Introduction to Deep Learning

Reza Rezazadegan
www.dreamintelligent.com

This article is a fast introduction to deep learning and sets the reader on the path to learning the subject in more detail.

Supervised Machine Learning

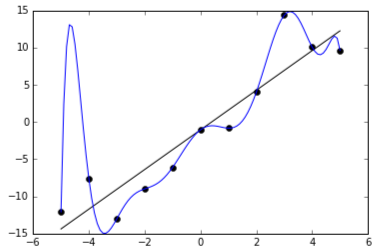
Supervised learning is a method of training a machine by giving it examples instead of explicitly programming it. In supervised learning we are given samples of labeled data $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ and we want to find a function $F(x)$ which maps the inputs $x^{(i)}$ to the *labels* $y^{(i)}$. The input $x = (x_1, x_2, \ldots, x_d)$ is represented by a vector, and its components x_j are called the *features* of the data. Examples include:

- x being a bitmap image and y the names of the persons or the objects in it. A $w \times h$ image with c color channels (usually 1 for monochrome or 3 for RGB (Red, Green, Blue) images), has $c \times w \times h$ features.
- x an image and y its caption.
- x a medical image (MRI, mammograph, X-ray, CAT Scan, etc.) and y the diagnosis, such as cancer, benign tumor, fracture, etc.
- x a piece of text and y its sentiments such as positive, negative, neutral, angry, etc.
- x time, and y the price of a stock or a foreign currency. The y can be any other time-dependent quantity such as temperature, blood sugar, or the monthly sales of a commodity.
- x a piece of text (e.g. an article) and y its topic (e.g. politics, technology, sports).
- x a piece of text and y its category, such as spam or not spam.
- x a piece of text in one language, and y its translation into another language.
- x a voice recording and y its text transcription (Speech-to-Text)

The function $F(x)$ that we find must have two fundamental properties:

- $F(x^{(i)})$ must be "close" to $y^{(i)}$ for "most" $x^{(i)} \in D$. This is formalized using a *loss function* L; discussed below.
- $F(x)$ must generalize well to the x that do not belong to D. *Generalization* is a central notion in learning.

The method for ensuring the second criterion is to use a fraction (say, 70%) of D (called *training set*) for finding F and using the rest (called *test set*) for testing the generalization capability of F.



A model F that does not generalize well, such as the blue curve in the above image, is said to *overfit* the data.

If the labels $y^{(i)}$ take values in a discrete set then the problem is called a *classification* problem. Otherwise it is called a *regression* problem. For classification, the model can be *probabilistic*, giving us the probabilities for each label i.e. $F(x) = (p_1, p_2, \ldots, p_m)$ where m is the number of labels and $\sum p_i = 1$. Note that the labels themselves can be represented by m-dimensional vectors whose entries are 1 at position i and zero elsewhere. Such a vector can be thought of as a concentrated probability distribution.

Data Representation

Note that computers can only understand numbers; individual numbers or vectors of numbers. Thus, all forms of data, such as text, images, audio and video has to be turned into numbers before can be used in machine learning. This means finding a function Φ, called an *embedding* or *representation*, that maps each datapoint into a vector in the Euclidean space $\mathbb{R}^d$ for some d. In the case of textual data, this means assigning to each word w in the vocabulary, a d-dimensional vector $\Phi(w)$. This embedding must be so that the words which have similar meanings are mapped into nearby vectors in the Euclidean space.
For some types of data, this is easy. For example, a digital image is represented by a 2D array (matrix) of pixel values. For color images, each pixel has 3 values corresponding to the strength of the red, green and blue colors at the pixel.

Sentences and texts can be converted to vectors too. The simplest method for finding such a Φ for a bunch (corpus) of documents, is to form the vocabulary V of the corpus then mapping each document T to the vector in $\mathbb{R}^d$ such that d is the size of the vocabulary and the k'th entry of $\Phi(T)$ is the number of the occurrences of the the k'th word of the vocabulary in T. More advanced method (such as word2vec and BERT) use Neural Networks for obtaining word and text representations.

