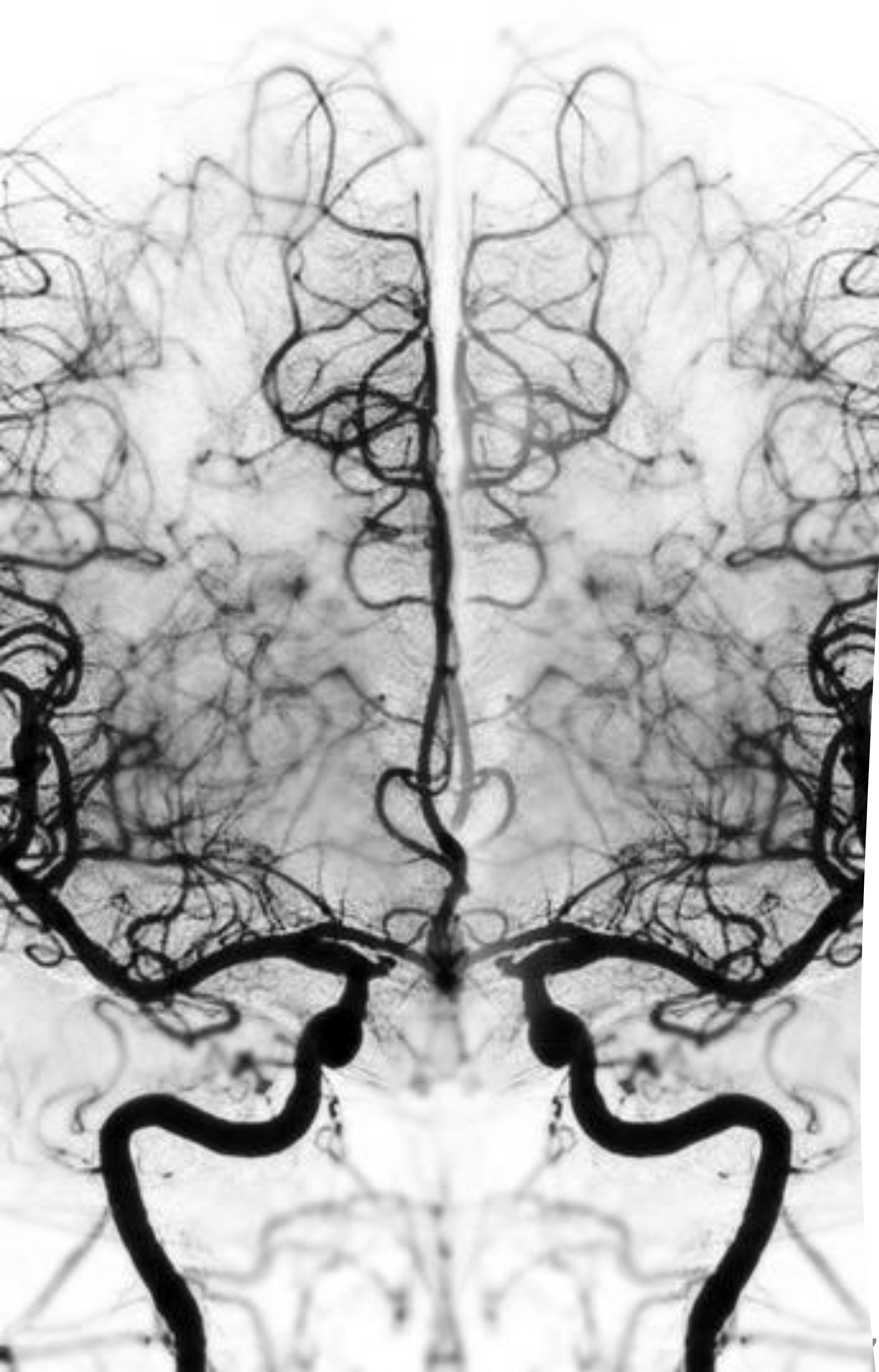
Artificial Neural Networks



Artificial Neural Networks

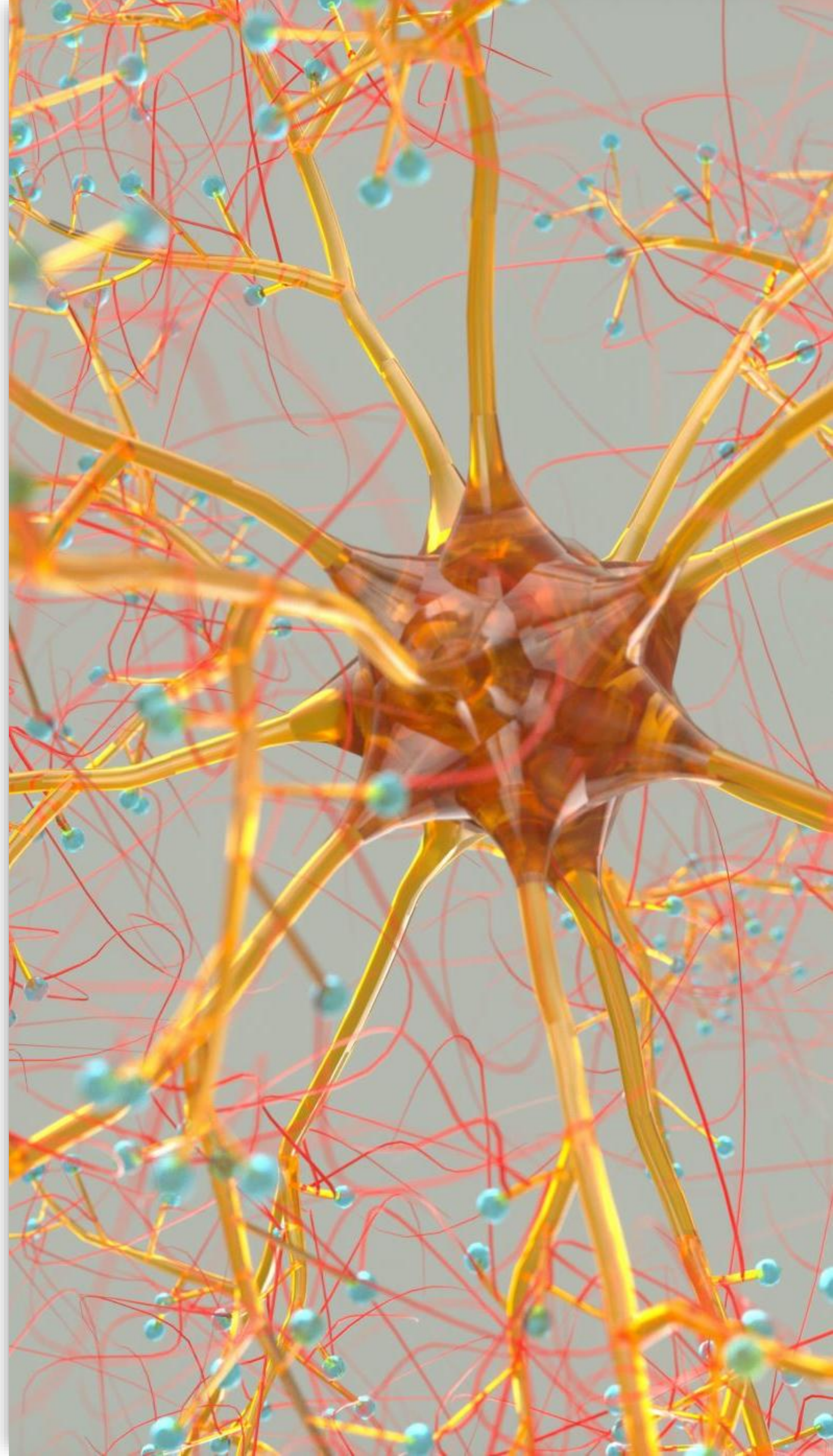
The concept of Artificial Neural Networks is to have an algorithm that simulates the behaviour of actual mammalian neurons. That makes the study of these algorithms of particular interest in the field of neuroscience. There are many different variations of neural networks, each with its own advantages and disadvantages.

Artificial neural networks have been used for aiding systems to learn. One classic example is image recognition. Artificial neural networks can be utilized to train an algorithm to recognise images. The network is composed of nodes often called artificial neurons, thus mimicking the structure of a biological brain. Each connection can transmit a signal from one node to another node, typicalap this signal is a real number. The output of each node/artificial neuron is computer using a non-linear function of the sum of the inputs.

Artificial Neural Networks

One of the most fundamental concepts in neural networks is Hebb's rule. Hebb's rule states that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. Therefore, if two neurons regularly fire simultaneously, then the connection between them will be strengthened. Hebb's rule also indicates that the opposite is also true: if two neurons rarely fire simultaneously, the connection between them will weaken. This has been put more colloquially as "neurons that fire together wire together".

Artificial neural networks consist of nodes that are connected to each other to form a network. The connections between the nodes can be directed and have weights. The nodes are organized into layers, the various layers work to process data. Those connections may flow in only one direction, or in both directions. The level of connectivity within an ANN also varies with different types of ANN.



McCulloch-Pitts Neuron

Established in 1943 by McCulloch and Pitts, by mimicking the functionality of a biological neuron

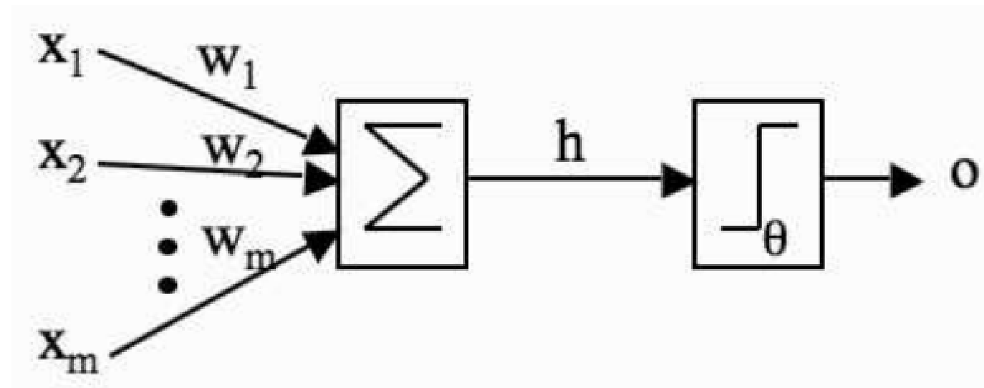


FIGURE 3.1 A picture of McCulloch and Pitts' mathematical model of a neuron. The inputs x_i are multiplied by the weights w_i, and the neurons sum their values. If this sum is greater than the threshold θ then the neuron fires; otherwise it does not.

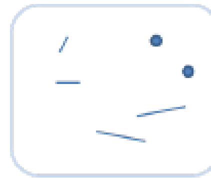
(1) **a set of weighted inputs** w_i that correspond to the synapses

(2) **an adder** that sums the input signals (equivalent to the membrane of the cell that collects electrical charge)

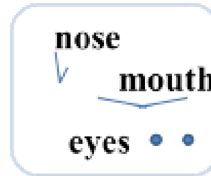
(3) **an activation function** (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs

Neural Network Function

https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1

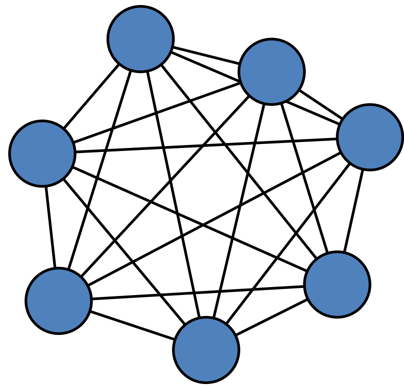


Layer 1: detect edges & corners

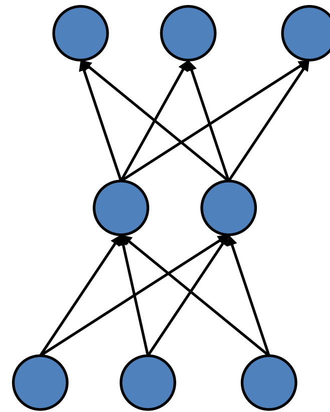


Layer 2: form feature groups

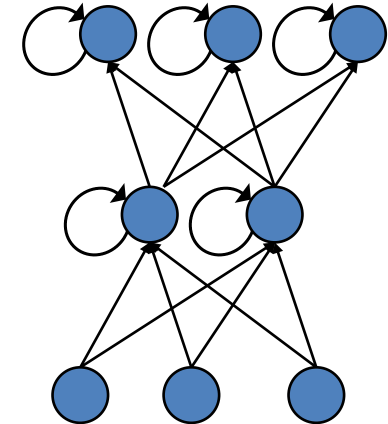
Layer 3: detect high level objects, faces, etc.



completely connected



feedforward (directed, a-cyclic)



recurrent (feedback connections)

Topologies of Neural Networks

ANN Glossary

Accuracy is a measure the performance of classification, classification with missing inputs, and transcription, accuracy is the proportion of examples for which the model produces the correct output. Accuracy is affected by weights and biases.
Cross Entropy is a metric for estimating how well a model would generalize to new data by testing the model against one or more non-overlapping data subsets withheld from the training set.
Cross Validation is repeated use of the same data, but split differently **(i.e., different training and testing sets).**
Dimensionality Reduction Dimension reduction summarizes a dataset using its common occurring patterns. Dimension reduction finds patterns in data, and uses these patterns to re-express it in a compressed form. This makes subsequent computation with the data much more efficient.
Error E, a function that computes the inaccuracies of the network as a function of the outputs y and targets t.
Error Rate is a performance metric of the classification of classification, classification with missing inputs, and transcription, the proportion of examples for which the model produces an incorrect output.
Feature measurable property of an object or event with respect to a set of characteristics.
Inputs An input vector is the data given as one input to the algorithm. This will usually be a vector (recall vectors from chapter 1). Written as x, with elements xi, where i runs from 0 to the number of input dimensions, n.
One-Hot this is a group of bits with a high bit 1 and .

