Attention mechanisms and the Transformer

Back to population - profit

Population	Profit
60000	175000
70000	200000
85000	210000

- What could the profit be for population = 65000?
- A reasonable prediction would be $\frac{175000 + 200000}{2}$.

Attention mechanism

Population	Profit
60000	175000
70000	200000
85000	210000

- Better to use all the information in the dataset!
- 175000*similarity(60000, **65000**) + 200000*similarity(70000, **65000**) + 210000*similarity(85000, **65000**)

Might be 0.4

Might be 0.4

Might be 0.2

What can we use as similarity function?

- Is `(x-x_i)^2` a good similarity function?

The input

Input value for which
we know the output

Known Population	Value of (x-x_i)^2 for x = 65000
60000	5000^2
70000	5000^2
85000	10000^2

Nadaraya-Watson similarity function

$$\text{similarity} = \text{softmax}(-(x - x_i)^2)$$

- Compute squared error.
- Negate squared error to turn it from a **distance** to a **similarity**.
- Use softmax to obtain probabilities.
- for **x=5:**

Known Population	Value of (x-x_i)^2	Value of -(x-x_i)^2	After softmax
10	5^2	-5^2	0.5
20	5^2	-5^2	0.5
30	15^2	-15^2	0

NW in PyTorch

The population we want to make a prediction for

```
def make_prediction(dataset, population):
    population_values = dataset[:, 0]
    profit_values = dataset[:, 1]
    similarities = F.softmax(-(population_values - population) ** 2)
    output = (similarities * profit_values).sum()
    return output
```

All known populations

similarities

All known profit values

Multiply each profit value with the similarity between its population value and the population we are interested in

Learnable embeddings

- 100.000 words in the english language => one hot vectors of 100.000 elements: occupies too much memory
- **Solution**: let the model learn embedding vectors of a size we decide

Learnable embeddings in PyTorch

```
embedding_layer = nn.Embedding(100000, 1024)
```

How many input "words" we have

Size of the output embedding vectors

- `embedding_layer(torch.tensor(0))` => vector of 1024 elements
- `embedding_layer(torch.tensor(200000))` => error, layer only has 100.000 elements

Questions

- What should the second parameter of `nn.Embedding(100000, ?)` be if we want the possibility of using one-hot embeddings?

- How does the model decide on the actual values of the embedding vectors?

Applying attention to the embedding vectors

Embedding vectors	v1, v2, v3
Similarities	sim(v1, v1), sim(v1, v2), sim(v1, v3) sim(v2, v1), sim(v2, v2), sim(v2, v3) sim(v3, v1), sim(v3, v2), sim(v3, v3)
Row-wise softmax	ssim(v1, v1), ssim(v1, v2), ssim(v1, v3) ssim(v2, v1), ssim(v2, v2), ssim(v2, v3) ssim(v3, v1), ssim(v3, v2), ssim(v3, v3)
What we replace v1 with	v1*ssim(v1, v1) + v2*ssim(v1, v2) + v3*ssim(v1, v3)
What we replace v2 with	v1*ssim(v2, v1) + v2*ssim(v2, v2) + v3*ssim(v2, v3)
What we replace v3 with	

Dot-product attention

- Compute similarity between 2 embedding vectors as their **dot product**
- **Formula**: `a*b = |a|*|b|*`**`cos(alpha)`**

alpha	cos(alpha)	Meaning
140 degrees	~-0.7	Large angle between vectors => similarity is negative, very dissimilar
90 degrees	0	Perpendicular vectors => similarity is 0 => really dissimilar
45 degrees	~0.7	Somewhat small angle between vectors => positive similarity, fairly similar
0 degrees	1	Vectors literally on top of each other => maximum similarity

Dot-Product attention in PyTorch

Embedding size

Number of heads, set to 1
for the normal attention

```
attention = nn.MultiheadAttention(256, 1, batch_first = True)

embeddings = torch.rand(2, 20, 256)

output = attention(embeddings, embeddings, embeddings)
```

Question: what is the O complexity of attention in terms of the number of words?

