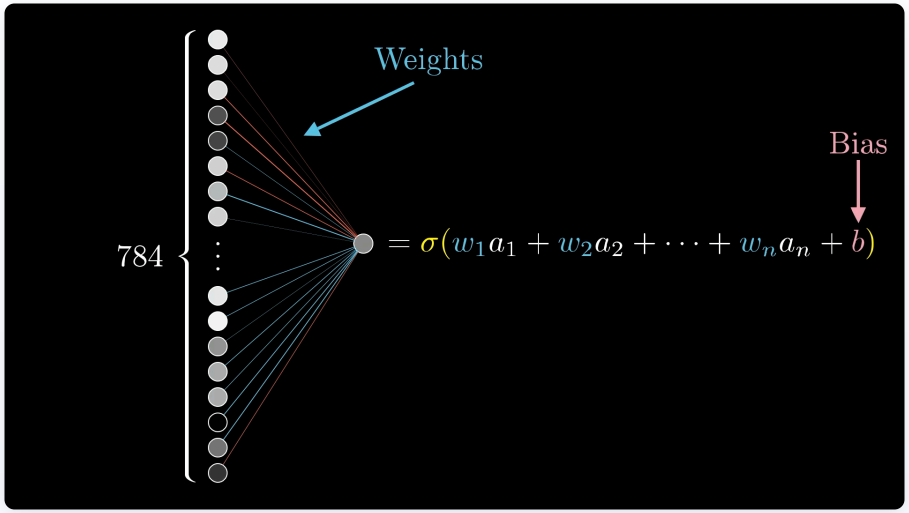
Session 3

Neural Networks and Backpropagation

Deep Learning course - SKEMA 2025 - Mastère Spécialisé® Chef de Projet Intelligence Artificielle - Salem Lahlou

Some figures adapted from 3Blue1Brown (YouTube)

(Reminder) Equation of a neuron



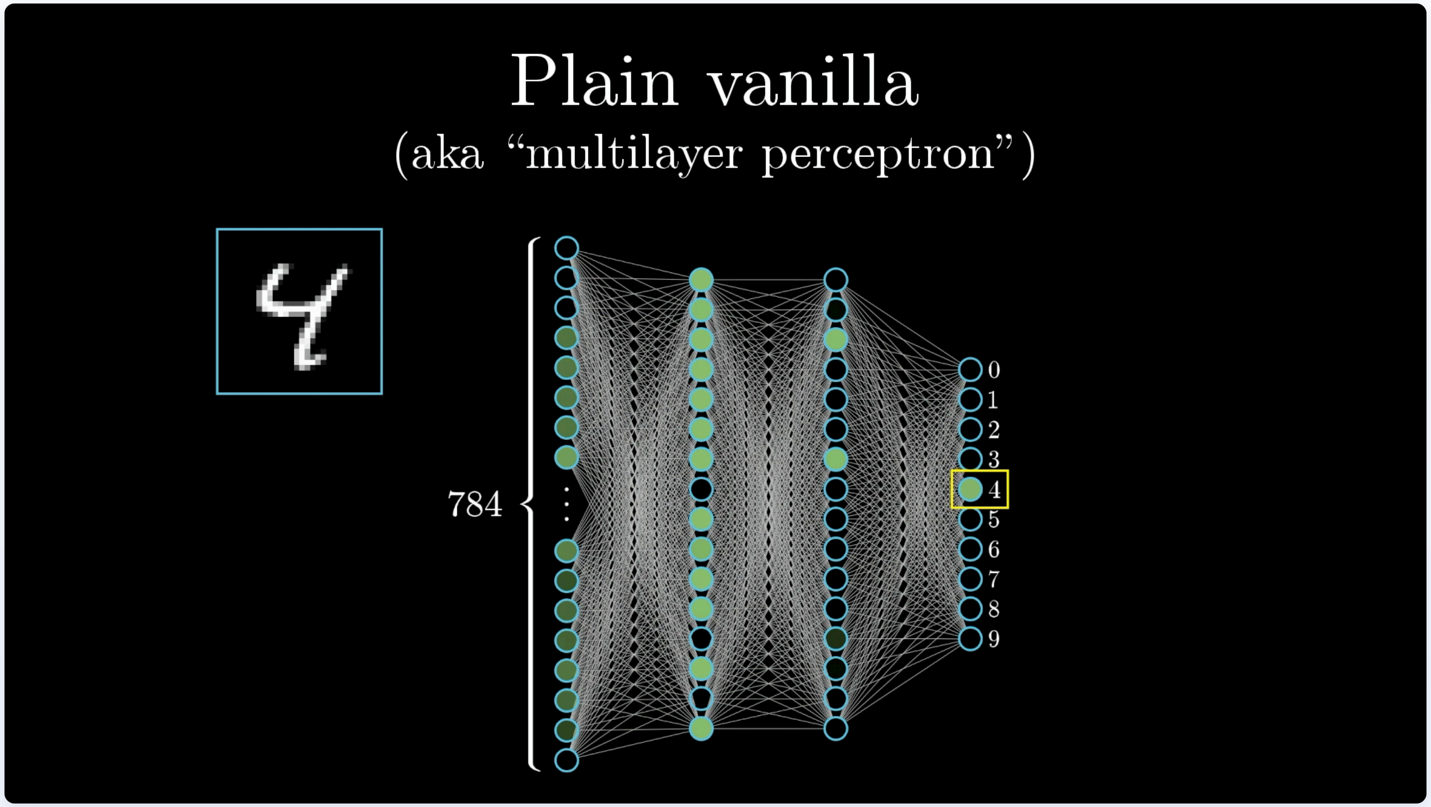
$$= \sigma(w_1 a_1 + w_2 a_2 + \cdots + w_n a_n + b)$$

