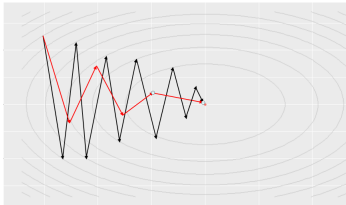


Deep Learning

Advanced Optimization



Learning goals

-
-
-

Momentum

MOMENTUM

- While SGD remains a popular optimization strategy, learning with it can sometimes be slow.
- Momentum is designed to accelerate learning, especially when facing high curvature, small but consistent or noisy gradients.
- Momentum accumulates an exponentially decaying moving average of past gradients:

$$\begin{aligned}\nu &\leftarrow \varphi \nu - \underbrace{\alpha \nabla_{\theta} \left[\frac{1}{m} \sum_i L(y^{(i)}, f(x^{(i)}, \theta)) \right]}_{\mathbf{g}_{\theta}} \\ \theta &\leftarrow \theta + \nu\end{aligned}$$

- We introduce a new hyperparameter $\varphi \in [0, 1)$, determining how quickly the contribution of previous gradients decay.
- ν is called “velocity” and derives from a physical analogy describing how particles move through a parameter space (Newton’s law of motion).

MOMENTUM

- So far the step size was simply the gradient \mathbf{g} multiplied by the learning rate α .
- Now, the step size depends on how **large** and how **aligned** a sequence of gradients is. The step size grows when many successive gradients point in the same direction.
- Common values for φ are 0.5, 0.9 and even 0.99.
- Generally, the larger φ is relative to the learning rate α , the more previous gradients affect the current direction.
- A very good website with an in-depth analysis of momentum:
<https://distill.pub/2017/momentum/>

MOMENTUM: EXAMPLE

$$\nu_1 \leftarrow \varphi \nu_0 - \alpha g(\theta^{[0]})$$

$$\theta^{[1]} \leftarrow \theta^{[0]} + \varphi \nu_0 - \alpha g(\theta^{[0]})$$

$$\nu_2 \leftarrow \varphi \nu_1 - \alpha g(\theta^{[1]})$$

$$= \varphi(\varphi \nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})$$

$$\theta^{[2]} \leftarrow \theta^{[1]} + \varphi(\varphi \nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})$$

$$\nu_3 \leftarrow \varphi \nu_2 - \alpha g(\theta^{[2]})$$

$$= \varphi(\varphi(\varphi \nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]})$$

$$\theta^{[3]} \leftarrow \theta^{[2]} + \varphi(\varphi(\varphi \nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]})$$

$$= \theta^{[2]} + \varphi^3 \nu_0 - \varphi^2 \alpha g(\theta^{[0]}) - \varphi \alpha g(\theta^{[1]}) - \alpha g(\theta^{[2]})$$

$$= \theta^{[2]} - \alpha(\varphi^2 g(\theta^{[0]}) + \varphi^1 g(\theta^{[1]}) + \varphi^0 g(\theta^{[2]})) + \varphi^3 \nu_0$$

$$\theta^{[t+1]} = \theta^{[t]} - \alpha \sum_{j=0}^t \varphi^j g(\theta^{[t-j]}) + \varphi^{t+1} \nu_0$$

MOMENTUM: EXAMPLE

Suppose momentum always observes the same gradient $g(\theta)$:

$$\begin{aligned}\theta^{[t+1]} &= \theta^{[t]} - \alpha \sum_{j=0}^t \varphi^j g(\theta^{[j]}) + \varphi^{t+1} \nu_0 \\ &= \theta^{[t]} - \alpha g(\theta) \sum_{j=0}^t \varphi^j + \varphi^{t+1} \nu_0 \\ &= \theta^{[t]} - \alpha g(\theta) \frac{1 - \varphi^{t+1}}{1 - \varphi} + \varphi^{t+1} \nu_0 \\ &\rightarrow \theta^{[t]} - \alpha g(\theta) \frac{1}{1 - \varphi} \quad \text{for } t \rightarrow \infty.\end{aligned}$$

Thus, momentum will accelerate in the direction of $-g(\theta)$ until reaching terminal velocity with step size:

$$-\alpha g(\theta)(1 + \varphi + \varphi^2 + \varphi^3 + \dots) = -\alpha g(\theta) \frac{1}{1 - \varphi}$$

E.g. a momentum with $\varphi = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

MOMENTUM: ILLUSTRATION

The vector ν_3 (for $\nu_0 = 0$):

$$\begin{aligned}\nu_3 &= \varphi(\varphi(\varphi\nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]}) \\ &= -\varphi^2(\alpha g(\theta^{[0]})) - \varphi(\alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]})\end{aligned}$$

