Deep Learning

Statistical Learning for Big Data

2025

Questions

https://slides.app.goo.gl/39U4q

What is Deep Learning?

Artificial Intelligence

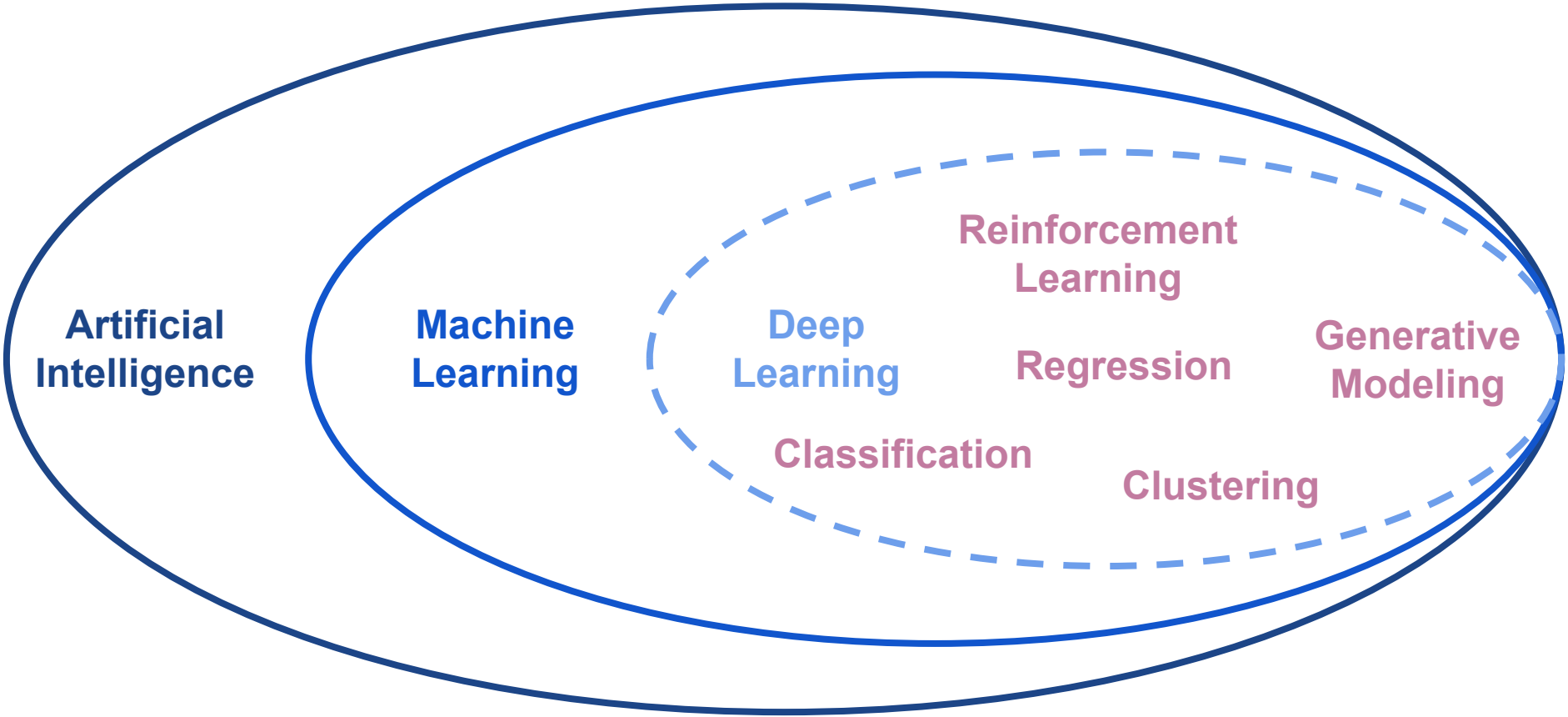
Rule-Based

Image Source

Pattern Matching

https://www.cleverbot.com/

What is Deep Learning?



Deep Learning = Machine Learning + Neural Networks

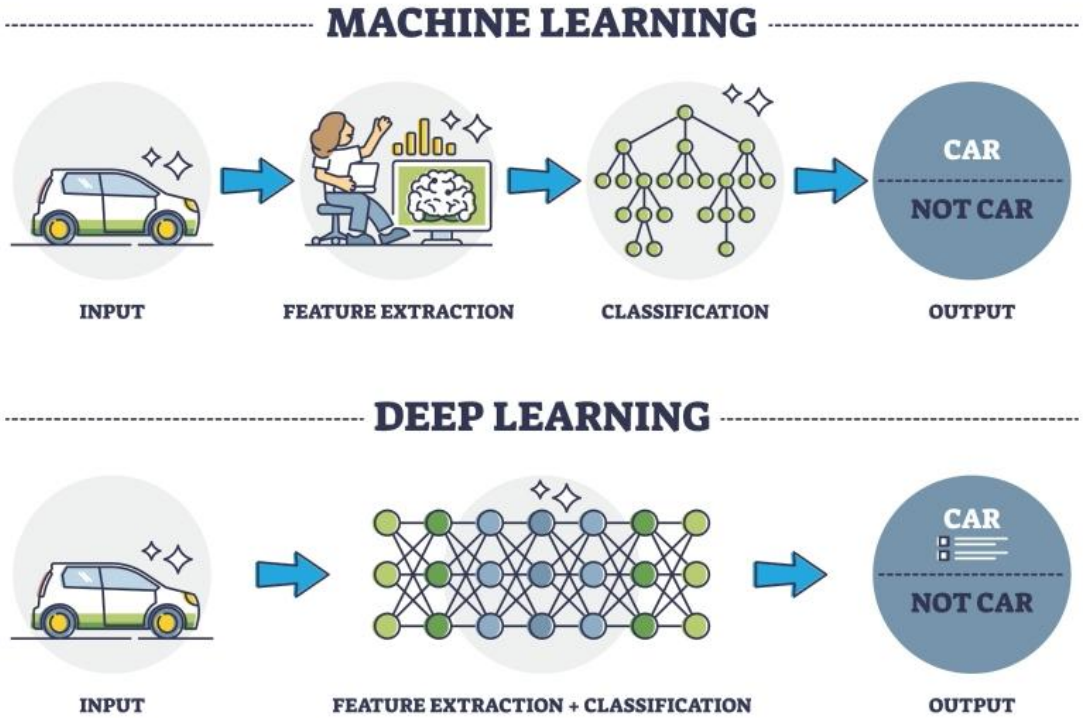
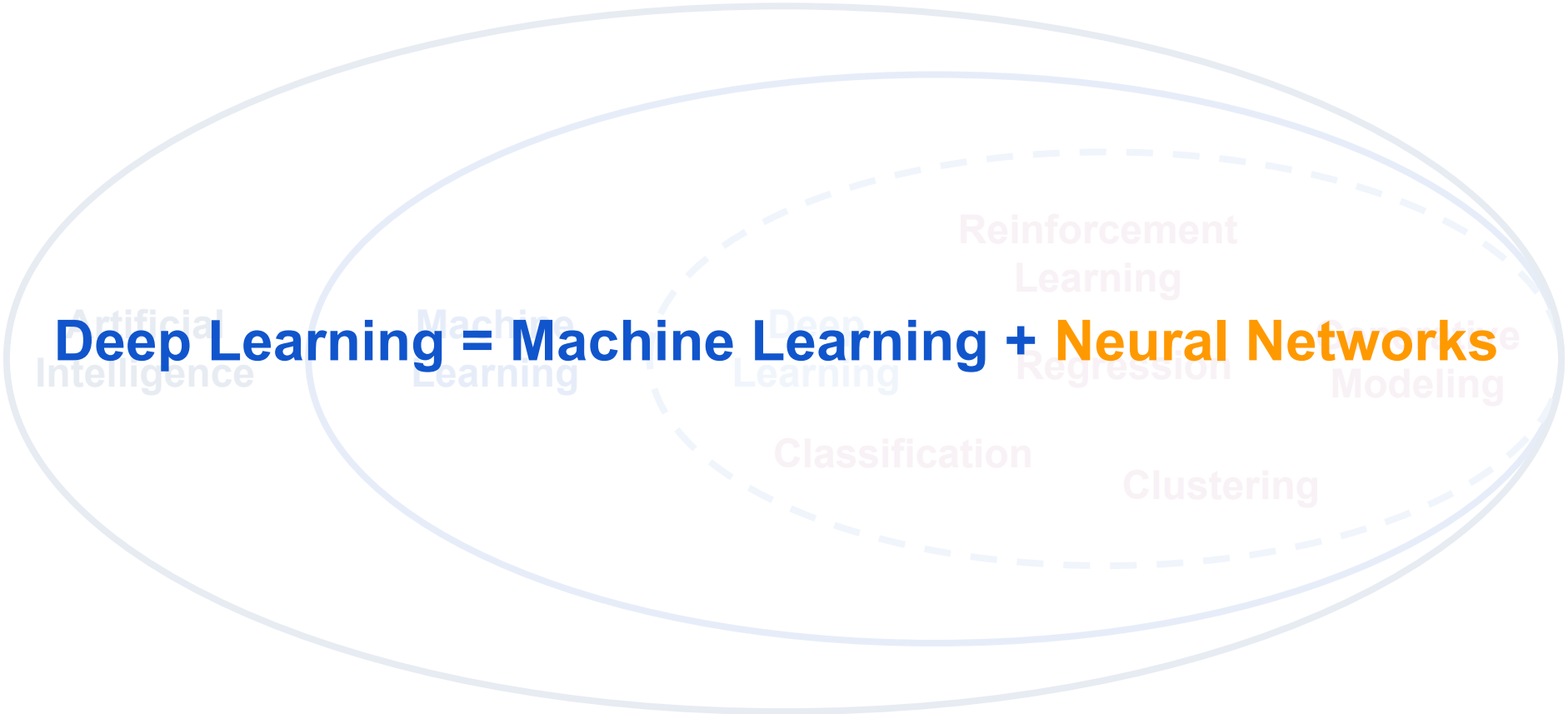


Image Source


Deep Learning = Machine Learning + Neural Networks
Reinforcement
Learning
Classification
Clustering
Modeling

Why Deep Learning?

- No expert knowledge required for feature extraction
- Better typical performance on highly complex data
- Allows modeling nonlinear relationships easily
- Scalability to Big Data

What you will learn

- Individual parts of neural networks
- Building your own neural network from scratch
- Training neural networks reliably and efficiently
- Modern neural network architectures
- State-of-the-art applications for deep learning
- Limits of deep learning and current research

Notation

- N = Number of Samples
- D = Number of Features
- x = Vector(s) of input features
- z = Vector(s) of latent features
- θ = Network Parameters
- $\mathcal{L}$ = Loss Function

This is a Hands-On Lecture

NumPy

Quickstart Tutorial

Deep Learning Frameworks

- PyTorch
  - Developed by Meta
  - Focus on research development
  - JIT Compilation is WIP
  - Easiest to Learn, but usually slowest
- JAX
  - Developed by Google
  - Purely functional, numpy-style API
  - Models are JIT compiled
  - Scales automatically to Multi-GPU
  - Hardest to learn, but usually fastest
- TensorFlow
  - Developed by Google
  - Focus on production environments
  - Compilation is usually static
- Keras
  - Developed by Google
  - High-level API usable with any backend

PyTorch

- Deep Learning revolves around “Tensors”
- “Tensors” are multi-dimensional arrays
  - Like numpy arrays
- Tensors:
  - Can record gradients
  - Can be stored on the GPU

```python
import torch

x = torch.zeros((32, 2))

assert x.shape == (32, 2)
assert str(x.device) == "cpu"
assert x.grad is None
```

PyTorch

- Tensors have to be stored on the same device to do arithmetic

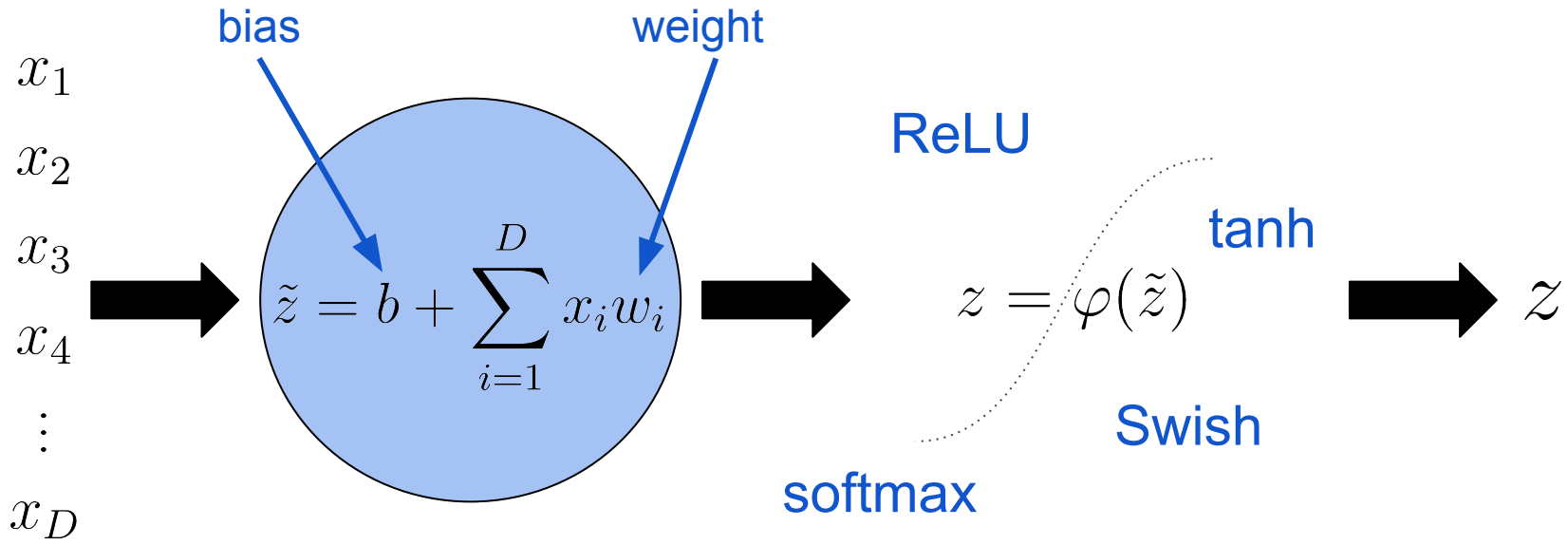
```python
import torch

x = torch.zeros((32, 2), device="cpu")
y = torch.zeros((32, 2), device="cuda")

z = x + y  # RuntimeError: Different Devices
```

Using the GPU is optional

for this lecture

Neural Networks

A Single Neuron



D-Dimensional Input

Linear Projection

What pattern does the neuron look for?

Non-Linear Activation

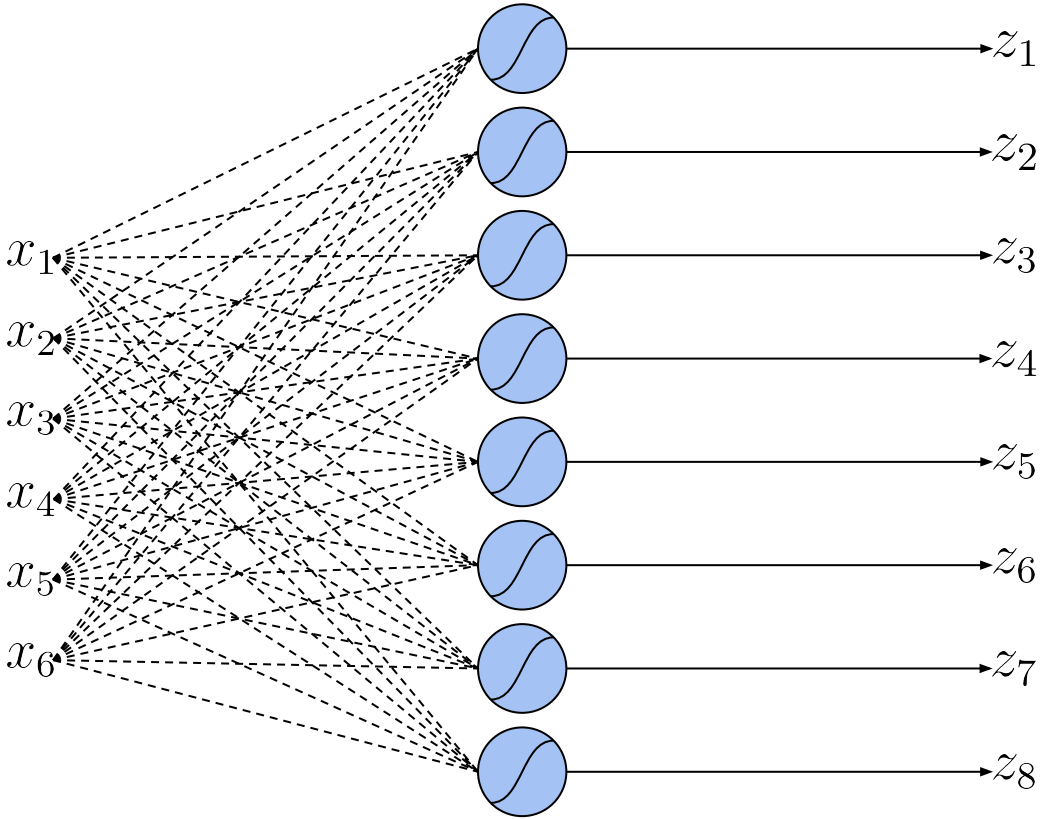
When does the neuron “fire”?

Single Output

Perceptron

$$z_i = \varphi\left(\sum_{j=1}^{D} x_j W_{i,j} + b_i\right)$$

$$z = \varphi\left(xW^T + b\right)$$

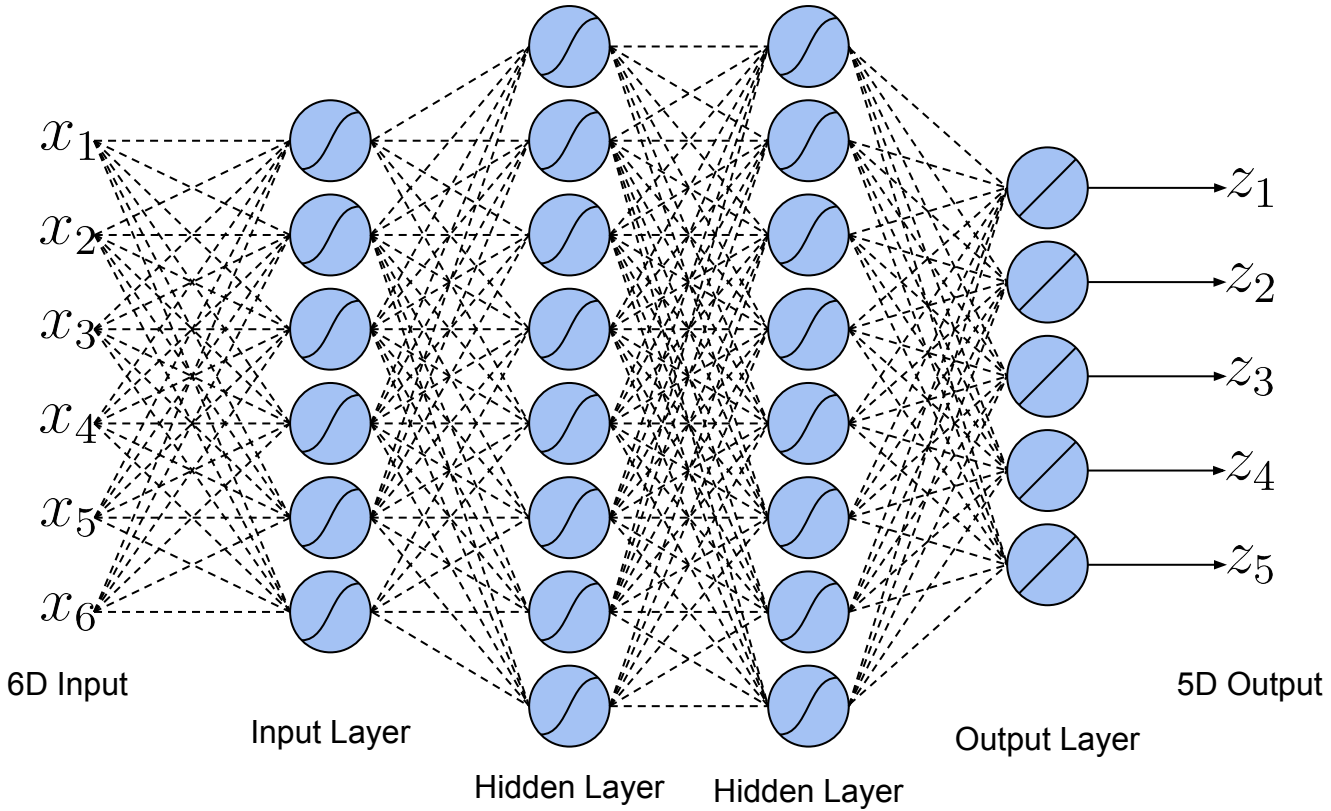


Perceptron

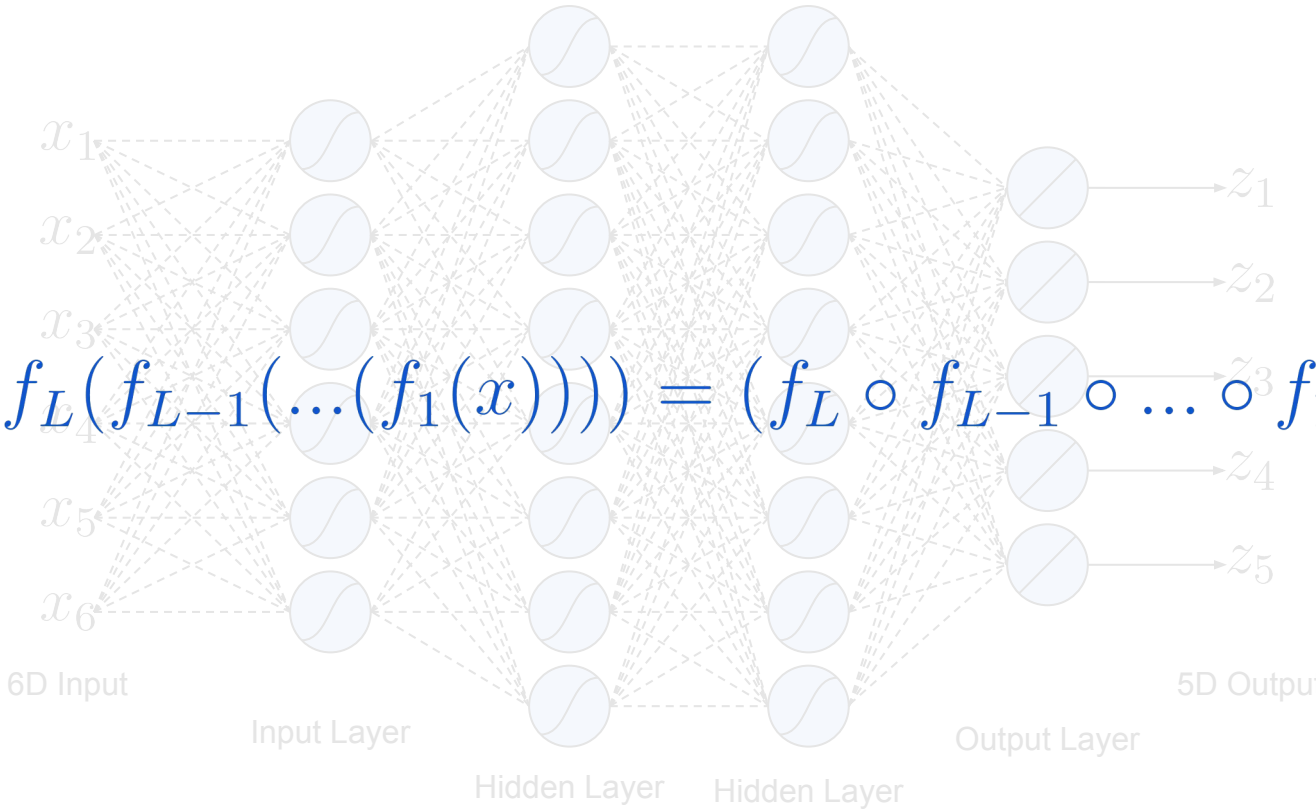
```python
import torch.nn as nn

input_features = 6
output_features = 8

linear_transform = nn.Linear(input_features, output_features)
activation = nn.Tanh()

perceptron = nn.Sequential(linear_transform, activation)
```

```python
import torch

batch_size = 32

x = torch.zeros((batch_size, input_features))

z = perceptron(x)

assert z.shape == (batch_size, output_features)
```

Multi-Layer Perceptron (MLP)



Multi-Layer Perceptron (MLP)



$$z = f_L(f_{L-1}(\ldots(f_1(x)))) = (f_L \circ f_{L-1} \circ \ldots \circ f_1)(x)$$

Multi-Layer Perceptron (MLP)

```python
import torch.nn as nn

in_features = 6
h1 = 128
h2 = 256
h3 = 128
out_features = 5

mlp = nn.Sequential(
    nn.Linear(in_features, h1),
    nn.ReLU(),
    nn.Linear(h1, h2),
    nn.ReLU(),
    nn.Linear(h2, h3),
    nn.ReLU(),
    nn.Linear(h3, out_features)
)
```

Always identical

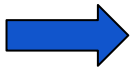
Output is usually not activated

Why not just linear activations?

