Lecture 2: Optimization Algorithms in Neural Networks

What is Optimization in Neural Networks

Neural networks are trained by adjusting their parameters (weights and biases) to minimize the loss function. This is achieved through optimization algorithms that iteratively update the parameters based on the gradients of the loss function. The efficiency and effectiveness of these optimization algorithms significantly impact the performance of the neural network.

* **Note:** Using **chain rule** in training of neural networks, because these networks are **composite** functions. Outputs are composed of functions of functions (e.g., activation, weighted sum, loss). So, to compute gradients, we apply the **chain rule** from calculus.

Important Deep Learning Terms

Epoch: The number of times the algorithm runs on the whole training dataset.

Sample: A single row of a dataset.

Batch: It denotes the number of samples to be taken to for updating the model parameters.

Learning rate: It is a parameter that provides the model a scale of how much model weights should be updated.

Cost Function/Loss Function: A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.

Weights/ Bias: The learnable parameters in a model that controls the signal between two neurons.

Common Optimization Algorithms in Neural Networks

1 Gradient Descent

Gradient Descent is the most basic and widely used optimization algorithm in neural networks. It involves updating the parameters in the direction of the negative gradient of the loss function. The key steps in Gradient Descent are

1. Initialize parameters: Randomly initialize the model parameters.

2. Compute the gradient: Calculate the gradient (derivative) of the loss function with respect to the parameters. The gradients are calculated via chain-rule.

3. Update parameters: Adjust the parameters by moving in the opposite direction of the gradient, scaled by the learning rate using the following formula:

$$\boldsymbol{w_{ji}(t+1) = w_{ji}(t) - \eta * g_{j,i}}$$

where $\boldsymbol{g_{j,i} = \delta_j * a_i}$, and $\boldsymbol{t}$ is a time step.

Where, $\boldsymbol{w_{ji}(t+1)}$ is the new weight value at $\boldsymbol{t+1^{th}}$ step, $\boldsymbol{w_{ji}(t)}$ is the old weight value at a previous step, $\boldsymbol{\eta}$ is the learning rate, which is a positive scalar that determines the step size for each iteration, $\boldsymbol{g_{j,i}}$ is the **gradient of the loss function** with respect to the weight ($\boldsymbol{w_{ij}}$).

* In traditional gradient descent, the **gradients are computed based on the entire dataset**, which can be computationally expensive for large datasets.

Variants of Gradient Descent:

2. Stochastic Gradient Descent (SGD)

It **updates the model parameters after each training example**, making it more efficient for large datasets compared to traditional Gradient Descent, which uses the entire dataset for each update. It uses the same training rule as Gradient Descent algorithm.

Advantages of SGD:

(1) Efficiency: Because it uses only one or a few data points to calculate the gradient, SGD can be much faster, especially for large datasets. Each step requires fewer computations, leading to quicker convergence.

(2) Memory Efficiency: Since it does not require storing the entire dataset in memory for each iteration, SGD can handle much larger datasets than traditional gradient descent.

(3) Escaping Local Minima: The noisy updates in SGD, caused by the stochastic nature of the algorithm, can help the model escape local minima or saddle points, potentially leading to better solutions in nonconvex optimization problems (common in deep learning).

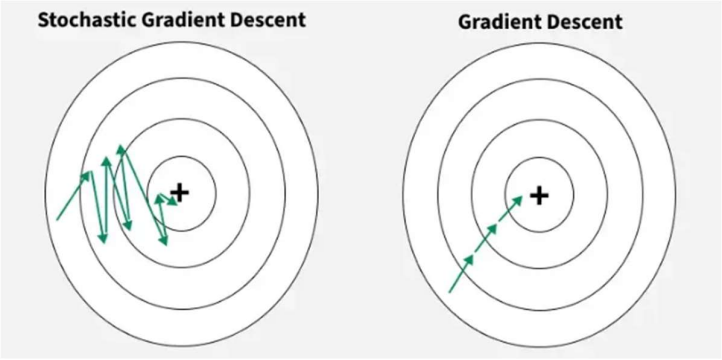
(4) Online Learning: SGD is well-suited for online learning, where the model is trained incrementally as new data comes in, rather than on a static dataset.

Disadvantages of SGD:

(1) Noisy Convergence: Since the gradient is estimated based on a single data point (or a small batch), the updates can be noisy, causing the cost function to fluctuate rather than steadily decrease. This makes convergence slower than in batch gradient descent.

