model_v1

October 15, 2023

[1]:
```
import torch
from torch.utils.data import IterableDataset, DataLoader
import numpy as np
from third_party.absl_py import app
from third_party.absl_py import flags
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import random_split
import random
import file.recordio as recordio

import projects.sc_eval.bw.bw_pb2 as bw_pb2
```

[2]:
```
def one_hot_race(race, num_classes):
    return [1 if i == race else 0 for i in range(num_classes)]


class BwExampleDataset(IterableDataset):
    def __init__(self, path, after_sec=0):
        self.path = path
        self.reader = recordio.RecordReader(path)
        self.after_sec = after_sec

    def __iter__(self):
        return self

    def __next__(self):
        example = None
        while True:
            message = self.reader.ReadMessage()
            if message is None:
                raise StopIteration()
            example = bw_pb2.BwExampleV1.FromString(message)
            if example.seconds >= self.after_sec:
                break
```

```
      p0_actions_global = torch.tensor(example.p0_actions_global, dtype=torch.
↪float32)
      p0_actions_recent = torch.tensor(example.p0_actions_recent, dtype=torch.
↪float32)
      p1_actions_global = torch.tensor(example.p1_actions_global, dtype=torch.
↪float32)
      p1_actions_recent = torch.tensor(example.p1_actions_recent, dtype=torch.
↪float32)

      counts = torch.cat(
          [
              p0_actions_global,
              p0_actions_recent,
              p1_actions_global,
              p1_actions_recent,
          ]
      )

      normalized_counts = torch.log(counts + 1)

      p0_race_encoded = torch.tensor(
          one_hot_race(example.p0_race, 4), dtype=torch.float32
      )
      p1_race_encoded = torch.tensor(
          one_hot_race(example.p1_race, 4), dtype=torch.float32
      )

      label = torch.tensor([1, 0] if example.p0_wins else [0, 1], dtype=torch.
↪float32)

      features = torch.cat(
          [
              normalized_counts,
              p0_race_encoded,
              p1_race_encoded,
          ]
      )

      return features, label


  def Restart(self):
      self.reader.Close()
      self.reader = recordio.RecordReader(self.path)
```

[3]:
```
# Hyperparameters
batch_size = 512
learning_rate = 0.001
epochs = 3
```

[4]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[5]:
```
train_data = BwExampleDataset("/jfs/sc_eval/bw/train.recordio")
test_data = BwExampleDataset("/jfs/sc_eval/bw/test.recordio")

train_loader = DataLoader(train_data, batch_size=batch_size)
test_loader = DataLoader(test_data, batch_size=batch_size)
```

[6]:
```
class SimpleClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleClassifier, self).__init__()

        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return nn.functional.softmax(x, dim=1)  # Apply softmax along dimension

ex = next(train_data)
input_size = ex[0].shape[0]
output_size = ex[1].shape[0]
# Common advice is that the hidden layer size should be the mean of the input +␣
 ↪output.
hidden_size = (input_size + output_size) // 2
model = SimpleClassifier(input_size=input_size, hidden_size=hidden_size,␣
 ↪output_size=output_size).to(device)
print(f"input={input_size} hidden={hidden_size} output={output_size}")
```

```
input=128 hidden=65 output=2
```

[7]:
```
for fea, lab in test_loader:
    print(fea.shape)
    break
```

```
torch.Size([512, 128])
```

[8]:
```
def RunModelOnTestData(loader=test_loader, num_batches=-1):
    num_correct = num_total = 0
```

```
    for features, labels in loader:
        features, labels = features.to(device), labels.to(device)  # TODO:␣
 ↪Maybe GPU is excessive/slower here?
        predictions = model(features).squeeze()
        _, indices = predictions.max(dim=1)
        actual_indices = torch.max(labels, 1)[1]
        num_correct += (indices == actual_indices).sum().item()
        num_total += batch_size

        if num_batches != -1:
            num_batches -= 1
        if num_batches == 0:
            break
    print(f"Test data got {num_correct} / {num_total} correct, or {num_correct/
 ↪num_total}")
```

[9]:
```
# Run a pass through the test dataset before training to confirm there's around␣
 ↪a 50% chance of getting it right.
RunModelOnTestData(num_batches=100)

# Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training Loop
for epoch in range(epochs):
    model.train()
    total_loss = 0
    i = 0

    # Decrease learning rate for last epoch.
    if epoch == epochs - 1:
        learning_rate *= 0.1

    for features, labels in train_loader:
        i += 1
        features, labels = features.to(device), labels.to(device)
        optimizer.zero_grad()

        # Forward pass.
        outputs = model(features)
        loss = criterion(outputs, torch.max(labels, 1)[1])
        total_loss += loss.item()

        # Backward pass and optimize.
        loss.backward()
        optimizer.step()
```

```
    train_data.Restart()
    test_data.Restart()

    RunModelOnTestData(num_batches=100)

RunModelOnTestData()
```

```
Test data got 24317 / 51200 correct, or 0.47494140625
Test data got 30328 / 51200 correct, or 0.59234375
Test data got 30293 / 51200 correct, or 0.59166015625
Test data got 30321 / 51200 correct, or 0.59220703125
Test data got 504434 / 887296 correct, or 0.5685070145701097
```

[13]:
```
test_data.Restart()
after_5_data = BwExampleDataset("/jfs/sc_eval/bw/test.recordio",␣
 ↪after_sec=60*20)
after_5_loader = DataLoader(test_data, batch_size=batch_size)
RunModelOnTestData(loader=after_5_loader)
```

```
Test data got 534755 / 938496 correct, or 0.5697999778368794
```

0.1 TODO:

Make a graph that looks at single replays and inspect games.