Twitter Emoji Prediction Project

Github project: https://github.com/aparnnaH/Twitter-Emoji-Prediction-Project (https://github.com/aparnnaH/Twitter-Emoji-Prediction-Project)

Video Link: https://youtu.be/ONjcuhJCyF4?si=QUwlnXrXgJQYWOoL (https://youtu.be/ONjcuhJCyF4?si=QUwlnXrXgJQYWOoL)

This project uses the Twitter Emoji Prediction dataset from Kaggle, which consists of tweets paired with emojis that reflect the context or sentiment of each post. Although the precise data collection method is not specified, the dataset was likely gathered using the Twitter API and preprocessed to emphasize the relationship between tweet text and emoji labels. It captures informal language, slang, and typical emoji usage found on social media.

Files used:

Dataset: https://www.kaggle.com/datasets/hariharasudhanas/twitter-emoji-prediction/data?select=Train.csv (https://www.kaggle.com/datasets/hariharasudhanas/twitter-emoji-prediction/data?select=Train.csv)

- train.csv: for model training and validation
- test.csv: for final evaluation on unseen data
- Mapping.csv: provides the mapping between emoji symbols and their numeric labels

This dataset is well-suited for training deep learning models that predict the most appropriate emoji for a given tweet.

Identifying a Deep Learning Problem

This task is a multi-class text classification problem: given a tweet, predict the correct emoji from a fixed set of labels. To address this, three distinct deep learning architectures are explored, each implemented in a separate Jupyter notebook:

- LSTM: Captures sequential structure and temporal dependencies in the tweet text.
- GRU: A simpler alternative to LSTM with faster training while still modeling sequences effectively.
- TextCNN: Uses convolutional layers to extract local n-gram features and position-invariant text patterns.

The objective is to compare these models in terms of prediction accuracy, training time, and ability to generalize. The comparison aims to identify which architecture is best suited for predicting emojis based on short, informal text inputs like tweets.

LSTM Model

In [3]:
```
# pip install wordcloud
```

In [6]:
```
# pip install emoji
```

In [14]:
```
import logging
import warnings
# Suppress matplotlib font_manager warnings
logging.getLogger('matplotlib.font_manager').setLevel(logging.ERROR)
# Also suppress UserWarnings related to fonts
warnings.filterwarnings("ignore", message="findfont: Font family 'Apple Color Emoji' not found.")
```

In [15]:
```
# Imports and Setup
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from torchtext.data.utils import get_tokenizer
from wordcloud import WordCloud
from collections import Counter
import emoji
import re
```

```
2025-06-20 17:24:59.420310: I tensorflow/core/platform/cpu_feature_guard.
cc:182] This TensorFlow binary is optimized to use available CPU instruct
ions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebu
ild TensorFlow with the appropriate compiler flags.
```

In [16]:
```
# Load the training dataset containing text and corresponding emoji label
s
train = pd.read_csv("train.csv")

# Load the mapping file that maps numeric labels to actual emoticons
mapping = pd.read_csv('Mapping.csv')

# Create a dictionary to map numeric labels to their corresponding emotic
ons
# 'number' column contains label IDs, 'emoticons' column contains emoji s
trings
emoji_map = dict(zip(mapping["number"], mapping["emoticons"]))
inv_map = {v: k for k, v in emoji_map.items()}
```

In [13]:
```
train.head(4)
```

Out[13]:

	Unnamed: 0	TEXT	Label
0	0	Vacation wasted ! #vacation2017 #photobomb #ti...	0
1	1	Oh Wynwood, you’re so funny! : @user #Wynwood ...	1
2	2	Been friends since 7th grade. Look at us now w...	2
3	3	This is what it looks like when someone loves ...	3

In [10]:
```
test_data = pd.read_csv('Test.csv')
test_data.head(4)
```

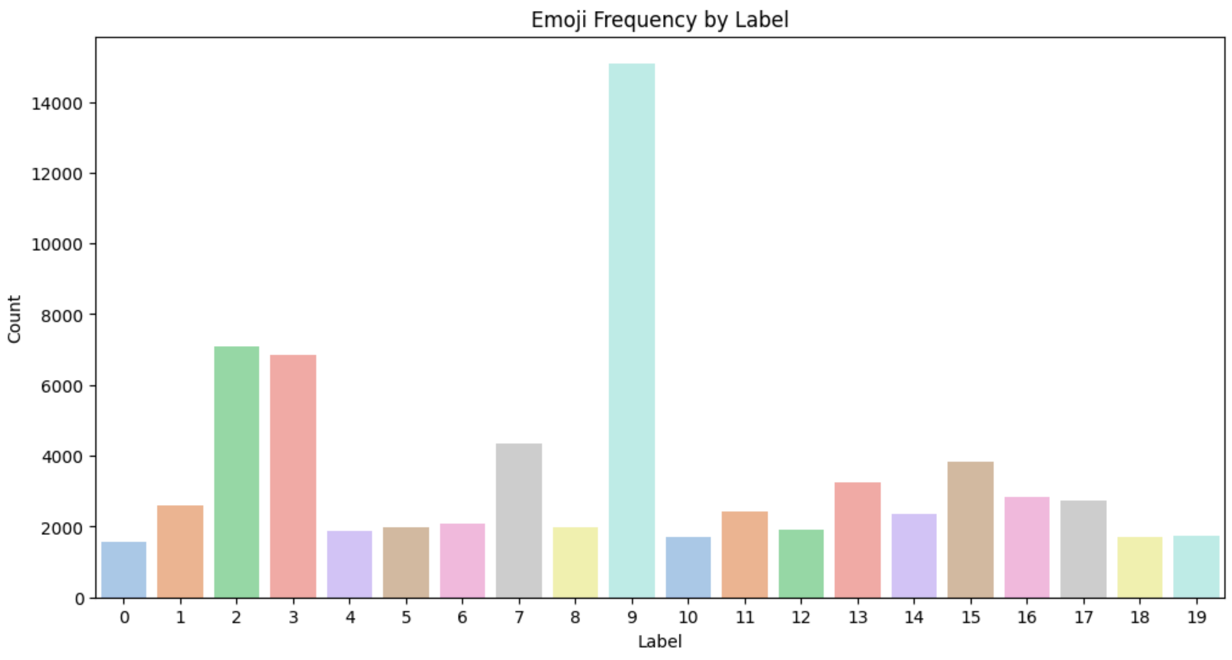
Out[10]:

	Unnamed: 0	id	TEXT
0	0	0	Thought this was cool...#Repost (get_repost) · ・...
1	1	1	Happy 4th! Corte madera parade. #everytownusa ...
2	2	2	Luv. Or at least something close to it. @ Unio...
3	3	3	There's a slice of pie under that whipped crea...

Preparing Text Data for Modeling

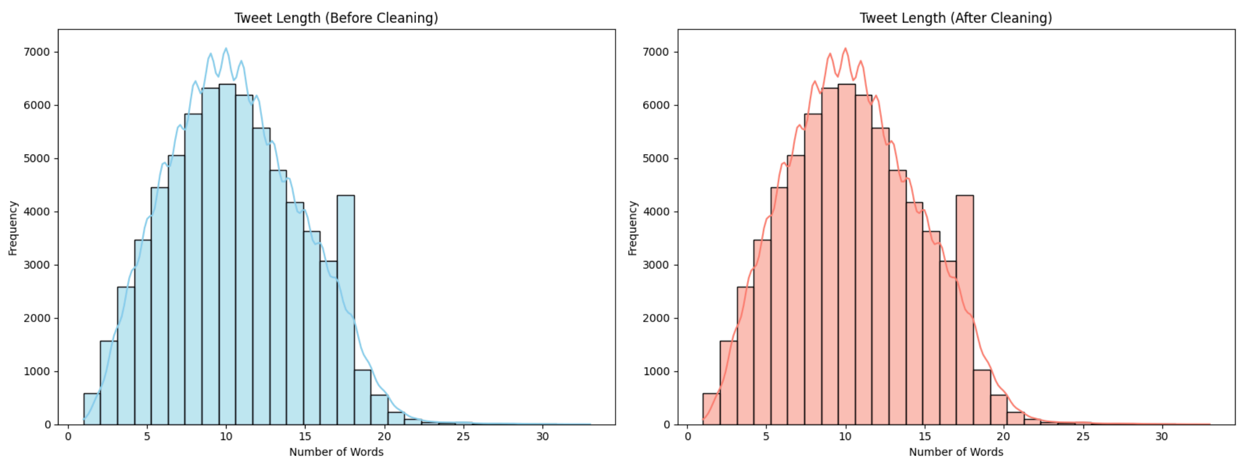
In [21]:
```
# Store original train data
train_org = train

# Clean Text Function
def clean_text(text):
    # Remove URLs
    text = re.sub(r"http\S+", "", text)
    # Remove @mentions
    text = re.sub(r"@\w+", "", text)
    # Remove special characters except basic punctuation
    text = re.sub(r"[^\w\s.,!?]", "", text)
    # Normalize spaces and convert to lowercase
    text = re.sub(r"\s+", " ", text).strip().lower()
    return text

# Apply text cleaning to all entries in the TEXT column
train["TEXT"] = train["TEXT"].astype(str).apply(clean_text)

# Tokenize and encode
# Initialize a basic english tokenizer
tokenizer = get_tokenizer("basic_english")

# Tokenize each cleaned tweet
train["tokens"] = train["TEXT"].apply(tokenizer)

# Build vocabulary from all unique tokens
vocab = set(token for tokens in train.tokens for token in tokens)

# Assign an index to each token
vocab_dict = {word: idx + 2 for idx, word in enumerate(vocab)}
vocab_dict["<pad>"] = 0  # padding token
vocab_dict["<unk>"] = 1  # unknown token

# Function to encode tokens into indices using vocab_dict
def encode(tokens):
    return [vocab_dict.get(t, 1) for t in tokens]

# Apply encoding to each list of tokens
train["encoded"] = train["tokens"].apply(encode)

MAX_LEN = 50
def pad(seq):
    return seq[:MAX_LEN] + [0] * (MAX_LEN - len(seq))

train["padded"] = train["encoded"].apply(pad)
```

```
In [22]: # Plot emoji frequency with numeric labels
         plt.figure(figsize=(12,6))
         sns.countplot(x='Label', data=train, palette='pastel')
         plt.title("Emoji Frequency by Label")
         plt.xlabel("Label")
         plt.ylabel("Count")
         plt.show()

         # Legend mapping label to emoji
         print("Emoji Label Mapping:", ', '.join([f"{label}: {emoji}" for label, e
         moji in emoji_map.items()]))
```

