Can You Learn an Algorithm? Generalizing from Easy to Hard Problems with Recurrent Networks

Özge Güney - 2100365

1-)The main problem addressed in the original paper

Deep neural networks are powerful machines for visual pattern recognition. However, while reasoning tasks are easy for humans, they can still be difficult for neural models. People often learn reasoning strategies on simple problems and they use them to solve difficult problems with longer thinking. For example, a person learning to solve small mazes can solve much larger mazes using the same techniques by spending more time thinking more. In computers, this behavior is usually achieved by using more difficult problem algorithms, but they do more computation and need more cost.

This study examines that trained recurrent networks solve simple problems with few recurrent steps and they can actually solve more complex problems by doing additional recurrences during test (inference). In this study, it is found that recurrent networks can solve harder problems simply by increasing their test time iteration budget. This means they think longer than they do at train time. In addition to this, they found that the performance of iterative models increased with iteration, without adding parameters or retraining. This ability is specific to recurrent networks. As opposed to hard-coded algorithms, this work interested in examining if learned processes can be generalized to even more difficult problems from the data they were trained on. For example, there are many ways to solve mazes: breadth first search is classical and value iteration networks is learned algorithm. But, they need more cost and more time.

This work demonstrates this algorithmic behavior of recurrent networks on prefix sum computation, mazes, and chess. In all three domains, networks trained on simple problem instances are able to improve their reasoning abilities at test time simply by thinking for longer.

2-) Description of the dataset used in the original paper

The datasets can all be downloaded from https://github.com/aks2203/easy-to-hard-data.

The datasets in this work are designed for studying easy to hard generalization. The training data consists of easy examples, and the testing data has harder examples. The datasets are as follows.

1. Prefix Sums

Each training sample is a binary string. The goal is to output a binary string of equal length, where each bit represents the cumulative sum of input bits modulo two. This models accept input strings of any size, and it is considered longer strings to be more difficult to process than shorter ones. Each dataset contains 10,000 uniform random binary strings without duplicates. Datasets with input lengths of 32, 44, and 48 bits are used.

- Compute the prefix sum modulo two of a binary input string.
- The length of the string determines the difficulty of the problem.
- Provide 52 different sets (10,000 samples per length) from which to choose one for training data and a longer one for testing.

2. Mazes

Mazes are generated by using a depth-first search algorithm. It is trained on 50,000 small (9 ⇥ 9) mazes, and it is tested on 10,000 larger (13 ⇥ 13) mazes. The models are convolutional, and receive a maze as a N ⇥ N three-channel image. The maze walls are black, and the start and goal locations are red and green. The label for each maze is a binary two-dimensional mask. The label is containing the locations of positions along the shortest path solution.

- Visually solve a maze where the input is a three channel image, and the output is a binary segmentation mask, which is the same size as the input, separating pixels, with ones at locations that are on the optimal path and zeros elsewhere.
- Provide many size mazes

3. Chess Puzzles

The third dataset is chess puzzle. The data is furnished by Lichess, an online open-source chess server. From this database, it is compiled labeled data where the inputs are 8 ⇥ 8 ⇥ 12 arrays. These are indicating the position of each piece on the board (one channel per piece type and color) . The outputs are 8 ⇥ 8 binary masks. They are showing the origin and destination positions for the optimal move.

- Choose the best next move
- The difficulty is determined by the Lichess puzzle rating.
- The first 600,000 easiest puzzles are for an easy training set. Testing can be done with any subset of puzzles with higher indices. The default test set uses indices 600,000 to 700,000.

3-) Methods used in the original paper

In this work, recurrent neural networks and feed forward models are compared. They explore the ability of recurrent neural networks to generalize to more difficult problems simply by thinking deeper. They find that recurrent networks can generalize to harder problems simply by increasing their test time iteration budget. They train and test on problems of different sizes/difficulties, their training and test distributions are disjoint, and systems must extrapolate to solve problems from the test distribution. They train networks to solve problems iteratively and the goal is to create recurrent architectures that are clever to learn an algorithm. There are three problems: computing prefix sums, solving mazes, and playing chess. For each problem, recurrent networks are trained on a set of "easy" problems using a constant number of iterations. After training is complete, they test the models extrapolation behaviors on "hard" problems with additional iterations. They find that recurrent models are even better at generalizing from easy to hard than their feed-forward. While there is only one way to test the feed-forward models, the recurrent models are allowed to think deeper about the harder problems.

The feed-forward prefix sum models are fully convolutional models that take in n ⇥ 1 arrays. The first layer is a one-dimensional convolution with a three entry wide kernel that strides by one entry with padding by one on either end on the input. The output of this first convolution has 120 channels of the same shape as the input. The next parts of the networks are residual blocks made up of four layers that are identical to the first layer with skip connections every two layers. After the residual blocks, there are three similar convolutional layers that output 60, 30, and two channels, respectively. For a network of depth d, there are (d 4)/4 residual blocks. The recurrent models are identical, except that all residual blocks share weights.

The feed-forward maze solving models are fully convolutional models that take in n ⇥ n ⇥ 3 arrays. The first layer is a two-dimensional convolution with a 3 ⇥ 3 kernel that strides by one entry and pads by one unit in each direction. The output of this first convolution has 128 channels of the same shape as the input. As above, the next parts of the networks are residual blocks made up of four layers that are identical to the first layer with skip connections every two layers. After the residual blocks, there are three similar convolutional layers that output 32, 8, and two channels, respectively. For a network of depth d, there are (d 4)/4 residual blocks. The recurrent models are identical, except that all residual blocks share weights.

