# Submission instructions

### Submission in pairs unless otherwise authorized

- This notebook contains all the questions. You should follow the instructions below.
- Solutions for both theoretical and practical parts should be written in this notebook

### Moodle submission

You should submit three files:

- IPYNB notebook:
  - All the wet and dry parts, including code, graphs, discussion, etc.
- PDF file:
  - Export the notebook to PDF. Make sure that all the cells are visible.

All files should be in the following format: "HW2_ID1_ID2.file"
Good Luck!

# Question 1 - Generalizrtion and Overfit (30 pt)

In this exercise, we will demonstrate overfitting to random labels.The settings are the following:

- Use the MNIST dataset.
- Work on the first 128 samples from the training dataset.
- Fix the following parameters:
  - Shuffle to False.
  - Batch size to 128.
- Generate random labels from Bernoulli distribution with a probability of $\frac{1}{2}$. I.e.,each sample is assigned a random label which is zero or one.

Show that by using a Fully Connected netwrok and cross-entropy loss, you are able to achieve a loss value of ~0 (the lower the better). Plot the accuracy and loss convergence for this data and the test data as a function of epochs. What is the accuracy value of the test data? Explain

In [1]:
```
import torch
import torch.nn as nn
import torchvision
from torch.utils.data import DataLoader

import pandas as pd
import matplotlib.pyplot as plt

# Constants
EPOCHS = 30
BATCH_SIZE = 128
NUM_OF_CLASSES = 2 # 0 or 1

# Transformation for the data
transform = torchvision.transforms.Compose(
    [torchvision.transforms.ToTensor(),
     torch.flatten])
```

In [ ]:

In [ ]:

In [2]:
```
# Create dataloaders
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
                                           download=True, transform=transform)


train_dataset = torch.utils.data.Subset(train_dataset, torch.arange(128))

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=False)


test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
                                          download=True, transform=transform)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

In [3]:
```
# Generate random labels from Bernoulli distribution with a p = 0.5
p = 0.5

y_train = torch.bernoulli(torch.full((len(train_dataset),), p))
y_test = torch.bernoulli(torch.full((len(test_dataset),), p))
```

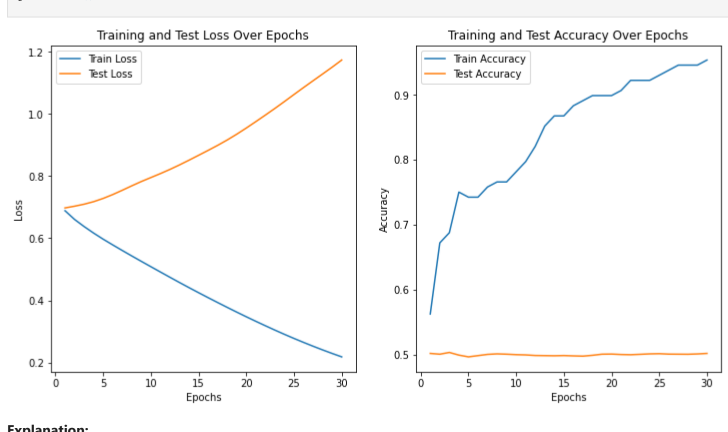
In [4]:
```
# Define the Fully Connected Network model

class FullyConnectedNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(FullyConnectedNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

In [5]:
```
model = FullyConnectedNN(784, 128, NUM_OF_CLASSES)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [6]:
```
train_loss = []
train_accuracy = []
test_loss = []
test_accuracy = []

for epoch in range(EPOCHS):

    # Training
    model.train() # Set the model to training mode
    epoch_train_loss = 0
    correct_train_predictions = 0

    for i, (train_inputs, _) in enumerate(train_dataloader):
        train_labels = y_train[(i * BATCH_SIZE) : ((i+1) * BATCH_SIZE)]

        # Forward pass
        train_outputs = model(train_inputs)
        loss = criterion(train_outputs, train_labels.long())
        epoch_train_loss += loss.item()

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        _, predicted_labels = torch.max(train_outputs, 1)
        correct_train_predictions += (predicted_labels == train_labels).sum().item()

    train_accuracy.append(correct_train_predictions / len(train_dataset))
    train_loss.append(epoch_train_loss / len(train_dataloader))

    # Testing
    model.eval() # Set the model to evaluation mode
    epoch_test_loss = 0
    correct_test_predictions = 0

    with torch.no_grad():
        for j, (test_inputs, _) in enumerate(test_dataloader):
            test_labels = y_test[(j * BATCH_SIZE) : ((j+1) * BATCH_SIZE)]

            test_outputs = model(test_inputs)
            loss = criterion(test_outputs, test_labels.long())
            epoch_test_loss += loss.item()

            _, predicted_labels = torch.max(test_outputs, 1)
            correct_test_predictions += (predicted_labels == test_labels).sum().item()

    test_accuracy.append(correct_test_predictions / len(test_dataset))
    test_loss.append(epoch_test_loss / len(test_dataloader))


#     Print the results for the current epoch
    print(f'Epoch [{epoch+1}/{EPOCHS}], Train Loss: {train_loss[-1]:.4f}, Train Accuracy: {train_accuracy[-1]*
          f'Test Loss: {test_loss[-1]:.4f}, Test Accuracy: {test_accuracy[-1]* 100:.2f}%')
```

