mnist

November 27, 2024

[22]:
```
import torch
import torch.nn as nn
import torchbnn as bnn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import make_grid

import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

%matplotlib inline
```

[21]:
```
print(torch.cuda.is_available())
```

```
True
```

[23]:
```
# Convert MNIST images into 4D tensors (#images, height, width, color channel)

transform = transforms.ToTensor()
```

[24]:
```
# Create training data
train_data = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
```

[25]:
```
# Create test data
test_data = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)
```

[26]:
```
# Params
filter = True
if filter:
    filtered_class = 5

load_model = True
```

[65]:
```python
# Remove the class 5 from the data
if filter:
    filtered_indices = [i for i, (_,label) in enumerate(train_data) if label!=5]
    train_data_filtered = torch.utils.data.Subset(train_data, filtered_indices)
else:
    train_data_filtered = train_data
```

[66]:
```python
# Create small batch size for images
train_loader = DataLoader(train_data_filtered, batch_size=10, shuffle=True)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)
```

[29]:
```python
#Define the Model class
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()

        # Bayesian Convolutional Layers
        self.conv1 = bnn.BayesConv2d(prior_mu=0, prior_sigma=0.1,␣
 ↪in_channels=1, out_channels=6, kernel_size=3, stride=1)
        self.conv2 = bnn.BayesConv2d(prior_mu=0, prior_sigma=0.1,␣
 ↪in_channels=6, out_channels=16, kernel_size=3, stride=1)

        # Bayesian Fully Connected Layers
        self.fc1 = bnn.BayesLinear(prior_mu=0, prior_sigma=0.1,␣
 ↪in_features=16*5*5, out_features=120)      #16 filters, 5x5 size of each␣
 ↪output "image" in the conv2 layer
        self.fc2 = bnn.BayesLinear(prior_mu=0, prior_sigma=0.1,␣
 ↪in_features=120, out_features=84)
        self.fc3 = bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=84,␣
 ↪out_features=10)

    def forward(self, X):
        # Pass through convolutional and pooling layers with ReLU activation
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2,2)  #kernel = 2x2, stride = 2
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X,2,2)

        # Flatten out the data
        X = X.view(-1, 16*5*5)      # -1 so that we can vary the batch size

        # Pass through the fully connected layers
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)

        return F.log_softmax(X, dim=1)
```

[30]:
```python
# Create an instance of the model
torch.manual_seed(41)
model = ConvolutionalNetwork()
```

[31]:
```python
# Move the model to the cuda device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

[31]:
```
ConvolutionalNetwork(
  (conv1): BayesConv2d(0, 0.1, 1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): BayesConv2d(0, 0.1, 6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=400,
out_features=120, bias=True)
  (fc2): BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=120,
out_features=84, bias=True)
  (fc3): BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=84,
out_features=10, bias=True)
)
```

[32]:
```python
# Select loss function and optimizer
ce_loss = nn.CrossEntropyLoss()
kl_loss = bnn.BKLLoss(reduction='mean', last_layer_only=False)
kl_weight=0.1
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

