NPFL140 Large Language Models
LLM Training

http://ufal.cz/courses/npf140

Ondřej Dušek
14.3.2024
Training Transformers

In parallel: feed in training data & try to predict 1 next token at each position

- layers – Transformer blocks: attention & fully connected
- embeddings (~100s of numbers)
- numbered subwords
- positional encoding

0 <start>
1 <end>
2 weather
3 expe
4 storms
5 some
6 is
... ...
10 thunder-
... ...

multiple (6-100) layers

0 = 1
3 = expect
5 = some
10 = thunder-
12 = storms
Gradient Descent

• any neural net (supervised) training—**gradient descent** methods
  • minimizing a **cost/loss function**
    (notion of error – given a model output, how far off are we?)
  • calculus: derivative = steepness/slope
  • **backpropagation**: derivatives of all parameters w. r. t. cost (compound function)
  • follow the slope to find the minimum – derivative gives the direction
  • **learning rate** = how fast we go (needs to be tuned)

• gradient averaged over **mini-batches**
  • random bunches of a few training instances
  • not as erratic as using just 1 instance,
    not as slow as computing over whole data
  • **stochastic gradient descent**
Cost/Loss Functions

• differ based on what we’re trying to predict

• **default: logistic / log loss** ("cross entropy")
  • for any classification / softmax – including **word prediction** in LMs
    • classes from the whole dictionary
    • correct class has <100% prob. → loss is >0
  • pretty stupid for sequences, but works
    • sequence shifted by 1 ⇒ everything wrong

• other options:
  • squared error loss – for regression (floats)
  • hinge loss – binary classification (SVMs), ranking
  • many others, variants

reference: **Blue Spice is expensive**
prediction: **cheap pricey in the expensive price range**

Learning Rate & Momentum

• **LR: most important parameter** in (stochastic) gradient descent

• tricky to tune:
  • too high LR = may not find optimum
  • too low LR = may take forever

• **Learning rate decay**: start high, lower LR gradually
  • make bigger steps (to speed learning)
  • slow down when you’re almost there (to avoid overshooting)

• **Momentum**: moving average of gradients
  • make learning less erratic
  • \( m = \beta \cdot m + (1 - \beta) \cdot \Delta \), update by \( m \) instead of \( \Delta \)
Optimizers

• Better LR management
  • change LR based on gradients, less sensitive to settings

• **AdaGrad** – all history
  • remember sum of total gradients squared: $\sum_t \Delta_t^2$
  • divide LR by $\sqrt{\sum \Delta_t^2}$
  • variants: **Adadelta, RMSProp** – slower LR drop

• **Adam** – per-parameter momentum
  • moving averages for $\Delta$ & $\Delta^2$:
    $$m = \beta_1 \cdot m + (1 - \beta_1) \Delta$$
    $$v = \beta_2 \cdot v + (1 - \beta_2) \Delta^2$$
  • use $m$ instead of $\Delta$, divide LR by $\sqrt{v}$
  • often used as default nowadays

(Kingma & Ba, 2015)
https://arxiv.org/abs/1412.6980
https://ruder.io/optimizing-gradient-descent/
Schedulers

- more fiddling with LR – **warm-ups**
  - start learning slowly, then increase LR, then reduce again
  - may be repeated (**warm restarts**), with lowered maximum LR
    - allow to diverge slightly – work around local minima

- multiple options:
  - cyclical (=warm restarts) – linear, cosine annealing
  - **one cycle** – same, just don’t restart
  - **Noam scheduler** – linear warm-up, decay by $\sqrt{\text{steps}}$

- combine with base SGD or Adam/Adadelta etc.
  - momentum updated inversely to LR
  - may have less effect with optimizers
    - trade-off: speed vs. sensitivity to parameter settings

- [cyclical scheduler (warm restarts)](https://nn.labml.ai/optimizers/noam.html)
- [one cycle with cosine annealing](https://nn.labml.ai/optimizers/noam.html)
- [Noam scheduler with different parameters](https://nn.labml.ai/optimizers/noam.html)
When to stop training

- generally, when cost stops going down
  - despite all the LR fiddling
- problem: **overfitting**
  - cost low on training set, high on validation set
  - network essentially memorized the training set
  - → **check on validation set** after each epoch (pass through data)
  - stop when cost goes up on validation set
  - regularization (e.g. dropout) helps delay overfitting
- **bias-variance** trade-off:
  - smaller models may underfit (high bias, low variance = not flexible enough)
  - larger models likely to overfit (too flexible, memorize data)
  - XXL models: overfit soo much they actually interpolate data → good (🤔 ?)

