Advanced Python AI Techniques

Core Libraries and Frameworks

Python has become the dominant language for AI development due to its simplicity, readability, and flexibility. The ecosystem of libraries and frameworks provides powerful tools for implementing advanced AI solutions.

NumPy for Advanced AI

NumPy is a fundamental library that provides support for large, multi-dimensional arrays and matrices. It forms the foundation of many AI applications through:

```
import numpy as np

# Creating advanced matrix operations for neural networks
matrix = np.random.rand(3,3)
inverse = np.linalg.inv(matrix)  # Matrix inversion
eigenvalues, eigenvectors = np.linalg.eig(matrix)  # Eigendecomposition

# Implementing custom activation functions
def advanced_activation(x):
    return np.where(x > 0, x, np.exp(x) - 1)  # ELU-like function

# Optimized tensor operations
tensor1 = np.random.randn(100, 100, 100)
tensor2 = np.random.randn(100, 100, 100)
result = np.tensordot(tensor1, tensor2, axes=([1,2], [0,1]))
```

Pandas for AI Data Processing

Pandas provides sophisticated data manipulation capabilities essential for AI preprocessing:

```
import pandas as pd

# Advanced feature engineering
df = pd.read_csv('data.csv')
df['log_feature'] = np.log1p(df['feature'])
df['interaction'] = df['feature1'] * df['feature2']

# Time series processing for sequential models
```

```
df['rolling_mean'] = df['value'].rolling(window=7).mean()
df['expanding_std'] = df['value'].expanding().std()

# Efficient categorical encoding
df = pd.get_dummies(df, columns=['category_feature'], drop_first=True)
```

Scikit-learn for Advanced ML

Scikit-learn provides sophisticated machine learning algorithms and tools:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

# Advanced pipeline with dimensionality reduction
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=10)),
    ('classifier', RandomForestClassifier(n_estimators=100))
])

# Stacking ensemble for superior performance
estimators = [
    ('rf', RandomForestClassifier(n_estimators=100)),
    ('svm', SVC(probability=True))
]
stacking_clf = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression()
)
```

Deep Learning Superpower Techniques

TensorFlow and PyTorch Advanced Patterns

```
import tensorflow as tf

# Custom gradient computation
@tf.custom_gradient
def custom_activation(x):
    result = tf.math.tanh(x)
    def grad(dy):
```

```
        return dy * (1 - tf.square(result)) * 1.05  # Gradient boosting
    return result, grad

# Advanced model architecture with residual connections
class ResidualBlock(tf.keras.layers.Layer):
    def __init__(self, filters, kernel_size=3):
        super(ResidualBlock, self).__init__()
        self.conv1 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn1 = tf.keras.layers.BatchNormalization()
        self.conv2 = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')
        self.bn2 = tf.keras.layers.BatchNormalization()

    def call(self, inputs, training=False):
        x = self.conv1(inputs)
        x = self.bn1(x, training=training)
        x = tf.nn.relu(x)
        x = self.conv2(x)
        x = self.bn2(x, training=training)
        return tf.nn.relu(x + inputs)  # Skip connection
```

PyTorch Dynamic Computation Graphs

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

# Dynamic computation graph with conditional execution
class DynamicNet(nn.Module):
    def __init__(self):
        super(DynamicNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.fc1 = nn.Linear(128 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x, use_residual=True):
        x1 = F.relu(self.conv1(x))
        x2 = self.conv2(x1)

        # Dynamic residual connection based on runtime condition
        if use_residual and x1.size() == x2.size():
            x = F.relu(x2 + x1)
        else:
            x = F.relu(x2)

        x = F.adaptive_avg_pool2d(x, (8, 8))
        x = x.view(-1, 128 * 8 * 8)
        x = F.dropout(F.relu(self.fc1(x)), training=self.training)
```

```
        x = self.fc2(x)
        return x
```

Advanced Reinforcement Learning

```
 import gym
import numpy as np
from collections import deque
import random

# Implementing Double Q-learning with experience replay
class AdvancedDQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=100000)
        self.gamma = 0.99  # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()
        self.target_model = self._build_model()
        self.update_target_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = tf.keras.Sequential([
            tf.keras.layers.Dense(24, input_dim=self.state_size, activation='relu'),
            tf.keras.layers.Dense(24, activation='relu'),
            tf.keras.layers.Dense(self.action_size, activation='linear')
        ])
        model.compile(loss='mse',
optimizer=tf.keras.optimizers.Adam(lr=self.learning_rate))
        return model

    def update_target_model(self):
        # copy weights from model to target_model
        self.target_model.set_weights(self.model.get_weights())

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])
```

```
    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                # Double DQN: select action using model, evaluate using target model
                a = np.argmax(self.model.predict(next_state)[0])
                target = reward + self.gamma * self.target_model.predict(next_state)[0][a]
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Generative AI and Transformers

```
 import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Advanced text generation with control codes
class ControlledTextGenerator:
    def __init__(self, model_name="gpt2-medium"):
        self.tokenizer = GPT2Tokenizer.from_pretrained(model_name)
        self.model = GPT2LMHeadModel.from_pretrained(model_name)
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.model.to(self.device)

    def generate_controlled_text(self, prompt, control_code, max_length=100):
        # Prepend control code to guide generation
        input_text = f"{control_code} {prompt}"
        input_ids = self.tokenizer.encode(input_text,
return_tensors="pt").to(self.device)

        # Generate with nucleus sampling for more creative outputs
        output = self.model.generate(
            input_ids,
            max_length=max_length,
            num_return_sequences=1,
            do_sample=True,
            top_p=0.92,
            top_k=50,
            temperature=0.85,
            repetition_penalty=1.2,
            no_repeat_ngram_size=2
        )

        return self.tokenizer.decode(output[0], skip_special_tokens=True)
```

Advanced Computer Vision Techniques

```
import torch
import torch.nn as nn
import torchvision.models as models

# Feature extraction and transfer learning with attention
class AttentionFeatureExtractor(nn.Module):
    def __init__(self, pretrained=True):
        super(AttentionFeatureExtractor, self).__init__()
        # Use pretrained ResNet as base
        resnet = models.resnet50(pretrained=pretrained)
        self.base = nn.Sequential(*list(resnet.children())[:-2])

        # Channel attention module
        self.channel_attention = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(2048, 512, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(512, 2048, kernel_size=1),
            nn.Sigmoid()
        )

        # Spatial attention module
        self.spatial_attention = nn.Sequential(
            nn.Conv2d(2048, 1, kernel_size=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        features = self.base(x)

        # Apply channel attention
        channel_weights = self.channel_attention(features)
        features = features * channel_weights

        # Apply spatial attention
        spatial_weights = self.spatial_attention(features)
        features = features * spatial_weights

        return features
```

Sources

- [Advanced Python Techniques for AI: Using NumPy, Pandas, and Scikit-learn with Examples]()
- [Advanced Techniques for Artificial Intelligence with Python]()