The Art & Science of Training Deep MLPs

M. R. Hasan

Deep Learning
Readings

• Geron: 11
The Art & Science of Training Deep MLPs

• We will discuss some key issues regarding the training of MLPs.

• Some issues concern the deep MLPs (many hidden layers and neurons) or more generally deep neural networks (DNNs).
### Issues Regarding Training Deep MLPs

- We will discuss the following issues.
  - Feature scaling
  - Intractability of computation
  - Computational complexity
  - Overfitting due to complex architecture
  - Overfitting due to overtraining
  - Weight initialization
  - Vanishing gradient & exploding gradient problem
  - Achilles’ Heel of Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues: Achilles’ Heel of SGD
Achilles’ Heel of SGD

• Cure for Achilles’ heel
  • Augment SGD to create **fast optimizers**.
  • We can overcome the limitations of the vanilla SGD as well as the topological factors that are due to the cost function by using fast optimizers.
  • These optimizers are created by augmenting SGD.
Achilles’ Heel of SGD

• We will discuss following fast optimizers.
  - Momentum optimization
  - Nesterov Accelerated Gradient
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

First two techniques use **scalar constant** learning rate.

The remaining optimizers adapt learning rate per dimension.
Momentum Optimization

- The Momentum optimization is a **small hack of SGD**.
- It **reduces** SGD’s oscillations and accelerates its movement.
Momentum Optimization

• To understand SGD’s oscillation issue, consider the following **bowl-shaped** 2D cost function.
• The slope varies gently along the direction of $w_1$, while it **varies heavily along the direction of $w_2$**.
• SGD keeps on bouncing back and forth with high amplitude as it moves from left to right.
Momentum Optimization

• An intuitive way to **dampen (smooth out) the oscillations** is by computing a **moving/local average** of the loss gradient.

The technique to compute this is known as **Exponentially Weighted Moving Average (EWMA)**.
Momentum Optimization

- To impose “smoothing”, the EWMA puts more weight on the past observations.
- Past observations build up “momentum” that stabilizes the fluctuations of the current observation.

\[
\text{Moving\_Average}(t) = \text{weight} \times \text{Past\_Observations} + (1 - \text{weight}) \times \text{Observation}(t)
\]

Past observations are weighted heavily by the weight parameter (denoted by β).

It determines how far we go into the past to determine the present.
Momentum Optimization

- How does the momentum optimization work?
- SGD update rule *does not care* about what the earlier gradients were.
- If the **local gradient** is tiny, SGD goes very slowly.
- The momentum optimization not only uses the gradient of the current step $t$ to guide the search, but also it accumulates the gradient of the past steps (0 to $t-1$) to determine the direction to go.

\[
W_{t+1}^{(k)} = W_t^{(k)} - \eta \nabla L(W_t^{(k)})
\]
Momentum Optimization

In momentum optimization, for a layer k’s weight matrix $W^{(k)}$, we keep an additional matrix of the same dimension $m$ (it is called the momentum).

Here $m$ is used to keep a moving average of the past gradients.

We add the weighted past gradients $(t - 1)$ with the current gradient $(t)$.

Vanilla SGD

$$W_{t+1}^{(k)} = W_t^{(k)} - \eta \nabla \mathcal{L} \left( W_t^{(k)} \right)$$

Momentum Optimization

$$m_t^{(k)} = \beta m_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L} \left( W_t^{(k)} \right)$$

$$W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)}$$

Moving-Average$(t) = \text{weight} \times \text{Past-Observations} + (1 - \text{weight}) \times \text{Observation}(t)$
Momentum Optimization

- At the beginning, $m$ is initialized with zeros.
- Then, it keeps a moving average of the past gradients.
- We update $W^{(k)}$ using the moving average of the past gradients.
- Here the momentum $m_t^{(k)}$ is the retained gradient from past iterations.

$$m_t^{(k)} = \beta m_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L} \left( W_t^{(k)} \right)$$

$$W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)}$$

Vanilla SGD

Momentum Optimization
Momentum Optimization

• We don’t want the momentum to grow unboundedly as it accumulates past gradients.
• This is prevented by introducing “friction” via a new hyperparameter $\beta$.
• It must be set between 0 (high friction) and 1 (no friction).
• A typical value of $\beta$ is 0.9.

