) = 0$

Eigen Vector:

- Every square matrix A defines a linear transformation X-> AX
- Direction of vectors representing the data distribution.

PCA Algorithms:

Application of PCA to face recognition

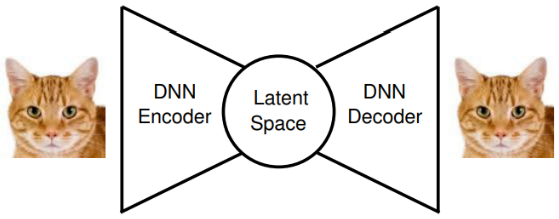
===

Variational Encoders :

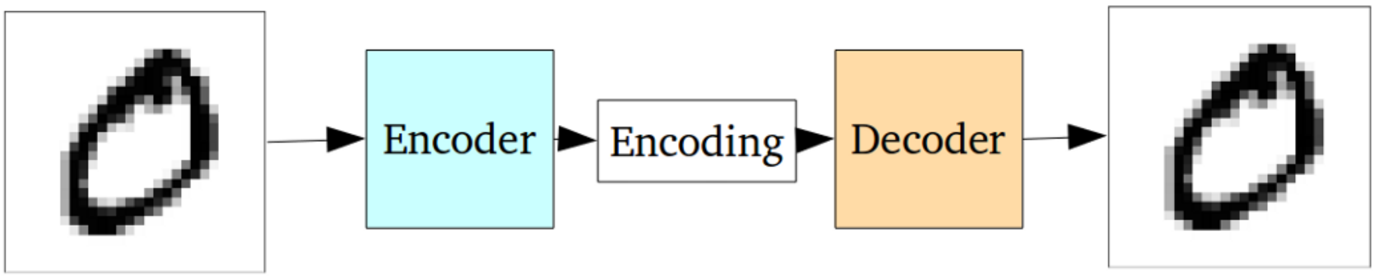
HTTPS://WWW.YOUTUBE.COM/WATCH?V=2M9E-ASXTL8&T=421S&AB_CHANNEL=STATQUESTWITHJOSHSTARMERSTATQUESTWITHJOSHSTARMERVERIFIED

A variational autoencoder consists of an encoder, a decoder, and a loss function. Dimensionality reduction is as data compression where the encoder compress the data (from the initial space to the encoded space, also called latent space) whereas the decoder decompress them. Our intension always remain to keep the maximum of information when encoding and, minimum of reconstruction error when decoding.

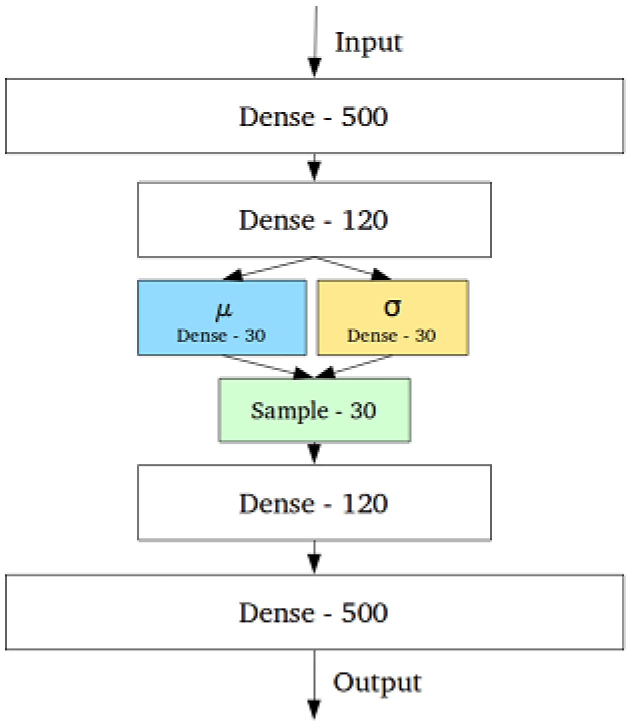
Deterministic Autoencoder:



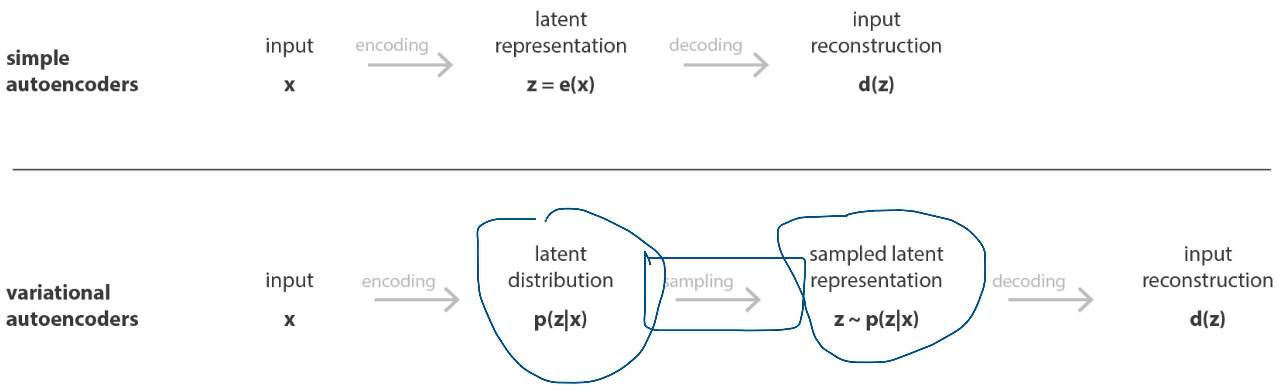
- ❖ **Encoder** is net trained to reconstruct ip data out of learnt internal representation the process that produce the "new features" representation from the "old features" representation Can we use
- ❖ **Is decoder** to generate data by sampling in latent space?
  - ➢ NO! Since we do not know the distribution of latent vars.
  - It can be seen as non linear extension of PCA.
  - The challenge we face with these are while generation, the latent space where they convert their inputs & the space where the encoded vectors lie may not be continuous.



-
- the encoding part that comes after Encoder and before Decoder section, the graph can not deal with inputs that the encoder has never seen before because different classes are clustered bluntly and those unseen inputs are encoded be to something located somewhere in the blank.
- To tackle this problem, the variational autoencoder was created by adding a layer containing a mean and a standard deviation for each hidden variable in the middle layer:



-

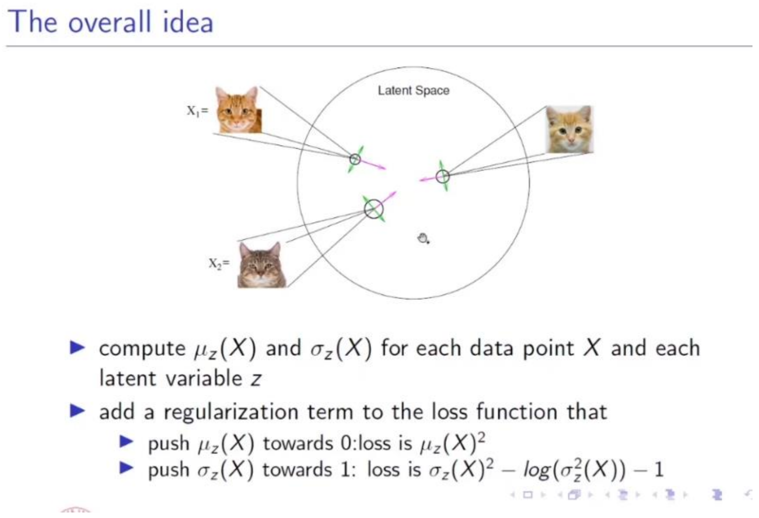


Variational Autoencoders (VAE): There is an extra step of Sampling.

- ❖ It is an encoder whose training is regularised to avoid overfitting.
- ❖ It ensures that the latent space has good properties to enable the generative process.
- ❖ Variational -> close relation between regularization and variational inference method in statistics.
- ❖ LINK
- ❖ We try to force latent variables to have a known distribution

We have the gaussian distribution which is the obvious choice given the mean and the variance.

Here we have the quadratic loss that is one we are pushing these points towards the centre i.e centre the latent space.

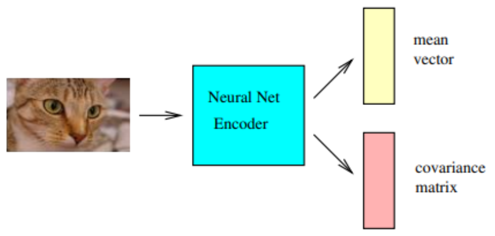


The overall idea

- ▶ compute $\mu_z(X)$ and $\sigma_z(X)$ for each data point X and each latent variable z
- ▶ add a regularization term to the loss function that
  - ▶ push $\mu_z(X)$ towards 0:loss is $\mu_z(X)^2$
  - ▶ push $\sigma_z(X)$ towards 1: loss is $\sigma_z(X)^2 - log(\sigma_z^2(X)) - 1$

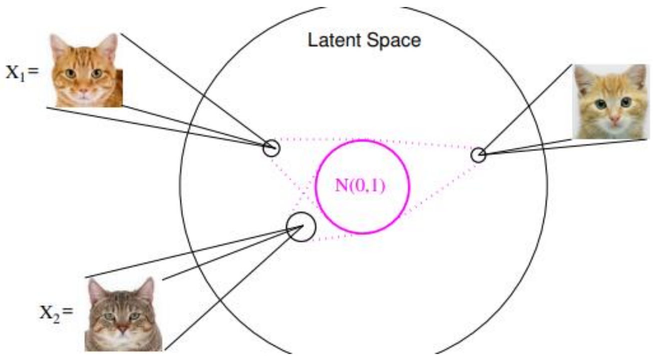
Why center? So the mean = 0, it prevents encoded distribution to be too far apart from each others.

We need to increase the variance by covering all the latent space



MNIST Case:

- ➢ Regularization term -> induce Gaussian distribution of points in latent space.



- ❖ Learning latent distribution
    - ▪ Encoder computes σ(X) together with μ(X)
    - ▪ How to compute "apropriate" variance?
        - Sample around μ(X) using the current variance σ(X).
        - Use the resulting point to reconstruct X
        - Use the reconstruction error to tune both μ(X) and σ(X).
- ❖ Sampling in latent space:
    - ➢ Sampling adds noise to encoding, improves robustness.

KULLBACK LEIBLER REGULARIZATION

It is is a measure of how different a specific probability distribution is from a reference distribution.

In a VAE, there are two components to the loss function:

- the reconstruction term
- the regularization term.

The reconstruction term measures the efficiency of the encoder-decoder with respect to the initial data and output layer. When we get 0 reconstruction loss, an autoencoder perfectly reconstructs the input data. This is a bad sign because of overfitting and a lack of interpretable latent features.

VAEs encode their inputs as Gaussian distribution. This is where we use the K-L divergence. It is better that the distributions of the VAE is regularized. Regualarizationto increases the amount of overlap within the latent space. K-L divergence measures this and is added into the loss function. There is a trade-off between reconstruction and regularization. If we want to reduce our reconstruction error, this comes at the expense of K-L divergence or regularization.

Averaging on all data:

$$\begin{aligned}
&\mathbb{E}_{X\sim P_{data}} KL(Q(z|X)||P(z)) \\
&= -\mathbb{E}_{X\sim P_{data}} \mathcal{H}(Q(z|X)) + \mathbb{E}_{X\sim P_{data}} \mathcal{H}(Q(z|X), P(z)) \\
&= -\mathbb{E}_{X\sim P_{data}} \mathcal{H}(Q(z|X)) + \mathbb{E}_{X\sim P_{data}} \mathbb{E}_{z\sim Q(z|X)}\, log P(z) \\
&= -\mathbb{E}_{X\sim P_{data}} \mathcal{H}(Q(z|X)) + \mathbb{E}_{z\sim Q(z)}\, log P(z) \\
&= \underbrace{-\mathbb{E}_{X\sim P_{data}} \mathcal{H}(Q(z|X))}_{\text{Entropy of } Q(z|X)} + \underbrace{\mathcal{H}(Q(z), P(z))}_{\text{Cross-entropy of } Q(X) \text{ vs } P(z)}
\end{aligned}$$

Why compute mean and variance??

- We want to push Q(z|X) close to a Normal N(0,1)
- When averaging on all X, we get Q(z) ~ N(0,1)

Latent Variable model: expressing probability of a data point X through marginalization over a vector of latent variables.

P(X) = Integration{ P(X|z, θ)P(z)dz} ≈ $\mathrm{Ez}_{\sim P(z)}$ P(X|z, θ)

===

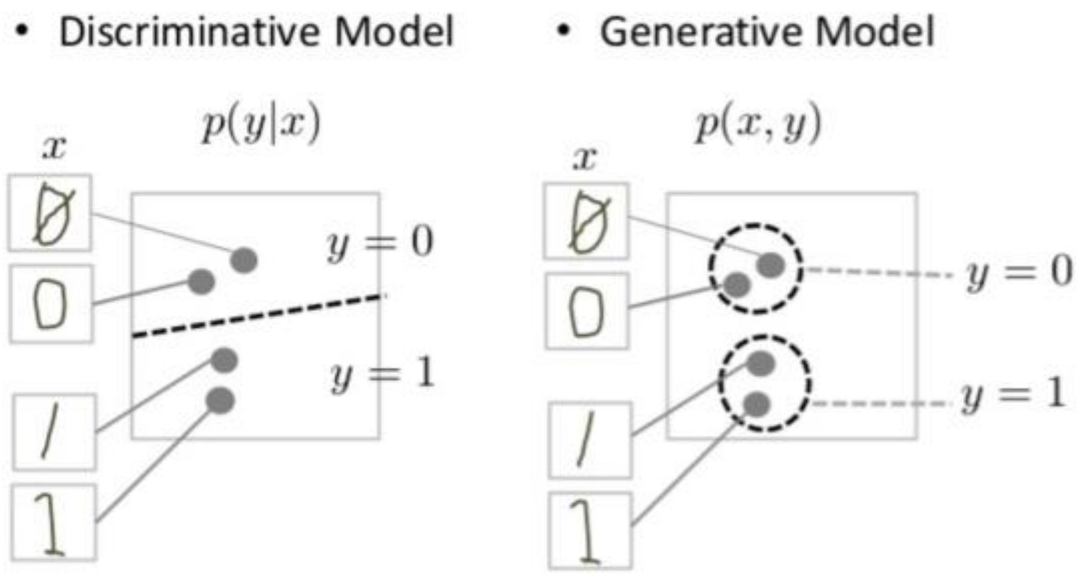
GENERATIVE ADVERSINAL NETWORKS

Generative Models: learns the actual distribution of real data from available samples.

GAN-

Suppose we have a set of data instances X and a labels Y:

- **Generative** models can generate new data instances. Generative models capture the joint probability p(X, Y), or just p(X) if there are no labels.
- **Discriminative** models discriminate between different kinds of data instances. models capture the conditional probability p(labels | data instance).



In pactice, DM has a line that divides space into parts while GM creates clusters in the data point and gives them label

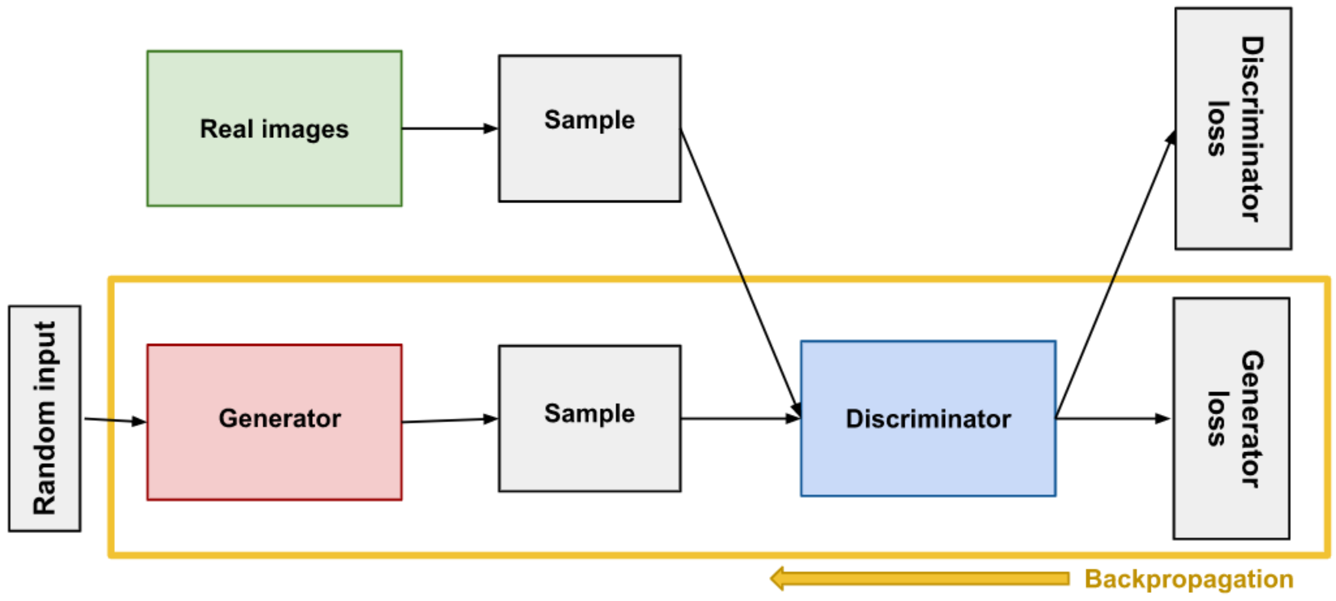
A generative model could generate new photos of animals that look like real animals, while a discriminative model could tell a dog from a cat.

GANs are just one kind of generative model.

A generative model includes the distribution of the data itself, and tells us how likely a given example is. I mean, what are the chances of something will come now. For example, when we use models to predict the next word in a sequence, these are generative models which are simpler than GANs because they can assign a probability to a sequence of words.

A discriminative model ignores the question of whether a given instance is about to happen or not but instead it tells us how likely a label is to apply to the instance.

- The **generator** learns to generate realistic data. The generated instances become negative training examples for the discriminator.



The generator feeds into the discriminator net, and the discriminator produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

This extra chunk of network must be included in backpropagation. Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output

backpropagation starts at the output and flows back through the discriminator into the generator.

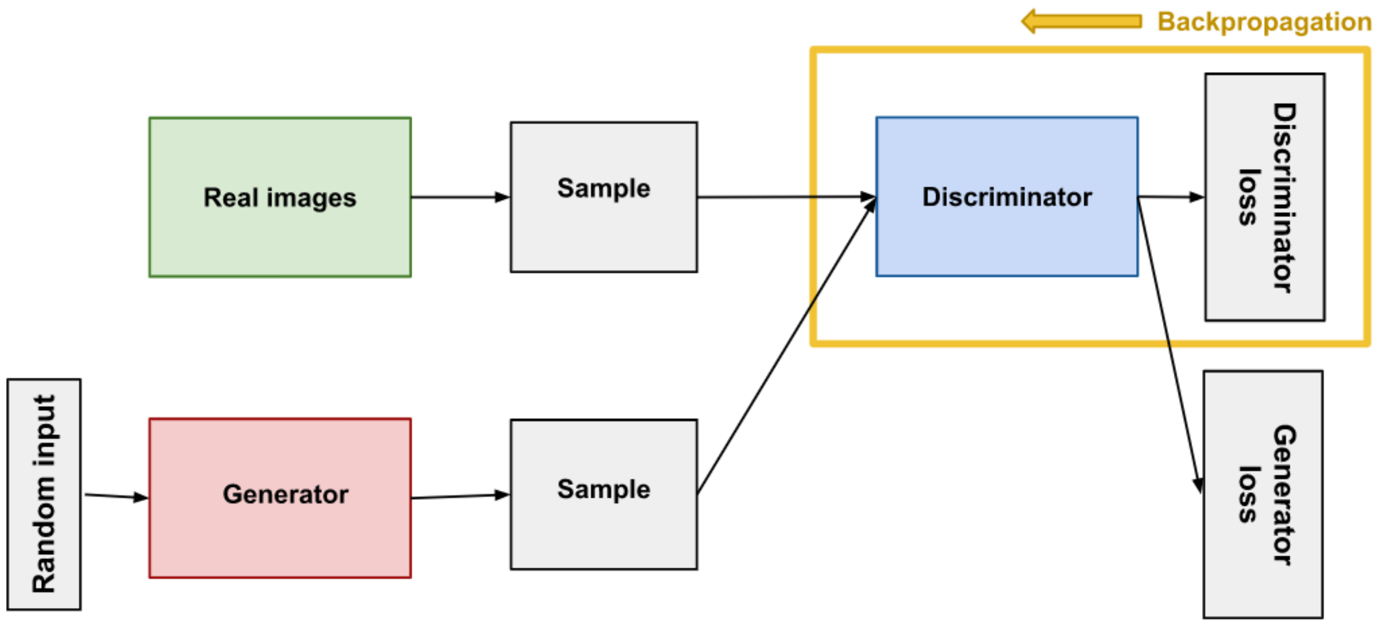
Training Generator steps:

So we train the generator with the following procedure:

1. Sample random noise.
2. Produce generator output from sampled random noise.
3. Get discriminator "Real" or "Fake" classification for generator output.
4. Calculate loss from discriminator classification.
5. Backpropagate through both the discriminator and generator to obtain gradients.
6. Use gradients to change only the generator weights.

