CS182
Deep Learning
Notes

Instructor: Marvin Zhang

Kelvin Lee

UC Berkeley

Contents

1

Machine Learning Basics

1.1 Machine Learning Method

1. Define your **model**.
2. Define your **loss function**.
3. Define your **optimizer**.
4. Run it on a big GPU.

1.1.1 Maximum Likelihood Principle

Given data $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$. Assume a set (family) of distributions on (x, y).

$$\begin{aligned}
\theta_{\mathsf{MLE}} &= \arg\max_{\theta \in \Theta} p(\mathcal{D} \mid \theta) \\
&= \arg\max_{\theta \in \Theta} \prod_{i=1}^{N} p(x_i) p_\theta(y_i \mid x_i) \\
&= \arg\max_{\theta \in \Theta} \sum_{i=1}^{N} \log p(x_i) + \log p_\theta(y_i \mid x_i) \\
&= \arg\max_{\theta \in \Theta} \sum_{i=1}^{N} \log p_\theta(y_i \mid x_i) \\
&= \arg\min_{\theta \in \Theta} \sum_{i=1}^{N} -\log p_\theta(y_i \mid x_i) \\
&= \arg\min_{\theta \in \Theta} \sum_{i=1}^{n} \ell(\theta; x_i, y_i).
\end{aligned}$$

1.1.2 Cross-entropy loss

Definition 1.1.1 (Cross-entropy loss)**.**

$$H(p, q) = -\sum_x p(x) \log q(x) = \mathbb{E}_p[-\log q(x)].$$

Let's plug in p_{data} (true data distribution) for p and p_θ for q:

$$\begin{aligned} H(p_{\text{data}}, p_\theta) &= \mathbb{E}_{p_{\text{data}}}[-\log p_\theta(x, y)] \\ &= \mathbb{E}_{p_{\text{data}}}[-\log p(x) - \log p_\theta(y \mid x)]. \end{aligned}$$

1.1.3 Optimization techniques

Gradient-based optimization:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; x_i, y_i)$$

Example 1.1.1 (Logistic Regression)**.** Given $x \in \mathbb{R}^d$, define $f_\theta(x) = \theta^\top x$, where θ is a $d \times K$ matrix. Then for class $c \in \{0, \ldots, K-1\}$, we have

$$p_\theta(y = c \mid x) = \text{softmax}(f_\theta(x))_c.$$

The loss function is

$$\ell(\theta; x, y) = -\log p_\theta(y \mid x).$$

Optimization:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; x_i, y_i).$$

1.2 Empirical Risk

Question. How do we determine whether we are satisfied with the model?

Definition 1.2.1 (Risk)**.** **Risk** is defined as expected loss:

$$R(\theta) = \mathbb{E}[\ell(\theta; x, y)].$$

It is sometimes called **true risk** to distinguish from empirical risk defined below.
Empirical risk is the average loss on the training set:

$$\hat{R}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; x_i, y_i).$$

Supervised learning is oftentimes **empirical risk minimization (ERM)**.

Question. Is this the same as true risk minimization? The empirical risk looks like a **Monte Carlo estimate** of the true risk, so shouldn't $\hat{R}(\theta) \approx R(\theta)$ Why might this not be the case?

- The issue here is that we are already using the training dataset to learn θ. We can't reuse the same data to then get an estimate of the risk.
- When the empirical risk is low, but the true risk is high, we are **overfitting**.
- When the empirical risk is high, but the true risk is also high, we are **underfitting**.
- Generally, the true risk won't be lower than the empirical risk.

1.2.1 Overfitting and underfitting

- Overfitting happens usually when the dataset is too small and/or the model is too "powerful".
- Underfitting happens usually when the model is too "weak" and/or the optimization doesn't work well (i.e., the training loss does not decrease satisfactorily)

1.2.2 Model class and capacity

Definition 1.2.2 (Model class)**.** **Model class** refers to the set of all possible functions that the chosen model can represent via different parameter settings. For example, the set of all linear functions.

Definition 1.2.3 (Capacity)**.** The **capacity** of a model (class) is a measure of how many different functions it can represent.