ANN Glossary

.

Outputs The output vector is y, with elements yj, where j runs from 1 to the number of output dimensions, n. We can write y(x,W) to remind ourselves that the output depends on the inputs to the algorithm and the current set of weights

Overfitting occurs when comparing the complexity of hypothesis class H with the complexity of the function underlying the data, H is too complex, and the data is not enough to constrain it.

Precision To measure the performance of classification, precision is the fraction of detections reported by the model that were correct.

Stochastic Gradient Descent Also known as incremental gradient descent, it allows one to approximate the gradient with a single data point instead of all available data. At each step of the gradient descent, a randomly chosen data point is used to compute the gradient direction.

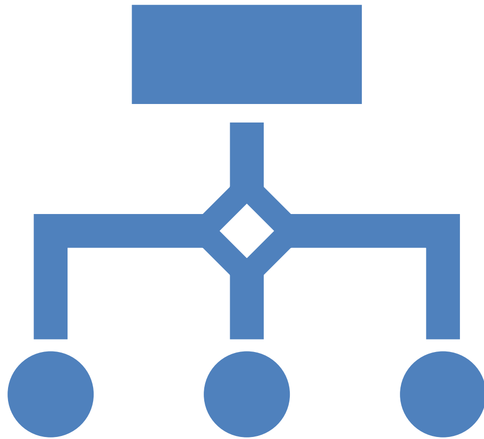
Targets The target vector t, with elements tj, where j runs from 1 to the number of output dimensions, n, are the extra data that we need for supervised learning, since they provide the 'correct' answers that the algorithm is learning about.

Tensors are primary data structure in TensorFlow programs. Tensors are N-dimensional data structures, most commonly scalars, vectors, or matrices. The elements of a Tensor can hold integer, floating-point, or string values.

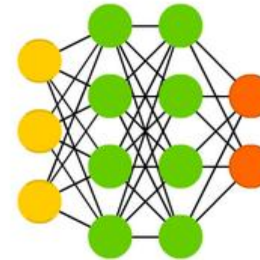
Training Set Data is typically divided into a training set and a testing set.

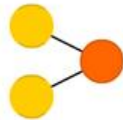
Underfitting occurs when comparing the complexity of hypothesis class H with the complexity of the function underlying the data, H is less complex than the function.

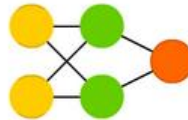
Weights wij, are the weighted connections between nodes i and j. For neural networks these weights are analogous to the synapses in the brain.

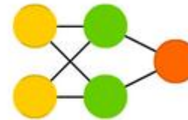


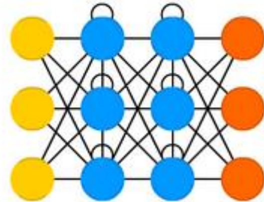
Neural Network Variations

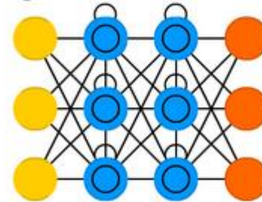


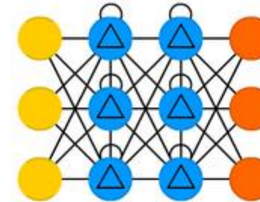


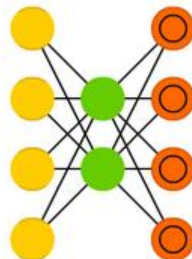


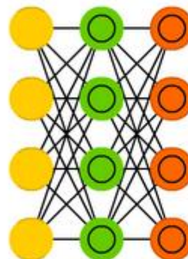


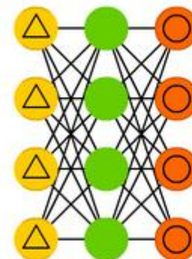








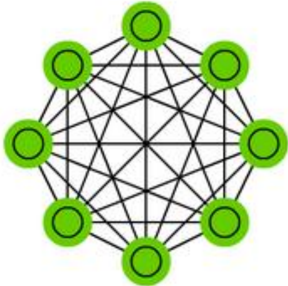




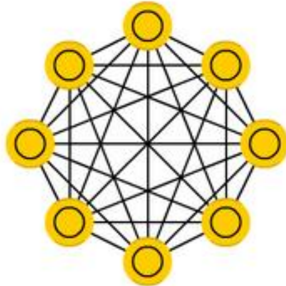
From https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

Neural Network Variations

Markov Chain (MC)

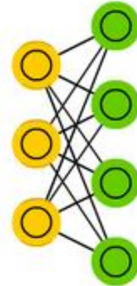


Hopfield Network (HN)

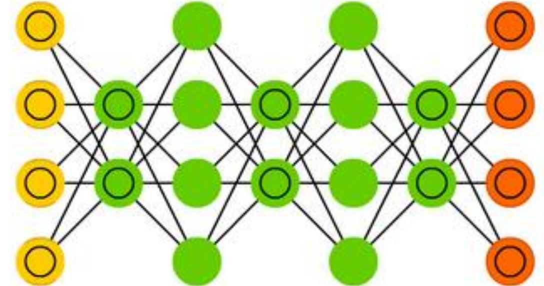


Boltzmann Machine (BM)

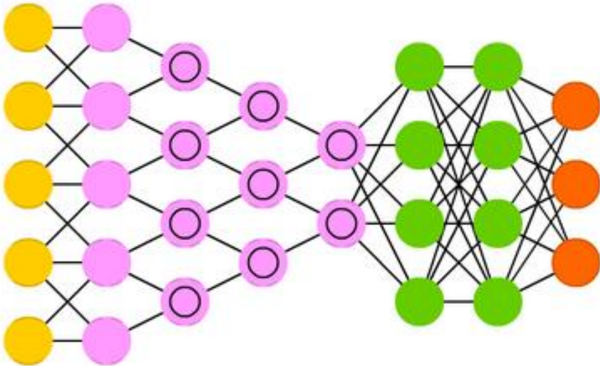
Restricted BM (RBM)



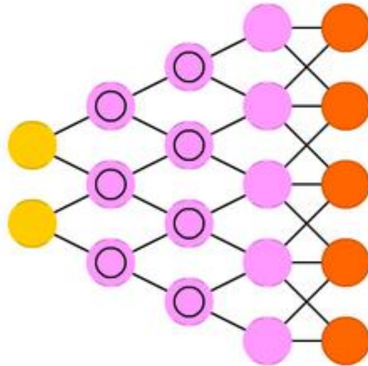
Deep Belief Network (DBN)



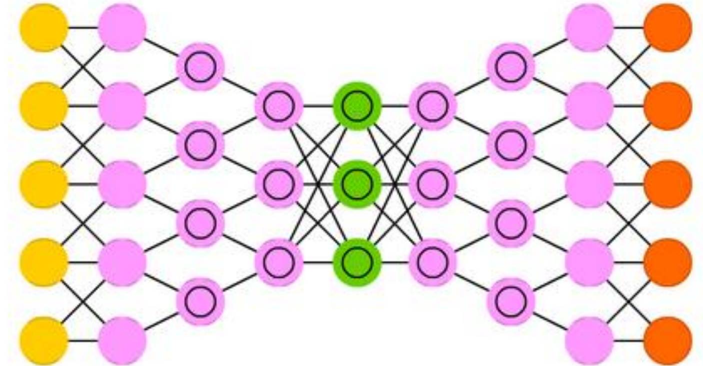
Deep Convolutional Network (DCN)



Deconvolutional Network (DN)



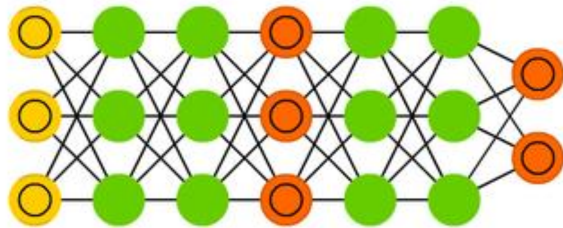
Deep Convolutional Inverse Graphics Network (DCIGN)

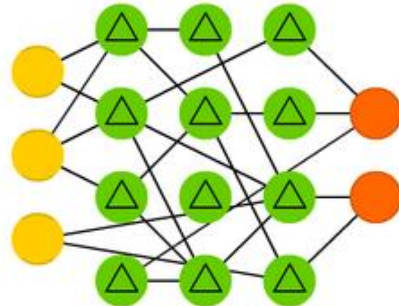


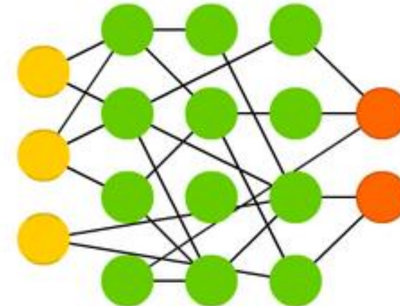
Generative Adversarial Network (GAN) Liquid State Machine (LSM) Extreme Learning Machine (ELM) Echo State Network (ESN)

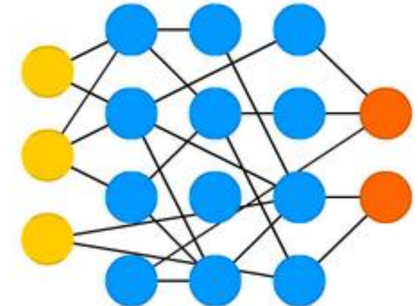
From https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

