CSCI 8960: Privacy-Preserving Data Analysis

PPML Summer School

Jaewoo Lee
jaewoo.lee@uga.edu

July 11, 2023

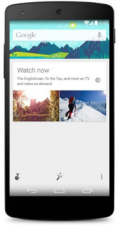
Department of Computer Science

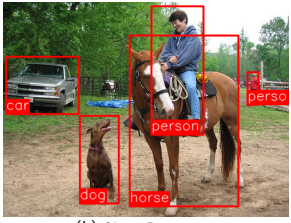
UNIVERSITY OF GEORGIA

Introduction

An Era of Deep Learning

(a) Recommendation

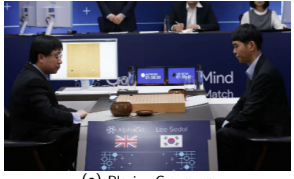




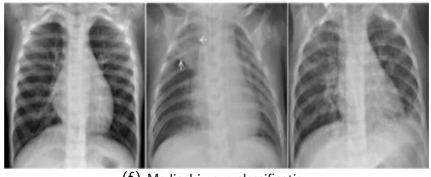
(b) Object Recognition

(c) Medical image classification

(d) Self-driving cars



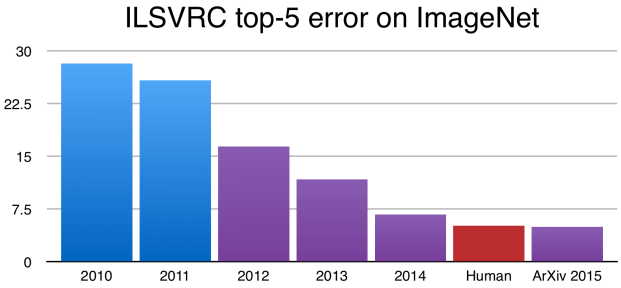
(e) Playing Go games



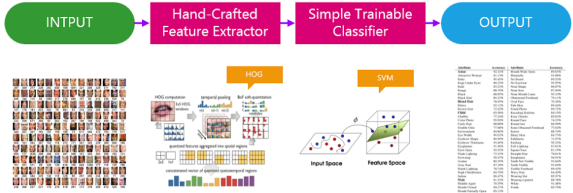
(f) Medical image classification

Fig. 1. Applications of deep learning

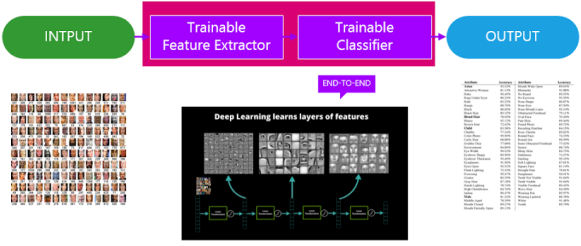
ImageNet Challenge



What is deep learning?



(a) Traditional Learning Models



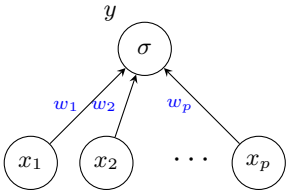
(b) Deep Learning Models

- Logistic regression

$$y = \phi(\mathbf{w}^\mathsf{T}\mathbf{x} + w_0) = \phi\left(\sum_{i=1}^{p} w_i x_i + w_0\right),$$

where

- $\mathbf{x} = (x_1, x_2, \ldots, x_p)$ feature vector
- $\mathbf{w} = (w_1, w_2, \ldots, w_p)$ weight vector (or parameter vector)
- w_0 bias (or intercept) term



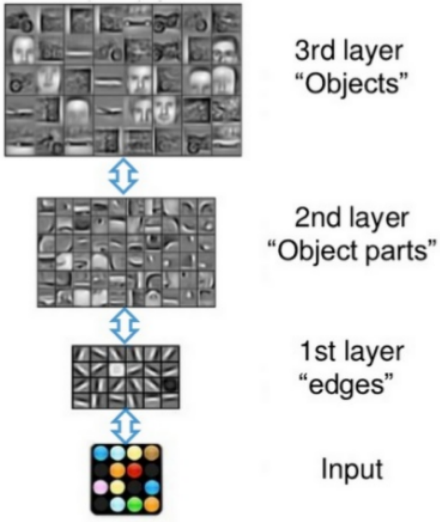
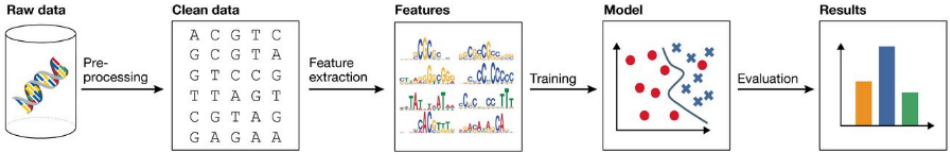


Fig. 3. Multiple layers of information processing

DL Pipeline



1. **Dataset**: an *i.i.d.* sample drawn from the population
   - $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
2. **Task**: predict Y given X
   - regression or classification?
3. **Model**: assumption on the (distribution) of data
   - linear, non-linear, parametric, non-parametric
4. **Loss**(a.k.a. score/cost): measure of how good a model is
   - $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$
5. **Training**: normally involve an optimization
   - (stochastic) gradient descent, Newton's method
6. **Evaluation**: validate the learning outcome

MLP

Multilayer Perceptrons

- Multiple *layers* of units
- Connecting units in one layer to those in the subsequent layer
- Feed-forward neural network
- No cycle
- All input units are connected to all output units (fully connected layer).