- The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:



It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying.

Training data for Discriminator

The discriminator's training data comes from two sources:

- Real data instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.
- Fake data instances created by the generator. The discriminator uses these instances as negative examples during training.

Training GAN: Training Discriminator + Training Generator

GAN contains two separately trained networks, its training algorithm must address two complications:GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs.
2. The generator trains for one or more epochs.
3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.

Click here: https://developers.google.com/machine-learning/gan/gan_structure

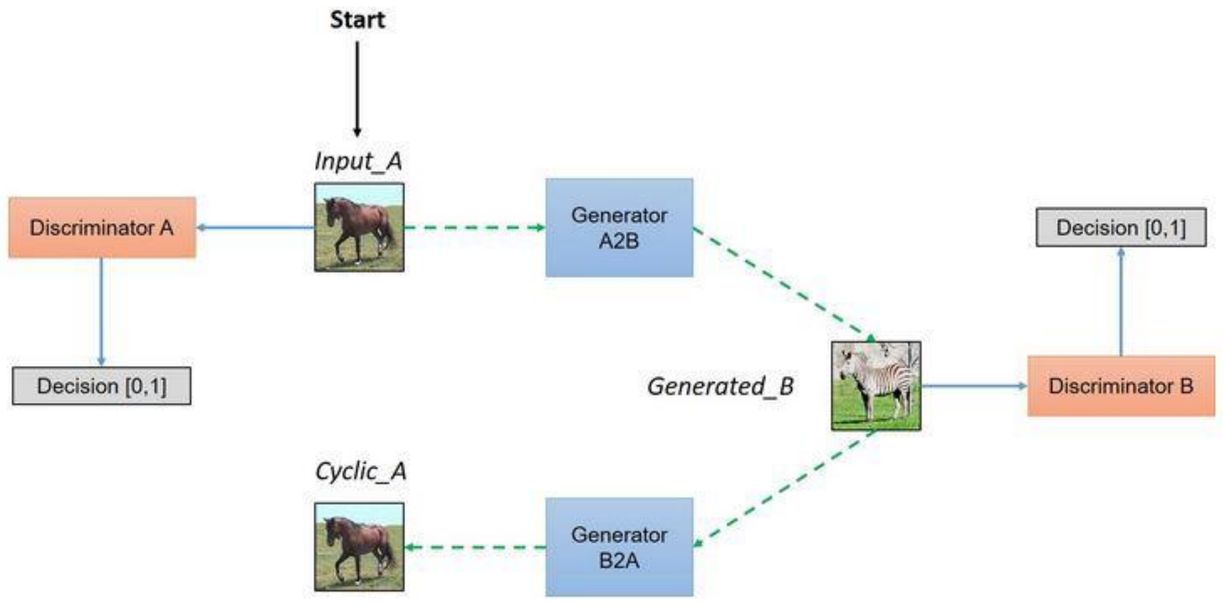
THE CYCLEGAN

It is technique for training unsupervised image translation models via the GAN architecture using unpaired collections of images from two different domains.

This method that can capture the characteristics of one image domain and figure out how these characteristics could be translated into another image domain, all in the absence of any paired training examples. CycleGAN uses a cycle consistency loss to enable training without the need for paired data. I mean to say the models are trained in an unsupervised manner using a collection of images from the source and target domain that do not need to be related in any way.

This technique is powerful because it can transfer the style n domains, interestingly we can see conversion photographs of horses to zebra, and the reverse.

The CycleGAN is an extension of the GAN architecture that involves the simultaneous training of two generator models and two discriminator models.



One generator takes images from the first domain as input and outputs images for the second domain, and the other generator takes images from the second domain as input and generates images for the first domain. Discriminator models are then used to determine how good the generated images are and update the generator models accordingly.

Generative Models-

Goal: build probability distribution p_{model} close to p_{data}.

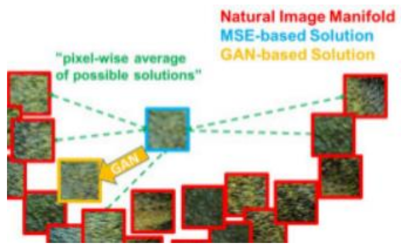
How?

- Explicitly estimate the distribution
- Build generator – to sample according to p_{model}

Why?

- It improves latent representation of data
- Encoding of complex high dimensional distribution
- We work with multi-modal outputs, where we need to choose rather than averaging.
  - We should provide constraints while data generation (color, orientation, etc) to get relevant results.

- Find a way to produce realistic samples probability distribution.
  - **Region of High probability –** pixel wise average of possible solutions that could produce new imgs. Model learns from all the real images, creates an average data from those & uses it to create a new img, this produces bl.



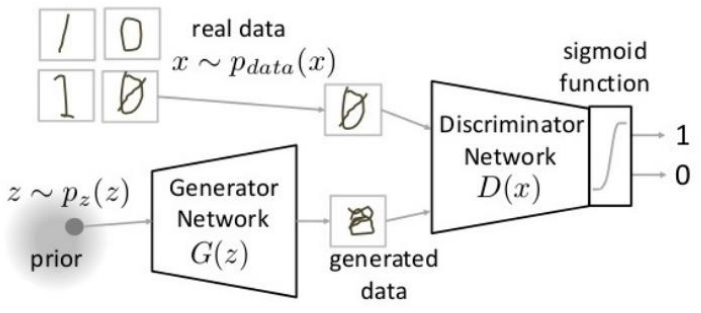
- Can be incorporated into reinforcement learning -> predicting possible future.
- **Another Application:** Super-Resolution. Instead of averaging -> make a choice and create sharp results.

Approach:
-Interrelation b/w generator and discriminator.

-Min Max Game

$\text{Min}_G \text{Max}_D(D,G)$

Between generator and Discriminator



V(D,G) = NCE(D)wrt true data distribution + NCE('false' D) wrt fake generator

$$V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[log\ D(x)] + \mathbb{E}_{z\sim p_z(z)}[log\ (1 - D(G(z)))]$$

*NCE -> Negative cross entropy

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

Problem with GANs:

- **Vanishing Gradients** - if our discriminator is too good, then generator training can fail due to vanishing gradients. So a good D, wont provide enough information for the generator to make progress. Can be solved using – Modified minmax loss
- **Mode collapse** – when the generator prodcuces realistic output, the generator might learn to produce the same output again and again. Because the G is always trying to find the output that seems most realistic to the discriminator. If G produces same output again and again, D tried to reject that output. But the next G of D gets stuck oin local minimum and doesn’t find best strategy to reject that value, then it is too easy for the next G iteration to find the most original o/p for the current D.

  Each iteration of generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the trap. As a result the generators rotate through a small set of output types. This form of GAN failure is called mode collapse.
- **Failure to converge**- we can solve it by adding noise to the discriminator input.
  - We add noise factor as well. Suppose we have MNIST dataset, while generation If you remove the noise vector z, then the only input to your generator is the one-hot label (y). So your 1generator becomes deterministic. It will produce one shape per digit. In other words, your generator is not really a generator anymore, but rather behaves like a simple mapping: 10 digits map to 10 images.
  - Since the generator is limited to producing only 1 out of 10 possible images, then the discriminator job becomes super easy. It manages to differentiate between real and fake images without having to learn higher level semantics, and your training doesn't converge.

- D gets fooled doesn’t mean fake is good. NN can be easily fooled

- Problem with counting, perspective, global structure, .. etc
  - They fail to differentiate how many particular objects should occur at a loc.
  - Have problem to adapt to 3D objects & global structure -> cant understand shape
- Mode collapse: generative specialization on a good, fixes example.

Integrating GANs with VAEs

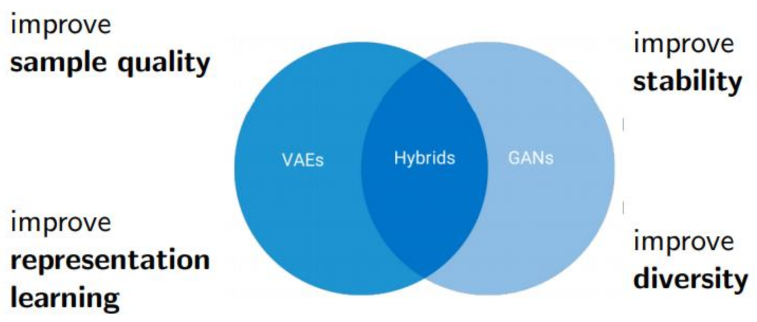
http://efrosgans.eecs.berkeley.edu/CVPR18_slides/VAE_GANS_by_Rosca.pdf

Problems with VAE:

- Similarity metric is crucial.
- Pixel wise metric (squared error) are too sensitive to local small translations/rotations.

Problem with GAN:

- GA -> problems to capture real data distribution
- Unstable & difficult learning


improve **sample quality**
improve **representation learning**
VAEs
Hybrids
GANs
improve **stability**
improve **diversity**

Main Approches:

❖ Acting in latent space :
  -replace KL divergence with Discreminator
  -match aggregated posterior Q(z) wit hexpected prior distribution P(z).
  -called ***Adversial Autoencoders***

❖ Acting in Visible space
  - Replace reconstruction loss with D

- Called ***VAE-GAN***

==

GAN VARIATIONS: PROGRESSIVE GAN & CONDITIONAL GENERATION

Progressive GAN: In a progressive GAN, the generator's first layers produce very low resolution images, and subsequent layers add details. This technique allows the GAN to train more quickly than comparable non-progressive GANs, and produces higher resolution images.

Conditional GAN- It was introduced by Mirza in 2014. They used MNIST for this approach. In Conditinal GAN we train on a labeled data set and let you specify the label for each generated instance. For example, an unconditional MNIST GAN would produce random digits, while a conditional MNIST GAN would let you specify which digit the GAN should generate.

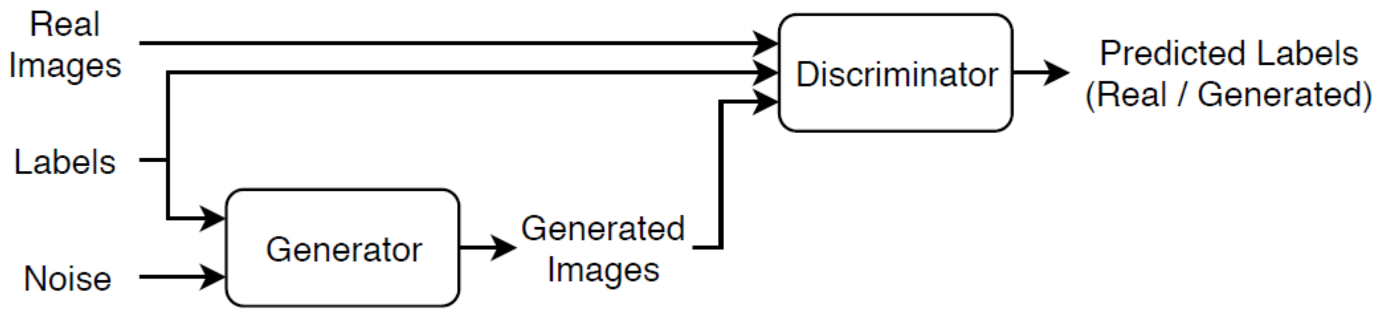
By providing additional information, we get two benefits:

1. Convergence will be faster. Even the random distribution that the fake images follow will have some pattern.
2. You can control the output of the generator at test time by giving the label for the image you want to generate.

Training Conditional GAN:

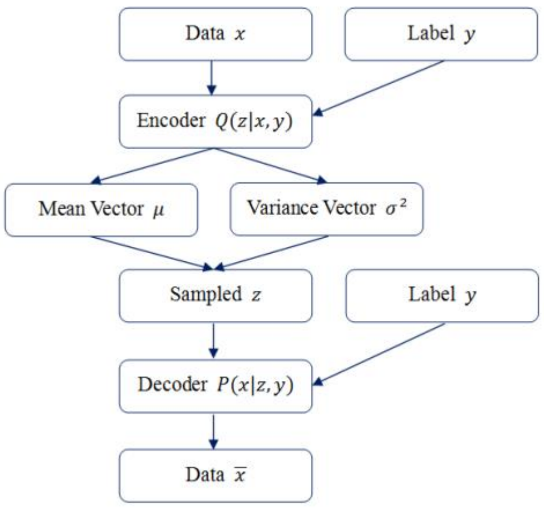
To train a conditional GAN, train both networks simultaneously to maximize the performance of both:

- Train the generator to generate data that "fools" the discriminator.
- Train the discriminator to distinguish between real and generated data.



- *To maximize the performance of the generator* -> maximize the loss of the discriminator
- *To maximize the performance of the discriminator* -> minimize the loss of the discriminator when given batches of both real and generated labeled data.

❖ Conditional VAE (CVAE)



- ➢ Both decoders Q(z|X) P(X|x) are parameterized with given condition: Q(z|X,x) & P(X|z,c)
- ➢ We can still work with - single, condition independent -prior (normal gaussian).
  - Simpler, but more burden on decoder
- ➢ We can use different – possibly learned -prior (different gaussian) for each condtion.
  - Complex, not beneficial

- ➢ Architecture--------------------------------------à

Conditional GANs

❖ G takes i/p condition as well as noise.

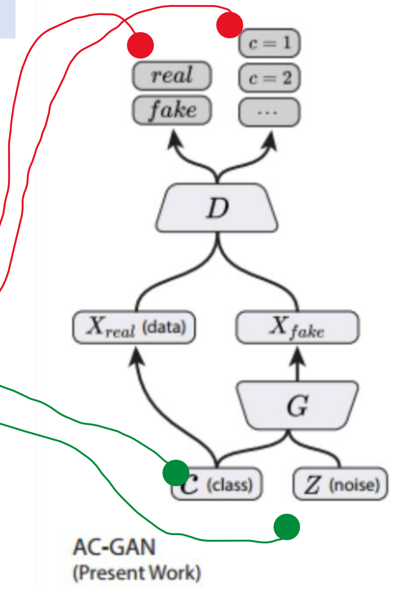
❖ Discriminator:
- ➢ uses conditions to discriminate fakes for real of given class (conditional GAN)
- ➢ classifies wrt different conditions in addition to true/fake discrimination. (Auxiliary Classifier GAN)

AC GAN: AUXILARY CLASSIFIER GAN

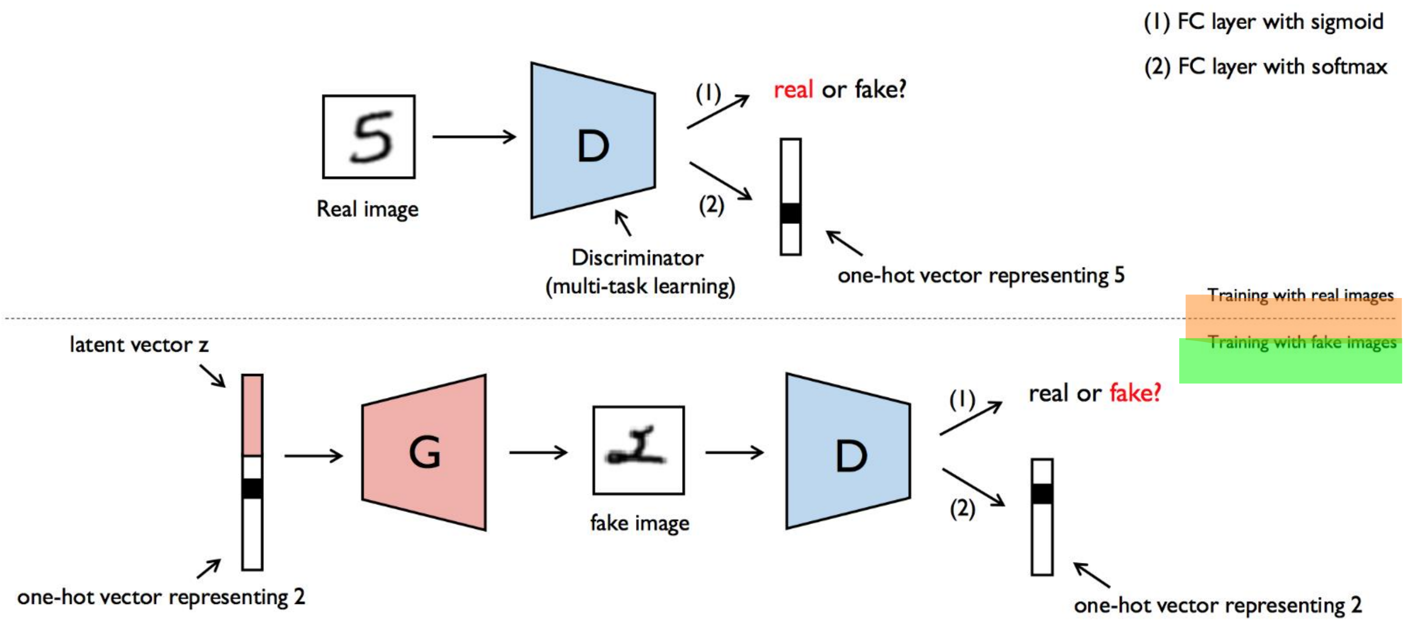
AC-GAN for short, is a further extension of the GAN architecture building upon the CGAN extension. Instead of receiving the image as input, it changes the discriminator to predict the class label of a given image. This helps is stabilizing the training process and allowing the generation of large high-quality images. Here it can learn the representation in a latent space that is independent of class label.

Inputs: class embedding and noise vector

Outputs: binary classifier (fake/real images) and multi-class classifier (image classes)



Training ACGAN:



(1) FC layer with sigmoid
(2) FC layer with softmax
Real image
D
Discriminator
(multi-task learning)
(1)
(2)
real or fake?
one-hot vector representing 5
Training with real images
Training with fake images
latent vector z
one-hot vector representing 2
G
fake image
D
(1)
(2)
real or fake?
one-hot vector representing 2

AC GAN loss function:

- The auxiliary classifier GAN is a type of conditional GAN that requires that the discriminator predict the class label of a given image.