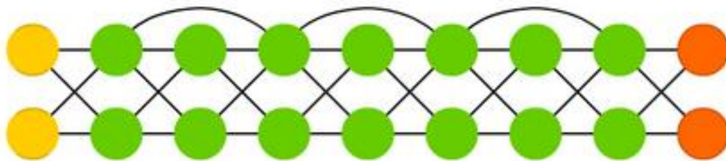
Neural Network Variations

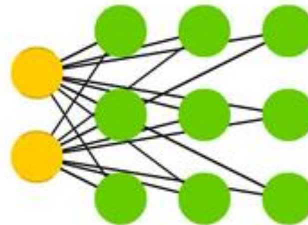


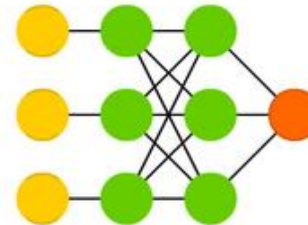


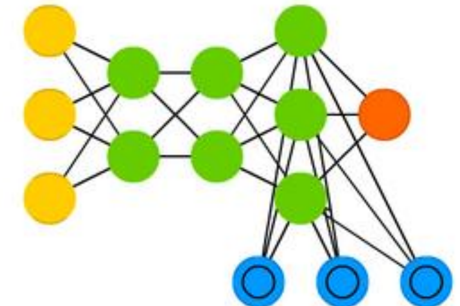








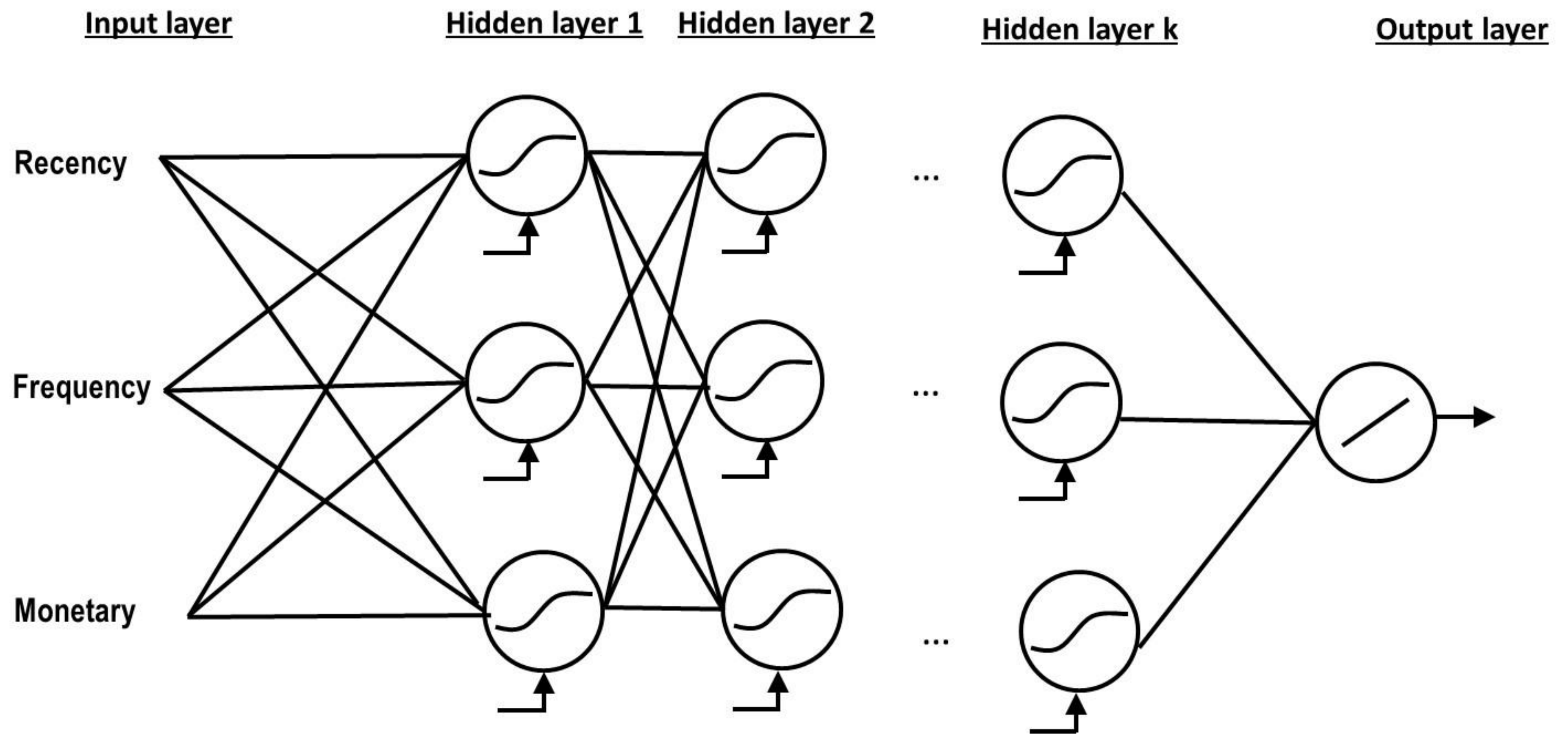




From https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

Deep learning

Essentially a multi-layer neural network



Activation Functions

An activation function takes the input and determines if the input to the current neuron is sufficient to cause that neuron to fire the next neuron. Activation functions define the learning pattern and determine the efficiency of the algorithm. This section describes some commonly used activation functions.

Google describes activation functions as "A function (for example, ReLU or sigmoid) that takes in the weighted sum of all of the inputs from the previous layer and then generates and passes an output value (typically nonlinear) to the next layer"

Activation Functions

ReLU (Rectified Linear Unit)
ReLU outputs zero for negative inputs and returns the input directly for positive values. It is computationally efficient and helps mitigate the vanishing gradient problem, making it the most commonly used activation function in hidden layers of deep neural networks.
Sigmoid (Logistic)
The sigmoid function maps inputs to a range between 0 and 1, making it useful for binary classification problems and probabilistic outputs. However, it suffers from vanishing gradients for very large or very small inputs, which can slow or prevent learning in deep networks.
Tanh (Hyperbolic Tangent)
The tanh function squashes inputs into a range between −1 and 1 and is zero-centered, which often leads to faster convergence than sigmoid. Like sigmoid, it can still suffer from vanishing gradient issues for extreme input values.
Leaky ReLU
Leaky ReLU is a variant of ReLU that allows a small, non-zero gradient for negative input values. This helps address the "dying ReLU" problem, where neurons become inactive and stop learning during training.
ELU (Exponential Linear Unit)
ELU uses an exponential function for negative inputs and a linear function for positive inputs. It tends to produce outputs closer to zero mean, which can speed up learning and improve robustness to noise compared to ReLU.

ReLU6

There are other variations of the ReLU function. One often used with Tensorflow is the ReLU6. This activation function has been shown to be faster than traditional ReLU. The ReLU6 function is defined by the formula in this equation.

$f(x) = \min(\max(0,x),6)$

Optimization Functions

Optimization algorithms are used to adjust parameters in order to minimize the cost (computational cost) of the function. For each of these algorithms, this section will present a general description. The details of these algorithms are not necessary because Tensorflow handles the mathematical implementation of the algorithm for you. This is not an exhaustive list of all optimization algorithms available in Tensorflow. This description of these key algorithms should help you understand optimization as a concept. You can view all current Tensorflow optimization algorithms at the Tensorflow website .

Gradient descent is an optimization algorithm used to minimize a given function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, gradient descents are useful to update the parameters of the model being used. Parameters refer to coefficients in linear regression and weights in neural networks. Put in more rigorous mathematical terms, the gradient descent algorithm is used to find the minimum of a function. Put more simply, gradient descent is an optimization algorithm used to find the values of parameters of a function that minimizes a cost function (computational cost). When using Tensorflow, the class SGD is the gradient descent optimizer. It should be noted that gradient descent is perhaps the most common optimization algorithm used.

Models

Models are basically files that are trained to recognize patterns. The model is trained using a set of training data. After training, the model can be used to make predictions from the data. NVIDIA, a graphics card and processor manufacturer, describes models in the following manner: “A machine learning model is an expression of an algorithm that combs through mountains of data to find patterns or make predictions. Fueled by data, machine learning (ML) models are the mathematical engines of artificial intelligence.” Tensorflow has an online repository of models that have already been trained for various purposes .

Gradient Descent (Batch Gradient Descent)

Gradient Descent is the foundational optimization algorithm for neural networks. It works by computing the gradient of the loss function with respect to all model parameters using the entire training dataset and then updating the parameters in the direction that minimizes the loss. While it provides stable and accurate updates, it can be computationally expensive and slow for large datasets because each update requires processing all training examples.

Adagrad (Adaptive Gradient Algorithm)

Adagrad adapts the learning rate individually for each parameter based on how frequently it has been updated. Parameters associated with rare features receive larger updates, while frequently updated parameters receive smaller ones. This makes Adagrad particularly useful for sparse data, though its learning rate can decay too aggressively, leading to premature convergence.

Adam (Adaptive Moment Estimation)

Adam combines the advantages of momentum and RMSprop by maintaining both a moving average of gradients and a moving average of squared gradients. This allows Adam to adapt learning rates for each parameter while benefiting from momentum-based acceleration. Adam is computationally efficient, requires little tuning, and is one of the most commonly used optimizers in modern neural networks.

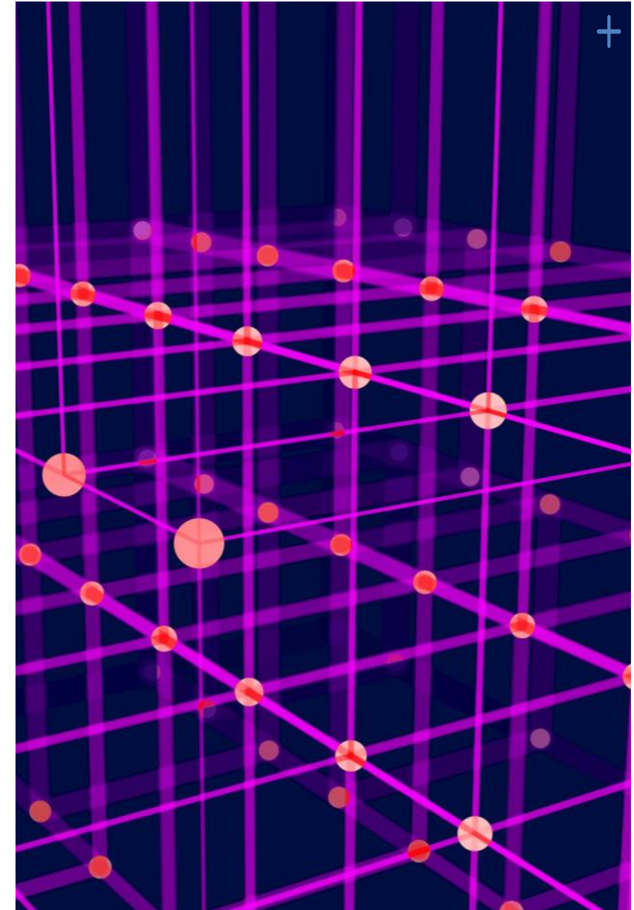
Momentum

Momentum enhances gradient descent by accumulating a moving average of past gradients and using it to update parameters. This helps accelerate learning in directions with consistent gradients and reduces oscillations in directions where gradients change frequently. As a result, momentum allows the optimizer to converge faster and more smoothly, particularly in deep networks with complex loss surfaces.

NADAM is a variation of ADAM that uses a Nesterov Momentum. This, of course, necessitates a discussion of what a Nesterov momentum is. Any gradient descent can be modified with momentum. In this context, momentum is some adjustment to the gradient descent parameter so that movement through the parameter space is averaged over multiple steps. Normally this is done by introducing velocity. The goal is that momentum will increase in those directions that lead to the most improvement. Nesterov momentum is a variation of that concept of momentum. Rather than calculate momentum with the actual positions in the search space, it calculates based on projected positions in the search space.

AdaGrad

- Adaptive Gradient (AdaGrad) is actually a group of closely related algorithms. As the name suggests, is a variation of gradient descent. A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable, thus AdaGrad seeks to overcome that limitation. AdaGrad allows step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients observed for the variable.



Optimization Functions

Optimizer	Learning Rate Type	Uses Momentum	Adaptive per-Parameter LR	Main Advantages	Main Disadvantages	Typical Use Cases
Batch Gradient Descent	Fixed	No	No	Stable, exact gradient	Very slow; high memory cost	Small datasets, theoretical analysis
Stochastic Gradient Descent (SGD)	Fixed	No	No	Fast updates; escapes local minima	Noisy convergence	Large datasets, online learning
Mini-Batch GD	Fixed	Optional	No	Efficient; GPU-friendly	Needs batch-size tuning	Most practical NN training
SGD + Momentum	Fixed	Yes	No	Faster convergence; reduced oscillation	Extra hyperparameter	Deep CNNs, vision tasks
Nesterov (NAG)	Fixed	Yes (look-ahead)	No	More accurate updates	Slightly complex	Deep networks needing stability
Adagrad	Adaptive	No	Yes	Works well with sparse data	Learning rate decays too fast	NLP, sparse features
RMSprop	Adaptive	No	Yes	Handles non-stationary objectives	Sensitive to hyperparameters	RNNs, deep learning
Adam	Adaptive	Yes	Yes	Fast convergence; little tuning	Can generalize poorly	Default choice for many models
AdamW	Adaptive	Yes	Yes	Better regularization than Adam	Slightly more computation	Transformers, large models
Nadam	Adaptive	Yes (Nesterov)	Yes	Faster, smoother convergence	More complex updates	Deep and complex architectures

Tensorflow Basics

TensorFlow was first developed by the GoogleBrain team for internal use. It was released to the public in late 2015. Current version is Tensorflow 2.0 TensorFlow provides Python and C APIs. It does not guarantee backwards compatibility. There are packages for C++, Java, Go, Swift, Matlab, C#, and others.

However, we will focus on Python

.

Tensorflow Basics

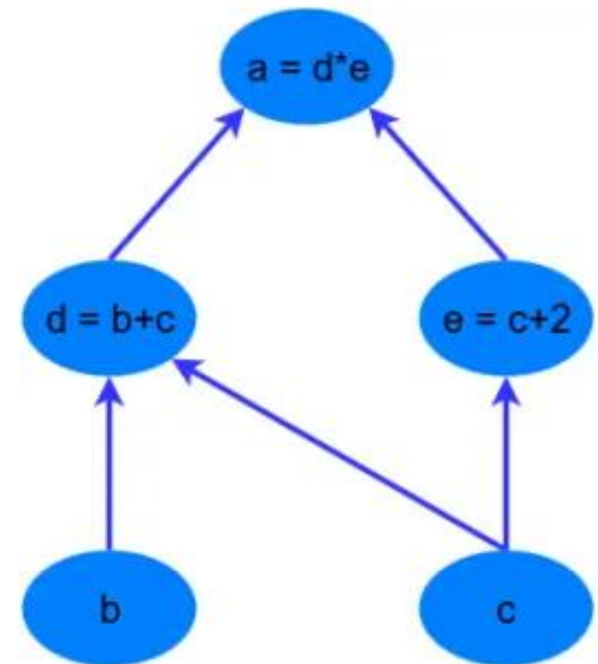
Tensors are multi-dimensional arrays with a uniform type.
TensorFlow is based on graph-based computation. A computational graph is a network of nodes and edges. This is an alternative way of conceptualizing mathematical calculations. Consider the expression:

We can break this function down into the following components

It can be represented graphically

$$a = (b + c) * (c + 2)$$

$$d = b + c$$
$$e = c + 2$$
$$a = d * e$$



Tensorflow Basics

The API known as TensorFlow core provides fine-grained lower level functionality. Because of this, this low-level API offers complete control while being used on models.

