Overview

CS 446

2021-01-26 17:39:03 -0600 (ff9ea4d)
Plan for today

- ML examples.
- ML math/coding background.
- ML setting and meta-algorithms.
- First technique: linear regression.
Machine learning: improving computational mechanisms by fitting them to data.
Application 1: image classification

- Birdwatcher takes photos of birds, organizes by species.
- **Goal**: automatically recognize bird species in new photos.

![Indigo bunting](image)

- **Why ML**: variation in lighting, occlusions, morphology.
Application 2: recommender system

- Netflix users watch movies and provide ratings.
- **Goal**: predict user's rating of unwatched movie.
- (**Real goal**: keep users paying customers.)
- (**Real effect**: reinforce stereotypes found in the data?)

![Diagram](Image credit: Koren, Bell, and Volinsky, 2009.)

- **Why ML**: easily adapt to and leverage movie attributes, viewer attributes, viewer relationships, etc.
Application 3: machine translation

- Linguists provide translations of all English language books into French, sentence-by-sentence.
- **Goal**: translate any English sentence into French.

![Translation example](image)

**Note**: the text-to-speech is via ML (recurrent network transformer).

- **Why ML**? Not only avoid hard-coding many rules, but also capture idiom and other nuances.
Application 4: chess

- Chess enthusiasts construct a large corpus of chess games. (Or: start with nothing, play random games!)
- **Goal**: Win chess games.

**Why ML?** Avoid hard-coding evaluation; magically interpolate between observed positions, as humans do.
Math/coding background

▶ Linear algebra (e.g., null spaces; eigendecomposition; SVD).
▶ Basic probability and statistics (e.g., variance of a random variable).
▶ Multivariable calculus (e.g., gradient of $\|A\mathbf{w} - \mathbf{b}\|^2_w$).
▶ Basic proof writing (e.g., prove $A^TA$ is positive semi-definite).

Coding.
▶ Python3. It's slow, but often computation will be inside fast libraries.
▶ Numpy, an easy-to-use numeric library.
▶ Pytorch, a numeric library with gpu support, auto-differentiation, and deep learning helpers.

My opinion. Pytorch is one of the nicest libraries I've ever used, for anything. I use it for much more than deep learning.
Math.

- Linear algebra (e.g., null spaces; eigendecomposition; SVD...).
- Basic probability and statistics (e.g., variance of a random variable).
- Multivariable calculus (e.g., gradient of $\|Aw - b\|^2$ wrt $w$).
- Basic proof writing (e.g., prove $A^T A$ is positive semi-definite).

Coding.

- python3. It's slow, but often computation will be inside fast libraries.
- numpy, an easy-to-use numeric library.
- pytorch, a numeric library with gpu support, auto-differentiation, and deep learning helpers.

My opinion. pytorch is one of the nicest libraries I've ever used, for anything. I use it for much more than deep learning.
Math.

▶ Linear algebra (e.g., null spaces; eigendecomposition; SVD...).
▶ Basic probability and statistics (e.g., variance of a random variable).
▶ Multivariable calculus (e.g., gradient of $\|Aw - b\|^2$ wrt $w$).
▶ Basic proof writing (e.g., prove $A^TA$ is positive semi-definite).

Coding.

▶ python3. It’s slow, but often computation will be inside fast libraries.
▶ numpy, an easy-to-use numeric library.
▶ pytorch, a numeric library with gpu support, auto-differentiation, and deep learning helpers.

My opinion. pytorch is one of the nicest libraries I’ve ever used, for anything. I use it for much more than deep learning.
```python
>>> import numpy
>>> import torch

>>> 3 / 2
1.5
>>> 3 // 2
1

>>> A = torch.randn(5,5)
>>> b = torch.randn(5,1)
>>> x = torch.gels(b, A)[0]
>>> (A @ x - b).norm()
tensor(3.7985e-06)

>>> x = numpy.linalg.lstsq(A, b)[0]
>>> (A @ torch.tensor(x) - b).norm()
tensor(1.1999e-06)
```
pytorch on gpu is easy

```python
>>> import torch

>>> A = torch.randn(5, 5)
>>> b = torch.randn(5)
>>> (A @ b).norm()
tensor(4.7746)

>>> device = torch.device("cuda:0")
>>> (A.to(device) @ b.to(device)).norm()
tensor(4.7746, device='cuda:0')

>>> (A.to(device) @ b).norm()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Expected object of type torch.cuda.FloatTensor but found type torch.FloatTensor for argument #2 'vec'
```

Note. Homeworks will be graded in a gpu-free container!
Homework 0

- Homework 0 is posted on the class webpage.
- It is a sanity check of basic (math and coding) background.
- It is due Tuesday, February 2.
- It has two gradescope components: 
  hw0 is multiple choice, hw0code is coding.
Main setting: supervised learning

Training data: labeled examples \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) where

▶ each input \(x_i\) is a machine-readable description of an instance (e.g., image, sentence), and
▶ each corresponding label \(y_i\) is an annotation relevant to the task—typically not easy to automatically obtain.

