6.883 Science of Deep Learning – Spring 2018 February 20, 2018

Lecture 4: Optimization Landscape of Deep Learning

Lecturer: Aleksander Mądry *Scribes: Nishanth Dikkala, Pritish Kamath, Devendra Shelar (Revised by Andrew Ilyas and Dimitris Tsipras)*

Disclaimer: *Our understanding of deep learning is still severely lacking and mostly conjectural. As a result, the statements in these lecture notes often take the point of view of the paper that introduced the respective techniques/conjecture and should not be treated as conclusive explanations for the intriguing phenomenon they aim to explain.*

1 Gradient Descent Methods

In Lecture 2, we covered several iterative continuous optimization methods, and showed that every update step can be seen as corresponding to solving a local approximation problem. In particular, we looked at:

- **Gradient descent**, where we use a locally linearization of the function f:

$$\Delta^{(t)} := \arg\min_{\Delta} g_t^\top \Delta + \frac{1}{2}\beta\|\Delta\|^2 \qquad = -\frac{1}{\beta} g_t,$$

$$x^{(t+1)} := x^{(t)} + \Delta^{(t)},$$

  where $g_t = \nabla f\left(x^{(t)}\right)$ is the gradient through the function f being optimized. We also considered the momentum variant, where we keep track of an exponentially weighted moving average of Δ, i.e. $\Delta^{(t)} = \gamma\Delta^{(t-1)} - \varepsilon\nabla f^\top(x^{(t)})$.

- **Newton's method**, where we use a second-order local approximation of f:

$$x^{(t+1)} := x^{(t)} - \eta \boldsymbol{H}_t^{-1} g_t,$$

  where again $g_t = \nabla f(x^{(t)})$ is the gradient of f at $x^{(t)}$ and $\boldsymbol{H}_t = \nabla^2 f(x^{(t)})$ is the Hessian. In **Quasi-Newton methods**, we use the same update step but instead use an estimate of $\boldsymbol{H}_t$ derived from only first-order information. BFGS and its variant L-BFGS were heuristics to approximate the inverse of Hessian of f. People often use some heuristics on top of these methods, which empirically seem to work well for deep learning.

1.1 Natural gradient descent

Natural gradient descent is another oft-used algorithm for training deep neural networks in the context of classification tasks.

Classification as cross-entropy minimization. Note that in the context of classification, we can view a deep neural network with parameters θ, input $x \in \mathcal{X}$, and possible labels $\mathcal{Y} \ni y$ as outputting a conditional probability distribution $P_\theta(y|x)$ rather than just labels. Indeed, the $\mathsf{softmax}$ activation function, ubiquitously applied to the output of the neural networks, actually ensures that the output of DNNs is a valid probability distribution. For network outputs $\boldsymbol{r}(x;\theta) = \{r_y(x;\theta)\ \forall\ y \in \mathcal{Y}\}$,

$$P_\theta(y|x) = \mathsf{softmax}\left(\boldsymbol{r}(x;\theta)\right) := \frac{e^{r_y(\theta,x)}}{\sum_y e^{r_y(\theta,x)}}\ .$$

Given a dataset $D = \{(x_i, y_i)\}$ of inputs and their true labels, the goal of classification is then to find parameters θ^* which minimize the cross-entropy between the output distribution $P_\theta(\cdot|x)$, and the ground-truth distribution $P^*(\cdot|x)$, which for any image x is

$$P(y|x) = \begin{cases} 1 & \text{if } y = y_i, \text{ the true label} \\ 0 & \text{otherwise} \end{cases}.$$

Recall that the cross-entropy between two distributions P and Q is given by:

$$H(P, Q) = -\mathbb{E}_{x\sim P}\left[\log(Q(x))\right],$$

and thus we can write the cross-entropy between the true and predicted distributions as:

$$H(P^*(\cdot|x_i), P_\theta(\cdot|x_i)) = -\log(P_\theta(y_i|x_i)$$

Using this, our classification objective becomes that of finding θ^* such that:

$$\theta^* = \arg\min_\theta \sum_{(x_i,y_i)\in D} \log(P_\theta(y_i|x_i))$$

Natural gradient. Recall that we derived gradient descent by minimizing a locally linear approximation of the function. In the natural gradient descent method, we do the same, but opt to use KL-divergence between the distributions $P_\theta(\cdot|x)$ and $P_{\theta+\Delta}(\cdot|x)$ as a distance measure, rather than the ℓ_2 distance $\|(x + \Delta) - x\|^2$. In particular

$$\Delta^{(t)} \;:=\; \arg\min_\Delta \nabla\mathsf{loss}(\theta^{(t)})^\top \cdot \Delta + \mathbb{E}_{x\sim\mathcal{D}}\left[D_{\mathsf{KL}}\left(P_\theta(\cdot|x) \;\middle\|\; P_{\theta+\Delta}(\cdot|x)\right)\right].$$

