# Task 1

1. Understanding and comparing various Optimizer methods in training machine learning models.
2. Exploring "Continual Learning and Test Production" in machine learning.

# Optimizers

## 1. Gradient Descent

The gradient of a function $f$ is the vector field $\nabla f$ whose value at a point $p$ gives the direction and rate of fastest increase. If the gradient of the function is non-zero at $p$, the direction of the gradient is the direction in which the function increases most quickly from $p$, and the magnitude of the gradient is the rate of increase in that direction.

The gradient of a function $f(p)$ for $f : \mathbb{R}^n \to \mathbb{R}$ where $p = (x_1, x_2, \ldots, x_n)$ is defined as:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

A point where the gradient is the zero vector is known as a stationary point. A stationary point is a point where the function "stops" increasing or decreasing.

- A **local minima** is one where the derivative of the function changes from negative to positive.
- A **local maxima** is one where the derivative of the function changes from positive to negative.

Using the gradient, we can (hopefully) find a way to reach the local minima of an unknown function $F$, given $\nabla F$ (often the case in the real world). This is called **Gradient descent**.

Considering the neighborhood of a point $\mathrm{a}$, $F$ decreases fastest if one goes from $\mathrm{a}$ in the direction of the negative gradient of $F$ at $\mathrm{a}$, $-\nabla F(a)$.

If $\mathrm{a}_{n+1} = \mathrm{a}_n - \gamma \nabla F(\mathrm{a})$, then $F(\mathrm{a}_n) \geq F(\mathrm{a}_{n+1})$ for a small enough learning rate $\gamma \in \mathbb{R}_+$. The term $\nabla F(\mathrm{a})$ is subtracted from $\mathrm{a}$ because we want to move against the gradient and toward the local minima.

The process of gradient descent starts with a guess $x_0$ for a local minima of $F$, then improve upon this guess for $x_1, x_2, \ldots$ such that $x_{n+1} = x_n - \gamma \nabla F(x), n \geq 0$. Hopefully, the sequence $(x_n)$ converges to the desired local minimum.

In the context of **Multivariate Linear Regression**, given a linear equation

$$y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

one can use Gradient Descent to find the weights $(w)$ given the input features $x$ and expected output $y$.

The function that we want to find the local minima of for this problem is a **loss function**, that is how far off the predicted output is from the actual output. There are many types of loss functions, one of which is the **Mean Squared Error** (MSE).

The **hypothesis** $\hat{y}$ given a learned set of weights $(w)$ is:

$$\hat{y} = \sum_{i=0}^{N} (w_j x_{ij})$$

The **Mean Square Error** loss function of the expected output $y$ and the hypothesis $\hat{y}$

$$L(w_0, w_1, \ldots, w_n) = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2$$

where $N$ is the number of samples in the training dataset.

The gradient $\nabla L$ is found by taking the derivative of the loss function

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix} = \begin{bmatrix} -\frac{2}{N}\Sigma_{i=0}^{N}(y_i - \hat{y}_i) \\ -\frac{2}{N}\Sigma_{i=0}^{N}(y_i - \hat{y}_i)x_{i1} \\ \vdots \\ -\frac{2}{N}\Sigma_{i=0}^{N}(y_i - \hat{y}_i)x_{in} \end{bmatrix}$$

Finally, we update the weights via the formula

$$w_j := w_j - \alpha\frac{\partial L}{\partial w_j}, j = 0, 1, \ldots n$$

where $\alpha$ is the learning rate and $n$ is the number of features.

This process is repeated until the algorithm converges to a minimum, which is indicated by the gradient becoming very small (almost zero). The learning rate $\alpha$ determines how big the steps the algorithm takes towards the minimum are. If $\alpha$ is too large, the algorithm might overshoot the minimum and diverge. If $\alpha$ is too small, the algorithm will need many iterations to converge and might get stuck in a local minimum.

The main challenge with Gradient Descent is that it can get stuck in local minima. This is not a problem when the error surface (the Loss Gradient $\nabla L$) is convex, such as in Linear Regression. However, in Neural Networks, the error surface can have many local minima, and Gradient Descent can get stuck in a suboptimal solution. This is why modifications and alternatives to Gradient Descent, such as Stochastic Gradient Descent and Mini-Batch Gradient Descent, are often used in practice. These methods introduce randomness into the algorithm, which can help it escape from local minima.

