jesús.rabanal-álvarez

April 9, 2024

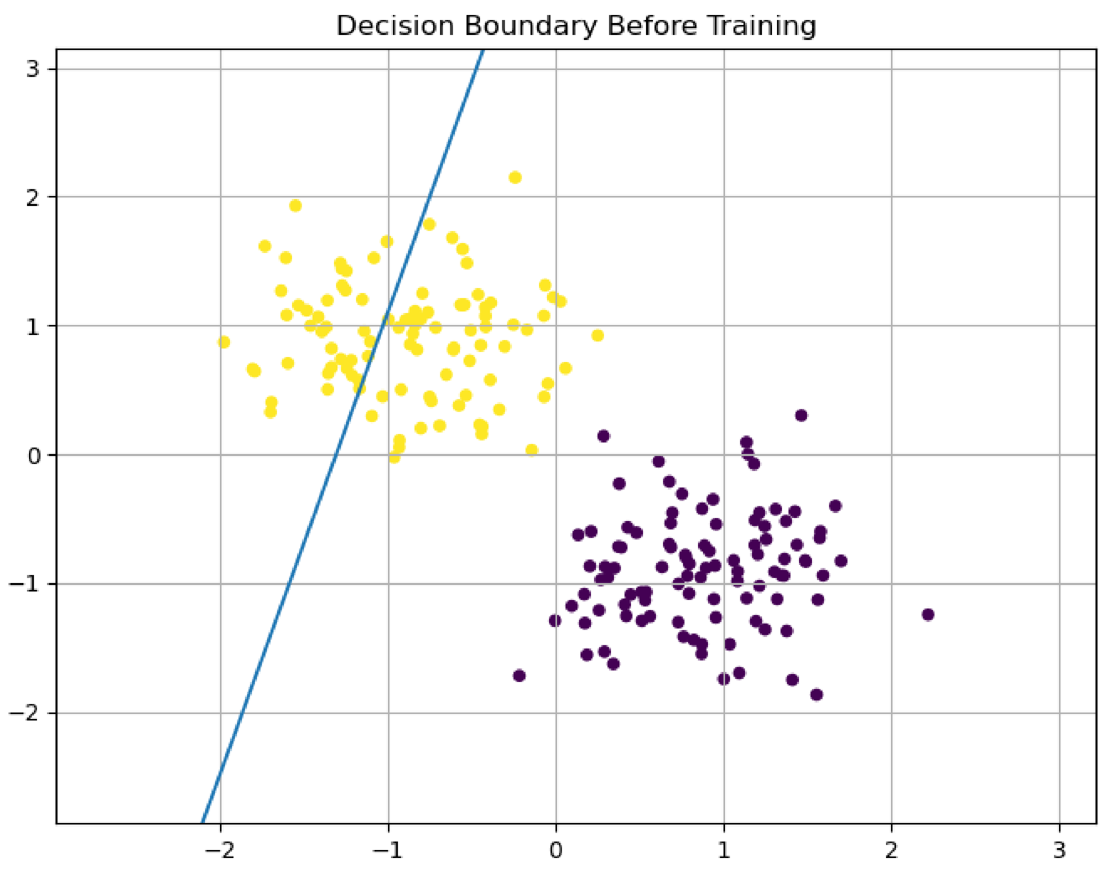
[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

EXERCISE 1 NEURAL NETWORKS

[2]:
```python
class SingleLayerNN:
    def __init__(self, input_dim, seed=42):
        np.random.seed(seed)
        self.input_dim = input_dim
        self.weights = np.random.randn(input_dim)
        self.bias = np.random.randn(1)

    def sigmoid(self, x):
        return 1.0 / (1.0 + np.exp(-x))

    def sigmoid_der(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def f_propagation(self, inputs, validation=False):
        if validation == True:
            z = np.dot(inputs, self.weights) + self.bias
        else:
            z = np.dot(inputs, self.weights) + self.bias
        return self.sigmoid(z)

    def loss(self, predictions, targets):
        eps = 1e-15
        predictions = np.maximum(eps, np.minimum(1-eps, predictions))
        loss = np.linalg.norm(-(targets * np.log(predictions) + (1 - targets) *␣
↪np.log(1 - predictions)))
        return loss

    def b_propagation(self, inputs, predictions, targets, learning_rate):
        delta_w = learning_rate * np.dot(inputs.T, (targets - predictions))
        delta_b = np.mean(learning_rate * (targets - predictions))
        return delta_w, delta_b

    def update_weights_and_biases(self, delta_w, delta_b):
        self.weights = self.weights + delta_w
        self.bias = self.bias + delta_b

    def train(self, inputs, targets, validation_data_inputs,␣
↪validation_data_targets, epochs=1000, learning_rate=0.01, showIterations =␣
↪True):
        for i in range(epochs):
            #training
            predictions = self.f_propagation(inputs)

            loss = self.loss(predictions, targets)
            delta_w, delta_b = self.b_propagation(inputs, predictions, targets,␣
↪learning_rate)
            self.update_weights_and_biases(delta_w, delta_b)

            #validation
            val_predictions = self.f_propagation(validation_data_inputs,␣
↪validation=True)
            val_loss = self.loss(val_predictions, validation_data_targets)

            if showIterations == True:
                print('Iteration: '+str(i)+', Loss:'+str(loss)+', Validation␣
↪Loss:'+str(val_loss))

    def predict(self, x):
        predictions = self.f_propagation(x)
        return predictions

    def plot_decision_boundary(self, inputs, targets, title):
        x_min , x_max = inputs[:, 0]. min() - 1, inputs[:, 0]. max() + 1
        x_values = np.linspace(x_min , x_max , 200)

        y_values = ((-self.weights[0] / self.weights[1]) * x_values) - (self.
↪bias / self.weights[1])

        plt.figure(figsize=(8, 6))
        plt.scatter(inputs[:, 0], inputs[:, 1], c=targets, s=20)

        plt.plot(x_values, y_values, label='Decision Boundary')
        plt.xlim(x_min, x_max)
        plt.ylim(inputs[:, 1]. min() - 1, inputs[:, 1]. max() + 1)
        plt.title(title)
        plt.grid(True)
        plt.show()
```