The intuition behind this method is that we care more about the output distribution of the network and not about the exact parameters governing the network per se. Thus, this choice of regularizer ensures that the output distribution of the network on input x does not change drastically in any gradient descent iteration. It turns out that the minimizer of the above expression is precisely

$$\Delta^{(t)} = -F_\theta^{-1}\nabla_\theta(\mathsf{loss}(\theta)))$$

where F_θ is the Fisher information matrix given as

$$F_\theta := \mathbb{E}_{x\sim\mathcal{D}}\left[\mathbb{E}_{y\sim P_\theta(\cdot|x)}\left[\nabla_\theta \log P_\theta(y|x) \cdot \nabla_\theta \log P_\theta(y|x)^\top\right]\right].$$

Note that the y's in the definition of the Fisher information matrix are sample from the *predicted* distribution. The matrix does not depend at all on the true labels.

Note that we don't have access to the true distribution $\mathcal{D}$, and so we cannot compute F_θ explicitly. Instead, we use the empirical distribution of the dataset to get the empirical Fisher information matrix $\hat{F}$. But even then, computing $\hat{F}^{-1}$ itself is also too expensive. In order to deal with this, the K-FAC heuristic (Kronecker-Factored Approximate Curvature) was introduced in [MG15] to empirically estimate $\hat{F}^{-1}$. Despite using merely an approximation to the inverse Fisher matrix, natural gradient descent with K-FAC seems to improve convergence: the training error plotted against time is given in Figure 1. That said, there are several issues/concerns regarding the K-FAC heuristic.

- Each iteration of K-FAC based gradient descent takes a lot of time as the heuristic itself is quite involved and requires highly non-trivial computation.
- The plot provided is for the training error on the MNIST dataset, so we don't know if the heuristic works well for other datasets.
- Most importantly, the plot only shows that the training loss reduces with time. But what about the loss on the test set? It turns out that this heuristic doesn't give good test loss.

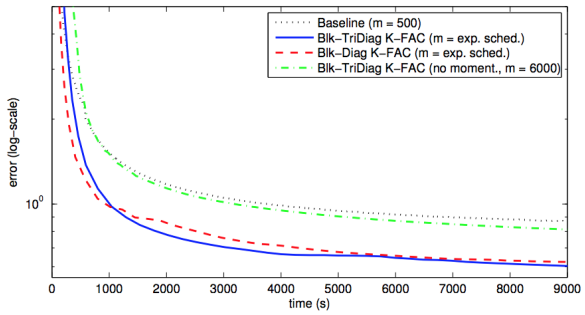


Figure 1: Training error using K-FAC heuristic [MG15]

1.2 Modern Gradient Descent algorithms

In this section, we discuss several more modern first-order methods that are commonly applied in deep learning. Some of these methods were designed explicitly with applications in deep learning in mind, others were first proposed in the context of convex optimization.

AdaGrad. In (both full-batch and stochastic) gradient descent, recall that we keep the same (fixed) learning rate for all the parameters of the model. In practice, however, these parameters might have drastically different effects on the network output (see for example Figure 2).

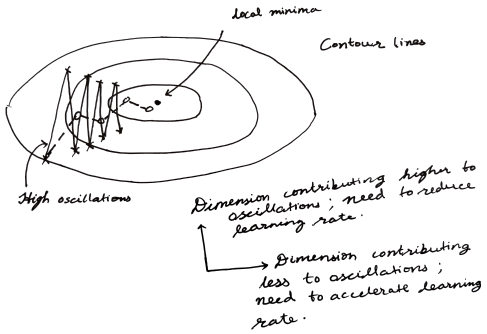


Figure 2: Dimensions contributing differently to oscillations. The dashed line represents the algorithms that induce different learning rates for different parameters based on momentum or by penalizing square magnitude gradients.

