Lab 01

Introduction to Deep Learning
(11-785/ 685/ 485)

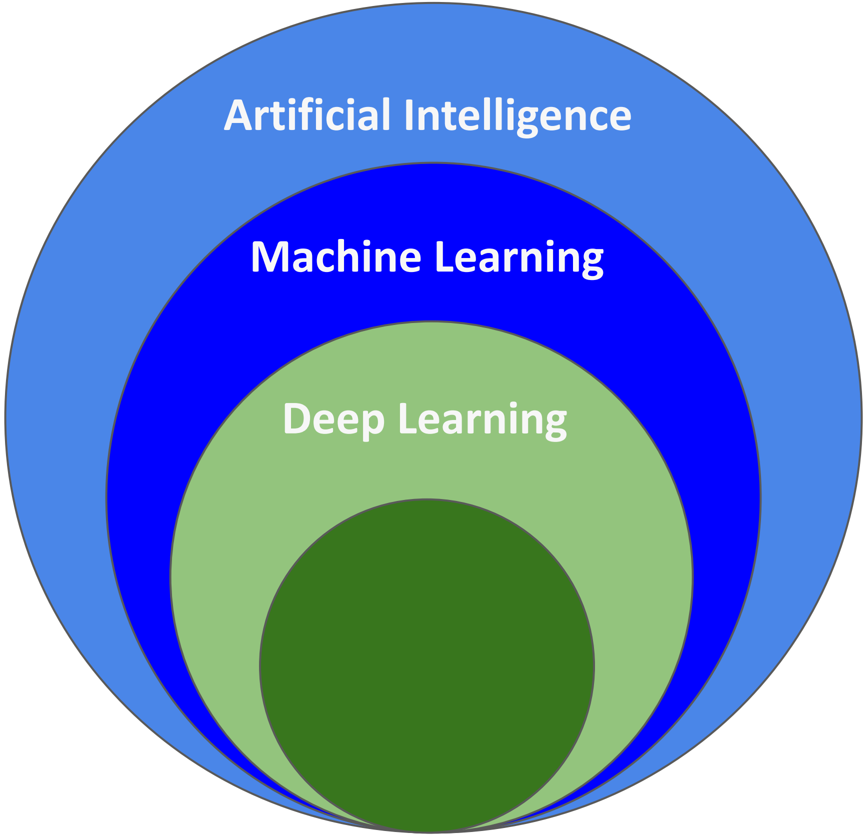
Aug 30 2024

Announcements

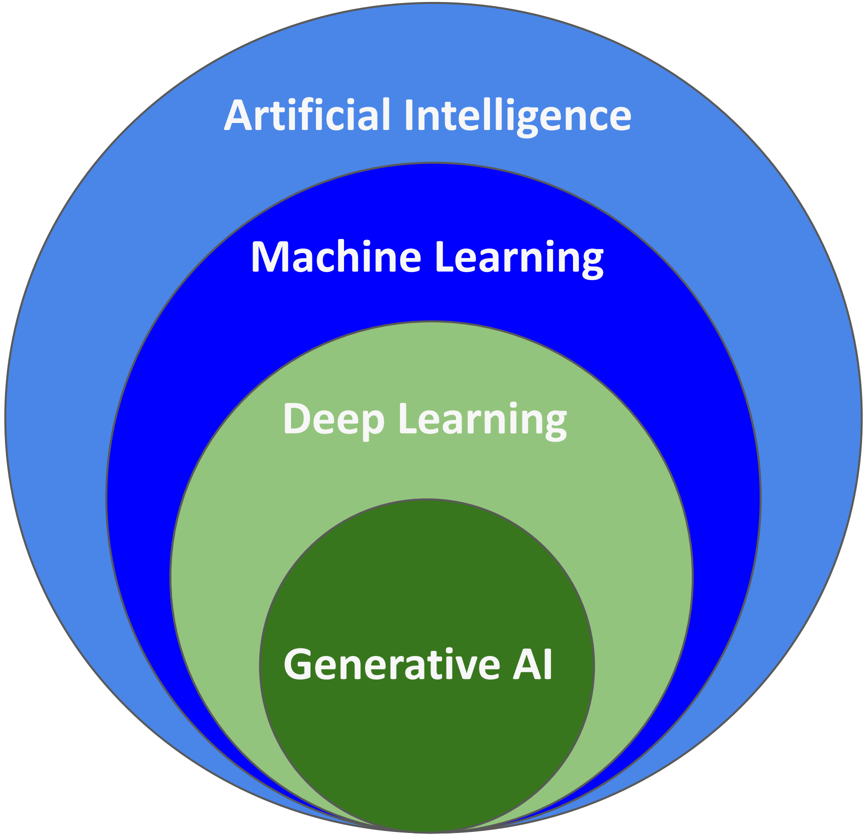
- HW1 released
- Early-bird submission due Sep 06 11:59 PM
- Don’t forget HW1 Quiz on Canvas; also due Sep 06 11:59 PM
- Quiz 1 will go out today at 11:59 PM; due on Sunday 11:59 PM
- Bootcamp tomorrow at 2:00 PM

What is Deep Learning?