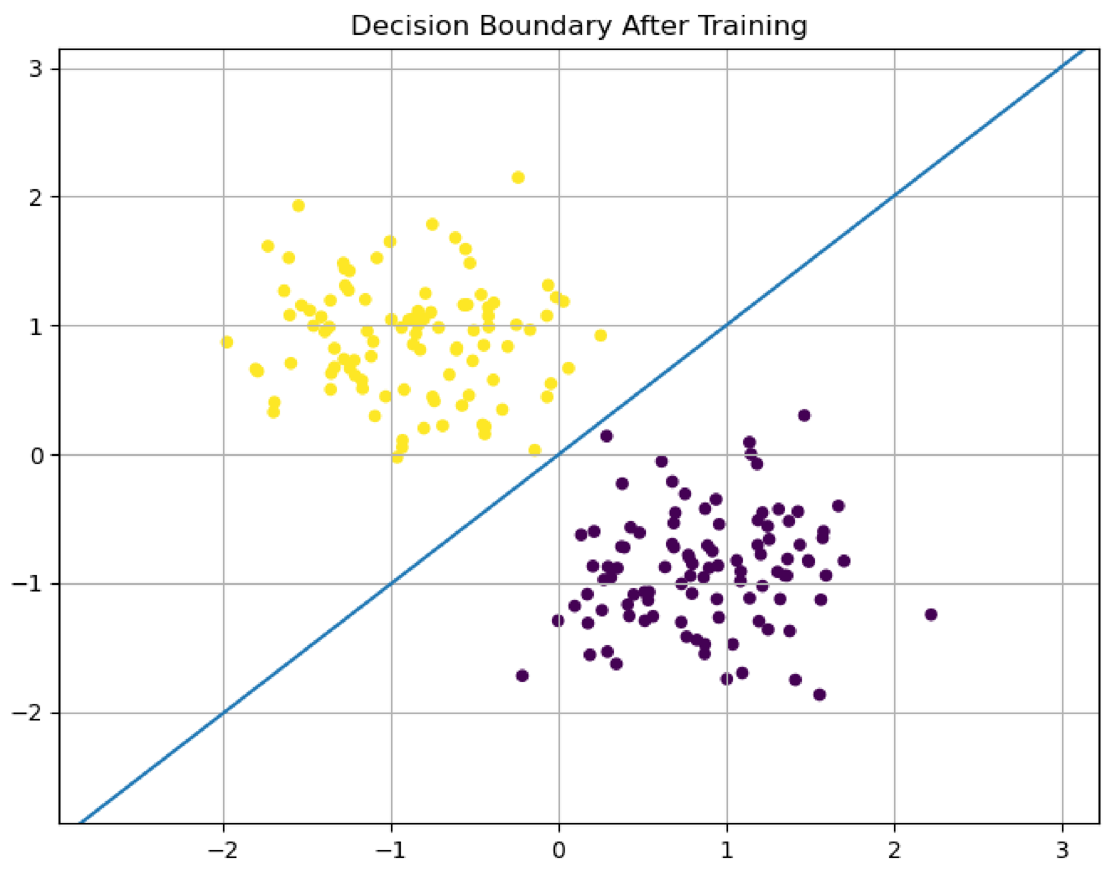
[3]:
```
#create synthetic dataset
n_samples = 100
features_class_0 = np.random.randn(n_samples, 2) + [2, -2] # Class 0
features_class_1 = np.random.randn(n_samples, 2) + [-2, 2] # Class 1
inputs = np.vstack((features_class_0, features_class_1))
targets = np.hstack((np.zeros(n_samples), np.ones(n_samples)))

#standardize the features
inputs = (inputs - np.mean(inputs, axis=0)) / np.std(inputs, axis=0)

#create validation data
n_samples_val = 100
features_class_0_val = np.random.randn(n_samples_val, 2) + [2, -2]  # Class 0
features_class_1_val = np.random.randn(n_samples_val, 2) + [-2, 2]  # Class 1
inputs_val = np.vstack((features_class_0_val, features_class_1_val))
targets_val = np.hstack((np.zeros(n_samples_val), np.ones(n_samples_val)))

# Standardize the validation features
inputs_val = (inputs_val - np.mean(inputs_val, axis=0)) / np.std(inputs_val,␣
 ↪axis=0)
```