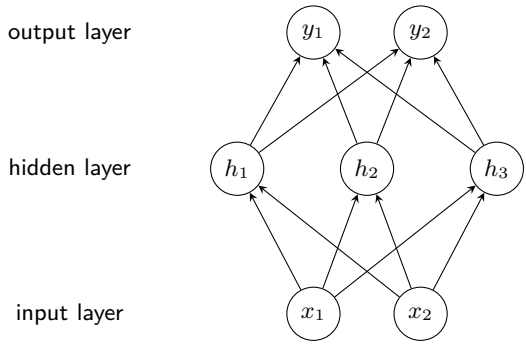


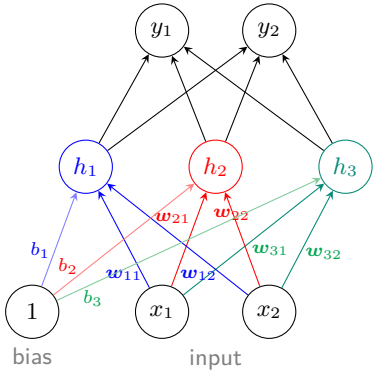
Fig. 4. Simple MLP

Use machine learning algorithms as a black box!

- Pre-activation
  - $z_1 = w_{11}x_1 + w_{12}x_2$
  - $z_2 = w_{21}x_1 + w_{22}x_2$
  - $z_3 = w_{31}x_1 + w_{32}x_2$
- Matrix notation

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

  - $\mathbf{h} = \phi(\underbrace{\mathbf{W}\mathbf{x} + \mathbf{b}}_{\text{pre-activation}})$



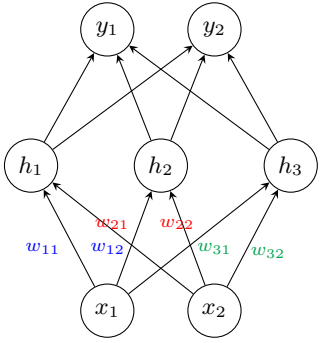
- $h_i = \phi(\mathbf{w}_i^\intercal \mathbf{x} + b)$
- Matrix notation

$$\mathbf{z} = \underbrace{\mathbf{W}\mathbf{x} + \mathbf{b}}_{\text{pre-activation}}$$

$$\mathbf{h} = (h_1, h_2, h_3) = \phi(\mathbf{z}),$$

where $\mathbf{W} \in \mathbb{R}^{m \times n}$.

- $\mathbf{W}$: weight matrix
- $\mathbf{b}$: bias term
- $\mathbf{z}$: pre-activation
- ϕ: activation function



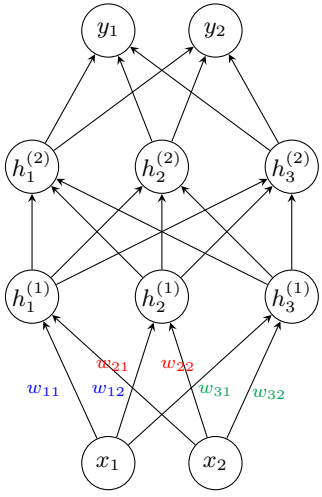
What if we have two hidden layers?

- $h_i = \phi(\mathbf{w}_i^\intercal \mathbf{x} + b)$
- Matrix notation

$$\mathbf{h}^{(1)} = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{h}^{(2)} = \phi(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

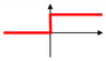
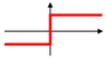
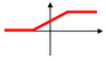
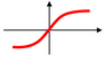
- Hierarchy

$$\mathbf{h}^{(2)} = \phi(\mathbf{W}^{(2)}\phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)})$$



Activation functions

- Determines what (or whether) to pass on from each neuron to the next layer
- Creates Non-linearity in the network

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

Is deep learning difficult?

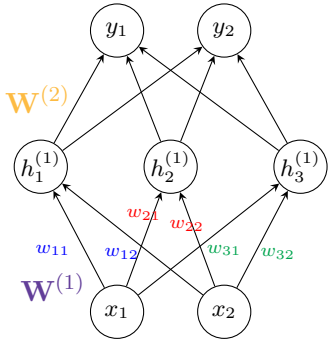
- Deep models:

$$\mathbf{y} = \phi\left(\mathbf{W}^{(2)}\phi(\mathbf{W}^{(1)}\mathbf{x})\right)$$

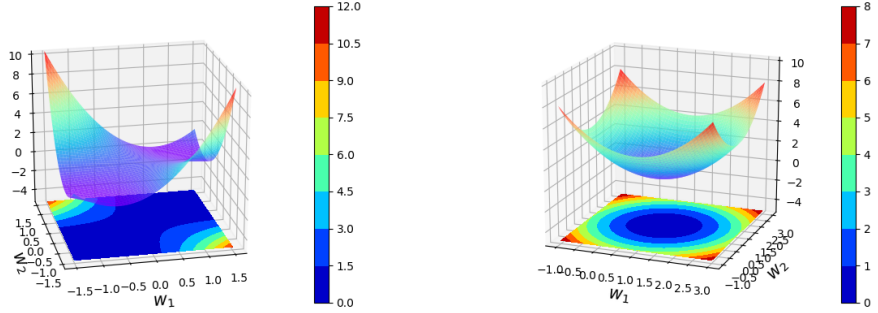
- Classical Models:

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x})$$

- Loss: $L(W) = \frac{1}{2}\|\hat{Y}(W, X) - Y\|_{\mathsf{F}}^2$
- Simplistic case: $\phi(x) = x$.
  - $W^{(1)} = w_1$, $W^{(2)} = w_2$, $X = Y = 1$



$$L_{\mathsf{deep}}(W) = (w_1 w_2 - 1)^2 \quad \textbf{VS} \quad L_{\mathsf{lin}}(W) = (w_1 - 1)^2 + (w_2 - 1)^2$$



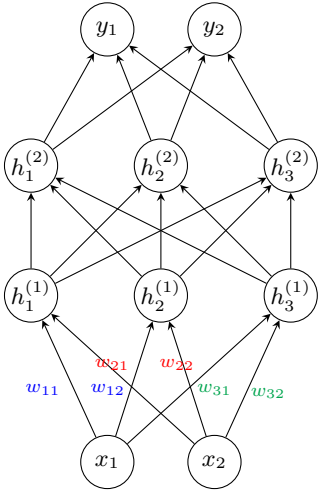
(a) L_{deep} (b) L_{lin}

Fig. 5. Landscape of loss functions $L(W)$

The depth of NN creates non-convex loss surfaces.

