Notes on Adam Optimizer

Ashwin Mathur Varun Mathur
{ashwinxmathur, varunm500}@gmail.com

March 3, 2025

1 Optimizer Progression and Inclusion Principles

As a hyperparameter tuning protocol approaches optimality, a more expressive optimizer can never underperform any of its specializations. This can be shown by the inclusion relations between the optimizers as follows:

- SGD $\subseteq$ MOMENTUM $\subseteq$ RMSPROP
- SGD $\subseteq$ MOMENTUM $\subseteq$ ADAM
- SGD $\subseteq$ NESTEROV $\subseteq$ ADAM

1.1 Optimizer Hyperparameters

- **Learning Rate (η):**

  The Learning Rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging since a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

- **Momentum (γ):**

  The Momentum hyperparameter allows the gradient search to build inertia in a direction in the search space, and overcome the oscillations of noisy gradients and converge to the local maxima/minima faster. Momentum has the effect of dampening down the change in the gradient and the step size with each new point in the search space.

- **Smoothing Constant (β):**

  The Smoothing Constant hyperparameter maintains a decaying average of squared gradients. Using a decaying moving average of the partial derivative allows the search to forget early partial derivative values and focus on the most recently seen shape of the search space. This helps the optimizer to discard history from the extreme past so that it can converge rapidly after finding a convex surface.

- **Exponential Decay Rate - First Order and Second Order Moments (β_1 and $\beta 2$):** The exponential decay rate for the first order moment estimates, and the second order moment estimates, are used for smoothing the path to convergence, and for also providing some momentum to cross a local minima or saddle point. These are similar to the smoothing constant in the other optimizers, but are used to decay the gradients instead of the learning rate.

2 Stochastic Gradient Descent (SGD)

New weights are the old weights minus the learning rate into the gradient of the change in the weights.

$$w_{t+1} = w_t - \eta \nabla w_t$$

$$\text{where,} \quad \nabla w_t = \left.\frac{\partial \mathcal{L}(w)}{\partial w}\right|_{w=w_t} \quad and \quad \eta \text{ is the Learning Rate.}$$

- Steep Curve $\rightarrow$ Large Gradient $\rightarrow$ Larger Update in weights
- Gentle Slope $\rightarrow$ Small Gradient $\rightarrow$ Small Update in weights
- Gets stuck in regions where the slope is gentle ie. gradients are small. Suffers from vanishing gradient problems.

3 Stochastic Gradient Descent with Momentum (SGD-Momentum)

The key idea here is to use history of updates as momentum. Make larger updates in the same direction, to increase the speed of convergence.

$$\begin{aligned} \text{update}_t &= \gamma \cdot \text{update}_{t-1} + \eta \nabla w_t \\ w_{t+1} &= w_t - \text{update}_t \end{aligned}$$

where, γ is the Momentum Parameter *and* η is the Learning Rate.

- The use of momentum helps increase the speed of convergence.
- It helps in areas of gentle slope. Helps solve the vanishing gradients to some extent.
- Due to the extra momentum, the optimizer can oscillate about the minima. Still, convergence is faster than SGD most of the times.
- **Inclusion:** When $\gamma = 0$, SDG with momentum converges to vanilla SGD. So, SGD $\subseteq$ MOMENTUM.

4 Nesterov Accelerated Gradient Descent (NAG)

NAG hinges on the principle *"Look before you leap"*. We look ahead by calculating the gradient of partially updated weight values, instead of the gradient based on the full gradient of the current weight values.

$$\begin{aligned} w_{\text{look ahead}} &= w_t - \gamma \cdot \text{update}_{t-1} \\ \text{update}_t &= \gamma \cdot \text{update}_{t-1} + \eta \nabla w_{\text{look ahead}} \\ w_{t+1} &= w_t - \text{update}_t \end{aligned}$$

where, γ is the Momentum Parameter *and* η is the Learning Rate.

Note here: We use $\nabla\ w_{\text{look ahead}}$ instead of $\nabla\ w_t$ as compared to SGD-Momentum.

- The look ahead, helps overcome the oscillation problem seen in SDG with momentum.
- **Inclusion:** When $\gamma = 0$, NAG converges to vanilla SGD.

$$w_{\text{look ahead}} = w_t \tag{1}$$

$$\text{update}_t = \eta \nabla w_{\text{look ahead}} \tag{2}$$

Based on the above equations we get:

$$\text{update}_t = \eta \nabla w_t \tag{3}$$

$$w_{t+1} = w_t - \eta \nabla w_t \tag{4}$$

5 Adaptive Learning Rates:

For faster convergence we want to use a high learning rate. The problem with that is, if the learning rate is too high, when we are close to the point of minima, the updates are too large and we oscillate.

