Lecture 2: Machine Learning Review - 1

COMPSCI/DATA 182: Deep Learning

09/03/2024

From Lecture 1

- *“Is overfitting less of any issue with Deep Learning models ? ....And why so ?”*

- Is Deep Learning *always* better ?
  - The “**no-free-lunch**” theorem
    - (in machine learning)

THERE'S NO SUCH THING AS A FREE LUNCH,
THERE'S NO SUCH THING AS A FREE LUNCH
imgflip.com

D. H. Wolpert, "The Lack of A Priori Distinctions Between Learning Algorithms," in Neural Computation, vol. 8, no. 7, pp. 1341-1390, Oct. 1996, doi: 10.1162/neco.1996.8.7.1341.

Homework Assignments

- The goals of the assignment:
  - understand **Neural Networks** and how they are arranged in layered architectures
  - understand and be able to implement (vectorized) **backpropagation**
  - implement various **update rules** used to optimize Neural Networks
  - implement **batch normalization** for training deep networks
  - implement **dropout** to regularize networks
  - effectively **cross-validate** and find the best hyperparameters for Neural Network architecture
  - understand the architecture of **Convolutional Neural Networks**
  - gain an understanding of how a modern deep learning library (PyTorch) works and gain practical experience using it to train models.
- You will be provided with pretty much ALL of the baseline code
- Assignment tasks will be specific code additions, validation exercises etc.

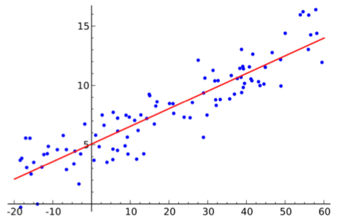
Today ….

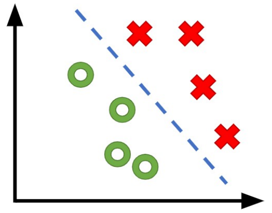
- In this lecture and the next lecture, we will go over concepts at the core of machine learning as a whole
    - We will focus on concepts that are the most relevant to deep learning
- Much of this will be review if you have already taken a machine learning course
- Today, we will focus on the supervised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

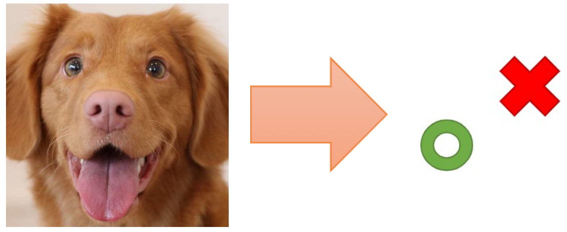
Different classes of learning problems

(non exhaustive)

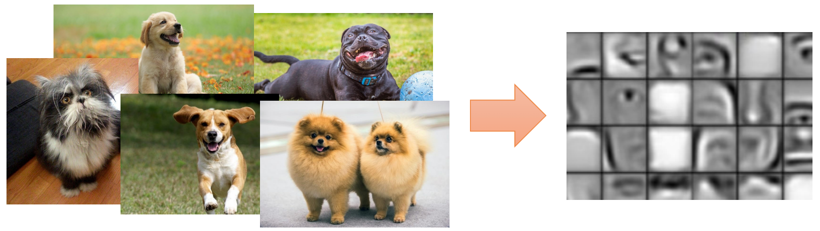
Supervised learning



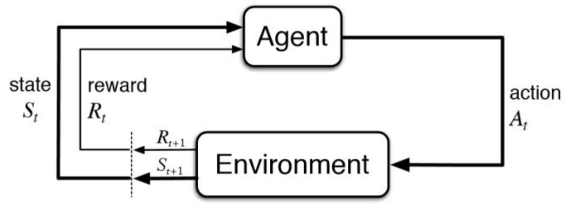




Unsupervised learning

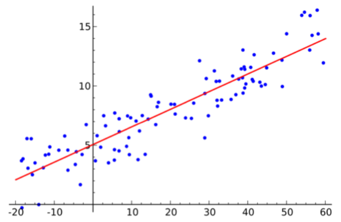


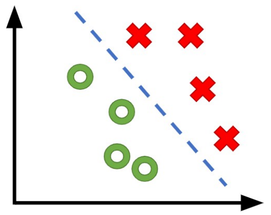
Reinforcement learning

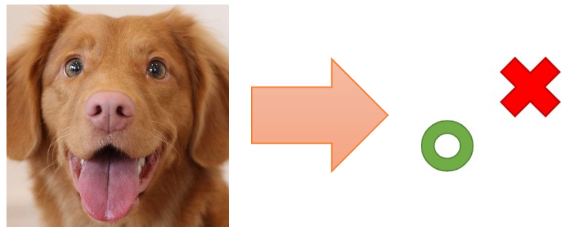


Supervised learning ….

