ECE 595: Homework 3

Rahul Deshmukh, PUID: 0030004932

(Fall 2019)

1 Problem 1: SGD -Minibatch Size

The major factors that contribute to the choice of minibatch size for SGD are :

- Accuracy of gradient: A Larger batch size (of size n) provides a more accurate estimate of the gradient but with less than linear returns but Larger batch size require higher computation time.
- Processing Power: extremely small batches result in under-utilization of multicore architectures. Also there the number of samples that can be processed parallelly is limited by the number of threads available, any size larger than this limit would need more increase the computational time.
- Memory limitation: The amount of memory scales with the batch size but the memory is limited by the hardware setup
- Specific sizes preferred by GPU: When using GPUs the number of cores available is always a power of 2. Therefore to optimize runtime, batch sizes of power of 2 are chose. Typical sizes range form 32 to 256.
- Regularization: Small batches offer regularization effect due the noise they add to the learning process. However, training a smaller batch size might require smaller learning rate to maintain stability because of high variance of estimate of gradient which increases total runtime.

For a batch size of 1, the generalization error will be the best due to the regularization effect as discussed above. However,due to high variance of gradient estimate the learning rate needs to be small for stability. Also, the the total runtime can be very high as a result to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.

2 Problem 2: SGD- Randomness

(a) To compute an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent therefore it is crucial that the minibatches be selected randomly. Also many datasets are naturally arranged in a way where successive examples are highly correlated and if we draw examples from the dataset in an ordered list then each of our minibatches will be extremely biased, therefore it is necessary to shuffle the examples before selecting the minibatches.

(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. 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 fixes 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.

3 Problem 3: Ill-Conditioning of Hessian

(a) The phenomenon of Hessian ill-conditioning is best explained by using the second-order Taylor expansion of the cost function:

$$\begin{aligned} J(\boldsymbol{\theta}) &= J(\boldsymbol{\theta}^{(0)}) + (\boldsymbol{\theta} - \boldsymbol{\theta}^{(0)})\boldsymbol{g} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^{(0)})^T H(\boldsymbol{\theta} - \boldsymbol{\theta}^{(0)}) \\ \text{with: } \boldsymbol{\theta} &= \boldsymbol{\theta}^{(0)} - \epsilon \boldsymbol{g} \\ J(\boldsymbol{\theta}) &= J(\boldsymbol{\theta}^{(0)}) + \frac{1}{2}\epsilon^2 \boldsymbol{g}^T H \boldsymbol{g} - \epsilon \boldsymbol{g}^T \boldsymbol{g} \qquad (1) \end{aligned}$$

From Eq 1 we can see that a gradient descent step of $-\epsilon\boldsymbol{g}$ adds $\frac{1}{2}\epsilon^2\boldsymbol{g}^T H\boldsymbol{g} - \epsilon\boldsymbol{g}^T\boldsymbol{g}$ to the loss.

Now, for gradient descent we want to reduce the loss ie $\frac{1}{2}\epsilon^2\boldsymbol{g}^T H\boldsymbol{g} < \epsilon\boldsymbol{g}^T\boldsymbol{g}$. However, if the hessian, H, is ill-conditioned then the term $\frac{1}{2}\epsilon^2\boldsymbol{g}^T H\boldsymbol{g}$ grows by more than an order of magnitude compared to $\epsilon\boldsymbol{g}^T\boldsymbol{g}$.

(b) As a result of ill-conditioning of Hessian, the learning becomes very slow despite the presence of a strong gradient because the learning rate, ϵ, must be shrunk to compensate for even stronger curvature (dependent on H).

4 Problem 4: Saddle Points

(a) Saddle point is where the Hessian matrix has both positive and negative eigenvalues whereas for a minima all the eigenvalues should be positive. For neural networks with large parameter space, we can think of the sign of each eigenvalue as if generated by flipping a coin, and it will be exponentially unlikely that all the eigenvalues a have the same sign and simultaneously it will be very likely that there is mix of signs of eigenvalues which means that we can expect saddle points to occur more frequently than local minima.

(b) While gradient descent is designed to move downhill to reduce the cost and is not explicitly designed to seek a critical point (ie gradient is zero) because of which gradient descent are unaffected from nearby saddle points. However, for second-order method such as Newton's method which is designed to solve for a point where the gradient is zero can reach a saddle point as the parameter space is abundant of saddle points.

