# Machine learning

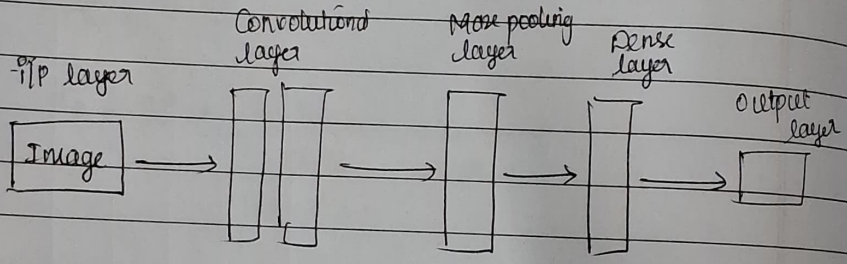
## 1) Convolutional Neural Network (CNN)

* CNN is a type of deep learning neural network architecture commonly used in computer vision, which enables computers to interpret and understand image or visual data

* CNN is an extended version of ANN which extract feature from datasets like images / vedios

### CNN Architecture

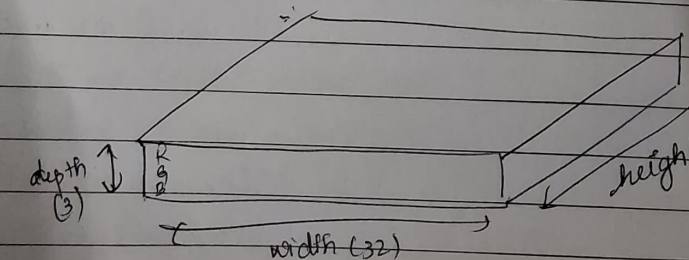
* CNN consists of multiple layers like input layer, Convolutional layer, pooling layer and fully connected layers




[CNN Architecture]

* CNN is a sequence of layers that transforms one volume to another through differential functions

To understand the working, let us consider an example of CNN converting image of dimension 32 x 32 x 3.

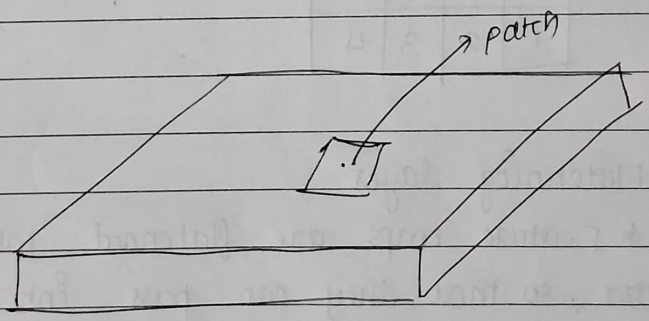


1) Input layer:

It's the layer in which we give input to the model. Generally an image or sequence of images. This layer holds raw input with width 32, height 32, and depth 3.

2) Convolutional layers:

* This is the layer which is used to extract feature from input dataset.
* Now imagine, we are taking a small patch of image and running small neural network, called filter or kernel with k outputs
* Now slide the neural network across whole image, as a result we will get another image with different width, height and depths.
* These filters must have smaller width & height than input image
* Instead of just R G B channels now we have more channels. This operation is called convolution.




* Each slide is called stride
* As a result we get a 2-D output for each filter & we can stack them together, so we get output depth equal to no. of filters / kernals, called feature map
* Suppose we take 12 filters, then output dimension will be 32 x 32 x 12.

