RAMAIAH INSTITUTE OF TECHNOLOGY

MSR NAGAR, BENGALURU, 560054

A Report on

Trigram Language Model

Subject Of

Numerical Analysis with Python

*Submitted in partial fulfilment of the **other component** requirements as a part of the Data Structures Lab subject with code ISAEC393 for the III Semester of degree of **Bachelor of Engineering in Information Science and Engineering***

Submitted by

Candidate Names

Suchit G (1MS22IS138)

Uttam K N (1MS22IS147)

Under the Guidance of

Faculty In-charge

Ms. Kavya K S

Assistant Professor

Dept. of ISE

Department of Information Science and Engineering

Ramaiah Institute of Technology

2023 - 2024

Trigram Language Model

February 20, 2024

0.0.1 Language Models

Language Models (hereafter, LM(s)) can be considered as black boxes that, given an input, produce non-deterministic outputs, with inputs and outputs both being some element of natural language like syllables, words, phrases, etc. In this project, we present a character-level language model that's a stepping stone to modern LMs.

0.0.2 Introdution to Trigram Language Model

Trigrams are a special case of n-gram (n=3), where n-gram refers to a sequence of symbols in sequential order. Therefore, an n-gram LM can be considered as an LM that takes in n input characters and gives out 1 output character. Formally, an n-gram LM gives out the probability distribution of all the characters given the previous (n-1)-characters from which one character is sampled out.

A Trigram LM takes in the previous 2 characters as input and gives out a probability distribution of all the characters from which 1 character is sampled out.

0.0.3 A Brief Overview of the Libraries Used

NumPy NumPy is an open-source library that enables fast and efficient numerical computing in Python. We make use of NumPy to create the probability distribution over our dataset and also to sample from the said distribution.

PyTorch PyTorch is a powerful open-source machine learning library used for the development of deep learning models especially useful for it's automatic differentiation engine. We make use of PyTorch in the second notebook to build the Trigram LM using a Neural Network.

Matplotlib Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It's widely used to visualize and present data in general. We make use of this open-source library to show probability distribution heatmap, loss visualizations, etc.

Seaborn Seaborn is another visualization library built on top of Matplotlib. It provides high level abstractions over Matplotlib to create attractive statistical graphics.

0.0.4 Project Structure

We first present the statistical version of the Trigram LM. In the second notebook, we demonstrate the Trigram LM by modelling the probability distribution table (as shown in the first notebook)

using a Neural Network. An empirical observation then follows that both the models are nearly equivalent.

As mentioned before, we also provide sufficient headings and comments documenting our code.

Disclaimer: Every line of code, and documentation is written by us, humans. No content has been directly copy-pasted from an LLM, or any other source, for that matter, except for understanding certain library syntax, and checking factual accuracy.

