Ultimate Guide to Fine-Tuning in PyTorch : Part 1 — Pre-trained Model and Its Configuration

Ruman · Follow
16 min read · Jul 17, 2023

353 5

Image source : dilithjay.com

When it comes to training deep learning models today, transfer learning through fine-tuning a pre-trained model on your own data has become the go-to approach. By fine-tuning these models, we can leverage their expertise and adapt them to our specific tasks, saving valuable time and computational resources.

This article is divided into four parts, with each part focusing on different aspects of fine-tuning models.

Part 1 : We will delve into defining a pre-trained model and configuring it to suit your target task.

Part 2 : The second part will explore various techniques to enhance the accuracy of your fine-tuned model.

Part 3 : Moving on to Part Three, we will cover the process of Defining Data and Applying Transformations tailored specifically to your target task.

Part 4 : Finally, in the last of this series, we'll address Model Training Observability, including which metrics to track during training and how to effectively manage model checkpoints, among other important aspects. **(Coming Soon)**

Discover all the articles from this series conveniently listed in one place:

Ruman

Ultimate Guide to Fine-Tuning in PyTorch

View list 4 stories

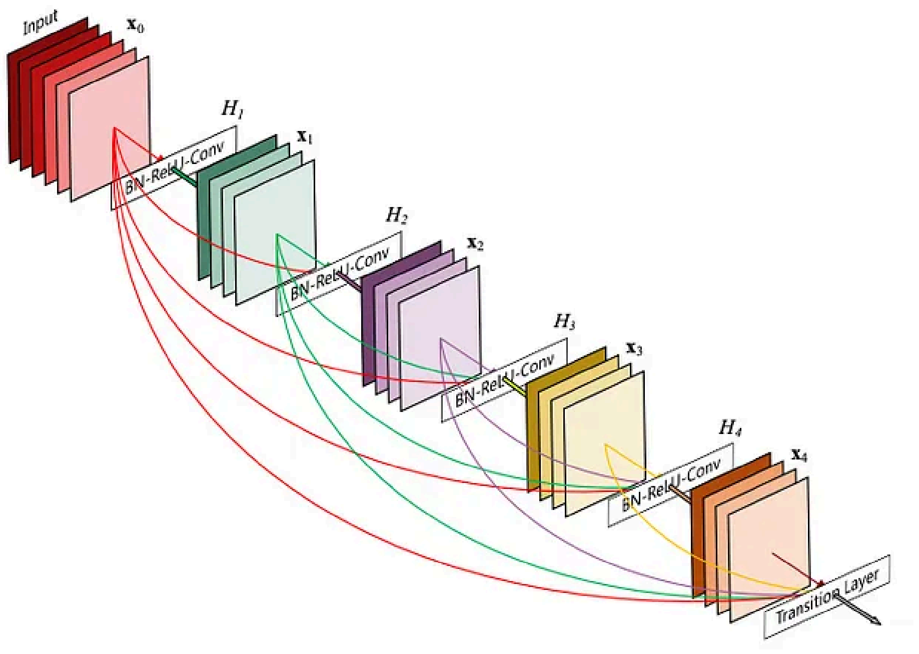
Outlines

- Introduction — The Model and Its Configuration
- Loading a pre-trained model
- Modifying model head
- Setting Learning Rate, Optimizer and Weight Decay
- Choosing Loss Function
- Freezing Full or Partial network
- Define Model Floating-point precision
- Training and Validation Mode
- Single GPU and Multi GPU
- Conclusion

Introduction

Defining a model includes a range of important decisions, including selecting the appropriate architecture, customizing the model head, configuring the loss function and learning rate, setting the desired floating-point precision, and determining which layers to freeze or fine-tune, and many more. In this article, we will explore each of these aspects in detail, providing valuable insights to help you effectively define and fine tune your model.

Loading a pre-trained model



Example of a DensNet architecture

Before loading a pre-trained model, it is crucial to have a clear understanding of your specific problem and choose an appropriate architecture accordingly. While this task may seem challenging, it is important not to randomly select a model architecture. Consider the requirements of your business and choose a suitable architecture that aligns with those needs.

For instance, if you are fine-tuning for classification and low latency is a priority, architectures like MobileNet would be a favourable choice. By making informed architecture decisions, you can optimize your fine-tuning experiments for better outcomes.

