## Advanced Control for Robotics (fall 2024)

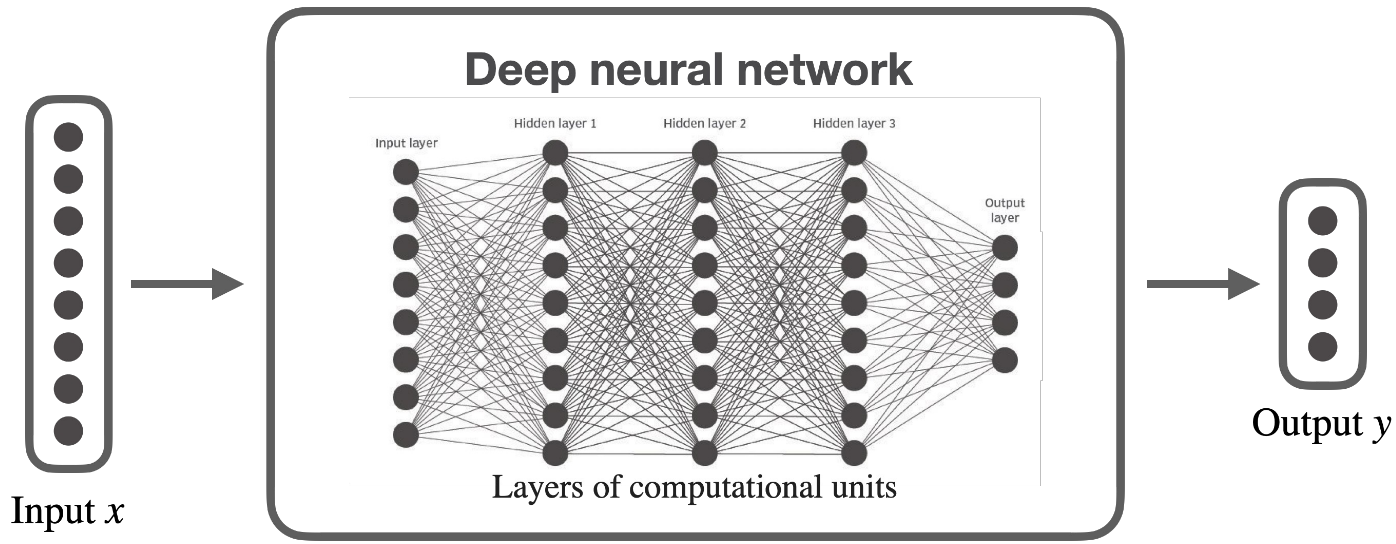
# Lecture Note 0
# Introduction to Neural Network

**Prof. Wei Zhang**
**Southern University of Science and Technology**

- **This lecture**: introduction to neural network
  - What is a neural network?
  - Key components of neural networks
  - A basis neural network structure: MLP
  - Loss functions for neural network learning
  - Optimizer in Neural Network Training
  - Construct a machine learning task in PyTorch

# What is a neural network

- A neural network is a mathematical model to approximate complex functions.



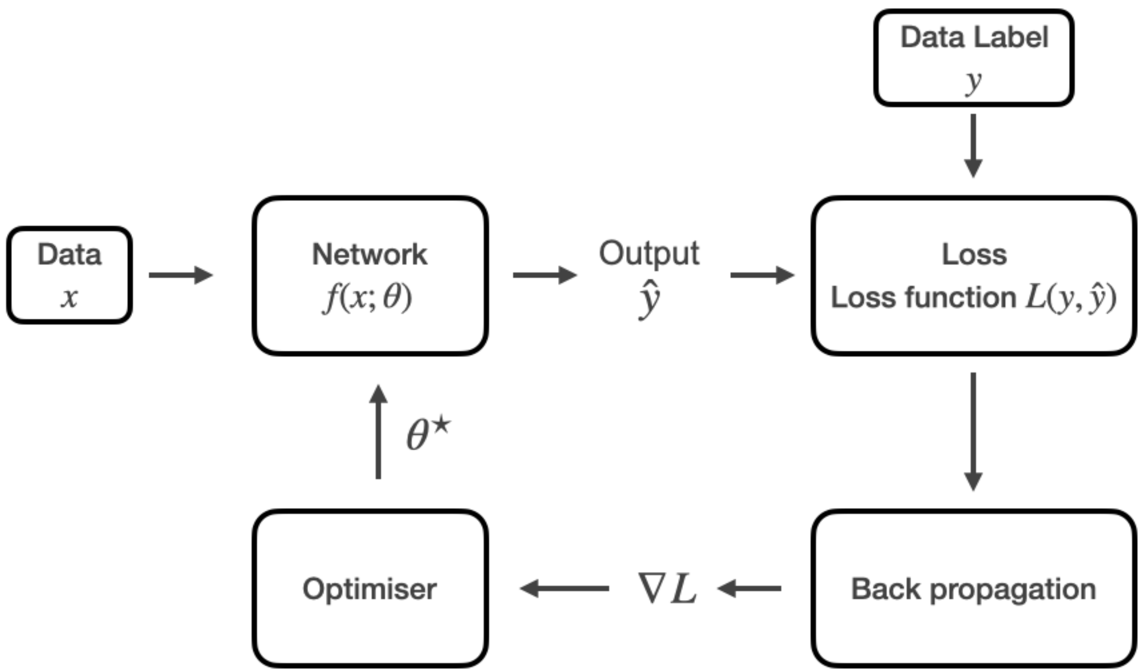

$$y = f(x;\ \theta)$$

- Approximate a function $f(x)$ that maps input data $x$ to output $y$ using numerical optimization:
  - $\hat{\theta} = \arg\min_{\theta} \mathcal{L}(f(x;\ \theta),\ y)$ ,

