MNIST

In [ ]:

```
import numpy as np
import torch
import torch.utils.data as tud
import torch.nn as tnn
```

In [ ]:

```
import matplotlib.pyplot as plt
```

In [ ]:

```
from copy import deepcopy as dcopy
```

In [ ]:

```
import torchvision as tv
import torchvision.datasets as tvds
```

In [ ]:

```
mnist_trainset = tvds.MNIST(root='./data', train=True, download=True, transform=tv.transforms.ToTensor())
mnist_testset = tvds.MNIST(root='./data', train=False, download=True, transform=tv.transforms.ToTensor())
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-
images-idx3-ubyte.gz
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-
labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-
idx3-ubyte.gz
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-
idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
Processing...
Done!
/usr/local/lib/python3.6/dist-packages/torchvision/datasets/mnist.py:480: UserWarning: The given NumPy
array is not writeable, and PyTorch does not support non-writeable tensors. This means you can write to
the underlying (supposedly non-writeable) NumPy array using the tensor. You may want to copy the array
to protect its data or make it writeable before converting it to a tensor. This type of warning will be
suppressed for the rest of this program. (Triggered internally at
/pytorch/torch/csrc/utils/tensor_numpy.cpp:141.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

In [ ]:

```
train_set, val_set = tud.random_split(mnist_trainset, [50000,10000], generator=torch.Generator().manual_s
```

In [ ]:

```
test_set=mnist_testset
```

In [ ]:

```
del mnist_testset, mnist_trainset
```

In [ ]:

```
# batch size increased for faster training
train_loader = tud.DataLoader(train_set, batch_size=64, shuffle=True, drop_last=True)
val_loader = tud.DataLoader(val_set, batch_size=10000, shuffle=True, drop_last=True)
test_loader = tud.DataLoader(test_set, batch_size=10000, shuffle=True, drop_last=True)
```

In [ ]:

```
class my_nn_2(tnn.Module):
  def __init__(self, h=128, d_in=784, d_out=10):
    super(my_nn_2, self).__init__()
    self.linear1 = tnn.Linear(d_in, h)
    self.linear2 = tnn.Linear(h, d_out)
    # https://medium.com/@zhang_yang/understanding-cross-entropy-implementation-in-pytorch-softmax-log-s
    # softmax + nll loss is worse than cross entropy loss

  def forward(self, x):
    h_relu = self.linear1(x).clamp(min=0)
    y_pred = self.linear2(h_relu)
    return y_pred
```

In [ ]:

```
crit_bce = tnn.CrossEntropyLoss()
crit_mse = tnn.MSELoss()
```

In [ ]:

```
def fit(h=128, epochs=10, criterion=tnn.CrossEntropyLoss(), lr=1e-3):
  model = my_nn_2(h)
  opt = torch.optim.SGD(model.parameters(), lr=lr)
```

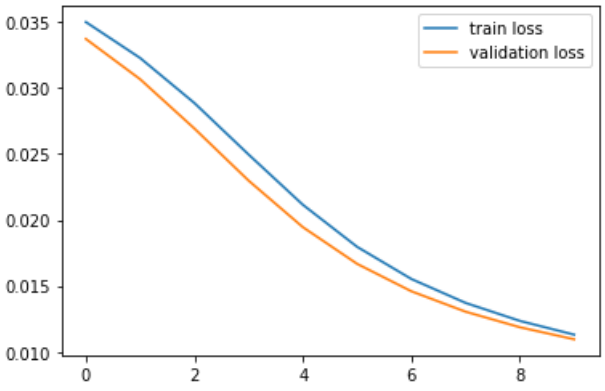
```
  tlh=[]
  tah=[]
  vlh=[]
  vah=[]
  vl_min=10**10
  for ep in range(epochs):
    model.train()
    training_loss=0
    acc=0
    for (idx, b) in enumerate(train_loader):
      xb=b[0].reshape(-1,784).float()
      yb=b[1]
      yb_pred = model(xb)
      loss=criterion(yb_pred, yb)
      training_loss+=loss
      acc+=torch.sum(torch.argmax(yb_pred, axis=1)==yb)
      opt.zero_grad()
      loss.backward()
      opt.step()
    tah.append(acc/50000)
    tlh.append(training_loss/50000)
    model.eval()
    with torch.no_grad():
      val_loss=0
      acc=0
      for (idx, b) in enumerate(val_loader):
        xb=b[0].reshape(-1,784).float()
        yb=b[1]
        yb_pred = model(xb)
        loss=criterion(yb_pred, yb)
        acc+=torch.sum(torch.argmax(yb_pred, axis=1)==yb)
        val_loss+=loss
      vah.append(acc/10000)
      vl=val_loss/64
      if vl<vl_min:
        best_model=dcopy(model)
        vl_min=vl
      vlh.append(vl)
  return best_model, tlh, tah, vlh, vah
```

In [ ]:

