Lab 8: Generating Surnames with a GRU RNN

In earlier notebooks we were trying to classify last names to predict what nationality/language the last name originates from. In this notebook we'll use a special type of RNN, a GRU, to try and generate new last names. There are 3 questions at the end of this notebook.

Note that the code and data in this notebook are derived from Chapter 7 of the PyTorchNLPBook, found here.

Imports

```
from argparse import Namespace
import os
import json

import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import functional as F
from torch.utils.data import Dataset, DataLoader
import tqdm.auto
```

Data Vectorization classes

Vocabulary

```
class Vocabulary(object):
    """Class to process text and extract vocabulary for mapping"""

    def __init__(self, token_to_idx=None):
        """
        Args:
            token_to_idx (dict): a pre-existing map of tokens to indices
        """
```

```
        if token_to_idx is None:
            token_to_idx = {}
        self._token_to_idx = token_to_idx

        self._idx_to_token = {idx: token
                              for token, idx in self._token_to_idx.items()}

    def to_serializable(self):
        """ returns a dictionary that can be serialized """
        return {'token_to_idx': self._token_to_idx}

    @classmethod
    def from_serializable(cls, contents):
        """ instantiates the Vocabulary from a serialized dictionary """
        return cls(**contents)

    def add_token(self, token):
        """Update mapping dicts based on the token.

        Args:
            token (str): the item to add into the Vocabulary
        Returns:
            index (int): the integer corresponding to the token
        """
        if token in self._token_to_idx:
            index = self._token_to_idx[token]
        else:
            index = len(self._token_to_idx)
            self._token_to_idx[token] = index
            self._idx_to_token[index] = token
        return index

    def add_many(self, tokens):
        """Add a list of tokens into the Vocabulary

        Args:
            tokens (list): a list of string tokens
        Returns:
            indices (list): a list of indices corresponding to the tokens
        """
        return [self.add_token(token) for token in tokens]

    def lookup_token(self, token):
        """Retrieve the index associated with the token

        Args:
            token (str): the token to look up
        Returns:
            index (int): the index corresponding to the token
        """
```

```
        return self._token_to_idx[token]

    def lookup_index(self, index):
        """Return the token associated with the index

        Args:
            index (int): the index to look up
        Returns:
            token (str): the token corresponding to the index
        Raises:
            KeyError: if the index is not in the Vocabulary
        """
        if index not in self._idx_to_token:
            raise KeyError("the index (%d) is not in the Vocabulary" % index)
        return self._idx_to_token[index]

    def __str__(self):
        return "<Vocabulary(size=%d)>" % len(self)

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


class SequenceVocabulary(Vocabulary):
    def __init__(self, token_to_idx=None, unk_token="<UNK>",
                 mask_token="<MASK>", begin_seq_token="<BEGIN>",
                 end_seq_token="<END>"):

        super(SequenceVocabulary, self).__init__(token_to_idx)

        self._mask_token = mask_token
        self._unk_token = unk_token
        self._begin_seq_token = begin_seq_token
        self._end_seq_token = end_seq_token

        self.mask_index = self.add_token(self._mask_token)
        self.unk_index = self.add_token(self._unk_token)
        self.begin_seq_index = self.add_token(self._begin_seq_token)
        self.end_seq_index = self.add_token(self._end_seq_token)

    def to_serializable(self):
        contents = super(SequenceVocabulary, self).to_serializable()
        contents.update({'unk_token': self._unk_token,
                         'mask_token': self._mask_token,
                         'begin_seq_token': self._begin_seq_token,
                         'end_seq_token': self._end_seq_token})
        return contents

    def lookup_token(self, token):
        """Retrieve the index associated with the token
          or the UNK index if token isn't present.
```

```
        Args:
            token (str): the token to look up
        Returns:
            index (int): the index corresponding to the token
        Notes:
            `unk_index` needs to be >=0 (having been added into the Vocabulary)
              for the UNK functionality
        """
        if self.unk_index >= 0:
            return self._token_to_idx.get(token, self.unk_index)
        else:
            return self._token_to_idx[token]
```

