Names: Fionna Zhang, Katherine Lee, YuJu Ku
NetIds: fwz2, ksl103, yk544

Question 1

How should the state space (current information) and action space (action selected) be represented for the model? How does it capture the relevant information, in a relevant way, for your model space? One thing to consider here is local vs global information.

Initially, we tried to capture all global information that the Repeated Forward A* agent would have had at each step along the path, including: the known gridworld at each step (dim x dim gridworld), the immediate blocked neighbors, the current position, the previous position, the goal, the next move along the path, whether we have hit a block, whether we are at the goal, as well as the goal_x and goal_y. However, we found that capturing this amount of information was infeasible. Specifically, capturing the entire known gridworld at each step was infeasible because the generated data would be about 49GB for an appropropriate number of grids to train on. In addition, it provided a lot of information that the current agent at a given step did not require to make its next move.

From here, we moved to capturing only the local information that the agent would have, and represented all information in the form of a grid, using the following to denote the block's known state:

- 0 = empty, unknown
- 1 = block, off the grid
- 2 = current
- 3 = previous

1	1	1
1	2	0
1	0	0

0	3	1
0	2	1
1	1	1

0	1	0
3	2	1
0	0	0

Each of the above represents an example potential local grid. The center block is always the current. In the leftmost grid, we see a case where we might be in the starting block. The agent is blocked entirely on the west and north sides because those blocks are off the grid. Similarly in the middle grid, we see what might potentially be the agent in the goal block. It is blocked on the east and south sides. The rightmost grid is potentially what the agent would see somewhere traveling along. It knows where its previous is, as well as blocks in its neighbors.

For each local grid, we also keep track of the next move that our agent will make as separate information, using integers to denote the cardinal direction that the agent will move next:

- 1 = West
- 2 = South
- 3 = East
- 4 = North
- 0 = Stay in the same position (used when we hit a block or if we have reached the goal)

Question 2:
How are you defining your loss function when training your model?

We use cross-entropy loss when training our model.

Question 3:
In training, how many episodes on how many different gridworlds were necessary to get good performance of your model on the training data?

We used a 70:30 relation between our training and testing data sets. In order to train each of our models, we used 7000 unique solvable gridworlds. And for testing we used 3000 unique solvable gridworlds. Since 10,000 is still a relatively small size for a dataset, 70:30 left us with sufficient enough data to both train and test our models. Though if we had used an even larger dataset, 50,000 grids or even hundreds of thousands of grids, we likely would have started with an 80:20 ratio or higher split ratio.

Question 4:
How did you avoid overfitting? Since you want the ML agent to mimic the original agent, should you avoid overfitting?

Our first inclination in building the ML model to mimic exactly what the agent knows at any given moment. The agent would know the goal, would know not just the blocks in its neighbors or the sensed blocks in its neighbors, but all of the blocks that it had passed previously. However, this presented aforementioned difficulties with storage space, but also likely would've presented overfitting errors. The repeated forward agent and example inference agent make decisions (unlike the probability sensing agents) based on the information that they have at any given time, and specifically based on the new information gained at each time step. By removing features that were not necessary to making the decision of where to move next, we were able to avoid overfitting by only presenting the ML agent with the local grid and its previous position in order to determine where to move next.

We also shuffled and normalized all of our training data in order to avoid overfitting. Because our data was output in such a way from our original agents that it was in the exact order of the paths they followed, we worried that our agents would learn not to identify where blocks are located, but those specific path order movements and then not be able to follow a path where there were blocks in the way, or would try to move into blocked cells. Shuffling our data helped us to avoid this problem because it meant the agent was unlikely to see the same path over and over again.

Because we trained our model iteratively, we were also able to employ an early stopping method in order to avoid overfitting. Over time, our new iterations improved in accuracy, but at a certain point would weaken as it began to overfit the data. From here we were able to choose a stopping point that would maximize our accuracy without overfitting the data.

Question 6:
Do you think increasing the size or complexity of your model would offer any improvements? Why or why not?

We believe that increasing the amount of information in the state space would have improved the performance of our models. By limiting the data to just a local 3x3 grid, we cut out data such as the current coordinates of the model in the gridworld as well as more information from the gridworld in general. We believe that it would increase the accuracy and the performance of the models as they traverse the gridworld at the expense of training time.

In training our model, we fixed our *p* value for density of our gridworlds to 0.25. We picked 0.25 because, based upon our previous testing of the agents themselves, at 0.25, the mazes are sufficiently difficult but generally still solvable (between 65 and 70% of the mazes are solvable). However, this also means that the ML agent may not run as well on mazes generated with a higher density. Training the agent on a variety of densities, between 0.0 and 0.33, could offer improvements to the model by introducing additional scenarios. That said, whether this would offer improvements is speculative because the model is only trained on local data and only sees itself and its immediate neighbors, the model may not perform better because training on 0.25 at 7000 grids does introduce every possible configuration of blocks and next moves.

In terms of the structure of our neural network, at first we attempted to add on two to three convolution layers, but found out that the performance got worse. The reason may be that a CNN uses the structure in the data to deliver very high Power Per Parameter Per Input (PPPPI), which means that when you have a lot of features (like an image does), using a CNN, you can get comparable learning potential with far fewer parameters. But simply arranging our features into a matrix and input to a CNN, we would find it works poorly. Similarly, this is why a large amount of CNN applications are related to image recognition / computer vision. As most of the people use CNN Structure in an image is determined by assuming that there is more relevance to a particular pixel, in pixels nearby. Neighboring pixels share similar information. What we could do to improve this is to make a part of our input data – some of the grids in the 3x3 local grid to be partially relevant, which would be helpful for offering the convolution layer to learn its pattern.

