Lecture 03: ML Review (2)

Data C182. Week 2, Thursday Sept 5th 2024.

Speaker: Eric Kim

Announcements: Course platforms

- **Ed**: We will make all announcements via the course Ed [link]. **It's your responsibility to actively monitor Ed so that you don't miss anything important!**
  - Ex: "Week 2 Announcements" [link]
- **Gradescope** [link]: this is where we will release homework, where you will submit, and where the assignment autograder lives.
- **Course website** [link]: this is where things like: lecture slides, schedule, office hour times, syllabus/policy lives.
- **bCourses** [link]: Mainly only used for hosting lecture recordings, in "Media Gallery" [link]

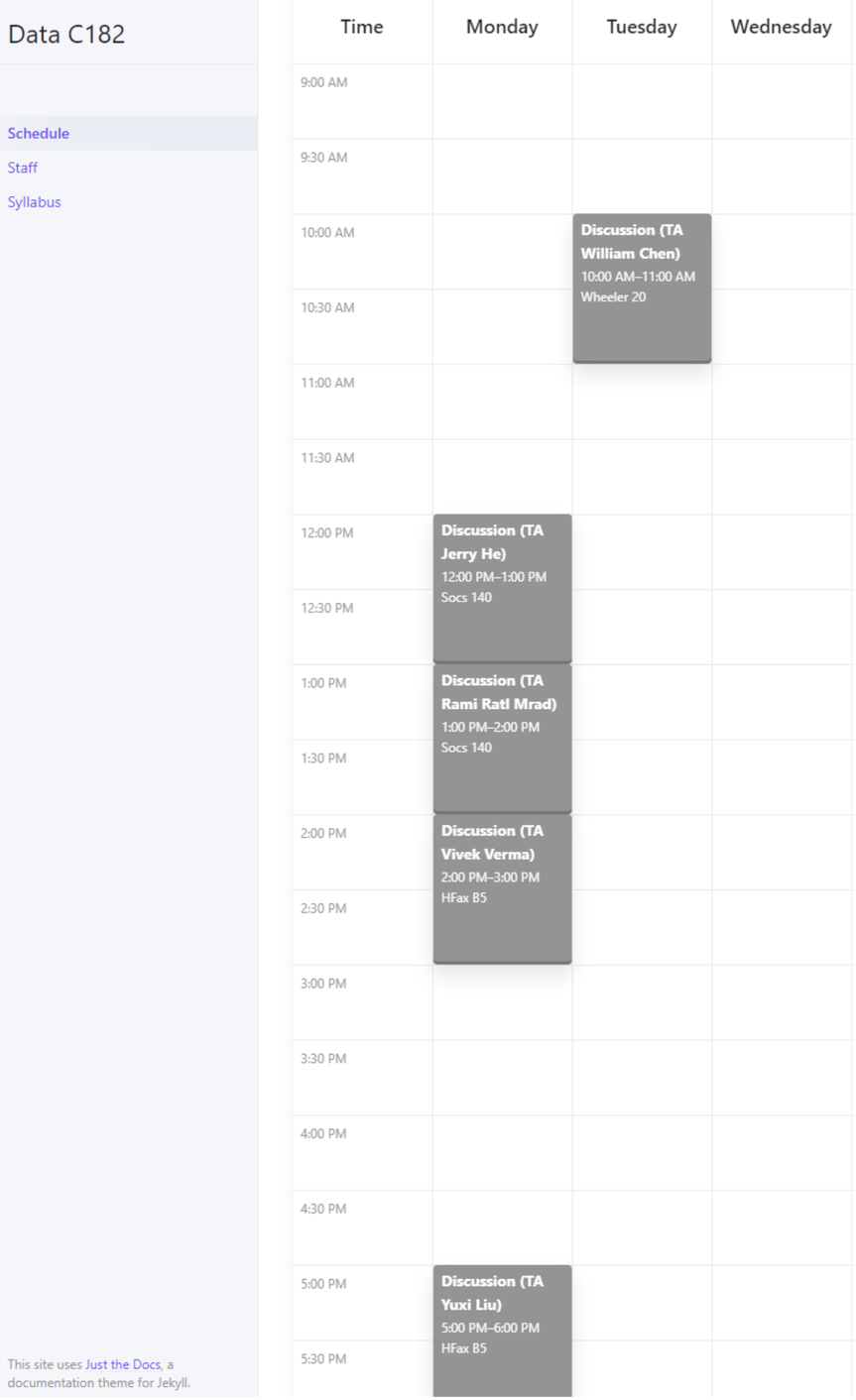
Credit: Data C182 Fall 2024 course logo designed by Sean Liu (2024): https://www.instagram.com/seanyliu/

HW01

- HW01: Released yesterday! **Due Tues Oct 1st at 11:59 PM PST.**
  - Tip: start early! This is a fairly hefty assignment.
  - Please ask questions on Ed! Ed is a great resource to use.
    - **For public posts**, please limit the amount of code you submit (eg don't submit code that others could use in their solution, "within reason/taste"). **For private posts**, you can post all the code that you want.
- Collaboration policy [link]: you can collaborate with others at a high level only: **all coding must be done on your own.**
- **Regarding issues around dependency installations:** we're working on it. We'll update on Ed when we've found a better way forward, particularly for Windows/MacOS students.

