11-685: Intro to Deep Learning - Training

Shaun Ho

February 2024

1 Derivatives

Derivatives work on the notion that at fine enough resolutions, any smooth continuous function is locally linear.

1.1 Scalar functions of scalar arguments

When dealing with scalar functions of scalar arguments, the derivative is defined as the rate of change of the function given some local x, where it is α in the relation

$$\Delta y = \alpha \Delta x$$

1.2 Scalar functions of vector arguments

We restate the scalar relation $\Delta y = \alpha \Delta x$ as:

$$\Delta y = \nabla_x y \Delta x$$

where $\nabla_x y$ the derivative is a row vector (comprising the partial derivatives of y with respect to each x_i) and Δx is a column vector. Δy is thus a scalar given by the inner product of $\nabla_x y$ and x:

$$\Delta y = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_D} \end{bmatrix} \cdot \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

$$\Delta y \approx \frac{\partial y}{\partial x_1}\Delta x_1 + \frac{\partial y}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial y}{\partial x_D}\Delta x_D$$

The **gradient** $\nabla_x y^T$ is the *transpose* of the derivative, giving it the same dimensionality as x.

Δy is a vector inner product. This makes it a function of the cosine of the angle θ between the gradient vector $\nabla_x y^T$ and the vector Δx (where the inner product of two vectors u and v is $|u||v|\cos\theta$). Since we know that the gradient vector $\nabla_x y^T$ is computed as the vector of partial derivatives of y with respect to all x_i, then we also we know that the vector inner product Δy is maximized, or in other words, the function y increases most steeply when Δx is set in such a way that its direction is aligned that of the gradient vector $\nabla_x y^T$ (since the value of $\cos\theta$ is maximized when the angle between the two vectors θ is 0. This is useful to know we would like to increase or decrease y as steeply as possible (out of the infinite directions in which we may perturb x in vector space).

1.3 Vector functions of vector arguments

As above, x is a column vector of shape $R \times 1$. If this time y is a column vector of shape $L \times 1$:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_L \end{bmatrix} = f\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_R \end{bmatrix}\right)$$

then for dimensions to match in matrix multiplication the derivative $\nabla_x y$ must be of shape $L \times R$. We call $\nabla_x y$ the Jacobian of y with respect to x.

The Jacobian is the matrix of partial derivatives of each individual output y_i against each individual input x_i, where each row is one y_i and each column is one x_i as given below:

$$J_y(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_D} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial z_1} & \frac{\partial y_M}{\partial z_2} & \cdots & \frac{\partial y_M}{\partial z_D} \end{bmatrix}$$

Note that scalar functions can also be expressed with a Jacobian where it is a diagonal matrix.

1.4 Critical points

How do we know if a point where the derivative is zero is a maximum or a minimum (or something else)?

1.4.1 Analytical approach

In the univariate/scalar case it's easy to compute $f''(x)$ and interpret from there, but it's not so simple in the multivariate case. For example, at a saddle point, a critical point could be a minimum in one direction and a maximum in another.

In the multivariate case, solve for x such that the derivative $\nabla_x f(x) = 0$, then to obtain the double derivative we compute the Hessian matrix $\nabla_x^2 f(x)$, which would give us positive eigenvalues if it is positive definite (and therefore a minimum) or negative eigenvalues if it is negative definite (and therefore a maximum) (where eigenvalues are computed locally).

The problem is that it's often simply not possible to solve for $\nabla_x f(x) = 0$ when the function to minimize/maximize may have an intractable form. In such a case, we turn to numerical solutions.

1.4.2 Numerical approach: Gradient Descent

The intuition is to set up an algorithm which takes in the local point, computes the derivative at that locality and returns the direction in the feature space in which we should move to reduce the function, stopping when the function no longer decreases.

In the vector gradient case, we know that the gradient at any x points in the direction of the steepest increase in $f(x)$. Hence, to move toward the minimum as steeply as possible, we move in the opposite direction of the gradient. Theoretically, we stop when the derivative becomes 0, but due to the possible intractability of $\nabla_x f(x)$ it is in practice more computationally efficient to stop the algorithm when either (1) the size of the next step falls below some threshold or (2) the local $\nabla_x f(x)$ falls below some threshold.

