# Dimensionality Reduction using Autoencoders

An introduction to deep learning and data analysis using autoencoders for dimensionality reduction. Explore methods for simplifying complex datasets while preserving essential information.

# What is Dimensionality Reduction?

### Definition

Reduces high-dimensional data to lower dimensions.

### Why do we need it?

- Faster processing
- Easier data visualization
- Noise reduction
- Better model performance

### Examples

- Images
- Audio
- Genetic data

# Traditional Methods: PCA

### PCA (Principal Component Analysis)

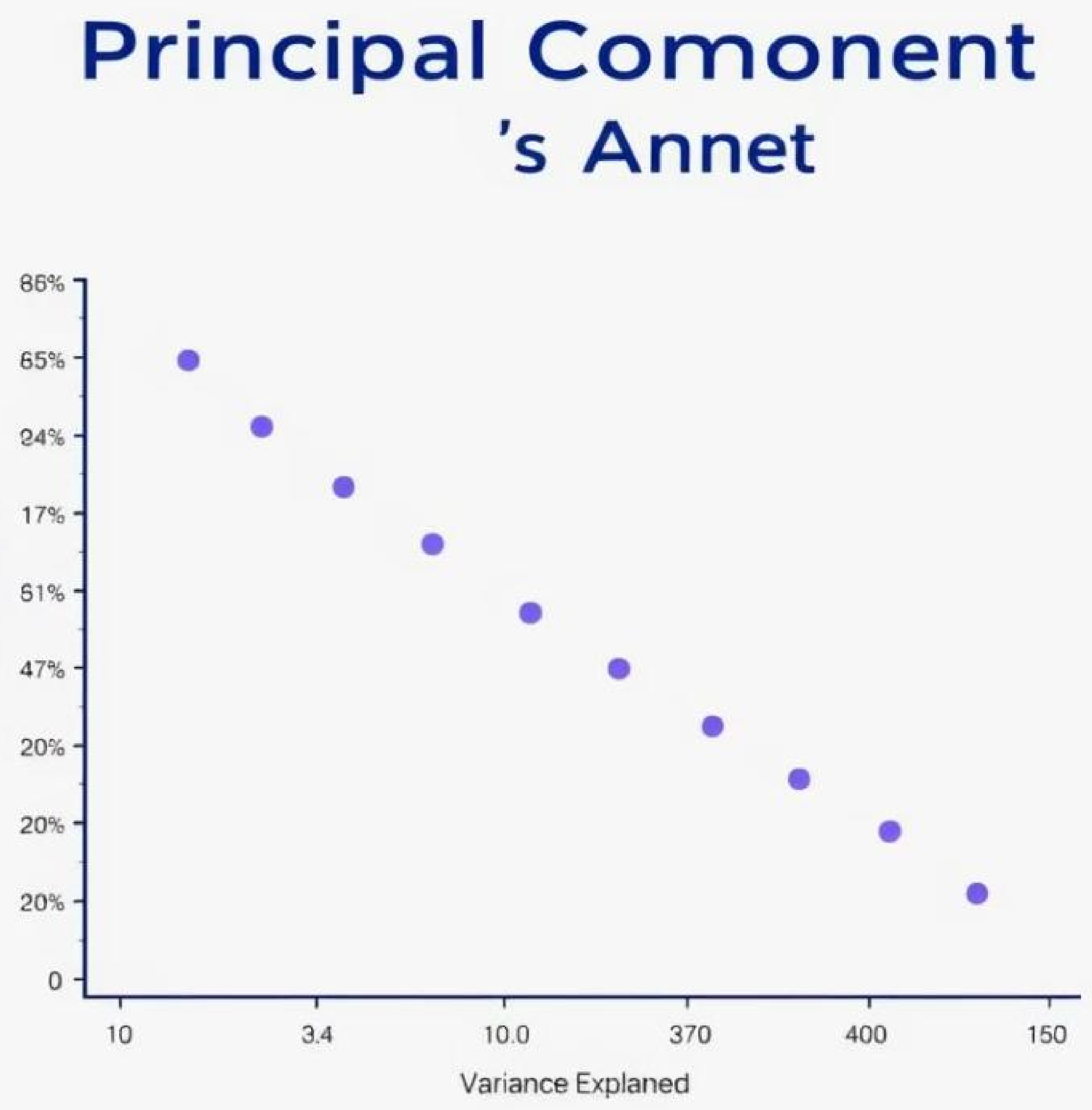
Based on linear transformation.