1 Statistical Trigram

1.1 Data Preparation

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

import seaborn as sns
```

[2]:
```
words = open("./names.txt", 'r').read().splitlines()
```

[3]:
```
words[:4]
```

[3]:
```
['emma', 'olivia', 'ava', 'isabella']
```

[4]:
```
chars = sorted(list(set(''.join(words))))
stoi = {s:i+1 for i,s in enumerate(chars)}
stoi['.'] = 0
itos = {i:s for s,i in stoi.items()}
```

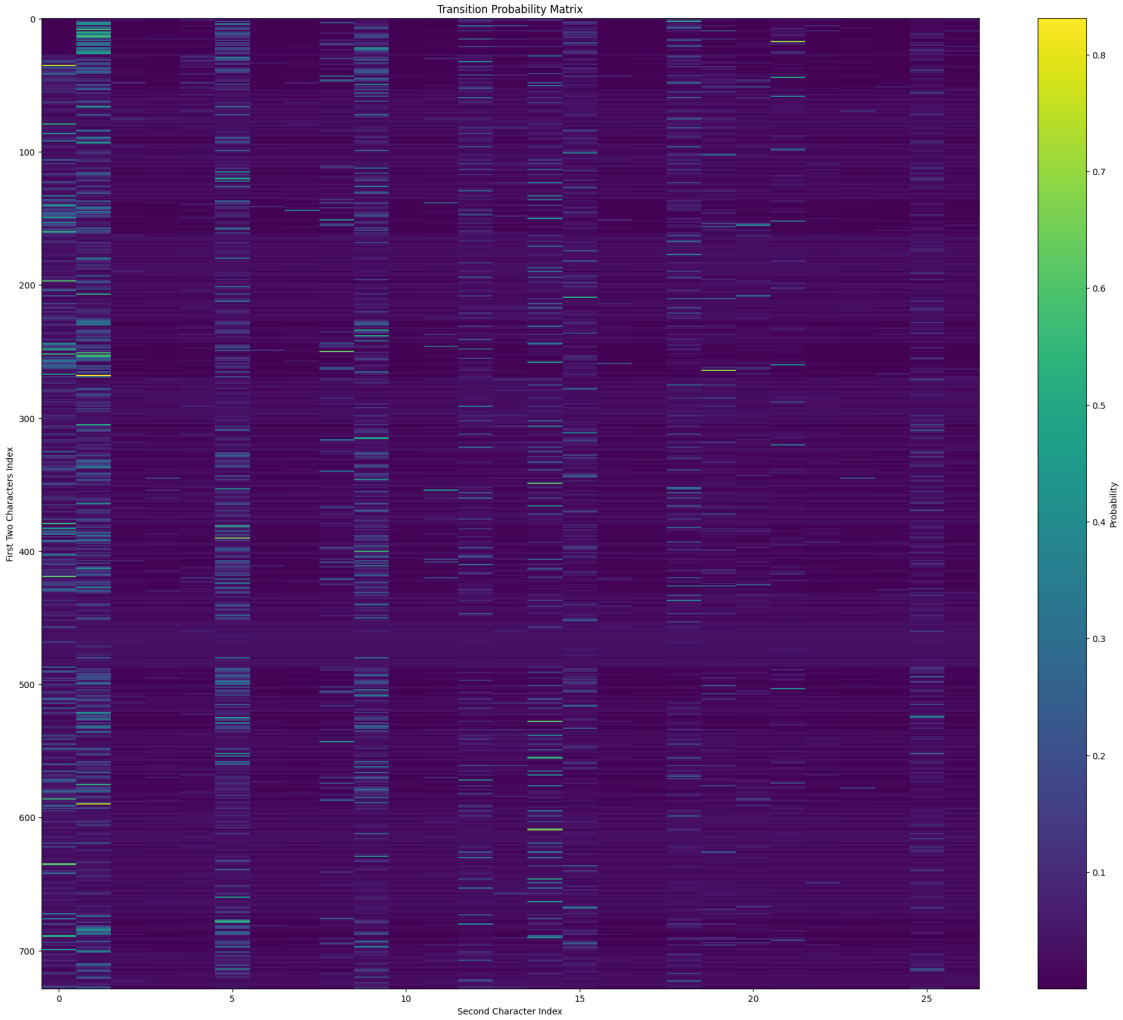
[5]:
```
two_chars = set()
for c1 in chars+["."]:
  for c2 in chars+["."]:
    two_chars.add(c1+c2)

two_chars = sorted(list(two_chars))

stoi2 = {s:i for i,s in enumerate(two_chars)}
itos2 = {i:s for i,s in enumerate(two_chars)}
```

1.2 Creating the Probability Distribution Matrix

[6]:
```
N = np.zeros((729, 27), dtype=np.float32)

# model smoothing
N += 1

# populating matrix with trigram counts
```

```
for w in words:
    chs = ["."] + list(w) + ["."]
    for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
        ix1 = stoi2[ch1+ch2]
        ix2 = stoi[ch3]
        N[ix1, ix2] += 1

# creating the probability distribution
P = N
P /= P.sum(1, keepdims=True)
```

[7]:
```
plt.figure(figsize=(24, 20))
plt.imshow(P, cmap='viridis', aspect='auto', interpolation='nearest')

cbar = plt.colorbar()
cbar.set_label('Probability')

plt.xlabel("Second Character Index")
plt.ylabel("First Two Characters Index")
plt.title("Transition Probability Matrix")

plt.show()
```



1.3 Sampling from the model

[8]:
```
g = np.random.default_rng(seed=2147483647)

# sequentially sampling from the probability distribution
for _ in range(10):
    ix = 1
    out = ["."]
    while True:
        if ix != 1:
            ix = stoi2[''.join(out[-2:])]
        p = P[ix]
        ix = g.multinomial(1, p, size=1).argmax()
        out.append(itos[ix])
        if out[-1][-1] == ".":
```

