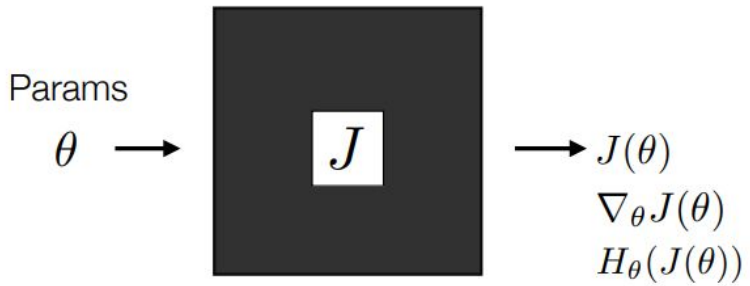
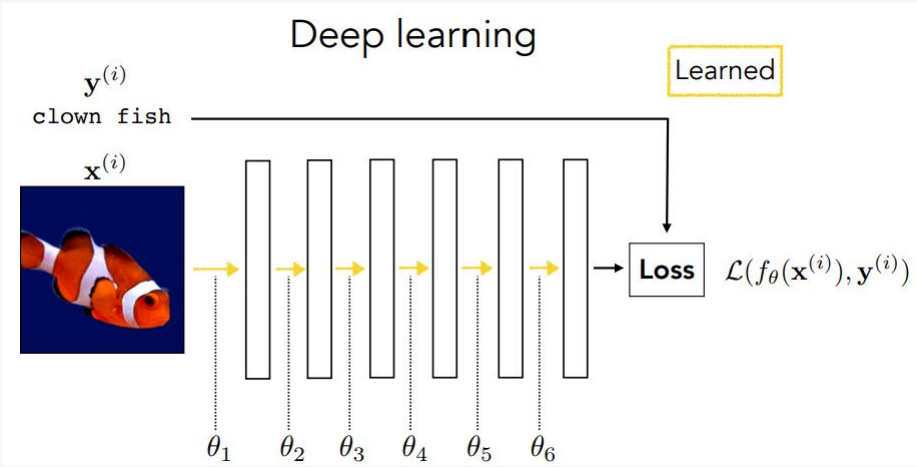
Lecture 2: How to train a neural net

정보량 관점에서 본 Optimization과 Backpropagation

핵심 개념

- 학습 목표는 전체 데이터에 대한 loss 합을 최소화하는 parameter **θ*** 를 찾는 것
- 최적화 문제는 **J(θ)** 에 대해 활용 가능한 정보량에 따라 세 단계로 구분
  - J(θ)만 구할 수 있음 → Black box optimization
  - J(θ)값 이랑 gradient값 구할 수 있음 → First order optimization
  - J(θ)값 + gradient + Hessian(2차 편미분) → Second order