> ***Please note that*** *— There are several sources from where you can load pre-trained models for fine-tuning. In this article I'm referring to* ***timm****(Pytorch Image Models) and* ***Torchvision models***

Here's an example to load a pre-trained resnet50 model from **torchvision** :

```
# From torchvision.models
from torchvision import models
model = models.resnet50(pretrained=False)
```

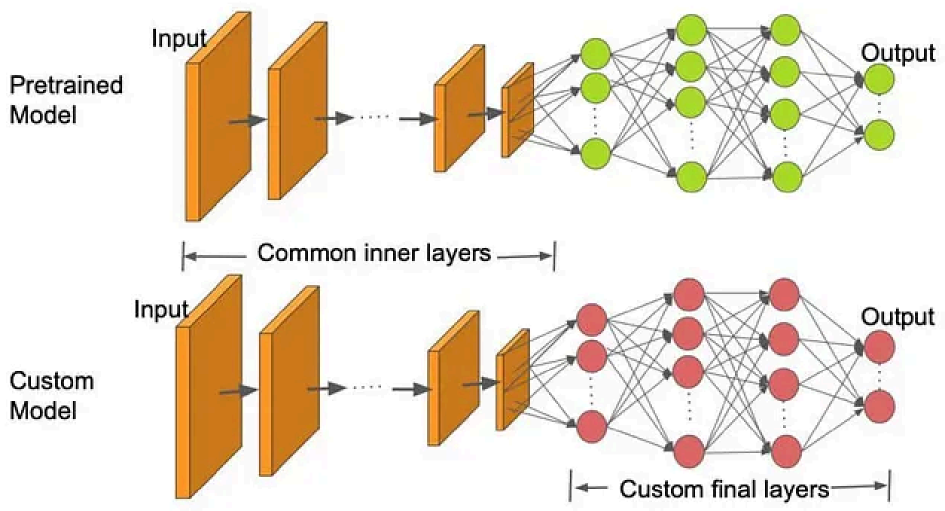
Load from **timm(Pytorch Image Models)** :

```
import timm

# from timm
pretrained_model_name = "resnet50"
model = timm.create_model(pretrained_model_name, pretrained=False)
```

It is important to note that regardless of the source of the pre-trained model, the key modification required is to adjust the fully connected FC layer(or could be linear/classifier/head) of the model. Additionally, for your target task, you may incorporate extra linear layers. We'll explore this further in the next section.

Modifying model head



Visual depiction of a pre-trained model adapted for the target task, featuring additional layers beyond the head modification. Source: OpenCV

Modifying the model's head is essential to align it with your specific target task. Pre-trained models are trained on large datasets like ImageNet for image classification or on text data like BooksCorpus and Wikipedia for text generation. By modifying the model's head, the pre-trained model can adapt to the new task and utilise the valuable features it has learned, enhancing its performance in the new task.

For example, you can modify RestNet head for classification task:

```
import torch.nn as nn
import timm

num_classes = 4 # Replace num_classes with the number of classes in your data

# Load pre-trained model from timm
model = timm.create_model('resnet50', pretrained=True)

# Modify the model head for fine-tuning
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, num_classes)
```

Or you can modify RestNet head for classification task along with **adding extra linear layers** to enhance the model's predictive power (ps — this is merely an illustrative example):

```
import torch.nn as nn
import timm

num_classes = 4 # Replace num_classes with the number of classes in your data

# Load pre-trained model from timm
model = timm.create_model('resnet50', pretrained=True)

# Modify the model head for fine-tuning
num_features = model.fc.in_features

# Additional linear layer and dropout layer
model.fc = nn.Sequential(
    nn.Linear(num_features, 256),  # Additional linear layer with 256 output fea
    nn.ReLU(inplace=True),         # Activation function (you can choose other a
    nn.Dropout(0.5),               # Dropout layer with 50% probability
    nn.Linear(256, num_classes)    # Final prediction fc layer
)
```

Or you can modify RestNet head for regression task:

```
model = timm.create_model('resnet50', pretrained=True)

# Modify the model head for regression
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 1) # Regression task has a single output
```