2 Operationalizing gradient descent

2.1 Objective

Our objective is to update W for every layer such that the loss function L is decreased, where the initial W is corrected by some coefficient η in the direction of steepest decrease in the loss function L. Expressed mathematically, for every component i in the network, we do at step k:

$$W_i^{k+1} = W_i^k + \eta^k \times -\frac{\delta L}{\delta W_i}$$

Until the improvement in loss over the previous iteration $|L(W^k) - L(W^{k-1})|$ falls below some threshold ϵ.

This algorithm provides us with a method for updating components in our layers in a way that minimizes the loss function. However, to rely on the information that the gradient provides us pertaining to the direction in which we should update our components, we first need to compute the derivatives with respect to every single component. There is a way of computing these derivatives in an approximate fashion which leverages the fact that layered networks are nested functions which are amenable to the application of the chain rule.

2.2 Backpropagation of gradients in nested functions using the chain rule

2.2.1 Chaining nested functions of single arguments

For the nested function
$$y = g(f(x))$$
We draw the influence diagram left to right in sequence:

$$\Delta x \longrightarrow \Delta g \longrightarrow \Delta y$$

$$\Delta x \longrightarrow \frac{dg}{dx} \longrightarrow \Delta g \longrightarrow \frac{dy}{dg} \longrightarrow \Delta y$$

We then chain right to left/last to first (with respect to the influence diagram):

$$\Delta y = \frac{dy}{dg(x)} \frac{dg(x)}{dx} \Delta x$$

2.2.2 Chaining nested functions of multiple arguments

Here, we take the sum of all partials. Expressed mathematically,
For $y = f(z_1, z_2, \ldots, z_M)$ where $z_i = g_i(x)$:

$$\Delta x \longrightarrow \Delta[z_1, z_2, \ldots, z_M] \longrightarrow \Delta y$$

$$\Delta y = \sum_i \frac{\partial y}{\partial z_i} \Delta z_i, \quad \Delta z_i = \frac{dz_i}{dx} \Delta x$$

Again, chaining last to first in terms of the inference diagram:

$$\frac{dy}{dx} = \frac{\partial y}{\partial z_1} \frac{dz_1}{dx} + \frac{\partial y}{\partial z_2} \frac{dz_2}{dx} + \ldots + \frac{\partial y}{\partial z_M} \frac{dz_M}{dx}$$

2.2.3 Chaining vector functions of vector inputs

Here, we also chain last to first in terms of the inference diagram:

$$x \longrightarrow z \longrightarrow y$$

$$\Delta x \longrightarrow \Delta z \longrightarrow \Delta y$$

$$\nabla_x y = \nabla_z y \, \nabla_x z$$

Restated as the Jacobians:

$$J_y(x) = J_y(z) \cdot J_z(x)$$

2.2.4 Chaining scalar functions of vector inputs

For a scalar function $D(y(z))$ of a vector z:

$$\nabla_z D = \nabla_y(D) J_y(z) \Delta z$$

2.3 Backpropagating the derivatives

To recap, the objective is as follows: To update W for every layer such that the loss function L is decreased, where the initial W is corrected by some coefficient η in the direction of steepest decrease in the loss function L. Expressed mathematically, for every component i in the network, we do at step k:

$$W_i^{k+1} = W_i^k + \eta^k \times -1 \times \frac{\partial L}{\partial W_i}$$

Until the improvement in loss over the previous iteration $|L(W^k) - L(W^{k-1})|$ falls below some threshold ϵ.