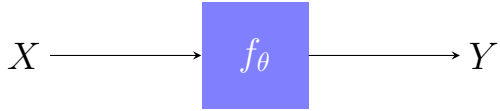
Model

- What functions can be expressed by my model?
  - A NN can be viewed as a function (or a mapping).
  - $f_{\boldsymbol{\theta}} \ : \ \mathbb{R}^2 \to \mathbb{R}^2$
  - $f_{\boldsymbol{\theta}} \ : \ \mathbf{x} \mapsto \mathbf{y}$
- Model parameter
  - $\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \mathbf{W}^{(3)}, \mathbf{b}^{(3)}\}$



Pipeline

- We want learn a map $f : \mathcal{X} \to \mathcal{Y}$.



  - input $\{(x_i, y_i)\}$, y_i=f evaluated at x_i
  - output $\{\hat{y}_i = f_\theta(x_i)\}$
  - parameter θ: weight vectors

- Training: optimize θ to approximate the mapping f

Cost Functions

What to optimize?

- Define a distribution over a variable to predict $p(y|x;\theta)$
- Maximum Likelihood Principle

$$L(\theta) = \mathbb{E}_{x,y\sim p_{\text{data}}}[\log p_{\text{model}}(y \mid x)]$$

- Example: when $p_{\text{model}}(y \mid x) = \mathcal{N}(y; f(x;\theta), \mathbf{I})$, MLE reduces to

$$L(\theta) = \frac{1}{2}\,\mathbb{E}_{x,y\sim p_{\text{data}}}\left[\|y - f(x;\theta)\|^2\right].$$

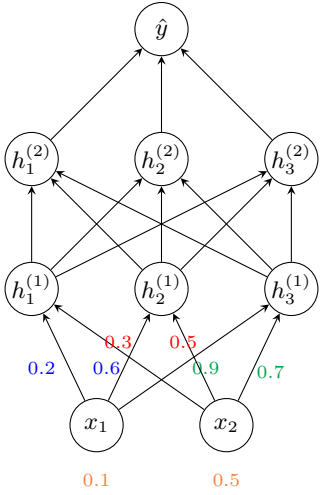
Forward Phase

x_1	x_2	y
0.1	0.5	1
0.2	0.7	0
0.9	0.1	0

Table 1. Input data

- Assume $\phi(z) = \max(0, z)$ (ReLU activation function)

$$z_1^{(1)} = 0.2 \cdot 0.1 + 0.6 \cdot 0.5 = 0.32$$
$$z_2^{(1)} = 0.3 \cdot 0.1 + 0.5 \cdot 0.5 = 0.28$$
$$z_3^{(1)} = 0.9 \cdot 0.1 + 0.7 \cdot 0.5 = 0.44$$

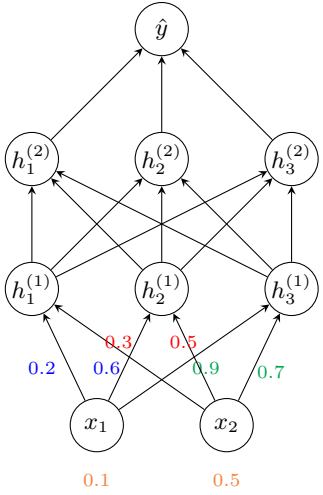


Forward Phase

x_1	x_2	$\hat{y}$	y
0.1	0.5	0.8	1
0.2	0.7	0.3	0
0.9	0.1	0.1	0

Table 2. After the forward step

- For each $\mathbf{x} \in D$, compute the *loss*
- Average the losses

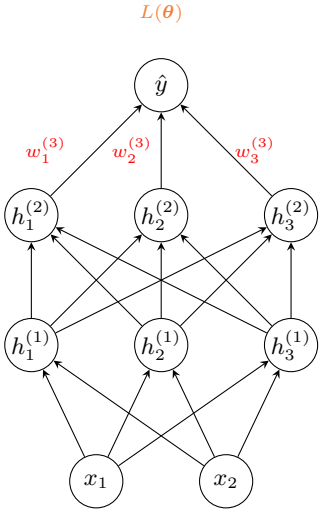


$$L(\boldsymbol{\theta}) = \frac{1}{3}\left\{(0.8-1)^2 + (0.3-0)^2 + (0.1-0)^2\right\}$$

Backward Phase

- Backpropagating *error correction* info.
  - How should we modify $\boldsymbol{\theta}_l = \{\mathbf{w}^{(l)}, \mathbf{b}^{(l)}\}$?
  - It is given by the gradient: $\dfrac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_l}$.
  - Let's start with $\dfrac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_3}$.

$$\hat{y} = (\mathbf{w}^{(3)})^\intercal \mathbf{h}^{(2)} = w_1^{(3)} h_1^{(2)} + w_2^{(3)} h_2^{(2)} + w_3^{(3)} h_3^{(2)}$$



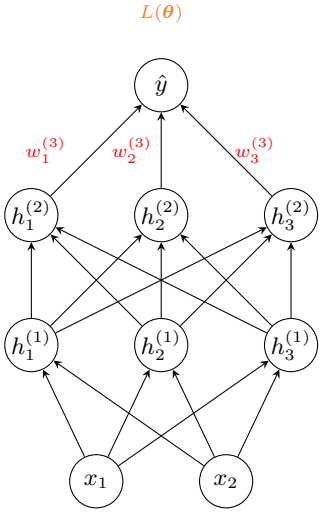
Backward Phase: $\boldsymbol{\theta}_3$

- Recall that $L(\boldsymbol{\theta}) = \frac{1}{2}(y - \hat{y})^2$.

$$\hat{y} = (\mathbf{w}^{(3)})^\intercal \mathbf{h}^{(2)} = w_1^{(3)} h_1^{(2)} + w_2^{(3)} h_2^{(2)} + w_3^{(3)} h_3^{(2)}$$

  - By the chain rule, we have

$$\frac{\partial L}{\partial \boldsymbol{\theta}_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{\theta}_3} .$$



Backward Phase: $\boldsymbol{\theta}_2$

- How about $\dfrac{\partial L}{\partial \boldsymbol{\theta}_2}$?
  - Again, by the chain rule, we have

$$\frac{\partial L}{\partial \boldsymbol{\theta}_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{\theta}_3}$$