Parametrized models; learning as optimization

The general method for solving a supervised learning problem in machine learning is to choose a family of functions $F(x, w)$ where w is a vector, representing the model *parameters* or *weights*. Such a model gives us a family of functions parametrized by w. The simplest such model is given by *linear regression* in which

$$F(x, w) = \langle x, w \rangle + b = \sum x_i w_i + b.$$

The black line in the above image is an example of linear regression.

Another example is given by *polynomial regression* of degree k in which all degree k combinations of the features are used; for example for $k = 2$ we have $F(x, w) = \sum_{i,j} W_{i,j}x_i x_j$+b. There are many other parametrized models used in Machine Learning such as Decision Trees, Support Vector Machines, etc.

In Neural Networks and Deep Learning we use combinations of **artificial neurons** (ANs), also called **Perceptrons** to approximate functions. An artificial neuron is given by

$$F(x, w) = h(\langle x, w\rangle + b)$$

where h is a fixed nonlinear function called the *activation function.*

After choosing a parametrized family, we want to know which element of the family, or in other words, which value of the parameter vector w gives us the best approximation of our data D. For this purpose, we first need to use a *loss function* (or *cost function* or *error*) which quantifies how far the prediction $\hat{y}$ is from the actual label y. For continuous labels, $C(y, \hat{y})$ can be simply $(y - \hat{y})^2$, called Square Error. For a probabilistic classification model, we can use *Cross Entropy*. If $\hat{p}$ is the real and predicted vectors of label probabilities and $\hat{p}_{correct}$ is the probability that our model predicts for the correct label then,

$$C(\hat{p}) = -\log(\hat{p}_{correct}).$$

(As mentioned above, the real label can be represented by a probability vector p whose entry is 1 at the correct label and zero otherwise. Then $C(p, \hat{p}) = -\log(\langle p, \hat{p}\rangle)$ i.e. negative the logarithm of the predicted probability of the correct class.

No matter what the choice of the loss function is, we take the average of the losses over the training set and this gives us the loss as a function of the parameters:

$$C(w) = \frac{1}{N}\sum_{i}^{N} C(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{N}\sum_{i}^{N} C(y^{(i)}, F(x^{(i)}, w))$$

Note that this cost function is a function of the parameter set w and quantifies the *fitness* of the model function $F(w, x)$ for approximating our data. Of course we want to find a w that minimizes the cost function.

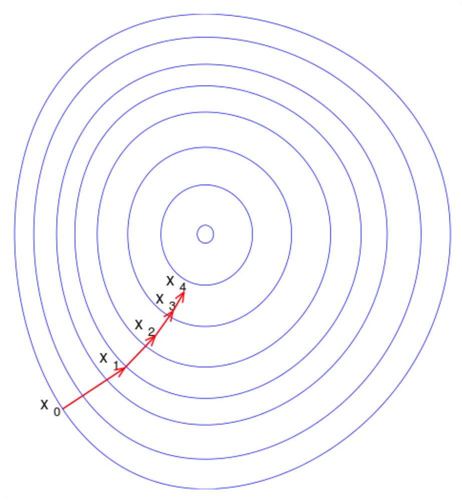
This way, supervised learning problems are converted to optimization problems in the weight space (also called hypothesis space) of the model. In the simple cases, such as linear regression, this problem can be solved explicitly by differentiation. In more complicated cases we may use a search algorithm such as *Gradient Decent* or the *Greedy Algorithm*.

Optimization by Gradient Decent

If the cost function C is smooth, we know that if we move in the direction of negative gradient $-\nabla_w C$, the value of C decreases. Thus, to find the minimum of C, we can use the Gradient Decent Algorithm:

$$w_{i+1} = w_i - \eta\nabla_w C(w_i).$$

The η is a hyperparameter called *learning rate*. The initial w_0 is chosen randomly.



Gradient Decent. The gradient of a smooth function is always perpendicular to its level sets. Image credit: Wikipedia

If the loss function is convex (as in linear regression), then it has only one minimum. However, for Neural Networks, the loss function is not convex and the search algorithm may end up in a local minimum.

The history of artificial neural networks and Deep Learning

As mentioned above, artificial neural networks use a parametrized model which is inspired by the working of neurons in the brain.

