📐 Matrix Multiplication & Vectorization

Matrix Multiplication Formula:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Example:

Given matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Their product $C = AB$:

$$C = \begin{bmatrix} (1)(5) + (2)(7) & (1)(6) + (2)(8) \\ (3)(5) + (4)(7) & (3)(6) + (4)(8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Vectorization: Replace loops with efficient operations (e.g., NumPy's `np.dot()`).

🧠 Broadcasting in NumPy & PyTorch: Deep Learning Notes

✅ What is Broadcasting?

Broadcasting allows element-wise operations between tensors of **different shapes** by **automatically expanding** their shapes *without copying data* .

It makes code **concise**, **fast**, and **vectorized**.

📏 Broadcasting Rule

To check if two shapes are broadcast-compatible:

1. Align shapes **from right to left**
2. For each dimension:
    - ✅ They must be equal, **or**
    - ✅ One of them must be `1`
    - ❌ Otherwise: broadcasting fails
3. If ranks differ, **left-pad the shorter shape with `1` s**

📐 Examples

✅ Valid Broadcasting

Tensor A Shape	Tensor B Shape	Explanation
`(3,)`	`(2, 3)`	`(3,)` becomes `(1, 3)`
`(2, 3)`	`(2, 1)`	`1` broadcasts to `3` (columns)
`(2, 3)`	`(1, 3)`	`1` broadcasts to `2` (rows)
`(4, 3, 2)`	`(1, 3, 1)`	Broadcasts across batch & last

❌ Invalid Broadcasting

Tensor A Shape	Tensor B Shape	Why It Fails
`(2, 3)`	`(4, 1)`	`2 ≠ 4` , neither is `1`
`(4, 3, 2)`	`(2, 1)`	`(2, 1)` becomes `(1, 2, 1)` , but `3 ≠ 2`

📤 Left Padding

If shapes differ in length, pad the **left side** of the smaller shape with `1` s.

$$\text{Example: } (4, 3, 2) \text{ and } (2, 1) \rightarrow (4, 3, 2) \text{ and } (1, 2, 1)$$

🧪 Bias Addition in Deep Learning

Broadcasting enables efficient bias addition in layers like:

$$y = x W^T + b$$

- x: shape $(\text{batch\_size}, \text{in\_features})$
- W: shape $(\text{out\_features}, \text{in\_features})$
- b: shape $(\text{out\_features},)$ → broadcast across batch

Result: $$y \in \mathbb{R}^{\text{batch\_size} \times \text{out\_features}}$$

🛠️ Useful PyTorch Tools

Function	Use Case
`.unsqueeze(dim)`	Add a singleton dimension
`.view(...)`	Reshape tensor (flexible)
`.expand(...)`	Broadcast without copying memory
`.repeat(...)`	Duplicate data (makes a copy)

🔥 Tips

- Always align shapes **right to left**
- Broadcasting is used in:
  - Bias addition
  - Normalization
  - Attention masking
  - Feature-wise operations across batches

🔥 Activation Functions & Logits

Logits: Raw, unnormalized outputs from neural networks.

Euler's number e:

- Approximately $e \approx 2.71828$
- Smooth differentiability: $\frac{d}{dx}(e^x) = e^x$

Detailed Activation Functions:

Activation	Formula	Output Range	Use
ReLU	$\max(0, x)$	$[0, \infty)$	Hidden layers
Sigmoid	$\frac{1}{1+e^{-x}}$	$(0, 1)$	Binary classification
Tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$(-1, 1)$	Hidden layers
Softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$	$[0, 1]$, sum=1	Multi-class output
GELU	See below	Smooth ReLU	Transformers

Detailed GELU (Gaussian Error Linear Unit):

Intuition: GELU smoothly activates neurons using Gaussian probability.

Exact Formula:

$$\text{GELU}(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

Approximation (common practice):

$$\text{GELU}(x) \approx 0.5x\left[1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right]$$

Why GELU?

- Smooth gradients
- Improved performance in modern NLP (Transformers, GPT models)

🔧 Activation Functions & Gradient Problems

✅ Why Use Activation Functions?

