Model Inference, Saving, and Cross-Framework Compatibility

Kaiser Hamid

December 16, 2024

Contents

1 Introduction

After training a deep learning model, the next important steps are:

- Running inference on new data.
- Saving the trained model so that it can be easily loaded and used by other modules.
- Ensuring that the saved model format is flexible enough to be used across different frameworks, if desired.

In this article, we will cover:

1. How to perform inference in TensorFlow (Keras).
2. How to perform inference in PyTorch.
3. How to use a common format (such as ONNX) for saving models, enabling inference across different frameworks.

2 Inference with TensorFlow (Keras)

Suppose you have trained a Keras model and saved it as `model.h5`. Now you want to load this model in a separate script and run inference on a new image.

2.1 Loading the Model

```
import tensorflow as tf
from tensorflow.keras.preprocessing import image
import numpy as np

# Load the saved model
model = tf.keras.models.load_model('model.h5')
```

Explanation: We use `load_model` to load the entire model (architecture, weights, and optimizer state) from the `.h5` file.

2.2 Preparing New Data for Inference

```
img_path = 'new_image.jpg'
img_height = 150
img_width = 150

img = image.load_img(img_path, target_size=(img_height,
   img_width))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)  # Create batch of size 1
x = x / 255.0  # Normalize if that was done during
   training
```

Explanation: We resize the image, convert it to a NumPy array, and normalize it according to the preprocessing used during training.

2.3 Performing Inference

```
predictions = model.predict(x)
print("Predictions:", predictions)
```

Explanation: `model.predict(x)` returns the model's output. For classification, this might be probabilities of each class.

2.4 Modular Usage

By saving your model (e.g., `model.save('model.h5')`), you can have:

- A training script that only produces `model.h5`.
- A separate inference script that loads `model.h5` and runs inference.

This separates training and inference into distinct modules. For more flexibility, consider using the TensorFlow SavedModel format:

```
model.save('saved_model')
```

This format can be loaded in multiple environments, including TensorFlow Serving for production.

3 Inference with PyTorch

Assume you trained a PyTorch model and saved its weights using:

```
torch.save(model.state_dict(), 'model.pth')
```

3.1 Loading the Model

```
import torch
from model_definition import SimpleCNN  # Assume the
    model class is defined here

device = torch.device('cuda' if torch.cuda.is_available
    () else 'cpu')

model = SimpleCNN()
model.load_state_dict(torch.load('model.pth',
    map_location=device))
model.to(device)
model.eval()
```

Explanation: The model class must be defined or imported so the state dictionary can be loaded. `model.eval()` puts the model in inference mode (disabling dropout, etc.).

3.2 Preparing Data for Inference

```
from PIL import Image
from torchvision import transforms

img_path = 'new_image.jpg'
img_height = 150
img_width = 150

val_transforms = transforms.Compose([
    transforms.Resize((img_height, img_width)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5],
                         [0.5, 0.5, 0.5])
])

img = Image.open(img_path).convert('RGB')
x = val_transforms(img).unsqueeze(0).to(device)
```

Explanation: We apply the same preprocessing steps used during training/validation. `unsqueeze(0)` creates a batch dimension.

3.3 Performing Inference

```
with torch.no_grad():
    outputs = model(x)
    preds = (outputs > 0.5).float()
    print("Predictions:", preds.item())
```

Explanation: We use `torch.no_grad()` since we do not need gradients for inference. The predictions are thresholded if it's a binary classification.

3.4 Modular Usage

By saving just the model's weights:

- The training script produces `model.pth`.
- The inference script imports the model definition and loads `model.pth`.

This modularity allows any script to load and run inference without the training code.

4 Common Saving Methods: Using ONNX for Cross-Framework Inference

Sometimes you may want to use a model trained in PyTorch within a TensorFlow environment, or vice versa. While there is no native format that both can load directly, the **ONNX** (Open Neural Network Exchange) format is designed to allow models to be transferred between different deep learning frameworks.

4.1 Exporting to ONNX

Suppose you have a PyTorch model that you want to convert to ONNX:

```
dummy_input = torch.randn(1, 3, 150, 150, device=device)
torch.onnx.export(
    model,                  # the model being exported
    dummy_input,            # sample input for tracing
    "model.onnx",           # output file name
    input_names=["input"],
    output_names=["output"],
    dynamic_axes={"input": {0: "batch_size"}, "output":
       {0: "batch_size"}}
)
```

Explanation: The `torch.onnx.export` function converts a PyTorch model into an ONNX file. We provide a dummy input of the correct shape. The ONNX model can then be loaded by other frameworks or tools that support ONNX.

4.2 Using ONNX in TensorFlow or Other Tools

- Tools like `onnx-tf` can convert ONNX models to TensorFlow graphs.
- ONNX Runtime can run ONNX models directly for inference, independent of TensorFlow or PyTorch.

This approach can facilitate a workflow where:

- Train in PyTorch (or TensorFlow).
- Export the model to ONNX.
- Load the ONNX model in TensorFlow, or run it using ONNX Runtime.

By using ONNX, you create a more framework-agnostic model that can be integrated into various inference pipelines, potentially broadening the deployment scenarios.

5 Best Practices for Modular and Cross-Framework Model Usage

- **Keep Model Definitions Separate:** Store the model architecture in a separate file (e.g., `model_definition.py`). This way, the training code and inference code can both import it without duplicating the definition.
- **Consistent Preprocessing:** Use the same image resizing, normalization, and transformations during inference as in training. Document these steps clearly so there is no confusion when loading the model in a different script or framework.
- **Use Standard Formats for Flexibility:** For TensorFlow, the SavedModel format is versatile. For PyTorch, `.pth` files are common. To enable cross-framework usage, consider exporting to ONNX format.

- **Version Control and Dependencies:** Keep track of the model's version and ensure that the inference environment has compatible library versions. If you convert to ONNX, verify that the runtime or conversion tools match the model's requirements.

6 Conclusion

In this article, we discussed:

- How to load and run inference with TensorFlow and Keras models saved in `.h5` or SavedModel format.
- How to load and run inference with PyTorch models saved as state dictionaries (`.pth` files).
- How to use the ONNX format as a common intermediate representation that can be loaded by different frameworks, enabling cross-framework inference.

By following these guidelines, you can train models once, save them, and use them across multiple scripts, environments, and even frameworks, all while keeping your pipeline flexible, maintainable, and deployment-ready.