RE50800 – Deep Learning-HW2

Image Classification with Multiple Models and Features

Author: RE6121045 數據所碩一 侯登耀
Github: https://github.com/DENGYAHOU/DL_HW2

1. Introduction:

In this study, I designed two different tasks of classification models for image classification using the TinyImageNet dataset, which comprises 50 classes, with 63325 training samples, 450 validation samples, and 450 testing samples. The tasks include I. design a dynamic convolution module that can adjust its weights based on the number of input channels. This can be achieved by learning a weight-generating network that takes the input channels as an input and generates corresponding convolution kernels. II. Design a two-layer to four-layer CNN network that can achieve 90% performance of ResNet34 on ImageNet-mini (i.e., with no more than 10% performance loss). The motivation behind this study is to design various deep learning models tailored for specific tasks, with the secondary goal of enhancing image classification accuracy.

2. Task I. Methodology:

Our solution leverages an attention mechanism to dynamically adjust convolutional parameters based on the input characteristics. This approach allows the module to adapt to different numbers of input channels without requiring re-initialization of the model. The core components of our module are:

1. Attention Mechanism: An attention module that dynamically computes softmax weights based on the input's global context. This mechanism guides the selection and combination of multiple sets of convolutional filters.

2. Dynamic Convolutional Weights: Instead of having a fixed set of convolutional filters, our module generates a weighted combination of filters from a predefined set based on the attention scores. This allows the module to adapt to varying input channels efficiently.

- Module Architecture

➢ Attention Module:

The attention module is designed to generate softmax weights based on the input's global context. The key steps are:

- Global Average Pooling: Reduces the spatial dimensions of the input to a single value per channel, summarizing the global information.
- Multi-Layer Perceptron (MLP): A sequence of convolutional layers that transforms the pooled input to generate attention scores for K different sets of filters.
- Softmax Activation: Normalizes the attention scores to sum to one, creating a probabilistic distribution over the K filter sets.

➢ VariableInputConv Module:

The main convolutional module adapts dynamically to the input's characteristics:

Dynamic Initialization: The attention module and the convolutional weights are dynamically initialized based on the input's channel size.

- Dynamic Weight Aggregation: Convolutional weights are combined based on the attention scores, creating a set of filters tailored to the current input.
- Efficient Convolution: The aggregated weights are used to perform the convolution operation on the input, producing the output.

- Classification Models:

To evaluate the effectiveness of the proposed module, we integrated VariableInputConv with ResNet-18 and performed classification tasks on the CIFAR-10 dataset.

3. Task I. Experiments:

1. Load and Preprocess Image Data

I load images and their labels from a file, perform image processing, and save them as numpy arrays. The processing

includes:

- Resizing: Standardizing images to a size of 124x124 pixels.
- Denoising: Applying Non-Local Means Denoising to reduce noise.
- Normalization: Scaling pixel values to enhance the contrast.

2. Create Datasets and Data Loaders

I Convert numpy arrays to PyTorch tensors and create TensorDataset and DataLoader objects for training, validation, and testing.

- Extract and prepare different combinations of color channels from the test dataset.
- Extract and prepare different combinations of color channels from the test dataset.
- I now use these data loaders to perform inference with the VariableInputConv model integrated into ResNet. Each data loader represents a different combination of channels.

3. Experimental Setup

- Training: Standard training procedures using Adam optimizer and cross-entropy loss.
- Metrics: Classification accuracy and parameter efficiency were measured to evaluate model performance.

4. Task I. Results and Discussion:

Figure 1. compares the computational costs and parameters of two models: ResNet18 and VariableInputConv. The ResNet18 model has approximately 11.1927 million parameters and requires about 251.993 GFLOPs for computation, whereas the VariableInputConv model, designed for handling variable input channels, has a slightly higher number of parameters at 11.1939 million and demands more computational power, with a requirement of approximately 301.994 GFLOPs.

Model / Compute Costs	ResNet18	VariableInput Conv
#PARAMS (M)	11.1927	11.1939
FLOPS (GFLOPS)	251.993	301.994

Figure 1. #PARAMS and FLOPS

Figure 2. presents a performance comparison between ResNet18 and the VariableInputConv model over 30 epochs. ResNet18 outperforms the VariableInputConv model significantly, achieving a training accuracy of 63.32% compared to 16.67%, and a lower training loss of 1.1510 versus 1.3967. Similarly, in validation, ResNet18 maintains higher accuracy at 59.11% and a lower loss of 1.4622, whereas the VariableInputConv model lags with a validation accuracy of 16.67% and a higher loss of 3.6941. When tested on RGB data, ResNet18 achieves 54.89% accuracy, while VariableInputConv only reaches 14.60%.