\[
m_t^{(k)} = \beta m_{t-1}^{(k)} + (1 - \beta)\nabla \mathcal{L}(W_t^{(k)})
\]

\[
W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)}
\]
Momentum Optimization

- When $\beta = 0$, we recover the vanilla SGD.
- But for $\beta = 0.9$ (or larger, much closer to 1), past gradients are weighted more heavily.

\[
m_t^{(k)} = \beta m_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L}(W_t^{(k)})
\]

\[
W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)}
\]

This technique creates an annealing effect (reduction of learning rate) by multiplying $\eta$ with the changing momentum (gradient) term.
Momentum Optimization

- In practice, a less intuitive version of the momentum update rule works better.
- Here $\eta$ is re-estimated for dropping the $(1 - \beta)$ term before the loss gradient.

\[
\begin{align*}
\mathbf{m}_t^{(k)} &= \beta \mathbf{m}_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L}(\mathbf{W}_t^{(k)}) \\
\mathbf{W}_{t+1}^{(k)} &= \mathbf{W}_t^{(k)} - \eta \mathbf{m}_t^{(k)}
\end{align*}
\]
Momentum Optimization

• To understand why the momentum optimization is effective, first understand why oscillation occurs in SGD.
• Consider the following bowl-shaped 2D loss function.
• The slope varies gently along the direction of $w_1$, while it varies heavily along the direction of $w_2$.
• SGD keeps on bouncing back and forth with high amplitude as it moves from left to right.
Momentum Optimization

- The momentum optimization **averages out the gradients** for both $w_1$ and $w_2$.
- The oscillations in the **steeper direction** will tend to average out to something closer to zero.
- Thus, in the steeper direction, where we want to slow things down, this will average out positive and negative numbers.

The average in the flatter direction will be pretty big.
Momentum Optimization

• If we average out the gradients for both $w_1$ and $w_2$, we find that the oscillations in the steeper direction will tend to average out to something closer to zero.

• Thus, in the steeper direction, where we want to slow things down, this will average out positive and negative numbers.

\[
\begin{align*}
m_{w_1t} &= \beta m_{w_1 t-1} + (1 - \beta) \nabla w_1 J(w_1) \\
m_{w_2t} &= \beta m_{w_2 t-1} + (1 - \beta) \nabla w_2 J(w_2)
\end{align*}
\]

The average in the flatter direction will be pretty big.
Achilles’ Heel of SGD

• We will discuss following fast optimizers.
  - Momentum optimization
  - Nesterov Accelerated Gradient
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

First two techniques use **scalar constant** learning rate.

The remaining optimizers adapt learning rate **per dimension**.
Nesterov Accelerated Gradient (NAG)

• Momentum optimization dampens SGD oscillations, but *doesn’t completely eliminate* it.
• Can we augment Momentum optimization to further dampen oscillations?
• NAG employs a simple idea.
• Previously we chose the *direction of movement* based on the current location.
• NAG looks at the momentum vector $m_{t-1}$ at time $(t - 1)$ to see *where it leads to* at the next time step at $t$. 
• NAG: instead of computing the gradient at $w_t$, it computes it at $w_t + \beta m_{t-1}$

- Maroon arrow: represents the regular Momentum update
- Purple + red arrow: represents the NAG update
- Black arrow: represents the shortest path toward the minimum
• NAG first makes a **big jump in the direction of the previous accumulated gradient** (purple vector).

• Then, it measures the gradient and moves towards the direction of gradient (red arrow).

• It results in the complete NAG update (black arrow).

