DRP-AI Extension Pack (Pruning Tool) Version 1.2.0

User's Manual

Contents

1. Overview

This section describes the operating environment and functions of the DRP-AI Extension Pack.

1.1 Product Configuration

Table 1.1 Product Configuration

Item	Description
r20ut5188ej0300-drp-ai-extension-pack.pdf	This manual
drpai-extension-pack_ver1.2.0.tar.gz	DRP-AI Extension Pack (product covered by this manual)

Table 1.2 Configuration of Files in drpai-extension-pack_ver1.2.0.tar.gz

Configuration of Files	Description
🗀 drpai-extension-pack_ver1.2.0.tar.gz	
🗀 drpai_compaction_tool	API library of the functions and class listed in Table 4.1
🗁 samples	
🗁 classification	
🗁 pytorch_mobilenetv2	Sample code using MobileNetV2 of the PyTorch version. See 3.8.1 for details.
🗁 tensorflow_cnn	Sample code using CNN of the TensorFlow version. See 3.8.2 for details.

1.2 Operating Environment

The operating environment and software to be installed for the DRP-AI Extension Pack in each case are shown in the following tables.

Table 1.3 Operating Environment (When Using PyTorch)

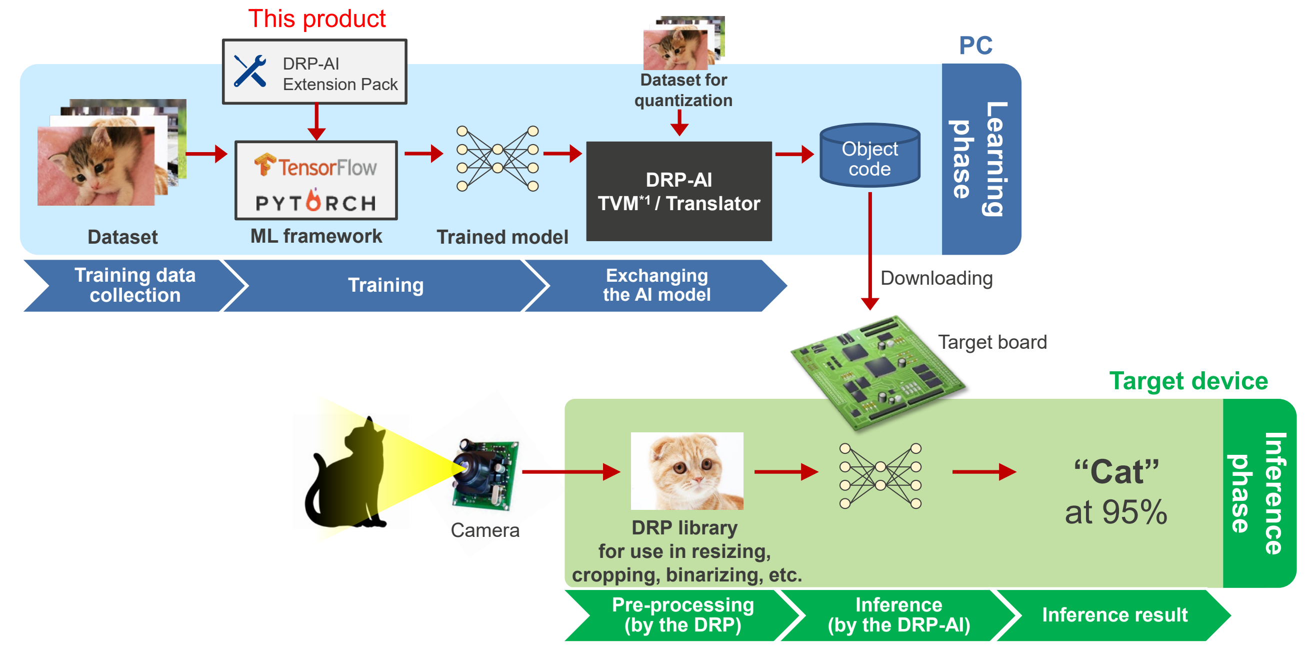
Item	Software Name and Version Number, etc.
Operating environment	Ubuntu 22.04 LTS, 64-bit version
	CUDA 11.8
Software to be installed	Python 3.10.16
	torch==1.13.1
	torchvision==0.14.1
	torchstat==0.0.7
	pandas==1.4.2
	onnx==1.14.0

Table 1.4 Operating Environment (When Using TensorFlow)

Item	Software Name and Version Number, etc.
Operating environment	Ubuntu 22.04 LTS, 64-bit version
	CUDA 12.5
Software to be installed	Python 3.10.16
	tensorflow==2.18.0
	tensorflow-model-optimization==0.8.0
	tf_keras==2.18.0
	tf2onnx==1.16.1
	onnx==1.14.0

1.3 Function

The DRP-AI Extension Pack provides a pruning function optimized for the DRP-AI. A general description of pruning is given under 1.4, Pruning, on the following page. This pruning function optimized for the DRP-AI can be used by using the DRP-AI Extension Pack in combination with the training code written with the use of PyTorch or TensorFlow.



*1 DRP-AI TVM is powered by EdgeCortix MERA™ Compiler Framework.

Figure 1-1 Deployment Flow

1.4 Pruning

Nodes are interconnected in a neural network as shown in the figure below. Methods of reducing the number of parameters by removing weights between nodes or removing nodes are referred to as “pruning”. A neural network to which pruning has not been applied is generally referred to as a dense neural network.

Applying pruning to a neural network leads to a slight deterioration in the accuracy of the model but can reduce the power required by hardware and accelerate the inference process.

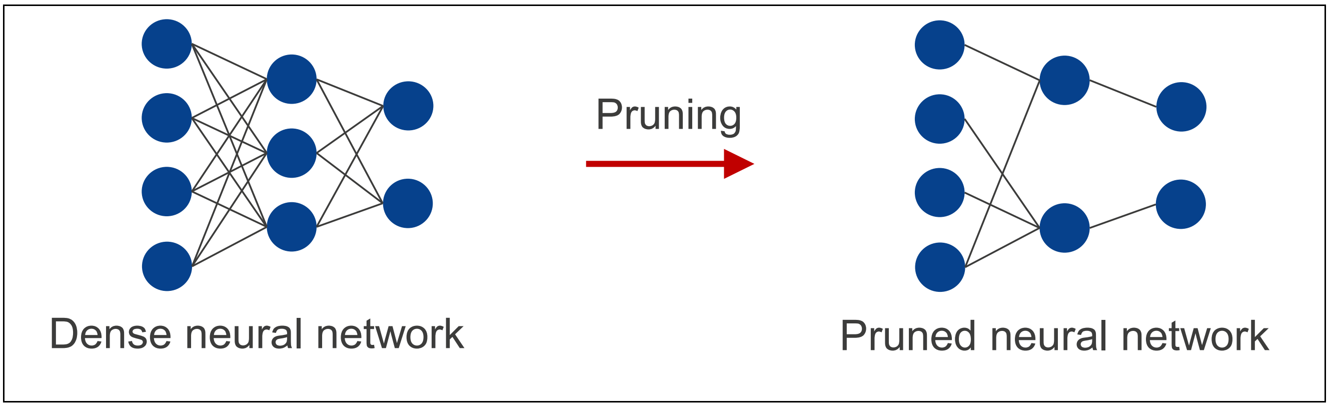


Figure 1-2 Schematic View of the Pruning of a Neural Network

Note: In the use of this product, we recommend pruning by at least 70% to improve the processing performance of the DRP-AI.

1.5 Relationship between Compressing AI Models and DRP-AI Performance

DRP-AI for RZ/V2H supports the feature of efficiently calculating the pruned AI model. Therefore, power efficiency is improved by using the pruned AI model.

The following graph provides an example of improvement in power efficiency when changing from an unpruned AI model to a pruned AI model. Compared to unpruned AI models, pruned AI models are significantly more power-efficient.

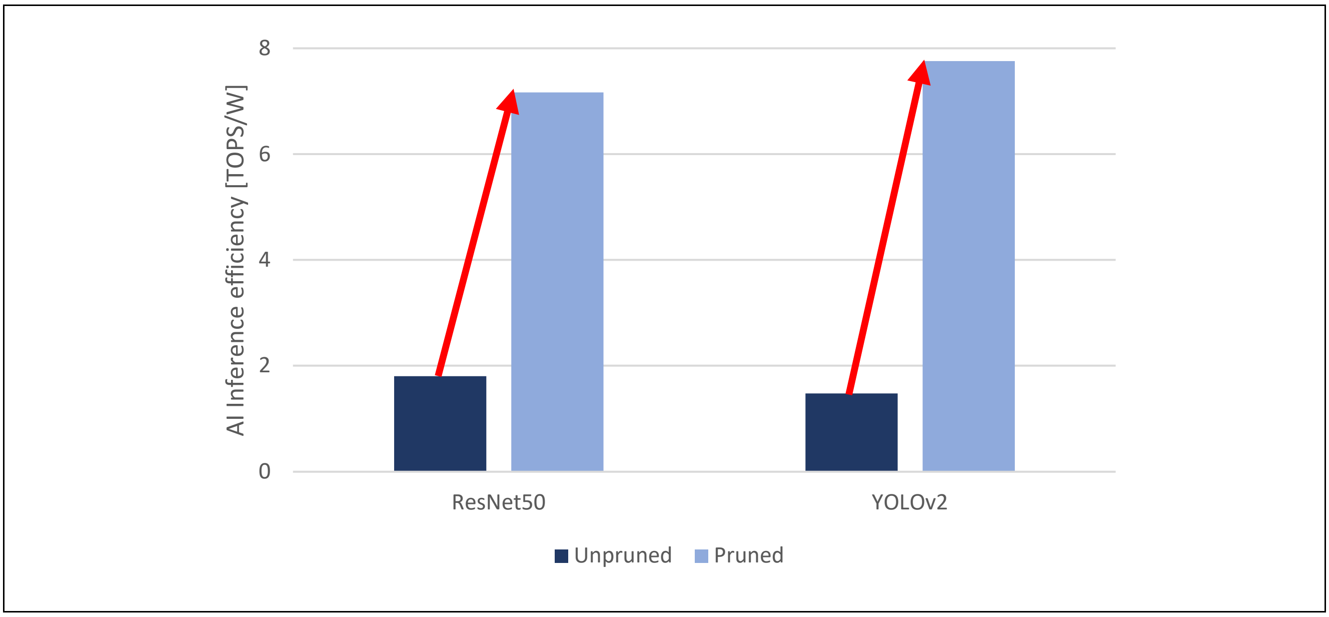
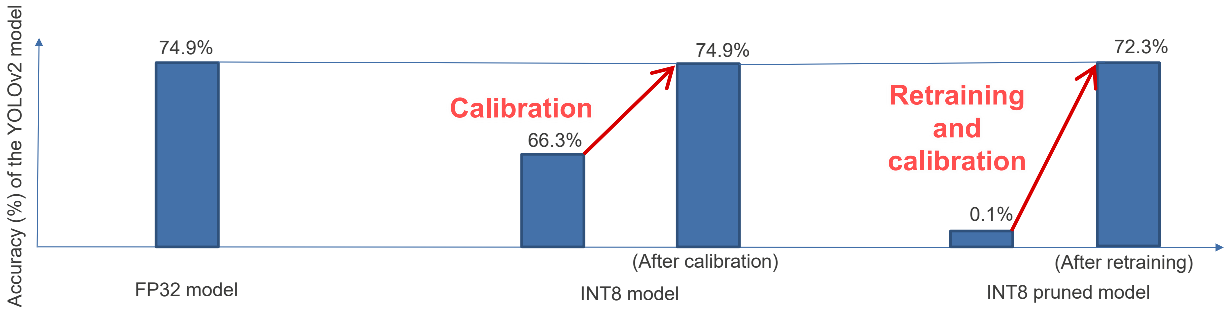


Figure 1-3 DRP-AI Performance after Compressing AI Models

Note: Quantization was applied to the AI models and measurements were performed.

Applying compressing processing such as pruning and quantization to AI models might generally lower the accuracy of models. Using the DRP-AI Extension Pack in pruning, however, allows ensuring the same or almost the same accuracy for the AI model as that before pruning by proceeding with retraining after pruning. The figure below shows the results of changes in accuracy with the use of the YOLOv2 models. Compared with an accuracy of 74.9% before compressing, that after compressing (pruning plus quantization) can reach 72.3%.



OSS Pre-trained Model	RZ/V2H	
FP32	INT8 dense (INT8 quantization)	INT8 sparse (INT8 quantization + pruning)

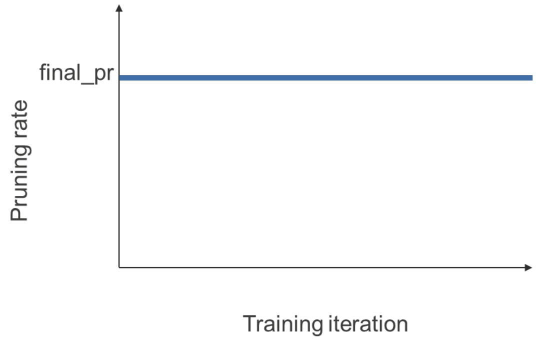
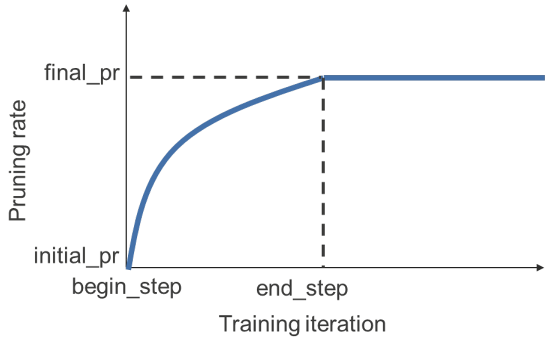
Figure 1-4 Changes in the Accuracy of YOLOv2 Models after Compressing

Note: For details on calibration and quantization, see the DRP-AI_Quantizer User's Manual.

1.6 Two Pruning Modes

This product supports two pruning modes. One-shot pruning is characterized by a relatively short training time being associated with pruning. In gradual pruning, longer training times are associated with pruning than in one-shot pruning, but the accuracy may be improved. Table 1.5 shows a comparison of the two pruning modes. The pruning rate rises as training proceeds in gradual pruning. For details on how to set one-shot pruning or gradual pruning, see 4.2.2 or 4.3.2, depending on whether you are using PyTorch or TensorFlow, respectively. Note that the initial application of one-shot pruning is recommended.

Table 1.5 Comparison of the Two Pruning Modes

One-shot pruning (recommended)	Pruning is applied only once as the initial step. 
Gradual pruning	Pruning is applied gradually. 

1.7 Updates in Version 1.2.0

The updates in DRP-AI Extension Pack V1.2.0 are as follows.

- ✓ Supported Ubuntu 22.04 and Python 3.10.
    - For the Dockerfile for environment setup, please refer to the following.
        - https://github.com/renesas-rz/rzv_drp-ai_tvm/tree/main/pruning/setup
- ✓ Changed the supported TensorFlow version from 2.5.0 to 2.18.0
    - Please note that there are usage considerations for using the pruning tool for TensorFlow. For more information, please refer to 6. Usage Notes.
- ✓ Supported PyTorch’s torch.nn.MultiheadAttention() layer for pruning Transformer structure.
    - PyTorch torch.nn.MultiheadAttention() : https://pytorch.org/docs/1.13/generated/torch.nn.MultiheadAttention.html#multiheadattention

2. Setting Up the DRP-AI Extension Pack

This section describes how to set up the DRP-AI Extension Pack. Descriptions in this section are on the assumption that Python 3.10.16 has been set up on a PC running Ubuntu.

