Deep Learning – CNN RNN LSTM Transformers GPT

Bhadri Vaidhyanathan – Personal Notes

Table of Contents

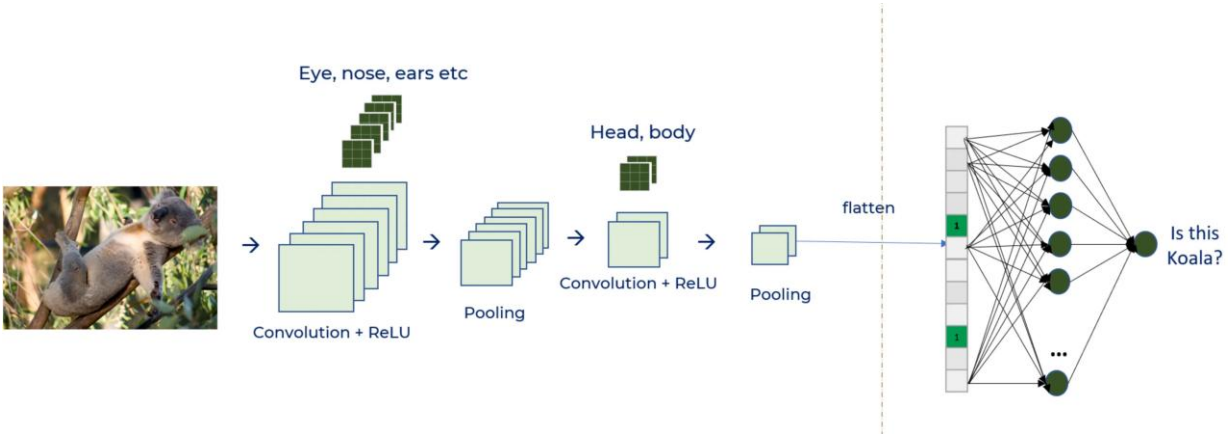
CNN - Convolutional Neural Network

CNNs can remember or properly processes “spatial information” that are really good with images and video data. In MLP/normal NN, the way the image data has to flattenened (matrix to one row form) and feed as one row leads to the loss of this spatial information or interrelation of image cells next to each other. CNNs have other benefits just perfect for image/video data. Besides, empirically, CNN just beats NNs in Computer Vision. All said, a MLP NN is used at the head of most CNNs for the classification task.

Unique aspects of CNN:

Things that make it different from a vanilla NN and what makes it so good for Computer Vision (CV).

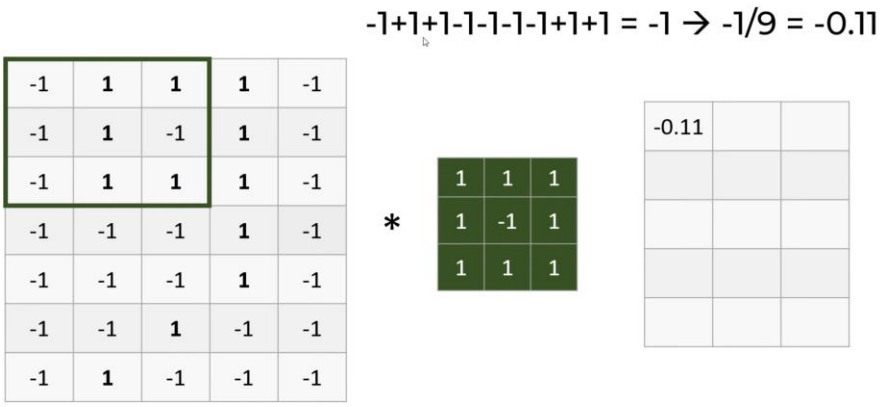
Convolution



Convolution: The operation of sliding a kernel/filter on an image or a feature map to produce a modified output.

A **Convolution** is a mathematical operation performed on a matrix. It is simply a copy of the image after passing thro a filter, the filter could reduce, scaleup, take only edges, only a specfic color or other.
It is simply the operation of taking a kernel applying it on a input matrix and resulting in a treated /convoluted matrix. It is an element-wise multiplication of two matrices followed by a sum and not the normal matrix multiplication.

-0.11 is the result of the **Convolution** operation performed with below kernel:(below a simple average is being done)



The mathematical operations:
Convolution: (the mathematical operation of multiplying values and adding sum to get a number)
Typical flow:
define and take a 'trace'/ kernel / convolution matrix (green matrix below)
The second matrix is the 'big matrix' of the object.
Multiply them, element-by-element (i.e., ***not* the dot-product**, just a simple multiplication) creating the **'Feature Map'**
the result is such that if the object had a loop in that location, the value will be higher. matching matrices, higher number in Feature map
Sum the elements together.
Average the result with the dimension of the matrix . 3X3

Decide on filter matrix. like a matrix only rep the loop of a 9.

Convolutions are one of the most *critical, fundamental building-blocks* in computer vision and image processing. Photoshop – blurring, or smoothing – they are convolutions. 'edge detection' which is key in image classification is also a convolution.

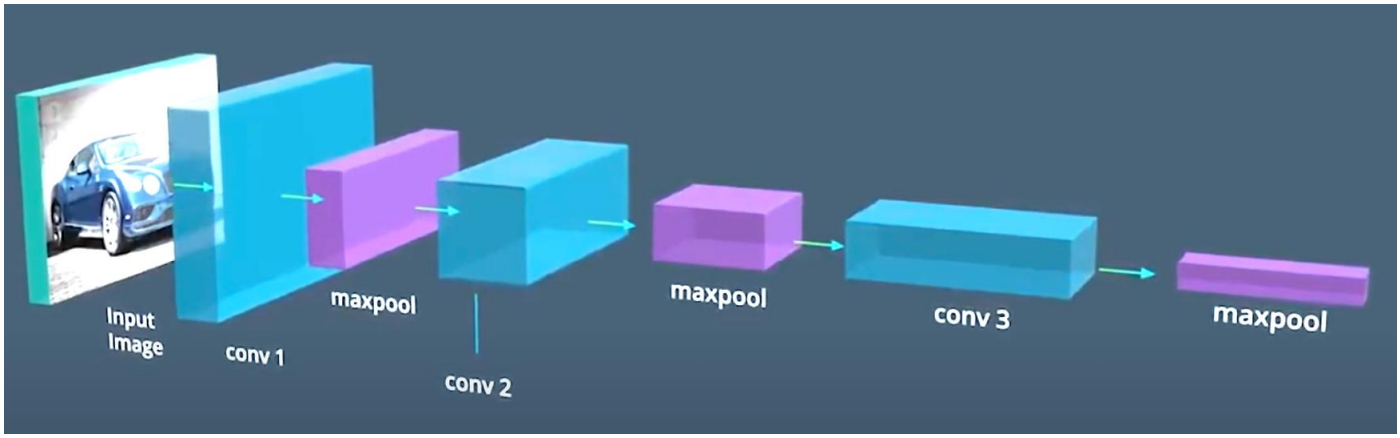
- Convolution is one run on the image input matrix to get a filtered/convoluted matrix. Like this many can be created in one layer itself.
- Increasing number of Convolutions in CNN is like upping neurons in a MLP.

Pooling:

In a typical CNN, many convolutions are applied in a single layer leading to numerous filtered/convoluted outputs aka feature maps. While many of these feature maps individually are

sparse with 0 in many cells. This calls for Pooling which is a way to average values from nearby cells thus scaling down the image. Pooling is always used in CNNs.

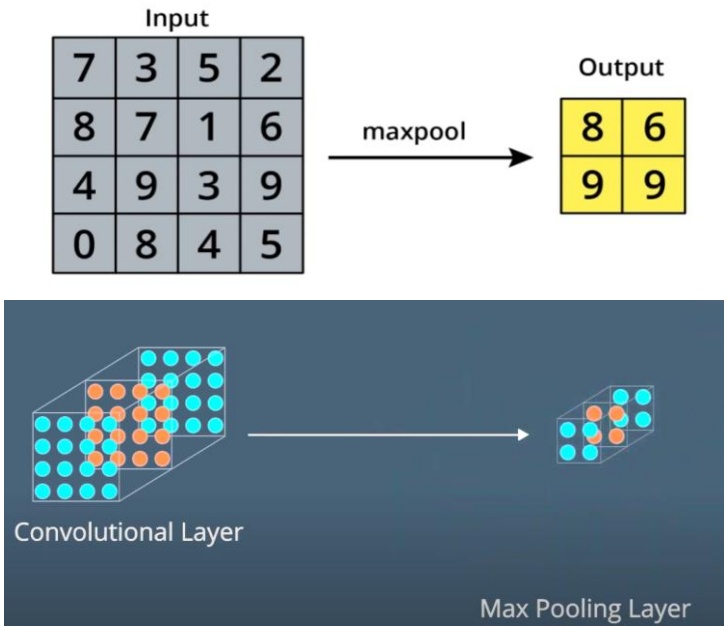
Pooling reduces the X Y scale of the picture but ends up gaining depth in the form of multiple feature maps. This is sensible in that the feature maps extracts separate features to understand the image while also losing the noise such as background pixels.



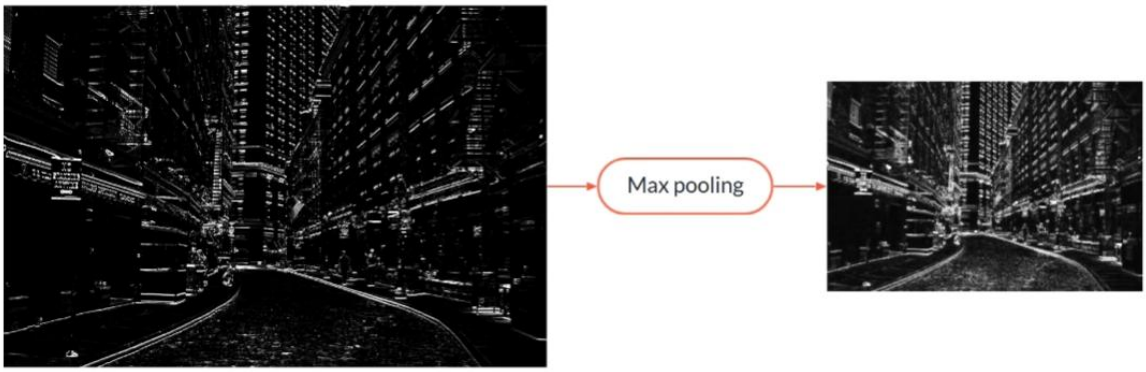
Note: Maxpool scales down image size but gains depth in form of multiple feature maps which is what will help the CNN to breakdown the image into different features and understand

Normal convolutions result in sparse matrices while pooling- averaging - like smoothing in Time series – the kernel and saving only the important values results in extracting edges and still identify edges

Max Pooling is when the max value of kernel is saved.



Example of Max Pooling



Max-Pooled image is smaller in size to store and lesser sparsity so making NN more efficient.

Important Note: All horizontal lines have been kept so output feature map contains same info as input

Max Pooling layer does:

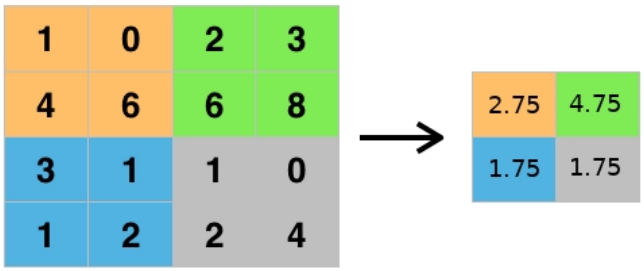
A pooling layer compresses the output of a convolutional layer while keeping most of the information.
Reduce the noise by selecting only max value
A Max Pooling layer allows the network to abstract concepts and gain approximate translation invariance.

Note:

MaxPooling **loses information** too aggressively. Use **stride = 2** in some **Conv layers** instead of pooling to counter.

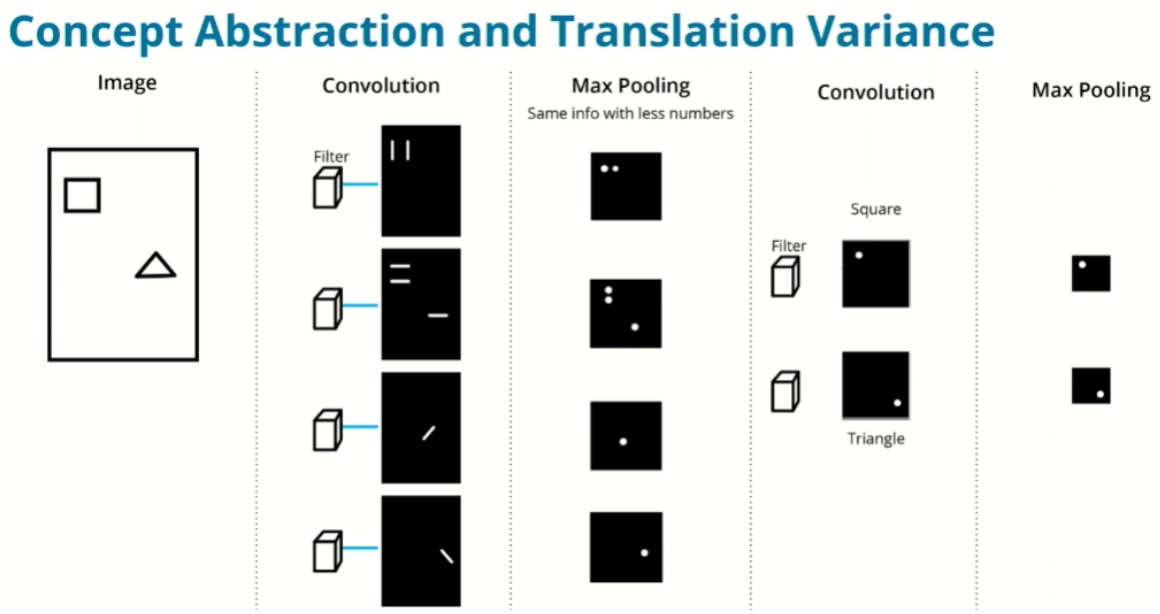
Average Pooling

This works similarly to the Max Pooling layer, but instead of taking the maximum for each window, we take the mean average of all the values in the window.



Average Pooling

Average Pooling is **not** typically used for image classification problems because Max Pooling is better at noticing the most important details about edges and other features in an image, but you may see average pooling used in applications for which *smoothing* an image is preferable. Sometimes, Average Pooling and Max Pooling are used together to extract both the maximum activation and the average activation.



A block consisting of a convolutional layer followed by a max pooling layer (and an activation function) is the typical building block of a CNN. By combining multiple such blocks, the network learns to extract more and more complex information from the image.

Sca
Translation Variance in the context of CNNs means that if the input image is shifted to left or right side of image, the feature maps will also shift by the same amount. This is not ideal for object recognition, where you want to recognize an object regardless of its position. Pooling is what helps make CNN **translation invariant.**

Moreover, combining multiple blocks allows the network to achieve **translation invariance (The same object can be anywhere on the image and the network will recognize it.)**, meaning it will be able to recognize the presence of an object wherever that object is moved within the image.
CNNs are invariant only to some operations like **translation**, but not for **rotations** or **distortions**.
Scale Invariance: Detect object despite its size – small or big
Rotation Invariance: Detect object despite its angle

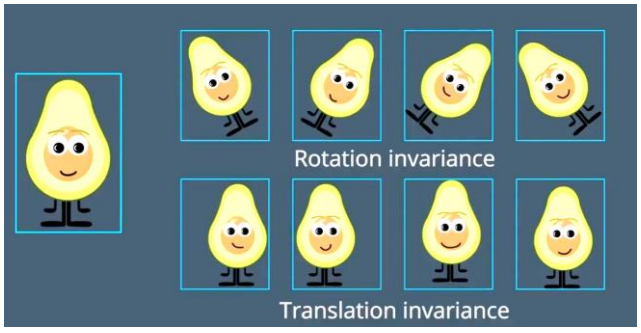


Image Filters aka Kernel

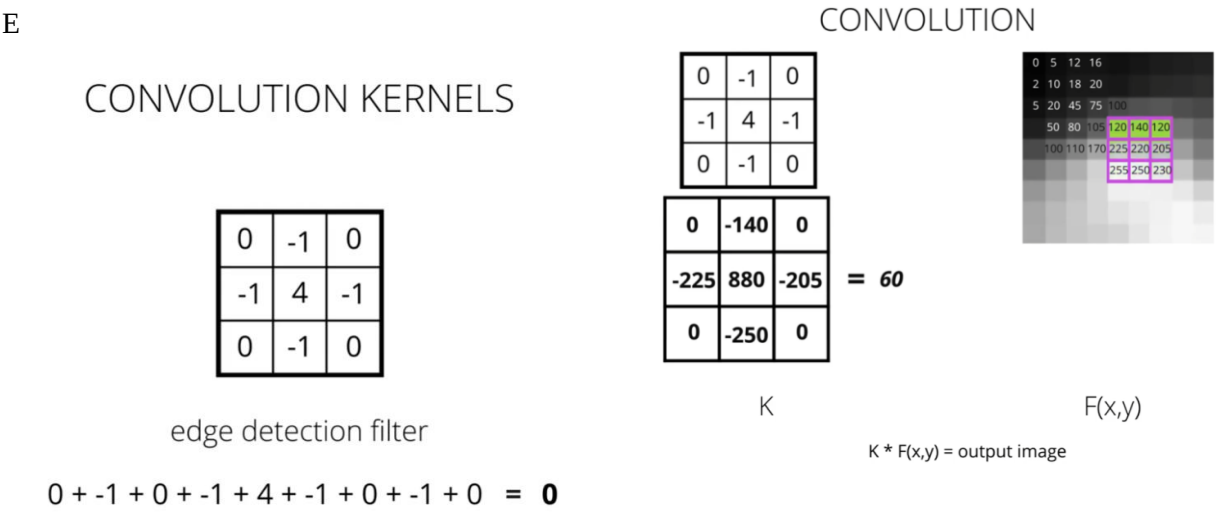
Image filters are a traditional concept in computer vision. They are small matrices that can be used to transform the input image in specific ways, for example, using them to bring out the edges alone of objects in the image.
An edge of an object is a place in an image where the intensity changes significantly.

To detect these changes in intensity within an image, you can create specific image filters that look at groups of pixels and react to alternating patterns of dark/light pixels. These filters produce an output that shows edges of objects and differing textures.

- Size of the kernel defines how big patterns in the image to be detected.
  - Increase size of kernel to detect larger patterns in the image. Eg: Face or tail or just eyes
- **<u>Kernel size range from 2X2 till 7X7 for large images; Maxpooling kernel size is usually 2X2</u>**

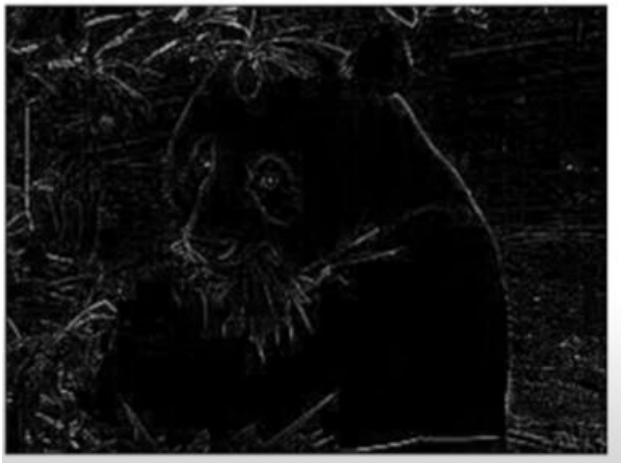
Convolution Kernel Types:

E



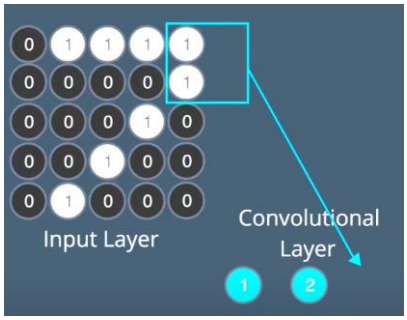
Note: A middle value and surrounding values such that sum is zero. Corner cells which are furthest, least influential are zeroed. This kernel is multiplied against pixel values of image to detect edges.

60 implies that a minor boundary(60 is quite close to 0 – Full zero means 100% boundary) was detected.
What is the logic of the value and polarity of the values?
- 4 in the middle is to multiply that value
- -1 around the 4 is to make sharp contrast (if it is a boundary then the negatives will contrast the 4 to bring it down to 0 which means 100% boundary value.
- 0 in the corner are furthest from point of contention which is at 4 and so zeroed to neglect

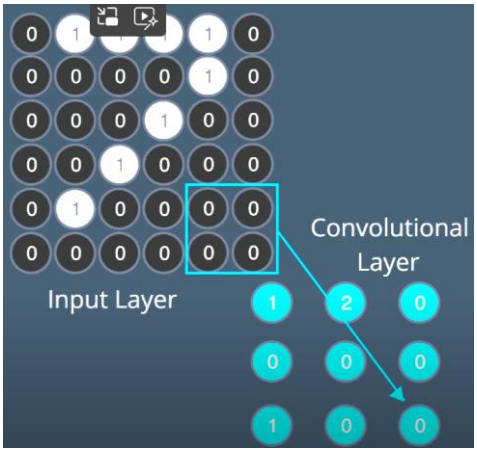
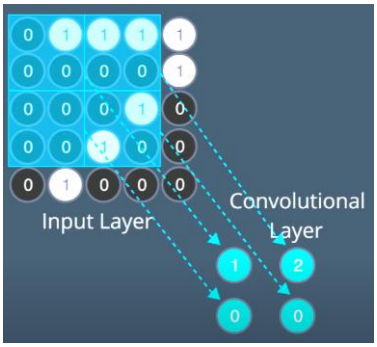


Result is an image like above where edges are detected

Corner Of Image Handling
Kernel convolution relies on centering a pixel and looking at its surrounding neighbors. So, what do you do if there are no surrounding pixels like on an image corner? Well, there are a number of ways to process the corners, which are listed below. It's most common to use padding, cropping, or extension.



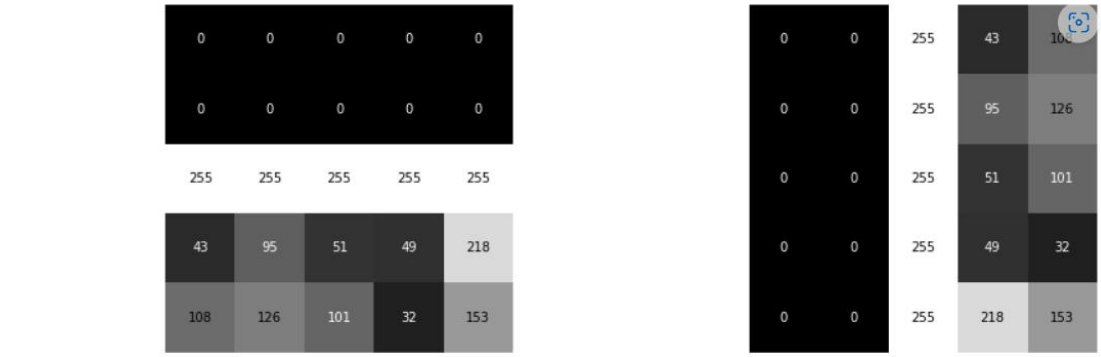
Padding - The image is padded with a border of 0's, black pixels. The zero-padding strategy is by far the most common
Cropping - Any pixel in the output image which would require values from beyond the corner/edge of picture is skipped. This method can result in the output image being smaller then the input image, with the edges having been cropped.

	
Note the extra column & row of zeroes added to ensure that kernel gets zero after the image ends - PADDING	Skipping to put any value for those that were just outside the kernels leads to Cropping. We in a way ignoring the edges of the image and thus cropping

Extension - The nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. Other edge pixels are extended in lines.

Vertical/ Horizontal Sobel Filters:

Edge detecting filters
Consider two simple pics



-1	-1	-1
-1	8	-1
-1	-1	-1

A

-1	0	1
-2	0	2
-1	0	1

B

0	-1	0
-2	6	-2
0	-1	0

C

-1	-2	-1
0	0	0
1	2	1

D

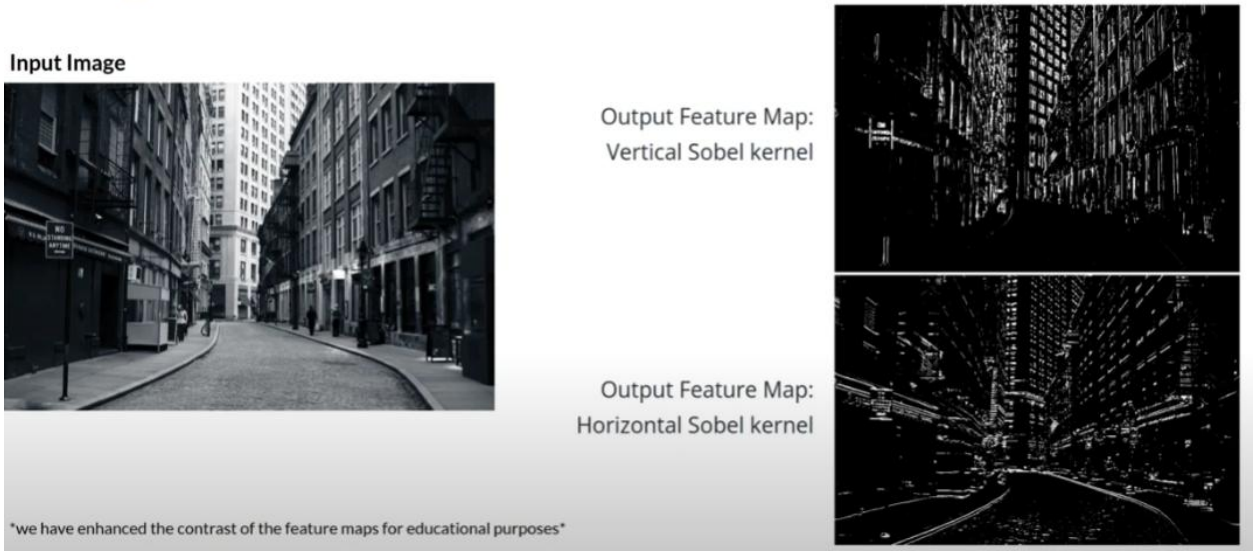
Answer: D is good for horizontal and B is good for vertical. They are called Sobel Filters

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

The Vertical and Horizontal Sobel filters

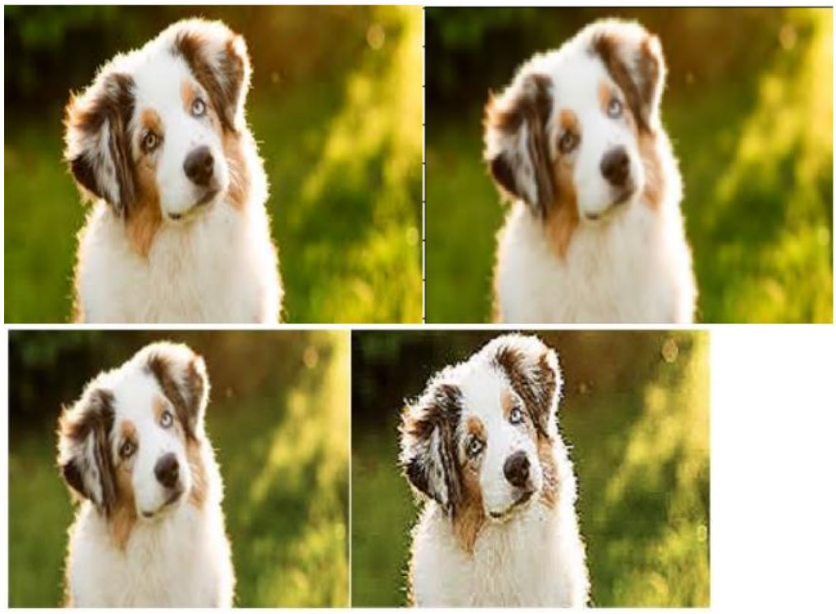
Ex: Applying Sobel on an image and the output feature maps



Filter to blur images

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

0	-1	0
-1	5	-1
0	-1	0



NOTE:

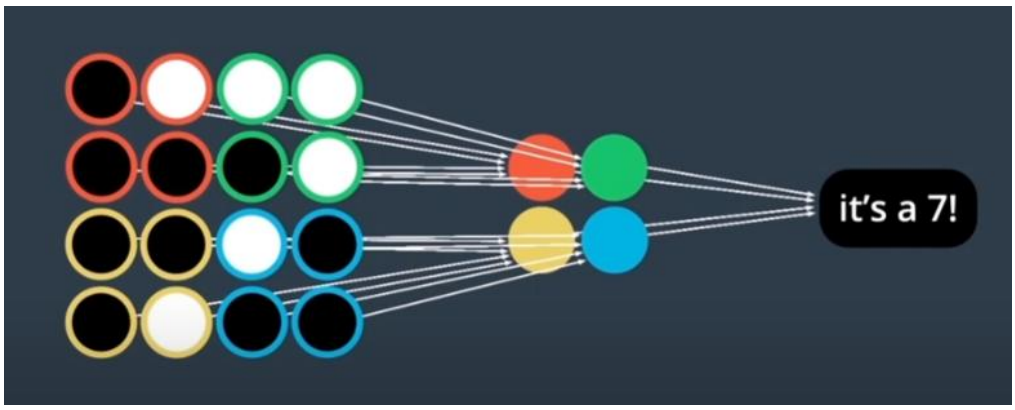
In CNNs, filters are not usually pre-defined/designed as implied above by the Data Scientist.
The value of each filter is learned (by algo itself) during the training process. CNN in fact uses many convolutions and creates many different feature maps which are processed thro the network and the error function helps it arrive at the best matrix to use in the end.
By being able to learn the values of different filters, CNNs can find more meaning from images that human designed filters which haven't been able to find.