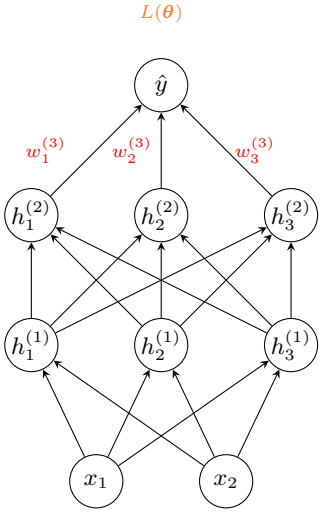
$$\frac{\partial L}{\partial \boldsymbol{\theta}_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}_2}{\partial \boldsymbol{\theta}_2}.$$

From the forward phase, we have

$$\hat{y} = (\mathbf{w}^{(3)})^\intercal \mathbf{h}^{(2)}$$

$$\mathbf{h}^{(2)} = \phi(\mathbf{z}^{(2)})$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)} \mathbf{h}^{(1)}.$$



Backward Phase: $\boldsymbol{\theta}_1$

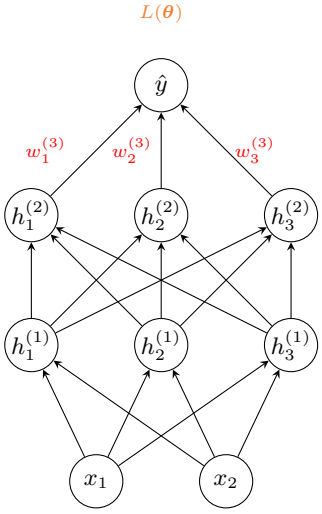
- Now we compute $\dfrac{\partial L}{\partial \boldsymbol{\theta}_1}$.
  - By the chain rule, we have

$$\frac{\partial L}{\partial \boldsymbol{\theta}_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{\theta}_3}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_2} = \underbrace{\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}}}_{\frac{\partial L}{\partial \mathbf{z}^{(2)}}} \frac{\partial \mathbf{z}^{(2)}}{\partial \boldsymbol{\theta}_2}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{z}_1} \frac{\partial \mathbf{z}_1}{\partial \boldsymbol{\theta}_1}$$

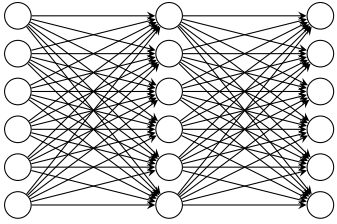
$$= \frac{\partial L}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{z}_1} \frac{\partial \mathbf{z}_1}{\partial \boldsymbol{\theta}_1}$$



CNN

Review: MLP

- MLPs are *universal* function approximators.
  - can learn a complex mapping $f : \mathcal{X} \to \mathcal{Y}$ between X and Y

$\mathbf{x}$  $\hat{y} = f_\theta(\mathbf{x})$

- Given input feature *vector* $\mathbf{x}$, f_θ returns desired output.
  - $f_\theta : \mathbf{x} \mapsto \hat{\mathbf{y}}$
  - *fully connected*
  - Forward step : matrix multiplication
    - $\mathbf{h}^{(1)} = \phi(\mathbf{W}_1^\intercal \mathbf{x})$
    - $\mathbf{h}^{(2)} = \phi(\mathbf{W}_2^\intercal \mathbf{h}^{(1)})$

MLP can be too complex!

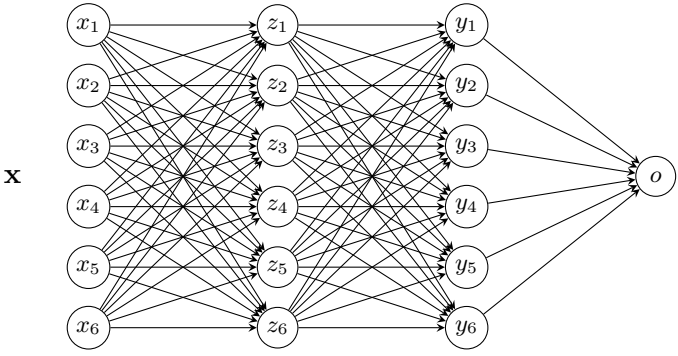
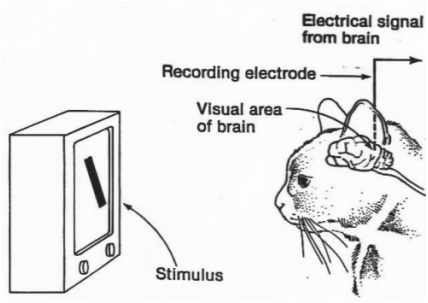
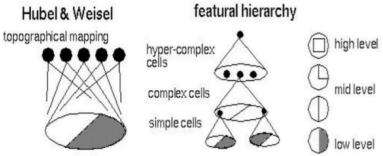


Fig. 6. A fully connected network with 2 hidden layers

- How *many* parameters to learn?
  - If we have n units in each layers, what is the total number of parameters?
  - high complexity may lead to *overfitting* and local optima
  - requires careful *initialization*

Hubel and Wiesel 1959





- measured neural responses in cat's striate cortex
- suggested a hierarchy of *feature detectors* in the visual cortex
- higher level features responding to patterns of activations in lower level cells

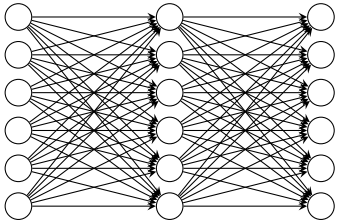
- Computer vision needs some robustness:
  - *translation invariance*

Cat

Cat

- need a network that recognizes the cat *regardless of its location*
- location might not be an important factor

28×28



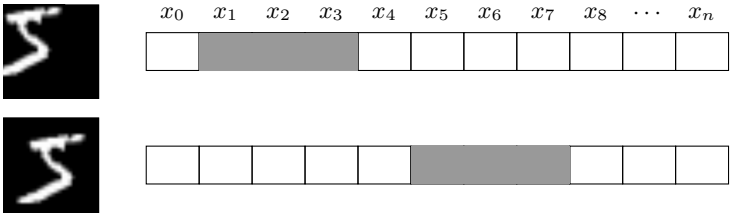
$\hat{y} = f_\theta(\mathbf{x})$

10 output units

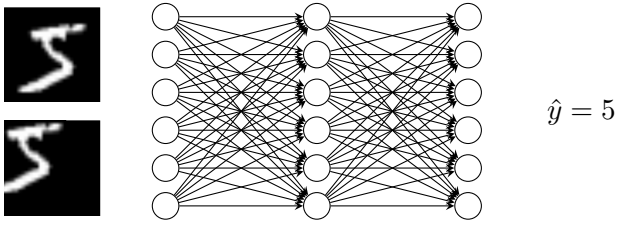
1000 hidden units

- How many parameters do we need for the first layer?
  - Parameters: $280 \times 280 \times 1000 = 78.4M$
- What if the input is slightly shifted?