```
            break

    print(''.join(out))
```

```
.reydechadane.
.den.
.valoick.
.esmaus.
.zaroniellyn.
.desaydevarindrekrit.
.muhwftquissadvhtfendshabby.
.brae.
.lah.
.leilaleeven.
```

1.4 Finding the average NLL with different dataset sizes

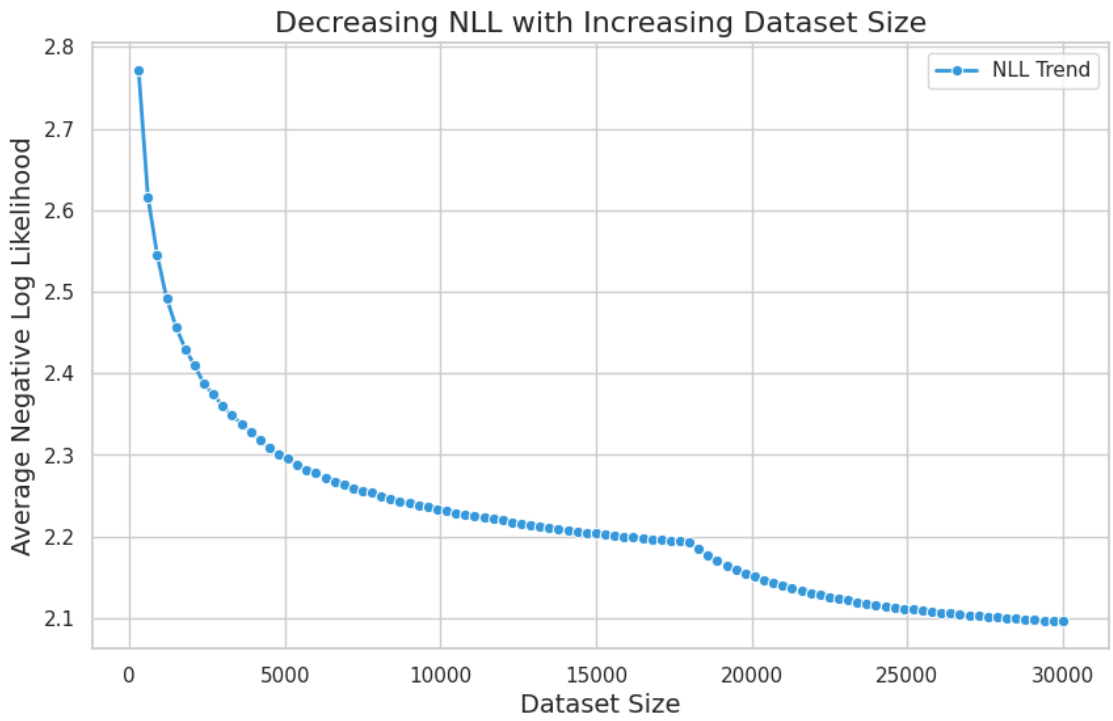
Here we experiment with varying sizes of the dataset to see its correlation with the negative log likelihood. As expected, increasing dataset size gives a better loss.

[9]:
```
combs = np.arange(300,30001,300)
```

[10]:
```
avg_nll = {}
for comb in combs:
    N = np.zeros((729, 27), dtype=np.float32)

    # model smoothing
    N += 1

    for w in words[:comb]:
        chs = ["."] + list(w) + ["."]
        for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
            ix1 = stoi2[ch1+ch2]
            ix2 = stoi[ch3]
            N[ix1, ix2] += 1

    P = N
    P /= P.sum(1, keepdims=True)

    n = 0
    log_likelihood = 0.0
    for w in words:
        chs = ["."] + list(w) + ["."]
        for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
            ix1 = stoi2[ch1+ch2]
            ix2 = stoi[ch3]
            prob = P[ix1, ix2]
            logprob = np.log(prob)
```

