Efficient Deep Learning
Optimization Methods

Josh Moavenzadeh, Kai Hu,
and Cody Smith
Outline

• 1 Review of optimization
• 2 Optimization practice
• 3 Training tips in PyTorch
1.1 Mini-batch gradient descent

• What is it?
  • Performs update for every mini-batch of data.

• Why mini-batch?
  • Batch gradient descent that uses the whole dataset for one update: slow and intractable for large datasets to fit into memory.
  • Stochastic gradient descent that updates for each data: high variance updates.
1.1 Mini-batch gradient descent (continue)

• Update equation
  • Let $F$ be our model, and $\theta$ is the parameter: $\hat{y} = F(x; \theta)$
  • The loss function is $L$, minimize the loss on the dataset:

$$g = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i)$$

• Let $\eta$ be the learning rate, compute the update:

$$\hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\theta} L(y_i, \hat{y}_i), \quad \theta = \theta - \eta \cdot \hat{g}$$
1.1 Mini-batch gradient descent (Continue)

- The good things of mini-batch gradient descent
  - Reduces variance of updates
  - Matrix multiplication is faster
- Have to decide mini-batch size now!
  - The common mini-batch size are 32-256.
  - Too small: Slow and high variance,
    Batch Norm requires a suitable batch size
  - Too big: Harder to escape from local minima.
    Decay in generalization (paper link).
1.1 Mini-batch gradient descent (Continue)

The figure shows why big batch size is not OK:

1.2 Momentum

• SGD has trouble navigating ravine. Momentum helps SGD accelerate.

• Adds a fraction $\gamma$ of the update vector of the past step $V_{t-1}$ to current update vector $V_t$. Momentum term $\gamma$ is usually set to 0.9.

• Update: $v_t = \gamma v_{t-1} + \eta \nabla_{\theta} L(\theta); \quad \theta = \theta - v_t$

• Example: find the minima of $z = x^2 + 50y^2$
1.2 Momentum (Continue)

- Reduces updates for dimensions whose gradients change directions.
- Increases updates for dimensions whose gradients point in the same directions.
1.3 Nesterov accelerated gradient (NAG)

• The moment uses history information for better update. NAG wants to add some future information.

• Update:

\[ v_t = \gamma v_{t-1} + \eta \nabla \theta L(\theta - \gamma v_{t-1}); \quad \theta = \theta - v_t \]

**A picture of the Nesterov method**

- **First** make a big jump in the direction of the previous accumulated gradient.
- **Then** measure the gradient where you end up and make a correction.

brown vector = jump, \quad red vector = correction, \quad green vector = accumulated gradient

blue vectors = standard momentum
1.4 Adagrad

• The previous methods: same learning rate for all parameters.
• Adagrad adapts the learning rate to the parameters
  large updates for infrequent parameters
  small updates for frequent parameters
• Adagrad divides the learning rate by the square root of the sum of squares of historic gradients.
• Update:

\[ r_t = \sum_{i=1}^{t} g_i^2, \quad \theta = \theta - \frac{\eta}{\sqrt{r_t + \epsilon}} \ast g_i \]

\( g_t \) is the sum of the squares of the gradients.

* is element-wise multiplication.
1.4 Adagrad (Continue)

• Pros
  1) Good when dealing with sparse data.
  2) Lesser need to manually tune learning rate.

• Cons
  Recall that the update is:

\[
\theta = \theta - \frac{\eta}{\sqrt{\sum_{i=1}^{t} g_i^2 + \epsilon}} \ast g_i
\]

Accumulates squared gradients in denominator.
Causes the learning rate to shrink and become infinitesimally small.
1.5 Adadelta

• In adagrad, the learning rate may become infinitesimally small.
• Adadelta was designed to solve this problem. It replaces the denominator by the running average of squared gradients:

\[ \mathbb{E}[g^2]_t = \gamma \mathbb{E}[g^2]_{t-1} + (1 - \gamma)g_i^2 \]

• Preliminary Adadelta update (Also named RMSprop):

\[ \theta = \theta - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} * g_i \]

• Compare with adagrad:

\[ \theta = \theta - \frac{\eta}{\sqrt{\sum_{i=1}^{t} g_i^2 + \epsilon}} * g_i \]
1.5 Adadelta (Continue)

- Denominator is called root mean squared (RMS) error of gradient, we can write the update as:

  \[ \Delta \theta_t = -\eta \frac{g_t}{RMS[g]_t} \]

- The units do not match!