Additionally, we tested several different batch and epoch sizes in training our models and found that increasing batch size and number of epochs only improved our model to a point.

Repeated Forward A*

Question 5:
How did you explore the architecture space, and test the different possibilities to find the best architecture?

In order to explore the architecture space, we tested a model with full dense layers and a model with two added convolutional layers, both with 10 epochs and a batch size of 40. During our training process we tested on a number of different epoch sizes and batch sizes, ranging from 5-20 epochs to 15-1000 in batch size. We found that testing at the higher epoch and batch sizes did not offer better accuracy over time and took significantly longer to process.

Our model with only full, dense layers had a best performance accuracy of about 80% and also converged to about 80%.

The model with 2 additional convolution layers had a best performance accuracy of 60% but converged to about 51%. We suspected that the model with the additional convolutional layers was overfitting over time, so we chose the epoch with the highest performance accuracy to carry out further testing.
The model with only full dense layers performed better than the model with convolutional layers added.

Question 7:
Does good performance on test data correlate with good performance in practice? Simulate the performance of your ML agent on new gridworlds to evaluate this.

No, good performance on test data does not correlate with good performance in practice.

Our Repeated Forward A* agents were not able to solve gridworlds. On a testing data set of 30 trials, each with 100 grid worlds of dimension 101, our Repeated Forward Fully Dense agent and Repeated Forward Convolutional agent were not able to solve any of the gridworlds.

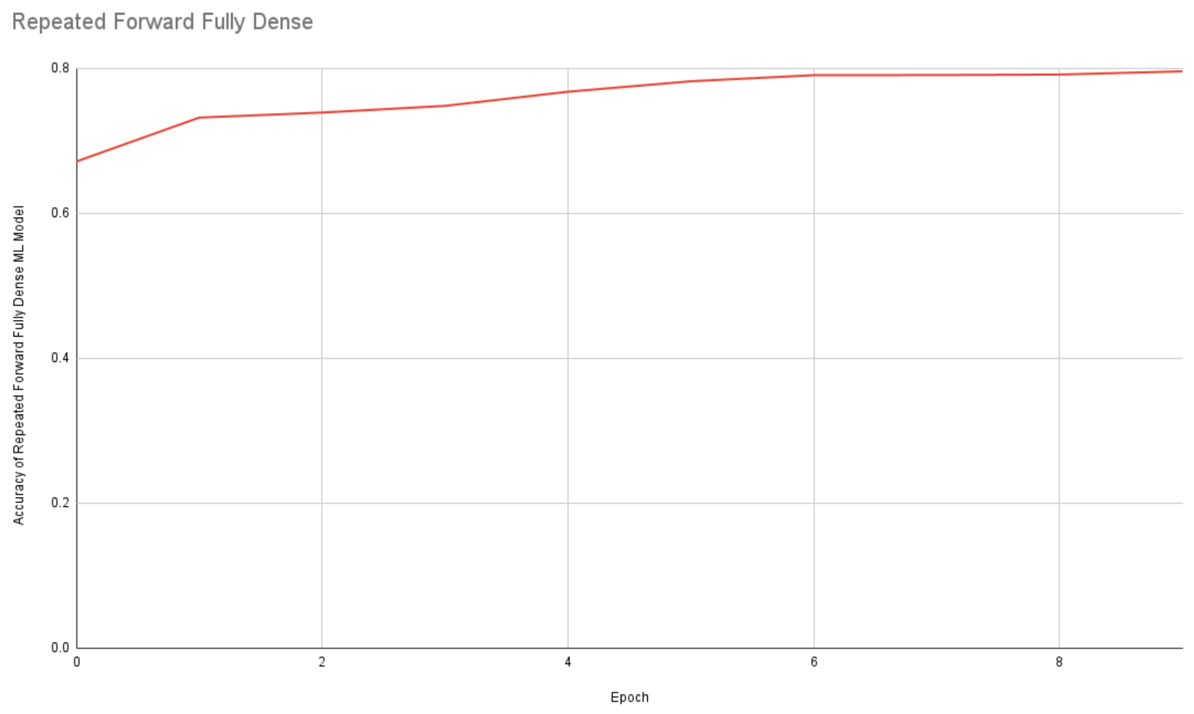
We found that our agents had a tendency to get stuck in infinite loops. That is, at every possible position in the gridworld, once it had made its decision on a final direction to move in, it would never make a different decision, should it come to that block again. Ub order to identify an infinite loop, we kept track of a number of the previous cells we had visited. Should a cell appear several times in our previous cells, we knew that we had gotten stuck and terminated the agent. There were also cases in which the agent might try to move into a known blocked cell, in this instance, we terminated the agent because it made an illegal move into a known block.

The average time taken for our Repeated Forward A* Fully Dense agent to terminate was 0.114 seconds, and the average number of cells it traversed prior to termination was 60. The average number of time taken for our Repeated Forward A* Convolutional Agent to terminate was 0.108 seconds and on average traversed 161 cells prior to termination.

