Gradient Descent, and Stochastic Gradient Descent (SGD)

Artificial Deep Neural Networks

Arman Afrasiyabi
Department of Neurosurgery
Yale University

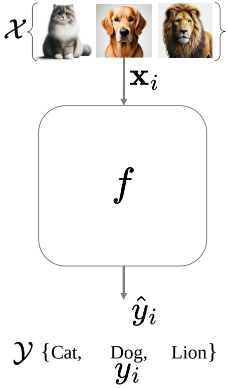
Machine learning

Given a **dataset** $\mathcal{D}$, we aim to find the best function f from a **model class** $\mathcal{H}$.

- **Model class**: A set of functions

$$\mathcal{H} = \{f(\mathbf{x}_i; \mathbf{w}) \mid \mathbf{w} \in \mathcal{W}\}$$

where each $f : \mathcal{X} \to \mathcal{Y}$ is parameterized by $\mathbf{w}$, and $\mathcal{X}$ and $\mathcal{Y}$ represent the input and output spaces respectively.



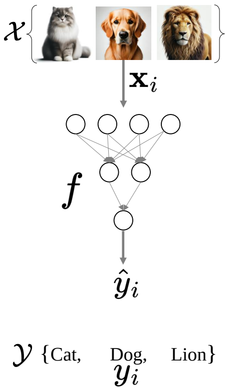
Deep learning

In the context of **deep learning**, f represents a type of neural network, such as an MLP (multilayer perceptron).

- Let's ℓ be a function measures the discrepancy between a prediction $\hat{y}_i = f(\mathbf{x}_i; \mathbf{w})$ and the true labels y_i.
- **Loss/Cost function**, L is an aggregate measure of the total loss over the entire training set $\mathcal{D}$:

$$L(\mathbf{w}; \mathcal{D}) = \frac{1}{n} \sum_i \ell(\hat{y}_i, \ y_i).$$

Here, n is the number of training examples in dataset $\mathcal{D}$, and $\mathbf{w}$ represents the parameters of the model.



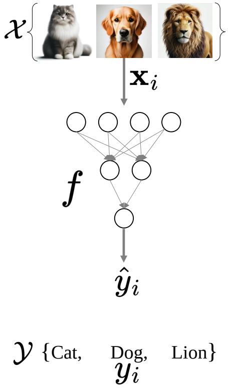
Deep learning

Training involves finding $\mathbf{w}^*$ that minimizes the loss on the training set, while the

- Goal is to find parameters that minimize the loss function:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \left(\sum_i \ell(f(\mathbf{x}_i; \mathbf{w}), y_i) + \lambda \sum_{j=1} w_j^2 \right)$$

- The loss function may also include regularization.
- Using the **validation** set, we evaluate the model while tuning the parameters of f
- Using **test** set, we conduct the final performance evaluation.



Learning and optimization

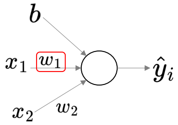
- **Objective**: Minimize a loss function to find the best model parameters.
- **"Learning"**: refers to the iterative process of adjusting model parameters to improve predictions based on training data.
  - It's crucial that the model not only performs well on training data but also generalizes to unseen test data.
- **Optimization in Learning**: Learning involves using optimization methods to minimize the loss function with respect to the model parameters.
- **Challenges**: Functions modeled by neural networks are typically non-linear and non-convex, making them complex for optimization.

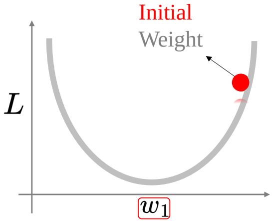
Loss function optimization

Let's define the mean squared error (MSE) as a criterion for measuring the discrepancy between the model predictions $\hat{y}_i$ and the ground truth y_i.

$$L(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_i (\hat{y}_i - y_i)^2$$

- $\mathbf{w}$ is the collection of all weights in the network.
- $\mathbf{b}$ is the collection of all biases in the network.