The chess playing models are the same as the maze models except that the first layer takes 8 ⇥ 8 ⇥ 12 inputs and outputs 512 channels.

4-) Experimental results obtained in the original paper

The first task is computing prefix sums. They study the problem of computing the prefix sums modulo two of binary input strings. When computing prefix sums, they try models with effective depths from 40 to 68 layers. They train models on easy data consisting of 32-bit input strings and test on harder 40-bit and 44-bit strings. They compare recurrent models to the best feed-forward models of comparable effective depth and we see the result in Figure 1. It is easy to understand that recurrent models generalize from easy to hard better than feed-forward networks. When the thought budget, or number of iterations, is increased, we see that recurrent models can get upwards of 90% of the harder testing examples correct. Here are the average accuracies of models trained on 32-bit inputs and tested on 40-bit inputs.

	Effective Depth (Layers)		
	40	44	48
Recurrent	24.96 ± 2.96	31.02 ± 2.56	35.22 ± 3.34
Feed-forward	22.17 ± 0.85	24.78 ± 1.65	22.79 ± 1.32

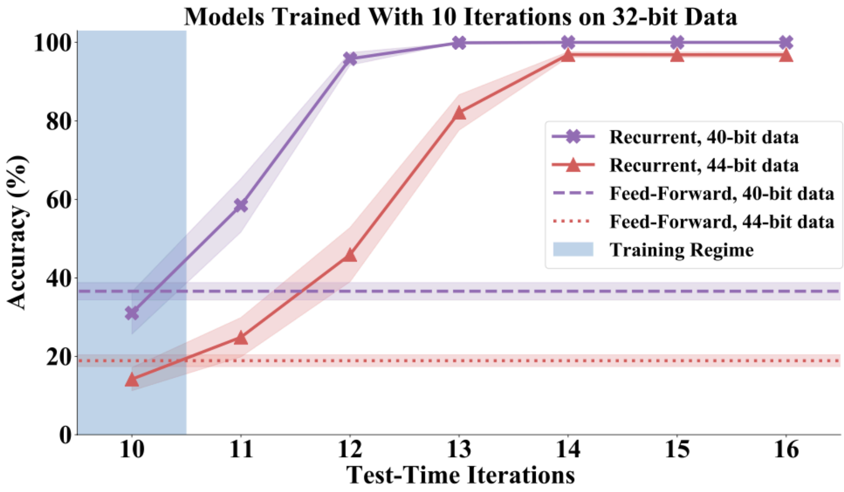

Figure 1: Extrapolating to longer input strings.

The second task is maze solving. They train models on a training set. It includes the easier small mazes, and they investigate the ability of networks by using larger, or harder mazes at test time. They find these results : First, the recurrent models make the leap from small to large mazes better than feed-forward models. Second, when allowed to think deeper, the recurrent models exhibit even higher performance. In Figure 2, we see that recurrent models can extrapolate to harder problems better than feed forward models. Models trained with 20 iterations can achieve upward of 70% accuracy on large mazes using 5 additional iterations at test time.

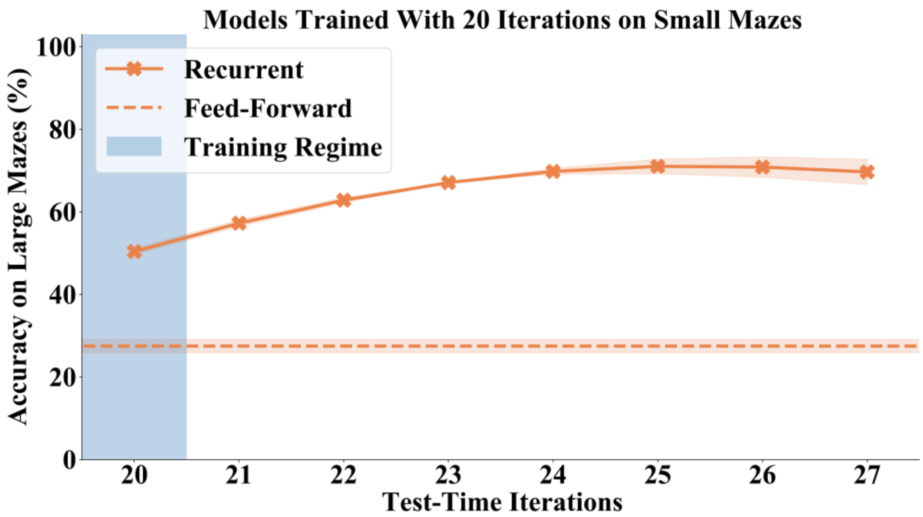

Figure 2: Generalizing from easy to hard mazes.

The third task is chess puzzles and they seek the best next move. Chess playing algorithms is complex and has components that use algorithms like Monte Carlo tree search as well as neural network based elements for evaluating positions. Once again, we see that recurrent models can solve more chess puzzles than their feed forward. Furthermore, by thinking deeper at test time, recurrent models can perform even better. In Figure 3, wee see recurrent models can solve more puzzles with more iterations.