(2) Learning Rate Tuning: SGD is highly sensitive to the choice of learning rate. A learning rate that is too large may cause the algorithm to diverge, while one that is too small can slow down convergence. Adaptive methods like Adam and RMSprop address this by adjusting the learning rate dynamically during training.

(3) Long Training Times: While each individual update is fast, the convergence might take a longer time overall since the steps are more erratic compared to batch gradient descent.



---

3. Mini Batch Stochastic Gradient Descent

It **consists of a predetermined number of training examples, smaller than the full dataset**. This approach combines the advantages of the previously mentioned variants. In one epoch, following the creation of fixed-size mini-batches, we execute the following steps:

(1) Select a mini-batch, **(2)** Compute the mean gradient of the mini-batch, **(3)** Apply the mean gradient obtained in step 2 to update the model's weights, and **(4)** Repeat steps 1 to 2 for all the mini-batches that have been created.

Advantages: Requires medium amount of memory and less time required to converge when compared to SGD. This provides a balance between the efficiency of SGD and the stability of batch gradient descent. It reduces the noise in the updates while maintaining the computational efficiency.

Disadvantage: May get stuck at local minima

Applications of Stochastic Gradient Descent

(1) Deep learning: In training deep neural networks, SGD is the default optimizer due to its efficiency with **large datasets** and its ability to work with **large models**.

(2) Natural Language Processing (NLP): Models like **Word2Vec** and **transformers** are trained using SGD variants to optimize large models on vast text corpora.

(3) Computer Vision: For tasks such as **image classification**, **object detection** and **segmentation**, SGD has been fundamental in training convolutional neural networks (**CNN**s).

(4) Reinforcement Learning: SGD is also used to optimize the parameters of models used in reinforcement learning, such as **deep Q-networks (DQNs)**.

---

4. SGD with Momentum

Momentum helps accelerate convergence by smoothing out the noisy gradients of SGD, thus reducing fluctuations and improving the speed of convergence. The gradients are computed via

$$\boldsymbol{v(t+1) = \beta * v(t) + (1-\beta) * g_{j,i}}$$

where $\boldsymbol{v(new)}$ is the new moment value, $\boldsymbol{\beta}$ is the momentum coefficient, and $\boldsymbol{v(t)} = 0$ at $\boldsymbol{t = 0}$.

Then, the model parameters are updated using:

$$\boldsymbol{w_{ji}(t+1) = w_{ji}(t) - \eta * v(new)}$$

This algorithm adds a fraction of the previous update to the current one. This allows the algorithm to keep moving in the same direction and can help overcome oscillations in the cost function.

Advantages: Mitigates oscillations, reduces variance, and faster convergence.

Disadvantages: Requires tuning the momentum coefficient $\boldsymbol{\beta}$.

Advanced Optimizers

5. AdaGrad

AdaGrad adapts the learning rate for each parameter based on the historical gradient information. The learning rate decreases over time, making AdaGrad effective for sparse features. Adagrad **adjusts the learning rate** for each parameter **based** on the **accumulated sum of squared gradients (*s*)**.

$$\boldsymbol{s(t+1) \leftarrow s(t) + g^2}$$

$$w_{ji}(t+1) = w_{ji}(t) - \frac{\eta}{\sqrt{s+\varepsilon}} * g_{j,i}$$

Where ε is a small constant to avoid division by zero, s is the sum of squared gradients, g^2 is a squared gradient value, and $s(t) = 0$ at $t = 0$.

Adagrad's formula automatically adapts the learning rate, here's how:

Frequent updates (large accumulated gradient): For parameters that are updated often, s become large, making the learning rate smaller. This prevents the parameter from changing too drastically and stabilizes learning.

Infrequent updates (small accumulated gradient): For parameters that are updated less frequently, s stay small, keeping the learning rate larger. This allows for more significant updates when necessary.

Advantages: Adapts the learning rate and eliminates the needs for manual tuning, improving training efficiency, help with sparse features and noisy data.

Disadvantages: Learning rate decays too quickly, causing slow convergence and early stopping, lacks momentum, making it harder to escape shallow local minima, Performance depends heavily on the initial learning rate choice.

