hw5_code

April 8, 2024

[ ]:
```
import sys
import time

import numpy as np
import matplotlib
import torch

%matplotlib inline
print(f"Python version: {sys.version}\nNumpy version: {np.
↪__version__}\nMatplotlib version: {matplotlib.__version__}\nPyTorch version:␣
↪{torch.__version__}")
```

```
Python version: 3.11.8 | packaged by conda-forge | (main, Feb 16 2024, 20:49:36)
[Clang 16.0.6 ]
Numpy version: 1.26.4
Matplotlib version: 3.8.0
PyTorch version: 2.2.1
```

1 Problem 1

1.1 Setup

[ ]:
```
def sigma(x):
    return torch.sigmoid(x)
def sigma_prime(x):
    return sigma(x)*(1-sigma(x))


torch.manual_seed(0)
L = 6
X_data = torch.rand(4, 1)
Y_data = torch.rand(1, 1)

A_list,b_list = [],[]
for _ in range(L-1):
    A_list.append(torch.rand(4, 4))
    b_list.append(torch.rand(4, 1))
A_list.append(torch.rand(1, 4))
```

```
b_list.append(torch.rand(1, 1))
```

1.2 Autograd

1.2.1 Option 1: directly use PyTorch's autograd feature

[ ]:
```
for A in A_list:
    A.requires_grad = True
for b in b_list:
    b.requires_grad = True

y = X_data
for ell in range(L):
    S = sigma if ell<L-1 else lambda x: x
    y = S(A_list[ell]@y+b_list[ell])

# backward pass in pytorch
loss = torch.square(y-Y_data)/2
loss.backward()

print(A_list[0].grad)
```

```
tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
        [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
        [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
        [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]])
```

1.2.2 Option 2: construct a NN model and use backprop

[ ]:
```
from torch import nn

class MLP(nn.Module) :
    def __init__(self) :
        super().__init__()
        self.linear = nn.ModuleList([nn.Linear(4,4) for _ in range(L-1)])
        self.linear.append(nn.Linear(4,1))
        for ell in range(L):
            self.linear[ell].weight.data = A_list[ell]
            self.linear[ell].bias.data = b_list[ell].squeeze()

    def forward(self, x) :
        x = x.squeeze()
        for ell in range(L-1):
            x = sigma(self.linear[ell](x))
        x = self.linear[-1](x)
        return x

model = MLP()
```

```
loss = torch.square(model(X_data)-Y_data)/2
loss.backward()

print(model.linear[0].weight.grad)
```

```
tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
        [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
        [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
        [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]])
```

1.2.3 Option 3: implement backprop yourself

[ ]:
```
# forward pass
y_list = [X_data]
y = X_data
for ell in range(L):
    S = sigma if ell<L-1 else lambda x: x
    y = S(A_list[ell]@y+b_list[ell])
    y_list.append(y)

# backward pass
dA_list = []
db_list = []
dy = y-Y_data  # dloss/dy_L
for ell in reversed(range(L)):  # ell = L-1, L-2, ..., 1, 0
    S = sigma_prime if ell<L-1 else lambda x: torch.ones(x.shape)
    A, b, y = A_list[ell], b_list[ell], y_list[ell]
    # NB: y=y_{ell-1} (while A=A_ell and b=b_ell) since len(y_list)==L+1 and␣
 ↪y_0=x_data
    # hence first y value is y_{L-1} alongside A_L and b_L (even though these␣
 ↪are all indexed with ell=L-1)

    S_diagonal = torch.diag(S(A @ y + b).reshape(-1))

    db = dy @ S_diagonal          # dloss/db_ell
    dA = S_diagonal @ dy.T @ y.T  # dloss/dA_ell
    dy = dy @ S_diagonal @ A      # dloss/dy_{ell-1}

    dA_list.insert(0, dA)
    db_list.insert(0, db)

print(dA_list[0])
```

```
tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
        [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
        [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
        [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]],
```

```
        grad_fn=<MmBackward0>)
```