Brain Neurons

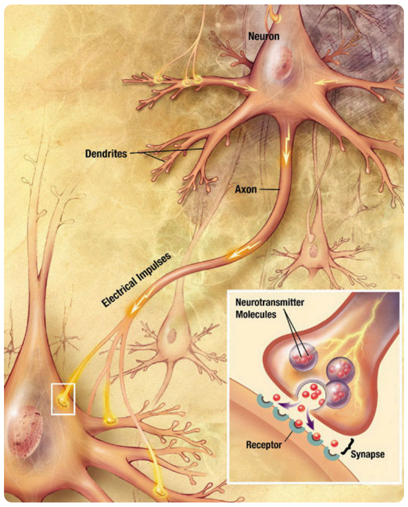


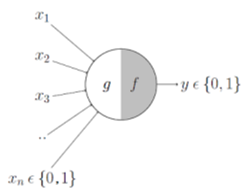
Image credit: Wikipedia

Human brain has some 86 billion neurons while the worm *Caenorhabditis elegans* has only 302 neurons. A neuron has several *dendrites* which carry electrical signals into the neuron and a single *axon* which carries signal into other neurons. *Synapses* are areas between the axon of one neuron and a dendrite of another, in which in which signals are passed by means of electricity or a neurotransmitter chemical. Each neuron has synaptic connections with up to 10^4 other neurons. There are about 10^{14} synapses in the brain. If the electrical signal received by a neuron changes by a large amount in a short time, an electrical pulse is transmitted through it axon.

Hebbian learning: neurons that fire together, wire together i.e. the strength of synaptic connections between them gets stronger. This way a *neural network* is formed which is believed to be the way new facts and skills are learned. This is also a special case of *neural plasticity* i.e. the ability of the nervous system to modify itself, structurally and functionally.

McCulloch and Pitts neurons

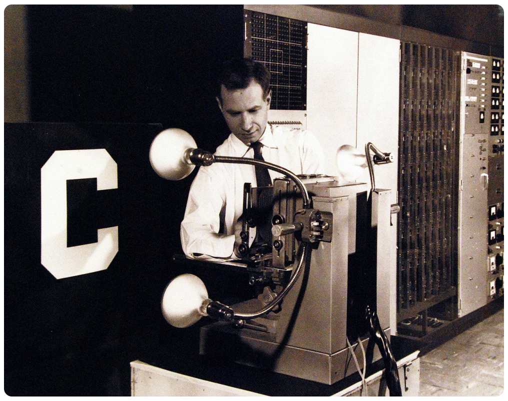
The first mathematical model of a biological neuron was developed by McCulloch and Pitts in 1943. It models the dendrites as binary inputs. The first part g of this *artificial neuron* aggregates the input values and the second part, f, makes a decision, , as whether the output should be 0 or 1. This decision is made according to a threshold parameter θ. If the value of the aggregate g is greater than θ then the output value is one. You can see that the OR and AND boolean functions are given by parameter values $\theta = 1$ and $\theta = n$ respectively.



The McCulloch-Pitts neuron can only represent boolean functions which can be represented linearly, i.e. if there is a hyperplane (of the form $x_1 + x_2 + \ldots + x_n = \theta$) which separates inputs that produce 0 from those which produce 1.

Perceptron

Perceptron was the first implementation of an artificial neuron, developed in 1958 by Frank Rosenblatt at Cornell University. It is simply given by $F(x, w) = sign(w_1x_1 + w_2x_2 + \ldots + w_nx_n)$ where both the input and the weights are boolean. Perceptron was implemented as a machine (hardware). Although perceptron initially caused a lot of excitement, it was soon showed that perceptron cannot represent many forms of functions. As a result, the interest in Perceptron died.



Homework 1.1: Show that the XOR function cannot be presented by a perceptron.

Multi-Layer Perceptron (MLP)

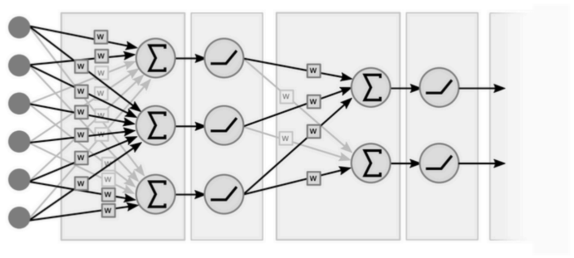
Neural networks reemerged in 1985 with the introduction of backpropagation (see below). The new neurons, unlike McCulloch-Pitts neurons, used continuous weights. Before 1980s multiplying floating point numbers on a computer was very slow.