```
            log_likelihood += logprob
            n += 1

    nll = -log_likelihood
    avg_nll[comb] = (nll/n)
```

[11]:
```
sns.set(style="whitegrid")

plt.figure(figsize=(10, 6))
sns.lineplot(x=list(avg_nll.keys()), y=list(avg_nll.values()), marker='o',␣
 ↪color='#3498db', linewidth=2, label='NLL Trend')

plt.xlabel("Dataset Size", fontsize=14)
plt.ylabel("Average Negative Log Likelihood", fontsize=14)
plt.title("Decreasing NLL with Increasing Dataset Size", fontsize=16)

plt.legend()
plt.show()
```



NB: The loss increases at a slightly higher rate after the dataset crosses the size fo 17k.

1.5 Optimal Model Smoothing

We now experiment with different values for the model smoothing factor. We can observe that any value over 1 gives a higher loss.

[12]:
```
nums = np.arange(1, 11)
nums
```

[12]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

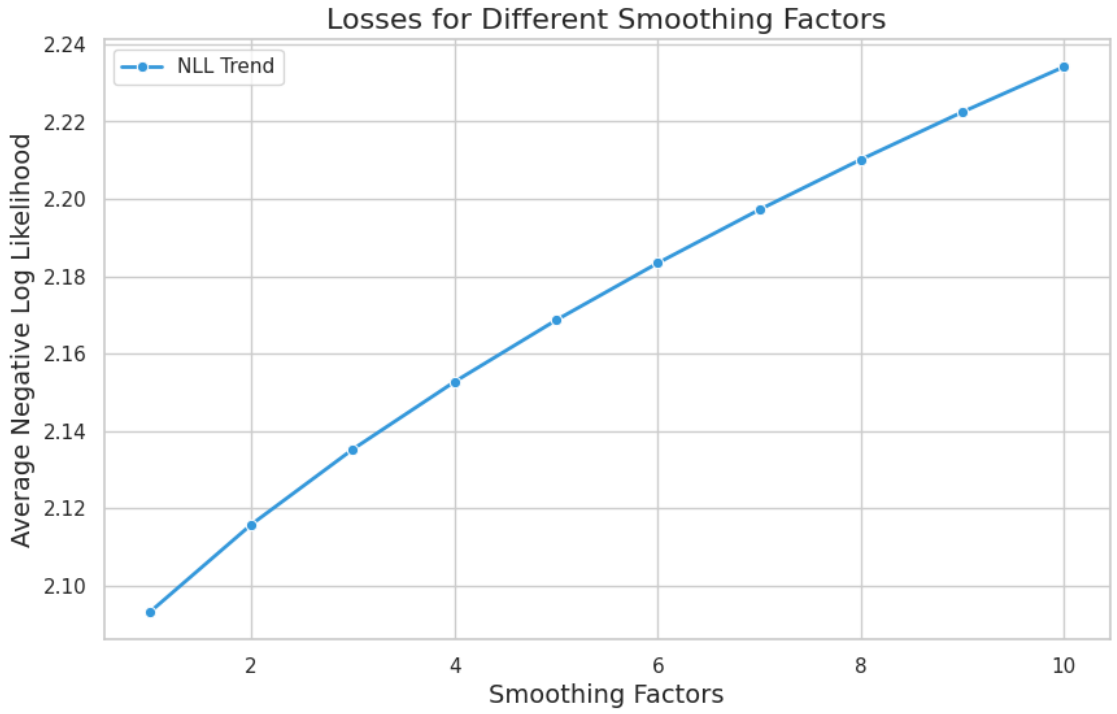
[13]:
```
smdl_nll = {}
for num in nums:
    N = np.zeros((729, 27), dtype=np.float32)
    N += num
    for w in words:
        chs = ["."] + list(w) + ["."]
        for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
            ix1 = stoi2[ch1+ch2]
            ix2 = stoi[ch3]
            N[ix1, ix2] += 1

    P = N
    P /= P.sum(1, keepdims=True)

    n = 0
    log_likelihood = 0.0
    for w in words:
        chs = ["."] + list(w) + ["."]
        for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
            ix1 = stoi2[ch1+ch2]
            ix2 = stoi[ch3]
            prob = P[ix1, ix2]
            logprob = np.log(prob)
            log_likelihood += logprob
            n += 1
    nll = -log_likelihood
    smdl_nll[num] = (nll/n).item()
```

[14]:
```
smdl_nll
```

[14]:
```
{1: 2.0930798649518163,
 2: 2.1157624440037672,
 3: 2.135246232745865,
 4: 2.1526845002373696,
 5: 2.1686244652998914,
 6: 2.1833893669899944,
 7: 2.19719318444803,
 8: 2.210188028218687,
```