[4]:
```
model = SingleLayerNN(input_dim=2)
model.plot_decision_boundary(inputs, targets, title='Decision Boundary Before␣
 ↪Training')
```



```
[7]: model_trained = SingleLayerNN(input_dim=2)
     model_trained.train(inputs, targets, inputs_val, targets_val, epochs=10,␣
      ↪learning_rate=1)
     model_trained.plot_decision_boundary(inputs, targets, title='Decision Boundary␣
      ↪After Training')
```

```
Iteration: 0, Loss:16.284488875397464, Validation Loss:10.032102003627465
Iteration: 1, Loss:8.573252019991854e-08, Validation Loss:10.032102001708308
Iteration: 2, Loss:8.573251942304992e-08, Validation Loss:10.032101999789152
Iteration: 3, Loss:8.573251853520006e-08, Validation Loss:10.032101997869995
Iteration: 4, Loss:8.573251775833144e-08, Validation Loss:10.032101995950839
Iteration: 5, Loss:8.573251698146282e-08, Validation Loss:10.032101994031683
Iteration: 6, Loss:8.573251620459419e-08, Validation Loss:10.032101992107476
Iteration: 7, Loss:8.573251542772556e-08, Validation Loss:10.03210199018832
Iteration: 8, Loss:8.57325145398757e-08, Validation Loss:10.032101988269163
Iteration: 9, Loss:8.573251376300708e-08, Validation Loss:10.032101986350007
```

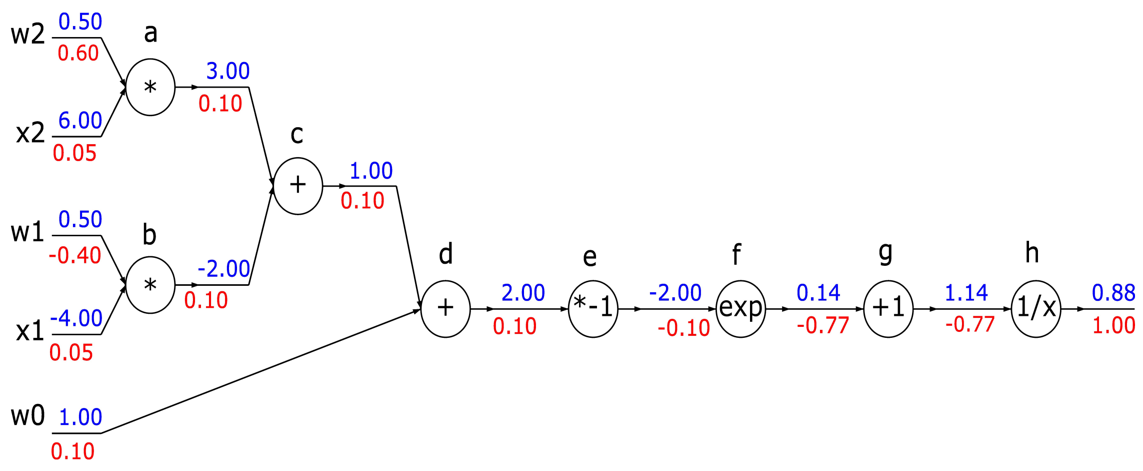


DISCUSSION IMPLEMENTATION AND RESULTS

- Before and after training, a visual examination of decision boundaries reveals significant disparities. The plot titled "Decision Boundary Before Training" illustrates the inability of non-updated weights and biases to delineate a coherent decision boundary. Conversely, the plot labeled "Decision Boundary After Training" demonstrates the model's attainment of a discernible decision boundary subsequent to weight and bias updates. - Furthermore, the selection of an optimal learning rate involved an iterative process, wherein various options were tested, and validation loss calculations were scrutinized. The learning rate plays a pivotal role in determining the speed and stability of convergence during training. Higher learning rates may lead to faster convergence but risk overshooting the optimal solution, while lower learning rates may necessitate longer training times but offer more stable convergence. Through this experimentation, it was determined that iteration 9 at a learning rate of 1, yielded the most satisfactory outcomes in terms of model performance. - The single-layer, single-neuron perceptron is effective for linearly separable datasets but struggles with non-linearly separable ones due to its linear decision boundary. This limitation arises from its inability to capture complex patterns, necessitating alternative models or feature representations for non-linearly separable datasets.

EXERCISE 2 BACKPROPAGATION

1. Calculate and report the local gradients.



For the $h(x) = 1/x$ with constant x, I used the derivative $h(x)' = -1/x^2$
$(-1/1.14^2) \cdot 1 = -0.77$

For the $g(x) = 1x + 1$ with constant x, I used the derivative $g(x)' = 1$
$1 \cdot (-0.77) = -0.77$

For the $f(x) = e^x$ with constant x, I used the derivative $f(x)' = e^x$
$(e^-2) \cdot (-0.77) = -0.10$

For the $e(x) = -1x$ with constant x, I used the derivative $e(x)' = -1$
$-1 \cdot (-0.10) = 0.10$