To build an environment using Docker, please refer to the following.

- https://github.com/renesas-rz/rzv_drp-ai_tvm/tree/main/pruning/setup

2.1 Installing the Library for Use by the DRP-AI Extension Pack

Install the following library on a PC running Ubuntu.

[When using PyTorch]

```
$ pip3 install torch==1.13.1 torchvision==0.14.1 \
                    --extra-index-url https://download.pytorch.org/whl/cu118
$ pip3 install torchstat==0.0.7 pandas==1.4.2 onnx==1.14.0
```

[When using TensorFlow]

```
$ pip3 install tensorflow==2.18.0 \
              tensorflow-model-optimization==0.8.0 \
              tf_keras==2.18.0 \
              tf2onnx==1.16.1 \
              onnx==1.14.0
```

2.2 Adding the Environment Variable

Register the working directory as an environment variable.

```
$ export WORK=/home/<Path to working directory>
```

Note: Change <Path to working directory> to suit the environment of the PC you are using.

2.3 Decompressing the DRP-AI Extension Pack

Place drpai-extension-pack_ver*.tar.gz in the working directory and execute the following command.

```
$ cd $WORK
$ tar -xvf drpai-extension-pack_ver*.tar.gz
[When using TensorFlow, also execute the following command.]
$ drpai_compaction_tool/scripts/setup_tf.sh
```

2.4 Adding the DRP-AI Extension Pack path to the environment variable

Execute the following command to add the DRP-AI Extension Pack path to the environment variable.

```
$ cd $WORK
$ export PYTHONPATH="$(pwd):$PYTHONPATH"
$ export TF_USE_LEGACY_KERAS=1
```

Note: Once you have ended the terminal session, re-execute the command stated above when you intend to use the extension pack again.

Execute the following command. With output of the version number, the setup processing is completed.

```
$ python3 -c "import drpai_compaction_tool; print(drpai_compaction_tool.__version__)"
<DRP-AI Extension Pack version>
[When using TensorFlow, ensure that the following commands do not generate any errors.]
$ python3 -c "from drpai_compaction_tool.tensorflow import Pruner"
```

Note: <DRP-AI Extension Pack version> depends on the version you are using.

3. Using the DRP-AI Extension Pack

This section describes how to use the DRP-AI Extension Pack.

3.1 Flow of Using the DRP-AI Extension Pack

Use the DRP-AI Extension Pack in combination with the training code written with the use of PyTorch or TensorFlow. Figure 3-1 shows the flow of using the DRP-AI Extension Pack.
The flow consists of two steps. The first step is initial training. Initial training involves training of the AI model without pruning. Use the code for use in initial training and a dataset you have prepared.

The second step is pruning and then retraining. This includes retraining of the AI model by adding the DRP-AI Extension Pack to the code for use in initial training. For details on how to add the DRP-AI Extension Pack to the code for use in initial training, see 3.2, [PyTorch] Adding the DRP-AI Extension Pack. Check the accuracy of the AI model after one round of pruning then retraining has been completed. Repeat pruning then retraining with increasingly high pruning rates while confirming that the rates in use do not create problems in terms of accuracy.

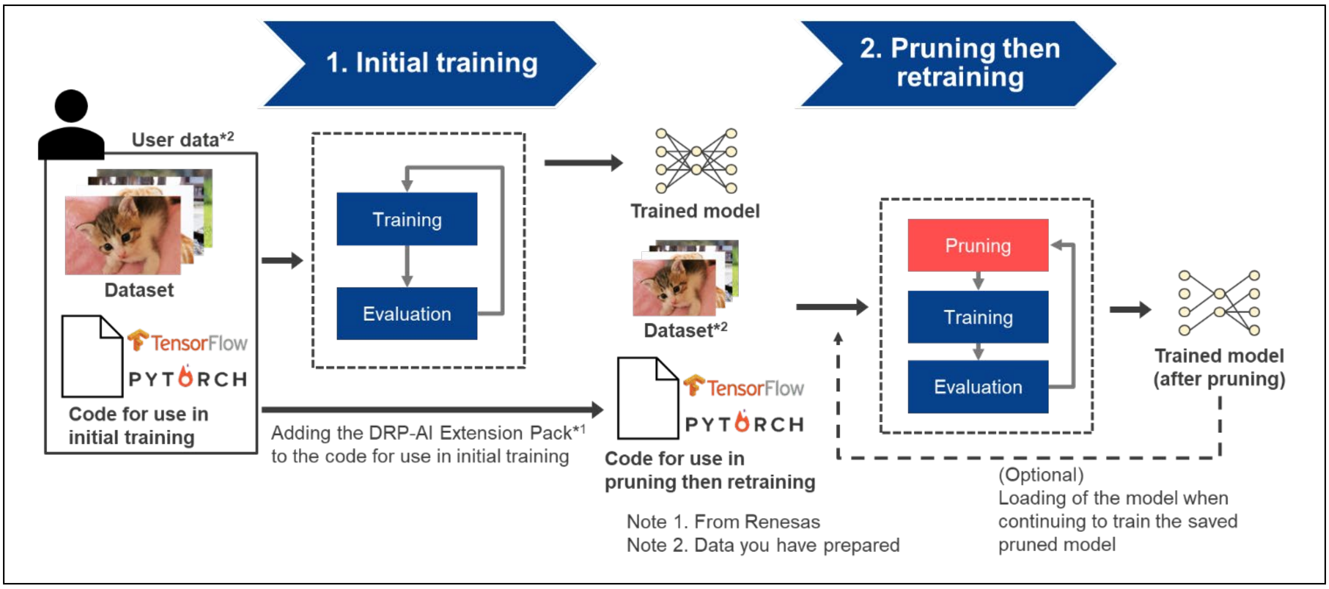
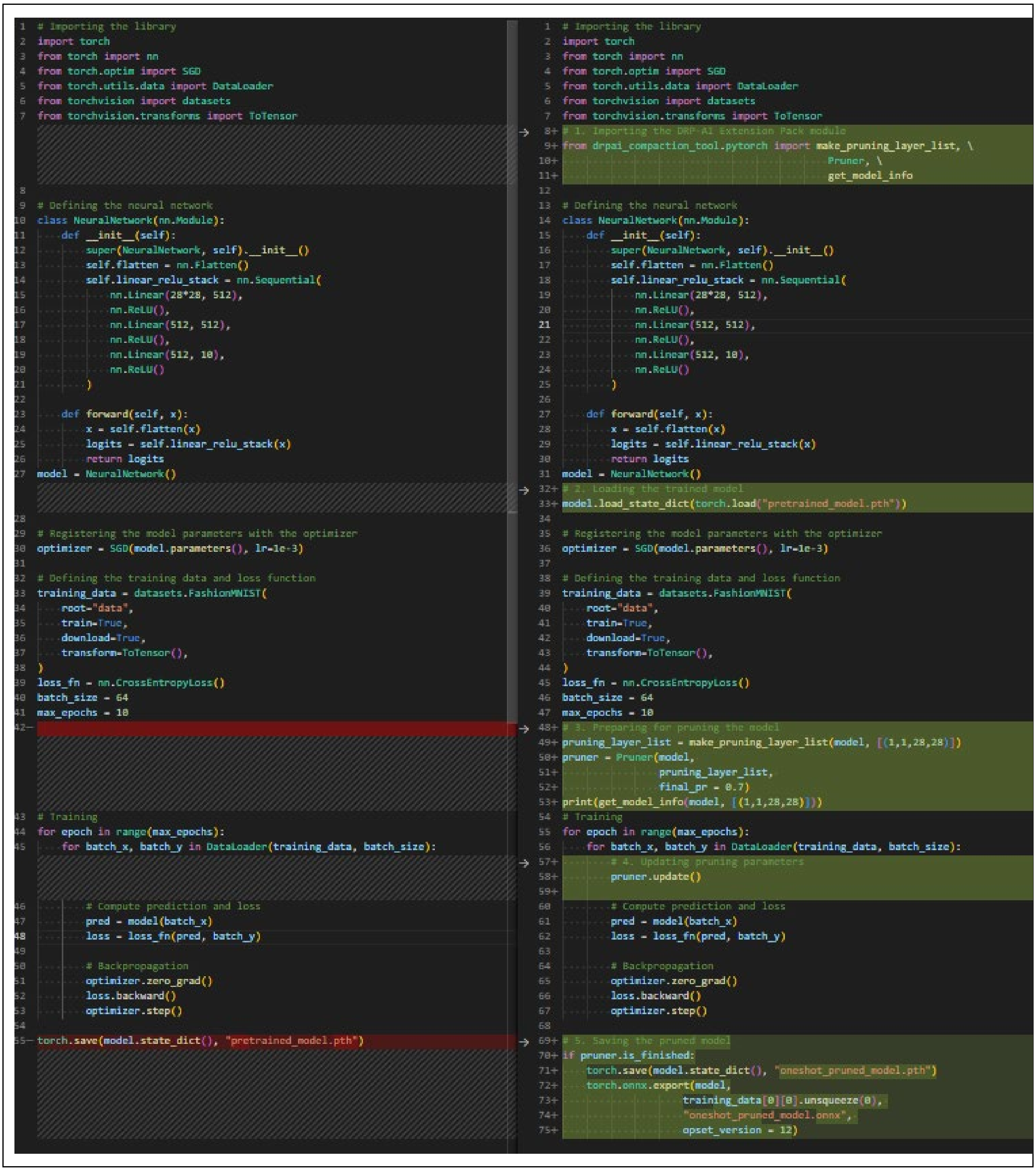

1. Initial training
2. Pruning then retraining
User data*2
Dataset
TensorFlow
PYTORCH
Code for use in initial training
Training
Evaluation
Trained model
Dataset*2
Adding the DRP-AI Extension Pack*1 to the code for use in initial training
Code for use in pruning then retraining
Pruning
Training
Evaluation
Trained model (after pruning)
(Optional) Loading of the model when continuing to train the saved pruned model
Note 1. From Renesas
Note 2. Data you have prepared

Figure 3-1 Flow of Using the DRP-AI Extension Pack

Figure 3-2 consists of listings of the code written with PyTorch for use in initial training without and with addition of the DRP-AI Extension Pack. The code in the left column is that for initial training and the code in the right column is that for pruning then retraining. The green shading indicates the differences between the two listings, that is, the several lines that are added to make the DRP-AI Extension Pack usable. For details on the case of using PyTorch, see 3.2. For details on the case of using TensorFlow, see 3.5.



```python
# Importing the library
import torch
from torch import nn
from torch.optim import SGD
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Defining the neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
model = NeuralNetwork()

# Registering the model parameters with the optimizer
optimizer = SGD(model.parameters(), lr=1e-3)

# Defining the training data and loss function
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)
loss_fn = nn.CrossEntropyLoss()
batch_size = 64
max_epochs = 10

# Training
for epoch in range(max_epochs):
    for batch_x, batch_y in DataLoader(training_data, batch_size):
        # Compute prediction and loss
        pred = model(batch_x)
        loss = loss_fn(pred, batch_y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

torch.save(model.state_dict(), "pretrained_model.pth")
```

```python
# Importing the library
import torch
from torch import nn
from torch.optim import SGD
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.pytorch import make_pruning_layer_list, \
                                          Pruner, \
                                          get_model_info

# Defining the neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
model = NeuralNetwork()
# 2. Loading the trained model
model.load_state_dict(torch.load("pretrained_model.pth"))

# Registering the model parameters with the optimizer
optimizer = SGD(model.parameters(), lr=1e-3)

# Defining the training data and loss function
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)
loss_fn = nn.CrossEntropyLoss()
batch_size = 64
max_epochs = 10
# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model, [(1,1,28,28)])
pruner = Pruner(model,
                pruning_layer_list,
                final_pr = 0.7)
print(get_model_info(model, [(1,1,28,28)]))
# Training
for epoch in range(max_epochs):
    for batch_x, batch_y in DataLoader(training_data, batch_size):
        # 4. Updating pruning parameters
        pruner.update()

        # Compute prediction and loss
        pred = model(batch_x)
        loss = loss_fn(pred, batch_y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# 5. Saving the pruned model
if pruner.is_finished:
    torch.save(model.state_dict(), "oneshot_pruned_model.pth")
    torch.onnx.export(model,
                      training_data[0][0].unsqueeze(0),
                      "oneshot_pruned_model.onnx",
                      opset_version = 12)
```

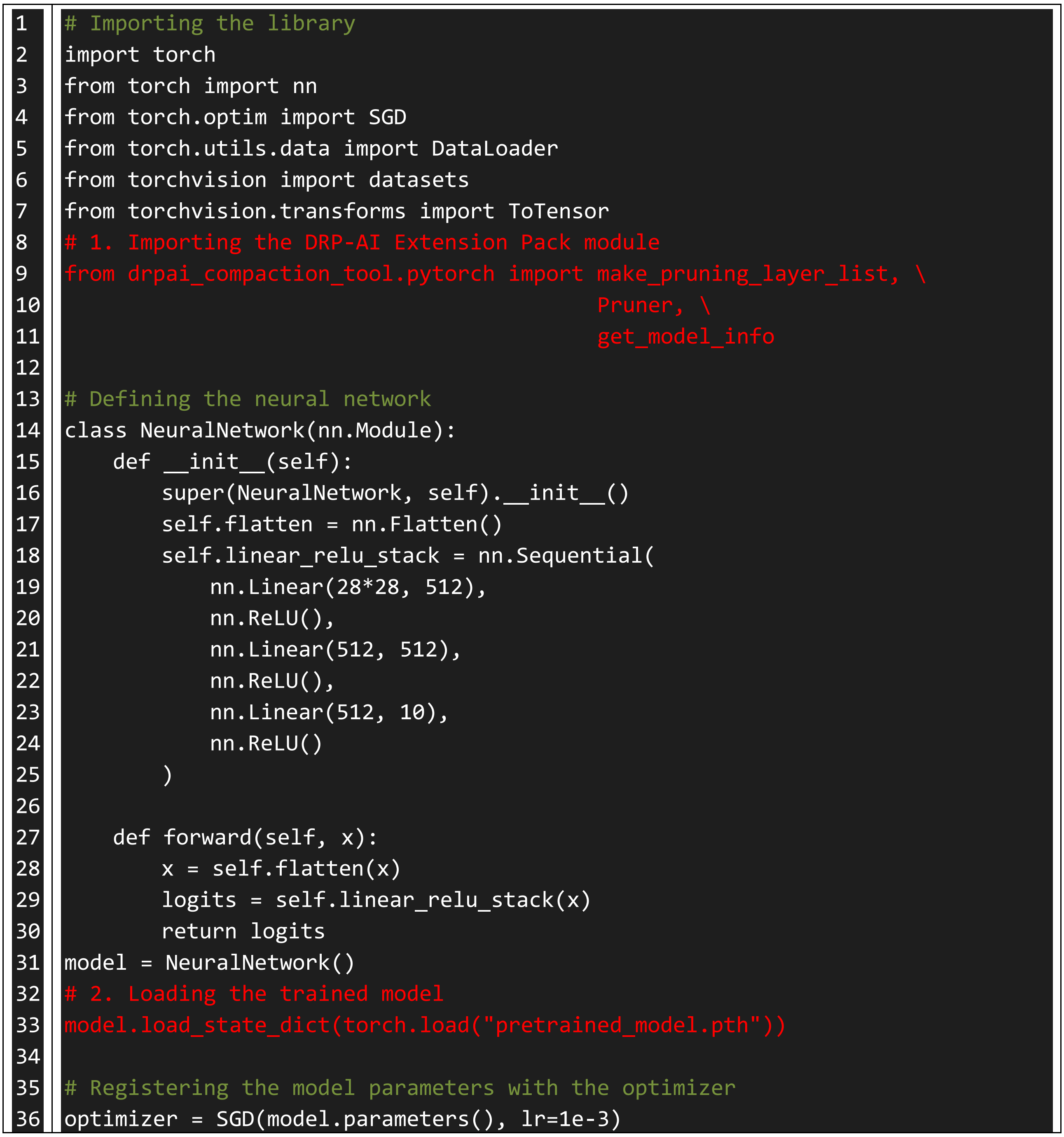
Figure 3-2 How to Add the DRP-AI Extension Pack to the Code for Use in Initial Training (Left: Code for Initial Training; Right: Code for Pruning Then Retraining)