To address this, the AdaGrad algorithm [DHS11] allows each parameter to have its own learning rate. Specifically, the learning rate for each parameter depends upon how large the magnitude of gradients along that parameter dimension has been in the previous iterations. For each parameter dimension, AdaGrad accumulates the square of magnitudes of gradient along that dimension in all the previous iterations ($\boldsymbol{r}_{i+1} \leftarrow \boldsymbol{r}_i + \boldsymbol{g}_i^2$) and divides the learning rate by square root of $\boldsymbol{r}_i$ along each dimension. The resulting update step can then be written as:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \operatorname{diag}\left(\sum_{\tau=1}^{t-1} g_\tau g_\tau^\top\right)^{-1/2} g_t \qquad \text{where } g_\tau = \nabla^{(\tau)} f(\theta_\tau), \text{ gradient collected at time } \tau \quad (1)$$

As such, AdaGrad heavily reduces the learning rate for parameters that have too large oscillations (as indicated by high-magnitude gradients), and has much less of an effect on the learning rate for parameters that do not contribute much to the oscillations (i.e. whose gradient magnitude decreases).

RMSProp. In the above, $\boldsymbol{r}_i$ is an accumulator of non-negative terms (magnitudes), and hence always increases. As a result, the learning rates for all the parameters decrease, and may in fact vanish (become

infinitesimally small). If the learning rate vanishes too quickly, AdaGrad will not be able to learn a value for the parameter and it will remain fixed and far from the optimum. (Note that such a concern only arises because of the non-convex nature of training deep neural networks—in the convex case, AdaGrad is proven to converge.) The RMSProp algorithm[1] was designed as an attempt to circumvent this issue.

Rather than using an accumulator of the squared gradients, RMSProp uses *weighted* sum of square magnitudes of gradients, where the weights exponentially decay (with rate ρ, a hyperparameter) over time. Thus, gradients of more recent iterations are assigned more weight than those in earlier iterations.

Intuitively, one can think of RMSProp as directly addressing the non-convexity of the training landscape. In particular, one might imagine that the steepness/shallowness of this landscape may change along the trajectory from initialization to final weights. By assigning more weight to recent gradient magnitudes, one can view RMSProp as implementing AdaGrad "locally," i.e. according to the current shape of the landscape. This view may help to explain why RMSProp tends to outperform AdaGrad in practice. Concretely, the update step implemented by RMSProp is as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \operatorname{diag}\left(\sum_{\tau=1}^{t-1} \rho^{t-\tau} g_\tau g_\tau^\top\right)^{-1/2} g_t,$$

with g_τ being defined as in (1) and $\rho < 1$ being a user-defined hyperparameter.

Adaptive Moment Estimate (Adam). The Adam [KB14] method, short for "ADAptive Moment" estimation, is a combination of ideas from RMSProp and Nesterov's momentum. The motivation is similar to that of AdaGrad and RMSProp , i.e. reducing the learning rate along dimensions contributing to high oscillation by taking into account historical gradient magnitude. Adam combines this idea with momentum by exponentially-weighted averaging both the first and second "moments" (the gradient g_t and its outer product $g_t g_t^\top$, respectively). The algorithm also introduces "bias correction terms" $1 - \beta_1^t$ and $1 - \beta_2^t$, which are meant to counteract the fact that we initialize the average squared magnitude as $\boldsymbol{r} = \mathbf{0}$. It turns out that this simple unification of RMSProp /AdaGrad is highly effective in many settings—since its inception Adam has attained widespread popularity (the original paper has 6000+ citations).

Despite this, Adam is still somewhat poorly understood from an optimization perspective. The original paper provides "recommended hyperparameters" that are not theoretically justified but seem to work well in practice on a wide variety of tasks. Furthermore, though the original paper presented a proof of convergence for convex functions, the proof was later shown to be incorrect, and a counterexample of convergence was recently published [RKK18] for a simple convex problem.

In fact, we still lack conclusive evidence that Adam provides significant benefit over stochastic gradient descent in a general sense (although it has been empirically shown to help in many specific cases).

Potential downsides of adaptive methods. A recent paper [WRS+17] shows some evidence that adaptive methods can in fact overfit when their non-adaptive counterparts don't. Consider, for example, the setting of overparameterized (i.e. underdetermined) linear regression. In this setting, we have a data matrix $X \in \mathbb{R}^{k \times d}$ and a label matrix $y \in \mathbb{R}^k$, and we are trying to fit a linear mapping between them parameterized by $w \in \mathbb{R}^d$, where $k << d$. In particular, we are looking for w^* such that:

$$w^* = \arg\min_w \|Xw - y\|^2$$

In the underdetermined setting, there are actually infinite solution to the above minimization. The solution with the minimum norm (and minimum expected error, c.f. [Mei94]) is: $w^* = X^\top (XX^\top)^{-1} y$. In practice SGD converges to such a solution. [WRS+17] construct a synthetic dataset where adaptive method find a different solution that does not generalize well.