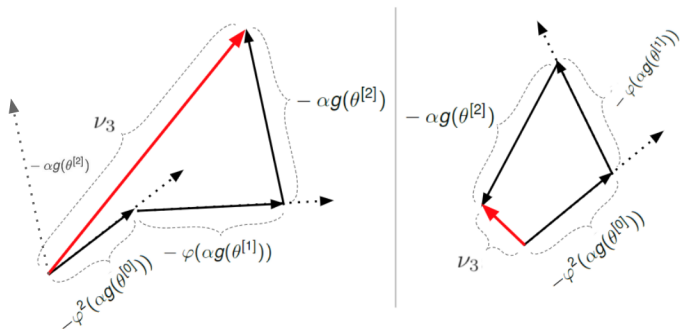


Figure: If consecutive (negative) gradients point mostly in the same direction, the velocity "builds up". On the other hand, if consecutive (negative) gradients point in very different directions, the velocity "dies down".

SGD WITH MOMENTUM

Algorithm 1 Stochastic gradient descent with momentum

- 1: **require** learning rate α and momentum φ
 - 2: **require** initial parameter θ and initial velocity ν
 - 3: **while** stopping criterion not met **do**
 - 4: Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \dots, \tilde{x}^{(m)}\}$
 - 5: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(y^{(i)}, f(\tilde{\mathbf{x}}^{(i)} | \theta))$
 - 6: Compute velocity update: $\nu \leftarrow \varphi \nu - \alpha \hat{\mathbf{g}}$
 - 7: Apply update: $\theta \leftarrow \theta + \nu$
 - 8: **end while**
-

SGD WITH MOMENTUM

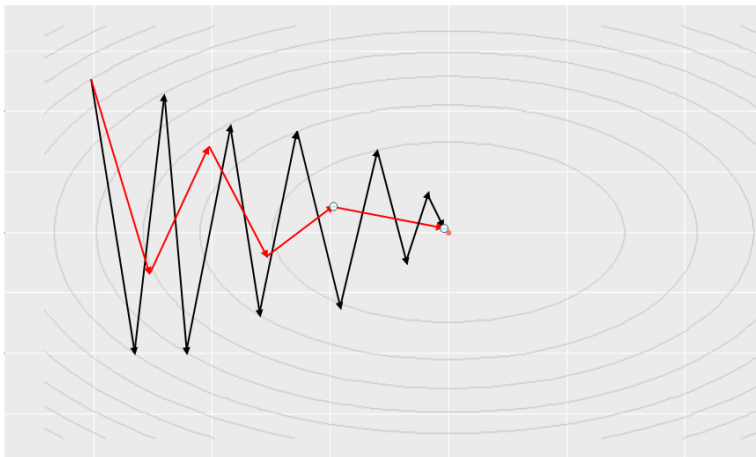


Figure: The contour lines show a quadratic loss function with a poorly conditioned Hessian matrix. The two curves show how standard gradient descent (black) and momentum (red) learn when dealing with ravines. Momentum reduces the oscillation and accelerates the convergence.

SGD WITH AND WITHOUT MOMENTUM

The following plot was created by our Shiny App. On the upper left you can explore different predefined examples. [▶ Click here](#)

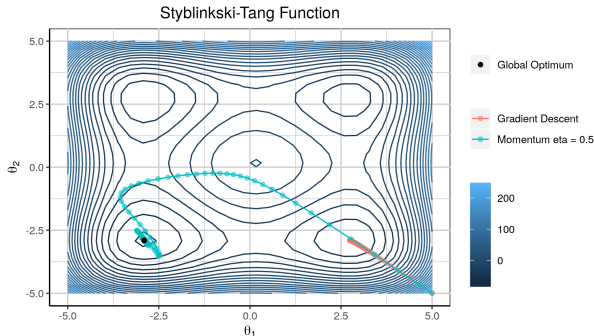
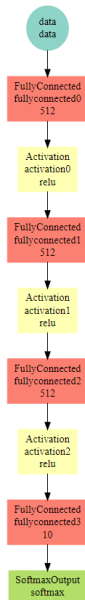


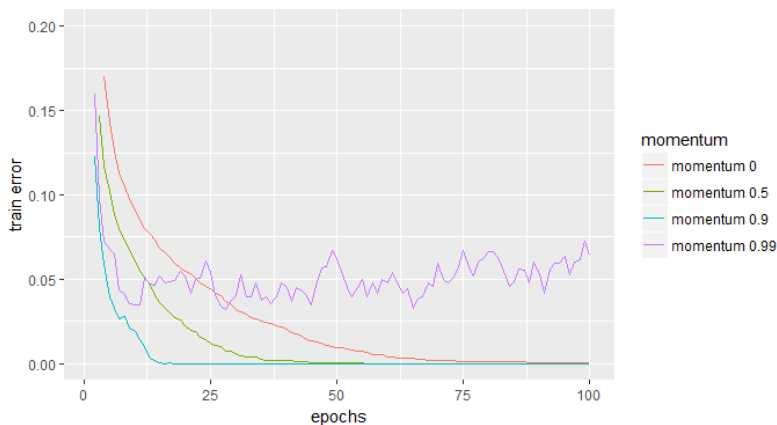
Figure: Comparison of SGD with and without momentum on the Styblinski-Tang function. The black dot on the bottom left is the global optimum. We can see that SGD without momentum (red line/points) cannot escape the local minimum, while SGD with momentum (blue line/dots) is able to escape the local minimum and finds the global minimum.