Question 8:
For your best model structure, for each architecture, plot a) performance on test data as a function of training rounds, and b) average performance in practice on new gridworlds. How do your ML agents stack up against the original agents? Do either ML agents offer an advantage in terms of training time?

Note that for (7) and (8) above, you'll need to be able to handle that the ML agent may not ever successfully reach the target - how should you fairly include this data? Don't just discard it.

Evaluation Accuracy Repeated Forward Fully Dense Model

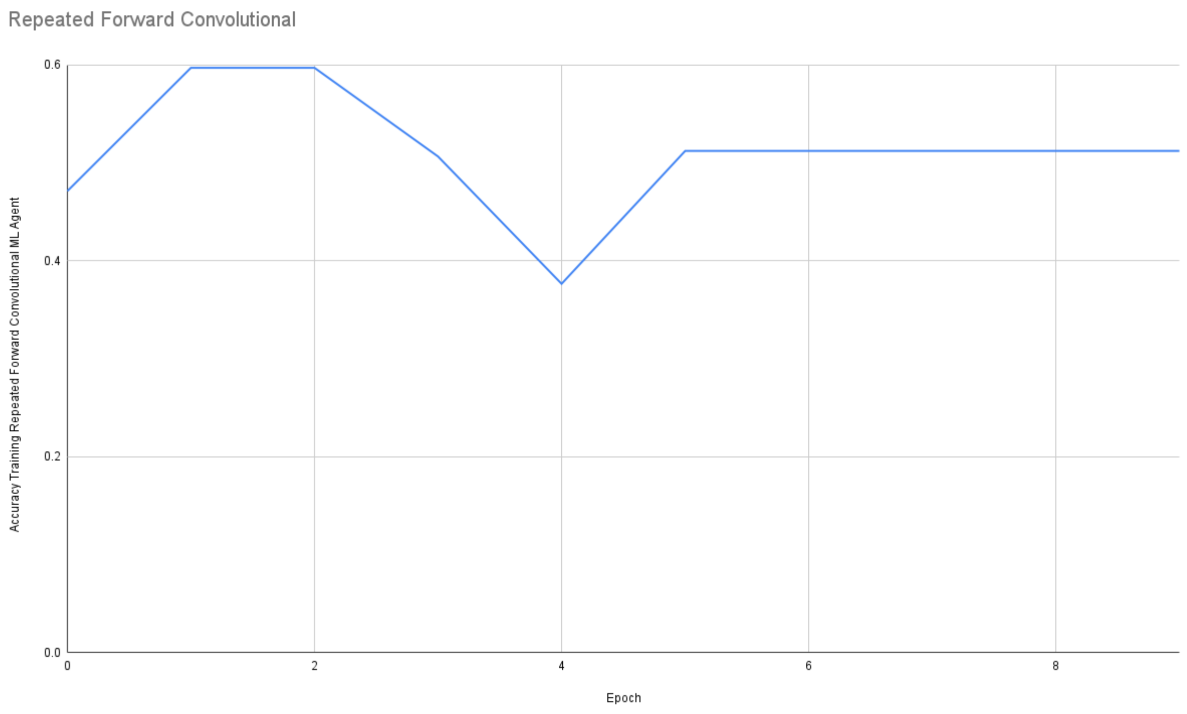


Epoch	Accuracy - Repeated Forward Fully Dense
0	0.6713058857
1	0.7319552694
2	0.7389221185

3	0.7481668034
4	0.7676888188
5	0.7821318217
6	0.7905213156
7	0.7907348627
8	0.7914065837
9	0.7958599933

Training Time: 106.0m 49.5721039772s

Evaluation Accuracy Repeated Forward Convolutional Model



Epoch	Accuracy - Repeated Forward Convolutional
0	0.4710247654
1	0.5968922379
2	0.5968922379
3	0.5063221975
4	0.3763381784
5	0.5119816971
6	0.5119987408

7	0.5119816971
8	0.5119987408
9	0.5119816971

Training Time: 725.0m 26.369863510131836s

There was a significant difference between training the fully dense and convolutional models for Repeated Forward A* ML agents. To train nine epochs, the fully dense model took about 106 minutes and 49 seconds to complete, while the convolutional model took about 725 minutes and 26 seconds to complete. It is worth noting that the drastic change in training times could also be due to training each of the models on separate computers.

Our original Repeated Forward A* agent, tested on 3000 mazes, was able to solve gridworlds in an average time of 1.704 seconds, visiting 194794 nodes and with an average trajectory length of 302 nodes from start to the goal, on average. By comparison, our ML agents are not able to solve gridworlds.

Example Inference Agent

Question 5:
How did you explore the architecture space, and test the different possibilities to find the best architecture?

Similar to the Repeated Forward Agent, we tested our Example Inference Agent as both a model with full dense layers only and a model with full dense layers + two additional convolutional layers. The model with full dense layers had a best accuracy performance of 94% and also converged to this number.

The model with two additional convolutional layers, had a best performance accuracy of about 76% and converged to a 76% accuracy.

Similar to the Repeated Forward A* ML agent, the ML agent with only full dense layers performed significantly better than the agent with convolutional layers added. We suspect this is because the agent with the full dense layers is making decisions more similarly to how the original agents actually work and is able to form the output based on every input, and understand the local state space better than the convolutional layer agent. The agent with convolutional layers on the other hand, only uses a subset of the input, and then applies it across the entirety of the training data. Because of this, the agent with convolutional layers may not capture all of the situations in which the agent could possibly find itself and is therefore not always able to accurately predict the next move.