Notation:

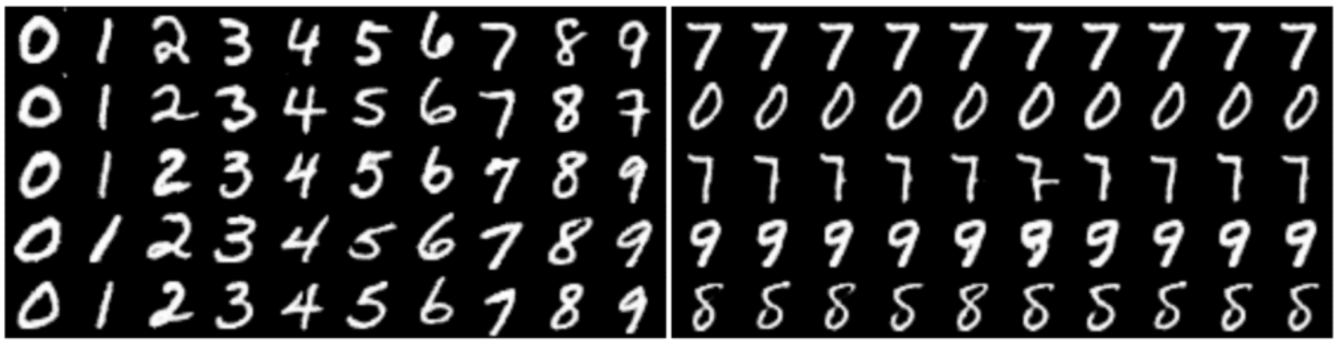
- $p^*(x, c)$ is **true** image-condition joint distribution
- $p_\theta(x, c)$ is the joint distibution of **generated** data
- $q_\theta(c|x)$ is the **classifier**

In addition to the usual GAN objective, we also try to minimize the following quantities:

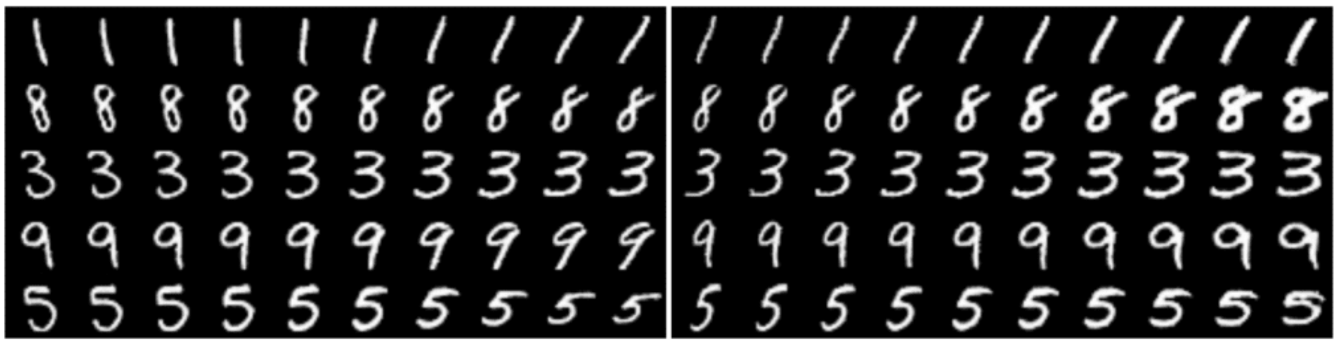
$$-\underbrace{\mathbb{E}_{p^*(x,c)} ln(q_\theta(c|x))}_{term\ 1} - \underbrace{\mathbb{E}_{p_\theta(x,c)} ln(q_\theta(c|x))}_{term\ 2}$$

term 1: the classifier should be consistent with the real distribution
term 2: the generator must create images easy to classify

AC GAN vs InfoGAN: Information Maximizing Generative Adversarial Network



(a) Varying c_1 on InfoGAN (Digit type) (b) Varying c_1 on regular GAN (No clear meaning)



(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation) (d) Varying c_3 from -2 to 2 on InfoGAN (Width)

- AC GAN are similar to InfoGan.
- the goal is to add specific semantic meaning to the variables in the latent space.
- When generating images from the MNIST dataset, the model chooses to allocate random variable to represent the numerical identity of the digit and

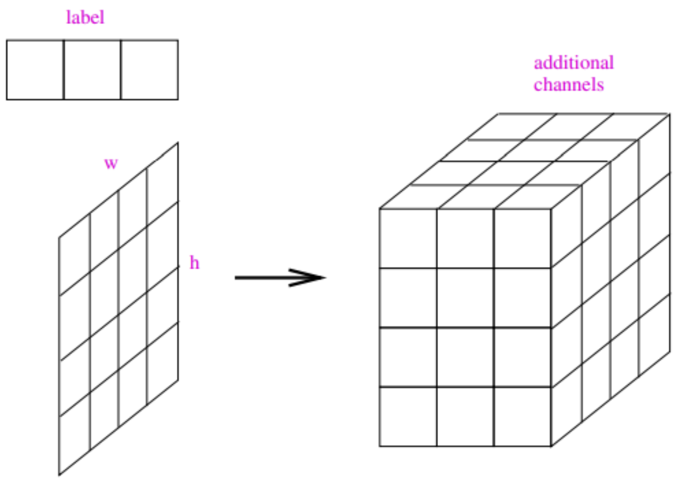
chose to have two additional variables that represent the digit's angle and thickness of the digit's stroke.

- In Infogan, we only have 1st term of the **loss function** that we has in AC GAN.
- Second term-> generate img far from boundaries b/w classes=> sharp img.

$$-\underbrace{\mathbb{E}_{p^*(x,c)} \ln(q_\theta(c|x))}_{\text{term 1}} - \underbrace{\mathbb{E}_{p_\theta(x,c)} \ln(q_\theta(c|x))}_{\text{term 2}}$$

Concrete Handling of condition?

- We pass labels/condn as additional i/p for Conditional n/w.
- Processin:
  - to add in dense layer: concatenate label with i/p
  - To add in convo layer we have 2 ways:
    - Vectorization: repeat label on every ip neuron, & stack them as new



channels.

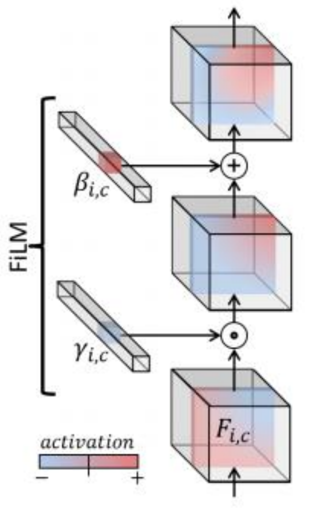
Feature-wise linear Modulation (FILM)

Use condition to give diff weight to each feature.

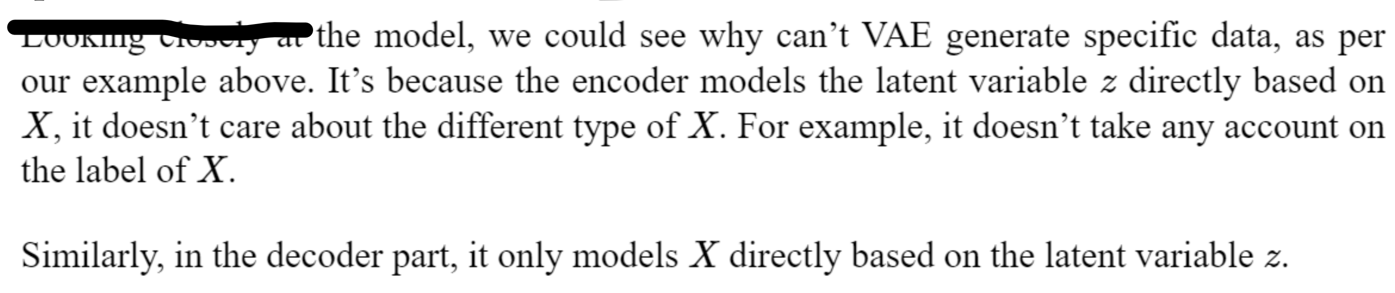
Use cond to generate 2 vectors γ(gamma) and β(beta) with size equal to the channels of the layer.

Rescale the layes by gamma & add beta

It is less invasive that parametrising the weights.

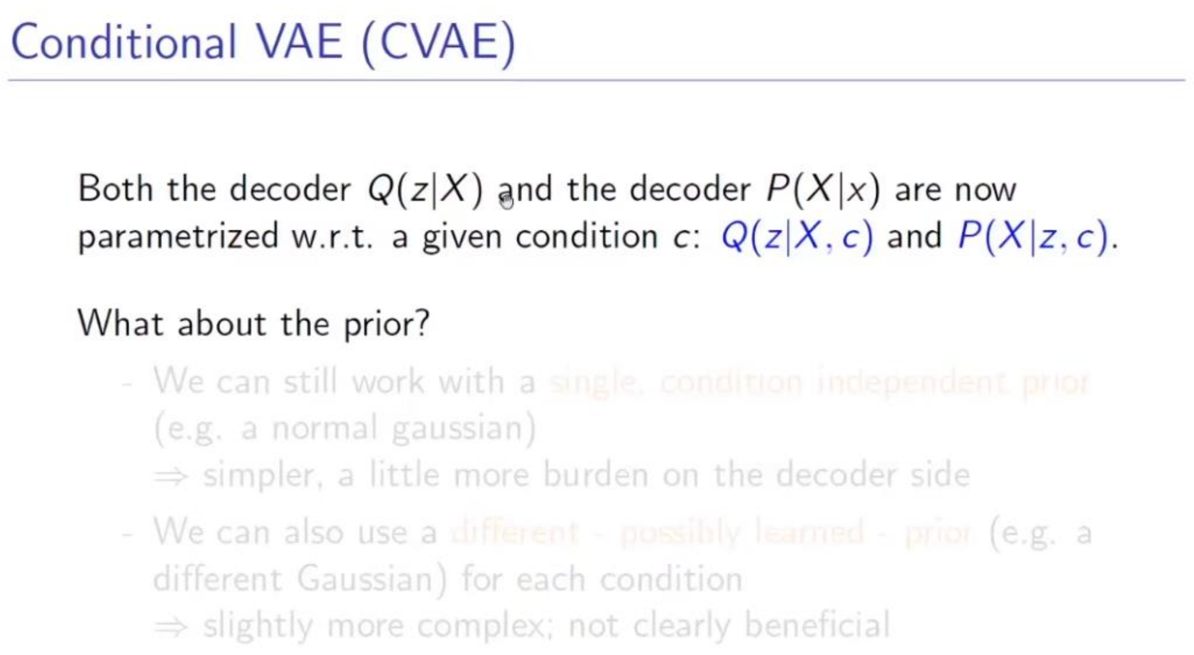


Variational AutoEncoders: are deep learning technique for learning latent [redacted]


[redacted] the model, we could see why can't VAE generate specific data, as per our example above. It's because the encoder models the latent variable z directly based on X, it doesn't care about the different type of X. For example, it doesn't take any account on the label of X.

Similarly, in the decoder part, it only models X directly based on the latent variable z.

- This is the reason we moved for VAE to Conditinal VAE, we can improve VAE by conditioning the encoder and decoder to another thing(s).
-


Conditional VAE (CVAE)

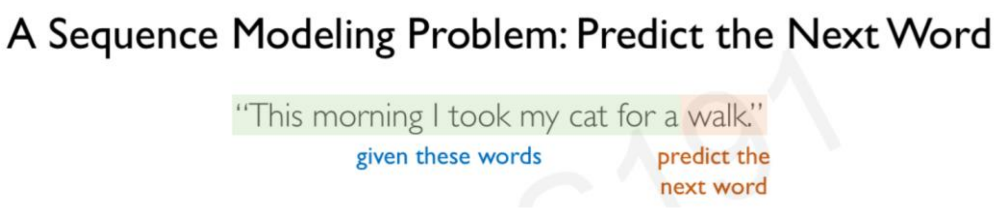
Both the decoder $Q(z|X)$ and the decoder $P(X|x)$ are now parametrized w.r.t. a given condition c: $Q(z|X,c)$ and $P(X|z,c)$.

What about the prior?

- We can still work with a single, condition independent prior (e.g. a normal gaussian)
  $\Rightarrow$ simpler, a little more burden on the decoder side
- We can also use a different - possibly learned - prior (e.g. a different Gaussian) for each condition
  $\Rightarrow$ slightly more complex; not clearly beneficial

Given an encoder and the decoder, an encoder is given with the latent variable given the input and the decoder with the latent space given the input.

SEQUENCE MODELLING



A Sequence Modeling Problem: Predict the Next Word

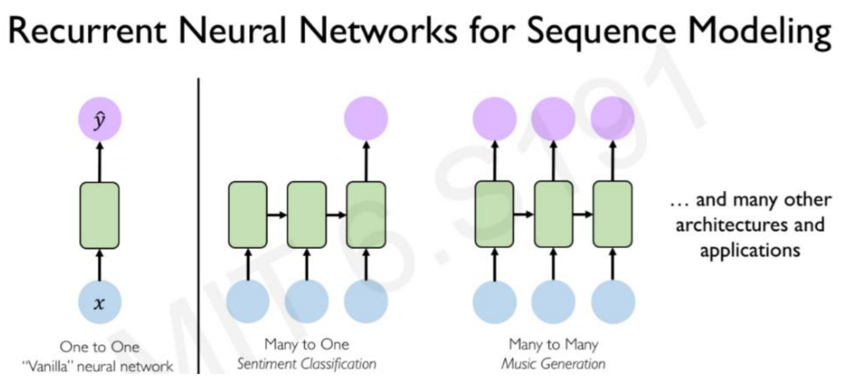
"This morning I took my cat for a walk."

given these words predict the next word

RNN: RECURRENT NEURAL NETWORKS

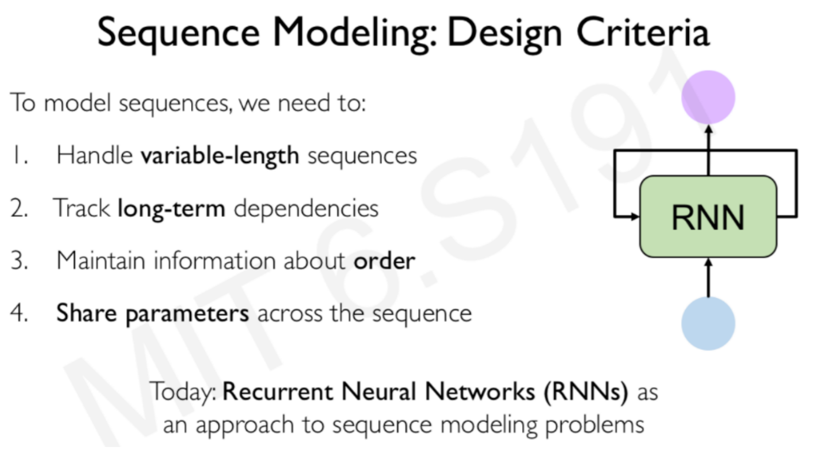
[MIT 6.S191 (2020): Recurrent Neural Networks]

===



Recurrent Neural Networks for Sequence Modeling

===============

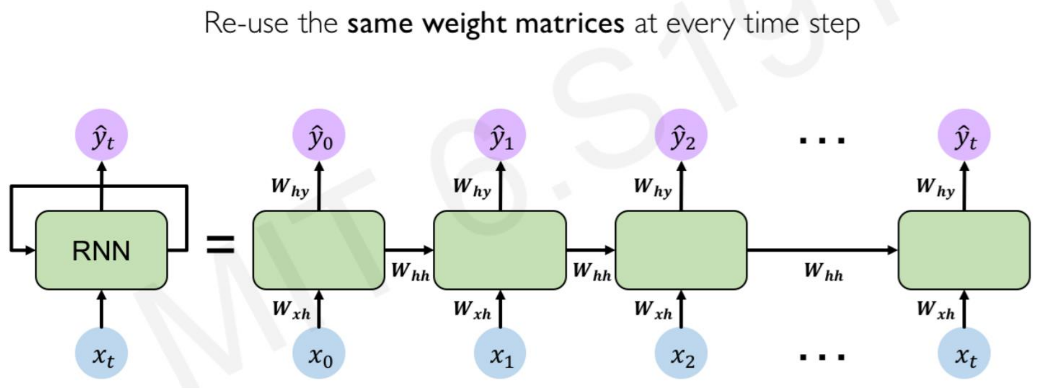


Sequence Modeling: Design Criteria

To model sequences, we need to:

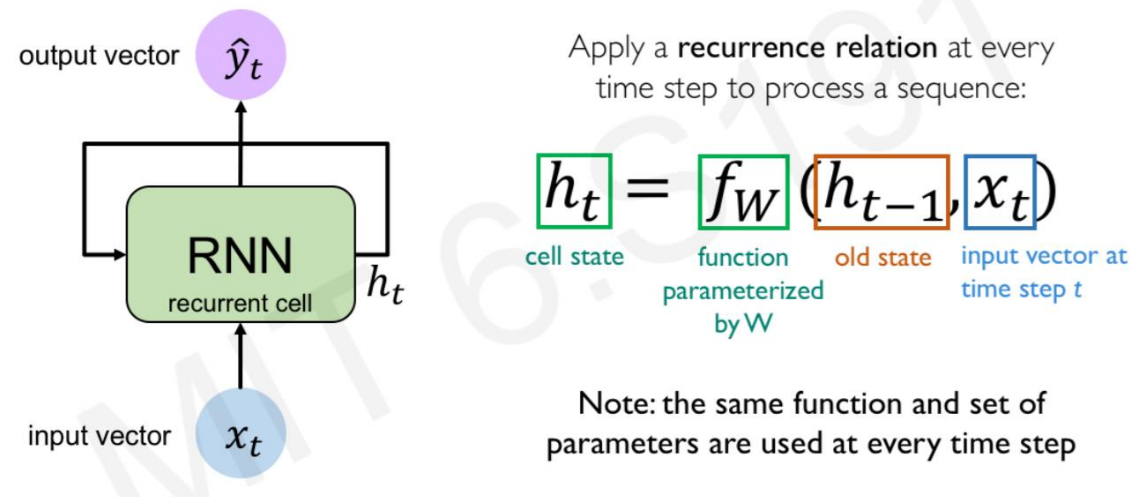
1. Handle **variable-length** sequences
2. Track **long-term** dependencies
3. Maintain information about **order**
4. **Share parameters** across the sequence

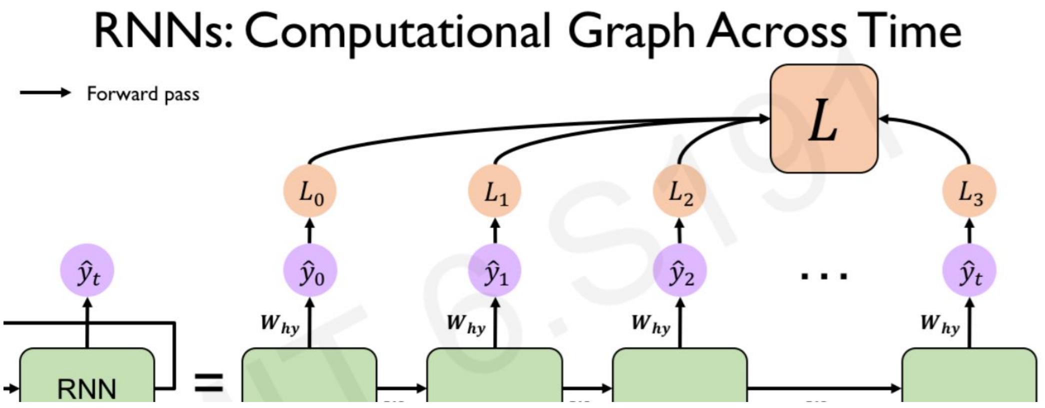
Today: **Recurrent Neural Networks (RNNs)** as an approach to sequence modeling problems

