Deep Learning Optimization Algorithms

- Training deep learning models means solving an optimization problem: The model is incrementally adapted to minimize an objective function.
- The optimizers used for training deep learning models are based on gradient descent, trying to shift the model's weights towards the objective function's minimum.
- A range of optimization algorithms is used to train deep learning models, each aiming to address a particular shortcoming of the basic gradient descent approach.
    - Stochastic Gradient Descent (SGD) and Mini-batch Gradient Descent speed up training and are suitable for larger datasets.
    - AdaGrad adapts learning rates to parameters but may slow down learning over time. RMSprop and AdaDelta build on AdaGrad's approach, addressing its diminishing learning rates, with AdaDelta removing the need for a set learning rate.
    - Adam combines the advantages of AdaGrad and RMSprop and is effective across a wide range of deep-learning tasks.

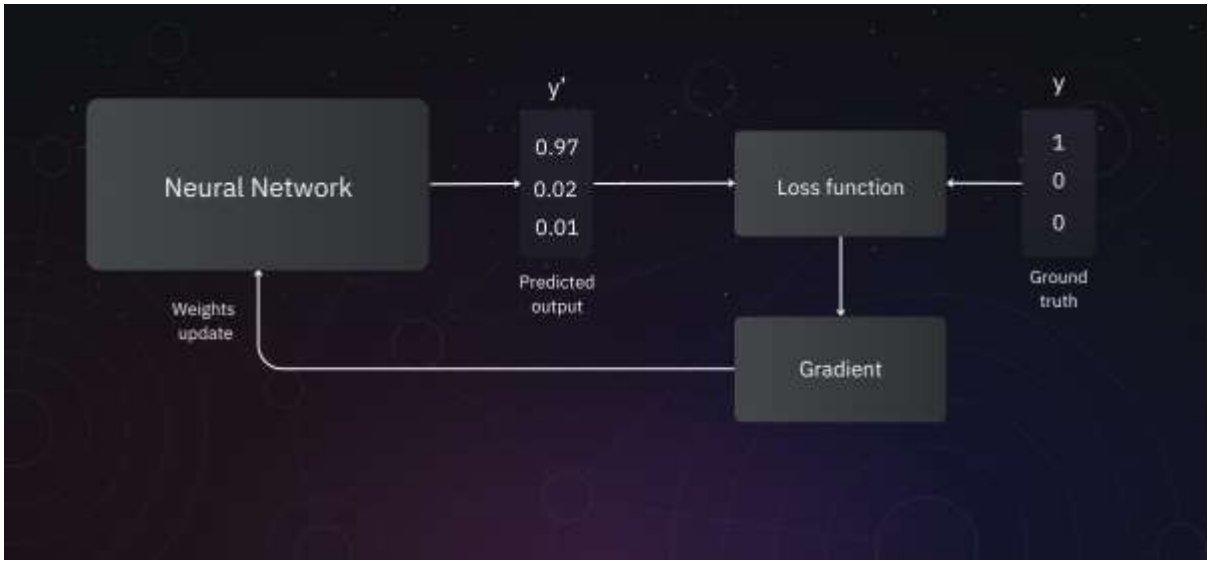
- Optimization algorithms play a crucial role in training deep learning models. They control how a neural network is incrementally changed to model the complex relationships encoded in the training data.

- With an array of optimization algorithms available, the challenge often lies in selecting the most suitable one for your specific project. Whether working on improving accuracy, reducing training time, or managing computational resources, understanding the strengths and applications of each algorithm is fundamental.

What is a model-optimization algorithm?

- A deep learning model comprises multiple layers of interconnected neurons organized into layers. Each neuron computes an activation function on the incoming data and passes the result to the next layer. The activation functions introduce non-linearity, allowing for complex mappings between inputs and outputs.

- The connection strength between neurons and their activations are parameterized by weights and biases. These parameters are iteratively adjusted during training to minimize the discrepancy between the model's output and the desired output given by the training data. The discrepancy is quantified by a loss function.