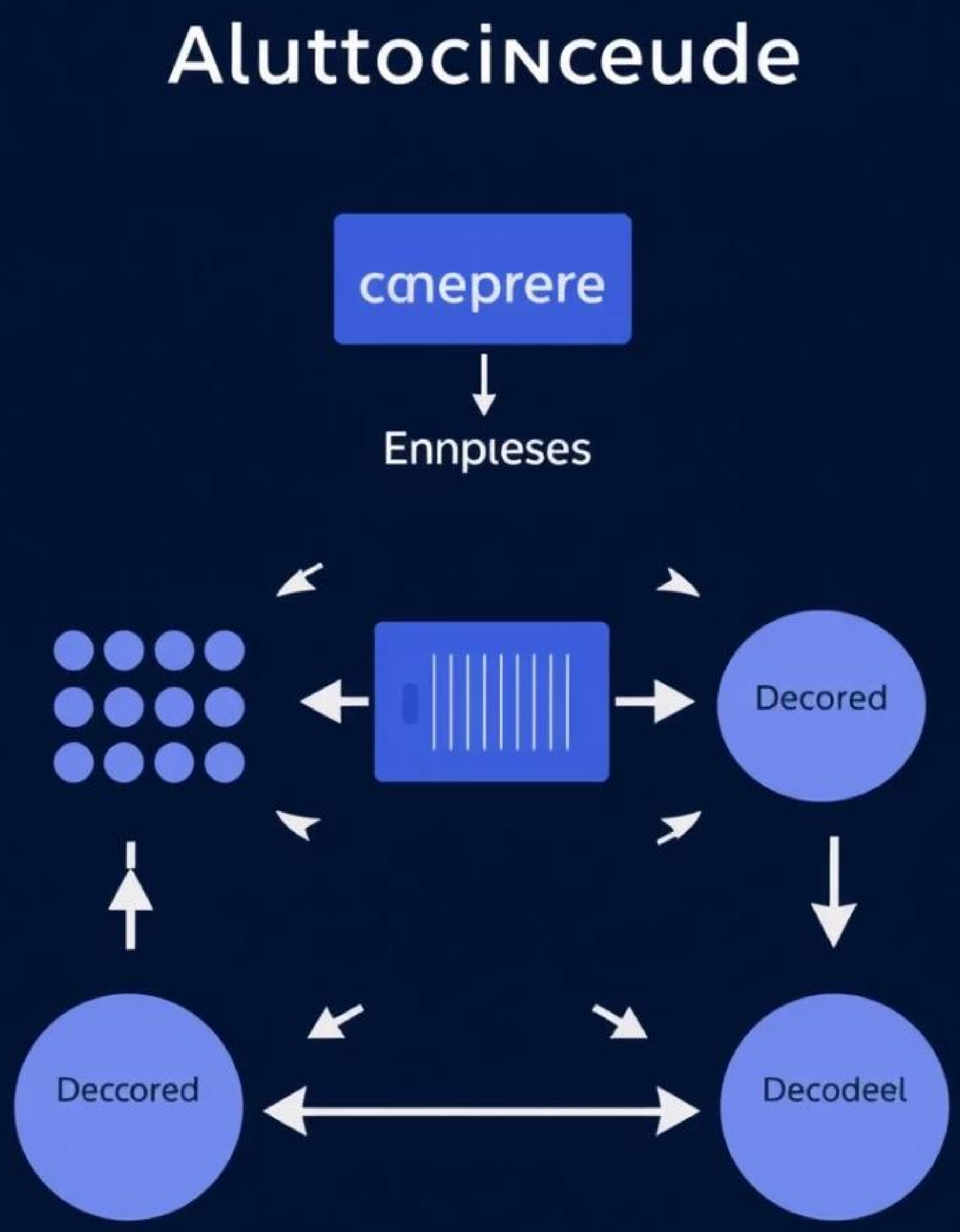
### Reduces dimensions

Captures the maximum variance.

### Easy to interpret

Limited with nonlinear data.






# What is an Autoencoder?

### Unsupervised neural network

Learns to compress data and reconstruct it.

### Encoder

Converts input to a lower dimension.

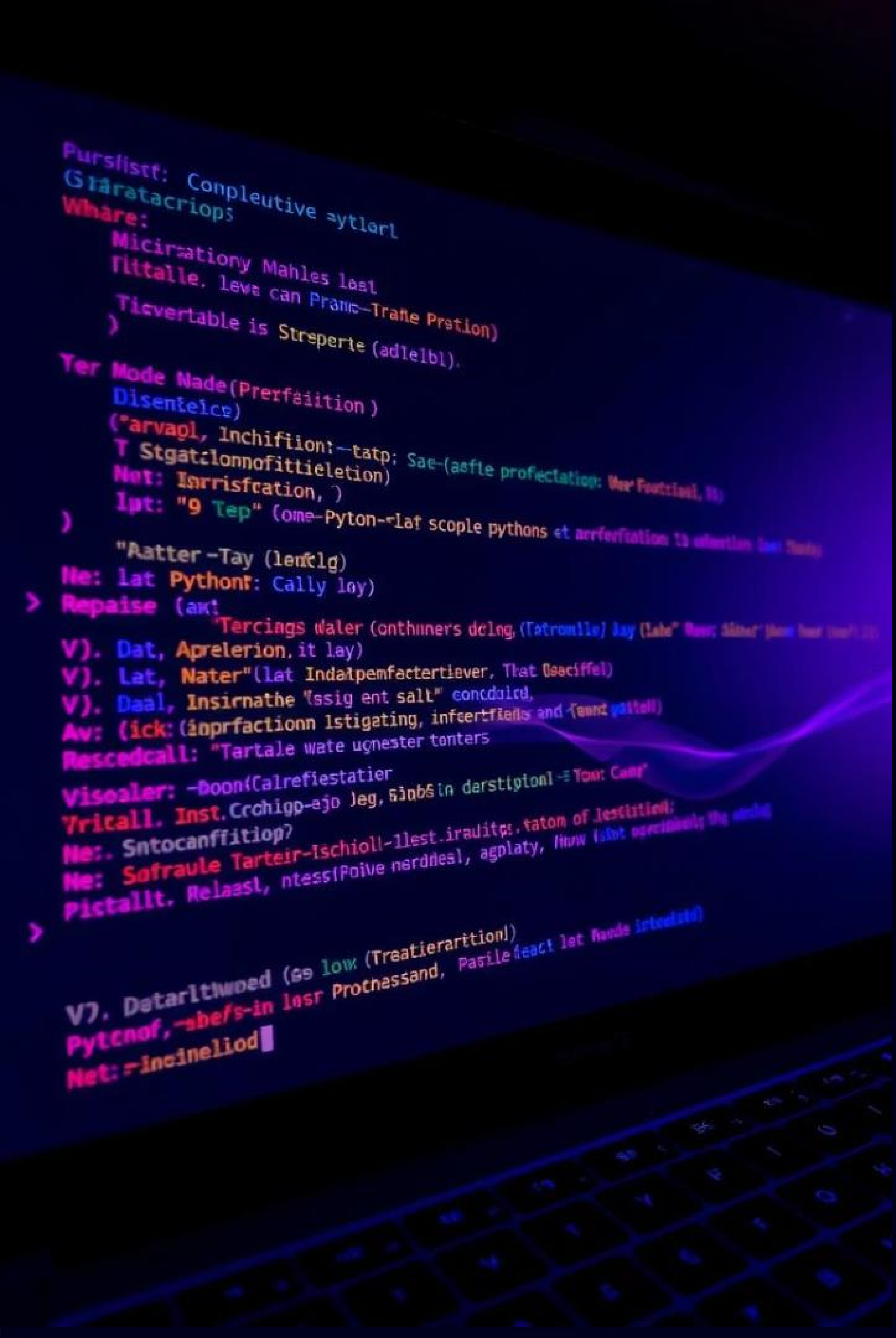
### Decoder

Reconstructs input from the encoded data.

