Hands-On Artificial Neural Network (ANN) Model Fitting & Hyperparameter Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
```

Tensorflow Dropout Layer : Regularization technique to control overfitting in neural network.

Adam Optimizer : Adaptive Moment Estimation optimizer is the one of the most popular and effective algorithms for training deep neural networks. It combines the benefits of two other optimization methods **Momentum** and **RMSProp**.

EarlyStopping Callback : The Kears EarlyStopping Callback monitors a specified metric during training and automatically stops the process if the metric stops improving for a defined number of epochs.

Creating a Data (Regression Problem)

```
np.random.seed(42)

X=np.random.rand(600,2)
y=4*X[:,0]+6*X[:,1]+np.random.randn(600)*0.2
```

```
# Splitting data into training and testing
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
```

```
# Scaling the data
scaler=StandardScaler()
X_train=scaler.fit_transform(X_train)
X_test=scaler.transform(X_test)
```

Practice 1: Effect of Number of Neurons

Train 3 models with different neurons(4, 16, 64)

```
def build_model(neurons):
  model=Sequential([Dense(neurons, activation='relu', input_shape=(2,)),
                    Dense(1)
                    ])
  model.compile(optimizer='adam',loss='mse',metrics=['mae'])
  return model
```

```python
for n in [4, 16, 64]:
  print(f"\nTraining model with {n} neurons")
  model=build_model(n)
  model.fit(X_train,y_train,epochs=30,batch_size=32,verbose=0)
  print(model.evaluate(X_test,y_test))
```

```
Training model with 4 neurons
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 18.2454 - mae: 3.8627
[19.011669158935547, 3.9483752250671387]

Training model with 16 neurons
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 2.4959 - mae: 1.3257
[2.4904489517211914, 1.3268623352050781]

Training model with 64 neurons
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.4530 - mae: 0.5671
[0.44928112626075745, 0.5576094388961792]
```

Conclusion:

1. Model with 4 Neurons :
    - Very high loss & MAE
    - Model is too simple
    - Cannot capture relationship in data

    Implies Underfitting, High Bias, Low Variance.

2. Model with 16 Neurons :
    - Big improvement in loss
    - MAE reduced significantly
    - Model starts learning useful patterns

    Implies Good Fit, Balance Bias-Variance.

3. Model with 64 Neurons :
    - Very low loss
    - MAE ≈ 0.56
    - Model has high representational power

    Implies best performance on test data & No Overfitting.

Overall Conclusion: Increasing the number of neurons increases model capacity, reduces bias, and improves performance until overfitting starts.

The **verbose=0** parameter in model fitting means that no output or logging messages will be displayed during the training process. The model will train in "silent mode," which is often used for production runs or batch processing where monitoring the progress in real-time is not necessary.

Practice 2 : Effect of Activation Functions

Relu, tanh, sigmoid

```
def build_activation_model(activation):
  model=Sequential([
      Dense(16, activation=activation,input_shape=(2,)),
      Dense(1)
  ])
  model.compile(optimizer='adam',loss='mse')
  return model


for act in ['relu','tanh','sigmoid']:
  print(f"\nActivation: {act}")
  model = build_activation_model(act)
  model.fit(X_train, y_train, epochs=30, verbose=0)
  print(model.evaluate(X_test,y_test))
```

```
Activation: relu
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 1.2365
1.2129470109939575

Activation: tanh
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.6763
0.7109176516532898

Activation: sigmoid
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 2.9409
3.1860952377319336
```

Conclusion:

1. ReLU Activation Function
    - Moderate loss
    - Fast convergence
    - No vanishing gradient problem
2. Tanh Activation Function
    - Lowest loss
    - Output centered around zero
    - Smooth gradient flow
3. Sigmoid Activation Function
    - Highest loss
    - Slow learning
    - Gradient saturation problem

Overall Conclusion: The activation function strongly influences ANN performance. In shallow regression models, tanh can work better than ReLU due to zero-centered outputs, while sigmoid often performs poorly because of vanishing gradients.

Practice 3 : Learning Rate Experiment

Learning Rate : 0.1, 0.01, 0.001

```
for lr in [0.1,0.01,0.001]:
  print(f"\nLearning rate: {lr}")

  model = Sequential([
      Dense(16, activation='relu',input_shape=(2,)),
      Dense(1)
  ])

  model.compile(optimizer=Adam(learning_rate=lr),loss='mse')
  model.fit(X_train, y_train, epochs=30, verbose=0)
  print(model.evaluate(X_test,y_test))
```

