Stanford Compendium in Mathematics and Computer Science

Adithya C. Ganesh

August 29, 2019

This document serves as a compendium of my notes at Stanford. Mostly for personal reference, still under construction.

These notes were mostly taken live in lecture, so they definitely contain typos. All errors (typographical or conceptual) are my own.

Contents

1

Independent + CS229: Statistical Learning

1.1 Matrix cookbook

This doesn't matter as much for analytic calculations, but is useful for implementing autodiff on tensors from scratch.

`http://www.math.uwaterloo.ca/~hwolkowi//matrixcookbook.pdf`

1.2 Large-scale distributed training

Goyal et al. 2017

- Key contribution: no loss in accuracy when training with large minibatch sizes up to 8192 images.
- Linear scaling rule for adjusting learning rates as a function of mini-batch size. Concretely, when minibatch size is multipled by k, multiply the learning rate by k.
- Warmup scheme that overcomes optimization challenges early in training. Specifically, use a *low constant* learning rate for the first few epochs of training. For a large minibatch of size kn, train with the low learning rate of η for the first 5 epochs and then return to the target learning rate of $\hat{\eta} = k\eta$.
- Update rule:

$$\hat{w}_{t+1} = w_t - \hat{\eta}\frac{1}{kn}\sum_{j<k}\sum_{x\in\mathcal{B}_j}\nabla l(x, w_t).$$

1.3 Hyperparameter optimization

1.3.1 Population-based training

Jaderberg et al. 2017

- Key contribution: asynchronous optimization algorithm which uses a fixed computational budget to jointly optimize a population of models and their hyperparameters

- Schedule of hyperparameters (instead of fixing a set for the whole course of training)
- Sequential optimization: run multiple training runs (potentially with early stopping)
- Parallel random/grid search: train multiple models in parallel with different weight initializations + hyperparameters, with the view that one of the models will be optimized the best.
- Population based training: starts like parallel search, randomly sampling hyperparameters and weight initializations. Each training run asynchronously evaluates its performance periodically. Explores new hyperparameters by modifying the better model's hyperparameters, before training is continued.

Algorithm 1 Codistillation

Input loss function $\phi(\texttt{label}, \texttt{prediction})$
Input distillation loss function $\psi(\texttt{aggregated\_label}, \texttt{prediction})$
Input prediction function $F(\theta, \texttt{input})$
Input learning rate η
for `n_burn_in` steps **do**
  for θ_i in `model_set` **do**
    y, f = `get_train_example()`
    $\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, F(\theta_i, f))\}$
  end for
end for
while not converged **do**
  for θ_i in `model_set` **do**
    y, f = `get_train_example()`
    $\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(\mathrm{y}, F(\theta_i, f)) + \psi(\{\frac{1}{N-1} \sum_{j \neq i} F(\theta_j, f)\}, F(\theta_i, f))\}$
  end for
end while

1.3.2 ENAS

Pham et al. 2018

- Key contribution: fast + inexpensive approach for automatic model design.
- Controller (trained with policy gradient) discovers neural network architectures by searching for an optimal subgraph within a large computational graph.
- To train the shared parameters ω of the child models: we fix controller policy $\pi(\mathbf{m}; \theta)$ and perform SGD on ω to minimize expected loss $\mathbb{E}_{\mathbf{m} \sim \pi}[\mathscr{L}(\mathbf{m}; \omega)]$. Gradient is computed using Monte Carlo estimate

$$\nabla_\omega \mathbb{E}_{\mathbf{m} \sim \pi(\mathbf{m}; \theta)}[\mathscr{L}(\mathbf{m}; \omega)] \approx \frac{1}{M} \sum_{i=1}^{M} \nabla_\omega \mathscr{L}(\mathbf{m}_i, \omega),$$

  where $\mathbf{m}_i$ are sampled from $\pi(\mathbf{m}; \theta)$.
- To train controller parameters θ: fix ω and update θ to maximize the expected reward $\mathbb{E}_{\mathbf{m} \sim \pi(\mathbf{m}; \theta)}[\mathscr{R}(\mathbf{m}, \omega)]$. Use Adam optimizer, and compute the gradient with REINFORCE, with a moving average baseline to reduce variance.