Question 7:

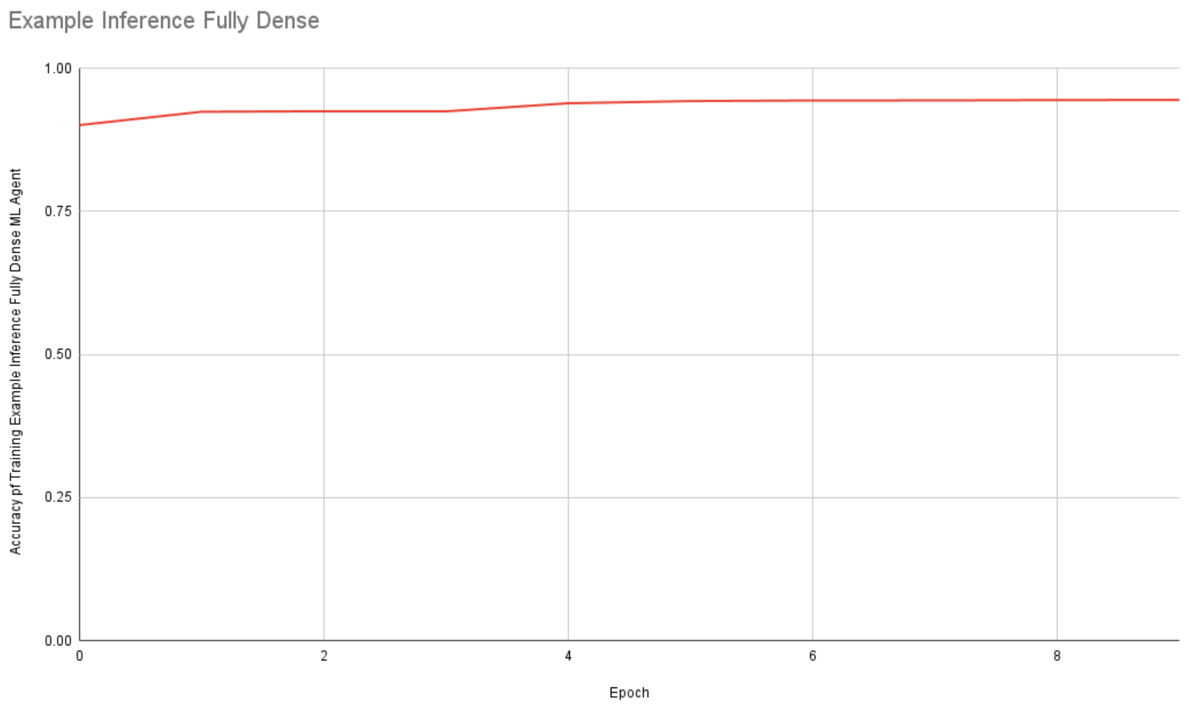
Does good performance on test data correlate with good performance in practice? Simulate the performance of your ML agent on new gridworlds to evaluate this.

We saw similar results to our Repeated Forward A* ML agents with our Example Inference ML agents. Out of 3000 possible solvable gridworlds and dimension 101, our Example Inference ML agents were not able to solve any gridworlds. Again, in closer testing, we saw our agents get stuck in infinite loops and not be able to get themselves out.

The Example Inference Fully Dense agent ran for approximately 0.15 seconds before terminating, with an average number of 75 cells traversed. And the Example Inference Convolutional agent ran for approximately 0.115 seconds and traversed approximately 170 cells prior to terminating.

Question 8:
For your best model structure, for each architecture, plot a) performance on test data as a function of training rounds, and b) average performance in practice on new gridworlds. How do your ML agents stack up against the original agents? Do either ML agents offer an advantage in terms of training time? Note that for (7) and (8) above, you'll need to be able to handle that the ML agent may not ever successfully reach the target - how should you fairly include this data? Don't just discard it.

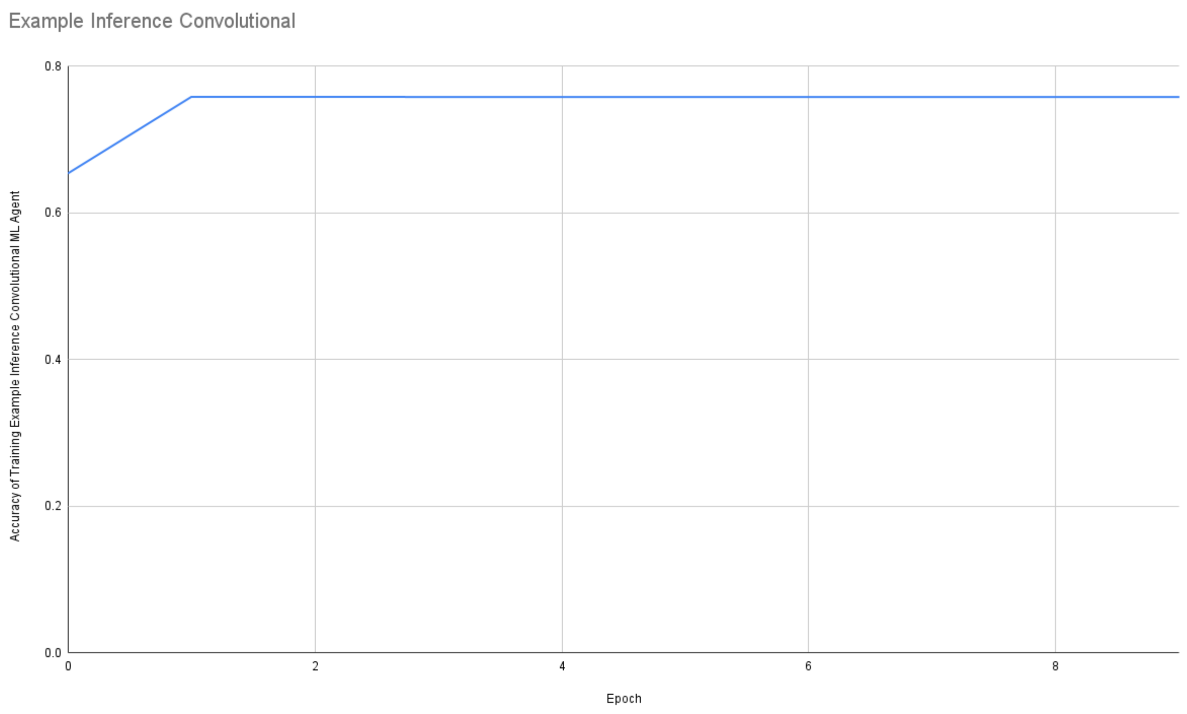
Evaluation Accuracy Example Inference Fully Dense Model



