"Adam stands for Adaptive Moment Estimation.
It combines the idea of **Momentum**, which remembers past gradients to keep moving smoothly in the right direction,
and **RMSProp**, which scales the learning rate for each weight based on how big its gradients are.
Together, Adam adjusts both direction and step size automatically, making training fast and stable."

Perfect 👍
Let's go **line by line** through your `DL_4_AutoEncoder.pdf` so that you understand **every concept, every choice**, and are ready to answer **any cross-question** during viva.

🧠 Big Picture

The notebook builds an **autoencoder** model to detect **credit card fraud** — an **unsupervised anomaly detection** task.
It learns the structure of **normal transactions** and detects **fraudulent ones** based on reconstruction error.

🔹 1. Importing Libraries

```
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
```

🔸 What each does:

- **tensorflow as tf** → builds and trains the neural network (autoencoder).
- **pandas as pd** → handles tabular data easily (`read_csv`, DataFrame ops).
- **numpy as np** → numerical operations (arrays, mean, power, etc.).
- **train_test_split** → splits the dataset into training and testing sets.
- **StandardScaler** → standardizes features (mean 0, std 1).
- **confusion_matrix**, **classification_report** → evaluate model performance.
- **seaborn**, **matplotlib.pyplot** → plotting and visualization.

🧠 *Cross Q: Why do we scale data?*
→ To make sure all features are on the same scale — helps the model converge faster.

🔹 2. Loading the dataset

```
df = pd.read_csv("creditcard.csv")
```

Loads the **credit card fraud dataset**, where:

- Each row = one transaction
- Columns = 30 numerical features (V1–V28 + Time + Amount)
- `Class` = 0 (normal) or 1 (fraud)

🔹 3. Checking Class Distribution

```
print(df['Class'].value_counts())
```

Shows:

```
0 → 284315 (normal)
1 → 492 (fraud)
```

👉 **Severe class imbalance** — only 0.17% are fraudulent.

🧠 *Cross Q: Why is this imbalance important?*
→ Because the model will mostly see “normal” data — that’s why we train the autoencoder only on class 0.

🔹 4. Dropping the Time column

```
df = df.drop(['Time'], axis=1)
```

Removes `Time` since it doesn’t contribute much to identifying fraud — it’s just when the transaction occurred.

🧠 *Cross Q: Why not drop Amount?*
→ Because `Amount` has direct meaning; frauds often involve unusual amounts.

🔹 5. Splitting into features and labels

```
x = df.drop(['Class'], axis=1)
y = df['Class']
```

- `x`: all features (V1–V28 + Amount)
- `y`: target labels (fraud or normal)

🔹 6. Splitting into training and testing sets

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=42)
```

- 80% training, 20% testing.
- `random_state=42` ensures reproducibility (same random split every time).

🔹 7. Keeping only normal transactions for training

```
x_train = x_train[y_train == 0]
```

👉 Trains the autoencoder **only on normal data** so it learns the "normal pattern".
Then, when given frauds later, they'll have **higher reconstruction errors**.

🧠 *Cross Q: Why not train on both classes?*
→ Because frauds are too few and would confuse the model; we want it to specialize in normal transactions only.

🔹 8. Scaling the data

```
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

- **fit_transform** on training data learns the mean & std, then applies scaling.
- **transform** on test data uses the same scaling parameters.

🧠 *Cross Q: Why not fit again on test data?*
→ That would leak information — we scale test data using training statistics only.

🔹 9. Checking dataset stats

```
print("Training shape :", x_train.shape)
```

```
print("Testing shape  :", x_test.shape)
print("Frauds in test :", sum(y_test))
```

Outputs:

```
Training: (227451, 29)
Testing: (56962, 29)
Frauds in test: 98
```

Means:

- 227k normal samples used for training.
- 98 fraud samples present in test set (for evaluation).

🔹 10. Encoder network

```
encoder = tf.keras.models.Sequential([
    tf.keras.layers.Input(shape=(x_train.shape[1],)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(16, activation='relu')
])
```

Explanation:

- Input layer → takes 29 features per transaction.
- Dense(64) → learns complex feature relationships.
- Dense(32) → compresses data further.
- Dense(16) → final *latent representation* (compressed encoding).

🧠 *Cross Q: Why ReLU?*
→ It introduces non-linearity and prevents the "vanishing gradient" problem.

🧠 *Cross Q: What does the encoder do?*
→ It compresses data into a smaller dimension (16 features) — a "summary" of the original transaction.