# How Autoencoders Work

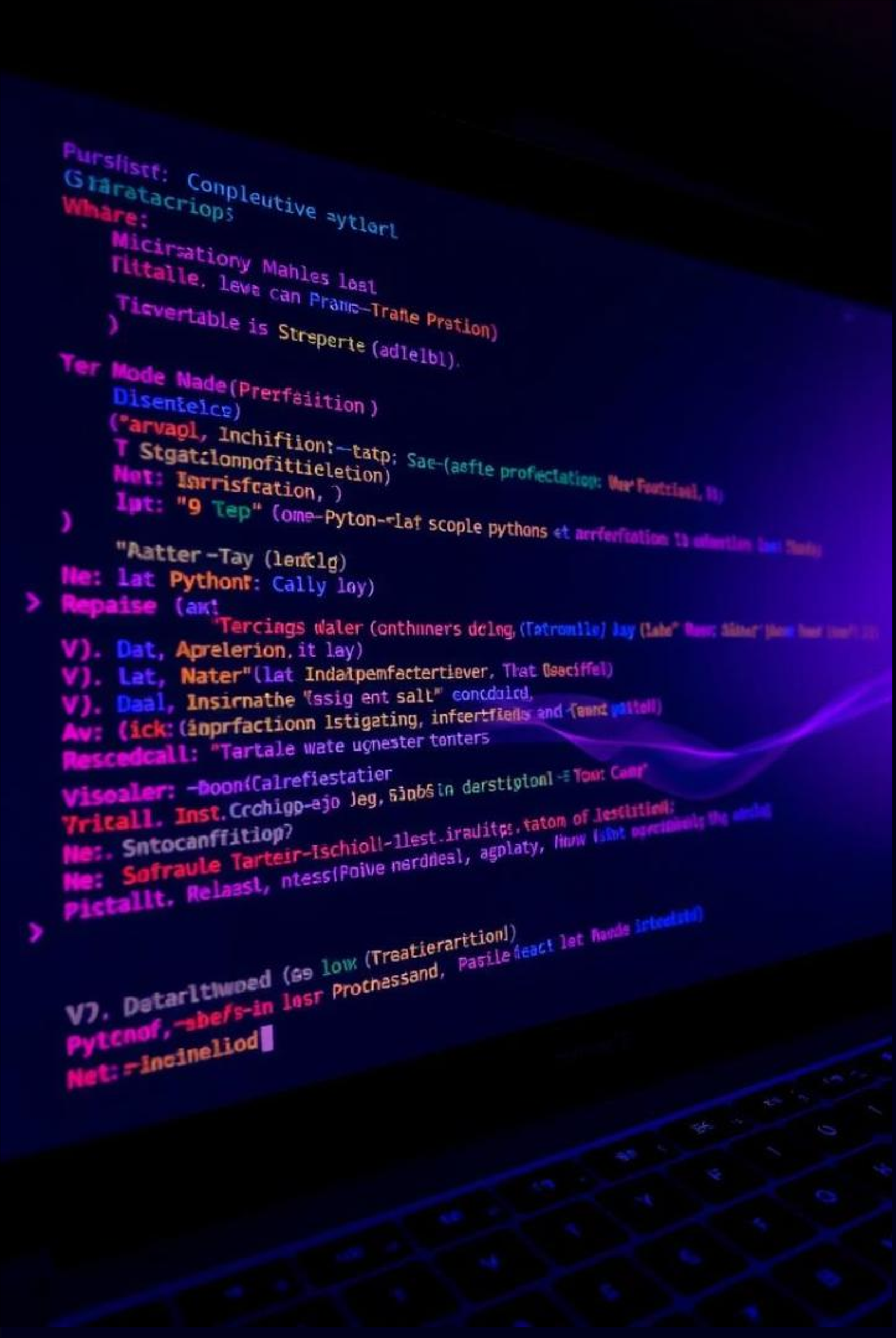
1. **Input**
2. **Encoder**
3. **Bottleneck**
4. **Decoder**
5. **Output**

Goal: Make the output as close as possible to the original input. The "bottleneck" layer holds the compressed representation.



