Optimization for neural networks

Toby Dylan Hocking

Supervised machine learning

- Goal is to learn a function $f(\mathbf{x}) = y$ where $\mathbf{x} \in \mathbb{R}^p$ is an input/feature vector and y is an output/label.
- $\mathbf{x} =$image of digit/clothing, $y \in \{0, \dots, 9\}$ (ten classes).
- $\mathbf{x} =$vector of word counts in email, $y \in \{1, 0\}$ (spam or not).

Optimization algorithms vs machine learning algorithms

- In the field of optimization, the ultimate goal is to minimize some function (which we can compute).
- In supervised machine learning our ultimate goal is to minimize prediction error on test set (which we can not compute).
- Instead we must assume train and test data are similar, and then minimize train error.
- Instead of directly minimizing the error function of interest (zero-one loss) our gradient descent algorithms attempt to minimize a differentiable surrogate loss (logistic / cross-entropy).
- But our goal has now been twice modified, so we need regularization in addition to optimization.

Basic gradient descent algorithm

- m is batch size.
- i is observation number in batch.
- $x^{(i)}, y^{(i)}$ are inputs/output.
- θ is full set of neural network parameters (weights + bias).
- Gradient computed via

$$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L[f(x^{(i)};\theta), y^{(i)}].$$

Parameters updated for each iteration via

$$\theta \leftarrow \theta - \epsilon g.$$

- $\epsilon > 0$ is step size = learning rate.

Decreasing learning rate schedule

Parameters updated for each iteration via learning rate ϵ_k which decreases with the number of iterations k.

$$\theta \leftarrow \theta - \epsilon_k g.$$

To avoid overshooting minima, learning rate decays/decreases until iteration τ, after which it is held constant.

$$\epsilon_k = (1 - a)\epsilon_0 + a\epsilon_\tau.$$

- $a = k/\tau$, $\tau \approx$ a few hundred epochs, $\epsilon_\tau/\epsilon_0 \approx 0.01$.
- You can implement this by instantiating an optimizer in each epoch, with a different learning rate that depends on the epoch number.
- `torch.optim.lr_scheduler` contains many classes which implement learning rates schedules.
- For example `OneCycleLR(anneal_strategy="linear")` is similar to the linear/additive rule described above, `StepLR` is multiplicative.

Momentum

- Accumulates an exponentially decaying moving average of past gradients.
- Depends on degree of momentum parameter $\alpha \in [0, 1)$.
- For which value of α do we recover the usual gradient descent? (no momentum)
- Common values 0.5, 0.9, 0.99.

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L[f(x^{(i)}; \theta), y^{(i)}] \right) = \alpha v - \epsilon g.$$

$$\theta \leftarrow \theta + v.$$

- Implement via `torch.optim.SGD(momentum=0.5)` etc.

Adagrad algorithm: adapting to gradients

- Adaptive learning rate for each model parameter by scaling them proportionally to the square root of the sum of squares of gradients over entire history.
- Greater progress in more gently sloped directions of parameter space.
- Hyper-parameters are global step size $\epsilon > 0$ and numerical stability constant $\delta \approx 10^{-7}$.
- Implement via `torch.optim.Adagrad`

$$g \leftarrow \frac{1}{m} \sum_i L[f(x^{(i)}; \theta), y^{(i)}], \quad r \leftarrow r + g \odot g,$$

$$\Delta\theta \leftarrow \frac{-\epsilon}{\delta + \sqrt{r}} \odot g, \quad \theta \leftarrow \theta + \Delta\theta.$$

RMSProp algorithm (root mean squared propagation)

- Modification of Adagrad to use exponentially weighted moving average instead of total sum of squares.
- Ignores history from distant past.
- Hyper-parameters are global step size $\epsilon > 0$, decay rate ρ, and numerical stability constant $\delta \approx 10^{-7}$.
- Can be modified to use momentum (exercise for the reader).
- Implement via `torch.optim.RMSprop`

$$g \leftarrow \frac{1}{m} \sum_i L[f(x^{(i)}; \theta), y^{(i)}], \quad r \leftarrow \rho r + (1 - \rho) g \odot g,$$

$$\Delta\theta \leftarrow \frac{-\epsilon}{\sqrt{\delta + r}} \odot g, \quad \theta \leftarrow \theta + \Delta\theta.$$

Adam algorithm: Adaptive moments.

- Combines ideas from RMSprop and momentum.
- Momentum used to estimate first order moment of the gradient.
- Bias correction to first and second moment estimates based on iteration number t.
- Hyper-parameters $\epsilon > 0$ step size, $\rho_1, \rho_2 \in [0, 1)$ decay rates for moment estimates, δ for numerical stability.
- Biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) g$.
- Biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$.
- Bias correction: $\hat{s} \leftarrow s/(1 - \rho_1^t), \hat{r} \leftarrow r/(1 - \rho_2^t)$.
- Update: $\Delta\theta \leftarrow -\epsilon \hat{s}/(\delta + \sqrt{\hat{r}})$.
- Implement via `torch.optim.Adam`