Model / Performance	ResNet18	Variable InputConv
# of epochs	30	30
Training Acc (%)	63.32%	16.67%
Training Loss	1.1510	1.3967
Validation Acc (%)	59.11%	16.67%
Validation Loss	1.4622	3.6941
Testing Acc on RGB (%)	54.89%	14.60%

Figure 2. Experiment results

Figure 3. details the testing accuracy of the VariableInputConv model across different color channel configurations. The model achieves its highest accuracy of 18.67% when tested with the red (R) channel alone, followed by 16.67% with the green (G) channel and 15.70% with the blue (B) channel. However, accuracy decreases when combining channels: 16.05% for red and green (RG), 15.60% for red and blue (RB), and 15.22% for green and blue (GB). Interestingly, the model performs worst with all three channels combined (RGB), achieving only 14.60% accuracy.

This suggests that the VariableInputConv model struggles to integrate information from multiple channels effectively, performing best when processing single channels independently.

Channel	Testing Accuracy
R	18.67%
G	16.67%
B	15.70%
RG	16.05%
RB	15.60%
GB	15.22%
RGB	14.60%

Figure 3. VariableInputConv model results across different color channel configurations.

Here are some summary of findings:

- VariableInputConv demonstrated flexibility in handling different input channels but struggled with performance, particularly in combining multiple channels.
- The module achieved better accuracy when processing single channels independently, suggesting room for improvement in the integration and processing of multi-channel inputs.
- Compared to the standard ResNet-18, the VariableInputConv model had significantly lower accuracy and higher loss, indicating that while innovative, the approach needs further refinement to compete with traditional convolutional models in terms of performance.
- The performance disparities highlight the complexity and challenges in designing dynamic convolutional architectures that can efficiently handle variable input channels.

5. Task II. Methodology:

In my project, I have designed a convolutional neural network (CNN) architecture for image classification that integrates advanced techniques such as self-attention mechanisms and deformable convolution networks. Here's a detailed breakdown of my approach:

1. Network Architecture

Here are the layers I used in my model:

- Conv2d Layer (layer1):
  - Input: (Batch Size, 3, 124, 124) - Assuming standard image input with 3 color channels (RGB).
  - Output: (Batch Size, 64, 62, 62) - Output channels: 64, spatial dimensions reduced by the convolution and stride.
  - Attention Layer (attention1):
  - This layer applies an element-wise product, modulating the input but does not change the dimensions of the input directly. Hence, it's not a primary input-output layer in terms of changing dimensions.
  - LayerNorm (bn1):
    - Input: (Batch Size, 64, 62, 62) - From the previous layer output.
    - Output: (Batch Size, 64, 62, 62) - Normalizes the input but keeps the dimensions the same.
  - MaxPool2d (pool1):
    - Input: (Batch Size, 64, 62, 62) - From the previous normalization layer.
    - Output: (Batch Size, 64, 31, 31) - Reduces spatial dimensions by half using pooling.
- Conv2d Layer (layer2):
  - Input: (Batch Size, 64, 31, 31) - Output from the previous pooling layer.
  - Output: (Batch Size, 64, 31, 31) - Keeps spatial dimensions same due to padding.
  - LayerNorm (bn):
    - Input: (Batch Size, 64, 31, 31) - From the previous layer output.
    - Output: (Batch Size, 64, 31, 31) - Normalizes the input but does not change dimensions.
- Conv2d Layer (layer3):

- Input: (Batch Size, 64, 31, 31) - Output from the previous normalization layer.
- Output: (Batch Size, 64, 31, 31) - Maintains the same dimensions due to padding.

- DeformableConv2d Layer (layer4):
  - Input: (Batch Size, 64, 31, 31) - From the previous convolutional layer.
  - Output: (Batch Size, 128, 31, 31) - Changes the number of channels to 128, keeps spatial dimensions same due to padding.
  - LayerNorm (bn2):
    - Input: (Batch Size, 128, 31, 31) - From the previous deformable convolution layer.
    - Output: (Batch Size, 128, 31, 31) - Normalizes the input without changing dimensions.
  - CBAM (attention2):
    - This layer applies channel and spatial attention. It modifies the feature maps through attention but keeps the dimensions unchanged.
  - MaxPool2d (pool2):
    - Input: (Batch Size, 128, 31, 31) - From the previous normalization layer.
    - Output: (Batch Size, 128, 15, 15) - Reduces spatial dimensions by half.
  - AdaptiveAvgPool2d (pooling):
    - Input: (Batch Size, 128, 15, 15) - From the previous pooling layer.
    - Output: (Batch Size, 128, 1, 1) - Reduces the spatial dimensions to a fixed size of 1x1.
    - Flatten Operation:
      - Input: (Batch Size, 128, 1, 1) - Flattening the pooled output for the fully connected layer.
      - Output: (Batch Size, 128) - Converts the 4D tensor to a 2D tensor.

- Fully Connected Layer (fc1):
  - Input: (Batch Size, 128) - From the flattened layer.
  - Output: (Batch Size, 50) - Outputs the class scores for 50 classes.

In total, there are 5 key input-output layers that change the input's dimensions or the number of channels:

- Conv2d (layer1)
- Conv2d (layer2)
- Conv2d (layer3)
- DeformableConv2d (layer4)
- Fully Connected Layer (fc1)

Each of these layers significantly transforms the input data either by altering the number of channels or by changing the spatial dimensions, ultimately leading to the final classification output.