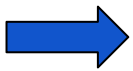
Training

Iterative Optimization

- Stochastic Gradient Descent
  - Use first derivative of loss
  - Many steps to converge: $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$
  - Individual steps are cheap: $\mathcal{O}(D)$
- Newton's Method
  - Use second derivative of loss
  - Few steps to converge: $\mathcal{O}\left(\log\log\frac{1}{\varepsilon}\right)$
  - Individual steps are slow: $\mathcal{O}(ND^2)$
- Dual Methods
  - Use first and second derivative of loss

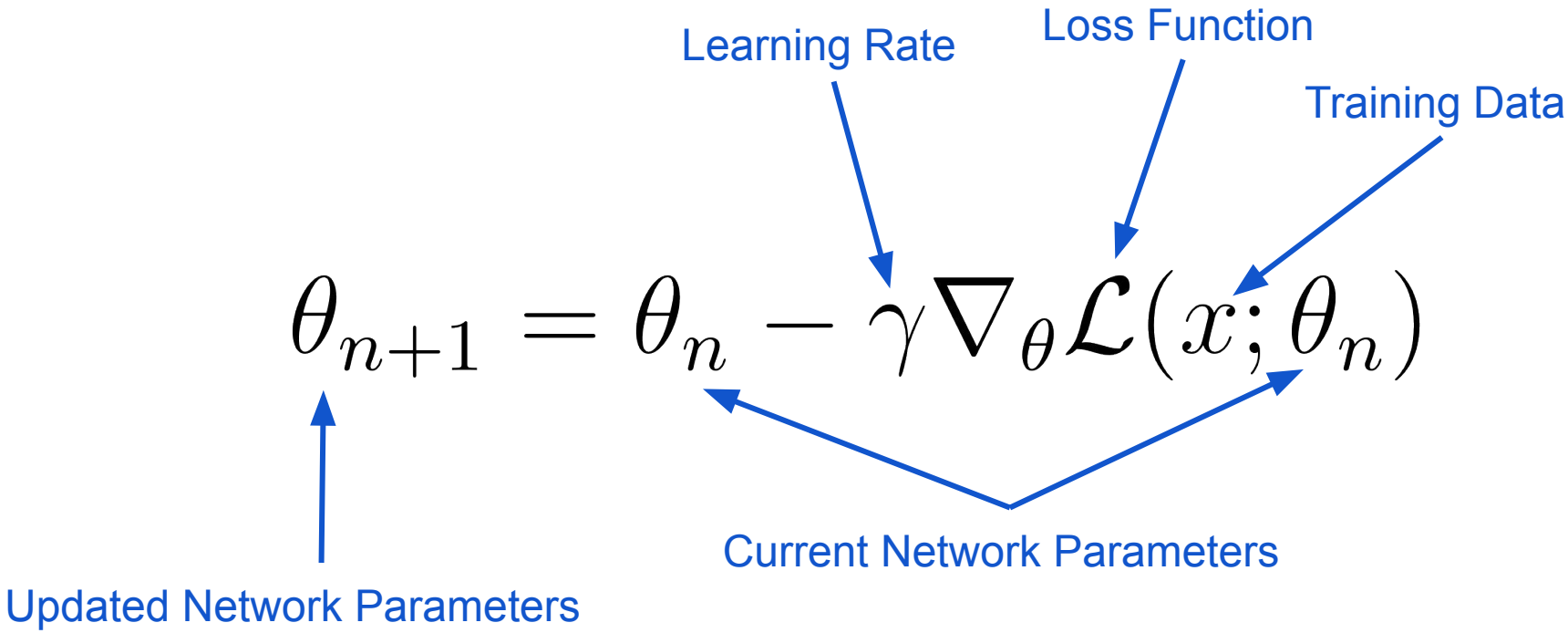


Best for big data



Best for small data

Gradient Descent



Gradient Descent

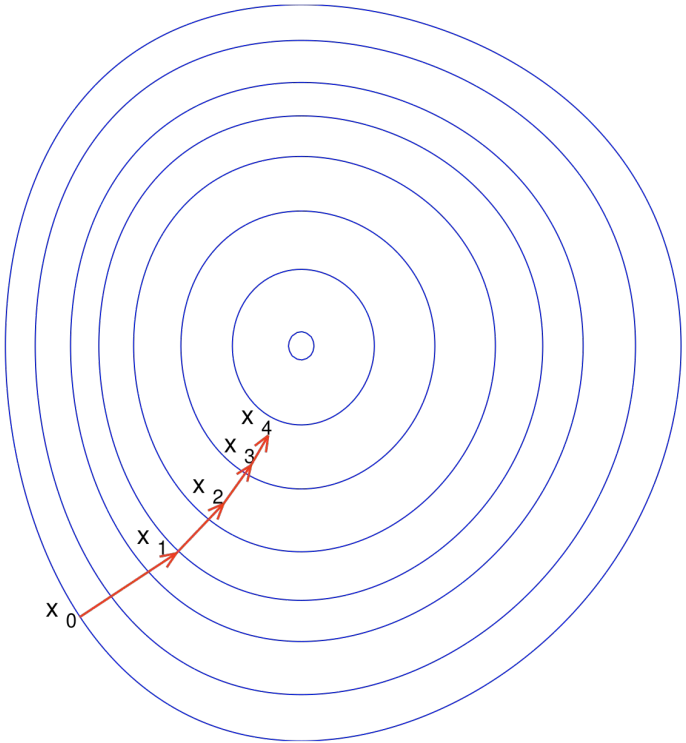
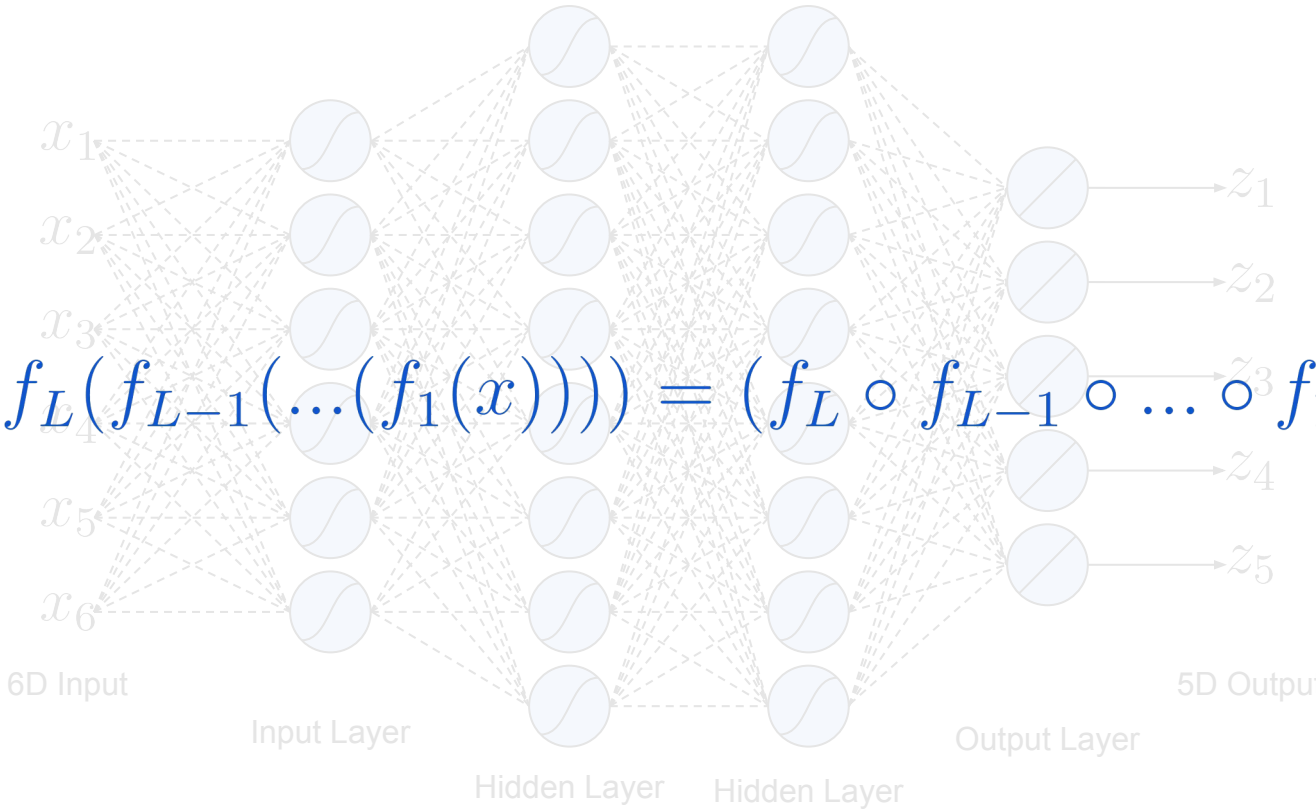


Image Source

Multi-Layer Perceptron (MLP)

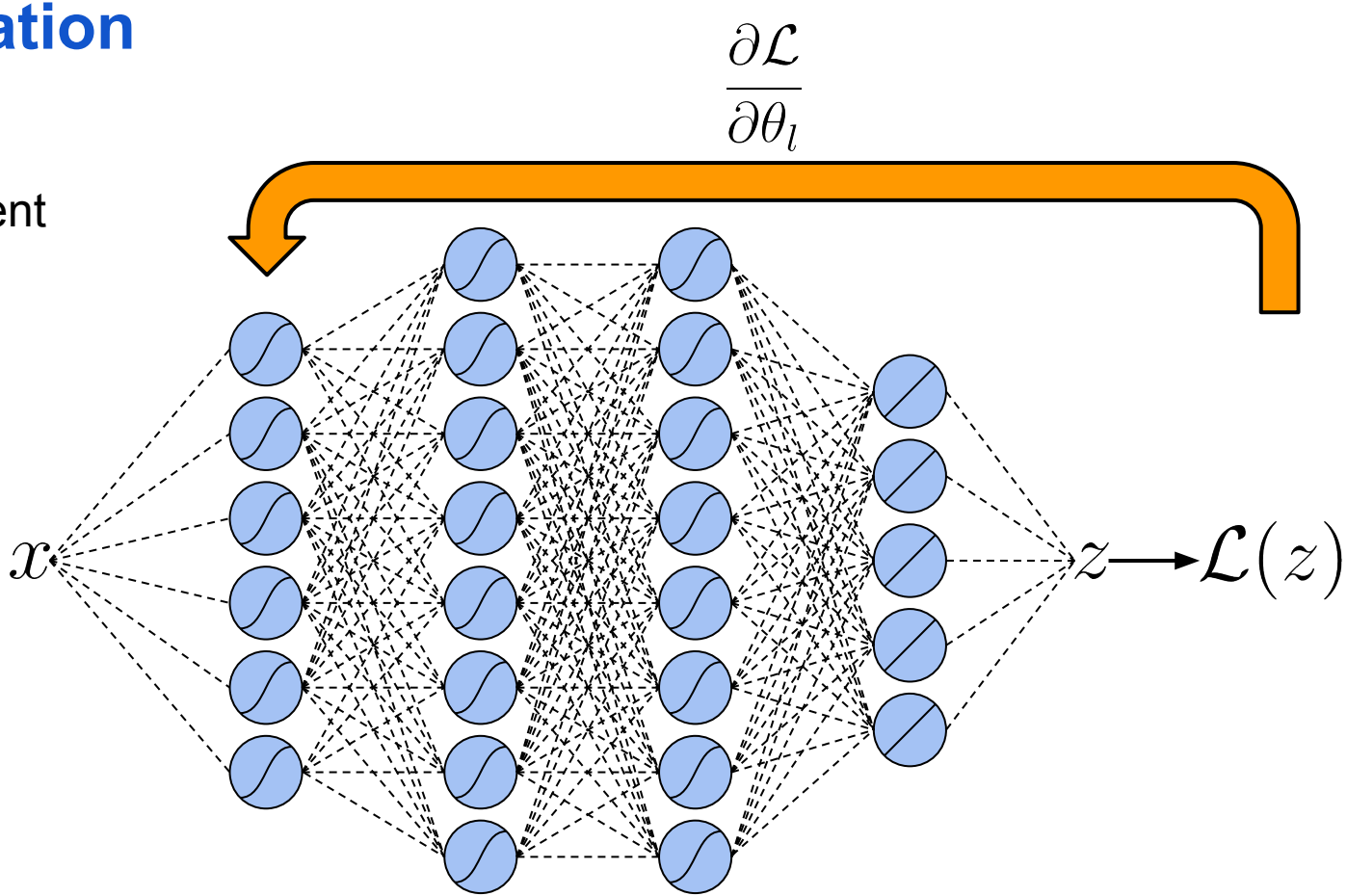


$$z = f_L(f_{L-1}(\ldots(f_1(x)))) = (f_L \circ f_{L-1} \circ \ldots \circ f_1)(x)$$

Backpropagation

To train with Gradient Descent, we need

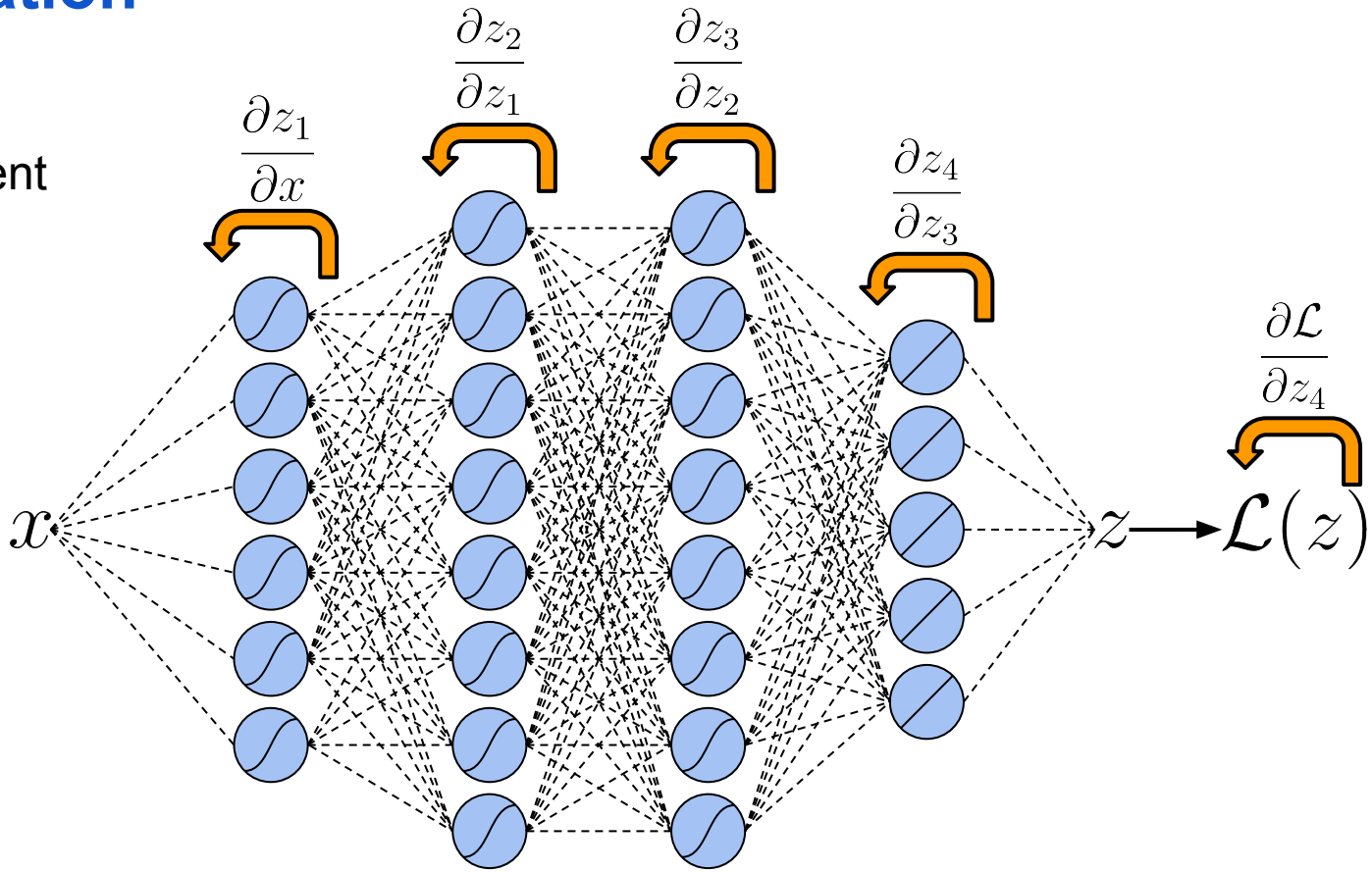
$$\nabla_\theta \mathcal{L}$$



Backpropagation

To train with Gradient Descent, we need

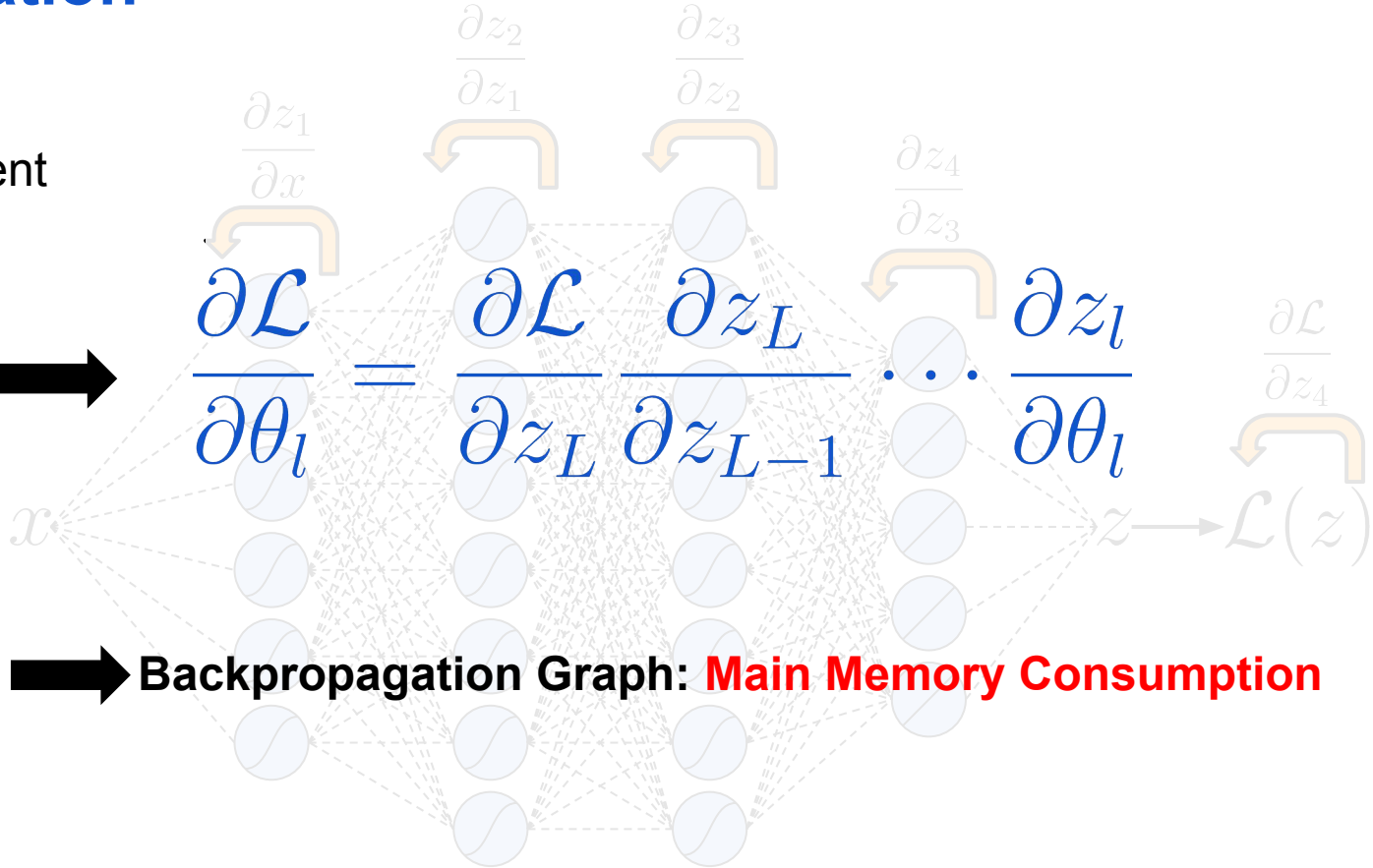
$$\nabla_\theta \mathcal{L}$$



Backpropagation

To train with Gradient Descent, we need

$$\nabla_\theta \mathcal{L}$$



Backpropagation Graph: Main Memory Consumption

Backpropagation Example

Backpropagation

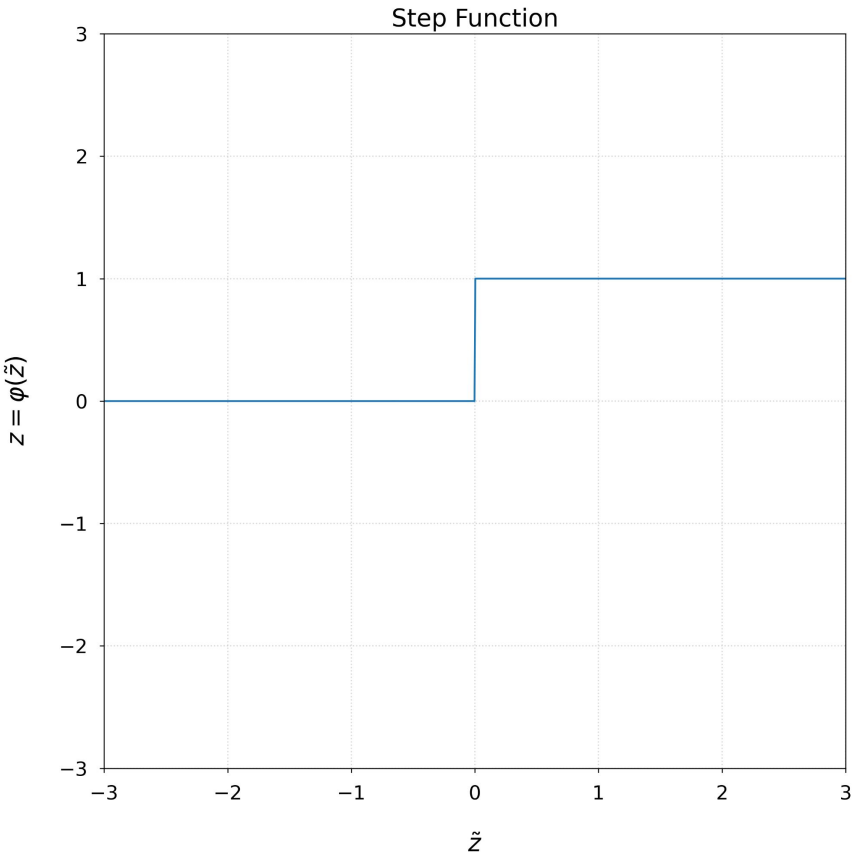
```
data, labels = ...

# build backpropagation graph
predictions = network(data)

assert predictions.requires_grad
assert predictions.grad_fn is not None

# compute scalar loss
loss = cross_entropy(predictions, labels)

# populate leaf gradients
loss.backward()
```

Backpropagation

https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

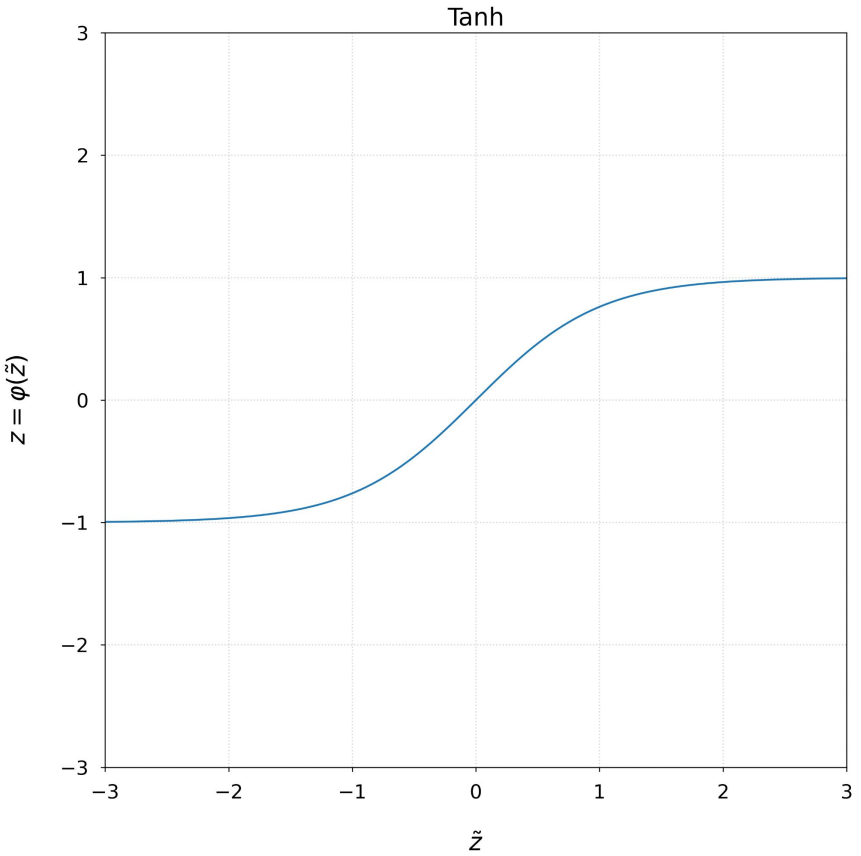
Activation Functions

$$\Theta(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$



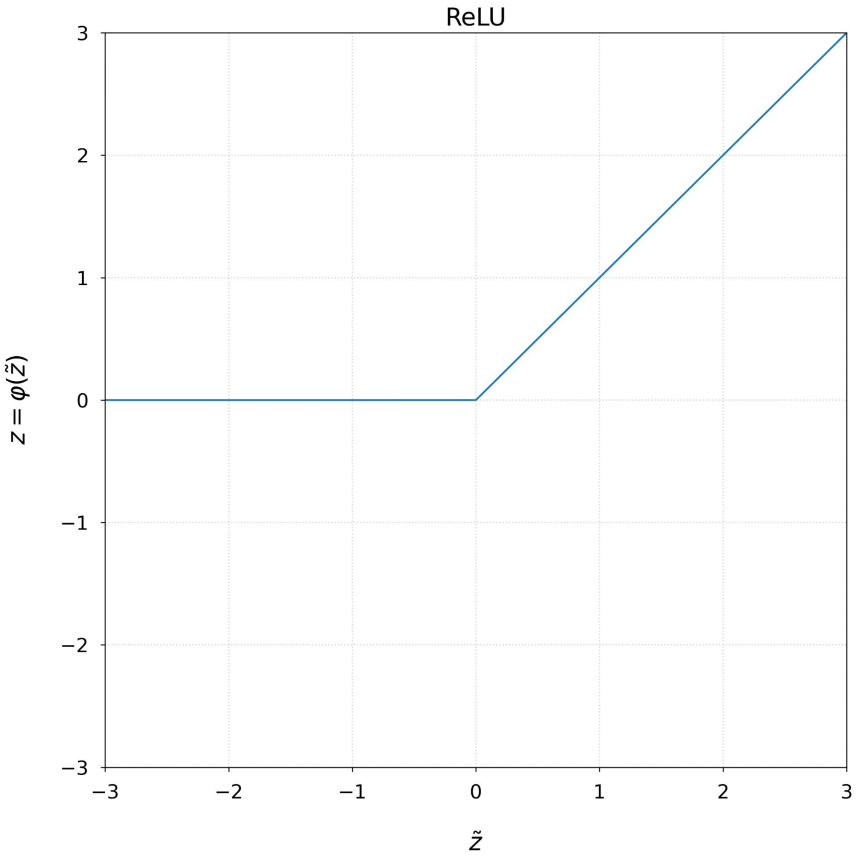
Activation Functions

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Activation Functions

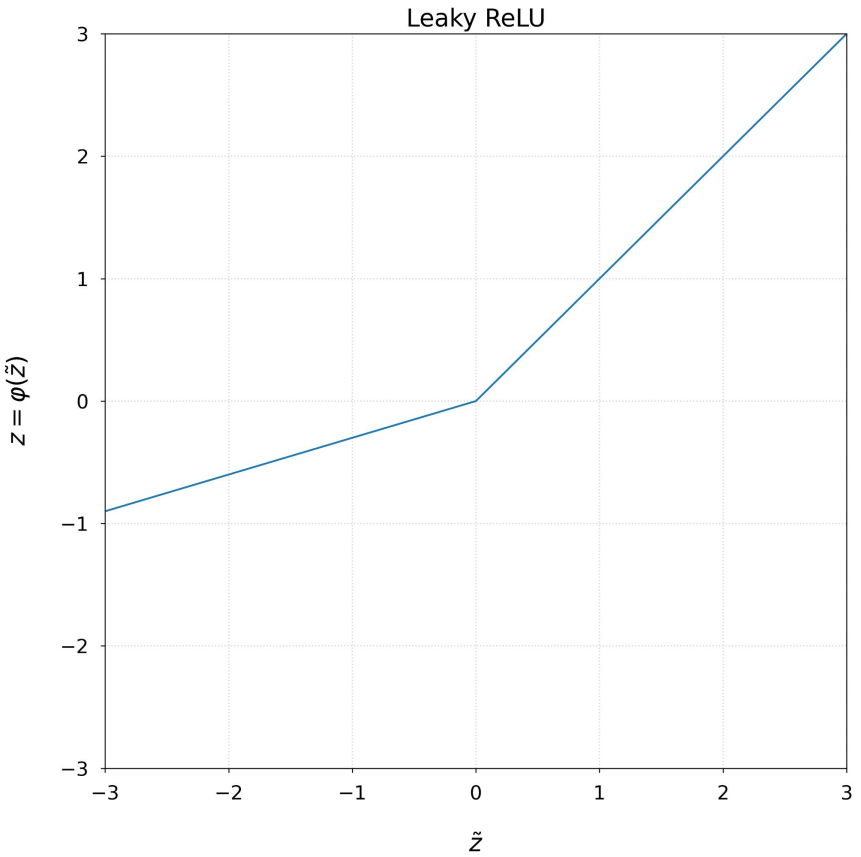
$$\text{ReLU}(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$



