KD$_{Lib}$

May 18, 2022

Contents:

CHAPTER 1

KD-Lib

A PyTorch model compression library containing easy-to-use methods for knowledge distillation, pruning, and quantization.

1.1 Installation

Building from source (recommended)

If you intend to install the latest unreleased version of the library (i.e from source), you can simply do:

```
$ git clone https://github.com/SforAiDl/KD_Lib.git
$ cd KD_Lib
$ python setup.py install
```

Stable release

KD_Lib is compatible with Python 3.6 or later and also depends on PyTorch. The easiest way to install KD_Lib is with pip, Python's preferred package installer.

```
$ pip install KD-Lib
```

Note that KD_Lib is an active project and routinely publishes new releases. In order to upgrade KD_Lib to the latest version, use pip as follows.

```
$ pip install -U KD-Lib
```

1.2 Usage

To implement the most basic version of knowledge distillation from Distilling the Knowledge in a Neural Network and plot losses

```
import torch
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import VanillaKD

# This part is where you define your datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

teacher_model = <your model>
student_model = <your model>

teacher_optimizer = optim.SGD(teacher_model.parameters(), 0.01)
student_optimizer = optim.SGD(student_model.parameters(), 0.01)

# Now, this is where KD_Lib comes into the picture

distiller = VanillaKD(teacher_model, student_model, train_loader, test_loader,
                      teacher_optimizer, student_optimizer)
distiller.train_teacher(epochs=5, plot_losses=True, save_model=True)    # Train the␣
↪teacher network
distiller.train_student(epochs=5, plot_losses=True, save_model=True)    # Train the␣
↪student network
distiller.evaluate(teacher=False)                                       # Evaluate␣
↪the student network
distiller.get_parameters()                                              # A utility␣
↪function to get the number of parameters in the teacher and the student network
```

To train a collection of 3 models in an online fashion using the framework in Deep Mutual Learning and log training details to Tensorboard

```
import torch
import torch.optim as optim
from torchvision import datasets, transforms
```

(continues on next page)

(continued from previous page)

```
from KD_Lib.KD import DML
from KD_Lib.models import ResNet18, ResNet50                                       # To␣
↪use models packaged in KD_Lib

# This part is where you define your datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

student_params = [4, 4, 4, 4, 4]
student_model_1 = ResNet50(student_params, 1, 10)
student_model_2 = ResNet18(student_params, 1, 10)

student_cohort = [student_model_1, student_model_2]

student_optimizer_1 = optim.SGD(student_model_1.parameters(), 0.01)
student_optimizer_2 = optim.SGD(student_model_2.parameters(), 0.01)

student_optimizers = [student_optimizer_1, student_optimizer_2]

# Now, this is where KD_Lib comes into the picture

distiller = DML(student_cohort, train_loader, test_loader, student_optimizers,␣
↪log=True, logdir="./Logs")

distiller.train_students(epochs=5)
distiller.evaluate()
distiller.get_parameters()
```

1.3 Implemented works

Some benchmark results can be found in the logs file.

Paper	Link	Repository (KD_Lib/)
Distilling the Knowledge in a Neural Network	https://arxiv.org/abs/1503.02531	KD/vision/vanilla
Improved Knowledge Distillation via Teacher Assistant	https://arxiv.org/abs/1902.03393	KD/vision/TAKD
Relational Knowledge Distillation	https://arxiv.org/abs/1904.05068	KD/vision/RKD
Distilling Knowledge from Noisy Teachers	https://arxiv.org/abs/1610.09650	KD/vision/noisy
Paying More Attention To The Attention	https://arxiv.org/abs/1612.03928	KD/vision/attention
Revisit Knowledge Distillation: a Teacher-free Framework	https://arxiv.org/abs/1909.11723	KD/vision/teacher_free
Mean Teachers are Better Role Models	https://arxiv.org/abs/1703.01780	KD/vision/mean_teacher
Knowledge Distillation via Route Constrained Optimization	https://arxiv.org/abs/1904.09149	KD/vision/RCO
Born Again Neural Networks	https://arxiv.org/abs/1805.04770	KD/vision/BANN
Preparing Lessons: Improve Knowledge Distillation with Better Supervision	https://arxiv.org/abs/1911.07471	KD/vision/KA
Improving Generalization Robustness with Noisy Collaboration in Knowledge Distillation	https://arxiv.org/abs/1910.05057	KD/vision/noisy
Distilling Task-Specific Knowledge from BERT into Simple Neural Networks	https://arxiv.org/abs/1903.12136	KD/text/BERT2LSTM
Deep Mutual Learning	https://arxiv.org/abs/1706.00384	KD/vision/DML
The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks	https://arxiv.org/abs/1803.03635	Pruning/ lottery_tickets
Regularizing Class-wise Predictions via Self- knowledge Distillation.	https://arxiv.org/abs/2003.13964	KD/vision/CSDK

