```
# ## Step 4a: Train the Network
# Our loss function, `NLLLoss`, checks a Negative Log Likelihood (NLL) tensor against a target tensor that specifies which index of the NLL tensor should be highest. Example:
# ✅ NLL tensor: `[-3, -1, -2]`, target: `[1]`
# ❌ NLL tensor: `[-1, -3, -2]`, target: `[1]`
#
# Let's train for **5 epochs**. We will also be performing basic evaluation *alongside* training. This is a good practice because you can then detect **overfitting**, **insufficient learning rate**, and other issues.

# In[7]:


from tqdm import tqdm
import matplotlib.pyplot as plt




LEARNING_RATE = 0.01
criterion = nn.NLLLoss()

train_average_losses = []
test_average_losses = []




for epoch_index in tqdm(range(5)):

    # TRAIN

    train_losses = []

    for step_index, batch in enumerate(train_dataloader):

        input_tensor, expected_output = batch
        hidden_tensor = rnn.initHidden(input_tensor.shape[0])

```

```
# ## Step 4a: Train the Network
# Our loss function, `NLLLoss`, checks a Negative Log Likelihood (NLL) tensor against a target tensor that specifies which index of the NLL tensor should be highest. Example:
# ✅ NLL tensor: `[-3, -1, -2]`, target: `[1]`
# ❌ NLL tensor: `[-1, -3, -2]`, target: `[1]`
#
# Let's train for **5 epochs**. We will also be performing basic evaluation *alongside* training. This is a good practice because you can then detect **overfitting**, **insufficient learning rate**, and other issues.

# In[7]:


from tqdm import tqdm
import matplotlib.pyplot as plt
from torch.optim import SGD
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter(flush_secs=1)

LEARNING_RATE = 0.01
criterion = nn.NLLLoss()

train_average_losses = []
test_average_losses = []

optimizer = SGD(rnn.parameters(), lr=LEARNING_RATE)

for epoch_index in tqdm(range(5)):

    # TRAIN

    train_losses = []

    for step_index, batch in enumerate(train_dataloader):

        input_tensor, expected_output = batch
        hidden_tensor = rnn.initHidden(input_tensor.shape[0])

```

```
        rnn.zero_grad()

        for i in range(input_tensor.shape[1]):
            if len(torch.nonzero(input_tensor[:, i, :])) > 0:
                output_tensor, hidden_tensor = rnn(input_tensor[:, i, :], hidden_tensor)

        loss = criterion(output_tensor, expected_output)
        loss.backward()

# Add parameters' gradients to their values, multiplied by learning rate
        for p in rnn.parameters():
            p.data.add_(-LEARNING_RATE, p.grad.data)

        train_losses.append(loss.item())

    train_average_losses.append(sum(train_losses) / len(train_losses))



    # TEST

    test_losses = []

    with torch.no_grad():
        for step_index, batch in enumerate(test_dataloader):

            input_tensor, expected_output = batch
            hidden_tensor = rnn.initHidden(input_tensor.shape[0])

            for i in range(input_tensor.shape[1]):
                if len(torch.nonzero(input_tensor[:, i, :])) > 0:
                    output_tensor, hidden_tensor = rnn(input_tensor[:, i, :], hidden_tensor)

            loss = criterion(ou
```

```
        rnn.zero_grad()

        for i in range(input_tensor.shape[1]):
            if len(torch.nonzero(input_tensor[:, i, :])) > 0:
                output_tensor, hidden_tensor = rnn(input_tensor[:, i, :], hidden_tensor)

        loss = criterion(output_tensor, expected_output)
        loss.backward()

        optimizer.step()




        train_losses.append(loss.item())

    train_average_losses.append(sum(train_losses) / len(train_losses))
    writer.add_scalar('Train Loss', sum(train_losses) / len(train_losses), epoch_index)

    # TEST

    test_losses = []

    with torch.no_grad():
        for step_index, batch in enumerate(test_dataloader):

            input_tensor, expected_output = batch
            hidden_tensor = rnn.initHidden(input_tensor.shape[0])

            for i in range(input_tensor.shape[1]):
                if len(torch.nonzero(input_tensor[:, i, :])) > 0:
                    output_tensor, hidden_tensor = rnn(input_tensor[:, i, :], hidden_tensor)

            loss = criterion(ou
```

```
    tput_tensor, expected_output)
            test_losses.append(loss.item())

    test_average_losses.append(sum(test_losses) / len(test_losses))



# In[10]:



plt.figure()
plt.plot(train_average_losses, "r")
plt.plot(test_average_losses, "b")
plt.xlabel("Epoch")
plt.ylabel("Average Loss")


# ## Step 4b: Add an optimizer
# You can use an optimizer included with PyTorch rather than writing your own.
# While the RNN trains here, we'll write this part in VS Code.

# ## Step 5: Evaluate the Network
# There are many metrics for evaluating a classifier, e.g. F1 score, accuracy, etc.
# For this tutorial, we'll create a confusion matrix. **Rows are actual/target languages, columns are predictions.**

# In[9]:


import matplotlib.ticker as ticker

confusion = torch.zeros(len(languages), len(languages))

with torch.no_grad():
    for step_index, batch in enumerate(test_dataloader):

        input_tensor, expected_output = batch
```

```
    tput_tensor, expected_output)
            test_losses.append(loss.item())

    test_average_losses.append(sum(test_losses) / len(test_losses))
    writer.add_scalar('Test Loss', sum(test_losses) / len(test_losses), epoch_index)


# In[10]:



plt.figure()
plt.plot(train_average_losses, "r")
plt.plot(test_average_losses, "b")
plt.xlabel("Epoch")
plt.ylabel("Average Loss")


# ## Step 4b: Add an optimizer
# You can use an optimizer included with PyTorch rather than writing your own.
# While the RNN trains here, we'll write this part in VS Code.

# ## Step 5: Evaluate the Network
# There are many metrics for evaluating a classifier, e.g. F1 score, accuracy, etc.
# For this tutorial, we'll create a confusion matrix. **Rows are actual/target languages, columns are predictions.**

# In[9]:


import matplotlib.ticker as ticker

confusion = torch.zeros(len(languages), len(languages))

with torch.no_grad():
    for step_index, batch in enumerate(test_dataloader):

        input_tensor, expected_output = batch
```