[35]:
```python
# Variables to track losses
epochs = 5

if load_model:
    model.load_state_dict(torch.load('mnist_bnn.pt'))

else:
    #For loop over epochs
    for i in range(epochs):
        trn_corr = 0
        tst_corr = 0

        #Train
        for b, (X_train, y_train) in enumerate(train_loader):
            b+=1                                            #Start the batch at␣
 ↪1

            # Move the data and labels to the cuda device, if available
            X_train, y_train = X_train.to(device), y_train.to(device)

            # Forward pass
            y_pred = model(X_train)
```

```
            ce = ce_loss(y_pred, y_train)
            kl = kl_loss(model)
            loss = ce+ kl*kl_weight

            # Backward pass - update parameters
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            #Print out some results
            if b%600==0:
                print(f"Epoch = {i} Batch = {b} Loss = {loss.item()}")
                print(f"Allocated memory: {torch.cuda.memory_allocated() /␣
↪1024**2} MB")
                print(f"Cached memory: {torch.cuda.memory_reserved() / 1024**2}␣
↪MB")

    # Save the model after training
    torch.save(model.state_dict(), 'mnist_bnn.pt')

    X_test = torch.stack([data[0] for data in train_data])
    y_test = torch.LongTensor([data[1] for data in train_data])

    _, predicted = torch.max(model(X_train).data, 1)
    total = y_train.size(0)
    correct = (predicted == y_train).sum()
    print('- Accuracy: %f %%' % (100 * float(correct) / total))
    print('- CE : %2.2f, KL : %2.2f' % (ce.item(), kl.item()))
```

```
Epoch = 0 Batch = 600 Loss = 0.24118585884571075
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 1200 Loss = 0.6067807674407959
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 1800 Loss = 0.3992709219455719
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 2400 Loss = 0.15570957958698273
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 3000 Loss = 0.7966600060462952
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 3600 Loss = 0.14562556147575378
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
```

```
Epoch = 0 Batch = 4200 Loss = 0.5586368441581726
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 4800 Loss = 0.19034206867218018
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 0 Batch = 5400 Loss = 0.15609468519687653
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 600 Loss = 0.6022573709487915
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 1200 Loss = 0.19605833292007446
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 1800 Loss = 0.15260295569896698
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 2400 Loss = 0.21523310244083405
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 3000 Loss = 0.7696467638015747
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 3600 Loss = 0.5838574767112732
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 4200 Loss = 0.1513512283563614
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 4800 Loss = 0.13453033566474915
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 1 Batch = 5400 Loss = 0.142412930727005
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 600 Loss = 0.14295876026153564
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 1200 Loss = 0.1258770227432251
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 1800 Loss = 0.7939824461936951
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 2400 Loss = 0.12541188299655914
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
```

```
Epoch = 2 Batch = 3000 Loss = 0.23478063941001892
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 3600 Loss = 0.1359878033399582
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 4200 Loss = 0.47669467329978943
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 4800 Loss = 0.21517550945281982
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 2 Batch = 5400 Loss = 0.14041993021965027
Allocated memory: 18.13916015625 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 600 Loss = 0.16538304090499878
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 1200 Loss = 0.12488064169883728
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 1800 Loss = 0.15677598118782043
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 2400 Loss = 0.24961453676223755
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 3000 Loss = 0.11143389344215393
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 3600 Loss = 0.1973770558834076
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 4200 Loss = 0.14807219803333282
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 4800 Loss = 0.13637088239192963
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 3 Batch = 5400 Loss = 0.14207343757152557
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 600 Loss = 0.14982527494430542
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 1200 Loss = 0.11261936277151108
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
```

```
Epoch = 4 Batch = 1800 Loss = 0.3657877445220947
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 2400 Loss = 0.3280436396598816
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 3000 Loss = 0.1905529797077179
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 3600 Loss = 0.12605077028274536
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 4200 Loss = 0.14452897012233734
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 4800 Loss = 0.15879561007022858
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
Epoch = 4 Batch = 5400 Loss = 0.12433543801307678
Allocated memory: 18.169921875 MB
Cached memory: 26.0 MB
- Accuracy: 100.000000 %
- CE : 0.00, KL : 1.09
```

[37]:

```
#Test
tst_corr = 0

with torch.no_grad():
    for b, (X_test, y_test) in enumerate(test_loader):
        X_test, y_test = X_test.to(device), y_test.to(device)
        y_val = model(X_test)
        predicted = torch.max(y_val.data, 1)[1]
        batch_corr = (predicted == y_test).sum()
        tst_corr += batch_corr

print(f"Test accuracy : {(tst_corr/10000)*100} %")
```

```
Test accuracy : 88.08999633789062 %
```

[38]:

```
X_test = torch.stack([data[0] for data in test_data])
y_test = torch.LongTensor([data[1] for data in test_data])
```

[40]:

```
torch.argwhere(y_test==5).T
```

[40]:

```
tensor([[   8,   15,   23,   45,   52,   53,   59,  102,  120,  127,  129,  132,
          152,  153,  155,  162,  165,  167,  182,  187,  207,  211,  218,  219,
          240,  253,  261,  283,  289,  317,  319,  333,  340,  347,  351,  352,
          356,  364,  367,  375,  395,  397,  406,  412,  433,  460,  469,  478,
```

