CSCI-6515 Natural Language Processing

Project 5: QA Systems with BiDAF, BERT, and RAG

Instructor: Samir Rustamov

Students: Abbas Aliyev, Huru Algayeva

Problem Statement

Part 1: Reading Comprehension

- **BiDAF Implementation**
  Extract answer spans from a passage using bidirectional attention.

- **BERT Integration**
  Enhance BiDAF with contextual embeddings from BERT.

- **Training & Evaluation**
  Train on SQuAD-like datasets and evaluate with EM and F1 scores.

Part 2: Open-Domain Question Answering

- **Document Retrieval**
  Retrieve relevant documents using sparse (TF-IDF) and dense (SBERT) methods.

- **RAG Pipeline**
  Combine retrieved context with the question; extract or generate answers using BiDAF or T5/BART.

- **End-to-End Evaluation**
  Measure system performance using EM, F1, BLEU, or ROUGE depending on output type.

Dataset -SQuaD version 1

A benchmark reading comprehension dataset with over 100,000 **question-answer pairs** on articles. Each question is answered **by extracting a span of text** from the given context. Used to train and evaluate extractive **QA models.**

University_of_Notre_ Dame	The Joan B. Kroc Institute for International Peace Studies at the University of Notre Dame is dedica...	In what year was the Joan B. Kroc Institute for International Peace Studies founded?	1986
University_of_Notre_ Dame	The Joan B. Kroc Institute for International Peace Studies at the University of Notre Dame is dedica...	To whom was John B. Kroc married?	Ray Kroc
University_of_Notre_ Dame	The Joan B. Kroc Institute for International Peace Studies at the University of Notre Dame is dedica...	What company did Ray Kroc own?	McDonald's
University_of_Notre_ Dame	The library system of the university is divided between the main library and each of the colleges an	How many stories tall is the main library at Notre Dame?	14

Part 1 - Tokenization features

Key Fields

- **id, title** – metadata (string)
- **context, question** – raw text input
- **answers** – contains:
  - text: correct answer(s)
  - answer_start: char index in the context

Tokenized Fields

- **tokenized_context, tokenized_question** – list of tokens after tokenization
- **context_token_char_offsets** – maps each token to its original char span
- **token_answer_start, token_answer_end** – exact start/end token indices of the answer span

```
1 squad_train_tokenized.features
```

```
{'id': Value(dtype='string', id=None),
 'title': Value(dtype='string', id=None),
 'context': Value(dtype='string', id=None),
 'question': Value(dtype='string', id=None),
 'answers': Sequence(feature={'text': Value(dtype='string', id=None), 'answer_start': Value(dtype='int32', id=None)}, length=-1, id=None),
 'tokenized_context': Sequence(feature=Value(dtype='string', id=None), length=-1, id=None),
 'tokenized_question': Sequence(feature=Value(dtype='string', id=None), length=-1, id=None),
 'context_token_char_offsets': Sequence(feature=Value(dtype='int64', id=None), length=-1, id=None),
 'token_answer_start': Value(dtype='int64', id=None),
 'token_answer_end': Value(dtype='int64', id=None)}
```

Vocabulary

This part creates a custom vocabulary from a tokenized SQuAD dataset:

1. **Counts word frequencies** in tokenized_context and tokenized_question.
2. **Creates mappings**:
   - word_to_idx → token to ID
   - idx_to_word → ID to token
   - Includes special tokens like <PAD> and <UNK>

```
Building vocabulary...
Found 103961 unique words in the training data.
Vocabulary size (including special tokens): 103963
First 10 words in vocab: {'<PAD>': 0, '<UNK>': 1, 'the': 2, ',': 3, 'of': 4, '.': 5, 'and': 6, 'in': 7, 'to': 8, 'a': 9}
Example: ID for '<UNK>' is 1
Example: ID for 'the' is 2
Saving vocabulary to ./squad_vocab_spacy.json...
Vocabulary saved.
```

Vocabulary

Load GloVe Vectors
Reads pre-trained word embeddings (e.g., glove.840B.300d.txt) into a dictionary.

