Lecture 24: Closing Lecture (1/2)

Data C182 (Fall 2024). Week 15. Tuesday Dec 3rd, 2024

Speaker: Eric Kim

Announcements

- HW04 ("Visual Transformer, Masked Autoencoder") out! Due: Sun Dec 15th 11:59 PM PST
- University-wide course evaluations released! Please fill it out, we'd greatly appreciate it!
  - https://course-evaluations.berkeley.edu/Berkeley/

Final project is out!

- For info, see the Ed post: [link]
- Due date: Fri Dec 20th 11:59 PM PST
- Group size: 2-4 people
- You'll build an end-to-end ML pipeline
  - Dataset preprocessing, Dataloader
  - Modeling
  - Offline evaluation pipeline

Today's lecture

- Final Project advice
- Closing thoughts
- General advice
- AMA

Final Project Advice

- This project makes heavy use of: torch, pandas, and sklearn.
  - Hint: embrace usage of sklearn functions, especially for data preprocessing and offline evaluation!
    - Reading the sklearn docs + Googling will be helpful
- Start early, and carefully read the project spec, as there are many hints
- Tip: you can do this project on a CPU runtime in Colab! For reference, the staff solution notebook takes ~15 minutes to run from start-to-finish.
- (Part 1), (Part 2), and (Part 3) are largely orthogonal to each other, so you can split up each part to different people.

Final Project Advice

- (Part 5) Tip: important hyperparameters to tune:
  - Learning rate
    - Advice: first do a coarse grained sweep (eg LR=[1e-2, 1e-3, 1e-4, 1e-5]), then once you've identified the "region of goodness", do a finer-grained search.
    - Note: a "good" LR is dependent on model architecture. A good LR for modelA may not be good for modelB!
  - Model complexity (too complex of a model will lead to overfitting!)
- Consider implementing
  - Early stopping (on validation set)
  - Learning rate scheduler (cosine decay, `torch.optim. CosineAnnealingLR` [link])

Advice

- The following are some general advice that I wish I could tell my past self

Working on your own ML/DS projects

- (Ideally) Identify a project that you personally find interesting
  - Example: if you enjoy music, work on an ML+music project!
- (Or) do whatever project your manager/advisor tells you to do
- These days, many fields are interested in applying ML/DS to their problems
  - Example: medicine/health tech. According to a doctor I know: "It's easier for a ML/DS person to learn the medicine stuff for a project, than it is for a doctor to learn the ML/DS/tech stuff.“
  - Lots of opportunities out there!

The typical ML pipeline

(Part 1) Training/eval dataset collection

- This is the most important part!
- Ideally: you can find an existing dataset online
- If not - ideally you can collect a dataset by, say, scraping some website or calling some API
  - Ex: scraping Google reviews/website-text
- Worst case: manually collect data via human annotations
- Tip: if you find an interesting dataset online, think of neat ways you can use it!
  - Ex: People organize mp3s/MIDIs on websites by genre/composer/etc. Can use this to: train a music generator / genre classifier / etc.
  - Ex: 2D pixel art sprite collections (Pokemon: [link])

(Part 1.5) Feature engineering

- **Important question**: is there enough signal in the training data and input features for the task?
  - If not - you should rethink your approach to the problem.
  - This is where your human intuition (and ML wisdom) plays an important role.
    - Aka “look at the data!”