By stacking layers of convolutions on top of each other, we can get more abstract and in-depth information from a CNN.

A second layer of convolution might be able to detect the shapes of eyes or the edges of a shoulder and so on. This also allows CNNs to perform hierarchical feature learning; which is how our brains are thought to identify objects.

Locally Connected NN

In CNNs, a neuron is connected only to a small number of relevant neurons in the previous layer and not all the neurons like in a typical NN.



CNNs are unique with **locally-connected layers**, i.e., layers where neurons are connected to only a limited pixels next to each other (instead of all the pixels like in fully-connected layers). Moreover, these neurons share their weights. The idea behind this weight-sharing is that the network is empowered to recognize the same pattern anywhere in the image.

Channels in CNN

Depth in Matrices like in Tensor (3D matrix) are channels.
In a CNN with more than one layer, the n_k filters in the first convolutional layer will operate on the input image with 1 or 3 channels (RGB) and generate n_k output feature maps. So in the case of an RGB image, 3 channels, the filters in the first convolutional layer will have a shape of kernel_size x kernel_size x 3. If we have 64 filters/channels/depth we will then have 64 output feature maps. Then, the second convolutional layer will operate on an input with 64 "channels" and therefore use filters that will be kernel_size x kernel_size x 64. Suppose we use 128 filters. Then the output of the second convolutional layer will have a depth of 128, so the filters of the third convolutional layer will be kernel_size x kernel_size x 128, and so on. For this reason, it is common to use the term "channels" also to indicate the feature maps of convolutional layers: a convolutional layer that takes feature maps with a depth of 64 and outputs 128 feature maps is said to have 64 channels as input and 128 as outputs.
- Channels are not compressed / averaged / normalized into one matrix and passed on but each filter is passed on as is. If a Conv layer creates 64 feature maps then that becomes the input to the next conv layer.

Number of Parameters in a Convolutional Layer

Let's see how we can compute the number of parameters in a convolutional layer,N.

- n: number of filters in the conv layer
- k: height and width of the onv kernel
- c: number of feature maps produced by the previous layer (or number of channels in input image)
- One layer to apply bias like the bias neuron function in a MLP NN

  Total number of parameters in the convolutional layer is: $N = n(c * k * k + 1)$.

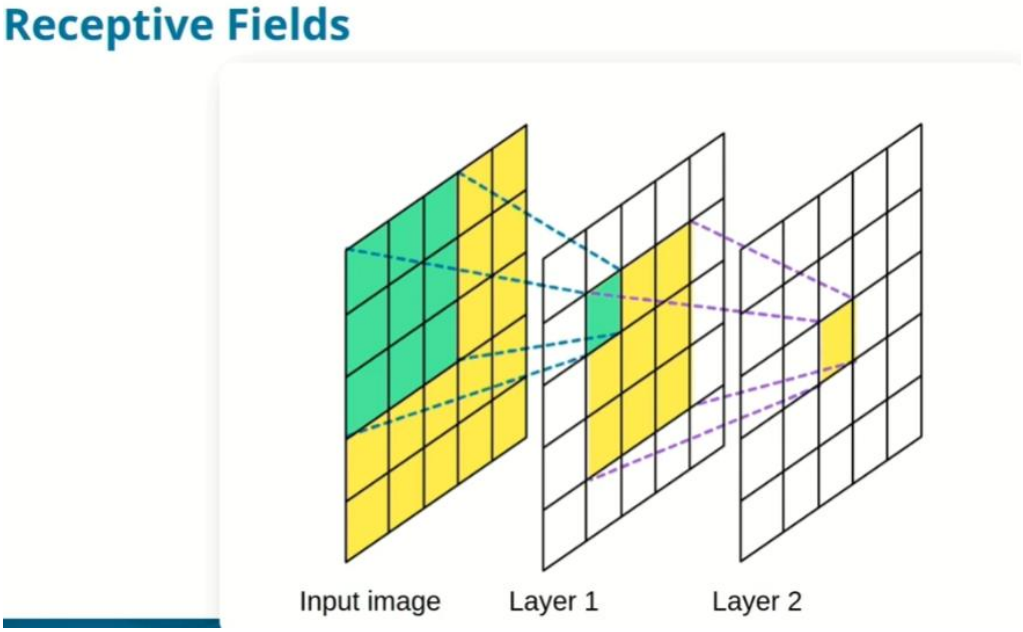
Formula for Convolutional Layers

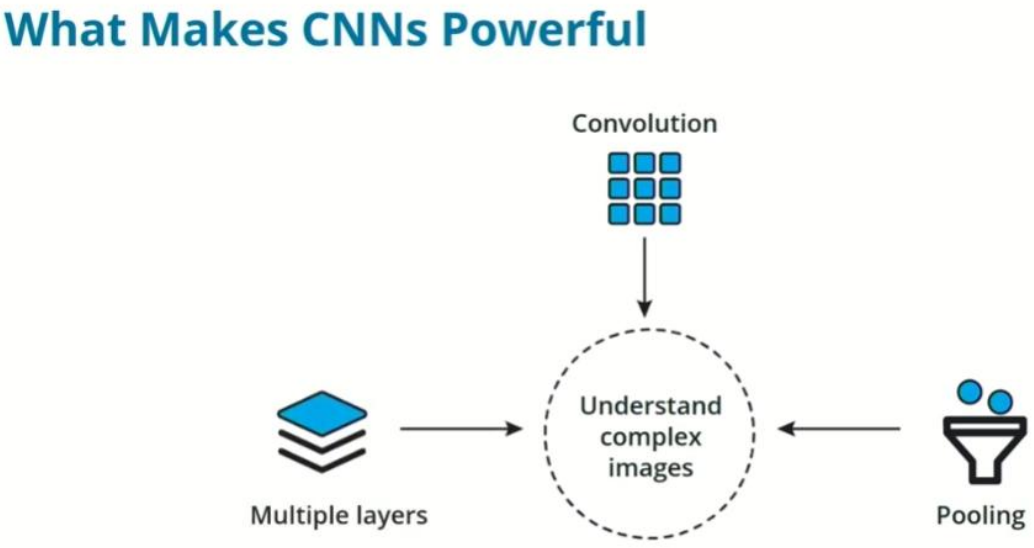
Needed when designing the CNN architecture.

$o = \left[\frac{i + 2p - k}{s}\right] + 1.$	• o – output size, I – input image size • k – size of kernel • s – stride • p - padding

Receptive Field:

Receptive field refers to the phenom that a single cell deep in the network has condensed info of a large part of the original information. Deeper the network, denser is the info each cell has of the original info.

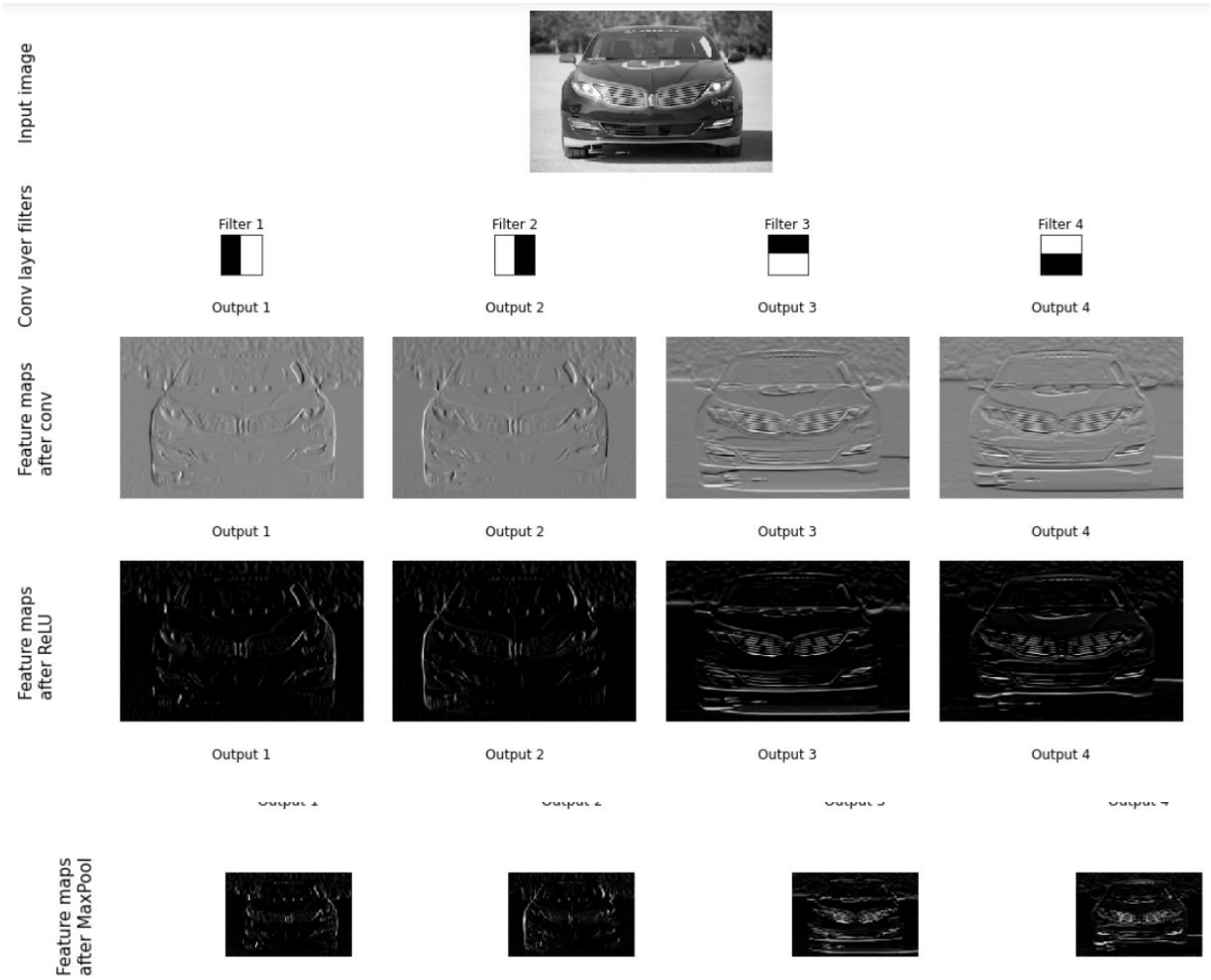




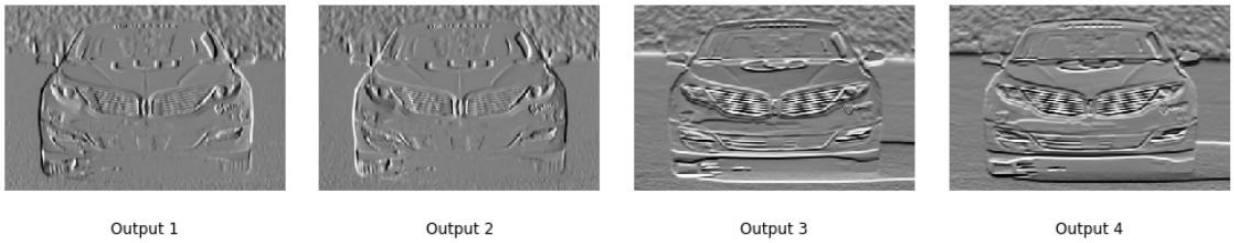
Convolution provides a mechanism to extract features – create multiple feature maps with different kernels

Pooling helps to take only the useful info and discard unuseful info

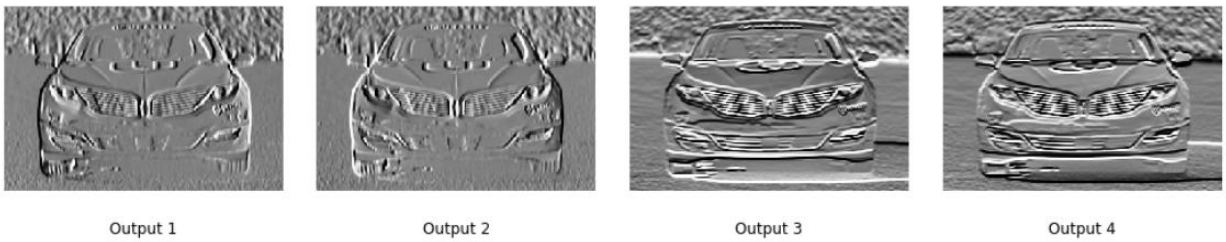
Multiple layers provides scalability to extract complex info from images



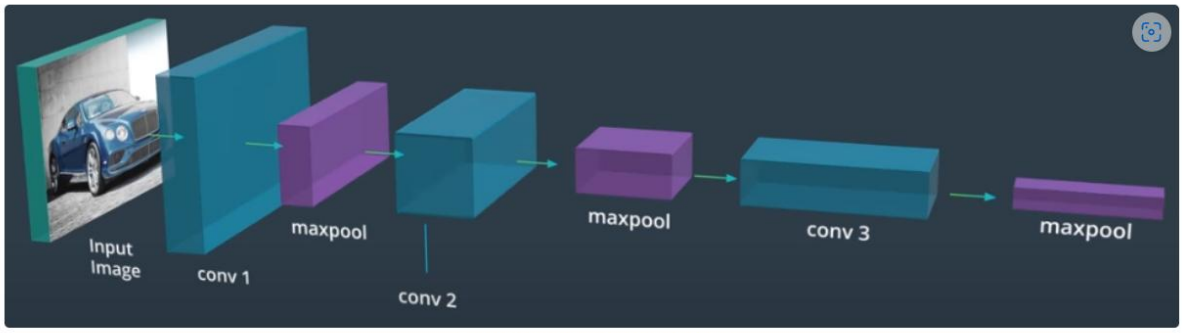
Sigmoid activation:



TanH activation:



CNN Structure:

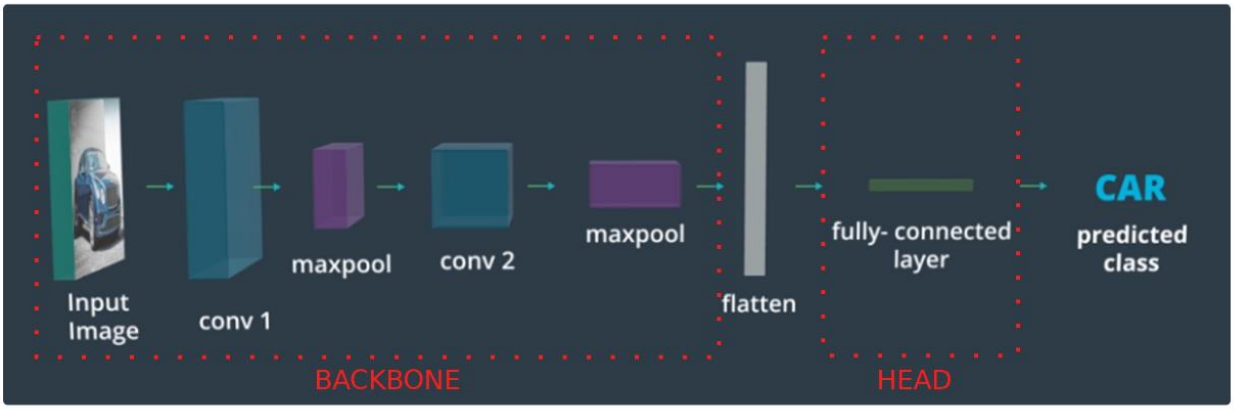


Structure of a Classical CNN

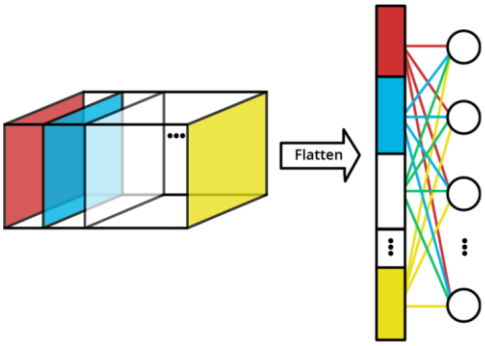
Note: The length of the cuboids above indicate the number of feature maps (no of output layers)

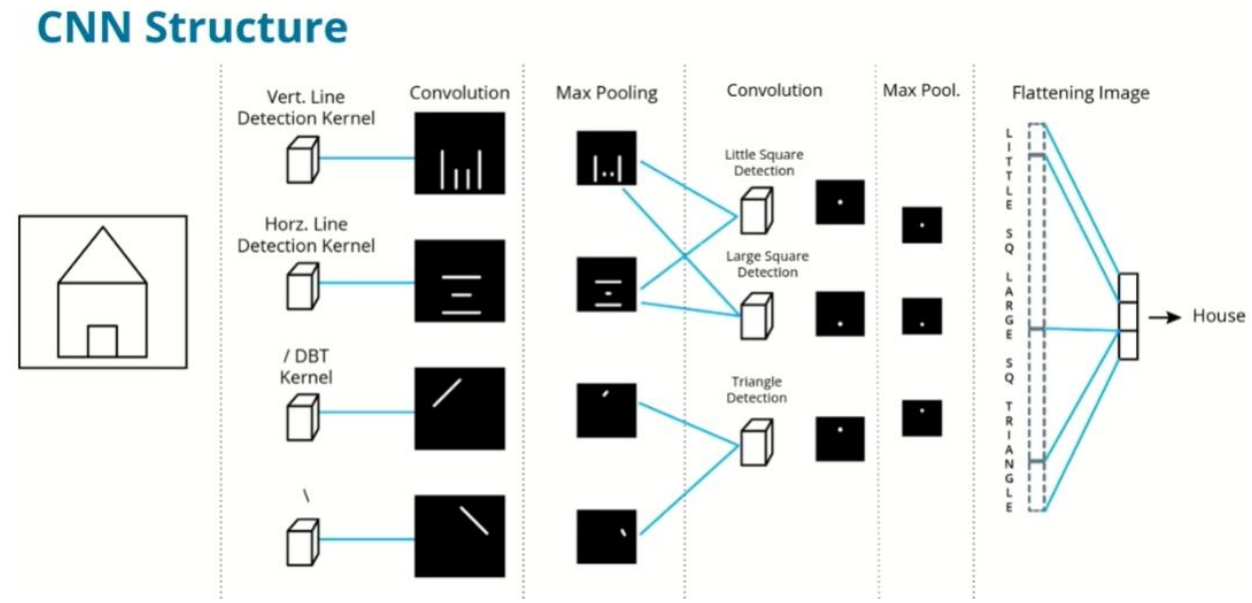
- Stacking different blocks of convolution followed by pooling is typical CNN.
- Typically the sizes of the feature maps shrink as you go deeper into the network, while the channel count (i.e., the number of feature maps and filters) increases, as above.
- As the signal goes deeper into the network, more and more details are dropped, and the content of the image is "abstracted." In other words, while the initial layers focus on the constituents of the objects (edges, textures, and so on), the deeper layers represent and recognize more abstract concepts such as shapes and entire objects.
- Typical Order
  - Conv
  - BatchNorm (More effective before activation)
  - RELU / Activation
  - MaxPooling

CNN (Feature Extraction) + MLP (Classification of Features)



CNN to MLP interface:





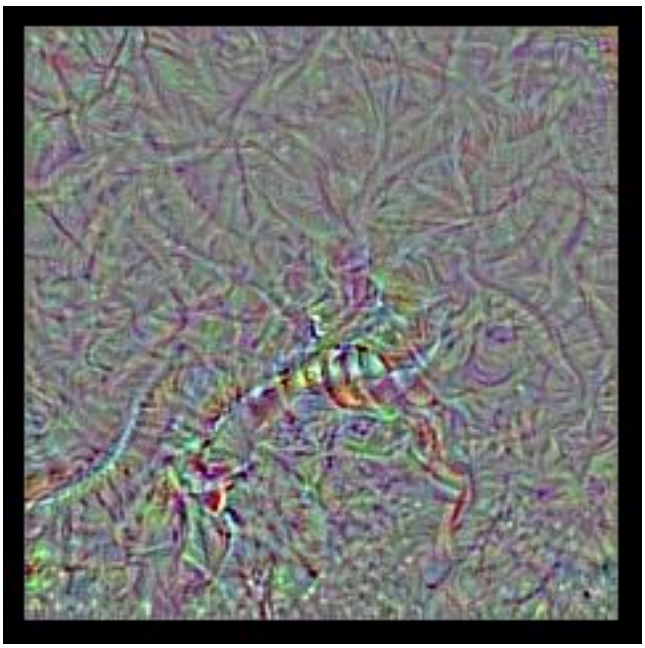
Demo: [2D convolutional network visualization]
[An Interactive Node-Link Visualization of Convolutional Neural Networks (adamharley.com)]

Visualizing CNNs:

1. CNN Explainer - https://poloclub.github.io/cnn-explainer/
   - Demo Video "CNN Explainer" - YouTube
2. Nice Demo with Commentary of CNN Visualizations - https://www.youtube.com/watch?v=AgkfIQ4IGaM&t=78s

Demo with VGG
How convolutional neural networks see the world

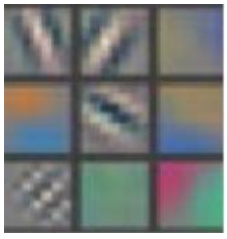
Printing photos that CNN sees and classifies

 Classified as Sea Snake which is correct!	 Classified as Magpie bird
 We can make out the sea snake so kudos!	 There seems to be no resemblance of a bird but yet the Convnet did a good job. Moral: Dont expect it to understand as humans do, besides don't think it understands the concept of a magpie

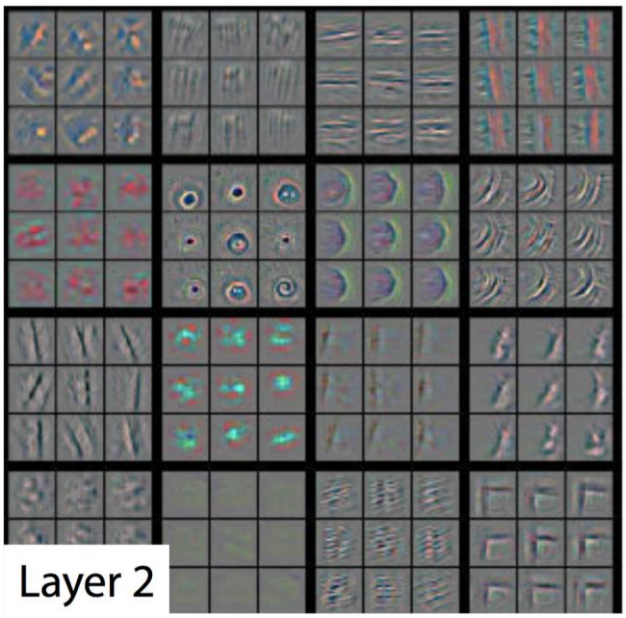
How does CNN do it? Or what does it really understands from a picture?

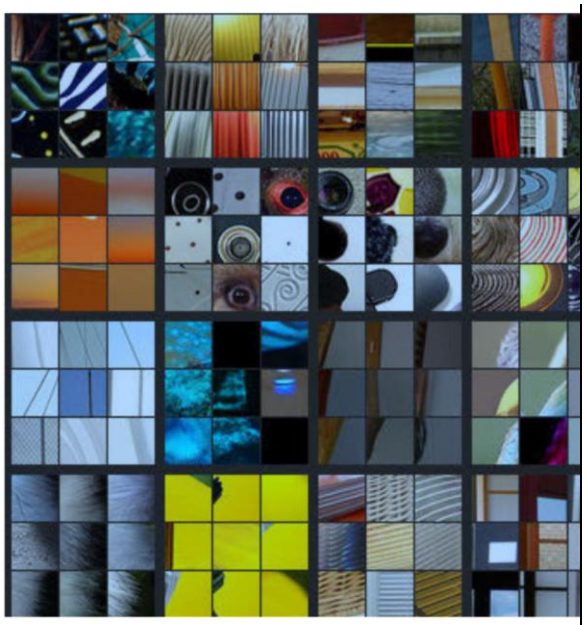
1. A CNN understands an image by breaking it down into different levels of features, where each level (built by conv layers) builds upon the previous one to detect increasingly complex patterns.
2. A CNN learns to connect specific combinations of detected features to different labels, then calculates probabilities of that image belonging to each possible label.

Matthew Zeiler and Rob Fergus' deep visualization toolbox demonstration on YouTube :

Layer 1	 Picking up simle lines, shapes

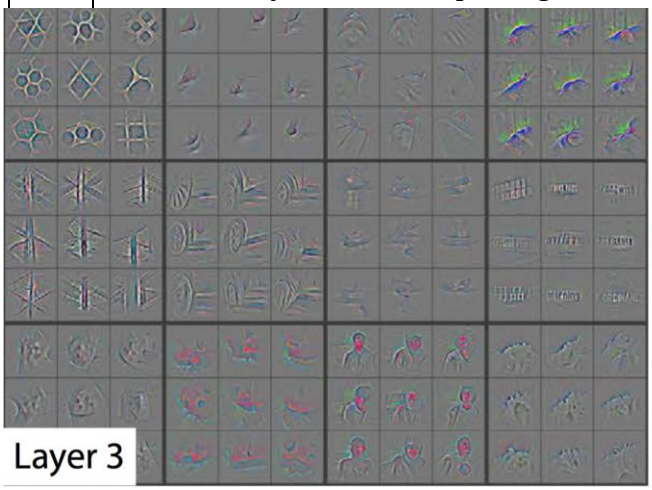
Layer 2

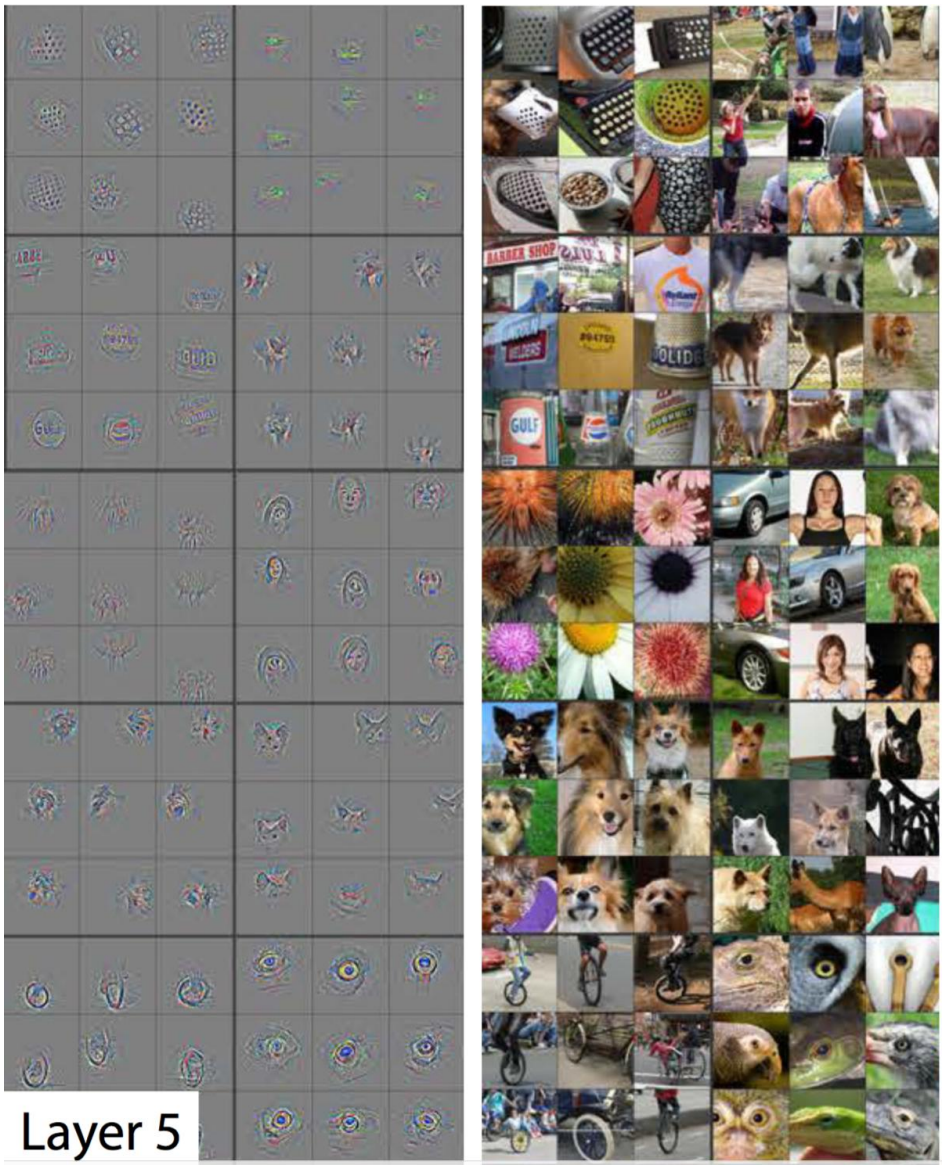




Left is what CNN sees after activation and right are the actual images.

