Assignment 2

In the following experiments we train a model using continual learning. We have an original model and a custom model based on the original one. The original model is the following:

```python
class Net(nn.Module):
    def __init__(self):
        '''
            Initialize the basic architecture.
        '''
        super(Net, self).__init__()
        # Convolution layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        # Dropout
        self.conv_drop = nn.Dropout2d(0.5)

        # Fully connected layers
        self.fc1 = nn.Linear(128*3*3, 256)
        self.fc2 = nn.Linear(256, 2)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.bn3(self.conv3(x)))
        x = F.max_pool2d(x, 2)
        x = self.conv_drop(x)
        x = rearrange(x, 'b c h w -> b (c h w)')
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

Our objective is to successfully train a model on two consecutive tasks without experiencing the issue of forgetting the first task upon learning the second—commonly known as 'Catastrophic forgetting.' In the initial task, the model is trained to recognize the first set of five digits (0, 1, 2, 3, 4) from the MNIST dataset, and subsequently, the second task involves training the model to learn the remaining five digits (5, 6, 7, 8, 9). The dataset comprises a total of 30,596 samples for the first task and 29,404 samples for the second task, resulting in a total of 60,000 samples.

Each training session is limited to three epochs for both the first and second tasks. The methodologies presented and implemented in the subsequent sections were introduced in 'Learning without Forgetting' [1], hereafter referred to as LWF. With the exception of the final section, where an alternative method is introduced.

In Section 1, we apply a method similar to 'joint training,' from LWF (though distinct, as we exclusively use data for the second task). Moving to Section 2, we employ the 'feature extraction' method introduced

in LWF. In Section 3, we revisit the 'joint training' method outlined in LWF. Subsequently, in Section 4, we experiment once again with the 'joint training' approach, this time using only a small subset (500 samples) of data from task 1, following the methodology introduced in LWF. Lastly, in Section 5, we introduce Elastic Weight Consolidation (EWC) [2] as an additional method for continual learning.

Section I

In this section we use a method very similar to the “joint training” method introduced in LWF. This method, however, has a difference because it does not include any data from the first task.

First, we train the model after adapting the last fully connected layer using the following code:

```
def adapt_last_layer(self, output_size):
    """Dynamically adapt the last layer for
       the given output size and retain previously learned weights."""
    if output_size == self.fc2.out_features:
        return

    # Step 1: Store the current FC layer's weights and biases
    current_weights = self.fc2.weight.data
    current_biases = self.fc2.bias.data

    # Step 2: Create a new FC layer with the updated size
    new_fc = nn.Linear(self.fc2.in_features, output_size).to(device)

    # Step 3: Copy the weights and biases from the old layer to the new layer
    new_fc.weight.data[:len(current_biases)] = current_weights
    new_fc.bias.data[:len(current_biases)] = current_biases

    # Assign the new FC layer to the model
    self.fc2 = new_fc
    print(f"extend model layers as a incremental classes in each task")
```

This code extends the output size and stores the already learned weights and biases in the neurons that are associated with a past task. We use this adaptation as well as the normal torch training flow, to train the first task across all the methods, including EWC.

After the training for the first task concludes, the evaluation on the test data reveals an accuracy of approximately 99.8% for task 1, as we can see in Table I.

TABLE I

TEST ACCURACY AFTER TRAINING FIRST TASK USING THE STANDARD METHOD

Task	Accuracy (%)
Task 1	99.79

Again, we extend the output size, this time to 10 classes and store the learn weights. Then, we fine tune all the original model and learn weights for the head for the second task.

In table II, we can notice that the model has suffered from catastrophic forgetting and now it is only able to correctly predict the outputs for task 2 while is completely unable to predict the outputs for the past task.

TABLE II
TEST ACCURACY AFTER TRAINING SECOND TASK USING A METHOD SIMILAR TO JOINT TRAINING BUT INCLUDING ONLY DATA FOR THE SECOND TASK

Task	Accuracy (%)
Task 1	0.00
Task 2	99.44
Both tasks	49.72

Section II

In this section we use the method "feature extraction" from LWF. In this method we freeze the original model and add a head for the second task, which is the only layer for which we update the parameters when learning the second task.

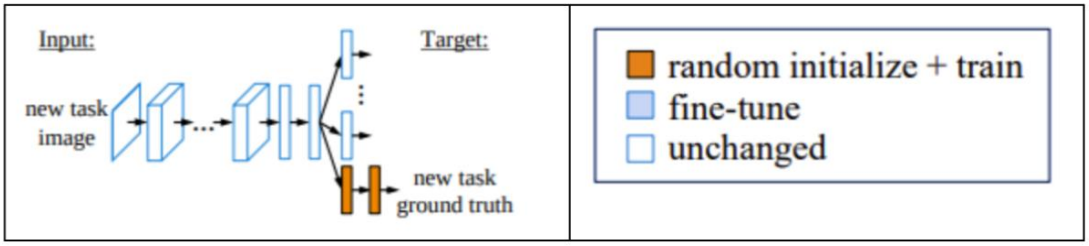

Fig. 1. Feature extraction method

We first implement a custom net using the original net as base. We add the head for the new task and overwrite the forward function. Then, we return the concatenated outputs. We use this custom net for all the methods, including EWC. The difference between methods lies in what layers are freeze or unfreeze, number of samples for training or loss function definition (Please, refer to the code of each section for exact implementation details other than those explained in each section).

