A tutorial on machine learning

Dwight Nwaigwe, Ph.D

Machine learning is a hot topic. Companies directly involved in it are among the most valuable, indicating its importance. As of this writing the top five valuable companies are all heavily involved in it (although this is not all they do) and are: Microsoft, NVIDIA, Apple, Amazon, and Alphabet with a combined market capitalization of around 12.2 trillion USD. The importance of machine learning's technological and economical impact has given it the moniker the fourth industrial revolution. Although it's not a silver bullet, at least yet, its impact and buzz make it worth discussing basic concepts and how it's used in practice.

1 Introduction

This tutorial is specifically on machine learning (ML), a subset of artificial intelligence. Machine learning deals with making machines learn from experience (i.e. data), whereas artificial intelligence is more general and deals with making machines have thinking abilities like humans in various domains. Artificial neural networks (ANN) have become the backbone of ML. They were first described by McCulloch and Pitts (1943) and the first implementation was the Stochastic Neural Analog Reinforcement Calculator built in 1951 under the supervision of Marvin Minsky and Dean Edmunds. Since then there have been several milestones which have made machine learning a practical and powerful tool. A small list is: the development of a technique called "backpropagation" (Werbos, 1974), (Rumelhart et al., 1986) which allows for the efficient training of ANNs; the development of specific ANNs (convolutional neural networks) which are great at recognizing images; deep Q-learning for reinforcement learning; the introduction of more advanced hardware which made it practical to do computations with ANN that contain a very large number of parameters (weights).

2 The three main categories of ML

ML can be seen as having three broad categories-statistical learning, deep learning, and reinforcement learning. As with most attempts of categorization, the boundaries are not always clear. This is especially true for deep learning and reinforcement learning as the former can be used in the latter.

2.1 Statistical machine learning

As the name implies, statistical machine learning is based upon statistics. A common reference book on the subject is Hastie et al. (2009). These methods work best on less complex data and less complex tasks. Some examples of statistical ML are principal component analysis ("PCA"); K-means clustering; logistic regression (see Nwaigwe and Rychlik (2025) for an interesting variation); and support vector machines (Cortes and Vapnik, 1995). In figure 1 the classification performance of PCA on the MNIST dataset (a collection of handwritten numbers) is shown. As more components are used the cross-validation accuracy increases but it stays below 95%. In contrast, ANNs can reach an accuracy $> 99\%$ on this dataset. Further, PCA does not do well on RGB images, an example of its relative weakness for complex data. K-means clustering is an example of *unsupervised* learning since the clustering algorithm is not told by the user which of the training data belongs to which clusters. In contrast, SVMs are an example of *supervised* learning since the algorithm must know which class a sample belongs to in order to start the training (fitting) process. SVMs are one of the best binary classifiers in statistical ML and for this reason we shall describe them in more detail. They were among the most popular machine learning tools between their introduction in 1995 until 2012, after which ANNs overtook them. The convolutional network introduced (commonly called "AlexNet") in Krizhevsky et al. (2012) is credited with renewing heavy interest in ANN. SVMs take a set of data points belonging to two classes and try to find an optimal hyperplane that separates these points. Often, points are not separable. To get around this, SVMs use the "kernel trick", a transformation which implicitly maps data points to a higher dimensional space, thereby increasing the likelihood of separability (figure 2). A drawback of support vector machines is that they are limited to handling two classes. To discriminate between three or more classes, multiple SVMs must be used.

Ensemble methods may also be considered a part of statistical learning. These methods use multiple classifiers to make a single stronger classifier. One example is a voting method where the output is given by the class which receives the most votes among the classifiers. Another ensemble algorithm is Adaboost (Freund and Schapire, 1997) which won the Gödel prize in 1997.

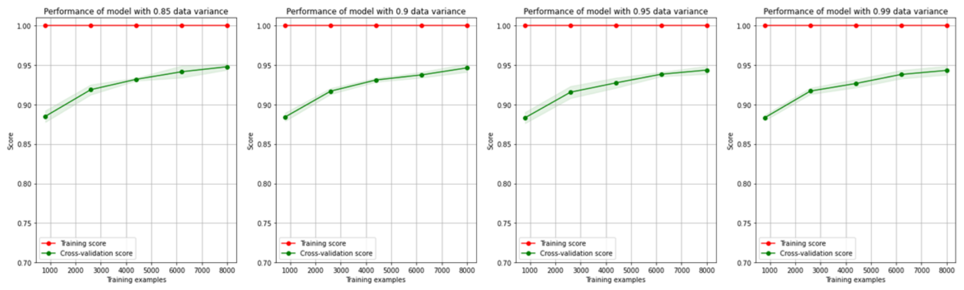


Figure 1: PCA achieves good performance on classifying MNIST data. However, it is still not as good as ANNs which reach > 99%. Nor is it effective on ImageNet which ANNs are.

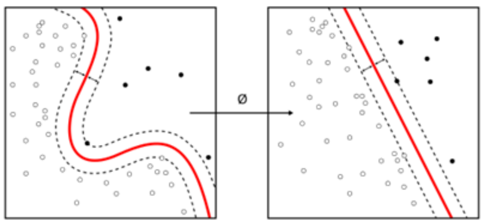


Figure 2: SVMs try to make nonseparable points separable through the mapping ϕ whose output is in a higher dimensional space. Then, a hyperplane (red) can be used as a discriminator. Commons (2023)

2.2 Deep learning

The second category of ML is deep learning which is based upon the use of ANNs. Typically there are many layers of neurons, hence the name “deep”. ANNs were originally based upon an attempt to model animal neurons. One major difference between ANN and biological neurons is that biological neuron activity is time dependent. This is shown in figure 3. For instance, after being activated there is a period in which a neuron can no longer fire (hyperpolarization). As mentioned before ANNs tend to do better than statistical ML methods in complex tasks. However, they are often criticized as needing lots of data and being difficult to interpret in comparison to statistical ML.

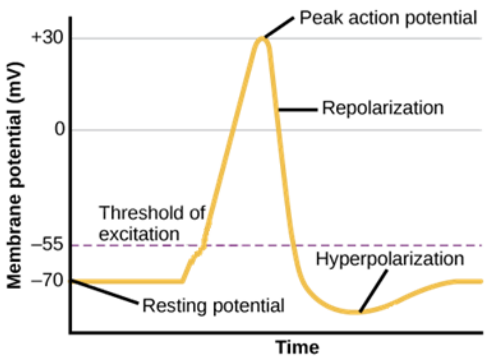


Figure 3: A real neuron's activations are a function of time unlike those of ANNs. Image modified from Clark et al. (2018).

We now talk more about how ANNs work. ANNs take an input and give an output. Let $\{(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})\}_{i=1}^{N}$ be a dataset that we would like to use to train an ANN. We denote by $\mathbf{y}^{(i)}$ the output from the ANN given the input $\mathbf{x}^{(i)}$. The output is a function of the input and the ANN's parameters and can be expressed as $\mathbf{y}^{(i)} = f(\mathbf{W}, \mathbf{x}^{(i)})$, where $\mathbf{W}$ represents the ANN's parameters. The symbol $\mathbf{W}$ may contain just one or billions of parameters. It can be considered as a matrix for ease of understanding but its shape is not important. The process of *training* an ANN is the act of optimizing (usually minimizing) a function of the form

$$L(W) = \sum_{i=1}^{N} C(y^{(i)}, t^{(i)}) = \sum_{i=1}^{N} C(f(W, x^{(i)}), t^{(i)}) \tag{1}$$

where C is some function. The function C usually penalizes how much $\mathbf{y}^{(i)}$ differs from $\mathbf{t}^{(i)}$. A common example is the case in which C outputs the squared error. Another common case is when L corresponds to a likelihood. In this case C corresponds to the *cross-entropy* between $\{\mathbf{y}^{(i)}\}_{i=1}^{N}$ and $\{\mathbf{t}^{(i)}\}_{i=1}^{N}$. In the ANN community, the function L is typically called a loss function. It has other names such as objective function, cost function, and risk function depending on the academic field. There are many possible ways to choose the function L and this is provided in software for ANNs such as Tensorflow, Pytorch, or JAX (see section 5 for more discussion on software and hardware) by specifying the name of the function you want to use. This optimization problem is usually non-convex which means that there are multiple minima and the software might not find the global minimum. In fact, it probably will not and this is an accepted fact in ML. To give an example training set, $\{(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})\}_{i=1}^{N}$ can be a set of pairs of the form (image, label). The quantity "label" can be a vector which is equal to 0 everywhere except the n^{th} entry, where n corresponds to the class label of the particular image. Lastly, we note that the examples given in this section are examples of *supervised* learning.

