Xi'an Jiaotong-Liverpool University

西交利物浦大學

INT305 Assessment 2

Ruoyu Xu (2142804)

December 11, 2024

Contents

1 Q1: Introduction to CNNs

Convolutional Neural Networks (CNNs), deep learning models specialized in processing grid-like data such as images, perform exceptionally well on computer vision tasks such as image classification and object detection.

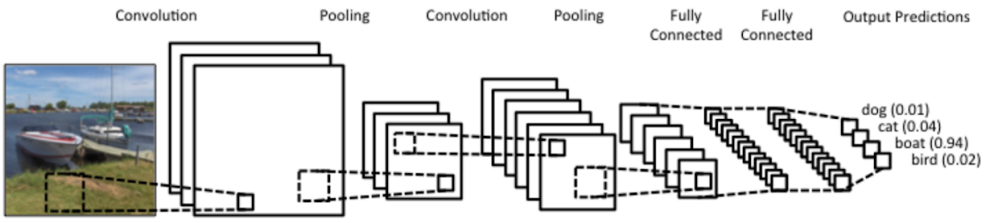

Figure 1: Process of CNN

Simply put, CNNs extract local features through convolution. Pooling reduces the dimensionality of the data while retaining key information. This process is repeated to extract more abstract and comprehensive features. Finally, the classification or prediction results are produced after fully connected layers process the extracted features [3].

1.1 Convolution Kernel

The convolution kernel is a core concept in convolution operations and even CNNs. It is a matrix of weights designed to extract different features.

The convolution operation can be written as:

$$O(i,j) = \sum_{p=0}^{m-1} \sum_{q=0}^{n-1} K(p,q) \cdot I(i+p, j+q)$$

- I represent the input matrix (e.g., image pixels).
- K represent the convolutional kernel (filter).
- O represent the output matrix (feature map).
- m, n be the dimensions of the kernel.
- i, j be the indices of the output feature map.

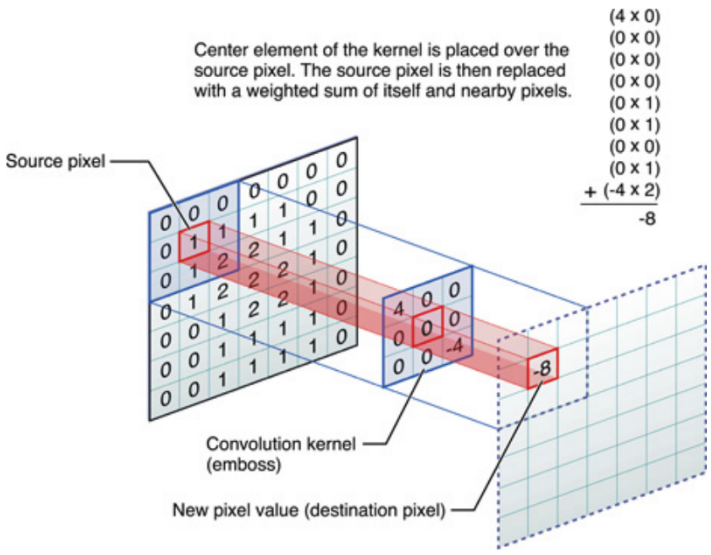

Figure 2: Operation of convolutional kernel

During runtime, the convolution kernel extracts local features from input data by sliding over it and performing element-wise multiplication and summation, producing a feature map [1]. A convolutional layer typically contains 32 or 64 convolutional kernels, each extracting distinct features, resulting in multiple feature maps. The weights of the kernels are learned from the data using backpropagation algorithms and are automatically adapted to suit the task requirements [4].

1.2 Loss Function

The loss function is a crucial component of model training, guiding the optimization algorithm to adjust model parameters by quantifying the difference between model predictions and true labels. The ultimate goal of the model is to minimize the loss function, thereby improving predictive accuracy.

The choice of the loss function influences learning dynamics. A suitable loss function can accelerate learning and penalize specific errors more effectively [6].