Vectorizer

```
class SurnameVectorizer(object):
    """ The Vectorizer which coordinates the Vocabularies and puts them to use"""
    def __init__(self, char_vocab, nationality_vocab):
        """
        Args:
            char_vocab (SequenceVocabulary): maps words to integers
            nationality_vocab (Vocabulary): maps nationalities to integers
        """
        self.char_vocab = char_vocab
        self.nationality_vocab = nationality_vocab

    def vectorize(self, surname, vector_length=-1):
        """Vectorize a surname into a vector of observations and targets

        The outputs are the vectorized surname split into two vectors:
            surname[:-1] and surname[1:]
        At each timestep, the first vector is the observation and the second vector is th

        Args:
            surname (str): the surname to be vectorized
            vector_length (int): an argument for forcing the length of index vector
        Returns:
            a tuple: (from_vector, to_vector)
            from_vector (numpy.ndarray): the observation vector
            to_vector (numpy.ndarray): the target prediction vector
        """
        indices = [self.char_vocab.begin_seq_index]
        indices.extend(self.char_vocab.lookup_token(token) for token in surname)
        indices.append(self.char_vocab.end_seq_index)

        if vector_length < 0:
            vector_length = len(indices) - 1
```

```
        from_vector = np.zeros(vector_length, dtype=np.int64)
        from_indices = indices[:-1]
        from_vector[:len(from_indices)] = from_indices
        from_vector[len(from_indices):] = self.char_vocab.mask_index

        to_vector = np.zeros(vector_length, dtype=np.int64)
        to_indices = indices[1:]
        to_vector[:len(to_indices)] = to_indices
        to_vector[len(to_indices):] = self.char_vocab.mask_index

        return from_vector, to_vector

    @classmethod
    def from_dataframe(cls, surname_df):
        """Instantiate the vectorizer from the dataset dataframe

        Args:
            surname_df (pandas.DataFrame): the surname dataset
        Returns:
            an instance of the SurnameVectorizer
        """
        char_vocab = SequenceVocabulary()
        nationality_vocab = Vocabulary()

        for index, row in surname_df.iterrows():
            for char in row.surname:
                char_vocab.add_token(char)
            nationality_vocab.add_token(row.nationality)

        return cls(char_vocab, nationality_vocab)

    @classmethod
    def from_serializable(cls, contents):
        """Instantiate the vectorizer from saved contents

        Args:
            contents (dict): a dict holding two vocabularies for this vectorizer
                This dictionary is created using `vectorizer.to_serializable()`
        Returns:
            an instance of SurnameVectorizer
        """
        char_vocab = SequenceVocabulary.from_serializable(contents['char_vocab'])
        nat_vocab =  Vocabulary.from_serializable(contents['nationality_vocab'])

        return cls(char_vocab=char_vocab, nationality_vocab=nat_vocab)

    def to_serializable(self):
        """ Returns the serializable contents """
        return {'char_vocab': self.char_vocab.to_serializable(),
                'nationality_vocab': self.nationality_vocab.to_serializable()}
```

Dataset

```
class SurnameDataset(Dataset):
    def __init__(self, surname_df, vectorizer):
        """
        Args:
            surname_df (pandas.DataFrame): the dataset
            vectorizer (SurnameVectorizer): vectorizer instatiated from dataset
        """
        self.surname_df = surname_df
        self._vectorizer = vectorizer

        self._max_seq_length = max(map(len, self.surname_df.surname)) + 2

        self.train_df = self.surname_df[self.surname_df.split=='train']
        self.train_size = len(self.train_df)

        self.val_df = self.surname_df[self.surname_df.split=='val']
        self.validation_size = len(self.val_df)

        self.test_df = self.surname_df[self.surname_df.split=='test']
        self.test_size = len(self.test_df)

        self._lookup_dict = {'train': (self.train_df, self.train_size),
                             'val': (self.val_df, self.validation_size),
                             'test': (self.test_df, self.test_size)}

        self.set_split('train')

    @classmethod
    def load_dataset_and_make_vectorizer(cls, surname_csv):
        """Load dataset and make a new vectorizer from scratch

        Args:
            surname_csv (str): location of the dataset
        Returns:
            an instance of SurnameDataset
        """

        surname_df = pd.read_csv(surname_csv)
        return cls(surname_df, SurnameVectorizer.from_dataframe(surname_df))

    @classmethod
    def load_dataset_and_load_vectorizer(cls, surname_csv, vectorizer_filepath):
        """Load dataset and the corresponding vectorizer.
        Used in the case in the vectorizer has been cached for re-use

        Args:
```

```
        Args:
            surname_csv (str): location of the dataset
            vectorizer_filepath (str): location of the saved vectorizer
        Returns:
            an instance of SurnameDataset
        """
        surname_df = pd.read_csv(surname_csv)
        vectorizer = cls.load_vectorizer_only(vectorizer_filepath)
        return cls(surname_df, vectorizer)

    @staticmethod
    def load_vectorizer_only(vectorizer_filepath):
        """a static method for loading the vectorizer from file

        Args:
            vectorizer_filepath (str): the location of the serialized vectorizer
        Returns:
            an instance of SurnameVectorizer
        """
        with open(vectorizer_filepath) as fp:
            return SurnameVectorizer.from_serializable(json.load(fp))

    def save_vectorizer(self, vectorizer_filepath):
        """saves the vectorizer to disk using json

        Args:
            vectorizer_filepath (str): the location to save the vectorizer
        """
        with open(vectorizer_filepath, "w") as fp:
            json.dump(self._vectorizer.to_serializable(), fp)

    def get_vectorizer(self):
        """ returns the vectorizer """
        return self._vectorizer

    def set_split(self, split="train"):
        self._target_split = split
        self._target_df, self._target_size = self._lookup_dict[split]

    def __len__(self):
        return self._target_size

    def __getitem__(self, index):
        """the primary entry point method for PyTorch datasets

        Args:
            index (int): the index to the data point
        Returns:
            a dictionary holding the data point: (x_data, y_target, class_index)
        """
        row = self._target_df.iloc[index]
```

```
        from_vector, to_vector = \
            self._vectorizer.vectorize(row.surname, self._max_seq_length)

        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_data': from_vector,
                'y_target': to_vector,
                'class_index': nationality_index}

    def get_num_batches(self, batch_size):
        """Given a batch size, return the number of batches in the dataset

        Args:
            batch_size (int)
        Returns:
            number of batches in the dataset
        """
        return len(self) // batch_size

def generate_batches(dataset, batch_size, shuffle=True,
                     drop_last=True, device="cpu"):
    """
    A generator function which wraps the PyTorch DataLoader. It will
      ensure each tensor is on the write device location.
    """
    dataloader = DataLoader(dataset=dataset, batch_size=batch_size,
                            shuffle=shuffle, drop_last=drop_last)

    for data_dict in dataloader:
        out_data_dict = {}
        for name, tensor in data_dict.items():
            out_data_dict[name] = data_dict[name].to(device)
        yield out_data_dict
```

