C3_W2_lecture_nb_3_perplexity

September 10, 2022

1 Working with JAX numpy and calculating perplexity: Ungraded Lecture Notebook

Normally you would import `numpy` and rename it as `np`.

However in this week's assignment you will notice that this convention has been changed.

Now standard `numpy` is not renamed and `trax.fastmath.numpy` is renamed as `np`.

The rationale behind this change is that you will be using Trax's numpy (which is compatible with JAX) far more often. Trax's numpy supports most of the same functions as the regular numpy so the change won't be noticeable in most cases.

[1]:
```
import numpy
import trax
import trax.fastmath.numpy as np

# Setting random seeds
numpy.random.seed(32)
```

One important change to take into consideration is that the types of the resulting objects will be different depending on the version of numpy. With regular numpy you get `numpy.ndarray` but with Trax's numpy you will get `jax.interpreters.xla.DeviceArray`. These two types map to each other. So if you find some error logs mentioning DeviceArray type, don't worry about it, treat it like you would treat an ndarray and march ahead.

You can get a randomized numpy array by using the `numpy.random.random()` function.

This is one of the functionalities that Trax's numpy does not currently support in the same way as the regular numpy.

[2]:
```
numpy_array = numpy.random.random((5,10))
print(f"The regular numpy array looks like this:\n\n {numpy_array}\n")
print(f"It is of type: {type(numpy_array)}")
```

```
The regular numpy array looks like this:

 [[0.85888927 0.37271115 0.55512878 0.95565655 0.7366696  0.81620514
  0.10108656 0.92848807 0.60910917 0.59655344]
 [0.09178413 0.34518624 0.66275252 0.44171349 0.55148779 0.70371249
```

```
 0.58940123 0.04993276 0.56179184 0.76635847]
[0.91090833 0.09290995 0.90252139 0.46096041 0.45201847 0.99942549
 0.16242374 0.70937058 0.16062408 0.81077677]
[0.03514717 0.53488673 0.16650012 0.30841038 0.04506241 0.23857613
 0.67483453 0.78238275 0.69520163 0.32895445]
[0.49403187 0.52412136 0.29854125 0.46310814 0.98478429 0.50113492
 0.39807245 0.72790532 0.86333097 0.02616954]]

It is of type: <class 'numpy.ndarray'>
```

You can easily cast regular numpy arrays or lists into trax numpy arrays using the `trax.fastmath.numpy.array()` function:

[3]:
```
trax_numpy_array = np.array(numpy_array)
print(f"The trax numpy array looks like this:\n\n {trax_numpy_array}\n")
print(f"It is of type: {type(trax_numpy_array)}")
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)

The trax numpy array looks like this:

 [[0.8588893  0.37271115 0.55512875 0.9556565  0.7366696  0.81620514
  0.10108656 0.9284881  0.60910916 0.59655344]
 [0.09178413 0.34518623 0.6627525  0.44171348 0.5514878  0.70371246
  0.58940125 0.04993276 0.56179184 0.7663585 ]
 [0.91090834 0.09290995 0.9025214  0.46096042 0.45201847 0.9994255
  0.16242374 0.7093706  0.16062407 0.81077677]
 [0.03514718 0.5348867  0.16650012 0.30841038 0.04506241 0.23857613
  0.67483455 0.7823827  0.69520164 0.32895446]
 [0.49403188 0.52412134 0.29854125 0.46310815 0.9847843  0.50113493
  0.39807245 0.72790533 0.86333096 0.02616954]]

It is of type: <class 'jaxlib.xla_extension.DeviceArray'>
```

Hope you now understand the differences (and similarities) between these two versions and numpy. **Great!**

The previous section was a quick look at Trax's numpy. However this notebook also aims to teach you how you can calculate the perplexity of a trained model.

1.1 Calculating Perplexity

The perplexity is a metric that measures how well a probability model predicts a sample and it is commonly used to evaluate language models. It is defined as:

$$P(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1,...,w_{n-1})}}$$

As an implementation hack, you would usually take the log of that formula (so the computation is less prone to underflow problems). You would also need to take care of the padding, since you do not want to include the padding when calculating the perplexity (to avoid an artificially good metric).

After taking the logarithm of $P(W)$ you have:

$$logP(W) = \log\left(\sqrt[N]{\prod_{i=1}^{N}\frac{1}{P(w_i|w_1,...,w_{n-1})}}\right)$$

$$= \log\left(\left(\prod_{i=1}^{N}\frac{1}{P(w_i|w_1,...,w_{n-1})}\right)^{\frac{1}{N}}\right)$$

$$= \log\left(\left(\prod_{i=1}^{N}P(w_i|w_1,...,w_{n-1})\right)^{-\frac{1}{N}}\right)$$

$$= -\frac{1}{N}\log\left(\prod_{i=1}^{N}P(w_i|w_1,...,w_{n-1})\right)$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_1,...,w_{n-1})$$

You will be working with a real example from this week's assignment. The example is made up of: - `predictions` : log probabilities for each element in the vocabulary for 32 sequences with 64 elements (after padding). - `targets` : 32 observed sequences of 64 elements (after padding).

