Exam: Problem of Module 13

To solve the salary prediction classification problem using a neural network, we will follow these steps:

1. Load and preprocess the dataset.
2. Build a neural network model.
3. Train the model.
4. Evaluate the model.
5. Tune hyperparameters and handle class imbalance if needed.
6. Visualize the results.

Let's proceed step-by-step:

```
import pandas as pd

# Load the dataset
file_path = '/mnt/data/salary.csv'
df = pd.read_csv(file_path)

# Display the first few rows of the dataset
df.head()
```

Preprocess the Dataset

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Fill missing values if any
df = df.fillna(method='ffill')

# Encode categorical variables
label_encoders = {}
for column in df.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    df[column] = le.fit_transform(df[column])
```

```
    label_encoders[column] = le

# Split the dataset into features and target
X = df.drop('salary', axis=1)
y = df['salary']

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the dataset 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)
```

Step 2: Build a Neural Network Model

We'll use TensorFlow and Keras to build a neural network model.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Build the neural network model
def create_model(input_shape):
    model = Sequential()
    model.add(Dense(64, activation='relu', input_shape=(input_shape,)))
    model.add(Dropout(0.5))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))  # Binary classification

    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
    return model

model = create_model(X_train.shape[1])
```

Step 3: Train the Model

We'll train the model on the training data and validate it on the validation data.

```
# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2, verbose=1)
```

Step 4: Evaluate the Model

Evaluate the model on the test data.

```
# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')
```

Step 5: Visualize the Results

Plot the training and validation accuracy and loss.

```
import matplotlib.pyplot as plt

# Plot training and validation accuracy/loss
history_dict = history.history

acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs_range = range(1, len(acc) + 1)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, 'bo', label='Training accuracy')
plt.plot(epochs_range, val_acc, 'b', label='Validation accuracy')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, 'bo', label='Training loss')
plt.plot(epochs_range, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Step 6: Handle Class Imbalance and Hyperparameter Tuning (Optional)

To handle class imbalance, you can use techniques like class weighting or oversampling. For hyperparameter tuning, consider using tools like Keras Tuner.

Here’s the complete code implementation:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import matplotlib.pyplot as plt

# Load the dataset
file_path = '/mnt/data/salary.csv'
df = pd.read_csv(file_path)

# Fill missing values if any
df = df.fillna(method='ffill')

# Encode categorical variables
label_encoders = {}
for column in df.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    df[column] = le.fit_transform(df[column])
    label_encoders[column] = le
```

```
# Split the dataset into features and target
X = df.drop('salary', axis=1)
y = df['salary']

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the dataset 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)

# Build the neural network model
def create_model(input_shape):
    model = Sequential()
    model.add(Dense(64, activation='relu', input_shape=(input_shape,)))
    model.add(Dropout(0.5))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))  # Binary classification

    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
    return model

model = create_model(X_train.shape[1])

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2, verbose=1)

# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')

# Plot training and validation accuracy/loss
history_dict = history.history
```

```
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs_range = range(1, len(acc) + 1)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, 'bo', label='Training accuracy')
plt.plot(epochs_range, val_acc, 'b', label='Validation accuracy')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, 'bo', label='Training loss')
plt.plot(epochs_range, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

This code provides a complete neural network implementation for the salary prediction classification problem using the provided dataset.

Run Command:

```
python m_13_exam_batch_03_roll_12.py
```

Return result:

You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`. DNN 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 off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.

age workclass fnlwgt education education-num marital-status occupation ... race sex capital-gain capital-loss hours-per-week native-country salary

0 39 State-gov 77516 Bachelors 13 Never-married Adm-clerical ... White Male 2174 0 40 United-States <=50K

1 50 Self-emp-not-inc 83311 Bachelors 13 Married-civ-spouse Exec-managerial ... White Male 0 13 United-States <=50K

2 38 Private 215646 HS-grad 9 Divorced Handlers-cleaners ... White Male 0 0 40 United-States <=50K

3 53 Private 234721 11th 7 Married-civ-spouse Handlers-cleaners ... Black Male 0 0 40 United-States <=50K

4 28 Private 338409 Bachelors 13 Married-civ-spouse Prof-specialty ... Black Female 0 0 40 Cuba <=50K

[5 rows x 15 columns]
Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
 super().__init__(activity_regularizer=activity_regularizer, **kwargs)

2024-06-19 18:33:44.295167: 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: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.7448 - loss: 0.5133 - val_accuracy: 0.8392 - val_loss: 0.3534
Epoch 2/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8185 - loss: 0.3882 - val_accuracy: 0.8425 - val_loss: 0.3354
Epoch 3/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 981us/step - accuracy: 0.8318 - loss: 0.3627 - val_accuracy: 0.8439 - val_loss: 0.3333
Epoch 4/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 975us/step - accuracy: 0.8328 - loss: 0.3569 - val_accuracy: 0.8429 - val_loss: 0.3300
Epoch 5/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 980us/step - accuracy: 0.8329 - loss: 0.3556 - val_accuracy: 0.8474 - val_loss: 0.3269
Epoch 6/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8382 - loss: 0.3485 - val_accuracy: 0.8481 - val_loss: 0.3246
Epoch 7/50

814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 972us/step - accuracy: 0.8400 - loss: 0.3519 - val_accuracy: 0.8495 - val_loss: 0.3225
Epoch 8/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 977us/step - accuracy: 0.8372 - loss: 0.3491 - val_accuracy: 0.8491 - val_loss: 0.3215
Epoch 9/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 983us/step - accuracy: 0.8428 - loss: 0.3377 - val_accuracy: 0.8521 - val_loss: 0.3207
Epoch 10/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8379 - loss: 0.3457 - val_accuracy: 0.8549 - val_loss: 0.3198
Epoch 11/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 992us/step - accuracy: 0.8388 - loss: 0.3398 - val_accuracy: 0.8557 - val_loss: 0.3176
Epoch 12/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 962us/step - accuracy: 0.8434 - loss: 0.3375 - val_accuracy: 0.8534 - val_loss: 0.3186
Epoch 13/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 973us/step - accuracy: 0.8485 - loss: 0.3332 - val_accuracy: 0.8481 - val_loss: 0.3203
Epoch 14/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 969us/step - accuracy: 0.8471 - loss: 0.3336 - val_accuracy: 0.8567 - val_loss: 0.3170
Epoch 15/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 977us/step - accuracy: 0.8410 - loss: 0.3445 - val_accuracy: 0.8534 - val_loss: 0.3174
Epoch 16/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 979us/step - accuracy: 0.8392 - loss: 0.3439 - val_accuracy: 0.8534 - val_loss: 0.3169
Epoch 17/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 961us/step - accuracy: 0.8483 - loss: 0.3307 - val_accuracy: 0.8532 - val_loss: 0.3169
Epoch 18/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8448 - loss: 0.3361 - val_accuracy: 0.8551 - val_loss: 0.3175
Epoch 19/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 991us/step - accuracy: 0.8492 - loss: 0.3334 - val_accuracy: 0.8544 - val_loss: 0.3168
Epoch 20/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 971us/step - accuracy: 0.8436 - loss: 0.3354 - val_accuracy: 0.8534 - val_loss: 0.3164
Epoch 21/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 984us/step - accuracy: 0.8451 - loss: 0.3358 - val_accuracy: 0.8537 - val_loss: 0.3175
Epoch 22/50

814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 962us/step - accuracy: 0.8382 - loss: 0.3446 - val_accuracy: 0.8551 - val_loss: 0.3160
Epoch 23/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 976us/step - accuracy: 0.8406 - loss: 0.3435 - val_accuracy: 0.8572 - val_loss: 0.3164
Epoch 24/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 991us/step - accuracy: 0.8446 - loss: 0.3369 - val_accuracy: 0.8571 - val_loss: 0.3156
Epoch 25/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8432 - loss: 0.3325 - val_accuracy: 0.8557 - val_loss: 0.3158
Epoch 26/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8468 - loss: 0.3330 - val_accuracy: 0.8561 - val_loss: 0.3156
Epoch 27/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8480 - loss: 0.3292 - val_accuracy: 0.8509 - val_loss: 0.3183
Epoch 28/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8469 - loss: 0.3331 - val_accuracy: 0.8587 - val_loss: 0.3149
Epoch 29/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8421 - loss: 0.3384 - val_accuracy: 0.8558 - val_loss: 0.3146
Epoch 30/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8468 - loss: 0.3328 - val_accuracy: 0.8537 - val_loss: 0.3156
Epoch 31/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8444 - loss: 0.3342 - val_accuracy: 0.8540 - val_loss: 0.3131
Epoch 32/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8462 - loss: 0.3399 - val_accuracy: 0.8563 - val_loss: 0.3141
Epoch 33/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8468 - loss: 0.3332 - val_accuracy: 0.8543 - val_loss: 0.3140
Epoch 34/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8432 - loss: 0.3321 - val_accuracy: 0.8537 - val_loss: 0.3154
Epoch 35/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8503 - loss: 0.3303 - val_accuracy: 0.8546 - val_loss: 0.3146
Epoch 36/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8447 - loss: 0.3363 - val_accuracy: 0.8569 - val_loss: 0.3140
Epoch 37/50

```
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8438 - loss: 0.3368 - val_accuracy: 0.8549 - val_loss: 0.3148
Epoch 38/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8435 - loss: 0.3332 - val_accuracy: 0.8537 - val_loss: 0.3129
Epoch 39/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8474 - loss: 0.3303 - val_accuracy: 0.8520 - val_loss: 0.3144
Epoch 40/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8460 - loss: 0.3320 - val_accuracy: 0.8540 - val_loss: 0.3153
Epoch 41/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8467 - loss: 0.3330 - val_accuracy: 0.8555 - val_loss: 0.3138
Epoch 42/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8465 - loss: 0.3301 - val_accuracy: 0.8551 - val_loss: 0.3127
Epoch 43/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8458 - loss: 0.3322 - val_accuracy: 0.8561 - val_loss: 0.3130
Epoch 44/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8476 - loss: 0.3296 - val_accuracy: 0.8558 - val_loss: 0.3138
Epoch 45/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8477 - loss: 0.3344 - val_accuracy: 0.8538 - val_loss: 0.3155
Epoch 46/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8474 - loss: 0.3236 - val_accuracy: 0.8554 - val_loss: 0.3141
Epoch 47/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8469 - loss: 0.3300 - val_accuracy: 0.8548 - val_loss: 0.3133
Epoch 48/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8471 - loss: 0.3328 - val_accuracy: 0.8551 - val_loss: 0.3144
Epoch 49/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8483 - loss: 0.3271 - val_accuracy: 0.8548 - val_loss: 0.3134
Epoch 50/50
814/814 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8472 - loss: 0.3317 - val_accuracy: 0.8538 - val_loss: 0.3136
204/204 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - accuracy: 0.8606 - loss: 0.3086
Test Accuracy: 0.8538
```