For the $d(x) = c + w0$ with constant $w0$, I used the derivative $d(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $d(x) = c + w0$ with constant c, I used the derivative $d(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $c(x) = a + b$ with constant a, I used the derivative $c(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $c(x) = a + b$ with constant b, I used the derivative $c(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $a(x) = w_2 \cdot x_2$ with constant w_2, I used the derivative $a(x)' = x_2$
$0.10 \cdot 6.00 = 0.60$

For the $a(x) = w_2 \cdot x_2$ with constant x_2, I used the derivative $a(x)' = w_2$
$0.10 \cdot 0.50 = 0.05$

For the $b(x) = w_1 \cdot x_1$ with constant w_1, I used the derivative $b(x)' = x_1$
$0.10 \cdot (-4.00) = -0.40$

For the $b(x) = w_1 \cdot x_1$ with constant x_1, I used the derivative $b(x)' = w_1$
$0.10 \cdot 0.50 = 0.05$

2. What patterns do you see for backpropagation ?

In the backpropagation that we have calculated we can see several patterns, in addition we get no variations due to derivative of the function given being 1 and in multiplication we can see a pattern where the derivative of the function is the other term making them intertwined meaning a change in one is gonna affect the other. The gradients are routed through addition and multiplication operations according to the chain rule of calculus. The chain rule states how to compute the derivative of a composite function. For addition operations, the gradients are simply added together. For multiplication operations, the gradients are multiplied by the value of the other input. These rules are applied recursively as gradients are backpropagated through the network during training.

3. Assume we are running gradient descent...

$w_{new} = (1.00, 0.50, 0.50) - 0.1 \times (0.1, -0.4, 0.6) = (1.00, 0.50, 0.50) - (0.01, -0.04, 0.06) = (1.00 - 0.01, 0.50 - (-0.04), 0.50 - 0.06) = (0.99, 0.54, 0.44)$

Then the updated weights for $(w0, w1, w2)$ are $(0.99, 0.54, 0.44)$.

And local gradients for the input features in this specific example are not needed.

Exercise_3,_4,_and_5

April 9, 2024

EXERCISE 3 NATURAL LANGUAGE PROCESSING

1. *EXPLORING WORD EMBEDDINGS*

[1]:
```
import gensim.downloader as api
import json
from sklearn.decomposition import PCA
import numpy as np
import matplotlib.pyplot as plt
import transformers
import nltk
from nltk.util import ngrams
import re
```

[2]:
```
glove_vectors = api.load("glove-wiki-gigaword-100")
```

[3]:
```
set_words = ['computer', 'laptop', 'queen', 'king']
top5_similar_words = []
for word in set_words:
    top5_similar_words.append(glove_vectors.most_similar(word, topn=5))
print(top5_similar_words)
```

```
[[('computers', 0.8751983046531677), ('software', 0.8373122215270996),
('technology', 0.7642159461975098), ('pc', 0.7366448640823364), ('hardware',
0.7290390729904175)], [('laptops', 0.8518659472465515), ('computers',
0.7559927105903625), ('phones', 0.7229112386703491), ('portable',
0.7157840728759766), ('desktop', 0.7085692286491394)], [('princess',
0.7947244644165039), ('king', 0.7507690191268921), ('elizabeth',
0.7355712056159973), ('royal', 0.7065026164054871), ('lady',
0.7044796943664551)], [('prince', 0.7682329416275024), ('queen',
0.7507690787315369), ('son', 0.7020888328552246), ('brother',
0.6985775828361511), ('monarch', 0.6977890729904175)]]
```

[4]:
```
vec1 = glove_vectors.get_vector('king') - glove_vectors.get_vector('man') +␣
 ↪glove_vectors.get_vector('woman')
vec2 = glove_vectors.get_vector('vehicle') - glove_vectors.
 ↪get_vector('computer') + glove_vectors.get_vector('laptop')

print(glove_vectors.most_similar(vec1, topn=5))
print(glove_vectors.most_similar(vec2, topn=5))
```

```
[('king', 0.8551837205886841), ('queen', 0.7834413647651672), ('monarch',
0.6933802366256714), ('throne', 0.6833109855651855), ('daughter',
0.680908203125)]
[('vehicle', 0.758844256401062), ('suv', 0.7331708073616028), ('minivan',
0.720077633857727), ('parked', 0.7133616805076599), ('jeep',
0.6985746622085571)]
```

Performing vector arithmetic on words using methods like word embeddings (in this case GloVe) allows us to capture semantic relationships between words. The resulting vector would ideally have a corresponding word but since it is not like this we have to use functions like most_similar to find the closest word to that. In the first example 'monarch' which funnily enough would be the perfect corresponding vector since it is a gender neutral word representing royalty.

