Homework 3 Solution

Fall 2019

Exercises

1. Minibatch sizes are generally driven by the following factors:

   i) Larger batches provide a more accurate estimate of the gradient, but with less than linear returns

   ii) Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.

   iii) If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.

   iv) Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models. With more recent architecture that parallelize the processing of a batch over multiple GPUs, these can reach 8192.

   v) The employed optimization algorithm also influences the batch size choice. For instance, a second order method that relies on estimating the term $\boldsymbol{H}^{-1}\boldsymbol{g}$ (inverse of the cost function's Hessian times gradient) could require a large batch size if $\boldsymbol{H}$ has a poor condition number, because in this case the product could be very sensitive to the noise in the estimate of the gradient $\boldsymbol{g}$

   vi) Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Optimization algorithms that use only a single example at a time are sometimes called **stochastic** and sometimes **online** methods. The term *online* is usually reserved for when the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made. Training with such a small batch size might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient. The total runtime can be very high as a result of the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set

2. a) Computing an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent (basically, we want the probability of any sample being drawn from the dataset to be uniform). We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent minibatches of examples should also be independent from each other.

   b) For very large datasets, for example, datasets containing billions of examples in a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch. Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion. This will impose a fixed set of possible minibatches of consecutive examples that all models trained thereafter will use, and each individual model will be forced to reuse this ordering every time it passes through the training data. This deviation

from true random selection does not seem to have a significant detrimental effect. Failing to ever shuffle the examples in any way can seriously reduce the effectiveness of the algorithm.

We should remember that the above strategy is *not* common in smaller-scale training of neural networks, for instance, the general practice for training modern convolutional neural networks on (the 1000-class subset of) ImageNet usually requires reshuffling the dataset per pass over it.

3. a) Ill-conditioning of a Hessian matrix essentially means that the Hessian of the cost function has a poor condition number (a very large one). In such cases, gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change in the derivative, so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer (and directions which the gradient has small components on). Poor condition number also makes choosing a good step size difficult. The step size must be small enough to avoid overshooting the minimum and going uphill in directions with strong positive curvature. This usually means that the step size is too small to make significant progress in other directions with less curvature.

   b) Assume a second-order Taylor approximation of the cost function. Then taking a gradient descent step of $-\epsilon \boldsymbol{g}$ results in adding $\frac{1}{2}\epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g} - \epsilon \boldsymbol{g}^T \boldsymbol{g}$ to the cost. Even if we have a strong gradient, if $\frac{1}{2}\epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ exceeds $\boldsymbol{g}^T \boldsymbol{g}$, we would have an increase in the quadratically approximated cost; consequently, at least for this approximated cost, we have to decrease the learning rate to compensate for the strong curvature and ensure descent, which in turn causes slow learning.

4. a) First of all, many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher-dimensional spaces, local minima are rare, and saddle points are more common. For a function $f : \mathbb{R}^n \to \mathbb{R}$ of this type, the expected ratio of the number of saddle points to local minima grows exponentially with n. To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In a single dimension, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will be heads.

   b) As Newton's method is primarily designed to solve for a point whose gradient is zero, with a saddle point satisfying this property, the method, in its base form, could be stuck at saddle points easily. The potential abundance of saddle points in the high-dimensional optimization landscape might explain the lack of success of second-order methods like Newton's. Modifications to the method had been proposed to mitigate the above issue of Newton's method, and is called saddle-free Newton's method. If methods like this could scale to large neural networks (right now computing the Hessian or slight variants of it is very costly in computational resources for large neural networks), it is likely for (some of) the methods to outperform first-order methods.

   Gradient descent, on the other hand, is designed to minimize the cost, instead of being attracted to critical points in the optimization landscape. In practice, for (stochastic) gradient descent, researchers observed empirically that it escapes saddle points easily in many cases, and seems to spend most of the training time traversing the relatively flat valley of the cost function, perhaps because of high noise in the gradient, poor conditioning of the Hessian matrix in the local regions, etc. These observations must be taken with a grain of salt though, as they had not seen strong theoretical support so far. However, a property of many random functions is that the eigenvalues of the Hessian become more likely to be positive in regions of low cost, supporting the evidence that gradient descent would be repelled from saddle points in its pursuit of regions of low cost.

