NPFL099 Statistical Dialogue Systems

4. Training Neural Nets

http://ufal.cz npfl099

Ondřej Dušek, Vojtěch Hudeček & Tomáš Nekvinda

25. 10. 2021
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 forward pass 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
  • 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** – for binary classification (SVMs), ranking
  • forcing the correct sign

• many others, variants

https://medium.com/@risingdeveloper/visualization-of-some-loss-functions-for-deep-learning-with-tensorflow-9f60be9d09f9
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
  - output = cost function
  - → derivatives of all parameters w. r. t. cost
  - one pass through the network only → easy & fast
  - NN frameworks do this automatically

### Rules & Derivatives

<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>(t' g - g' t) / 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>

[https://www.mathsisfun.com/calculus/derivatives-rules.html](https://www.mathsisfun.com/calculus/derivatives-rules.html)

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 $\Delta$

[Image: http://cs231n.github.io/neural-networks-3/]
Optimizers

• Better LR management
  • change LR based on gradients
  • much less sensitive to user setting

• 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 $\sqrt{v}$
  • used as default in most applications
  • variant: AdamW – decoupled LR drop


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


http://kaeken.hatenablog.com/entry/2016/11/10/203151
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 – linear, cosine annealing
  • **one cycle** – same, just don’
  • **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 improving
  • despite all the LR fiddling

• problem: **overfitting**
  • cost is 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
  - so the network doesn’t “forget”
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 manually created 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 (see ➔)

• good to **pretrain** the network for a final task

• unsupervised, but with supervised approaches

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 ~ “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-1bfe67eb5daf
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

[Links]
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 $p(z)\sim N(0,1)$ 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
  • $z \sim \mu + \sigma \cdot N(0,1)$, then differentiate w.r.t. $\mu$ and $\sigma$
    • differentiating w.r.t. $\mu$ & $\sigma$ still works, no hard sampling on that path

[Link to Wise Odd's blog post on variational autoencoders](https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/)
Discrete VAE: Gumbel-Softmax

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

• “reparameterization trick for discrete distributions”
  • same idea, just with a **discrete/categorial 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
  • approx. sample of the true distribution
  • fully differentiable
  • $g_i$ bypassed in differentiation, same as $\mathcal{N}(0,1)$ in Gaussian sampling

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

• Direct control over types of things to generate
• Additional conditioning on a given label/type/class $c$
  • $c$ can be anything (discrete, continuous…)
    • image class: MNIST digit
    • sentiment
    • “is this a good reply?”
    • coherence level
  • just concatenate to input
  • given to both encoder & decoder at training time

• Generation – need to provide $c$
  • CVAE will generate a sample of type $c$
  • Latent space is partitioned by $c$
    • same latent input with different $c$ will give different results

https://ijdykeman.github.io/ml/2016/12/21/cvae.html
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

• **BERT** + variants: multilingual, **RoBERTa** (optimized)
• **GPT**(-2/-3): Transformer decoder only, next-word prediction
• **BART**: BERT as denoising autoencoder (more below)
• **T5**: generalization, many variants

• a lot of pretrained models released plug-and-play
  • you only need to finetune (and sometimes, not even that)

---

(Devlin et al., 2019)
https://www.aclweb.org/anthology/N19-1423
https://github.com/google-research/bert


(Radford et al., 2019)
https://openai.com/blog/better-language-models/
(Brown et al., 2020)


---

[https://github.com/huggingface/transformers](https://github.com/huggingface/transformers)
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

https://www.displayr.com/what-is-hierarchical-clustering/
https://towardsdatascience.com/gaussian-mixture-models-d13a5e915c8e
https://www.youtube.com/watch?v=9YA2t78Ha68
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
• 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

• Unsupervised
  • generative adversarial nets
  • clustering

• Reinforcement learning (more to come later)
Contact us:
https://ufaldsg.slack.com/
{odusek,hudecek,nekvinda}@ufal.mff.cuni.cz
Troja N230/231/233 (by agreement)

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/