  where $\theta$ is the function's parameter

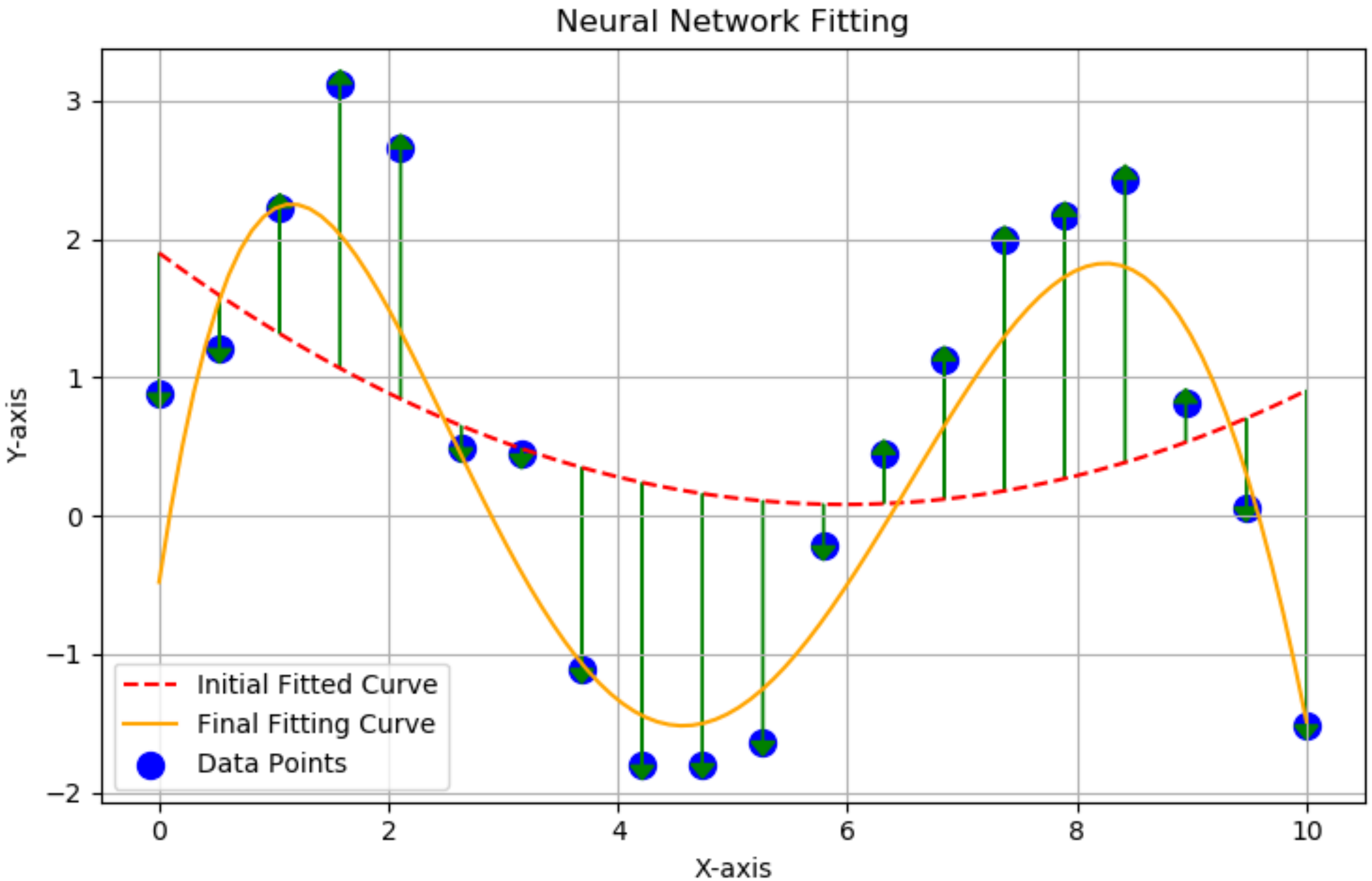
# Key Components of Neural Networks

- Neural networks consist of several key components that works together.
  - Data
  - Network structures: MLP/CNN etc.
  - Loss calculation
  - Back propagation
  - Optimization



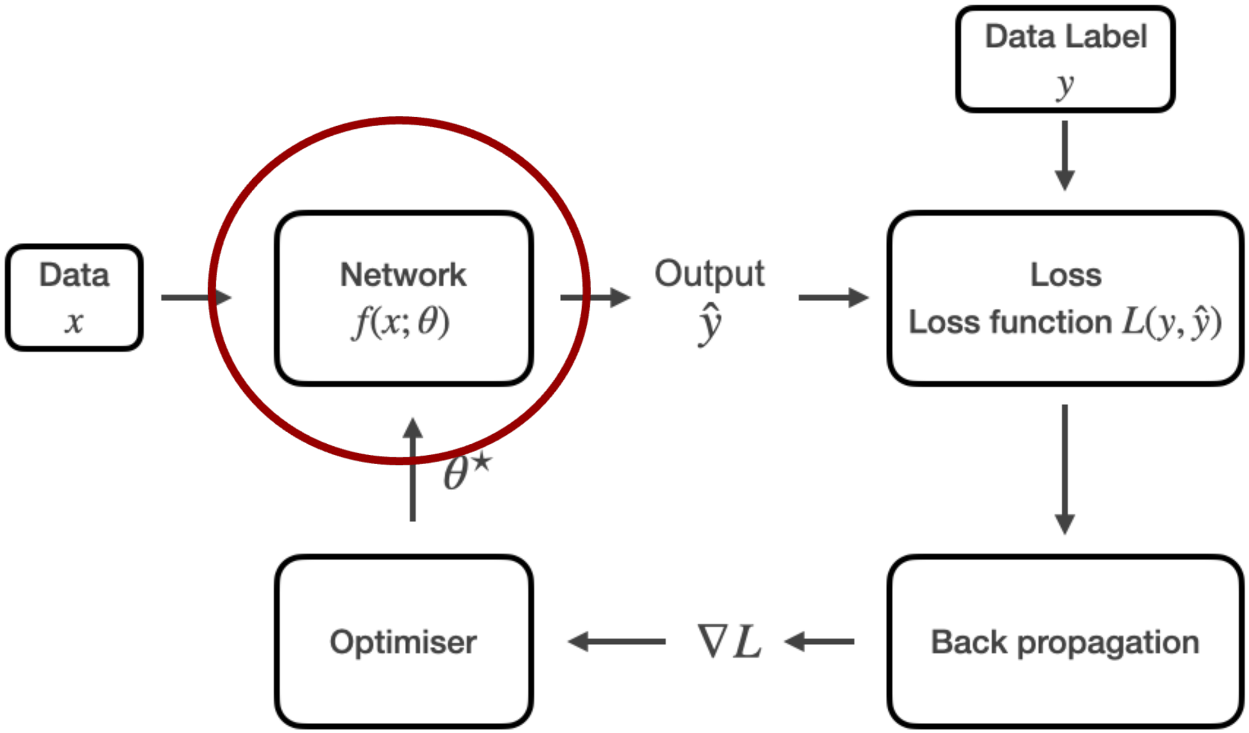
# What is a neural network

- Visual example: curve fitting.



## ▪ A Common Network Structure: Multi-layer Perceptron (MLP)

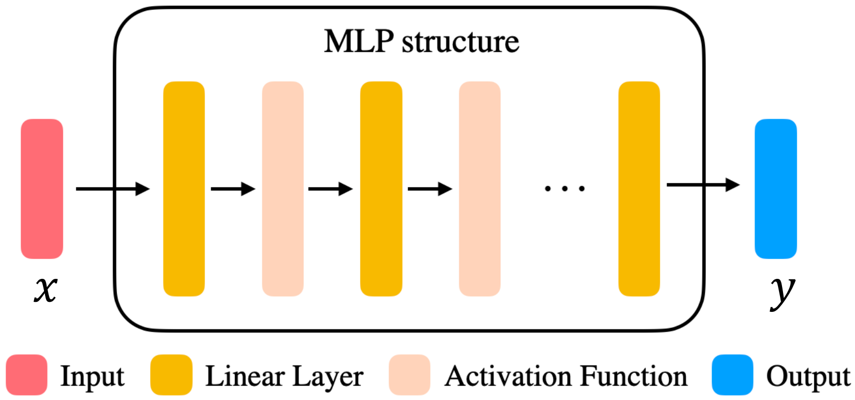
- There are lots of neural network structures, today we introduce one of the most used structure: the multi-layer perceptron.