5. a) A sufficient condition to guarantee convergence of SGD is the following:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \text{ and } \sum_{k=1}^{\infty} \epsilon_k^2 < \infty \tag{1}$$

   where ϵ_k is the learning rate at iteration k.

b) Let $J : \mathbb{R}^p \to \mathbb{R}, \boldsymbol{\theta} \mapsto J(\boldsymbol{\theta})$ be the objective/cost function of the optimization. We define the **excess error** to be the function $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. For optimizing a convex cost function, the excess error is $\mathcal{O}(\frac{1}{\sqrt{k}})$ after k iterations for SGD, and it is $\mathcal{O}(\frac{1}{k})$ for strictly convex cost functions ($\mathcal{O}(\cdot)$ represents the big-O notation).

Furthermore, the Cramér-Rao bound states that generalization error cannot decrease faster than $\mathcal{O}(\frac{1}{k})$. In [1], it is hence argued that it may not be worthwhile to pursue an optimization algorithm that converges faster than $\mathcal{O}(\frac{1}{k})$ for machine learning tasks, as faster convergence presumably corresponds to overfitting. Moreover, the asymptotic analysis obscures many advantages that stochastic gradient descent has after a small number of steps. With large datasets, the ability of SGD to make *rapid initial progress* while evaluating the gradient for very few examples (note that SGD's time complexity does not scale with the size of the dataset) outweighs its slow *asymptotic* convergence.

6. a) SGD with momentum

Algorithm 1 SGD with momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$. initial velocity vector $\boldsymbol{v}$
while Stopping criterion not met **do**
  Obtain m samples from training set, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{m}$
  Estimate gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(x^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)})\right)$
  Update velocity: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
  Update parameter: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

b) The larger α is to ϵ, the greater the gradients of previous iterations affect the current gradient. Oppositely, if $\alpha \ll \epsilon$, then the current gradient dominates the gradient update of the current iteration. So, the value of α controls how fast past gradients are *forgotten*.

c) Viscous drag, in physics terms, corresponds to the force that is proportional to the negative of the velocity $v(t)$ (for simplicity we consider 1-D motion here). Part of the reason we use $-v(t)$ is mathematical convenience. Yet other physical systems have other kinds of drag based on other integer powers of the velocity. For example, a particle traveling through the air experiences **turbulent drag**, with force proportional to the **square** of the velocity, while a particle moving along the ground experiences **dry friction**, with a force of **constant** magnitude. We can reject each of these options. Turbulent drag, proportional to the square of the velocity, becomes very weak when the velocity is small. It is not powerful enough to force the particle to come to rest. A particle with a nonzero initial velocity that experiences only the force of turbulent drag can move away from its initial position forever, with the distance from the starting point growing as $\mathcal{O}(\log t)$. We must therefore use a lower power of the velocity. If we use a power of zero, representing dry friction, then the force is too strong. When the force due to the gradient of the cost function is small but nonzero, the constant force due to friction can cause the particle to come to rest before reaching a local minimum. Viscous drag avoids both of these problems - it is weak enough that the gradient can continue to cause motion until a minimum is reached, but strong enough to prevent motion if the gradient does not justify moving.

d) The only real difference between SGD with Nesterov momentum and SGD with basic momentum lies in step 4 of the algorithm below. The Nesterov variant potentially speeds up algorithm convergence by performing a "look-ahead" operation when estimating the gradient and make "corrections" to the momentum. This advantage can be easily seen by considering a simple single-variate convex quadratic cost function (in terms of θ), and suppose that the parameter at our current iteration $\theta^{(k)}$ is near the optimal point θ^*, with the velocity v pointing from $\theta^{(k)}$ toward θ^*. The base momentum SGD would push $\theta^{(k)}$ in the direction it was moving in before, as the estimated gradient g and velocity v point in the same direction toward θ^*, so we could overshoot over θ^* easily (assuming α is not overly small). On the other hand, Nesterov's gradient is estimated at the look-ahead position $\theta^{(k)} + \alpha v$, so assuming $\theta^{(k)}$ sufficiently close to θ^*, the estimated gradient then is in the opposite direction of the velocity vector, so we have smaller, or even no overshoot over θ^* in this iteration (assuming α not overly small). One should be cautious to apply the above intuition to nonconvex optimization problems such as neural network optimization, as Nesterov SGD does not necessarily converge faster or converge to similar or better optimum than does the basic momentum SGD.