The samples are generated as follows. For the i^{th} vector x_i with label y_i,

[1] Interestingly, the RMSProp algorithm was never published—it was originally described by Geoff Hinton in a Coursera lecture.

- The first feature is the label itself,
- The second and third features are always 1,
- The features from $\{4+5(i-1),\ldots,4+5(i-1)+(1-y_i)\}$ are set to 1 (Note that this induces a unique set of features for each example).
- The remaining features have value 0.

The y_i's are 1 with probability $p > 0.5$ and -1 otherwise, so that for large enough values of k, the majority of the examples are positively labeled. Observe that the only discriminative feature is the first one— the rest are either constant or unique to a single example.

If we initialize training with $w = 0$, the weight on the unique features of the test will be zero. Furthermore, SGD will converge to a solution where the weight of the first feature is larger that the total weight of the second and third one hence achieving perfect accuracy on the test set (perfect generalization).

However, all adaptive methods we considered (AdaGrad , RMSProp , Adam) will converge to a solution $w'\alpha\text{sign}(X^\top y)$. Giving equal weight to all features. While this performs well on the training set due to the unique features each example has, on the test set these unique features do not help in classification. Instead, the classifier will consider consider the first three features with equal weight and always predict 1, bringing test-set accuracy arbitrarily close to random (in particular, $\approx p$).

2 Issues with Applying Gradient Descent Methods in Deep Learning Setting

Last time we talked about different types of gradient descent methods, including the ones used in deep learning. Now, it is time to discuss several issues that may arise when we use first-order methods to train deep neural networks.

Just to build some intuition, consider a fully connected neural network (with a softmax layer on top). Let X^0 denote the data matrix, i.e., the m-by-n matrix whose each of n columns corresponds to one of the data points. One can view this neural network as a sequence of layer-wise transformations of that data matrix. That is, the "data matrix" after passing through i-th layer of that network is given by

$$X^i = \sigma(W^i X^{i-1} + B^i),$$

where W^i is the weight matrix of the i-th layer, B^i is the matrix of added (and learnable) biases, and X^{i-1} is the "data matrix" after passing the previous layers. Here, σ denotes the non-linear activations applied coordinate-wise. A typical choice of the non-linear activation σ is the ReLU activation function given as $\sigma(a) = \max\{a, 0\}$.

Now, to get a feel for how the information about the gradient updates propagate during training, let us ignore the bias matrices and focus on some specific data point j and its representation X_j^1 after passing through the first layer. In that case, the gradient of the change $\nabla_{X_j^1} X_j^\ell$ of the representation X_j^ℓ of that data point after the final (ℓ-th) layer with respect to change in X_j^1 is given by:

$$\nabla_{X_j^1} X_j^\ell = D^\ell W^\ell \cdots D^2 W^2, \tag{2}$$

where each D^i is a diagonal matrix with $D^i_{kk} = \mathbf{1}\{(X_j^i)_k \geq 0\}$, where $(X_j^i)_k$ is the k-th coordinate of the representation X_j^i. (Note that, strictly speaking, in training we can't directly modify the representation vector X_j^1—we can do it only indirectly via modifying W^1.)

Vanishing/exploding gradients. The formula (2) should already hint that training neural networks can be tricky. In particular, the gradient signal can exhibit bad behavior.

For example, suppose each weight matrix W^i is a random orthogonal matrix (i.e. a rotation)—then, due to the fact that each matrix D^i is "killing" half of the dimensions on average, the gradient $\nabla_{X_j^1} X_j^\ell$

would be of magnitude $\exp(-\ell)$ (if not 0), decreasing exponentially with the number of layers in the network. This would be an instance of a broader phenomenon known as the *vanishing gradient* effect.

Another illustration of what can go wrong is the situation if the entries of each W^i are larger than 1. Then, the "signal" in each layer could build up uncontrollably, making the gradient $\nabla_{X_j^1} X_j^\ell$ be of magnitude $\exp(\ell)$. This corresponds to the *exploding gradients* effect.[2]

Issues at initialization. The above discussion should already have hinted that one needs to be careful about how to initialize deep neural networks. Indeed, issues with initialization extend to far more complex cases than this, and there have been a multitude of recent studies showing that initialization can actually cause large issues for network training (see, for example [HR18]).

