Coursework 2 for INT305: MNIST Image classification

Jiaming Huang[1]

[1]Xi'an Jiaotong-Liverpool University
111 Ren'ai Road - 215123 - Suzhou - Jiangsu Province - China

Jiaming.Huang21@student.xjtlu.edu.cn

Abstract. *This report addresses three problems outlined in Coursework 2. The first question involves explaining concepts related to CNN. The second problem focuses on implementing a CNN-based model for image analysis. The third problem searched for approaches to improve model accuracy, where two strategies are explored: hyperparameter tuning and SOTA models examination. Finally, an ensemble model is built to further enhance accuracy and improve the robustness of the predictor. In the code section, key code snippets related to Problems 2 and 3 are provided, while the full code is also available on GitHub via the attached link.*

1 Problem 1

Problem 1: Please describe the 2 key components in the CNN framework: the convolutional kernel and the loss functions used in the framework. (20%)

Answer: **Convolutional Kernel:** The convolutional kernel, or filter, is a small trainable matrix, such as 3x3 or 5x5, that extracts features from input data through the convolution operation, sliding over the input and computing dot products with overlapping regions. It captures low-level features like edges and textures in the early layers and high-level patterns in deeper layers. The formula for calculating the size of the output is as follows:

$$O = \frac{(W - K + 2P)}{S} + 1 \tag{1}$$

where:

- O: the output size.
- W: the input width.
- K: the kernel size.
- P: the padding size.
- S: the stride.

Loss Functions: The loss function quantifies the discrepancy between the predicted output and the ground truth, guiding the optimization process by calculating gradients during backpropagation. Common loss functions include cross-entropy loss for classification tasks, mean squared error for regression tasks.

Cross-entropy loss is a loss function used in classification tasks to measure the difference between the predicted probability distribution and the true distribution. It encourages the model to assign high probabilities to the correct classes while penalizing incorrect predictions. By minimizing cross-entropy loss, the model learns to make predictions that are more confident and accurate, aligning the predicted probabilities with the true labels. The formula for cross-entropy loss is as follows:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}) \tag{2}$$

where:

- N is the number of samples in the dataset.
- C is the total number of classes.
- $y_{i,c}$ is the ground truth indicator, which is 1 if sample i belongs to class c, and 0 otherwise.
- $\hat{y}_{i,c}$ is the predicted probability for sample i belonging to class c, obtained from a softmax function.
- $\mathcal{L}$ is the cross-entropy loss, commonly used for multi-class classification tasks.

Mean Squared Error (MSE) loss is a commonly used metric in regression tasks that quantifies the average squared difference between predicted values and actual target values. It evaluates how well a model's predictions align with the true values by penalizing larger errors more heavily due to the squaring of differences. This makes MSE particularly sensitive to outliers, as large errors contribute disproportionately to the overall loss. By minimizing MSE, a model is trained to produce predictions that are as close as possible to the actual values, improving its accuracy and performance. The formula for mean squared error loss is as follows:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{3}$$

where:

- $\mathcal{L}_{\text{MSE}}$: the mean squared error loss.
- N: the number of samples.
- y_i: the true value for the i-th sample.
- $\hat{y}_i$: the predicted value for the i-th sample.

2 Problem 2

Problem 2: Please train (or fine-tune) and test the framework on MNIST and report the final accuracy performance that you have achieved. Please also report some well classified and misclassified images by including the images and corresponding classification confidence value. (40%).

Answer: To address this problem, the provided CNN architecture is trained and tested on the

MNIST dataset. Model performance is evaluated using six metrics: accuracy (ACC), calculated as the ratio of correctly classified samples to the total number of samples; sensitivity (Sn); specificity (Sp); F1 score (F1); Matthews correlation coefficient (MCC); and area under the curve (AUC). These metrics are computed using the "macro" method, which averages the metrics across all classes equally, regardless of class size. The results, summarized in **Table 1**, indicate a final training accuracy of 98.97% and a testing accuracy of 98.69%.