- At Layer 2, CNN is picking more complex shapes like circles, rectangles and stripes



Layer 5 (Skipping 4 since it does similar to 3)	 picks out the highest order ideas that we care about for classification, like dog faces, bird faces, and bicycles

Input and Output of Conv Layer:

conv1 = nn.Conv2d(in_channels, out_channels, kernel_size)

1. in_channels - The number of input feature maps (also called channels). If this is the first layer, this is equivalent to the number of channels in the input image, i.e., 1 for grayscale images, or 3 for color images (RGB). Otherwise, it is equal to the output channels of the previous convolutional layer.
2. out_channels - The number of output feature maps (channels), i.e. the number of filtered "images" that will be produced by the layer. This corresponds to the unique convolutional kernels that will be applied to an input, because each kernel produces one feature map/channel. Determining this number is an important decision to make when designing CNNs, just like deciding on the number of neurons is an important decision for an MLP.

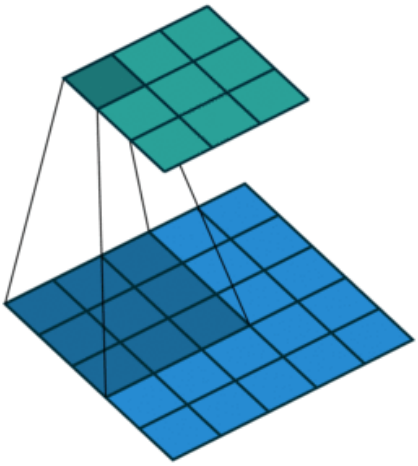
3. kernel_size - Number specifying both the height and width of the (square) convolutional kernel.
4. Formula to decide output image size:

$$o = \left[\frac{i + 2p - k}{s}\right] + 1$$

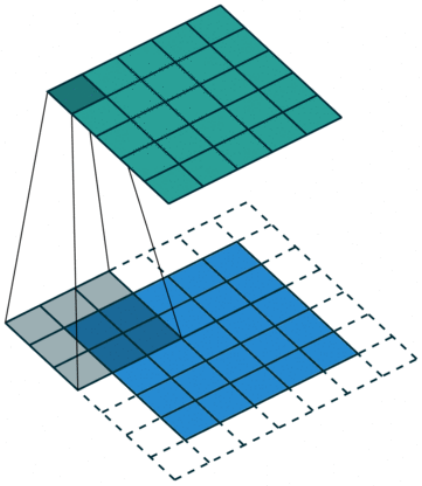
output size o,
the size of the input image i,
the size of the kernel k,
the stride s , and the padding p

Padding and Striding:

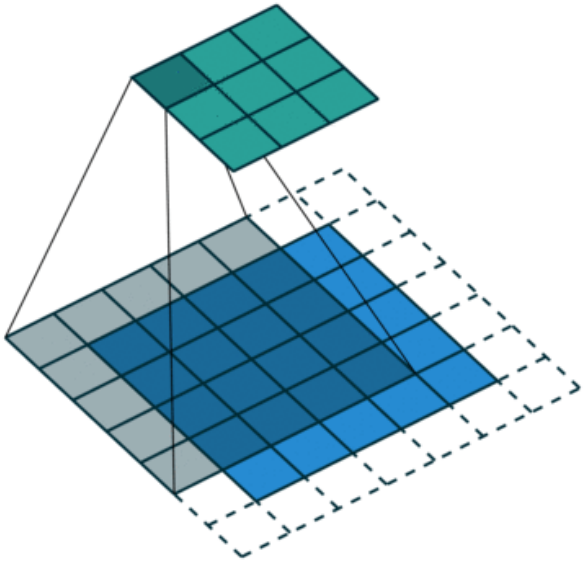
Striding – how many cells to move by kernel during convolution
Padding – When window goes out of bounds on input image, how to handle the missing cells, add zeroes? This is padding.



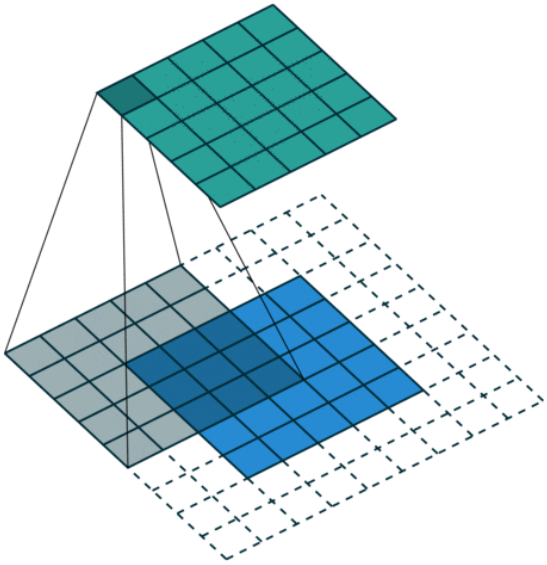
Padding Zero



Padding = 1



Padding = 1



Padding = 2

PyTorch Padding Types:

In PyTorch you can also use padding="same" or padding="valid"
The valid padding is equivalent to using a padding of 0.
The same padding instructs PyTorch to automatically compute the amount of padding necessary to give you an output feature map with the same shape as the input. Note this option only works if you are using a stride of 1.

padding_mode="reflect": padding pixels filled with copies of values in input image taken in opposite order, in a mirroring fashion

13	12	11	12	13	14	15	14	13
8	7	6	7	8	9	10	9	8
3	2	1	2	3	4	5	4	3
8	7	6	7	8	9	10	9	8
13	12	11	12	13	14	15	14	13
18	17	16	17	18	19	20	19	18
13	12	11	12	13	14	15	14	13
8	7	6	7	8	9	10	9	8

1	1	1	2	3	4	5	5	5
1	1	1	2	3	4	5	5	5
1	1	1	2	3	4	5	5	5
6	6	6	7	8	9	10	10	10
11	11	11	12	13	14	15	15	15
16	16	16	17	18	19	20	20	20
16	16	16	17	18	19	20	20	20
16	16	16	17	18	19	20	20	20

padding_mode="replicate": padding pixels filled with value of closest pixel in input image

padding_mode="circular": like reflect mode, but image is first flipped horizontally and vertically

14	15	11	12	13	14	15	11	12
19	20	16	17	18	19	20	16	17
4	5	1	2	3	4	5	1	2
9	10	6	7	8	9	10	6	7
14	15	11	12	13	14	15	11	12
19	20	16	17	18	19	20	16	17
4	5	1	2	3	4	5	1	2
9	10	6	7	8	9	10	6	7

The zero-padding strategy is by far the most common.

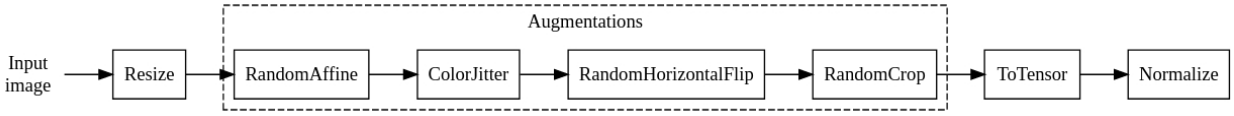
CNN Optimization Techniques:

Data Augmentation:

The basic idea of image augmentation is the following: if you want your network to be insensitive to changes such as rotation, translation, and dilation, you can use the same input image and rotate it, translate it, and scale it and ask the network not to change its prediction!

Data Augmentation Advantages: - Increases effective training dataset size - Improves model generalization - Reduces overfitting - Creates more robust feature representations

A typical training augmentation pipeline is represented in this diagram.



Code:

```
train_transforms = T.Compose(
    [
        # The size here depends on your application. Here let's use 256x256
        T.Resize(256),

        # Let's apply random affine transformations (rotation, translation, shear)
        # (don't overdo here!)
        T.RandomAffine(scale=(0.9, 1.1), translate=(0.1, 0.1), degrees=10),

        # Color modifications. Here I exaggerate to show the effect
        T.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),

        # Apply an horizontal flip with 50% probability (i.e., if you pass
        # 100 images through around half of them will undergo the flipping)
        T.RandomHorizontalFlip(0.5),

        # Finally take a 224x224 random part of the image
        T.RandomCrop(224, padding_mode="reflect", pad_if_needed=True),  # -


        T.ToTensor(),
        T.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),    ])
```

Results:

Transformation Pipelines for Validation and Test

Only Training sets are augmented in many ways like above while validation and test sets
Gets only resized, cropped – simple augmentations since this reflects the real world data more than exhaustive training approach above.
Note:

- The resize and crop should be the same as applied during training for best performance

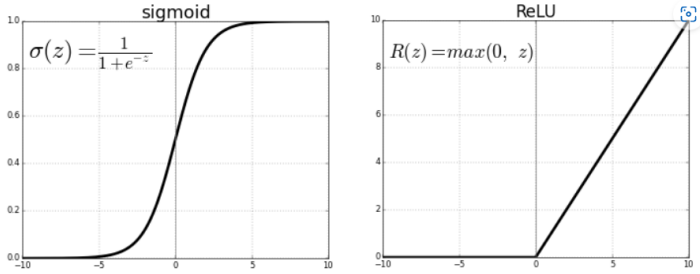
- The normalization should be the same between training and inference (validation and test)

AutoAugment Transforms

There is a special class of transforms defined in torchvision, referred to as **AutoAugment**. These classes implements augmentation policies that have been optimized in a data-driven way, by performing large-scale experiments on datasets such as ImageNet and testing many different recipes, to find the augmentation policy giving the best result. It is then proven that these policies provide good performances also on datasets different from what they were designed for.

Preprocessing in CNN:

Activation functions are performant when the inputs to it are close to zero. As the input gets further away from 0 the activation funcs become useless



Images' values are all over the place so must be normalized and the most common is standardization with mean and std and thus convert all values to -1 to 1.

Batch Normalization Pt.2:

Link to above notes: Ctrl+Click **Error! Bookmark not defined.**

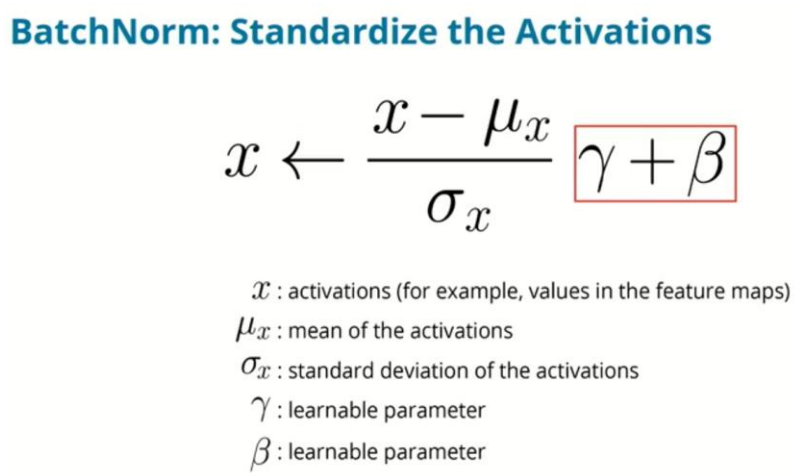
BatchNorm simply helps the working of Gradient Descent greatly by handling the possible variance that could occur in the data. Makes the solution more faster & robust and NN more efficient & easier.
Normalization helps ML models in all levels and BatchNorm is just done between layers in NN including CNN. BatchNorm is more important than any normalization done to input images before they are feed into the CNN because, the inputs are usually curated and largely less variant but the values between CNN layers with randomly chosen filters leads to a lot of variance between different runs of the algo. Thus BatchNorm is very important.

Covariate Shift – drastic change and influence in the final model due to variance in data.
BatchNorm mitigates Covariate shift.

BatchNorm During Gradient Descent:

Gamma and Beta terms are included which get influenced/controlled during Grad. Descent.

Gamma – changes width of the distribution
Beta – changes the center of the distribution like bias in NN

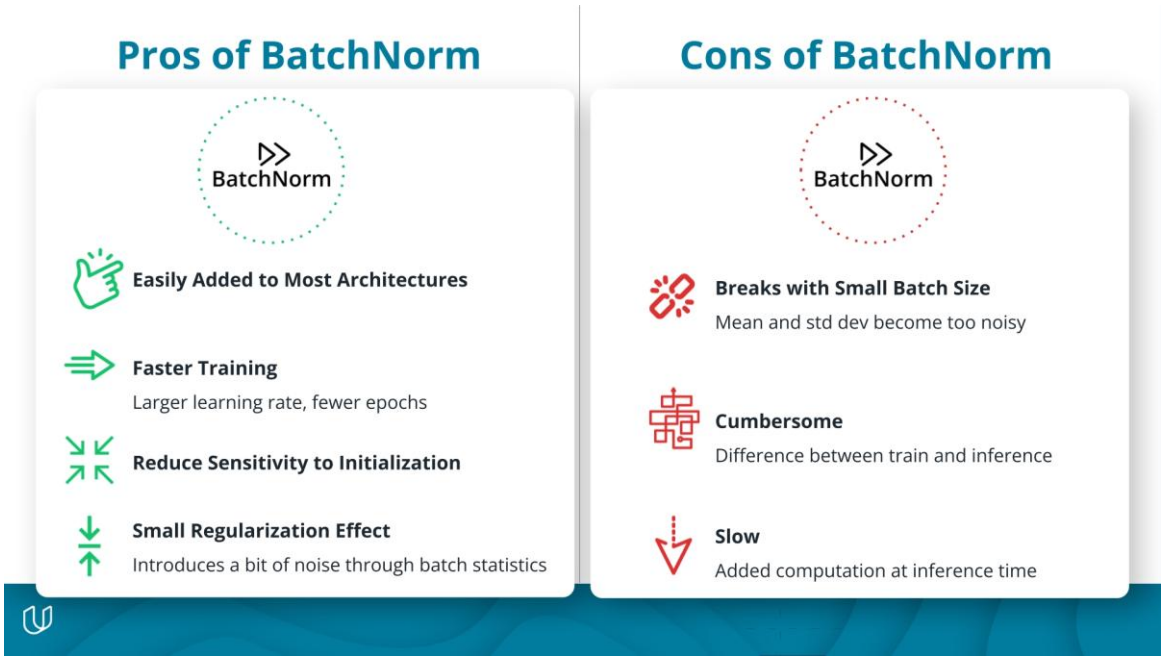


BatchNorm During Training and Inference:

During Training you have big batches of data to calc mean and variance which get used during BatchNorm to Normalize. But during usage of CNN in the real world, you might get only one image as input and so no mean or variance to do BatchNorm. This is where BatchNorm is different. It stores the mean and Variance from training data and uses the same during Inference.

During training, the BatchNorm layer also keeps a running average of the mean and the variance, to be used during inference.

During inference we don't have mini-batches. Therefore, the layer uses the mean and the variance computed during training (the running averages).



One key **BatchNorm advantage is that it allows us to have deeper Nns** against worries for vanishing/exploding gradient.

These advantages of using BatchNorm generally outweigh these disadvantages, so BatchNorm is widely used in almost all CNN implementations today.

BatchNorm and data augmentation help mitigate challenges in training deeper networks through two key mechanisms:

BatchNorm: - Reduces internal covariate shift - Stabilizes activation distributions - Allows higher learning rates - Helps gradients flow more smoothly through deeper networks	Data Augmentation: - Increases effective training dataset size - Improves model generalization - Reduces overfitting - Creates more robust feature representations

Together, they address gradient-related challenges:
- BatchNorm normalizes activations, preventing extreme gradient behaviors
- Data augmentation introduces variability that helps gradient descent explore better optimization landscapes

These techniques enable training deeper networks more effectively by making the optimization process more stable and less prone to vanishing/exploding gradient problems.

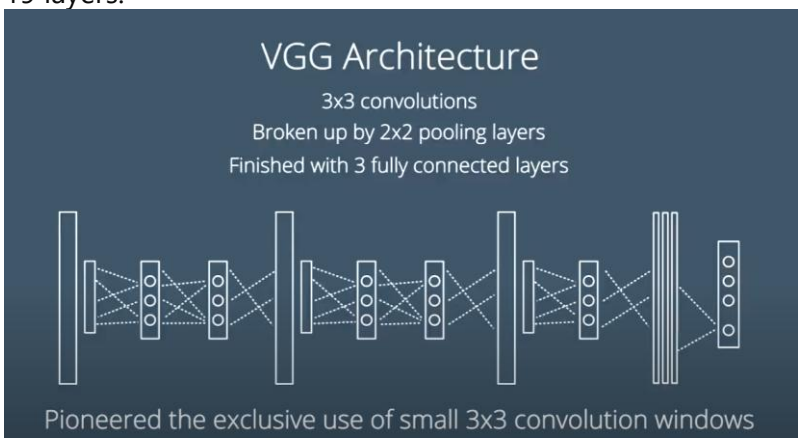
Key Side Topics in CNN:

Few Historically Popular CNN Architectures:

ImageNet Database: ImageNet is a large database containing over 10 million labeled images across 1,000 categories.

AlexNet
AlexNet was the first major breakthrough, utilizing GPUs for training. It introduced the ReLU activation function and dropout to prevent overfitting.

VGG
VGGNet, from the Visual Geometry Group at Oxford, featured a simple architecture with small 3x3 convolution windows, contrasting with AlexNet's larger 11x11 windows. It has versions with 16 and 19 layers.



VGG 16 – 16 layers

VGG 19 – 19 total layers

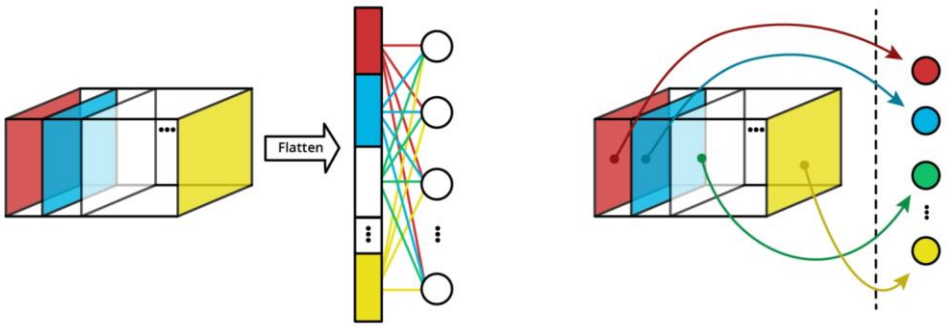
ResNet (2015): ResNet, created by Microsoft Research, achieved superhuman performance with a groundbreaking 152-layer architecture. It addressed the vanishing gradient problem by introducing skip connections, allowing gradients to bypass layers and improving training for deeper networks.

Residual Net usage of Skip connections innovative architecture allowed it to increase performance even if layers were kept on added – layer additions helped it get better and not worser hence popular.

Global Average Pooling (GAP) Layer

Input image size is a big controlling factor since the MLP at the head has a fixed input size. Images of bigger size would not fit since the Conv+Pooling layers just scaled it down but the final output size is very dependent on the input image size.
One innovative approach is to use GAP layer at the head of the MLP where the input is full average of each feature map at the end of the Conv+Pooling layers. This way, given any image size the no. of feature maps at the end wil always be the same and thus input to the MLP is always the same.

All said, GAP trained on a certain image size **will not respond well** to drastically different image sizes though this architecture is found to have found some usage in real world.

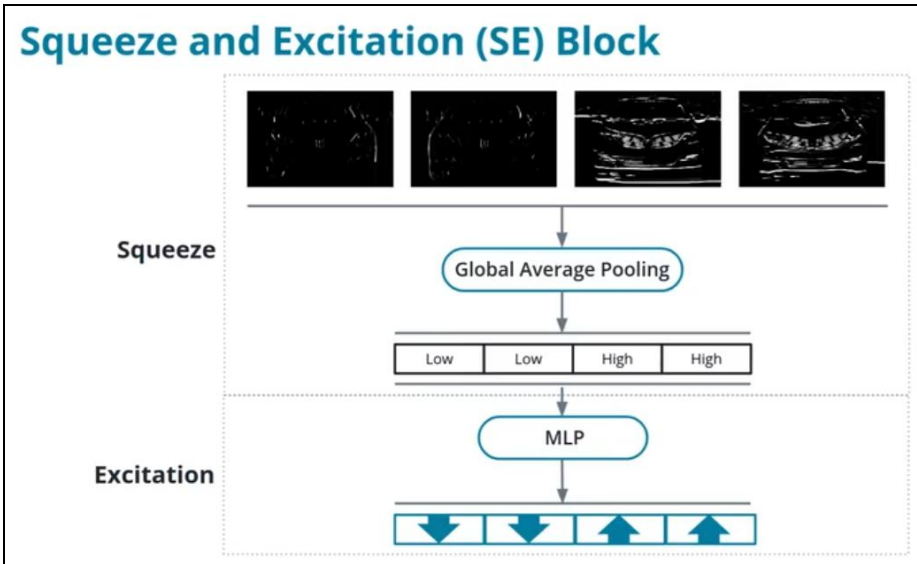


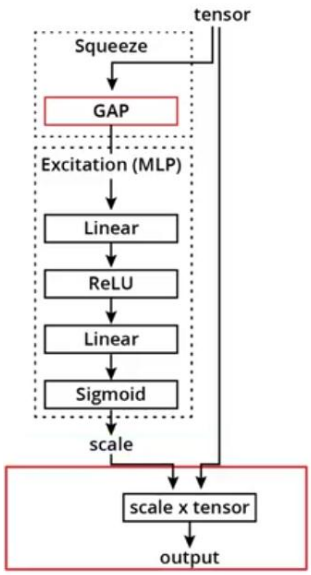
Channel Attention: Squeeze and Excitation – SE Block

Attention: It is a simple idea: the network should learn to boost some information that is helpful, and decrease the importance of information that is not useful

Channel Attention: Which channel to pay attention to? SE Block will tell us which.

Like in the 4 feature maps of the car, the last 2 have got more features than the first 2. So we boost the last two and relegate the last 2 using SE block. How? Squeeze and Excitation block





Computes a scale factor for each input feature map and then multiply each feature map by its scale factor, boosting important feature maps.

1. Squeeze Block: Finds the maps with more features using GAP layer
2. Excitation Block – A MLP block that amplifies feature maps with more features and suppresses those with less features
   SE Block above calculates the scale factor while Integ block below brings it back into the mainstream NN flow.

3. Integration block which joins the code back is what scales the right features maps up and other download
   - SE is a side block which helps classify the feature maps and not in the mainstream/flow of the NN. GAP removes a lot of useful data but needed only to help decide which needs attention.

Vision Transformers

Transformers are a family of neural networks originally developed for Natural Language Processing (NLP) applications. They are very good at modeling sequences, such as words in a sentence. They have been extended to deal with images by transforming images to sequences. In short, the image is divided in patches, the patches are transformed into embedded representations, and these representations are fed to a Transformer that treats them as a sequence.
Vision Transformers and self attention allows the network to learn how to pay attention to the relationship between different parts of an image.

VTs need a lot of data and slow to train which limits their popularity.

Intro: https://www.youtube.com/watch?v=WS1uVMGhlWQ

Transformers (NLP) Intro:
- **Introduction to Transformers - Part 1**
- **Introduction to Transformers - Part 2**
- **Introduction to Transformers - Part 3**

State of the Art Models for Computer Vision

1. CNNs – Majority market share (EfficientNet V2, ConvNet)
2. Vision Transformers – Rising star (ViT, Swin)
3. Hybrid – Niche use cases – CoatNet

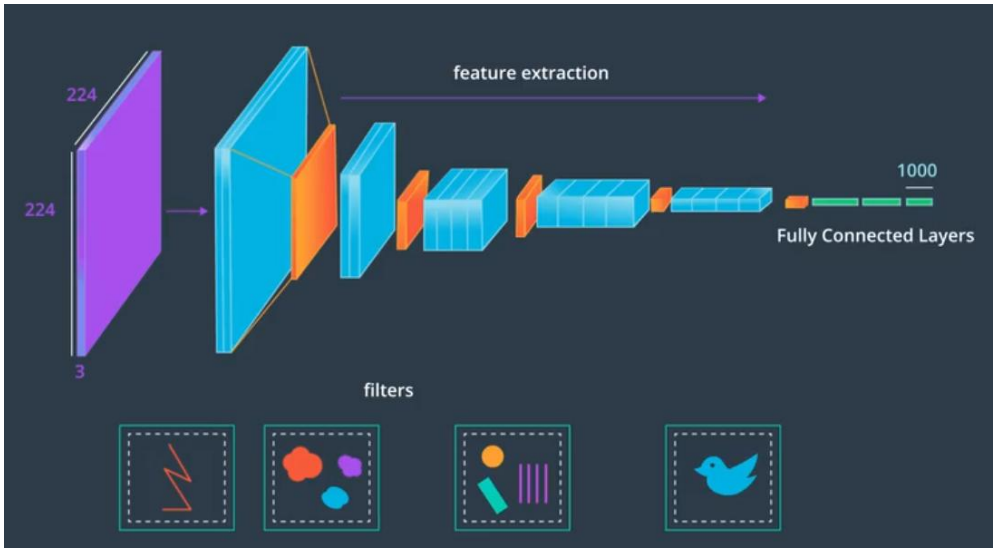
Transfer Learning

Transfer learning is a technique that allows you take a neural network that has been already trained of one of these very large datasets, and tweak it slightly to adapt it to a new dataset.

Essentially the **Transfer Learning** consists of taking a pre-trained model, freezing some of the initial layers and freeing or substituting some late layers, then training on our dataset.

CNN have two main parts:

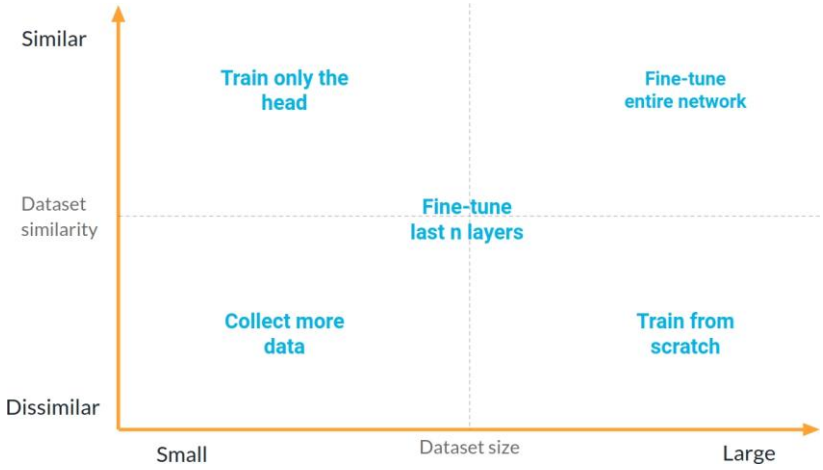
1. Feature Extraction : with the Conv + Pooling + Specialized layers
2. Classification : with the fully connected MLP to classify the features
    - This part alone could be re-trained for items quite similar to the OG train data and thus re-purpose a CNN and achieve Transfer Learning



How many layers we keep or modify slightly, and how many layers we change dramatically or even replace, depends on how similar our dataset is to the original dataset and on how much data we have.

Two parameters of the dataset affect Transfer Leraning:
Dataset Size and Dataset Similarity with the OG training data.

