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.

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

# Setting random seeds
trax.supervised.trainer_lib.init_random_number_generators(32)
numpy.random.seed(32)
```

```
INFO:tensorflow:tokens_length=568 inputs_length=512 targets_length
=114 noise_density=0.15 mean_noise_span_length=3.0
```

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.

In [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  0.816205
14
  0.10108656 0.92848807 0.60910917 0.59655344]
 [0.09178413 0.34518624 0.66275252 0.44171349 0.55148779 0.7037124
9
  0.58940123 0.04993276 0.56179184 0.76635847]
 [0.91090833 0.09290995 0.90252139 0.46096041 0.45201847 0.9994254
9
  0.16242374 0.70937058 0.16062408 0.81077677]
 [0.03514717 0.53488673 0.16650012 0.30841038 0.04506241 0.2385761
3
  0.67483453 0.78238275 0.69520163 0.32895445]
 [0.49403187 0.52412136 0.29854125 0.46310814 0.98478429 0.5011349
2
  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:

In [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)}")
```

```
The trax numpy array looks like this:

 [[0.8588893  0.37271115 0.55512875 0.9556565  0.7366696  0.816205
14
  0.10108656 0.9284881  0.60910916 0.59655344]
 [0.09178413 0.34518623 0.6627525  0.44171348 0.5514878  0.7037124
6
  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 0.2385761
3
  0.67483455 0.7823827  0.69520164 0.32895446]
 [0.49403188 0.52412134 0.29854125 0.46310815 0.9847843  0.5011349
3
  0.39807245 0.72790533 0.86333096 0.02616954]]

It is of type: <class 'jax.interpreters.xla.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.

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:

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

As an implementation hack, you would usually take the log of that formula (to enable us to use the log probabilities we get as output of our `RNN` , convert exponents to products, and products into sums which makes computations less complicated and computationally more efficient). You should also take care of the padding, since you do not want to include the padding when calculating the perplexity (because we do not want to have a perplexity measure artificially good). The algebra behind this process is explained next:

$$\begin{aligned} logP(W) &= log\Big(\sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1,\ldots,w_{n-1})}}\Big) \\ &= log\Big(\prod_{i=1}^{N} \frac{1}{P(w_i|w_1,\ldots,w_{n-1})}\Big)^{\frac{1}{N}} \\ &= log\Big(\prod_{i=1}^{N} P(w_i|w_1,\ldots,w_{n-1})\Big)^{-\frac{1}{N}} \\ &= -\frac{1}{N} log\Big(\prod_{i=1}^{N} P(w_i|w_1,\ldots,w_{n-1})\Big) \\ &= -\frac{1}{N}\Big(\sum_{i=1}^{N} logP(w_i|w_1,\ldots,w_{n-1})\Big) \end{aligned}$$

You will be working with a real example from this week's assignment. The example is made up of:

- `predictions` : batch of tensors corresponding to lines of text predicted by the model.
- `targets` : batch of actual tensors corresponding to lines of text.

In [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`.

In [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 perplexity can be computed:

In [6]:
```
total_log_ppx = 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.

In [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 total log perplexity and the non_pad tensor we remove the effect of padding on the metric:

In [8]:
```
real_log_ppx = total_log_ppx * non_pad
print(f'real perplexity still has shape: {real_log_ppx.shape}')
```

```
real perplexity still has shape: (32, 64)
```

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

In [9]:
```
print(f'log perplexity tensor before filtering padding: \n\n {total
_log_ppx}\n')
print(f'log perplexity tensor after filtering padding: \n\n {real_l
og_ppx}')
```

```
log perplexity tensor before filtering padding:

 [[ -5.396545    -1.0311184   -0.66916656 ... -22.37673    -23.187
71
  -21.843483  ]
 [ -4.5857706   -1.1341286   -8.538033   ... -20.15686    -26.8370
97
  -23.57502   ]
 [ -5.2223887   -1.2824144   -0.17312431 ... -21.328228   -19.8544
12
  -33.88444   ]
 ...
 [ -5.396545   -17.291681    -4.360766   ... -20.825802   -21.0658
38
  -22.443115  ]
 [ -5.9313164  -14.247417    -0.2637329  ... -26.743248   -18.3843
3
  -22.355278  ]
 [ -5.670536    -0.10595131   0.         ... -23.332523   -28.0873
76
  -23.878807  ]]

log perplexity tensor 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.        ]]
```

To get a single average log perplexity across all the elements in the batch you can sum across both dimensions and divide by the number of elements. Notice that the result will be the negative of the real log perplexity of the model:

In [10]:
```
log_ppx = np.sum(real_log_ppx) / np.sum(non_pad)
log_ppx = -log_ppx
print(f'The log perplexity and perplexity of the model are respecti
vely: {log_ppx} and {np.exp(log_ppx)}')
```

```
The log perplexity and perplexity of the model are respectively: 2
.3281209468841553 and 10.258646965026855
```

Congratulations on finishing this lecture notebook! Now you should have a clear understanding of how to work with Trax's numpy and how to compute the perplexity to evaluate your language models. **Keep it up!**