GRADIENT DESCENT — In deep learning, the task often boils down to optimizing a complex, non-linear function that measures how well the model performs a given task. The function, known as the loss function L, quantifies the difference between the model's predictions and the actual target values. The goal is the find the parameters θ that minimize $\mathcal{L}(\theta)$.

Why is it so important to find an efficient algorithm for this? The dimensions of θ can be extremely high, typically in the millions or billions for deep neural networks. Without an efficient algorithm, the training would be computationally infeasible, making it impossible to train on large datasets or complicated models. Gradient descent, along with its various derivatives, is one of the most fundamental yet extremely powerful tool for such optimization tasks. The simplicity and effectiveness of gradient descent made it the go-to algorithm for deep learning. The idea is to update each parameter in the direction opposite to the gradient $\nabla\mathcal{L}$ of the loss function at the current point:

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \alpha\nabla\mathcal{L}(\theta_{\text{old}})$$

where α is the learning rate governing the step size. The learning rate can have a significant impact on the quality and speed of the training process: if the learning rate is too high, the network may overshoot the minimum and diverge; if the learning rate is too low, the network may be stuck at local minimums, or converge very slowly. Despite its simplicity, gradient descent has proved to be effective in solving a wide variety of tasks, ranging from image recognition, natural language processing to voice recognition. Next, we introduce some improved versions of vanilla gradient descent:

1. Stochastic gradient descent (SGD): instead of computing the gradient based on the entire dataset, SGD computes the gradient using a randomly sampled training example. The frequent updates can lead to faster convergence, and the noisy updates can help escape from local minima. However, the noisy updates can also lead to high variances and unstable convergence.
2. Mini-batch gradient descent: combines the advantages of SGD and vanilla gradient descent by computing the gradient of a subset of training samples. This method makes it easier to utilize parallel processors such as GPUs. Mini-batch GD is commonly combined with batch normalization, which accelerates and stabilizes training by mitigating internal covariate shift.
3. Momentum: instead of updating the weights solely based on the current gradient, momentum considers the past gradients to smooth out the update. This typically enables faster convergence and helps overcome local minimas or saddle points. Mathematically, it can be described as:

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot \nabla\mathcal{L}(\theta)$$

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \alpha \cdot v_t$$

where v_t denotes the velocity at time t (v_0 is typically set to 0), and β denotes the momentum coefficient.

4. Changing learning rates: instead of using a fixed learning rate for the entire training, we can adjust the learning rate accordingly. Popular methods include Adagrad (adjusts the learning rate for each parameter based on historical gradients, such that parameters with frequent large updates get a reduced learning rate, and those with small updates get an increased rate), RMSprop (uses a moving average of past squared gradients to address Adagrad's issue of vanishing learning rate), Adam (combines momentum and RMSprop), cyclic (uses an oscillating learning rate, such that the high rate enables faster convergence and escape from non-optimal points, and low rate refines the solution) and decreasing learning rate (starts with a high learning rate for faster convergence, and a decaying rate to stabilize learning).

Newton's method in optimization — Gradient descent operates primarily on first-order derivative information for its updates, potentially resulting in slow convergence, particularly in gradient-free zones like saddle points and local minima. When we delve into high-dimensional spaces, random Gaussian theory suggests that the prevalence of saddle points compared to local minima increases exponentially. To understand this, consider a critical point in a $\mathcal{D}$-dimensional space. The Hessian matrix at this point will have $\mathcal{D}$ eigenvalues. For the point to be a local minimum, all these eigenvalues must be positive. In contrast, a saddle point requires the Hessian to exhibit both positive and negative eigenvalues, indicating regions where the function's curvature varies. If we assume that any given eigenvalue is equally likely to be positive or negative, then saddle points become exponentially more common as the dimensionality increases. Given this characteristic landscape, there's a growing need for optimization methods that are robust against saddle points. This is where Newton's method shines. By incorporating second-order derivative information, it harnesses both gradient and curvature data to inform its updates, offering a more robust approach in high-dimensional contexts. More formally,