## 2. Stochastic Gradient Descent (SGD)

**Stochastic Gradient Descent** (SGD) is an optimization method commonly used in machine learning algorithms, especially those dealing with large datasets. The main idea behind SGD is to use a single or a few randomly selected samples to estimate the gradient, rather than using the entire dataset.

Consider the loss function

$$L(w) = \frac{1}{N}\sum_{i=0}^{N}L_i(w)$$

where $L_i(w)$ is the loss for the $i$-th sample given the weights $w$, and $N$ is the number of samples in the dataset.

In SGD, for each epoch, we shuffle the dataset, then update the weights for each example $i$ using the following rule:

$$w := w - \alpha\nabla L_i(w)$$

Here, $\alpha$ is the learning rate, and $\nabla L_i(w)$ is the gradient of the loss for the $i$-th sample with respect to the weights.

The key difference between SGD and Gradient Descent is that in SGD, we perform the update on each training sample (or a small batch of samples), instead of updating the weights according to the entire dataset. This is denoted by $\nabla L_i(w)$ versus $\frac{\partial L}{\partial w_j} \in \nabla L$.

**This approach has several advantages:**

- **Computational efficiency**: SGD is much faster than Gradient Descent when dealing with large datasets, as it only needs to load one sample (or a small batch) into memory at a time.
- **Frequent updates**: With frequent updates, SGD can have a faster convergence and can also escape local minima more easily.
- **Ability to handle redundant data**: In some cases, the training data may contain redundant information. In such cases, SGD can reach the optimal solution faster by not using all the redundant data.

**However, SGD also has some disadvantages:**

- **Noisy updates**: Since SGD uses only one sample (or a small batch) to compute the gradient, the updates can be noisy, leading to a non-smooth path towards the minimum.
- **Requires tuning of learning rate**: The learning rate $\alpha$ often needs to be decreased over time to ensure convergence to the minimum, which adds an extra hyperparameter to tune.

Despite these challenges, SGD remains a popular optimization method due to its efficiency and simplicity.

## 3. Mini-Batch Gradient Descent

**Mini-Batch Gradient Descent** is a variant of Gradient Descent that splits the dataset into small subsets or "mini-batches". The algorithm then computes the gradient and updates the weights based on each of these mini-batches. It is a compromise between Gradient Descent and SGD.

Similar to SGD, we first shuffle the dataset, but then, we split the dataset into mini-batches of size $m$. For each mini-batch $B_k$, we compute the gradient of the average loss over the mini-batch with respect to the weights and update the weights as follows:

$$w := w - \alpha \nabla \left( \frac{1}{m} \sum_{i \in B_k} L_i(w) \right)$$

Here, $\alpha$ is the learning rate, and $\nabla \left( \frac{1}{m} \sum_{i \in B_k} L_i(w) \right)$ is the gradient of the average loss over the mini-batch with respect to the weights.

This update is performed for each mini-batch, and the process is repeated for a number of epochs until the algorithm converges or a stopping criterion is met.

The key difference with standard Gradient Descent and SGD is that the update is performed after each mini-batch (a subset of samples), not after the entire dataset (or one random sample in the case of SGD). This makes Mini-Batch Gradient Descent faster and more suitable for large datasets, while still providing more stable and less noisy updates than SGD.

## 4. Momentum

**Momentum** is a method that helps accelerate Stochastic Gradient Descent (SGD) in the relevant direction and dampens oscillations. It does this by adding a fraction of the update vector of the past time step to the current update vector.

The weights vector $w$ can be thought of as a particle traveling through parameter space. The particle accumulates acceleration from the gradient of the loss. This particle would keep moving in the same direction, instead of oscillating around the same stationary point. It can also escape a local minima if the momentum is high enough.