(Part 2) Modeling

- Formulate the task. Is this a classification task? Regression? Recommendation?
  - Ideally you should be able to identify a reasonable loss function

Model architecture

- Advice: start with the simplest model (eg logistic regression for classification) to establish a baseline.
- Using your baseline, run the entire train+eval pipeline end-to-end. Do things look correct?
  - Aka a "sanity check" for your ML pipeline
- Once you're convinced things look right, then proceed with model architecture iteration (eg training fancier models)

Model experiment advice

- Learning how to debug ML models is a skill gained by practice and experience
- Standard software bugs are obvious (eg Exceptions/errors)
- ML bugs are typically more subtle: rather than outright errors, they instead manifest as poor performance or unstable training
  - This is an art + skill!
- Tip: when creating a new model architecture, see if you can overfit on a single data sample. If you can't, then there's an issue with your code/model/pipeline/data.
- Google has decent guides: "The Rules for ML" [link], "Tuning Playbook" [link]

(Part 3) Evaluation

- Formulate the evaluation pipeline. Which metrics will you (or can you) compute?
  - Ex: accuracy/precision-recall-f1/MSE
- Eval metrics should be
  - (1) Easy to compute
  - (2) Easy to interpret
- Always prefer metrics that are “standard” for that kind of problem (eg in academia/industry).

(Part 4) Iterate!

- Often, you'll first iterate on modeling tweaks
  - Ex: "let's try modeling approach A, or model architecture B"
- But, you can (and should!) iterate on each part of the pipeline!
- **Training data**: maybe the training dataset isn't large enough, or is too noisy?
- **Feature engineering**: maybe the input features just aren't good enough?
- **Evaluation**: maybe our evaluation dataset isn't reliable/clean enough?
- Success often means getting your hands dirty, and diving deep into each part of the ML stack to identify any deficiencies
  - Rarely, do you just click "Train" and pop out an amazing model on your first try

Best thing to do?

- The single best, most consistent way I've learned to improve your model?
- Collect more (and better!) training data!

What I’ve learned from industry

Industry: early career advice

- Always aim to learn and grow (especially in your first 5 years!)
- Many people advise job-hopping every 3-4 years to maximize compensation ($). Instead, I have a different philosophy: maximize learning + growth.
  - When you leave a company and join a new one, you have to “start all over”
    - Establish connections, your internal “reputation”, etc.
    - But: if you’re in a stagnant role where it’s unlikely you have the opportunity to grow / get promoted, sometimes the best move is to switch companies.
- Tip: If you feel like you're stagnant in a role, proactively speak to your manager
  - If nothing changes and/or you feel that there isn't a good road forward, consider looking for another role (either at the same company, or at a different company)

Industry: Communication

- **Knowledge sharing**. Keep an eye out for any learnings/takeaways you make on projects, and prioritize sharing them out to the broader team
- **Documentation/run-books**. Diligently keep track of all commands/scripts/etc required to reproduce your work
  - Helps others (and “future you”) remember how to run things!
- Ex: write a running (or per-project) Google Doc that keeps track of your personal notes/learnings/commands/etc

Industry: Tooling

- Learn how to effectively utilize productivity tools (Git, CLI, etc).
  - Ex: learn how to use IDE's like PyCharm/VSCode. Practically a requirement when working on large codebases of hundreds of files and hundreds-of-thousands of lines of code.
    - Ex: "Jump to definition", "search entire repo for symbols", etc
- Learn how to write good Git commits
  - Common pitfall (esp. interns): writing one massive Git PR that contains multiple logically distinct changes.
    - Difficult for me to review
    - Better: break up large PR's into multiple smaller, logically separate PR's

Industry: Mentoring/upleveling

- As you grow in your career, keep an eye out for helping the rest of your team grow + uplevel as well
  - Ex: knowledge shareouts, writing + maintaining wikis/runbooks (“How to run infra pipeline X”), building reusable tooling/systems for broader use, etc
- Mentoring is not only personally fulfilling, but the team/management will appreciate your contributions to team growth (and, at higher levels, this becomes more of an expectation).
  - “good for you, your career, and your team!”