1. Regression Loss Functions

   - **Mean Squared Error (MSE) Loss:** One of the most widely used loss functions for regression problems, MSE calculates the average squared differences between predicted and actual values. It works well with gradient-based optimization but is highly sensitive to outliers.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2$$

   - **Mean Absolute Error (MAE) Loss:** MAE is less sensitive to outliers than MSE, making it an alternative for robust regression tasks.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} \mid y_i - \widehat{y}_i \mid$$

     Here, n represents the number of data point, y_i is the actual label, and $\widehat{y}_i$ is the predicted label.

2. Classification Loss Functions

   - **Binary Cross-Entropy Loss:** Suitable for binary classification tasks, this function outputs predicted probabilities but can be sensitive to imbalanced datasets.

$$\text{Binary Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

     n represents the number of data point, y_i is the actual label(0 or 1), and $\widehat{y}_i$ is the predicted label.

   - **Categorical Cross-Entropy Loss:** Designed for multi-class classification tasks, it measures the performance of a model outputting probability distributions across multiple classes.

$$\text{Categorical Cross-Entropy Loss} = -\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(\hat{y}_{ij})$$

     n represents the number of data point, k is the number of classes, y_{ij} is the binary indicator if the class label j is the correct classification for data point i, and $\widehat{y}_{ij}$ is the predicted possibility for class j.

2 Q2: Constructing CNN for MNIST

To construct a basic CNN model for training on the MNIST dataset, we first performed data preprocessing, transforming the data into tensors and normalizing it. A simple CNN architecture

was then defined, starting with a convolutional layer to extract image features, followed by a pooling layer to down sample the feature map and reduce its dimensionality. The extracted features were subsequently mapped to 10 output classes through two fully connected layers.

For training, **CrossEntropyLoss** was selected as the loss function to measure the discrepancy between model predictions and true labels. **Stochastic Gradient Descent (SGD)** was employed as the optimizer, with a learning rate of 0.01 to adjust model parameters and enhance classification performance.

Table 1: Training and Testing Results for Basic CNN model

Epoch	Loss	Train Accuracy	Test Accuracy
1	0.6602	82.69%	91.58%
2	0.2304	93.30%	95.16%
3	0.1610	95.30%	94.91%

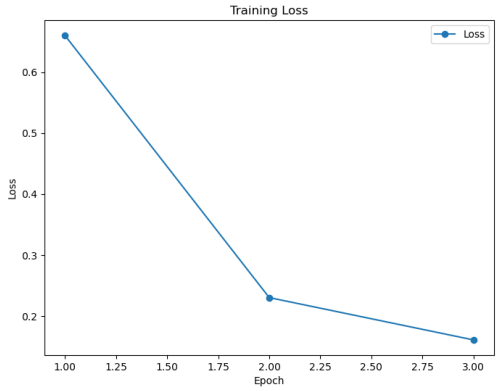

Figure 3: Loss function for CNN basic model

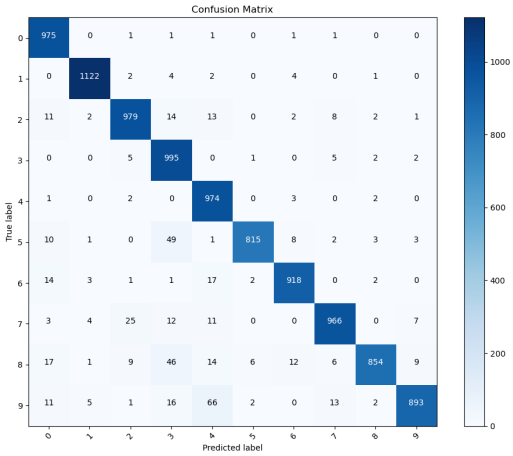

Figure 4: Confusion matrix for CNN basic model

This model achieved rapid convergence within three training cycles. The training loss decreased from 0.6602 to 0.1610, and both training and test accuracies improved significantly. Despite achieving a high test accuracy of 94.91%, the model exhibited certain limitations. Firstly, the lack of data augmentation meant only raw data was utilized for training, potentially compromising the model's robustness. Secondly, the model's architecture was simplistic, consisting of only one convolutional layer and one pooling layer, which limited its feature extraction capacity. Additionally, the basic optimizer and the small number of hidden units in the fully connected layers restricted the model's learning capacity.