5 Problem 5: SGD

(a) A sufficient condition on learning rate for convergence of stochastic gradient descent algorithm is:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty$$
$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

A typical schedule for satisfying this condition in practice is to decay the learning rate linearly until iteration τ using:

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

with $\alpha = \frac{k}{\tau}$ and after iteration τ it is common to leave ϵ constant. Where usually, τ is set to the number of iteration required to make a few hundred passes through the training set and ϵ_τ is usually set to $\frac{\epsilon_0}{100}$. Where ϵ_0 is chosen using trial and error by observing the value of the loss for several training iterations for several candidates of learning rate and then chose the best performing learning rate.

(b) Excess error is measure used to study the convergence rate of an optimization algorithm. It is defined as: $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the mount by which the current loss function exceeds the minimum possible cost. The order of decay of the excess error for SGD for convex function is $O(\frac{1}{\sqrt{k}})$ and for strictly convex function is $O(\frac{1}{k})$, where k is the number of iterations.

The Cramer-Rao bound states that the generalization error cannot decrease faster than $O(\frac{1}{k})$, therefore it may not be worthwhile to pursue an optimization algorithm that converges faster than $O(\frac{1}{k})$ (ie that of SGD), as faster convergence may correspond to overfitting.

6 Problem 6: Momentum

(a) the pseudo-code for stochastic gradient descent with momentum is given by:

Algorithm 1 Stochastic gradient descent with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$ and initial velocity $v$
while stopping criteria not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while
```

Therefore from Algorithm 1 we can see that the velocity term $\boldsymbol{v}$ accumulates an exponentially decaying moving average of past gradients and influences the next direction of descent.

(b) With momentum, the larger the value of α is relative to ϵ the more the previous gradients affect the current direction. Also, if the gradient direction is consistent, the regular SGD will make a step size of $\epsilon||\boldsymbol{g}||$ whereas with momentum the step length will be $\frac{\epsilon||\boldsymbol{g}||}{1-\alpha}$.

(c) SGD with momentum can be interpreted as a physical viscous drag on a unit mass particle which experiences another physical force $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ which pushes the particle downhill along the cost function surface. The addition of viscous force helps SGD to reach equilibrium, because without the viscous force the particle might never come to rest. A viscous drag is preferable over a turbulent drag ($\propto \boldsymbol{v}^2$) as turbulent drag becomes very weak when the velocity is small and is not powerful enough to force the particle to come to rest and causing the particle to move away from its initial position forever. Also, a dry-friction formulation is also not suitable because the friction force is too strong when gradients are small which will lead to pre-mature stopping of the particle before it reaches local minima.

(d) The pseudo-code for SGD with Nesterov momentum is as follows:

Algorithm 2 Stochastic gradient descent with Nesterov momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$ and initial velocity $v$
while stopping criteria not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient at interim point: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}, \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while
```

With Nesterov momentum in convex batch gradient case it has been shown that the rate of convergence of excess error rate goes from $O(\frac{1}{k})$ to $O(\frac{1}{k^2})$.

7 Problem 7: Parameter Initialization

(a) From an optimization perspective the weights should be large enough to propagate information successfully ie activations and gradients are significant so that no signal is lost, but regularization encourage making them smaller. The choice of optimization algorithm such as SGD which makes small incremental changes to weights and tend to halt in areas that are nearer to the initial parameter expresses a prior that the final parameter should be close to the initial parameters. Therefore, if we chose parameters $\boldsymbol{\theta}$ using a Gaussian prior with mean 0, it says that more likely the units do not interact with each other. Whereas when initialized with a mean of $\boldsymbol{\theta}_0$ as a large value, then the prior specifies which units should interact with each other and how they should interact.

(b) Sparse initialization is a scheme where each unit is initialized to have exactly k non-zero weights which helps in achieving more diversity among the units at initialization time. However, it imposes a very strong prior on the weights that are chosen to have large Gaussian values, because it takes a long time for gradient descent to shrink the incorrect large values. Therefore, this scheme can cause problems for units such as maxout units that have several filters that must be coordinates with each other.

