CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 4

1 Reading

8. Optimization for Training Deep Models

SOURCE: GOODFELLOW ET AL.

The gradient specifies not the optimal step size, but the optimal direction within an infinitesimal region. **Gradient clipping** prevents any extraordinarily large steps from being taken (which run the risk of leaving the region of optimality). Gradient descent is based on making small local moves!

Vanishing gradients make it difficult to determine which direction the parameters should move in order to lower the output of the cost function, while **exploding gradients** can make learning unstable.

In the setting of SGD or related minibatch gradient-based optimization, *computation time per update* does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large.

The **momentum** algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. It is designed to accelerate learning, especially in the face of small but consistent gradients or noisy gradients. The step size will now depend on how large and how aligned a *sequence* of gradients are. It is largest when many successive gradients point in exactly the same direction.

AdaGrad individually adapts the learning rates of all model parameters by scaling them in an inversely proportional fashion to the square root of the sum of all the historical squared values of the gradient. The parameters with the largest partial derivative of the loss will therefore have the most rapid decrease in their learning rates.

RMSProp modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average. Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam ("adaptive moments") is another adaptive learning rate optimization algorithm that combines ideas from RMSProp and momentum.

Batch normalization is a method of adaptive reparametrization which significantly reduces the problem of coordinating updates across many layers. Let H be a minibatch of activations of the layer to normalize, arranged such that activations for each example appear in one row of the matrix. To normalize H, we replace it with

$$H' = \frac{H - \mu}{\sigma}$$

where μ is a vector containing the mean of each unit and σ is a vector containing the standard deviation of each unit. The rest of the network then operates on H' in the same way that the original network operated on H.

Pretraining refers to strategies that train simple models on simple tasks before moving on to harder tasks (e.g. the final task). It is sometimes more effective to train a model on a simpler task, then move on to confront the final task. It can also be more effective to train a simpler model on the final task, then make the model more complex.

Optimization

SOURCE: CS 231N

There are three key components of an image classification task:

- A (parameterized) **score function**, which maps raw pixels to class scores.
- A **loss function**, which measures the quality of a set of parameters based on how well the scores agree with the ground truth labels.
- **Optimization**, which finds the set of parameters W that minimize the loss function.

The negative gradient of the loss function (with respect to the weights) gives us the direction along which we should change our weight vector that will achieve the steepest descent (at least in the limit as the step size approaches zero).

Backpropagation

SOURCE: CS 231N

Backpropagation is a method for computing gradients through recursive application of the chain rule. The gradient ∇f of $f(x, y)$, for example, is the vector of partial derivatives $\left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$. The partial derivative on each variable tells us the rate of change of a function with respect to that variable, over an infinitesimally small region near a particular point:

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In another sense, a derivative tells us the sensitivity of the function output on a variable's value. For example, if $f(x, y)$ were xy and y were -3, then $\frac{\partial f}{\partial x} = y = -3$. This tells us that if we were to increase the value of x by a tiny amount h, the effect on the entire expression xy would be to decrease it by $-3h$.

Say that $f(x, y, z) = (x + y)z$. We can break this expression into two: $q = x + y$ and $f = qz$. Although f does not directly depend on x, the chain rule tells us that we can compute $\frac{\partial f}{\partial x}$ as

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

The chain rule serves as the foundation of backpropagation. During the forward pass, each gate in a circuit diagram gets some inputs and can immediately compute two things: (1) its output value and (2) the local gradient of its inputs with respect to its output value. During the backward pass (i.e. backpropagation), the gate will learn about the gradient of its local output value on the output of the entire circuit, and it can multiply that gradient with every local gradient it computed for its inputs.

For example, if a gate took inputs a and b and produced output c, it could compute $\partial c/\partial a$ and $\partial c/\partial b$ during the forward pass. However, say this gate were part of a larger circuit and we were *really* interested in the gradient of the eventual output f. During the backward pass, the subsequent gate would provide us with $\partial f/\partial c$ and we would be able to compute the two results

$$\frac{\partial f}{\partial a} = \frac{\partial c}{\partial a}\frac{\partial f}{\partial c} \quad \text{and} \quad \frac{\partial f}{\partial b} = \frac{\partial c}{\partial b}\frac{\partial f}{\partial c}$$

Generally, we will always assume that the gradient is with respect to the final output f, and we will say such things as "the gradient on x" to denote $\partial f/\partial x$.

Intuitively, we might think of the overall circuit as wanting to output a higher value. Say the gradient on a gate's output were -4. In our construction, the circuit would want the output of that gate to be lower, and it would want this with a "force" of 4. If the output of the gate were lower, then theoretically the final output would increase!

