Leveraging Autoencoders for Anomaly Detection: A Case Study with the KDD Cup 1999 Dataset

Rany ElHousieny · Follow
Published in Level Up Coding
14 min read · Mar 16, 2024

Listen Share More

Anomaly detection is a crucial task in various domains such as cybersecurity, fraud detection, and industrial systems monitoring. It involves identifying patterns in data that do not conform to expected behavior. Autoencoders, a type of neural network, have gained popularity in anomaly detection due to their ability to learn compressed representations of data, making them adept at capturing normal patterns and highlighting anomalies.

This article is a continuation of the following article:

Demystifying AutoEncoders: The Architects of Data Compression and Reconstruction

In the vast and ever-evolving landscape of machine learning, AutoEncoders stand out as a fascinating subset of neural…

www.linkedin.com

Autoencoders Dimensionality Reduction with by Example

In the world of machine learning, dimensionality reduction is a critical process, especially in tasks involving...

www.linkedin.com

Understanding Autoencoders

An autoencoder is a neural network that aims to learn a compressed, encoded representation of input data, typically for the purpose of dimensionality reduction

autoencoder is a close approximation of the input.

In the context of anomaly detection, autoencoders are trained on normal data to learn the underlying patterns. When new data is presented, the autoencoder attempts to reconstruct it. If the reconstruction error is significantly high, the data is considered an anomaly, as it deviates from the learned normal pattern.

Case Study: Anomaly Detection with the KDD Cup 1999 Dataset

The KDD Cup 1999 dataset is a widely used benchmark dataset for evaluating anomaly detection algorithms. It contains network connection records, each labeled as either normal or an attack (anomaly).

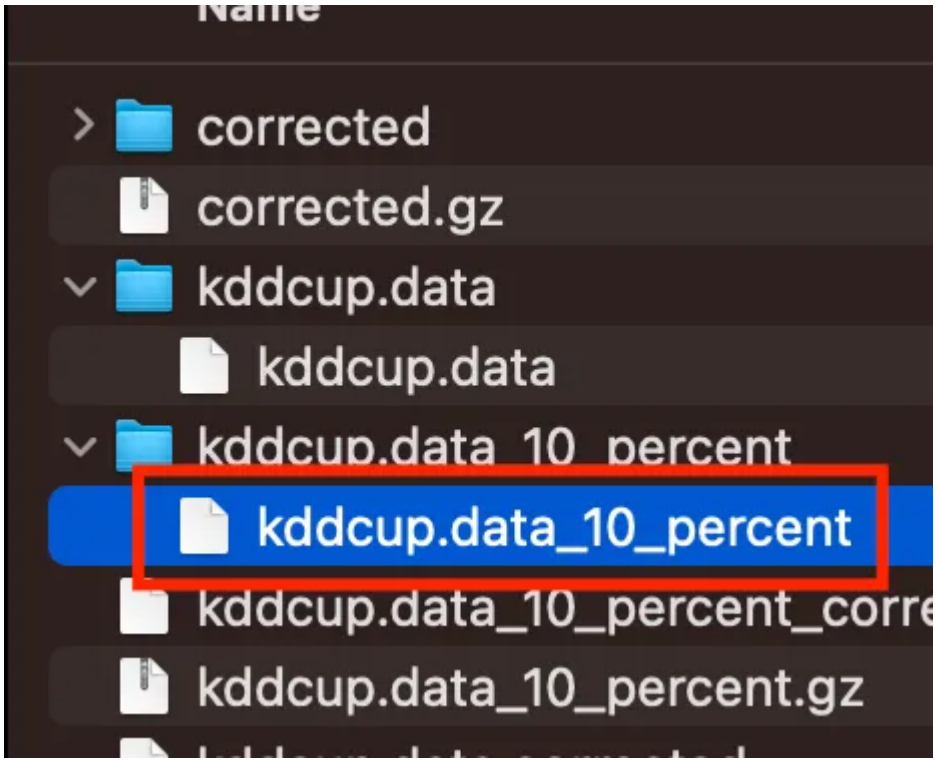
Download the KDD dataset from Kaggle:

KDD Cup 1999 Data

Computer network intrusion detection

www.kaggle.com

After you download the folder and uncompress it, we will use the 10% sample. Copy it to your local folder with the notebook under a folder named data



Exploratory Data Analysis

```
import pandas as pd

# load data into pandas dataframe
data = pd.read_csv('./data/kddcup.data_10_percent', header=None)
```

	0	1	2	3	4	5	6	7	8	9	...	32	33	34	35	36	37	38	39	40	41
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9	1.0	0.0	0.11	0.0	0.0	0.0	0.0	0.0	normal.
1	0	tcp	http	SF	239	486	0	0	0	0	...	19	1.0	0.0	0.05	0.0	0.0	0.0	0.0	0.0	normal.
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29	1.0	0.0	0.03	0.0	0.0	0.0	0.0	0.0	normal.

As you can see, there is no feature names. Let's add the column names using the following instructions https://kdd.ics.uci.edu/databases/kddcup99/task.html

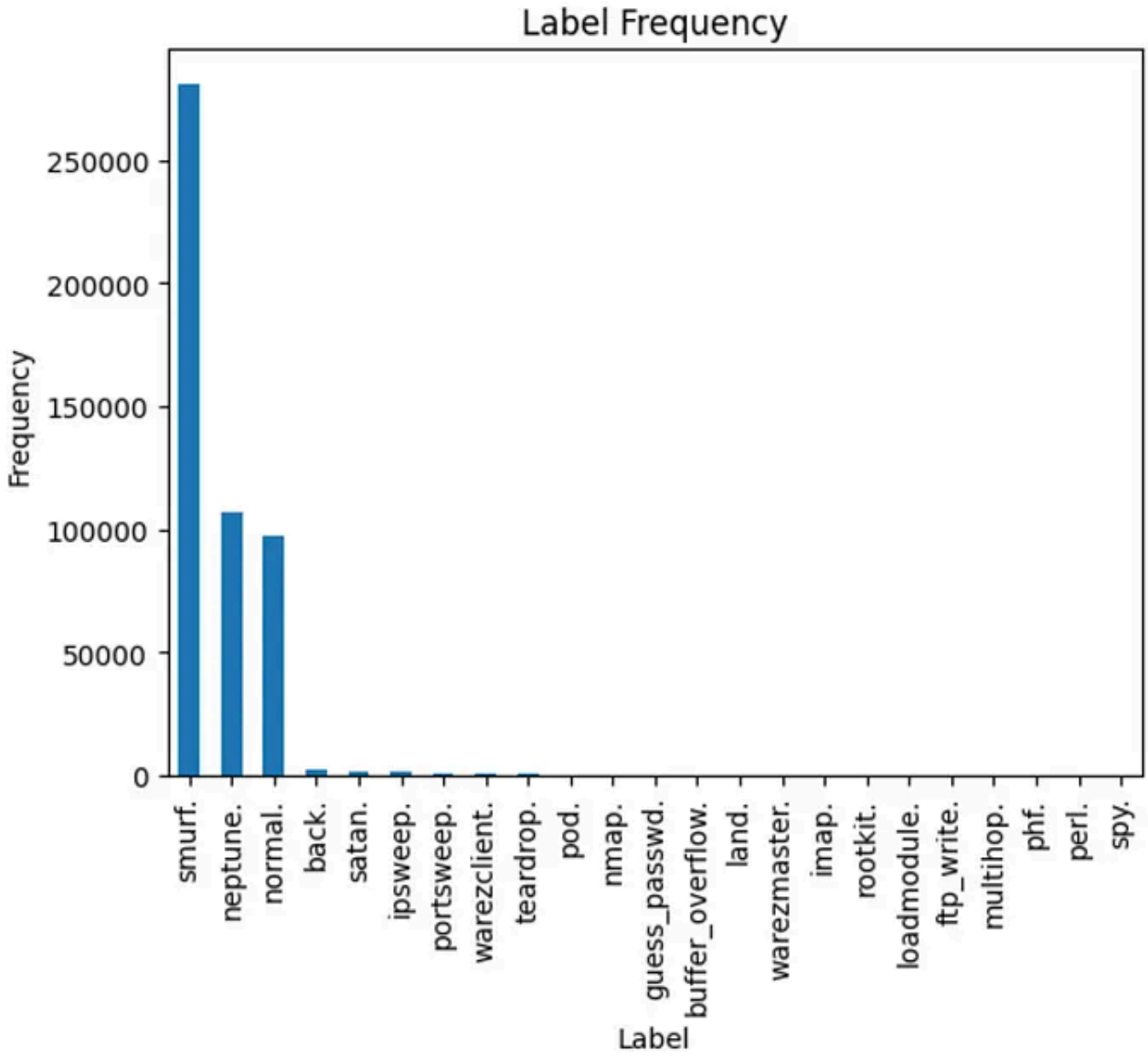
```
# Define the column names
column_names = ["duration", "protocol_type", "service", "flag", "src_bytes",
                "dst_bytes", "land", "wrong_fragment", "urgent", "hot",
                "num_failed_logins", "logged_in", "num_compromised",
                "root_shell", "su_attempted", "num_root", "num_file_creations",
```

```
                    "num_shells", "num_access_files", "num_outbound_cmds",
                    "is_host_login", "is_guest_login", "count", "srv_count",
                    "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_ra
                    "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
                    "dst_host_count", "dst_host_srv_count", "dst_host_same_srv_rate
                    "dst_host_diff_srv_rate", "dst_host_same_src_port_rate",
                    "dst_host_srv_diff_host_rate", "dst_host_serror_rate",
                    "dst_host_srv_serror_rate", "dst_host_rerror_rate",
                    "dst_host_srv_rerror_rate", "label"]

# Assign the column names to the dataset
data.columns = column_names
```

```
# Display the first 10 rows of the dataset
pd.set_option('display.max_columns', None) # Display all columns
data.head(10)
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes
0	0	tcp	http	SF	181	5450
1	0	tcp	http	SF	239	486
2	0	tcp	http	SF	235	1337
3	0	tcp	http	SF	219	1337

