Lab1

October 7, 2024

1 Chapter 2

[2]:
```
import torch
```

1.1 Create a tensor `a` from `list(range(9))` . Predict then check what the size, offset, and strides are.

[3]:
```
a = torch.tensor(list(range(9)))
```

Predict:

- size: [9]
- offset: 0
- strides: (1)

Check:

[4]:
```
print(a)
print("size:", a.size())
print("offset:", a.storage_offset())
print("strides:", a.stride())
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
size: torch.Size([9])
offset: 0
strides: (1,)
```

1.2 Create a tensor `b = a.view(3, 3)` . What is the value of `b[1, 1]` ?

[5]:
```
b = a.view(3, 3)
b
```

[5]:
```
tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

value of `b[1, 1]` :

[6]:
```
b[1, 1]
```

[6]:
```
tensor(4)
```

1.3 Create a tensor `c = b[1:,1:]` . Predict then check what the size, offset, and strides are.

[7]:
```
c = b[1:,1:]
```

Predict:
- size: [2, 2]
- offset: 4
- strides: (3, 1)

Check:

[8]:
```
print(c)
print("size:", c.size())
print("offset:", c.storage_offset())
print("strides:", c.stride())
```

```
tensor([[4, 5],
        [7, 8]])
size: torch.Size([2, 2])
offset: 4
strides: (3, 1)
```

2 Chapter 3

2.1 Take several pictures of red, blue, and green items with your phone or other digital camera, or download some from the internet if a camera isn't available.

Pictures (download from the internet):

[9]:
```
from PIL import Image
import matplotlib.pyplot as plt

red_img = Image.open("../data/Ferrari.jpg")
blue_img = Image.open("../data/blueberry.jpg")
green_img = Image.open("../data/frog.jpg")
```

[10]:
```
red_img
```

[10]:

[11]:
```
blue_img
```

[11]:

[12]:
```
green_img
```

[12]:

2.1.1 Load each image, and convert it to a tensor.

[13]:
```
import torchvision.transforms as transforms

red_tensor = transforms.ToTensor()(red_img).to(torch.device("cuda"))
red_tensor
```

[13]:
```
tensor([[[0.7176, 0.7176, 0.7176,  …, 0.7529, 0.7490, 0.7490],
         [0.7176, 0.7176, 0.7176,  …, 0.7529, 0.7529, 0.7490],
         [0.7176, 0.7176, 0.7176,  …, 0.7569, 0.7529, 0.7529],
         …,
         [0.6863, 0.4275, 0.5333,  …, 0.7882, 0.7490, 0.7137],
         [0.5451, 0.4392, 0.4039,  …, 0.7569, 0.7059, 0.6627],
         [0.3765, 0.5216, 0.3882,  …, 0.7059, 0.6392, 0.5882]],

        [[0.7059, 0.7059, 0.7059,  …, 0.7373, 0.7333, 0.7333],
         [0.7059, 0.7059, 0.7059,  …, 0.7373, 0.7373, 0.7333],
         [0.7059, 0.7059, 0.7059,  …, 0.7412, 0.7373, 0.7373],
         …,
         [0.6863, 0.4275, 0.5333,  …, 0.7765, 0.7373, 0.7020],
         [0.5451, 0.4392, 0.4039,  …, 0.7451, 0.6941, 0.6510],
```

```
          [0.3765, 0.5216, 0.3882,  …, 0.6941, 0.6275, 0.5765]],

         [[0.6784, 0.6784, 0.6784,  …, 0.7255, 0.7216, 0.7216],
          [0.6784, 0.6784, 0.6784,  …, 0.7255, 0.7255, 0.7216],
          [0.6784, 0.6784, 0.6784,  …, 0.7294, 0.7255, 0.7255],
          …,
          [0.6863, 0.4275, 0.5333,  …, 0.7490, 0.7098, 0.6745],
          [0.5451, 0.4392, 0.4039,  …, 0.7176, 0.6667, 0.6235],
          [0.3765, 0.5216, 0.3882,  …, 0.6667, 0.6000, 0.5490]]],
        device='cuda:0')
```

[14]:
```
red_tensor.shape
```

[14]:
```
torch.Size([3, 1080, 1920])
```

[15]:
```
blue_tensor = transforms.ToTensor()(blue_img).to(torch.device("cuda"))
blue_tensor
```

[15]:
```
tensor([[[0.5137, 0.5098, 0.5020,  …, 0.6314, 0.5922, 0.5451],
         [0.5137, 0.5098, 0.5059,  …, 0.6431, 0.6078, 0.5569],
         [0.5176, 0.5137, 0.5098,  …, 0.6549, 0.6078, 0.5686],
         …,
         [0.1882, 0.2196, 0.2118,  …, 0.1804, 0.1725, 0.1490],
         [0.2196, 0.1608, 0.0745,  …, 0.1843, 0.1882, 0.1725],
         [0.1686, 0.1137, 0.0275,  …, 0.1882, 0.2078, 0.2039]],