The Model: SurnameGenerationModel

```
class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size, num_nationalities,
                 rnn_hidden_size, batch_first=True, padding_idx=0, dropout_p=0.5):
        """
        Args:
            char_embedding_size (int): The size of the character embeddings
            char_vocab_size (int): The number of characters to embed
            num_nationalities (int): The size of the prediction vector
            rnn_hidden_size (int): The size of the RNN's hidden state
            batch_first (bool): Informs whether the input tensors will
```

```
                have batch or the sequence on the 0th dimension
        """
        super(SurnameGenerationModel, self).__init__()

        self.char_emb = nn.Embedding(num_embeddings=char_vocab_size,
                                     embedding_dim=char_embedding_size,
                                     padding_idx=padding_idx)

        self.nation_emb = nn.Embedding(num_embeddings=num_nationalities,
                                       embedding_dim=rnn_hidden_size)

        self.rnn = nn.GRU(input_size=char_embedding_size,
                          hidden_size=rnn_hidden_size,
                          batch_first=batch_first)

        self.fc = nn.Linear(in_features=rnn_hidden_size,
                            out_features=char_vocab_size)

        self.drop = nn.Dropout(dropout_p)
        self._dropout_p = dropout_p

    def forward(self, x_in, nationality_index, apply_softmax=False):

        x_embedded = self.char_emb(x_in)

        # hidden_size: (num_layers * num_directions, batch_size, rnn_hidden_size)
        nationality_embedded = self.nation_emb(nationality_index).unsqueeze(0)

        y_out, _ = self.rnn(x_embedded, nationality_embedded)

        batch_size, seq_size, feat_size = y_out.shape
        y_out = y_out.contiguous().view(batch_size * seq_size, feat_size)

        y_out = self.fc(self.drop(y_out))

        if apply_softmax:
            y_out = F.softmax(y_out, dim=1)

        new_feat_size = y_out.shape[-1]
        y_out = y_out.view(batch_size, seq_size, new_feat_size)

        return y_out


def sample_from_model(model, vectorizer, nationalities, sample_size=20,
                      temperature=1.0):
    """Sample a sequence of indices from the model

    Args:
        model (SurnameGenerationModel): the trained model
        vectorizer (SurnameVectorizer): the corresponding vectorizer
```

```
        nationalities (list): a list of integers representing nationalities
        sample_size (int): the max length of the samples
        temperature (float): accentuates or flattens
            the distribution.
            0.0 < temperature < 1.0 will make it peakier.
            temperature > 1.0 will make it more uniform
    Returns:
        indices (torch.Tensor): the matrix of indices;
        shape = (num_samples, sample_size)
    """
    num_samples = len(nationalities)
    begin_seq_index = [vectorizer.char_vocab.begin_seq_index
                       for _ in range(num_samples)]
    begin_seq_index = torch.tensor(begin_seq_index,
                                   dtype=torch.int64).unsqueeze(dim=1)
    indices = [begin_seq_index]
    nationality_indices = torch.tensor(nationalities, dtype=torch.int64).unsqueeze(dim=0)
    h_t = model.nation_emb(nationality_indices)

    for time_step in range(sample_size):
        x_t = indices[time_step]
        x_emb_t = model.char_emb(x_t)
        rnn_out_t, h_t = model.rnn(x_emb_t, h_t)
        prediction_vector = model.fc(rnn_out_t.squeeze(dim=1))
        probability_vector = F.softmax(prediction_vector / temperature, dim=1)
        indices.append(torch.multinomial(probability_vector, num_samples=1))
    indices = torch.stack(indices).squeeze().permute(1, 0)
    return indices

def decode_samples(sampled_indices, vectorizer):
    """Transform indices into the string form of a surname

    Args:
        sampled_indices (torch.Tensor): the inidces from `sample_from_model`
        vectorizer (SurnameVectorizer): the corresponding vectorizer
    """
    decoded_surnames = []
    vocab = vectorizer.char_vocab

    for sample_index in range(sampled_indices.shape[0]):
        surname = ""
        for time_step in range(sampled_indices.shape[1]):
            sample_item = sampled_indices[sample_index, time_step].item()
            if sample_item == vocab.begin_seq_index:
                continue
            elif sample_item == vocab.end_seq_index:
                break
            else:
                surname += vocab.lookup_index(sample_item)
        decoded_surnames.append(surname)
    return decoded_surnames
```

```
    return decoded_surnames
```

