Lecture 2: ML review (1)

CS 182/282A (“Deep Learning”)

2022/01/24

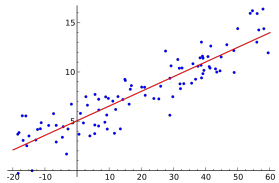
Today's lecture

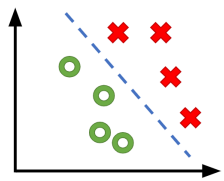
- In this lecture and the next lecture, we will go over concepts at the core of machine learning as a whole
  - We will focus on concepts that are the most relevant to deep learning
- Much of this will be review if you have already taken a machine learning course
- Today, we will focus on the supervised learning problem setup, go over the general machine learning method, and define **probabilistic models**, **likelihood based loss functions**, and **gradient based optimization**

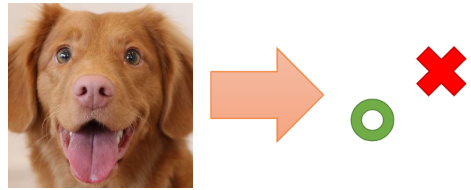
Different classes of learning problems

(non exhaustive)

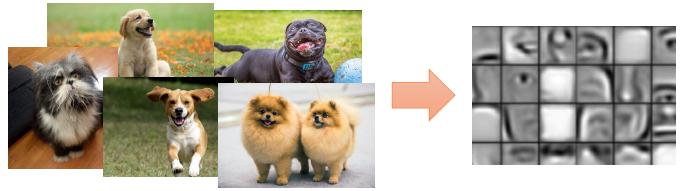
Supervised learning



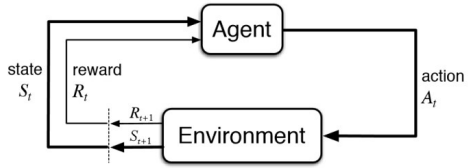


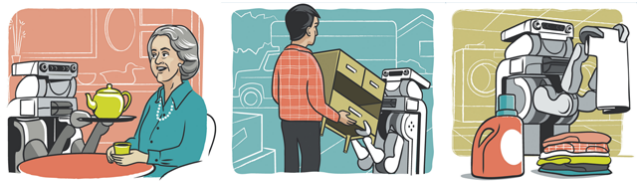


Unsupervised learning



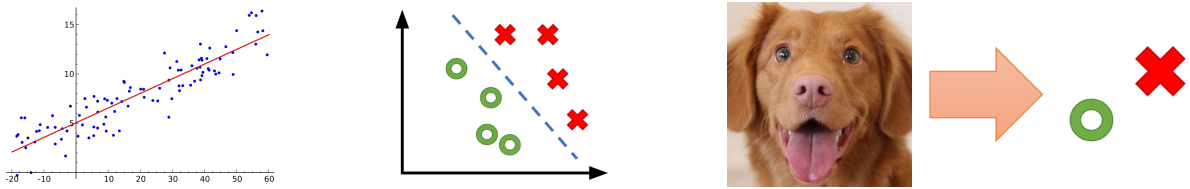
Reinforcement learning





Let’s start with supervised learning

Supervised learning



- In supervised learning, we are given a dataset $\mathscr{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$
- Our goal is to learn a model that predicts outputs given inputs: $f_\theta(\mathbf{x}) = y$
- This setup encompasses the overwhelming majority of machine learning that is used in industry (a multi-billion $/year industry)
- The basic principles are simple, and we will cover them in this lecture

Examples of supervised learning problems

(that deep learning has done really well on)

$\mathbf{x}$	y
image of object	category of object
sentence in English	sentence in French
audio utterance	text of what was said
amino acid sequence	3D protein structure

Should the model just output y?

What could go wrong?

Image	0?	1?	2?	3?	4?	5?	6?	7?	8?	9?
	0%	0%	0%	60%	0%	35%	0%	0%	0%	5%
	0%	0%	0%	0%	50%	0%	0%	0%	0%	50%
	30%	0%	70%	0%	0%	0%	0%	0%	0%	0%

Predicting probabilities

- Often, it makes more sense to have the model predict output *probabilities*, rather than the outputs themselves
  - This can better capture when the model is *uncertain* about difficult inputs
  - We'll also see later why this makes the learning process easier
- So instead of the model output $f_\theta(\mathbf{x})$ being a single y, it will instead be an entire distribution over all possible y!
  - E.g., for digit recognition, the output will be 10 numbers between 0 and 1 that sum to 1

How do we output probabilities?