What about the equation of multiple neurons in the hidden layers?

Mathematical Representation of Neural Networks

Single hidden layer network architecture:

- Input layer: $\mathbf{x} \in \mathbb{R}^{n_x}$
- Hidden layer: $\mathbf{a}^{[1]} \in \mathbb{R}^{n_h}$
- Output layer: $\mathbf{a}^{[2]} \in \mathbb{R}^{n_y}$
- Parameters: $\mathbf{W}^{[1]} \in \mathbb{R}^{n_h \times n_x}$, $\mathbf{b}^{[1]} \in \mathbb{R}^{n_h}$, $\mathbf{W}^{[2]} \in \mathbb{R}^{n_y \times n_h}$, $\mathbf{b}^{[2]} \in \mathbb{R}^{n_y}$



Forward Propagation: Single Example (Scalar Form)

For a single training example:

Hidden layer computation:

- For each hidden unit j:
  - $z_j^{[1]} = \sum_{i=1}^{n_x} w_{ji}^{[1]} x_i + b_j^{[1]}$
  - $a_j^{[1]} = g^{[1]}(z_j^{[1]})$

Output layer computation:

- For each output unit k:
  - $z_k^{[2]} = \sum_{j=1}^{n_h} w_{kj}^{[2]} a_j^{[1]} + b_k^{[2]}$
  - $a_k^{[2]} = g^{[2]}(z_k^{[2]})$

Where $g^{[1]}$ and $g^{[2]}$ are activation functions for the respective layers.

Forward Propagation: Single Example (Vector Form)

For a single training example:

Hidden layer computation:

- $\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$
- $\mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]})$

Output layer computation:

- $\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$
- $\mathbf{a}^{[2]} = g^{[2]}(\mathbf{z}^{[2]})$

Dimensions check:

- $\mathbf{z}^{[1]}, \mathbf{a}^{[1]} \in \mathbb{R}^{n_h}$
- $\mathbf{z}^{[2]}, \mathbf{a}^{[2]} \in \mathbb{R}^{n_y}$

Forward Propagation: Batch Processing (Matrix Form)

For a batch of m examples:

Input: $\mathbf{X} \in \mathbb{R}^{n_x \times m}$ (each column is one example)

Hidden layer computation:

- $\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]}$
- $\mathbf{A}^{[1]} = g^{[1]}(\mathbf{Z}^{[1]})$

Remember that: $\mathbf{W}^{[1]} \in \mathbb{R}^{n_h \times n_x}$, $\mathbf{b}^{[1]} \in \mathbb{R}^{n_h}$,