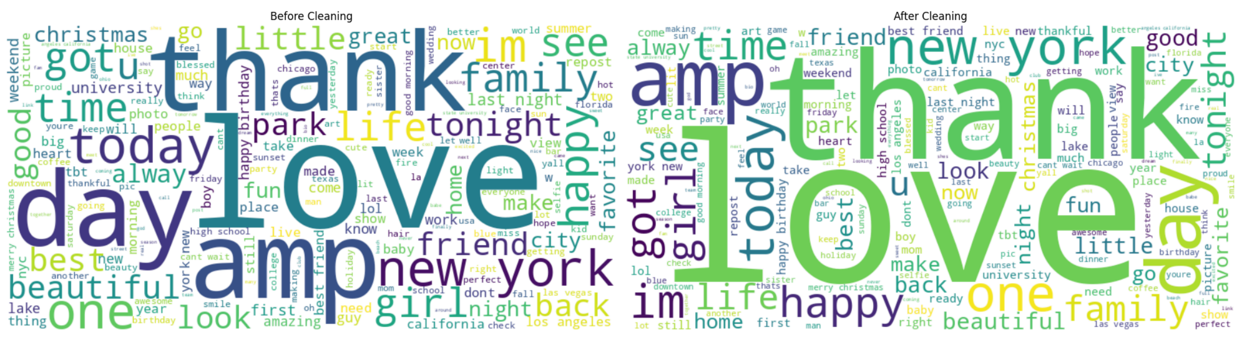


```
Emoji Label Mapping: 0: 😜, 1: 📸, 2: 😍, 3: 😂, 4: 😉, 5: 🎄, 6: 📷, 7:
🔥, 8: 😘, 9: ❤, 10: 😁, 11: 🇺🇸, 12: ☀, 13: ✨, 14: 💙, 15: 💕, 16: 😎,
17: 😊, 18: 💜, 19: 💯
```

```
In [23]: import matplotlib.pyplot as plt
         import seaborn as sns

         # Calculate tweet lengths
         train_org['text_length'] = train_org['TEXT'].apply(lambda x: len(str(x).s
         plit()))
         train['text_length'] = train['TEXT'].apply(lambda x: len(str(x).split()))

         # Plot side by side
         fig, axs = plt.subplots(1, 2, figsize=(16, 6))

         # Before Cleaning
         sns.histplot(train_org['text_length'], bins=30, kde=True, ax=axs[0], colo
         r='skyblue')
         axs[0].set_title("Tweet Length (Before Cleaning)")
         axs[0].set_xlabel("Number of Words")
         axs[0].set_ylabel("Frequency")

         # After Cleaning
         sns.histplot(train['text_length'], bins=30, kde=True, ax=axs[1], color='s
         almon')
         axs[1].set_title("Tweet Length (After Cleaning)")
         axs[1].set_xlabel("Number of Words")
         axs[1].set_ylabel("Frequency")

         plt.tight_layout()
         plt.show()
```



```
In [24]: # Before cleaning
text_before = " ".join(train_org["TEXT"].astype(str))

# After cleaning
text_after = " ".join(train["TEXT"].astype(str))

# Create WordClouds
wordcloud_before = WordCloud(width=800, height=400, background_color='whi
te').generate(text_before)
wordcloud_after = WordCloud(width=800, height=400, background_color='whit
e').generate(text_after)

# Plot side by side
fig, axs = plt.subplots(1, 2, figsize=(20, 8))

axs[0].imshow(wordcloud_before, interpolation='bilinear')
axs[0].axis("off")
axs[0].set_title("Before Cleaning")

axs[1].imshow(wordcloud_after, interpolation='bilinear')
axs[1].axis("off")
axs[1].set_title("After Cleaning")

plt.tight_layout()
plt.show()
```



Building Dataset and DataLoader Objects

```
In [25]: # Preprocessing
texts = train.TEXT.astype(str).tolist()
labels = train.Label.astype(int).tolist()
num_classes = len(set(labels))

max_words = 10000
max_len = 40

tokenizer = Tokenizer(num_words=max_words, oov_token="<UNK>")
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
X = pad_sequences(sequences, maxlen=max_len)
y = tf.keras.utils.to_categorical(labels, num_classes=num_classes)
```

Defining and Training the LSTM model

```
In [26]: # Train and test split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, ra
ndom_state=42)
```

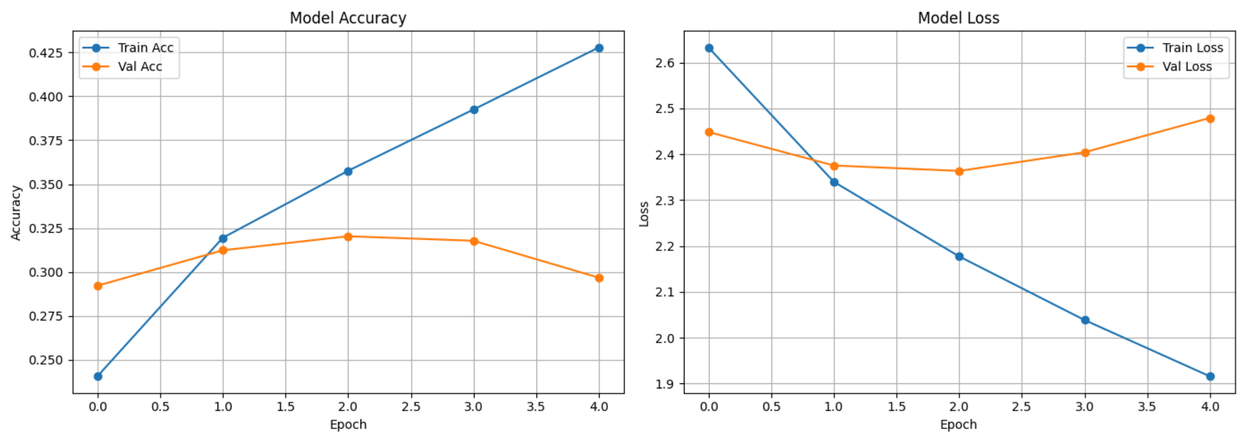
```
In [27]: # Model
model = Sequential([
    Embedding(max_words, 50, input_length=max_len),
    LSTM(64, return_sequences=False),
    Dense(num_classes, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=
['accuracy'])
```

In [28]:
```python
# Train the model
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=5,
    batch_size=128
)
```

```
Epoch 1/5
438/438 [==============================] - 15s 29ms/step - loss: 2.6329 -
accuracy: 0.2407 - val_loss: 2.4484 - val_accuracy: 0.2922
Epoch 2/5
438/438 [==============================] - 13s 29ms/step - loss: 2.3398 -
accuracy: 0.3195 - val_loss: 2.3754 - val_accuracy: 0.3124
Epoch 3/5
438/438 [==============================] - 13s 30ms/step - loss: 2.1771 -
accuracy: 0.3576 - val_loss: 2.3635 - val_accuracy: 0.3204
Epoch 4/5
438/438 [==============================] - 14s 32ms/step - loss: 2.0384 -
accuracy: 0.3924 - val_loss: 2.4042 - val_accuracy: 0.3178
Epoch 5/5
438/438 [==============================] - 13s 30ms/step - loss: 1.9156 -
accuracy: 0.4278 - val_loss: 2.4792 - val_accuracy: 0.2968
```

The model's training accuracy improves steadily from 25% to 45%, showing it's learning. However, validation accuracy only slightly improves, peaking around 34%. The gap between training and validation shows overfitting better on training data.

