Machine Learning, with Examples in PyTorch

Matthew Mahowald

August 13, 2021

Outline

What is machine learning?

What is machine learning?

A possible definition:

Machine learning is a class of general purpose algorithms that utilize data to make predictions or decisions without being explicitly programmed to do so.

A quick note on data

Most (but not all) machine learning algorithms take in real-valued vectors and produce real-valued vectors.

$$\begin{bmatrix} 1 & 2.5 & -3.6 & 0 \end{bmatrix} \overset{f}{\mapsto} \begin{bmatrix} 0.9 & 7 & -4.0 \end{bmatrix}$$

But what if your problem doesn't obviously involve numbers?

It turns out most things can be represented as vectors.

Example 1: Coin flip

Heads $= [1]$, Tails $= [0]$

Example 2: Classification: dog/cat/bird

Dog $= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$, Cat $= \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$, Bird $= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$

More examples of data encodings

Example 3: The alphabet

$$\text{``a''} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}$$
$$\text{``b''} = \begin{bmatrix} 0 & 1 & \cdots & 0 \end{bmatrix}$$
$$\vdots$$
$$\text{``z''} = \begin{bmatrix} 0 & 0 & \cdots & 1 \end{bmatrix}$$

(this is called a "one-hot" encoding)

Example 4: Images

- Each pixel may be represented as a tuple (r, g, b), where r determines the red content, g the green content, and b the blue content.
- A whole image that is W pixels wide and H pixels high can be represented by an array of size $W \times H \times 3$.

Why vectors?

Fundamentally, all machine learning models are mathematical functions, and therefore they must act on mathematical objects. (And an array of real numbers is the easiest nontrivial mathematical object to do computations with.)

3 (& $\frac{1}{2}$) Types of Machine Learning Algorithms

1. Supervised learning
2. Unsupervised learning

   (2.5) Self-supervised learning
3. Reinforcement learning

In all 3.5 cases, the output of the algorithm is a **model**, or more specifically, a collection of parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_n)$ that define a specific instance of a particular class of models.

The type of algorithm determines *what* class of model, and *how* the parameters θ are chosen.

Supervised learning

Supervised : The algorithm is “taught” to produce outputs from inputs on a “training” dataset.

Many (most?) of the successful business applications of machine learning are supervised.

Examples:

- **Image recognition**: Multi-label classification problem
- **Machine language translation**: Sequence-to-sequence multi-label classification
- **Credit scoring**: Estimate probability of loan default

1. Supervised learning

Main ingredients:

- A set of input data $X = \{x_1, x_2, \ldots, x_n\}$
- A set of desired outputs $Y = \{y_1, y_2, \ldots, y_n\}$

The goal is to construct a function (the *model*) $f : X \mapsto Y$ such that

$$f(x_i) \approx y_i$$

for all i.

Here, $\approx$ means that we want to minimize some measure of the error between the model's prediction $f(x_i)$ and the correct output y_i.

1. Supervised learning

$$f(x_i) \approx y_i$$

We can formalize the measure of error through a **loss function**:

$$\mathcal{L}(f(x_i), y_i)$$

Then, the goal is to find f that minimizes the loss function $\mathcal{L}(f(x_i), y_i)$.

Example

When $y_i \in \mathbb{R}$, common choices are the *mean squared error*:

$$\mathcal{L}(f(x_i), y_i) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

And the *mean absolute error*:

$$\mathcal{L}(f(x_i), y_i) = \frac{1}{n} \sum_{i=1}^{n} |y_i - f(x_i)|$$

1. Supervised learning: What about categorical problems?

The coinflip problem: $y_i = H$ with some (unknown) probability p, and otherwise $y_i = T$. Suppose we also have a set of observed flips $\{y_1 = H, y_2 = T, \ldots, y_n = H\}$. How do we estimate p?

The probability of that particular collection of flips is:

$$P(\{y_1, y_2, \ldots, y_n\}) = \left(\prod_{y_i=H} p\right)\left(\prod_{y_j=T}(1-p)\right)$$

Maximizing this probability (the *likelihood*) is the same as maximizing the log-probability:

$$\log P = \sum_{y_i=H} \log p + \sum_{y_i=T} \log(1-p)$$

1. Supervised learning: Categorical cross-entropy

The log-probability was

$$\log P = \sum_{y_i = H} \log p + \sum_{y_i = T} \log(1 - p)$$

Now suppose that the probability varies with each flip based on some feature x_i, and set $f(x_i) := P(y_i = H | x_i)$. Then the *log likelihood* is

$$\log P(\{y_1, y_2, \ldots, y_n\}) = \sum_{y_i = H} \log f(x_i) + \sum_{y_j = T} \log f(x_j)$$

Thus the negative log likelihood gives a *categorical loss function*:

$$\mathcal{L}(y_i, f(x_i)) := -\left(\sum_{y_i = H} \log f(x_i) + \sum_{y_j = T} \log f(x_j) \right)$$

This is called *categorical cross entropy*.