An MLP consists of multiple layers of perceptrons. Each layer consists of a number of artificial neurons. The first layer, called the *input layer* has as many neurons as the number of dimensions (features) of the data. the output of each neuron is connected to the input of all the neurons in the next layer. A layer satisfying this property is called *fully connected*.

This means that if the layer j has n inputs, corresponding to n input features $\mathbf{z}^{(j)} = (z_1^{(j)}, \dots, z_n^{(j)})$, and m outputs, corresponding to m output features $\mathbf{z}^{(j+1)} = (z_1^{(j)}, \dots, z_m^{(j)})$, then

$$\mathbf{z}^{j+1} = h_j(W^{(j)}\mathbf{z}^{(j)} + b_j)$$

where $W^{(j)}$ is a $n \times m$ matrix of weights (model parameters which are learned during training) and h_j is a nonlinear *activation function*. In the early days of ANNs, the sign function or the Heaviside step function were used as activation functions. However they were later replaced with the sigmoid function and the positive part function (which is called *ReLU* in this context.)



Credit: Yann LeCun, Deep Learning

The term b_j is called the *bias* (not to be confused with the bias of a model, as in the bias-variance tradeoff) and can be omitted from the notation if we add an extra feature which is constant and equal to 1.

The last layer of an MLP is called the *output layer* and its size (the number of neurons in it) equals the number of classes (for classification) or the number of output features (for regression).

The layers other than the input and output, are called *hidden layers*. An MLP is called *deep* if it has more than one hidden layer.

Homework 1.2. Explicitly write the function represented by an MLP with 2-d input, 1-d output and 2 hidden layers of size 2, with ReLU activation.

MLP as a Universal Function Approximator

The Universal Approximation Theorem states that MLPs of large enough width or depth are universal approximators.

More precisely, let σ be a continuous non-polynomial function from $\mathbb{R}$ to $\mathbb{R}$ then for every compact $K \subset \mathbb{R}^n$, continuous function $f : K \to \mathbb{R}^m$ and $\epsilon > 0$, there are matrices $A_{k\times n}, C_{m\times k}$ for some k and a vector $b \in \mathbb{R}^k$ such that the function $g = C(\sigma \circ (Ax + b))$ is closer than ϵ to f on each point of K.

Note that g is in fact a neural network with one hidden layer of "large enough" dimension k. This number k is called the *width* of the network.

The second version of the UAT states that any integrable function from $\mathbb{R}^n \to \mathbb{R}^m$ can be arbitrarily approximated (in the L^p norm) by a MLP with ReLU activation, with width equal to $\max(n+1, m)$ and large enough depth.

There is also a version of the theorem with bounds on both width and depth of the network: any continuous function of d variables can be approximated by a network of depth=3 and width=$2d+2$. When $d = 1$, depth can be taken to be 2.

Note that even though any function can be approximated by a 2-layer network, there are reasons for using more than two layers and the reason being that the size of the hidden layer in a 2-layer network may have to be very large.

Backpropagation

So far we know what the structure of multi-layer perceptron is and that almost any function can be approximated by an MLP. But given a function (represented by labeled data), how do we go about finding the MLP that approximates it? The training of a neural networks involves finding the values of the parameters $W^{(1)}, W^{(2)}, \ldots, W^{(n)}$ in such a way that minimizes the loss. The loss function for such a network is the same as the ones mentioned in the section on supervised learning. The loss function for a neural network is nonconvex and has a complicated fitness landscape. For this reason, the method of Stochastic Gradient Decent is for training.

Once the number of layers and the size of each one is chosen, the optimal values of the parameters are found using the method of Gradient Decent. But, how should we adjust the value of the parameters in each iteration, based on the error?

