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:

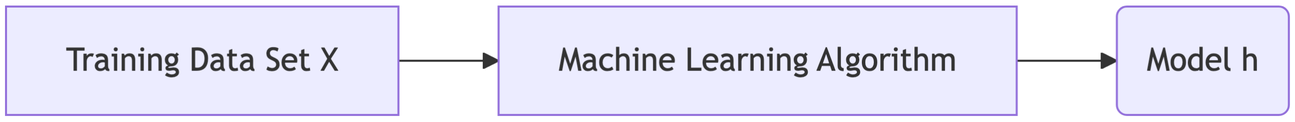
1. Write it out! (Document, Code along)
2. EXPERIMENT and explore
3. Visualize things you don't understand
4. Ask Questions
5. Answer Exercise and Problems (stretch your knowledge)
6. Share with like-minded individuals

Deep Learning Systems

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

Chapter 1: Machine Learning Refresher

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.



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

$$l_{ce}(h(x), y) = -log(prob[label = y]) = -h_y(x) + log \sum_{j=1}^{k} exp(h_j(x))$$

This is commonly known as the **negative log loss** or **cross-entropy loss**. This is also a case of *convex optimization.*

3rd Element: Optimization

How do we find good values for θ?

This element we will spend the most time to cover because we not only what to know what the optimization is, but how we are optimizing it.

The following problem is the problem that almost all machine algorithms are solving. Here is the problem,

$$\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} l(h_\theta(x^{(i)}), y^{(i)})$$

We are searching over all possible values of θ, the one that minimizes the *average* loss. For example, here's softmax regression,

$$\min_{\theta} f(\theta) = \min_{\theta} \frac{1}{m} \sum_{i=1}^{m} l_{ce}(\theta^T x^{(i)}, y^{(i)})$$

Now we know the "what", but, how do we find that? How do we solve $\min_\theta f(\theta)$?

The Gradient

For our $f(\theta)$ (the function that we are trying to minimize), remember, this function takes the parameter (which is of dimension n examples by k output classes) and outputs a loss scalar, in other words

$$f : \mathbb{R}^{\mathbb{n} \times \mathbb{k}} \to \mathbb{R}$$

$$f(\theta) \in \mathbb{R}$$

Remember that the gradient is a multidimensional derivative that has a direction, which points to the direction of *sharpest increase (locally).* The gradient operator can only act on a scalar and return a vector.

Here is the definition of the gradient in our case,

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f}{\partial \theta_{11}} \frac{\partial f}{\partial \theta_{12}} \cdots \frac{\partial f}{\partial \theta_{1k}} \\ \frac{\partial f}{\partial \theta_{21}} \frac{\partial f}{\partial \theta_{22}} \cdots \frac{\partial f}{\partial \theta_{2k}} \\ \vdots \\ \frac{\partial f}{\partial \theta_{n1}} \frac{\partial f}{\partial \theta_{n2}} \cdots \frac{\partial f}{\partial \theta_{nk}} \end{bmatrix} \in \mathbb{R}^{\mathbb{n} \times \mathbb{k}}$$

The derivative of a function is the slope of the function, change in y over change in x.

Gradient Descent

If the gradient points in the direction of maximum increase, to minimize a function, we can repeatedly step in the opposite direction. In other words,

$$\theta = \theta - \alpha \nabla_\theta f(\theta)$$

where α is called the *learning rate* or *step size.* Note that the learning rate must be positive for the stepping direction to be in the negative direction from the gradient. **This basic idea powers all deep learning. It is really hard to encapsulate how impactful this one line of math has been.**

The choice of the step size α is really really really important. Too small slows down progress, but too big will overshoot.

Stochastic Gradient Descent

We split up the dataset into *minibatches*, which are subsets of data of size B.

We repeat the process of sampling minibatches and taking steps to update θ.

- Sample: $X \in \mathbb{R}^{\mathbb{B} \times \mathbb{n}}, y \in \{1, \dots, k\}^B$
- Update: $\theta = \theta - \frac{\alpha}{B} \sum_{i=1}^{B} \nabla_\theta l(h_\theta(x^{(i)}), y^{(i)})$

Calculating the gradient in practice

In order to calculate the gradient of $f(\theta)$, which is essentially the sum of gradients,

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} l(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta l(h_\theta(x^{(i)}), y^{(i)})$$

We have to calculate the gradient m times, which is very expensive. *Can we reduce the number of times we take gradient?* Yes, and that's what we actually do in practice.

As an example, how do we compute the gradient for softmax objective? We can do it by hand, but it is cumbersome. You can use something like chain rule. We want derivative of a vector with respect to a matrix……We need some more general and generic way to take derivatives. What can we do?

In practice, we just specify the hypothesis function and the loss function, and use **Automatic Differentiation**. But how?

We can either do it through the "right" way, use matrix differential calculus, jacobians, kronecker products, and vectorization. Or we could take a shortcut (**what everyone actually does**): We pretend everything is scalar, use typical chain rule, and then transpose/rearrange outputs to make sizes work, and then check your answers numerically.

$$\frac{\partial}{\partial \theta} l_{ce}(\theta^T x, y) = \frac{\partial l_{ce}(\theta^T x, y)}{\partial \theta^T x} \cdot \frac{\partial \theta^T x}{\partial \theta}$$

Optics

Q1: One Dimensional Wave Equation

The wave equation is given by

$$\psi(x,t) = \frac{3}{[10(x-vt)^2+1]}$$

Show, using brute force, that this is a solution to the one dimensional differential wave equation.

Great! Let's start with what is a wave.

Def. *A classical traveling wave is a self-sustaining disturbance ψ of a medium, and the disturbance ψ moves through space transporting energy and momentum.*

Everything is waves.

Sound! A type of **longitudinal** wave, where the displacement vector points parallel to the direction of motion.

Guitar string! A type of **transverse** wave, where the displacement vector points perpendicular to the direction of motion.

A wave is not a stream of particles! Because the individual atoms stay in equilibrium, but only the disturbance advances through them. Leonardo da Vinci was one of the first person to realize waves does not transport the medium through which it travels.

Imagine disturbance ψ moves in positive direction x with constant velocity v.

$$\psi = f(x,t)$$

What is $f(x,0)$? it is the shape (aka the **profile**) of ψ at $t=0$. For example, try visualizing $f(x) = e^{-ax^2}$, you'll see that it is a **gaussian function**. Setting $t=0$ is taking a snapshot of the pulse as it travels by.

In order to understand this better, let's ignore t by introducing a coordinate system $S^{'}$ that travels with the pilse at the speed v. As we move with $S^{'}$, the wave looks stationary! So

$$\psi = f\left(x^{'}\right)$$

where $x^{'} = x - vt$, because after time t the same point on ψ moved a distance of vt.

General Form of One Dimensional Wave Function

$$\psi(x,t) = f(x - vt)$$

Jean Le Rond d'Alembert was the one that brought partial differential equations to physics and formulated the differential wave equation.