Moving backward with respect to the influence diagram, we compute the derivatives of the components in the order where the derivatives of the outer (later) function come first in the chain. In the case of most networks, the first derivative to be computed is the derivative of the loss with respect to the output layer, which can be obtained since the loss function has been specified and we know that the gradient of the loss function is the transpose of the derivative. Each backward computation comprises a pair of derivatives in the order of the outer function followed by the inner function, where the outer function is always previously computed and the inner function is always the new term. For example, the derivative of the loss with respect to the affine layer consists of the known outer term (the derivative of the loss with respect to the output y) and the new inner term (the derivative of the output layer with respect to the affine layer):

$$\nabla_{z_N} \mathrm{Div} = \nabla_y \mathrm{Div} \cdot \nabla_{z_N} Y$$

3 Computation of derivatives for backpropagation

Now observe how we said we needed to compute the derivatives. If we want to do so we need to figure out how a small perturbation in the first component affects the next one in the sequence. But we have to move backwards. If you want to compute how much a small perturbation in one component affects the next one, you would already have to know how much a small perturbation in the next component affects the one following that one.

3.1 (Loss) Derivative of loss with respect to network output $Y^{(N)}$

Since the loss function is something we designed, we can simply obtain the derivative of the loss function as we have specified it. This allows us to obtain the partial derivatives of the loss with respect to each network output.

3.2 (Activation) Derivative of the loss with respect to $Z^{(N)}$

What is the derivative of the output with respect to Z? That's simply the derivative of the activation function, which we also know since this was specified by us too.

$$\frac{\partial \text{Div}}{\partial Z_i^{(N)}} = \frac{\partial \text{Div}}{\partial Y_i^{(N)}} \cdot f_N' \left(Z_i^{(N)} \right)$$

3.3 (Weights) Derivative of the loss with respect to weights in $W^{(N)}$

How do we obtain the derivative of the loss with respect to **every single** weight?

We know that we can isolate the contribution of each individual weight to each $Z_j^{(N)}$:

$$Z_j^{(N)} = Y_i^{(N-1)} \cdot W_{ij}^{(N)} + \sum_{k \neq i} Y_j^{(N-1)} \cdot W_{kj}^{(N)}$$

This gives us:

$$\frac{\partial Z_j^{(N)}}{\partial W_{ij}^{(N)}} = Y_i^{(N-1)}$$

Which shows that the derivative of Z with respect to each weight equals to the output of the corresponding node at the previous layer/the input to that weight.

The intuition behind this is that $Y_i^{(N-1)}$, being passed into (and thus multiplied by) $W_{ij}^{(N-1)}$, represents the per-unit increase in $Z_j^{(N)}$ for a perturbation of $W_{ij}^{(N-1)}$. Thus we obtain for each weight $W_ij^{(N)}$:

$$\frac{\partial \text{Div}}{\partial W_{ij}^{(N)}} = Y_i^{(N-1)} \frac{\partial \text{Div}}{\partial Z_j^{(N)}}$$

Note that for the bias term, $Y_0^{(N-1)} = 1$.

3.4 (Previous Output) Derivative of the loss with respect to $Y^{(N-1)}$

Isolating again the contribution of each individual $Y^{(N-1)}$ to $Z^{(N)}$:

$$Z_j^{(N)} = Y_i^{(N-1)} \cdot W_{ij}^{(N)} + \sum_{k \neq i} Y_j^{(N-1)} \cdot W_{kj}^{(N)}$$

$$\implies \frac{\partial Z_{ij}^{(N)}}{\partial \mathrm{Y}_i^{(N-1)}} = w_{ij}^{(N)}$$

Again, the intuition is that $Y_i^{(N-1)}$ is being passed into (and thus multiplied by) $W_{ij}^{(N-1)}$. Holding $W_{ij}^{(N-1)}$ constant this time, we see that $Y_i^{(N-1)}$ thus represents the per-unit increase in $Z_j^{(N)}$ for a perturbation of $Y_i^{(N-1)}$.

As such, we have:

$$\frac{\partial \mathrm{Div}}{\partial y_i^{(N-1)}} = \sum_j w_{ij}^{(N)} \frac{\partial \mathrm{Div}}{\partial z_j^{(N)}}$$

3.5 Vector computations

For layered networks it is generally simpler to think of the process in terms of vector operations. Since matrix libraries make operations much faster and the arithmetic is much simpler, any real system states the entire process in vector terms.
Recall: The derivative of a vector function with respect to vector input is called a Jacobian.

