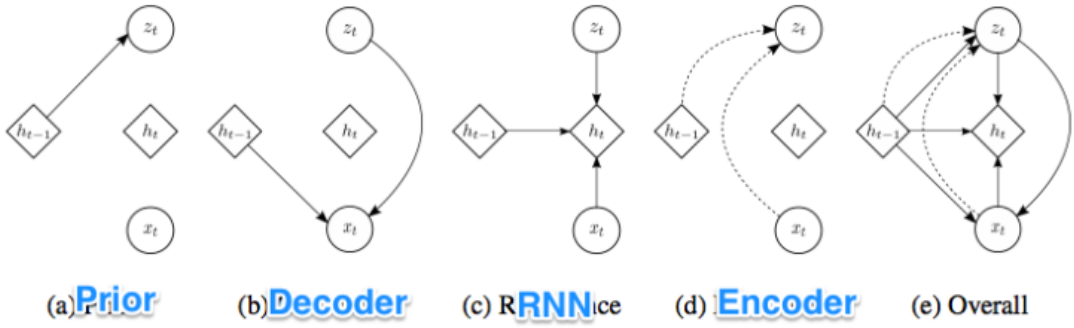
Variational Recurrent Neural Network (VRNN) with Pytorch

For an introduction on Variational Autoencoder (VAE) check this post. VAE contains two types of layers: deterministic layers, and stochastic latent layers. Stochastic nature is mimic by the reparameterization trick, plus a random number generator.
VRNN, as suggested by the name, introduces a third type of layer: hidden layers (or recurrent layers). Hidden layers has sequential dependence on its previous timestep, thus we could model time series data.

Network structure

In the discuss following, x is training data (randomly chosen from the training set), h is hidden state, and z is latent state (randomly sampled from its prior distribution). Subscripts indicates the time sequence.

The figure below is from Chung's paper. I found names a bit confusing, so I renamed them.



The notations mentioned so far are quite abstract, but it is enough to understand what is going on. There are four core sub components in the network.
– Encoder net: (used only in training)
– RNN net: $x_t + z_t + h_{t-1} -> h_t$ (could use any kind of RNN)
– Decoder net: $z_t + h_{t-1} -> x_t$ (reconstruct x_t, not x_{t+1}!)
– Prior net: $h_{t-1} -> z_t$

1. Generation phase (after training)

Only the last three components are used after training. To generate new sequences, we repeat the follow cycle, starting with an initial h_0.
– Sample z_1 using hyper-parameters from Prior net
– Get/sample x_1 from decoder net
– Get h_1 from RNN net, for use in the next cycle

In the second step, whether we get a deterministic output, or sample a stochastic one depends on autoencoder-decoder net design. In Chung's paper, he used an Univariate Gaussian Model autoencoder-decoder, which is irrelevant to the variational design.

2. Training phase

During training we have only sequential data x at hand. And the goal is to reconstruct x at the output. To do this, the encoder net is introduced. Here, we assume sampling z from `Prior net` is equivalent to sampling x and then encoding it. As both Prior net and encoder net output hyper parameters, this assumption is equivalent to say they should output the identical hyperparameters. So in the training phase z is sampled using hyperparameters from the encoder net instead. The validity of the assumption is expressed in a KL divergence between the encoder distribution and the prior distribution.
Now we can put pieces together for the training phase. First, we forward data through the network each cycle. Starting with training data x_1 and hidden state h_0
– Sample z_1 from hyper-parameters *from Encoder*
– Get/sample $x_{1,reconstruct}$ from decoder net
– Get h_1 from RNN net, for use in the next cycle
What about loss function?
– Loss 1: Difference between x_1 and $x_{1,reconstruct}$. MaxEnt, MSE, Likehoods, or anything.
– Loss 2: Difference between `Prior net` and `Encoder net`. KL divergence, always positive.
To calculate KL divergence we need hyper-parameters from `Prior net` as well, so
– Keep hyper-parameters from `Encoder net`
– Get hyper-parameters from `Prior net`
VRNNCell is defined in a GRUCell style, and every configuration is hard coded.
– To handle the ASCII encoding, an embedding layer is added.
– A nn.GRUCell( ) with 64 neurons. Could use nn.GRU( ) for multi layer RNN.
– Output are logits, training with cross-entropy loss

Parameters used for training the network.
– Adam, learning rate = 0.001
– mini batch size = 64, sequence size = 300

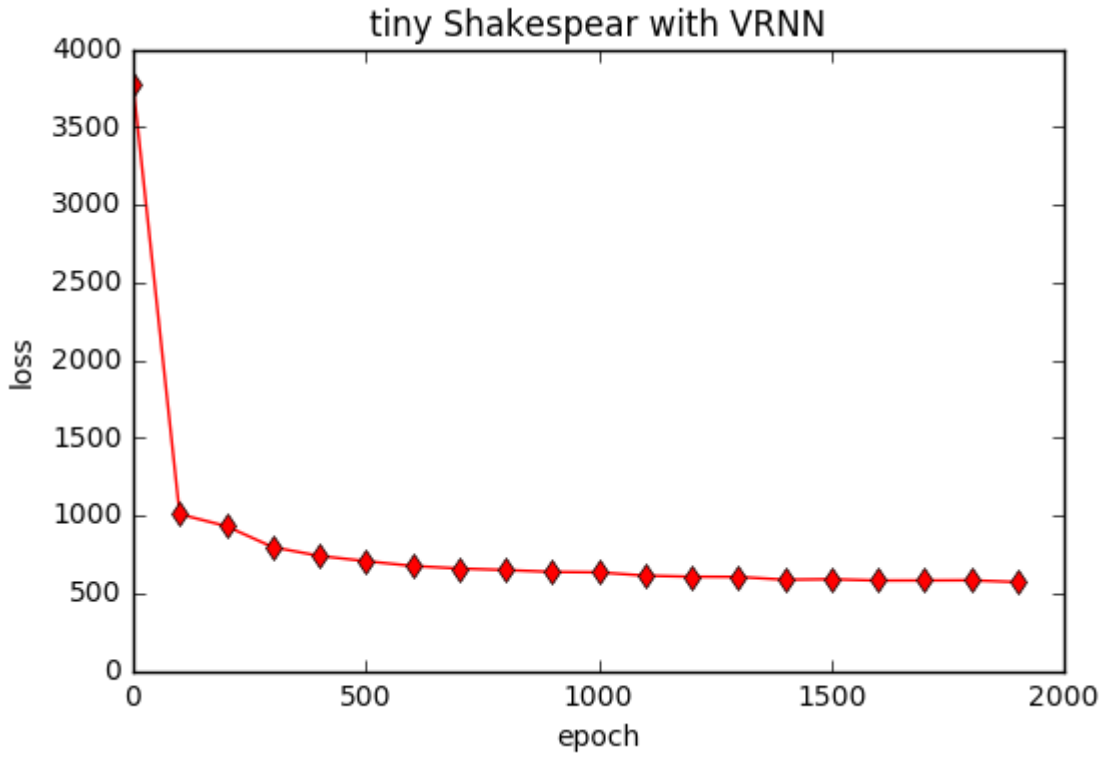


Figure. Training loss over 2000 epochs

I am being lazy here, because the embedding layer maps all 128 ASCII, which might be system control symbols. But such symbols only show up in early generation results. Here is a continuously generated piece after 2000 epochs. One comment about `temperature`. In my previous RNN example, it seems using 0.8 is appropriate. But for VRNN I feel a higher temperature is allowed. Like char-rnn demo, the overall dialogue format is well reserved. Otherwise, no good. For better results, train longer time and use multi-layer RNN modules.