Alex Meng's Notes

Alex Meng

2025-06-06

Table of contents

Preface

If you are reading this, you may be interesed in seeing what is "Alex's Notes".

These notes are just things that I am documenting, that I wish could become a useful resource for my future students, either when I TA or become a professor.

Here's how to learn anything:

1. Write it out! (Document, Code along)
2. EXPERIMENT and explore
3. Visualize things you don't understand
4. Ask Questions
5. Answer Exercise and Problems (stretch your knowledge)
6. Share with like-minded individuals

Part I

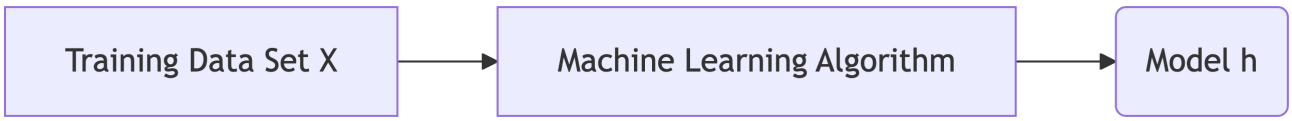
MACHINE LEARNING

Deep Learning Systems

In this set of notes, we will create a minimal version of PyTorch / Tensorflow from scratch, of what I call "PyStickOnFire".

1 ML Basics

The (Supervised) Machine learning idea: we take a bunch of labeled data, feed them to a machine learning algorithm, and it outputs a "program" that solves the task.



In this set of notes, we will focus on what the machine learning algorithm box contains. In general, it consists of three things:

1. The hypothesis class (the structure of h in terms of a set of parameters)
2. The loss function (specifies how good a given hypothesis is)
3. An optimization method (the way to minimize the loss function)

All alogrithms in machine learning fit in this structure. Let's look at **softmax regression** to illustrate these three basic components.

Multi-class Classification (Softmax Regression)

Consider a *k-class classification setting*, where we have

- training data:
$$x^{(i)} \in \mathbb{R}^n, y^{(i)} \in \{1, \dots, k\} \text{ for } i = 1, \dots, m$$
  - n = dimensionality of input data
  - k = number of different classes / labels
  - m = number of data points in the training data

where the training data are vectors that looks like

$$X = \{ \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ \vdots \\ x_n^{(1)} \end{bmatrix}, \dots, \begin{bmatrix} x_1^{(m)} \\ x_2^{(m)} \\ \vdots \\ x_n^{(m)} \end{bmatrix} \}$$

and the labels are just a set of scalars of size k.

1st Element: The Hypothesis Function

The hypothesis function is a mapping from one input to one output. (Duhh… just like every other function there is).

$$h : \mathbb{R}^{\mathbb{n}} \to \mathbb{R}^{\mathbb{k}}$$

$$h(x) = \begin{bmatrix} h_1(x) \\ h_2(x) \\ \vdots \\ h_k(x) \end{bmatrix}$$

So what really is $h_i(x)$? It is the hypothesis, the "belief", the probability of how likely x maps to class i.

A **linear hypothesis function** uses matrix multiplication, or some other linear way, for this transformation:

$$h_\theta(x) = \theta^T x$$

for parameters $\theta \in \mathbb{R}^{\mathbb{n} \times \mathbb{k}}$ (n rows and k columns, so transpose becomes $k \times n$, and $x \in \mathbb{R}^{\mathbb{n} \times \mathbb{1}}$, so multiplication will work). Now we say h_θ because θ is the parameters.

Notice how so far we only have one input and one output, h is only working on one instance of the training set. However, in order to implement these operations efficiently in the future, we shall use the **matrix batch notation**.

$$X \in \mathbb{R}^{\mathbb{m} \times \mathbb{n}} = \begin{bmatrix} -x^{(1)^T} - \\ -x^{(2)^T} - \\ \vdots \\ -x^{(m)^T} - \end{bmatrix}$$

$$y \in \{1, \dots, k\}^m = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

$$h_\theta(X) = \begin{bmatrix} -h_\theta(x^{(1)})^T- \\ -h_\theta(x^{(2)})^T- \\ \vdots \\ -h_\theta(x^{(m)})^T- \end{bmatrix} = \begin{bmatrix} -x^{(1)^T}\theta- \\ -x^{(2)^T}\theta- \\ \vdots \\ -x^{(m)^T}\theta- \end{bmatrix} = X\theta$$

- n = dimensionality of input data
- k = number of different classes / labels
- m = number of data points in the training data

Each row is for a data point, the first example is in first row (originally a column vector, now we transposed it to row vector), and the second example is in second row, and so on… Note that this is not merely a notation change, but rather how to implement them more efficiently in code later on.

2nd Element: The Loss Function

How are we going to evaluate the quality of our predictions?

Classification Error:

$$l_{err}(h(x), y) = \begin{cases} 0, & \text{if } argmax_i h_i(x) = y \\ 1, & \text{otherwise} \end{cases}$$

The error is not differentiable, so it is not good for optimization.

A better choice: Cross-Entropy Loss or Softmax

The idea is that we want to map our outputs into being actual probabilities

$$h_i(x) \rightarrow prob[label == i]$$

Probability has to be positive and sum to 1. In order to ensure $h_i(x)$ is positive, we can exponentiate it. In order to ensure all $h_i(x)'s$ to sum to 1, we need to normalize them.

$$prob[label == i] = normalize(\exp(h(x))) = \frac{\exp(h_i(x))}{\sum_{j=1}^{k} \exp(h_j(x))} = softmax(h(x))$$

This is called the **softmax operation**, a mapping between scalar values and a probability distribution.

So now we have a probability, We need some way of quantifying whether the vector of probabilities $softmax(h(x))$ is good or not. We want $prob[label == y]$ to be high, as large as

possible, so the loss function idea can be minimizing the negative of this probability (double negative makes a positive!).

$$l_{cross-entropy}(h(x), y) = -prob[label = y]$$

Because minimizing probabilities is not numerically good: Probabilities are bounded between 0 and 1, so their gradients near 0 can become tiny (vanishing gradients). Logs transform that range $(0, 1)$ into $(-\infty, 0)$, making the loss surface smoother and the gradients more useful. So we take the log of it

$$l_{ce}(h(x), y) = -log(prob[label = y])$$

This is commonly known as the **negative log loss** or **cross-entropy loss**.