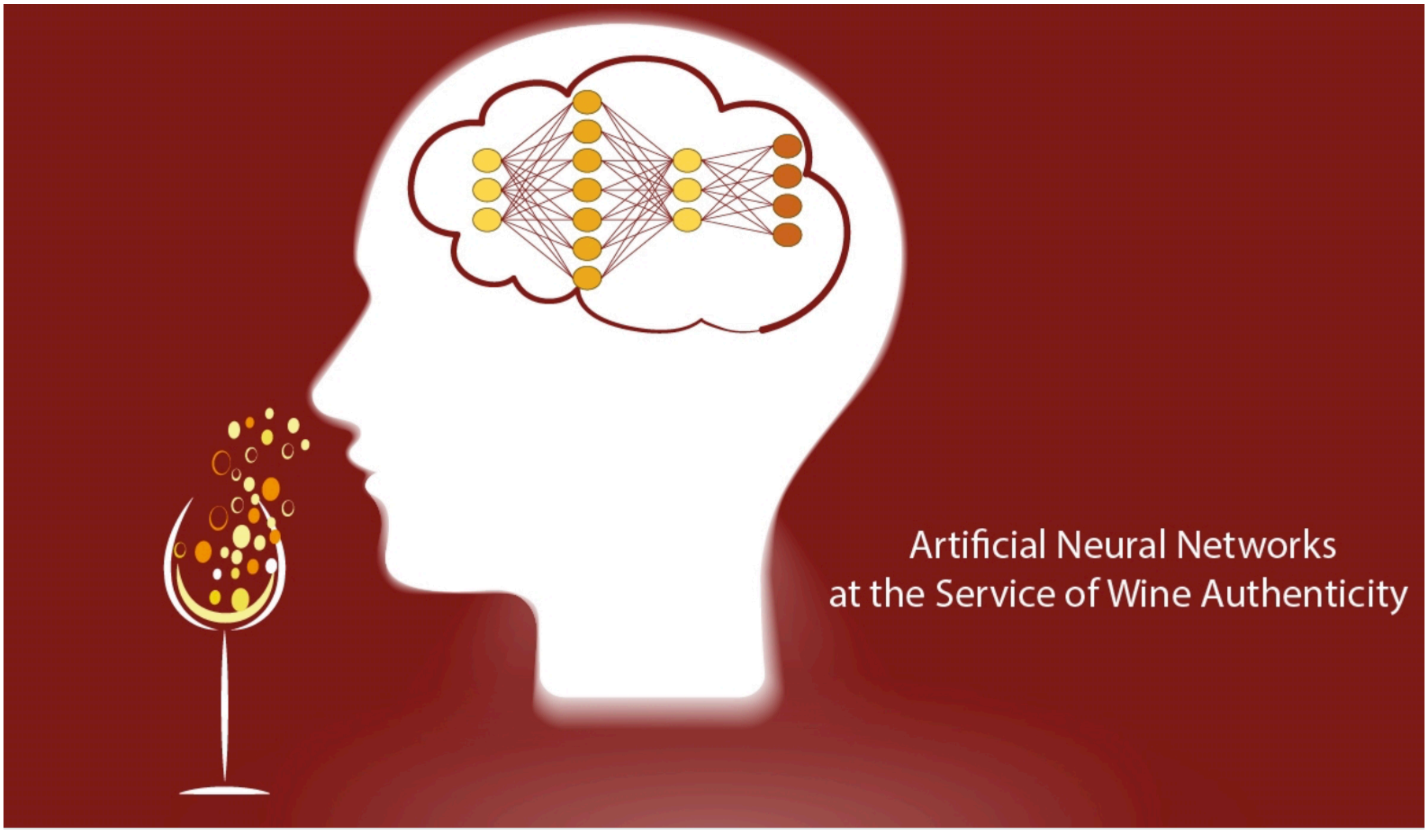
Train a Neural Network to Predict Quality of Wine



- In this lab, you will first train a neural network on a public dataset, then make several enhancements to the lab.
- Tasks breakdown:
    - Code running: 10%
    - Enhancement 1: 15%
    - Enhancement 2: 15%
    - Enhancement 3: 10%
    - Enhancement 4: 10%
    - Enhancement 5: 40%

Imports

```
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import AdamW
from torch.utils.data import Dataset, DataLoader
from tqdm.notebook import tqdm
```

Dataset

```
data_df = pd.read_csv('/winequality-red.csv')
```

```
data_df.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

Next steps: Generate code with data_df | View recommended plots | New interactive sheet

```
# how many features?
len(data_df.columns) - 1
```

```
11
```

```
# how many labels? If yours is a binary classification task, then you'll have 2 labels.
data_df.quality.unique()
```

```
array([5, 6, 7, 4, 8, 3])
```

```
# convert these quaity measures to labels (0 to 5)
def get_label(quality):
    if quality == 3:
        return 0
    elif quality == 4:
        return 1
    elif quality == 5:
        return 2
    elif quality == 6:
        return 3
    elif quality == 7:
        return 4
    else:
        return 5

labels = data_df['quality'].apply(get_label)

# normalize data
data_df = (data_df - data_df.mean()) / data_df.std()
data_df['label'] = labels


data_df.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	-0.528194	0.961576	-1.391037	-0.453077	-0.243630	-0.466047	-0.379014	0.558100	1.288240	-0.579025	-0.959946	-0.787576
1	-0.298454	1.966827	-1.391037	0.043403	0.223805	0.872365	0.624168	0.028252	-0.719708	0.128910	-0.584594	-0.787576
2	-0.298454	1.296660	-1.185699	-0.169374	0.096323	-0.083643	0.228975	0.134222	-0.331073	-0.048074	-0.584594	-0.787576
3	1.654339	-1.384011	1.483689	-0.453077	-0.264878	0.107558	0.411372	0.664069	-0.978798	-0.461036	-0.584594	0.450707
4	-0.528194	0.961576	-1.391037	-0.453077	-0.243630	-0.466047	-0.379014	0.558100	1.288240	-0.579025	-0.959946	-0.787576

Next steps: Generate code with data_df | View recommended plots | New interactive sheet

```
# sumamry statistics of the data
data_df.describe()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	
count	1.599000e+03	1.599000e+03	1.599000e+03	1.599000e+03	1.599000e+03	1.599000e+03	1.599000e+03	1.599000e+03	1.5990
mean	4.088176e-16	1.599721e-16	-8.887339e-17	-1.155354e-16	2.132961e-16	-4.443669e-17	3.554936e-17	-3.466062e-14	2.8794
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.0000
min	-2.136377e+00	-2.277567e+00	-1.391037e+00	-1.162333e+00	-1.603443e+00	-1.422055e+00	-1.230199e+00	-3.537625e+00	-3.6992
25%	-7.004996e-01	-7.696903e-01	-9.290275e-01	-4.530767e-01	-3.711129e-01	-8.484502e-01	-7.438076e-01	-6.075656e-01	-6.5493
50%	-2.410190e-01	-4.367545e-02	-5.634264e-02	-2.402999e-01	-1.798892e-01	-1.792441e-01	-2.574163e-01	1.759533e-03	-7.2104
75%	5.056370e-01	6.264921e-01	7.650078e-01	4.340257e-02	5.382858e-02	4.899619e-01	4.721707e-01	5.766445e-01	5.7574
max	4.353787e+00	5.876138e+00	3.742403e+00	9.192806e+00	1.112355e+01	5.365606e+00	7.372847e+00	3.678904e+00	4.5268

Load this dataset for training a neural network

```
# The dataset class
class WineDataset(Dataset):

    def __init__(self, data_df):
        self.data_df = data_df
        self.features = []
        self.labels = []
        for _, i in data_df.iterrows():
          self.features.append([i['fixed acidity'], i['volatile acidity'], i['citric acid'], i['residual sugar'], i['chl
          self.labels.append(i['label'])

    def __len__(self):
        return len(self.data_df)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        features = self.features[idx]
        features = torch.FloatTensor(features)

        labels = torch.tensor(self.labels[idx], dtype = torch.long)

        return {'labels': labels, 'features': features}

wine_dataset = WineDataset(data_df)
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(wine_dataset, [0.8, 0.1, 0.1])

# The dataloader
train_dataloader = DataLoader(train_dataset, batch_size = 4, shuffle = True, num_workers = 0)
val_dataloader = DataLoader(val_dataset, batch_size = 4, shuffle = False, num_workers = 0)
test_dataloader = DataLoader(test_dataset, batch_size = 4, shuffle = False, num_workers = 0)
```

```
# peak into the dataset
for i in wine_dataset:
  print(i)
  break
```

```
{'labels': tensor(2), 'features': tensor([-0.5282,  0.9616, -1.3910, -0.4531, -0.2436, -0.4660, -0.3790,  0.5581,
         1.2882, -0.5790, -0.9599])}
```

Neural Network

```
# change the device to gpu if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
class WineModel(torch.nn.Module):

    def __init__(self):
        super(WineModel, self).__init__()

        self.linear1 = torch.nn.Linear(11, 1000)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(1000, 6)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x

winemodel = WineModel().to(device)
```

Training

```
# Define and the loss function and optimizer
criterion = nn.CrossEntropyLoss().to(device) #mse
optimizer = AdamW(winemodel.parameters(), lr = 1e-3) #gradient decent


# Lets define the training steps
def accuracy(preds, labels):
    preds = torch.argmax(preds, dim=1).flatten()
    labels = labels.flatten()
    return torch.sum(preds == labels) / len(labels)

def train(model, data_loader, optimizer, criterion):
  epoch_loss = 0
  epoch_acc = 0

  model.train()
  for d in tqdm(data_loader):
    inputs = d['features'].to(device)
    labels = d['labels'].to(device)
    outputs = winemodel(inputs)

    _, preds = torch.max(outputs, dim=1)
    loss = criterion(outputs, labels)
    acc = accuracy(outputs, labels)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    epoch_loss += loss.item()
    epoch_acc += acc.item()

  return epoch_loss / len(data_loader), epoch_acc / len(data_loader)

# Lets define the testing steps
def evaluate(model, data_loader, criterion):
    epoch_loss = 0
    epoch_acc = 0

    model.eval()
    with torch.no_grad():
      for d in data_loader:
        inputs = d['features'].to(device)
        labels = d['labels'].to(device)
        outputs = winemodel(inputs)

        _, preds = torch.max(outputs, dim=1)
        loss = criterion(outputs, labels)
```

```
        acc = accuracy(outputs, labels)

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(data_loader), epoch_acc / len(data_loader)
```

```
# Let's train our model
for epoch in range(100):
    train_loss, train_acc = train(winemodel, train_dataloader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(winemodel, val_dataloader, criterion)

    print(f'| Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}% | Val. Loss: {valid_l
```

Lab Enhancements

- These tasks are additional enhancements with less guidance.
- Report results means give us the accuracy, precision, recall and F1-score.

Enhancement 1: The current code does not actually evaluate the model on the test set, but it only evaluates it on the val set. When you write papers, you would ideally split the dataset into train, val and test. Train and val are both used in training, and the model trained on the training data, and evaluated on the val data. So why do we need test split? We report our results on the test split in papers. Also, we do cross-validation on the train/val split (covered in later labs).

Report the results of the model on the test split. (Hint: It would be exactly like the evaluation on the val dataset, except it would be done on the test dataset.)

```
test_loss, test_acc = evaluate(winemodel, test_dataloader, criterion)
print(f'| Test. Loss: {test_loss:.3f} | Test. Acc: {test_acc*100:.2f}% |')
```

10 epochs

test_loss = 1.466

test_acc = 59.58

We want to see how the model performs on data that it hasn't seen before.

Enhancement 2: Increase the number of epochs (and maybe the learning rate). Does the accuracy on the test set increase? Is there a significant difference between the test accuracy and the train accuracy? If yes, why?

Epochs = 623

test_loss = 1.453

test_acc = 57.71

The test accuracy decreased indicating the model has has likely overfit. There is also a significant difference between the test and training accuracy because the model has overfit to the training set.

Enhancement 3: Increase the depth of your model (add more layers). Report the parts of the model definition you had to update. Report results.

Hidden layer of 200 added.

Epochs = 100

test_loss = 1.439

test_acc = 60.00

Enhancement 4: Increase the width of your model's layers. Report the parts of the model definition you had to update. Report results.

I got rid of the additional deep layer and changed the hidden layer from 200 to 1000.

Epochs = 100

test_loss = 1.463

test_acc = 57.71

Enhancement 5: Choose a new dataset from the list below. Search the Internet and download your chosen dataset (many of them could be available on kaggle). Adapt your model to your dataset. Train your model and record your results.

- cancer_dataset - Breast cancer dataset.
- crab_dataset - Crab gender dataset.
- glass_dataset - Glass chemical dataset.
- iris_dataset - Iris flower dataset.
- ovarian_dataset - Ovarian cancer dataset.
- thyroid_dataset - Thyroid function dataset.

```
glass = pd.read_csv("/glass.csv")
```

```
glass.head()
```

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	1
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1

Next steps: Generate code with glass View recommended plots New interactive sheet

```
#Tells us that there are 9 features
len(glass.columns) - 1
```

9

```
# how many types of glass?
glass.Type.unique()
```

```
array([1, 2, 3, 5, 6, 7])
```

```
# convert these type measures to labels (0 to 5)
def get_label(Type):
    if Type == 1:
        return 0
    elif Type == 2:
        return 1
    elif Type == 3:
        return 2
    elif Type == 5:
        return 3
    elif Type == 6:
        return 4
    else:
        return 5

labels = glass['Type'].apply(get_label)

# normalize data
glass = (glass - glass.mean()) / glass.std()
glass['label'] = labels
```

```
glass.head()
```

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type	label
0	0.870826	0.284287	1.251704	-0.690822	-1.124446	-0.670134	-0.145425	-0.352051	-0.585079	-0.84629	0
1	-0.248750	0.590433	0.634680	-0.170061	0.102080	-0.026152	-0.791877	-0.352051	-0.585079	-0.84629	0
2	-0.719631	0.149582	0.600016	0.190465	0.437760	-0.164148	-0.827010	-0.352051	-0.585079	-0.84629	0
3	-0.232286	-0.242285	0.697076	-0.310266	-0.052850	0.111844	-0.517838	-0.352051	-0.585079	-0.84629	0
4	-0.311315	-0.168810	0.648546	-0.410413	0.553957	0.081178	-0.623237	-0.352051	-0.585079	-0.84629	0

Next steps: Generate code with glass | View recommended plots | New interactive sheet

```
glass.describe()
```

	RI	Na	Mg	Al	Si	K	Ca	Ba	
count	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000
mean	-2.870393e-14	2.158191e-15	-1.328117e-16	-2.988264e-16	9.504339e-16	5.395476e-17	-2.988264e-16	-6.640586e-17	-4.7729
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000
min	-2.375945e+00	-3.279254e+00	-1.861147e+00	-2.313192e+00	-3.667872e+00	-7.621317e-01	-2.478273e+00	-3.520514e-01	-5.8507
25%	-6.068499e-01	-6.127214e-01	-3.948486e-01	-5.105589e-01	-4.789059e-01	-5.743035e-01	-5.037845e-01	-3.520514e-01	-5.8507
50%	-2.257001e-01	-1.320720e-01	5.514857e-01	-1.700615e-01	1.795445e-01	8.884491e-02	-2.508251e-01	-3.520514e-01	-5.8507
75%	2.608215e-01	5.108348e-01	6.346799e-01	3.707284e-01	5.636406e-01	1.731759e-01	1.514506e-01	-3.520514e-01	4.412072
max	5.125215e+00	4.864232e+00	1.251704e+00	4.116199e+00	3.562172e+00	8.759606e+00	5.082401e+00	5.983182e+00	4.648981

```
# The dataset class
class GlassDataset(Dataset):

    def __init__(self, data_df):
        self.data_df = data_df
        self.features = []
```

```
        self.labels = []
        for _, i in data_df.iterrows():
          self.features.append([i['RI'], i['Na'], i['Mg'], i['Al'], i['Si'], i['K'], i['Ca'], i['Ba'], i['Fe']])
          self.labels.append(i['label'])

    def __len__(self):
        return len(self.data_df)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        features = self.features[idx]
        features = torch.FloatTensor(features)

        labels = torch.tensor(self.labels[idx], dtype = torch.long)

        return {'labels': labels, 'features': features}

glass_dataset = GlassDataset(glass)
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(glass_dataset, [0.8, 0.1, 0.1])

# The dataloader
train_dataloader = DataLoader(train_dataset, batch_size = 4, shuffle = True, num_workers = 0)
val_dataloader = DataLoader(val_dataset, batch_size = 4, shuffle = False, num_workers = 0)
test_dataloader = DataLoader(test_dataset, batch_size = 4, shuffle = False, num_workers = 0)


# peak into the dataset
for i in glass_dataset:
  print(i)
  break
```

```
{'labels': tensor(0), 'features': tensor([ 0.8708,  0.2843,  1.2517, -0.6908, -1.1244, -0.6701, -0.1454, -0.3521,
        -0.5851])}
```

```
# change the device to gpu if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class GlassModel(torch.nn.Module):

    def __init__(self):
        super(GlassModel, self).__init__()

        self.linear1 = torch.nn.Linear(9, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 6)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x

glassmodel = GlassModel().to(device)


# Define and the loss function and optimizer
criterion = nn.CrossEntropyLoss().to(device) #mse
optimizer = AdamW(glassmodel.parameters(), lr = 1e-3) #gradient decent


# Lets define the training steps
def accuracy(preds, labels):
    preds = torch.argmax(preds, dim=1).flatten()
    labels = labels.flatten()
    return torch.sum(preds == labels) / len(labels)

def train(model, data_loader, optimizer, criterion):
  epoch_loss = 0
```

```
    epoch_acc = 0

    model.train()
    for d in tqdm(data_loader):
      inputs = d['features'].to(device)
      labels = d['labels'].to(device)
      outputs = glassmodel(inputs)

      _, preds = torch.max(outputs, dim=1)
      loss = criterion(outputs, labels)
      acc = accuracy(outputs, labels)

      loss.backward()
      optimizer.step()
      optimizer.zero_grad()

      epoch_loss += loss.item()
      epoch_acc += acc.item()

    return epoch_loss / len(data_loader), epoch_acc / len(data_loader)

  # Lets define the testing steps
  def evaluate(model, data_loader, criterion):
      epoch_loss = 0
      epoch_acc = 0

      model.eval()
      with torch.no_grad():
        for d in data_loader:
          inputs = d['features'].to(device)
          labels = d['labels'].to(device)
          outputs = glassmodel(inputs)

          _, preds = torch.max(outputs, dim=1)
          loss = criterion(outputs, labels)
          acc = accuracy(outputs, labels)

          epoch_loss += loss.item()
          epoch_acc += acc.item()

      return epoch_loss / len(data_loader), epoch_acc / len(data_loader)


  # Let's train our model
  for epoch in range(10):
      train_loss, train_acc = train(glassmodel, train_dataloader, optimizer, criterion)
      valid_loss, valid_acc = evaluate(glassmodel, val_dataloader, criterion)

      print(f'| Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}% | Val. Loss: {valid_l
```

```
100%    43/43 [00:00<00:00, 522.44it/s]
| Epoch: 01 | Train Loss: 1.751 | Train Acc: 41.86% | Val. Loss: 1.656 | Val. Acc: 62.50% |
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1736: UserWarning: Implicit dimension choice for
  return self._call_impl(*args, **kwargs)
100%    43/43 [00:00<00:00, 522.39it/s]
| Epoch: 02 | Train Loss: 1.612 | Train Acc: 56.40% | Val. Loss: 1.471 | Val. Acc: 75.00% |
100%    43/43 [00:00<00:00, 523.05it/s]
| Epoch: 03 | Train Loss: 1.528 | Train Acc: 59.88% | Val. Loss: 1.403 | Val. Acc: 75.00% |
100%    43/43 [00:00<00:00, 550.27it/s]
| Epoch: 04 | Train Loss: 1.494 | Train Acc: 61.63% | Val. Loss: 1.361 | Val. Acc: 75.00% |
100%    43/43 [00:00<00:00, 468.00it/s]
| Epoch: 05 | Train Loss: 1.474 | Train Acc: 61.63% | Val. Loss: 1.341 | Val. Acc: 79.17% |
100%    43/43 [00:00<00:00, 434.97it/s]
| Epoch: 06 | Train Loss: 1.456 | Train Acc: 63.95% | Val. Loss: 1.331 | Val. Acc: 75.00% |
100%    43/43 [00:00<00:00, 377.11it/s]
| Epoch: 07 | Train Loss: 1.434 | Train Acc: 66.28% | Val. Loss: 1.307 | Val. Acc: 79.17% |
100%    43/43 [00:00<00:00, 345.76it/s]
| Epoch: 08 | Train Loss: 1.419 | Train Acc: 66.86% | Val. Loss: 1.304 | Val. Acc: 79.17% |
100%    43/43 [00:00<00:00, 406.05it/s]
| Epoch: 09 | Train Loss: 1.398 | Train Acc: 70.93% | Val. Loss: 1.294 | Val. Acc: 79.17% |
100%    43/43 [00:00<00:00, 418.81it/s]
| Epoch: 10 | Train Loss: 1.378 | Train Acc: 73.26% | Val. Loss: 1.295 | Val. Acc: 79.17% |
```

```
test_loss, test_acc = evaluate(glassmodel, test_dataloader, criterion)
print(f'| Test. Loss: {test_loss:.3f} | Test. Acc: {test_acc*100:.2f}% |')
```

```
| Test. Loss: 1.408 | Test. Acc: 75.00% |
```

For the **first attempt**, I created a NN with one hidden layer of size 200. The learning rate was 1e-3. I trained for 10 epochs.

The results...

```
Training loss and accuracy: 1.344, 73.84%

Validation loss and accuracy: 1.451, 58.33%

Test loss and accuracy: 1.377, 75.00%
```

For the **second attempt**, I created a NN with one hidden layer of size 1000. The learning rate was 1e-3. I trained for 10 epochs. The results...

```
Training loss and accuracy: 1.274, 77.91%

Validation loss and accuracy: 1.437, 58.33%

Test loss and accuracy: 1.354, 70.83%
```

For the **third attempt**, I created a NN with two hidden layers of size 200 and 300. The learning rate was 1e-3. I trained for 200 epochs. The results...

```
Training loss and accuracy: 1.183, 86.05%

Validation loss and accuracy: 1.456, 58.33%

Test loss and accuracy: 1.352, 70.83%
```

For the **fourth attempt**, I created a NN with one hidden layer of size 200. The learning rate was 1e-4. I trained for 10 epochs.

The results...

```
Training loss and accuracy: 1.614, 68.02%

Validation loss and accuracy: 1.600, 58.33%

Test loss and accuracy: 1.663, 66.67%
```

For the **fifth attempt**, I created a NN with one hidden layer of size 200. The learning rate was 1e-2. I trained for 10 epochs.

The results...

```
Training loss and accuracy: 1.273, 77.33%

Validation loss and accuracy: 1.467, 54.17%
```