L8 Transfer Learning

Transfer Learning

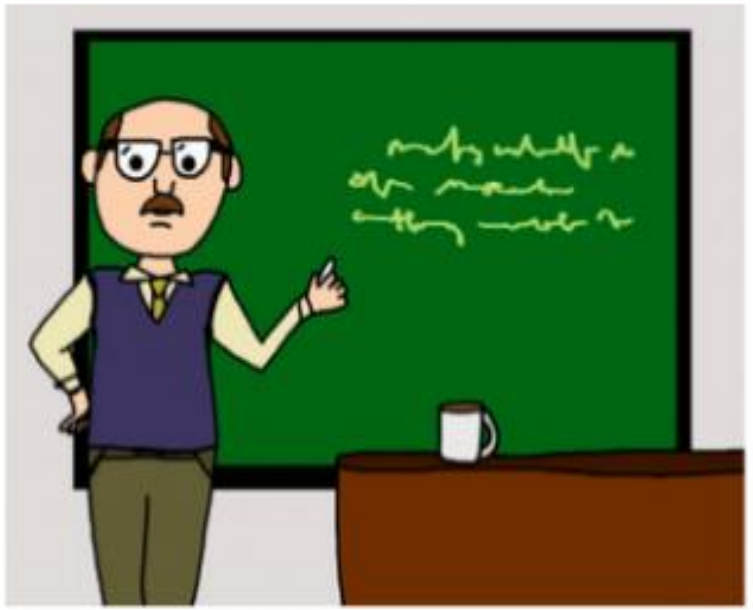
- Given a huge size of the dataset, building a model from scratch is a real challenge.
- Transfer learning is helpful tool especially when working with limited time and computational power.
- So PyTorch transfer learning and pre-trained models could be used to leverage on a real-world project

What is Transfer Learning?

- How do you learn a topic from a new domain ?
- What are the different approaches you would take to understand the topic?
  - Search online for resources
  - Read articles and blogs
  - Refer to books
  - Look out for video tutorials, and so on
- Disadv: you are the only person who is putting in all the effort.
- But there is another approach, which might yield better results in a short amount of time.
- You can consult a domain/topic expert who has a solid grasp on the topic you want to learn.
- This person will transfer their knowledge to you, thus expediting your learning process.

What is Transfer Learning?

Learning from Scratch



Transfer Learning

The first approach, where you are putting in all the effort alone, is an example of learning from scratch.

The second approach is referred to as transfer learning. There is a knowledge transfer happening from an expert in that domain to a person who is new to it

Transfer Learning

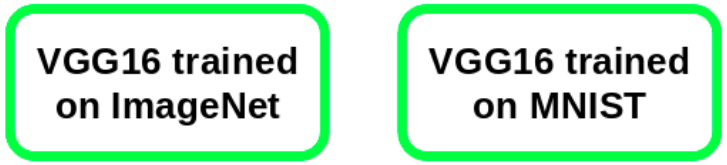
- Neural Networks and Convolutional Neural Networks (CNNs) are examples of learning from scratch.
- Both these networks extract features from a given set of images (in case of an image related task) and
- then classify the images into their respective classes based on these extracted features.
- This is where transfer learning and pre-trained models are useful in deep learning
- When we do not have the unlimited computational power of the top tech organizations, we need to make with our local machines

Pre-trained models

- A pre-trained model, is a model already designed and trained by a certain person or team to solve a specific problem.
- Recall that we learn the weights and biases while training models like Neural Network and CNNs.
- These weights and biases, when multiplied with the image pixels, help to generate features.
- Pre-trained models share their learning by passing their weights and biases matrix to a new model.
- For transfer learning, first select the right pre-trained model and then pass its weight and bias matrix to the new model.
- Ex: pre-trained models available - BERT, ULMFiT, and VGG16.
- BERT and ULMFiT are used for language modeling and VGG16 is used for image classification tasks
- We need to decide which will be the best-suited model for our problem.