The update rule for the Momentum optimizer can be expressed as follows:

$$v := \beta v - \alpha \nabla L(w)$$

$$w := w + v$$

Here:

- $v$ is the velocity vector, which is an accumulation of the gradient elements.
- $\beta$ is the momentum term, which determines the amount of past gradients to retain. A typical value for $\beta$ is 0.9.
- $\alpha$ is the learning rate.
- $\nabla L(w)$ is the gradient of the loss function with respect to the weights.
- $w$ are the parameters of the model.

The momentum term $\beta$ is typically set to a value between 0 and 1, with a higher value resulting in a more stable optimization process. The larger the momentum term, the smoother the moving average, and the more resistant it is to changes in the gradients.

## 5. Adaptive Gradient Algorithm (Adagrad)

Adagrad is a gradient-based optimization algorithm that adapts the learning rate for each parameter during the optimization process, based on the past gradients observed for that parameter. It performs larger updates (high learning rates) for parameters related to infrequent features and smaller updates (low learning rates) for frequent ones. This makes it well-suited for dealing with sparse data.

The update rule for Adagrad is:

$$w := w - \frac{\alpha}{\sqrt{G + \epsilon}} \cdot \nabla L(w)$$

where $w$ are the parameters, $\alpha$ is the learning rate, $G$ is the sum of squares of past gradients, $\nabla L(w)$ is the gradient of the loss function, and $\epsilon$ is a small smoothing term to avoid division by zero (typically on the order of $1e-8$).

or written as per-parameter updates,

$$w_j := w_j - \frac{\alpha}{\sqrt{G_{j,j} + \epsilon}} \cdot g_j$$

$G_{j,j}$ is the sum of the squares of the past gradients with respect to $w_j$ up to the current time step. It's the $j$-th diagonal element of the matrix $G$.

$G$ is a diagonal matrix where each diagonal element $j, j$ is the sum of the squares of the gradients w.r.t. $w_j$ up to time step $t$, while all non-diagonal elements are 0. Here $w_j$ is the $j$-th parameter of the model.

Mathematically, each element in the diagonal $G_{j,j}$ is calculated as:

$$G_{j,j} = \sum_{\tau=1}^{t} g_{\tau,j}^2$$

where $g_{\tau,i}$ is the gradient at time step $\tau$ w.r.t. $w_j$.

The $G$ matrix is used to adaptively adjust the learning rates of the model parameters. The learning rate for each weight is scaled by the inverse square root of the sum of the squares of past gradients for that weight.

This means that weights associated with frequently occurring features (large gradients) get smaller updates, and weights associated with infrequent features (small gradients) get larger updates.

This adaptive learning rate makes Adagrad particularly useful for dealing with sparse data. However, the accumulation of the squared gradients in the denominator can cause the learning rate to shrink too much during training, which is a common criticism of Adagrad. Other algorithms such as Adadelta and Adam were proposed to overcome this issue.

## 6. Root Mean Square Propagation (RMSProp)

**RMSProp** is an optimization algorithm that adapts the learning rate for each parameter during the optimization process. It uses a decaying average of past squared gradients to adapt the step size for each parameter. This allows the algorithm to forget early gradients and focus on the most recently observed partial gradients.

The update rule for RMSProp is:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2$$

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

Here:

- $E[g^2]_t$ is the decaying average of past squared gradients.
- $\beta$ is the decay rate, which determines the rate at which past squared gradients are forgotten. A typical value for $\beta$ is 0.9.
- $g_t$ is the gradient at the current time step.
- $\alpha$ is the learning rate.
- $w$ are the parameters of the model.
- $\epsilon$ is a small smoothing term to avoid division by zero (typically on the order of $1e-8$).

To calculate $E[g^2]_t$, we take the square of each past gradient, and then compute a running (or moving) average of these squared gradients. The decay rate $\beta$ determines how much weight is given to the past gradients. A high value of $\beta$ means that more of the past gradients are taken into account, while a low value means that the algorithm "forgets" the past gradients more quickly.

## 7. Adaptive Moment Estimation (Adam)

**Adaptive Moment Estimation** (Adam) is an update to the RMSProp optimizer by combining it with the main feature of the Momentum method. It uses running averages of both the gradients and the second moments (squared gradients) of the gradients.