2.1 Proposed solutions to issues in DNN training

The above problems were plaguing early (modern) deep neural network architectures. Fortunately, over time, we managed to come up with architectural changes (as well as corresponding initialization schemes) that improved the situation considerably. We briefly describe the two most important ones below.

Batch normalization [IS15] Batch normalization was originally proposed to address the hypothesized problem of *internal covariate shift*. Specifically, internal covariate shift refers to the fact that the "data matrix" X^i that the optimization/learning process at the corresponding layer i has as an input constantly changes due to updates to the lower layers of the network. Ioffe and Szegedy viewed this effect as very detrimental to network training and proposed to alleviate it via so-called *batch normalization* layers.

The goal of batch normalization is to ensure that the distribution of activation in each data matrix is more stable. This is achieved by normalizing the distributions to have the same first and second moments for the distributions in each layer. In particular, consider a size-k minibatch of inputs to layer i (i.e., the subset of k columns of the matrix X^i), which we denote as $\{x^{(1)}, \ldots, x^{(k)}\}$, and let $\{y^{(1)}, \ldots, y^{(k)}\}$ be the corresponding vectors of activations after leaving the layer i. Batch normalization involves adding a new layer after layer i whose goal is to reparametrize activations so as they become $\{\hat{y}^{(1)}, \ldots, \hat{y}^{(k)}\}$, where each coordinate $\hat{y}_r^{(b)}$ of $\hat{y}^{(b)}$ is given by:

$$\hat{y}_r^{(b)} = \frac{y_r^{(b)} - \frac{1}{k}\sum_{s=1}^{k} y_r^{(s)}}{\sqrt{\frac{1}{k}\sum_{s=1}^{k}\left(y_r^{(s)} - \frac{1}{k}\sum_{l=1}^{k} y_r^{(l)}\right)^2}}.$$

Intuitively, this transformation whitens the "distribution" $\{y_r^{(1)}, \ldots, y_r^{(k)}\}$ so as it has zero mean and unit variance. (In practice, one sometimes allows this whitened mean and variance to be a (trainable) parameter as well.)

It turns out that adding such batch normalizing layers after each original layer—and, crucially, allowing the training to back-propagate through these layers—leads to dramatic improvement of training reliability. In particular, vanishing and exploding gradients become much less of an issue. (And generalization tends to be somewhat improved as well.)

But, does it mean that the internal covariance shift was indeed the root of the problem here? Recent work [STIM18] has cast doubt that it is the case. Specifically, it demonstrated that for a number of natural ways of measuring internal covariate shifts, batch norm does not seem to have visible effect on reducing that measure. Still, it has an important beneficial effect: it makes the loss landscape be much more navigable for first order methods.

Residual Neural Networks [HZRS16]. Residual networks were proposed as a way to try and address a number of issues related to training networks with large depth (think 30+ layers). An intriguing problem observed with networks of such a large depth is that addition of layers after a certain point start

[2]Note, however, that both these two examples are very cartoonish/unrealistic. An interesting question is though: why do we observe vanishing/exploding gradients in practice?

to degrade the performance (and this is not due to overfitting as even the training loss degrades). The authors who proposed residual networks hypothesized that this is potentially due to the signal "learnt" by the lower layers getting lost by the time we reach the higher layers. To overcome this, they propose that each layer learns a residual function instead. That is, to have our activation matrices X^i evolve as

$$X^i = \sigma(W^i X^{i-1} + B^i) + X^{i-1}.$$

That is, instead of a set of stacked layers learning a function $H(x)$ the authors propose to let them learn the "residual" function $F(x) = H(x) - x$ instead. (This assumes that the input x and output $H(x)$ are of the same dimension which roughly holds for deep networks.) Architecturally, this corresponds to adding "shortcut" (referred to as residual) connections bypassing each one of the layers. The resulting architectures indeed perform significantly better in training, making ResNets (the architectures implementing residual connections) a very popular class of models.

3 Visualizing the landscape of Loss Functions

It is quite hard to visualize the landscape of the loss function because of the extremely high dimensionality of the space.