Industry: IC vs Manager

- Two tracks:
  - IC: "Individual Contributor". Aka software devs, data scientists, etc (aka "not manager").
  - Management
    - Engineering manager ("EM")
    - Program/project manager ("PM, TPM")

Managers

- What does a manager do?
  - One philosophy I've heard about managers is: "The manager's main job is to keep their direct reports happy, and to get them promoted."
    - Aka "People manager"
  - Other job: come up with the vision + direction on what the team needs to do
  - Other other job: shield their team from distractions (eg politics, etc)
  - Other other other job: (lots of) planning to ensure that everyone is working on projects that aligns with what the company needs
- (I personally don't know as much about this world, how they are evaluated, etc)

Switching tracks

- It's possible to switch from, say, an IC to a Manager (and back again!)
  - Some people enjoy one or the other. Both are valid!
- Note: traditionally (eg before tech companies like Google), the wisdom was that one needed to become a manager to "climb the ladder"
- However: in many newer tech companies, it's perfectly fine to remain an IC, as there's "plenty of ladder" to climb as an IC.

Industry leveling

- While each company has their own leveling structure + guidelines, I've found that the following guide on leveling/promotions helpful and fairly ubiquitous.
- The key concepts here are "**impact**" and "**scope**"
  - You are primarily promoted based on your **impact** to the company.
  - As you are promoted, your **scope** of impact will widen: first at the team-level, then (eventually) to the company-level.
    - Similarly the amount of people you internally work with (directly or indirectly) grows as you are promoted
- Note: I will use Google's software engineer leveling system [link] as a reference point (many companies, like Pinterest, base their levels off of Google's).

Industry: typical trajectory (L3 -> L4 expectations)

- L3: Entry level (first 1-2 years of your career)
- You'll learn how to work in that company's/team's codebase + ecosystem.
- Codebase: github, code reviews, etc.
- Ecosystem: internal tooling, processes, infrastructure (eg cloud clusters, etc)
- How will you be assessed?
  - Can you effectively deliver well-organized, maintainable code?
    - Aka "can you code well?"
  - Can you work well with others?

Industry: L4 -> L5 expectations

- L4: Typically 2+ years of experience.
- Here, you've demonstrated that you are a competent developer.
- Next, to get to L5: you need to demonstrate "impact" to the company
  - Example: "By shipping some code/model, I improved business metrics (clickthrough rate / ad revenue / etc) by X%"
- For L4->L5, the level of impact is expected to be somewhat small in scope, eg impact just within your team.
  - Example: "By completing my project, I was able to improve engagement metrics for a product feature my team owns by X%"
  - Example: "By completing my project, I was able to reduce our team's EC2 infra costs by X%, saving $Y per year."

Industry: L5 -> L6 expectations

- L5: typically 4-5+ years of experience. "Senior engineer".
  - In some companies, this is considered a "terminal level"
- Here, you've demonstrated that you can deliver impact at the team-level.
- Next: to get to L6, you need to demonstrate impact **outside** of your team.
  - This is trickier, as now it requires a certain amount of luck/strategic thinking
  - Not all projects have "L6 scope". Thus, sometimes you (and your manager) need to "play the game" a little to identify important projects that are "worthy" to get you promoted to L6
- Example: "By completing my ML infra project, I enabled ML teams to train larger-scale DNN models, which directly led to multiple engagement metrics wins."

Industry: L6 -> L7 expectations

- L6: "Staff engineer".
- Here, you've demonstrated that you can deliver impact both within and outside of your team.
- Next, to get to L7, you need to demonstrate impact across the entire org.
- The expectation is that you will lead initiatives with substantial scope
  - While this does require significant technical knowledge, you also need to be very good with people skills
  - Aka your goal is to convince other people to join your "cause"
  - Requires top-notch technical leadership skills.
- ...Beyond L7 I don't have a good barometer yet...

“Tactical” industry advice

- Prioritize learning and growth, both technical and “soft” skills
- Plan out projects in a way that allows “failing fast”
  - Ex: Rather than spend a year working on something that isn’t bearing fruit, instead structure things so that, if things don’t work after 2-3 months, communicate to your team/manager and pivot to a more promising direction.
- Work on projects that have a clear, measurable impact on important things
  - Aka “Work on things that matter”. Helps with motivation/morale too!
  - Ex: improving engagement metrics, infrastructure, revenue, dev quality of life, etc

Industry vs Academia?