1.3 Compare results

[ ]:
```
A_list[0].grad, model.linear[0].weight.grad, dA_list[0]
```

[ ]:
```
(tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
         [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
         [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
         [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]]),
 tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
         [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
         [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
         [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]]),
 tensor([[2.3943e-05, 3.7064e-05, 4.2687e-06, 6.3700e-06],
         [3.4104e-05, 5.2794e-05, 6.0804e-06, 9.0735e-06],
         [2.4438e-05, 3.7831e-05, 4.3571e-06, 6.5019e-06],
         [2.0187e-05, 3.1250e-05, 3.5991e-06, 5.3707e-06]],
        grad_fn=<MmBackward0>))
```

2 Problem 6

2.1 Data prep

[ ]:
```
import torch
import torch.nn as nn
from torchvision import datasets
from torchvision.transforms import transforms
from torch.utils.data import DataLoader

# Make sure to use only 10% of the available MNIST data.
# Otherwise, experiment will take quite long (around 90 minutes).

MNIST_DATA_PATH = '../../Lectures Slides {MFDNN}/Notebooks {MFDNN}/mnist_data'
# MNIST training data
full_set = datasets.MNIST(root=MNIST_DATA_PATH, train=True,␣
 ↪transform=transforms.ToTensor(), download=True)

# only use first 10% of the data, subset of 6000 images
train_set = torch.utils.data.Subset(full_set, range(len(full_set)//10))
# randomise the labels
for i in range(len(train_set)):
    train_set.dataset.targets[train_set.indices[i]] = torch.randint(0, 10,␣
 ↪(1,)).item()
```

2.2 Model setup (given)

[ ]:
```
# (Modified version of AlexNet)
class AlexNet(nn.Module):
    def __init__(self, num_class=10):
        super(AlexNet, self).__init__()

        self.conv_layer1 = nn.Sequential(
            nn.Conv2d(1, 96, kernel_size=4),
            nn.ReLU(inplace=True),
            nn.Conv2d(96, 96, kernel_size=3),
            nn.ReLU(inplace=True)
        )
        self.conv_layer2 = nn.Sequential(
            nn.Conv2d(96, 256, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.conv_layer3 = nn.Sequential(
            nn.Conv2d(256, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )

        self.fc_layer1 = nn.Sequential(
            nn.Dropout(),
            nn.Linear(6400, 800),
            nn.ReLU(inplace=True),
            nn.Linear(800, 10)
        )

    def forward(self, x):
        output = self.conv_layer1(x)
        output = self.conv_layer2(output)
        output = self.conv_layer3(output)
        output = torch.flatten(output, 1)
        output = self.fc_layer1(output)
        return output
```

[ ]:
```
learning_rate = 0.1
batch_size = 64
epochs = 150
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

model = AlexNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

train_loader = DataLoader(dataset=train_set, batch_size=64, shuffle=True)
```