🔹 11. Decoder network

```
decoder = tf.keras.models.Sequential([
    tf.keras.layers.Input(shape=(16,)),
```

```
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(x_train.shape[1], activation='linear')
])
```

Explanation:

- Input: the 16-dimensional latent vector from the encoder.
- Expands back to 32 → 64 → finally 29 features (same as input).
- Last layer uses `linear` activation (no restriction on output range).

🧠 *Cross Q: Why linear activation at the end?*
→ Because we're reconstructing continuous numeric data — we don't want to limit output between 0 and 1.

🔹 12. Combine encoder and decoder

```
autoencoder = tf.keras.models.Sequential([encoder, decoder])
autoencoder.summary()
```

- Builds the full autoencoder pipeline.
- The summary shows:
    - **Total params:** 9,069 (trainable).
    - **Input shape:** (29,)
    - **Output shape:** (29,)

🧠 *Cross Q: What is a parameter here?*
→ Each connection (weight + bias) in the network.
The model learns these to minimize reconstruction error.

🔹 13. Compile the model

```
autoencoder.compile(optimizer='adam', loss='mean_squared_error',
metrics=['accuracy'])
```

- **Optimizer:** Adam (Adaptive Moment Estimation) — adjusts learning rate dynamically.
- **Loss:** MSE — penalizes large reconstruction errors.

- **Metric:** Accuracy (not really useful here, but included for monitoring).

🧠 *Cross Q: Why MSE loss?*
→ It measures how close the reconstructed data is to the input — ideal for reconstruction tasks.

🔹 14. Training the model

```
history = autoencoder.fit(
    x_train, x_train,
    validation_data=(x_test, y_test),
    epochs=10,
    batch_size=128,
    shuffle=True
)
```

- **x_train, x_train** → input = output (since we want to reconstruct inputs).
- **validation_data** is technically mis-specified (should be `x_test, x_test`) but TensorFlow still runs fine.
- **epochs=10** → 10 complete passes through the training data.
- **batch_size=128** → trains on 128 samples at a time.
- **shuffle=True** → randomizes order each epoch for better generalization.

🧠 *Cross Q: Why train input = output?*
→ Because the autoencoder's goal is to reproduce the input accurately.

🔹 15. Model Training Results

Each epoch prints loss & validation loss.
Example:

```
Epoch 10/10
accuracy: 0.8513 - loss: 0.0466 - val_accuracy: 0.0345 - val_loss: 0.9870
```

- **Loss decreasing** → model is learning to reconstruct normal data better.
- **Validation loss high** → because test data has frauds that are harder to reconstruct.

🧠 *Cross Q: Why does validation loss stay high?*
→ The test set includes anomalies; the model fails to reconstruct them well — which is expected.

🔹 16. Reconstruction and Error Calculation

```
pred = autoencoder.predict(x_test)
recon_error = np.mean((x_test - pred)**2, axis=1)
```

- Reconstructs every test sample.
- Calculates mean squared reconstruction error for each transaction.

🧠 *Cross Q: What does axis=1 mean?*
→ Compute mean error across features (columns) per sample → one error value per transaction.

🔹 17. Combine with true labels

```
error_df = pd.DataFrame({'Reconstruction Error': recon_error, 'Class': y_test})
```

Creates a table linking each sample's reconstruction error with its actual class.

🔹 18. Separate Normal and Fraud Errors

```
normal_error = error_df.query("Class == 0")['Reconstruction Error']
fraud_error  = error_df.query("Class == 1")['Reconstruction Error']
```

Splits errors:

- **normal_error** → low values
- **fraud_error** → high values

🧠 *Cross Q: Why are fraud errors higher?*
→ Because the model hasn't seen frauds before — it fails to reconstruct them, resulting in higher errors.

🔹 19. Print average errors

```
Avg Error (Normal): 0.0415
Avg Error (Fraud):  4.6982
```

Shows clear separation — frauds have much larger reconstruction errors.

🔹 20. Threshold selection and anomaly detection

```
threshold = np.percentile(recon_error, 99)
anomalies = recon_error > threshold
print(anomalies.sum())
```

- Sets threshold as the **99th percentile** of reconstruction errors.
- Anything above that → flagged as an **anomaly (fraud)**.
- `anomalies.sum()` shows number of flagged transactions (≈570).

🧠 *Cross Q: Why 99th percentile?*
→ It filters out the top 1% of high-error samples — likely to be frauds.