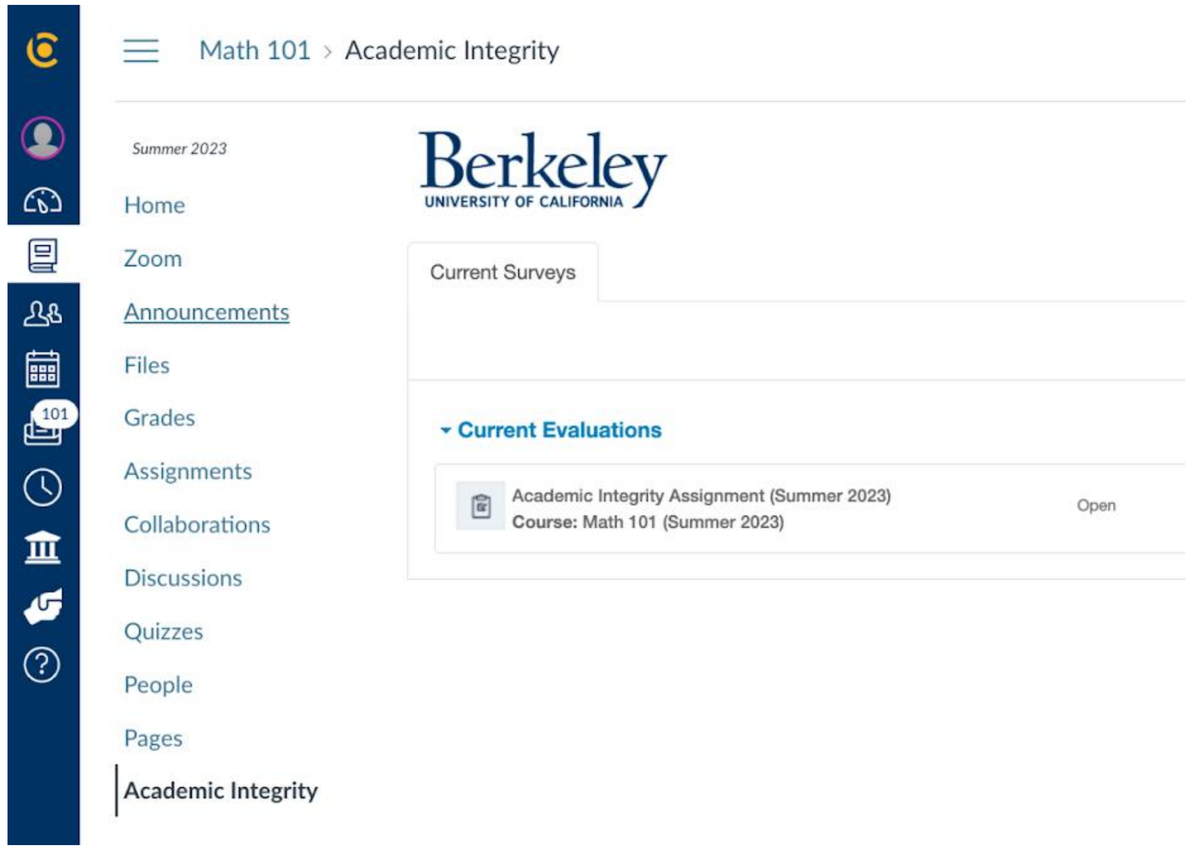
Discussion Section

- **Recall**: Data C182 students are not assigned to a discussion section yet
- **Solution**: this week, DSUS sent out emails to all students asking them their preferences for discussion sections.
- **Please fill this form out ASAP, due Friday (9/6) at noon.**
  - Ed post: [link]
- Discussion schedule is on the course webpage [link]
- Staff bios: [link]



Academic Integrity (bCourses)

- This week, you all will receive an assignment on bCourses titled something like "Academic Integrity Assignment", along with an email notice. [link]
- **Important**: all students must complete this assignment in bCourse by Week 4!
- From above: "In accordance with federal requirements established by the Department of Education, we need to verify that students are participating in their courses **by the end of the fourth week of classes…confirm the eligibility of your students to receive financial aid**.



Today's lecture

- Last lecture, we laid out the general machine learning method, and we defined **probabilistic models** (for classification), **likelihood based loss functions**, and **gradient based optimization**
- Now that we have a general recipe for how to learn parameters, we can ask:
  - If my learned parameters minimize the training loss, am I done? Should I deploy my model and move on?
  - How do I determine whether I am "satisfied" with the model?
  - What can I do if I am not satisfied with the model?
  - How to formulate the above questions/answers into an organized framework?

True risk and empirical risk

- **Risk** is defined as expected loss: $R(\theta) = \mathbb{E}[\ell(\theta; X, Y)]$
  - This is sometimes called **true risk** to distinguish from empirical risk below
- **Empirical risk** is the average loss on the training set: $\hat{R}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$
  - Supervised learning is oftentimes **empirical risk minimization (ERM)**
  - Is this the same as true risk minimization?