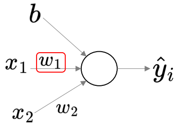


Goal: develop an algorithm to find best weights ($\mathbf{w}$) and biases (b) which minimizes $L(\mathbf{w}, \mathbf{b})$.

MSE Loss function

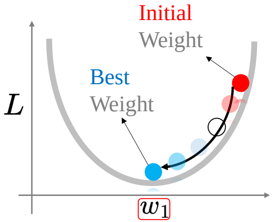
When, the loss function L approaches zero,1
$\hat{y}_i = f(\mathbf{x}_i; \mathbf{w}, b) \approx y_i$ for all training inputs. This indicates:

- The algorithm has performed well if it can find weights and biases such that $L(\mathbf{w}, b) \approx 0$.



Goal of the training algorithm:

- Choose $\mathbf{w}$ and b to minimize $L(\mathbf{w}, b)$.



Optimization Method:

- We'll use Gradient Descent.

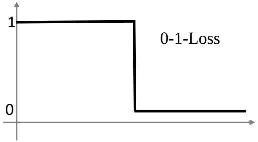
MSE Loss function

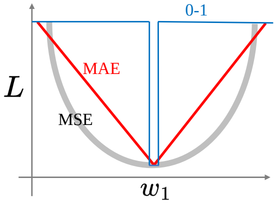
Smoothness and Differentiability:

- MSE is smooth and differentiable w.r.t. model parameters
- 0-1 Loss is non-differentiable and not smooth

Sensitivity to Parameter Changes:

- MSE change continuously as weights and biases are adjusted, providing incremental improvements and detailed feedback.
- 0-1 Loss changes only when a misclassification is corrected, offering less frequent updates.

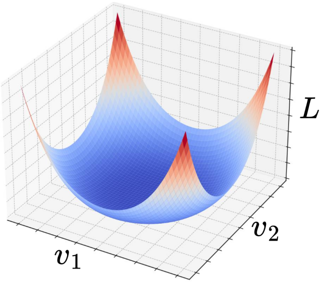




Gradient descent

How do we minimize a loss function in general?

Suppose we want to minimize some function $L(\mathbf{v})$ where $\mathbf{v} = (v_1, v_2, \ldots)$.

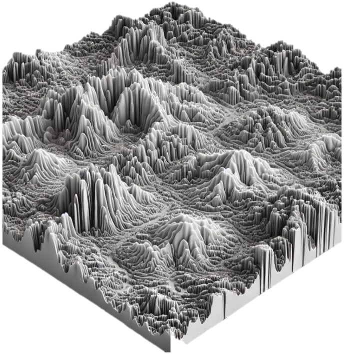


Gradient descent

What if we have more variables?

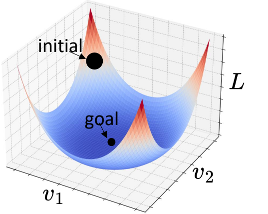
We could try calculus to find the extremum of L:

- Difficult when we have lots of variables.
- Largest neural networks have billions of weights and biases, complicating the calculus approach.

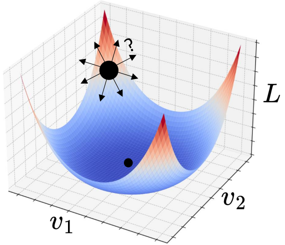


Big picture: gradient descent

- Think of $L(\mathbf{v})$ as a **height map** over the plane of parameters $\mathbf{v} = (v_1, v_2)^\top$.
- We pick a random starting point and *roll a ball* downhill to a valley (a minimum of L).



- **Question:** In which direction should we move to change L the most per unit step?
- Let $\hat{\mathbf{r}}$ be a **unit** direction ($||\hat{\mathbf{r}}|| = 1$). We want the $\hat{\mathbf{r}}$ that maximizes the rate of change of L.