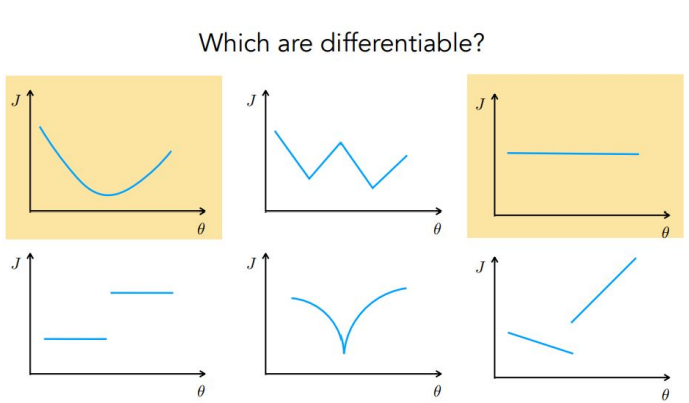


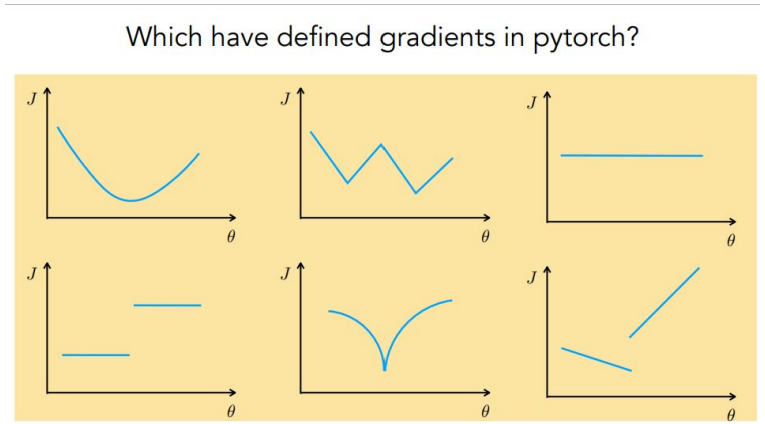


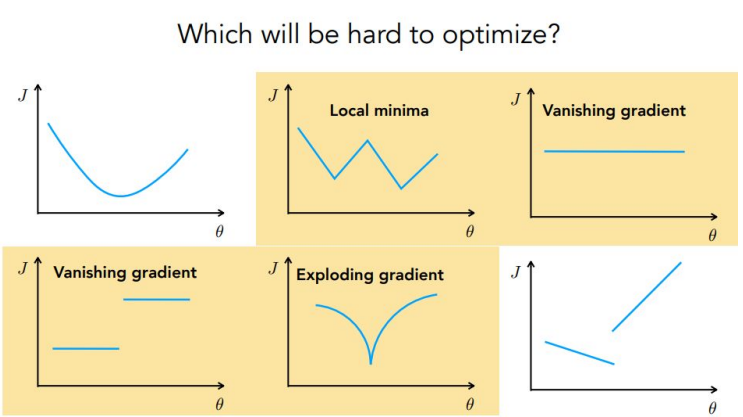
$$\theta^* = \arg\min_\theta \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})}_{J(\theta)}$$

“미분 가능”과 “최적화가 잘 된다”는 별개의 축

- 강의자료는 세 가지를 각각 **다른 질문**으로 분리해 제시
  - ① differentiable 여부
  - ② PyTorch에서 gradient 정의 여부
  - ③ optimization 난이도
- 세 속성은 서로 함의하지 않는 **독립적인 축**







핵심 개념

- **GD** : 전체 데이터에 대한 gradient로 업데이트
- **SGD** : Batch subset에 대해서만 gradient 계산 → noisy하지만 빠름
- **Momentum** : 이전 update 방향의 누적치 m□를 추가 반영
- gradient가 병적(vanishing·exploding·zero, 소실·발산·0)이면 GD/SGD는 **실패** 할 수 있음

$$\theta^* = \arg\min_\theta \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})}_{J(\theta)}$$

One iteration of gradient descent:

$$\theta^{k+1} \leftarrow \theta^k - \eta \nabla_\theta J(\theta^k)$$

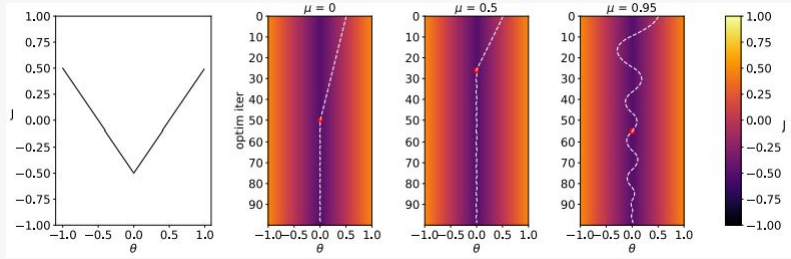
learning rate

Gradient Descent

학습률: η, 모멘텀 계수: α

$$\theta^{t+1} \leftarrow \theta^t - \eta \nabla f(\theta^t) - \alpha m^t$$

모멘텀 계수: μ



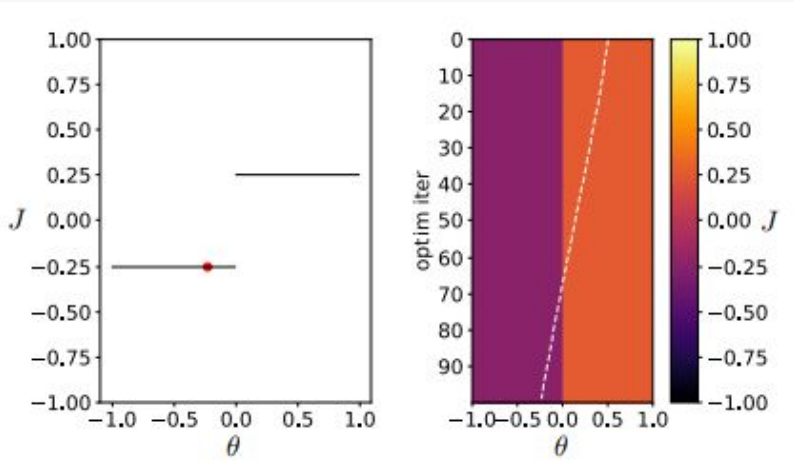
학습률: α , 모멘텀 계수: β

Momentum

Evolution Strategies: Black box optimization도 ‘gradient-like’하게 작동하게 한다

- 실제 gradient를 계산하지 않고도 **gradient-like** 하게 동작
- parameter에 작은 perturbation을 여러 개 뿌림
- loss가 낮아지는 방향의 가중평균으로 update 방향 생성
- black box optimization이 **first order(1차미분을 쓰는 경사하강법)처럼 행동하는 다리** 역할