Output layer computation:

- $\mathbf{Z}^{[2]} = \mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{b}^{[2]}$
- $\mathbf{A}^{[2]} = g^{[2]}(\mathbf{Z}^{[2]})$

Remember that $\mathbf{W}^{[2]} \in \mathbb{R}^{n_y \times n_h}$, $\mathbf{b}^{[2]} \in \mathbb{R}^{n_y}$

Dimensions check:

- $\mathbf{Z}^{[1]}, \mathbf{A}^{[1]} \in \mathbb{R}^{n_h \times m}$
- $\mathbf{Z}^{[2]}, \mathbf{A}^{[2]} \in \mathbb{R}^{n_y \times m}$
- $\mathbf{b}^{[1]}$ is broadcast to match dimensions

Broadcasting in Neural Network Computations

Broadcasting: Automatically expanding dimensions to enable operations

Example in forward propagation:

- $\mathbf{b}^{[1]} \in \mathbb{R}^{n_h \times 1}$ but we need to add it to $\mathbf{W}^{[1]}\mathbf{X} \in \mathbb{R}^{n_h \times m}$
- Broadcasting: $\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]}$
- Effectively: $\mathbf{b}^{[1]}$ is copied m times to become $\mathbb{R}^{n_h \times m}$

```
W^[1]X   : [n_h × m]
b^[1]    : [n_h × 1]  ->  [n_h × m]  (broadcast)
Z^[1]    : [n_h × m]
```

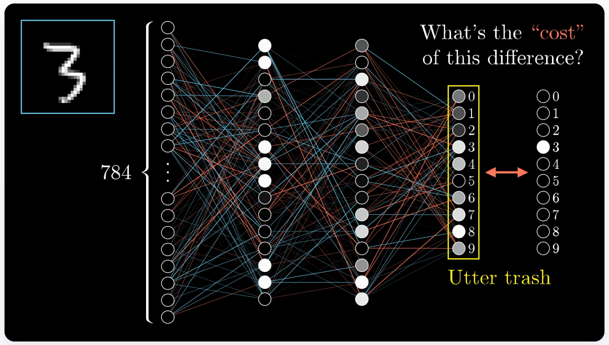
Quiz time

Go to the course homepage: https://la7.lu/sk25

Click on "Quiz 3"

Don't worry. It's anonymous!

We need to minimize the cost function



Remember the gradient descent algorithm?

Pseudocode for Gradient Descent:

```
1. Initialize parameters $\theta$ randomly
2. Repeat until convergence:
    ◦ Compute cost $J(\theta)$
    ◦ Compute gradient $\nabla_\theta J(\theta)$
    ◦ Update parameters: $\theta = \theta - \alpha \nabla_\theta J(\theta)$
```

For neural networks:

- θ includes all weights $W^{[l]}$ and biases $b^{[l]}$
- Updates for each layer l:
    - $W^{[l]} = W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}}$
    - $b^{[l]} = b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}}$

For that, we need gradients

13,002 weights and biases

How to nudge all weights and biases

$$\vec{\mathbf{W}} = \begin{bmatrix} 2.25 \\ -1.57 \\ 1.98 \\ \vdots \\ -1.16 \\ 3.82 \\ 1.21 \end{bmatrix} \qquad -\nabla C(\vec{\mathbf{W}}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ 0.40 \\ -0.32 \\ 0.82 \end{bmatrix}$$

Common Activation Functions and Their Derivatives

ReLU (Rectified Linear Unit):

- $\mathrm{ReLU}(z) = \max(0, z)$
- $\mathrm{ReLU}'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$
- (In practice, we often set $\mathrm{ReLU}'(0) = 0$ or $\mathrm{ReLU}'(0) = 1$)

Sigmoid:

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

Tanh:

- $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- $\tanh'(z) = 1 - \tanh^2(z)$

Loss Functions for Neural Networks

Binary Cross-Entropy (for binary classification):

- $\mathcal{L}(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$
- For m examples: $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, \hat{y}^{(i)})$

Categorical Cross-Entropy (for multi-class classification):

- $\mathcal{L}(y, \hat{y}) = -\sum_{j=1}^{C} y_j \log(\hat{y}_j)$
- For m examples: $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, \hat{y}^{(i)})$