Please cite our pre-print if you find KD_Lib useful in any way :)

```
@misc{shah2020kdlib,
  title={KD-Lib: A PyTorch library for Knowledge Distillation, Pruning and
↪Quantization},
  author={Het Shah and Avishree Khare and Neelay Shah and Khizir Siddiqui},
  year={2020},
  eprint={2011.14691},
  archivePrefix={arXiv},
  primaryClass={cs.LG}
}
```

CHAPTER 2

Installation

2.1 From source (recommended)

If you intend to install the latest unreleased version of the library (i.e from source), you can simply do:

```
$ git clone https://github.com/SforAiDl/KD_Lib.git
$ cd KD_lib
$ python setup.py install
```

2.2 Stable release

KD_Lib is compatible with Python 3.6 or later and also depends on PyTorch. KD-Lib can be installed from PyPI via pip,

```
$ pip install KD-Lib
```

Note that KD_Lib is an active project and routinely publishes new releases. In order to upgrade KD_Lib to the latest version, use pip as follows.

```
$ pip install -U KD-Lib
```

CHAPTER 3

Tutorials

3.1 VanillaKD using KD_Lib

To implement the most basic version of knowledge distillation from Distilling the Knowledge in a Neural Network and plot losses

```
import torch
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import VanillaKD

# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
```

(continues on next page)

(continued from previous page)

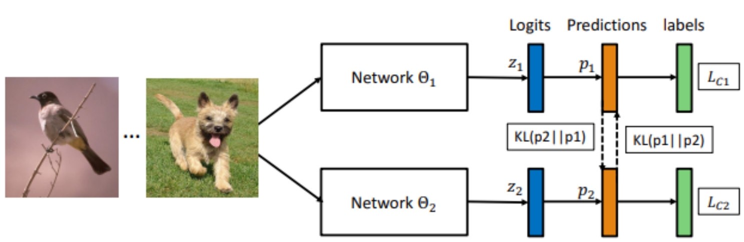
```
    batch_size=32,
    shuffle=True,
)

teacher_model = <your model>
student_model = <your model>

teacher_optimizer = optim.SGD(teacher_model.parameters(), 0.01)
student_optimizer = optim.SGD(student_model.parameters(), 0.01)

# Now, this is where KD_Lib comes into the picture

distiller = VanillaKD(teacher_model, student_model, train_loader, test_loader,
                      teacher_optimizer, student_optimizer)
distiller.train_teacher(epochs=5, plot_losses=True, save_model=True)    # Train the␣
↪teacher network
distiller.train_student(epochs=5, plot_losses=True, save_model=True)    # Train the␣
↪student network
distiller.evaluate(teacher=False)                                       # Evaluate␣
↪the student network
distiller.get_parameters()                                              # A utility␣
↪function to get the number of parameters in the teacher and the student network
```

3.2 Deep Mutual Learning using KD_Lib

Paper

- Deep Mutual Learning is an *online* algortihm wherein an ensemble of students learn collaboratively and teach each other throughout the training process.
- Rather performing a one way transfer from a powerful and large and pre-trained teacher network, DML uses a pool of untrained students who learn simultaneously to solve the task together.
- Each student is trained with two losses: a conventional supervised learning loss, and a mimicry loss that aligns each student's class posterior with the class probabilities of other students.

Snippet from the paper illustrating the DML algorithm -



To use DML with KD_Lib, create a list of student models (student cohort) to be used for collective training and a list of optmizers for them as well. The student models may have different architectures. Remember to match the order of the students with that of their optimizers in the list.

To use DML with 3 students on MNIST -

```
import torch
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import DML

# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

# Set device to be trained on

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Define a cohort of student models

student_model_1 = <your model>
student_model_2 = <your model>
student_model_3 = <your model>

student_cohort = (student_model_1, student_model_2, student_model_3)

# Make a list of optimizers for the models keeping in mind the order

student_optimizer_1 = optim.SGD(student_model_1.parameters(), 0.01)
student_optimizer_2 = optim.SGD(student_model_2.parameters(), 0.01)
student_optimizer_3 = optim.SGD(student_model_3.parameters(), 0.01)

optimizers = [student_optimizer_1, student_optimizer_2, student_optimizer_3]

# Train using KD_Lib

distiller = DML(student_cohort, train_loader, test_loader, optimizers,
                device=device)
distiller.train_students(epochs=5, plot_losses=True, save_model=True)    # Train the student cohort
```

