PyTorch Classes for ML Models

1. Linear Regression

```python
import torch
import torch.nn as nn

class LinearRegression(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return self.linear(x)
```

2. Feedforward Neural Network (FNN)

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

class FeedForwardNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedForwardNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

3. Convolutional Neural Network (CNN)

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

class CNN(nn.Module):
    def __init__(self, input_channels, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)  # Assumes input images are 32x32
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = F.relu(self.fc1(x))
```

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

4. U-Net

```
import torch
import torch.nn as nn

class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1):
        super(UNet, self).__init__()

        def conv_block(in_ch, out_ch):
            return nn.Sequential(
                nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1),
                nn.ReLU(inplace=True)
            )

        self.encoder1 = conv_block(in_channels, 64)
        self.encoder2 = conv_block(64, 128)
        self.encoder3 = conv_block(128, 256)
        self.encoder4 = conv_block(256, 512)

        self.pool = nn.MaxPool2d(2, 2)

        self.decoder1 = conv_block(512 + 256, 256)
        self.decoder2 = conv_block(256 + 128, 128)
        self.decoder3 = conv_block(128 + 64, 64)

        self.upconv1 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.upconv3 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)

        self.final_conv = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        # Encoder path
        enc1 = self.encoder1(x)
        enc2 = self.encoder2(self.pool(enc1))
        enc3 = self.encoder3(self.pool(enc2))
        enc4 = self.encoder4(self.pool(enc3))

        # Decoder path
        dec1 = self.upconv1(enc4)
        dec1 = torch.cat((dec1, enc3), dim=1)
        dec1 = self.decoder1(dec1)

        dec2 = self.upconv2(dec1)
        dec2 = torch.cat((dec2, enc2), dim=1)
        dec2 = self.decoder2(dec2)

        dec3 = self.upconv3(dec2)
        dec3 = torch.cat((dec3, enc1), dim=1)
        dec3 = self.decoder3(dec3)

        out = self.final_conv(dec3)
        return out
```

5. General Autoencoder

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

```

class AutoEncoder(nn.Module):
    def _init_(self, input_dim, hidden_dim, bottleneck_dim):
        super(AutoEncoder, self)._init_()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, bottleneck_dim),
            nn.ReLU()
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(bottleneck_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()  # Assuming normalized input (e.g., values between 0 and 1)
        )

    def forward(self, x):
        # Encode to latent space
        z = self.encoder(x)
        # Decode to reconstruct input
        recon_x = self.decoder(z)
        return recon_x
```

6. Variational Autoencoder (VAE)

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

class VAE(nn.Module):
    def _init_(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self)._init_()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        self.mu_layer = nn.Linear(hidden_dim, latent_dim)  # Mean of latent space
        self.logvar_layer = nn.Linear(hidden_dim, latent_dim)  # Log-variance of latent space

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()  # Assuming normalized input (e.g., images in [0, 1])
        )

    def encode(self, x):
        h = self.encoder(x)
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        return mu, logvar

    def reparameterize(self, mu, logvar):
```

```
        """Reparameterization trick: z = mu + std * epsilon"""
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + std * eps

    def decode(self, z):
        return self.decoder(z)

    def forward(self, x):
        # Encode input to latent space
        mu, logvar = self.encode(x)
        # Reparameterize to sample latent vector
        z = self.reparameterize(mu, logvar)
        # Decode to reconstruct input
        recon_x = self.decode(z)
        return recon_x, mu, logvar
```

7. Convolutional Autoencoder

Encoder

```
class ConvEncoder(nn.Module):
    def _init_(self):
        super(ConvEncoder, self)._init_()
        self.model = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),  # Output: 16x14x14
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1),  # Output: 32x7x7
            nn.ReLU()
        )

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

Decoder

```
class ConvDecoder(nn.Module):
    def _init_(self):
        super(ConvDecoder, self)._init_()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2, padding=1, output_padding=1),  # Output: 16x14x14
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, kernel_size=3, stride=2, padding=1, output_padding=1),  # Output: 1x28x28
            nn.Sigmoid()
        )

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

Complete Convolutional Autoencoder

```
class ConvAutoEncoder(nn.Module):
    def _init_(self):
        super(ConvAutoEncoder, self)._init_()
        self.encoder = ConvEncoder()
        self.decoder = ConvDecoder()

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