Algorithm 2 SGD with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$. initial velocity vector $\boldsymbol{v}$
1: **while** Stopping criterion not met **do**
2: Obtain m samples from training set, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{m}$
3: Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
4: Estimate gradient at interim point: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\boldsymbol{x}^{(i)}, \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)}) \right)$
5: Update velocity: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
6: Update parameter: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
7: **end while**

7. a) The perspectives of regularization and optimization can give very different insights into how we should initialize a network. The optimization perspective suggests a strong symmetry-breaking effect (basically, if two units in a network are connected to the same input and initialized with the same parameters, then with a deterministic cost and deterministic learning algorithm, the two units will be updated in the same way throughout training; this is undesirable), helping to avoid redundant units. The optimization perspective may also suggest using large weights to help avoid losing signal during forward or back-propagation through the linear component of each layer.

On the other hand, some regularization concerns encourage making weights smaller. In addition to explicit weight norm regularization that encourages the weights to stay closer to the origin, the use of an optimization algorithm, such as stochastic gradient descent, that makes small incremental changes to the weights, tends to halt in areas that are nearer to the initial parameters (whether due to getting stuck in a region of low gradient, or due to triggering some early stopping criterion based on overfitting) and expresses a prior that the final parameters should be close to the initial parameters. Recall that early stopping is equivalent (or close) to weight decay for certain models with parameters initialized at (or near) the origin. Although in general, the two regularization techniques are not really equivalent and often not even close to each other, this analogy helps in thinking about the effect of initialization. We can think of initializing the parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ being similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$. From this point of view, it makes sense to choose $\boldsymbol{\theta}_0$ to be near $\mathbf{0}$. This prior says that it is more likely that units do not interact with each other than that they do interact. Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. On the other hand, if we initialize $\boldsymbol{\theta}_0$ to large values, then our prior specifies which units should interact with each other, and how they should interact.

b) Sparse initialization basically initializes the weights (for the input of each neuron) to have a pre-specified number of zeros. The idea is to keep the total amount of input to the unit independent from the number of inputs without making the magnitude of individual weight elements shrink

with the number of inputs. Sparse initialization helps to achieve more diversity among the units at initialization time. However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values. Because it can take a long time for gradient descent to shrink "incorrect" **large** values, this initialization scheme can cause problems for units, such as maxout units, that have several filters that must be carefully coordinated with each other.

8. a)

Algorithm 3 AdaGrad

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ (perhaps 10^{-7}) for numerical stability
1: Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
2: **while** Stopping criterion not met **do**
3: Obtain m samples from training set, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{m}$
4: Estimate gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$
5: Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
6: Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\epsilon/(\delta + \sqrt{\boldsymbol{r}}) \odot \boldsymbol{g}$ (division and square root applied elementwise)
7: Update parameter: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
8: **end while**

b)

Algorithm 4 RMSProp with Nesterov Momentum

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ (usually 10^{-6}) for numerical stability
1: Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
2: **while** Stopping criterion not met **do**
3: Obtain m samples from training set, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{m}$
4: Interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{v}$
5: Estimate gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\boldsymbol{x}^{(i)}, \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)}) \right)$
6: Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
7: Compute update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon/(\sqrt{\delta + \boldsymbol{r}}) \odot \boldsymbol{g}$ $(1/(\sqrt{\delta + \boldsymbol{r}})$ applied elementwise)
8: Update parameter: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
9: **end while**

c) The main difference between the above two algorithms - other than the application of Nesterov Momentum - lies in step 6 (accumulated squared gradient), where RMSProp modifies AdaGrad's gradient accumulation into an exponentially weighted moving average. AdaGrad shrinks the learning rate according to the entire history of the squared gradient (noting that the learning trajectory might have passed through many different types of landscapes) and may have made the learning rate too small before arriving at a convex structure (such as a locally convex bowl). RMSProp uses the exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

d) The name "Adam" is derived from the phrase "adaptive moments". In the context of the earlier algorithms, it is perhaps best seen as a variant of the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin. RMSProp also incorporates an estimate of the (uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