483, 491, 502, 509, 518, 540, 570, 588, 604, 618, 638, 645,
654, 674, 692, 694, 710, 711, 720, 739, 751, 766, 778, 779,
785, 791, 797, 812, 856, 857, 866, 869, 897, 934, 935, 951,
955, 970, 978, 1003, 1022, 1032, 1041, 1046, 1070, 1073, 1082, 1087,
1089, 1102, 1113, 1115, 1131, 1135, 1144, 1146, 1168, 1169, 1190, 1221,
1233, 1235, 1243, 1252, 1258, 1272, 1281, 1285, 1289, 1299, 1331, 1334,
1339, 1340, 1370, 1376, 1378, 1393, 1405, 1406, 1421, 1447, 1460, 1466,
1467, 1471, 1473, 1476, 1493, 1510, 1521, 1525, 1550, 1598, 1618, 1629,
1635, 1637, 1639, 1641, 1653, 1670, 1672, 1677, 1684, 1693, 1737, 1747,
1752, 1755, 1761, 1810, 1833, 1846, 1847, 1860, 1866, 1874, 1879, 1896,
1902, 1910, 1911, 1917, 1931, 1940, 1948, 1954, 1967, 1970, 1999, 2001,
2003, 2021, 2029, 2030, 2035, 2037, 2040, 2064, 2073, 2077, 2078, 2100,
2103, 2113, 2114, 2125, 2134, 2159, 2162, 2180, 2192, 2207, 2214, 2224,
2237, 2241, 2247, 2279, 2282, 2291, 2322, 2339, 2346, 2369, 2400, 2413,
2445, 2452, 2460, 2476, 2487, 2515, 2518, 2525, 2526, 2540, 2545, 2546,
2554, 2556, 2558, 2559, 2569, 2573, 2574, 2581, 2586, 2597, 2604, 2606,
2611, 2616, 2644, 2653, 2668, 2670, 2682, 2686, 2689, 2697, 2698, 2727,
2743, 2768, 2772, 2773, 2775, 2790, 2797, 2798, 2805, 2810, 2814, 2829,
2832, 2839, 2850, 2855, 2903, 2909, 2913, 2919, 2922, 2925, 2930, 2948,
2951, 2956, 2957, 2969, 2970, 2986, 2987, 3007, 3022, 3028, 3053, 3093,
3095, 3100, 3102, 3113, 3115, 3117, 3119, 3127, 3145, 3157, 3171, 3183,
3199, 3220, 3275, 3295, 3311, 3312, 3321, 3334, 3335, 3336, 3345, 3372,
3393, 3408, 3414, 3416, 3462, 3468, 3470, 3506, 3537, 3552, 3556, 3558,
3565, 3569, 3570, 3590, 3591, 3595, 3619, 3623, 3631, 3636, 3645, 3654,
3663, 3678, 3691, 3702, 3750, 3754, 3756, 3763, 3776, 3778, 3788, 3797,
3806, 3810, 3814, 3826, 3837, 3855, 3860, 3863, 3877, 3890, 3893, 3898,
3902, 3907, 3917, 3918, 3928, 3929, 3952, 3955, 3957, 3960, 3968, 3994,
4031, 4052, 4054, 4056, 4059, 4067, 4072, 4076, 4094, 4108, 4118, 4131,
4152, 4177, 4196, 4202, 4219, 4226, 4233, 4236, 4254, 4255, 4261, 4263,
4271, 4300, 4302, 4307, 4310, 4312, 4315, 4323, 4330, 4338, 4340, 4355,
4356, 4359, 4360, 4364, 4368, 4374, 4378, 4381, 4420, 4422, 4440, 4461,
4463, 4472, 4520, 4529, 4548, 4569, 4577, 4583, 4596, 4637, 4645, 4689,
4696, 4711, 4712, 4722, 4728, 4749, 4762, 4763, 4766, 4771, 4809, 4810,
4828, 4830, 4844, 4867, 4888, 4892, 4902, 4915, 4933, 4942, 4971, 4979,
5020, 5021, 5056, 5083, 5098, 5102, 5111, 5134, 5152, 5160, 5170, 5174,
5187, 5194, 5196, 5197, 5206, 5207, 5222, 5223, 5229, 5275, 5285, 5295,
5302, 5325, 5339, 5347, 5351, 5364, 5374, 5389, 5397, 5400, 5410, 5420,
5432, 5445, 5451, 5473, 5480, 5488, 5510, 5518, 5528, 5570, 5571, 5572,
5574, 5579, 5598, 5608, 5618, 5624, 5632, 5633, 5658, 5662, 5668, 5682,
5697, 5706, 5711, 5726, 5735, 5742, 5752, 5769, 5779, 5802, 5807, 5821,
5833, 5843, 5852, 5862, 5867, 5874, 5885, 5891, 5910, 5913, 5922, 5937,
5947, 5957, 5964, 5972, 5981, 5982, 5985, 5997, 6028, 6042, 6043, 6053,
6067, 6077, 6087, 6095, 6120, 6136, 6142, 6146, 6148, 6155, 6165, 6186,
6196, 6206, 6215, 6216, 6227, 6236, 6244, 6257, 6270, 6277, 6282, 6291,
6314, 6324, 6333, 6341, 6368, 6385, 6386, 6390, 6392, 6405, 6414, 6415,
6476, 6483, 6486, 6491, 6500, 6518, 6522, 6525, 6530, 6537, 6544, 6548,
6573, 6598, 6600, 6611, 6620, 6638, 6706, 6716, 6728, 6746, 6775, 6788,

```
       6803, 6813, 6823, 6832, 6860, 6866, 6879, 6880, 6884, 6886, 6899, 6908,
       6909, 6932, 6942, 6952, 6964, 6965, 6977, 6981, 6991, 7003, 7018, 7029,
       7036, 7057, 7067, 7077, 7090, 7108, 7134, 7142, 7155, 7160, 7178, 7187,
       7195, 7240, 7241, 7264, 7274, 7284, 7294, 7304, 7306, 7315, 7324, 7351,
       7352, 7372, 7388, 7393, 7397, 7403, 7414, 7430, 7437, 7448, 7451, 7454,
       7474, 7475, 7478, 7498, 7511, 7521, 7531, 7541, 7542, 7559, 7577, 7578,
       7583, 7602, 7612, 7622, 7630, 7643, 7649, 7659, 7672, 7673, 7676, 7679,
       7698, 7715, 7732, 7742, 7752, 7777, 7779, 7793, 7797, 7808, 7809, 7819,
       7826, 7842, 7850, 7859, 7870, 7888, 7918, 7938, 7948, 7958, 7965, 7974,
       7988, 7996, 7997, 8034, 8035, 8038, 8049, 8062, 8072, 8082, 8089, 8122,
       8132, 8142, 8149, 8158, 8160, 8170, 8180, 8185, 8192, 8214, 8224, 8232,
       8270, 8275, 8299, 8327, 8331, 8348, 8366, 8386, 8412, 8415, 8444, 8445,
       8447, 8453, 8463, 8473, 8487, 8502, 8507, 8531, 8539, 8553, 8563, 8571,
       8578, 8601, 8630, 8632, 8643, 8645, 8652, 8653, 8656, 8665, 8676, 8686,
       8696, 8702, 8710, 8711, 8737, 8741, 8747, 8761, 8774, 8783, 8788, 8803,
       8813, 8823, 8834, 8835, 8847, 8853, 8855, 8863, 8878, 8909, 8940, 8948,
       8964, 8982, 8987, 9013, 9035, 9065, 9075, 9085, 9109, 9114, 9117, 9119,
       9132, 9133, 9159, 9160, 9176, 9184, 9194, 9228, 9234, 9260, 9268, 9277,
       9289, 9290, 9298, 9315, 9329, 9331, 9337, 9338, 9349, 9360, 9372, 9382,
       9391, 9398, 9400, 9422, 9427, 9428, 9465, 9478, 9481, 9482, 9493, 9503,
       9513, 9523, 9533, 9545, 9583, 9584, 9588, 9590, 9600, 9606, 9616, 9626,
       9651, 9671, 9675, 9685, 9702, 9709, 9719, 9729, 9747, 9749, 9754, 9770,
       9777, 9786, 9814, 9830, 9831, 9841, 9853, 9870, 9877, 9883, 9907, 9941,
       9970, 9982, 9988, 9998]])
```