## A Common Network Structure: Multi-layer Perceptron (MLP)

- Basic structure of MLP

  Input – Linear layer – Activation function – Linear layer …. Linear layer- Output

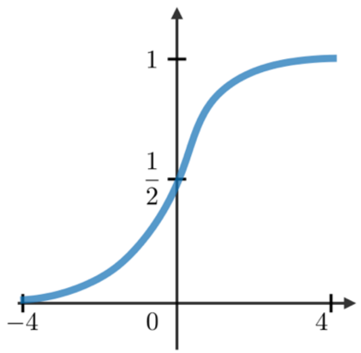
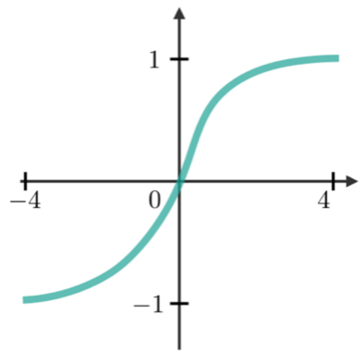
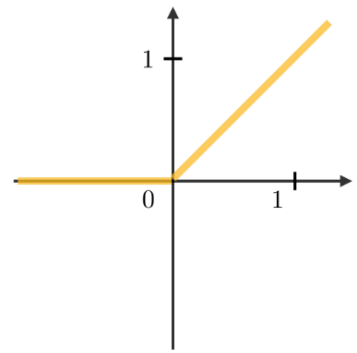
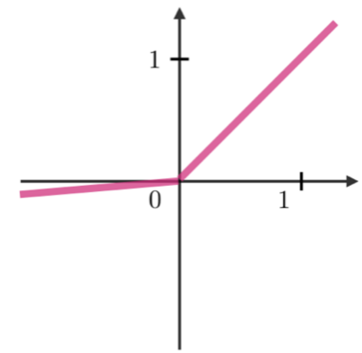


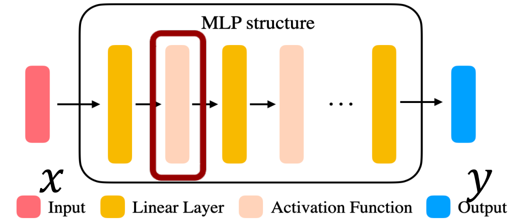

```
def pseudocode_for_mlp(x):
    z = linear_layer_1(x)
    z = activation_1(z)
    z = linear_layer_2(z)
    ...
    z = activation_n(z)
    out = linear_layer_n_plus_1(z)
    return out
```

- Linear layer / full connected layer
  - $y = Wx + b$
- Activation functions $g(\cdot)$: introduce non-linearity.
- Why use activation functions? (e.g. ReLu $g(x) = \max(0, x)$ )
- Simplest MLP:
  - $y = W_2 g_1(W_1 x + b_1) + b_2$

# A Common Network Structure: Multi-layer Perceptron (MLP)

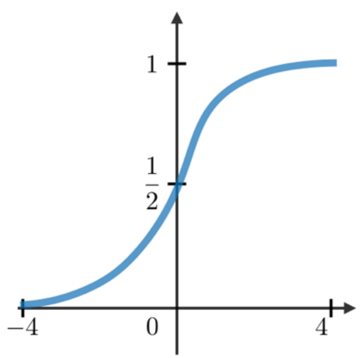
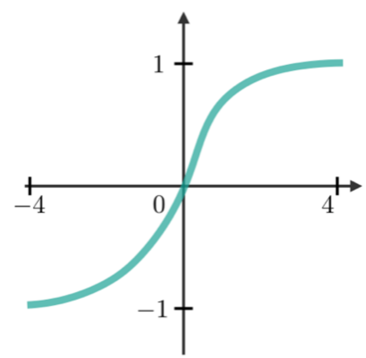
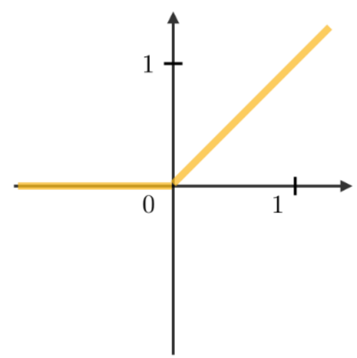
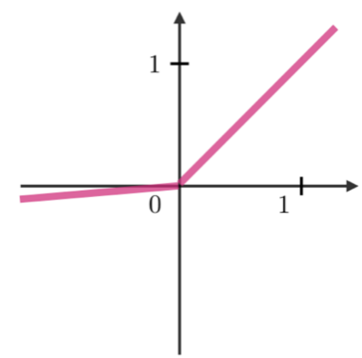
- Commonly Activation functions

| Sigmoid | Tanh | ReLU | Leaky ReLU |
|---|---|---|---|
| $g(z) = \frac{1}{1+e^{-z}}$ | $g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ | $g(z) = \max(0, z)$ | $g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$ |
|  |  |  |  |



## ▪ A Common Network Structure: Multi-layer Perceptron (MLP)

- Commonly Activation functions

| Sigmoid | Tanh | ReLU | Leaky ReLU |
|---|---|---|---|
| $g(z) = \frac{1}{1+e^{-z}}$ | $g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ | $g(z) = \max(0, z)$ | $g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$ |
|  |  |  |  |

