All Deep Learning & Reinforcement Learning Algorithms With Code & Use case

Type	@datasciencebrain

Deep Learning Algorithms (DL)

1. Convolutional Neural Networks (CNNs)

Overview: Used for processing image and video data. CNNs extract spatial hierarchies through convolutional layers.

Use Cases:

- Image classification (e.g., MNIST, CIFAR-10)
- Object detection (e.g., autonomous vehicles)
- Medical imaging (e.g., tumor detection)

Code (PyTorch):

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

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3)
```

```
        self.fc1 = nn.Linear(32 * 5 * 5, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Advantages:

- Efficient spatial feature extraction
- Fewer parameters than fully connected networks

Disadvantages:

- Computationally intensive
- Sensitive to transformations

---

2. Recurrent Neural Networks (RNNs)

Overview: Designed for sequential data by maintaining internal memory.

Use Cases:

- Time-series forecasting
- Text generation
- Speech recognition

Code (PyTorch):

```
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])
        return out
```

Advantages:

- Handles sequential data effectively
- Parameter sharing across time steps

Disadvantages:

- Vanishing/exploding gradients
- Difficult to model long-term dependencies

---

3. Long Short-Term Memory Networks (LSTMs)

Overview: A type of RNN that handles long-term dependencies using memory cells and gates.

Use Cases:

- Language modeling
- Machine translation
- Speech recognition

Code (PyTorch):

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.lstm(x)
```

```
        out = self.fc(out[:, -1, :])
        return out
```

Advantages:

- Handles long-range dependencies
- Reduces vanishing gradient issue

Disadvantages:

- Complex architecture
- Slower training

---

4. Gated Recurrent Units (GRUs)

Overview: A variant of LSTM with fewer parameters, combining forget and input gates.

Use Cases:

- Real-time speech recognition
- Text classification

Code (PyTorch):

```
class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRUModel, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.gru(x)
        out = self.fc(out[:, -1, :])
        return out
```

Advantages:

- Faster and simpler than LSTMs

- Comparable performance

Disadvantages:

- Less expressive than LSTMs for complex tasks

---

5. Transformer Models

Overview: Attention-based architecture, processes sequences in parallel unlike RNNs.

Use Cases:

- Language translation (e.g., Google Translate)
- Text summarization
- Chatbots (e.g., ChatGPT)

Code (HuggingFace Transformers):

```
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')
inputs = tokenizer("Deep learning with Transformers", return_tensors="pt")
outputs = model(**inputs)
```

Advantages:

- Captures long-term dependencies
- Parallel processing

Disadvantages:

- Requires large datasets and computational resources

---

6. Autoencoders

Overview: Unsupervised learning models that learn efficient data representations by encoding and decoding input data.

Use Cases:

- Dimensionality reduction
- Anomaly detection
- Image denoising

Code (PyTorch):

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, 784),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Advantages:

- Learns compact representations
- Useful for pretraining

Disadvantages:

- Reconstruction loss may not reflect meaningful features
- Poor performance on complex datasets

---

7. Generative Adversarial Networks (GANs)

Overview

Generative Adversarial Networks (GANs) consist of two neural networks — a generator and a discriminator — that compete with each other. The generator creates fake data, while the discriminator tries to distinguish between real and fake data. Training continues until the generator produces data indistinguishable from real samples.

Real-life Use Case

- Image synthesis: Creating realistic images of people (e.g., StyleGAN)
- Data augmentation: Generating additional training data in medical imaging
- Super-resolution: Enhancing the resolution of low-quality images

Working Code (PyTorch)

```
import torch
import torch.nn as nn

# Generator
class Generator(nn.Module):
    def __init__(self, noise_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(noise_dim, 128),
            nn.ReLU(),
            nn.Linear(128, output_dim),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z)

# Discriminator
```

```
class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

# Training setup
z_dim = 100
generator = Generator(z_dim, 784)
discriminator = Discriminator(784)
criterion = nn.BCELoss()
g_optimizer = torch.optim.Adam(generator.parameters(), lr=0.0002)
d_optimizer = torch.optim.Adam(discriminator.parameters(), lr=0.0002)