2.3 Reinforcement learning

The third broad category of ML is reinforcement learning. This topic deals with scenarios in which continuous feedback is injected into the learning process, hence the name *reinforcement*. The scenarios are thought of as a sequence of tuples whose elements are states, actions, and rewards. For instance, in a game of checkers, the state is the arrangement of pieces on the board and the possible actions are those which are allowed by the game, e.g. move to an empty space diagonally. Because different moves result in different levels of advantage, each state-action pairing can be associated with a different "reward". In this example the goal of reinforcement learning is to maximize the reward throughout the game. Or equivalently, learn the optimal actions against an adversary. The book by Sutton and Barto (2018) is one of the most common references for reinforcement learning but the author finds it unnecessarily verbose. A short and to-the-point set of notes can be found at Jeen (2021).

Typically, reinforcement learning is thought of as an interaction between an agent (or actor) and its environment. Figure 4 shows how the agent and environment interact with each other. More formally, at time t, the likelihood of taking action a_t given s_t is denoted $\pi(a_t|s_t)$, where π is the *policy*. The reward gained after taking action a_t in state s_t is given by r_t. The reward after T time steps is given by

$$\sum_{t=1}^{T} r_t. \tag{2}$$

In some cases T is very large or infinite, and to make the prior sum quickly (or converge at all), a discount factor γ is used, where $0 < \gamma < 1$. The sum then becomes

$$\sum_{t=1}^{T} \gamma r_t. \tag{3}$$

The goal is to maximize the expected values of expressions 2 or 3 for all possible starting states so that we can pick the best policy. Let the expected cumulative reward starting from state s be denoted by $v(s)$, and let the optimal $v(s)$ be denoted by $v_*(s)$. It can be shown that $v_*(s)$ must satisfy the Bellman optimality equation:

$$v_*(s) = max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{4}$$

where $\mathcal{A}$ is the set of possible actions, $p(s', r|s, a)$ is the probability of transitioning to state s' with reward r after being in state s and taking action a. There are a variety of algorithms to find $v_*(s)$. Once $v_*(s)$ is found then an optimal policy, π_*, is found by always choosing an action which results in the next state s' such that $v_*(s')$ is greatest. We note that there may be more than one optimal policy. A variation of equation (4) involves the estimation of *action-value* (or Q) functions, the expected reward when starting from a specified state-action pair. This is in contrast to value functions which average over all actions in a state. Among the most common ways to find the optimal policy is deep Q-learning (Mnih et al., 2015), an ANN-based approach. Deep Q-learning estimates the optimal action-value function hence its name. As mentioned before, deep learning and reinforcement learning are not mutually-exclusive, and deep Q-learning is an example. Deep Q-learning made big gains in advancing the ability of computers to play against humans in many games (Mnih et al., 2015). The successful use of ANNs in reinforcement learning shows the impact of ANNs not just in ML but in general.

Figure 5 shows an implementation of reinforcement learning taken from Weirich (2024). In this example, a grid is used to simulate roads and obstacles. The road is white, obstacles are red, gray areas are rough terrain. The truck is is penalized if it hits an obstacle or goes into rough terrain. The truck is given a reward if it does any of the following: picks up a load from the top right square (yellow) while it is unloaded; drops off a load at the bottom right square (purple) while loaded; makes an appropriate move in the direction of the load or unload sites. There is source code at Weirich (2024) for the readers' convenience. Upon running the code, the truck learns what it is supposed to. However, we caution that the code uses Keras RL, an older and somewhat obsolete software framework. Software is mentioned in more detail in section 5.

3 Some fundamental ANN architectures

There are three fundamental architectures that are used to make ANNs in practice. These are multi-layer perceptrons ("MLP"), recurrent neural

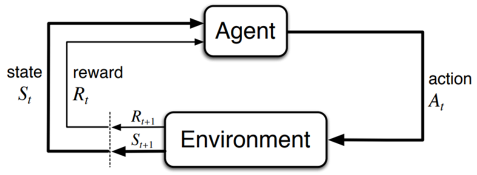
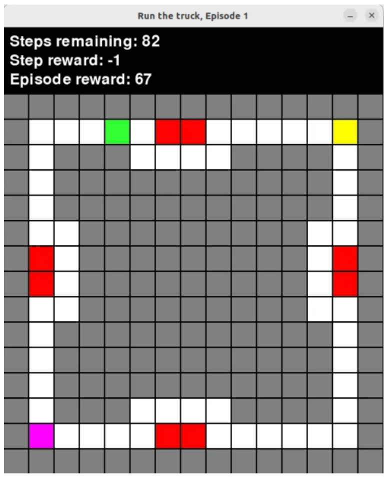


Figure 4: How an agent interacts with its environment. (Sutton and Barto, 2018).



Steps remaining: 82
Step reward: -1
Episode reward: 67

Figure 5: Reinforcement learning is used to train a truck (green square) to: follow white squares to the yellow square where it gets a load; unload it at the purple square; repeat. Red and gray scares are obstacles and rough terrain, respectively. (Weirich, 2024)

networks (RNN), and convolutional neural networks ("CNN"). A fourth, "transformer", is just a *feedforward* ANN that largely consists of MLPs but because of its significance we consider it separately. We will discuss each of these.

3.1 MLP

MLPs are the simpleset ANNs and go back to McCulloch and Pitts (1943). For ease of visualization, an MLP can be thought of as made of several layers, where each layer is a stack of neurons. The only connections that exist are between neurons from adjacent layers. Further, all connections move in one direction, i.e. from input layer towards output layer. In other words, information flows in one direction. An example MLP is shown in figure 6. MLPs are a case of feedforward ANN. It has been shown that an MLP consisting of one hidden layer can approximate any reasonable function (Hornik et al., 1989). The MLP can be described as a sequence of mathematical operations as follows. The value of hidden layer 1 is equal to

$$\mathbf{h}_1 = \sigma_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{5}$$

where $\mathbf{x}$ is the input, $\mathbf{W}_1$ some matrix, $\mathbf{b}_1$ is a bias vector, and σ_1 some nonzero function called the activation function. Likewise, the value of the second hidden layer is given by

$$\mathbf{h}_2 = \sigma_2(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2) \tag{6}$$

and so forth for the other intermediate layers. The final layer, the output layer, is of the same form. Even though figure 6 shows one unit for the output there can be as many outputs as desired. The number of outputs is equal to the number of classes in a classification problem.