3.2 [PyTorch] Adding the DRP-AI Extension Pack

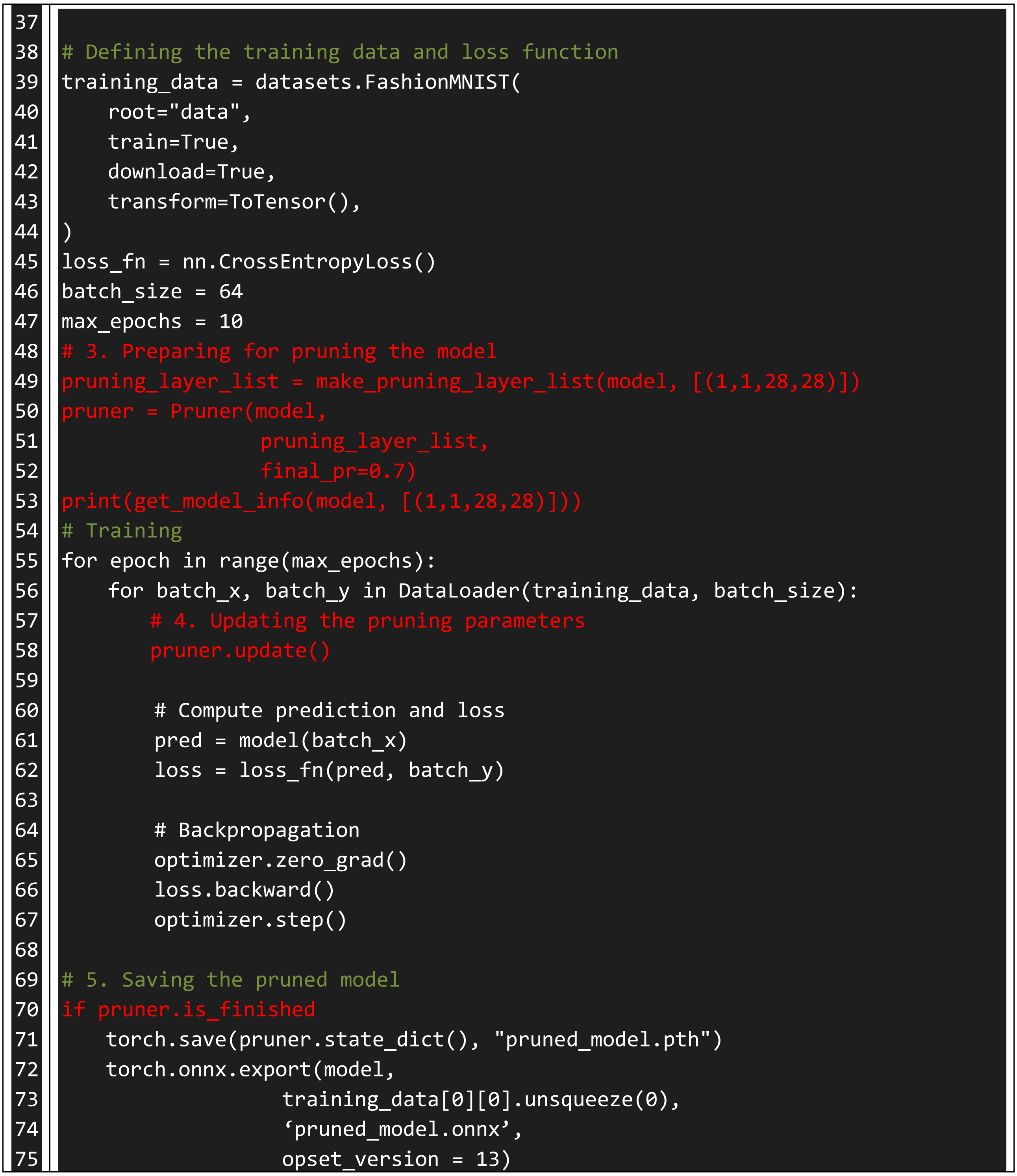
The steps involved in adding the DRP-AI Extension Pack to the code written with PyTorch for use in initial training and then proceeding with pruning and retraining are given below. Implement the five processes listed below in the code for initial training.

1. Importing the DRP-AI Extension Pack module
2. Loading the trained model
3. Preparing for pruning the model
4. Updating the pruning parameters
5. Saving the pruned model

The figure below shows a listing of the code for retraining, which is obtained by adding the DRP-AI Extension Pack to the code written with PyTorch for use in initial training. In this figure, red text indicates the statements added to the code for use in initial training.



```
# Importing the library
import torch
from torch import nn
from torch.optim import SGD
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.pytorch import make_pruning_layer_list, \
                                          Pruner, \
                                          get_model_info

# Defining the neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
model = NeuralNetwork()
# 2. Loading the trained model
model.load_state_dict(torch.load("pretrained_model.pth"))

# Registering the model parameters with the optimizer
optimizer = SGD(model.parameters(), lr=1e-3)
```

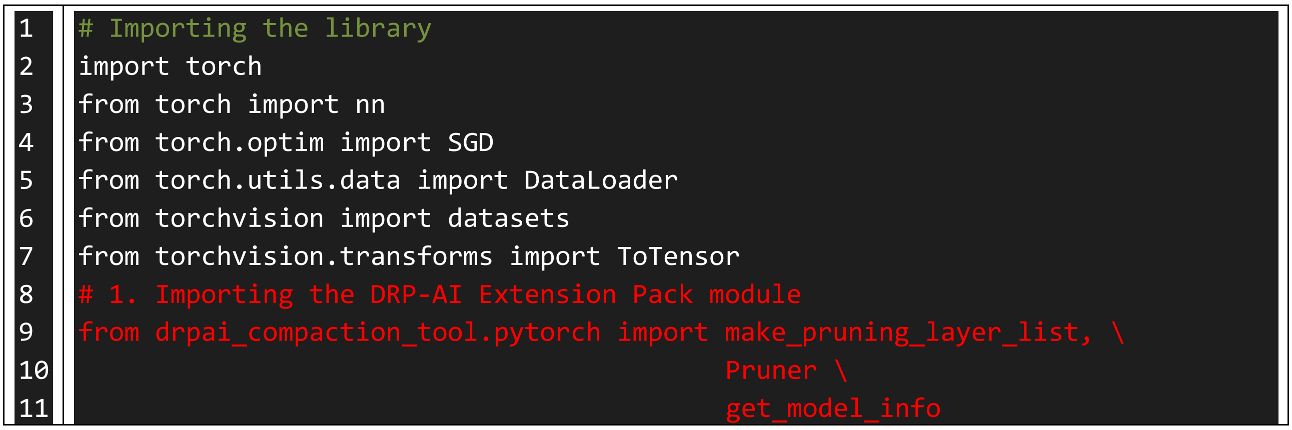


```

# Defining the training data and loss function
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)
loss_fn = nn.CrossEntropyLoss()
batch_size = 64
max_epochs = 10
# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model, [(1,1,28,28)])
pruner = Pruner(model,
                pruning_layer_list,
                final_pr=0.7)
print(get_model_info(model, [(1,1,28,28)]))
# Training
for epoch in range(max_epochs):
    for batch_x, batch_y in DataLoader(training_data, batch_size):
        # 4. Updating the pruning parameters
        pruner.update()

        # Compute prediction and loss
        pred = model(batch_x)
        loss = loss_fn(pred, batch_y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# 5. Saving the pruned model
if pruner.is_finished
    torch.save(pruner.state_dict(), "pruned_model.pth")
    torch.onnx.export(model,
                      training_data[0][0].unsqueeze(0),
                      ‘pruned_model.onnx’,
                      opset_version = 13)
```

Figure 3-3 Training Code for Pruning Then Retraining

3.2.1 [PyTorch] Importing the DRP-AI Extension Pack Module

Import the DRP-AI Extension Pack module to the code written with PyTorch for use in initial training.

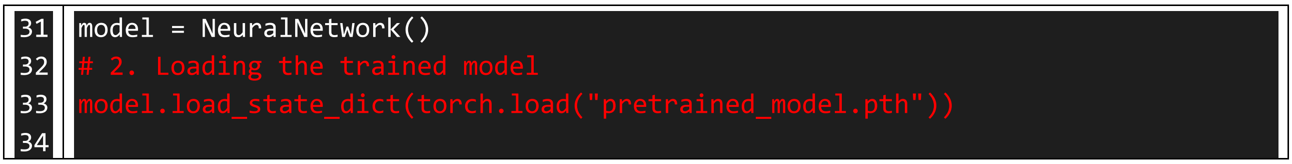


```python
# Importing the library
import torch
from torch import nn
from torch.optim import SGD
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.pytorch import make_pruning_layer_list, \
                                          Pruner \
                                          get_model_info
```

Figure 3-4 Importing the DRP-AI Extension Pack Module

3.2.2 [PyTorch] Loading the Trained Model

Define the model and load the trained model.

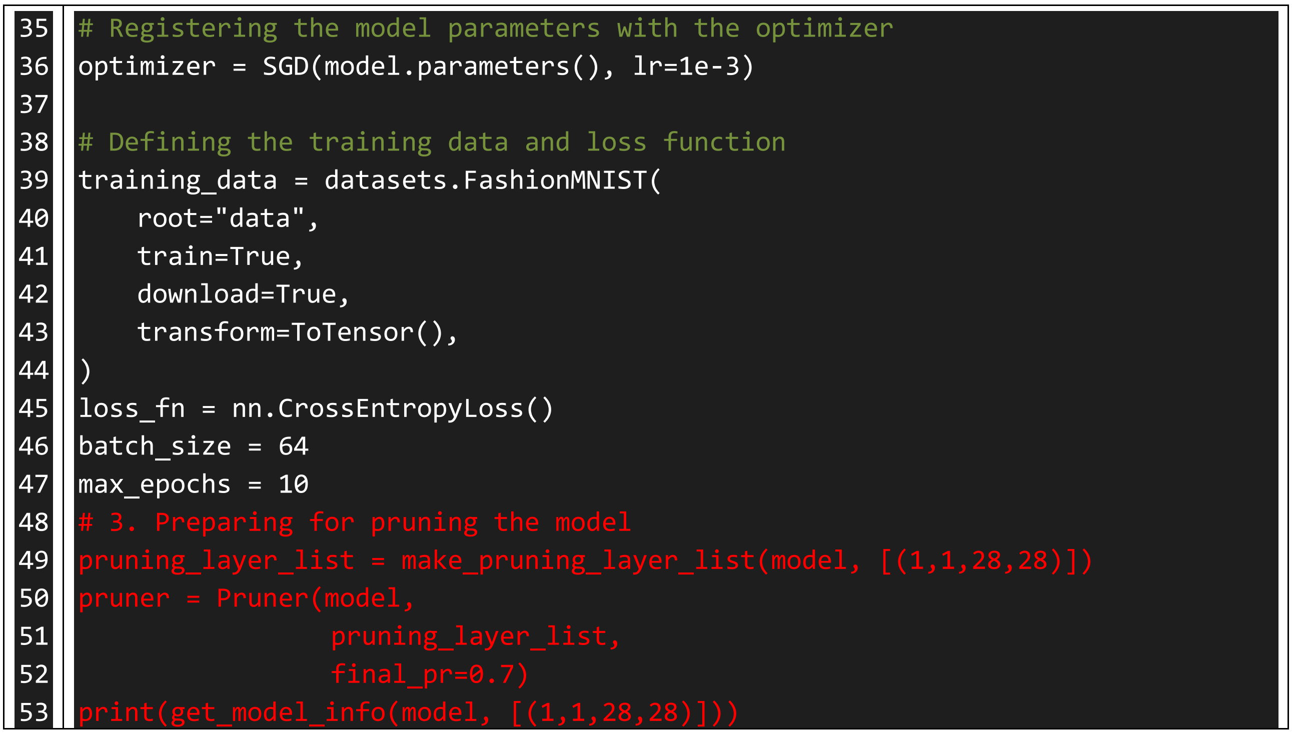


```python
model = NeuralNetwork()
# 2. Loading the trained model
model.load_state_dict(torch.load("pretrained_model.pth"))

```

Figure 3-5 Loading the Trained Model

3.2.3 [PyTorch] Preparing for Pruning the Model

Register the model parameters with the optimizer and then execute the API function for pruning. After the API function for pruning has been executed, confirming that pruning has been performed with the get_model_info() function is recommended.



```
# Registering the model parameters with the optimizer
optimizer = SGD(model.parameters(), lr=1e-3)

# Defining the training data and loss function
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)
loss_fn = nn.CrossEntropyLoss()
batch_size = 64
max_epochs = 10
# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model, [(1,1,28,28)])
pruner = Pruner(model,
                pruning_layer_list,
                final_pr=0.7)
print(get_model_info(model, [(1,1,28,28)]))
```

Figure 3-6 Pruning the Model

Note: Execute the API function for pruning (Pruner) after registering the model parameters with the optimizer.