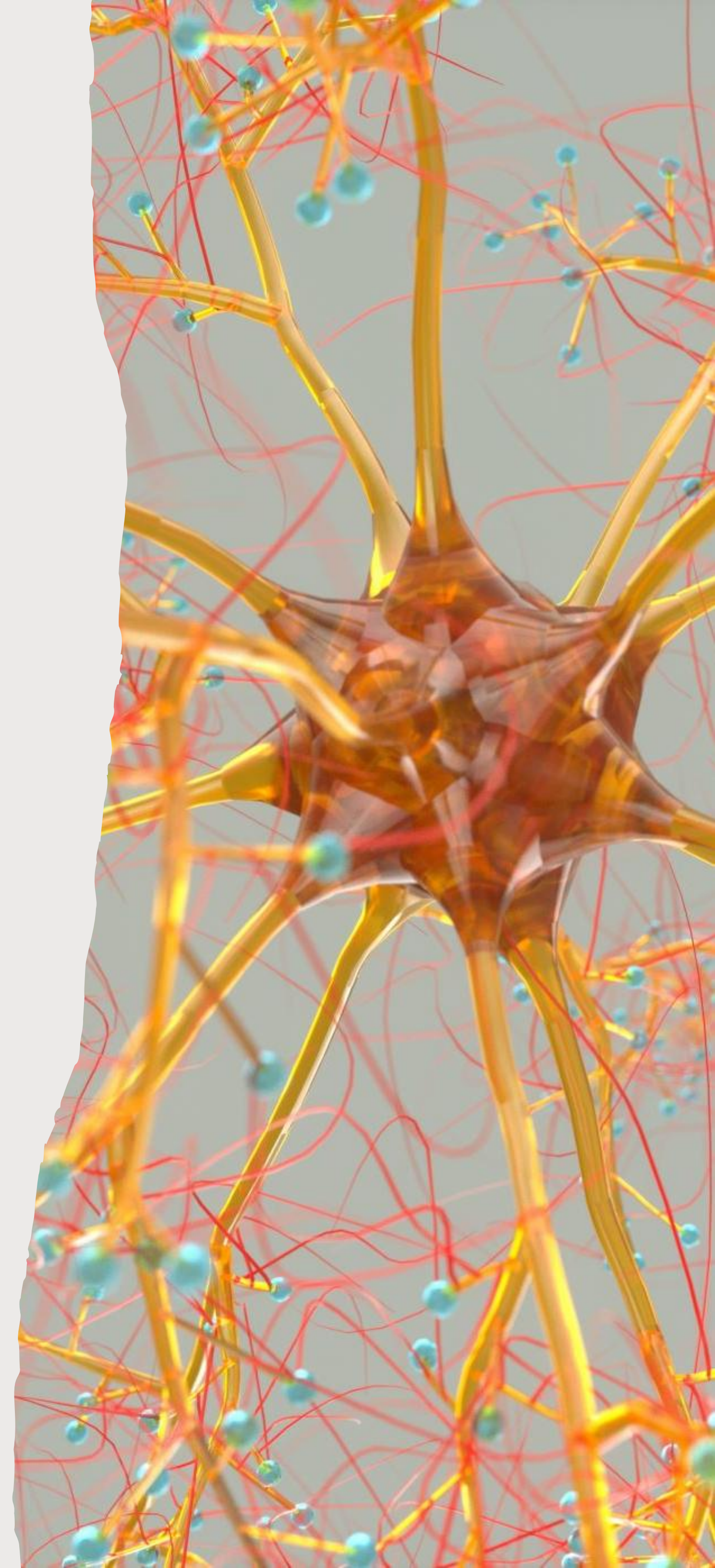
High-level API: These APIs provide high-level functionalities that have been built on TensorFlow core and are comparatively easier to learn and implement. Some high-level APIs include Estimators, Keras, TFLearn, TFSlim, and Sonnet.

.

Neural Networks in Tensorflow

There are three ways of creating a model in tf.keras: Sequential API , Functional API, and Model subclassing. Sequential is simplest and you will see most.

If each neuron in a layer is connected to all neurons located in the previous layer, and to all the neurons in the following layer is referred to as dense



Neural Networks in Tensorflow

BATCH_SIZE is the number of samples you feed in to your network at a time,

EPOCK is how many iterations

VALIDATION_SPLIT is how much of the training data is reserved for validation

RESHAPING is for the input data. For example, a 20X20 pixel image could be reshaped into 400 neurons/nodes, 1 for each pixel

NB_CLASSES = 10

RESHAPED = 784

```
model = tf.keras.models.Sequential()
model.add( keras.layers.Dense(
NB_CLASSES,
```

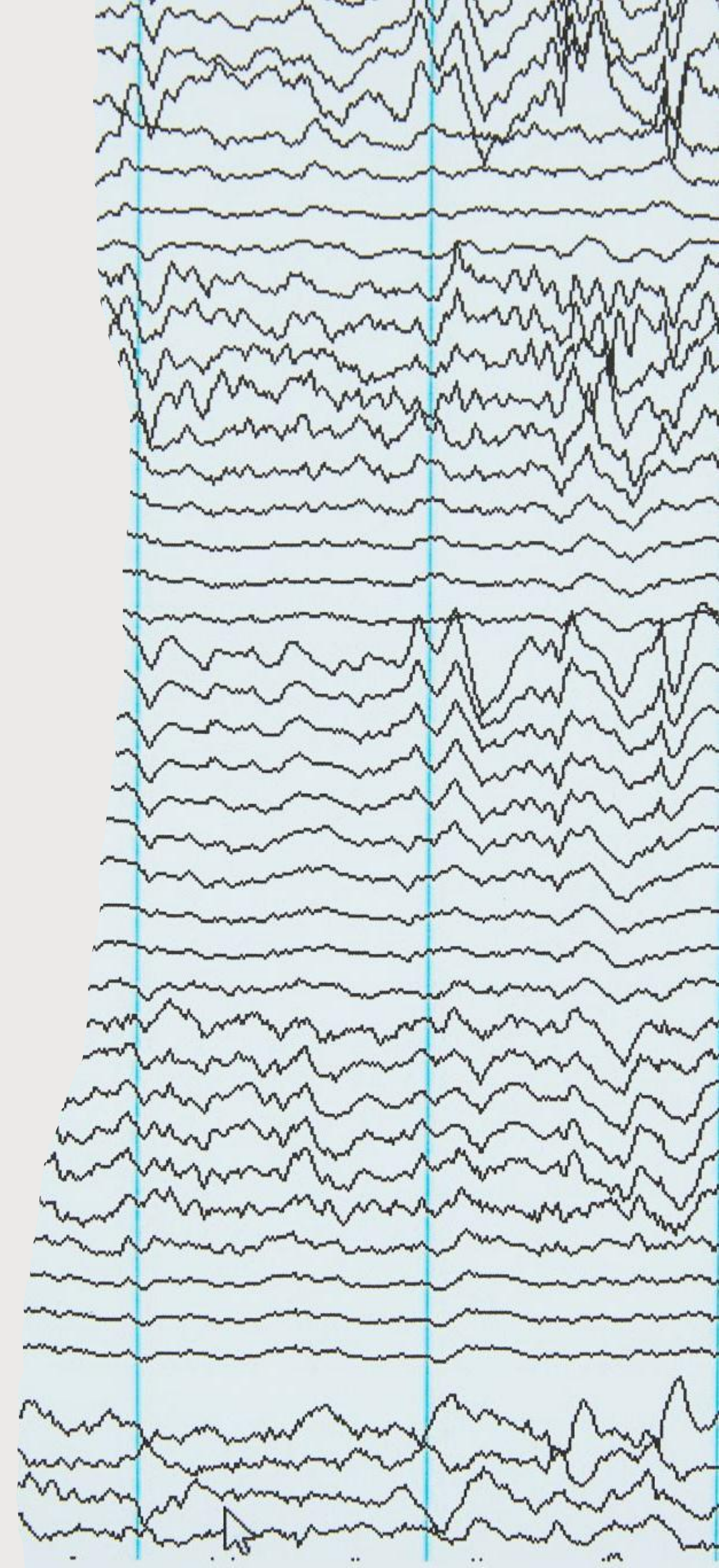
Neural Networks in Tensorflow

Initializers

- random_uniform: Weights are initialized to uniformly random small values in the range -0.05 to 0.05.
- random_normal: Weights are initialized according to a Gaussian distribution, with zero mean and a small standard deviation of 0.05. For those of you who are not familiar with Gaussian distribution, think about a symmetric "bell curve" shape.
- zero: All weights are initialized to zero.

Encoding

- One-hot encoding (OHE)
- If you are encoding digits 0 to 9, then 4 would be encoded as
- [0,0,0,0,1,0,0,0,0,0}



Neural Networks in Tensorflow

Activation functions

- sigmoid
- tanh
- relu
- ELU
- LeakyReLU

You will see softmax frequently, it is an implementation of sigmoid

Neural Networks in Tensorflow

Some common choices for metrics are:

- Accuracy, which defines the proportion of correct predictions with respect to the targets
- Precision, which defines how many selected items are relevant for a multi-label classification
- Recall, which defines how many selected items are relevant for a multi-label classification

Now back to Python

The first step will always be

#!/usr/bin/python

import tensorflow as tf

Tensorflow Data Types

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

Keras Tool

Keras is a high-level library in Python supporting some data analytics tools (e.g., neural network)

It is built on a well-established low-level library (e.g., TensorFlow and Theano)

Keras is the first high-level library added to the core TensorFlow at Google.

It was started to be incorporated in 2017.

Many industrial people uses Keras (on top of TensorFlow).

Keras Tool

There are many other existing Python libraries for data analytics

- TensorFlow: Developed for Google Brain
- Theano: University of Montreal
- SciKit-Learn: A student in Google's Summer of Code
- Caffe: UC Berkley
- PyTorch: Facebook
- Microsoft Cognitive Toolkit: Microsoft
- Apache MXNet: Apache Software Foundation

Gradient Descent

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

Ultimately this algorithm was designed to find the minimum of a function. Starting at the top of the mountain, we take our first step downhill in the direction specified by the negative gradient. Next we recalculate the negative gradient (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this process iteratively until we get to the bottom of our graph, or to a point where we can no longer move downhill–a local minimum. image source. The size of these steps is called the learning rate. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing.

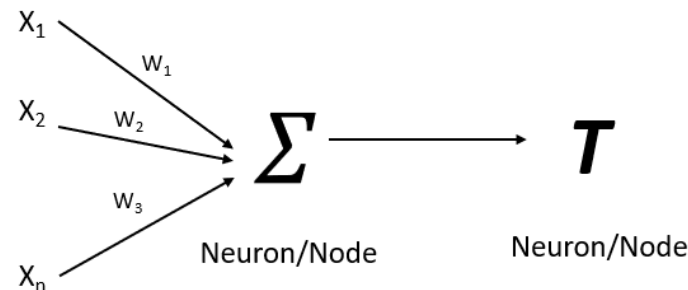
Put simply: Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

Feed Forward Network

The name of these networks derives from the fact that outputs from nodes in one layer can only go to nodes in the next layer. There is no possibility for a cycle or loopback. The information 'feeds forward', thus the name. This is among the simplest forms of a neural network. In fact, many of the concepts in the Feed Forward Network are found in all neural networks. Thus, this particular variation bears close study.

The simplest type of feed forward network is the single layer perceptron. The concept of a perceptron predates not only machine learning, but even digital computers. Warren McCulloch and Walter Pitts published an interesting paper in 1943 intitled "A Logical Calculus of Ideas Immanent in Nervous Activity". Their goal was to simply model how neurons work. This modeling led, eventually, to the creation of artificial neurons. Each neuron takes in input, sums the input, and based on that summation, determines if it will fire or not.

https://link.springer.com/content/pdf/10.1007/BF02478259.pdf



Preceding nodes (X_1 through X_n) send input to the node in question. Each of these has a specific weight associated with it (W_1 through W_n). That node sums the input (thus the summation symbol) and if the sum of the weighted input meets or exceeds a specific threshold, then the node in question will fire and send a signal to the next node, the target node (thus the T representing it). The McCulloh Pitts neuron leads us to what is usually called Heb's rule. Hebb's rule was discussed previously, but as a reminder, consider this quote from another machine learning textbook "Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. So, if two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger. However, if the two neurons never fire simultaneously, the connection between them will die away. The idea is that if two neurons both respond to something, then they should be connected.

Back Propagation

This is an algorithm used in training artificial neural networks, particularly with feedforward artificial neural networks. Backpropagation is used to compute the gradient of the loss function with respect to the weights. The loss function is also sometimes called a cost or error function. It is a function that is used to represent some cost associated with some activity or event.

The backpropagation algorithm is used to train a neural network using a technique called chain rule. After each forward pass through a network, backpropagation performs a backward pass while adjusting the model's parameters. Normally the parameters adjusted are the weights and biases.