Mean Squared Error (for regression):

- $\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$
- For m examples: $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, \hat{y}^{(i)})$

Gradient Descent Recap

Objective: Find parameters θ that minimize cost function $J(\theta)$

Update rule:
$\theta = \theta - \alpha \nabla_\theta J(\theta)$

For neural networks:

- θ includes all weights W and biases b
- $\nabla_\theta J(\theta)$ includes $\frac{\partial J}{\partial W^{[l]}}$ and $\frac{\partial J}{\partial b^{[l]}}$ for all layers l
- Need an efficient way to compute these gradients → backpropagation

The Chain Rule: Foundation of Backpropagation

Chain Rule in Calculus:

If $z = f(y)$ and $y = g(x)$, then:
$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$

Extended to multiple variables:

If $z = f(y_1, y_2, \ldots, y_n)$ and each $y_i = g_i(x)$, then:
$\frac{dz}{dx} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \cdot \frac{dy_i}{dx}$

You have seen this in high school:

- $(fog)'(x) = f'(g(x))g'(x)$
- Example: $(e^{x^2})' = 2xe^{x^2}$

Backpropagation: The Big Picture

Key insight: Efficiently compute gradients by working backwards

The process:

1. Perform forward propagation to compute all activations
2. Compute the output error (derivative of loss with respect to output)
3. Propagate error backwards through the network
4. Compute gradients for all parameters

Why it's efficient:

- Avoids redundant calculations
- Reuses intermediate results
- Computational complexity proportional to network size

Backpropagation: Mathematical Derivation (Part 1)

Starting from the cost function for a single example:
$\mathcal{L}(y, a^{[2]})$

We want to compute:

- $\frac{\partial \mathcal{L}}{\partial W^{[2]}}, \frac{\partial \mathcal{L}}{\partial b^{[2]}}$
- $\frac{\partial \mathcal{L}}{\partial W^{[1]}}, \frac{\partial \mathcal{L}}{\partial b^{[1]}}$

Defining intermediate derivatives:

- $\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial z^{[2]}}$
- $\delta^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}}$

These represent the error at each layer

Backpropagation: Mathematical Derivation (Part 2)

For the output layer:

$$\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[2]}} \cdot g^{[2]'}(z^{[2]})$$

For hidden layers:

$$\delta^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} = \frac{\partial \mathcal{L}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}}$$

But $\frac{\partial \mathcal{L}}{\partial a^{[1]}}$ requires more calculation:

$$\frac{\partial \mathcal{L}}{\partial a^{[1]}} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} = \delta^{[2]} \cdot W^{[2]}$$

Therefore:

$$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \odot g^{[1]'}(z^{[1]})$$

Where $\odot$ denotes element-wise multiplication.

Backpropagation: Mathematical Derivation (Part 3)

Computing gradients using the errors:

For weights:

- $\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \delta^{[2]}(a^{[1]})^T$
- $\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \delta^{[1]}x^T$

For biases:

- $\frac{\partial \mathcal{L}}{\partial b^{[2]}} = \delta^{[2]}$
- $\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \delta^{[1]}$

For a batch of m examples:

- $\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}^{(i)}}{\partial W^{[l]}}$
- $\frac{\partial J}{\partial b^{[l]}} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}^{(i)}}{\partial b^{[l]}}$

Backpropagation: Vector/Matrix Form for Mini-batches

Output layer error:
$\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial Z^{[2]}} = \frac{\partial \mathcal{L}}{\partial A^{[2]}} \odot g^{[2]'}(Z^{[2]})$

For binary cross-entropy with sigmoid:
$\delta^{[2]} = A^{[2]} - Y$ (simplified form)

Hidden layer error:
$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \odot g^{[1]'}(Z^{[1]})$

Gradients for weights and biases:

- $\frac{\partial J}{\partial W^{[2]}} = \frac{1}{m} \delta^{[2]} (A^{[1]})^T$
- $\frac{\partial J}{\partial b^{[2]}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[2](i)}$
- $\frac{\partial J}{\partial W^{[1]}} = \frac{1}{m} \delta^{[1]} X^T$
- $\frac{\partial J}{\partial b^{[1]}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[1](i)}$