        [[0.6510, 0.6471, 0.6392,  …, 0.7647, 0.7373, 0.7059],
         [0.6510, 0.6471, 0.6431,  …, 0.7686, 0.7490, 0.7137],
         [0.6510, 0.6471, 0.6431,  …, 0.7765, 0.7490, 0.7137],
         …,
         [0.3961, 0.3569, 0.2863,  …, 0.4000, 0.4078, 0.4000],
         [0.3725, 0.3294, 0.2627,  …, 0.4039, 0.4235, 0.4235],
         [0.3216, 0.2824, 0.2157,  …, 0.4078, 0.4431, 0.4549]],

        [[0.8784, 0.8745, 0.8667,  …, 0.9529, 0.9412, 0.9098],
         [0.8784, 0.8745, 0.8706,  …, 0.9529, 0.9451, 0.9176],
         [0.8902, 0.8863, 0.8824,  …, 0.9608, 0.9451, 0.9216],
         …,
         [0.6000, 0.5765, 0.5137,  …, 0.6941, 0.6941, 0.6824],
         [0.5961, 0.5373, 0.4510,  …, 0.6980, 0.7098, 0.7059],
         [0.5451, 0.4902, 0.4039,  …, 0.7020, 0.7294, 0.7373]]],
       device='cuda:0')
```

[16]:
```
blue_tensor.shape
```

[16]:
```
torch.Size([3, 467, 700])
```

[17]:
```
green_tensor = transforms.ToTensor()(green_img).to(torch.device("cuda"))
green_tensor
```

[17]:
```
tensor([[[0.0196, 0.0196, 0.0196,  …, 0.1490, 0.1412, 0.1647],
         [0.0196, 0.0196, 0.0196,  …, 0.0941, 0.1176, 0.0588],
         [0.0196, 0.0196, 0.0196,  …, 0.1255, 0.1804, 0.1059],
         …,
         [0.1255, 0.1255, 0.1176,  …, 0.2275, 0.2314, 0.2314],
         [0.0980, 0.0941, 0.0902,  …, 0.2314, 0.2314, 0.2353],
         [0.0784, 0.0745, 0.0745,  …, 0.2314, 0.2353, 0.2353]],

        [[0.2000, 0.2000, 0.2000,  …, 0.3490, 0.3412, 0.3647],
         [0.2000, 0.2000, 0.2000,  …, 0.2941, 0.3176, 0.2588],
         [0.2000, 0.2000, 0.2000,  …, 0.3255, 0.3804, 0.3059],
         …,
         [0.4353, 0.4314, 0.4235,  …, 0.4863, 0.4902, 0.4902],
         [0.3961, 0.3882, 0.3843,  …, 0.4902, 0.4902, 0.4941],
         [0.3725, 0.3647, 0.3529,  …, 0.4902, 0.4941, 0.4941]],

        [[0.0078, 0.0078, 0.0078,  …, 0.0392, 0.0314, 0.0549],
         [0.0078, 0.0078, 0.0078,  …, 0.0000, 0.0078, 0.0000],
         [0.0078, 0.0078, 0.0078,  …, 0.0157, 0.0706, 0.0000],
         …,
         [0.0000, 0.0000, 0.0000,  …, 0.0000, 0.0039, 0.0039],
         [0.0000, 0.0000, 0.0000,  …, 0.0039, 0.0039, 0.0078],
         [0.0000, 0.0000, 0.0000,  …, 0.0039, 0.0078, 0.0078]]],
       device='cuda:0')
```

[18]:
```
green_tensor.shape
```

[18]:
```
torch.Size([3, 1464, 2200])
```

2.1.2 For each image tensor, use the `.mean()` method to get a sense of how bright the image is.

[19]:
```
red_tensor.mean()
```

[19]:
```
tensor(0.5535, device='cuda:0')
```

[20]:
```
blue_tensor.mean()
```

[20]:
```
tensor(0.4189, device='cuda:0')
```

[21]:
```
green_tensor.mean()
```

[21]:
```
tensor(0.2255, device='cuda:0')
```

2.1.3 Now take the mean of each channel of your images. Can you identify the red, green, and blue items from only the channel averages?

[22]:
```
red_RGB_mean = red_tensor.mean(dim=[1, 2])
red_RGB_mean
```

[22]:
```
tensor([0.6218, 0.5122, 0.5265], device='cuda:0')
```

[23]:
```
green_RGB_mean = green_tensor.mean(dim=[1, 2])
green_RGB_mean
```

[23]:
```
tensor([0.2180, 0.3940, 0.0643], device='cuda:0')
```

[24]:
```
blue_RGB_mean = blue_tensor.mean(dim=[1, 2])
blue_RGB_mean
```

[24]:
```
tensor([0.2538, 0.3962, 0.6068], device='cuda:0')
```

According to the results above, we can identify the three images just by the channel averages: **the main color in each image has the highest average number.**

2.2 Select a relatively large file containing Python source code.

Use `sample.py` as source file, which is copied from `__init__.py` in `numpy` .

[25]:
```
with open("../data/sample.py", encoding='utf8') as f:
    source_file = f.read()
len(source_file)
```

[25]:
```
80120
```

2.2.1 Build an index of all the words in the source file. (Feel free to make your tokenization as simple or as complex as you like; we suggest starting by replacing `r"[^a-zA-Z0-9_]+"` with spaces.)

[26]:
```
import re