MLP

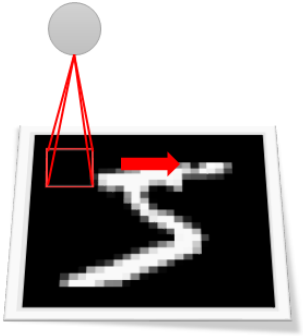


- A unit's activation is dependent on the outputs of *all* nodes in the previous layer.



- Each unit in the hidden layer is affected by the *entire* inputs.
- How can we detect the same feature at different positions?
- expect low-level features to be *local*
- expect high-level features to be *coarser*

Local Pattern



- Let's have a feature detector
- An interesting pattern can be anywhere in the image.
  - Define a kernel (or filter) $\mathbf{W}$ that search for a specific pattern.
  - The detector slides $\mathbf{W}$ over the given image.
  - Inspect $\langle \mathbf{X}[\cdot, \cdot], \mathbf{W} \rangle$ and determine if the region contains the pattern
- Let's have multiple of them!

Convolution

What is special about $\langle \mathbf{X}[\cdot,\cdot], \mathbf{W} \rangle$ operation?

- Suppose we have a 1D image $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ and a 1D kernel $\mathbf{g} = (g_1, \ldots, g_m)$.
- $\langle \mathbf{X}[\cdot,\cdot], \mathbf{W} \rangle$ in 1D corresponds to

$$h[i] = (f * g)[i] = \sum_{j=1}^{m} g[j] \cdot f[i - j + m/2] \,.$$

$f =$

9	0	2	1	0	9

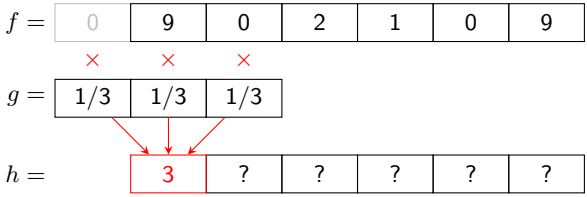
$g =$

1/3	1/3	1/3

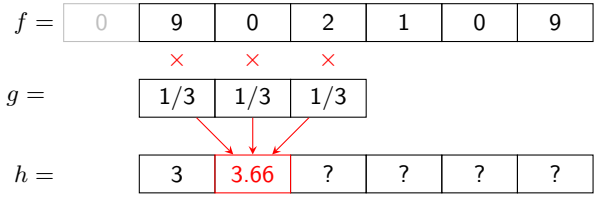
$h =$

?	?	?	?	?	?

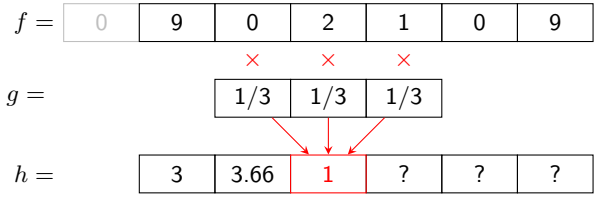
1D Convolution



1D Convolution

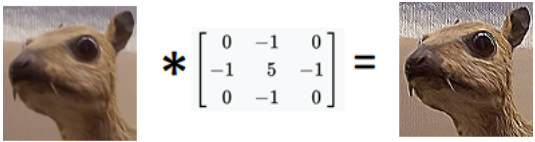


1D Convolution

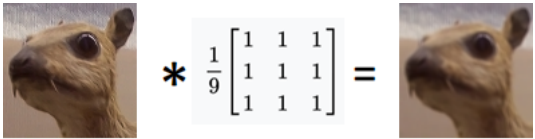


- This can be easily extended to 2D.
- But, is h useful?

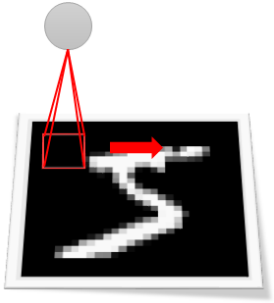
(a) Edge detection



(b) Sharpen



(c) Blur

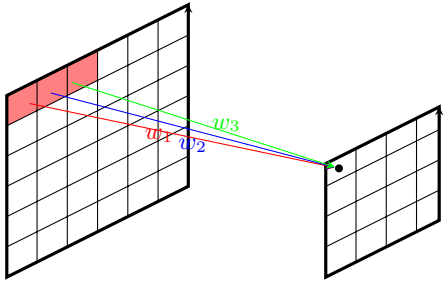


- But how do we know which filter to apply?

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

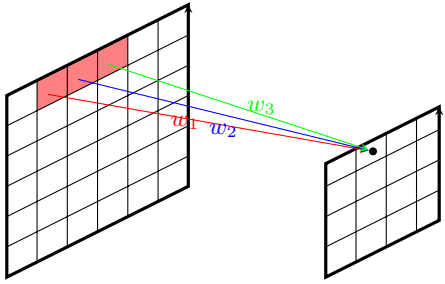
  - Let our NN *learn* it from the data.
  - $\mathbf{W}$ are *shared* across different locations.