Schematic visualization of the deep learning model training process.

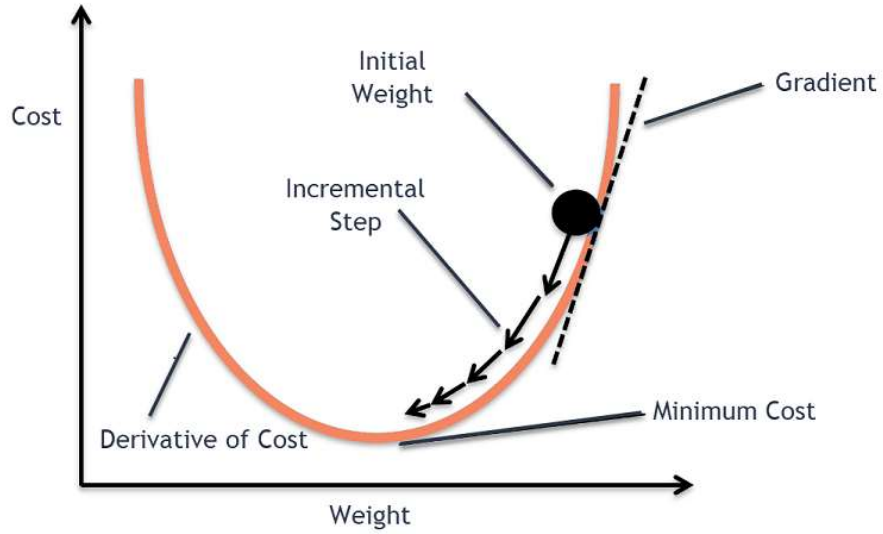
In each iteration of the training cycle, the neural network produces predictions on a batch of training samples. This predicted output is compared to the ground truth using a loss function. The gradient of the loss function with respect to the neural network's weights uncovers how these weights have to be updated to bring the model's outputs closer to the ground truth.

This adjustment is governed by an optimization algorithm. Optimizers utilize gradients computed by back propagation to determine the direction and magnitude of parameter updates, aiming to navigate the model's high-dimensional parameter space efficiently. Optimizers employ various strategies to balance exploration and exploitation, seeking to escape local minima and converge to optimal or near-optimal solutions.

Understanding different optimization algorithms and their strengths and weaknesses is crucial for any data scientist training deep learning models. Selecting the right optimizer for the task at hand is paramount to achieving the best possible training results in the shortest amount of time.

1. Gradient Descent

Gradient Descent is an algorithm designed to minimize a function by iteratively moving towards the minimum value of the function. It's akin to a hiker trying to find the lowest point in a valley shrouded in fog. The hiker starts at a random location and can only feel the slope of the ground beneath their feet. To reach the valley's lowest point, the hiker takes steps in the direction of the steepest descent. (For a more thorough mathematical explanation, refer these MIT lecture notes.)



The gradient descent optimization algorithm applied to a cost function. The cost function is convex, with a unique minimum. The gradient descent algorithm starts with a randomly selected initial weight. The gradient vector indicates the direction of the steepest ascent. The optimization process is illustrated by arrows representing incremental steps taken in the opposite direction of the gradient, moving the initial weight toward the minimum cost point on the curve.

All deep learning model optimization algorithms widely used today are based on Gradient Descent.

Objective: Gradient Descent aims to find a function's parameters (weights) that minimize the cost function. In the case of a deep learning model, the cost function is the average of the loss for all training samples as given by the loss function. While the loss function is a function of the model's output and the ground truth, the cost function is a function of the model's weights and biases.