True risk and empirical risk

- The empirical risk looks just like a *Monte Carlo estimate* of the true risk, so shouldn't we have $\hat{R}(\theta) \approx R(\theta)$? Why might this not be the case?
  - Intuitively, the issue here is that we are already using the training dataset to learn θ — we can't "reuse" the same data to then get an estimate of the risk!
- When the empirical risk is low, but the true risk is high, we are **overfitting**
- When the empirical risk is high, and the true risk is also high, we are **underfitting**

Overfitting and underfitting: intuition

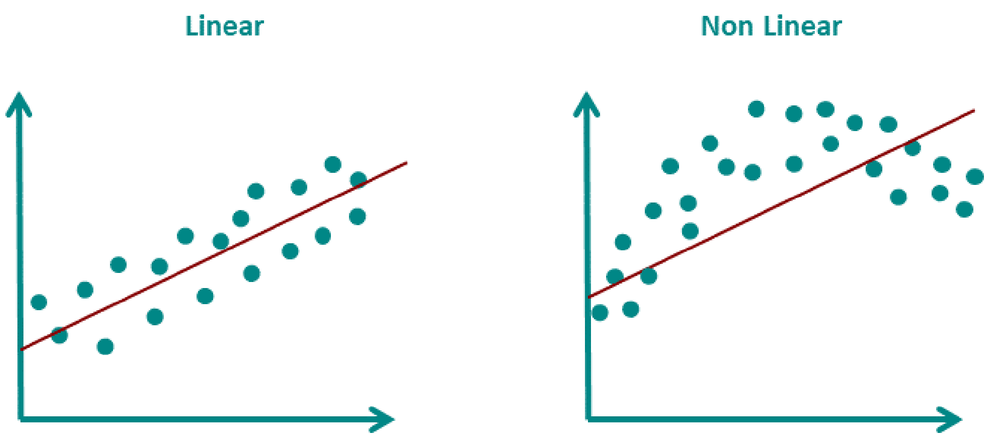
- Two very common failure modes of ML models
- **Overfitting**: "My model does well on the training data, but does poorly on unseen (test) data."
  - (In the risk framework) When the empirical risk is low, but the true risk is high
  - Common causes: This can happen if the dataset is too small and/or the model is too "powerful"

Overfitting and underfitting: intuition

- **Underfitting**: "My model does poorly on the training data, and also does poorly on unseen (test) data."
  - (In the risk framework) When the empirical risk is high, and the true risk is also high
  - Common causes: This can happen if the model is too “weak” and/or the optimization doesn’t work well (i.e., the training loss does not decrease satisfactorily)
  - What constitutes “high”? Often, that is up to the practitioner — that is, one must ask: “How well do I expect my model to work for this problem?”
- Generally, the true risk won’t be lower than the empirical risk

Model capacity

- Intuition: The **capacity** of a model is a measure of what kinds of functions/patterns it can represent.
- Examples:
  - A **linear regression** model is a very low capacity model: while it can model linear phenomenon well, it can't model anything nonlinear.
  - Large language models (**LLMs**) like GPT-3 are (very) high capacity models that can model human language and knowledge to an impressive quality.



Linear models are a poor fit for nonlinear phenomenon

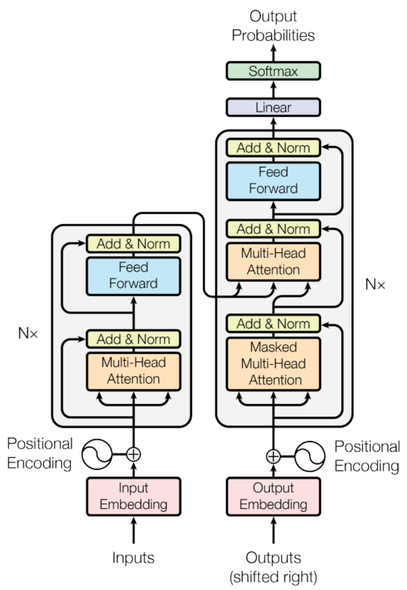


Figure 1: The Transformer - model architecture.

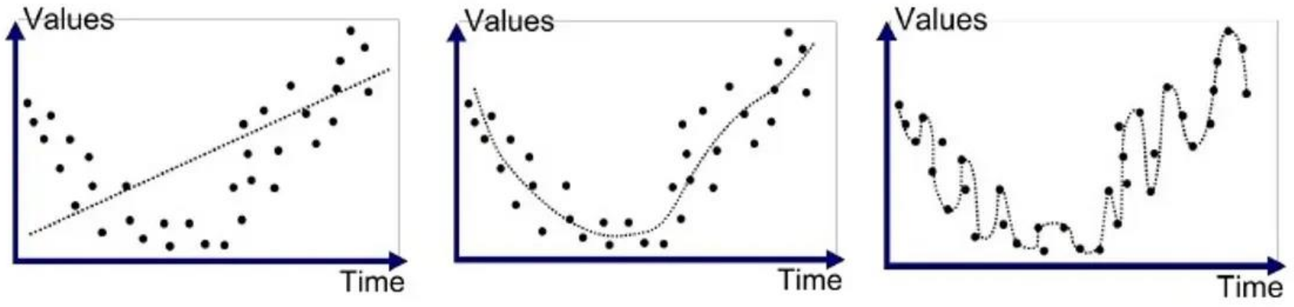
Prompt: please describe a study strategy to prepare for Data C182 "Deep Neural Networks", a course that is a deep dive into neural networks.