source_file = re.sub(r"[^a-zA-Z0-9_]+", " ", source_file)
len(source_file)
```

[26]:
```
63098
```

Build index for `source_file` :

[27]:
```
def clean_words(input_str: str):
    """ return a set of unique words,
        cleaned of punctuation from a str
    """
    punctuation = '.,;:"!?/-#'
```

```
    word_set = input_str.lower().replace('\n', ' ').split()
    word_set = {word.strip(punctuation) for word in word_set}
    return word_set

def build_word_index(input_str: str):
    """ return dict{word: index} from a str
    """
    word_set = clean_words(input_str)
    word_set = sorted(word_set)
    word_index = {word: i for i, word in enumerate(word_set)}
    return word_index
```

[28]:
```
source_file_index = build_word_index(source_file)
list(source_file_index.items())[:10]
```

[28]:
```
[('0', 0),
 ('0016', 1),
 ('06376', 2),
 ('0x00001100', 3),
 ('0x0001', 4),
 ('1', 5),
 ('10', 6),
 ('109438', 7),
 ('109940', 8),
 ('10x', 9)]
```

2.2.2 Compare your index with the one you made for Pride and Prejudice. Which is larger?

Build index for *Pride and Prejudice* :

[29]:
```
with open("../data/Pride and Prejudice.txt", encoding="utf8") as f:
    Pride_and_Prejudice = f.read()
len(Pride_and_Prejudice)
```

[29]:
```
775741
```

[30]:
```
Pride_and_Prejudice_index = build_word_index(Pride_and_Prejudice)
list(Pride_and_Prejudice_index.items())[:10]
```

[30]:
```
[('', 0),
 ('$5,000)', 1),
 ("'as-is'", 2),
 ('($1', 3),
 ('(801)', 4),
 ('(a)', 5),
 ('(an', 6),
 ('(and', 7),
```

```
 ('(any', 8),
 ('(available', 9)]
```

[31]:
```
len(source_file_index), len(Pride_and_Prejudice_index)
```

[31]:
```
(1720, 8633)
```

As seen above, index for *Pride and Prejudice* is larger.

2.2.3 Create the one-hot encoding for the source code file.

[32]:
```
def build_onehot_tensor(input_str: str):
    """ build one-hot encoding for a str
        and save it in a tensor
    """
    word_set = clean_words(input_str)
    word_index = build_word_index(input_str)
    onehot_tensor = torch.zeros(len(word_set), len(word_index))
    onehot_tensor = onehot_tensor.to(torch.device("cuda"))

    for id, word in enumerate(word_set):
        onehot_tensor[id][word_index[word]] = 1

    return onehot_tensor
```

[33]:
```
source_file_onehot = build_onehot_tensor(source_file)
source_file_onehot, source_file_onehot.shape
```

[33]:
```
(tensor([[0., 0., 0.,  …, 0., 0., 0.],
         [0., 0., 0.,  …, 0., 0., 0.],
         [0., 0., 0.,  …, 0., 0., 0.],
         …,
         [0., 0., 0.,  …, 0., 0., 0.],
         [0., 0., 0.,  …, 0., 0., 0.],
         [0., 0., 0.,  …, 0., 0., 0.]], device='cuda:0'),
 torch.Size([1720, 1720]))
```

2.2.4 What information is lost with this encoding? How does that information compare with what's lost in the Pride and Prejudice encoding?

With this encoding, information like each word/item's **frequency** is lost.

Build one-hot encoding for *Pride and Prejudice* :

[34]:
```
Pride_and_Prejudice_onehot = build_onehot_tensor(Pride_and_Prejudice)
Pride_and_Prejudice_onehot, Pride_and_Prejudice_onehot.shape
```

[34]:
```
(tensor([[1., 0., 0.,  …, 0., 0., 0.],
         [0., 0., 0.,  …, 0., 0., 0.],
```