3.2.4 [PyTorch] Updating the Pruning Parameters

Update the pruning parameters during training. The API function in red text below (pruner.update()) must be called at the start of each iteration.

```
# Training
for epoch in range(max_epochs):
    for batch_x, batch_y in DataLoader(training_data, batch_size):
        # 4. Updating the pruning parameters
        pruner.update()

        # Compute prediction and loss
        pred = model(batch_x)
        loss = loss_fn(pred, batch_y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Figure 3-7 Updating the Pruning Parameters

3.2.5 [PyTorch] Saving the Pruned Model

After having confirmed the completion of pruning, use the PyTorch method to save the pruned model. If the model is to be exported to ONNX, specify 13 for opset_version.

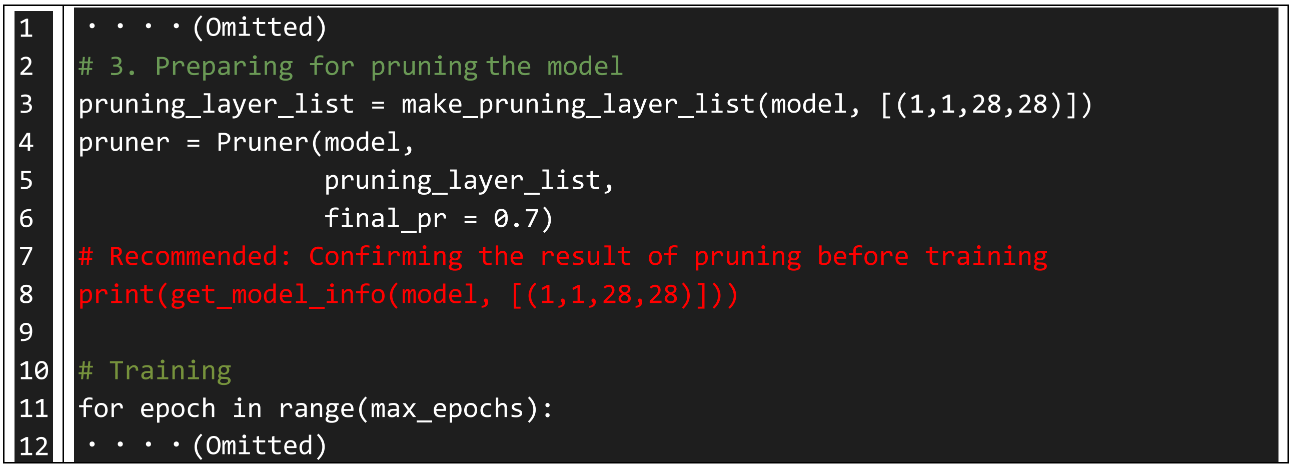
```
# 5. Saving the pruned model
if pruner.is_finished:
    torch.save(pruner.state_dict(), "pruned_model.pth")
    torch.onnx.export(model,
                      training_data[0][0].unsqueeze(0),
                      'pruned_model.onnx',
                      opset_version = 13)
```

Figure 3-8 Saving the Pruned Model

3.3 [PyTorch] Confirming the Result of Pruning

The steps involved in confirming the result of pruning are given below. The function (get_model_info) provided by the DRP-AI Extension Pack can be used to confirm how many parameters were pruned in which layers and the reductions in the number of multiply-and-accumulate calculations. For details on how to use the get_model_info function, see 4.2.5. Calling this function is possible both before and after pruning.

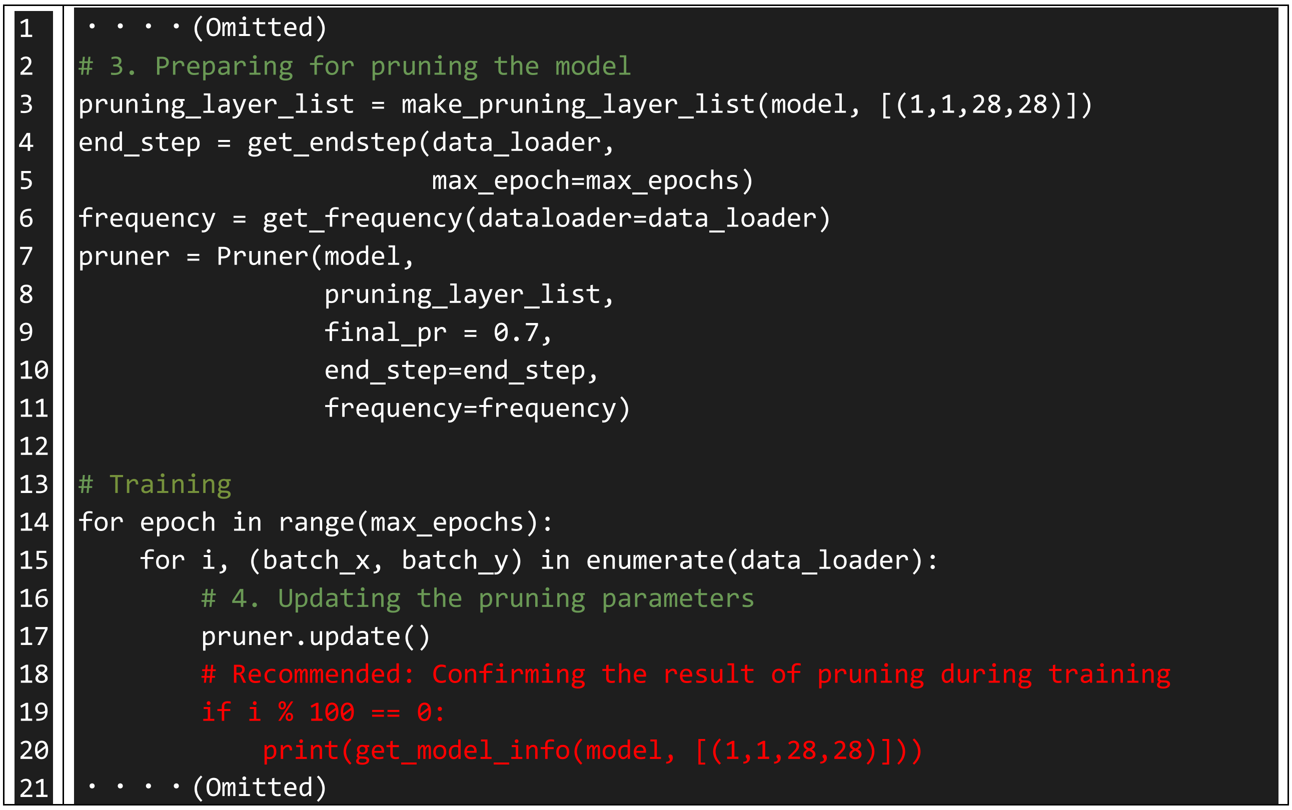
The figure below shows a listing of the sample code in one-shot pruning. In one-shot pruning, the confirmation of pruning being applied before training is recommended.



```
・・・・(Omitted)
# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model, [(1,1,28,28)])
pruner = Pruner(model,
                pruning_layer_list,
                final_pr = 0.7)
# Recommended: Confirming the result of pruning before training
print(get_model_info(model, [(1,1,28,28)]))

# Training
for epoch in range(max_epochs):
・・・・(Omitted)
```

Figure 3-9 Confirming the Result of Pruning: One-Shot Pruning

The figure below shows a listing of the sample code in gradual pruning. In gradual pruning, the confirmation of pruning being applied during training is recommended.



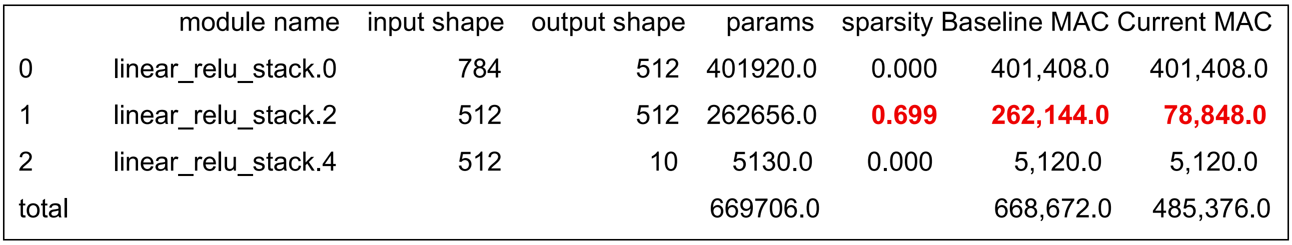
```
・・・・(Omitted)
# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model, [(1,1,28,28)])
end_step = get_endstep(data_loader,
                       max_epoch=max_epochs)
frequency = get_frequency(dataloader=data_loader)
pruner = Pruner(model,
                pruning_layer_list,
                final_pr = 0.7,
                end_step=end_step,
                frequency=frequency)

# Training
for epoch in range(max_epochs):
    for i, (batch_x, batch_y) in enumerate(data_loader):
        # 4. Updating the pruning parameters
        pruner.update()
        # Recommended: Confirming the result of pruning during training
        if i % 100 == 0:
            print(get_model_info(model, [(1,1,28,28)]))
・・・・(Omitted)
```

Figure 3-10 Confirming the Result of Pruning: Gradual Pruning

The figure below shows the result of executing get_model_info. The meanings of the headings in the figure are as follows.

“module name”: Layer name
“input shape”: Input size to a layer
“output shape”: Output size from a layer
“params”: Number of parameters
“sparsity”: Pruning rate
“Baseline MAC”: Number of multiply-and-accumulate calculations before pruning
“Current MAC”: Number of multiply-and-accumulate calculations after pruning

The result of pruning shown below indicates that pruning by about 70% was applied in the "linear_relu_stack.2" layer. It also indicates that pruning reduces the number of multiply-and-accumulate calculations from 262,144 before pruning to 78,848 after pruning.



	module name	input shape	output shape	params	sparsity	Baseline MAC	Current MAC
0	linear_relu_stack.0	784	512	401920.0	0.000	401,408.0	401,408.0
1	linear_relu_stack.2	512	512	262656.0	**0.699**	**262,144.0**	**78,848.0**
2	linear_relu_stack.4	512	10	5130.0	0.000	5,120.0	5,120.0
total				669706.0		668,672.0	485,376.0

Figure 3-11 Result of Executing get_model_info

3.4 [PyTorch] Training or Inference with a Saved Pruned Model

The steps involved in loading a saved pruned model are given below. Refer to this section when continuing to train a saved pruned model or performing inference with a pruned model. There are 2 methods to load the saved pruned model. The method by using load_pruned_state_dict() is recommended because it is easy to use.

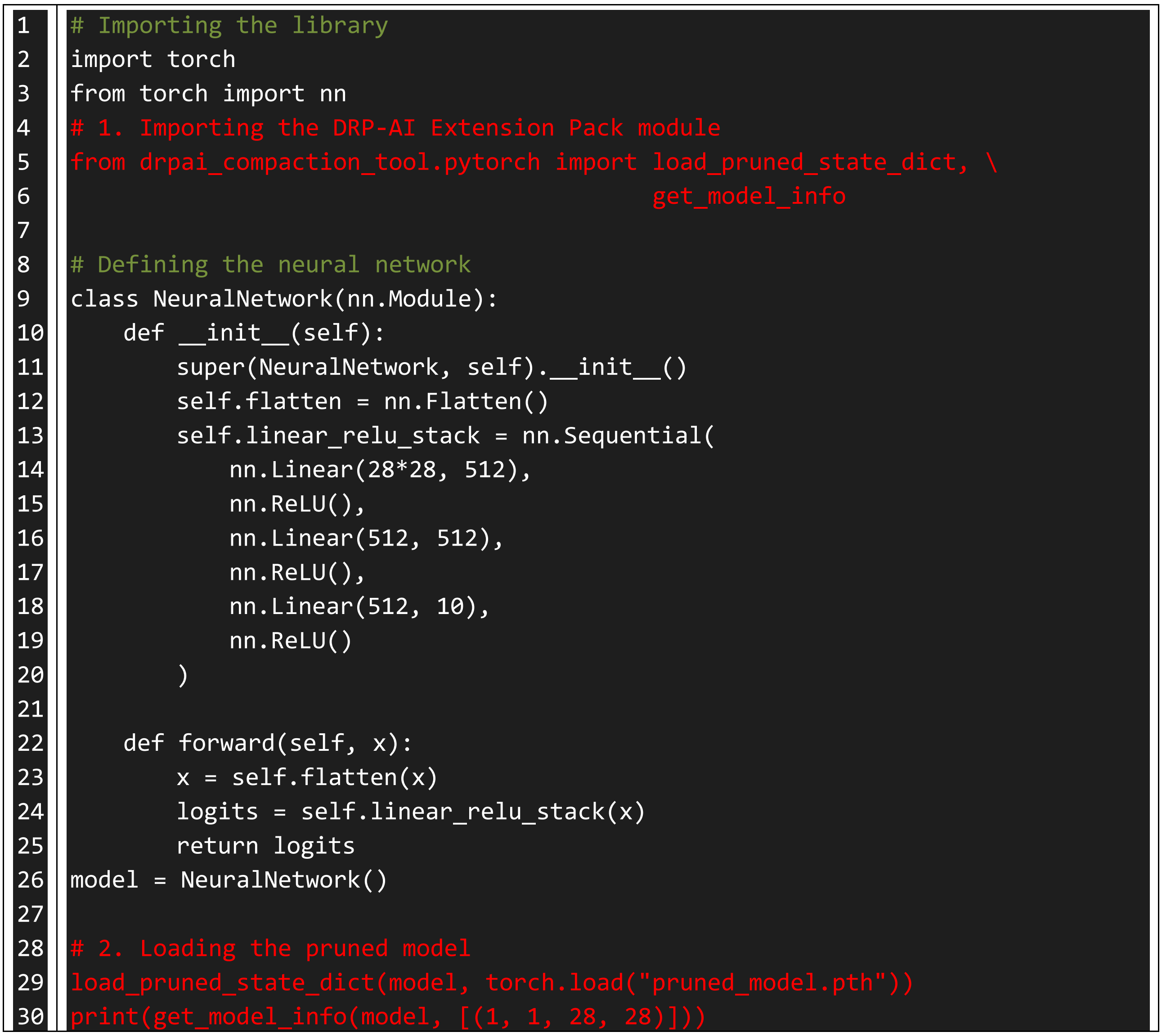
3.4.1 [Recommend] How to load the pruned model with load_pruned_state_dict()

Load the saved pruned model through the steps listed below.

1. Importing the DRP-AI Extension Pack module
2. Loading the pruned model

Calling the get_model_info function to check the pruning rate after loading of the pruned model is recommended. For details on how to use the get_model_info function, see 4.2.5.

The figure below shows a list of the sample code.