Epoch	Accuracy - Example Inference Fully Dense
0	0.9009129977
1	0.9242455346
2	0.925137331
3	0.924962337
4	0.9390774001
5	0.942898103
6	0.94396265
7	0.9440389294
8	0.9447254445
9	0.9448376201

Training Time: 113.0m 51.1506860256s

Evaluation Accuracy Example Inference Convolutional Model



Epoch	Accuracy - Example Inference Convolutional
0	0.6535600628
1	0.757951852

2	0.757951852
3	0.757790319
4	0.757790319
5	0.757790319
6	0.757790319
7	0.757790319
8	0.757790319
9	0.757790319

Training Time: 337.0m 38.37278890609741s

Similar to the training times of Repeated Forward A*, there was a significant difference in the training times of the Convolutional models for mimicking the Example Inference Agent. However this was a less dramatic change. Training the fully dense model took roughly 113 minutes and 51 seconds, very similar to the training time for the fully dense Repeated Forward A* model, while the training of the convolutional model took about 337 minutes and 38 seconds, cutting the time from the Repeated Forward A* convolutional model in about half.

The ML agents do not offer any advantage over the agents because they are not able to solve the gridworlds. Our original Example Inference Agent, tested on 3000 mazes, was able to solve gridworlds in an average of 2.226 seconds, on average, visiting 256701 nodes and with a final trajectory length of 326 nodes.

Given the way that our ML agent is trained on local data, there seems to be no way to simulate the vision properties of the non-ML agents (both Repeated Forward and Example Inference). The non-ML agents, unlike the ML agents, do not make decisions at every cell along their path. The blocks that they sense as they move forward may affect their local knowledge and may be saved for if a path needs to be redrawn, but until the agent finds a block directly in front of them in their predetermined path, the existing path remains unchanged. This, in contrast, with the ML agents, who must at each cell, determine the next direction to move in.

If the ML agents had been able to see the entire gridworld, rather than just the local data, it seems likely that there is a way of training and input data formatting that does get them a path from the start to the goal. It also seems feasible that this could be done in fewer cells traversed than the non-ML agents.