2.3 Training (and stats)

[ ]:
```
tick = time.time()
training_performance_data = []
for epoch in range(epochs):
    print(f"Epoch {epoch + 1:>3}/{epochs}")
    train_loss, correct = 0, 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        output = model(images)
        loss = loss_function(output, labels)
        train_loss += loss.item()
        correct += torch.eq(torch.argmax(output, dim=1), labels).sum().item()
        loss.backward()

        optimizer.step()
    training_performance_data.append((train_loss/len(train_loader),  # loss␣
↪added to train_loss variable is mean loss
                                      correct/len(train_set)))       # accuracy␣
↪added to correct variable is total num
    print(f"Loss: {training_performance_data[-1][0]:.4f}, Accuracy:␣
↪{training_performance_data[-1][1]:.4f}")
tock = time.time()
print(f"Total training time: {tock - tick}")
```

```
Epoch   1/150
Loss: 2.3028, Accuracy: 0.1042
Epoch   2/150
Loss: 2.3022, Accuracy: 0.1068
Epoch   3/150
Loss: 2.3023, Accuracy: 0.1073
Epoch   4/150
Loss: 2.3022, Accuracy: 0.1052
Epoch   5/150
Loss: 2.3023, Accuracy: 0.1085
Epoch   6/150
```

```
Loss: 2.3022, Accuracy: 0.1085
Epoch   7/150
Loss: 2.3022, Accuracy: 0.1083
Epoch   8/150
Loss: 2.3023, Accuracy: 0.1060
Epoch   9/150
Loss: 2.3023, Accuracy: 0.1087
Epoch  10/150
Loss: 2.3022, Accuracy: 0.1085
Epoch  11/150
Loss: 2.3022, Accuracy: 0.1072
Epoch  12/150
Loss: 2.3021, Accuracy: 0.1072
Epoch  13/150
Loss: 2.3022, Accuracy: 0.1083
Epoch  14/150
Loss: 2.3023, Accuracy: 0.1085
Epoch  15/150
Loss: 2.3022, Accuracy: 0.1085
Epoch  16/150
Loss: 2.3021, Accuracy: 0.1085
Epoch  17/150
Loss: 2.3022, Accuracy: 0.1037
Epoch  18/150
Loss: 2.3022, Accuracy: 0.1075
Epoch  19/150
Loss: 2.3020, Accuracy: 0.1085
Epoch  20/150
Loss: 2.3022, Accuracy: 0.1057
Epoch  21/150
Loss: 2.3022, Accuracy: 0.1085
Epoch  22/150
Loss: 2.3022, Accuracy: 0.1085
Epoch  23/150
Loss: 2.3022, Accuracy: 0.1085
Epoch  24/150
Loss: 2.3021, Accuracy: 0.1068
Epoch  25/150
Loss: 2.3021, Accuracy: 0.1085
Epoch  26/150
Loss: 2.3022, Accuracy: 0.1058
Epoch  27/150
Loss: 2.3021, Accuracy: 0.1078
Epoch  28/150
Loss: 2.3020, Accuracy: 0.1085
Epoch  29/150
Loss: 2.3021, Accuracy: 0.1085
Epoch  30/150
```

```
Loss: 2.3020, Accuracy: 0.1090
Epoch  31/150
Loss: 2.3022, Accuracy: 0.1085
Epoch  32/150
Loss: 2.3021, Accuracy: 0.1088
Epoch  33/150
Loss: 2.3020, Accuracy: 0.1083
Epoch  34/150
Loss: 2.3021, Accuracy: 0.1037
Epoch  35/150
Loss: 2.3020, Accuracy: 0.1085
Epoch  36/150
Loss: 2.3019, Accuracy: 0.1068
Epoch  37/150
Loss: 2.3020, Accuracy: 0.1078
Epoch  38/150
Loss: 2.3020, Accuracy: 0.1082
Epoch  39/150
Loss: 2.3018, Accuracy: 0.1065
Epoch  40/150
Loss: 2.3018, Accuracy: 0.1087
Epoch  41/150
Loss: 2.3019, Accuracy: 0.1067
