4. Training Neural Nets

http://ufal.cz/npfl099

Ondřej Dušek, Simone Balloccu, Zdeněk Kasner, Mateusz Lango, Ondřej Plátek, Patrícia Schmidtové

24. 10. 2023
Recap: Neural Nets

- **complex functions, composed of simple functions** (=layers)
  - linear, ReLU, tanh, sigmoid, softmax
- **fully differentiable**
- different arrangements:
  - feed forward / multi-layer perceptron
  - CNNs
  - RNNs (LSTM/GRU)
  - attention
  - Transformer
- input: binary, float, embedding
- tasks/problems: classification, regression, structured (sequences/ranking)
Supervised Training: Gradient Descent

- supervised training—**gradient descent** methods
  - minimizing a **cost/loss function**
    (notion of error – given system output, how far off are we?)
  - calculus: derivative = steepness/slope
  - follow the slope to find the minimum – derivative gives the direction
  - **learning rate** = how fast we go (needs to be tuned)

- gradient typically computed (=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**
  - batches may be **accumulated** to fit into memory
    - e.g. your GPU only fits one instance → compute gradients multiple times, then do 1 update

https://hackernoon.com/gradient-descent-aynk-7cbe95a778da
Cost/Loss Functions

• differ based on what we’re trying to predict

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

• squared error loss – for regression
  • forcing the predicted float value to be close to actual one

• hinge loss – binary classif. (SVMs), ranking
  • forcing the correct sign

• many others, variants

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

Backpropagation

- network ~ computational graph
  - reflects function/layer composition
- composed function derivatives – simple rules
  - basically summing over different paths
  - factoring ~ merging paths at every node
- **backpropagation** = reverse-mode differentiation
  - going back from output to input
  - ~ how every node affects the output
  - your graph output = cost function
    - → derivatives of all parameters w. r. t. cost
  - one pass through the network only → easy & fast
  - NN frameworks do this automatically

<table>
<thead>
<tr>
<th>Rules</th>
<th>Function</th>
<th>Derivative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication by constant</td>
<td>cf</td>
<td>cf</td>
</tr>
<tr>
<td>Power Rule</td>
<td>x^n</td>
<td>nx^{n-1}</td>
</tr>
<tr>
<td>Sum Rule</td>
<td>f + g</td>
<td>f' + g'</td>
</tr>
<tr>
<td>Difference Rule</td>
<td>f - g</td>
<td>f' - g'</td>
</tr>
<tr>
<td>Product Rule</td>
<td>fg</td>
<td>f g' + f' g</td>
</tr>
<tr>
<td>Quotient Rule</td>
<td>t/g</td>
<td>( \frac{f' g - g' f}{g^2} )</td>
</tr>
<tr>
<td>Reciprocal Rule</td>
<td>1/f</td>
<td>-f'/f^2</td>
</tr>
<tr>
<td>Chain Rule (as &quot;Composition of Functions&quot;)</td>
<td>f ∘ g</td>
<td>(f ∘ g) × g'</td>
</tr>
</tbody>
</table>
Learning Rate ($\alpha$) & Momentum

- **$\alpha$: most important parameter** in (stochastic) gradient descent
- tricky to tune:
  - too high $\alpha$ = may not find optimum
  - too low $\alpha$ = may take forever
- **Learning rate decay**: start high, lower $\alpha$ gradually
  - make bigger steps (to speed learning)
  - slow down when you’re almost there (to avoid overshooting)
  - linear, stepwise, exponential
  - **reduce-on-plateau** – check every now and then if we’re still improving, reduce LR if not
- **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 \( \Delta \) & \( \Delta^2 \):
    \[
m = \beta_1 \cdot m + (1 - \beta_1)\Delta, \quad v = \beta_2 \cdot v + (1 - \beta_2)\Delta^2
    \]
  • use \( m \) instead of \( \Delta \), divide LR by \( \sqrt{v} \)
  • often used as default nowadays
  • variant: **AdamW** – better regularization
    • not much difference though

