AMRITA SCHOOL OF ENGINEERING

19EEE362- PROJECT REPORT

DEEP LEARNING FOR VISUAL COMPUTING

Predicting of MOSFET Drain Current Using Neural Networks

SUBMITED BY :

MRUTHYUNJAI SRIRAM E	CB.EN.U4ECE21029
S.BALASUBRAMANIYAM	CB.EN.U4ECE21145
VAIBAV V	CB. EN. U4ECE21060
RUBAGTARUN	CB.EN.U4ECE21044

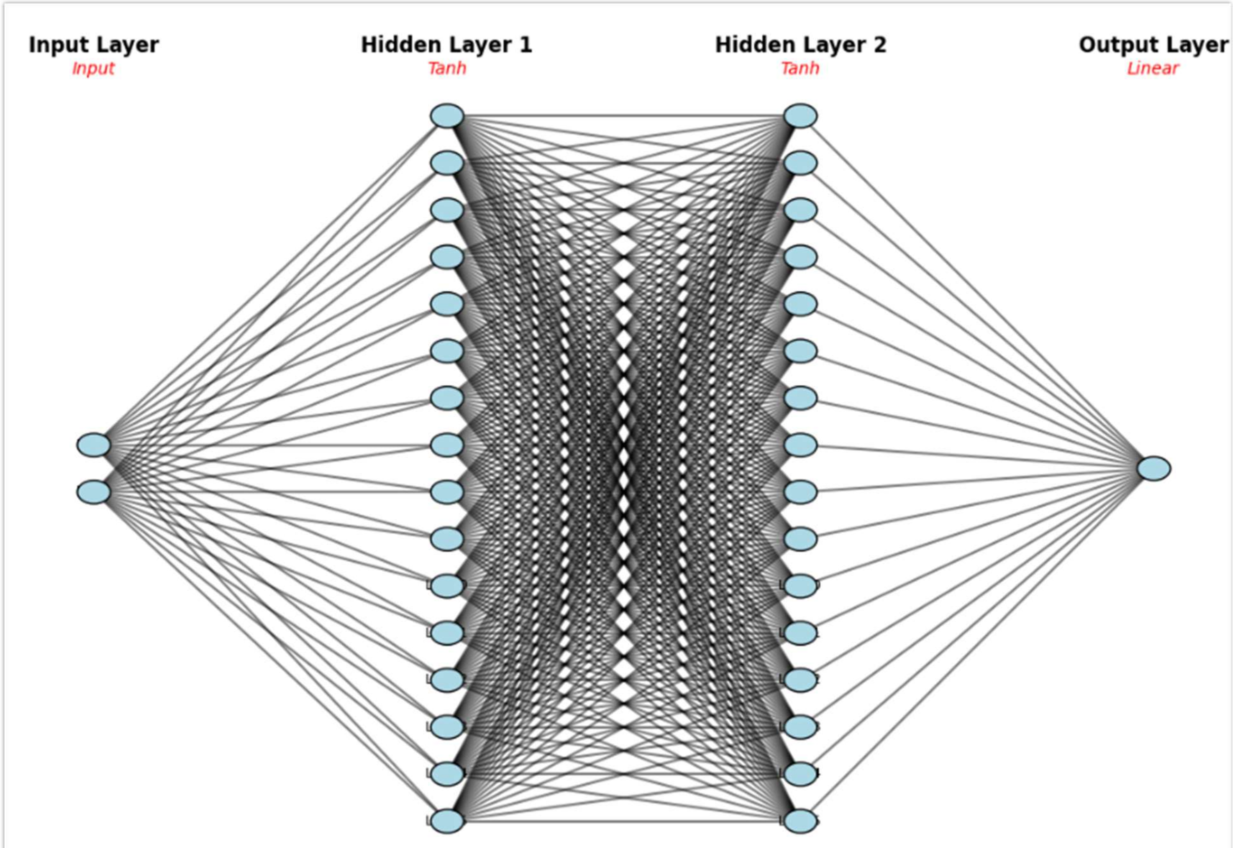
FACULTY IN CHARGE :
LEKSHMI R.R

Introduction:

- The Metal Oxide Insulator Field Effect Transistor (MOSFET) plays a crucial role in modern VLSI chip design, serving as the fundamental building block for integrating various chip-level components. Before physically fabricating a chip, behavioural simulations are essential to estimate device and circuit performance accurately. The accuracy of these simulations heavily relies on the precision of semiconductor device models used.
- As MOS transistors shrink, conventional physics-based models struggle to capture all characteristics, leading to increased interest in data-driven machine learning models. These models, trained on real device data or simulations, offer a middle ground between accuracy and development time.
- In this project, we aim to develop a neural network-based MOSFET model using training data collected from a circuit simulator. This model will predict drain current based on gate and drain voltages, paving the way for faster and more accurate circuit simulations in VLSI design.

NN architecture:

- Input Layer:
  - 2 parameters(Vgs and Vds)
- Hidden Layer1:
  - 16 neurons with tanh activation function
- Hidden Layer2:
  - 16 neurons with tanh activation function
- Output Layer:
  - 1 neuron with linear activation function



Process Flow :

- Import Libraries:

importing libraries such as keras,tensorflow,matplotlib.pyplot,pandas,numpy etc.

- Reading Dataset:

Load the dataset from a CSV file.

- Preprocessing Data:

Replace zero values in Id, Vgs, and Vds columns to avoid issues with logarithmic scaling.

Extract features (Vgs and Vds) and the target variable (Id).

Apply logarithmic transformation to the features to scale them.

- Using train_test_split:

Convert the target variable to logarithmic scale.

Split the data into training and validation sets using an 80-20 split.

❖ Define Model Architecture:

The model is defined as a Sequential model with:

An input layer with 2 neurons.

Two hidden layers, each with 16 neurons and tanh activation.

An output layer with 1 neuron for predicting the drain current (Id).

Compile the model with model.compile()

❖ Plotting Training and Validation Loss:

Extract the training and validation loss values from the training history (history_1).

Plot the training and validation loss values against epochs, starting from the chosen epoch.

❖ Predicting and Plotting New Data:

Read the new data (features and target values) from a CSV file.

Predict the target values using the trained model (model.predict()).

Plot the actual target values against a feature (e.g., gate voltage) in a scatter plot using a red marker.

Plot the predicted target values against the same feature using a blue line.

Add labels to the plot for clarity (e.g., x-axis: Gate Voltage, y-axis: Drain Current).

Add a legend to differentiate between actual and predicted values.