Agarap (2018) - Deep Learning using Rectified Linear Units (ReLU)

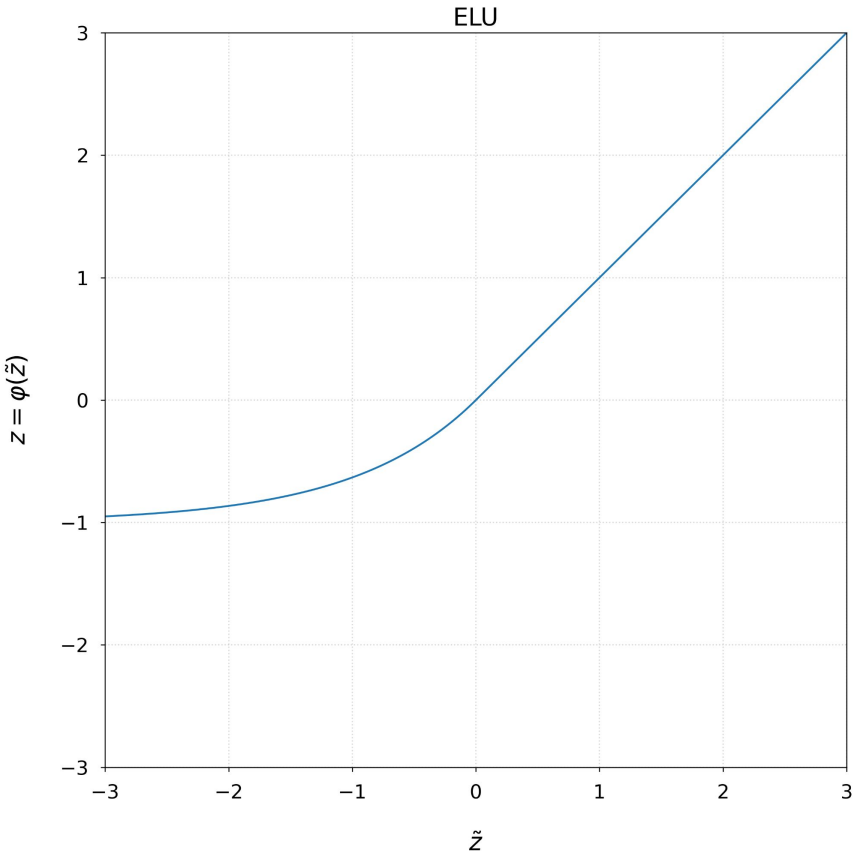
Activation Functions

$$\text{Leaky ReLU}(x) = \begin{cases} \alpha x, & x \leq 0 \\ x & x > 0 \end{cases}$$



Activation Functions

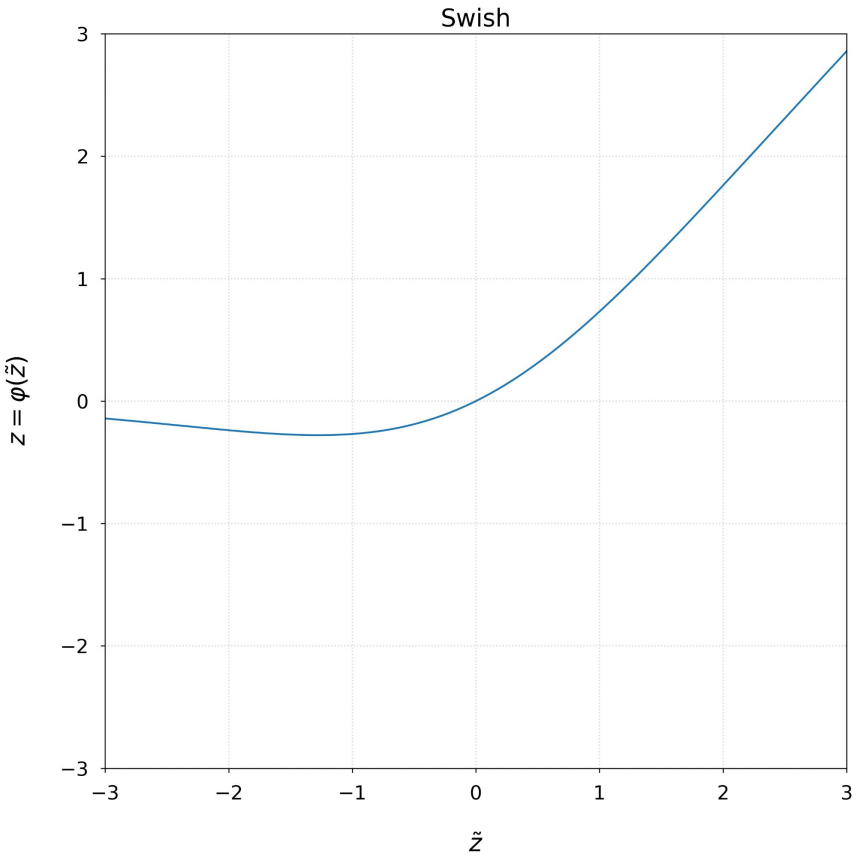
$$\mathrm{ELU}(x) = \begin{cases} \alpha\left(e^{x} - 1\right), & x \leq 0 \\ x, & x > 0 \end{cases}$$



Clevert et al. (2015) - Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

Activation Functions

$$\text{Swish}(x) = x\,\text{sigmoid}(x) = \frac{x}{1 + e^{-x}}$$



Ramachandran et al. (2017) - Searching for Activation Functions

Activation Functions

THREE DECADES OF ACTIVATIONS: A COMPREHENSIVE SURVEY OF 400 ACTIVATION FUNCTIONS FOR NEURAL NETWORKS

A PREPRINT

Vladimír Kunc
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
kuncvlad@fel.cvut.cz

Jiří Kléma
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
klema@fel.cvut.cz

February 15, 2024

ABSTRACT

Neural networks have proven to be a highly effective tool for solving complex problems in many areas of life. Recently, their importance and practical usability have further been reinforced with the advent of deep learning. One of the important conditions for the success of neural networks is the choice of an appropriate activation function introducing non-linearity into the model. Many types of these functions have been proposed in the literature in the past, but there is no single comprehensive source containing their exhaustive overview. The absence of this overview, even in our experience, leads to redundancy and the unintentional rediscovery of already existing activation functions. To bridge this gap, our paper presents an extensive survey involving 400 activation functions, which is several times larger in scale than previous surveys. Our comprehensive compilation also references these surveys; however, its main goal is to provide the most comprehensive overview and systematization of previously published activation functions with links to their original sources. The secondary aim is to update the current understanding of this family of functions.

Keywords adaptive activation functions, deep learning, neural networks

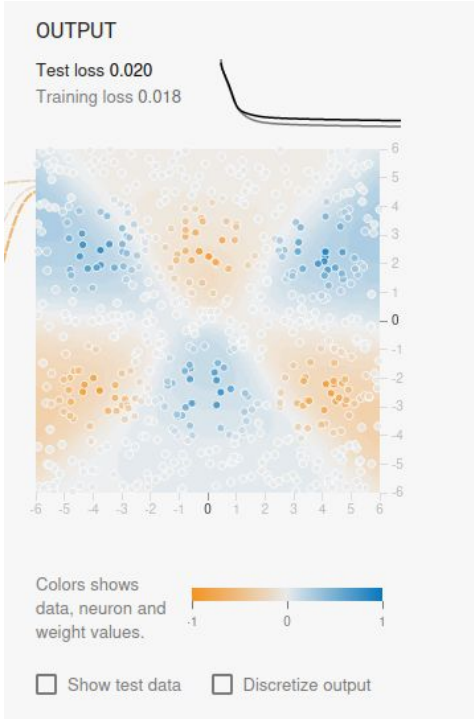
1 Introduction

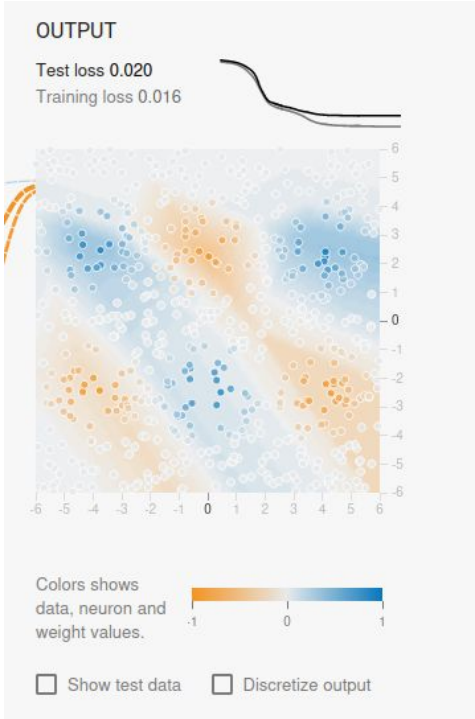
https://arxiv.org/abs/2402.09092

Using the right activation function is critical

https://playground.tensorflow.org

Tanh vs. ReLU


OUTPUT
Test loss 0.020
Training loss 0.018
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output


OUTPUT
Test loss 0.020
Training loss 0.016
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output

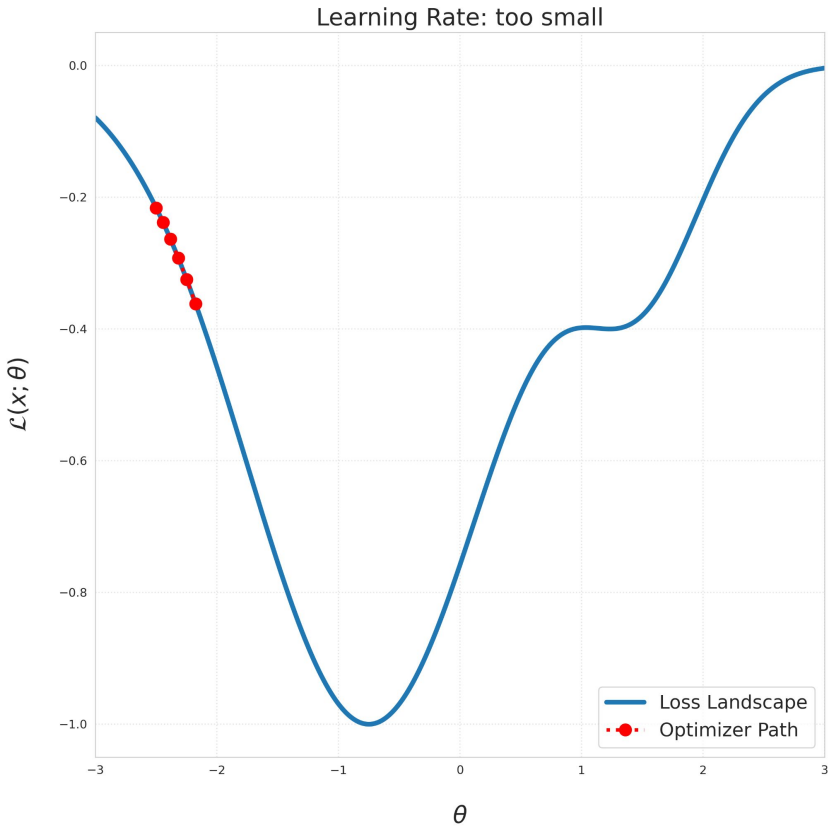
Learning rate is critical for fast convergence

Gradient Descent

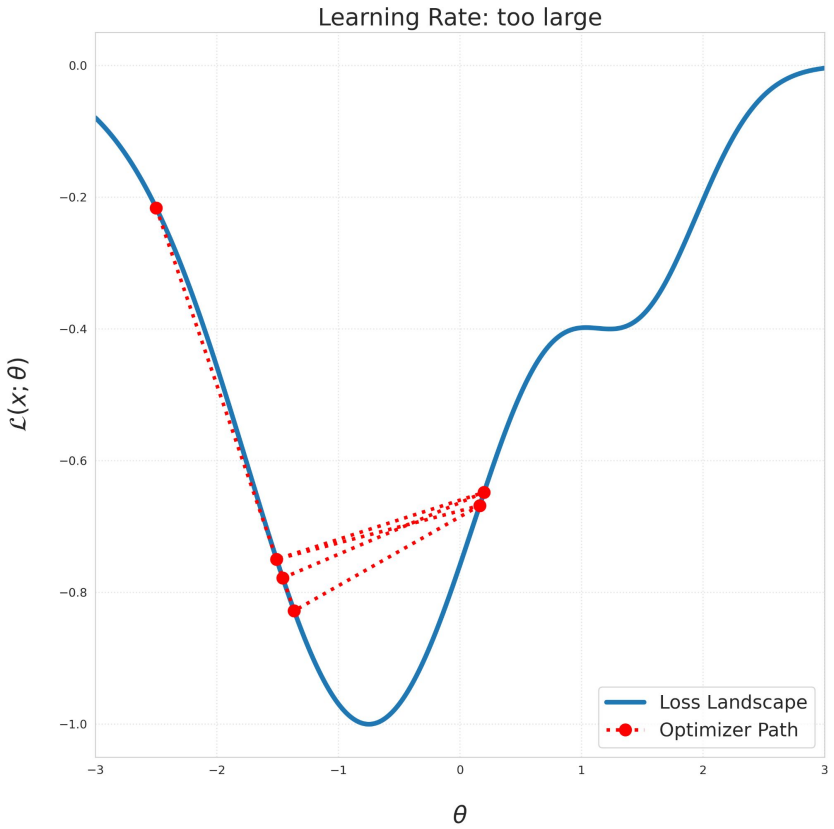
Learning Rate

$$\theta_{n+1} = \theta_n - \gamma \nabla_\theta \mathcal{L}(x; \theta_n)$$

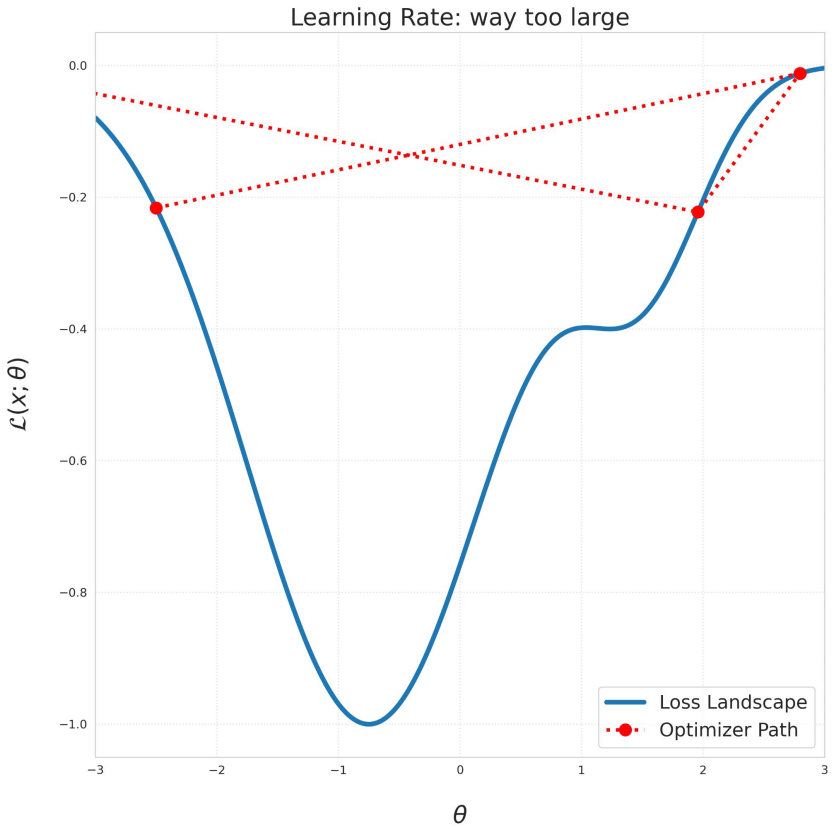
Gradient Descent



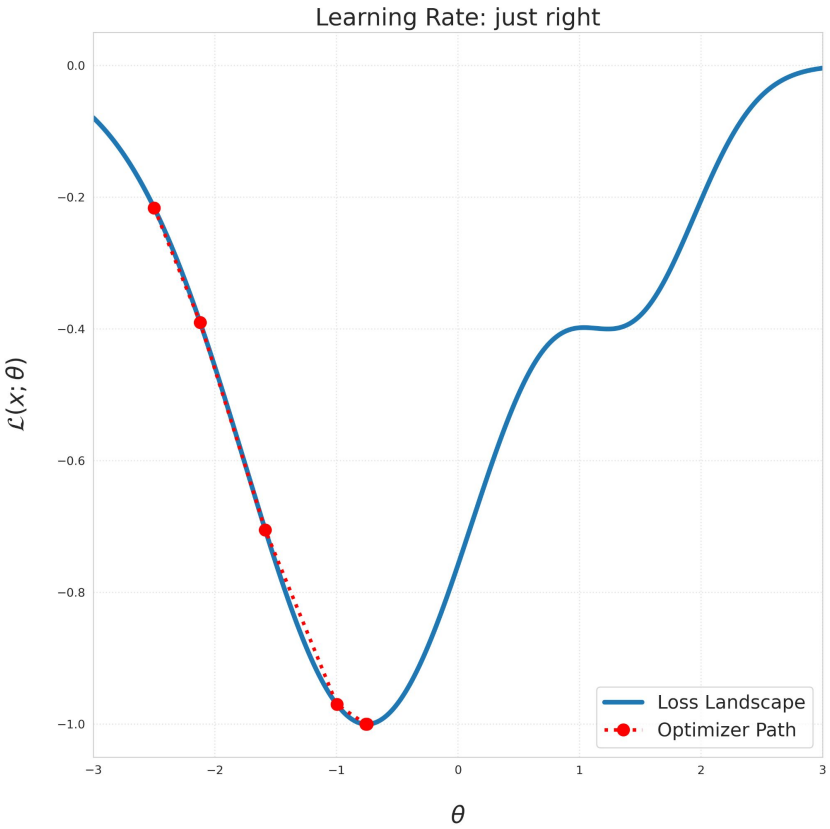
Gradient Descent



Gradient Descent



Gradient Descent



Stochastic Gradient Descent

- Datasets can be huge
  - e.g. LAION with 400M image-text pairs

- True Gradient Descent is $\mathcal{O}(ND)$

  ⇒ In practice: estimate gradient with mini-batches instead

Adaptive Gradient

- Plain SGD
  - Fixed learning rate for all parameters
  - Struggles with sparse gradients
- Fixed by Adaptive Gradient
  - Gives each parameter an individual learning rate
  - Adjusts learning rate based on multiple update steps
  - Improves convergence

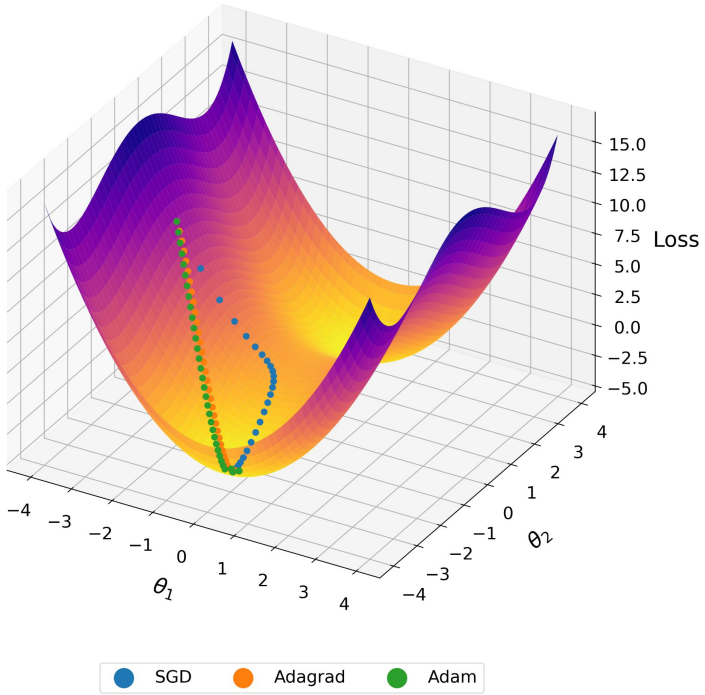
$$g_t = \nabla_{\theta_t} \mathcal{L}(x; \theta_t)$$

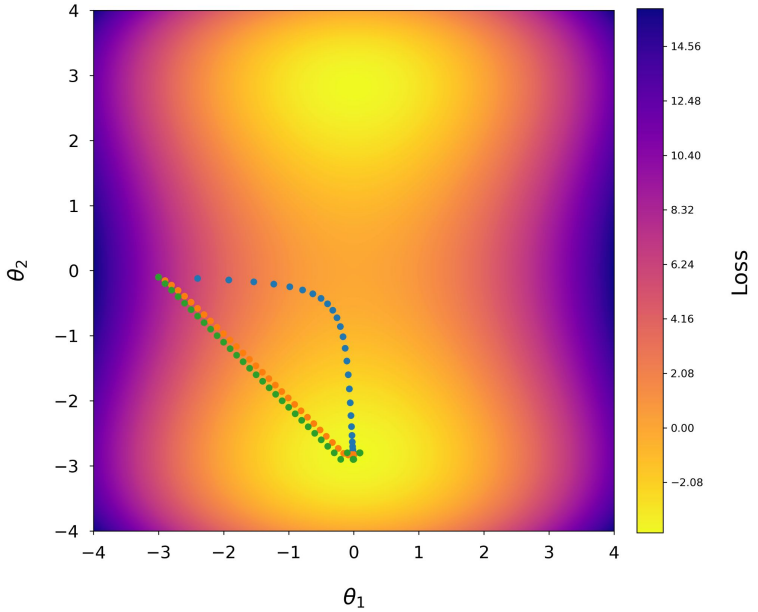
$$G_t = G_{t-1} + g_t^2$$

$$\theta_t = \theta_{t-1} + \frac{\eta}{\sqrt{G_t + \varepsilon}} g_t$$

Duchi et al. (2011) - Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

Adaptive Gradient





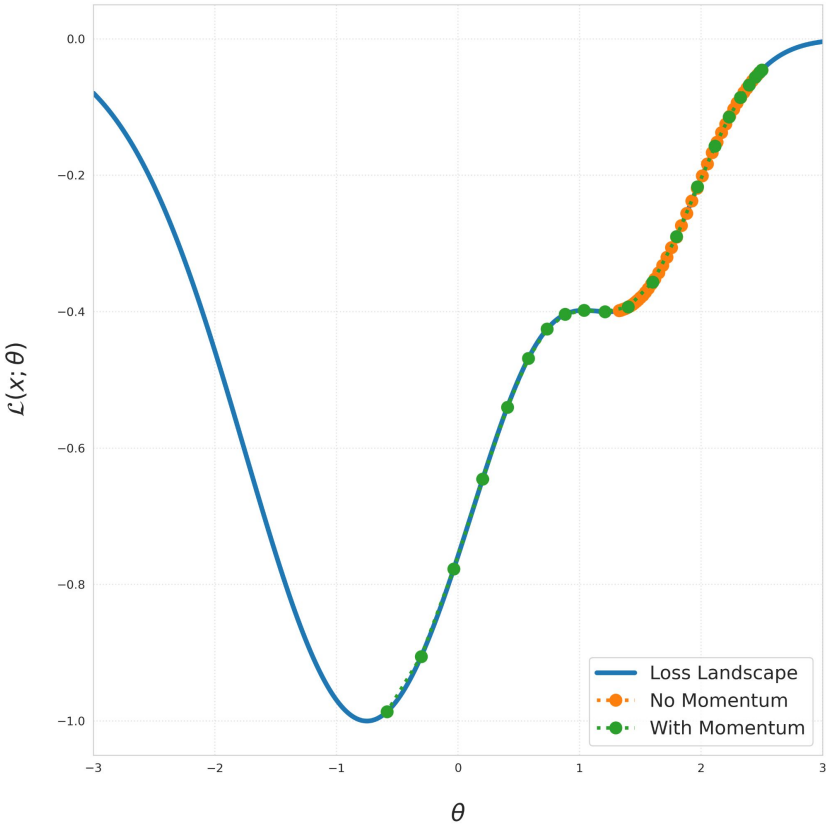
Momentum

- Plain SGD
  - Gets stuck in local minima
  - Is prone to oscillations
- Fixed by Momentum
  - Accumulates gradients over multiple steps
  - Reduces oscillations by cancellation
  - Smooths updates to parameters
  - Helps escape local minima