3) Activation layer:

* Activation layer added to the output of preceding layer will produce non linearity to the network
* Some common activation functions are RELU: max(0, x), Leaky RELU etc.,
* The volume remains unchanged (32x32x12)

4) Pooling layer:

* Main function of this layer is to reduce dimensionality that makes computation fast, reduces memory and also prevents overfitting
* Two types of pooling are max pooling, average pooling
* If we use max pooling with 2x2 filter and stride 2, the resultant dimension will be 16x16x12

| | | | |
|---|---|---|---|
| 1 | 1 | 2 | 4 |
| 5 | 6 | 7 | 8 |
| 3 | 2 | 1 | 0 |
| 1 | 2 | 3 | 4 |

→

| | |
|---|---|
| 6 | 8 |
| 3 | 4 |

5) Flatenning layer

* Feature maps are flatenned into 1-dimensional vector, so that they can pass into completly linked layer

6) Fully connected layer

It take input from previous layer and performs final classification / regression task

7) Output layer

The output of Fully connected layer is fed into logistic function like softmax which converts output of each class into probability score of each class.

## Multi-Layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer* (see Figure 10-7). The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

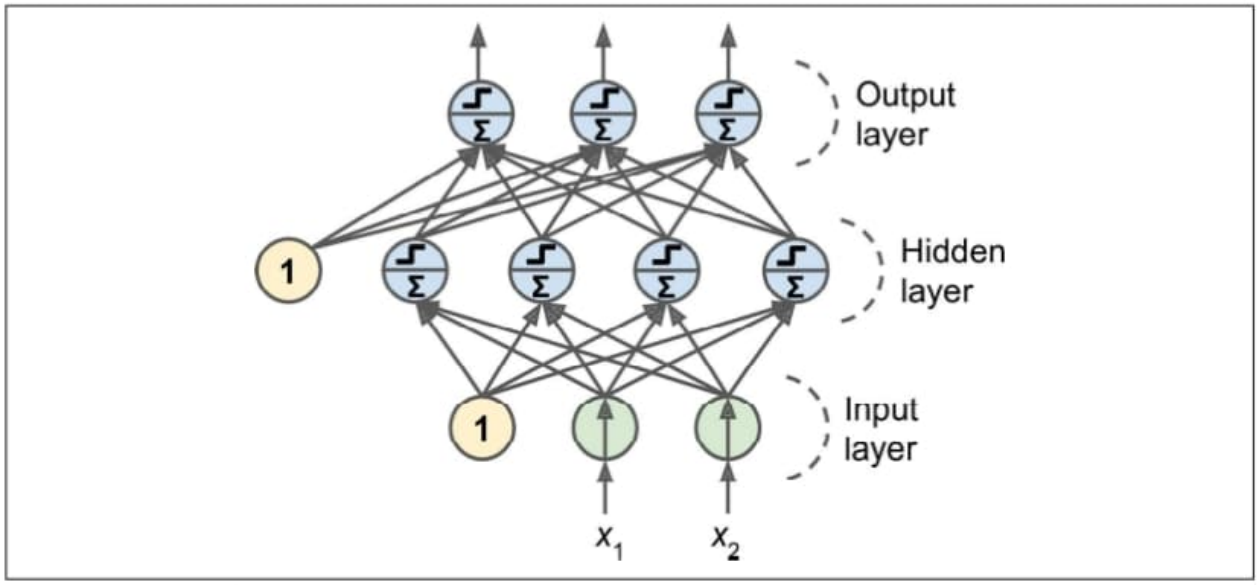



*Figure 10-7. Multi-Layer Perceptron*

The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

When an ANN contains a deep stack of hidden layers[8], it is called a *deep neural network* (DNN). The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations. However, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton and Ronald Williams published a groundbreaking paper[9] introducing the *backpropagation* training algorithm, which is still used today. In short, it is simply Gradient Descent (introduced in Chapter 4)

using an efficient technique for computing the gradients automatically[10]: in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regards to every single model parameter. In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

Automatically computing gradients is called *automatic differentiation*, or *autodiff*. There are various autodiff techniques, with different pros and cons. The one used by backpropagation is called *reverse-mode autodiff*. It is fast and precise, and is well suited when the function to differentiate has many variables (e.g., connection weights) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out ???.