Code:

```
import tensorflow as tf

from tensorflow import keras

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt


df = pd.read_csv('plot.csv')

df["Id"].replace({0:1e-13}, inplace=True)

df["Vgs"].replace({0:1e-3}, inplace=True)
```

```
df["Vds"].replace({0:1e-3}, inplace=True)
id=df["Id"]
Vgs=df["Vgs"]
Vds=df["Vds"]
yy=np.ravel(id)
X1=df.iloc[:,0:2]
X=np.log10(X1)

# Split train and test dataset
from sklearn.model_selection import train_test_split
#Normalize data before training
y=np.log10(yy)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
y_train=y_train.reshape(-1,1)
y_test=y_test.reshape(-1,1)

# We'll use Keras to create a Neural network
model = tf.keras.Sequential()
model.add(keras.layers.Dense(16, activation='tanh', input_shape=(2,)))
model.add(keras.layers.Dense(16,activation='tanh'))
model.add(keras.layers.Dense(1))
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
history_1 = model.fit(X_train, y_train, epochs=1000,
validation_data=(X_test, y_test))

# Exclude the first few epochs so the graph is easier to read
loss = history_1.history['loss']
val_loss = history_1.history['val_loss']
```

```
epochs = range(1, len(loss) + 1)
SKIP = 300
plt.plot(epochs[SKIP:], loss[SKIP:], 'g.', label='Training loss')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# New data input for testing
df1 = pd.read_csv('val02.csv')
val_feature=df1.iloc[:,0:2]
xnew=val_feature
ynew=np.ravel(df1["id"])
#Normalize validation data
xval=np.log10(xnew)
#xval=xnew/5
yval=np.log10(ynew)
yval=yval.reshape(-1,1)
#print(yval)
# Predict the new dataset
y_pred = model.predict(xval)
#plot the result
vg1=df1["vgs"]
vg=np.ravel(vg1)
vg=vg.reshape(-1,1)
#print(vg)
```

```
plt.plot(vg, 10**(yval), 'ro', label='Actual')

plt.plot(vg,10**(y_pred), 'b', label='Predicted')

#plt.yscale("log")

plt.title('Actual and Predicted Value')

plt.xlabel('Gate Voltage')

plt.ylabel('Drain Current')

plt.legend()

plt.show()
```

```
Epoch 1/1000
c:\users\vvaib\appdata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\python
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - loss: 33.4795 - mae: 4.4769 - val_loss: 1.5769 - val_mae: 1.0178
Epoch 2/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 1.3500 - mae: 0.9577 - val_loss: 0.8866 - val_mae: 0.7579
Epoch 3/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.7394 - mae: 0.6844 - val_loss: 0.3626 - val_mae: 0.4359
Epoch 4/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.2828 - mae: 0.3782 - val_loss: 0.1478 - val_mae: 0.2353
Epoch 5/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.1159 - mae: 0.2165 - val_loss: 0.0919 - val_mae: 0.1670
Epoch 6/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 956us/step - loss: 0.0654 - mae: 0.1581 - val_loss: 0.0659 - val_mae: 0.1456
Epoch 7/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.0527 - mae: 0.1383 - val_loss: 0.0471 - val_mae: 0.1230
Epoch 8/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.0439 - mae: 0.1179 - val_loss: 0.0416 - val_mae: 0.1040
Epoch 9/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.0373 - mae: 0.1033 - val_loss: 0.0340 - val_mae: 0.0945
Epoch 10/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.0352 - mae: 0.0916 - val_loss: 0.0305 - val_mae: 0.0834
Epoch 11/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.0297 - mae: 0.0839 - val_loss: 0.0282 - val_mae: 0.0791
Epoch 12/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.0324 - mae: 0.0796 - val_loss: 0.0263 - val_mae: 0.0756
Epoch 13/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 967us/step - loss: 0.0218 - mae: 0.0717 - val_loss: 0.0288 - val_mae: 0.0740
...
Epoch 999/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 5.0465e-05 - mae: 0.0047 - val_loss: 4.7043e-05 - val_mae: 0.0039
Epoch 1000/1000
634/634 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 4.9556e-05 - mae: 0.0047 - val_loss: 5.5686e-05 - val_mae: 0.0045
```

Result comparison between activation functions :

Relu activation function :

```python
# Model definition
model = tf.keras.Sequential()
model.add(keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dropout(0.3))
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(32, activation='relu'))
model.add(keras.layers.Dense(1))

# Learning rate scheduler
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', patience=10, factor=0.1, min_lr=1e-6)

# Early stopping
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)

# Custom callback to print learning rate
class PrintLearningRate(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        lr = self.model.optimizer.learning_rate
        if isinstance(lr, tf.keras.optimizers.schedules.LearningRateSchedule):
            lr = lr(epoch).numpy()
        else:
            lr = lr.numpy()
        print(f"Epoch {epoch + 1}: Learning rate is {lr}")

# Compile the model
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
              loss='mse',
              metrics=['mae'])

# Train the model
history = model.fit(X_train, y_train, epochs=500,
                    validation_data=(X_test, y_test),
                    callbacks=[lr_scheduler, early_stopping, PrintLearningRate()])
```