The Classes



The Classes


Artificial Intelligence
Machine Learning
Deep Learning
Generative AI

Machine Learning vs Deep Learning

- **Machine Learning** is subset of AI that employs algorithms to analyze and learn from data, enabling systems to make predictions or decisions without being explicitly programmed for specific tasks.
- Often relies on structured data and requires feature engineering.
    - **Common Algorithms:** Linear Regression, Decision Trees, Support Vector Machines (SVM), k-Nearest Neighbors (k-NN).
- **Deep Learning** is a subset of **machine learning** that uses neural networks with multiple layers (deep neural networks) to model complex patterns in large datasets.
    - **Common Architectures:** Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Transformers.

Machine Learning

Deep Learning

Key Differences

Aspect	**Machine Learning**	**Deep Learning**
Data Type	Primarily structured data (e.g., spreadsheets, databases)	Both structured and unstructured data (e.g., images, audio, text)
Feature Engineering	Requires manual feature engineering based on domain knowledge	Features are automatically extracted by Neural Networks
Model complexity	Models are simpler and more interpretable (decision Trees)	Models are complex with many layers, often seen as “black boxes”
Computational Power	Requires less computation power	Requires significant computation power. Usually requires GPUs, TPUs

Generative AI

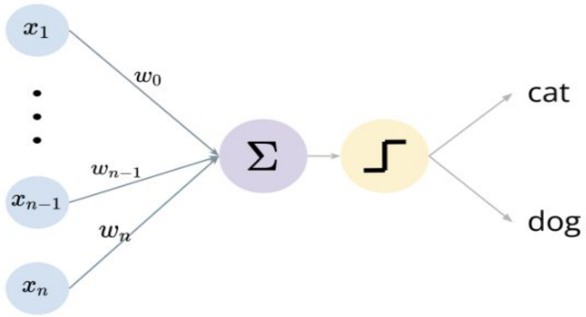
- OpenAI ChatGPT
- Anthropic Claude
- OpenAI DALL-E
- Meta LLama
- Apple Intelligence

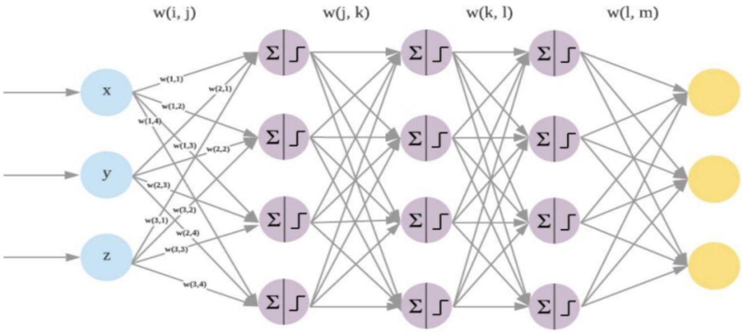
Your First MLP

In PyTorch

MLP Architecture

```python
class Network(torch.nn.Module):

    def __init__(self, input_size, output_size):

        super(Network, self).__init__()

        self.layers = torch.nn.Sequential(
            torch.nn.Linear(input_size, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, output_size)
        )

    def forward(self, x):
        out = self.layers(x)
        return out
```