Table 1: Model Performance for Provided CNN

	ACC(%)	Sn(%)	Sp(%)	F1	MCC	AUC
Train	98.97	98.95	99.89	0.9896	0.9885	0.9998
Test	98.69	98.68	99.85	0.9868	0.9854	0.9997

To provide a case study, two correctly classified and two misclassified images from the MNIST test set are presented alongside their respective classification probabilities. For correctly classified examples, an image of "8" and an image of "3" are provided, with prediction probabilities of 0.9274 and 0.9747, respectively. For misclassified examples, an image of "6" is incorrectly classified as "0" with a prediction probability of 0.7432, and an image of "9" is misclassified as "4" with a prediction probability of 0.6084. Furthermore, CAM-based methods, including Grad-CAM[1], Grad-CAM++[2], Layer-CAM[3], and Score-CAM[4], are applied to enhance visual interpretability by generating heatmap overlays on the original images. The combined results of both predictions and CAM visualizations are shown in **Figure 1**.

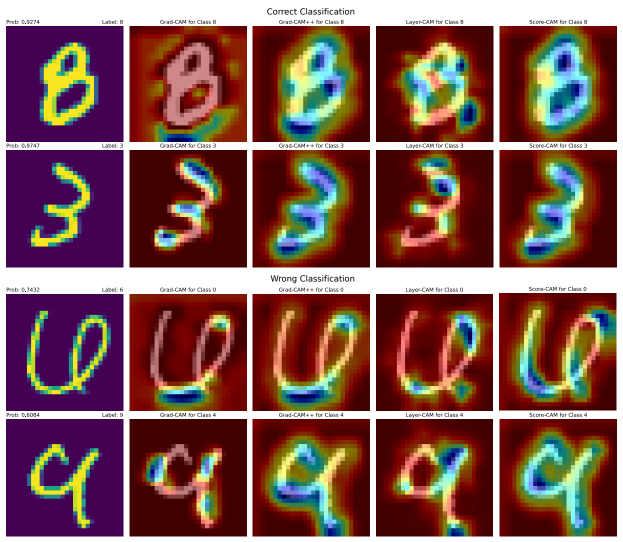

Figure 1: Examples of correctly classified and misclassified images with their prediction probabilities and CAM-based interpretations

Figure 1 shows that for correctly classified images, the CAM analysis heatmaps indicate that the model effectively captures the peripheral contour information of the digits, which is valuable for classification. Conversely, for misclassified images, the heatmaps reveal the model's focus on non-peripheral contours. Since these contours are often shared among various digits, the model may be misled into making incorrect predictions by associating the image with digits that share these common patterns, rather than recognizing the target digit's unique peripheral features.

3 Problem 3

Problem 3: Propose your own method to further improve the classification performance or reduce the model size. You need also compare different methods with the performance you obtained and explain why. The final classification accuracy is not the most important part, you may better refer to some latest published papers and code these state of the art methods to improve the performance. The explanation and analysis of your adopted method is highly related to your final score. (40%)

Answer: To enhance the accuracy of the provided CNN model, two strategies are explored: tuning the hyperparameters of the CNN and optimizing the training process, as well as conducting a comparative study with state-of-the-art algorithms. The top-performing models from this study are then ensembled to achieve greater accuracy and robustness.

3.1 Strategy 1: Hyperparameter Tuning

Here, three combinations of kernel size, stride, and padding, which are (3,1,1), (5,1,2), and (7,1,3), are explored in both the Conv1 and Conv2 blocks. Additionally, two optimizer choices are investigated: Adam with a learning rate of 0.001 and SGD with a learning rate of 0.001 and momentum of 0.9. These settings lead to numerous combinations for model training. A detailed and clear specification of models is provided in **Table 2**.

Table 2: Model Specification

Conv1 Block	Conv2 Block	Optimizer	Model
(3,1,1)	(3,1,1)	Adam	CNN1
		SGD	CNN2
	(5,1,2)	Adam	CNN3
		SGD	CNN4
	(7,1,3)	Adam	CNN5
		SGD	CNN6
(5,1,2)	(3,1,1)	Adam	CNN7
		SGD	CNN8
	(5,1,2)	Adam	CNN9
		SGD	CNN10
	(7,1,3)	Adam	CNN11
		SGD	CNN12
(7,1,3)	(3,1,1)	Adam	CNN13
		SGD	CNN14
	(5,1,2)	Adam	CNN15
		SGD	CNN16
	(7,1,3)	Adam	CNN17
		SGD	CNN18

The six performance metrics for all 18 CNN models, evaluated on both the training and testing sets, are presented in **Table 3**.