---

6. RMSProp

RMSProp improves upon AdaGrad by introducing a **decay factor** to **prevent the learning rate from decreasing too rapidly**. Im more detail, instead of accumulating all past squared gradients as AdaGrad does, RMSProp uses a moving average. This means that RMSProp gives more weight to recent gradients, allowing it to adapt more effectively without diminishing the learning rate too quickly. The **steps of RMSProp** algorithm are **(1)** calculate the gradient (g) at time step t, **(2)** Update the moving average of squared gradients (ss), and **(3)** Update the parameters using the adjusted learning rate.

$$ss(t+1) = \rho * ss(t) + (1-\rho) * g^2$$

$$w_{ji}(t+1) = w_{ji}(t) - \frac{\eta}{\sqrt{ss+\varepsilon}} * g_{j,i}$$

Where ρ is the decay rate, ss is the exponentially moving average of squared gradients, and $ss(t) = 0$ at $t = 0$.

• **Advantages:** Prevents excessive decay of learning rates by adapting learning rates based on the moving average of squared gradients, which helps in dealing with the

challenges of non-stationary objectives and sparse gradients commonly encountered in deep learning tasks.

• **Disadvantages:** Computationally expensive due to the additional parameter, does not inherently incorporate momentum.

---

7. Adam (Adaptive Moment Estimation)

Adam combines the advantages of **Momentum** and **RMSProp**. It uses both the first moment (**mean**) and second moment (**variance**) of gradients to adapt the learning rate for each parameter. The steps of Adam algorithm are **(1)** estimate the first moment, **(2)** estimate the second moment, **(3)** correct bias since both moments are initialized at zero, they tend to be biased, and **(4)** update the parameters.

1) Update the first moment, $\boldsymbol{m(t+1) = \beta_1 * m(t) + (1-\beta_1) * g}$

2) Update the second moment, $\boldsymbol{v(t+1) = \beta_2 * v(t) + (1-\beta_2) * g^2}$

3) Bias correction, $\boldsymbol{\hat{m} = \frac{m}{1-\beta_1^t}}$ and $\boldsymbol{\hat{v} = \frac{v}{1-\beta_2^t}}$

4) Update weights, $\boldsymbol{w_{ji}(t+1) = w_{ji}(t) - \frac{\eta}{\sqrt{\hat{v}+\varepsilon}} * \hat{m}}$

Where $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_2}$ are the decay rates for the first and second moments. $\boldsymbol{m(t)}$ and $\boldsymbol{v(t)}$ start at 0 for $\boldsymbol{t = 0}$.

• **Advantages:** Fast convergence, it works well with large datasets and complex models because it adapts the learning rate for each parameter automatically.

• **Disadvantages:** Requires significant memory due to the need to store first and second moment estimates.

Optimizer	Advantages	Disadvantages
SGD	Simple, easy to implement	Slow convergence, requires tuning
Mini-Batch SGD	Faster than SGD	Computationally expensive, stuck in local minima
SGD with Momentum	Faster convergence, reduces noise	Requires careful tuning of βββ
AdaGrad	Adaptive learning rates	Decays too fast, slow convergence
RMSProp	Prevents fast decay of learning rates	Computationally expensive
Adam	Fast, combines momentum and RMSProp	Memory-intensive, computationally expensive

8. AdamW

AdamW is a smarter version of Adam as it decouples weight decay from the gradient update step. Instead of adding weight decay to the loss function, it applies weight decay directly during the parameter update, leading to more consistent regularization and better generalization. The steps pf AdamW algorithm are