```python
### Mine: Create a class to edit the net
class CustomNet(nn.Module):
    ### Mine: define init for custom net
    def __init__(self, net_instance, f2_new_task_output_size):
        super(CustomNet, self).__init__()

        ### Mine: Use the Net given as base model
        self.net = net_instance

        ### Mine: Freeze all layers of the original net
        for param in self.net.parameters():
            param.requires_grad = False

        ### Mine: Define a new head for the new task
        self.fc2_new_task_head = nn.Sequential(nn.Linear(self.net.fc1.out_features, f2_new_task_output_size))

        ### Mine: Unfreeze the parameters of the new task head
        for param in self.fc2_new_task_head.parameters():
            param.requires_grad = True

    ### Mine: Define the forward function based on original net
    def forward(self, x):
        ### Mine: Forward pass through the original network
        x = F.relu(self.net.bn1(self.net.conv1(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.net.bn2(self.net.conv2(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.net.bn3(self.net.conv3(x)))
        x = F.max_pool2d(x, 2)
        x = self.net.conv_drop(x)
        x = rearrange(x, "b c h w -> b (c h w)")
        x = F.relu(self.net.fc1(x))
        x = F.dropout(x, training=self.net.training)

        ### Mine: pass x to the head for new task (fc2_new_task)
        output_fc2_new_task = self.fc2_new_task_head(x)

        ### Mine: pass x to the Original head (fc2) output
        output_fc2 = self.net.fc2(x)

        ### Mine: Concatenate both outputs
        final_output = torch.cat((output_fc2, output_fc2_new_task), dim=1)

        return final_output
```

Then for the first task we simply adapt the original model and for the second task we use the custom net.

```python
### Mine: If seen classes is only first 5 classes (0,1,2,3,4)
if len(seen_classes_list) == 5:
   print("\n")
   print("Network: NET")

   ### Mine: Simple adapt last layer using the given functions
   model.adapt_last_layer(len(seen_classes_list))

    ### Mine: If seen classes is more than first 5 classes (0,1,2,3,4)
    else:
        print("\n")
        print("Network: CUSTOM NET")

        ### Mine: Use customnet
        model = CustomNet(model, len(seen_classes_list))
        model.to(device)
```

After training for task 2, we can see in table III that now the model is able to remember the first task, but it could not learn the second task. This makes sense because the layers of the original model were all frozen and the new head only had three epochs to update the parameters for the second task.

TABLE III
TEST ACCURACY AFTER TRAINING SECOND TASK USING FEATURE EXTRACTION METHOD

Task	Accuracy (%)
Task 1	99.79
Task 2	0.00
Both tasks	49.89

Section III

In this section we use the method “joint training”. In this method the layers of the original model are all unfreeze and a head for the new task is added. It is called joint training because we use the data from both task 1 and task 2.

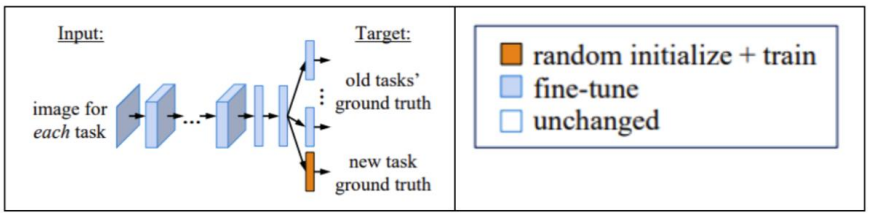

Fig. 2. Joint training method

After unfreezing all the layers, the only thing we need to add to our previous implementation is that we must accumulate the data from the first task and second task together, so that, when we fine tune for the second task, we use data from both tasks. First, we initialize two empty variables, one for the images and one the targets.

```
### Mine: Define two variables to accumulate the training data
accumulated_data_x = None
accumulated_data_t = None
```

Then we accumulate the data for the two tasks as follows:

```python
### Mine: If seen classes is only first 5 classes (0,1,2,3,4)
if task_idx == 0:
    ### Mine: accumulate the sample data
    accumulated_data_x = x_train_task

    ### Mine: accumulate the target data
    accumulated_data_t = t_train_task

### Mine: If seen classes is more than first 5 classes (0,1,2,3,4)
else:
    ### Mine: concatenate the accumulated data
    print(accumulated_data_t.size())
    print(t_train_task.size())
    accumulated_data_x = torch.concat((accumulated_data_x, x_train_task), axis=0)
    accumulated_data_t = torch.concat((accumulated_data_t, t_train_task), axis=0)

train_dataset = TensorDataset(accumulated_data_x, accumulated_data_t)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

We can see in table IV that this time the model is able to remember pretty well the first task while learning the second task.