This algorithm was first introduced in the 1970's but achieved wide recognition due to a 1986 paper in the journal Nature written by David Rumelhart, Geoffrey Hitton, and Ronald Williams. It is useful to consider how these authors described backpropagation:

"The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure"

Another description from the International Dictionary of Artificial Intelligence could be useful in aiding your understanding of backpropagation:

"A classical method for error propagation when training Artificial Neural Networks (ANNs). For standard backpropagation, the parameters of each node are changed according to the local error gradient. The method can be very slow to converge although it can be improved through the use of methods that slow the error propagation and by batch processing. Many alternate methods such as the conjugate gradient and Levenberg-Marquardt algorithms are more effective and reliable."

Fortunately, the backpropagation algorithm is taken care of by libraries such as TensorFlow. You don't have to implement the details yourself. I say this is fortunate, because this algorithm uses partial derivatives which may be beyond some readers mathematical skillset.

https://www.nature.com/articles/323533a0

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.375.8194&rep=rep1&type=pdf

Recurrent Neural Network

The recurrent neural network (RNN) is a common variation of neural networks. These are derived from feedforward neural networks. Recurrent nets are a variation of artificial neural network intended to distinguish patterns in sequences of data. The patterns can be almost anything, but the pattern recognition makes RNN's idea for identifying handwriting, numbers, time series data from sensors, stock market data, and more. RNN's consider time and sequence. This gives these algorithms a temporal dimension.

The manner in which RNNs function is to have cycles that permit output from some nodes to affect subsequent input into the same nodes. There are several types of RNN's including:

- Encoder decoder or sequence to sequence RNNs,
- Bidirectional RNNs,
- Recursive RNNs,
- Gated Recurrent Unit (GRU),
- LSTM RNNs.

The following description of RNN's may help clarify this class of algorithms for you "A family of neural networks for processing sequential data. RNNs share the same weights across different discrete time steps: each member of the output is a function of the previous members of the output; each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph. RNNs are very powerful dynamic systems for tasks that involve sequential inputs, such as speech and language."

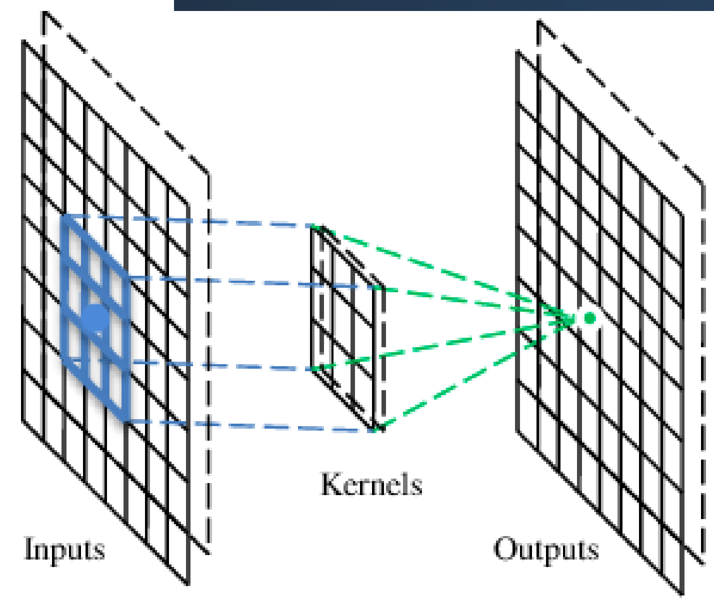
Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. Deep Learning (p. 373). MIT Press.

LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey. Deep Learning. Nature 521, 436-444 (2015)

Convolutional Neural Network

A convolutional neural network (CNN) is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that do convolutional operations.

The different layers perform different aspects of the learning. Convolutional Neural Networks (sometimes called ConvNets) are often used with visual imagery. This makes them an ideal candidate for examining diagnostic imagery. Put more formally, convolutional neural networks use a mathematical operation called convolution rather than basic matrix multiplication in at least one of the layers. The architecture of a convolutional neural network will include an input layer, one or more hidden layers, and an output layer. The input is provided as a tensor with a shape. The convolutional neural layer transforms that image to a feature map with a shape. Some sources refer to the feature map as an activation map. The convolutional layers each respond to input only for their own receptive field. This is quite similar to how the brain processes visual imagery.



GAN

A GAN consists of **two neural networks competing against each other**:
Generator (G)
Discriminator (D)
These networks are trained simultaneously in a **competitive process similar to a game**.
The **generator** tries to create fake data that looks real.
The **discriminator** tries to distinguish between real data and fake data produced by the generator.
Through this adversarial training process, the generator gradually becomes better at producing realistic data.

1. Generator Network
Purpose: Create synthetic data that resembles real data.
Input
Random noise vector (usually from a Gaussian or uniform distribution)
Output
Synthetic sample (e.g., image, audio, text)
Function
The generator transforms random noise into structured outputs using neural network layers.
Example process:
Random noise → Neural network → Generated image
The generator learns the **distribution of the training data** so it can produce realistic samples.

2. Discriminator Network
Purpose: Determine whether data is real or generated.
Input
Real data from the dataset
Fake data from the generator
Output
Probability that the input is real
Example output:
Real image → 0.98 probability real
Generated image → 0.05 probability real
The discriminator is essentially a **binary classifier**.

Adversarial Training Process

GAN training works like a **two-player minimax game**.

Step 1: Train the Discriminator

The discriminator learns to distinguish:

Real samples from the dataset

Fake samples produced by the generator

Goal: maximize classification accuracy.

Step 2: Train the Generator

The generator attempts to **fool the discriminator** by producing more realistic samples.

Goal: minimize the discriminator's ability to detect fake data.

The following is a mathematical description of a GAN

The GAN objective function is:

$$\min_G \max_D V(D, G)$$

Where:

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

Meaning:

- Discriminator maximizes correct classification of real and fake data.
- Generator minimizes discriminator success.

This creates a **dynamic equilibrium** where the generator produces highly realistic samples.

GAN

Typical GAN training procedure:

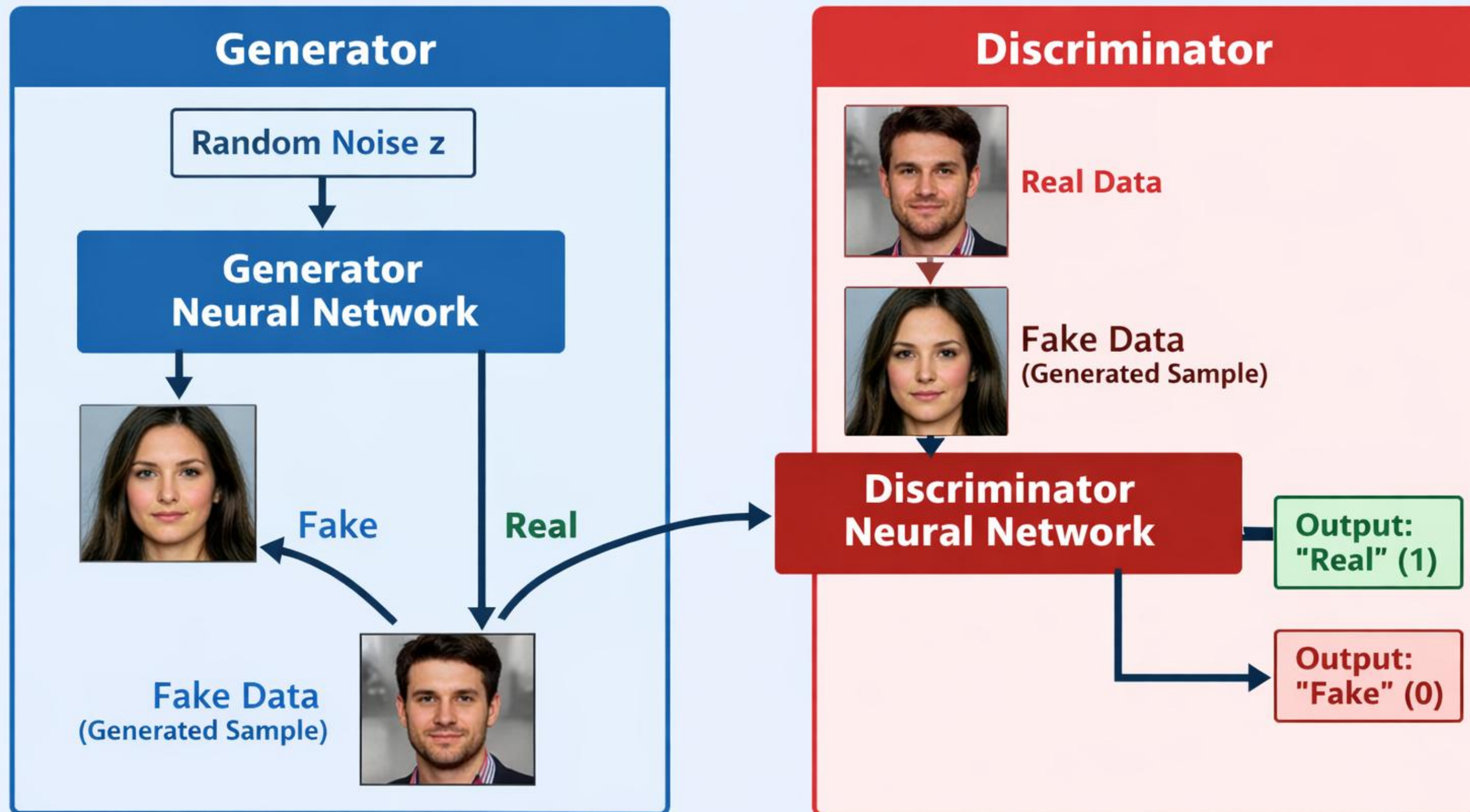
1. Sample real data from dataset
2. Sample random noise
3. Generate fake samples
4. Train discriminator using real and fake samples
5. Train generator using discriminator feedback
6. Repeat many iterations

Over time:

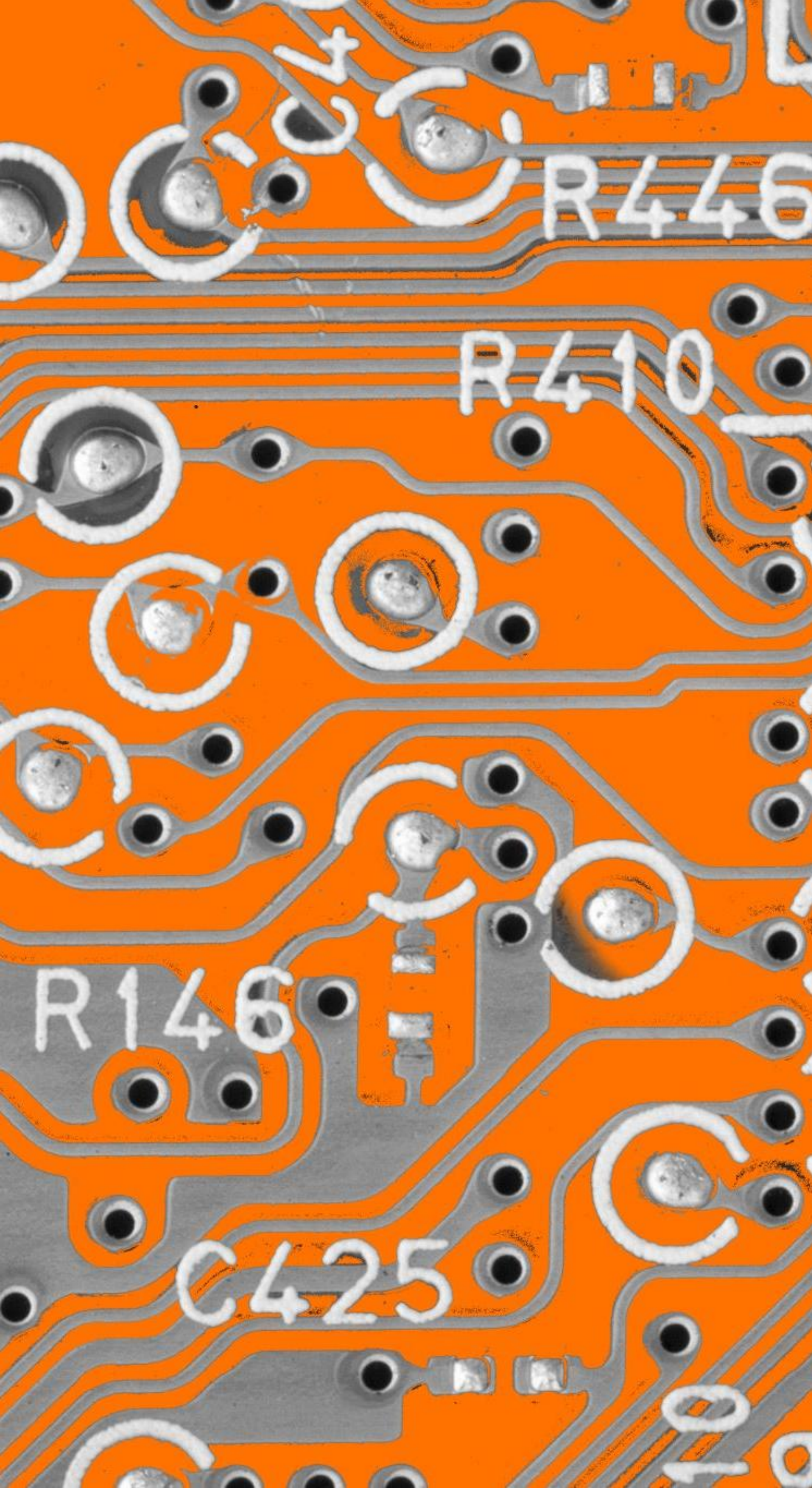
Discriminator becomes more accurate

Generator becomes more realistic

Eventually, the generator produces samples that are **indistinguishable from real data**.



