CSE 475: Statistical Methods in AI **Monsoon 2019**

SMAI-M-2019 19: More on Backpropagation

Lecturer: C. V. Jawahar *Date: DATE*

We had seen the back-propagation algorithm as one that iteratively minimizing the loss/error over the samples. We discussed the algorithm as two steps:

1. **Forward Pass:** Do a forward pass for all the samples and compute the loss over the training set.
2. **Backward Pass:** Do refine all the learnable parameters (weights) iteratively as:

$$\theta^{k+1} \leftarrow \theta^k - \eta \frac{\partial J}{\partial \theta}$$

To make the discussions and notations simple, we assume that all the learnable parameters are part of θ, and the loss/objective is $J(\theta)$. We use this notation throughout.

19.121 Stochasticity

In practice we do not compute the loss over all the samples and then update the parameters in one go. We do this over a randomly selected subset of the samples. This leads to stochastic mini batch backpropagation algorithm.

Note that the batch mode of the BP computes loss over all the samples. This is actually an approximation of the "true" gradient which we are not able to compute, since we do not know the analytic function form. If the true gradient can be approximated with sum of gradients over a number of samples, thus approximation can be computed from a subset of the samples also. However, computing the gradient from a smaller set may have larger error than that computed from all the samples. However, this is much more efficient. Therefore, we can have BP implemented as batch, single sample and mini-batch. Minibatch is preferred.

The convergence and properties of stochastic gradient descent methods have been analyzed in both convex minimization and stochastic approximation. In many related areas, it has been shown that the stochastic gradient descent will converge to the batch (not stochastic) estimates in many practical situations.

19.121.1 Epochs and Iterations

In neural network literature, it is common to use the word epoch. In the neural network terminology, one epoch consists of one forward pass and one backward pass of all the training examples. However, as we discussed above, batch size (i.e., the number of training examples in one forward/backward pass) could be much smaller than the entire data. Though it is possible to randomly create the batch size every time, it becomes computationally efficient to create random batches once (in the beginning of the training) and use the same batches throughout. When the batch size is large, the memory requirement of the training could increase.

19.122 Sub-gradients

Another issue is the sub-gradient. many functions that we use in the modern deep neural networks are not truely differentiable. Eg. ReLU. How do we handle these?

Eg1: ReLu A popular activation function is ReLU

$$\phi(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Eg2: Hinge Loss Hinge loss has been a key component in the success of SVMs.

$$max(0, 1 - t \cdot y)$$

Though in these two cases, the function is not continuous, we can compute the derivatives at each point, except the point of discontinuity. This is done with the help of "subgradients". In mathematics, the subgradient, generalizes the derivative to convex functions which are not necessarily differentiable.

19.123 Refinements

19.123.1 Initialization

. Initialization is very important for any iterative solution to the non-convex optimization. It is always feared

that performance of the neural networks could be highly dependent on this lucky initialization. (Is it really true? Did you find so?)

Q: If we initialize all the weights as zero. What would happen? If we initialize all the weights as non-zero, but a constant value, what could happen?

It is observed that the random initializations are ideal for the neural networks. In the earlier days, it was common to try multiple initializations and pick the best. Even if we randomly initialize the weights, the output of a neuron depends on the inputs. The variance of the output directly depends on the inputs. Glorot and Bengio (2010) proposed to randomly initialize the weights with a variance that depends on the number of incoming (fan-in) and outgoing (fan-out) connections. This method (popularly known as Xavier's initilaization (AISTAT 2010) works well in practice.

19.123.2 Better Update Rules

We know the update rule as:

$$\theta^{k+1} \leftarrow \theta^k - \Delta\theta$$

Gradient Update In the simple gradient descent (and backpropagation) we saw, the change in weight $\Delta\theta$ is proportional to the derivative of the loss/objective.

$$\Delta\theta = \eta \frac{\partial L}{\partial \theta}$$

Momentum Momentum based optimization provided better convergence properties, and results. The idea is simple. If we conistently change the weights/parameters in certain direction, we can make larger steps instead of small steps.

$$\Delta\theta = \eta \frac{\partial L}{\partial \theta} + \gamma \Delta\theta^{t-1}$$

If we change the directions frequently, then we make small step. Connecting to the analogies from physics, we make big steps, if the surface is smooth. If surface is rough, we make small steps. Typically the momentum is set to 0.9. This classical momentum term could carry the ball beyond the minimum point. Ideally, we would like "the ball" to slow down when the ball reaches the minimum point and the slope starts increasing. This is achieved by the Nesterov Momentum.