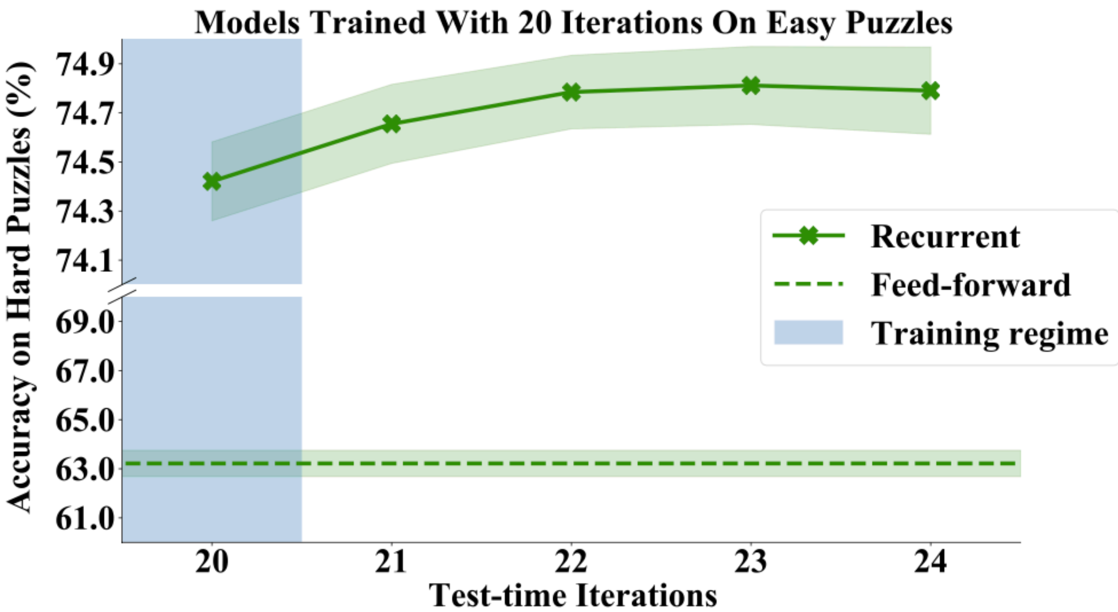

Figure 3: Generalizing from easy to hard chess puzzles.

5-) Contributions of the original paper

In this work, they demonstrate that neural networks are capable of solving sequential reasoning tasks and then extrapolating this knowledge to solve problems of greater complexity than they were trained on. These recurrent models are largely inspired by the classical theory of mind. They ask “Can we build neural networks that can think for even longer? “ And “Is it feasible to have models whose performance only increases with added compute time? “ They will be answered in the future. They believe that this paper raises some questions that motivates future work. They believe that they leave an in depth investigation into other neural network designs for future work.

To sum up, the resulting models outperform at solving problems that are classically solved by hand-crafted algorithms; prefix sums are computed using reduction trees, mazes are classically solved by depth/breadth first search, and chess is solved by Monte-Carlo tree search.

6-) The experiments you have performed on the related dataset

In this project, I use Python and mostly torch package. I compare recurrent neural networks and feed forward networks on prefix sum problem. Firstly, I create the data as import PrefixSumDataset from easy_to_hard_data. They provide 52 different sets. For each sequence length, they provide a set of 10,000 input/output pairs. And, I can change these parameters from the code:

```
train_batch_size:   int, Size of mini batches for training
test_batch_size:    int, Size of mini batches for testing
train_data:         int, Number of bits in the training set
eval_data:          int, Number of bits in the training set
train_split:        float, Portion of training data to use for training (vs. testing in-distribution)
parser.add_argument("--train_batch_size", default=128, type=int, help="batch size for training”)
parser.add_argument("--test_batch_size", default=500, type=int, help="batch size for testing”)
parser.add_argument("--train_data", default=16, type=int, help="what size train data”)
parser.add_argument("--eval_data", default=20, type=int, help="what size eval data”)
parser.add_argument("--depth", default=8, type=int, help="depth of the network”)
parser.add_argument("--width", default=4, type=int, help="width of the network")
train_split=0.8
dataset = PrefixSumDataset("./data", num_bits=train_data)
evalset = PrefixSumDataset("./data", num_bits=eval_data)
num_train = int(train_split * len(dataset))
trainset, testset = torch.utils.data.random_split(dataset,
                                [num_train, int(len(dataset) - num_train)],
                                generator=torch.Generator().manual_seed(42))
trainloader = data.DataLoader(trainset, num_workers=0, batch_size=train_batch_size,
                shuffle=shuffle, drop_last=True)
testloader = data.DataLoader(testset, num_workers=0, batch_size=test_batch_size,
                shuffle=False, drop_last=False)
evalloader = data.DataLoader(evalset, num_workers=0, batch_size=test_batch_size,
                shuffle=False, drop_last=False)
```

The samples in train data are 16-bits input string and the samples in test data are 20-bits input string. The batch size is equal to the number of samples in the dataset. In train dataset, number of samples is 128 and in test dataset, it is 500. As we see above, I use train split rate is 0.8.

Then, I create a module folder that includes two classes: recurrent_net.py and feed_forward_net.py. In these classes, I implement the algorithms. It is import to see the difference of accuracy between them. While test duration, for two algorithms, I define the number of iterations is 20 for training. Then, while test duration, I define the number of iterations are 25. During inference, both input lengths and iterations are increasing.

For training : parser.add_argument(“--test_iterations", default=20, type=int, help="number iterations”)

For testing : parser.add_argument("--test_iterations", default=25, type=int, nargs=“+", help="number iterations”)

Then, I change the model to see the difference between algorithm performance.

parser.add_argument("--model", default="recur_net", type=str, help="model for training”)

Then, I use “ff_net” instead of “recur_net”.

