An Introduction to Practical 1

Lecture 3 for Advanced Deep Learning Systems

Aaron Zhao, Imperial College London, a.zhao@imperial.ac.uk

Table of contents

Introduction

Introduction

Two labs are in Practical 1

- Lab 1: Experiment training of a JSC network
- Lab 2: Write a quantization transformation pass

Deliverable

- A Markdown file: with all answers (plots, tables ...) of the questions and optional questions.
- Corresponding code in your forked repository.

Examination (15%)

- Submission requires the Markdown files only.
- Lab oral to check on your code and Q&A.

Lab 1: Training and evaluating a network using MASE

Using Google Colab

If you do not have a powerful enough work station with a GPU, it is likely you will have to use Google Colab for this course.

There is a quick introduction on how to set it up correctly and use it. The department will reimburse the cost of using Colab Pro.

Suggested and tested usage is through the command line interface in Colab: treat it as a serer not a notebook!

Accessing MASE

MASE has two modes to be accessed

- Direct usage through the command line interface, use it as a tool.
- Interactive usage, import it as a package

In the first lab, we are dealing with the command line interface.

Installing MASE

Avoid installing directly using your native Python

- Docker: OS-level virtualization, containers
- Conda: Environment controlling

Follow the Readme on the installation, pick the one you like.

Training a network

The train command

```
# You will need to run this command
./ch train jsc-tiny jsc --max-epochs 10 --batch-size 256
```

- ‘./ch’ to execute
- ‘train’ is the action
- ‘jsc-tiny’ is the target network
- ‘jsc’ is the target dataset
- ‘–parameter’ gives values to parameters through the command line interface

Test and evlaute a network

The test command

```
./ch test jsc-tiny jsc --load your_ckpt
```

You will need to test the trained network (load its trained weights),

Training a network

The train command

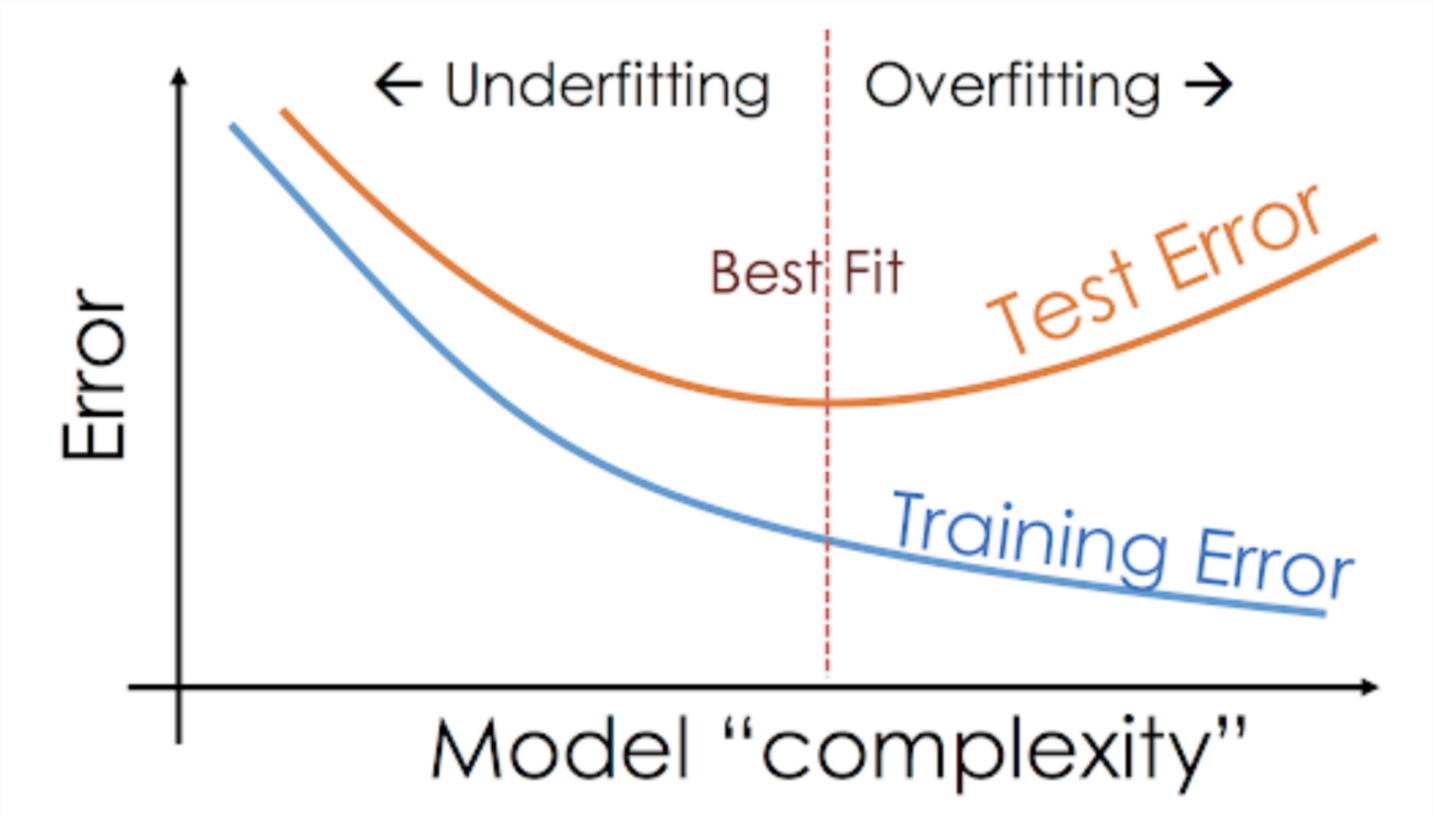
```
# You will need to run this command
./ch train jsc-tiny jsc --max-epochs 10 --batch-size 256
```