```
        [0., 0., 0.,  …, 0., 0., 0.],
        …,
        [0., 0., 0.,  …, 0., 0., 0.],
        [0., 0., 0.,  …, 0., 0., 0.],
        [0., 0., 0.,  …, 0., 0., 0.]], device='cuda:0'),
 torch.Size([8633, 8633]))
```

After one-hot encoding, words in the novel become unique and sorted. Therefore, their **frequency** and **context** information is lost.

3 Chapter 4

[35]:
```python
import torch.optim as optim

# dataset preprocess
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]

t_c = torch.tensor(t_c)
t_u = torch.tensor(t_u)

n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)  # train:num : val:num = 8:2

shuffled_indices = torch.randperm(n_samples)  # shuffle the index
train_indices = shuffled_indices[:-n_val]

val_indices = shuffled_indices[-n_val:]

train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]

val_t_u = t_u[val_indices]
val_t_c = t_c[val_indices]

train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u


# model architecture
params1 = torch.tensor([1.0, 0.0], requires_grad=True)

def model1(t_u, w, b): # linear model
    return w * t_u + b


# optimizer
```

```
optimizer1 = optim.SGD([params1], lr=1e-2)
```

[36]:
```
# loss function
def loss_fn(t_p, t_c):  # don't need to change
    squard_diffs = (t_p - t_c) ** 2
    return squard_diffs.mean()


# train
def training_loop(n_epochs, optimizer, params,
                  train_t_u, val_t_u,
                  train_t_c, val_t_c,
                  model):

    train_loss_list = []
    val_loss_list = []

    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        with torch.no_grad():
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print(f'Epoch {epoch}, '
                  f'Training loss {float(train_loss):.8f}, '
                  f'Validation loss {float(val_loss):.8f}')

        train_loss_list.append(train_loss.item())
        val_loss_list.append(val_loss.item())

    return train_loss_list, val_loss_list


train_loss_list1, val_loss_list1 = training_loop(
    n_epochs=1000,
    optimizer=optimizer1,
    params=params1,
    train_t_u=train_t_un,
    val_t_u=val_t_un,
```

```
    train_t_c=train_t_c,
    val_t_c=val_t_c,
    model=model1
)
```

```
Epoch 1, Training loss 96.99889374, Validation loss 5.50884914
Epoch 2, Training loss 40.26292038, Validation loss 6.54995728
Epoch 3, Training loss 32.46815491, Validation loss 17.15825653
Epoch 500, Training loss 6.53861380, Validation loss 11.47949314
Epoch 1000, Training loss 3.07114291, Validation loss 7.61611652
```

3.1 Redefine the model to be $w_2 {t_u}^2 + w_1 t_u + b$.

[37]:
```
def model2(t_u, w_1, w_2, b):
    return w_2 * t_u ** 2 + w_1 * t_u + b
```

3.1.1 What parts of the training loop and so on must be changed to accommodate this redefinition? What parts are agnostic to swapping out the model?

Parts to be changed

1. `params` when training the model in each epoch, as the new model takes w_1 , w_2 and b .

[38]:
```
# model architecture
params2 = torch.tensor([1.0, 1.0, 0.0], requires_grad=True)
```

2. `optimizer` need be be changed correspondingly. Here, **learning rate is set to be lower** as the model changed.

   I used the same learning rate `lr=1e-2` at first, leading to exploding gradients and even NaN losses after epochs of training.

   This may blame to the rapid increase of gradients caused by the squard term ${t_u}^2$.

   After trying, `lr=1e-6` seems to be a reasonable learning rate here.

[39]:
```
# optimizer
optimizer2 = optim.SGD([params2], lr=1e-6)
```

[40]:
```
train_loss_list2, val_loss_list2 = training_loop(
    n_epochs=1000,
    optimizer=optimizer2,
    params=params2,
    train_t_u=train_t_un,
    val_t_u=val_t_un,
    train_t_c=train_t_c,
    val_t_c=val_t_c,
    model=model2
)
```