```
import matplotlib.pyplot as plt

# Define a function to plot the value counts of each label in the dataset
def plot_value_count(data):
    label_counts = data['label'].value_counts()
    label_counts.plot(kind='bar')
    plt.xlabel('Label')
    plt.ylabel('Frequency')
    plt.title('Label Frequency')
    plt.show()

plot_value_count(data)
```



The bar plot above, shows the frequency distribution of labels in the KDD Cup 1999 dataset. Here's a detailed explanation of what the plot indicates:

1. X-axis (Label): This axis represents the different labels (types of network connections) in the dataset. These labels indicate whether a connection is normal or a specific type of attack, such as 'smurf', 'neptune', 'back', etc.

2. Y-axis (Frequency): This axis shows the number of occurrences of each label in the dataset.

3. Bars: Each bar represents the count of a specific label.

Observations from the Plot:

Dominant Labels:

- The labels 'smurf' and 'neptune' are the most frequent in the dataset, with 'smurf' being the most common, followed by 'neptune'.

- The ‘normal’ label is also quite frequent, indicating a significant number of normal connections in the dataset.

Rare Labels:

- Several labels, such as ‘pod’, ‘nmap’, ‘phf’, ‘spy’, etc., have very low frequencies, indicating that these types of connections are rare in the dataset.

```
# print label column
print(data['label'].value_counts())
```

```
label
smurf.              280790
neptune.            107201
normal.              97278
back.                 2203
satan.                1589
ipsweep.              1247
portsweep.            1040
warezclient.          1020
teardrop.              979
pod.                   264
nmap.                  231
guess_passwd.           53
buffer_overflow.        30
land.                   21
warezmaster.            20
imap.                   12
rootkit.                10
loadmodule.              9
ftp_write.               8
multihop.                7
phf.                     4
perl.                    3
spy.                     2
Name: count, dtype: int64
```

Data Preprocessing

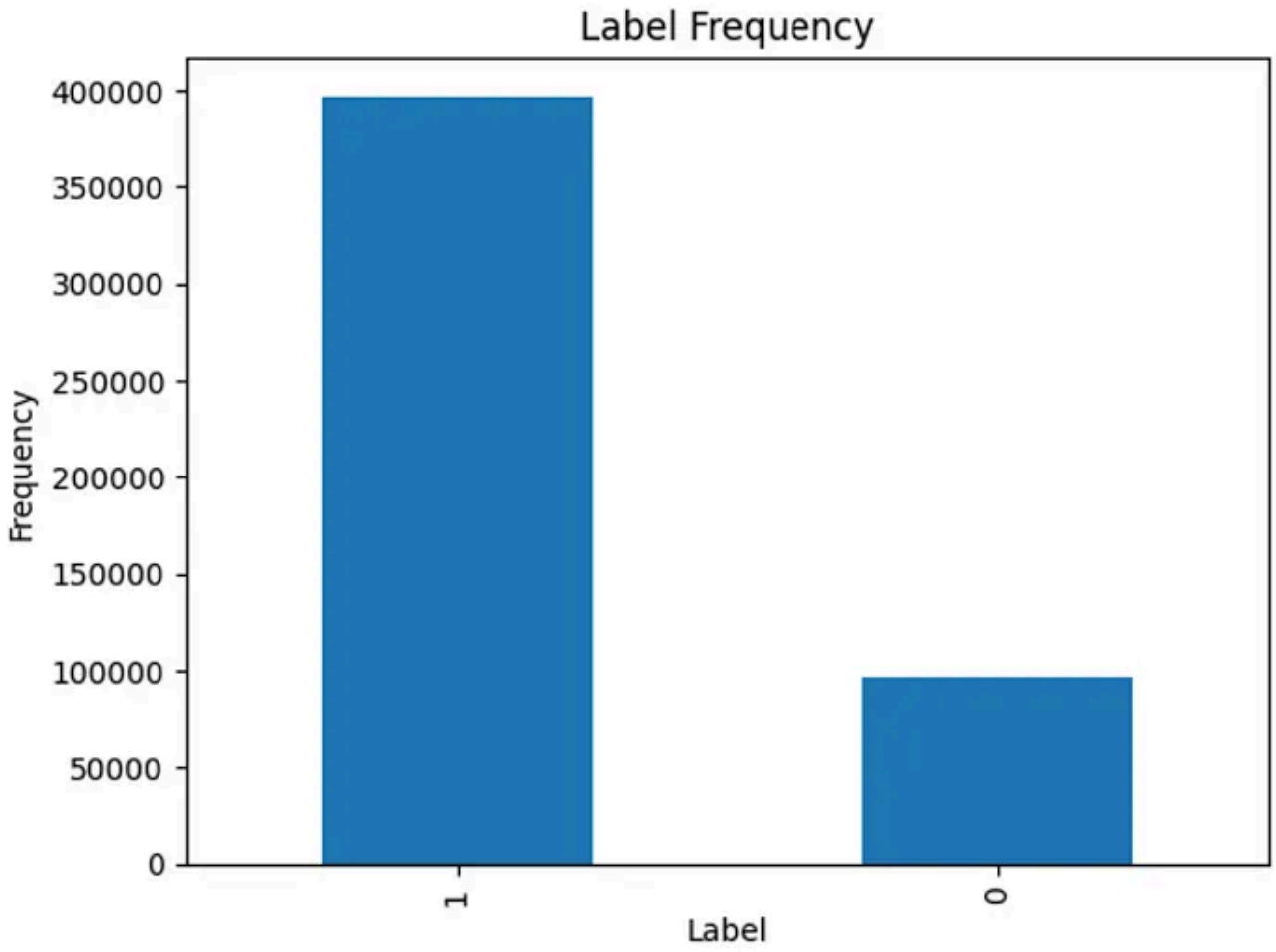
Before applying the autoencoder, the dataset must be preprocessed:

- Feature Selection: Select relevant features for the anomaly detection task.

Anything but normal is an anomaly.

```
# Convert all "normal" labels to a value "0", while all other labels to a value

data['label'] = data['label'].apply(lambda x: 0 if x == 'normal.' else 1)
```

```
# the "label" column in the data df should contain the two labels "0" and "1" r
plot_value_count(data)
```



Splitting: Divide the dataset into training and test sets, ensuring that the training set contains only normal data.

```
from sklearn.model_selection import train_test_split
```

```
# Separate the labels from the features
X = data.drop("label", axis=1)
y = data["label"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random
```

Note: We split the data before one-hot-encoding to avoid leakage

Encoding: Convert categorical features to numerical format, if any.

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wror
0	0	tcp	http	SF	181	5450	0	
1	0	tcp	http	SF	239	486	0	
2	0	tcp	http	SF	235	1337	0	
3	0	tcp	http	SF	219	1337	0	
4	0	tcp	http	SF	217	2032	0	
5	0	tcp	http	SF	217	2032	0	
6	0	tcp	http	SF	212	1940	0	
7	0	tcp	http	SF	159	4087	0	
8	0	tcp	http	SF	210	151	0	
9	0	tcp	http	SF	212	786	0	

```
categorical_features = ["protocol_type", "service", "flag"]
```

```
# Now convert the categorical features in One Hot representation. The code shou

from sklearn.preprocessing import OneHotEncoder

# Initialize the OneHotEncoder
encoder = OneHotEncoder(handle_unknown='ignore')

# Fit and transform the categorical features for training data
X_train_encoded = encoder.fit_transform(X_train[categorical_features])

# Transform the categorical features for test data
X_test_encoded = encoder.transform(X_test[categorical_features])

# Create DataFrames with the one-hot encoded features
```

```
X_train_encoded_df = pd.DataFrame(X_train_encoded.toarray(), columns=encoder.ge
X_test_encoded_df = pd.DataFrame(X_test_encoded.toarray(), columns=encoder.get_

# Drop the original categorical features
X_train_dropped = X_train.drop(categorical_features, axis=1)
X_test_dropped = X_test.drop(categorical_features, axis=1)

# Concatenate the one-hot encoded features with the original datasets
X_train_final = pd.concat([X_train_dropped, X_train_encoded_df], axis=1)
X_test_final = pd.concat([X_test_dropped, X_test_encoded_df], axis=1)
```

```
X_train_encoded.shape
✓ 0.0s
(395216, 79)
```

> *Note: Three columns were encoded into 79*

```
X_train_encoded_df.head(10)
✓ 0.0s
```

	protocol_type_icmp	protocol_type_tcp	protocol_type_udp	service_IRC	service_X11	service_Z39_50	service_au
0	1.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	1.0	0.0	0.0	0.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	0.0	0.0	
4	0.0	1.0	0.0	0.0	0.0	0.0	
5	1.0	0.0	0.0	0.0	0.0	0.0	
6	0.0	0.0	1.0	0.0	0.0	0.0	
7	0.0	1.0	0.0	0.0	0.0	0.0	
8	0.0	1.0	0.0	0.0	0.0	0.0	
9	0.0	1.0	0.0	0.0	0.0	0.0	