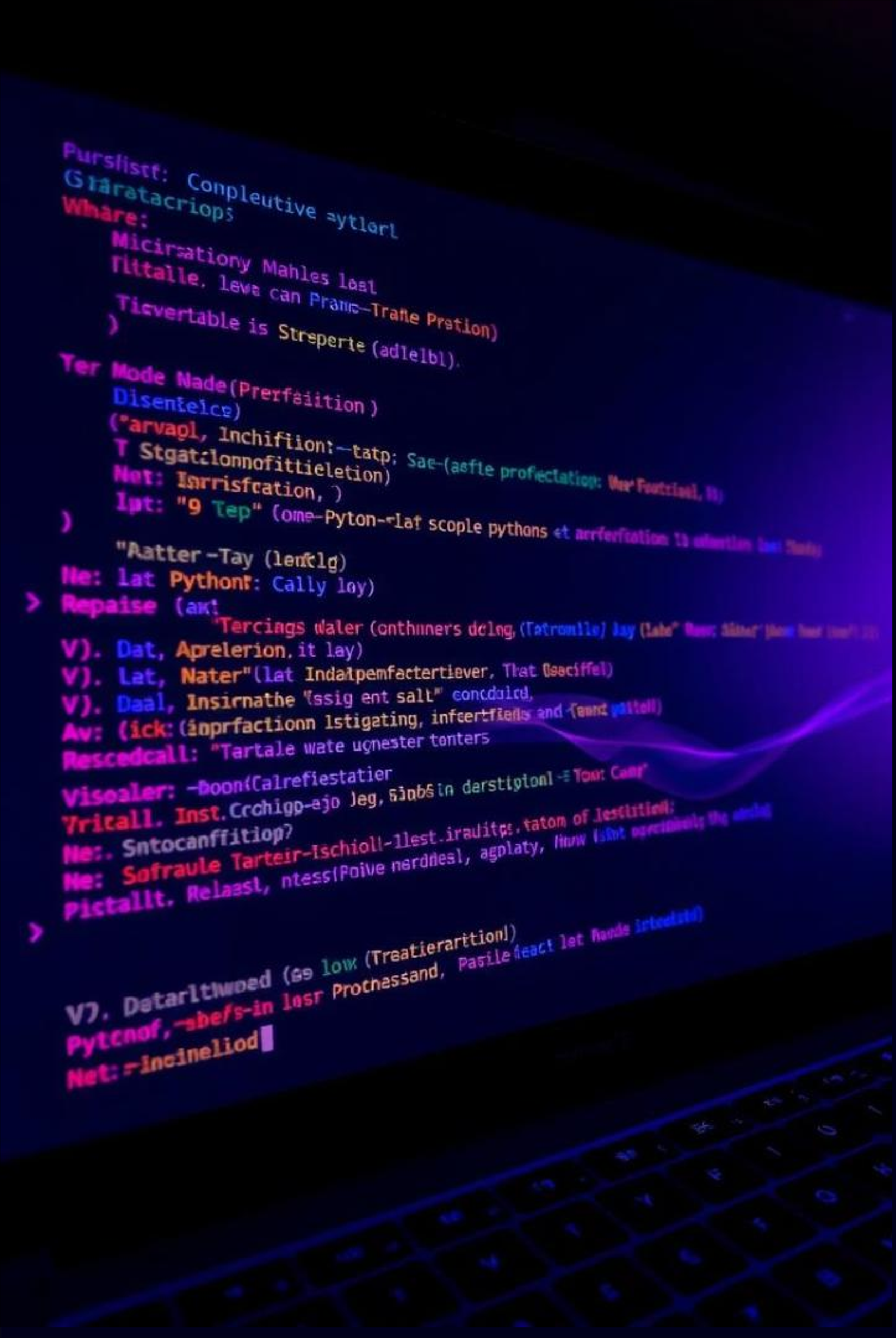
# Practical Example: Code

```
import numpy as np # type: ignore
import pandas as pd # type: ignore
import matplotlib.pyplot as plt # type: ignore
from sklearn.decomposition import PCA # type: ignore
from sklearn.preprocessing import MinMaxScaler # type: ignore
from tensorflow.keras.models import Model # type: ignore
from tensorflow.keras.layers import Input, Dense # type: ignore
from tensorflow.keras.optimizers import Adam # type: ignore
from tensorflow.keras.losses import MeanSquaredError # type: ignore

# 1. Load and preprocess diabetes data
df = pd.read_csv('diabetes.csv')
x_data = df.values

# Scale data to [0, 1] range
scaler = MinMaxScaler()
x_data = scaler.fit_transform(x_data)

# Split into train/test (80/20)
split_idx = int(0.8 * len(x_data))
x_train = x_data[:split_idx]
x_test = x_data[split_idx:]
```