The Gradient Decent is given by the equation

$$W_{i+1} = W_i - \lambda \nabla_W C(W_i),$$

However in neural networks we have a hierarchy of weights $W_i^{(1)}, W_i^{(2)}, \ldots W_i^{(n)}$ corresponding to different layers of the network (here we have n layers). Therefore we have

$$W_{i+1}^{(j)} = W_i^{(j)} - \lambda \nabla_{W^{(j)}} C(W_i)$$

Computing $\nabla_{W^{(j)}} C$ is based on the chain rule and it is performed from the last layer of the network back to the input layer. Hence the name *backpropagation*. Intuitively, we propagate the error from the output layer back towards the input layer and adjust the weights accordingly.

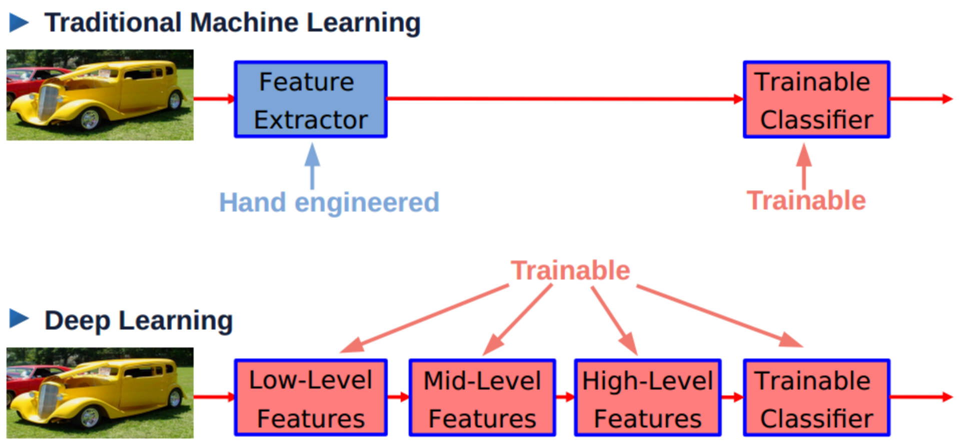
In 1989 LeCun used Convolutional Neural Networks (discussed below) with backpropagation for handwritten digit classification. However interest in NNs died in mid 90s because of the introduction of the Support Vector Machine, Ensemble Learning, etc.

The Deep Learning revolution

The real success of Deep Learning started in 2010 when it was used for speech recognition in smartphones. In 2012-13 the same happened in computer vision and in 2016 for NLP, especially for machine translation. In recent years it has produced marvels such as image generation models (such as Dalle) and and Large Language Models such as ChatGPT, Gemini, DeepSeek, etc., and is continuing its growth at a rapid pace.

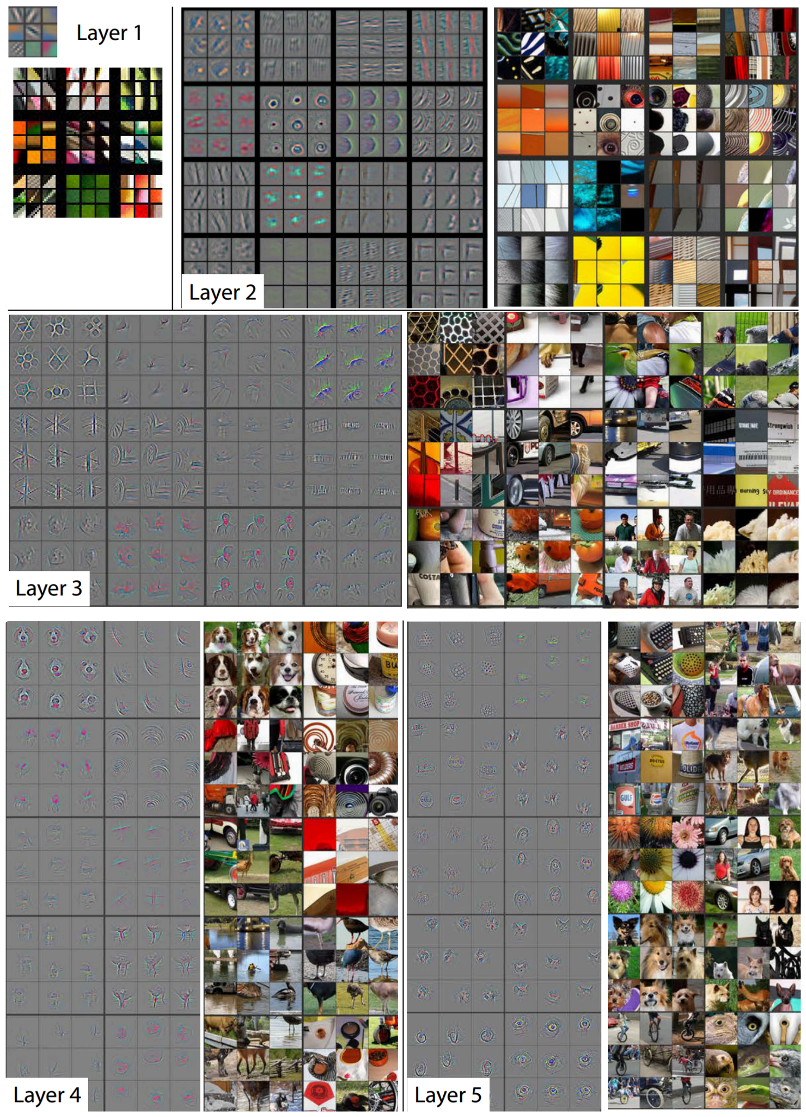
Images generated by Dalle from text prompts