2. Unsupervised learning

Unsupervised : The algorithm is not supplied with an "answer key", and instead assigns data to (machine-learned) clusters based on patterns.

Compared to supervised learning, the main disadvantage is that clusters identified by these approaches may not have useful/actionable interpretations.

Examples

- **Anomaly detection**: Identify observations that are distinct and unusual relative to the rest of the data
- **Recommendation engines**: Users with similar characteristics are supplied similar product recommendations
- **Topic modeling**: Determine the topic associated with a particular sequence of text

2. Unsupervised learning

Main ingredients:

- A set of input data $\{x_1, x_2, \ldots, x_n\}$
- A distance/similarity metric $d(x, y)$

Distance metric: In order to tell how similar two observations x_i and x_j are to each other, we need to define a distance metric d such that $d(x_i, x_i) = 0$ and $d(x_i, x_j) \geq 0$ for all i, j.

When no notion of distance is supplied, the implied choice is the *Euclidean distance*:

$$d(x_i, x_j) = \sqrt{(x_{i,1} - x_{j,1})^2 + (x_{i,2} - x_{j,2})^2 + \cdots + (x_{i,m} - x_{j,m})^2}$$

But sometimes this is not the right choice!

2.5 Self-supervised learning

A blend of supervised and unsupervised learning.

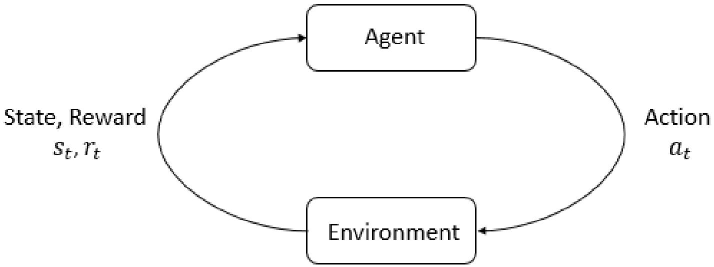
- Supervised: $y_i = x_i$
- Unsupervised: $\mathcal{L}(y_i, f(x_i)) = \mathcal{L}(x_i, f(x_i)) = d(x_i, f(x_i))$

Examples

- **Anomaly detection** (again): Measure the distance between the input and the output; anomalous values will not be easily reconstructed.
- **Compression/Encoder-decoder**: The input x_i is mapped (encoded) to a smaller representation, and then decompressed (decoded) back to the original input.

3. Reinforcement learning

An agent is taught to act in an environment, with the goal of maximizing some reward.



Examples

- **AlphaZero/AlphaStar/etc**: game-playing models
- **Robot control**: e.g. teach a robotic arm to install a car bumper

3. Reinforcement learning

Main ingredients:

- An **environment** which takes an action a_t and a state s_t and produces a new state s_{t+1}
- An **agent** which takes a state s_t and produces an action a_t
- A **reward function** R which takes a tuple (s_t, a_t, s_{t+1}) and produces a reward r_t

A sequence $\tau = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$ is called a **trajectory**.

The goal is to find a function π (the **policy**) which maximizes the expected reward across many trajectories:

$$J(\pi) := \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=1}^{n} R(s_t, a_t = \pi(s_t), s_{t+1}) \right]$$

Advantages and Drawbacks of Different Approaches

	Supervised	Unsupervised	Self-supervised	Reinforcement
Advantages	· Easiest to implement · Most control over performance	· No explicit labels needed · Good for data exploration	· No explicit labels needed	· Closest match to biological learning · Highly flexible behaviors
Disadvantages	· Need explicit labels · Sensitive to choice of loss function · Models are only as good as the labels	· Results not always useful for the selected task	· Limited applications	· Very sample-inefficient · May not converge to viable policy · Difficult to design a good reward

Deep learning

Linear regression

$$\hat{y}_i = \theta_0 + \theta_1 x_{i,1} + \theta_2 x_{i,2} + \cdots + \theta_n x_{i,n}$$

The input $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})$ are the observations (vectors), where the $x_{i,k}$ is the kth *feature* of the observation. $\hat{y}_i$ is the model's *predicted* valule given the input x_i. θ_k are the *parameters* of the model.