Algorithm 5 Adam

Require: Global learning rate ϵ
Require: Decay rates $\rho_1, \rho_2 \in [0, 1)$ for moment estimates
Require: Small constant δ (suggested: 10^{-8}) for numerical stability
Require: Initial parameter $\boldsymbol{\theta}$
Initialize first and second moment variables $\boldsymbol{s} = \boldsymbol{0}, \boldsymbol{r} = \boldsymbol{0}$
while Stopping criterion not met **do**
  Obtain m samples from training set, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{m}$
  Estimate gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
  Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
  Correct bias in the first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
  Correct bias in the second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
  Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\epsilon / (\delta + \sqrt{\hat{\boldsymbol{r}}}) \odot \hat{\boldsymbol{s}}$ $(1/(\delta + \sqrt{\hat{\boldsymbol{r}}})$ applied elementwise)
  Update parameter: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

9. a) We first look at how the steepest descent operates. Basically, at iteration t, we take the gradient of the cost function J w.r.t the parameter $\boldsymbol{\theta}$ at $\boldsymbol{\theta}_{t-1}$, and search in the direction of the gradient (starting from the current point $\boldsymbol{\theta}_{t-1}$) such that the cost J is minimized at $\boldsymbol{\theta}_t$ on the line. Now for notational simplicity define $\boldsymbol{d}_t$ to be the search direction of iteration t (it's the normalized $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})$). It can be shown that $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) \cdot \boldsymbol{d}_t = 0$ (to see this, think about what the line search is doing, and its optimality condition); in other words, $\boldsymbol{d}_{t-1}$ is orthogonal to $\boldsymbol{d}_t$ for every t, so we have the "zig-zag" trajectory for steepest descent. However, this means that the minimum achieved in the previous gradient direction is *not* preserved in the current line search; in a sense, we are undoing progress we have already made in the direction of the previous line search, which leads to potentially long training time. In the method of conjugate gradients, we seek to find a search direction that is *conjugate* to the previous line search direction; that is, it will not undo progress made in that direction.

   This leads to the condition that $\boldsymbol{d}_t$ must be conjugate to $\boldsymbol{d}_{t-1}$, where the two are conjugate if $\boldsymbol{d}_t^T \boldsymbol{H} \boldsymbol{d}_{t-1} = 0$, where $\boldsymbol{H}$ is the Hessian of the cost at the current iteration. Imposing this condition on the search directions helps us avoid the undoing of previous progress for the conjugate gradient method, however, such advantage is *only* theoretically guaranteed for *quadratic* cost functions. To elaborate, for a quadratic loss surface, the conjugate directions ensure that the gradient along the previous direction does not increase in magnitude. We therefore stay at the minimum along the previous directions. As a consequence, in a k-dimensional parameter space, the conjugate gradient method requires *at most* k line searches to achieve the minimum, which can be much smaller than the number of line searches performed by steepest descent.

Note: There are more general conditions on the cost function to ensure convergence of the algorithm, although they differ between the Polak-Ribière method and the Fletcher-Reeves method. Further discussion on this is beyond the scope of this course.

Algorithm 6 Conjugate Gradient Method

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of m samples, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^m$
Initialize $\boldsymbol{d}_0 = \mathbf{0}$
Initialize $\boldsymbol{g}_0 = \mathbf{0}$
Set $t = 1$
while Stopping criterion not met **do**
    Initialize gradient: $\boldsymbol{g}_t = \mathbf{0}$
    Obtain m samples from training set, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^m$
    Compute: $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^T \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^T \boldsymbol{g}_{t-1}}$ (This is the Polak-Ribière method, can also use Fletcher-Reeves)
    Compute line search direction: $\boldsymbol{d}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{d}_{t-1}$
    Perform line search: $\epsilon^* = \underset{\epsilon > 0}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^m \mathcal{L}(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}_t + \epsilon \boldsymbol{d}_t), \boldsymbol{y}^{(i)})$
    Update parameter: $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{d}_t$
    $t \leftarrow t + 1$
end while

b) Given Hessian matrix $\boldsymbol{H}$, two search directions $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$ are defined to be "conjugate" to each other if $\boldsymbol{d}_t^T \boldsymbol{H} \boldsymbol{d}_{t-1} = 0$. Imposing this condition on the line-search directions allows the conjugate gradient method to have the theoretical guarantee on quadratic-cost problems we discussed above. However, the question of how to efficiently compute β_t such that $\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) + \beta_t \boldsymbol{d}_{t-1}$ is conjugate to $\boldsymbol{d}_{t-1}$ remains. Although we could just compute the Hessian matrix to obtain β_t, such a way more or less defeats the purpose of seeking an algorithm that could scale to larger problems than second-order methods like Newton's method. Fortunately, two computationally efficient methods for computing β_t exist:

$$\begin{aligned} \beta_t &= \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \quad \text{(Fletcher-Reeves)} \\ \beta_t &= \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \quad \text{(Polak-Ribière)} \end{aligned} \tag{2}$$

10. We first discuss the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, as the limited-memory BFGS is a variant of it.