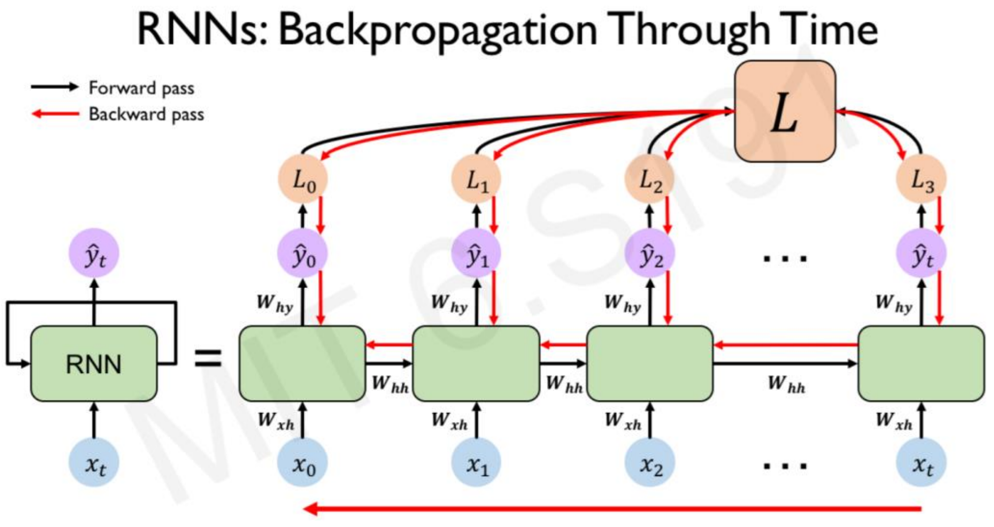


Re-use the **same weight matrices** at every time step

RNN : Recurrent NN


output vector
$\hat{y}_t$
RNN
recurrent cell
h_t
input vector
x_t
Apply a recurrence relation at every time step to process a sequence:
$h_t = f_W(h_{t-1}, x_t)$
cell state
function parameterized by W
old state
input vector at time step t
Note: the same function and set of parameters are used at every time step
MIT 6.S191





- All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, foe example - **single tanh layer**.
- Modelling Sequences of data
- Problems –

- ➢ Turn ip sequence into output seq. (translation of languages, speech sound recg)
- ➢ Predict the next term in seq.
- ➢ Predict result from a temporal sequence of states [reinforcement Learning, robotics]

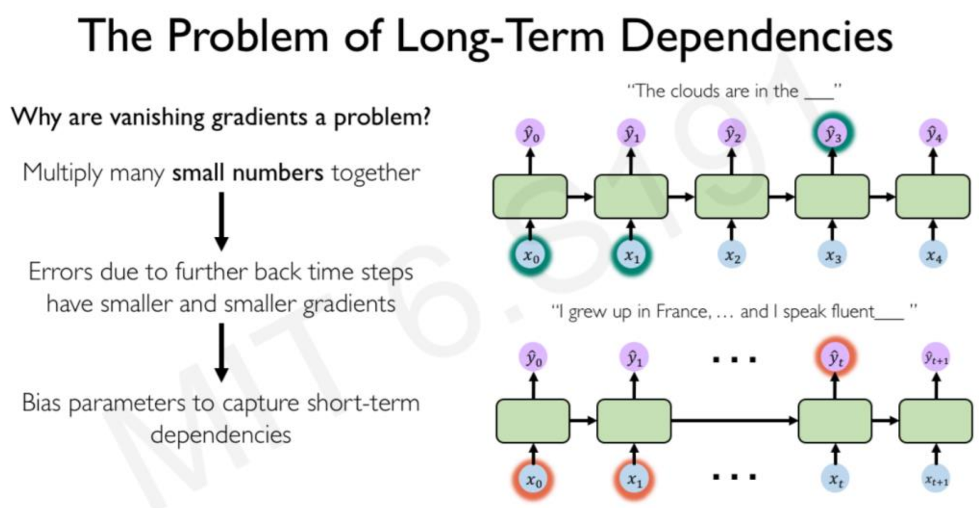
- ❖ Memoryless approach but it is very difficult to deal with very long-term dependencies.
  - ➢ Here, we Compute op as result of a fixed number of elements in ip seq.
- ❖ So, we talk about RNN:
  - ➢ Here we have backward connections, so that hidden states depend in the previous values of net.
  - ➢ Hidden states update at each step & it is a complex process.
  - ➢ The process is know as **Temporal Unfolding** where
    - ▪ Weights update happens after certain steps.
    - ▪ RNN is layered net -> uses same weights matrix again and again.
  - ➢ Sharing weight through time.
    - ▪ Compute gradient as usual & average the gradients st. the have same update.
    - ▪ If initial weights satisfy constraints, they will keep satisfying.
- ❖ BP through time: BPTT
  - ➢ RNN -> layered – feed fwd net with shared weights – train feed forward net with weight constraints.
  - ➢ Reasoning in the time domain:
    - ▪ **Fwd pass** keeps stack of activities of all units at each time step.
    - ▪ **Bwd pass** removes activities off the stack – compute error derivatives at each time step.
    - ▪ Finally add together derivatives at all different times for each weight.

- ❖ **Hidden state initialization**
  - ➢ Specify initial activity state of all hidden & op units.
  - ➢ Treated as parameters- learned in same way as weights.
    - ▪ Start with random guess for initial state
    - ▪ @ end of each training sequence, BP through time all the waay to initial states- get gradient of error functions wrt each initial state
    - ▪ Adjust initial states by following -ve gradient.

Training RNN is difficult:

- Backward pass is linear:
  - Fwd pass
    - we use squashing f() to prevent activity vecotrs from exploding.
    - What are squashing f?
      - There are mainly four **activation functions** (step, sigmoid, tanh and relu) used in neural networks in **deep learning**. These are also called **squashing functions** as these **functions squash** the output under certain range. We will also see various advantages and disadvantages of different **activation functions**.
    - Determine slope of linear f() used for BP

  - Bwd pass- linear, if we double the error derivatives at the final layer – all error derivatives will double. Very complex process.



The Problem of Long-Term Dependencies

Why are vanishing gradients a problem?

Multiply many **small numbers** together

Errors due to further back time steps have smaller and smaller gradients

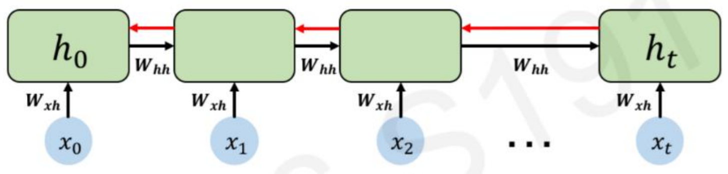
Bias parameters to capture short-term dependencies

"The clouds are in the ___"
$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$
x_0 x_1 x_2 x_3 x_4
"I grew up in France, … and I speak fluent___"
$\hat{y}_0$ $\hat{y}_1$ … $\hat{y}_t$ $\hat{y}_{t+1}$
x_0 x_1 … x_t x_{t+1}

Exploding gradients are a problem where large error gradients add up and result in very large updates to neural network model weights during training. This causes the model to become unstable and unable to learn from your training data.C

Common solution to exploding gradients is to change the error derivative before propagating it backward through the network and using it to update the weights. By rescaling the error derivative, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow.

Standard RNN Gradient Flow: Exploding Gradients

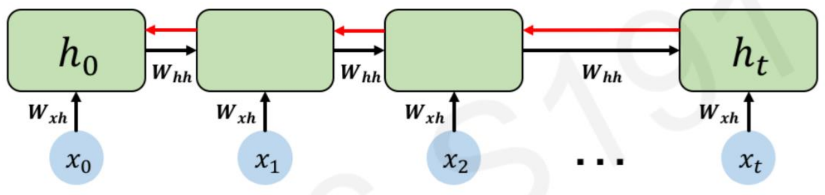


Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation!**

Many values > 1:
exploding gradients

Gradient clipping to scale big gradients

Standard RNN Gradient Flow: Vanishing Gradients



Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation!**

Many values > 1:
exploding gradients

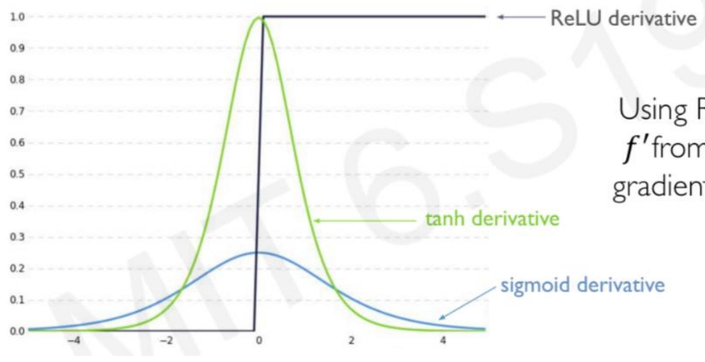
Gradient clipping to scale big gradients

Many values < 1:
vanishing gradients

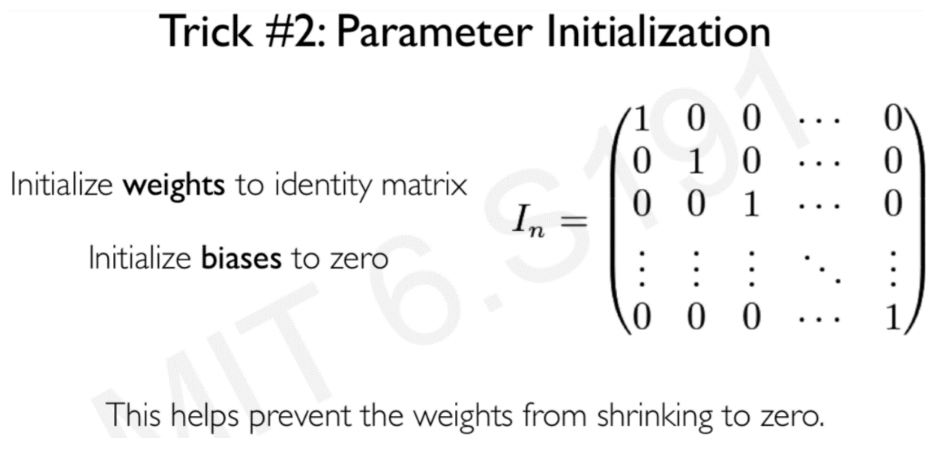
1. Activation function
2. Weight initialization
3. Network architecture

How to deal with Vanishing Gradients:

Trick #1: Activation Functions



Using ReLU prevents f' from shrinking the gradients when $x > 0$



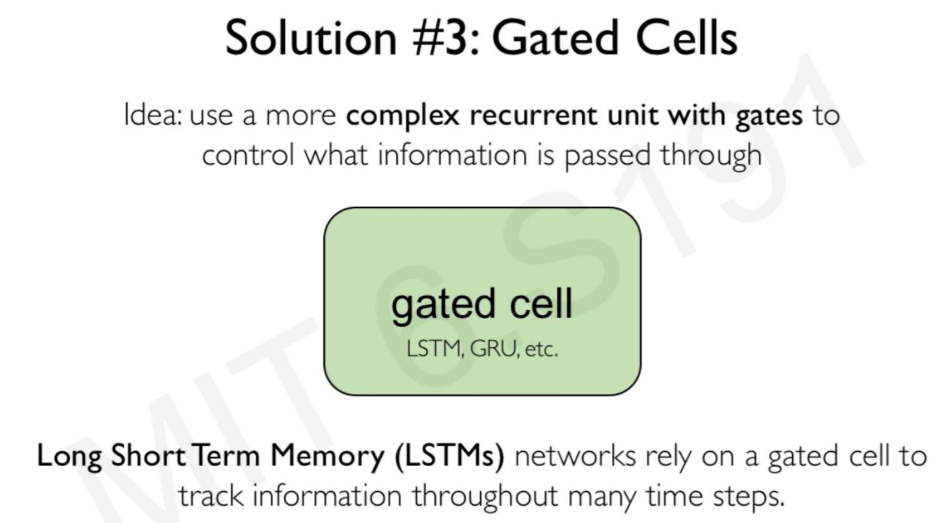
Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

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

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.

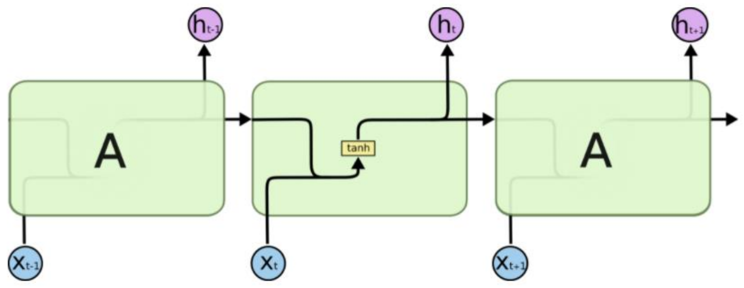
RNN applications:

Music Generation

Sentiment Analysiss

Machine Translation of Languages

LSTM – LONG SHORT TERM MEMORY

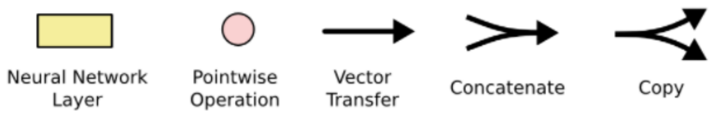


RNNs might be able to connect previous information to the present task. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the ___,". RNN can solve this. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information. But sometimes the gap between the information becomes too large. And it is difficult to keep tranck of information for RNN. So we move toLSTM. LSTMs are explicitly designed to avoid the long-term dependency problem.

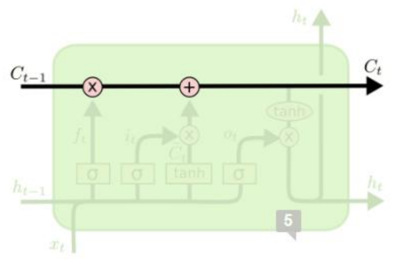
LSTM have this chain like structure, but the repeating module has a different structure than RNN. Instead of having a single neural network layer like RNN, there are four, interacting in a very special way.

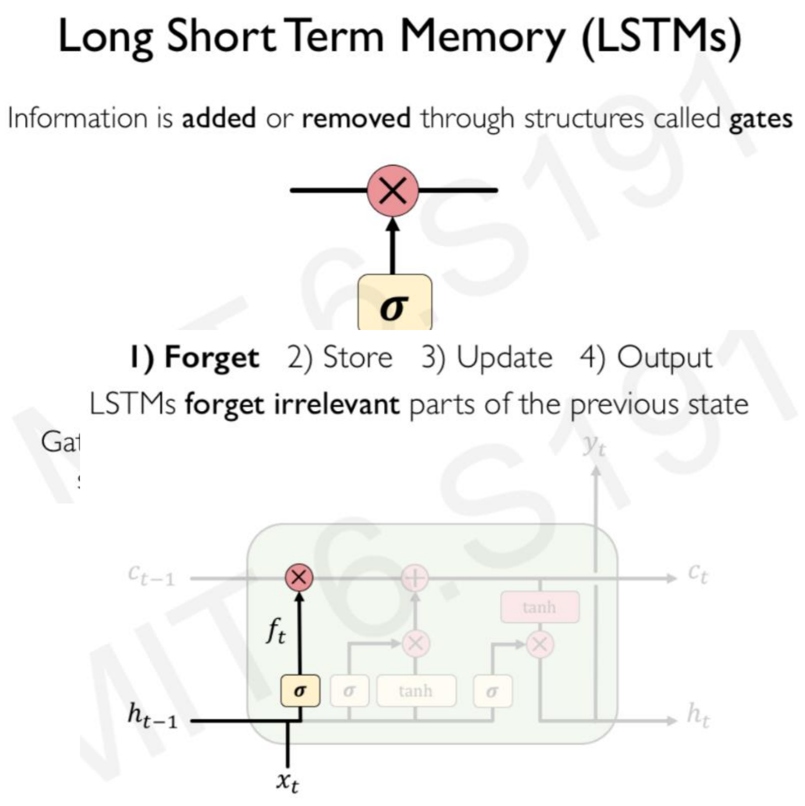
Understanding LSTM Networks -- colah's blog

- ❖ LSTMs don't have this problem!
  - ➢ capable of learning long-term dependencies.



1. like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. Step by step:



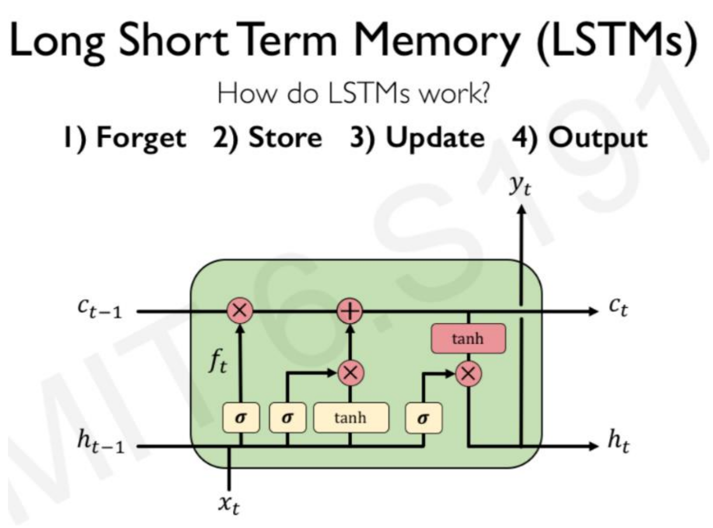


Long Short Term Memory (LSTMs)

Information is **added** or **removed** through structures called **gates**

1) Forget 2) Store 3) Update 4) Output

LSTMs **forget irrelevant** parts of the previous state



Long Short Term Memory (LSTMs)

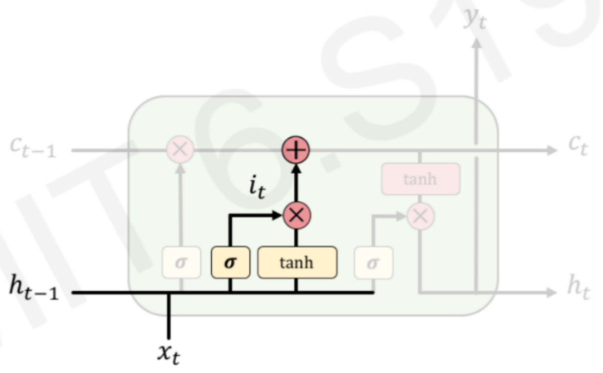
How do LSTMs work?