Let's run through this algorithm in a bit more detail:

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*, as we saw in Chapter 4.
- Each mini-batch is passed to the network's input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output connection contributed to the error. This is done analytically by simply applying the *chain rule* (perhaps the most fundamental rule in calculus), which makes this step fast and precise.
- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule—and so on until the algorithm reaches the input layer. As we explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the

network by propagating the error gradient backward through the network (hence the name of the algorithm).

- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

This algorithm is so important, it's worth summarizing it again: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).

## Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object on an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So you end up with 4 output neurons.

11 Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that ReLU generally works better in ANNs. This is one of the cases where the biological analogy was misleading.



In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values. However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function, or the *softplus* activation function in the output layer. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and scale the labels to the appropriate range: 0 to 1 for the logistic function, or –1 to 1 for the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.

The Huber loss is quadratic when the error is smaller than a threshold δ (typically 1), but linear when the error is larger than δ. This makes it less sensitive to outliers than the mean squared error, and it is often more precise and converges faster than the mean absolute error.

Table 10-1 summarizes the typical architecture of a regression MLP.

*Table 10-1. Typical Regression MLP Architecture*

| Hyperparameter | Typical Value |
|---|---|
| # input neurons | One per input feature (e.g., 28 x 28 = 784 for MNIST) |
| # hidden layers | Depends on the problem. Typically 1 to 5. |
| # neurons per hidden layer | Depends on the problem. Typically 10 to 100. |
| # output neurons | 1 per prediction dimension |
| Hidden activation | ReLU (or SELU, see Chapter 11) |
| Output activation | None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs) |
| Loss function | MSE or MAE/Huber (if outliers) |

## Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. Obviously, the estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks (see Chapter 3). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent



or non-urgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to one. This lets the model output any combination of labels: you can have non-urgent ham, urgent ham, non-urgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of 3 or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the *softmax* activation function for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to one (which is required if the classes are exclusive). This is called multiclass classification.

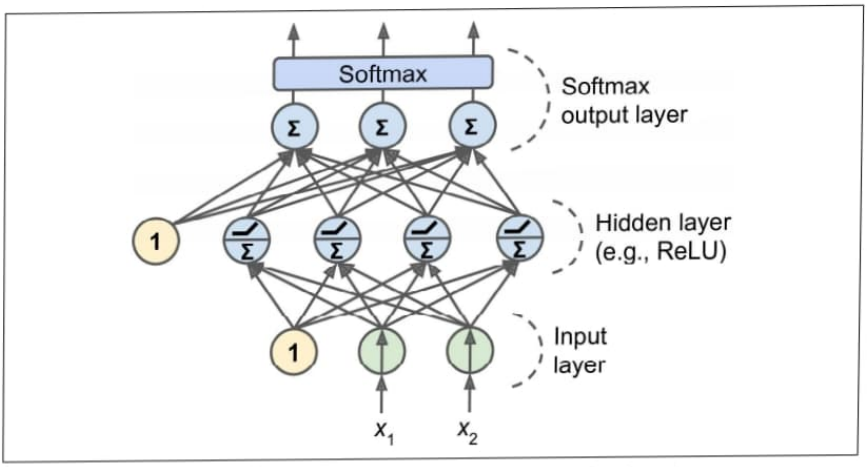



*Figure 10-9. A modern MLP (including ReLU and softmax) for classification*

Regarding the loss function, since we are predicting probability distributions, the cross-entropy (also called the log loss, see Chapter 4) is generally a good choice.

*Table 10-2. Typical Classification MLP Architecture*

| Hyperparameter | Binary classification | Multilabel binary classification | Multiclass classification |
|---|---|---|---|
| Input and hidden layers | Same as regression | Same as regression | Same as regression |
| # output neurons | 1 | 1 per label | 1 per class |
| Output layer activation | Logistic | Logistic | Softmax |



| Hyperparameter | Binary classification | Multilabel binary classification | Multiclass classification |
|---|---|---|---|
| Loss function | Cross-Entropy | Cross-Entropy | Cross-Entropy |