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \mathcal{H}(\theta_{\text{old}})^{-1} \nabla \mathcal{L}(\theta_{\text{old}})$$

where $\mathcal{H}(\theta)$ denotes the Hessian matrix at θ. To understand how Newton's method work, let's consider the first three terms of the Taylor expansion of the loss function at θ_0:

$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta_0) + \nabla \mathcal{L}(\theta_0)^T (\theta - \theta_0) + \frac{1}{2} (\theta - \theta_0)^T \mathcal{H}(\theta_0)(\theta - \theta_0)$$

The minimum of this local approximation can be found by setting its gradient to zero:

$$\nabla \mathcal{L}(\theta_0) + \mathcal{H}(\theta_0)(\theta - \theta_0) = 0$$

where we yield the update as shown above. Observe that if we only consider the first two terms of the Taylor expansion (excluding the quadratic term), we simply get:

$$\nabla \mathcal{L}(\theta_0) = 0$$

which yields the vanilla gradient descent. Note that for Newton's Method, the term $\mathcal{H}(\theta_{old})^{-1}$ acts as an adaptive step size as it provides information of how steep of flat the function is: when the curvature is large, the effective step size is small, and vice versa. By taking into consideration of the curvature magnitude and direction, Newton's method is an effective optimization method that can lead to faster convergence without externally setting the learning rate.

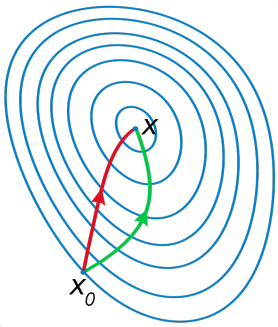

Figure 25: Comparing gradient descent (green) with Newton's method (red) for minimizing the loss, we see that Newton's method leverages curvature information to take a more direct path. Source: Wikipedia

Despite its powerful convergence properties, Newton's method faces practical challenges that made it less popular than first-order methods such as gradient descent. For deep neural networks with millions to billions of parameters, computing and inverting the Hessian matrix becomes computationally infeasible, both in terms of time and space complexity. While considering additional terms in the Taylor expansion may enable the network to converge in less steps, the computational overhead for each added term makes it impossible to scale for large models.

Back-Propagation — Back-propagation is an optimization method used for reducing the error in the predictions of a neural network. The algorithm enables efficient computation of gradients, and adjusts the weights and biases in a way that minimizes the difference between the predicted value and the actual output. Here, we dive into how neural networks are trained.

The training of feed-forward neural networks consists of two components: the forward phase and the back-propagation phase. In the forward phase, the network takes in the input data and processes it sequentially through its various layers. This begins at the input layer, where the data is initially fed. As data moves through the network, each neuron applies a weighted sum of its inputs, adds a bias, and then passes this result through an activation function to produce its output. This process repeats layer by layer until the data reaches the final layer, usually known as the output layer. Here, the network provides its prediction for the given input, which is an educated guess based on its current weights and biases. More formally, consider a fully connected layer with an ReLU activation function. The output y pf the layer can be represented as:

$$y = R(W \cdot x + b)$$

where W denotes the weight matrix, x is the input to the layer, b is the bias, and

$R(x) \triangleq \max(0, x)$ is the ReLU activation function.

Next, the training enters the back-propagation phase, which is the stage where the network "learns" from the given outputs. Consider the last layer, where x is the input to the layer, y is the output after the activation function, and y^* is the target output. Let $L(y, y^*)$ denote the loss function. The gradient of the loss with respect to the inputs can be computed as:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x}$$

The gradient provides information on how much the loss would change if we adjust the inputs, and these adjustments are propagated back through the network, layer by layer, to adjust the weights and biases of the network.

The training of a neural network alternates between forward and backward phases: during forward pass, the network predicts the input data using the current weights and biases. In the backward pass, the network assesses the prediction errors and adjusts its weights and biases to minimize the loss. One complete cycle of prediction and correction is called one "epoch", and thousands or millions of epochs are typically required before a network converges.