```
 9: 2.2224874687609204,
 10: 2.234179399642144}
```

[15]:
```
sns.set(style="whitegrid")

plt.figure(figsize=(10, 6))

sns.lineplot(x=list(smdl_nll.keys()), y=list(smdl_nll.values()), marker='o',␣
 ↪color='#3498db', linewidth=2, label='NLL Trend')

plt.xlabel("Smoothing Factors", fontsize=14)
plt.ylabel("Average Negative Log Likelihood", fontsize=14)
plt.title("Losses for Different Smoothing Factors", fontsize=16)

plt.legend()
plt.show()
```



[ ]:

[ ]:

1 Trigram using Neural Network

[1]:
```
import torch
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.model_selection import train_test_split
```

1.1 Data Prep

[2]:
```
words = open("./names.txt", 'r').read().splitlines()
```

[3]:
```
train_words, temp_words = train_test_split(words, train_size=0.8,␣
 ↪random_state=42)
dev_words, test_words = train_test_split(temp_words, test_size=0.5,␣
 ↪random_state=42)
```

[4]:
```
len(train_words), len(dev_words), len(test_words)
```

[4]:
```
(25626, 3203, 3204)
```

[5]:
```
chars = sorted(list(set(''.join(train_words))))

stoi = {s:i+1 for i,s in enumerate(chars)}
stoi['.'] = 0
itos = {i:s for s,i in stoi.items()}

two_chars = set()
for c1 in chars+["."]:
    for c2 in chars+["."]:
        two_chars.add(c1+c2)

two_chars = sorted(list(two_chars))

stoi2 = {s:i for i,s in enumerate(two_chars)}
itos2 = {i:s for i,s in enumerate(two_chars)}
```

```
N = torch.zeros((729, 27), dtype=torch.int32)
```

1.2 Trigram NN

[6]:
```
xs, ys = [], []
for w in train_words:
    chs = ["."] + list(w) + ["."]
    for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
        ix1 = stoi2[ch1+ch2]
        ix2 = stoi[ch3]
        xs.append(ix1)
        ys.append(ix2)

xs = torch.tensor(xs)
ys = torch.tensor(ys)

W = torch.empty(0)
```

[7]:
```
dev_xs, dev_ys = [], []
for w in dev_words:
    chs = ["."] + list(w) + ["."]
    for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
        ix1 = stoi2[ch1+ch2]
        ix2 = stoi[ch3]
        dev_xs.append(ix1)
        dev_ys.append(ix2)

dev_xs = torch.tensor(dev_xs)
dev_ys = torch.tensor(dev_ys)
```

[8]:
```
test_xs, test_ys = [], []
for w in test_words:
    chs = ["."] + list(w) + ["."]
    for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
        ix1 = stoi2[ch1+ch2]
        ix2 = stoi[ch3]
        test_xs.append(ix1)
        test_ys.append(ix2)