Another aspect is that we have only one parameter for the learning rate. Can we have different parameters/scales for each dimension? Adaptive gradients and its variations are aimed at this.

Other Recent Advances(*) A number of refinements have surfaced in the recent years:

- Nesterov Momentum [Nesterov 1983]
- AdaGrad [Duchi 2011]
- AdaDelta [Zeiler 2012]
- RMSProp [Tieleman and Hinton, 2012]

Though some of these refinements were recent, they have all reached the end users through the popular libraries for neural networks.

The ADAptive Moment Estimation (ADAM) [Kingma and Ba, ICLR 2014] is a popular refinement of the update rule similar to the momentum techniuques. The main difference between Adam and its two predecessors (RMSprop and AdaDelta) is that the updates are estimated by using both the first moment and the second moment of the gradient. A running average of gradients (mean) is maintained along with a running average of the squared gradients.

19.123.3 Termination Criteria

Termination is often when there is no significant change in the loss/objective. However, a larger number of iterations can lead to overfitting. Often an "early stop" is performed by looking at how the loss/objective change over the validation data set. When the loss starts increasing on the validation data, iterations are stopped.

19.124 Loss Functions

We know loss function as a measure of discrepancy between the ground truth (true value) and the predicted output. The mean square error

$$\sum_{i=1}^{N} (t_i - o_i)^2$$

is a useful measure in this regard. This is popular for many regression tasks, where t_i and o_i are reals. If there are more than one neurons in the output layer (say M), then one can compute it by summing up over all the M neurons

$$\sum_{i=1}^{N} \sum_{j=1}^{M} (t_i^j - o_i^j)^2$$

This is not the apt measure for classification related tasks. When the output has M neuons and output of these neurons corresponds to the probabilities to these classes, then it is typical to use softmax in the output layer.

$$o^k = \frac{e^{o^k}}{\sum_{j=1}^{M} e^{o^j}}$$

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 19.19: Adam: Algorithm. taken from Kingma and Ba, ICLR 2014

. The softmax output o^k could be considered as the probability to be in class k. p^k.

The popular loss in this case is cross entropy loss

$$= \sum_{j=1}^{M} y^j \log p^j$$

for a M class classification problem. To appreciate this loss better, let us look at how can we compute the distance/dissimilarity between two probabilistic distributions. The classical measure for this is called Bhattacharya distance (or some people also call it as KL-divergemce)

$$D_{KL}(p(x)||q(x)) = \sum_x p(x) \ln \frac{p(x)}{q(x)}$$

A symmetric version $(D_{KL}(p(x)||q(x)) + D_{KL}(q(x)||p(x)))$ is also preferred in many cases. See Wikipedia on KL divergnce to appreciate the relationship between cross entropy and KL divergence.

19.124.1 Regularization

It is common to regularize the neural networks in different ways to prevent overfitting. One classical method is to look for simple/small neural network that can solve the problem of interest. The simplicity of such a neural network is measured with the number of non-zero weights (L0 norm of the weights) or sum of absolute value of the weights (L1 norm of the weights). Both L0 and L1 norms are hard to work with in most cases. In practice L2 norm is a good choice and the new loss then becomes old loss plus the L2 norm of the weights in the neural networks.

$$J' = J + ||\theta||_2^2$$

Please note that the new term is an addition. (remember $u+v$) and the new update rule will be almost the same as the old update rule plus an additional term corresponding to the L2 norm of the weights.

One can attempt to minimize $L0$ norm by pruning (removing or explicitly making it as zero) some of the tiny weights. Usually removal of some of the tiny weights need not change the network performance (output values) significantly. One of two iterations of the backpropagation

algorithm can recover any loss in performance due to this. However, having zero elements in the weight matrices will not save memory or computation if your inherent data structure is matrix/tensor. One needs to use appropriate sparse matrix computing techniques in this case.

19.125 Second Order Methods

The popular backpropagation algorithm that we studied till now uses only the first order derivatives. It takes linear steps along the negative gradient. Assume we have knowledge about the curvature/shape of the curve, then we could obtain better estimate of the solution in every step. Second order method uses Hessian (matrix of second order derivatives) in every step. (Recollect our discussions related to second order gradient descent methods elsewhere.)

