E0 294: Systems for Machine Learning Assignment 2

Rohit Jorige (22166)
Indian Institute of Science, Bangalore

February 17, 2025

Problem 1

a) Two Possible Configurations for Weights and Layer

Consider an input of size 8×8 with 5 channels and a desired output of dimensions $5 \times 5 \times 3$. The ouput dimension is given by:

$$\text{Output dimension} = \left\lfloor \frac{\text{Input dimension} + 2p - k}{s} \right\rfloor + 1,$$

where k is the kernel size (assuming square kernels), s is the stride, and p is the zero-padding.

Configuration 1:
Let the stride be $s = 1$ and use no padding ($p = 0$). Then, for the spatial dimension we have:

$$5 = (8 - k + 1) \quad \Longrightarrow \quad k = 4.$$

Thus, using a 4×4 kernel will produce an output of size 5×5. Since there are 3 filters (to produce 3 output channels) and the input has 5 channels, the weight tensor will have dimensions:

$$[\, C_{\text{out}} \times C_{\text{in}} \times k \times k \,] = [3 \times 5 \times 4 \times 4].$$

Configuration 2:
Alternatively, let us set $s = 2$ and $p = 1$. Then the output spatial dimension is given by:

$$5 = \left\lfloor \frac{8 + 2(1) - k}{2} \right\rfloor + 1 = \left\lfloor \frac{10 - k}{2} \right\rfloor + 1.$$

For exact division, set

$$\frac{10 - k}{2} + 1 = 5 \quad \Longrightarrow \quad \frac{10 - k}{2} = 4 \quad \Longrightarrow \quad k = 2.$$

Thus, using a 2×2 kernel with $s = 2$ and $p = 1$ will also produce an output of 5×5. In this case, the weight tensor has dimensions:

$$[3 \times 5 \times 2 \times 2].$$

Feasibility of 3×3 Kernels:
For a 3×3 kernel ($k = 3$) with $s = 2$, we use the formula:

$$5 = \left\lfloor \frac{8 + 2p - 3}{2} \right\rfloor + 1 = \left\lfloor \frac{5 + 2p}{2} \right\rfloor + 1.$$

Choosing $p = 2$ gives:

$$\left\lfloor \frac{5 + 4}{2} \right\rfloor + 1 = \left\lfloor \frac{9}{2} \right\rfloor + 1 = 4 + 1 = 5.$$

Thus, a 3×3 kernel is feasible provided we choose $s = 2$ and $p = 2$, resulting in weight dimensions:

$$[3 \times 5 \times 3 \times 3].$$

Feasibility of 2×2 Kernels:
As shown in Configuration 2, a 2×2 kernel works with $s = 2$ and $p = 1$.

b) Final Dimension Calculation

Given an input of size 13×15 with 3 channels, we apply the following operations sequentially:

Step 1: Convolution with 7 Filters of Dimension 4×6 (stride=1, no zero padding)
The output dimensions are given by:

$$H_{out1} = 13 - 4 + 1 = 10, \quad W_{out1} = 15 - 6 + 1 = 10.$$

Thus, the output of this layer is 10×10 with 7 channels.

Step 2: Convolution with 8 Filters of Dimension 3×4 (stride=2, padding=1)
The input to this layer is the output from Step 1 (10×10 with 7 channels). Using the convolution formula:

$$H_{out2} = \left\lfloor \frac{10 + 2(1) - 3}{2} \right\rfloor + 1 = \left\lfloor \frac{9}{2} \right\rfloor + 1 = 4 + 1 = 5,$$

$$W_{out2} = \left\lfloor \frac{10 + 2(1) - 4}{2} \right\rfloor + 1 = \left\lfloor \frac{8}{2} \right\rfloor + 1 = 4 + 1 = 5.$$

Thus, the output after this convolution is 5×5 with 8 channels.

Step 3: Max-Pooling with a 2×2 Filter
Assuming the max-pooling uses a stride equal to the filter size (i.e., stride=2), the output dimensions are computed as:

$$H_{out3} = \left\lfloor \frac{5 - 2}{2} \right\rfloor + 1 = \left\lfloor \frac{3}{2} \right\rfloor + 1 = 1 + 1 = 2,$$

$$W_{out3} = \left\lfloor \frac{5 - 2}{2} \right\rfloor + 1 = 2.$$

The number of channels remains unchanged.

