An Introduction to Labs

Aaron Zhao, Imperial College London

Overview

- Lab 0:
  - Introducing MASE and FX Graph
  - Insert a LoRA adaptor
- Lab 1:
  - Quantization-aware training
  - Unstructured pruning
- Lab 2:
  - NAS with Bayesian optimization

Introduction to the Mase IR, MaseGraph and Torch FX passes

Tools

Use 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.

Accessing MASE

MASE has two modes to be accessed

- Direct usage through the command line interface, use it as a tool (Not suggested...)
- Interactive usage, import it as a package

In all the labs, we are dealing with the second usage.

Installing MASE

Avoid installing directly using your native Python

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

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

Loading a pre-trained BERT model

In today's ML world, we typically use pre-trained models and fine-tune them on our data.

This lab aims to show you how to do that.

First, we load a model from the Hugging Face model hub, which is argubaly the most popular model hub.

```python
from transformers import AutoModelForSequenceClassification

model =
AutoModelForSequenceClassification.from_pretrained("prajjwal1/
bert-tiny")
```

FX Graph

There are 6 different types of nodes in an FX graph

- `placeholder`
- `get_attr`
- `call_function`
- `call_module`
- `call_method`
- `output`

A generated FX graph can be hard to visualize because of its size and complexity.

MASE IR and Passes

- FX Graph is a symbolic representation of a PyTorch model.
- FX IR has no information regarding the workload being executed by the graph - that's where the Mase IR comes in
- The major benefit of the Mase IR is in offering a common abstraction layer for both hardware and software workloads
- So that we can represent every optimization/analysis as a pass!

```python
def dummy_pass(mg, pass_args={}):

    # ... do some setup
    pass_outputs = {}

    for node in mg.fx_graph.nodes:
        # ... do stuff

    return mg, pass_outputs
```

An example of a Pass

```
from chop.tools import get_logger

logger = get_logger("mase_logger")
logger.setLevel("INFO")

def count_dropout_analysis_pass(mg, pass_args={}):
    dropout_modules = 0
    dropout_functions = 0
    for node in mg.fx_graph.nodes:
        if node.op == "call_module" and "dropout" in node.target:
            logger.info(f"Found dropout module: {node.target}")
            dropout_modules += 1
        else:
            logger.debug(f"Skipping node: {node.target}")
    return mg, {"dropout_count": dropout_modules
dropout_functions}
```

Polymorphism of Passes

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

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

This then makes chaining passes together very easy.

```python
mg = MaseGraph(model=model)
mg, _ = pass_a(mg, pass_args)
mg, _ = pass_b(mg, pass_args)
mg, _ = pass_c(mg, pass_args)
...
```

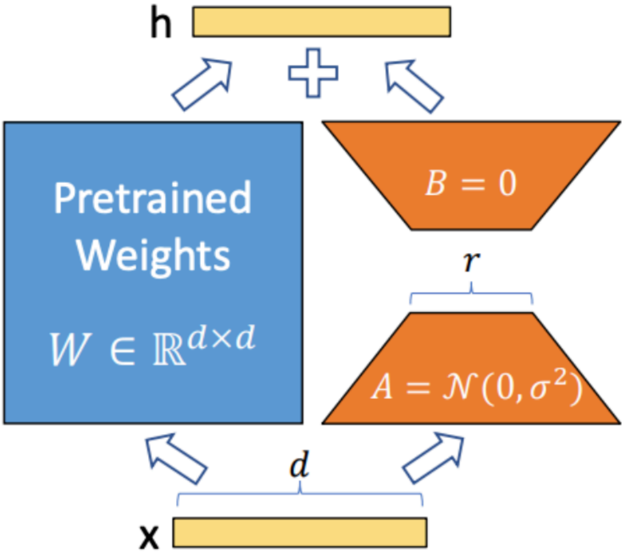
Finetuning Bert for Sequence Classification using a LoRA adapter

LoRA

- Full model fine-tuning can be expensive and time-consuming.
  - GPUs have limited memory.
  - More parameters mean more computation.
  - Can we fine-tune only a part of the model?
  - We have LoRA for that!
- LoRA is a lightweight adapter for fine-tuning pre-trained models.

  $y = xW = X(W + AB)$ where A and B are low-rank matrices

LoRA (ii)


h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

LoRA (iii)

The insertion of the LoRA adapter is done through a pass.

```
mg, _ = passes.insert_lora_adapter_transform_pass(
    mg,
    pass_args={
        "rank": 6,
        "alpha": 1.0,
        "dropout": 0.5,
    },
)
```

Some training parameters you might use

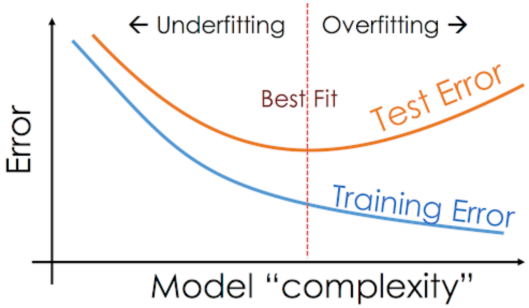
Canonical training parameters

- `batch_size`: how many inputs to proceed concurrently
- `learning_rate`: the critical parameter for your optimizer
- `max_epochs`: maximum number of epochs to train
- `dropout`: the dropout rate

LoRA parameters

- `rank`: the rank of the low-rank matrices
- `alpha`: the weight of the adapter

Underfitting and Overfitting



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

Causes of Underfitting and Overfitting

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

The underlying system

You need 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? What is the interconnect bandwidth? And is it a bottleneck?