3.5.1 (Loss) Derivative of loss with respect to network output $Y^{(N)}$

$\nabla_Y Div$ is the derivative of the divergence function.

3.5.2 (Activation) Derivative of loss with respect to $Z^{(N)}$

As before:

$$\nabla_{z_N} \mathrm{Div} = \nabla_y \mathrm{Div} \cdot \nabla_{z_N} Y$$

In vector terms, we use the Jacobian of Y with respect to $Z^{(N)}$:

$$\nabla_{z_N} \mathrm{Div} = \nabla_y \mathrm{Div} \cdot J_Y(Z_N)$$

3.5.3 (Weights) Losses with respect to $W^{(N)}$ and $B^{(N)}$

$$\nabla_{W_N} \mathrm{Div} = Y_{N-1} \nabla_{Z_N} \mathrm{Div}$$

$$\nabla_{B_N} \mathrm{Div} = \nabla_{Z_N} \mathrm{Div}$$

3.5.4 (Previous Output) Derivative of loss with respect to $Y^{(N-1)}$

$$\nabla_{Y_{N-1}}\text{Div} = \nabla_{Z_N}\text{Div} \cdot \nabla_{Y_{N-1}} Z_N$$

Applying weights by right-multiplying the weight matrix:

$$\nabla_{Y_{N-1}}\text{Div} = \nabla_{Z_N}\text{Div} \cdot W_N$$

4 Convergence

4.1 Backpropagation vs optimal classifiers

Backpropagation sometimes does not succeed in finding the optimal solution. Perceptrons can always find linear separators but backpropagation may sacrifice outlying points that are added to the feature space. The conclusion is that the addition of one training instance would force the perceptron to swing wildly in order to find the unbiased estimator, making high variance the cost of the unbiasedness. In contrast, backpropagation is a low-variance estimator at the potential cost of bias. This is not necessarily bad if outlying points represent noise.

Thus, backpropagation will often not find a separating solution even if that function is within the set of functions that is learnable by the network, if that solution is not a feasible optimum for the loss function (the loss function is minimized at some place that is not the separating solution). This resulting behavior, that a backpropagation-trained neural network classifier has lower variance than an optimal classifier for the training data, is described as a benefit.

4.2 Global minima

The above assumes that there exists a global optimum in the loss function. More often than not, there exist many unpleasant local optima and saddle points. In large networks, saddle points are much more common than local minima.

It turns out that when there are lots of local minima, they are generally equivalent, but these will tend to dominate the function which introduces bias to the solution.

Will gradient descent allow us to arrive at the global minimum? This is hard to analyze because as we have seen in the case of most MLPs, the loss function can be hideous.

Not knowing how the loss function looks like, we proceed on the heuristic that it is a convex quadratic function, with the feature space being a convex set. This is useful as we can always also write quadratic approximations to non-quadratic functions.

4.3 Convergence with gradient descent

Pertaining to these convex functions, an iterative algorithm is said to converge when it is said to monotonically or statistically arrive at the solution, but it can also oscillate and jitter or even diverge. We want to determine if our gradient descent will converge, oscillate, or diverge and what conditions lead to those behaviors.

The algorithm is as follows: Use the rule $w^k = w^{k-1} - \eta \frac{df(w)}{dw}$.

Is there a way to reach the minimum in just one step?