MOMENTUM IN PRACTICE

- Lets try out different values of momentum (with SGD) on the MNIST data.
- We apply the same architecture we have used a dozen of times already (note that we used $\varphi = 0.9$ in all computations so far, i.e. in chapter 1 and 2)!

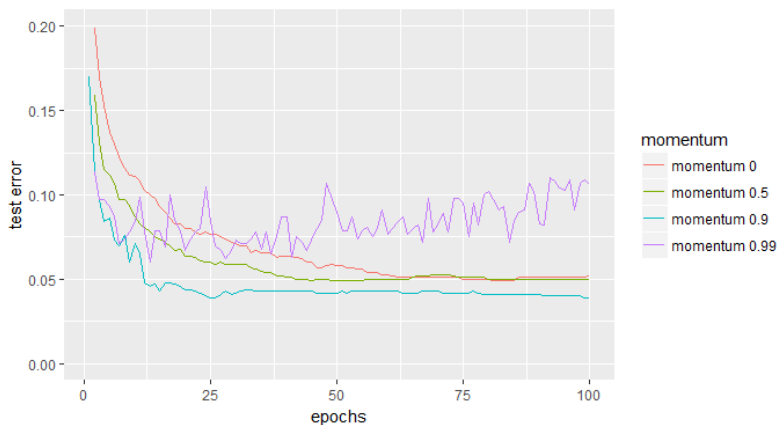


MOMENTUM IN PRACTICE



The higher momentum, the faster SGD learns the weights on the training data, but if momentum is too large, the training and test error fluctuates.

MOMENTUM IN PRACTICE



The higher momentum, the faster SGD learns the weights on the training data, but if momentum is too large, the training and test error fluctuates.

NESTEROV MOMENTUM

- Momentum aims to solve poor conditioning of the Hessian but also variance in the stochastic gradient.
- Nesterov momentum modifies the algorithm such that the gradient is evaluated after the current velocity is applied:

$$\begin{aligned}\nu &\leftarrow \varphi\nu - \alpha \nabla_{\theta} \left[\frac{1}{m} \sum_i L(y^{(i)}, f(x^{(i)}, \theta + \varphi\nu)) \right] \\ \theta &\leftarrow \theta + \nu\end{aligned}$$

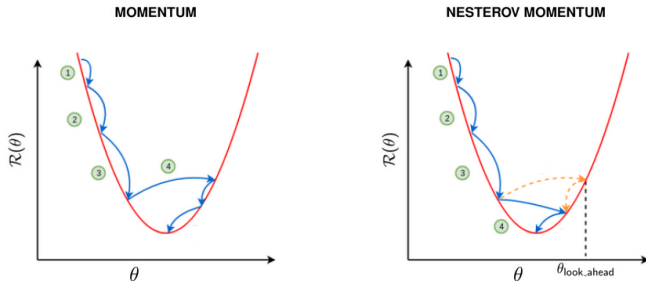
- We can interpret Nesterov momentum as an attempt to add a correction factor to the basic method.
- The method is also called Nesterov accelerated gradient (NAG).

SGD WITH NESTEROV MOMENTUM

Algorithm 2 Stochastic gradient descent with Nesterov momentum

- 1: **require** learning rate α and momentum φ
 - 2: **require** initial parameter θ and initial velocity ν
 - 3: **while** stopping criterion not met **do**
 - 4: Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \dots, \tilde{x}^{(m)}\}$
 - 5: Apply interim update: $\tilde{\theta} \leftarrow \theta + \varphi \nu$
 - 6: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(y^{(i)}, f(x^{(i)}, \tilde{\theta}))$
 - 7: Compute velocity update: $\nu \leftarrow \varphi \nu - \alpha \hat{\mathbf{g}}$
 - 8: Apply update: $\theta \leftarrow \theta + \nu$
 - 9: **end while**
-

MOMENTUM VS. NESTEROV MOMENTUM



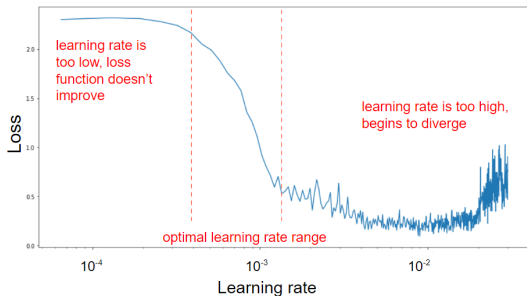
Credits: Chandra (2015)

Figure: Comparison GD with momentum (left) and GD with Nesterov momentum (right) for one parameter θ . The first three updates of θ are very similar in both cases and the updates become larger due to momentum (accumulation of previous negative gradients). Update 4 is different. In case of momentum, the update overshoots as it makes an even bigger step due to the gradient history. In contrast, Nesterov momentum first evaluates a "look-ahead" point $\theta_{\text{look_ahead}}$, detects that it overshoots, and slightly reduces the overall magnitude of the fourth update. Thus, Nesterov momentum reduces overshooting and leads to smaller oscillations than momentum.