The update rule for Adam is as follows:

First, compute the running averages of the gradients (first moment) and the squared gradients (second moment):

#### Momentum Component

An estimate of the first moment (the mean) of the gradients, which is analogous to the momentum term in the Momentum method. This is represented by $m_t$ in the Adam update rule:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

Here, $m_{t-1}$ is the estimate of the first moment at the previous time step, $\beta_1$ is the decay rate for the first moment estimates (typically set to 0.9), and $g_t$ is the gradient at the current time step.

#### RMSProp Component

Next an estimate of the second moment (the uncentered variance) of the gradients, which is a key component of the RMSProp method. This is represented by $v_t$ in the Adam update rule:

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

Here, $v_{t-1}$ is the estimate of the second moment at the previous time step, $\beta_2$ is the decay rate for the second moment estimates (typically set to 0.999), and $g_t$ is the gradient at the current time step.

Then, correct the bias in the first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, update the parameters:

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

### 8. Summary

1. **Gradient Descent (GD)**: Uses the entire dataset to compute the gradient and update the weights. Can be slow for large datasets.
2. **Stochastic Gradient Descent (SGD)**: Updates weights using the gradient from a single randomly chosen data point. Faster and more efficient than GD, but updates can be noisy.
3. **Mini-Batch Gradient Descent**: A compromise between GD and SGD. Computes the gradient and updates weights based on a small random subset of data, leading to more stable updates than SGD.
4. **Momentum**: Accelerates SGD by adding a fraction of the past update vector to the current one, helping to dampen oscillations.
5. **Adagrad**: Adapts the learning rate for each parameter based on past gradients, performing larger updates for infrequent parameters and smaller ones for frequent parameters.
6. **RMSProp**: An extension of Adagrad that uses a moving average of squared gradients to normalize the gradient, making it well-suited for non-stationary objectives and problems with noisy or sparse gradients.
7. **Adam**: Combines RMSProp and Momentum by storing exponentially decaying averages of past gradients and squared gradients, making the method invariant to diagonal rescale of the gradients.

## Continual Learning and Test Production

### Continual Learning

**Continual Learning** (also known as Incremental Learning, Life-long Learning) is a method in which, input data is continuously used to extend the existing model's knowledge i.e. to further train the model. The aim of Continual Learning is for the learning model to adapt to new data without forgetting its existing knowledge.

Continual Learning (CL) is a blend of supervised unsupervised learning, frequently applied to data streams or big data such as stock trend prediction and user profiling, where new data becomes continuously available, or its size is out of system memory limits.

In contrast to traditional machine learning algorithms, which are trained on a fixed dataset and assume that the data distribution does not change, continual learning algorithms can continuously adapt and improve over time.

In CL, we aim to learn a sequence of tasks $\{T_1, T_2, \ldots, T_n\}$, where each task $T_i$ is defined by a specific data distribution $p_i(x, y)$. The goal is to improve or maintain the performance on all tasks as new tasks are encountered.

One of the significant challenges in CL is **catastrophic forgetting**, which can be mathematically described. If we denote the parameters of the model after learning task $T_i$ as $\theta_i$, and the loss function for task $T_i$ as $L_i(\theta)$, catastrophic forgetting can occur when:

$$L_i(\theta_{i+1}) > L_i(\theta_i)$$

This means the performance on task $T_i$ has worsened after learning task $T_{i+1}$.

To mitigate catastrophic forgetting, techniques like **regularization** and **memory-augmented networks** are used. Regularization adds constraints to the learning process to prevent overfitting to new data. A common method is **Elastic Weight Consolidation** (EWC), which adds a regularization term to the loss function:

$$L_{i+1}(\theta) = L_{i+1}^{new}(\theta) + \sum_j \frac{\lambda}{2} F_j(\theta_j - \theta_j^*)^2$$

Here,

- $L_{i+1}^{new}(\theta)$ is the loss on the new task.
- $\lambda$ is a hyperparameter controlling the importance of the regularization term. i.e how important the old task is compared to the new one.
- $F_j$ is the Fisher information matrix that measures the importance of parameter $\theta_j$ to the previous tasks.
- $\theta_j^*$ is the optimal parameter for the previous tasks.