Backpropagation for a 1-HL FNN: Algorithm Summary

Forward propagation:

1. $Z^{[1]} = W^{[1]}X + b^{[1]}$, $A^{[1]} = g^{[1]}(Z^{[1]})$
2. $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$, $A^{[2]} = g^{[2]}(Z^{[2]})$
3. Compute cost J

Backward propagation:

1. $\delta^{[2]} = A^{[2]} - Y$ (for binary classification)
2. $dW^{[2]} = \frac{1}{m}\delta^{[2]}A^{[1]T}$
3. $db^{[2]} = \frac{1}{m}\sum_{i=1}^{m}\delta^{[2](i)}$
4. $\delta^{[1]} = W^{[2]T}\delta^{[2]} \odot g^{[1]'}(Z^{[1]})$
5. $dW^{[1]} = \frac{1}{m}\delta^{[1]}X^T$
6. $db^{[1]} = \frac{1}{m}\sum_{i=1}^{m}\delta^{[1](i)}$

Update parameters:

1. $W^{[2]} = W^{[2]} - \alpha dW^{[2]}$
2. $b^{[2]} = b^{[2]} - \alpha db^{[2]}$
3. $W^{[1]} = W^{[1]} - \alpha dW^{[1]}$
4. $b^{[1]} = b^{[1]} - \alpha db^{[1]}$

Extending to Deep Networks

The general backpropagation algorithm for L layers:

Forward propagation (for $l = 1, 2, \ldots, L$):

1. $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$ (with $A^{[0]} = X$)
2. $A^{[l]} = g^{[l]}(Z^{[l]})$

Backward propagation:

1. $\delta^{[L]} = \nabla_{A^{[L]}} \mathcal{L} \odot g^{[L]'}(Z^{[L]})$ (output layer error)
2. For $l = L-1, L-2, \ldots, 1$:
   $\delta^{[l]} = W^{[l+1]T} \delta^{[l+1]} \odot g^{[l]'}(Z^{[l]})$
3. Compute gradients:
   $dW^{[l]} = \frac{1}{m} \delta^{[l]} A^{[l-1]T}$
   $db^{[l]} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[l](i)}$

Important Note: Automatic Differentiation

While understanding backpropagation is crucial...

- It helps build intuition
- It's essential for debugging complex models
- It's foundational knowledge

...you rarely implement it manually!

- Modern deep learning frameworks (PyTorch, TensorFlow, JAX) use **Automatic Differentiation (Autodiff)**.
- They automatically compute the necessary gradients using the chain rule during the backward pass.

Focus shifts from manual derivation to defining the forward pass correctly.

- We will see this practical automation in the Jupyter Notebook session.

Improving Gradient Descent: Batch Variants

Batch Gradient Descent (BGD):

- Uses the entire dataset to compute gradient
- $\theta = \theta - \alpha \nabla_\theta J(\theta)$
- Advantages: Stable, guaranteed convergence to local minimum
- Disadvantages: Slow, memory-intensive for large datasets

Stochastic Gradient Descent (SGD):

- Updates parameters using a single random example
- $\theta = \theta - \alpha \nabla_\theta J_i(\theta)$
- Advantages: Fast, can escape local minima, works with streaming data
- Disadvantages: High variance, noisy updates, may never converge exactly

Mini-batch Gradient Descent:

- Updates parameters using a small batch of examples
- $\theta = \theta - \alpha \nabla_\theta J_B(\theta)$ where B is mini-batch
- Advantages: Good balance of stability and speed, vectorization benefits
- Disadvantages: Requires tuning batch size

Batching: Training on Multiple Examples at Once

Benefits of mini-batch training:

- Parallelization on modern hardware (GPUs)
- More stable gradient estimates than SGD
- Faster convergence than full-batch GD
- Better generalization through noise

Implementation for a mini-batch:

- X contains multiple examples (columns)
- Forward and backward propagation operate on matrices
- Each column of activation matrices corresponds to one example
- Gradients are averaged over the mini-batch

Batch sizes:

- Too small: Noisy updates, poor hardware utilization
- Too large: Poor generalization, memory issues
- Common values: 32, 64, 128, 256

Dimensions in Mini-batch Processing

For a neural network with n_x inputs, n_h hidden units, n_y outputs, and batch size m:

Variable	Dimensions	Description
X	(n_x, m)	Input data, m examples
$W^{[1]}$	(n_h, n_x)	Weights connecting input to hidden
$b^{[1]}$	$(n_h, 1)$	Hidden layer biases
$Z^{[1]}$	(n_h, m)	Pre-activation at hidden layer
$A^{[1]}$	(n_h, m)	Activation at hidden layer
$W^{[2]}$	(n_y, n_h)	Weights connecting hidden to output
$b^{[2]}$	$(n_y, 1)$	Output layer biases
$Z^{[2]}$	(n_y, m)	Pre-activation at output layer
$A^{[2]}$	(n_y, m)	Output predictions
Y	(n_y, m)	Target values

Computation Efficiency with Vectorization

Vectorized operations vs. loops:

Loop implementation (slow):

```
for i in range(m):
    z[i] = np.dot(w, x[i]) + b
    a[i] = sigmoid(z[i])
```

Vectorized implementation (fast):

```
z = np.dot(w, x) + b
a = sigmoid(z)
```

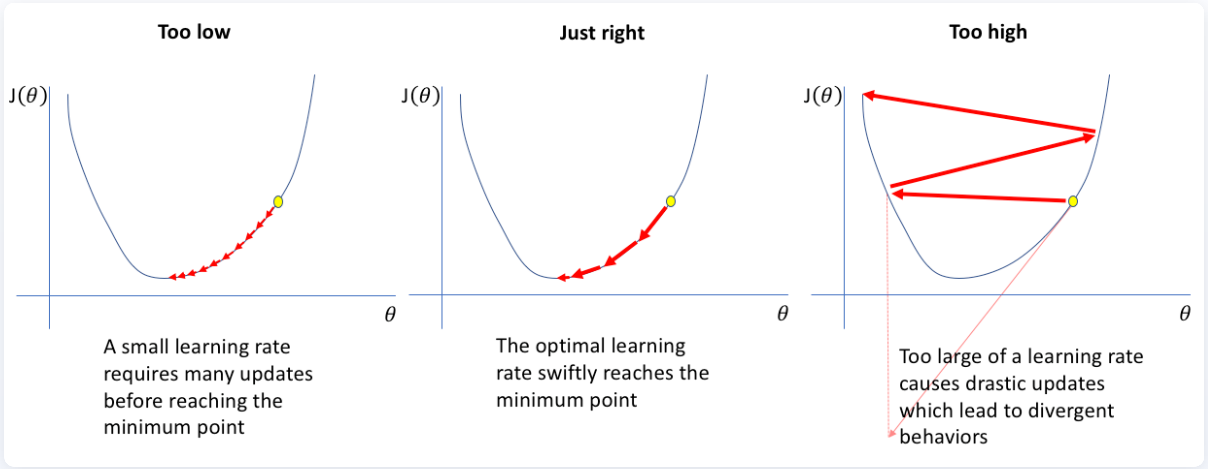
Benefits of vectorization:

- Orders of magnitude faster
- Better utilizes hardware (CPU/GPU)
- Cleaner code
- Essential for large datasets

The Importance of Learning Rate

Learning rate controls step size in gradient descent:

- Too small: Slow convergence, may get stuck
- Too large: Overshooting, divergence, instability


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Source: Jeremy Jordan

Mathematical explanation:

- Gradient gives direction
- Learning rate gives step size
- $W^{[l]} = W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}}$

Learning rate scheduling:

- Start with larger learning rate
- Gradually decrease over time
- Allows both fast progress and fine convergence

Evolution from SGD to Advanced Optimizers

Problem with basic SGD:

- Fixed learning rate for all parameters
- Equal updates in all directions
- No memory of previous gradients
- Easily stuck in saddle points

SGD with Momentum:

- Adds "memory" of previous updates
- $v_t = \gamma v_{t-1} + \alpha \nabla_\theta J(\theta)$
- $\theta = \theta - v_t$
- Accelerates in consistent directions, dampens oscillations