Learning Rates

LEARNING RATE

- The learning rate is a very important hyperparameter.
- To systematically find a good learning rate, we can start at a very low learning rate and gradually increase it (linearly or exponentially) after each mini-batch.
- We can then plot the learning rate and the training loss for each batch.
- A good learning rate is one that results in a steep decline in the loss.



Credit: jeremyjordan

LEARNING RATE SCHEDULE

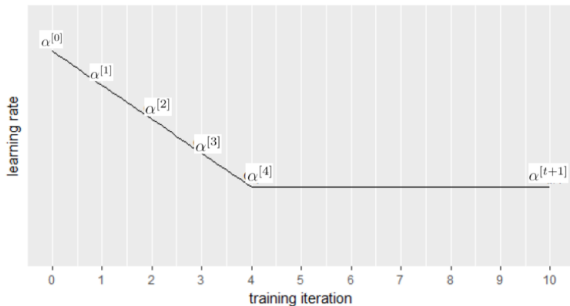
- We would like to force convergence until reaching a local minimum.
- Applying SGD, we have to decrease the learning rate over time, thus $\alpha^{[t]}$ (learning rate at training iteration t).
 - The estimator $\hat{\mathbf{g}}$ is computed based on small batches.
 - Random sampling m training samples introduces noise, that does not vanish even if we find a minimum.
- In practice, a common strategy is to decay the learning rate linearly over time until iteration τ :

$$\alpha^{[t]} = \begin{cases} \left(1 - \frac{t}{\tau}\right) \alpha^{[0]} + \frac{t}{\tau} \alpha^{[\tau]} = t \left(-\frac{\alpha^{[0]} - \alpha^{[\tau]}}{\tau}\right) + \alpha^{[0]} & \text{for } t \leq \tau \\ \alpha^{[\tau]} & \text{for } t > \tau \end{cases}$$

LEARNING RATE SCHEDULE

Example for $\tau = 4$:

iteration t	t/τ	$\alpha^{[t]}$
1	0.25	$(1 - \frac{1}{4}) \alpha^{[0]} + \frac{1}{4} \alpha^{[\tau]} = \frac{3}{4} \alpha^{[0]} + \frac{1}{4} \alpha^{[\tau]}$
2	0.5	$\frac{2}{4} \alpha^{[0]} + \frac{2}{4} \alpha^{[\tau]}$
3	0.75	$\frac{1}{4} \alpha^{[0]} + \frac{3}{4} \alpha^{[\tau]}$
4	1	$0 + \alpha^{[\tau]}$
...		$\alpha^{[\tau]}$
$t + 1$		$\alpha^{[\tau]}$

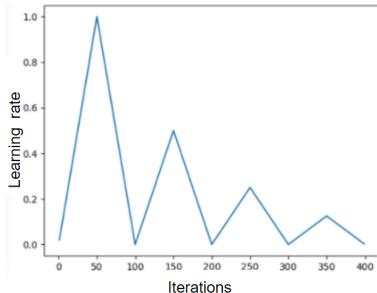
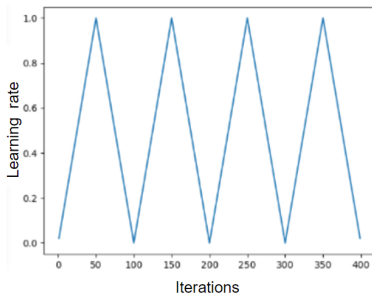


CYCLICAL LEARNING RATES

- Another option is to have a learning rate that periodically varies according to some cyclic function.
- Therefore, if training does not improve the loss anymore (possibly due to saddle points), increasing the learning rate makes it possible to rapidly traverse such regions.
- Recall, saddle points are far more likely than local minima in deep nets.
- Each cycle has a fixed length in terms of the number of iterations.

CYCLICAL LEARNING RATES

- One such cyclical function is the "triangular" function.