Build Embedding Matrix
For each word in our vocab:

- If it's in GloVe → insert the GloVe vector.
- If not → keep random vector.
- For <PAD> → insert all-zero vector.
- Optionally customize <UNK>

```
Loading GloVe vectors from ./glove.840B.300d.txt...
Loaded 2195884 word vectors from GloVe.
Creating embedding matrix...
Embedding matrix shape: (103963, 300)
Words found in GloVe (hits): 86556
Words not found in GloVe (misses): 17407 (these will use random initialization or UNK vector)
Vector for UNK token (index 1): [-0.18693003  0.59416074 -0.40828347  0.6995341  -0.15922594  0.29638168
  0.30654564 -0.00461517 -0.2958318   0.5320532 ]...
Vector for PAD token (index 0): [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]...
Embedding matrix converted to PyTorch tensor with shape: torch.Size([103963, 300])
Saving embedding matrix tensor to ./squad_embedding_matrix_spacy.pt...
Embedding matrix tensor saved.
```

BiDAF (Bidirectional Attention Flow) pipeline

This sets up and loads everything needed to **train the BiDAF model**:

- Loads vocab files, GloVe embeddings, and processed SQuAD datasets.
- Prepares **word + character-level inputs**.
- Defines DataLoaders for training and validation.
- Gets model inputs ready for batch-based training using PyTorch.

```
class CharEmbedding(nn.Module):
    def forward(self, x_char_ids):
        ...
        # char CNN + max-pool
        return final_char_emb
```

```
class BiDAFAttention(nn.Module):
    def forward(self, C_contextual, Q_contextual, C_mask, Q_mask):
        ...
        return G
```

BiDAF Initialization

```
model = BiDAF(
    word_vocab_size=WORD_VOCAB_SIZE,
    word_embedding_dim=WORD_EMBEDDING_DIM,
    pretrained_word_embeddings=pretrained_word_embeddings,
    word_padding_idx=WORD_PADDING_IDX,
    char_vocab_size=CHAR_VOCAB_SIZE,
    char_embedding_dim=CHAR_EMBEDDING_DIM,
    char_cnn_out_channels=CHAR_CNN_OUT_CHANNELS,
    char_cnn_kernel_size=CHAR_CNN_KERNEL_SIZE,
    char_padding_idx=CHAR_PADDING_IDX,
    hidden_size=HIDDEN_SIZE,
    num_highway_layers=NUM_HIGHWAY_LAYERS,
    dropout_rate=DROPOUT_RATE
).to(device)
```

Model Setup
Instantiates the **BiDAF model** with word & character embeddings, highway layers, LSTMs, and attention.

Loss & Optimizer
Uses CrossEntropyLoss for start/end span prediction and Adam optimizer with learning rate scheduler.

Training Loop

- Runs BiDAF forward pass on each batch
- Calculates loss for predicted start/end positions
- Applies gradient clipping to stabilize training
- Tracks progress with tqdm

```
# --- Loss Function and Optimizer ---
criterion = nn.CrossEntropyLoss(ignore_index=-1) # Use -1 if your targets might have it for unanswerable, though SQuAD 1.1 shouldn't
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, verbose=True)
```

Evaluation

Evaluation Loop

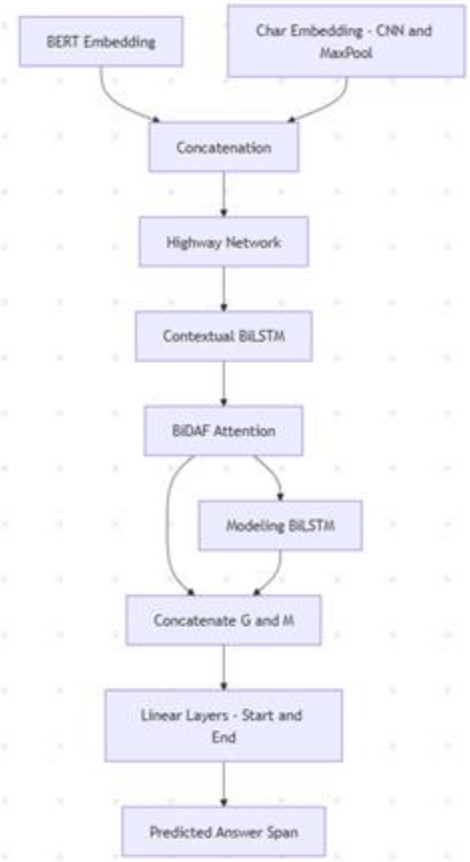
- Runs inference without gradients
- Computes **token-level EM and F1** for each prediction
- Saves model checkpoint with best validation F1

```
Train Loss: 1.2436
Val Loss: 4.0594 | Val EM: 42.86% | Val F1: 59.38%

Training complete.
Best Validation F1: 60.65% (Model saved at ./bidaf_best_model.pt)
```