7-) The results obtained with your own experiments on the related dataset
As I change the above parameters, the accuracy change. As I increase the number of iterations during test time, when the other parameters are constant, the accuracy is increase. While using recur_net or recurrent neural network, accuracy increase higher than ff_net (feed forward neural network).

Source Code:

```
""" feed_forward_net.py
    Prefix sum solving convolutional neural network.

"""

import torch
import torch.nn as nn
import torch.nn.functional as F

from icecream import ic

# Ignore statemenst for pylint:
#     Too many branches (R0912), Too many statements (R0915), No member (E1101),
#     Not callable (E1102), Invalid name (C0103), No exception (W0702)
# pylint: disable=R0912, R0915, E1101, E1102, C0103, W0702, R0914


class BasicBlock(nn.Module):
    """Basic residual block class"""
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv1d(
            in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)

        self.conv2 = nn.Conv1d(planes, planes, kernel_size=3,
                               stride=1, padding=1, bias=False)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv1d(in_planes, self.expansion*planes,
                          kernel_size=1, stride=stride, bias=False)
            )

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = self.conv2(out)
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class FFNet(nn.Module):
    """Modified ResidualNetworkSegment model class"""

    def __init__(self, block, num_blocks, width, depth):
        super(FFNet, self).__init__()
        assert (depth - 4) % 4 == 0, "Depth not compatible with recurrent architectue."
        self.iters = (depth - 4) // 4
        self.in_planes = int(width)
```

```
        self.conv1 = nn.Conv1d(1, width, kernel_size=3,
                               stride=1, padding=1, bias=False)
        layers = []
        for _ in range(self.iters):
            for i in range(len(num_blocks)):
                layers.append(self._make_layer(block, width, num_blocks[i], stride=1))

        self.recur_block = nn.Sequential(*layers)
        self.conv2 = nn.Conv1d(width, width, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.conv3 = nn.Conv1d(width, int(width/2), kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.conv4 = nn.Conv1d(int(width/2), 2, kernel_size=3,
                               stride=1, padding=1, bias=False)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for strd in strides:
            layers.append(block(self.in_planes, planes, strd))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = self.recur_block(out)
        thought = F.relu(self.conv2(out))
        thought = F.relu(self.conv3(thought))
        thought = self.conv4(thought)
        return thought


def ff_net(depth, width, **kwargs):
    return FFNet(BasicBlock, [2], width, depth)
```