Thus, the final output dimensions are:

$$2 \times 2 \times 8.$$

But if we have stride 1 max - pooling then we get :

$$4 \times 4 \times 8.$$

Problem 2

a) Implementing LeNet-5 in PyTorch

- **Input:** $1 \times 32 \times 32$ images (1 channel, 32 pixels width, 32 pixels height).
- **C1:** Convolution layer with 6 feature maps (filters), kernel size 5×5, producing $6 \times 28 \times 28$.
- **S2:** Average Pooling with kernel size 2×2 and stride 2, reducing to $6 \times 14 \times 14$.
- **C3:** Convolution layer with 16 feature maps, kernel size 5×5, producing $16 \times 10 \times 10$.
- **S4:** Average Pooling with kernel size 2×2 and stride 2, reducing to $16 \times 5 \times 5$.
- **F5:** Fully connected layer with 120 outputs.
- **F6:** Fully connected layer with 84 outputs.
- **Output:** Typically, a fully connected layer with 10 outputs (for 10 classes).

In this assignment, i have replace the final linear classification layer with a custom *RBF layer* to replicate the structure described in the paper.

Radial Basis Function (RBF) Layer

- Instead of projecting the last hidden features to logits with a linear layer, we compute distances to learned *centers* (one center per class, or more if desired).

Implementation details:

- Suppose the layer input is $B \times D$ (batch size B, feature dimension D).
- We maintain C centers, each of dimension D. Hence the centers form a $C \times D$ matrix.
- For each sample, we compute the squared Euclidean distance to each center, yielding a $B \times C$ distance matrix.
- The final classification decision can be made by taking $\arg\min$ of the distance vector, or via a custom loss function.

MAP Loss (Equation 9) in the LeNet-5 paper

In place of a standard cross-entropy or MSE, we use a custom loss inspired by the Maximum A Posteriori (MAP) formulation with an added term that encourages separation between classes and penalizes incorrect classes. The loss is computed as:

$$E(W) = \frac{1}{P} \sum_{i=1}^{P} \Big(y_i^{\mathrm{D}}(Z^p, W) + \log \big(e^{-j} + \sum_c e^{-(Z^p)_c} \big) \Big),$$

where y_i^{D} is the distance for the correct class, j is a constant, and $(Z^p)_c$ is the distance to class c. The idea is to minimize the distance to the correct class while limiting the distance from the other classes from collapsing.

Listing 1: RBF, MAPLoss, and LeNet-5 Classes

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# 1) RBF Layer
class RBFLayer(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        # Learnable centers: shape (out_features, in_features)
        self.centers = nn.Parameter(torch.randn(out_features, in_features))

    def forward(self, x):
        # x is (batch_size, in_features)
        # Compute squared Euclidean distance to each center
        x_expanded = x.unsqueeze(1).expand(-1, self.out_features, -1)
        centers_expanded = self.centers.unsqueeze(0).expand(x.size(0), -1, -1)
        distances = torch.sum((x_expanded - centers_expanded) ** 2, dim=2)
        return distances

# 2) MAPLoss (Equation 9)
class MAPLoss(nn.Module):
    def __init__(self, j=1.0):
        super().__init__()
        self.j = j

    def forward(self, outputs, targets):
        """
        outputs: (batch_size, num_classes) - RBF distances
        targets: (batch_size,) ground-truth class indices
        """
        batch_size = outputs.size(0)
        # Distance for the correct class for each sample
        correct_class_distances = outputs[torch.arange(batch_size), targets]

        # e^(-distance) for all classes
        exp_neg_distances = torch.exp(-outputs)

        # Convert self.j to a tensor on the same device
        j_tensor = torch.tensor(self.j, dtype=outputs.dtype,
            ↪ device=outputs.device)
        exp_neg_j = torch.exp(-j_tensor)

        # Summation term
        sum_exp_terms = exp_neg_distances.sum(dim=1) + exp_neg_j
        log_sum_exp_term = torch.log(sum_exp_terms)