Appendix

```
### NN CODE ###
import os
import torch
from torch import float32, nn
import torch.nn.functional as F
```

```
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor
import numpy as np
import pandas as pd

# data

# transfer string data to np array
def to_nparray(data, idx, dim):
    temp_list = list(data[idx]) # items in this list are str type
    data_list = []
    for item in temp_list:
        try:
            data_list.append(float(item))
        except:
            pass
    data_array = np.array(data_list).reshape((dim, dim))
    data_array[1][1] = 0.0
    norm = np.linalg.norm(data_array)
    normal_array = data_array/norm
    return normal_array

def min_max_normalize(data):
    for column in data.columns:
            max = data[column].max()
            min = data[column].min()
            if max != min:
                data[column] = (data[column] - min) / (max - min)
            elif max == min != 0:
                data[column] = data[column] / max
            else:
                pass
    return data

class RepeatedForwardDataset(Dataset):
    #TODO: how to determine data's state -> train/val/test?
    def __init__(self, csv_file, dim):
        super(RepeatedForwardDataset, self).__init__()
        self.dataset = pd.read_csv(csv_file)
        self.gridworld = self.dataset["local_grid"]
        self.dim = dim
        #print(self.dataset)
    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):

        gridworld = to_nparray(self.gridworld, idx, self.dim)
        gridworld = torch.tensor(gridworld)
        label = self.dataset["next_dir"][idx]
        # label equals to next move (agent's action)

        return gridworld, label

class ConvBlock(nn.Module):
    def __init__(self, in_size, out_size):
        super(ConvBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_size, out_size, kernel_size=(1, 1), bias=False),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(out_size),
            nn.MaxPool2d(kernel_size=(2, 2)),
            nn.Dropout2d(p=0.5)
        )
    def forward(self, x):
        x = self.block(x)
        return x

class NN_repeatedForward(nn.Module):
    def __init__(self, num_classes = 5):
        """
        Args:
            num_classes: number of classes
        """
        super(NN_repeatedForward, self).__init__()
        #self.flatten = nn.Flatten()

        # pass gridworld here
        # gridworld only has one channel
        self.conv1 = ConvBlock(1, 8)
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout2d(p=0.25)
        self.conv2 = ConvBlock(8, 32)
        self.fc1 = nn.Linear(32, 64)
        self.fc2 = nn.Linear(64, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.relu(self.fc3(x))

        x = self.fc4(x)
        return x

import os
import torch
from torch import float32, nn
import torch.nn.functional as F

from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor
import numpy as np
import pandas as pd

# data

# transfer string data to np array
def to_nparray(data, idx, dim):
    temp_list = list(data[idx]) # items in this list are str type
    data_list = []
    for item in temp_list:
        try:
            data_list.append(float(item))
        except:
            pass
    data_array = np.array(data_list).reshape((dim, dim))
    data_array[1][1] = 0.0
    norm = np.linalg.norm(data_array)
    normal_array = data_array/norm
    return normal_array

def min_max_normalize(data):
    for column in data.columns:
            max = data[column].max()
            min = data[column].min()
            if max != min:
                data[column] = (data[column] - min) / (max - min)
            elif max == min != 0:
                data[column] = data[column] / max
            else:
                pass
    return data

class RepeatedForwardDataset(Dataset):
    #TODO: how to determine data's state -> train/val/test?
    def __init__(self, csv_file, dim):
        super(RepeatedForwardDataset, self).__init__()
        self.dataset = pd.read_csv(csv_file)
        self.gridworld = self.dataset["local_grid"]
        self.dim = dim
        #print(self.dataset)
    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):

        gridworld = to_nparray(self.gridworld, idx, self.dim)
        gridworld = torch.tensor(gridworld)
        label = self.dataset["next_dir"][idx]
        # label equals to next move (agent's action)

        return gridworld, label
"""
class ConvBlock(nn.Module):
    def __init__(self, in_size, out_size):
        super(ConvBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_size, out_size, kernel_size=(1, 1), bias=False),
            nn.ReLU(),
            nn.BatchNorm2d(out_size),
            nn.MaxPool2d(kernel_size=(2, 2)),
            nn.Dropout2d(p=0.5)
        )
    def forward(self, x):
        x = self.block(x)
        return x
"""

class NN_repeatedForward(nn.Module):
    def __init__(self, num_classes = 5):
        """
        Args:
            num_classes: number of classes
        """
        super(NN_repeatedForward, self).__init__()
        #self.flatten = nn.Flatten()
        # pass gridworld here
        # gridworld only has one channel
        #self.conv1 = ConvBlock(1, 8)
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout2d(p=0.25)
        #self.conv2 = ConvBlock(8, 32)
        self.fc1 = nn.Linear(8, 16)
        self.fc2 = nn.Linear(16, 64)
        self.fc3 = nn.Linear(64, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, num_classes)

    def forward(self, x):
        #x = self.conv1(x)
        #x = self.conv2(x)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.relu(self.fc3(x))
        x = self.dropout(x)
        x = F.relu(self.fc4(x))

        x = self.fc5(x)
        return x

import sys
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import os
import errno
from torch.utils.data import Dataset, DataLoader, dataloader
from torchvision import transforms, utils, models, datasets
import pandas as pd
import time
import torch.backends.cudnn as cudnn

import os.path as osp
from NN_repeatedForward_justgrid_dense import NN_repeatedForward, RepeatedForwardDataset

### device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'Using {device} device')

### TRAINING CODE ###
import argparse
parser = argparse.ArgumentParser("Repeated Forward")
parser.add_argument('--seed', type=int, default=5)
parser.add_argument('--gpu', type=str, default='0')
parser.add_argument('--use-cpu', action='store_true')
parser.add_argument('--num_classes', type=int, default=5)
parser.add_argument('--dim', type=int, default=3)
parser.add_argument('--batch_size', type=int, default=40)
args = parser.parse_args()
```

```
csv_pth = "repeatedforward_data_101_nextdir"

def split_train_val(csv_filename):
    data = pd.read_csv(csv_filename + ".csv")
    data = data.sample(frac=1).reset_index(drop=True)
    n = int(len(data)*0.7)
    train_data = data.iloc[0:n-1, ]
    print(n)
    print(train_data)
    val_data = data.iloc[n:, ]
    train_data.to_csv(csv_filename + "_train.csv")
    val_data.to_csv(csv_filename + "_val.csv")

### split data
split_train_val(csv_pth)

### load data
#TODO: does gridworld data needs to be transformed?
repeated_dataset = {x: RepeatedForwardDataset(csv_pth+"_{}.csv".format(x), dim=args.dim)for x
in ['train', 'val']}
dataloaders = {x: DataLoader(repeated_dataset[x], batch_size=args.batch_size, shuffle=True)
for x in ['train', 'val']}
dataset_sizes = {x: len(repeated_dataset[x]) for x in ['train', 'val']}

"""
### initialize NN model
def NNAgent_repeatedForward(num_classes, dim):
    model = NN_repeatedForward(num_classes, dim)
    num_features = model.last_linear.in_features
    model.last_linear = COCOLoss(num_classes, num_features)
    # use COCOLoss to optimize performance
    return model
"""

"""
refer to Pytorch tutorial:
    https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
"""
def mkdir_if_missing(directory):
    if not osp.exists(directory):
        try:
            os.makedirs(directory)
        except OSError as e:
            if e.errno != errno.EEXIST:
                raise
class Logger(object):
    """
    Write console output to external text file.

    Code imported from https://github.com/Cysu/open-reid/blob/master/reid/utils/logging.py.
    """
    def __init__(self, fpath=None):
        self.console = sys.stdout
        self.file = None
        if fpath is not None:
            mkdir_if_missing(os.path.dirname(fpath))
            self.file = open(fpath, 'w')

    def __del__(self):
        self.close()

    def __enter__(self):
            pass

    def __exit__(self, *args):
        self.close()

    def write(self, msg):
        self.console.write(msg)
        if self.file is not None:
            self.file.write(msg)

    def flush(self):
        self.console.flush()
        if self.file is not None:
            self.file.flush()
            os.fsync(self.file.fileno())

    def close(self):
        self.console.close()
        if self.file is not None:
            self.file.close()

def train(model, criterion, optimizer, num_epochs = 5):

    start = time.time()
    best_acc = 0.0

    #for param in model.parameters():
    #    param.requires_grad = True

    for epoch in range(num_epochs):
        print("Epoch {}/{}".format(epoch, num_epochs-1))
        print("-"*10)

        for state in ['train', 'val']:
            if state == 'train':
                # tell that model is training
                model.train()
            else:
                # tell that model is doint validation
                model.eval()
            total_loss = 0.0
            corrects = 0

            for idx, input_data in enumerate(dataloaders[state]):
                gridworld, labels = input_data
                gridworld = torch.unsqueeze(gridworld, 1)

                if torch.cuda.is_available():
                    gridworld, labels = gridworld.to(device, dtype=torch.float),
labels.to(device, dtype=torch.long)
                    model.to(device)

                # sets gradient calculation to ON when state == train
                with torch.set_grad_enabled(state == 'train'):
                    # set zero to the parameter gradients
                    optimizer.zero_grad()
                    logits = model(gridworld)
                    loss = criterion(logits, labels)
                    preds = torch.argmax(logits, 1)
                if state == 'train':
                    loss.backward()
                    optimizer.step()
                    #scheduler.step()

                print("#{} batch prediction:{}".format(idx, preds))
                print("#{} batch label:{}".format(idx, labels.data))
                total_loss += loss.item()
                corrects += torch.sum(preds==labels.data)
                print('#{} batch loss: {}'.format(idx, loss.item()),
                      '#{} batch corrects: {}'.format(idx, torch.sum(preds == labels.data)),
                      '#{} batch accuracy: {}%'.format(idx, torch.sum(preds ==
labels.data)*100. /args.batch_size))

        epoch_loss = total_loss/dataset_sizes[state]
        epoch_acc = corrects.double() / dataset_sizes[state]

        print('{} Loss: {} Acc: {}'.format(state, epoch_loss, epoch_acc))

        if state == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
        temp_model_path = "./repeatedForwardAgent_{}.pth".format(epoch)
        # save models generated in each epoch
        torch.save(model.state_dict(), temp_model_path)

    print("-" * 10)
    time_elapsed = time.time() - start
    print("training time: {}m {}s".format(time_elapsed//60, time_elapsed%60))
    print("best validation accuracy: {}".format(best_acc))

    return model

def main():
    #torch.manual_seed(args.seed)
    os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu
    if torch.cuda.is_available():
        use_gpu = True
    else:
        use_gpu = False

    if use_gpu:
        print("Currently using GPU")
        cudnn.benchmark = True
        #torch.cuda.manual_seed_all(args.seed)
    else:
        print("Currently using CPU")

    model = NN_repeatedForward(num_classes=args.num_classes)
    model.cuda()
    criterion = nn.CrossEntropyLoss()
    sys.stdout = Logger(osp.join("./log_train_repeatedForward.txt"))

    # optimize
    params_to_update = model.parameters()
    optimizer = optim.SGD(params_to_update, lr=0.0001, momentum=0.9)

    #exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

    model_train = train(model=model,
                        criterion=criterion,
                        optimizer=optimizer,
                        num_epochs=10)
    final_model_path = "./Final_repeatedForwardAgent.pth"
    torch.save(model_train.state_dict(), final_model_path)
```