""" **recurrent_net.py**

```
    Parity solving recurrent convolutional neural network.

"""

import torch
import torch.nn as nn
import torch.nn.functional as F

from icecream import ic

# Ignore statemenst for pylint:
#     Too many branches (R0912), Too many statements (R0915), No member (E1101),
#     Not callable (E1102), Invalid name (C0103), No exception (W0702)
# pylint: disable=R0912, R0915, E1101, E1102, C0103, W0702, R0914


class BasicBlock(nn.Module):
    """Basic residual block class"""
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
```

```
        self.conv1 = nn.Conv1d(
            in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)

        self.conv2 = nn.Conv1d(planes, planes, kernel_size=3,
                               stride=1, padding=1, bias=False)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv1d(in_planes, self.expansion*planes,
                          kernel_size=1, stride=stride, bias=False)
            )

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = self.conv2(out)
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class RecurNet(nn.Module):
    """Modified ResidualNetworkSegment model class"""

    def __init__(self, block, num_blocks, width, depth):
        super(RecurNet, self).__init__()
        assert (depth - 4) % 4 == 0, "Depth not compatible with recurrent architecture."
        self.iters = (depth - 4) // 4
        self.in_planes = int(width)
        self.conv1 = nn.Conv1d(1, width, kernel_size=3,
                               stride=1, padding=1, bias=False)
        layers = []
        for i in range(len(num_blocks)):
            layers.append(self._make_layer(block, width, num_blocks[i], stride=1))

        self.recur_block = nn.Sequential(*layers)
        self.conv2 = nn.Conv1d(width, width, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.conv3 = nn.Conv1d(width, int(width/2), kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.conv4 = nn.Conv1d(int(width/2), 2, kernel_size=3,
                               stride=1, padding=1, bias=False)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for strd in strides:
            layers.append(block(self.in_planes, planes, strd))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        if self.training:
            self.thoughts = None

            out = F.relu(self.conv1(x))
            for i in range(self.iters):
                out = self.recur_block(out)
            thought = F.relu(self.conv2(out))
            thought = F.relu(self.conv3(thought))
```

```
            thought = self.conv4(thought)
        else:
            self.thoughts = torch.zeros((self.iters, x.size(0), 2, x.size(2))).to(x.device)

            out = F.relu(self.conv1(x))
            for i in range(self.iters):
                out = self.recur_block(out)
                thought = F.relu(self.conv2(out))
                thought = F.relu(self.conv3(thought))
                self.thoughts[i] = self.conv4(thought)
            thought = self.thoughts[-1]

        return thought


def recur_net(depth, width, **kwargs):
    return RecurNet(BasicBlock, [2], width, depth)
```

```
"""train.py
   Train, test, and save models
   Developed as part of Easy-To-Hard project
   April 2021
"""

import argparse
import os
import sys
from collections import OrderedDict

from icecream import ic
import numpy as np
import torch
from torch.optim.lr_scheduler import MultiStepLR, CosineAnnealingLR
# from torch.utils.tensorboard import SummaryWriter

import warmup
from utils import train, test, OptimizerWithSched, load_model_from_checkpoint, get_dataloaders, to_json,
get_optimizer, to_log_file, now, get_model


# Ignore statements for pylint:
#     Too many branches (R0912), Too many statements (R0915), No member (E1101),
#     Not callable (E1102), Invalid name (C0103), No exception (W0702),
#     Too many local variables (R0914), Missing docstring (C0116, C0115).
# pylint: disable=R0912, R0915, E1101, E1102, C0103, W0702, R0914, C0116, C0115


def main():

    print("\n_____________________________________________________\n")
    print(now(), "train.py main() running.")

    parser = argparse.ArgumentParser(description="Deep Thinking")
    parser.add_argument("--checkpoint", default="check_default", type=str,
                        help="where to save the network")
    parser.add_argument("--clip", default=1.0, help="max gradient magnitude for training")
    parser.add_argument("--data_path", default="../data", type=str, help="path to data files")
    parser.add_argument("--debug", action="store_true", help="debug?")
    parser.add_argument("--depth", default=8, type=int, help="depth of the network")
```

```
parser.add_argument("--epochs", default=20, type=int, help="number of epochs for training")
parser.add_argument("--eval_data", default=20, type=int, help="what size eval data")
parser.add_argument("--json_name", default="test_stats", type=str, help="name of the json file")
parser.add_argument("--lr", default=0.1, type=float, help="learning rate")
parser.add_argument("--lr_decay", default="step", type=str, help="which kind of lr decay")
parser.add_argument("--lr_factor", default=0.1, type=float, help="learning rate decay factor")
parser.add_argument("--lr_schedule", nargs="+", default=[100, 150], type=int,
                    help="how often to decrease lr")
parser.add_argument("--model", default="recur_net", type=str, help="model for training")
parser.add_argument("--model_path", default=None, type=str, help="where is the model saved?")
parser.add_argument("--no_shuffle", action="store_false", dest="shuffle",
                    help="shuffle training data?")
parser.add_argument("--optimizer", default="sgd", type=str, help="optimizer")
parser.add_argument("--output", default="output_default", type=str, help="output subdirectory")
parser.add_argument("--save_json", action="store_true", help="save json")
parser.add_argument("--save_period", default=None, type=int, help="how often to save")
parser.add_argument("--test_batch_size", default=200, type=int, help="batch size for testing")
parser.add_argument("--test_iterations", default=25, type=int,
                    help="how many, if testing with a different number iterations")
parser.add_argument("--test_mode", default="default", type=str, help="testing mode")
parser.add_argument("--train_batch_size", default=200, type=int, help="batch size for training")
parser.add_argument("--train_data", default=16, type=int, help="what size train data")
parser.add_argument("--train_log", default="train_log.txt", type=str,
                    help="name of the log file")