```
print(X_train_final.shape)
print(X_test_final.shape)
✓ 0.0s
(395216, 117)
(98805, 117)
```

As you can see the number of features increased because of the encoding

> *Let me explain the OneHotEncoder (Skip if you are already familiar)*
>
> *One-hot encoding is a technique used to convert categorical data into a numerical format that can be used by machine learning algorithms. The idea is to create a binary column for each category in a categorical feature, where only one of these columns can take the value 1 (indicating the presence of that category) while the rest will be 0.*
>
> *Example:*
>
> *Suppose you have a dataset with a categorical feature "Color" that has three possible values: "Red", "Green", and "Blue".*
>
> *Original Data:*

Color
Red
Blue
Green
Red
Blue

> *After applying one-hot encoding, this feature is transformed into three new binary features, one for each category:*

Color_Red	Color_Green	Color_Blue
1	0	0
0	0	1
0	1	0
1	0	0
0	0	1

Using `OneHotEncoder` from scikit-learn:

```
from sklearn.preprocessing import OneHotEncoder

# Sample data
data = [['Red'], ['Blue'], ['Green'], ['Red'], ['Blue']]
# Initialize the OneHotEncoder
encoder = OneHotEncoder(sparse=False)  # sparse=False to get a 2D array instead
# Fit and transform the data
encoded_data = encoder.fit_transform(data)
# Print the encoded data
print(encoded_data)
```

Output:

```
[[1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]]
```

> *In this example, the `OneHotEncoder` has created three new binary features corresponding to the three colors. Each row in the encoded data represents one of the original data points, with a 1 in the column corresponding to its color and 0s in the other columns. You can notice that one feature/Column (Color) encoded into three features/Columns (Color_Red, Color_Green, Color_Blue). This is a simple example to what happened in the previous case where three columns were encoded into 79*
>
> *Note 2: Ensure that the one_hot_encoder is fit on the train only, not the test, to avoid leakage.*
>
> *Note 3: CatBoost Models can do that automatically as part of the training https://catboost.ai/. CatBoost is a machine learning algorithm developed by Yandex that is based on gradient boosting over decision trees. CatBoost stands for "Category Boosting" and is designed to handle categorical features naturally and efficiently without requiring extensive preprocessing or one-hot encoding.*

=======

Normalization: Scale numerical features to a common scale.

```
continuous_features = [x for x in column_names if x not in categorical_features
print('Total number of non-categorical features: ', len(continuous_features))

# Total number of non-categorical features:  38
```

```
# Normalize the non categorical features in the dataset. The code should update

from sklearn.preprocessing import MinMaxScaler

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Reset the index of X_train and X_test
X_train.reset_index(drop=True, inplace=True)
X_test.reset_index(drop=True, inplace=True)

# Fit and transform the continuous features in X_train
X_train_continuous = X_train[continuous_features]
X_train_normalized = scaler.fit_transform(X_train_continuous)
X_train_final = pd.DataFrame(data=X_train_normalized, columns=continuous_featur

# Transform the continuous features in X_test
X_test_continuous = X_test[continuous_features]
X_test_normalized = scaler.transform(X_test_continuous)
X_test_final = pd.DataFrame(data=X_test_normalized, columns=continuous_features

# Display the normalized datasets
print("Normalized X_train:")
print(X_train_final.head())
print("\nNormalized X_test:")
print(X_test_final.head())
```

```
Normalized X_train:
   duration  src_bytes  dst_bytes  land  wrong_fragment  urgent  hot  \
0       0.0   0.000001        0.0   0.0             0.0     0.0  0.0
1       0.0   0.000000        0.0   0.0             0.0     0.0  0.0
```

```
Normalized X_test:
   duration     src_bytes  dst_bytes  land  wrong_fragment  urgent  hot  \
0       0.0  1.488371e-06   0.000000   0.0             0.0     0.0  0.0
1       0.0  1.488371e-06   0.000000   0.0             0.0     0.0  0.0
```

Building the Autoencoder

A simple autoencoder architecture for this task might consist of an input layer, one or more hidden layers for the encoder and decoder, and an output layer with the same dimensions as the input layer.

Encoder

We will use the Model class from Keras. The `Model` class in TensorFlow's Keras library is a central concept used to define and work with neural networks. It is a subclass of `Layer`, which is the basic building block in Keras. The `Model` class is used to create a graph of layers, representing the architecture of a neural network. It provides methods for training, evaluating, and making predictions with the network. Here is how to build the encoder using the Model class:

Define the Model Architecture:

You create an instance of the `Model` class by specifying its input and output layers. The layers between the input and output define the architecture of the neural network.

```
# Initialize the input dimension
input_dim = X_train_final.shape[1]  # Number of features

print(input_dim)
```

```
input_dim = X_train_final.shape[1]  # Number of features

print(input_dim)
```

✓ 0.0s

```
117
```

```
# Define the input shape
input_shape = (input_dim,)  # input_dim is the number of dimensions in the trai

print(input_shape)
```

```
# Define the input shape
input_shape = (input_dim,)  # input_dim is the number of dimensions in the training dataset

print(input_shape)
```

✓ 0.0s

```
(117,)
```

Define the Input Layer:

```
from keras.layers import Input

# define the input layer
input_layer = Input(shape=input_shape)

print(input_layer)
```

```
KerasTensor(
  type_spec=TensorSpec(shape=(None, 117),
  dtype=tf.float32, name='input_1'),
  name='input_1',
  description="created by layer 'input_1'")
```

The previous code snippet demonstrates how to define the input layer for the encoder using Keras. Here's a breakdown of each line:

1. `from keras.layers import Input` : This line imports the `Input` class from the `keras.layers` module. The `Input` class is used to instantiate a Keras tensor, which is a symbolic representation of the input to a neural network, the encoder in this case.

2. `input_layer = Input(shape=input_shape)` : This line creates an input layer for the neural network. The `shape` argument specifies the shape of the input data, which is a tuple indicating the dimensions of the input. (117,)

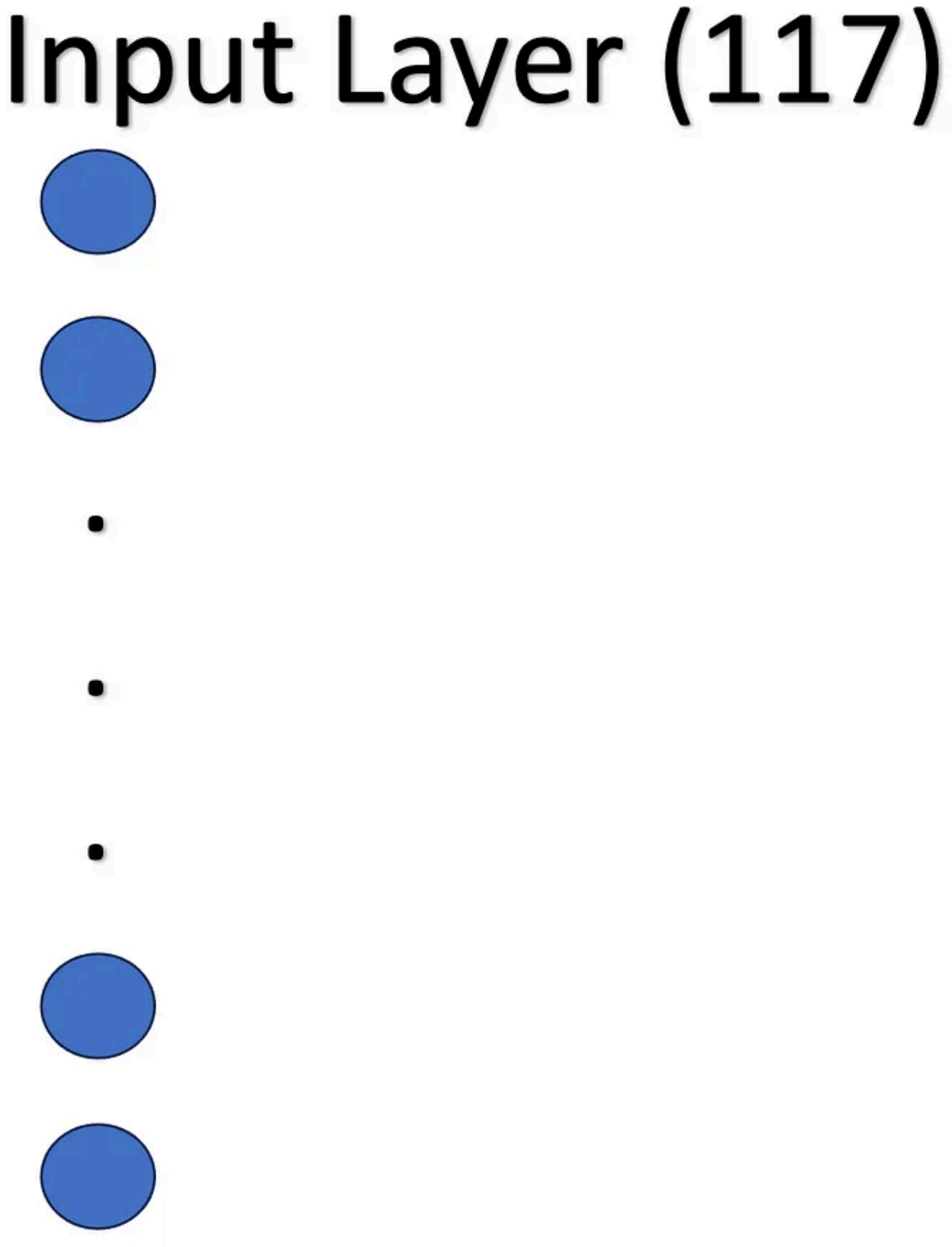
3. `print(input_layer)` : This line prints the created input layer to the console. The output will be a symbolic tensor that represents the input to the neural network. It will include information about the shape of the input and the data type (type).

The result is creating an input layer

input_2	input:	[(None, 117)]
InputLayer	output:	[(None, 117)]

The input layer is designed to accept input samples with 117 features each, which can be thought of as 117 neurons if you visualize the input in that way.

- input: [(None, 117)]: This indicates the shape of the input. The `None` value represents the batch size, which can vary, and `117` is the dimensionality of each input sample. So, each input sample has 117 features or neurons.
- InputLayer: This is the type of layer, which is an input layer.
- output: [(None, 117)]: This indicates the shape of the output of the input layer. Since the input layer is simply passing the input data to the next layer without any transformations, the output shape is the same as the input shape.

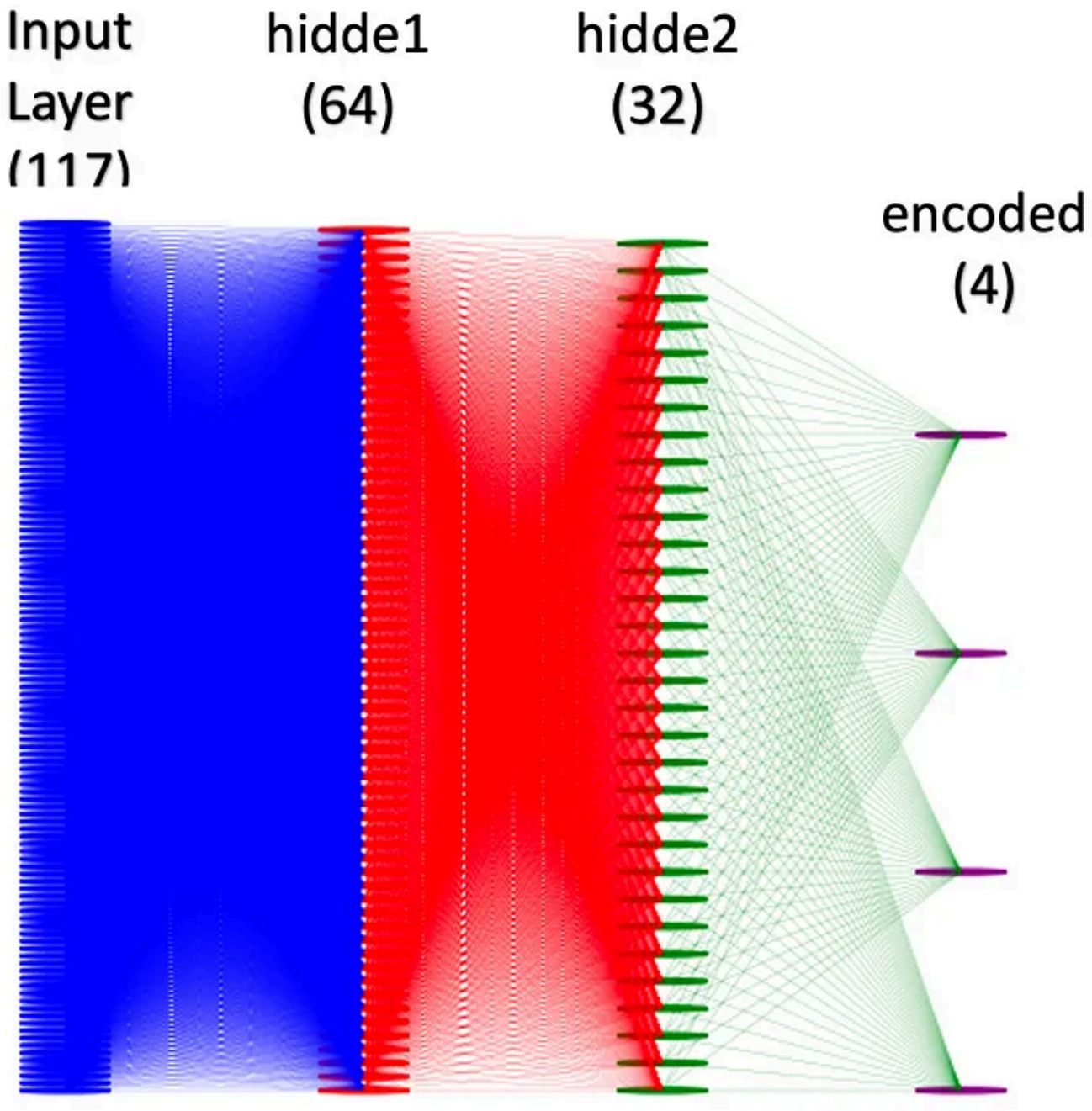


Define the Hidden layers:

We will select arbitrary number of neurons for each layer. [64, 32] and we will reduce the encoder to only 4 Neurons.

```
from tensorflow.keras.layers import Dense
```

```
# Define the Encoder Layers
hidden_layer_1 = Dense(64, activation='relu')(input_layer)
hidden_layer_2 = Dense(32, activation='relu')(hidden_layer_1)
encoded_representation = Dense(4, activation='relu')(hidden_layer_2)
```



The code above defines the encoder part of the autoencoder using Keras. In summary, this code defines an encoder with three layers, reducing the dimensionality of the input data to a 4-dimensional encoded representation. Here's a breakdown of the code:

Import the Dense Layer:

```
from tensorflow.keras.layers import Dense
```

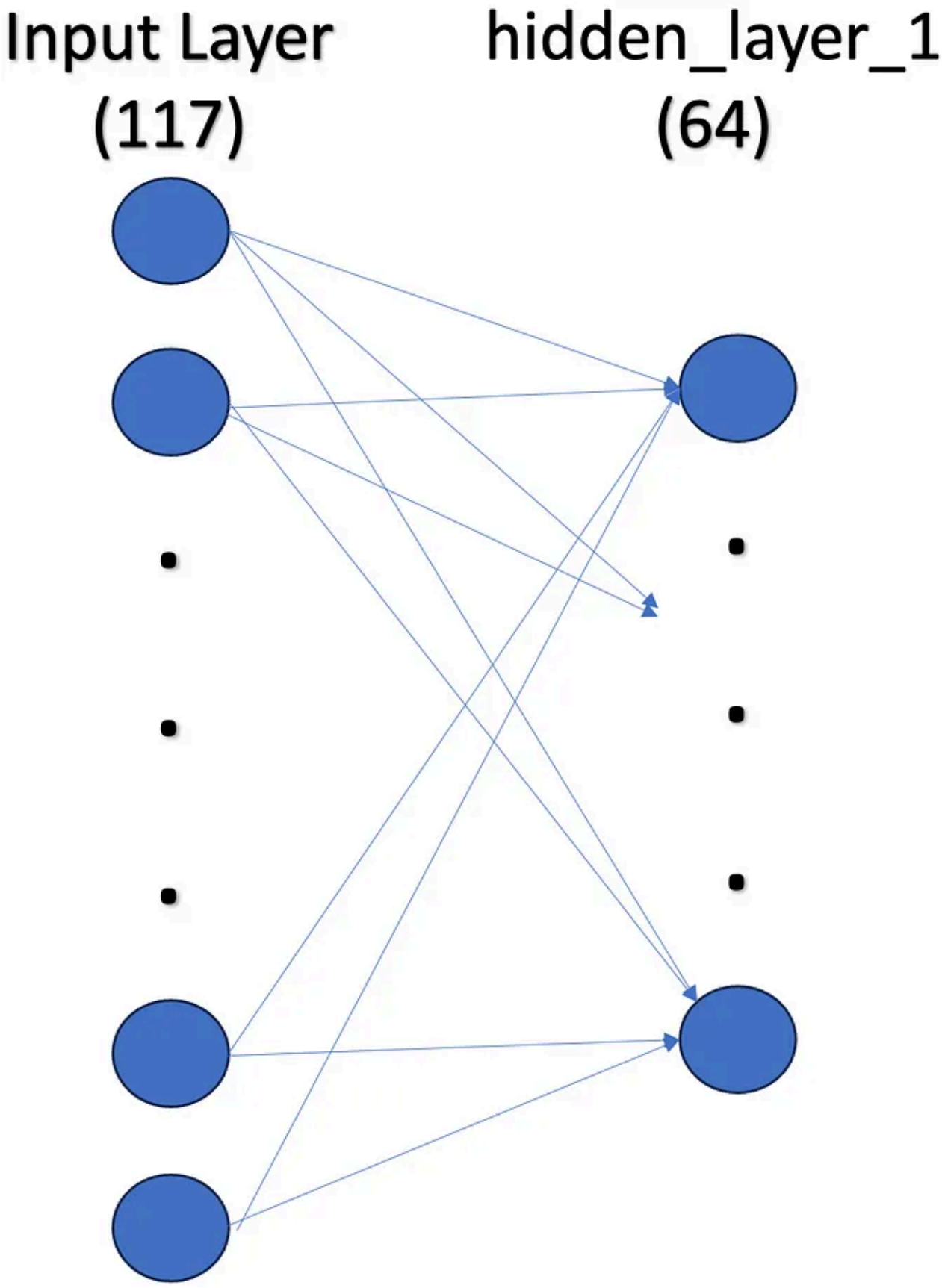
This line imports the `Dense` class from Keras. A `Dense` layer is a fully connected neural network layer, where each neuron in the layer is connected to all the neurons in the previous layer.

Define the Encoder Layers:

- First Hidden Layer (64):