Pre-trained models

- How to decide the right pre-trained model based on our problem?
- VGG16 can have different weights, i.e. VGG16 trained on ImageNet or VGG16 trained on MNIST



- ImageNet vs. MNIST
- To decide the right pre-trained model for our problem, we should explore these ImageNet and MNIST datasets.
- The ImageNet dataset consists of 1000 classes and a total of 1.2 million images. Some of the classes in this data are animals, cars, shops, dogs, food, instruments
- MNIST is trained on handwritten digits. It includes 10 classes from 0 to 9
- If our problem statement is to classify images vehicles, then VGG16 model trained on the ImageNet dataset would be more useful

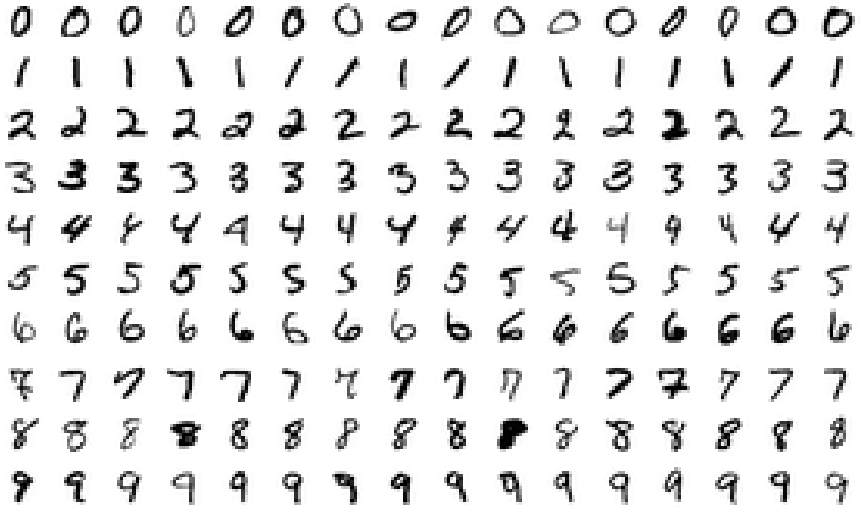
Pre-trained models - ImageNet vs. MNIST

ImageNet Challenge

IMAGENET

- 1,000 object classes (categories).
- Images:
  - 1.2 M train
  - 100k test.

MNIST



Pre-trained Model Approach

- Select Source Model. A pre-trained source model is chosen from available models. Many research institutions release models on large and challenging datasets that may be included in the pool of candidate models from which to choose from.
- Reuse Model. The pre-trained model can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the model, depending on the modeling technique used.
- Tune Model. Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

The Transfer Learning Workflow

1. Selecting a Pre-trained Model

- The first step in transfer learning is to choose a pre-trained model. Pre-trained models are trained on large datasets for common tasks such as image classification or natural language processing. Popular choices include VGG16, ResNet, and BERT

2. Understanding Model Architecture

- Once a pre-trained model is selected, it is crucial to understand its architecture. This involves examining the layers, parameters, and the specific task it was originally designed for. Familiarizing with the model's structure is essential for effective transfer learning

3. Modifying the Model for the New Task

- The next step is to adapt the pre-trained model to the specifics of the new task. This involves modifying the final layers of the model to match the number of classes in the target task.

The Transfer Learning Workflow

4. Feature Extraction

One common approach in transfer learning is feature extraction. This involves using the pre-trained model as a fixed feature extractor, where the early layers of the model are frozen, and only the final layers are adapted to the new task. This way, the model retains the knowledge learned from the source task while adjusting to the nuances of the target task

5. Fine-tuning

Another strategy is fine-tuning, where not only the final layers but also some of the earlier layers are adapted to the new task. This allows the model to adjust its weights more flexibly based on the target task's data. Fine-tuning is beneficial when the source and target tasks are closely related

Applications of Transfer Learning

1. Image Classification

- Transfer learning has proven highly effective in image classification tasks. Pre-trained models trained on massive image datasets, such as ImageNet, can be repurposed for specific image classification tasks. This is particularly valuable when dealing with limited labeled data for a specific domain.