TABLE IV
TEST ACCURACY AFTER TRAINING SECOND TASK USING JOINT TRAINING METHOD

Task	Accuracy (%)
Task 1	99.24
Task 2	99.05
Both tasks	99.15

Section IV

In this section, we use again the joint training method but this time we reduced the amount of data for the first task to only 500 random samples.

The only implementation we are required to do in this part is to randomly sample 500 samples from task 1 before accumulating the data with that of the second task.

```python
### Mine: define number of samples
num_samples = 500

### Mine: Generate random indices using torch
indices = torch.randperm(accumulated_data_x.size(0))[:num_samples]

### Mine: use random indices to get samples from first task
sampled_x_train_tasks = accumulated_data_x[indices]

### Mine: use random indices to get target samples from first task
sampled_t_train_tasks = accumulated_data_t[indices]

### Mine: concatenate the accumulated data
accumulated_data_x = torch.concat((sampled_x_train_tasks, x_train_task), axis=0)
accumulated_data_t = torch.concat((sampled_t_train_tasks, t_train_task), axis=0)
```

We observe that the model effectively retains knowledge of the first task while learning the second, even with a substantial reduction in the amount of training data for the initial task. This indicates that it is unnecessary to utilize the entirety of the training data from task 1 when fine-tuning for task 2. In essence, we can achieve efficient learning even with a reduced dataset for the first task.

TABLE V
TEST ACCURACY AFTER TRAINING SECOND TASK USING JOINT TRAINING METHOD WITH ONLY 500 SAMPLES FROM THE TASK 1

Task	Accuracy (%)
Task 1	89.36
Task 2	99.53
Both tasks	94.44

Section V

In this section we implement EWC. This method is a regularization method that lets us decide the importance of task 1 by changing the value of the hyperparameter lambda.

$$L_B(\theta) + \sum \frac{\lambda}{2} F_i \left(\theta_i - \theta^*_{A,i}\right)^2 \quad \text{eq (I)}$$

EWC harness fisher information F as an estimate of how important a certain parameter is for a previous task; Fisher information can be approximated using the accumulation of gradients to the power of 2. The parameters θ^*_A of the past task A are used as the mean of a gaussian distribution that approximates the posterior distribution that contains information about which parameters were important when learning task A.

We obtain both the fisher information and the mean with the following code:

```
### Mine: Initialize a dict for the fisher information of current task
for name, param in model.named_parameters():
    ### Mine: adjust name for compatibility with custom_net
    new_name = "net." + name

    ### Mine: store all parameters
    old_par_dict[task_id][new_name] = param.data.clone()

    ### Mine: Use accumulation of gradients to the power of 2 as
    ### approximation of fisher information
    fisher_dict[task_id][new_name] = param.grad.data.clone().pow(2)
```

Then we also create another function which is called “ewc_training” for training the second task. The function is similar to the normal training function except that we use the loss function of EWC as follows:

```
### Mine: get loss for current task using criterion
loss = criterion(outputs, labels)

### Mine: For parameter we get the fisher information and the old parameter
for name, param in model.named_parameters():
    ### Mine: Skip parameters for fc2 because we just added this to the network

    if not name.startswith("fc2"):
        ### Mine: Get fisher information from old task for the given parameters
        fisher_info = fisher_dict[task_id - 1][name]

        ### Mine: Get parameters from old task
        old_parameter = old_par_dict[task_id - 1][name]

        ### Mine: Compute loss based of Elastic weight consolidation
        loss += (fisher_info * (param - old_parameter).pow(2)).sum() * ewc_lambda
```

We experimented with various lambda values during training, including smaller ones such as 0.5, 1, or even 100. However, the discernible impact of EWC training was not evident until we set lambda to 10,000. As reflected in Table VI, at this lambda value, the model demonstrated a noteworthy ability to retain knowledge of task 1, achieving an accuracy of 43.51%. This underscores that while EWC may not match the efficacy of joint training from LWF in this context, it serves as a valuable regularization mechanism, enabling the model to preserve information about task 1 while simultaneously acquiring knowledge for task 2.

TABLE VI

TEST ACCURACY AFTER TRAINING SECOND TASK USING EWC

Task	Accuracy (%)
Task 1	43.51
Task 2	98.00
Both tasks	70.76

Details of implementation

Because we use a custom net, to load the model .pth file properly you must follow the following steps

```
model = Net().to(device)
model.adapt_last_layer(5)
model = CustomNet(model, 10)
model.to(device)

model_path = pth_file_path   # For each .pth file
model.load_state_dict(torch.load(model_path))
```

References

[1] Li, Z., & Hoiem, D. (2017). Learning without forgetting. IEEE transactions on pattern analysis and machine intelligence, 40(12), 2935-2947.

[2] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. Proceedings of the national academy of sciences, 114(13), 3521-3526.