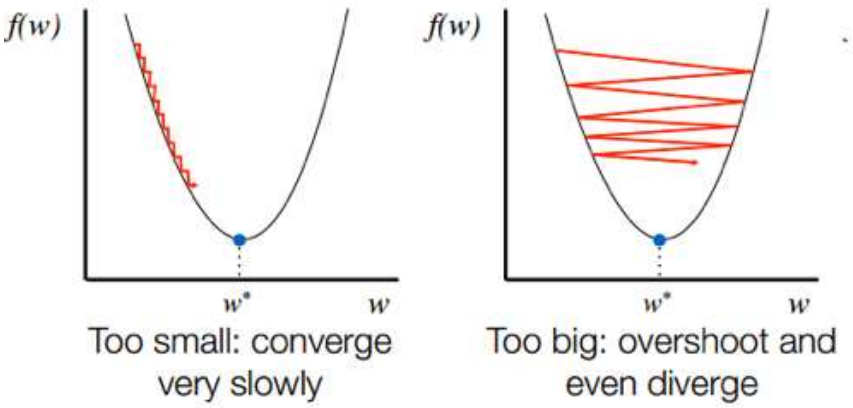
How it works:

- Initialization: Start with random values for the model's weights.
- Gradient computation: Calculate the gradient of the cost function with respect to each parameter. The gradient is a vector that points in the direction of the steepest increase of the function. In the context of optimization, we're interested in the negative gradient, which points towards the direction of the steepest decrease.
- Update parameters: Adjust the model's parameters in the direction opposite to the gradient. This step is done by subtracting a fraction of the gradient from the current values of the parameters. The size of this step is determined by the learning rate, a hyper-parameter that controls how fast or slow we move toward the optimal weights.
- Mathematical representation: the update rule for each parameter $\boldsymbol{w}$ can be mathematically represented as

$$w := w - \alpha \nabla_w J(w)$$

Where w represents the model's parameters (weights) and α is the learning rate. $\Delta w(\mathrm{w})$ is the gradient of the cost function $J(\mathrm{w})$ with respect to w.

The learning rate is a crucial hyper-parameter that needs to be chosen carefully. If it's too small, the algorithm will converge very slowly. If it's too large, the algorithm might overshoot the minimum and fail to converge.



An illustration of how different learning rate configurations can affect the convergence of the algorithm. If the learning rate is too small, convergence to the optimal value is slow. If the learning rate is too high, the optimization overshoots the minimum

Challenges:

- Local minima and saddle points: In complex cost landscapes, Gradient Descent can get stuck in local minima or saddle points, especially in non-convex optimization problems common in deep learning.
- Choosing the right learning rate: Finding an optimal learning rate requires experimentation and tuning.

2. Stochastic Gradient Descent (SGD)

- Stochastic Gradient Descent (SGD) is a variant of the traditional Gradient Descent optimization algorithm that introduces randomness into the optimization process to improve convergence speed and potentially escape local minima. To understand the intuition behind SGD, we can again invoke the analogy of a hiker descending a foggy valley. If Gradient Descent represents a cautious hiker who carefully evaluates the slope around them before taking a step, Stochastic Gradient Descent is akin to a more impulsive hiker who decides their next step based only on the slope of the ground immediately beneath their feet.

- **Objective**: like Gradient Descent, the primary goal of SGD is to minimize the cost function of a model by iteratively adjusting its parameters (weights). However, SGD aims to achieve this goal more efficiently by using only a single training example at a time to inform the update of the model's parameters.

- **How it works**:
  - Initialization: Start with a random set of parameters for the model.
  - Gradient computation: Instead of calculating the gradient of the cost function over the entire training data, SGD computes the gradient based on a single randomly selected training example.
  - Update parameters: Update the model's parameters using this computed gradient. The parameters are adjusted in the direction opposite to the gradient, similar to basic Gradient Descent.

- **Mathematical representation:**
  - The parameter update rule in SGD is similar to that of Gradient Descent but applies to a single example i:

$$w := w - \alpha \nabla_w J_i(w)$$

Here, w represents the model's parameters (weights), α is the learning rate, and $\Delta wJi(w)$ is the gradient of the cost function Ji(w) for the ith training example with respect to w.