What are we trying to do? (Setup for steepest change)

- The **gradient** $\nabla L(\mathbf{v})$ is an arrow that points *straight uphill* (the direction of biggest increase of height). Formally, $L(\mathbf{v})$ is a function of several variables $(v_1, v_2, \ldots, v_n)$, the gradient is:

  $$\nabla L(\mathbf{v}) = \left[\frac{\partial L}{\partial v_1}, \frac{\partial L}{\partial v_2}, \ldots, \frac{\partial L}{\partial v_n}\right]^T.$$

  It is a collection of all the partial derivatives — one for each variable.

  Analogy: You're on a hill with a compass. The compass needle ∇L points uphill. Walking exactly with the needle is fastest uphill; walking exactly opposite is fastest downhill.

What are we trying to do? (Setup for steepest change)

We also choose a **direction to move** given by a *unit* arrow

$$\|\hat{\mathbf{r}}\| = 1$$

Think of $\hat{\mathbf{r}}$ as "which way we step next."
The reason we introduce the direction vector $\hat{\mathbf{r}}$ is because it is the formal way of asking:

"If I take a small step, which way should I go to change the loss the most?"

The gradient $\nabla L(\mathbf{v})$ does not let you compare all possible directions under a fixed step length.

How fast does L change if we step in a direction?

Directional derivative of the loss $D_{\hat{\mathbf{r}}} L(\mathbf{v})$: The *local* change rate of L at $\mathbf{v}$ when moving an infinitesimal step in the unit direction $\hat{\mathbf{r}}$ is

$$
\begin{aligned}
D_{\hat{\mathbf{r}}} L(\mathbf{v}) &= \nabla L(\mathbf{v})^\top \hat{\mathbf{r}} \\
&= \|\nabla L(\mathbf{v})\| \, \|\hat{\mathbf{r}}\| \, \cos\phi \\
&= \|\nabla L(\mathbf{v})\| \cos\phi
\end{aligned}
$$

where ϕ is the angle between ∇L and $\hat{\mathbf{r}}$.

Dot product:
Algebraic form: $\mathbf{a}^\top \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$
Geometric form: $\mathbf{a}^\top \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$
where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

How fast does L change if we step in a direction?

- $\phi = 0°$ (same) $\Rightarrow \cos\phi = 1$: largest *positive* change.
- $\phi = 90°$ (perpendicular) $\Rightarrow \cos\phi = 0$: no instantaneous change.
- $\phi = 180°$ (opposite) $\Rightarrow \cos\phi = -1$: largest *negative* change.

Which direction maximizes change? (Set-up)

Goal (unit step constraint):

$$\arg\max_{\|\hat{\mathbf{r}}\|=1} D_{\hat{\mathbf{r}}} L(\mathbf{v}) \qquad \text{(choose a unit direction that increases } L \text{ the most)}$$

Directional derivative:

$$D_{\hat{\mathbf{r}}} L(\mathbf{v}) = \nabla L(\mathbf{v})^\top \hat{\mathbf{r}}.$$

Dot product identity (vector geometry):

$$\nabla L(\mathbf{v})^\top \hat{\mathbf{r}} = \|\nabla L(\mathbf{v})\| \, \|\hat{\mathbf{r}}\| \cos\phi = \|\nabla L(\mathbf{v})\| \cos\phi \quad (\|\hat{\mathbf{r}}\| = 1).$$

Key observation: Maximizing $\nabla L^\top \hat{\mathbf{r}}$ over unit vectors is the same as maximizing $\cos\phi$.

