ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Projeto 1

Ambiente Virtuais Interativos e Inteligentes

Trabalho realizador por:

Duarte Valente | A47657

João Valido | A51090

Docente:

Eng. Arnaldo Abrantes

Curso: MEIM

2023

Índice

Índice de ilustrações

1.Introdução

a) Descrição

Este trabalho tinha como objetivo introduzir o tema da IA Generativa, isto é, ferramentas com capacidade para produzir conteúdos novos, como textos, imagens, vídeos, etc., a partir de modelos de ML, treinados sobre um grande conjunto de dados. Tem ainda como objetivo refletir acerca do impacto destas ferramentas de IA na sociedade, nomeadamente nas indústrias de entretenimento e jogos.

b) Âmbito

Este relatório insere-se no âmbito da realização da ficha laboratorial unidade curricular de Ambientes Virtuais Interativos e Inteligentes, do Mestrado em Engenharia Informática e Multimédia do DEETC do ISEL.

c) Documentação

Documentação de apoio à unidade curricular de Ambientes Virtuais Interativos e Inteligentes.

A. Mãos na massa

1. Modelos baseados em bigramas

a) Contagens no conjunto de treino

- **criar lista de todos os bigramas encontrados no conjunto de nomes e as respetivas vezes que ocorrem**

```
# criar lista de todos os bigramas encontrados no conjunto de nomes e as respetivas vezes que ocorrem
b = {}
for w in words:
  w = ['<s>'  ]+ list (w) +['<e>']
  for ch1, ch2 in zip(w, w[1:]):
    bigram = (ch1,ch2)
    b[bigram] = b.get(bigram, 0) + 1
```

Figura 1 - Criação de bigramas

- **criar a lista de todos os caracteres "chars", fazer as correspondências para inteiros "stoi", fazer a correspondência para string "itos"**

```
# criar a lista de todos os caracteres "chars",
# fazer as correspondencia para inteiros "stoi", fazer a correspondencia para string "itos"
chars = sorted(list(set(''.join(words))))
chars = ['.'] + chars
stoi = {s:i for i,s in enumerate(chars)}
itos = {i:s for i,s in enumerate(chars)}
```

Figura 2 - Criação de variáveis

- **para cada palavra da lista de nomes, preencher a matriz N com o número de vezes que cada bigrama aparece**

```
# para cada palavra da lista de nomes, preencher a matriz N com o numero de vezes que cada bigrama aparece
for w in words:
  w = ['.'  ]+ list (w) +['.']
  for ch1, ch2 in zip(w, w[1:]):
    ix1 = stoi[ch1]
    ix2 = stoi[ch2]
    N[ix1,ix2] += 1
```

Figura 3 - Contagem de bigramas

- **gerar 5 palavras com base na probabilidade da próxima letra a ser gerada**

```
# gerar 5 palavras com base na probabilidade da proxima letra a ser gerada
g = torch.Generator().manual_seed(123789)
for i in range(5):
  out = []
  ix = stoi['.']
  while True:
    pp = p[ix]
    ix = torch.multinomial(pp, num_samples=1, replacement=True, generator=g).item()
    out.append(itos[ix])
    if ix == 0:
      break
  print(''.join(out))
```

Figura 4 - Código para gerar palavras

```
nariel.
sa.
sy.
a.
welesty.
```

Figura 5 - Palavras obtidas

```
lossi = []
nll = 0
n = 0
for w in words:
  w = ['.'  ]+ list (w) +['.']
  for ch1, ch2 in zip(w, w[1:]):
    ix1 = stoi[ch1]
    ix2 = stoi[ch2]
    logprob = torch.log(p[ix1,ix2])
    nll -= logprob
    n += 1
  lossi.append(nll/n)
print(nll/n)
```