The solution is to use an adaptive learning rate that reduces as we make updates to the weights. This can be done using a Learning Rate Scheduler or using a optimizer that uses Learning Rate decay.

Adaptive Learning Rate based optimizers:

- AdaGrad
- RMSProp
- Adam (and other versions of Adam like NAdam, and AdamW)

5.1 Learning Rate Scheduler

- The learning rate governs the magnitude of the step taken by the optimizer in its pursuit of minimizing the loss function.
- A learning rate schedule is a predefined framework that adjusts the learning rate between epochs or iterations as the training process progresses.
- During the initial stages of training, the learning rate is set to a higher value to facilitate the attainment of a set of weights that are reasonably accurate. As training continues, these weights are fine-tuned through the application of a smaller learning rate, thereby achieving higher accuracy.
- Learning rate schedules aim to adjust the learning rate during the training process by reducing the learning rate according to a predefined schedule. Commonly employed learning rate schedules include time-based decay, step decay, and exponential decay.
- This approach enables the model to make substantial updates at the beginning of training when the parameters are far from their optimal values, and smaller updates later when the parameters are closer to their optimal values, thereby allowing for faster convergence.

6 Adagrad

We decay the learning rate for those weights which get more updates (using the previous update history).

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla w_t$$

where, v_t is the Learning Rate Decay factor,

ϵ is a small constant for numerical stability (for non-zero division)

and η is the Learning Rate.

- The learning rate decays very aggressively when using AdaGrad. The learning factor keeps increasing as the number of updates increases. This can make updates very small if the initial gradients are very large.

7 RMSprop

Rather than using a constantly decaying factor for the learning rate, we decay the decaying factor as the number of updates increase. This way we solve the problem faced in AdaGrad.

$$\begin{aligned}
v_0 &= 1 \\
v_t &= \beta v_{t-1} + (1-\beta)(\nabla w_t)^2 \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla w_t
\end{aligned}$$

where, v_t is the Learning Rate Decay factor,

β is the smoothing constant

ϵ is a small constant for numerical stability (for non-zero division)

and η is the Learning Rate.

- Using Momentum, we effectively increase our learning rate if the previous gradients are in the same direction. In RMSProp, we look at the "steepness" of the error surface for each parameter to update the learning rate adaptively. Parameters with high gradients get smaller update steps, and low gradients allow bigger steps.
- So, while Momentum is about accelerating in consistent directions, RMSProp is based on controlling overshooting by modulating step size.
- We maintain a moving (discounted) average of the square of gradients as the decay factor for the learning rate. This factor prioritizes newer gradients.
- We divide the learning rate by the root of this average as the decay for the learning rate.
- Using a decaying moving average of the partial derivative allows the search to forget early partial derivative values and focus on the most recently seen shape of the search space.
- This helps the optimizer to discard history from the extreme past so that it can converge rapidly after finding a convex surface.
- **Inclusion:** When Smoothing Constant (β) is set to 1 and ϵ tends to 0, RMSProp becomes SGD. So SGD $\subseteq$ RMSPROP.

$$\begin{aligned}
v_0 &= 1 \\
v_t &= v_{t-1}
\end{aligned}$$

Based on the above equations we get:

$$w_{t+1} = w_t - \eta \nabla w_t$$

8 Adam

So far, we have used the momentum term to determine the velocity of the gradient and update the weight parameter in the direction of that velocity.

In the case of AdaGrad and RMSProp, we used the sum of squared gradients to scale the current gradient so that we could update the weights in each space dimension with the same ratio. Adam is just a combination of these ideas.

Adam uses the Learning rate decay in AdaGrad and RMSProp while also using an additional moving average of the gradients to speedup the convergence.

$$\begin{aligned}
v_0 &= 1 \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t
\end{aligned}$$

where, m_t is the Moving average of gradients

v_t is the Learning Rate Decay factor,

β_1 is the Exponential Decay Rate - First Order Moment,

β_2 is the Exponential Decay Rate - Second Order Moment,

ϵ is a small constant for numerical stability (for non-zero division)

and η is the Learning Rate.