# Fake and real data labels
real_labels = torch.ones(64, 1)
fake_labels = torch.zeros(64, 1)

# One step of training loop (simplified)
z = torch.randn(64, z_dim)
fake_images = generator(z)
outputs = discriminator(fake_images.detach())
d_loss = criterion(outputs, fake_labels)

# Update discriminator
d_optimizer.zero_grad()
d_loss.backward()
d_optimizer.step()
```

Advantages

- Capable of generating highly realistic data
- Useful in data augmentation for improving model robustness
- Flexible and can be applied to images, text, and audio

Disadvantages

- Training is unstable and often suffers from mode collapse
- Requires careful tuning of hyperparameters and architecture
- Difficult to evaluate the performance objectively

---

8. Deep Reinforcement Learning (DRL)

Overview

Deep Reinforcement Learning combines neural networks with reinforcement learning principles. It enables agents to learn optimal behaviors through interactions with an environment by maximizing cumulative rewards.

Real-life Use Case

- Game playing (e.g., AlphaGo, DQN in Atari games)
- Robotics (e.g., robotic arm manipulation)
- Self-driving cars (decision-making and path planning)

Working Code (using OpenAI Gym and PyTorch - simplified DQN)

```
import gym
import torch
import torch.nn as nn
import torch.optim as optim
import random
import numpy as np

class DQN(nn.Module):
```

```
    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )

    def forward(self, x):
        return self.net(x)

env = gym.make("CartPole-v1")
model = DQN(env.observation_space.shape[0], env.action_space.n)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.MSELoss()

# Sample training step
state = torch.FloatTensor(env.reset())
q_values = model(state)
action = torch.argmax(q_values).item()
```

Advantages

- Learns directly from high-dimensional sensory inputs
- Capable of mastering complex tasks without manual feature engineering
- Adaptable to dynamic environments

Disadvantages

- Sample inefficient: requires many interactions with the environment
- Sensitive to hyperparameters and reward shaping
- Hard to train and debug due to stochastic nature

---

9. Multilayer Perceptrons (MLPs)

Overview

Multilayer Perceptrons are the simplest form of feedforward neural networks with fully connected layers. They are used for classification and regression tasks where spatial or temporal patterns are not crucial.

Real-life Use Case

- Tabular data classification (e.g., credit scoring, fraud detection)
- Predictive maintenance
- Regression models for sales or demand forecasting

Working Code (PyTorch)

```
class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(MLP, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Linear(64, output_dim)
        )

    def forward(self, x):
        return self.model(x)

x = torch.randn(32, 10)
model = MLP(10, 2)
output = model(x)
```

Advantages

- Simple to implement and fast to train
- Works well on structured data
- Flexible for regression and classification

Disadvantages

- Poor performance on unstructured data (e.g., images, audio)
- Not suitable for capturing spatial/temporal dependencies
- May overfit on small datasets without regularization

---

10. Deep Belief Networks (DBNs)

Overview

Deep Belief Networks are generative models consisting of multiple layers of Restricted Boltzmann Machines (RBMs). They are trained greedily layer-by-layer in an unsupervised manner before fine-tuning with supervised learning.

Real-life Use Case

- Pretraining deep networks before the advent of ReLUs and batch normalization
- Dimensionality reduction
- Handwritten digit recognition (MNIST)

Working Code (Simplified using `BernoulliRBM` from scikit-learn)

```
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target)

rbm = BernoulliRBM(n_components=64)
logistic = LogisticRegression(max_iter=1000)
```

```
classifier = Pipeline(steps=[('rbm', rbm), ('logistic', logistic)])
classifier.fit(X_train, y_train)
```

Advantages

- Capable of unsupervised pretraining
- Learns hierarchical representations
- Suitable for binary input data

Disadvantages

- Largely replaced by more efficient methods (e.g., autoencoders, deep CNNs)
- Hard to train and scale to large datasets
- Lack of GPU-based implementations

---

11. Capsule Networks (CapsNets)

Overview

Capsule Networks aim to address the limitations of CNNs by using groups of neurons (capsules) to preserve spatial hierarchies. They use dynamic routing algorithms to route information between layers.

Real-life Use Case

- Image classification with pose awareness
- Medical imaging for detecting rotated or scaled anomalies
- Digit recognition on rotated MNIST

Working Code (Simplified, using `capsulelayers` library – concept only)