(Kingma & Ba, 2015)
https://arxiv.org/abs/1412.6980

(Loshchilov & Hutter, 2019)
https://arxiv.org/abs/1711.05101

https://ruder.io/optimizing-gradient-descent/

- **LAMB** – Layer-wise Adaptive Moments for Batches
  - for larger batches & allowing to use larger LR (~unstable otherwise)

- **LARS** layer-wise adaptive rate scaling
  - layer-wise LRs, always multiplied by a trust ratio:
    \[ \alpha^l = \alpha \cdot \frac{|w^l|}{|\Delta^l|} \]
    - norm of weights/ norm of gradients
  - higher trust ratio = faster updates
  - start of training:
    low \( w \), high \( \Delta \) → slow **warm up**
  - towards convergence:
    higher \( w \), low \( \Delta \) → faster training

- **LAMB** ≈ **LARS** + AdamW

(You et al., 2020) 
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

https://spell.ml/blog/lr-schedulers-and-adaptive-optimizers-YHmwMhAAACYADm6F
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 (see →) 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)
Regularization: Dropout

- Regularization: preventing overfitting
  - making it harder for the network to learn, adding noise

- **Dropout** – simple regularization technique
  - more effective than e.g. weight decay (L2)
  - zero out some neurons/connections in the network at random
  - technically: multiply by dropout layer
    - 0/1 with some probability (typically 0.5–0.8)
  - at training time only – full network for prediction
  - weights scaled down after training
    - they end up larger than normal because there’s fewer nodes
    - done by libraries automatically
  - may need larger networks to compensate

(Srivastava et al., 2014)
http://jmlr.org/papers/v15/srivastava14a.html
Regularization: Multi-task Learning

• achieve better generalization by **learning more things at once**
  • a form of regularization
  • implicit data augmentation
  • biasing/focusing the model
    • e.g. by explicitly training for an important subtask

• parts of network shared, parts task-specific
  • hard sharing = parameters truly shared (most common)
  • soft sharing = regularization by parameter distance
  • different approaches w. r. t. what to share

• training – **alternating** between tasks
  • **catastrophic forgetting**: if you don’t alternate, the network forgets previous tasks

(Fan et al., 2017) http://arxiv.org/abs/1706.04326
(Luong et al., 2016) http://arxiv.org/abs/1511.06114
Autoencoders

- Using NNs as **generative models**
  - more than just classification – modelling the whole distribution
    - (of e.g. possible texts, images)
    - generate new instances that look similar to training data

- **Autoencoder**: input $\rightarrow$ encoding $\rightarrow$ input
  - encoding $\sim$ “embedding” in latent space (i.e. some vector)
  - trained by reconstruction loss
  - problem: can’t easily get valid embeddings for generating new outputs
    - parts of embedding space might be unused – will generate weird stuff
    - no easy interpretation of embeddings – no idea what the model will generate

- extension – **denoising autoencoder**:
  - add noise to inputs, train to generate clean outputs
  - use in multi-task learning, representations for use in downstream tasks

https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bf67eb5daf
Variational Autoencoders

• Making the encoding latent space more useful
  • using Gaussians – continuous space by design
  • encoding input into vectors of means $\mu$ & std. deviations $\sigma$
  • sampling encodings from $N(\mu, \sigma)$ for generation
    • samples vary a bit even for the same input
    • decoder learns to be more robust
  • model can degenerate into normal AE ($\sigma \to 0$)
    • we need to encourage some $\sigma$, smoothness, overlap ($\mu \sim 0$)
    • add 2nd loss: KL divergence from $N(0,1)$
      • VAE learns a trade-off between using unit Gaussians & reconstructing inputs

• Problem: still not too much control of the embeddings
  • we can only guess what kind of output the model will generate

https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf
https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73
http://kvfrans.com/variational-autoencoders-explained/
VAE details