Challenges:

- **Variance**: The updates can be noisy due to the reliance on a single example, potentially causing the cost function to fluctuate. As a result, the algorithm does not converge to a minimum but jumps around the cost landscape.
- **Hyper-parameter tuning**: Correctly setting the learning rate requires experimentation.

Advantages:

- **Efficiency**: Using only one example at a time, SGD significantly reduces the computational requirements, making it faster and more scalable than Gradient Descent.
- **Escape local minima**: The inherent noise in SGD can help the algorithm escape shallow local minima, potentially leading to better solutions in complex cost landscapes.
- **Online learning**: SGD is well-suited for online learning scenarios where the model needs to update continuously as new data arrives.

3. Mini-batch Gradient Descent

Mini-batch Gradient Descent strikes a balance between the thorough, calculated approach of Gradient Descent and the unpredictable, swift nature of Stochastic Gradient Descent (SGD). Imagine a group of hikers navigating through a foggy valley. Each hiker independently assesses a small, distinct section of the surrounding area before the group decides on the best direction to take.

Based on a broader but still limited view of the terrain, this collective decision-making process allows for a more informed and steady progression toward the valley's lowest point compared to an individual hiker's erratic journey.

- **Objective**: Similar to other gradient descent variants, the aim of Mini-batch Gradient Descent is to optimize the model's parameters to minimize the cost function. It seeks to combine the efficiency of SGD with the stability of Gradient Descent by using a subset of the training data to compute gradients and update parameters.

- **How it works:**

  Initialization: Start with initial random values for the model's parameters.

  Gradient computation: Instead of calculating the gradient using the entire dataset (as in Gradient Descent) or a single example (as in SGD), Mini-batch Gradient Descent computes the gradient using a small subset of the training data, known as a mini-batch.

  Update parameters: Adjust the parameters in the direction opposite to the computed gradient. This adjustment is made based on the gradient derived from the mini-batch, aiming to reduce the cost function.

- **Mathematical representation**:

  The parameter update rule for Mini-batch Gradient Descent can be represented as

$$w := w - \alpha \nabla_w J_{\text{mini-batch}}(w)$$

  Where w represents the model's parameters (weights), α is the learning rate, and $\Delta wJi(w)$ is the gradient of the cost function $Jmini\text{-}batch(w)$ for the current mini-batch of training samples with respect to w.

Challenges

- **Hyper-parameter tuning**: Like with the other variants we've discussed so far, selecting the learning rate requires experimentation. Further, we need to choose the batch size. If the batch size is too small, we face the drawbacks of SGD, and if the batch size is too large, we're prone to the issues of basic Gradient Descent.

- **Advantages**:

  Efficiency and stability: Mini-batch Gradient Descent offers a compromise between the computational efficiency of SGD and the stability of Gradient Descent.

  Parallelization: Since mini-batches only contain a small, fixed number of samples, they can be computed in parallel, speeding up the training process.

  Generalization: By not using the entire dataset for each update, Mini-batch Gradient Descent can help prevent overfitting, leading to models that generalize better on unseen data.

4. AdaGrad (Adaptive Gradient Algorithm)

AdaGrad (Adaptive Gradient Algorithm) introduces an innovative twist to the conventional Gradient Descent optimization technique by dynamically adapting the learning rate, allowing for a more nuanced and effective optimization process. Imagine a scenario where our group of hikers, navigating the foggy valley, now has access to a map highlighting areas of varying difficulty. With this map, they can adjust their pace — taking smaller steps in steep, difficult terrain and larger strides in flatter regions — to optimize their path toward the valley's bottom.

Objective: AdaGrad aims to fine-tune the model's parameters to minimize the cost function, similar to Gradient Descent. Its distinctive feature is individually adjusting learning rates for each parameter based on the historical gradient information for those parameters. This leads to more aggressive learning rate adjustments for weights tied to rare but important features, ensuring these parameters are optimized adequately when their respective features play a role in predictions.