Typically, Small = 1,000-10,000 images Large = 100,000+ images



Based on these factors, here are the recommended approaches for above scenarios:

For Small, Similar Dataset:

- Train only the head (final layers)
- Keep pre-trained layers frozen
- Add new fully-connected layers
- Train for fewer epochs
- Use regularization if needed

For Large, Similar Dataset:

- First train the head for 1-2 epochs
- Then fine-tune the entire network
- Use conservative learning rate
- Optional: Use different learning rates for different layers

For Large, Different Dataset:

- Train from scratch
- Can still use proven architectures

For Small, Different Dataset:

- Most challenging scenario
- Either gather more data
- Or use semi-supervised learning

For Intermediate Cases:

- Fine-tune the head plus some later convolutional layers
- Keep earlier layers frozen
- Experiment to find optimal approach

TIMM: A Very Useful Library for Fine-Tuning

TIMM (py**T**orch **IM**age **M**odels) library is a valuable tool for transfer learning, offering:

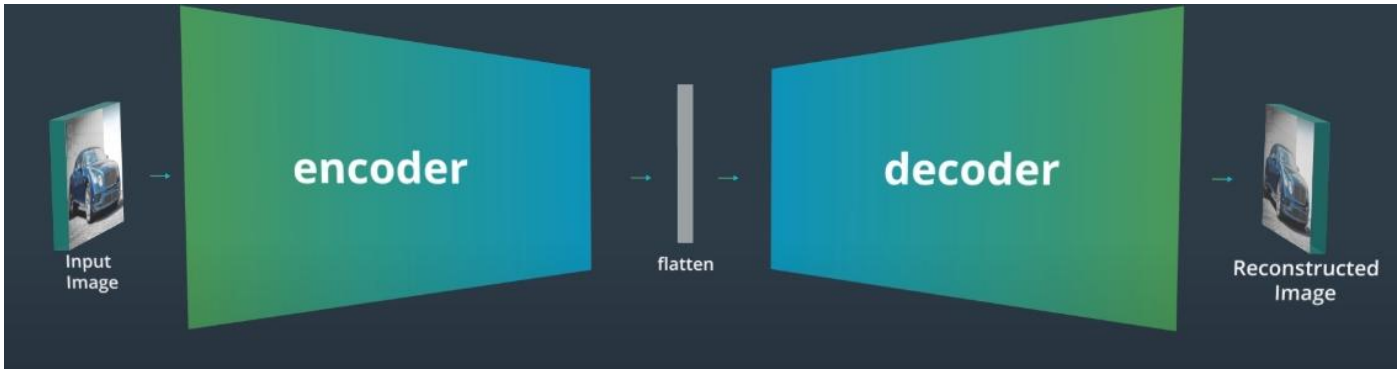
- Hundreds of pre-trained models
- Easy-to-use interface

- Automatic head building
- Regular updates with new architectures

AutoEncoders

Autoencoders are mirrored NN or CNN across a middle layer called Embedding layer.

- Has an encoder which is a typical NN or CNN with conv+pooling and other typical layers
- Decoder is usually the mirror opposite of the encoder
  - UnPooling upsamples/scales up in decoder rather than scaling down in encoder
- The middle layer is called embedding layer since it has a numeric representation of the image.
  - A 1d vector that encodes the information contained in the input image



The usual Conv+Pooling layers is mirrored on the other side of the fully connected MLP layers known as decoder. UnPooling upscales in decoder instead of pooling which downscales in encoder

Uses of Autoencoders

- Compress data
- **Denoise** data
- Find outliers (do **anomaly detection**) in a dataset
- **Do inpainting** (i.e., reconstruct missing areas of an image or a vector)

AutoEncoder's Loss Function

The autoencoder is concerned with encoding the input to a compressed representation, and then re-constructing the original image from the compressed representation. The signal to train the network comes from the differences between the input and the output of the autoencoder.

Mathematically, we can just MSE to give us the difference in the form of a numeric value.

Upscaling in Decoder

Transposed Convolution

A special type of convolution that can be used to intelligently upsample an image or a feature map.
A layer that intelligently upsample an image, by using a learnable convolutional kernel

Transposed Convolution can perform an upsampling of the input with learned weights. In particular, a Transposed Convolution with a 2 x 2 filter and a stride of 2 will double the size of the input image.

Whereas a Max Pooling operation with a 2 x 2 window and a stride of 2 reduces the input size by half, a Transposed Convolution with a 2 x 2 filter and a stride of 2 will double the input size.

Transposed Convolutions tend to produce checkerboard artifacts in the output of the networks

Therefore, nowadays many practitioners replace them with a nearest-neighbor upsampling operation followed by a convolution operation. The convolution makes the image produced by the nearest-neighbors smoother.

Nearest-Neighbor Scaling:
Copies nearest value and fills the empty cells.
An upsampling technique that copies the value from the nearest pixel

Convolutional Autoencoder

Linear Autoencoder uses linear layers while Convolutional are used in Conv.

The simplest autoencoder using CNNs can be constructed with a convolutional layer followed by Max Pooling, and then an unpooling operation (such as a Transposed Convolution) that brings the image back to its original size

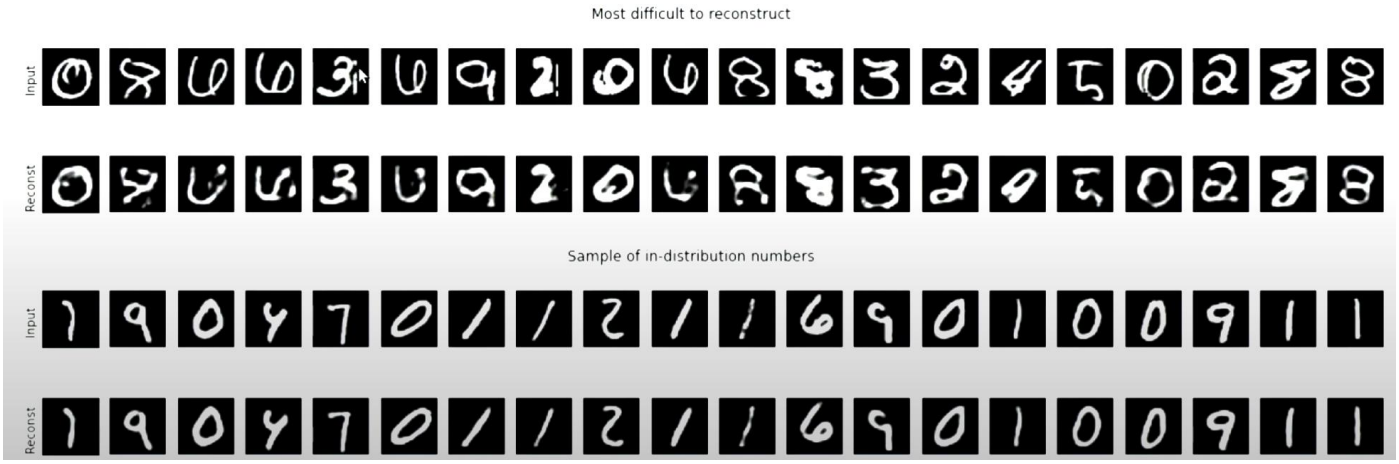
Autoencoder in Anomaly Detection

Autoencoders compress the visual information contained in images into a compact, latent representation (the embedding) that has a much lower dimensionality than the input image. By asking the decoder to reconstruct the input from this compact representation, we force the network to learn an embedding that stores meaningful information about the content of the image.

For example, in the solution I compressed 28 x 28 images (so 784 pixels) into a vector of only 32 elements, but I was still able to reconstruct most of the images very well.

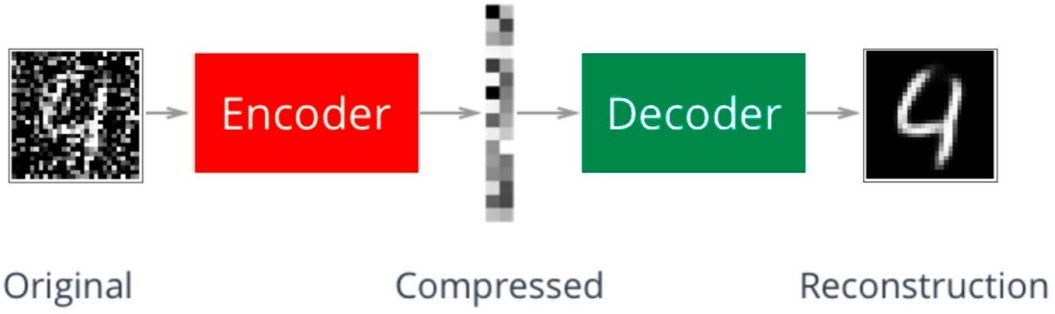
When applying it to a test set that the network has never seen, most images were reconstructed well, but some of them were not. This means that the compression that the network has learned on the training dataset works well for the vast majority of the examples in this new set, but not for these anomalous ones. These anomalies have characteristics that the network is not well equipped to reconstruct, and therefore the decoder cannot recreate them faithfully during decoding.

Through scoring each example by the loss, we are able to identify anomalies by simply taking the examples with the highest loss.



Denoising

Denoising the task of removing noise from an image by reconstructing a denoised image. This is a task that convolutional autoencoders are well-suited for. A denoising autoencoder is a normal autoencoder, but trained in a specific way.



How Do We Train a Denoising Autoencoder?

In order to train a denoising autoencoder we need to have access to the denoised version of the images. The easiest way to do this is to build a training dataset by taking clean images and adding

noise to them. Then we will feed the image with the added noise into the autoencoder, and ask it to reconstruct the denoised (original) version.

It is very important that we then compute the loss by comparing the input uncorrupted image (without noise) and the output of the network. DO NOT use the noisy version when computing the loss, otherwise your network will not learn!

Why Does it Work?

Let's consider an autoencoder trained on a noisy version of the MNIST dataset. During training, the autoencoder sees many examples of all the numbers. Each number has noisy pixels in different places. Hence, even though each number is corrupted by noise, the autoencoder can piece together a good representation for each number by learning different pieces from different examples. Here the convolutional structure helps a lot, because after a few layers the convolution smooths out a lot of the noise in a blurry but useful image of the number. This is also why generally you need to go quite deep with CNN autoencoders if you want to use them for denoising.

Autoencoders and Generative Models

how an autoencoder can be used to generate new images, although it's important to note that basic autoencoders aren't the best tools for image generation (that's why we have VAEs and GANs).

Here's how it works:

1. First, the autoencoder is trained on a dataset (let's say MNIST digits) where it learns to:
    - Compress images into embeddings (via encoder)
    - Reconstruct images from these embeddings (via decoder)
2. To generate new images, you can:
    - Take an existing embedding from a real image
    - Modify this embedding slightly (like averaging embeddings of two similar digits)
    - Feed this modified embedding through the decoder
    - The decoder will produce a new image

However, there's a significant limitation: the results are often unpredictable because:

- The embedding space is discontinuous
- A small change in the embedding might create a completely distorted image
- Some embeddings might produce images that don't look like real data at all

Think of it like this: if you have embeddings for a "3" and a "5", and you try to generate something halfway between them, you might get a weird hybrid that doesn't look like any real number, rather than a smooth transition between 3 and 5.

This is precisely why more sophisticated models like VAEs were developed - they create a continuous, well-structured embedding space that's much better suited for generating new, realistic images.

Computer Vision - Object Detection and Image Segmentation

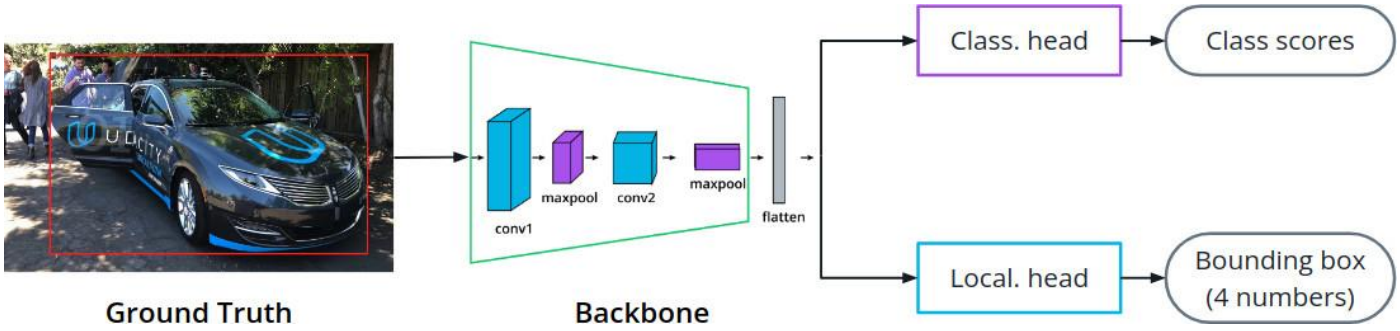
Terminologies:

Object localization: Assign a label to most prominent object, define a box around that object. Object localization is the task of assigning a label and determining the bounding box of an object of interest in an image.
Bounding box is a rectangular box that completely encloses the object, whose sides are parallel to the sides of the image.
Object detection: Assign a label and define a box for all objects in an image
Semantic segmentation: Determine the class of each pixel in the image. Separate the categories like people vs car
Instance segmentation: Determine the class of each pixel in the image distinguishing different instances of the same class. Separate each item, person a, person b, item a etc.

The different tasks of computer vision

Localization - Multi-Head Network :

In a typical CNN, the classification head has a parallel head for localization and bounding box.



Inputs:
3 inputs as ground truth:

1. the image,
2. the label of the object ("car") and
3. the bounding box for that object (4 numbers defining a bounding box)

Two outputs:

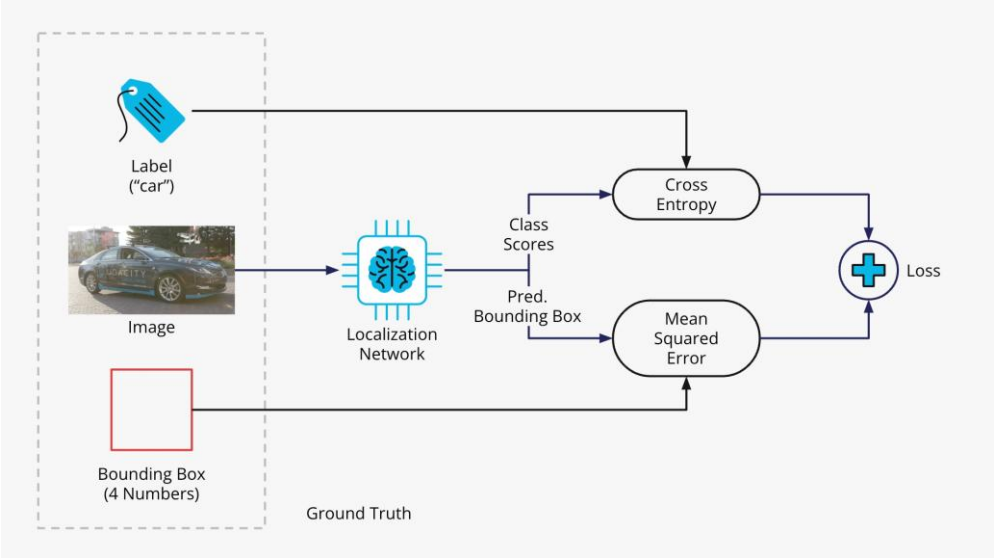
1.The class scores, that need to be compared with the input label

2.The predicted bounding box, that needs to be compared with the input bounding box.

Loss in Multi-Head Network

Cross Entropy for the label classification

MSE for the bounding box coordinates



The two losses are then summed to provide the total loss L.

Object Detection

The task of object detection consists of detecting and localizing *all* the instances of the objects of interest.

Sliding Window
A window smaller than image goes over each section of the image and identifies no of object. This approach has disadvantages, particularly, objects that are far and have a low aspect ratio. To detect near and far objects, we need separate windows thus leading to having windows of many sizes and aggregating sense of each.

This approach is old and could lead to huge number of windows blowing out compute power. Advanced methods like below exist but out of scope.

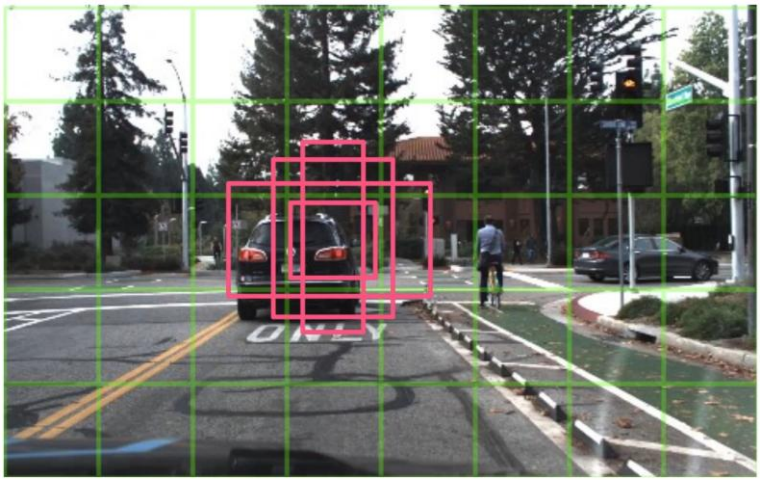
One Stage Detection
All images have a fized number and location of windows to detect images

Two-Stage Detection
Two algo work, one identifies the different regions in a picture and how many windows. The second algo detects the objects in each region.

RetinaNet network

Example of One Stage network.

RetinaNet uses anchors to detect objects at different locations in the image, with different scales and aspect ratios. Anchors are windows with different sizes and different aspect ratios, placed in the center of cells defined by a grid on the image.

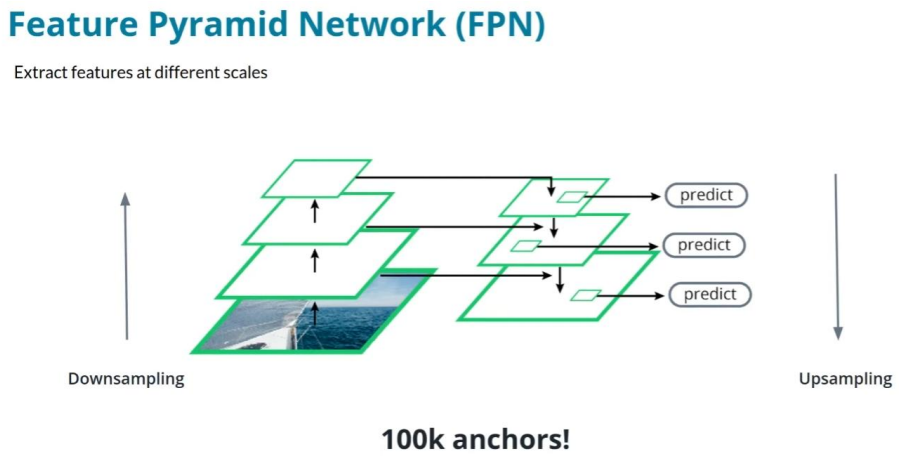


We divide the image with a regular grid. Then for each grid cell we consider a certain number of windows with different aspect ratios and different sizes. We then "anchor" the windows in the center of each cell. If we have 4 windows and 45 cells, then we have 180 anchors.

We run a localization network considering the content of each anchor. If the class scores for a particular class are high for an anchor, then we consider that object detected for that anchor, and we take the bounding box returned by the network as the localization of that object. This tends to return duplicated objects, so we post-process the output with an algorithm like Non Maximum Suppression.

Feature Pyramid Networks (FPNs)

RetinaNet uses a special backbone called Feature Pyramid Network.



Focal Loss

When using a lot of anchors on multiple feature maps, RetinaNet encounters a significant class balance problem: most of the tens of thousands of anchors used in a typical RetinaNet will not contain objects.

The **Focal Loss** adds a factor in front of the normal cross-entropy loss to dampen the loss due to examples that are already well-classified so that they do not dominate. This factor introduces a hyperparameter γ: the larger γ, the more the loss of well-classified examples is suppressed.

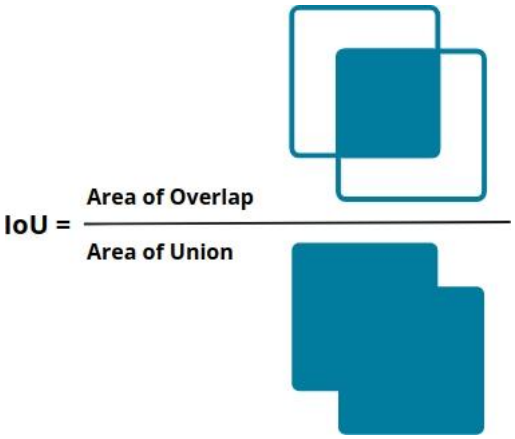
Object Detection Metrics

Mean Average Precision (mAP)
Mean Average Precision (mAP) conveys a measurement of precision averaged over the different object classes.
It is obtained by computing the Average Recall for each class of objects, as twice the integral of the Recall vs IoU curve, and then by averaging the Average Recall for each class.

Mean Average Recall (mAR)
It is obtained by computing the Average Precision (AP) for each class. The AP is computed by integrating an interpolation of the Precision-Recall curve. The mAP is the mean average of the AP over the classes.

Intersection over Union (IoU)

The **IoU** is a measure of how much two boxes (or other polygons) coincide. As the name suggests, it is the ratio between the area of the intersection, or overlap, and the area of the union of the two boxes or polygons:



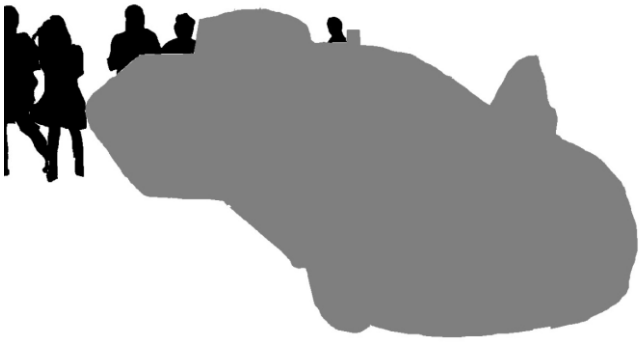
Intersection over Union (IoU) calculation for bounding boxes

IoU is a fundamental concept useful in many domains, and is a key metric for the evaluation of object detection algorithms.
Object detection is a success if the overlap is 50% or 0.5 when IoU threshold is 0.5.

Image Segmentation

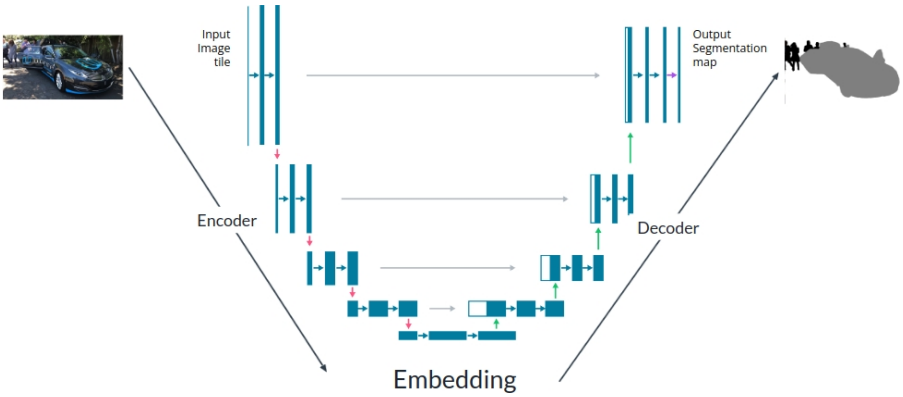
Semantic segmentation

The task of **semantic segmentation** consists of classifying each pixel of an image to determine to which class it belongs. For example, if we want to distinguish between people, cars, and background, the output looks like this:



In this image we have color-coded the people class as black, the car class as gray, and the background as white.

Unet Architecture:



The UNet is a specific architecture for semantic segmentation. It has the structure of a standard autoencoder, with an encoder that takes the input image and encodes it through a series of convolutional and pooling layers into a low-dimensional representation.

Dice loss: A useful measure of loss for semantic segmentation derived from the F1 score, which is the geometric mean of precision and recall. The Dice loss tends to balance precision and recall at the pixel level.

RNN

RNNs (Recurrent Neural Networks) are neural networks designed to process sequential data by using feedback loops, allowing information to persist across time steps. They are ideal for tasks like time series prediction, text, and speech processing.

Imagine you're reading a book. When you reach page 50, your understanding of the story doesn't just come from that single page - it's influenced by everything you've read in the previous 49 pages. This is exactly what makes RNNs special: they have a form of memory that allows them to consider previous information when processing new input.

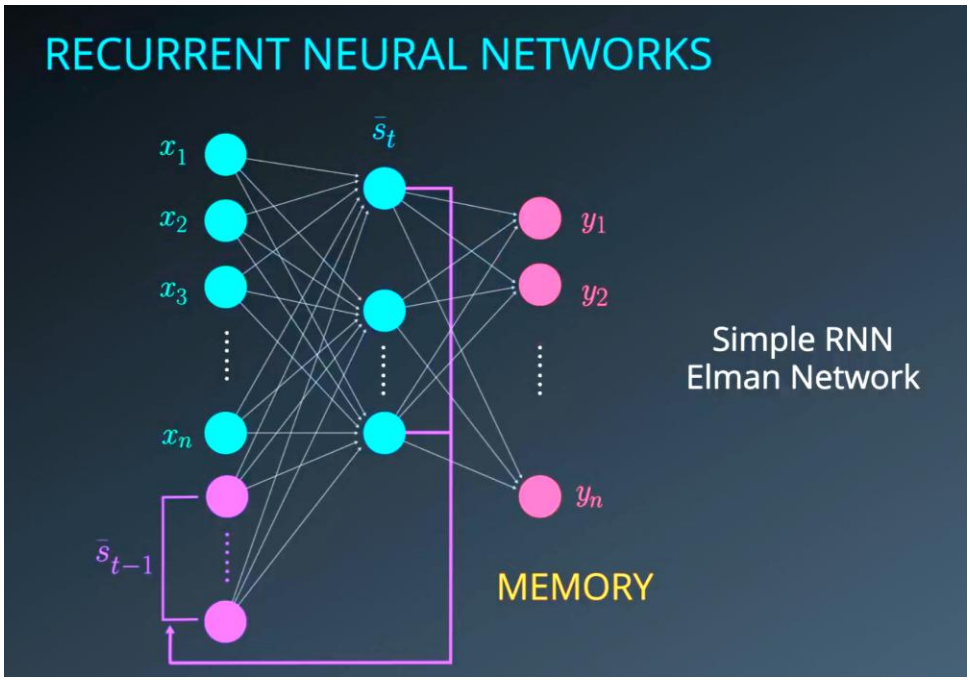
Traditional neural networks process each input independently, like trying to understand a story by reading random pages in isolation. But RNNs maintain an internal state (like your memory of previous pages) that gets updated as new information comes in. They do this by having connections that loop back on themselves - hence the term "recurrent."

States: RNN basically have a memory storing previous steps information in different ways and they are called States.
“State” – Comes from State Machine Concept whose concepts are used here in RNN.

Here's how an RNN processes a sequence:

1. It takes the current input (let's say a word in a sentence)
2. Combines it with its internal state (information from previous words)
3. Produces an output
4. Updates its internal state for the next word

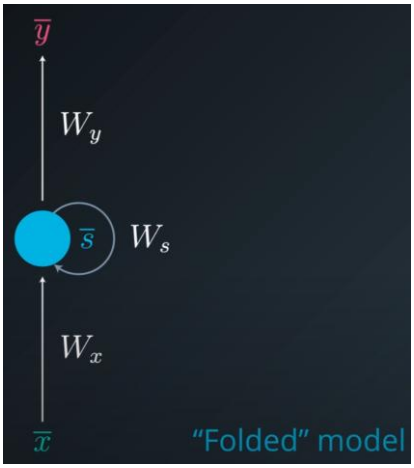
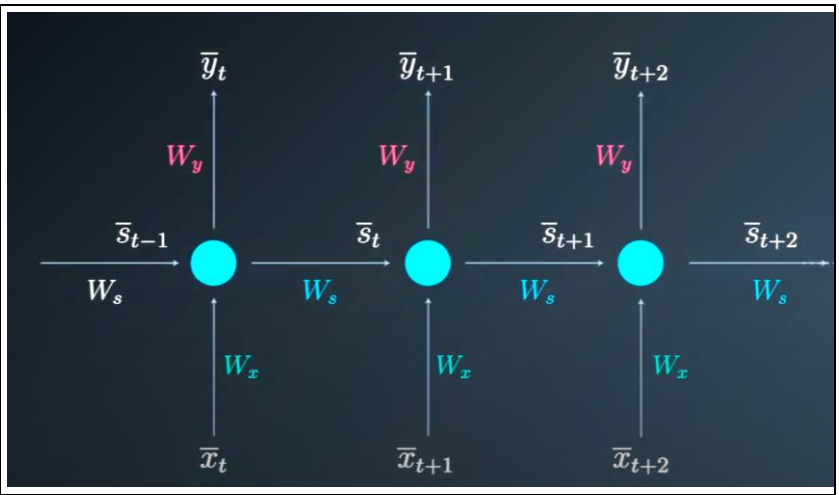
However, basic RNNs face a significant challenge: they struggle with long-term dependencies. Think about trying to complete the sentence: "I grew up in France... I speak fluent ___." Even though the relevant information (France) was mentioned much earlier, it's crucial for predicting "French." Basic RNNs often "forget" such long-range connections due to what's called the vanishing gradient problem - essentially, the influence of earlier information fades away too quickly during training.