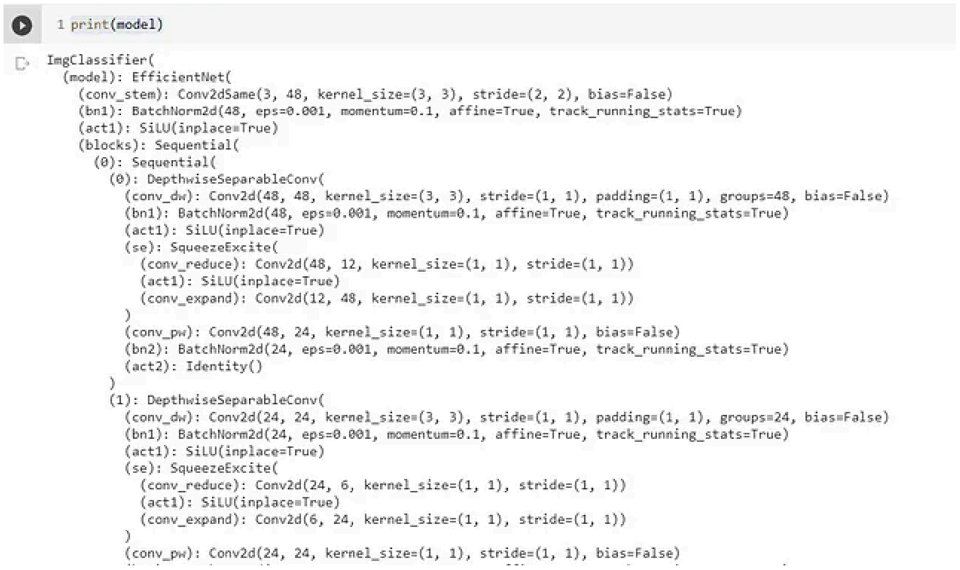
One thing to note that — the model does not always have a FC (fully connected) layer that we modify output features(e.g, num_classes). The architecture of models can vary, and the name and position of the layer we need to modify can differ.

In many pre-trained models, especially in convolutional neural network (CNN) architectures, there is often a linear layer or FC layer at the end of the model that performs the final classification. However, this is not a strict rule, and some models may have a different structure.