- Add **non-linearity** to the network.
- Allow stacking layers to model complex, non-linear patterns.
- Without activation functions, the network is just a linear mapping:
  $$f(W_2 (W_1 x)) = (W_2 W_1) x$$

⚡ Common Activation Functions

Function	Formula	Gradient Behavior	Notes		
ReLU	$\max(0, x)$	1 for $(x > 0)$, 0 for $(x < 0)$	Fast, simple, but can "die"		
GELU	$x \cdot \Phi(x) \approx 0.5x[1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3))]$	Smooth, non-zero for all (x)	Used in Transformers		
Sigmoid	$\frac{1}{1 + e^{-x}}$	Vanishes for large (	x	)	Not zero-centered
Tanh	$\tanh(x)$	Vanishes for large (	x	)	Zero-centered

🧠 Vanishing Gradient Problem

What It Is:

- Gradients shrink **exponentially** during backpropagation.
- Eventually, they become so small that **weights stop updating** → network stops learning.

When It Happens:

- Deep networks
- Using **sigmoid** or **tanh**
- Poor weight initialization

Visual Clue:

- Gradient of sigmoid/tanh flattens out for large $(|x|)$

💥 Exploding Gradient Problem

What It Is:

- Gradients grow **exponentially** during backpropagation.
- Leads to **instability**, large weights, or NaNs.

When It Happens:

- Deep or recurrent networks
- Large initial weights
- ReLU without proper controls

🚀 Why ReLU & GELU Are Popular

ReLU:

- No vanishing gradient for $(x > 0)$
- Sparse activation helps generalization
- Risk of "dying neurons" (stuck at 0)

GELU:

- Smooth, differentiable everywhere
- Allows small negative values
- Excellent gradient flow
- Standard in **Transformer** architectures

📊 Gradient Behavior Summary (Visual Insight)

- **ReLU**: 0 for (x < 0), 1 for (x > 0)
- **GELU**: Smooth curve; never flat like sigmoid
- **Sigmoid/Tanh**: Gradient vanishes as (|x|) increases → problem in deep nets

---

🧮 Summary Table of Activation Functions

Activation	Output Range	Advantages	Issues
ReLU	$[0, \infty)$	Fast, reduces vanishing gradients	Can "die"
Sigmoid	$(0, 1)$	Probabilities	Vanishing gradients
Tanh	$(-1, 1)$	Zero-centered	Vanishing gradients
Softmax	$[0, 1]$, sums=1	Multi-class probability	Computation cost
GELU	Smooth ReLU	Stable gradients, modern NLP	Slightly complex

🔁 Neural Network Layer Stacking (Shape Flow)

🧮 Tensor Dimensions Across Layers (Batch Size = 16)

```
Input: x → (16, 100)

Layer 1: Linear(100 → 64)
Weights: W1 → (100, 64)
Bias:    b1 → (64,)
Output:  (16, 64)

Activation: ReLU
Output: (16, 64)

Layer 2: Linear(64 → 32)
Weights: W2 → (64, 32)
Bias:    b2 → (32,)
Output:  (16, 32)

Activation: ReLU
Output: (16, 32)

Layer 3: Linear(32 → 1)
Weights: W3 → (32, 1)
Bias:    b3 → (1,)
Output:  (16, 1)

Activation: Sigmoid
Final Output: (16, 1)
```

📌 Each layer performs `output = input @ weights + bias`

🎯 3. Gradient Descent & Optimization

Gradient Descent (Basics)

Parameter update to minimize loss $L(\theta)$:

$$\theta = \theta - \eta \nabla_{\theta} L(\theta)$$

🏠 House Price Gradient Descent Example (Intuitive):

Predict house price from square footage:

$$\text{Price} = w \times (\text{SqFt}) + b$$

Given data:

- $x = 1000$, actual price $y = 200000$.
- Initial guess: $w = 100$, $b = 0$, prediction = $100,000 (error = $100,000)

Loss (MSE):

$$L = (y - (wx + b))^2$$

Gradients:

- Gradient w.r.t. weight w:
$$\frac{\partial L}{\partial w} = -2x(y - (wx + b))$$
- Gradient w.r.t. bias b:
$$\frac{\partial L}{\partial b} = -2(y - (wx + b))$$

Numerical Gradient Calculation:

- w.r.t. w: $-2 \times 1000 \times (200000 - 100000) = -200,000,000$
- w.r.t. b: $-2 \times (200000 - 100000) = -200,000$

Update (learning rate $\eta = 0.00000001$):

- New $w = 102$, New $b = 0.002$ (Loss decreases iteratively)

🚀 Adam Optimizer

Combines momentum and adaptive scaling:

Formulas:

- First moment (momentum):
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
- Second moment (adaptive scaling):
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
- Bias correction:
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
- Parameter update:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Typical hyperparameters: $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \eta = 0.001,\ \epsilon = 10^{-8}$

📘 Adam Optimizer — Summary Notes

🔍 What is Adam?