        # Final MAP loss: mean over the batch
        loss = (correct_class_distances + log_sum_exp_term).mean()
```

```
        return loss

# 3) LeNet-5 Architecture
class LeNet5(nn.Module):
    def __init__(self):
        super().__init__()
        # Convolutional layers:
        # Input: 1x32x32 -> Conv -> 6x28x28 -> AvgPool -> 6x14x14
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        # Next: 6x14x14 -> Conv -> 16x10x10 -> AvgPool -> 16x5x5
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)

        # Fully connected layers:
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)

        # Replace final linear layer with an RBF layer
        self.rbf = RBFLayer(84, 10)

    def forward(self, x):
        # First convolution + Tanh + Pool
        x = torch.tanh(self.conv1(x))
        x = self.pool1(x)
        # Second convolution + Tanh + Pool
        x = torch.tanh(self.conv2(x))
        x = self.pool2(x)
        # Flatten
        x = torch.flatten(x, 1)
        # Fully connected layers + Tanh
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        # RBF distances
        x = self.rbf(x)
        return x
```

Assumptions/Changes made:

- I have split the training set into train and validation set in order to encourage early stopping.
- Training set: 54,000 images (90% of original 60,000)
- Validation set: 6,000 images (10% of original 60,000)
- Test set: 10,000 images
- Image preprocessing: All images resized to 32×32 pixels by adding padding $= 2$ to the 28×28 MNIST images in order to replicate the paper's original image dimensions. The training ran for 24 epochs (early stopping with respect to the validation set and reported an accuracy of 98.94.

b) Implementing Depth-wise Separable Convolution

Listing 2: ModifiedLeNet-5 Classes

```
class ModifiedLeNet5(nn.Module):
    def __init__(self):
        super().__init__()
        # For conv1:
        #   Depth-wise: input channels = 1, output channels = 1, kernel_size =
            ↪ 5.
        #   Point-wise: input channels = 1, output channels = 6.
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 1, kernel_size=5, stride=1, padding=0, groups=1), #
                ↪ Depth-wise
            nn.Conv2d(1, 6, kernel_size=1, stride=1, padding=0)        #
                ↪ Point-wise
        )
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)

        # For conv2:
        #   Depth-wise: input channels = 6, output channels = 6, kernel_size =
            ↪ 5, groups=6.
        #   Point-wise: input channels = 6, output channels = 16.
        self.conv2 = nn.Sequential(
            nn.Conv2d(6, 6, kernel_size=5, stride=1, padding=0, groups=6), #
                ↪ Depth-wise
            nn.Conv2d(6, 16, kernel_size=1, stride=1, padding=0)       #
                ↪ Point-wise
        )
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)