To identify the layer that needs modification, you can do, for example:

```
import torch
import timm

# Load pre-trained model from timm
model = timm.create_model('resnet50', pretrained=True)
print(model)
```



```
print(model)
```

```
ImgClassifier(
  (model): EfficientNet(
    (conv_stem): Conv2dSame(3, 48, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn1): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    (act1): SiLU(inplace=True)
    (blocks): Sequential(
      (0): Sequential(
        (0): DepthwiseSeparableConv(
          (conv_dw): Conv2d(48, 48, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=48, bias=False)
          (bn1): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (act1): SiLU(inplace=True)
          (se): SqueezeExcite(
            (conv_reduce): Conv2d(48, 12, kernel_size=(1, 1), stride=(1, 1))
            (act1): SiLU(inplace=True)
            (conv_expand): Conv2d(12, 48, kernel_size=(1, 1), stride=(1, 1))
          )
          (conv_pw): Conv2d(48, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn2): BatchNorm2d(24, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (act2): Identity()
        )
        (1): DepthwiseSeparableConv(
          (conv_dw): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=24, bias=False)
          (bn1): BatchNorm2d(24, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
          (act1): SiLU(inplace=True)
          (se): SqueezeExcite(
            (conv_reduce): Conv2d(24, 6, kernel_size=(1, 1), stride=(1, 1))
            (act1): SiLU(inplace=True)
            (conv_expand): Conv2d(6, 24, kernel_size=(1, 1), stride=(1, 1))
          )
          (conv_pw): Conv2d(24, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

Printed model architecture showing identification of the specific last layer requiring modification. Source: kakaocdn.net

By printing the model, you can see its architecture and identify the appropriate layer to modify. Look for a linear or FC layer that serves as the final classification layer, and replace it with a new layer that matches the number of classes or your task requirement.

Setting Optimizer, Learning Rate, Weight Decay, and Momentum

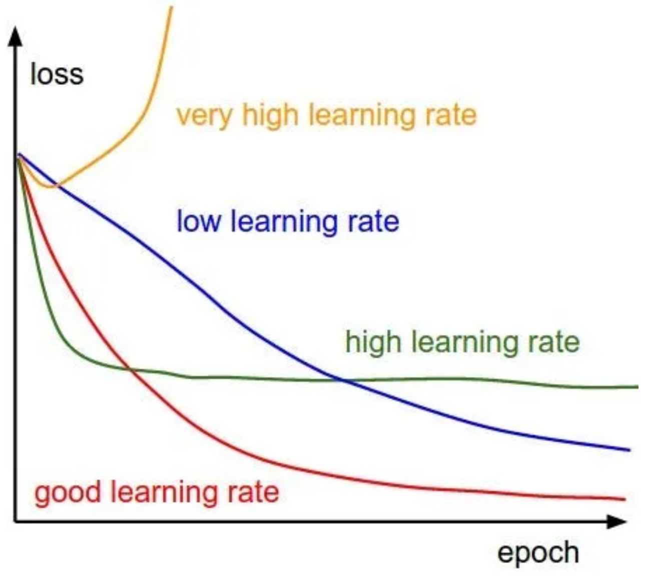


Image source

In fine-tuning, the learning rate, loss function, and optimizer are interrelated components that collectively influence the model's ability to adapt to the new task while leveraging the knowledge acquired from pre-training. A well-chosen learning rate ensures effective model convergence at a reasonable speed, a carefully selected loss function aligns the loss minimization during training process with the target task, and an appropriate optimizer optimizes the model's parameters effectively.

Fine-tuning demands careful experimentation and iterative adjustment of these components to strike the right balance and achieve the desired level of performance in the fine-tuned model.

Optimizers

> *For a more detailed understanding of selecting the right optimizer for fine-tuning, I recommend referring to the blog https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e*

The optimizer determines the algorithm to be used to to update the model's parameters based on the gradients computed during backpropagation. Different optimizers, such SGD, Adam, or RMSprop, have distinct paramters update rules and convergence properties. The choice of optimizer can significantly impact the model training and the final performance of the fine-tuned model. Selecting the most appropriate optimizer involves considering factors like the nature of the task, the size of the dataset, and the computational resources available.

Learning Rate, Momentum and Weight Decay

While defining the optimizer we also have to set the learning rate(LR) which is a hyperparameter that determines the step size at each iteration during optimization. It controls how much the model's parameters are updated in response to the calculated gradients during backpropagation. Choosing an appropriate learning rate is crucial, as setting it too high may cause the optimization process to oscillate or diverge or overshoot the optimal solution, while setting it too low may result in slow convergence or getting trapped in local minima.

In addition to the learning rate, there are other crucial hyperparameters to consider when defining the optimizer, such as weight decay and momentum (**specific to SGD**). Let's take a quick look at both of these hyperparameters:

- Weight decay, also known as L2 regularization, is a technique used to prevent overfitting and encourage the model to learn simpler, more generalizable representations.
- Momentum is used in Stochastic Gradient Descent (SGD), to accelerate convergence and escape local minima.

```
import torch.optim as optim

# Define your optimizer with weight decay
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=0.

# Define your optimizer with weight decay
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.0001)
```

Choosing Loss functions

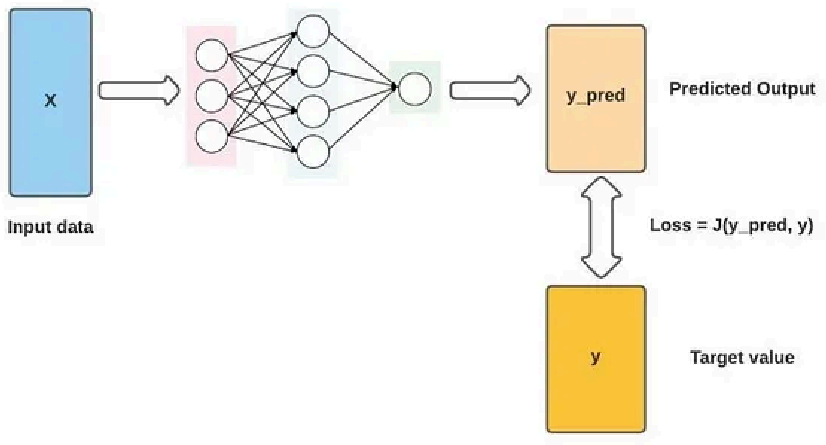


Image source : naptune.ai

> *To explore and select a suitable loss function for your target task, I recommend referring to the official PyTorch documentation on loss functions.*

The loss function measures the difference or gap between the model's predicted outputs and the actual correct answers. It gives us a way to understand how well the model is performing on the task. When fine-tuning a pre-trained model, it's important to choose a loss function that suits the specific task we're working on. For example, for classification tasks, cross-entropy loss is commonly used, while mean squared error is more suitable for regression problems. Selecting the right loss function ensures that the model focuses on optimizing the desired objective during training.