Training Routine

Helper functions

```
def make_train_state(args):
    return {'stop_early': False,
            'early_stopping_step': 0,
            'early_stopping_best_val': 1e8,
            'learning_rate': args.learning_rate,
            'epoch_index': 0,
            'train_loss': [],
            'train_acc': [],
            'val_loss': [],
            'val_acc': [],
            'test_loss': -1,
            'test_acc': -1,
            'model_filename': args.model_state_file}

def update_train_state(args, model, train_state):
    """Handle the training state updates.
    Components:
     - Early Stopping: Prevent overfitting.
     - Model Checkpoint: Model is saved if the model is better

    :param args: main arguments
    :param model: model to train
    :param train_state: a dictionary representing the training state values
    :returns:
        a new train_state
    """

    # Save one model at least
    if train_state['epoch_index'] == 0:
        torch.save(model.state_dict(), train_state['model_filename'])
        train_state['stop_early'] = False

    # Save model if performance improved
    elif train_state['epoch_index'] >= 1:
        loss_tm1, loss_t = train_state['val_loss'][-2:]

        # If loss worsened
        if loss_t >= loss_tm1:
            # Update step
            train_state['early_stopping_step'] += 1
        # Loss decreased
```

```
        else:
            # Save the best model
            if loss_t < train_state['early_stopping_best_val']:
                torch.save(model.state_dict(), train_state['model_filename'])
                train_state['early_stopping_best_val'] = loss_t

            # Reset early stopping step
            train_state['early_stopping_step'] = 0

        # Stop early ?
        train_state['stop_early'] = \
            train_state['early_stopping_step'] >= args.early_stopping_criteria

    return train_state

def normalize_sizes(y_pred, y_true):
    """Normalize tensor sizes

    Args:
        y_pred (torch.Tensor): the output of the model
            If a 3-dimensional tensor, reshapes to a matrix
        y_true (torch.Tensor): the target predictions
            If a matrix, reshapes to be a vector
    """
    if len(y_pred.size()) == 3:
        y_pred = y_pred.contiguous().view(-1, y_pred.size(2))
    if len(y_true.size()) == 2:
        y_true = y_true.contiguous().view(-1)
    return y_pred, y_true

def compute_accuracy(y_pred, y_true, mask_index):
    y_pred, y_true = normalize_sizes(y_pred, y_true)

    _, y_pred_indices = y_pred.max(dim=1)

    correct_indices = torch.eq(y_pred_indices, y_true).float()
    valid_indices = torch.ne(y_true, mask_index).float()

    n_correct = (correct_indices * valid_indices).sum().item()
    n_valid = valid_indices.sum().item()

    return n_correct / n_valid * 100

def sequence_loss(y_pred, y_true, mask_index):
    y_pred, y_true = normalize_sizes(y_pred, y_true)
    return F.cross_entropy(y_pred, y_true, ignore_index=mask_index)
```

General utilities

```
def set_seed_everywhere(seed, cuda):
    np.random.seed(seed)
    torch.manual_seed(seed)
    if cuda:
        torch.cuda.manual_seed_all(seed)

def handle_dirs(dirpath):
    if not os.path.exists(dirpath):
        os.makedirs(dirpath)
```