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

import sys
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import os
import errno
from torch.utils.data import Dataset, DataLoader, dataloader
from torchvision import transforms, utils, models, datasets
import pandas as pd
import time
import torch.backends.cudnn as cudnn

import os.path as osp
from NN_repeatedForward_justgrid import NN_repeatedForward, RepeatedForwardDataset

### device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'Using {device} device')

import argparse
parser = argparse.ArgumentParser("Repeated Forward")
parser.add_argument('--seed', type=int, default=5)
parser.add_argument('--gpu', type=str, default='0')
parser.add_argument('--use-cpu', action='store_true')
parser.add_argument('--num_classes', type=int, default=5)
parser.add_argument('--dim', type=int, default=3)
parser.add_argument('--batch_size', type=int, default=50)
args = parser.parse_args()

csv_pth = "C:/Users/Ruby/Desktop/520_project4/repeatedforward_data_101_nextdir"

def split_train_val(csv_filename):
    data = pd.read_csv(csv_filename + ".csv")
    data = data.sample(frac=1).reset_index(drop=True)
    n = int(len(data)*0.7)
    train_data = data.iloc[0:n-1, ]
    print(n)
    print(train_data)
    val_data = data.iloc[n:, ]
    train_data.to_csv(csv_filename + "_train.csv")
    val_data.to_csv(csv_filename + "_val.csv")