Table 3: Model Performance for 18 CNN Models

Model		ACC(%)	Sn(%)	Sp(%)	F1	MCC	AUC
CNN1	**Train**	98.97	98.95	99.89	0.9896	0.9885	0.9998
	Test	**98.69**	98.68	99.85	0.9868	0.9854	0.9997
CNN2	**Train**	99.63	99.63	99.96	0.9963	0.9959	0.9999
	Test	**99.59**	99.15	99.91	0.9915	0.9907	0.9999
CNN3	**Train**	98.26	98.25	99.81	0.9824	0.9808	0.9997
	Test	97.98	97.95	99.78	0.9794	0.9776	0.9995
CNN4	**Train**	99.59	99.59	99.95	0.9959	0.9955	0.9999
	Test	99.19	99.18	99.91	0.9918	0.9910	0.9999
CNN5	**Train**	99.26	99.26	99.92	0.9926	0.9918	0.9999
	Test	98.82	98.81	99.87	0.9882	0.9869	0.9997
CNN6	**Train**	99.62	99.62	99.96	0.9962	0.9957	0.9999
	Test	99.24	99.23	99.92	0.9923	0.9916	0.9999
CNN7	**Train**	98.32	98.33	99.81	0.9830	0.9814	0.9997
	Test	97.77	97.77	99.75	0.9774	0.9753	0.9993
CNN8	**Train**	99.67	99.67	99.96	0.9967	0.9963	0.9999
	Test	**99.39**	99.39	99.93	0.9939	0.9932	0.9999
CNN9	**Train**	99.22	99.22	99.91	0.9922	0.9913	0.9999
	Test	99.00	99.00	99.89	0.9900	0.9889	0.9998
CNN10	**Train**	99.66	99.66	99.96	0.9966	0.9962	0.9999
	Test	99.10	99.09	99.90	0.9908	0.9900	0.9999
CNN11	**Train**	99.35	99.35	99.93	0.9934	0.9927	0.9999
	Test	99.01	99.00	99.89	0.9900	0.9890	0.9998
CNN12	**Train**	99.69	99.69	0.9997	0.9969	0.9965	0.9999
	Test	**99.40**	99.40	99.93	0.9939	0.9933	0.9999
CNN13	**Train**	98.96	98.95	99.88	0.9895	0.9884	0.9998
	Test	98.63	98.62	99.85	0.9862	0.9848	0.9997
CNN14	**Train**	99.49	99.49	99.94	0.9947	0.9943	0.9999
	Test	99.17	99.17	99.91	0.9915	0.9908	0.9999
CNN15	**Train**	98.72	98.72	99.86	0.9871	0.9858	0.9998
	Test	98.45	98.44	99.83	0.9843	0.9828	0.9996
CNN16	**Train**	99.62	99.62	99.96	0.9962	0.9957	0.9999
	Test	99.17	99.17	99.91	0.9916	0.9908	0.9999
CNN17	**Train**	99.06	99.07	99.90	0.9905	0.9895	0.9999
	Test	98.69	98.69	99.85	0.9868	0.9855	0.9996
CNN18	**Train**	99.78	99.78	99.98	0.9978	0.9975	0.9999
	Test	99.29	99.29	99.92	0.9929	0.9921	0.9999

From **Table 3**, it can be observed that the top three models, which exceed the accuracy of CNN1 (98.68%), are CNN2, CNN8, and CNN12, with accuracies of 99.59%, 99.39%, and 99.40%, respectively. It is important to note that although none of these three CNN models use the (7,1,3) configuration in the Conv1 Block, the models with the (7,1,3) configuration in Conv1 Block all achieve an average accuracy above 98% and demonstrate greater stability. This suggests that a larger receptive field improves the overall accuracy and stability of the model.

To further provide a visual interpretation of the model performance improvement, CAM-based approaches are applied to CNN1, CNN2, CNN8, and CNN12. The results of the four types of CAM interpretation are summarized in **Figure 2**.