Deep Learning can be defined as learning a hierarchy of features from data in such a way that at each stage ("layer"), new features are learned from the features of the last stage.



Credit: Yann LeCun, Deep Learning

Thus, deep learning is really about learning representations of data. The representations of data in the hidden layers of a neural network are called *latent representations*.



From Zeiler and Fergus, Visualizing and Understanding Convolutional Networks. In a neural network trained for classification, the data (features) in each layer are projected back to the "pixel space" (input layer). This is what you see on the left hand side and the right hand columns show the corresponding image segments. For each layer, the top samples that cause the highest activations are shown.

The success of Deep Learning is dependent on the *compositionality* of the world i.e. we have subatomic particles then atoms then molecules then polymers then compounds then parts of objects then objects then systems of objects (like a car or a building) then cities then countries then planets, etc.

- Images: pixels -> edges -> multi-edge shapes -> motifs -> parts of image
- Text: letters -> words -> sentences -> paragraphs -> articles -> books -> corpora
- Speech: samples -> bands -> sounds -> phones -> phonemes -> whole words -> sentences

Deep Learning Libraries

In the early days of Deep Learning, neural networks and backpropagation had to be coded from scratch. However we now have several great Deep Learning libraries, of which we mention a few.

Theano

The grandfather of DL libraries, is superseded by newer competitors such as Tensorflow and Pytorch, however its development continues to this date.

TensorFlow and Keras

Developed by Google and ideal for simple, plug-and-play implementation of neural networks. It is used more in business and engineering applications.

PyTorch

Developed by Meta, it has a low level API and it is used more in research. However there are multiple high level APIs built on top of PyTorch such as Fast.ai, HuggingFace and Pytorch-Lightning which "decouples research from engineering".

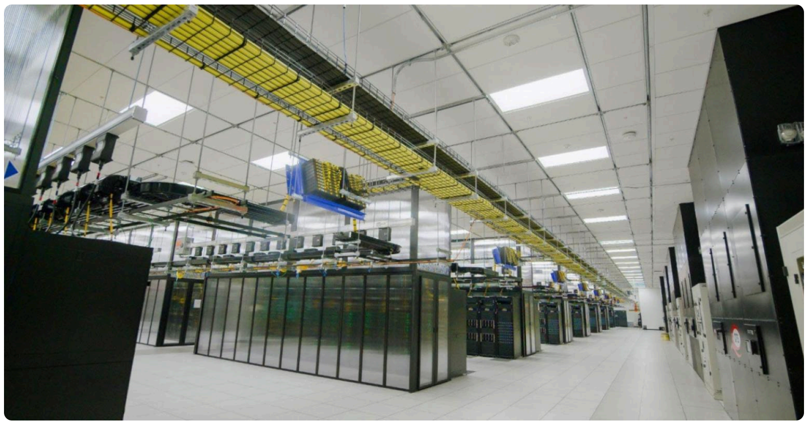
MXNET

Backed by Amazon and not as famous as the other ones, it likely has an edge in time series forecasting (GluonTS module).