In [29]:
```python
# Accuracy & Loss Plots
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Acc', marker='o')
plt.plot(history.history['val_accuracy'], label='Val Acc', marker='o')
plt.title("Model Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss', marker='o')
plt.plot(history.history['val_loss'], label='Val Loss', marker='o')
plt.title("Model Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



Evaluation

In [51]:
```
# Classification Report
print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=list(emoji_map.values())))
```

```
Classification Report:
              precision    recall  f1-score   support

           😜       0.00      0.00      0.00       282
           📸       0.22      0.26      0.24       531
           😍       0.22      0.28      0.25      1408
           😂       0.33      0.47      0.39      1384
           😉       0.07      0.01      0.01       372
           🎄       0.56      0.72      0.63       387
           📷       0.22      0.11      0.14       431
           🔥       0.37      0.45      0.41       875
           😘       0.25      0.01      0.02       377
           ❤       0.46      0.70      0.55      3049
           😁       0.00      0.00      0.00       355
          🇺🇸       0.43      0.54      0.48       509
           ☀       0.39      0.31      0.34       370
           ✨       0.19      0.21      0.20       644
           💙       0.24      0.04      0.07       466
           💕       0.16      0.06      0.09       728
           😎       0.13      0.20      0.16       587
           😊       0.12      0.04      0.06       531
           💜       0.00      0.00      0.00       358
           💯       0.22      0.03      0.06       356

    accuracy                           0.34     14000
   macro avg       0.23      0.22      0.20     14000
weighted avg       0.28      0.34      0.29     14000
```

The classification report reveals that while the model achieves a moderate accuracy of 34%, its performance varies widely across different emoji classes. Emojis like ❤️ , 🎄 , 🇺🇸 , and 😂 are predicted with relatively high precision and recall, likely due to their strong, consistent contextual signals in the data. On the other hand, several emojis such as 😜 , 😁 , and 💜 show zero precision and recall, meaning the model failed to predict them altogether. This suggests issues with class imbalance, insufficient training examples, or overlapping usage contexts. The low macro F1-score of 0.20 highlights that many classes are underperforming. Overall, while the model captures common emojis fairly well, there's significant room for improvement in handling rarer or more ambiguous ones.

In [9]:
```
# Load the test dataset
test_data = pd.read_csv('Test.csv')
print(test_data.columns) # check column names

# Select the first 5 tweets
test_tweets = test_data['TEXT'].tolist()[:5]

# Predict emojis for each tweet and visualize the top 3 predictions
for tweet in test_tweets:
    pred = predict_emoji(tweet)
    # Get indices of top 3 predictions and probabilities
    top_indices = pred.argsort()[-3:][::-1]
    top_probs = [pred[i] for i in top_indices]

    # Print emoji predictions and confidence scores
    print("Top predicted emojis and confidence:")
    for i, idx in enumerate(top_indices):
        print(f"{i+1}. {emoji_map[idx]} – {top_probs[i]:.2f}")
    print("\n")
```

```
Thought this was cool...#Repost (get_repost)・・・Colorview. by shay_image
s…
Top predicted emojis and confidence:
 1. 😎 – 0.35
 2. 📸 – 0.13
 3. 🔥 – 0.11

Happy 4th! Corte madera parade. #everytownusa #merica @ Perry's on…
Top predicted emojis and confidence:
 1. 🇺🇸 – 0.99
 2. ✨ – 0.00
 3. 💙 – 0.00

Luv. Or at least something close to it. @ Union Hill, Richmond, Virginia
Top predicted emojis and confidence:
 1. 😂 – 0.14
 2. 😉 – 0.12
 3. 💯 – 0.10
```

Conculsion

The LSTM model performs well on tweets with clear and specific context, such as those mentioning holidays, food, or expressions of gratitude. It shows strong performance on structured language and can confidently predict emojis in cases like 🇺🇸 for a 4th of July tweet. However, it tends to struggle with tweets that contain slang, casual phrasing, or ambiguous meaning. While some predictions are precise, others include minor noise, though they often remain contextually relevant. Improving the model's handling of informal language could lead to further gains in accuracy.

TextCNN Model

In [3]:
```
# pip install torchtext
```

In [4]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchtext.data.utils import get_tokenizer
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
import re
from wordcloud import WordCloud
```

In [5]:
```
# Load the training dataset containing text and corresponding emoji label
s
train = pd.read_csv("train.csv")

# Load the mapping file that maps numeric labels to actual emoticons
mapping = pd.read_csv('Mapping.csv')

# Create a dictionary to map numeric labels to their corresponding emotic
ons
# 'number' column contains label IDs, 'emoticons' column contains emoji s
trings
emoji_map = dict(zip(mapping["number"], mapping["emoticons"]))
```

Preparing Text Data for Modeling

In [8]:
```python
# Clean Text Function
def clean_text(text):
    # Remove URLs
    text = re.sub(r"http\S+", "", text)
    # Remove @mentions
    text = re.sub(r"@\w+", "", text)
    # Remove special characters except basic punctuation
    text = re.sub(r"[^\w\s.,!?]", "", text)
    # Normalize spaces and convert to lowercase
    text = re.sub(r"\s+", " ", text).strip().lower()
    return text

# Apply text cleaning to all entries in the TEXT column
train["TEXT"] = train["TEXT"].astype(str).apply(clean_text)

# Tokenize and encode
# Initialize a basic english tokenizer
tokenizer = get_tokenizer("basic_english")

# Tokenize each cleaned tweet
train["tokens"] = train["TEXT"].apply(tokenizer)

# Build vocabulary from all unique tokens
vocab = set(token for tokens in train.tokens for token in tokens)

# Assign an index to each token
vocab_dict = {word: idx + 2 for idx, word in enumerate(vocab)}
vocab_dict["<pad>"] = 0  # padding token
vocab_dict["<unk>"] = 1  # unknown token

# Function to encode tokens into indices using vocab_dict
def encode(tokens):
    return [vocab_dict.get(t, 1) for t in tokens]

# Apply encoding to each list of tokens
train["encoded"] = train["tokens"].apply(encode)

MAX_LEN = 50
def pad(seq):
    return seq[:MAX_LEN] + [0] * (MAX_LEN - len(seq))

train["padded"] = train["encoded"].apply(pad)
```

Building Dataset and DataLoader Objects

In [12]:
```
# custom dataset class for tweets
class TweetDataset(Dataset):
    def __init__(self, texts, labels):
        # Convert input sequences and labels to torch tensors
        self.texts = torch.tensor(texts, dtype=torch.long)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        # total number of samples
        return len(self.labels)

    def __getitem__(self, idx):
        # single sample by index
        return self.texts[idx], self.labels[idx]

# Split data into training and validation sets (80% train, 20% val)
X_train, X_val, y_train, y_val = train_test_split(
    train.padded.tolist(),
    train.Label.tolist(),
    test_size=0.2
)

# dataset instances for training and validation
train_ds = TweetDataset(X_train, y_train)
val_ds = TweetDataset(X_val, y_val)

# dataLoader objects
train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=64)
```