To address these drawbacks, the following optimizations were implemented:

1. **Data Augmentation**: Techniques such as random flipping and rotation were applied to augment data with an ultimate goal to enhance the model's robustness.
2. **Model Enhancement**:Two convolutional layers were introduced to extract both low-level and high-level features, with pooling layers following each convolutional layer to reduce feature map size and mitigate overfitting.

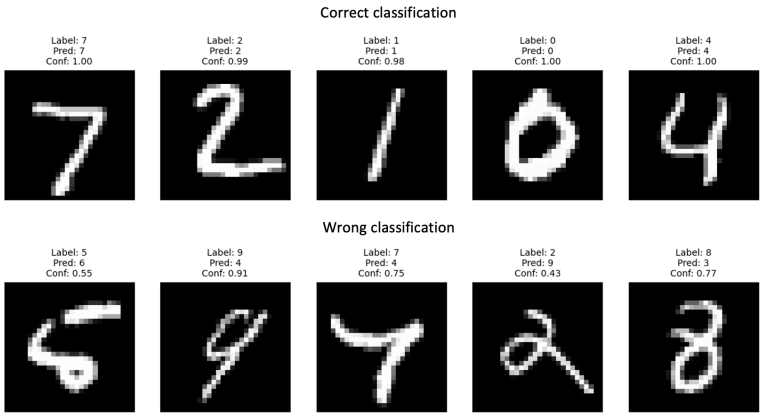

Figure 5: Samples of correct classification(CNN Basic)

3. **Optimizer Improvement**:The Adam optimizer was used to accelerate convergence by automatically adjusting the learning rate for each parameter, leveraging momentum and gradient squares. Additionally, the learning rate was reduced to 0.001 to enable fine-grained parameter adjustments, promoting stable convergence.

Table 2: Training and Testing Results for Optimized CNN model

Epoch	Loss	Train Accuracy	Test Accuracy
1	0.1593	95.12%	98.46%
2	0.0449	98.60%	98.65%
3	0.0311	99.05%	98.81%
4	0.0216	99.34%	98.55%
5	0.0167	99.47%	98.95%

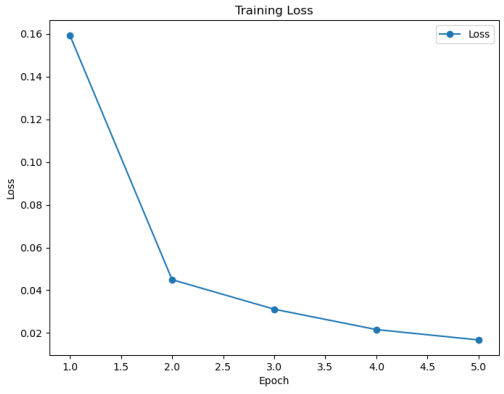

Figure 6: Loss function for CNN optimized model

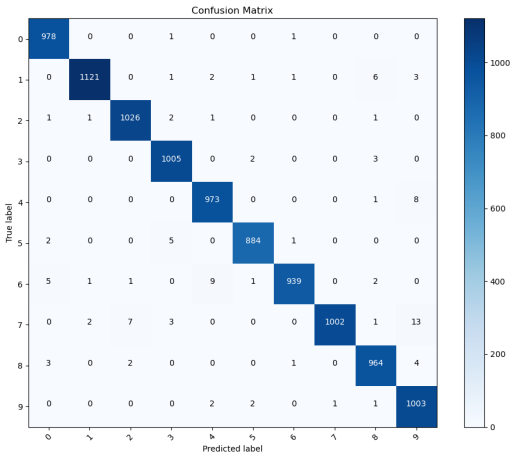

Figure 7: Confusion matrix for CNN optimized model

The improved model demonstrated superior performance, achieving a final test accuracy of 98.95%. The training loss decreased consistently from 0.1593 to 0.0167 in the final epoch, indicating effective learning and convergence. Both training and test accuracies remained closely