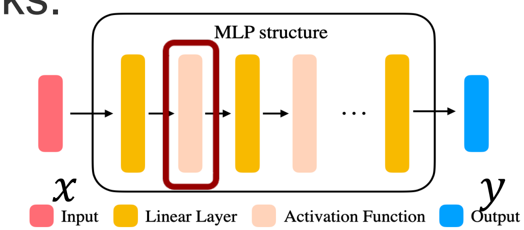








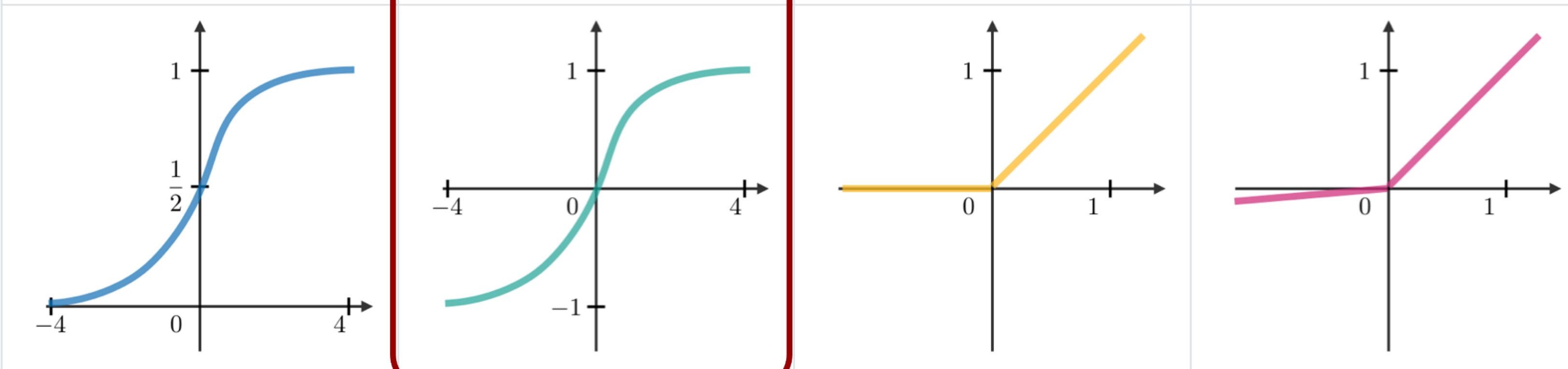
- Outputs values in the range (0,1).
- Useful for probabilistic interpretation in binary classification.
- Can cause vanishing gradient problem in deep networks.



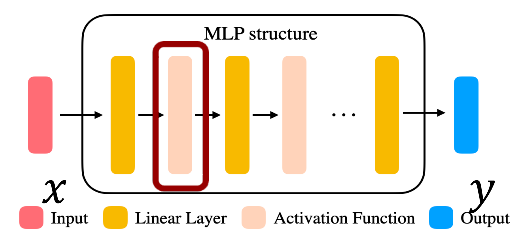
## ▪ A Common Network Structure: Multi-layer Perceptron (MLP)

- Commonly Activation functions

| Sigmoid | Tanh | ReLU | Leaky ReLU |
|---|---|---|---|
| $g(z)=\dfrac{1}{1+e^{-z}}$ | $g(z)=\dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ | $g(z)=\max(0,z)$ | $g(z)=\max(\epsilon z,z)$ with $\epsilon \ll 1$ |



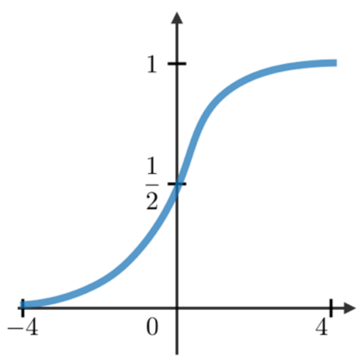

  - Outputs values in the range (-1,1).
  - Zero-centered, more balanced mapping compared to Sigmoid
  - Still can cause vanishing gradient.

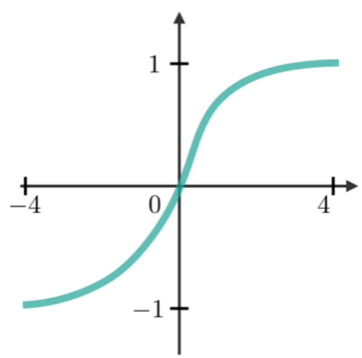


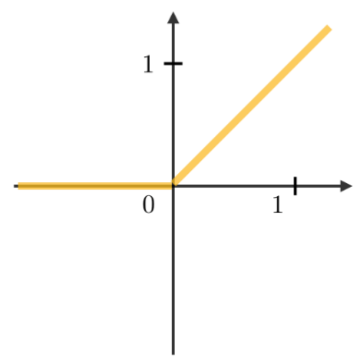
## ▪ A Common Network Structure: Multi-layer Perceptron (MLP)

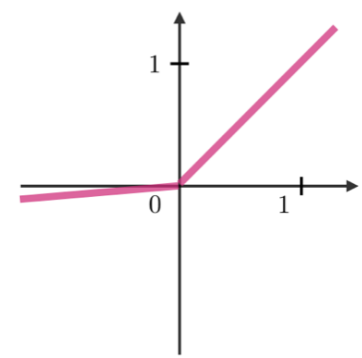
- Commonly Activation functions

| Sigmoid | Tanh | ReLU | Leaky ReLU |
|---|---|---|---|
| $g(z) = \frac{1}{1+e^{-z}}$ | $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ | $g(z) = \max(0, z)$ | $g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$ |

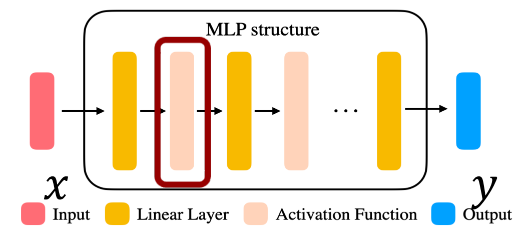


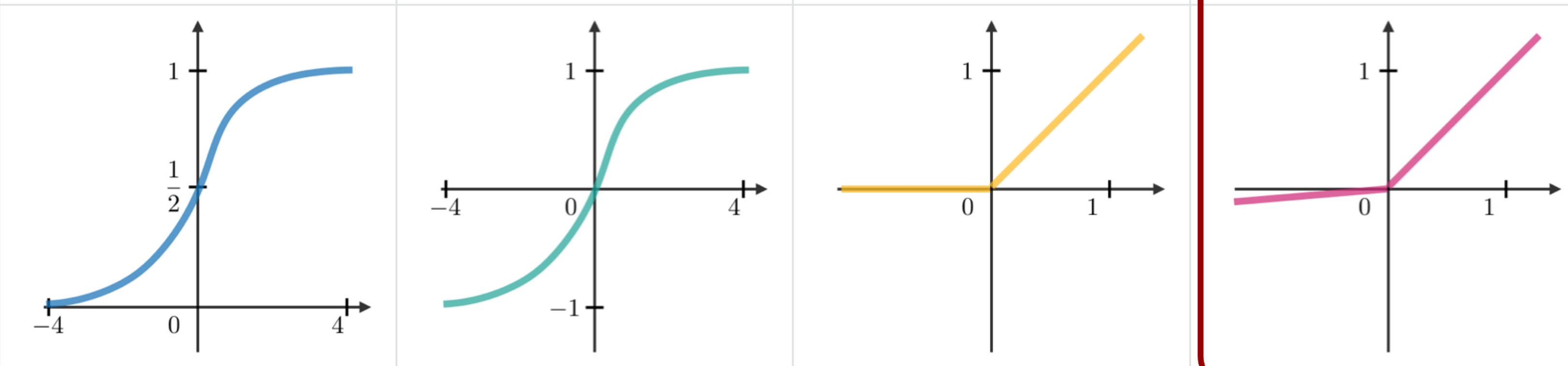

- Computation efficient
- Gradient is 0 when the input smaller than 0.
  - Sparse propagation
  - Dead neurons
- Gradient discontinuous when $z = 0$.



