Deploying PyTorch Models in Production: PyTorch Playbook

PERSISTING AND LOADING PYTORCH MODELS

Janani Ravi

CO-FOUNDER, LOONYCORN

www.loonycorn.com

Overview

Persist trained models and load trained models

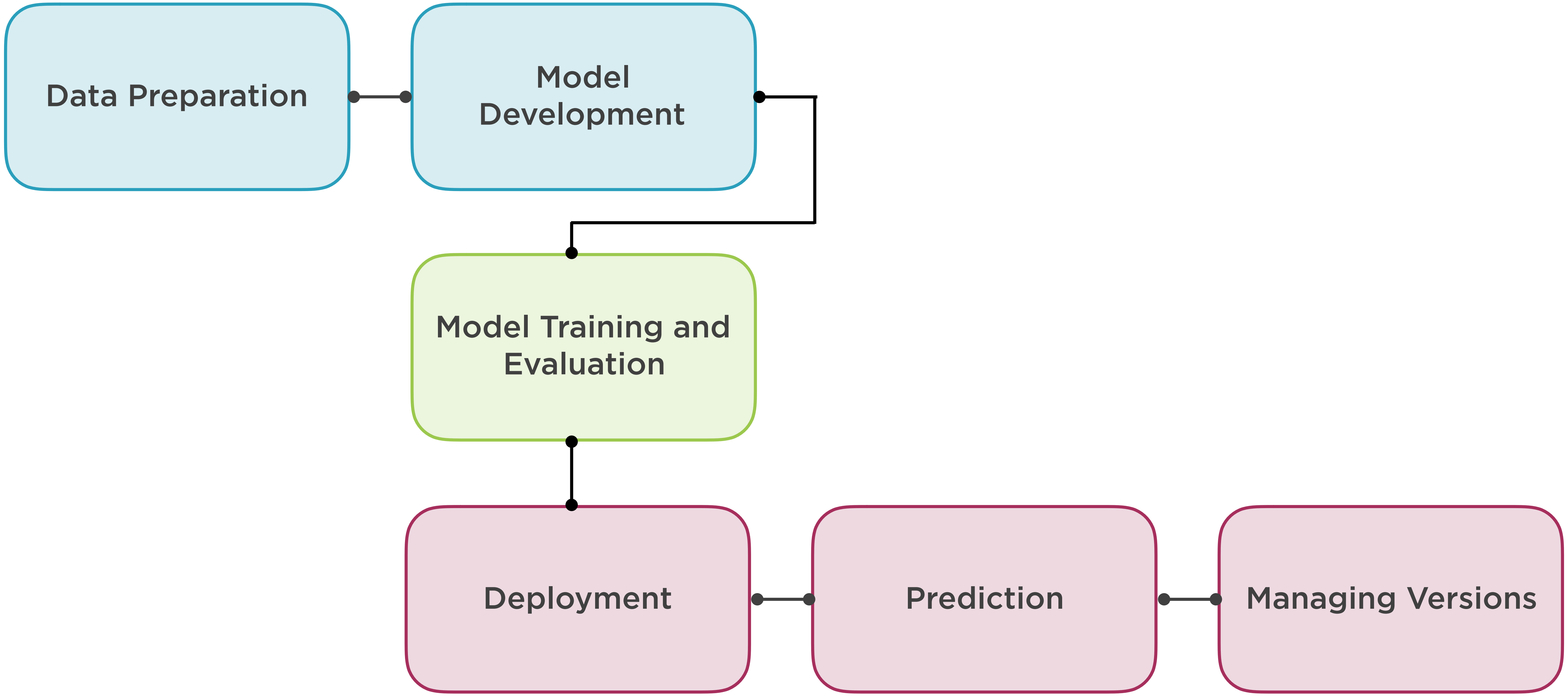
Correctly use torch.save() and torch.load()