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 J}{\theta_1^2} & \frac{\partial^2 J}{\theta_1 \theta_2} & \dots & \frac{\partial^2 J}{\theta_1 \theta_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial^2 J}{\theta_N \theta_1} & \frac{\partial^2 J}{\theta_N \theta_2} & \dots & \frac{\partial^2 J}{\theta_N^2} \end{bmatrix}$$

Remembering the truncated Taylor series expansion,

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla J(\theta_0) + (\theta - \theta_0)^T H (\theta - \theta_0)$$

We could take the derivative and equate to zero:

$$\nabla_\theta J(\theta) = \nabla_\theta J(\theta_0) + H(\theta - \theta_0) = \mathbf{0}$$

or

$$\theta = \theta_0 - \mathbf{H}^{-1} \nabla_\theta J(\theta_0)$$

If the function was quadratic, this step directly takes us to the minimum. (Q: why?). If not, this takes us to the minima of the quadratic approximation.

In general, we need to now compute g, compute H and then update as $\theta^{k+1} \leftarrow \theta^k - \mathbf{H}^{-1}\mathbf{g}$ Where $\mathbf{g} = \nabla_\theta J(\theta)$

The above mentioned second order method (alias Newtons method) requires, at every iteration, calculating and then inverting the Hessian. If the network has N parameters, then inverting the Hessian is $O(N^3)$. This renders Newtons method impractical for the modern deep neural networks.

When the Hessian has negative eigenvalues, then steps along the corresponding eigenvectors are gradient ascent steps. To counteract this, it is possible to regularize the Hessian, so that the updates become:

$$\theta \leftarrow \theta^k - (\mathbf{H} + \alpha I)^{-1} \nabla_\theta J(\theta_0)$$

As α becomes larger, this turns into first order gradient descent with learning rate $\frac{1}{\alpha}$.

19.126 Discussions (*)

19.126.1 Vanishing and Exploding Gradients

We had seen the derivation of the gradients using chain rule. If the gradients are small, then the product can "vanish" very fast. Similarly if the gradients are larger, then the products can explode very fast.

When cost function has "cliffs", where small changes in θ, drastically change the cost function. (This usually happens if parameters are repeatedly multiplied together, as in recurrent neural networks.) Similar to exploding gradients, repeated multiplication of a matrices/vectors can cause vanishing gradients.

These problems of vanishing and exploding problems have bothered for long time in numerically training deep neural networks.

19.126.2 More on Second Order Methods

To get around the problem of having to compute and invert the Hessian, quasi-Newton methods are often used. Amongst the most well-known is the BFGS (Broyden Fletcher Goldfarb Shanno) update.Quasi-Newton methods usually require a full batch (or very large mini-batches) since errors in estimating the inverse Hessian can result in poor steps.

CG methods are also beyond the scope of this class, but we bring it up here in case helpful to look into further. Again, ECE 236C is recommended if youd like to learn more about these techniques.

- CG methods find search directions that are conjugate with respect to the Hessian, i.e., that $g_k^T H g_{k1} = 0$.
- It turns out that these derivatives can be calculated iteratively through a recurrence relation.
- Implementations of Hessian-free CG methods have been demonstrated to converge well (e.g., Martens et al., ICML 2011).

19.126.3 Further Reading

This area has many advanced reading material. Interested students may read the original papers, tutorial notes online. They are beyond the scope of this course.

Homeworks

1. Consider the following initialization for a binary classification problem (i) All weights as zero (ii) All weights as one (unit value) (iii) All weights as random. Which may do well in practice? Give your

reasons. Demonstrate it with a simple experiment. Write your experiment design and methodology on paper, and submit the corresponding code.

2. Consider a 2 input, 3 hidden layers with neurons as 3, 5, 3 respectively and 1 output neuron. All three hiddden layers have sigmoid nonlinearity.

   - Derive the backpropagation algorithm in matrix form with clear definition of matrices and the derivatives, when the loss is
     - MSE loss
     - Hinge loss
   - Implement your matrix/vector/tensor equations in python and implement a backpropagation algorithm. Test it on a synthetic data of two multivariate Gaussians in 2D. Submit loss vs iterations plot for both cases.

3. We know the notion of regularization. Let us see how to implement that:

   - Let us assume that we regularize the MSE loss with L2 norm. The new loss is now MSE + L2 norm of the weights. Derive BP algorithm and validate on your synthetic data set. Show how weights change and how the "accuracies" change with regularization.

     (You can submit plots of accuracy vs iterations for both regularized and non-regularized cases)
   - Let us assume we want to do an L0 regularization. Let us do in the following way.
     - We decay and reduce the weights systematically. Each step, we also retrain so that overall performance is not compromised. Convert this into idea into a computational procedure and validate.
     - (Read about the following two terms "weight decay" and "pruning" in neural networks)