```python
# Importing the library
import torch
from torch import nn
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.pytorch import load_pruned_state_dict, \
                                           get_model_info

# Defining the neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
model = NeuralNetwork()

# 2. Loading the pruned model
load_pruned_state_dict(model, torch.load("pruned_model.pth"))
print(get_model_info(model, [(1, 1, 28, 28)]))
```

Figure 3-12 [PyTorch] Loading the Pruned Model (Method1)

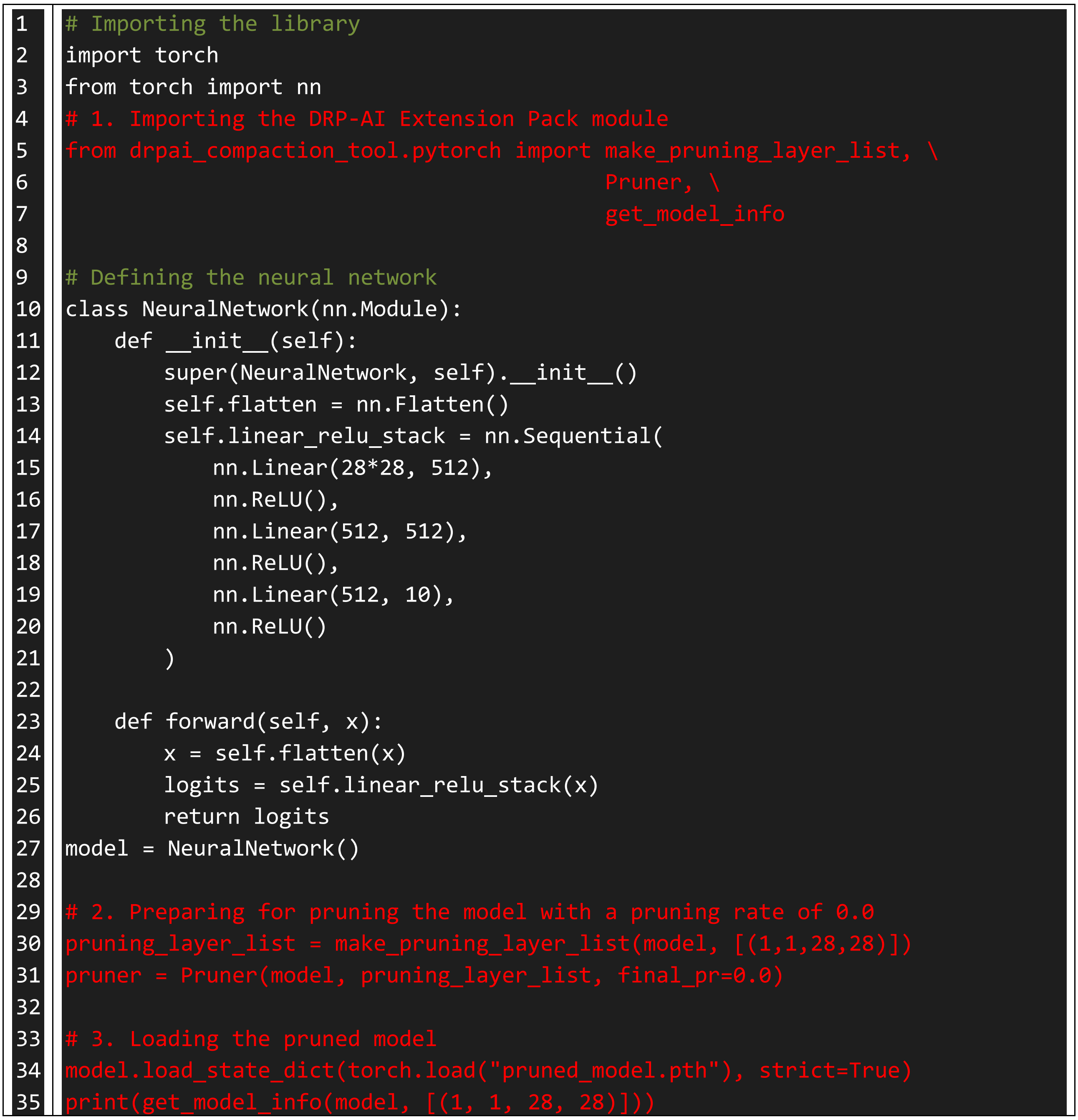
3.4.2 How to load the pruned model with make_pruning_layer_list() and Pruner()

Loads the saved pruned model through the steps listed below.

1. Importing the DRP-AI Extension Pack module
2. Preparing for pruning the model with a pruning rate of 0.0
3. Loading the pruned model

Calling the get_model_info function to check the pruning rate after loading of the pruned model is recommended. For details on how to use the get_model_info function, see 4.2.5.

The figure below shows a list of the sample code.



```python
# Importing the library
import torch
from torch import nn
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.pytorch import make_pruning_layer_list, \
                                          Pruner, \
                                          get_model_info

# Defining the neural network
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
model = NeuralNetwork()

# 2. Preparing for pruning the model with a pruning rate of 0.0
pruning_layer_list = make_pruning_layer_list(model, [(1,1,28,28)])
pruner = Pruner(model, pruning_layer_list, final_pr=0.0)

# 3. Loading the pruned model
model.load_state_dict(torch.load("pruned_model.pth"), strict=True)
print(get_model_info(model, [(1, 1, 28, 28)]))
```

Figure 3-13 [PyTorch] Loading the Pruned Model (Method2)

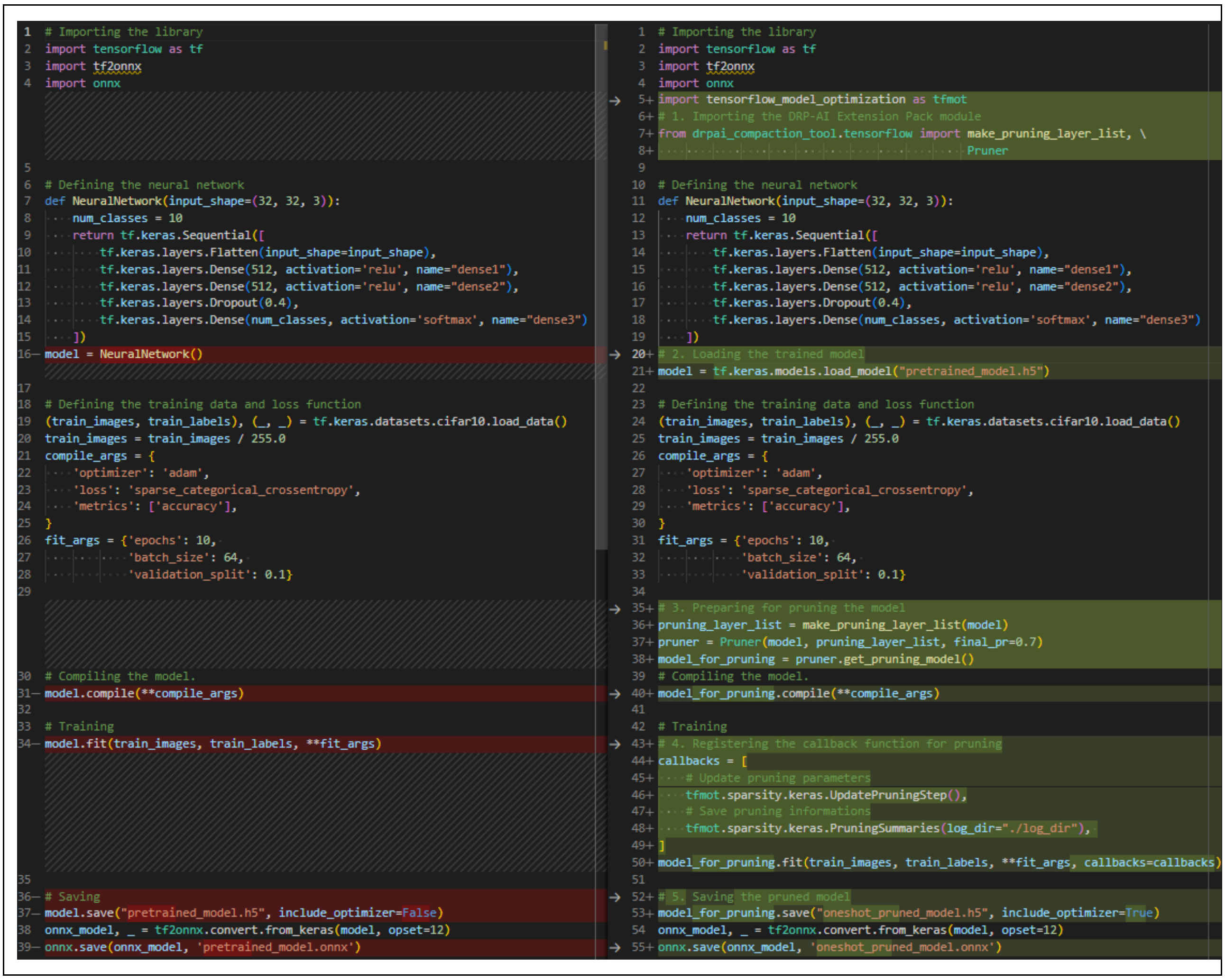
Note: When loading the weights with load_state_dict() function, set strict argument to “True”. When this argument is set to “False”, weights may not be loaded correctly.

3.5 [TensorFlow] Adding the DRP-AI Extension Pack

The steps involved in adding the DRP-AI Extension Pack to the code written with TensorFlow for use in initial training and then proceeding with pruning and retraining are given below. Implement the five processes listed below in the code for initial training.

1. Importing the DRP-AI Extension Pack module
2. Loading the trained model
3. Preparing for pruning the model
4. Registering the callback function for pruning
5. Saving the pruned model

Figure 3-14 consists of listings of the code written with TensorFlow for use in initial training without and with addition of the DRP-AI Extension Pack. The code in the left column is that for initial training and the code in the right column is that for pruning then retraining. The green shading indicates the differences between the two listings, that is, the several lines that are added to make the DRP-AI Extension Pack usable.



```
# Importing the library
import tensorflow as tf
import tf2onnx
import onnx

# Defining the neural network
def NeuralNetwork(input_shape=(32, 32, 3)):
    num_classes = 10
    return tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=input_shape),
        tf.keras.layers.Dense(512, activation='relu', name="dense1"),
        tf.keras.layers.Dense(512, activation='relu', name="dense2"),
        tf.keras.layers.Dropout(0.4),
        tf.keras.layers.Dense(num_classes, activation='softmax', name="dense3")
    ])
model = NeuralNetwork()

# Defining the training data and loss function
(train_images, train_labels), (_, _) = tf.keras.datasets.cifar10.load_data()
train_images = train_images / 255.0
compile_args = {
    'optimizer': 'adam',
    'loss': 'sparse_categorical_crossentropy',
    'metrics': ['accuracy'],
}
fit_args = {'epochs': 10,
            'batch_size': 64,
            'validation_split': 0.1}

# Compiling the model.
model.compile(**compile_args)

# Training
model.fit(train_images, train_labels, **fit_args)

# Saving
model.save("pretrained_model.h5", include_optimizer=False)
onnx_model, _ = tf2onnx.convert.from_keras(model, opset=12)
onnx.save(onnx_model, 'pretrained_model.onnx')
```

```
# Importing the library
import tensorflow as tf
import tf2onnx
import onnx
import tensorflow_model_optimization as tfmot
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.tensorflow import make_pruning_layer_list, \
                                            Pruner

# Defining the neural network
def NeuralNetwork(input_shape=(32, 32, 3)):
    num_classes = 10
    return tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=input_shape),
        tf.keras.layers.Dense(512, activation='relu', name="dense1"),
        tf.keras.layers.Dense(512, activation='relu', name="dense2"),
        tf.keras.layers.Dropout(0.4),
        tf.keras.layers.Dense(num_classes, activation='softmax', name="dense3")
    ])
# 2. Loading the trained model
model = tf.keras.models.load_model("pretrained_model.h5")

# Defining the training data and loss function
(train_images, train_labels), (_, _) = tf.keras.datasets.cifar10.load_data()
train_images = train_images / 255.0
compile_args = {
    'optimizer': 'adam',
    'loss': 'sparse_categorical_crossentropy',
    'metrics': ['accuracy'],
}
fit_args = {'epochs': 10,
            'batch_size': 64,
            'validation_split': 0.1}

# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model)
pruner = Pruner(model, pruning_layer_list, final_pr=0.7)
model_for_pruning = pruner.get_pruning_model()
# Compiling the model.
model_for_pruning.compile(**compile_args)

# Training
# 4. Registering the callback function for pruning
callbacks = [
    # Update pruning parameters
    tfmot.sparsity.keras.UpdatePruningStep(),
    # Save pruning informations
    tfmot.sparsity.keras.PruningSummaries(log_dir="./log_dir"),
]
model_for_pruning.fit(train_images, train_labels, **fit_args, callbacks=callbacks)

# 5. Saving the pruned model
model_for_pruning.save("oneshot_pruned_model.h5", include_optimizer=True)
onnx_model, _ = tf2onnx.convert.from_keras(model, opset=12)
onnx.save(onnx_model, 'oneshot_pruned_model.onnx')
```

Figure 3-14 How to Add the DRP-AI Extension Pack to the Code for Use in Initial Training (Left: Code for Initial Training; Right: Code for Pruning Then Retraining)

3.5.1 [TensorFlow] Importing the DRP-AI Extension Pack Module

Import the DRP-AI Extension Pack module to the code written with TensorFlow for use in initial training.

```python
# Importing the library
import tensorflow as tf
import tf2onnx
import onnx
import tensorflow_model_optimization as tfmot
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.tensorflow import make_pruning_layer_list, \
                                             Pruner
```

Figure 3-15 Importing the DRP-AI Extension Pack Module

Note: Import tensorflow_model_optimization.

3.5.2 [TensorFlow] Loading the Trained Model

Load the trained model according to the usage method of TensorFlow.

```python
# 2. Loading the trained model
model = tf.keras.models.load_model("pretrained_model.h5")
```

Figure 3-16 Loading the Trained Model

3.5.3 [TensorFlow] Preparing for Pruning the Model

After having executed the API function for pruning, obtain the model to which pruning is to be applied by using the get_pruning_model() function. After that, execute compilation of the model.

```python
# 3. Preparing for pruning the model
pruning_layer_list = make_pruning_layer_list(model)
pruner = Pruner(model, pruning_layer_list, final_pr=0.7)
model_for_pruning = pruner.get_pruning_model()
# Compiling the model.
model_for_pruning.compile(**compile_args)
```

Figure 3-17 Preparing for Pruning the Model

3.5.4 [TensorFlow] Registering the Callback Function for Pruning

Register the callback function for pruning when carrying out training. For details on UpdatePruningStep() and PruningSummaries(), see 3.6 and the official documents of TensorFlow.

```
# Training
# 4. Registering the callback function for pruning
callbacks = [
    # Update pruning parameters
    tfmot.sparsity.keras.UpdatePruningStep(),
    # Save pruning informations
    tfmot.sparsity.keras.PruningSummaries(log_dir="./log_dir"),
]
model_for_pruning.fit(train_images,
                      train_labels,
                      **fit_args,
                      callbacks=callbacks)
```

Figure 3-18 Registering the Callback Function for Pruning

Note: Only executing the step of preparing for pruning a model, which was described in 3.5.3, does not lead to actual pruning of the model. Make sure to always execute that step in combination with the callback function.

3.5.5 [TensorFlow] Saving the Pruned Model

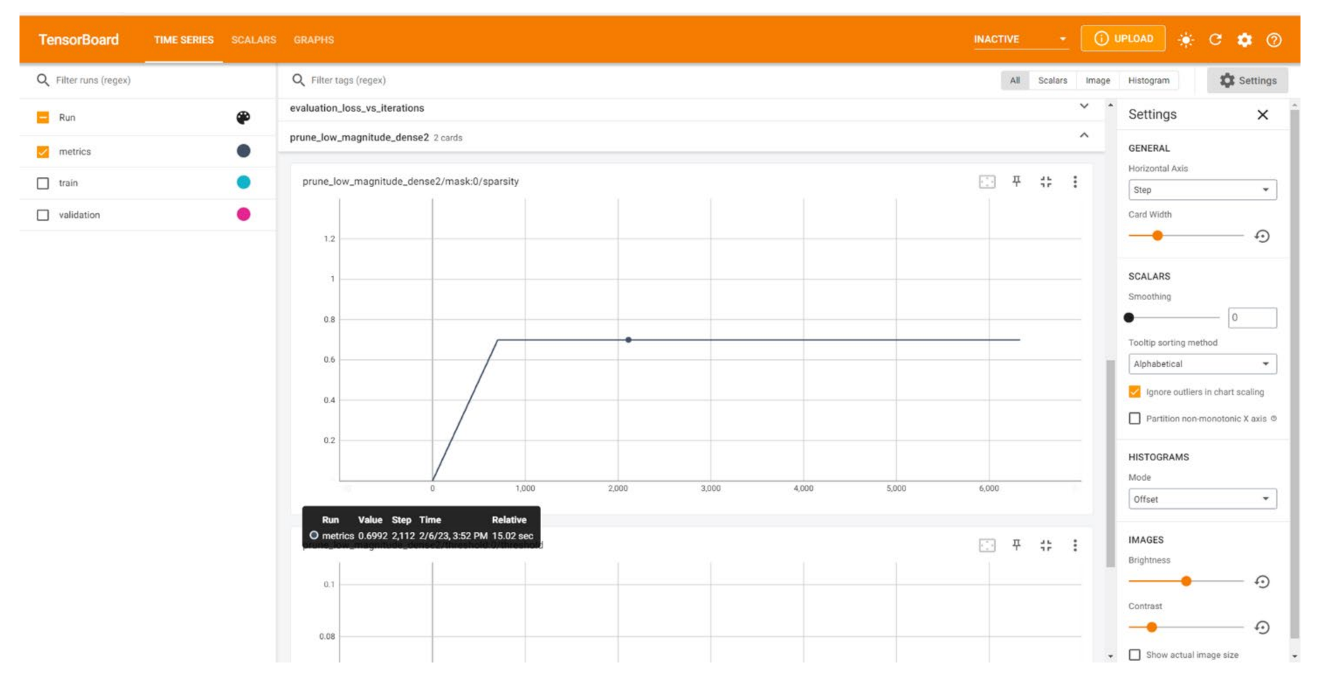
Save the model according to the usage method of TensorFlow. If the model is to be exported to ONNX, specify 13 for opset.