2. Object Detection

- For tasks involving object detection in images, transfer learning accelerates the training process. Pre-trained models like Faster R-CNN or YOLO can be fine-tuned on datasets with a smaller number of object classes, making them adept at detecting objects specific to the target task.

Applications of Transfer Learning

3. Natural Language Processing (NLP)

- pre-trained language models like BERT and GPT have revolutionized transfer learning. Python libraries such as Hugging Face's Transformers facilitate the use of these models for a variety of NLP tasks, including sentiment analysis, text classification, and named entity recognition.

4. Speech Recognition

- Pre-trained models trained on large-scale speech datasets can be adapted to recognize specific voices or languages

5. Medical Imaging

- Pre-trained models can be applied to tasks such as tumor detection or organ segmentation with minimal labeled medical data, offering potential breakthroughs in diagnostics.

Challenges in Transfer Learning

- Domain Shift: The source and target tasks should be closely related to ensure transfer learning effectiveness. A significant domain shift may lead to suboptimal performance.
- Overfitting: When adapting a pre-trained model to a new task, there is a risk of overfitting, especially when dealing with limited labeled data. Techniques such as regularization and data augmentation can mitigate this challenge.
- Model Size: Pre-trained models can be large, requiring significant computational resources for fine-tuning or feature extraction. Efficient handling of model size is crucial for practical implementation

Transfer Learning

- Deep convolutional neural network models may take days or even weeks to train on very large datasets.
- In deep learning, transfer learning is a technique whereby a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest.
- A way to short-cut this process is to re-use the model weights from pre-trained models that were developed for standard computer vision benchmark datasets, such as the ImageNet image recognition tasks.
- Top performing models can be downloaded and used directly, or integrated into a new model for our own computer vision problems.
- Transfer learning generally refers to a process where a model trained on one problem is used in some way on a second related problem.
- Ex: In a supervised learning context, the input is the same but the target may be of a different nature. we may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting

Transfer Learning for Image Recognition

- A range of high-performing models have been developed for image classification and demonstrated on the annual ImageNet Large Scale Visual Recognition Challenge, or ILSVRC.
- This challenge, referred as ImageNet, given the source of the image used in the competition, has resulted in a number of innovations in the architecture and training of convolutional neural networks.
- many of the models used in the competitions have been released under a permissive license.
- These models can be used as the basis for transfer learning in computer vision applications.

- Useful Learned Features: The models have learned how to detect generic features from photographs, given that they were trained on more than 1,000,000 images for 1,000 categories.
- State-of-the-Art Performance: The models achieved state of the art performance and remain effective on the specific image recognition task for which they were developed.
- Easily Accessible: The model weights are provided as free downloadable files and many libraries provide convenient APIs to download and use the models directly.

How to Use Pre-Trained Models

- The use of a pre-trained model is limited only by our creativity.
- Ex: a model may be downloaded and used as-is, such as embedded into an application and used to classify new photographs.
- Alternately, models may be downloaded and use as feature extraction models. Here, the output of the model from a layer prior to the output layer of the model is used as input to a new classifier model.
- Recall that convolutional layers closer to the input layer of the model learn low-level features such as lines, that layers in the middle of the layer learn complex abstract features that combine the lower level features extracted from the input, and layers closer to the output interpret the extracted features in the context of a classification task.
- So a level of detail for feature extraction from an existing pre-trained model can be chosen. For example, if a new task is quite different from classifying objects in photographs (e.g. different to ImageNet), then perhaps the output of the pre-trained model after the few layers would be appropriate. If a new task is quite similar to the task of classifying objects in photographs, then perhaps the output from layers much deeper in the model can be used, or even the output of the fully connected layer prior to the output layer can be used.
- The pre-trained model can be used as a separate feature extraction program, in which case input can be pre-processed by the model or portion of the model to a given an output (e.g. vector of numbers) for each input image, that can then use as input when training a new model.
- Alternately, the pre-trained model or desired portion of the model can be integrated directly into a new neural network model. In this usage, the weights of the pre-trained can be frozen so that they are not updated as the new model is trained. Alternately, the weights may be updated during the training of the new model, perhaps with a lower learning rate, allowing the pre-trained model to act like a weight initialization scheme when training the new model.

