CSE 559A: Computer Vision

Fall 2020: T-R: 11:30-12:50pm @ Zoom

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Course Staff: Adith Boloor, Patrick Williams

http://www.cse.wustl.edu/~ayan/courses/cse559a/

Nov 17, 2020
Why not just fit the more complex model?

Too simple

Just Right

Too complex

Quadratic Fit
While we train on empirical loss,

\[ \frac{1}{T} \sum_t L(y_t, f(x_t)) \]

we care about the actual expected loss:

\[ \int_X \int_Y L(y, f(x)) \, P_{XY(x,y)} \, dy \, dx \]

Why?
While we train on empirical loss,

\[ \frac{1}{T} \sum_t L(y_t, f(x_t)) \]

we care about the actual expected loss:

\[ \int \int L(y, f(x)) \, P_{XY(x,y)} \, dy \, dx \]

Why? Because we don’t want to explain the training set. We want to do well on new inputs. We want to “generalize” from the training set.

This is why a more complex function that exactly fits the training set is bad, when it “generalizes” poorly.
Error = Bayes Error + Approximation Error + Estimation Error

- **Bayes Error**: This is the error due to the uncertainty in $p(y|x)$. This is the error you would have even if you knew the exact distribution and could craft a function $f$ with infinite complexity.

  \[
  \text{Bayes Error} = \int \left( \int \min_{\hat{y}} L(y, \hat{y}) P_{XY}(x, y) dy \right) dx
  \]

- **Approximation Error**: This is the error due to the limited capacity of our hypothesis space $\mathcal{H}$. It is the error of the true best function $f \in \mathcal{H}$, minus the Bayes error, assuming we had perfect knowledge of $P_{XY}$. Also called the “Bias”\(^2\).

- **Estimation Error**: This is the remaining component of error, caused by the fact that we don’t know the true $P_{XY}$, but only have a limited number of samples. This depends on the size of our training set (and also, how well we are able to do the minimization). Called “variance”.


MACHINE LEARNING

Error = Bayes Error + Approximation Error + Estimation Error

Bias-Variance Tradeoff

Choosing a simple function class: higher approximation error, lower estimation error.

Choosing a complex function class: lower approximation error, higher estimation error.

How do I decrease Bayes Error?
Error = Bayes Error + Approximation Error + Estimation Error

**Bias-Variance Tradeoff**

Choosing a simple function class: higher approximation error, lower estimation error.

Choosing a complex function class: lower approximation error, higher estimation error.

How do I decrease Bayes Error? By getting better inputs!
MACHINE LEARNING

Overfitting

• Definitions of complexity of a function or hypothesis space $\mathcal{H}$: VC-dimension, Rademacher complexity
• Try to capture that one function or function space provides a “simpler” explanation than another
• Useful as intuition. But often don’t “work” for very complex functions and tasks.
• But the idea is:
  ■ Given two functions with the same error, you want to choose one that’s simpler.
  ■ You never want to consider a class of functions that can fit ‘random’ $T$ pairs of $(x, y)$, where $T$ is the size of your training set.
• Choose hypothesis space based on size of training set.
• Add a “regularizer” (for example, a squared penalty on higher order polynomial coefficients).

Public Service Announcement: Any regularizer is biased by what you think is “simple”. There is no universal definition of simple.
MACHINE LEARNING

• Obtain a function \( f : \mathcal{X} \rightarrow \mathcal{Y} \) from data
  ■ Maps inputs from domain \( \mathcal{X} \) to outputs from domain \( \mathcal{Y} \)

• Components
  ■ Training set of pairs \((x_i, y_i)\)
  ■ Loss function \( L(y, \hat{y}) \)
  ■ Hypothesis Space \( \mathcal{H} \) to search over for \( f \)

\[
f = \arg \min_{f \in \mathcal{H}} \sum_i L(y_i, f(x_i))
\]

• Basically, algorithm design by trial and error (on training set)
• A better way of solving problems when the problems are **ill-posed**
• Need to watch out for over-fitting the training set
• The only guarantee that you are not overfitting is empirical verification!
Overfitting: Good ML “Hygiene”

Remember, you can overfit not just the parameters, but your design choices!