1.4 Distillation

Hinton et al. 2015

- Ensembling: train many different models on the same data and then average their prediction
- Distillation: compress the knowledge in an ensemble into a single model which is much easier to deploy.

Anil et al. 2018

- Online distillation enables extra parallelism.
- Codistillation algorithm:

Algorithm 1 Codistillation

Input loss function $\phi(\texttt{label}, \texttt{prediction})$
Input distillation loss function $\psi(\texttt{aggregated\_label}, \texttt{prediction})$
Input prediction function $F(\theta, \texttt{input})$
Input learning rate η
for `n_burn_in` steps **do**
  for θ_i in `model_set` **do**
    y, f = `get_train_example()`
    $\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, F(\theta_i, f))\}$
  end for
end for
while not converged **do**
  for θ_i in `model_set` **do**
    y, f = `get_train_example()`
    $\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, F(\theta_i, f)) + \psi(\{\frac{1}{N-1} \sum_{j \neq i} F(\theta_j, f)\}, F(\theta_i, f))\}$
  end for
end while

1.5 ConvNet architectures

1.5.1 Recognition

- PNASNet-5-Large
  - Similar to NAS, but performs search progressively (starting with models of low complexity).
- NASNet-A-Large
  - Uses a 50-step RNN as a controller to generate cell specifications.
- SENet154
- PolyNet

1.5.2 Detection

- Faster RCNN
- YOLO
- RetinaNet

1.5.3 Segmentation

- FCNet
- DeepLabv4
- Dilated convolutions

1.5.4 WaveNet

Uses dilated convolutions:

- Let F be a discrete function, and k be a discrete filter. The discrete convolution operator $*$ is defined as
$$(F * k)(\mathbf{p}) = \sum_{\mathbf{s}+\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t}).$$
More generally, let l be a dilation factor. The l-dilated convolution can be defined as
$$(F *_l k)(\mathbf{p}) = \sum_{\mathbf{s}+l\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t}).$$
- Implemented in TensorFlow as `tf.nn.atrous_conv2d`

1.5.5 Self-attention networks

1.6 PyTorch

1.7 Backpropagation: CS231 intuitions

1.8 Backpropagation: a graph theory perspective

Notes from Chris Olah's post: `http://colah.github.io/posts/2015-08-Backprop/`

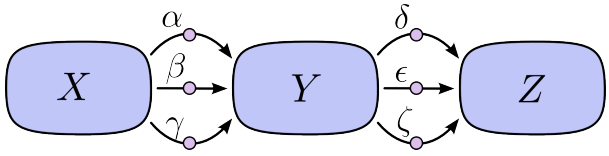
Backpropagation is a very common algorithm, and is often referred to as "reverse-mode differentation." At its core, it is a tool for calculating derivatives quickly.

Why are computational graphs a good abstraction? To apply the multivariate chain rule:

1. Sum over all possible paths from one node to the other.
2. Multiply the derivatives on each edge of the path together.

1.8.1 Combinatorial explosion

This is all just standard chain rule. But how do you deal with cases like this?[1]



There are 9 paths in the above diagram. Instead of naively summing over the paths, we can factor them:

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \varepsilon + \zeta).$$

There are two algorithms we can leverage here.

1. **Forward-mode differentiation.** Start at an input to the graph, and move towards the end. Sum all the paths feeding in. The operator here is $\frac{\partial}{\partial X}$; similar to standard calculus.

$$\begin{aligned}\frac{\partial X}{\partial X} &= 1\\ \frac{\partial Y}{\partial X} &= \alpha + \beta + \gamma\\ \frac{\partial Z}{\partial X} &= (\alpha + \beta + \gamma)(\delta + \varepsilon + \zeta).\end{aligned}$$

2. **Reverse-mode differentiation.** Start at an output of the graph, and move towards the beginning. At each node, merge all paths which started at that node. The operator here is $\frac{\partial Z}{\partial}$.