Convolution



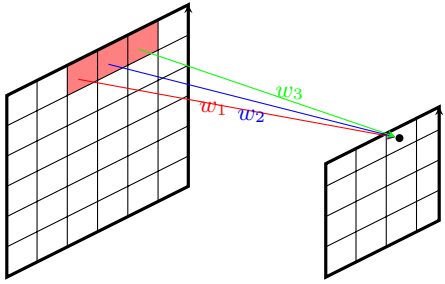
- $h_1 = w_1x_1 + w_2x_2 + w_3x_3$

Convolution



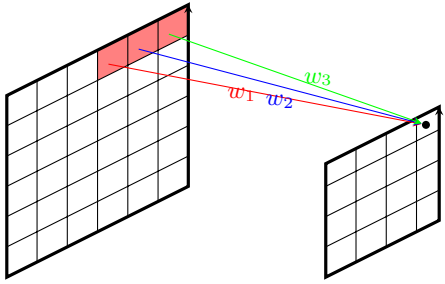
- $h_2 = w_1 x_2 + w_2 x_3 + w_3 x_4$

Convolution



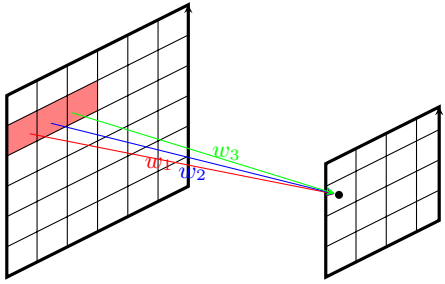
- $h_3 = w_1 x_3 + w_2 x_4 + w_3 x_5$

Convolution

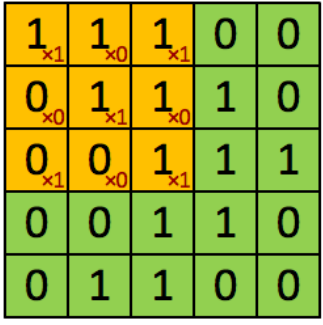


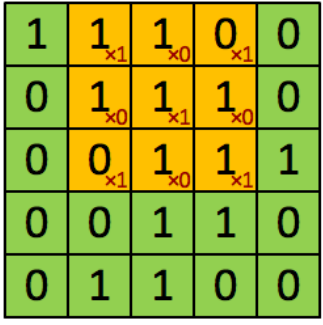
- $h_4 = w_1 x_4 + w_2 x_5 + w_3 x_6$

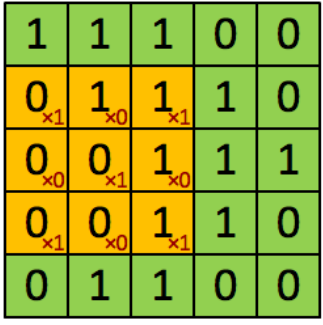
Convolution

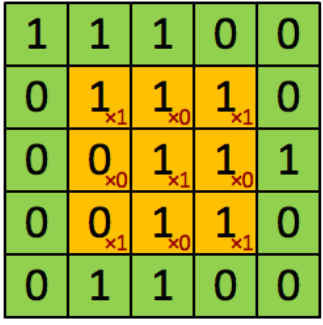


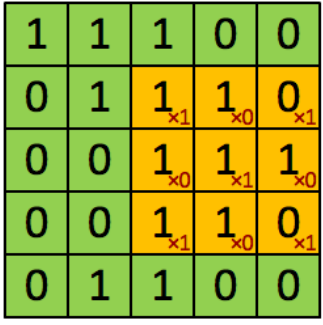
- $h_5 = w_1 x_7 + w_2 x_8 + w_3 x_9$

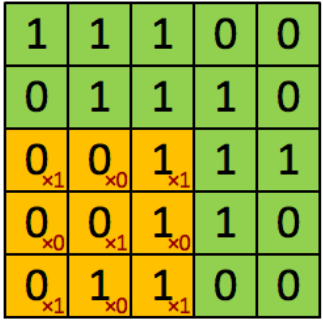

1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0
Image
4
Convolved Feature


1 1×1 1×0 0×1 0
0 1×0 1×1 1×0 0
0 0×1 1×0 1×1 1
0 0 1 1 0
0 1 1 0 0
Image
4 3
Convolved Feature

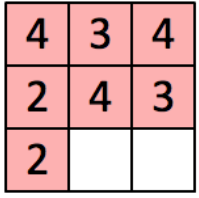

1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0
×1 ×0 ×1
×0 ×1 ×0
×1 ×0 ×1
Image
4 3 4
2
Convolved Feature


1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0
1 0 1
0 1 0
1 0 1
Image
4 3 4
2 4
Convolved
Feature


1 1 1 0 0
0 1 1×1 1×0 0×1
0 0 1×0 1×1 1×0
0 0 1×1 1×0 0×1
0 1 1 0 0
Image
4 3 4
2 4 3
Convolved Feature

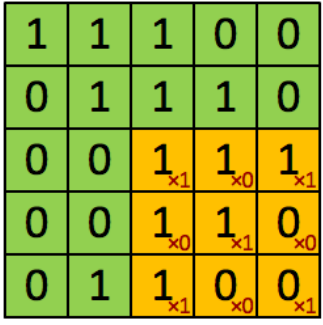


Image

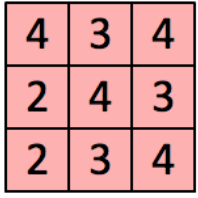


Convolved Feature


1 1 1 0 0
0 1 1 1 0
0 0×1 1×0 1×1 1
0 0×0 1×1 1×0 0
0 1×1 1×0 0×1 0
Image
4 3 4
2 4 3
2 3
Convolved Feature



Image



Convolved Feature

What does a Convolution do?