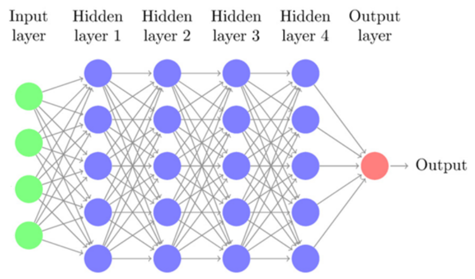


Figure 6: From Pérez-Enciso and Zingaretti (2019). The input can be thought of a vector that has been “flattened” into shape, say from a matrix. Although only one output is shown there can be more than one.

3.2 RNN

Recurrent neural networks are a type of ANN with a feedback loop. That is, current outputs are used to inform the next outputs which is unlike the case of MLPs. Transformers have made RNNs less important than what they used to be. A simple RNN is shown in figure 7. RNNs were difficult to use in practice until the introduction of long short term memory (“LSTM”) units (Hochreiter, 1991), (Hochreiter and Schmidhuber, 1997). LSTMs solved the “vanishing gradient” problem that made plain RNNs ineffective at learning dependencies on old data. These days, when one speaks of RNN, they usually mean LSTM or a gated recurrent unit (GRU) which is a simplification of an LSTM. Plain RNNs tend to not be used in practice. In figure 8 an LSTM unit is shown. Note that because the LSTM unit has an additional value associated with it, C_t, the diagram in figure 7 would need to be modified if the RNN cell is an LSTM cell. For completeness we list the equations for a version of an LSTM below. Note that these equations do not need to be implemented by practitioners since software takes care of this detail.

$$
\begin{aligned}
\mathbf{f}_t &= sigmoid(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f) &(7)\\
\mathbf{i}_t &= sigmoid(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) &(8)\\
\mathbf{i}_o &= sigmoid(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o) &(9)\\
\tilde{\mathbf{c}}_o &= hyperbolic\ tanh(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c) &(10)\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t &(11)\\
\mathbf{h}_t &= \mathbf{o}_t \odot hyperbolic\ tanh(\mathbf{c}_t) &(12)
\end{aligned}
$$

where $\mathbf{x}_t$ is a vector, $\mathbf{W}'s$ and $\mathbf{U}'s$ are matrices, $\mathbf{b}'s$ are vectors of the same size as $\mathbf{x}_t$, and $\odot$ is the Hadamard product.

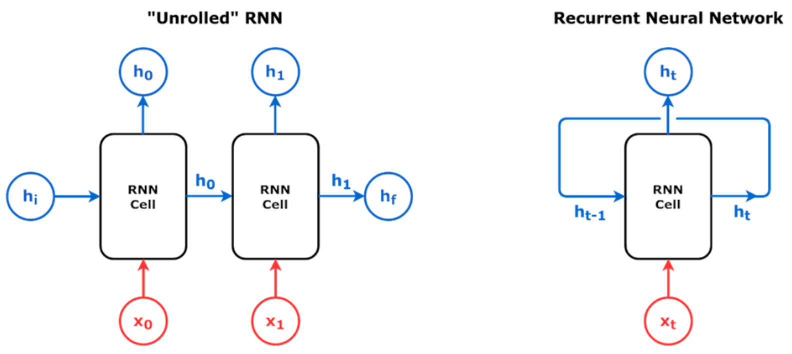


Figure 7: Two perspectives of a simple RNN (Godoy, 2021). Left: in “unrolled” or full form. Right: in compact or “rolled” form. The red circles are inputs. The blue circles are intermediate values which are used to compute a final value which is not shown. The RNN cell perform operations on the current input and previous intermediate output to produce a current intermediate output.

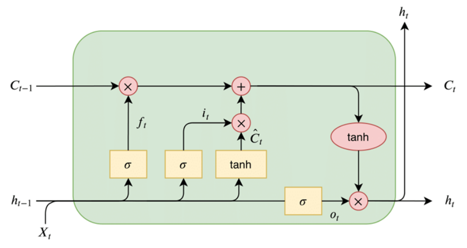


Figure 8: An LSTM cell.

3.3 CNN

The use of CNNs in ML has led to effective artificial vision. The Neocognition introduced in Fukushima (1980) is the basis for CNN. Fukushima was inspired by the neursoscience research works Hubel and Wiesel (1962), Hubel and Wiesel (1965) which show that neurons in cats' visual cortex respond differently to regions in the visual field. Figure 9 shows his interpretation of this fact. This fact inspired the use of the convolution operation thereby giving CNNs their name. In their simplest form CNNs are feedforward networks like MLPs, but the connections between layers have a different structure to reflect their sensitivity to specific neurons. Because of convolution or pool operations not all neurons from one layer connect to an adjacent layer, unlike in MLPs. The convolution operation is shown in figure 10 while the pooling operation is shown in figure 11. Pooling layers are often called downsampling layers and their job is to scale down an input while preserving information. The weights of the filter, for instance in figure 10, are determined during the training process.

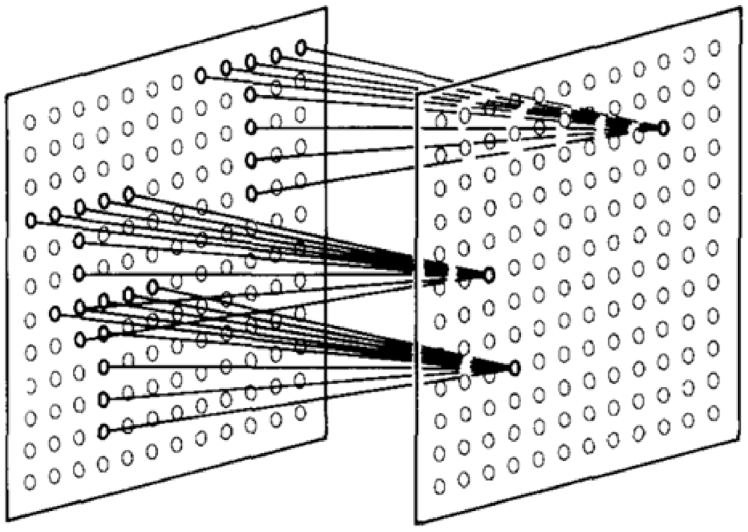


Figure 9: Image from Fukushima (1980) inspired by Hubel and Wiesel (1962) and Hubel and Wiesel (1965)

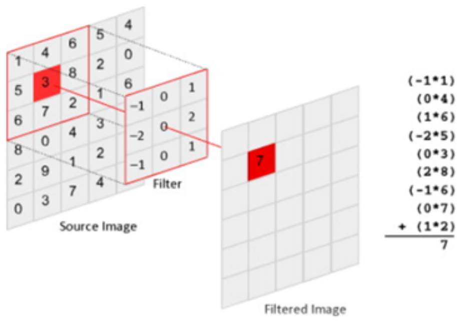


Figure 10: The convolution operation on a 2-dimensional image, Gaster et al. (2012). The filter is shown with its parameters determined for simplicity. In actuality the parameters are found during the training process.