RNN have a series of neurons that loops back an output again during the next run thus adding information from the previous step to the current step. These Additional Neurons (pink) are called as **Memory**.

These Memory neurons remember not just previous step but gets accumulated so the values were affected by runs before the previous one as well.

FOLDED and UNFOLDED Model

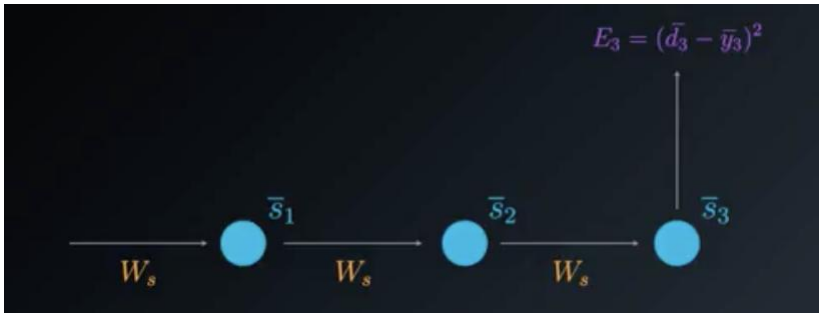
	
RNN each run is showed **<u>Bottom to Top</u>** is the flow of information.	Left to Right is Time, thus the right most is current, middle is previous and first on left is the 2 steps back. This visually shows that y_{t+2} or current step is influenced by the state vectors from previous steps cumulatively.

BackPropagation Through Time (BPTT):

BPTT is the bp process in RNN with recurrent architecture. It back propagates the error to the weight matrix Ws as well. Ws

BPTT is similar to how backpropagation in a normal NN uses Chain rule to calculate the weights for every preceding layer and not just the last layer. Chain Rule helps to include the weights of the initial layer weights to the impact it has on the Error. In same way the weights of each state (previous run) are included in the gradient estimation formula.

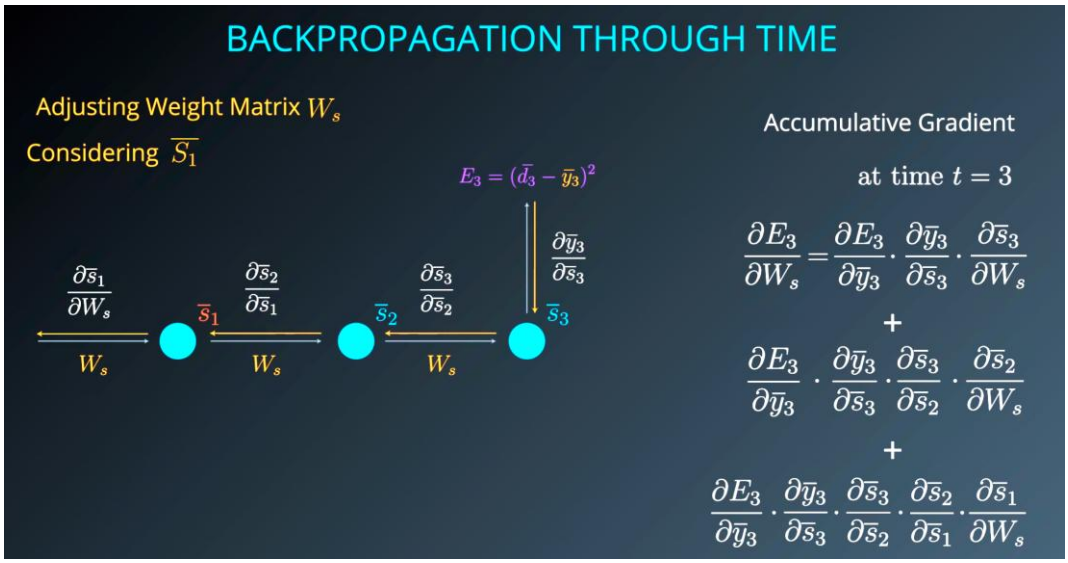
State Weight Matrix Ws:



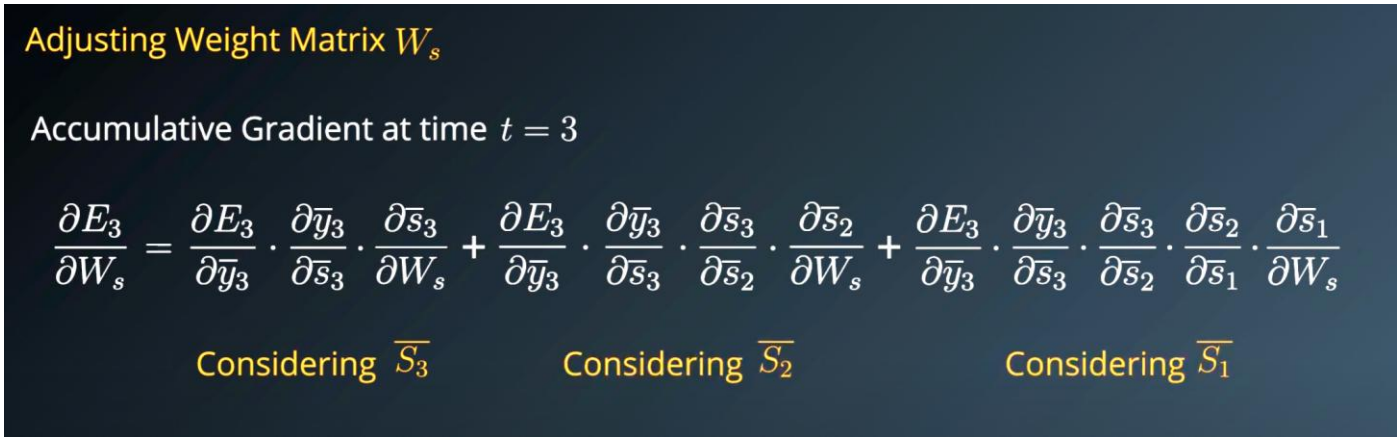
Let’s first consider a simplified view of the relevant elements in the RNN architecture diagram.

When calculating the partial derivative of the Loss Function for Ws, we need to consider all of the states contributing to the output. In the case of this example, it will be states s3 which depends on its predecessor s2 which depends on its predecessor s1, the first state.

In BPTT, we will consider every gradient stemming from each state, accumulating all of these contributions.

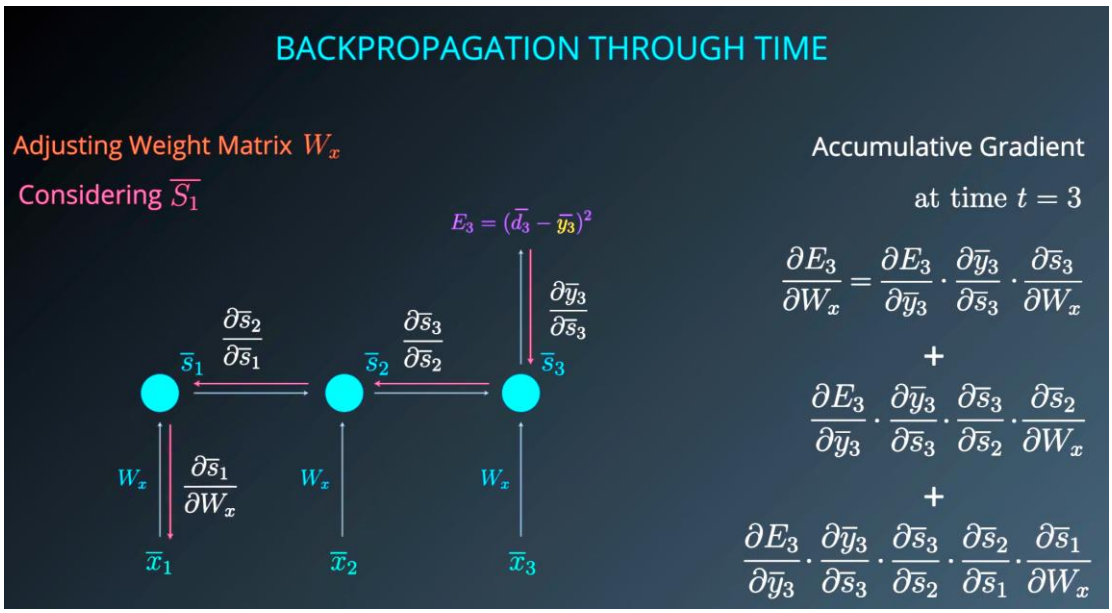


At timestep t=3, the contribution to the gradient stemming from $s3$,$s2$ and $s1$ is the following :



Adjusting / Updating Wx

Wx of each of the different states also affect the gradient of the Wx and so using Chain Rule we include all the involved weights and factors going back to the start s1.



How many time steps?
Studies have shown 8-10 to be viable and any more leads to vanishing gradient problem.

Is Exploding Gradient not a prob then?
Gradient Clipping technique simply handles this well.

Frequency of the weight updates?
In RNNs, it is found that waiting to update the weights after a few batches to be more efficient than updating every single batch. When training RNNs using BPTT, we can choose to use mini-batches, where we update the weights in batches periodically (as opposed to once every inputs sample). We calculate the gradient for each step but do not update the weights immediately. Instead, we update the weights once every fixed number of steps. This helps reduce the complexity of the training process and helps remove noise from the weight updates.

RNN and NLP

RNN have a good application in NLP so Udacity provided a series of NLP101 which I am not repeating here.
Topics like Stopword removal, stemming, lemmatization, word embeddding etc.

LSTM:

RNN have a major fault when it comes to real world applications which is the vanishing gradient problem and its limitations on the number of timesteps.

This is where LSTMs (Long Short-Term Memory) and GRUs (Gated Recurrent Units) come in. They're like upgraded versions of RNNs that solve this memory problem. Let me explain each:

LSTM networks are like RNNs with a sophisticated memory management system. They have three "gates" that control information flow:

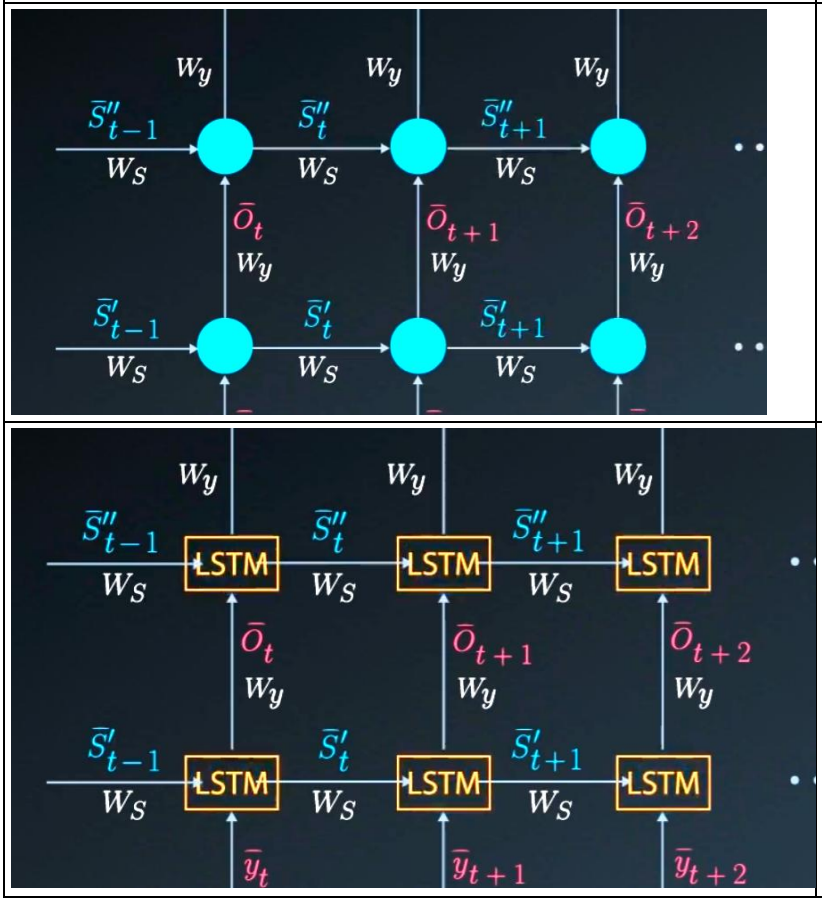
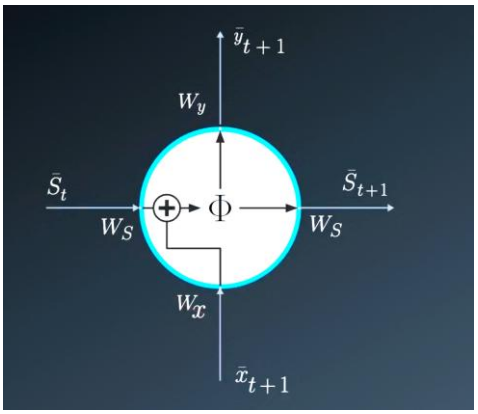
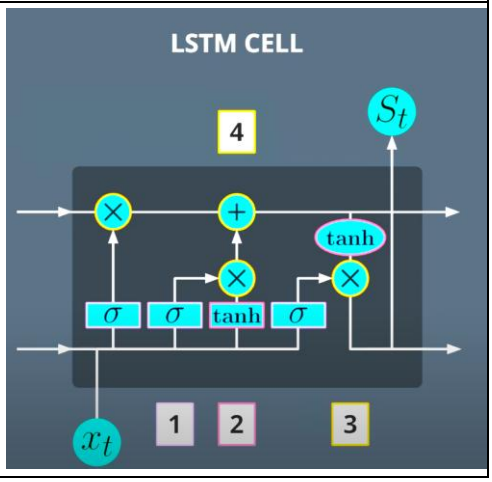
- An input gate deciding what new information to store
- A forget gate deciding what old information to discard
- An output gate determining what information to use for predictions

Think of it like having a personal assistant managing your notes. They decide what new information is important enough to write down, what old notes can be thrown away, and what notes are relevant for your current task.

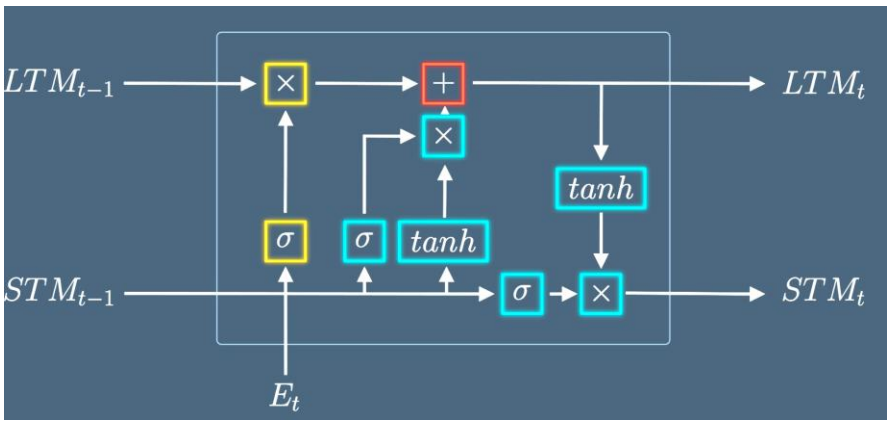
GRUs are a streamlined version of LSTMs. They combine the input and forget gates into a single "update gate" and make some other simplifications. Going back to our assistant metaphor, instead of having separate processes for adding new notes and removing old ones, they make these decisions together. GRUs are often just as effective as LSTMs but are simpler and faster to train.

Lets look at LSTM first:

The LSTM picture you often see is of one neuron and not an network.

Full network Architecture	Each RNN Neuron with respect current and previous run
	
	

LSTM Cell:

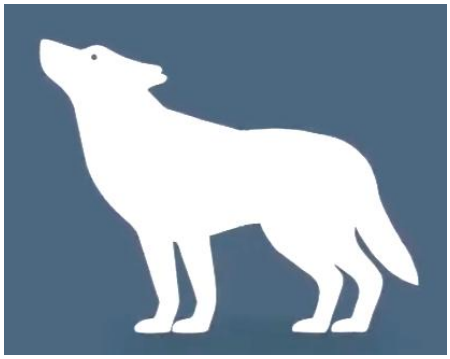


- LSTM Cell are 4 different steps that happens in sequence:
- LSTM works in backpropagation; All the mathematical operations are differentiable which is important for calculating the gradient during BP
- LSTM do not have the restriction like RNN that we can do only 8-10 timesteps instead can do 1000+
- Sigmoids in LSTM help it to:

Know which data to remove / forget	What goes into cell
Know which data to store	What retains within cell
When to use	What passes to output
When to update previous state to next	

LSTM Functioning Overview:

Situation: We have a picture of a wolf that LSTM could easily classify as a Dog but the current context is that we are processing a nature documentary of a forest and its animals. Given this context where in previous rounds we processed an image or text of trees, bear, fox, we want the LSTM to classify image/text as wolf which is correct.



Dog or wolf??

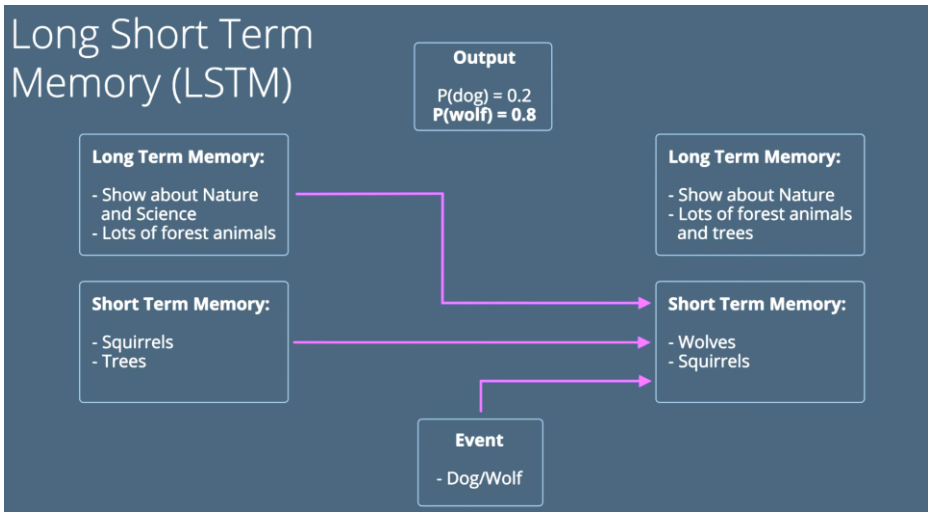
LSTM Inputs and Outputs:

For each LSTM cell the inputs are:

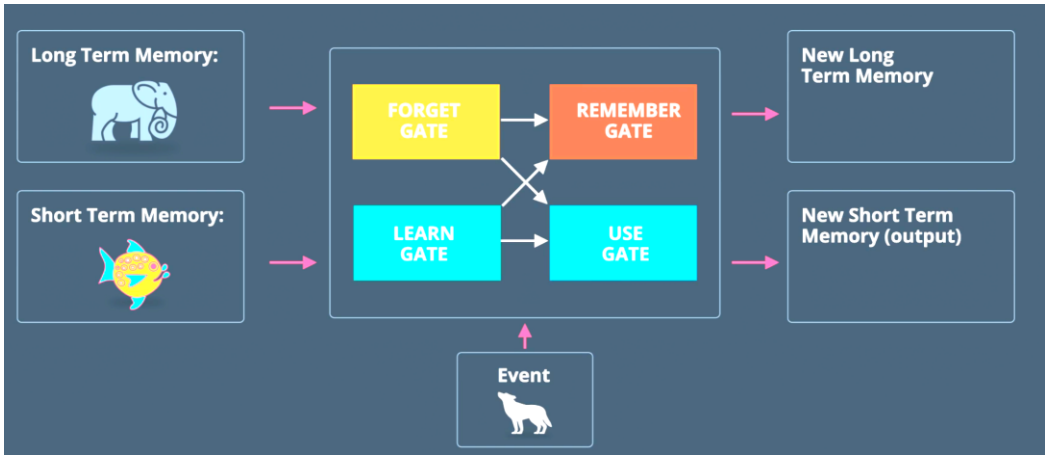
1. Current value - incoming traffic item
2. Short Term memory – Things in context in the recent few inputs
3. Long Term memory – Big picture – things in context with regard the entire corpus or dataset

Outputs:

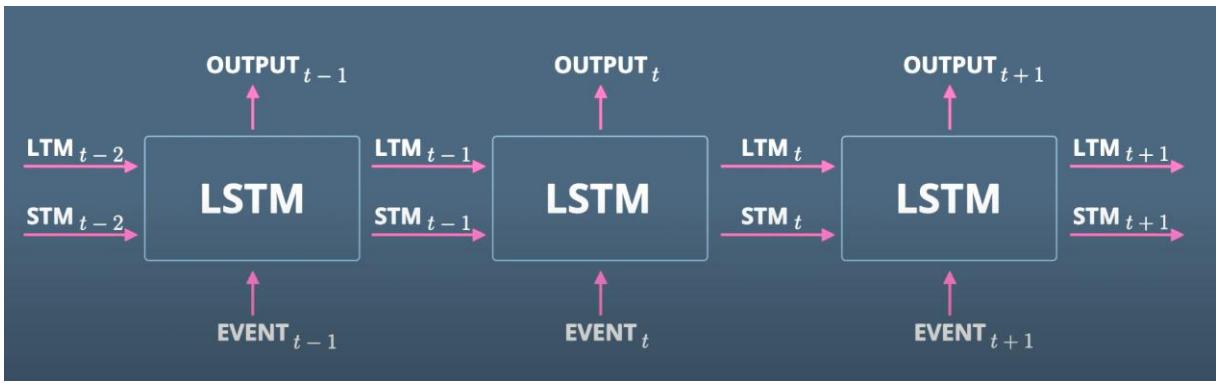
1. New Output – prediction that it is a wolf
    - LSTM uses the Long term memory to correctly classify input as wolf
2. Updated Long term memory
    - Update the Long term memory appropriately if only needed
3. Updated Short Term Memory
    - Update the short term memory so wolf is in the list



LSTM Gates:



- Above the New Short Term Memory box also represents the Output prediction value



Transformers:

So while LSTMs have been very effective in handling sequential data, they do have some limitations:

1.Limited attention span - They struggle to capture long term dependencies in sequences as they maintain a limited amount of information in memory.

2.Computation efficiency - LSTMs are computationally expensive to train.

3.Handling multiple sequences - LSTMs are designed to handle one sequence at a time.

Transformers overcome all these limitations of LSTM by using **self-attention** and **parallel processing**.

Transformers

LSTMs / RNNs have been very effective in handling sequential data, they do have some limitations:

1.**Slow: No Parallelization!** Handling multiple sequences - LSTMs are designed to handle one sequence at a time which is just not feasible since Scientists knew that they needed to process a lot of text (1Mil years worth text done the old non-transformer way) in-order to achieve the foundational models. **Hence Transformers were created/born**. Transformers can do it all parallelly and all at once helping achieve the foundational LLM models.

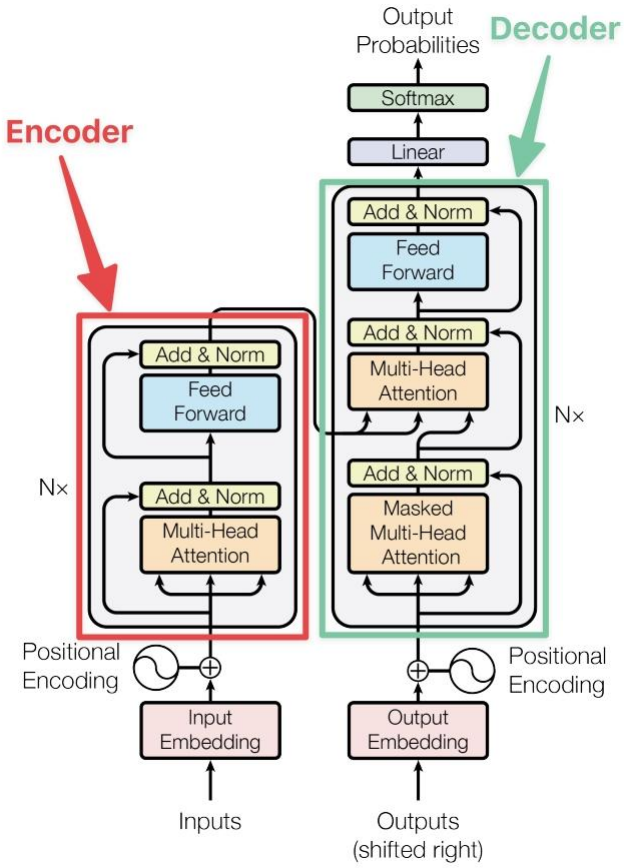
2.Limited attention span - They struggle to capture long term dependencies in sequences as they maintain a limited amount of information in memory. RNNs also limited due to vanishing gradient issue.

3.Performance: Transformers outperform LSTM / RNN in many benchmarks.

Transformers overcome all these limitations of LSTM by using **self-attention** and **parallel processing**.

Transformer models have been shown to achieve state-of-the-art performance on a wide range of NLP tasks, including: language translation, text generation, question answering, sentiment analysis and named-entity recognition

This has led to their widespread adoption in industry and academia, and they are now the dominant approach for many NLP applications. Their impact has been particularly significant in the development of large-scale language models, such as Bidirectional Encoder Representation Transformer (BERT), and Generative Pre-trained Transformer (GPT), which have revolutionized the field of NLP across a wide range of tasks.

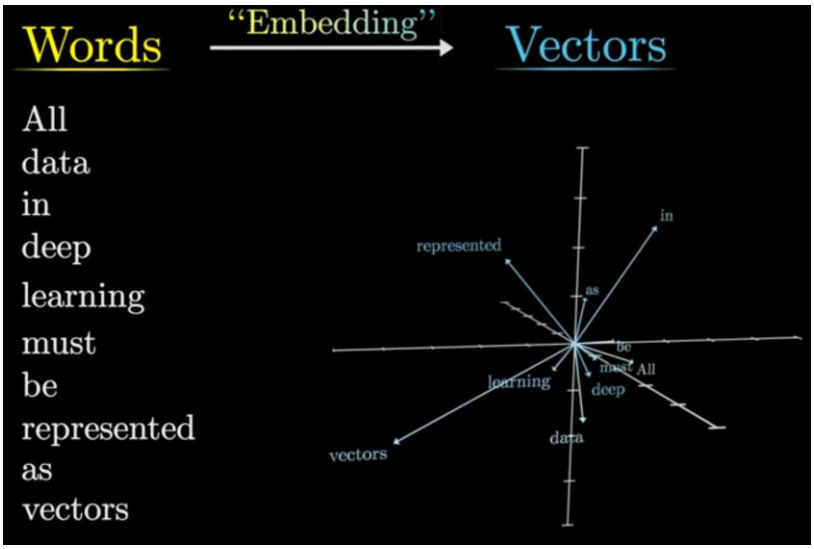


At a high level, the transformer architecture consists of an encoder and a decoder.

- The encoder takes in a sequence of input tokens and produces a sequence of hidden representations
- The decoder takes in the encoder's output and generates a sequence of output tokens.

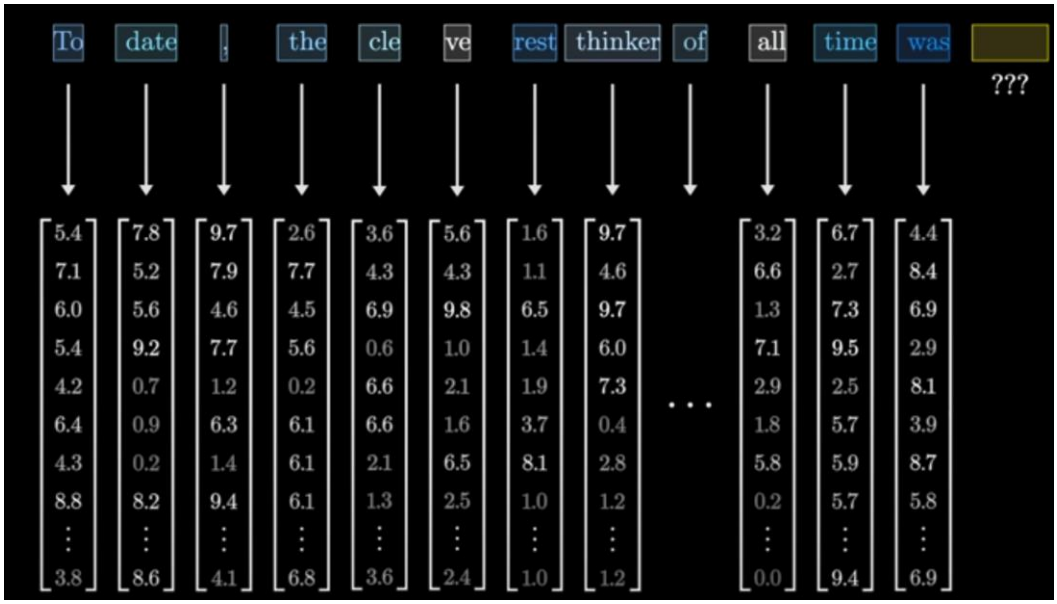
Word Embedding

Word Embedding is a way to convert words into dense numerical vectors that represent semantic meaning. These vectors represent words in a multi-dimensional space, where similar words are located closer to each other.



Embedding in English = to fix something firmly into something else eg: fixing/embedding a nail into a wall

Embedding in ML = A word/object is given a fixed position in mathematical space. Embedding refers to coordinates in a multi-dimensional space and thus every word / object gets embedded in this space and one can then mathematical find their relationship with a simple distance between them. The axes of this space represent the separate factors/relationship perspective that ML has learned. This is not limited to words but any "object" which the machine will convert to an embedding with numbers to represent that object



Note: words isnt always full words but part of a word aka **tokens**.

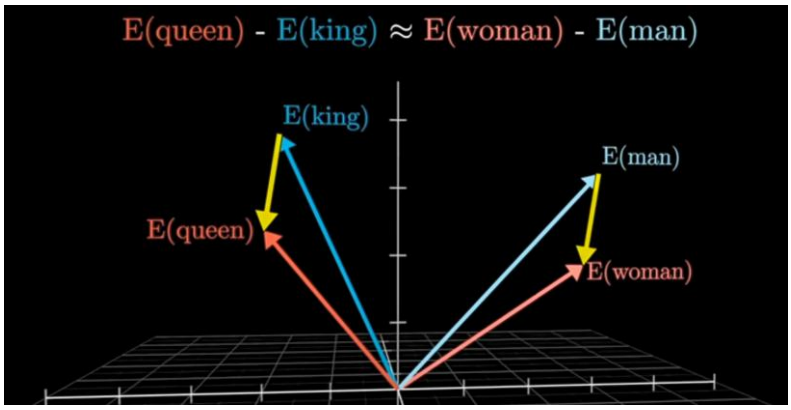
Embedding is conversion of text to a number which is better understood by a machine. The number is in vector form and usually in multiple dimensions but lets keep it 3D (x , y and z coordinates) for examples below.

"king" = [0.9, 0.1, 0.2] "queen" = [0.8, 0.2, 0.2]

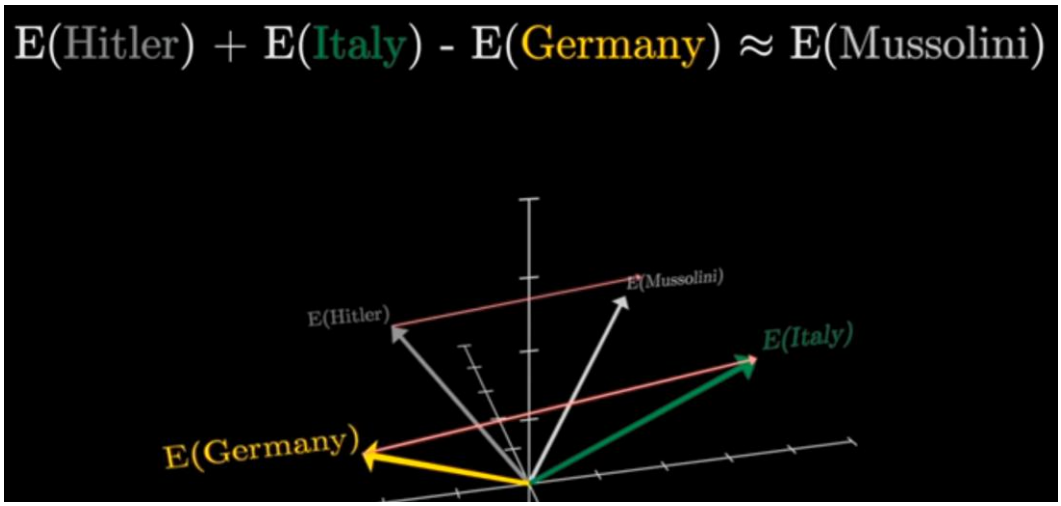
"man" = [0.5, 0.1, 0.1] "woman" = [0.4, 0.2, 0.1]

"dog" = [0.2, 0.6, 0.1] "cat" = [0.2, 0.5, 0.1]

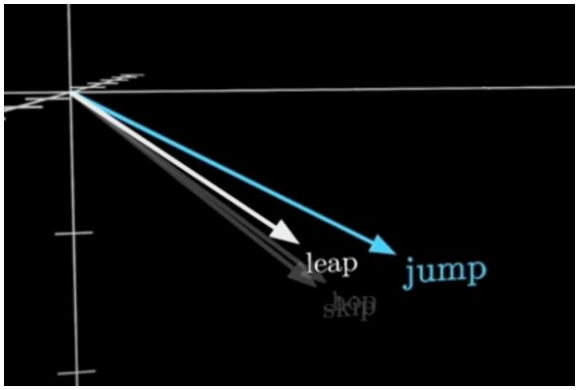
The first number in the array could be referring to “Human” , second number for “animal”



An even more interesting example is that if you add Hitler to Italy and subtract germany you actually get Mussolini



The embeddings capture relationships: **Semantic similarity**: "king" and "queen" are closer to each other than to "man" or "woman." or dog and cat.



How does machine makes use of these numbers to understand the relationship between the words?

King - man + woman = queen

[0.9, 0.1, 0.2] - [0.5, 0.1, 0.1] + [0.4, 0.2, 0.1] ≈ [0.8, 0.2, 0.2]

How are embeddings created?

Embeddings are typically learned during training on large text datasets. The process involves adjusting the embedding vectors based on the relationships between words observed in the data. For example:If "king" and "queen" often appear in similar contexts, their vectors will be adjusted to be close to each other.

- Models like **Word2Vec**, **GloVe**, and **FastText** pre-train embeddings on huge corpora to capture general semantic relationships.

Are embeddings created from scratch for every dataset? Why not have a global, universal vector database?

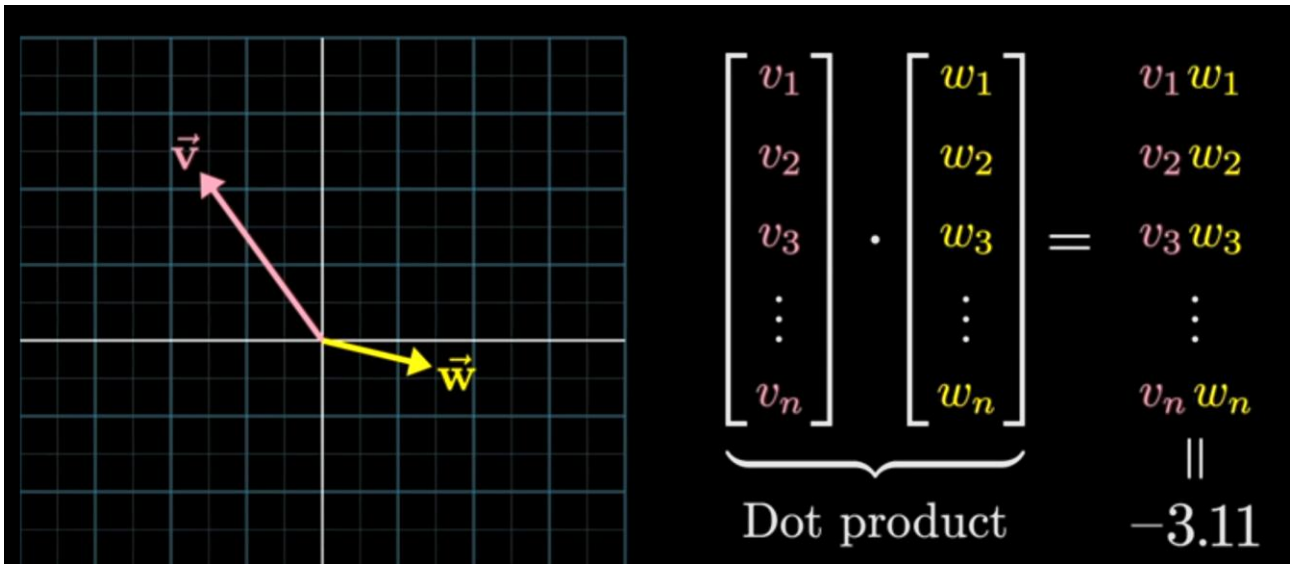
Moslty but not always. Pretrained embeddings like **GloVe** or **FastText** give a good starting point and like Transfer learning, can be used like pre-trained embedding db to re-train for the dataset at hand. Training from scratch is also many times the only way like in case of medical/legal use case where the words have a totally different meaning and so must be done from scratch.

Words are too nuanced with different meanings with time, regions and domains (medical/legal) hence a universal vector does not work but using pre-trained vector db do help

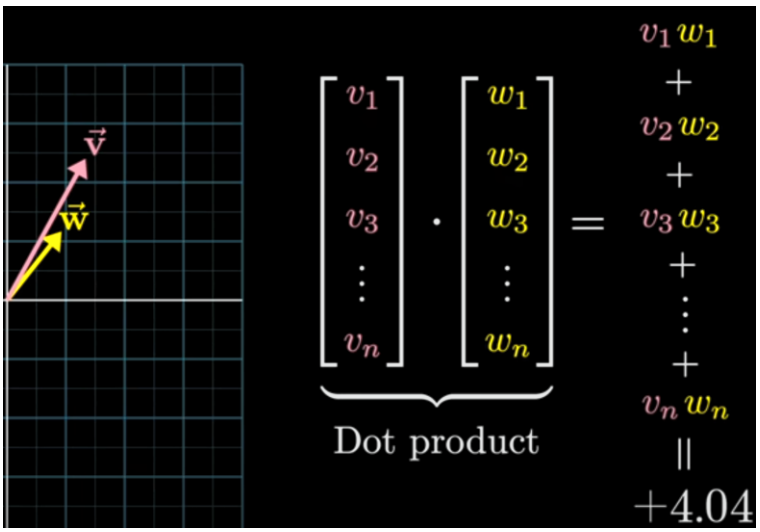
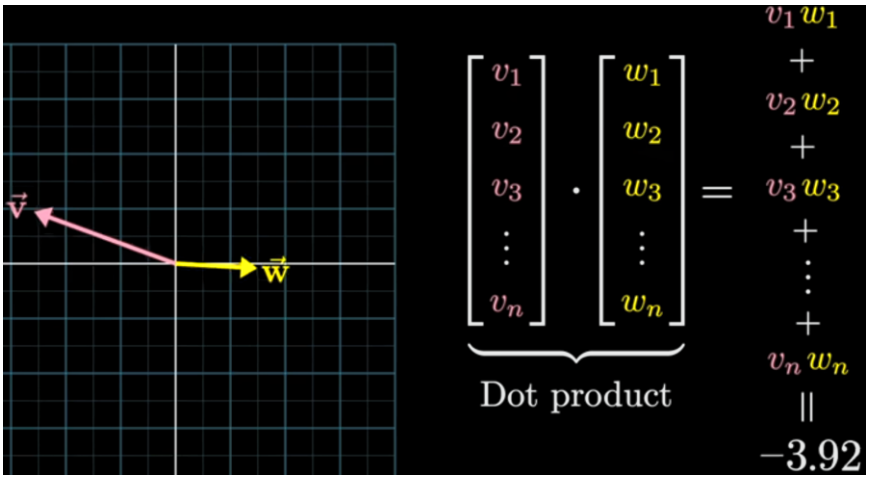
Basic Linear Algebra:

Dot Product of Embeddings:

Multiply corresponding elements and add them together to get one value like weighted sum.



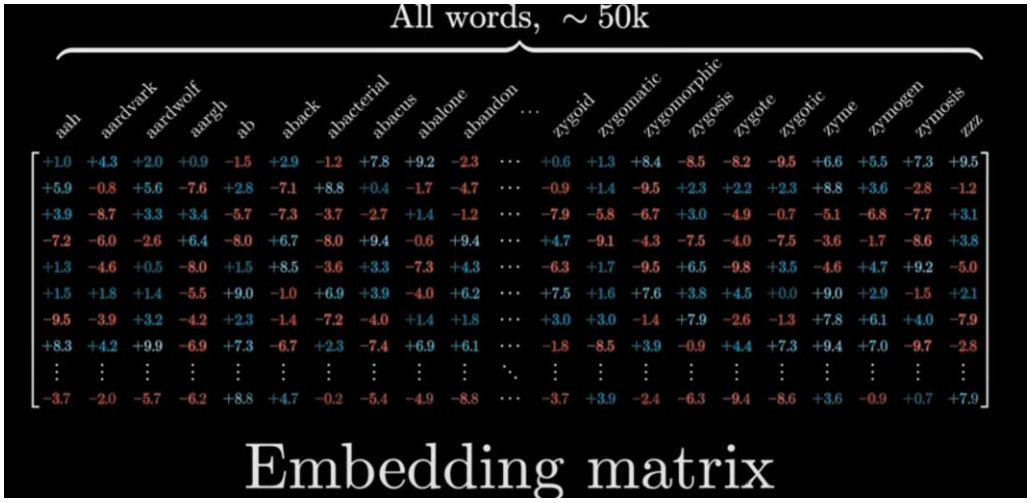
Dot product between two embedding indicate their similarity.

	Dot product is Positive if the embeddings are similar
	Dot product is negative if the embeddings are in opposite directions

Matrices in Transformers:

Embedding Matrix

Model's dictionary in numeric form

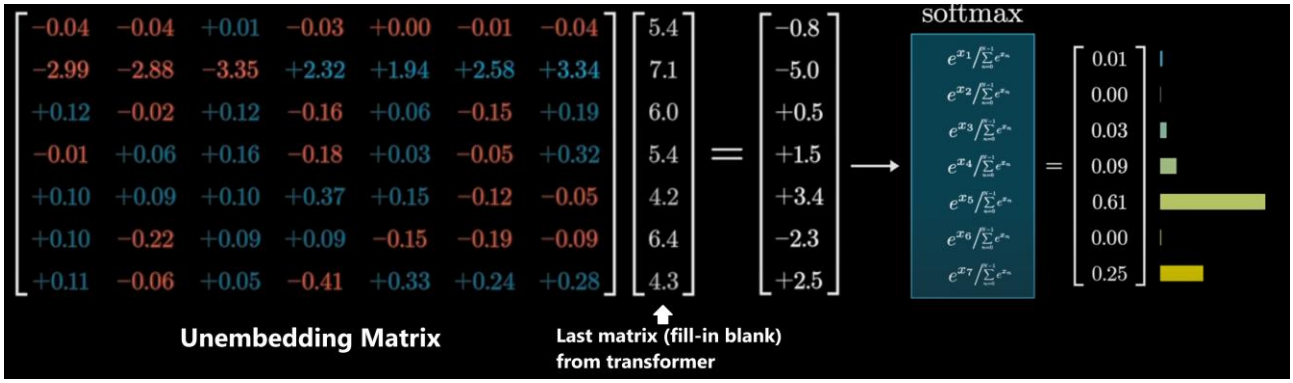


Model picks these matrices for the initial vectors. Real Life Embedding matrices will not be words but just tokens so partial words. What's the Height of the vectors? GPT3's are 12K long.

Note: These words are like global average or vector of individual word in isolation. While Words in context of other words around it mean entirely different. Eg: River “bank”, Financial “Bank”. So these initial vectors get manipulated by the Transformers a lot to arrive at a values that define it contextually. Thus the “bank” of River “bank” ends up correctly different from Financial “Bank”.

UnEmbedding Matrix W_u

Used at the end to find the prediction word.

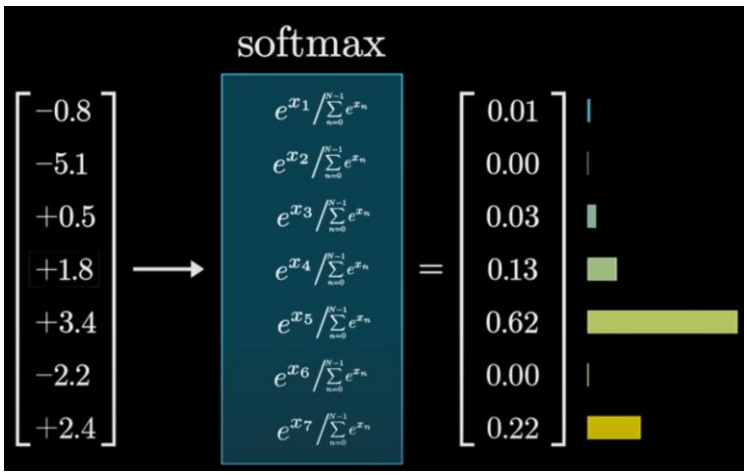


- W_u has random values in the beginning and gets updated during training process
- Has 1 row for each word in the Transformer’s pretrained dictionary and its length is 12K long which represents a token in the Embedding Matrix above but in Transposed form.
-

Softmax

1. Raises all values given to the power of e (aka Logits)
   - $e^{-0.8}$, $e^{-5.1}$, $e^{0.5}$ …
2. Sums all the e^x values created : $e^{-0.8} + e^{-5.1} + e^{0.5}$ … and divides each with the sum

   The output value created is like a probability distribution with a total of 1.



Terminologies:

Transformer Multi-Head

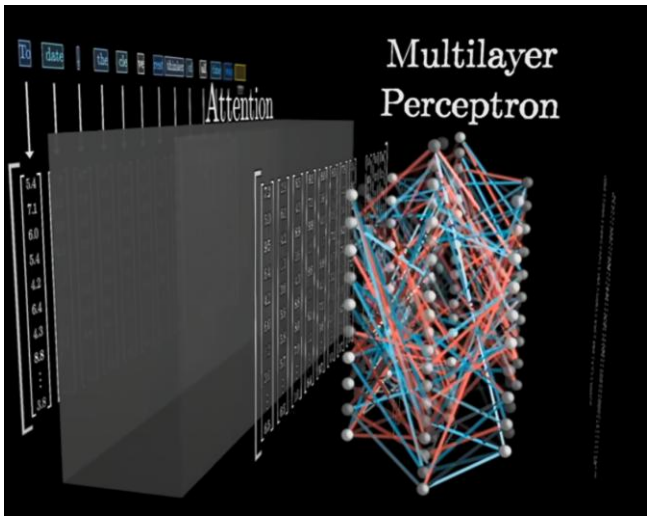
Multi-head attention is a mechanism for allowing the model to focus on different parts of the input sequence at different levels of abstraction. This can help it capture more complex relationships between words in a sentence. It will allow the model to attend to multiple parts of the input sequence simultaneously, so multi-head attention can help it handle longer sequences more effectively.

Context Size :

The fixed number of vectors a transformer could process at a time. GPT3 was 2048.

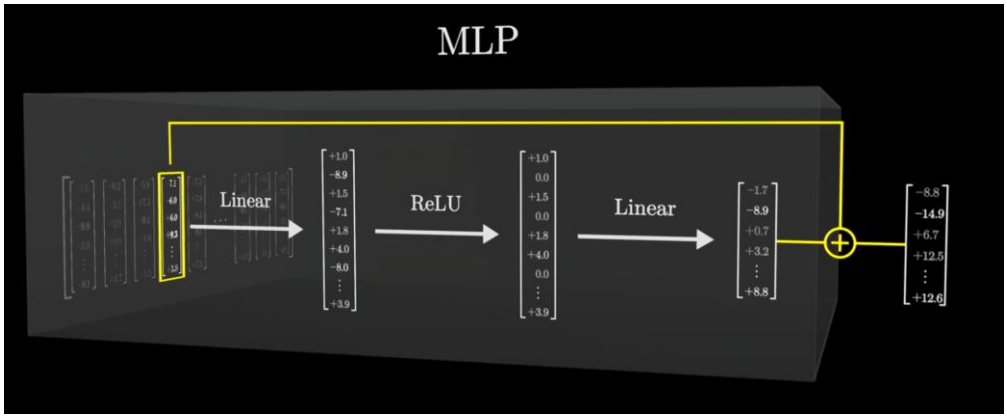
MLP / FFNN in Transformers

3bluw1brown: https://www.youtube.com/watch?v=9-Jl0dxWQs8



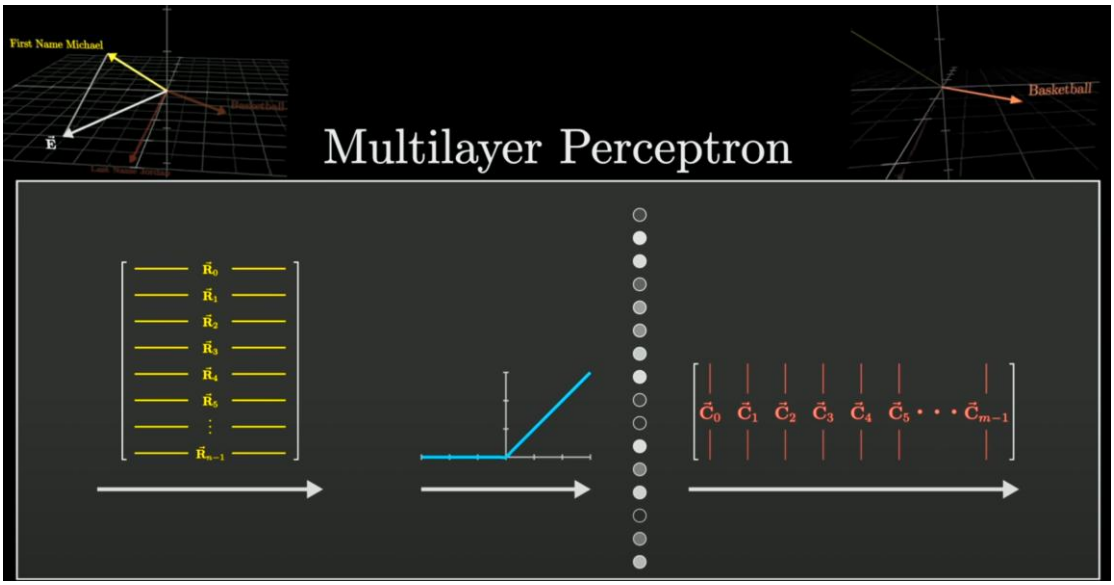
Transformers usually has the Attention blocks followed by few MLP layers before another sequence of Attention+MLP blocks.

What happens in the Attention Blocks and what happens in the MLP?

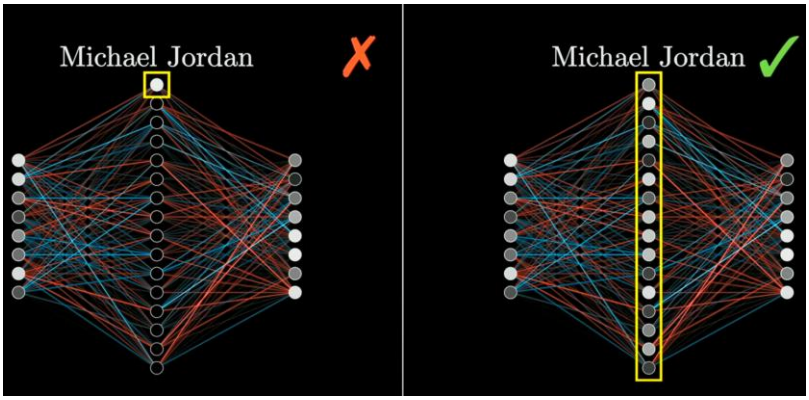


- Each embedding vector from Attention block goes through series of selective encoding:
  - Input Emb vector -> Linear -> Activation -> Linear
- **What does each above do?**

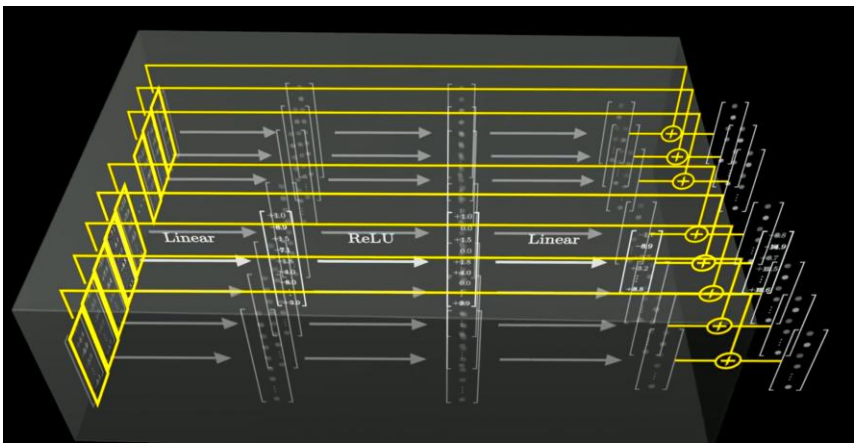
1. First Linear is a questionnaire that checks to see if any of the Embedding vector aligns with a data dimension or simply a question. This dot product action, if high tells yes they are aligned and dot product output if low tells not aligned
    - Eg: Is the embedding about Michael Jordan?
2. The activation forces the neuron to fire or not
3. If a neuron fires, then the 2nd Linear layer adds relevant information
    - Michael Jordan data came in -> add Basketball direction to the embedding



- Note: Each single Neuron is not storing some data point but combinations of firings of many neurons in the whole network – like brain neurons



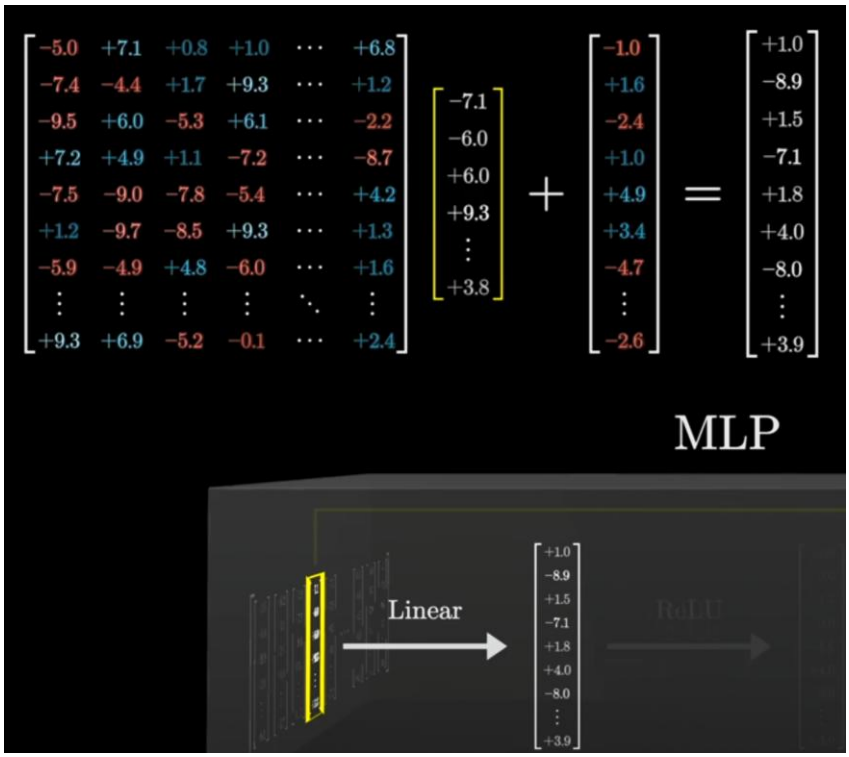
- Every vector coming out of Attention block is treated similarly

- 

- MLP do the work of RNNs here by updating the Attention block outputs. The MLP structure instead of RNN architecture allows parallelization and is many folds faster.
- Attention blocks help to give context (inter-relations between words) – all about changing the generic vector of word “bank” to vector that is for river banks. While the weights of the MLP seem to hold some facts.
  Eg: Michael Jordan is the best in the sport of _____.
  To fill up above blank, none of the Attention block weights will help. So its got to be the MLP weights. Somewhere in the MLP weights this info is encoded.
- In the GPT3 parameter specs of 175B only 1/3rd are Attention weights the rest are MLP weights. Which shows that a lot of data has been memorized into the MLP weights.