How to Use Pre-Trained Models

Summary of usage patterns

1. Classifier: The pre-trained model is used directly to classify new images.
2. Standalone Feature Extractor: The pre-trained model, or some portion of the model, is used to pre-process images and extract relevant features.
3. Integrated Feature Extractor: The pre-trained model, or some portion of the model, is integrated into a new model, but layers of the pre-trained model are frozen during training.
4. Weight Initialization: The pre-trained model, or some portion of the model, is integrated into a new model, and the layers of the pre-trained model are trained in concert with the new model.

- Each approach can be effective and save significant time in developing and training a deep convolutional neural network model.
- It may not be clear as to which usage of the pre-trained model may yield the best results on your new computer vision task, therefore some experimentation may be required.

Models for Transfer Learning

- There are top-performing models for image recognition that can be downloaded and used as the basis for image recognition and related computer vision tasks.
- Popular models
- VGG (e.g. VGG16 or VGG19)
- GoogLeNet (e.g. InceptionV3)
- Residual Network (e.g. ResNet50)
- These models are widely used for transfer learning
  - because of their performance,
  - because they were examples that introduced specific architectural innovations, namely consistent and repeating structures (VGG), inception modules (GoogLeNet), and residual modules (ResNet)

Where to find pretrained models

Location	What's there?	Link(s)
PyTorch domain libraries	Each of the PyTorch domain libraries (torchvision, torchtext) come with pretrained models of some form. The models there work right within PyTorch.	torchvision.models, torchtext.models, torchaudio.models, torchrec.models
HuggingFace Hub	A series of pretrained models on many different domains (vision, text, audio and more) from organizations around the world. There's plenty of different datasets too.	https://huggingface.co/models, https://huggingface.co/datasets
timm (PyTorch Image Models) library	Almost all of the latest and greatest computer vision models in PyTorch code as well as plenty of other helpful computer vision features.	https://github.com/rwightman/pytorch-image-models
Paperswithcode	A collection of the latest state-of-the-art machine learning papers with code implementations attached. You can also find benchmarks here of model performance on different tasks.	https://paperswithcode.com

What is a state_dict in PyTorch?

- The state_dict is a python dictionary object which is used for saving or loading models from PyTorch.
- A state_dict maps each layer to its parameter tensors. The learnable parameters of a model (convolutional layers, linear layers, etc.) and registered buffers (BatchNorm's running_mean) have entries in state_dict
- the weights and biases or the learnable parameters of neural networks or "torch.nn.Module" model are contained in the models parameters which are accepted by model.parameter() function
- Optimizer objects (torch.optim) also have a state_dict, which contains information about the optimizer's state, as well as the hyperparameters used.
- the dictionary i.e state_dict maps the each layer to its parameter tensor.
- These dictionary object can be easily updated, saved, altered, and restored adding a great deal of modularity to the PyTorch models and the optimizers as well.

Accessing state_dict and checkpoints

1. Accessing state_dict
2. Save and Load the Entire PyTorch Model
3. Use state_dict To Save And Load PyTorch Models
4. Save and Load PyTorch Model From a Checkpoint

Steps in accessing state_dict

1. Import library
2. Define and Initialize Neural network
3. Initializing optimizer
4. Accessing Model’s state_dict
5. Accessing Optimizer’s state_dict

Steps in accessing state_dict

- Step 1 - Import library

```
import torch
import torch.nn as nn
import torch.optim as optim
```

Steps in accessing state_dict

- Step 2 - Define and Initialize Neural network

```
class Neuralnet(nn.Module):
    def __init__(self):
        super(Neuralnet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def pass_forward(self, X_data):
        X_data = self.pool(F.relu(self.conv1(X_data)))
        X_data = self.pool(F.relu(self.conv2(X_data)))
        X_data = X_data.view(-1, 16 * 5 * 5)
        X_data = F.relu(self.fc1(X_data))
        X_data = F.relu(self.fc2(X_data))
        X_data = self.fc3(X_data)
        return X_data
network = Neuralnet()
print("This is our neural network parameters:",network)
```