2. *INVESTIGATING SOCIETAL BIASES*

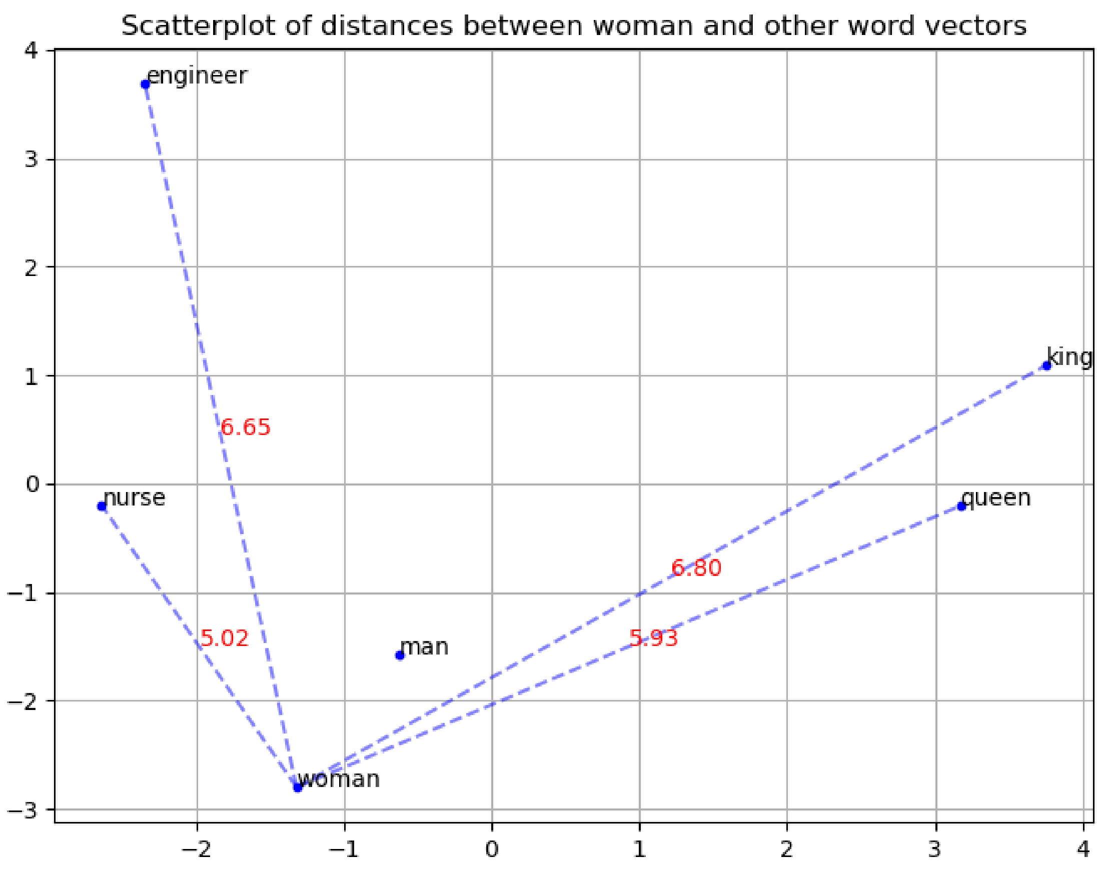
[5]:
```
professions = ['engineer', 'nurse', 'scientist']
names = ['james', 'emily', 'mohammed', 'ling', 'juan']
similarities = {}

for profession in professions:
    similarities[profession] = []
    for name in names:
        similarities[profession].append([name, float(glove_vectors.
 ↪similarity(profession, name))])

json_dict = json.dumps(similarities, indent=3)
print(json_dict)
```

```
{
   "engineer": [
      [
         "james",
         0.43955788016319275
      ],
      [
         "emily",
         0.15777018666267395
      ],
      [
         "mohammed",
         0.258592814207077
      ],
      [
         "ling",
         0.11636736989021301
      ],
      [
      "juan",
      0.27362167835235596
    ]
  ],
  "nurse": [
    [
      "james",
      0.23846927285194397
    ],
    [
      "emily",
      0.36251455545425415
    ],
    [
      "mohammed",
      0.1562427282333374
    ],
    [
      "ling",
      0.20062392950057983
    ],
    [
      "juan",
      0.16719740629196167
    ]
  ],
  "scientist": [
    [
      "james",
      0.4182989299297333
    ],
    [
      "emily",
      0.23490199446678162
    ],
    [
      "mohammed",
      0.28646785020828247
    ],
    [
      "ling",
      0.2678428888320923
    ],
    [
      "juan",
      0.18118484318256378
    ]
  ]
```

