Introduction: Handwritten digit recognition is a popular and challenging problem in the field of machine learning and computer vision. It involves recognizing the digits (0-9) that are written by hand in various styles and formats. In this project, we aim to build a Fully Convolutional Network (FCN) model to recognize handwritten digits using the MNIST dataset.

Objective: The objective of this project is to build a FCN model that achieves high accuracy in recognizing handwritten digits. We will explore various techniques and approaches to improve the performance of the model and analyze the sources of error. Additionally, we will implement mixture density networks to model the uncertainty in the predictions of the model.

Methodology: The project will be divided into six modules, each addressing a specific aspect of the problem.

- In Module 1, we will load the MNIST dataset and preprocess the data for training and testing the model. We will also visualize some of the training data to understand the structure of the images and normalize the data to ensure that the pixel values fall within a certain range.
- In Module 2, we will implement the feed-forward network functions that will be used in the FCN model. We will choose an appropriate number of hidden layers and neurons for the model and train it using stochastic gradient descent.
- In Module 3, we will analyze the model's performance on the training and testing datasets to identify potential sources of error. We will calculate the confusion matrix and visualize it using a heatmap to identify which digits are being misclassified.
- In Module 4, we will implement L2 regularization to prevent overfitting of the model. We will choose an appropriate regularization parameter to balance between the training error and the regularization term and compare the performance of the model with and without regularization.
- In Module 5, we will implement mixture density networks to model the uncertainty in the predictions of the model. We will choose an appropriate number of mixture components to model the data and evaluate the performance of the model with and without mixture density networks.
- In Module 6, we will evaluate the performance of the MDN model using evaluation metrics such as accuracy, precision, recall, and F1-score. Finally, we will display the results of the project.

Conclusion: In this project, we aim to build a FCN model that can accurately recognize handwritten digits using the MNIST dataset. We will explore various techniques and approaches to improve the performance of the model and analyze the sources of error. By implementing mixture density networks, we will also model the uncertainty in the predictions of the model. Overall, this project will provide a comprehensive approach to solving the problem of handwritten digit recognition using FCN.

Code Explanation:

Module 1:

The first part of the code displays the first 25 images from the training set using the Matplotlib library. It creates a grid of 5x5 subplots, with each subplot showing an image from the training set. The imshow() function is used to display the image, and the set_title() function is used to show the corresponding label for the image.

The next part of the code reshapes the input data and normalizes it to ensure that the pixel values fall within a certain range. The reshape() function is used to change the shape of the input data from a 3D tensor to a 2D tensor of shape (60000, 784) for the training set and (10000, 784) for the test

set. The astype() function is used to convert the data type to float32. The division by 255 is done to normalize the pixel values between 0 and 1.

Finally, the code converts the class labels to categorical variables using the to_categorical() function of the Keras library. The number of classes is set to 10 as there are 10 different digits in the dataset. The new shape of the input data and the class labels are printed to verify the changes made to the data.

MODULE 2:

This code is for building and training a feedforward neural network model for handwritten digit recognition using the MNIST dataset.

- **input_size = 784**: This sets the number of neurons in the input layer of the neural network. In this case, each input is a 28x28 image, which is flattened into a 784-dimensional input vector.
- **batch_size = 200**: This sets the number of training examples used in each iteration of the stochastic gradient descent optimization algorithm.
- **hidden1 = 400** and **hidden2 = 20**: These variables set the number of neurons in the two hidden layers of the neural network.
- **epochs = 25**: This sets the number of iterations over the entire training dataset during training.
- **model = Sequential()**: This initializes an empty sequential model object.
- **model.add(Dense(hidden1, input_dim=input_size, activation='relu'))**: This adds a fully connected (dense) layer to the model with **hidden1** neurons, a **input_size** dimensional input, and ReLU activation.
- **model.add(Dense(hidden2, activation='relu'))**: This adds another fully connected layer to the model with **hidden2** neurons and ReLU activation.
- **model.add(Dense(classes, activation='softmax'))**: This adds the output layer to the model with **classes** neurons and softmax activation, which will output the predicted probability distribution over the 10 possible digit classes.
- **model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='sgd')**: This compiles the model using categorical cross-entropy as the loss function, accuracy as the metric to evaluate model performance, and stochastic gradient descent as the optimizer.
- **model.summary()**: This prints a summary of the model architecture.
- **model.fit(x_train, y_train, batch_size=batch_size, epochs=10, verbose=2)**: This trains the model for 10 epochs on the training data (**x_train**, **y_train**), using the specified batch size and verbose level.
- **history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test))**: This trains the model for **epochs** number of epochs on the training data, and validates the model performance on the test data (**x_test**, **y_test**).
- **score = model.evaluate(x_test, y_test, verbose=0)**: This evaluates the trained model on the test data, and returns the test loss and accuracy.

- **print("Test loss:", score[0])** and **print("Test accuracy:", score[1])**: These print the test loss and accuracy of the trained model.

MODULE 3:

This code is performing error analysis on the predictions of the model.