Figura 6 - Cálculo do loss

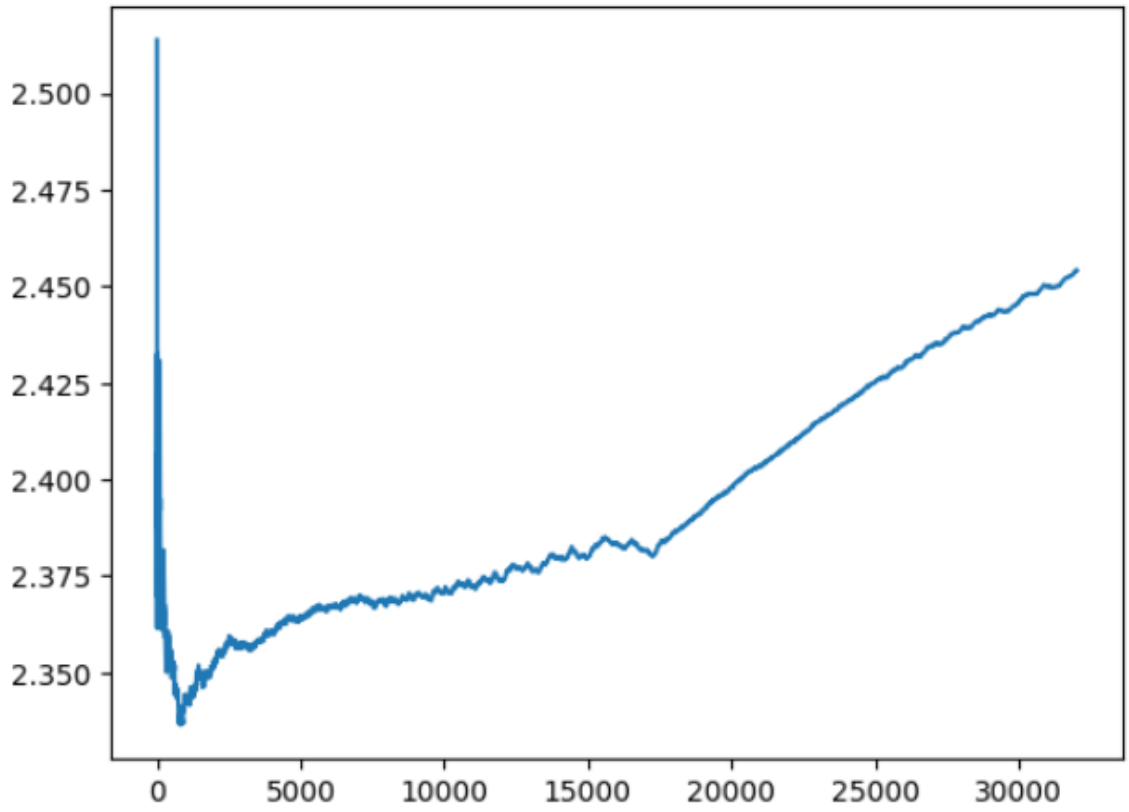

Figura 7 - Demonstração gráfica da aprendizagem do modelo

b) Otimização do critério de ML

- **criar lista de todos os bigramas encontrados no conjunto de nomes e as respetivas vezes que ocorrem**

```
1 # criar lista de todos os bigramas encontrados no conjunto de nomes e as respetivas vezes que ocorrem
2 b = {}
3 for w in words:
4   w = ['<s>'  ]+ list (w) +['<e>']
5   for ch1, ch2 in zip(w, w[1:]):
6     bigram = (ch1,ch2)
7     b[bigram] = b.get(bigram, 0) + 1
```

Figura 8 - Criação de bigramas

- **criar a lista de todos os caracteres "chars", fazer as correspondências para inteiros "stoi", fazer a correspondência para string "itos"**

```
# criar a lista de todos os caracteres "chars",
# fazer as correspondencia para inteiros "stoi", fazer a correspondencia para string "itos"
chars = sorted(list(set(''.join(words))))
chars = ['.'] + chars
stoi = {s:i for i,s in enumerate(chars)}
itos = {i:s for i,s in enumerate(chars)}
```

Figura 9 - Criação de variáveis

- **criar o modelo de treino**

```
# criar o modelo de treino
xs, ys =[], []
for w in words:
  chs = ['.' ] + list(w) + ['.']
  for ch1, ch2 in zip(chs, chs[1:]):
    ix1 = stoi[ch1]
    ix2 = stoi[ch2]
    xs.append(ix1)
    ys.append(ix2)
xs = torch.tensor(xs)
ys = torch.tensor(ys)
num = xs.shape[0]
```