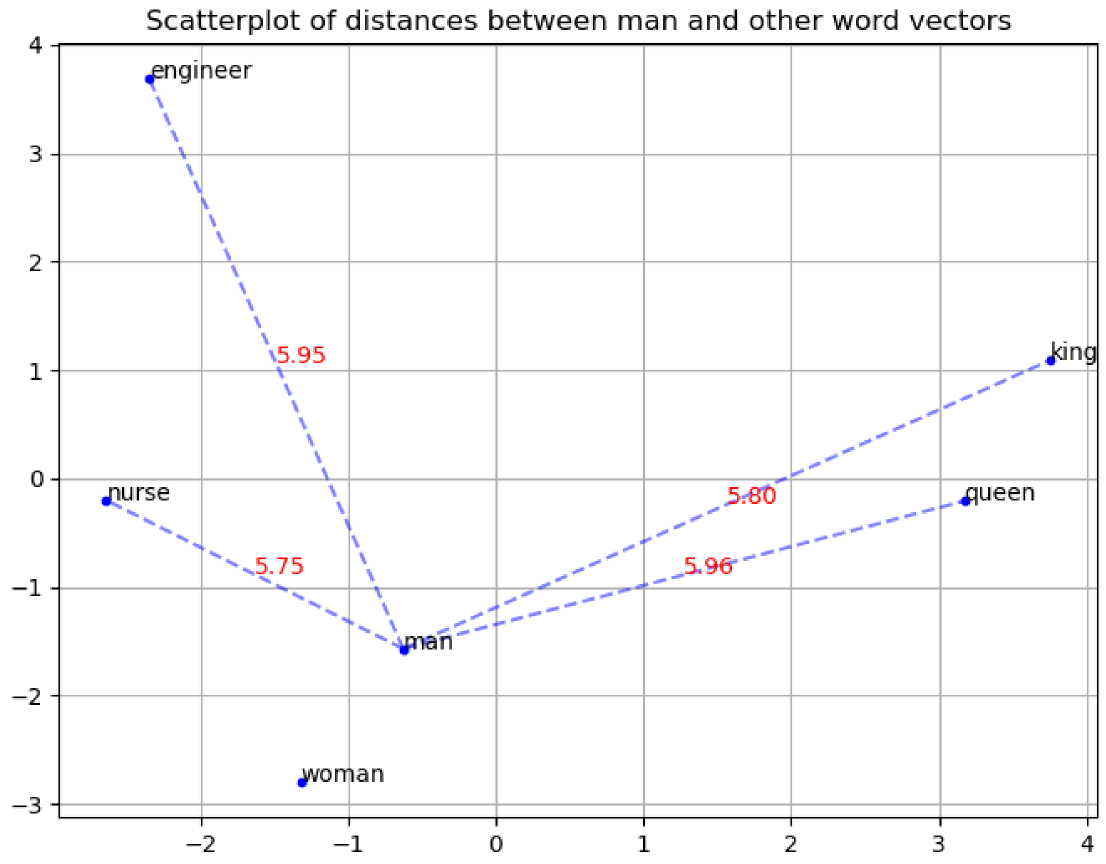
```
}
```

```
[6]: words = ["king", "queen", "man", "woman", "nurse", "engineer"]
word_vectors = np.array([glove_vectors[word] for word in words])

pca_model = PCA(n_components=2, random_state=42)
word_vectors_2d = pca_model.fit_transform(word_vectors)

professions = ["nurse", "engineer"]
non_professions = ["king", "queen", "woman"]


plt.figure(figsize=(8, 6))
plt.scatter(word_vectors_2d[:, 0], word_vectors_2d[:, 1], c='b', s=10)

#annotate points with word labels
for i, word in enumerate(words):
    plt.annotate(word, (word_vectors_2d[i, 0], word_vectors_2d[i, 1]))

woman_index = words.index("woman")
for word in words:
    if word != "woman" and word != "man":
        word_index = words.index(word)
        plt.plot([word_vectors_2d[woman_index, 0], word_vectors_2d[word_index,␣
 ↪0]],
                 [word_vectors_2d[woman_index, 1], word_vectors_2d[word_index,␣
 ↪1]],
                 color='blue', linestyle='--', alpha=0.5)

        distance = np.linalg.norm(word_vectors[woman_index] -␣
 ↪word_vectors[word_index])

        # Annotate the line with the distance
        plt.annotate(f'{distance:.2f}',
                     ((word_vectors_2d[woman_index, 0] +␣
 ↪word_vectors_2d[word_index, 0]) / 2,
                      (word_vectors_2d[woman_index, 1] +␣
 ↪word_vectors_2d[word_index, 1]) / 2),
                     color='red', fontsize=10, )

plt.title('Scatterplot of distances between woman and other word vectors')
plt.grid(True)
plt.show()

#lets do the same but for man

professions = ["nurse", "engineer"]
```

```
non_professions = ["king", "queen", "man"]

plt.figure(figsize=(8, 6))
plt.scatter(word_vectors_2d[:, 0], word_vectors_2d[:, 1], c='b', s=10)

#annotate points with word labels
for i, word in enumerate(words):
    plt.annotate(word, (word_vectors_2d[i, 0], word_vectors_2d[i, 1]))

man_index = words.index("man")
for word in words:
    if word != "woman" and word != "man":
        word_index = words.index(word)
        plt.plot([word_vectors_2d[man_index, 0], word_vectors_2d[word_index,␣
 ↪0]],
                  [word_vectors_2d[man_index, 1], word_vectors_2d[word_index,␣
 ↪1]],
                  color='blue', linestyle='--', alpha=0.5)

        distance = np.linalg.norm(word_vectors[man_index] -␣
 ↪word_vectors[word_index])

        # Annotate the line with the distance
        plt.annotate(f'{distance:.2f}',
                     ((word_vectors_2d[man_index, 0] +␣
 ↪word_vectors_2d[word_index, 0]) / 2,
                      (word_vectors_2d[man_index, 1] +␣
 ↪word_vectors_2d[word_index, 1]) / 2),
                     color='red', fontsize=10, )

plt.title('Scatterplot of distances between man and other word vectors')
plt.grid(True)
plt.show()
```



Scatterplot of distances between woman and other word vectors
engineer
king
nurse
queen
man
woman
6.65
6.80
5.02
5.93
4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4



These two plots illustrate the influence of biases on word vectors and how they can perpetuate stereotypes. In the second plot, we see the vector for “engineer” biased towards the vector for ‘man’ radically differently than to the vector for ‘woman’, reflecting a stereotype where men are more commonly associated with this profession. This bias can negatively impact systems relying on word embeddings, reinforcing gender stereotypes and potentially leading to discriminatory outcomes. Recognizing and mitigating biases in word embeddings is crucial for developing fair and equitable AI systems that avoid perpetuating harmful stereotypes.