[15] 33.1s

```
c:\users\vvaib\appdata\local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-packages\python311\site-packa
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/500
627/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.5784 - mae: 2.3957Epoch 1: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 14.4547 - mae: 2.3800 - val_loss: 0.1014 - val_mae: 0.1730 - learning_rate: 0.0010
Epoch 2/500
618/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.3720 - mae: 0.4174Epoch 2: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.3710 - mae: 0.4166 - val_loss: 0.0757 - val_mae: 0.1225 - learning_rate: 0.0010
Epoch 3/500
634/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.2329 - mae: 0.3010Epoch 3: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.2329 - mae: 0.3009 - val_loss: 0.0589 - val_mae: 0.1369 - learning_rate: 0.0010
Epoch 4/500
632/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.1639 - mae: 0.2474Epoch 4: Learning rate is 0.0010000000474974513
```

```
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/500
627/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.5784 - mae: 2.3957Epoch 1: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 14.4547 - mae: 2.3800 - val_loss: 0.1014 - val_mae: 0.1730 - learning_rate: 0.0010
Epoch 2/500
618/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.3720 - mae: 0.4174Epoch 2: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.3710 - mae: 0.4166 - val_loss: 0.0757 - val_mae: 0.1225 - learning_rate: 0.0010
Epoch 3/500
634/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.2329 - mae: 0.3010Epoch 3: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.2329 - mae: 0.3009 - val_loss: 0.0589 - val_mae: 0.1369 - learning_rate: 0.0010
Epoch 4/500
632/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.1639 - mae: 0.2474Epoch 4: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.1639 - mae: 0.2474 - val_loss: 0.1064 - val_mae: 0.1817 - learning_rate: 0.0010
Epoch 5/500
628/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.1301 - mae: 0.2182Epoch 5: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.1301 - mae: 0.2181 - val_loss: 0.0602 - val_mae: 0.1407 - learning_rate: 0.0010
Epoch 6/500
633/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0988 - mae: 0.1810Epoch 6: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0988 - mae: 0.1810 - val_loss: 0.1052 - val_mae: 0.2397 - learning_rate: 0.0010
Epoch 7/500
626/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0632 - mae: 0.1508Epoch 7: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0631 - mae: 0.1507 - val_loss: 0.0850 - val_mae: 0.2536 - learning_rate: 0.0010
Epoch 8/500
624/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0427 - mae: 0.1172Epoch 8: Learning rate is 0.0010000000474974513
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0427 - mae: 0.1172 - val_loss: 0.0816 - val_mae: 0.2416 - learning_rate: 0.0010
Epoch 9/500
...
634/634 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0086 - mae: 0.0532 - val_loss: 0.0875 - val_mae: 0.2345 - learning_rate: 1.0000e-04
Epoch 23/500
621/634 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0088 - mae: 0.0526Epoch 23: Learning rate is 1.0000000656873453e-05
634/634 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0088 - mae: 0.0526 - val_loss: 0.0860 - val_mae: 0.2397 - learning_rate: 1.0000e-04
```

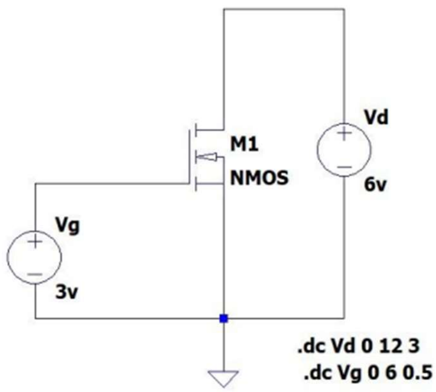
<u>Regression</u>:

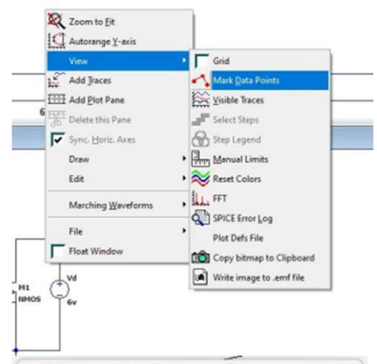
```
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

x_scaled = scale.fit_transform(X)

from sklearn.linear_model import LinearRegression
model = LinearRegression()
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x_scaled, y, test_size=0.2)
model.fit(X_train,y_train)
y_pred = model.predict(X_test)

from sklearn.metrics import mean_absolute_error,mean_squared_error

print("MSE:" + str(mean_squared_error(y_test,y_pred)))
print("MAE: " + str(mean_absolute_error(y_test,y_pred)))
✓ 0.0s

MSE:5.526419126607065
MAE: 1.8959863847097724
```

