Mapping amazon Reviews to Stars

Irene Won Choi & Khushant Khurana

Contents

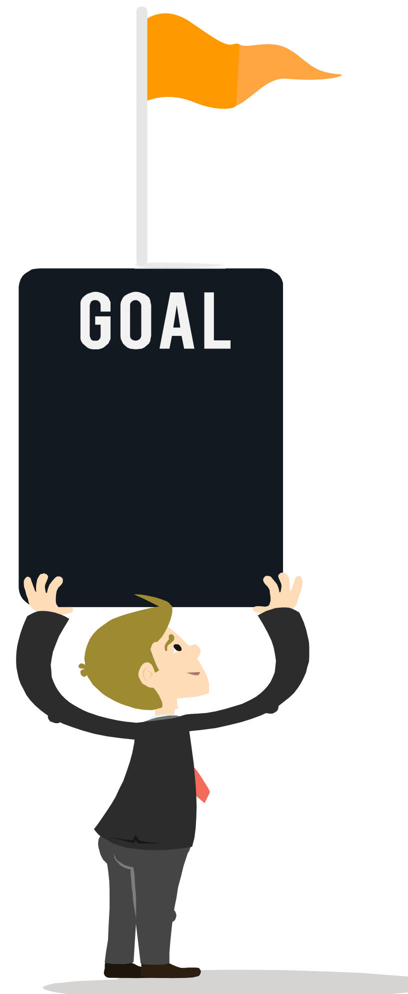
Problem Statement

OBJECTIVE

- Develop a machine learning model to classify textual reviews by predicting what the rating of a product will be.

WHY?

- Stars offer a quantitative measure!
- Trends of given stars over time can be used for marketing because of their anonymous nature.



PART 1

INTRO

- Acquiring the dataset!
- Raw Data Exploration

Approach

Source

Kaggle: “Sentiment Analysis Python” by Rob Mulla

Structure

Features inclide product ID, Score, Text Review and small summary for the text review.

Size

The dataset contains 568454 reviews.

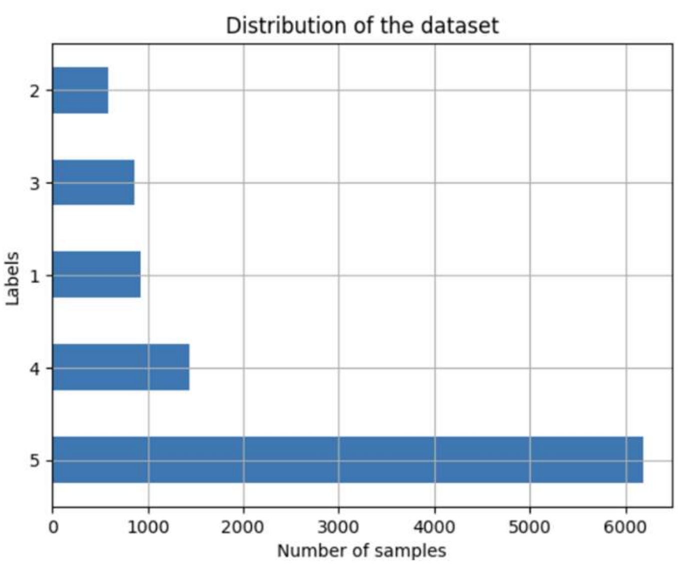
Distribution

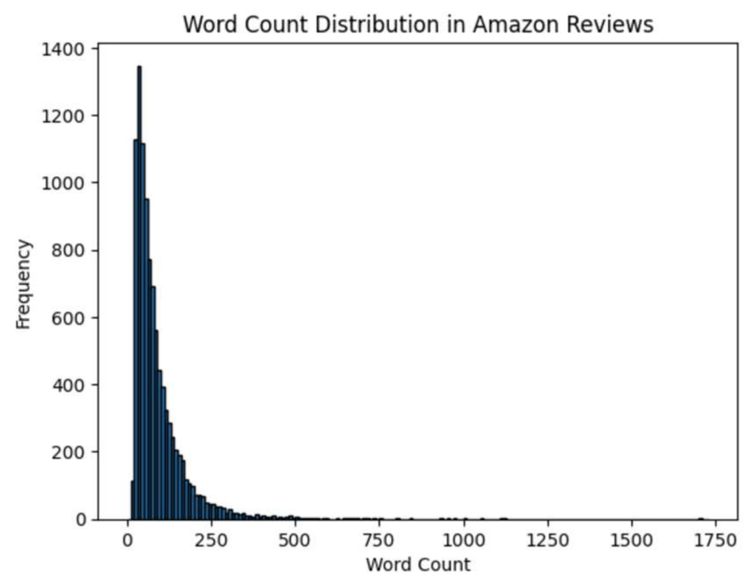
The dataset contains a lot more text reviews with 5 stars.