GAN



Autoencoder

An autoencoder has two main parts:

1. Encoder

Takes input data (e.g., a vector of features)

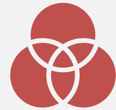
Compresses it into a smaller representation called the **latent space** (or embedding)

2. Decoder

Takes that compressed representation

Tries to reconstruct the original input

Autoencoder – common uses



Dimensionality reduction (like PCA, but nonlinear)

Feature learning

Noise reduction (denoising autoencoders)

Anomaly detection

Autoencoder

- An autoencoder is a type of artificial neural network designed to learn a compressed, meaningful representation of data in an unsupervised manner by training the network to reconstruct its own input. At a high level, it consists of two main components: an encoder, which maps the input data $x$into a lower-dimensional latent representation z, and a decoder, which attempts to reconstruct the original input from that latent code. The network is trained by minimizing a reconstruction loss function—commonly mean squared error (for continuous data) or binary cross-entropy (for binary data)—that measures the difference between the original input and its reconstruction. Through this process, the autoencoder is forced to capture the most salient features of the data while discarding noise or redundancy, effectively performing nonlinear dimensionality reduction.

- The encoder can be thought of as a feature extractor. It applies a series of transformations (typically layers of linear mappings followed by nonlinear activation functions such as ReLU or sigmoid) to compress the input into a compact latent vector. This latent space, often called the bottleneck, is critical: if it has lower dimensionality than the input, the network must learn an efficient encoding that preserves essential structure. The decoder mirrors this process, taking the latent vector and transforming it back into the original input space. In a well-trained autoencoder, the latent representation captures underlying patterns such as correlations, structure, or manifold geometry, rather than simply memorizing the data.

Autoencoder

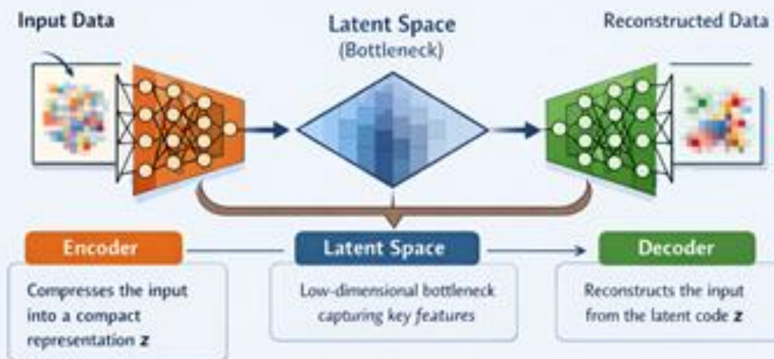
- Autoencoders are closely related to classical dimensionality reduction techniques like Principal Component Analysis (PCA), but they are significantly more powerful because they can model nonlinear relationships. While PCA finds a linear subspace that maximizes variance, autoencoders can learn complex, curved manifolds in high-dimensional data. This makes them especially useful for tasks such as image compression, denoising, anomaly detection, and representation learning. For example, in image data, an autoencoder might learn to encode edges, textures, or object parts in its latent space, even without any labels.
- There are several important variants of autoencoders that extend the basic idea. A denoising autoencoder is trained by corrupting the input (e.g., adding noise) and forcing the network to reconstruct the clean version, which encourages robustness and feature learning. A sparse autoencoder introduces a sparsity constraint on the latent representation, ensuring that only a small number of neurons are active at once, which promotes interpretability and feature disentanglement. A variational autoencoder (VAE) takes a probabilistic approach by modeling the latent space as a distribution rather than a single point, enabling generative capabilities such as sampling new data points. These variants highlight that autoencoders are not just compression tools, but flexible frameworks for learning structured representations.

Autoencoder

Autoencoders

Unsupervised Learning & Dimensionality Reduction

An **autoencoder** is a neural network that learns to encode and **decode** data for reconstruction.



Typical Applications

Variants of Autoencoders

Restricted Boltzman Machine

Restricted Boltzmann Machines (RBMs)

The fundamental building block of a DBN is the **Restricted Boltzmann Machine**.

RBM Components

An RBM consists of:

Visible units (v) – represent observed data

Hidden units (h) – capture hidden patterns

Weights (W) connecting visible and hidden layers

Important property:

There are **no connections within the same layer**.

Visible Layer ↔ Hidden Layer

This restriction simplifies learning.

Restricted Boltzman Machine

Probabilistic Model
RBMs are energy-based models.
The probability distribution is defined by an energy function:

The probability distribution is defined by an **energy function**:

$$E(v,h) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_{i,j} v_i W_{ij} h_j$$

Where:

- v = visible units
- h = hidden units
- W_{ij} = weights
- b_i, c_j = biases

Lower energy states correspond to more probable configurations.

Deep Belief Network

A **Deep Belief Network** is a **generative probabilistic model** composed of several stacked layers of **Restricted Boltzmann Machines (RBMs)**.
Each layer learns to represent patterns in the data at increasing levels of abstraction.
Example hierarchy:
Raw input data

Low-level features

Intermediate patterns

High-level concepts
For example in image recognition:
Pixels → edges → shapes → objects

Architecture of a Deep Belief Network
A DBN typically consists of:
Visible layer
Multiple hidden layers
Stacked Restricted Boltzmann Machines
Structure
Input Layer (Visible Units)

Hidden Layer 1 (RBM)

Hidden Layer 2 (RBM)

Hidden Layer 3 (RBM)

Output / Classifier
Each RBM learns features from the output of the previous layer.

Deep Belief Network

Training occurs in **two phases**:

1. Unsupervised Pretraining

Each RBM is trained **layer-by-layer** using unsupervised learning.

Process:

Train the first RBM using input data

Use the hidden layer activations as input to the next RBM

Train the next RBM

Repeat for deeper layers

This is called **greedy layer-wise training**.

Why this helps

Early neural networks struggled with:

vanishing gradients

poor initialization

DBN pretraining provides **good weight initialization**.

2. Supervised Fine-Tuning

After pretraining:

A **classifier layer** is added

The entire network is trained using **backpropagation**

This improves performance for tasks like classification.

Deep Belief Network

RBMs are trained using **Contrastive Divergence (CD)**.

Steps:

Start with training data

Compute hidden unit activations

Reconstruct visible units

Update weights based on difference between original and reconstructed data

This approximates gradient descent on the likelihood function.

Contrastive Divergence

Contrastive Divergence is an approximate learning algorithm that updates model parameters by comparing:

statistics computed from the training data distribution

statistics computed from a reconstructed sample generated by the model

The algorithm adjusts model parameters to reduce the difference between the real data distribution and the model's reconstructed distribution.

Hopfield Network

A Hopfield network is a type of recurrent artificial neural network used primarily for associative memory and optimization problems. Introduced by physicist John Hopfield in 1982, it is one of the earliest neural network models that demonstrated how networks of simple processing units can collectively store and retrieve patterns. Hopfield networks are particularly important historically because they connect neural computation with ideas from statistical physics and energy minimization.

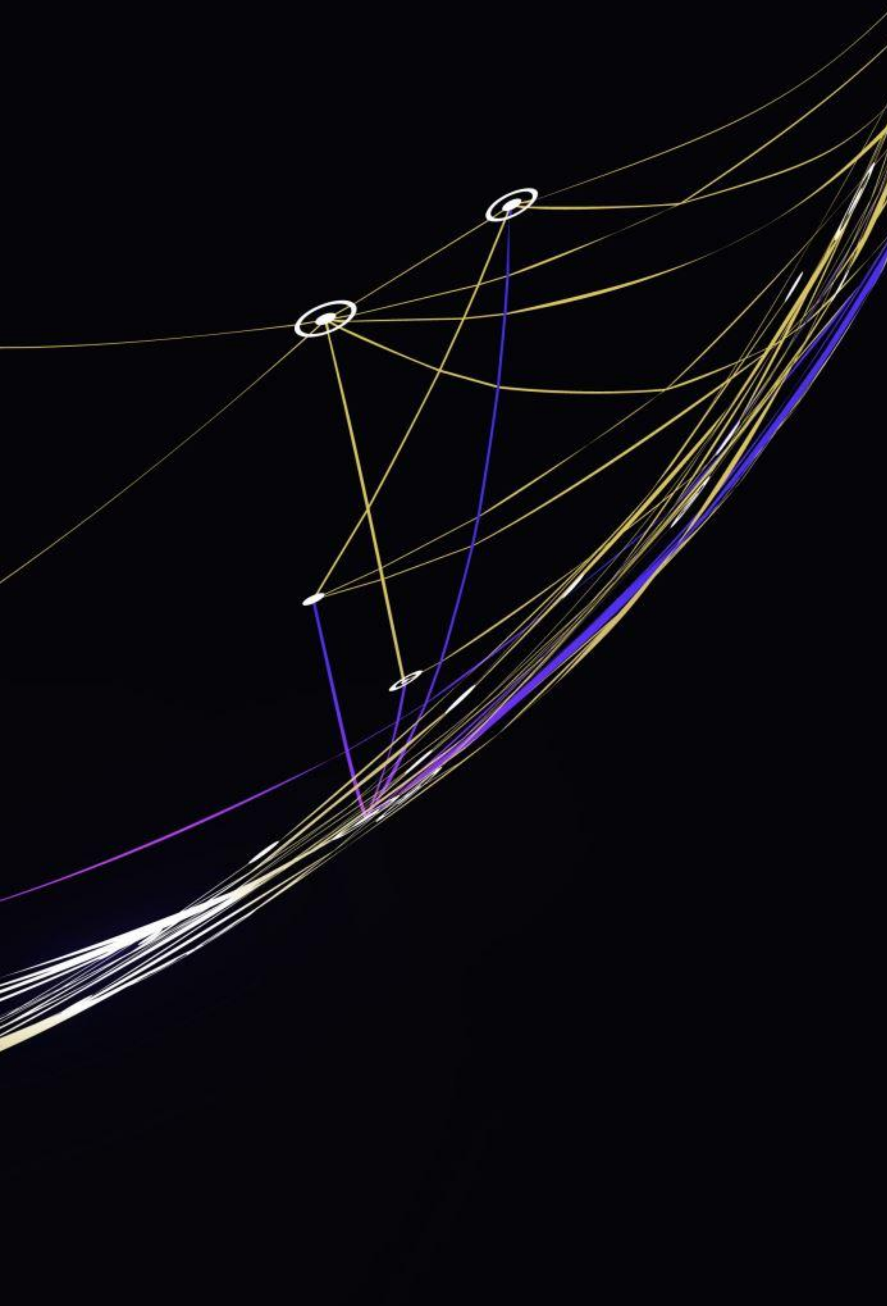
A Hopfield network consists of a fully connected set of neurons, where each neuron is connected to every other neuron in the network, but not to itself. The connections between neurons have symmetric weights, meaning the weight from neuron $i$to neuron $j$is the same as from $j$to i. Each neuron typically has a binary state, often represented as either $+1$or -1. The state of a neuron is updated based on the weighted sum of the states of all other neurons connected to it. If the sum exceeds a threshold, the neuron becomes $+1$; otherwise, it becomes -1. Because of the symmetry of the weights, the network exhibits well-defined dynamics that converge to stable configurations.