parser.add_argument("--train_mode", default="xent", type=str, help="training mode")
parser.add_argument("--train_split", default=0.8, type=float,
                    help="percentile of difficulty to train on")
parser.add_argument("--val_period", default=20, type=int, help="how often to validate")
parser.add_argument("--warmup_period", default=5, type=int, help="warmup period")
parser.add_argument("--width", default=4, type=int, help="width of the network")

args = parser.parse_args()
args.train_mode, args.test_mode = args.train_mode.lower(), args.test_mode.lower()
device = "cuda" if torch.cuda.is_available() else "cpu"

if args.save_period is None:
    args.save_period = args.epochs

for arg in vars(args):
    print(f"{arg}: {getattr(args, arg)}")

# TensorBoard
train_log = args.train_log
try:
    array_task_id = train_log[:-4].split("_")[-1]
except:
    array_task_id = 1
# if not args.debug:
#     to_log_file(args, args.output, train_log)
#     writer = SummaryWriter(log_dir=f"{args.output}/runs/{train_log[:-4]}")
# else:
#     writer = SummaryWriter(log_dir=f"{args.output}/debug/{train_log[:-4]}")
####################################################
#              Dataset and Network and Optimizer
trainloader, testloader, evalloader = get_dataloaders(args.train_batch_size, args.test_batch_size, args.train_data,
                                                      args.eval_data, args.train_split, shuffle=args.shuffle)

# load model from path if a path is provided
if args.model_path is not None:
    print(f"Loading model from checkpoint {args.model_path}...")
```

```
    net, start_epoch, optimizer_state_dict = load_model_from_checkpoint(args.model,
                                                                       args.model_path,
                                                                       args.width,
                                                                       args.depth)
    start_epoch += 1

else:
    net = get_model(args.model, args.width, args.depth)
    start_epoch = 0
    optimizer_state_dict = None

if device == "cuda":
    device_ids = [int(i) for i in os.environ["CUDA_VISIBLE_DEVICES"].split(",")]
    if args.test_iterations and len(device_ids) > 1:
        print(f"{ic.format()}: Can't test on multiple GPUs. Exiting")
        sys.exit()
    net = torch.nn.DataParallel(net, device_ids=device_ids)
net = net.to(device)
pytorch_total_params = sum(p.numel() for p in net.parameters())
optimizer = get_optimizer(args.optimizer, args.model, net, args.lr)

if args.debug:
    print(net)
print(f"This {args.model} has {pytorch_total_params/1e6:0.3f} million parameters.")
print(f"Training will start at epoch {start_epoch}.")

if optimizer_state_dict is not None:
    print(f"Loading optimizer from checkpoint {args.model_path}...")
    optimizer.load_state_dict(optimizer_state_dict)
    warmup_scheduler = warmup.ExponentialWarmup(optimizer, warmup_period=0)
else:
    warmup_scheduler = warmup.ExponentialWarmup(optimizer, warmup_period=args.warmup_period)

if args.lr_decay.lower() == "step":
    lr_scheduler = MultiStepLR(optimizer, milestones=args.lr_schedule, gamma=args.lr_factor,
                               last_epoch=-1)
elif args.lr_decay.lower() == "cosine":
    lr_scheduler = CosineAnnealingLR(optimizer, args.epochs, eta_min=0, last_epoch=-1,
                                     verbose=False)
else:
    print(f"{ic.format()}: Learning rate decay style {args.lr_decay} not yet implemented."
          f"Exiting.")
    sys.exit()

optimizer_obj = OptimizerWithSched(optimizer, lr_scheduler, warmup_scheduler, args.clip)
torch.backends.cudnn.benchmark = True
####################################################

####################################################
#       Train
print(f"==> Starting training for {args.epochs - start_epoch} epochs...")

for epoch in range(start_epoch, args.epochs):

    loss, acc = train(net, trainloader, args.train_mode, optimizer_obj, device)

    print(f"{now()} Training loss at epoch {epoch}: {loss}")
    print(f"{now()} Training accuracy at epoch {epoch}: {acc}")

    # if the loss is nan, then stop the training
```

```
        if np.isnan(float(loss)):
            print(f"{ic.format()} Loss is nan, exiting...")
            sys.exit()

        # TensorBoard loss writing
        # writer.add_scalar("Loss/loss", loss, epoch)
        # writer.add_scalar("Accuracy/acc", acc, epoch)

        # for i in range(len(optimizer.param_groups)):
        #     writer.add_scalar(f"Learning_rate/group{i}", optimizer.param_groups[i]["lr"], epoch)

        if (epoch + 1) % args.val_period == 0:
            train_acc = test(net, trainloader, args.test_mode, device)
            test_acc = test(net, testloader, args.test_mode, device)
            eval_acc = test(net, evalloader, args.test_mode, device)
            # eval_acc = 0
            print(f"{now()} Training accuracy: {train_acc}")
            print(f"{now()} Testing accuracy: {test_acc}")
            print(f"{now()} Eval accuracy (hard data): {eval_acc}")

            stats = [train_acc, test_acc, eval_acc]
            stat_names = ["train_acc", "test_acc", "eval_acc"]
            for stat_idx, stat in enumerate(stats):
                stat_name = os.path.join("val", stat_names[stat_idx])
                # writer.add_scalar(stat_name, stat, epoch)

        if (epoch + 1) % args.save_period == 0 or (epoch + 1) == args.epochs:
            state = {
                "net": net.state_dict(),
                "epoch": epoch,
                "optimizer": optimizer.state_dict()
            }
            out_str = os.path.join(args.checkpoint,
                                   f"{args.model}_{args.optimizer}"
                                   f"_depth={args.depth}"
                                   f"_width={args.width}"
                                   f"_lr={args.lr}"
                                   f"_batchsize={args.train_batch_size}"
                                   f"_epoch={args.epochs-1}"
                                   f"_{array_task_id}.pth")

            print(f"{now()} Saving model to: ", args.checkpoint, " out_str: ", out_str)
            if not os.path.isdir(args.checkpoint):
                os.makedirs(args.checkpoint)
            torch.save(state, out_str)

    # writer.flush()
    # writer.close()
    ####################################################

    ####################################################
    #        Test
    print("==> Starting testing...")

    if int(args.test_iterations) > 0:
        assert isinstance(net.module.iters, int), f"{ic.format()} Cannot test " \