```
hidden_layer_1 = Dense(64, activation='relu')(input_layer)
```

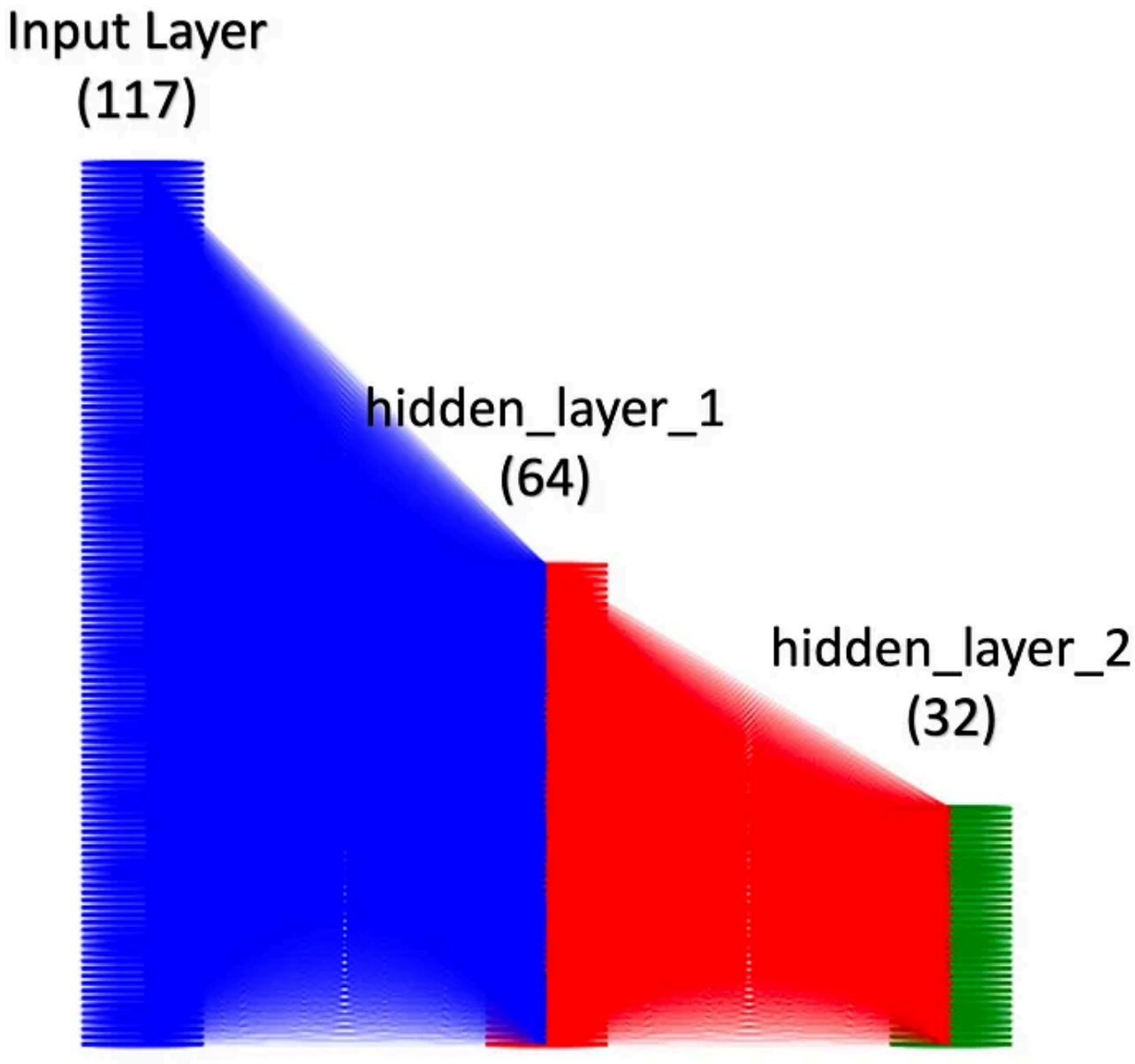
- This line creates the first hidden layer of the encoder with 64 neurons. The `activation='relu'` argument specifies that the Rectified Linear Unit (ReLU) activation function should be used for the neurons in this layer. The `input_layer` represents the input to the encoder.



Second Hidden Layer:

```
hidden_layer_2 = Dense(32, activation='relu')(hidden_layer_1)
```

- This line creates the second hidden layer of the encoder with 32 neurons. It takes the output of the first hidden layer (`hidden_layer_1`) as its input.



Encoded Representation:

```
encoded_representation = Dense(4, activation='relu')(hidden_layer_2)
```

- This line creates the final layer of the encoder, which produces the encoded representation of the input data. It has 4 neurons and uses the ReLU activation function. The output of this layer is a 4-dimensional encoded representation of the input data.

Create the encoder model

```
from tensorflow.keras.models import Model

# Create the encoder model
encoder = Model(input_layer, encoded_representation)
```

```
# Print Summary of the encoder encoder Model
encoder.summary()
```

✓ 0.0s

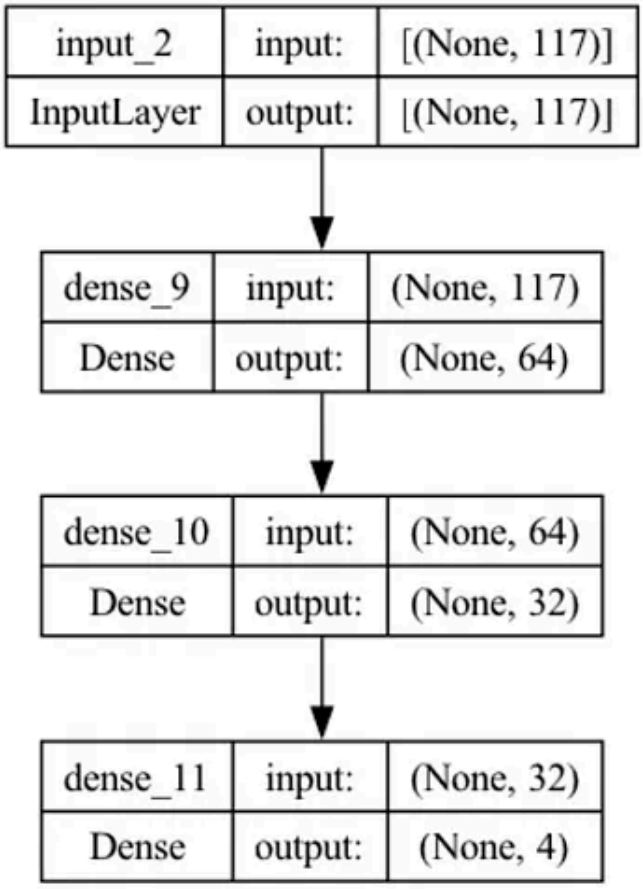
```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 117)]             0

 dense_9 (Dense)             (None, 64)                7552

 dense_10 (Dense)            (None, 32)                2080

 dense_11 (Dense)            (None, 4)                 132

=================================================================
Total params: 9764 (38.14 KB)
Trainable params: 9764 (38.14 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Plot the Encoder model:

```
from tensorflow.keras.utils import plot_model

# Plot the model
plot_model(encoder, to_file='encoder.png', show_shapes=True)
```

```
from tensorflow.keras.utils import plot_model

# Plot the model
plot_model(encoder, to_file='encoder.png', show_shapes=True, show_layer_names=True)
```

✓ 1.0s



Building the Decoder

```
# Define the layers of the Decoder
encoded_input = Input(shape=(4,))
decoded = Dense(32, activation='relu')(encoded_input)
decoded = Dense(64, activation='relu')(decoded)
decoded = Dense(input_dim, activation='sigmoid')(decoded)

# Initialize the Decoder Model using the above created architecture
decoder = Model(encoded_input, decoded)
```