Hopfield Network

One of the defining features of a Hopfield network is its energy function, sometimes called a Lyapunov function, which provides a mathematical way to analyze the network's behavior. The network evolves over time in a way that monotonically decreases the energy, eventually settling into a local minimum. Each local minimum corresponds to a stored pattern or memory. The energy function is typically expressed as:

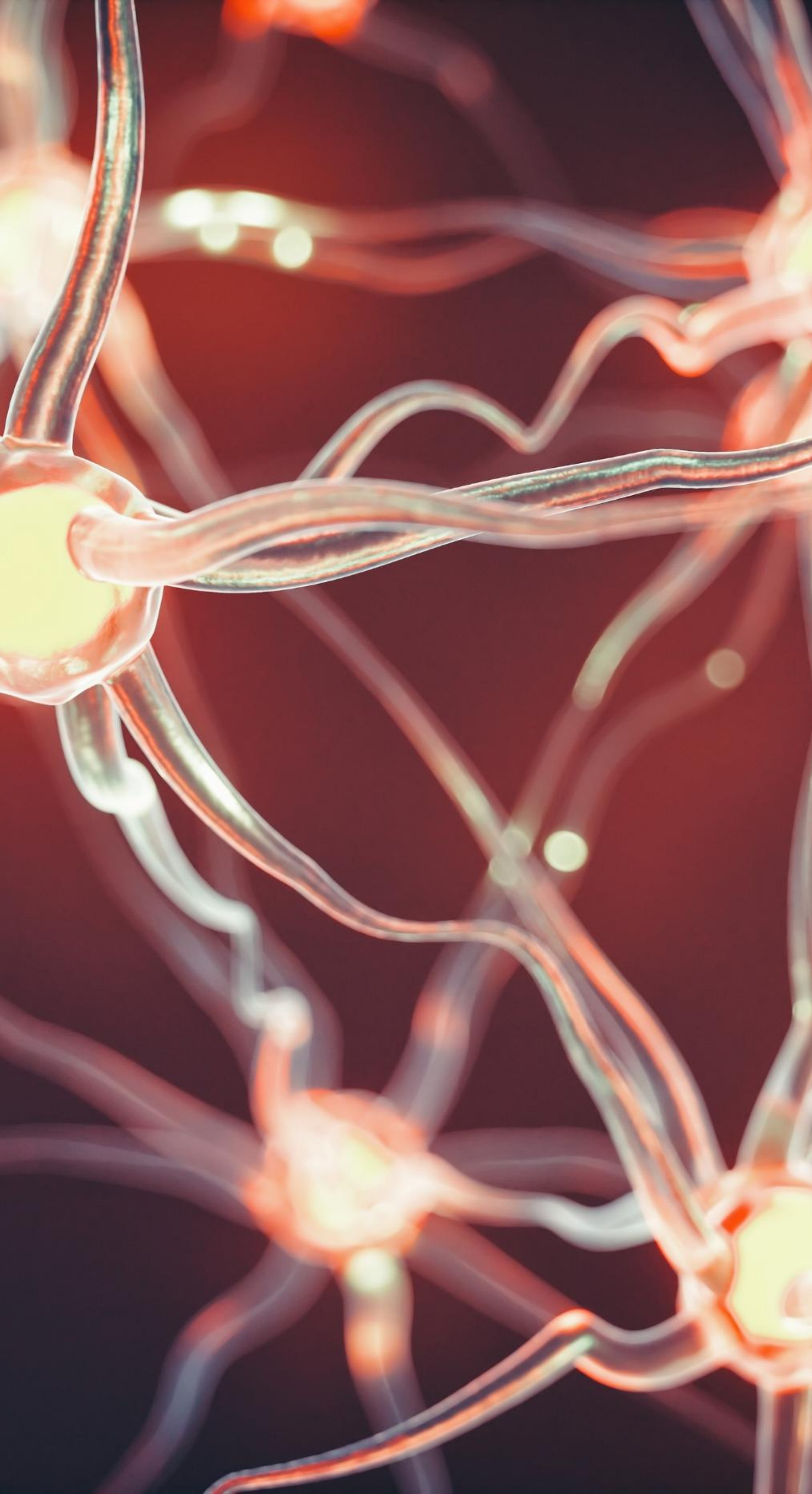
$$E = -\frac{1}{2} \sum_{i \neq j} w_{ij} s_i s_j$$

where w_{ij} is the weight between neurons i and j, and s_i is the state of neuron i. During operation, the network updates neuron states until no further changes occur, meaning the system has reached a stable state with minimal energy.



Hopfield Network

Hopfield networks function as content-addressable memory systems, meaning they can retrieve a stored pattern when given only partial or noisy information. For example, if several patterns are stored in the network and the network is initialized with a corrupted version of one of them, the network dynamics will converge toward the stored pattern that most closely resembles the input. This behavior makes Hopfield networks a classic model for associative memory, where memories are retrieved by similarity rather than by explicit address.



Hopfield Network

Patterns are typically stored in the network using a Hebbian learning rule, inspired by biological learning principles. The weights between neurons are calculated by summing the outer products of the stored patterns. For a set of patterns x^k ,the weight matrix is commonly defined as:

$$w_{ij} = \frac{1}{N} \sum_{k} x_i^k x_j^k$$

where N is the number of neurons and x_i^k is the i-th component of pattern k. This learning rule reinforces connections between neurons that are activated together.
Despite their elegant theoretical properties, Hopfield networks have limited storage capacity. For a network with N neurons, the number of patterns that can be reliably stored is approximately 0.138N. If too many patterns are stored, the network develops spurious states, which are unintended local minima that do not correspond to any real stored pattern. These spurious states arise from interference among stored memories.

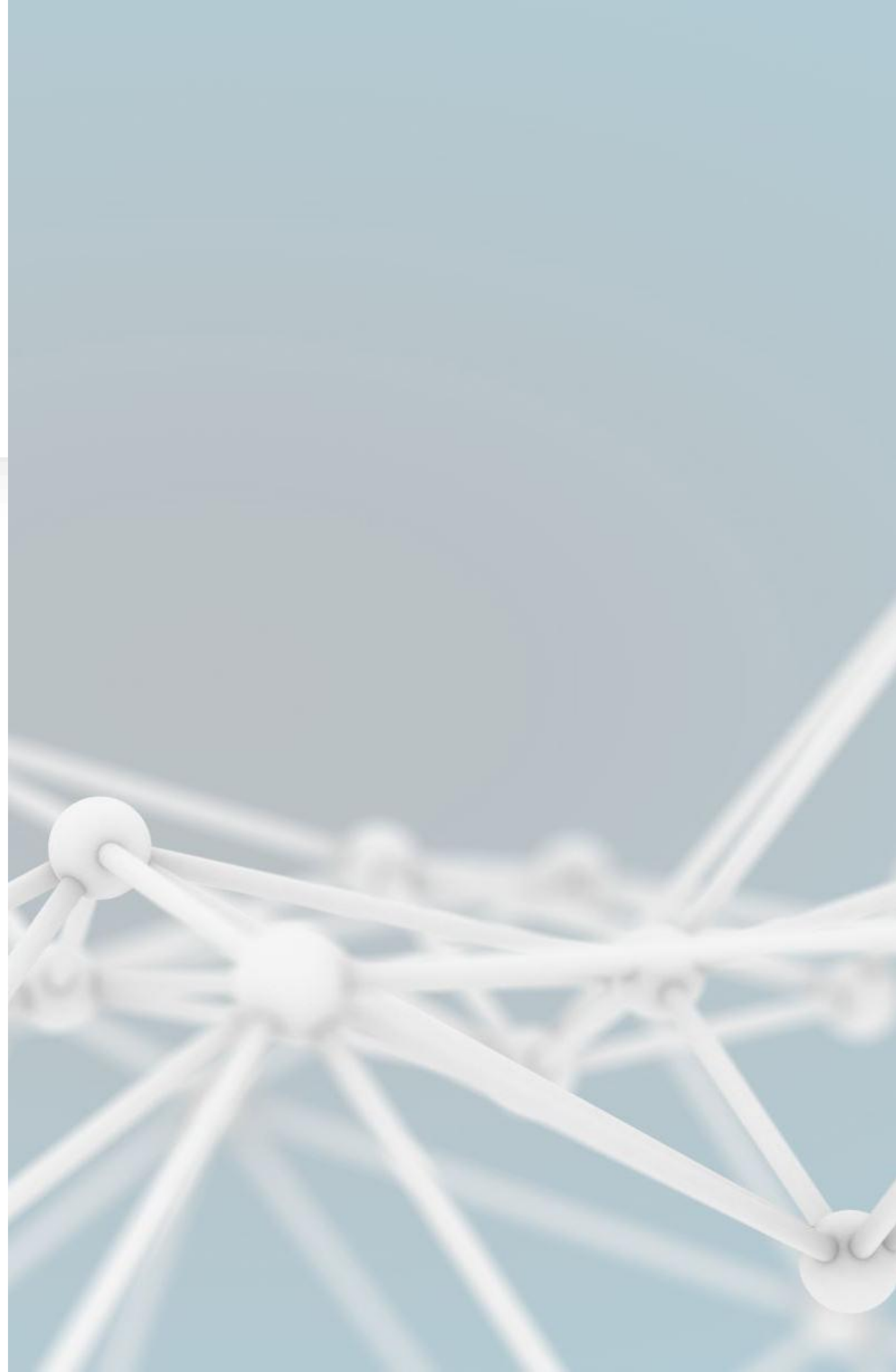
Hopfield Network

Hopfield networks can operate in either asynchronous or synchronous update modes. In asynchronous updating, one neuron is updated at a time, usually chosen randomly, which guarantees convergence to a stable state. In synchronous updating, all neurons update simultaneously, which can sometimes lead to oscillations or unstable behavior. Because of this, asynchronous updates are more common in theoretical analysis.

Beyond associative memory, Hopfield networks can also be used for combinatorial optimization problems. By designing the energy function appropriately, optimization problems such as the traveling salesman problem, scheduling, or constraint satisfaction can be mapped onto the network. The network then evolves toward a configuration that minimizes the energy function, ideally representing an optimal or near-optimal solution.

There are several variants of Hopfield networks. One important extension is the continuous Hopfield network, where neuron states take continuous values rather than binary ones. These models use sigmoid activation functions and differential equations to describe the network dynamics. Continuous Hopfield networks were used in early analog computing systems and have applications in optimization and control.

Hopfield Network



Another important modern development is the modern Hopfield network, which is closely related to the attention mechanism used in transformer models. Research has shown that transformer attention can be interpreted as a form of continuous Hopfield network with exponentially large storage capacity. This connection has renewed interest in Hopfield networks because it links classic neural models with modern deep learning architectures.

Although Hopfield networks are not widely used in modern large-scale machine learning systems, they remain important in the study of neural dynamics, memory models, and theoretical neuroscience. They provide a clear example of how complex collective behavior can emerge from simple interacting units, and they illustrate the deep relationship between machine learning, physics, and dynamical systems theory.

Deep Convolutional Network

A deep convolutional network, more commonly called a deep convolutional neural network (CNN), is a type of artificial neural network designed primarily for processing data that has a grid-like structure, such as images. CNNs are widely used in fields such as computer vision, medical imaging, autonomous vehicles, and pattern recognition because they are particularly effective at identifying spatial patterns in data. The term “deep” refers to the presence of multiple layers of processing, which allow the network to learn increasingly complex features from the input data.

In a deep convolutional network, the input—often an image represented as a matrix of pixel values—is passed through a sequence of specialized layers. The most important of these are convolutional layers, which apply small filters (also called kernels) that slide across the input image. Each filter detects a specific pattern such as edges, textures, or shapes. During training, the network learns the optimal values of these filters so that they respond strongly to useful visual features. Because the same filter is applied across the entire image, convolutional layers dramatically reduce the number of parameters compared with traditional fully connected neural networks, making them more efficient and better suited for large images.

Deep Convolutional Network

Following convolutional layers, CNNs often include activation functions such as the Rectified Linear Unit (ReLU). These functions introduce nonlinearity into the model, allowing the network to learn complex patterns rather than simple linear relationships. After activation, pooling layers are frequently used to reduce the spatial size of the feature maps produced by convolutional layers. Pooling operations, such as max pooling or average pooling, summarize regions of the image and help make the network more robust to small translations or distortions in the input.