1. Momentum and adaptive learning rate:

$$\boldsymbol{m(t+1) = \beta_1 * m(t) + (1-\beta_1) * g}$$

$$\boldsymbol{v(t+1) = \beta_2 * v(t) + (1-\beta_2) * g^2}$$

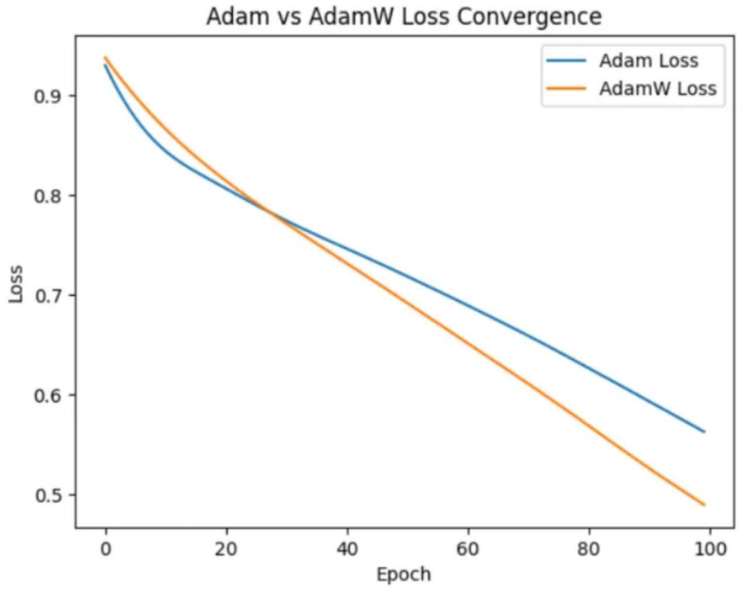
2. Bias-corrected estimates:

$$\boldsymbol{\hat{m} = \frac{m}{1-\beta_1^t}} \quad \text{and} \quad \boldsymbol{\hat{v} = \frac{v}{1-\beta_2^t}}$$

3. Parameter update with decoupled weight decay: In AdamW, weight decay is applied directly to the parameters after the gradient update. The update rule is

$$\boldsymbol{w_{ji}(t+1) = w_{ji}(t) - \eta * \frac{\hat{m}}{\sqrt{\hat{v}+\varepsilon}} - \eta * \lambda * w_{ji}(t)}$$

Here, $\boldsymbol{\lambda}$ is the weight decay factor. The decoupled weight decay term $\boldsymbol{\lambda * w_{ji}(old)}$ ensures that regularization is applied independently of the gradient update, which is the key difference from Adam.



Advantages of AdamW Over Adam

1. Decoupled weight decay. By separating weight decay from gradient updates, AdamW allows for more precise control over regularization, leading to better model generalization.

2. Enhanced generalization. AdamW reduces the risk of overfitting, especially in large-scale models, making it suitable for tasks involving extensive datasets and intricate architectures.

3. Stability during training. The design of AdamW helps maintain stability throughout the training process, which is essential for models that require careful tuning of their hyperparameters.

4. Scalability. AdamW is particularly effective for scaling up models, as it can handle the increased complexity of deep networks without sacrificing performance, allowing it to be applied in state-of-the-art architectures.

Practical Recommendations:

When to Choose Adam: You can use Adam for quick prototyping or simpler tasks where regularization is not crucial. It may converge faster initially but can suffer from poor generalization due to interference from weight decay.

When to Choose AdamW: In case you have larger models or when training on complex, high-dimensional data, it's better to choose AdamW, because the decoupled weight decay helps achieve better generalization and stable convergence.

Hyperparameter Tuning in AdamW

Hyperparameter tuning is the process of selecting the best values for parameters that govern the training of a machine learning model but are not learned from the data itself. These parameters directly influence how the model optimizes and converges.

Proper tuning of these hyperparameters in AdamW is essential for achieving efficient training, avoiding overfitting, and ensuring the model generalizes well to unseen data.

Best practices for choosing learning rates and weight decay

1. The learning rate is a hyperparameter that controls how much to adjust the model weights with respect to the loss gradient during each training step. A **higher learning rate** speeds up training but may cause the model to overshoot optimal weights, while a **lower rate** allows for more fine-tuned adjustments but can make training slower or get stuck in local minima.

2. Weight decay is a regularization technique used to prevent overfitting by penalizing large weights in the model. Namely, weight decay adds a small penalty proportional to the size of the model weights during training, helping to reduce model complexity and improve generalization to new data.

To choose optimal learning rates and weight decay values for AdamW:

1. Start with a moderate learning rate – For AdamW, a learning rate around **1e-3** is often a good starting point. You can adjust it based on how well the model converges, lowering it if the model struggles to converge or increasing it if training is too slow.

2. Experiment with weight decay. Start with a value around **1e-2** to **1e-4**, depending on the model size and dataset. A slightly higher weight decay can help prevent overfitting for larger, complex models, while smaller models may need less regularization.

3. Use learning rate scheduling. Implement learning rate schedules (like step decay or cosine annealing) to dynamically reduce the learning rate as training progresses, helping the model fine-tune its parameters as it approaches convergence.

4. Monitor performance. Continuously track model performance on the validation set. If you observe overfitting, consider increasing weight decay, or if the training loss plateaus, lower the learning rate for better optimization.