Response: Sure, I can do that for you. First, I'd recommend getting familiar with matrix/vector calculations, gradients, optimization, the pytorch library…

Source: https://machinelearningmastery.com/the-transformer-model/
https://arxiv.org/abs/1706.03762

Model capacity

- Concrete example: suppose we have a regression dataset of points in R^2. Let's restrict our model to the space of polynomial functions:
  - f(x) = a_0 * x + a_1 + x^2 + a_2 + a^3 + …
- A "high capacity" model can fit to many kinds of phenomenon
  - Ex: high-degree polynomials.
- A "low capacity" model can represent only a limited amount of phenomenon
  - Ex: degree-1 polynomial (aka straight line)



Underfitted | Good Fit/Robust | Overfitted

Low degree polynomial (eg linear function)

Very high degree polynomial

Simpler → More Complex

Source: https://medium.com/swlh/machine-learning-how-to-prevent-overfitting-fdf759cc00a9

Model capacity

- Similar to the n-degree polynomial regression example, we can make similar statements when comparing different ML model approaches

Linear regression, logistic regression

SVMs, random forests, "shallow" neural networks

"Deep" neural networks

Data C182!

"Classic" ML, eg CS 189

Simpler → More Complex

Notably: adding more **training data** to low capacity models stops helping after a certain (fairly easy to attain) point. But, for very-high capacity models typically one can keep adding more training data, and performance will continue to get better.

(a simplified caricature, but useful for intuitions)

Model capacity

- …and even within DNNs, there's stark differences in model capacity.
- While not 100% precise, a model's "parameter count" is an often-used measure for a model's capacity.
- Ex: ResNet50 (a successful ConvNet from 2015) has **25.6M** parameters ([link])
- GPT-3 (OpenAI, 2020) has **20B – 175B** parameters ([link] [link])

Two layer NNs

Convolutional networks, lightweight text encoders like LSTM, etc

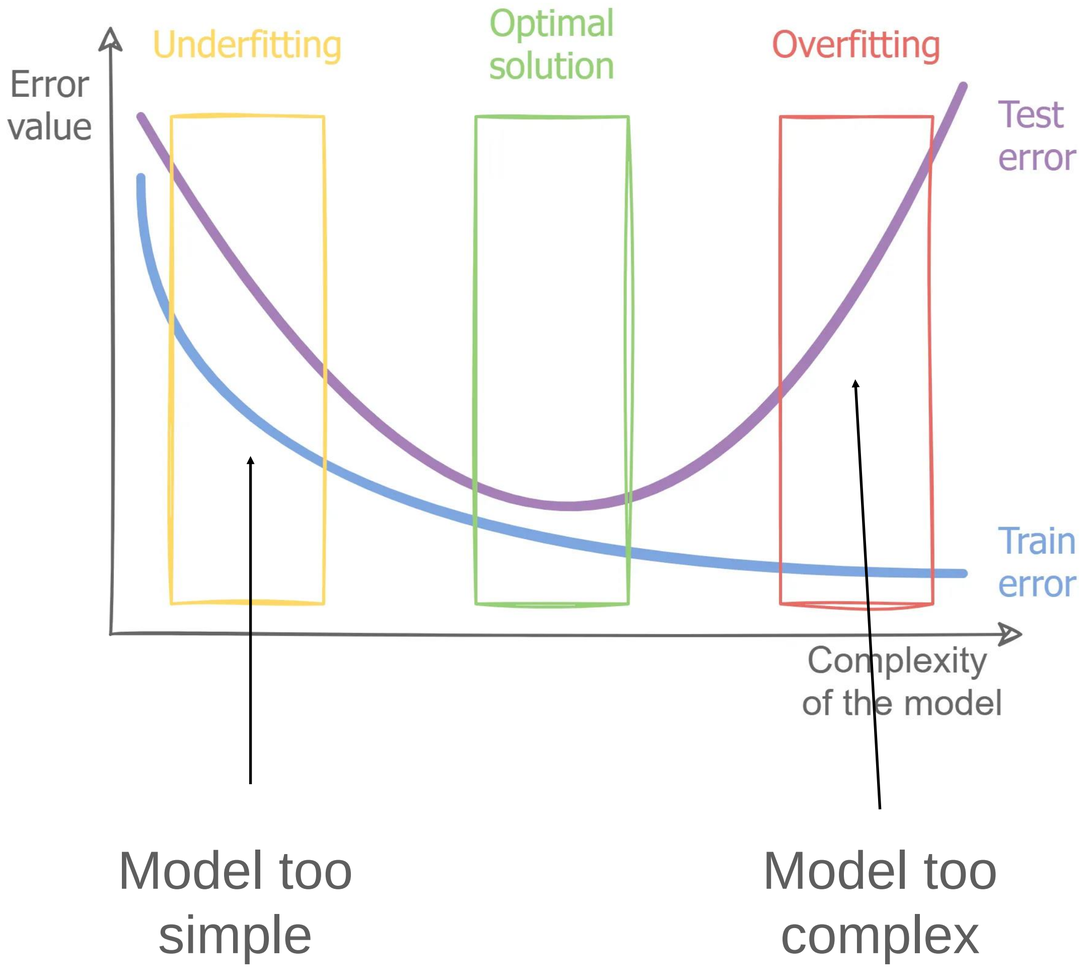
Large language models (LLMs), generative AI

Simpler → More Complex

(a simplified caricature, but useful for intuitions)

Model capacity and under/overfitting

- One useful knob for controlling underfitting vs overfitting is: model complexity (aka "capacity")
- For a fixed training dataset, we have the following rule of thumb (or "good guesses"):
  - **If you're underfitting**: often means that the model is too simple (ex: trying to fit a linear model to a nonlinear data distribution)
  - **If you're overfitting**: often means that your model is too complex and is fitting to the noise in the data.