• This **anticipatory update** dampens the oscillation and expedites the movement.
Nesterov Accelerated Gradient (NAG)

• NAG update rule:

1. \( m_t = \beta m_{t-1} + (1 - \beta) \nabla_w J(w_t + \beta m_{t-1}) \)

2. \( w_{t+1} = w_t - \eta m_t \)
Beyond Momentum Optimization & NAG

• Momentum optimization and NAG use a **scalar constant** learning rate for all dimension.
• They augment SGD by adapting the weight vector using past gradients.
• However, some problems require **adapting learning rates per dimension**.
• The remaining optimization algorithms are designed to do this.
Beyond Momentum Optimization & NAG

- The per-dimension fast optimizers draw their inspiration from second-order SGD, in which the update is based on the Newton method:

\[ \mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{H}^{-1} \mathbf{g} \]

A computationally feasible approximation of the Newton method uses the diagonal approximation of the Hessian, instead of the full Hessian:

\[ \mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\text{diag}(\mathbf{H})} \mathbf{g} \]
Beyond Momentum Optimization & NAG

• A useful insight from Newton’s algorithm is:
• Find any **preconditioner matrix** $B$ that improves the performance of SGD, but **without involving expensive computation** ($\approx O(d)$) of the preconditioner:

$$w_{t+1} = w_t - \frac{\eta}{B} g$$

$$w_{t+1} = w_t - \frac{\eta}{\text{diag}(H)} g$$

Here, the diagonal matrix $B$ is used to **scale the learning rate** $\eta$ per dimension.

Fast optimizers compute the preconditioner matrix $B$ in such a fashion that it provides **some approximation of the diagonal approximation of Hessian**.
Per Dimension LR Adaptation

- Adapt learning rate per dimension by using a *Hessian-like* preconditioner matrix $B$
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

$$w_{t+1} = w_t - \frac{\eta}{B} g$$

$$w_{t+1} = w_t - \frac{\eta}{\text{diag}(H)} g$$
Adaptive Gradient (AdaGrad)

• For scaling $\eta$ per dimension, AdaGrad approximates the second-order preconditioner matrix $B$ by taking the outer product of the d-dimensional gradient vector $\nabla J(w)$ with itself and creates a $d \times d$ diagonal matrix.

$$B \equiv \text{diag}(\nabla J(w) \otimes \nabla J(w))$$

Using this diagonal preconditioner matrix $B$, the AdaGrad update rule is:

1. $s_t = s_{t-1} + B_t$

2. $w_{t+1} = w_t - \frac{\eta}{B} g$

$$w_{t+1} = w_t - (\eta \otimes \sqrt{s_t + \epsilon}) \nabla w J(w_t)$$
Adaptive Gradient (AdaGrad)

• Observe that this algorithm **relies only on first-order information**, but has some properties of second-order methods via $s$.

• It also has the **annealing effect** as it accumulates the past 2\(^{nd}\)-order terms.

• Thus, the learning rate decays as iteration increases.

• However, the decay is **not same for all weights**.

\begin{align*}
1. \quad & s_t = s_{t-1} + B_t \\
2. \quad & w_{t+1} = w_t - (\eta \odot \sqrt{s_t + \epsilon}) \nabla_w J(w_t)
\end{align*}

• The decay happens faster for steeper dimensions
• The decay happens slowly for flatter dimensions
Adaptive Gradient (AdaGrad)

- Limitations of AdaGrad:
  - We need to **manually select** a global learning rate (default is 0.01).
  - Learning rate gets **scaled down so much** that the algorithm stops entirely before reaching the global optimum (caused by the accumulation of the squared gradients).

\[
B \equiv diag(\nabla J(w) \otimes \nabla J(w))
\]

1. \( s_t = s_{t-1} + B_t \)
2. \( w_{t+1} = w_t - (\eta \otimes \sqrt{s_t + \epsilon}) \nabla_w J(w_t) \)
Per Dimension LR Adaptation

- Adapt learning rate **per dimension** by using a *Hessian-like* preconditioner matrix $B$
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

\[
  w_{t+1} = w_t - \frac{\eta}{B} g
\]

\[
  w_{t+1} = w_t - \frac{\eta}{\text{diag}(H)} g
\]
AdaDelta

- Instead of accumulating all past squared gradients, Adadelta defines the **sum of gradients recursively** as an exponentially decaying average of all past squared gradients.
- The moving average at the current time step $t$ depends (as a fraction $\gamma$ similarly to the Momentum term) only on the previous average and the current gradient:

\[
E[\nabla_w J(w)^2]_t = \gamma E[\nabla_w J(w)^2]_{t-1} + (1 - \gamma) \nabla_w J(w_t)^2
\]

\[
\begin{align*}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta}{\sqrt{E[\nabla_w J(w)^2]_t + \epsilon}} \nabla_w J(w_t)
\end{align*}
\]
AdaDelta