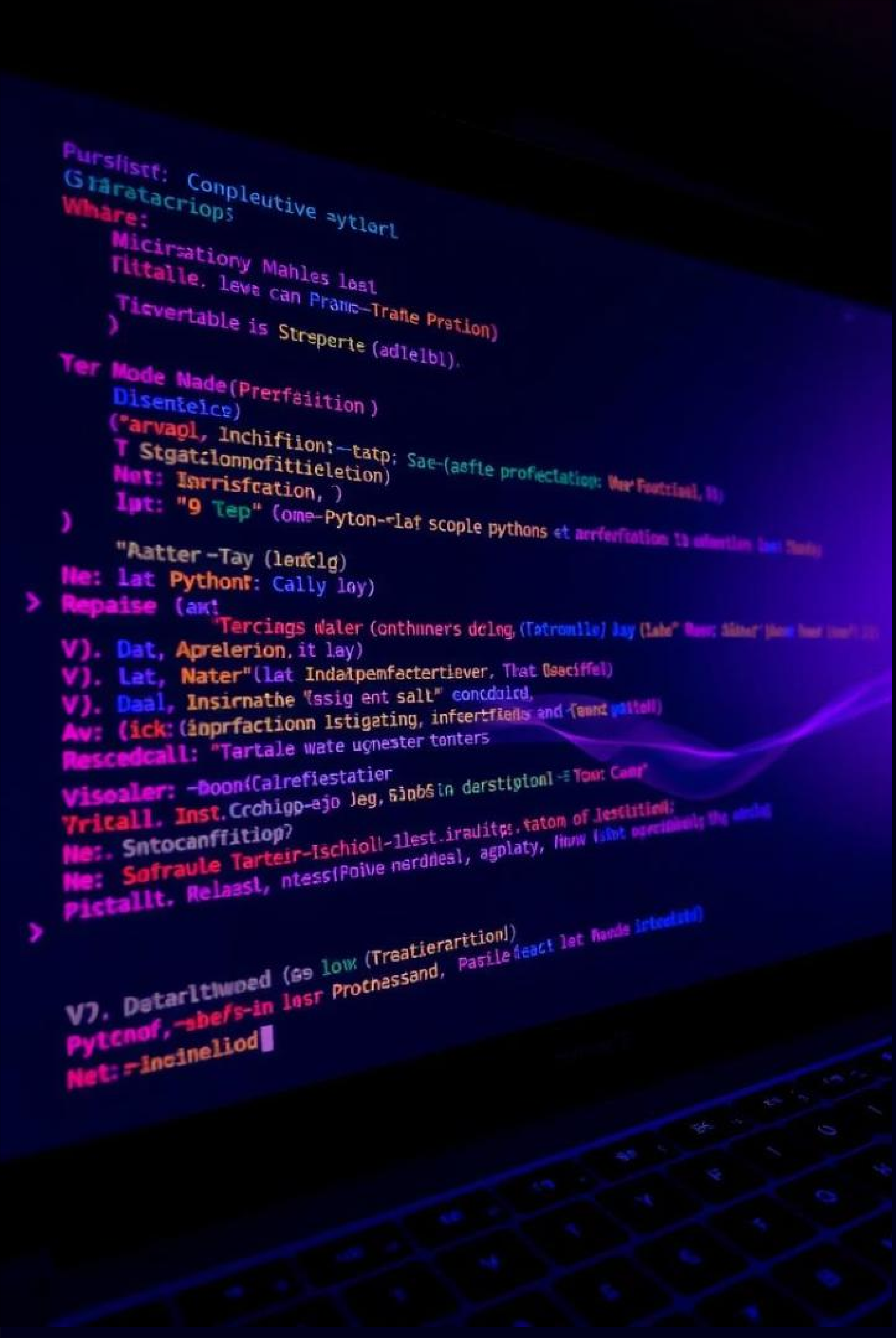
# Practical Code

```
# 2. Atoencoder Model
input_dim = x_train.shape[1]
encoding_dim = 4  # Smaller bottleneck for tabular data
# Encoder
input_layer = Input(shape=(input_dim,))
encoded = Dense(16, activation='relu')(input_layer)
encoded = Dense(8, activation='relu')(encoded)
encoded = Dense(encoding_dim, activation='relu')(encoded)  # Bottleneck
# Decoder
decoded = Dense(8, activation='relu')(encoded)
decoded = Dense(16, activation='relu')(decoded)
decoded = Dense(input_dim, activation='sigmoid')(decoded)

# Full model
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer=Adam(learning_rate=0.001),
loss=MeanSquaredError())
```



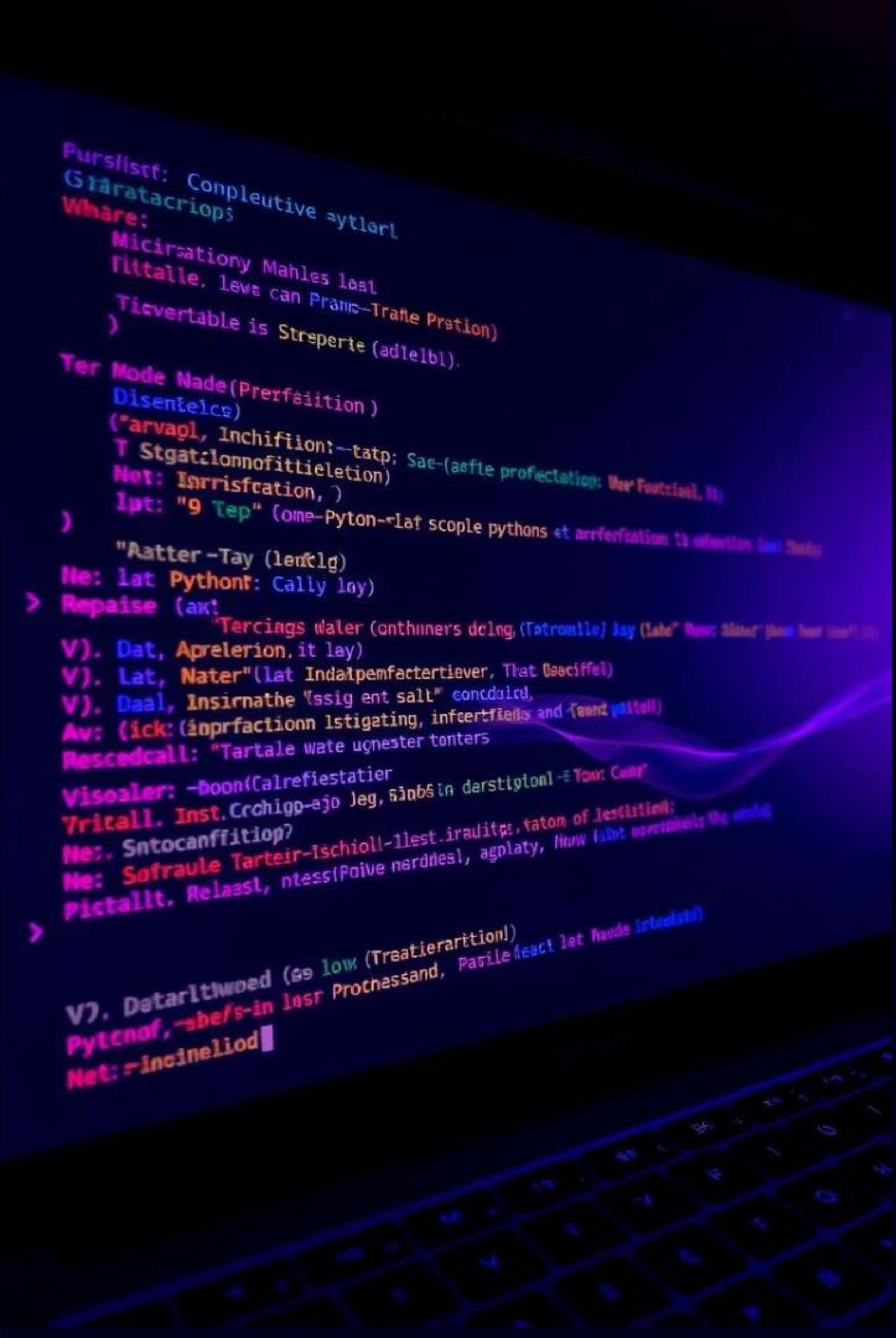
# Practical Example: Code

```
# 3. Train the model
history = autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=32,
                shuffle=True,
                validation_data=(x_test, x_test))

# 4. Reconstruct test data
reconstructed = autoencoder.predict(x_test)

# 5. Compare with PCA
pca = PCA(n_components=encoding_dim)
pca.fit(x_train)
x_pca = pca.transform(x_test)
x_pca_inverse = pca.inverse_transform(x_pca)

# 6. Calculate and compare MSE
def calculate_mse(original, reconstructed, label):
   mse = np.mean(np.square(original - reconstructed))
   print(f"{label} MSE: {mse:.5f}")

calculate_mse(x_test, reconstructed, "Autoencoder")
calculate_mse(x_test, x_pca_inverse, "PCA")
```