3. *DISCUSSION*

Some implications that these biases imply are: - Reinforcing stereotypes: Word embeddings often reflect societal biases present in the data they are trained on. Historical data has shown that gender biases appear in different job roles. For example, we can see how the word ‘man’ is closer to ‘engineer’ than ‘woman’ is. This can lead to perpetuating stereotypes and discrimination in automated systems. - Biased search results: In search engines, biased word embeddings can lead to biased search results. For example, if a user searches for ‘engineer’, a system relying on embeddings like this will prioritize male engineers over female engineers due to the bias present. - Discriminatory Hiring Practices: In resume screening systems, biased word embeddings can lead to discriminatory hiring practices. If a system is trained on biased data, it may inadvertently favor candidates who fit certain demographic profiles encoded in the embeddings (i.e.: male candidates for engineering). - Limited representation: Biases in word embeddings can also resul in limited representation of certain groups or concepts. For instance, if certain occupations are underrepresented in the training data, their corresponding word embeddings may not accurately capture the nuances of those occupations, leading to misinterpretation or exclusion in applications. - Feedback loops: Finally, the biases present in word embeddings can create feedback loops where biased systems perpetuate and reinforce existing biases in society. For example, if a biased resumé screening system consistently selects candidates from certain demographic groups, it may further marginalize underrespresented groups.

EXERCISE 4 NATURAL LANGUAGE PROCESSING

1. Implement a method to predict the most likely next word for a given prefix by examining the last word in the prefix, and predicting the most likely next word, frequency wise (with regards to the given text corpus).

[7]:
```
def tokenize_text(file):
    tokenized_text = []
    with open(file, 'r') as file:
        text = file.read()
        tokenized_text = re.findall(r"[\w']+|[.,!?;]", text.lower())
        words = [word for word in tokenized_text if word.isalpha()]
    return words

tokens = tokenize_text('text_corpus.txt')

vocab = len(set(tokens))
bigram_list = list(nltk.bigrams(tokens))

cfreq_sherlock_bigrams = nltk.ConditionalFreqDist(bigram_list)

cprob_sherlock_bigrams = nltk.ConditionalProbDist(cfreq_sherlock_bigrams, nltk.
 ↪MLEProbDist)

prefixes = ['to', 'is', 'of']
for prefix in prefixes:
    print(f"Prefix: {prefix}")
    print(f"Next Probable word: {cprob_sherlock_bigrams[prefix].generate()}")
    print('-----------------------------------')
```

```
Prefix: to
Next Probable word: day
-----------------------------------
Prefix: is
Next Probable word: clear
-----------------------------------
Prefix: of
Next Probable word: the
-----------------------------------
```

```
[8]: trigram_list = nltk.trigrams(tokens)
     condition_pairs = (((w0, w1), w2) for w0, w1, w2 in trigram_list)
     cfreq_sherlock_trigrams = nltk.ConditionalFreqDist(condition_pairs)
     cprob_sherlock_trigrams = nltk.ConditionalProbDist(cfreq_sherlock_trigrams,␣
      ↪nltk.MLEProbDist)

     cprob_sherlock_trigrams['is', 'the'].generate()

     prefixes = [['me','to'], ['holmes', 'is'], ['mystery', 'of']]
     for prefix in prefixes:
         print(f"Prefix: {prefix}")
         print(f"Next Probable word: {cprob_sherlock_trigrams[prefix[0], prefix[1]].
      ↪generate()}")
         print('-----------------------------------')
```

```
Prefix: ['me', 'to']
Next Probable word: read
-----------------------------------
Prefix: ['holmes', 'is']
Next Probable word: it
-----------------------------------
Prefix: ['mystery', 'of']
Next Probable word: the
-----------------------------------
```

