Ramakrishna Mission Vivekananda Educational & Research Institute
Belur Math, Howrah, West Bengal
School of Mathematical Sciences, Department of Computer Science

Assignment - 3: Sentiment analysis using CNN
M.Sc. Computer Science and Big Data Analytics Date: 29 March 2026
Course: **CS411: Applications of Computer Vision and Deep Learning**
Deadline: 13th-April-2026, 11:59 P.M.
Instructor: Jimut Bahan Pal Max marks: 100

Instructions: Read Carefully and Attempt All Questions

Implementation Guidelines:

- All code must run end-to-end on **Google Colaboratory** with **GPU enabled** (required for training efficiency)
- Your notebook must be self-contained: include all `!pip install` commands at the top
- Use **PyTorch** (`torch`) throughout — do **not** use TensorFlow or Keras
- Use `matplotlib` for all visualisations; every plot must have a title and labelled axes
- Add brief inline comments explaining what each code block does
- You may discuss ideas with peers, but all code must be written individually

Submission Format:

- Submit your **Jupyter notebook** (`.ipynb`) with all cell outputs visible
- Compress the file as **Name_ROLL.zip**

Submission Process:

- Email your submission to **jimutbahanpal@yahoo.com**
- Add **jpal.cs@gm.rkmvu.ac.in** as CC (Carbon Copy)
- Optionally CC your personal email to confirm delivery

Academic Integrity:
While you may use LLMs or agentic AI tools, you must:

- Fully understand any AI-generated code
- Be prepared to explain and justify every line during the viva
- Note: Inability to justify your work during the viva will result in negative marking

Deadline:
Submissions received after the deadline will incur a penalty of -5 **marks per day**. Plan accordingly and start early!

Dataset:
Download the Stanford Twitter Sentiment dataset from `http://cs.stanford.edu/people/alecmgo/trainingandtestdata.zip`. The dataset contains a CSV file with columns: `sentiment`, `id`, `date`, `query`, `user`, `text`.

For faster training iteration, use a subset of the data:

- Recommended: 100,000 or 200,000 samples (approximately 5–10 minutes per epoch on GPU)
- Full dataset: 1,600,000 samples (3+ hours per epoch)

Extract the zip file and load the CSV named `training.1600000.processed.noemoticon.csv`.

---

1. **Data Loading and Exploratory Data Analysis** (10)

Download, extract, and load the Stanford Twitter sentiment dataset. Perform comprehensive exploratory analysis on the loaded data.

1. **Download and extract dataset.** **[2]**
   Download the zip file from Stanford server using `urllib.request.urlretrieve()`. Extract using `zipfile.ZipFile()`. Load the CSV file training.1600000.processed.noemoticon.csv using pandas. Limit to 100,000 or 200,000 samples using the `nrows` parameter for faster processing.
2. **Display dataset structure.** **[3]**
   Print basic statistics:
   - Total number of loaded samples
   - Column names: sentiment, id, date, query, user, text
   - Class distribution (count and percentage for each sentiment label)
   - Text length statistics: minimum, maximum, mean, median length

   Note: sentiment values 0 and 4 represent negative and positive labels respectively. Replace 4 with 1 for binary classification.
3. **Create visualisations.** **[3]**
   Generate two plots:
   - Bar chart showing sentiment class distribution (absolute counts and percentages)
   - Histogram of text lengths with appropriate bins and labels

   Both plots must have titles and labelled axes.
4. **Display sample data.** **[2]**
   Show 3 example tweets from the negative class (label 0) and 3 from the positive class (label 1). In a markdown cell, write 2–3 observations about the data characteristics.

Required packages (first cell):

```
!pip install pandas numpy matplotlib torch sentencepiece tqdm beautifulsoup4

```

2. **Text Preprocessing and Cleaning** (10)

Implement a complete text cleaning pipeline to prepare tweets for tokenisation. Use the exact cleaning procedure as specified in the reference code.

1. **Implement the cleaning function.** **[5]**
   Write a function `clean_tweet(tweet: str) -> str` that performs:
   - HTML tag removal using BeautifulSoup with lxml parser
   - Remove @mentions (user references)

- Remove URLs (http and https links)
- Keep only letters and punctuation (period, exclamation, question mark, apostrophe)
- Collapse multiple consecutive spaces into single spaces
- Strip leading and trailing whitespace

Implement the function with docstring and inline comments.

2. **Apply cleaning to all tweets.** **[3]**
   Apply your `clean_tweet()` function to all tweets in the dataset. Store cleaned tweets in a new DataFrame column or list. Use `tqdm` for progress tracking.
3. **Before-and-after comparison.** **[2]**
   Display a table with 5 original–cleaned tweet pairs. Include a brief comment explaining what was removed from each example.

3. **Subword Tokenisation using SentencePiece BPE** (10)

Train a SentencePiece tokeniser on the cleaned tweets and encode all sequences. Use target vocabulary size of 2^{16} (65,536 tokens) as in the reference implementation.

