Lecture 3

Optimization Algorithms

In machine learning, we employ a number of optimization algorithms to ensure that parameters for our models converge to optimal values by minimizing (or maximizing) some objective function. For example, in regression, we typically want to minimize the sum of square errors.

What is Optimization?
Optimization is the process of finding the best parameters for a model to minimize (or maximize) an objective function. In machine learning, the objective function often measures error (like Mean Squared Error in regression) that we want to minimize.

Why is Optimization Important?
Determines the performance of the machine learning model. Efficient optimization leads to faster convergence and better results.

Commonly Used Optimization Algorithms:

- Gradient Descent (and its variants)
- Stochastic Gradient Descent (SGD)
- Mini-Batch Gradient Descent
- Newton's Method
- Adam Optimizer

Gradient Descent
General framework:

$$\text{Repeat until converge } \{$$
$$\text{w} = \text{w} - \alpha \left[\frac{\partial Loss}{\partial w}\right]$$
$$\text{b} = \text{b} - \alpha \left[\frac{\partial Loss}{\partial b}\right]$$
$$\}$$

Example for ordinary least squares (single variable) linear regression:

$$RMSE = ((1/n)sum(y - y\text{^}2)\text{^}2)^{-2}$$
$$y = mx + b$$

Getting our first-order derivative for this function would be

$$\frac{\partial}{\partial \boldsymbol{m}} = \frac{2}{\boldsymbol{n}} \sum_{i=1}^{n} -\boldsymbol{x_i}(\boldsymbol{y_i} - (\boldsymbol{mx_i} + \boldsymbol{b}))$$
$$\frac{\partial}{\partial \boldsymbol{b}} = \frac{2}{\boldsymbol{n}} \sum_{i=1}^{n} -(\boldsymbol{y_i} - (\boldsymbol{mx_i} + \boldsymbol{b}))$$

Gradient Descent is an iterative optimization algorithm used to find the minimum of a function. It updates parameters by moving in the opposite direction of the gradient of the objective function. The gradient is a concept from multivariable calculus that is a vector of the partial derivatives of the objective function to be optimized.

$$\theta_{new} = \theta_{old} - \alpha \times \nabla J(\theta)$$

Where:

- θ are the parameters,

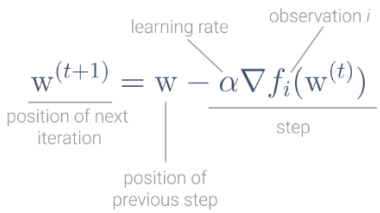
- α is the learning rate,
- $\nabla J(\theta)$ is the gradient of the objective function.

```
# Objective Function: Mean Squared Error
mse <- function(y, y_pred) {
  mean((y - y_pred)^2)
}

# Gradient Descent Function
gradient_descent <- function(x, y, learning_rate = 0.01, n_iter = 1000) {
  m <- length(y)
  theta <- 0  # Start with an initial value of 0 for theta

  for (i in 1:n_iter) {
    y_pred <- theta * x
    gradient <- -(2/m) * sum(x * (y - y_pred))
    theta <- theta - learning_rate * gradient
  }

  return(theta)
}

# Example data
x <- 1:10
y <- 2 * x + 1  # Linear relationship with some noise

# Apply Gradient Descent
theta_optimal <- gradient_descent(x, y)
print(theta_optimal)
```

The learning rate α controls the size of steps. If it's too large, the algorithm might overshoot; if too small, convergence will be slow. Depending on the problem, you may need to adjust α manually to obtain the best results.

Stochastic Gradient Descent

Instead of using the entire dataset to compute the gradient, SGD uses only one data point at a time, which speeds up computation but introduces noise into the optimization process.



$$\theta_{new} = \theta_{old} - \alpha \times \nabla J(\theta; x_i, y_i)$$

Where (x_i, y_i) is a single data point.

```
# Stochastic Gradient Descent Function
stochastic_gradient_descent <- function(x, y, learning_rate = 0.01, n_iter = 100) {
  m <- length(y)
  theta <- 0  # Start with an initial value of 0 for theta

  for (i in 1:n_iter) {
    for (j in 1:m) {
      y_pred <- theta * x[j]
      gradient <- -(2/m) * x[j] * (y[j] - y_pred)
      theta <- theta - learning_rate * gradient
    }
  }

  return(theta)
}

# Apply Stochastic Gradient Descent
theta_sgd <- stochastic_gradient_descent(x, y)
print(theta_sgd)
```

SGD is faster but more volatile than batch Gradient Descent. It might take more iterations to converge to the minimum. You may be able to reduce the volatility by including a decaying learning rate (one that gets smaller as the function iterates).