[42]:
```
n_models = 100
X_test = X_test.to(device)
models_result = [model(X_test[3115]) for k in range(n_models)]
```

[48]:
```
models_result[0].argmax().item()
```

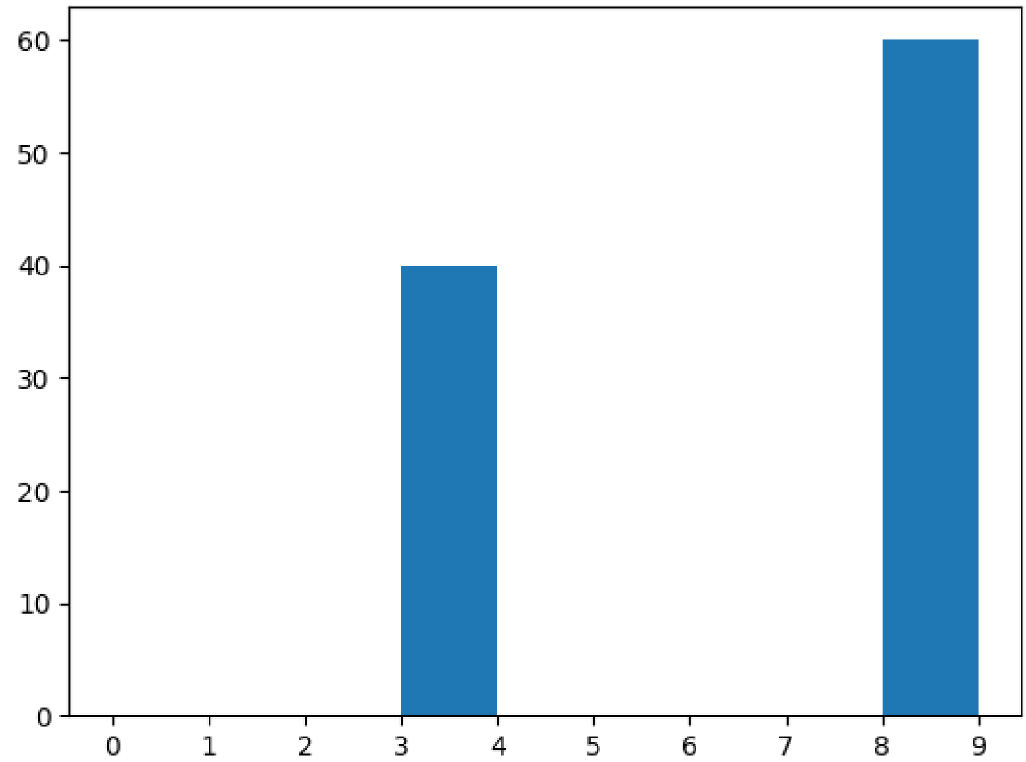
[48]:
```
3
```

[49]:
```
results = np.zeros(n_models)
for i in range(n_models):
  results[i] = models_result[i].argmax().item()
```