Linear-algebra

Having $\mathbf{a}^\top \mathbf{b} = c$, writing $\mathbf{b}$ in terms of $\mathbf{a}$ and c

Assuming $c \in \mathbb{R}$ and $\mathbf{a} \neq 0$, the **minimum-norm** solution is

$$\boxed{\mathbf{b} = \frac{c}{\|\mathbf{a}\|^2}\,\mathbf{a}}$$

Result & intuition: steepest up/down directions

$$\nabla L(\mathbf{v})^\top \hat{\mathbf{r}} = \|\nabla L(\mathbf{v})\|,$$

$$\hat{\mathbf{r}} = \frac{\nabla L(\mathbf{v})}{\|\nabla L(\mathbf{v})\|^2}\|\nabla L(\mathbf{v})\|$$

Steepest increase (uphill): (same direction as the gradient, unit length)

$$\boxed{\hat{\mathbf{r}}_{\text{up}} = \frac{\nabla L(\mathbf{v})}{\|\nabla L(\mathbf{v})\|}}$$

Steepest decrease (downhill): (opposite direction as the gradient, unit length)

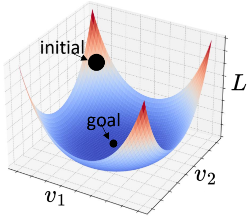
$$\boxed{\hat{\mathbf{r}}_{\text{down}} = -\frac{\nabla L(\mathbf{v})}{\|\nabla L(\mathbf{v})\|}}.$$

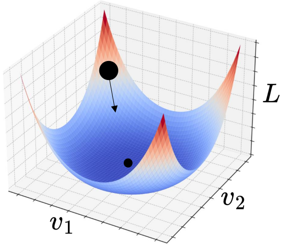
From direction to a practical step

- In practice, we absorb the normalization into the step size and use the classic form

$$\boxed{\Delta\mathbf{v} \;=\; -\,\alpha\,\nabla L(\mathbf{v})} \qquad (\alpha > 0).$$

- *Analogy:* α is how big a **stride** you take each step. Too big: you overshoot; too small: you crawl.





The update rule (first-order guarantee)

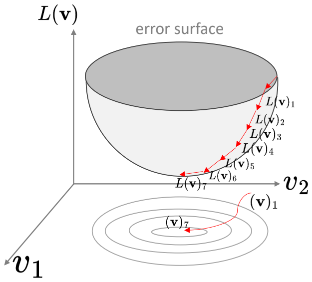
- Standard gradient descent update:

$$\boxed{\mathbf{v}_{t+1} = \mathbf{v}_t - \alpha_t \nabla L(\mathbf{v}_t)}.$$

- *Notes students find helpful:*
  - α_t (learning rate) may be constant or scheduled.
  - If α_t is too large, the Taylor approximation breaks and the loss can *increase*.
  - Local minima/saddle points can slow progress — momentum/Adam help, but the basic direction is still $-\nabla L$.

Summary of gradient descent

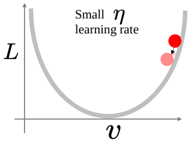
- Compute the gradient ∇L.
- Move in the **opposite direction**:
  - It's like falling down the slope of the valley.

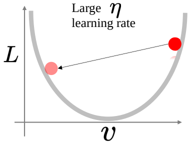


Adapted from https://allmodelsarewrong.github.io/images/ols/gradient-error-surface3.svg

Choosing the learning rate

- Need to choose η small enough that the approximation $\Delta L \approx -\eta \nabla L \cdot \nabla L$ is good.
    - Otherwise, we may end up with $\Delta L > 0$.
- On the other hand, very small η implies tiny steps:
    - Slow convergence to the minimum.





Gradient descent in neural networks

In neural networks,

- **Goal:** Use gradient descent to find weights and biases that minimize

$$L(\mathbf{w}, b) = \frac{1}{2n} \sum_i (y_i - \hat{y}_i)^2$$

- **Gradient descent update rules:**

$$w_k \to w'_k = w_k - \eta \frac{\partial L}{\partial w_k}, \quad b_l \to b'_l = b_l - \eta \frac{\partial L}{\partial b_l}$$

- Repeatedly applying the update enables us to ”roll down the hill” to the minimum of the cost function.