Mini-Batch Gradient Descent
Combines the advantages of both Batch Gradient Descent and Stochastic Gradient Descent by using a small batch of data points in each iteration.

```
# Mini-Batch Gradient Descent Function
mini_batch_gradient_descent <- function(x, y, learning_rate = 0.01, n_iter = 100, batch_size = 3)
{
  m <- length(y)
  theta <- 0  # Start with an initial value of 0 for theta

  for (i in 1:n_iter) {
    indices <- sample(1:m, batch_size)
    x_batch <- x[indices]
    y_batch <- y[indices]

    y_pred <- theta * x_batch
    gradient <- -(2/batch_size) * sum(x_batch * (y_batch - y_pred))
    theta <- theta - learning_rate * gradient
  }

  return(theta)
}

# Apply Mini-Batch Gradient Descent
```

```
theta_mini_batch <- mini_batch_gradient_descent(x, y)
print(theta_mini_batch)
```

Newton's Method

This is a method that students learn in Calculus to find the zero of a function (or an intersection). Since optimization involves finding the zero of the derivative(s) of an objective function, we can use this method here as well, with some caveats (Newton's doesn't always converge, for example). Since we are finding the zero of the derivative, we'll need the derivative of that. In multivariable calculus, since the gradient is a vector, the second derivative is a matrix called the Hessian.

An optimization technique that uses the second derivative (Hessian) of the objective function to adjust parameters. Converges faster but is computationally expensive.

$$\theta_{new} = \theta_{old} - \frac{J'(\theta)}{J''(\theta)}$$

```
# Newton's Method for a quadratic function
newtons_method <- function(x, y, n_iter = 10) {
  theta <- 0  # Initial guess
  for (i in 1:n_iter) {
    grad <- -2 * sum(x * (y - theta * x))
    hessian <- 2 * sum(x^2)
    theta <- theta - grad / hessian
  }
  return(theta)
}

# Apply Newton's Method
theta_newton <- newtons_method(x, y)
print(theta_newton)
```

Newton's Method converges quickly for convex functions, but the computation of the Hessian can be prohibitive in high dimensions.

Adam Optimizer

An advanced optimization algorithm that combines the benefits of AdaGrad and RMSProp. It adjusts the learning rate based on first and second moments of gradients.

Step 1: while w_t do not converges

do{

Step 2: Calculate gradient $g_t = \frac{\partial f(x,w)}{\partial w}$

Step 3: Calculate $p_t = m_1 \cdot p_{t-1} + (1 - m_1) \cdot g_t$

Step 4: Calculate $q_t = m_2 \cdot q_{t-1} + (1 - m_2) \cdot g_t^2$

Step 5: Calculate $\widehat{p_t} = p_t / (1 - m_1^t)$

Step 6: Calculate $\widehat{q_t} = q_t / (1 - m_2^t)$

Step 7: Update the parameter $w_t = w_{t-1} - \alpha \cdot \widehat{p_t} / (\sqrt{\widehat{q_t}} + \epsilon)$

}

Step 8: return w_t

$$For\ each\ Parameter\ w^j$$
$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$
$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$
$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$
$\beta_1, \beta_2 : Hyperparameters$

```
# Adam Optimizer (simplified version)
adam_optimizer <- function(x, y, learning_rate = 0.01, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8,
n_iter = 1000) {
  m <- length(y)
  theta <- 0
  m_t <- 0
  v_t <- 0
  for (t in 1:n_iter) {
    y_pred <- theta * x
    grad <- -(2/m) * sum(x * (y - y_pred))

    m_t <- beta1 * m_t + (1 - beta1) * grad
    v_t <- beta2 * v_t + (1 - beta2) * grad^2

    m_t_hat <- m_t / (1 - beta1^t)
    v_t_hat <- v_t / (1 - beta2^t)

    theta <- theta - learning_rate * m_t_hat / (sqrt(v_t_hat) + epsilon)
  }

  return(theta)
}

# Apply Adam Optimizer
theta_adam <- adam_optimizer(x, y)
print(theta_adam)
```

Adam is widely used due to its adaptability and efficiency in handling noisy gradients. It's especially popular in neural network models.

Customizing Algorithms

Different problems may require tweaking of optimization algorithms to improve performance. Customizing learning rates, batch sizes, or even hybridizing algorithms can yield better results.

- *Learning Rate Schedules:* Implement a decaying learning rate to prevent overshooting as the algorithm converges.
- *Hybrid Approaches:* Combine SGD with momentum or Adam with a decaying learning rate.

```
# Custom Gradient Descent with a Decaying Learning Rate + check
gradient_descent_decay <- function(x, y, initial_lr = 0.1, decay = 0.001, n_iter = 1000) {
  m <- length(y)
  theta <- 0  # Start with an initial value of 0 for theta

  for (i in 1:n_iter) {
    y_pred <- theta * x
    gradient <- -(2/m) * sum(x * (y - y_pred))

    # Update learning rate
    learning_rate <- initial_lr / (1 + decay * i)

    # Update theta
    theta <- theta - learning_rate * gradient

    # Print theta to debug if it goes to NaN
    if (is.nan(theta)) {
      print(paste("Theta became NaN at iteration:", i))
      break
    }
  }

  return(theta)
}

# Example data
x <- 1:10
y <- 2 * x + 1  # Linear relationship

# Apply Custom Gradient Descent
theta_custom <- gradient_descent_decay(x, y, initial_lr = 0.01, decay = 0.0001)
print(theta_custom)
```

If you run this code using the default learning rate and decay rate, this example will go to NaN around 420 iterations in. If this happens in general, adjust the learning and/or decay rate until the NaN stops happening.