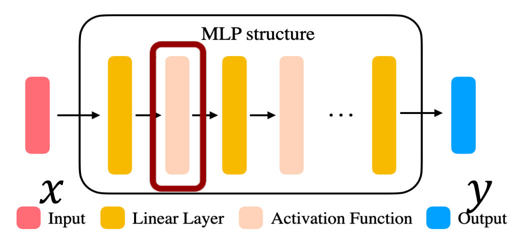
# A Common Network Structure: Multi-layer Perceptron (MLP)

- Commonly Activation functions

| Sigmoid | Tanh | ReLU | Leaky ReLU |
|---|---|---|---|
| $g(z) = \frac{1}{1+e^{-z}}$ | $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ | $g(z) = \max(0, z)$ | $g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$ |



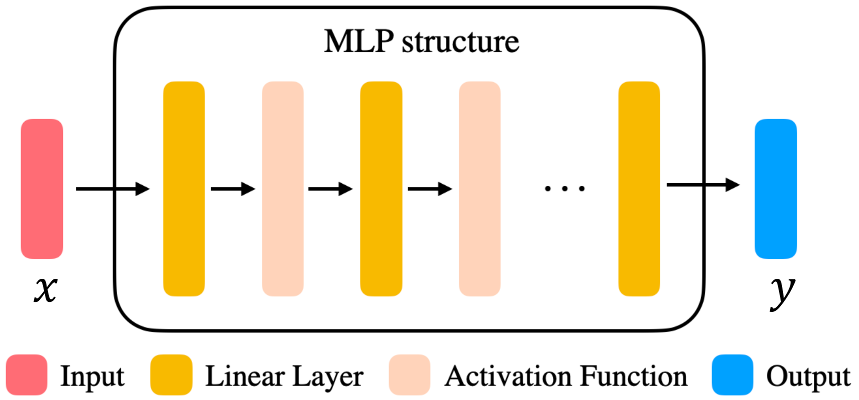

- Computation efficient
- Gradient is not continuous in at $z = 0$



## A Common Network Structure: Multi-layer Perceptron (MLP)

- Basic structure of MLP

  Input – Linear layer – Activation function – Linear layer …. Linear layer- Output




```python
def pseudocode_for_mlp(x):
    z = linear_layer_1(x)
    z = activation_1(z)
    z = linear_layer_2(z)
    ...
    z = activation_n(z)
    out = linear_layer_n_plus_1(z)
    return out
```

  - Simplest MLP:
    - $y = W_2 g_1(W_1 x + b_1) + b_2$

- Practice: simple MLP， suppose we use ReLu() as the activation.

```python
def manual_mlp(x, W1, b1, W2, b2):
    """Calculate the output of simple MLP by ourselves.
    """
    z = torch.matmul(x, W1.transpose(-1,-2)) + b1
    print("The output after first linear layer:", z)
    z = torch.max(torch.tensor(0.0), z)
    print("The output after the activation layer:", z)
    out = torch.matmul(z, W2.transpose(-1,-2)) + b2
    print("The output after first linear layer:", out)
    return out
```

```python
# import necessary packages
import torch
```

## ▪ A Common Network Structure: Multi-layer Perceptron (MLP)

- Practice: write a simple MLP in PyTorch
  - Import necessary packages

```
# import necessary packages
import torch
import torch.nn as nn
```

  - Understand how to create different layers

**Linear layer:**

```
fc1 = nn.Linear(input_size, output_size)
```

*Note that PyTorch use $y = xW^T + b$ instead

*Suppose we have $x \in R^{N\times3}$, and we want $y \in R^{N\times5}$, nn.Linear(3, 5) set the $W \in R^{3\times5}$, $b \in R^{1\times5}$

**Activation function:**

```
activation = nn.ReLU() # nn.Sigmoid(), nn.Tanh(), nn.LeakyReLU()
```

# A Common Network Structure: Multi-layer Perceptron (MLP)

- Practice
  - Write a simple MLP in PyTorch

```python
class SimpleMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.activation = nn.ReLU() # nn.Sigmoid(), nn.Tanh(), nn.LeakyReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        return out
```

```python
torch_mlp = SimpleMLP(input_size=3,hidden_size=4,output_size=2)
```

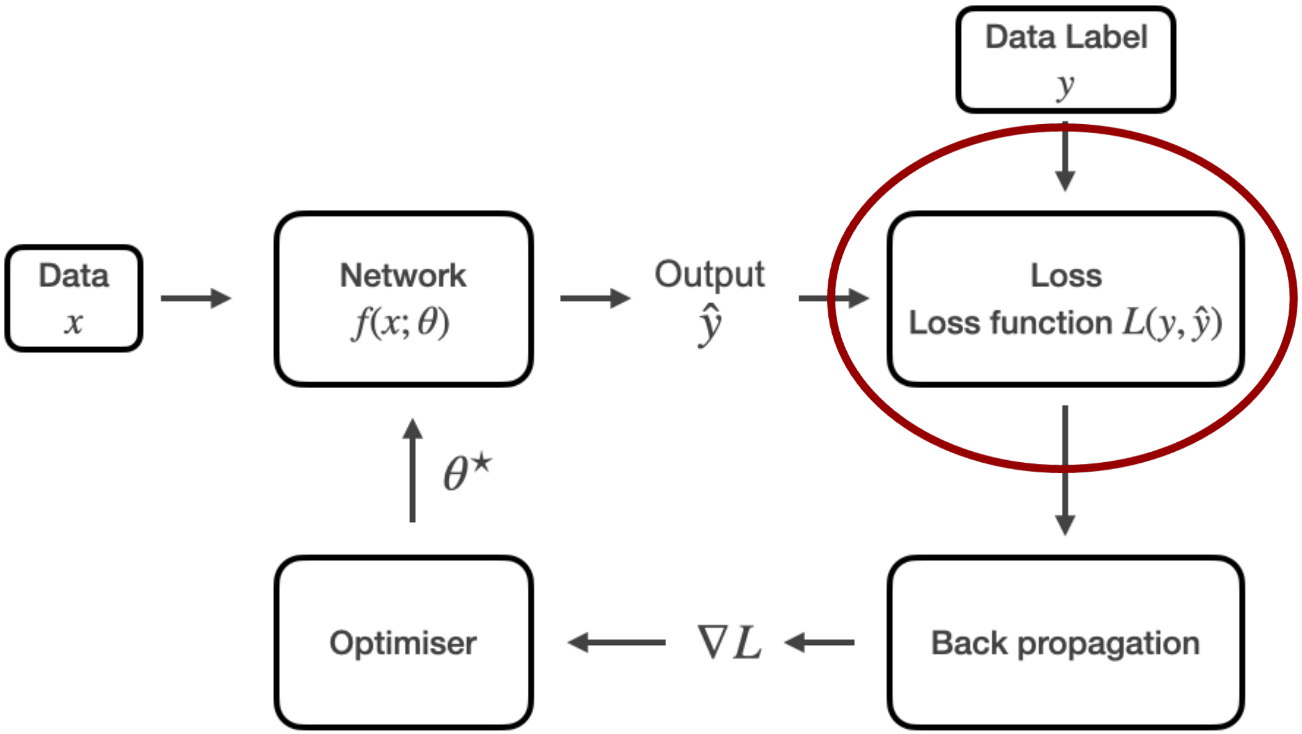
```python
# import necessary packages
import torch
import torch.nn as nn
```

  - Verify the behavior of our manual MLP and the PyTorch MLP (e.g. $x$ is a set of two points $[[-1,1,2],[2,-1,0]]$)

```python
x = torch.tensor([[-1.,1.,2.],[2.,-1.,0.]])
y = torch_mlp(x)
```

## ▪ Loss Functions for Neural Network Training

- Loss functions measure how well a neural network's predictions match the actual target values.
- They guide the optimization process during training, allowing the model to learn by minimizing the error.