Steps in accessing state_dict

- Step 3 - Initializing optimizer

```
state_optim = optim.SGD(network.parameters(), lr=0.01, momentum=0.9)
```

- Step 4 - Accessing Model's state_dict

```
print("Accessing the model state_dict")
for values in network.state_dict():
    print(values, "\t", network.state_dict()[values].size())
```

- Step 5 - Accessing Optimizer's state_dict

```
print("Accessing the optimizers state_dict")
for elements in state_optim.state_dict():
  print(elements, "\t", state_optim.state_dict()[elements])
```

Steps in accessing state_dict - Output

```
This is our neural network parameters: Neuralnet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
 Accessing the model state_dict
 conv1.weight     torch.Size([6, 3, 5, 5])
 conv1.bias       torch.Size([6])
 conv2.weight     torch.Size([16, 6, 5, 5])
 conv2.bias       torch.Size([16])
 fc1.weight       torch.Size([120, 400])
 fc1.bias         torch.Size([120])
 fc2.weight       torch.Size([84, 120])
 fc2.bias         torch.Size([84])
 fc3.weight       torch.Size([10, 84])
 fc3.bias         torch.Size([10])

Accessing the optimizers state_dict
state    {}
param_groups     [{'lr': 0.01, 'momentum': 0.9, 'dampening': 0, 'weight_decay': 0, 'nesterov': False, 'maximize': False, 'foreach': None,
'differentiable': False, 'params': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}]
```

Steps in accessing state_dict – Example of MNIST_CNN Program

```
# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)


# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict().keys():
    print(param_tensor, "\t",model.state_dict()[param_tensor].size())

print()

# Print optimizer's state_dict
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])

EPOCHS = 2
for epoch in range(EPOCHS):
```

Save and Load the Entire PyTorch Model - Syntax

- Save and Load the Entire PyTorch Model
- Save the entire model in PyTorch and not just the state_dict.
- However, this is not a recommended way of saving the model.
- Save
- torch.save(model, 'save/to/path/model.pt’)
- Load
- model = torch.load('load/from/path/model.pt’)
- Problem Statement 1- Use existing MNIST_CNN model as a pre-trained program to Save and Load the Entire PyTorch Model

Problem Statement 1- Use existing MNIST_CNN model as a pre-trained program

- Use existing MNIST_CNN model as a pre-trained program to perform classification on FashionMNIST

- Fashion-MNIST shares the same image size, data format and the structure of training and testing splits with the original MNIST.
- Steps - MNIST_CNN.py :

1. Use state_dict to save model parameters and Optimizer information.
2. Re-run the MNIST_CNN.py program by appending the following command
   torch.save(model, "./ModelFiles/model.pt") at the end.

Note: Make sure ModelFiles folder exists in the current working directory.

Steps in accessing state_dict – Example of MNIST_CNN Program

```
# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)


# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict().keys():
    print(param_tensor, "\t",model.state_dict()[param_tensor].size())

print()

# Print optimizer's state_dict
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])

EPOCHS = 2
for epoch in range(EPOCHS):
```

Problem Statement 1- Use existing MNIST_CNN model as a pre-trained program

- Steps – Create a new file - FashionMNIST_CNN.py :
- Using the pretrained model for inference (FashionMNIST_CNN.py):
- Step 1: Import libraries same as in MNIST_CNN.py

- Step 2: Define the class with same as in MNIST_CNN.py

- Step 3: There is no need to train the model hence use only testloader.
- mnist_testset = datasets.FashionMNIST(root='./data', train=False, download=True, transform = ToTensor())
- test_loader = DataLoader(mnist_testset, batch_size=batch_size, shuffle=False)

Problem Statement 1- Use existing MNIST_CNN model as a pre-trained program

- Steps - FashionMNIST_CNN.py :
- Using the pretrained model for inference (FashionMNIST_CNN.py):
- Step 4: Load the pretrained model on to the device
- device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

- **# Load the model to GPU**

```
model = CNNClassifier()
model = torch.load("./ModelFiles/model.pt")
model.to(device)
```