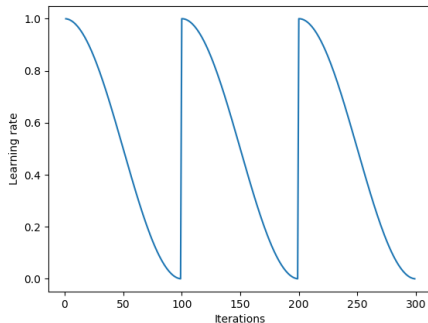


Credit: Hafidz Zulkifli

- In the right image, the range is cut in half after each cycle.

CYCLICAL LEARNING RATES

- Yet another option is to abruptly "restart" the learning rate after a fixed number of iterations.
- Loshchilov et al. (2016) proposed "cosine annealing" (between restarts).



Credit: Hafidz Zulkifli

Algorithms with Adaptive Learning Rates

ADAPTIVE LEARNING RATES

- The learning rate is reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on the models performance.
- Naturally, it might make sense to use a different learning rate for each parameter, and automatically adapt them throughout the training process.

ADAGRAD

- Adagrad adapts the learning rate to the parameters.
- In fact, Adagrad scales learning rates inversely proportional to the square root of the sum of the past squared derivatives.
 - Parameters with large partial derivatives of the loss obtain a rapid decrease in their learning rate.
 - Parameters with small partial derivatives on the other hand obtain a relatively small decrease in their learning rate.
- For that reason, Adagrad might be well suited when dealing with sparse data.
- Goodfellow et al. (2016) say that the accumulation of squared gradients can result in a premature and overly decrease in the learning rate.

ADAGRAD

Algorithm 3 Adagrad

```
1: require Global learning rate  $\alpha$ 
2: require Initial parameter  $\theta$ 
3: require Small constant  $\beta$ , perhaps  $10^{-7}$ , for numerical stability
4: Initialize gradient accumulation variable  $\mathbf{r} = 0$ 
5: while stopping criterion not met do
6:   Sample a minibatch of  $m$  examples from the training set  $\{\tilde{\mathbf{x}}^{(1)}, \dots, \tilde{\mathbf{x}}^{(m)}\}$ 
7:   Compute gradient estimate:  $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(y^{(i)}, f(\tilde{\mathbf{x}}^{(i)} | \theta))$ 
8:   Accumulate squared gradient  $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ 
9:   Compute update:  $\nabla \theta = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$  (division and square root applied element-wise)
10:  Apply update:  $\theta \leftarrow \theta + \nabla \theta$ 
11: end while
```

- “ \odot ” is called Hadamard or element-wise product.
- Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, \text{ then } A \odot B = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix}$$

RMSPROP

- RMSprop is a modification of Adagrad.
- It's intention is to resolve Adagrad's radically diminishing learning rates.
- The gradient accumulation is replaced by an exponentially weighted moving average.
- Theoretically, that leads to performance gains in non-convex scenarios.
- Empirically, RMSProp is a very effective optimization algorithm. Particularly, it is employed routinely by deep learning practitioners.

RMSPROP

Algorithm 4 RMSProp

- 1: **require** Global learning rate α and decay rate ρ
 - 2: **require** Initial parameter θ ,
 - 3: **require** Small constant β , perhaps 10^{-6} , to stabilize division by small numbers
 - 4: Initialize gradient accumulation variable $r = 0$
 - 5: **while** stopping criterion not met **do**
 - 6: Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \dots, \tilde{x}^{(m)}\}$
 - 7: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(y^{(i)}, f(\tilde{\mathbf{x}}^{(i)} | \theta))$
 - 8: Accumulate squared gradient $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
 - 9: Compute update: $\nabla_{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
 - 10: Apply update: $\theta \leftarrow \theta + \nabla_{\theta}$
 - 11: **end while**
-

ADAM

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
- Adam uses the first and the second moments of the gradients.
 - Adam keeps an exponentially decaying average of past gradients (first moment).
 - Like RMSProp it stores an exponentially decaying average of past squared gradients (second moment).
 - Thus, it can be seen as a combination of RMSProp and momentum.
- Basically Adam uses the combined averages of previous gradients at different moments to give it more “persuasive power” to adaptively update the parameters.