4.4 Optimal step size with Taylor expansion

$$f(x) \approx f(a)+f'(a)(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)}{3!}(x-a)^3+\cdots+\frac{f^{(n)}(a)}{n!}(x-a)^n$$

The intuition is that the first k derivatives of the expansion are always locally approximate to the first k derivatives of the function. We approximate a quadratic loss function as follows:

$$E(w) = E(w^{(k)}) + E'(w^{(k)})(w - w^{(k)}) + \frac{1}{2}E''(w^{(k)})(w - w^{(k)})^2$$

Differentiating and equating to 0 we obtain:

$$W_{\text{min}} = w^{(k)} - [E''(w^{(k)})]^{-1}E'(w^{(k)})$$

The optimal step size for the function $ax^2 + bx + c$ is thus:

$$\eta_{\text{opt}} = [E''(w^{(k)})]^{-1} = a^{-1} = \frac{1}{a}$$

4.5 Other step sizes

We get convergence when $\eta < 2 \cdot \eta_{\text{opt}}$.
We get oscillation when $\eta_{\text{opt}} < \eta < 2 \cdot \eta_{\text{opt}}$.
We get divergence when $\eta > 2 \cdot \eta_{\text{opt}}$.

4.6 In vector space

4.6.1 Taylor series approximation

This can still be done in the multivariate scenario. Here the optimal step size is inversely proportional to the eigenvalues of the Hessian. The inverse of the largest eigenvalue is going to be the optimal step size for the steepest direction.

4.7 Setting a learning rate

Divergence is not necessarily a bad thing, since it may allow us to escape local minima. Although it is possible to enter into another local minimum, the intuition is that if the step size is large enough, it may be large enough to enter sufficiently deep minimum points which constitute good minima.

As such, we obtain the following algorithm: Start with a divergent learning rate, while iteratively decreasing it along some schedule, such as linear, quadratic, or exponential decay.

4.8 The problem of decoupled descents

For a two-dimensional input. Then when A is diagonal:

$$E = \frac{1}{2}w^T A w + w^T b + c = \frac{1}{2}\sum_i (a_{ii}w_i^2 + b_i w_i) + c$$

This often gives us contour functions which are steeper along one axis than the other.

Since we want smaller step sizes for steeper functions (to avoid overshooting of the minima), we arrive at a problem where the optimal step size in one dimension differs from that in the other. Along dimension 1:

$$E = \frac{1}{2}a_{11}w_1^2 + b_1 w_1 + c + C(-w_1) \implies \eta_{1,\text{opt}} = a_{11}^{-1}$$

Along dimension 2:

$$E = \frac{1}{2}a_{22}w_2^2 + b_2 w_2 + c + C(-w_2) \implies \eta_{2,\text{opt}} = a_{22}^{-1}$$

Using the same learning rate could result in convergence along one axis and divergence along another, or extremely slow convergence along one direction, and even then there is no guarantee for monotonic convergence. Thus, the challenge pertains to the *ratio* between the optimal step sizes of each dimension. In general, this becomes more and more challenging as the number of features increases.

5 Optimizing convergence with learning rates

How do we decide the optimal learning rates for each direction? We can think of it in two ways:

1. Manipulating the derivative itself
2. Manipulating the step size
3. Some combination of the two.

5.1 RProp

RProp or resilient propagation is a derivative-inspired algorithm that operates independently for each feature, and makes no convexity assumption:

1. If the sign of the derivative has not changed between steps, it is recommending that we continue in the same direction, hence we increase the step size. We heuristically place a ceiling on the step size.

2. If the sign has changed between steps, we revert to the previous step and proceed with a smaller step size. We heuristically place a floor on the step size.

5.2 QuickProp

QuickProp is a less-used but remains one of the best algorithms to date.

5.3 Momentum methods

Momentum methods maintain a running average of the instantaneous derivatives obtained at each iteration:

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

The momentum factor β, often 0.9, demonstrates the propensity of the next step to be influenced by the running average (as opposed to the instantaneous derivative). In directions where there is smooth convergence, the running average is a large value. In directions where there are oscillations, the frequent sign changes during oscillation would cancel themselves out in the running average computation, thus reducing ΔW, the step size. This mechanism compresses oscillations induced by step sizes along unsteady dimensions while maintaining relatively larger step sizes for the dimensions where derivatives are steadier. **Nesterov's Accelerated Gradient** operates by first extending the previous step, then adds the gradient step at the resultant position, in that order.