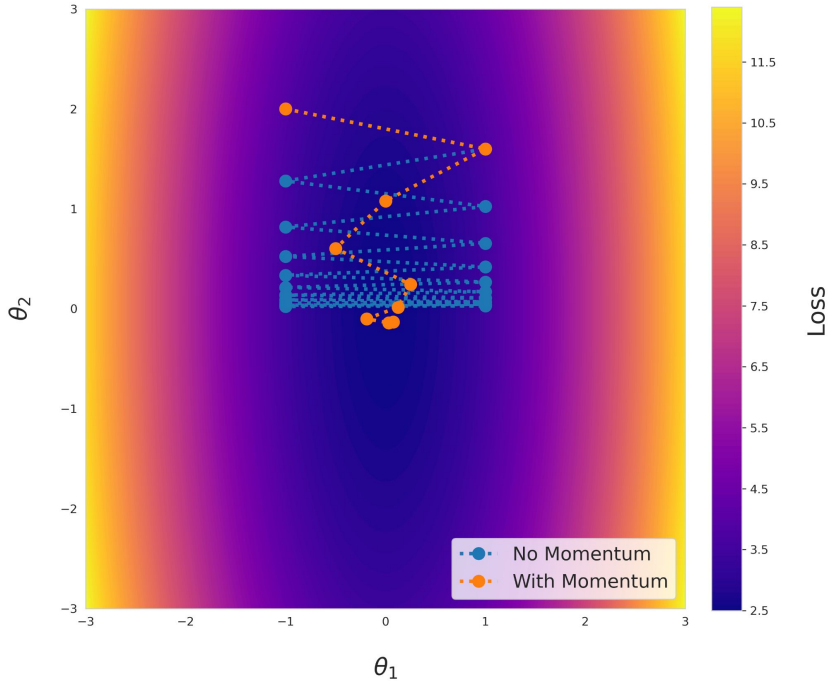
$$v_t = \beta v_{t-1} + (1-\beta)\nabla_{\theta_{t-1}}\mathcal{L}(x;\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - \gamma v_t$$

Momentum

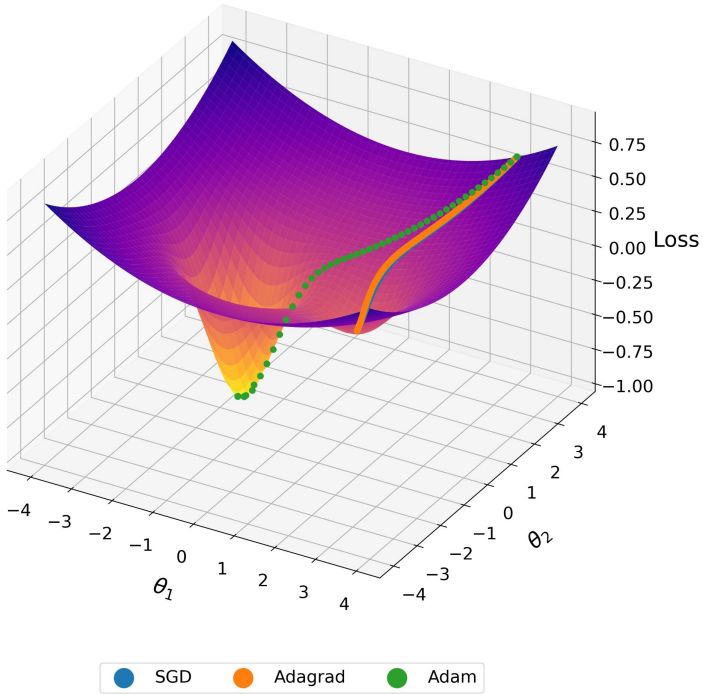

0.0
−0.2
−0.4
−0.6
−0.8
−1.0
$\mathcal{L}(x;\theta)$
−3
−2
−1
0
1
2
3
θ
Loss Landscape
No Momentum
With Momentum

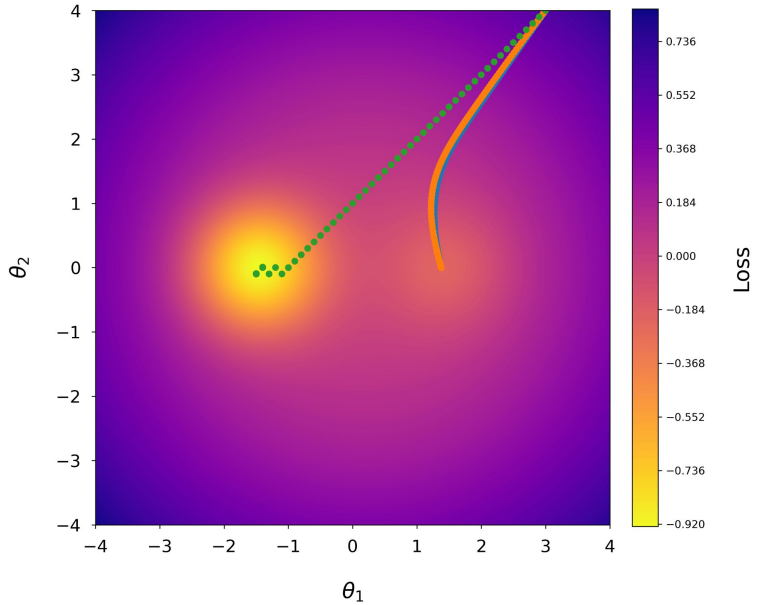
Momentum



8 steps with Momentum vs. 20 steps without Momentum

Momentum


0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Loss
−4 −3 −2 −1 0 1 2 3 4
θ_1
θ_2
SGD
Adagrad
Adam


4 3 2 1 0 −1 −2 −3 −4
θ_2
−4 −3 −2 −1 0 1 2 3 4
θ_1
0.736
0.552
0.368
0.184
0.000
−0.184
−0.368
−0.552
−0.736
−0.920
Loss

Adam: Adaptive Gradient + Momentum

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Kingma et al. (2014) - Adam: A Method for Stochastic Optimization

Iterative Optimization

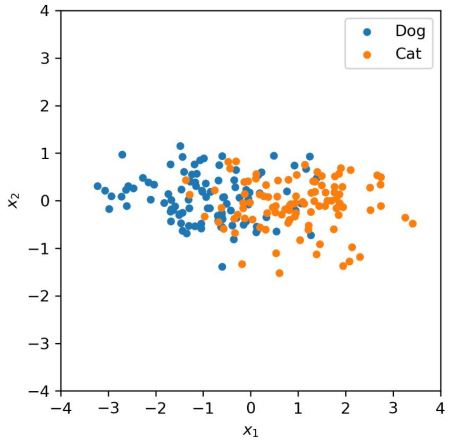
```
optimizer = torch.optim.Adam(network.parameters(), lr=1e-3)

for epoch in range(max_epochs):
    for batch in data:
        optimizer.zero_grad()

        loss = ...

        loss.backward()
        optimizer.step()
```

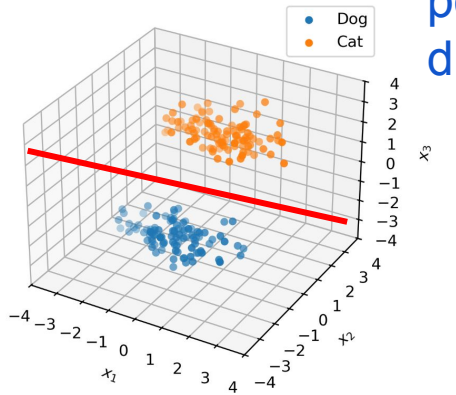
Training Tricks

Kernel Trick

- Kernel method = Linear solver for non-linear problem
- Idea: Project data into easily (i.e., linearly) separable space

Original Data is not linearly separable





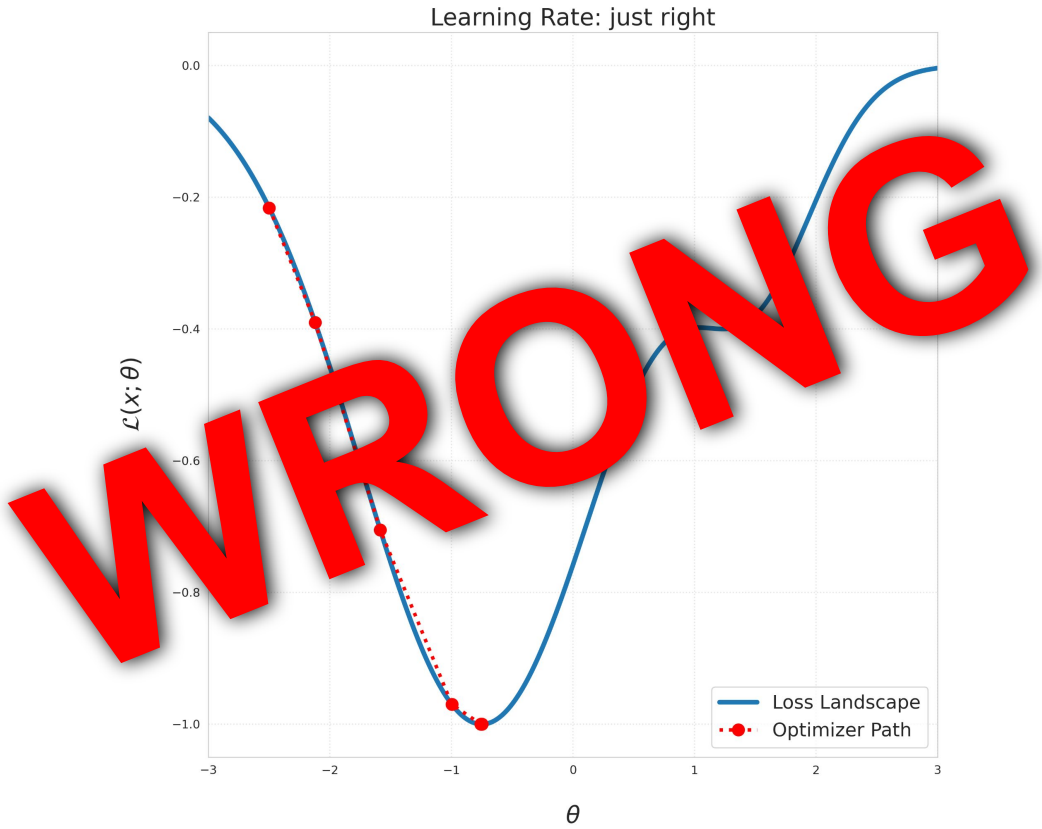
Linear separation is possible in higher dimensional space

➡ Shape networks to **expand** the data first

Kernel Trick

```
import torch.nn as nn

network = nn.Sequential(
    nn.Linear(num_features, 64),
    nn.ReLU(),
    nn.Linear(64, 128),
    nn.ReLU(),
    nn.Linear(128, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 64),
    nn.ReLU(),
    nn.Linear(64, num_classes)
)
```

Gradient Descent



Learning Rate Scheduling

- In practice
  - Change LR throughout training
- Common strategy
  - Warm up LR to target
  - Then anneal to zero
- Effects:
  - Border effects are dampened
  - Convergence is improved
  - Final minimum is deeper

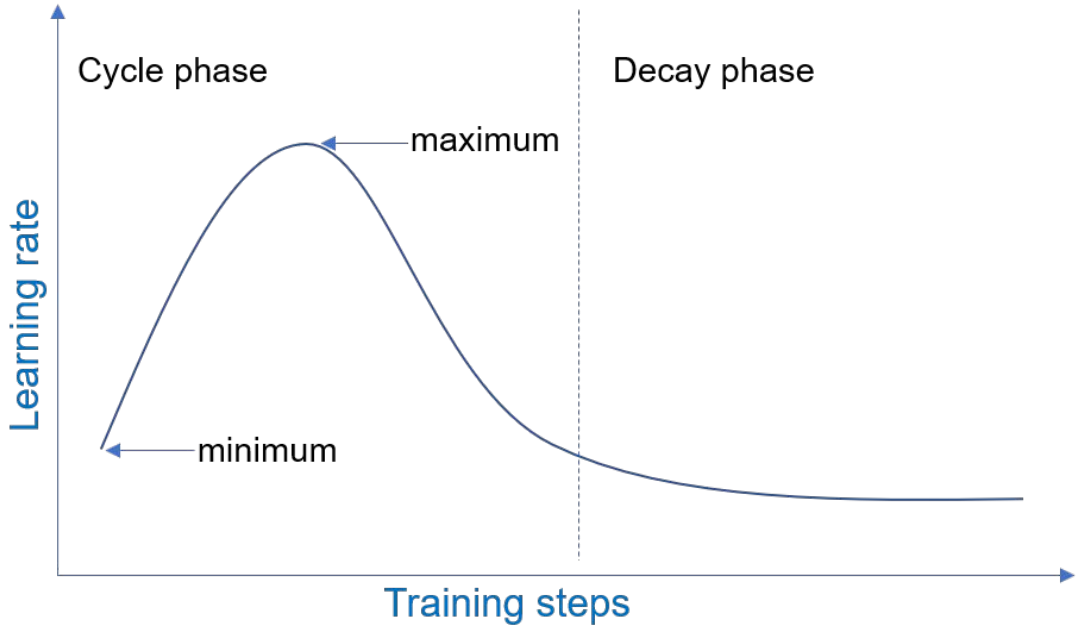


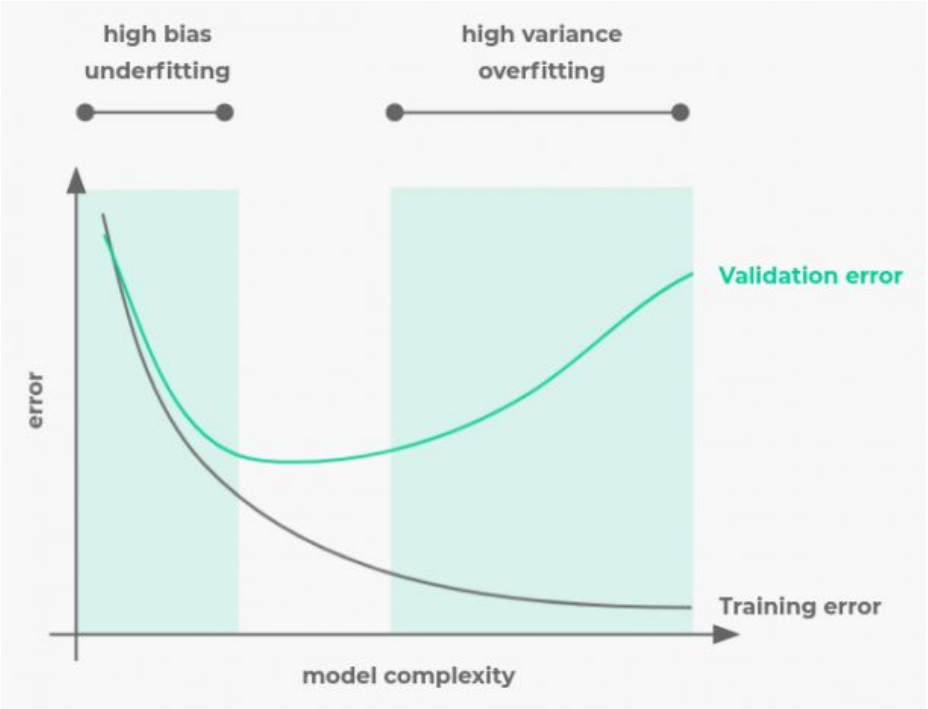
Image Source

Learning Rate Scheduling

```
optimizer = ...
scheduler = torch.optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=1e-3,
    ...
)

...

optimizer.step()
scheduler.step()
```

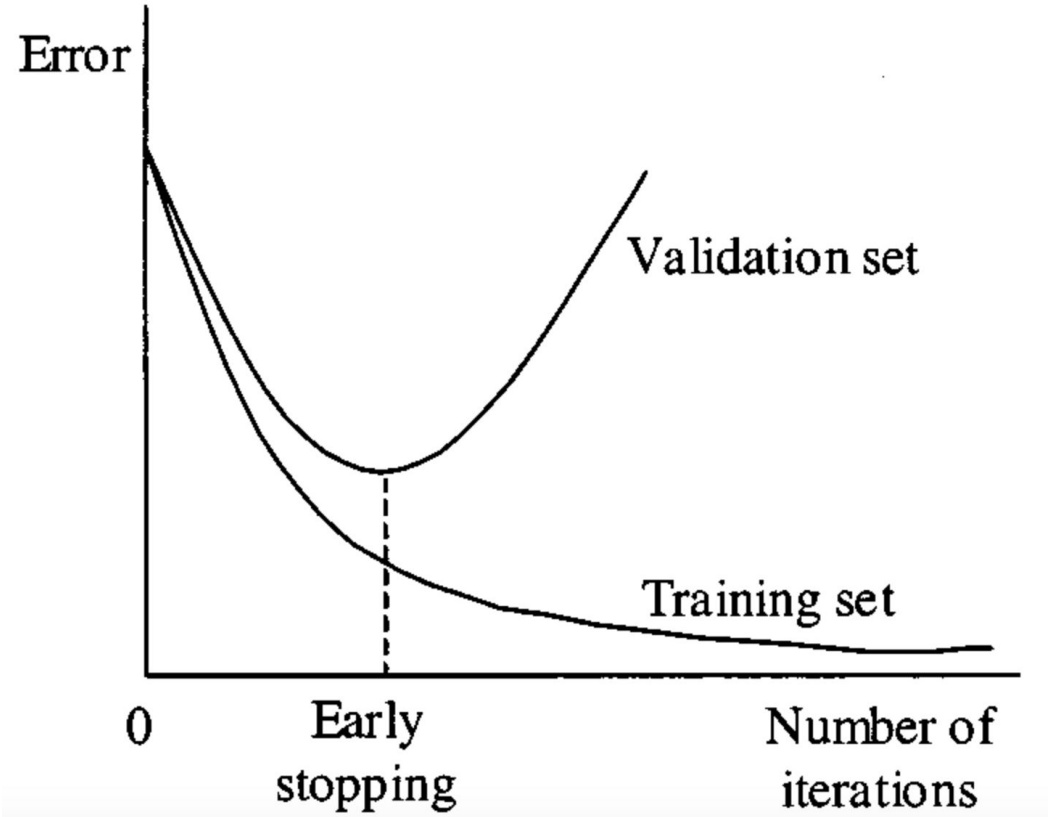
Regularization

- Neural networks overfit when
  - They are too big
  - They are trained for too long
  - They are not regularized



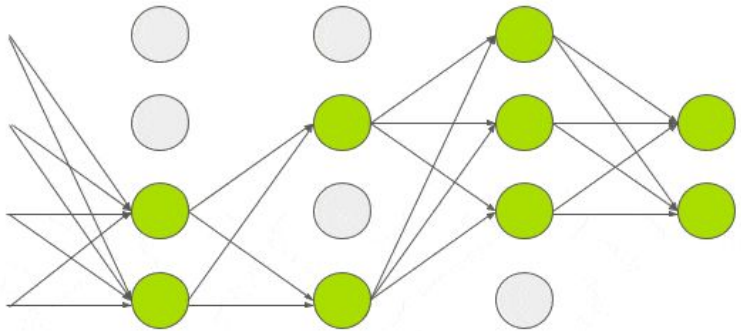
- Regularization:
  - Early Stopping
  - Dropout
  - L1 (LASSO)
  - L2 (ridge regression / weight decay)

Early Stopping



Dropout

- During training
  - Randomly zero activations with probability p
  - Scale up other activations by 1 / (1 - p)
- During inference
  - Do nothing



- Effects:
  - Reduce dependence on single neurons
  - Increases training variance for p in [0, 0.5]
  - Network implicitly becomes an ensemble of smaller networks

Image Source

Dropout

- Use as a layer in network

```
net = nn.Sequential(
    nn.Linear(3, 128),
    nn.ReLU(),

    # use 0 <= p <= 0.5
    nn.Dropout(p=0.5),

    nn.Linear(128, 256),
    ...
)
```

Dropout

- Important: Use eval mode for inference!
  - This turns dropout off

```
net.eval()
```

L1 / L2 Regularization

- Add L1 / L2 norm of parameters to loss

$$\mathcal{L}(x;\theta) = \mathcal{L}_0(x;\theta) + \lambda\|\theta\|$$

- L1:
  - Encourages sparse weights
  - Discourages large weights (overfitting)
  - Helpful for feature selection and model pruning

- L2:
  - Encourages spread out weights
  - Discourages large weights (overfitting)

L2 Regularization

```
optimizer = torch.optim.AdamW(network.parameters(), lr=1e-3, weight_decay=0.01)
```

Batch Normalization

- Problems:
  - Unbounded activations can become large quickly
  - Internal covariate shift slows training
- Solution:
  - Standardize batch of outputs of individual layers
  - Stabilizes training
  - Reduces internal covariate shift
- After Initialization:
  - Keep running estimate of mean and std
  - Freeze at inference time with model.eval()
  - Variants treat mean and std as learnable instead

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

BatchNorm

- Usually apply BatchNorm after Linear Transform
- Torch has 1d, 2d and 3d variants

```
net = nn.Sequential(
    nn.Linear(3, 128),
    nn.BatchNorm1d(128),
    nn.ReLU(),
    ...
)
```

Gradient Clipping

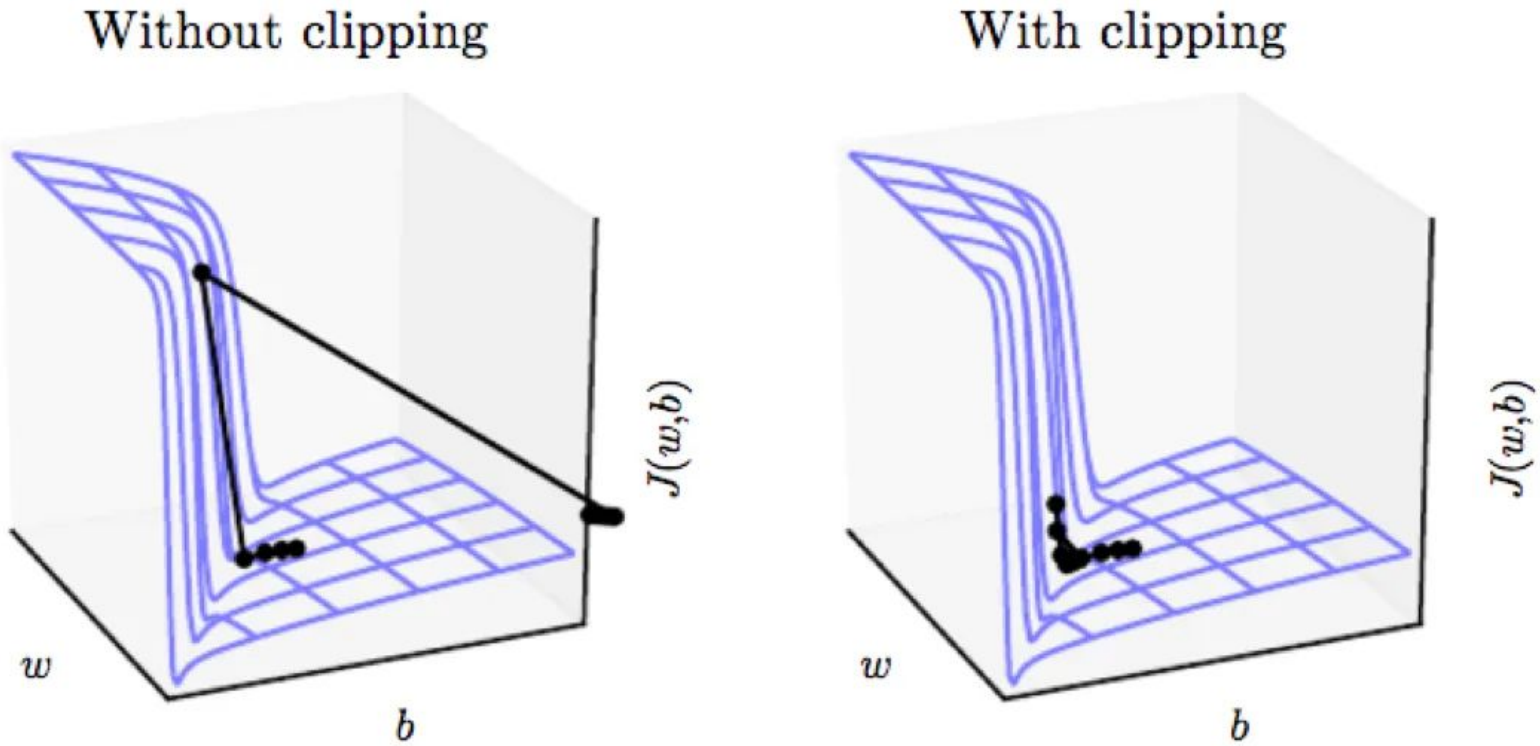


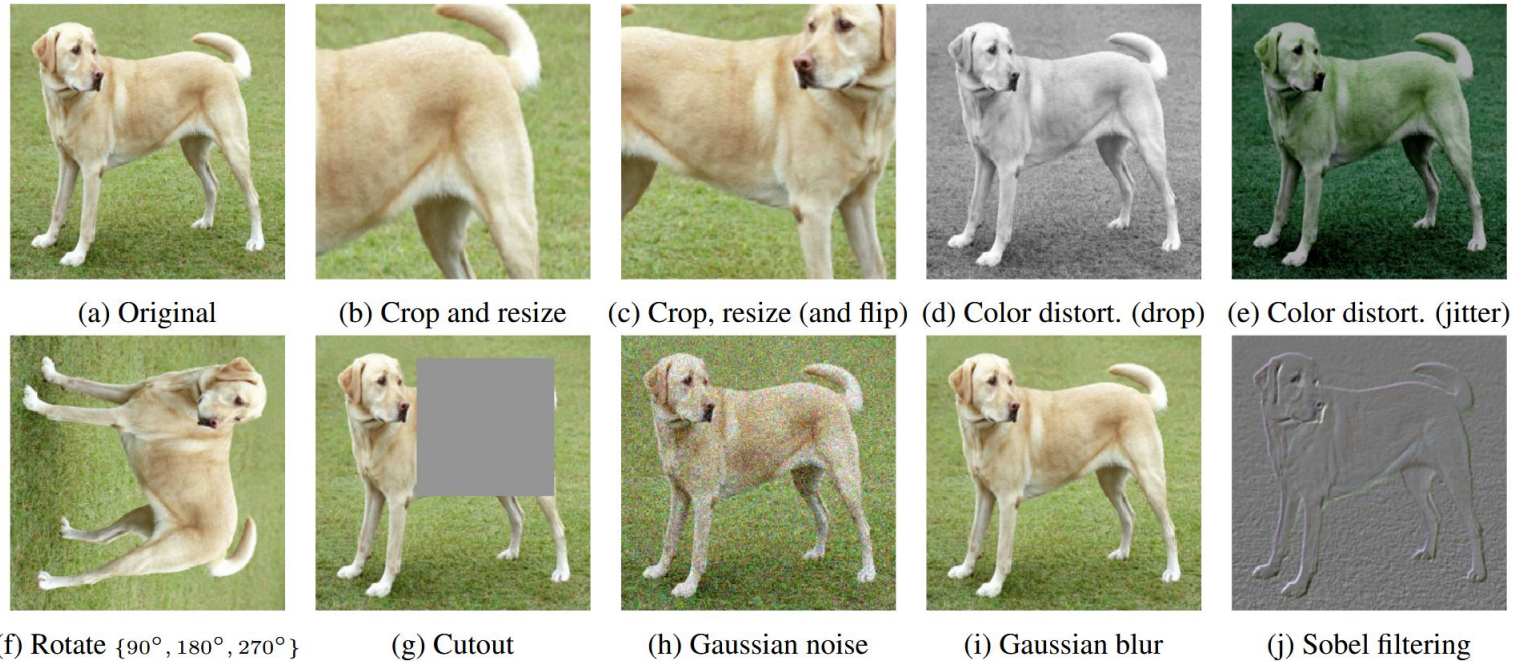
Image Source

Gradient Clipping

```
optimizer.zero_grad()

loss = loss_fn(...)

loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
```