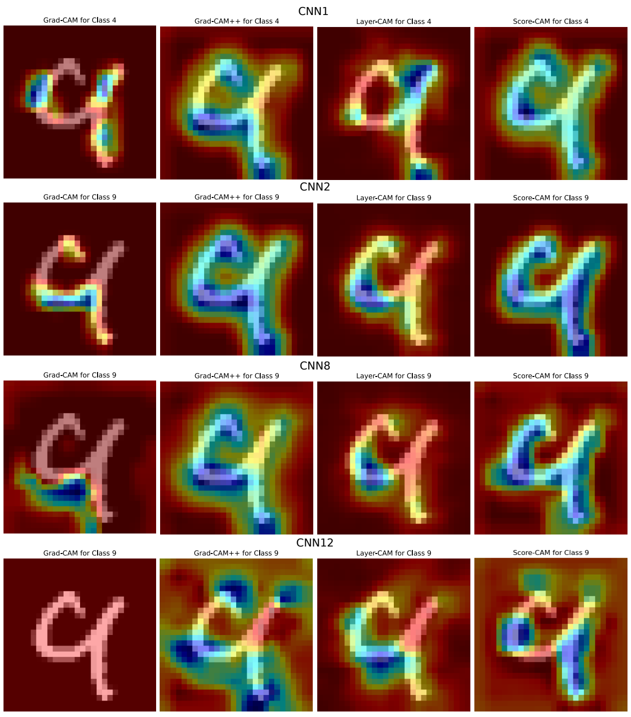

Figure 2: CAM-based interpretation of CNN2, CNN8, and CNN12 on a CNN1-misclassified image

From the CAM results, particularly in the Layer-CAM, we can observe that CNN2, CNN8, and CNN12 consistently focus on the upper right corner of the digit "9," while they largely ignore the lower left corner. In contrast, CNN1 emphasizes the lower left corner pattern of the digit "9" and downplays the upper right corner. The stark difference in focus between the CNNs that make correct judgments and those that misclassify provides further evidence of trustworthiness behind the improved accuracy of CNNs, as well as an explanation for the misclassification of certain models.

3.2 Strategy 2: SOTA Model Examination

In addition to the aforementioned models, a comprehensive evaluation is conducted on nine additional state-of-the-art algorithms, including Residual Network (ResNet)[5], known for its innovative skip connections that facilitate the training of deeper networks; Long Short-Term Memory (LSTM)[6], which excels at capturing long-range dependencies in sequential data; and Vision Transformer (ViT)[7], leveraging self-attention mechanisms for novel image processing. EfficientNet[8] balances model size and accuracy by optimizing depth and width, while MobileNet[9] is recognized for its lightweight architecture, making it suitable for mobile and resource-constrained environments. Furthermore, AlexNet[10] revolutionized image classification with deep learning techniques, GoogLeNet[11] introduced inception modules for multi-scale feature extraction, and ShuffleNet[12] is noted for its efficient computation and performance on mobile devices through unique channel shuffle operations. Together, these models represent a diverse array of methodologies aimed at enhancing classification accuracy. In the following part, the examination of state-of-the-art (SOTA) models is divided into two parts: a comparative study of individual models and the construction of ensemble models. This workflow is also illustrated in **Figure 3**.