Training a Deep Learning Model

1. Forward Propagation
2. Loss Calculation
3. Backpropagation
4. Weight Update
5. Iterate over multiple epochs

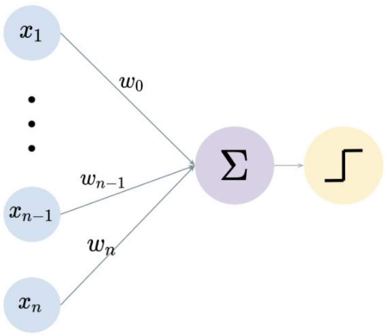
Forward Propagation

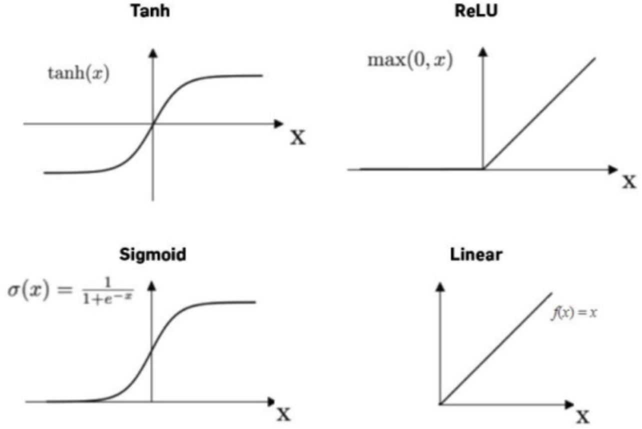
Weighted sum

$$\mathbf{z}_i = \mathbf{W}_i\mathbf{x} + \mathbf{b}_i$$

Activation function

$$\mathbf{a}_i = f_i(\mathbf{z}_i)$$






```
model = Network(input_size, output_size)
predictions = model(x) # equivalent to model.forward(x)
```

Loss Functions

Classification vs Regression

- Regression: MSE, MAE, RMSE
- Classification: Cross Entropy, KL

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

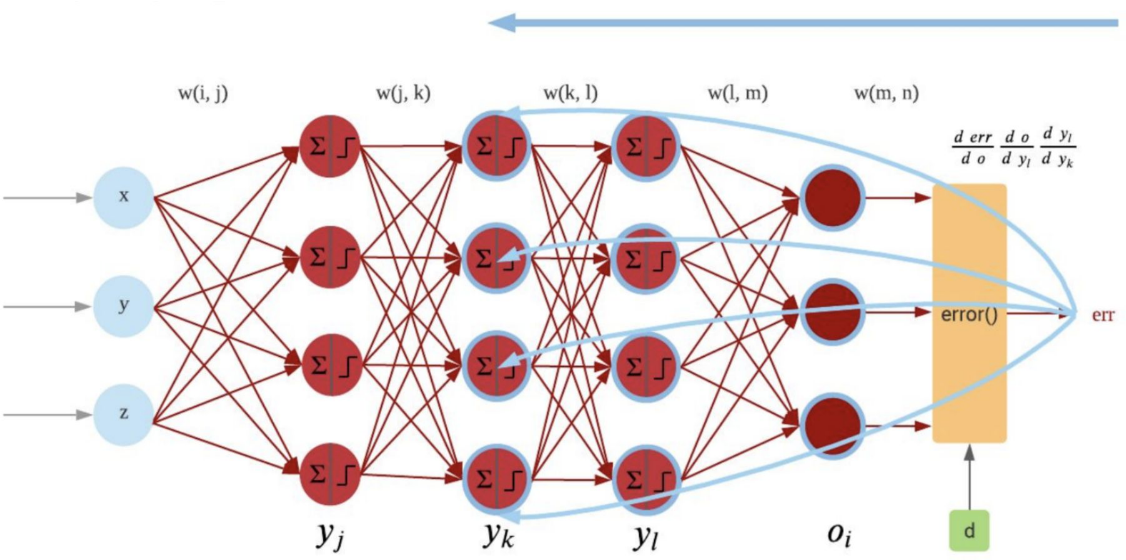
```
criterion = torch.nn.MSELoss()
loss = criterion(predictions, labels)
```

$$\mathrm{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Backpropagation

Computes gradient of loss function wrt network parameters

`loss.backward()`



Optimization & Parameter Update

Gradient Descent

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J(\theta)$$

Adam

$$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

```
optimizer = torch.optim.SGD(model.parameters(), lr= initial_lr)
optimizer.zero_grad()
optimizer.step()
```

Access gradients with

```
for param in model.parameters():
   print(param.grad)
```

Train vs Eval mode

Models may behave differently in training vs evaluation mode

- Dropout
- Batch normalization
- Data augmentation

```
model.train()
```

```
model.eval()
```

Device: Cuda and CPU

Model and data must be on the same device

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = model.to(device)

for i, (input, labels) in enumerate(dataloader):
    input     = input.to(device)
    labels    = labels.to(device)

    predictions = model(input)
```

Training One Epoch

```
def train_one_epoch(model, dataloader, optimizer,
criterion):

    model.train()
    train_loss = 0

    for i, (input, labels) in enumerate(dataloader):

        optimizer.zero_grad()

        input       = input.to(device)
        labels      = labels.to(device)

        predictions   = model(input)

        loss      = criterion(predictions, labels)

        loss.backward()

        optimizer.step()

        train_loss    += loss


    train_loss    /= len(dataloader)

    return train_loss
```

Evaluation - Inference Mode

Inference is more efficient than training

```
with torch.inference_mode():
        predictions  = model(input)
        loss     = criterion(predictions, labels)
```

Also remember `model.eval()`

Thank You.