Serialize models using pickle

Use a persisted state_dict to save learnable parameters

Use ONNX for model portability

Production ML Workflow



Data Preparation

Model Development

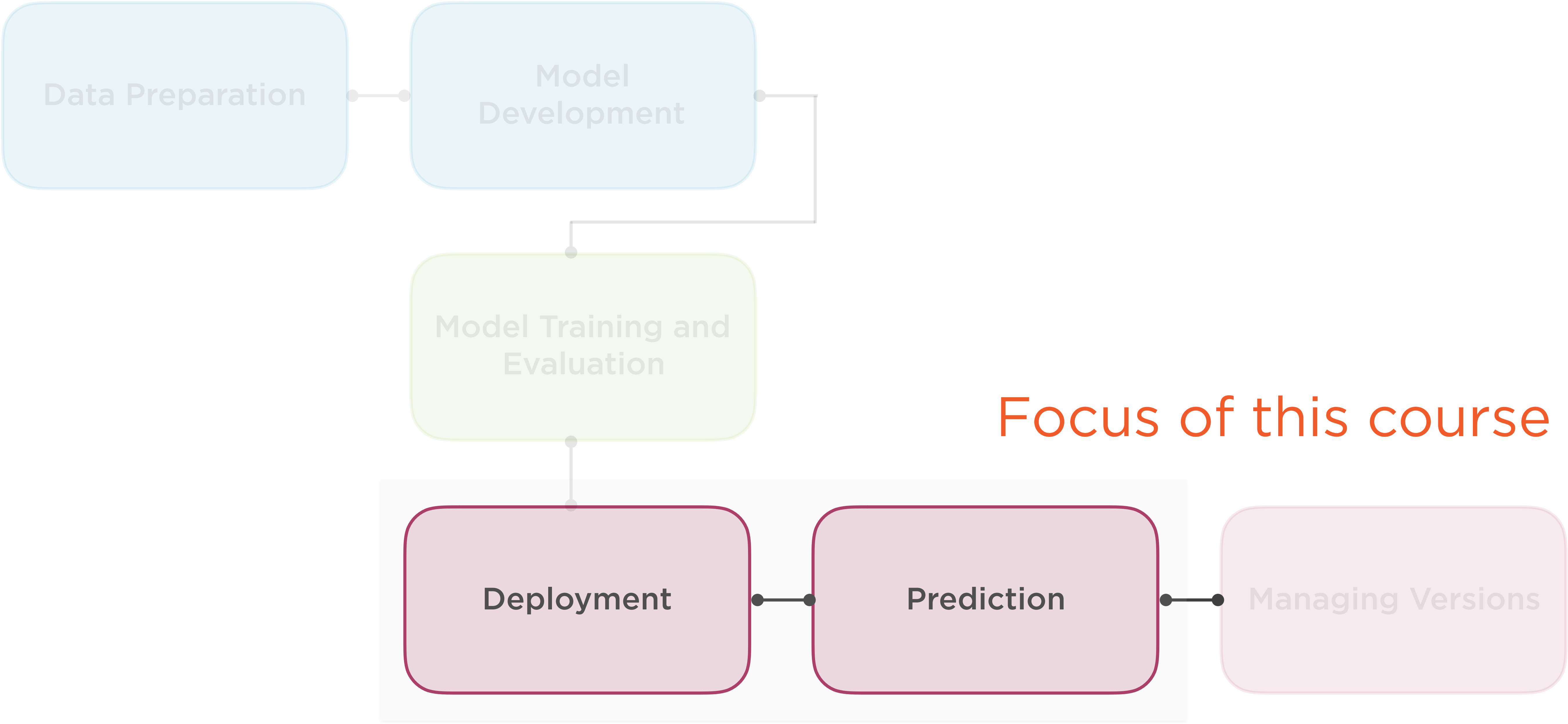
Model Training and Evaluation

Deployment

Prediction

Managing Versions

Production ML Workflow



Focus of this course

Prerequisites and Course Outline

Prerequisites

Basic Python programming

Basic knowledge of PyTorch

Basic knowledge of distributed computing

Prerequisite Courses

Foundations of PyTorch

Building Your First PyTorch Solution

Course Outline

Persisting and loading models

Training with single and multiple processors

Distributed training on multiple machines

Deploying models to production

Saving and Loading PyTorch Models

Saving and Loading Models in PyTorch

torch.save

torch.load

torch.nn.Module.
load_state_dict

Saving and Loading Models in PyTorch

torch.save

torch.load

torch.nn.Module.
load_state_dict

torch.save()

Save serialized object to disk

Uses Python pickle utility

Models, tensors, dictionaries

torch.load()

Extract deserialized object from disk

Uses Python pickle utility

Can specify device to load into into

torch.save() and torch.load()

Pros

- Simplest, most intuitive syntax
- Saves entire module using pickle

torch.save() and torch.load()

Cons

- Serialized data bound to specific classes
- Exact directory structure saved
- Model class not saved in isolation
- Introduces dependencies, fragility

The recommended approach is to save the **state_dict** for maximum flexibility during restoration

state_dict

A Python dictionary that maps each layer to a corresponding tensor of learnable parameters (weights and biases)

state_dict

A Python dictionary that maps each layer to a corresponding tensor of learnable parameters (weights and biases)

Saving and Loading Models in PyTorch

torch.save

torch.load

torch.nn.Module.
load_state_dict

torch.nn.Module.
load_state_dict

Load a model’s parameter dictionary

Uses deserialized state_dict

state_dict

Contains entries for

- Layers with learnable parameters
- Registered buffers

state_dict

Objects that possess a state_dict

- torch.nn.Module models
- torch.optim

state_dict

Just ordinary Python dictionaries

Also contain hyperparameter information

Can be easily saved, updated, altered and restored

Makes state of models and optimizers very modular

Checkpoints

Can be used to resume training for a model. During checkpointing, it is important to save state_dict for both the model as well as the optimizer objects.

Checkpoints

Can be used to resume training for a model. During checkpointing, it is important to save state_dict for both the model as well as the optimizer objects.