1) Forget 2) Store 3) Update 4) Output

[Understanding LSTM Networks -- colah's blog]

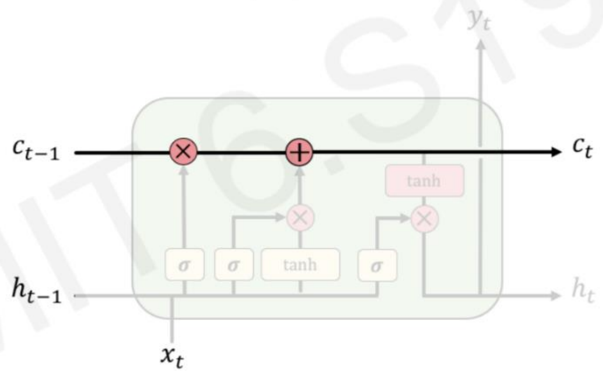
1) Forget **2) Store** 3) Update 4) Output

LSTMs **store relevant** new information into the cell state



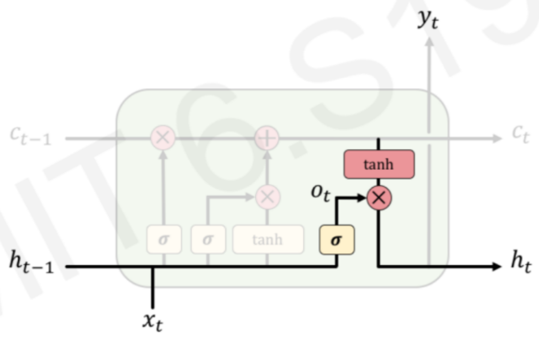
1) Forget 2) Store **3) Update** 4) Output

LSTMs **selectively update** cell state values

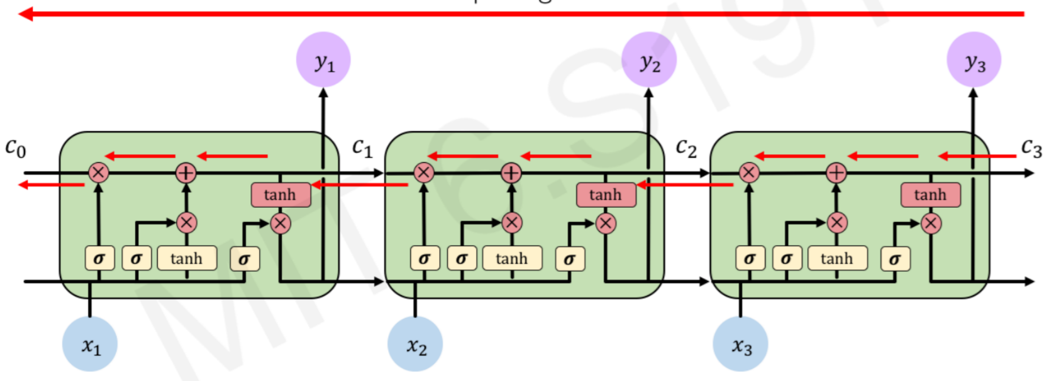


1) Forget 2) Store 3) Update **4) Output**

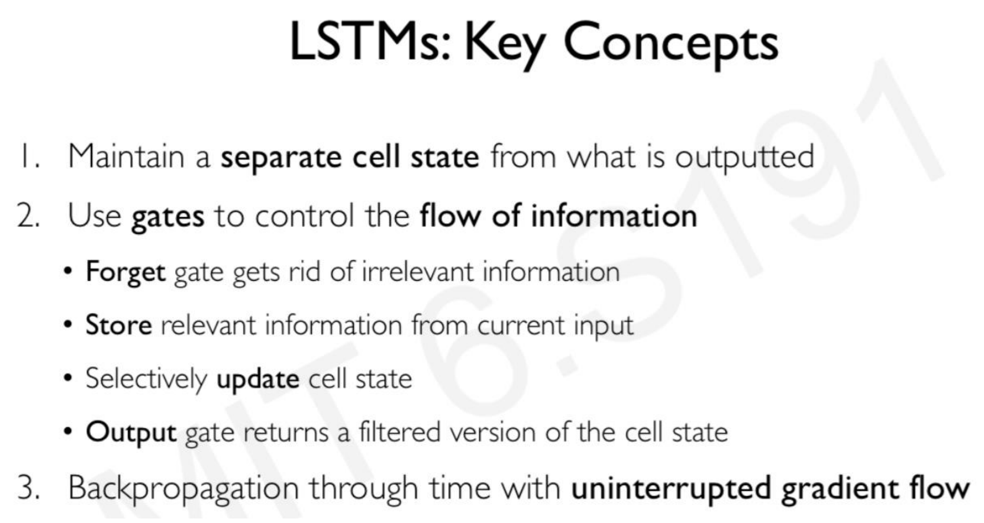
The **output gate** controls what information is sent to the next time step



Uninterrupted gradient flow!

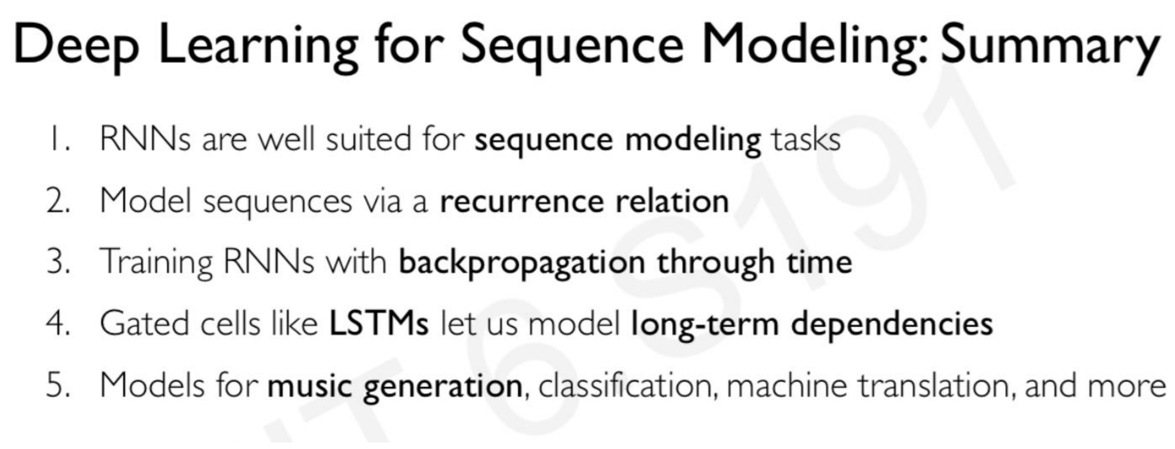


These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.



LSTMs: Key Concepts

1. Maintain a **separate cell state** from what is outputted
2. Use **gates** to control the **flow of information**
   - **Forget** gate gets rid of irrelevant information
   - **Store** relevant information from current input
   - Selectively **update** cell state
   - **Output** gate returns a filtered version of the cell state
3. Backpropagation through time with **uninterrupted gradient flow**



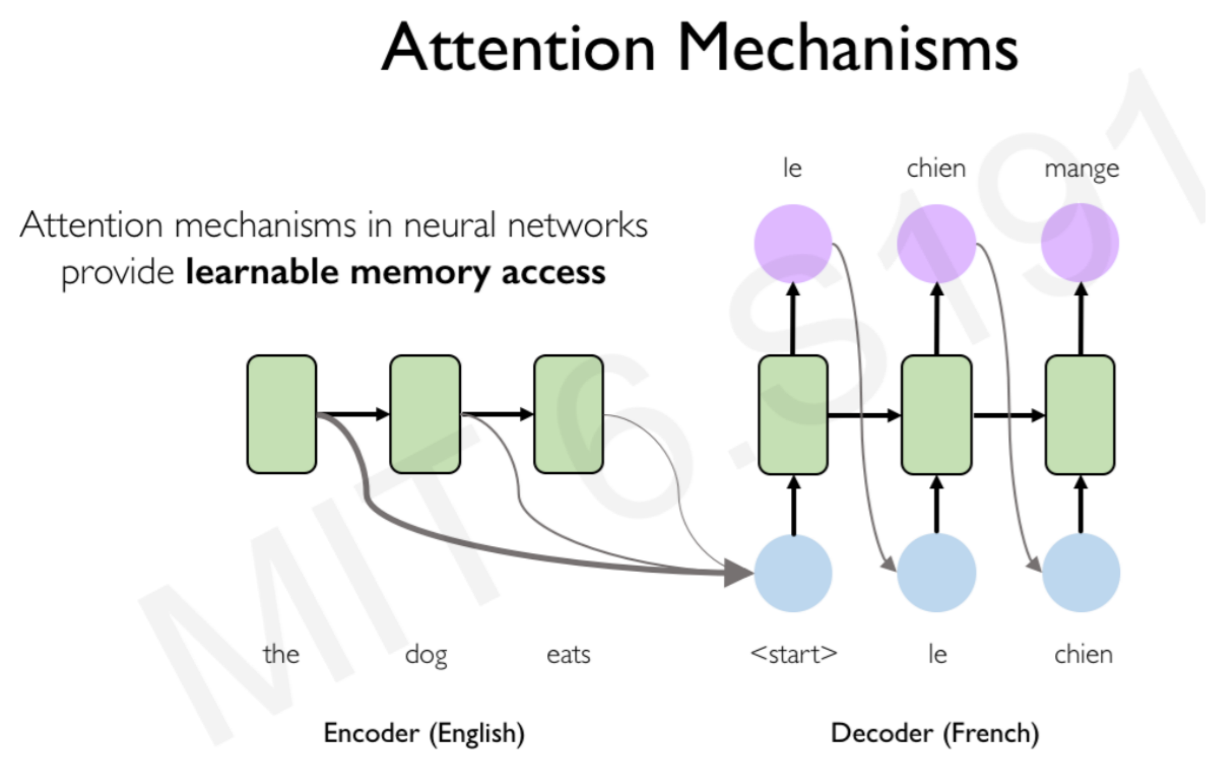
Deep Learning for Sequence Modeling: Summary

1. RNNs are well suited for **sequence modeling** tasks
2. Model sequences via a **recurrence relation**
3. Training RNNs with **backpropagation through time**
4. Gated cells like **LSTMs** let us model **long-term dependencies**
5. Models for **music generation**, classification, machine translation, and more

==

ATTENTION & TRANSFORMERS

Attention:



Attention Mechanisms

Attention mechanisms in neural networks provide **learnable memory access**

- ability to focus on different parts of the input, according to the requirements of the problem being solved.
- In NN- attention mechanism – differentiable so we learn to focus by standard BP
- Current approach – focus everywhere.
- As Gating maps:
  - Gating maps dynamically generated by NN – allowing focus on diff part of i/p
  - Forget map, input map and output map in LSTMs are examples of attention mechanisms.

- **squeeze and excitation model**
  - https://arxiv.org/pdf/1709.01507.pdf
  - easy-to-plug-in module called a Squeeze-and-Excite block (abbreviated as SE-block) which consists of three components (shown in the figure above):
    - Squeeze Module
    - Excitation Module
    - Scale Module
  - It is designed to improve the representational power of a network by enabling it to perform dynamic channel-wise feature recalibration.

- The block has a convolutional block as an input.

- Each channel is "squeezed" into a single numeric value using average pooling.
- A dense layer followed by a ReLU adds non-linearity and output channel complexity is reduced by a ratio.
- Another dense layer followed by a sigmoid gives each channel a smooth gating function.
- Finally, we weight each feature map of the convolutional block based on the side network; the "excitation".

- **modular multi purpose layer:**
  - attention layer is based on the key-value paradigm, implementing a sort of associative memory.
  - attention layer is based on the key-value paradigm, implementing a sort of associative memory.
  - values are also used as keys (self-attention).

For each key k_i compute the **scores** a_i as

$$a_i = \alpha(q, k_i)$$

obtain **attention weights** via softmax:

$$\vec{b} = softmax(\vec{a})$$

retrun a weighted sum of the values:

$$o = \sum_{i=1}^{n} b_i v_i$$

- **Typical score functions**
  - Different score functions lead to different attention layers
    - Dot product: query and key must have same dimension.

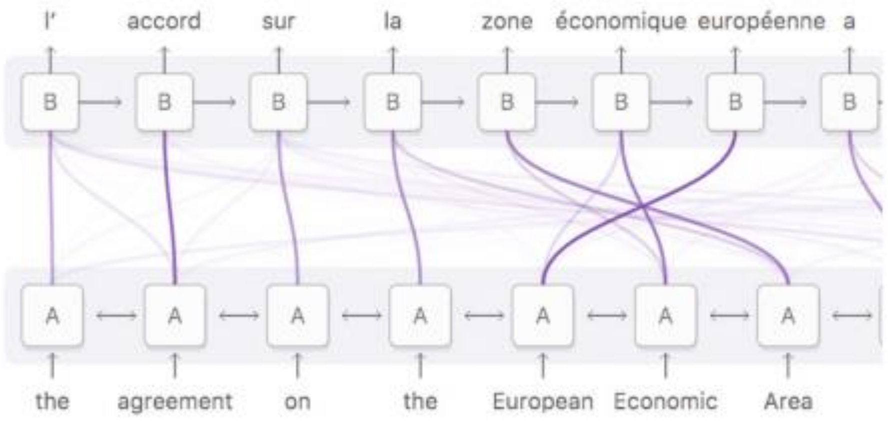
$$\alpha(q, k) = q \cdot k / \sqrt(d)$$

    - MLP: α is computed by a neural network.

$$\alpha(k, q) = tanh(W_k \vec{k} + W_q \vec{q})$$

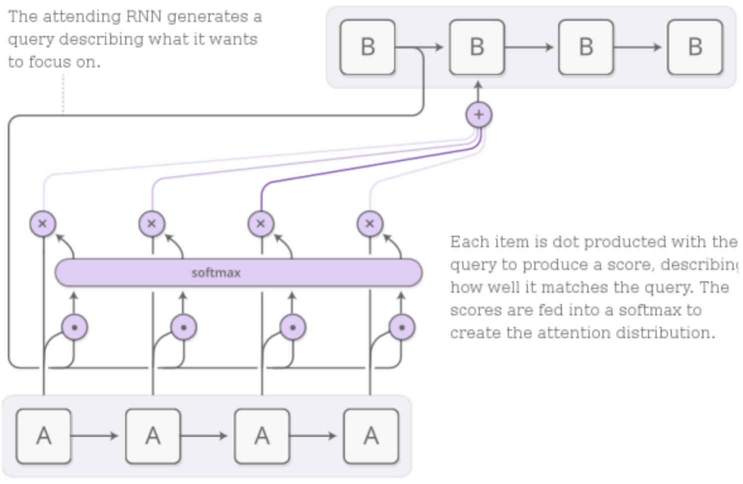
Application to translation:

- https://arxiv.org/pdf/1409.0473.pdf
- Translation has two parts: alignment and translation.
  - **Alignment**: identifying which part of ip seq are relevant to each word in op.
    - Alignment is form of attention:



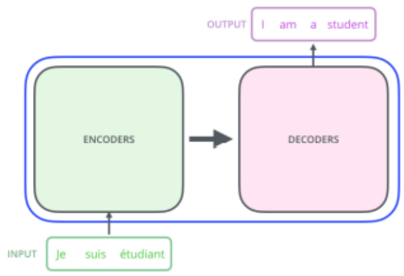
- **Translation**: using relevant information to select appropriate o/p.

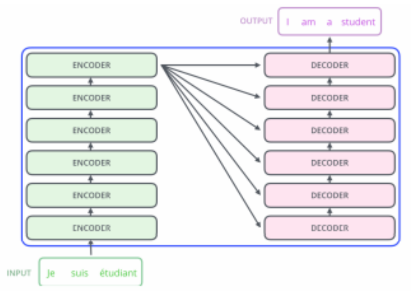
➢ **Producing attention maps:**


The attending RNN generates a query describing what it wants to focus on.
B
softmax
Each item is dot producted with the query to produce a score, describing how well it matches the query. The scores are fed into a softmax to create the attention distribution.
A

- Decoder decides which parts of the source sentence to pay attention.
- It is in control of attention mechanism, Encoder has less burden to encode all info in source sentence as fixed length vtr.
- By this approach- info spreads throughout the seq of annotations.

TRANSFORMERS





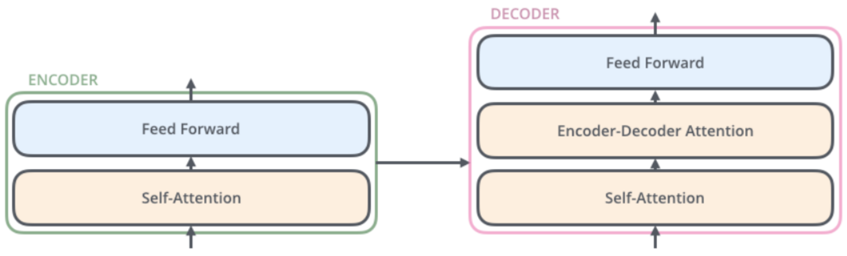
➢ The transformer is a new encoder-decoder architecture that uses only the **attention mechanism** instead of RNN to encode each position.

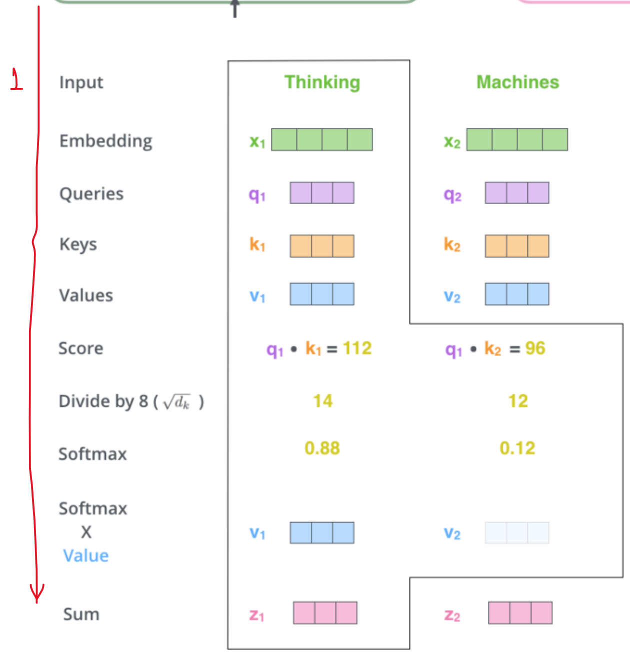
➢ encoding component is a stack of encoders. Similarly, the decoding component is a stack of decoders.

- ▪ encoder is organized as a self-attention layer (query, key and value are shared), followed by feedforward component (a couple of dense layers). They do not share weights.

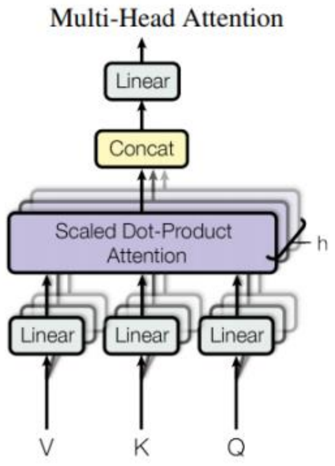
- The encoder's inputs first flow through a self-attention layer. The encoder look at other words in the input sentence when it is performing encoding. The outputs of the self-attention layer are fed to a feed-forward neural network.
- The decoder has both those layers, but between them is an additional layer known as attention layer that helps the decoder focus on relevant parts of the input sentence.
- decoder is similar, with an additional attention layer -helps to focus on relevant parts of the ip sentence.

➢ Applications like Bert and GPT, (with all relative families) are based on Transformers.

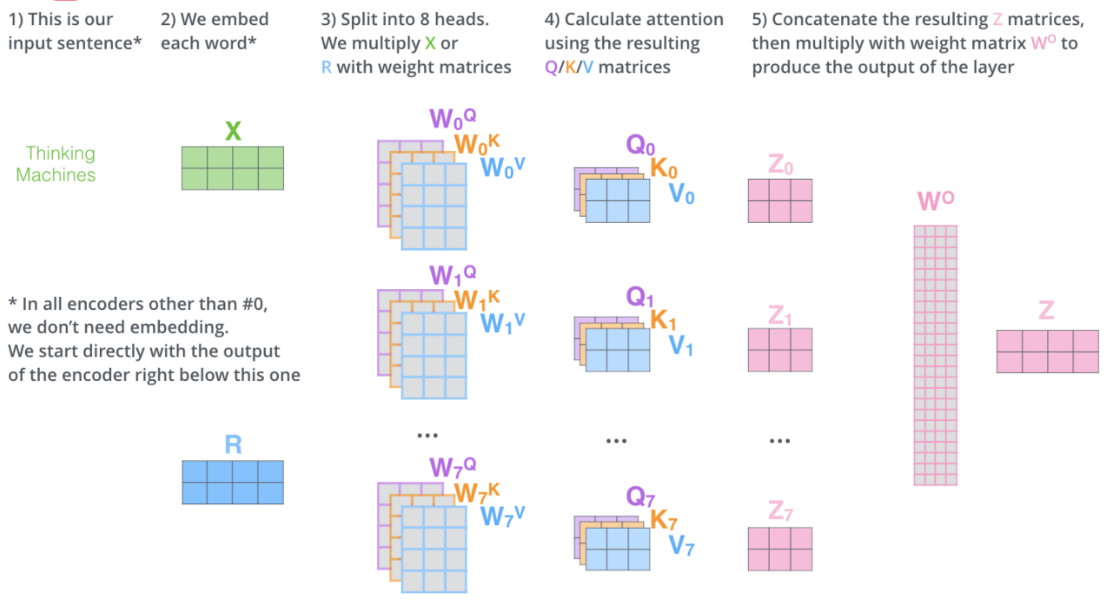




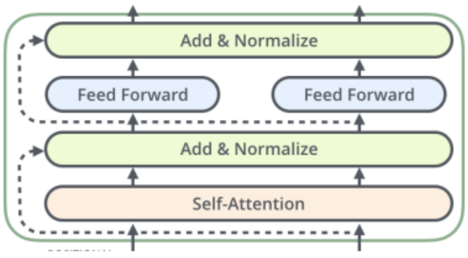
Multi head attention:



- ➢ expands the model's ability to focus on different positions, for different purposes.
- ➢ multiple "representation subspaces" – focus on different attributed of ip.
- ➢ we can apply a Boolean mask to the input, to hide part of its content.
- ➢ **Training of multihead attention:**


1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
R
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

❖ **Residual connections:**



- ➢ Each sub layer (self-attention, ffnn) in each encoder has a residual connection around it, and it is followed by a layer normalization step.