Goal: learn a predictor (a function) \(\hat{f}\) from labeled examples, that accurately "predicts" the labels of new (previously unseen) inputs.
Main setting: supervised learning

**Training data:** labeled examples

\[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\]

where

- each input \(x_i\) is a machine-readable description of an instance (e.g., image, sentence), and

- each corresponding label \(y_i\) is an annotation relevant to the task—typically not easy to automatically obtain.
Main setting: supervised learning

**Training data:** labeled examples

\[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\]

where

- each input \(x_i\) is a machine-readable description of an instance (e.g., image, sentence), and
- each corresponding label \(y_i\) is an annotation relevant to the task—typically not easy to automatically obtain.

**Goal:** learn a predictor (a function) \(\hat{f}\) from labeled examples, that accurately “predicts” the labels of new (previously unseen) inputs.
Other settings

- **Unsupervised learning**: find structure in some examples \((x_i)_{i=1}^n\) (there are no labels!).
- **Time series modeling**: the label of \(x_i\) also depends on \((x_{i-1}, x_{i-2}, \ldots)\).
- **Reinforcement learning**: the machine learning method makes decisions, not just predictions: the outputs affect future inputs. (E.g., driving a car).
What are the difficulties?

Consider easy setting: learn $\hat{f} : X \rightarrow Y$ from examples $((x_i, y_i))_{i=1}^n$.

Not easy.
What are the difficulties?

Consider easy setting: learn $\hat{f} : X \rightarrow Y$ from examples $(x_i, y_i)_{i=1}^{n}$.

Not easy.

- How to clean/improve/augment data?
Consider easy setting: learn \( \hat{f} : X \rightarrow Y \) from examples \(((x_i, y_i))_{i=1}^{n}\).

Not easy.

- How to clean/improve/augment data?
- How to choose structure/model for \( \hat{f} \)?
What are the difficulties?

Consider easy setting: learn $\hat{f} : X \rightarrow Y$ from examples $((x_i, y_i))_{i=1}^n$.

Not easy.

- How to clean/improve/augment data?
- How to choose structure/model for $\hat{f}$?
- How to algorithmically fit $\hat{f}$ to data?
What are the difficulties?

Consider easy setting: learn \( \hat{f} : X \rightarrow Y \) from examples \( ((x_i, y_i))_{i=1}^{n} \).

Not easy.

- How to clean/improve/augment data?
- How to choose structure/model for \( \hat{f} \)?
- How to algorithmically fit \( \hat{f} \) to data?
- How to ensure \( \hat{f} \) does not overfit, meaning it fits \( ((x_i, y_i))_{i=1}^{n} \) well, but is useless on future data?
“pytorch meta-algorithm”

1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.
“pytorch meta-algorithm”

1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

Is that all of ML?
1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

Is that all of ML?
No, but these days it’s much of it!
Model choice?

Should we use 1-nearest-neighbor or a 2-layer ReLU network?
Model/algorithm choice?

- Linear or Logistic Regression: 83.7%
- Decision Trees or Random Forests: 78.1%
- Gradient Boosting Machines (xgboost, lightgbm, etc.): 61.4%
- Convolutional Neural Networks: 43.2%
- Bayesian Approaches: 31.4%
- Recurrent Neural Networks: 30.2%
- Neural Networks (MLPs, etc.): 28.2%
- Transformer Networks (BERT, gpt-3, etc.): 14.8%
- Generative Adversarial Networks: 7.3%
- Evolutionary Approaches: 6.5%
- Other: 4.5%
- None: 1.7%

(From kaggle 2020 survey.)
Linear regression — basic setup

1. Start from training data \(((x_i, y_i))_{i=1}^n\), with \(x_i \in \mathbb{R}^d\) and \(y_i \in \mathbb{R}\).

2. Model is a linear predictor: pick \(w \in \mathbb{R}^d\) with

\[ x_i \mapsto w^T x_i \approx y_i. \]

3. Loss function is squared loss (standard regression loss):

\[ \ell(w^T x_i, y_i) = \frac{1}{2}(w^T x_i - y_i)^2. \]

We will minimize the empirical risk (average loss over training examples):

\[ \hat{R}(w) = \frac{1}{n} \sum_{i=1}^n \ell(w^T x_i, y_i) = \frac{1}{2n} \sum_{i=1}^n (w^T x_i - y_i)^2. \]

Convenient form using matrices:

\[ \hat{R}(w) = \frac{1}{2n} \|Xw - y\|^2 \quad \text{where} \quad X := \begin{bmatrix} \leftarrow x_1^T \rightarrow \\ \vdots \\ \leftarrow x_n^T \rightarrow \end{bmatrix} \in \mathbb{R}^{n \times d}. \]
3. Minimize the empirical risk

$$\hat{R}(w) = \frac{1}{2n} \|Xw - y\|^2$$

where $X := \begin{bmatrix} \leftarrow x_1^\top & \rightarrow \\ \vdots \\ \leftarrow x_n^\top & \rightarrow \end{bmatrix} \in \mathbb{R}^{n \times d}$.

4. Basic method: gradient descent. Set $w_0 = 0$, and thereafter

$$w_{i+1} := w_i - \eta \nabla \hat{R}(w_i) = w_i - \frac{\eta}{n} X^\top (Xw_i - y),$$

where $\eta$ is a learning rate (step size).

```python
w = torch.zeros(d)
for _ in range(niters):
    w = w - X.T @ (X @ w - y) / 100 / n
```
Summary for today

- ML examples.
- ML math/coding background.
- ML setting and meta-algorithms.
- First technique: linear regression.