Graphical Processing Units (GPUs) in Deep Learning

As mentioned before, the forward pass in training an MLP involves computing $F(x_i, w)$ for the x_i in our dataset. Parallel computing can greatly increase the speed of such computation and GPUs can do parallel computations much faster than CPUs. Therefore training large neural networks is usually done on GPUs, often arrays of multiple GPUs. The first time Neural Networks were trained on GPU, in 2010 by a student of Hinton.

The GPUs used in Deep Learning are almost exclusively made by NVIDIA and Deep Learning libraries make use of NVIDIA's CUDA library for parallel computation on the GPU. In fact NVIDIA has had a great impact on the progress of Deep Learning. Supercomputers used by Meta, Google and other FANG companies are comprised of hundreds of NVIDIA GPUs.


Meta's supercomputer with 16000 NVIDIA GPUs.

If you set up your own computer for Deep Learning, CUDA version and Pytorch version you use must be compatible with each other. However you can use cloud GPU services such as the free Google Colab, which rid you of the need for setting up these libraries.

Types of Neural Networks

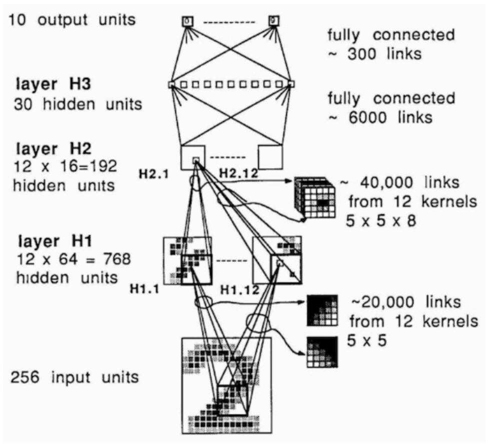
So far we have only discussed the MLP which is a fully connected and feed-forward neural network. However there are many different architectures of neural networks for use with different types of data.

Convolutional Neural Networks (CNNs)

In a fully connected network, the features (and their corresponding neurons in the input layer) are assumed to be uncoupled from each other. In other words, the data is assumed to be a flat vector of features, e.g. age, income, gender, etc. However, for other types of data such as images or sequential data (e.g. time series or text) this is not the case and we need other types of NNs. To feed an image (which is a 2 or 3 dimensional matrix or grid) into an MLP, we have to flatten it into a 1 dimensional vector. This approach has two problems: first, the proximity information of pixels is lost and then, a pixel in one layer will be connected to *all* the pixels in the next layer, even the ones which are far away in the image, and this is superfluous.
Instead we consider a $w \times h$ window around each pixel and connect each input neuron (pixel) $x_{i,j}$ only to the $w \times h$ neurons around it, in the next layer. Mathematically, instead of $y = Wx$, from one layer to the next, we have

$$y_{k,l} = \sum_{i=1}^{w} \sum_{j=1}^{h} W_{i,j} \cdot x_{i+k,j+l}.$$

Note that this operation reduces the dimension (resolution) of the image. This too is somewhat inspired by biology. The human eye has 100 megapixels. However only 1 millions fibers out of each eye and this is because the neurons in front of retina compress data.


10 output units
fully connected ~ 300 links
layer H3 30 hidden units
fully connected ~ 6000 links
layer H2 12 x 16=192 hidden units
H2.1
H2.12
~ 40,000 links from 12 kernels 5 x 5 x 8
layer H1 12 x 64 = 768 hidden units
H1.1
H1.12
~20,000 links from 12 kernels 5 x 5
256 input units

CNNs have had an important role in the progress of Deep Learning. Figure from LeCun's 1989 paper on handwritten digit recognition.