```
Learning rate: 0.1
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0458
0.04770714417099953

Learning rate: 0.01
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0548
0.05751919001340866

Learning rate: 0.001
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 2.4881
2.562166929244995
```

Conclusion:

1. Learning rate = 0.1
    - Lowest loss
    - Model reached good solution quickly
    - No instability observed
2. Learning rate = 0.01
    - Slightly higher loss
    - Still stable learning
3. Learning rate = 0.001
    - Very high loss
    - Model learned too slowly
    - Did not converge in 30 epochs

Overall Conclusion: Learning rate controls the speed of learning. A very small learning rate may lead to slow convergence, while a moderately high learning rate can give better results for simple problems within limited epochs.

Practice 4 : Batch Size Impact

Batch Size = 8, 32, 128

```
for bs in [8, 32, 128]:
  print(f"\nBatch size: {bs}")

  model = Sequential([
      Dense(16, activation='relu',input_shape=(2,)),
      Dense(1)
  ])

  model.compile(optimizer='adam',loss='mse')
  model.fit(X_train,y_train,epochs=30,batch_size=bs, verbose=0)
  print(model.evaluate(X_test,y_test))
```

```
Batch size: 8
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.2126
0.2120523750782013

Batch size: 32
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 1.1255
1.0945647954940796

Batch size: 128
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 21.7778
22.793216705322266
```

Conclusion:

1. Batch Size = 8
   - Lowest loss
   - More frequent weight updates
   - Better learning of patterns

   Implies small batch size helped the model learn better.

2. Batch Size = 32
   - Moderate loss
   - Balanced but not optimal here

   Slightly slower or less effective than batch size 8.

3. Batch size = 128
   - Very high loss
   - Too few weight updates per epoch
   - Model failed to converge

   Implies Batch size is too large for this dataset & Poor learning within limited epochs.

Practice 5 : Overfitting and Regularization

Step 1: Create Overfitting Model : Dense(128), Dense(128), Dense(64)

Step 2: Fixing it using : Dropout(0.3), L2 Regularization, EarlyStopping

Overfitting Model

```
model=Sequential([
    Dense(128,activation='relu',input_shape=(2,)),
    Dense(128,activation='relu'),
    Dense(64,activation='relu'),
    Dense(1)
])

model.compile(optimizer='adam',loss='mse')

history=model.fit(
    X_train, y_train,
    epochs=80,
    validation_split=0.2,
    verbose=0
)
```

The **validation_split=0.2** parameter in model fitting (typically in Keras/TensorFlow) automatically reserves the last 20% of the training data to be used for validation.

Fixing model with Dropout and EarlyStopping

```
model=Sequential([
    Dense(128, activation='relu',input_shape=(2,)),
    Dropout(0.3),
    Dense(64,activation='relu'),
    Dropout(0.3),
    Dense(1)
])

model.compile(optimizer='adam',loss='mse')

es=EarlyStopping(monitor='val_loss',patience=5,restore_best_weights=True)

history=model.fit(
    X_train, y_train,
    epochs=100,
    validation_split=0.2,
    callbacks=[es],verbose=0
)
```

Practice 6 : Error Metrics

```
y_pred=model.predict(X_test)

print('MAE:',mean_absolute_error(y_test,y_pred))
print('MSE:',mean_squared_error(y_test,y_pred))
print('RMSE',np.sqrt(mean_squared_error(y_test,y_pred)))
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
MAE: 0.2052432504451226
MSE: 0.06591776722163095
RMSE 0.2567445563622157
```

Conclusion:

1. MAE(Mean Absolute Error)
    - On average, predictions are off by ~0.20 units
    - Easy to interpret
    - Treats all errors equally

   Implies model predictions are quite close to actual values.
2. MSE(Mean Squared Error)
    - Squared errors penalize large mistakes more
    - Low value indicates few large errors

      Implies model is not making extreme prediction errors.

3. RMSE(Root Mean Squared Error)
    - Same unit as target variable
    - Slightly larger than MAE due to squaring

   Implies errors are small and consistent.

Practice 7 : ANN VS Linear Regression

```
from sklearn.linear_model import LinearRegression

lr=LinearRegression()
lr.fit(X_train,y_train)

y_lr=lr.predict(X_test)

print('Linear Regression MAE:',mean_absolute_error(y_test,y_lr))
print('ANN MAE:',mean_absolute_error(y_test,y_pred))
```