Figura 10 - Criar modelo de treino

- **Treino do modelo**

```
# treino do modelo

for k in range(1000):
  xenc = F.one_hot(xs, num_classes=27).float()
  logits = xenc @ W
  counts = logits.exp()
  probs = counts / counts.sum(1, keepdim = True)

  loss = -probs[torch.arange(xs.shape[0]), ys].log().mean()
  lossi.append(loss.item())
  W.grad = None
  loss.backward()

  W.data += -10*W.grad

print (loss)
```

Figura 11 - Treino do modelo

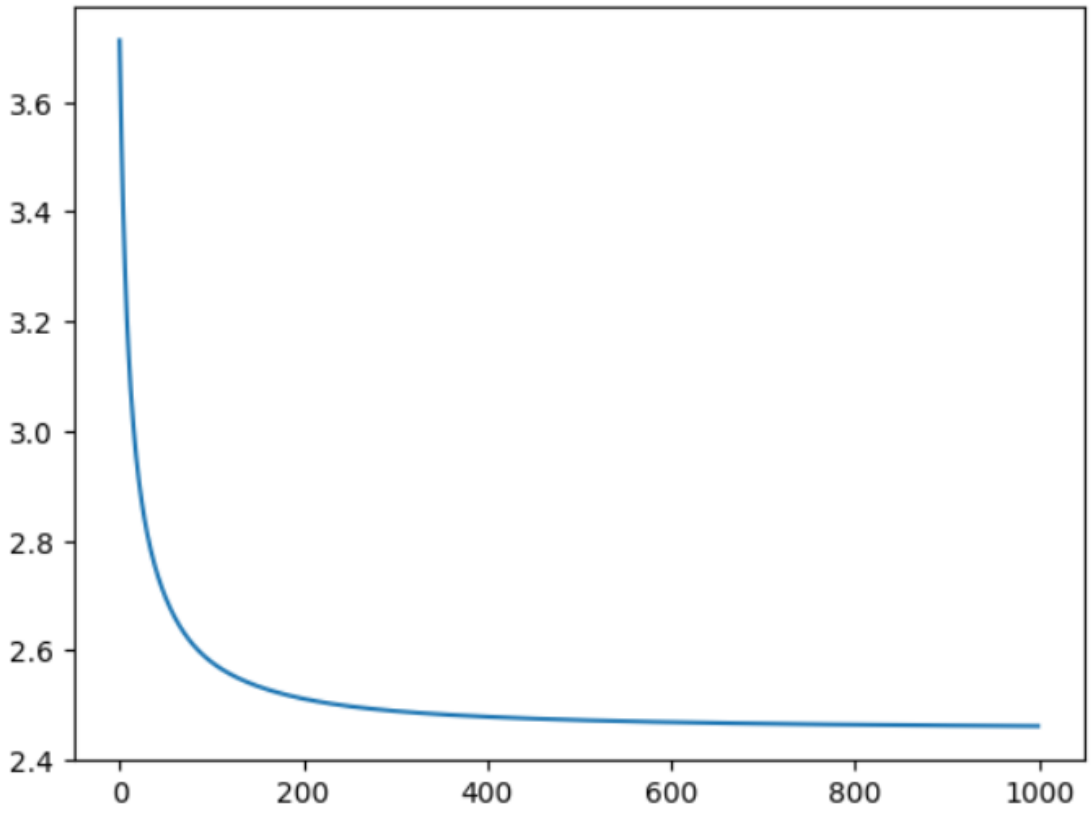

Figura 12 - Demonstração gráfica da aprendizagem do modelo

2. Modelo MLP

- **criar a lista de todos os caracteres "chars", fazer as correspondências para inteiros "stoi", fazer a correspondência para string "itos"**

```
# criar a lista de todos os caracteres "chars",
# fazer as correspondencia para inteiros "stoi", fazer a correspondencia para string "itos"
chars = sorted(list(set(''.join(words))))
chars = ['.'] + chars
stoi = {s:i for i,s in enumerate(chars)}
itos = {i:s for s,i in stoi.items()}
```