Dataset

Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
1	B001E4KFG0	A3SGXH7A	delmartian	1	1	5	1.3E+09	Good Quality Dog Fc	I have bought several of the Vitality canned dog food products and have found
2	B00813GRG4	A1D87F6Z	dll pa	0	0	1	1.35E+09	Not as Advertised	Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actually sı
3	B000LQOCH0	ABXLMWJ	Natalia Corres "	1	1	4	1.22E+09	"Delight" says it all	This is a confection that has been around a few centuries. It is a light, pillowy c
4	B000UA0QIQ	A395BORC	Karl	3	3	2	1.31E+09	Cough Medicine	If you are looking for the secret ingredient in Robitussin I believe I have found
5	B006K2ZZ7K	A1UQRSCI	Michael D. Bighi	0	0	5	1.35E+09	Great taffy	Great taffy at a great price. There was a wide assortment of yummy taffy. Deli
6	B006K2ZZ7K	ADT0SRK1	Twoapennythin	0	0	4	1.34E+09	Nice Taffy	I got a wild hair for taffy and ordered this five pound bag. The taffy was all very
7	B006K2ZZ7K	A1SP2KVK	David C. Sullivai	0	0	5	1.34E+09	Great! Just as good	This saltwater taffy had great flavors and was very soft and chewy. Each candy
8	B006K2ZZ7K	A3JRGQVE	Pamela G. Willia	0	0	5	1.34E+09	Wonderful, tasty taf	This taffy is so good. It is very soft and chewy. The flavors are amazing. I woul
9	B000E7L2R4	A1MZYO9	R. James	1	1	5	1.32E+09	Yay Barley	Right now I'm mostly just sprouting this so my cats can eat the grass. They love
10	B00171APVA	A21BT40V	Carol A. Reed	0	0	5	1.35E+09	Healthy Dog Food	This is a very healthy dog food. Good for their digestion. Also good for small pu

Raw dataset





PART

2

Preprocessing

- Tokenization
- Preparing Data for the Model

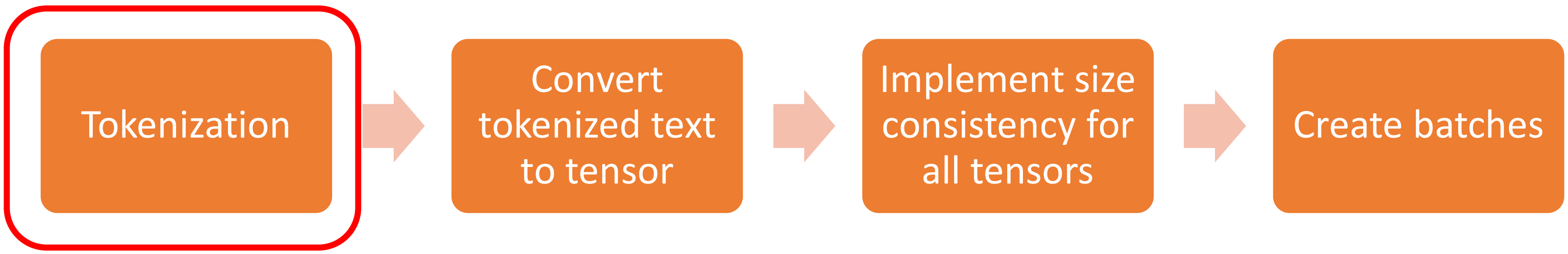
Framework used

PyTorch

```
# Importing torch and its functionalities for text processing
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torch.nn.utils.rnn import pad_sequence
```

Importing torch and its extensions.

Methodology For Pre-Processing



Tokenization


```
# Preprocess data and create batches

tokenizer = get_tokenizer("basic_english")

def yield_tokens(data_iter):
    for text in data_iter:
        yield tokenizer(text)
```