Can re-write LR as a matrix equation:

$$\hat{y}_i = \theta_0 + \begin{bmatrix} \theta_1 & \theta_2 & \cdots & \theta_n \end{bmatrix} \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{bmatrix}$$

Linear regression

What if y_i is a vector, too?

Then there are a lot of parameters $\theta_{j,k}$!

$$\begin{bmatrix} \hat{y}_{i,1} \\ \hat{y}_{i,2} \\ \vdots \\ \hat{y_{i,m}} \end{bmatrix} = \begin{bmatrix} \theta_{1,1} & \theta_{1,2} & \cdots & \theta_{1,n} \\ \theta_{2,1} & \theta_{2,2} & \cdots & \theta_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{m,1} & \theta_{m,2} & \cdots & \theta_{m,n} \end{bmatrix} \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{bmatrix} + \begin{bmatrix} \theta_{0,1} \\ \theta_{0,2} \\ \vdots \\ \theta_{0,m} \end{bmatrix}$$

Or more compactly:

$$\hat{y}_i = Wx_i + b$$

Observe that **linear regression is a supervised learning problem**. For univariate regression, the loss function is:

$$\mathcal{L}(y_i, f(x_i)) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Logistic regression

Sometimes, we want to predict probabilities (classification) rather than real numbers (regression). In this case, we have

$$\hat{y}_i = \sigma(Wx_i + b)$$

where σ is some *linking function*—for example, the sigmoid function:

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

(the sigmoid function maps every real number to a number on the interval $[0, 1]$).

Fitting this kind of model is also an example of supervised learning (where the loss function is the categorical cross-entropy).

How do we find weight matrices that minimize the loss function?

For classical univariate linear regression, you can actually solve the equation analytically. In general, you need to use a numerical optimization algorithm.

One way is gradient descent:

$$\theta_{k+1} = \theta_k - \gamma \, \nabla_\theta \mathcal{L}(y, f_\theta(x))\big|_{\theta=\theta_k}$$

where γ is the "learning rate" (the step size).

If the loss function is convex in θ, then gradient descent is guaranteed to converge to the global minimum.

Okay, but I thought this talk was about deep learning. . .

Logistic regression is a neural net!

Recall that logistic regression was defined by:

$$\hat{y} = \sigma(Wx + b)$$

where x, $\hat{y}$, and b are all vectors, and σ is applied elementwise.

Now imagine that instead of a single matrix of weights W and bias b, we stack a sequence of them:

$$\hat{y} = W_3\left(\sigma\left(W_2\left(\sigma\left(W_1x + b_1\right)\right) + b_2\right)\right) + b_3$$

This is the mathematical definition of a “feed forward” neural net with 3 hidden layers and activation function σ.

A picture of a neural net

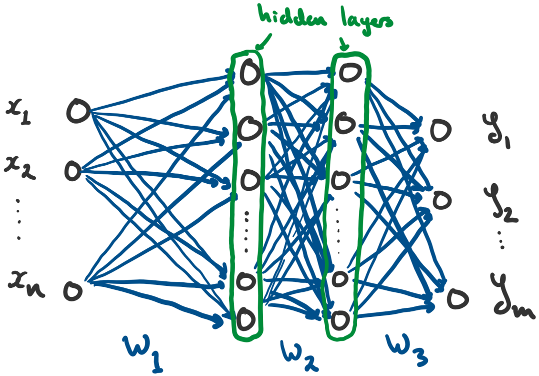


Figure 1: A fully-connected neural net.

- Each “neuron” is a component of some vector
- The weights W_i, b_i define the “connections” between neurons in the network
- Biological neural nets do not really function this way.

What are hidden layers?

Suppose that x is an n-dimensional vector, and y is an m-dimensional vector.

Consider our example:

$$\hat{y} = W_3 \left(\sigma \left(W_2 \left(\sigma \left(W_1 x + b_1\right)\right) + b_2\right)\right) + b_3$$

Matrix W_1 must be dimension $k_1 \times n$ for some k_1, W_2 must be dimension $k_2 \times k_1$, and W_3 must be dimension $m \times k_3$.

In particular we are free to choose k_1, k_2, and k_3 to be arbitrary positive integers. These intermediate vectors are the "hidden layers" in the network.