```
import torch.nn as nn

# Define the loss function - For classification problem
loss_function = nn.CrossEntropyLoss()

# Define the loss function - For regression problem
loss_function = nn.MSELoss()  # Mean Squared Error loss
```

Also to note that there several additional considerations and techniques that can be applied regarding the choice and handling of the loss function. Some example of such are :

- **Custom Loss function —** You may need to modify or customize the loss function to suit specific requirements. One example is incorporating a 10x penalisation for misclassification on an important individual class. Below is an example code that demonstrates the implementation of custom loss to provide you with an idea of how it can be done :

```
import torch
import torch.nn.functional as F

class CustomLoss(torch.nn.Module):
    def __init__(self, class_weights):
        super(CustomLoss, self).__init__()
        self.class_weights = class_weights

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(inputs, targets, reduction='none')
        weights = torch.ones_like(targets).float()
        for class_idx, weight in enumerate(self.class_weights):
            weights[targets == class_idx] = weight
        weighted_loss = ce_loss * weights
        return torch.mean(weighted_loss)

# Assuming you have a model and training data
model = YourModel()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
# Assuming 5 classes so class_weights = [1.0, 1.0, 1.0, 1.0, 10.0]
```

```
criterion = CustomLoss(class_weights=[1.0, 1.0, 1.0, 1.0, 10.0])

# Inside the training loop
optimizer.zero_grad()
outputs = model(inputs)
loss = criterion(outputs, targets)
loss.backward()
optimizer.step()
```

- **Metric-Based Loss —** In some cases, the performance of the model might be evaluated based on metrics other than the loss itself. In such cases, you can design or adapt the loss function to directly optimize for these metrics.
- **Regularization —** Regularization methods, such as L1 or L2 regularization, can be incorporated into the loss function during fine-tuning to prevent overfitting and improve model generalization. Regularization terms can help control the complexity of the model and reduce the risk of overemphasising specific patterns or features in the data. L2 regularization can be applied by setting the `weight_decay` value in the optimizer, while L1 regularization requires a slightly different

L2 regularization implementation :

```
# Define a loss function
criterion = nn.CrossEntropyLoss()

# L2 regularization
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=0.01)
```

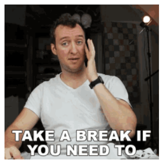
L1 Regularization implementation :

```
# Define a loss function
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Inside the training loop
optimizer.zero_grad()
outputs = model(inputs)
loss = criterion(outputs, targets)

# L1 regularization
regularization_loss = 0.0
for param in model.parameters():
    regularization_loss += torch.norm(param, 1)
loss += 0.01 * regularization_loss#Adjust regularization strength as needed
```

Also one more thing 😁



Source : tenor.com

Freezing Full or Partial network

Image source : www.therescuevets.com

When we're referring to freezing that means fixing the weight of specific layer or entire network during fine tuning process. Netwrok freezing allows us to retain the knowledge captured by the pre-trained model while only updating certain layers to adapt to the target task. So it's very crucial and this should not be ignored if you're fine tuning a pre-trained model.

Deciding whether you should freeze all the layers(full network) or partial layers of the pre-trained model before fine tuning it all boils down to your specific target task.

For example, if the pre-trained model has been trained on a large-scale dataset similar to the target task, freezing the entire network can help preserve the learned representations, preventing them from being overwritten. In this case, only the model's head is modified and trained from scratch.

On the other hand, freezing only a portion of the network can be useful when the pre-trained model's lower layers capture general features that are likely to be relevant for the new task. By freezing these lower layers, we can leverage the pre-trained model's knowledge while updating the higher layers to specialize in the task-specific features. This approach is particularly useful in scenarios where the target dataset is small or significantly different from the dataset on which the pre-trained model was trained.

To implement freezing in PyTorch, you can access individual layers or modules within the model and set their `requires_grad` attribute to `False`. This prevents the gradients from being computed and the weights from being updated during the backward pass.

Below is an example code that demonstrates the implementation of **freezing entire network** :

```
# Freeze all the layers of the pre-trained model
for param in model.parameters():
    param.requires_grad = False