[50]:
```
results
```

[50]:
```
array([3., 9., 3., 3., 3., 3., 9., 9., 3., 9., 9., 9., 9., 3., 9., 9., 9.,
       3., 9., 9., 3., 3., 9., 9., 9., 9., 9., 3., 9., 3., 9., 9., 3., 3.,
       9., 9., 3., 3., 9., 9., 3., 9., 3., 9., 3., 9., 9., 9., 3., 3., 9.,
       9., 9., 9., 9., 3., 9., 3., 3., 3., 3., 9., 3., 9., 9., 3., 9., 3.,
       3., 9., 9., 3., 9., 9., 9., 9., 3., 9., 3., 9., 9., 3., 9., 3., 9.,
       9., 3., 9., 9., 9., 3., 9., 3., 9., 9., 3., 9., 9., 9., 3.])
```

[63]:
```
plt.hist(results, bins=[0,1,2,3,4,5,6,7,8,9], align='mid')
plt.xticks([0,1,2,3,4,5,6,7,8,9], ha='center')
plt.show()
```



[89]:
```
# Create slice of test dataset that contains only the filteredClass
if filter:
    filtered_indices = [i for i, (_,label) in enumerate(test_data) if label==5]
    test_data_filtered_unseen = torch.utils.data.Subset(test_data,␣
 ↪filtered_indices)
```

[71]:
```
test_data_filtered_unseen
```

[71]: <torch.utils.data.dataset.Subset at 0x78bafc2c82e0>

[88]:
```
test_filt_loader = DataLoader(test_data_filtered_unseen, batch_size =␣
 ↪len(test_data_filtered_unseen))

images, labels = next(iter(test_filt_loader))
images, labels = images.to(device), labels.to(device)
```

[91]:
```
samples = torch.zeros((n_models, len(test_data_filtered_unseen), 10))
```

```
for i in range(n_models) :
  print("\r", "\tTest run {}/{}".format(i+1, n_models), end="")

  samples[i,:,:] = torch.exp(model(images))
```

```
        Test run 1/100
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-91-2a0d76abcb85> in <cell line: 3>()
      4   print("\r", "\tTest run {}/{}".format(i+1, n_models), end="")
      5 
----> 6   samples[i,:,:] = torch.exp(model(images))

RuntimeError: The expanded size of the tensor (892) must match the existing size
  ↪(1032) at non-singleton dimension 0.  Target sizes: [892, 10].  Tensor sizes:
  ↪[1032, 10]
```

[81]:

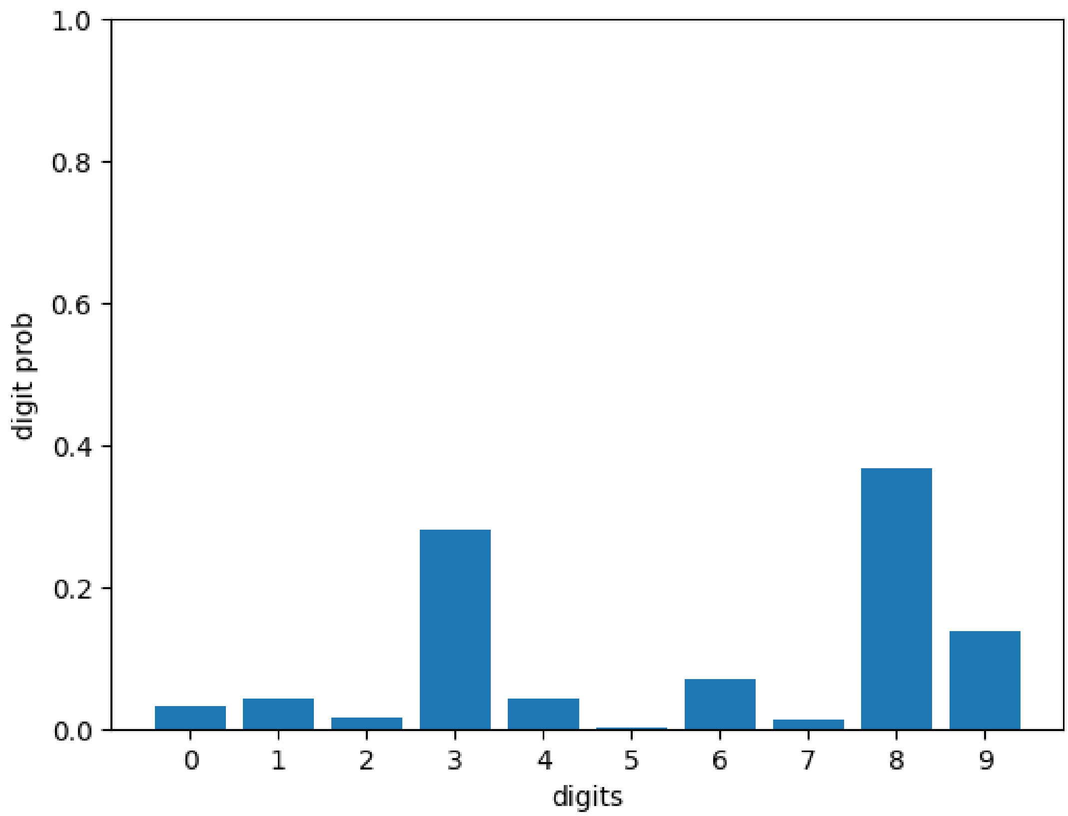
```
withinSampleMean = torch.mean(samples, dim=0)
samplesMean = torch.mean(samples, dim=(0,1))

withinSampleStd = torch.sqrt(torch.mean(torch.var(samples, dim=0), dim=0))
acrossSamplesStd = torch.std(withinSampleMean, dim=0)

print("")
print("Class prediction analysis:")
print("\tMean class probabilities:")
print(samplesMean)
print("\tPrediction standard deviation per sample:")
print(withinSampleStd)
print("\tPrediction standard deviation across samples:")
print(acrossSamplesStd)
```

```
Class prediction analysis:
        Mean class probabilities:
tensor([0.0320, 0.0422, 0.0144, 0.2814, 0.0420, 0.0010, 0.0691, 0.0131, 0.3674,
        0.1375], grad_fn=<MeanBackward1>)
        Prediction standard deviation per sample:
tensor([0.0367, 0.0353, 0.0122, 0.0760, 0.0326, 0.0010, 0.0526, 0.0129, 0.0813,
        0.0530], grad_fn=<SqrtBackward0>)
        Prediction standard deviation across samples:
tensor([0.1022, 0.1386, 0.0227, 0.3617, 0.1260, 0.0020, 0.1945, 0.0568, 0.3720,
        0.2364], grad_fn=<StdBackward0>)
```