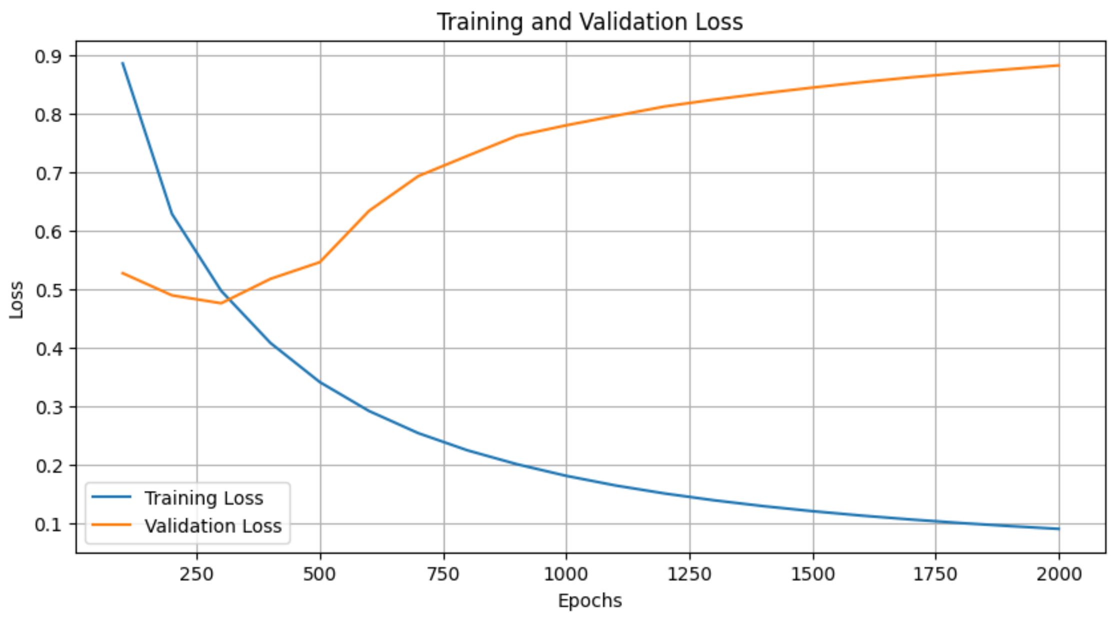
Implemented a bigram and a trigram model with nltk to predict word depending on the frequencies. From what we can see we can infer a few things, firstly that human language is complex and that longer ngrams tend to predict better than single unigrams, this is because human language is composed of very complex relationships between words, traces, and secondly that using 1, 2 or 3 words to determine the following word is quite undesiderable. Nonetheless, ngram models can be useful for several things but we do have to be careful on our use of them as they might be the better at making coherent predictions due to this lack of attention.

I could not manage to make the gpt-2 model work properly with the corpus data. I invested quite a lot of time but i kept getting this error “https://stackoverflow.com/questions/76448287/how-can-i-solve-importerror-using-the-trainer-with-pytorch-requires-accele” tried quite a lot but i still couldn’t get it neither to work on my machine nor on the GColab, tried changing CPU environment to GPU rendering as i saw it suggested as a possible solution for it but to no avail. I understand if i get deducted points but I just wanted to clarify that i tried to make it work ;)

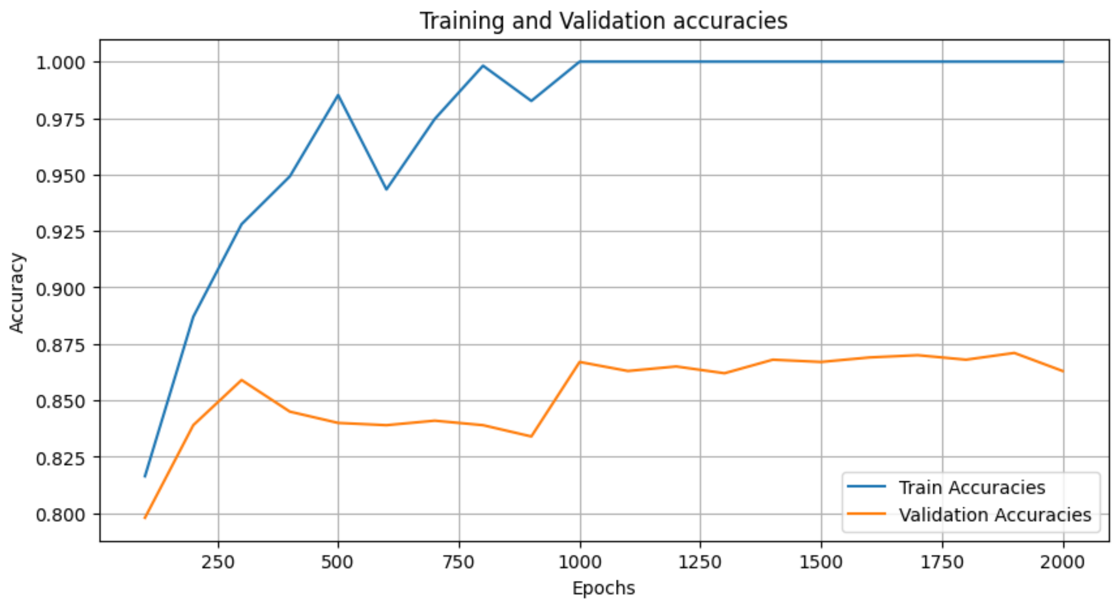
EXERCISE 5 COMPUTER VISION

(Code for this exercise is in the .ipynb called “Code for exercise 5”)

1) A plot containing the training and validation loss curves.



2. A plot showing the development of the training and validation accuracies throughout the training process.



3. Describe the plots created in steps above. Based on your observations, would you recommend changing the number of training epochs? If so, why?

- The plots generated exhibit a trend whereby the performance metrics plateau or exhibit diminishing returns after approximately 250 epochs. This phenomenon is indicative of the model's transition towards learning the intricacies of the training data itself rather than generalizing to make accurate predictions on unseen data. Consequently, it suggests that further increasing the number of training epochs may not significantly improve the model's predictive capacity but may instead lead to overfitting.

4. Report the test accuracy of your model at the best validation loss and plot the confusion matrix on the test dataset.

- The best test accuracy at the best validation loss is: 0.859000027179718

References - Chat GPT version 3.5 was used as a writing assistance tool.