Settings and some prep work

```
args = Namespace(
    # Data and Path information
    surname_csv="https://raw.githubusercontent.com/sgeinitz/DSML4220/main/data/surnames.csv
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="./models/surnames/rnn_conditional",
    # Model hyper parameters
    char_embedding_size=150,
    rnn_hidden_size=100,
    # Training hyper parameters
    seed=42,
    learning_rate=0.001,
    batch_size=32,
    num_epochs=100,
    early_stopping_criteria=5,
    # Runtime options
    catch_keyboard_interrupt=True,
    cuda=True,
    expand_filepaths_to_save_dir=True,
    reload_from_files=False,
)

if args.expand_filepaths_to_save_dir:
    args.vectorizer_file = os.path.join(args.save_dir,
                                        args.vectorizer_file)

    args.model_state_file = os.path.join(args.save_dir,
                                         args.model_state_file)

    print("Expanded filepaths: ")
    print("\t{}".format(args.vectorizer_file))
    print("\t{}".format(args.model_state_file))

# Check CUDA
if not torch.cuda.is_available():
```

```
    args.cuda = False

args.device = torch.device("cuda" if args.cuda else "cpu")

print("Using CUDA: {}".format(args.cuda))

# Set seed for reproducibility
set_seed_everywhere(args.seed, args.cuda)

# handle dirs
handle_dirs(args.save_dir)
```

```
Expanded filepaths:
        ./models/surnames/rnn_conditional/vectorizer.json
        ./models/surnames/rnn_conditional/model.pth
Using CUDA: False
```

Initializations

```
if args.reload_from_files:
    # training from a checkpoint
    dataset = SurnameDataset.load_dataset_and_load_vectorizer(args.surname_csv,
                                                              args.vectorizer_file)
else:
    # create dataset and vectorizer
    dataset = SurnameDataset.load_dataset_and_make_vectorizer(args.surname_csv)
    dataset.save_vectorizer(args.vectorizer_file)

vectorizer = dataset.get_vectorizer()

model = SurnameGenerationModel(char_embedding_size=args.char_embedding_size,
                               char_vocab_size=len(vectorizer.char_vocab),
                               num_nationalities=len(vectorizer.nationality_vocab),
                               rnn_hidden_size=args.rnn_hidden_size,
                               padding_idx=vectorizer.char_vocab.mask_index,
                               dropout_p=0.3)
```

```
vectorizer.nationality_vocab.lookup_index(4)
```

```
'English'
```

```
dataset.train_df.iloc[0]
```

	0
nationality	Arabic
nationality_index	15

split	train
surname	Totah

dtype: object

```
dataset.set_split('train')
dataset[0]
```

```
{'x_data': array([2, 4, 5, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'y_target': array([4, 5, 6, 7, 8, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'class_index': 0}
```

```
for i in range(12):
    print(f" vocab at index: {i} corresponds to token: {vectorizer.char_vocab.lookup_inde
```

```
 vocab at index: 0 corresponds to token: <MASK>
 vocab at index: 1 corresponds to token: <UNK>
 vocab at index: 2 corresponds to token: <BEGIN>
 vocab at index: 3 corresponds to token: <END>
 vocab at index: 4 corresponds to token: T
 vocab at index: 5 corresponds to token: o
 vocab at index: 6 corresponds to token: t
 vocab at index: 7 corresponds to token: a
 vocab at index: 8 corresponds to token: h
 vocab at index: 9 corresponds to token: A
 vocab at index: 10 corresponds to token: b
 vocab at index: 11 corresponds to token: u
```