Source: https://towardsdatascience.com/overfitting-and-underfitting-principles-ea8964d9c45c

Model class and capacity

- We use the term **model class** to describe the set of all possible functions that the chosen model can represent via different parameter settings
  - E.g., the set of all linear functions, the set of all neural network functions with a certain network architecture, …
- Roughly speaking, the **capacity** of a model (class) is a measure of how many different functions it can represent
  - E.g., neural networks have greater capacity than linear models, because neural networks can represent linear functions and more

Questions for the rest of the lecture

- How do we know whether/if we are overfitting or underfitting?
- Given a dataset of a particular size, how do we select:
  - a model class?
  - an algorithm?
  - **hyperparameters**?

Diagnosing overfitting and underfitting

- As mentioned, we cannot rely on the empirical risk $\hat{R}(\theta)$ being an accurate estimate of the true risk $R(\theta)$
  - But we need to estimate $R(\theta)$ in order to diagnose overfitting and underfitting!
- What's the problem? We want to use the dataset for ***two purposes***: learning θ and estimating $R(\theta)$
  - This suggests a natural solution: divide the dataset into two parts, one part for learning θ and one part for estimating $R(\theta)$

Training and validation sets

- We use the training set for training, i.e., learning θ
  - The loss on the training set also informs us of whether or not the empirical risk is "high" — if so, we are underfitting
  - Thus, we also use the training set for making sure that the optimization is working, i.e., decreasing training loss satisfactorily
- We reserve the validation set for diagnosing overfitting
  - The loss on the validation set should be an accurate estimate of the true risk, thus we can compare losses on these two sets

Remember: the machine learning method

- (or, at least, the deep learning method)

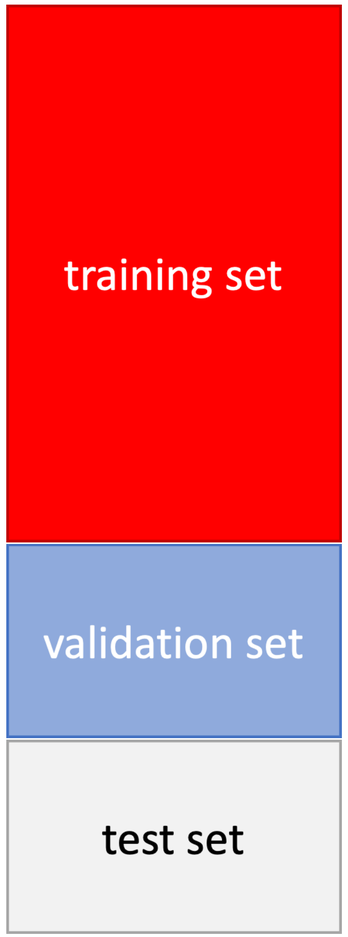
1. Define your **model** — which neural network, what does it output, …
2. Define your **loss function** — which parameters are good vs. bad?
3. Define your **optimizer** — how do we find good parameters?
4. Run it on a big GPU

Introducing: the machine learning *workflow*

1. Define your model
2. Define your loss function
3. Define your optimizer
4. Run it on a big GPU

1. Learn θ on the training set
   - if the training loss is not low enough…
   - you are **underfitting**! increase model capacity, improve optimizer, …
   - and go back to step 1
2. Measure loss on the validation set
   - if the training loss is much smaller than the validation loss…
   - you are **overfitting**! decrease model capacity, collect more data, …
   - and go back to step 1
3. Not overfitting or underfitting? You're done

You're done?



- What does "you're done" mean?
  - In industry, maybe it means: deploy your model
  - In research, competitions, this class, etc., it means: report your model's performance on a **test set**
- The test set is reserved for reporting final performance **only** and must never, ever be used for anything else

Combating overfitting

- Generally, underfitting is not as common of a concern as overfitting
  - Especially with deep learning, we can just keep making the network bigger…
  - … sometimes even without regard for overfitting! More on this later
- What tools and techniques do we have at our disposal if overfitting does occur?
  - Make the network smaller? But we like big models
  - Collect more data? This is a great option, *if possible*
  - Add more inductive biases — let's discuss how to do this via **regularization**

Regularization

- Broadly speaking, a **regularizer** is anything we add to the loss function and/or optimization that does not depend on the data
  - We add it to encode some prior belief about what a "good" model looks like — hence, it is a form of inductive bias
- Bayesian perspective: we can think of many forms of regularization as switching from a maximum likelihood approach to a **maximum a posteriori (MAP)** approach
  - I.e., from $\underset{\theta}{\mathrm{argmax}} \sum_{i=1}^{N} \log p_\theta(y_i|\mathbf{x}_i)$ to $\underset{\theta}{\mathrm{argmax}} \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i, \theta) + \log p(\theta)$

Maximum a posteriori estimation

- MLE is equivalent to optimizing the negative log likelihood (NLL) loss function
- MAP estimation is equivalent to adding a regularizer to the NLL loss function, in the form of $-\log p(\theta)$
  - What might be a reasonable choice for this regularizer?
- By far the most commonly used regularizer, when interpreted through the lens of MAP, can be thought of as setting $p(\theta) = \mathcal{N}(\theta; 0, \sigma^2 I)$
  - Then, we have $-\log p(\theta) = \sum_{i=1}^{D} \frac{1}{2} \frac{\theta_i^2}{\sigma^2} + const. = \lambda \parallel \theta \parallel_2^2$, where $\lambda = \frac{1}{2\sigma^2}$