Adam (Adaptive Moment Estimation) is an optimizer that combines the benefits of:

- **Momentum**: Smooths the gradient using an exponentially weighted moving average
- **RMSProp**: Adapts the learning rate for each parameter based on gradient magnitudes

It's fast, robust to noise, and a strong default for deep learning tasks.

📐 Adam Update Rules

Let g_t be the gradient at time step t. Adam keeps two moving averages:

1. **First moment (mean of gradients)**

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

2. **Second moment (uncentered variance of gradients)**

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

🧮 Bias-Corrected Estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

🔁 Parameter Update:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

🔧 Default Hyperparameters

- Learning rate: $\alpha = 0.001$
- $\beta_1 = 0.9$ (momentum term)
- $\beta_2 = 0.999$ (RMSProp term)
- $\epsilon = 10^{-8}$ (prevents division by zero)

🧠 Intuition

- m_t tracks **direction** (momentum)
- v_t tracks **magnitude** (adaptive step size)
- Bias correction ensures values are accurate early on
- Works well with **noisy** or **sparse** gradients

🧪 Example Calculation

Given:

- $g_1 = 0.4$, $m_0 = 0$, $v_0 = 0$
- $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\alpha = 0.001$

Then:

- $m_1 = 0.1 \cdot 0.4 = 0.04$

- $v_1 = 0.001 \cdot (0.4)^2 = 0.00016$
- $\hat{m}_1 = \frac{0.04}{1-0.9} = 0.4$
- $\hat{v}_1 = \frac{0.00016}{1-0.999} = 0.16$
- Step size:

$$\alpha \cdot \frac{\hat{m}_1}{\sqrt{\hat{v}_1} + \epsilon} \approx 0.001 \cdot \frac{0.4}{0.4} = 0.001$$

✅ Pros

- Fast convergence
- Good for noisy or sparse data
- Adaptive step sizes per parameter
- Little tuning required

❌ Cons

- Sometimes worse generalization than SGD
- Can converge to sharp minima
- Slightly higher memory usage

💡 Tips

- Use **Adam** as your starting optimizer
- Try **SGD with momentum** for better generalization
- Use **AdamW** instead of L2 weight decay with Adam (better regularization)

📘 Gradient Descent, SGD, and Adam — Notes

🔁 Gradient Descent (GD)

Goal: Minimize a loss function $J(\theta)$ with respect to parameters θ.

Update rule:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Where:

- η: Learning rate
- $\nabla_\theta J(\theta)$: Gradient of the loss function

In **batch GD**, the gradient is computed over the entire dataset:

$$\nabla_\theta J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta J(\theta; x_i)$$

🎲 Stochastic Gradient Descent (SGD)

Key idea: Use a single (or a few) randomly selected data point(s) to approximate the gradient.

SGD update (single data point):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x_i)$$

Mini-batch SGD update (batch B):

$$\theta = \theta - \eta \cdot \frac{1}{|B|} \sum_{x_i \in B} \nabla_\theta J(\theta; x_i)$$

Why use SGD?

- Faster updates
- Can handle large datasets
- Noisy updates help escape local minima

⚡ Adam Optimizer (Adaptive Moment Estimation)

Adam improves SGD by adapting the learning rate using running averages of gradient moments.

Step-by-step:

1. Compute gradient:

$$g_t = \nabla_\theta J(\theta_t)$$

2. Update biased first moment estimate (mean):

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

3. Update biased second moment estimate (uncentered variance):

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

4. Bias-correct the estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

5. Update parameters:

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typical hyperparameters:

- $\eta = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\epsilon = 10^{-8}$

✅ Practical Notes

- **Adam is usually used with mini-batches**, not full-batch.
- SGD and its variants (like Adam) benefit from shuffling and batching data.
- Adam often converges faster and works well out-of-the-box.

```
In [ ]:
```

```
In [ ]:
```