How it works:

Initialization: Begin with random values for the model's parameters and initialize a gradient accumulation variable, typically a vector of zeros, of the same size as the parameters.
Gradient computation: Square and accumulate the gradients in the gradient accumulation variable, which consequently tracks the sum of squares of the gradients for each parameter.
Adjust learning rate: Modify the learning rate for each parameter inversely proportional to the square root of its accumulated gradient, ensuring parameters with smaller gradients to have larger updates.
Update parameters: Update each parameter using its adjusted learning rate and the computed gradient.

Mathematical representation:
The parameter update rule for Mini-batch Gradient Descent can be represented as

$$w := w - \frac{\alpha}{\sqrt{G_t + \epsilon}} \cdot \nabla_w J(w)$$

Where w represents the model's parameters (weights), α is the initial learning rate, G is the accumulation of the squared gradients, $\in$ is a small smoothing term to prevent division by zero, and $\Delta wJ(w)$ is the gradient of the cost function J(w) for the training samples with respect to w.

Advantages:
Adaptive learning rates: By adjusting the learning rates based on past gradients, AdaGrad can effectively handle data with sparse features and different scales.
Simplicity and efficiency: AdaGrad simplifies the need for manual tuning of the learning rate, making the optimization process more straightforward.

Challenges:

Diminishing learning rates: As training progresses, the accumulated squared gradients can grow very large, causing the learning rates to shrink and become infinitesimally small. This can prematurely halt the learning process.

5. RMSprop (Root Mean Square Propagation)

RMSprop (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm designed to address AdaGrad's diminishing learning rates issue.

Continuing with the analogy of hikers navigating a foggy valley, RMSprop equips our hikers with an adaptive tool that allows them to maintain a consistent pace despite the terrain's complexity. This tool evaluates the recent terrain and adjusts their steps accordingly, ensuring they neither get stuck in difficult areas due to excessively small steps nor overshoot their target with overly large steps. RMSprop achieves this by modulating the learning rate based on a moving average of the squared gradients.

Objective: RMSprop, like its predecessors, aims to optimize the model's parameters to minimize the cost function. Its key innovation lies in adjusting the learning rate for each parameter using a moving average of recent squared gradients, ensuring efficient and stable convergence.

How it works:

Initialization: Start with random initial values for the model's parameters and initialize a running average of squared gradients, typically as a vector of zeros of the same size as the parameters.

Compute gradient: Calculate the gradient of the cost function with respect to each parameter using a selected subset of the training data (mini-batch).

Update squared gradient average: Update the running average of squared gradients using a decay factor, γ, often set to 0.9. This moving average emphasizes more recent gradients, preventing the learning rate from diminishing too rapidly.

Adjust learning rate: Scale down the gradient by the square root of the updated running average, normalizing the updates and allowing for a consistent pace of learning across parameters.

Update parameters: Apply the adjusted gradients to update the model's parameters.

Mathematical representation:

The parameter update rule for RMSprop can be represented as follows:

$$w := w - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot \nabla_w J(w)$$

Here, w represents the parameters, α is the initial learning rate, $E[g^2]_t$ is the running average of squared gradients at a certain time, ∈ is a small smoothing term to prevent division by zero, and $\Delta w J(w)$ is the gradient of the cost function $J(w)$ with respect to w.

Advantages:
Adaptive learning rates: RMSprop dynamically adjusts learning rates, making it robust to the scale of gradients and well-suited for optimizing deep neural networks.
Overcoming AdaGrad's limitations: By focusing on recent gradients, RMSprop prevents the aggressive, monotonically decreasing learning rate problem seen in AdaGrad, ensuring sustained progress in training.

6. AdaDelta

AdaDelta is an extension of AdaGrad that seeks to reduce its aggressively decreasing learning rate.

Imagine our group of hikers now has an advanced tool that not only adapts to recent terrain changes but also ensures their gear weight doesn't hinder their progress. This tool dynamically adjusts their stride length, ensuring they can maintain a steady pace without the burden of past terrain slowing them down. Similarly, AdaDelta modifies the learning rate based on a window of recent gradients rather than accumulating all past squared gradients.