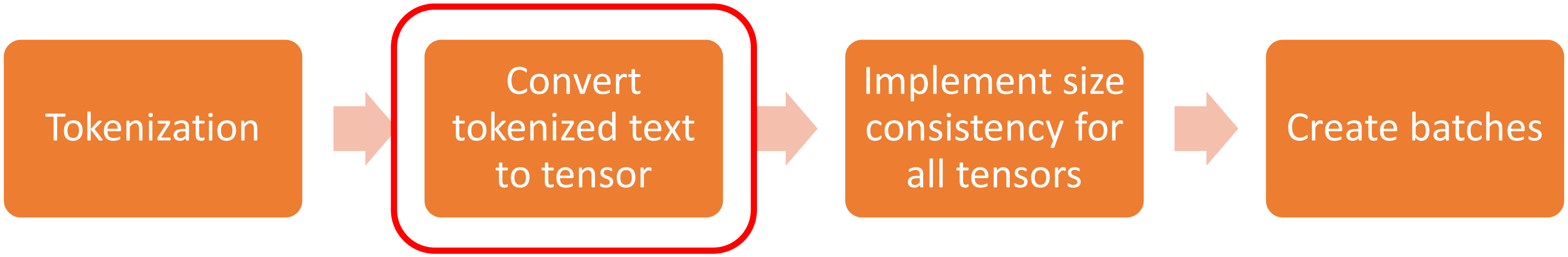
PyTorch Tokenizer!

Convert text to tokens

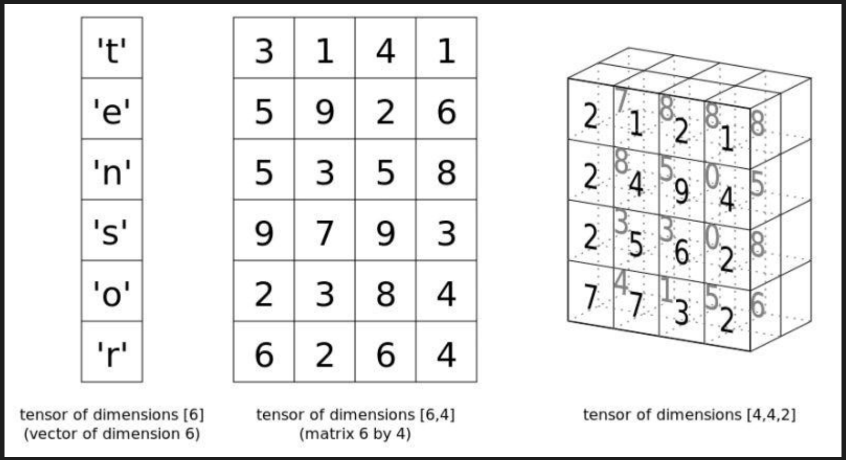
```
# Suppose 'review_data' is your dataset with reviews
vocab = build_vocab_from_iterator(yield_tokens(x_data), specials=["<unk>"])
vocab.set_default_index(vocab["<unk>"])
```

Building vocab object!

