‹ Return to Classroom

Deploying a Sentiment Analysis Model

REVIEW

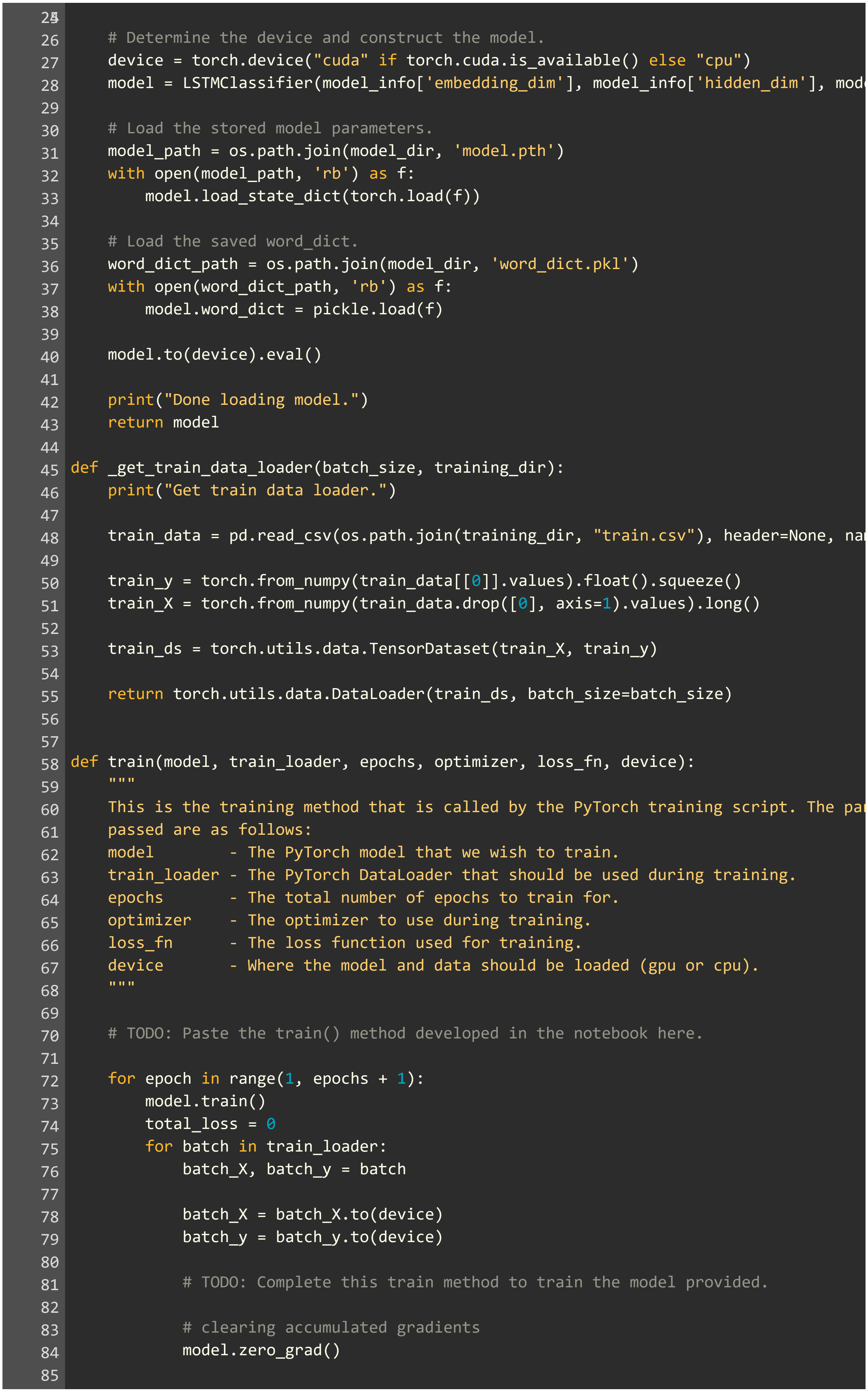
CODE REVIEW 3

HISTORY

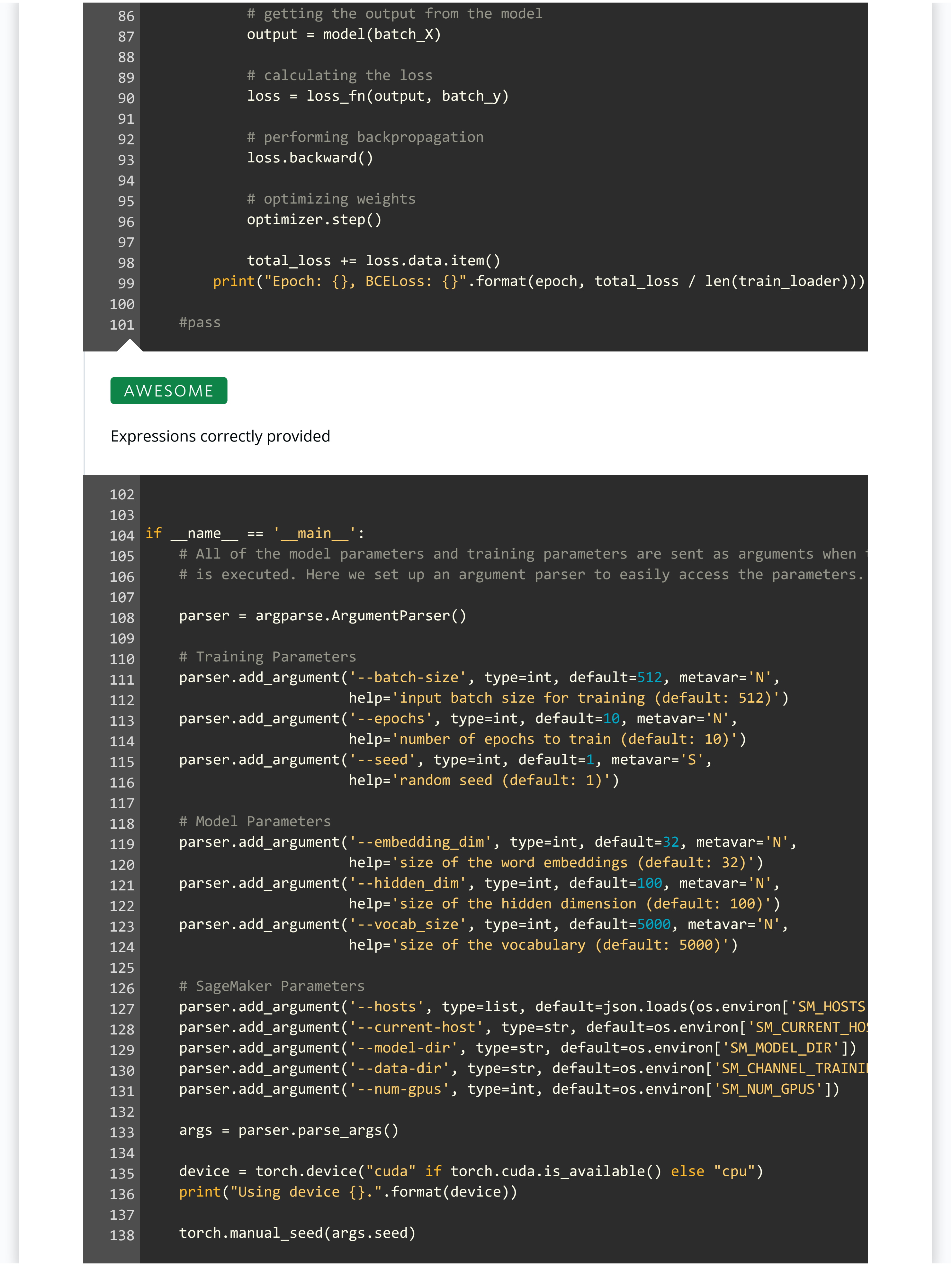
▸ website/index.html 1

▾ train/train.py 1

```
import argparse
import json
import os
import pickle
import sys
import sagemaker_containers
import pandas as pd
import torch
import torch.optim as optim
import torch.utils.data

from model import LSTMClassifier

def model_fn(model_dir):
    """Load the PyTorch model from the `model_dir` directory."""
    print("Loading model.")

    # First, load the parameters used to create the model.
    model_info = {}
    model_info_path = os.path.join(model_dir, 'model_info.pth')
    with open(model_info_path, 'rb') as f:
        model_info = torch.load(f)

    print("model_info: {}".format(model_info))
```



```

    # Determine the device and construct the model.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = LSTMClassifier(model_info['embedding_dim'], model_info['hidden_dim'], mode

    # Load the stored model parameters.
    model_path = os.path.join(model_dir, 'model.pth')
    with open(model_path, 'rb') as f:
        model.load_state_dict(torch.load(f))

    # Load the saved word_dict.
    word_dict_path = os.path.join(model_dir, 'word_dict.pkl')
    with open(word_dict_path, 'rb') as f:
        model.word_dict = pickle.load(f)

    model.to(device).eval()

    print("Done loading model.")
    return model

def _get_train_data_loader(batch_size, training_dir):