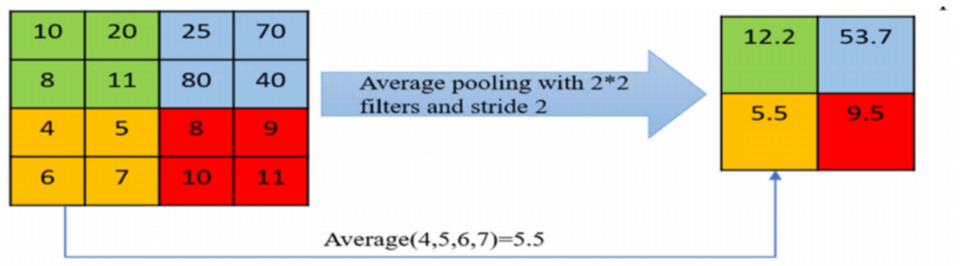


Figure 11: A pooling operation on a 2-dimensional image. Zafar et al. (2022)

3.4 Transformer

Transformers are a complex architecture based on feedforward ANNs and they are the main component of large language models like ChatGPT. Although they are just based upon feedforward ANNs and recursion we describe them in greater detail due to their success and dominance for complex tasks such as language translation, general LLM tasks, or sequence prediction. Transformers are the latest attempt of trying to solve the "sequence to sequence" ("Seq2Seq") problem. This is the problem of trying to map a variable length sequence to another, for instance translating a sentence between two languages. Before the advent of transformers this was done using *attention* with RNN (Bahdanau et al., 2015). Transformers removed the RNN part, thereby allowing processing to be done in parallel as opposed to sequentially. Figure 12 shows a transformer. As shown in figure 12, sequences are first embedded into vectors whose entries are real values. For instance, assume there is a vocabulary denoted V of size $|V|$. Then each element of V is embedded into $\mathbb{R}^{|V|}$ by assigning to it an element of the standard basis. In machine-learning terms this is called *one-hot* encoding. To be more concrete, if $V = \{$"*hound*", "*dog*", "*jaguar*"$\}$, then an example of a one-hot encoding can be hound $\mapsto (1, 0, 0)$, dog $\mapsto (0, 1, 0)$, jaguar $\mapsto (0, 0, 1)$. Secondly, a matrix of dimensions d by $|V|$ multiplies these embedded vectors to create an embedding into $\mathbb{R}^d$, where d is some chosen dimension. The resulting sequence of embeddings is then given to an *attention* block. Attention is the heart of transformers, shown also in figure 12. Attention is a method of determining relationships between elements of a sequence. The idea is that given the current element of a sequence, we would like to know how it "attends" or relates to prior elements of the sequence. Attention is used in predicting the next element of a sequence by weighing the relative importance of prior elements of the sequence. To help demystify transformers it is instructive to list the operations they perform:

$$\mathbf{q}_i = \mathbf{W^Q}\mathbf{x}_i \tag{13}$$
$$\mathbf{k}_j = \mathbf{W^K}\mathbf{x}_j \tag{14}$$
$$\mathbf{v}_j = \mathbf{W^V}\mathbf{x}_i \tag{15}$$
$$score(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{16}$$
$$\alpha_{ij} = softmax(score(\mathbf{x}_i, \mathbf{x}_j)) \quad j \leq i \tag{17}$$
$$\mathbf{head}_i = \sum_{j\leq i} \alpha_{ij}\mathbf{v}_j \tag{18}$$
$$\mathbf{a}_i = \mathbf{W^O}\ \mathbf{head}_i \tag{19}$$
$$\tag{20}$$

where $\{\mathbf{x}_i\}$ are elements of the embedded sequence previously mentioned, $\mathbf{W^Q}$ and $\mathbf{W^K}$ are d_k by d matrices, $\mathbf{W^V}$ is a d_v by d matrix, $\mathbf{W^Q}$ is a d by d_v matrix, and $softmax$ is the function that takes some vector $\mathbf{z}$ and outputs a vector whose i^{th} entries are $\frac{e^{\mathbf{z}_i}}{\sum_j e^{\mathbf{z}_j}}$. Note that $softmax$ produces a probability distribution, i.e. the sum over all elements of $softmax(\mathbf{z})$ is equal to 1 and each component is greater than or equal to zero. There is considerable leeway in picking the constants d, d_k, d_v. In Vaswani et al. (2017), these values are respectively equal to 512, 64, and 64. Also note that the output of $\mathbf{a}_i$ is a vector of size d just as the starting embedding vector. This is important because since the goal of attention is sequence prediction there must be consistency between its input and output dimensions.

We just described single-head attention where as in figure 12 multi-head attention is used. Multi-head attention is single-head attention in parallel and conceptually it suffices to just explain single-head attention. Other components of the transformer architecture are a simple feed forward network (i.e. MLP) and final linear and softmax operations. The final linear operation is another matrix multplication to decode the inputs from $\mathbb{R}^d$ to $\mathbb{R}^{|V|}$. The final softmax gives a probability distribution over V, i.e. it is the likelihood of outputting a particular element from V. There are some variations of attention but the description provided is used in the original work (Vaswani et al., 2017).

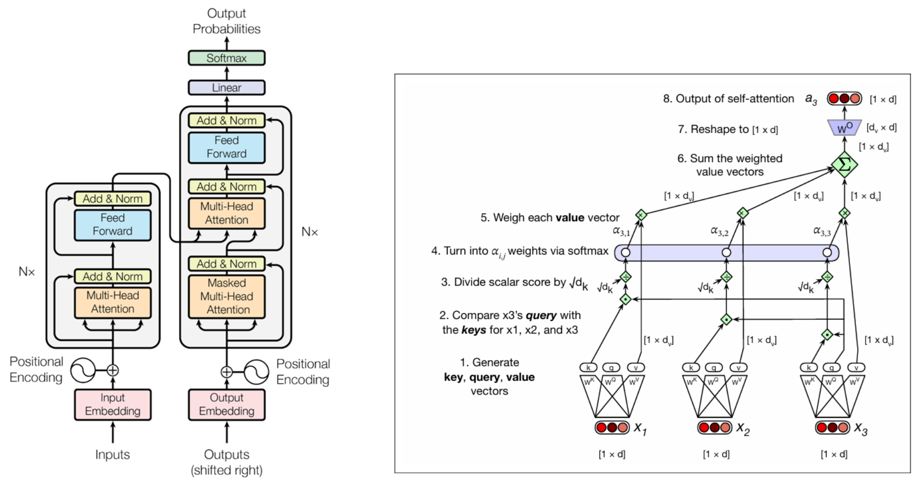


Figure 12: Left: the transformer as introduced in Vaswani et al. (2017). Right: pictorial representation of attention from Jurafsky and Martin (2024)

4 Training ANNs

ML software takes care of the mathematics and other stuff under the hood. The only things the practitioner needs to do is have data, code the model (or use a model that is already preloaded), choose a loss function (one or two lines of code), and choose an optimizer (one or two lines of code). There can be some differences in model accuracy that come from the optimizer used, but one should not expect much difference (Schmidt et al., 2021).

Most optimizers used with ANN are *first-order methods*, meaning that only the first-order derivatives are used. The simplest example is plain gradient descent given by

$$\mathbf{W}^{i+1} = \mathbf{W}^{i} - \eta \nabla L(\mathbf{W}^{i}) \tag{21}$$

where η is a pre-determined constant called the learning rate. The bigger the learning rate the bigger the jumps between the successive iterates of equation (21). Higher-order methods require higher-order derivatives and are costly to compute. Higher-order methods are not necessarily better than first-order methods, it is situation-dependent. Practitioners usually have the option to choose from a variety of optimizers including ones that use momentum and adaptivity. Adaptive methods speed convergence near a point where the gradient is zero by increasing the learning rate. A particularly popular one is ADAM. In practice, an optimizer does not operate on the full dataset at once as that would require too much memory. Instead, only part of the data is used at each optimization pass. One way to do this is to partition the training data into batches and sequentially apply the optimizer to the batches- an implicit assumption is that the loss function can be decomposed as a sum where each term corresponds to a batch of training data. This is almost always the case. Another alternative is to randomly subsample from the dataset at each optimization pass, i.e. stochastic gradient descent. In either case, this is easily done by software.

One thing practitioners can do to improve accuracy is to use data augmentations. Data augmentations are a way of increasing the size of your dataset. The dataset is copied and the new set is modified with operations that the user deems valid, i.e. the modified image still represents a sample that can be drawn from the set of all possible samples. For instance, rotations of an image could be one valid augmentation. Another possible augmentation can be reflections or shear. The copied dataset is then combined with the original dataset to make the training dataset.