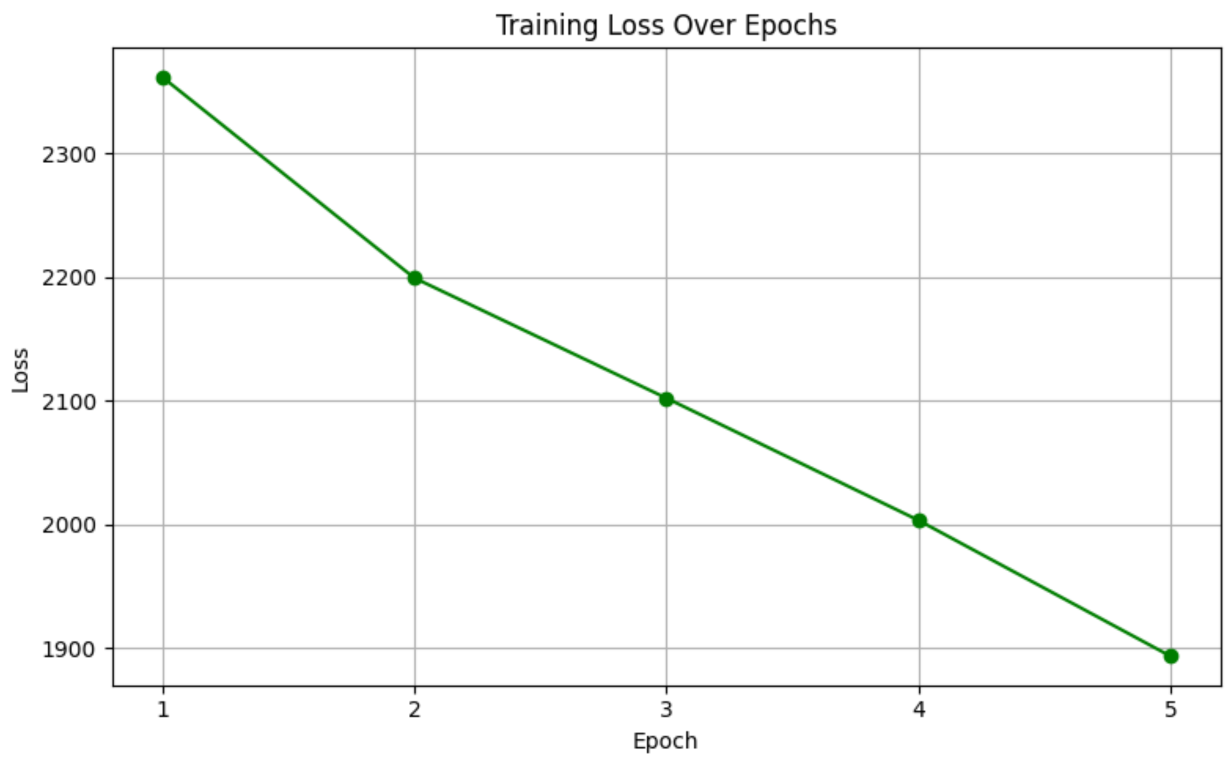
Defining and Training the TextCNN Model

In [14]:
```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_classes, kernel_sizes=
[3, 4, 5], num_filters=100, dropout=0.5):
        super().__init__()
        # Embedding layer maps word indices to dense vectors
        self.embedding = nn.Embedding(vocab_size, embed_dim)

        # Convolutional layers with multiple kernel sizes for capturing d
ifferent n-gram features
        self.convs = nn.ModuleList([
            nn.Conv1d(in_channels=embed_dim, out_channels=num_filters, ke
rnel_size=k)
            for k in kernel_sizes
        ])

        # Dropout layer to reduce overfitting
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(num_filters * len(kernel_sizes), num_classes)

    def forward(self, x):
        # Convert word indices to embeddings
        x = self.embedding(x)
        # Rearrange to (B, E, L)
        x = x.permute(0, 2, 1)
        # Apply convolution + ReLU activation
        x = [F.relu(conv(x)) for conv in self.convs]

        # Max pool over the time dimension to get fixed-size vector
        x = [F.max_pool1d(c, kernel_size=c.size(2)).squeeze(2) for c in
x]
        # Concatenate all outputs
        x = torch.cat(x, dim=1)

        return self.fc(self.dropout(x))
```

In [15]:
```
# Set device to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize TextCNN model
model = TextCNN(vocab_size=len(vocab_dict), embed_dim=100, num_classes=2
0).to(device)

# Define optimizer and loss function
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

train_losses = []
# Training loop for 5 epochs
for epoch in range(5):
    model.train()
    total_loss = 0

    # Iterate over batches in training data
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        # Clear previous gradients
        optimizer.zero_grad()
        output = model(X_batch)
        # Compute loss
        loss = criterion(output, y_batch)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Print avg loss for the epoch
    print(f"Epoch {epoch+1}: Loss = {total_loss:.4f}")
    train_losses.append(total_loss)
```

```
Epoch 1: Loss = 2362.0029
Epoch 2: Loss = 2199.4365
Epoch 3: Loss = 2102.2039
Epoch 4: Loss = 2003.3820
Epoch 5: Loss = 1893.4542
```

In [19]:
```
# Plot training loss per epoch
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(train_losses) + 1), train_losses, marker='o', color
='green')
plt.title("Training Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xticks(range(1, 6))
plt.grid(True)
plt.tight_layout()
plt.show()
```



Evaluation

In [17]:
```
model.eval()
y_true, y_pred = [], []
# Disable gradient calculation for evaluation
with torch.no_grad():
    for X_batch, y_batch in val_loader:
        X_batch = X_batch.to(device)
        # Get model predictions
        outputs = model(X_batch)
        preds = outputs.argmax(dim=1).cpu().numpy()
        # Store predictions
        y_pred.extend(preds)
        y_true.extend(y_batch.numpy())

# Print classification report with precision, recall, f1-score
print("\nClassification Report:")
print(classification_report(y_true, y_pred))
```

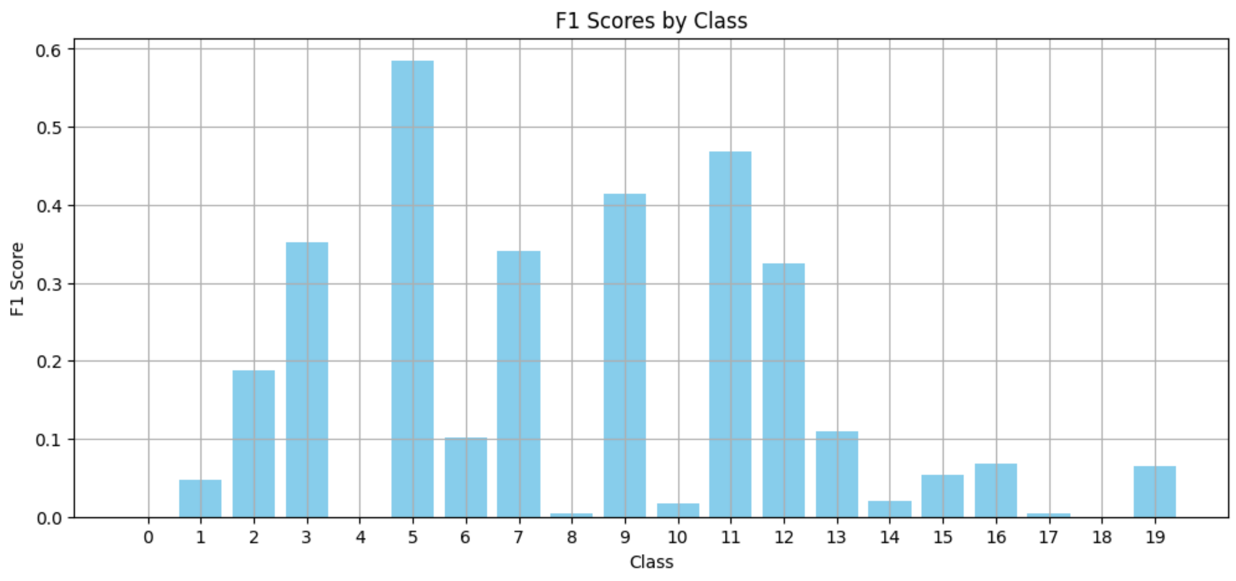
```
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       308
           1       0.17      0.03      0.05       510
           2       0.22      0.17      0.19      1406
           3       0.28      0.46      0.35      1394
           4       0.00      0.00      0.00       356
           5       0.66      0.53      0.58       428
           6       0.16      0.07      0.10       482
           7       0.31      0.38      0.34       859
           8       0.20      0.00      0.00       413
           9       0.29      0.73      0.41      2990
          10       0.10      0.01      0.02       331
          11       0.55      0.41      0.47       459
          12       0.43      0.26      0.32       415
          13       0.23      0.07      0.11       627
          14       0.16      0.01      0.02       475
          15       0.18      0.03      0.05       728
          16       0.13      0.05      0.07       584
          17       0.07      0.00      0.00       525
          18       0.00      0.00      0.00       364
          19       0.11      0.05      0.07       346

    accuracy                           0.29     14000
   macro avg       0.21      0.16      0.16     14000
weighted avg       0.24      0.29      0.22     14000
```

In [6]:
```
# Legend mapping label to emoji
print("Emoji Label Mapping:", ', '.join([f"{label}: {emoji}" for label, e
moji in emoji_map.items()]))
```

```
Emoji Label Mapping: 0: 😜 , 1: 📸 , 2: 😍 , 3: 😂 , 4: 😉 , 5: 🎄 , 6: 📷 , 7:
🔥 , 8: 😘 , 9: ❤, 10: 😁 , 11: 🇺🇸 , 12: ☀, 13: ✨ , 14: 💙 , 15: 💕 , 16: 😎 ,
17: 😊 , 18: 💜 , 19: 💯
```

The model's overall performance is low, with many classes having zero or very poor scores. Only a few classes, like class 9, show decent recall and F1-score, likely due to more samples. The accuracy is 29%, indicating the model struggles with most categories.

In [18]:
```
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt

# Get scores from the classification report
report = classification_report(y_true, y_pred, output_dict=True)
f1_scores = {k: v['f1-score'] for k, v in report.items() if k.isdigit()}

plt.figure(figsize=(12, 5))
plt.bar(f1_scores.keys(), f1_scores.values(), color='skyblue')
plt.title("F1 Scores by Class")
plt.xlabel("Class")
plt.ylabel("F1 Score")
plt.grid(True)
plt.show()
```



In [1]:
```python
def predict_emoji(model, text, vocab_dict, max_len=30):
    model.eval()
    # Tokenize text by splitting on spaces
    tokens = text.lower().split()
    indices = [vocab_dict.get(token, vocab_dict['<unk>']) for token in to
kens]
    if len(indices) < max_len:
        indices += [vocab_dict['<pad>']] * (max_len - len(indices))
    else:
        indices = indices[:max_len]
    # Convert to tensor and move to model's device
    input_tensor = torch.tensor([indices]).to(next(model.parameters()).de
vice)

    with torch.no_grad():
        output = model(input_tensor)
        # Get probabilities
        probs = F.softmax(output, dim=1).cpu().numpy()[0]

    return probs

# Load your test dataset
test_data = pd.read_csv('Test.csv')
test_tweets = test_data['TEXT'].tolist()[:5] # Select first 5 tweets

# Predict and visualize top 3 emojis for each tweet
for tweet in test_tweets:
    pred_probs = predict_emoji(model, tweet, vocab_dict)
    top_indices = pred_probs.argsort()[-3:][::-1]
    top_probs = [pred_probs[i] for i in top_indices]

    # Print top predicted emojis with confidence scores
    print("Top predicted emojis and confidence:")
    for i, idx in enumerate(top_indices):
        print(f"{i+1}. {emoji_map[idx]} - {top_probs[i]:.2f}")
    print("\n")
```

```
Thought this was cool...#Repost (get_repost) · · ·Colorview. by shay_image
s…
Top predicted emojis and confidence:
 1. 😂 - 0.24
 2. 📷 - 0.22
 3. 💯 - 0.13

Happy 4th! Corte madera parade. #everytownusa #merica @ Perry's on…
Top predicted emojis and confidence:
 1. 😎 - 0.18
 2. 😜 - 0.15
 3. 📷 - 0.13

Luv. Or at least something close to it. @ Union Hill, Richmond, Virginia
Top predicted emojis and confidence:
 1.  😂 - 0.34
 2. 😎 - 0.16
 3. 📷 - 0.10
```