Supervised Learning







- In supervised learning, we are given a dataset ! $= \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$
- 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)
- Simple basic principles

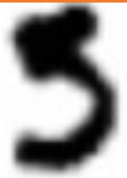
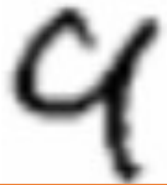
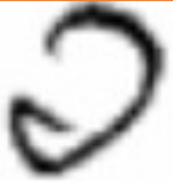
Examples of supervised learning problems

(that deep learning has done really well on !)

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 $\boldsymbol{y}$?

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 positive
  - In this context, the most commonly used choice is e^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:

$$\mathrm{softmax}(f_\theta(x))_c = \frac{e^{f_\theta(x)_c}}{\sum_{k=0}^{K-1} e^{f_\theta(x)_k}}$$

$$= P_\theta\,(y = c \mid x)$$

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, or rather deep learning recipe

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

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)**

Maximum Likelihood

- WHAT IS: a FRUIT TYPE & mostly ROUND & SMALL (not very though) & SOFT & colored ORANGE/YELLOW/GREEN ……. ?

- **Most Likely** : Citrus fruits ! versus ….

The maximum likelihood principle & estimation (MLE)

$$\begin{tikzcd} p_\theta(x,y) = p(x)\,p_\theta(y|x) \arrow[d] \\ \{p_\theta : \theta \in \Theta\} \end{tikzcd}$$

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

assume a set (family) of distributions on $(\mathbf{x}, y)$

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

the parameters dictate the conditional distribution of given

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

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

From MLE to a Loss Function

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

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

working with a product of terms is tricky and messy…

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

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

constant w.r.t. θ

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

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

Why are we *minimizing* something, versus just maximizing likelihood ?

- Congruence with Errors we have seen: in statistical analysis and traditional (feature-driven) machine learning
- Congruence with optimization processes
  - Such as gradient descent etc.

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 and some p_Θ for :

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

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

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

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

$$\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$$

I've cut slices so thin,
I couldn't even see them.
How did you know you cut it?
Well, I guess I just assumed.

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) 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**: 0 if the model is correct, 1 otherwise

Simple example: Logistic Regression
(aka the Linear Neural Network)

- Given $\mathbf{x}$ e R^d, define $f_\Theta(\mathbf{x}) = \Theta^{\mathrm{T}}\mathbf{x}$, where is a $d \times K$ matrix
- Then, for class c e $\{0, \ldots, K - 1\}$, we have $p_\Theta(y = c \,|\, \mathbf{x}) = \mathrm{softmax}(f_\Theta(\mathbf{x}))_c$
- Loss function: $l(\Theta; \mathbf{x}, y) = -\log p_\Theta(y \,|\, \mathbf{x})$
- Optimization: $$\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$$

For Binary Classification

$$\text{softmax}(f_\theta(x))_c = \frac{e^{f_\theta(x)_c}}{\sum_{k=0}^{K-1} e^{f_\theta(x)_k}}$$

$$z = \theta^T x$$

$$p(y=1|x) = \frac{e^{\theta^T x}}{e^{\theta^T x}+1}$$

$$p(y=0|x) = \frac{1}{e^{\theta^T x}+1}$$

$$\ell(\theta; x, y) = -\left[y \log\left(\frac{e^{\theta^T x}}{e^{\theta^T x}+1}\right) + (1-y)\log\left(\frac{1}{e^{\theta^T x}+1}\right)\right]$$

$$\theta \leftarrow \theta - \alpha \nabla_\theta \text{Loss}(\theta)$$

Note: The recommended reading *"Binary classification and logistic regression"* (by L. Chen) discusses this well and in detail.

Exercise: $$\nabla_\theta \ell(\theta; x, y) = \left[\frac{e^{\theta^T x}}{1+e^{\theta^T x}} - y\right] x$$