Another thing that can be done to improve accuracy and/or training speed is transfer learning. This is the use of all or some of the weights of a pre-trained ANN for training on a new dataset. For instance, if one wants to use a state-of-the-art ANN for image recognition like ResNet for their own images, then transfer learning would mean getting a copy of a ResNet that is already trained on ImageNet, and then either: 1) freeze all weights except those of the last few fully-connected layers and then train ResNet on your data 2) train on your data with all weights unfrozen. These two options are in contrast to training from scratch in which one would train with a ResNet architecture that was never trained on. This would take longer and/or result in less accuracy.

Another consideration is architecture. Getting an accurate ANN is largely an art as it is not clear from the beginning what type of architecture would work the best. There is an area of research called *neural architecture search* devoted to this problem (Elsken et al., 2019), White et al. (2023). One may make the mistake of insisting on a well-known state-of-the art classifier that has hundreds of millions of parameters instead of experimenting with simpler custom models. For certain datasets, simpler custom models may provide more accuracy and will almost certainly be quicker to train and evaluate. The author is aware that shallow ANNs of just one or two layers are used in a particular financial company for fraud detection.

5 Hardware and software

Software for ANNs include Tensorflow (released by Google Brain in 2015), Pytorch (released by Facebook in 2016), and JAX (released by Google in 2020), all written in the Python programming language and free and open-source. Tensorflow has waned in popularity over the years. Figure 13 show this trend. JAX is a newer edition to the ANN toolset. Although it's growing, its community is smaller than those of Tensorflow or Pytorch. Of note, there is a popular library called Keras which is a wrapper around the three frameworks (versions ≥ 3.0 are for all frameworks while older verions are only for Tensorflow) which makes them substantially easier to use. Lastly, software for statistical ML can be found in various Python packages such as Sci-kit learn. Often these packages are pre-installed with Python unlike Tensorflow, Pytorch, Keras, and JAX. It is worth noting that MATLAB has a deep learning toolbox which follows MATLAB's easy-to-use interface and is an easy way to become involved in ML. The author cautions however that for heavier or more custom work it is not advisable to use MATLAB due to the resource-intensive nature of the application and the greater difficulty to customize compared to the other frameworks. These deep-learning frameworks also can be used for reinforcement learning. An API that is commonly used with deep-learning frameworks for reinforcement learning is OpenAI Gymnasium. It contains many standardized environments for benchmarking.

Deep-learning frameworks come with many state-of-the-art models preloaded and they can be used in just a few lines of code. Other models which are not preloaded can be downloaded from Huggingface or GitHub.

The majority of serious work in industry and research is done using high performance computing including graphical processing units which can speed training and inference times by an order of magnitude. Many of the operations in ANNs (i.e. matrix multiplication) can be implemented in parallel which makes using graphical processing units advantageous. Although this and other optimizations are done in ANN frameworks practitioners must be sure that they have proper hardware resources to take advantage of it, e.g. access to a graphics-processing unit. Google has developed a more specialized version of a graphical processing unit called the tensor processing unit ("TPU") to better address the computational demands of ANNs. It is only interoperable with Tensorflow though. Given figure 13, the author wonders how much longer TPUs will come with Tensorflow. Maybe they will come with JAX in the future.

The reader should not get the impression that one cannot do deep learning on a personal computer- it absolutely can be done, but it would be much more difficult than using higher-end hardware when dealing with large datasets and models with a large number of parameters.

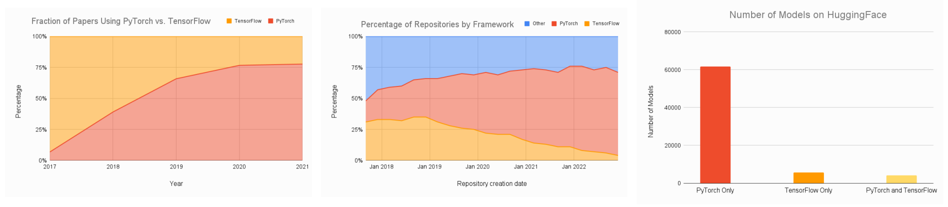


Figure 13: Popularity of Pytorch versus Tensorflow over time. O'Connor (2021)

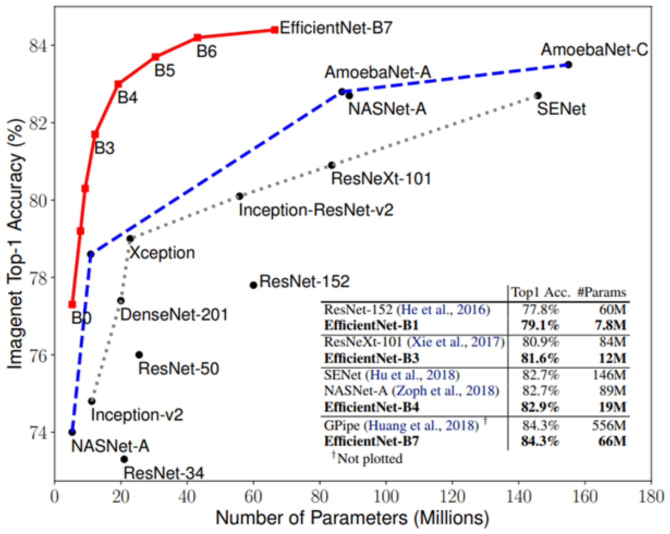
6 Applications

We now discuss some applications of ML. There are many but we will focus on computer vision and generative AI.

6.1 Computer vision

Computer vision is a pervasive use of ML. In some cities there are red-light cameras perched above intersections that take photographs of license plates of cars that run red lights. The license plates are searched in a database, the offender tracked and sent a ticket. Another example are smartphones which are able detect faces in their cameras field of view. Users can see this when a bounding box appears around their head in the smartphone's screen. Computer vision is based on relatively deep (many layers) CNNs. State-of-the-art CNNs for computer vision have parameters up to several hundred million. However there are some popular and practical scaled-down deep CNNs such as EfficientNet (Tan and Le, 2019) or MobileNet (Howard et al., 2017). In particular, MobileNet being designed for smartphones only has a few million parameters. See figures 14 and 15 which show numbers of parameters vs classifier accuracy. Small hardware like smartphones are limited in their resources and so ML software is usually lightened on these devices. For instance, to run ANN-based applications smartphones use Tensorflow Lite which is made for model inference only.

One weakness of computer vision are adversarial images. In practice, adversarial images are not a concern but we discuss them here for more completeness. An adversarial image is a small modification to an image



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

†Not plotted

Figure 14: Accuracy vs number of parameters for various well-known CNNs. Tan and Le (2019)

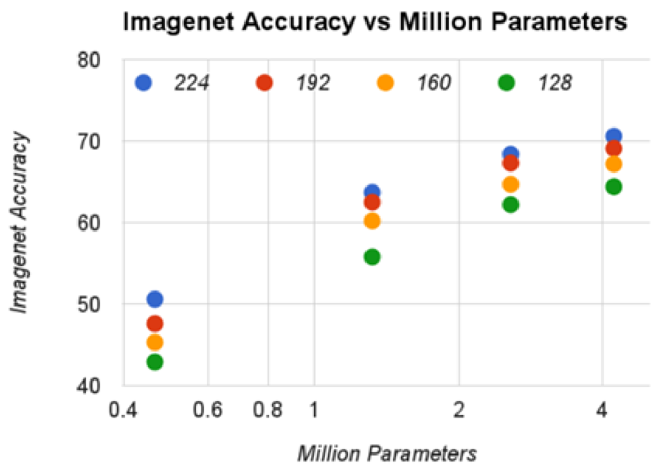


Figure 15: Accuracy vs number of parameters for Mobilenet. The colors represent input image resolution. Howard et al. (2017)

which to humans seems minor or imperceptible but which causes a classifier to misclassify the image (Goodfellow et al., 2015). In figure 16 an example is shown in which an ANN goes from believing (correctly) an image is probably a panda to believing (incorrectly) that the slightly changed image is probably a gibbon. There have been many studies at detecting adversarial attacks, among them Nwaigwe et al. (2024).