Data Augmentation

- More data improves models, but is very expensive
- Idea: Create more data artificially, without altering labels

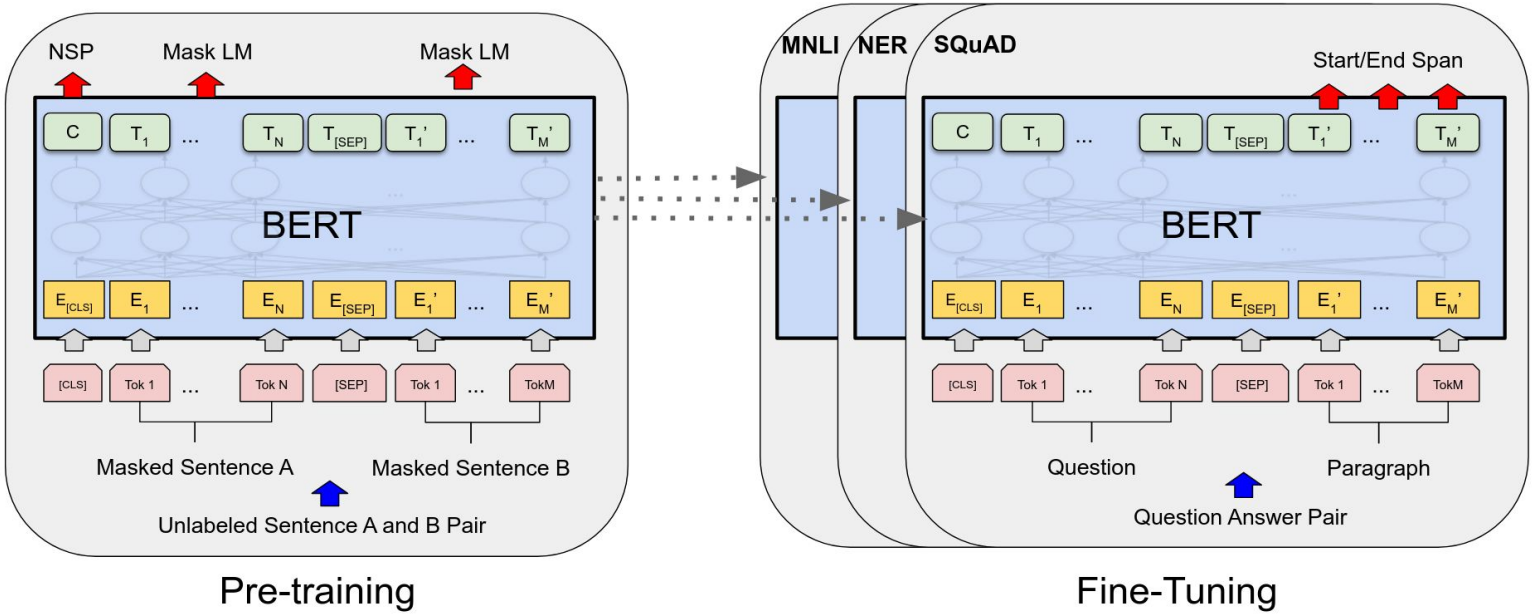


Chen et al. (2020) - A Simple Framework for Contrastive Learning of Visual Representations

Data Augmentation

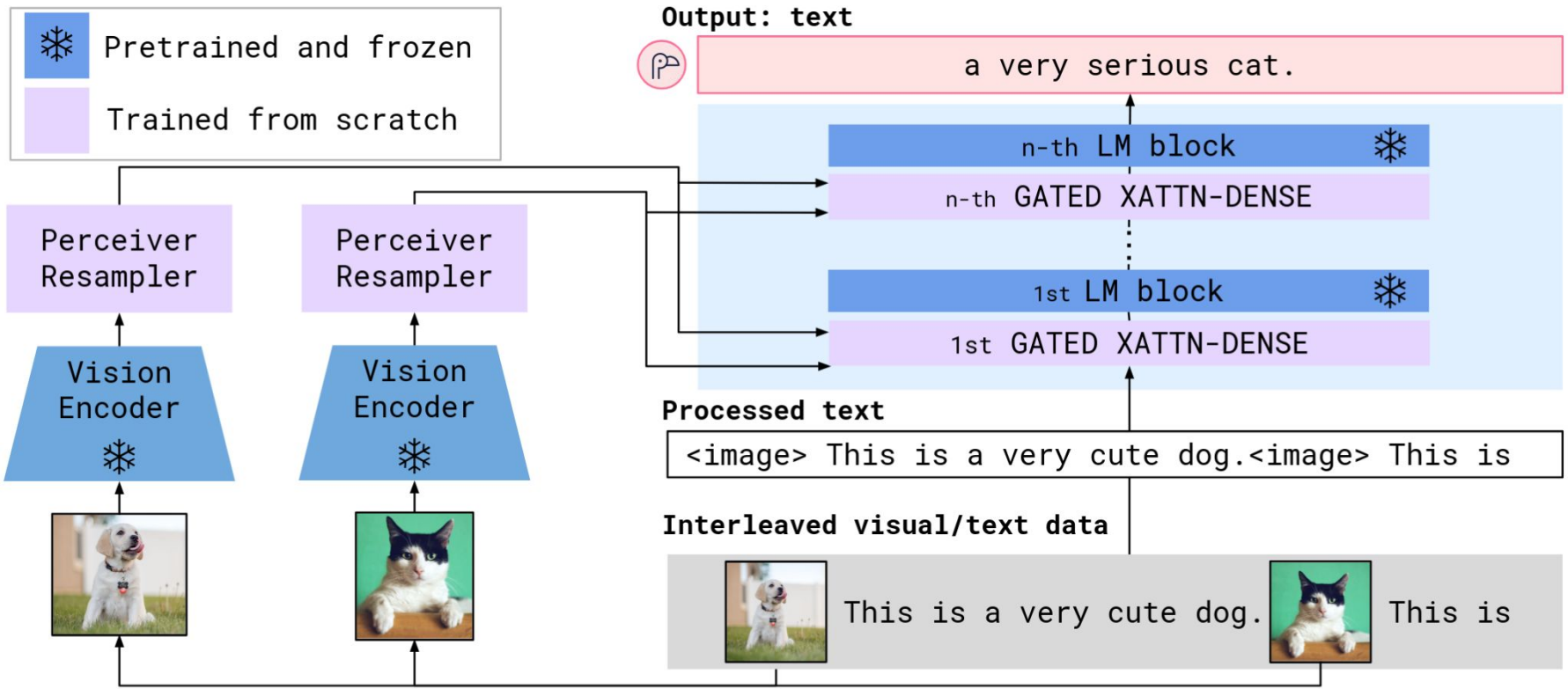
```python
from torchvision.transforms import v2
transforms = v2.Compose([
    v2.ToImage(),  # Convert to tensor, only needed if you had a PIL image
    v2.ToDtype(torch.uint8, scale=True),  # optional, most input are already uint8 at this point
    # ...
    v2.RandomResizedCrop(size=(224, 224), antialias=True),  # Or Resize(antialias=True)
    # ...
    v2.ToDtype(torch.float32, scale=True),  # Normalize expects float input
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Using Pre-Trained Networks



Devlin et al. (2019) - BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Using Pre-Trained Networks



Alayrac et al. (2022) - Flamingo: a Visual Language Model for Few-Shot Learning

Using Pre-Trained Networks

```python
import torch.hub as hub

resnet50 = hub.load("pytorch/vision", "resnet50", pretrained=True)
```

```python
# Use Huggingface's transformers library
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained("Qwen/QwQ-32B-Preview")
model = AutoModelForCausalLM.from_pretrained("Qwen/QwQ-32B-Preview")
```

Qwen2 (2024)

What if my model is too big?

Gradient Checkpointing

- Stored activations cost a lot of memory for deep models
- Idea: Recompute activations instead of storing
  - Reduces memory consumption
  - Costs compute
- Done by wrapping model:

```
torch.utils.checkpoint_sequential(net, chunks=10, input=x)
```

Accumulating Gradient

- Memory consumption grows linearly with batch size
- Instead: accumulate gradient over multiple steps
  - Increases effective batch size
  - Costs compute

```
for epoch in range(max_epochs):
  for i, batch in enumerate(data):
    loss = ...

    # accumulate gradient
    loss.backward()

    if i % n == 0:
      optimizer.step()
      optimizer.zero_grad()
```

Multi-GPU Training

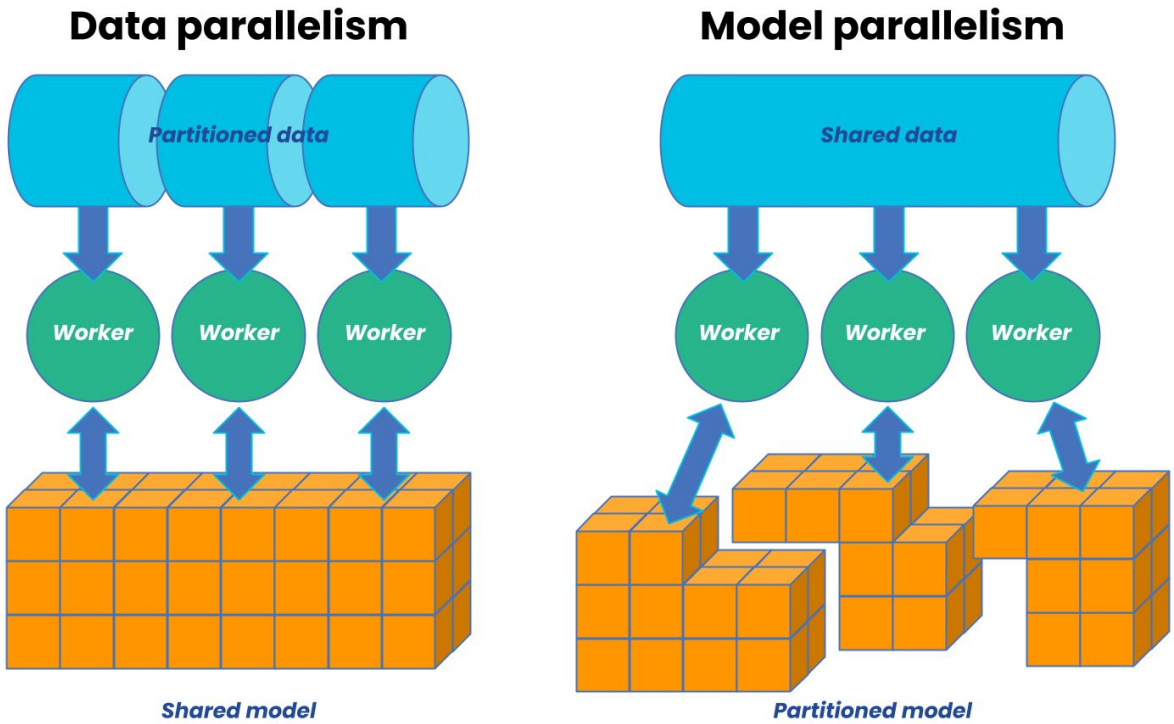


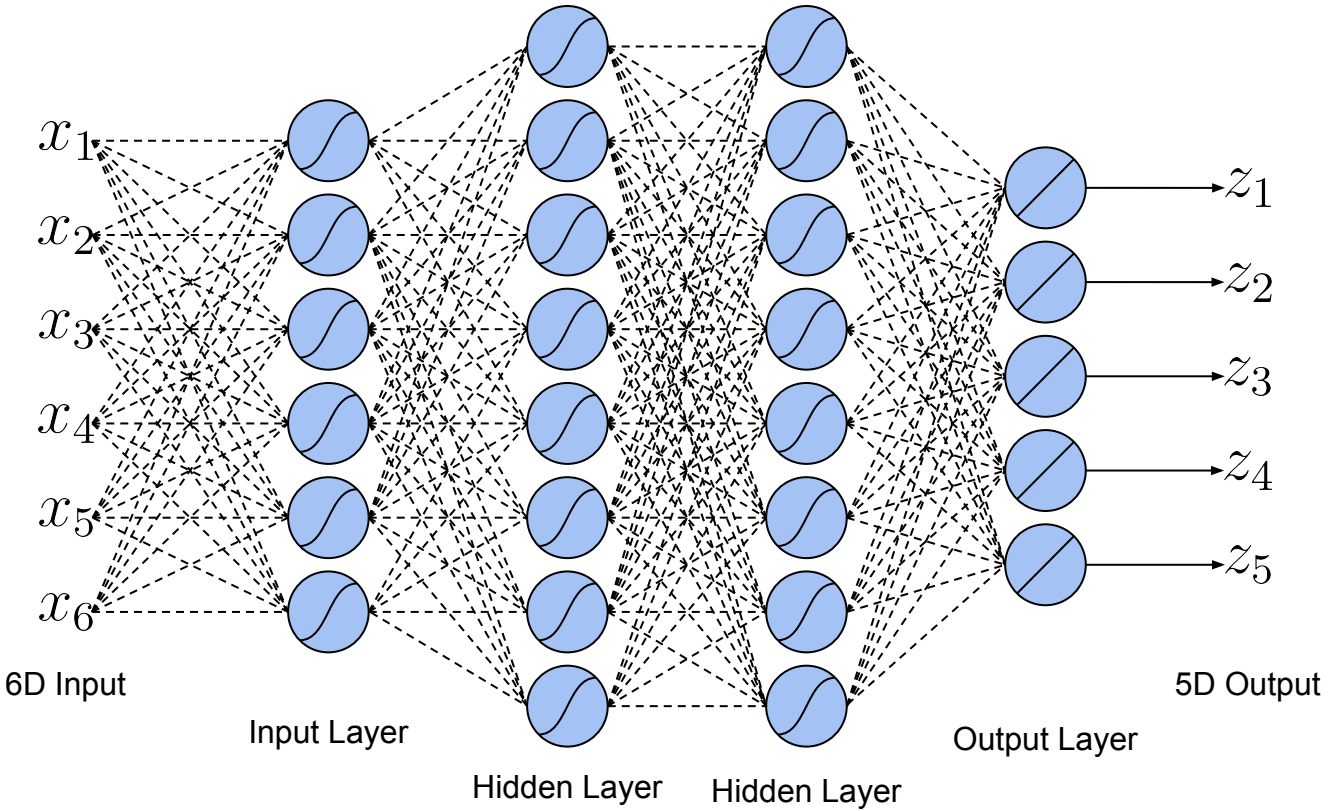
Image Source

Distributed Training in PyTorch

https://pytorch.org/tutorials/beginner/dist_overview.html

Network Architectures

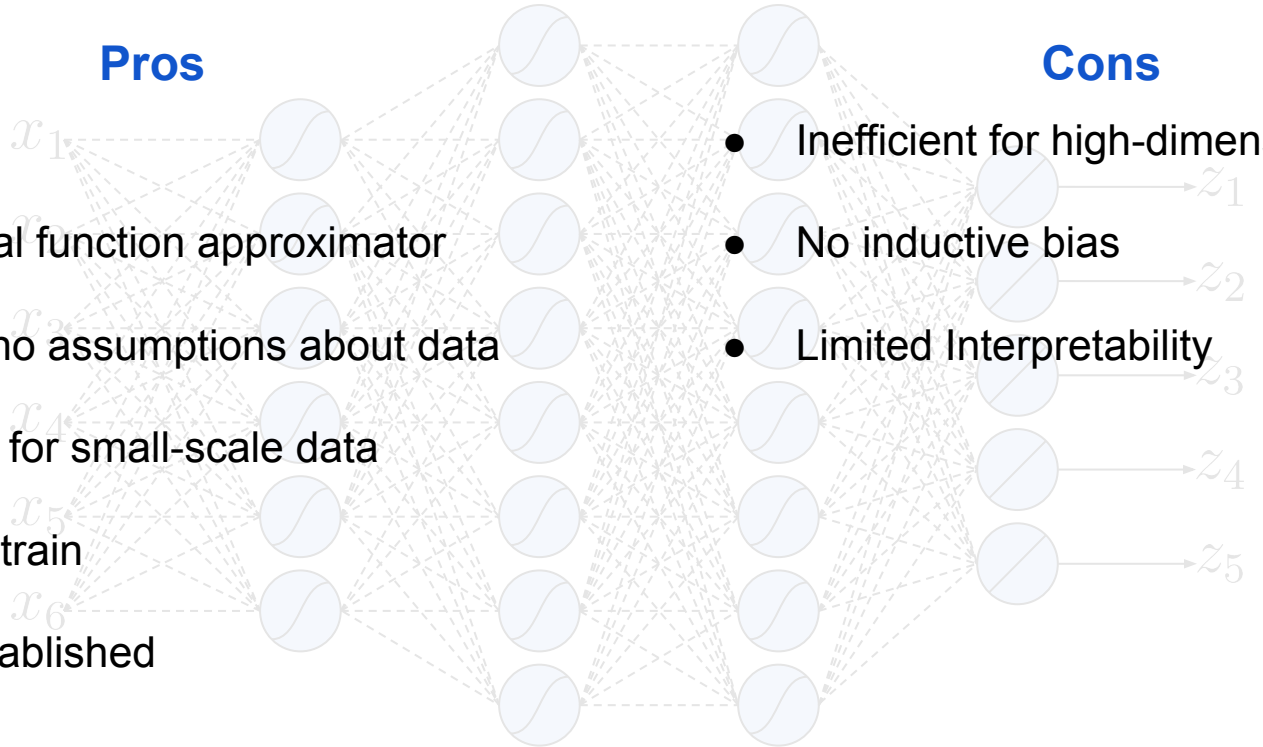
Multi-Layer Perceptron (MLP)



Multi-Layer Perceptron (MLP)

Pros

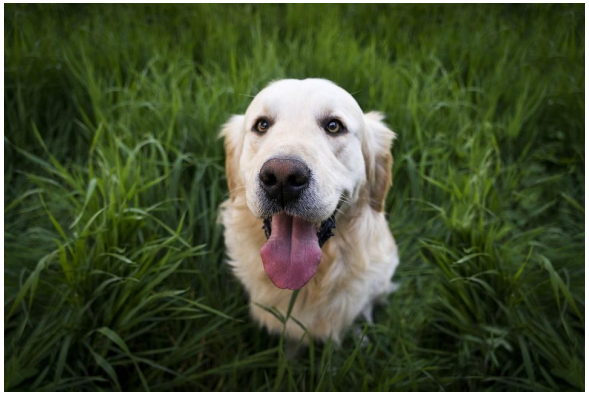
- Simple
- Universal function approximator
- Almost no assumptions about data
- Efficient for small-scale data
- Easy to train
- Well established



Cons

- Inefficient for high-dimensional inputs
- No inductive bias
- Limited Interpretability

Images



Data Types

Sequences

Using Neural Networks on Images

- Very high-dimensional
- 256x256 px RGB Image
  - >196M parameters in the first layer
- Pixels mostly correlate locally
- Patterns are repetitive
  - Edges
  - Textures
- Different input sizes

Convolutional Neural Network (CNN)

- Idea: Use locally fully connected network → Convolution
- Uses fixed number of parameters
  - Independent of input size
- Receptive field is increased by subsequent layers

Convolutional Neural Network (CNN)

Input Image

Kernel

$$(x * k)[n] = \sum_{m=-M}^{M} x[n-m]k[m]$$

Localized Weighted Sum

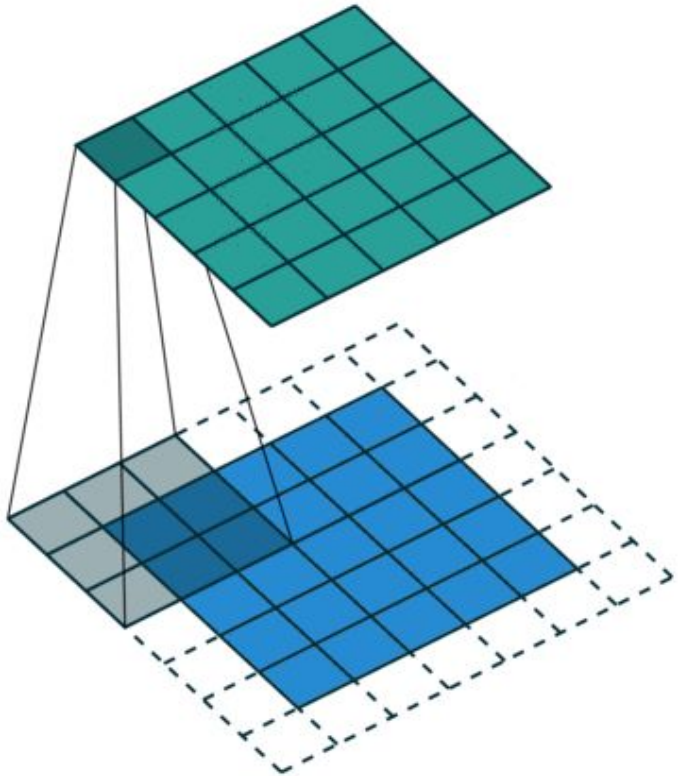
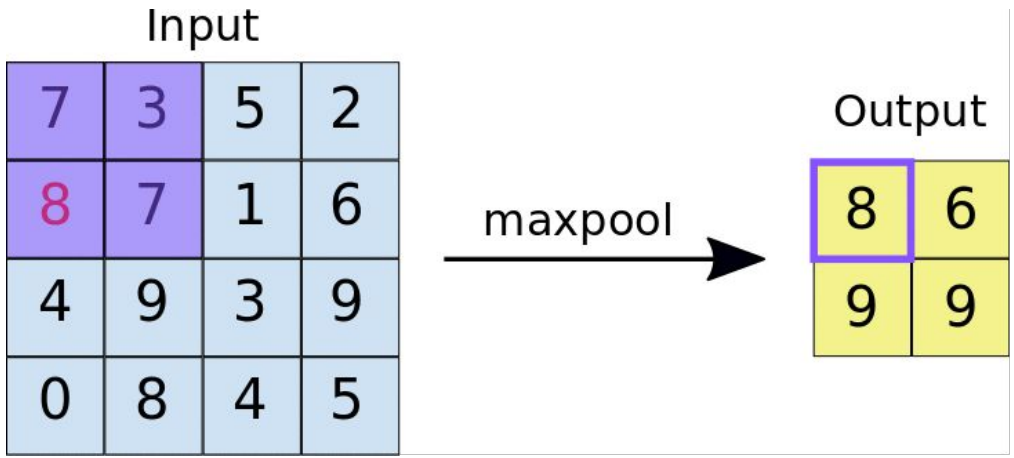


Image Source

Convolutional Neural Network (CNN)

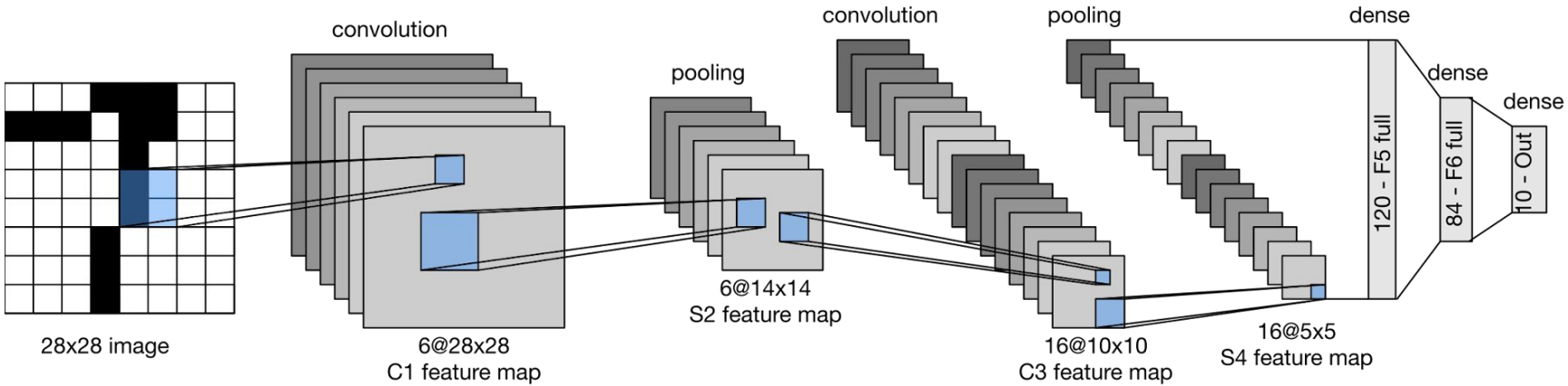
- Data dimensionality is still high
- Idea: Look for patches that match a filter
- Activation measures patch similarity
  - Match → strong activation
  - No match → weak activation
- Keep only matches
  - Discards fewest information



Convolutional Neural Network (CNN)