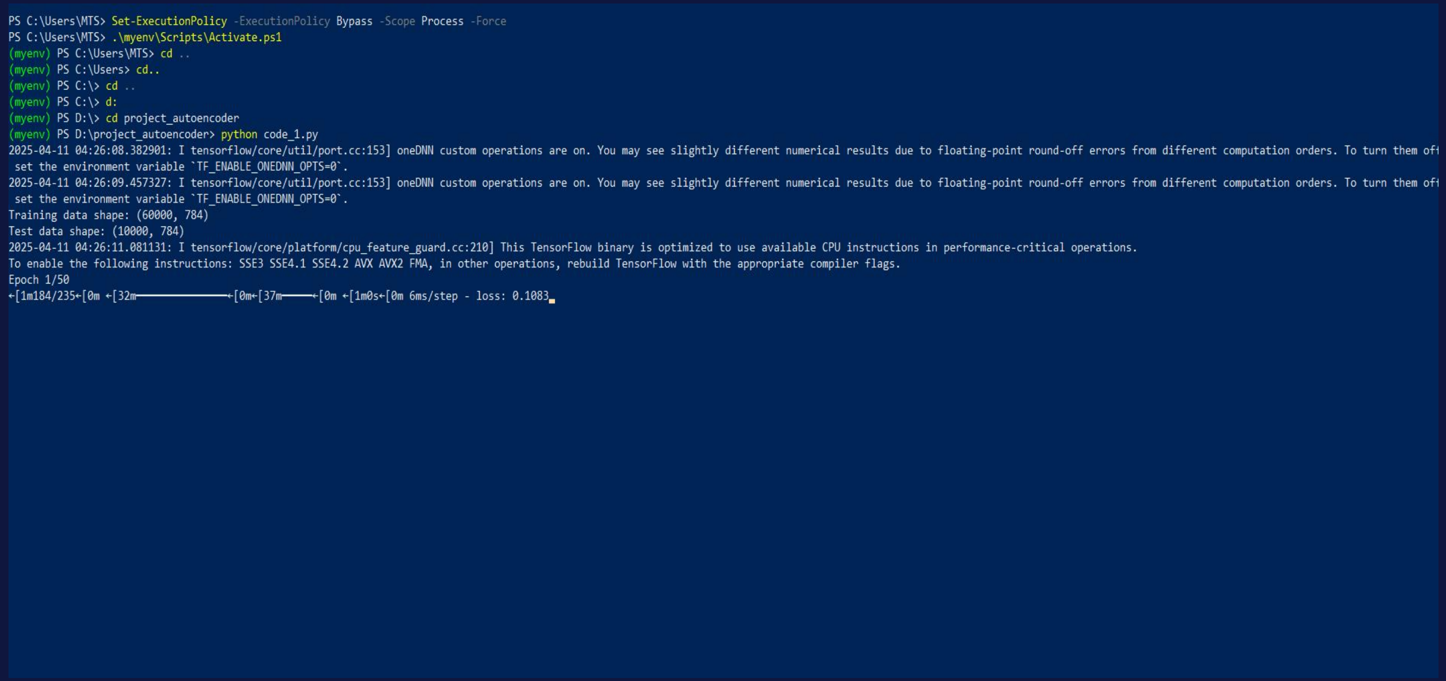


# Practical Example: Code

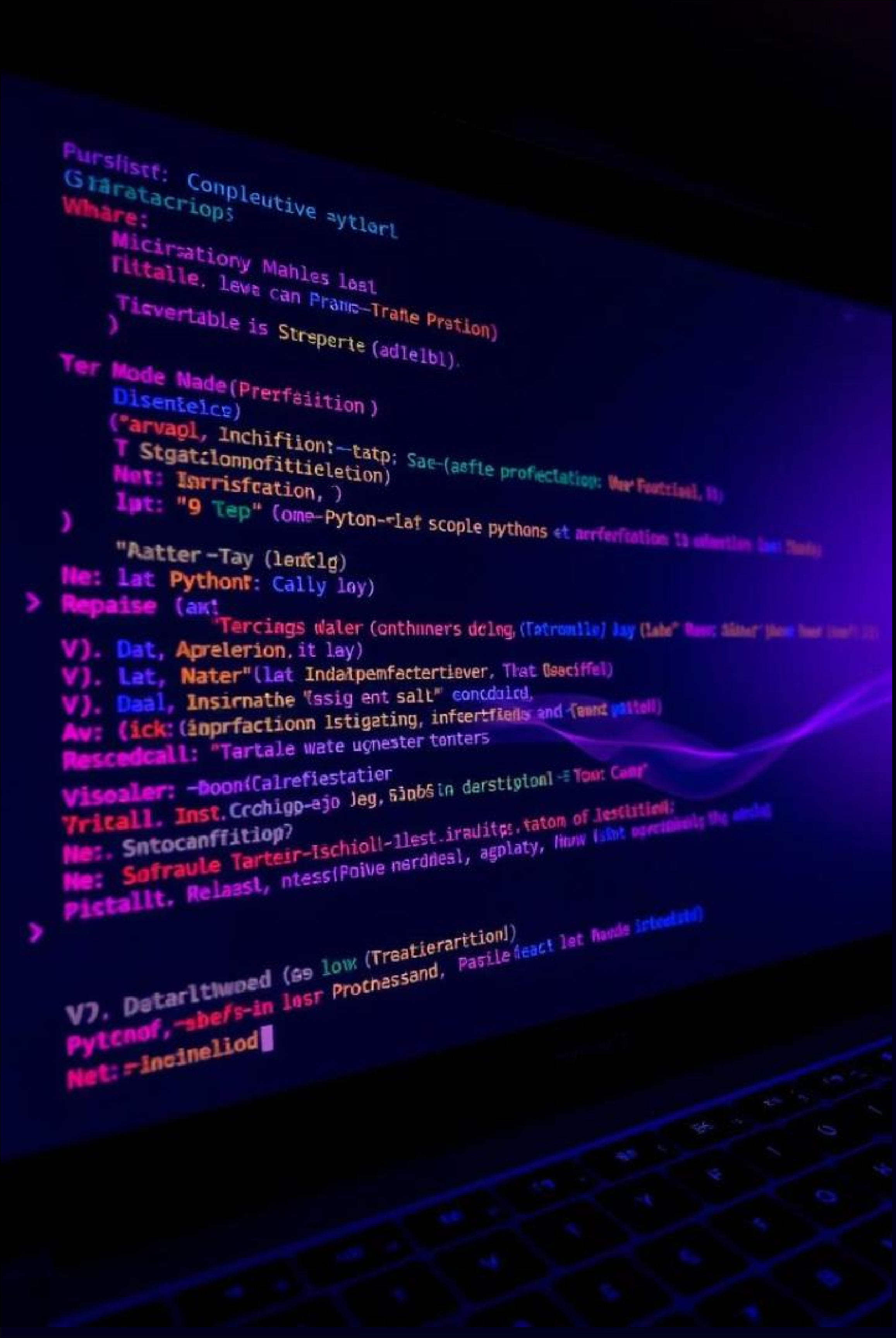
```
# 7. Visualization - Plot first 5 features
plt.figure(figsize=(15, 10))
for feat in range(5):  # Plot first 5 features
    plt.subplot(5, 1, feat+1)
    plt.plot(x_test[:50, feat], label='Original', marker='o')
    plt.plot(reconstructed[:50, feat], label='Autoencoder', marker='x')
    plt.plot(x_pca_inverse[:50, feat], label='PCA', marker='^')
    plt.title(f'Feature {feat+1} Comparison')
    plt.legend()
plt.tight_layout()
plt.show()

# 8. Plot training history
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Training History')
plt.ylabel('Loss (MSE)')
plt.xlabel('Epoch')
plt.legend()
plt.grid(True)
plt.show()
```



# output



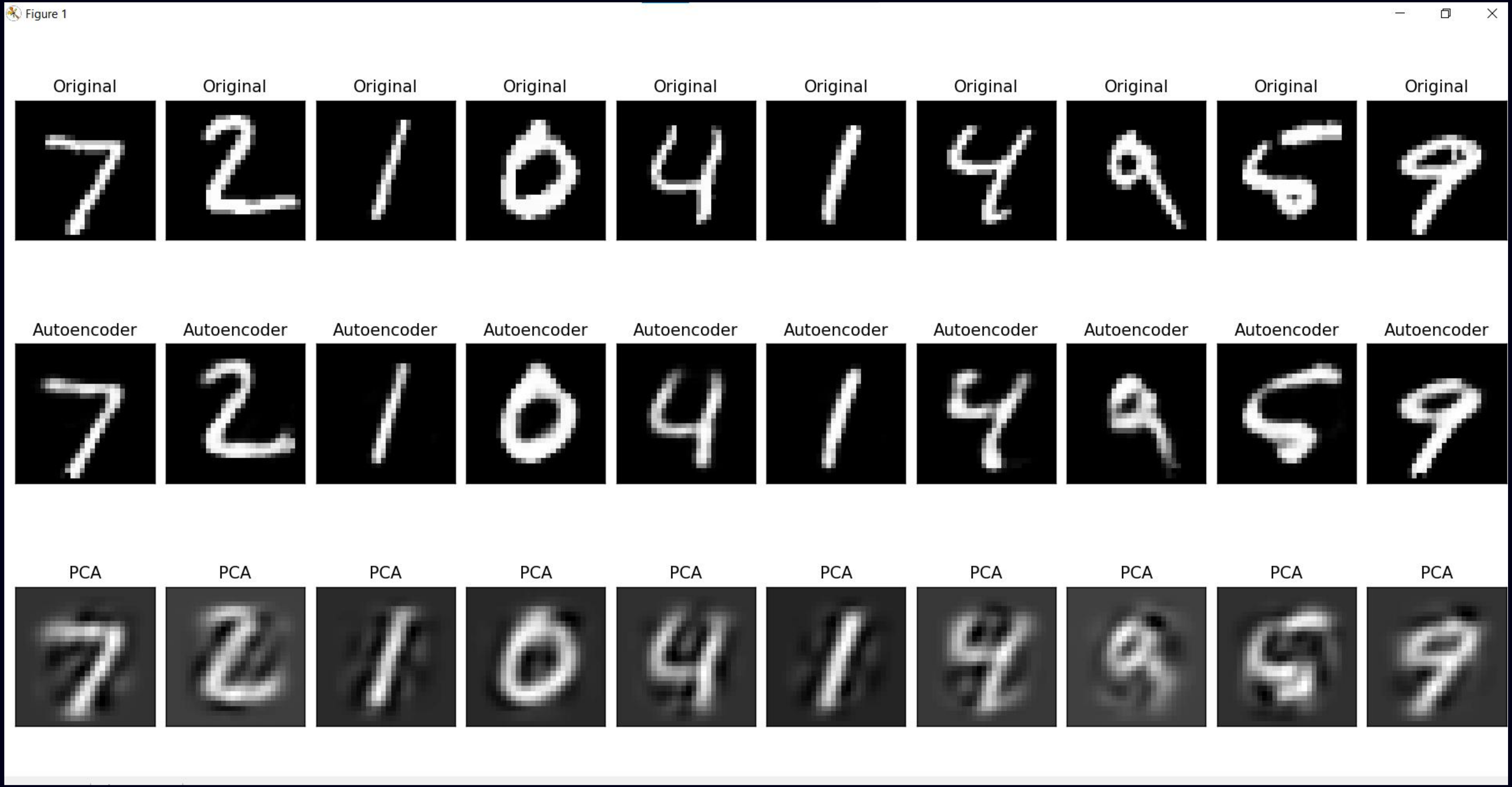
```
PS C:\Users\MTS> Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process -Force
PS C:\Users\MTS> .\myenv\Scripts\Activate.ps1
(myenv) PS C:\Users\MTS> cd ..
(myenv) PS C:\Users> cd..
(myenv) PS C:\> cd ..
(myenv) PS C:\> d:
(myenv) PS D:\> cd project_autoencoder
(myenv) PS D:\project_autoencoder> python code_1.py
2025-04-11 04:26:08.382901: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them of
 set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-04-11 04:26:09.457327: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them of
 set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
Training data shape: (60000, 784)
Test data shape: (10000, 784)
2025-04-11 04:26:11.081131: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/50
←[1m184/235←[0m ←[32m━━━━━━━━━━━━━━━←[0m←[37m━━━━←[0m ←[1m0s←[0m 6ms/step - loss: 0.1083
```