- **VAE objective:**
  - "AE" • **reconstruction loss** (maximizing $p(x|z)$ in the decoder), MLE as per usual
  - "V" • **latent loss** (KL-divergence from ideal $p(z) \sim \mathcal{N}(0,1)$ in the encoder)

$$
\mathcal{L} = - \mathbb{E}_q [\log p(x|z)] + KL[q(z|x)||p(z)]
$$

- This is equivalent to maximizing true $\log p(x)$ with some error
  - i.e. maximizing **evidence lower bound** (ELBO) / variational lower bound:

$$
\mathbb{E}_q [\log p(x|z)] - KL[q(z|x)||p(z)] = \log p(x) - KL[q(z|x)||p(z|x)]
$$

- **Sidestepping sampling** – **reparameterization trick**
  - $z \sim \mu + \sigma \cdot \mathcal{N}(0,1)$, then differentiate w. r. t. $\mu$ and $\sigma$
  - differentiating w. r. t. $\mu$ & $\sigma$ still works, no hard sampling on that path

https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/
Discrete VAE: Gumbel-Softmax

- “reparameterization trick for discrete distributions”
  - same idea, just with a **discrete/categorical distribution**
  - this makes the latent space better interpretable

**Gumbel-max trick:**
- categorial distribution \( \pi \) with probabilities \( \pi_i \)
- sampling from \( \pi \): \( z = \text{onehot}(\arg\max_i (\log \pi_i + g_i)) \)

- Swap argmax for softmax with temperature \( \tau \)
  - differs from \( \pi \) if \( \tau > 0 \), but may be close to \( i \)
  - approx. sample of the true distribution
  - fully differentiable
  - \( g_i \) bypassed in differentiation, same as \( \mathcal{N}(0,1) \) in Gaussian sampling

\[
\gamma_i = \frac{\exp\left(\frac{\log(\pi_i) + g_i}{\tau}\right)}{\sum_{j=1}^{\mathcal{N}} \exp\left(\frac{\log(\pi_j) + g_j}{\tau}\right)}
\]

Gumbel noise: \( g_i = -\log(-\log(\text{Uniform}(0,1))) \)

\( \tau \rightarrow 0 \): more like one-hot
\( \tau \rightarrow \infty \): more like uniform

(Jang et al., 2017)
https://arxiv.org/abs/1611.01144

https://anotherdatum.com/gumbel-gan.html
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
    - MLM – masked word prediction (~like word2vec)
  - **autoencoding**: predict your own input
  - good to **pretrain** the network for a final task
- unsupervised, but with supervised approaches

http://jalammar.github.io/illustrated-bert/
Pretraining & Finetuning

• 2-step training:
  1. **Pretrain** a model on a huge dataset (**self-supervised**, language-based tasks)
  2. **Fine-tune** for your own task on your smaller data (**supervised**)

• ~pretrained embeddings, many variants
  • mostly Transformer architecture
  • pretraining tasks vary and make a difference

• typical tasks:
  • masked language modelling (masked words/spans)
  • next-word prediction
  • denoising scrambled texts

(Lewis et al., 2020) https://aclanthology.org/2020.acl-main.703/
Pretrained (Large) Language Models (PLMs/LLMs)

- **BERT/RoBERTa**: Transformer encoder
  - masked word prediction, sentence order
  - (Devlin et al., 2019) https://aclanthology.org/N19-1423/
  - (Liu et al., 2019) http://arxiv.org/abs/1907.11692

- **BART** – encoder-decoder
  - denoising autoencoder: masking, word removal… → generate original sentence
  - (Lewis et al., 2020) https://aclanthology.org/2020.acl-main.703/

- **T5**: generalization of ↑ (multi-task, different prompts)
  - (Xue et al., 2021) https://aclanthology.org/2021.naacl-main.41

- **GPT-2**, most LLMs (**GPT-3, LlaMa, Falcon, Mistral…**): Transformer decoder
  - next-word prediction (=language modeling)
  - (Radford et al., 2019) https://openai.com/blog/better-language-models/
  - (Touvron et al., 2023) http://arxiv.org/abs/2307.09288
  - https://huggingface.co/blog/falcon
  - (Jiang et al., 2023) https://arxiv.org/abs/2310.06825

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

- **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)