1. **Train the SentencePiece model.** **[4]**
   Train a BPE (Byte Pair Encoding) tokeniser with:
   - Vocabulary size: 65,536 tokens (2 to the power of 16)
   - Model type: BPE
   - Character coverage: 1.0
   - Pad ID: 0, Unknown ID: 1
   - No BOS or EOS tokens

   Save the model file and print the vocabulary size.
2. **Encode all tweets.** **[3]**
   Use the trained SentencePiece encoder to convert all cleaned tweets into token ID sequences. Store as a list of lists or NumPy array.
3. **Analyse tokenisation.** **[3]**
   For 5 sample tweets, display:
   - Original cleaned text
   - Token IDs (as Python list)
   - Decoded token strings
   - Number of tokens per tweet

   Report aggregate statistics: average tokens per tweet, maximum sequence length, minimum sequence length, standard deviation.

4. **Sequence Padding and Tensor Conversion** (10)

Pad sequences to uniform length and prepare NumPy arrays for neural network training.

1. **Compute sequence statistics.** **[3]**
   From the encoded token sequences, calculate:
   - Maximum length (MAX_LEN)
   - Minimum length
   - Mean length

- Median length
- Standard deviation

Create a histogram of original sequence lengths and mark MAX_LEN as a vertical line.

2. **Implement padding function.** **[3]**
   Write a function that pads shorter sequences with zeros at the end to match MAX_LEN.
3. **Apply padding and convert to arrays.** **[2]**
   Pad all encoded sequences to MAX_LEN using padding ID = 0. Convert to NumPy arrays with `dtype=np.int64`. Verify all sequences have identical length.
4. **Verification.** **[2]**
   Display 3 examples showing original length to padded length transformation. Print final array shapes and estimated memory footprint (in megabytes).

5. **Balanced Train/Test Data Splitting** (10)

Create balanced train and test sets using stratified random sampling with reproducible seed.

1. **Implement balanced splitting.** **[4]**
   Split data with:
   - Train: 90 percent of data
   - Test: 10 percent of data
   - Both sets balanced (equal positive and negative samples or close to equal)
   - Random seed = 42 for reproducibility

   Use `np.random.default_rng(42)` for random seeding.
2. **Verify class distributions.** **[3]**
   Create a table showing:
   - Total samples per set
   - Class 0 count and percentage
   - Class 1 count and percentage

   Plot side-by-side bar charts comparing original, train, and test distributions.
3. **Confirm no data leakage.** **[3]**
   Verify that train and test index sets do not overlap. Print train and test set sizes with class balance information.

6. **Creating PyTorch DataLoaders** (10)

Convert NumPy arrays to PyTorch tensors and create efficient batch DataLoaders for GPU training.

1. **Create tensors and verify.** **[3]**
   Convert to PyTorch tensors:
   - Inputs: `torch.long` data type (token IDs)
   - Labels: `torch.float32` data type

   Verify tensor properties: shape, data type, and device placement.
2. **Create TensorDataset and DataLoaders.** **[4]**
   Build two DataLoaders with batch size 32 (or 128 for faster training):

- Training loader: shuffle enabled
- Test loader: shuffle disabled
- Pin memory for GPU acceleration

3. **Test DataLoaders.** **[3]**
   Iterate through one batch and display:
   - Batch input shape and labels shape
   - Sample token IDs and labels from first 3 samples
   - Total number of batches in training and test loaders

7. **DCNN Model Architecture Implementation** (10)

Implement the complete DCNN model in PyTorch. Show model architecture using PyTorch `print(model)` and visualise using torchviz or similar.

1. **Build DCNN class with exact architecture.** **[5]**
   Implement `DCNN(nn.Module)` with:
   - **Embedding**: vocabulary size by 200 dimensions, padding index 0
   - **Three parallel Conv1D layers**:
     - Bigram: kernel size 2, output channels 100, no padding
     - Trigram: kernel size 3, output channels 100, no padding
     - Fourgram: kernel size 4, output channels 100, no padding
   - **Activation**: ReLU after each convolution
   - **Global Max Pooling**: `torch.max(x, dim=2).values` per branch
   - **Concatenation**: merge three pooled outputs (300 features total)
   - **Dense layer**: 300 to 256 with ReLU activation
   - **Dropout**: rate 0.2 (active during training only)
   - **Output layer**: 256 to 1 with Sigmoid activation (binary classification)

   Include docstrings and comments explaining each component.

2. **Document forward pass with tensor shapes.** **[2]**
   Show tensor shapes at each step from input to final output. Create a summary table listing layer names, input shapes, output shapes.