First, the predicted labels are obtained by applying the model to the test set using **model.predict(x_test).** These predicted labels are then converted to their corresponding class indices using **np.argmax(y_pred, axis=1).** Similarly, the true labels are obtained by converting the one-hot encoded labels of the test set **y_test** to their class indices using **np.argmax(y_test, axis=1).**

The confusion matrix is computed using **confusion_matrix(y_true_labels, y_pred_labels)** from the scikit-learn library. The matrix is then visualized using a color map and the x and y axis labels show the true and predicted labels.

Finally, some of the misclassified examples are plotted using a 5x5 grid of subplots. The indices of the misclassified examples are obtained using **np.where(y_pred_labels != y_true_labels)[0].** The misclassified images are then displayed along with their true and predicted labels.

MODULE 4:

This code is comparing the performance of a neural network model with and without L2 regularization on a handwritten digit recognition task. The code is divided into two parts: one that builds and trains the model without regularization, and another that builds and trains the model with L2 regularization.

In the first part, the model is defined using the Sequential() function from Keras. The model architecture consists of three fully connected layers, with the first layer having 400 hidden units, the second layer having 20 hidden units, and the output layer having 10 units (corresponding to the 10 digits to be recognized). The input layer has 784 units, corresponding to the flattened 28x28 pixel images of the digits in the MNIST dataset. The activation function used in the hidden layers is the rectified linear unit (ReLU), and the output layer uses the softmax activation function to produce probability distributions over the 10 classes. The model is compiled with a categorical cross-entropy loss function, SGD optimizer, and accuracy metric.

The model is then trained on the MNIST dataset with a batch size of 200, for 25 epochs. The training and validation accuracy and loss values are stored in the 'history_without_reg' variable. Finally, the test accuracy and loss of the model are evaluated and printed.

In the second part, a similar model is defined and trained, but with the addition of L2 regularization to the weight matrices of the two hidden layers. The regularization strength is set to 0.01. The model is then compiled, trained, and evaluated in the same way as the first model, with the training and validation accuracy and loss values stored in the 'history_with_reg' variable.

The two trained models are then plotted side-by-side using Matplotlib. The left subplot shows the training and validation accuracy of the two models over the course of training, and the right subplot shows the training and validation loss. The curves for the model with regularization are shown in red, and those for the model without regularization are shown in blue. The plots allow the user to compare the performance of the two models visually.

MODULE 5:

This code implements a Mixture Density Network (MDN) using Keras and TensorFlow. An MDN is a neural network that models a probability density function as a mixture of multiple Gaussian distributions. The MDN is trained on a dataset of input-output pairs, where the output is a continuous variable.

Here's a brief explanation of the code:

1. First, the necessary libraries are imported: **tensorflow.keras.layers**, **tensorflow_probability**, and **numpy**.
2. The number of mixture components is defined as **num_components=5**.
3. An input layer is defined with **Input(shape=(784,))**. This will take in a 784-dimensional input.
4. Two hidden layers are defined with **Dense** layers. The first has 400 neurons and uses ReLU activation, while the second has 20 neurons and also uses ReLU activation.
5. The MDN layer is defined with **Dense(num_components * 3, activation=None)**. This layer outputs three sets of parameters for each Gaussian mixture component: mean, standard deviation, and a coefficient. The coefficients are softmaxed so they sum to 1.
6. The MDN model is defined with **keras.Model(inputs=input_layer, outputs=output_layer)**.
7. A custom layer is defined called **MDNSplitter**, which splits the MDN layer output into the mean, standard deviation, and coefficients using **tf.split** and concatenates them with **tf.concat**.
8. The **mdn_loss** function is defined. This computes the negative log-likelihood of the true output given the predicted distribution. This uses **tfp.distributions.MixtureSameFamily** to define a Gaussian mixture distribution with the predicted parameters, and computes the log-probability of the true output.
9. The MDN model is compiled with **model_mdn.compile(loss=mdn_loss, optimizer='sgd')**.
10. The model is trained with **model.fit** for 10 epochs, using the **mdn_loss** as the loss function.
11. The model is evaluated with **model.evaluate** on the test data, and the test loss and accuracy are printed.
12. The model is trained for an additional 50 epochs with **model.fit**, and the training is printed.
13. Samples are generated from the MDN model using **model_mdn.predict** and **tfp.distributions.MixtureSameFamily.sample**. These samples are plotted as a histogram using **plt.hist**.

MODULE 6:

The first block of code evaluates the performance of a machine learning model on test data. It first predicts the labels of the test data using the trained model and calculates the accuracy, precision, recall, and F1-score using the predicted labels and the true labels. Then, it displays these evaluation metrics using the print function.

Next, it generates a classification report using the classification_report function from the sklearn.metrics library and a confusion matrix using the confusion_matrix function from the same library. It displays the classification report using the print function and displays the confusion matrix using the imshow function from the matplotlib library.

The third block of code predicts the classes of the test images using the trained model and displays the first 15 test images along with their predicted class and actual class using the subplot function from the matplotlib library.

Finally, the fourth block of code evaluates the accuracy of the model on the test data and displays the accuracy using the print function. If the accuracy is greater than 95%, it saves the trained model in a file named "model_mdn.h5" using the save method of the model object.