    Recall that Newton's update is given by

    $$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \boldsymbol{H}_J(\boldsymbol{\theta}_{t-1})^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}) \tag{3}$$

    with $\boldsymbol{H}_J(\boldsymbol{\theta}_{t-1})$ the Hessian of J (w.r.t $\boldsymbol{\theta}$) evaluated at $\boldsymbol{\theta}_{t-1}$. The approach adopted by quasi-Newton methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix $\boldsymbol{M}_t$ that is iteratively refined by low-rank updates to become a better approximation of $\boldsymbol{H}_J(\boldsymbol{\theta}_{t-1})^{-1}$.

    Once $\boldsymbol{M}_t$ is computed, we define the line-search direction to be $\boldsymbol{d}_t = \boldsymbol{M}_t \boldsymbol{g}_t$, and we perform line search

    $$\epsilon^* = \underset{\epsilon > 0}{\operatorname{argmin}} J(\boldsymbol{\theta}_{t-1} + \epsilon \boldsymbol{d}_t) \tag{4}$$

    and update the parameter $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \epsilon^* \boldsymbol{d}_t$.

    Like the method of conjugate gradients, the BFGS algorithm iterates a series of line searches with the direction incorporating second-order information. Unlike conjugate gradients, however, the success of the approach is not heavily dependent on the line search finding a point very close to the true minimum

along the line. Thus, relative to conjugate gradients, BFGS has the advantage that it can spend less time refining each line search. A disadvantage of the BFGS method lies in its memory requirement, as it needs to store the approximate inverse of the Hessian matrix, requiring $\mathcal{O}(n^2)$ memory consumption, where n is the number of parameters, making the method impractical for optimizing modern deep neural networks.

This is where limited-memory BFGS (L-BFGS) comes in. The L-BFGS algorithm computes the approximation $\boldsymbol{M}_T$ using the same method as the BFGS algorithm but beginning with the assumption that $\boldsymbol{M}_{t-1}$ is the identity matrix and relying on the past m values of gradient and parameter updates, where m is fixed (typically < 10), rather than storing the approximation from the previous step. The L-BFGS strategy hence requires $\mathcal{O}(n)$ memory per step.

Bonus

11. a) Whenever we make a step in the gradient descent algorithm, the "change" applied to the weight of one unit is computed by assuming that all the other units in the network are fixed, however, after we finish updating every weight in the network, there are high-order interactions between these updates.

To elaborate, let's consider a very simple neural network with a scalar input x, l layers and and only one unit per layer. That is, the network operates as follows:

$$f(x; \boldsymbol{w}) = \left(\prod_{i=1}^{l} w_i \right) x \tag{5}$$

Note that f is linear in x, but polynomial (so nonlinear) in $\boldsymbol{w}$. Suppose our task is also simple: we want to minimize f w.r.t $\boldsymbol{w}$.

Consider applying a gradient descent step to this function to update $\boldsymbol{w}$, with step size ϵ. Denoting $\nabla_{w_i} f(x; \boldsymbol{w}) = g_i$ (for instance, $g_1 = (\prod_{i=2}^{l} w_i)x$), we have

$$f(x; \boldsymbol{w}') = \left(\prod_{i=1}^{l} (w_i - \epsilon g_i) \right) x \tag{6}$$

What we see is that the update terms interact with each other in complicated ways, for instance, the term $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$ is present in the updated weights; this kind of high-order interference with the gradient update can cause more general issues than Hessian ill-conditioning in quadratic problems. More specifically, note that a first order approximation to f tells us that $f(x; \boldsymbol{w}')$ will decrease by $\epsilon \|\boldsymbol{g}\|_2^2$ from $f(x; \boldsymbol{w})$, however, the higher order terms like the one mentioned here could interfere with the descent, and $f(x; \boldsymbol{w}')$ might actually increase from $f(x; \boldsymbol{w})$; in such cases, we need to choose ϵ very carefully to ensure no ascent actually happens, just like in dealing with Hessian ill-conditioning, except now the situation is much more complicated due to higher-than-second-order effects in play now. This kind of phenomenon makes choosing the learning rate of training deep neural networks a very nontrivial task. This problem becomes more sever as the depth of the network increases.