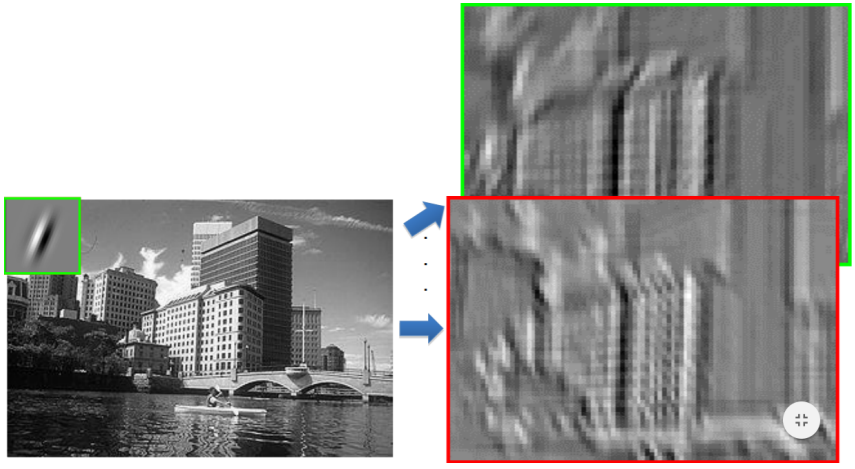


Fig. 8. Image credit: I. Kokkinos

Output Size

In our example, we have

- a 5×5 image,
- a Kernel (or filter) of size 3×3, and
- Stride of 1.

This gives an output of size

$$\frac{N - K}{S} + 1 \,.$$

- Suppose we have
  - 7×7 image
  - 3×3 kernel
  - stride of 3
- What is the output size?

Zero Padding

0	0	0	0	0	0	0	0	0	0
0	77	80	82	78	70	82	82	140	0
0	83	78	80	83	82	77	94	151	0
0	87	82	81	80	74	75	112	152	0
0	87	87	85	77	66	99	151	167	0
0	84	79	77	78	76	107	162	160	0
0	86	72	70	72	81	151	166	151	0
0	78	72	73	73	107	166	170	148	0
0	76	76	77	84	147	180	168	142	0
0	0	0	0	0	0	0	0	0	0

- image size $= 8 \times 8$
- kernel $= 3 \times 3$ with stride of 1
- What is the size of output? $\frac{10-3}{1} + 1 = 8$

In General,

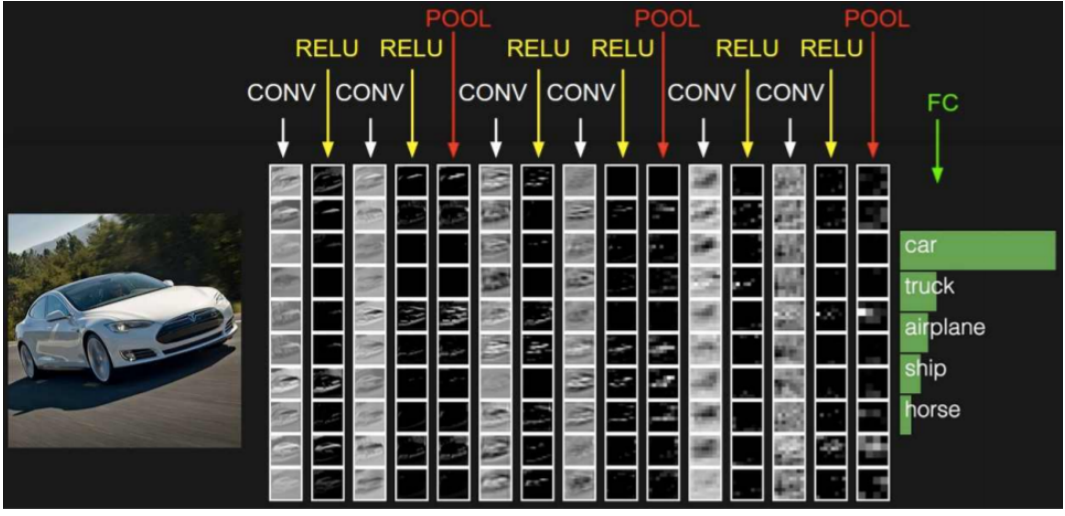
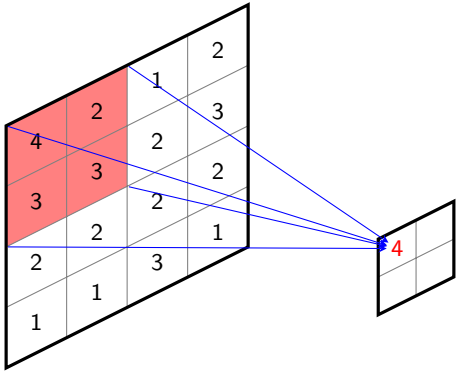
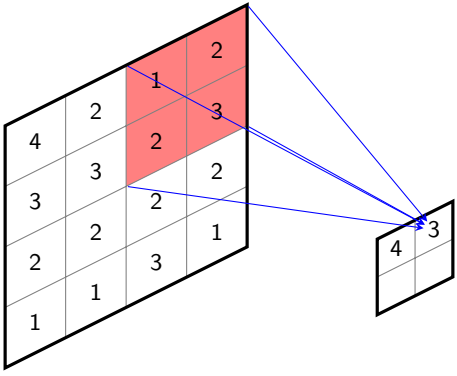


Fig. 9. Image source: Andrej Karpathy

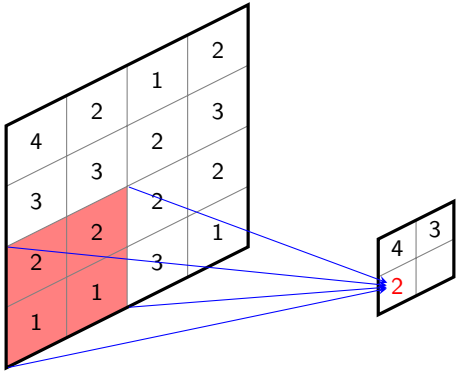
Pooling Layer



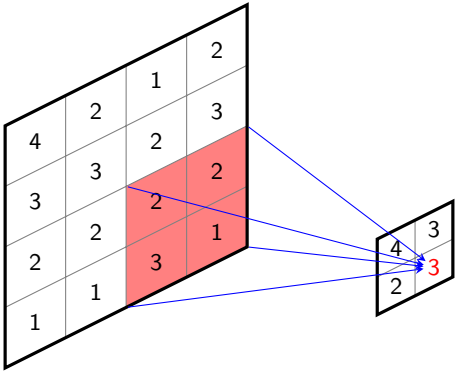
Pooling Layer



Pooling Layer



Pooling Layer



Pooling Layer

- What does the pooling layer do?
  - result in a smaller representation (shrunk image)
  - target feature values becomes less sensitive to small change in the image
  - stride > 1 is called "down sampling"
- Max pooling
- Average pooling
- L_p pooling ($p = 1 \Rightarrow$ average, $p = \infty \Rightarrow$ max pooling)