test_xs = torch.tensor(test_xs)
test_ys = torch.tensor(test_ys)
```

1.2.1 Training

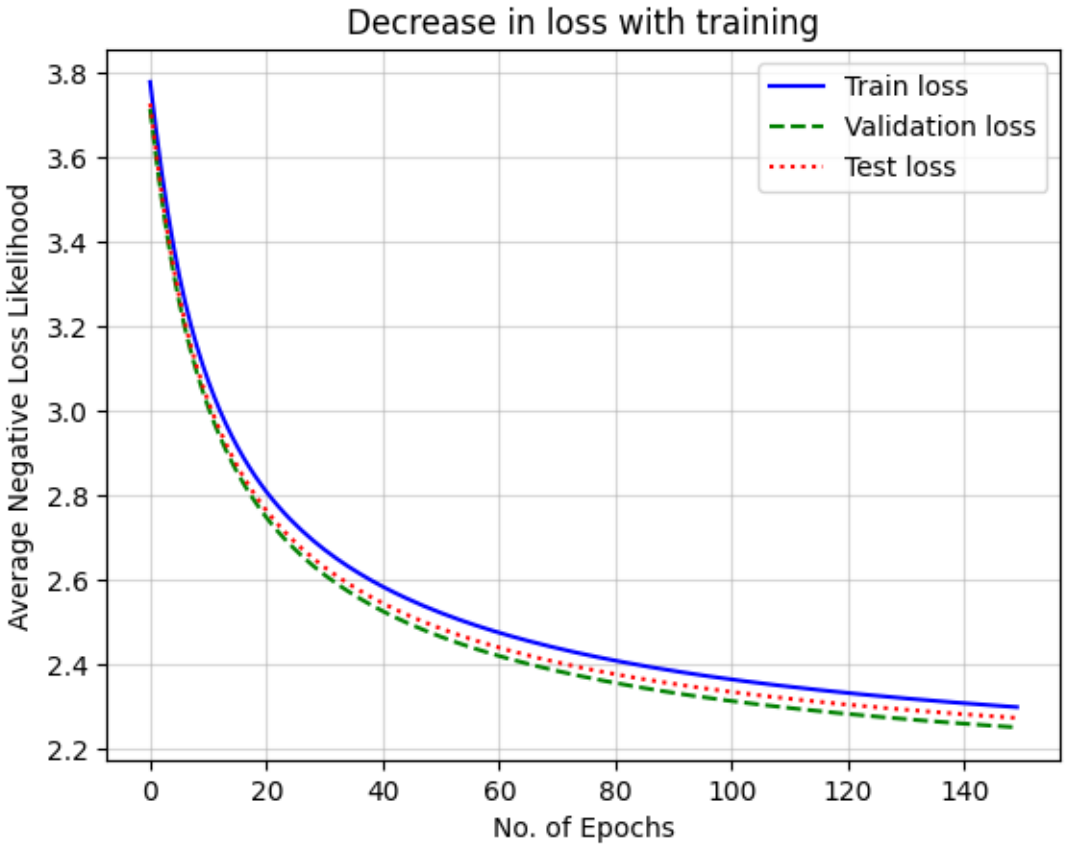
In comparison to the statistical model, the neural network weights are essentially trained to represent the dataset's probability distribution.

[9]:
```
def get_epoch_loss(weights, word_set):
    if (word_set == "dev_words"):
        xs = torch.tensor(dev_xs)
        ys = torch.tensor(dev_ys)
    else:
        xs = torch.tensor(test_xs)
        ys = torch.tensor(test_ys)

    with torch.no_grad():
        logits = weights[xs]
        counts = logits.exp()
        probs = counts / counts.sum(1, keepdim=True)

        nll = -probs[torch.arange(xs.nelement()), ys].log().mean()

    return nll.item()
```

[10]:
```
def train(reg_factor=0.0550, epochs=150):
    global W
    g = torch.Generator().manual_seed(2147483647)
    W = torch.randn((729, 27), generator=g, requires_grad=True)

    train_losses = {}
    dev_losses = {}
    test_losses = {}

    for i in range(epochs):
        # forward pass
        logits = W[xs]
        counts = logits.exp()
        probs = counts / counts.sum(1, keepdim=True)
        loss = -probs[torch.arange(xs.nelement()), ys].log().mean() +␣
 ↪reg_factor*(W**2).mean()

        train_losses[i] = loss.item()
        dev_losses[i] = get_epoch_loss(W, "dev_words")
        test_losses[i] = get_epoch_loss(W, "test_words")

      # backward pass
        W.grad = None
        loss.backward()
        with torch.no_grad():
            W.data += -75 * W.grad

    return (train_losses, dev_losses, test_losses)
```

[ ]:
```
train_losses, dev_losses, test_losses = train();
```

[12]:
```
plt.plot(train_losses.keys(), train_losses.values(), 'b-', label="Train loss")
plt.plot(dev_losses.keys(), dev_losses.values(), 'g--', label="Validation loss")
plt.plot(test_losses.keys(), test_losses.values(), 'r:', label="Test loss")
plt.xlabel("No. of Epochs")
plt.ylabel("Average Negative Loss Likelihood")
plt.legend()
plt.title("Decrease in loss with training")
plt.grid(alpha=0.5)
plt.show()
```



1.2.2 Sampling from the Neural Network Model

In our tests, using the same seed (from the same library) for sampling in the statistical model and the neural network model gave nearly the same outputs. We do not show that here because we have used NumPy in the earlier notebook in view of demonstrating usage of multiple libraries.

[13]:
```
g = torch.Generator().manual_seed(2147483647)
for _ in range(10):
  ix = 1
  out = ["."]
```