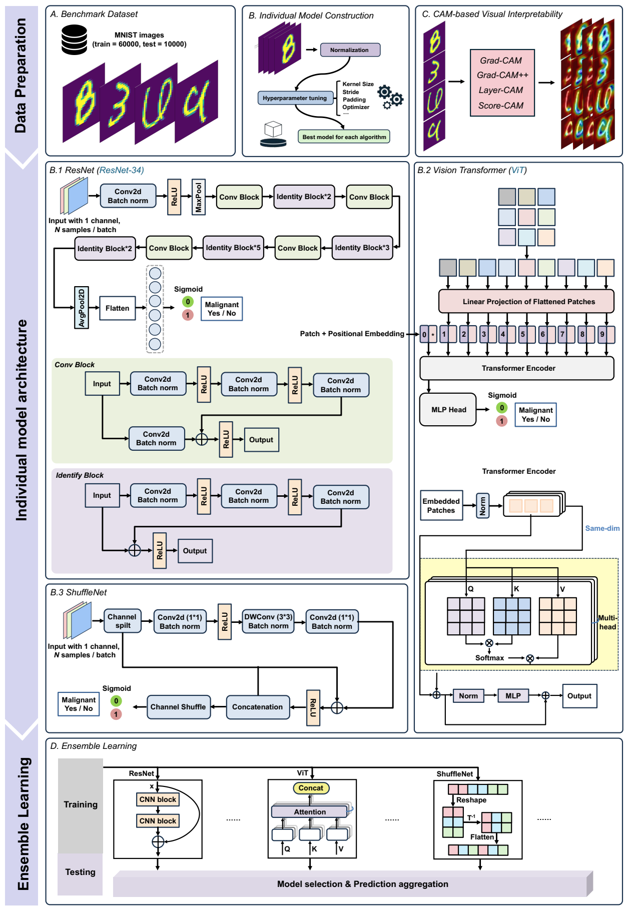

Figure 3: Overall Workflow for SOTA Models Examination

3.2.1 Comparsion Study

The six performance metrics for these models in both the training and testing stages are presented in **Table 4**. From the table, ResNet, Vision Transformer, and ShuffleNet are the top three models, achieving accuracies of 98.82%, 98.86%, and 98.83%, respectively. For ResNet, the impressive accuracy can be attributed to its innovative skip connections, which allow for the training of much deeper networks without encountering issues like vanishing gradients. In the case of ViT, its superior performance is likely due to the self-attention mechanisms it employs, enabling the model to focus on relevant parts of an image while disregarding less important features, thus enhancing its context understanding. Meanwhile, ShuffleNet's notable accuracy may be a result of its unique channel shuffle operation, which improves computational efficiency and allows the model to maintain high performance even with limited resources.

Table 4: Model Performance for Comparison Study

Model		ACC(%)	Sn(%)	Sp(%)	F1	MCC	AUC
CNN	**Train**	98.97	98.95	99.89	0.9896	0.9885	0.9998
	Test	98.69	98.68	99.85	0.9868	0.9854	0.9997
ResNet	**Train**	99.80	99.79	99.98	0.9979	0.9977	0.9993
	Test	**98.82**	98.81	99.87	0.9881	0.9869	0.9991
LSTM	**Train**	94.73	94.70	99.42	0.9468	0.9415	0.9978
	Test	94.49	94.47	99.39	0.9444	0.9389	0.9980
ViT	**Train**	98.91	98.92	99.82	0.9892	0.9818	0.9997
	Test	**98.86**	98.87	99.46	0.9513	0.9887	0.9971
EfficientNet	**Train**	98.74	98.73	99.86	0.9873	0.9860	0.9983
	Test	97.66	97.66	99.74	0.9765	0.9740	0.9978
MobileNet	**Train**	99.29	99.29	99.92	0.9929	0.9921	0.9973
	Test	97.08	97.06	99.68	0.9706	0.9675	0.9963
AlexNet	**Train**	99.22	99.21	99.91	0.9922	0.9914	0.9999
	Test	98.81	98.79	99.87	0.9880	0.9868	0.9999
GoogLeNet	**Train**	99.57	99.57	99.95	0.9957	0.9952	0.9999
	Test	98.80	98.80	99.84	0.9880	0.9879	0.9999
ShuffleNet	**Train**	98.97	98.92	99.89	0.9896	0.9886	0.9999
	Test	**98.83**	98.85	99.71	0.9779	0.9707	0.9994

3.2.2 Ensemble Model

The top three models based on accuracy from the comparison study were selected to construct an ensemble model, aiming to further enhance both accuracy and robustness. The final prediction is determined by averaging the predictions from each model in the designed combination. The performance metrics on the testing set are displayed in **Table 5**.

Table 5: Model Performance for Ensemble Models

Model	ACC(%)	Sn(%)	Sp(%)	F1	MCC	AUC
ResNet+ViT+ShuffleNet	**98.89**	98.96	99.84	0.9896	0.9885	0.9999
ResNet+ViT	98.82	98.61	99.85	0.9861	0.9853	0.9998
ResNet+ShuffleNet	98.81	98.68	99.87	0.9868	0.9854	0.9997
ViT+ShuffleNet	98.85	98.94	99.86	0.9868	0.9894	0.9998

According to the table above, the ensemble model combining ResNet, ViT, and ShuffleNet achieves the highest accuracy of 98.89%. This indicates a strong predictive capability for this model combination.

4 Code Availability

The complete code used in this coursework is available at *GitHub Repository*. Below are the key code snippets for Problems 2 and 3, as requested.

