Backpropagation

chain rule

- efficiently compute gradients of a loss with respect to all parameters
  - based on the chain rule
  - enables training of deep neural networks via gradient-based optimization
- propagate error signals from the output layer backward through the network

    - reusing intermediate derivatives → avoid redundant computations
- computation flow
    - forward pass: compute activations and loss
    - backward pass: compute gradients layer by layer from output to input
- **Chain rule**
    - each layer contributes a local derivative → gradients = products of local derivatives
- dynamic programming
    - cache intermediate activations during forward pass
    - reuse activations during backward pass to compute gradients efficiently
- complexity: linear in number of parameters

Gradient Accumulation

- simulate a larger batch size by accumulating gradients over multiple forward-backward passes
    - delays parameter updates until enough gradients are accumulated
- useful for overcoming limitations of GPU memory

Gradient Checkpointing

- store only a subset of activations during the forward pass → saving memory
    - decompose the network into segments with checkpoints
- recompute the rest during backpropagation
    - reruns forward computation from the nearest checkpoint when needed
- trade additional computation for reduced memory usage

Optimizer

- update model parameters to minimize a loss function
- determine update direction and efficient step size

Gradient Descent

$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$

- iterative optimization algorithm using the full dataset
  - updating parameters in the direction of the negative gradient of a loss fuctnion
  - moving parameters toward the direction of steepest descent
- assumption: local surface is locally smooth
- computationally expensive

Learning Rate

- hyperparameter (η) controlling how aggressively parameters are updated

Step Size

- $\text{step size} = \eta \cdot \|\nabla L(\theta)\|$ learning rate × gradient magnitude
- the actual magnitude of the parameter update
- **Gradient Clipping**
  - decouple the step size from the gradient magnitude
  - prevent exploding gradients without artificially stalling the model's convergence
    - only reducing the learning rate may not solve the problem

Stochastic Gradient Descent

- leverage a mini-batch to estimate the gradient instead of using the full dataset
  - produce a noisy estimate of the true gradient
- pros
  - faster iterations & scalable to large datasets
  - noise can help escape shallow local minima and saddle points
- cons
  - may oscilliate around minima
    - especially in high-curvature directions
  - noisy updates
- computational efficiency v.s. better generalization

Momentum

velocity 이전에 업데이트 방향으로 계속 이동

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

$$\theta_{t+1} = \theta_t - \eta v_t$$

- accumulate a velocity vector
  - builds speed in consistent descent directions
  - damp oscillations in steep directions → zig-zagging ↓

RMSProp

$$s_t = \rho s_{t-1} + (1-\rho)\, g_t^2$$

EMA of squared gradients

$$\theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{s_t + \epsilon}}$$

- track an exponential moving average of squared gradients → adaptive scaling
  - normalizing gradients by recent magnitude
- small updates in high-curvature directions

Adam

gradients < first – momentum
Second – adaptive scaling (RMSprop)

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

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

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

Variance ⇑ 방향 → 덜 이동

- using first and second moment of gradients
  - first moment (mean of gradients): momentum
  - second moment (variance of gradients): adaptive scaling
- fast convergence
- robust to noisy or sparse gradients

Activation Function

- introduce non-linearity
- control how gradients propagate through the network
- examples
  - **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
    - derivative: $\sigma(x)(1 - \sigma(x))$
  - **Tanh**: $\frac{e^x - e^{-x}}{e^x + e^x}$
    - derivatives saturate near 0
    - vanishing gradient risk
      - graidents shrink exponentially
    - exploding gradients
      - uncontrollably growing gradients
      - common in RNNs
  - **ReLU**
    - derivative is 1 for positie inputs
    - mitigate vanishing gradients

    - C controls tradeoff
        - $C \uparrow \rightarrow$ fewer violations, tighter fit (variance ↑)
        - $C \downarrow \rightarrow$ larger margin, more tolerance (bias ↑)
- hinge loss view
    - equivalent form:
        - $\min_{w,b}; \frac{1}{2}|w|^2 + C\sum_i \max(0, 1 - y_i f(x_i))$
    - margin violations are penalized linearly

$y_i f(x_i) \geq 1 \rightarrow$ margin 규칙 잘 만족

margin 위반할 정도만큼 penalty 줌

Kernel trick

- handle nonlinear boundaries by mapping inputs to a higher-dimensional feature space
    - explicit mapping $\phi(x)$ is not computed
    - use kernel $K(x, x') = \phi(x)^\top \phi(x')$
- dual form depends only on dot products
    - prediction:
        - $f(x) = \sum_{i \in SV} \alpha_i y_i K(x_i, x) + b$

kernel 사용 → 데이터 간 내적으로 표현

각 data sample이 support vector와 얼마나 유사한지

- common kernels
    - linear: $K(x, x') = x^\top x'$
    - RBF: $K(x, x') = \exp(-\gamma|x - x'|^2)$
    - polynomial: $K(x, x') = (x^\top x' + c)^d$

primal: 직접 변수 w, b을 최적화

경계의 parameter 직접 찾기

* dual: 각 data point에 대응하는 변수 α_i 최적화

각 sample이 경계를 얼마나 밀고 당기는지

→ w: data의 linear combination

* duality: optimization 문제를 다른 변수로 다시 씀

Lagrangian minimize $f(x)$ + subject to $h(x) = 0$ or $g(x) \leq 0$

$\mathcal{L}(x, \lambda) = f(x) + \lambda h(x)$ λ: Lagrangian multiplier

$\mathcal{L}(x, \alpha) = f(x) + \alpha g(x)$ where $\alpha \geq 0$

SVM에서는 Lagrangian 관점에서

$g(x) = 1 - y_i f(x_i) \leq 0$

$\min_x \max_{\alpha \geq 0} \mathcal{L}(x, \alpha)$ dual problem 1) α 고정하고 $\mathcal{L}$을 최소화하는 x 찾기

2) 그 x를 고정해놓고 $\mathcal{L}$을 최대화하는 α 찾기

Initialization

- choosing initial values of model parameters before training
    - affect gradient flow & training stability

    - poort initialization → vanishin or exploding gradients & failed convergence
    - during backpropagation
        - weights ↓ → gradients ↓ (shrinking)
        - weights ↑ → gradients ↑ (exploding)
- preserve variance of activations & gradients across layers
    - keep the variance of activations roughly constant

Random Initialization

$$W_{ij} \sim \mathcal{N}(0, \sigma^2) \quad \text{or} \quad \mathcal{U}(-a, a)$$

- break symmetry between neurons
    - but naive variance choices → instability ↑

Xavier Initialization

$$\text{Var}(W) = \frac{2}{\text{fan\_in} + \text{fan\_out}}$$

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}, \sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}} \right)$$

- balance variance of activations across layers
- assumes symmetric, saturating activations
    - e.g., tanh-like activations
    - zero-centered output → keep activations balanced
- harmonic mean on the numbers of nodes in input and output

HE (Kaiming) Initialization

$$\text{Var}(W) = \frac{2}{\text{fan\_in}}$$

$$W \sim \mathcal{N}\left(0, \frac{2}{\text{fan\_in}}\right)$$

- designed for ReLU-style activations