• AdaDelta removes the global learning rate parameter.
• It does so by a unit matching technique.
• Observe that the unit of the last term in the above update equation is not \( w \): the two gradient terms cancels out remaining a unitless \( \eta \).
• To fix this mismatch, look at the Newton equation below.
• The unit of the term \( H^{-1}g \) matches with the unit of \( w_i \).

\[
\begin{align*}
    w_{t+1} &= w_t - \frac{\eta}{\sqrt{E[\nabla_w J(w)^2]_t + \epsilon}} \nabla_w J(w_t) \\
    w_{t+1} &= w_t - H^{-1}g \\
    H^{-1}g &\propto \frac{\partial J(w)}{\partial w_i} \propto \text{(units of } w_i) \end{align*}
\]
AdaDelta

- Adadelta resolves the unit mismatch issue by substituting the global scalar constant learning rate $\eta$ by an expression that has the unit of $w$.

- Adadelta defines this expression by using an exponentially decaying average of squared weight updates:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[\nabla_w J(w)^2]_t + \epsilon}} \nabla_w J(w_t)$$

Root mean squared error of the weight parameter updates:

$$RMS[w]_t = \sqrt{E[w^2]_t + \epsilon}$$
AdaDelta

- AdaDelta replaces the learning rate $\eta$ with $\text{RMS}[w]_{t-1}$

$$w_{t+1} = w_t - \eta \frac{\nabla w J(w_t)}{\sqrt{E[\nabla w J(w)^2]}_t + \epsilon}$$

$$w_{t+1} = w_t - \frac{\text{RMS}[w]_{t-1}}{\text{RMS}[J(w)]_t} \nabla w J(w_t)$$

By resolving the unit mismatch issue, Adadelta removes the scalar global learning learning rate.

It uses a pseudo-curvature (diagonal curvature approximation) per dimension to mimic a Newton-like update.
AdaDelta

- One limitation of Adadelta is that it does not have an explicit annealing schedule, which might delay convergence near a minimum.
- For this reason Momentum optimization, that does annealing, is comparatively faster than Adadelta.
- With momentum, oscillations that can occur near a minima are smoothed out, whereas with Adadelta these can accumulate in the numerator.
Per Dimension LR Adaptation

- Adapt learning rate **per dimension** by using a *Hessian-like* preconditioner matrix B
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam
Both RMSProp and Adadelta were developed independently around the same time stemming from the need to resolve AdaGrad’s radically diminishing learning rates.

RMSProp, in fact, is identical to the first update vector of Adadelta that we derived previously.

Following is the RMSProp update rule.

1. \( s_t = \beta s_{t-1} + (1 - \beta) \nabla_w J(w_t) \otimes \nabla_w J(w_t) \)
2. \( w_{t+1} = w_t - (\eta \otimes \sqrt{s_t + \epsilon}) \nabla_w J(w_t) \)

Newton Algorithm (diagonal approximation):

\[
\begin{align*}
    w_{t+1} &= w_t - \frac{\eta}{\text{diag}(H)} g
\end{align*}
\]
Per Dimension LR Adaptation

• Adapt learning rate per dimension by using a Hessian-like preconditioner matrix $B$
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

\[
\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{B} \mathbf{g}
\]

\[
\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\text{diag}(H)} \mathbf{g}
\]
Adaptive Moment Estimation (ADAM)

- Adam **combines** the ideas of Momentum optimization and RMSProp.
- Just like Momentum optimization, it keeps track of an exponentially decaying average of *past gradients*.
- And just like RMSProp and Adadelta, it keeps track of an exponentially decaying average of *past squared gradients*.