ℓ_2-regularization

- With this choice of regularization, our final summed loss becomes $\sum_{i=1}^{N} - \log p(y_i|\mathbf{x}_i, \theta) + \lambda \parallel \theta \parallel_2^2$ — we call this ℓ_2-regularization
  - We usually pick λ directly rather than specifying σ^2 — thus, λ is a hyperparameter
- Why is this a good idea? Smaller parameters typically correspond to *smoother functions* that change less dramatically as the input changes
- You may have already seen this regularizer before in *ridge regression*
- In classification, this is often (somewhat erroneously) referred to as **weight decay**

Perspectives on regularization

- From a Bayesian perspective, the regularizer encodes our prior beliefs about which parameters are (or should be) more likely vs. less likely
- We can also interpret regularization through other perspectives:
  - Numerical perspective: sometimes the regularizer makes an underdetermined problem well determined
  - Optimization perspective: sometimes the regularizer makes the loss function better conditioned and thus easier to “traverse”
    - Paradoxically, more regularization can actually lead to less *underfitting*!

Recap

- So far: how do we know whether/if we are overfitting or underfitting?
  - By measuring and comparing training set loss vs. validation set loss
  - Then, we “tune the knobs” of model capacity, optimization, regularization, …
- Next: given a dataset of a particular size, how do we select settings for these knobs?
  - There are two approaches to answering this question that seem somewhat at odds: the “traditional”/statistical approach, which posits a “bias-variance tradeoff”, and the “deep learning” approach, which suggests that we just keep cranking the knobs up
  - Resolving the apparent inconsistency between these two views is the subject of much ongoing research

A probabilistic model for continuous outputs

for this part, we'll focus on regression, where the outputs $y \in \mathbb{R}$ are real values

"Label Noise"

"Ideal/true" function, deterministic.

$\varepsilon \sim \mathcal{N}(0, \sigma^2)$

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

we assume the data was sampled according to $X \sim p_x$, $Y|X = f(x) + \varepsilon$

how do we define a model that outputs a distribution over $y \,|\, \mathbf{x}$? one option:

$Y|X \sim \mathcal{N}(f_\theta(x), 1)$ the negative log likelihood loss is

$$-\log p_\theta(y_i | x_i) = \underbrace{\frac{1}{2}(f_\theta(x_i) - y_i)^2}_{} + (\text{const. w.r.t. } \theta)$$

Let's model our attempt to achieve the "ideal/true" distribution as a Gaussian

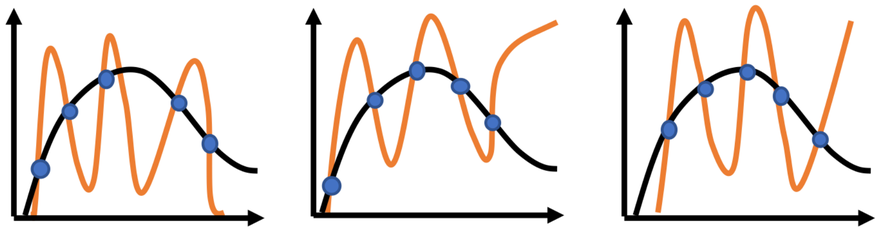
f_theta is our learned ML model

This lines up with our intuition that the regression loss should look something like **distance(predicted, target)**. Here, assuming that the distribution is normally distributed leads to **minimizing the (squared) L2 error**.

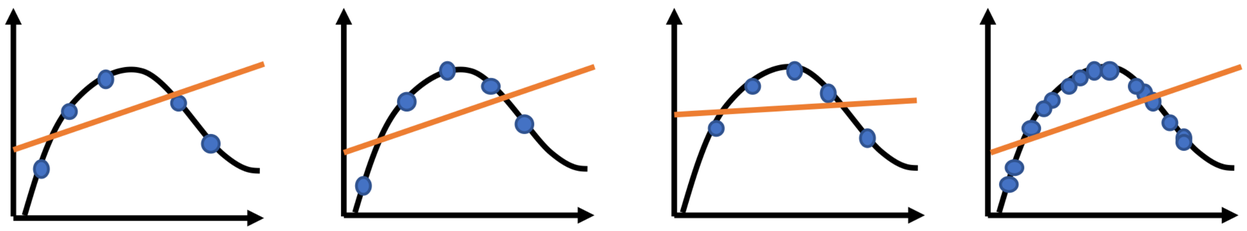
Intuition: bias and variance

- Since we assume the training data $\mathcal{D}$ was randomly sampled, we can ask the question: how does our model change for **different training sets**?
- If the model is overfitting, it will learn a different function for each training set
- If the model is underfitting, it learns similar functions, even if we combine all the training sets together — and all the learned functions are bad

overfitting



underfitting



The bias-variance decomposition ("tradeoff")

(let $\theta(\mathcal{D})$ be the MLE for $\mathcal{D}$, let $f_{\mathcal{D}} = f_{\theta(\mathcal{D})}$)

let's take a look at expected error for a test point $(\mathbf{x}^\star, \mathbf{y}^\star)$, where the expectation is over different training datasets $\mathscr{D}$:

$$\mathbb{E}\left[\left(f_{\mathcal{D}}(x') - y'\right)^2\right]$$

this is the only RV!