8 Problem 8: Adaptive Learning Rate

(a) The pseudocode for AdaGrad Algorithm is given by 3. AdaGrad individually adapts the leaning rates of all model parameters by scaling them inversely proportional to the square-root of the sum of all historical squared values of the gradient. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

```
Algorithm 3 AdaGrad Algorithm
  Require: Global learning rate $\epsilon$
  Require: Initial parameter $\boldsymbol{\theta}$
  Require: Small constant $\delta$,perhaps $10^{-7}$ for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = 0$
  while stopping criteria not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient at interim point: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate the squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\delta+\sqrt{r}} \odot \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

(b) The RMSProp with Nesterov momentum algorithm as shown in Algorithm 4 modifies Ada-Grad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average and uses Nesterov momentum scheme for updating the parameters.

(c) The biggest difference between AdaGrad and RMSPropr will be that while AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have may have made the learning rate too small before arriving at a minima, RMSProp uses an exponentially decaying average to discard history from the extreme past (that is irrelevant to current iteration) so that it can converge rapidly after finding a convex bowl. Also, compared to AdaGrad, RMSProp

Algorithm 4 RMSProp algorithm with Nesterov Momentum

```
**Require:** Global learning rate $\epsilon$, decay rate $\rho$, momentum parameter $\alpha$
**Require:** Initial parameter $\boldsymbol{\theta}$ and initial velocity $v$
Initialize gradient accumulation variable $\boldsymbol{r} = 0$
**while** stopping criteria not met **do**
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient at interim point: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}, \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{r}} \odot \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
**end while**
```

requires an additional hyper-parameter ρ which controls the length scale of the moving average. Therefore it might require additional hyper-parameter tuning.

(d) Adam is another popular adaptive learning rate optimization algorithm as described in Algorithm 5 with two important distinctions from RMSProp. First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. Second, Adam includes bias corrections to the estimates of both the first-order moments and the second-order moments to account for their initialization at the origin. RMSProp also incorporates an estimate of the second-order moment however, it lacks the correction factor.

Algorithm 5 Adam algorithm

```
**Require:** Step size $\epsilon$ (default: 0.001)
**Require:** Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in [0,1). (default values: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (default $10^{-8}$)
Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = 0, \boldsymbol{r} = 0$
Initialize time step $t = 0$
**while** stopping criteria not met **do**
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1) \boldsymbol{g}$
    Update biased second moment estimate $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2) \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
**end while**
```

9 Problem 9: Conjugate Gradients

(a) The method of conjugate gradients helps in overcoming the problem of steepest descent. In steepest descent, given a previous search direction ($\boldsymbol{d}_{t-1}$) the next direction ($\boldsymbol{d}_t$) is obtained where the directional derivative is zero ie $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\boldsymbol{d}_{t-1} = 0$ and $\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ which means $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$ are orthogonal. However this does not ensure that the gradient along $\boldsymbol{d}_{t-1}$ does not change while searching along $\boldsymbol{d}_t$. Conjugate gradients overcomes this problem by enforcing that $< H\boldsymbol{d}_{t-1}, \boldsymbol{d}_t >= 0$, where $H\boldsymbol{d}_{t-1}$ measures the change in gradient along $\boldsymbol{d}_{t-1}$ and if its projection along $\boldsymbol{d}_t$ is zero then we don't end up undoing the progress we made in the previous step. At training iteration t the next search direction $\boldsymbol{d}_t$ is given by:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}$$

Where β_t is a coefficient whose magnitude controls how much of the direction $\boldsymbol{d}_t$ should we add back to the current search direction.

(b) As $\boldsymbol{d}_t^T H \boldsymbol{d}_{t-1} = 0$, to impose conjugacy we need to calculate β_t which would involve calculation of eigenvectors of H which amounts to a cost of $O(k^3)$ and thus equivalent to Newton's method computational cost. However, to overcome this problem we can estimate β_t using approximations. Two popular methods for computing it are:

1. Fletcher-Reeves:
$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

2. Polak-Ribiere:
$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

10 Problem 10: L-BFGS

The problem with classic BFGS algorithm is that we need to store the matrix M which is an approximation to H^{-1}, this requires $O(n^2)$ memory which makes it infeasible for neural networks where the number of weights n can reach up-to a million.

Limited Memory BFGS (or L-BFGS) overcomes this problem by avoiding the storing of M. It computes M using the same method as the classic BFGS but beginning with the assumption that $M_{(t-1)}$ is identity matrix, rather than storing the approximation from one step to another. Further, to include more information about the Hessian, L-BFGS stores m previous values of vectors $\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$ and $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})$ to estimate M_t, this requires $O(n)$ memory.