Methodology For Pre-Processing



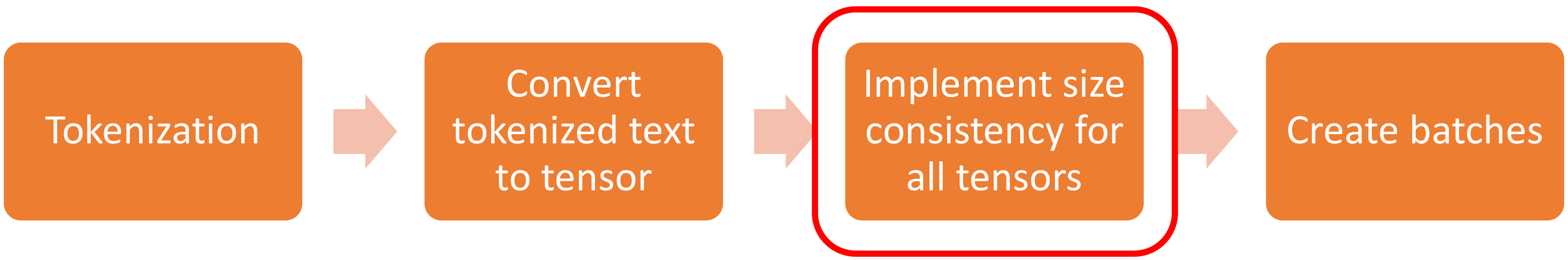
Converting tokenized text to tensors



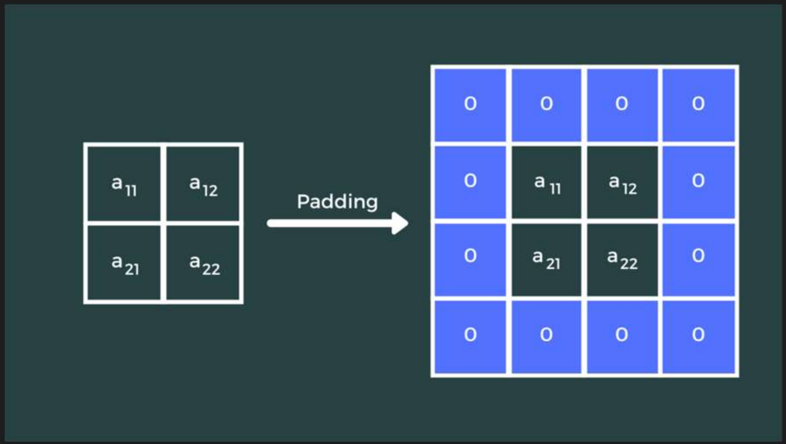

```python
# Custom Dataset
class ReviewDataset(Dataset):
    def __init__(self, texts, labels, vocab):
        self.texts = [
            torch.tensor(vocab(tokenizer(text)), dtype=torch.long) for text in texts
        ]
        self.labels = labels

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

    def __getitem__(self, idx):
        return self.texts[idx], self.labels[idx]
```

Function to tokenize text and convert to "long" datatype tensor.

Methodology For Pre-Processing

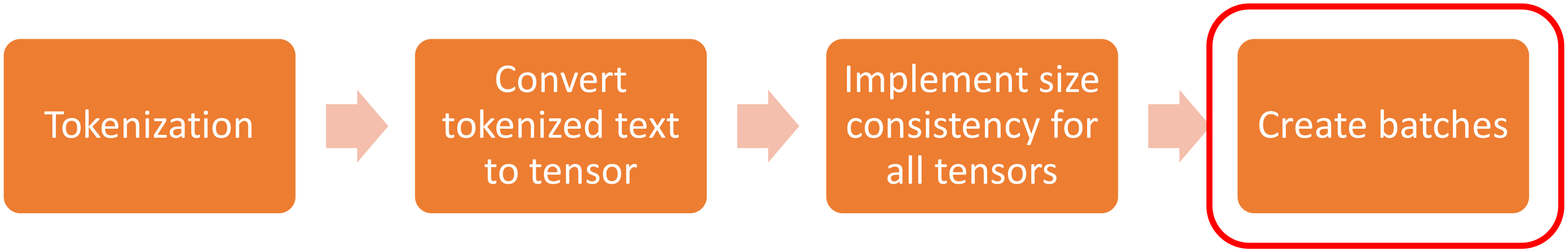


Implementing size consistency for tensors




```
def collate_fn(batch):
    texts, labels = zip(*batch)
    texts_padded = pad_sequence(texts, padding_value=vocab["<pad>"])
    labels = torch.tensor(labels, dtype=torch.long)
    return texts_padded, labels
```

Methodology For Pre-Processing



Creating batches

```
x_train, x_test, y_train, y_test = train_test_split(
    x_data, y_data, test_size=0.2, shuffle=True
)

# Creating corresponding datasets and dataloaders.
# These can be treated as collection of data stored in a convenient format for Pytorch to pass around the neural network.

train_data = ReviewDataset(x_train, y_train, vocab) # Converting the texts to tokens and then to tensors.
train_loader = DataLoader(train_data, batch_size=32, shuffle=True, collate_fn = collate_fn) # Ensuring size consistency and preparing batches.
test_data = ReviewDataset(x_test, y_test, vocab)
test_loader = DataLoader(test_data, batch_size=32, shuffle=True, collate_fn = collate_fn)
```