Epoch  42/150
Loss: 2.3018, Accuracy: 0.1088
Epoch  43/150
Loss: 2.3018, Accuracy: 0.1063
Epoch  44/150
Loss: 2.3016, Accuracy: 0.1102
Epoch  45/150
Loss: 2.3016, Accuracy: 0.1082
Epoch  46/150
Loss: 2.3016, Accuracy: 0.1075
Epoch  47/150
Loss: 2.3014, Accuracy: 0.1057
Epoch  48/150
Loss: 2.3014, Accuracy: 0.1087
Epoch  49/150
Loss: 2.3012, Accuracy: 0.1070
Epoch  50/150
Loss: 2.3006, Accuracy: 0.1092
Epoch  51/150
Loss: 2.3006, Accuracy: 0.1120
Epoch  52/150
Loss: 2.3006, Accuracy: 0.1057
Epoch  53/150
Loss: 2.3001, Accuracy: 0.1055
Epoch  54/150
```

```
Loss: 2.2997, Accuracy: 0.1140
Epoch  55/150
Loss: 2.2995, Accuracy: 0.1113
Epoch  56/150
Loss: 2.2989, Accuracy: 0.1153
Epoch  57/150
Loss: 2.2990, Accuracy: 0.1160
Epoch  58/150
Loss: 2.2984, Accuracy: 0.1180
Epoch  59/150
Loss: 2.2978, Accuracy: 0.1188
Epoch  60/150
Loss: 2.2967, Accuracy: 0.1152
Epoch  61/150
Loss: 2.2952, Accuracy: 0.1178
Epoch  62/150
Loss: 2.2959, Accuracy: 0.1208
Epoch  63/150
Loss: 2.2947, Accuracy: 0.1157
Epoch  64/150
Loss: 2.2944, Accuracy: 0.1140
Epoch  65/150
Loss: 2.2932, Accuracy: 0.1160
Epoch  66/150
Loss: 2.2918, Accuracy: 0.1187
Epoch  67/150
Loss: 2.2904, Accuracy: 0.1180
Epoch  68/150
Loss: 2.2893, Accuracy: 0.1233
Epoch  69/150
Loss: 2.2872, Accuracy: 0.1278
Epoch  70/150
Loss: 2.2860, Accuracy: 0.1198
Epoch  71/150
Loss: 2.2823, Accuracy: 0.1303
Epoch  72/150
Loss: 2.2812, Accuracy: 0.1298
Epoch  73/150
Loss: 2.2799, Accuracy: 0.1322
Epoch  74/150
Loss: 2.2790, Accuracy: 0.1357
Epoch  75/150
Loss: 2.2712, Accuracy: 0.1397
Epoch  76/150
Loss: 2.2690, Accuracy: 0.1425
Epoch  77/150
Loss: 2.2631, Accuracy: 0.1473
Epoch  78/150
```

```
Loss: 2.2573, Accuracy: 0.1522
Epoch  79/150
Loss: 2.2539, Accuracy: 0.1448
Epoch  80/150
Loss: 2.2488, Accuracy: 0.1580
Epoch  81/150
Loss: 2.2413, Accuracy: 0.1627
Epoch  82/150
Loss: 2.2276, Accuracy: 0.1687
Epoch  83/150
Loss: 2.2221, Accuracy: 0.1657
Epoch  84/150
Loss: 2.2066, Accuracy: 0.1847
Epoch  85/150
Loss: 2.1900, Accuracy: 0.1932
Epoch  86/150
Loss: 2.1717, Accuracy: 0.2072
Epoch  87/150
Loss: 2.1544, Accuracy: 0.2175
Epoch  88/150
Loss: 2.1306, Accuracy: 0.2195
Epoch  89/150
Loss: 2.1005, Accuracy: 0.2347
Epoch  90/150
Loss: 2.0631, Accuracy: 0.2533
Epoch  91/150
Loss: 2.0208, Accuracy: 0.2715
Epoch  92/150
Loss: 1.9706, Accuracy: 0.2887
Epoch  93/150
Loss: 1.9199, Accuracy: 0.3137
Epoch  94/150
Loss: 1.8446, Accuracy: 0.3410
Epoch  95/150
Loss: 1.7694, Accuracy: 0.3715
Epoch  96/150
Loss: 1.6922, Accuracy: 0.4068
Epoch  97/150
Loss: 1.5902, Accuracy: 0.4410
Epoch  98/150
Loss: 1.4905, Accuracy: 0.4738
Epoch  99/150
Loss: 1.4161, Accuracy: 0.5092
Epoch 100/150
Loss: 1.2817, Accuracy: 0.5610
Epoch 101/150
Loss: 1.1700, Accuracy: 0.5903
Epoch 102/150
```