# Modify the model's head for a new task
num_classes = 10
model.fc = nn.Linear(model.fc.in_features, num_classes)
```

Freezing **only convolutional layers** in the network :

```
# Freeze only the convolutional layers of the pre-trained model
for param in model.parameters():
    if isinstance(param, nn.Conv2d):
        param.requires_grad = False

# Modify the model's head for a new task
num_classes = 10
model.fc = nn.Linear(model.fc.in_features, num_classes)
```

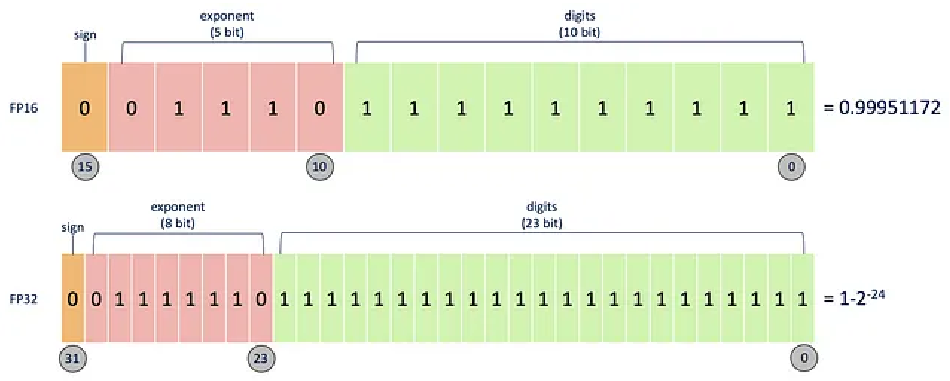
Freezing **only specific layers** in the network :

```
# Freeze specific layers (e.g.,the first two convolutional layers) of the pre-tr
for name, param in model.named_parameters():
    if 'conv1' in name or 'layer1' in name:
        param.requires_grad = False

# Modify the model's head for a new task
num_classes = 10
model.fc = nn.Linear(model.fc.in_features, num_classes)
```

It's important to note that freezing layers should be done thoughtfully, considering the specific requirements of your task and dataset. It's a delicate balance between leveraging the pre-trained knowledge and allowing the model to adapt to the new task effectively.

Define Model Floating-point precision



Understanding Mixed Precision Training by Jonathan Davis

To quickly sum up the definition model floating point precision refers to the data type used to represent numerical values during computation in deep learning model. In PyTroch 32-bit (float32 or FP32) and 16-bit (float16 or FP16 or Half precision) are two commonly used floating-point precision.

- float32 — This precision provides a wide dynamic range and high numerical precision, allowing for accurate computations but consuming more memory. FP32 uses 32 bits to represent a number.
- float16 — This lower precision can reduce the memory footprint and computational requirements of the model, leading to potential gains in efficiency and speed. However, it may result in a loss of numerical precision and can impact the model's accuracy or convergence. FP16 uses 16 bits to represent a number.

FP16 and FP32 are referred as **Single Precision** and both comes with it's own set of pros and cons as we saw in above pointers. To leverage advantages of both we have **Mixed Precision** which combines FP16 and FP32 floating point precision within training pipeline. Mixed precision offers improved computational efficiency, reduced memory footprint, accelerated training, and increased model capacity.

> *For a more detailed understanding of mixed precision training, I recommend referring to the article "Understanding Mixed Precision Training"*

Here's an example of how to implement mixed precision training in a PyTorch training pipeline using the Automatic Mixed Precision (AMP) library:

```
import torch
from torch import nn, optim
from torch.cuda.amp import autocast, GradScaler

# Define your model and optimizer
model = YourModel()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Define loss function
criterion = nn.CrossEntropyLoss()

# Define scaler for automatic scaling of gradients
scaler = GradScaler()

# Define your training loop
for epoch in range(num_epochs):
    for batch_idx, (data, targets) in enumerate(train_loader):
        data, targets = data.to(device), targets.to(device)

        # Zero the gradients
        optimizer.zero_grad()

        # Enable autocasting for mixed precision
        with autocast():
            # Forward pass
            outputs = model(data)
            loss = criterion(outputs, targets)