                                                  f"feed-forward model with iterations."
        net.module.iters = args.test_iterations

    test_acc = test(net, testloader, args.test_mode, device)
```

```python
    train_acc = test(net, trainloader, args.test_mode, device)
    eval_acc = test(net, evalloader, args.test_mode, device)
    # eval_acc = 0

    print(f"{now()} Training accuracy: {train_acc}")
    print(f"{now()} Testing accuracy: {test_acc}")
    print(f"{now()} Eval accuracy (hard data): {eval_acc}")

    model_name_str = f"{args.model}_depth={args.depth}_width={args.width}"
    stats = OrderedDict([("epochs", args.epochs),
                         ("eval_acc", eval_acc),
                         ("learning rate", args.lr),
                         ("lr", args.lr),
                         ("lr_factor", args.lr_factor),
                         ("model", model_name_str),
                         ("num_params", pytorch_total_params),
                         ("optimizer", args.optimizer),
                         ("test_acc", test_acc),
                         ("test_iter", args.test_iterations),
                         ("test_mode", args.test_mode),
                         ("train_acc", train_acc),
                         ("train_batch_size", args.train_batch_size),
                         ("train_mode", args.train_mode)])

    if args.save_json:
        args.json_name += ".json"
        to_json(stats, args.output, args.json_name)
    ####################################################


if __name__ == "__main__":
    main()
```

```python
""" utils.py
    utility functions and classes
    Developed as part of Easy-To-Hard project
    April 2021
"""

from collections import OrderedDict
from dataclasses import dataclass
import datetime
import json
import os
import sys

from easy_to_hard_data import PrefixSumDataset
from icecream import ic
import torch
import torch.utils.data as data
from torch.optim import SGD, Adam, AdamW, Adadelta
from tqdm import tqdm

from models.feed_forward_net import ff_net
from models.recurrent_net import recur_net
from models.recurrent_dilated_net import recur_dilated_net
from typing import Any
```

```
# Ignore statemenst for pylint:
#     Too many branches (R0912), Too many statements (R0915), No member (E1101),
#     Not callable (E1102), Invalid name (C0103), No exception (W0702),
#     Too many local variables (R0914), Missing docstring (C0116, C0115),
#     Unused import (W0611).
# pylint: disable=R0912, R0915, E1101, E1102, C0103, W0702, R0914, C0116, C0115, W0611


def get_dataloaders(train_batch_size, test_batch_size, train_data, eval_data, train_split=0.8, shuffle=True):
    """ Function to get pytorch dataloader objects
    input:
        dataset:            str, Name of the dataset
        train_batch_size:   int, Size of mini batches for training
        test_batch_size:    int, Size of mini batches for testing
        train_data:         int, Number of bits in the training set
        eval_data:          int, Number of bits in the training set
        train_split:        float, Portion of training data to use for training (vs. testing in-distribution)
        shuffle:            bool, Data shuffle switch
    return:
        trainloader:    Pytorch dataloader object with training data
        testloader:     Pytorch dataloader object with testing data
    """
    if train_split >= 1.0 or train_split <= 0:
        print(f"{ic.format()}: Split {train_split} is not between 0 and 1 in "
              f"get_dataloaders(). Exiting.")
        sys.exit()

    dataset = PrefixSumDataset("./data", num_bits=train_data)
    evalset = PrefixSumDataset("./data", num_bits=eval_data)

    num_train = int(train_split * len(dataset))

    trainset, testset = torch.utils.data.random_split(dataset,
                                                       [num_train, int(len(dataset) - num_train)],
                                                       generator=torch.Generator().manual_seed(42))


    trainloader = data.DataLoader(trainset, num_workers=0, batch_size=train_batch_size,
                                  shuffle=shuffle, drop_last=True)
    testloader = data.DataLoader(testset, num_workers=0, batch_size=test_batch_size,
                                 shuffle=False, drop_last=False)
    evalloader = data.DataLoader(evalset, num_workers=0, batch_size=test_batch_size,
                                 shuffle=False, drop_last=False)
    return trainloader, testloader, evalloader


def get_model(model, width, depth):
    """Function to load the model object
    input:
        model:      str, Name of the model
        width:      int, Width of network
        depth:      int, Depth of network
    return:
        net:        Pytorch Network Object
    """
    model = model.lower()
    net = eval(model)(depth=depth, width=width)
    return net
```

```
def get_optimizer(optimizer_name, model, net, lr):
    optimizer_name = optimizer_name.lower()
    model = model.lower()

    base_params = [p for n, p in net.named_parameters()]
    recur_params = []
    iters = 1

    # if "recur" in model:
    #     base_params = [p for n, p in net.named_parameters() if "recur" not in n]
    #     recur_params = [p for n, p in net.named_parameters() if "recur" in n]
    #     iters = net.iters
    # else:
    #     base_params = [p for n, p in net.named_parameters()]
    #     recur_params = []
    #     iters = 1

    all_params = [{"params": base_params}, {"params": recur_params, "lr": lr / iters}]

    if optimizer_name == "sgd":
        optimizer = SGD(all_params, lr=lr, weight_decay=2e-4, momentum=0.9)
    elif optimizer_name == "adam":
        optimizer = Adam(all_params, lr=lr, weight_decay=2e-4)
    elif optimizer_name == "adamw":
        optimizer = AdamW(all_params, lr=lr, betas=(0.9, 0.999), eps=1e-08, weight_decay=0.01,
                          amsgrad=False)
    elif optimizer_name == "adadelta":
        optimizer = Adadelta(all_params, lr=lr, rho=0.9, eps=1e-06, weight_decay=0)
    else:
        print(f"{ic.format()}: Optimizer choise of {optimizer_name} not yet implmented. Exiting.")
        sys.exit()

    return optimizer


def load_model_from_checkpoint(model, model_path, width, depth):
    net = get_model(model, width, depth)
    device = "cuda" if torch.cuda.is_available() else "cpu"
    state_dict = torch.load(model_path, map_location=device)
    state_dict["net"] = remove_parallel(state_dict["net"])
    net.load_state_dict(state_dict["net"])
    net = net.to(device)

    return net, state_dict["epoch"], state_dict["optimizer"]


def now():