$$\epsilon_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$
$$s_i = J(\theta + \sigma\epsilon_i)$$
$$\theta^{k+1} \leftarrow \theta^k - \eta \frac{1}{\sigma M} \sum_{i=1}^{M} s_i \epsilon_i$$



Gradient clipping: exploding gradient를 막기 위한 ‘hack’

- Gradient clipping은 기울기의 최댓값을 강제로 제한하는 방법(기울기벡터의 value or norm으로 제한)
- 교안은 이를 *“Useful, and commonly used hack”* 이라고 직접 표현
- elegant한 수학적 해법이 아니라, **exploding gradient** 라는 실패 모드를 막기 위한 실용적 응급처치
- ‘elegant한 해법’이 아님을 강의가 의도적으로 강조

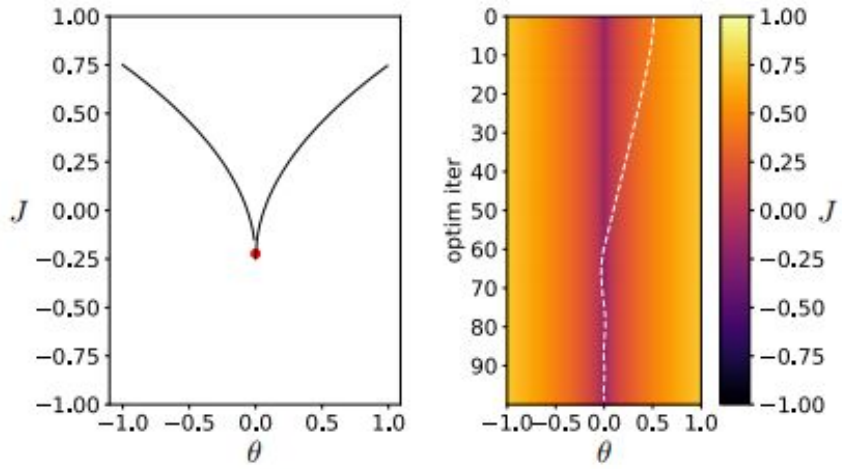
$$\mathbf{v} = \nabla_\theta J(\theta^k)$$

Value로 제한

$$\theta^{k+1} \leftarrow \theta^k - \eta[\texttt{clip}(v_1, -m, m), \ldots, \texttt{clip}(v_M, -m, m)]^\mathsf{T}$$

Norm으로 제한

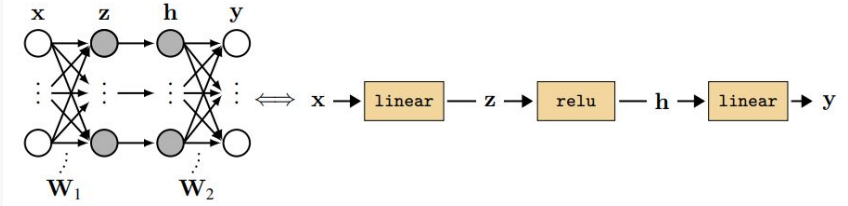
$$\theta^{k+1} \leftarrow \theta^k - \eta \left(\mathbf{v} \times \frac{m}{\max(\|\mathbf{v}\|, m)} \right)$$

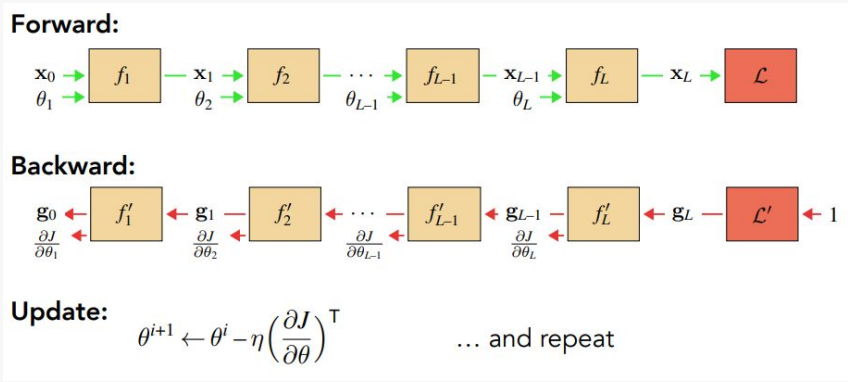


핵심 개념

- **Computation graph** : functional transformation들을 연결한 DAG, 각 node는 differentiable
- **Backpropagation** : 모든 derivative를 따로 계산하는 대신,
  - 공유되는 항(error signal g)을 graph를 따라 한 번만 전파해 재사용