2

Optimization

2.1 Stochastic Gradient Descent

Computing $\nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \boldsymbol{x}_i, y_i)$ every iteration for large N is a bad idea. Thus, we use **SGD**:

- Pick a **batch size** (**mini batch size**) $B << N$, randomly sample $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_B, y_B\}$ from the training data and compute
$$\nabla_\theta \frac{1}{B} \sum_{i=1}^{B} \ell(\theta; \boldsymbol{x}_i, y_i).$$
- Sampling the mini batch i.i.d. is rather slow due to random memory accesses.
- Instead, we *shuffle* the dataset and construct mini batches from consecutive data points.
- After each pass on the training data (**epoch**), we reshuffle.

2.2 Learning rate adjustment

- Commonly, a learning rate **schedule** will be used rather than a constant.
- **Linear decay** decreases the learning rate a constatn amount each iteration:
$$\alpha_i = \alpha_0 \left(1 - \frac{i}{max\_steps}\right)$$
- **Cosine annealing** decays the learning rate according to :
$$\alpha_i = \alpha_0 \cdot 0.5 \left[1 + \cos\left(\pi \cdot \frac{i}{max\_steps}\right)\right].$$

2.3 Momentum

Intuitively, we want the optimization to remember the gradient steps it has taken.

- We do so by modifying the update rule:
$$\theta \leftarrow \theta - \alpha \boldsymbol{g}$$

- Before $\boldsymbol{g} = \nabla_\theta \frac{1}{N} \sum_i \ell(\theta; \boldsymbol{x}_i, y_i)$. Now,

$$\boldsymbol{g} \leftarrow \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \boldsymbol{x}_i, y_i) + \mu \boldsymbol{g}$$

- This is an example of an *exponential moving average*: gradients further in the past have exponentially less weight.

2.4 Nesterov's accelerated gradient

- **Nesterov's accelerated gradient** is another optimization approach which enjoys interesting theoretical guarantees on some problems
- It can be interpreted as a variant on the momentum approach we described.
- The difference is

$$\boldsymbol{g} \leftarrow \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta + \mu \boldsymbol{g}; \boldsymbol{x}_i, y_i) + \mu \boldsymbol{g}.$$

- It looks ahead to see if the choice of direction is a good idea.

2.5 Gradient directions vs magnitudes

- The sign of the gradient is useful for telling us which direction to move in.
- However, the magnitude of the gradient is not as useful/trustworthy.
  - We may have loss landscapes that are not sufficiently smooth.
  - Gradient magnitudes also tend to start out large and end up very small.
- Normalizing the gradient magnitudes along each dimension can lead to an effective optimization strategy.

2.6 Adam

Basic idea: combine momentum with a second moment adjustment.

$$\theta \leftarrow \theta - \alpha \boldsymbol{g}.$$

Define momentum m such that

$$m \leftarrow (1 - \beta_1) \nabla_\theta \ell + \beta_1 m.$$

Second moment estimate:

$$v \leftarrow (1 - \beta_2)(\nabla_\theta \ell)^2 + \beta_2 v.$$

Bias correction:

$$\hat{m} = \frac{m}{1-\beta_1^t}$$
$$\hat{v} = \frac{v}{1-\beta_2^t}.$$

Then

$$\boldsymbol{g} = \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

Question. What's so great about Adam?

- Empirically, Adam seems to work well out of the box for many neural networks.
- It combines momentum with a cheap approximation of second order information —actual second order methods like Newton's method are far too expensive.
- There's also some relationship to methods which adapt the learning rate separately for each parameter.

2.7 Weight decay vs ℓ_2-regularization

- Remember that adding $\lambda\|\theta\|_2^2$ to the loss function is ℓ_2-regularization.
- Weight decay is an extra step in the optimization: after taking a gradient step, we do $\theta \leftarrow (1-\lambda)\theta$ (shrinking the parameters toward zero).
- For stochastic gradients, ℓ_2-regularization and weight decay are the same.

2.8 Tuning the optimization

- $\alpha_0 = 0.001$ is a good number to start from but this usually requires tuning.
- A useful rule-of-thumb: if some α_0 is good for some B, then $k\alpha_0$ is often a good value for kB.
- $\mu = 0.9$ is a good default value for momentum.
- $\beta_1 = 0.9, \beta_2 = 0.999, e = 10^{-8}$ for Adam usually don't require tuning.