❖ **Positional encoding:**

- Information about the relative positions of words are embedded in vector of same dimensions of word embedding.

Attention Is All You Need (vandergoten.ai)

The Annotated Transformer (harvard.edu)

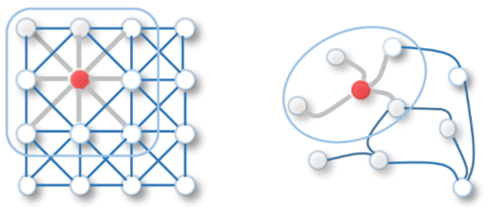
Transformer model for language understanding | Text | TensorFlow

10.7. Transformer — Dive into Deep Learning 0.16.6 documentation (d2l.ai)

Transformer Architecture: The Positional Encoding - Amirhossein Kazemnejad's Blog

==

BEYOND EUCLIDEAN DATA: GEOMETRIC DEEP LEARNING



(a) 2D Convolution. Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size.

(b) Graph Convolution. To get a hidden representation of the red node, one simple solution of the graph convolutional operation is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are unordered and variable in size.

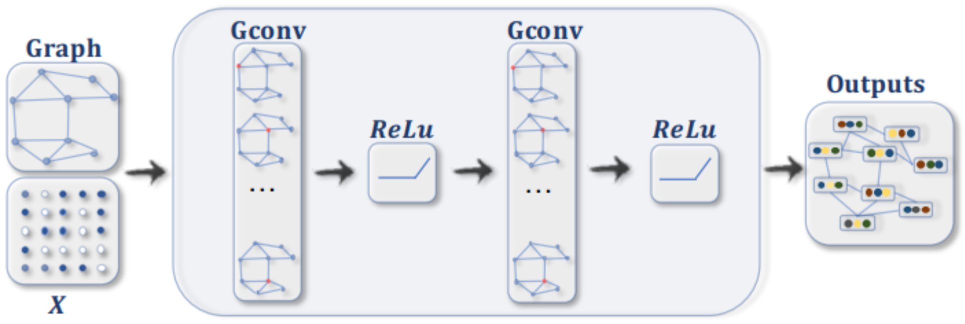
Fig. 1: 2D Convolution vs. Graph Convolution.

❖ Paper: A Comprehensive Survey on Graph Neural Networks
  - In real life we must deal with data generated from non-Euclidean domains and are represented as graphs with complex relationships and interdependency between objects. The complexity of graph data has imposed significant challenges on existing machine learning algorithms.
  - divide the state-of-the-art graph neural networks into four categories, namely
    - **recurrent graph neural networks**
      - ♦ learn node representations with recurrent neural architectures.
      - ♦ They assume a node in a graph constantly exchanges information/message with its neighbors until a stable equilibrium is built
      - ♦ the idea of message passing is inherited by space based convolutional graph neural networks.

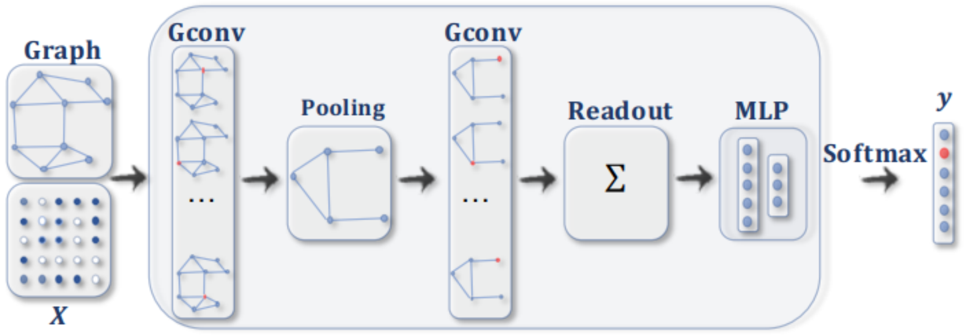
CONVOLUTIONAL GRAPH NEURAL NETWORKS

- ♦ generalize the operation of convolution from grid data to graph data.
- ♦ **Main idea** -generate a node v's representation by aggregating its own features x_v and neighbors' features x_u, where $u \in N(v)$.
- ♦ Different from RecGNNs, ConvGNNs stack multiple graph convolutional layers to extract high-level node representations.

- ConvGNNs play a central role in building up many other complex GNN models.



(a) A ConvGNN with multiple graph convolutional layers. A graph convolutional layer encapsulates each node's hidden representation by aggregating feature information from its neighbors. After feature aggregation, a non-linear transformation is applied to the resulted outputs. By stacking multiple layers, the final hidden representation of each node receives messages from a further neighborhood.

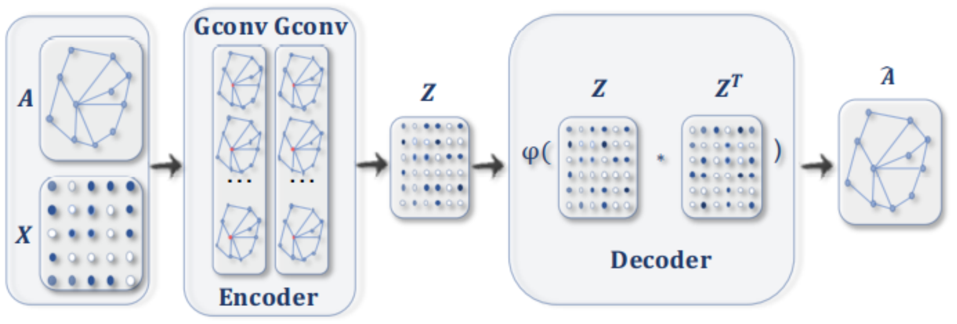


(b) A ConvGNN with pooling and readout layers for graph classification [21]. A graph convolutional layer is followed by a pooling layer to coarsen a graph into sub-graphs so that node representations on coarsened graphs represent higher graph-level representations. A readout layer summarizes the final graph representation by taking the sum/mean of hidden representations of sub-graphs.

GRAPH AUTO ENCODERS,

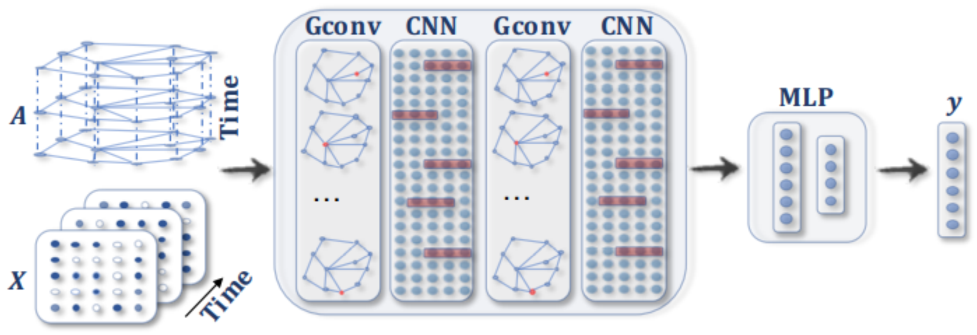
- unsupervised learning frameworks
- encode nodes/graphs into a latent vector space and reconstruct graph data from the encoded information.
- GAEs are used to learn network embeddings and graph generative distributions.
  - For network embedding, GAEs learn latent node representations through reconstructing graph structural information such as the graph adjacency matrix.

- For graph generation, some methods generate nodes and edges of a graph step by step while other methods output a graph all at once.



(c) A GAE for network embedding [61]. The encoder uses graph convolutional layers to get a network embedding for each node. The decoder computes the pair-wise distance given network embeddings. After applying a non-linear activation function, the decoder reconstructs the graph adjacency matrix. The network is trained by minimizing the discrepancy between the real adjacency matrix and the reconstructed adjacency matrix.

- spatial-temporal graph neural networks.
  - aim to learn hidden patterns from spatial-temporal graphs, which become increasingly important in a variety of applications such as traffic speed forecasting, driver maneuver anticipation, and human action recognition.
  - idea -consider spatial dependency & temporal dependency at the same time.
  - Many current approaches integrate graph convolutions to capture spatial dependency with RNNs or CNNs to model the temporal dependency.

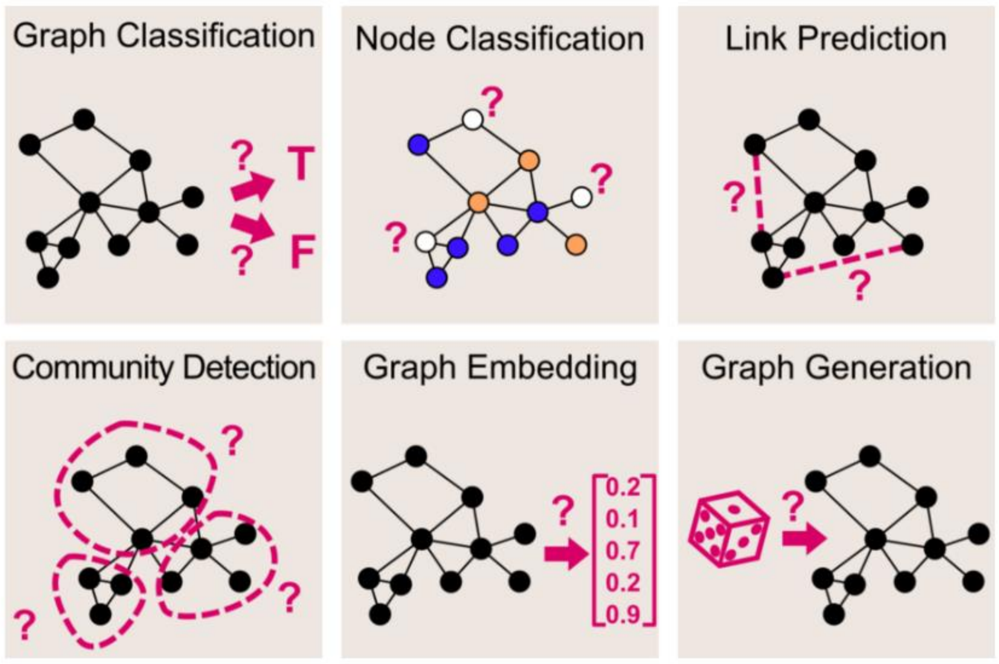


(d) A STGNN for spatial-temporal graph forecasting [74]. A graph convolutional layer is followed by a 1D-CNN layer. The graph convolutional layer operates on A and $X^{(t)}$ to capture the spatial dependency, while the 1D-CNN layer slides over X along the time axis to capture the temporal dependency. The output layer is a linear transformation, generating a prediction for each node, such as its future value at the next time step.

GRAPH LEARNING TASKS

graph data are non-structured and non-Euclidean.

One the one hand the connections between nodes carry essential information, on the other hand it is not trivial to find a way to process this kind of information.



Machine Learning tasks on graphs (image by author)

Link TO PAPER: [Graph Convolutional Networks —Deep Learning on Graphs | by Francesco Casalegno | Towards Data Science]

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

WHY CONVOLUTIONS?

incredibly efficient at extracting complex features, and convolutional layers nowadays represent the backbone of many Deep Learning models. CNNs have been successful with data of any dimensionality:

in 1D, to process audio signals — e.g. for sound classification

in 2D, to process images — e.g. for early caries detection

in 3D, to process scans — e.g. for MRI brain registration

What makes CNNs so effective is their ability to learn a sequence of filters to extract more and more complex patterns.

Defining graph convolution

On Euclidean domains, convolution is defined by taking the product of translated functions. translation is undefined on irregular graphs, so we need to look at this concept from a different perspective.

the key idea is to **use a Fourier transform**. In the frequency domain, thanks to the Convolution Theorem, the (undefined) convolution of two signals becomes the (well-defined) component-wise product of their transforms.

$$x * y = \mathcal{F}^{-1}\{\mathcal{F}\{x\} \cdot \mathcal{F}\{y\}\}$$

Convolution theorem (image by author)

- how do we define a graph Fourier transform?

  Let's take the case of a function defined on the real line. Its Fourier transform is its decomposition in frequency terms, obtained by projecting the function on an orthonormal basis of sinusoidal waves. And in fact, these waves are precisely the eigenfunctions of the Laplacian:

$$\frac{d^2}{dx^2}u = \lambda u \quad \rightarrow \quad u_\omega(t) = e^{i\omega t} \qquad \hat{x}(\omega) = \int_{-\infty}^{+\infty} e^{-i\omega t}\, x(t)\,dt \qquad \omega \in \mathbb{R}$$

Fourier transform in 1D (image by author)

we can define the Fourier transform of a function as its projection on an orthonormal basis of eigenfunctions of the Laplacian.

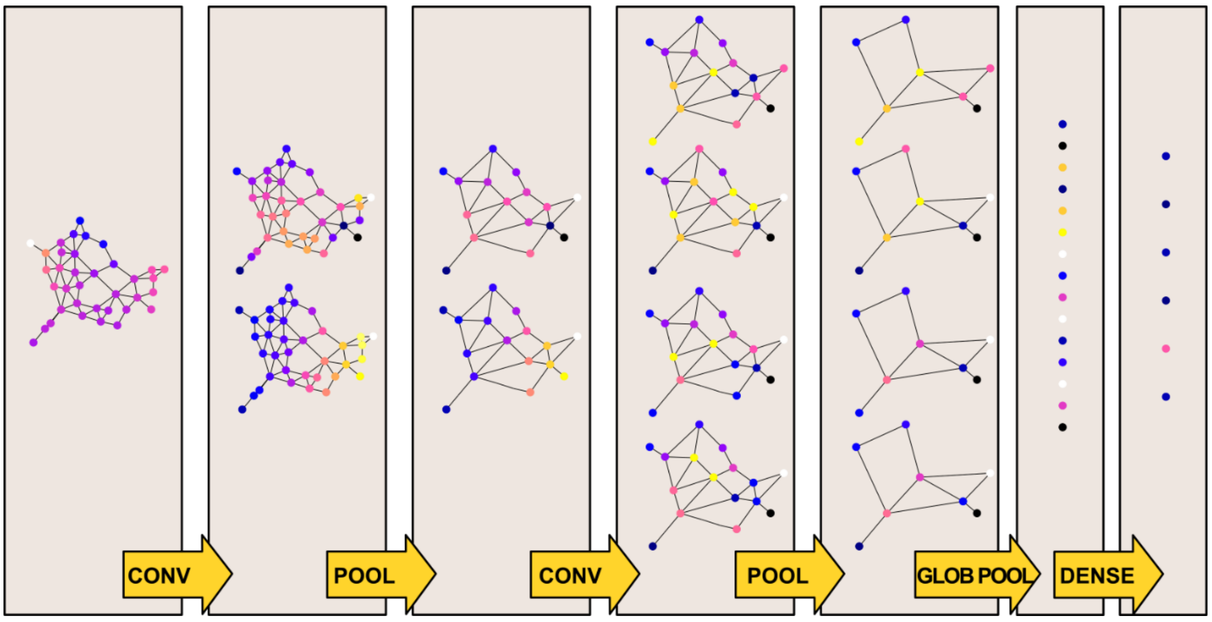
Laplacian Matrix [L] = Degree Matrix [D]- adjacency Matrix[A]

BUILDING THE FULL NEURAL NETWORK

Architecture of CNN for img recognition used similar structure.

1. Features are extracted by passing the **HxWxC** input image through a series of localized convolution filters and pooling layers.
2. The resulting feature channels are mapped into a fixed-size vector using e.g. a global pooling layer.
3. Finally, a few fully-connected layers are used to produce the final classification output.

architecture of Graph Convolution Networks follows exactly the same structure!



case of a GCN, our input is represented by the following elements:

- the NxC array x containing, for each of the N nodes of the graphs, C features
- the NxN adjacency matrix

Conclusions

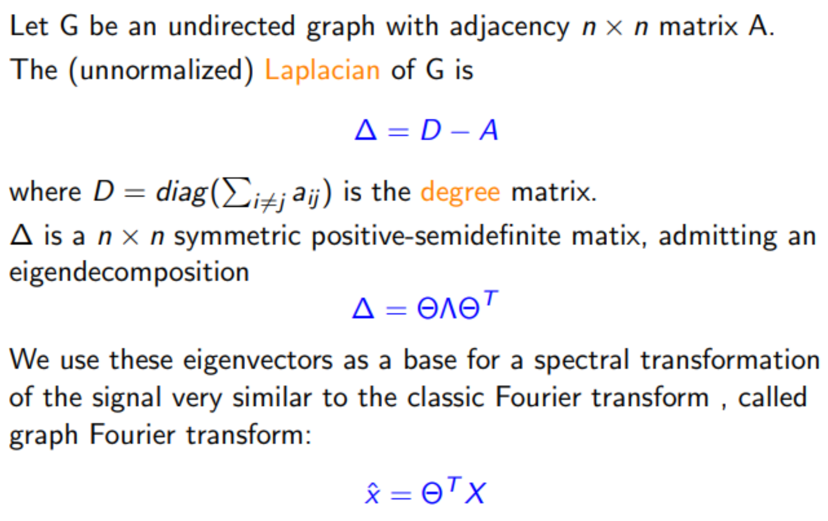
These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

- From knowledge graphs to social networks, graph applications are ubiquitous.
- Convolutional Neural Networks (CNNs) have been successful in many domains, and can be generalized to Graph Convolutional Networks (GCNs).
- Convolution on graphs are defined through the graph Fourier transform.
- The graph Fourier transform, on turn, is defined as the projection on the eigenvalues of the Laplacian. These are the "vibration modes" of the graph.
- As for traditional CNNs, a GCN consists of several convolutional and pooling layers for feature extraction, followed by the final fully-connected layers.
- To ensure that the convolutional filters have compact support, we use a polynomial parametrization. Chebyshev polynomials allow to reduce the computational complexity.

Graph Laplacian

EIGENVECTORS:

Encode info on graphs of increasing frequencies.


Let G be an undirected graph with adjacency $n \times n$ matrix A. The (unnormalized) Laplacian of G is

$$\Delta = D - A$$

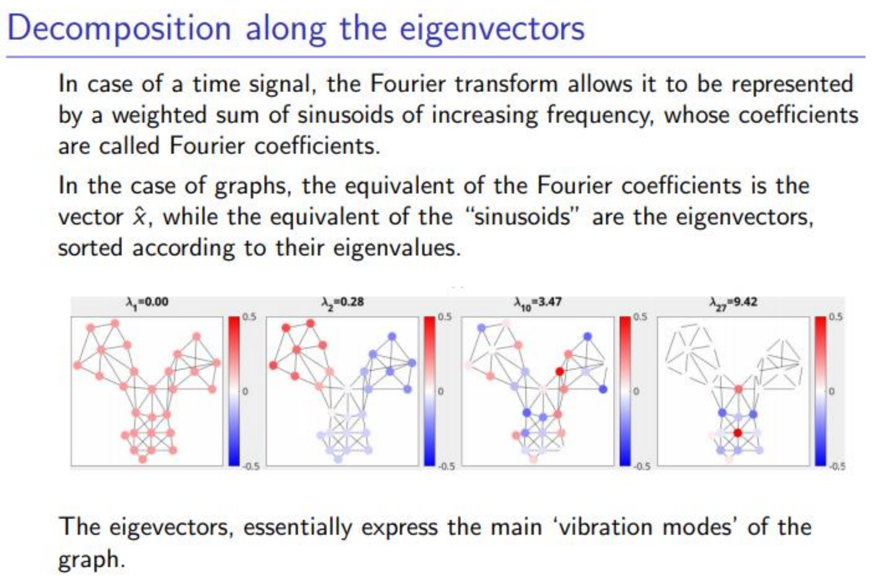
where $D = diag(\sum_{i \neq j} a_{ij})$ is the degree matrix.
Δ is a $n \times n$ symmetric positive-semidefinite matix, admitting an eigendecomposition

$$\Delta = \Theta \Lambda \Theta^T$$

We use these eigenvectors as a base for a spectral transformation of the signal very similar to the classic Fourier transform , called graph Fourier transform:

$$\hat{x} = \Theta^T X$$

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.