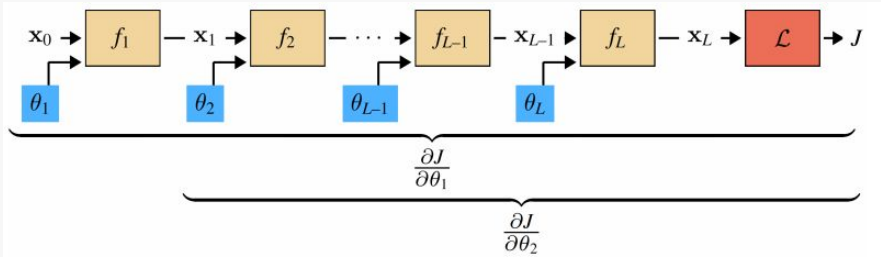
혹시… 다들 왜 Backpropagation이 있어야 하는지 아시나요?





Backprop의 본질은 '중복 계산 재사용' = Dynamic Programming

- Chain rule만으로도 모든 weight의 gradient 계산 가능
- 그러나 naive하게 하면 같은 항을 **여러 번 다시 계산**
- Backprop의 기여 = 공유 항을 graph 안에서 전파해 재사용
- dynamic programming 과 유사



$$\frac{\partial J}{\partial \theta_1} = \frac{\partial J}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}}\cdots\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \theta_1}$$

$$\frac{\partial J}{\partial \theta_2} = \frac{\partial J}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}}\cdots\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\partial \theta_2}$$

Batch averaging: backprop 바깥의 단순 평균

- batch 단위 backprop은 **특별한 알고리즘은 아님**
- 각 example마다 독립적으로 backprop을 수행한 뒤 **결과를 평균** 내는 것
- 주로 Mini-batch Gradient Descent를 사용할 때 사용
- 전체 데이터를 대표하는 안정적이고 정확한 방향으로 모델을 학습시키기 위해서 사용

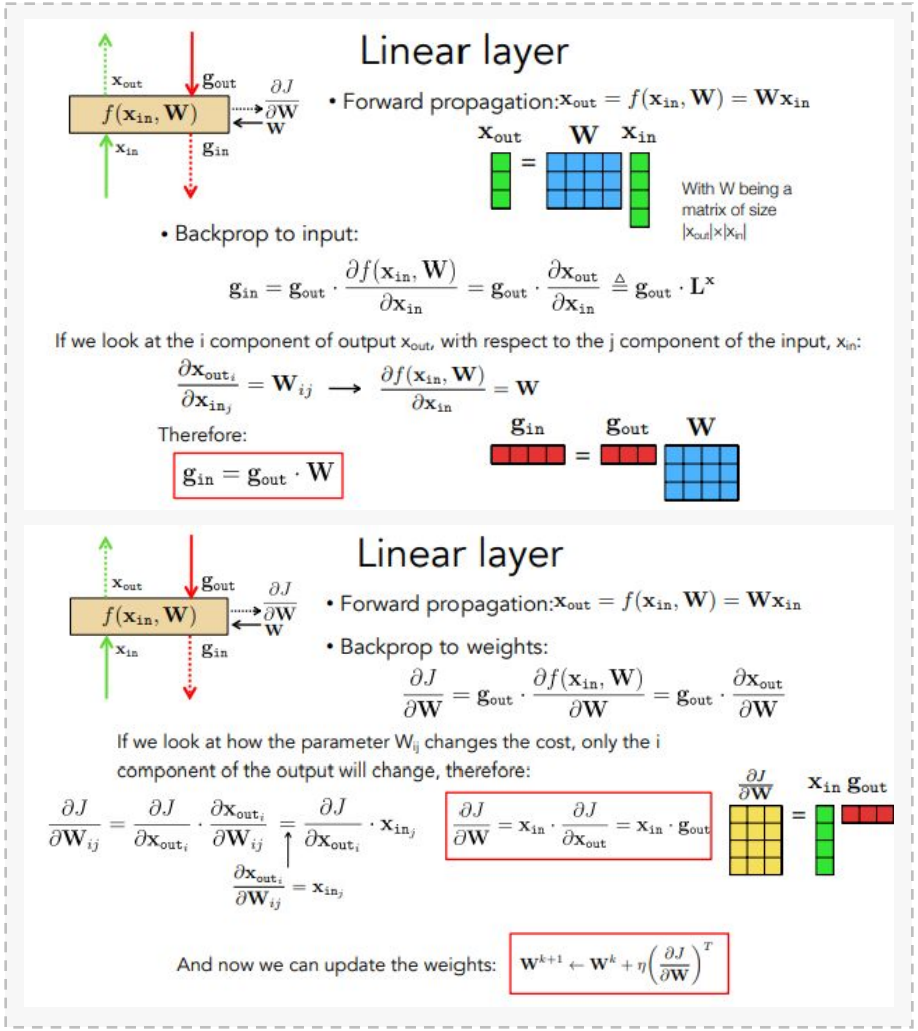
$$J = \frac{1}{N}\sum_{i=1}^{N} J_i(\mathbf{x}^i, \theta)$$