   In particular:

$$\begin{aligned}\frac{\partial Z}{\partial Z} &= 1\\ \frac{\partial Z}{\partial Y} &= \delta + \varepsilon + \zeta\\ \frac{\partial Z}{\partial X} &= (\alpha + \beta + \gamma)(\delta + \varepsilon + \zeta)\end{aligned}$$

What is the difference between forward and reverse mode differentiation? "Forward-mode differentiation tracks how one input affects every node." "Reverse-mode differentiation tracks how every node affects one output."

[1] Image source: Chris Olah.

$$\texttt{forward mode: } \frac{\partial}{\partial X}$$
$$\texttt{reverse mode: } \frac{\partial Z}{\partial}$$

Reverse mode differtiation gives us the derivative of the output w.r.t. every node. This is exactly what we want.

On a large computational graph, this means reverse mode differentiation can get them all in one fell swoop.

In summary: derivatives are ridiculously computationally cheap.

2

CS236: Deep Generative Models

2.1 Variational Autoencoder

- Observations: $\mathbf{x} \in \{0, 1\}^d$.
- Latent variables $\mathbf{z} \in \mathbb{R}^k$.
- Goal: learn a latent variable model that satisfies

$$\begin{aligned} p_\theta(\mathbf{x}) &= \int p_\theta(\mathbf{x}, \mathbf{z})\, d\mathbf{z} \\ &= \int p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})\, d\mathbf{z}. \end{aligned}$$

In particular, the VAE is defined by the following generative process:

$$\begin{aligned} p(\mathbf{z}) &= \mathcal{N}(\mathbf{z}|0, I) \\ p(\mathbf{x}|\mathbf{z}) &= \text{Ber}(\mathbf{x}|f_\theta(\mathbf{z})), \end{aligned}$$

where $f_\theta(\mathbf{z})$ is a neural network decoder to obtain the parameters of the d Bernoulli random variables which model the pixels in each image.

For inference, we want good values of the latent variables given observed data (that is, $p(\mathbf{z}|\mathbf{x})$.

Indeed, by Bayes' theorem, we can write

$$\begin{aligned} p(\mathbf{z}|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})} \\ &= \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{\int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\, dz}. \end{aligned}$$

We want to maximize the marginal likelihood $p_\theta(\mathbf{x})$, but the integral over all possible $\mathbf{z}$ is intractable. Therefore, we use a variational approximation to the true posterior.

We write

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\mu_\phi(\mathbf{x}), \text{diag}(\sigma^2_\phi(\mathbf{x}))).$$

Variational inference approximates the posterior with a family of distributions $q_\phi(\mathbf{z}|\mathbf{x})$.

To measure how well our variational posterior $q(\mathbf{z}|\mathbf{x})$ approximates the true posterior $p(\mathbf{z}|\mathbf{x})$, we can use the KL-divergence.

The optimal approximate posterior is

$$\begin{aligned} q_\phi(\mathbf{z}|\mathbf{x}) &= \text{argmin}_\phi \, KL(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) \\ &= \text{argmin}_\phi \, \mathbb{E}_q \left[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p(\mathbf{x}, \mathbf{z})\right] + \log p(\mathbf{x}). \end{aligned}$$

But this is impossible to compute directly, since we end up getting $p(\mathbf{x})$ in the divergence.

We then maximize the lower bound to the marginal log-likelihood:

$$\begin{aligned} \log p_\theta(\mathbf{x}) &\geq \text{ELBO}(\mathbf{x}; \theta, \phi) \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \end{aligned}$$

And this ELBO is tractable, so we can optimize it.