- Define the running average of squared parameter updates and RMS:

  \[ \mathbb{E}[\Delta \theta^2]_t = \gamma \mathbb{E}[\Delta \theta^2]_{t-1} + (1 - \gamma) \Delta \theta_t^2 \]

- Now we can replace \( \eta \) with \( RMS[\theta]_{t-1} \) for the final update:

  \[ \Delta \theta_t = -RMS[\theta]_{t-1}\frac{g_t}{RMS[g]_t} \]
1.6 Adam

- Now we have two kinds of ideas for improving SGD:
  1) Momentum and Nesterov: use more gradients
  2) Adagrad and Adadelta: different LR for different parameters.
- Combine the two ideas. Adam!
- Update. First store the mean and uncentered variance of gradients:

\[ m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \]
\[ v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \]

- \( m_t \) is the running mean of gradients and \( v_t \) is the running uncentered variance of gradients
1.6 Adam (Continue)

- \( m_t \) and \( v_t \) are initializes as zero vectors. So they are biased estimation and we want to correct them as:

\[
\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}
\]

- The update rule is:

\[
\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t
\]

- Question: Adam looks like RMS with Momentum, what are the differences?
2.1 Parameter Initialization

• Can we start with zero initial weights?
• Can we have equal initial weights?
• Methods to initialize
  • Random (typically gaussian)
  • Xavier
  • He initialization with ReLU
  • Pretraining
2.1 Parameter Initialization (Continue)

• Xavier: Uniform distribution from [-a, a].
• You want the variance of Input and Output to be the same:

\[ y = \sum_{i=1}^{n} w_i x_i \]

• If you work out the math, \( \text{Var}(w_i) = \frac{1}{n} \)
• But you do not have only one output, you may have \( m \) outputs:

\[ \text{var}(\text{distribution from } [a, b]) = \frac{2}{n + m} \]

• The variance of Uniform distribution from \([a, b]\) is \( \frac{(b - a)^2}{12} \)
• Solve for \( a \)
2.1 Parameter Initialization (Continue)

• He Initialization for ReLU
  Uniform distribution from \([-a, a]\).

• About half output will return zero after ReLU
  
  \[ y = ReLU(\sum_{i=1}^{n} w_i x_i) \]

• The variance changes to \(\text{Var}(w_i) = 2/n\)
• Re-solve for \(a\)
2.1 Parameter Initialization (Continue)

• He Initialization for ReLU
  Uniform distribution from \([-a, a]\).

• About half output will return zero after ReLU
  \[ y = ReLU(\sum_{i=1}^{n} w_i x_i) \]

• The variance changes to \( \text{Var}(w_i) = 2/n \)
• Re-solve for \( a \)
2.2 Annealing the learning rate

• Usually helpful to anneal the learning rate over time

• High learning rates can cause the parameter vector to bounce around chaotically, unable to settle down into deeper, but narrower parts of the loss function

• **Step decay**: Reduce the learning rate by some factor after some number of epochs (i.e. reduce by a half every 5 epochs, or by 0.1 every 20 epochs).

• **Plateau decay**: Watch the validation error or loss while training with a fixed learning rate, and reduce the learning rate by a constant factor whenever the validation performance stops improving

• **Exponential decay**: It has the mathematical form \( lr = lr_0 \cdot e^{(-kt)} \), where \( lr_0, k \) are hyperparameters and \( t \) is the iteration number
2.2 Annealing the learning rate (Continue)

• Learning rate schedulers in PyTorch

  • `torch.optim.lr_scheduler.<StepLR|ExponentialLR|ReduceLROnPlateau>`
  
  • Each type of scheduler requires hyperparameters unique to it on initialization – read the docs

  • `scheduler.step(val_loss)`
    
    • At end of each epoch – maintains history of epoch loss to determine when to decay the learning rate
2.3 Random Dropout

• Implementation
  Dropout each unit with probability \( p \)
  No parameters dropped at test time

• Results
  Network is forced to learn a distributed representation
  Improves generalization by eliminating neuron co-dependencies within a layer

• In PyTorch
  \texttt{nn.Dropout(p = \_)}
  Typical dropout probability is around 0.1 to 0.5
2.4 Others:

• Shuffle the dataset
  If not shuffle, the network will remember the data order!
  In hw1p2, it is a frame-level task, so you need to shuffle in frames.

• Weight decay:
  L2 regularization for (not) overfitting:

\[
loss = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \frac{1}{2} w\theta^2
\]

\[
\theta = \theta - \Delta \theta - w\theta
\]

• Early Stopping for (not) overfitting