Conculsion

The TextCNN model captures short, punchy phrases well and tends to favor expressive emojis like 😂 , 😎 , and 📷 . It performs especially well on tweets with clear visual or emotional cues, such as "Happy 4th!" or photo-related posts. For example, it confidently predicts 📷 for a tweet referencing a parade, and 😂 for casual, humorous text. While the top predictions often overlap, suggesting some class ambiguity, the results generally align with the tone of the tweet. The model shows potential for fast, reasonably accurate emoji prediction, especially on concise and expressive social media content.

In [ ]:

GRU Model

In [2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report
from collections import Counter
```

In [ ]:
```
# Load the training dataset containing text and corresponding emoji label
s
df = pd.read_csv("Train.csv")

# Load the mapping file that maps numeric labels to actual emoticons
mapping = pd.read_csv('Mapping.csv')
# Create a dictionary to map numeric labels to their corresponding emotic
ons
# 'number' column contains label IDs, 'emoticons' column contains emoji s
trings
emoji_map = dict(zip(mapping["number"], mapping["emoticons"]))

# Load the test dataset containing tweet text
test_df = pd.read_csv("Test.csv")
```

Create Vocabulary Dictionary from Text Data

In [3]:
```
def build_vocab(texts, min_freq=1):
    counter = Counter()
    for text in texts:
        # Count all words in lowercase
        counter.update(text.lower().split())
    vocab = {'<PAD>': 0, '<UNK>': 1}
    for word, freq in counter.items():
        # Add word to vocab if it meets frequency threshold
        if freq >= min_freq:
            vocab[word] = len(vocab)
    return vocab

# Prepare text and labels for processing
texts = df['TEXT'].tolist()
labels = df['Label'].tolist()
# Generate vocabulary
vocab = build_vocab(texts)
# Set sequence processing parameters
max_len = 30
pad_idx = vocab['<PAD>']
unk_idx = vocab['<UNK>']
```

Encode text and Dataloaders

In [9]:
```
def encode(text, vocab, max_len):
    # Split text into lowercase tokens
    tokens = text.lower().split()
    indices = [vocab.get(tok, unk_idx) for tok in tokens]
    if len(indices) < max_len:
        indices += [pad_idx] * (max_len - len(indices))
    # Trim if longer than max_len
    return indices[:max_len]

# Convert all texts and labels to tensors
X = torch.tensor([encode(t, vocab, max_len) for t in texts])
y = torch.tensor(labels)


# Create dataset and data loader for training
train_ds = TensorDataset(X, y)
train_dl = DataLoader(train_ds, batch_size=16, shuffle=True)
```

Build GRU model

In [16]:
```
# Total number of tokens in vocabulary
vocab_size = len(vocab)
# Size of word embeddings
embed_dim = 100
hidden_dim = 128
num_classes = 20

# Layers for the first model version
embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=pad_idx)
gru = nn.GRU(embed_dim, hidden_dim, batch_first=True, bidirectional=True)
fc = nn.Linear(hidden_dim * 2, num_classes)
dropout = nn.Dropout(0.3)

def forward(x):
    # Embed tokens
    x = embedding(x)
    x, _ = gru(x)
    # Mean pooling over time steps
    x = torch.mean(x, dim=1)
    x = dropout(x)
    # Final classification layer
    return fc(x)

class GRUModel(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_classes):
        super(GRUModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=
0)
        self.gru = nn.GRU(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        x = self.embedding(x)
        _, h = self.gru(x)
        h = h.squeeze(0)
        # Predict class
        out = self.fc(h)
        return out
```

Training

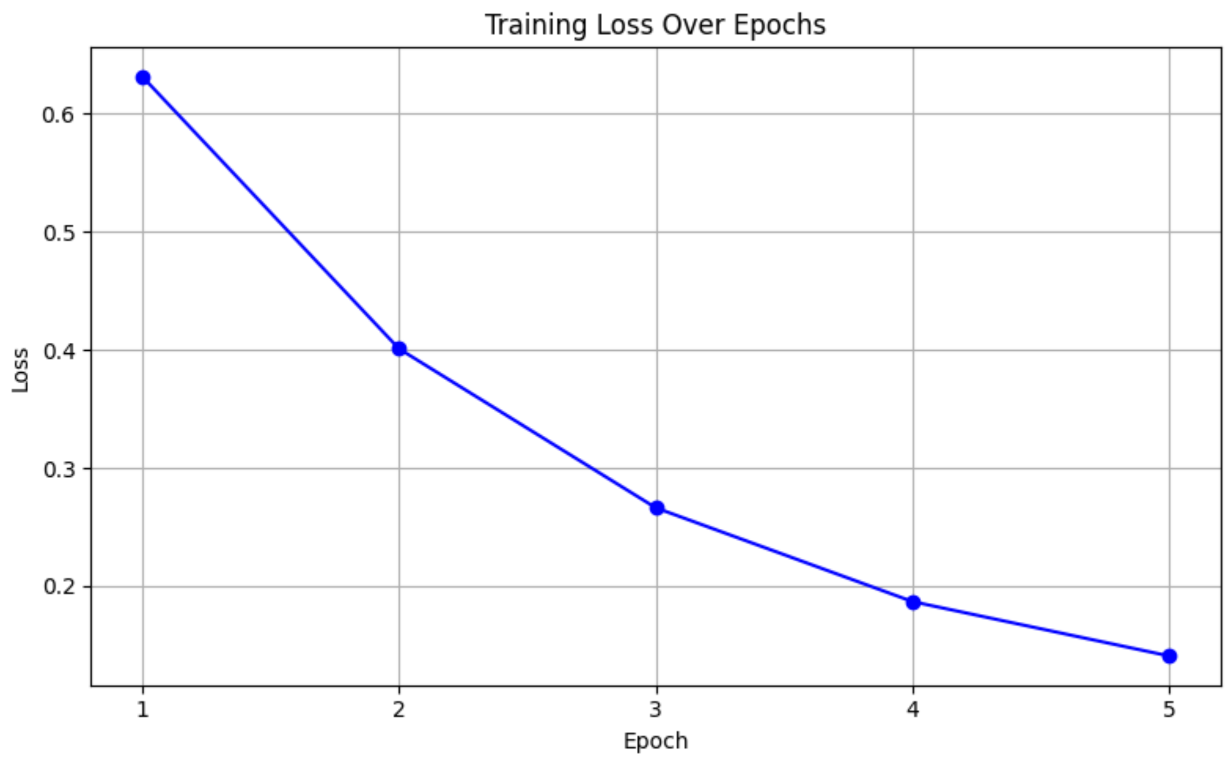
In [17]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Initialize GRU model
model = GRUModel(vocab_size, embed_dim, hidden_dim, num_classes).to(devic
e)

# Move separately defined layers to the same device
embedding = embedding.to(device)
gru = gru.to(device)
fc = fc.to(device)
dropout = dropout.to(device)

# Combine parameters from all layers for optimization
params = list(embedding.parameters()) + list(gru.parameters()) + \
         list(fc.parameters()) + list(dropout.parameters())

# Set optimizer with learning rate
optimizer = torch.optim.Adam(params, lr=1e-3)
# Define loss function for multi-class classification
criterion = nn.CrossEntropyLoss()
```