    return datetime.datetime.now().strftime("%Y%m%d %H:%M:%S")


@dataclass
class OptimizerWithSched:
    """Attributes for optimizer, lr schedule, and lr warmup"""
    optimizer: Any
    scheduler: Any
    warmup: Any
    clip: Any
```

```
def remove_parallel(state_dict):
    """state_dict: state_dict of model saved with DataParallel()
    returns state_dict without extra module level"""
    new_state_dict = OrderedDict()
    for k, v in state_dict.items():
        name = k[7:] # remove module.
        new_state_dict[name] = v
    return new_state_dict


def test(net, testloader, mode, device):
    try:
        accuracy = eval(f"test_{mode}")(net, testloader, device)
    except NameError:
        print(f"{ic.format()}: test_{mode}() not implemented. Exiting.")
        sys.exit()
    return accuracy


def test_bit_wise_per_iter(net, testloader, device):
    net.eval()
    net.to(device)
    total = 0
    nm = net.module

    with torch.no_grad():
        for i, (inputs, targets) in tqdm(enumerate(testloader), leave=False):
            inputs, targets = inputs.to(device), targets.to(device)
            net(inputs)
            for j, thought in enumerate(nm.thoughts):
                predicted = thought.argmax(1)
                if i == 0 and j == 0:
                    correct = torch.zeros(len(nm.thoughts), inputs.size(-1)).to(device)
                correct[j] += (predicted == targets).sum(0)
            total += targets.size(0)

    accuracy = 100.0 * correct / total
    # print(accuracy)
    return accuracy


def test_bit_wise(net, testloader, device):
    net.eval()
    net.to(device)
    total = 0

    with torch.no_grad():
        for i, (inputs, targets) in tqdm(enumerate(testloader), leave=False):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)

            predicted = outputs.argmax(1)
            if i == 0:
                correct = (predicted == targets).sum(0)
            else:
                correct += (predicted == targets).sum(0)
            total += targets.size(0)

    accuracy = 100.0 * correct / total
    # print(accuracy)
```

```
    return accuracy


def test_default(net, testloader, device):
    net.eval()
    net.to(device)
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in tqdm(testloader, leave=False):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)

            predicted = outputs.argmax(1)
            correct += torch.amin(predicted == targets, dim=[1]).sum().item()

            total += targets.size(0)

    accuracy = 100.0 * correct / total
    return accuracy


def test_max_conf(net, testloader, device):
    net.eval()
    net.to(device)
    correct = 0
    total = 0
    softmax = torch.nn.functional.softmax
    nm = net.module

    with torch.no_grad():
        for inputs, targets in testloader:

            inputs, targets = inputs.to(device), targets.to(device)
            net(inputs)
            confidence_array = torch.zeros(nm.iters, inputs.size(0))
            for i, thought in enumerate(nm.thoughts):
                conf = softmax(thought.detach(), dim=1).max(1)[0]
                confidence_array[i] = conf.sum([1])

            exit_iter = confidence_array.argmax(0)
            best_thoughts = nm.thoughts[exit_iter, torch.arange(nm.thoughts.size(1))].squeeze()
            if best_thoughts.shape[0] != inputs.shape[0]:
                best_thoughts = best_thoughts.unsqueeze(0)
            predicted = best_thoughts.argmax(1)
            correct += torch.amin(predicted == targets, dim=[1]).sum().item()
            total += targets.size(0)

    accuracy = 100.0 * correct / total
    return accuracy


def to_json(stats, out_dir, log_name="test_stats.json"):
    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)
    fname = os.path.join(out_dir, log_name)

    if os.path.isfile(fname):
        with open(fname, "r") as fp:
```

```
            data_from_json = json.load(fp)
            num_entries = data_from_json["num entries"]
        data_from_json[num_entries] = stats
        data_from_json["num entries"] += 1
        with open(fname, "w") as fp:
            json.dump(data_from_json, fp)
    else:
        data_from_json = {0: stats, "num entries": 1}
        with open(fname, "w") as fp:
            json.dump(data_from_json, fp)


def to_log_file(out_dict, out_dir, log_name="log.txt"):
    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)
    fname = os.path.join(out_dir, log_name)

    with open(fname, "a") as fh:
        fh.write(str(now()) + " " + str(out_dict) + "\n" + "\n")

    print("logging done in " + out_dir + ".")


def train(net, trainloader, mode, optimizer_obj, device):
    try:
        train_loss, acc = eval(f"train_{mode}")(net, trainloader, optimizer_obj, device)
    except NameError:
        print(f"{ic.format()}: train_{mode}() not implemented. Exiting.")
        sys.exit()
    return train_loss, acc


def train_xent(net, trainloader, optimizer_obj, device):
    net.train()
    net = net.to(device)
    optimizer = optimizer_obj.optimizer
    lr_scheduler = optimizer_obj.scheduler
    warmup_scheduler = optimizer_obj.warmup
    criterion = torch.nn.CrossEntropyLoss()

    train_loss = 0
    correct = 0
    total = 1

    for inputs, targets in tqdm(trainloader, leave=False):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs).squeeze()
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*targets.size(0)
        predicted = outputs.argmax(1)
        correct += torch.amin(predicted == targets, dim=[1]).sum().item()
        total += targets.size(0)

    train_loss = train_loss / total
    acc = 100.0 * correct / total

    lr_scheduler.step()
```

```
    warmup_scheduler.dampen()

    return train_loss, acc


def train_xent_clipped(net, trainloader, optimizer_obj, device):
    net.train()
    net = net.to(device)
    optimizer = optimizer_obj.optimizer
    lr_scheduler = optimizer_obj.scheduler
    warmup_scheduler = optimizer_obj.warmup
    clip = optimizer_obj.clip
    criterion = torch.nn.CrossEntropyLoss()

    train_loss = 0
    correct = 0
    total = 1

    for inputs, targets in tqdm(trainloader, leave=False):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs).squeeze()
        loss = criterion(outputs, targets)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()
        train_loss += loss.item()*targets.size(0)
        predicted = outputs.argmax(1)
        correct += torch.amin(predicted == targets, dim=[1]).sum().item()
        total += targets.size(0)

    train_loss = train_loss / total
    acc = 100.0 * correct / total

    lr_scheduler.step()
    warmup_scheduler.dampen()

    return train_loss, acc
```