let $\bar{f}(\mathbf{x}^\star)$ be the expected prediction for $\mathbf{x}^\star$, where the expectation is again over the different training datasets (and the parameters that would be learned)

$$\bar{f}(x) = \mathbb{E}\left[f_{\mathcal{D}}(x)\right]$$

The bias-variance decomposition ("tradeoff")

$$\mathbb{E}\left[\left(f_{\mathcal{D}}(x') - y'\right)^2\right]$$

Tip: cross terms evaluate to 0 due to independence/deterministic. For derivation details, see: [link]

$$= \mathbb{E}\left[\left(f_{\mathcal{D}}(x') - f(x') + f(x') - y'\right)^2\right]$$

$$= \mathbb{E}\left[\left(f_{\mathcal{D}}(x') - f(x')\right)^2\right] + \underbrace{\mathbb{E}\left[\left(y' - f(x')\right)^2\right]}_{\sigma^2}$$

$\mathbb{E}[y']$

$$= \mathbb{E}\left[\left(f_{\mathcal{D}}(x') - \bar{f}(x') + \bar{f}(x') - f(x')\right)^2\right] + \sigma^2$$

$$= \left(\bar{f}(x') - f(x')\right)^2 + \underbrace{\mathbb{E}\left[\left(f_{\mathcal{D}}(x') - \bar{f}(x')\right)^2\right]}_{\mathrm{Var}(f_{\mathcal{D}}(x'))} + \sigma^2$$

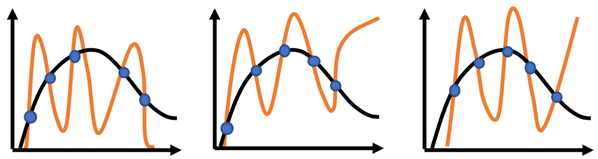
The bias-variance decomposition

- So: $\underbrace{\mathbb{E}[(f_{\theta(\mathcal{D})}(\mathbf{x}') - y')^2]}_{\text{Expected model error (over all training datasets)}} = \underbrace{(\bar{f}(\mathbf{x}') - f(\mathbf{x}'))^2}_{\text{Bias}} + \underbrace{\mathbb{E}[(f_{\theta(\mathcal{D})}(\mathbf{x}') - \bar{f}(\mathbf{x}'))^2]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible error}}$
  - The first term is called Bias^2 — how wrong is the model on expectation, regardless of the dataset it is trained on? Ex: simple models (eg linear reg) will have high bias for nonlinear datasets
  - The second term is Variance — regardless of the true function f, how much does the model change based on the training dataset? Ex: complex models (eg DNNs) have high variance
  - The last term is irreducible error — i.e., the noise in the data process itself
- So far, this is just a decomposition — where is the "tradeoff"?

The bias-variance tradeoff?

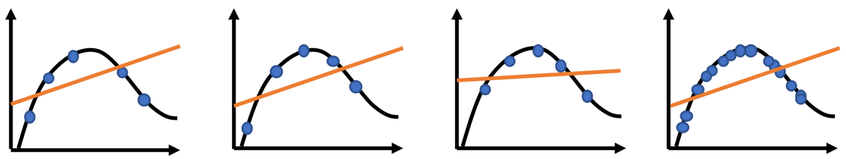
- Traditional statistics views bias and variance as “competing” sources of error that are regulated by model complexity
- High variance means insufficient data + a complex model class — overfitting
- High bias means an insufficiently complex model class — underfitting
- Traditional wisdom: find the "sweet spot" for model complexity to balance variance+bias (aka get your "best" model that generalizes well to unseen data)

overfitting

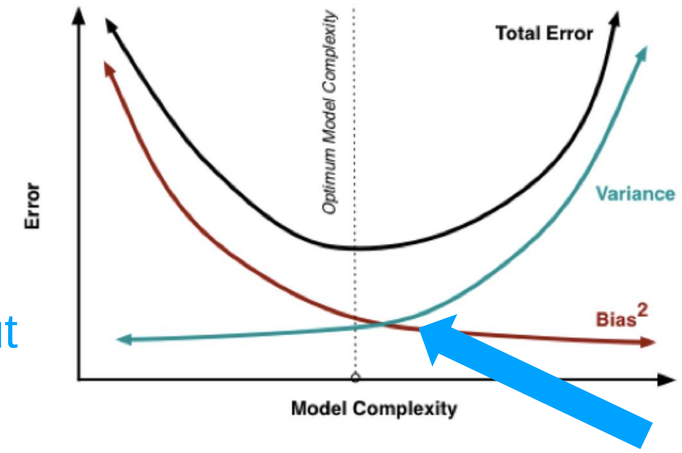


Ex: complex models (DNNs) have low bias, but high variance, potentially leading to overfitting

underfitting



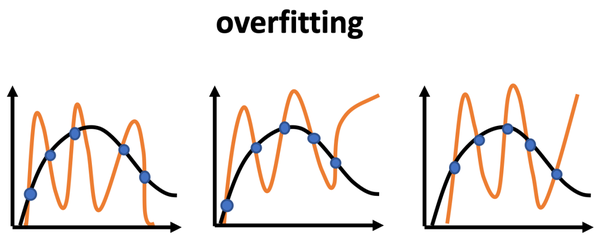
Ex: simple models (linear reg) have high bias (for nonlinear datasets), but low variance, potentially leading to underfitting