Function composition

Deep neural networks utilize a significant amount of function composition.

- **Basic neuron structure:**

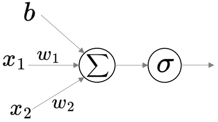
$$f(\mathbf{x}) = \sigma(g(\mathbf{x})), \quad \text{where } g(\mathbf{x}) = \mathbf{w}\mathbf{x} + b$$

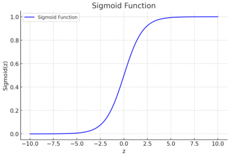
- The activation function $\sigma(z)$ can be a sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- **Differentiation using the chain rule:**

$$f(\mathbf{x})' = \sigma'(g(\mathbf{x}))g'(\mathbf{x})$$



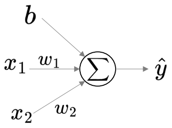


Gradient Descent for Linear Regression

Suppose that we have

- a **linear model:**

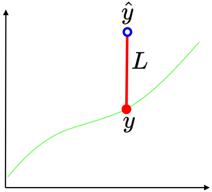
$$f(\mathbf{x};\mathbf{w},b) = w_1 x_1 + w_2 x_2 + b$$



- a **loss function**:

$$L(\mathbf{w},b) = (\hat{y} - y)^2$$

which compares the predicted value $\hat{y} = f(\mathbf{x};\mathbf{w},b)$ to the true value y

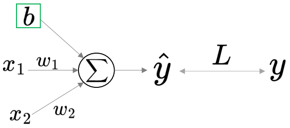


- **Compute the gradients using learning rate η:**

Update rules involve derivatives of L with respect to $w_1, w_2,$ and b.

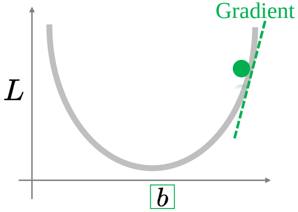
Gradients of b

- a **linear model:** $\hat{y} = f(\mathbf{x}; \mathbf{w}, b) = w_1 x_1 + w_2 x_2 + b$
- **Loss function** L: compares the predicted value $\hat{y}$ to the true value y



$$
\begin{aligned}
L(\mathbf{w}, b) &= (y - \hat{y})^2 \\
&= y^2 - 2y\hat{y} + \hat{y}^2
\end{aligned}
$$

- **Gradient of** L **w.r.t.** b: using the chain rule:



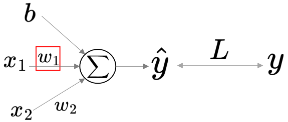
$$\Rightarrow \frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = 2(\hat{y} - y)$$

$$\frac{\partial L}{\partial \hat{y}} = 2\hat{y} - 2y; \qquad \frac{\partial \hat{y}}{\partial b} = 1$$

Gradients of w_1

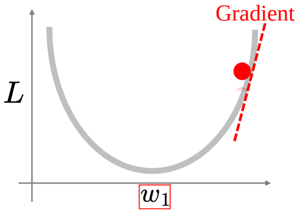
Linear model and loss function:

$$\hat{y} = f(\mathbf{x}; \mathbf{w}, b) = w_1 x_1 + w_2 x_2 + b$$
$$L(\mathbf{w}, b) = y^2 - 2y\hat{y} + \hat{y}^2$$



Gradient of L with respect to w_1:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1}$$



From the chain rule:

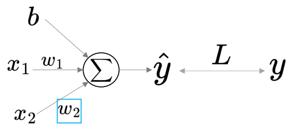
$$\frac{\partial L}{\partial \hat{y}} = 2\hat{y} - 2y, \quad \frac{\partial \hat{y}}{\partial w_1} = x_1$$

$$\Rightarrow \frac{\partial L}{\partial w_1} = (2\hat{y} - 2y)x_1$$

Gradients of w_2 and update rule

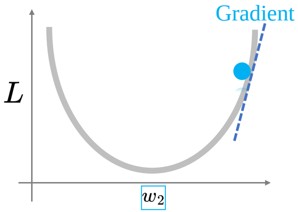
- Linear model and loss function:

$$\hat{y} = f(\mathbf{x}; \mathbf{w}, b) = w_1 x_1 + w_2 x_2 + b$$
$$L(\mathbf{w}, b) = y^2 - 2y\hat{y} + \hat{y}^2$$