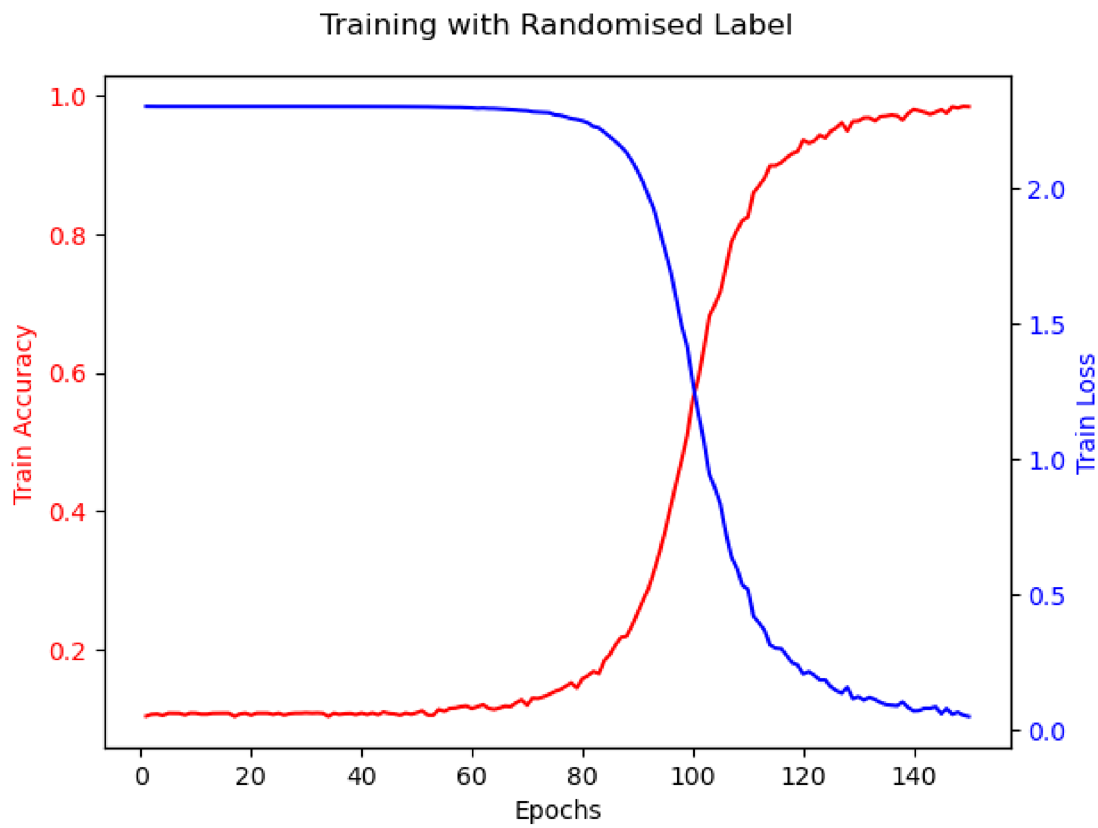
```
Loss: 1.0667, Accuracy: 0.6338
Epoch 103/150
Loss: 0.9425, Accuracy: 0.6812
Epoch 104/150
Loss: 0.8927, Accuracy: 0.6977
Epoch 105/150
Loss: 0.8322, Accuracy: 0.7163
Epoch 106/150
Loss: 0.7249, Accuracy: 0.7513
Epoch 107/150
Loss: 0.6372, Accuracy: 0.7882
Epoch 108/150
Loss: 0.5951, Accuracy: 0.8050
Epoch 109/150
Loss: 0.5318, Accuracy: 0.8195
Epoch 110/150
Loss: 0.5168, Accuracy: 0.8242
Epoch 111/150
Loss: 0.4177, Accuracy: 0.8595
Epoch 112/150
Loss: 0.3928, Accuracy: 0.8695
Epoch 113/150
Loss: 0.3664, Accuracy: 0.8800
Epoch 114/150
Loss: 0.3115, Accuracy: 0.8983
Epoch 115/150
Loss: 0.3000, Accuracy: 0.8983
Epoch 116/150
Loss: 0.2980, Accuracy: 0.9028
Epoch 117/150
Loss: 0.2732, Accuracy: 0.9098
Epoch 118/150
Loss: 0.2461, Accuracy: 0.9157
Epoch 119/150
Loss: 0.2372, Accuracy: 0.9190
Epoch 120/150
Loss: 0.2044, Accuracy: 0.9353
Epoch 121/150
Loss: 0.2130, Accuracy: 0.9310
Epoch 122/150
Loss: 0.1996, Accuracy: 0.9340
Epoch 123/150
Loss: 0.1821, Accuracy: 0.9423
Epoch 124/150
Loss: 0.1823, Accuracy: 0.9383
Epoch 125/150
Loss: 0.1571, Accuracy: 0.9482
Epoch 126/150
```

```
Loss: 0.1424, Accuracy: 0.9532
Epoch 127/150
Loss: 0.1335, Accuracy: 0.9603
Epoch 128/150
Loss: 0.1549, Accuracy: 0.9487
Epoch 129/150
Loss: 0.1131, Accuracy: 0.9623
Epoch 130/150
Loss: 0.1205, Accuracy: 0.9628
Epoch 131/150
Loss: 0.1075, Accuracy: 0.9675
Epoch 132/150
Loss: 0.1183, Accuracy: 0.9677
Epoch 133/150
Loss: 0.1098, Accuracy: 0.9630
Epoch 134/150
Loss: 0.0990, Accuracy: 0.9692
Epoch 135/150
Loss: 0.0910, Accuracy: 0.9700
Epoch 136/150
Loss: 0.0889, Accuracy: 0.9715
Epoch 137/150
Loss: 0.0870, Accuracy: 0.9703