BiDAF with BERT embeddings

- **CharEmbedding**
  CNN + max-pooling over character tokens → captures subword patterns.
- **HighwayNetwork**
  Allows the model to flexibly combine raw and transformed embeddings.
- **BiDAFAttention**
  Computes attention between context and question using similarity matrix.
- **BiDAF_BERT_Char (Main Model)**
  - Inputs: BERT token IDs, attention masks, and char-level IDs.
  - BERT provides semantic context, char-CNN adds fine-grained features.
  - Outputs: start and end logits over context tokens.



Evaluation

Epoch 9 results:

exact match – 54.15%

F1 score – 72.68%

Already better than **bidaf with glove.**

```
def evaluate(model, dataloader, criterion, device):
    model.eval()
    epoch_loss = 0
    all_pred_starts, all_pred_ends = [], []
    all_true_starts, all_true_ends = [], []

    progress_bar = tqdm(dataloader, desc="Evaluating", leave=False, dynamic_ncols=True)

    with torch.no_grad():
        for batch in progress_bar:
            context_ids = batch['context_input_ids'].to(device)
            context_mask = batch['context_attention_mask'].to(device)
            context_char_ids = batch['context_token_char_ids'].to(device)
            question_ids = batch['question_input_ids'].to(device)
            question_mask = batch['question_attention_mask'].to(device)
            question_char_ids = batch['question_token_char_ids'].to(device)
            true_start_indices = batch['start_token_bert'].to(device)
            true_end_indices = batch['end_token_bert'].to(device)

            start_logits, end_logits = model(
                context_bert_ids=context_ids, context_bert_mask=context_mask, context_bert_char_ids=context_char_ids,
                question_bert_ids=question_ids, question_bert_mask=question_mask, question_bert_char_ids=question_char_ids
            )

            loss_start = criterion(start_logits, true_start_indices)
            loss_end = criterion(end_logits, true_end_indices)
            total_loss = loss_start + loss_end

            if not torch.isnan(total_loss): # Only accumulate if loss is valid
                epoch_loss += total_loss.item()

            pred_start_batch = torch.argmax(start_logits, dim=1)
            pred_end_batch = torch.argmax(end_logits, dim=1)

            all_pred_starts.extend(pred_start_batch.cpu().tolist())
            all_pred_ends.extend(pred_end_batch.cpu().tolist())
            all_true_starts.extend(true_start_indices.cpu().tolist())
            all_true_ends.extend(true_end_indices.cpu().tolist())

            progress_bar.set_postfix({'loss': f'{total_loss.item() if not torch.isnan(total_loss) else "NaN":.4f}'})

    avg_loss = epoch_loss / len(dataloader) if len(dataloader) > 0 else 0.0
    em, f1 = compute_metrics(all_pred_starts, all_pred_ends, all_true_starts, all_true_ends)
```

Part 2 - Data extraction

This code snippet loads and preprocesses the first 30,000 examples from the SQuAD training set:

- load_dataset("squad", split="train[:30000]"): loads the first 30k examples from the SQuAD train split.
- corpus: extracts the context passages.
- questions: extracts the questions.
- references: extracts the first reference answer if available, otherwise an empty string (to handle missing answers).

```
squad = load_dataset("squad", split="train[:30000]")

# Extract all relevant fields for later evaluation
corpus = [item["context"] for item in squad]
questions = [item["question"] for item in squad]
references = [item["answers"]["text"][0] if item["answers"]["text"] else "" for item in squad]  # avoid empty answers
```

Retrievers

```python
# TF-IDF Sparse Retriever
class SparseRetriever:
    def __init__(self, docs):
        self.vectorizer = TfidfVectorizer().fit(docs)
        self.doc_vectors = self.vectorizer.transform(docs)
        self.docs = docs

    def retrieve(self, query, k=5):
        q_vec = self.vectorizer.transform([query])
        scores = np.dot(self.doc_vectors, q_vec.T).toarray().squeeze()
        top_k_idx = scores.argsort()[-k:][::-1]
        return [self.docs[i] for i in top_k_idx]
```

TF-IDF Sparse Retriever

- Uses TfidfVectorizer to convert documents and query into sparse vectors.
- Calculates cosine similarity via dot product.
- Returns top-**k** documents based on TF-IDF relevance.