PART 3

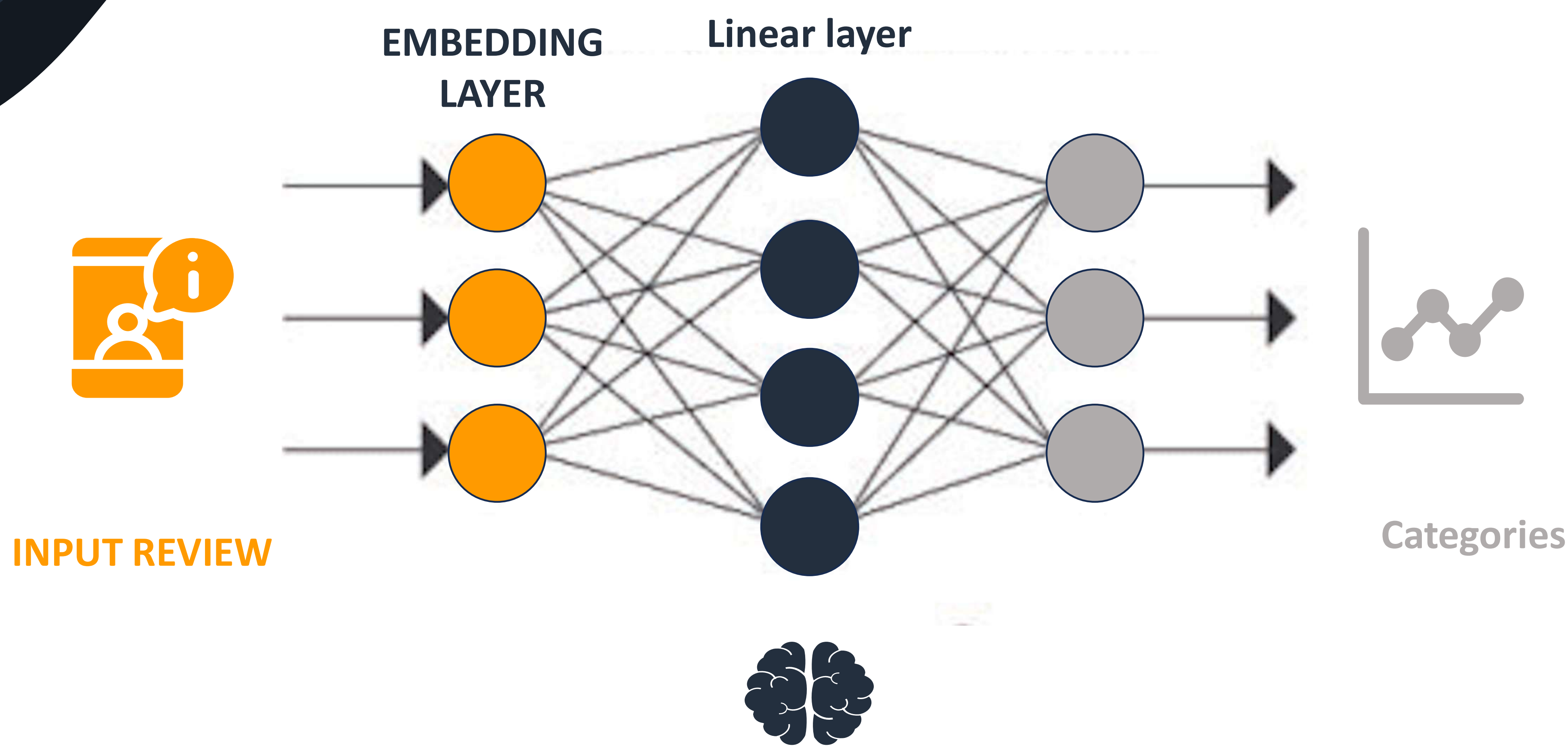
Model Development

- Creating Neural Network Model
- Defining training and validating functions
- Training and testing the model!

Neural Networks

We used **Neural Networks**, which are a type of model inspired by human brain functioning to analyze the sentiment of Amazon reviews.

Embedding Layer: model that translates numbers (representing words) into vectors that capture the meaning of words.


EMBEDDING
LAYER
Linear layer
INPUT REVIEW
Categories

Creating the neural network

```
class neural(nn.Module):
    def __init__(self, num_class):

      # The super() function ensures proper initialization of the inherited properties and methods from the parent class.
        super(neural, self).__init__()

        # Define the embedding layer
        self.embedding = nn.Embedding(vocab_size, hidden_size)

        # Setting the gradient to be true because the weights need to be optimized.
        self.embedding.weight.requires_grad = True

        # A fully-connected linear layer for classification
        self.fc = nn.Linear(hidden_size, num_classes)

        # Initializing demo weights
        self.init_weights()

    def init_weights(self):

        # initialize weights for neural layers
        initrange = 0.05 # chosen randomly.
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_() # Setting the initial bias to 0.

    def forward(self, x):
        # forward input x into embedding layer
        x = self.embedding(x)

        # average the output of embedding layer and forward it to linear fc layer
        x = torch.mean(x, 1)

        # Passing through the 1st linear layer.
        x = self.fc(x)
        return x
```