RNNs

The neural networks we have considered so far are called *feed forward networks*, meaning that the flow of information in them is in one direction. For sequential data, such as times series, speech or text, each data point is of the form $\{x_t\}_{t=0}^T$ and thus we need to have *two dimensions* in our neural network: one given by t and the other given by the forward flow of the network. RNNs take a sequence $\{x_t\}_{t=0}^T$ as input and outputs another sequence $\{y_t\}_{t=0}^T$. However they have a second, *internal* output called the *state* which we denote by $\{s_t\}_{t=0}^T$. At time t, the network takes x_t and the state s_{t-1} from the last time, and feeds them to the network, which in turn produces an output y_t and another state s_t:

$$(\mathbf{y}_t, s_t) = F(w, \mathbf{x}_t, s_{t-1})$$

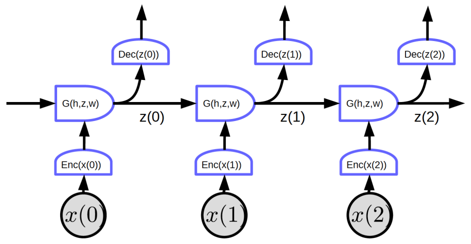


Diagram of an RNN. The state is denoted by z and h is the output of the encoder Enc. G is a (feed forward) neural network and w is its parameter set. Credit: Yann LeCun, Deep Learning.

In supervised learning, RNN are used e.g. for handwriting or speech recognition. Note that the weights w are the same for all the instances of G (this is called *weight sharing*) and thus RNNs can be trained using backpropagation as well.

Long Short Term Memory (LSTM)

A problem with RNNs is that the signal from x_t (which is turned into the state s_t) may be lost as we go further in the t dimension. (This is an example of *vanishing gradients* which can happen in any neural network with many layers or more generally when a vector is multiplied by weight matrices many times. The opposite *exploding gradients* can happen too.) To remedy this, in Long Short Term Memory (LSTM), we add another component called *carry* c_t whose purpose is to carry information from t to $t+T$ where T is a constant.

Attention mechanism and Transformers

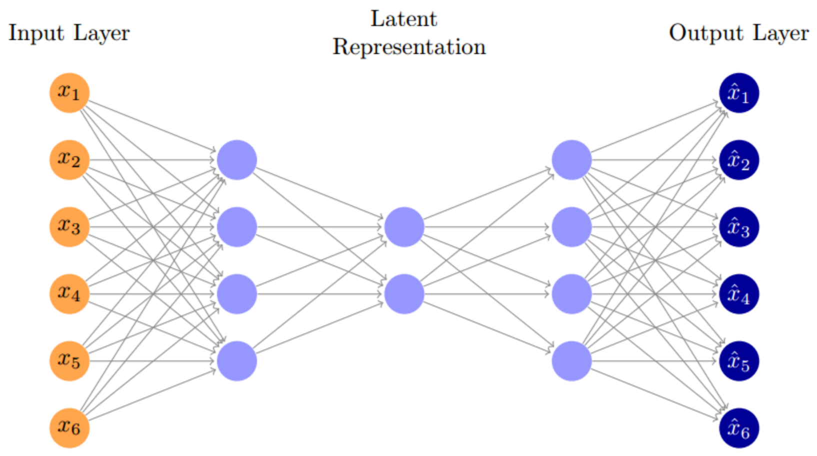
To be added.

Energy-Based Models

EBMs are used when there is no unique output for a given input, e.g. for generative models. EBMs learn an underlying data distribution by analyzing a sample dataset.

Autoencoders

Autoencoders are neural networks that consist of two parts: an *encoder* that reduces data dimension, followed by a *decoder* which increases dimension back to the original. The loss function is given by the norm of the difference of the input and the output. Thus, Autoencoders are *unsupervised learning* methods meaning that they use unlabeled data. They can be used for reducing the dimension of data. The representation of data in the middle of an autoencoder is called a *latent representation*.



An example of a AE that reduces the dimension from 6 to 2 and then increases it back to 6. Credit: Murphey, Probabilistic Machine Learning.

Variational Autoencoders

To be added.

Graph Neural Networks (GNNs)

GNNs are used for data which involves relations. In other words, in GNNs there are relations among the neurons in the same layer.

Contrastive Learning

To be added.

Neuromorphic computing and the third generation of artificial neurons

To be added.

Further learning

- Yann LeCun's Deep Learning course at NYU: https://atcold.github.io/NYU-DLSP21/
- Aston Zhang, et al. Dive into Deep Learning: https://d2l.ai/