```
model, tlh, tah, vlh, vah = fit()
```

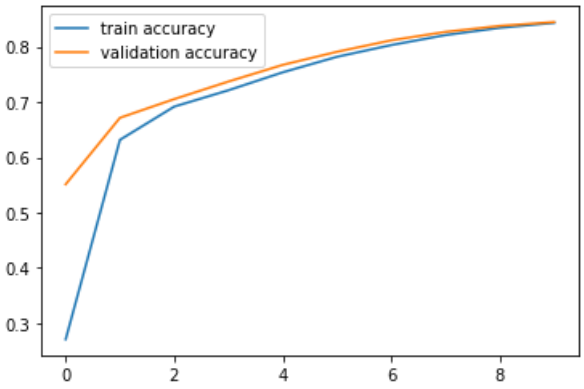
In [ ]:

```
plt.plot(tlh, label="train loss")
plt.plot(vlh, label="validation loss")
plt.legend()
plt.show()
```



In [ ]:

```
plt.plot(tah, label="train accuracy")
plt.plot(vah, label="validation accuracy")
plt.legend()
plt.show()
```

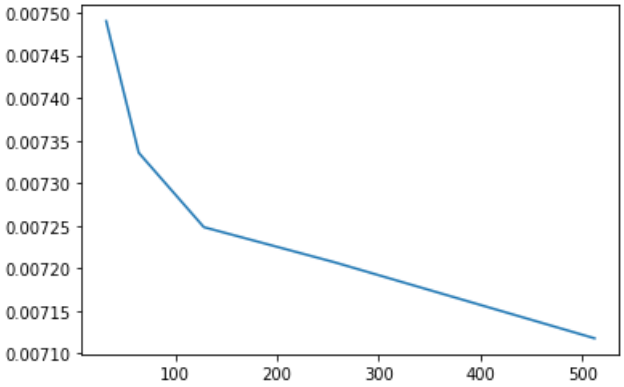


In [ ]:

```
hl_sizes=[32,64,128,256,512]
best_vl=[]
for hl in hl_sizes:
  _, tlh, tah, vlh, vah = fit(h=hl, epochs=20, lr=1e-3)
  best_vl.append(np.min(vlh))
```

In [ ]:

```
plt.plot(hl_sizes, best_vl)
plt.show()
```



In [ ]:

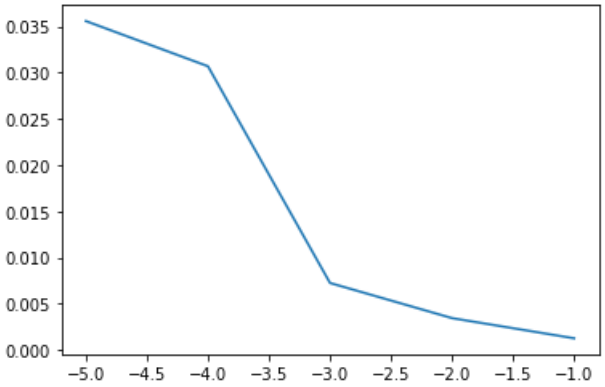
```
# h=512 gives best results
```

In [ ]:

```
lr_vals=[10**(i-5) for i in range(5)]
best_vl=[]
for lr in lr_vals:
  _, tlh, tah, vlh, vah = fit(epochs=20, lr=lr)
  best_vl.append(np.min(vlh))
```

In [ ]:

```
plt.plot(np.log10(lr_vals), best_vl)
plt.xlabel = "Learning rate as a power of 10"
plt.show()
```



In [ ]:

```
# lr=0.1 gives best results
```

In [ ]: