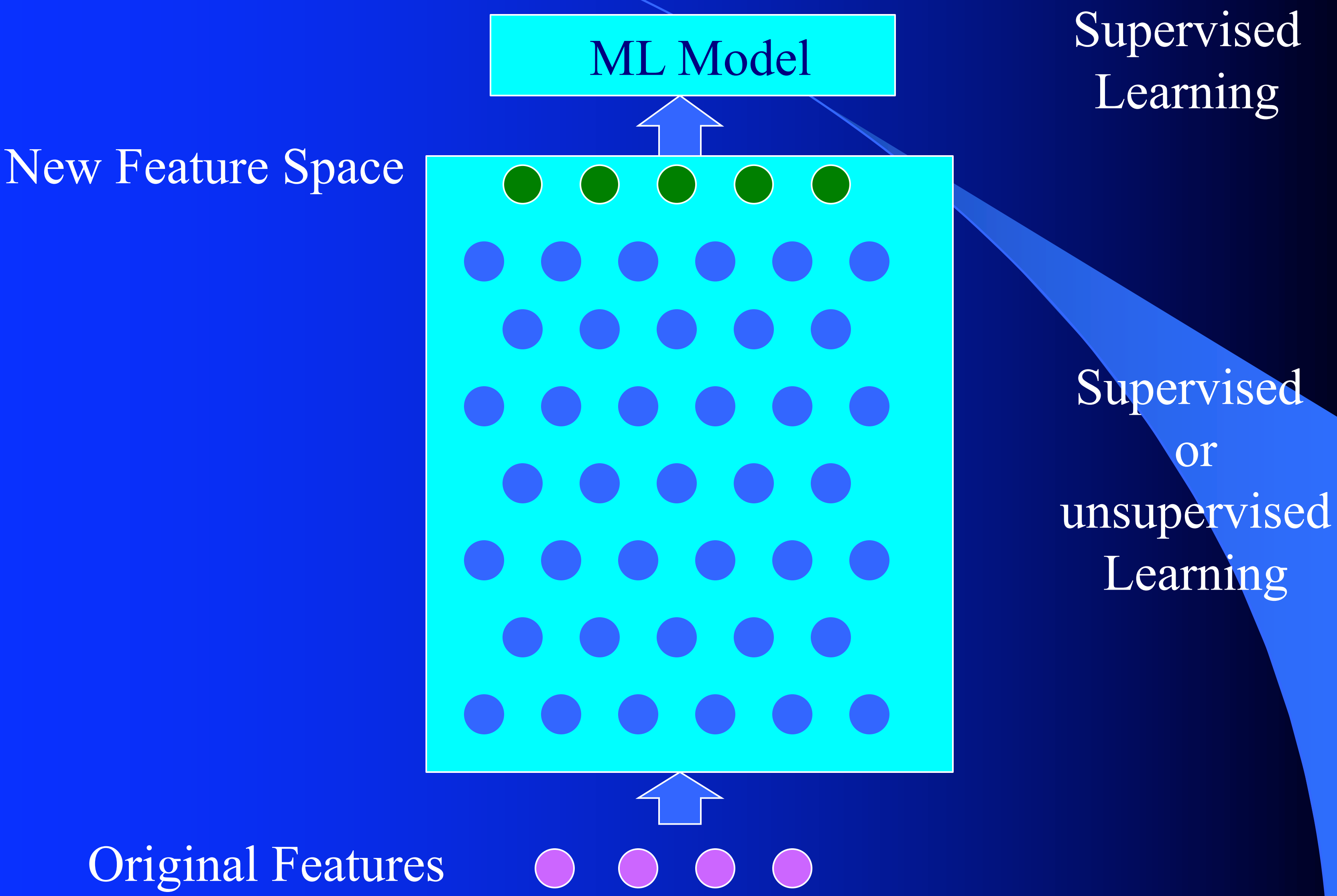
# Deep Learning

- Basic Philosophy
- Why Deep Learning
- Deep Backpropagation
- CNNs – Convolutional Neural Networks
- First "Deep" Nets - Unsupervised Pre-Training Networks
- Deep Supervised Networks with managed gradient approaches
  - Learning tricks, Dropout, Batch normalization, ResNets, etc.
- GANs
- Recurrent Networks - LSTM, GRU
- Deep Reinforcement Learning
- Transformers - GPT

## Deep Learning Overview

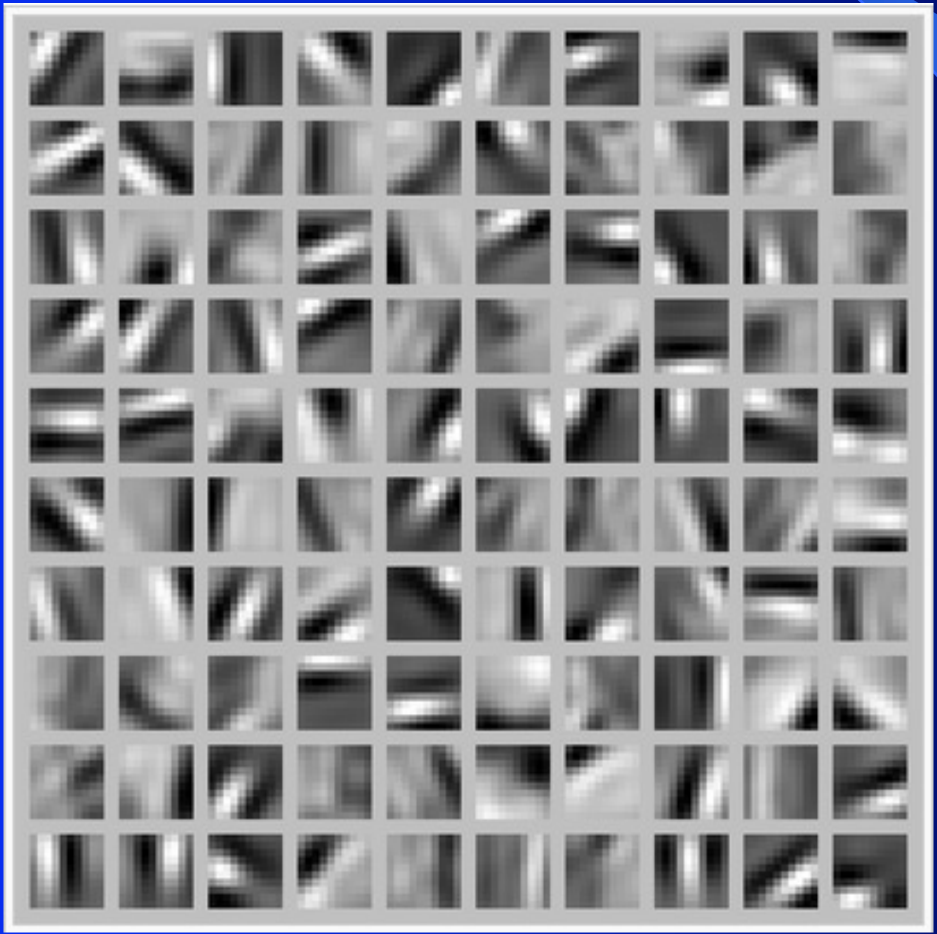
- Train neural networks with many layers (vs. shallow nets with just a couple of layers)
- Multiple layers work to build an improved feature space
  - First layer learns 1st order features (e.g. edges…)
  - 2nd layer learns higher order features (combinations of first layer features, combinations of edges, etc.)
  - Some models learn in an unsupervised mode and discover general features of the input space – serving multiple tasks related to the unsupervised instances (image recognition, etc.)
  - Final layer of transformed features are fed into supervised layer(s)

# Deep Net Feature Transformation



## Deep Learning Tasks

- Images Example: view of a learned vision feature layer (Basis)
- Each square in the figure shows the input image that maximally activates one of  the 100 units



## Why Deep Learning

- Biological Plausibility – e.g. Visual Cortex
- Hastad proof - Problems which can be represented with a polynomial number of nodes with *k* layers, may require an exponential number of nodes with *k*-1 layers (e.g. parity)
- Highly varying functions can be efficiently represented with deep architectures
  - Less weights/parameters to update than a less efficient shallow representation
- Sub-features created in deep architecture can potentially be shared between multiple tasks
  - Type of Transfer/Multi-task learning

## Deep Neural Networks Sketch

- More than just MLPs, but…
- Rumelhart 1986 – great early success
- Interest subsides a bit in the late 90's as other models are introduced – SVMs, Graphical models, etc. – each "wave" ...
- Convolutional Neural Nets –LeCun 1989-> Image, speech, etc.
- Deep Belief nets (Hinton) and Stacked auto-encoders (Bengio) – 2006 – Unsupervised pre-training followed by supervised. Good feature extractors.
- 2012 -> Initial successes with supervised approaches which overcome vanishing gradient, etc., and are more generally applicable. Current explosion, but don't drop the other tools in your kit!
  - Stay the course

# Early Work

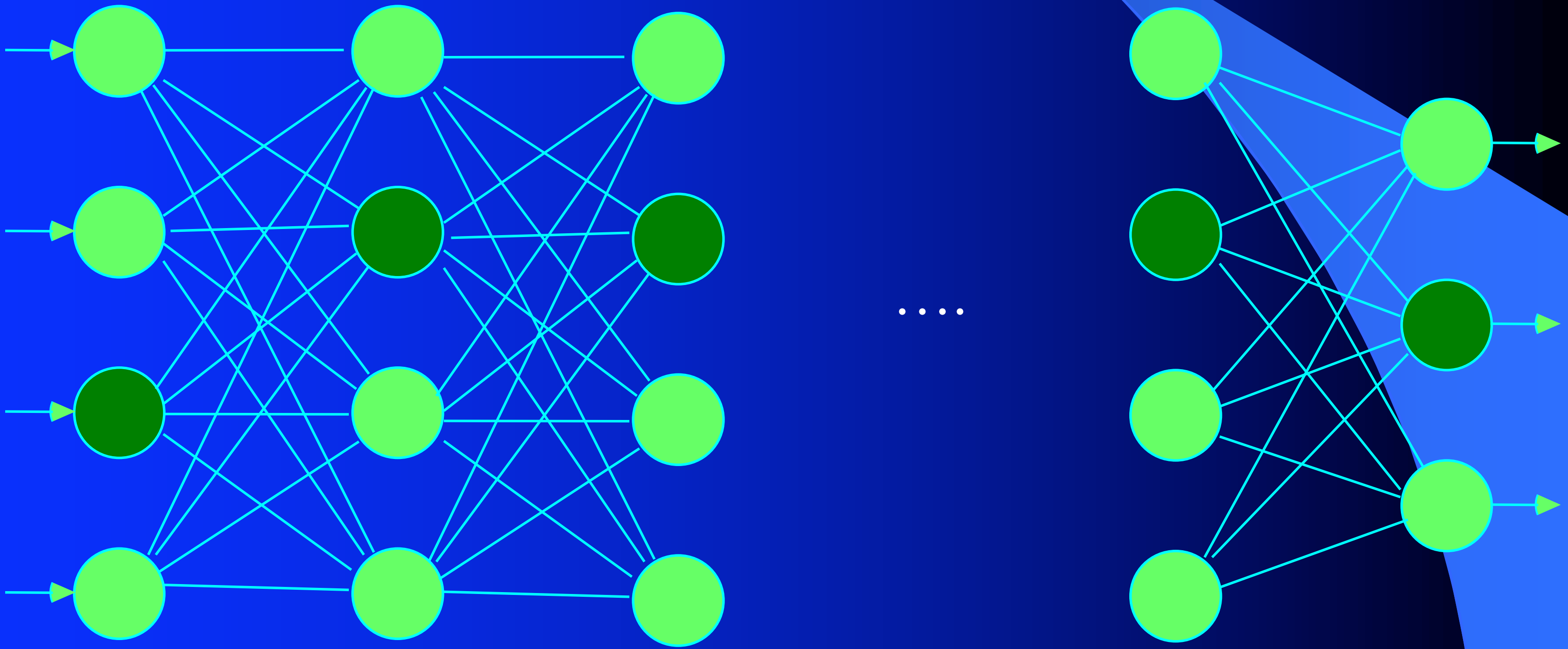
- Fukushima (1980) – Neo-Cognitron
- LeCun (1989) – Convolutional Neural Networks (CNN)
  - Miminal interest at the time before other critical advances
- Many layered MLP with backpropagation
  - Tried early but without much success
    - Very slow
    - Vanishing gradient
  - More recent work demonstrated significant accuracy improvements by "patiently" training deeper MLPs with BP using fast machines (GPUs)
    - More general learning!
    - Much improved since 2012 with some important extensions to the original MLP/BP approach

# Vanishing/Unstable Gradient

- Vanishing Gradient – error attenuates as it propagates to earlier layers – $t$ - $z$ < 1, $f'(net)$, scaled by small initial weights
- Leads to very slow training (especially at early layers when top layers saturate and have small $f'(net)$)
- Exacerbated since top couple layers can usually learn any task "pretty well" and thus the error to earlier layers drops quickly as the top layers "mostly" solve the task
- if lower layers never get the opportunity to use their capacity to improve results, they can just be stuck with their initial random feature mapping
- Need a way for early layers to do effective work
- Instability of gradient in deep networks: Vanishing *or* exploding gradient
  - Product of many terms, which unless “balanced” just right, is unstable
  - Either early or late layers stuck while “opposite” layers are learning

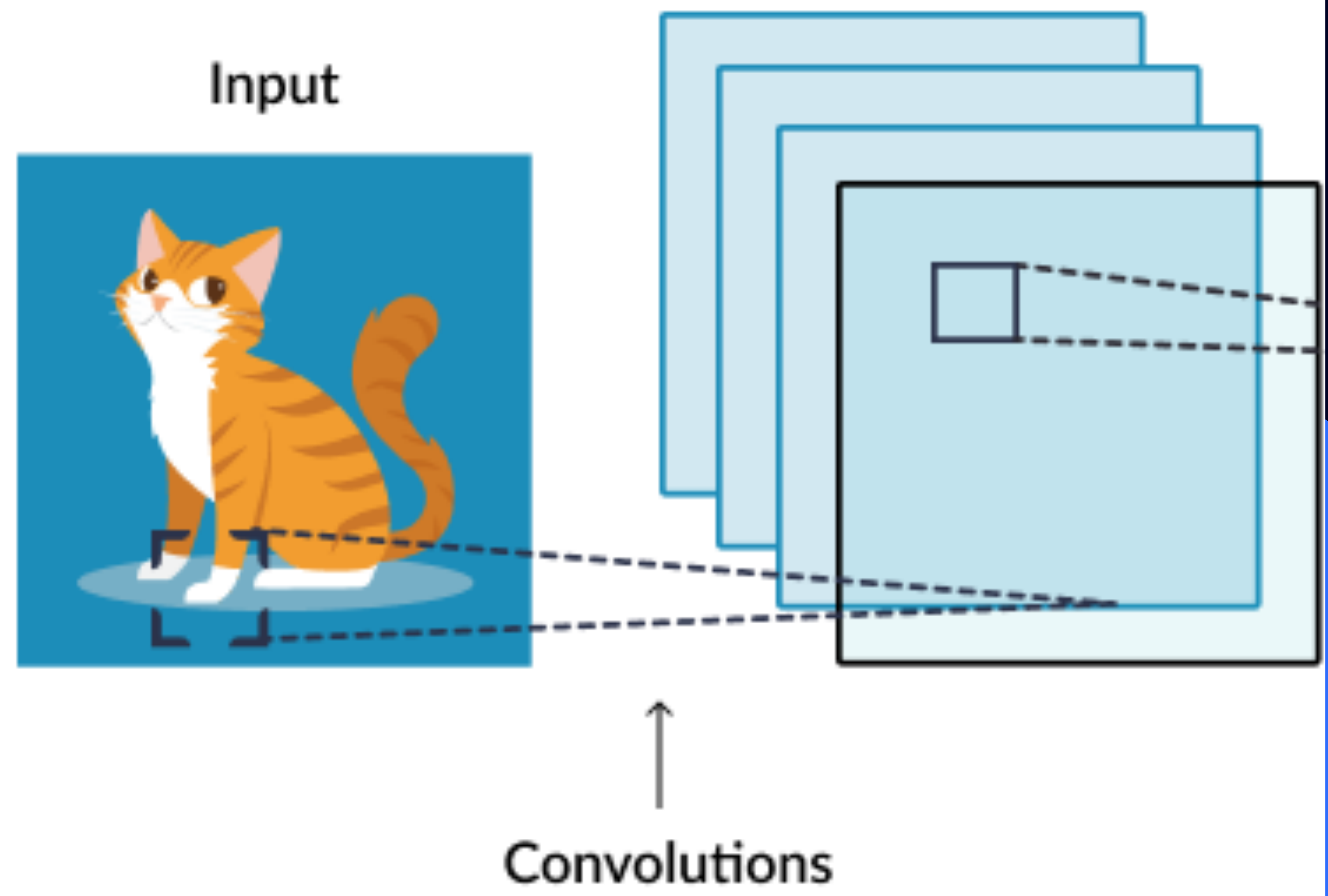
## Vanishing/Exploding Gradient

• Error attenuation, long patient training with GPUs, etc
• Recent algorithmic improvements - Rectified Linear Units, better weight initialization, normalization between layers, residual deep learning, etc. – 1000’s of layers being effectively trained, will discuss later

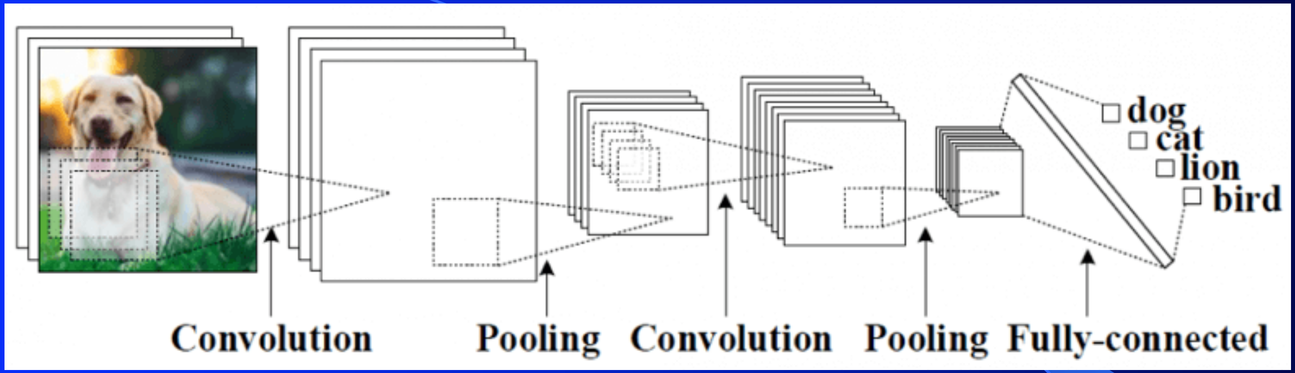


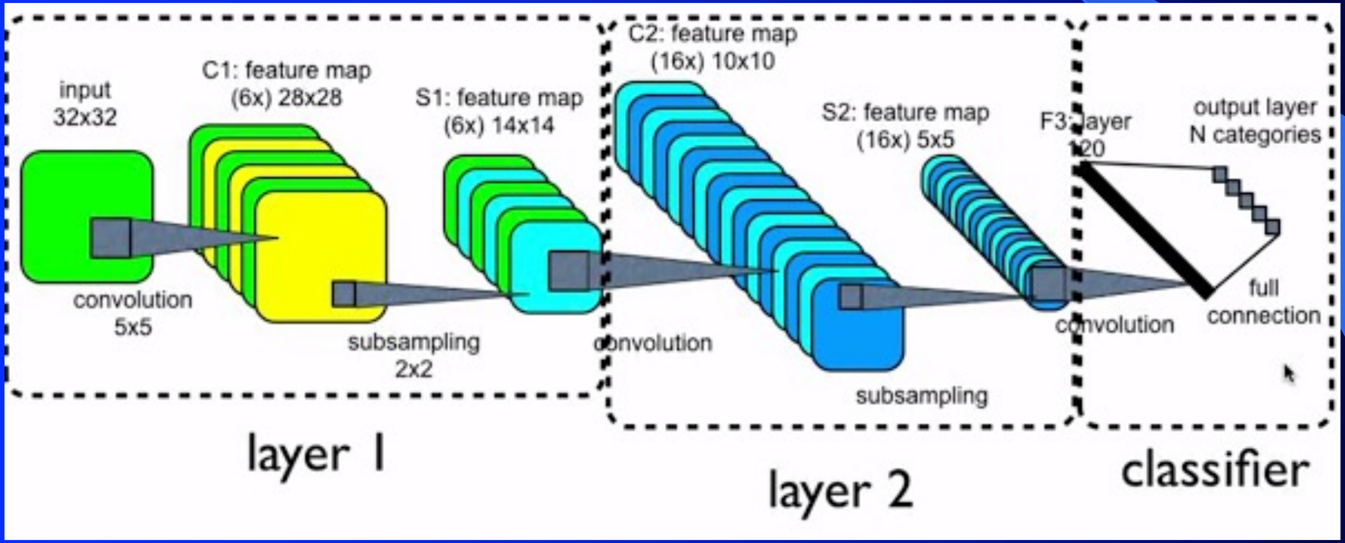
# Convolutional Neural Networks

- "Niche" networks built specifically for problems with low dimensional (e.g. 2-*d*) grid-like local structure – e.g. Images
  - Vision, Character recognition, Speech, Games, Images - where neighboring features have high correlations (pixels, words, etc.), while distant features (pixels, words) are less correlated
    - Typically just uses raw features (e.g. pixels) with no preprocessing
  - Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part
  - While standard NN nodes take input from all nodes in the previous layer, CNNs enforce that a node receives only a small set of features which are spatially or temporally close to each other called *receptive fields* from one layer to the next (e.g. 3x3, 5x5), thus enabling ability to handle local 2-D structure.
    - Can find edges, corners, endpoints, etc.
    - Good for problems with local 2-D structure, but lousy for general learning with abstract features having no prescribed feature ordering or locality


Input
Convolutions

# Convolutional Neural Networks



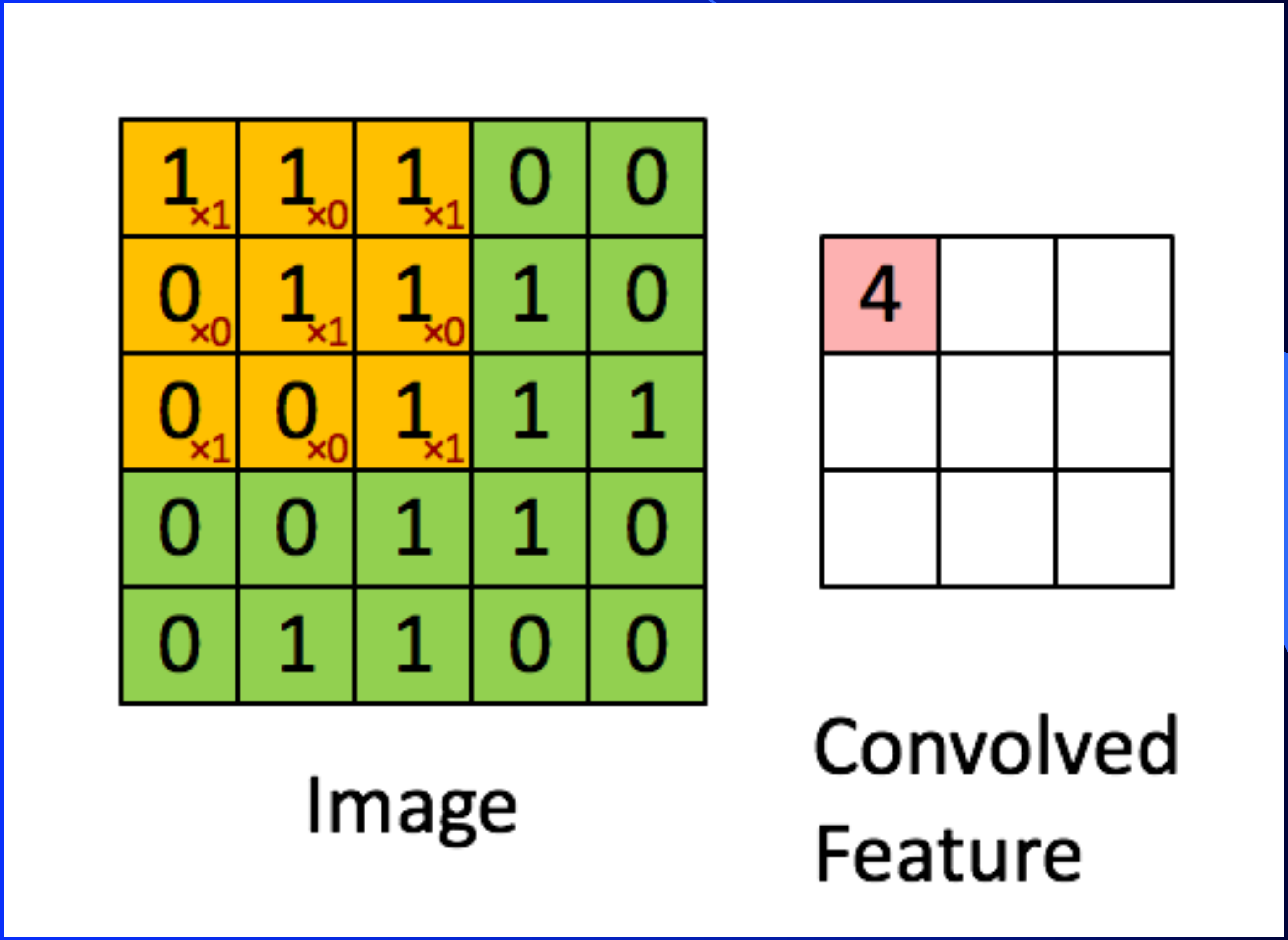


- Big Picture: Each feature map learns a different feature. Each node in feature map has same translated receptive field (and weights)
- Brute force search to see if and where certain features exist
- Pooling/sub-sampling – Does the feature exist in a general area
- Final standard supervised layer with improved feature space

# Convolutions

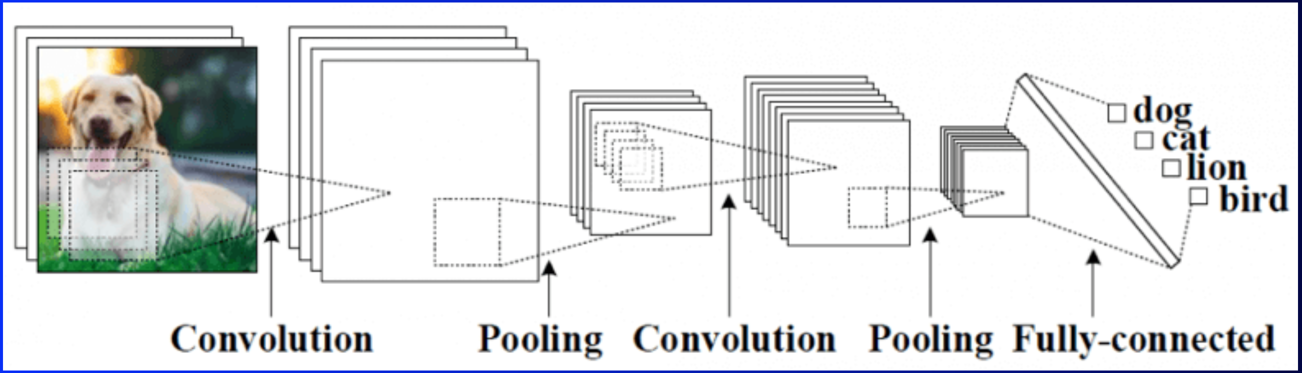
- Typical MLPs have a connection from every node in the previous layer, and the net value for a node is the scalar dot product of the inputs and weights (e.g. matrix multiply). Convolutional nets are somewhat different:
  - Nodes still do a scalar dot product from the previous layer, but with only a small portion (receptive field) of the nodes in the previous layer – *Sparse representation*
  - Every node in a feature map has the exact same weight values from the preceding layer – *Shared parameters*, tied weights, a LOT less unique weight values. Regularization by having same weights looking at lots of input positions (Convolutional filter – same weights)
  - Each node has its shared weight convolution computed on a receptive field slightly shifted, from that of its neighbor, in the previous layer – *Translation invariance*.
  - Each node's convolution scalar (*net* value) is then passed through a non-linear activation function (ReLU, tanh, sigmoid, etc.)

# Convolution Example

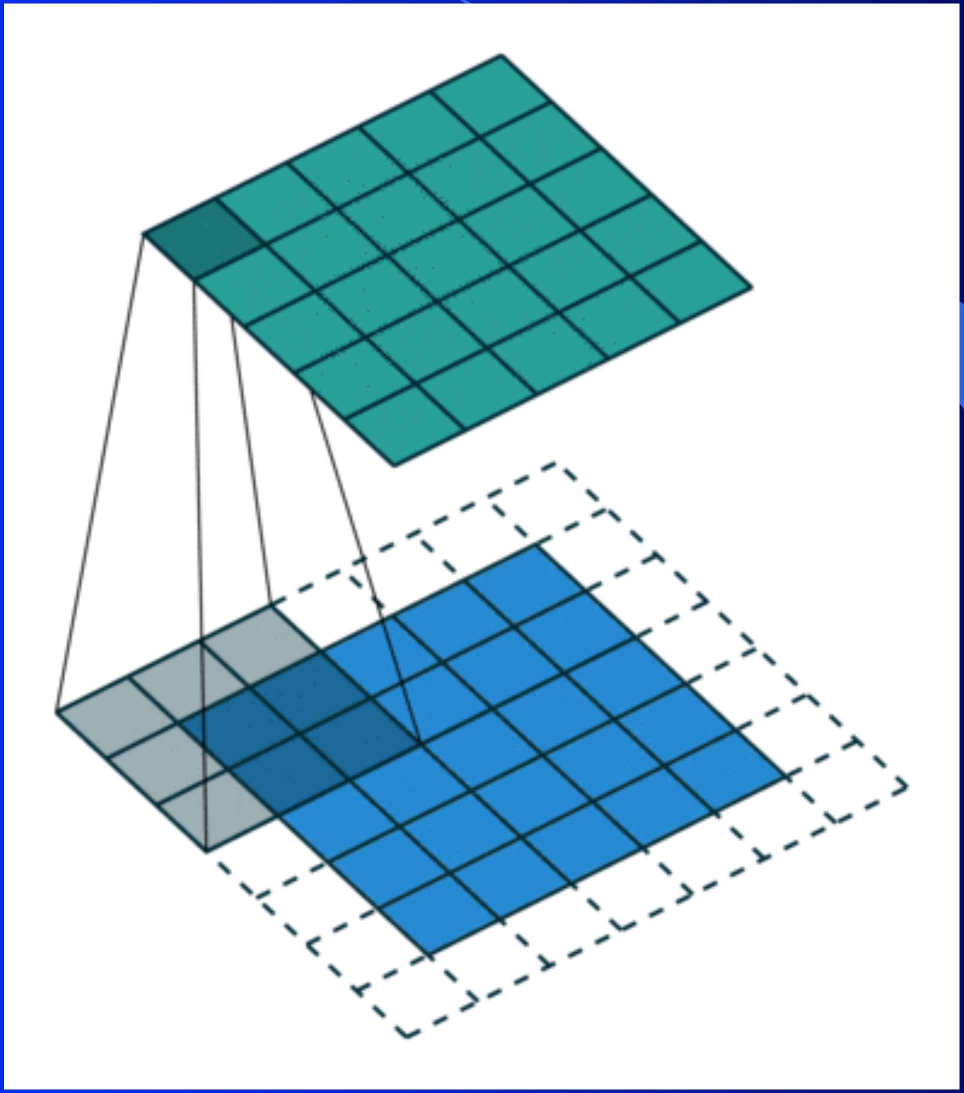


# CNN

- The 2-*d* planes of nodes (or their outputs) at subsequent layers in a CNN are called *feature maps*
- Thus each *feature map* searches the full previous layer to see if, where, and how often its feature occurs (precise position less critical)
  - The output will be high at each node in the map corresponding to a receptive field where the feature occurs (e.g. edge, curve)
  - Convolution layers search across all feature maps of the previous layer
  - Later layers can concern themselves with higher order combinations of features and rough relative positions – e.g. eyes next to each other with nose below

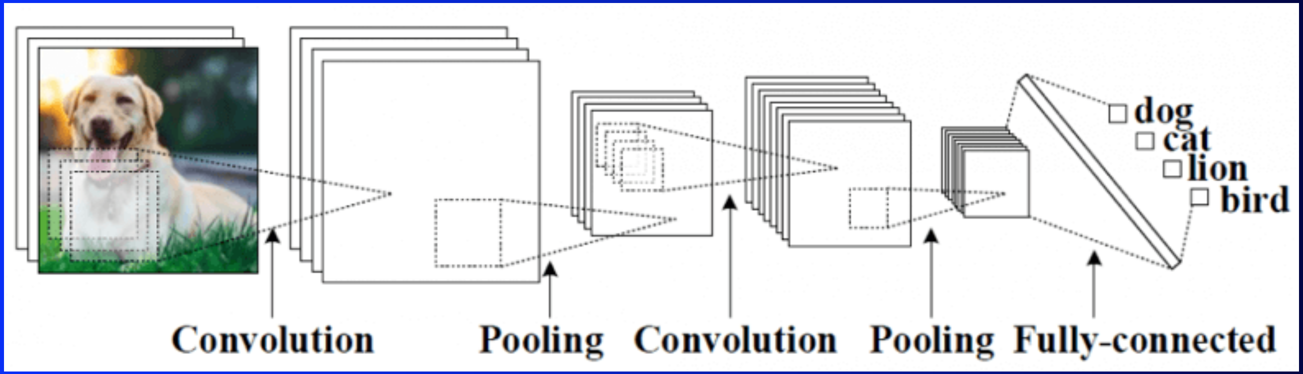


## Convolutional Example
## 0 padding = 1 and stride = 1

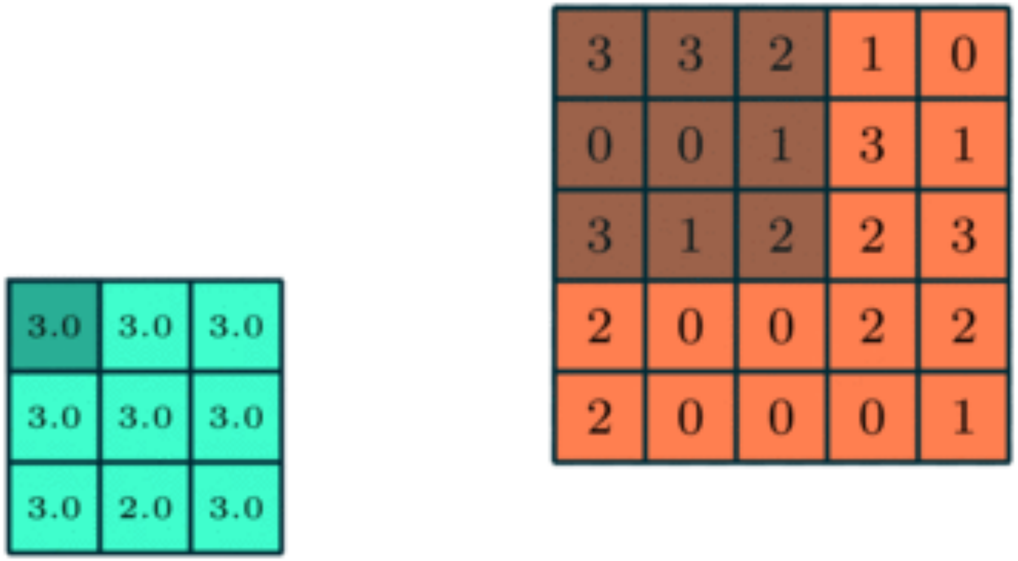


# Sub-Sampling (Pooling)

- Convolution and sub-sampling layers can be interleaved
- Sub/Down-sampling (Pooling) allows the number of features to be diminished, and to pool information
  - Pooling replaces the network output at a certain point with a summary statistic of nearby outputs
  - Max-Pooling common (Just as long as the feature is there, take the max, as exact position is not that critical), also averaging, etc.
  - Pooling smooths the data and reduces spatial resolution and thus naturally decreases importance of exactly where a feature was found, just keeping the rough location – translation invariance
  - 2x2 pooling would do 4:1 compression, 3x3 9:1, etc.
  - Convolution may increase number of feature maps per layer, pooling keeps same number of *reduced* maps (one-to-one correspondence of convolution map to pooled map) as the previous layer
  - Less pooling layers common in recent architectures to allow more depth



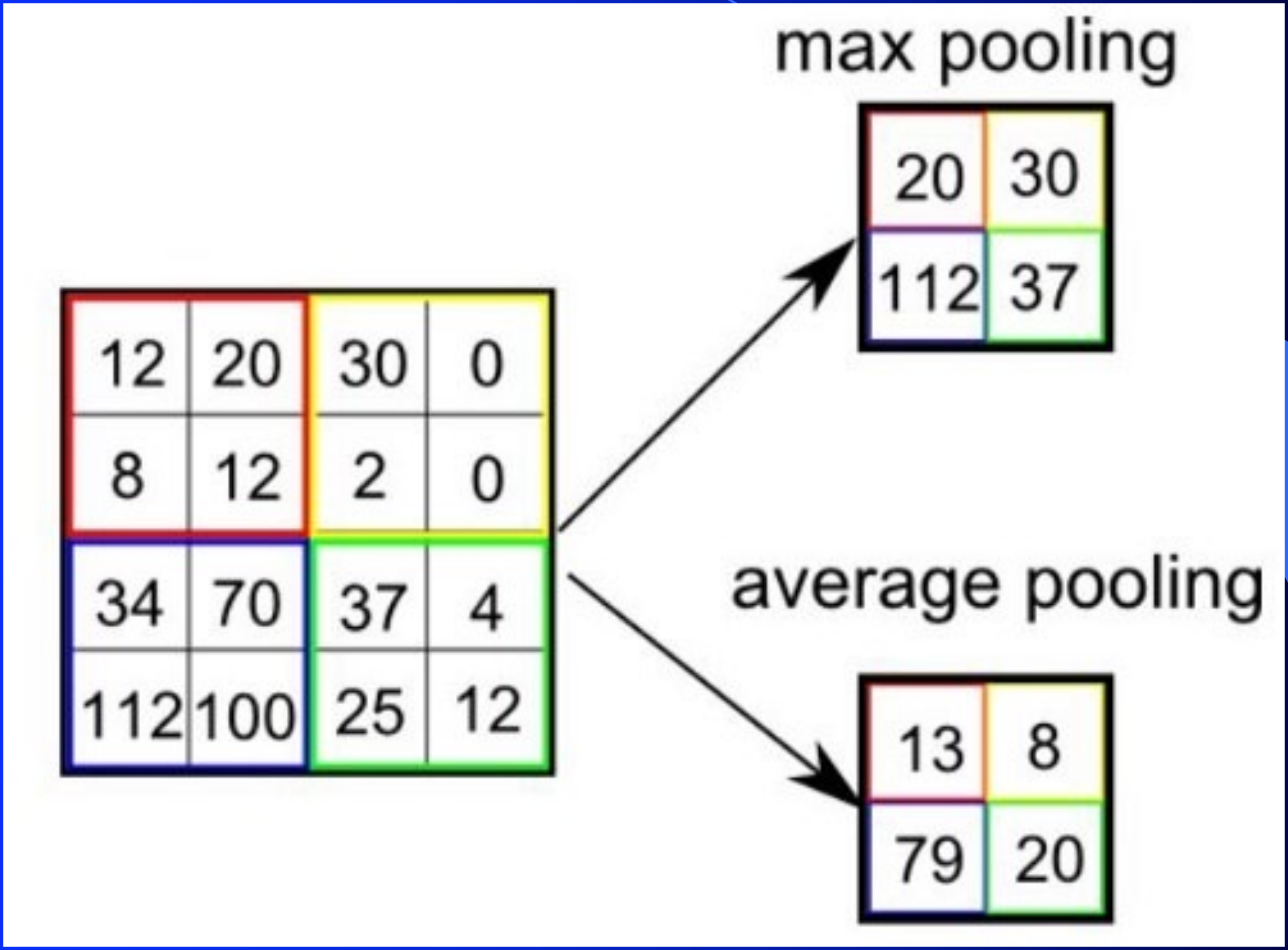
# Max Pooling Example (Sum or Average sometimes used)



| | | |
|---|---|---|
| 3.0 | 3.0 | 3.0 |
| 3.0 | 3.0 | 3.0 |
| 3.0 | 2.0 | 3.0 |

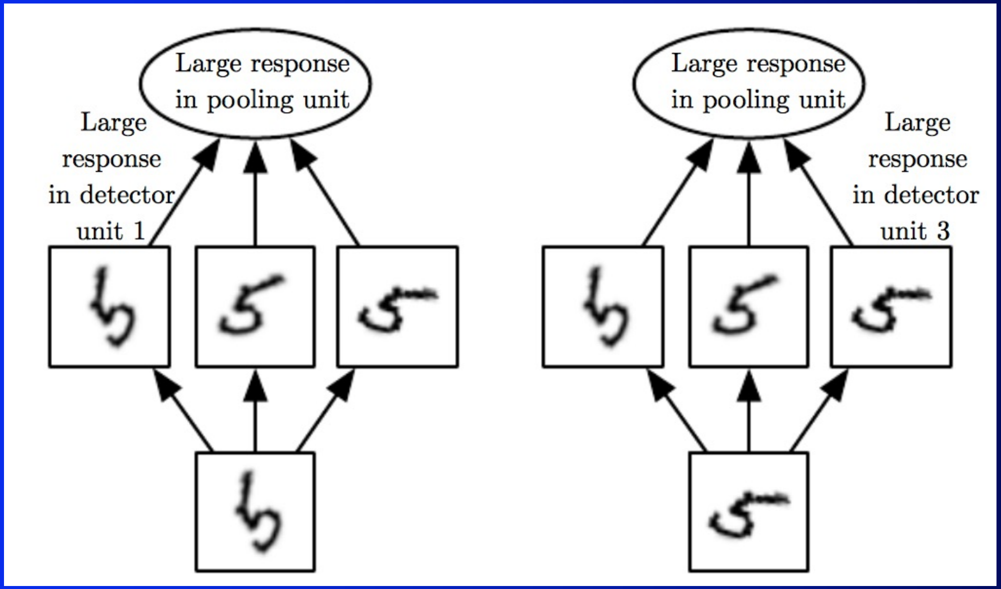
| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |

# Max and Average Pooling Non overlapping: Stride = 2



## Pooling (cont.)

- Common layers are convolution, non-linearity, then pool (repeat)
- Note that pooling/down-sampling decreases map sizes (unless pool stride = 1, highly overlapped), making real deep nets more difficult. Pooling is sometimes used only after multiple convolved layers and sometimes not at all.
- At later layers pooling can make network invariant to more than just translation – *learned invariances*


Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

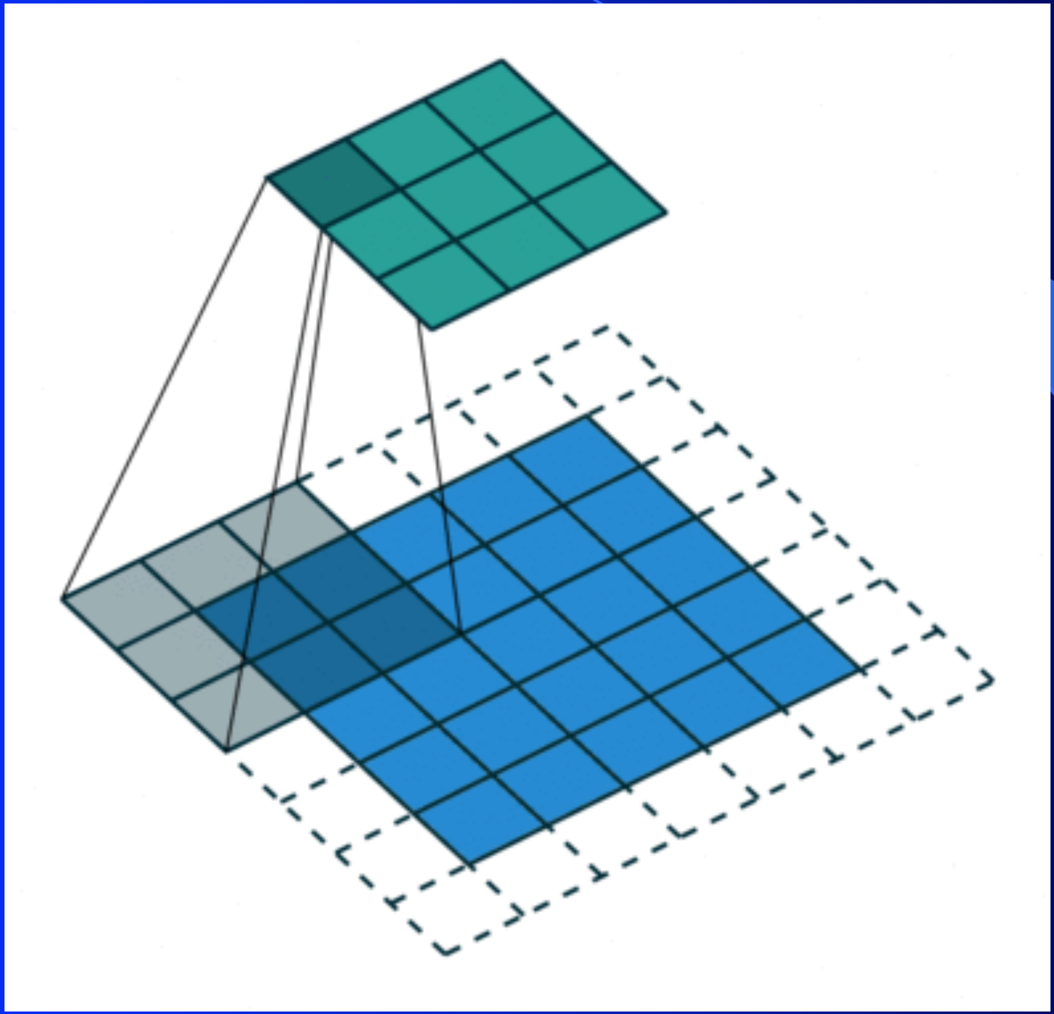
## CNN Training

- Trained with BP, with weight tying in each feature map
  - Randomized initial weights throughout entire network, standard training on final fully connected network
- Feature Maps
  - Each feature map has one weight for each input and one bias
  - Thus a feature map with a 5x5 receptive field (filter) would have a total of 26 weights, which are the same coming into each node of the feature map
  - If a convolution layer had 10 feature maps with 5x5 receptive fields, then a total of 260 unique weights would be trained in that layer (much less than an arbitrary deep net layer without sharing)
  - Calculate weight updates independently into each node but don't update yet. Average the weight updates over the tied weights and update each the same.
- Sub-Sampling (Pooling) Layer
  - All elements of receptive field max'd, averaged, summed, etc.  No trainable weights necessary
- While all weights are trained, the structure of the CNN is currently usually hand crafted with trial and error, including number of total layers, number of receptive fields, size of receptive fields, size of sub-sampling (pooling) fields, etc.

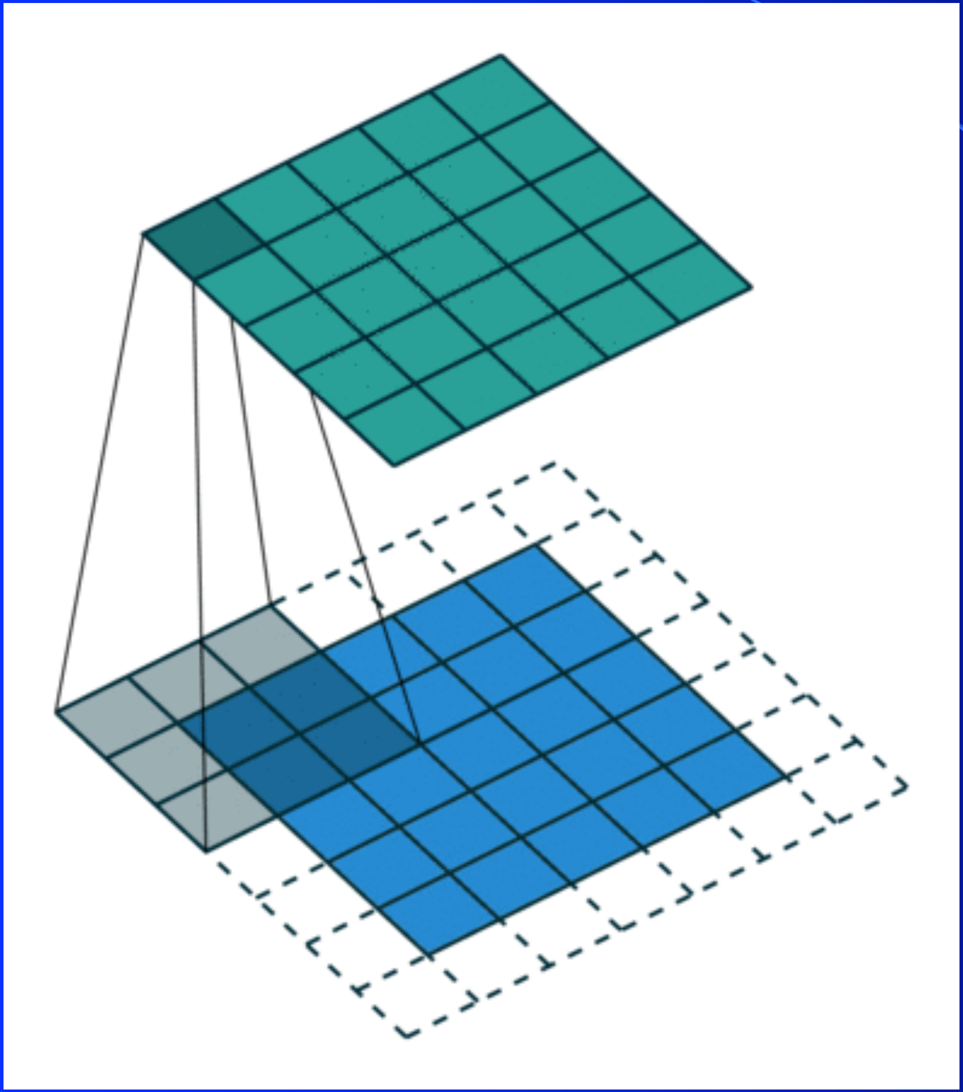
## CNN Hyperparameters

- Structure itself, number of layers, size of filters, number of feature maps in convolution layers, connectivity between layers, activation functions, final supervised layers, etc.
- Drop-out often used in final fully connected layers for overfit avoidance – less critical in convolution/pooling layers which already regularize due to weight sharing
- As is, the feature map would always decrease in volume which is not usually desirable - *Zero-padding* avoids this and lets us maintain up to the same volume
  - Would shrink fast for large kernel/filter sizes and would limit the depth (number of layers) in the network, smaller kernels common (3x3)
  - Also allows the different filter sizes to fit arbitrary map widths
- *Stride* – Don't have to test every location for the feature (i.e. stride = 1), could sample more coarsely
  - Another option (besides pooling) for down-sampling

## Zero-Pad = 1, Stride = 2



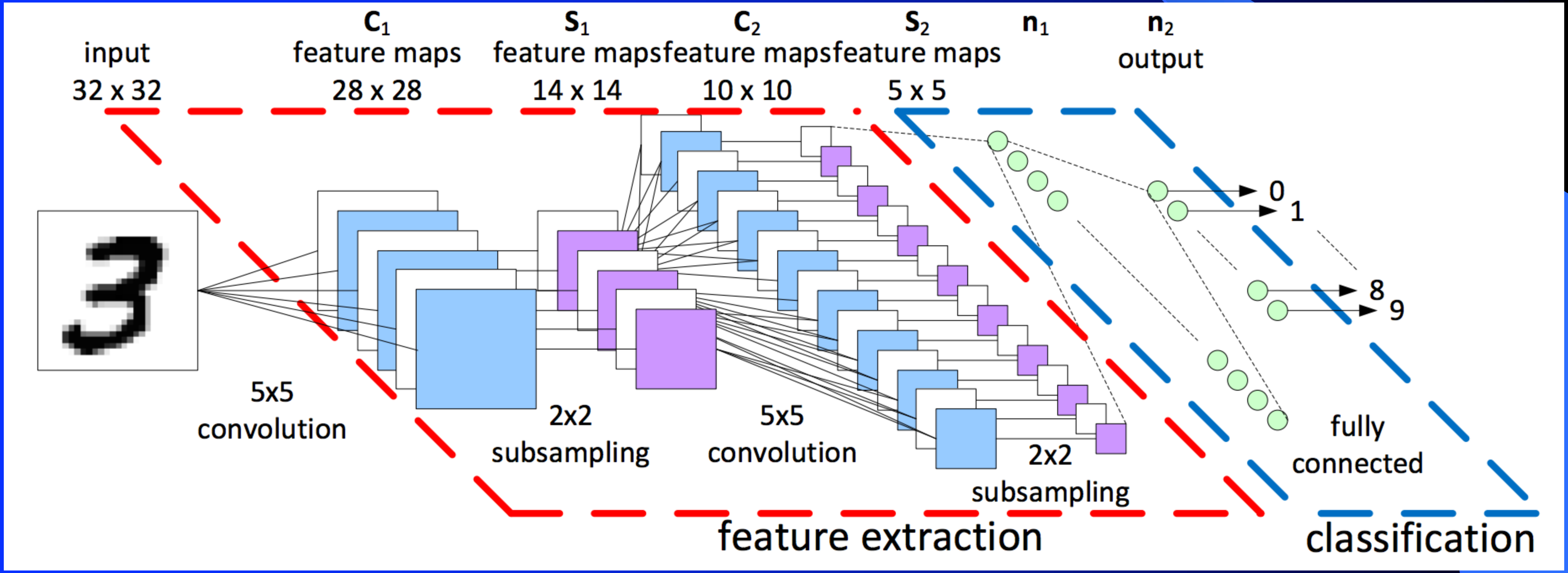
# Convolutional Example
## Zero pad = 1 and stride = 1



Note that the next layer (top) is still 5x5 due to the zero padding. What size would the next layer have been without zero padding?

## Example – CNN MNIST Classification

- Roughly based on LeCun's original model. To help it all sink in:
- How many connections and trainable weights at each layer?
  - MNIST data input is 28x28 pixels
  - Stride = 1 for convolutions, and pooling is non-overlapping



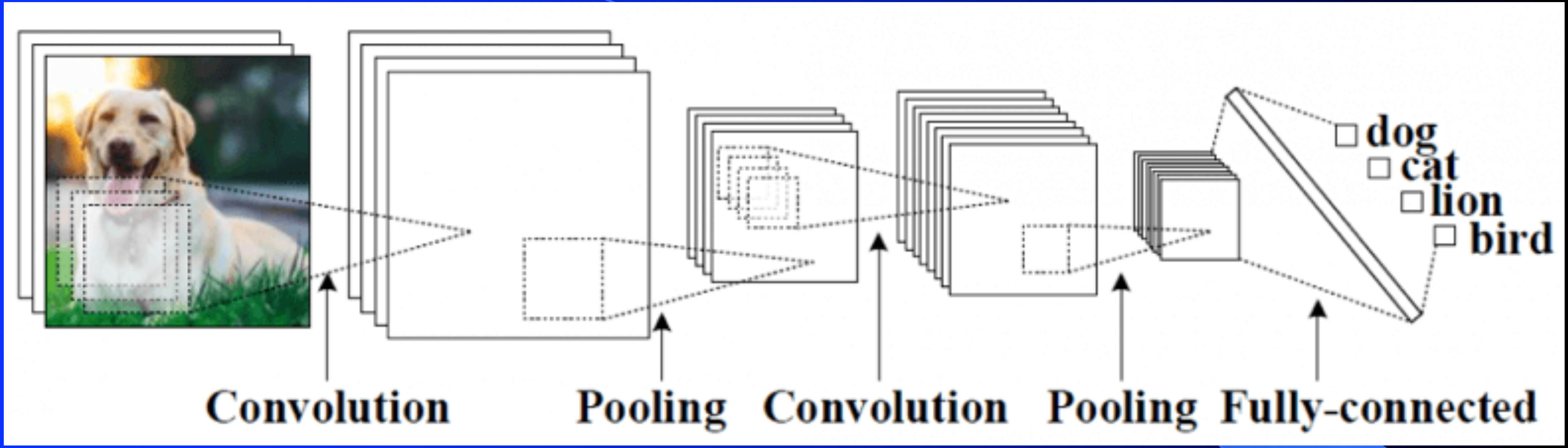
6 maps    16 maps    120 nodes

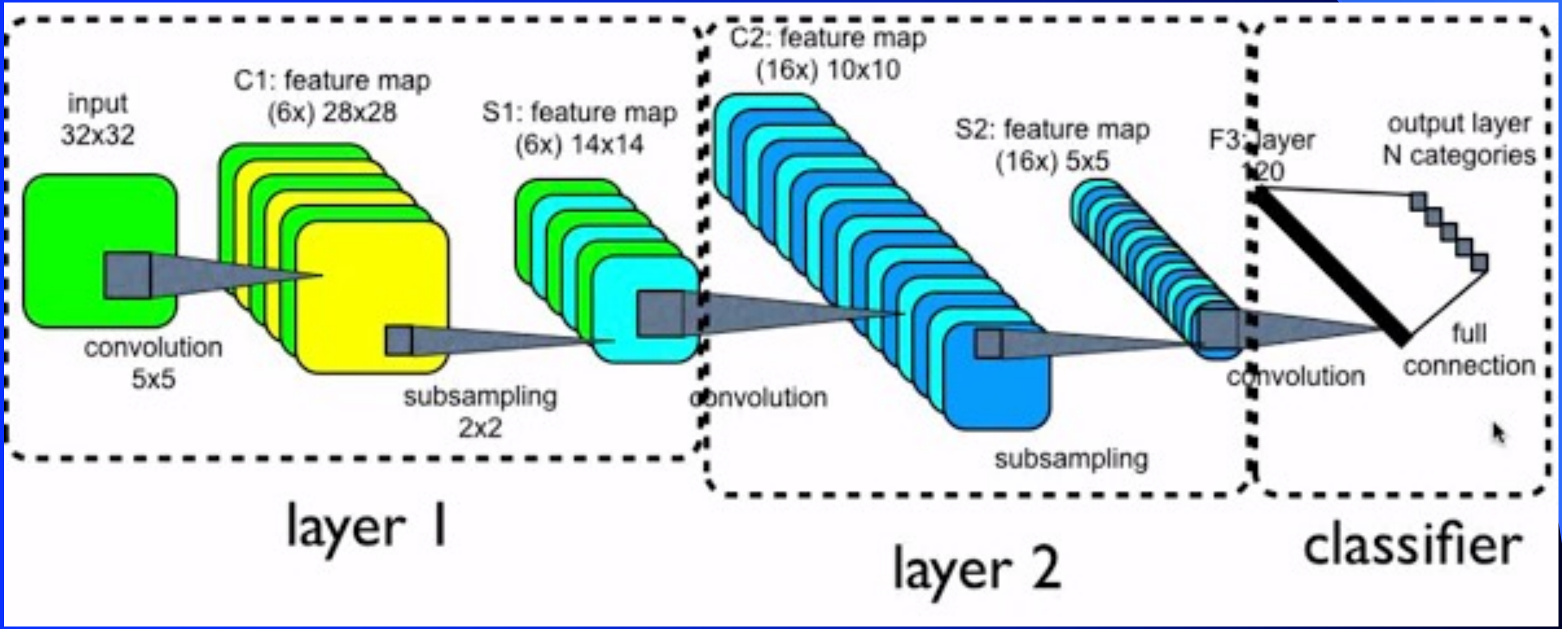
# Basic CNN Example

| Layer | Trainable Weights per layer | Total Connections |
|---|---|---|
| C1 | (25+1)*6 = 156 | 156*28*28 = 122,304 |
| S1 | 0 (LeCun had (1+1)*6 = 12) | 4 (2x2 links) *6*14*14 = 4704 |
| C2 | (5*5*6+1)*16 = 2416 | 2416*10*10 = 241,600 |
| S2 | 0 | 4 (2x2 links) *16*5*5 = 1600 |
| N1 | (5*5*16+1)*120 = 48,120 | Same since fully connected MLP at this point |
| Output | (120+1)*10 = 1210 | Same |

- Why 32x32 to start with? Actual characters never bigger than 28x28. Just padding the edges so for example the top corner node of the feature map can have a pad of two up and left for its feature map (since receptive field is 5x5). Same things happens with 14x14 to 10x10 drop from S1 to C2.
- S1 and S2 non-overlapping and pool (max most common) – We include here the 4 unweighted connections
  - LeCun had a trainable weight and a bias in pool layer followed by a non-linearity, not really necessary and not used these days
- C2: Connects to all preceding maps, but no zero-padding so maps decrease in size
  - LeCun had each map connect to a subset of the preceding maps
- S2: Final number of extracted features to go to the MLP: (5*5)*16 = 400
- 419,538 total connections, with 51,902 trainable parameters 95% of which are in the final MLP. Only 2572 trainable weights in CNN.

# Convolutional Neural Networks





input 32x32
C1: feature map (6x) 28x28
S1: feature map (6x) 14x14
C2: feature map (16x) 10x10
S2: feature map (16x) 5x5
F3: layer 120
output layer N categories
convolution 5x5
subsampling 2x2
convolution
subsampling
convolution
full connection
layer 1
layer 2
classifier

## CNN Homework

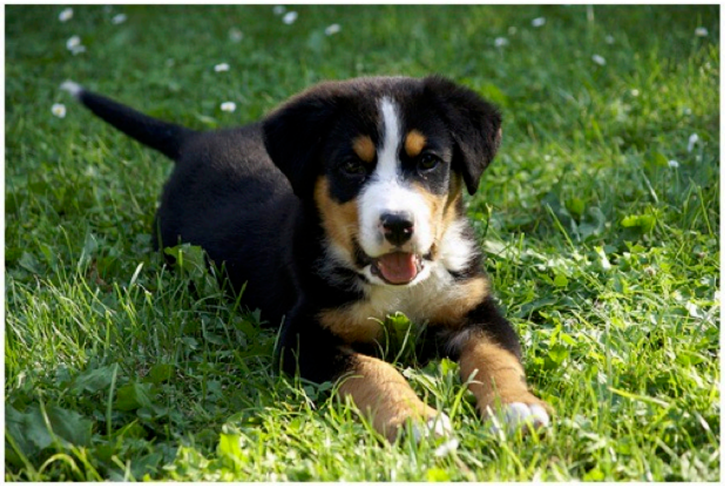
- Assume a traditional CNN with an initial input image of 16x16, followed by a convolutional layer with 8 feature maps using 5x5 receptor fields, followed by a max pooling layer with 2x2 receptive fields, followed by a convolution layer with 10 feature maps using 3x3 receptor fields. Those outputs go straight into (no additional pooling layer) a fully connected MLP with 20 hidden nodes followed by 3 output nodes for 3 possible output classes. Assume no zero-padding and stride=1 for convolution layers, no overlap and no trainable weights for the one pooling layer, and convolutional maps connect to all maps in the previous layer. Sketch the network. For each layer state a) What is the size of the maps in the layer (e.g. the input layer is 16x16), b) how many unique trainable weights are there per layer, and c) total connections in the layer. Show your work and explain your numbers in each case (similar to the previous slide).

## ILSVRC Image net Large Scale Vision Recognition Competition

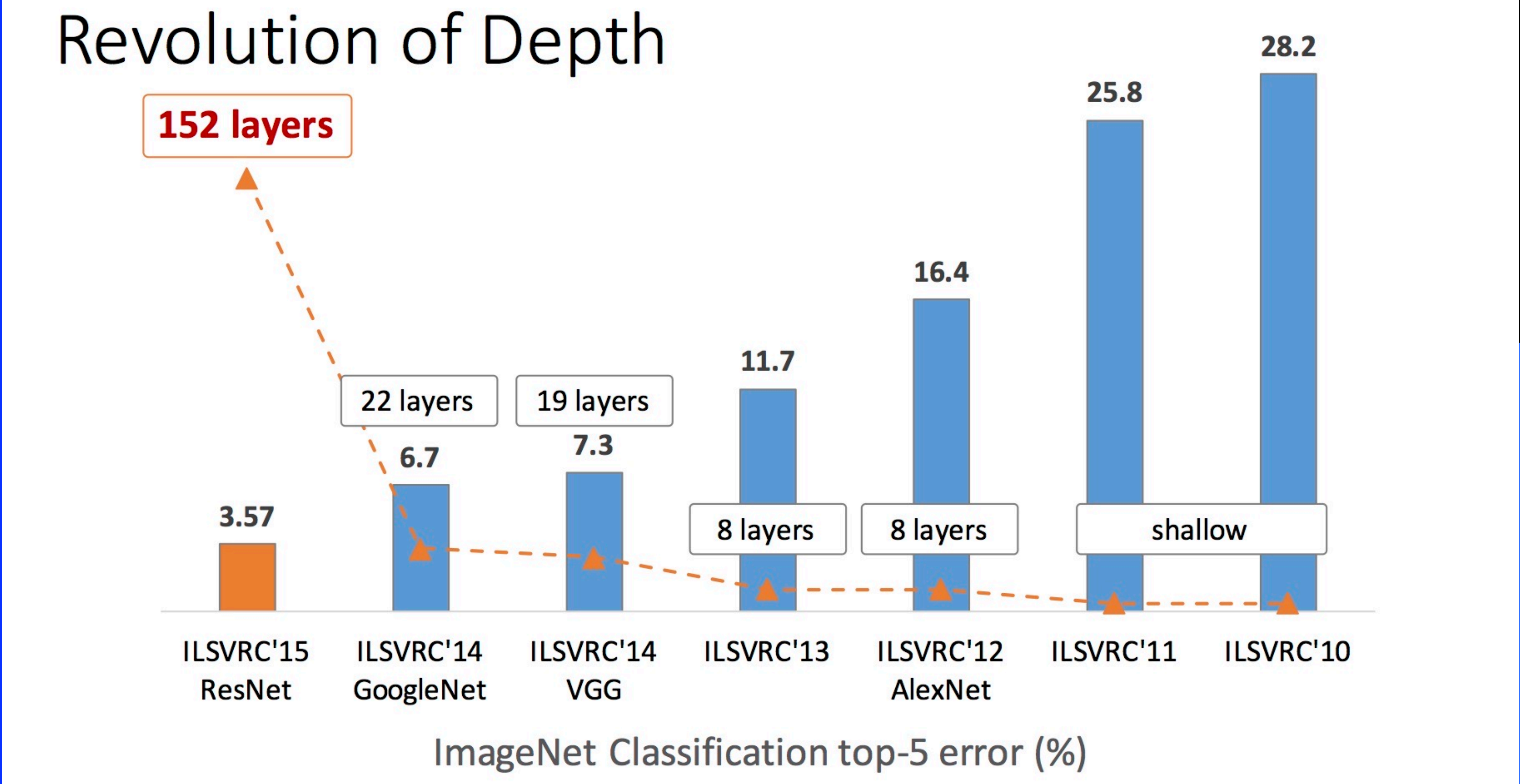
RGB: 224 x 224 x 3 = 150,528 raw real valued features

- Annual competition of image classification at large scale
- 1.2M images in 1K categories
- Classification: make 5 guesses about the image label

EntleBucher

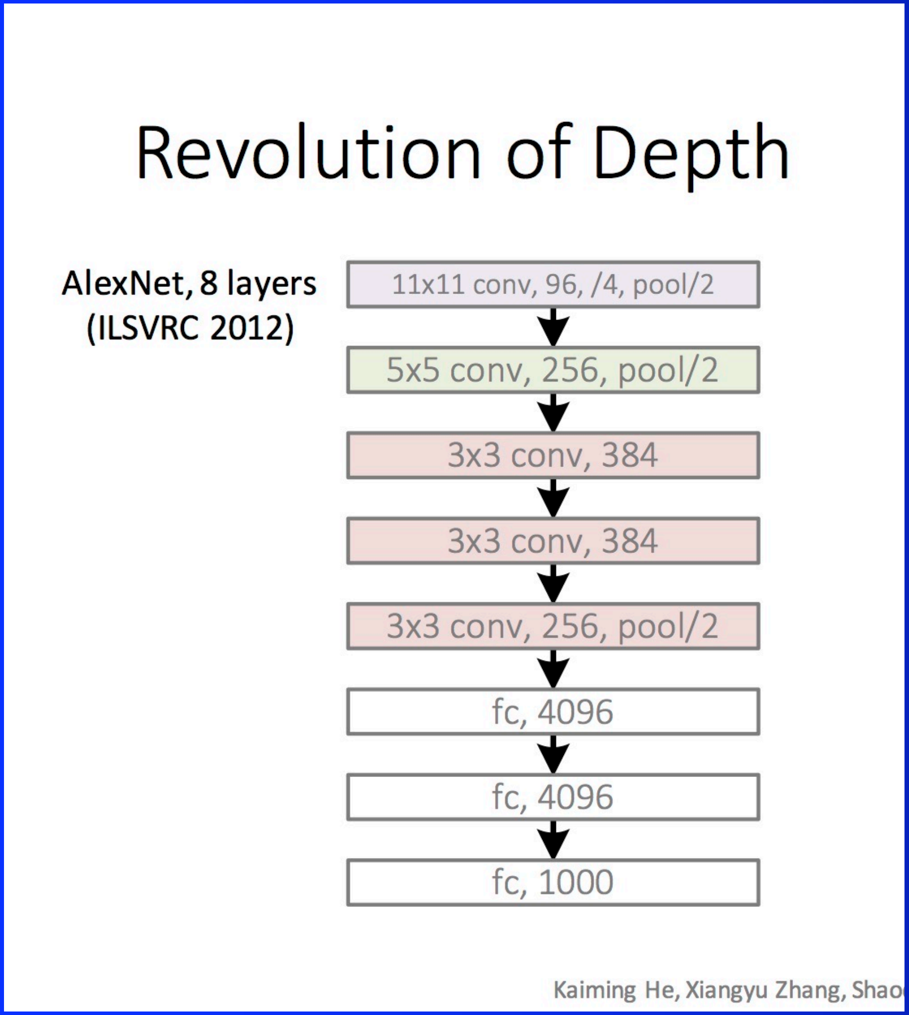

Appenzeller

# Increasing Depth



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR

## Example CNNs Structures ILSVRC winners

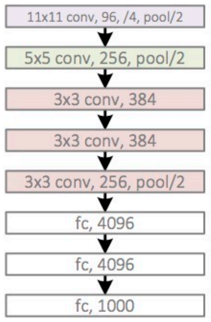


- Note Pooling considered part of the layer
- 96 convolution kernels (maps), then 256, then 384
- Stride of 4 for first convolution kernel, 1 for the rest
- Pooling layers with 3x3 receptive fields and stride of 2 throughout
- Finishes with fully connected (fc) MLP with 2 hidden layers and 1000 output nodes for classes

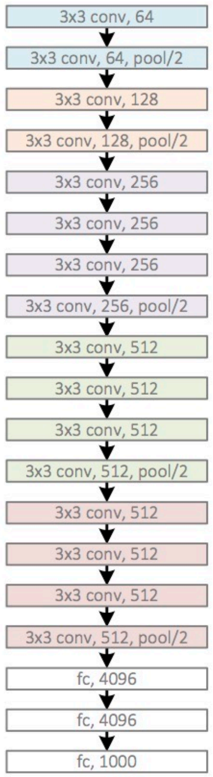
## Example CNNs Structures ILSVRC winners

### Revolution of Depth

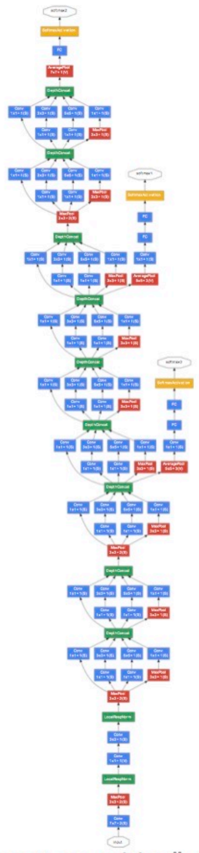
AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.

## CNN Summary

- High accuracy for image applications – Breaking all records and doing it using just using just raw pixel features!
- Special purpose net –Works for images or problems with strong grid-like local spatial/temporal correlation (Speech, games, etc.)
- Once trained on one problem (e.g. vision) could use same net (often tuned) for a new similar problem – general creator of vision features, pre-trained nets
- Unlike traditional nets, handles variable sized inputs
  - Same filters and weights, just convolve across different sized image and dynamically scale size of sub-sampling (pooling or increase strid with convolutions), or # of nodes, to normalize
  - Different sized images, different length speech segments, etc.
- Lots of hand crafting and CV tuning to find the right recipe of receptive fields, layer interconnections, etc.
  - Lots more Hyperparameters than standard nets, since the structures of CNNs are more handcrafted
  - CNNs getting wider and deeper with speed-up techniques (e.g. GPU, ReLU, etc.) and lots of current research, excitement, and success

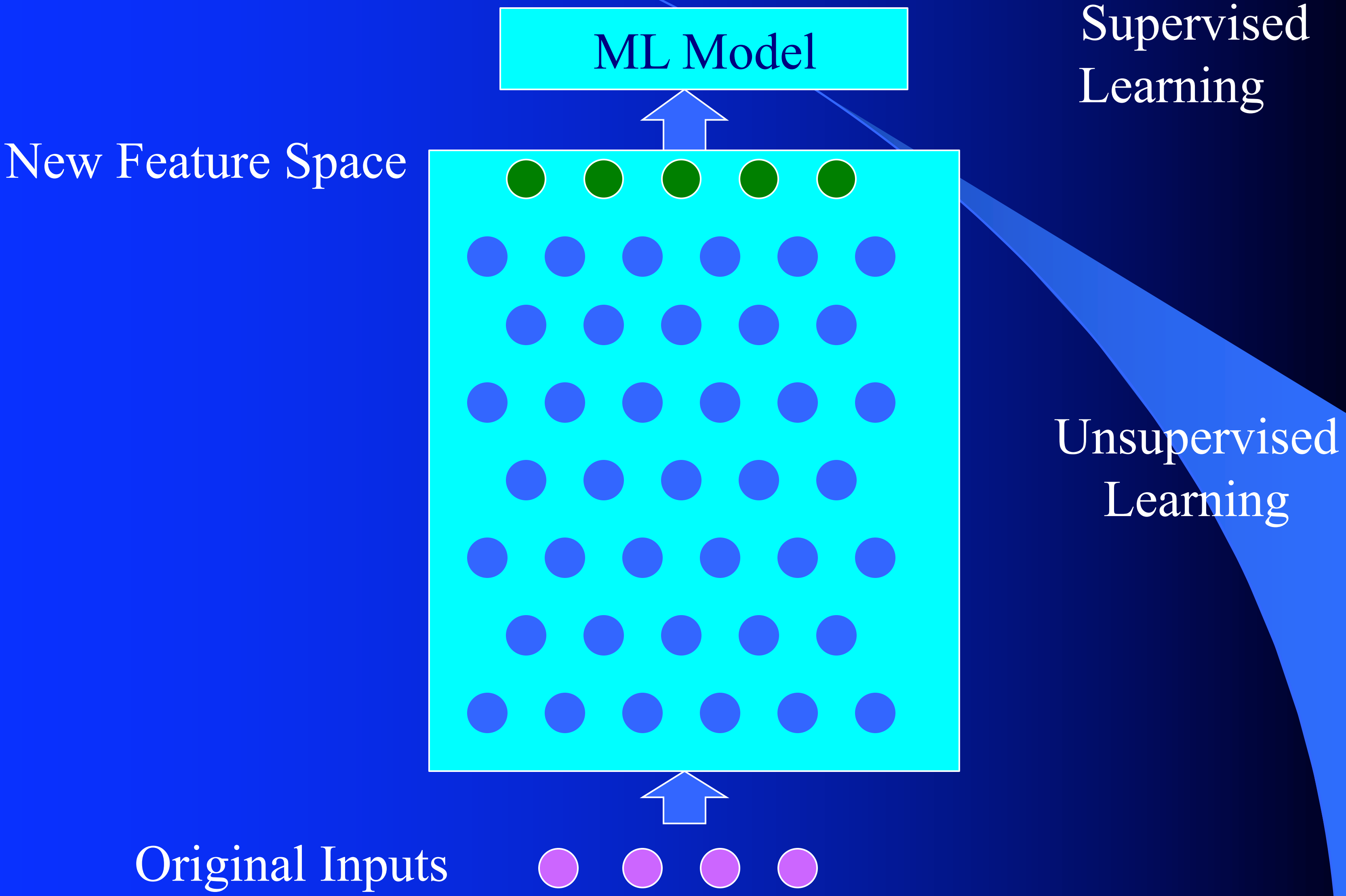
## Unsupervised Pre-Training

- Began the hype of Deep-Learning (2006)
  - Before CNNs were fully recognized for what they could do
  - Less popular at the moment for supervised learning, with recent supervised success, but still at the core of many new research directions
  - Unsupervised Pre-Training uses unsupervised learning in the deep layers to transform the inputs into features that are easier to learn by a final supervised model
  - Unsupervised training between layers can decompose the problem into distributed sub-problems (with higher levels of abstraction) to be further decomposed at subsequent layers
- Often not a lot of labeled data available while there may be lots of unlabeled data. Unsupervised Pre-Training can take advantage of unlabeled data. Can be a huge issue for some tasks.

# Self Taught vs Unsupervised Learning

- When using Unsupervised Learning as a pre-processor to supervised learning you are typically given examples from the same distribution as the later supervised instances will come from
  - Assume the distribution comes from a set containing just examples from a defined set of possible output classes, but the label is not available (e.g. images of car vs trains vs motorcycles)
- In Self-Taught Learning we do not require that the later supervised instances come from the same distribution
  - e.g., Do self-taught learning with any images, even though later you will do supervised learning with just cars, trains and motorcycles.
  - These types of distributions are more readily available than ones which just have the classes of interest (i.e. not labeled as car *or* train *or* motorcycle)
  - However, if distributions are *very* different…
- New tasks share concepts/features from existing data and statistical regularities in the input distribution that many tasks can benefit from
  - Can re-use well-trained nets as starting points for other tasks
  - Note similarities to supervised multi-task and transfer learning
- Both unsupervised and self-taught approaches reasonable in deep learning models

# Deep Net with Greedy Layer Wise Training



## Greedy Layer-Wise Training

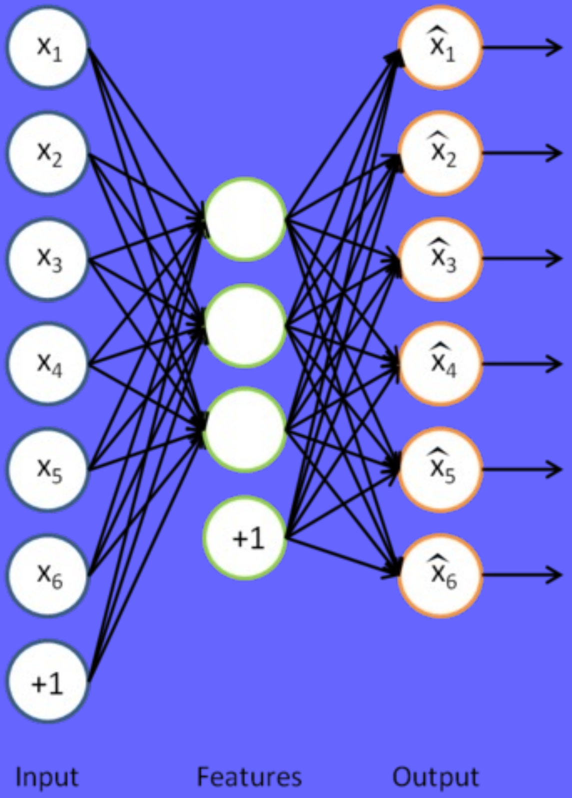
1. Train first layer using your data without the labels (unsupervised)
   - Since there are no targets at this level, labels don't help.  Could also use the more abundant unlabeled data which is not part of the training set
2. Then freeze the first layer parameters and start training the second layer using the output of the first layer as the unsupervised input to the second layer
3. Repeat this for as many layers as desired
   - This builds the set of robust features
4. Use the outputs of the final layer as inputs to a supervised layer/model and train the last supervised layer(s) (leave early weights frozen)
5. Unfreeze all weights and fine tune the full network by training with a supervised approach, given the *pre-training* weight settings

## Greedy Layer-Wise Training

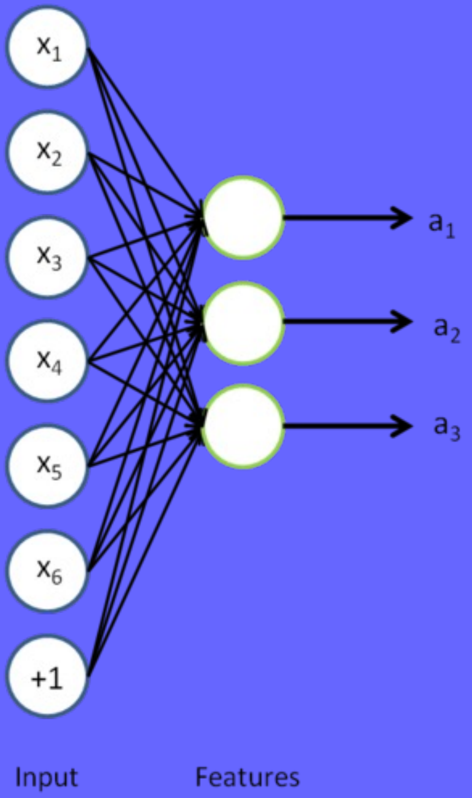
- Greedy layer-wise training avoids many of the problems of trying to train a deep net in a supervised fashion
  - Each layer gets full learning focus in its turn since it is the only current "top" layer (no unstable gradient issues, etc.)
  - Can take advantage of unlabeled data
  - When you finally tune the entire network with supervised training the network weights have already been adjusted so that you are in a good error basin and just need fine tuning. This helps with problems of
    - Ineffective early layer learning
    - Deep network local minima
- The two early landmark approaches
  - Deep Belief Networks
  - Stacked Auto-Encoders

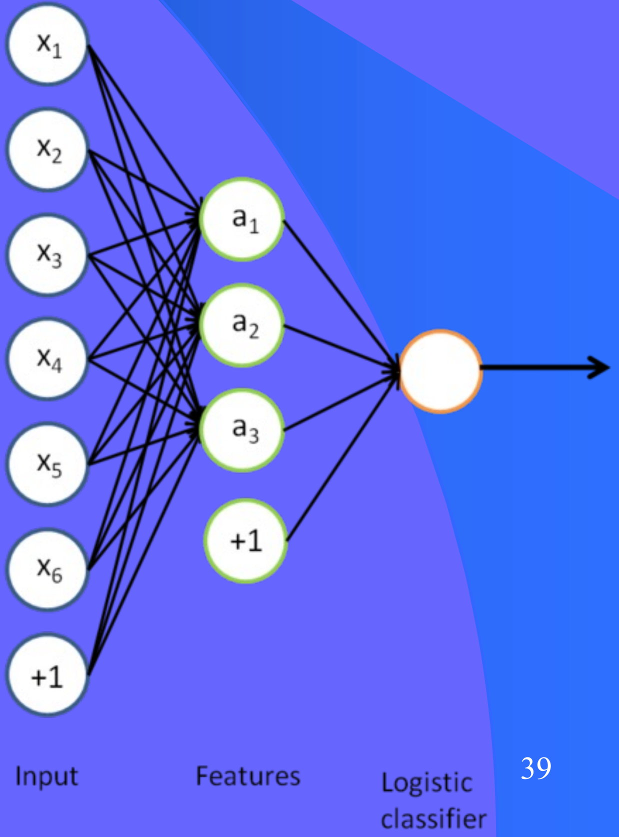
# Auto-Encoders

- A type of unsupervised learning which discovers generic features of the data
  - Learn identity function by learning important sub-features
  - Compression, etc. – Undercomplete $|\mathbf{h}| < |\mathbf{x}|$
  - For $|\mathbf{h}| \geq |\mathbf{x}|$ (Overcomplete case more common in deep nets) use regularized autoencoding: Loss function includes regularizer to make sure we don't just pass through the data (e.g. sparsity, noise robustness, etc.)



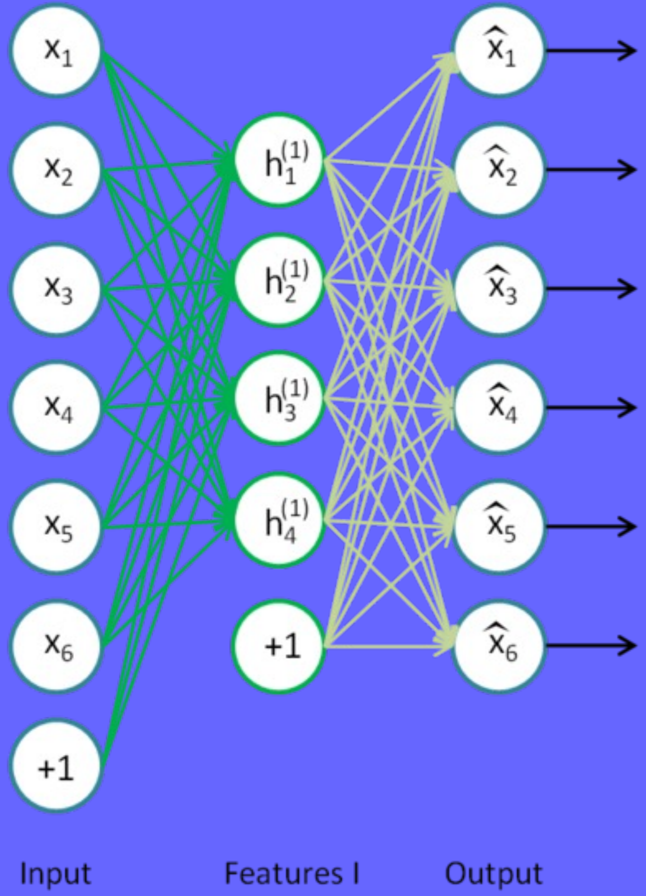
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ +1
+1
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$
Input
Features
Output



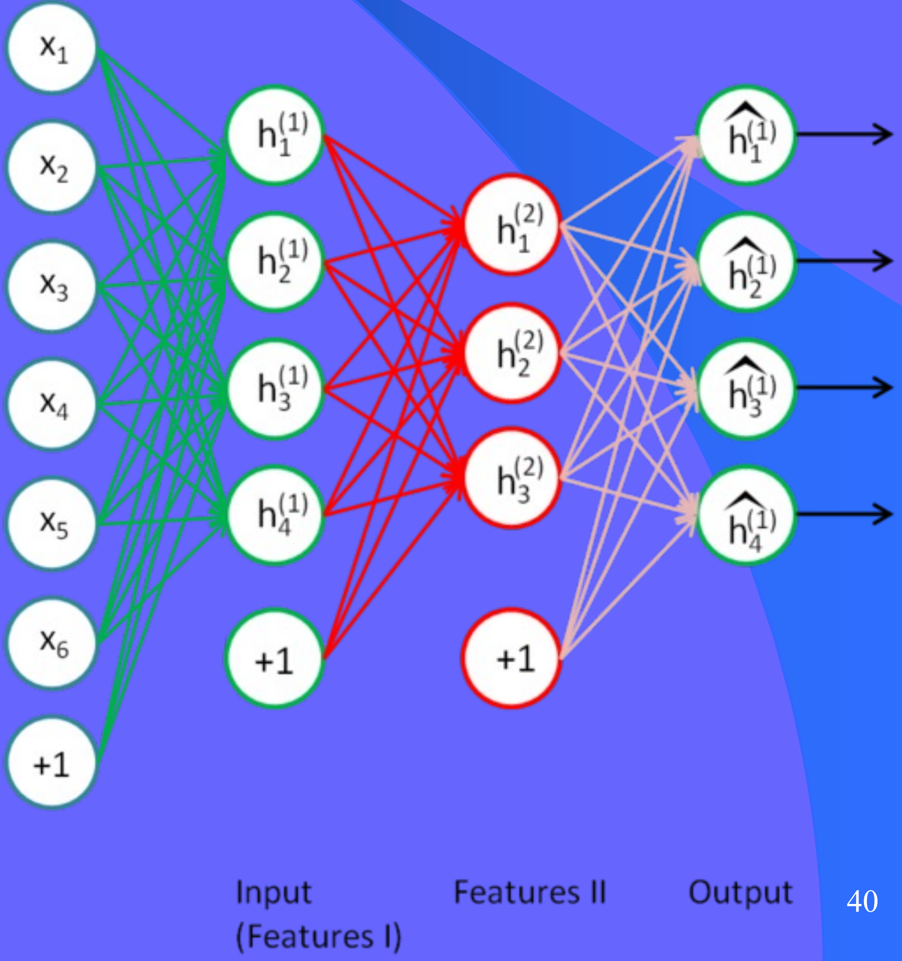


## Stacked Auto-Encoders

- Bengio (2007) – After Deep Belief Networks (2006)
- Stack many (sparse) auto-encoders in succession and train them using greedy layer-wise training
- Drop the decode output layer each time



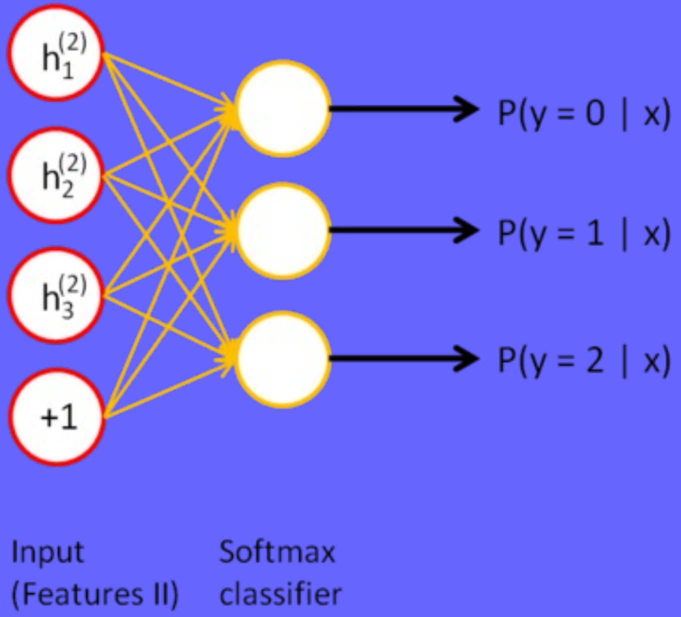
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ +1
$h_1^{(1)}$ $h_2^{(1)}$ $h_3^{(1)}$ $h_4^{(1)}$ +1
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$
Input
Features I
Output

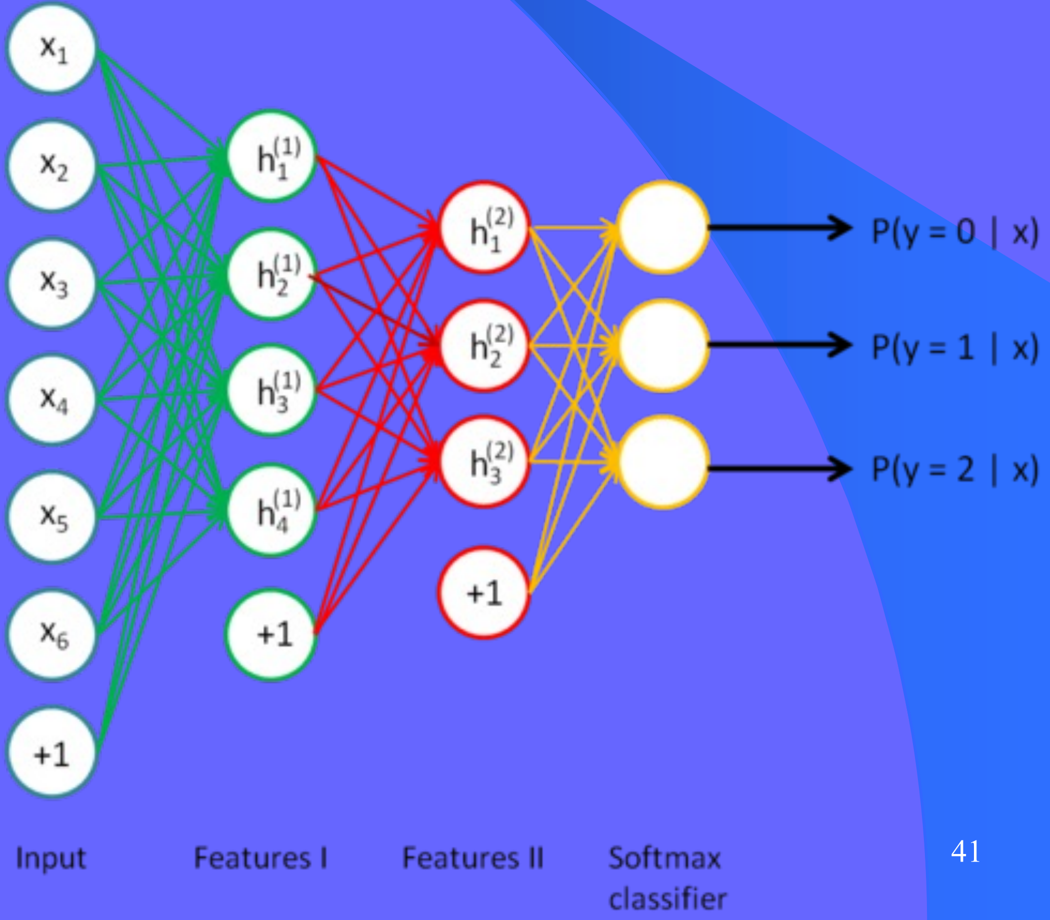


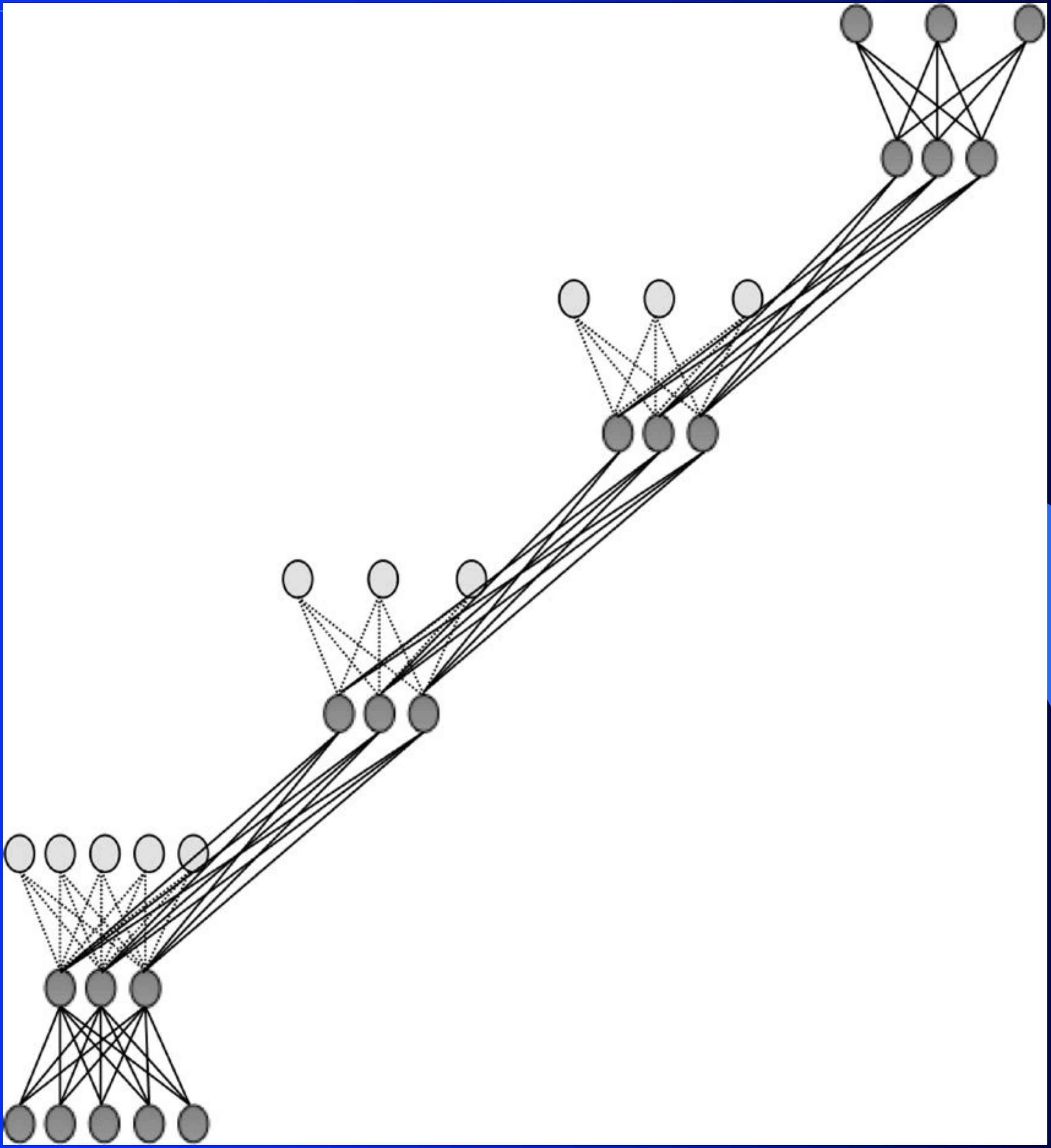
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ +1
$h_1^{(1)}$ $h_2^{(1)}$ $h_3^{(1)}$ $h_4^{(1)}$ +1
$h_1^{(2)}$ $h_2^{(2)}$ $h_3^{(2)}$ +1
$\hat{h}_1^{(1)}$ $\hat{h}_2^{(1)}$ $\hat{h}_3^{(1)}$ $\hat{h}_4^{(1)}$
Input (Features I)
Features II
Output

## Stacked Auto-Encoders

- Do supervised training (can now only used labeled examples) on the last layer using final features
- Then do supervised training on the entire network to fine-tune all weights





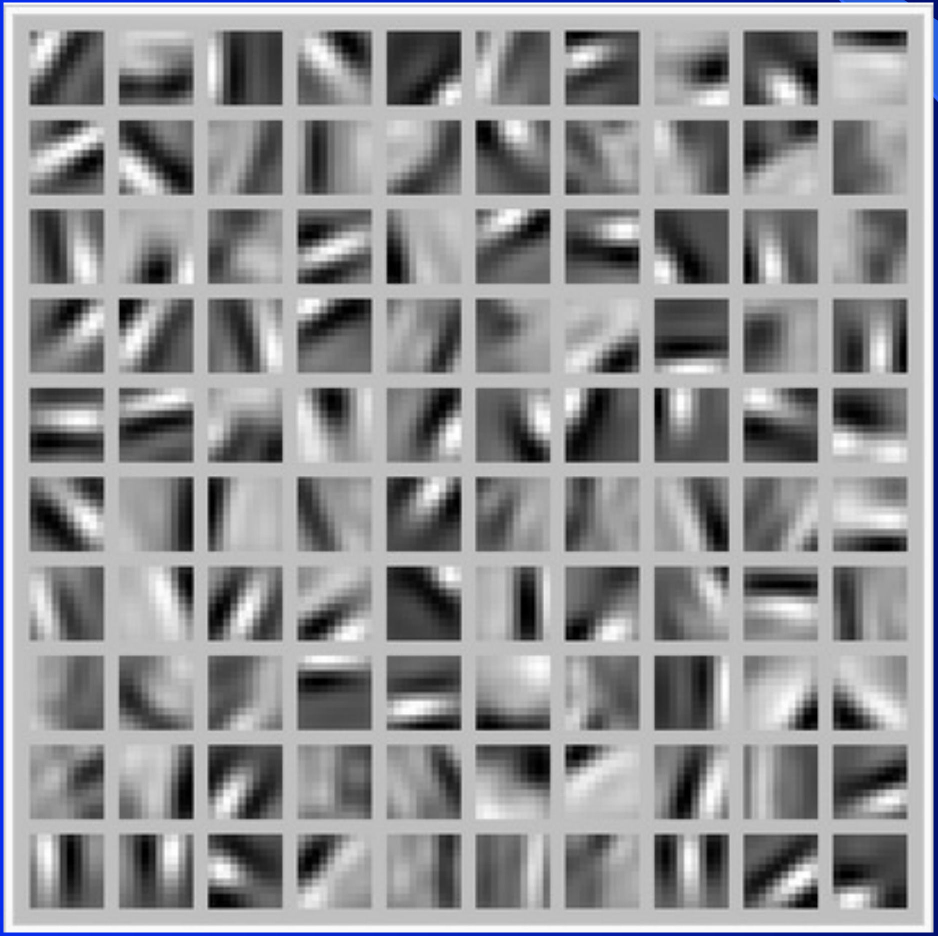


## Sparse Encoders

- Auto encoders will often do a dimensionality reduction
  - PCA-like or non-linear dimensionality reduction
- This leads to a "dense" representation which is nice in terms of parsimony
  - All features typically have non-zero values for any input and the combination of values contains the compressed information
- However, this distributed and entangled representation can often make it more difficult for successive layers to pick out the salient features
- A *sparse* representation uses more features where at any given time many/most of the features will have a 0 value (ReLUs help)
  - Thus there is an implicit compression each time but with varying nodes
  - This leads to more localist variable length encodings where a particular node (or small group of nodes) with value 1 signifies the presence of a high-order feature (small set of bases)
  - A type of simplicity bottleneck (regularizer)
  - This is easier for subsequent layers to use for learning

## Sparse Representation

- For bases below, which is easier to see intuition for current pattern - if a few of these are on and the rest 0, or if all have some non-zero value?
- Easier to learn if sparse



## How do we implement a sparse Auto-Encoder?

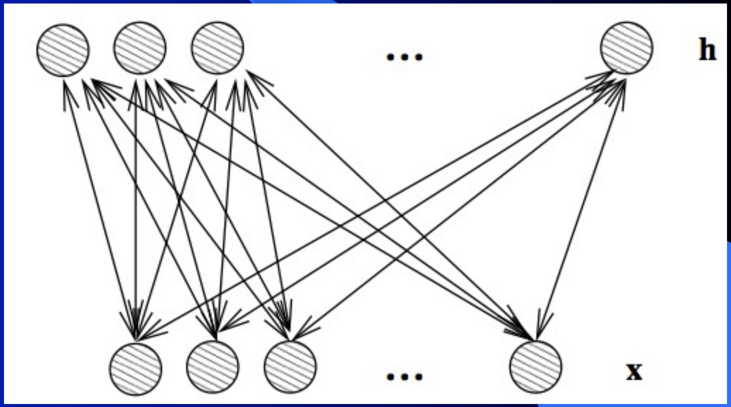
- Use more hidden nodes in the encoder
- Use regularization techniques which encourage sparseness (e.g. a significant portion of nodes have 0 output for any given input)
  - Penalty in the learning function for non-zero nodes
  - Weight decay
  - etc.
- De-noising Auto-Encoder
  - Stochastically corrupt training instance each time, but still train auto-encoder to decode the uncorrupted instance, forcing it to de-noise and learn conditional dependencies within the instance
  - Improved empirical results, handles missing values well

## Stacked Auto-Encoders

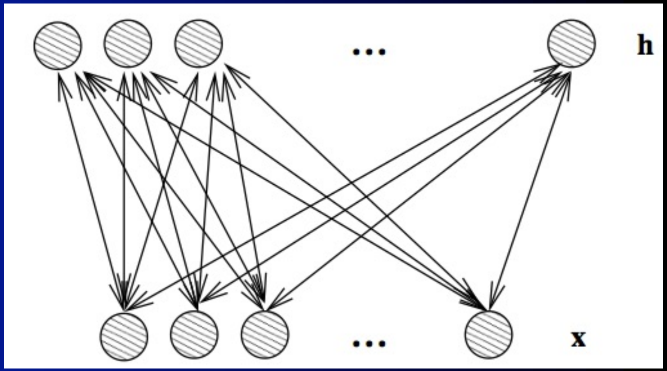
- Concatenation approach (i.e. using both hidden features and original features in final (or other) layers) can be better if not doing fine tuning. If fine tuning, the pure replacement approach works well.
- Always fine tune if there is a sufficient amount of labeled data
- For real valued inputs, auto-encode training is regression and thus could use linear output node activations (thrown out after anyways), still ReLU/non-linear at hidden which are final nodes
- Stacked Auto-Encoders empirically not quite as accurate as DBNs (Deep Belief Networks)
  - (with De-noising auto-encoders, stacked auto-encoders competitive with DBNs)
  - Not generative like DBNs, though recent work with de-noising auto-encoders may allow generative capacity

## Deep Belief Networks (DBN)

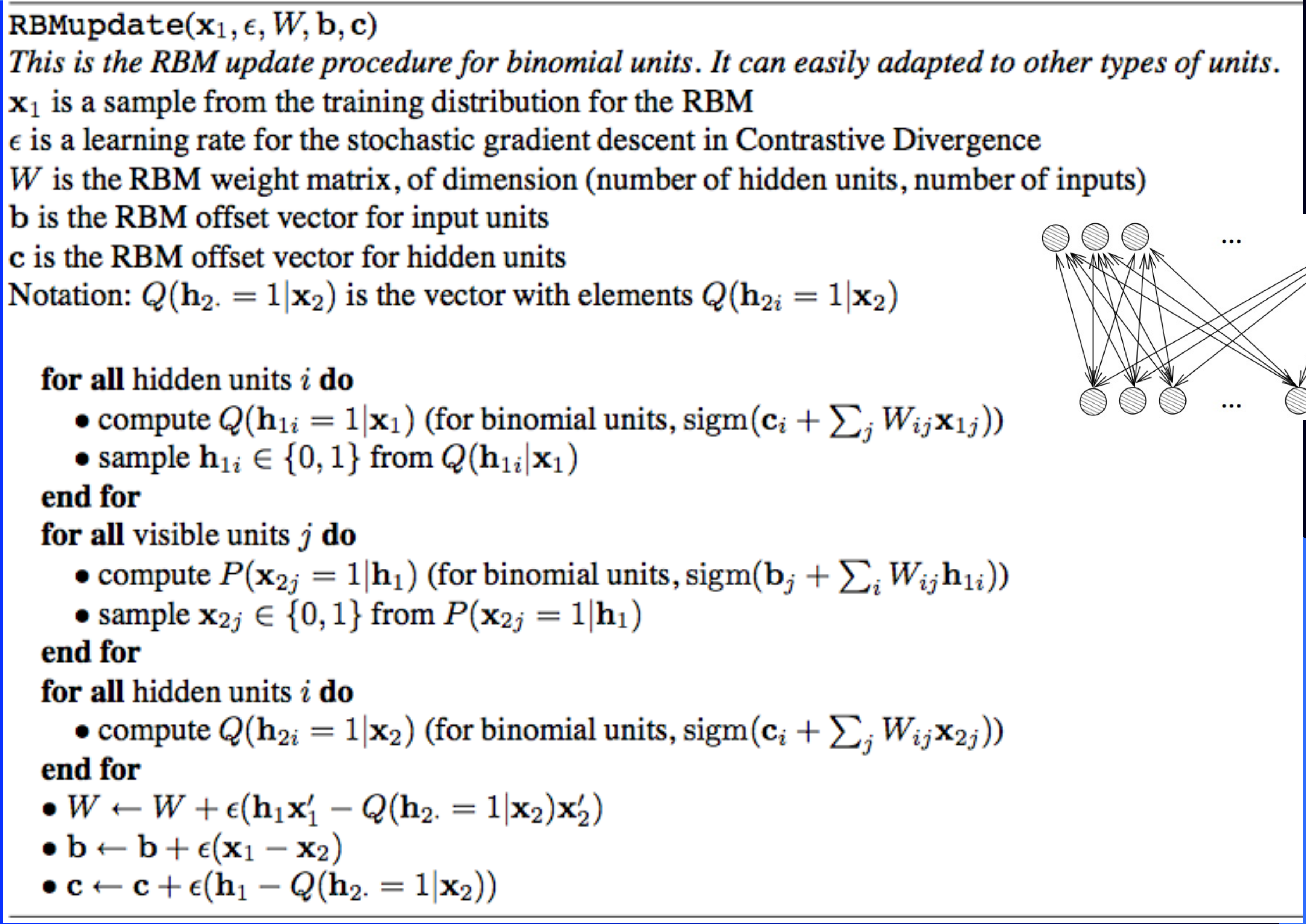
- Geoff Hinton (2006) – Beginning of Deep Learning hype – outperformed kernel methods on MNIST – Also generative
- Uses Greedy layer-wise training but each layer is an RBM (Restricted Boltzmann Machine)
- RBM is a constrained Boltzmann machine with
  - No lateral connections between hidden (**h**) and visible (**x**) nodes
  - Symmetric weights
  - Does not use annealing/temperature, but that is all right since each RBM not seeking a global minima, but rather an incremental transformation of the feature space
  - Typically uses probabilistic logistic node, but other activations possible



# RBM Sampling and Training



- Initial state typically set to a training example **x** (can be real valued)
- Because of RBM, sampling is simple iterative back and forth process
  - $P(h_i = 1|\mathbf{x}) = \text{sigmoid}(W_i\mathbf{x} + c_i) = 1/(1+e^{-net(h_i)})$ // $c_i$ is hidden node bias
  - $P(x_i = 1|\mathbf{h}) = \text{sigmoid}(W'_i\mathbf{h} + b_i) = 1/(1+e^{-net(x_i)})$ // $b_i$ is visible node bias
- Contrastive Divergence (CD-*k*): How much contrast (in the statistical distribution) is there in the divergence from the original training example to the relaxed version after *k* relaxation steps
- Then update weights to decrease the divergence as in Boltzmann
- Typically just do CD-1 (Good empirical results)
  - Since small learning rate, doing many CD-1 updates is similar to doing fewer versions of CD-*k* with $k > 1$
  - Note CD-1 just needs to get the gradient direction right, which it usually does, and then change weights in that direction according to the learning rate



$\texttt{RBMupdate}(\mathbf{x}_1, \epsilon, W, \mathbf{b}, \mathbf{c})$

*This is the RBM update procedure for binomial units. It can easily adapted to other types of units.*

$\mathbf{x}_1$ is a sample from the training distribution for the RBM

$\epsilon$ is a learning rate for the stochastic gradient descent in Contrastive Divergence

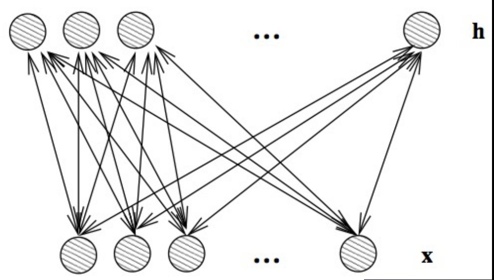
$W$ is the RBM weight matrix, of dimension (number of hidden units, number of inputs)

$\mathbf{b}$ is the RBM offset vector for input units

$\mathbf{c}$ is the RBM offset vector for hidden units

Notation: $Q(\mathbf{h}_{2\cdot} = 1|\mathbf{x}_2)$ is the vector with elements $Q(\mathbf{h}_{2i} = 1|\mathbf{x}_2)$

```
for all hidden units $i$ do
    • compute $Q(\mathbf{h}_{1i} = 1|\mathbf{x}_1)$ (for binomial units, $\text{sigm}(\mathbf{c}_i + \sum_j W_{ij}\mathbf{x}_{1j})$)
    • sample $\mathbf{h}_{1i} \in \{0,1\}$ from $Q(\mathbf{h}_{1i}|\mathbf{x}_1)$
end for
for all visible units $j$ do
    • compute $P(\mathbf{x}_{2j} = 1|\mathbf{h}_1)$ (for binomial units, $\text{sigm}(\mathbf{b}_j + \sum_i W_{ij}\mathbf{h}_{1i})$)
    • sample $\mathbf{x}_{2j} \in \{0,1\}$ from $P(\mathbf{x}_{2j} = 1|\mathbf{h}_1)$
end for
for all hidden units $i$ do
    • compute $Q(\mathbf{h}_{2i} = 1|\mathbf{x}_2)$ (for binomial units, $\text{sigm}(\mathbf{c}_i + \sum_j W_{ij}\mathbf{x}_{2j})$)
end for
• $W \leftarrow W + \epsilon(\mathbf{h}_1\mathbf{x}_1' - Q(\mathbf{h}_{2\cdot} = 1|\mathbf{x}_2)\mathbf{x}_2')$
• $\mathbf{b} \leftarrow \mathbf{b} + \epsilon(\mathbf{x}_1 - \mathbf{x}_2)$
• $\mathbf{c} \leftarrow \mathbf{c} + \epsilon(\mathbf{h}_1 - Q(\mathbf{h}_{2\cdot} = 1|\mathbf{x}_2))$
```



$$\Delta w_{ij} = \varepsilon(h_{1,j} \cdot x_{1,i} - Q(h_{k+1,j} = 1 | \mathbf{x}_{k+1})x_{k+1,i})$$

$$\Delta w_{ij} = \varepsilon(initial\_h\_sample \cdot initial\_x - final\_h\_probabilty \cdot final\_x\_sample)$$

## RBM Update Notes and Variations

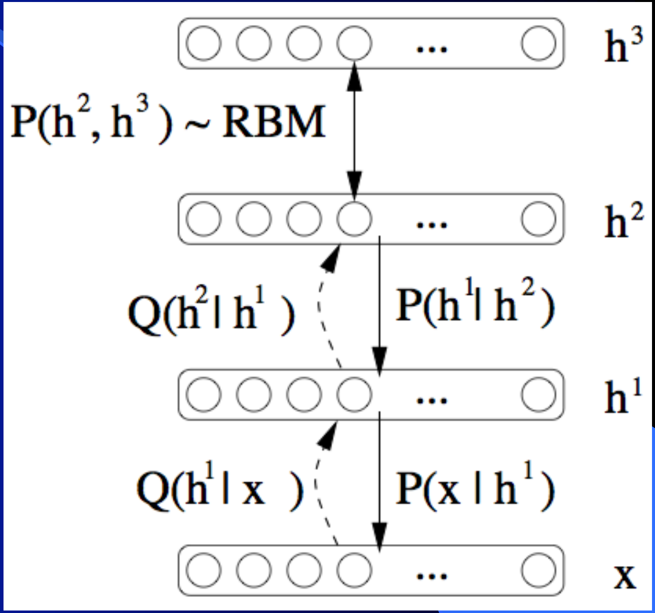
- Example (based on HW with different values) and Homework
- Binomial unit means the standard MLP sigmoid unit
- *Q* and *P* are probability distribution vectors for hidden (**h**) and visible/input (**x**) vectors respectively
- During relaxation/weight update can alternatively do updates based on the real valued probabilities (sigmoid(*net*)) rather than the 1/0 sampled logistic states
  - Always use actual/binary values from initial x -> h
    - Doing this makes the hidden nodes a sparser bottleneck and is a regularizer helping to avoid overfit
  - Could use probabilities on the h -> x and/or final x -> h
    - in CD-*k* the final update of the hidden nodes usually uses the probability value to decrease the final arbitrary sampling variation (sampling noise)
- Lateral restrictions of RBM allow this fast sampling

## RBM Update Variations and Notes

- Initial weights, small random, 0 mean, sd ~ .01
  - Don't want hidden node probabilities early on to be close to 0 or 1, else slows learning, since less early randomness/mixing? Note that this is a bit like annealing/temperature in Boltzmann
- Set initial **x** bias values as a function of how often node is on in the training data, and **h** biases to 0 or negative to encourage sparsity
- Better speed when using momentum (~.5)
- Weight decay good for smoothing and also encouraging more mixing (hidden nodes more stochastic when they do not have large net magnitudes)

# Deep Belief Network Training

- Same greedy layer-wise approach
- First train lowest RBM ($h^0 - h^1$) using RBM update algorithm (note $h^0$ is x)
- Freeze weights and train subsequent RBM layers
- Then connect final outputs to a supervised model and train that model
- Finally, unfreeze all weights, and fine tune as an MLP using the initial weights found by DBN training
- Can do execution as just the tuned MLP or as the RBM sampler with the tuned weights

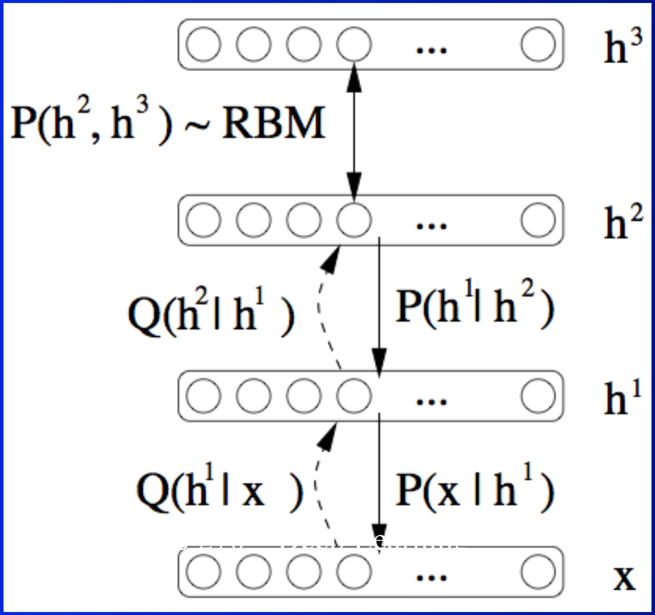


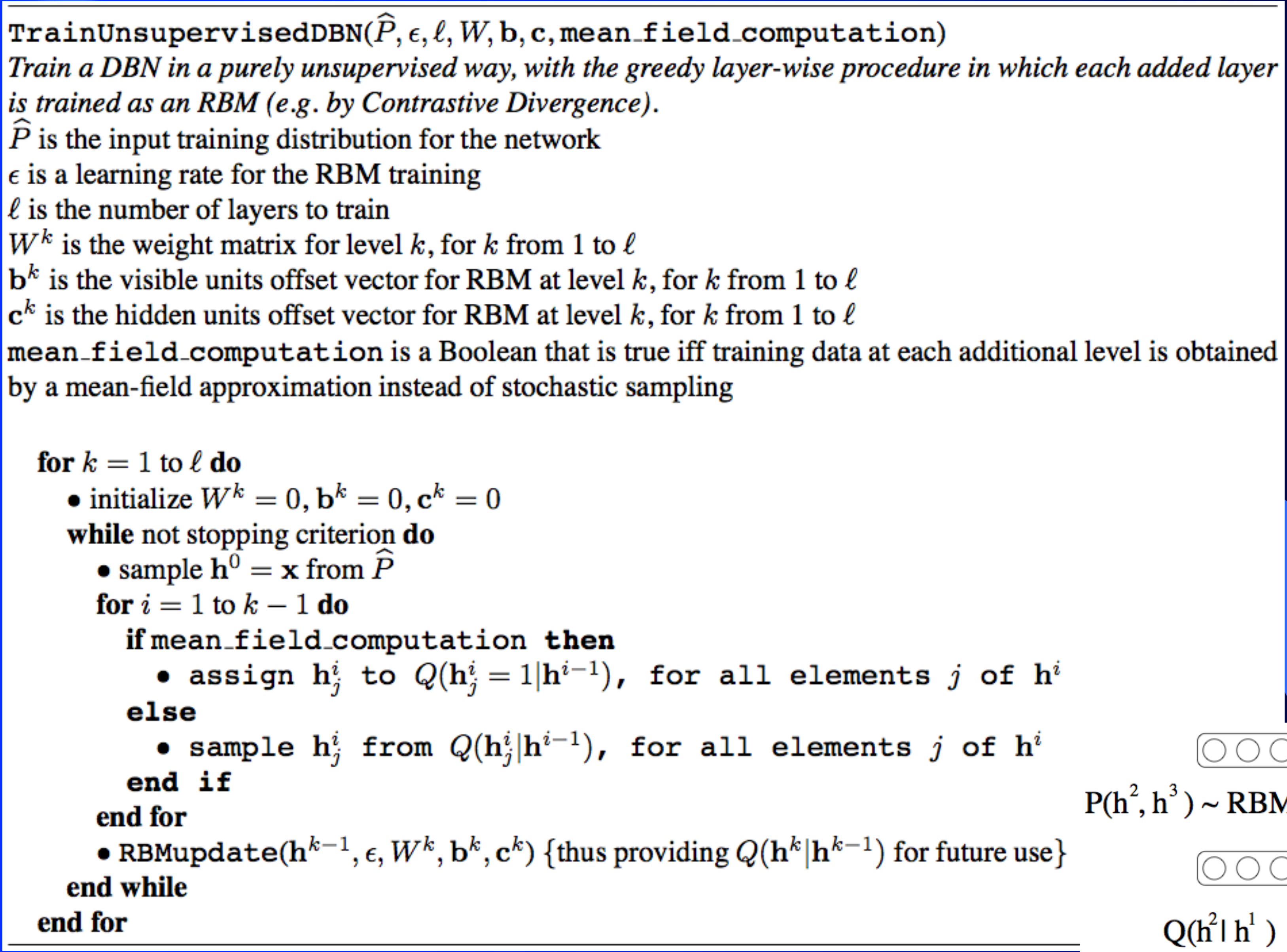
During execution can iterate multiple times at the top RBM layer

## Can use DBN as a Generative model to create sample x vectors

1. Initialize top layer to an arbitrary vector (commonly a training set vector)
   - Gibbs sample (relaxation) between the top two layers *m* times
   - If we initialize top layer with values obtained from a training example, then need less Gibbs samples
2. Pass the vector down through the network, sampling with the calculated probabilities at each layer
3. Last sample at bottom is the generated x vector (can be real valued if we use the probability vector rather than sample)

Alternatively, can start with an x at the bottom, relax to a top value, then start from that vector when generating a new x, which is the dotted lines version. More like standard Boltzmann machine processing.





TrainUnsupervisedDBN($\widehat{P}, \epsilon, \ell, W, \mathbf{b}, \mathbf{c}$, mean_field_computation)

*Train a DBN in a purely unsupervised way, with the greedy layer-wise procedure in which each added layer is trained as an RBM (e.g. by Contrastive Divergence).*

$\widehat{P}$ is the input training distribution for the network

$\epsilon$ is a learning rate for the RBM training

$\ell$ is the number of layers to train

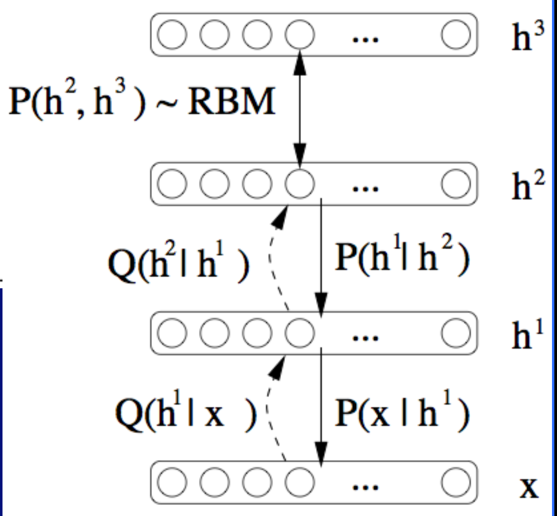
$W^k$ is the weight matrix for level $k$, for $k$ from 1 to $\ell$

$\mathbf{b}^k$ is the visible units offset vector for RBM at level $k$, for $k$ from 1 to $\ell$

$\mathbf{c}^k$ is the hidden units offset vector for RBM at level $k$, for $k$ from 1 to $\ell$

mean_field_computation is a Boolean that is true iff training data at each additional level is obtained by a mean-field approximation instead of stochastic sampling

```
for $k = 1$ to $\ell$ do
  • initialize $W^k = 0, \mathbf{b}^k = 0, \mathbf{c}^k = 0$
  while not stopping criterion do
    • sample $\mathbf{h}^0 = \mathbf{x}$ from $\widehat{P}$
    for $i = 1$ to $k - 1$ do
      if mean_field_computation then
        • assign $\mathbf{h}^i_j$ to $Q(\mathbf{h}^i_j = 1|\mathbf{h}^{i-1})$, for all elements $j$ of $\mathbf{h}^i$
      else
        • sample $\mathbf{h}^i_j$ from $Q(\mathbf{h}^i_j|\mathbf{h}^{i-1})$, for all elements $j$ of $\mathbf{h}^i$
      end if
    end for
    • RBMupdate($\mathbf{h}^{k-1}, \epsilon, W^k, \mathbf{b}^k, \mathbf{c}^k$) {thus providing $Q(\mathbf{h}^k|\mathbf{h}^{k-1})$ for future use}
  end while
end for
```



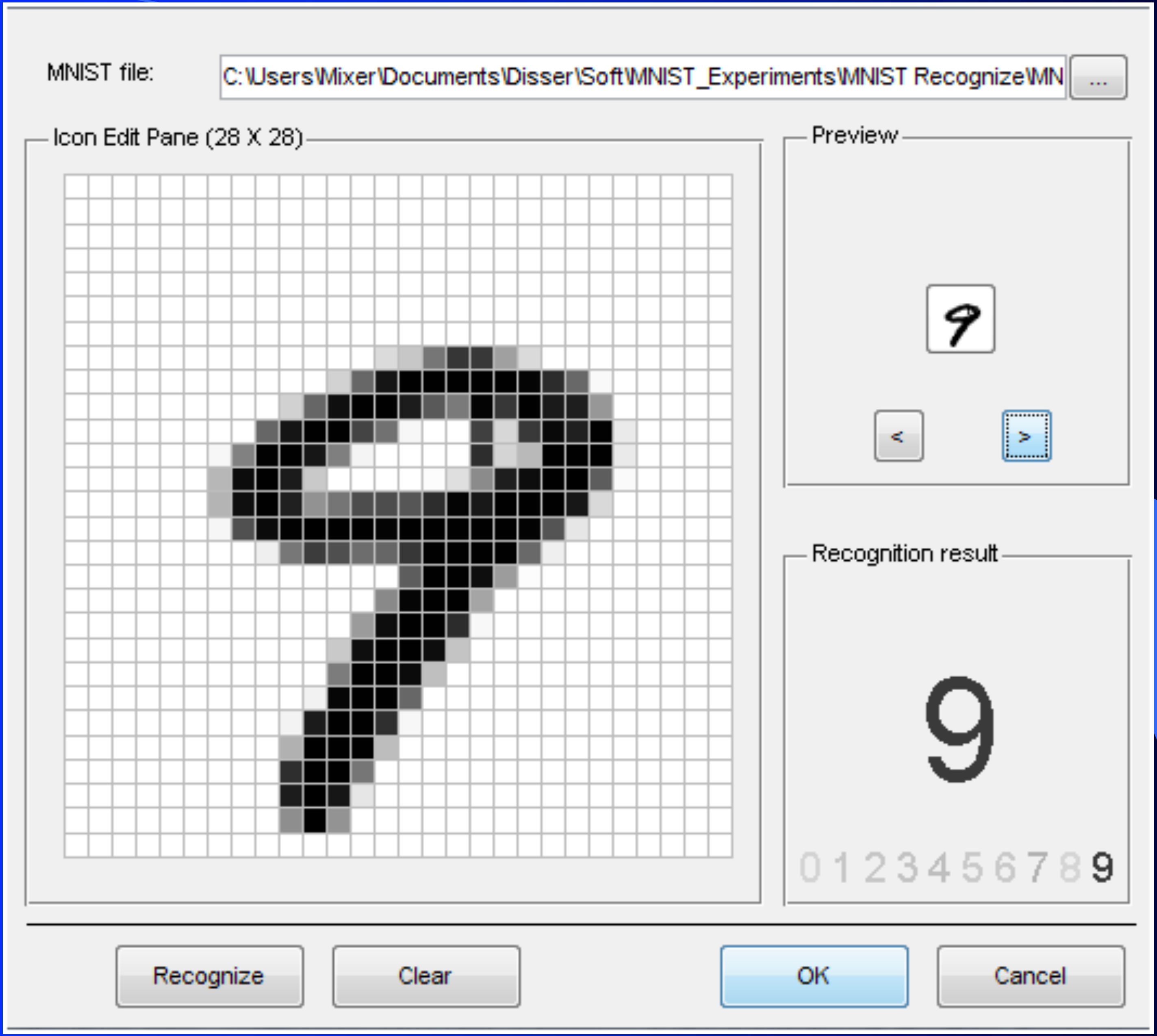
Middle for-loop samples from input up to current RBM layer being updated – none for 1st layer, mean_field_computation just a flag on whether to sample or use real values

## DBN Execution

- After all layers have learned then the output of the last layer can be input to a supervised learning model
- Note that at this point we could potentially throw away the downward bias weights in the network as they will not actually be used during the feedforward discriminative execution process (as we did with the Stacked Auto Encoder)
  - If we are relaxing *M* times in the top layer then we would still need the downward weights for that layer
  - Also if we are generating **x** values we would need all of them
- The final weight tuning is usually done as an MLP with backpropagation, which only updates the feedforward weights
- Can do execution as just the tuned MLP or as the RBM sampler with the tuned weights

## DBN Learning Notes

- RBM stopping criteria still in issue.
- Each layer updates weights so as to make training sample patterns more likely (lower energy) in the free state (and non-training sample patterns less likely).
- This unsupervised approach learns broad features (in the hidden/subsequent layers of RBMs) which can aid in the process of making the types of patterns found in the training set more likely. This discovers features which can be associated across training patterns, and thus potentially helpful for other goals with the training set (classification, compression, etc.)
- Note still pairwise weights in RBMs, but because we can choose the number of hidden units and layers, we can represent any arbitrary distribution



MNIST file: C:\Users\Mixer\Documents\Disser\Soft\MNIST_Experiments\MNIST Recognize\MN

## DBN Project Notes

- To be consistent just use 28 × 28 (764) data set of gray scale values (0-255)
  - Normalize to 0-1
  - Could try better preprocessing if want and helps in published accuracies, but start/stay with this
  - Small random initial weights
- Parameters
  - Hinton Paper, others – do a little searching and e-mail me a reference for extra credit points
  - http://yann.lecun.com/exdb/mnist/ for sample approaches
- Straight 200 hidden node MLP does quite good ~98%
  - Rough Hyperparameters - LR: ~.05-.1, Momentum ~.5
- Best class DBN results: ~98.5% - which is competitive
  - About half students never beat MLP baseline
  - Can you beat the 98.5%?

# Deep Learning Project Past Experience

- Structure: ~3 hidden layers, ~500ish nodes/layer, more nodes/layers can be better but training is longer
- Training time:
  - DBN: ~10 epochs with the 60K set, small LR ~.005 often good
  - Can go longer, does not seem to overfit with the large data set
  - SAE: Can saturate/overfit, ~3 epochs good, but will be a function of your de-noising approach, which is essential for sparsity, use small LR ~.005, long training – up to 50 hours, got 98.55
    - Larger learning rates often lead to low accuracy for both DBN and SAE
- Sampling vs using real probability value in DBN
  - Best results found when using real values vs. sampling
  - Some found sampling on the back-step of learning helps
  - When using sampling, probably requires longer training, but could actually lead to bigger improvements in the long run
  - Typical forward flow non-sampled during execution, but could do some sampling on the *m* iterations at the top layer. Some success with back-step at the top layer iteration (most don't do this at all)
  - We need to try/discover better variations

## Deep Learning Project Past Experience

- Note: If we held out 50K of the dataset as unsupervised, then deep nets would more readily show noticeable improvement over BP
- A final full network fine tune with BP always helps
  - But can take 20+ hours
- Key take away – Most actual time spent training with different parameters. Thus, start early, and then you will have time to try multiple long runs to see which variations work. This does not take that much personal time, as you simply start it with some different parameters and go away for a day. If you wait until the last few days, there is no time to do these experiments.

## DBN Notes

- Can use lateral connections in RBM (no longer RBM) but sampling becomes more difficult (intractable, approximation such as MCMC) – ala standard Boltzmann requiring longer sampling chains.
  - Lateral connections can capture pairwise dependencies allowing the hidden nodes to focus on higher order issues. Can get better results.
- Conditional and Temporal RBMs – allow node probabilities to be conditioned by some other inputs – context, recurrence (time series changes in input and internal state), etc.

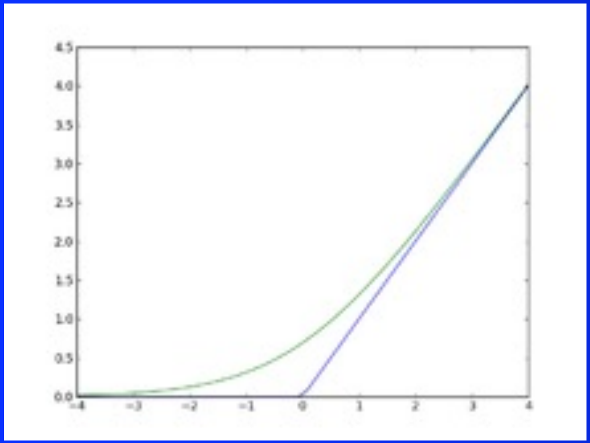
## Discrimination with Deep Belief Networks

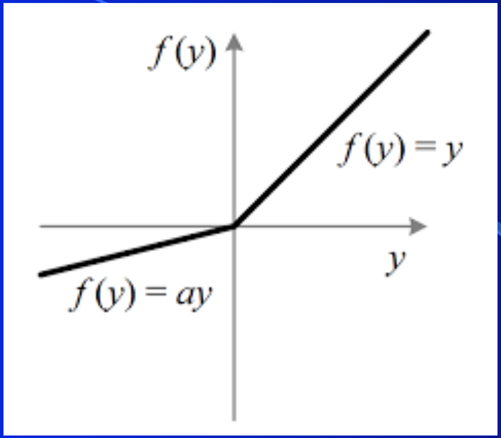
- Discrimination approaches with DBNs (Deep Belief Net)
  - Use outputs of DBNs as inputs to supervised model (i.e. just an unsupervised preprocessor for feature extraction)
    - Basic approach we have been discussing
  - Train a DBN for each class. For each clamp the unknown x and iterate *m* times. The DBN that ends with the lowest normalized free energy (softmax variation) is the winner.
  - Train just one DBN for all classes, but with an additional visible unit for each class. For each output class:
    - Clamp the unknown x, relax, and then see which final state has lowest free energy – no need to normalize since all energies come from the same network.
- See http://deeplearning.net/demos/

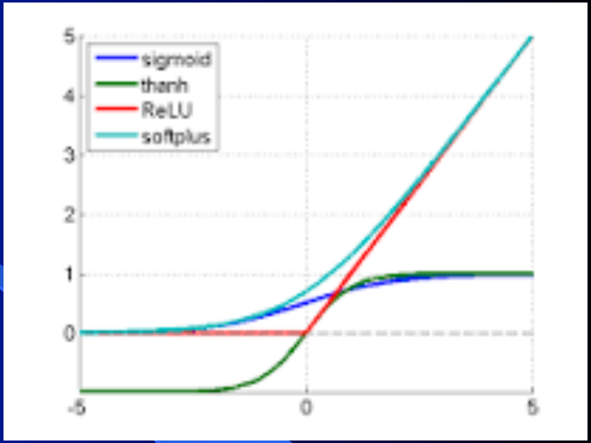
## More Efficient Deep Learning

- Recent success in doing supervised deep learning with extensions which diminish the effect of early learning difficulties (unstable gradient, etc.)
- Patience (now that we know it can be worth it), faster computers, and use of GPUs/TPUs
- More efficient activation functions (e.g. ReLU) in terms of both computation and avoiding $f'(net)$ saturation
  - Also can be helpful to have 0 mean activations at each level, so sigmoid is frowned upon these days. If you want a saturating activation function, tanh is often preferred.
- Speed up and regularization approaches
- Improved Hyperparameters
- Batch Normalization – re-normalize activations at each layer
- Residual Nets

# Rectified Linear Units







- $f(x)$ = Max(0,$x$) More efficient gradient propagation, derivative is 0 or constant, just fold into learning rate
  - Helps $f'(net)$ issue, but still left with other unstable gradient issues
- More efficient computation: Only comparison, addition and multiplication.
  - Leaky ReLU $f(x) = x$ if $x > 0$ else $ax$, where $0 \leq a <= 1$, so that derivate is not 0 and can do some learning for net < 0 (does not "die").
  - Lots of other variations
- Sparse activation: For example, in a randomly initialized network, only about 50% of hidden units are activated (having a non-zero output)
- Learning in linear range easier for most learning models

## Speed up variations of SGD

- Use mini-batch rather than single instance for better gradient estimate – Helpful if using GD variation more sensitive to bad gradient, and *especially* for parallel implementations
- Momentum (i.e. Adaptive learning rate) approaches are important since anything to speed-up learning is helpful
  - Standard Momentum
    - Note the these approaches already do an averaging of grading also making mini-batch less critical
  - Nesterov Momentum – Calculate point you would go to if using normal momentum. Then, compute gradient at that point. Do normal update using *that* gradient and momentum.
  - Rprop – Resilient BP, if gradient sign inverts, decrease the node's individual LR, else increase it – common goal is faster in the flats, there are variants that backtrack a step, etc.
  - Adagrad – Scale LRs inversely proportional to sqrt(sum( historical values)) – LRs with smaller derivatives are decreased less
  - RMSprop – Adagrad but uses exponentially weighted moving average, older updates basically forgotten
  - Adam (Adaptive moments) –Momentum terms on both gradient and squared gradient (1st and 2nd moments) – update based on both

# Regularization – Dropout Common

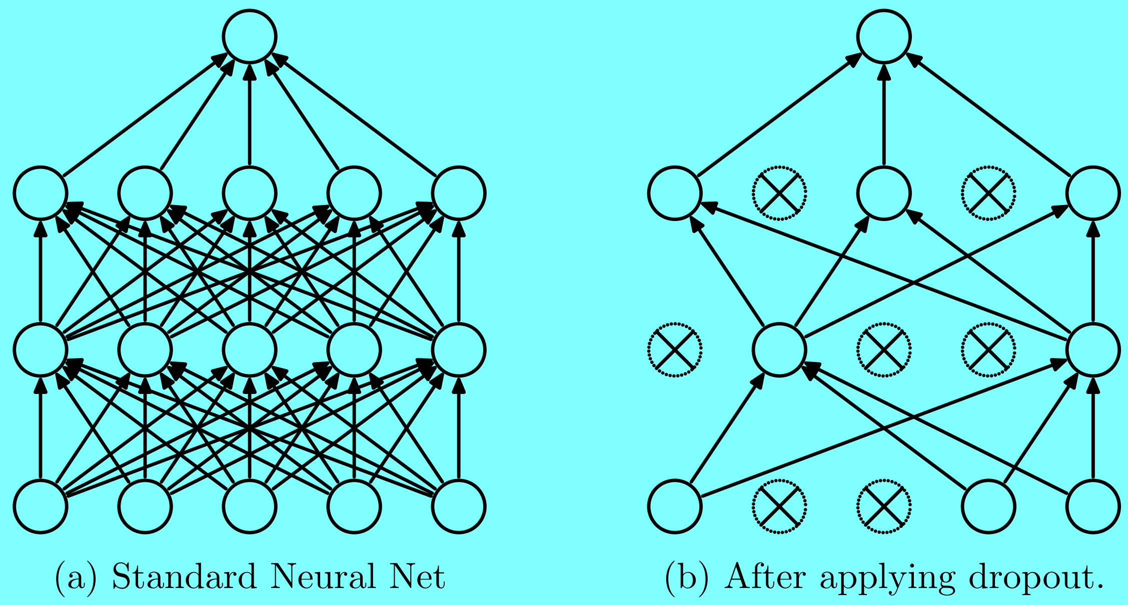


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

- For each instance drop a node (hidden or input) and its connections with probability *p* and train
- Final net just has all averaged weights (actually scaled by 1-*p* since that better matches the expected values at training time)
- As if ensembling $2^n$ different network substructures
- Lots of variations – Dropconnect, etc.

## Improved Initial Hyperparameter Settings

- Deep networks are more sensitive than shallow networks to hyperparameter settings
- More critical for deep learning in order to get more balanced learning across all layers
- Smaller LRs – patience
- To encourage sparsity sometimes initial biases set negative or more initial 0 weights are interspersed
- Initial weights – initialize a little larger in effort to find a balance which learns well across all layers. Common is to select initial weights from a uniform distribution between
  - -*c*/root(node fan-in), *c*/root(node fan-in) (*c* = 1 Xavier, *c* = 2 He)
  - -*c*/root(node fan-in + fan-out), *c*/root(node fan-in + fan-out)
    - Can do Gaussian distribution with above as variances
    - Lots of other variations and current work

# Batch Normalization (2015)

- Rather than just guess initial parameters to maintain learning balance, renormalize activations at each layer to maintain balance
- Just like it is critical to normalize our initial features, we consider each layer to be a new feature set which can also be normalized
- We like 0-mean and unit variance inputs (standard 1st layer normalization), we can just re-normalize the net value (less commonly the activation value) for each input dimension *k* at each layer

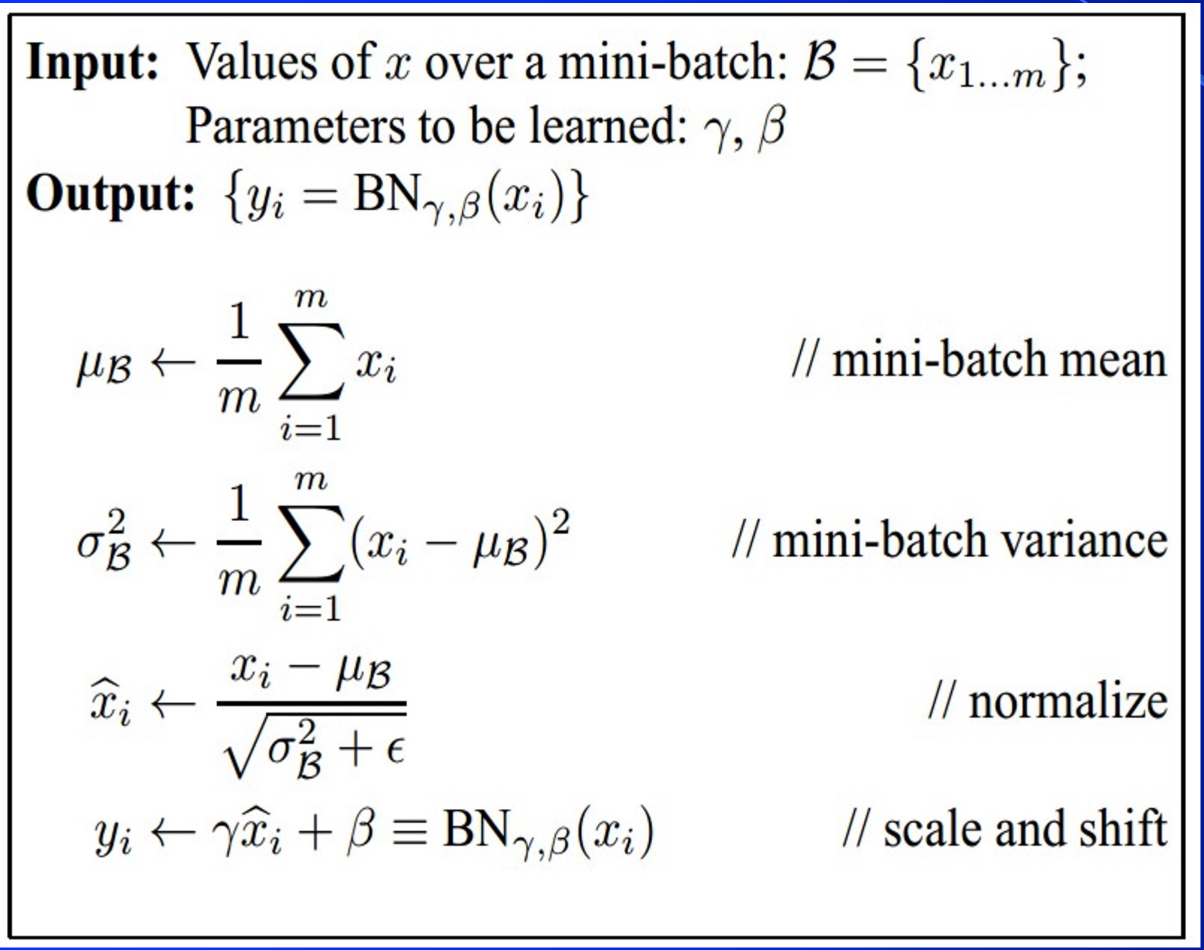
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

- Think of each net value as a new feature. Want mean and variance of that activation for the entire data set. Changing! Approximate the empirical mean and variance over a mini-batch of instances.
- Goes beyond standard normalization by allowing scaling and shifting of the normalized values with 2 learnable weights per input, $\gamma$ and $\beta$, to attain the final batch normalization (allows recovery of initial or any needed function)

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

# Batch Normalization

[Ioffe and Szegedy, 2015]

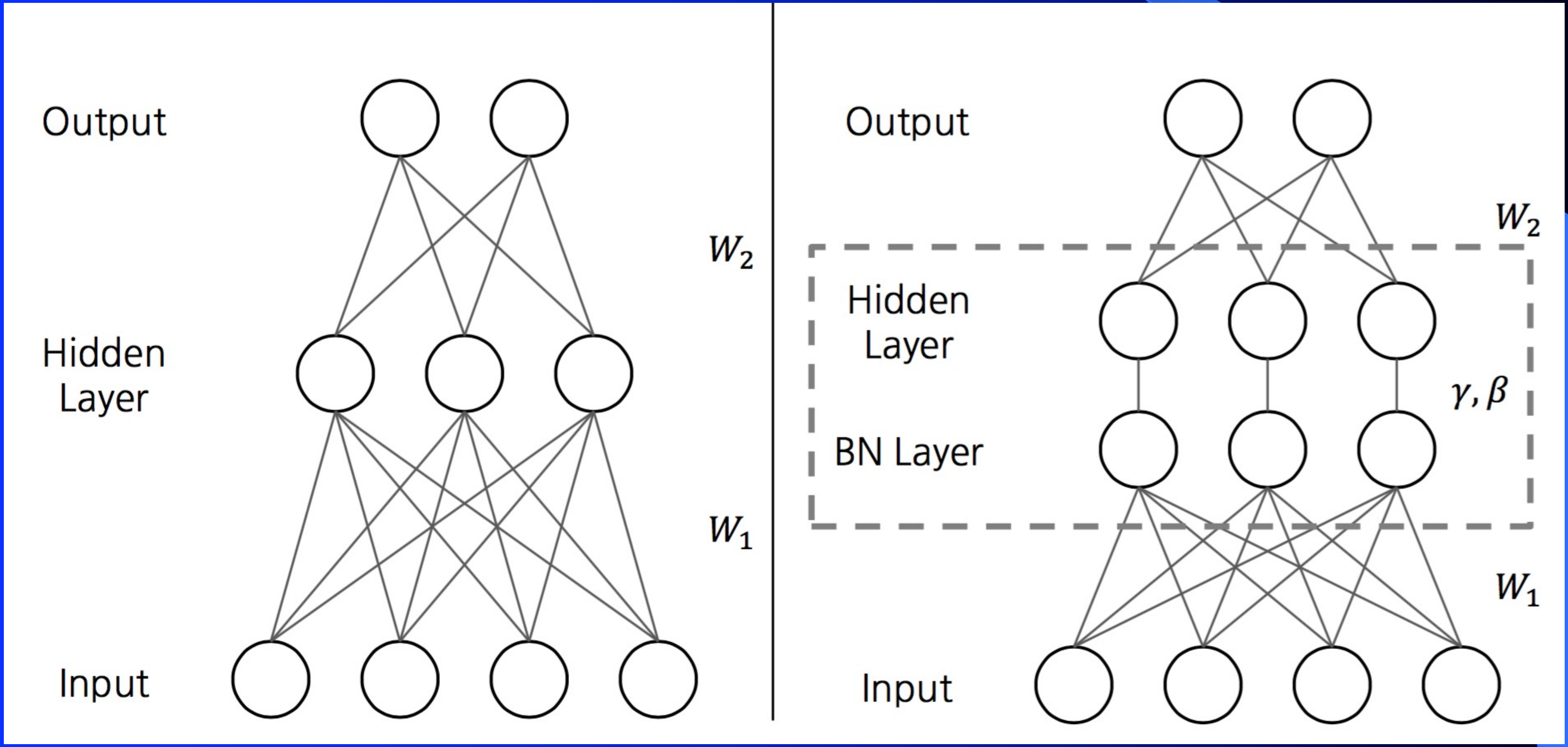


```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                  // mini-batch mean

$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$          // mini-batch variance

$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                // normalize

$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$              // scale and shift
```

- Note: Dropped *k* for simplicity
- *γ* and *β* learned as part of gradient descent
- At test time BatchNorm layer functions differently:
- The mean/std are not computed based on a batch. Instead, we use empirical means of values found during training
- (e.g. can be estimated during training with running averages)

## Batch Normalization

- Typically normalize before the non-linearity (e.g. ReLU)
- Allows larger LR (faster learning), improves gradient flow and reduces dependence on initialization
- Doubles the layers, giving more learnable parameters



## Layer Normalization (2016)

- Shortly after batch normalization (Ba, Kiros, and Hinton)
- Rather than normalize a single feature across a batch, normalize the net value of each feature across the layer (the width of the feature vector)
- No need of a batch, do normalization for each instance
- Exact same at learning and execution time
- More common for Recurrent neural networks and transformers

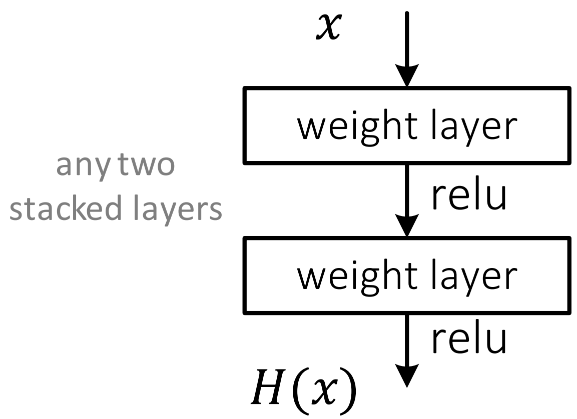
## Deep Residual Learning

- Residual Nets – 100s of layers
- 2015 ILSVRC winner
  - CNN
  - Also used Batch Normalization
- Learns the residual mapping with respect to the identity – i.e. the *difference* (residual) between the current input and the goal mapping
- Simple concept which tends to make the function to be learned simpler across depth

## Deep Residual Learning

### Deep Residual Learning

- Plain net

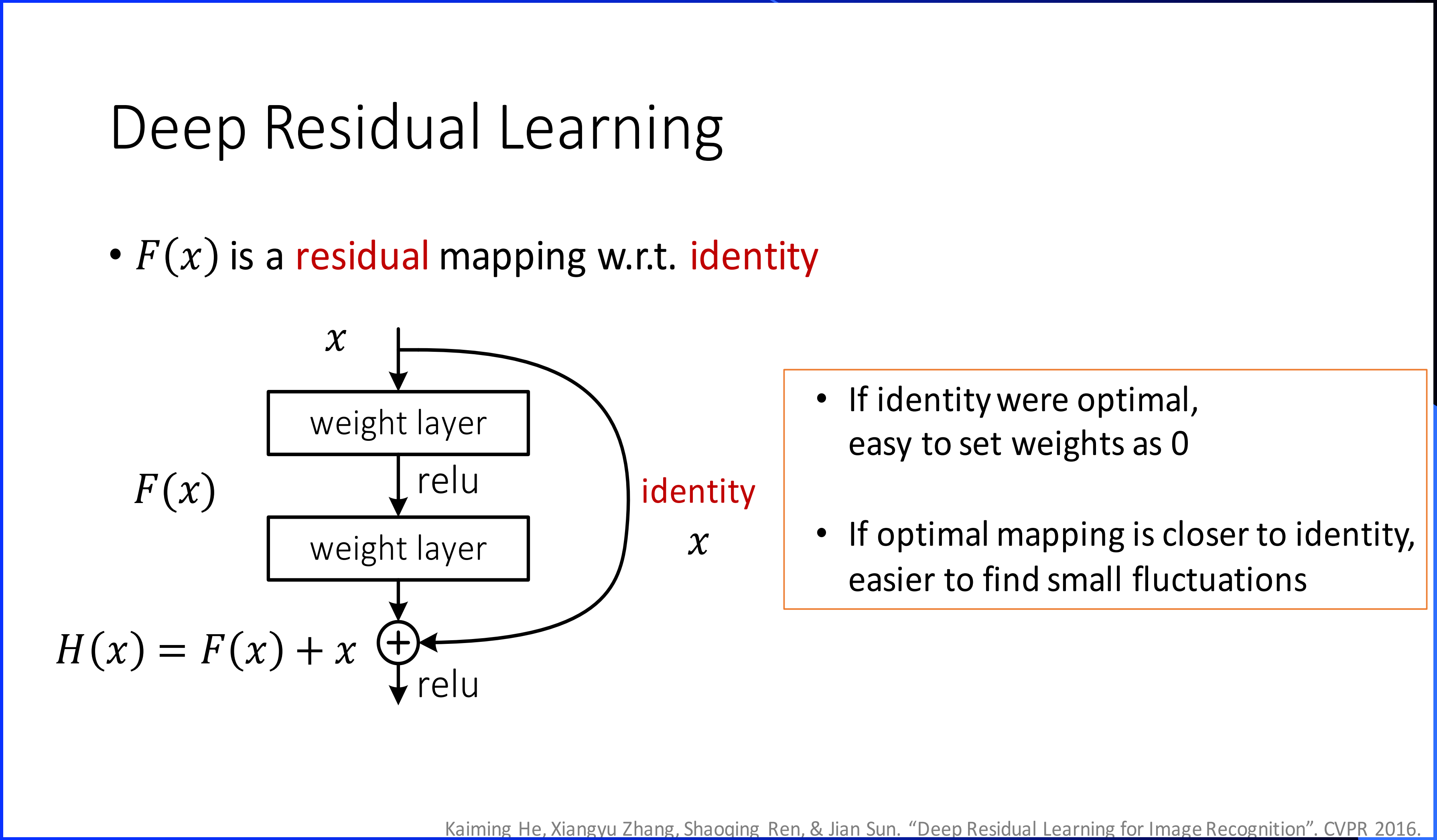


$H(x)$ is any desired mapping,

hope the 2 weight layers fit $H(x)$

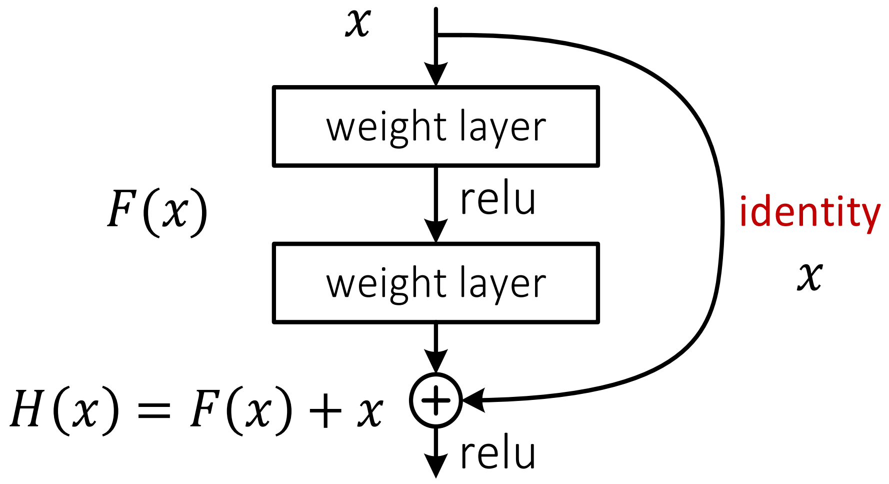
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

# Deep Residual Learning



## Deep Residual Learning

- $F(x)$ is a residual mapping w.r.t. identity



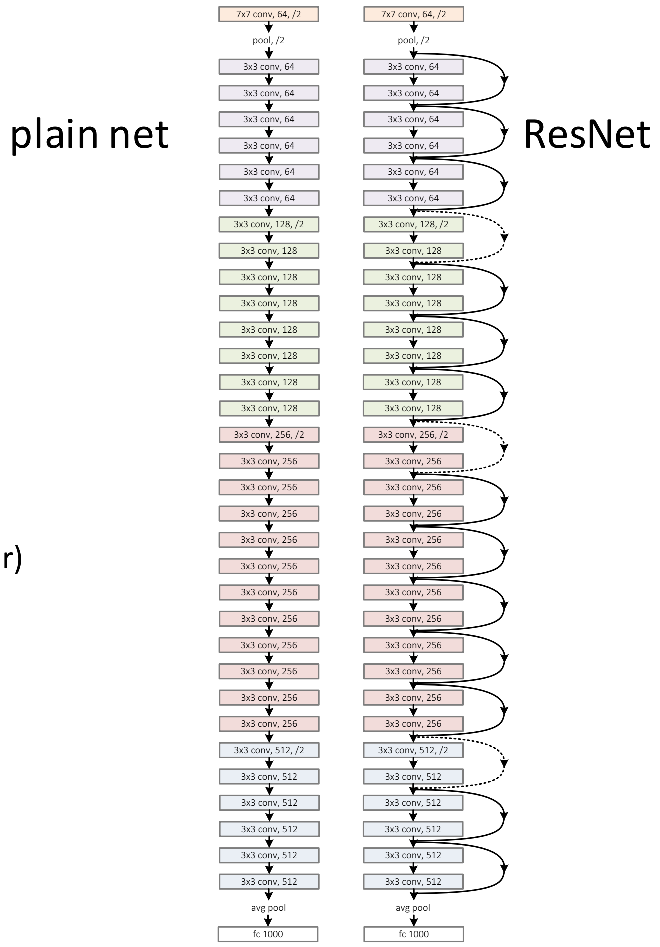
- If identity were optimal, easy to set weights as 0
- If optimal mapping is closer to identity, easier to find small fluctuations

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

# Deep Residual Learning

## Network "Design"

- Keep it simple
- Our basic design (VGG-style)
  - all 3x3 conv (almost)
  - spatial size /2  => # filters x2 (~same complexity per layer)
  - Simple design; just deep!
- Other remarks:
  - no hidden fc
  - no dropout



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

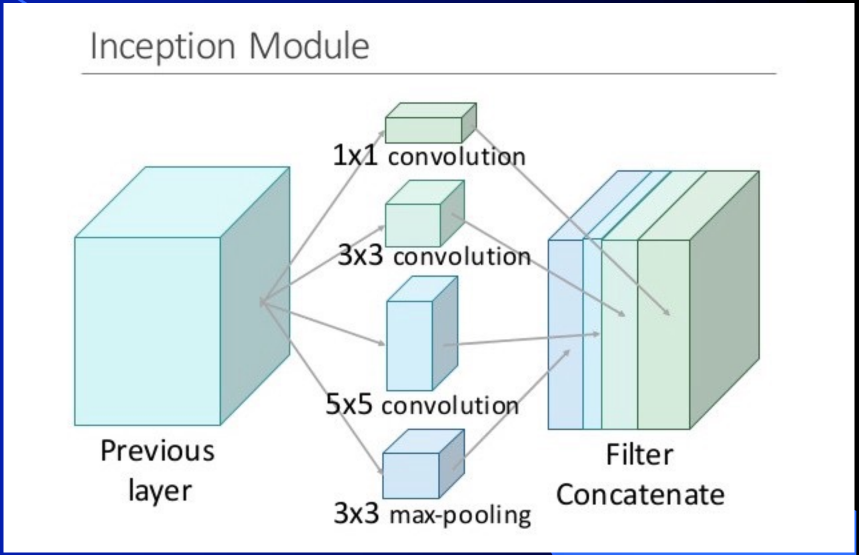
How many layers to add residuals, etc.? – Trial and error

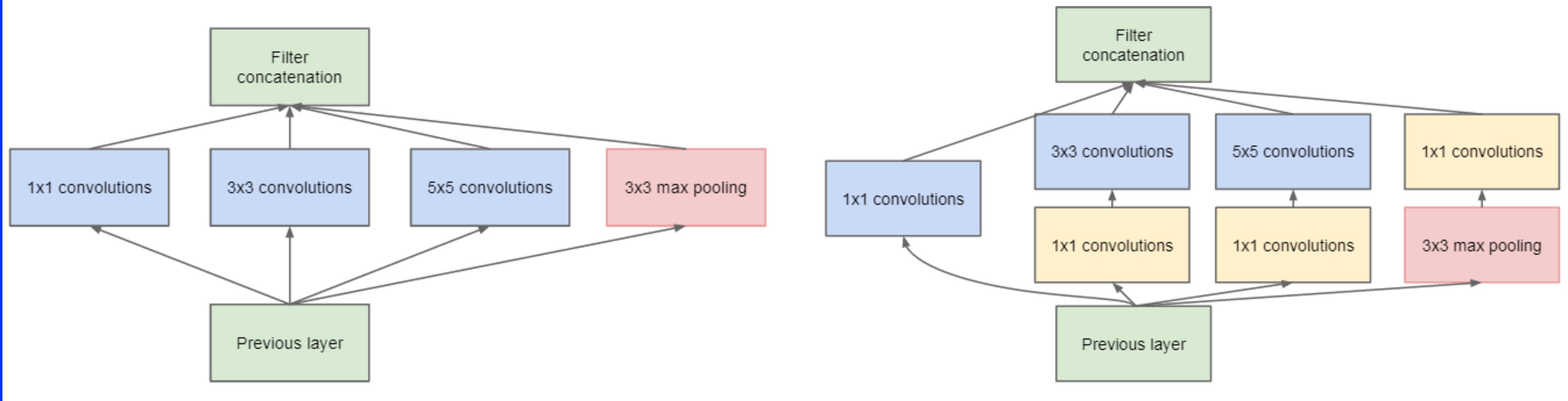
## Residual Nets

- Going from an $x$ to an $H(x)$ which is quite different from $x$ requires more learning, larger weights, etc.
- However, if $H(x)$ is similar to $x$, and assuming we start with small weights anyways, it takes a lot less updates to learn it – Easier!
- Adding $x$ to a later layer allows it to learn this simpler mapping
- Also, if the net had already learned a particular feature, we can just maintain that feature with 0 weights (since x will be added to our 0 output from 0 weights), without having to relearn it all the time – avoid "feature attrition”
  - Less worry about too many layers, since need enough to learn, then can retain it with residual (skip) connections

# Inception - Google

- "Network in a network" – CNN – Deeper and *Wider* - GoogLeNet
- Could replace the basically linear convolution with a more complex non-linear network – e.g. MLP
- Basic Inception does different size convolutions and combines results into one output
- Reduces added complexity by first doing dimensionality reduction with 1x1 convolution filters – 1 input from each preceding feature map, reduced to 1 value

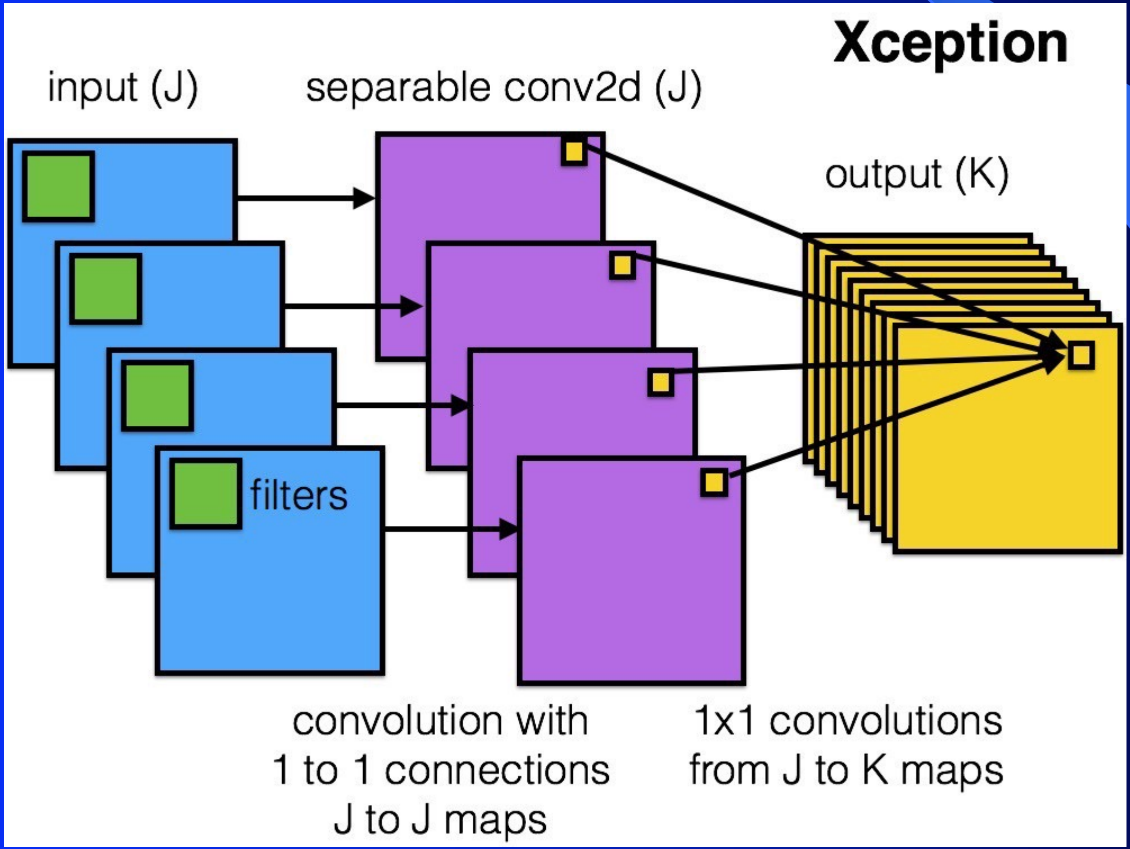




(a) Inception module, naïve version (b) Inception module with dimension reductions

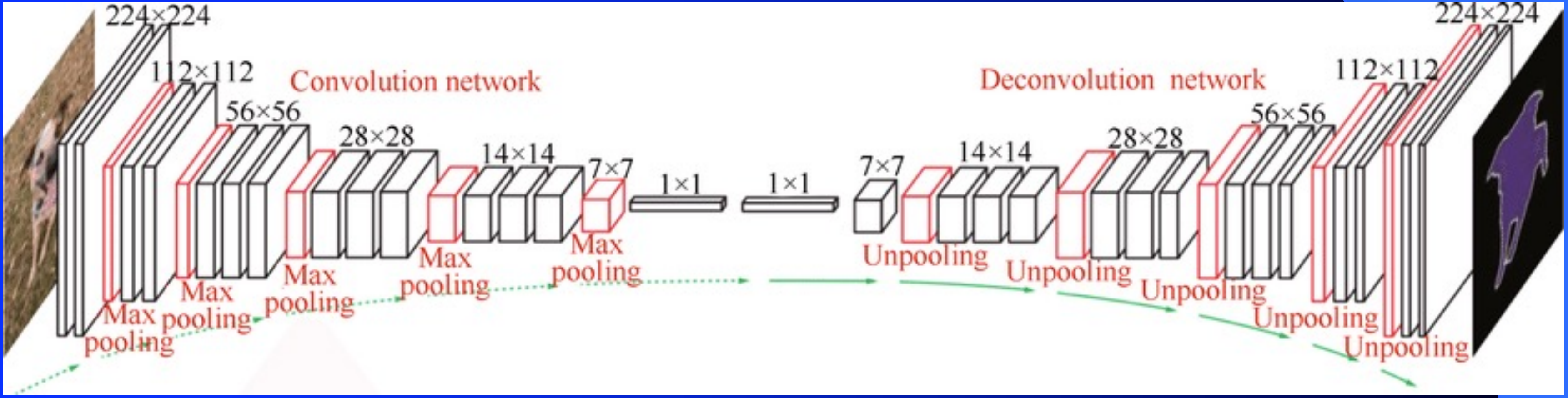
## Xception

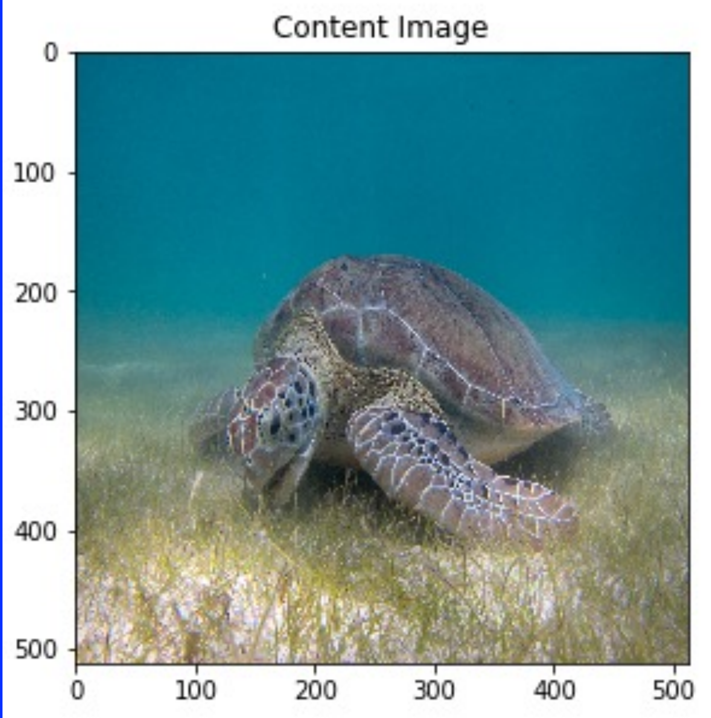
- Xception (extreme inception) decouples spatial and cross-channel correlations – channels only connected at extra layers of 1x1 convolutional maps - wider

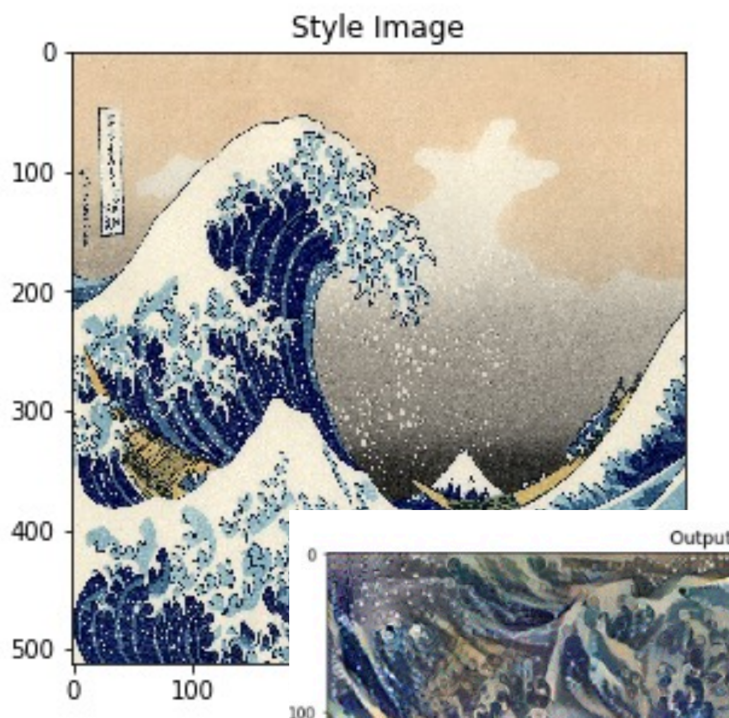


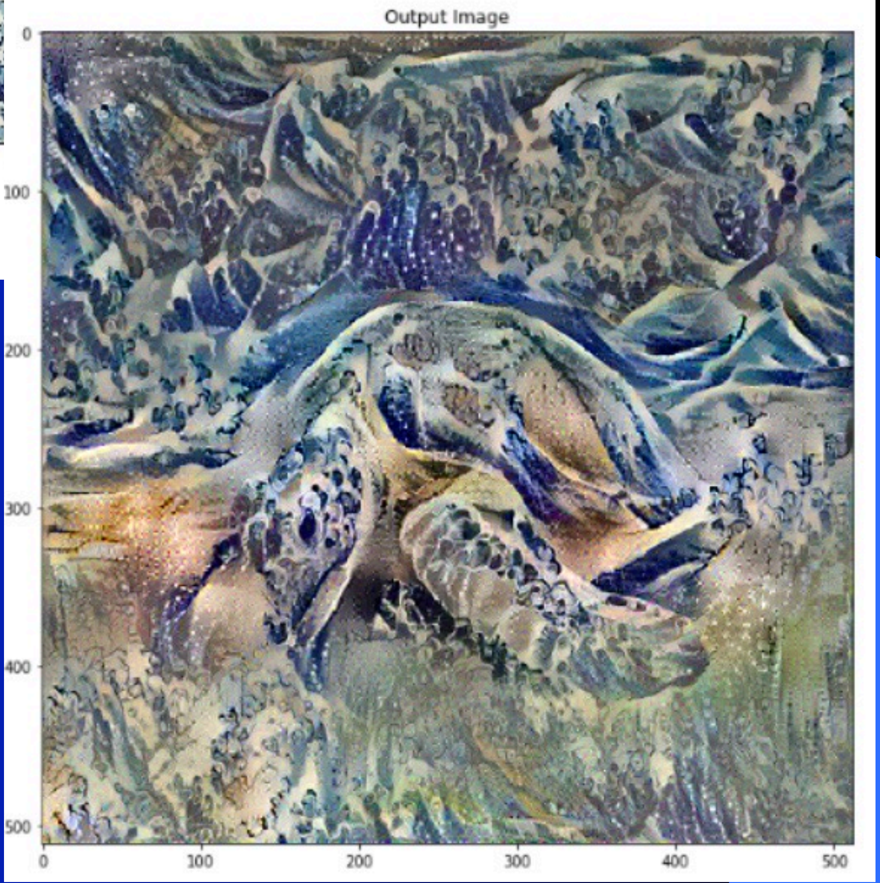
# Deep Generative Models

- Lots of research on generative models to create probabilistic models of training data with ability to generate new images, sentences, etc.
- Deconvolutional Neural Networks can generate images (deconvolution)





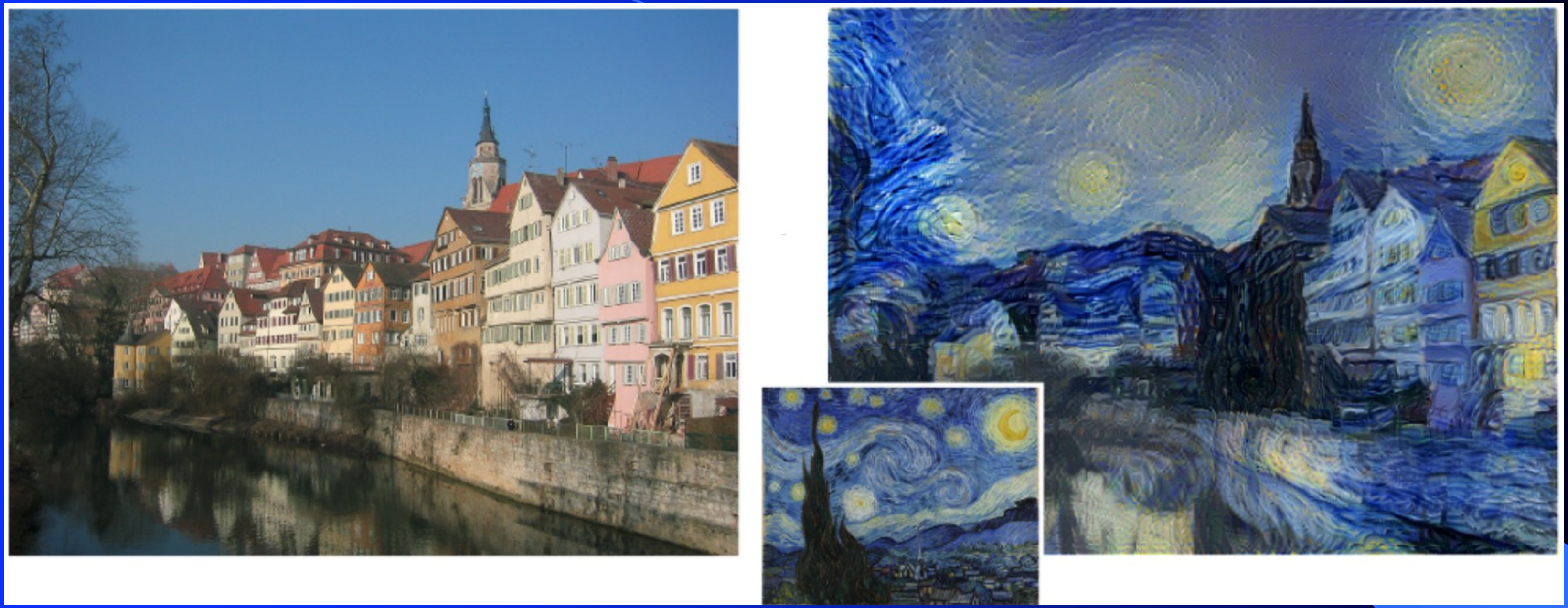


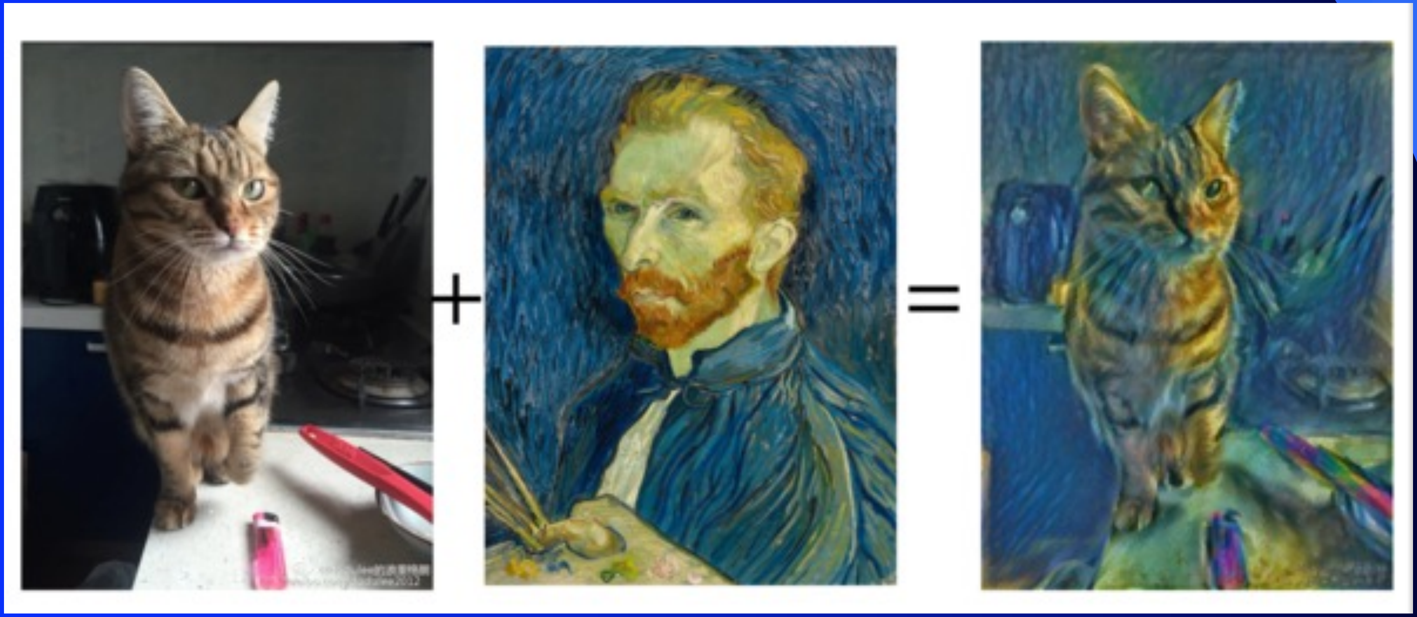


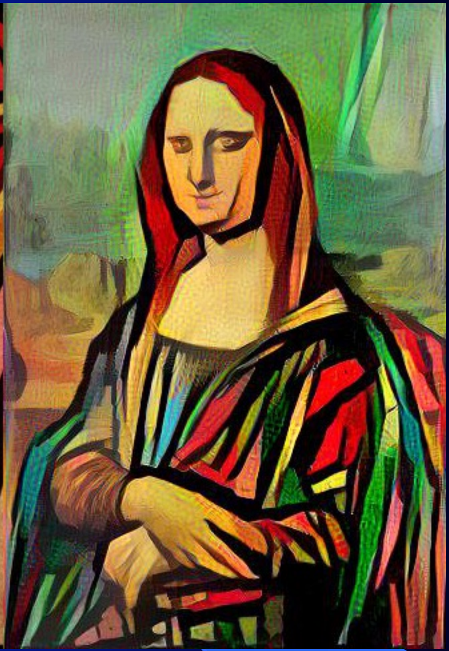
## Neural Style Transfer

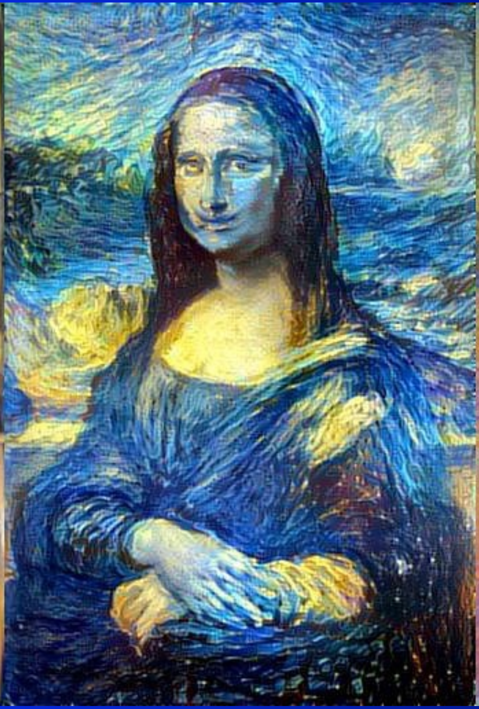
- Train on lots of images and styles
- CNN trained with two loss functions
  - content and style
- Then supply any content and style image
- Creates new image of content, but in the style of the style image

# Neural Style Transfer

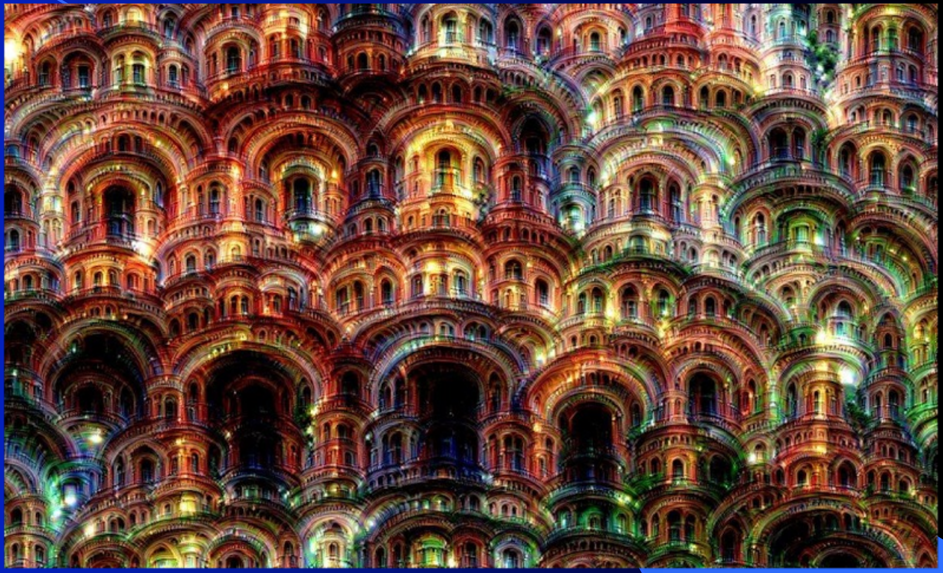
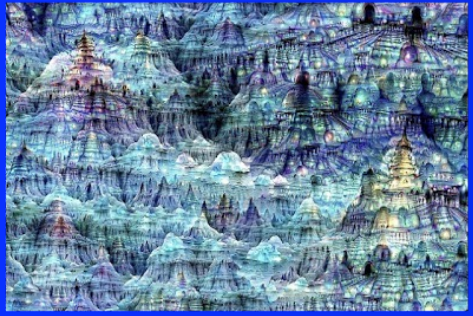


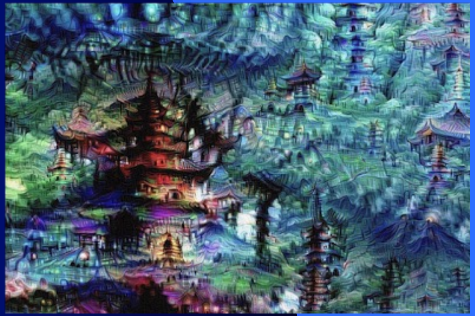






## Deep Dreaming

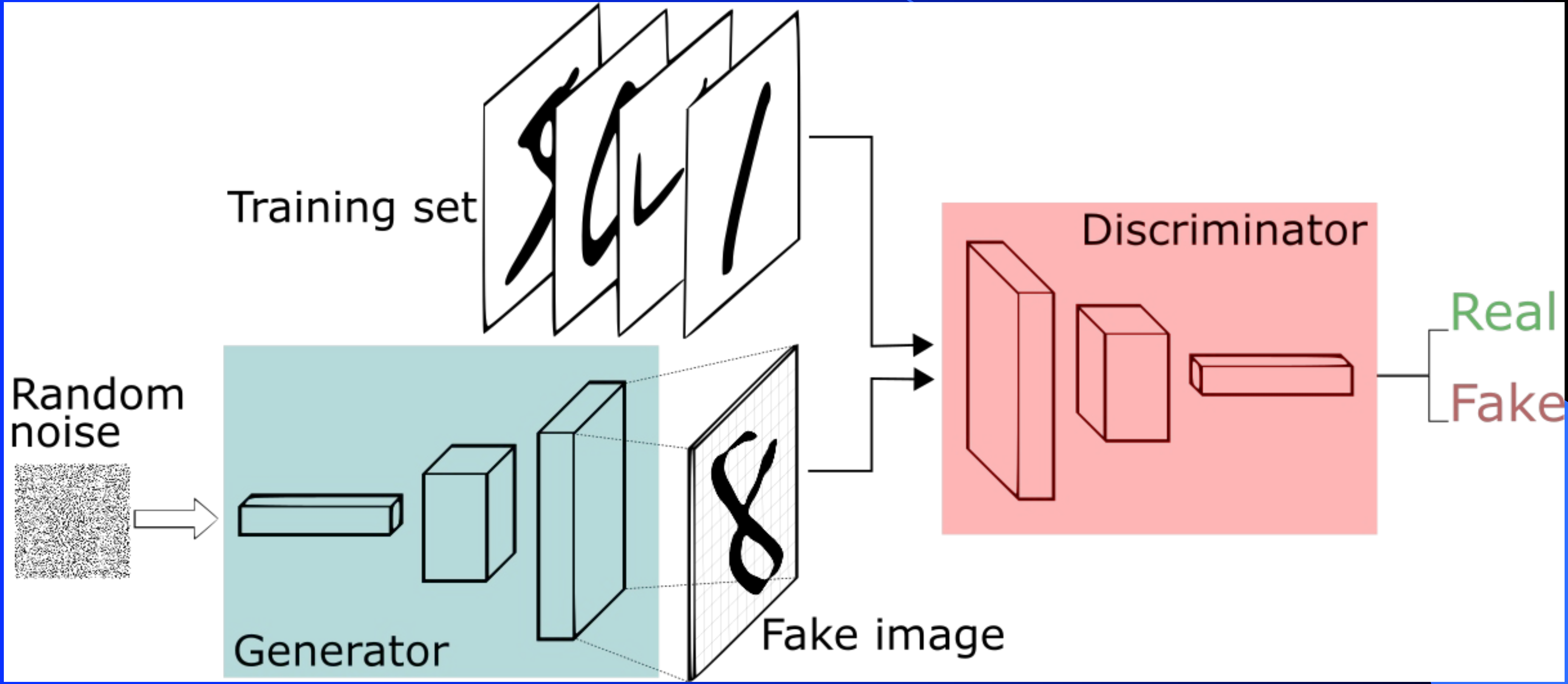



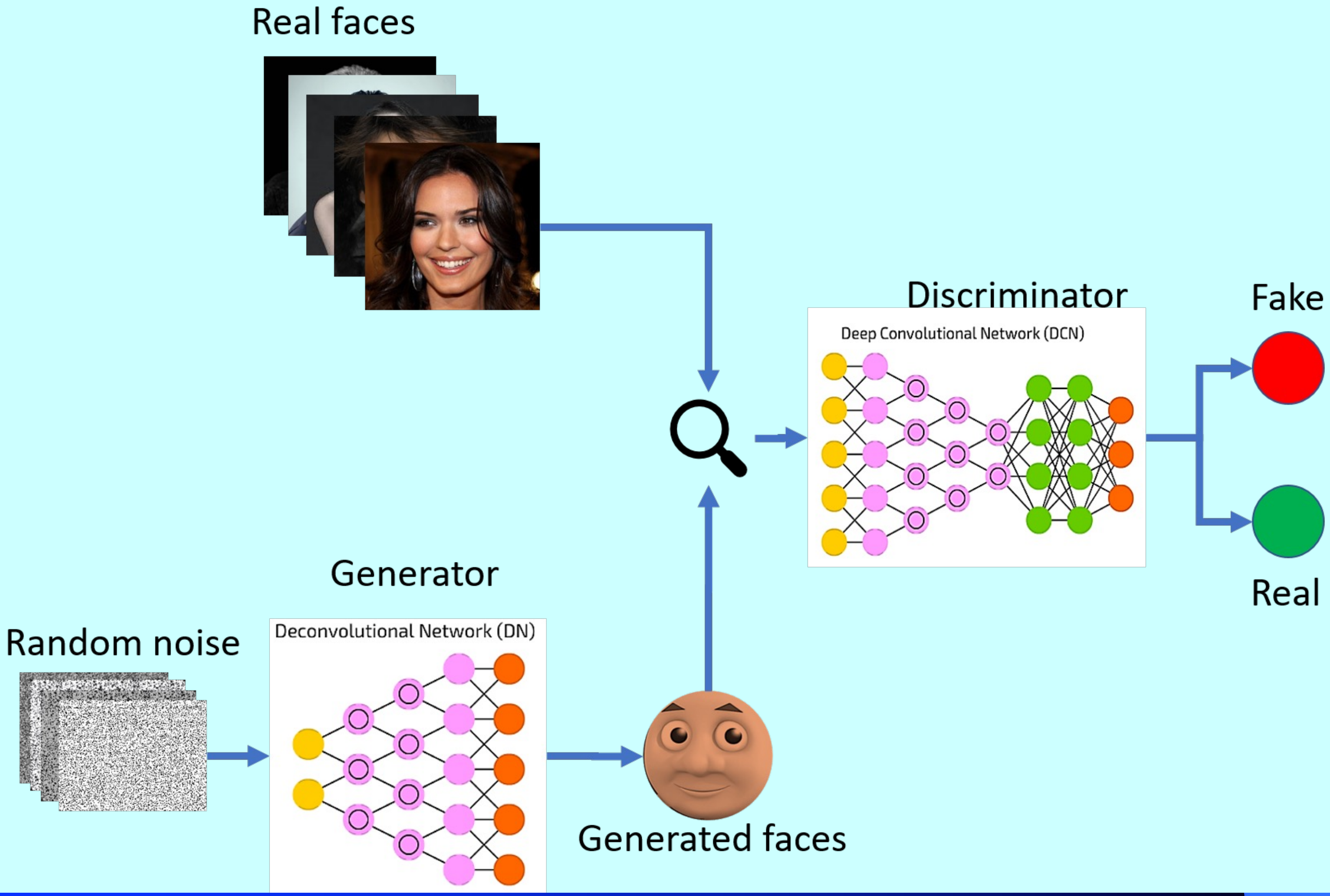


# Generative Adversarial Networks GANs (2014 Ian Goodfellow)

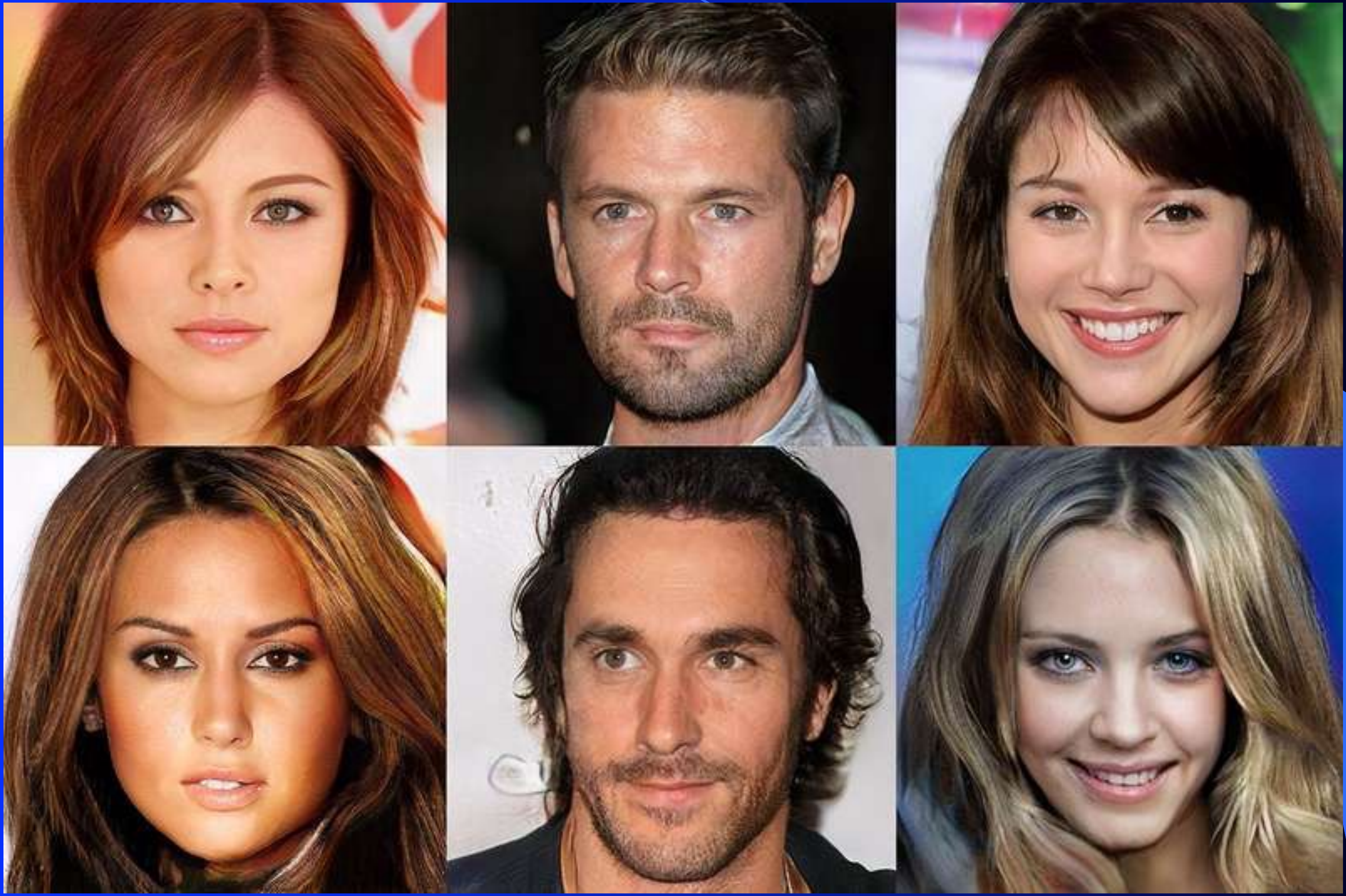
- Unsupervised in that no labels needed
- Generative networks which generate novel samples similar to training samples (images, text, etc.)
- Discriminative net (adversary) must differentiate between samples from the generative net and the training set
- Use loss feedback on discriminator net to create gradient for both nets, until discriminator can no longer distinguish, then can discard discriminator net – increasingly difficult for humans to distinguish
- "Universal loss function" for lots of "difficult/creative" applications

# GAN - Generative Adversarial Network

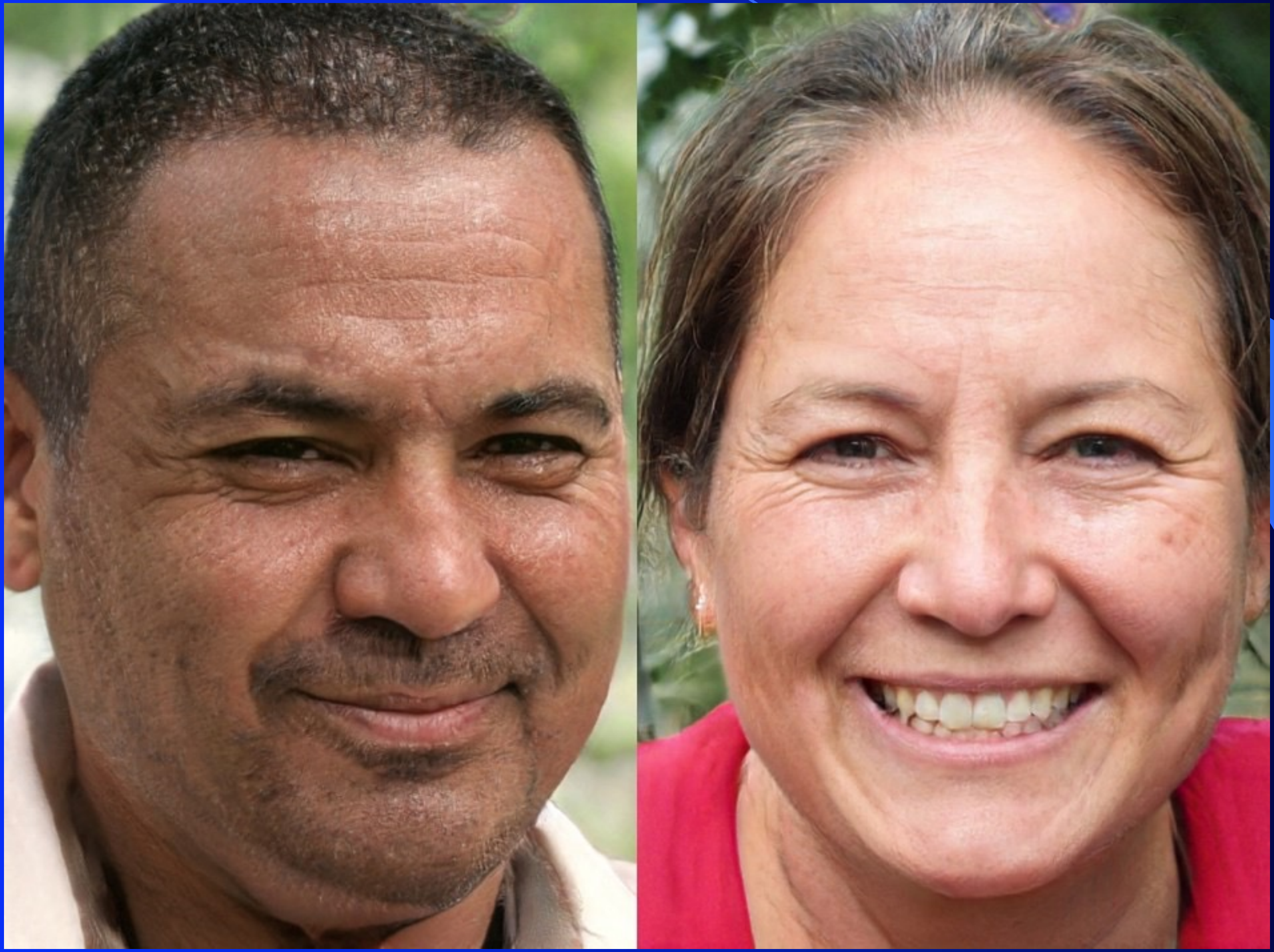



Real faces
Discriminator
Deep Convolutional Network (DCN)
Fake
Real
Generator
Random noise
Deconvolutional Network (DN)
Generated faces

## GAN – Celebrity Data Set (2017)

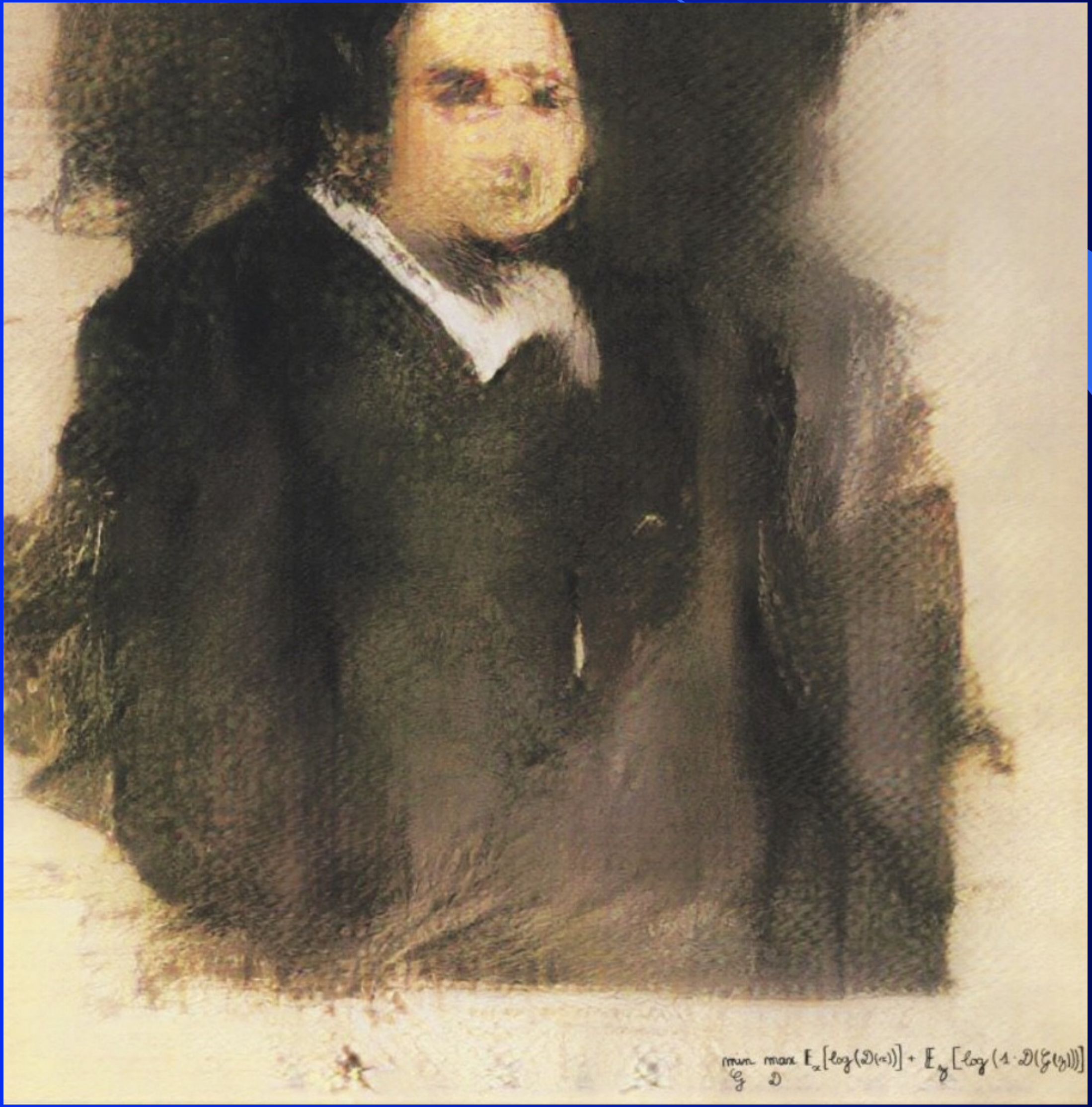


## GAN 2.0 NVIDIA (2018) - Improved Face Generator



## Edmond de Belamy

- Created by a GAN and sold at auction in 2018 for $432,500
- Note the author inscription – it is the GAN loss function



2021 GauGan2: Create your own images by sketching or by just typing in a description of the image you want to create

*An output from GauGAN2 for the phrase "coast ripples cliffs."*

## GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models

"a hedgehog using a calculator"

"a corgi wearing a red bowtie and a purple party hat"

"robots meditating in a vipassana retreat"

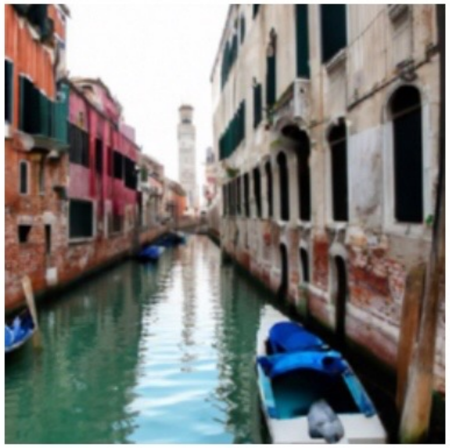
"a fall landscape with a small cottage next to a lake"

"a surrealist dream-like oil painting by salvador dalí of a cat playing checkers"

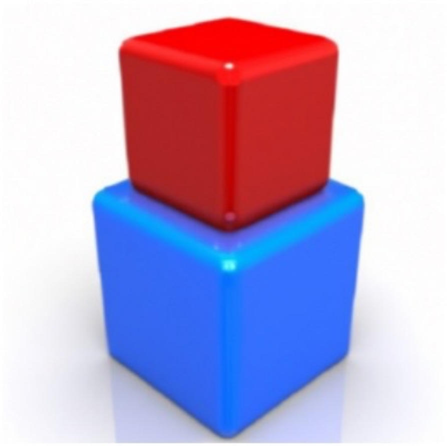
"a professional photo of a sunset behind the grand canyon"

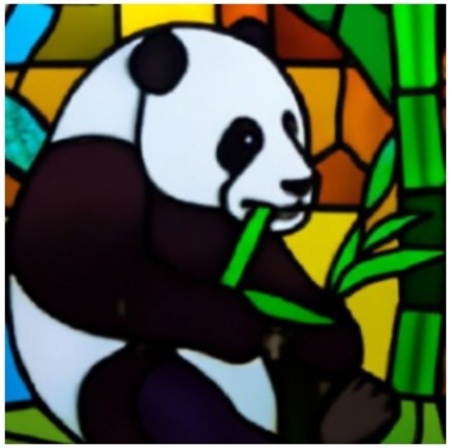
"a high-quality oil painting of a psychedelic hamster dragon"

"an illustration of albert einstein wearing a superhero costume"


"a boat in the canals of venice"

"a painting of a fox in the style of starry night"


"a red cube on top of a blue cube"


"a stained glass window of a panda eating bamboo"

"a crayon drawing of a space elevator"

"a futuristic city in synthwave style"

"a pixel art corgi pizza"

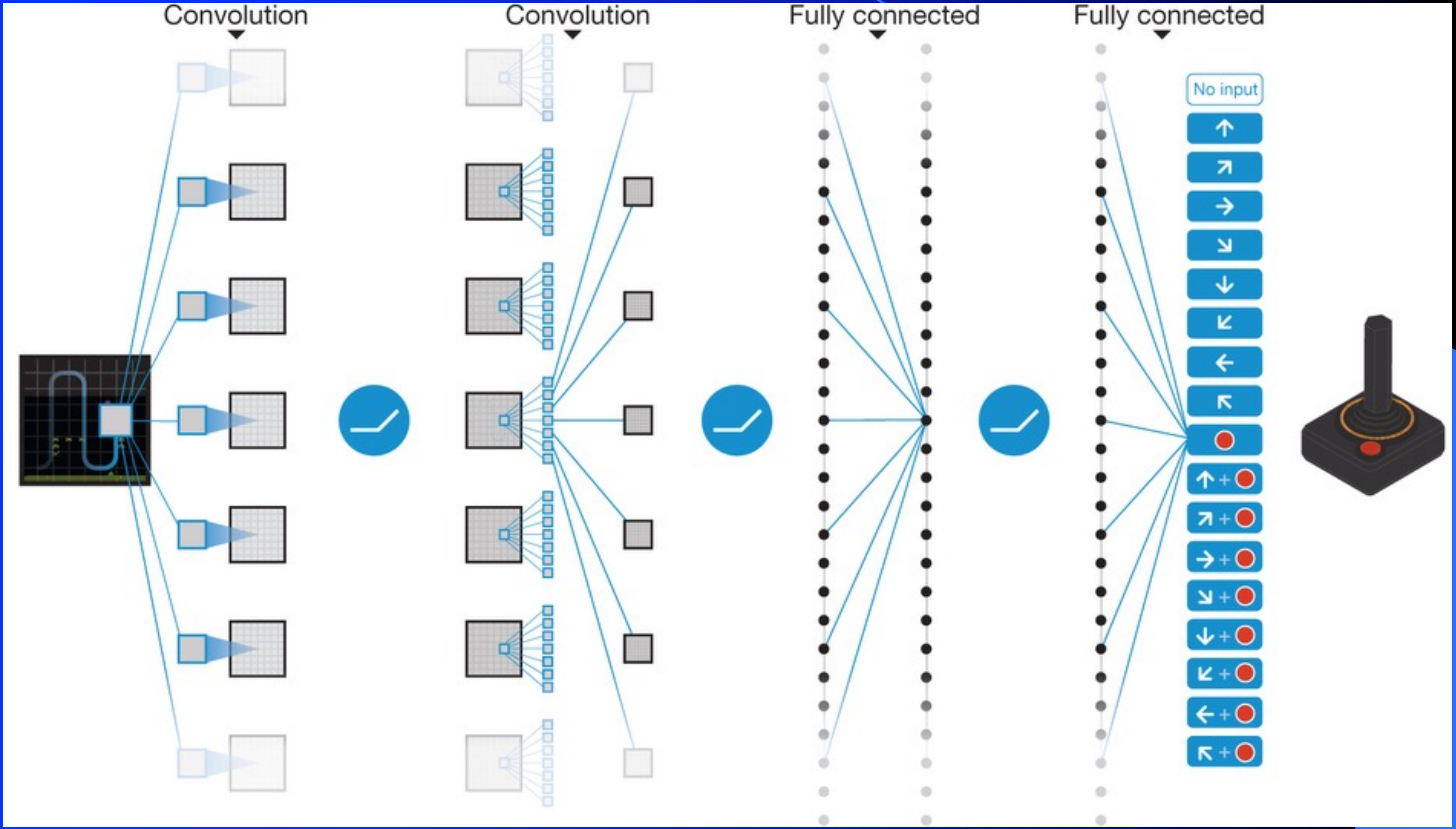
"a fog rolling into new york"

## Real vs Fake?

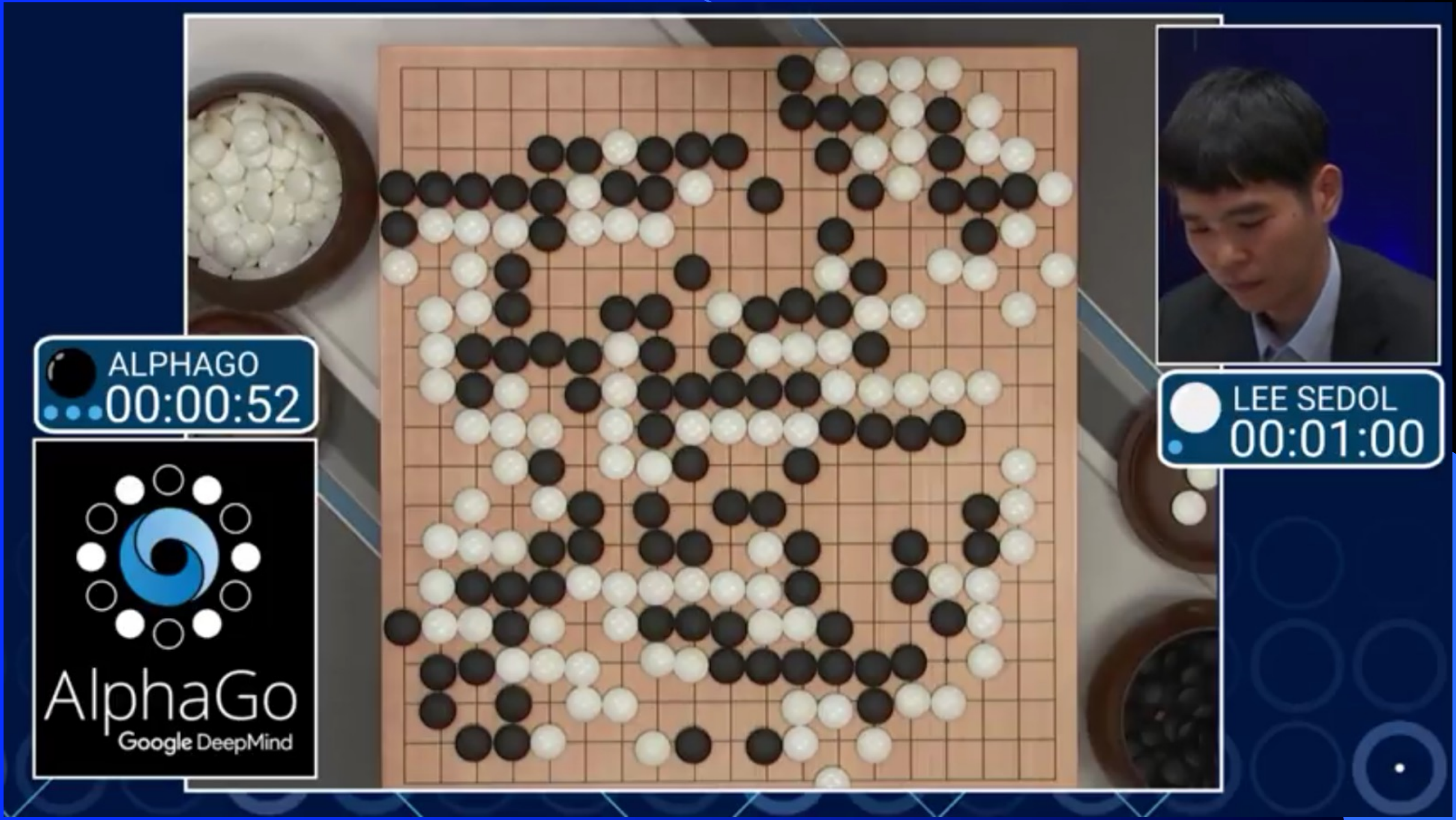
President Russel M Nelson:

"If we are to have any hope of sifting through the myriad of voices and the philosophies of men that attack truth, we must learn to receive revelation.
In coming days, it will not be possible to survive spiritually without the guiding, directing, comforting, and constant influence of the Holy Ghost."

## Deep Reinforcement Learning: Deep Q Network – 49 Classic Atari Games



# AlphaGo - Google DeepMind



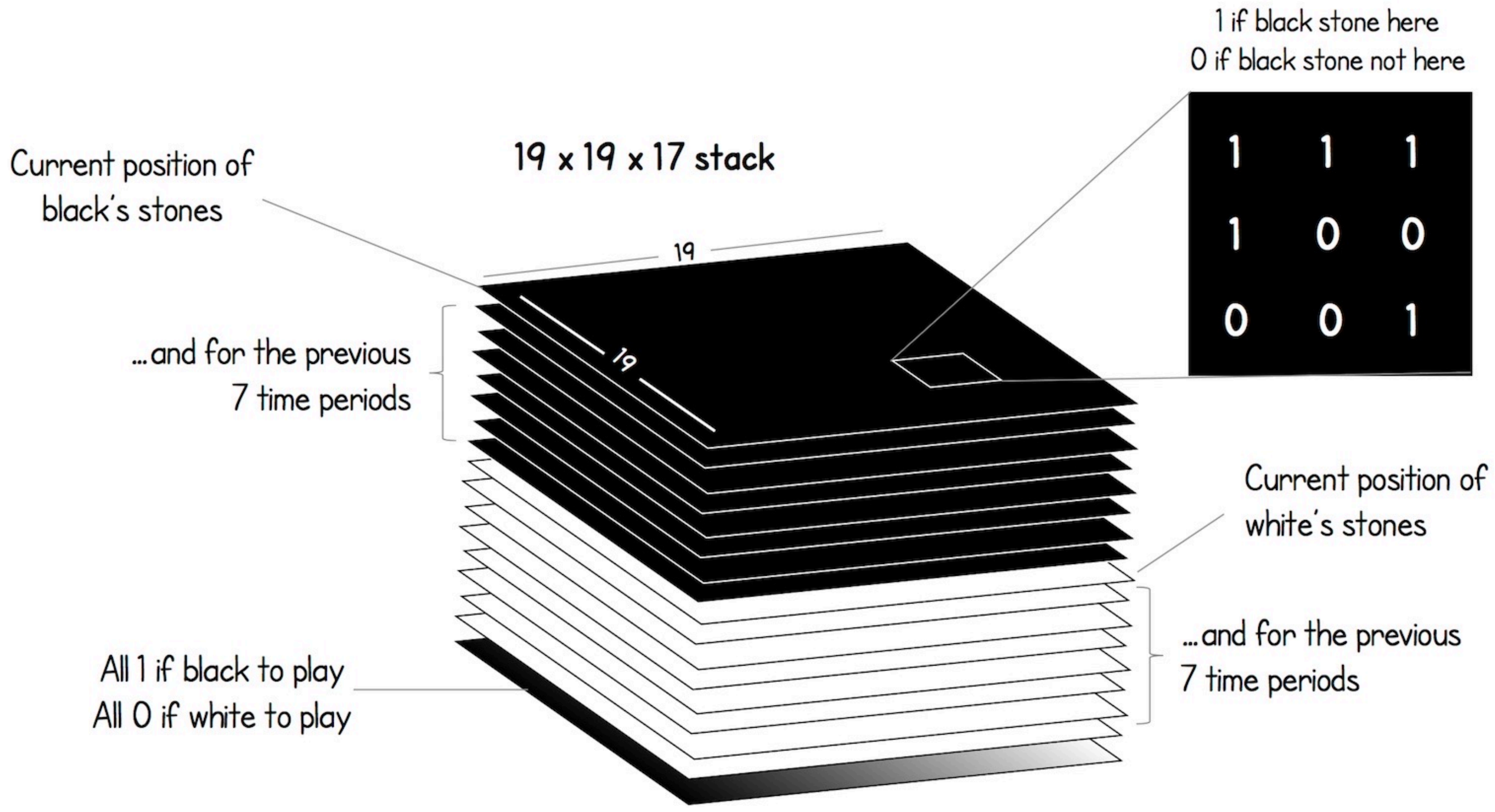
# Alpha Go

- Reinforcement Learning with Deep Net learning the value and policy functions
- Challenges world Champion Lee Se-dol in March 2016
  - AlphaGo Movie – Netflix, check it out, fascinating man/machine interaction!
- AlphaGo Master (improved with more training) then beat top masters on-line 60-0 in Jan 2017
- 2017 – Alpha Go Zero
  - Alpha Go started by learning from 1000's of expert games before learning more on its own, and with lots of expert knowledge
  - Alpha Go Zero starts from zero (Tabula Rasa), just gets rules of Go and starts playing itself to learn how to play – not patterned after human play – More creative
  - Beat AlphaGo Master 100 games to 0 (after 3 days of playing itself)

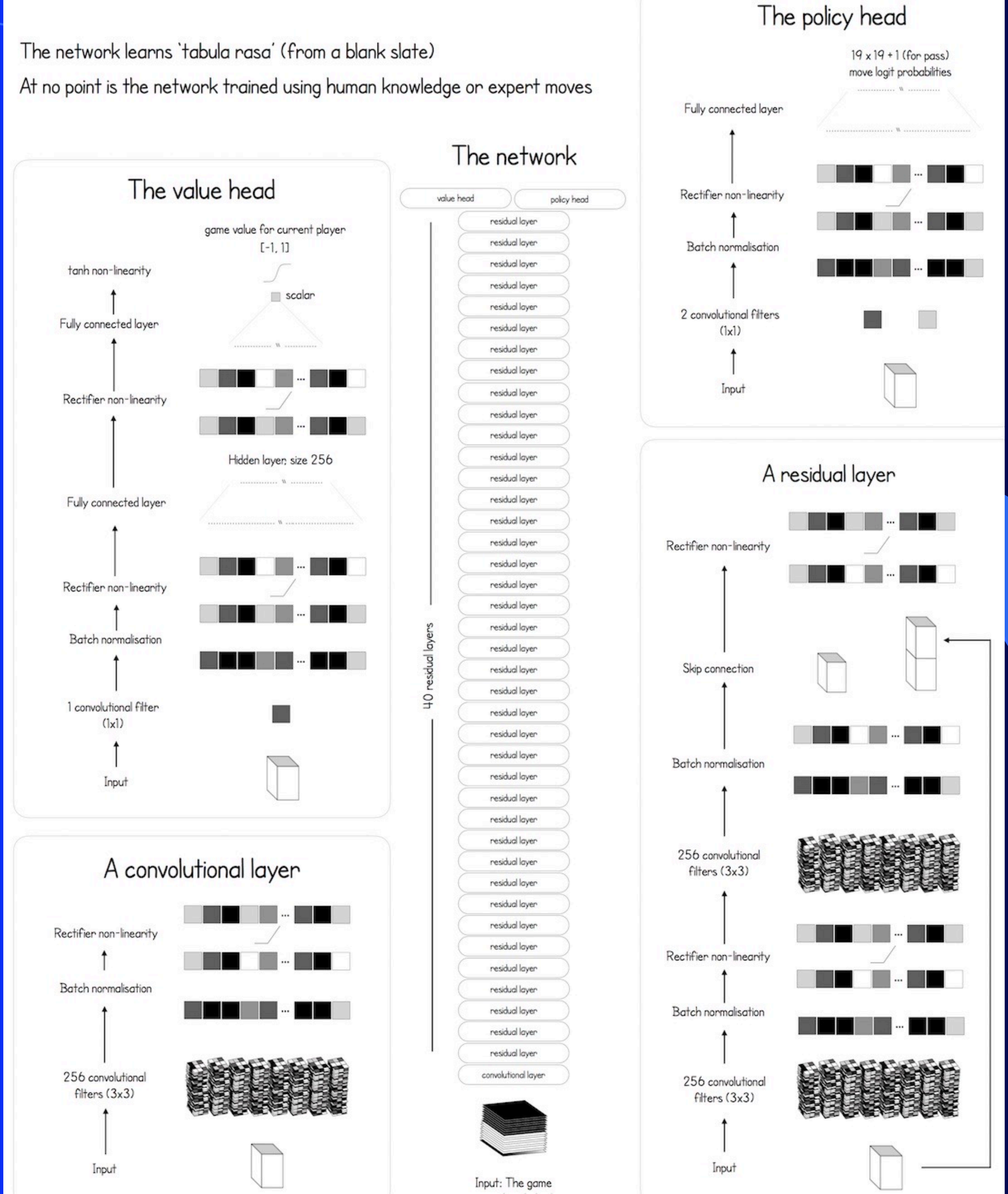
## Alpha Zero

- Alpha Zero (late 2017)
- Generic architecture for any board game
  - Compared to AlphaGo (2016 - earlier world champion with extensive background knowledge) and AlphaGo Zero (2017)
- No input other than rules and self-play, <u>and</u> not set up for any specific game, except different board input
- With no domain knowledge and starting from random weights, beats worlds best players and computer programs (which were specifically tuned for their games over many years)
  - Go – after 8 hours training (44 million games) beats AlphaGo Zero (which had beat AlphaGo 100-0) – 1000's of TPU's for training
    - AlphaGo had taken many months of human directed training
  - Chess – after 4 hours training beats Stockfish8 28-0 (+72 draws)
    - Doesn't pattern itself after human play
  - Shogi (Japanese Chess) – after 2 hours training beats Elmo

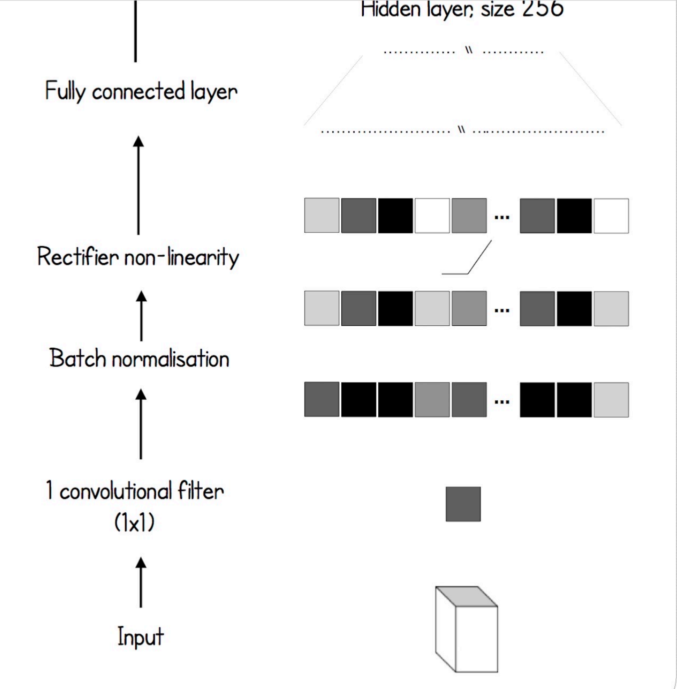
# WHAT IS A 'GAME STATE'


1 if black stone here
0 if black stone not here
19 x 19 x 17 stack
Current position of black's stones
19
19
1 1 1
1 0 0
0 0 1
...and for the previous 7 time periods
Current position of white's stones
...and for the previous 7 time periods
All 1 if black to play
All 0 if white to play

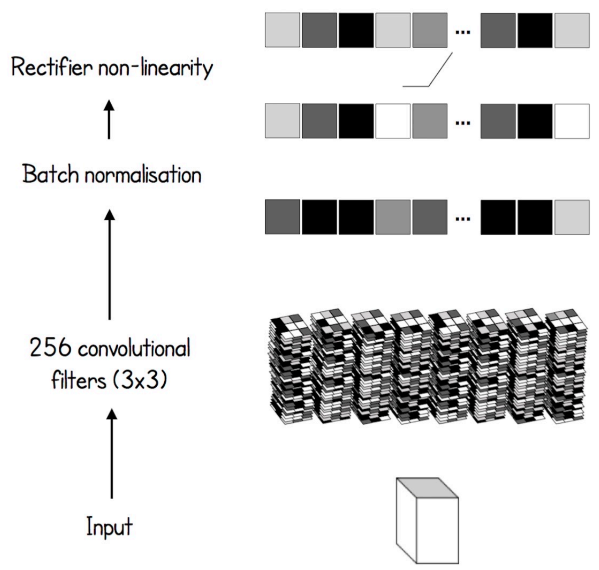
This stack is the input to the deep neural network

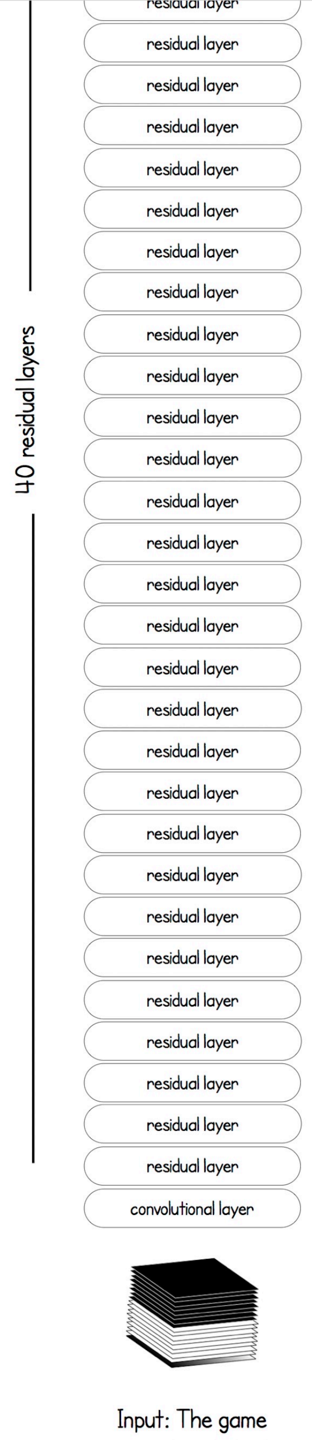

The network learns 'tabula rasa' (from a blank slate)

At no point is the network trained using human knowledge or expert moves

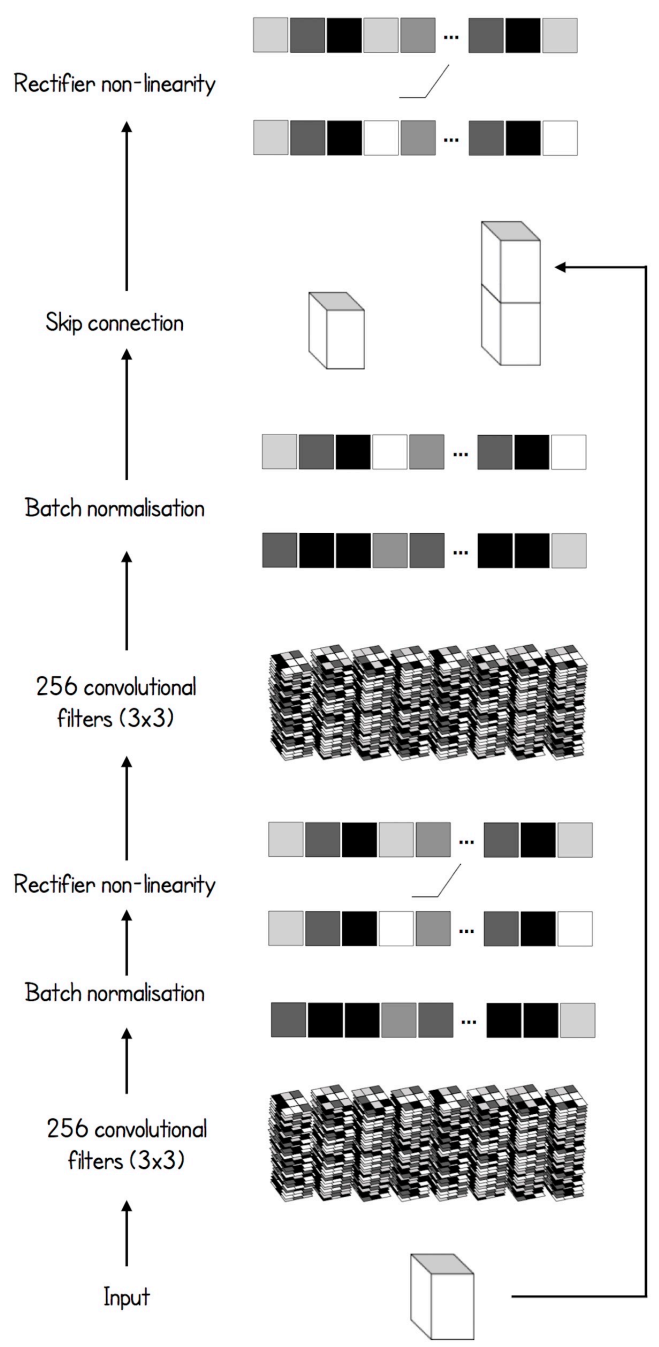


## A convolutional layer



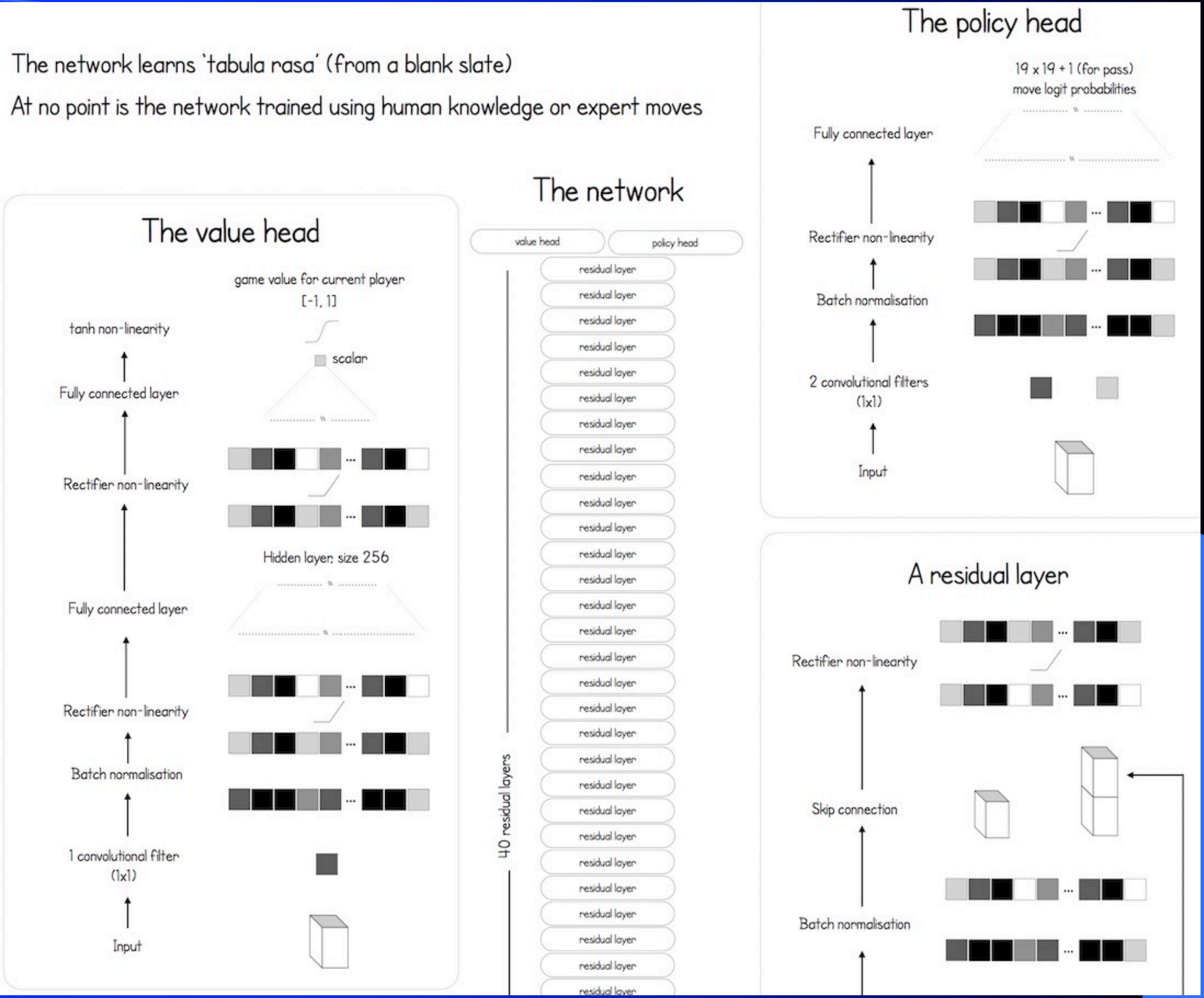


## A residual layer



The network learns 'tabula rasa' (from a blank slate)

At no point is the network trained using human knowledge or expert moves



The training pipeline for AlphaGo Zero consists of three stages, executed in parallel

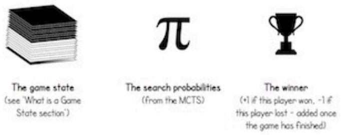
## SELF PLAY

Create a 'training set'

The best current player plays 25,000 games against itself

See MCTS section to understand how AlphaGo Zero selects each move

At each move, the following information is stored


The game state
(see 'What is a Game State section')
The search probabilities
(from the MCTS)
The winner
(+1 if this player won, -1 if this player lost - added once the game has finished)

## RETRAIN NETWORK

Optimise the network weights

A TRAINING LOOP

Sample a mini-batch of 2048 positions from the last 500,000 games

Retrain the current neural network on these positions
- The game states are the input (see 'Deep Neural Network Architecture')

Loss function
Compares predictions from the neural network with the search probabilities and actual winner

PREDICTIONS $p$ Cross-entropy $\pi$ ACTUAL
+
$v$ Mean-squared error
+
Regularisation

After every 1,000 training loops, evaluate the network

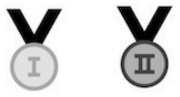
## EVALUATE NETWORK

Test to see if the new network is stronger

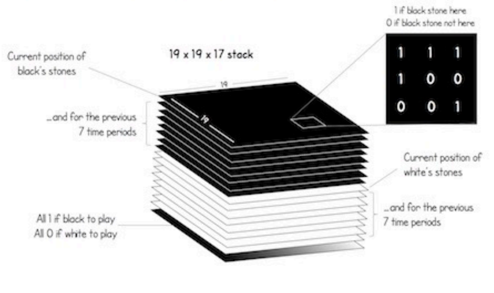
Play 400 games between the latest neural network and the current best neural network

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes

Latest player must win 55% of games to be declared the new best player



## WHAT IS A 'GAME STATE'


1 if black stone here
0 if black stone not here
19 x 19 x 17 stack
Current position of black's stones
...and for the previous 7 time periods
19
19
1 1 1
1 0 0
0 0 1
Current position of white's stones
...and for the previous 7 time periods
All 1 if black to play
All 0 if white to play

This stack is the input to the deep neural network

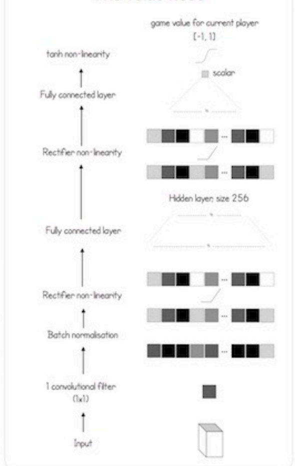
## THE DEEP NEURAL NETWORK ARCHITECTURE

How AlphaGo Zero assesses new positions

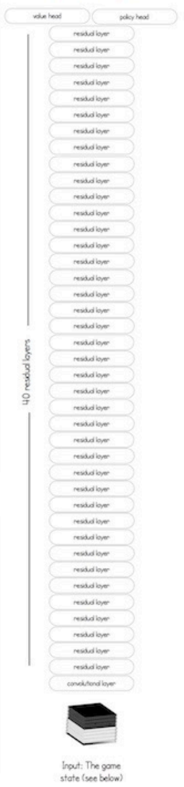
The network learns 'tabula rasa' (from a blank slate)

At no point is the network trained using human knowledge or expert moves

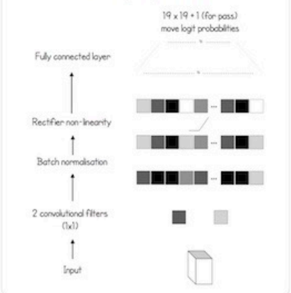
### The value head



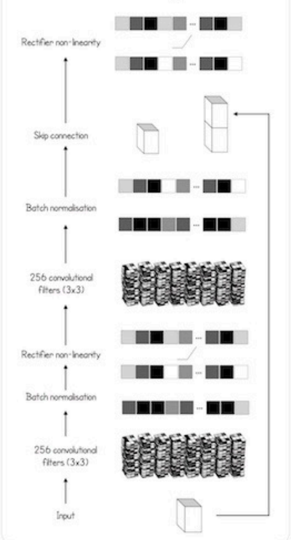
### The network



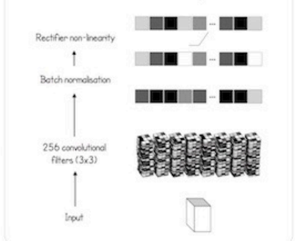
### The policy head



### A residual layer

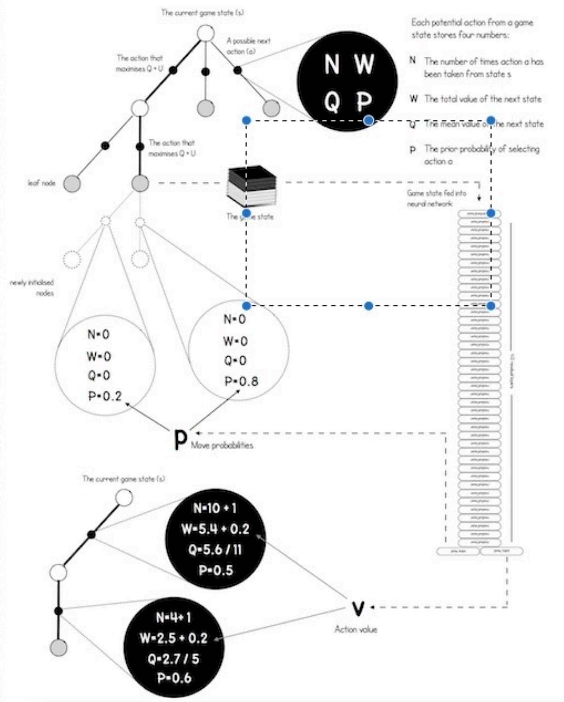


### A convolutional layer



## MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move


The current game state (s)
A possible next action (a)
The action that maximises Q + U
N W
Q P
Each potential action from a game state stores four numbers:
N The number of times action a has been taken from state s
W The total value of the next state
Q The mean value of the next state
P The prior probability of selecting action a
The action that maximises Q + U
leaf node
The game state
Game state fed into neural network
newly initialised nodes
N=0
W=0
Q=0
P=0.2
N=0
W=0
Q=0
P=0.8
p Move probabilities
The current game state (s)
N=10 + 1
W=5.4 + 0.2
Q=5.6 / 11
P=0.5
N=4+1
W=2.5 + 0.2
Q=2.7 / 5
P=0.6
v Action value

### First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

1. Choose the action that maximises...

$$Q + U$$

The mean value of the next state

A function of **P** and **N** that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation, U dominates (more exploration), but later, Q is more important (less exploration)

2. Continue until a leaf node is reached

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

**p** Move probabilities

**v** Value of the state (for the current player)

The move probabilities p are attached to the new feasible actions from the leaf node

3. Backup previous edges

Each edge that was traversed to get to the leaf node is updated as follows:

$$N \rightarrow N + 1$$
$$W \rightarrow W + v$$
$$Q = W / N$$

### ...then select a move

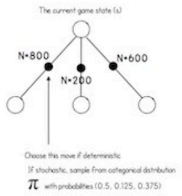
After 1,600 simulations, the move can either be chosen:

**Deterministically** (for competitive play)
Choose the action from the current state with greatest **N**

**Stochastically** (for exploratory play)
Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where $\tau$ is a temperature parameter, controlling exploration


The current game state (s)
N=800
N=200
N=600
Choose this move if deterministic
If stochastic, sample from categorical distribution
π with probabilities (0.5, 0.125, 0.375)

### Other points

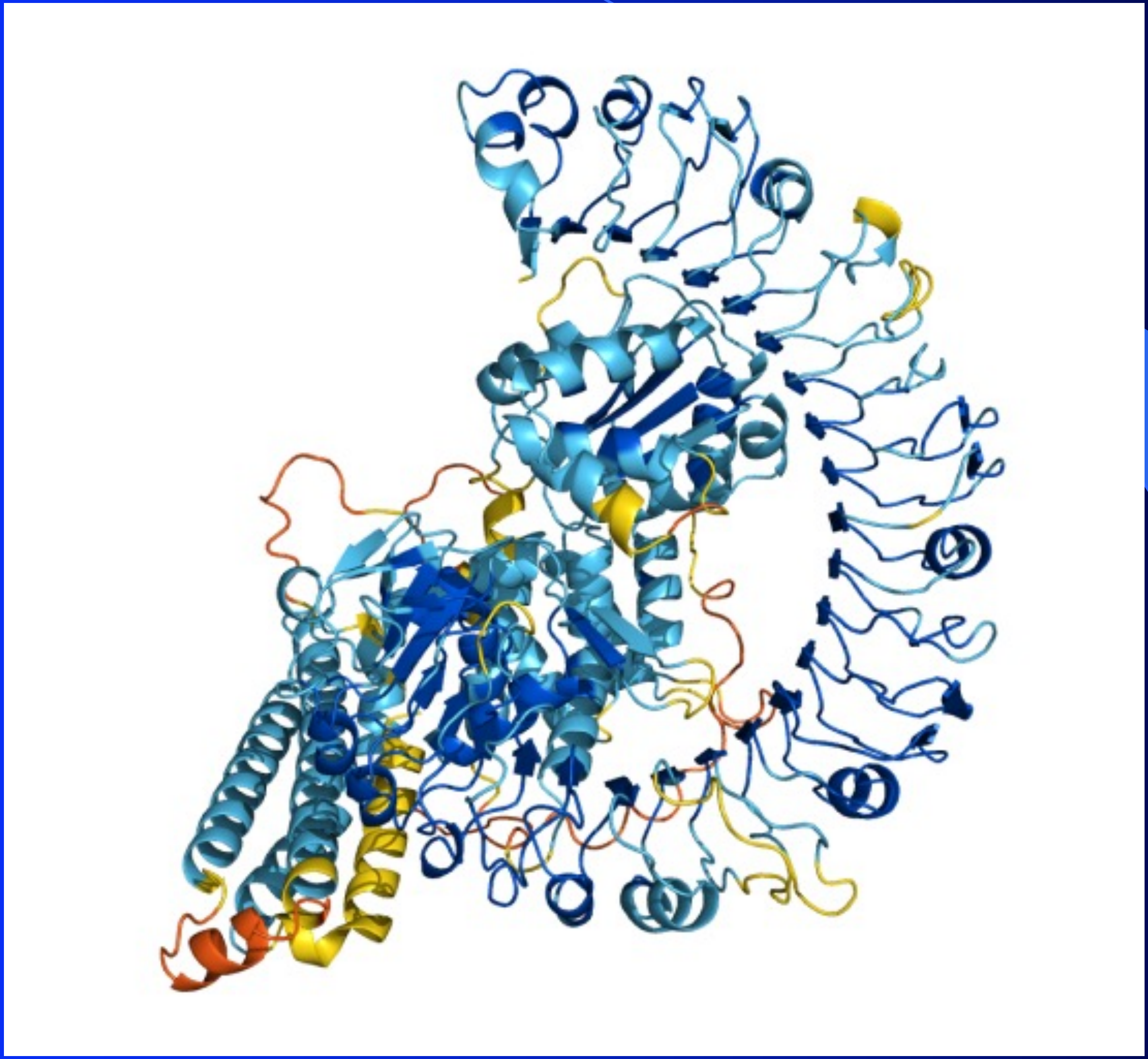
- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

## AlphaStar

- DeepMind considered "perfect information" board games solved
- Next step was – Starcraft II - AlphaStar
  - Considered a next "Grand AI Challenge"
  - Complex, long-term strategy, stochastic, hidden info, real-time
  - Beats best Pros - AlphaStar limited to human speed in actions/clicks per minute – so just comparing strategy

Workers: 3/3
Workers: 25/16
Workers: 3/3
14:32
Catalyst LE
0 AlphaStar 177 /200 945 +2015 758 +873 64 113 940
SUPPLY MINERALS GAS WORKERS ARMY APM PRODUCTION
0 LiquidTLO 147 /172 335 +1595 442 +1030 61 86 1377

## AlphaFold – Moved to Transformers, World's most accurate protein folding



# Recurrent Neural Networks

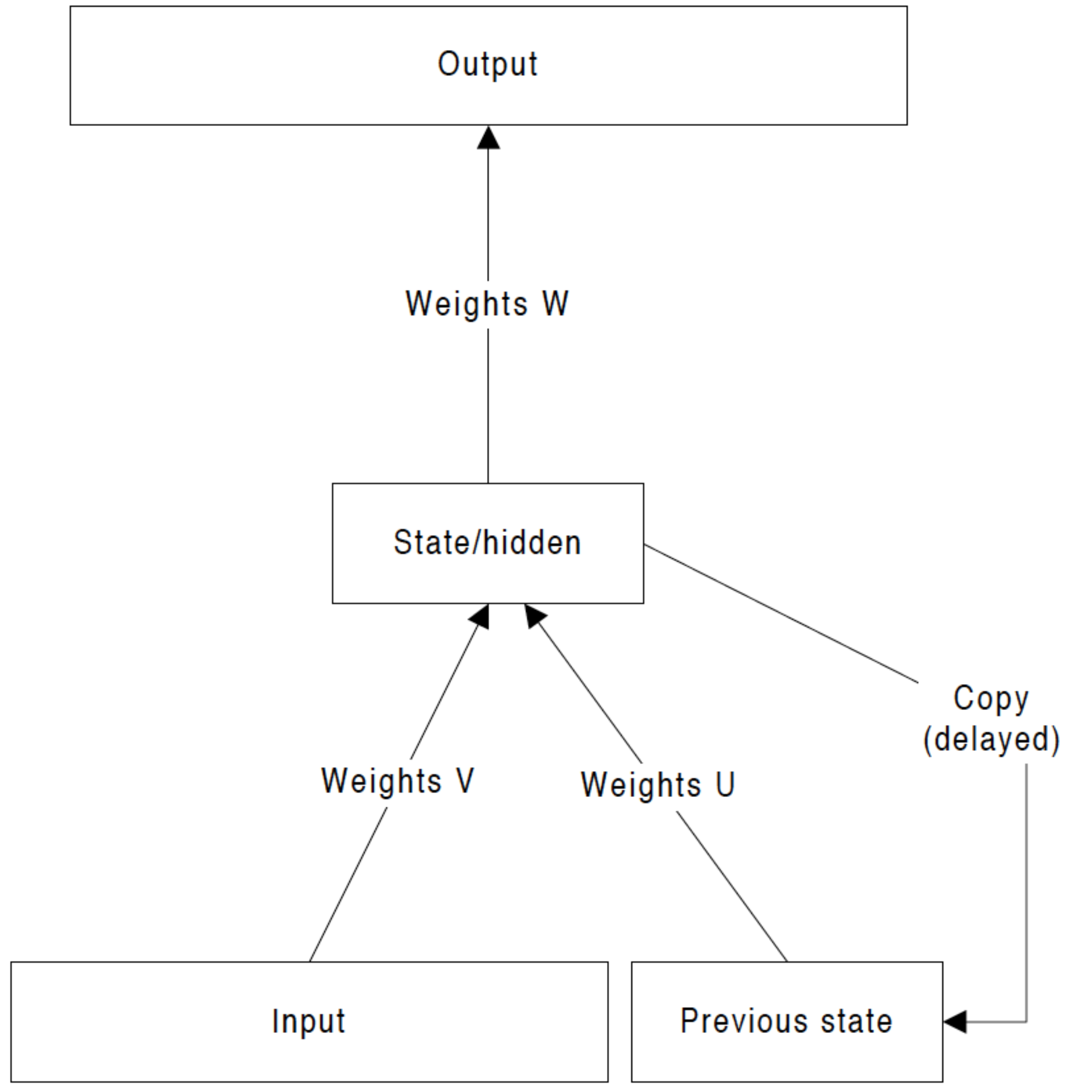


Figure 4: A simple recurrent network.

# Recurrent Neural Networks

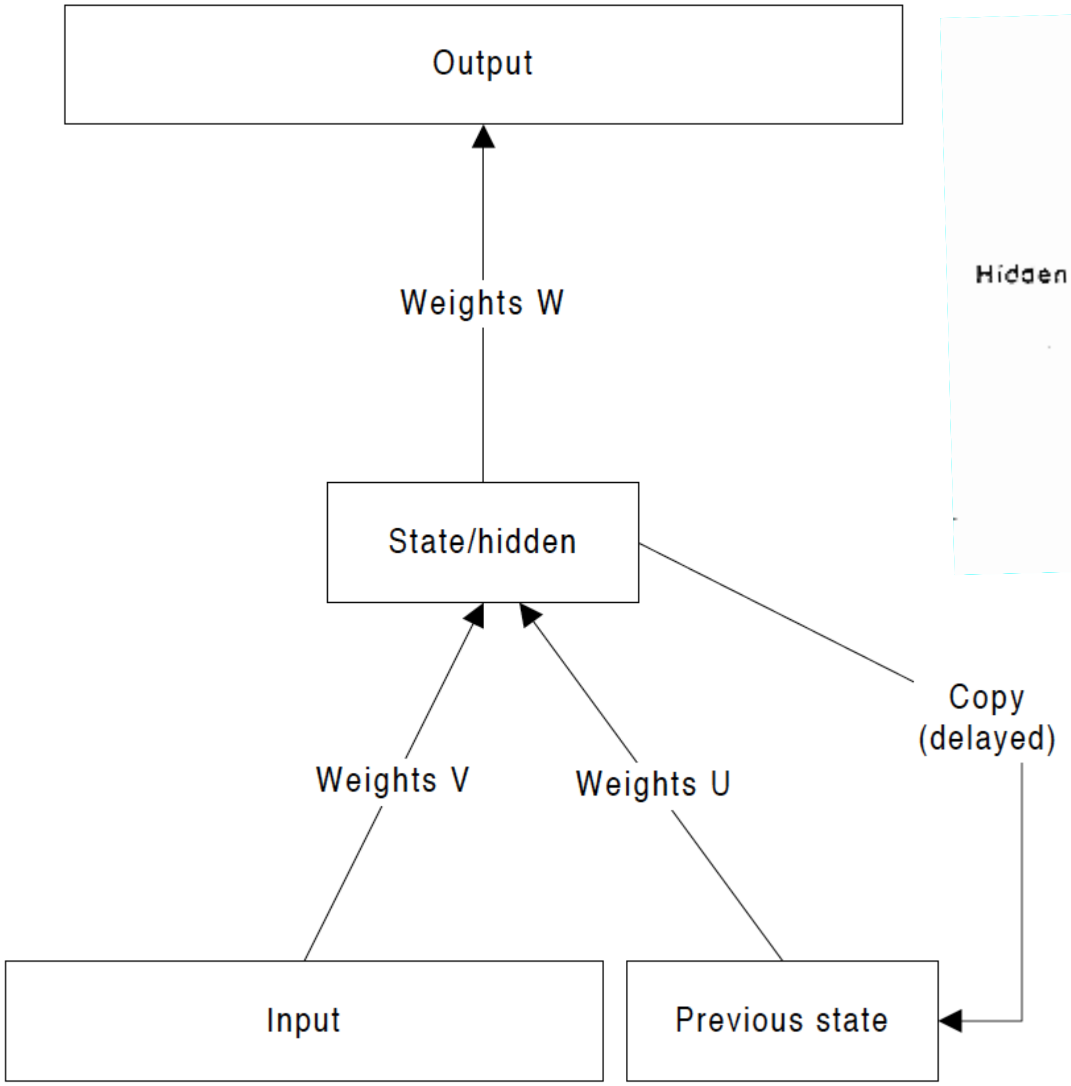
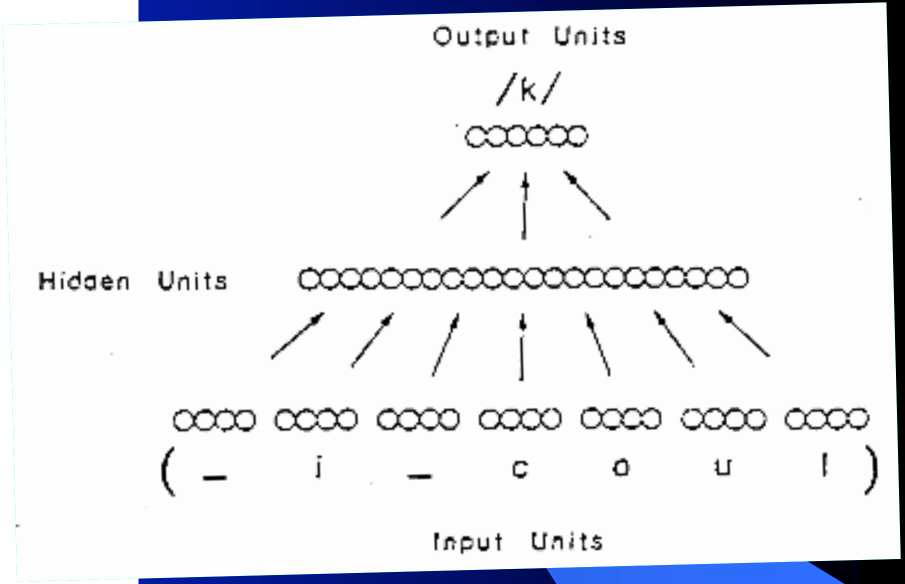
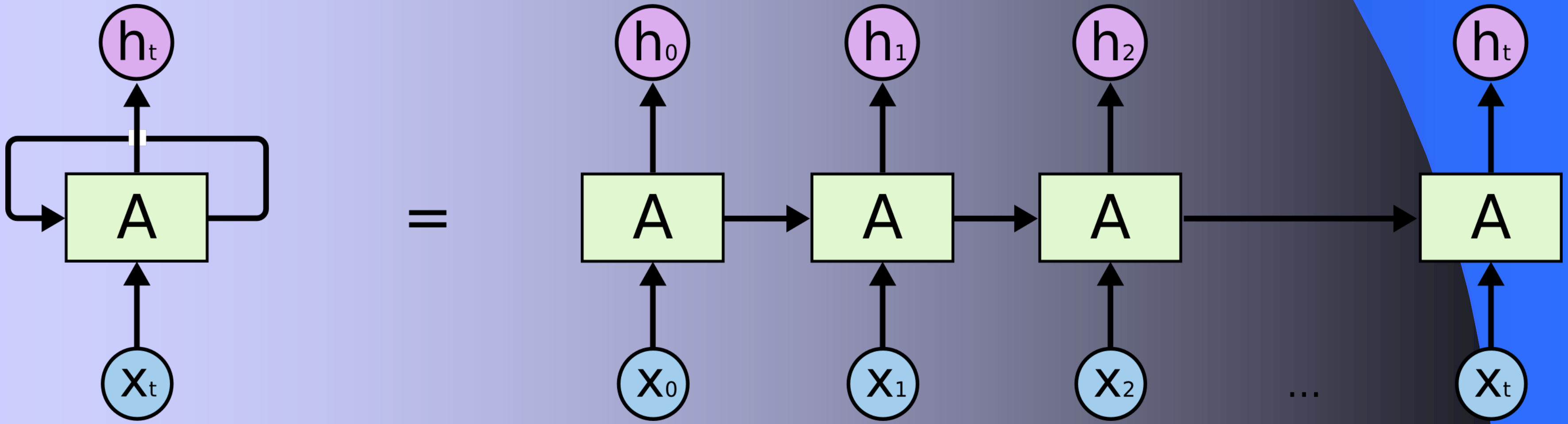


Figure 4: A simple recurrent network.



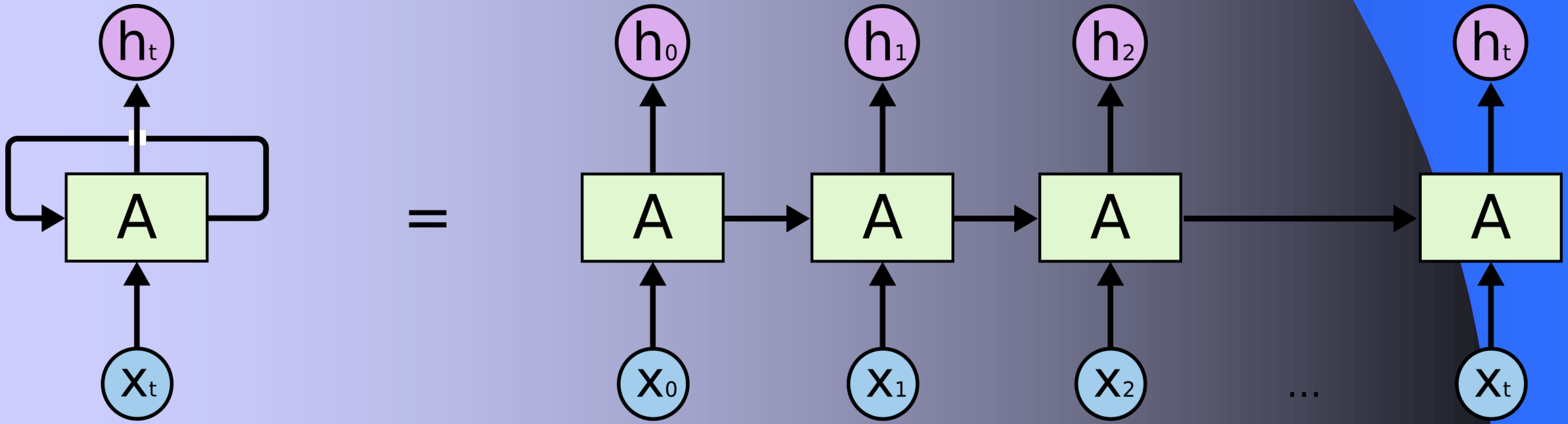
## Unfolding Recurrent Net in Time

- Can consider an equivalent feedforward network unfolded *t* steps in time
- Then can train it as if it was a regular feedforward network – Backpropagation through Time (BPTT)
- Only difference is that the weights are tied (use average)
- Becomes a deep net in time –Has vanishing gradient issues



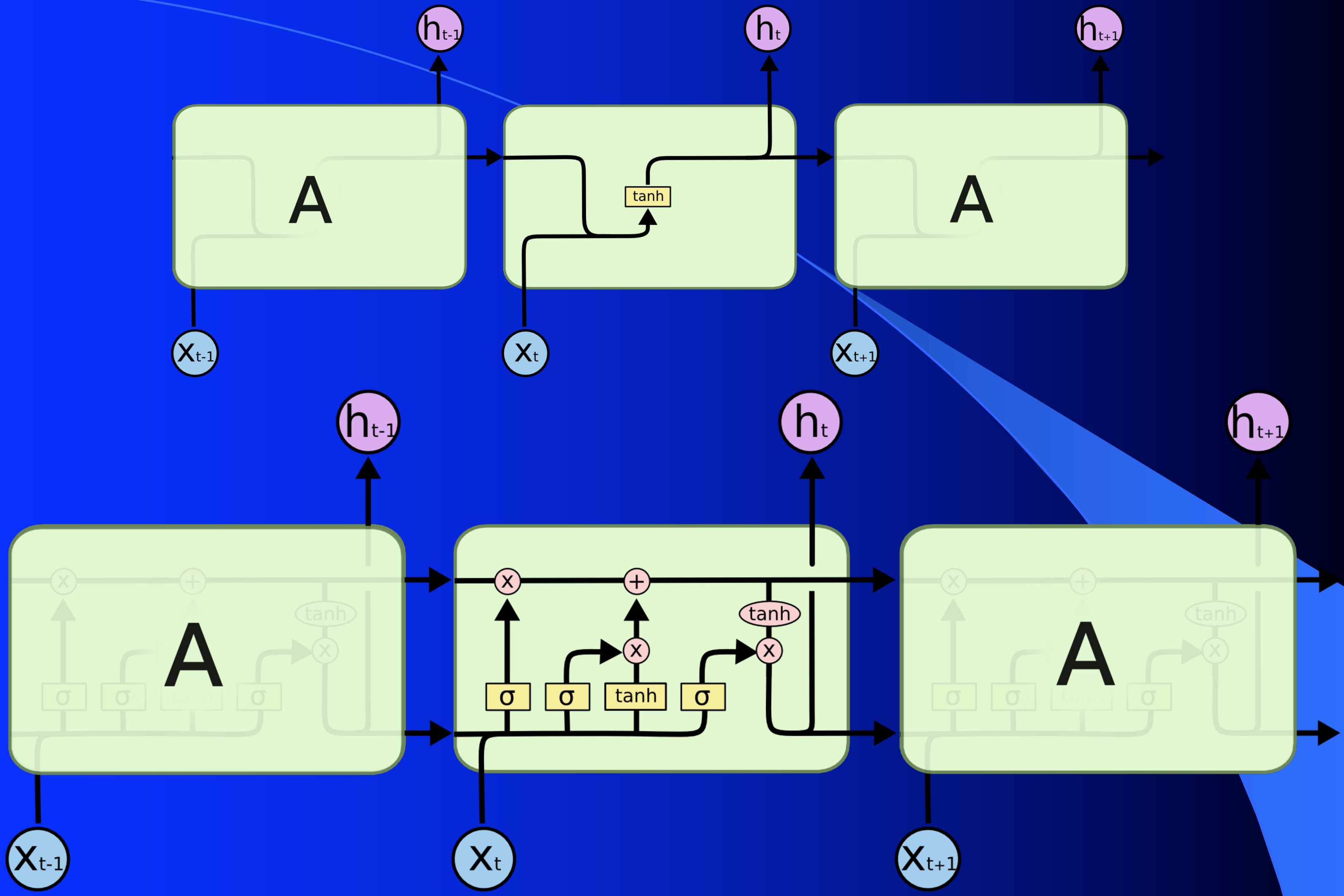
# LSTM/GRU

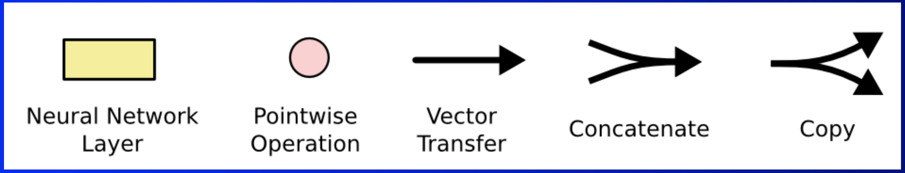
- Long Short-Term Memory/Gated Recurrent Unit
- *LSTM* – (*GRU* is an LSTM subset) – More powerful RNN
  - LSTM replaces the standard RNN nodes (A below) with a more complicated LSTM node with more learnable parameters
  - Train with BPTT but bigger *k*'s (a full sequence if not too large), or some pretty big chunk (25-100) since LSTM lets us avoid the vanishing gradient.



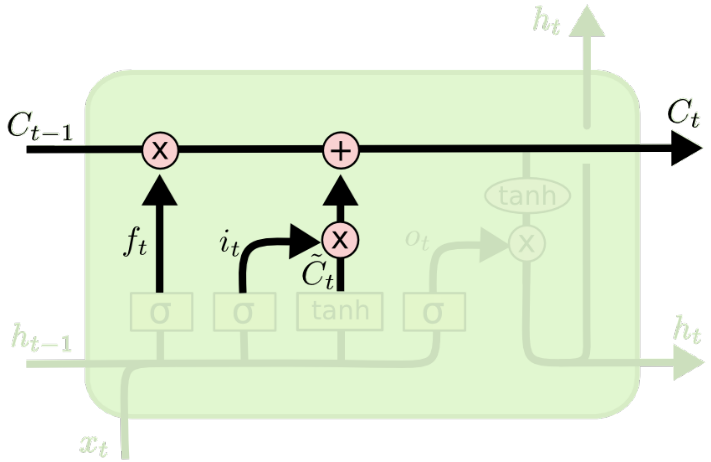
# LSTM – Long Short-Term Memory

- LSTM unit can just plug in for a standard node in an RNN
- Adds a state memory plus input, forget, and output gates
- Vanishing/exploding gradient avoidance
  - Cell value (state) has a self-feedback loop and can maintain its value indefinitely. Has derivate 1 with no vanishing gradient.
  - The cell value is multiplied by a forget gate output (0-1), which decides when and how much to forget, giving much more power.
- LSTM unit has lots more parameters but still trained with standard BP/SGD learning: typically BPTT
  - Forget gates, etc. don’t know that is their job, but the capacity is there during training to learn that job as the overall network minimizes loss
  - Capacity plus training finds a way to solve the problem, capacity that wasn’t there with simple network nodes

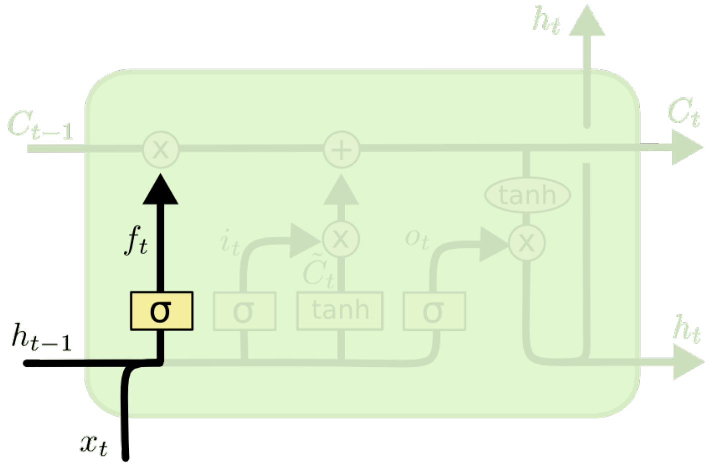




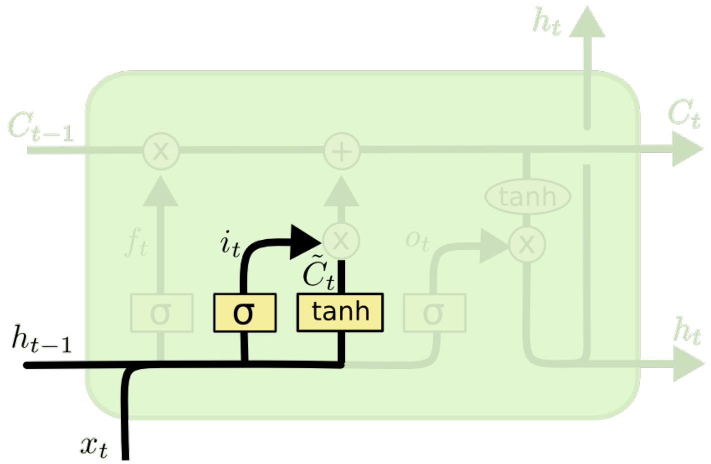
- $C_t$ is cell state, $h_t$ is context/output. Forget, Input, and Output gates



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



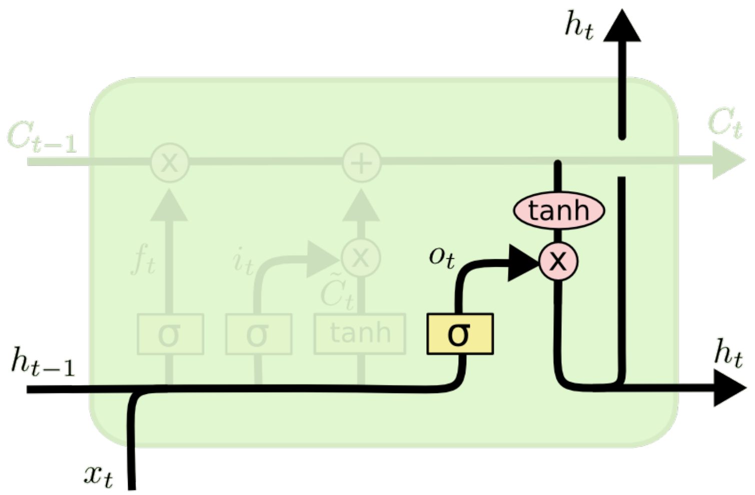
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

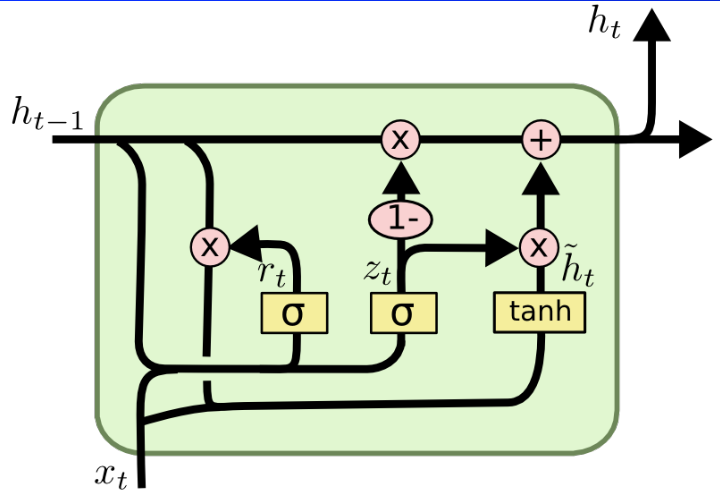
$C_t$ (state) and $h_t$ (output and/or context) are vectors.

$C_t$ can be maintained as long as needed. Only updated by forget and input gates, which are learned functions.



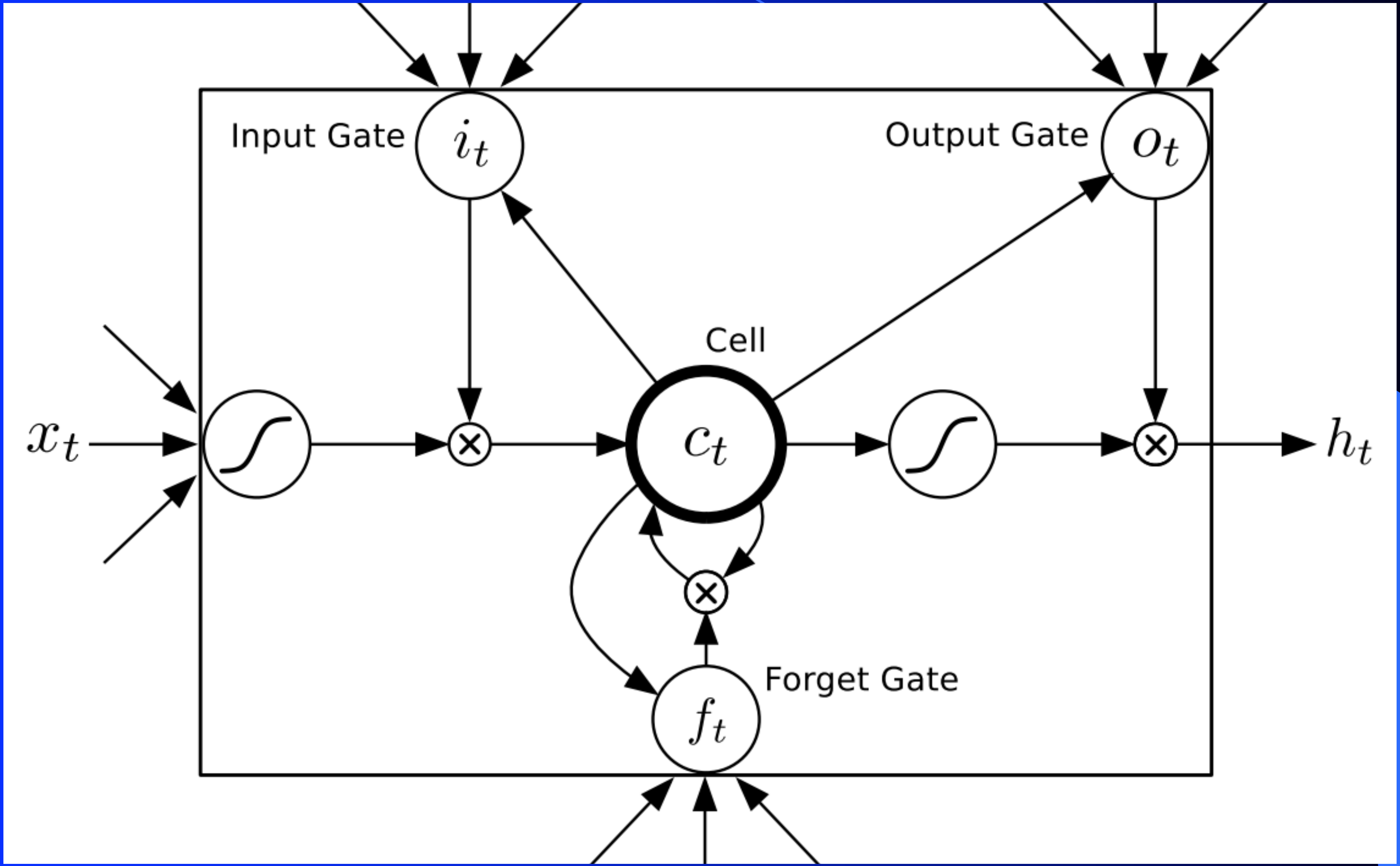
$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

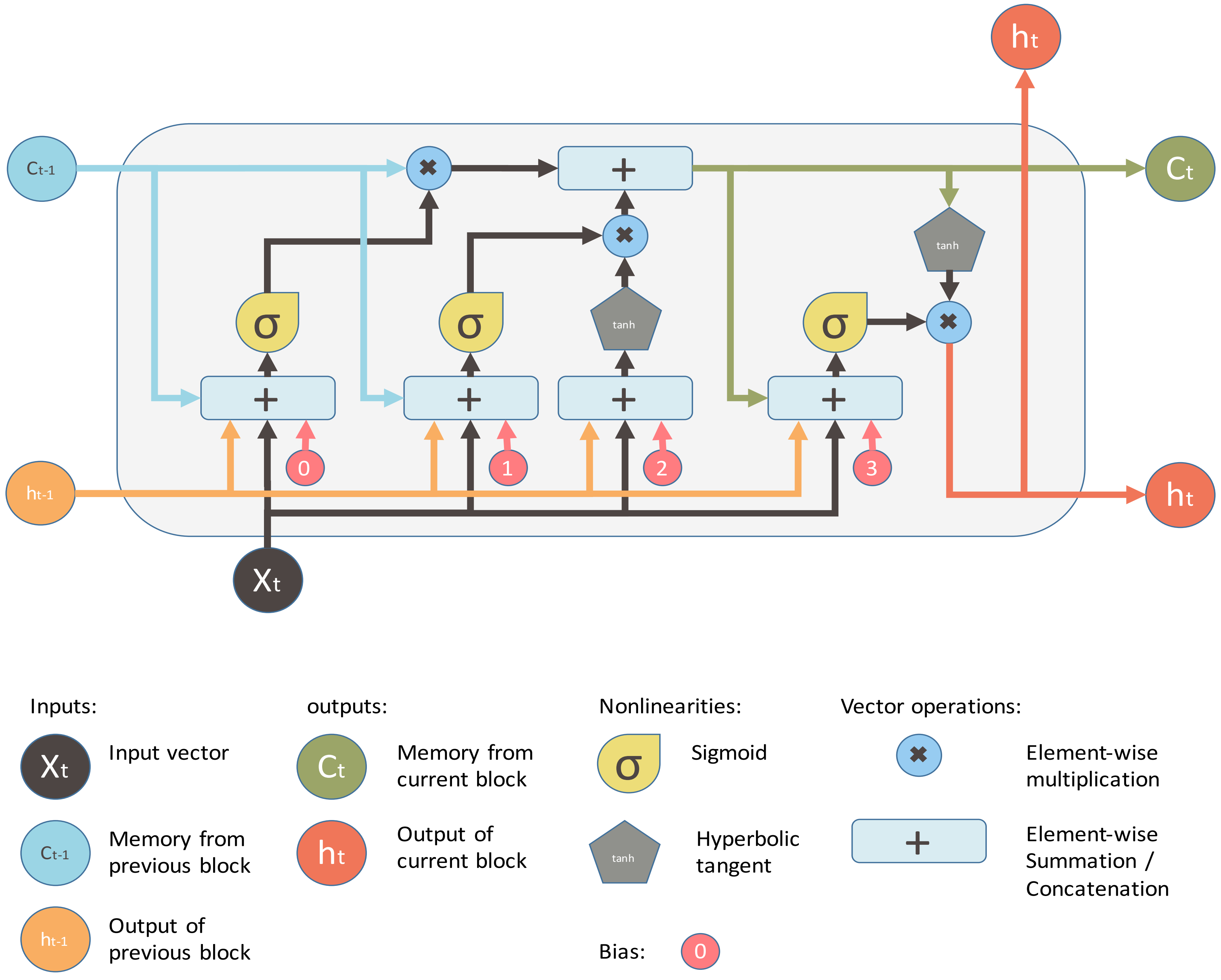
- Output gate above finishes full LSTM node
  - tanh normalizes *C* to *h*/*x* scale (between -1/1) before output gate chooses what parts of state to pass on as output/context
- Gated Recurrent Unit (GRU) below combines *C* and *h*
  - *r*: reset gate – How much of context $h_{t\text{-}1}$ to use in standard tanh
  - *z*: update gate - Combines forget and input gates



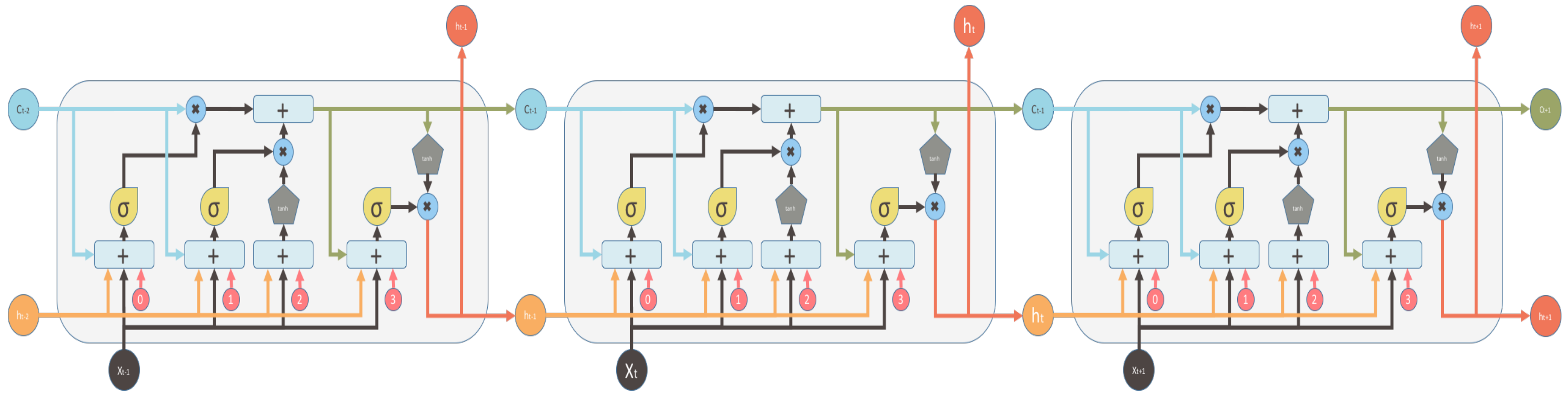
$$z_t = \sigma\left(W_z \cdot \left[h_{t-1}, x_t\right]\right)$$
$$r_t = \sigma\left(W_r \cdot \left[h_{t-1}, x_t\right]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot \left[r_t * h_{t-1}, x_t\right]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

## Basic Cell value has a self-feedback loop
## Does not forget until told (stable gradient)





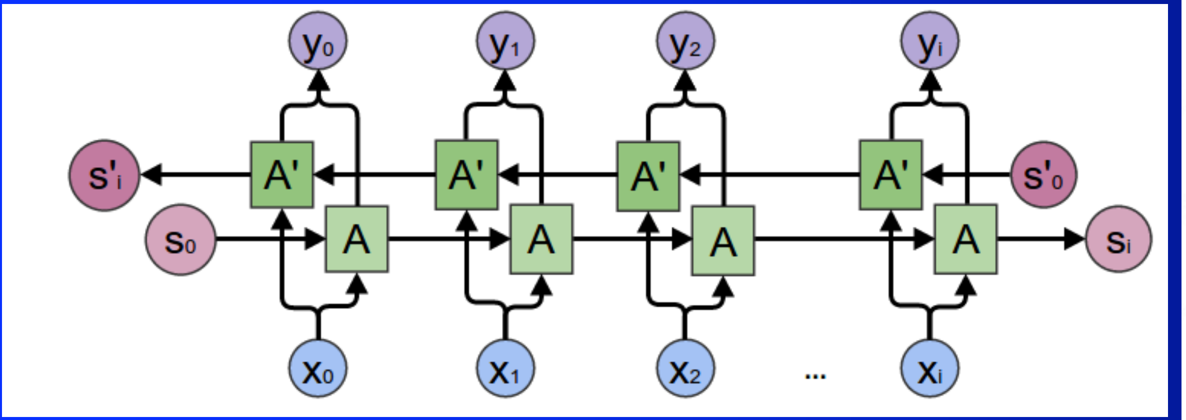
ht
Ct-1
Ct
tanh
σ
tanh
ht-1
0
1
2
3
ht
Xt
Inputs:
Xt
Input vector
Ct-1
Memory from previous block
ht-1
Output of previous block
outputs:
Ct
Memory from current block
ht
Output of current block
Nonlinearities:
σ
Sigmoid
tanh
Hyperbolic tangent
Bias:
0
Vector operations:
Element-wise multiplication
Element-wise Summation / Concatenation

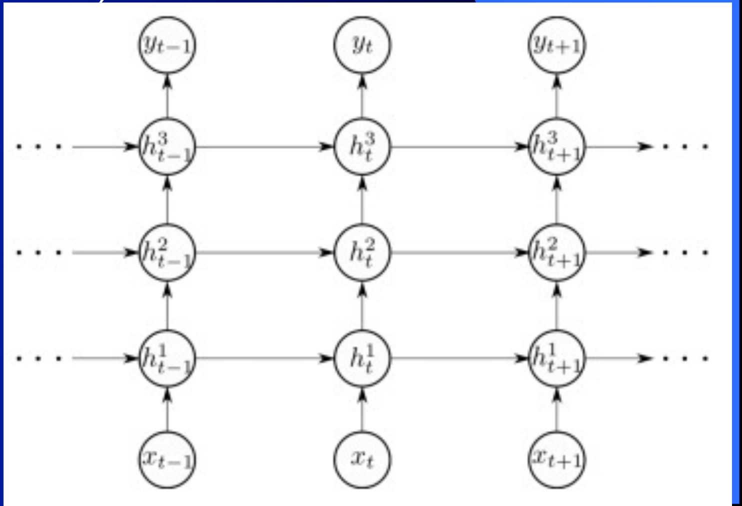


$h_{t-1}$
$h_t$
$h_{t+1}$
$C_{t-2}$
$C_{t-1}$
$C_{t+1}$
$h_{t-2}$
$h_{t-1}$
$h_t$
$h_{t+1}$
σ
tanh
0
1
2
3
$X_{t-1}$
$X_t$
$X_{t+1}$

# LSTM Variations

- Lots of node and structural variations
- Could replace any node in a deep network with LSTM node, but typically just used for sequential problems - RNN (time or space)
- Bidirectional LSTMS -Unlike conventional RNNs, bidirectional RNNs utilize both the previous and future context, by processing the data from two directions with two separate hidden layers. One layer processes the input sequence in the forward direction, while the other processes the input in the reverse direction. The output of the current time step is then learned using a combination of both layer's hidden vectors
- Stacked LSTMS - extra layers above (usually not many) can capture latent info (e.g. different time scales)



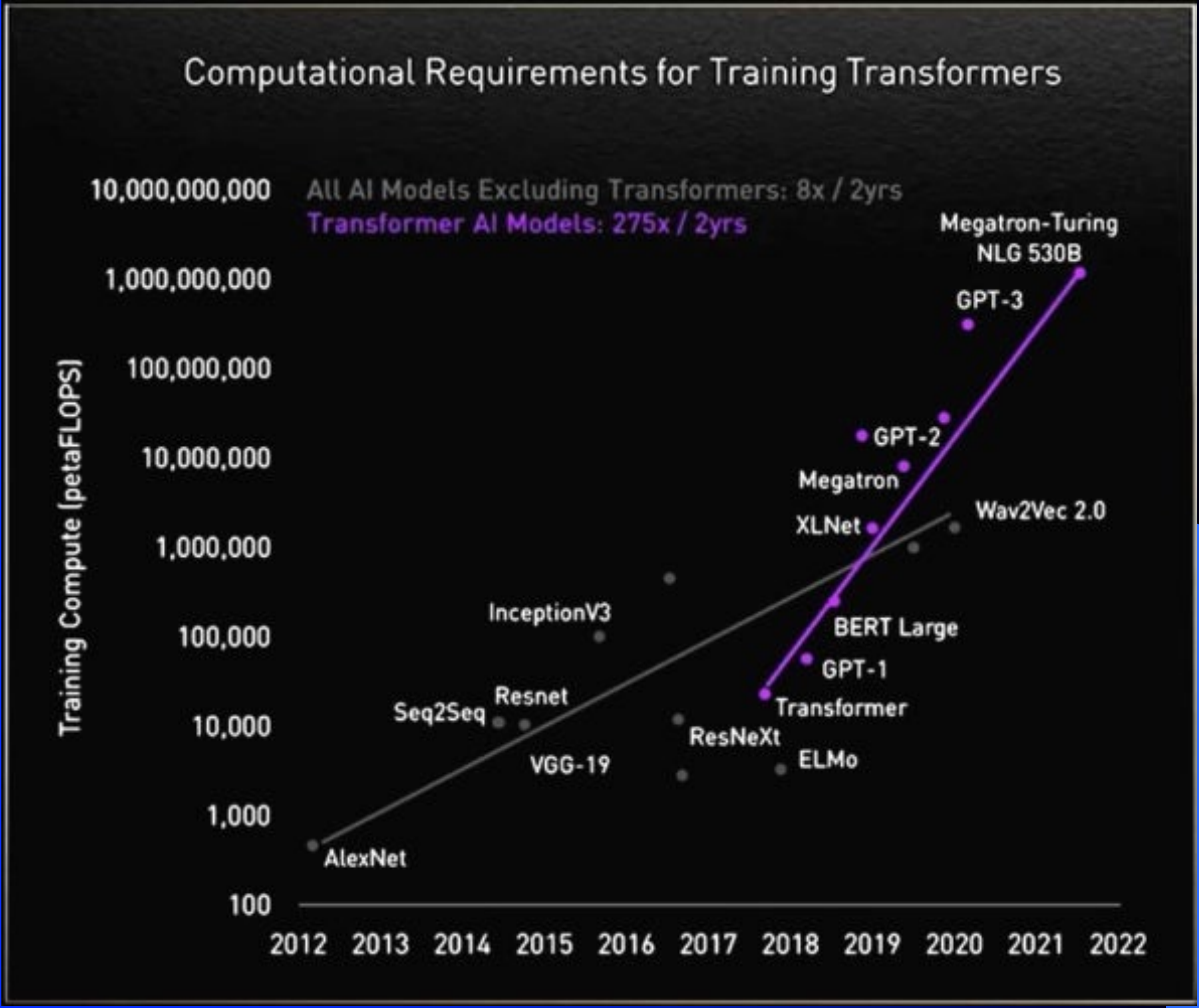


# Transformers - GPT

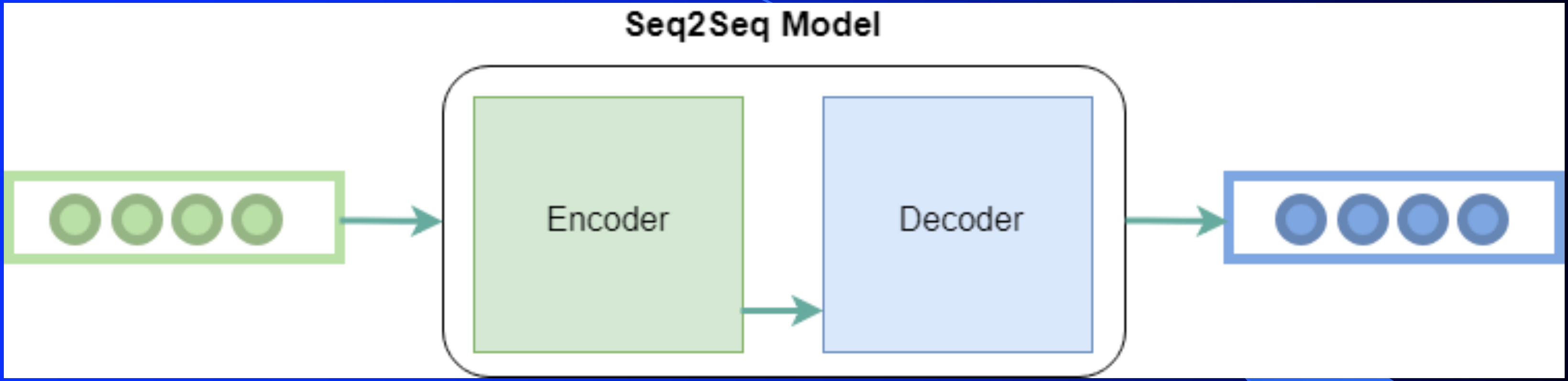
- Replacing CNNs and RNNs in many cases
- Rather than use convolutional filters or recurrence to find and track relationships they use *attention*
- We need short and long-distance relationships between features (e.g. words, pixels, atoms, etc.). What does "it" in a following sentence refer to?
- Initial paper – Google 2017 - "Attention is all you need"
- Immediately followed by lots of variations such as BERT (Bidirectional Representations from Transformers)
- GPT 1-3 are basically the same 2017 Google model
- More parameters/weights make a huge difference
  - GPT 2: 1.5B parameters, GPT 3 same basic architecture as GPT 2: 175B parameters

## Transformers Intro

- Long training (though with some improved efficiencies over CNNs), high power hardware, and lots of training data – big companies right now, Google, Nvidia, IBM, Microsoft, etc. coming out with their own versions
  - Now developing trillion parameter models
  - Latest work is increasing the ability for models to learn faster and better
- Large parameter sizes make them challenging to run locally
- Still not doing real reasoning, just finds patterns from lots of data and mimics those
  - Which is still pretty sweet
  - Don't overestimate current wisdom of content, but it will definitely "appear" like great human style output
- A future goal is to learn more like humans with less data needed


Computational Requirements for Training Transformers
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
Training Compute (petaFLOPS)
All AI Models Excluding Transformers: 8x / 2yrs
Transformer AI Models: 275x / 2yrs
Megatron-Turing NLG 530B
GPT-3
GPT-2
Megatron
XLNet
Wav2Vec 2.0
InceptionV3
BERT Large
GPT-1
Resnet
Seq2Seq
Transformer
ResNeXt
VGG-19
ELMo
AlexNet
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022

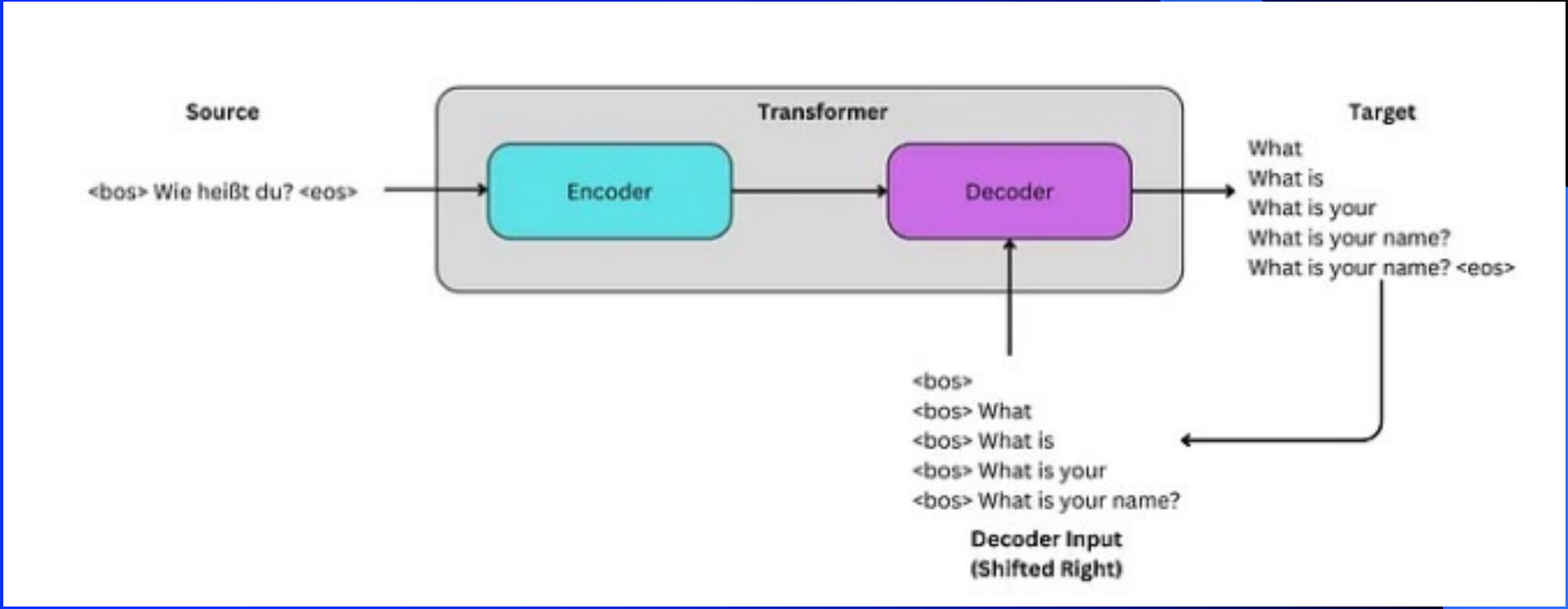
# Seq2Seq (Sequence to Sequence)



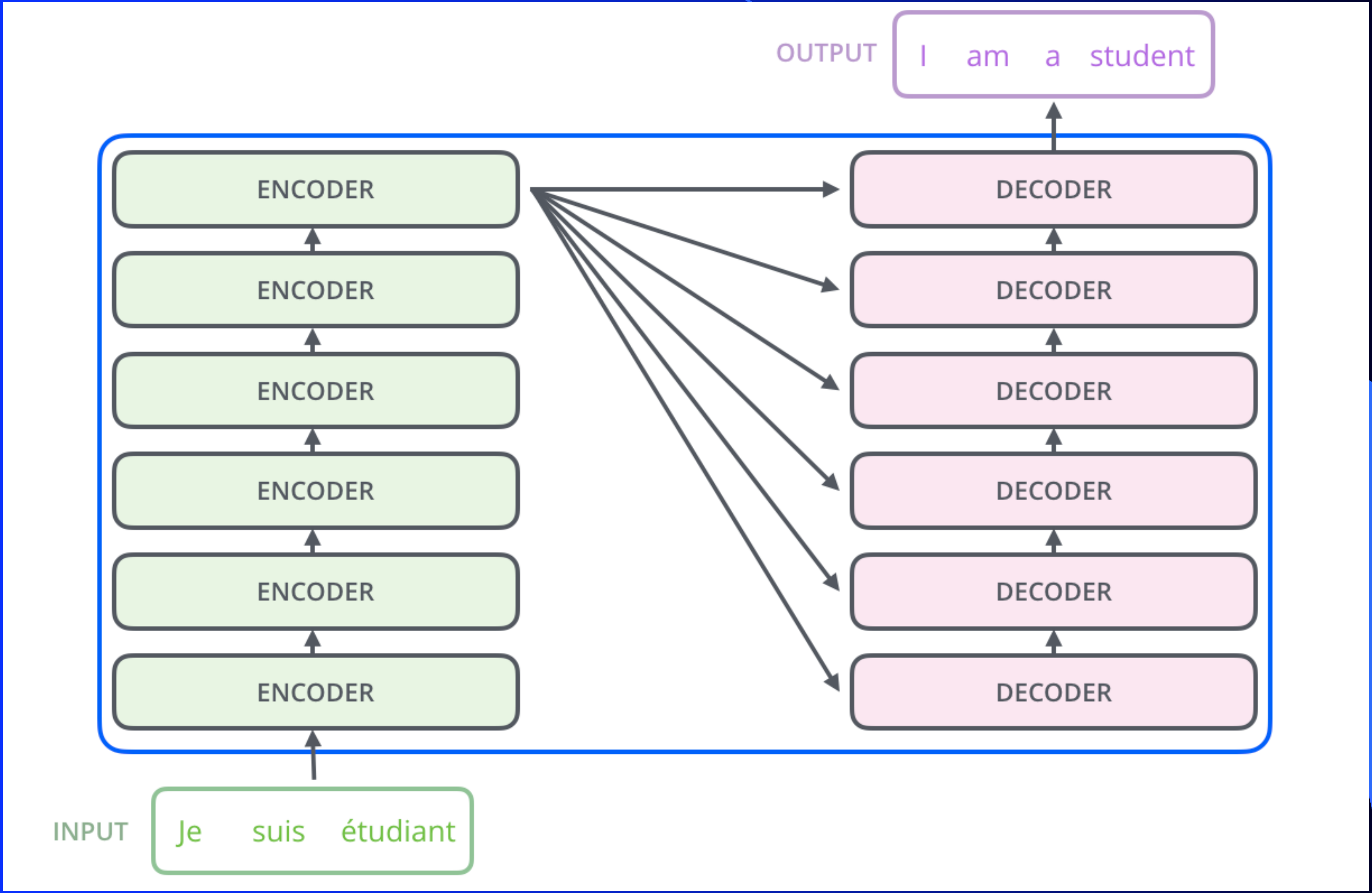
- Encoder and Decoder was typically done with RNN (LSTM/GRU) or CNN
- Encoded hidden layer vector representation (embeddings) of current tokens (e.g. words)
- Represents meaning and context of the current token – Final hidden layer (full meaning) passed to decoder
  - Translating English to French
- Decoder unravels embedding to create the output sequence

## Transformers: "Attention is all you need"

- Autoregressive – Just decide next word. Use the encoder embedding and the words inferred so far by the decoder as input to decide the next word
- For training, the next word is the current target



# Transformer Flow



# Transformer

- Full input sentence entered (Tokens)
- Learns embedding
- Adds position
- Skip connections and layer normalizations
- Weights how much certain words/tokens in the sequence are tied to others
- Allows more distant dependencies
- Self-Attention and encoder-decoder attention
- Masked on decoder so only considers past tokens
- Can also just use Encoder and Decoder models

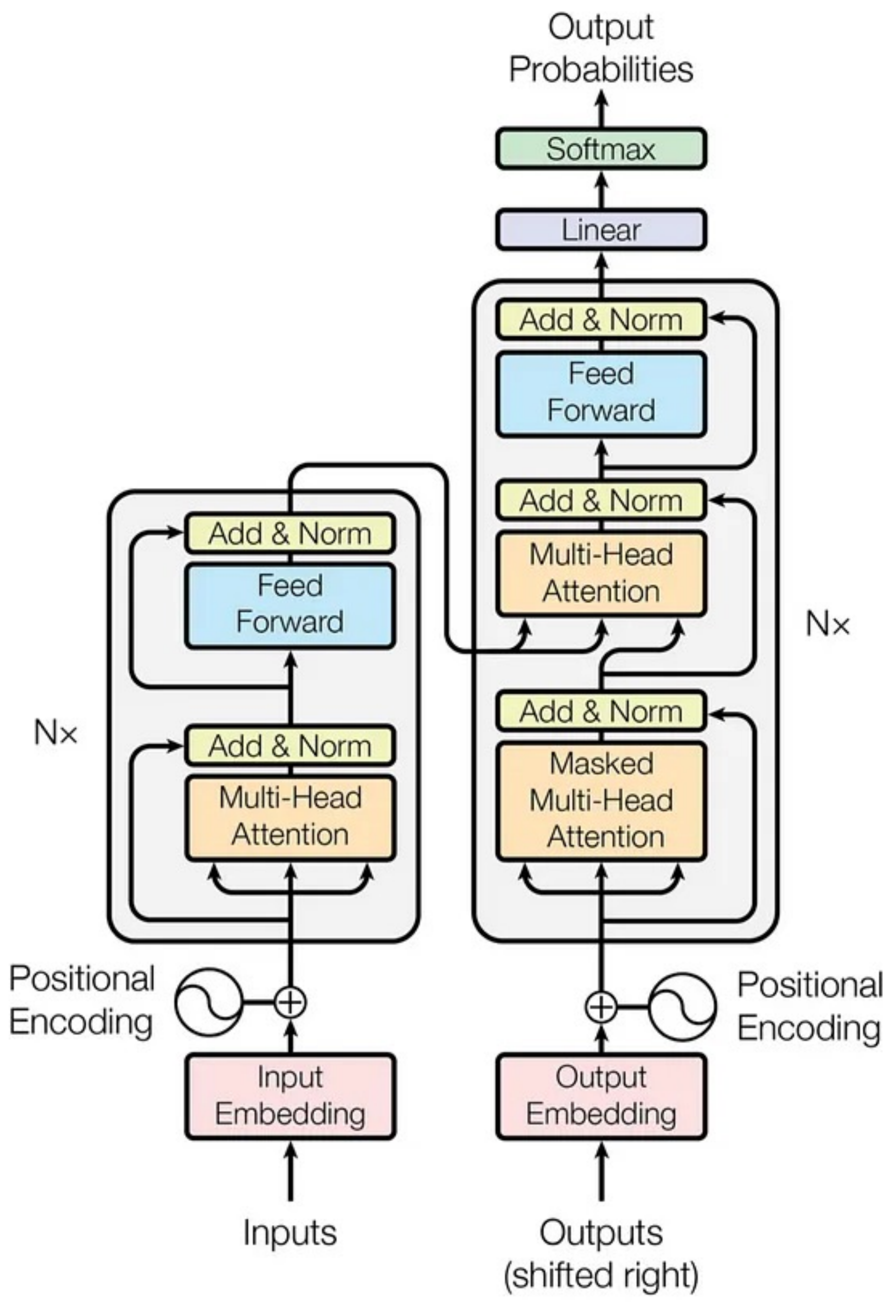
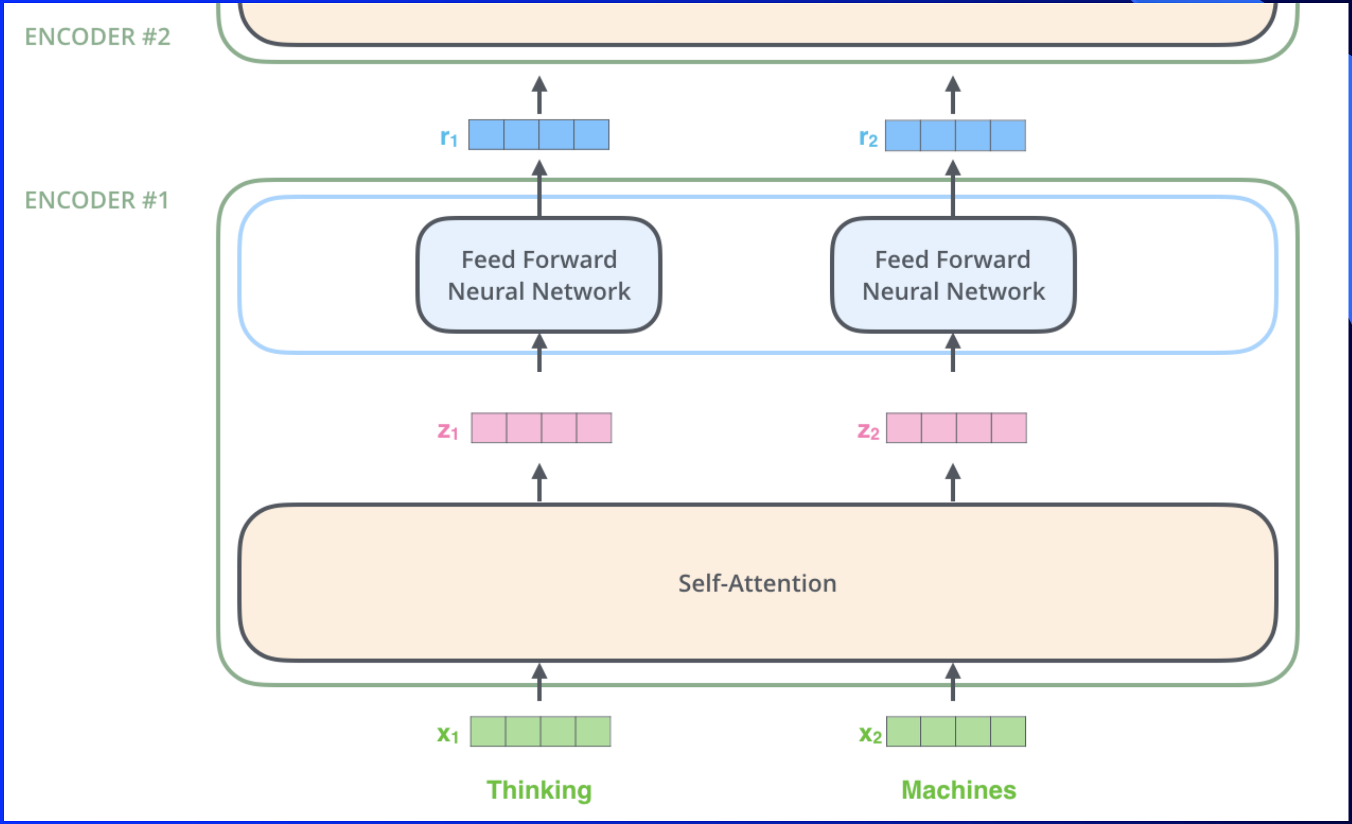


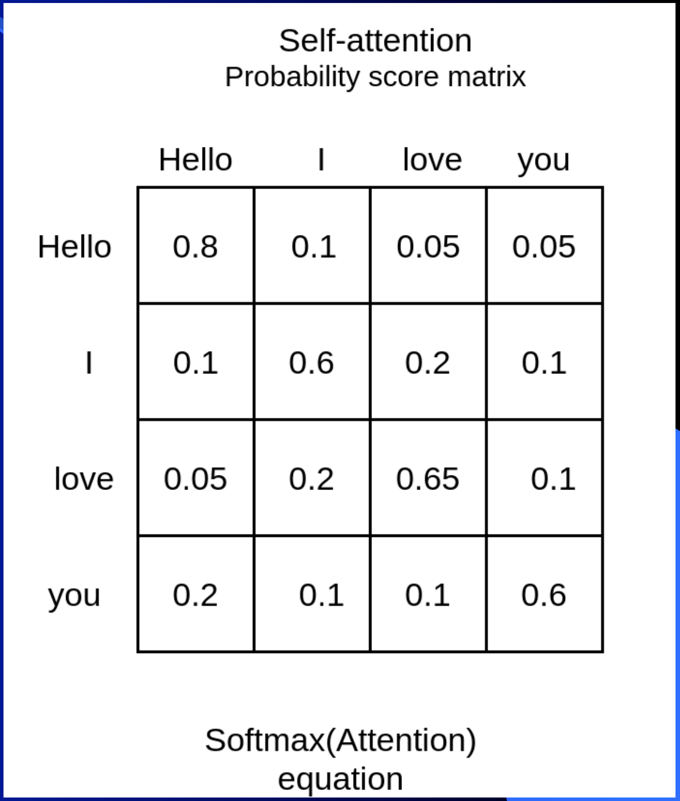
Figure 1: The Transformer - model architecture.

- 512 value token vectors in 2017 Google version (hyperparameter, 11288 in GPT)
- All token vectors flow independently and in parallel through encoder/decoder and only interact in the attention block
- Each token position has its own MLP with a single hidden layer with hidden nodes usually 4x the inputs - 2048
- 6 encoder and decoder layers for original version, (hyperparameter, 12 for GPT)
- Make wide enough for longest sentence in data set, fixed token window (2k+) for GPT



## Attention Mechanism

- For *n* tokens, the attention block basically builds *nxn* matrices measuring the relative connectedness of all the tokens
- Self attention (picture)
- Encoder-Decoder attention might have a sentence in French on the left and the response in English on the right
  - All cells relating to tokens to the right of the latest decoded word are masked to 0
- Builds matrix by doing a linear matrix multiply of the token vectors and learned arrays (Q, K, and V)
- Results are dynamic weights for relative importance between tokens



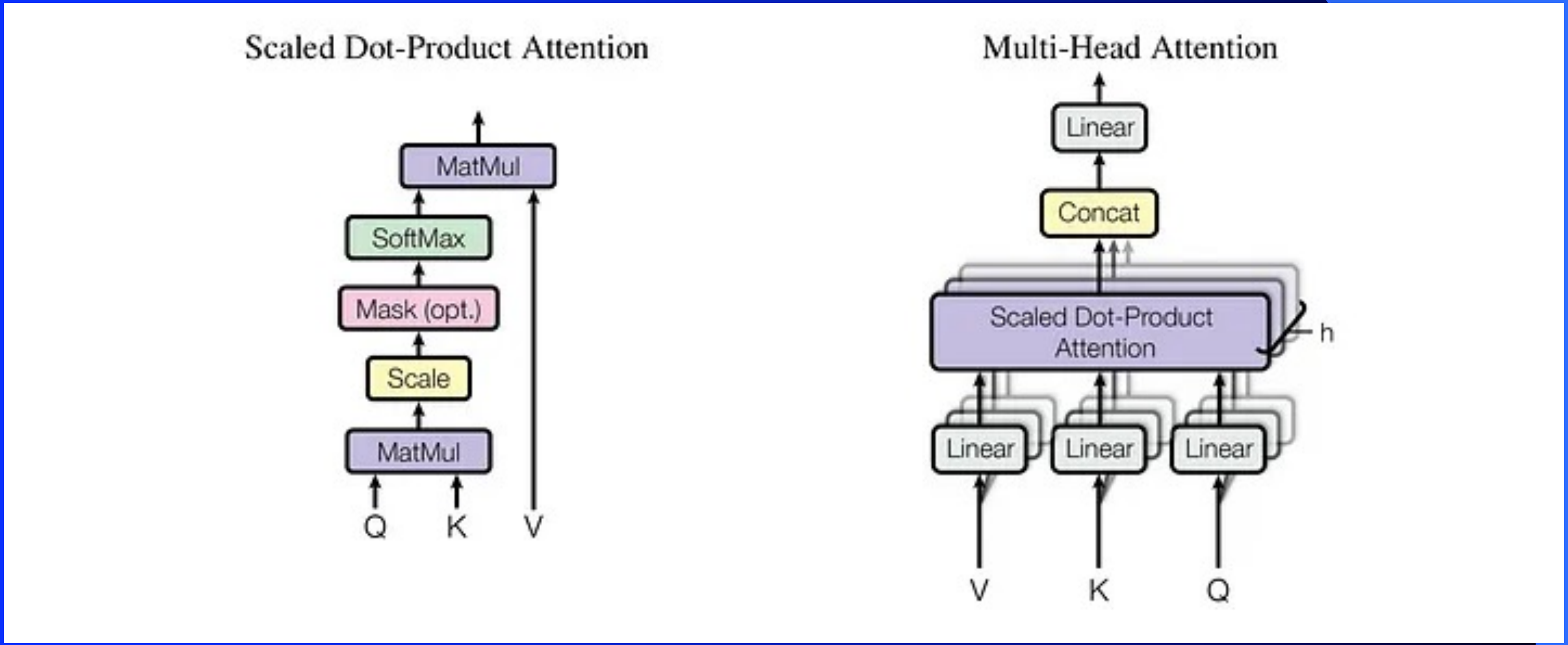
Self-attention
Probability score matrix

|  | Hello | I | love | you |
|---|---|---|---|---|
| Hello | 0.8 | 0.1 | 0.05 | 0.05 |
| I | 0.1 | 0.6 | 0.2 | 0.1 |
| love | 0.05 | 0.2 | 0.65 | 0.1 |
| you | 0.2 | 0.1 | 0.1 | 0.6 |

Softmax(Attention) equation

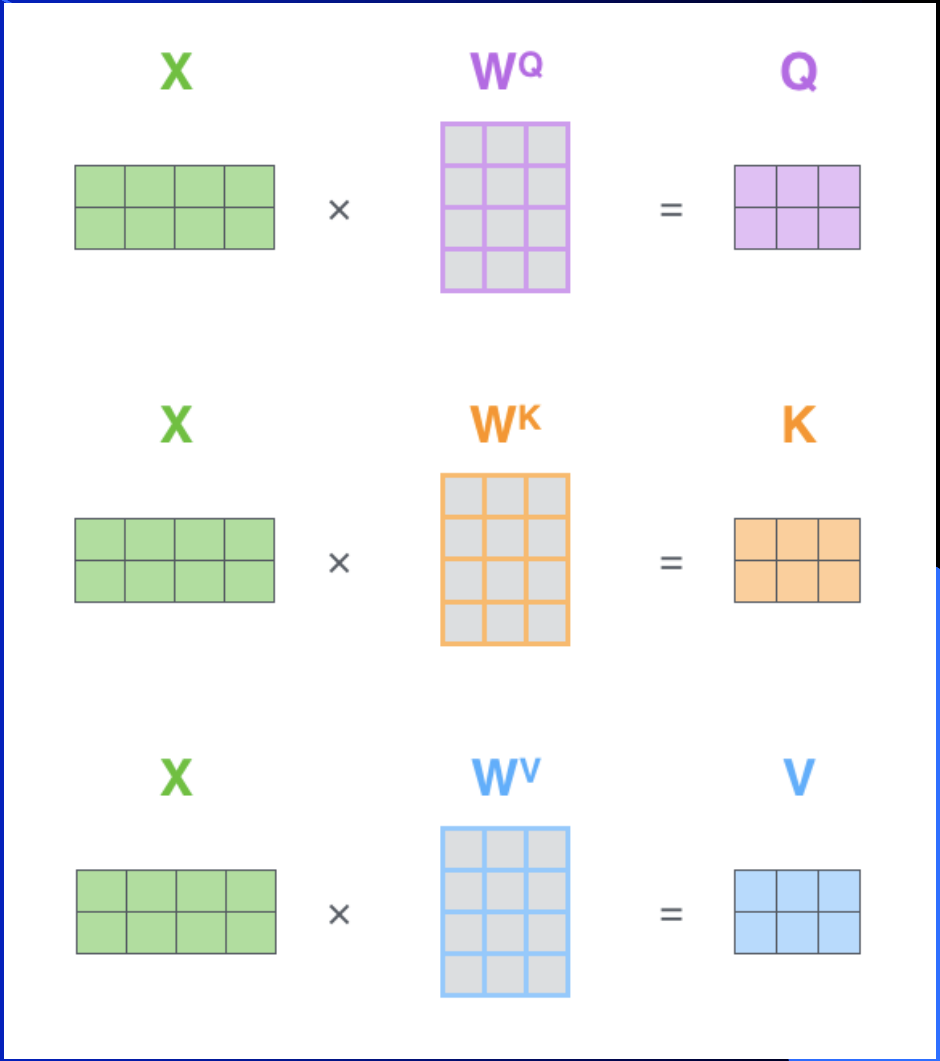
## Attention Mechanism

- The incoming token vectors are multiplied by learned arrays (the linear blocks in right part of image) to create Query, Key, and Value arrays
  - For self attention the V, K, and Q matrices are all the same
  - For encoder-decoder attention: Q is the matrix of decoder token vectors and, K=V=final token matrix output by the encoder
- In one scaled dot-product attention the "updated" Q, K, and V arrays are begin used

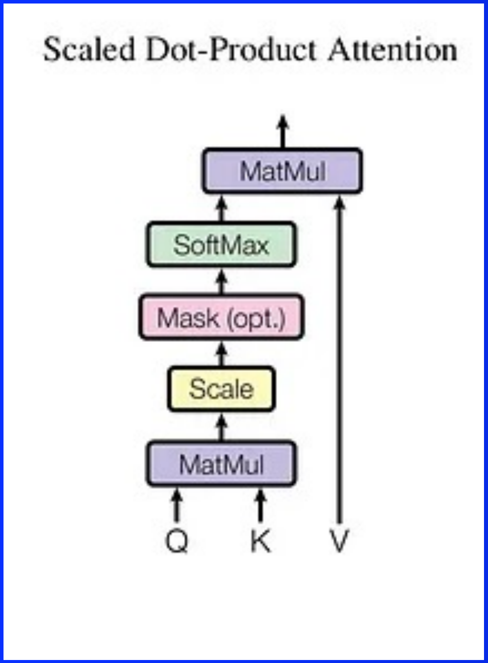


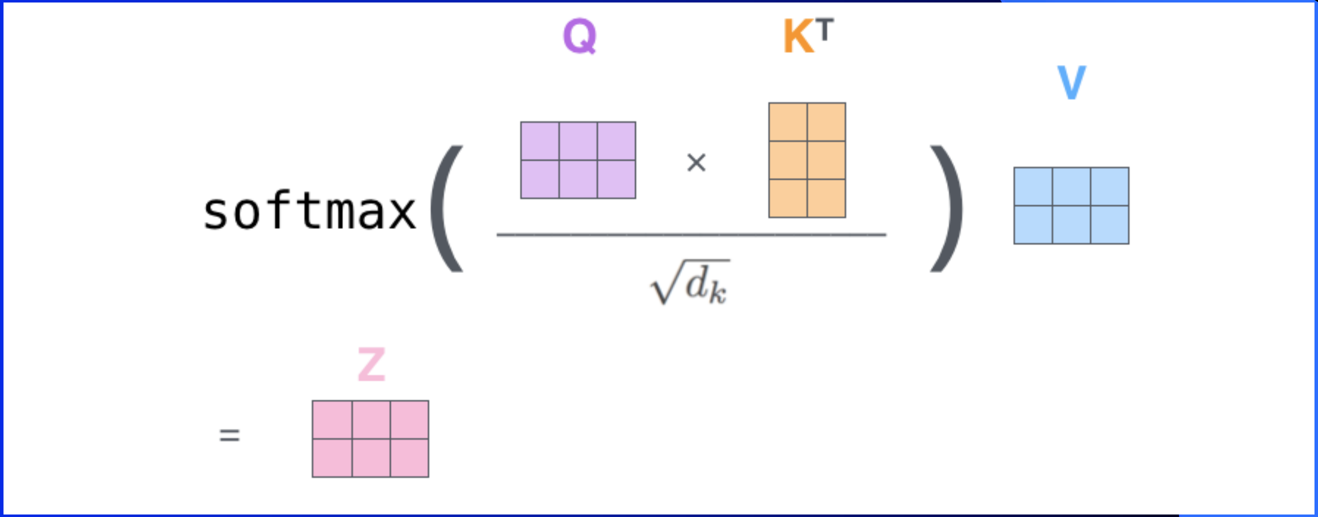
# Initial Attention Multiplies

- X is our array of token vectors (just 2 words in this example)
- For self attention multiply the same token vector array X by 3 learned matrices $W^Q$, $W^K$, and $W^V$ to get the updates arrays Q, K, and V
  - Rows must match columns of token vectors
  - We choose columns of W matrices (and thus length of attention vectors) as hyperparameter (3 here), 64 in 2017 Google version
- For encoder-decoder attention X is from encoder for Q, and from decoder for K and V

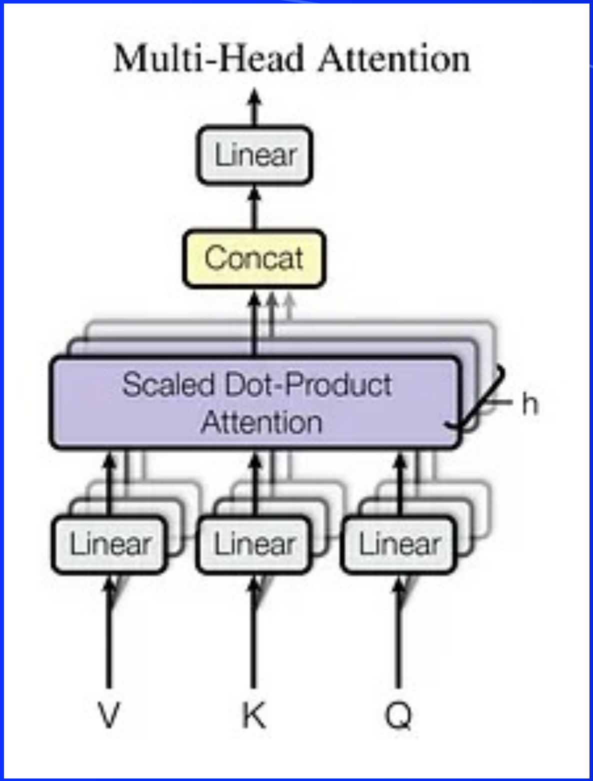


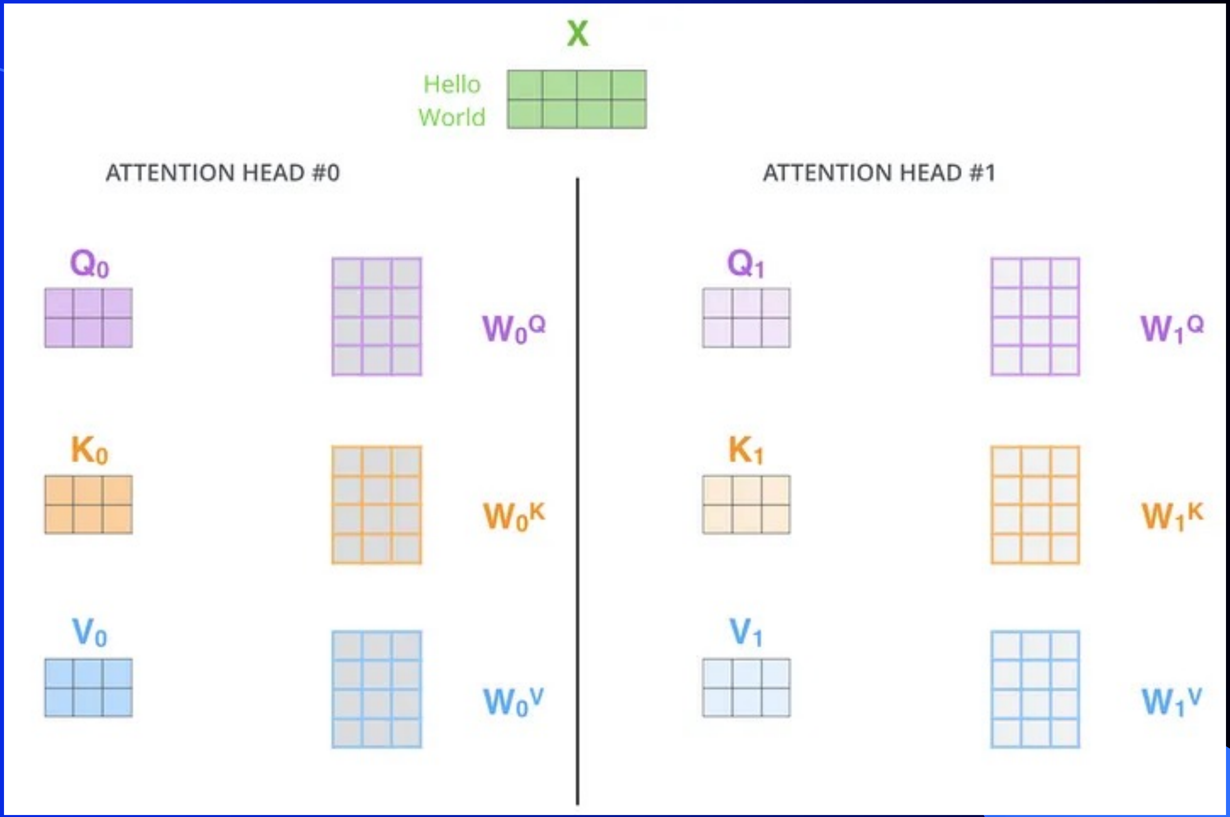
1. Q, V, and K created through matrix multiples
2. Q*K calculates attention with dot product array multiplication
3. Scale divides by the square root of the size of the attention vectors (8 in this case) and leads to more stable gradients
4. Softmax turns the attention array in probabilities
5. Multiply by V. The low probability scores from the softmax basically nullify the irrelevant words for each word token

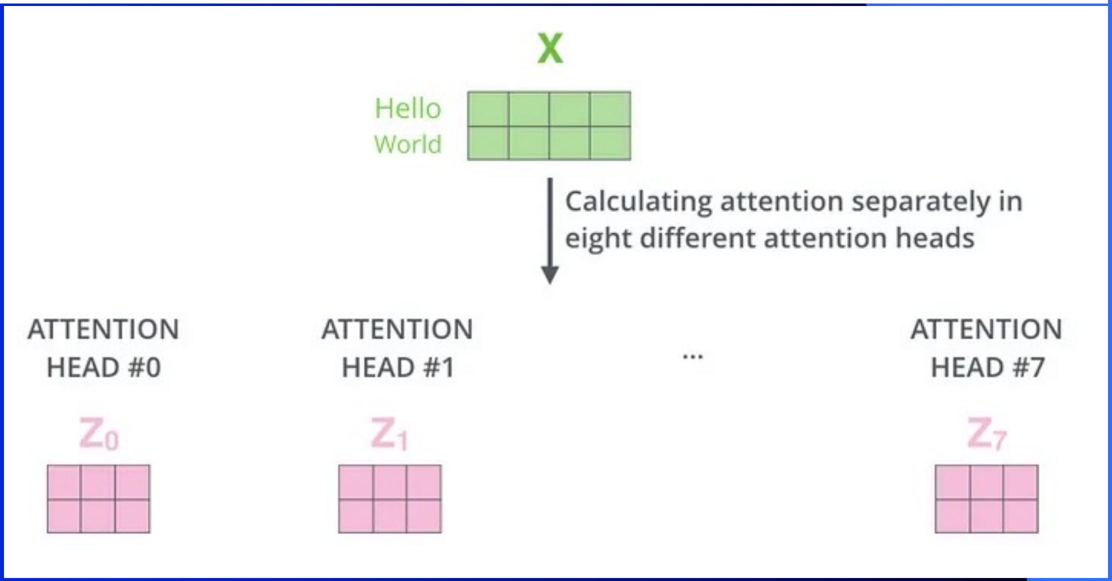




$$\mathrm{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$







Multiple heads allow learning of multiple representations.
8 heads in 2017 version
96 in GPT3

# Final Output

- Goes through all N layers of the decoder
- Final linear layer is a fully connected neural network which goes from the final vector to real value for every word in the vocabulary
- Softmax turns this into a probability of each word
- Add word with max probability to the output and then repeat for the next word of output

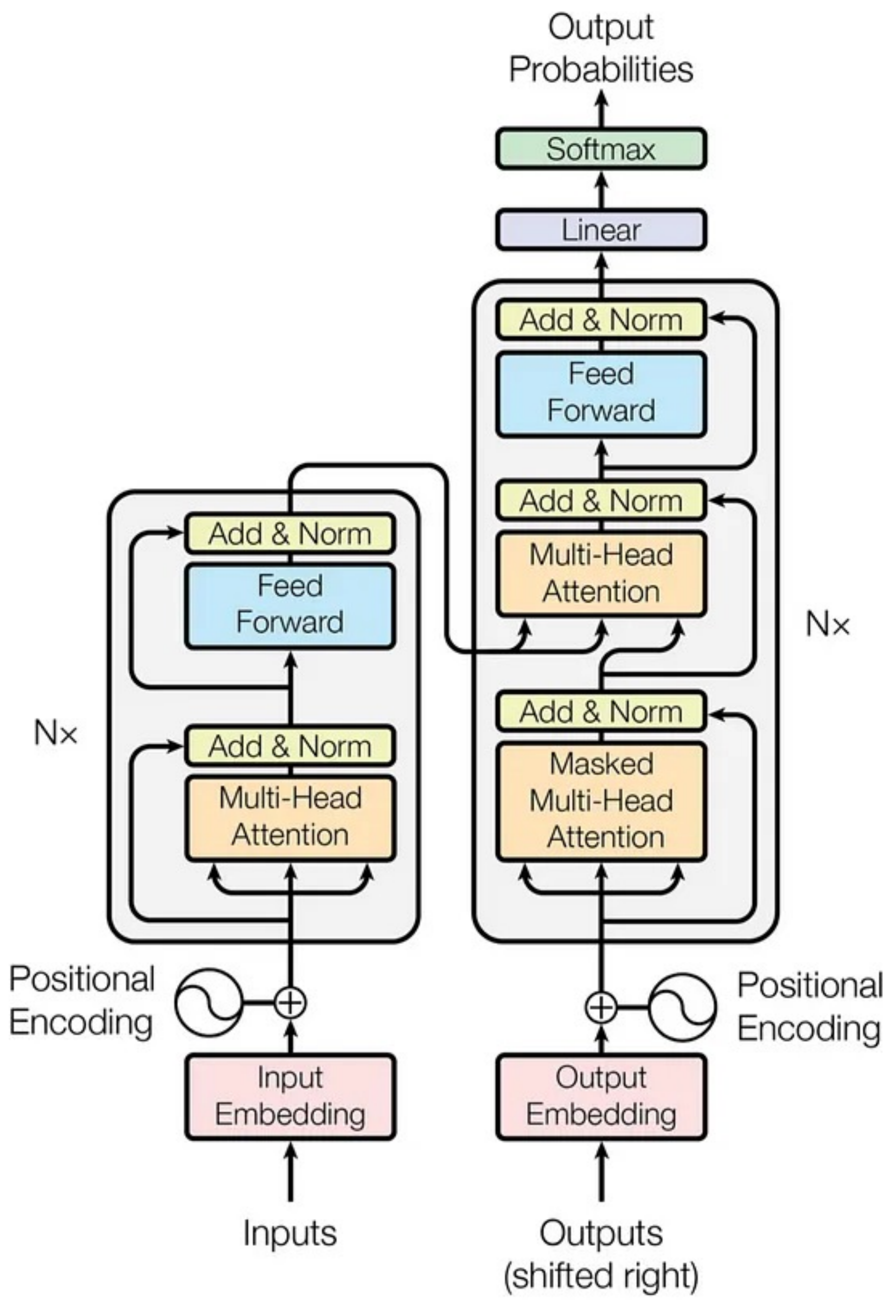


Figure 1: The Transformer - model architecture.

## Training

- Sentence to encode on left (English)
- Decode target on right (French)
- Shifted right so must predict output at time *i* using only inferred outputs at 1 … *i*-1.
- Target is the next word in the output sequence
- During inference start with full input sequence and empty output sequence.
- Add inferred word each time to output sequence which it can now use to help infer next word – Autoregressive
- Loss function – cross-entropy, difference between the one-hot rep of the target output word and the softmax vector

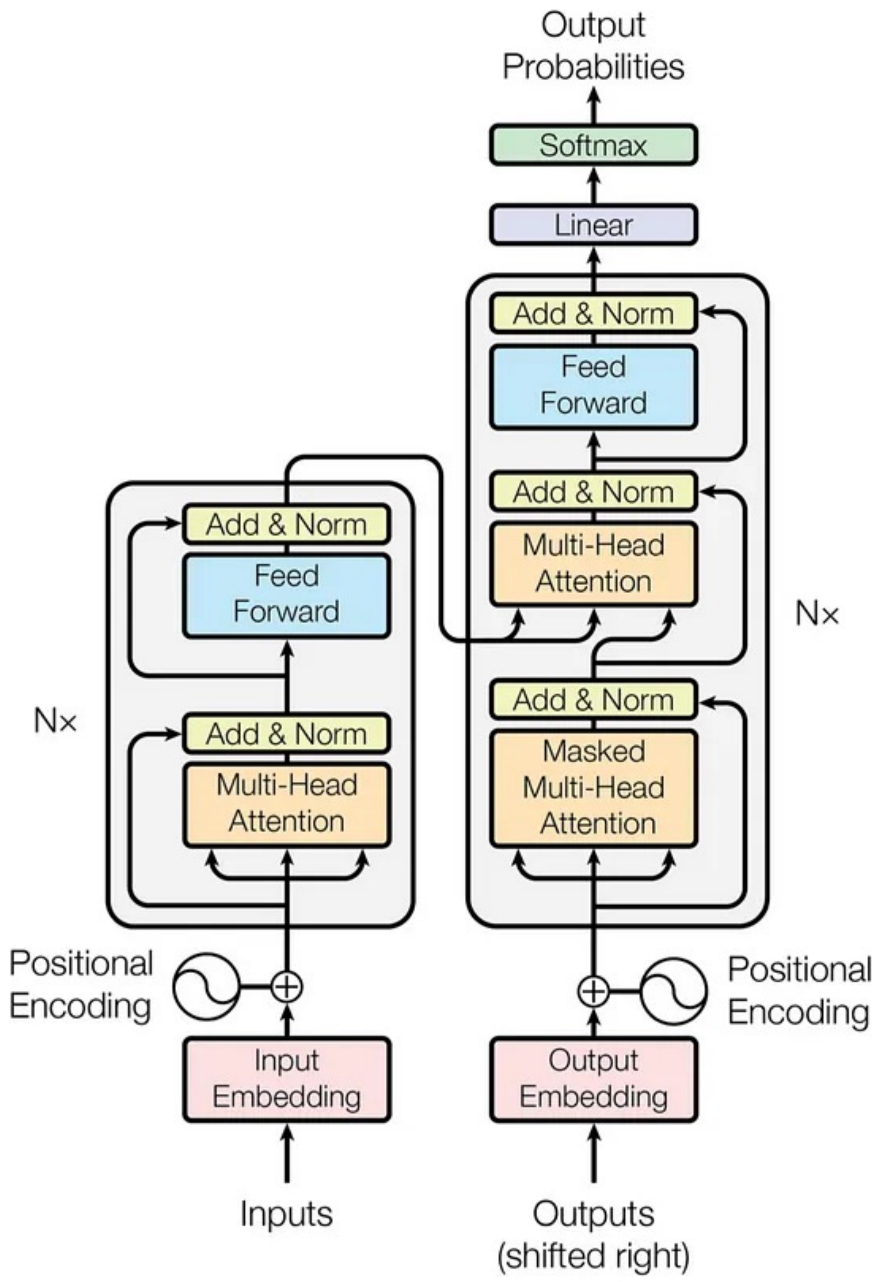


Figure 1: The Transformer - model architecture.

## GPT (Generative Pre-trained Transformer)

- OpenAI– Some of the original Google authors, Elon Musk and others fund, Microsoft funding and Partnership
- Decoder only transformer (just right side of transformer model with the encoder-decoder attention head dropped)
- Autoregressive: Initialize with an initial sequence (the query), then repeatedly output the next word
- Does not need labeled data! Finds associations while training with large amounts of unlabeled data. Just uses next word in text as target during training.
- GPT-2 1.5B, GPT-3 similar to GPT-2 but with 175 B parameters
- 800GB to store
- Pre-trained with a diverse corpus of unlabeled text datasets from web scraping (WebText)
- No fine tuning necessary (Though you can for specific tasks)

## GPT-3 Architecture

- Always uses a fixed 2k word/token sequence (some can be empty)
  - What is a Taco? (This would be the first 4 words of decoder. It will then Fill in the rest of the output one word at a time)
- Encoding – Each of 50,257 English words/tokens GPT-3 uses has an integer code – one hot encode into 50K length vectors
- Encoder transforms tokens (words or subwords) into a 12,288 real-value vector embedding (meaning, etc.) from the embedding weight matrix which is learned during training
- Thus, a 2k x 12288 matrix sequence
- For each token a positional encoding is calculated giving a 2k x 12288 positional encodings matrix
- These 2 matrices are summed and passed into the first layer

## GPT-3 - Continued

- 96x wide multi-head attention in each layer
- 12 layer decoder only transformer – Each layer includes layer norms, skip connections, multi-head attention, and MLP
- Decode – Just take the final 12288 vector after going through all the layers which represents the next word, and do the inverse encoding (NN) back into original word codes. Since real valued vector, use SoftMax to get final word probabilities and set the next word to the highest word probability
  - Or choose any of the high probability words if want variety
  - Actually outputs probabilities for the following word of every one of the 2K tokens, though next word is the one we concentrate on most
- Initialize with the query and start generating text
- Training: Small random weights (175B). Train with current target being next word in the sequence of unlabeled training data
- As token windows get wider, attention matrices grow by the square – GPT-3 uses Sparse Attention to be more efficient

# Sparse Attention

- As token window grows the attention matrices grow by the square! Expensive and slow
- Most inter-word connections are about 0 so they are sparse matrices and lots of savings possible
- GPT approach based on Big Bird (came after Bert)
- Has complexity O($n$) rather than O($n^2$) for the arrays
- One of few differences (other than number of parameters) between GPT-2 and GPT-3

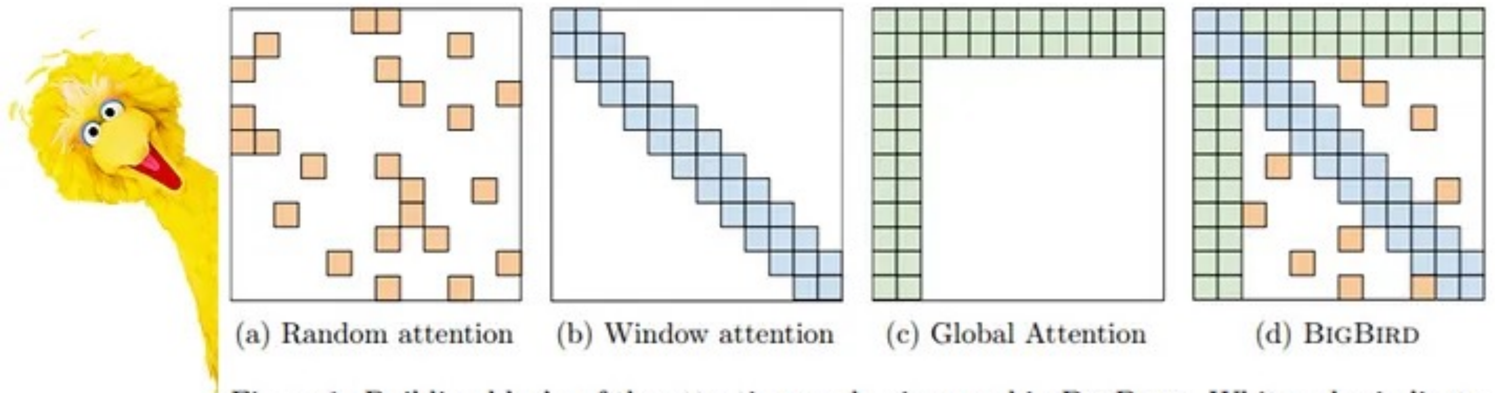


Figure 1: Building blocks of the attention mechanism used in BigBird. White color indicates absence of attention. (a) random attention with $r = 2$, (b) sliding window attention with $w = 3$ (c) global attention with $g = 2$. (d) the combined BigBird model.

## GPT/Transformer Advances

- GPT 3.5
  - Adds RLHF (Reinforcement Learning from Human Feedback) - Turbo
  - 4K token span (vs 2K for GPT 3)
- GPT 4
  - Multi-modal: Images, voice as part of queries and responses
  - ~1.7 Trillion parameters
  - 32K token span
- Other large companies building competitive models (Google Bard, Microsoft Bing, etc.)
- Pretty amazing, but still not doing deep understanding, rather an amazingly complex statistical *n*-gram model – In language, coding, etc., it models what humans have demonstrated in training data.  (i.e. Don’t expect it to give new wisdom yet)

## Transformers Conclusion

- Any applications with lots of examples with sequential features
  - Words in text
  - Atoms in protein sequences
  - Notes in songs
- NLP, Document generation and summarization
- Language Translation
- Computer Vision
- Audio and Speech Generation and Processing
- Computer Coding
- Biological sequence analysis
- Etc.

## Deep Learning Conclusion

- Much recent excitement, still much to be discovered
- Impressive results
- More work needed to understand how and why deep learning works so well – How deep should we go?
- Potential for significant improvements
- Works well in features spaces with local correlations in space and/or time – CNNs, RNNs, Transformers.
  - Important research question: To what extent can we use Deep Learning in arbitrary feature spaces?