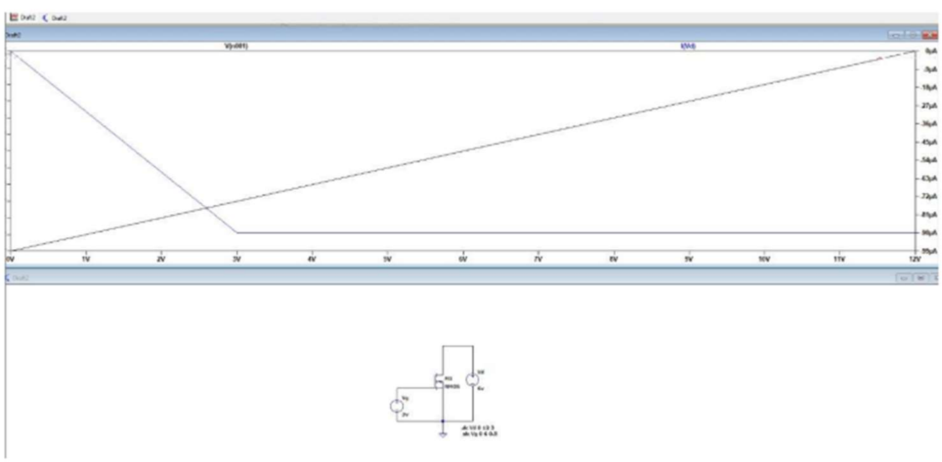
Here we have clearly shown results by trying out different activation functions in order to reduce the error rate , where in regression model the error rate is higher than the ReLu model and tanh gives us more accurate and efficient results

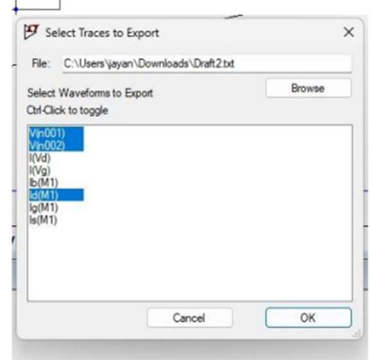
Data acquisition for the MOSFET model using LT Spice :

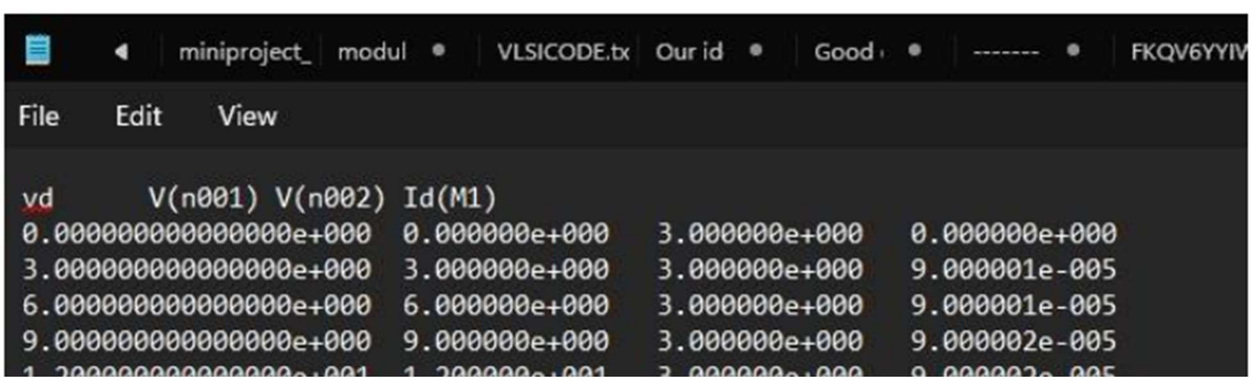
<u>Lt spice model of MOSFET</u>











vd	V(n001)	V(n002)	Id(M1)
0.000000000000000e+000	0.000000e+000	3.000000e+000	0.000000e+000
3.000000000000000e+000	3.000000e+000	3.000000e+000	9.000001e-005
6.000000000000000e+000	6.000000e+000	3.000000e+000	9.000001e-005
9.000000000000000e+000	9.000000e+000	3.000000e+000	9.000002e-005

Conclusion:

Here we have developed a neural network-based MOSFET model using training data collected from a circuit simulator. This model will predict drain current based on gate and drain voltages, paving the way for faster and more accurate circuit simulations in VLSI design.