[4]:
```
from trax import layers as tl

# Load from .npy files
predictions = numpy.load('predictions.npy')
targets = numpy.load('targets.npy')

# Cast to jax.interpreters.xla.DeviceArray
predictions = np.array(predictions)
targets = np.array(targets)

# Print shapes
print(f'predictions has shape: {predictions.shape}')
print(f'targets has shape: {targets.shape}')
```

```
predictions has shape: (32, 64, 256)
targets has shape: (32, 64)
```

Notice that the predictions have an extra dimension with the same length as the size of the vocabulary used.

Because of this you will need a way of reshaping `targets` to match this shape. For this you can use `trax.layers.one_hot()`.

Notice that `predictions.shape[-1]` will return the size of the last dimension of `predictions`.

[5]:
```
reshaped_targets = tl.one_hot(targets, predictions.shape[-1]) #trax's one_hot␣
↪function takes the input as one_hot(x, n_categories, dtype=optional)
print(f'reshaped_targets has shape: {reshaped_targets.shape}')
```

```
reshaped_targets has shape: (32, 64, 256)
```

By calculating the product of the predictions and the reshaped targets and summing across the last dimension, the total log propbability of each observed element within the sequences can be computed:

[6]:
```
log_p = np.sum(predictions * reshaped_targets, axis= -1)
```

Now you will need to account for the padding so this metric is not artificially deflated (since a lower perplexity means a better model). For identifying which elements are padding and which are not, you can use `np.equal()` and get a tensor with `1s` in the positions of actual values and `0s` where there are paddings.

[7]:
```
non_pad = 1.0 - np.equal(targets, 0)
print(f'non_pad has shape: {non_pad.shape}\n')
print(f'non_pad looks like this: \n\n {non_pad}')
```

```
non_pad has shape: (32, 64)

non_pad looks like this:

 [[1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]
 …
 [1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]]
```

By computing the product of the log probabilities and the non_pad tensor you remove the effect of padding on the metric:

[8]:
```
real_log_p = log_p * non_pad
print(f'real log probabilities still have shape: {real_log_p.shape}')
```

```
real log probabilities still have shape: (32, 64)
```

You can check the effect of filtering out the padding by looking at the two log probabilities tensors:

[9]:
```
print(f'log probabilities before filtering padding: \n\n {log_p}\n')
print(f'log probabilities after filtering padding: \n\n {real_log_p}')
```

```
log probabilities before filtering padding:

 [[ -5.396545    -1.0311184   -0.66916656 … -22.37673    -23.18771
  -21.843483  ]
 [ -4.5857706   -1.1341286   -8.538033   … -20.15686    -26.837097
  -23.57502   ]
 [ -5.2223887   -1.2824144   -0.17312431 … -21.328228   -19.854412
  -33.88444   ]
 …
 [ -5.396545   -17.291681    -4.360766   … -20.825802   -21.065838
  -22.443115  ]
 [ -5.9313164  -14.247417    -0.2637329  … -26.743248   -18.38433
  -22.355278  ]
 [ -5.670536    -0.10595131   0.         … -23.332523   -28.087376
  -23.878807  ]]

log probabilities after filtering padding:

 [[ -5.396545    -1.0311184   -0.66916656 …  -0.          -0.
    -0.        ]
 [ -4.5857706   -1.1341286   -8.538033   …  -0.          -0.
    -0.        ]
 [ -5.2223887   -1.2824144   -0.17312431 …  -0.          -0.
    -0.        ]
 …
 [ -5.396545   -17.291681    -4.360766   …  -0.          -0.
    -0.        ]
 [ -5.9313164  -14.247417    -0.2637329  …  -0.          -0.
    -0.        ]
 [ -5.670536    -0.10595131   0.         …  -0.          -0.
    -0.        ]]
```

Finally, to get the average log perplexity of the model across all sequences in the batch, you will sum the log probabilities in each sequence and divide by the number of non padding elements (which will give you the negative log perplexity per sequence). After that, you can get the mean of the log perplexity across all sequences in the batch.

[10]:
```
log_ppx = np.sum(real_log_p, axis=1) / np.sum(non_pad, axis=1)
log_ppx = np.mean(-log_ppx)
print(f'The log perplexity and perplexity of the model are respectively:␣
 ↪{log_ppx} and {np.exp(log_ppx)}')
```

```
The log perplexity and perplexity of the model are respectively:
2.621184825897217 and 13.752007484436035
```

Congratulations on finishing this lecture notebook! Now you should have a clear under-

standing of how to work with Trax's numpy and how to compute the perplexity to evaluate your language models. **Keep it up!**