```
Epoch [1/30], Train Loss: 0.6885, Train Accuracy: 56.25%, Test Loss: 0.6975, Test Accuracy: 50.18%
Epoch [2/30], Train Loss: 0.6605, Train Accuracy: 67.19%, Test Loss: 0.7031, Test Accuracy: 50.06%
Epoch [3/30], Train Loss: 0.6376, Train Accuracy: 68.75%, Test Loss: 0.7095, Test Accuracy: 50.33%
Epoch [4/30], Train Loss: 0.6165, Train Accuracy: 75.00%, Test Loss: 0.7176, Test Accuracy: 49.92%
Epoch [5/30], Train Loss: 0.5969, Train Accuracy: 74.22%, Test Loss: 0.7281, Test Accuracy: 49.65%
Epoch [6/30], Train Loss: 0.5786, Train Accuracy: 74.22%, Test Loss: 0.7405, Test Accuracy: 49.84%
Epoch [7/30], Train Loss: 0.5606, Train Accuracy: 75.78%, Test Loss: 0.7543, Test Accuracy: 50.03%
Epoch [8/30], Train Loss: 0.5430, Train Accuracy: 76.56%, Test Loss: 0.7686, Test Accuracy: 50.11%
Epoch [9/30], Train Loss: 0.5256, Train Accuracy: 76.56%, Test Loss: 0.7824, Test Accuracy: 50.06%
Epoch [10/30], Train Loss: 0.5084, Train Accuracy: 78.12%, Test Loss: 0.7952, Test Accuracy: 49.99%
Epoch [11/30], Train Loss: 0.4913, Train Accuracy: 79.69%, Test Loss: 0.8079, Test Accuracy: 49.95%
Epoch [12/30], Train Loss: 0.4744, Train Accuracy: 82.03%, Test Loss: 0.8214, Test Accuracy: 49.86%
Epoch [13/30], Train Loss: 0.4576, Train Accuracy: 85.16%, Test Loss: 0.8359, Test Accuracy: 49.84%
Epoch [14/30], Train Loss: 0.4410, Train Accuracy: 86.72%, Test Loss: 0.8511, Test Accuracy: 49.82%
Epoch [15/30], Train Loss: 0.4248, Train Accuracy: 86.72%, Test Loss: 0.8668, Test Accuracy: 49.85%
Epoch [16/30], Train Loss: 0.4089, Train Accuracy: 88.28%, Test Loss: 0.8826, Test Accuracy: 49.80%
Epoch [17/30], Train Loss: 0.3932, Train Accuracy: 89.06%, Test Loss: 0.8988, Test Accuracy: 49.76%
Epoch [18/30], Train Loss: 0.3776, Train Accuracy: 89.84%, Test Loss: 0.9160, Test Accuracy: 49.89%
Epoch [19/30], Train Loss: 0.3624, Train Accuracy: 89.84%, Test Loss: 0.9346, Test Accuracy: 50.06%
Epoch [20/30], Train Loss: 0.3474, Train Accuracy: 89.84%, Test Loss: 0.9544, Test Accuracy: 50.08%
Epoch [21/30], Train Loss: 0.3328, Train Accuracy: 90.62%, Test Loss: 0.9751, Test Accuracy: 50.01%
Epoch [22/30], Train Loss: 0.3185, Train Accuracy: 92.19%, Test Loss: 0.9961, Test Accuracy: 49.98%
Epoch [23/30], Train Loss: 0.3045, Train Accuracy: 92.19%, Test Loss: 1.0175, Test Accuracy: 50.04%
Epoch [24/30], Train Loss: 0.2909, Train Accuracy: 92.19%, Test Loss: 1.0397, Test Accuracy: 50.11%
Epoch [25/30], Train Loss: 0.2777, Train Accuracy: 92.97%, Test Loss: 1.0623, Test Accuracy: 50.14%
Epoch [26/30], Train Loss: 0.2650, Train Accuracy: 93.75%, Test Loss: 1.0846, Test Accuracy: 50.07%
Epoch [27/30], Train Loss: 0.2527, Train Accuracy: 94.53%, Test Loss: 1.1065, Test Accuracy: 50.06%
Epoch [28/30], Train Loss: 0.2408, Train Accuracy: 94.53%, Test Loss: 1.1282, Test Accuracy: 50.05%
Epoch [29/30], Train Loss: 0.2294, Train Accuracy: 94.53%, Test Loss: 1.1502, Test Accuracy: 50.10%
Epoch [30/30], Train Loss: 0.2185, Train Accuracy: 95.31%, Test Loss: 1.1730, Test Accuracy: 50.18%
```

In [7]:
```
# Plot the accuracy and loss as a function of the epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, EPOCHS + 1), train_loss, label='Train Loss')
plt.plot(range(1, EPOCHS + 1), test_loss, label='Test Loss')
plt.title('Training and Test Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, EPOCHS + 1), train_accuracy, label='Train Accuracy')
plt.plot(range(1, EPOCHS + 1), test_accuracy, label='Test Accuracy')
plt.title('Training and Test Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```




**Explanation:**

While the network achieves decreasing training loss and increasing training accuracy, indicating learning from the training data, the testing loss increases steadily, suggesting low generalization. The testing accuracy remains stagnant around 50%, indicating the model's failure to perform better than random guessing on unseen data. This highlights the issue of overfitting, where the model memorizes the training data but fails to generalize. Therefore, although the network achieves a low training loss, it does not translate to satisfactory performance on the test data, underscoring the need for addressing overfitting and improving generalization capabilities.

---

# Question 2 - Sentiment Analysis - Classification (70 pt)

### Exercise

The goal of this exercise is to get familiar with recurrent neural networks.

The field of detecting which emotion is represented in a text is developing and being studied due to its usefulness. For example, detecting if a review is positive or negative and more.

In this exercise, you will detect the emotion of a sentence. You should get at least 47% accuracy on the test set.

You should

- Try different model architectures - Vanilla RNN and Gated model (GRU/LSTM)
- Use different optimization and regularization methods
- Try different combinations of hyperparamters

### Data

The data is a csv file containing tweets and their labels according to the emotion – {happiness, sadness, neutral}. Every row in the file (except for the header) is an example.
Examples: (Notepad++ view)

- happiness,Welcome @doeko ! Really glad to know you here. Your products rox man
- sadness,Disappointment really sucks! I'm getting used to it.
- neutral,I just want to Sleep.

You have a train file – "trainEmotions.csv" and a test file – "testEmotions.csv". Both files can be found in the "HW2_data.zip" file.

### a) EDA (10 pt)

Explore and analyze your data. Explain your data cleaning and processing pipeline.

In [8]:
```
import glob
import unidecode
import torch
import unicodedata
import string
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

```
C:\Users\USER\anaconda3\lib\site-packages\scipy\__init__.py:138: UserWarning: A NumPy version >=1.16.5 and <1.2
3.0 is required for this version of SciPy (detected version 1.24.3)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is required for this version of "
```

In [9]:
```
#clean the data from unknown signs
def clean_text(text):
    return unidecode.unidecode(text)

# Find letter index from all_letters, e.g. "a" = 0
def letterToIndex(letter):
    return torch.tensor(all_letters.find(letter), dtype=torch.long).unsqueeze(0)