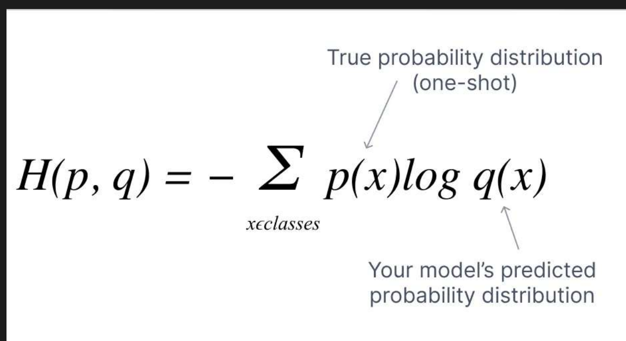
Defining embedding layer

Setting gradient calculation to TRUE

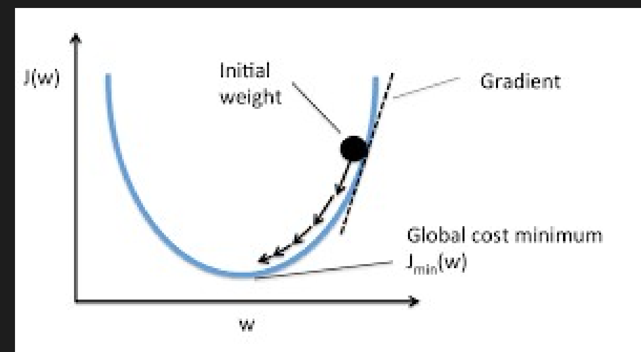
Defining linear layer

Defining the hierarchy of the neural net

Initiating models and other features



Cross entropy Loss



Stochastic Gradient Descent

```
model = neural(num_classes).to(device)
loss_function = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr = .5)
```

Hyper-parameter

Training and validating models

1. Clear optimizer gradient
2. Forward the batch to the model.
3. Compute loss.
4. Back propagate loss
5. Return accuracy.

```python
def train_func(model, train_loader, optimizer, loss_function):

    # Initializing variables.
    train_loss = 0
    train_acc = 0
    num_examples = 0
    model.train()

    # for each batch in train data loader
    for idx, batch in enumerate(train_loader):

      #Parse the batch and extract the tensor for text and corresponding label.
        input_text, labels = batch[0], batch[1]

        input_text = input_text.t()

        # clear optimizer gradient
        optimizer.zero_grad()

        # forward input_text through model
        output = model(input_text)

        # compute loss
        # backpropagate loss
        loss = loss_function(output, labels)
        loss.backward()

        optimizer.step()

        # Compute total loss and accuracy
        train_loss += loss.item()
        train_acc += (output.argmax(1) == labels).sum().item()
        num_examples += labels.size(0)

    return train_loss / num_examples, train_acc / num_examples
```

PART 4

RESULTS

- Training and Testing Accuracy scores.

```
Training Progress:  84%|████████▍ | 42/50 [02:13<00:22,  2.81s/epoch]Epoch: 42  | time in 0 minutes, 3 seconds
        Loss: 0.0345(train)     |       Acc: 62.2%(train)
        Loss: 0.0356(valid)     |       Acc: 62.3%(valid)
Training Progress:  86%|████████▌ | 43/50 [02:16<00:19,  2.74s/epoch]Epoch: 43  | time in 0 minutes, 2 seconds
        Loss: 0.0347(train)     |       Acc: 62.1%(train)
        Loss: 0.0342(valid)     |       Acc: 62.5%(valid)
Training Progress:  88%|████████▊ | 44/50 [02:19<00:16,  2.70s/epoch]Epoch: 44  | time in 0 minutes, 2 seconds
        Loss: 0.0349(train)     |       Acc: 61.9%(train)
        Loss: 0.0346(valid)     |       Acc: 62.6%(valid)
Training Progress:  90%|█████████ | 45/50 [02:22<00:14,  2.90s/epoch]Epoch: 45  | time in 0 minutes, 3 seconds
        Loss: 0.0345(train)     |       Acc: 62.2%(train)
        Loss: 0.0355(valid)     |       Acc: 62.5%(valid)
Training Progress:  92%|█████████▏| 46/50 [02:25<00:11,  2.89s/epoch]Epoch: 46  | time in 0 minutes, 2 seconds
        Loss: 0.0347(train)     |       Acc: 61.9%(train)
        Loss: 0.0386(valid)     |       Acc: 62.6%(valid)
Training Progress:  94%|█████████▍| 47/50 [02:28<00:09,  3.01s/epoch]Epoch: 47  | time in 0 minutes, 3 seconds
        Loss: 0.0347(train)     |       Acc: 61.9%(train)
        Loss: 0.0349(valid)     |       Acc: 62.1%(valid)
Training Progress:  96%|█████████▌| 48/50 [02:31<00:05,  2.90s/epoch]Epoch: 48  | time in 0 minutes, 2 seconds
        Loss: 0.0344(train)     |       Acc: 62.2%(train)
        Loss: 0.0346(valid)     |       Acc: 62.3%(valid)
Training Progress:  98%|█████████▊| 49/50 [02:33<00:02,  2.74s/epoch]Epoch: 49  | time in 0 minutes, 2 seconds
        Loss: 0.0343(train)     |       Acc: 62.5%(train)
        Loss: 0.0364(valid)     |       Acc: 62.6%(valid)
Training Progress: 100%|██████████| 50/50 [02:35<00:00,  3.11s/epoch]Epoch: 50  | time in 0 minutes, 2 seconds
        Loss: 0.0346(train)     |       Acc: 62.5%(train)
        Loss: 0.0351(valid)     |       Acc: 62.7%(valid)
```

The accuracy seems to cap off around 63 percent.