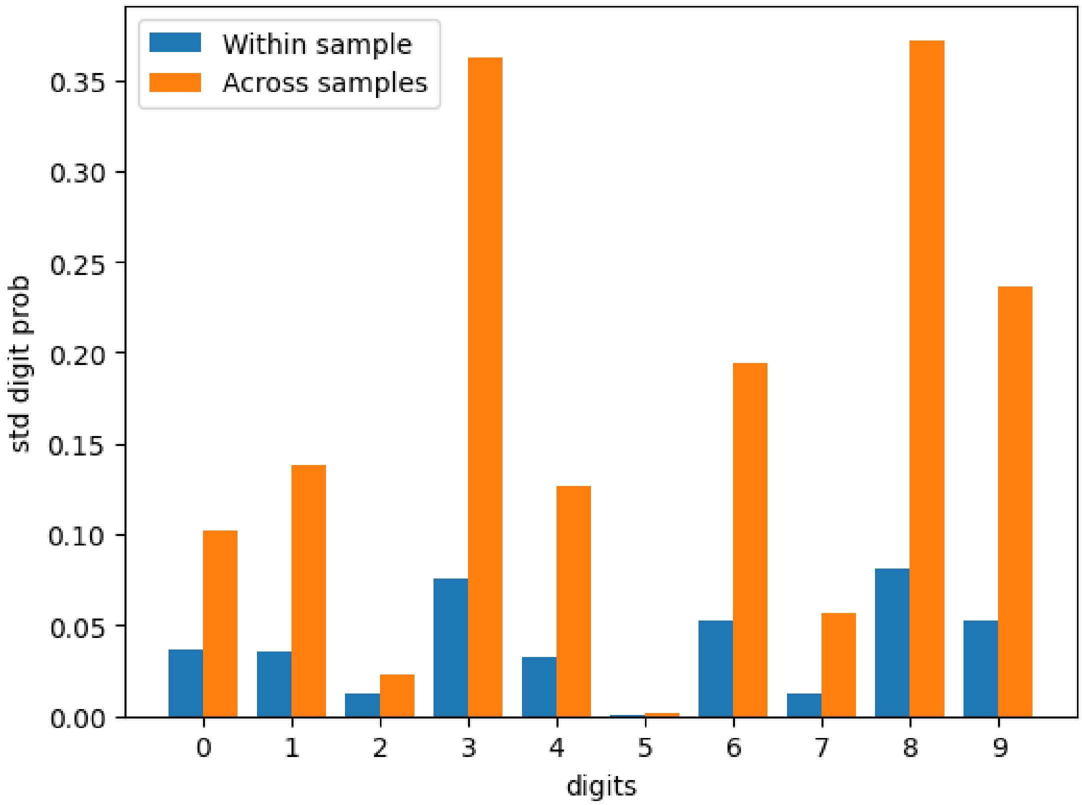
[85]:

```
plt.figure("Unseen class probabilities")
plt.bar(np.arange(10), samplesMean.detach().numpy())
```

```
plt.xlabel('digits')
plt.ylabel('digit prob')
plt.ylim([0,1])
plt.xticks(np.arange(10))
plt.show()
```