- Similarly, **gradient of L with respect to w_2**:

$$\frac{\partial L}{\partial w_2} = (2\hat{y} - 2y)x_2$$



- Update rules:

$$b' = b - \eta \frac{\partial L}{\partial b}, \qquad w_1' = w_1 - \eta \frac{\partial L}{\partial w_1}, \qquad w_2' = w_2 - \eta \frac{\partial L}{\partial w_2}.$$

Gradient of a sigmoidal neuron

- Sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

- Use the quotient rule:

$$\frac{d}{dz}\left(\frac{\text{num}}{\text{den}}\right) = \frac{\text{den} \cdot \frac{d}{dz}(\text{num}) - \text{num} \cdot \frac{d}{dz}(\text{den})}{\text{den}^2}$$

- Chain rule:

$$\frac{d}{dz}\left(\sigma(g(z))\right) = \sigma'(g(z)) \cdot g'(z)$$

- Result:

$$\frac{d\sigma(z)}{dz} = \sigma(z)\left(1 - \sigma(z)\right)$$

- For more details, see: Understanding the Derivative of the Sigmoid Function

SGD for Large Sample Challenges

- **Challenge:** The loss function $L(\mathbf{w}, b)$ is defined as an average over training example costs:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_i \ell_i, \quad \text{where } \ell_i = \frac{(y_i - \hat{y}_i)^2}{2}$$

- The gradient of the loss function is:

$$\nabla L = \frac{1}{n}\sum_i \nabla \ell_i$$

Stochastic Gradient Descent for large sample challenges

- The gradient of the loss function is:

$$\nabla L = \frac{1}{n} \sum_i \nabla \ell_i$$

  - Computing the gradients for each training input x can be slow for large sample sizes.
  - $\Rightarrow$ Learning occurs slowly.
- **Solution:** We can speed things up by computing $\nabla \ell_i$ for a small random sample of training inputs:
  - Provides an estimate of ∇L.
  - Speeds up gradient descent and learning.
  - This approach is known as **Stochastic Gradient Descent (SGD)**.

Stochastic Gradient Descent (SGD)

- **Pick a mini-batch which is a random set of m training inputs $x_1, x_2, \ldots, x_m$:**
- **If m is large enough,** the average value of $\nabla \ell_j$ will be approximately equal to the average over all ∇L:

$$\frac{1}{m}\sum_{j=1}^{m} \nabla \ell_j \approx \frac{1}{n}\sum_{x} \nabla \ell_i = \nabla L$$

- **In neural networks, this gives update steps:**

$$w_k \to w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial \ell_j}{\partial w_k}$$

$$b_l \to b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial \ell_j}{\partial b_l}$$

Summary: Stochastic gradient descent

A training **epoch**:

1. Pick a random subset of the training data.
    - Referred to as a **mini-batch**.
2. Update the weights and biases using the gradient estimates from the mini-batch.
3. Pick another random mini-batch from the remaining training points and repeat step 2.
    - Repeat until all training inputs have been used.

Repeat multiple epochs until stopping conditions are met.

Analogy to political polling

- It is much easier to carry out a poll than to run a full election.
- Similarly, it's much easier to estimate gradients from mini-batches than the entire training set.
- **Downside:** Gradient estimates will be noisier in SGD.
- That's okay: we only need to move in a general direction that decreases L.
    - Don't need an extremely accurate estimate of the gradient.
- **In practice:** SGD is used extensively in learning neural networks.