        # Fully connected layers and RBF output.
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.rbf = RBFLayer(84, 10)

    def forward(self, x):
        # Input x: (batch, 1, 32, 32)
        x = self.pool1(torch.tanh(self.conv1(x))) # After conv1: (batch, 6,
            ↪ 28, 28) -> Pool: (batch, 6, 14, 14)
        x = self.pool2(torch.tanh(self.conv2(x))) # After conv2: (batch, 16,
            ↪ 10, 10) -> Pool: (batch, 16, 5, 5)
        x = torch.flatten(x, 1)
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        x = self.rbf(x) # RBF distances (batch, 10)
        return x
```

The Training ran for 17 epochs (early stopping with respect to validation set) for a batch size of 64 and achieved a test accuracy of 98.48%. This shows that depth wise seperable acheived similar accuarcy as the normal LeNet-5 with lesser parameters and far less multiplication and additions.

(Refer to attached code file for complete code.)

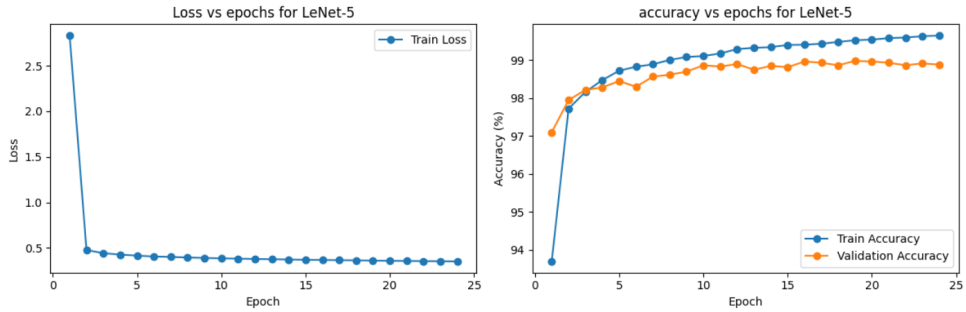

Figure 1: Plots for LeNet-5 Training

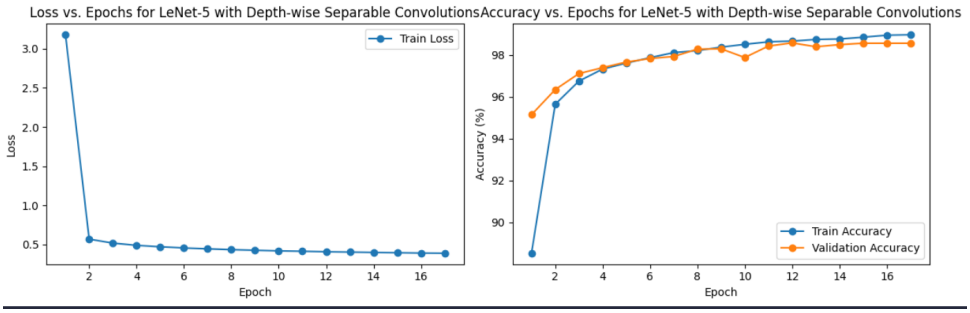

Figure 2: Plots for Depthwise Seperable LeNet-5

c) Multiply and Addition Operations

Caclculation procedure:

- Each weight is multiplied by its corresponding input pixel.
- The resulting products are summed (using $N-1$ additions for N products) and then the bias is added (one additional addition).
- Hence, if a filter has N weights, then per output pixel there are:

$$\text{Multiplications} = N, \quad \text{Additions} = (N-1)+1 = N.$$

Regular LeNet-5

Conv2d-1:

- **Input:** $1 \times 32 \times 32$
- **Kernel:** 6 filters of size 5×5. Each filter has

$$1 \times 5 \times 5 = 25 \text{ weights} \quad \text{and} \quad 1 \text{ bias},$$

  for a total of $25 + 1 = 26$ parameters per filter.
- **Output:** $6 \times 28 \times 28$ (since $32 - 5 + 1 = 28$).

Thus, for each output pixel:

$$\text{Multiplications per pixel} = 25,$$
$$\text{Additions per pixel} = 25.$$

The total number of output pixels is:

$$6 \times 28 \times 28 = 4704.$$

So, the total operations for Conv2d-1 are:

$$\text{Multiplications} = 4704 \times 25 = 117\,600,$$
$$\text{Additions} = 4704 \times 25 = 117\,600.$$

Conv2d-3:

- **Input:** from the pooling layer, size $6 \times 14 \times 14$.
- **Kernel:** 16 filters of size 5×5 applied over 6 channels. Each filter has

  $$6 \times 5 \times 5 = 150 \text{ weights} \quad \text{and} \quad 1 \text{ bias},$$

  for a total of $150 + 1 = 151$ parameters per filter.
- **Output:** $16 \times 10 \times 10$ (since $14 - 5 + 1 = 10$).

Thus, for each output pixel:

$$\text{Multiplications per pixel} = 150,$$
$$\text{Additions per pixel} = 150.$$

The total number of output pixels is:

$$16 \times 10 \times 10 = 1600.$$

So, the total operations for Conv2d-3 are:

$$\text{Multiplications} = 1600 \times 150 = 240\,000,$$
$$\text{Additions} = 1600 \times 150 = 240\,000.$$

Total (Regular LeNet-5 Conv Layers):

$$\text{Total Multiplications} = 117\,600 + 240\,000 = 357\,600,$$
$$\text{Total Additions} = 117\,600 + 240\,000 = 357\,600.$$

Depthwise Separable LeNet-5

In the depthwise separable version, each standard convolution is replaced by a depthwise convolution followed by a pointwise convolution.

Conv1 Block: (a) Depthwise Convolution:

- **Input:** $1 \times 32 \times 32$