    print("Get train data loader.")

    train_data = pd.read_csv(os.path.join(training_dir, "train.csv"), header=None, na

    train_y = torch.from_numpy(train_data[[0]].values).float().squeeze()
    train_X = torch.from_numpy(train_data.drop([0], axis=1).values).long()

    train_ds = torch.utils.data.TensorDataset(train_X, train_y)

    return torch.utils.data.DataLoader(train_ds, batch_size=batch_size)


def train(model, train_loader, epochs, optimizer, loss_fn, device):
    """
    This is the training method that is called by the PyTorch training script. The pa
    passed are as follows:
    model        - The PyTorch model that we wish to train.
    train_loader - The PyTorch DataLoader that should be used during training.
    epochs       - The total number of epochs to train for.
    optimizer    - The optimizer to use during training.
    loss_fn      - The loss function used for training.
    device       - Where the model and data should be loaded (gpu or cpu).
    """

    # TODO: Paste the train() method developed in the notebook here.

    for epoch in range(1, epochs + 1):
        model.train()
        total_loss = 0
        for batch in train_loader:
            batch_X, batch_y = batch

            batch_X = batch_X.to(device)
            batch_y = batch_y.to(device)

            # TODO: Complete this train method to train the model provided.

            # clearing accumulated gradients
            model.zero_grad()