3. **Display model architecture and parameter count.** **[3]**
   - Print the model using `print(model)` to show architecture
   - Count and report total trainable parameters
   - Create a layer-by-layer breakdown table showing:
     - Layer name
     - Input shape
     - Output shape
     - Parameter count
   - Optionally: create a model graph/diagram using torchviz (`from torchviz import make_graph`)

Starter code:

```python
import torch.nn as nn

class DCNN(nn.Module):
        def __init__(self, vocab_size, emb_dim=200,
                nb_filters=100, ffn_units=256,
                nb_classes=2, dropout_rate=0.2):
                super().__init__()

                self.embedding = nn.Embedding(vocab_size, emb_dim,
                padding_idx=0)
                self.bigram = nn.Conv1d(emb_dim, nb_filters,
                kernel_size=2)
                self.trigram = nn.Conv1d(emb_dim, nb_filters,
                kernel_size=3)
                self.fourgram = nn.Conv1d(emb_dim, nb_filters,
                kernel_size=4)

                self.relu = nn.ReLU()
                self.dropout = nn.Dropout(p=dropout_rate)
                self.dense_1 = nn.Linear(nb_filters * 3, ffn_units)
                self.last_dense = nn.Linear(ffn_units, 1)
                self.output_act = nn.Sigmoid()

        def forward(self, x):
                # some code here

                return self.output_act(out)

```

8. **Training Loop Implementation** (10)

Implement a complete training pipeline with loss computation, optimization, and evaluation metrics. Train for 5 epochs.

1. **Setup training infrastructure.** **[2]**
   Define:
   - Loss function: `nn.BCELoss()` (binary cross-entropy)
   - Optimizer: `optim.Adam()`
   - Device: GPU (`cuda`) if available, else CPU
   - Hyperparameters: 5 epochs, batch size 32
2. **Implement run_epoch function.** **[4]**
   Write `run_epoch(loader, training, epoch_num)` that:
   - Sets model to `train()` or `eval()` mode appropriately
   - Iterates over batches with tqdm progress bar
   - Forward pass: `out = model(X_batch)`
   - Computes loss on squeezed output: `loss = criterion(out.squeeze(1), y_batch)`
   - If training: `zero_grad()`, `backward()`, `optimizer.step()`
   - Computes prediction: `pred = (out.squeeze(1) > 0.5).float()`
   - Returns average loss and accuracy
3. **Run training loop.** **[2]**
   Train for 5 epochs:

- Each epoch: training phase followed by test phase
- Print per-epoch: train loss, train accuracy, test loss, test accuracy, epoch time
- Track best test accuracy and corresponding epoch
- Save best model checkpoint to disk

4. **Create visualisations.** **[2]**

   Plot three graphs with proper titles and labels:

   - Training loss over epochs
   - Training accuracy over epochs
   - Train versus test loss comparison (overlaid on same plot)

9. **Model Evaluation and Error Analysis** (10)

Evaluate the trained model comprehensively and analyse failure cases to understand model limitations.

1. **Compute evaluation metrics.** **[4]**

   On the full test set, report:

   - Final test loss and accuracy
   - Confusion matrix values: TP, TN, FP, FN
   - Precision, Recall, F1-Score

   Create a 2-by-2 confusion matrix heatmap with proper labels and colorbar.

2. **Visualise model performance.** **[2]**

   Plot two diagnostic graphs:

   - ROC curve (TPR on y-axis, FPR on x-axis)
   - Confusion matrix heatmap with cell labels

3. **Error analysis.** **[4]**

   Find and display 10 misclassified samples from the test set:

   - Show: original text, predicted probability, true label, predicted label
   - Format as a markdown table

   In a markdown cell, analyse:

   - What linguistic features cause misclassification (e.g., sarcasm, negation, slang)?
   - Which sentiment class is harder to predict and why?
   - Write 4–5 sentences explaining the model's failure modes

10. **Model Saving, Loading, and Inference** (10)

Save the trained model, load it back, implement inference on custom text, and verify reproducibility with fixed random seeds.

1. **Save model checkpoint.** **[2]**

   Use `torch.save()` to save a checkpoint dictionary containing:

   - `model_state`: model state dict
   - `optimizer_state`: optimizer state dict
   - Hyperparameters: VOCAB_SIZE, MAX_LEN, EMB_DIM, NB_FILTERS, FFN_UNITS, DROPOUT_RATE
   - Metadata: best accuracy, best epoch, timestamp

- Dataset size used

2. **Load model from checkpoint.** **[2]**

   Reconstruct the DCNN model with saved hyperparameters. Load state dict into model. Verify weights match by comparing a sample of layer weights before and after loading.

3. **Implement inference function.** **[3]**

   Write `predict_sentiment(text: str)` function that:

   - Cleans input text using `clean_tweet()`
   - Tokenises with SentencePiece encoder
   - Pads to MAX_LEN (minimum 4 tokens for kernel size 4)
   - Converts to tensor and moves to correct device
   - Runs inference in `model.eval()` mode with `torch.no_grad()`
   - Returns dictionary with: text, probability, label (POSITIVE/NEGATIVE)

   Include docstring explaining parameters and return values.

4. **Test inference on custom samples.** **[2]**

   Run inference on 10 custom sentences (5 clearly positive, 5 clearly negative). Display results as a markdown table:

   - Original text
   - Cleaned text
   - Predicted probability (0 to 1)
   - Predicted label

5. **Verify reproducibility.** **[1]**

   Set random seeds:

   - `torch.manual_seed(42)`
   - `np.random.seed(42)`

   Load the checkpoint and run inference twice on the same text. Verify that predicted probabilities are identical (no randomness).

Total: 100 marks ■