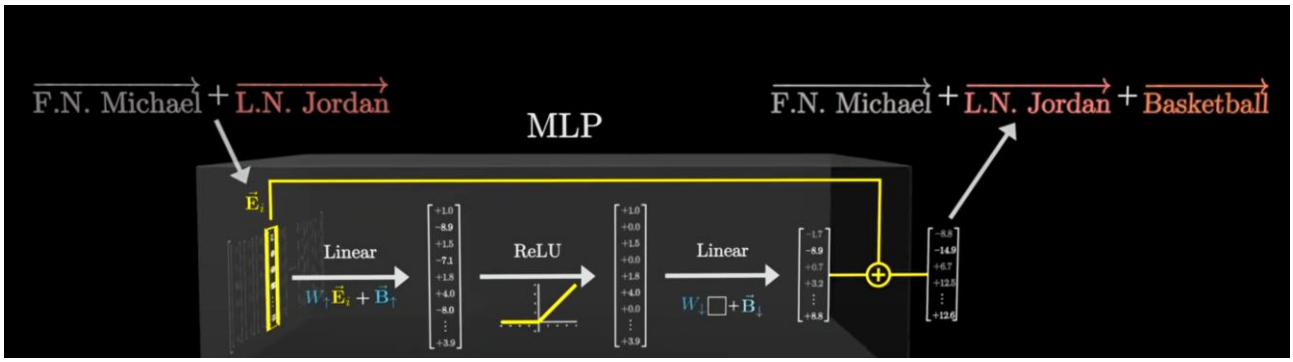
Linear in MLP

Each Linear transformation involves Wx + b change, first multiplied by a matrix of weights and added a bias matrix.

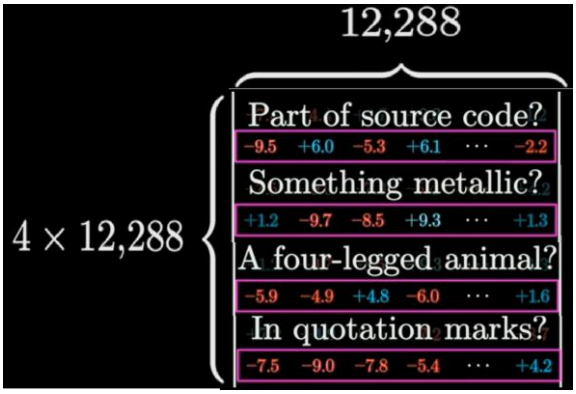


What is happening in the Linear transformation?

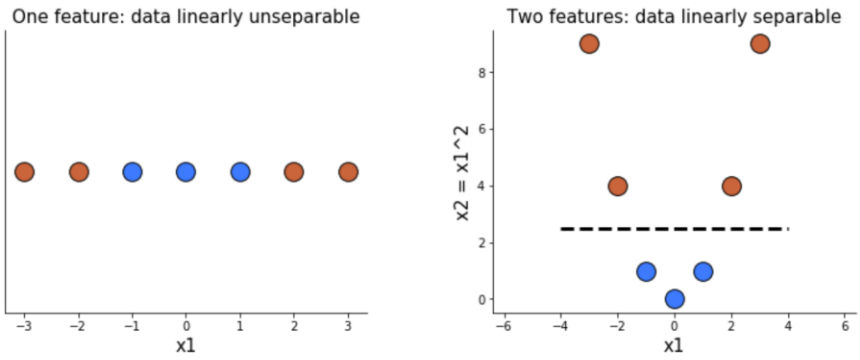
- MLP seem to function like the memory where hard facts seem to be encoded into them which get added to the embedding vectors. Basketball direction gets added when the input has Michael Jordan



- Each row of the weight matrix seem to be some information that is encoded in the weights and gets added to the embeddings

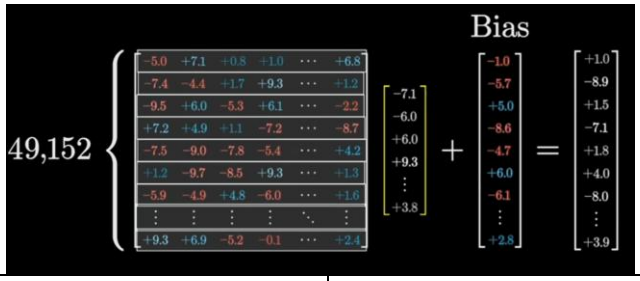


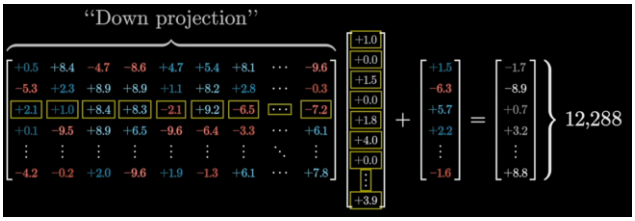
- The height is the number of questions / different dimensions of info that is being encoded
- Each different dimension of data gets added and passed on only if the neuron fires which is based on the input text so all the 50k dimensions dont all gets added to the Embedding during each run. Only when the right data comes in, will the relevant datapoint get added to it.
- The length is the same as the number of dimensions in each Embedding Vector. Thus 4 X 12K different dimensions of info are being added to the Embedding Vector coming out of Attention block
- The first Linear block is actually taking the Embedding vector to a higher dimension space where these data points are being added and then its scaled down in the Second Linear Block.
    - This is similar to SVM handling non-linearly separable data where its Kernel takes the data to a higher dimension, separate the data and then box it back to lower dimension.



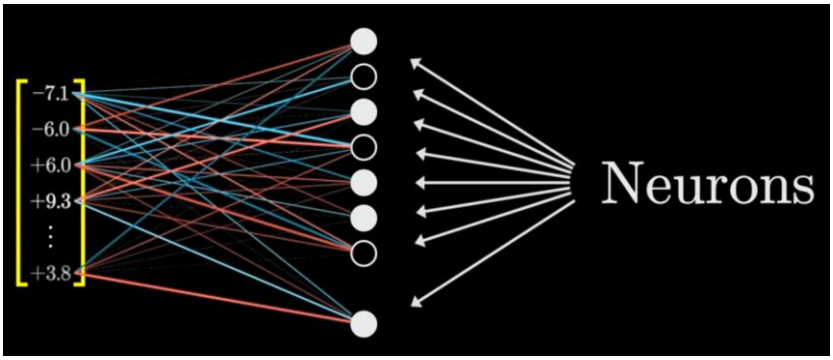
Adding another feature makes data linearly separable.

-



	
First Linear projects data upto a higher dimension.	Second Linear projects data down to lower dimension

- The Activation in between the two linear adds some non-linearity to the data
- Each of the Embedding Vector value is handled separately in each Neuron as it goes through the MLP
-



Superpositioning

Research indicates some hint on how these Transformers seem to hold so much info and scale well.

5. First, in an N dimensional space, one can fit N perpendicular vectors – N different categories of information.

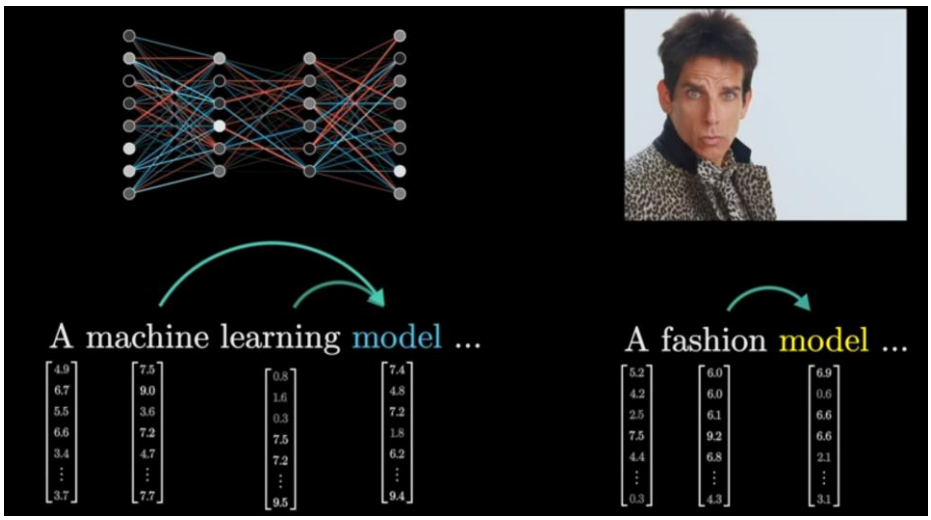
- So GPT3 Embedding vector has 12K vectors, so it implies 12K dimensions of info being remembered and checked by the GPT3 or is it only so? Why? Cos, 12K is too small for amount of perception these GPT3s seem to have.

6. Research suggest that these Transformers might be hacking by having not exactly perpendicular spaces to separate different dimensions/categories but nearly perpendicular - 88^{o} to 92^{o}. In this space you can fit a lot of different ideas / dimensions thus making it possible for the Transformers to have exponentially huge huge space to play around and categorize different ideas/facts and not just the 12K that we give it in form of a Embedding vector.

   This also explain why models with bigger dimensions do better.

Attention Block - Self Attention Mechanism

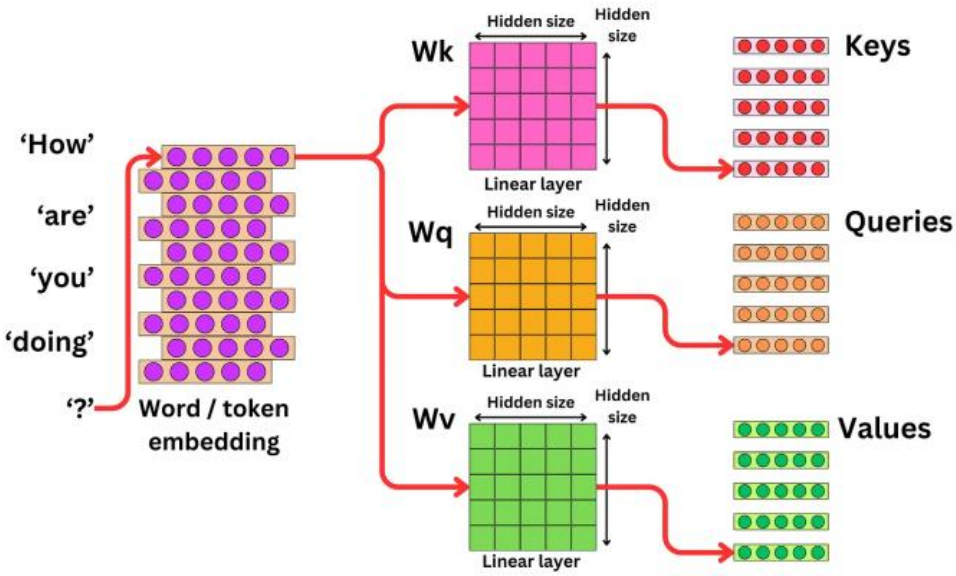
Self Attention is all about changing the embeddings relative the text around it. Initial embedding could be a global average but the self attention makes it more contextual by updating the embeddings during this process. Besides, while doing these cross multiplications to influence each of the words around themselves, if you put a blank matrix at the end of the sentence and it gets filled up by the end of all the runs, you end up with a vector that if you look at global embeddings will give you options for what the predicted word could be. Here you use a custom language model to choose the best of the options provided.



Self-Attention updates the embedding values based on context.

_ Note: Most of the actions below seem sequential but actually are parallelized in GPU

Self-attention mechanism allow the model to selectively focus on different parts of the input sequence when computing the hidden representations.



Transformers takes each word and passes through 3 linear matrices (like convolutional layer) and creates 3 vectors:

- Key : what the word offers to others

- Query : what the word is looking for

- Value : the actual information the word carries

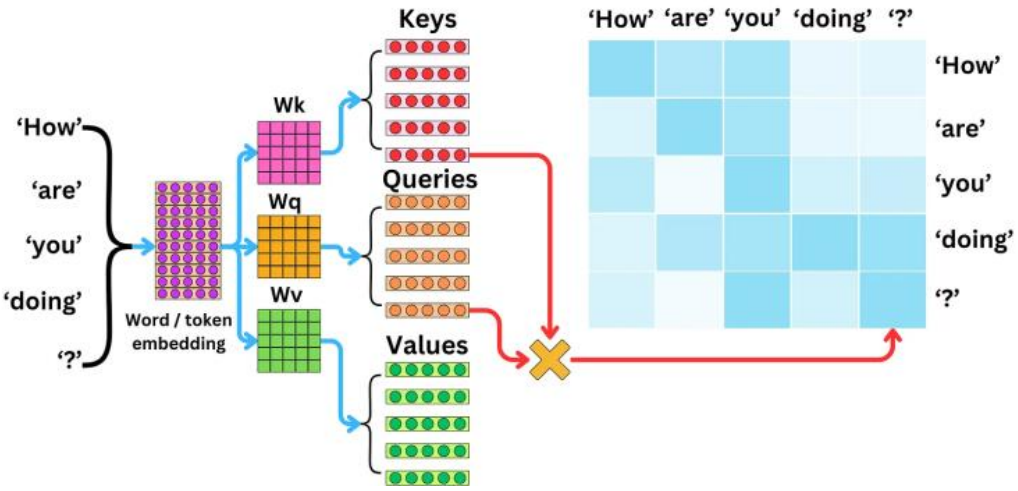
Key, Value, and Query vectors are used in transformers to help the model focus on important parts of the input sequence and produce more accurate and relevant output.

What Key and Query, does is still not well understood. They like thro multiple numbers at each other and then Key matrix is dot producted with Query to represent the numeric for inter-relations of every word with each other. One thing is that the Word embedding matrix has a N height where N are the different possible dimensions a word is being expressed. The Key-Query Matrix has the same dimensions so its like taking a shot at all these embedding dimension and also giving a way for the Transformer an option to change these during training. With all the training, the Key-Query matrix gets a form which has quantified these inter-relations between the words and make them contextual.

To apply them back on the word embedding matrix and update their vectors, we multiply this Key-Query interaction matrix with Value Matrix. The result is the delta or the gradient that then must be applied on the Embedding Matrix updating its values such that the new vectors are more contextual of each other.

Workflow:

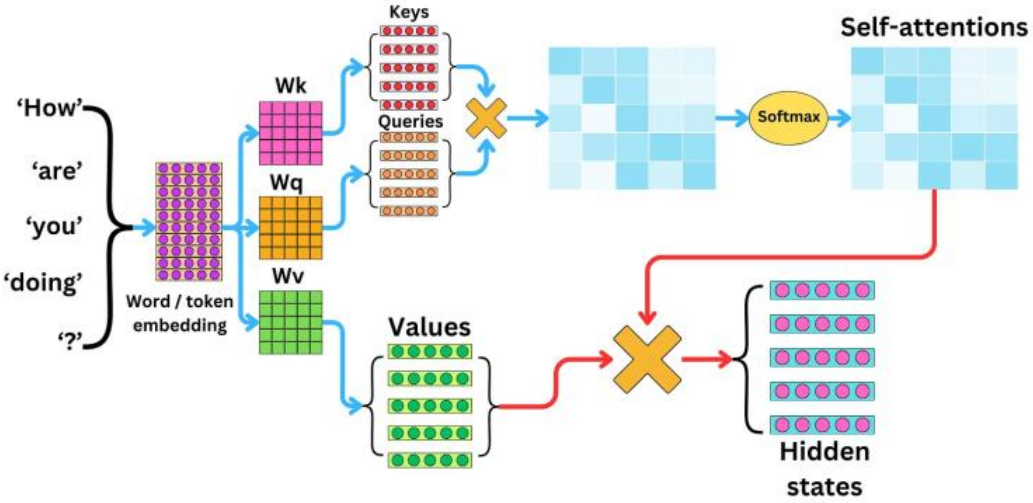
Key and Query are multiplied to get the Key-Query matrix or Interaction score matrix:

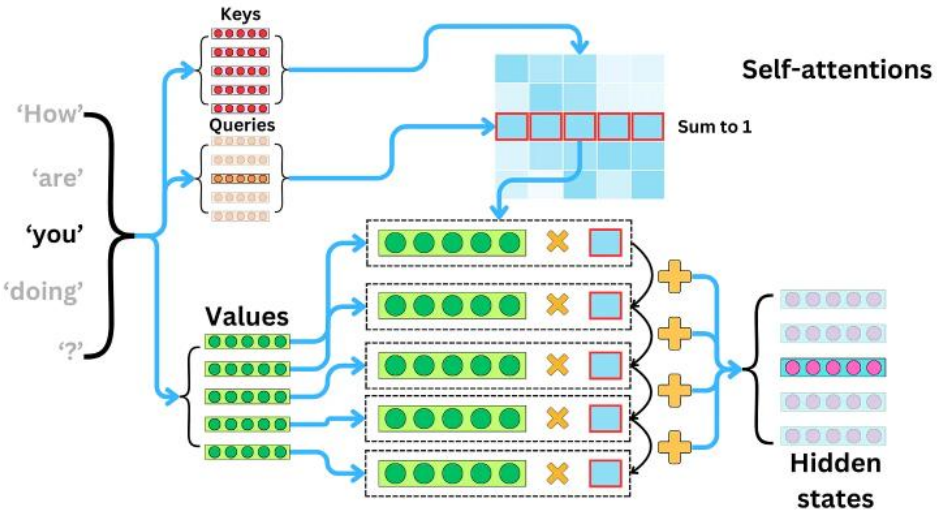


Then Normalization is done to the Key-Query matrix with Softmax application to to convert the interactions to probabilities and thus forming the **Self-Attention Matrix**. The softmax is such that each row sums to 1. The values are called Attention Weights. Attention weights indicate how much attention to give to each value vector.

Note: The softmax also brings in non-linearity into the eqn.

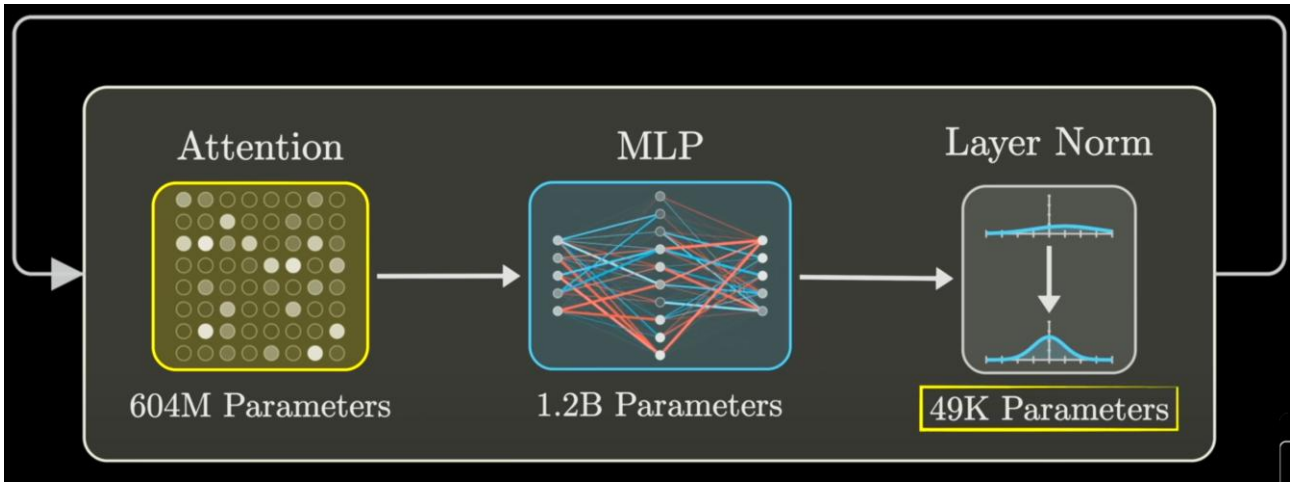
Self-Attention Matrix or the attention weights are multiplied with Values Vectors from Step 1 to create the Hidden State vectors.





The self-attention mechanism works by computing attention weights between each input token and all other input tokens and using these weights to compute a weighted sum of the input token embeddings. The attention weights are computed using a softmax function applied to the dot product of a query vector, a key vector, and a scaling factor. The query vector is derived from the previous layer's hidden representation, while the key and value vectors are derived from the input embeddings. The resulting weighted sum is fed into a multi-layer perceptron (MLP) to produce the next layer's hidden representation.

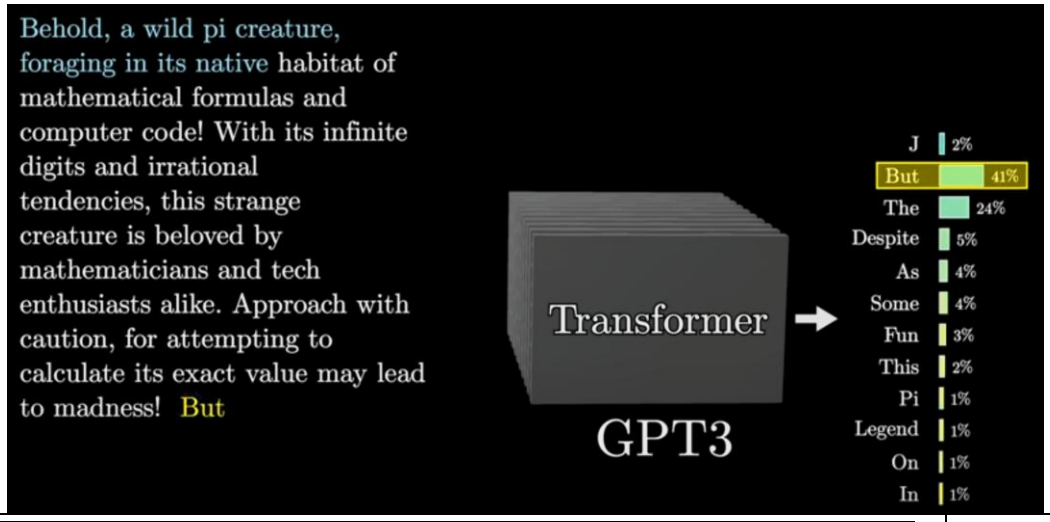
Layer Norm :

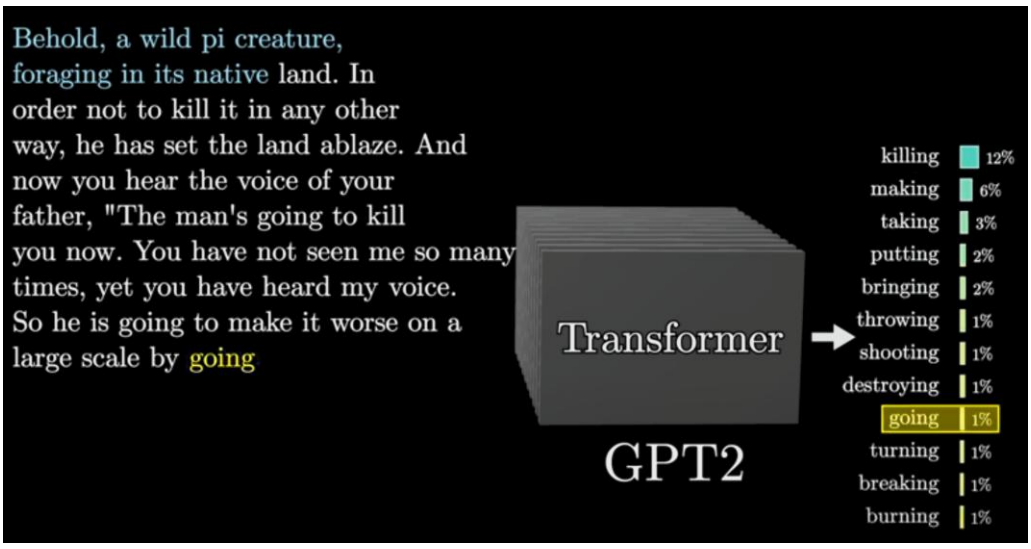


GPT - Generative Pre-Trained Transformer

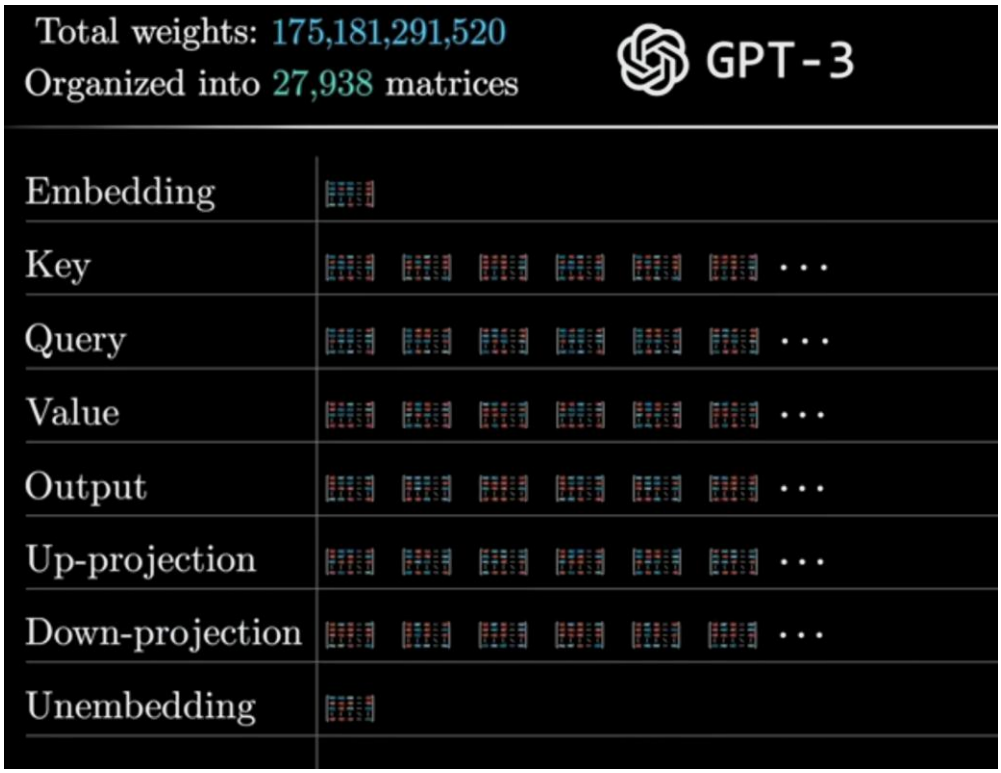
GPT (text)'s goals is simply predict the next word!

GPT generate text by having a starter snippet which is the prompt and then predicts the next word. The predicted word is added to the prompt and the next word is predicted. Same is repeated to get whole lot of text.



	Prompt in GPT2 did not give good results.
	But GPT3 which is a bigger (100x larger parameters) model did well

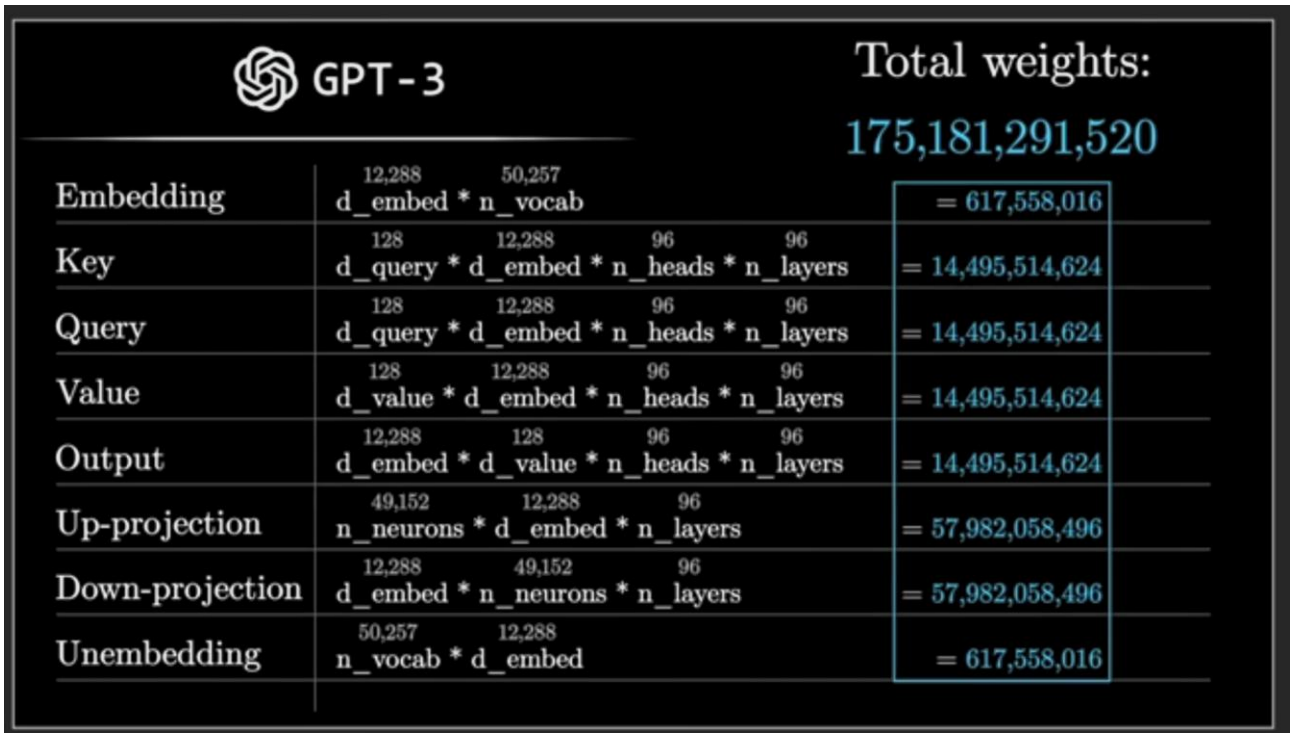
Parameter Specification:



GPT3 is a 175B parameter model.