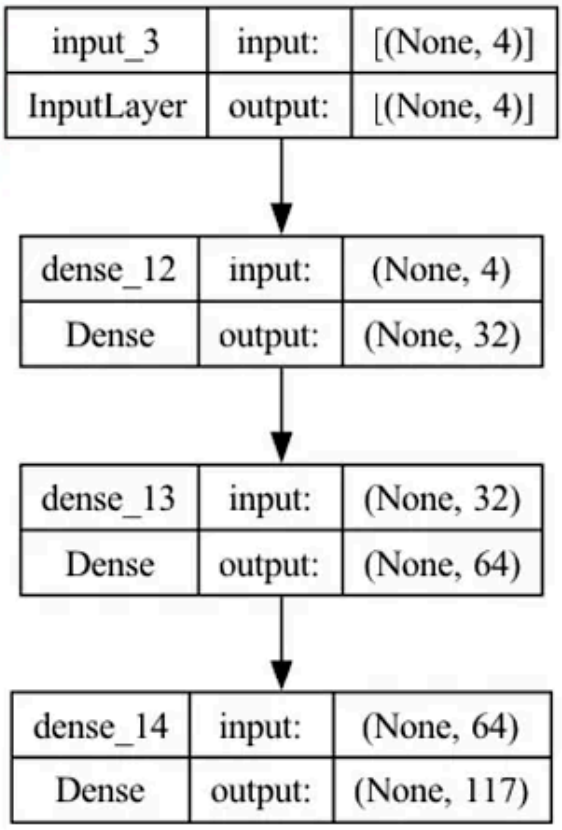
```
# Print decoder summary
decoder.summary()
```

✓ 0.0s

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 4)]               0

 dense_12 (Dense)            (None, 32)                160

 dense_13 (Dense)            (None, 64)                2112

 dense_14 (Dense)            (None, 117)               7605

=================================================================
Total params: 9877 (38.58 KB)
Trainable params: 9877 (38.58 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
plot_model(decoder, to_file='autoencoder.png', show_shapes=True, show_layer_names=True)
```

✓ 1.1s



The code above defines a decoder that takes a 4-dimensional encoded representation as input and reconstructs the original data through two hidden layers and an output layer. The architecture of the decoder is typically chosen to mirror the encoder, with the number of neurons in each layer increasing symmetrically to the decrease in the encoder. Here's a breakdown of the code:

Define the Input to the Decoder:

```
encoded_input = Input(shape=(4,))
```

This line creates an input layer for the decoder. The shape `(4,)` corresponds to the size of the encoded representation produced by the encoder, which is a 4-dimensional vector in this case.

Define the Layers of the Decoder:

First Layer:

```
decoded = Dense(32, activation='relu')(encoded_input)
```

This line creates the first layer of the decoder with 32 neurons and uses the ReLU activation function. It takes the encoded input as its input.

Second Layer:

```
decoded = Dense(64, activation='relu')(decoded)
```

This line creates the second layer of the decoder with 64 neurons, again using the ReLU activation function. It takes the output of the first layer as its input.

Output Layer:

```
decoded = Dense(input_dim, activation='sigmoid')(decoded)
```

This line creates the output layer of the decoder with a number of neurons equal to the original input dimension (`input_dim`). The `sigmoid` activation function is used, which is common for binary input data or data normalized between 0 and 1, since we are classifying Normal or Anamoly (0 or 1).

Initialize the Decoder Model:

```
decoder = Model(encoded_input, decoded)
```

This line creates a `Model` instance representing the decoder. It specifies that the input to the model is `encoded_input` , and the output is the result of the final `Dense` layer (`decoded`).

Combine the Encoder and the Decoder

```
# Combine encoder and decoder models to create the Autoencoder model
autoencoder = Model(input_layer, decoder(encoder(input_layer)))
```

```
autoencoder.summary()
```

✓ 0.0s

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 117)]             0

 model_1 (Functional)        (None, 4)                 9764

 model_2 (Functional)        (None, 117)               9877

=================================================================
Total params: 19641 (76.72 KB)
Trainable params: 19641 (76.72 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

> *Note: If we did not want to separate the Encoder from the Decoder as in this case, you can build the previous AutoEncoder using Sequential as follows:*

```
from keras.models import Sequential
from keras.layers import Dense

input_dim = X_train_final.shape[1]  # Number of features
# Define the autoencoder architecture
autoencoder = Sequential([
    Dense(64, activation='relu', input_shape=(input_dim,)),
    Dense(32, activation='relu'),
    Dense(4, activation='relu'),  # Encoder
    Dense(32, activation='relu'),
    Dense(64, activation='relu'),
    Dense(input_dim, activation='sigmoid')  # Output layer
])
```

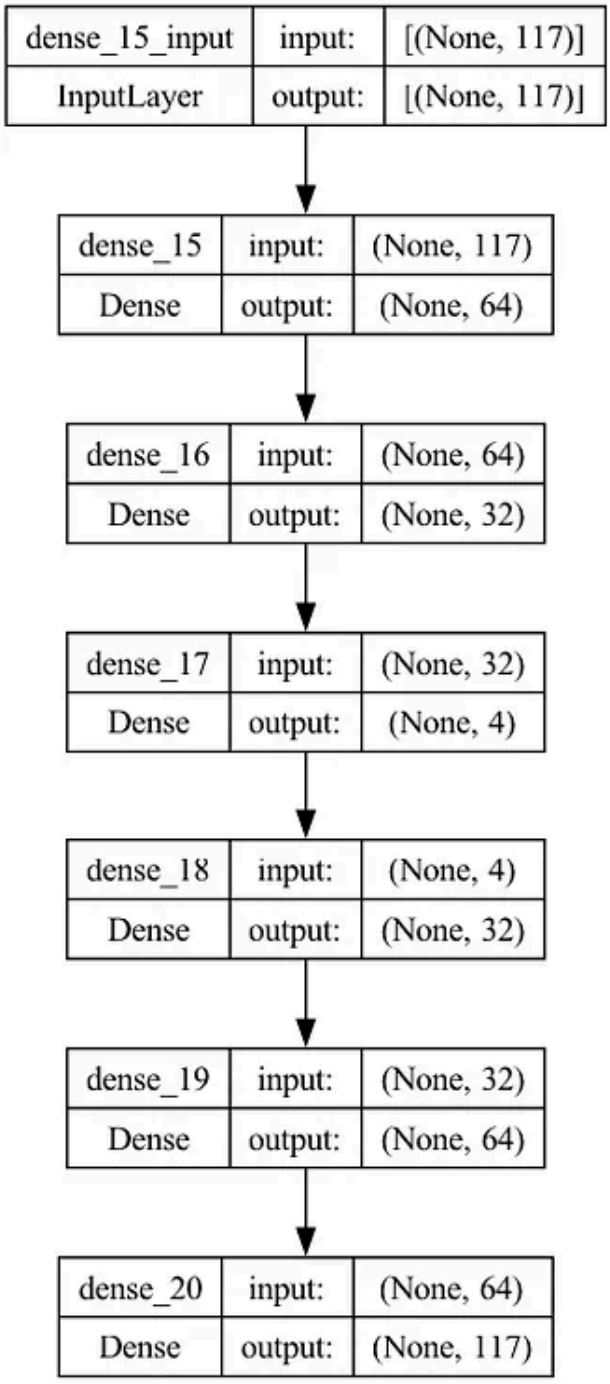
```
autoencoder.summary()
```

✓ 0.0s

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_15 (Dense)            (None, 64)                7552

 dense_16 (Dense)            (None, 32)                2080

 dense_17 (Dense)            (None, 4)                 132

 dense_18 (Dense)            (None, 32)                160

 dense_19 (Dense)            (None, 64)                2112

 dense_20 (Dense)            (None, 117)               7605

=================================================================
Total params: 19641 (76.72 KB)
Trainable params: 19641 (76.72 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
plot_model(autoencoder, to_file='autoencoder.png', show_shapes=True, show_layer_names=True)
```