```
Epoch 1, Training loss 722.80590820, Validation loss 464.24230957
Epoch 2, Training loss 719.11248779, Validation loss 462.24468994
Epoch 3, Training loss 715.43817139, Validation loss 460.25659180
Epoch 500, Training loss 66.01004028, Validation loss 74.00518799
Epoch 1000, Training loss 17.45697021, Validation loss 26.06868744
```

Besides, the `loss_fn()` and `training_loop()` are agnostic to swapping out the model, remaining the same.

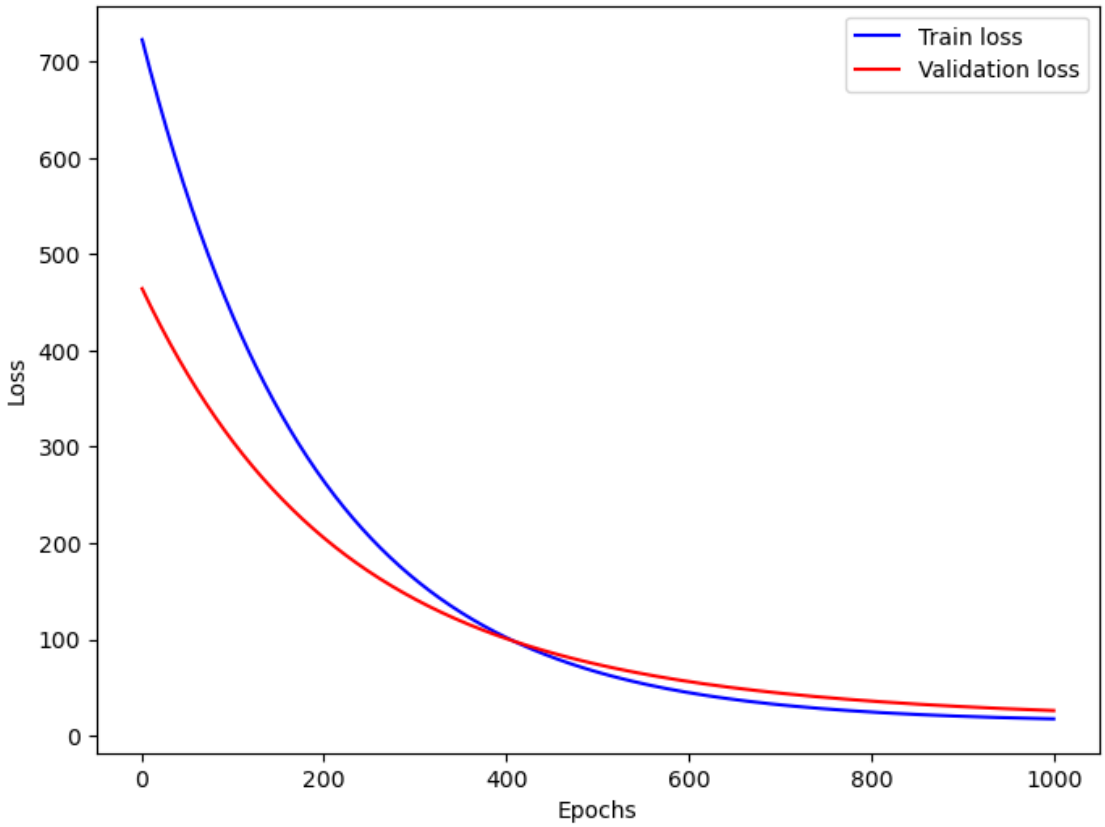
3.1.2 Is the resulting loss higher or lower after training? Is the result better or worse?

As seen above, the resulting loss is lower and the result is better after training.

3.1.3 Draw the relationship between the epoch and train/validation loss in one picture by matpolotlib.

[41]:
```python
import matplotlib.pyplot as plt

epoches = list(range(1, 1001))
plt.figure(figsize=(8, 6))
plt.plot(epoches, train_loss_list2, label="Train loss", color='blue')
plt.plot(epoches, val_loss_list2, label="Validation loss", color='red')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



[42]:
```
params2
```

[42]:
```
tensor([ 0.8904,  0.3258, -0.0197], requires_grad=True)
```

[43]:
```
params2[2]
```

[43]:
```
tensor(-0.0197, grad_fn=<SelectBackward0>)
```

[44]:
```
model2(35.7, params2[0], params2[1], params2[2])
```

[44]:
```
tensor(446.9565, grad_fn=<AddBackward0>)
```

4 Chapter 5

[45]:
```
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]

t_c = torch.tensor(t_c).unsqueeze(1)
t_u = torch.tensor(t_u).unsqueeze(1)
```

```
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)

shuffled_ind = torch.randperm(n_samples)

train_ind = shuffled_ind[:-n_val]
val_ind = shuffled_ind[-n_val:]

train_t_u = t_u[train_ind]
train_t_c = t_c[train_ind]

val_t_u = t_u[val_ind]
val_t_c = t_c[val_ind]

train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u
```

4.1 Experiment with the number of hidden neurons in your simple neural network model, as well as the learning rate.

Design the model:

[46]:
```
import torch.nn as nn
```

[47]:
```
class MyModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_linear = nn.Linear(input_size, hidden_size)
        self.hidden_activation = nn.Tanh()
        self.output_linear = nn.Linear(hidden_size, output_size)

    def forward(self, input):
        hidden_t = self.hidden_linear(input)
        activated_t = self.hidden_activation(hidden_t)
        output_t = self.output_linear(activated_t)
        return output_t
```

[48]:
```
def training_nn_loop(n_epochs, optimizer, loss_fn,
                     train_input, val_input,
                     train_output, val_output,
                     model):
    for epoch in range(1, n_epochs + 1):
        model.train()
        optimizer.zero_grad()

        train_pred = model(train_input)
        train_loss = loss_fn(train_pred, train_output)
```

```
        model.eval()
        with torch.no_grad():
            val_pred = model(val_input)
            val_loss = loss_fn(val_pred, val_output)
            # assert val_loss.requires_grad == False

        train_loss.backward()
        optimizer.step()

        if epoch == 1 or epoch % 1000 == 0:
            print(f'Epoch {epoch}, '
                  f'Training loss {float(train_loss):.8f}, '
                  f'Validation loss {float(val_loss):.8f}')
```

Test with different number of hidden neurons and learning rate:

[49]:
```
def test_training(hidden_size, learning_rate, n_epochs=5000):
    nn_model = MyModel(1, hidden_size, 1)
    training_nn_loop(
        n_epochs=n_epochs,
        optimizer=optim.SGD(nn_model.parameters(), lr=learning_rate),
        loss_fn=nn.MSELoss(),
        train_input=train_t_un,
        val_input=val_t_un,
        train_output=train_t_c,
        val_output=val_t_c,
        model=nn_model
    )
```

[50]:
```
test_training(5, 1e-2)
```

```
Epoch 1, Training loss 209.46965027, Validation loss 76.94982147
Epoch 1000, Training loss 8.12751293, Validation loss 20.63431358
Epoch 2000, Training loss 18.91850471, Validation loss 17.84290504
Epoch 3000, Training loss 19.23334503, Validation loss 20.71952057
Epoch 4000, Training loss 18.73767471, Validation loss 16.39104271
Epoch 5000, Training loss 19.08134079, Validation loss 17.82775688
```

[51]:
```
test_training(5, 1e-3)
```

```
Epoch 1, Training loss 201.50762939, Validation loss 71.26271057
Epoch 1000, Training loss 6.06451750, Validation loss 8.98670769
Epoch 2000, Training loss 5.17308617, Validation loss 16.69307899
Epoch 3000, Training loss 2.70810795, Validation loss 8.55501461
Epoch 4000, Training loss 1.89315844, Validation loss 5.32746792
Epoch 5000, Training loss 1.69002330, Validation loss 3.94300413
```

[52]:
```
test_training(1, 1e-3)
```

```
Epoch 1, Training loss 201.59486389, Validation loss 70.78569031
Epoch 1000, Training loss 20.71945572, Validation loss 7.96618080
Epoch 2000, Training loss 11.69736671, Validation loss 0.25762042
Epoch 3000, Training loss 11.41577053, Validation loss 1.22996652
Epoch 4000, Training loss 12.91863441, Validation loss 3.13597822
Epoch 5000, Training loss 12.98825073, Validation loss 3.65429354
```

[53]:
```
nn_model = MyModel(1, 5, 1)
training_nn_loop(
    n_epochs=5000,
    optimizer=optim.SGD(nn_model.parameters(), lr=1e-3),
    loss_fn=nn.MSELoss(),
    train_input=train_t_un,
    val_input=val_t_un,
    train_output=train_t_c,
    val_output=val_t_c,
    model=nn_model
)
```

```
Epoch 1, Training loss 221.25567627, Validation loss 86.03329468
Epoch 1000, Training loss 6.13190603, Validation loss 8.03980637
Epoch 2000, Training loss 4.67741060, Validation loss 0.56954968
Epoch 3000, Training loss 2.65848184, Validation loss 1.24144673
Epoch 4000, Training loss 1.93740737, Validation loss 2.22630501
Epoch 5000, Training loss 1.73304272, Validation loss 3.32154417
```

4.1.1 What changes result in a more linear output from the model?

Lower number of hidden neurons results in a more linear output from the model. A larger hidden size allows the model to learn more complex patterns, therefore, lower number of hidden neurons makes it less complex and more linear.