```
# 4. Saving the pruned model
model_for_pruning.save("one-shot_pruned_model.h5", include_optimizer=True)
onnx_model, _ = tf2onnx.convert.from_keras(model, opset=13)
onnx.save(onnx_model, 'one-shot_pruned_model.onnx')
```

Figure 3-19 Saving the Pruned Model

3.6 [TensorFlow] Confirming the Result of Pruning

The steps involved in confirming the result of pruning are given below. The callback function (PruningSummaries()) provided by TensorFlow can be used to obtain the result of how many parameters were pruned in which layers. TensorBoard provided by TensorFlow can be used to display the obtained information in a way that allows confirming the result as shown below.



☐ Run
☑ metrics
☐ train
☐ validation

Run	Value	Step	Time	Relative
○ metrics	0.6992	2,112	2/6/23, 3:52 PM	15.02 sec

Settings

GENERAL

Horizontal Axis: Step

Card Width

SCALARS

Smoothing: 0

Tooltip sorting method: Alphabetical

☑ Ignore outliers in chart scaling

☐ Partition non-monotonic X axis

HISTOGRAMS

Mode: Offset

IMAGES

Brightness

Contrast

☐ Show actual image size

Figure 3-20 Using TensorBoard to Confirm the Pruning Rate

The changes in the pruning rate of the "prune_low_magnitude_dense2" layer are shown in the above figure. The horizontal axis indicates the number of steps (iterations) and the vertical axis indicates the pruning rate.

For example, PruningSummaries() may be set as follows:

```
tfmot.sparsity.keras.PruningSummaries(log_dir="./logdir")
```

The result of pruning can be confirmed by starting up TensorBoard as follows:

```
$ tensorboard –logdir ./logdir
```

For details, see the official documents of TensorFlow.

3.7 [TensorFlow] Training or Inference with a Saved Pruned Model

The steps involved in loading a saved pruned model are given below. Refer to this section when continuing to train a saved pruned model or performing inference with a pruned model.

Load the saved pruned model through the steps listed below.

1. Importing the DRP-AI Extension Pack module
2. Preparing for pruning the model with a pruning rate of 0.0
3. Loading the pruned model

The figure below shows a listing of the sample code.

```
# Importing the library
import tensorflow as tf
import tensorflow_model_optimization as tfmot
# 1. Importing the DRP-AI Extension Pack module
from drpai_compaction_tool.tensorflow import make_pruning_layer_list, \
                                             Pruner

def print_sparsity(model):
    import numpy as np
    from tensorflow_model_optimization.python.core.sparsity.keras \
                                               import pruning_wrapper

    layer_info = {}
    for layer in model.layers:
        if not isinstance(layer, pruning_wrapper.PruneLowMagnitude):
            continue
        for weight, mask, threshold in layer.pruning_vars:
            np_mask = tf.keras.backend.get_value(mask)
            sparsity = 1.0 - np.count_nonzero(np_mask) / float(np_mask.size)
            layer_info[layer.name] = sparsity

    max_len = len(max(layer_info.keys(), key=lambda name: len(name)))
    for name, sparsity in layer_info.items():
        print(f'{name:{max_len+1}s} |  {sparsity:0.2f}')

# Defining the neural network
def NeuralNetwork(input_shape=(32, 32, 3)):
    num_classes = 10
    return tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=input_shape),
        tf.keras.layers.Dense(512, activation='relu', name="dense1"),
        tf.keras.layers.Dense(512, activation='relu', name="dense2"),
        tf.keras.layers.Dropout(0.4),
        tf.keras.layers.Dense(num_classes,
                                activation='softmax', name="dense3")
    ])
model = NeuralNetwork()

# 2. Preparing for pruning the model with a pruning rate of 0.0
pruning_layer_list = make_pruning_layer_list(model)
```

```
pruner = Pruner(model, pruning_layer_list, final_pr=0.0)
model_for_pruning = pruner.get_pruning_model()
print_sparsity(model_for_pruning)

# 3. Loading the pruned model
model_for_pruning.load_weights("pruned_model.h5")
print_sparsity(model_for_pruning)
```

Figure 3-21 [TensorFlow] Loading the Pruned Model

3.8 Sample Code

This subsection describes how to execute the sample code and gives an outline of its operation. The accuracy of a model after pruning can be confirmed with the use of the sample code.

3.8.1 classification/pytorch_mobilenetv2

This sample code employs the MobileNetV2 architecture of PyTorch and is for use in initial training and pruning then retraining. The code is for use with the CIFAR-10 dataset. The following three files are provided. The method for adding the DRP-AI Extension Pack module can be confirmed by comparing train.py, code for use in initial training with the files with names of the form retrain*.py, containing the two variants of the code for use in pruning then retraining.

Table 3.1 List of Provided Files

File Name	Description
train.py	MobileNetV2 sample code for initial training
retrain_with_oneshot_pruning.py	MobileNetV2 sample code for pruning then retraining (one-shot pruning)
retrain_with_gradual_pruning.py	MobileNetV2 sample code for pruning then retraining (gradual pruning)

Pruning then retraining with the MobileNetV2 architecture can be performed by executing the following two steps.

```
# Step 1: Initial training
$ python3 train.py
# Step 2: Pruning then retraining
# In one-shot pruning
$ python3 retrain_with_oneshot_pruning.py
# In gradual pruning
$ python3 retrain_with_gradual_pruning.py
```

Figure 3-22 Executing the MobileNetV2 Sample Code for Initial Training and Pruning Then Retraining

After executing the sample code, the files listed in the table below will have been output.

Table 3.2 List of Output Files

File Name	Description
pretrained_mobilenetv2.pth	Trained model file (pth format)
pretrained_mobilenetv2.onnx	Trained model file (ONNX format)
oneshot_pruned_mobilenetv2.pth	Model file after pruning in one-shot pruning mode then retraining (pth format)
oneshot_pruned_mobilenetv2.onnx	Model file after pruning in one-shot pruning mode then retraining (ONNX format)
gradual_pruned_mobilenetv2.pth	Model file after pruning in gradual pruning mode then retraining (pth format)
gradual_pruned_mobilenetv2.onnx	Model file after pruning in gradual pruning mode then retraining (ONNX format)

Command-line options are listed in the table below.

Table 3.3 List of Options of the MobileNetV2 Sample Code for Initial Training and Pruning Then Retraining

Option Argument	Description
-h, --help	Outputs a help message. Example: $ python3 train.py -h
--lr LR	Sets the learning rate. Set a small learning rate for a case where the loss varies greatly. Set a large learning rate for a case where the loss does not vary. Example: $ python3 train.py --lr 0.2
--max_epochs MAX_EPOCHS	Specifies the maximum number of epochs. If you want a greater accuracy, set a value greater than the default so that learning proceeds for a longer time. Example: $ python3 train.py --max_epochs 3
--pretrained_weight	Specifies the name of a file (.pth format) for a model for initial training. Note: Can only be set for code for pruning then retraining. Example: $ python3 retrain_with_oneshot_pruning.py \\ –pretrained_weight ./pretrained_mobilenetv2.pth
--pruning_rate	Specifies the pruning rate. Note: Can only be set for code for pruning then retraining. Example: $ python3 retrain_with_oneshot_pruning.py \\ –pruning_rate 0.7

3.8.2 classification/tensorflow_cnn

This sample code employs the CNN model of TensorFlow and is for use in initial training and pruning then retraining. The code is for use with the CIFAR-10 dataset. The following three files are provided. The method for adding the DRP-AI Extension Pack module can be confirmed by comparing train.py, code for use in initial training with the files with names of the form retrain*.py, containing the two variants of the code for use in pruning then retraining.

Table 3.4 List of Provided Files

File Name	Description
train.py	CNN sample code for initial training
retrain_with_oneshot_pruning.py	CNN sample code for pruning then retraining (one-shot pruning)
retrain_with_gradual_pruning.py	CNN sample code for pruning then retraining (gradual pruning)

Pruning then retraining with the CNN model can be performed by executing the following two steps.