- By adapting the learning rate during training, Adam converges much more quickly than SGD.
- Only requiring first-order gradients and second-order gradients, Adam is computationally quick and easy to implement.
- Adaptive learning rates: Adam adapts each parameter's learning rate based on the gradients' first and second moments. This allows it to automatically adjust the step size for each parameter, making it well-suited for sparse and noisy data.
- Adam tweaks the gradient descent method by considering the moving average of the first and second-order moments of the gradient. This allows it to adapt the learning rates for each parameter intelligently.
- At its core, Adam is designed to adapt to the characteristics of the data. It does this by maintaining individual learning rates for each parameter in your model. These rates are adjusted as the training progresses, based on the data it encounters.
- Adam keeps track of the gradients from previous steps, allowing it to make informed adjustments to the parameters. This memory isn't just a simple average; it's a sophisticated combination of recent and past gradient information, giving more weight to the recent.
- Moreover, in areas where the gradient (the slope of the loss function) changes rapidly or unpredictably, Adam takes smaller, more cautious steps. This helps avoid overshooting the minimum. Instead, in areas where the gradient changes slowly or predictably, it takes larger steps.
- This adaptability is key to Adam's efficiency, as it navigates the loss landscape more intelligently than algorithms with a fixed step size.
- This adaptability makes Adam particularly useful in scenarios where the data or the function being optimized is complex or has noisy gradients
- Faster convergence: Adam combines the benefits of the Adagrad and RMSprop optimizers using adaptive learning rates and momentum. This often leads to faster convergence and better performance than other optimization algorithms.
- Robustness to hyperparameters: Adam is relatively insensitive to using hyperparameters such as the learning rate and momentum. This makes it easier and more robust to different data types and architectures.
- Suitable for large datasets: Adam is well-suited for large datasets and high-dimensional parameter spaces, as it efficiently computes and stores the first and second moments of the gradients.
- Widely used and supported: Adam is a popular optimization algorithm widely used and kept in popular deep learning frameworks such as TensorFlow and PyTorch.
- The small constant ϵ is added to prevent any issues with division by zero, which is especially important when the second moment estimate v$\hat{\text{t}}$ is very small.
- **Inclusion:** When β_1 is set to 0, β_2 is set to 1 and ϵ tends to 0, Adam becomes SGD. So SGD $\subseteq$ Adam.

$$v_0 = 1$$
$$m_t = \nabla w_t$$
$$v_t = v_{t-1}$$

Based on the above equations we get:

$$w_{t+1} = w_t - \eta \nabla w_t$$

8.1 Why Adam works best?

There are two reasons - Adaptive Learning Rate and Momentum speedup using Second order moments.

Adaptive Learning Rate:

- The Adam (Adaptive Moment Estimation) optimizer uses adaptive learning rates for each parameter, which helps in dealing with the problem of sparse gradients, exploding gradients, and vanishing gradients.
- Unlike traditional stochastic gradient descent (SGD) where the learning rate is static and must be manually tuned, Adam computes individual learning rates for different parameters.
- It does this by maintaining an exponentially decaying average of past gradients (m_t) and past squared gradients (v_t).
- This ensures that parameters with frequently changing gradients get lower learning rates, while parameters with infrequent updates get higher learning rates, allowing for more stable and efficient convergence.

Momentum Speedup:

- Adam incorporates momentum by using the first moment estimate (m_t). This means that the optimizer not only considers the current gradient but also accumulates an exponentially decaying moving average of past gradients to smooth out the update process.
- This accumulation helps in dampening the oscillations and allows the optimizer to navigate along long, narrow valleys and avoid getting stuck in local minima. By doing so, Adam effectively combines the benefits of both RMSprop (adaptive learning rates) and Momentum (smooth updates), leading to faster convergence especially in scenarios involving noisy gradients or non-stationary objectives.

8.2 What are disadvantages behind Adam? When it does not work?

- **Memory Requirements:** Adam stores the first and second moment estimates of gradients for each parameter. This can be computationally expensive for large models with a high number of parameters.
- **Susceptibility to Noise:** Adam's adaptive learning rate makes it sensitive to noisy gradients, particularly in scenarios with sparse data.

  Noisy gradients can lead the optimizer astray, resulting in suboptimal convergence or even divergence (failure to reach a minimum).
- **Biased Moment Estimates:** Early in training, Adam's first and second moment estimates are biased towards zero. This bias can slow down convergence and require more iterations to reach the optimal solution.
- **Hyperparameter Sensitivity:** Compared to other optimizers, Adam is generally less sensitive to hyperparameter tuning. However, it still requires careful selection of the learning rate, beta parameters (β_1, β_2), and epsilon (ϵ) to achieve optimal performance.

  Inappropriate hyperparameter choices can significantly impact the training process.
- **Convergence Issues:** In some cases, Adam may struggle to converge to the optimal solution in certain areas of the loss landscape.

  This limitation has led to the development of AMSGrad, an extension of Adam that aims to improve convergence by addressing large fluctuations in learning rates.
- **Weight Decay Issues:** The standard Adam implementation combined with L2 regularization might lead to worse generalization compared to SGD with momentum. This is because L2 regularization and weight decay have different effects in Adam.

  AdamW (Adam with decoupled weight decay) addresses this issue by incorporating weight decay in a way that improves generalization performance.

