Lecture 4: Optimization

CS 182 Spring 2021 – Taught by Sergey Levine

Notes by Michael Zhu

Gradient Descent

General Algorithm:

1. Find a direction v where $L(\theta)$ decreases
2. Update $\theta \leftarrow \theta + \alpha v$

$$\text{Loss func: } L(\theta) = -\sum_i \log p_\theta(y_i|x_i)$$

$$\text{Optimization objective: } \theta^* \leftarrow \arg\min_\theta L(\theta)$$

For each dimension, $L(\theta)$ decreases in the direction **opposite the slope**

We can compute the slope at each direction by taking the **partial derivative of the loss** with respect to each parameter $dL(\theta)/\theta_i$

This is also known as the **gradient**:

$$\nabla_\theta L(\theta) = \begin{pmatrix} dL(\theta)/d\theta_1 \\ dL(\theta)/d\theta_2 \\ \ldots \\ dL(\theta)/d\theta_n \end{pmatrix}$$

Issues with Gradient Descent

Gradient descent often works well for optimizing **convex** loss functions. Negative log likelihood loss for logistic regression is an example of a convex loss function.

However, the loss surface of a neural network is usually **non-convex**.

Local optima

This is the most obvious issue with non-convex loss landscapes.

Gradient descent could converge to a solution that is **arbitrarily worse** than the global optimum.

Surprisingly, this local optima are not a huge issue for large networks. There is a lot of evidence that shows local optima in big neural nets are usually not that much worse than the global optima.[1]

Plateaus

If your algorithm reaches a **plateau** with a **small learning rate**, it could get **stuck**.

Momentum helps us deal with plateaus.

[1]The Loss Surfaces of Multilayer Networks by Choromanska et al. https://arxiv.org/pdf/1412.0233.pdf

Saddle Points

Definition: a point that is a **local minimum** in some dimensions and a **local maximum** in other dimensions

Gradients become very small (aka **vanish**) at saddle points; similar effect as plateaus that **slows** down gradient descent

Very common for high dimensional data

Critical points: where gradient is zero $\nabla_\theta L(\theta) = 0$; can be a **maximum**, **minimum**, or **saddle**

For **2-dimensions**, saddle points don't exist; you can figure out if a critical point is a max or min using the **second derivative**

- **local maximum**: $d^2L/d\theta^2 < 0$
- **local minimum**: $d^2L/d\theta^2 > 0$

In **higher dimensions**, we can use the **Hessian matrix** to classify critical points

$$\begin{pmatrix} \frac{\partial^2 L}{\partial\theta_1\partial\theta_1} & \cdots & \frac{\partial^2 L}{\partial\theta_1\partial\theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial\theta_n\partial\theta_1} & \cdots & \frac{\partial^2 L}{\partial\theta_n\partial\theta_n} \end{pmatrix}$$

At a critical point, the **Hessian matrix** will be a **diagonal matrix**

- **maximum**: **all** diagonal entries are **positive**
- **minimum**: **all** diagonal entries are **negative**
- **saddle**: diagonal entries have **mismatched** signs

At higher dimensions, it's pretty **unlikely** for all the diagonal entries to have the same sign. This tells us that **saddle points** are the **most common** critical points at higher dimensions.

Momentum

Gradient descent always moves in the direction of **steepest descent**

However, this is **NOT** always the most optimal direction (imagine a **zig-zag** in a valley shaped loss surface; its highly repetitive and slow)

Momentum is a **heuristic** that speeds up gradient descent by **adding past gradient steps**

$$\text{Update rule: } \theta_{k+1} = \theta_k - \alpha g_k$$

$$\text{Without momentum: } g_k = \nabla_\theta L(\theta_k)$$

$$\text{With momentum: } g_k = \nabla_\theta L(\theta_k) + \mu g_{k-1}$$

Possible scenarios (for intuition, not guaranteed to occur):

1. **Gradient steps zig-zag**: momentum causes "zigging" directions to **cancel** each other out and amplifies the magnitude of the **aggregate direction**
2. **Gradient steps DON'T zig-zag**: momentum causes gradient steps to **accelerate towards current direction** (useful for **plateaus**)

Note: momentum does not have many theoretical guaratees but it's a heuristic that generally works well in practice with virtually **no added computational cost**

Note: can look up "Nesterov accelerated gradient" which is a similar idea that **does** have some appealing guaratees in convex optimization, but for neural nets in practice we just use momentum

RMSProp

Issue: gradients can have very extreme scaling (i.e. super small or super big) based on the difference between $f_\theta(x)$ and y