(continues on next page)

(continued from previous page)

```
distiller.evaluate()                                                    # Evaluate the student models
```

3.3 Label Smooth Regularization using KD_Lib

Paper

- Considering a sample x of class k with ground truth label distribution l = δ(k), where $\delta(\cdot)$ is impulse signal, the LSR label is given as -

$$l' = (1 - \epsilon)\delta(k) + \epsilon/K,$$

where K is the number of classes

To use the label smooth regularization with incorrect teacher predictions replaced with labels where the correct classes have a probability of 0.9 -

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import LabelSmoothReg

# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)
```

(continues on next page)

(continued from previous page)

```
# Set device to be trained on

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Define student and teacher models

teacher_model = <your model>
student_model = <your model>

# Define optimizers

teacher_optimizer = optim.SGD(teacher_model.parameters(), lr=0.01)
student_optimizer = optim.SGD(student_model.parameters(), lr=0.01)

# Train using KD_Lib

distiller = LabelSmoothReg(teacher_model, student_model, train_loader, test_loader,␣
↪teacher_optimizer,
                           student_optimizer, correct_prob=0.9, device=device)
distiller.train_teacher(epochs=5)                                       # Train the␣
↪teacher model
distiller.train_students(epochs=5)                                      # Train the␣
↪student model
distiller.evaluate(teacher=True)                                        # Evaluate␣
↪the teacher model
distiller.evaluate()                                                    # Evaluate␣
↪the student model
```

3.4 Probability Shift using KD_Lib

Paper

- Given an incorrect soft target, the probability shift algorithm simply swaps the value of ground truth (the theoretical maximum) and the value of predicted class (the predicted maximum), to assure the maximum confidence is reached at ground truth label

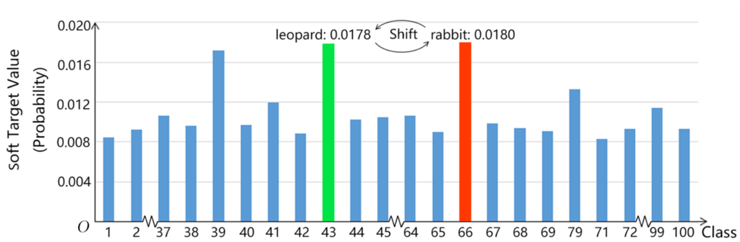


Fig. 2. PS on a misjudged sample's soft target. The image is from CIFAR-100 training data, whose ground truth label is "leopard" but ResNet-50 teacher's prediction is "rabbit". The value of "leopard" is still large, which indicates that the teacher does not go ridiculous. Shift operation is carried out towards these two classes.

To use the probability shift algorithm to train a student on MNIST for 5 epcohs -

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import ProbShift
```

(continues on next page)

(continued from previous page)

```
# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

# Set device to be trained on

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Define student and teacher models

teacher_model = <your model>
student_model = <your model>

# Define optimizers

teacher_optimizer = optim.SGD(teacher_model.parameters(), lr=0.01)
student_optimizer = optim.SGD(student_model.parameters(), lr=0.01)

# Train using KD_Lib

distiller = ProbShift(teacher_model, student_model, train_loader, test_loader,␣
↪teacher_optimizer,
                      student_optimizer, device=device)
distiller.train_teacher(epochs=5)                                    # Train the␣
↪teacher model
distiller.train_students(epochs=5)                                   # Train the␣
↪student model
distiller.evaluate(teacher=True)                                     # Evaluate␣
↪the teacher model
distiller.evaluate()                                                 # Evaluate␣
↪the student model
```

3.5 Route Constrained Optimization using KD_Lib

Paper

- The route constrained optimization algorithm considers knowledge distillation from the perspective of curriculum learning by routing
- Instead of supervising the student model with a converged teacher model, it is supervised with some anchor points selected from the route in parameter space that the teacher model passed by
- This has been demonstrated to greatly reduce the lower bound of congruence loss for knowledge distillation, hint and mimicking learning

Algorithm 1 Route Constrained Optimization

Require: anchor points set from pre-trained teacher network: $C_1, C_2, ..., C_n$, student network with parameter W_i
  $i = 1$
  Randomly initialize W_i
  while $i \leq n$ **do**
    Initialize teacher network with C_i anchor, get W_{C_i}
    if $i > 1$ **then**
      Initialize W_i with W_{i-1}
    end if
    update the W_i by optimizing $L_{KD}(W_i, W_{C_i})$
    $i = i + 1$
  end while
  get W_n as the final weights of student.