ADAM

Algorithm 5 Adam

- 1: **require** Step size α (suggested default: 0.001)
 - 2: **require** Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$ (suggested defaults: 0.9 and 0.999 respectively)
 - 3: **require** Small constant β (suggested default 10^{-8})
 - 4: **require** Initial parameters θ
 - 5: Initialize time step $t = 0$
 - 6: Initialize 1st and 2nd moment variables $\mathbf{s}^{[0]} = \mathbf{0}, \mathbf{r}^{[0]} = \mathbf{0}$
 - 7: **while** stopping criterion not met **do**
 - 8: $t \leftarrow t + 1$
 - 9: Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \dots, \tilde{x}^{(m)}\}$
 - 10: Compute gradient estimate: $\hat{\mathbf{g}}^{[t]} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(y^{(i)}, f(\tilde{\mathbf{x}}^{(i)} | \theta))$
 - 11: Update biased first moment estimate: $\mathbf{s}^{[t]} \leftarrow \rho_1 \mathbf{s}^{[t-1]} + (1 - \rho_1) \hat{\mathbf{g}}^{[t]}$
 - 12: Update biased second moment estimate: $\mathbf{r}^{[t]} \leftarrow \rho_2 \mathbf{r}^{[t-1]} + (1 - \rho_2) \hat{\mathbf{g}}^{[t]} \odot \hat{\mathbf{g}}^{[t]}$
 - 13: Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}^{[t]}}{1 - \rho_1^t}$
 - 14: Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}^{[t]}}{1 - \rho_2^t}$
 - 15: Compute update: $\nabla \theta = -\alpha \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \beta}}$
 - 16: Apply update: $\theta \leftarrow \theta + \nabla \theta$
 - 17: **end while**
-

ADAM

- Adam initializes the exponentially weighted moving averages \mathbf{s} and \mathbf{r} as $\mathbf{0}$ (zero) vectors.
- As a result, they are biased towards zero.
- This means $\mathbb{E}[\mathbf{s}^{[t]}] \neq \mathbb{E}[\hat{\mathbf{g}}^{[t]}]$ and $\mathbb{E}[\mathbf{r}^{[t]}] \neq \mathbb{E}[\hat{\mathbf{g}}^{[t]} \odot \hat{\mathbf{g}}^{[t]}]$ (where the expectations are calculated over minibatches).
- To see this, let us unroll the computation of $\mathbf{s}^{[t]}$ for a few time-steps:

$$\mathbf{s}^{[0]} = \mathbf{0}$$

$$\mathbf{s}^{[1]} = \rho_1 \mathbf{s}^{[0]} + (1 - \rho_1) \hat{\mathbf{g}}^{[1]} = (1 - \rho_1) \hat{\mathbf{g}}^{[1]}$$

$$\mathbf{s}^{[2]} = \rho_1 \mathbf{s}^{[1]} + (1 - \rho_1) \hat{\mathbf{g}}^{[2]} = \rho_1 (1 - \rho_1) \hat{\mathbf{g}}^{[1]} + (1 - \rho_1) \hat{\mathbf{g}}^{[2]}$$

$$\mathbf{s}^{[3]} = \rho_1 \mathbf{s}^{[2]} + (1 - \rho_1) \hat{\mathbf{g}}^{[3]} = \rho_1^2 (1 - \rho_1) \hat{\mathbf{g}}^{[1]} + \rho_1 (1 - \rho_1) \hat{\mathbf{g}}^{[2]} + (1 - \rho_1) \hat{\mathbf{g}}^{[3]}$$

- Therefore, $\mathbf{s}^{[t]} = (1 - \rho_1) \sum_{i=1}^t \rho_1^{t-i} \hat{\mathbf{g}}^{[i]}$.
- Note that the contribution of the earlier $\hat{\mathbf{g}}^{[i]}$ to the moving average shrinks rapidly.

ADAM

- The expected value of $\mathbf{s}^{[t]}$ is:

$$\begin{aligned}\mathbb{E}[\mathbf{s}^{[t]}] &= \mathbb{E}[(1 - \rho_1) \sum_{i=1}^t \rho_1^{t-i} \hat{\mathbf{g}}^{[i]}] \\ &= \mathbb{E}[\hat{\mathbf{g}}^{[t]}](1 - \rho_1) \sum_{i=1}^t \rho_1^{t-i} + \zeta \\ &= \mathbb{E}[\hat{\mathbf{g}}^{[t]}](1 - \rho_1^t) + \zeta\end{aligned}$$

where we approximate $\hat{\mathbf{g}}^{[i]}$ with $\hat{\mathbf{g}}^{[t]}$ which allows us to move it outside the sum. ζ is the error that results from this approximation.

- Therefore, $\mathbf{s}^{[t]}$ is a biased estimator of $\hat{\mathbf{g}}^{[t]}$ and the effect of the bias vanishes over the time-steps (because $\rho_1^t \rightarrow 0$ for $t \rightarrow \infty$).
- Ignoring ζ (as it can be kept small), we correct for the bias by setting $\hat{\mathbf{s}}^{[t]} = \frac{\mathbf{s}^{[t]}}{(1 - \rho_1^t)}$.
- Similarly, we set $\hat{\mathbf{r}}^{[t]} = \frac{\mathbf{r}^{[t]}}{(1 - \rho_2^t)}$.