# Turn a line into a <line_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def lineToTensor(line):
    return torch.nn.functional.one_hot(
        torch.stack([letterToIndex(letter) for letter in line]),
        num_classes=n_letters)
```

In [10]:
```
# Upload Train Data
train_df = pd.read_csv('trainEmotions.csv')
train_df.head(5)
```

Out[10]:

| | emotion | content |
|---|---|---|
| 0 | happiness | victory for the bulldogs was celebrated by 3 w... |
| 1 | happiness | @saraLDS Thanks for that, Sara |
| 2 | happiness | @Tony_Mandarich well welcome back from the dar... |
| 3 | happiness | @sai_shediddy lol , you gotta share too |
| 4 | happiness | first up, make up for lost time with jelly. Ja... |

In [11]:
```
def check_unique():
    unique_chars = set()

    # Iterate over each element in the 'content' column
    for tweet in train_df['content']:
        unique_chars.update(tweet)

    return unique_chars

unique_chars = check_unique()
print(f"there are {len(unique_chars)} unique characters in train_df['content']: ")
```

```
there are 98 unique characters in train_df['content']:
```

In [12]:
```
#all printable characters
all_letters = set(string.printable)
non_printable_chars = unique_chars - all_letters
print(f"there are {len(non_printable_chars)} not printable characters in train_df['content']: {non_printable_ch
```

```
there are 6 not printable characters in train_df['content']: {'´', 'ï', '½', '\xa0', 'Å', '¿'}
```

In [13]:
```
# we want to remove to following chars
train_df['content'] = train_df['content'].apply(clean_text)

unique_chars = check_unique()
print(f"there are {len(unique_chars)} unique characters in train_df['content']: ")
```

```
there are 92 unique characters in train_df['content']:
```

In [14]:
```
# Split the DataFrame into features (X) and labels (y)
X_train = train_df['content']  # Features (text content)
y_train = train_df['emotion']   # Labels (emotional states)
```

In [15]:
```
# Upload Test Data and clean as well
test_df = pd.read_csv('testEmotions.csv')
test_df['content'] = test_df['content'].apply(clean_text)

# Split the DataFrame into features (X) and labels (y)
X_test = test_df['content']  # Features (text content)
y_test = test_df['emotion']   # Labels (emotional states)
```

**Explanation of our data cleaning and processing**

We removed any characters from the 'content' column that couldn't be transformed into ASCII representation. This standardization of the text data enhances its suitability for processing later in the RNN model.

---

### b) Main (50 pt)

Define 2 models, as requested. Train and eval them.

- Plot the gated model's accuracy and loss (both on train and test sets) as a function of the epochs.
- Plot a confusion matrix

In [16]:
```
#from tutorial - {label: list of content} dict
def categories_dict():
    label_examples_dict = {}
    for index, content in enumerate(train_df['content']):
        label = y_train.iloc[index]

        # Check if the label is already in the dictionary
        if label in label_examples_dict:
            label_examples_dict[label].append(content)
        else:
            label_examples_dict[label] = [content]
    return label_examples_dict
```

In [17]:
```
#all printable characters
all_letters = string.printable
n_letters = len(all_letters)

# Build the category_lines dictionary, a list of names per label
category_lines = categories_dict()

#labels names
all_categories = ['happiness', 'sadness', 'neutral']
n_categories = len(all_categories)
```

In [18]:
```
# Create custom dataset class to load the tokenized data
class CustomDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

## 1. Vanilla RNN

In [19]:
```
class RNN(nn.Module):
    def __init__(self, input_size= 15797, hidden_size=128, output_size=3):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fully_connected = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        hidden = self.init_hidden()
        output, hidden = self.rnn(x.unsqueeze(0), hidden)
        output = output.squeeze(0)
        output = self.fully_connected(output)
        return output, hidden

    def init_hidden(self):
        hidden = torch.zeros(1, 1, self.hidden_size)
        return hidden

n_hidden = 128
```

In [20]:
```
# Initialize LabelEncoder & transform labels
label_encoder = LabelEncoder()

y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
```

In [21]:
```
# Tokenize text using CountVectorizer
vectorizer = CountVectorizer(stop_words='english')
X_train = vectorizer.fit_transform(train_df['content']).toarray()
y_train = train_df['emotion'].values

X_test = vectorizer.transform(test_df['content']).toarray()
y_test = test_df['emotion'].values
```

In [22]:
```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_encoded, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_encoded, dtype=torch.long)
```

In [ ]:

In [23]:
```
# Define DataLoader
train_dataset = CustomDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=1000, shuffle=True)

test_dataset = CustomDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=100, shuffle=True)
```

In [24]:
```
# Instantiate the model
model = RNN(input_size= X_train.shape[1], hidden_size=n_hidden, output_size=n_categories)
```

In [25]:
```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

### train the model

In [26]:
```
# Initialize lists to store metrics
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

max_test_accuracy = 0

num_epochs = 10

# Training loop
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    correct_train = 0
    total_train = 0
    running_loss = 0.0

    for batch_inputs, batch_labels in train_loader:
        optimizer.zero_grad()


        # Forward pass
        outputs, _ = model(batch_inputs)
        loss = criterion(outputs, batch_labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        # Compute training accuracy
        _, predicted = torch.max(outputs, 1)
        correct_train += (predicted == batch_labels).sum().item()
        total_train += batch_labels.size(0)

        running_loss += loss.item()

    # Compute average training loss and accuracy
    train_loss = running_loss / len(train_loader)
    train_accuracy = correct_train / total_train

    # Append to lists
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    # Evaluate on test set
    model.eval()  # Set the model to evaluation mode
    correct_test = 0
    total_test = 0
    test_running_loss = 0.0

    with torch.no_grad():
        for batch_inputs, batch_labels in test_loader:

            outputs, _ = model(batch_inputs)
            loss = criterion(outputs, batch_labels)

            # Compute test accuracy
            _, predicted = torch.max(outputs, 1)
            correct_test += (predicted == batch_labels).sum().item()
            total_test += batch_labels.size(0)

            test_running_loss += loss.item()

    # Compute average test loss and accuracy
    test_loss = test_running_loss / len(test_loader)
    test_accuracy = correct_test / total_test

    # Append to lists
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    if test_accuracy > max_test_accuracy:
        max_test_accuracy = test_accuracy