Another perspective on neural nets

Consider again

$$\hat{y} = W_3\left(\sigma\left(W_2\left(\sigma\left(W_1 x + b_1\right)\right) + b_2\right)\right) + b_3$$

Let $\tilde{x}$ denote the "latent representation"

$$\tilde{x} = \sigma\left(W_2\left(\sigma\left(W_1 x + b_1\right)\right) + b_2\right)$$

Then, the defining equation of the neural net is just linear (or logistic) regression on this latent representation!

$$\hat{y} = W_3 \tilde{x} + b_3$$

$\Longrightarrow$ **We can think of a neural net as being a combination of feature engineering and then classic linear or logistic regression!**

Yet another perspective on neural nets

Neural nets are approximately piecewise linear functions:

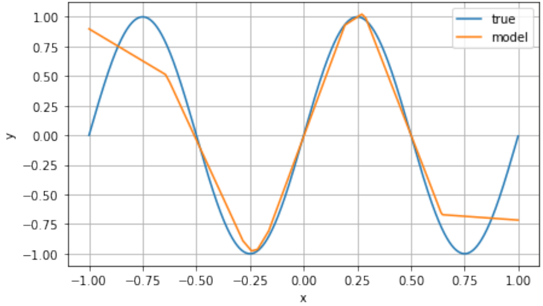


Figure 2: A low-parameter net fit to a sine curve. This net has only 97 free parameters, and uses a ReLU activation function.

When the activation function is ReLU, they are exactly piecewise linear!

Why are neural nets so successful?

1. They are universal approximators

More precisely, they're "dense in $L^2(\mathbb{R}^n)$": for any function f and any $\epsilon > 0$, there exists a neural net g such that

$$\sqrt{\int_{\mathbb{R}^n} |f(x) - g(x)|^2 \, dx} \leq \epsilon$$

- $L^2(\mathbb{R}^n)$ includes basically all but the very worst-behaved real-valued functions, so this means **in theory** a net can model just about any function
- But, being dense in $L^2(\mathbb{R}^n)$ isn't all that special—polynomials are too!

Why are neural nets so successful?

2. They are conservative interpolators

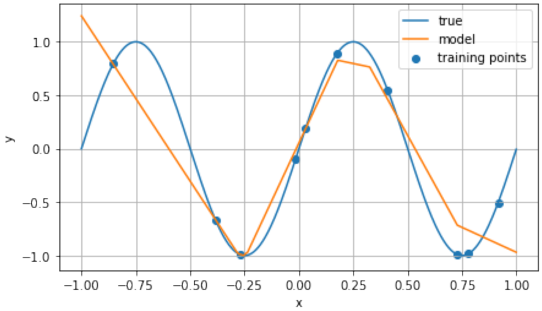


Figure 3: In contrast to a polynomial, a net will linearly interpolate between training observations. This means that in-domain predictions are a weighted average of learned observations, and out-of-sample predictions are a linear extrapolation from the training data.

Why are neural nets so successful?

3. Gradient descent is surprisingly effective in high dimensions

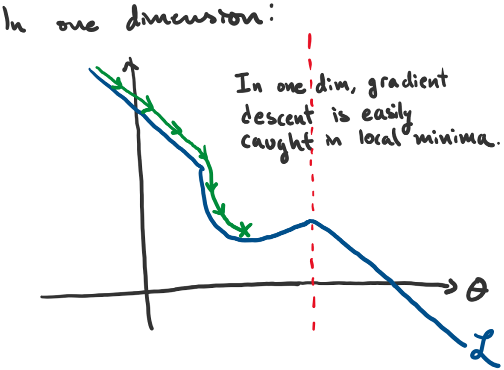

In one dimension:
In one dim, gradient descent is easily caught in local minima.
θ
$\mathcal{L}$

Figure 4: In low dimensions, gradient descent can easily get stuck. In fact, if there is a local minimum, you have a 50% chance of picking a starting point that leads to it!

Why are neural nets so successful?

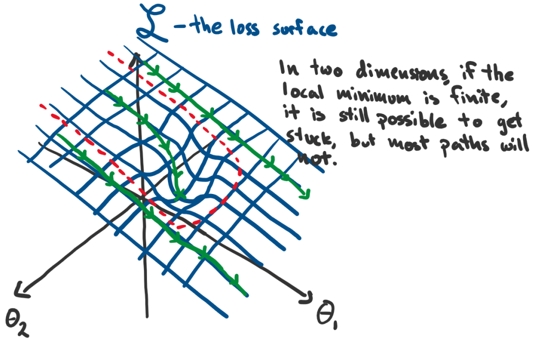

$\mathcal{L}$ - the loss surface
In two dimensions, if the local minimum is finite, it is still possible to get stuck, but most paths will not.
θ_2
θ_1

Figure 5: In two dimensions, it's still easy to get stuck, but as long as the local minimum is finite, you are less likely to pick a bad starting point.

As the dimension of the parameter space increases, as long as the local minimum is finite, it becomes increasingly difficult to find gradient paths that will get stuck.

A typical neural net may have a 100,000-dim'l parameter space!