2 Lecture

Recap

Hinge loss penalizes data points that lie inside the margin around the decision boundary:

$$L = \max(0, 1 - yf(x))$$

A linear classifier that minimizes hinge loss is called an SVM.

Optimization

Gradient-based methods typically perform better than gradient-free methods, so we should use gradient-based methods if possible.

When we write $\nabla_W L(W)$, we mean the vector of partial derivatives w.r.t. all coordinates of W:

$$\nabla_W L(W) = \left[\tfrac{\partial L}{\partial W_1}, \tfrac{\partial L}{\partial W_2}, \ldots, \tfrac{\partial L}{\partial W_m}\right]^T$$

Note that $\partial L/\partial W_i$ measures how the loss changes w.r.t. change in W_i. When $\nabla_W L(W) = 0$, it means that all of the partials are zero, i.e. the loss is not changing in any direction, i.e. we are at a local optimum (or at least a saddle point).

To reach a minimum of loss, we should follow the negative gradient, i.e. take tiny steps in the direction

$$-\nabla_W L(W)$$

Thus each gradient descent step is defined as

$$W^{i+1} = W^i - \alpha \nabla_W L(W)$$

where α is the learning rate.

Caveat: Saddle Points

Following the negative gradient should eventually take us to a loss minimum. But it may also take a long time, and one reason for this is saddle points (where the gradient also vanishes). Minima and maxima have curvatures, i.e. the second derivative along a line. For a true minimum or maximum, the curvatures along all directions will have the same sign; for a saddle point, the curvatures will have different signs (or at least they won't all have the same sign).

A saddle point is neither a minimum nor a maximum! However, the gradient will still slow down as we approach a saddle point. Unfortunately, with neural networks there are often a lot of saddle points!

Caveat: Efficiency

The loss L is a sum of the losses of all the data items, so computing the gradient w.r.t. W requires a full pass through the dataset. Getting to the loss minimum may require millions of gradient steps, which is very expensive.

Therefore, we typically use only small subsets of samples, called minibatches, and compute a gradient only over one of these. The minibatch is ideally a random sample of size m from the full dataset, but in practice it may just be m consecutive samples. If N is our dataset size and m is our minibatch size, we can now perform N/m updates to the weights for only one pass over the dataset. Our new gradient is

$$g^{(i)} = \frac{1}{m} \sum_{j=i_1,\ldots,i_m \in \{1,\ldots,N\}} \nabla_W L(x_j, y_j, W)$$

and our new gradient update is

$$W^{(i+1)} = W^{(i)} - \alpha g^{(i)}$$

where superscripts denote iteration numbers. This algorithm is called **stochastic gradient descent**, and it's really the cornerstone of deep learning.

SGD uses $g^{(i)}$, the gradient of a minibatch, instead of the true average gradient on the full dataset. It should be clear that $\mathbb{E}[g^{(i)}] = g$, i.e. $g^{(i)}$ is an unbiased estimate of g. As a tradeoff, $g^{(i)}$ will typically have a lot of variance compared to g. On average, it will make good progress though.

Newton's Method

The Taylor series for $f(x+h)$ gives us

$$f(x+h) = f(x) + hf'(x) + O(h^2)$$

so to find a zero of f, we can set $f(x+h) = 0$ and observe that

$$h \approx -f(x)/f'(x)$$

Thus each iteration of Newton's method will perform

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

in order to find the zeros of $f(x)$ (i.e. x such that $f(x) = 0$). Conveniently, we know that at local optima of $f(x)$ we have $f'(x) = 0$. Therefore we can use

$$x_{n+1} = x_n - f'(x_n)/f''(x_n)$$

in order to find the optima of a scalar function. If $f''(x)$ is constant (i.e. f is quadratic), Newton's method will find the optimum in one step! Generally, Newton's method has a quadratic rate of convergence.

When finding the optimum of a function $f(x)$ for high-dimensional x, we want $\nabla f(x) = 0$. Since Taylor's expansion gives us

$$\nabla f(x+h) = \nabla f(x) + h^T H_f(x) + O(\|h\|^2)$$

(where H_f is the Hessian, i.e. the matrix of second-order partial derivatives of f), our update is

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

In practice, Newton's update converges quickly but is rarely used in deep learning. Why? Because it's expensive (inverting the Hessian is no simple task), it's numerically unstable (as, again, it involves a matrix inverse), and it **doesn't help**! The second-order terms in Newton's method only allow it to more quickly get to the nearest zero gradient, *which includes saddle points.*

It's a great... saddle point finder.