For any given task:

- Have train, dev, val, and test set.
- Train your estimators on the train set.
- Choose hyperparameters based on dev set.
  - Function class
  - Regularization weight
  - Number of iterations to train, etc.
- Keep periodically checking to see if it generalizes to val set.
- Look at the test set only at the “end” of the project.
Classification

Consider the case when $y$ is binary, i.e., $\mathcal{Y} = \{0, 1\}$.

How do you define the loss function then?

- Ideally, $L(y, \hat{y})$ is 0 if they are equal, 1 otherwise.

But don’t know how to solve that. What if we solved by regression?

$$w = \arg \min_w \frac{1}{T} \sum_t (y_t - w^T \tilde{x}_t)^2$$

And at test time, we can output $y = 1$ if $w^T \tilde{x} > 0.5$ and 0 otherwise.

The problem is the loss function will penalize $w^T \tilde{x}_t > 1$ when $y_t = 1$. While at test time, this would give us exactly the right answer!
Logistic regression

- Learn a function \( f(x) = P(y = 1) \) which regresses to the probability \( y \) is 1.
- We have to choose \( f \) such that the domain of \( f(x) \) lies between \([0, 1]\).

\[
f(x; w) = \sigma (w^T \tilde{x}) , \quad \sigma(p) = \frac{\exp(p)}{1 + \exp(p)}
\]

- This ensures that the output of \( f \) is between \([0, 1]\)
- \( w^T \tilde{x} \) can be interpreted as the log of the odds, or log of ratio between \( P(y = 1) \) to \( P(y = 0) \)
- \( \tilde{x} \) is some augmented “feature vector” derived from \( x \).
  - “Linear Classifier” if \( \tilde{x} = [x^T ; 1]^T \) (log-odds are linear)
  - Could be polynomial \( \tilde{x} = [1, x, x^2, x^3] \)
  - Or other arbitrary non-linear functions of \( x \)
  - Can apply even when \( x \) is non-numeric, as long as \( \tilde{x} \) is numeric.
Logistic Regression

For Binary Classification: \( \mathcal{X} \rightarrow [0, 1] \)

\[
f(x; w) = \sigma(w^T \tilde{x}) = \frac{\exp(w^T \tilde{x})}{1 + \exp(w^T \tilde{x})}
\]

- To classify, \( y = 1 \) if \( P(y = 1) > 0.5 \) or 0 otherwise.
- That is, \( y = 1 \) if \( w^T \tilde{x} > 0 \) and 0 otherwise.

- Note: Classifier is linear in chosen encoding \( \tilde{x} \).
- \( w^T \tilde{x} <> 0 \) defines a “separating hyperplane” between positive and negative part of the space of \( \tilde{x} \).
Decision "boundary" Hyperplane
Will correspond to a non-linear boundary in the original x space.

Decision "boundary" Hyperplane.
Logistic regression

\[ P(y = 1) = f(x) = \sigma (w^T \tilde{x}) \]

What about the loss?

Cross-Entropy Loss

If true \( y \) is 1, we want \( f(x) \) to be high, and if it is 0, we want it to be low.

\[
L(y, f(x)) = - \begin{cases} 
\log P(y = 1) = \log f(x) & \text{if } y = 1 \\
\log P(y = 1) = \log 1 - f(x) & \text{if } y = 0 
\end{cases}
\]

\[
L(y, f(x)) = -y \log f(x) - (1 - y) \log(1 - f(x))
\]

There’s a minus because this is the loss.

Minimizing \( \sum_t L(y_t, f(x_t)) \) can be viewed as maximizing the sum of the log-probabilities, or the product of the probabilities of the labels \( y_t \) under our predicted distribution.