```
# Conceptual example only — actual implementation is complex and needs cu
stom layers
```

```
# See: https://github.com/XifengGuo/CapsNet-Pytorch for full implementation
```

Advantages

- Better at modeling spatial hierarchies
- More robust to affine transformations like rotation and scaling
- Requires fewer training examples compared to CNNs

Disadvantages

- Computationally expensive
- Difficult to train and tune
- Limited support in mainstream frameworks

Reinforcement Learning Algorithms

1. Q-Learning

Overview: Q-Learning is a value-based off-policy reinforcement learning algorithm. It aims to learn the optimal action-selection policy using the Bellman equation by updating Q-values for each state-action pair.

Use Cases:

- Path planning in robotics
- Game AI (e.g., simple board games)
- Autonomous navigation in grid environments

Code (Simple Q-Learning in Grid World):

```
import numpy as np

q_table = np.zeros((state_size, action_size))
learning_rate = 0.1
discount_factor = 0.95
epsilon = 0.1
```

```
for episode in range(episodes):
    state = env.reset()
    done = False
    while not done:
        if np.random.rand() < epsilon:
            action = np.random.choice(action_size)
        else:
            action = np.argmax(q_table[state])

        next_state, reward, done, _ = env.step(action)
        best_next = np.max(q_table[next_state])
        q_table[state, action] += learning_rate * (reward + discount_factor * best
_next - q_table[state, action])
        state = next_state
```

Advantages:

- Simple and easy to implement
- Converges to optimal policy for small state spaces

Disadvantages:

- Not scalable to large or continuous state spaces
- Requires a table for every state-action pair

---

2. Deep Q-Network (DQN)

Overview: DQN uses a deep neural network to approximate Q-values instead of a table, enabling reinforcement learning in high-dimensional spaces.

Use Cases:

- Playing Atari games from pixel input
- Stock trading bots
- Self-driving car decision making

Code (Simplified):

```
import torch
import torch.nn as nn
import torch.optim as optim

class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )

    def forward(self, x):
        return self.model(x)

# Training would involve using replay buffer and target network
```

Advantages:

- Handles large state spaces with neural networks
- Learns directly from raw inputs (e.g., images)

Disadvantages:

- Requires large memory and compute resources
- Sensitive to hyperparameters like learning rate and epsilon

---

3. Proximal Policy Optimization (PPO)

Overview: PPO is a policy-gradient method that uses a surrogate objective to constrain the update step, improving training stability.

Use Cases:

- Continuous control in robotics
- Game AI (e.g., OpenAI Five for Dota 2)

- Simulated environments like MuJoCo

Code (Using Stable Baselines3):

```
from stable_baselines3 import PPO
from stable_baselines3.common.envs import DummyVecEnv
import gym

env = DummyVecEnv([lambda: gym.make("CartPole-v1")])
model = PPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10000)
```

Advantages:

- Stable and efficient
- Supports both discrete and continuous action spaces

Disadvantages:

- Sample inefficient compared to value-based methods
- Sensitive to hyperparameter tuning

---

4. Actor-Critic Methods

Overview: Combines the benefits of value-based and policy-based methods by training two networks: the actor (policy) and the critic (value function).

Use Cases:

- Real-time strategy games
- Continuous robotic control

Code (Simplified Outline):

```
# Actor selects actions, Critic evaluates them
# Actor loss = log(prob) * advantage
# Critic loss = MSE(reward - value)
```

Advantages:

- Lower variance compared to pure policy gradient
- Suitable for continuous actions

Disadvantages:

- More complex to implement and tune
- Potential instability between actor and critic updates

---

5. A3C (Asynchronous Advantage Actor-Critic)

Overview: An advanced actor-critic algorithm that runs multiple agents in parallel with separate environments to stabilize and speed up training.

Use Cases:

- Large-scale simulation environments
- Real-time decision making in complex scenarios

Code (Conceptual):

```
# Each agent runs in parallel thread, updates global network asynchronously
# Each thread maintains a local network to interact with its environment
```

Advantages:

- Fast convergence using parallel training
- Improved exploration

Disadvantages:

- Complex parallelization and synchronization
- Requires CPU-intensive infrastructure
