Custom Embedding Nets

<u>Introduction to Embedding Nets</u>

- The data that we get from the simulator may contain a lot of irrelevant or unstructured data which makes analysis and training of the NNs much more inefficient
- We use embedding nets to extract the most relevant data from the simulator in a structured, often lower dimensional form
- Especially for time series data, or images, compression to a lower-dimensional latent space is often very important
- Embedding networks can also learn problem-specific representations of the data, emphasizing features most relevant to the posterior

<u>Integration into SBI – where do Embedding Nets sit</u>

- The overall process in SBI works as follows:
  - Simulator produces some data *x* from a prior sample
  - The data is put through an embedding network which produces a representation of the data, *f(x)*
  - The density estimator models the posterior, $p(\Theta|x)$ – we can approximate this as $p(\Theta|f(x))$
  - The density estimator is trained optimised to learn the posterior conditioned on these embeddings – the embedding network is often trained jointly with the density estimator
  - In the code – order is define embedding net class, instantiate it, define the density estimator using the neural network version defined (e.g. MAF) and the embedding net, and then initialise the SNPE with the defined density estimator
- The posterior_nn method takes in the model and the embedding net as inputs; this is then the defined density estimator to put into the SNPE method

<u>Things to include within the Embedding Net class:</u>

- Linear and ReLu layers – most basic version would be Linear to a hidden dimension size, ReLu, Linear again, ReLu, and then Linear back down to the output dimension size
- For grid like data can also include convolutional layers, such as nn.Conv2d or nn.Conv3D
- Can use residual layers as well, such as ResNet blocks, to help learn representations without vanishing gradients
- For sequential data, can use recurrent layers such as nn.LSTM or nn.GRU for temporal dependencies
- Attention mechanisms: helps the embedding net focus on the most relevant parts of the input – self-attention mechanism (transformers): nn.MultiheadAttention, or cross-attention for mapping two different types of data
- In addition to all of these, can introduce batch normalisation, dropout of neurons, weight regularisation and dynamic weight generation (using hypernetworks)

<u>Understanding the shapes within the Convolutional Embedding net:</u>

- For the linear layers and similar- they take in input and output dimension values, and the way they work is if we take the nn.Linear(a,b) module – the input data should be of the shape m x a tensor, and the output will be a m x b tensor
- The 1D convolutional embedding net will take in:
  - an in_channels input (the second dimension of the input data)
  - an out_channels input (the second dimension of the output data)
  - kernel_size input (size of the filter)
  - stride input (step size of the by which the filter moves along the sequence)
  - padding input (controls how the boundaries of sequences are handled, adding 0s)
- The ReLu activation function takes no inputs; same with other versions of the Lu function
- There is a MaxPool function, which has a window size over which the maximum value is computed

Custom Embedding Nets

- There is also a flatten, which reduces the multi-dimensional feature maps which are the output of the convolutional and pooling layers into a 1D output
- This 1D output is then taken into a FC layer – built up in the same way as above
- All of this can be extended into convolutional embedding nets of 2 or 3D, by just adjusting the dimensions to be tensors of the appropriate size

<u>Recurrent Neural Networks in the Embedding Net</u>

- RNNs are particularly useful for sequential input data and can be combined with convolutional embedding nets for spatiotemporal data – one example of built in RNNs is nn.LSTM (Long Short-Term Memory – designed to avoid vanishing gradient problem)
- Simple version – use LTSM followed by a sequential FC layer
- LSTM input – input_features (feature size of the input) and hidden_size (number of hidden units in the LSTM)
- If using convolutional networks as well - apply convolutional layers first, then flatten spatial dimensions to create a sequence, then pass through the LSTM and then finally through a fully connected layer