Task8

March 25, 2025

1 Task 8 : Vision Transformer

We will start by importinh all the requires libraries

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

1.1 Transformer Block

Multi-Head Self-Attention: Helps the model focus on different parts of the image.

Layer Normalization: Normalizes data for stable learning.

MLP (Feedforward Neural Network): Learns deep representations.

Residual Connections: Adds previous layer's output back to avoid loss of information.

[2]:
```
class TransformerBlock(nn.Module):
    def __init__(self, embed_dim, num_heads, mlp_dim, dropout=0.1):
        super(TransformerBlock, self).__init__()
        self.norm1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, mlp_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(mlp_dim, embed_dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        # x shape:(seq_len,batch_size,embed_dim)
        x2 = self.norm1(x)
```

```
        attn_output, _ = self.attn(x2, x2, x2)
        x = x + attn_output  # Residual connection
        x2 = self.norm2(x)
        x = x + self.mlp(x2)  # Residual connection
        return x
```

1.2 Vision Transformer

Patch Embedding Layer: Converts an image into small patches using Conv2D.

Class Token: A special token that helps in classification.

Positional Embeddings: Helps the model understand patch positions.

Transformer Blocks: A sequence of TransformerBlock layers.

Normalization & Head Layer: Normalizes the data and applies a final linear layer to classify digits (0-9).

1.2.1 Forward pass

Extract Patches → Flatten → Embed.

Add Class Token.

Apply Transformer Blocks.

Extract Class Token Output and pass it through a fully connected layer.

[3]:
```
class VisionTransformer(nn.Module):
    def __init__(self, image_size=28, patch_size=7, in_channels=1,␣
↪num_classes=10,
                 embed_dim=64, depth=6, num_heads=4, mlp_dim=128, dropout=0.1):
        super(VisionTransformer, self).__init__()

        assert image_size % patch_size == 0, "Image dimensions must be␣
↪divisible by the patch size."
        self.num_patches = (image_size // patch_size) ** 2

        # Patch embedding using a convolution (acts as patch extractor and␣
↪linear projection)
        self.patch_embed = nn.Conv2d(in_channels, embed_dim,␣
↪kernel_size=patch_size, stride=patch_size)

        # token and positional embeddings
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(torch.zeros(1, self.num_patches + 1,␣
↪embed_dim))
        self.pos_drop = nn.Dropout(dropout)

        self.blocks = nn.ModuleList([
```

```
            TransformerBlock(embed_dim, num_heads, mlp_dim, dropout)
            for _ in range(depth)
        ])

        self.norm = nn.LayerNorm(embed_dim)

        # Classification
        self.head = nn.Linear(embed_dim, num_classes)

        # parameters initialzised
        nn.init.trunc_normal_(self.pos_embed, std=0.02)
        nn.init.trunc_normal_(self.cls_token, std=0.02)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            nn.init.trunc_normal_(m.weight, std=0.02)
            if m.bias is not None:
                nn.init.zeros_(m.bias)
        elif isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode='fan_out')
            if m.bias is not None:
                nn.init.zeros_(m.bias)

    def forward(self, x):
        B = x.shape[0]
        # x: (B, in_channels, image_size, image_size)
        x = self.patch_embed(x)  # shape: (B, embed_dim, H', W')
        x = x.flatten(2).transpose(1, 2)  # shape: (B, num_patches, embed_dim)


        cls_tokens = self.cls_token.expand(B, -1, -1)  # shape: (B, 1, embed_dim)
        x = torch.cat((cls_tokens, x), dim=1)  # shape: (B, num_patches+1, embed_dim)

        x = x + self.pos_embed
        x = self.pos_drop(x)

        # Transformer expects input shape (sequence_length, batch_size, embed_dim)
        x = x.transpose(0, 1)
        for block in self.blocks:
            x = block(x)
        x = self.norm(x)

        # Use class token output for classification
```

```
        cls_output = x[0]  # shape: (B, embed_dim)
        logits = self.head(cls_output)
        return logits
```

1.3 Loading the MNIST dataset

[4]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_dataset = datasets.MNIST(root="./data", train=True, download=True,␣
 ↪transform=transform)
test_dataset  = datasets.MNIST(root="./data", train=False, download=True,␣
 ↪transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

1.4 Train and Test

[5]:
```
model = VisionTransformer().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()


def train(model, loader, optimizer, criterion, device):
    model.train()
    total_loss = 0
    for data, target in loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * data.size(0)
    return total_loss / len(loader.dataset)

# Evaluation loop
def evaluate(model, loader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
```

```
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()
            total += data.size(0)
    return correct / total
```

1.5 Train the model

[6]:
```
num_epochs = 5
for epoch in range(1, num_epochs + 1):
    loss = train(model, train_loader, optimizer, criterion, device)
    acc = evaluate(model, test_loader, device)
    print(f"Epoch {epoch}: Train Loss = {loss:.4f}, Test Accuracy = {acc*100:.
 ↪2f}%")
```

```
Epoch 1: Train Loss = 0.6251, Test Accuracy = 92.80%
Epoch 2: Train Loss = 0.2130, Test Accuracy = 95.74%
Epoch 3: Train Loss = 0.1655, Test Accuracy = 96.28%
Epoch 4: Train Loss = 0.1385, Test Accuracy = 96.79%
Epoch 5: Train Loss = 0.1200, Test Accuracy = 97.04%
```

2 Quantum Vision Transformer

QVT is a model that provides transformer architecture with quantum computing

2.1 Architecture

2.1.1 1. Input and Patch Embedding (Classical)

- Input: MNIST image of size 28×28.
- Patch Extraction: Divide the image into **7×7 patches** (16 patches in total).
- Linear Projection: Map each patch (**flattened 49 pixels**) into a higher-dimensional embedding (e.g., 64 dimensions).

2.1.2 2. Quantum Encoding Layer

- Encoding Circuit:
  - For each patch embedding vector

    $$x \in \mathbb{R}^{64}$$

    , use a quantum circuit where each element of the vector controls rotation gates (e.g., R_y(theta) gates).
  - This encodes the classical data into a quantum state.
- Quantum Register:
  - Use a set of qubits sufficient to represent the embedding.
  - Techniques like qubit re-use or amplitude encoding can be used to reduce qubit count.

2.1.3 3. Quantum Transformer Block

2.1.4 Quantum Self-Attention

- Quantum Circuits for Q, K, V:
  - Create separate parameterized quantum circuits that transform the encoded patch state into quantum representations of queries (Q), keys (K), and values (V).
- Attention via Quantum Similarity:
  - Implement a subroutine (e.g., **swap test**) that calculates the **similarity (overlap)** between the quantum states of queries and keys.
  - These similarity scores act as the **attention weights**.

2.1.5 Quantum Feed-Forward (Variational Circuit)

- Pass the weighted quantum states through another parameterized quantum circuit that mimics the function of a classical MLP.
- Hybrid Update: Use measurements to convert the quantum information back to **classical data before feeding it to the next block (or to a classical MLP).

2.1.6 4. Classification Head (Classical)

- Aggregate the processed information from the class token (or a pooled representation of the patch outputs).
- Final Linear Layer: Feed the classical vector into a fully connected layer to obtain class logits for digit classification.

2.1.7 5. Training Considerations

- Hybrid Optimization: Train the entire network using a mix of classical backpropagation and quantum-specific gradient techniques (e.g., parameter shift rule)
- Resource Constraints:
  - Given current quantum hardware limitations the quantum circuits should be kept shallow and use few qubits
  - Simulation-based experiments on classical hardware can be used during the prototyping phase.