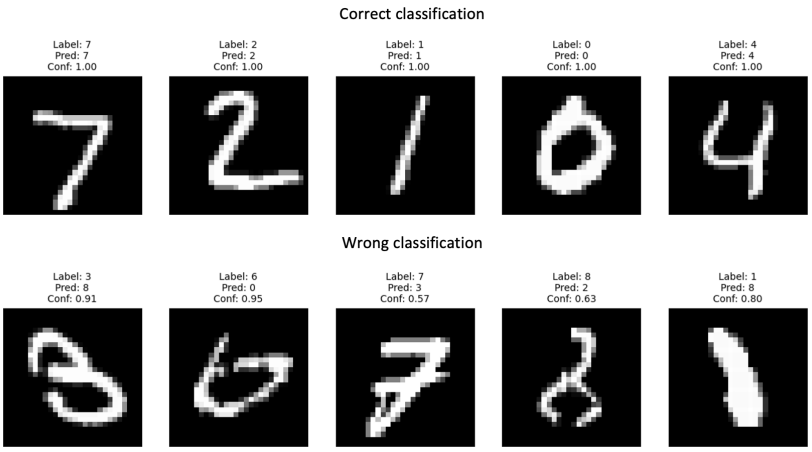

Figure 8: Samples of correct classification(CNN Optimized)

aligned throughout the training process, confirming the model's robustness. These results underscore the effectiveness of the optimized CNN for the MNIST classification task.

3 Q3: Model comparision

Although traditional CNNs perform well on computer vision tasks, they do have notable limitations. To address these, various CNN variants have been developed, including MobileNet, which focuses on lightweight efficiency.

3.1 MobileNet

MobileNet employs a unique technique: depthwise separable convolution [2]. This technique splits the traditional convolution operation into two steps:

1. **Depthwise Convolution** Convolution is applied independently to each input channel, extracting spatial features within each channel. This step does not alter inter-channel relationships therefore reducing computational complexity.
2. **Pointwise Convolution** A 1*1 convolution kernel is used to fuse features across channels, capturing inter-channel relationships and producing a new channel representation.

By separating these two steps, MobileNet achieves feature extraction with significantly reduced computational cost compared to standard convolutions. Each layer in MobileNet consists of paired depthwise and pointwise convolutions, followed by a global pooling layer and a fully connected layer for classification.

For the MNIST dataset, MobileNet achieved a test accuracy of 98.88%, comparable to CNN's 98.95%. MobileNet displayed slightly lower loss values during training, indicating better optimization. However, the computational efficiency of MobileNet is not evident in this

Table 3: Training and Testing Results for MobileNet

Epoch	Loss	Train Accuracy	Test Accuracy
1	0.1980	94.89%	97.84%
2	0.0413	98.77%	98.66%
3	0.0297	99.05%	98.41%
4	0.0241	99.25%	98.96%
5	0.0202	99.38%	98.88%

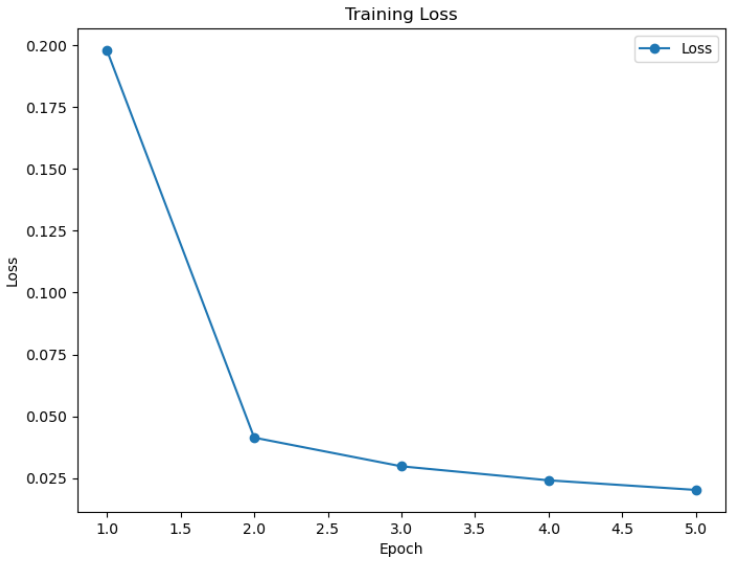

Figure 9: Loss function for MobileNet

simple task involving low-resolution grayscale images. While CNN is more suitable for MNIST due to its faster training time and marginally better performance, MobileNet's architectural innovations are likely to excel in complex tasks or larger datasets [5].