To use RCO with the the student mimicking the teacher's trajectory at an interval of 5 epochs -

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import RCO

# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)
```

(continues on next page)

(continued from previous page)

```
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

# Set device to be trained on

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Define student and teacher models

teacher_model = <your model>
student_model = <your model>

# Define optimizers

teacher_optimizer = optim.SGD(teacher_model.parameters(), lr=0.01)
student_optimizer = optim.SGD(student_model.parameters(), lr=0.01)

# Train using KD_Lib

distiller = RCO(teacher_model, student_model, train_loader, test_loader, teacher_
↪optimizer,
                student_optimizer, epoch_interval=5, device=device)
distiller.train_teacher(epochs=20)                                      # Train the␣
↪teacher model
distiller.train_students(epochs=20)                                     # Train the␣
↪student model
distiller.evaluate(teacher=True)                                        # Evaluate␣
↪the teacher model
distiller.evaluate()                                                    # Evaluate␣
↪the student model
```

3.6 Self Training using KD_Lib

Paper

- The student model is first trained in the normal way to obtain a pre-trained model, which is then used as the teacher to train itself by transferring soft targets

To use the self training algorithm to train a student on MNIST for 5 epcohs -

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import SelfTraining
```

(continues on next page)

(continued from previous page)

```
# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

# Set device to be trained on

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Define student model

student_model = <your model>

# Define optimizer

student_optimizer = optim.SGD(student_model.parameters(), lr=0.01)


# Train using KD_Lib

distiller = SelfTraining(student_model, train_loader, test_loader, student_optimizer,
                         device=device)
distiller.train_student(epochs=5)                                    # Train the␣
↪student model
distiller.evaluate()                                                  # Evaluate␣
↪the student model
```

3.7 Hyperparameter Tuning using Optuna

Hyperparameter optimization is one of the crucial steps in training machine learning models. It is often quite a tedious process with many parameters to optimize and long training times for models. Optuna is an automatic hyperparameter optimization software framework, particularly designed for machine learning You can find more about Optuna here.

Optuna an be installed using *pip* -

```
$ pip install optuna
```

or using *conda* -

```
$ conda install -c conda-forge optuna
```

To search for the best hyperparameters for the VanillaKD algorithm -

```
import torch
import torch.optim as optim
from torchvision import datasets, transforms
from KD_Lib.KD import VanillaKD

import optuna
from sklearn.externals import joblib

# Define datasets, dataloaders, models and optimizers

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "mnist_data",
        train=False,
        transform=transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        ),
    ),
    batch_size=32,
    shuffle=True,
)

# Set device to be trained on

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Optuna requires defining an objective function
# The hyperparameters are then optimized for maximizing/minimizing this objective
↪function

def tune_VanillaKD(trial):

    teacher_model = <your model>
    student_model = <your model>
```

(continues on next page)

(continued from previous page)

```
    # Define hyperparams and choose what ranges they should be trialled for

    lr = trial.suggest_float("lr", 1e-4, 1e-1)
    momentum = trial.suggest_float("momentum", 0.9, 0.99)
    optimizer = trial.suggest_categorical('optimizer',[optim.SGD, optim.Adam])

    teacher_optimizer = optimizer(teacher_model.parameters(), lr, momentum)
    student_optimizer = optimizer(student_model.parameters(), lr, momentum)

    temperature = trial.suggest_float("temperature", 5.0, 20.0)
    distil_weight = trial.suggest_float("distil_weight", 0.0, 1.0)

    loss_fn = trial.suggest_categorical("loss_fn",[nn.KLDivLoss(), nn.MSELoss()])

    # Instiate disitller object using KD_Lib and train

    distiller = VanillaKD(teacher_model, student_model, train_loader, test_loader,
                          teacher_optimizer, student_optimizer, loss_fn,
                          temperature, distil_weight, device)
    distiller.train_teacher(epochs=10)
    distiller.train_student(epochs=10)
    test_accuracy = disitller.evaluate()

    # The objective function must return the quantity we're trying to maximize/
↪minimize

    return test_accuracy

# Create a study

study = optuna.create_study(study_name="Hyperparameter Optimization",
                            direction="maximize")
study.optimize(tune_VanillaKD, n_trials=10)

# Access results

results = study.trials_dataframe()
results.head()

# Get best values of hyperparameter

for key, value in study.best_trial.__dict__.items():
print("{} : {}".format(key, value))

# Write results of the study

joblib.dump(study, <your path>)

# Access results at a later time

study = joblib.load(<your path>)
results = study.trials_dataframe()
results.head()
```