4.1 Problem 2

```
# CNN architexture
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv_block1 = nn.Sequential(
```

```python
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),# make the revision here
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),# make the revision here
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, stride=2),
        )
        self.conv_block2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, stride=2),
        )
        self.fcs = nn.Sequential(
            nn.Linear(2304, 1152),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(1152, 576),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(576, 10)
        )
    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fcs(x)
        return x

# training settings
device = "cuda"
model = CNN().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.SGD(params=model.parameters(), lr=0.001, momentum=0.9)  # make the revision here, can change into Adam
num_epochs = 10
```

4.2 Problem 3

4.2.1 ResNet

```python
class ResNet(nn.Module):
    def __init__(self,
                 block,
                 blocks_num,
                 num_classes=1000,
                 include_top=True,
                 groups=1,
                 width_per_group=64):
        super(ResNet, self).__init__()
        self.include_top = include_top
        self.in_channel = 64

        self.groups = groups
        self.width_per_group = width_per_group
        self.conv1 = nn.Conv2d(1, self.in_channel, kernel_size=7, stride=2,
                               padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(self.in_channel)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, blocks_num[0])
        self.layer2 = self._make_layer(block, 128, blocks_num[1], stride=2)
        self.layer3 = self._make_layer(block, 256, blocks_num[2], stride=2)
        self.layer4 = self._make_layer(block, 512, blocks_num[3], stride=2)

        if self.include_top:
            self.avgpool = nn.AdaptiveAvgPool2d((1, 1))  # output size = (1, 1)
            self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
        self.sigmoid = nn.Sigmoid()

    def _make_layer(self, block, channel, block_num, stride=1):
        downsample = None
        if stride != 1 or self.in_channel != channel * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channel, channel * block.expansion, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(channel * block.expansion))

        layers = []
        layers.append(block(self.in_channel, channel, downsample=downsample, stride=stride, groups=self.groups, width_per_group=self.width_per_group))

        self.in_channel = channel * block.expansion

        for _ in range(1, block_num):
            layers.append(block(self.in_channel, channel, groups=self.groups, width_per_group=self.width_per_group))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        if self.include_top:
            x = self.avgpool(x)
            x = torch.flatten(x, 1)
            x = self.fc(x)
            x = self.sigmoid(x)

        return x
```

4.2.2 LSTM

```python
class Rnn(nn.Module):
    def __init__(self, in_dim, hidden_dim, n_layer, n_classes):
        super(Rnn, self).__init__()
        self.n_layer = n_layer
        self.hidden_dim = hidden_dim
        self.lstm = nn.LSTM(in_dim, hidden_dim, n_layer, batch_first=True)
        self.classifier = nn.Linear(hidden_dim, n_classes)

    def forward(self, x):
        out, (h_n, c_n) = self.lstm(x)
        # x = out[:, -1, :]
        x = h_n[-1, :, :]
        x = self.classifier(x)
        return x
```

4.2.3 ViT

```python
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads, mlp_dim, channels=3):
        super().__init__()
        assert image_size % patch_size == 0, 'image dimensions must be divisible by the patch size'
        num_patches = (image_size // patch_size) ** 2
        patch_dim = channels * patch_size ** 2
        self.patch_size = patch_size
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.patch_to_embedding = nn.Linear(patch_dim, dim)
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.transformer = Transformer(dim, depth, heads, mlp_dim)
        self.to_cls_token = nn.Identity()
        self.mlp_head = nn.Sequential(
            nn.Linear(dim, mlp_dim),
            nn.GELU(),
            nn.Linear(mlp_dim, num_classes)
        )

    def forward(self, img, mask=None):
        p = self.patch_size
        x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = p, p2 = p)
        x = self.patch_to_embedding(x)
        cls_tokens = self.cls_token.expand(img.shape[0], -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding
        x = self.transformer(x, mask)
        x = self.to_cls_token(x[:, 0])
        return self.mlp_head(x)
```