[87]:
```
plt.figure("Unseen inner and outter sample std")
plt.bar(np.arange(10)-0.2, withinSampleStd.detach().numpy(), width = 0.4,␣
 ↪label="Within sample")
plt.bar(np.arange(10)+0.2, acrossSamplesStd.detach().numpy(), width = 0.4,␣
 ↪label="Across samples")
plt.legend()
plt.xlabel('digits')
plt.ylabel('std digit prob')
plt.xticks(np.arange(10))
plt.show()
```



0.1 Testing against seen class

[90]:
```
# Create slice of test dataset that contains only the seen class (here, say 2)

filtered_indices = [i for i, (_,label) in enumerate(test_data) if label==2]
test_data_filtered_seen = torch.utils.data.Subset(test_data, filtered_indices)

test_filt_loader = DataLoader(test_data_filtered_seen, batch_size =␣
 ↪len(test_data_filtered_seen))

images, labels = next(iter(test_filt_loader))
images, labels = images.to(device), labels.to(device)
```

[92]:
```
samples = torch.zeros((n_models, len(test_data_filtered_seen), 10))

for i in range(n_models) :
  print("\r", "\tTest run {}/{}".format(i+1, n_models), end="")

  samples[i,:,:] = torch.exp(model(images))
```

```
Test run 100/100
```

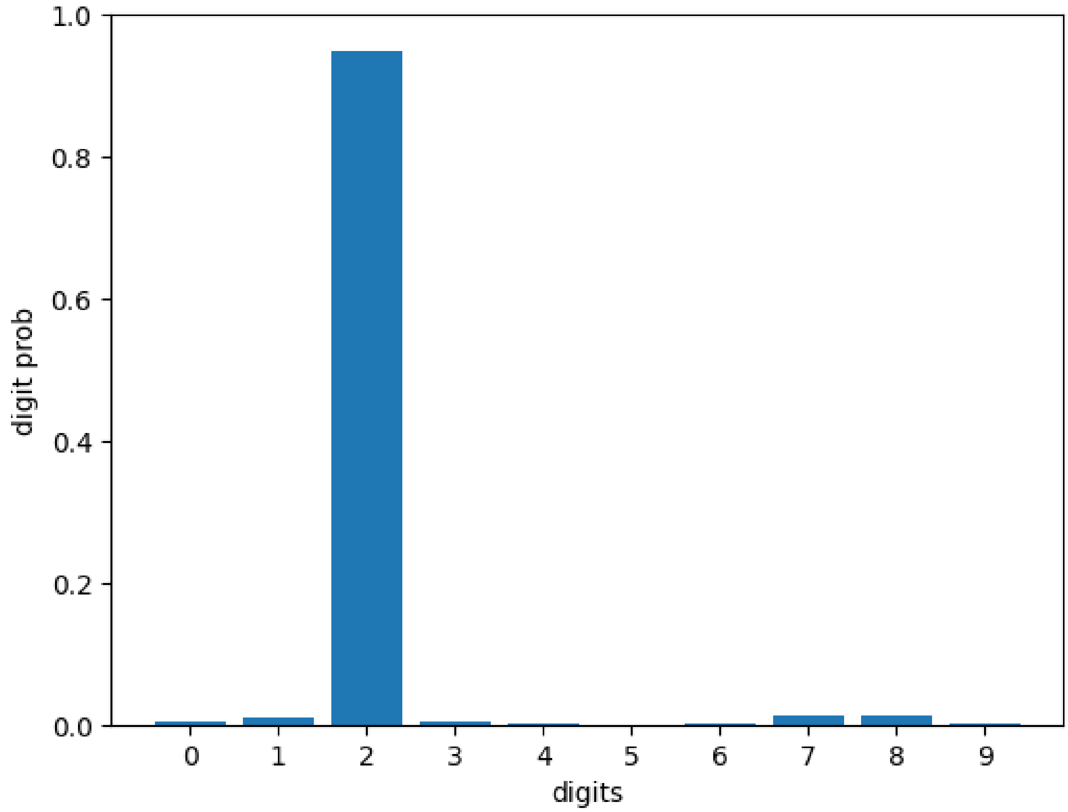
[93]:

```
withinSampleMean = torch.mean(samples, dim=0)
samplesMean = torch.mean(samples, dim=(0,1))

withinSampleStd = torch.sqrt(torch.mean(torch.var(samples, dim=0), dim=0))
acrossSamplesStd = torch.std(withinSampleMean, dim=0)

print("")
print("Class prediction analysis:")
print("\tMean class probabilities:")
print(samplesMean)
print("\tPrediction standard deviation per sample:")
print(withinSampleStd)
print("\tPrediction standard deviation across samples:")
print(acrossSamplesStd)
```