CHAPTER 4

Knowledge Distillation

4.1 Vision

4.1.1 KD_Lib.KD.vision.BANN

KD_Lib.KD.vision.BANN.BANN module

4.1.2 KD_Lib.KD.vision.DML

KD_Lib.KD.vision.DML.dml module

4.1.3 KD_Lib.KD.vision.KA

KD_Lib.KD.vision.KA.LSR module

KD_Lib.KD.vision.KA.PS module

4.1.4 KD_Lib.KD.vision.RCO

KD_Lib.KD.vision.RCO.rco module

4.1.5 KD_Lib.KD.vision.RKD

KD_Lib.KD.vision.RKD.loss_metric module

4.1.6 KD_Lib.KD.vision.TAKD

KD_Lib.KD.vision.TAKD.takd module

4.1.7 KD_Lib.KD.vision.CSKD

KD_Lib.KD.vision.CSKD.cdkd module

4.1.8 KD_Lib.KD.vision.attention

KD_Lib.KD.vision.attention.attention module

KD_Lib.KD.vision.attention.loss_metric module

4.1.9 KD_Lib.KD.vision.mean_teacher

KD_Lib.KD.vision.mean_teacher.mean_teacher module

4.1.10 KD_Lib.KD.vision.noisy

KD_Lib.KD.vision.noisy.messy_collab module

KD_Lib.KD.vision.noisy.noisy_teacher module

KD_Lib.KD.vision.noisy.soft_random module

KD_Lib.KD.vision.noisy.utils module

4.1.11 KD_Lib.KD.vision.teacher_free

KD_Lib.KD.vision.teacher_free.self_training module

KD_Lib.KD.vision.teacher_free.virtual_teacher module

4.1.12 KD_Lib.KD.vision.vanilla

KD_Lib.KD.vision.vanilla.vanilla_kd module

4.2 Text

4.2.1 KD_Lib.KD.text.BERT2LSTM package

Submodules

KD_Lib.KD.text.BERT2LSTM.bert2lstm module

KD_Lib.KD.text.BERT2LSTM.utils module

Module contents

4.2.2 KD_Lib.KD.text.utils package

Submodules

KD_Lib.KD.text.utils.bert module

`KD_Lib.KD.text.utils.bert.`**`df_to_bert_dataset`**(*df*, *max_length*, *tokenizer*)

`KD_Lib.KD.text.utils.bert.`**`df_to_bert_format`**(*df*, *max_length*, *tokenizer*)

`KD_Lib.KD.text.utils.bert.`**`get_bert_dataloader`**(*df*, *tokenizer*, *max_seq_length=64*, *batch_size=16*, *mode='train'*)

Helper function for generating dataloaders for BERT

Module contents

4.3 Common

4.3.1 KD_Lib.KD.common.base_class module

`class` `KD_Lib.KD.common.base_class.`**`BaseClass`**(*teacher_model*, *student_model*, *train_loader*, *val_loader*, *optimizer_teacher*, *optimizer_student*, *loss_fn=KLDivLoss()*, *temp=20.0*, *distil_weight=0.5*, *device='cpu'*, *log=False*, *logdir='./Experiments'*)

Bases: `object`

Basic implementation of a general Knowledge Distillation framework

Parameters

- **`(torch.nn.Module)`** (*`loss_fn`*) – Teacher model
- **`(torch.nn.Module)`** – Student model
- **`(torch.utils.data.DataLoader)`** (*`val_loader`*) – Dataloader for training
- **`(torch.utils.data.DataLoader)`** – Dataloader for validation/testing
- **`(torch.optim.*)`** (*`optimizer_student`*) – Optimizer used for training teacher
- **`(torch.optim.*)`** – Optimizer used for training student
- **`(torch.nn.Module)`** – Loss Function used for distillation
- **`(float)`** (*`distil_weight`*) – Temperature parameter for distillation
- **`(float)`** – Weight paramter for distillation loss
- **`(str)`** (*`logdir`*) – Device used for training; 'cpu' for cpu and 'cuda' for gpu
- **`(bool)`** (*`log`*) – True if logging required
- **`(str)`** – Directory for storing logs

`calculate_kd_loss`(*y_pred_student*, *y_pred_teacher*, *y_true*)

Custom loss function to calculate the KD loss for various implementations

Parameters

- **`(Tensor)`** (*`y_true`*) – Predicted outputs from the student network
- **`(Tensor)`** – Predicted outputs from the teacher network
- **`(Tensor)`** – True labels

evaluate(*teacher=False*)
Evaluate method for printing accuracies of the trained network

Parameters **(bool)** (*teacher*) – True if you want accuracy of the teacher network

get_parameters()
Get the number of parameters for the teacher and the student network

post_epoch_call(*epoch*)
Any changes to be made after an epoch is completed.