Solution: we can **normalize** the magnitude along each dimension

$$s_k \leftarrow \beta s_{k-1} + (1-\beta)(\nabla_\theta L(\theta_k))^2$$
$$\theta_{k+1} = \theta_k - \alpha \frac{\nabla_\theta L(\theta_k)}{\sqrt{s_k}}$$

s_k is essentially a **running average** of the magnitude of gradient at dimension k

Note: we need to square $\nabla_\theta L(\theta_k)$ to get magnitude so it's not affected by direction

Like **momentum**, RMSProp works pretty well in practice but doesn't have many theoretical guarantees

AdaGrad

$$s_k \leftarrow s_{k-1} + (\nabla_\theta L(\theta_k))^2$$
$$\theta_{k+1} = \theta_k - \alpha \frac{\nabla_\theta L(\theta_k)}{\sqrt{s_k}}$$

Compared to RMSProp, AdaGrad is the **cumulative magnitude** per dimension instead of a running average

- Can be proven to work well for **convex** problems
- The learning rate $\alpha/\sqrt{s_k}$ decreases over time, which is good for convex problems
- Only works for nonconvex problems if the learning algorithm can find the optimum region quickly before learning rate decays too much
  - RMSProp works better for **nonconvex** because it effectively "forgets" past gradients while AdaGrad never "forgets"

Adam

Idea: use both momemtum and RMSProp

$$m_k = (1-\beta_1)\nabla_\theta L(\theta_k) + \beta_1 m_{k-1} \quad \text{first moment estimate (similar to "momentum")}$$
$$v_k = (1-\beta_2)(\nabla_\theta L(\theta_k))^2 + \beta_2 v_{k-1} \quad \text{second moment estimate (similar to "RMSProp")}$$
$$\hat{m_k} = \frac{m_k}{1-\beta_1^k}$$
$$\hat{v} = \frac{v_k}{1-\beta_2^k}$$
$$\text{Update step: } \theta_{k+1} = \theta_k - \alpha \frac{\hat{m_k}}{\sqrt{\hat{v_k}} + \epsilon}$$

Popular default hyperparameter values:

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$

m_k and v_k start at 0 so will be numerically small for the first few iterations

$\hat{m_k}$ and $\hat{v_k}$ are used to speed up the first few iterations by dividing by $1-\beta^k$

Stochastic Gradient Descent

Regular gradient descent is very expensive for large datasets

$$L(\theta) = -\frac{1}{N}\sum_{i}^{N} \log p_\theta(y_i|x_i)$$

Each gradient step requires summing over **all datapoints** in the dataset

Idea: break the dataset into seperate **batches** of size **B** which we use for each gradient step. This works if each batch is **randomly sampled** because they are an **unbiased approximation** of the expected loss

$$L(\theta) = -\frac{1}{N}\sum_{i}^{N} \log p_\theta(y_i|x_i) \approx -E_{p(x_i,y_i)}[\log p_\theta(y_i|x_i)] \approx -\frac{1}{B}\sum_{j}^{B} \log p_\theta(y_{i_j}|x_{i_j})$$

SGD with minibatches procedure

1. Sample $B \subset D$
2. Estimate $g_k \leftarrow -\nabla_\theta \sum_i^B \log p_\theta(y_i|x_i) \approx \nabla_\theta L(\theta)$
3. Update $\theta_{k+1} \leftarrow \theta_k - \alpha g_k$; apply momentum, ADAM, RMSProp, etc.

Each iteration uses a different **minibatch**

Every pass through the entire dataset is an **epoch**

For efficient memory access:

- In practice, we randomly shuffle the dataset once at the start and then index into each minibatch sequentially (recall *principle of locality*)
- Batch sizes B are generally powers of two $\rightarrow$ 1, 2, 4, 8, 16, 32, 64, 128, ...

Tuning SGD

Hyperparameters

- batch size B: larger batches $\rightarrow$ less noise in gradients but more memory expensive
- learning rate α: ideally use largest learning rate that doesn't diverge; decay over time if necessary
- momentum μ: 0.99 is good default
- Adam $\beta_1,\ \beta_2$: 0.9 and 0.999 are good defaults usually

Technically, we want to tune these **optimization hyperparameters** on the **training loss**

In practice, people often tune on **validation loss** as well (recall validation loss is usually used for **regularization hyperparameters**)

Stochastic gradient descent can sometimes have a regularizing effect since each batch keeps the other batches from overfitting

Learning rate decay

- Learning rate too high $\rightarrow$ learning algorithm either diverges or keeps jumping around the optimum
- Learning rate too low $\rightarrow$ increases training time and can get stuck on plateaus

Idea: **Start with a high learning rate** that doesn't diverge (avoid plateaus and quickly get near optimum); **gradually decrease** it to get closer to optimum

Learning rate decay is usually needed for best performance from SGD+momentum

Often not needed with ADAM