- **Kernel:** A single 5×5 filter (since input channels = 1). Parameters: $5 \times 5 = 25$ weights and 1 bias $\Rightarrow 25 + 1 = 26$ parameters.
- **Output:** $1 \times 28 \times 28$.

Per output pixel:

$$\text{Multiplications} = 25,$$
$$\text{Additions} = 25.$$

Total output pixels: $28 \times 28 = 784$. Thus:

$$\text{Multiplications (depthwise)} = 784 \times 25 = 19\,600,$$
$$\text{Additions (depthwise)} = 784 \times 25 = 19\,600.$$

(b) Pointwise Convolution:

- **Input:** $1 \times 28 \times 28$
- **Kernel:** 1×1 convolution with 6 filters. Each filter has 1 weight and 1 bias $\Rightarrow 1 + 1 = 2$ parameters.
- **Output:** $6 \times 28 \times 28$.

Per output pixel:

$$\text{Multiplications} = 1,$$
$$\text{Additions} = 1.$$

Total output pixels: $6 \times 28 \times 28 = 4704$. Thus:

$$\text{Multiplications (pointwise)} = 4704 \times 1 = 4\,704,$$
$$\text{Additions (pointwise)} = 4704 \times 1 = 4\,704.$$

Total for Conv1 Block:

$$\text{Multiplications} = 19\,600 + 4\,704 = 24\,304,$$
$$\text{Additions} = 19\,600 + 4\,704 = 24\,304.$$

Conv2 Block: (a) Depthwise Convolution:

- **Input:** $6 \times 14 \times 14$
- **Kernel:** For each of the 6 channels, a 5×5 filter. Each channel: $5 \times 5 = 25$ weights and 1 bias $\Rightarrow 25 + 1 = 26$ parameters.
- **Output:** $6 \times 10 \times 10$ (since $14 - 5 + 1 = 10$).

Per output pixel (per channel):

$$\text{Multiplications} = 25,$$
$$\text{Additions} = 25.$$

Total output pixels: $6 \times 10 \times 10 = 600$. Thus:

$$\text{Multiplications (depthwise)} = 600 \times 25 = 15\,000,$$
$$\text{Additions (depthwise)} = 600 \times 25 = 15\,000.$$

(b) Pointwise Convolution:

- **Input:** $6 \times 10 \times 10$
- **Kernel:** 1×1 convolution with 16 filters. Each filter has 6 weights and 1 bias $\Rightarrow 6 + 1 = 7$ parameters.
- **Output:** $16 \times 10 \times 10$.

Per output pixel:

$$\text{Multiplications} = 6,$$
$$\text{Additions} = 6.$$

Total output pixels: $16 \times 10 \times 10 = 1600$. Thus:

$$\text{Multiplications (pointwise)} = 1600 \times 6 = 9\,600,$$
$$\text{Additions (pointwise)} = 1600 \times 6 = 9\,600.$$

Total for Conv2 Block:

$$\text{Multiplications} = 15\,000 + 9\,600 = 24\,600,$$
$$\text{Additions} = 15\,000 + 9\,600 = 24\,600.$$

Overall Total (Depthwise Separable Conv Layers):

$$\text{Total Multiplications} = 24\,304 + 24\,600 = 48\,904,$$
$$\text{Total Additions} = 24\,304 + 24\,600 = 48\,904.$$

Final Answer:

Regular LeNet-5 (Conv Layers):
Multiplications = 357 600
Additions = 357 600

Depthwise Separable LeNet-5 (Conv Layers):
Multiplications = 48 904
Additions = 48 904

d) Mathematical Function Derivation

Assume a regular convolution layer has the following parameters:

- Input channels: C_{in}
- Output channels: C_{out}
- Kernel size: $K \times K$
- Output spatial size: $H_{\text{out}} \times W_{\text{out}}$

For each output pixel in a regular convolution, the number of multiplications is

$$N_{\text{reg}} = C_{\text{in}} K^2,$$

and, including the bias (one addition per output pixel), we count the additions similarly (i.e., roughly $C_{\text{in}} K^2$ operations). Thus, the total number of operations (ignoring constant overhead) is

$$x = H_{\text{out}} W_{\text{out}} \cdot C_{\text{out}} \cdot \left(C_{\text{in}} K^2\right).$$

In a **depthwise separable convolution**, the computation is split into two parts:

1. **Depthwise convolution:** Each input channel is convolved separately with a $K \times K$ filter. The cost per output pixel is

   $$N_{\text{dw}} = K^2.$$

   The total cost is

   $$H_{\text{out}} W_{\text{out}} \cdot C_{\text{in}} \cdot K^2.$$

2. **Pointwise convolution:** A 1×1 convolution that maps C_{in} channels to C_{out} channels. The cost per output pixel is

   $$N_{\text{pw}} = C_{\text{in}},$$

   and the total cost is

   $$H_{\text{out}} W_{\text{out}} \cdot \left(C_{\text{in}} \cdot C_{\text{out}}\right).$$

Thus, the total number of operations for the depthwise separable convolution is

$$y = H_{\text{out}} W_{\text{out}}\, C_{\text{in}} \left(K^2 + C_{\text{out}}\right).$$

Since the regular convolution has

$$x = H_{\text{out}} W_{\text{out}}\, C_{\text{in}}\, C_{\text{out}}\, K^2,$$

we can express y in terms of x as follows:

$$\frac{y}{x} = \frac{H_{\text{out}} W_{\text{out}}\, C_{\text{in}}\, (K^2 + C_{\text{out}})}{H_{\text{out}} W_{\text{out}}\, C_{\text{in}}\, C_{\text{out}}\, K^2} = \frac{K^2 + C_{\text{out}}}{C_{\text{out}}\, K^2}.$$

Thus, the mapping function is

$$\boxed{f(x, K, C_{\text{out}}) = \frac{x\,(K^2 + C_{\text{out}})}{C_{\text{out}}\, K^2}.}$$

Verification with LeNet-5

Example 1: Conv2d-1 (Regular Convolution)

For Conv2d-1 of regular LeNet-5:

- $C_{\text{in}} = 1$, $K = 5$, $C_{\text{out}} = 6$, and $H_{\text{out}}W_{\text{out}} = 28 \times 28 = 784$.
- The total operations are

$$x = 784 \cdot (1 \cdot 6 \cdot 25) = 784 \cdot 150 = 117\,600.$$

Using our mapping function:

$$y = f(x, 5, 6) = 117\,600 \cdot \frac{25 + 6}{6 \times 25} = 117\,600 \cdot \frac{31}{150}.$$

Since

$$\frac{117\,600}{150} = 784, \quad \text{we have} \quad y = 784 \times 31 = 24\,304.$$

This matches the manual calculation for the depthwise separable Conv1 block:

$$\text{Depthwise (Conv1)} : 19\,600 \quad \text{and} \quad \text{Pointwise (Conv1)} : 4\,704, \quad \text{total} = 19\,600{+}4\,704 = 24\,304.$$

Example 2: Conv2-3 (Regular Convolution)

For Conv2d-3 of regular LeNet-5:

- $C_{\text{in}} = 6$, $K = 5$, $C_{\text{out}} = 16$, and $H_{\text{out}}W_{\text{out}} = 10 \times 10 = 100$.
- The total operations are

$$x = 100 \cdot (6 \cdot 16 \cdot 25) = 100 \cdot 2400 = 240\,000.$$

Using our mapping function:

$$y = f(x, 5, 16) = 240\,000 \cdot \frac{25 + 16}{16 \times 25} = 240\,000 \cdot \frac{41}{400}.$$

Since

$$\frac{240\,000}{400} = 600, \quad \text{we get} \quad y = 600 \times 41 = 24\,600.$$

This agrees with the manual calculation for the depthwise separable Conv2 block:

$$\text{Depthwise (Conv2)} : 15\,000 \quad \text{and} \quad \text{Pointwise (Conv2)} : 9\,600, \quad \text{total} = 15\,000{+}9\,600 = 24\,600.$$

Final Answer:

$$\boxed{f(x, K, C_{\text{out}}) = \frac{x\,(K^2 + C_{\text{out}})}{C_{\text{out}}\,K^2}}$$

which correctly maps the number of operations x in a regular convolution layer to the number of operations y in a depthwise separable convolution layer producing the same output size.

Problem 3

a) Transformer Implementation

Assumptions for the Modified Transformer Implementation

I have made a few assumptions while implementing the modified Transformer model, I have listed them below:

1. **Random Embeddings for Words:** Instead of using a learned embedding layer or a tokenizer, I have directly split the input sentence by whitespace and assign each word a random 256-dimensional vector. This means we do not have a real vocabulary or trained embeddings; each unique word simply maps to a fresh random vector.