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

```
class MyDataset(Dataset):
  def __len__(self):
    return len(self.X)

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

Dataset definition

- `__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 MyModel(nn.Module):
  def __init__(self):
    super(MyModel, self).__init__()
    self.blocks = nn.Sequential(...)

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

is standard Pytorch semantics

- We define a neural network by subclassing `nn.Module`. If you do not understand semantics such as `super`, `__init__`, `forward`, you should look them up and study OOP in your own time.
- We initialize the neural network layers in `__init__`.
- Every `nn.Module` subclass implements the operations on input data in the forward method.

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.
- 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.

Pytorch 101 (ii)

- 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.

```python
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.

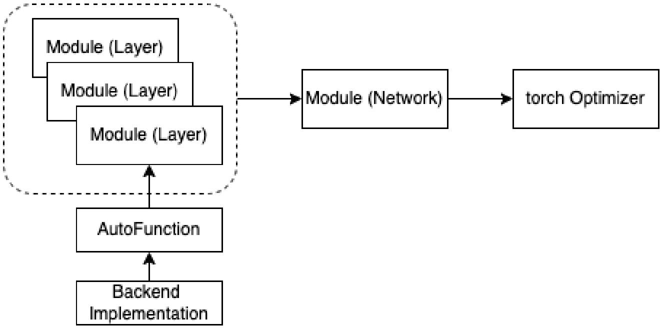
```python
class Conv2d(nn.Module): ...
  def _conv_forward(self, input, weight, bias): ...
    return F.conv2d(
      input, weight, bias,
      self.stride, self.padding,
      self.dilation, self.groups)

  def forward(self, input: Tensor) -> Tensor:
    return self._conv_forward(input, self.weight, self.bias)
```

Pytorch 101: High-level overview

- Optimizers (the optimizer we use for training) 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)
```



Running Quantization-Aware Training (QAT) on Bert

Two quantization paradigms

- Quantization-Aware Training (QAT): the model is trained with quantization in mind. We apply quantization to the model during training.
- Post-training quantization: the model is trained normally. We apply quantization to the model after training.

Which one do you think is more popular?

Applying a Quantization Pass

```
import chop.passes as passes

quantization_config = {
    "by": "type",
    ...
    "linear": {
        "config": {
            "name": "integer",
            "data_in_width": 8,
            "data_in_frac_width": 4,
            "weight_width": 8,
            "weight_frac_width": 4,
            "bias_width": 8,
            "bias_frac_width": 4,}}}

mg, _ = passes.quantize_transform_pass(
    mg, pass_args=quantization_config)
```

Unstructured Pruning on Bert

Various Pruning Techniques

Based on Granularity

- Unstructured pruning: removing individual weights from the model.
- Structured pruning: removing entire neurons, layers, or channels from the model.

Do you think unstructured pruning is more popular than structured pruning?

Based on Strategy

- Progressive pruning: gradually pruning the model over time.
- Iterative pruning: pruning the model in multiple iterations.
- Prune and retrain: pruning the model and retraining it.

and so on

Applying a Pruning Pass

```python
import chop.passes as passes

pruning_config = {
    "weight": {
        "sparsity": 0.5,
        "method": "l1-norm",
        "scope": "local",
    },
    "activation": {
        "sparsity": 0.5,
        "method": "l1-norm",
        "scope": "local",
    },
}

mg, _ = passes.prune_transform_pass(mg, pass_args=pruning_config)
```

Neural Architecture Search (NAS) with Mase and Optuna

Network Architecture Search

NAS is a technique to automatically search for the best neural networkNAS is a technique to automatically search for the best neural network architecture. In this lab, we will use Optuna to perform NAS on a simple BERT model

- Define the search space
- Write the model constructor
- Write the objective function

Defining the Search Space

The search space is defined as a dictionary, where the keys are the hyperparameters and the values are the search spaces.

```python
import torch.nn as nn
from chop.nn.modules import Identity

search_space = {
    "num_layers": [2, 4, 8],
    "num_heads": [2, 4, 8, 16],
    "hidden_size": [128, 192, 256, 384, 512],
    "intermediate_size": [512, 768, 1024, 1536, 2048],
    "linear_layer_choices": [
        nn.Linear,
        Identity,
    ],
}
```

Model constructor and objective function

Write a model constructor

The model constructor takes an "instance" or a sample of a set of hyperparameters and constructs a model.

Objective function

In this lab, the objective function is simply the evaluation accuracy.

In the real world, we normally use a more complex objective function, involving parameters such as the FLOPs, the number of parameters, or the inference time.

Bayesian Optimization

Optuna uses Bayesian optimization to search for the best hyperparameters under the hood.

Bayesian optimization (BO) is an optimization technique that uses probabilistic models to predict the performance of a model at a given hyperparameter configuration. It can then use this model to guide the search for the best hyperparameters.