Towards the goal of visualizing the landscape, Goodfellow et al [GVS14] considered the line joining the initial point θ^0 and the final point θ^T, and plotted the empirical loss on the training set along this line for both MNIST and CIFAR (Figure 3). Things actually look surprisingly convex! Observe that zooming in on the first part of the loss on CIFAR (**??**), there is a clear non-convexity that training got around.

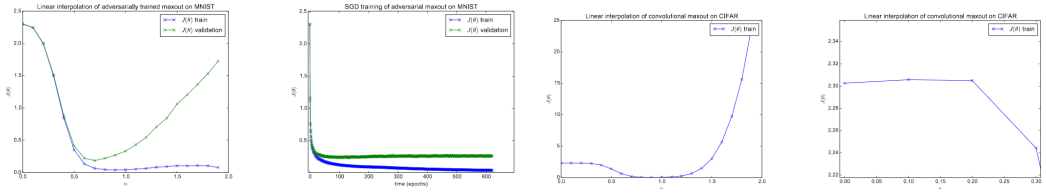


Figure 3: Empirical loss along the line connecting θ^0 and θ^T for MNIST (left) and CIFAR (right) [GVS14]

Another work along this line is by [LTSM18], where they plot the empirical loss function along random directions and also along the gradient directions. looks nice along a random direction but rather complicated in the 'total gradient' direction. Figure 4. Figure 4 shows how the empirical loss changes along certain parameter dimensions. Along some directions, the loss is smooth and convex quadratic-like curve, while on other directions (blue curve) it is extremely non-convex. One good comment made in class to give a possible explanation was as follows: Because the neural network has many layers, some directions will involve parameters in the first layer, while other directions will involve parameters in the last layer. So, it is likely that the loss value is smoother with respect to the parameters in the last layer, while very non-convex with respect to parameters in the first layer. Hence, it is unfair to compare the loss values along randomly chosen parameter dimensions. A fairer comparison would be to choose parameters from the same layer, and evaluate the loss landscape along those parameters.

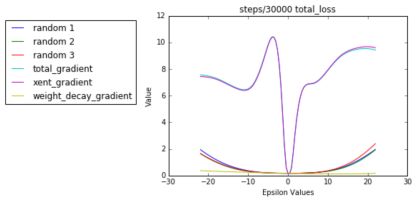
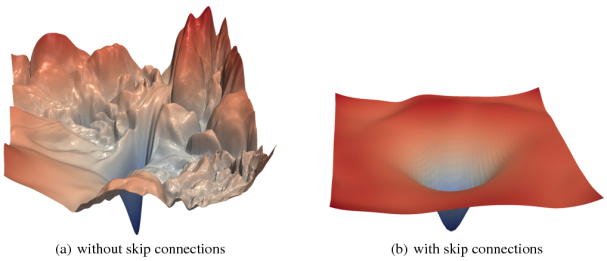


Figure 4: Empirical loss along random and gradient directions [LTSM18]

Another recent work [LXTG17] proposes a method for generating visualizations of the loss landscape based on a technique which the authors call *filter normalization*. Filter normalization tries to address the exact issue we previously raised—how should we (fairly) pick parameter directions along which the loss should be plotted? In particular, filter normalization tries to account for the scale-invariance of many of the parameters in a deep neural network by carefully scaling the plotted directions according to the actual values of the parameters. We refer the reader to [LXTG17] for more details on the technique—in general, the resulting visualizations are at least compelling:



(a) without skip connections (b) with skip connections

The authors of [LXTG17] also claim that visualizations which employ filter normalization tend to actually be predictive of generalization performance.

4 Extra Figures

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    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: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Figure 5: The Adaptive Moment (Adam) algorithm for first-order optimization.

References

[DHS11] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[GVS14] Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. Qualitatively characterizing neural network optimization problems. *CoRR*, abs/1412.6544, 2014.

[HR18] Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. *arXiv preprint arXiv:1803.01719*, 2018.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[LTSM18] Li, Tran, Schmidt, and Mądry. Unpublished work. *Unpublished*, 2018.

[LXTG17] Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets. *CoRR*, abs/1712.09913, 2017.

[Mei94] Ron Meir. Bias, variance and the combination of least squares estimators. In *NIPS*, 1994.

[MG15] James Martens and Roger B. Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2408–2417, 2015.

[RKK18] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the Convergence of Adam and Beyond. In *International Conference on Learning Representations*, 2018.

[STIM18] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?(no, it is not about internal covariate shift). *arXiv preprint arXiv:1805.11604*, 2018.

[WRS+17] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 4151–4161, 2017.