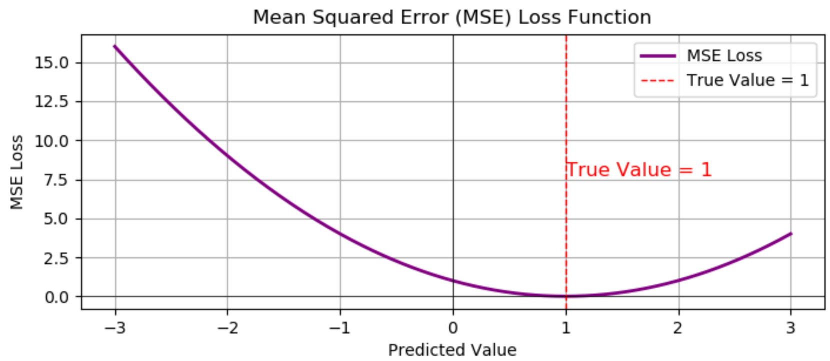


## ▪ Loss Functions for Neural Network Training

- Mean Squared Error (MSE)
  - Measures the average squared difference between predicted and actual values.
  - Formular:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where $y_i$ is the actual value, $\widehat{y_i}$ is the predicted value, and $n$ is the number of samples.




  - Used in regression tasks where the goal is to predict continuous values.
  - Sensitive to outliers.
  - Converge fast when error is large
- Code example

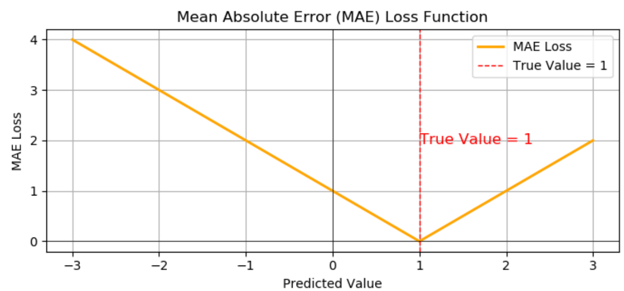
```
import torch.nn as nn
mse_loss = nn.MSELoss()
y_hat = torch.tensor([1.5, 1.0])
y = torch.tensor([1.0, 1.0])
loss = mse_loss(y_hat, y)
print(loss)
```

## Loss Functions for Neural Network Training

- Mean Absolute Error (MAE) / L1 Loss
  - Measures the average absolute difference between predicted and actual values.
  - Formular:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

where $y_i$ is the actual value, $\widehat{y_i}$ is the predicted value, and $n$ is the number of samples.




  - Used in regression tasks where the goal is to predict continuous values.
  - Can lead to less stable convergence
  - Less sensitive to outliers
  - Code example

```
l1_loss = nn.L1Loss()
y_hat = torch.tensor([1.5, 1.0])
y = torch.tensor([1.0, 1.0])
loss = l1_loss(y_hat, y)
```

## ▪ Loss Functions for Neural Network Training

- Kullback-Leibler (KL) Divergence Loss
  - Measures how one probability distribution diverges from a second, expected probability distribution.
  - Formular: $D_{KL}(P||Q) = \sum_i P(i) \log(\frac{P(i)}{Q(i)})$

  where $P$ is the true distribution, $Q$ is the predicted distribution, $i$ represent each elements.
  - Examples

$$P = [0.4, 0.6],\ \mathrm{Q} = [0.3, 0.7]$$

$$D_{KL}(P||Q) = 0.4 \log \frac{0.4}{0.3} + 0.6 \log \frac{0.6}{0.7}$$

```python
import numpy as np

# True distribution (target)
P = np.array([0.4, 0.6])

# Predicted distribution (model output)
Q = np.array([0.3, 0.7])

# Calculate KL Divergence manually
kl_divergence = np.sum(P * np.log(P / Q))

print("KL Divergence (Manual Calculation):", kl_divergence)
```

```
KL Divergence (Manual Calculation): 0.022582421084357485
```

```python
# True distribution (target)
P_torch = torch.tensor([0.4, 0.6]).unsqueeze(0)

# Predicted distribution (model output)
Q_torch = torch.tensor([0.3, 0.7]).unsqueeze(0)

# Calculate KL Divergence in PyTorch
kl_loss = nn.KLDivLoss(reduction='batchmean')
loss = kl_loss(Q_torch.log(), P_torch)

print("KL Divergence (PyTorch with nn.KLDivLoss):", loss.item())
```

✓ 0.0s

```
KL Divergence (PyTorch with nn.KLDivLoss): 0.022582478821277762
```

## ▪ Loss Functions for Neural Network Training

- Kullback-Leibler (KL) Divergence Loss
  - Measures how one probability distribution diverges from a second, expected probability distribution.
  - Formular: $D_{KL}(P||Q) = \sum_i P(i) \log(\frac{P(i)}{Q(i)})$

  where $P$ is the true distribution, $Q$ is the predicted distribution, $i$ represent each elements
  - $D_{KL}(P||Q) \neq D_{KL}(\mathrm{Q}||\mathrm{P})$
  - Pros
    - Useful for variational autoencoders and reinforcement learning, minimizing KL Divergence helps in better fitting the model to the true data distribution.
    - Can be extended to continuous distributions and is often used in cases where distributions are not discrete.
  - Cons
    - Not sensitive to large $Q(i)$ when $P(i)$ close to zero.

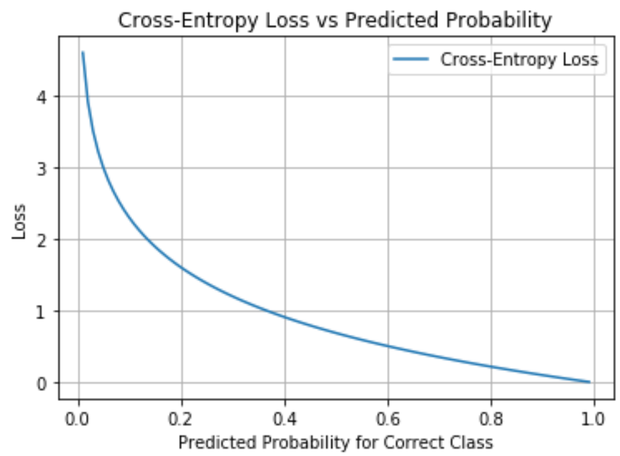
## Loss Functions for Neural Network Training

- Cross-Entropy Loss (Categorical Loss)
  - Cross-entropy loss measures the difference between two probability distributions – the predicted probability from the model and the true distribution (one-hot encoded labels)
  - Formular:

$$\text{CEL} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

where $C$ is the number of classes, $y_i$ is the true label (1 if the class is correct, otherwise 0), $\hat{y}_i$ is the predicted probability for the category.

