P1. ADADELTA: AN ADAPTIVE LEARNING RATE METHOD, Zeiler et al., 2012

Idea: The idea presented in this paper was derived from ADAGRAD in order to improve upon the two main drawbacks of the method:

1) the continual decay of learning rates throughout training, and
2) the need for a manually selected global learning rate.

In the ADAGRAD method the denominator accumulates the squared gradients from each iteration starting at the beginning of training. Since each term is positive, this accumulated sum continues to grow throughout training, effectively shrinking the learning rate on each dimension. After many iterations, this learning rate will become infinitesimally small. Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w. Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{1}$$

set γ to a similar value as the momentum term, around 0.9.
Vanilla SGD update is :

$$\Delta\theta_t = -\eta \cdot g_{t,i} \tag{2}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{3}$$

The parameter update vector of Adagrad :

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{4}$$

Now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{5}$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient,

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{6}$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates.

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \tag{7}$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{8}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in AdaGrad update formula with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule.
AdaDelta Update formula

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{9}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{10}$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.
The benefits of ADADELTA are as follows:

• No manual setting of a learning rate.
• Insensitive to hyperparameters.
• Separate dynamic learning rate per-dimension.
• Minimal computation over gradient descent.
• Robust to large gradients, noise and architecture choice.
• Applicable in both local or distributed environments.

While AdaDelta is assumed to be parameter-free, there are some hyperparameters that need to be carefully tuned in order to exploit the most out of the algorithm. These parameters include ρ in running averages and the ϵ constant. However, mispecification of such hyperparameters is less sensitive to erroneous performance, compared to mispecificition of the learning rates.

Having no explicit annealing schedule imposed on the learning rate could be why momentum with the proper hyperparameters outperforms ADADELTA later in training as seen in above figure. With momentum, oscillations that can occur near a minima are smoothed out, whereas with ADADELTA these can accumulate in the numerator. An annealing schedule could possibly be added to the ADADELTA method to counteract this in future work.
Section 4.3 in the paper explains that when the algorithm approaches the optimal value, both the gradients and the δx values become smaller and smaller. Consequently, both $E\left[g^2\right]_t$ and $E[\Delta\theta^2]_{t-1}$ both become much smaller than ϵ and so

$$\frac{\text{RMS}\left[\Delta x\right]_{t-1}}{\text{RMS}\left[g\right]_t} = \frac{\sqrt{E\left[\Delta x^2\right]_{t-1} + \varepsilon}}{\sqrt{E\left[g^2\right]_t + \varepsilon}} \tag{11}$$

approaches 1 , i.e. the learning rate approaches 1.
This is not as bad as the learning rate getting uncontrollably larger, but the authors of the paper admit that 1 might be a relatively high learning rate. Thus, they conclude that adding an annealing schedule to ADADELTA might be a good idea.
So a annealing schedule in this case could just be a form of a conditional where if gradients and δx values becomes very less than ϵ then we fix the learning rate for the further process to a value near to 10^{-4} or find the optimal value just like the vanilla SGD case. Here since the learning is low so chances of missing the optimal point is less as compared to the the learning rate of 1 as per Adadelta formula.

Conclusion

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule. ADADELTA is a new learning rate method based on only first order information which shows promising result on MNIST and a large scale Speech recognition dataset.

P2. Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe et al 2015

The range of values of raw training data often varies widely Example: Bool feature in 0,1 Value of car: 500-100k dollars In machine learning algorithms, the functions involved in the optimization process are sensitive to normalization For example: Distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. After, normalization, each feature contributes approximately proportionately to the final distance. In general, Gradient descent converges much faster with feature scaling than without it. Good practice for numerical stability for numerical calculations, and to avoid ill-conditioning when solving systems of equations.
Two methods are usually used for rescaling or normalizing data:
(i) Scaling data all numeric variables to the range [0,1]. One possible formula is given below:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{12}$$

(ii) To have zero mean and unit variance:

$$x_{new} = \frac{x - \mu}{\sigma} \tag{13}$$

Internal Covariate shift: Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with

saturating nonlinearities. We refer to this phenomenon as internal covariate shift, and address the problem by normalizing layer inputs

Inorder to reduce internal covariate shift Batch Normalization method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout

BN reduces Covariate Shift. That is the change in distribution of activation of a component. By using BN, each neuron's activation becomes (more or less) a Gaussian distribution, i.e. its usually not active, sometimes a bit active, rare very active. Covariate Shift is undesirable, because the later layers have to keep adapting to the change of the type of distribution (instead of just to new distribution parameters, e.g. new mean and variance values for Gaussian distributions). BN reduces effects of exploding and vanishing gradients, because every layer becomes roughly normal distributed. Without BN, low activations of one layer can lead to lower activations in the next layer, and then even lower ones in the next layer and so on.

Batch Normalization also makes training more resilient to the parameter scale. Normally, large learning rates may increase the scale of layer parameters, which then amplify the gradient during backpropagation and lead to the model explosion However, with Batch Normalization, backpropagation through a layer is unaffected by the scale of its parameters, i.e. The BN transformation is scalar invariant

BN reduces demand for regularization, e.g. dropout or L2 norm. Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch. I.e. the network can no longer just memorize values and their correct answers.) BN allows higher learning rates. (Because of less danger of exploding/vanishing gradients.)

Algorithm Summary: Normalization via Mini-Batch Statistics

(i) Each feature (component) is normalized individually (ii) Normalization according to:

$$componentNormalizedValue = \frac{componentOldValue - E[component]}{\sqrt{(Var(component)}} \tag{14}$$

(iii) A new layer is added so the gradient can see the normalization and made adjustments if needed. The new layer has the power to learn the identity function to de-normalize the features if necessary.

$$newValue = \gamma * componentNormalizedValue + \beta \tag{15}$$

γ and β are learned per component. E and Var are estimated for each mini batch. BN is fully differentiable. Formulas for gradients/backpropagation can be easily derived.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 1:

Application and Conclusion

Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batchnormalized networks, authors improved upon the then best published result on ImageNet classification: reaching 4.9 % top-5 validation error (and 4.8 % test error)