Problem Statement 1- Use existing MNIST_CNN model as a pre-trained program

- Steps - FashionMNIST_CNN.py :
- Steps 6 & 7 are same as in MNIST_CNN
- Step 6: Print the model state dictionary. (Same can be done in MNIST_CNN.py)
- # Print model's state_dict- only the size of the parameter

```
print("Model's state_dict:")
for param_tensor in model.state_dict().keys():
    print(param_tensor, "\t",model.state_dict()[param_tensor].size())
print()
```

Problem Statement 1- Use existing MNIST_CNN model as a pre-trained program

- Steps - FashionMNIST_CNN.py :

Step 7: Evaluate the model

```
model.eval()
correct = 0
total = 0
for i, vdata in enumerate(test_loader):
    tinputs, tlabels = vdata
    tinputs = tinputs.to(device)
    tlabels = tlabels.to(device)
    toutputs = model(tinputs)
    #Select the predicted class label which has the
    # highest value in the output layer
    _, predicted = torch.max(toutputs, 1)
    print("True label:{}".format(tlabels))
    print('Predicted: {}'.format(predicted))
    # Total number of labels
    total += tlabels.size(0)

    # Total correct predictions
    correct += (predicted == tlabels).sum()

accuracy = 100.0 * correct / total
print("The overall accuracy is {}".format(accuracy))
```

Use state_dict To Save And Load PyTorch Models – Syntax

Use state_dict To Save And Load PyTorch Models

- Save

```
torch.save(model.state_dict(), 'save/to/path/model.pth’)
```

- Load
- model = MyModel (args)

```
model.load_state_dict(torch.load('load/from/path/model.pth’))
```

Note: Using state_dict To Save And Load PyTorch Models is applied in checkpoints

Managing a PyTorch Training Process with Checkpoints

- A large deep learning model can take a long time to train.
- We lose a lot of work if the training process interrupted in the middle.
- But sometimes, we actually want to interrupt the training process in the middle if going any further would not give a better model.
- We can control the training loop such that we can resume an interrupted process, or early stop the training loop using checkpoints.

Managing a PyTorch Training Process with Checkpoints

- Checkpoint is done to resume training from the last or best checkpoint.
- It is also a safeguard in case the training gets disrupted due to some unforeseen issue.
- Saving following information that would require to resume training using a checkpoint :
  - the model's state_dict
  - optimizer's state_dict,
  - last epoch number,
  - Loss

Saving the checkpoint – Syntax & Usage

Save

```
torch.save({'epoch': EPOCH,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': LOSS},
            'save/to/path/model.pth')
```

```
#Save the check point
check_point ={"last_loss":avg_loss, "last_epoch":EPOCHS, "model_state":model.state_dict(), "optimizer_state":optimizer.state_dict()}
torch.save(check_point,"./checkpoints/checkpoint.pt")
```

Loading the checkpoint – Syntax & Usage

Load

```
model = MyModelDefinition(args)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

checkpoint = torch.load('load/from/path/model.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

```
#Reload from the checkpoint. This can be done in another file as well
check_point = torch.load("./checkpoints/checkpoint.pt")
model.load_state_dict(check_point["model_state"])
```

```
optimizer.load_state_dict(check_point["optimizer_state"])
loss = check_point["last_loss"]
EPOCHS = check_point["last_epoch"]
```

Problem Statement 2- Implement check points

- Implement check points in PyTorch by saving
  - model state_dict,
  - optimizer state_dict,
  - epochs and
  - loss during training
- So that the training can be resumed at a later point.
- Illustrate the use of check point to save the best found parameters during training.
- Use the original source program renamed as MNIST_CNN_Checkpoint.py

Problem Statement 2- Implement check points

- Step 1:
- Use state_dict to save model parameters and Optimizer information.