Let's look at some more complex examples with multivariable regression.

```
data(mtcars)
mtcars_data <- mtcars[, c("mpg", "disp", "hp", "wt", "qsec", "drat")]  # Selecting 5 variables + mpg

# Normalizing the data for better convergence
```

```
normalize <- function(x) {
  return((x - mean(x)) / sd(x))
}

mtcars_data <- as.data.frame(lapply(mtcars_data, normalize))

# Separate predictors and target
x <- as.matrix(mtcars_data[, -1])
y <- mtcars_data$mpg

# Add a column of ones for the intercept
x <- cbind(1, x)
```

This last step here will get explained more when we discuss the normal equation next week.

```
gradient_descent <- function(x, y, learning_rate = 0.01, n_iter = 1000) {
  m <- nrow(x)
  theta <- rep(0, ncol(x))  # Initialize theta

  for (i in 1:n_iter) {
    y_pred <- x %*% theta
    gradient <- -(2/m) * t(x) %*% (y - y_pred)
    theta <- theta - learning_rate * gradient
  }

  return(theta)
}

theta_gd <- gradient_descent(x, y)
print(theta_gd)

stochastic_gradient_descent <- function(x, y, learning_rate = 0.01, n_iter = 100) {
  m <- nrow(x)
  theta <- rep(0, ncol(x))  # Initialize theta

  for (i in 1:n_iter) {
    for (j in 1:m) {
      idx <- sample(1:m, 1)
      x_i <- x[idx, , drop = FALSE]
      y_i <- y[idx]

      y_pred <- x_i %*% theta
      gradient <- -(2) * t(x_i) %*% (y_i - y_pred)
      theta <- theta - learning_rate * gradient
    }
  }

  return(theta)
```

```
}

theta_sgd <- stochastic_gradient_descent(x, y)
print(theta_sgd)

mini_batch_gradient_descent <- function(x, y, learning_rate = 0.01, n_iter = 100, batch_size = 5)
{
  m <- nrow(x)
  theta <- rep(0, ncol(x))  # Initialize theta

  for (i in 1:n_iter) {
    indices <- sample(1:m, batch_size)
    x_batch <- x[indices, , drop = FALSE]
    y_batch <- y[indices]

    y_pred <- x_batch %*% theta
    gradient <- -(2/batch_size) * t(x_batch) %*% (y_batch - y_pred)
    theta <- theta - learning_rate * gradient
  }

  return(theta)
}

theta_mbgd <- mini_batch_gradient_descent(x, y)
print(theta_mbgd)

newtons_method <- function(x, y, n_iter = 10) {
  m <- nrow(x)
  theta <- rep(0, ncol(x))  # Initialize theta

  for (i in 1:n_iter) {
    y_pred <- x %*% theta
    gradient <- -(2/m) * t(x) %*% (y - y_pred)
    hessian <- (2/m) * t(x) %*% x

    theta <- theta - solve(hessian) %*% gradient
  }

  return(theta)
}

theta_newton <- newtons_method(x, y)
print(theta_newton)

adam_optimizer <- function(x, y, learning_rate = 0.01, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8,
n_iter = 1000) {
  m <- nrow(x)
  theta <- rep(0, ncol(x))  # Initialize theta
```

```
  m_t <- rep(0, ncol(x))
  v_t <- rep(0, ncol(x))

  for (t in 1:n_iter) {
   y_pred <- x %*% theta
   gradient <- -(2/m) * t(x) %*% (y - y_pred)

   m_t <- beta1 * m_t + (1 - beta1) * gradient
   v_t <- beta2 * v_t + (1 - beta2) * (gradient^2)

   m_t_hat <- m_t / (1 - beta1^t)
   v_t_hat <- v_t / (1 - beta2^t)

   theta <- theta - learning_rate * m_t_hat / (sqrt(v_t_hat) + epsilon)
  }

  return(theta)
}

theta_adam <- adam_optimizer(x, y)
print(theta_adam)
```

Comparison of Algorithms:
Each algorithm has strengths and weaknesses. Gradient Descent is reliable but can be slow. SGD is faster but noisy. Mini-Batch offers a balance. Newton's method is fast but computationally expensive. Adam combines the best of several approaches but requires tuning.

Convergence:
You may notice that different algorithms produce slightly different coefficients due to the way they converge. This is especially true for algorithms like SGD and Adam that use randomness.

Customization:
You can tweak learning rates, batch sizes, and other parameters to optimize performance for your specific dataset.

Error Handling:
Numerical instability (e.g., `NaN` values) can occur if learning rates are too high or if the data isn't properly normalized. Normalization and regularization can help address this.

Resources:

1. https://towardsdatascience.com/understanding-optimization-algorithms-in-machine-learning-edfdb4df766b
2. https://vtantravahi.medium.com/math-behind-optimization-techniques-9c3b200d9cca
3. https://www.linkedin.com/pulse/optimization-machine-learning-comprehensive-guide-using-koraichi-onr3e/
4. https://www.kdnuggets.com/2018/05/optimization-using-r.html