b) Let $\boldsymbol{A}$ be the matrix of minibatch of activations of a layer of the network (activated on m samples), where each row of $\boldsymbol{A}$ corresponds to the activations of the layer for one sample. In batch normalization, during *training*, we normalize $\boldsymbol{A}$ as follows:

$$\boldsymbol{A}' = \frac{\boldsymbol{A} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{7}$$

where

$$\begin{aligned} \boldsymbol{\mu} &= \frac{1}{m} \sum_{i=1}^{m} [\boldsymbol{A}]_{i,:} \\ \boldsymbol{\sigma} &= \sqrt{\delta + \frac{1}{m} \sum_{i=1}^{m} ([\boldsymbol{A}] - \boldsymbol{\mu})_i^2}, \quad \delta \approx 10^{-8} \text{ for numerical stability} \end{aligned} \tag{8}$$

The arithmetic here is based on broadcasting the vector $\boldsymbol{\mu}$ and the vector $\boldsymbol{\sigma}$ to be applied to every row of the matrix $\boldsymbol{A}$. Within each row, the arithmetic is element-wise, so $[\boldsymbol{A}]_{i,j}$ is normalized by subtracting μ_j and dividing by σ_j. Backpropagating through these operations during training means that, the gradient will never propose an operation that acts simply to increase the standard deviation or mean of the layer's activations; the normalization operations remove the effect of such an action and zero out its component in the gradient.

Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network containing that unit. To maintain the expressive power of the network, it is common to replace the normalized $\boldsymbol{A}'$ by $\boldsymbol{\gamma}\boldsymbol{A}' + \boldsymbol{\beta}$. The variables $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned parameters that allow the new variable to have *any* mean and standard deviation. The purpose of the new parametrization is to represent the same family of functions of the input as the old parametrization (the one where we do nothing to $\boldsymbol{A}$), but the new parametrization has different *learning dynamics*. In the old parametrization, the mean of $\boldsymbol{A}$ was determined by a complicated interaction between the parameters in the layers below $\boldsymbol{A}$ (i.e. layers closer to the input than the layer of $\boldsymbol{A}$). In the new parametrization, the mean of the activation $\boldsymbol{\gamma}\boldsymbol{A}' + \boldsymbol{\beta}$ is determined solely by $\boldsymbol{\beta}$.

12. a) The Polyak averaging meta-algorithm consists of averaging several points in the trajectory through parameter space visited by an optimization algorithm. Suppose that through gradient descent, the parameters (points in the parameter space) $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, ..., \boldsymbol{\theta}_t$ had been visited, then Polyak averaging produces

$$\hat{\boldsymbol{\theta}}_t = \frac{1}{t}\sum_{i=1}^{t} \boldsymbol{\theta}_i \tag{9}$$

This meta-algorithm provides strong guarantees in convergence for convex optimization problems. Intuitively, the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all the locations on either side should be close to the bottom of the valley though.

In nonconvex problems, the optimization landscape can be very complicated, and it does not necessarily make sense to include points from the extreme past in the current average, so an exponentially decaying moving average is used:

$$\hat{\boldsymbol{\theta}}_t = \alpha\hat{\boldsymbol{\theta}}_{t-1} + (1-\alpha)\boldsymbol{\theta}_t,\ \alpha \in [0, 1) \tag{10}$$

b) An example of greedy supervised pretraining is provided in [2] (see also Figure 8.7 of recommended text). In that original version of greedy supervised pretraining, each stage consists of a supervised learning training task involving only a subset of the layers in the final neural network. A very simple version would consist of first training a shallow architecture with one hidden layer between the input and output, and then incrementally adding and training one hidden layer per step, while maintaining the parameter setting of layers pretrained in previous steps.

References

[1] Bottou, L. and Bousquet, O., "The tradeoffs of large scale learning," in *Conference on Neural Information Processing Systems*, 2008.

[2] Bengio, Y. and LeCun, Y., "Scaling learning algorithms towards AI," in *Large Scale Kernel Machines*, 2007.