Training loop

```
mask_index = vectorizer.char_vocab.mask_index

model = model.to(args.device)


optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer=optimizer,
                                                 mode='min', factor=0.5,
                                                 patience=1)

losses = {'train':[], 'val':[]}

train_state = make_train_state(args)

epoch_bar = tqdm.notebook.tqdm(desc='training routine',
                               total=args.num_epochs,
                               position=0, leave=True)
```

```
dataset.set_split('train')
train_bar = tqdm.notebook.tqdm(desc='split=train',
                          total=dataset.get_num_batches(args.batch_size),
                          position=1,
                          leave=True)
dataset.set_split('val')
val_bar = tqdm.notebook.tqdm(desc='split=val',
                        total=dataset.get_num_batches(args.batch_size),
                        position=1,
                        leave=True)

try:
    for epoch_index in range(args.num_epochs):
        train_state['epoch_index'] = epoch_index

        # Iterate over training dataset

        # setup: batch generator, set loss and acc to 0, set train mode on
        dataset.set_split('train')
        batch_generator = generate_batches(dataset,
                                           batch_size=args.batch_size,
                                           device=args.device)
        running_loss = 0.0
        running_acc = 0.0
        model.train()

        for batch_index, batch_dict in enumerate(batch_generator):
            # the training routine is these 5 steps:

            # --------------------------------------
            # step 1. zero the gradients
            optimizer.zero_grad()

            # step 2. compute the output
            y_pred = model(x_in=batch_dict['x_data'],
                           nationality_index=batch_dict['class_index'])

            # step 3. compute the loss
            loss = sequence_loss(y_pred, batch_dict['y_target'], mask_index)
            losses['train'].append(loss.item())


            # step 4. use loss to produce gradients
            loss.backward()

            # step 5. use optimizer to take gradient step
            optimizer.step()
            # -----------------------------------------
            # compute the  running loss and running accuracy
            running_loss += (loss.item() - running_loss) / (batch_index + 1)
```

```
        running_loss += (loss.item() - running_loss) / (batch_index + 1)
        acc_t = compute_accuracy(y_pred, batch_dict['y_target'], mask_index)
        running_acc += (acc_t - running_acc) / (batch_index + 1)

        # update bar
        train_bar.set_postfix(loss=running_loss,
                              acc=running_acc,
                              epoch=epoch_index)
        train_bar.update()

    train_state['train_loss'].append(running_loss)
    train_state['train_acc'].append(running_acc)

    # Iterate over val dataset

    # setup: batch generator, set loss and acc to 0; set eval mode on
    dataset.set_split('val')
    batch_generator = generate_batches(dataset,
                                       batch_size=args.batch_size,
                                       device=args.device)
    running_loss = 0.
    running_acc = 0.
    model.eval()

    for batch_index, batch_dict in enumerate(batch_generator):
        # compute the output
        y_pred = model(x_in=batch_dict['x_data'],
                       nationality_index=batch_dict['class_index'])

        # step 3. compute the loss
        loss = sequence_loss(y_pred, batch_dict['y_target'], mask_index)
        losses['val'].append(loss.item())

        # compute the  running loss and running accuracy
        running_loss += (loss.item() - running_loss) / (batch_index + 1)
        acc_t = compute_accuracy(y_pred, batch_dict['y_target'], mask_index)
        running_acc += (acc_t - running_acc) / (batch_index + 1)

        # Update bar
        val_bar.set_postfix(loss=running_loss, acc=running_acc,
                        epoch=epoch_index)
        val_bar.update()

    train_state['val_loss'].append(running_loss)
    train_state['val_acc'].append(running_acc)

    train_state = update_train_state(args=args, model=model,
                                     train_state=train_state)

    scheduler.step(train_state['val_loss'][-1])
```

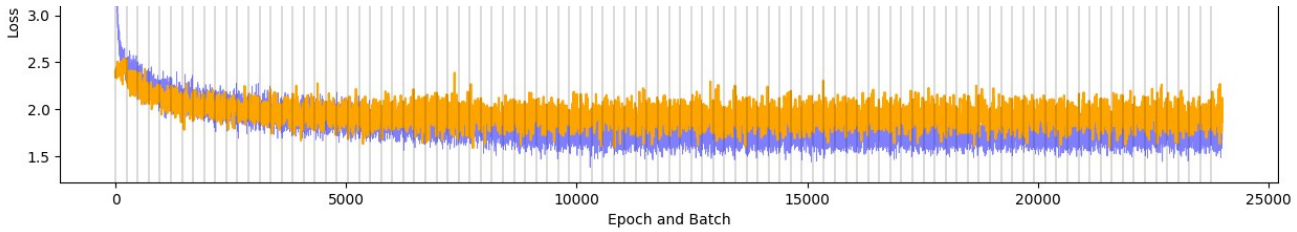
```
        if train_state['stop_early']:
            break

        # move model to cpu for sampling

        nationalities = np.random.choice(np.arange(len(vectorizer.nationality_vocab)), re
        model = model.cpu()
        sampled_surnames = decode_samples(
            sample_from_model(model, vectorizer, nationalities=nationalities),
            vectorizer)

        sample1 = "{}->{}".format(vectorizer.nationality_vocab.lookup_index(nationalities
                                  sampled_surnames[0])
        sample2 = "{}->{}".format(vectorizer.nationality_vocab.lookup_index(nationalities
                                  sampled_surnames[1])
        epoch_bar.set_postfix(sample1=sample1,
                              sample2=sample2)
        # move model back to whichever device it should be on
        model = model.to(args.device)

        train_bar.n = 0
        val_bar.n = 0
        epoch_bar.update()

except KeyboardInterrupt:
    print("Exiting loop")
```

training routine: 100% 100/100 [13:48<00:00, 8.20s/it, sample1=Chinese->Jie, sample2=Arabic->Handal]