  - Loss curve



## Loss Functions for Neural Network Training

- Practice of Cross-Entropy Loss $\text{CEL} = -\sum_{i=1}^{C} y_i \log\left(\widehat{y}_i\right)$
  - Suppose we have 3 categories, and 2 samples.
  - Labels: $[0,1]$, prediction raw output: $[[2.0, 1.0, 0.1], [0.1, 2.0, 1.0]]$,
  - Manual calculation

    Step 1: map the raw output to probability distribution

$$\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

```python
def softmax(logits):
    exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)
```

    Step 2: Calculate cross-entropy loss

```python
def cross_entropy_loss(probs, true_labels):
    N = probs.shape[0]
    log_probs = -np.log(probs[range(N), true_labels])
    return np.mean(log_probs)
```

```python
raw_prediction = np.array([[2.0, 1.0, 0.1], [0.1, 2.0, 1.0]])
# True labels (index of the correct class for each example)
true_labels = np.array([0, 1])
# Get predicted probabilities using softmax
predicted_probs = softmax(raw_prediction)
# Calculate the manual cross-entropy loss
loss = cross_entropy_loss(predicted_probs, true_labels)
```

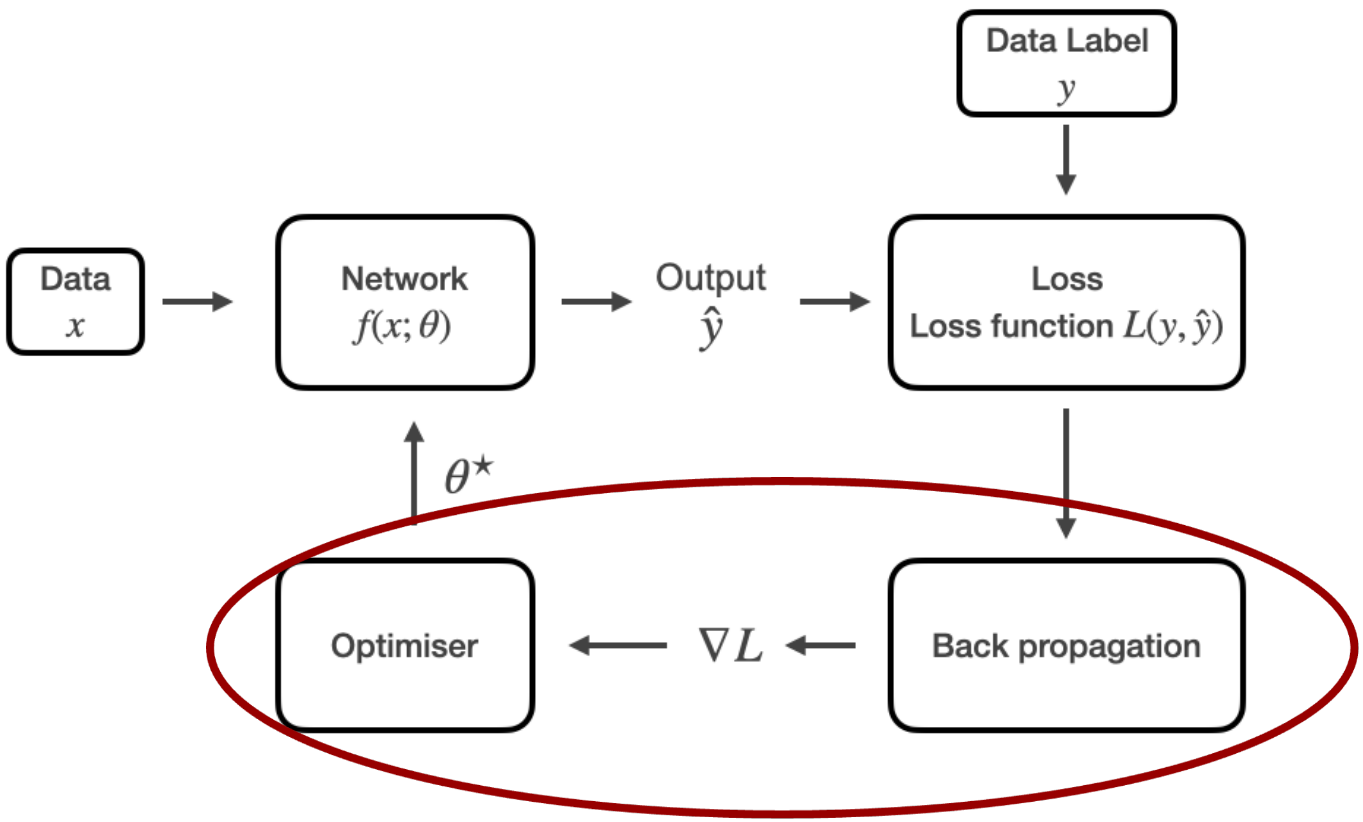
  - Calculate CEL in PyTorch

```python
loss_fn = nn.CrossEntropyLoss()
# Suppose we have 3 classes and 2 examples in the batch
predictions = torch.tensor([[2.0, 1.0, 0.1], [0.1, 2.0, 1.0]])
labels = torch.tensor([0, 1])  # true labels

loss = loss_fn(predictions, labels)
```

## Optimizer in Neural Network Training

- The optimizer adjusts the network parameters by taking a step in the direction that minimizes the loss, using the gradients and a specified learning rate.

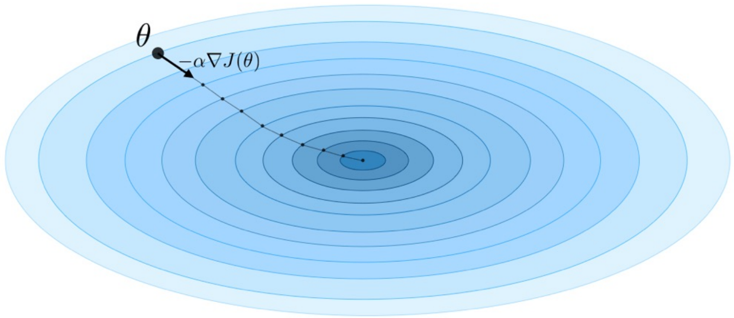


## Optimizer in Neural Network Training

- Denoting the learning rate as $\alpha$, this process follows the general formula:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta)$$

where $\theta$ represents the network parameter, and $\nabla_\theta J(\theta)$ is the gradient of the loss function.




- Learning rate is a key hyperparameter that controls how big a step the optimizer takes in the direction of the gradient.
  - Too high: may cause the model to “overshoot” the optimal parameters, and the model might fail to converge or oscillate.
  - Too low: may take a long time to converge, or even get stuck in a local minimum.