```



```python
            # getting the output from the model
            output = model(batch_X)

            # calculating the loss
            loss = loss_fn(output, batch_y)

            # performing backpropagation
            loss.backward()

            # optimizing weights
            optimizer.step()

            total_loss += loss.data.item()
        print("Epoch: {}, BCELoss: {}".format(epoch, total_loss / len(train_loader)))

    #pass
```

AWESOME

Expressions correctly provided

```python


if __name__ == '__main__':
    # All of the model parameters and training parameters are sent as arguments when
    # is executed. Here we set up an argument parser to easily access the parameters.

    parser = argparse.ArgumentParser()

    # Training Parameters
    parser.add_argument('--batch-size', type=int, default=512, metavar='N',
                        help='input batch size for training (default: 512)')
    parser.add_argument('--epochs', type=int, default=10, metavar='N',
                        help='number of epochs to train (default: 10)')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')

    # Model Parameters
    parser.add_argument('--embedding_dim', type=int, default=32, metavar='N',
                        help='size of the word embeddings (default: 32)')
    parser.add_argument('--hidden_dim', type=int, default=100, metavar='N',
                        help='size of the hidden dimension (default: 100)')
    parser.add_argument('--vocab_size', type=int, default=5000, metavar='N',
                        help='size of the vocabulary (default: 5000)')

    # SageMaker Parameters
    parser.add_argument('--hosts', type=list, default=json.loads(os.environ['SM_HOSTS
    parser.add_argument('--current-host', type=str, default=os.environ['SM_CURRENT_HO
    parser.add_argument('--model-dir', type=str, default=os.environ['SM_MODEL_DIR'])
    parser.add_argument('--data-dir', type=str, default=os.environ['SM_CHANNEL_TRAINI
    parser.add_argument('--num-gpus', type=int, default=os.environ['SM_NUM_GPUS'])

    args = parser.parse_args()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print("Using device {}.".format(device))

    torch.manual_seed(args.seed)
    # Load the training data.
    train_loader = _get_train_data_loader(args.batch_size, args.data_dir)

    # Build the model.
    model = LSTMClassifier(args.embedding_dim, args.hidden_dim, args.vocab_size).to(d

    with open(os.path.join(args.data_dir, "word_dict.pkl"), "rb") as f:
        model.word_dict = pickle.load(f)

    print("Model loaded with embedding_dim {}, hidden_dim {}, vocab_size {}.".format(
        args.embedding_dim, args.hidden_dim, args.vocab_size
    ))

    # Train the model.
    optimizer = optim.Adam(model.parameters())
    loss_fn = torch.nn.BCELoss()

    train(model, train_loader, args.epochs, optimizer, loss_fn, device)

    # Save the parameters used to construct the model
    model_info_path = os.path.join(args.model_dir, 'model_info.pth')
    with open(model_info_path, 'wb') as f:
        model_info = {
            'embedding_dim': args.embedding_dim,
            'hidden_dim': args.hidden_dim,
            'vocab_size': args.vocab_size,
        }
        torch.save(model_info, f)

    # Save the word_dict
    word_dict_path = os.path.join(args.model_dir, 'word_dict.pkl')
    with open(word_dict_path, 'wb') as f:
        pickle.dump(model.word_dict, f)

    # Save the model parameters
    model_path = os.path.join(args.model_dir, 'model.pth')
    with open(model_path, 'wb') as f:
        torch.save(model.cpu().state_dict(), f)

```

▸ serve/predict.py 1

RETURN TO PATH