What is this refering to? 175B different weights which are actually maintained in matrices of different categories – 8 categs for GPT3.

Breaking the numbers down:



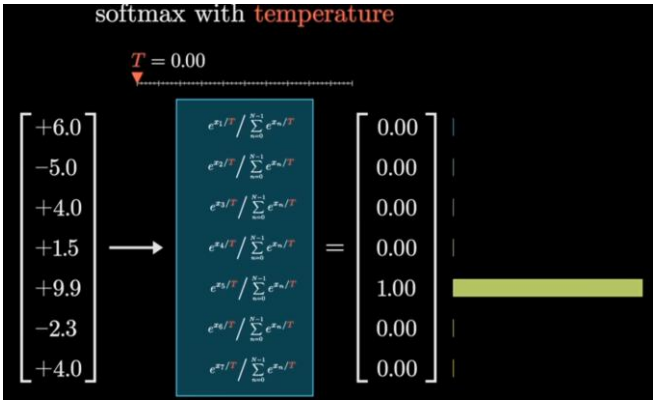
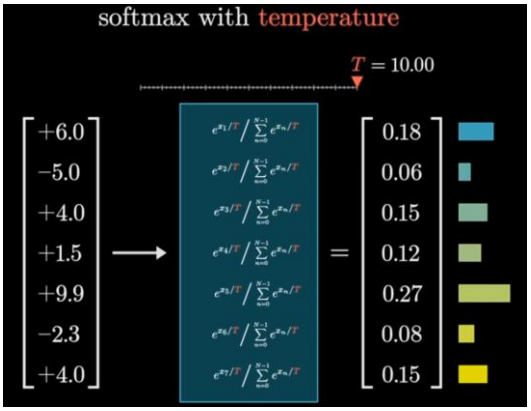
n_vocab = number of words (actually tokens) in dictionary

d_embed = Length of a vector to define each token

Softmax Temperature in Transformers :

To spice up the LLM Outputs one can use the temperature to allow slightly less likely options as the output using Temperature parameter.

While doing softmax of attentions weights, we introduce an new element T for temperature that divides each x value in softmax function. This control element allows us to control where the probabilities will be distributed among all the inputs.

	
If T = 0, then all the probability will go to only the max valu	If T = N, then the probabilities are averaged and distributed which will lead to more equivalent distribution.

BERT

BERT (Bidirectional Encoder Representations from Transformers) is a Machine Learning (ML) model for natural language processing developed by Google in 2018. BERT is a versatile model that can handle a range of natural language processing (NLP) tasks.

Model	Transformer Layers	Hidden Size	Attention Heads	Parameters
BERTbase	12	768	12	110M
BERTlarge	24	1024	16	340M

The above table provides some key specifications of two different versions of the BERT model: BERTbase and BERTlarge.

Transformer Layers: This refers to the number of transformer layers in the BERT model. Transformer layers are a key component of BERT and are responsible for processing the input text.

Hidden Size: This refers to the number of hidden units in each layer of the BERT model. This is an important parameter as it determines the capacity of the model to learn complex patterns in the input data.

Attention Heads: This refers to the number of attention heads used in each transformer layer. Attention heads are responsible for computing the attention scores between different parts of the input sequence, which allows the model to focus on the most relevant parts of the input.

BERT mechanism:

To achieve its remarkable performance, BERT utilizes the following components:

Extensive training data

BERT was trained on a colossal dataset of 3.3 billion words, which is one of the main factors that contributed to its success. Specifically, it was trained on two vast datasets: Wikipedia (about 2.5 billion words) and Google's BooksCorpus (about 800 million words). By using these vast and varied datasets, BERT gained a deep understanding of natural language.

MLM (Masked Language Modeling)

MLM is a technique used by BERT to learn about the relationships between words in a sentence. In this process, BERT is trained to predict what a masked word should be based on the other words in the sentence.

Example:

Let's say we have the following sentence: "The cat sat on the [MASK]".

During pre-training, BERT may randomly mask one of the words in the sentence. In this case, let's say BERT masks the word "mat". The sentence would then look like this: "The cat sat on the [MASK]".

BERT is then trained to predict what the masked word should be based on the other words in the sentence. In this case, the correct answer is "mat". By considering the other words in the sentence, such as "cat" and "sat", BERT is able to make an educated guess that the missing word is "mat".

This process is repeated many times over with different sentences and different masked words, allowing BERT to learn about the relationships between words in a sentence and build a deep understanding of language.

NSP (Next Sentence Prediction)

NSP is another technique used by BERT during pre-training to help it better understand the overall structure and flow of language. In this process, BERT is trained to predict whether two sentences are likely to appear together in a piece of text.

Example: Let's say we have two sentences:

"The cat sat on the mat." ; "It was a beautiful day outside."

During pre-training, BERT may be given these two sentences and asked to predict whether they are likely to appear together in a piece of text. In this case, the answer would be "no" since the two sentences do not seem to be related to each other.

BERT is trained using many different pairs of sentences, some of which are related and some of which are not. By learning to predict whether pairs of sentences are related or not, BERT gains a better understanding of the overall structure and flow of language.

This technique is important because it helps BERT understand the context in which sentences appear, which is crucial for many natural language processing tasks such as question answering and text classification.

Note: Fine tuning BERT takes a long time (**even on GPUs**), hence we are not providing a workspace for this demo. Please try this on your local machine.

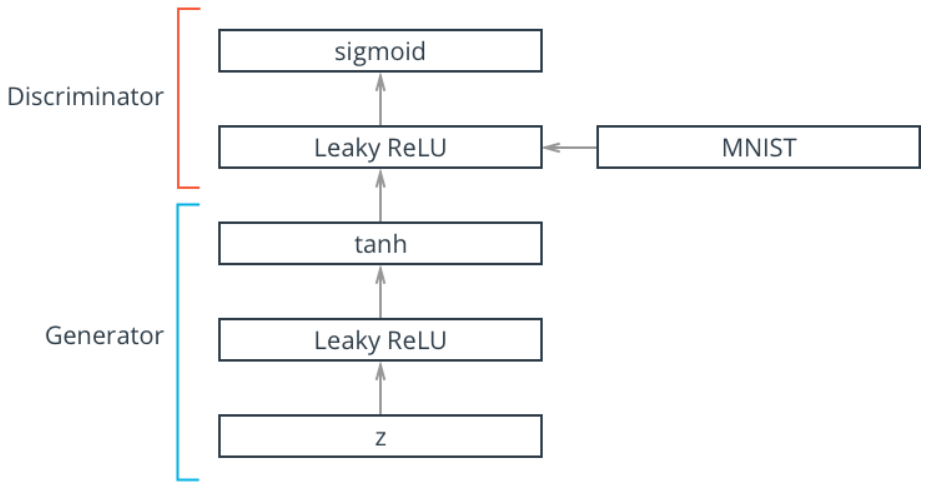
GAN – Generative Adversarial Network

What is a GAN? A **Generative Adversarial Network (GAN)** is a type of deep learning model that consists of two neural networks:

1. **Generator (G)** – Creates fake data (e.g., images, text) that resembles real data.
2. **Discriminator (D)** – Evaluates whether the data is real (from the dataset) or fake (generated by G).

These two networks are trained together in a **competitive game** (hence "adversarial" in the name).

While **Transformers** are primarily **sequence models**, often used for language understanding, generation, and other sequential tasks. **Transformers** architectures are designed for **sequence modeling and language tasks**, using **self-attention** to capture context and relationships across sequences.



Simple Architecture

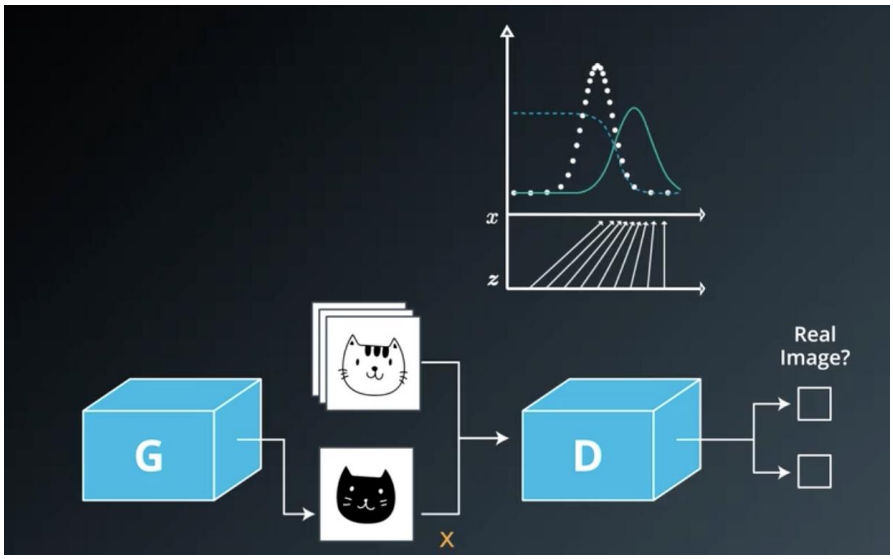
Components of a GAN

1. Generator (G)

- Takes a **random noise vector** (e.g., Gaussian noise).
- Transforms it into a structured output (e.g., an image).
- Goal: Generate data so realistic that the Discriminator **cannot distinguish it from real data**.

2. Discriminator (D)

- A binary classifier (e.g., a CNN for images).
- Takes in an input (either real or fake).
- Outputs a probability **D(x)D(x)D(x)** (higher means real, lower means fake).
- Goal: **Correctly classify real vs. fake**.



Discriminators determine the probability of an image being "real" or "fake"

How GANs Work (Training Process)

1. **Generator produces fake samples** from random noise.
2. **Discriminator evaluates** both real and fake samples.
3. **Loss is calculated**:
    - The Generator tries to **fool the Discriminator**.
    - The Discriminator tries to **catch fakes**.
4. **Backpropagation updates both networks**:
    - The Generator improves to create more realistic samples.
    - The Discriminator improves to better detect fakes.

This **adversarial process** continues until the Generator creates data so realistic that the Discriminator **cannot distinguish real from fake** (i.e., it outputs ~50% confidence for both).

- GANs use a **min-max game**:
- The **Discriminator (D)** tries to maximize; Maximize correct classification
- The **Generator (G)** tries to minimize: Minimize how well D detects fake samples
    - In **practice**, the Generator's loss is often modified so it directly maximizes the probability that the Discriminator thinks its outputs are real.

Sample Flow of Data in GAN:

Let’s say the text prompt is:
📜 **"A man wearing glasses and smiling."**

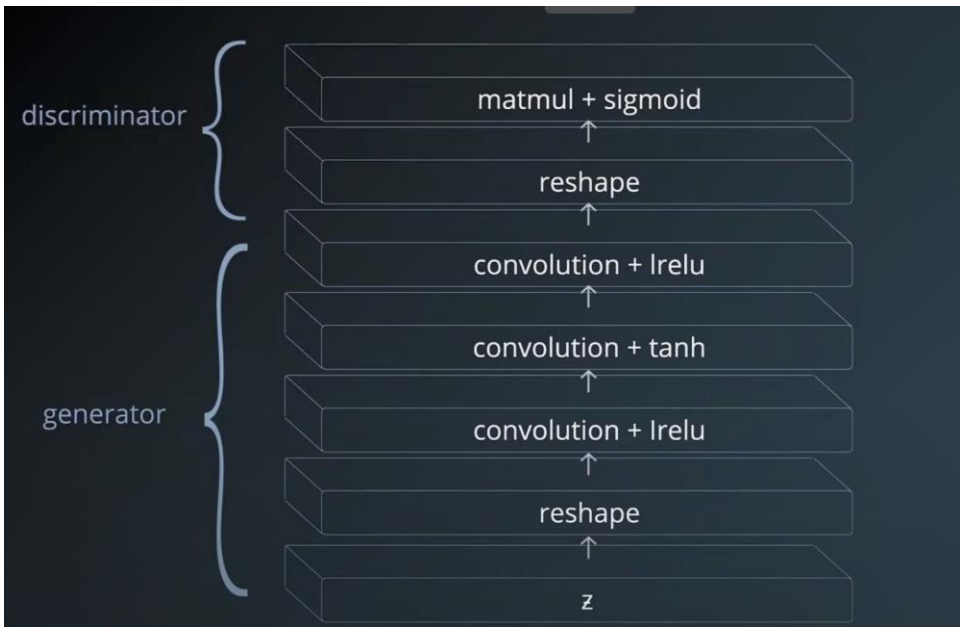
🖼 **First fake image:** A blurry mess → Discriminator: **0.1** (very fake)
🎨 **Second fake image:** A dog → Discriminator: **0.2** (still wrong)
👴 **Third fake image:** A blurry human shape → Discriminator: **0.4** (closer)
👓 **Fourth fake image:** A human with glasses but not smiling → Discriminator: **0.6**
😁 **Fifth fake image:** A smiling man with glasses → Discriminator: **0.9**

Each time, the **gradients help the Generator adjust** toward features that increase realism and match the text.

Popular Architectures for Generator & Discriminator

Generators in GANs can use either **fully connected layers (MLPs)** or **CNNs**, depending on the type of data:

- **For images** → CNN-based architectures (e.g., DCGAN, StyleGAN) are common.
- **For non-image data** (e.g., text, tabular data) → Fully connected layers (MLPs) may be used.



Example GAN with CNN instead of Fully connected toy examples

1. Deep Convolutional GAN (DCGAN)

- **Generator**: Uses **transposed convolutions** (also called **deconvolutions**) to upsample noise into an image.
- **Discriminator**: Uses standard **CNN layers** to classify images as real or fake.

✅ **Key features**:

- BatchNorm for stability.
- Leaky ReLU in Discriminator, ReLU in Generator.
- No fully connected layers after the initial noise input.

2. Wasserstein GAN (WGAN) & WGAN-GP

- Improves **stability** by using a different loss function based on **Wasserstein distance**.
- Uses **weight clipping** (WGAN) or **gradient penalty (WGAN-GP)** to enforce smooth training.

Component	Choice	Why?
Hidden Layer Activation (Generator)	**Leaky ReLU**	Prevents dying neurons, stabilizes training
Output Activation (Generator)	**tanH**	Matches image normalization (-1,1), avoids saturation
Loss Function	**BCE with Logits Loss**	Stable binary classification loss for real vs. Fake. BCE with Logits Loss comes with sigmoid output so not needed specifically. Any other loss function would need one.
Dropout (only for **Discriminator**)		**Prevents Overfitting** – The Discriminator checks if image is fake or not using a fixed dataset. Since the real dataset is fixed, it might memorize real samples instead of learning

general features. Dropout helps regularize the network.
Generator's output is the last and final output. Dropouts in Generator will provide grainy outputs so no dropouts.

Loss Functions in GANs:

GANs have two separate loss functions, one for the Generator and one for the Discriminator, because they are competing against each other in a min-max game.

1. Discriminator Loss (D Loss)

The Discriminator is trained to get binary classification (1 or 0) correct for real and fake images appropriately using **Binary Cross Entropy (BCE) loss.**

2. Generator Loss (G Loss)

Gradient Flow Issues

Gradient flow is a major problem in GANs, and that's why many choices as below are made:

1. Leaky ReLU : Helps maintain **gradient flow** even when activations are negative
2. Tan H activation for Gen Output: Tanh has strong gradients around **[-1,1]**, unlike Sigmoid, which saturates near 0 or 1.
3. Input Data Normalization : If images are in **[0,255]** or **[0,1]**, activations like **Sigmoid** or **ReLU** can saturate (i.e., produce near-zero gradients). Normalizing to **[-1,1]** keeps activations within a range where gradients remain strong.

Wasserstein Loss & Wasserstein Distance

- **Wasserstein Distance (Earth Mover's Distance)**: Measures how much "work" is needed to move the generated distribution to match the real one.
- **Wasserstein Loss**: Used in **Wasserstein GAN (WGAN)** to minimize this distance, leading to **more stable** training compared to BCE loss.

WGANs (GAN using Wasserstein Loss) train slower, needs more compute power, harder to train and requires advanced tune-up (weight clipping, gradient penalty).

When to Use Wasserstein Loss?

- When **mode collapse** is a problem.
- When BCE causes **instability** (e.g., in high-dimensional or complex distributions).
- When training **standard GANs fails to converge properly**.

Wasserstein Loss is more **stable and meaningful**, but it comes at the cost of **complexity, training speed, and tuning difficulty**. Standard BCE is often **sufficient** unless mode collapse or instability is a serious issue.

Mode Collapse in GANs

Mode collapse occurs when the Generator **fails to produce diverse outputs** and instead generates a limited set of repetitive or nearly identical samples. This happens when the Generator finds a "shortcut" to fool the Discriminator without truly capturing the full data distribution.

Why Does it Happen?

- The Discriminator **quickly learns** to reject certain outputs, forcing the Generator to stick to a few samples that work.
- The Generator **fails to capture the full diversity** of the real dataset.
- **Poor training dynamics** where one network (Generator or Discriminator) dominates the other.

How to Prevent?

1. **Use Wasserstein Loss (WGANs)** → Makes training more stable.
2. **Introduce Noise in the Discriminator** → Stops the Discriminator from overpowering
3. **Mini-batch Discrimination** → Encourages diversity by penalizing repetitive outputs.
4. **Feature Matching** → Forces the Generator to match feature statistics instead of just fooling the Discriminator.
5. **Unrolled GANs** → Looks ahead at future steps to avoid local minima.
6.

Latent Vector and Latent Dimension

Latent Vector and Latent Dimension are unique to GANs.

In GANs, the latent dimension refers to the size/dimensionality of the input space that the generator network uses to create synthetic data. A latent vector is a specific point in this latent space - typically a random vector of numbers that serves as input to the generator.

For example, if your latent dimension is 100, each latent vector would be an array of 100 random numbers. The generator learns to map these latent vectors to realistic-looking output data (like images).

Think of latent vectors as "seeds" that the generator uses to create new data. Different latent vectors produce different outputs, while similar latent vectors tend to produce similar outputs.

CNN in GANs:

GANs that use CNNs, the Generator does the opposite of a typical CNN used for classification.

1. CNN in a Classifier (Downsampling)

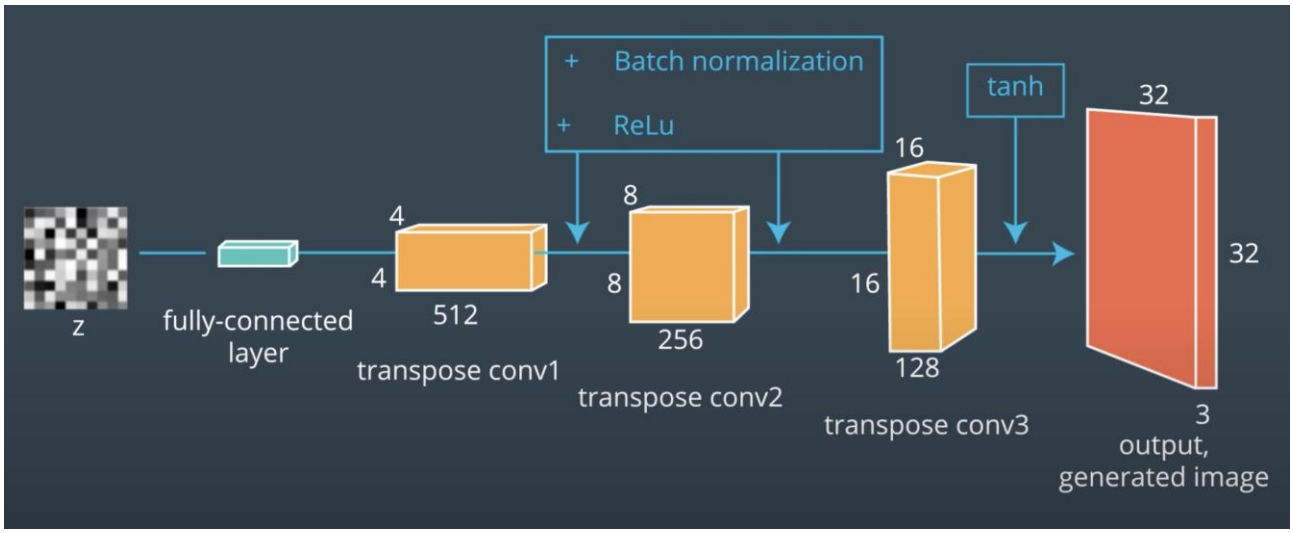
- In a standard CNN (e.g., for classification), the input is a **large image** (e.g., 128×128).
- The CNN **reduces spatial dimensions** using **convolutions, pooling, and strides** while increasing the number of **feature maps**.
- The final output is a **small feature vector** used for classification.

👉 **Large image → Small feature representation**

2. CNN in a GAN Generator (Upsampling)

- The Generator starts with a **small latent vector** (e.g., 100-dim Gaussian noise).
- This is passed through **fully connected layers**, reshaped into a small **low-resolution feature map** (e.g., 4×4 or 8×8).
- Then, **Transposed Convolutions (a.k.a. Deconvolutions), Upsampling, or PixelShuffle** are used to **increase** the spatial size while reducing feature depth.

👉 **Small latent vector → Small feature map → Large generated image**



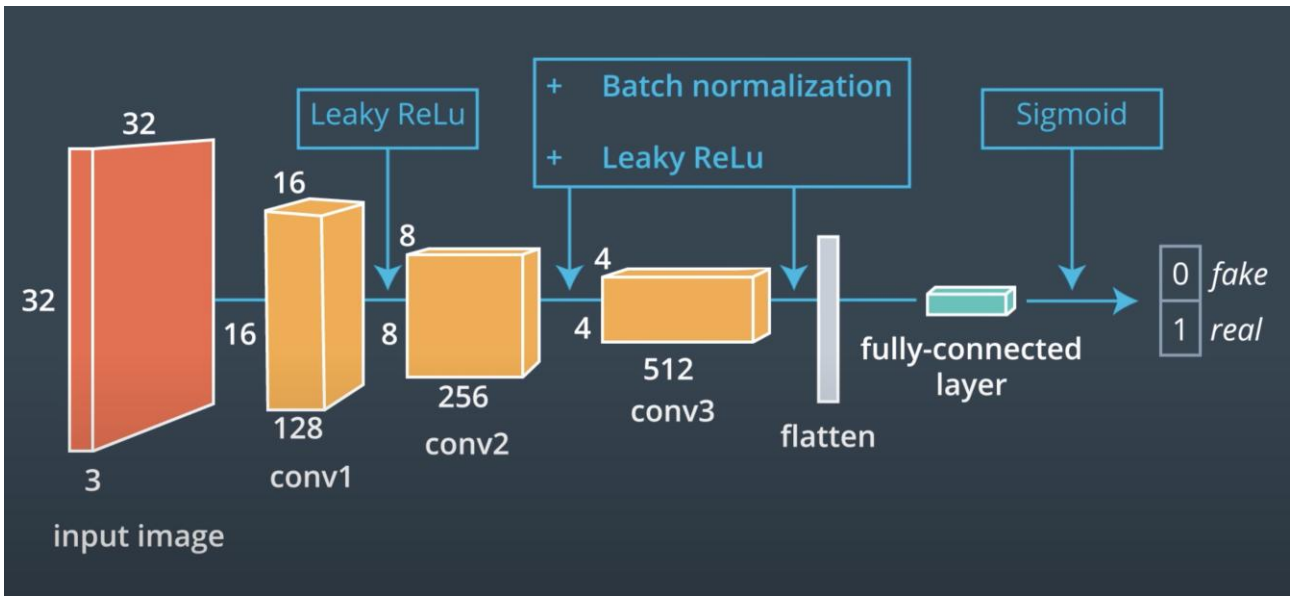
TIPS and Tricks in real life GANs:

Transposed Convolutions in GANs

Transposed Convolutions (also called **Deconvolutions** or **Fractionally Strided Convolutions**) are used in GANs **to increase the spatial resolution** of a feature map, which is essential for the **Generator** to create realistic high-resolution images.

- A **transposed convolution** does the opposite: it increases spatial size by "spreading out" the input values.
- Transposed Convolutions allow the **Generator** to progressively **increase** image resolution in a **learnable way**.
- Unlike simple **bilinear or nearest-neighbor upsampling**, transposed convolutions **learn the best way** to upscale images while adding important details.
- Helps GANs **generate detailed images** by allowing the model to learn **spatial correlations** effectively.

📌 **Key idea:** Instead of sliding a kernel over the input, transposed convolutions **insert gaps** between pixels and then apply a normal convolution to "fill in" these gaps.



Label Smoothing in GANs

Discriminators usually become overconfident with their classification leading to overfitting (memorizing training data too aggressively) and to poor gradient flow for the Generator.

Label Smoothing in GANs is a regularization technique where the **real labels (1s) and fake labels (0s) are slightly modified**.

How It Works?

- We use **softened labels**, such as:
    - **Real images → Randomly between 0.8 and 1.0**
    - **Fake images → Randomly between 0.0 and 0.2**

Batch Normalization in GANs

Batch normalization normalizes the output of a previous layer by subtracting the batch mean and dividing by the batch standard deviation. It's called "batch" normalization because, during training, we normalize each layer's inputs by using the mean and standard deviation (or variance) of the values in the current batch. These are sometimes called the batch statistics.

Batch Normalization (BN) is **crucial** in CNN-based GANs because it stabilizes training, prevents mode collapse, and helps the Generator produce higher-quality outputs.

Why?

1. **Stabilizes Training** → GANs are notoriously unstable; BN normalizes activations
2. **Improves Gradient Flow** → Reduces issues like vanishing/exploding gradients.
3. **Encourages Diverse Outputs** → Prevents mode collapse by reducing dependency on specific feature statistics.
4. **Speeds Up Convergence** → Helps GANs learn faster by keeping activations well-scaled.

Normal Bias term disabled:

When Batch Normalization (BatchNorm) is applied in any type of NN, it is common practice to disable bias terms in the preceding layer. This is because BatchNorm already includes a bias term which simply makes the normal bias term redundant.
Not just DCGANs, in **CNNs, ResNets, Transformers**, and MLPS using BatchNorm, bias is typically disabled in layers preceding BatchNorm.

✅ **Generator** → BN is **commonly used** after each convolutional layer **(except the output layer)** to ensure smooth feature scaling.
🚫 **Discriminator** → BN is **usually avoided** in the Discriminator because it can introduce artifacts and contradict adversarial learning. Instead, techniques like **Spectral Normalization** or **Dropout** are preferred.

Two Times Update Rule

Use a higher learning rate for the discriminator (D) and a lower learning rate for the generator (G). The Two Time-Scale Update Rule (TTUR) is a training technique for GANs, including DCGANs, where the generator and discriminator are updated with different learning rates.

- The **discriminator** is often stronger initially, making it hard for the generator to learn.

TTUR helps stabilize training:

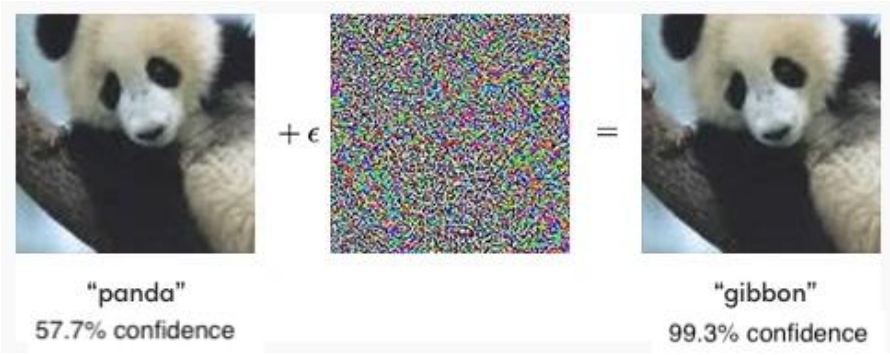
- If the discriminator learns too fast, the generator receives poor gradients.
- If the generator learns too fast, it may exploit weaknesses in the discriminator.
- **TTUR balances this by assigning different learning rates** to G and D.
-

Semi-Supervised Learning

Semi-supervised models are used when you only have a few labeled data points. The motivation for this kind of model is that, we increasingly have a lot of raw data, and the task of labelling data is tedious, time-consuming, and often, sensitive to human error. Semi-supervised models give us a way to learn from a large set of data with only a few labels, and they perform surprisingly well even though the amount of labeled data you have is relatively tiny.

Adversarial Examples

Small perturbations in images can cause a classifier (like AlexNet or a known image classifier) to fail pretty spectacularly! Adversarial examples can be used to "attack" existing models, and has potential security issues. And one example of a perturbation that causes misclassification can be seen below.



Adding a small amount of noise to an image of a panda causes a model to misclassify it as a gibbon, which is a kind of ape. One of the interesting parts of this is the model's confidence. With this noise it is 99.3% confident that this is an image of a gibbon, when we can pretty clearly see that it is a panda!