    # Print progress
    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.4f}, Test
```

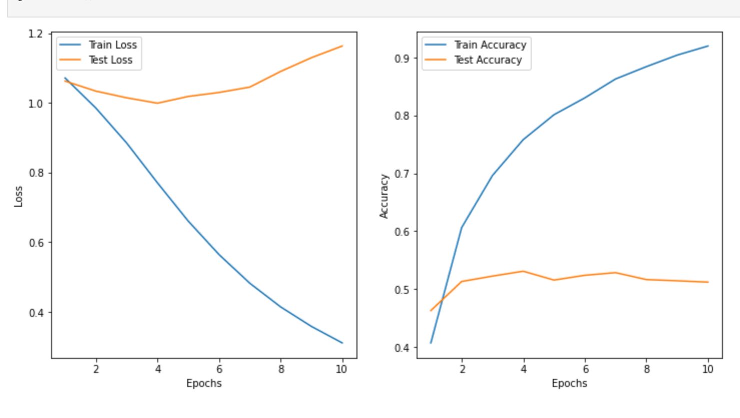
```
Epoch [1/10], Train Loss: 1.0713, Train Acc: 0.4069, Test Loss: 1.0624, Test Acc: 0.4627
Epoch [2/10], Train Loss: 0.9849, Train Acc: 0.6060, Test Loss: 1.0336, Test Acc: 0.5129
Epoch [3/10], Train Loss: 0.8846, Train Acc: 0.6960, Test Loss: 1.0144, Test Acc: 0.5222
Epoch [4/10], Train Loss: 0.7703, Train Acc: 0.7581, Test Loss: 0.9989, Test Acc: 0.5307
Epoch [5/10], Train Loss: 0.6605, Train Acc: 0.8012, Test Loss: 1.0183, Test Acc: 0.5154
Epoch [6/10], Train Loss: 0.5646, Train Acc: 0.8303, Test Loss: 1.0297, Test Acc: 0.5237
Epoch [7/10], Train Loss: 0.4823, Train Acc: 0.8629, Test Loss: 1.0452, Test Acc: 0.5282
Epoch [8/10], Train Loss: 0.4144, Train Acc: 0.8842, Test Loss: 1.0900, Test Acc: 0.5162
Epoch [9/10], Train Loss: 0.3584, Train Acc: 0.9042, Test Loss: 1.1295, Test Acc: 0.5142
Epoch [10/10], Train Loss: 0.3111, Train Acc: 0.9199, Test Loss: 1.1630, Test Acc: 0.5119
```

In [27]:
```
print(f'Reached Max Accuracy of: {max_test_accuracy*100:.2f} on Test')
```

```
Reached Max Accuracy of: 53.07 on Test
```

### plot the results

In [28]:
```
# Plot the accuracy and loss as a function of the epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label='Train Accuracy')
plt.plot(range(1, num_epochs + 1), test_accuracies, label='Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

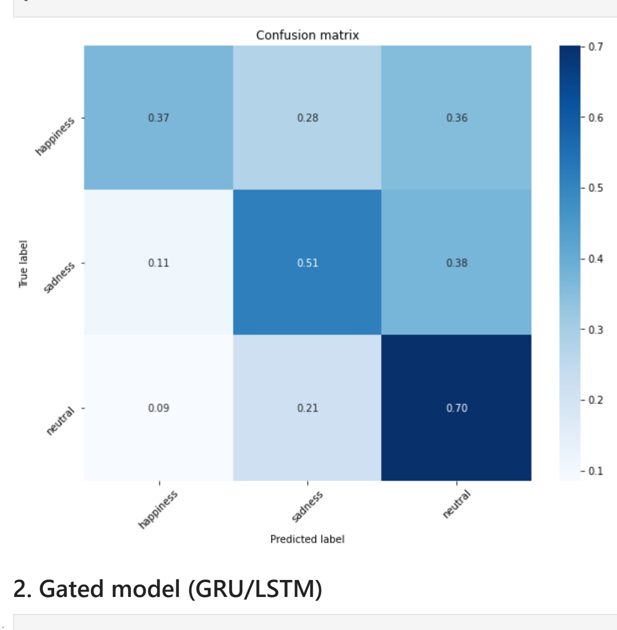



### confusion matrix

In [29]:
```
def categoryFromOutput(output):
    max_val, argmax = output.max(dim=1,  keepdim=True)
    category_i = argmax.item()
    return all_categories[category_i], category_i
```

In [30]:
```
# Function to evaluate the model
def evaluate_model(model, test_loader):
    all_predictions = []
    all_targets = []

    model.eval()  # Set model to evaluation mode
    with torch.no_grad():
        for batch_inputs, batch_labels in test_loader:
            outputs, _ = model(batch_inputs)
            _, predicted = torch.max(outputs, 1)
            all_predictions.extend(predicted.tolist())
            all_targets.extend(batch_labels.tolist())

    return all_predictions, all_targets
```

In [31]:
```
# Evaluate the model
predictions, targets = evaluate_model(model, test_loader)


# Create confusion matrix
confusion = confusion_matrix(targets, predictions)

# Normalize the confusion matrix
confusion = confusion.astype('float') / confusion.sum(axis=1)[:, np.newaxis]

# Plot the confusion matrix with heatmap colors and numbers
plt.figure(figsize=(10, 8))
sns.heatmap(confusion, annot=True, cmap='Blues', fmt='.2f')  # Change the colormap as desired
plt.title('Confusion matrix')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.yticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.show()
```




## 2. Gated model (GRU/LSTM)

In [74]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, rnn_type='lstm'):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])  # Taking the output from the last time step
        return out
```

In [75]:
```
# Initialize LabelEncoder & transform labels

label_encoder = LabelEncoder()

y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
```

In [76]:
```
# Tokenize text using CountVectorizer
vectorizer = CountVectorizer(stop_words='english')
X_train = vectorizer.fit_transform(train_df['content']).toarray()
y_train = train_df['emotion'].values

X_test = vectorizer.transform(test_df['content']).toarray()
y_test = test_df['emotion'].values
```

In [77]:
```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_encoded, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_encoded, dtype=torch.long)
```

In [86]:
```
# Create custom dataset class to load the tokenized data
class CustomDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

In [87]:
```
# Define DataLoader
train_dataset = CustomDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=1000, shuffle=True)

test_dataset = CustomDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=100, shuffle=True)
```

In [88]:
```
# Instantiate the model
model = RNN(input_size= X_train.shape[1], hidden_size=512, output_size=n_categories)
```

In [89]:
```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

### train the model

In [90]:

```python
# Initialize lists to store metrics
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

max_test_accuracy = 0

num_epochs = 10

# Training loop
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    correct_train = 0
    total_train = 0
    running_loss = 0.0

    for batch_inputs, batch_labels in train_loader:
        optimizer.zero_grad()

        batch_inputs = batch_inputs.unsqueeze(1)

        # Forward pass
        outputs = model(batch_inputs)
        loss = criterion(outputs, batch_labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        # Compute training accuracy
        _, predicted = torch.max(outputs, 1)
        correct_train += (predicted == batch_labels).sum().item()
        total_train += batch_labels.size(0)

        running_loss += loss.item()