This approach allows for a more robust and adaptive learning rate adjustment over time.

Objective: AdaDelta aims to minimize the cost function by adaptively adjusting the model's parameters while avoiding the rapid decrease in learning rates encountered by AdaGrad. It focuses on using a limited window of past squared gradients to compute adjustments, thus avoiding the pitfalls of a perpetually diminishing learning rate. Unlike RMSprop, which addresses the diminishing learning rate issue by using a moving average of squared gradients, AdaDelta eliminates the need for an external learning rate parameter entirely. Instead, parameter updates are calculated utilizing the ratio of the RMS of parameter updates to the RMS of gradients.

How it works:
- **Initialization**: start with random initial parameters and two state variables: one for accumulating gradients and another for accumulating parameter updates, both initially set to zero.
- **Compute gradient**: determine the gradient of the cost function with respect to each parameter using a subset of the training data (mini-batch).
- **Accumulate gradients**: update the first state variable with the squared gradients, applying a decay factor to maintain a focus on recent gradients.
- **Compute update**: determine the parameter updates based on the square root of the accumulated updates state variable divided by the square root of the accumulated gradients state variable, adding a small constant to avoid division by zero.
- **Accumulate updates**: update the second state variable with the squared parameter updates.
- **Update parameters**: modify the model's parameters using the computed updates.

Mathematical representation:
The parameter update rule for AdaDelta is more complex due to its iterative nature but can be generalized as

$$w = w - \frac{\text{RMS}[u]_{t-1}}{\text{RMS}[g]_t} \cdot \nabla_w J(w)$$

Where w represents the parameters, $RMS[u]_{t-1}$ is the root mean square of parameter updates up to the previous time step, $RMS[g]_t$ is the root mean square of gradients at the current time step, and $\Delta w J(w)$ is the gradient of the cost function $J(w)$ with respect to w.

Advantages:

- **Self-adjusting learning rate**: AdaDelta requires no initial learning rate setting, making it easier to configure and adapt to different problems.
- **Addressing diminishing learning rates**: by limiting the accumulation of past gradients to a fixed-size window, AdaDelta mitigates the issue of diminishing learning rates, ensuring more sustainable and effective parameter updates.

Challenges:

- **Complexity**: AdaDelta's mechanism, involving the tracking and updating of two separate state variables for gradients and parameter updates, adds complexity to the algorithm. This can make it more challenging to implement and understand compared to simpler methods like SGD.
- **Convergence rate**: AdaDelta might converge more slowly than other optimizers like Adam, especially on problems where the learning rate needs more aggressive tuning. The self-adjusting learning rate mechanism can sometimes be overly cautious, leading to slower progress.

7. Adam (Adaptive Moment Estimation)
Adam (Adaptive Moment Estimation) combines the best properties of AdaGrad and RMSprop to provide an optimization algorithm that can handle sparse gradients on noisy problems.

Using our hiking analogy, imagine that the hikers now have access to a state-of-the-art navigation tool that not only adapts to the terrain's difficulty but also keeps track of their direction to ensure smooth progress. This tool adjusts their pace based on both the recent and accumulated gradients, ensuring they efficiently navigate towards the valley's bottom without veering off course. Adam achieves this by maintaining estimates of the first and second moments of the gradients, thus providing an adaptive learning rate mechanism.

Objective: Adam seeks to optimize the model's parameters to minimize the cost function, utilizing adaptive learning rates for each parameter. It uniquely combines momentum

(keeping track of past gradients) and scaling the learning rate based on the second moments of the gradients, making it effective for a wide range of problems.

How it works:

- **Initialization**: Start with random initial parameter values and initialize a first moment vector (m) and a second moment vector (v). Each "moment vector" stores aggregated information about the gradients of the cost function with respect to the model's parameters:
    - The first moment vector accumulates the means (or the first moments) of the gradients, acting like a momentum by averaging past gradients to determine the direction to update the parameters.

    - The second moment vector accumulates the variances (or second moments) of the gradients, helping adjust the size of the updates by considering the variability of past gradients.