```
loss_fn = torch.nn.CrossEntropyLoss()

# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict().keys():
    print(param_tensor, "\t",model.state_dict()[param_tensor].size())
print()

# Print optimizer's state_dict
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])

EPOCHS = 2
for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch + 1))
```

Problem Statement 2- Implement check points

- Step 2:
- Re-run the MNIST program by appending the following command at the end.
- #Save the check point
- check_point ={"last_loss":avg_loss, "last_epoch":EPOCHS, "model_state":model.state_dict(), "optimizer_state":optimizer.state_dict()}
- torch.save(check_point,"./checkpoints/checkpoint.pt")
- Here, a checkpoint has been established soon after two epochs.
- Note: Make sure checkpoints folder exists in the current working directory.

Problem Statement 2- Implement check points

- Resuming the model for training
- Create a new file MNIST_CNN_Use_Checkpoint.py:

- The earlier checkpoint is loaded now for resuming the training loop to run for remaining number of epochs as shown below.

Problem Statement 2- Implement check points

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNNClassifier().to(device)
#Reload from the checkpoint. This can be done in another file as well.
check_point = torch.load("./checkpoints/checkpoint.pt")
model.load_state_dict(check_point["model_state"])

# add the criterion which is the MSELoss
loss_fn = torch.nn.CrossEntropyLoss()

# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)


optimizer.load_state_dict(check_point["optimizer_state"])
loss = check_point["last_loss"]
EPOCHS = check_point["last_epoch"]

NEW_EPOCHS = 5
for epoch in range(EPOCHS, NEW_EPOCHS):
```

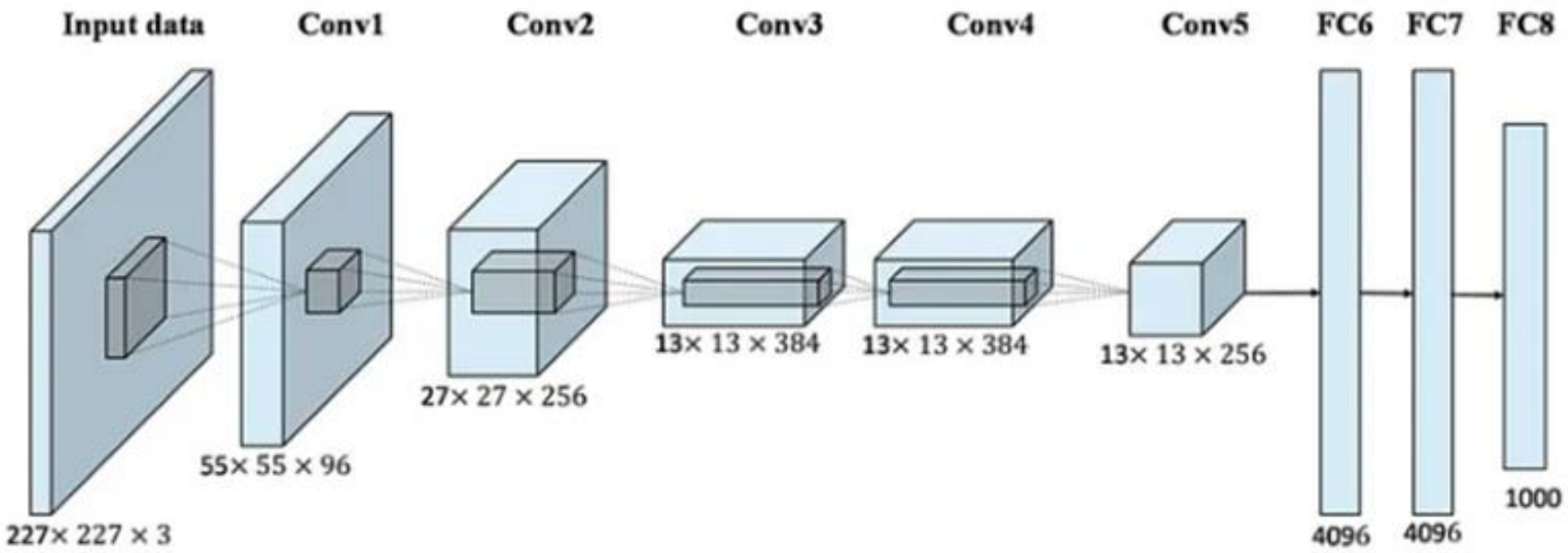
Checkpoints - Demerits

- checkpoints could add an unnecessary overhead to a session such as
  - costs related to memory usage and
  - more time for training (as checkpoints saving also consumes a fair amount of time).
- So, It is advisable to create checkpoints only when the model best performs at a particular epoch.