- When I was an undergrad (2007-2012), the two routes seemed completely disjoint and incompatible.
- **Academia**: work on exciting state-of-the-art research + interesting problems. Get to teach and mentor others.
- **Industry**: work on projects (that may not be exciting to me) to make CompanyX more money.
  - Aka feel like a tiny cog in a giant machine.

Industry vs Academia?

- Now (2024): I feel strongly that there is a lot of interesting, exciting work in industry, especially in AI/ML/DS.
- Tech companies need good recommendation systems, which often boils down to learning good representations
  - Ex: Google/Meta/Pinterest/Amazon/etc
- The big tech companies often also have dedicated research labs doing state-of-the-art research
  - Google DeepMind, Facebook AI Research ("FAIR"), Open AI
  - ...and more!
- And: you get paid much more in industry (ha)

Industry AND academia

- Remember: you don't have to choose one or the other. You can try out both!
- You can always apply to grad school after working for a few years!

Grad school vs undergrad?

- One characterization I've heard is: "In undergrad, you learn how to follow instructions. In grad school, you learn how to **think**."
- Alt: The primary value of a PhD is not the thesis itself. Instead the main value is the ability to, when given a challenging problem with significant ambiguity, have the **drive** and **grit** to find solutions.
- **Drive**: a PhD will likely be largely driven by you (with guidance from your advisor). You'll need to learn how to work independently on problems.
- **Grit**: a PhD will have ups and downs. You will likely suffer from low morale/motivation at times, eg when you're struggling with your research project. A valuable life skill is to be able to get through the tough times and push on through.
  - Cautionary tale: PhD has ~50% attrition rate!

Grad school vs undergrad

- ML engineer roles frequently require at least an MS
- My take on why this is:
  - ML projects contain significant ambiguity and nontrivial technical complexity. Grad school tends to prepare you for this in a way that undergrad isn't able to.
  - ML is an advanced, challenging topic that is difficult to "master" in a few undergrad courses.
  - Like any craft, one needs to work on at least a few ML projects (ideally "from scratch") to truly learn the skills to become an effective ML practitioner

PhD advice: should I do a PhD?

- You should do a PhD only if you love your field, you love research, and you are OK with working on a single project in a single field for 4-6 years.
- IF you're ok with the above, then my single biggest advice is: make sure that you are compatible with your PhD advisor!
  - Your PhD advisor will make or break your PhD experience. There are many horror stories online of PhD student's lives made miserable by an advisor.
- Cautionary tale: the PhD attrition rate is ~50%!
- (for fun) PhD Comics is a humorous (yet often accurate!) view on the ups and downs of grad school life. Read it to get a peek into grad world: [link]

How did I get into ML? (short story)

- (Undergrad, 1st-2nd year) I didn't know what I wanted to do, fairly unfocused
- (2.5 year) Discovered my love for CS and teaching (thanks CS 61BL!)
- (3rd/4th year) My undergrad research exposed me to computer vision and image processing, thought it was neat!
- (5th year) I took a ML course (first offering of cs189!), thought it was neat!
- (Master's) I went to grad school focusing in computer vision / AI / ML.
  - Research thesis: face recognition. This thesis was one of the hardest things I've ever done, but I learned a LOT (aka my "mini PhD"). 3 years!
- (Industry, Pinterest) Joined the Visual Search team, worked on computer vision problems!

AMA

- Questions Ed post: [link]

- Given you have gone through graduate education, what are the greatest differences between undergrad DS/ML education and graduate? Also, not sure if you can speak to this, but were your essays for graduate school very technical in detail?

- Do you recommend masters vs undergrad vs phd for ML engineering?
- Any recommendations for college students who want to become a ML engineer?

- What was your path to be a ML engineer like? What skills, do you think, will make one a good ML engineer?