    # Compute average training loss and accuracy
    train_loss = running_loss / len(train_loader)
    train_accuracy = correct_train / total_train

    # Append to lists
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    # Evaluate on test set
    model.eval()  # Set the model to evaluation mode
    test_predictions = []
    test_labels = []
    correct_test = 0
    total_test = 0
    test_running_loss = 0.0

    with torch.no_grad():
        for batch_inputs, batch_labels in test_loader:

            batch_inputs = batch_inputs.unsqueeze(1)

            outputs = model(batch_inputs)
            loss = criterion(outputs, batch_labels)

            # Compute test accuracy
            _, predicted = torch.max(outputs, 1)
            correct_test += (predicted == batch_labels).sum().item()
            total_test += batch_labels.size(0)
            test_predictions.extend(predicted.tolist())
            test_labels.extend(batch_labels.tolist())

            test_running_loss += loss.item()

    # Compute average test loss and accuracy
    test_loss = test_running_loss / len(test_loader)
    test_accuracy = correct_test / total_test

    # Append to lists
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    if test_accuracy > max_test_accuracy:
        max_test_accuracy = test_accuracy

    # Print progress
    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.4f}, Test
```

```
Epoch [1/10], Train Loss: 1.0768, Train Acc: 0.3967, Test Loss: 1.0673, Test Acc: 0.4153
Epoch [2/10], Train Loss: 0.9862, Train Acc: 0.5695, Test Loss: 1.0296, Test Acc: 0.5386
Epoch [3/10], Train Loss: 0.8341, Train Acc: 0.7472, Test Loss: 1.0114, Test Acc: 0.5243
Epoch [4/10], Train Loss: 0.6623, Train Acc: 0.8105, Test Loss: 1.0267, Test Acc: 0.5233
Epoch [5/10], Train Loss: 0.5116, Train Acc: 0.8462, Test Loss: 1.0812, Test Acc: 0.5177
Epoch [6/10], Train Loss: 0.3994, Train Acc: 0.8822, Test Loss: 1.1485, Test Acc: 0.5154
Epoch [7/10], Train Loss: 0.3157, Train Acc: 0.9088, Test Loss: 1.2281, Test Acc: 0.5121
Epoch [8/10], Train Loss: 0.2515, Train Acc: 0.9327, Test Loss: 1.3176, Test Acc: 0.5007
Epoch [9/10], Train Loss: 0.2046, Train Acc: 0.9482, Test Loss: 1.4152, Test Acc: 0.4929
Epoch [10/10], Train Loss: 0.1682, Train Acc: 0.9582, Test Loss: 1.4900, Test Acc: 0.4929
```

In [91]:

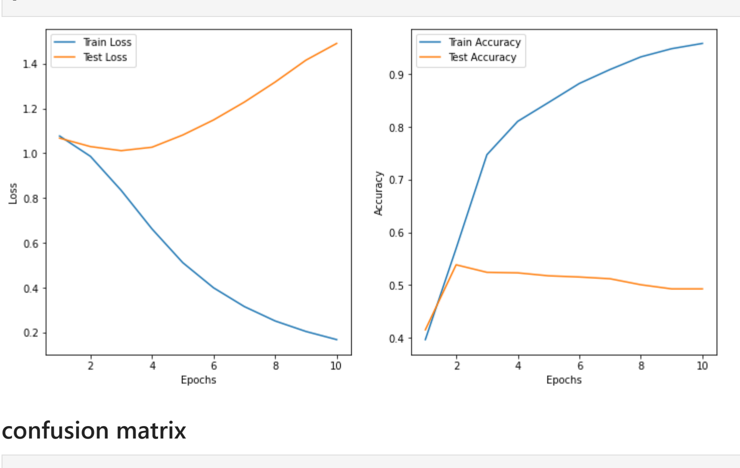
```python
print(f'Reached Max Accuracy of: {max_test_accuracy*100:.2f} on Test')
```

```
Reached Max Accuracy of: 53.86 on Test
```

## plot the results

In [92]:

```python
# Plot the accuracy and loss as a function of the epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label='Train Accuracy')
plt.plot(range(1, num_epochs + 1), test_accuracies, label='Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```




## confusion matrix

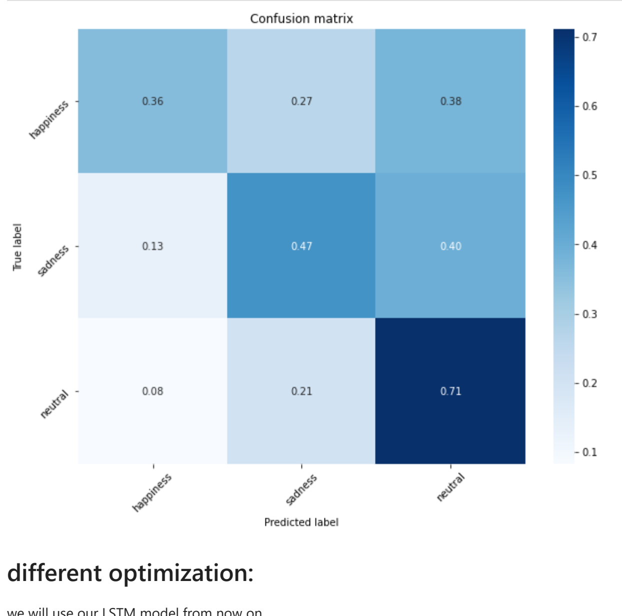
In [93]:

```python
def categoryFromOutput(output):
    max_val, argmax = output.max(dim=1,  keepdim=True)
    category_i = argmax.item()
    return all_categories[category_i], category_i
```

In [ ]:

In [94]:

```python
# Evaluate the model
#predictions, targets = evaluate_model(model, test_loader)


# Create confusion matrix
confusion = confusion_matrix(test_labels, test_predictions)

# Normalize the confusion matrix
confusion = confusion.astype('float') / confusion.sum(axis=1)[:, np.newaxis]

# Plot the confusion matrix with heatmap colors and numbers
plt.figure(figsize=(10, 8))
sns.heatmap(confusion, annot=True, cmap='Blues', fmt='.2f')  # Change the colormap as desired
plt.title('Confusion matrix')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.yticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.show()
```




## different optimization:

we will use our LSTM model from now on

In [ ]:

In [45]:

```python
def train(model):
    # Initialize lists to store metrics
    train_losses = []
    train_accuracies = []
    test_losses = []
    test_accuracies = []

    max_test_accuracy = 0

    num_epochs = 10

    # Training loop
    for epoch in range(num_epochs):
        model.train()  # Set the model to training mode
        correct_train = 0
        total_train = 0
        running_loss = 0.0

        for batch_inputs, batch_labels in train_loader:
            optimizer.zero_grad()

            batch_inputs = batch_inputs.unsqueeze(1)

            # Forward pass
            outputs = model(batch_inputs)
            loss = criterion(outputs, batch_labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

            # Compute training accuracy
            _, predicted = torch.max(outputs, 1)
            correct_train += (predicted == batch_labels).sum().item()
            total_train += batch_labels.size(0)

            running_loss += loss.item()

        # Compute average training loss and accuracy
        train_loss = running_loss / len(train_loader)
        train_accuracy = correct_train / total_train

        # Append to lists
        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)

        # Evaluate on test set
        model.eval()  # Set the model to evaluation mode
        test_predictions = []
        test_labels = []
        correct_test = 0
        total_test = 0
        test_running_loss = 0.0

        with torch.no_grad():
            for batch_inputs, batch_labels in test_loader:

                batch_inputs = batch_inputs.unsqueeze(1)

                outputs = model(batch_inputs)
                loss = criterion(outputs, batch_labels)

                # Compute test accuracy
                _, predicted = torch.max(outputs, 1)
                correct_test += (predicted == batch_labels).sum().item()
                total_test += batch_labels.size(0)
                test_predictions.extend(predicted.tolist())
                test_labels.extend(batch_labels.tolist())

                test_running_loss += loss.item()

        # Compute average test loss and accuracy
        test_loss = test_running_loss / len(test_loader)
        test_accuracy = correct_test / total_test

        # Append to lists
        test_losses.append(test_loss)
        test_accuracies.append(test_accuracy)

        if test_accuracy > max_test_accuracy:
            max_test_accuracy = test_accuracy

        # Print progress
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.4f},
    return train_losses, train_accuracies, test_losses, test_accuracies, max_test_accuracy, test_predictions, t
```

In [46]:

```python
# Instantiate the model
model = RNN(input_size= X_train.shape[1], hidden_size=512, output_size=n_categories)

# define optimizer as SGD
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)

train_losses, train_accuracies, test_losses, test_accuracies, max_acc, test_predictions, test_labels = train(mo
```

```
Epoch [1/10], Train Loss: 1.0914, Train Acc: 0.3613, Test Loss: 1.0902, Test Acc: 0.3736
Epoch [2/10], Train Loss: 1.0877, Train Acc: 0.3749, Test Loss: 1.0862, Test Acc: 0.3738
Epoch [3/10], Train Loss: 1.0829, Train Acc: 0.3851, Test Loss: 1.0845, Test Acc: 0.3738
Epoch [4/10], Train Loss: 1.0783, Train Acc: 0.3902, Test Loss: 1.0818, Test Acc: 0.4222
Epoch [5/10], Train Loss: 1.0728, Train Acc: 0.4666, Test Loss: 1.0782, Test Acc: 0.3924
Epoch [6/10], Train Loss: 1.0660, Train Acc: 0.4475, Test Loss: 1.0741, Test Acc: 0.4712
Epoch [7/10], Train Loss: 1.0573, Train Acc: 0.5046, Test Loss: 1.0681, Test Acc: 0.4794
Epoch [8/10], Train Loss: 1.0466, Train Acc: 0.5103, Test Loss: 1.0617, Test Acc: 0.4769
Epoch [9/10], Train Loss: 1.0325, Train Acc: 0.5122, Test Loss: 1.0536, Test Acc: 0.4703
Epoch [10/10], Train Loss: 1.0172, Train Acc: 0.5131, Test Loss: 1.0445, Test Acc: 0.4869
```

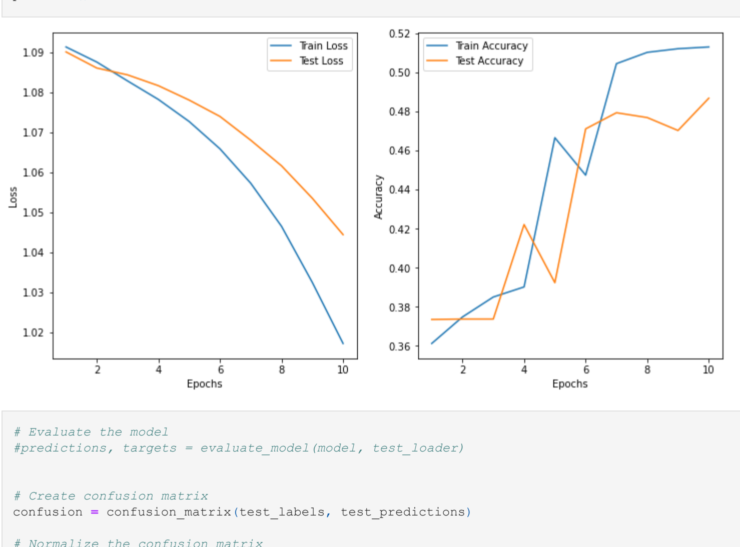
In [47]:

```python
print(f'Reached Max Accuracy of: {max_acc*100:.2f} on Test')
```

```
Reached Max Accuracy of: 48.69 on Test
```

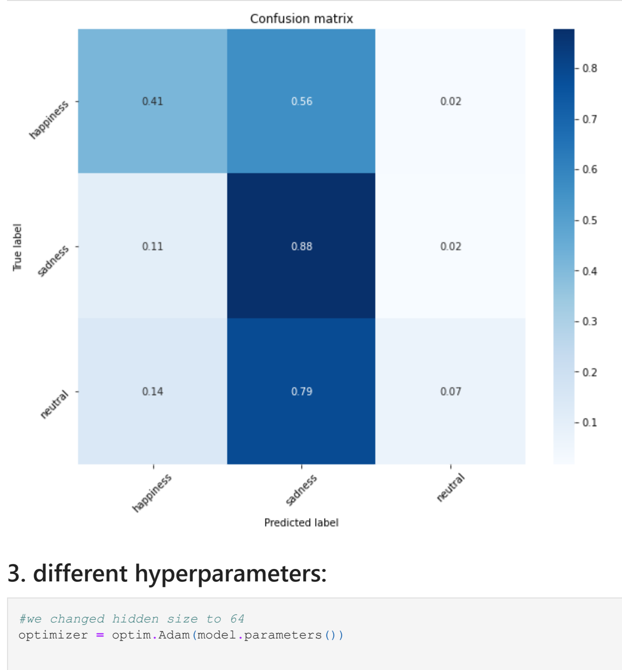
In [48]:

```python
# Plot the accuracy and loss as a function of the epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label='Train Accuracy')
plt.plot(range(1, num_epochs + 1), test_accuracies, label='Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```




In [49]:

```python
# Evaluate the model
#predictions, targets = evaluate_model(model, test_loader)


# Create confusion matrix
confusion = confusion_matrix(test_labels, test_predictions)

# Normalize the confusion matrix
confusion = confusion.astype('float') / confusion.sum(axis=1)[:, np.newaxis]

# Plot the confusion matrix with heatmap colors and numbers
plt.figure(figsize=(10, 8))
sns.heatmap(confusion, annot=True, cmap='Blues', fmt='.2f')  # Change the colormap as desired
plt.title('Confusion matrix')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.yticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.show()
```




## 3. different hyperparameters:

In [70]:

```python
#we changed hidden size to 64
optimizer = optim.Adam(model.parameters())


model = RNN(input_size= X_train.shape[1], hidden_size=64, output_size=n_categories)
train_losses, train_accuracies, test_losses, test_accuracies, max_acc, test_predictions, test_labels = train(mo
```

```
Epoch [1/10], Train Loss: 1.0972, Train Acc: 0.3582, Test Loss: 1.0976, Test Acc: 0.3579
Epoch [2/10], Train Loss: 1.0973, Train Acc: 0.3582, Test Loss: 1.0977, Test Acc: 0.3579
Epoch [3/10], Train Loss: 1.0973, Train Acc: 0.3582, Test Loss: 1.0979, Test Acc: 0.3579
Epoch [4/10], Train Loss: 1.0974, Train Acc: 0.3582, Test Loss: 1.0977, Test Acc: 0.3579
Epoch [5/10], Train Loss: 1.0974, Train Acc: 0.3582, Test Loss: 1.0977, Test Acc: 0.3579
Epoch [6/10], Train Loss: 1.0972, Train Acc: 0.3582, Test Loss: 1.0974, Test Acc: 0.3579
Epoch [7/10], Train Loss: 1.0973, Train Acc: 0.3582, Test Loss: 1.0978, Test Acc: 0.3579
Epoch [8/10], Train Loss: 1.0972, Train Acc: 0.3582, Test Loss: 1.0977, Test Acc: 0.3579
Epoch [9/10], Train Loss: 1.0975, Train Acc: 0.3582, Test Loss: 1.0976, Test Acc: 0.3579
Epoch [10/10], Train Loss: 1.0973, Train Acc: 0.3582, Test Loss: 1.0978, Test Acc: 0.3579
```

In [71]:

```python
print(f'Reached Max Accuracy of: {max_acc*100:.2f} on Test')
```

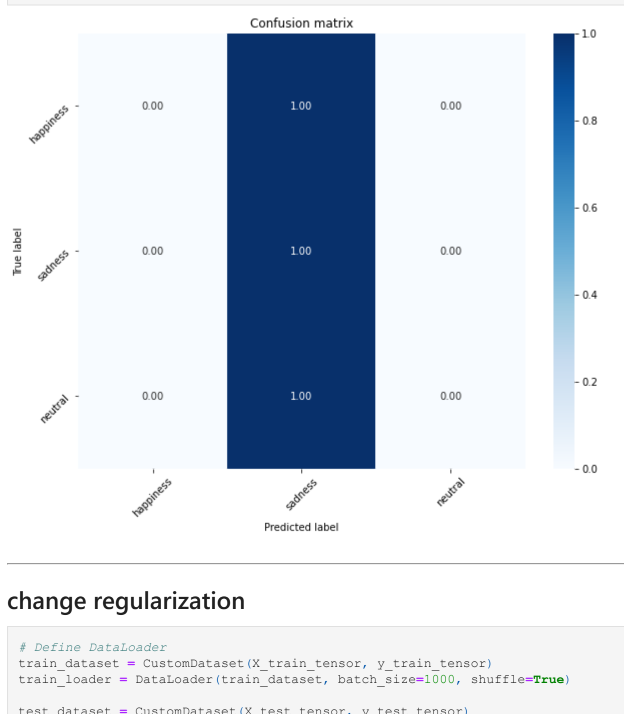
```
Reached Max Accuracy of: 35.79 on Test
```

In [72]:

```python
# Plot the accuracy and loss as a function of the epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label='Train Accuracy')
plt.plot(range(1, num_epochs + 1), test_accuracies, label='Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```


In [73]:

```python
# Evaluate the model
#predictions, targets = evaluate_model(model, test_loader)


# Create confusion matrix
confusion = confusion_matrix(test_labels, test_predictions)

# Normalize the confusion matrix
confusion = confusion.astype('float') / confusion.sum(axis=1)[:, np.newaxis]

# Plot the confusion matrix with heatmap colors and numbers
plt.figure(figsize=(10, 8))
sns.heatmap(confusion, annot=True, cmap='Blues', fmt='.2f')  # Change the colormap as desired
plt.title('Confusion matrix')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.yticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.show()
```




---

## change regularization

In [59]:

```python
# Define DataLoader
train_dataset = CustomDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=1000, shuffle=True)

test_dataset = CustomDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=100, shuffle=True)
```

In [60]:

```python
class RNNWithDropout(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, rnn_type='lstm', dropout=0.0):
        super(RNNWithDropout, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_type = rnn_type

        if rnn_type == 'lstm':
            self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        elif rnn_type == 'gru':
            self.rnn = nn.GRU(input_size, hidden_size, batch_first=True)
        else:
            raise ValueError("Unsupported RNN type. Choose 'lstm' or 'gru'.")

        self.fc = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.dropout(out)  # Applying dropout regularization
        out = self.fc(out[:, -1, :])  # Taking the output from the last time step
        return out
```