AlexNet

- AlexNet is a deep convolutional neural network, initially developed by Alex Krizhevsky and his colleagues back in 2012.
- It was designed to classify images for the ImageNet LSVRC-2012 competition where it achieved state of the art results.
- it operated with 3-channel images that were (224x224x3) in size.
- It used max pooling along with ReLU activations when subsampling.
- The kernels used for convolutions were either 11x11, 5x5, or 3x3
- kernels used for max pooling were 3x3 in size.
- It classified images into 1000 classes. It also utilized multiple GPUs.

AlexNet Architecture



AlexNet path

```
model_urls = {
    'alexnet': 'https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth',
}
```

https://pytorch.org/vision/stable/models.html

AlexNet Class definition

```
class AlexNet(nn.Module):

    def __init__(self, num_classes: int = 1000) -> None:
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Load Alexnet as a pretrained model

- How to load Alexnet as a pretrained model – 2 ways

```
from torchvision.models import AlexNet_Weights
model = torch.hub.load('pytorch/vision:v0.10.0', model='alexnet', weights=AlexNet_Weights.DEFAULT)
```

```
from torchvision.models import alexnet
model = alexnet(pretrained=True)
```

Input Image preprocess

```
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

AlexNet

- Once dataset has been downloaded onto the Data directory we need to properly annotate Dog and Cat Pictures.
- We find that there are two folders train and validation which holds our Training and validation Dataset.
- These contain Image Names which we will be using to annotate our Dataset.
- Need to manipulate and parse file and directory paths, to programmatically access data files
- Ex: use the os and glob packages to access files and directories and to create lists of paths which can be used to parse to extract useful information from the file and directory names

glob – to return list of all files in the directory

- glob helps to filter through large datasets and pull out only files that are of interest.
- The glob() function uses the rules of Unix shell to organize files
- glob uses different operators to broaden its searching abilities. The primary operator is *.
- The * is a wildcard that can be used to search for items that have differences in their names. Whatever text does not match can be replaced by a *.
- Ex: if we want every file in a directory to be returned, we can put a * at the end of a directory path.
- glob will return a list of all of the files in that directory.

```
class MyDataset(Dataset):
        def __init__(self, transform, str="train"):
            self.imgs_path = ".\\data\\cats_and_dogs_filtered\\"+ str + "\\"
            file_list = glob.glob(self.imgs_path + "*")
            print("File list",file_list)
```

Annotating and mapping

```
class MyDataset(Dataset):
    def __init__(self, transform, str="train"):
        self.imgs_path = ".\\data\\cats_and_dogs_filtered\\"+ str + "\\"
        file_list = glob.glob(self.imgs_path + "*")
        print("File list",file_list)
        self.data = []
        for class_path in file_list:
            class_name = class_path.split("\\")[-1]
            for img_path in glob.glob(class_path + "\\*.jpg"):
                self.data.append([img_path, class_name])
        print(self.data)
        self.class_map = {"dogs" : 0, "cats": 1}
        self.transform = transform

    def __len__(self):
        return len(self.data)
```

Import libraries

Step 1: Import libraries

```
import PIL.Image
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
from torchvision import transforms
import glob
from torchvision.models import AlexNet_Weights
```

Alexnet as feature extractor

- ImageNet Dataset has 1000 Output Classes while Dog-Cat dataset has only 2.
- We can achieve this by changing the last layer of the model and freezing the pre-trained model by setting the variables to non-trainable.

```
#Use the feature extractor as it is
for param in model.features.parameters():
        param.requires_grad = False
#Modify the classification head
# Changing only the last layer of classification head
num_ftrs = model.classifier[6].in_features
model.classifier[6] = nn.Linear(num_ftrs, 2)
# Load AlexNet model with pre-trained weights
```