PART 5

Hyperparameter tuning.

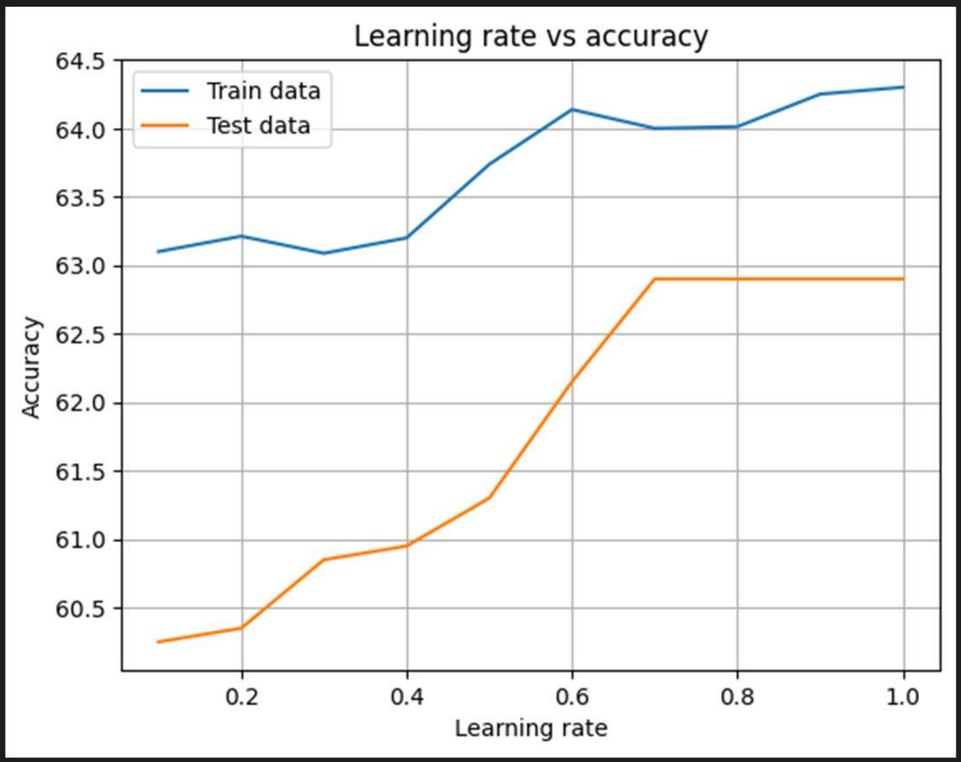
- Evolving the neural net to deep neural net.
- Varying learning rate.
- Varying number of epochs.
- Varying batch size.

Evolving the net to a deep net!

```
Training Progress:  97%|██████████| 97/100 [05:11<00:10,  3.39s/epoch]E
poch: 98  | time in 0 minutes, 3 seconds
        Loss: nan(train)        |       Acc: 9.3%(train)
        Loss: nan(valid)        |       Acc: 9.6%(valid)
Training Progress:  98%|██████████| 98/100 [05:15<00:06,  3.36s/epoch]E
poch: 99  | time in 0 minutes, 3 seconds
        Loss: nan(train)        |       Acc: 9.3%(train)
        Loss: nan(valid)        |       Acc: 9.6%(valid)
Training Progress:  99%|██████████| 99/100 [05:18<00:03,  3.41s/epoch]E
poch: 100  | time in 0 minutes, 3 seconds
        Loss: nan(train)        |       Acc: 9.3%(train)
        Loss: nan(valid)        |       Acc: 9.6%(valid)
Training Progress: 100%|██████████| 100/100 [05:22<00:00,  3.22s/epoch]
```