- ‘./ch’ to execute
- ‘train’ is the action
- ‘jsc-tiny’ is the target network
- ‘jsc’ is the target dataset
- ‘–parameter’ gives values to parameters through the command line interface

Some training parameters you might use

- ‘batch_size’ how many inputs to proceed concurrently
- ‘learning_rate’ the critical parameter for your optimizer
- ‘max_epochs’ maximum number of epochs to train
- ‘accelerator’ choose to use GPU or not

Underfitting and Overfitting



- Underfitting: the model is too simple to capture the data.
- Overfitting: the model is too complex and memorizes the data.

Causses of underfitting and overfitting

- Model sizes: too small or too large.
- Learning rate: too small or too large.

The underlying system

You have to have a rough idea of what is happening in the underlying hardware system.

- Data: is your data on the disk or on the device? If it is on the device, is it in CPU RAM or GPU VRAM?
- Data pre-processing: is this on CPU or GPU?
- Actual Training: is this on CPU or GPU? If it is on GPU, how much data do you need to ship from CPU to avoid idling GPUs?

MASE automatically handles some of these things, but you need to have an idea of what is going on.

Use 'nvidia-smi' and 'htop' to inspect the system while running your stuff!

'wathc -n 0.5 nvidia-smi' inspects the GPU usage and refreshes every 0.5 seconds.

Dataset definition

```
@add_dataset_info(
    ...
)
class JetSubstructureDataset(Dataset):
    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.Y[idx]

    def prepare_data(self) -> None:
        ...

    def setup(self) -> None:

        ...
```

Dataset definition

- If you haven't seen @ in python, go search and learn what is a Python decorator.
- '__len__' defines the size of the dataset
- '__getitem__' defines how elements in a dataset is accessed

You can almost tell that X and Ys are prepared when instantiating this class, from the '__getitem__' function you can tell elements are accessed through an internal list (CPU RAM or GPU VRAM).

Model definition

```
class JSC_Tiny(nn.Module):
    def __init__(self):
        super(JSC_Tiny, self).__init__()
        self.seq_blocks = nn.Sequential(
            # 1st LogicNets Layer
            nn.BatchNorm1d(16),  #  batch norm layer
            nn.Linear(16, 5),  # linear layer
        )

    def forward(self, x):
        return self.seq_blocks(x)
```

Pytorch 101

This is standard Pytorch semantics

- We define our neural network by subclassing 'nn.Module'.
- We initialize the neural network layers in '__init__'.
- Every nn.Module subclass implements the operations on input data in the forward method.

Pytorch 101

```
model = JSC_Tiny()
print(model)
```

```
JSC_Tiny(
    (seq_blocks): Sequential(
        (0): BatchNorm1d(16, eps=1e-05, momentum=0.1,
        ↪  affine=True, track_running_stats=True)
        (1): Linear(in_features=16, out_features=5,
        ↪  bias=True)
    )
)
```

Pytorch 101

- nn.Sequential: an ordered container of modules. The data is passed through all the modules in the same order as defined. You can use sequential containers to put together a quick network.
- nn.Linear and nn.BatchNorm1d: these are the layers of the network.

Pytorch 101: Modules and Funcitons

- nn.Modules
  - PyTorch uses modules to represent neural networks.
  - Modules are: Building blocks of stateful computation.
  - PyTorch provides a robust library of modules.
  - Tightly integrated with PyTorch's autograd system, directly interacts with Optimizers.
- nn.functional
  - Implementation of many basic autograd functions.
  - Lower-level than nn.Modules, normally directly connects to backend (CUDA/CPU) functions.
  - One can imagine nn.Module as a wrapper of nn.functional.