- How do we make our model output numbers between 0 and 1 that sum to 1?
- Idea: first let our model output whatever numbers it wants
  - Then, make all the numbers positive and *normalize* (divide by the sum)
- There are many ways to make a number z positive
  - In this context, the most commonly used choice is $\exp(z)$, which is bijective
  - In this case, the (raw) model outputs are called **logits**

A probabilistic model for discrete labels

if there are K possible labels, then $f_\theta(\mathbf{x})$ is a vector of length K

we represent the final probabilities using the **softmax** function:

$$\text{softmax}(f_\theta(x))_c = \frac{\exp\{f_\theta(x)_c\}}{\sum_{i=1}^{K} \exp\{f_\theta(x)_i\}} = p_\theta(y=c \mid x)$$

(aside: sometimes, $f_\theta(\mathbf{x})$ is used to denote $\arg\max_y p_\theta(y \mid \mathbf{x})$, but I won't do that)

Some examples of the softmax function

supposing $K = 4$, let's work through some examples

softmax([0, 0, 0, 0]) = [0.25, 0.25, 0.25, 0.25]

softmax([-100, -100, -100, -100]) = [0.25, 0.25, 0.25, 0.25]

softmax([0, 0, 100, 0]) ≈ [0, 0, 1, 0]

softmax([-100, -100, 0, -100]) ≈ [0, 0, 1, 0]

softmax([2, 1, 0, 0]) = [0.6103, 0.2245, 0.0826, 0.0826]

Recap

- So far, we have defined what our probabilistic model is going to look like
  - In the case of discrete labels, it will output K numbers that will be exponentiated and normalized to form an output distribution
- What else do we need?
  - How do we know whether or not the model parameters are good?
  - How do we find good parameters?

The machine learning method

(or, at least, the deep learning method)

1. Define your **model** — which neural network, what does it output, …
2. Define your **loss function** — which parameters are good vs. bad?
3. Define your **optimizer** — how do we find good parameters?
4. Run it on a big GPU

The machine learning method

(or, at least, the deep learning method)

1. Define your **model** — which neural network, what does it output, …
2. Define your **loss function** — which parameters are good vs. bad?
3. Define your **optimizer** — how do we find good parameters?
4. Run it on a big GPU

What loss function should we use?

- In deciding on a loss function, we have a few desiderata:
  - If our parameters perfectly explain the data, we should incur minimal loss
  - The loss should be “easy” to optimize
  - We don’t want to have to engineer new loss functions for every problem
- We will satisfy these desiderata by leveraging the most widely used tool in statistical inference — **maximum likelihood estimation (MLE)**

The maximum likelihood principle

$p_\theta(x, y) = p(x)\, p_\theta(y \mid x)$

given data $\mathscr{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$

assume a set (family) of distributions on $(\mathbf{x}, y)$ $\{ p_\theta : \theta \in \Theta \}$

assume some p_{data} generated $\mathcal{D}$

the parameters θ dictate the conditional distribution of y given $\mathbf{x}$

the objective/definition: "recover $\hat{\theta}$" (sort of)

$$\theta_{MLE} = \underset{\theta \in \Theta}{\arg\max}\; p(\mathcal{D} \mid \theta) = \underset{\theta \in \Theta}{\arg\max} \prod_{i=1}^{N} p(x_i)\, p_\theta(y_i \mid x_i)$$

From MLE to a loss function

we are given $\mathscr{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$

our goal is to find $\arg\max_{\theta \in \Theta} \prod_{i=1}^{N} p(x_i)\, p_\theta(y_i \mid x_i)$

working with a product of terms is tricky and messy…

idea: take the $\log$ instead! this leads to the **negative log likelihood** loss function:

$$\arg\max_{\theta \in \Theta} \sum_{i=1}^{N} \log p(x_i) + \log p_\theta(y_i \mid x_i) = \arg\max_{\theta \in \Theta} \sum_{i=1}^{N} \log p_\theta(y_i \mid x_i)$$

constant w.r.t. θ

$$= \arg\min_{\theta \in \Theta} \sum_{i=1}^{N} \underbrace{-\log p_\theta(y_i \mid x_i)}_{\ell(\theta;\, x_i, y_i)}$$