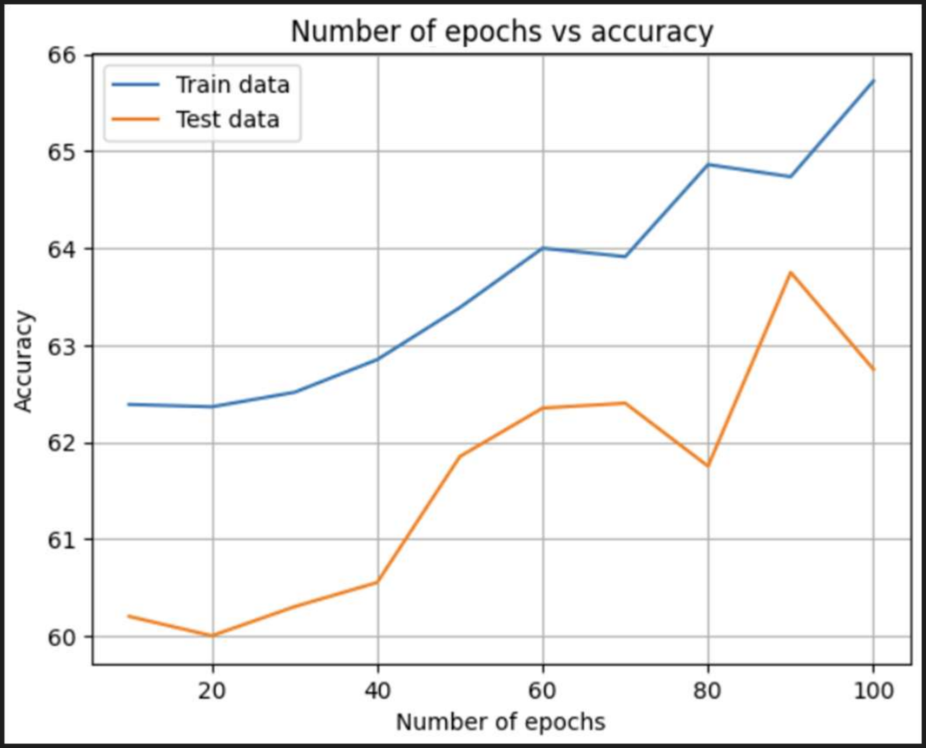
Deep nets are extremely bad. The resulting accuracy is less than 10%.

Iterating over learning rate



The best learning rate is around 0.8. After that the validation accuracy becomes constant.

Iterating over number of epochs



It seems like that the accuracy keeps increasing with the number of epochs. Lets test that!

Trying number of epochs = 200

```
Training Progress:  98%|                    | 197/200 [08:18<00:06,  2.18s/epoch]E
poch: 198  | time in 0 minutes, 1 seconds
        Loss: 0.0351(train)     |       Acc: 56.8%(train)
        Loss: 0.0335(valid)     |       Acc: 62.6%(valid)
Training Progress:  99%|                    | 198/200 [08:20<00:03,  1.93s/epoch]E
poch: 199  | time in 0 minutes, 1 seconds
        Loss: 0.0352(train)     |       Acc: 56.9%(train)
        Loss: 0.0330(valid)     |       Acc: 62.5%(valid)
Training Progress: 100%|                    | 199/200 [08:21<00:01,  1.78s/epoch]E
poch: 200  | time in 0 minutes, 1 seconds
        Loss: 0.0352(train)     |       Acc: 57.0%(train)
        Loss: 0.0342(valid)     |       Acc: 62.6%(valid)
Training Progress: 100%|                    | 200/200 [08:22<00:00,  2.51s/epoch]
```

So the accuracy does not keep rising with number of epochs. Accordingly, there must be a cutoff. Regardless, the accuracy is around 65 percent. But it could be said that the best accuracy comes around epoch size to be 100.

Iterating over batch size

It seems that batch size of 32 has the highest accuracy.

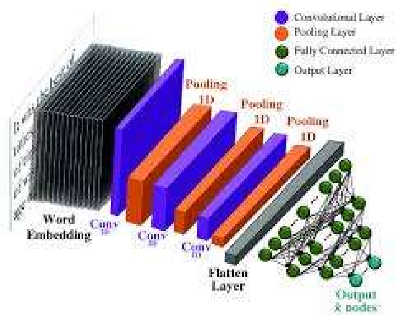
PART 6

Future Work

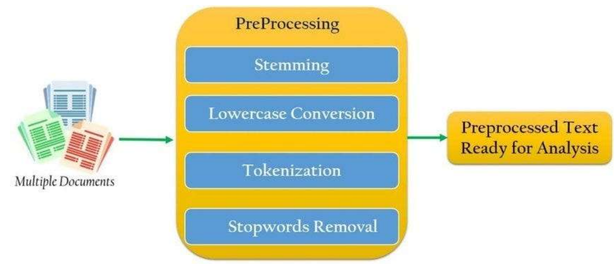
- Possible steps to increase model accuracy.

Future Work

Pre-trained word embeddings



Convolutional neural network



Better pre-processing for text data.