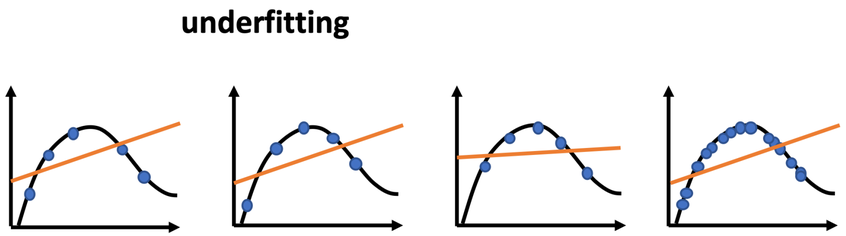


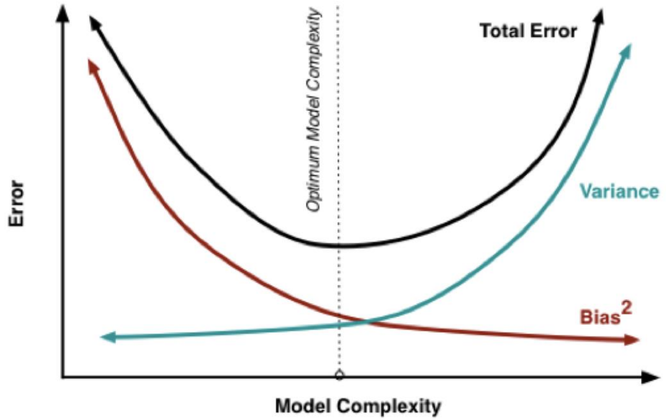
"sweet spot"

Bias-variance tradeoff: Why care?

- The Bias-Variance tradeoff gives us a nice theoretical foundation for intuitions like:
  - As I increase the complexity of my model, I'll expect my training error to (perhaps arbitrarily) go down ("bias"), but my model's generalizability on unseen data may suffer ("variance").
  - For DNNs, when training the same model on the same dataset multiple times (modulo different random seeds), performance can vary quite a bit (model weights certainly will!). This is the "variance" term at play!







Enter the deep learning perspective…

Allow me to quote Prof. Jitendra Malik

- "Modern neural network practice doesn't treat this as a tradeoff — go as high capacity as you can (e.g., networks like GPT-3 push the boundary of current computational hardware)"
- "We don't fear overfitting!"

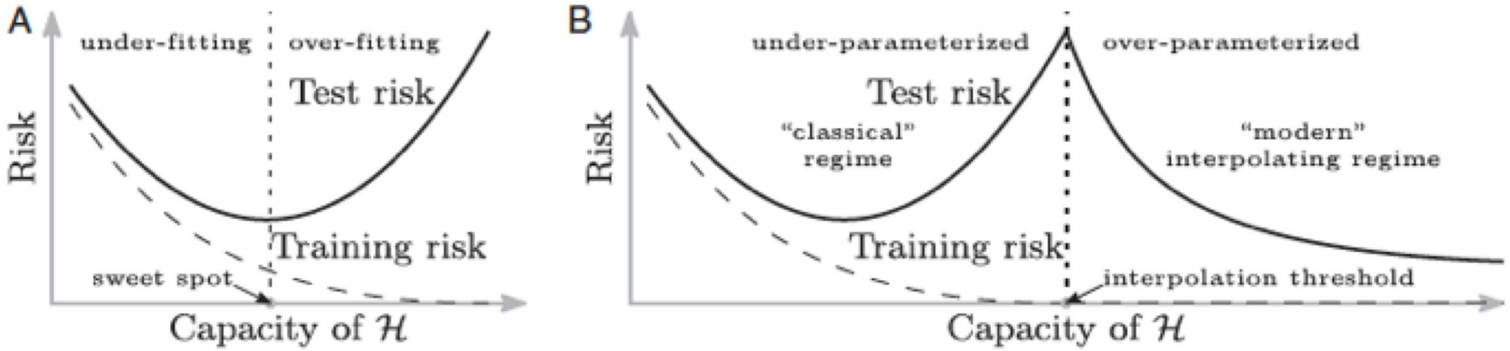


Fig. 1. Curves for training risk (dashed line) and test risk (solid line). (*A*) The classical U-shaped risk curve arising from the bias–variance trade-off. (*B*) The double-descent risk curve, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high-capacity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

Source: https://arxiv.org/abs/1812.11118

An industry perspective

- In my personal experience working in industry: go for as high-capacity ("expressive") ML models as possible, while being constrained by compute resources and cost ($)
  - **Train resource constraints**: GPU memory, training throughput
    - Practical tip: you want a model arch that can train relatively quickly (eg 1-2 days max ideally) so that you can run many experiments. Developer iteration speed is very important.
  - **Inference (serving) resource constraints**: GPU memory, inference latency/throughput.
  - GPU memory is a valuable, limited resource. Use it well!

Next steps

- At this point, we've provided a theoretical justification for the standard ML DNN pipeline of training your model on a labeled dataset with an appropriate loss function (aka "fitting a model to data via MLE")
- Next, we'll start getting into the details of neural network models: how they're constructed, how they're trained, etc.