(usually, we divide by N to work with *average loss* rather than summed loss)

The negative log likelihood loss function

this loss is oftentimes called the **cross-entropy** loss — what is cross-entropy?

$$H(p, q) = -\sum_x p(x) \log q(x) = \mathbb{E}_p[-\log q(X)]$$

let's plug in p_{data} (the true data distribution) for p and some p_θ for q:

$$H(p_{\text{data}}, p_\theta) = \mathbb{E}_{p_{\text{data}}}[-\log p_\theta(X, Y)]$$

constant w.r.t. θ

$$= \mathbb{E}_{p_{\text{data}}}[-\log p(X) - \log p_\theta(Y|X)] \approx \sum_{i=1}^{N} -\log p(x_i) - \log p_\theta(y_i|x_i)$$

maximizing log likelihood is approximately equivalent to minimizing cross-entropy!

Should we use the negative log likelihood loss?

Revisiting our desiderata

- If our parameters perfectly explain the data, we should incur minimal loss
  - Given sufficient data, the log likelihood is maximized by the “true” parameters, if our model is able to represent the underlying data distribution
  - This is related to an attractive property of MLE called *consistency*
- The loss should be “easy” to optimize — more on this next
- We don’t want to have to engineer new loss functions for every problem
  - Many commonly used loss functions, such as **squared error** for regression, can be derived/motivated from log likelihood for different modeling assumptions

The machine learning method

(or, at least, the deep learning method)

1. Define your **model** — which neural network, what does it output, …
2. Define your **loss function** — which parameters are good vs. bad?
3. Define your **optimizer** — how do we find good parameters?
4. Run it on a big GPU

What optimizer should we use?

- Deep learning relies on **iterative optimization** to find good parameters
  - Starting from an initial "guess", continually refine that guess until we are satisfied with our final answer
- By far the most commonly used set of iterative optimization techniques in deep learning is (first order) gradient based optimization and variants thereof
  - Basically, move the parameters in the direction of the *negative gradient* of the average loss: $\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$

Synergy between loss function and optimizer

- The gradient tells us how the loss value changes for small parameter changes
  - We decrease the loss if we move (with a small enough α) along the direction of the negative gradient (basically, go “opposite the slope” in each dimension)
- This motivates choosing the loss function and model carefully, such that the loss function is *differentiable* with respect to the model parameters
  - The negative log likelihood fulfills this for many reasonable problem setups
  - What loss function would not be differentiable?
  - For example, the **0-1 loss function**: 1 if the model is correct, 0 otherwise

A small example: logistic regression

The “linear neural network”, if we’re being weird

- Given $\mathbf{x} \in \mathbb{R}^d$, define $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$, where θ is a $d \times K$ matrix
- Then, for class $c \in \{1, \ldots, K\}$, we have $p_\theta(y = c \mid \mathbf{x}) = \text{softmax}(f_\theta(\mathbf{x}))_c$
- Loss function: $\ell(\theta; \mathbf{x}, y) = -\log p_\theta(y \mid \mathbf{x})$
- Optimization: $\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$

Let's work this out for binary classification

(in general, θ <u>could</u> be a $d \times (K-1)$ matrix)

if we have $K = 2$, then θ can actually just be a $d \times 1$ vector! why?

if we know $p(y=0 \mid x)$, we know $p(y=1 \mid x)$

let $f_\theta(x) = \theta^T x \in \mathbb{R}$ be the logit for class 1;

let the class 0 logit be fixed to 1.0

so $p_\theta(y=1 \mid x) = \text{softmax}([1.0, f_\theta(x)])_1 = \dfrac{\exp \theta^T x}{\exp \theta^T x + 1}$

$$\ell(\theta; x_i, y_i) = (1-y_i)(\log\{\exp \theta^T x_i + 1\}) - y_i(\theta^T x - \log\{\exp \theta^T x_i + 1\})$$

exercise: $\nabla_\theta \ell(\theta; x_i, y_i) = \left(\dfrac{\exp \theta^T x}{\exp \theta^T x + 1} - y \right) x$

The machine learning method

(or, at least, the deep learning method)

1. Define your **model** — which neural network, what does it output, …
2. Define your **loss function** — which parameters are good vs. bad?
3. Define your **optimizer** — how do we find good parameters?
4. Run it on a big GPU