6 Optimizing convergence with minibatches

Under the vanilla formulation of a gradient descent training loop, the algorithm iteratively measures losses over the entire set of training data before making the weight updates, where there is one update for a training set of T samples. Since this requires the function to be adjusted at all training points simultaneously, this is computationally expensive since incremental updates to the approximated function move slowly.

6.1 Update the function with each training point (SGD)

An alternative formulation of the training loop would update the weight parameters after the divergence is computed for each single point in the training set, where there are T updates over one epoch (pass) over a training set of T samples. Pertinently, training data is randomly permuted in order to avoid cyclic behavior in the function update.

SGD is said to converge faster because the function gets T times as many updates in the training loop as would be under the batch gradient descent algorithm. The improvement is pronounced when the data exhibit low variance, thus speeding up the rate at which the entire relevant feature space is updated with each T-th update. For larger variations within the training data, the effect of T additional updates is diluted since it is distributed over a broader feature space, although the benefits do not entirely vanish.

The risk of the single-instance update training loop "chasing" the last input by updating the function to minimize errors on the latest instance only can be ameliorated by iteratively shrinking the rate at which the function updates with each training instance (the learning rate).

Specifically, the sum of the step sizes must be infinite in the limit to allow the entire parameter space to be searched. To account for the constraint that the steps must shrink, we indicate that the sum of squared step sizes must be finite in the limit.

This gives us Stochastic Gradient Descent.

6.2 Variance problems with SGD

While SGD is seen to converge one or two orders of magnitude quicker than batch gradient descent, it does so to a poorer minimum and more importantly, with a significantly larger variation between runs. This behavior is explained by the way we constructed the loss function as an estimator of the true function, and the resulting sensitivities to the size of the training sets used to perform the batch update. In particular, we will observe that variations in the loss, and thus the parameter updates, are inversely proportional to the size of the training set seen before each update.

Specifically, instead of learning the weights $\hat{W}$ that minimize the expected divergence between our approximation $f(X;W)$ and the true function $g(X)$ as follows:

$$\hat{W} = \underset{W}{\operatorname{argmin}}\, E\left[\operatorname{div}(f(\mathbf{x};W), g(\mathbf{x}))\right]$$

We minimize the loss between our approximation f and a finite set of training points d which we believe represents the underlying function g:

$$\text{Loss}(W) = \frac{1}{N}\sum_{i=1}^{N} \text{div}(f(\mathbf{x}_i; W), d_i)$$

$$\hat{W} = \underset{W}{\text{argmin}}\ \text{Loss}(W)$$

Where the loss (empirical risk) is an unbiased estimator of the expected divergence to the true function:

$$E[\text{Loss}(W)] = E[\text{div}(f(\mathbf{X}; W), g(\mathbf{X}))]$$

It can be shown that the variance of the empirical risk is an estimator of the divergence to the true function as given by this relation:

$$Var(\text{Loss}) = \frac{1}{N} Var(\text{Div})$$

Crucially, the variance of the estimator is inversely proportional to the size of the training set. The larger the variance, the greater the likelihood that $\hat{W}$ obtained under training loss minimization will differ significantly from the $\hat{W}$ that minimizes the expected divergence to the underlying function.

Under SGD, which computes the divergence on a sample-by-sample basis, the sample error (to the training point) remains an unbiased estimate of the expected divergence (to the true function). Turning to the variance, the variance of the divergence that results from SGD updates over N independent and identically distributed training points is N times that of the variance of the empirical average minimized under batch update.

The intuition of the above is as follows: Feeding too few training samples to the model before it updates makes its loss estimate swing wildly from one sample position to another. Since our estimator minimizes its loss estimate by updating $\hat{W}$, the learned $\hat{W}$ too can swing wildly.

6.3 Optimizing batch sizes with mini-batch updates

We manage the trade-off between speed of convergence and instability in the resulting estimates by using larger subsets of samples for each model update. As before, we conduct random permutations in the training set to avoid the issues of cyclic behavior.