SGD Enhancements

- **Momentum.** Momentum addresses the problem of oscillation (failure to find the average gradient). An object's momentum is its "memory" of its motion state, implemented in practice as momentum plus a loss of energy (viscous drag). The momentum update for step i is

  $$p^{(i+1)} = \mu p^{(i)} + g^{(i)}$$

  where $p^{(i)}$ is the momentum, $g^{(i)}$ is the minibatch gradient, and $\mu \in [0,1]$ is the momentum constant. The weight update is

  $$W^{(i+1)} = W^{(i)} - \alpha p^{(i)}$$

  where α is the learning rate. Note that momentum may overshoot the target, but will generally still make it to the minimum more quickly than vanilla SGD.

- **Nesterov momentum.** We take a momentum step, and then compute the gradient at the *more updated* point (i.e. after taking the momentum step). This surprisingly simple change performs remarkably well on convex optimization problems. Its convergence rate (on convex problems) is $O(1/k^2)$ – i.e. its error is bounded by a constant times $1/k$, where k is the number of steps – which is a marked improvement over vanilla SGD's convergence rate of $O(1/k)$.
- **RMSProp.** Part of the trouble is that our gradients are very different in x and y. There might be a big gradient in the y direction, but a small gradient in the x direction. So we might want to make a scaling of the gradient; this is what RMSProp does.

  Because the gradients can vary very widely between different parameter values, we will scale each coordinate of the weights by some reasonable measure of how strong the gradient is. Our measure will be the moving average mean-squared gradient (which is something like the standard deviation of that gradient component). Specifically, we will scale the gradients by the inverse of a moving average, the RMS (root-mean-squared) gradient. If we define

  $$s^{(i+1)} = \mu s^{(i)} + (1-\mu)(g^{(i)})^2$$

  where $s^{(i)}$ is the (moving average) mean-squared gradient at step i, $g^{(i)}$ is the normal minibatch gradient at step i, $\mu \in [0,1]$ is a moving average decay factor, and $(g^{(i)})^2$ is the element-wise square of $g^{(i)}$, our RMSProp update will be as follows:

  $$W^{(i+1)} = W^{(i)} - \alpha \frac{g^{(i)}}{\sqrt{s^{(i)}}}$$

- **AdaGrad** (adaptive gradient). Instead of taking some kind of moving average of squared gradients, we'll take the cumulative sum of squared gradients. This sum just keeps growing. Here, our weight update will be

  $$W^{(i+1)} = W^{(i)} - \alpha \frac{g^{(i)}}{\sqrt{c^{(i)}}}$$

  where $c^{(i)}$ is the cumulative squared gradient at step i. Since $c^{(i)}$ tends to grow linearly over time, AdaGrad decreases its effective learning rate over time as $1/\sqrt{i}$.

Because AdaGrad and RMSProp normalize gradient magnitudes, they work very well on datasets with a wide range of gradient magnitudes. As contrast, RMS is a heuristic method while AdaGrad has formal bounds on its convergence rate (although only for convex problems). RMSProp also has a fixed learning rate and is more suitable for tasks with long-running training, while the learning rate of AdaGrad decays as $1/\sqrt{i}$ and is better for short, easy-to-train models.

Note that momentum + RMSProp $\approx$ **Adam**. In Adam, we compute moving averages of the gradient and the squared gradient, treat them as moments, and then normalize the momentum update.

Generally, learning rate should decay over time.

Caveat: Local Minima

Local minima are not as much of a problem as saddle points, because in practice most of them have similar loss to the global minimum.

Also, as network complexity increases, local minima in the loss end up being clustered more and more closely to the global minimum. They find more ways to get closer to the global optimum. Hence it's fine, even desirable, to design complex networks to mitigate the local minima problem.

Summary

- Momentum addresses the problem of oscillation, while AdaGrad and RMSProp address the issue of differences in scale of gradient components. Adam is the combination of momentum and RMSProp.

Backpropagation

Backpropagation is just the chain rule for differentiation. Our task is to find derivatives of some weights in the middle of a network, or rather the derivative of the loss of the network all the way back to the weights somewhere in the middle.

The chain rule in vector form is

$$J_L(W) = J_L(f)J_f(W)$$

where $J_f(W)$ is a Jacobian matrix. (The derivative from a vector input to a vector output is a matrix. It's called the Jacobian, but we can just think of it as a derivative.)

References

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`

[2] CS 231n optimization notes, `http://cs231n.github.io/optimization-1/`

[3] CS 231n backpropagation notes, `http://cs231n.github.io/optimization-2/`