```
# Step 1: Initial training
$ python3 train.py
# Step 2: Pruning then retraining
# In one-shot pruning
$ python3 retrain_with_oneshot_pruning.py
# In gradual pruning
$ python3 retrain_with_gradual_pruning.py
```

Figure 3-23 Executing the CNN Sample Code for Initial Training and Pruning Then Retraining

After executing the sample code, the files listed in the table below will have been output.

Table 3.5 List of Output Files

File Name	Description
pretrained_cnn.h5	Trained model file (h5 format)
pretrained_cnn.onnx	Trained model file (ONNX format)
oneshot_pruned_cnn.h5	Model file after pruning in one-shot pruning mode then retraining (h5 format)
oneshot_pruned_cnn.onnx	Model file after pruning in one-shot pruning mode then retraining (ONNX format)
gradual_pruned_cnn.h5	Model file after pruning in gradual pruning mode then retraining (h5 format)
gradual_pruned_cnn.onnx	Model file after pruning in gradual pruning mode then retraining (ONNX format)

Command-line options are listed in the table below.

Table 3.6 List of Options of the CNN Sample Code for Initial Training and Pruning Then Retraining

Option Argument	Description
-h, --help	Outputs a help message. Example: $ python3 train.py -h
--lr LR	Sets the learning rate. Set a small learning rate for a case where the loss varies greatly. Set a large learning rate for a case where the loss does not vary. Example: $ python3 train.py --lr 0.2
--max_epochs MAX_EPOCHS	Specifies the maximum number of epochs. If you want a greater accuracy, set a value greater than the default so that learning proceeds for a longer time. Example: $ python3 train.py --max_epochs 3
--pretrained_weight	Specifies the name of a file (.h5 format) of a model for initial training. Note: Can only be set for code for pruning then retraining. Example: $ python3 retrain_with_oneshot_pruning.py \\ –pretrained_weight ./pretrained_cnn.h5
--pruning_rate	Specifies the pruning rate. Note: Can only be set for code for pruning then retraining. Example: $ python3 retrain_with_oneshot_pruning.py \\ –pruning_rate 0.7

4. Details on the DRP-AI Extension Pack API

This section describes the API functions and class provided by the DRP-AI Extension Pack.

4.1 List of DRP-AI Extension Pack API Functions and Class

The API functions and class provided by the DRP-AI Extension Pack are listed in the table below.

Table 4.1 List of DRP-AI Extension Pack API Functions and Class

Module	Function/Class Name	Description	Section
drpai_compaction_tool.pytorch	make_pruning_layer_list	Sets layers to which pruning is not to be applied and creates the list of the target layers for pruning.	4.2.1
	Pruner	Applies pruning to the model.	4.2.2
	get_endstep	Gets the step at which pruning ends.	4.2.3
	get_frequency	Gets the frequency for updating of the pruning rate.	4.2.4
	get_model_info	Gets a list of information on the model, such as the numbers of parameters.	4.2.5
	deepcopy_model	Deep copy the model. (Deep copy means copies that are completely reproduced.)	4.2.6
	load_pruned_state_dict	Loads the model which weights is pruned	4.2.7
	RewriterContext	Changes the format of the pruned weights in the context.	4.2.8
drpai_compaction_tool.tensorflow	make_pruning_layer_list	Sets layers to which pruning is not to be applied and creates the list of the target layers for pruning.	4.3.1
	Pruner	Applies pruning to the model.	4.3.2
	get_endstep	Gets the step at which pruning ends.	4.3.3
	get_frequency	Gets the frequency for updating of the pruning rate.	4.3.4

The API functions and class are described in terms of the following items on the following pages.

[Overview]	Describes the API class or function in outline.
[Function/class name]	Function name or class name
[Calling format]	Describes the format for calling the function or the class as a function.
[Argument]	Describes the arguments.
[Returns]	Describes the return value.
[Feature]	Describes the function of the API.
[Remarks]	Describes points to note.

4.2 [PyTorch]

4.2.1 make_pruning_layer_list

[Overview]	Sets layers to which pruning is not to be applied and creates the list of the target layers for pruning.	
[Function/class name]	make_pruning_layer_list	
[Calling format]	make_pruning_layer_list(model: torch.nn.Module, input_size: List[Tuple[int]], input_data: Union[List[Any], Mapping[Any]], prune_last: bool = False, prune_dwise: bool = False) -> List[str]	
[Argument]	model: torch.nn.Module	PyTorch model
	input_size: List[Tuple[int]]	Shape of input data Default: None Set either input_size or input_data. Input values in the order of [batch size, number of channels, height, width]. A value of at least 2 should be set as the batch size when using batch normalization.
	input_data: Union[List[Any], Mapping[Any]]	Input data Default: None Set either input_size or input_data.
	prune_last: bool	When this argument is set to “true”, the last layer is included among the targets for pruning. Default: False
	prune_dwise: bool	When this argument is set to “true”, the depthwise convolution layer is included among the targets for pruning. Default: False
[Returns]	pruning_layer_list: List[str]	List of the target layers for pruning
[Feature]	Creates the list of the target layers for pruning based on the input model. Pruning is to be applied to the layers defined with torch.nn.Conv2d or torch.nn.Linear.	

[Remarks]	Pruning cannot be applied to the first layer because doing so significantly worsens the accuracy of the model.
	Pruning also cannot be applied to a layer for which the number of input channels is not a multiple of 32.
	Pruning is not applied to the last layer or depthwise convolution layer by default because doing so significantly worsens the accuracy of the model.
	For details on torch.nn.Module, torch.nn.Conv2d, and torch.nn.Linear, see the official documents of PyTorch.
	Usage example 1:
	>>>import torchvision
	>>> model = torchvision.models.resnet18(num_classes=1000)
	# Set a value of at least 2 as the batch size because batch normalization is used in ResNet18.
	>>> make_pruning_layer_list(model, \\
	input_size=[(2,3,224,224)])
	Usage example 2:
	>>>import torchvision
	>>> model = torchvision.models.detection.ssd300_vgg16()
	# Inputting in the list format
	>>> make_pruning_layer_list(model, \\
	input_data=[[torch.rand(3, 300, 300)], \\
	[{'boxes': torch.tensor([[0, 0, 100, 100]]),\\
	'labels': torch.tensor([0])}]])
	# Inputting in the dict format
	>>> make_pruning_layer_list(model,
	input_data={
	"images": [torch.rand(3, 300, 300)], \\
	"targets": [{'boxes': torch.tensor([[0, 0, 100, 100]]),\\
	'labels': torch.tensor([0])}] \\
	} \\
	)

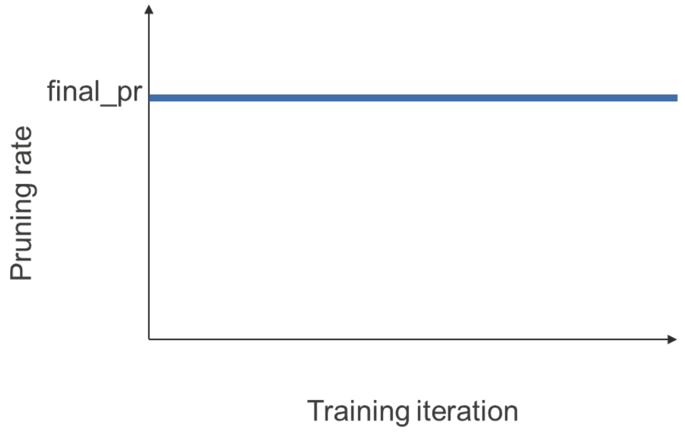
4.2.2 Pruner

[Overview]	Controls the pruning parameters.	
[Function/class name]	Pruner	
[Calling format]	class Pruner(model: torch.nn.Module, pruning_layer_list: List[str], initial_pr: float, final_pr: float, begin_step: int, end_step: int, frequency: int,)	
[Argument]	model: torch.nn.Module	PyTorch model
	pruning_layer_list: List[str]	List of the target layers for pruning
	initial_pr: float	Initial value of pruning rate Default: 0.01 Input range: $0 \leq \mathit{initial\_pr} < 1$
	final_pr: float	Final value of pruning rate Default: 0.7 Input range: $0 \leq \mathit{final\_pr} < 1$
	begin_step: int	Number of the step (iteration) where pruning starts Default: 0 Input range: $0 \leq \mathit{begin\_step}$
	end_step: int	Number of the step (iteration) where pruning ends Default: -1 Input range: $-1 \leq \mathit{end\_step}$
	frequency: int	Frequency for executing pruning (number of iterations) Default: 100 Input range: $0 < \mathit{frequency}$
[Returns]	pruner: object	Object for setting up pruning
[Feature]	The model is only pruned once when end_step is -1. (Any settings of initial_pr, begin_step, and frequency will be ignored in this case.) When a value other than -1 is set, gradual pruning is applied to the model. For details, refer to [Remarks] below.	

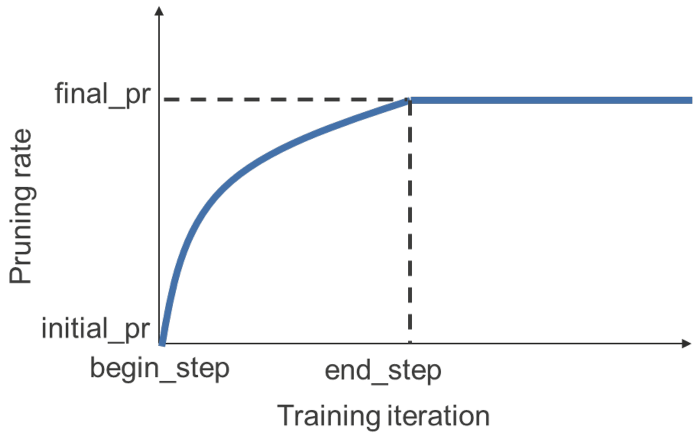
<table>
<tr>
<td>[Remarks]</td>
<td>
Call this API function after registering the parameters of the model to the optimizer.

The setting of begin_step = end_step is prohibited.
When end_step = -1, the settings of initial_pr, begin_step, and frequency are ignored.
Pruning is carried out over the number of iterations set by [begin_step, end_step]. To complete pruning, training needs to have been performed for the number of iterations represented by (end_step – begin_step + 1).
initial_pr and final_pr must be values in the range [0.0, 1.0).

The use of the default values for initial_pr and begin_step is recommended.
For end_step, setting a value around 70% of the total number of iterations in training is recommended. (For example, when the total number of iterations was 100, set 70 iterations.) Note that get_endstep() can be used to set the value.
For frequency, setting the total number of iterations per epoch is recommended. Note that get_frequency() can be used to set the value.
For details on torch.nn.Module, see the official documents of PyTorch.

The setting of end_step determines the pruning mode.
The initial use of one-shot pruning is recommended. When this leads to an excessively great deterioration in the accuracy of the model, gradual pruning should be used.
<table>
<tr><td><b><u>When end_step = -1 (one-shot pruning)</u></b>
Pruning is executed when this API function is called.
The settings of initial_pr, begin_step, and frequency are ignored.</td></tr>
<tr><td>


</td></tr>
<tr><td></td></tr>
</table>
</td>
</tr>
</table>

	When end_step is other than -1 (gradual pruning) The pruning rate changes with the frequency. Pruning is executed when the update() function (see 4.2.2.1) is called. 

Variables

Variable Name	Description
is_finished	The value of this variable becoming “true” indicates the completion of pruning.

Methods

Method Name	Description
update()	Updates the pruning parameters.
state_dict()	Returns the settings of Pruner in the dict format.
load_state_dict()	Loads the settings of Pruner.

4.2.2.1 update

[Overview]	Updates the pruning parameters.	
[Function/class name]	update	
[Calling format]	update() -> None	
[Argument]	—	—
[Returns]	—	—
[Feature]	Updates the pruning parameters.	
[Remarks]	This API function must always be called per iteration. The timing for calling this API function is at the start (beginning) of each iteration. To execute pruning during iterations of [begin_step, end_step], this API function must be called at least the number of times represented by (end_step – begin_step + 1). **In one-shot pruning** Updates the pruning parameters. The pruning rate of the model does not change. **In gradual pruning** Updates the pruning parameters and the pruning rate of the model.	

4.2.2.2 state_dict

[Overview]	Returns the settings of Pruner in the dict format.	
[Function/class name]	state_dict	
[Calling format]	state_dict() -> Dict[str, Any]	
[Argument]	—	—
[Returns]	—	Data in dict-format that includes the settings of Pruner
[Feature]	Returns the settings of Pruner in the dict format.	

[Remarks]	This API function is used when saving the settings of Pruner.
	Usage example:
	pruner = Pruner(model, pruning_layer_list)
	torch.save(pruner.state_dict(),"pruner.pth")	

4.2.2.3 load_state_dict

[Overview]	Loads the settings of Pruner.	
[Function/class name]	load_state_dict	
[Calling format]	load_state_dict(state_dict: Dict[str, Any]) -> None	
[Argument]	state_dict: Dict[str, Any]	Data in dict-format that includes the settings of Pruner
[Returns]	—	—
[Feature]	Loads the settings of Pruner.	

[Remarks]	This API function is used when loading the settings of Pruner.
	Usage example:
	pruner = Pruner(model, pruning_layer_list)
	pruner.load_state_dict(torch.load("pruner.pth"))	

4.2.3 get_endstep

[Overview]	Gets the step at which pruning ends.	
[Function/class name]	get_endstep	
[Calling format]	get_endstep(dataloader: torch.utils.data.DataLoader, max_iter: int, max_epoch: int, ratio: float) -> int	
[Argument]	dataloader: torch.utils.data.DataLoader	PyTorch data loader Set the data loader for use in training.
	max_iter: int	Maximum number of iterations in training Default: None Input range: $0 < \mathrm{max}\_iter$ Set either max_iter or max_epoch. Both cannot be set at the same time.
	max_epoch: int	Maximum number of epochs in training Default: None Input range: $0 < \mathrm{max}\_epoch$ Set either max_iter or max_epoch. Both cannot be set at the same time.
	ratio: float	Ratio of the step where pruning ends to the maximum number of iterations Default: 0.7 Input range: $0 < ratio$
[Returns]	end_step: int	Number of the step (iteration) where pruning ends
[Feature]	Gets the step at which pruning ends.	
[Remarks]	Set either max_iter or max_epoch. Both cannot be set at the same time. The use of the default value for ratio is recommended. The step where pruning ends can be obtained from the following equation. $$end\_step = Maximum\ iteration \times ratio$$ 70% of the maximum number of iterations is returned by default. When training was performed for 100 iterations, this API function by default returns 70 iterations for pruning.	

4.2.4 get_frequency

[Overview]	Gets the frequency for updating of the pruning rate.	
[Function/class name]	get_frequency	
[Calling format]	get_frequency(dataloader: torch.utils.data.DataLoader, ratio: float) -> int	
[Argument]	dataloader: torch.utils.data.DataLoader	PyTorch data loader Set the data loader for training.
	ratio: float	Ratio for controlling the frequency for updating of the pruning rate Default: 1.0 Input range: $0 < ratio$
[Returns]	frequency: int	Frequency (iteration) for updating of the pruning rate
[Feature]	Gets the frequency for updating of the pruning rate.	
[Remarks]	The use of the default value for ratio is recommended. The total number of iterations per epoch is returned by default. When 1 epoch consists of 100 iterations, this API function returns 100 iterations. When ratio is 1.0, the pruning rate is updated once every epoch. When ratio is 0.5, the pruning rate is updated twice every epoch.	

4.2.5 get_model_info

<table>
<tr><td>[Overview]</td><td colspan="2">Gets a list of information on the model, such as the numbers of parameters.</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Function/class name]</td><td colspan="2">get_model_info</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Calling format]</td><td colspan="2">get_model_info (model: torch.nn.Module,
input_size: List[Tuple[int]],
input_data: Union[List[Any], Mapping[Any]],
) -> pandas.core.frame.DataFrame</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Argument]</td><td>model: torch.nn.Module</td><td>PyTorch model</td></tr>
<tr><td></td><td>input_size: List[Tuple[int]]</td><td>Shape of input data
Default: None
Set either input_size or input_data.
Input values in the order of [batch size, number of channels, height, width].</td></tr>
<tr><td></td><td>input_data:
Union[List[Any],
Mapping[Any]]</td><td>Input data
Default: None
Set either input_size or input_data.</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Returns]</td><td>model_info:
pandas.core.frame.DataFrame</td><td>List including the input shape, output shape, number of parameters, sparsity, number of multiply-and-accumulate calculations before pruning, and number of multiply-and-accumulate calculations after pruning</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Feature]</td><td colspan="2">Gets information on the convolution layers and fully connected layers of the model in terms of the input shape, output shape, number of parameters, sparsity, number of multiply-and-accumulate calculations before pruning, and number of multiply-and-accumulate calculations after pruning.
The values in terms of the items listed below are obtained for each layer.
<table>
<tr><td>module name</td><td>Layer name</td></tr>
<tr><td>input shape</td><td>Input shape</td></tr>
<tr><td>output shape</td><td>Output shape</td></tr>
<tr><td>params</td><td>Number of parameters</td></tr>
<tr><td>sparsity</td><td>Sparsity rate (pruning rate)</td></tr>
<tr><td>Baseline MAC</td><td>Number of multiply-and-accumulate calculations before pruning</td></tr>
<tr><td>Current MAC</td><td>Number of multiply-and-accumulate calculations after pruning</td></tr>
</table></td></tr>
<tr><td></td><td colspan="2"></td></tr>
</table>

[Remarks]	This function is only applicable to the convolution layers and fully connected layers.
	The numbers of multiply-and-accumulate calculations are calculated on the assumption that each set of calculations in a multiply-and-accumulate operation is handled as a single bundle.
	For details on torch.nn.Module, see the official documents of PyTorch.
	For details on the pandas.core.frame.DataFrame class, see the official documents of pandas.
	Usage example:
	>>> import torch.nn as nn
	>>> import torch.nn.functional as F
	>>> class NeuralNetwork(nn.Module):
	>>> def __init__(self):
	>>> super(NeuralNetwork, self).__init__()
	>>> self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding='same')
	>>> self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding='same')
	>>> self.fc1 = nn.Linear(64*32*32, 50)
	>>> self.fc2 = nn.Linear(50, 10)
	>>> def forward(self, x):
	>>> x = F.relu(F.max_pool2d(self.conv1(x), 2))
	>>> x = F.relu(F.max_pool2d(self.conv2(x), 2))
	>>> x = x.view(-1, 64*32*32)
	>>> x = F.relu(self.fc1(x))
	>>> x = self.fc2(x)
	>>> return F.log_softmax(x, dim=1)
	>>> model = NeuralNetwork()
	>>> from drpai_compaction_tool.pytorch import get_model_info
	>>> print(get_model_info(model, [(1, 3, 128, 128)]))

4.2.6 deepcopy_model

[Overview]	Deep copy the model. (Deep copy means copies that are completely reproduced.)	
[Function/Class Name]	deepcopy_model	
[Calling format]	deepcopy_model(model: torch.nn.Module) -> torch.nn.Module	
[Argument]	model: torch.nn.Module	PyTorch model
[Returns]	copied_model: torch.nn.Module	Deep copied PyTorch model
[Feature]	Deep copy the model. Deep copy means copies that are completely reproduced.	

[Remarks]	deepcopy() cannot be executed for a pruned model.
	In case of need to deep copy a pruned model, please use the deepcopy_model() function instead of using deepcopy().
	For details on torch.nn.Module, see the official documents of PyTorch.
	Usage example:
	>>> import torch
	>>> from collections import OrderedDict
	>>> model = torch.nn.Sequential(OrderedDict([
	("fc1", torch.nn.Linear(3,1024)),
	("fc2", torch.nn.Linear(1024,10))
	]))
	>>> from drpai_compaction_tool.pytorch import Pruner, deepcopy_model
	>>> _ = Pruner(model, ["fc1" "fc2"])
	>>> copied_model = deepcopy_model(model)	

4.2.7 load_pruned_state_dict

<table>
<tr><td>[Overview]</td><td colspan="2">Loads the model which weights (stated_dict) is pruned</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Function/Class Name]</td><td colspan="2">load_pruned_state_dict</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Calling format]</td><td colspan="2">load_pruned_state_dict(model: torch.nn.Module,
pruned_state_dict: Dict,
strict: bool) -> None</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="3">[Argument]</td><td>model: torch.nn.Module</td><td>PyTorch model</td></tr>
<tr><td>pruned_state_dict: Dict</td><td>Data in dict-format that includes the pruned weights.</td></tr>
<tr><td>strict: bool</td><td>Whether to strictly enforce that the keys in pruned_state_dict match the keys returned by the model's state_dict() function.
Default: True
When this argument is set to “true”, an error is returned if the keys do not match.</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Returns]</td><td>-</td><td>-</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Feature]</td><td colspan="2">Loads the model which weights (stated_dict) is pruned.
The pruned weights (state_dict) means weight parameters stored in weight_orig / weight_mask format.</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>[Remarks]</td><td colspan="2">This function performs the following behavior depending on each parameter.
<table>
<tr><th>model</th><th>pruned_state_dict</th><th>behavior</th></tr>
<tr><td>sparse (pruned)</td><td>sparse (pruned)</td><td>This function loads the pruned_state_dict to model.</td></tr>
<tr><td>sparse (pruned)</td><td>dense</td><td>This function raises an error.</td></tr>
<tr><td>dense</td><td>sparse (pruned)</td><td>This function loads the pruned_state_dict to model.
Note: After the dense model to be input to this function, the weight parameters will change to weight_org/weight_mask format.</td></tr>
<tr><td>dense</td><td>dense</td><td>This function raises an error.</td></tr>
</table>
For details on torch.nn.Module, see the official documents of PyTorch.

Usage example:
>>> import torch, torchvision
>>> pruned_model = torchvision.models.resnet18(num_classes=1000)</td></tr>
</table>

	>>> from drpai_compaction_tool.pytorch import Pruner, load_pruned_state_dict
	# This example prunes the layer called "layer1.0.conv1".
	>>> _ = Pruner(pruned_model, ["layer1.0.conv1"])
	>>> torch.save(pruned_model.state_dict(), "pruned_state_dict.pth")
	>>> dense_model = torchvision.models.resnet18(num_classes=1000)
	# Note: After the dense model to be input to this function, the weight parameters will change to weight_org/weight_mask format.
	>>> load_pruned_state_dict(dense_model, torch.load("pruned_state_dict.pth"))

4.2.8 RewriterContext

[Overview]	Changes the format of the pruned weights in the context.	
[Function/Class Name]	RewriterContext	
[Calling format]	RewriterContext (model: torch.nn.Module) -> object	
[Argument]	model: torch.nn.Module	PyTorch model
[Returns]	object	Python object
[Feature]	Changes the format of the pruned weights in the context. The pruned weights mean weight parameters stored in weight_orig / weight_mask format. In the context, the model maintains weights parameters in the weight format instead of the weight_orig / weight_mask format.	

[Remarks]	It is recommended to use this class when exporting a model that utilizes the torch.nn.MultiheadAttention() layer to ONNX. For more information, please refer to 6. Usage Notes.
	For details on torch.nn.Module, see the official documents of PyTorch.
	Usage example:
	>>> import torch, torchvision
	>>> pruned_model = torchvision.models.resnet18(num_classes=1000)
	>>> from drpai_compaction_tool.pytorch import Pruner, RewriterContext
	# This example prunes the layer called "layer1.0.conv1".
	>>> _ = Pruner(pruned_model, ["layer1.0.conv1"])
	>>> img = torch.randn(1,3,224,224)
	# Export an onnx format model in the context.
	>>> with RewriterContext(pruned_model):
	    torch.onnx.export(pruned_model,
	        img,
	        "exported_model.onnx",
	        input_names=['input'],
	        output_names=["output"])	

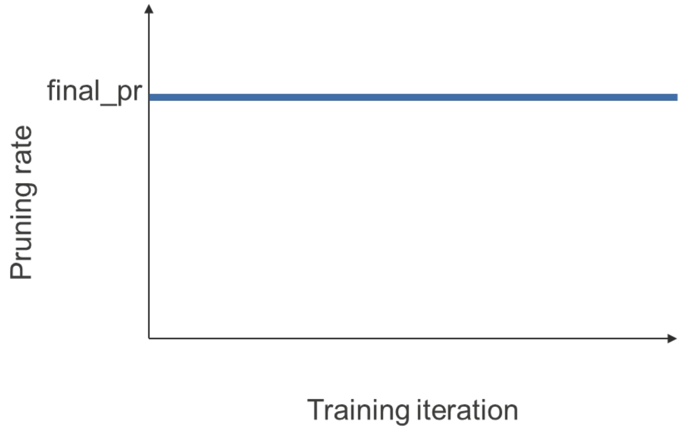
4.3 [TensorFlow]

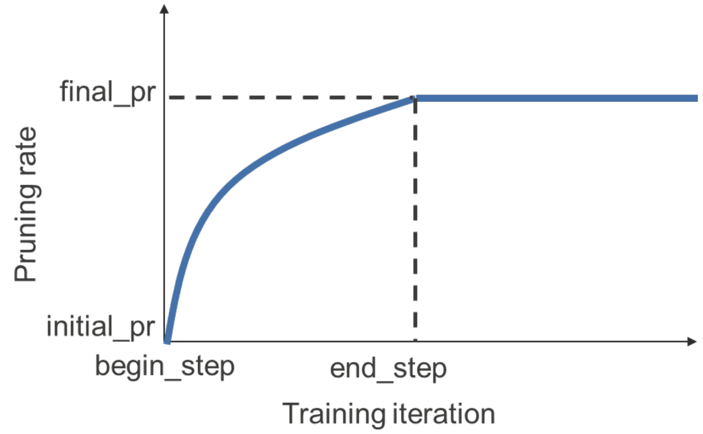
4.3.1 make_pruning_layer_list

[Overview]	Sets layers to which pruning is not to be applied and creates the list of the target layers for pruning.	
[Function/class name]	make_pruning_layer_list	
[Calling format]	make_pruning_layer_list(model: tensorflow.python. keras.engine.functional.Functional, prune_last: bool = False, prune_dwise: bool = False) -> List[str]	
[Argument]	model: tensorflow.python. keras.engine. functional.Functional	TensorFlow model Only a functional model or sequential model can be input.
	prune_last: bool	When this argument is set to “true”, the last layer is included among the targets for pruning. Default: False
	prune_dwise: bool	When this argument is set to “true”, the depthwise convolution layer is included among the targets for pruning. Default: False
[Returns]	pruning_layer_list: List[str]	List of the target layers for pruning
[Feature]	Creates the list of the target layers for pruning based on the input model. Pruning is to be applied to the layers defined with tensorflow.keras.layers.Conv2D or tensorflow.keras.layers.Dense.	
[Remarks]	Pruning cannot be applied to the first layer because doing so significantly worsens the accuracy of the model. Pruning also cannot be applied to a layer for which the number of input channels is not a multiple of 32. Pruning is not applied to the last layer or depthwise convolution layer by default because doing so significantly worsens the accuracy of the model. For details on tensorflow.keras.Model, tensorflow.keras.layers.Conv2D, and tensorflow.keras.layers.Dense, see the official documents of TensorFlow.	

4.3.2 Pruner

[Overview]	Controls the pruning parameters.	
[Function/class name]	Pruner	
[Calling format]	class Pruner (model: tensorflow.keras.Model, pruning_layer_list: List[str], initial_pr: float, final_pr: float, begin_step: int, end_step: int, frequency: int,)	
[Argument]	model: tensorflow.keras.Model	TensorFlow model
	pruning_layer_list: List[str]	List of the target layers for pruning
	initial_pr: float	Initial value of pruning rate Default: 0.01 Input range: $0 \leq initial\_pr < 1$
	final_pr: float	Final value of pruning rate Default: 0.7 Input range: $0 \leq final\_pr < 1$
	begin_step: int	Number of the step (iteration) where pruning starts Default: 0 Input range: $0 \leq begin\_step$
	end_step: int	Number of the step (iteration) where pruning ends Default: -1 Input range: $-1 \leq end\_step$
	frequency: int	Frequency for executing pruning (number of iterations) Default: 100 Input range: $0 < frequency$
[Returns]	pruner: object	Object for setting up pruning
[Feature]	The model is only pruned once when end_step is -1. (Any settings of initial_pr, begin_step, and frequency will be ignored in this case.) When a value other than -1 is set, gradual pruning is applied to the model. For details, refer to [Remarks] below.	

[Remarks]	When end_step = -1, the settings of initial_pr, begin_step, and frequency are ignored. The setting of begin_step = end_step is prohibited. Pruning is carried out over the number of iterations set by [begin_step, end_step]. To complete pruning, training needs to have been performed for the number of iterations represented by (end_step – begin_step + 1). initial_pr and final_pr must be values in the range [0.0, 1.0). The use of the default values for initial_pr and begin_step is recommended. For end_step, setting a value around 70% of the total number of iterations in training is recommended. (For example, when the total number of iterations was 100, set 70 iterations.) Note that get_endstep() can be used to set the value. For frequency, setting the total number of iterations per epoch is recommended. Note that get_frequency() can be used to set the value. For details on tensorflow.keras.Model, see the official documents of TensorFlow. The setting of end_step determines the pruning mode. The initial use of one-shot pruning is recommended. When this leads to an excessively great deterioration in the accuracy of the model, gradual pruning should be used. **When end_step = -1 (one-shot pruning)** Pruning is executed only once at the beginning. The settings of initial_pr, begin_step, and frequency are ignored. 

	When end_step is other than -1 (gradual pruning) The pruning rate changes with the frequency.
	

Variables

Variable Name	Description
—	—

Methods

Method Name	Description
get_pruning_model()	Gets the model for pruning.

4.3.2.1 get_pruning_model

[Overview]	Gets the model for pruning.	
[Function/class name]	get_pruning_model	
[Calling format]	get_pruning_model() -> tensorflow.keras.Model	
[Argument]	—	—
[Returns]	pruning_model: tensorflow.keras.Model	Model for pruning
[Feature]	Gets the model for pruning.	
[Remarks]	—	

4.3.3 get_endstep

[Overview]	Gets the step at which pruning ends.	
[Function/class name]	get_endstep	
[Calling format]	get_endstep(num_data: int, batch_size: int, max_iter: int, max_epoch: int, ratio: float) -> int	
[Argument]	num_data: int	Number of training data
	batch_size: int	Batch size in training
	max_iter: int	Maximum number of iterations in training Default: None Input range: $0 < max\_iter$ Set either max_iter or max_epoch. Both cannot be set at the same time.
	max_epoch: int	Maximum number of epochs in training Default: None Input range: $0 < max\_epoch$ Set either max_iter or max_epoch. Both cannot be set at the same time.
	ratio: float	Ratio of the step where pruning ends to the maximum number of iterations Default: 0.7 Input range: $0 < frequency$
[Returns]	end_step: int	Number of the step (iteration) where pruning ends
[Feature]	Gets the step at which pruning ends.	
[Remarks]	Set either max_iter or max_epoch. Both cannot be set at the same time. The use of the default value for ratio is recommended. The step where pruning ends can be obtained from the following equation. $end\_step\ =\ Maximum\ iteration\ \text{x}\ ratio$ 70% of the maximum number of iterations is returned by default. When training was performed for 100 iterations, this API function by default returns 70 iterations for pruning.	

4.3.4 get_frequency

[Overview]	Gets the frequency for updating of the pruning rate.	
[Function/class name]	get_frequency	
[Calling format]	get_frequency(num_data: int, batch_size: int, ratio: float) -> int	
[Argument]	num_data: int	Number of training data
	batch_size: int	Batch size in training
	ratio: float	Ratio for controlling the frequency for updating of the pruning rate Default: 1.0 Input range: $0 < ratio$
[Returns]	frequency: int	Frequency (iteration) for updating of the pruning rate
[Feature]	Gets the frequency for updating of the pruning rate.	
[Remarks]	The use of the default value for ratio is recommended. The total number of iterations per epoch is returned by default. When 1 epoch consists of 100 iterations, this API function returns 100 iterations. When ratio is 1.0, the pruning rate is updated once every epoch. When ratio is 0.5, the pruning rate is updated twice every epoch.	

5. Recommendations during Application of Pruning

This section gives recommendations on how to suppress deterioration of the accuracy of the model due to the application of pruning. Attempt the recommended measures if unacceptably low accuracy is encountered after pruning then retraining.

- Perform pruning then retraining with 70% pruning rate to check the accuracy and processing performance. After that, change pruning rate depending on the accuracy and processing performance, and perform pruning then retraining again to check the accuracy and processing performance.
- Pruning should initially be performed as one-shot pruning. If the resulting accuracy is too low, try gradual pruning.
- Do not apply pruning to the first and last layers.
- Do not apply pruning to the depthwise convolution layer.
- Use the same parameters in training, such as the learning rate, optimizer, epoch, as those that were set for initial training.

Note: The recommended measures are not guaranteed to always suppress unacceptable deterioration of the accuracy.

6. Usage Notes

- deepcopy() cannot be executed for a pruned model.
  To copy a pruned model, please use deepcopy_model() function. For more details about this function, see 4.2.6.
- When loading a saved pruned model, see 3.4 and 3.7.
- Do not use early stopping with gradual pruning. Doing so may cause training to be terminated when pruning has not yet been completed. For details, see the TensorFlow sample code.
- Do not use Exponential Moving Average (EMA) with gradual pruning. It may cause incorrect pruning result. After pruning then retraining, you can confirm whether the pruning rate is correct by using get_model_info().
- When using the PruningSummaries() callback function of TensorFlow, do not also use the TensorBoard() callback function of TensorFlow. Since the TensorBoard() callback function is initialized in the PruningSummaries() callback function, use of both callback functions is judged to represent a double definition and an error will occur. For details, see the TensorFlow sample code.
- When using the TensorFlow version of the pruning tool, only Keras2 is supported. Therefore, please set the environment variable TF_USE_LEGACY_KERAS as shown below before using this tool.