4.2.4 EfficientNet

```python
class EfficientNetV2(nn.Module):
    def __init__(self,
                 model_cnf: list,
                 num_classes: int = 1000,
                 num_features: int = 1280,
                 dropout_rate: float = 0.2,
                 drop_connect_rate: float = 0.2):
        super(EfficientNetV2, self).__init__()

        for cnf in model_cnf:
            assert len(cnf) == 8

        norm_layer = partial(nn.BatchNorm2d, eps=1e-3, momentum=0.1)
        stem_filter_num = model_cnf[0][4]

        self.stem = ConvBNAct(1,
                              stem_filter_num,
                              kernel_size=3,
                              stride=2,
                              norm_layer=norm_layer)  # default activation function is SiLU

        total_blocks = sum([i[0] for i in model_cnf])
        block_id = 0
        blocks = []
        for cnf in model_cnf:
            repeats = cnf[0]
            op = FusedMBConv if cnf[-2] == 0 else MBConv
            for i in range(repeats):
                blocks.append(op(kernel_size=cnf[1],
                                 input_c=cnf[4] if i == 0 else cnf[5],
                                 out_c=cnf[5],
                                 expand_ratio=cnf[3],
                                 stride=cnf[2] if i == 0 else 1,
                                 se_ratio=cnf[-1],
                                 drop_rate=drop_connect_rate * block_id / total_blocks,
                                 norm_layer=norm_layer))
                block_id += 1
        self.blocks = nn.Sequential(*blocks)

        head_input_c = model_cnf[-1][-3]
        head = OrderedDict()

        head.update({"project_conv": ConvBNAct(head_input_c,
                                               num_features,
                                               kernel_size=1,
                                               norm_layer=norm_layer)})  # default activation function is SiLU

        head.update({"avgpool": nn.AdaptiveAvgPool2d(1)})
        head.update({"flatten": nn.Flatten()})

        if dropout_rate > 0:
            head.update({"dropout": nn.Dropout(p=dropout_rate, inplace=True)})
        head.update({"classifier": nn.Linear(num_features, num_classes)})

        self.head = nn.Sequential(head)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode="fan_out")
                if m.bias is not None:
                    nn.init.zeros_(m.bias)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.ones_(m.weight)
                nn.init.zeros_(m.bias)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.zeros_(m.bias)

        self.sigmoid = nn.Sigmoid()

    def forward(self, x: Tensor) -> Tensor:
        x = self.stem(x)
        x = self.blocks(x)
        x = self.head(x)
        x = self.sigmoid(x)

        return x
```

4.2.5 MobileNet

```python
class MobileNetV2(nn.Module):
    def __init__(self, num_classes=1000, alpha=1.0, round_nearest=8):
        super(MobileNetV2, self).__init__()
        block = InvertedResidual
        input_channel = _make_divisible(32 * alpha, round_nearest)
        last_channel = _make_divisible(1280 * alpha, round_nearest)

        inverted_residual_setting = [
            # t, c, n, s
            [1, 16, 1, 1],
            [6, 24, 2, 2],
            [6, 32, 3, 2],
            [6, 64, 4, 2],
            [6, 96, 3, 1],
            [6, 160, 3, 2],
            [6, 320, 1, 1],
        ]

        features = []
        features.append(ConvBNReLU(1, input_channel, stride=2))
        for t, c, n, s in inverted_residual_setting:
            output_channel = _make_divisible(c * alpha, round_nearest)
            for i in range(n):
                stride = s if i == 0 else 1
                features.append(block(input_channel, output_channel, stride, expand_ratio=t))
                input_channel = output_channel
        features.append(ConvBNReLU(input_channel, last_channel, 1))
        self.features = nn.Sequential(*features)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.classifier = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(last_channel, num_classes)
        )

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out')
                if m.bias is not None:
                    nn.init.zeros_(m.bias)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.ones_(m.weight)
                nn.init.zeros_(m.bias)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.zeros_(m.bias)

        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        x = self.sigmoid(x)

        return x
```

4.2.6 AlexNet

```python
class AlexNet(nn.Module):
    def __init__(self,width_mult=1):
        super(AlexNet,self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1,32,kernel_size=3,padding=1),
            nn.MaxPool2d(kernel_size=2,stride=2),
            nn.ReLU(inplace=True),
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(32,64,kernel_size=3,padding=1),
            nn.MaxPool2d(kernel_size=2,stride=2),
            nn.ReLU(inplace=True),
        )

        self.layer3 = nn.Sequential(
            nn.Conv2d(64,128,kernel_size=3,padding=1),
        )

        self.layer4 = nn.Sequential(
            nn.Conv2d(128,256,kernel_size=3,padding=1),
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(256,256,kernel_size=3,padding=1),
            nn.MaxPool2d(kernel_size=3,stride=2),
            nn.ReLU(inplace=True),
        )

        self.fc1 = nn.Linear(256 * 3 * 3,1024)
        self.fc2 = nn.Linear(1024,512)
        self.fc3 = nn.Linear(512,10)


    def forward(self,x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x = x.view(-1,256*3*3)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)

        return x
```

4.2.7 GoogLeNet

```python
class SimpleGoogLeNet(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleGoogLeNet, self).__init__()
        self.inception1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.inception2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.inception3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.fc = nn.Sequential(
            nn.Linear(128 * 3 * 3, 1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, num_classes)
        )

    def forward(self, x):
        x = self.inception1(x)
        x = self.inception2(x)
        x = self.inception3(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

4.2.8 ShuffleNet