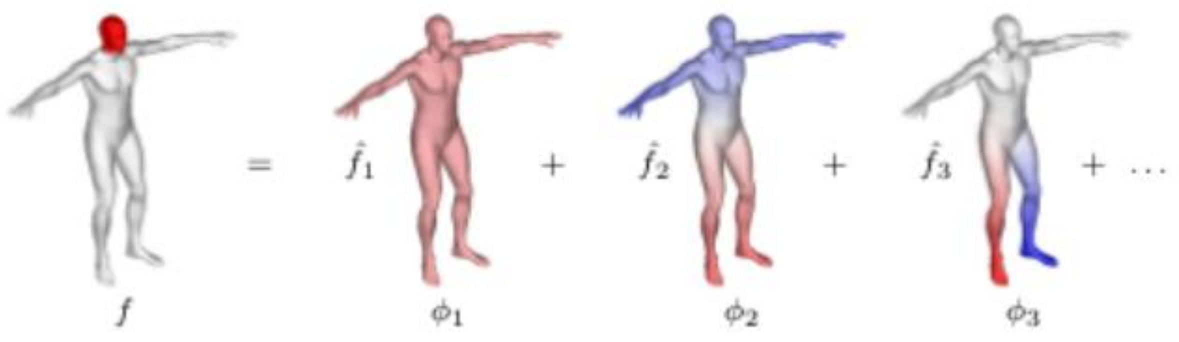
Decomposition along the eigenvectors

In case of a time signal, the Fourier transform allows it to be represented by a weighted sum of sinusoids of increasing frequency, whose coefficients are called Fourier coefficients.

In the case of graphs, the equivalent of the Fourier coefficients is the vector $\hat{x}$, while the equivalent of the "sinusoids" are the eigenvectors, sorted according to their eigenvalues.

The eigevectors, essentially express the main 'vibration modes' of the graph.

Fourier Transformation



Fourier basis = Laplacian eigenfunctions: $\Delta\phi_k(x) = \lambda_k\phi_k(x)$

GCN corresponds to substituting multiplication by Λ with a simpler linear filter on the eigenvalues of the Laplacian matrix.

Graph convolutional networks: a comprehensive review | Computational Social Networks | Full Text (springeropen.com)

===

SPACIAL METHODS IN GEOMETRIC DEEP LEARNING

[1706.02216] Inductive Representation Learning on Large Graphs (arxiv.org)

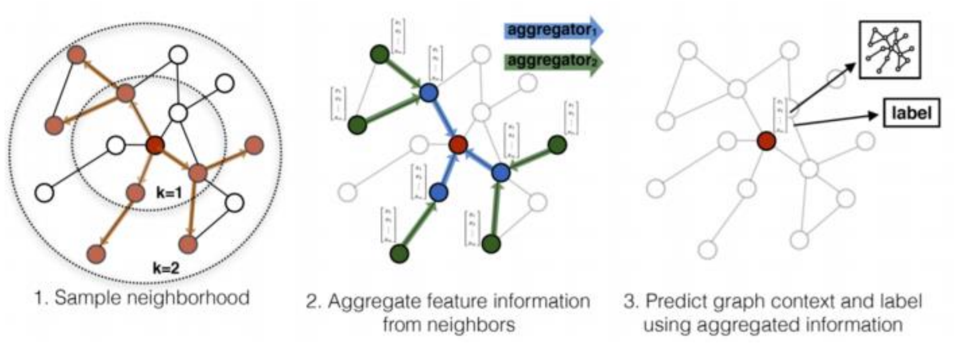
- ❖ Propagating based on Graph CN
  - ➢ A traditional convolution computes the output value for a node as a (learned) linear composition of its neighbours.
  - ➢ We look at efficient way to compute op value fr node as leared aggregation of its adjacent nodes.

$$\hat{x}_i = \underset{j}{\square} h_\theta(x_i, y_i)$$

where

- ☐ is the **aggregation** operator
- ▪ - $h_\theta(\cdot,\cdot)$ is a learnable **edge feature function**

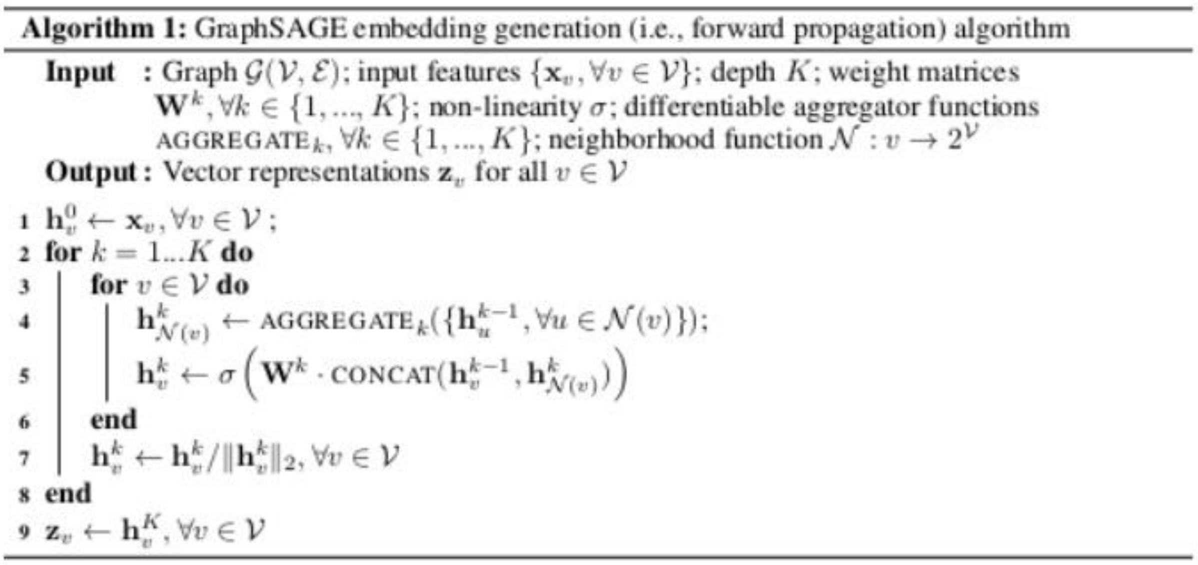
- ❖ Spatial approaches
  - ➢ GraphSage
    - ▪ Processing is composed of three steps:



1. Sample neighborhood 2. Aggregate feature information from neighbors 3. Predict graph context and label using aggregated information

- Neighbourhood sampling
- Aggregation
- Prediction

- ▪ GS forward pass: Each aggregator function aggregates information from a different number of hops - or search depth - away from a given node.



Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

```
Input  : Graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices
         $\mathbf{W}^k, \forall k \in \{1,...,K\}$; non-linearity $\sigma$; differentiable aggregator functions
         $\text{AGGREGATE}_k, \forall k \in \{1,...,K\}$; neighborhood function $\mathcal{N} : v \to 2^{\mathcal{V}}$
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;
for $k = 1...K$ do
    for $v \in \mathcal{V}$ do
        $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
        $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
    end
    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$
```

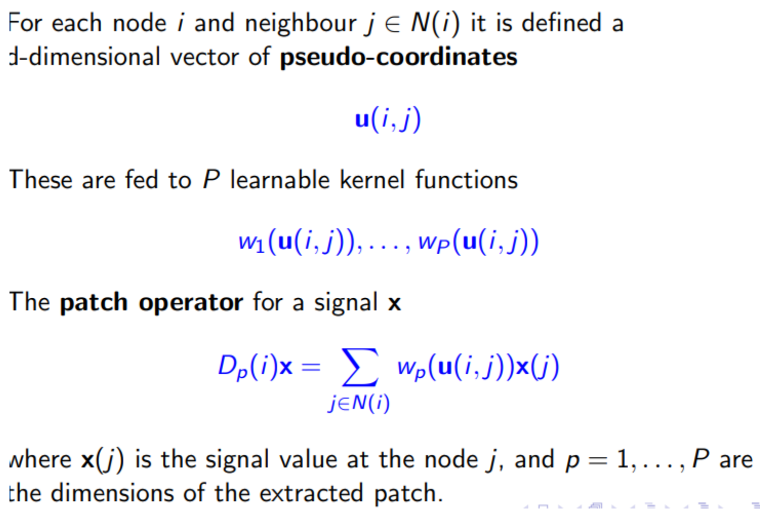
- **Aggregation:**
  - <u>Mean Ag</u> - averaging all the neighborhood node features (can be weighted average)
  - <u>LSTM Ag</u> - using an LSTM cell to sequentially aggregate neighborhood node features (ordered randomly)
  - <u>Pooling Ag</u> - Max pooling only takes the "highest" feature into consideration (performed the best in experiments)
- **Prediction and training**
  - Results after Ag are compared to feature map extracted from an img after the conv layers (excluding top dense layers)
  - aggregated neighbourhood node features collected by each node are used to compute a resulting value, according to the logic of the problem (node classification, structure/context determination.
  - This is where the learning happens, according to problem-specific loss functions

MIXTURE MODEL NETWORK (MONET)

- 3 important contributions to the field of geometric deep learning:
  - <u>**generalization**</u> of various Graph Learning approaches, *unifies spatial and spectral approaches*
  - **parametric kernels** and **pseudo-coordinates**, integrating well with existing models (Anistropic CNN, Geodesic CNN, . . .)

- a wide series of experiments performed on different benchmark manifolds, graphs, and networks

▪ Pseudo coordinates and path operators à



For each node i and neighbour $j \in N(i)$ it is defined a d-dimensional vector of **pseudo-coordinates**

$$\mathbf{u}(i,j)$$

These are fed to P learnable kernel functions

$$w_1(\mathbf{u}(i,j)), \ldots, w_P(\mathbf{u}(i,j))$$

The **patch operator** for a signal $\mathbf{x}$

$$D_p(i)\mathbf{x} = \sum_{j \in N(i)} w_p(\mathbf{u}(i,j))\mathbf{x}(j)$$

where $\mathbf{x}(j)$ is the signal value at the node j, and $p = 1, \ldots, P$ are the dimensions of the extracted patch.

- Flexibility of Pseudo coords: many gcn are sub cases of Monet->

By carefully selecting pseudo cooridnates u(i, j) and kernel functions wp(u) many known graph convolutional network models can be viewed as a specific case of MoNet.

▪ Learning weights
  - Monet gives smoother weight functions

==
==============

BASIC RE ENFORCEMENT LEARNING

❖ Agent – interacts with enviornment and provides numeric rewards.
This is based on time, policy & steps.
policy π(a|s) à probability distribution of actions given states.
- Agent at time t selects action a/c policy
- Enviornment answers with local reward
- Agents now moves into new state

Utility- the utility of taking action *a* in some state *s* is the expected immediate reward for that action plus the sum of the long-term rewards over the rest of the agent's lifetime, assuming it acts using the best policy.

- ❖ Best way to act à **best policy** [actions taken in order to maximize their utility in the pursuit of some goals.]

  Objective? Maximise future comulative rewards.

  R = $\text{Sum}_{1<=i}$ (r,i)

- ❖ Future Discount cumulative Rewards
  - Chances of picking rewards that are far are less likely than closer ones, that are more predictable.
  - multiply the reward by a discount rate $0 < \gamma <= 1$ exponentially decreasing with time.
  - **discount factor** essentially determines how much the reinforcement learning agents cares about rewards in the distant future relative to those in the immediate future. If $\gamma=0$, it will only choose those action which have short term rewards.

- ❖ **Markov Decision Process**
  - ➢ Current state completely characterises the state of the world: future actions only depend on the current state.
    - ▪ Defined by a tuple (S, A, R,P, γ)
      - S: set of possible states
      - A: set of possible actions
      - R: reward probability given (state, action) pair
      - P: transition probability to next state given (state, action) pair
      - γ: discount factor

  The optimal policy
  - a policy that maximizes the value of all states at the same time
  - Policy produces trajectories/paths.
  - We find the optimal policy à

$$\pi^* = argmax_\pi \mathbb{E} \sum_{t \geq 0} \gamma^t r_t$$

- ❖ **Model Free vs Model based**
  - ➢ Model-based reinforcement learning has an agent try to understand the world and create a model to represent it. Here the model is trying to capture 2 functions, the transition function from states T and the reward function R. From this model, the agent has a reference and can plan accordingly.
  - ➢ However, it is not necessary to learn a model, and the agent can instead learn a policy directly using algorithms like Q-learning or policy gradient.
  - ➢ A simple check to see if an RL algorithm is model-based or model-free is:
  - ➢ *If, after learning, the agent can make predictions about what the next state and reward will be before it takes each action, it's a model-based RL algorithm.*
  - ➢ If it can't, then it's a model-free algorithm.

- transition from state st to state st+1 is not always determinstic, but governed by some probability P(st+1 | st , at).
- If the learning model needs to learn this probability P(st+1 | st , at), then it is called model-based.
- In model-free approaches, this information is left implicit: you learn to take actions from past experience relying on trial-and-error.

❖ **Exploration/Exploitation** trade of

- Exploration is finding more information about the environment.
- Exploitation is taking advantage of the available information to maximize the reward.

❖ **Model free approaches**

- Value-based --we shall choose the action taking us to the next state with the best evaluation. we don't store any explicit policy, only a value function. The policy is here implicit and can be derived directly from the value function (pick the action with the best value).
- Policy-Based --we directly try to improve the current policy, hopefully optimizing it. methods we explicitly build a representation of a policy (mapping π:s→a) and keep it in memory during learning.
  - policy defines the agent behavior at a given state: a = π(s)
  - better, π(s) is the probability to perform a in state s.

❖ **Value-based approaches**

- Value function and Q-function.
  - **Value f()** – tells about *how good is the state*.
  - **Q function** – tells *how good an action* a is for the state- s.
- Relation bwtween Value & Q:
  - Sum every action-value weighted by the probability π(a | s) to take that action.
  - EASY TO COMPUTE V from Q
  - TO COMPUTE Q from V à we need model based approach!
    - S à state ; aà action.

$$V(s) = \sum_a \pi(a|s) * Q(s,a)$$

$$Q(s,a) = \sum_{s'} \mathcal{P}(s'|s,a) * V(s')$$

❖ **Optimal policy:**

- Q-value function Q∗ (s, a) is the maximum expected cumulative reward achievable from state s performing action a

$$Q^*(s,a) = max_\pi \mathop{\mathbb{E}}_{\substack{s_0=s \\ a_0=a}} \sum_{t\geq 0} \gamma^t r_t$$

❖ Bellman eq

- expresses a relation between the solution for a given problem in terms of the solutions for subproblems.

Q^* satisfies the following Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q^*(s',a')]$$

Indeed, $R_{s'} = max_{a'} Q^*(s',a') = V^*(s')$ is the optimal future cumulative reward from s', and the optimal future cumulative reward from s when taking action a is $r_0 + \gamma R_{s'}$

optimal policy π ∗ consists in taking the best action in any state as specified by Q*

- Computing Q ∗ via iterative update
  - Q∗ satisfies the Bellman equation: $Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q^*(s',a')]$
  - perform iterative update on progressive approximations Qi of Q∗ :

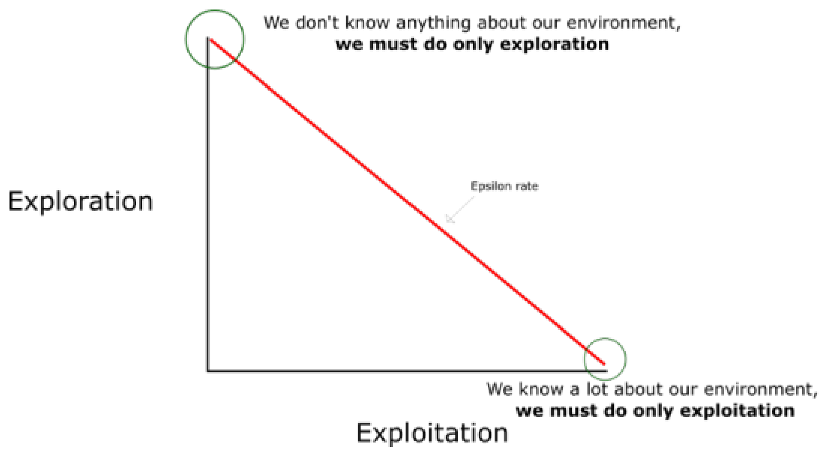
$$\underbrace{Q^{i+1}(s,a)}_{\text{next estimation}} \leftarrow \underbrace{Q^i(s,a)}_{\text{current estimation}} + \alpha(\underbrace{r_0 + \gamma max_{a'} Q^i(s',a') - Q^i(s,a)}_{\text{recursive update}})$$

RECURSIVE UPDATE : derivation of quadratic distance between Qi(s,a) & $r0 + \gamma max_{a'} Q^i (s\,0 , a\,0)$

  - Add more details about Q learning

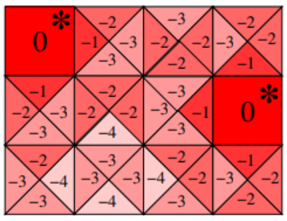
- Exploration vs. exploitation
  - At start, the Q-table is not informative. Taking actions according to it could introduce biases, and prevent exploration.
  - early stages à random exploration. Later rely on table

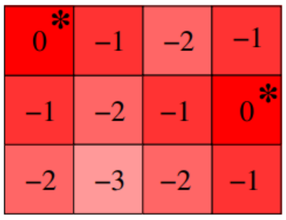
- **Epsilon Greedy Strategy**
  - exploration rate , initially equal to 1.
  - Generate random number, if it is greater than Exploration rate – refer to Q table to the info collected, else choose random exploration
  - Reduce Exploration rate with time.


We don't know anything about our environment,
we must do only exploration
Epsilon rate
Exploration
We know a lot about our environment,
we must do only exploitation
Exploitation

❖ Q value and V value:

```
for n in range(0,episodes):
  s0 = random_state()
  while not term(s0):
      #choose action
      if np.random.random() > epsilon:
          a = np.argmax(Qtable[s0]) #exploit Qtable
      else:
          a = np.random.randint(4) #random move
      s1 = move(s0,a)
      T = term(s1)
      if T:
          R = -1
      else:
          R = -1 + gamma*np.max(Qtable[s1])
      Qtable[s0][a] = Qtable[s0][a] + alpha*(R-Qtable[s0][a])
      s0 = s1
  epsilon = epsilon * epsilon_rate #decrease epsilon
```


Optimal Q-value



Optimal V-value

e V-value is just the max of the Q-values, over all possible ions:

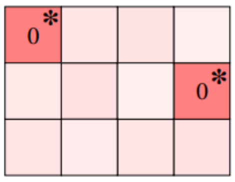
$$V(s) = max_a Q(s,a)$$

❖ How learning takes place in practice?

When we start, we only know the right values for terminal state

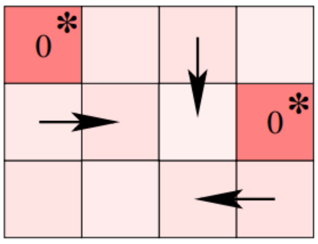
The other states will get a random value.

Actions = {
1. right
2. up
3. left
4. down
}

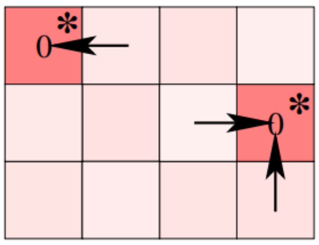


A negative rewa
for each transit
(e.g. r = -1)

Most of the actions produce meaningless updates, since the current estimation of the Q-value function is erroneous



The relevant actions are those leading to states whose Q-value is accurate; at the beginning these are just terminal states



Integrating with Deep Learning : DQN

- Q-learning exploits Bellman's equation

$$Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q^*(s',a')]$$

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

We compute Q^* via iterative updates:

$$\underbrace{Q^{i+1}(s,a)}_{\text{next estimation}} \leftarrow \underbrace{Q^{i}(s,a)}_{\text{current estimation}} + \alpha\underbrace{(r_0 + \gamma max_{a'} Q^{i}(s',a') - Q^{i}(s,a))}_{\text{recursive update}}$$

$Q^i \rightarrow Q^*$ when $i \rightarrow \infty$.