split=train: 100% 239/240 [13:47<00:05, 5.65s/it, acc=46, epoch=99, loss=1.73]

```
import matplotlib
import matplotlib.pyplot as plt
matplotlib.rc('figure', figsize=(15,4))
val_ticks = [(i+1)*len(losses['train'])/len(losses['val']) for i in range(len(losses['val
plt.plot(range(len(losses['train'])), losses['train'], c='blue', lw=0.5, alpha=0.5)
plt.plot(val_ticks, losses['val'], c='orange')
for i in range(args.num_epochs):
    plt.axvline(x=i*len(losses['train'])/args.num_epochs, c='black', lw=0.2)
plt.ylabel('Loss')
plt.xlabel('Epoch and Batch')
plt.legend(('Train','Validation'))
```

```
<matplotlib.legend.Legend at 0x7efc48fe1790>
```




```
# compute the loss & accuracy on the test set using the best available model

model.load_state_dict(torch.load(train_state['model_filename']))

model = model.to(args.device)

dataset.set_split('test')
batch_generator = generate_batches(dataset,
                                   batch_size=args.batch_size,
                                   device=args.device)
running_acc = 0.
model.eval()

for batch_index, batch_dict in enumerate(batch_generator):
    # compute the output
    y_pred = model(x_in=batch_dict['x_data'],
                   nationality_index=batch_dict['class_index'])

    # compute the loss
    loss = sequence_loss(y_pred, batch_dict['y_target'], mask_index)

    # compute the running loss and running accuracy
    running_loss += (loss.item() - running_loss) / (batch_index + 1)
    acc_t = compute_accuracy(y_pred, batch_dict['y_target'], mask_index)
    running_acc += (acc_t - running_acc) / (batch_index + 1)

train_state['test_loss'] = running_loss
train_state['test_acc'] = running_acc


print("Test loss: {};".format(train_state['test_loss']))
print("Test Accuracy: {}".format(train_state['test_acc']))
```

```
Test loss: 1.9428046614516017;
```

```
Test Accuracy: 41.08545598561189
```

Sampling

Q1: Consider the following example in which a linear output of `[-2, 1, 0.5]` is modified using *temperature*, and then put through softmax.

- How many different classes are represented by this output?

There are three different classes

- What value of the temperature will effectively result in only one class being predicted?

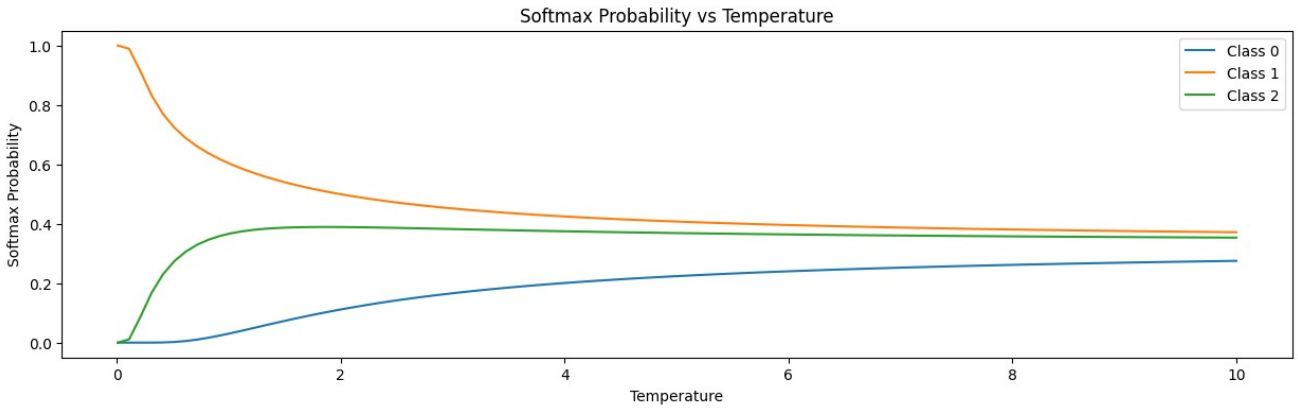
At a temperature of 0, it will only take the value from class 1.

- What value of the temperature will effectively result in each class being equally likely?

At a temperature of 10 or higher, the values for all three classes will converge.

```
# Create a plot with temperature on the x-axis ranging from 0.1 to 10 and with three curv
temperature = np.linspace(0.01, 10, 100)
ex = torch.tensor([-2, 1, 0.5])
softmax = F.softmax(ex / temperature[:, None], dim=1).numpy()
plt.plot(temperature, softmax)
plt.xlabel('Temperature')
plt.ylabel('Softmax Probability')
plt.title('Softmax Probability vs Temperature')
plt.legend(['Class 0', 'Class 1', 'Class 2'])
plt.show()
```

```
<ipython-input-21-dee4f77a4cf7>:4: DeprecationWarning: __array_wrap__ must accept cor
  softmax = F.softmax(ex / temperature[:, None], dim=1).numpy()
```



```
# If you want to experiment with different temperature values manually, you can do so her
temperature = 5
F.softmax(ex / temperature, dim=0).numpy().round(2)
```

```
array([0.22, 0.41, 0.37], dtype=float32)
```

```
---
#### Q2: Using only the temperature parameter
for each nationality that the model (based or
the most likely last name.

* How many unique last names are sampled for
why?

The temperature will output two unique names
1 becuase of how many names are being sampled
higher the temperature value.

---
```

Q2: Using only the temperature parameter below, try to sample last names for each nationality that the model (based on the training data) believes to be the most likely last name.