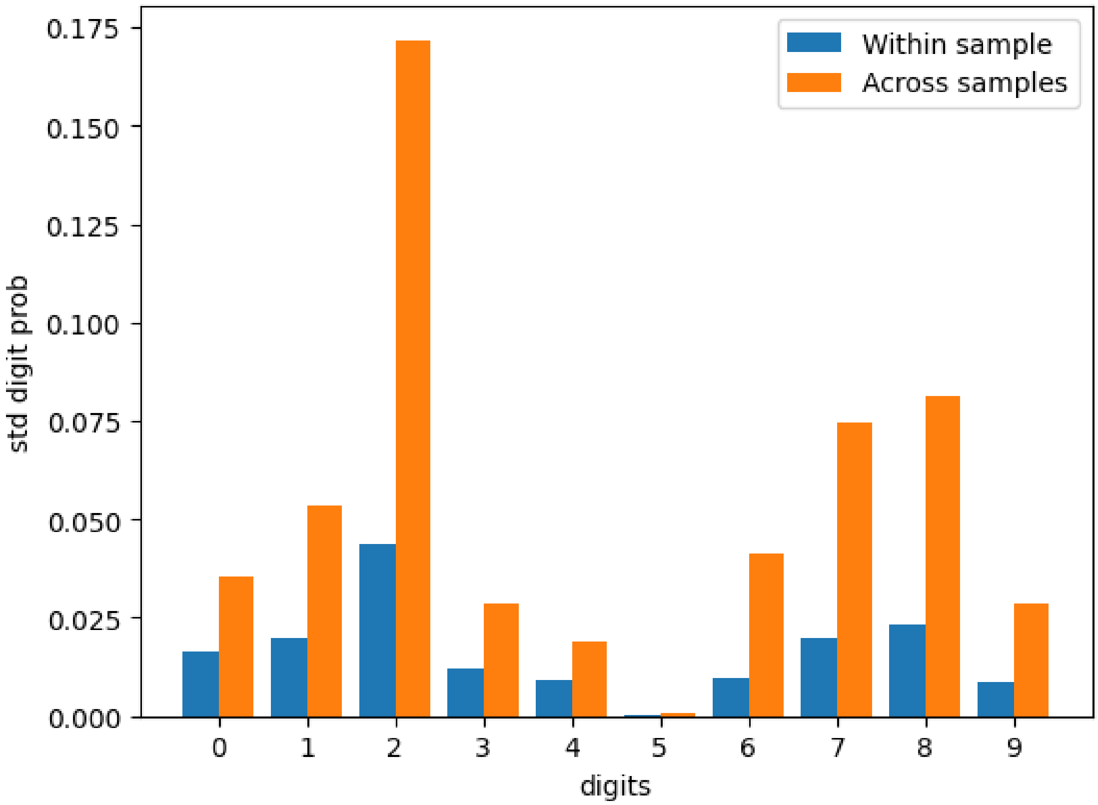
```
Class prediction analysis:
        Mean class probabilities:
tensor([3.7897e-03, 9.6894e-03, 9.4795e-01, 4.8016e-03, 3.0841e-03, 1.4895e-04,
        2.9987e-03, 1.1763e-02, 1.3214e-02, 2.5628e-03],
       grad_fn=<MeanBackward1>)
        Prediction standard deviation per sample:
tensor([0.0162, 0.0198, 0.0440, 0.0119, 0.0093, 0.0003, 0.0097, 0.0199, 0.0234,
        0.0086], grad_fn=<SqrtBackward0>)
        Prediction standard deviation across samples:
tensor([0.0353, 0.0533, 0.1717, 0.0288, 0.0191, 0.0006, 0.0414, 0.0744, 0.0812,
        0.0287], grad_fn=<StdBackward0>)
```

[94]:

```
plt.figure("Seen class probabilities")
plt.bar(np.arange(10), samplesMean.detach().numpy())
plt.xlabel('digits')
plt.ylabel('digit prob')
plt.ylim([0,1])
plt.xticks(np.arange(10))
plt.show()
```



[95]:
```
plt.figure("Seen inner and outter sample std")
plt.bar(np.arange(10)-0.2, withinSampleStd.detach().numpy(), width = 0.4,␣
 ↪label="Within sample")
plt.bar(np.arange(10)+0.2, acrossSamplesStd.detach().numpy(), width = 0.4,␣
 ↪label="Across samples")
plt.legend()
plt.xlabel('digits')
plt.ylabel('std digit prob')
plt.xticks(np.arange(10))
plt.show()
```



Within sample
Across samples
std digit prob
digits
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
1
2
3
4
5
6
7
8
9