4.1.2 Can you get the model to obviously overfit the data?

We can try to achieve that by increasing the number of hidden neurons and training epochs:

[55]:
```
test_training(40, 1e-2, 20000)
```

```
Epoch 1, Training loss 195.33477783, Validation loss 66.43927765
Epoch 1000, Training loss 11.51179790, Validation loss 39.22739410
Epoch 2000, Training loss 2.19814205, Validation loss 9.12819672
Epoch 3000, Training loss 1.64178514, Validation loss 6.60537243
Epoch 4000, Training loss 1.51837003, Validation loss 6.11152458
Epoch 5000, Training loss 1.45395446, Validation loss 6.02271938
Epoch 6000, Training loss 1.39768744, Validation loss 5.96009684
Epoch 7000, Training loss 1.37130046, Validation loss 6.13231945
Epoch 8000, Training loss 1.36864769, Validation loss 6.45567322
Epoch 9000, Training loss 1.33371961, Validation loss 6.47659683
Epoch 10000, Training loss 1.29789114, Validation loss 6.45613384
```

```
Epoch 11000, Training loss 1.26881409, Validation loss 6.45955563
Epoch 12000, Training loss 1.24642062, Validation loss 6.49491739
Epoch 13000, Training loss 1.23037851, Validation loss 6.56604147
Epoch 14000, Training loss 1.22062266, Validation loss 6.68006468
Epoch 15000, Training loss 1.21554053, Validation loss 6.83614349
Epoch 16000, Training loss 1.21104574, Validation loss 7.00807333
Epoch 17000, Training loss 1.20445287, Validation loss 7.16288948
Epoch 18000, Training loss 1.19556212, Validation loss 7.28522825
Epoch 19000, Training loss 1.18512666, Validation loss 7.37604332
Epoch 20000, Training loss 1.17400908, Validation loss 7.44287300
```

As seen above, with a large hidden size (40) and training epochs (20000), the model's training loss reaches a quite low level. While it still shows a trend to decrease, the validation is obviously higher and tend to increase, indicating the overfit of data.

4.2 The third-hardest problem in physics is finding a proper wine to celebrate discoveries. Load the wine data from chapter 3 and create a new model with the appropriate number of input parameters.

[56]:
```
import pandas as pd
```

[57]:
```
wine_df = pd.read_csv("../data/winequality-white.csv", sep=';')
```

Data preparation:

[58]:
```
wine_tensor = torch.tensor(wine_df.to_numpy())
wine_tensor
```

[58]:
```
tensor([[ 7.0000,  0.2700,  0.3600,  …,  0.4500,  8.8000,  6.0000],
        [ 6.3000,  0.3000,  0.3400,  …,  0.4900,  9.5000,  6.0000],
        [ 8.1000,  0.2800,  0.4000,  …,  0.4400, 10.1000,  6.0000],
        …,
        [ 6.5000,  0.2400,  0.1900,  …,  0.4600,  9.4000,  6.0000],
        [ 5.5000,  0.2900,  0.3000,  …,  0.3800, 12.8000,  7.0000],
        [ 6.0000,  0.2100,  0.3800,  …,  0.3200, 11.8000,  6.0000]],
       dtype=torch.float64)
```

[59]:
```
wine_features = wine_tensor[:, :-1]
wine_quality = wine_tensor[:, -1]

wine_f = wine_features.float()
wine_q = wine_quality.long()

# Normalize the featuress
mean_val = wine_f.mean(dim=0)
std_val = wine_f.std(dim=0)
wine_fn = (wine_f - mean_val) / std_val
```

```
# Divide the training set and the validation set
n_samples = wine_fn.shape[0]
n_val = int(0.2 * n_samples) # training : validation = 8 : 2
shuffled_ind = torch.randperm(n_samples)

train_ind = shuffled_ind[:-n_val]
val_ind = shuffled_ind[-n_val]

wine_train_f = wine_fn[train_ind]
wine_train_q = wine_q[train_ind]

wine_val_f = wine_fn[val_ind]
wine_val_q = wine_q[val_ind]
```

Train the model:

[60]:
```
wine_model = MyModel(wine_f.shape[1], 10, wine_q.max() + 1)

training_nn_loop(
    n_epochs=10000,
    optimizer=optim.SGD(wine_model.parameters(), lr=1e-2),
    loss_fn=nn.CrossEntropyLoss(), # for label classification task
    train_input=wine_train_f,
    train_output=wine_train_q,
    val_input=wine_val_f,
    val_output=wine_val_q,
    model=wine_model
)
```

```
Epoch 1, Training loss 2.47133517, Validation loss 2.14867425
Epoch 1000, Training loss 1.22653985, Validation loss 1.02317214
Epoch 2000, Training loss 1.14560723, Validation loss 1.01812577
Epoch 3000, Training loss 1.11532116, Validation loss 1.00699139
Epoch 4000, Training loss 1.09880638, Validation loss 0.98274595
Epoch 5000, Training loss 1.08817911, Validation loss 0.95575792
Epoch 6000, Training loss 1.08031094, Validation loss 0.93403524
Epoch 7000, Training loss 1.07387221, Validation loss 0.92149884
Epoch 8000, Training loss 1.06836236, Validation loss 0.91774428
Epoch 9000, Training loss 1.06348538, Validation loss 0.91898346
Epoch 10000, Training loss 1.05908632, Validation loss 0.92282122
```

4.2.1 Can you explain what factors contribute to the training times?

Model complexity, size and shape of data, learning rate, hardcore, etc.