        # Perform backward pass and gradient scaling
        scaler.scale(loss).backward()

        # Update model parameters
        scaler.step(optimizer)
        scaler.update()

        # Print training progress
        if batch_idx % log_interval == 0:
            print(f"Epoch {epoch+1}/{num_epochs} | Batch {batch_idx}/{len(train_
```

In the above example the `GradScaler()` object is used to perform gradient scaling. Here's a breakdown of the methods used:

1. `scaler.scale(loss)` : This method scales the loss value by the appropriate factor determined by the scaler. It returns a scaled loss that will be used for the backward pass.
2. `scaler.step(optimizer)` : This method updates the optimizer's parameters using the gradients calculated during the backward pass. It performs an optimizer step as usual but takes into account the gradient scaling performed by the scaler.

3. `scaler.update()` : This method adjusts the scale factor used by the scaler for the next iteration. It helps **prevent underflow or overflow** issues by dynamically adjusting the scale based on the gradients magnitude. It is called after `scaler.step(optimizer)` .

The purpose of using `GradScaler()` and the associated methods is to mitigate potential numerical instability issues that may arise when using lower-precision (FP16) computations. By scaling the loss and gradients appropriately, the scaler ensures that the optimizer's updates remain in a stable range.

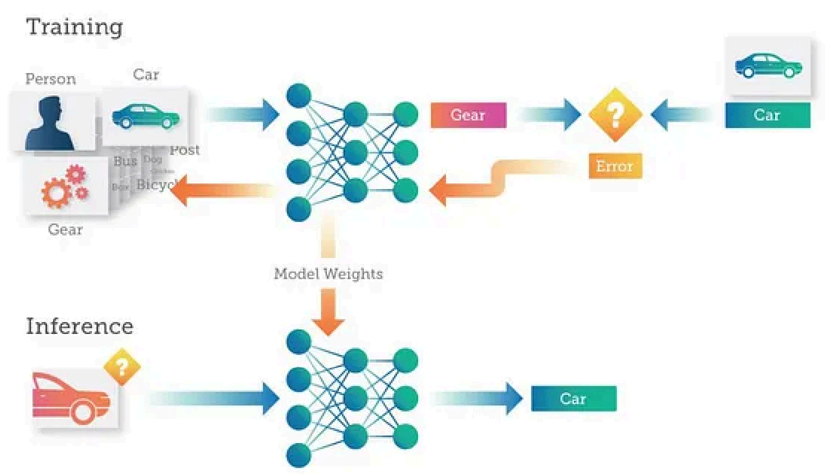
This implementation of mixed precision training with PyTorch's AMP library allows for efficient utilization of FP16 computations for improved training speed and reduced memory usage, while maintaining the necessary precision for accurate weight updates using FP32.

> *While mixed precision training can bring several benefits, there are cases where it may not be suitable or could potentially harm the training process.*

The potential harms of using mixed precision training include:

- Loss of Numerical Precision because of FP16 and this loss of precision may lead to reduced model accuracy, particularly in tasks that require high precision. Mixed precision might not be suitable when it comes to numerical precision-critical tasks.
- Increased Vulnerability to Underflow and Overflow because of numerical instability during training which could affect the convergence and performance of the model.
- Increased Complexity because of mixed precision which requires additional considerations, such as managing precision transitions, scaling gradients, and handling possible issues related to precision mismatch.
- **If your model suffers from severe gradient explosion or vanishing problems, switching to lower-precision computations (FP16) in mixed precision training may worsen these issues.** In such cases, it's crucial to address the underlying instability problems before considering mixed precision training.

Training and Validation Mode



An image reference for training vs validation (inference) mode. Image source : teledyneimaging.com

When fine-tuning a model, it is initially in training mode by default after loading a pre-trained model. However, we can switch the model to validation mode during inference or validation. These mode changes alter the model's behaviour accordingly.

Training mode

When the model is in training mode, it enables specific operations that are required during the training process, such as computing gradients, updating parameters, and applying regularization techniques like dropout. In this mode, the model behaves as if it is being trained on a training dataset and is ready to learn from the data.