```python
import torch.nn as nn


cnn = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding="same"),
    nn.ReLU(),
    nn.Conv2d(16, 32, 3, padding="same"),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(...),
    ...,
    nn.Flatten(),
    nn.Linear(...),
    ...,
    nn.Linear(...),
)
```

LeNet-5



LeCun et al. (1998) - Gradient-based learning applied to document recognition

MNIST

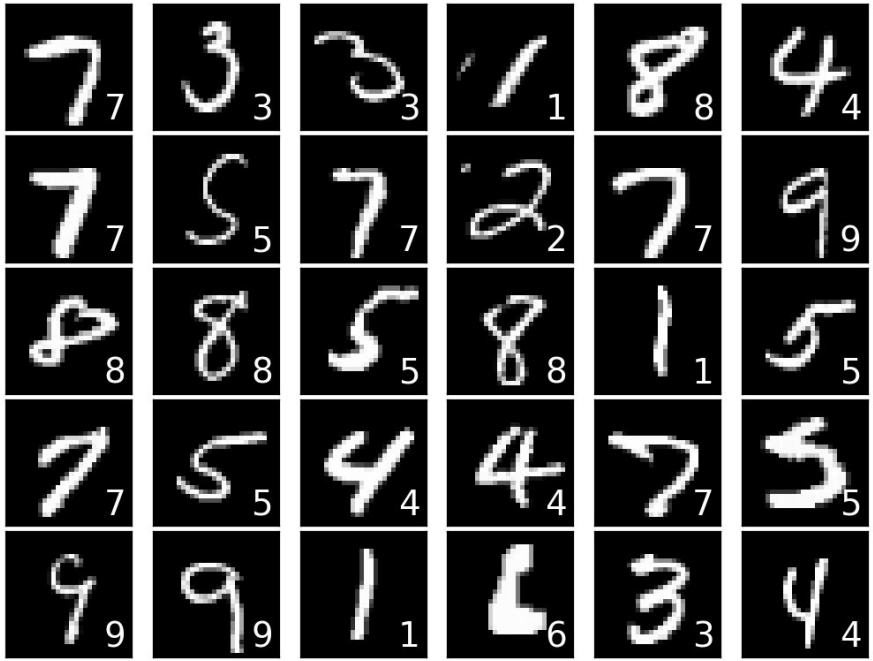
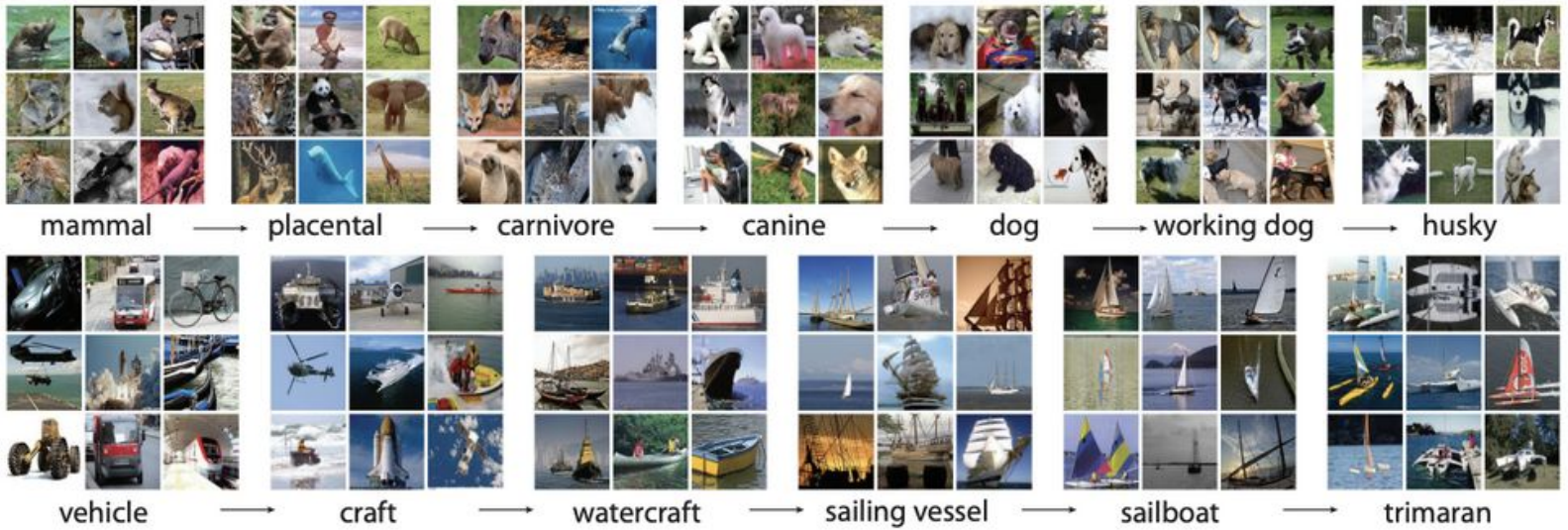


Image Source

ImageNet Competition

- Started in 2010
- 14 million images, 1000 labels
- Very hard for classical models
  - >25% top-5 error



Deng et al. (2009) - ImageNet: A Large-Scale Hierarchical Image Database

ImageNet Competition

- AlexNet (2012)
  - Uses ReLU & GPU to speed up training
  - Uses dropout and augmentation for regularization
- VGGNet (2014)
  - Uses very deep architecture (16 layers)

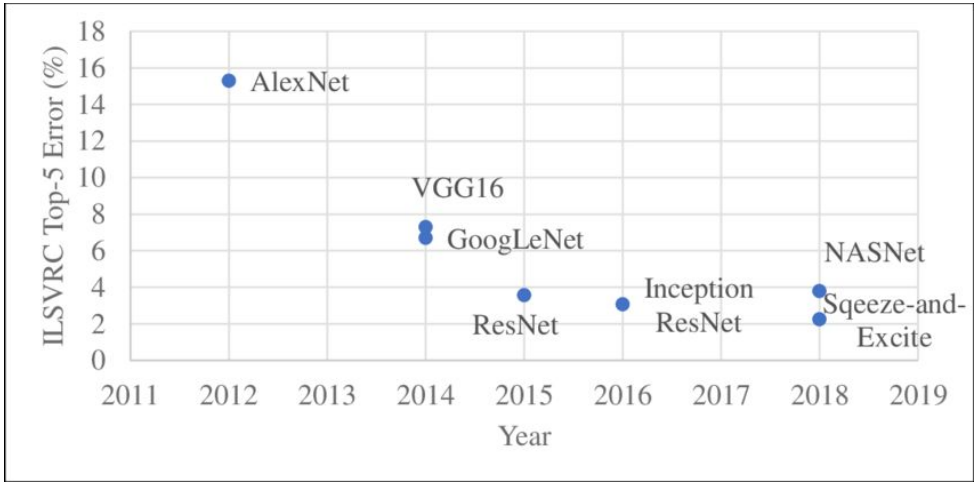


Image Source

Simonyan et al. (2014) - Very Deep Convolutional Networks for Large-Scale Image Recognition

Krizhevsky et al. (2012) - ImageNet Classification with Deep Convolutional Neural Networks

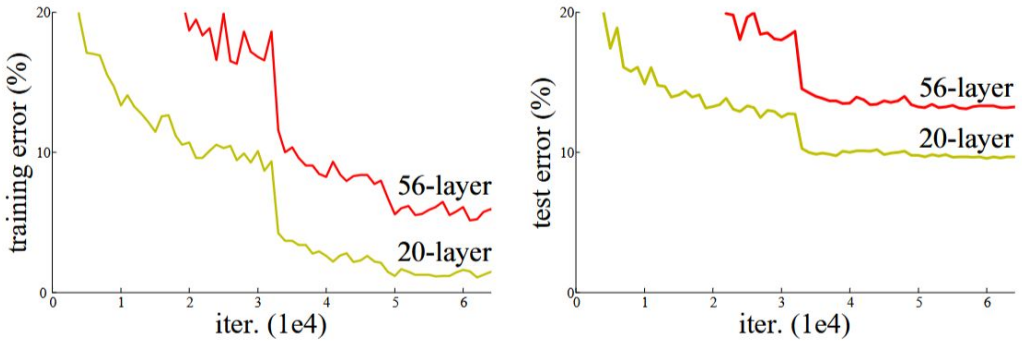
Feature Visualization

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

The problem with deep networks

- Deep networks are the most expressive, but
- Gradient magnitude is exponential in depth
- Very deep networks → Very large or small gradient

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial z_L} \frac{\partial z_L}{\partial z_{L-1}} \cdots \frac{\partial z_l}{\partial \theta_l}$$



Kaiming et al. (2015) - Deep Residual Learning for Image Recognition

Residual Neural Network (ResNet)

- Idea: Introduce Skip Connections

$$f(x) = x + g(x) \qquad \frac{\partial f}{\partial x} = 1 + g'(x)$$

  - Fix Vanishing / Exploding Gradient
  - Allow training of arbitrarily deep networks
  - Works for dimension-preserving g
    - Use padding for convolution
    - Variant: Concatenate instead of add

Kaiming et al. (2015) - Deep Residual Learning for Image Recognition

Residual Neural Network (ResNet)

```python
import torch.nn as nn


class Skip(nn.Module):
    def __init__(self, inner: nn.Module):
        super().__init__()
        self.inner = inner

    def forward(self, x):
        return x + self.inner(x)
```

Residual Neural Network (ResNet)

```python
import torch.hub as hub

resnet50 = hub.load("pytorch/vision", "resnet50", pretrained=True)
```

Kaiming et al. (2015) - Deep Residual Learning for Image Recognition

How can we output images?

Semantic Segmentation

- Assign each pixel a label

Input

segmented

1: Person
2: Purse
3: Plants/Grass
4: Sidewalk
5: Building/Structures

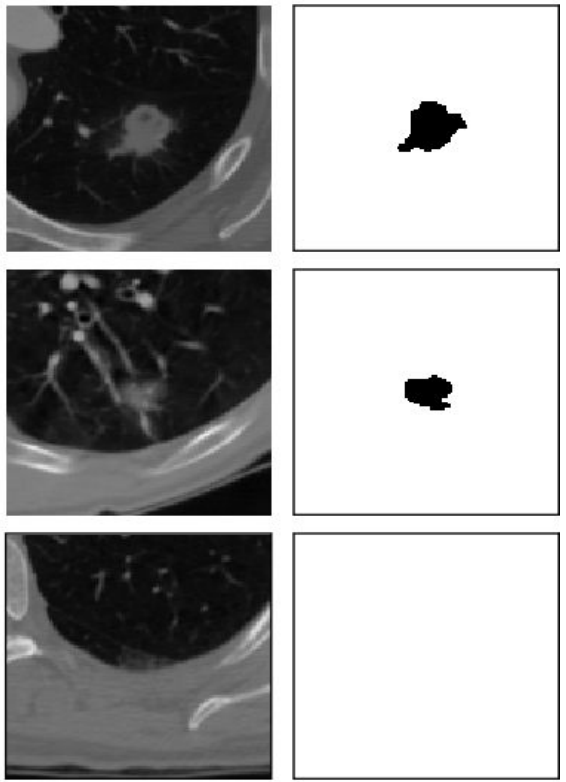
3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	5
3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	5
3	3	3	3	3	3	1	1	3	3	3	3	5	5	5	5	5	5
3	3	3	3	3	1	1	1	1	3	3	3	5	5	5	5	5	5
3	3	3	3	3	3	1	1	3	3	3	5	5	5	5	5	5	5
5	5	3	3	3	3	1	1	3	3	5	5	5	5	5	5	5	5
4	4	3	4	1	1	1	1	1	1	4	4	4	5	5	5	5	5
4	4	3	4	1	1	1	1	1	1	4	4	4	4	4	5	5	5
4	4	4	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4
3	3	3	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4
3	3	3	1	2	2	1	1	1	1	1	4	4	4	4	4	4	4
3	3	3	1	2	2	1	1	1	1	1	4	4	4	4	4	4	4

Semantic Labels

Image Source

Semantic Segmentation

Given Image

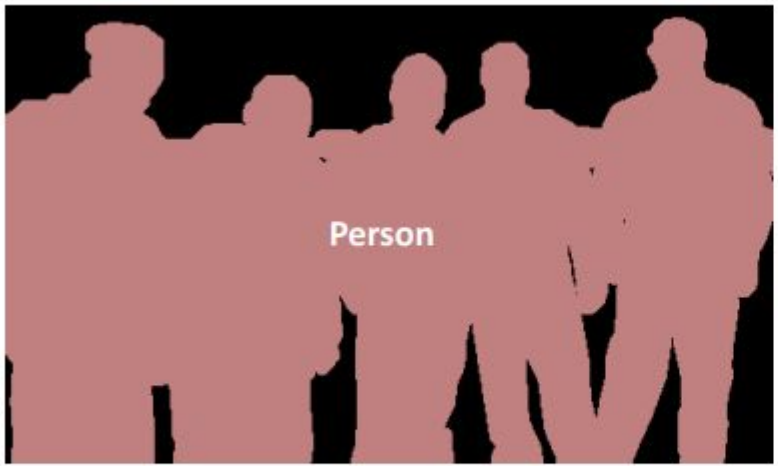


Find Cancer

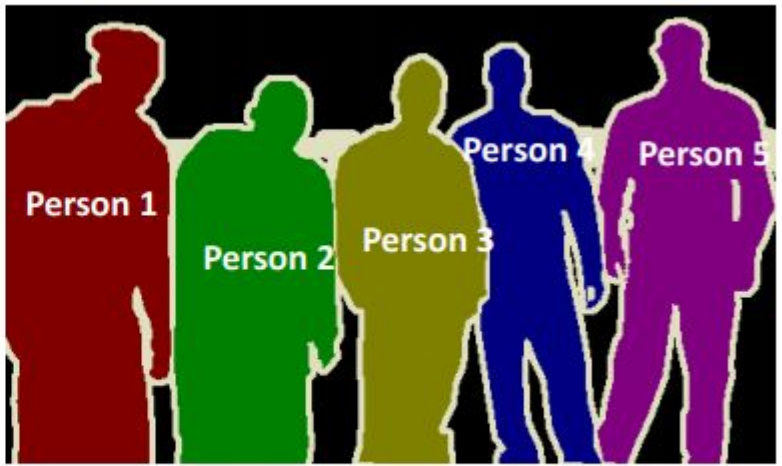
= Segmentation Map

Zbinden et al. (2023) - Stochastic Segmentation with Conditional Categorical Diffusion Models

Instance Segmentation



Semantic Segmentation

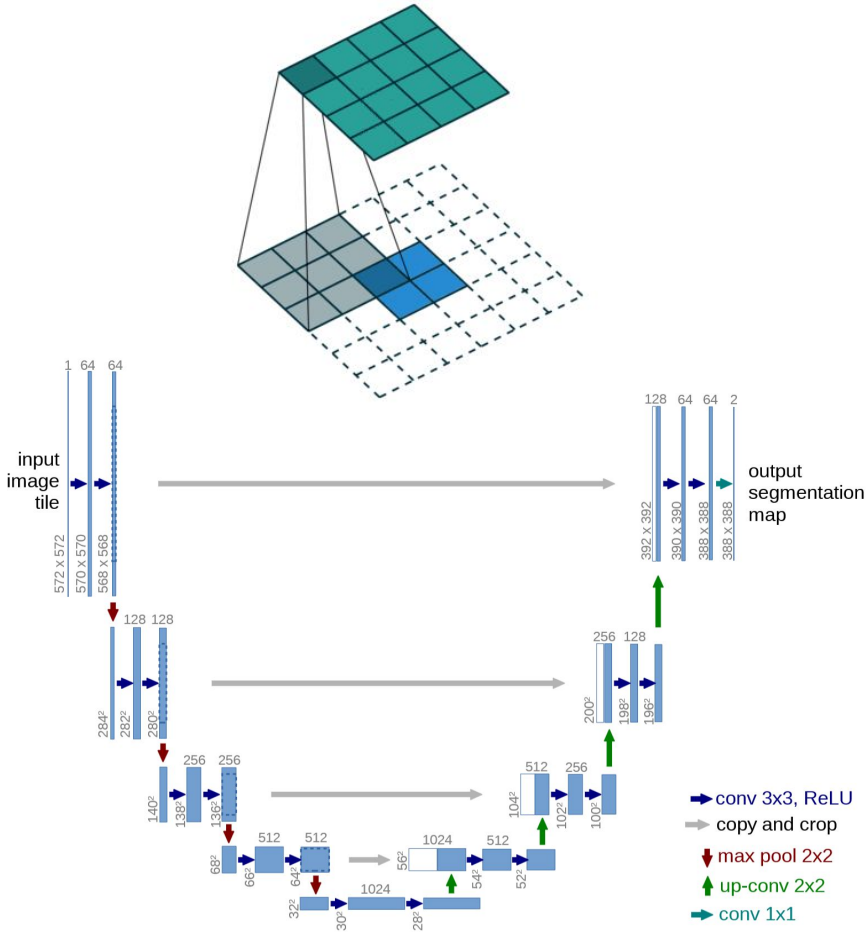


Instance Segmentation

Image Source

U-Net

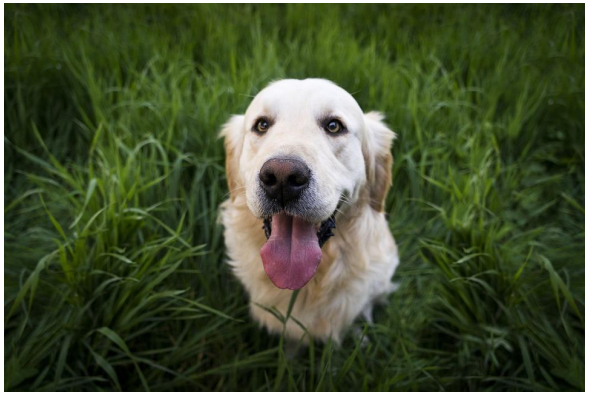
- Plain CNNs only reduce the dimension
- Idea: Use pseudo-inverse of CNN
  - Maxpool for downscaling
  - Up-convolution for upscaling
- Skip-connection for each stage
- Flatten and use MLP in bottom stage
  - Expressive power is concentrated here
- U-shape → U-Net



Ronneberger et al. (2015) - U-Net: Convolutional Networks for Biomedical Image Segmentation

UNet in Exercise

Images



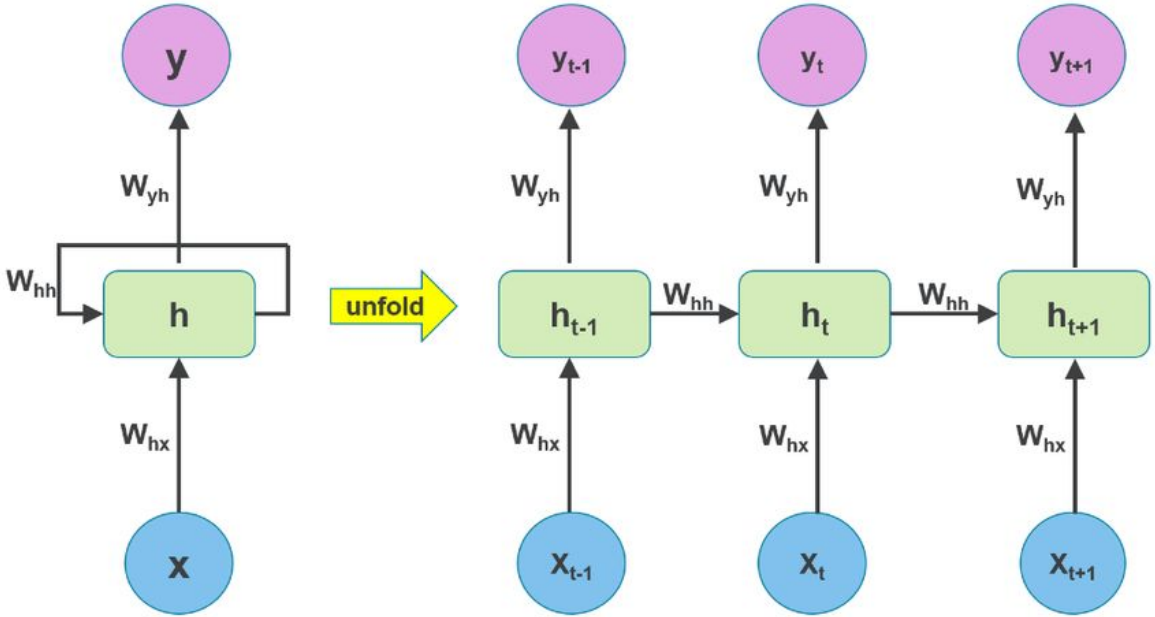
Data Types

Sequences

Recurrent Neural Network (RNN)

- Maintain hidden state h
- Output is computed from hidden state
- Input only updates hidden state

$$h_t = f(W_h h_{t-1} + W_x x_t + b_h)$$
$$y_t = g(W_y h_t + b_y)$$



- Problems:
  - Hidden state initialization
  - Sequential iteration is slow
  - Limited long-term memory
  - Vanishing Gradient ⇒ LSTM

$$\frac{\partial \mathcal{L}}{\partial \theta} \propto \prod_{t=1}^{T} f'(x_t)$$

Image Source

Recurrent Neural Network (RNN)

```python
import torch
from torch.nn import RNN

batch_size = 32
sequence_length = 128

input_features = 2
hidden_features = 128

rnn = RNN(input_features, hidden_features, batch_first=True)

# random input data
x = torch.randn((batch_size, sequence_length, input_features))

# initialize hidden state
hidden_state = torch.zeros((batch_size, hidden_features))

# update hidden state from full sequence
z, hidden_state = lstm(x, hidden_state)
```

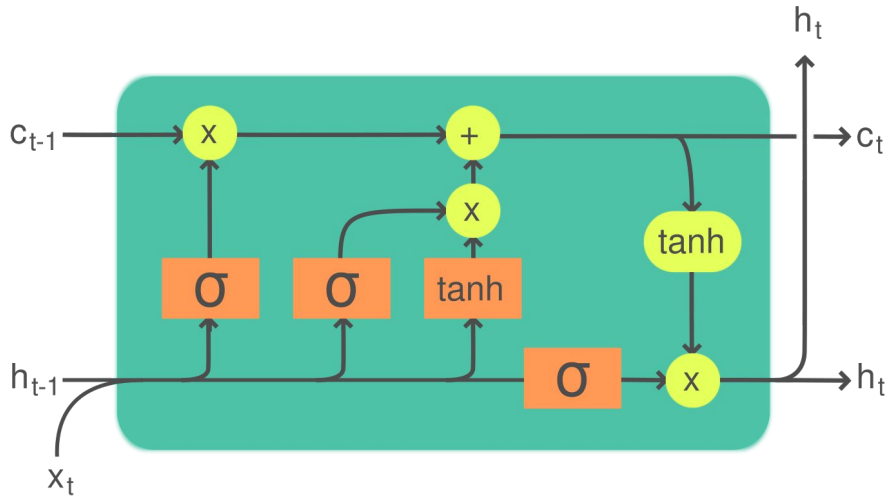
Long Short-Term Memory (LSTM)

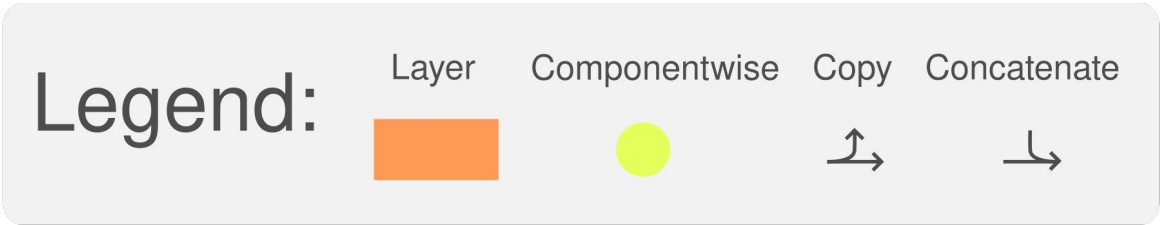
- Core Ideas:
  - Use gates to control the memory
  - Use additive updates like a skip-connection
- Mitigates Vanishing Gradient
- Allows for longer memory retention
- Introduces separate cell state c
  - Serves as the memory
  - Updated additively by input / forget gates

Long Short-Term Memory (LSTM)

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \sigma_c(W_c x_t + b_c) \\
h_t &= o_t \odot \sigma_h(c_t)
\end{aligned}
$$

Long Short-Term Memory (LSTM)





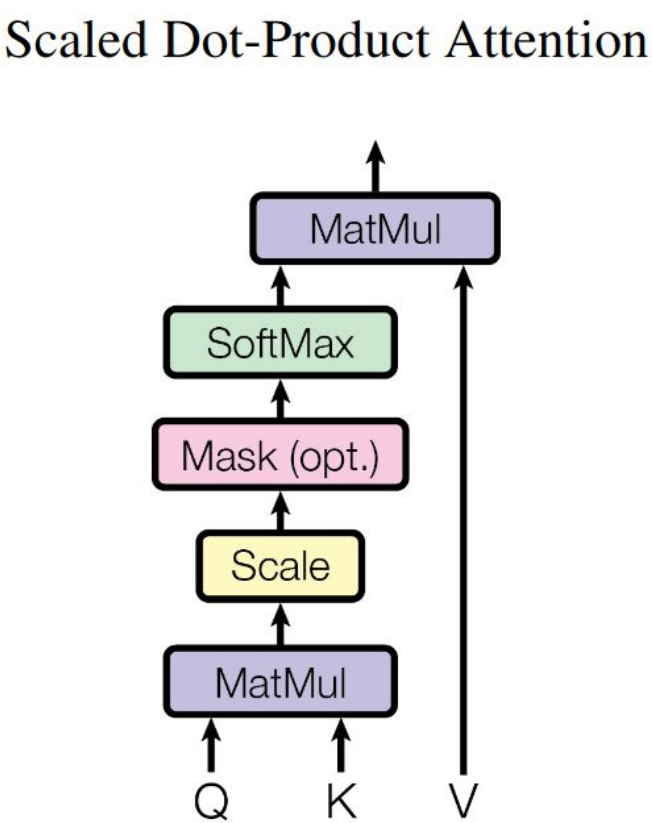
Long Short-Term Memory (LSTM)

```python
import torch
from torch.nn import LSTM

batch_size = 32
sequence_length = 128

input_features = 2
hidden_features = 128

lstm = LSTM(input_features, hidden_features, batch_first=True)