- How many unique last names are sampled for each nationality? Can you explain why?

The temperature will output two unique names per nationality when the value is 1 becuase of how many names are being sampled. It looks like jibberish the higher the temperature value.

```
model = model.cpu()
for index in range(len(vectorizer.nationality_vocab)):
    nationality = vectorizer.nationality_vocab.lookup_index(index)
    print("Sampled for {}: ".format(nationality))
    sampled_indices = sample_from_model(model, vectorizer, nationalities=[index] * 2, tem
    for sampled_surname in decode_samples(sampled_indices, vectorizer):
        print("-  " + sampled_surname)
```

```
 Sampled for Arabic:
 -  Mansour
 -  Rahal
 Sampled for Chinese:
 -  Fean
```

```
- Cneo
Sampled for Czech:
- Tlakowozerso
- Rida
Sampled for Dutch:
- Riderdleber
- Sennadden
Sampled for English:
- Rinney
- Reyilse
Sampled for French:
- Lenko
- Ciamonnes
Sampled for German:
- Tosher
- Hortz
Sampled for Greek:
- Zaigeros
- Neradakis
Sampled for Irish:
- Firgent
- Aidham
Sampled for Italian:
- Bultono
- Cardala
Sampled for Japanese:
- Takazo
- Tsui
Sampled for Korean:
- Jond
- Tou
Sampled for Polish:
- Kezacin
- Janogak
Sampled for Portuguese:
- Mastura
- Daria
Sampled for Russian:
- Tyneziutz
- Makhinnikov
Sampled for Scottish:
- Gurb
- Mutos
Sampled for Spanish:
- Veidera
- Giezla
Sampled for Vietnamese:
- Van
- To
```

Q3: Using only the temperature parameter below, try to sample last names so that the names generated for one nationality are statistically no different than names

from any other nationality.

- Now how many unique last names are sampled for each nationality? Can you explain why?

When the temperature value is 10, the model treats all of the surnames uniformly and will not create statisitcally different surnames as shown in the graph above.

```
model = model.cpu()
for index in range(len(vectorizer.nationality_vocab)):
    nationality = vectorizer.nationality_vocab.lookup_index(index)
    print("Sampled for {}: ".format(nationality))
    sampled_indices = sample_from_model(model, vectorizer, nationalities=[index] * 2, tem
    for sampled_surname in decode_samples(sampled_indices, vectorizer):
        print("-  " + sampled_surname)
```

```
 Sampled for Arabic:
 -  łFtN<MASK>pVe'd
 -  DDßaavJeêÉFà
 Sampled for Chinese:
 -  mwUe<MASK>ñUŻx/xçäolÉqNgv
 -  yeTáYàDiNgfczaeqhłA1
 Sampled for Czech:
 -  vWgüSfOBbgwàO<UNK>pliQõn
 -  AçcõChvcòLèTöbbkñkñy
 Sampled for Dutch:
 -  gEYqsqèuBò
 -  ãłVŚyyAMxáóiKbcKòdhè
 Sampled for English:
 -  vxCònItwöñápcphvUBiH
 -  cíüYüàÁnwseHöKBñswŚŚ
 Sampled for French:
 -  ooDùHãMnnñVõfró1wcäN
 -  JOżVzrqbóxmyzFòIŚsph
 Sampled for German:
 -  C/jFAjjShBöP<UNK>wyHnreG
 -  U:tCkąodi<MASK>ññ1Hn'Áòh'
 Sampled for Greek:
 -  ãàswßödAõfAKcnk-ZGTü
 -  òYnmmñ'éTiwQArixnúñ1
 Sampled for Irish:
 -  J'Miüy:Bç
 -  quLa
 Sampled for Italian:
 -  Pì-i'úêöóóeFMêY:èßdõ
 -  'lloZSêapySapgAEDcmB
 Sampled for Japanese:
 -  iturH'ìłRg'í/Qmúąutc
 -  ZEłFSgHYàjbŻNyus-fMv
 Sampled for Korean:
 -  êòkCmXoèpéh
 -  IqùFìY1żmtjCpąŻliwñJ
 Sampled for Polish:
```

```
- TaKòõeypłvsjk-cWsóéY
- 'rrùb<MASK>OKVjsòxQgrdÉk
Sampled for Portuguese:
- FsoKe
- /
Sampled for Russian:
- ZfTHüsknuèKXêÉSYkebS
- MruYslEx
Sampled for Scottish:
- cBsncovìqmualgzó'UAp
- v1erOijZjU-hxúMöùjQr
Sampled for Spanish:
- Ávè<MASK>1gkòXiŚãçHeZoŚeS
- MiLRùąZBìTAöéCHõöask
Sampled for Vietnamese:
- JãérnñìbHxSE:aWkúfVG
- dãáydõ
```