In [61]:

```python
model = RNNWithDropout(input_size= X_train.shape[1], hidden_size=64, output_size=n_categories, rnn_type='lstm',
optimizer = optim.Adam(model.parameters())

train_losses, train_accuracies, test_losses, test_accuracies, max_acc, test_predictions, test_labels = train(mo
```

```
Epoch [1/10], Train Loss: 1.0873, Train Acc: 0.3751, Test Loss: 1.0857, Test Acc: 0.3732
Epoch [2/10], Train Loss: 1.0674, Train Acc: 0.4079, Test Loss: 1.0768, Test Acc: 0.3874
Epoch [3/10], Train Loss: 1.0385, Train Acc: 0.4929, Test Loss: 1.0644, Test Acc: 0.4443
Epoch [4/10], Train Loss: 0.9940, Train Acc: 0.6027, Test Loss: 1.0486, Test Acc: 0.5127
Epoch [5/10], Train Loss: 0.9308, Train Acc: 0.6893, Test Loss: 1.0302, Test Acc: 0.5336
Epoch [6/10], Train Loss: 0.8529, Train Acc: 0.7503, Test Loss: 1.0161, Test Acc: 0.5371
Epoch [7/10], Train Loss: 0.7676, Train Acc: 0.7857, Test Loss: 1.0127, Test Acc: 0.5243
Epoch [8/10], Train Loss: 0.6852, Train Acc: 0.8087, Test Loss: 1.0119, Test Acc: 0.5282
Epoch [9/10], Train Loss: 0.6102, Train Acc: 0.8300, Test Loss: 1.0287, Test Acc: 0.5193
Epoch [10/10], Train Loss: 0.5423, Train Acc: 0.8482, Test Loss: 1.0437, Test Acc: 0.5187
```

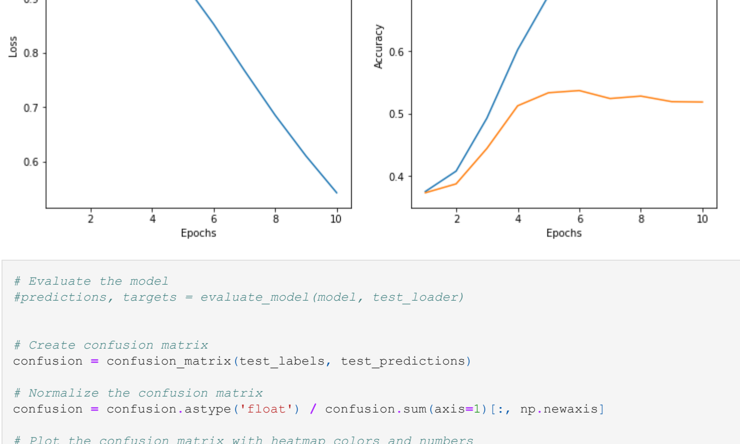
In [62]:

```python
print(f'Reached Max Accuracy of: {max_acc*100:.2f} on Test')
```

```
Reached Max Accuracy of: 53.71 on Test
```

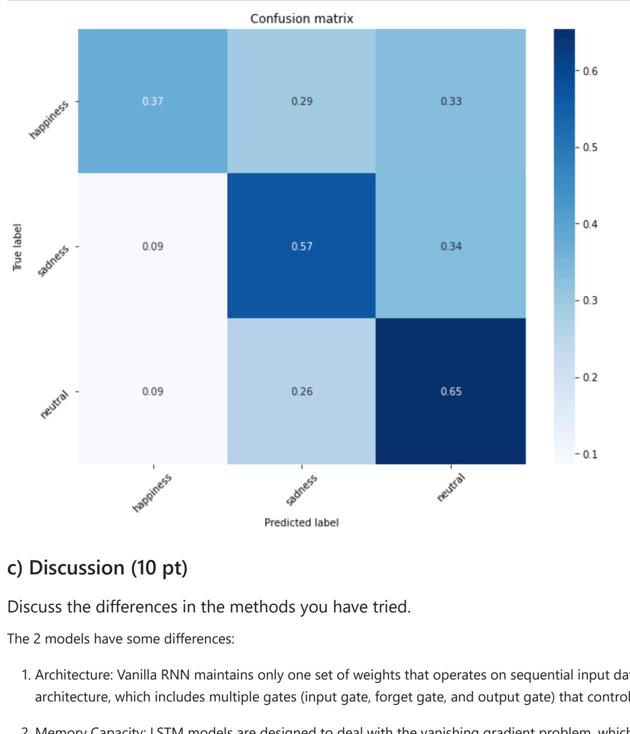
In [63]:

```python
# Plot the accuracy and loss as a function of the epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label='Train Accuracy')
plt.plot(range(1, num_epochs + 1), test_accuracies, label='Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```




In [64]:

```python
# Evaluate the model
#predictions, targets = evaluate_model(model, test_loader)


# Create confusion matrix
confusion = confusion_matrix(test_labels, test_predictions)

# Normalize the confusion matrix
confusion = confusion.astype('float') / confusion.sum(axis=1)[:, np.newaxis]

# Plot the confusion matrix with heatmap colors and numbers
plt.figure(figsize=(10, 8))
sns.heatmap(confusion, annot=True, cmap='Blues', fmt='.2f')  # Change the colormap as desired
plt.title('Confusion matrix')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.yticks(np.arange(len(all_categories)) + 0.5, all_categories, rotation=45)
plt.show()
```




### c) Discussion (10 pt)

Discuss the differences in the methods you have tried.

The 2 models have some differences:

1. Architecture: Vanilla RNN maintains only one set of weights that operates on sequential input data, while LSTMs have a more complex architecture, which includes multiple gates (input gate, forget gate, and output gate) that control the flow of information
2. Memory Capacity: LSTM models are designed to deal with the vanishing gradient problem, which is common in Vanilla RNNs, by maintain information and dependencies over longer periods.
3. Training: vanilla RNNs may struggle to converge on long-term dependencies capturing tasks due to the vanishing gradient problem. LSTMs are mostly achieve better performances on tasks with long sequences processing.
4. Training complexity: As we saw in our model, LSTMs require more resources and time than the Vanilla RNN model.

In the data we recieved, we didn't see any big difference in our results of each model. We believe the reason for that is the sentences we trained our models on were too short for significant long term processing.

We also noticed that the LSTM training took much more time than the Vanilla RNN model, as explained above.

Different hyperparameters - we tried to use a lower hidden size in our model. Smaller hidden size means fewer parameters learned in the hidden layer, which reduces the model's ability to learn complex patterns in the data.

Different optimizer and regularization - we tried to use sgd instead of adam. It seems the model did a little worse than the original LSTM model, but still gave us good enough score. As for the regularization, we tried to use dropout regularization, in order to reduce overfitting. The regularized model gave us almost the same result. That could happen because our data and model are to simple, so it could cause underfitting, and in this case adding regularization won't help.

---