4.2.2 Can you get the loss to decrease while training on this data set?

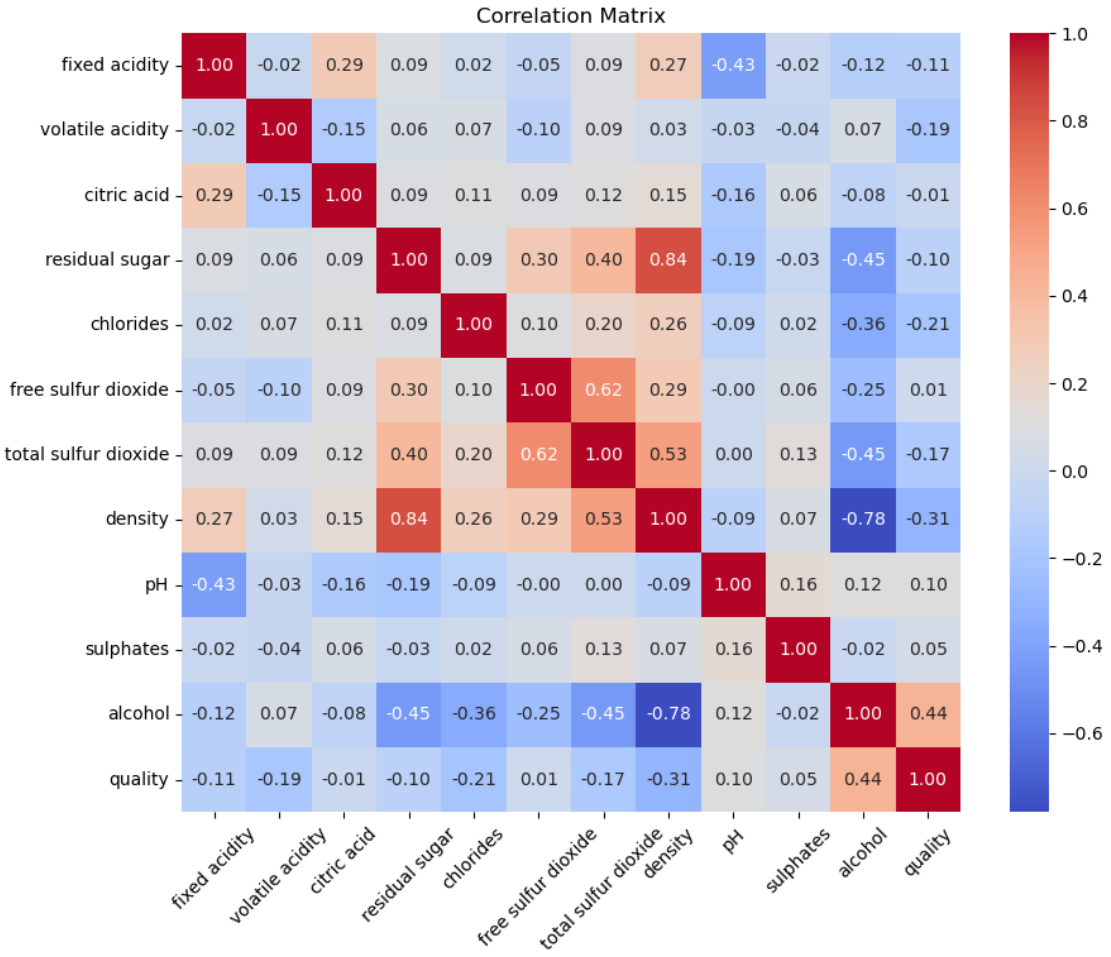
Yes, as seen above, both training loss and validation loss decrease while training.

4.2.3 How would you go about graphing this data set?

[61]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Calculate the correlation matrix
corr = wine_df.corr()

# Create a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm', square=True)
plt.xticks(rotation=45)
plt.title('Correlation Matrix')
plt.show()
```



[62]:
```
# Box plot for each feature against quality
features = wine_df.columns[:-1]
```

```
plt.figure(figsize=(15, 10))
for i, feature in enumerate(features):
    plt.subplot(4, 3, i + 1)
    sns.boxplot(x='quality', y=feature, data=wine_df)

plt.tight_layout()
plt.show()
```