Another approach is to use **memory-augmented networks**, which store some data or representations from previous tasks and use them when learning new tasks. This can be seen as a replay mechanism, where the model continues to learn from the previous tasks while learning the new task.

CL is used in many applications, such as anomaly detection, personalization, and forecasting, where it's crucial to adapt to new data while retaining knowledge from past data.

1. Anomaly detection: Continual learning helps constantly monitor a system's behavior to identify deviations or anomalies. This is particularly useful in sectors like finance, where transaction patterns evolve over time.

2. Personalization: Continual learning enables recommendation systems to adapt to a user's preferences and behavior over time, providing more accurate and personalized recommendations.

3. Forecasting: In forecasting, continual learning algorithms train on a stream of data, updating their understanding and predictions as new data becomes available. This is beneficial in areas like financial or time-series data forecasting.

## Test Production

In Machine Learning, **Test Production** is the process of ensuring that a deployed model operates correctly in a production environment. Model testing and evaluation are similar to what we call diagnosis and screening in medicine.

Benefits of model testing includes:

1. **Detecting Model and Data Drift**: Spot changes in model predictions and data over time.
2. **Finding Anomalies in Dataset**: Identify unusual observations that could impact model performance.
3. **Checking Data and Model Integrity**: Ensure data is accurate and consistent, and the model works as expected.
4. **Detect Possible Root Cause for Model Failure**: Identify reasons for model performance degradation or failure.
5. **Eliminating Bugs and Errors**: Identify and fix issues in the data preprocessing steps, learning algorithm, or prediction pipeline.
6. **Reducing False Positives and False Negatives**: Minimize these errors in classification tasks through regular testing and tweaking of the model.
7. **Encouraging Retraining the Model Over a Certain Period of Time**: Determine optimal retraining frequency to maintain model performance.
8. **Creating a Production-Ready Model**: Ensure the model can handle real-world data and use cases, and perform reliably in a production environment.
9. **Ensuring Robustness of ML Model**: Test the model's performance on unseen data and under different conditions to ensure it's robust.

10. **Finding New Insights Within the Model**: Gain insights about important features, the model's decision-making process, and novel patterns in the data.

Key steps involved in testing production in machine learning are:

1. **Validating Input Data**: Check the input data for correct format, range, and consistency. Handle missing values appropriately.
2. **Monitoring Model Performance**: Keep track of the model's performance metrics (like accuracy, precision, recall) and system-level metrics (like memory usage, latency).
3. **Detecting Model and Data Drift**: Monitor for changes in the data the model is processing in production compared to the data it was trained on. Adjust the model as necessary to maintain performance.
4. **Testing Integration Between Pipeline Components**: Ensure that the model works correctly as part of a larger system or pipeline.
5. **Model Retraining**: Periodically retrain the model with new data to prevent performance degradation.
6. **A/B Testing**: Test two versions of the model in production simultaneously to determine which one performs better.
7. **Testing Quality of Live Model on Served Data**: Verify that the model's behavior is correct both before training (with pre-train tests) and after training (with post-train tests).

Sources:


1. Wikipedia - Gradient
2. Wikipedia - Stationary point
3. Wikipedia - Gradient Descent
4. Wikipedia - Stochastic gradient descent
5. Baeldung - Differences Between Gradient, Stochastic and Mini Batch Gradient Descent
6. Analytics Vidhya - Why use the momentum optimizer with minimal code example
7. databricks - adagrad
8. OpenGenus IQ - adagrad
9. Ruder, Sebastian - An overview of gradient descent optimization algorithms
10. Duchi, John, Elad Hazan, and Yoram Singer - Adaptive subgradient methods for online learning and stochastic optimization
11. OpenGenus IQ - RMSprop
12. Papers with code - RMSProp
13. Papers with code - Continual Learning
14. Wikipedia - Incremental learning
15. iMerit - A complete introduction to continual learning
16. James Kirkpatrick et al. - Overcoming catastrophic forgetting in neural networks
17. neptune.ai - 5 Tools That Will Help You Setup Production ML Model Testing
18. Google - Testing Pipelines in Production
19. Nanonets - Health Checks for Machine Learning - A Guide to Model Retraining and Evaluation
20. ApplyingML - Machine Learning in Production - Testing