2. **Sinusoidal Positional Encoding:** We add sinusoidal positional encodings to each word embedding. The formula used is the standard one from the original Transformer paper, but it is applied on top of the random embeddings.

3. **16-bit Floating-Point Emulation:** All values (embeddings, weights, intermediate activations) are stored as Python `floats` but rounded to three decimal places to emulate half precision (16-bit). This rounding is done at every step (e.g., after each matrix multiplication or addition). I do *not* actually store values in IEEE 754 `float16` format; we only approximate the precision loss.

4. **Randomly Initialized Weights:** All weight matrices (e.g., W_q, W_k, W_v, W_o, and the final linear layers) are randomly generated at run time. Because there is no training or fine-tuning, these weights do not adapt to the data; they remain random throughout.

5. **Single Encoder and Single Decoder Block:** As per the modified architecture, I have only implemented:

   - One encoder block, which includes Multi-Head Self-Attention followed by Add & Norm.
   - One decoder block, which includes:
     (a) Masked Multi-Head Self-Attention (to attend to previous tokens in the output),
     (b) Add & Norm,
     (c) Cross-Attention (where the decoder attends to the encoder outputs),
     (d) Another Add & Norm.
   - No feed-forward sublayers are used (they are "skipped").

6. **Simple Forward Pass Only (No Training Loop):** We perform a single forward pass that takes in an input sentence and a target (output) sentence. The decoder sees the entire output sentence at once, with a causal (look-back) mask so each position can only attend to previous positions. In real training, you would compare the output probabilities to a ground-truth target and backpropagate; here, we simply compute the probabilities.

7. **Output Probabilities (Softmax Over 10 Classes):** At the end of the decoder, we apply a final linear projection to a small vocabulary size of 10, then apply `softmax` to get probabilities. In a real setting, this projection would be to the entire vocabulary and would be trained. Here, we again use random weights and do not train them.

8. **Interpretation of the Output:** Each time step in the decoder output corresponds to one token position in the target sentence. The softmax gives a probability distribution across 10 arbitrary classes. Because all parameters are random, the model often produces nearly uniform or skewed distributions that do not reflect meaningful linguistic predictions.

9. **Fixed Random Seed (Optional):** In the `main()` function, we set `random.seed(0)` to ensure that the random embeddings and weights are the same every time the code runs. Removing or changing this seed would produce different random outputs on each run.

10. **No Batches included** In the original transformer architecture the input is of the format seq length × embedding size × Batch size. I have not included the batches here hence input will only be a 2-d array.

I have only used the random and math libraries for the implementation and for 16-bit floating-point arithmetic I have rounded of the number to 3 decimal places and have been using it for every operation.

Example output:

```
INPUT:
input_sentence = "hello how are you"
output_sentence = "i am fine sure"

OUTPUT:
Final output probabilities per decoder time-step (for a 10-word 'vocab'):
Time step 0: [0.0, 0.0, 0.0, 0.0, 0.863, 0.0, 0.0, 0.001, 0.136, 0.0]
Time step 1: [0.0, 0.0, 0.0, 0.0, 0.862, 0.0, 0.0, 0.001, 0.137, 0.0]
Time step 2: [0.0, 0.0, 0.0, 0.0, 0.86, 0.0, 0.0, 0.001, 0.139, 0.0]
Time step 3: [0.0, 0.0, 0.0, 0.0, 0.859, 0.0, 0.0, 0.001, 0.141, 0.0]
```

b) Communication Cost Analysis

In our modified Transformer model, the encoder processes the input sequence and produces an output matrix of dimensions

$$L \times d_{\text{model}},$$

where:

- L is the sequence length (number of tokens in the input), and
- d_{model} is the embedding dimension (256).

For the decoder's cross-attention, only the **key (K)** and **value (V)** matrices are required. These are computed from the encoder output as follows:

$$K = \text{EncOutput} \times W_k \quad \text{and} \quad V = \text{EncOutput} \times W_v,$$

with both K and V having dimensions:

$$L \times d_{\text{model}}.$$

Since each element in these matrices is stored as a 16-bit floating point number, the number of bits needed to store one such matrix is:

$$\text{Bits per matrix} = L \times d_{\text{model}} \times 16.$$

Because both the key and value matrices are communicated from the encoder to the decoder, the total number of bits transferred is:

$$\text{Total Bits} = 2 \times (L \times d_{\text{model}} \times 16) = 32\, L\, d_{\text{model}}.$$

So the linear layer recieves L $\times d_{\text{model}} \times 16$. from the add and norm block after the final multihead head attention block in the decoder block. Now it transforms this into $L \times$ vocab size $\times$ 16 bits before passing it onto the softmax layer.

Scaling with Embedding Size

Assuming that the sequence length L remains fixed, the total communication cost between the encoder and decoder scales linearly with the embedding size d_{model}. That is:

$$\text{Total Bits} \propto d_{\text{model}}.$$

For example, doubling d_{model} will double the number of bits transmitted.

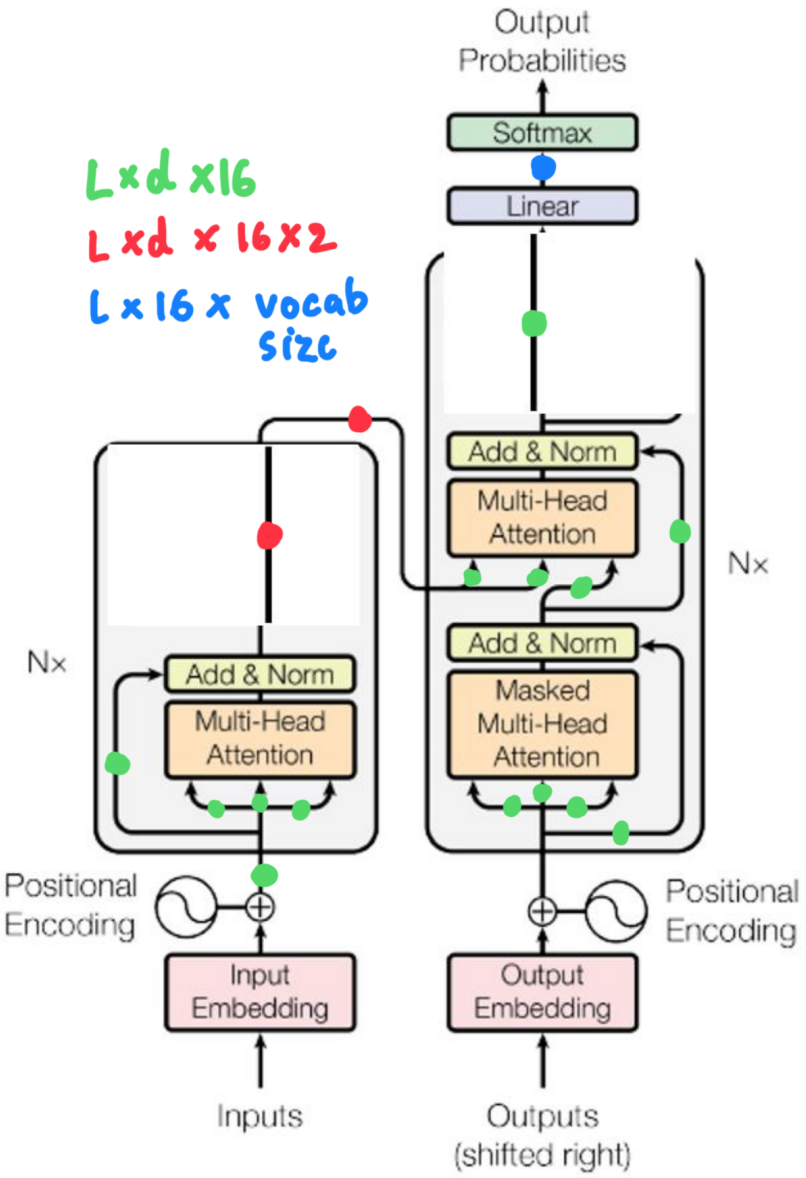

Figure 3: Number of bits flow between each block