### Momentum Optimization

1. \( m_t = \beta m_{t-1} + (1 - \beta) \nabla_w J(w_t) \)
2. \( w_{t+1} = w_t - \eta m_t \)

### RMSProp

1. \( s_t = \beta s_{t-1} + (1 - \beta) \nabla_w J(w_t) \otimes \nabla_w J(w_t) \)
2. \( w_{t+1} = w_t - \eta \sqrt{s_t + \epsilon} \nabla_w J(w_t) \)

\[ w_{t+1} = w_t - \frac{\eta}{B} g \]
Adaptive Moment Estimation (ADAM)

- ADAM update rule.

\[
\begin{align*}
1. & \quad m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w J(w_t) \\
2. & \quad s_t = \beta_2 s_{t-1} + (1 - \beta_2) \nabla_w J(w_t) \\
3. & \quad \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \\
4. & \quad \hat{s}_t = \frac{s_t}{1 - \beta_2^t} \\
5. & \quad w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{s}_t + \epsilon}} \hat{m}_t
\end{align*}
\]

Exp. Decaying avg of \textbf{gradient}

Exp. Decaying avg of \textbf{squares of gradient}

Bias correction: Since \( m \) and \( s \) are initialized at 0, they will be biased toward 0 at the beginning of training (\( \beta \approx 0.9 \)), so these two steps will help boost \( m \) and \( s \) at the beginning of training.

Multiply the learning rate by average of the gradient (Eq. 3) and divide it by the root mean square of the exponential average of square of gradients (Eq. 4).
Adaptive Moment Estimation (ADAM)

• Since Adam is an adaptive learning rate algorithm, it requires less tuning of the learning rate hyperparameter $\eta$.
• We often use the default value $\eta = 0.001$, which makes Adam even easier to use than SGD.
• The momentum decay hyperparameter $\beta_1$ is typically initialized to 0.9.
• The scaling decay hyperparameter $\beta_2$ is often initialized to 0.999.

\[
\begin{align*}
1. & \quad m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w J(w)_t \\
2. & \quad s_t = \beta_2 s_{t-1} + (1 - \beta_2) \nabla_w J(w)_t \otimes \nabla_w J(w)_t \\
3. & \quad \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \\
4. & \quad \hat{s}_t = \frac{s_t}{1 - \beta_2^t} \\
5. & \quad w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{s}_t + \epsilon}} \hat{m}_t
\end{align*}
\]
Adaptive Moment Estimation (ADAM)

• The *smoothing term* $\varepsilon$ is usually initialized to a tiny number such as $10^{-7}$.
• By setting $\beta_1$ to zero, we can use Adam without Momentum optimization.
• In some problems, it is useful to *tweak this Momentum hyperparameter*.

1. $m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w J(w)_t$
2. $s_t = \beta_2 s_{t-1} + (1 - \beta_2) \nabla_w J(w)_t \circ \nabla_w J(w)_t$
3. $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$
4. $\hat{s}_t = \frac{s_t}{1 - \beta_2^t}$
5. $w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{s}_t} + \varepsilon} \hat{m}_t$
Per Dimension LR Adaptation

- Adapt learning rate per dimension by using a Hessian-like preconditioner matrix $B$
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam
Nesterov-accelerated Adaptive Moment Estimation (NADAM)

- **Combines** Adam with NAG.
- Thus, it often converges *slightly faster* than Adam.
- In order to incorporate NAG into Adam, we need to modify its momentum term.

**ADAM**

1. \( m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w J(w_t) \)
2. \( s_t = \beta_2 s_{t-1} + (1 - \beta_2) \nabla_w J(w_t) \otimes \nabla_w J(w_t) \)
3. \( \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \)
4. \( \hat{s}_t = \frac{s_t}{1 - \beta_2^t} \)
5. \( w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \hat{m}_t \)

**NAG**

1. \( m_t = \beta m_{t-1} + (1 - \beta) \nabla_w J(w_t + \beta m_{t-1}) \)
2. \( w_{t+1} = w_t - \eta m_t \)
Nesterov-accelerated Adaptive Moment Estimation (NADAM)

- Unlike NAG, Nadam applies the look-ahead momentum vector directly to update the current weight parameters:

1. $m_t = \beta m_{t-1} + (1 - \beta) \nabla_w J(w_t)$
2. $w_{t+1} = w_t - \eta (\beta m_t + (1 - \beta) \nabla_w J(w_t))$