✓ 0.2s



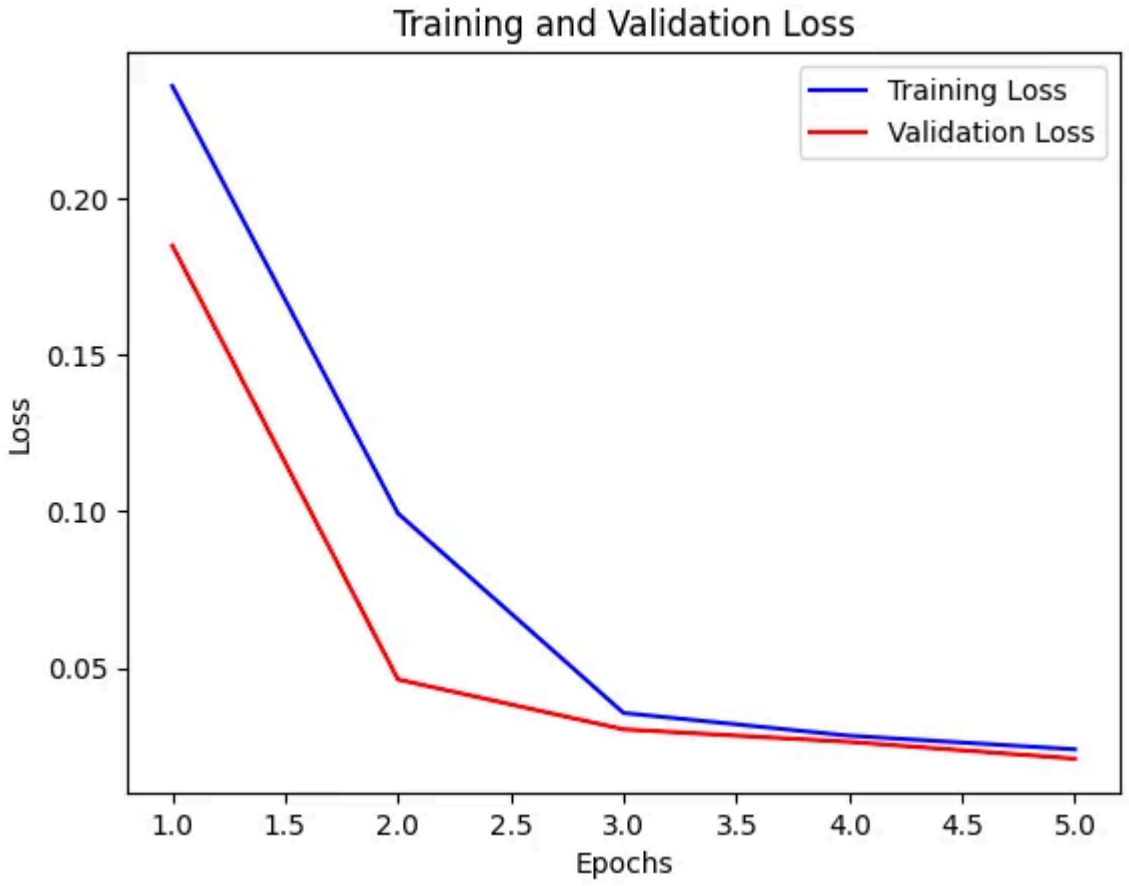
Training the Autoencoder

The autoencoder is trained on the normal data using a reconstruction loss function, such as mean squared error (MSE), to minimize the difference between the input and the reconstructed output.

```
# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='mse')
```

```
# Train the autoencoder
autoencoder.fit(X_train, X_train, epochs=5, batch_size=256, shuffle=True, valic
```

```
Epoch 1/5
1544/1544 [==============================] - 3s 1ms/step - loss: 0.2267 - val_l
Epoch 2/5
1544/1544 [==============================] - 2s 1ms/step - loss: 0.0989 - val_l
Epoch 3/5
1544/1544 [==============================] - 2s 1ms/step - loss: 0.0349 - val_l
Epoch 4/5
1544/1544 [==============================] - 2s 1ms/step - loss: 0.0229 - val_l
Epoch 5/5
1544/1544 [==============================] - 2s 1ms/step - loss: 0.0148 - val_l
```



> *Note that we reduced the features/dimensions from 117 to 4*

```
autoencoder.fit(X_train, X_train ...
```

> *Note: We use X-train as target and input because this is the autoencoder where we encode the X-train and then decode it again and compare it with itself to make sure the decoding is accurate and use the error function against that.*

```
# Obtain the encoded features
encoded_train_features = encoder.predict(X_train)
encoded_test_features = encoder.predict(X_test)
```


```
encoded_train_features = encoder.predict(X_train)
```

- `encoder` : This is the trained encoder part of the previous autoencoder model.
- `.predict(X_train)` : The `predict` method is used to generate predictions from the model. In this case, it takes `X_train` , the training data, and passes it through the encoder to obtain the encoded representations.
- `encoded_train_features` : This variable stores the encoded (compressed) features of the training data.

In an autoencoder, the encoder part is responsible for compressing the input data into a lower-dimensional representation. This lower-dimensional representation captures the most important features of the input data in a compressed form (similar to PCA).

Anomaly Detection

Once the autoencoder is trained, it can be used to detect anomalies. For each data point in the test set, the reconstruction error is calculated. Data points with a reconstruction error above a certain threshold are flagged as anomalies.

```
reconstruction_error = np.mean(np.power(X_test - autoencoder.predict(X_test), 2
threshold = np.percentile(reconstruction_error, 95)  # Set threshold as the 95t
anomalies = reconstruction_error > threshold
```

Conclusion

Autoencoders offer a powerful approach to anomaly detection by learning a compressed representation of normal data and identifying deviations from this norm. The KDD Cup 1999 dataset provides a practical case study to apply and evaluate the effectiveness of autoencoders in a real-world anomaly detection scenario. By fine-tuning the autoencoder architecture and the anomaly threshold, one can achieve a balance between sensitivity and specificity in detecting anomalies.

References

1. KDD Cup 1999 Dataset: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
2. Autoencoders for Anomaly Detection: A Comprehensive Guide. *Journal of Machine

Artificial Intelligence | Machine Learning | AI | Autoencoder | Anomaly Detection

Follow

Written by Rany ElHousieny

667 Followers · 34 Following

https://www.linkedin.com/in/ranyelhousieny Software/AI/ML/Data engineering manager with extensive experience in technical management, AI/ML, AI Solutions Archit

No responses yet

What are your thoughts?

Respond

More from Rany ElHousieny and Level Up Coding

Rany ElHousieny

Introduction to Strings in Python

In Python, strings are used for representing textual data. A string is a sequence of characters enclosed in either single quotes (‘) or…

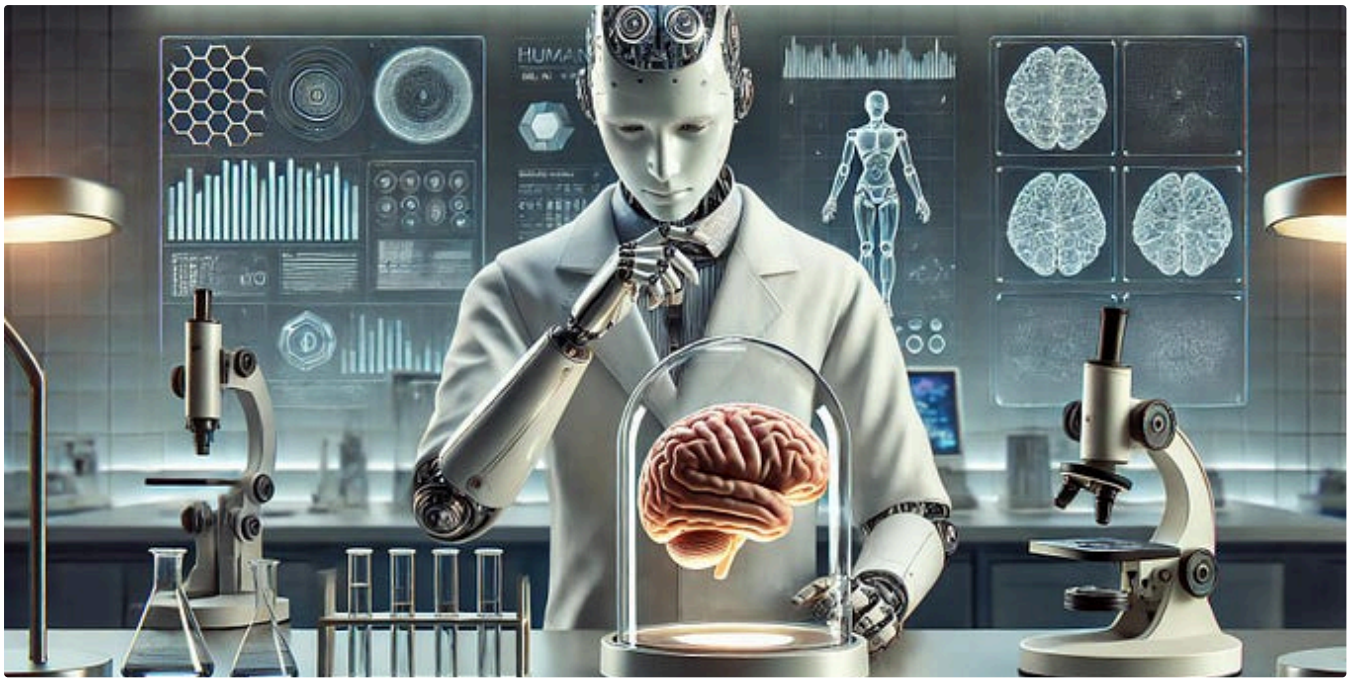
Aug 10, 2023 5 1

In Level Up Coding by Ian Mundy

Effective Technical Leads

Lessons learned from building teams from the ground up

Nov 21 212 5



In Level Up Coding by Salvatore Raieli

The Cybernetic Neuroscientist: Smarter Than Experts?

Exploring How AI Outperforms Human Expertise in Predicting Neuroscience Breakthroughs

1d ago 561 9

Rany ElHousieny

Ensemble Methods in Machine Learning

Ensemble methods are a cornerstone of modern machine learning, offering robust techniques to improve model performance by combining...