```
model.train() # sets model in training mode
```

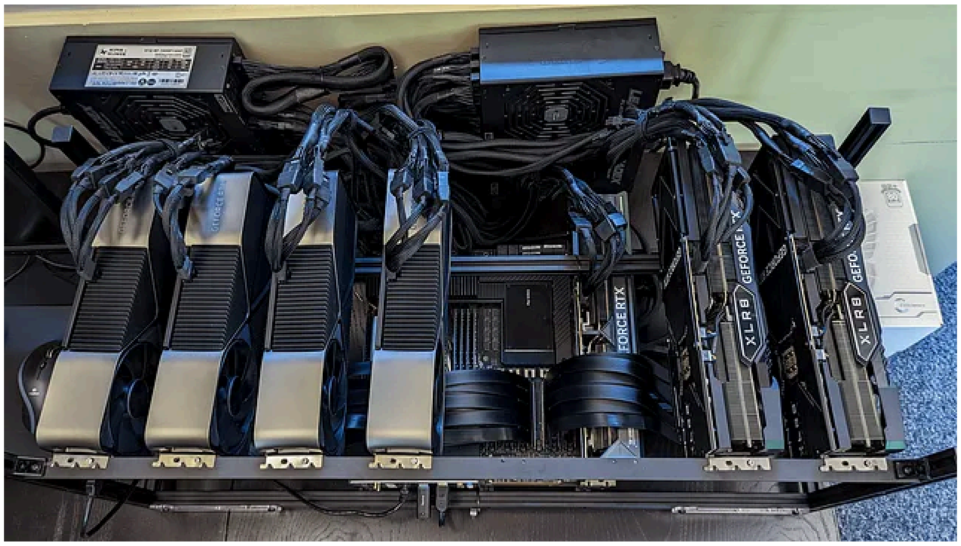
Validation mode

When the model is in evaluation mode, it disables certain operations that are only necessary during training, such as computing gradients, dropout, and updating parameters. This mode is typically used during validation or testing when you want to assess the model's performance on unseen data.

```
model.val() # sets model in validation mode
```

Very crucial to set model to right mode during fine-tuning as setting the model to the right mode ensures consistent behaviour and proper operations for each phase (training or evaluation). This leads to accurate results, efficient resource usage, and prevents issues like overfitting or inconsistent normalization.

Single GPU and Multi GPU



A machine with multiple GPUs. Image source : Puget System

GPUs are essential for deep learning and fine-tuning tasks because they excel at performing highly parallel computations, which significantly speed up the training process. In cases where you have access to multiple GPUs, you can leverage their collective power to further accelerate training. Here's an example of how to utilize multiple GPUs if you have them available:

```
# Define your model
model = MyModel()
model = model.to(device)  # Move the model to the desired device (CPU or GPU)

# Check if multiple GPUs are available
if torch.cuda.device_count() > 1:
    print("Using", torch.cuda.device_count(), "GPUs for training.")
    model = nn.DataParallel(model)  # Wrap the model with DataParallel

# Define your loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

This code snippet first checks if multiple GPUs are available using `torch.cuda.device_count()`. If there are more than one GPU, the model is wrapped with `nn.DataParallel`, allowing it to utilise all available GPUs for training. Each GPU processes a portion of the data simultaneously, resulting in faster training times. If only a single GPU is present, the code runs on that GPU without utilizing `nn.DataParallel`.

Conclusion

In this first part of the Ultimate Guide to Fine-Tuning in PyTorch, we have explored the essential steps involved in fine-tuning a pre-trained model to suit our specific tasks. By leveraging transfer learning, we can save significant time and computational resources while achieving impressive results. Throughout this article, we have learned how to load a pre-trained model, adapt its head architecture to match the target task, and customize hyperparameters like learning rate, optimizer, and weight decay to optimize the fine-tuning process. Additionally, we looked into selecting appropriate loss functions and the benefits of freezing specific parts of the network for more controlled fine-tuning.

Furthermore, we addressed the significance of defining the model's floating-point precision and the importance of toggling between training and validation modes during fine-tuning. Moreover, we explored how to make the most out of both single GPU and multi-GPU setups to accelerate training and boost performance.

In Part 2 of this series, we will delve deeper into advanced techniques to enhance the accuracy and generalization capabilities of our fine-tuned models. We will explore approaches like data augmentation, learning rate schedules, gradient clipping, and ensembling to further improve the model's performance on diverse and challenging datasets.

So, stay tuned for the continuation of this journey towards mastering the art of fine-tuning in PyTorch!

Further reading

Ultimate Guide to Fine-Tuning in PyTorch : Part 2 — Techniques for Enhancing Model Accuracy

Uncover Proven Techniques for Boosting Fine-Tuned Model Accuracy. From Basics to Overlooked Strategies, Unlock Higher…

rumn.medium.com

Share your thoughts