NAG

1. $m_t = \beta m_{t-1} + (1 - \beta) \nabla_w J(w_t + \beta m_{t-1})$
2. $w_{t+1} = w_t - \eta m_t$
Nesterov-accelerated Adaptive Moment Estimation (NADAM)

• Consider the Adam update rule.
• We excluded the s terms as we don’t need to modify it.

\[
\begin{align*}
    \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
    w_{t+1} &= w_t - \frac{\eta}{\sqrt{s_t} + \epsilon} \hat{m}_t \\
    m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w J(w)_t
\end{align*}
\]
Nesterov-accelerated Adaptive Moment Estimation (NADAM)

- The update rule can be rewritten as follows:

\[
\begin{align*}
    w_{t+1} &= w_t - \frac{\eta}{\sqrt{s_t} + \epsilon} \frac{m_t}{1 - \beta_1^t} \\
    &= w_t - \frac{\eta}{\sqrt{s_t} + \epsilon} \left( \frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) \nabla_w J(w_t)}{1 - \beta_1^t} \right)
\end{align*}
\]

\[
\begin{align*}
    m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w J(w_t) \\
    \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
    w_{t+1} &= w_t - \frac{\eta}{\sqrt{s_t} + \epsilon} \hat{m}_t
\end{align*}
\]
Nesterov-accelerated Adaptive Moment Estimation (NADAM)

- We see that \( \frac{\beta_1 m_{t-1}}{1 - \beta_1^t} \) is just the bias-corrected estimate of the momentum vector of the previous timestep.
- We thus replace it with \( \hat{m}_{t-1} \)

\[
\begin{align*}
  w_{t+1} &= w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \frac{m_t}{1 - \beta_1^t} \\
  &= w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \left( \frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) \nabla_w J(w_t)}{1 - \beta_1^t} \right)
\end{align*}
\]

\[
\begin{align*}
  w_{t+1} &= w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \left( \beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) \nabla_w J(w_t)}{1 - \beta_1^t} \right)
\end{align*}
\]
NADAM

• We now add Nesterov momentum just as we did previously by simply replacing this bias-corrected estimate of the momentum vector $\hat{m}_{t-1}$ of the previous timestep with the bias-corrected estimate of the current momentum vector $\hat{m}_t$, which gives us the Nadam update rule:

\[
\begin{align*}
1. & \quad m_t = \beta m_{t-1} + (1 - \beta) \nabla_w J(w_t) \\
2. & \quad w_{t+1} = w_t - \eta (\beta m_t + (1 - \beta) \nabla_w J(w_t))
\end{align*}
\]

\[
\begin{align*}
\hat{m}_t &= \beta \hat{m}_{t-1} + (1 - \beta) \nabla_w J(w_t) \\
\hat{m}_t &= \beta \hat{m}_{t-1} + (1 - \beta) \nabla_w J(w_t)
\end{align*}
\]
Achilles’ Heel of SGD

• We have discussed following fast optimizers.
  - Momentum optimization
  - Nesterov Accelerated Gradient
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

First two techniques use scalar constant learning rate.

The remaining optimizers adapt learning rate per dimension.
Achilles’ Heel of SGD

• The **per-dimension adaptive optimization** techniques (Adagrad, Adadelta, RMSProp, Adam, Nadam) use some sort **pseudo-hessians** to adapt learning rate per dimension.

• But they are **not based on the actual Hessian**.

• These optimizers don’t have access to the **true curvature** of the cost space.

• Thus, there is **no guarantee** that these optimizers will always converge fast to a good solution.

• It depends on the problem.

Newton Algorithm (diagonal approximation):

\[ w_{t+1} = w_t - \frac{\eta}{\text{diag}(H)}g \]
Achilles’ Heel of SGD

• It has been shown that these adaptive optimizers can lead to solutions that generalize poorly on some datasets.
  

• Thus, we should *first try* SGD with momentum or NAG.

• Often times momentum optimization will provide faster convergence to a good solution when coupled with a good learning schedule.
Achilles’ Heel of SGD

• Summary:

• Try momentum optimization or NAG with a good learning schedule.

• The 1cycle learning schedule provides faster convergence and allows to use large learning rate.

• See the following notebook for a discussion on various learning schedules.