$$\frac{\partial J}{\partial \theta} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial J_i(\mathbf{x}^i, \theta)}{\partial \theta}$$

4. Layer별 Backward의 실체

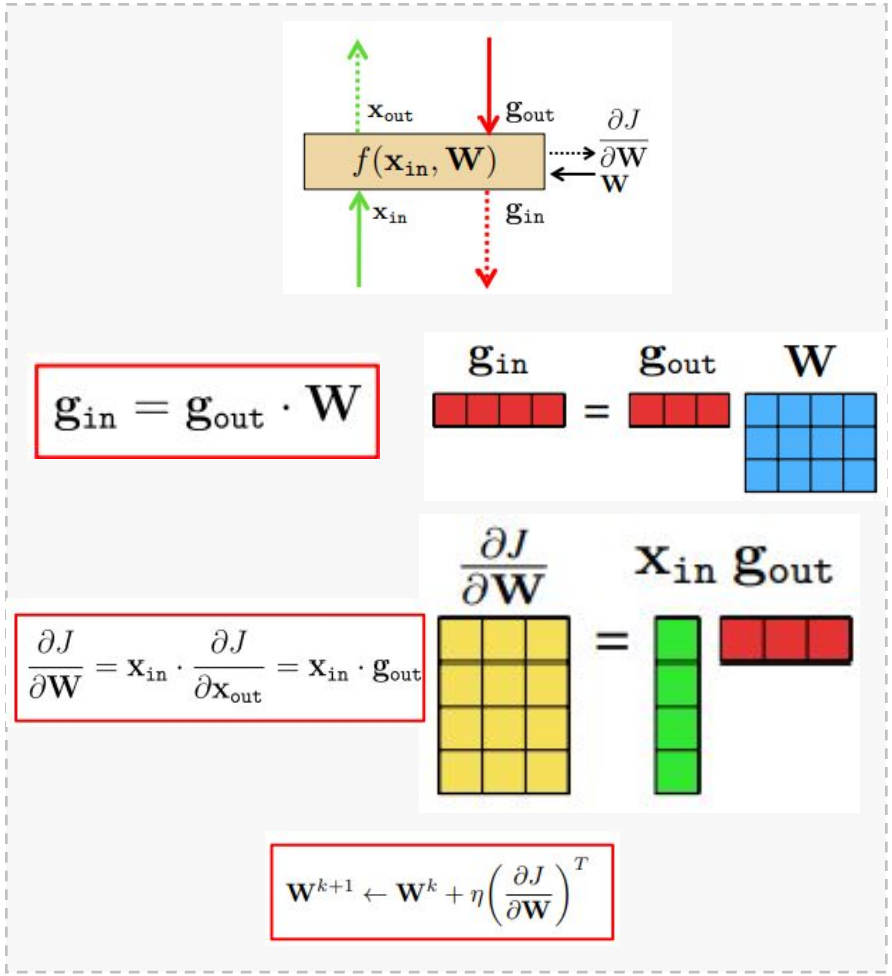
핵심 개념

- Linear layer : forward는 x_out = W x_in
  - weight gradient : ∂J/∂W = x_in g_out
  - input backward : g_in = g_out W
- ReLU 같은 pointwise nonlinearity는 diagonal Jacobian을 가짐