- Both moment vectors are initialized to zero at the start of the optimization. Their size is identical to the size of the model's parameters (i.e., if a model has N parameters, both vectors will be vectors of size N).

- Adam also introduces a bias correction mechanism to account for these vectors being initialized as zeros. The vectors' initial state leads to a bias towards zero, especially in the early stages of training, because they haven't yet accumulated enough gradient information. To correct this bias, Adam adjusts the calculations of the adaptive learning rate by applying a correction factor to both moment vectors. This factor grows smaller over time and asymptotically approaches 1, ensuring that the influence of the initial bias diminishes as training progresses.

    - **Compute gradient:** For each mini-batch, compute the gradients of the cost function with respect to the parameters.
    - **Update moments**: Update the first moment vector (m) with the bias-corrected moving average of the gradients. Similarly, update the second moment vector (v) with the bias-corrected moving average of the squared gradients.
    - **Adjust learning rate**: Calculate the adaptive learning rate for each parameter using the updated first and second moment vectors, ensuring effective parameter updates.
    - **Update parameters**: Use the adaptive learning rates to update the model's parameters.
    - The second moment vector accumulates the variances (or second moments) of the gradients, helping adjust the size of the updates by considering the variability of past gradients.

- **Mathematical representation**:

The parameter update rule for Adam can be expressed as

$$w = w - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where w represents the parameters, α is the learning rate, and m_t and v_t are bias-corrected estimates of first and second moments of the gradients, respectively.

- **Advantages**:
  - **Adaptive learning rates**: Adam adjusts the learning rate for each parameter based on the estimates of the gradients' first and second moments, making it robust to variations in gradient and curvature.
  - **Bias correction**: The inclusion of bias correction helps Adam to be effective from the very start of the optimization process.
  - **Efficiency**: Adam is computationally efficient and is well-suited for problems with large datasets or parameters.

Choose an algorithm to optimize your model

Algorithm	Pros	Cons	When to use
Gradient Descent	• Simple and easy to implement	• Computationally expensive on large datasets • May get stuck in local minima	• Small datasets • When simplicity and clarity are preferred
SGD	• Fast • Handles large datasets well	• High variance in updates can lead to instability	• Large datasets; Online or incremental learning scenarios
Mini-batch Gradient Descent	• Balances between efficiency and stability • More generalizable updates	• Requires tuning of batch size	• Most deep learning tasks, especially when working with moderate to large datasets
AdaGrad	• Adapts learning rate for each parameter • Good for sparse data	• Learning rate can diminish to zero, stopping learning	• Sparse datasets like text and images where learning rate needs to adapt to feature

Algorithm	Pros	Cons	When to use
			frequency
RMSprop	• Addresses diminishing learning rate of AdaGrad • Adapts learning rate based on recent gradients	• Can still get stuck in local minima on non-convex problems	• Non-convex optimization problems • Situations where AdaGrad's learning rate diminishes too fast
AdaDelta	• Eliminates the need to set a default learning rate • Addresses diminishing learning rate issue	• More complex than RMSprop and AdaGrad	• Similar to RMSprop, but when you want to avoid manual learning rate setting
Adam	• Combines advantages of AdaGrad and RMSprop • Adaptive learning rates; Includes bias correction	• Requires tuning of hyperparameters (though it often works well with defaults)	• Most deep learning applications, due to its efficiency and effectiveness

References:

1. notes_chapter_Gradient_Descent.pdf (mit.edu)
2. Deep Learning (srdas.github.io)
3. How to Visualize Deep Learning Models (neptune.ai)
4. minibatch_sgd.pdf (cmu.edu)
5. duchi11a.dvi (jmlr.org)
6. lecture_slides_lec6.pdf (toronto.edu)
7. [1212.5701v1] ADADELTA: An Adaptive Learning Rate Method (arxiv.org)
8. [1412.6980] Adam: A Method for Stochastic Optimization (arxiv.org)