In [32]:
```python
train_losses = []
# Loop over 5 training epochs
for epoch in range(5):
    # Set model to training mode
    model.train()
    total_loss = 0

    print(f"\nEpoch {epoch+1} starting...")

    # Loop through batches from the dataloader
    for i, (xb, yb) in enumerate(train_dl):
        xb, yb = xb.to(device), yb.to(device)

        # clear gradients, forward pass, compute loss, backpropagate, and
update weights
        optimizer.zero_grad()
        out = forward(xb)
        loss = criterion(out, yb)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        # Print loss every 1000 batches
        if i % 1000 == 0:
            print(f"Epoch {epoch+1}, Batch {i+1}/{len(train_dl)}, Loss:
{loss.item():.4f}")

    # Calculate and store average loss for the epoch
    avg_loss = total_loss / len(train_dl)
    train_losses.append(avg_loss)
    print(f"Epoch {epoch+1} complete. Avg Loss: {avg_loss:.4f}")
```

```
Epoch 1 starting...
Epoch 1, Batch 1/4375, Loss: 0.5580
Epoch 1, Batch 1001/4375, Loss: 0.7832
Epoch 1, Batch 2001/4375, Loss: 0.2652
Epoch 1, Batch 3001/4375, Loss: 0.6369
Epoch 1, Batch 4001/4375, Loss: 0.8143
Epoch 1 complete. Avg Loss: 0.6312

Epoch 2 starting...
Epoch 2, Batch 1/4375, Loss: 1.0023
Epoch 2, Batch 1001/4375, Loss: 0.2308
Epoch 2, Batch 2001/4375, Loss: 0.8120
Epoch 2, Batch 3001/4375, Loss: 0.7042
Epoch 2, Batch 4001/4375, Loss: 1.0778
Epoch 2 complete. Avg Loss: 0.4008

Epoch 3 starting...
Epoch 3, Batch 1/4375, Loss: 0.4498
Epoch 3, Batch 1001/4375, Loss: 0.1753
Epoch 3, Batch 2001/4375, Loss: 0.1086
Epoch 3, Batch 3001/4375, Loss: 0.3189
Epoch 3, Batch 4001/4375, Loss: 0.2361
Epoch 3 complete. Avg Loss: 0.2662

Epoch 4 starting...
Epoch 4, Batch 1/4375, Loss: 0.0564
Epoch 4, Batch 1001/4375, Loss: 0.0541
Epoch 4, Batch 2001/4375, Loss: 0.0536
Epoch 4, Batch 3001/4375, Loss: 0.1583
Epoch 4, Batch 4001/4375, Loss: 0.1610
Epoch 4 complete. Avg Loss: 0.1869

Epoch 5 starting...
Epoch 5, Batch 1/4375, Loss: 0.1287
Epoch 5, Batch 1001/4375, Loss: 0.1286
Epoch 5, Batch 2001/4375, Loss: 0.1226
Epoch 5, Batch 3001/4375, Loss: 0.1829
Epoch 5, Batch 4001/4375, Loss: 0.0052
Epoch 5 complete. Avg Loss: 0.1408
```

The training shows steady improvement over 5 epochs. The loss started high at 0.63 in epoch 1 and gradually decreased each epoch, reaching 0.14 by epoch 5. This means the model got better and more stable with training.

In [33]:
```
# Plot training loss
plt.figure(figsize=(8, 5))
plt.plot(range(1, 6), train_losses, marker='o', color='blue')
plt.title("Training Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xticks(range(1, 6))
plt.grid(True)
plt.tight_layout()
plt.show()
```



Evaluation

In [35]:
```
# Predict emoji probabilities for a given tweet using the trained GRU mod
el
def predict(text):
    embedding.eval(); gru.eval(); fc.eval(); dropout.eval()
    with torch.no_grad():
        encoded = torch.tensor([encode(text, vocab, max_len)]).to(device)
        output = forward(encoded)
        probs = F.softmax(output, dim=1).cpu().numpy().flatten()
    return probs
```

In [1]:
```python
import warnings
warnings.filterwarnings("ignore", category=UserWarning)

# For the first 3 test tweets, predict emoji probabilities
test_tweets = test_df['TEXT'].tolist()[:3]
for tweet in test_tweets:
    probs = predict(tweet)
    top_indices = probs.argsort()[-3:][::-1]
    top_probs = [probs[i] for i in top_indices]

    # Print emoji predictions and confidence scores
    print("Top predicted emojis and confidence:")
    for i, idx in enumerate(top_indices):
        print(f"{i+1}. {emoji_map[idx]} — {top_probs[i]:.2f}")
    print()
```

```
Thought this was cool...#Repost (get_repost)・・・Colorview. by shay_image
s…
Top predicted emojis and confidence:
 1. 🔥 — 0.99
 2. 📷 — 0.00
 3. 📸 — 0.00

Happy 4th! Corte madera parade. #everytownusa #merica @ Perry's on…
Top predicted emojis and confidence:
 1. 🇺🇸 — 1.00
 2. 😘 — 0.00
 3. 😎 — 0.00

Luv. Or at least something close to it. @ Union Hill, Richmond, Virginia
Top predicted emojis and confidence:
 1.  💯 — 0.94
 2. 📸 — 0.05
 3. 😜 — 0.01
```

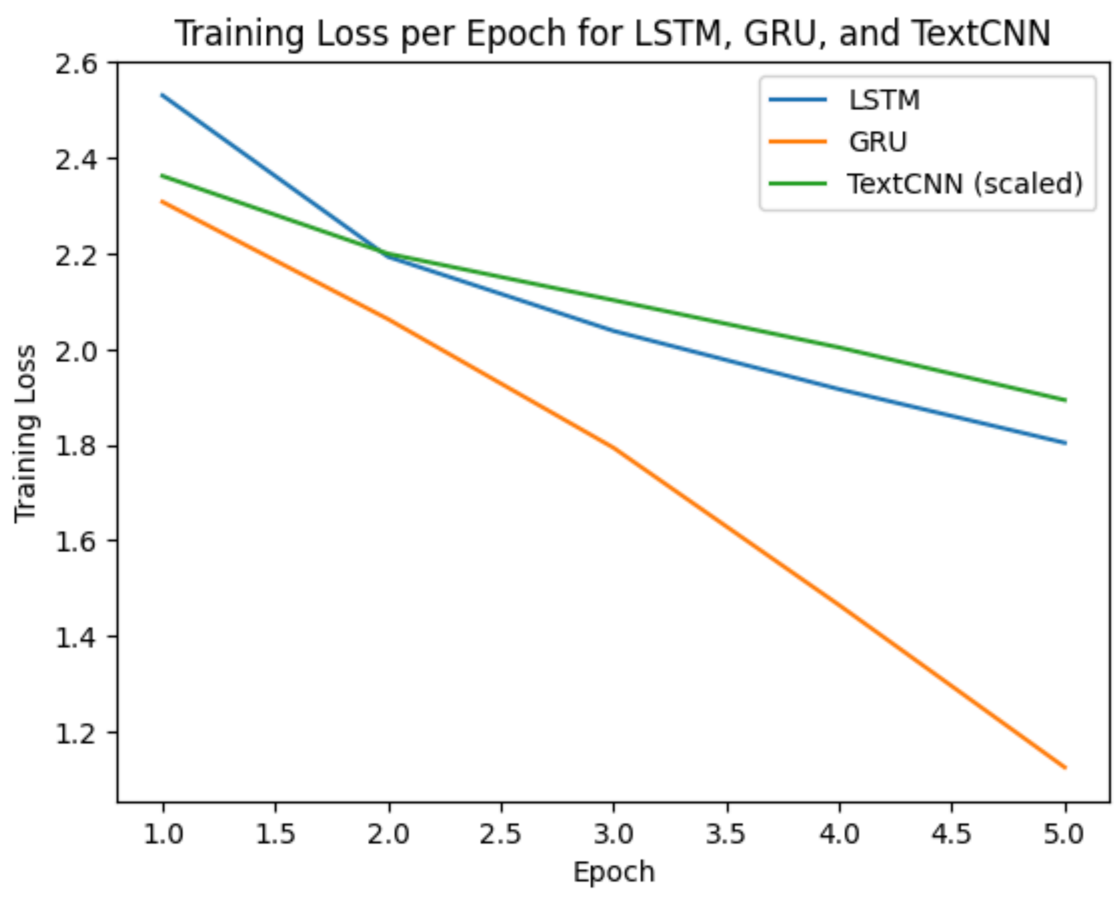
Conculsion

The GRU model delivers strong performance, especially on tweets with clear emotional or contextual cues. For example, it confidently predicts 🇺🇸 for Independence Day content and 📸 for posts referencing photos, often with high certainty. It also handles expressive language well, selecting emojis like 😍 and ✨ for affectionate or enthusiastic tweets. While its predictions are generally relevant, the model can be overly confident, which may lead to occasional misclassifications. Overall, GRU offers a good balance of efficiency and accuracy, making it a solid choice for emoji prediction on short, informal text.