Linear backward는 forward의 W를 transpose해 재사용

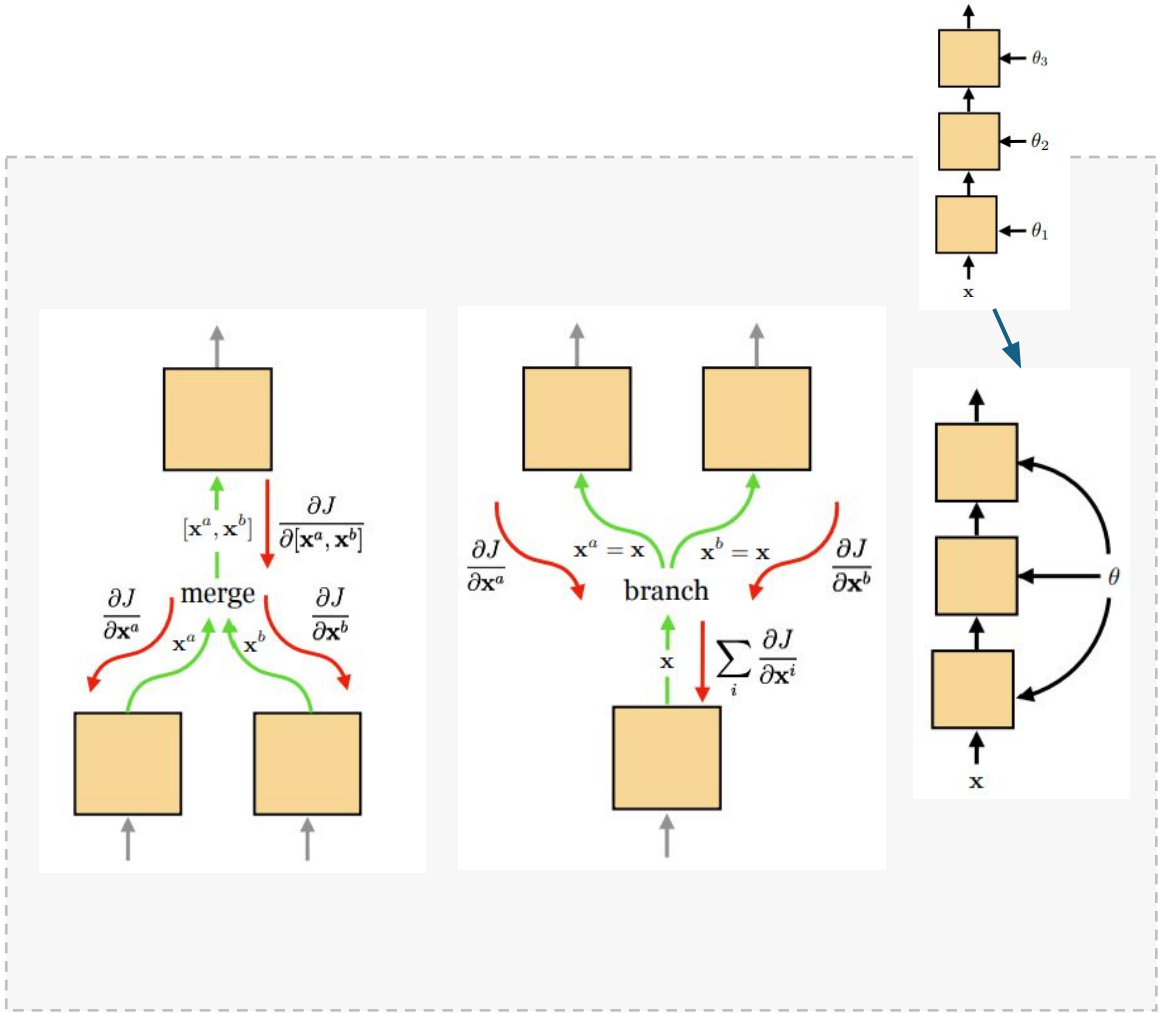
- ‘input을 다음 layer로 미는 연산’과 ‘error signal을 이전 layer로 미는 연산’이 **같은 행렬 W를 방향만 바꿔 재사용**
- weight update 식에서도 $(\partial J/\partial W)^T$ 처럼 transpose가 일관되게 등장



