Final Exam

Tchelidze Lasha-Giorgi

Task1

Comprehensive Description of the Transformer Neural Network

Transformers are a class of neural network architectures designed to handle sequence data (text, time series, logs, etc.) using attention instead of recurrence or convolutions. They were introduced in the paper "Attention Is All You Need" (Vaswani et al., 2017) and are now the backbone of large language models (BERT, GPT, etc.).

Core idea: Replace recurrent and convolutional layers entirely with self-attention mechanisms that allow the model to weigh the importance of different words (or tokens) in a sequence relative to each other, regardless of their distance.

Main components:

1. **Input Embedding + Positional Encoding** Tokens are converted to dense vectors (embeddings). Since Transformers have no built-in notion of order, fixed or learned positional encodings are added to the embeddings.
2. **Multi-Head Self-Attention** The heart of the architecture. For each token, three vectors are computed: Query (Q), Key (K), and Value (V). Attention scores are calculated as:

   Attention(Q, K, V) = softmax(QK^T / √d_k) · V

   Multi-head means several attention operations run in parallel, allowing the model to attend to different representation subspaces.

3. **Feed-Forward Neural Network (FFN)** After attention, each position goes through a position-wise fully connected feed-forward network (two linear layers with ReLU or GELU in between).
4. **Layer Normalization and Residual Connections** Each sub-layer (attention and FFN) is wrapped in a residual connection followed by layer normalization: LayerNorm(x + Sublayer(x))
5. **Encoder and Decoder Stacks**
   - Encoder: N identical layers (typically 6–24) of multi-head self-attention + FFN.
   - Decoder: N identical layers with three sub-layers: masked self-attention (causal, prevents seeing future tokens), encoder-decoder attention (queries from decoder, keys/values from encoder), and FFN.
6. **Scaled Dot-Product Attention** The scaling factor √d_k prevents vanishing gradients when dimensions are large.

Key advantages:

- Fully parallelizable (unlike RNNs)

- O(1) time complexity for distance between tokens (vs linear in RNNs)
- Captures long-range dependencies effortlessly