```
$ export TF_USE_LEGACY_KERAS=1
```

- Notes on using torch.nn.MultiheadAttention() in models
  - ✓ Considerations for exporting to ONNX

    When exporting a model that uses torch.nn.MultiheadAttention() to ONNX, please call the following code snippet highlighted in red before exporting to ONNX.

```
# Define a sample model using MultiheadAttention()
class SelfAttentionLike(torch.nn.Module):
    def __init__(self, embed_dim = 32, num_heads = 2):
        super(SelfAttentionLike, self).__init__()
        self._embed_dim = embed_dim
        self._num_heads = num_heads
        self.linear1 = torch.nn.Linear(self._embed_dim,
                                       self._embed_dim)
        self.multihead_attention = \
              torch.nn.MultiheadAttention(self._embed_dim,
                                          self._num_heads)
        self.linear2 = torch.nn.Linear(self._embed_dim,
                                       self._embed_dim)

        def forward(self, x):
            x = self.linear1(x)
            attn_output, _ = self.multihead_attention(query=x,
                                                      key=x,
                                                      value=x)
            output = self.linear2(attn_output)
            return output

# Create a model
embed_dim = 32
seq_len = 5
batchsize = 10
input_size = [(batchsize, seq_len, embed_dim)]
model = SelfAttentionLike(embed_dim=embed_dim)

# Perform a pruning to a model
pruning_layer_list = make_pruning_layer_list(model,
                                             input_size=input_size)
pruner = Pruner(model,
                pruning_layer_list,
                final_pr=0.7)
print(get_model_info(model,
                     input_size=input_size))

# Export a model as ONNX
img = torch.randn(input_size[0])
with RewriterContext(model):
    torch.onnx.export(model,
                      img,
                      "exported_model.onnx",
                      input_names=['input'],
                      output_names=["output"])

```

- Resolving PT_GMI_002 errors when using get_model_info()

  When using get_model_info() to get the pruning rate, the following error may occur:

```
[PT_GMI_002] Failed to run. See above stack traces for more details. If that does not help, please contact Renesas.
```

If this error occurs, please use the print_sparsity() function instead of get_model_info() as follows to verify the pruning rate:

```python
def calc_sparsity(module):
    if hasattr(module, 'weight_mask'):
        m = module.weight_mask
        sparsity = 1.0 - m.count_nonzero().item() / m.nelement()
    else:
        w = module.weight
        sparsity = 1.0 - w.count_nonzero().item() / w.nelement()
    return sparsity

def print_sparsity(model):
    for name, module in model.named_modules():
        if not isinstance(module, (torch.nn.Conv2d, torch.nn.Linear)):
            continue
        sparsity = calc_sparsity(module)
        print(f'{name} = {sparsity:.04f}')

# Use the print_sparsity() function instead of get_model_info().
# Before
# print(get_model_info(model, [(1,3,224,224)]))
# After
print_sparsity(model)

```

Revision History

Rev.	Date	Description	
		Page	**Summary**
1.00	Dec.05.23	—	First edition issued
2.00	Oct.08.24	10, 54	Added updates in DRP-AI Extension Pack V1.1.0 Added usage notes on using torch.nn.MultiheadAttention()
3.00	Jun.30.25	9, 49, 56	Added updates in DRP-AI Extension Pack V1.2.0 Added the specifications for RewriterContext() Added usage notes on using TensorFlow version of the pruning tool Updated usage notes on using torch.nn.MultiheadAttention()

Notice

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.