```
  while True:
    if ix != 1:
      ix = stoi2[''.join(out[-2:])]
    with torch.no_grad():
        logits = W[ix]
        counts = logits.exp()
        probs = counts / counts.sum()

    ix = torch.multinomial(probs, num_samples=1, generator=g).item()
    out.append(itos[ix])

    if out[-1][-1] == ".":
      break

  print(''.join(out))
```

```
.luwjde.
.ilyasiz.
.ufofyywocnzq.
.di.
.ritoniabraree.
.viameiauriniadvhassdbyainrwibwlassiyanaylartleigvmumtrifoetumj.
.nonn.
.lenariani.
.rose.
.yae.
```

1.2.3 Evaluation

[14]:
```
def get_loss(word_set):
    xs_t, ys_t = [], []
    for w in word_set:
        chs = ["."] + list(w) + ["."]
        for ch1,ch2,ch3 in zip(chs, chs[1:], chs[2:]):
            ix1 = stoi2[ch1+ch2]
            ix2 = stoi[ch3]
            xs_t.append(ix1)
            ys_t.append(ix2)

    xs_t = torch.tensor(xs_t)
    ys_t = torch.tensor(ys_t)

    with torch.no_grad():
        logits = W[xs_t]
        counts = logits.exp()
        probs = counts / counts.sum(1, keepdim=True)

        nll = -probs[torch.arange(xs_t.nelement()), ys_t].log().mean()
```

```
    return nll.item()
```

[15]:
```
train_nll = get_loss(train_words)
print("--- Train split loss ---")
print(f"Avg NLL: {train_nll:.4f}")
```

```
--- Train split loss ---
Avg NLL: 2.2437
```

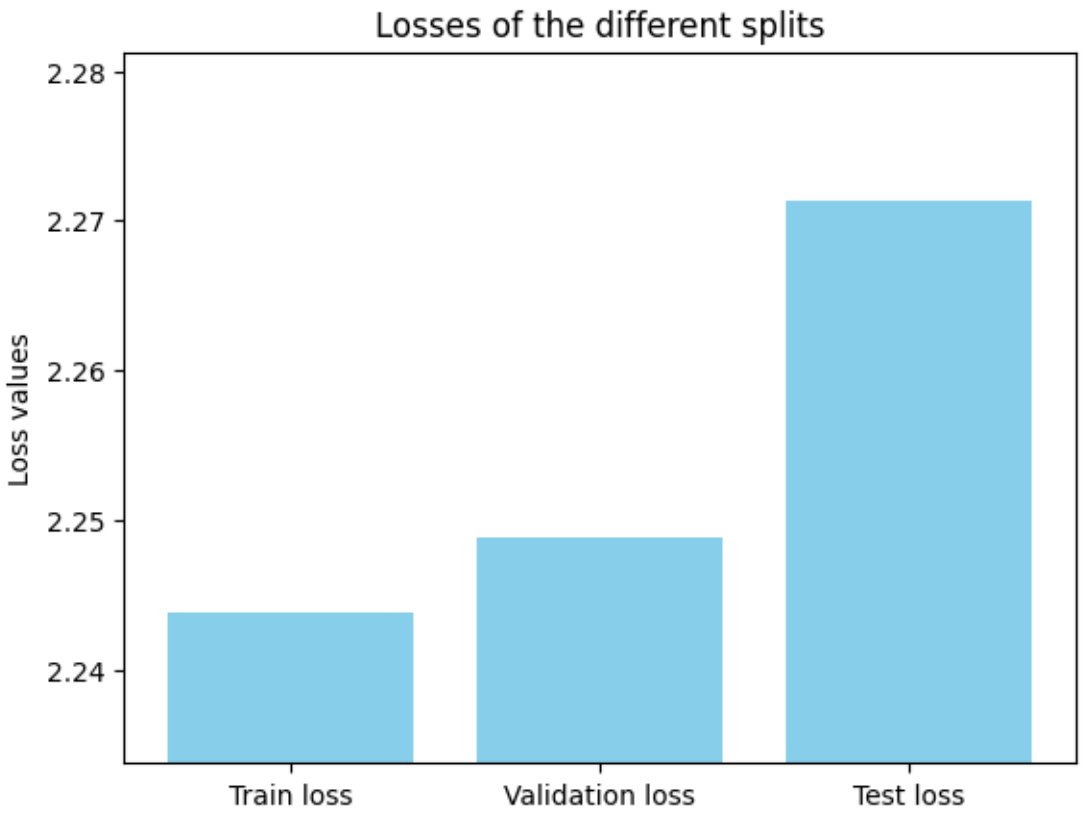
[16]:
```
dev_nll = get_loss(dev_words)
print("--- Dev split loss ---")
print(f"Avg NLL: {dev_nll:.4f}")
```