Figura 13 - criação de variáveis

- **Definir dataset para treino, validação e teste, 80%, 10%, 10% respetivamente**

```
block_size = 3 # context length: how many characters do we take to predict the next one?

def build_dataset(words):
  X, Y = [], []
  for w in words:

    #print(w)
    context = [0] * block_size
    for ch in w + '.':
      ix = stoi[ch]
      X.append(context)
      Y.append(ix)
      #print(''.join(itos[i] for i in context), '--->', itos[ix])
      context = context[1:] + [ix] # crop and append

  X = torch.tensor(X)
  Y = torch.tensor(Y)
  print(X.shape, Y.shape)
  return X, Y

random.seed(42)
random.shuffle(words)
n1 = int(0.8*len(words))
n2 = int(0.9*len(words))

Xtr, Ytr = build_dataset(words[:n1])
Xdev, Ydev = build_dataset(words[n1:n2])
Xte, Yte = build_dataset(words[n2:])
```

Figura 14 - Definir datasets para treino, validação e teste

```
g = torch.Generator().manual_seed(2461359) # for reproducibility
C = torch.randn((27, 3), generator=g) # embedding space
W1 = torch.randn((9, 1000), generator=g)
b1 = torch.randn(1000, generator=g)
W2 = torch.randn((1000, 27), generator=g)
b2 = torch.randn(27, generator=g)
parameters = [C, W1, b1, W2, b2]
for p in parameters:
  p.requires_grad = True
```

Figura 15 - Definição de parâmetros e weights e biases

```
for i in range(10000):
  # minibatches
  ix = torch.randint(Xtr.shape[0], (32,))

  # forward pass
  emb = C[Xtr[ix]]
  h = torch.tanh(emb.view(-1, 9) @ W1 + b1)
  logits = h @ W2 + b2
  loss = F.cross_entropy(logits, Ytr[ix])
  #print(i, loss.item())
  lossi.append(loss.item())

  # backward pass
  for p in parameters:
    p.grad = None
  loss.backward()

  # update
  for p in parameters:
    p.data += -0.1 * p.grad

print(loss.item())
```

```
1.9709446430206299
```