Promotes a high probability for the correct label > uniform distribution (low confidence) over both labels > high probability for incorrect label.

But now, how do we minimize this function in terms of \( w \)? No longer least-squares.
Logistic Regression

\[ f(x; w) = \sigma(w^T \tilde{x}) = \frac{\exp(w^T \tilde{x})}{1 + \exp(w^T \tilde{x})} \]

- Cross-entropy / Negative Likelihood Loss

\[ L(y, f(x; w)) = -y \log f(x; w) - (1 - y) \log(1 - f(x; w)) \]

\[ f(x; w) = \frac{1}{1 + \exp(-w^T \tilde{x})} \]

\[ 1 - f(x; w) = \frac{1}{1 + \exp(w^T \tilde{x})} \]
Logistic Regression

\[ f(x; w) = \sigma(w^T \tilde{x}) = \frac{\exp(w^T \tilde{x})}{1 + \exp(w^T \tilde{x})} \]

- Cross-entropy / Negative Likelihood Loss

\[ L(y, f(x; w)) = y \log[1 + \exp(-w^T \tilde{x})] + (1 - y) \log[1 + \exp(w^T \tilde{x})] \]

- Putting it all together, given a training set of \( \{(x_t, y_t)\} \):

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log[1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log[1 + \exp(w^T \tilde{x}_t)] \]
Logistic Regression

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)] \]

- You can show that this loss is a convex function of \( w \)
  (compute the Hessian matrix and show that it’s eigenvalues are non-negative)
- So it has a single global minimum.

But how do we find it?
**GRADIENT DESCENT**

### Logistic Regression

\[
w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)]
\]

### More General Form

\[
w = \arg \min_w C(w) \quad C(w) = \frac{1}{T} \sum_t C_t(w)
\]

### Iterative algorithm

- Given a current estimate of \( w \), approximate \( C(w) \) as a linear function of \( w \)
  - \( C(w) = \alpha^T w \)
- Do this fit by computing the gradient of \( C(w) \) wrt \( w \)
  - \( \alpha = \nabla_w C(w) \) (would be true if \( C(w) = \alpha^T w \))

Think of \([C(w), w]\) as the co-ordinates on a plane. Which direction to move in \( w \)-space to reduce \( C(w) \) ?

\(-\alpha\)
GRADIENT DESCENT

\[ w = \arg \min_w C(w) \quad C(w) = \frac{1}{T} \sum_t C_t(w) \]

- Begin with initial guess \( w_0 \)
- At each iteration \( i \):
  - \[ w_{i+1} \leftarrow w_i - \gamma \nabla_w C(w_i) \]
- At each iteration, we update the parameters \( w \) by “moving”, in \( w \)-space, in the opposite direction of the gradient (at that point \( w_i \)).
- \( \gamma \) is the step-size. When running optimization for training, often called the “learning rate”.
- In some cases, \( \gamma \) can be set by doing a line-search
  - Check values of \( C(w - \gamma \nabla_w) \) and pick \( \gamma \) which minimizes the cost
- In other cases, we choose a fixed value of \( \gamma \) (or change it in some pre-determined schedule per iteration)
  - Then, we are moving by a distance that is proportional to magnitude of the gradient
GRADIENT DESCENT

C

W

Gradient
GRADIENT DESCENT
GRADIENT DESCENT
GRADIENT DESCENT
GRADIENT DESCENT
GRADIENT DESCENT
If you select optimal step size by doing a “line search” for $\gamma$, can prove that gradient-descent will converge.

If function is convex, converge to unique global minimum.

Second order variants that consider the Hessian matrix: Newton & Quasi-Newton Methods
- Gauss-Newton, Levenberg-Marquardt, …

But simple gradient descent suffices / our only choice when:

Function isn’t convex.

Can’t afford to do line search.

So many parameters that can’t compute Hessian.

Also, no theoretical guarantees.

Theory still catching up. Meanwhile, we’ll try to understand the “behavior” of the gradients.