Epoch 138/150
Loss: 0.1006, Accuracy: 0.9643
Epoch 139/150
Loss: 0.0806, Accuracy: 0.9737
Epoch 140/150
Loss: 0.0678, Accuracy: 0.9798
Epoch 141/150
Loss: 0.0693, Accuracy: 0.9778
Epoch 142/150
Loss: 0.0775, Accuracy: 0.9755
Epoch 143/150
Loss: 0.0772, Accuracy: 0.9727
Epoch 144/150
Loss: 0.0828, Accuracy: 0.9755
Epoch 145/150
Loss: 0.0570, Accuracy: 0.9793
Epoch 146/150
Loss: 0.0768, Accuracy: 0.9742
Epoch 147/150
Loss: 0.0554, Accuracy: 0.9833
Epoch 148/150
Loss: 0.0632, Accuracy: 0.9815
Epoch 149/150
Loss: 0.0519, Accuracy: 0.9842
Epoch 150/150
```

```
Loss: 0.0469, Accuracy: 0.9837
Total training time: 1413.3870449066162
```

2.4 Plotting

[ ]:
```python
import matplotlib.pyplot as plt

# Sample data
epoch_range = range(1, epochs+1)  # 150 epochs
train_loss = [x[0] for x in training_performance_data]
train_accuracy = [x[1] for x in training_performance_data]

fig, ax1 = plt.subplots()

# Plotting training accuracy on the primary y-axis
color = 'r'
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Train Accuracy', color=color)
ax1.plot(epoch_range, train_accuracy, color=color)
ax1.tick_params(axis='y', labelcolor=color)

# Creating a second y-axis for the training loss
ax2 = ax1.twinx()
color = 'b'
ax2.set_ylabel('Train Loss', color=color)
ax2.plot(epoch_range, train_loss, color=color)
ax2.tick_params(axis='y', labelcolor=color)

# Adding title and a tight layout
fig.suptitle('Training with Randomised Label')
fig.tight_layout()

# Show the plot
plt.show()
```


Training with Randomised Label
Train Accuracy
Train Loss
Epochs
1.0
0.8
0.6
0.4
0.2
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100
120
140