## ▪ Optimizer in Neural Network Training

- Stochastic Gradient Descent (SGD)
  - Gradient Descent uses the entire dataset to compute gradients for updating the parameters.
  - SGD updates the parameters using one mini-batch of data at a time, making it faster and more efficient for large datasets.
  - Formula:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta; x_i, y_i)$$

    where $(x_i,\ y_i)$ is a random minibatch from the dataset.
  - Advantages
    - Faster updates
    - Can help escape local minima by introducing noise
  - Disadvantages:
    - May oscillate or struggle to converge.
  - Code

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

  - Other optimizers: Adam, RMSProp, etc.

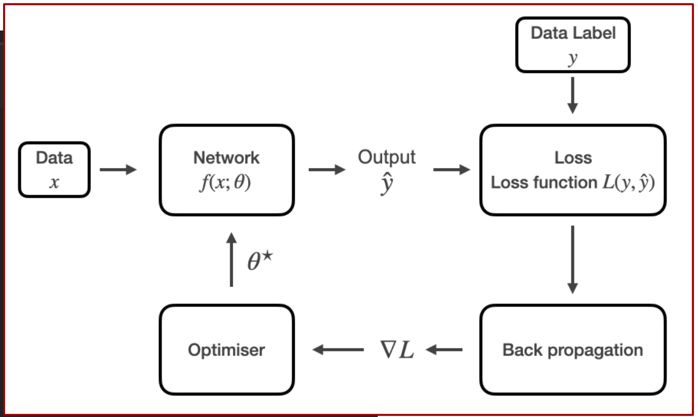
## Overall Structure: Code Flow

- Code example for a neural network training.

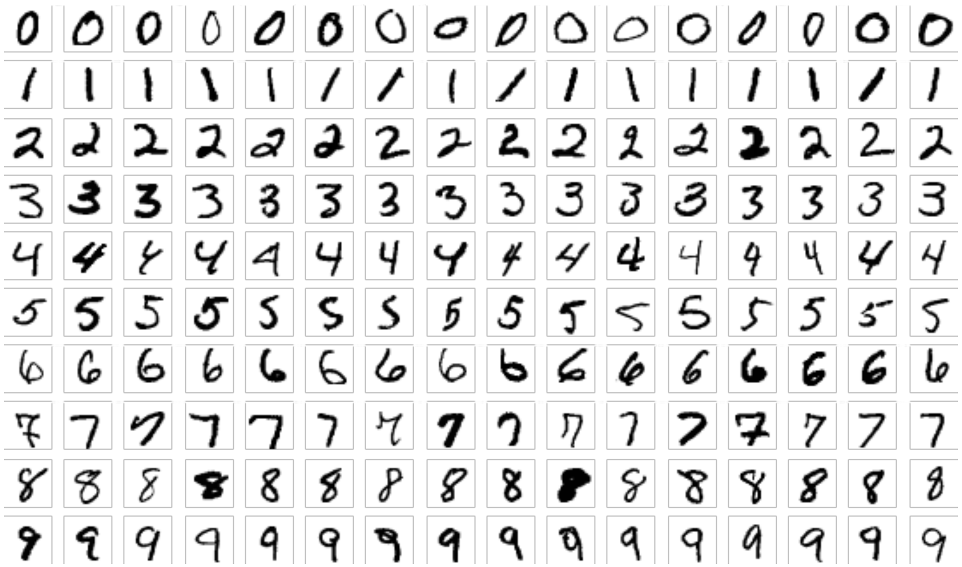
```python
import torch
import torch.optim as optim

# Model parameters
input_size = 10
hidden_size = 5
output_size = 1
model = SimpleMLP(input_size, hidden_size, output_size)

# Loss and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    for inputs, labels in dataloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```



- **Construct a Machine Learning Task in PyTorch**
  - Practice: classification of handwritten digits



    - Input: images, output: category
    - Dataset: MNIST is a dataset of 70,000 handwritten digits, It consists of 60,000 training images and 10,000 test images, each 28x28 pixels in size, with digits from 0 to 9.
    - The goal is to classify each image into one of 10 categories (0 to 9).

## ▪ Construct a Machine Learning Task in PyTorch

- Practice: classification of handwritten digits
  - Loss: cross-entropy loss
  - Model: MLP
  - Input dimension: 28*28, output dimension: 10
  - Import necessary packages

```
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
```

  - Data preparation

```
# Load the MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = datasets.MNIST(root='./data', train=False, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

## Construct a Machine Learning Task in PyTorch

- Practice
  - Construct a neural network model, loss, and optimizer

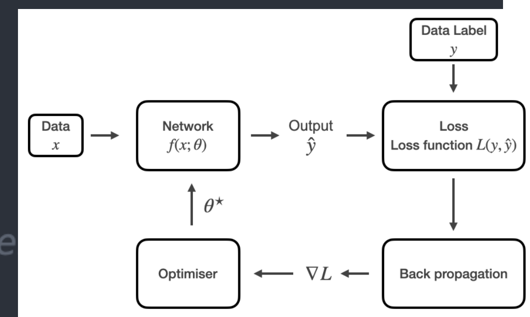
```python
# Define the MLP model
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)  # First hidden layer
        self.fc2 = nn.Linear(128, 64)     # Second hidden layer
        self.fc3 = nn.Linear(64, 10)      # Output layer for 10 classes

    def forward(self, x):
        x = x.view(-1, 28*28)  # Flatten the input
        x = torch.relu(self.fc1(x))  # Activation function
        x = torch.relu(self.fc2(x))  # Activation function
        x = self.fc3(x)  # Output layer
        return x
# Initialize the model, loss function, and optimizer
model = SimpleMLP()
criterion = nn.CrossEntropyLoss()  # Loss function
optimizer = optim.SGD(model.parameters(), lr=0.01)  # Optimizer
```

## Construct a Machine Learning Task in PyTorch

- Practice
  - Write the training loop

```python
# Training the model
def train(model, train_loader, criterion, optimizer, epochs=1):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()  # Zero the gradie
            outputs = model(images)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights
            running_loss += loss.item()
        print(f'Epoch {epoch+1}, Loss: {running_loss/len(train_loader)}')
```




```python
train(model, train_loader, criterion, optimizer, epochs=5)
```

- **Construct a Machine Learning Task in PyTorch**
  - Practice
    - Write the test function

```
# Test the model
def test(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f'Accuracy: {100 * correct / total}%')
```

- **Construct a Machine Learning Task in PyTorch**
  - Homework
    - Design your own MLP for handwritten digits classification to achieve at least 95% accuracy.
    - Dataset: MNIST
    - You could change hidden layer numbers, hidden unit size, activation functions, optimizers (web browser is your friend)
    - Submit in Jupyter Notebook with necessary explanation and results. REMEMBER TO SET THE RANDOM SEED so that the results can be reproduced!
    - Students achieved the top 3 accuracy will get bonus.
    - Due: $27^{th}$ September