```
Linear Regression MAE: 0.16127674638972617
ANN MAE: 0.2052432504451226
```

Conclusion:

1. Linear Regression
    - Lower MAE
    - Simpler model fits data well
    - Relationship in data is mostly linear

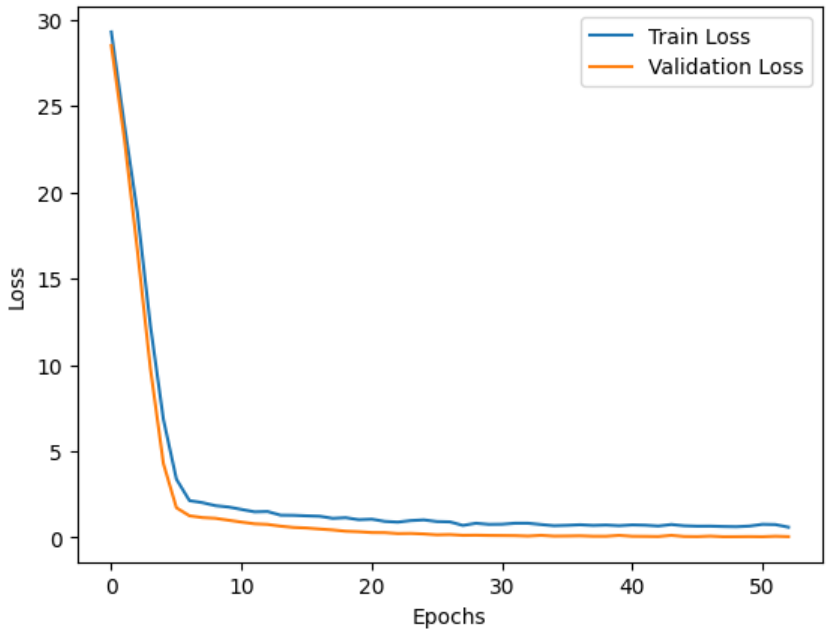
    Implies Linear Regression is more suitable for this problem.

2. ANN Model
    - Higher MAE
    - More complex than required
    - Needs more data / tuning to outperform linear model

    Implies ANN adds unnecessary complexity & Risk of overfitting or inefficient learning.

Practice 8 : Loss Curve Plot

```
plt.plot(history.history['loss'],label='Train Loss')
plt.plot(history.history['val_loss'],label='Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



Interpretation:

1. Rapid decrease at the beginning
    - Both training and validation loss drop sharply in early epochs.
    - Model quickly learns the main pattern in data.
2. Parallel behavior of train & validation loss
    - Training and validation curves stay close to each other.
    - No large gap between them.
3. Smooth convergence after some epochs
    - Loss decreases slowly and then stabilizes.
    - No sudden spikes or divergence.
4. Validation loss slightly lower than training loss

    This can happen due to:
    - Regularization
    - Noise in training data
    - Validation split being slightly easier

Overall Conclusion: The loss curves show steady convergence with training and validation losses decreasing together, indicating a well-fitted model with good generalization and no overfitting.

Overall Conclusions of entire ANN practice:

1) Model capacity matters

Increasing the number of neurons improves learning by reducing underfitting, but excessive capacity can lead to overfitting.

2) Activation function choice affects performance

In shallow regression models, tanh can outperform ReLU due to zero-centered outputs, while *sigmoid* performs poorly because of vanishing gradients.

3) Learning rate controls convergence speed

A very small learning rate leads to slow or incomplete learning, while a suitably higher learning rate enables faster and better convergence for simple datasets.

4) Batch size influences learning stability

Smaller batch sizes provide more frequent weight updates and better generalization, whereas very large batch sizes may slow learning and increase error.

5) Proper regularization prevents overfitting
Techniques like Dropout and Early Stopping help control model complexity and improve generalization.

6) Different error metrics provide different insights
MAE, MSE, and RMSE together show prediction accuracy, error consistency, and sensitivity to outliers.

7) Complex models are not always superior
Linear Regression outperformed ANN in this case, highlighting that simpler models can be more effective when the data relationship is linear.

8) Loss curves are essential diagnostic tools
Parallel and smoothly decreasing training and validation loss curves indicate good convergence and absence of overfitting.

9) Hyperparameter tuning is crucial
ANN performance depends strongly on neurons, activation functions, learning rate, and batch size.

10) Model choice should be data-driven
The best model and configuration depend on data complexity, size, and underlying relationships.