Not scalable!

Must compute $Q(s,a)$ for every state-action pair.

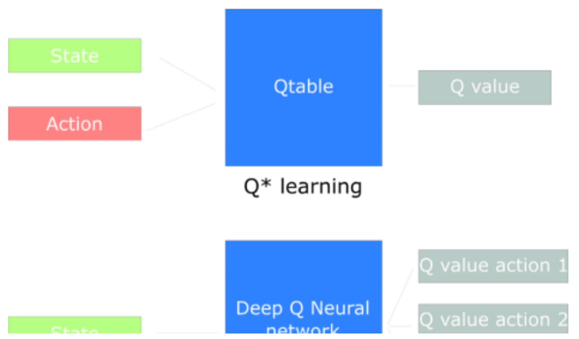
DEEP Q LEARNING

Deep Q-learning: use a function approximator (a Neural Network!) to estimate the optimal action-value function

$$Q(s,a,\theta) \approx Q^*(s,a)$$

θ are the function parameters to be learned.

Instead of taking a as input, it is customary to return a value for each possible action a (the two functions are isomorphic)



Loss Function

Given (s,a) the current Q-value estimate of the network

$$Q(s,a,\theta)$$

The expected value, given by the Bellman equation is

$$\mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q(s',a',\theta)]$$

$L(\theta) = (\mathbb{E}_{s'}[r + \gamma max_{a'} Q(s',a')] - Q(s,a,\theta))^2$ t the fixpoint

$$L(\theta) = (\mathbb{E}_{s'}[r_0 + \gamma max_{a'} Q(s',a')] - Q(s,a,\theta))^2$$

Experience Replay:

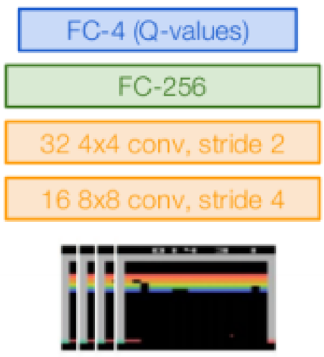
- For learning – we need loss function and gradient

These notes are not a substitute to the original slides provided by professor but shall only be considered as additional material for the course.

- To compute them we need transitions. $(s_i, a_i, r_i, T_i, s_{i+1})$
- We can store these transitions in experience memory, and replay them at leisure training {experience reply}
- Better than learning for the batches of consecutive samples coz-
  - Samples tend to be correlated à inefficient learning
  - Greater risk of introducing biases during learning

The Atari Q-learning architecture

A single feed-forward pass to compute Q-values for all actions from the current state



Last layer has an output for each action a, returning Q(s,a).

Number of actions between 4-18 depending on Atari game

state: 84x84x4 stack of last 4 frames (RGB→grayscale conversion, rescaling)

Why so we stack frames?

- To capture the movements
- Alternatevly we can use LSTM layer (after processing the state, and before computing Q values)

Atari Games and Q learning

- Works well for reactive game, not for planning games!

Rewards:

- Differ game to game

Links: https://arxiv.org/abs/1707.06203

https://arxiv.org/pdf/1605.01335.pdf

https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

===
==

Deep Q learning Improvements

- Fixed Q-targets

- Double Q-learning

- Prioritized Experience Replay

- Dueling

- Noisy Networks

- Distributional RL

- Rainbow

❖ Fixed Q targets

- ➢ With Q-learning, we try to approximate the optimal target Q-function Q∗ , through progressive updates:

  Q ∗ (s, a) − Q(s, a)

- ➢ Moreover we approximate computation of Q*(s,a) as ro + maxa 0Q(s 0 , a 0), giving us q learning rate-

$$\underbrace{r_0 + \gamma max_{a'} Q(s', a')}_{\text{approximated target } Q^*} - \underbrace{Q(s, a)}_{\text{current estimation}}$$

- ➢ Shared weights- in loss function the same nw is used to provide two different estimates of Q functions:

$$\underbrace{r_0 + \gamma max_{a'} Q(s', a', w)}_{\text{approximated target } Q^*} - \underbrace{Q(s, a, w)}_{\text{current estimation}}$$

- ➢ At every step of training, Q value shifts but also the "target value" shifts

  We are getting closer to target -> but target also keeps moving causing oscillations in training

- ➢ Fixed Q targets

- Use a separate network Q with fixed parameters for estimating the TD target.

$$\underbrace{r_0 + \gamma max_{a'} \overline{Q(s',a')}}_{\text{approximated target } Q^*} - \underbrace{Q(s,a)}_{\text{current estimation}}$$

-
- Periodically, copy the parameters from Q to Q, to update the target network.

➢ Implementation

Implementing fixed q-targets is straightforward:

- create two (identical) networks: DQNetwork, TargetNetwork
- define a function to transfer parameters from DQNetwork to TargetNetwork
- during training, compute the TD target using our target network. Update the target network with the DQNetwork every tau steps (tau is a user-defined hyper-parameter).

❖ Double Q learning:

➢ Action values overestimates

- approximation of target action value is computed using a maximum over actions:

$$\underbrace{r_0}_{\text{local reward for taking action a}} + \gamma \cdot \underbrace{max_{a'} Q(s',a')}_{\text{max Q-value over all possible actions}}$$

- Approximation is noisy, it is possible to prove that this will eventually result in +ve bias à overestimation of correct value.

❖ Decoupling action choice and its estimations

➢ https://papers.nips.cc/paper/3964-double-q-learning

➢ The Double Q-learning approach consists in decoupling the choice of the action from its estimation, using two networks QA and QB .

```
1.  Initialize $Q^A, Q^B, s$
2.  **repeat**
3.      choose $a$ using $\epsilon, Q^A, Q^B$; observe $r, s'$
4.      choose (e.g. random) between UPDATE-A and UPDATE-B
5.      **if** UPDATE-A: # use $Q^A$ to choose action, $Q^B$ to estimate it
6.          $a^* = arg\ max_a Q^A(s', a)$
7.          $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(r + Q^B(s,a^*) - Q^A(s,a))$
8.      **else if** UPDATE-B: # use $Q^B$ to choose action, $Q^A$ to estimate it
9.          $a^* = arg\ max_a Q^B(s', a)$
10.         $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(r + Q^A(s,a^*) - Q^B(s,a))$
11.     $s \leftarrow s'$
12. **until** end
```

If we use Q^A to select best action $a^* = arg\ max_a Q^A(s', a)$, the value of $Q^A(s', a^*)$ could be biased by the choice (we know it will be large: it was the maximum!).

Since Q^B was updated on the same problem, but with a different set of experience samples, $Q^B(s', a^*)$ provides a better, unbiased estimate for the value of action a^*.

Prioritized Experience Replay : https://arxiv.org/abs/1511.05952

- PER:
  - idea – some experience may be more important than others, and thus replay more frequently.

We want to give higher priority to transitions for which there is a large difference between our prediction and the expected target. For a transition $t = (s, a, r, F, s')$, its update is

$$\delta_t = r + \gamma max_{a'} Q(s', a') - Q(s, a)$$

and we set its priority to

$$p_t = |\delta_t|$$

- Stochastic Prioritization
  - The probability of being chosen for replay is computed a/c to following rule: $P_t = \frac{p_t^\alpha}{\sum_t p_t^\alpha}$

If α = 0, all transistions have same probability; if α is large, it priveleges transitions with high priority pt .

High priority sample have more chances of being selected, although there exists risk of overfitting on small portion of experiences that we presume to be interesting.

- Importance sampling weights

We can correct the bias with **importance sampling weights**.
If N is the dimension of the replay buffer, then

$$w_t = (N \cdot P_t)^{-\beta}$$

that compensate the non-uniform probabilities P_t when $\beta = 1$ (if some transition has a high probability, we reduce its weight).

These weights are folded into the Q-learning update by using $w_t \delta_t$ instead of δ_t.

For stability reasons, weights are normalized by $1/(max_t w_t)$ so that they only scale the update δ_t downwards.

- β is initialized to 1 and goes to 0 during training.

- Dueling
  - Advantage

Each Q-value $Q(s, a)$ estimates how good it is to take action a in state s: it depends both on the action a and the value of the given state s.

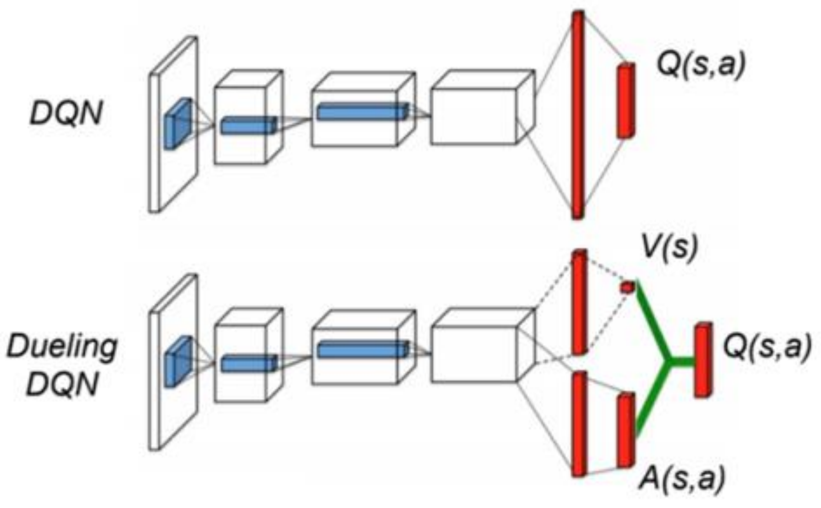
We can decompose $Q(s, a)$ as the sum of:

- V(s): the value of being at state s
- A(s,a): the advantage of taking action a in state s, measuring how much better is to take action a versus all other possible actions at that state.

$$Q(s, a) = \underbrace{V(s)}_{value} + \underbrace{A(s, a)}_{advantage}$$

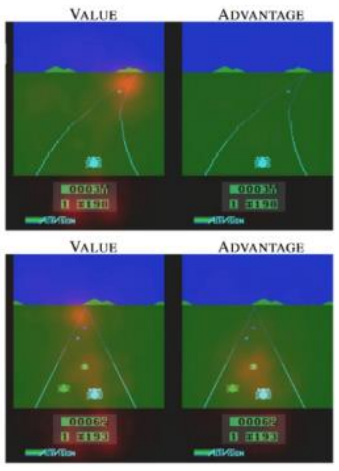
  - Dueling DQN : https://arxiv.org/pdf/1511.06581.pdf

Dueling Network Architectures (DDQN) split the computation of $V(s)$ and $A(s,a)$ in two different streams:



- Dueling learns whether states are valuable or not without having to learn the effect of each action for each state.
- Useful when-> actions don't affect the env in any way
- Converslyà when action is relevant, if can focus on advantage without caring for the current evaluation of state.

➢ Saliency maps on Enduro : https://arxiv.org/abs/1312.6034
  - For extimaing Value the n/w focus on
    - Horizon
    - score
  - Saliency maps- orange blurs, computed with jacobians (partial derivatives) on ip images



no car close by: action is not relevant

pay attention to the car in front: action is crucial

❖ Naif aggregation

$$Q(s,a) = V(s) + A(s,a)$$

❖ Aggregating Values and Advantage

$$\cancel{Q(s,a) = V(s) + A(s,a)}$$

Force A to have **zero advantage** for the best action a^*:

$$Q(s,a) = V(s) + A(s,a) - max_a A(s,a)$$

If $a^* = argmax_a A(s,a)$, $Q(s,a^*) = V(s)$ and $A(s,a^*) = 0$.

- Alternative: mean instead of Max
  - Experimentally, replacing max with mean seem to work better

$$Q(s,a) = V(S) + A(s,a) - mean_a A(s,a)$$

  - Subtracting the mean helps to improve stability during training.

- **Noisy Networks :**Link
  - Noise helps in random explorations.
  - Randomicity in choice of actions
  - As noisy weights are learned, resulting noise is state dependent à n/w randomly explores env at diff rates in diff part of state space.
  - Bettern tha epsilon greedy strategy

Noisy Networks are characterized by **noisy dense layers**, combining a deterministic and a noisy stream:

$$y = \underbrace{b + Wx}_{usual\ layer} + \underbrace{(b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^w)x)}_{noisy\ stream}$$

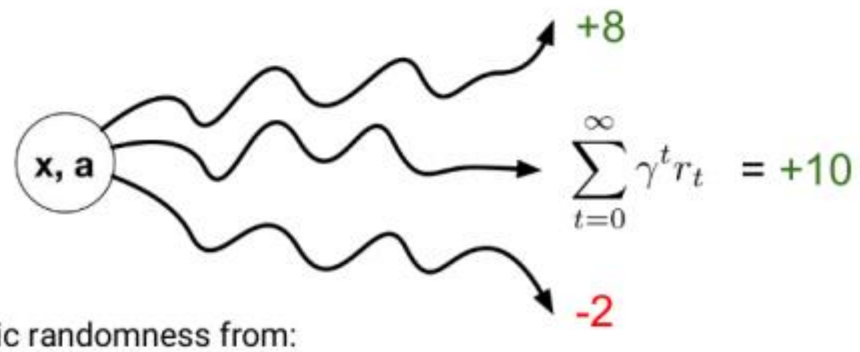
- $W, b, W_{noisy}, b_{noisy}$ are learned parameters
- ϵ^b, ϵ^w are randomly generated

This layer is used in substitution of any standard dense layer (doubling the number of parameters).

- Distributional RL : http://proceedings.mlr.press/v70/bellemare17a/bellemare17a.pdf
  - Try to learn probab distribution of future cumulative reward instead of the traditional approach of modeling the expectation of this return.
  - Works on random return Z whose expectation is value Q.

- The cumulative reward random variable Z : https://physai.sciencesconf.org/data/pages/distributional_RL_Remi_Munos.pdf

The r.v. Return $Z^{\pi}(x,a) = \sum_{t\geq 0} \gamma^t r(x_t, a_t)\big|_{x_0=x, a_0=a, \pi}$



Captures intrinsic randomness from:

- Immediate rewards
- Stochastic dynamics
- Possibly stochastic policy

➢

❖ Distributional Bellman

Let $a*$ be the best possible action in s'.
The Bellman equation is

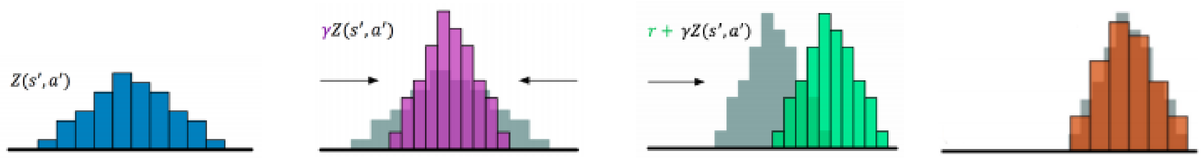
$$Q(x,a) = R(x,a) + \gamma Q(x', a^*)$$

Similarly,

$$Z(x,a) = R(x,a) + \gamma Z(s', a^*)$$

➢

❖ Discretization
- ➢ Discretized over a support in given range b/w Vmin and Vmac using fixed number of bins
- ➢ Loss b/w two distribution can be computed with KL-divergence.



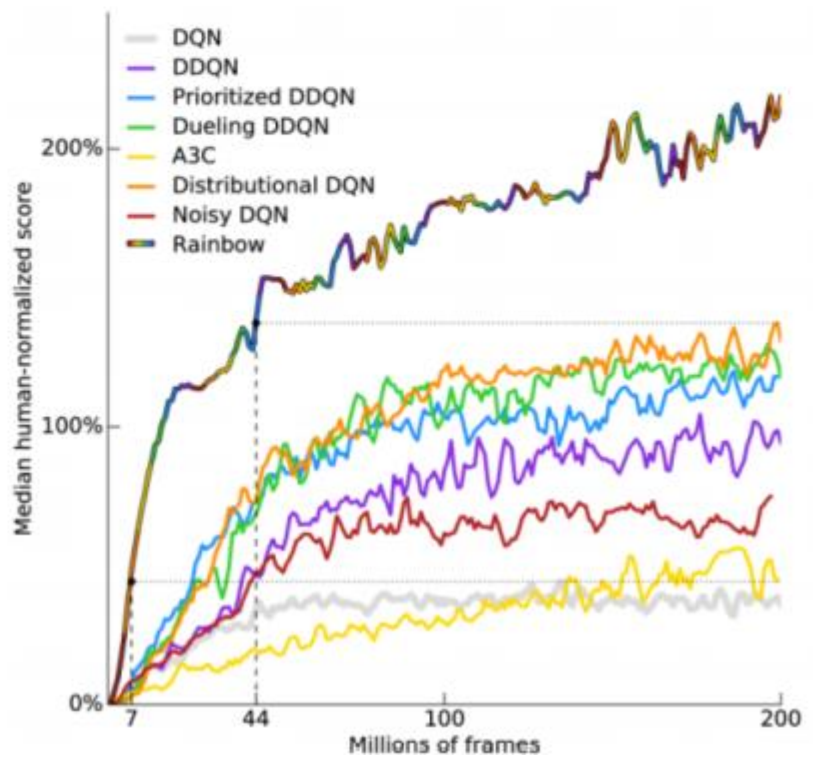
(1) Z at s' (2) discounted Z (3) shift by r (4) project

C51- pseudocode : https://flyyufelix.github.io/2017/10/24/distributional-bellman.html

Algorithm 1 Categorical Algorithm

```
input  A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0,1]$
  $Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$
  $a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$
  $m_i = 0, \quad i \in 0, \dots, N-1$
  for $j \in 0, \dots, N-1$ do
    # Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$
    $\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$
    $b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z$  # $b_j \in [0, N-1]$
    $l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
    # Distribute probability of $\hat{\mathcal{T}} z_j$
    $m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
    $m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
  end for
output  $-\sum_i m_i \log p_i(x_t, a_t)$  # Cross-entropy loss
```

Rainbow-



==

Polilcy Gradient Techniques

- Sarsa
- Actor-critic methods
- A3C and A2C

Problem of Q Learning

- It is 1-step method since it updates the action value Q(s,a) towards the one-step return

$$r + \gamma max_{a'} Q^{i}(s', a').$$

- This directly affects value of state action pair (s,a) that lead to reward.
- The value of other s,a pairs are affected à indirectly though updated value Q(s,a).
- This makes learning slow

Can we learn policy directly?

Finding best policy from collection of policies?

- On policy vs off techniques
  - Q learning is off policy technique. It does not rely on policy and only needs local transitions.
    - Can take advtg of experience replay.
  - On policy techniques try to improve the current policy
    - Sampling of long trajectories a/c to current strategy.
    - Need many diversified trajectories (| | el agents).
- SARSA
  - State Action Rewards State Action
  - Learning Algo similar to Q learning
  - Updating for Q learning-
    - $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$
  - Updating for SARSA:
    - $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$
  - Instead of connsidering best action at time t+1 (greedy choice), we consider actual action a_{t+1} under current policy
- SARSA vs QLearning
  - QL : Single step transitions : (s_i, a_i, r_i, s_{i+1})
  - SARSA mini trajectories (2 steps): $(s_i, a_i, r_i, s_{i+1}, a_{i+1})$

- Mouse and Cliff scenario
  LINK
  With QL- mouse ends up running along the edge of cliff, but occasionally jumping off & plummeting to its death.
  With SARSA – mouse learnt – with time he commits errors- best path is not to run strtaight to cheese along the edge of cliff but take safer route, little far.
  Even if a random action is chosen there – there is less chance of dying.

A mouse (blue) is trying to get to a piece of cheese (green). Additionally, there is a cliff in the map (red) that must be avoided, or the mouse falls and dies.