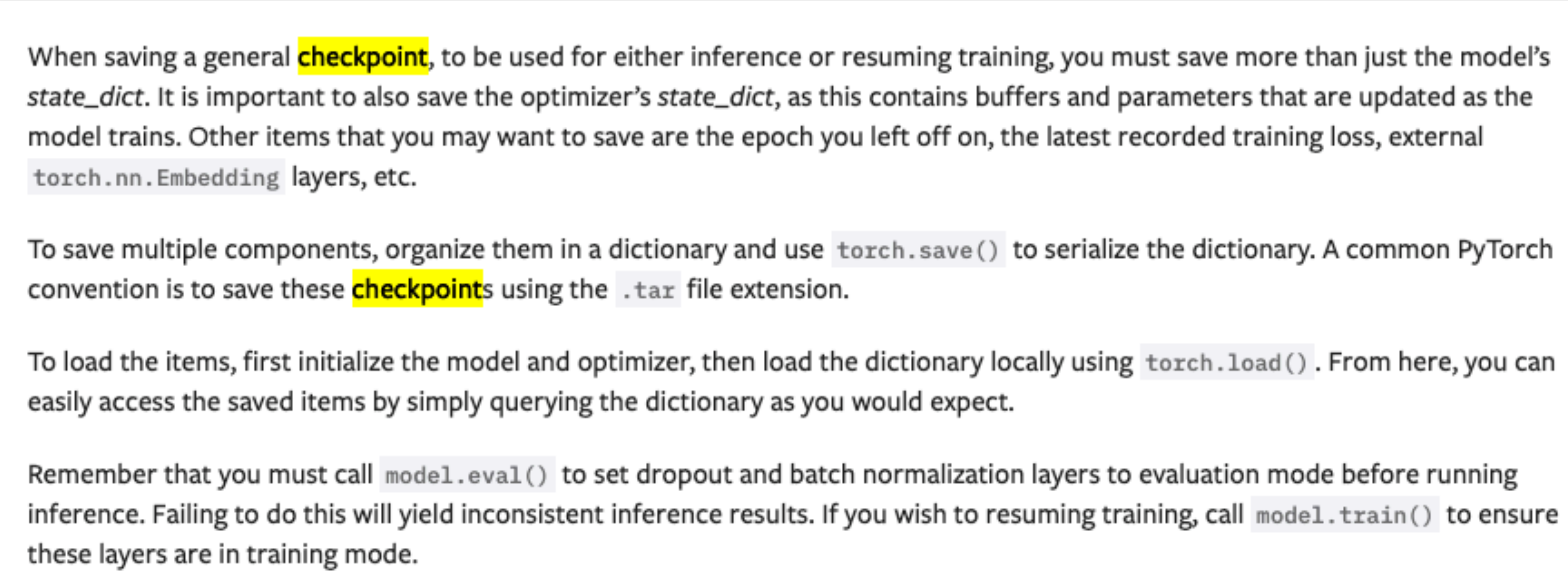
Persisted Model Parameters

Upon loading must remember to call model.eval()

To set dropout and batch normalization

Failing to do yields inconsistent inference results

To resume training, must call model.train()



When saving a general checkpoint, to be used for either inference or resuming training, you must save more than just the model's *state_dict*. It is important to also save the optimizer's *state_dict*, as this contains buffers and parameters that are updated as the model trains. Other items that you may want to save are the epoch you left off on, the latest recorded training loss, external `torch.nn.Embedding` layers, etc.

To save multiple components, organize them in a dictionary and use `torch.save()` to serialize the dictionary. A common PyTorch convention is to save these checkpoints using the `.tar` file extension.

To load the items, first initialize the model and optimizer, then load the dictionary locally using `torch.load()`. From here, you can easily access the saved items by simply querying the dictionary as you would expect.

Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference. Failing to do this will yield inconsistent inference results. If you wish to resuming training, call `model.train()` to ensure these layers are in training mode.

Demo

Using torch.save() and torch.load() to save and load models

Demo

Saving learnable parameters using the state_dict

Demo

Saving checkpoints to resume training

Introducing ONNX

ONNX

ONNX is an open format to represent deep learning models that allows models to be re-used across frameworks

ONNX

Community of partners

- Amazon AWS
- Facebook Open Source
- Microsoft
- NVIDIA

ONNX

ONNX models supported in

- Caffe2
- Microsoft Cognitive Toolkit (CNTK)
- Apache MXNet
- PyTorch

ONNX in Caffe2

Caffe2 supports native import and export of ONNX models

ONNX in PyTorch

PyTorch models can be exported to ONNX

PyTorch cannot import ONNX models

Demo

Exporting a PyTorch model to ONNX

Loading an ONNX model in Caffe2

Summary

Persist trained models and load trained models

Correctly use torch.save() and torch.load()

Serialize models using pickle

Use a persisted state_dict to save learnable parameters

Use ONNX for model portability