May 30 43

See all from Rany ElHousieny

See all from Level Up Coding

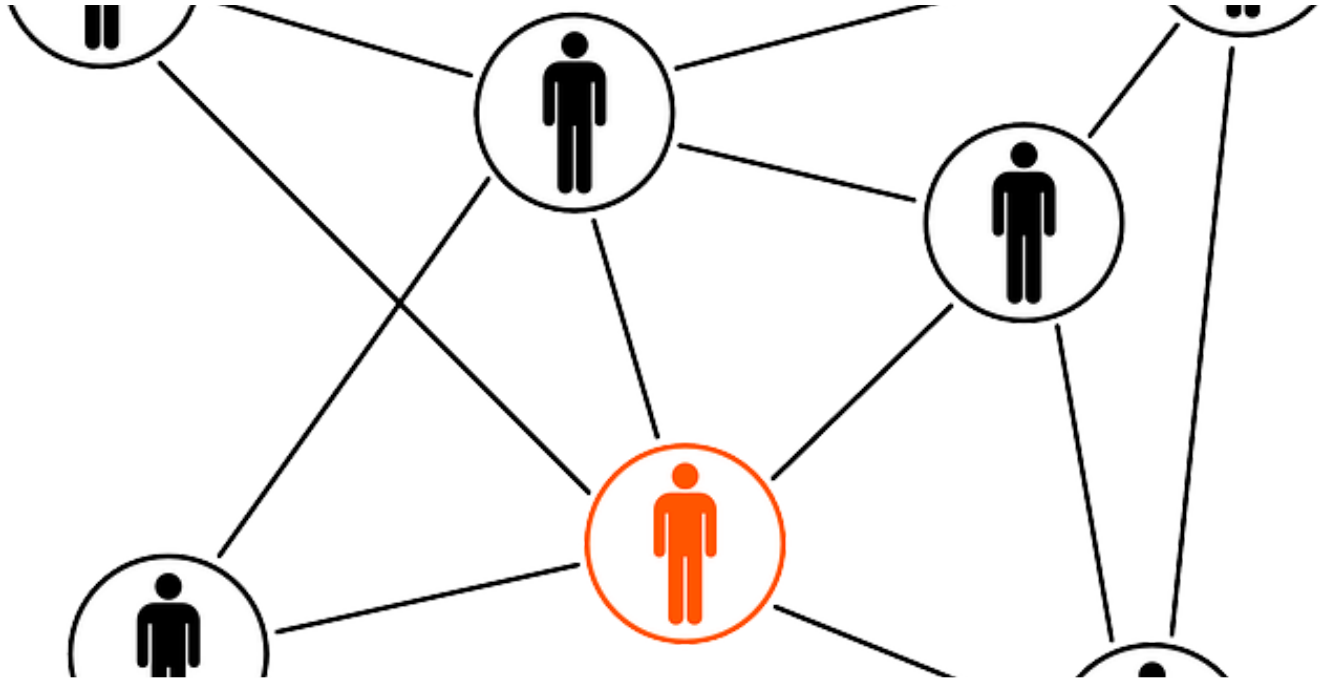
Recommended from Medium

TechwithJulles

Convolutional Neural Networks (CNNs) for Time Series Data—Training and Evaluating the CNN Model...

Introduction

Oct 17 10



In Towards Data Science by Claudia Ng

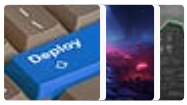
How to Build a Graph-based Neural Network for Anomaly Detection in 6 Steps

Learn to build a Graph Convolutional Network that can handle heterogeneous graph data for link prediction

Feb 13 573

Lists

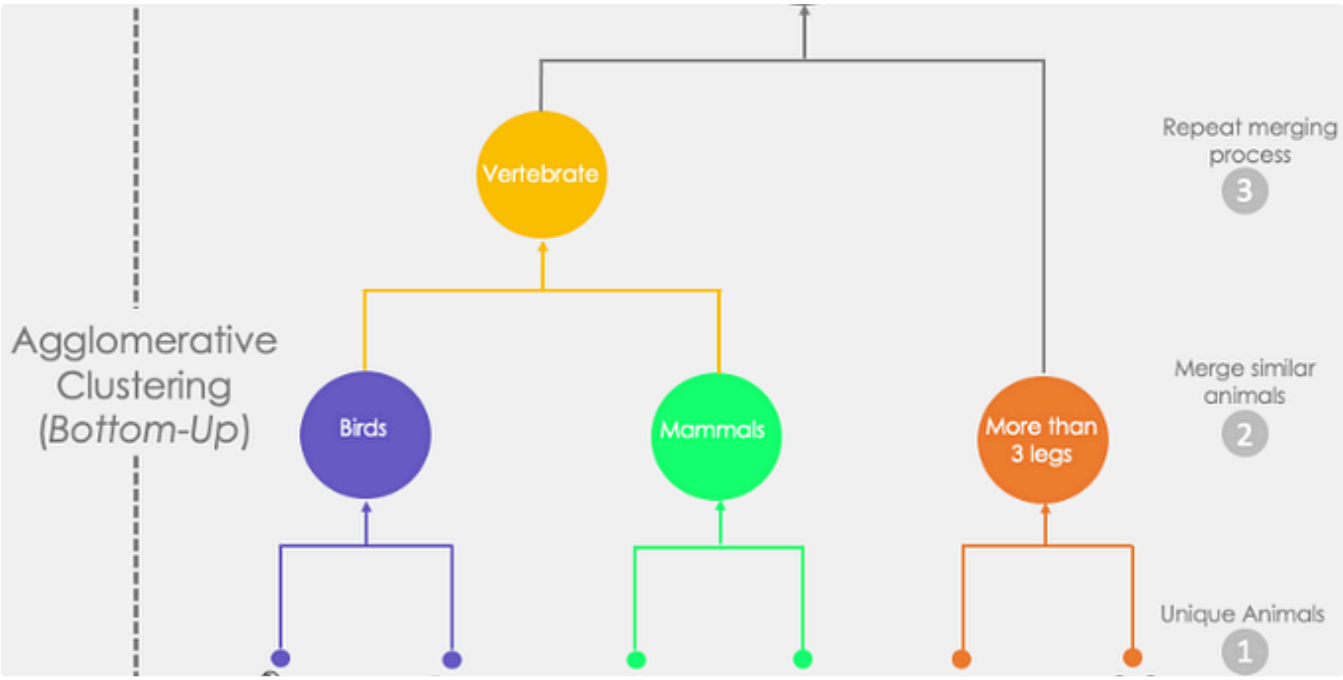
Natural Language Processing
1842 stories · 1466 saves



Predictive Modeling w/ Python
20 stories · 1700 saves

AI Regulation
6 stories · 639 saves

Generative AI Recommended Reading
52 stories · 1532 saves



In GoPenAI by Nermin Babalık

Unsupervised Learning—A Comprehensive Guide

Unsupervised learning methods like K-Means, hierarchical clustering, and PCA are crucial tools for data analysis, enabling better...

Nov 24 251

In AI Advances by Turibio Hilaire

200X Faster: Speed Up Your Music Production with AI

Boost Your Workflow and Create Music Faster Than Ever with The Help of AI

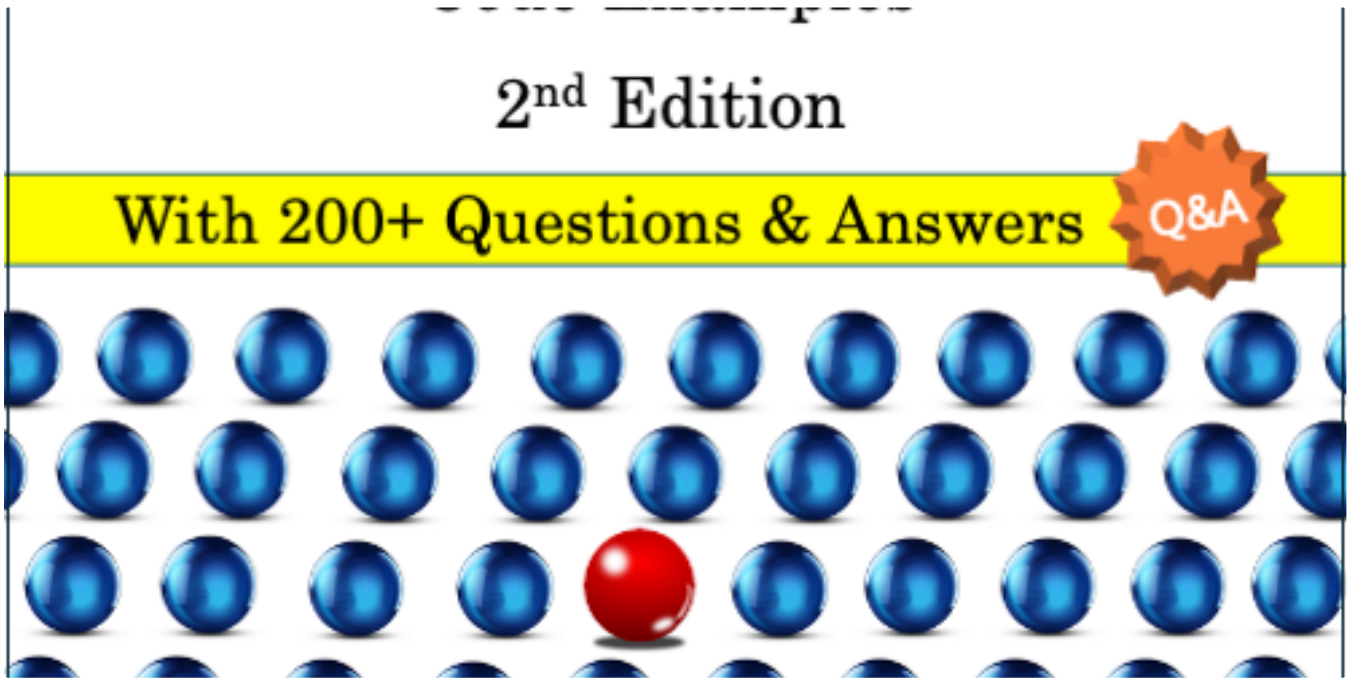
3d ago 131 1

In Towards AI by Shenggang Li

Dynamic Weight Models: Bridging GLM and Neural Networks

Introduction

4d ago 229 5



In Dataman in AI by Chris Kuo/Dr. Dataman

Handbook of Anomaly Detection—(3) ECOD

In the previous chapter, we have introduced HBOS (Histogram-Based Outlier Score). This chapter introduces Empirical Cumulative...

Aug 24 6