AdaGrad:

- Adapts learning rate per parameter based on historical gradients
- $\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla_\theta J(\theta_{t-1})$
- Where G_t accumulates squared gradients
- Problem: Learning rate decreases too much over time

Advanced Optimization Algorithms

RMSProp:

- Addresses AdaGrad's diminishing learning rates
- Uses exponential moving average of squared gradients
- $G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_\theta J(\theta))^2$
- $\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{G_t+\epsilon}} \nabla_\theta J(\theta_{t-1})$

Adam (Adaptive Moment Estimation):

- Combines momentum and RMSProp
- Maintains both first moment (mean) and second moment (variance)
- First moment: $m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta)$
- Second moment: $v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2$
- Bias correction: $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$, $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
- Update: $\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t+\epsilon}}$

Learning Rate Scheduling Techniques

Constant learning rate:

- $\alpha_t = \alpha$
- Simple but often suboptimal

Step decay:

- Reduce learning rate by a factor after fixed number of epochs
- $\alpha_t = \alpha_0 \times \gamma^{\lfloor t/s \rfloor}$
- Where γ is decay factor, s is step size

Exponential decay:

- Continuous decay at exponential rate
- $\alpha_t = \alpha_0 \times e^{-kt}$

Cosine annealing:

- Cyclical learning rate that follows cosine function
- $\alpha_t = \alpha_{min} + \frac{1}{2}(\alpha_{max} - \alpha_{min})(1 + \cos(\frac{t\pi}{T}))$
- Where T is cycle length

Quiz time

Go to the course homepage: https://la7.lu/sk25

Click on "Quiz 4"

Introduction to PyTorch

PyTorch philosophy:

- Pythonic and intuitive API
- Dynamic computation graph
- Easy debugging
- Strong GPU acceleration

Core components:

- `torch.Tensor`: Multi-dimensional array with automatic differentiation
- `torch.nn`: Neural network layers and components
- `torch.optim`: Optimization algorithms
- `torch.utils.data`: Data loading utilities

Basic example:

```python
import torch
# Create tensors
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = torch.tensor([4.0, 5.0, 6.0])
# Forward pass
z = x * y
# Backward pass
loss = z.sum()
loss.backward()
# View gradients
print(x.grad)  # Output: tensor([4., 5., 6.])
```

PyTorch Model Implementation

Defining models with nn.Module:

```python
import torch.nn as nn
import torch.nn.functional as F

class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instantiate model
model = SimpleNN(input_size=28*28, hidden_size=128, output_size=10)
```

Key components:

- `__init__`: Define layers and parameters
- `forward`: Define forward pass computation
- Model inherits from `nn.Module`

Training Loop in PyTorch

Complete training loop:

```python
def train(model, train_loader, criterion, optimizer, device, epochs=10):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for batch_idx, (data, target) in enumerate(train_loader):
            # Move data to device
            data, target = data.to(device), target.to(device)
            # Zero gradients
            optimizer.zero_grad()
            # Forward pass
            output = model(data)
            # Calculate loss
            loss = criterion(output, target)
            # Backward pass
            loss.backward()
            # Update parameters
            optimizer.step()
            # Track statistics
            running_loss += loss.item()

        # Print epoch statistics
        print(f'Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}')
```

Evaluation with PyTorch

Evaluation loop:

```python
def evaluate(model, test_loader, criterion, device):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():  # No gradients needed for evaluation
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            # Forward pass
            output = model(data)
            # Calculate loss
            test_loss += criterion(output, target).item()
            # Get predictions
            pred = output.argmax(dim=1, keepdim=True)
            # Count correct predictions
            correct += pred.eq(target.view_as(pred)).sum().item()

    # Calculate metrics
    test_loss /= len(test_loader)
    accuracy = 100. * correct / len(test_loader.dataset)

    print(f'Test Loss: {test_loss}, Accuracy: {accuracy}%')
    return test_loss, accuracy
```

Quiz time

Go to the course homepage: https://la7.lu/sk25

Click on "Quiz 4"

Don't worry. It's anonymous!

Jupyter notebook time!

Go to the course homepage: https://la7.lu/sk25

Click on "Colab 1"