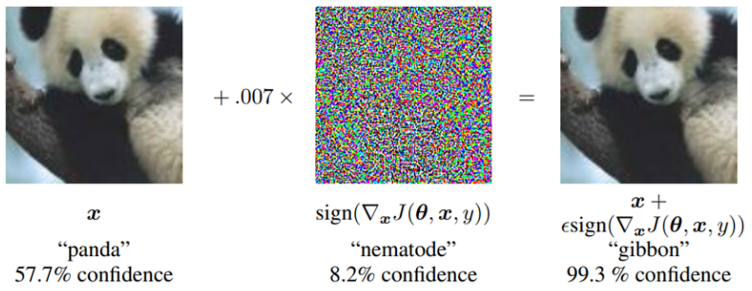

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 16: Example of an adversarial image, Goodfellow et al. (2015)

Another instance of computer vision is the use of machines at the US Post office to read the writing on letters. This has been in practice since 1965. The practice of reading writing has historically been called optical character recognition, although that term is now somewhat dated. A more modern (or at least accurate) term is image-to-text recognition. The state-of-the-art in this field (and in the sister field of speech transcription) are ANN-based approaches including connectionist temporal classification "CTC" Graves et al. (2006), transfomer-based approaches, and ANN-hidden Markov model hybrids. The previous state-of-the-art technology for reading writing or transcribing speech was based purely on Hidden Markov models, a statistical approach. In Figure 17 an example image-to-text recognition engine is shown. It is a pure ANN approach and relies on a CNN, attention (see the earlier discussion on transformers), and RNNs. A different approach, but still ANN-based, is shown in figure 18. In this appraoch, an RNN operates on a sample image ("that") by divding the images into rectangles and giving it to a bi-directional RNN. The RNN is trained using CTC (which predates transformers), which outputs labels for each rectangle. The labels are then collapsed into a word using a simple rule which states repeating labels are merged and hyphens (character transistions) are removed. Readers interested in learning about the the state-of-the art should consult AlKendi et al. (2024).

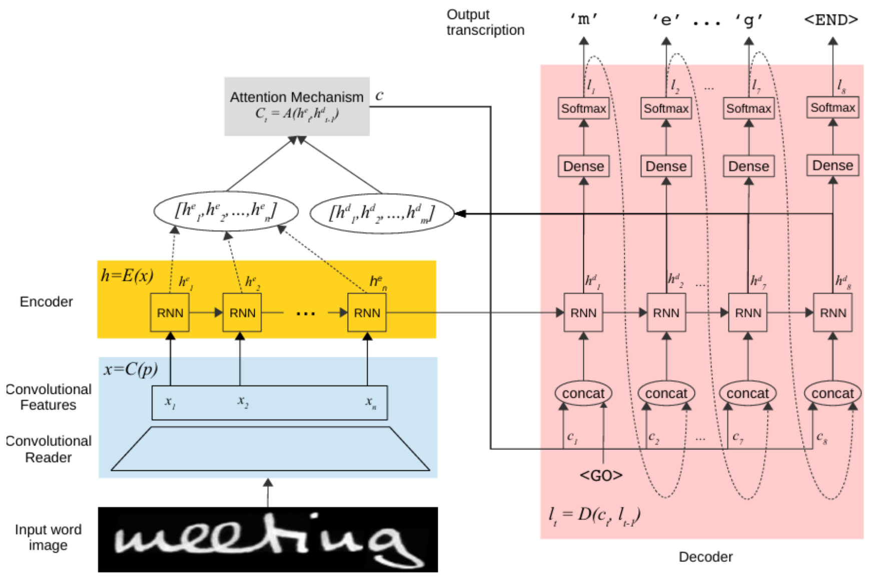
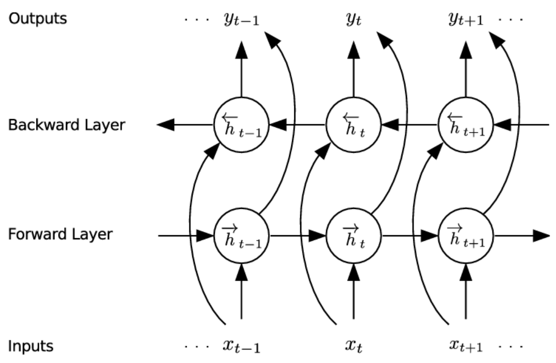


Figure 17: An example image-to-text recognition system from Revuelta (2021). It uses a CNN, attention, and RNN.



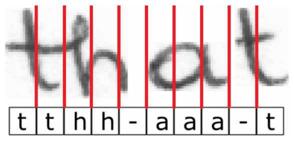


Figure 18: Left: A bi-directional RNN (Graves and Jaitly, 2014) used for image-to-text recognition. Right: An input image is divided into rectangles, Each rectangle corresponds to some x_t from the bi-directional RNN shown left. Under each rectangle are the outputs y_t from the bi-directional RNN. Image modified from Revuelta (2021).

6.2 Generative AI

Generative AI refers to AI that makes things- typically images and text. This is the hottest topic in AI and typically means image and text generation. Examples of technologies used for text generation are large language models and small language models (LLM and SLM). The technology behind these are transformers. Examples of LLMs are Chat-GPT (by OpenAI), LLAMA (by Meta), and Claude (by Anthropic). LLMs have multiple uses- they can summarize text, translate text, generate text, and engage in interact answer-response sessions (chatbot). Lots of money has been poured into this area, in part because of a widespread belief that LLMs are a path to artificial general intelligence (a belief the author does not share). Figure 19 shows how transformers are used for word prediction. At each time step the output of the transformer is combined with the previous outputs to create a new input sequence to the transformer. This new input sequence is used to predict the next output and the process repeats. The training process of LLMs can be involved and often incorporates reinforcement learning (Lambert et al., 2022).

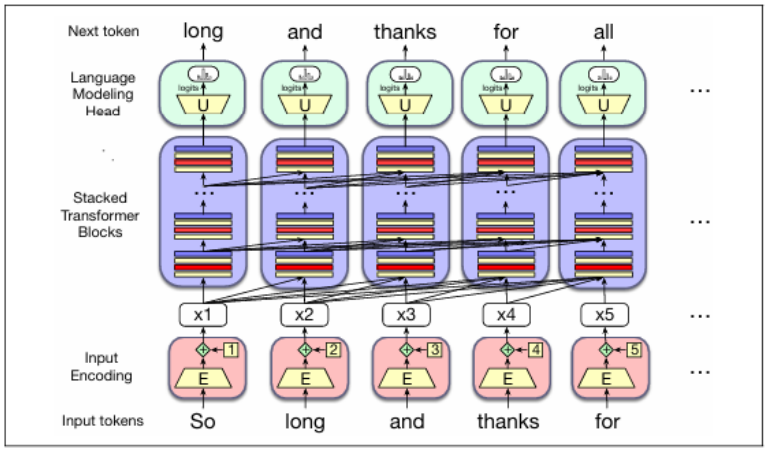


Figure 19: Using a transformer for sequence (word) prediction (Jurafsky and Martin, 2024).