:param epoch (int) : current epoch number :return : nothing (void)

train_student(*epochs=10*, *plot_losses=True*, *save_model=True*, *save_model_pth='./models/student.pt'*)
Function that will be training the student

Parameters

- **(int)** (*epochs*) – Number of epochs you want to train the teacher
- **(bool)** (*save_model*) – True if you want to plot the losses
- **(bool)** – True if you want to save the student model
- **(str)** (*save_model_pth*) – Path where you want to save the student model

train_teacher(*epochs=20*, *plot_losses=True*, *save_model=True*, *save_model_pth='./models/teacher.pt'*)
Function that will be training the teacher

Parameters

- **(int)** (*epochs*) – Number of epochs you want to train the teacher
- **(bool)** (*save_model*) – True if you want to plot the losses
- **(bool)** – True if you want to save the teacher model
- **(str)** (*save_model_pth*) – Path where you want to store the teacher model

CHAPTER 5

Pruning

5.1 KD_Lib.Pruning.lottery_tickets

5.1.1 KD_Lib.Pruning.lottery_tickets.lottery_tickets module

class KD_Lib.Pruning.lottery_tickets.lottery_tickets.**LotteryTicketsPruner**(*model*, *train_loader*, *test_loader*, *loss_fn=CrossEntropyLo* *de-* *vice='cpu'*)

Bases: KD_Lib.Pruning.common.iterative_base_class.BaseIterativePruner

Implementation of Lottery Tickets Pruning for PyTorch models.

Parameters

- **model** (*torch.nn.Module*) – Model that needs to be pruned
- **train_loader** (*torch.utils.data.DataLoader*) – Dataloader for training
- **test_loader** (*torch.utils.data.DataLoader*) – Dataloader for validation/testing
- **loss_fn** (*torch.nn.Module*) – Loss function to be used for training
- **device** (*torch.device*) – Device used for implementation ("cpu" by default)

prune_model(*prune_percent=10*)

Function used for pruning

Parameters prune_percent (*int*) – Pruning percent per iteration (percentage of alive weights to zero per pruning iteration)

CHAPTER 6

Quantization

6.1 KD_Lib.Quantization.common

6.1.1 KD_Lib.Quantization.common.base_class module

class KD_Lib.Quantization.common.base_class.**Quantizer**(*model*, *qconfig*, *train_loader=None*, *test_loader=None*, *optimizer=None*, *criterion=None*, *device=device(type='cpu')*)

Bases: `object`

Basic Implementation of Quantization for PyTorch models.

Parameters

- **model** (*torch.nn.Module*) – Model that needs to be pruned
- **qconfig** (*Qconfig*) – Configuration used for quantization
- **train_loader** (*torch.utils.data.DataLoader*) – DataLoader used for training
- **test_loader** (*torch.utils.data.DataLoader*) – DataLoader used for testing
- **optimizer** (*torch.optim.**) – Optimizer for training
- **criterion** (*Loss_fn*) – Loss function used for calibration
- **device** (*torch.device*) – Device used for training ("cpu" or "cuda")

get_model_sizes()

Function for printing sizes of the original and quantized model

get_performance_statistics()

Function used for reporting inference performance of original and quantized models Note that performance here referes to the following: 1. Accuracy achieved on the testset 2. Time taken for evaluating on the testset

quantize()
Function used for quantization

6.2 KD_Lib.Quantization.dynamic

6.2.1 KD_Lib.Quantization.dynamic.dynamic_quantization module

class KD_Lib.Quantization.dynamic.dynamic_quantization.**Dynamic_Quantizer**(*model*, *test_loader*, *qconfig_spec=None*)

Bases: *KD_Lib.Quantization.common.base_class.Quantizer*

Implementation of Dynamic Quantization for PyTorch models.

Parameters

- **model** (*torch.nn.Module*) – Model that needs to be quantized
- **qconfig_spec** (*Qconfig_spec*) – Qconfig spec
- **test_loader** (*torch.utils.data.DataLoader*) – DataLoader used for testing

quantize(*dtype=torch.qint8*, *mapping=None*)
Function used for quantization

Parameters

- **dtype** (*torch.dtype*) – dtype for quantized modules
- **mapping** (*mapping*) – maps type of a submodule to a type of corresponding dynamically quantized version with which the submodule needs to be replaced

6.3 KD_Lib.Quantization.static

6.3.1 KD_Lib.Quantization.static.static_quantization module

class KD_Lib.Quantization.static.static_quantization.**Static_Quantizer**(*model*, *train_loader*, *test_loader*, *qconfig=QConfig(activation=functo* *'torch.quantization.observer.M* *reduce_range=True)*, *weight=functools.partial(<cla* *'torch.quantization.observer.M* *dtype=torch.qint8*, *qscheme=torch.per_tensor_sy* *criterion=CrossEntropyLoss()*, *device=device(type='cpu')*)

Bases: *KD_Lib.Quantization.common.base_class.Quantizer*

Implementation of Static Quantization for PyTorch models.