Modern variants (GPT, Llama) use only the decoder part (decoder-only Transformers) with causal masking for autoregressive generation.

```
# !pip install torch torchtext pandas scikit-learn

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import re

# ------------------------------
# 1. Load and preprocess data
# ------------------------------
# Download dataset: https://www.kaggle.com/datasets/shashwatwork/phishing-dataset-for-machine-learning
# Here we assume you have 'Phishing_Legitimate_full.csv' or similar

# For reproducibility, we'll create a small synthetic sample
import numpy as np
np.random.seed(42)

def create_sample_data(n_samples=5000):
    legit_domains = ['google.com', 'github.com', 'stackoverflow.com', 'wikipedia.org', 'amazon.com']
    phishing_patterns = ['login', 'secure', 'account', 'update', 'verify', 'bank', 'paypal', 'webscr']

    data = []
    for _ in range(n_samples // 2):
        # Legitimate
        url = f"https://www.{np.random.choice(legit_domains)}/{np.random.choice(['login', 'search', 'profile', 'settings'])}"
        data.append((url, 0))
        # Phishing-like
        fake_domain = ''.join(np.random.choice(list('abcdefghijklmnopqrstuvwxyz0123456789'), 10)) + '.tk'
        suspicious_path = '-'.join(np.random.choice(phishing_patterns, 3))
        url = f"http://{fake_domain}/session/{suspicious_path}@www.{np.random.choice(legit_domains)}"
        data.append((url, 1))
    return pd.DataFrame(data, columns=['url', 'label'])
```

```
df = create_sample_data(8000)
print(df['label'].value_counts())

# ------------------------------
# 2. Tokenization & Vocabulary
# ------------------------------
tokenizer = get_tokenizer('basic_english')

def tokenize_url(url):
    # Simple character-level + some heuristics
    url = re.sub(r'(http|https)://', '', url.lower())
    url = re.sub(r'[./?=&]', ' ', url)
    tokens = list(url)  # character level
    tokens.extend(tokenizer(url))  # also add word-level
    return tokens

def yield_tokens(data_iter):
    for url in data_iter:
        yield tokenize_url(url)

vocab = build_vocab_from_iterator(yield_tokens(df['url']), specials=['<unk>', '<pad>'])
vocab.set_default_index(vocab['<unk>'])

# ------------------------------
# 3. Dataset class
# ------------------------------
class URLDataset(Dataset):
    def __init__(self, urls, labels, vocab, max_len=128):
        self.urls = urls
        self.labels = labels
        self.vocab = vocab
        self.max_len = max_len

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

    def __getitem__(self, idx):
        tokens = tokenize_url(self.urls[idx])[:self.max_len]
        ids = [vocab[t] for t in tokens]
        if len(ids) < self.max_len:
            ids += [vocab['<pad>']] * (self.max_len - len(ids))
        return torch.tensor(ids, dtype=torch.long), torch.tensor(self.labels[idx], dtype=torch.float)

# Train/validation split
X_train, X_val, y_train, y_val = train_test_split(df['url'], df['label'], test_size=0.2, random_state=42,
stratify=df['label'])
```

```
train_dataset = URLDataset(X_train.values, y_train.values, vocab)
val_dataset   = URLDataset(X_val.values, y_val.values, vocab)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader   = DataLoader(val_dataset, batch_size=64)

# ----------------------------
# 4. Transformer Encoder Classifier
# ----------------------------
class TransformerURLClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, n_heads=8, n_layers=4, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=vocab['<pad>'])
        self.pos_encoder = nn.Embedding(128, embed_dim)  # simple learned positional encoding

        encoder_layer = nn.TransformerEncoderLayer(d_model=embed_dim, nhead=n_heads,
                                                   dim_feedforward=512, dropout=dropout, batch_first=True)
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=n_layers)

        self.classifier = nn.Sequential(
            nn.Linear(embed_dim, 256),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(256, 1)
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        batch_size, seq_len = x.shape
        positions = torch.arange(seq_len, device=x.device).expand(batch_size, seq_len)

        x = self.embedding(x) + self.pos_encoder(positions)
        x = self.transformer(x)          # (B, L, D)
        x = x.mean(dim=1)                # global average pooling over sequence
        x = self.dropout(x)
        logits = self.classifier(x).squeeze(-1)
        return logits

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = TransformerURLClassifier(vocab_size=len(vocab)).to(device)

criterion = nn.BCEWithLogitsLoss()
optimizer = optim.AdamW(model.parameters(), lr=3e-4, weight_decay=1e-2)

# ----------------------------
# 5. Training loop
# ----------------------------
def train_epoch(model, loader, optimizer, criterion):
```

```
    model.train()
    total_loss = 0
    for ids, labels in loader:
        ids, labels = ids.to(device), labels.to(device)

        optimizer.zero_grad()
        logits = model(ids)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    return total_loss / len(loader)

def evaluate(model, loader):
    model.eval()
    preds, trues = [], []
    with torch.no_grad():
        for ids, labels in loader:
            ids = ids.to(device)
            logits = model(ids)
            pred = (torch.sigmoid(logits) > 0.5).cpu().numpy()
            preds.extend(pred)
            trues.extend(labels.numpy())
    return accuracy_score(trues, preds)

# Train
for epoch in range(8):
    loss = train_epoch(model, train_loader, optimizer, criterion)
    acc = evaluate(model, val_loader)
    print(f"Epoch {epoch+1:02d} | Loss: {loss:.4f} | Val Accuracy: {acc:.4f}")

# ----------------------------
# 6. Inference example
# ----------------------------
def predict_url(url):
    model.eval()
    ids = torch.tensor([train_dataset[0][0][:128]], dtype=torch.long).to(device)  # dummy shape
    tokens = tokenize_url(url)[:128]
    ids = [vocab[t] if t in vocab else vocab['<unk>'] for t in tokens]
    if len(ids) < 128:
        ids += [vocab['<pad>']] * (128 - len(ids))
    ids = torch.tensor([ids], dtype=torch.long).to(device)

    with torch.no_grad():
        logit = model(ids)
        prob = torch.sigmoid(logit).item()
    return "PHISHING" if prob > 0.5 else "LEGITIMATE", prob
```

```
test_urls = [
    "https://www.google.com/search?q=hello",
    "http://paypal-security-update-2025.tk/verify/login.php",
    "https://github.com/torvalds/linux"
]

for u in test_urls:
    label, prob = predict_url(u)
    print(f"{u}\n → {label} (confidence: {prob:.4f})\n")
```