Embedding

October 23, 2024

[2]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
import random
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
```

[3]:
```
# Dataset class for arithmetic problems
class ArithmeticDataset(Dataset):
    def __init__(self, max_length=20, num_samples=1000):
        self.max_length = max_length
        self.num_samples = num_samples
        # Vocabulary: 0-9 for digits, 10 for '+', 11 for '=', 12 for padding,␣
↪13 for EOS
        self.vocab = {str(i): i for i in range(10)}
        self.vocab.update({'+': 10, '=': 11, '<PAD>': 12, '<EOS>': 13})
        self.inv_vocab = {v: k for k, v in self.vocab.items()}
        self.data = self.generate_data()

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

    def __getitem__(self, idx):
        return self.data[idx]

    def generate_number(self, length):
        return random.randint(10**(length-1), 10**length - 1)
```

```
    def tokenize(self, s):
        return [self.vocab[c] for c in s if c in self.vocab]

    def pad_sequence(self, seq, max_length):
        return seq + [self.vocab['<PAD>']] * (max_length - len(seq))

    def decode(self, tensor):
        return ''.join(self.inv_vocab[t.item()] for t in tensor if t.item() not␣
↪in
                       [self.vocab['<PAD>'], self.vocab['<EOS>']])[::-1]

    def generate_data(self):
        data = []
        samples_per_combination = max(1, self.num_samples // (self.max_length␣
↪** 2))

        for i in range(1, self.max_length + 1):
            for j in range(1, self.max_length + 1):
                for _ in range(samples_per_combination):
                    num1 = self.generate_number(i)
                    num2 = self.generate_number(j)
                    result = num1 + num2

                    # Create reversed input string
                    input_str = f"{num1}+{num2}="
                    input_str = input_str[::-1]

                    # Create reversed target string
                    target_str = f"{result}"[::-1]

                    # Tokenize and pad
                    input_tokens = self.tokenize(input_str)
                    target_tokens = self.tokenize(target_str) + [self.
↪vocab['<EOS>']]
                    max_seq_length = self.max_length * 2 + 2

                    input_padded = self.pad_sequence(input_tokens,␣
↪max_seq_length)
                    target_padded = self.pad_sequence(target_tokens,␣
↪max_seq_length)

                    # Convert to tensors
                    input_tensor = torch.tensor(input_padded, dtype=torch.long)
                    target_tensor = torch.tensor(target_padded, dtype=torch.
↪long)

                    data.append((input_tensor, target_tensor))
```

```
        return data
```

[9]:
```
# Model Architecture Components
class AbacusEmbedding(nn.Module):
    def __init__(self, vocab_size, embed_size, max_length):
        super().__init__()
        # Create an embedding layer for the input tokens
        self.embed = nn.Embedding(vocab_size, embed_size)
        # Create a separate embedding layer for positional encodings
        self.pos_embed = nn.Embedding(max_length, embed_size)
        self.max_length = max_length

    def forward(self, x):
        # Get the sequence length of the input
        seq_length = x.size(1)

        # Generate position indices
        pos = torch.arange(seq_length, device=x.device).unsqueeze(0)

        # Truncate positions to max_length
        # This ensures that positions beyond max_length use the same embedding
        pos = torch.clamp(pos, max=self.max_length - 1)

        # Get the token embeddings
        embedded = self.embed(x)

        # Get the positional embeddings
        positional = self.pos_embed(pos)

        # Combine token embeddings and positional embeddings
        return embedded + positional[:, :seq_length]

class SmallTransformer(nn.Module):
    def __init__(self, vocab_size, embed_size, num_heads, ff_dim, num_layers,␣
↪max_length):
        super().__init__()
        # Initialize the custom Abacus Embedding layer
        self.embedding = AbacusEmbedding(vocab_size, embed_size, max_length)

        # Create a single Transformer encoder layer
        self.transformer_layer = nn.TransformerEncoderLayer(
            d_model=embed_size,
            nhead=num_heads,
            dim_feedforward=ff_dim,
            batch_first=True
        )
```

```python
        # Create the full Transformer encoder by stacking multiple layers
        self.transformer = nn.TransformerEncoder(self.transformer_layer,␣
↪num_layers=num_layers)

        # Final linear layer to project to vocabulary size
        self.fc_out = nn.Linear(embed_size, vocab_size)

    def forward(self, x):
        try:
            # Apply Abacus Embedding
            x = self.embedding(x)

            # Pass through the Transformer encoder
            x = self.transformer(x)

            # Project to vocabulary size
            return self.fc_out(x)
        except Exception as e:
            print(f"Error in SmallTransformer forward pass: {str(e)}")
            raise e
```

[10]:
```python
# Function to load a given model
def load_arithmetic_model(model_path):
    # Load the saved model
    checkpoint = torch.load(model_path)

    # Recreate the model architecture
    model = SmallTransformer(
        checkpoint['vocab_size'],
        checkpoint['embed_size'],
        checkpoint['num_heads'],
        checkpoint['ff_dim'],
        checkpoint['num_layers'],
        checkpoint['max_seq_length']
    )


    model.load_state_dict(checkpoint['model_state_dict'])
    model.eval()
    return model

# Test function for trying an addition problems
def test_addition(model, dataset, num1, num2):
    """Test the model on a specific addition problem"""
    # Create input string in the same format as training data
    input_str = f"{num1}+{num2}="
```

```python
    input_str = input_str[::-1]  # Reverse the string

    # Tokenize and pad
    input_tokens = dataset.tokenize(input_str)
    max_seq_length = dataset.max_length * 2 + 2
    input_padded = dataset.pad_sequence(input_tokens, max_seq_length)
    input_tensor = torch.tensor([input_padded], dtype=torch.long)

    # Generate prediction
    with torch.no_grad():
        output = model(input_tensor)
        predicted = torch.argmax(output, dim=-1)
        result = dataset.decode(predicted[0])

    print(f"\nInput: {num1} + {num2} = ?")
    print(f"True result: {num1 + num2}")
    print(f"Model prediction: {result}")
    print(f"Correct: {int(result) == num1 + num2}")
```

[11]:
```python
# Load model
loaded_model = load_arithmetic_model('trained_addition_model.pth')

# Create dataset instance
dataset = ArithmeticDataset(max_length=20, num_samples=200_000)

# Test cell - try a few addition problems
test_addition(loaded_model, dataset, 123, 456)
#test_addition(loaded_model, dataset, 45, 67)
#test_addition(loaded_model, dataset, 1234, 5678)
```

```
Input: 123 + 456 = ?
True result: 579
Model prediction: 579
Correct: True
```

[ ]:
```python

```