Parameters

- **model** (*torch.nn.Module*) – Model that needs to be pruned
- **qconfig** (*Qconfig*) – Configuration used for quantization
- **train_loader** (*torch.utils.data.DataLoader*) – DataLoader used for training (calibration)
- **test_loader** (*torch.utils.data.DataLoader*) – DataLoader used for testing
- **criterion** (*Loss_fn*) – Loss function used for calibration
- **device** (*torch.device*) – Device used for training ("cpu" or "cuda")

quantize(*num_calibration_batches=10*)

Function used for quantization

Parameters **num_calibration_batches** (*int*) – Number of batches used for calibration

6.4 KD_Lib.Quantization.qat

6.4.1 KD_Lib.Quantization.qat.qat module

class KD_Lib.Quantization.qat.qat.**QAT_Quantizer**(*model, train_loader, test_loader, optimizer, qconfig=QConfig(activation=functools.partial(<class 'torch.quantization.fake_quantize.FakeQuantize'>, observer=<class 'torch.quantization.observer.MovingAverageMinMaxObserver'>, quant_min=0, quant_max=255, reduce_range=True), weight=functools.partial(<class 'torch.quantization.fake_quantize.FakeQuantize'>, observer=<class 'torch.quantization.observer.MovingAveragePerChannelMinMa quant_min=-128, quant_max=127, dtype=torch.qint8, qscheme=torch.per_channel_symmetric, reduce_range=False, ch_axis=0)), criterion=CrossEntropyLoss(), device=device(type='cpu')*)

Bases: *KD_Lib.Quantization.common.base_class.Quantizer*

Implementation of Quantization-Aware Training (QAT) for PyTorch models.

Parameters

- **model** (*torch.nn.Module*) – (Quantizable) Model that needs to be quantized
- **train_loader** (*torch.utils.data.DataLoader*) – DataLoader used for training
- **test_loader** (*torch.utils.data.DataLoader*) – DataLoader used for testing
- **optimizer** (*torch.optim.**) – Optimizer for training
- **qconfig** (*Qconfig*) – Configuration used for quantization

- **criterion** (*Loss_fn*) – Loss function used for training
- **device** (*torch.device*) – Device used for training ("cpu" or "cuda")

quantize(*num_train_epochs=10*, *num_train_batches=10*, *param_freeze_epoch=3*, *bn_freeze_epoch=2*)

Function used for quantization

Parameters

- **num_train_epochs** (*int*) – Number of epochs used for training
- **num_train_batches** (*int*) – Number of batches used for training
- **param_freeze_epoch** (*int*) – Epoch after which quantizer parameters need to be freezed
- **bn_freeze_epoch** (*int*) – Epoch after which batch norm mean and variance stats are freezed

CHAPTER 7

Models

7.1 KD_Lib.models.lenet module

class `KD_Lib.models.lenet.`**`LeNet`**(*img_size=32*, *num_classes=10*, *in_channels=3*)

Bases: `torch.nn.modules.module.Module`

Implementation of a LeNet model

Parameters

- **`(int)`** (*`in_channels`*) – Dimension of input image
- **`(int)`** – Hidden layer dimension
- **`(int)`** – Number of classes for classification
- **`(int)`** – Number of channels in input specimens

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class `KD_Lib.models.lenet.`**`ModLeNet`**(*img_size=32*, *num_classes=10*, *in_channels=3*)

Bases: `torch.nn.modules.module.Module`

Implementation of a ModLeNet model

Parameters

- **`(int)`** (*`in_channels`*) – Dimension of input image
- **`(int)`** – Hidden layer dimension

- **(int)** – Number of classes for classification
- **(int)** – Number of channels in input specimens

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

7.2 KD_Lib.models.lstm module

class `KD_Lib.models.lstm.`**LSTMNet**(*input_dim=100*, *embed_dim=50*, *hidden_dim=32*, *num_classes=2*, *num_layers=5*, *dropout_prob=0*, *bidirectional=False*, *pad_idx=0*)
Bases: `torch.nn.modules.module.Module`