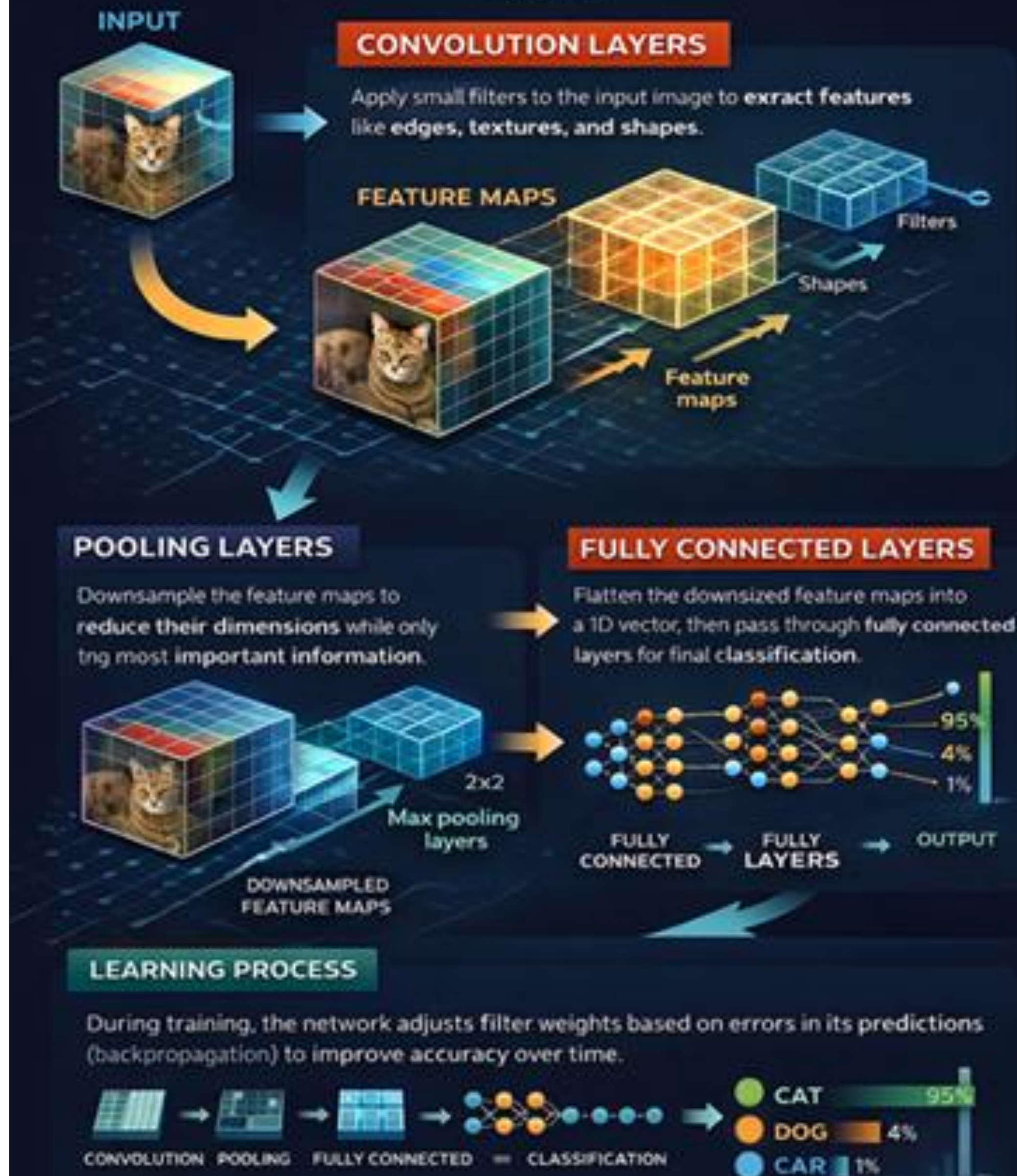
As the data moves deeper through the network, the learned features become more abstract. Early layers might detect simple edges or color gradients, while intermediate layers identify shapes or textures. The deepest layers capture high-level features, such as objects or object parts. These extracted features are then passed to fully connected layers, which perform the final classification or prediction task. For example, in an image recognition system, the final layer might output probabilities indicating whether the image contains a cat, dog, car, or other object.

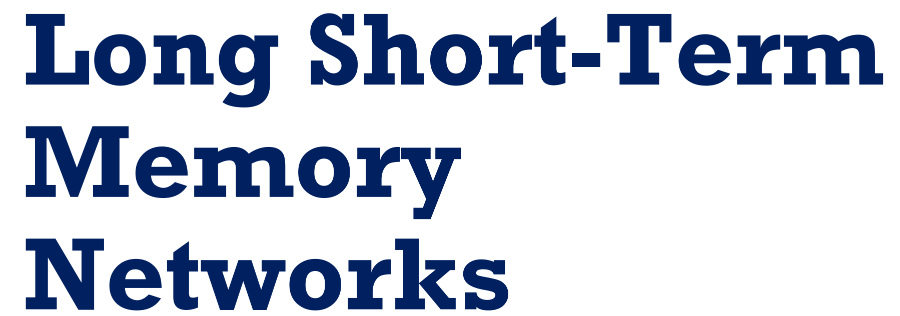
Deep Convolutional Network

Training a deep convolutional network involves presenting the model with many labeled examples and adjusting the network's weights through backpropagation and gradient descent. The network's predictions are compared with the correct labels using a loss function, and the error is propagated backward through the layers to update the filters and weights. Over time, the network learns to extract features that are most useful for making accurate predictions.

Deep convolutional networks have been responsible for major advances in artificial intelligence, especially after their success in large-scale image recognition competitions such as the ImageNet challenge. Architectures such as AlexNet, VGGNet, GoogLeNet (Inception), and ResNet demonstrated that deeper networks with carefully designed convolutional structures could significantly outperform earlier methods. Today, CNNs remain a fundamental building block in many AI systems and are often combined with other techniques such as recurrent networks or transformers to solve complex perception and analysis tasks.

Deep Convolutional Network



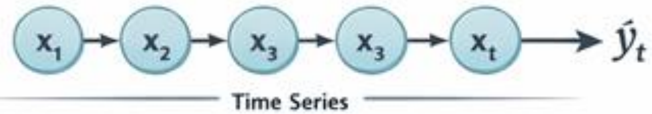


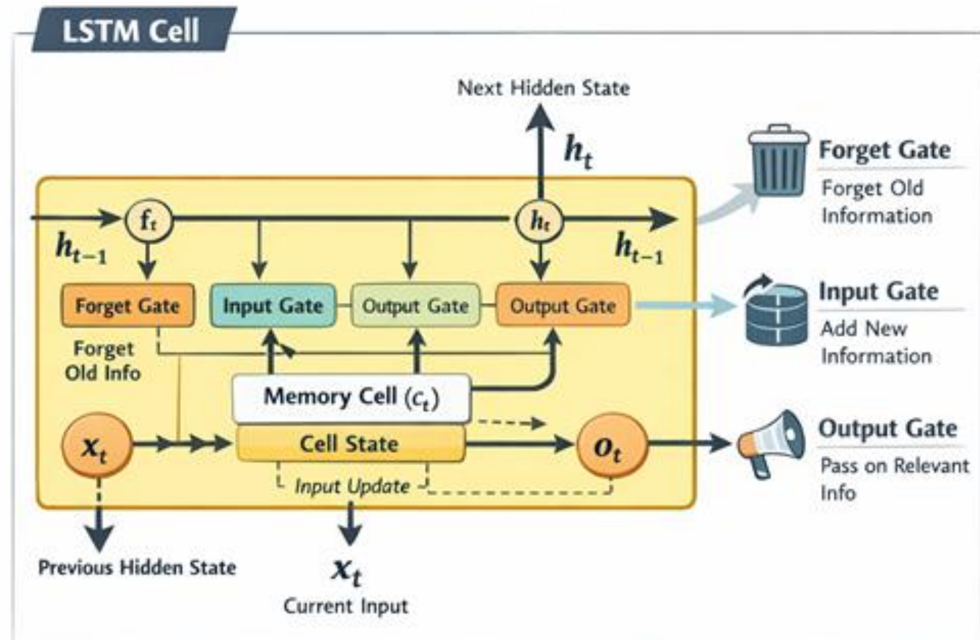
Long Short-Term Memory Networks

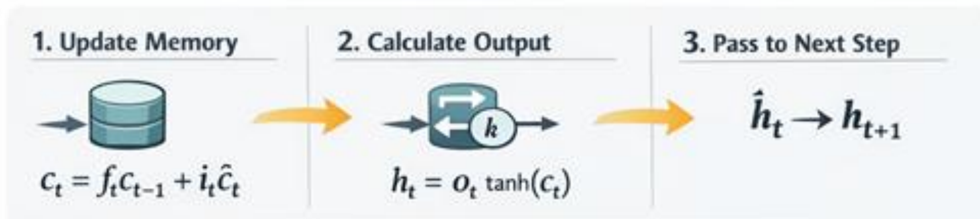
Long Short-Term Memory (LSTM) networks are a specialized type of recurrent neural network (RNN) designed to model sequential and time-dependent data while overcoming the limitations of traditional RNNs, particularly the vanishing and exploding gradient problems that make learning long-term dependencies difficult. An LSTM maintains a memory cell that stores information across time steps, allowing the network to remember important patterns from earlier in a sequence. This memory is regulated by a set of gating mechanisms—the input gate, forget gate, and output gate—which control what information is added to the memory, what information is removed, and what information is passed to the next hidden state. By selectively retaining or discarding information, LSTMs can capture relationships across long sequences, making them especially effective for problems involving ordered data such as time series forecasting, speech recognition, natural language processing, and user behavior analysis. In data science and machine learning workflows, LSTMs are commonly applied when the prediction at a given time depends on patterns that occurred many steps earlier in the sequence.

Long Short-Term Memory Networks

How LSTM Networks Work





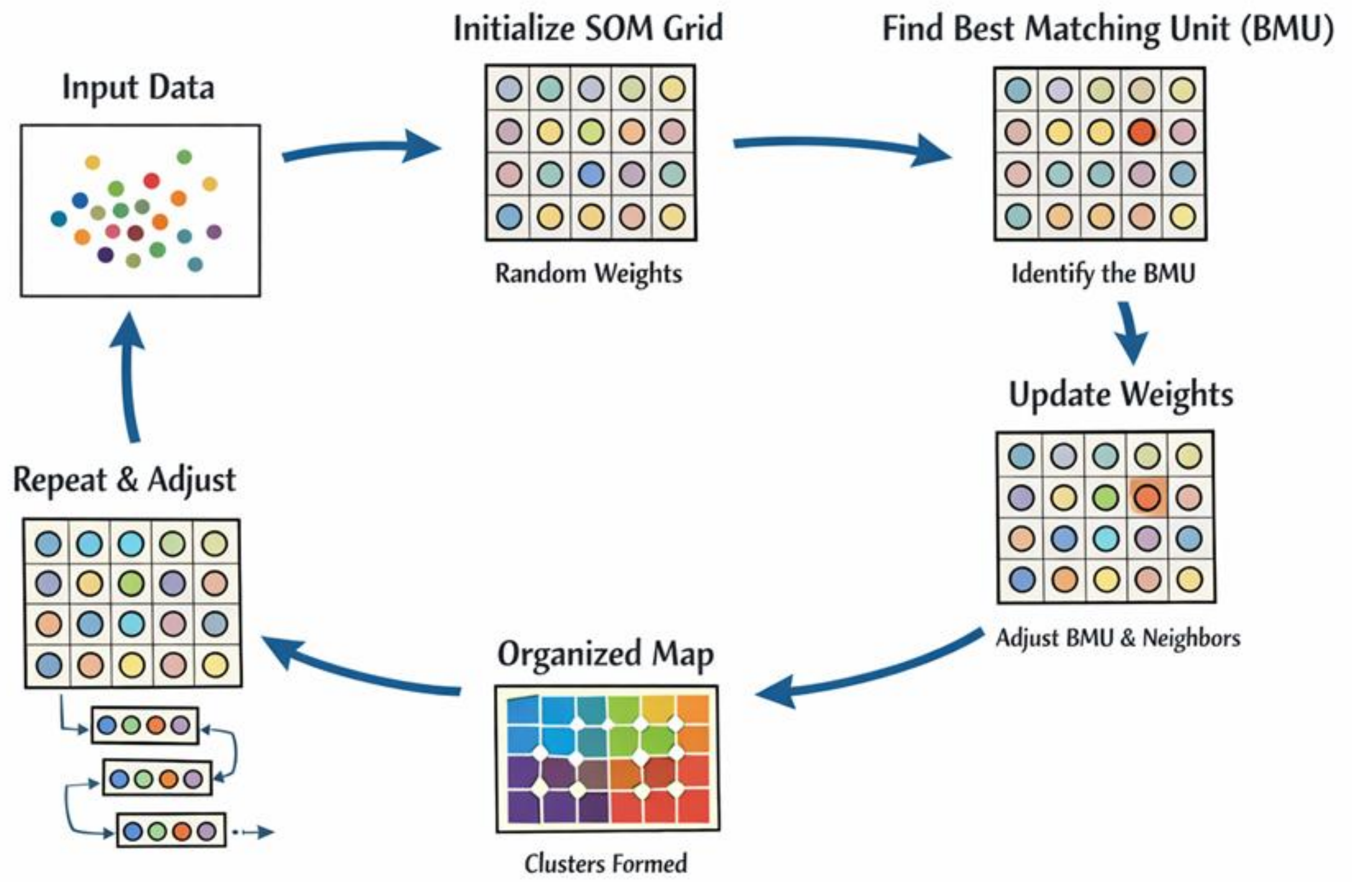


Capturing Long-Term Dependencies in Sequences

Self Organizing Map

A Self-Organizing Map (SOM), also known as a Kohonen map, is an unsupervised neural network algorithm used for dimensionality reduction and data visualization. It works by mapping high-dimensional input data onto a usually two-dimensional grid of neurons while preserving the topological relationships of the original data. During training, each neuron in the grid has a weight vector of the same dimension as the input data. When an input sample is presented, the neuron whose weight vector is most similar to the input—called the Best Matching Unit (BMU)—is identified. The BMU and its neighboring neurons then adjust their weights to become more like the input vector. Over many iterations, this competitive learning process causes the map to organize itself so that similar data points are placed close together on the grid. As a result, SOMs are commonly used for clustering, pattern recognition, and visualization of complex datasets, helping reveal hidden structures or relationships in the data without requiring labeled training examples.

Self Organizing Map



Self-Organizing Map Process