```python
class ShuffleNetV1(nn.Module):
    def __init__(self, groups = 3, in_channels=3, num_classes=1000):

        super(ShuffleNetV1, self).__init__()

        self.groups = groups
        self.stage_repeats = [3, 7, 3]
        self.in_channels = in_channels
        self.num_classes = num_classes

        if groups == 1:
            self.stage_out_channels = [-1, 24, 144, 288, 567]
        elif groups == 2:
            self.stage_out_channels = [-1, 24, 200, 400, 800]
        elif groups == 3:
            self.stage_out_channels = [-1, 24, 240, 480, 960]
        elif groups == 4:
            self.stage_out_channels = [-1, 24, 272, 544, 1088]
        elif groups == 8:
            self.stage_out_channels = [-1, 24, 384, 768, 1536]
        else:
            raise ValueError(
                """{} groups is not supported for 1x1 Grouped Convolutions""".format(groups))

        self.conv1 = conv3x3(self.in_channels,
                             self.stage_out_channels[1],  # stage 1
                             stride=2)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.stage2 = self._make_stage(2)
        self.stage3 = self._make_stage(3)
        self.stage4 = self._make_stage(4)

        num_inputs = self.stage_out_channels[-1]
        self.fc = nn.Linear(num_inputs, self.num_classes)
        self.init_params()

    def init_params(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                init.kaiming_normal(m.weight, mode='fan_out')
                if m.bias is not None:
                    init.constant(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                init.constant(m.weight, 1)
                init.constant(m.bias, 0)
            elif isinstance(m, nn.Linear):
                init.normal(m.weight, std=0.001)
                if m.bias is not None:
                    init.constant(m.bias, 0)

    def _make_stage(self, stage):
        modules = OrderedDict()
        stage_name = "ShuffleUnit_Stage{}".format(stage)
        grouped_conv = stage > 2

        first_module = ShuffleUnit(
            self.stage_out_channels[stage - 1],
            self.stage_out_channels[stage],
            groups=self.groups,
            grouped_conv=grouped_conv,
            combine='concat'
        )
        modules[stage_name + "_0"] = first_module

        for i in range(self.stage_repeats[stage - 2]):
            name = stage_name + "_{}".format(i + 1)
            module = ShuffleUnit(
                self.stage_out_channels[stage],
                self.stage_out_channels[stage],
                groups=self.groups,
                grouped_conv=True,
                combine='add'
            )
            modules[name] = module
        return nn.Sequential(modules)

    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        x = self.stage2(x)
        x = self.stage3(x)
        x = self.stage4(x)
        x = F.avg_pool2d(x, x.data.size()[-2:])
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

References

[1] Ramprasaath R. Selvaraju, Abhishek Das, R. D. Gupta, and Devi Parikh, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 618-626, 2017.

[2] K. Chattopadhay, A. Sarkar, P. Howlader, and A. V. K. M. S. K. Das, "Grad-CAM++: Generalized Visual Explanations for Deep Convolutional Networks," *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 839-847, 2018.

[3] H. Wang, J. Huang, H. Xu, and L. Liu, "Layer-CAM: A Novel Approach for Visualizing Convolutional Neural Networks," *arXiv preprint arXiv:2006.10173*, 2020.

[4] R. Wang, X. Zhang, Z. Yin, and Z. Yang, "Score-CAM: Score-Weighted Visual Explanations for Convolutional Neural Networks," *Proceedings of the*

IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3901-3911, 2020.

[5] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, 2016.

[6] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[7] A. Dosovitskiy, E. M. M. Keller, Y. Kolesnikov, A. Beyer, and D. G. Z. Schiele, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” *arXiv preprint arXiv:2010.11929*, 2020.

[8] M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[9] A. Howard, M. Sandler, G. Chu, L. Chen, W. Chen, Y. Tan, Q. Wang, T. Yang, and H. Zhang, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *arXiv preprint arXiv:1704.04861*, 2017.

[10] A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Advances in Neural Information Processing Systems*, pp. 1097-1105, 2012.

[11] C. Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dongyang Anguelov, “Going Deeper with Convolutions,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1-9, 2015.

[12] X. Zhang, X. Wu, Z. Zhang, and J. Yao, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6848-6856, 2018.