3.2 Vision Transformer(ViT)

In 2020, the Vision Transformer (ViT) introduced a novel approach by combining attention mechanisms with convolutional architectures. ViT segments images into fixed-size patches, flattens them, and maps them into a fixed-dimensional vector space via linear projections [7]. These vectors, along with positional encodings, are fed into a Transformer architecture. Using self-attention mechanisms, ViT captures global image features by computing relationships between patches.

For MNIST, ViT achieved a test accuracy of 98.90%, comparable to CNN and MobileNet. However, it required more training iterations (10 epochs) to converge and lacked the spatial inductive bias of CNNs, resulting in weaker initial performance on small datasets. ViT is more suitable for high-dimensional, complex datasets where its global feature extraction capabilities can be fully leveraged.

Table 4: Training and Testing Results for ViT

Epoch	Loss	Train Accuracy	Test Accuracy
1	0.6988	76.61%	95.02%
2	0.2802	91.17%	96.82%
3	0.2161	93.16%	97.65%
4	0.1346	95.70%	98.60%
5	0.1121	96.52%	98.77%
6	0.1058	96.67%	98.74%
7	0.1014	96.91%	98.86%
8	0.0993	96.83%	98.91%
9	0.0961	97.04%	98.92%
10	0.0959	96.95%	98.90%

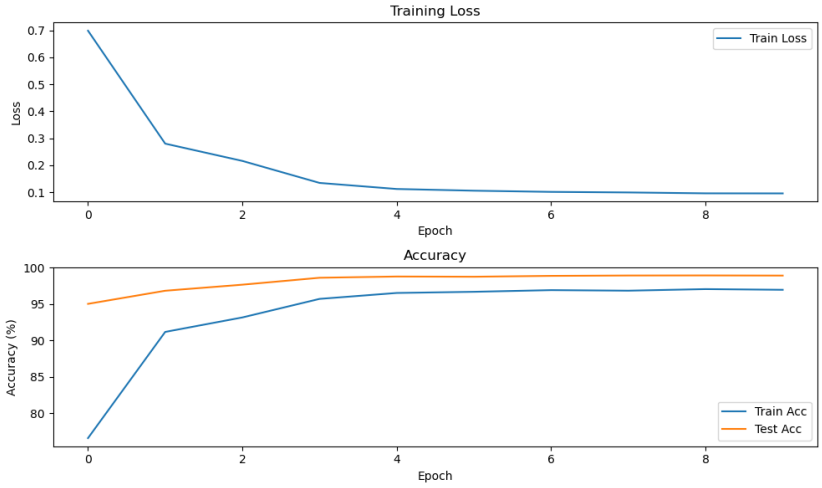

Figure 10: Loss function and accuracy for ViT

References

[1] David Haussler et al. Convolution kernels on discrete structures. Technical report, Citeseer, 1999.

[2] Andrew G. Howard et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv*, Apr. 2017. Cornell University.

[3] Tim Kattenborn, Jens Leitloff, Frank Schiefer, and Stefan Hinz. Review on convolutional neural networks (cnn) in vegetation remote sensing. *ISPRS Journal of Photogrammetry and Remote Sensing*, 173:24–49, Mar. 2021.

[4] David W Romero, Anna Kuzina, Erik J Bekkers, Jakub M Tomczak, and Mark Hoogendoorn. Ckconv: Continuous kernel convolution for sequential data. *arXiv preprint arXiv:2102.02611*, 2021.

[5] Anumol C S. Advancements in cnn architectures for computer vision: A comprehensive review. In *2023 Annual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS)*, pages 1–7, 2023.

[6] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, pages 1–26, 2020.

[7] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zi-Hang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. Tokens-to-token vit: Training vision transformers from scratch on imagenet. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 558–567, 2021.