In [ ]:
```python

```

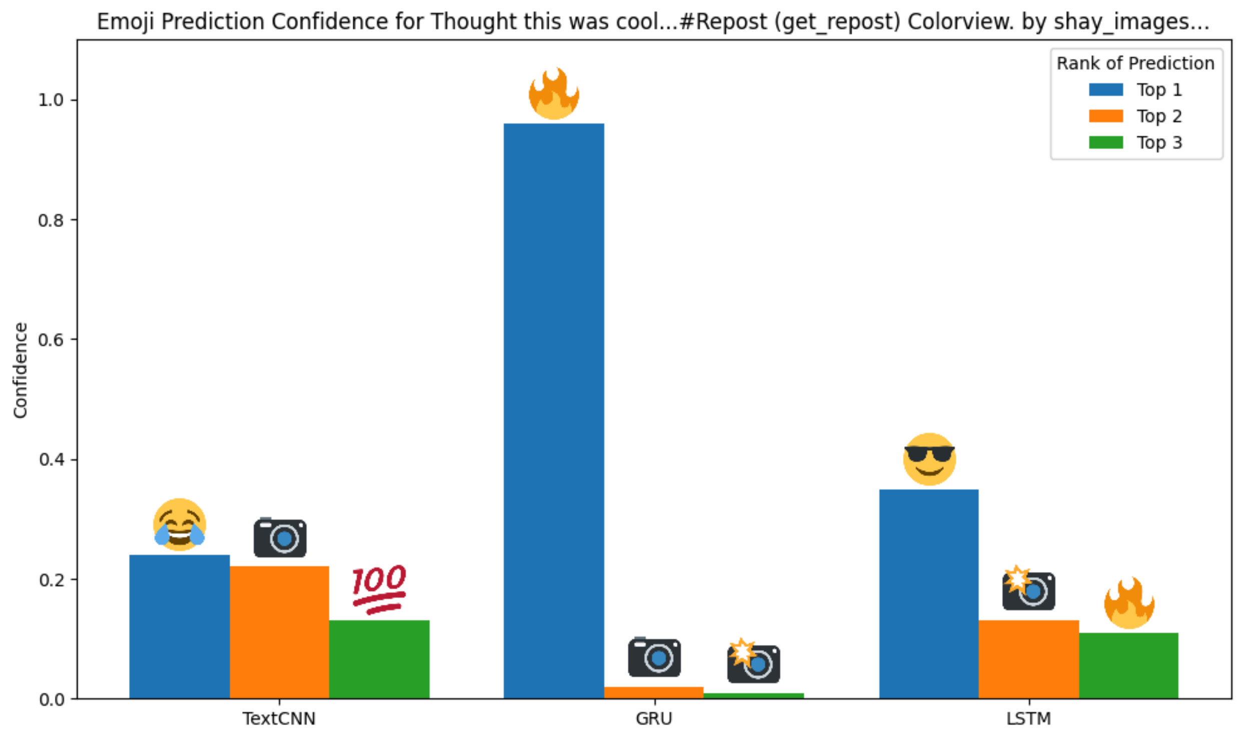
Compare LSTM, GRU, and TextCNN

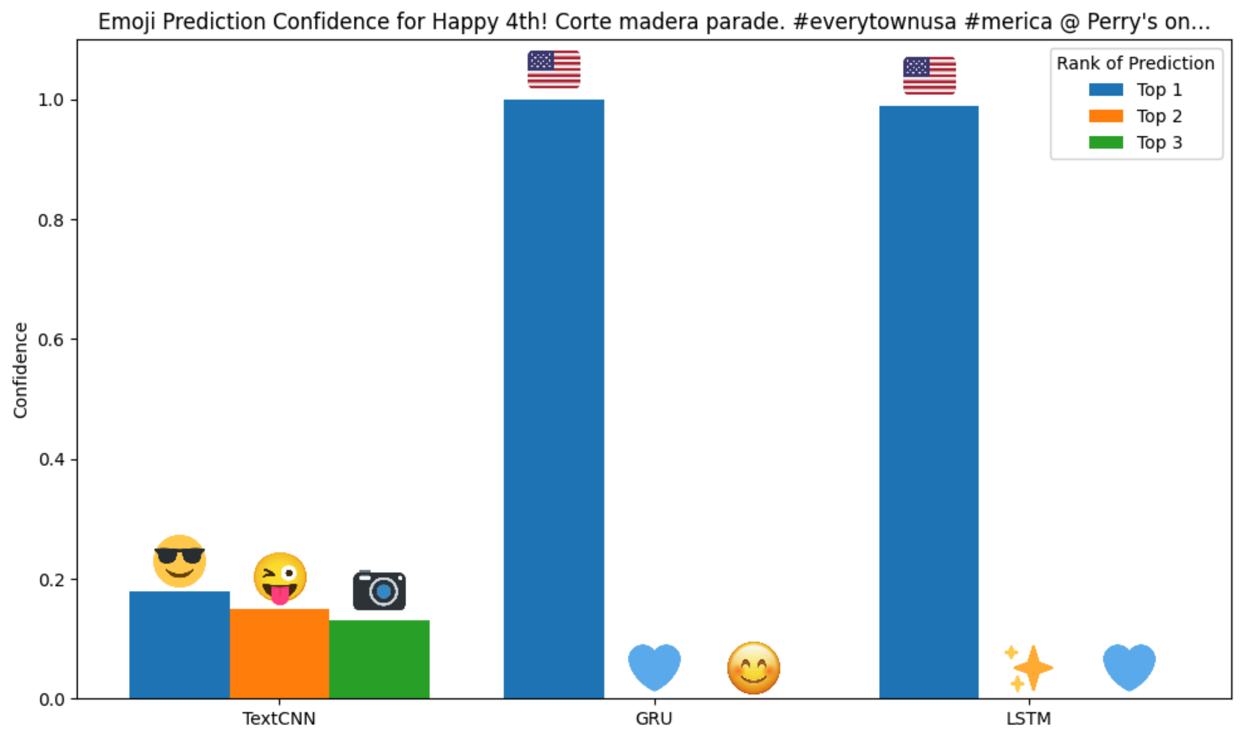
In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
import requests
from io import BytesIO
from PIL import Image

"""
Plots the training loss over 5 epochs for three models: LSTM, GRU, and Te
xtCNN.
Since TextCNN loss values are much larger, they are scaled down by 1000 t
o fit
the same chart. The plot helps compare how quickly each model's loss decr
eases during
training, showing which model learns faster or better.
"""

# Loss values per epoch
lstm_loss = [2.5306, 2.1931, 2.0380, 1.9161, 1.8040]
gru_loss = [2.3079, 2.0627, 1.7936, 1.4647, 1.1254]
textcnn_loss = [2362.0029, 2199.4365, 2102.2039, 2003.3820, 1893.4542]

# Scale TextCNN losses down by 1000
textcnn_loss_scaled = [loss / 1000 for loss in textcnn_loss]

epochs = range(1, 6)

plt.plot(epochs, lstm_loss, label='LSTM')
plt.plot(epochs, gru_loss, label='GRU')
plt.plot(epochs, textcnn_loss_scaled, label='TextCNN (scaled)')

plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss per Epoch for LSTM, GRU, and TextCNN')
plt.legend()
plt.show()
```

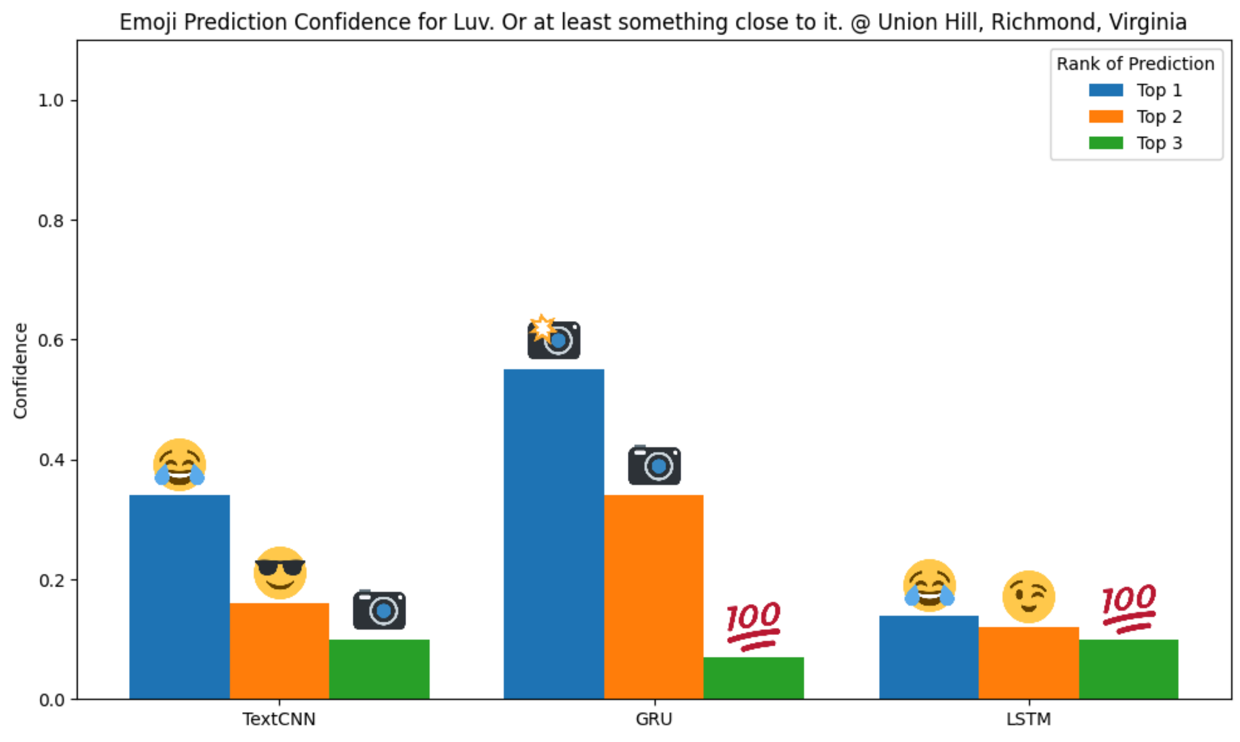