Figura 16 - Treino do modelo

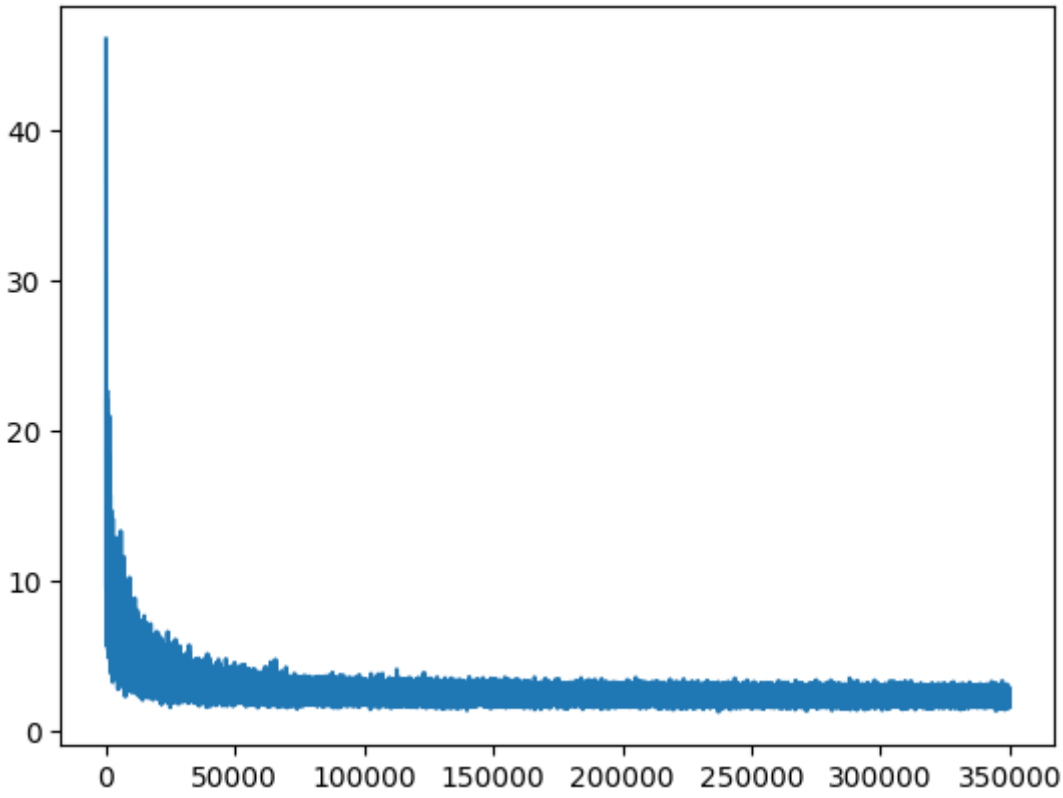

Figura 17 - Demonstração gráfica da aprendizagem do modelo

- **Através das avaliações dos datasets podemos verificar que o resultado da validação é um pouco maior que o resultado do treino, logo o modelo não se encontra em overfit**

```
# Evaluate in the training set
emb = C[Xtr]
h = torch.tanh(emb.view(-1, 9) @ W1 + b1)
logits = h @ W2 + b2
loss = F.cross_entropy(logits, Ytr)
loss
```

```
tensor(2.2858, grad_fn=<NllLossBackward0>)
```

```
# Evaluate in the validation set (is the model overfitting?)
emb = C[Xdev]
h = torch.tanh(emb.view(-1, 9) @ W1 + b1)
logits = h @ W2 + b2
loss = F.cross_entropy(logits, Ydev)
loss
```

```
tensor(2.3150, grad_fn=<NllLossBackward0>)
```

```
# Evaluate in the test set
emb = C[Xte]
h = torch.tanh(emb.view(-1, 9) @ W1 + b1)
logits = h @ W2 + b2
loss = F.cross_entropy(logits, Yte)
loss
```

```
tensor(2.3329, grad_fn=<NllLossBackward0>)
```

Figura 18 - Avaliação dos datasets

```python
# Samplimg from the model
g = torch.Generator().manual_seed(2461359)

for _ in range(20):

    out = []
    context = [0] * block_size # initialize with all ...
    while True:
      emb = C[torch.tensor([context])] # (1,block_size,d)
      h = torch.tanh(emb.view(1, -1) @ W1 + b1)
      logits = h @ W2 + b2
      probs = F.softmax(logits, dim=1)
      ix = torch.multinomial(probs, num_samples=1, generator=g).item()
      context = context[1:] + [ix]
      out.append(ix)
      if ix == 0:
        break

    print(''.join(itos[i] for i in out))
```

```
ale.
maraus.
annase.
even.
elyn.
eve.
dite.
dephillanassimaysany.
ryxa.
ia.
lynn.
jayron.
meilyn.
aulaj.
san.
y.
sana.
eveleigany.
zaryas.
saur.
```

Figura 19 - Palavras geradas

B. Conclusão

Em conclusão, a comparação de modelos baseados em bigramas através da realização de contagens no conjunto de treino e da otimização do critério de ML usando o algoritmo de descida de gradiente e o modelo MLP com camada de "Embedding" foi um estudo valioso e esclarecedor. Os resultados mostraram que o modelo MLP com camada de "Embedding" apresentou melhor desempenho na tarefa de predição de palavras em um corpus de texto, indicando a importância de incluir uma camada de representação de palavras em modelos de processamento de linguagem natural. No entanto, é importante ressaltar que a escolha do modelo ideal dependerá do contexto e dos objetivos específicos da análise. Portanto, estudos adicionais são necessários para aprimorar ainda mais os modelos e suas aplicações em diferentes cenários, uma vez que também verificámos que um dataset mais volumoso não indica diretamente uma melhor aprendizagem por parte do modelo.