There are four ways to make realistic artificial images- diffusions(Ho et al. (2020), Sohl-Dickstein et al. (2015)), generative adversarial networks (Goodfellow et al., 2014), flows, and variational autoencoders. The first two methods are by far the most common and so we will focus on them. All four methods are based on ANNs. Figure 20 gives an overview of these different methods. What all four methods have in common is that they sample a random vector from a pre-defined distribution, and this random vector is transformed to produce an artificial image.

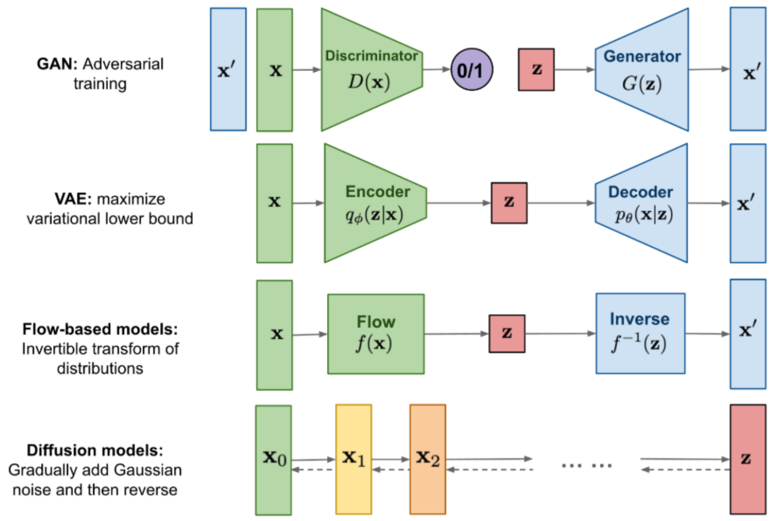

GAN: Adversarial training
$\mathbf{x}'$
$\mathbf{x}$
Discriminator $D(\mathbf{x})$
0/1
$\mathbf{z}$
Generator $G(\mathbf{z})$
$\mathbf{x}'$
VAE: maximize variational lower bound
$\mathbf{x}$
Encoder $q_\phi(\mathbf{z}|\mathbf{x})$
$\mathbf{z}$
Decoder $p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$
Flow-based models: Invertible transform of distributions
$\mathbf{x}$
Flow $f(\mathbf{x})$
$\mathbf{z}$
Inverse $f^{-1}(\mathbf{z})$
$\mathbf{x}'$
Diffusion models: Gradually add Gaussian noise and then reverse
$\mathbf{x}_0$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{z}$

Figure 20: The four principal ways to create artificial images (Weng, 2021).

Generative autoencoders are based upon two ANNs playing a competitive game, one trying to generate a fake but convincing image while the other tries to discriminate the image from a real one. The discriminator is an ANN (an MLP in the original formulation) that is a function of two arguments- ANN parameters and an image. The generator (also an MLP in the original formulation) takes two arguments, parameters and a vector which is traditionally termed "noise". The noise vector is a sample from a predetermined distribution. The discriminator outputs 1 if it believes an image is real and 0 otherwise. The optimization is given by

$$min_{\mathbf{W}_G}\ max_{\mathbf{W}_D}\ (\mathbb{E}_{\mathbf{X}}\ log(D(\mathbf{W}_D,\mathbf{X})) + \mathbb{E}_{\mathbf{Z}}\ log(1 - D(\mathbf{W}_D, G(\mathbf{W}_G,\mathbf{Z})))) \tag{22}$$

where D is the discriminator, G is the generator, $\mathbf{W}_D$ are the discriminator's parameters, $\mathbf{W}_G$ are the generator's parameters, $\mathbf{X}$ is the set of training images and $\mathbf{Z}$ is the set of sampled noise vectors. The first term of equation (22) seeks to make D correctly classify real images. The second term is more difficult to interpret but it is clear that it has two competing objectives- influence D to correctly identify false images and influence G to fool D. After the training procedure is complete, $\mathbf{Z}$ can be sampled and given to G

to produce artificial images. The reader who is interested in a mathematical analysis of GANs should refer to Wang (2020).

The idea behind a diffusion model is to convert a known distribution into a target distribution by a sequence of transformations, and then to learn how to undo this process. Diffusion models attempt to learn this process so that they can sample from the target distribution and map it to a sample from the starting distribution, i.e. generate a realistic image from a random vector. These ideas are shown in figures 21 and 22. In the first figure, the top row shows how a sample image is transformed into a Gaussian distribution over time. The second row shows how a sample from the Gaussian distribution is transformed back into what is supposed to be the sample space. Let the sequence of random variables $(\mathbf{x}_t)_{t=0}^{t=T}$ represent the data under the forward process at each time step. The forward process is described by the distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$. It gives the distribution of data at time t given $\mathbf{x}_{t-1}$. Specifically,

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) \sim \mathcal{N}(\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})|_{\mathbf{x}_t} \tag{23}$$

where β_t is chosen. The reverse process is given by

$$p(\mathbf{x}_{t-1}|\mathbf{x}_t) \sim \mathcal{N}(\mu_\theta(\mathbf{x}_t, t), \mathbf{\Sigma}_\theta(\mathbf{x}_t, t))|_{\mathbf{x}_{t-1}} \tag{24}$$

where μ_θ and $\mathbf{\Sigma}_\theta$ are to be learned by the ANN (θ represents the ANN's parameters) Figure 22 shows how p and q act in opposite directions.

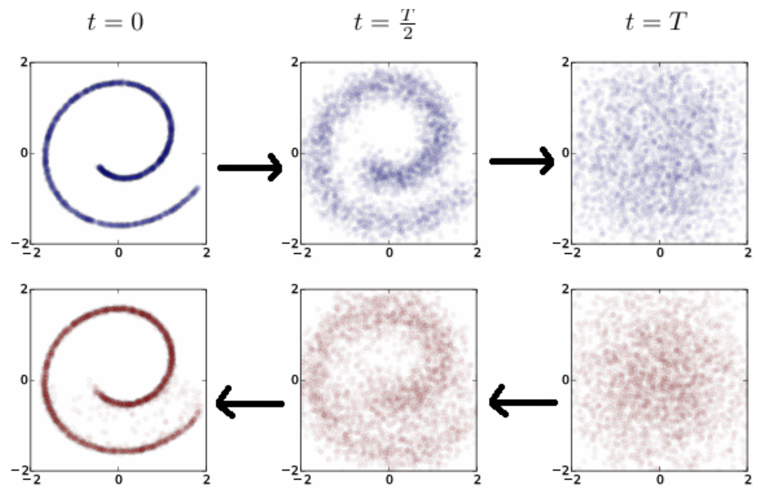


Figure 21: The top row show the forward process and the bottom row shows the reverse process. Image adapted from Sohl-Dickstein et al. (2015)

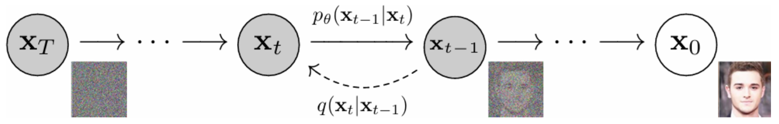

$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_{t-1}$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{x}_0$

Figure 22: Image from Ho et al. (2020).

