Optimizers

Generally speaking, the goal of the optimizer of a DNN is to find the best parameters to minimize the loss function. The gradient descent (GD) is historically the most frequently used optimization method for machine learning.

Gradient Descent (GD) is an optimization algorithm used to minimize a loss function $J(\theta)$, where θ represents the parameters of a machine learning model. The goal of GD is to find the values of θ that minimize the loss.

The algorithm iteratively updates the parameters in the opposite direction of the gradient of the loss function with respect to θ, because the gradient points towards the steepest increase of the function.

Here's how GD works using the formula for updating θ:

1. **Initialization**:
   - Initialize the parameters θ with some random values.
   - Choose a learning rate α, which is a hyperparameter that determines the size of the steps taken in the parameter space.

2. **Update Rule**:
   - At each iteration t, the parameters are updated using the following formula:
     $$
     \theta_{t+1} = \theta_{t} - \alpha \nabla J(\theta_{t})
     $$
   where:
     - θ_{t} represents the parameter values at iteration t.
     - $\nabla J(\theta_{t})$ is the gradient of the loss function J with respect to θ evaluated at θ_{t}.
     - α is the learning rate, which controls the size of the steps taken during each update.

3. **Convergence**:
   - Repeat the update process until a stopping criterion is met, such as a predefined number of iterations or until the change in $J(\theta)$ falls below a certain threshold.

4. **Interpretation**:
   - The gradient $\nabla J(\theta)$ points in the direction of the steepest increase of the loss function. Thus, the update rule moves θ in the direction that minimizes the loss.

5. **Loss Reduction**:
   - As the algorithm progresses through iterations, the loss function $J(\theta)$ typically decreases. Eventually, it converges to a local minimum (or, in convex problems, the global minimum) where the gradient is zero.

6. **Learning Rate Selection**:
   - The choice of learning rate α is crucial. A too small α leads to slow convergence, while a too large α can cause overshooting and oscillations.

7. **Stochastic Gradient Descent (SGD)**:
   - In practice, a variant called Stochastic Gradient Descent (SGD) is often used. It randomly samples a subset (mini-batch) of the training data to compute the gradient at each step. This can be computationally more efficient, especially for large datasets.

GD is a fundamental optimization algorithm widely used but it was essentially just the first step toward the modern tools, such as Adam, that are mostly used today.

To understand why the Adam optimizer is considered so much better, let's first explore the evolution of optimizers, starting with Gradient Descent (GD), and then delve into how Adam works.

1. **Gradient Descent (GD)**:
   - **How it works**:
     - GD is the most basic optimization algorithm. It aims to find the minimum of a loss function by iteratively moving in the direction of steepest descent (negative gradient).
     - At each iteration, it updates the parameters (weights) by subtracting a fraction of the gradient of the loss with respect to the parameters.
   - **Limitations**:
     - GD has trouble with noisy or ill-conditioned data. It can get stuck in local minima or plateaus, which can lead to slower convergence or suboptimal solutions.

2. **Momentum**:
   - **How it works**:
     - Momentum is an enhancement of GD. It accumulates a moving average of the gradients to dampen oscillations in the search path and accelerate convergence.
   - **Limitations**:
     - Momentum may overshoot the optimal point and struggle with highly non-convex functions.

3. **RMSprop** (Root Mean Square Propagation):
   - **How it works**:
     - RMSprop adapts the learning rate of each parameter based on the magnitude of its gradients. It divides the learning rate by an exponentially decaying average of squared gradients.
   - **Limitations**:
     - RMSprop can sometimes excessively reduce the learning rate for some parameters, leading to slower convergence.

4. **Adaptive Moment Estimation (Adam)**:
  - **How it works**:
    - Adam combines the benefits of both Momentum and RMSprop. It maintains two moving averages for each parameter: one for the first moment (mean of gradients) and one for the second moment (uncentered variance of gradients).
    - It then uses these moving averages to adaptively adjust the learning rates for each parameter.
    - The algorithm also incorporates bias correction to account for the initialization of the moving averages.
  - **Advantages**:
    - **Adaptability**: Adam adapts the learning rates for each parameter individually. This can be crucial for handling noisy gradients or varying scales of parameters.
    - **Efficiency**: Adam often converges faster than other optimizers and requires less hyperparameter tuning.
    - **Robustness**: It is well-suited for a wide range of problems and has been shown to perform well in practice.

  - **Limitations**:
    - **Sensitivity to Hyperparameters**: Although Adam is known for being robust, it can still be sensitive to the choice of hyperparameters like the learning rate and momentum decay.

Overall, Adam's adaptability and efficiency make it an excellent optimizer for a wide range of machine learning tasks. It has become a popular choice for training deep neural networks due to its ability to handle complex, high-dimensional optimization landscapes effectively.

So given that we generally have an optimizer that works great out of the box, why would we need to customize?

There are several scenarios where using a custom optimizer might be necessary or beneficial:

1. **Specialized Architectures or Loss Functions**:
  - Some architectures or loss functions might have specific characteristics that are not well-suited for standard optimizers. In such cases, designing a custom optimizer tailored to the problem can lead to better convergence and performance.

2. **Domain-Specific Knowledge**:
  - Domain experts may have insights into the problem that can be leveraged to design an optimizer that takes advantage of specific characteristics of the data or task.

3. **Non-Standard Constraints**:
  - If the optimization problem involves constraints (e.g., parameter bounds or linear equality/inequality constraints), a custom optimizer can be designed to handle these constraints efficiently.

4. **Performance Optimization**:
  - Standard optimizers may not perform optimally for certain types of models or datasets. Custom optimizers can be fine-tuned to exploit specific properties of the problem, leading to faster convergence and better solutions.

5. **Noise or Non-Stationarity in Gradients**:
  - In some cases, gradients can be noisy or non-stationary. Custom optimizers can incorporate techniques to handle noisy gradients or adapt learning rates dynamically.

6. **Research or Experimental Purposes**:
  - In research settings, creating a custom optimizer can be a way to explore novel optimization strategies and test hypotheses about optimization techniques.

7. **Hybrid Approaches**:
  - Combining elements of different optimization algorithms can lead to a custom optimizer that leverages the strengths of each component. This can be especially useful in complex, non-convex optimization problems.

8. **Hardware or Resource Constraints**:
  - The hardware or computational resources available may influence the choice of optimizer. Custom optimizers can be designed to make efficient use of specific hardware architectures.

9. **Historical Reasons**:
  - Legacy systems or codebases may have custom optimizers developed for historical reasons. These custom optimizers might still be used if they have been proven effective for a specific task.

10. **Learning Rate Schedules**:
    - Custom optimizers can incorporate sophisticated learning rate schedules or adaptively adjust learning rates based on the progress of optimization.

With what we are doing we know that there are many local minima that are not relevant to the solution of interest. We also know that the phase space is ultimately quite complex and there may need to be much exploration of both optimizer and loss function to truly find the most accurate and precise results. Generally, we are attempting to study a more simplified version of the very complex problem that involves a large covariance matrix of experimental error with unusual distribution shapes and its own kinematic sensitivity. In the simplified case, much of the error is simplified or ignored. We will gradually address this issue as we improve our extraction techniques and Monte Carlo.