Linear layers in the Transformer

- **Attention:** allows interaction between all words
- **Linear layers:** performs computation on each word!

```
class MLP(nn.Module):
    def __init__(self, embedding_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(embedding_size, hidden_size)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x
```

Simple Transformer architecture

```
class SimpleTransformer(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, 2048)
        attention = nn.MultiheadAttention(2048, 1, batch_first = True)
        self.mlp = MLP(2048, 1024, 2048)
        self.to_output = nn.Linear(2048, vocab_size)

    def forward(self, x):
        x = self.embedding(x)
        x = self.attention(x, x, x)
        x = self.mlp(x)
        x = self.to_output(x)
        return x
```

Questions about the SimpleTransformer

- How long can the input of the Transformer be?
- If the input sequence of the Transformer contains 100 words, how many words can we obtain as output?

Example usage

- Question answering: input to the model is the question, train such that output is the answer
- Machine translation: input is portuguese, output is english
- Text generation: input is some text, output is text that completes the input
- **Problem:** output may be smaller or larger than input!

“Hello, my name is Alex” => “Salut, eu sunt Alex”

Translation with Encoder - Decoder model

Model input in French	Model input in English	Model output in English
[“START”, “Je”, “suis”, “Alex”]	[“START”]	[“I”]
[“START”, “Je”, “suis”, “Alex”]	[“START”, “I”]	[“I”, “AM”]
[“START”, “Je”, “suis”, “Alex”]	[“START”, “I”, “AM”]	[“I”, “AM”, “ALEX”]
[“START”, “Je”, “suis”, “Alex”]	[“START”, “I”, “AM”, “ALEX”]	[“I”, “AM”, “ALEX”, “STOP”]

Translation - size differences

- We need to integrate english and french information!
- We need to support inputs of any sizes: 10 french words and 5 english words, but also 8 french words and 3 english words

Attention to the rescue

- French input: “Salut, je suis Alex”
- Current english input: “Hello, my name”

French words	Salut, je, suis, Alex
English words	Hello, my, name
Similarities	sim(Hello, Salut), sim(Hello, je), sim(Hello, suis), sim(Hello, Alex) sim(my, Salut), sim(my, je), sim(my, suis), sim(my, Alex) sim(name, Salut), sim(name, je), sim(name, suis), sim(name, Alex)
Replace Hello with	sim(Hello, Salut)*Salut + sim(Hello, je)*je + sim(Hello, suis)*suis + sim(Hello, Alex)*Alex
Replace my with	sim(my, Salut)*Salut + sim(my, je)*je + sim(my, suis)*suis + sim(my, Alex)*Alex
Replace name with	*Your turn*

Cheating through *attention*

Model input in French: “Je suis calme”

Second model input in English: “I am”

Expected model output in English: “am calm”

We will try to predict “am” from this model output
Cheating: we want to predict “am” from something that contains “am”

English embeddings	I, am
Similarities	sim(I, I), sim(I, am) sim(am, I), sim(am, am)
“I” gets replaced with	I*sim(I, I)+am*sim(I, am)
“am” gets replaced with	I*sim(am, I)+am*sim(am, am)

Is *cheating* a problem?

- Yes, the model will obtain a low loss, but will be useless
- Model input: “I”
- Expected output: “am”
- Model learned to predict “am” by using “I” and “am”
- Model can’t predict “am” just from “I”

Cheat prevention

- set similarities between a word and any words that follow it to **0**
- "I" now gets replaced by `I*sim(I, I)+am*0` => model no longer sees "am", so is forced to predict "am" from "I" alone

English embeddings	I, am
Similarities	sim(I, I), sim(I, am) sim(am, I), sim(am, am)
"I" gets replaced with	I*sim(I, I)+am*sim(I, am)
"am" gets replaced with	I*sim(am, I)+am*sim(am, am)

Cheat prevention in PyTorch

```
attention = nn.MultiheadAttention(256, 1, batch_first = True)

embeddings = torch.rand(2, 20, 256)

output = attention(

    embeddings,

    embeddings,

    embeddings,

    is_causal = True,

    attn_mask = nn.Transformer.generate_square_subsequent_mask(20)

)
```

```
nn.Transformer.generate_square_subsequent_mask(4)

[[0., -inf, -inf, -inf],
[0.,    0., -inf, -inf]
[0.,    0.,   0., -inf],
[0.,    0.,   0.,   0.]])
```

Question: why *-inf*?

Final model

See notebook :)