2. Explanation on DeformableConv2d

The DeformableConv2d class in my network is an implementation of deformable convolutions, a powerful variation of the standard convolutional layer that allows for dynamic and flexible adjustments to the receptive field based on the input features. This ability to adapt the shape and size of the convolutional kernel helps in capturing spatial transformations and irregularities in the input data, making it particularly useful for tasks like object detection and segmentation.

Here's a detailed breakdown of each component and function within the DeformableConv2d class:

- Initialization (__init__ method)
  1. Parameter Initialization:
     - in_channels: Number of input channels (depth) of the input tensor.
     - out_channels: Number of output channels produced by the convolution.
     - kernel_size: Size of the convolution kernel/filter. It can be an integer or a tuple for specifying different dimensions.
     - stride: The stride of the convolution. It can be a single integer or a tuple for specifying different strides along each dimension.
     - padding: Amount of padding added to the input tensor.

- dilation: Spacing between kernel elements (dilation rate).
- bias: Whether to include a bias term in the convolution.

2. Kernel Size and Stride:
   - Ensures that kernel_size and stride are tuples, converting integers to tuples if necessary. This allows flexibility in specifying dimensions.

3. Offset Convolution (self.offset_conv):
   - Computes the offsets for each pixel in the input feature map. These offsets determine where the kernel should sample the input values.
   - The number of output channels for this convolution is 2 * kernel_size[0] * kernel_size[1] because it computes 2D offsets (x and y) for each position in the kernel.
   - The offsets are initialized to zero, meaning no initial displacement, allowing the model to learn the optimal offsets during training.

4. Modulator Convolution (self.modulator_conv):
   - Computes modulation scalar values for each kernel position, which adjust the importance of each sampled point.
   - The number of output channels for this convolution is kernel_size[0] * kernel_size[1], representing a single modulator value per kernel position.
   - The modulators are initialized to zero, and they are scaled to [0, 2] through a sigmoid function, providing a modulation factor that can reduce or enhance the influence of certain points.

5. Regular Convolution (self.regular_conv):
   - This is the standard convolutional layer that performs the actual convolution operation with learned weights.
   - Takes into account the offsets and modulations computed by the previous layers to adjust how the convolution is applied to the input.
   - This layer is responsible for producing the output feature map with the specified number of out_channels.

- Key Features of Deformable Convolutions
  - Adaptive Receptive Fields: The deformable convolution can change its receptive field dynamically based on the input data. This means it can effectively capture features even when there are distortions or variations in the spatial layout of the input.
  - Enhanced Spatial Awareness: By learning offsets, the network can focus more precisely on relevant parts of the input, enhancing its ability to recognize objects or features regardless of their spatial configuration.
  - Modulation of Feature Importance: The modulation mechanism allows the network to adjust the significance of different parts of the input dynamically, making it more flexible in handling complex and varied input patterns.

6. Task II. Experiments:

1. Load and Preprocess Image Data

I load images and their labels from a file, perform image processing, and save them as numpy arrays. The processing includes:

- Resizing: Standardizing images to a size of 124x124 pixels.
- Denoising: Applying Non-Local Means Denoising to reduce noise.
- Normalization: Scaling pixel values to enhance the contrast.

2. Create Datasets and Data Loaders

I Convert numpy arrays to PyTorch tensors and create TensorDataset and DataLoader objects for training, validation, and testing.

- I now use these data loaders to train and test my proposed model and I also use ResNet34 model for comparison.

3. Experimental Setup
   - Training: Standard training procedures using Adam optimizer and cross-entropy loss.
   - Metrics: Classification accuracy and parameter efficiency were measured to evaluate model performance.

4. Task II. Results and Discussion:

Compared to the ResNet34 model, the VariableInputConv model demonstrates a significantly lower computational cost, with only 1.109 million parameters and 23.282 GFLOPS, making it far more efficient than ResNet34's 21.3103 million parameters and 59.874 GFLOPS.

Model / Compute Costs	ResNet34	VariableInput Conv
#PARAMS (M)	21.3103	1.109
FLOPS (GFLOPS)	59.874	23.282

Figure 4. #PARAMS and FLOPS

While the VariableInputConv model achieves much lower computational costs than ResNet34, its performance metrics suggest room for improvement; over 30 epochs, it reached 50.55% training accuracy, 46.88% validation accuracy, and 44.00% testing accuracy, which are considerably lower than ResNet34's corresponding accuracies of 97.83%, 48.88%, and 46.22%, respectively, albeit with comparable validation and testing loss values.

Model / Performance	ResNet34	Variable InputConv
# of epochs	30	30
Training Acc (%)	97.83%	50.55%
Training Loss	0.0018	0.0514
Validation Acc (%)	48.88%	46.88%
Validation Loss	0.0586	0.0514
Testing Acc (%)	46.22%	44.00%

Figure 5. Experiment results

5. References:

1. Deformable Convolutional Networks
2. flops