(Dar et al., 2021) [https://arxiv.org/abs/2109.02355](https://arxiv.org/abs/2109.02355)
Self-supervised training

• train supervised, but **don’t provide labels**
  • use naturally occurring labels
  • create labels automatically somehow
    • corrupt data & learn to fix them
    • learn from rule-based annotation (not ideal!)
  • use specific tasks that don’t require manual labels

• good to train on huge amounts of data
  • language modelling
    • **next-word prediction** (~ most LLMs)
    • **MLM** – masked word prediction (~ encoder LMs, e.g. BERT)

• good to **pretrain** a LM self-supervised
  before you **finetune** it fully supervised (on your own task-specific data)

---

http://jalammar.github.io/illustrated-bert/
Pretrained (Large) Language Models (PLMs/LLMs)

- **BERT/RoBERTa**: Transformer encoder
  - masked word prediction, sentence order

- **BART** – encoder-decoder
  - denoising: masking, word removal… → regenerate original sentence

- **T5**: generalization of ↑ (multi-task, different prompts)

- multilingual: **XLM-RoBERTa, mBART, mT5**

- **GPT-2**, most LLMs (**GPT-3, LlaMa, Falcon, Mistral…**): Transformer decoder
  - next-word prediction

- many models released plug-and-play
  - **you only need to finetune** (and sometimes, not even that)
  - !! others (GPT-3/ChatGPT/GPT-4, Claude… closed & API-only)

---

**Notes**:
- Devlin et al., 2019 [https://aclanthology.org/N19-1423/](https://aclanthology.org/N19-1423/)
- Jiang et al., 2023 [https://huggingface.co/blog/falcon](https://huggingface.co/blog/falcon)
Parameter-efficient Finetuning

• Finetuning large models: don’t update all parameters
  • less memory-hungry (fewer gradients/momentums etc.)
  • trains faster
  • less prone to overfitting (~ regularization)

• Add few parameters & only update these
  • **Adapters** – small feed-forward networks after/on top of each layer
  • **Soft prompts** – tune a few special embeddings & use them on input
  • **LoRA** (low-rank adaptation):
    • 2 decomposition matrixes $A, B$ (parallel to each layer)
    • update = multiplication $AB$
    • $2 \times r \times d$ is much smaller than full weights ($d^2$)
    • update is added to original weights on the fly
  • **QLoRA** – LoRA + quantized 4/8-bit computation
    • to fit large models onto a small GPU

---

(Dettmers et al., 2023) [http://arxiv.org/abs/2305.14314](http://arxiv.org/abs/2305.14314)
Instruction Tuning

• Finetune for use with prompting
  • “in-domain” for what it’s used later

• Use **instructions** (task description) + **solution** in prompts
  • Many different tasks, specific datasets available

• Some LLMs released as base (“foundation”) & instruction-tuned versions

---

 Wei et al., 2022: https://arxiv.org/abs/2109.01652

https://nlpnewsletter.substack.com/p/instruction-tuning-vol-1
Reinforcement Learning

• Learning from **weaker supervision**
  • only get feedback once in a while, not for every output
  • good for globally optimizing sequence generation
    • you know if the whole sequence is good
    • you don’t know if step X is good
  • sequence ~ whole generated text

• Framing the problem as **states & actions & rewards**
  • “robot moving in space”, but works for text generation too
  • state = generation so far (prefix)
  • action = one generation output (subword)
  • defining rewards might be an issue

• Training: **maximizing long-term reward**
  • optimizing policy = way of choosing actions, i.e. predicting tokens

(Sutton & Barto, 2018)
RL from Human/AI Feedback (RLHF/RLAIF)

• RL improvements on top of instruction tuning (~InstructGPT/ChatGPT):
  1) generate lots of outputs for instructions
  2) have humans rate them (RLAIF variant: replace humans with an off-the-shelf LLM)
  3) learn a reward model (some kind of other LM: instruction + solution → score)
  4) use rating model’s score as reward in RL
• main point: reward is global (not token-by-token)

(Chuang et al., 2022)
http://arxiv.org/abs/2203.02155
https://openai.com/blog/chatgpt
https://huggingface.co/blog/rlhf
Direct Preference Optimization

- Trying to do the same thing, but without RL, with supervised learning
- Special loss function to check pairwise text preference
  - increases probability of preferred response
  - includes weighting w.r.t. reference model

$$L_{DPO}(\pi_{\theta}; \pi_{ref}) = -E_{(x,y_{w},y_{l}) \sim D} \left[ \log \sigma \left( \beta \log \frac{\pi_{\theta}(y_{w}|x)}{\pi_{ref}(y_{w}|x)} - \beta \log \frac{\pi_{\theta}(y_{l}|x)}{\pi_{ref}(y_{l}|x)} \right) \right]$$

![Diagram showing Reinforcement Learning from Human Feedback (RLHF) and Direct Preference Optimization (DPO)]