### split data
split_train_val(csv_pth)

### load data
#TODO: does gridworld data needs to be transformed?
repeated_dataset = {x: RepeatedForwardDataset(csv_pth+"_{}.csv".format(x), dim=args.dim)for x
in ['train', 'val']}
dataloaders = {x: DataLoader(repeated_dataset[x], batch_size=args.batch_size, shuffle=True)
for x in ['train', 'val']}
dataset_sizes = {x: len(repeated_dataset[x]) for x in ['train', 'val']}

"""
### initialize NN model
```

```
def NNAgent_repeatedForward(num_classes, dim):
    model = NN_repeatedForward(num_classes, dim)
    num_features = model.last_linear.in_features
    model.last_linear = COCOLoss(num_classes, num_features)
    # use COCOLoss to optimize performance
    return model
"""

"""
refer to Pytorch tutorial:
    https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
"""
def mkdir_if_missing(directory):
    if not osp.exists(directory):
        try:
            os.makedirs(directory)
        except OSError as e:
            if e.errno != errno.EEXIST:
                raise
class Logger(object):
    """
    Write console output to external text file.

    Code imported from https://github.com/Cysu/open-reid/blob/master/reid/utils/logging.py.
    """
    def __init__(self, fpath=None):
        self.console = sys.stdout
        self.file = None
        if fpath is not None:
            mkdir_if_missing(os.path.dirname(fpath))
            self.file = open(fpath, 'w')

    def __del__(self):
        self.close()

    def __enter__(self):
        pass

    def __exit__(self, *args):
        self.close()

    def write(self, msg):
        self.console.write(msg)
        if self.file is not None:
            self.file.write(msg)

    def flush(self):
        self.console.flush()
        if self.file is not None:
            self.file.flush()
            os.fsync(self.file.fileno())

    def close(self):
        self.console.close()
        if self.file is not None:
            self.file.close()

def train(model, criterion, optimizer, num_epochs = 5):

    start = time.time()
    best_acc = 0.0
    #for param in model.parameters():
    #    param.requires_grad = True

    for epoch in range(num_epochs):
        print("Epoch {}/{}".format(epoch, num_epochs-1))
        print("-"*10)

        for state in ['train', 'val']:
            if state == 'train':
                # tell that model is training
                model.train()
            else:
                # tell that model is doint validation
                model.eval()
            total_loss = 0.0
            corrects = 0

            for idx, input_data in enumerate(dataloaders[state]):
                gridworld, labels = input_data
                gridworld = torch.unsqueeze(gridworld, 1)

                if torch.cuda.is_available():
                    gridworld, labels = gridworld.to(device, dtype=torch.float),
labels.to(device, dtype=torch.long)
                    model.to(device)

                # sets gradient calculation to ON when state == train
                with torch.set_grad_enabled(state == 'train'):
                    # set zero to the parameter gradients
                    optimizer.zero_grad()
                    logits = model(gridworld)
                    loss = criterion(logits, labels)
                    preds = torch.argmax(logits, 1)
                if state == 'train':
                    loss.backward()
                    optimizer.step()
                    #scheduler.step()

                print("#{} batch prediction:{}".format(idx, preds))
                print("#{} batch label:{}".format(idx, labels.data))
                total_loss += loss.item()
                corrects += torch.sum(preds==labels.data)
                print('#{} batch loss: {}'.format(idx, loss.item()),
                      '#{} batch corrects: {}'.format(idx, torch.sum(preds == labels.data)),
                      '#{} batch accuracy: {}%'.format(idx, torch.sum(preds ==
labels.data)*100. /args.batch_size))

        epoch_loss = total_loss/dataset_sizes[state]
        epoch_acc = corrects.double() / dataset_sizes[state]

        print('{} Loss: {} Acc: {}'.format(state, epoch_loss, epoch_acc))

        if state == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
        temp_model_path = "./repeatedForwardAgent_{}.pth".format(epoch)
        # save models generated in each epoch
        torch.save(model.state_dict(), temp_model_path)

    print("-" * 10)
    time_elapsed = time.time() - start
    print("training time: {}m {}s".format(time_elapsed//60, time_elapsed%60))
    print("best validation accuracy: {}".format(best_acc))
    return model

def main():
    #torch.manual_seed(args.seed)
    os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu
    if torch.cuda.is_available():
        use_gpu = True
    else:
        use_gpu = False

    if use_gpu:
        print("Currently using GPU")
        cudnn.benchmark = True
        #torch.cuda.manual_seed_all(args.seed)
    else:
        print("Currently using CPU")

    model = NN_repeatedForward(num_classes=args.num_classes)
    model.cuda()
    criterion = nn.CrossEntropyLoss()
    sys.stdout = Logger(osp.join("./log_train_repeatedForward.txt"))

    # optimize
    params_to_update = model.parameters()
    optimizer = optim.SGD(params_to_update, lr=0.0001, momentum=0.9)

    #exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

    model_train = train(model=model,
                        criterion=criterion,
                        optimizer=optimizer,
                        num_epochs=10)
    final_model_path = "./Final_repeatedForwardAgent.pth"
    torch.save(model_train.state_dict(), final_model_path)

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