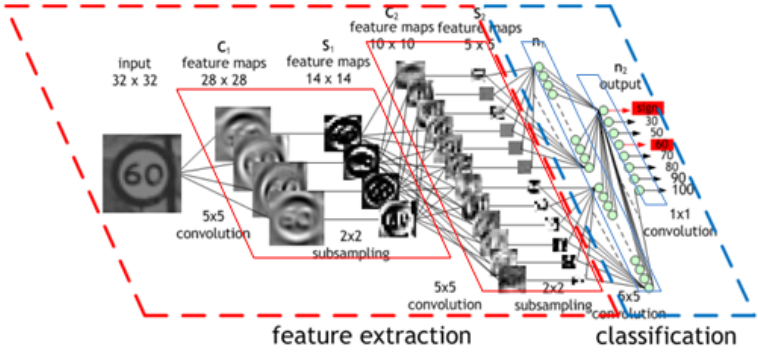


Fig. 10. Image source: NVIDIA

- The input to the next layer is *down sampled* image.
- Higher-layer filters see a larger region of the input than the equal-szied filters in the lower layers.

Feature Learning

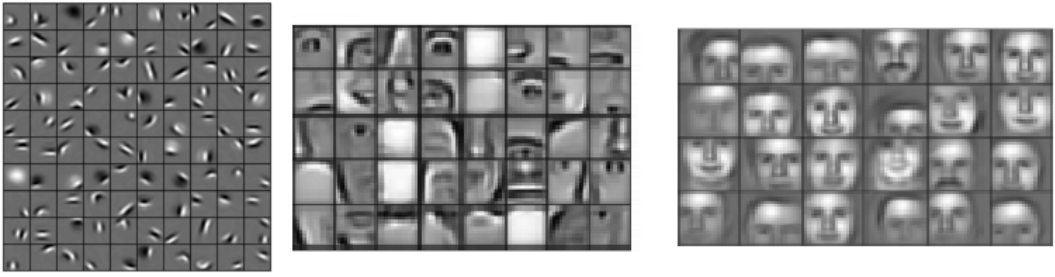


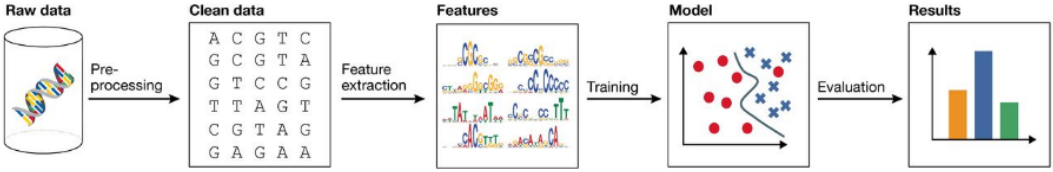
Fig. 11. Image source: Honglak Lee et al. 2011

PyTorch

What is PyTorch?

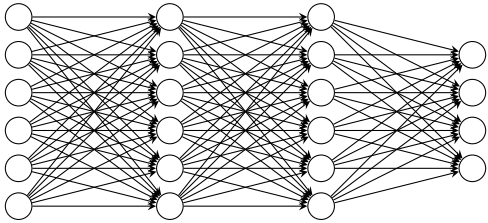
- Python implementation of deep learning tools
  - A collection of classes
  - https://pytorch.org
- Developed by Meta

Review: DL Pipeline



1. **Dataset**: an *i.i.d.* sample drawn from the population
   - $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
2. **Task**: predict Y given X
   - regression or classification?
3. **Model**: assumption on the (distribution) of data
   - linear, non-linear, parametric, non-parametric
4. **Loss**(a.k.a. score/cost): measure of how good a model is
   - $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$
5. **Training**: normally involve an optimization
   - (stochastic) gradient descent, Newton's method
6. **Evaluation**: validate the learning outcome

Building a Neural Network



$$\hat{y} = f_\theta(\mathbf{x})$$

- Input: MNIST dataset
  - 28×28 gray scale image
  - vectorize the image to $\mathbf{x} \in \mathbb{R}^{764}$
- Architecture
  - 1st hidden layer: 512 units
  - 2nd hidden layer: 512 units
  - Output layer: 10 units (10 classes)

Building a Neural Network (1)

```python
import torch.nn as nn

# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

Building a Neural Network (2)

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

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(28*28, 512)
        self.linear2 = nn.Linear(512, 512)
        self.classifier = nn.Linear(512, 10)

    def forward(self, x):                # define how the input tensor is processed
        x = self.flatten(x)
        out = F.relu(self.linear1(x))
        out = F.relu(self.linear2(out))
        logits = self.classifier(out)

        return logits

model = NeuralNetwork().to(device)
print(model)
```

Loading Datasets

Defining Datasets

```python
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",    # path to the directory containing the dataset
    train=True,
    download=True,  # If True, downloads the dataset from the internet
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

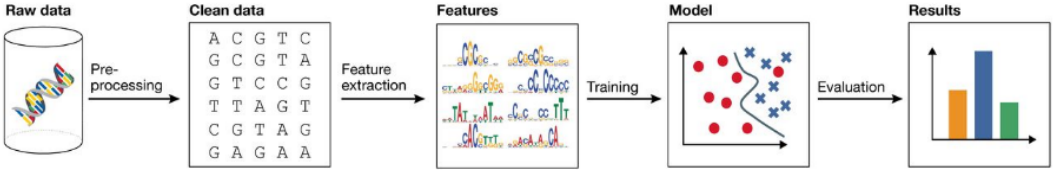
DataLoader

```python
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

Training NNs




```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

- `nn.MSELoss`
- `nn.NLLLoss`
- `nn.L1Loss`
- ...
- See here for other available functions.

Training NNs

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()

    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Testing the model

```python
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Training Loop

```
epochs = 5

for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```