Examples

Common neural architectures: Fully connected feed-forward (“multilayer perceptron”)

This is the most basic type of network:

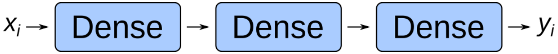


Figure 6: A dense, fully connected network.

In PyTorch this can be implemented by:

```
model = nn.Sequential(
    nn.Linear(n, k1),
    nn.ReLU(),
    nn.Linear(k1, k2),
    nn.ReLU(),
    # or add more hidden layers
    nn.Linear(k2, m),
    )
```

Common neural architectures: Autoencoders

Autoencoders are (typically) just a special kind of feedforward network:

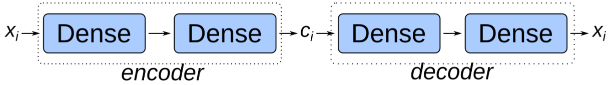


Figure 7: An autoencoder consists of two stacked fully connected networks: an encoder network, and a decoder network.

The network is trained to reproduce its output x_i from its input x_i. However, along the way, it must produce an “encoded” (typically, lower-dimensional) representation of the input, c_i.

Common neural architectures: Autoencoders

Here's an example of an autoencoder in PyTorch:

```
class Autoencoder(nn.Module):
    def __init__(self, input_size, encoded_size):
        super().__init__()

        self.encode = nn.Sequential(
            nn.Linear(input_size, 128), nn.ReLU(),
            nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, encoded_size))
        self.decode = nn.Sequential(
            nn.Linear(encoded_size, 128), nn.ReLU(),
            nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, input_size))

    def forward(self, x):
        return self.decode(self.encode(x))
```

Common neural architectures: Recurrent networks

A recurrent network maintains a “hidden state”, which is updated with each new input.

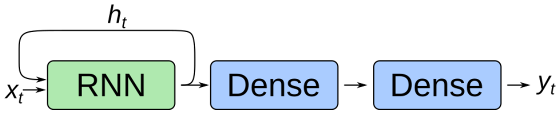


Figure 8: A recurrent network with one recurrent layer and two dense layers

The most basic recurrent layer is just a dense layer:

$$h_t = \sigma(W(x_t, h_{t-1}) + b)$$

where, if x_t is dimension n and h_t is dimension k, then W is a $k \times (k + n)$-dimensional matrix.

Typically, the hidden state h_t is then used to as an input to subsequent feedforward layers to generate a prediction.

Common neural architectures: Recurrent networks

Here's an explicit example of the previous slide's RNN:

```
class ModelRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()

        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size)
        self.dense = nn.Sequential(
            nn.Linear(hidden_size, 128), nn.ReLU(),
            nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, 1))

    def forward(self, x, h):
        x = torch.unsqueeze(x, 0)
        x, h = self.rnn(x, h)
        x = torch.squeeze(x, 0)
        x = self.dense(x)
        return x, h
```

Tips and tricks to get the most out of your nets

- **Normalize your inputs**: Nets work best when each feature is drawn from an approximately normal distribution with mean 0 and std dev 1. (Do this by subtracting the *in-sample* mean and dividing by the *in-sample* std dev)
- **Use dropout and early stopping**: Dropout randomly zeros neurons during training; this helps prevent memorization of training data. And early stopping halts training when performance starts to degrade on the validation set.
- **When in doubt, enlarge and add more layers**: Traditional statistical wisdom about parameter counts (e.g. AIC/BIC) doesn't seem to apply as well to nets.
- **Check that your code works on CPU before running it on GPU**: If it breaks, at least you will know whether it's a GPU problem or not!

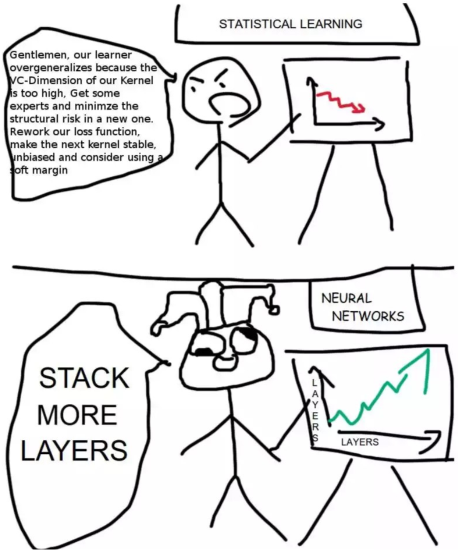

STATISTICAL LEARNING
Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high, Get some experts and minimze the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin
NEURAL NETWORKS
STACK MORE LAYERS
LAYERS
LAYERS

image credit: unknown redditor

Thank you!