SBERT Dense Retriever

- Uses SentenceTransformer (multi-qa-MiniLM-L6-cos-v1) for dense semantic embeddings.
- Applies semantic_search to find top-**k** semantically closest documents.
- More powerful for capturing meaning beyond keywords.

```python
# SBERT Dense Retriever
class DenseRetriever:
    def __init__(self, docs):
        self.model = SentenceTransformer('multi-qa-MiniLM-L6-cos-v1')
        self.docs = docs
        self.doc_embeddings = self.model.encode(docs, convert_to_tensor=True)

    def retrieve(self, query, k=5):
        query_embedding = self.model.encode(query, convert_to_tensor=True)
        hits = util.semantic_search(query_embedding, self.doc_embeddings, top_k=k)[0]
        return [self.docs[hit['corpus_id']] for hit in hits]
```

T5Generator (Generative QA)

T5-base, a pre-trained encoder-decoder model, to generate answers.

- **Tokenizer**: Prepares the input (question + docs) in T5 format.
- **Model**: Generates an answer in text form.
- **Output**: Decoded text answer.

Why T5?
T5 is strong at **generative** tasks. Instead of extracting spans, it **writes answers** based on understanding context—ideal when answers aren't exact spans or when summarization helps.

```
# Combine documents into context
def augment_context(query, docs):
    return query + " " + " ".join(docs)

# Generative QA (T5 example)
class T5Generator:
    def __init__(self):
        self.tokenizer = T5Tokenizer.from_pretrained("t5-base")
        self.model = T5ForConditionalGeneration.from_pretrained("t5-base")

    def generate(self, input_text, max_len=64):
        inputs = self.tokenizer("question: " + input_text, return_tensors="pt", truncation=True)
        output = self.model.generate(**inputs, max_length=max_len)
        return self.tokenizer.decode(output[0], skip_special_tokens=True)
```

BERT for Extractive QA

```python
from transformers import AutoTokenizer, AutoModelForQuestionAnswering

# Better extractive QA model
tokenizer_bert = AutoTokenizer.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")
model_bert = AutoModelForQuestionAnswering.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")
model_bert.eval()
```

This snippet loads a **pretrained extractive QA model**:

- **Model**: bert-large-uncased-whole-word-masking-finetuned-squad
  - Fine-tuned specifically on the SQuAD dataset.
  - Outputs **start and end positions** of the answer span.

- **Why use it?**
  - It's a **stronger baseline** than vanilla BiDAF.
  - Leverages deep contextual understanding without building BiDAF from scratch.

Results

```
query = "Who was the first president of the United States?"
answer = rag_pipeline(query, retriever, answer_mode='generate')
print("Answer:", answer)
```

```
Answer: George Washington
```

```
query = "What is the chemical symbol for hydrogen?"
answer = rag_pipeline(query, retriever, answer_mode='generate')
print("Answer:", answer)
```

```
Answer: H
```

Results

```
query = "Where is the Eiffel Tower located?"
answer = rag_pipeline(query, retriever, answer_mode='generate')
print("Answer:", answer)
```

```
Answer: Notre Dame cathedral
```

```
query = "When did World War II end?"
answer = rag_pipeline(query, retriever, answer_mode='generate')
print("Answer:", answer)
```

```
Answer: September 11, 1776
```

Evaluation

```
SQuAD Evaluation:
Exact Match (EM): 48.00
F1 Score: 54.13
```

- **Exact Match (EM): 48.00**
  48% of predicted answers exactly matched the ground truth span.
- **F1 Score: 54.13**
  Measures overlap between predicted and actual answers (word-level). Higher than EM, as partial matches still count.

```
Sparse Retriever Evaluation:
Retriever Evaluation: Recall@5: 62.71%
Dense Retriever Evaluation:
Retriever Evaluation: Recall@5: 70.41%
```

- **Sparse Retriever (TF-IDF)**
  Recall@5: 62.71%
  → Relevant doc was in the top 5 ~63% of the time.

- **Dense Retriever (SBERT)**
  Recall@5: 70.41%
  → SBERT outperfomed TF-IDF, finding relevant docs more reliably.

Thank you!