2.1.1 Reparametrization trick

Instead of sampling

$$z \sim \mathcal{N}(\mu, \Sigma),$$

we can sample

$$\begin{aligned} &z = \mu + L\epsilon; \\ &\epsilon \sim \mathcal{N}(0, I); \Sigma = LL^T \end{aligned}$$

Allows for low variance estimates.

2.1.2 GMVAE

Same set up as vanilla VAE, except the prior is a mixture of Gaussians. That is,

$$p_\theta(\mathbf{x}) = \sum_{i=1}^{k} \frac{1}{k} \mathcal{N}(\mathbf{z}|\mu_i, \text{diag}(\sigma_i^2))$$

However, the KL term cannot be computed analytically between a Gaussian and a mixture of Gaussians. We can obtain an unbiased estimator, however:

$$\begin{aligned} D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) &\approx \log q_\phi(\mathbf{z}^{(1)}|\mathbf{x}) - \log p_\theta(\mathbf{z}^{(1)}) \\ &= \log \mathcal{N}(\mathbf{z}^{(1)}|\mu_\phi(\mathbf{x}), \text{diag}(\sigma_\phi^2(\mathbf{x}))) - \log \sum_{i=1}^{k} \frac{1}{k} \mathcal{N}(\mathbf{z}^{(1)}|\mu_i, \text{diag}(\sigma_i^2)). \end{aligned}$$

2.1.3 IWVAE

The ELBO bound may be loose if $q_\phi(\mathbf{z}|\mathbf{x})$ is a poor approximation to $p_\theta(\mathbf{z}|\mathbf{x})$. For a fixed $\mathbf{x}$, the ELBO is, in expectation, the log of the unnormalized density ratio

$$\frac{p_\theta(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} = \frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} p_\theta(\mathbf{x}),$$

where $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$.

1. Prove that IWAE is a valid lower bound of the log-likelihood.

The parentheses here are broken

$$\log p_\theta(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(m)} \sim q_\phi(\mathbf{z}|\mathbf{x})} \left[\log \frac{1}{m} \sum_{i=1}^{m} \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(i)})}{q_\phi(\mathbf{z}^{(i)}|\mathbf{x})}\right]$$

$$\geq \mathbb{E}_{z^{(1)} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log \frac{p_\theta(\mathbf{x}, \mathbf{z}^{(1)})}{q_\phi(\mathbf{z}^{(1)}|\mathbf{x})}$$

Jensen states that for convex functions, $\mathbb{E} f[X] \geq f\mathbb{E}[X]$. log is concave. So

2.1.4 Questions

- Why is the reparametrization trick lower variance? (Asked on Piazza.)

3

CS168: The Modern Algorithmic Toolbox

Contents

3.1 Lecture 3

Administrative updates:

- Mini project 1: due 11:59pm tomorrow
- Mini project 2: posted tonight (due in 8 days)

Core problem. How can we quickly find similar datapoints?
Two variations on this problem:

- One: given a dataset, search for similar pairs within the dataset.
- Two: given a new datapoint, quickly process the query and find points that are similar (nearest neighbor search problem).

Question. How do we define "similarity?"
Motivation / applications.

- Similarity for e.g. documents, webpages, source code. Search engines, for instance, perform a lot of deduplication to ensure that results are not repeated twice.
- Collaborative filtering (think Amazon / Netflix for recommendations). Idea: compute which individuals are similar, or compute which movies / items are similar.
- Machine learning via nearest neighbor search / similarity.

3.1.1 Similarity Metrics

Jacard Similarity. This is a notion that applies between sets / multi-sets S and T.

$$J(S,T) = \frac{|S \cap T|}{|S \cup T||}.$$

In the case of multisets - just count things with redundancy.
Example. Say $S = \{a, b, c, d, e\}$, and $T = \{a, e, f, g\}$, then the Jacard similarity is

$$J(S,T) = \frac{2}{7}.$$