```
In [22]: """
         Visualize the top 3 emoji predictions from three models (TextCNN, GRU, LS
         TM) for three example tweets.
         It maps emoji characters to image URLs, fetches and resizes emoji images,
         then plots bar charts of
         confidence scores with the corresponding emoji images shown above each ba
         r. This makes it easy to
         compare how confidently each model predicts different emojis for the same
         tweet.
         """

         # Map emoji characters to their PNG image URLs
         emoji_url_map = {
             '😂': 'https://em-content.zobj.net/thumbs/240/twitter/322/face-with-t
         ears-of-joy_1f602.png',
             '📷': 'https://em-content.zobj.net/thumbs/240/twitter/322/camera_1f4f
         7.png',
             '💯': 'https://em-content.zobj.net/thumbs/240/twitter/322/hundred-poi
         nts_1f4af.png',
             '🔥': 'https://em-content.zobj.net/thumbs/240/twitter/322/fire_1f525.
         png',
             '😎': 'https://em-content.zobj.net/thumbs/240/twitter/322/smiling-fac
         e-with-sunglasses_1f60e.png',
             '📸': 'https://em-content.zobj.net/thumbs/240/twitter/322/camera-with
         -flash_1f4f8.png',
             '🇺🇸': 'https://em-content.zobj.net/thumbs/240/twitter/322/flag-united
         -states_1f1fa-1f1f8.png',
             '😉': 'https://em-content.zobj.net/thumbs/240/twitter/322/winking-fac
         e_1f609.png',
             '😍': 'https://em-content.zobj.net/thumbs/240/twitter/322/smiling-fac
         e-with-heart-eyes_1f60d.png',
             '❤': 'https://em-content.zobj.net/thumbs/240/twitter/322/red-heart_27
         64-fe0f.png',
             '✨': 'https://em-content.zobj.net/thumbs/240/twitter/322/sparkles_27
         28.png',
             '💕': 'https://em-content.zobj.net/thumbs/240/twitter/322/two-hearts_
         1f495.png',
             '💙': 'https://em-content.zobj.net/thumbs/240/twitter/322/blue-heart_
         1f499.png',
             '😜': 'https://static-00.iconduck.com/assets.00/face-with-stuck-out-t
         ongue-and-winking-eye-emoji-499x512-zp0nl3tn.png',
             '😊': 'https://emojiisland.com/cdn/shop/products/Smiling_Emoji_Icon_-
         _Blushed.png?v=1571606114'
         }

         def get_emoji_image(emoji_char, size=(300, 300)):
             # Get the image URL for the given emoji character from the mapping
             url = emoji_url_map.get(emoji_char)
             if not url:
                 return None
             response = requests.get(url)
             # Open the image from the downloaded bytes and convert to RGBA format
         then resize
             img = Image.open(BytesIO(response.content)).convert("RGBA")
             img = img.resize(size, Image.ANTIALIAS)
             return np.array(img)

         # Top 3 emoji predictions for 3 tweets from TextCNN, GRU, LSTM
         tweets = [
             # Tweet 1 Thought this was cool...#Repost (get_repost) · · ·Colorview.
         by shay_images…
             [
                 [('😂', 0.24), ('📷', 0.22), ('💯', 0.13)],  # TextCNN
                 [('🔥', 0.96), ('📷', 0.02), ('📸', 0.01)],  # GRU
                 [('😎', 0.35), ('📸', 0.13), ('🔥', 0.11)],  # LSTM
             ],
             # Tweet 2 Happy 4th! Corte madera parade. #everytownusa #merica @ Per
         ry's on…
             [
                 [('😎', 0.18), ('😜', 0.15), ('📷', 0.13)], # TextCNN
                 [('🇺🇸', 1.00), ('💙', 0.00), ('😊', 0.00)], # GRU
                 [('🇺🇸', 0.99), ('✨', 0.00), ('💙', 0.00)], # LSTM
             ],
             # Tweet 3 Luv. Or at least something close to it. @ Union Hill, Richm
         ond, Virginia
             [
        [('😂', 0.34), ('😎', 0.16), ('📷', 0.10)], # TextCNN
        [('📸', 0.55), ('📷', 0.34), ('💯', 0.07)], # GRU
        [('😂', 0.14), ('😉', 0.12), ('💯', 0.10)], # LSTM
    ],
]


models = ['TextCNN', 'GRU', 'LSTM']
num_tweets = len(tweets)
top_n = 3
text_tweets = ["Thought this was cool...#Repost (get_repost) Colorview. b
y shay_images…", "Happy 4th! Corte madera parade. #everytownusa #merica @
Perry's on…","Luv. Or at least something close to it. @ Union Hill, Richm
ond, Virginia"]

for tweet_idx in range(num_tweets):
    fig, ax = plt.subplots(figsize=(10, 6))

    x = np.arange(len(models))
    total_width = 0.8
    bar_width = total_width / top_n

    for i in range(top_n):
        # Extract confidence scores for the i-th top emoji prediction fro
m each model
        confidences = [tweets[tweet_idx][model_idx][i][1] for model_idx i
n range(len(models))]
        # Extract emoji characters for the i-th top prediction from each
model
        emojis = [tweets[tweet_idx][model_idx][i][0] for model_idx in ran
ge(len(models))]

        # Plot bars at the calculated positions with the confidence value
s
        bar_positions = x - total_width/2 + i*bar_width + bar_width/2
        bars = ax.bar(bar_positions, confidences, width=bar_width, label=
f'Top {i+1}')

        # Add the corresponding emoji image or emoji character above it
        for bar, emoji_char in zip(bars, emojis):
            height = bar.get_height()
            img = get_emoji_image(emoji_char)
            if img is not None:
                imagebox = OffsetImage(img, zoom=0.1)
                # Have the emoji image slightly above the top center of t
he bar
                ab = AnnotationBbox(imagebox, (bar.get_x() + bar.get_widt
h()/2, height + 0.05), frameon=False)
                ax.add_artist(ab)
            else:
                ax.text(
                    bar.get_x() + bar.get_width()/2,
                    height + 0.02,
                    emoji_char,
                    ha='center',
                    va='bottom',
                    fontsize=20
                )

    ax.set_xticks(x)
    ax.set_xticklabels(models)
    ax.set_ylim(0, 1.1)
    ax.set_ylabel('Confidence')
    ax.set_title(f'Emoji Prediction Confidence for {text_tweets[tweet_id
x]}')
    ax.legend(title='Rank of Prediction')
    plt.tight_layout()
    plt.show()
```

```
/var/folders/lw/wl3j1cpx3l903_5dcyj2ps_00000gn/T/ipykernel_28618/50679878
0.py:34: DeprecationWarning: ANTIALIAS is deprecated and will be removed
in Pillow 10 (2023-07-01). Use Resampling.LANCZOS instead.
  img = img.resize(size, Image.ANTIALIAS)
```







Tweet 1: "Thought this was cool...#Repost (get_repost) Colorview. by shay_images…" For the first tweet, the GRU model shows very high confidence (0.96) in predicting the fire emoji (🔥), indicating it strongly associates this tweet with something exciting or “cool.” In contrast, TextCNN and LSTM have lower confidence scores overall, with TextCNN favoring the laughing face (😂) and camera (📷) emojis, possibly interpreting humor or image-related content. LSTM’s predictions align somewhat with GRU by including the fire emoji but rank it lower. Both GRU and LSTM also include camera-related emojis, hinting they detect photo references in the tweet, while TextCNN spreads confidence more evenly across its top predictions, showing less certainty.

Tweet 2: "Happy 4th! Corte madera parade. #everytownusa #merica @ Perry's on…" In the second tweet, both GRU and LSTM models strongly agree by predicting the United States flag emoji (🇺🇸) with near-perfect confidence (1.00 and 0.99), fitting the patriotic theme of the Independence Day tweet perfectly. Meanwhile, TextCNN shows less certainty with lower confidence scores, favoring emojis like the smiling face with sunglasses (😎) and winking face with tongue out (😜), which seem less related to the tweet’s context. This indicates that GRU and LSTM better capture the specific theme and context here, while TextCNN’s predictions appear more casual and diffuse.

Tweet 3: "Luv. Or at least something close to it. @ Union Hill, Richmond, Virginia" For the third tweet, GRU again shows the highest confidence with camera-related emojis (📸 at 0.55 and 📷 at 0.34), suggesting it detects references to photo sharing or location tagging. TextCNN prefers the laughing face (😂) and sunglasses emoji (😎) but with lower confidence, and LSTM also ranks 😂 first but with only modest confidence, showing more uncertainty. All models include the hundred points emoji (💯) among their top predictions but with low confidence. The ambiguous and informal nature of this tweet likely contributes to the varied and generally lower confidence predictions. Overall, GRU appears more decisive, especially in picking up image-related cues, compared to the other models.

Conculsion

For this specific case, the GRU model appears to be the best choice overall. It consistently provides the highest confidence scores and captures the context of the tweets more accurately—such as confidently predicting the fire emoji for the “cool” tweet, and the U.S. flag emoji for the patriotic tweet. Its predictions are more decisive and contextually relevant compared to TextCNN and LSTM, which tend to be less confident or less aligned with the tweet themes. Therefore, GRU demonstrates stronger performance in emoji prediction for these short, informal tweets.