Implementation of an LSTM model for classification

Parameters

- **(int)** (*batch_size*) – Size of the vocabulary
- **(int)** – Embedding dimension (word vector size)
- **(int)** – Hidden dimension for LSTM layers
- **(int)** – Number of classes for classification
- **(int)** – Dropout probability
- **(int)** – True if bidirectional LSTM needed
- **(int)** – Batch size of input

forward(*x*, *x_len*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

7.3 KD_Lib.models.nin module

class `KD_Lib.models.nin.`**NetworkInNetwork**(*num_classes=10*, *in_channels=3*)
Bases: `torch.nn.modules.module.Module`

Implementation of a Network In Network model

Parameters

- **(int)** (*in_channels*) – Number of classes for classification
- **(int)** – Number of channels in input specimens

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

7.4 KD_Lib.models.resnet module

class KD_Lib.models.resnet.**BasicBlock**(*in_planes*, *planes*, *stride=1*)
Bases: `torch.nn.modules.module.Module`

expansion = 1

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class KD_Lib.models.resnet.**Bottleneck**(*in_planes*, *planes*, *stride=1*)
Bases: `torch.nn.modules.module.Module`

expansion = 4

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class KD_Lib.models.resnet.**MeanResnet**(*block*, *num_blocks*, *params*, *num_channel=3*, *num_classes=10*)
Bases: *KD_Lib.models.resnet.ResNet*

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class KD_Lib.models.resnet.**ResNet**(*block*, *num_blocks*, *params*, *num_channel=3*, *num_classes=10*)

Bases: `torch.nn.modules.module.Module`

forward(*x*, *out_feature=False*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

KD_Lib.models.resnet.**ResNet101**(*parameters*, *num_channel=3*, *num_classes=10*, *att=False*, *mean=False*)

Function that creates a ResNet 101 model

Parameters

- **(list or tuple)** (*parameters*) – List of parameters for the model
- **(int)** (*num_classes*) – Number of channels in input specimens
- **(int)** – Number of classes for classification
- **(bool)** (*mean*) – True if attention needs to be used
- **(bool)** – True if mean teacher model needs to be used

KD_Lib.models.resnet.**ResNet152**(*parameters*, *num_channel=3*, *num_classes=10*, *att=False*, *mean=False*)

Function that creates a ResNet 152 model

Parameters

- **(list or tuple)** (*parameters*) – List of parameters for the model
- **(int)** (*num_classes*) – Number of channels in input specimens
- **(int)** – Number of classes for classification
- **(bool)** (*mean*) – True if attention needs to be used
- **(bool)** – True if mean teacher model needs to be used

KD_Lib.models.resnet.**ResNet18**(*parameters*, *num_channel=3*, *num_classes=10*, *att=False*, *mean=False*)

Function that creates a ResNet 18 model

Parameters

- **(list or tuple)** (*parameters*) – List of parameters for the model
- **(int)** (*num_classes*) – Number of channels in input specimens
- **(int)** – Number of classes for classification
- **(bool)** (*mean*) – True if attention needs to be used
- **(bool)** – True if mean teacher model needs to be used

`KD_Lib.models.resnet.`**`ResNet34`**(*parameters*, *num_channel=3*, *num_classes=10*, *att=False*, *mean=False*)

Function that creates a ResNet 34 model

Parameters

- **(list or tuple)** (*parameters*) – List of parameters for the model
- **(int)** (*num_classes*) – Number of channels in input specimens
- **(int)** – Number of classes for classification
- **(bool)** (*mean*) – True if attention needs to be used
- **(bool)** – True if mean teacher model needs to be used

`KD_Lib.models.resnet.`**`ResNet50`**(*parameters*, *num_channel=3*, *num_classes=10*, *att=False*, *mean=False*)

Function that creates a ResNet 50 model

Parameters

- **(list or tuple)** (*parameters*) – List of parameters for the model
- **(int)** (*num_classes*) – Number of channels in input specimens
- **(int)** – Number of classes for classification
- **(bool)** (*mean*) – True if attention needs to be used
- **(bool)** – True if mean teacher model needs to be used

class `KD_Lib.models.resnet.`**`ResnetWithAT`**(*block*, *num_blocks*, *params*, *num_channel=3*, *num_classes=10*)

Bases: *KD_Lib.models.resnet.ResNet*

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

7.5 KD_Lib.models.shallow module

class `KD_Lib.models.shallow.`**`Shallow`**(*img_size=28*, *hidden_size=800*, *num_classes=10*, *num_channels=1*)

Bases: `torch.nn.modules.module.Module`

Implementation of a Shallow model

Parameters

- **(int)** (*num_classes*) – Dimension of input image
- **(int)** – Hidden layer dimension
- **(int)** – Number of classes for classification

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

B

C

D

E

F

G

K

L

M

N

P

Q

R

S

T