Appendix

Code for model construction

Optimized CNN

```

# 2. Define Optimized CNN Model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleCNN().to(device)

# 3. Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

MobileNet

```

# Define Depth-wise separable convolution
class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(DepthwiseSeparableConv, self).__init__()
        self.depthwise = nn.Conv2d(in_channels, in_channels, kernel_size=3, stride=stride, padding=1, groups=in_channels)
        self.pointwise = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        x = self.depthwise(x)
        x = self.pointwise(x)
        return x

# Define MobileNet Model
class MobileNet(nn.Module):
    def __init__(self, num_classes=10):
        super(MobileNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),  #
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),

            DepthwiseSeparableConv(32, 64),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),

            DepthwiseSeparableConv(64, 128, stride=2),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),

            DepthwiseSeparableConv(128, 128),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),

            DepthwiseSeparableConv(128, 256, stride=2),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),

            DepthwiseSeparableConv(256, 256),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),

            nn.AdaptiveAvgPool2d(1)  #
```

```
        )
        self.classifier = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  #
        x = self.classifier(x)
        return x
model = MobileNet().to(device)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

ViT

```

# Define transformer
class TransformerEncoderBlock(nn.Module):
    def __init__(self, dim, num_heads, mlp_ratio=4.0, dropout=0.1):
        super(TransformerEncoderBlock, self).__init__()
        self.norm1 = nn.LayerNorm(dim)
        self.attn = nn.MultiheadAttention(dim, num_heads, dropout=dropout,
        ↪  batch_first=True)
        self.norm2 = nn.LayerNorm(dim)

        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = nn.Sequential(
            nn.Linear(dim, mlp_hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(mlp_hidden_dim, dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        # Self-attention
        x = x + self.attn(self.norm1(x), self.norm1(x), self.norm1(x))[0]
        # MLP
        x = x + self.mlp(self.norm2(x))
        return x

# Define the ViT Model
class VisionTransformer(nn.Module):
```

```
    def __init__(self, image_size=28, patch_size=7, dim=128, depth=4,
    ↪  num_heads=4, mlp_ratio=4.0, num_classes=10, dropout=0.1):
        super(VisionTransformer, self).__init__()
        assert image_size % patch_size == 0, "image_size must be divisible
        ↪  by patch_size"
        self.num_patches = (image_size // patch_size) ** 2
        self.patch_size = patch_size
        patch_dim = patch_size * patch_size  # MNIST1
        self.dim = dim

        # patchdim
        self.patch_embed = nn.Linear(patch_dim, dim)

        # CLS token
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        #
        self.pos_embed = nn.Parameter(torch.randn(1, self.num_patches + 1,
        ↪  dim))
        self.dropout = nn.Dropout(dropout)

        # Transformer Encoder Block
        self.blocks = nn.ModuleList([
            TransformerEncoderBlock(dim, num_heads, mlp_ratio,
            ↪  dropout=dropout) for _ in range(depth)
        ])

        self.norm = nn.LayerNorm(dim)
        self.head = nn.Linear(dim, num_classes)

    def forward(self, x):
        # x: [B,1,28,28]
        B, C, H, W = x.shape
        # patch
        patches = x.unfold(2, self.patch_size, self.patch_size).unfold(3,
        ↪  self.patch_size, self.patch_size)
        # patches: [B, 1, H/ps, W/ps, ps, ps]
        patches = patches.contiguous().view(B, -1,
        ↪  self.patch_size*self.patch_size)
        x = self.patch_embed(patches)  # [B, num_patches, dim]

        # CLS token
        cls_tokens = self.cls_token.expand(B, -1, -1) # [B,1,dim]
        x = torch.cat((cls_tokens, x), dim=1) # [B, num_patches+1, dim]

        #
        x = x + self.pos_embed[:, :x.size(1), :]
        x = self.dropout(x)

```

```
        # Transformer Encoder
        for blk in self.blocks:
            x = blk(x)

        x = self.norm(x)
        # CLS token
        cls_out = x[:,0]
        logits = self.head(cls_out)
        return logits

net = VisionTransformer().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(net.parameters(), lr=0.001, weight_decay=1e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)
```