COMPARISON OF OPTIMIZERS: ANIMATION

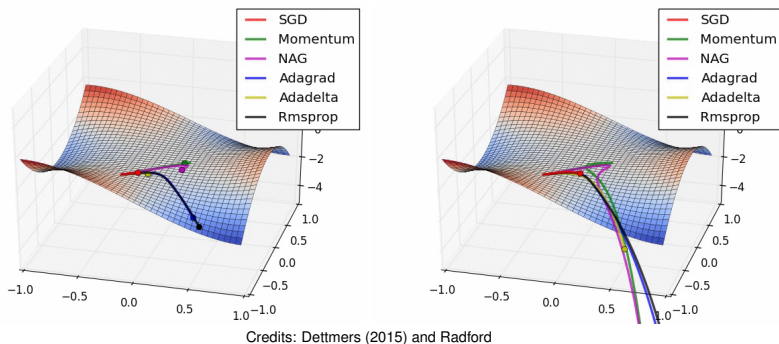


Figure: Excerpts from an animation to compare the behavior of momentum and other methods compared to SGD for a saddle point. Left: After a few seconds; Right: A bit later. The animation shows that all showed methods accelerate optimization compared to the standard SGD. The highest acceleration is obtained using Rmsprop followed by Adagrad as learning rate strategies. You can find the animation [here](#) or click on the images above.

Batch Normalization

BATCH NORMALIZATION

- Batch Normalization (BatchNorm) is an extremely popular technique that improves the training speed and stability of deep neural nets.
- It is an extra component that can be placed between each layer of the neural network.
- It works by changing the "distribution" of activations at each hidden layer of the network.
- We know that it is sometimes beneficial to normalize the inputs to a learning algorithm by shifting and scaling all the features so that they have 0 mean and unit variance.
- BatchNorm applies a similar transformation to the activations of the hidden layers (with a couple of additional tricks).

BATCH NORMALIZATION

- For a hidden layer with neurons $z_j, j = 1, \dots, J$, BatchNorm is applied to each z_j by considering the activations of z_j **over a given minibatch** of inputs.
- Let $z_j^{(i)}$ denote the activation of z_j for input $x^{(i)}$ in the minibatch (of size m).
- The mean and variance of the activations are

$$\mu_j = \frac{1}{m} \sum_i^m z_j^{(i)}$$
$$\sigma_j^2 = \frac{1}{m} \sum_i^m (z_j^{(i)} - \mu_j)^2$$

- Each $z_j^{(i)}$ is then normalized

$$\tilde{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where a small constant, ϵ , is added for numerical stability.

BATCH NORMALIZATION

- It may not be desirable to normalize the activations in such a rigid way because potentially useful information can be lost in the process.
- Therefore, we commonly let the training algorithm decide the "right amount" of normalization by allowing it to re-shift and re-scale $\tilde{z}_j^{(i)}$ to arrive at the batch normalized activation $\hat{z}_j^{(i)}$:

$$\hat{z}_j^{(i)} = \gamma_j \tilde{z}_j^{(i)} + \beta_j$$

- γ_j and β_j are learnable parameters that are also tweaked by backpropagation.
- $\hat{z}_j^{(i)}$ then becomes the input to the next layer.
- Note: The algorithm is free to scale and shift each $\tilde{z}_j^{(i)}$ back to its original (unnormalized) value.

BATCH NORMALIZATION: ILLUSTRATION

- Recall: $z_j = \sigma(W_j^T x + b_j)$
- So far, we have applied batch-norm to the activation z_j . It is possible (and more common) to apply batch norm to $W_j^T x + b_j$ *before* passing it to the nonlinear activation σ .

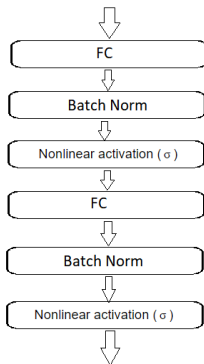
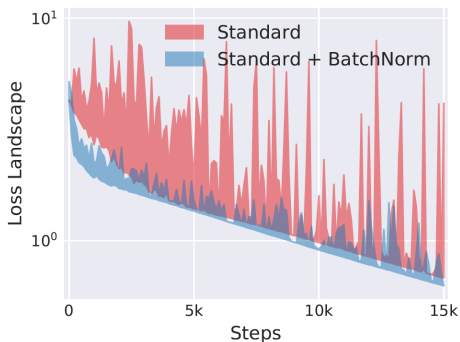


Figure: FC = Fully Connected layer. BatchNorm is applied *before* the nonlinear activation function.

BATCH NORMALIZATION

- The key impact of BatchNorm on the training process is this: It reparametrizes the underlying optimization problem to **make its landscape significantly more smooth**.
- One aspect of this is that the loss changes at a smaller rate and the magnitudes of the gradients are also smaller (see Santurkar et al. 2018).