# output

```
Epoch 22/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0105 - val_loss: 0.0102
Epoch 23/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0103 - val_loss: 0.0102
Epoch 24/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0102 - val_loss: 0.0099
Epoch 25/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0100 - val_loss: 0.0097
Epoch 26/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0098 - val_loss: 0.0095
Epoch 27/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0097 - val_loss: 0.0095
Epoch 28/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0096 - val_loss: 0.0093
Epoch 29/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0094 - val_loss: 0.0092
Epoch 30/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0093 - val_loss: 0.0092
Epoch 31/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0092 - val_loss: 0.0090
Epoch 32/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0091 - val_loss: 0.0089
Epoch 33/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0090 - val_loss: 0.0089
Epoch 34/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0090 - val_loss: 0.0089
Epoch 35/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0089 - val_loss: 0.0089
Epoch 36/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0088 - val_loss: 0.0087
Epoch 37/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0088 - val_loss: 0.0087
Epoch 38/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0087 - val_loss: 0.0087
Epoch 39/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0087 - val_loss: 0.0086
Epoch 40/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0086 - val_loss: 0.0086
Epoch 41/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 6ms/step - loss: 0.0086 - val_loss: 0.0084
Epoch 42/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 5ms/step - loss: 0.0085 - val_loss: 0.0084
Epoch 43/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0085 - val_loss: 0.0085
Epoch 44/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0084 - val_loss: 0.0084
Epoch 45/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0084 - val_loss: 0.0084
Epoch 46/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0083 - val_loss: 0.0084
Epoch 47/50
←[1m235/235←[0m ←[32m━━━━━━━━━━━━━━━━━━━━←[0m←[37m←[0m ←[1m1s←[0m 4ms/step - loss: 0.0083 - val_loss: 0.0083
```

# final output



# Autoencoder vs PCA Results

**Original Images**

**Autoencoder Reconstruction**

**PCA Reconstruction**

# Challenges & Key Takeaways

1. **Small Datasets**
   Difficult to train.
2. **Bottleneck Size**
   Choosing the right size.
3. **Overfitting**
   Risk of overfitting.
4. **Outperforms PCA**
   On nonlinear data.
5. **Convolutional Layers**
   Can be extended for images.
6. **Unsupervised**
   No need for labeled data.

# Conclusion

### Powerful Tool

For dimensionality reduction.

### Preprocessing

Boosts other models performance.

### Better Results

For complex data vs PCA.

## The Power of Autoencoders

At the heart of dimensionality reduction lies an innovative technique - autoencoders. These neural networks possess a remarkable ability to uncover the underlying structure of complex data, allowing us to extract the most salient features while dramatically reducing the dimensionality. Autoencoders work by learning to encode the input data into a compact, low-dimensional representation, and then decode that representation back into the original input. Through this process, the network is able to identify the essential characteristics that define the data, discarding the superfluous details.

Unlike traditional methods like Principal Component Analysis (PCA), autoencoders are not limited by linear assumptions. They can capture intricate non-linear relationships, unlocking new possibilities for dimensionality reduction and feature extraction. This makes them a powerful tool for a wide range of applications, from image processing to natural language understanding.

By leveraging the power of autoencoders, we can gain deeper insights, streamline our workflows, and uncover hidden patterns that may have eluded us using conventional techniques. The key is to harness this innovative approach and unlock the true potential of your data.

Further research autoencoders and their applications in real-world scenarios. Experiment with different architectures and datasets to gain a deeper understanding.