# random input data
x = torch.randn((batch_size, sequence_length, input_features))

# initialize hidden state
hidden_state = torch.zeros((batch_size, hidden_features))

# initialize LSTM cell state
cell_state = torch.zeros((batch_size, hidden_features))

# update hidden and cell state from full sequence
z, (hidden_state, cell_sate) = lstm(x, (hidden_state, cell_state))
```

Attention

- Sequential processing is limiting
  - Slow
  - Limited long-term memory
- Core ideas:
  - Use positional encoding
  - Process the whole sequence in parallel
  - Focus on relevant parts of the sequence

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Vaswani et al. (2017) - Attention Is All You Need

Attention

The quick brown **fox** jumps over the lazy dog

The quick brown fox jumps **over** the lazy dog

The quick brown fox jumps over the lazy **dog**

Attention

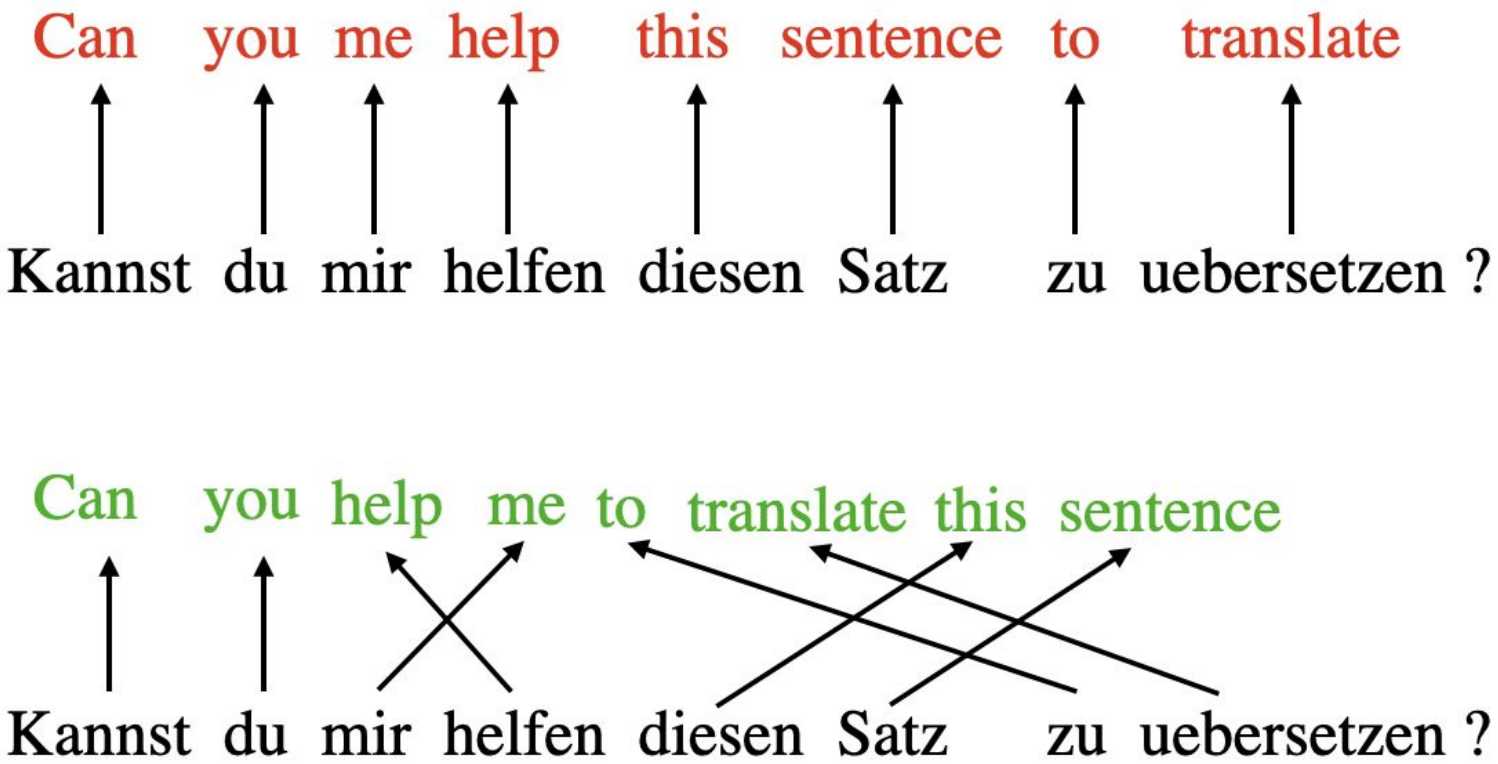


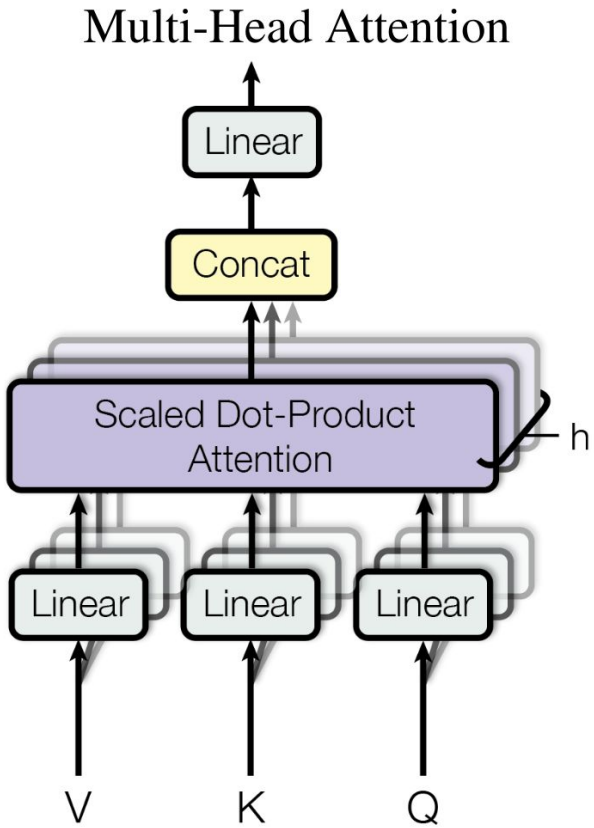
Image Source

Visualizing Attention Maps

with BertViz

Multi-Head Attention

- Attention uses a single representation
  - Limiting, especially since the projection is linear
- Use multiple projections instead
  - Multiple “heads”



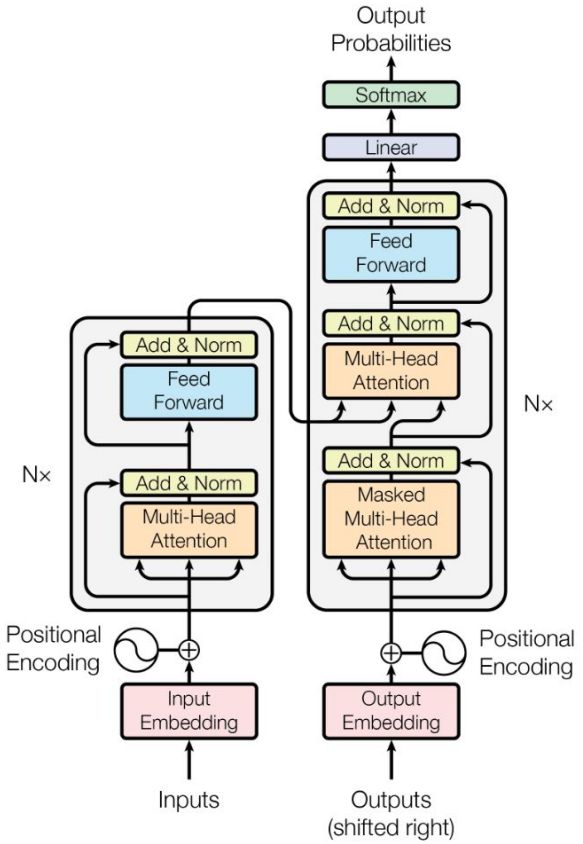
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Vaswani et al. (2017) - Attention Is All You Need

Transformer

- Stacks multiple attention blocks
- LayerNorm between blocks
  - Ensures attention-map stability for deep transformers
- **Encoder**
  - “Reads and understands” the input sequence
  - Gives **one** informative embedding of the **whole** sequence
- **Decoder**
  - Uses **one** embedding and a **sequence** of inputs
  - Allows autoregressive generation of a **new** sequence
- Encoder and Decoder are **both optional**
  - Different combinations have different roles



Vaswani et al. (2017) - Attention Is All You Need

Transformers

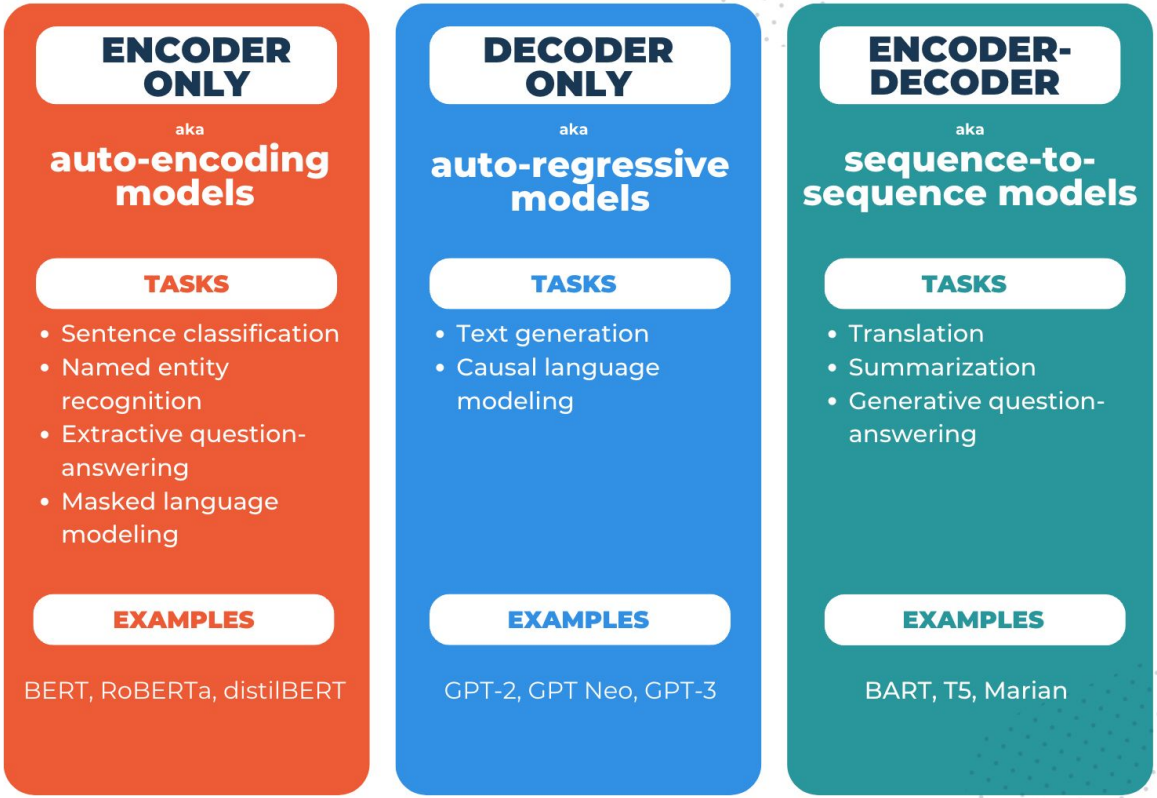
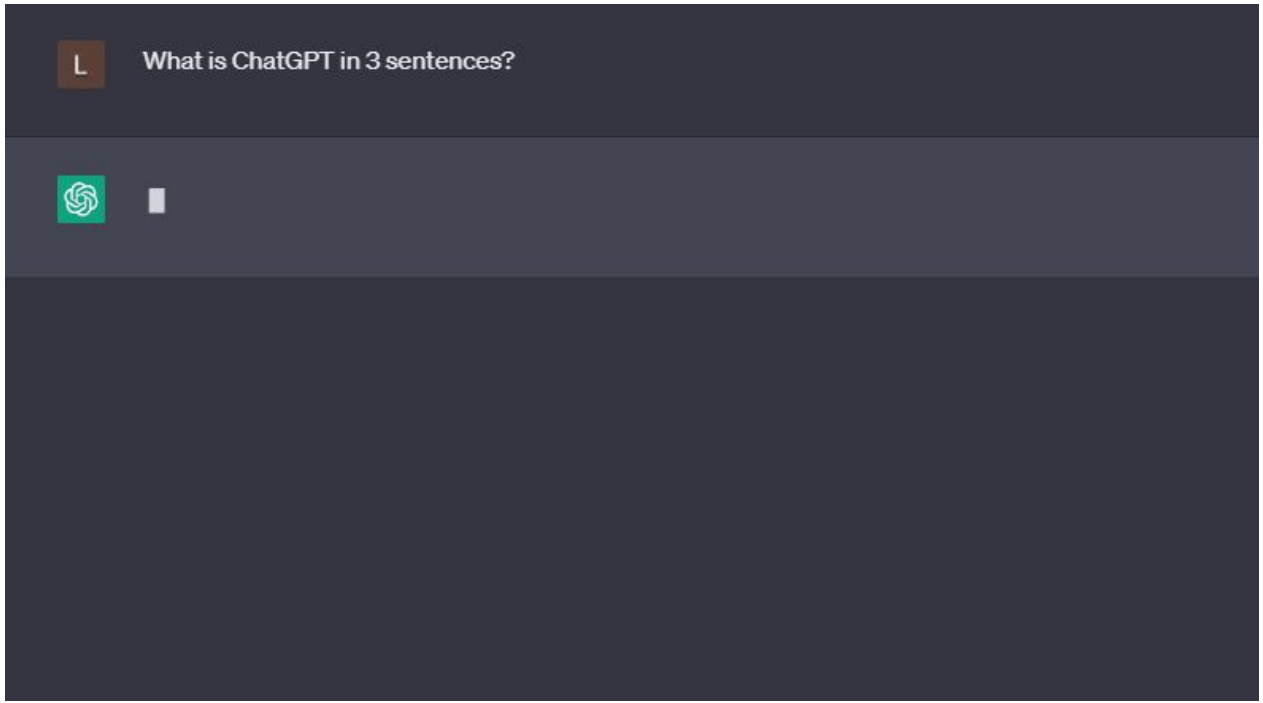
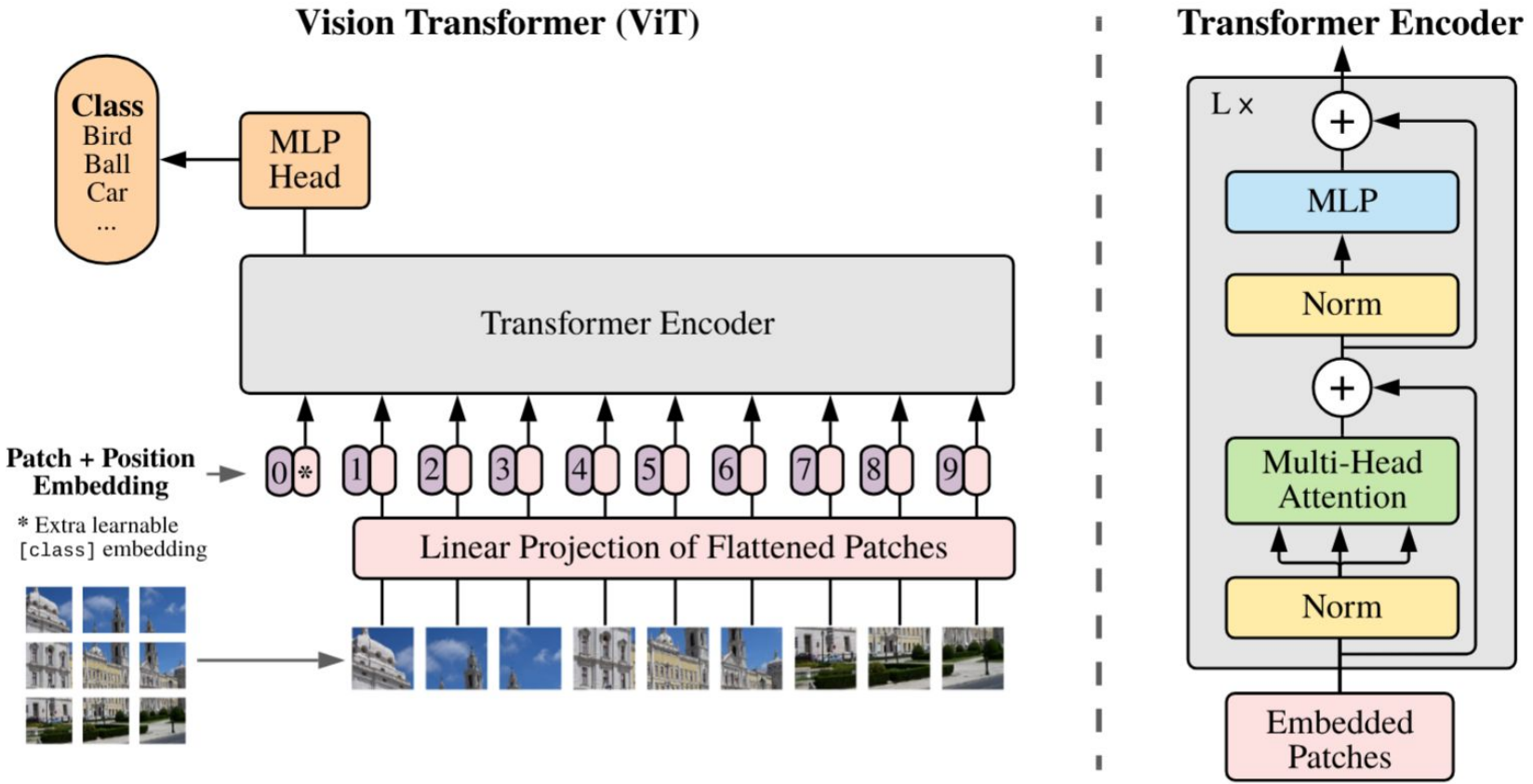


Image Source

GPT - Generative Pre-Trained Transformer



Vision Transformer



An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Generative Vision Transformers

https://openai.com/index/sora/

Transformers for Unordered Sets

Set Transformer

Transformers in PyTorch

```python
import torch
import torch.nn as nn

transformer = nn.Transformer(nhead=16, num_encoder_layers=12)

# careful: sequence dimension comes first!
input_sequence = torch.rand((10, 32, 512))
target_sequence = torch.rand((20, 32, 512))
out = transformer(input_sequence, target_sequence)

print(out.shape)  # torch.Size([20, 32, 512])
```

Transformer Building Blocks

PyTorch Tutorial

Pre-Trained Transformers

From PyTorch Hub

Generative Models

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

Generative Model Recipe

```
class GenerativeModel:
    def compute_loss(self, x):
        ...

    def sample(self, n):
        ...

    @optional
    def log_prob(self, x):
        ...
```

Neural Networks make great Generative Models

https://www.whichfaceisreal.com/index.php

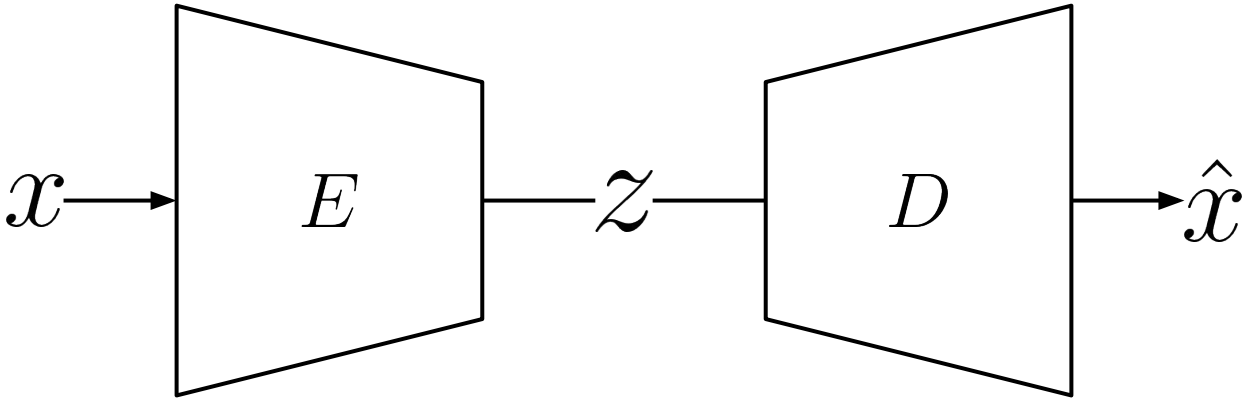
Autoencoder

Manifold Hypothesis:

Many high-dimensional datasets lie on a low-dimensional manifold.

- High redundancy
  - Computationally expensive
  - Can compress data without (significant) loss
  - Yield low-dimensional latent space
- Idea: learn optimal compression algorithm
- Compression should retain information
  - Train with reconstruction loss
  - Can also apply cycle consistency

Autoencoder



$$\mathcal{L}(x, \hat{x}) = \mathrm{MSE}(x, \hat{x})$$

Autoencoder

```python
import torch.nn as nn

encoder = nn.Sequential(
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 32)
)

decoder = nn.Sequential(
    nn.Linear(32, 64),
    nn.ReLU(),
    nn.Linear(64, 128),
    nn.ReLU(),
    nn.Linear(128, 256)
)
```

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

...

loss = F.mse_loss(x, decoder(encoder(x)))
```

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

alpha = 0.5

...

z = encoder(x)
xhat = decoder(z)
zhat = encoder(xhat)

reconstruction_loss = F.mse_loss(x, xhat)
cycle_consistency_loss = F.mse_loss(z, zhat)

loss = alpha * reconstruction_loss + (1 - alpha) * cycle_consistency_loss
```

Are Autoencoders Generative Models?

Generative Model Recipe

```
class GenerativeModel:
    def compute_loss(self, x):
        ...

    def sample(self, n):
        ...