5. DAG 확장: Merge / Branch / Parameter Sharing

핵심 개념

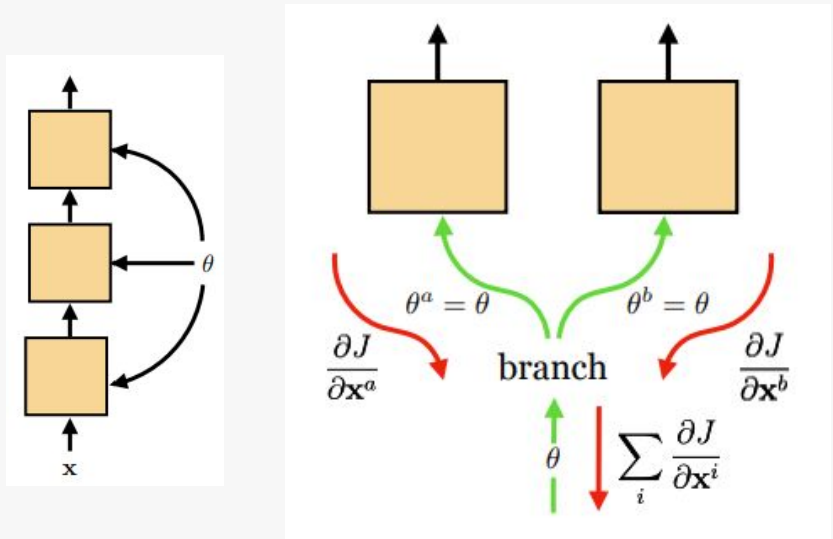
- 실제 네트워크는 단순한 chain이 아니라 **DAG 구조 (방향성 비순환 그래프)**
- 여러 입력이 merge되거나([x_a, x_b]) 하나의 값이 branch됨(x_a=x, x_b=x)
- 같은 parameter θ가 여러 경로에서 공유될 수도 있음(parameter sharing)



5. DAG 확장 · 잘 모르는 지점 ①

Parameter sharing의 backward 규칙 = 들어오는 gradient를 단순히 더한다

- Branch(x→x_a, x_b)와 parameter sharing(θ→θ_a, θ_b)은 구조적으로 동일하게 취급
- backward 시 각 경로에서 온 gradient를 합산하면 끝
- weight tying, RNN의 시간축 공유, CNN의 커널 공유가 모두 이 한 규칙으로 설명



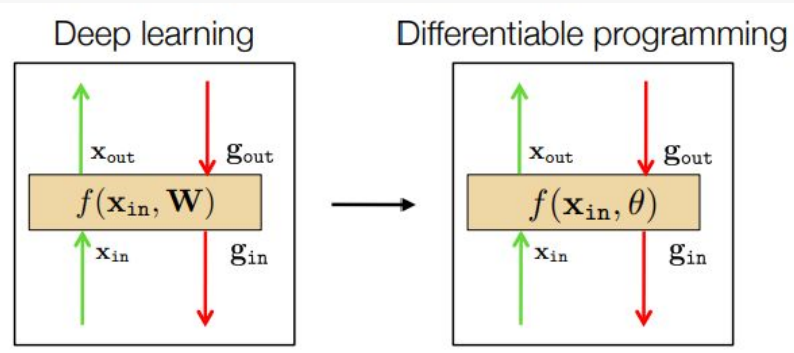
6. Differentiable Programming으로의 확장

핵심 개념





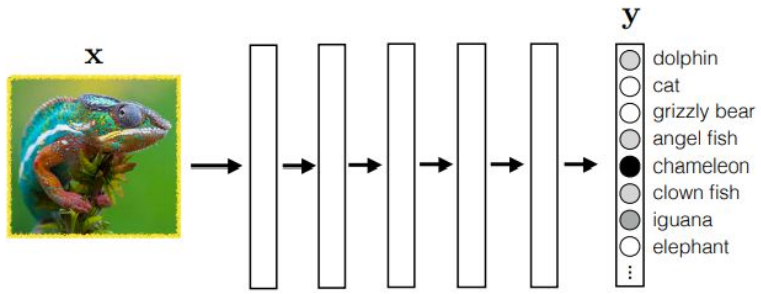
- Backprop은 parameter θ뿐 아니라 computation graph의 어떤 node(function)나 edge(variable) 에 대해서도 scalar cost를 최적화 가능
- Deep Learning을 전통적 코딩에 적용하면 Differentiable Programming이 됨
- **Differentiable Programming** : 프로그램 전체를 하나의 거대한 신경망처럼 취급하여, 프로그램 내부의 모든 매개변수를 제어 흐름(조건문, 반복문 등)과 관계없이 자동 미분(Automatic Differentiation)을 통해 최적화할 수 있도록 만드는 패러다임




```
for i, data in enumerate(dataset):
  iter_start_time = time.time()
  if total_steps % opt.print_freq == 0:
      t_data = iter_start_time - iter_data_time
  visualizer.reset()
  total_steps += opt.batch_size
  epoch_iter += opt.batch_size
  model.set_input(data)
  model.optimize_parameters()
```

예시1: Feature Visualization

Optimizing parameters versus optimizing inputs



$\frac{\partial y_j}{\partial \mathbf{x}}$ ← How much the "chameleon" score is increased or decreased by changing the image pixels.