The minibatch loss is also an unbiased estimate of the expected divergence:

$$E[\text{MinibatchLoss}(W)] = E[\text{div}(f(\mathbf{X}; W), g(\mathbf{X}))]$$

Pertinently, the variance of the minibatch loss is smaller:

$$Var(\text{Loss}) = \frac{1}{b} Var(\text{Div})$$

Where b is the batch size, making it b times more stable than the variance of the divergence under SGD (where sample size is 1).

Although there is in theory a degradation in the convergence rate compared to SGD, with the right learning rates, mini-batches become more effective in practice where loss functions are generally not convex. Computationally, there are also benefits from the ability to use vector processing. Empirically, it performs similarly to batch learning with an improvement in convergence speed of orders of magnitude (when estimating loss as testing error or average batch-level losses).

6.3.1 Optimal minibatch size

Given the relation:

$$Var(\text{Loss}) = \frac{1}{b} Var(\text{Div})$$

We see that the benefit of minibatching (decreased variability in the divergence) is reaped quickly as the batch size increases. There must exist some b in the elbow of the curve for which there is optimization with respect to the increased convergence speed of a smaller batch and the increased stability of a larger one, but this is difficult to compute analytically. In practice, the mini-batch size is generally set to the largest that can be supported by the relevant hardware, in memory, without compromising overall compute time.

6.4 Learning algorithms for minibatch convergence

The simple technique would be to fix the learning rate until the error plateaus, and then reduce the learning rate by a fixed factor such as 10. More advanced methods provide adaptive updates where the learning rate is determined as part of the overall estimation.

6.4.1 Momentum methods

As stated above, momentum methods keep a running average of the derivatives computed at each training iteration with the effect that step sizes get longer in dimensions where the gradient retains the same sign, and shorter in dimensions where the gradient oscillates. They operate similarly as described above.

6.4.2 Learning rate methods

Unlike momentum methods which operate on the step size by fixing gradients independently across dimensions, learning rate methods influence the step size by adjusting learning rates independently across dimensions. These methods rely on (1) the direction of oscillation and/or (2) the magnitude of absolute movement in one direction.

RMSProp takes this into account by taking a running mean of the squared derivative at each stage of computation. The update rule scales down learning rates for terms with large mean squared derivatives and scales up components with small mean squared derivatives.

First, the weighted running mean is computed with γ determining the weight of past instances in computing the mean:

$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1-\gamma)(\partial_w^2 D)_k$$

Next, the step size is determined as before, by applying a learning rate to the instantaneous derivative, only this time with the learning rate being adjusted by the running mean:

$$W_{k+1} = W_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$

Typical parameters include $\eta = 0.001, \delta = 0.9$.

6.4.3 Methods combining learning rate and derivative adjustments

ADAM combines the derivative-smoothing property of Nesterov's momentum with the learning-rate smoothing property of RMSProp.

The smoothed derivative:

$$m_k = \delta m_{k-1} + (1-\delta)(\partial_w D)_k$$

The smoothed learning rate:

$$v_k = \gamma v_{k-1} + (1-\gamma)(\partial_w^2 D)_k$$

Note that at the outset, the running means δm_{k-1} and γv_{k-1} are 0 as initialized. Given further that the weight parameters δ and γ are often close to 1 (to give effect to an intention for most weight to be placed on historical instances of the derivative as opposed to the instantaneous derivative), we observe that the value of the smoothed terms m_k and v_k will stay close to the initial running mean 0 for a significant burn-in period.

To overcome this issue, we adjust the running means as follows:

$$\hat{m}_k = \frac{m_k}{1-\delta^k}$$

$$\hat{v}_k = \frac{v_k}{1-\gamma^k}$$

Where raising δ and γ to the power of k ensures that the denominator term is small at the outset (thus emphasizing the weight of instantaneous derivatives in the early stages of training) and close to 1 as the iteration number k becomes large, thereby implementing a more stable transition into heavily weighting the running mean.

Then, the final adjustment rule is as follows:

$$W_{k+1} = W_k - \frac{\eta}{\sqrt{\hat{v}_k} + \epsilon} \hat{m}_k$$

Typical parameters include $\eta = 0.001, \delta = 0.9, \gamma = 0.999$.