```
--- Dev split loss ---
Avg NLL: 2.2488
```

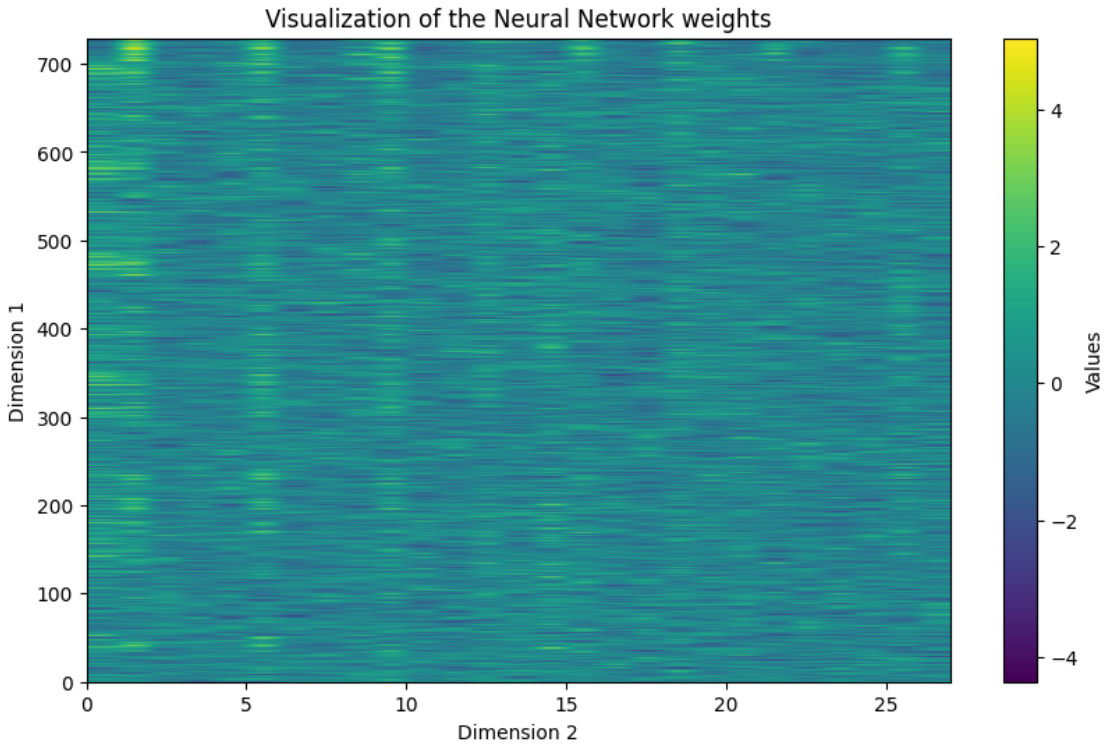
[17]:
```
test_nll = get_loss(test_words)
print("--- Test split loss ---")
print(f"Avg NLL: {test_nll:.4f}")
```

```
--- Test split loss ---
Avg NLL: 2.2713
```

[18]:
```
split_losses = [train_nll, dev_nll, test_nll]
loss_names = ["Train loss", "Validation loss", "Test loss"]

plt.bar(loss_names, split_losses, color="skyblue")
plt.ylim(min(split_losses) - 0.01, max(split_losses)+0.01)

plt.ylabel("Loss values")
plt.title("Losses of the different splits")
plt.show()
```



[19]:
```
plt.figure(figsize=(10, 6))
plt.imshow(W.detach(), cmap='viridis', aspect='auto', extent=[0, 27, 0, 729])
plt.colorbar(label='Values')

plt.xlabel('Dimension 2')
plt.ylabel('Dimension 1')
plt.title('Visualization of the Neural Network weights')

plt.show()
```



[ ]:

[ ]:

References

Karpathy, A., [Andrej Karpathy]. (2022, September 8). The spelled-out intro to language modeling: building makemore [Video]. YouTube. https://www.youtube.com/watch?v=PaCmpygFfXo

Jurafsky, Daniel and James H. Martin. N-Gram Language Models. 3 Feb. 2023, https://web.stanford.edu/~jurafsky/slp3/3.pdf.

Appendix

To further demonstrate the empirical equivalence of the statistical and the neural network model, we refer you to the video provided in the first reference source.