Unit visualization

$$\arg\max_{\mathbf{x}} y_j + \lambda R(\mathbf{x})$$

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \eta \frac{\partial (y_j(\mathbf{x}) + \lambda R(\mathbf{x}))}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^k}$$

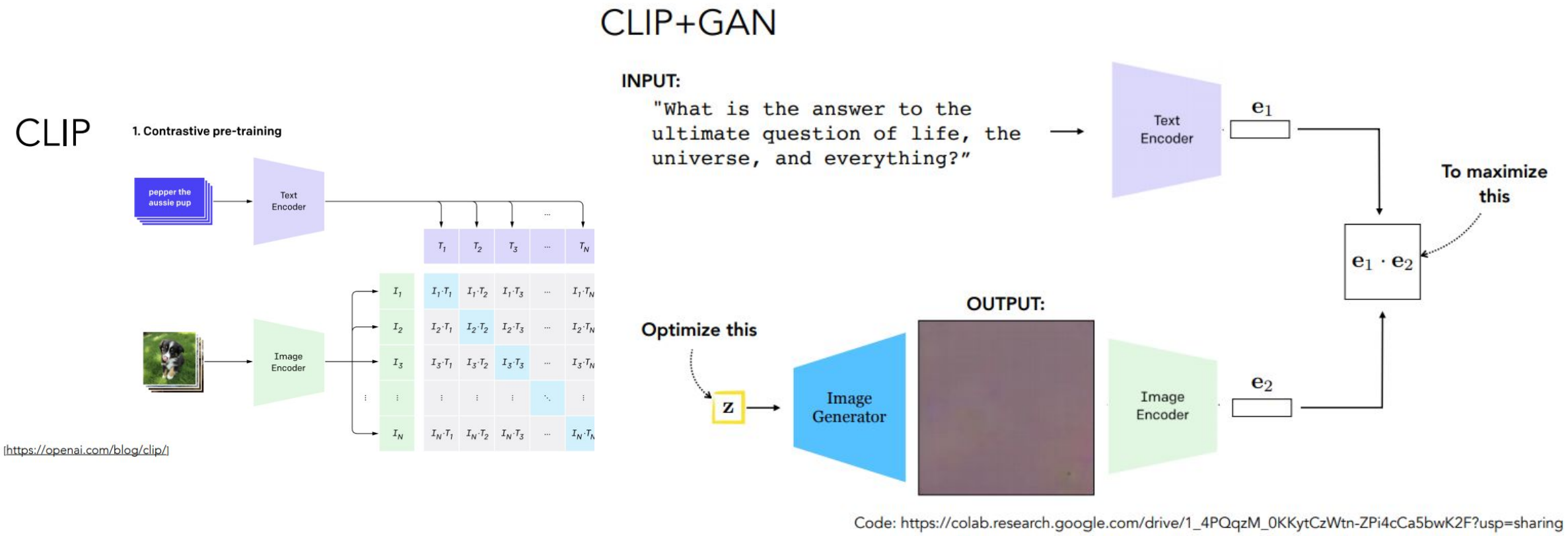
Make an image that maximizes the "cat" output neuron:



[https://distill.pub/2017/feature-visualization/]

Link!

예시2: CLIP+GAN(생성적 적대 신경망)



Link!

부록

Backpropagation이 필요한 이유 — 연산량 비교

- 목표: scalar loss의 gradient를 **모든 parameter** 에 대해 한 번에 구하기
- 수치미분 · forward-mode: parameter 1개당 forward pass 1회 필요 → 비용이 parameter 수 P에 비례
- Backpropagation(reverse-mode): 공유 항(error signal)을 한 번만 역전파 → forward의 **상수배 비용, P와 무관**

- 예시 망: MLP 784 → 100 → 10 (MNIST), P ≈ 79,500, forward 1회 ≈ 159K FLOP

방식	gradient 1회 계산	이 예시
수치미분 $\frac{L(w+\epsilon)-L(w)}{\epsilon}$	forward pass × (P+1)회	≈ 79,500회
Backpropagation $\frac{\partial L}{\partial w_n} = \frac{\partial L}{\partial w_{n+1}} \cdot \frac{\partial w_{n+1}}{\partial w_n}$	forward + backward ≈ 2~3회	≈ 3회
결과: 약 26,000배 빠름 — parameter가 늘수록 격차는 더 벌어진다		