Denote $q(\mathbf{x}_0)$ as the starting distribution of the data. Similarly define $p(\mathbf{x}_T)$ as the starting point on the reverse trajectory. The ending distribution given by the reverse process is given by

$$p(\mathbf{x}_0) = \int d\mathbf{x}_T d\mathbf{x}_{T-1} ... d\mathbf{x}_0 p(\mathbf{x}_T) \prod_{t=1}^{t=T} p(\mathbf{x}_{t-1}|\mathbf{x}_t). \tag{25}$$

We would like $p(\mathbf{x}_0)$ to be equal to our starting distribution of data, $q(\mathbf{x}_0)$. One way to do this is to maximize the quantity

$$L = \int d\mathbf{x}_0 log(p(\mathbf{x}_0))^{q(\mathbf{x}_0)}. \tag{26}$$

Optimizing L directly is difficult, so instead a lower bound K is found which is optimized. Through tedious manipulations it can be shown that $L > K$ where

$$K = \sum_{t=2}^{T} d\mathbf{x}_0 d\mathbf{x}_t q(\mathbf{x}_0, \mathbf{x}_t) D_{KL}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)||p(\mathbf{x}_{t-1}|\mathbf{x}_t)) \tag{27}$$

$$+ H_q(\mathbf{x}_t|\mathbf{x}_0) - H_q(\mathbf{x}_1|\mathbf{x}_0) - H_p(\mathbf{x}_t), \tag{28}$$

D_{KL} is Kullback-Leibler divergence, and H_q and H_p are entropies taken with respect to the distributions $p(\cdot|\cdot)$ and $q(\cdot|\cdot)$. For details, the reader should consult the aforemtentioned references.

References

W. AlKendi, F. Gechter, L. Heyberger, and C. Guyeux. Advancements and challenges in handwritten text recognition: A comprehensive survey. *J. Imaging*, 10(1), Jan. 2024.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/abs/1409.0473.

M. A. Clark, J. Choi, and M. Douglas. *Biology 2e*. OpenStax, Apr. 2018.

W. Commons, 2023. URL `https://commons.wikimedia.org/wiki/File:Kernel_Machine.svg`.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20 (3):273–297, Sept. 1995. ISSN 1573-0565. doi: 10.1007/bf00994018. URL `http://dx.doi.org/10.1007/BF00994018`.

T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL `http://jmlr.org/papers/v20/18-598.html`.

Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1504. URL `https://www.sciencedirect.com/science/article/pii/S002200009791504X`.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr. 1980. ISSN 1432-0770. doi: 10.1007/bf00344251. URL `http://dx.doi.org/10.1007/BF00344251`.

B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. Chapter 4 - basic opencl examples. In B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, editors, *Heterogeneous Computing with OpenCL*, pages 67–85. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-387766-6. doi: https://doi.org/10.1016/B978-0-12-387766-6.00027-X. URL `https://www.sciencedirect.com/science/article/pii/B978012387766600027X`.

D. V. Godoy, 2021. URL `https://github.com.dvgodoy/dl-visuals/`. [Accessed 25-05-2025].

I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. 2015.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL `https://proceedings.neurips.cc/paper_files/paper/2014/file/f033ed80deb0234979a61f95710dbe25-Paper.pdf`.

A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1764–1772, Bejing, China, 22–24 Jun 2014. PMLR.

A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning - ICML '06*, ICML '06, page 369–376. ACM Press, 2006. doi: 10.1145/1143844.1143891. URL `http://dx.doi.org/10.1145/1143844.1143891`.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer New York, 2009. ISBN 9780387848587. doi: 10.1007/978-0-387-84858-7. URL `http://dx.doi.org/10.1007/978-0-387-84858-7`.

J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851. Curran Associates, Inc., 2020.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis*, (5), 1991.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997. ISSN 1530-888X. doi: 10.1162/neco.1997.9.8.1735. URL `http://dx.doi.org/10.1162/neco.1997.9.8.1735`.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, Jan. 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8. URL `http://dx.doi.org/10.1016/0893-6080(89)90020-8`.

A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. URL `https://arxiv.org/abs/1704.04861`.

D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of*

Physiology, 160(1):106–154, Jan. 1962. ISSN 1469-7793. doi: 10.1113/jphysiol.1962.sp006837. URL `http://dx.doi.org/10.1113/jphysiol.1962.sp006837`.

D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat. *Journal of Neurophysiology*, 28(2):229–289, Mar. 1965. ISSN 1522-1598. doi: 10.1152/jn.1965.28.2.229. URL `http://dx.doi.org/10.1152/jn.1965.28.2.229`.

S. Jeen. In *Notes on Reinforcement Learning:An Introduction(2ndEdition)*, 2021. URL `https://enjeeneer.io/sutton_and_barto/rl_notes.pdf`.

D. Jurafsky and J. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 2024.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

N. Lambert, L. Castricato, L. Von Werra, and A. Havrilla. Illustrating reinforcement learning from human feedback (rlhf). `https://huggingface.co/blog/rlhf`, 2022. [Accessed 28-07-2025].

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, Dec. 1943. ISSN 0007-4985, 1522-9602. doi: 10.1007/BF02478259. URL `http://link.springer.com/10.1007/BF02478259`.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL `http://dx.doi.org/10.1038/nature14236`.

D. Nwaigwe and M. Rychlik. The existence of the maximum likelihood estimate in multinomial logistic regression for mixed-membership models. *Communications in Statistics - Theory and Methods*, page 1–11, June

2025. ISSN 1532-415X. doi: 10.1080/03610926.2025.2505986. URL `http://dx.doi.org/10.1080/03610926.2025.2505986`.

D. Nwaigwe, L. Carboni, M. Mermillod, S. Achard, and M. Dojat. Graph-based methods coupled with specific distributional distances for adversarial attack detection. *Neural Networks*, 169:11–19, Jan. 2024. ISSN 0893-6080. doi: 10.1016/j.neunet.2023.10.007. URL `http://dx.doi.org/10.1016/j.neunet.2023.10.007`.

R. O'Connor, 2021. URL `https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/`. [Accessed 25-05-2025].

M. Pérez-Enciso and L. M. Zingaretti. A guide on deep learning for complex trait genomic prediction. *Genes*, 10(7):553, July 2019. ISSN 2073-4425. doi: 10.3390/genes10070553. URL `http://dx.doi.org/10.3390/genes10070553`.

J. J. S. Revuelta. *Continuous Onine Handwriting Recognition using Deep Learning Models*. PhD thesis, Universidad Rey Juan Carlos, 2021.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct. 1986. ISSN 1476-4687. doi: 10.1038/323533a0. URL `http://dx.doi.org/10.1038/323533a0`.

R. M. Schmidt, F. Schneider, and P. Hennig. Descending through a crowded valley - benchmarking deep learning optimizers. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9367–9376. PMLR, 18–24 Jul 2021. URL `https://proceedings.mlr.press/v139/schmidt21a.html`.

J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2256–2265, Lille, France, 07–09 Jul 2015. PMLR.

R. S. Sutton and A. Barto. *Reinforcement learning: an introduction*. The MIT Press, second edition edition, 2018. ISBN 9780262352703.

M. Tan and Q. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/tan19a.html.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Y. Wang. A mathematical introduction to generative adversarial nets (gan), 2020. URL https://arxiv.org/abs/2009.00169.

A. Weirich. Reinforcement learning with tensorflow, keras-rl and gym. https://medium.com/@alfred.weirich/experiments-with-reinforcement-learning-cff75b7d783c/, 2024. [Accessed 28-05-2025].

L. Weng. What are diffusion models? https://lilianweng.github.io/posts/2021-07-11-diffusion-models//, July 2021.

P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Phd dissertation, Harvard University, 1974.

C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter. Neural architecture search: Insights from 1000 papers, 2023. URL https://arxiv.org/abs/2301.08727.

A. Zafar, M. Aamir, N. Mohd Nawi, A. Arshad, S. Riaz, A. Alruban, A. K. Dutta, and S. Almotairi. A comparison of pooling methods for convolutional neural networks. *Applied Sciences*, 12(17), 2022. ISSN 2076-3417. doi: 10.3390/app12178643. URL https://www.mdpi.com/2076-3417/12/17/8643.