🔹 21. Confusion Matrix and Report

```
cm = confusion_matrix(y_test, anomalies)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.show()

print(classification_report(y_test, anomalies))
```

- Compares **actual labels** vs **predicted anomalies**.
- Visualized via a heatmap.
- Prints precision, recall, F1-score.

🧠 *Cross Q: What does each term mean?*

Metric	Meaning
Precision	Of the samples predicted as fraud, how many are actually fraud.
Recall	Of all actual frauds, how many were correctly detected.
F1-score	Balance between precision and recall.
Accuracy	Overall fraction of correct predictions.

Example output:

```
Precision: 0.13, Recall: 0.73, F1-score: 0.22
```

So the model catches ~73% of frauds but with some false positives.

🧾 Summary of What Happens

Step	Action	Outcome
1	Load + clean data	29 features + class labels
2	Train only on normal data	Learns normal patterns
3	Reconstruct test data	Fraud samples have high error
4	Use threshold	Detect anomalies
5	Evaluate	High recall, decent accuracy

🎯 Viva one-liner summary:

> "This notebook builds an autoencoder trained only on normal transactions.
> It reconstructs normal data well, but frauds — which look different — show higher reconstruction errors.
> By setting a threshold on reconstruction error, we detect frauds as anomalies."

Would you like me to make a **short viva cheat sheet** (one-liners for each key block + typical cross-questions + 2-line answers)? That's the best way to prep fast before your presentation.

You said:

explain tp, tn, ...

ChatGPT said:

Perfect — this is one of the most **common viva checkpoints** after showing a confusion matrix 👏

Let's go through it **slowly, clearly, and technically** so you can answer *any* cross-question.

🧠 What is a Confusion Matrix?

A **confusion matrix** is a 2×2 table that compares:

- The **actual labels** (from dataset) vs
- The **predicted labels** (from your model)

It shows how many predictions were **correct** and how many were **wrong** — broken down by class.

⚙️ In your case:

You have **two classes**:

- **0 → Normal transaction**
- **1 → Fraudulent transaction**

So the confusion matrix will look like this:

Actual / Predicted	Predicted Normal (0)	Predicted Fraud (1)
Actual Normal (0)	True Negative (**TN**)	False Positive (**FP**)
Actual Fraud (1)	False Negative (**FN**)	True Positive (**TP**)

🧩 Let's explain each term in context:

Term	Meaning	Example in your fraud case
TP (True Positive)	Model correctly predicted *fraud* as fraud	A fraudulent transaction that was **correctly flagged**
TN (True Negative)	Model correctly predicted *normal* as normal	A normal transaction that was **not flagged**
FP (False Positive)	Model wrongly predicted *normal* as fraud	A normal transaction **wrongly flagged as fraud** (false alarm)
FN (False Negative)	Model wrongly predicted *fraud* as normal	A fraudulent transaction **missed by the model**

🔹 Example

Let's say your confusion matrix looked like this:

```
[[56700, 164],
 [   26,  72]]
```

Metric	Value	Meaning
TN = 56700	56,700 normal transactions correctly classified	
FP = 164	164 normal transactions wrongly flagged as fraud	
FN = 26	26 frauds missed by the model	
TP = 72	72 frauds correctly detected	

📊 From these, we derive performance metrics:

Metric	Formula	Meaning
Accuracy	(TP + TN) / (TP + TN + FP + FN)	Overall correctness of the model
Precision	TP / (TP + FP)	Out of all predicted frauds, how many were actually fraud
Recall (Sensitivity)	TP / (TP + FN)	Out of all actual frauds, how many were detected
F1-Score	2 × (Precision × Recall) / (Precision + Recall)	Balance between precision and recall

---

🧠 Interpretation in your model:

From your results:

```
Precision: 0.13
Recall: 0.73
F1-score: 0.22
Accuracy: 0.99
```

- **Accuracy** looks high because there are many normal samples.
- **Recall (0.73)** → model detects 73% of frauds (good!).
- **Precision (0.13)** → but many false alarms — most predicted frauds are actually normal.
- **F1** balances both.

💡 Viva-style one-liners

Question	Answer
What is TP?	Correctly predicted fraud transactions.
What is TN?	Correctly predicted normal transactions.
What is FP?	Normal transactions incorrectly flagged as fraud (false alarms).
What is FN?	Fraud transactions missed by the model.
Why can accuracy be misleading?	Because the dataset is imbalanced — many normal cases make accuracy look high even if fraud detection is poor.

Which is more important in fraud detection: precision or recall?	Usually **recall**, because missing a fraud (FN) is worse than a false alarm (FP).