    @optional
    def log_prob(self, x):
        ...
```

Variational Autoencoder (VAE)

- Variational Assumption:
  - Probabilistic latent space
  - Each input sample maps to a latent normal distribution $N(\mu(x), \Sigma(x))$
  - Latent marginal is the standard normal $N(0, 1)$
  - Train with KL-Divergence
- We can now sample:
  - Draw $z \sim N(0, 1)$
  - Pass through the decoder: $x = D(z)$

Variational Autoencoder (VAE)

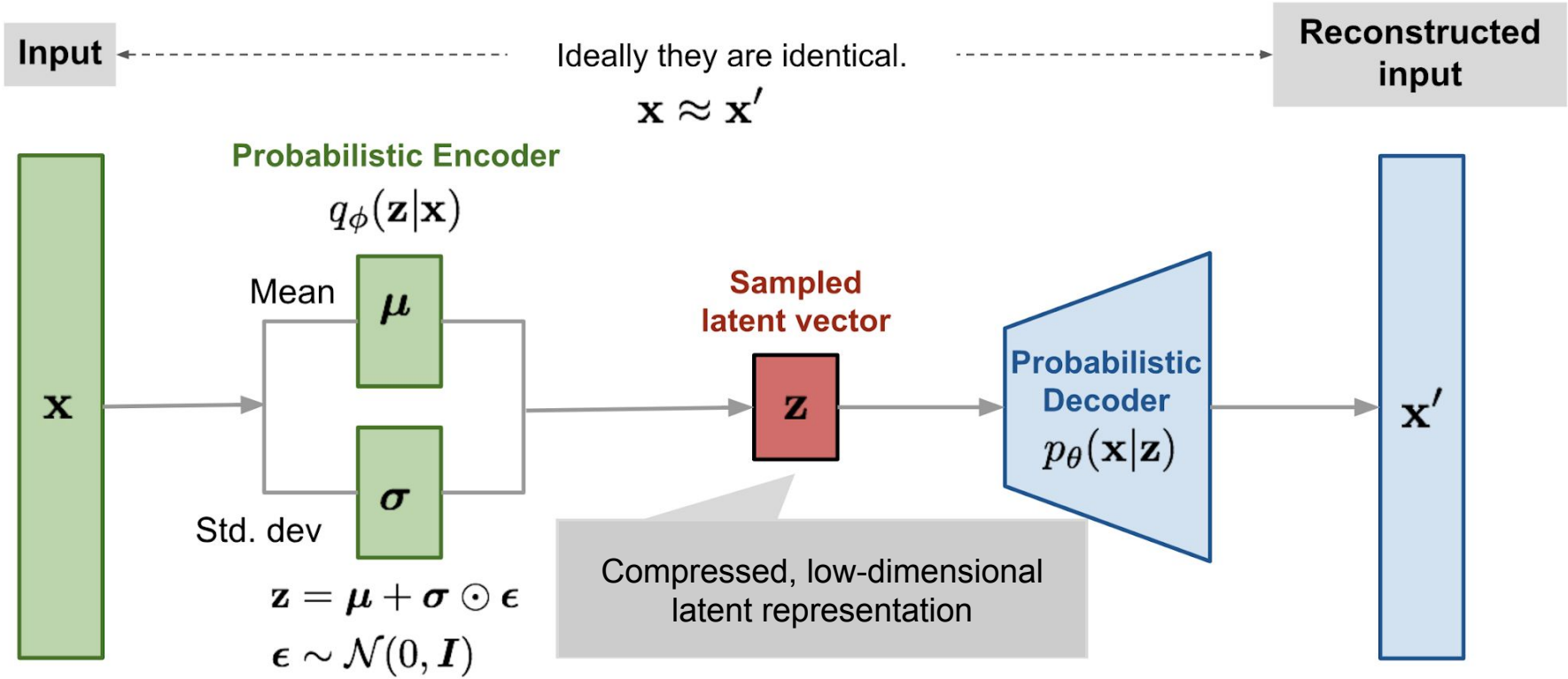


Image Source

Variational Autoencoder (VAE)

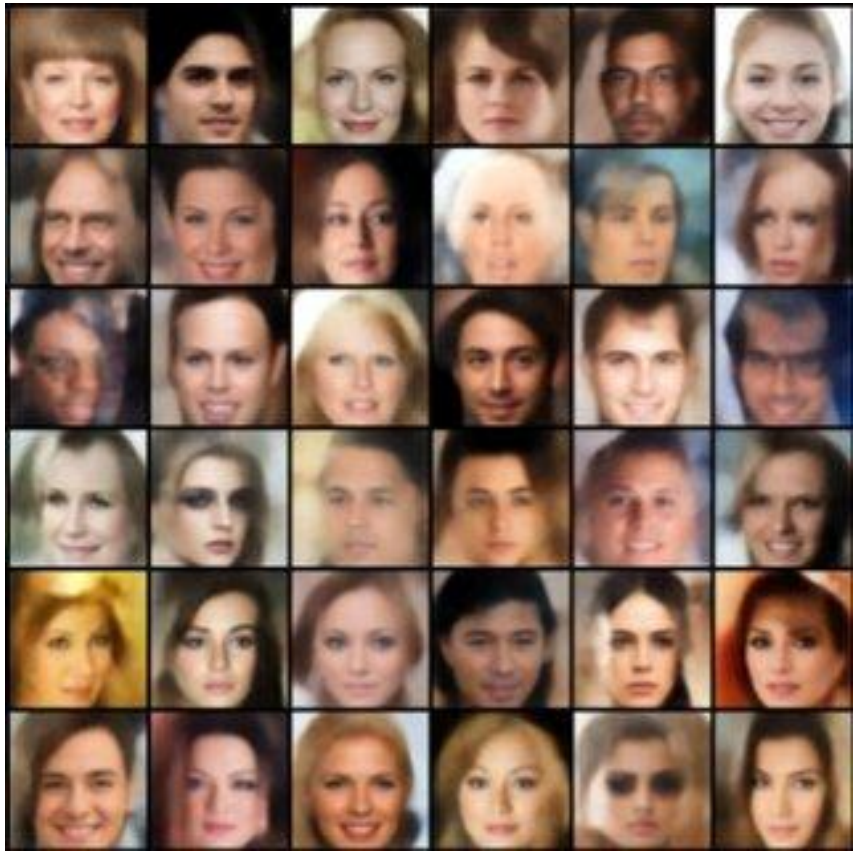


Image Source

Normalizing Flow

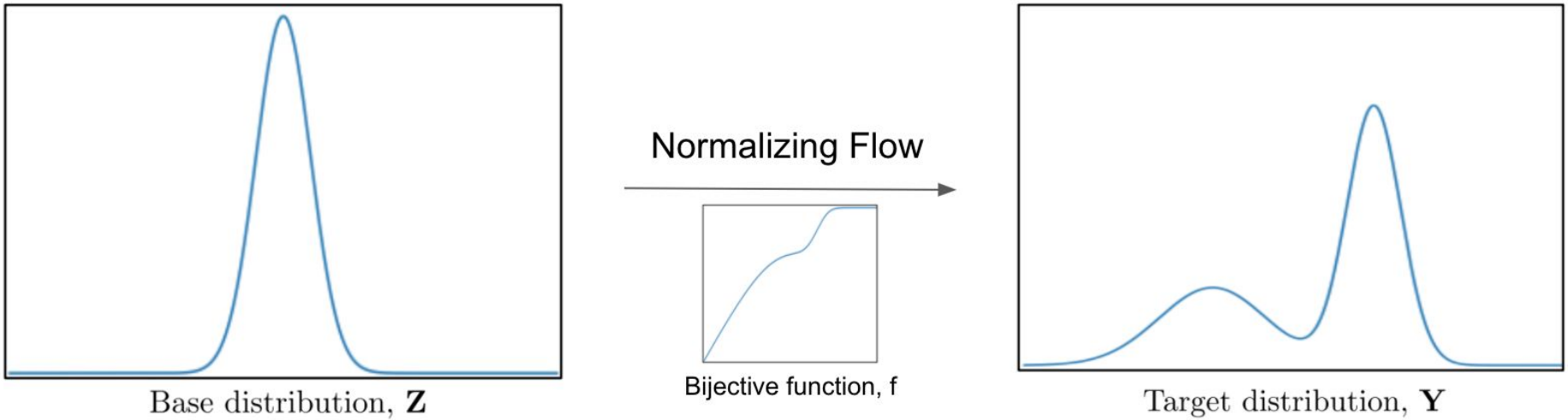
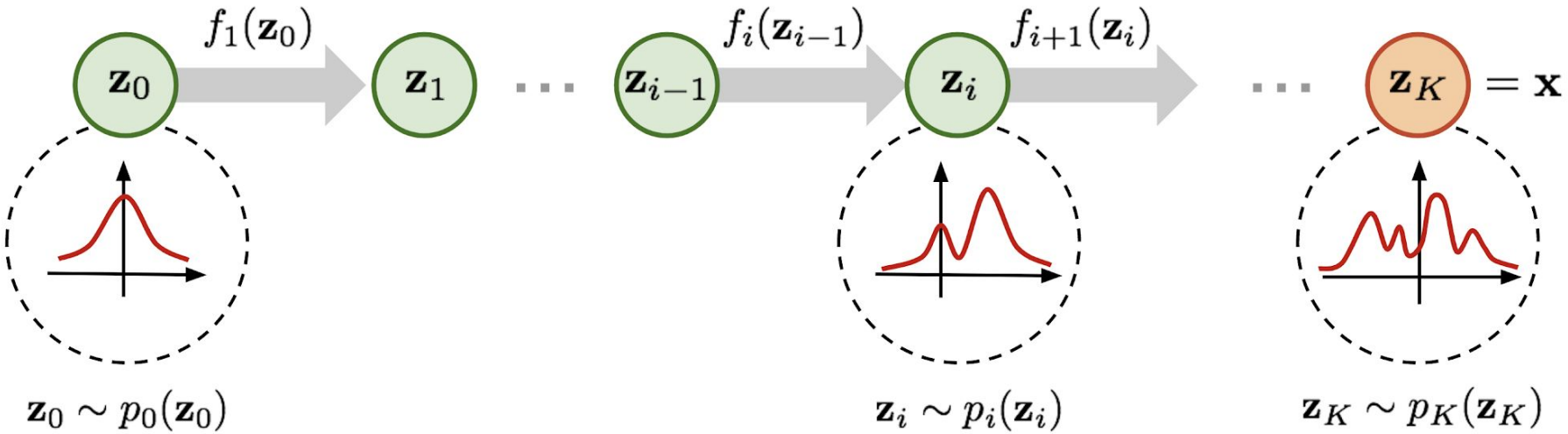


Image Source

Change of Variables Formula

$$\log p(x) = \log p(z) + \log|J_F|$$

Normalizing Flow



Normalizing Flow

$$f(x_1, x_2) = \text{concat}\left(x_1, \tilde{f}(x_2; x_1)\right)$$

$$\tilde{f}(x_2; x_1) = s(x_1)x_2 + t(x_1)$$

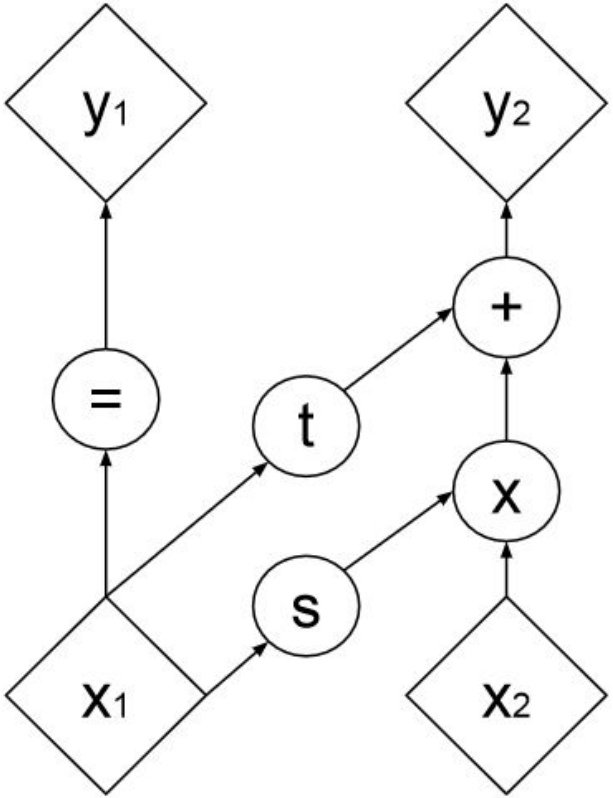
Normalizing Flow

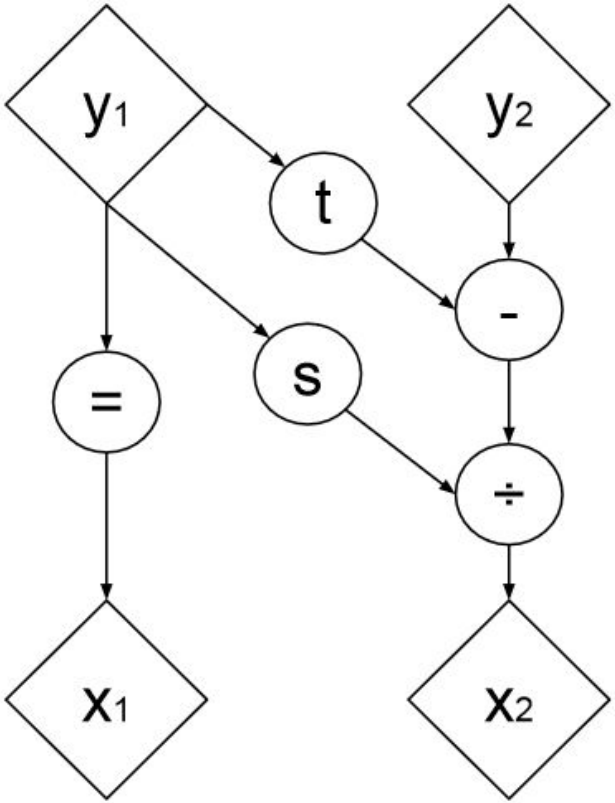
$$\det J_f = \det \begin{pmatrix} \frac{\partial x_1}{\partial x_1} & \frac{\partial x_1}{\partial x_2} \\ \frac{\partial \tilde{f}(x_2;x_1)}{\partial x_1} & \frac{\partial \tilde{f}(x_2;x_1)}{\partial x_2} \end{pmatrix}$$

$$= \det \begin{pmatrix} I & 0 \\ \frac{\partial \tilde{f}(x_2;x_1)}{\partial x_1} & \frac{\partial \tilde{f}(x_2;x_1)}{\partial x_2} \end{pmatrix}$$

$$= \det \frac{\partial \tilde{f}(x_2;x_1)}{\partial x_2} = \det s(x_1)$$

Normalizing Flow



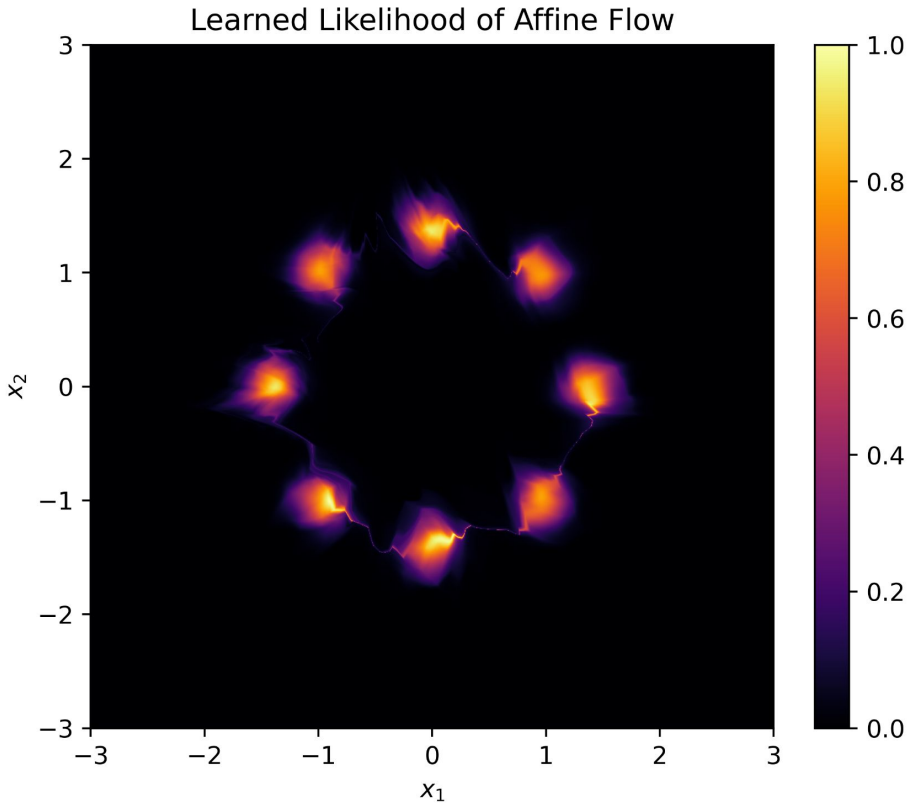


Dinh et al. (2016) - Density estimation using Real NVP

Normalizing Flow

Kingma et al. (2018) - Glow: Generative Flow with Invertible 1x1 Convolutions

Normalizing Flow



Recent Developments

Diffusion Models

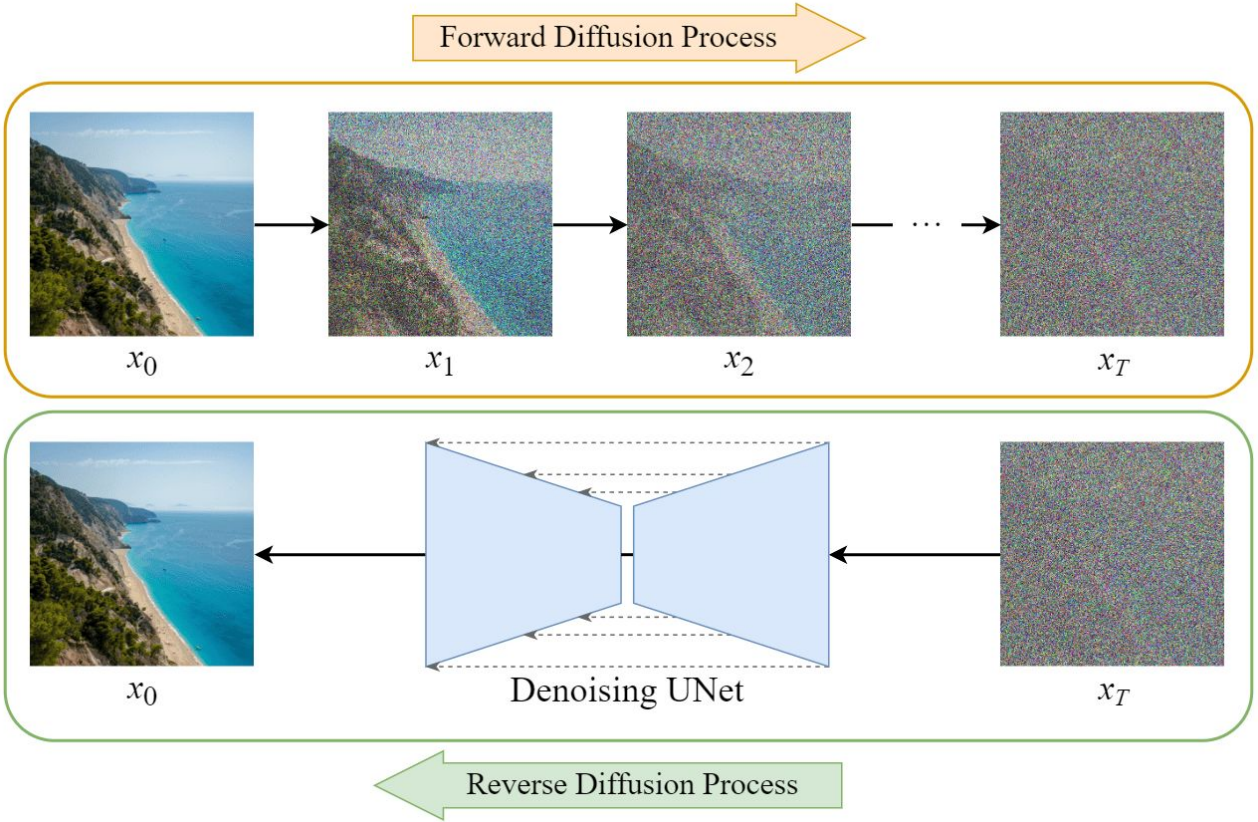
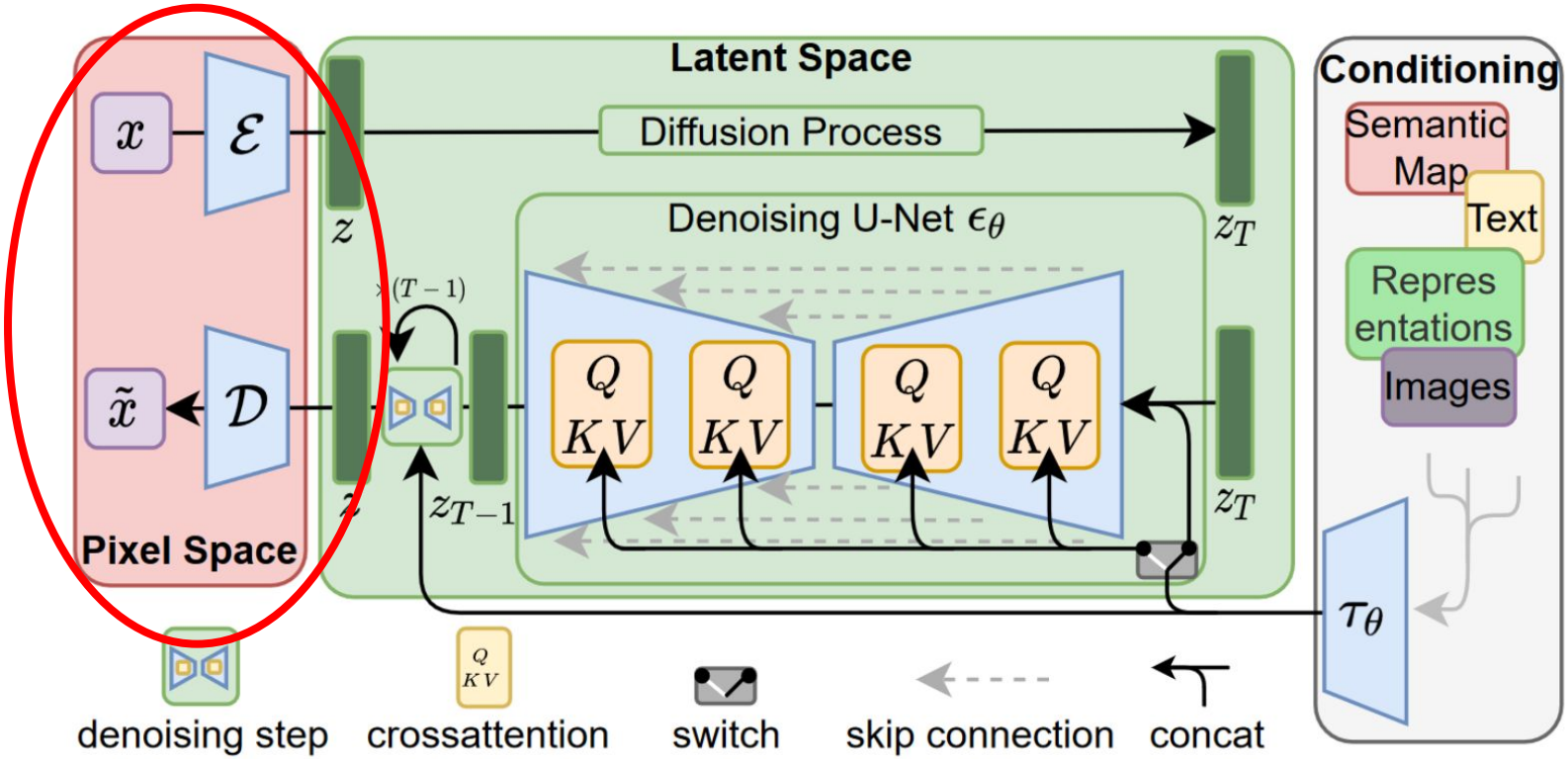


Image Source

Stable Diffusion

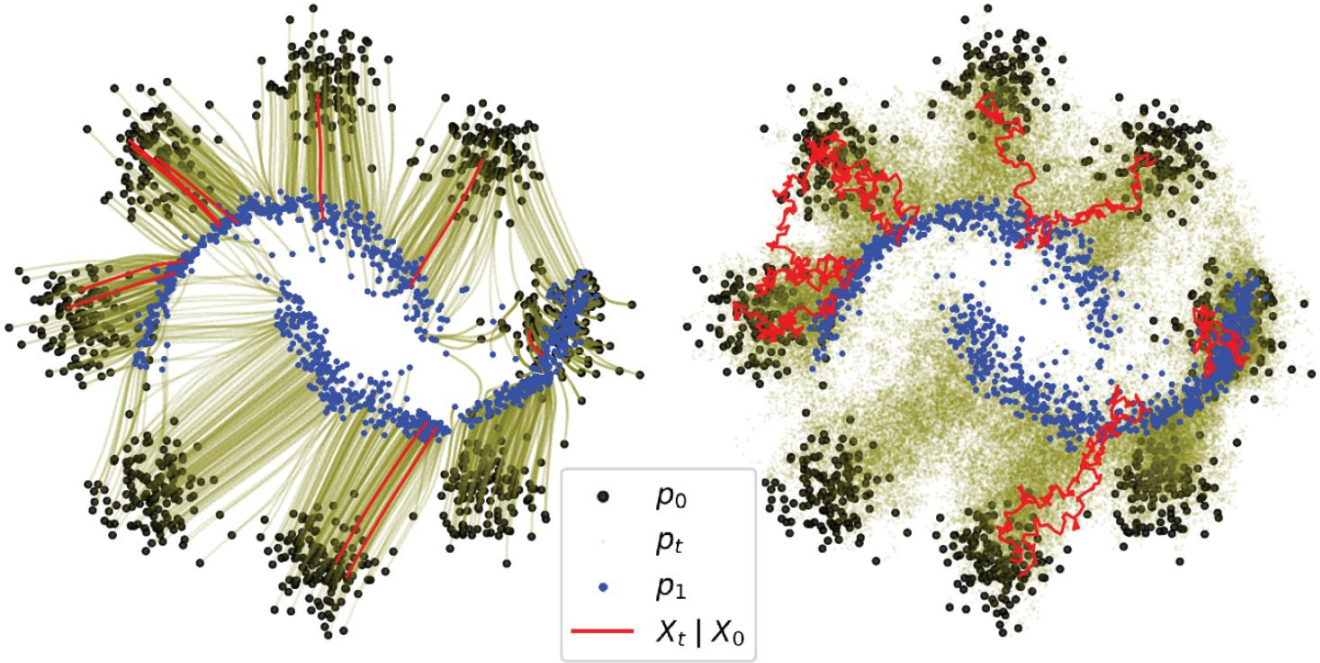


Rombach et al. (2021) - High-Resolution Image Synthesis with Latent Diffusion Models

Stable Diffusion

Image Source

Flow Matching

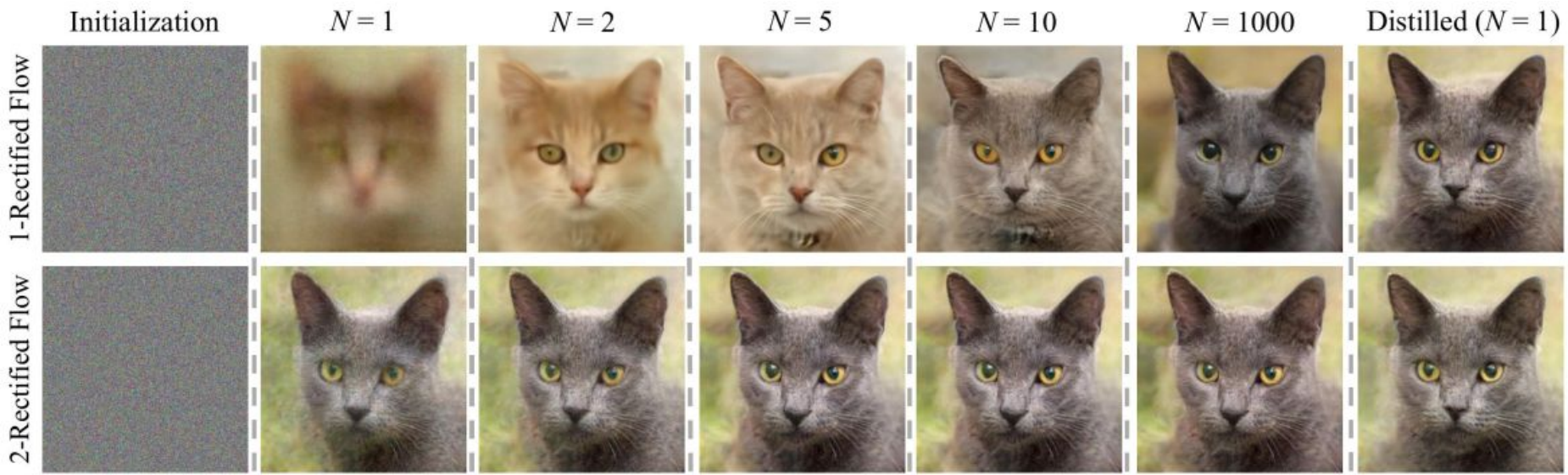


Flow Matching

Diffusion

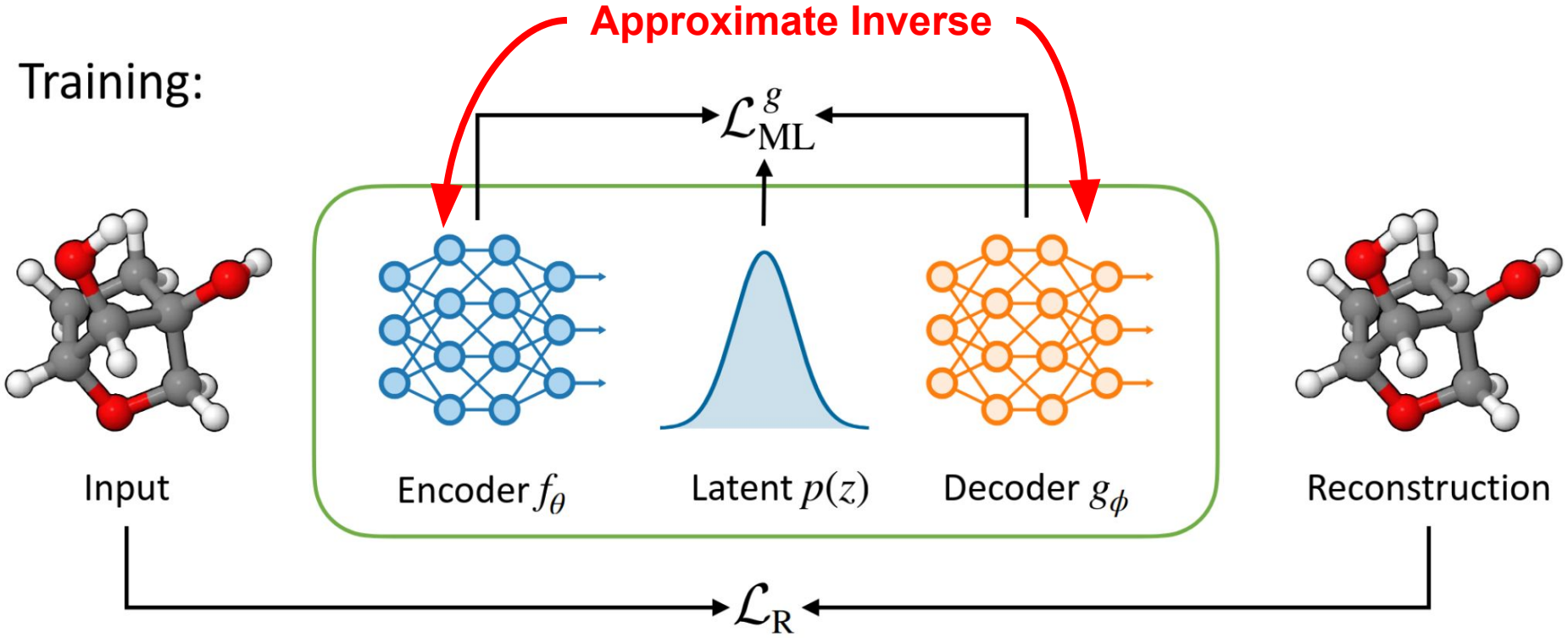
Tong et al. (2023) - Simulation-free Schrödinger bridges via score and flow matching

Flow Matching



Liu et al. (2022) - Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow

Free Form Flows



Draxler et al. (2023) - Free-form Flows: Make Any Architecture a Normalizing Flow

Generative Model Overview

Model	Training Speed	Sampling Speed	Exact Likelihood	Free-Form Jacobian
Gaussianization	★	▲	✓	✗
Coupling Flows	▲	▲	✓	✗
CNFs	●	●	✓	✓
Diffusion Models	▲	●	✗	✓
Flow Matching	▲	●	✓	✓

★	▲	●	●	✓	✗
No Backprop	Single-Step	Few-Step	Many-Step	Yes	No