Pytorch 101: Autograd Funcitons

- Autograd functions normally take the following form.
- Customized forward and backward functions.
- ctx is a context memory for buffering intermediate values.
- Some commonly used functions such as torch.functional.conv2d are directly implemented in C++/CUDA.

```
class LegendrePolynomial3(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

Pytorch 101: Module

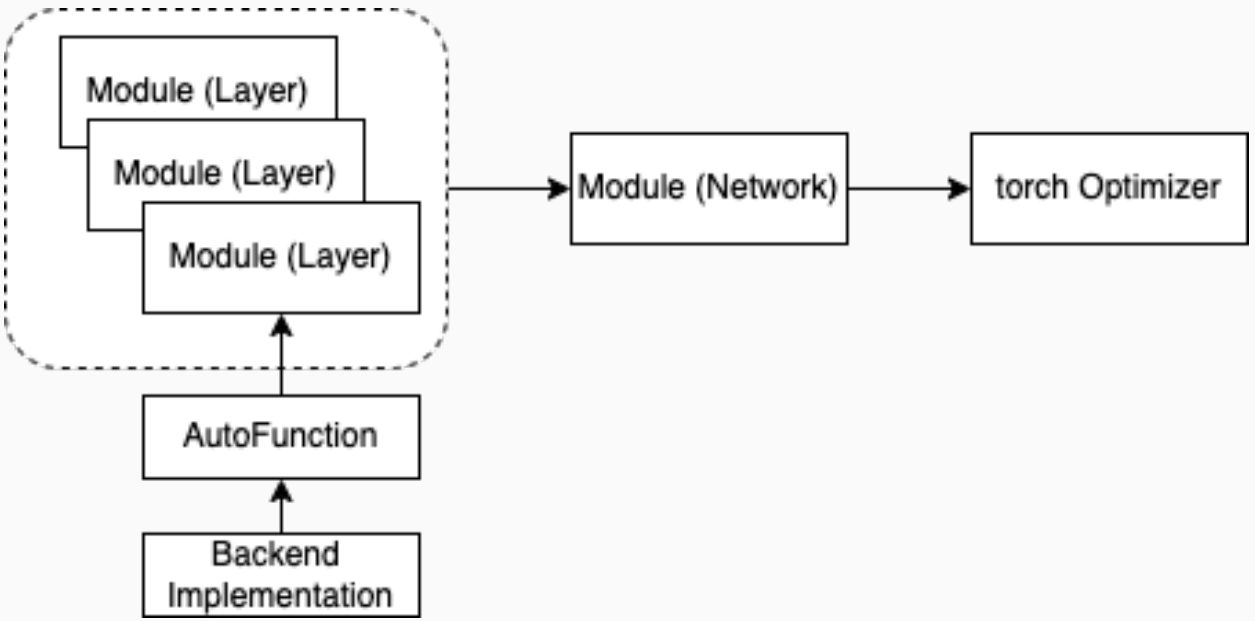
- Wraps autograd functions, drop them directly in forward.

```
1  class Conv2d(nn.Module):
2      ...
3  def _conv_forward(self, input: Tensor, weight: Tensor,
   ↪  bias: Optional[Tensor]):
4      ...
5      return F.conv2d(input, weight, bias, self.stride,
6                      self.padding, self.dilation,
                       ↪  self.groups)
7
8  def forward(self, input: Tensor) -> Tensor:
9      return self._conv_forward(input, self.weight,
       ↪  self.bias)
10
11
```

Pytorch 101: High-level overview

- Optimizers are only accessing nn.Module. Trainable parameters that are not wrapped in modules may not be tracked!

```
optimizer = torch.optim.SGD(model.parameters(),
↪  lr=learning_rate)
```



Lab 2: Applying a MASE transformation

Interactive usage

In the interactive usage mode, you are including various modules in MASE.

You will need to figure out the correct path for doing that.

```
# figure out the correct path
machop_path = Path(".").resolve().parent.parent /"machop"
assert machop_path.exists(), "Failed to find machop at:
↪  {}".format(machop_path)
sys.path.append(str(machop_path))
```

You can print out 'sys.path' to see whether the path is correctly added.

Interactive usage: instantiate the same dataset

```
batch_size = 8
model_name = "jsc-tiny"
dataset_name = "jsc"

data_module = MaseDataModule(
    name=dataset_name,
    batch_size=batch_size,
    model_name=model_name,
    num_workers=0,
)
data_module.prepare_data()
data_module.setup()
```

- Datasets that are supported in MASE cane be accessed through MaseDataModule

Interactive usage: instantiate the same model

```
model_info = get_model_info(model_name)
model = get_model(
    model_name,
    task="cls",
    dataset_info=data_module.dataset_info,
    pretrained=False)
```

- Datasets that are supported in MASE cane be accessed through MaseDataModule

Note: You can also use a Pytorch native model in MASE, but you might have to wrap it with the decrorater.

Generate a MaseGraph and apply passes

```
mg = MaseGraph(model=model)

# analysis pass
mg = init_metadata_analysis_pass(mg, None)
mg = add_common_metadata_analysis_pass(mg, dummy_in)
mg = add_software_metadata_analysis_pass(mg, None)

# transofrm
mg = quantize_transform_pass(mg, pass_args)
```

- As explained before, a MaseGraph is added with various metadata before being applied with a transformation pass.

Polymorphism of Passes

A pass, no matters whether it is an analysis pass or a transform pass, takes the following format

```
# pass_args is a dict
def pass(mg, pass_args):
    ...
# info a a dict
return mg, info
```