- **you only need to finetune (and sometimes, not even that)**

- **!! others (GPT-3/ChatGPT/GPT-4, Claude… closed & API-only)**

- **https://github.com/huggingface/transformers**
Parameter-efficient Finetuning

- Finetuning large models: don’t update all parameters
  - faster, 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 in a prompt
  - **LoRA** (low-rank adaptation):
    - updates = 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)
• No model finetuning, just show a few examples in the input (=prompt)
• pretrained LMs can do various tasks, given the right prompt
  • they’ve seen many tasks in training data
  • only works with the larger LMs (>1B)
• adjusting prompts often helps
  • “prompt engineering”
  • zero-shot (no examples) vs. few-shot
• chain-of-thought prompting: “let’s think step by step”
• adding / rephrasing instructions (see → →)

http://ai.stanford.edu/blog/understanding-incontext/

(Liu et al., 2023) https://arxiv.org/abs/2107.13586
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 = e.g. sentence, dialogue

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

• **Training: maximizing long-term reward**
  • via state/action values (Q function)
  • directly – optimizing policy

(Sutton & Barto, 2018)
Instruction Tuning / RL from Human Feedback

- 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

- RL improvements on top of this (~InstructGPT/ChatGPT):
  1) generate lots of outputs for instructions
  2) have humans rate them
  3) learn a rating model (some kind of other LM: instruction + solution → score)
  4) use rating model score as reward in RL
- main point: reward is global (not token-by-token) – RL-free alternatives exist

https://nlpnewsletter.substack.com/p/instruction-tuning-vol-1
(Wei et al., 2022) https://arxiv.org/abs/2109.01652
(Wei et al., 2022) https://arxiv.org/abs/2203.02155
(Ouyang et al., 2022) https://openai.com/blog/chatgpt
(Ouyang et al., 2022) http://arxiv.org/abs/2203.02155
(Rafailov et al., 2023) http://arxiv.org/abs/2305.18290
Adversarial Learning / Generative Adversarial Nets

• Training generative models to generate believable outputs
  • to do so, they necessarily get a better grasp on the distribution

• Getting loss from a 2nd model:
  • discriminator $D$ – “adversary” classifying real vs. generated samples
  • generator $G$ – trained to fool the discriminator
    • the best chance to fool the discriminator is to generate likely outputs

• Training iteratively (EM style)
  • generate some outputs
  • classify + update discriminator
  • update generator based on classification
  • this will reach a stable point

(Goodfellow et al, 2014)
Clustering

• Unsupervised, finding similarities in data
• basic algorithms
  • k-means: assign into $k$ clusters randomly, iterate:
    • compute means (centroids)
    • reassign to nearest centroid
  • Gaussian mixture: similar, but soft & variance
    • clusters = multivariate Gaussian distributions
    • estimating probabilities of belonging to each cluster
    • cluster mean/variance based on data weighted by probabilities
  • hierarchical (bottom up):
    start with one cluster per instance, iterate:
    • merge 2 closest clusters
    • end when you have $k$ clusters / distance is too big
  • hierarchical top-down (reversed $\rightarrow$)
• distance metrics & features decide what ends up together
Summary

• Supervised training
  • cost function
  • stochastic gradient descent – minibatches
  • backpropagation
  • learning rate tricks – optimizers (Adam), schedulers
  • regularization: dropout, multi-task training
• Self-supervised learning (~kinda unsupervised)
  • autoencoders, denoising, variational autoencoders
  • (masked) language models
• PLMs/LLMs: pretraining & finetuning, prompting, instruction tuning
• Reinforcement learning (more to come later)
• Unsupervised: GANs, clustering
Thanks

Contact us:
https://ufaldsg.slack.com/
{odusek,hudecek,kasner}@ufal.mff.cuni.cz
Zoom/Skype/Troja

Get the slides here:
http://ufal.cz/npfl099

References/Further:
Kim et al. (2018): Tutorial on Deep Latent Variable Models of Natural Language
(http://arxiv.org/abs/1812.06834)

Neural nets tutorials:
• https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0
• https://minitorch.github.io/index.html
• https://objax.readthedocs.io/en/latest/