9 Coupled Adam

Coupled Adam improves the quality of learned embeddings by addressing the anisotropy issue observed when training with the Adam optimizer.

9.1 Anisotropy in Embedding Space

- Standard Adam causes learned embeddings to drift away from the origin during training.
- This leads to *anisotropic embeddings*, where vectors are unevenly distributed in space.
- Anisotropy reduces the geometric and semantic expressiveness of embeddings, negatively impacting downstream performance.

9.2 Per-Token Update Scaling in Adam

- Adam computes individual second moment estimates per parameter:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- Parameters associated with frequent tokens accumulate larger v_t values over time.
- Adam applies an adaptive update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$$

- As v_t increases, the denominator grows, resulting in **smaller updates** for frequent tokens.
- Infrequent tokens, with lower v_t, receive relatively larger updates.

9.3 Unbalanced Update Accumulation

- This frequency-dependent scaling causes imbalance in how embeddings evolve.
- The sum of embedding updates under Adam does not cancel out:

$$\sum_i \Delta\theta_i^{\text{Adam}} \neq 0$$

- Over time, this leads to a **global drift** in the embedding space, further increasing anisotropy.

9.4 SGD Doesn't Exhibit This Behavior

- With vanilla SGD, all tokens are treated equally:

$$\Delta\theta_i^{\text{SGD}} = -\eta \cdot g_t$$

- The updates across embeddings typically sum to zero:

$$\sum_i \Delta\theta_i^{\text{SGD}} = 0$$

- As a result, embeddings stay centered and isotropically distributed.

9.5 Using a global average instead of a second moment

- Instead of maintaining a separate second moment $v_{t,i}$ for each embedding parameter, use a global average:
$$\bar{v}_t = \frac{1}{N}\sum_{i=1}^{N} v_{t,i}$$
- Apply this average to all embedding updates:
$$\theta_{t+1} = \theta_t - \eta \cdot \frac{m_t}{\sqrt{\bar{v}_t} + \epsilon}$$
- This ensures that all embeddings receive updates scaled by the same adaptive factor, **coupling their dynamics**.
- This modification applies **only to embedding parameters** (e.g., token embeddings, position embeddings).
- Other weights in the model (e.g., attention, feedforward layers) continue using the standard Adam formulation.
- The implementation requires a minimal change to the optimizer logic — simply average v_t across embeddings.

9.6 Benefits of Coupled Adam

- Embedding updates become balanced and centered.
- Embedding vectors maintain an isotropic geometry throughout training.
- Results in **lower anisotropy scores**, indicating better spatial uniformity.
- Enhances the semantic quality of embeddings, improving representation usefulness.
- Leads to **slightly better perplexity** in language modeling tasks, across both small and large models.
- Demonstrated to work across various model scales (125M to 2.6B parameters) and datasets (OpenWebText, SlimPajama).
- Averaging the second moment cancels out the directional bias.
- Empirical experiments confirm improved isotropy and language model performance.
- The Coupled Adam optimizer retains all of Adam's core strengths: fast convergence, stability on sparse gradients, and adaptive learning rates.

10 Gradient Clipping

- Gradient clipping is a technique that tackles exploding gradients. The idea of gradient clipping is very simple: If the gradient gets too large, rescale it to keep it small.
- In Gradient clipping by value, if a gradient exceeds a threshold value, the gradient is clipped to the threshold. If the gradient is less than the lower limit then it is is clipped too, to the lower limit of the threshold.
- In Norm gradient clipping, the gradients are clipped by multiplying the unit vector of the gradients with the threshold.
- This ensures that the global norm of all the gradients calculated is not more than 1.

11 Gradient Accumulation

- During the training of neural networks, data is typically divided into mini-batches, which are processed iteratively. For each mini-batch, the network computes predicted labels, and the loss is calculated with respect to the ground truth targets. Subsequently, a backward pass is performed to compute the gradients, facilitating the updating of model weights in the direction of the gradients.
- Gradient accumulation modifies the final step of the training process. Instead of updating the network weights after processing each mini-batch, the gradient values are accumulated. The model proceeds to the next mini-batch, and the new gradients are added to the accumulated gradients. The weight update is then performed after several mini-batches have been processed by the model.
- The primary advantage of gradient accumulation is that it effectively simulates a larger batch size during training.
- Gradient accumulation provides a means to virtually increase the batch size during training, which is particularly beneficial when the available GPU memory is insufficient to accommodate the desired batch size.
- In the gradient accumulation technique, gradients are computed for smaller mini-batches and accumulated (typically through summation or averaging) over multiple iterations, rather than updating the model weights after each mini-batch. Once the accumulated gradients reach the target ”virtual” batch size, the model weights are updated using the accumulated gradients.