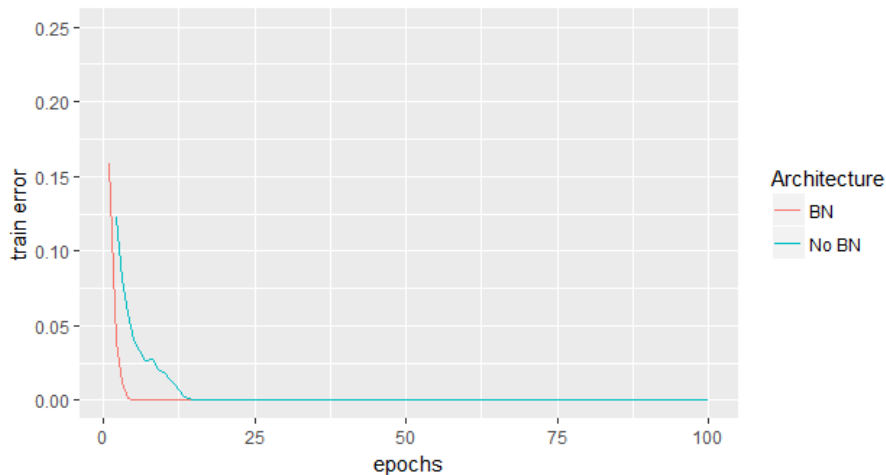
BATCH NORMALIZATION: PREDICTION

- Once the network has been trained, how can we generate a prediction for a single input (either at test time or in production)?
- One option is to feed the entire training set to the (trained) network and compute the means and standard deviations.
- More commonly, during training, an exponentially weighted running average of each of these statistics over the minibatches is maintained.
- The learned γ and β parameters are then used (in conjunction with the running averages) to generate the output.

BATCH NORMALIZATION

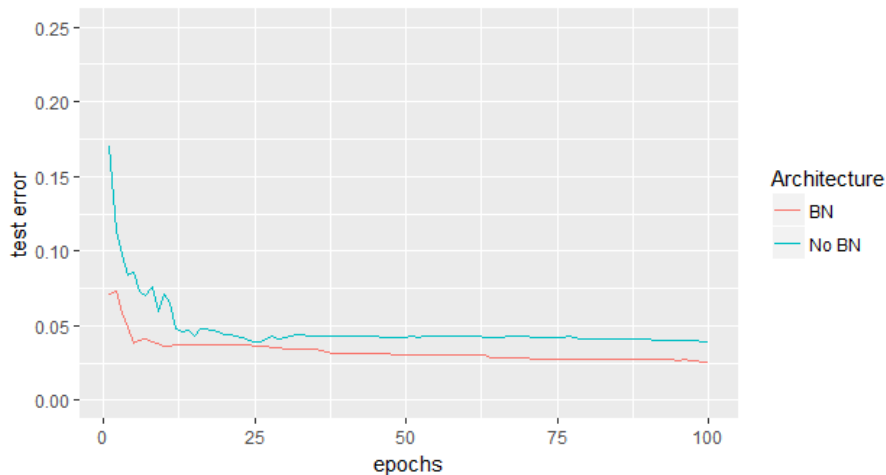
- For our final benchmark in this chapter we compute two models to predict the mnist data.
- One will extend our basic architecture such that we add batch normalization to all hidden layers.
- We use SGD as optimizer with a momentum of 0.9, a learning rate of 0.03 and weight decay of 0.001.

BATCH NORMALIZATION



Batch Normalization accelerated learning.

BATCH NORMALIZATION



Batch Normalization resulted in a lower test error.

REFERENCES



Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016)

Deep Learning

<http://www.deeplearningbook.org/>



Yann Dauphin, Razvan Pascanu, Çağlar Gülçehre, Kyunghyun Cho, Surya Ganguli, Yoshua Bengio (2014)

Identifying and attacking the saddle point problem in high-dimensional non-convex optimization

<https://arxiv.org/abs/1406.2572>



Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein (2017)

Visualizing the Loss Landscape of Neural Nets

<https://arxiv.org/abs/1712.09913>



Tim Dettmers (2015)

Deep Learning in a Nutshell: History and Training

<https://devblogs.nvidia.com/deep-learning-nutshell-history-training/>

REFERENCES



Hafidz Zulkifli (2018)

Understanding Learning Rates and How It Improves Performance in Deep Learning

<https://towardsdatascience.com>



Ilya Loshchilov, Frank Hutter (2016)

SGDR: Stochastic Gradient Descent with Warm Restarts

<https://arxiv.org/abs/1608.03983>



Jeremy Jordan (2018)

Setting the learning rate of your neural network

<https://www.jeremyjordan.me/nn-learning-rate/>




Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry (2018)

How Does Batch Normalization Help Optimization?

<https://arxiv.org/abs/1805.11604>

REFERENCES

-  Akshay Chandra (2015)
Learning Parameters, Part 2: Momentum-Based & Nesterov Accelerated Gradient Descent
*[https://towardsdatascience.com/
learning-parameters-part-2-a190bef2d12](https://towardsdatascience.com/learning-parameters-part-2-a190bef2d12)*