# Final Project: Character-Level Language Modeling for Text Generation via Deep Markov Models

Armaan Kohli - ECE467 Natural Language Processing
Spring 2020

## *Remarks*

We attempted to use a deep markov model (DMM) to make a character-level language model. This work is based on recent developments in the understanding of discrete time series, such as MIDI, as well as natural language processing. Using a DMM, we were able to generate text that yielded quantitative performance approaching state of the art for character-based language models. However, training remains unstable and more research into DMMs is likely required for their performance for language models to improve.

## *Deep Markov Models*

Traditional markov models are a method representing complex temporal dependencies in observed data. A markov model has a chain of latent variables, with each latent (or hidden) variable in the chain is conditioned on the previous latent variable. This is a useful approach, but if we want to represent complex data with complex dynamics, such as text, we would like to be able to model dynamics that are potentially highly non-linear.

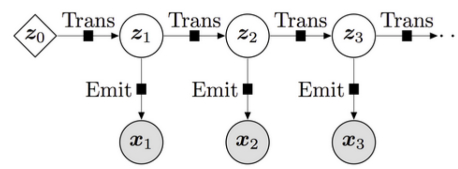

Figure 1: An illustration of a DMM. Each of the black squares represent an RNNs that determine the probability of emission or transmission. Image replicated from Pyro documentation. [ADD CITATION]

This brings forth the idea of a deep markov model, wherein we allow the transition probabilities governing the dynamics of the latent variables as well as the the emission probabilities that govern how the observations are generated by the latent dynamics to be parametrized by (non-linear) neural networks. DMMs were first used in the setting of polyphonic music generation. Using a MIDI representation of musical notes, Krishnan et. al were able to generate high-quality songs and learn a representaton of electronic health record data [1].

Even though this method was originally designed for music generation, character-level language models can be thought of in a similar way. At each time step, music can be represented by an 88-dimensional binary vector. Similarly, characters in a phrase can be represented by a one-hot vector with a dimension given by the size of the learned dictionary. Research by the Harvard Intellegnt Probabalistic Systems (HIPS) group takes a similar approach, using the a neural network for both polyphonic music generation and character-level language modeling, the only change being the distribution from which the data is drawn from, the obervation liklihod (Bernoulli vs categorical) [2]. HIPS uses a generative flow model for character-level language modelling as opposed to a DMM, however. The inference strategy we're going to use called variational inference (VI), which requires specifying a parametrized family of distributions that can be used to approximate the posterior distribution over the latent random variables. Due to the complex temporal relations we seek to model, we can expect the posterior distribution to be highly non-trivial, necessitating a probabilistic approach. Thus, we use `PyTorch` as our choice of deep learning framework, as well as `Pyro`, a probabilistic programming language integrated into `PyTorch` to effectively sample and perform VI on our model.

## *Implementation Details*

We use a single-layer RNN for our emission and transmission probabilities. Our objective function is the ELBO (evidence-based lower bound) with a KL-annealing term $\beta$, inspired by [3].

$$\mathcal{F}(\theta, \phi, \beta; \boldsymbol{x}, \boldsymbol{z}) \geq \mathcal{L}(\theta, \phi; \boldsymbol{x}, \boldsymbol{z}) = \mathbb{E}_{q_\phi(z|x)}[log_{p_\theta}(\boldsymbol{x}|\boldsymbol{z})] - \beta D_{KL}(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})) \tag{1}$$

We use Monte Carlo estimates of the KL divergence term.

We train the language model using the Penn Treebank (PTB) corpus. We perform treat every line in the corpus as a distinct sequence, or sentence, and tokenize each character in each sentence, adding the `<unk>` token for low-frequency or unknown words, and `<eos>` to demarcate the end of a sentence. The size of the dictionary was 52. In order to generate a character embedding, we simply encoded our character dictionary as a one-hot 52-dimensional vector. This was an appropriate choice due to the small dictionary size. We opt for a batch size of 16. For full details see github.com/armaank/textDMM for the full codebase and the parameters used to train the network.

## *Results & Discussion*

We were able to ....

These aforementioned issues might be resolved by using a different method for KL annealing, which can improve stability during training. Furthermore, we use a Monte Carlo estimates of the KL divergence, leading to higher variance gradient estimates of the ELBO loss, which can also destabilize performance during early training periods. We might also trying using an LSTM architecture to parametrize our transmission and emission probabilities over the so-called 'vanilla' RNN. On a related note, one possibility is that exploding gradients are caused by lengthy input sequences, so one way to resolve this issue would be to only train on shorter sequences of characters.

## *Conclusion*

In conclusion, we were able to successfully train a DMM as a character-level language model and achieve performance close to that of traditional character-level language models using purely RNNs/LSTMs. However, though more research is needed to improve DMMs for NLP tasks. The power of DMMs and other probabilistic models is their flexibility, in that the same model can generate MIDI music, missing EHR data and text with only the changing distribution governing the observation likelihood.

## *References*

[1] R. G. Krishnan, U. Shalit, and D. Sontag, "Structured inference networks for nonlinear state space models," 2016.

[2] Z. M. Ziegler and A. M. Rush, "Latent normalizing flows for discrete sequences," 2019.

[3] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework," in *ICLR*, 2017.

## *Appendix A: Code*

The code below is `dmm.py`, the main model code.

```
"""dmm
"""
import argparse
import os

import numpy as np
import torch
import torchtext
import pyro

import torch.nn as nn
import pyro.distributions as dist
import pyro.poutine as poutine

from torch.autograd import Variable
from pyro.distributions import TransformedDistribution

import utils


class Emitter(nn.Module):
    """
    parameterizes the categorical observation likelihood p(x_t|z_t)
    """

    def __init__(self, input_dim, z_dim, emission_dim):
        super().__init__()
        """
        initilize the fcns used in the network
        """
        self.lin_z_to_hidden = nn.Linear(z_dim, emission_dim)
        self.lin_hidden_to_hidden = nn.Linear(emission_dim, emission_dim)
        self.lin_hidden_to_input = nn.Linear(emission_dim, input_dim)
        self.relu = nn.ReLU()

        pass

    def forward(self, z_t):
        """
        given z_t, compute the probabilities that parameterizes the categorical distribution p(x_t|z_t)
        """
        h1 = self.relu(self.lin_z_to_hidden(z_t))
        h2 = self.relu(self.lin_hidden_to_hidden(h1))
        probs = torch.sigmoid(
            self.lin_hidden_to_input(h2)
        )  # might need to change to argmax, max?, softmax?

```

```
        return probs


class GatedTransition(nn.Module):
    """
    parameterizes the gaussian latent transition probability p(z_t | z_{t-1})
    """

    def __init__(self, z_dim, transition_dim):
        super().__init__()
        """
        initilize the fcns used in the network
        """
        self.lin_gate_z_to_hidden = nn.Linear(z_dim, transition_dim)
        self.lin_gate_hidden_to_z = nn.Linear(transition_dim, z_dim)
        self.lin_proposed_mean_z_to_hidden = nn.Linear(z_dim, transition_dim)
        self.lin_proposed_mean_hidden_to_z = nn.Linear(transition_dim, z_dim)
        self.lin_sig = nn.Linear(z_dim, z_dim)
        self.lin_z_to_loc = nn.Linear(z_dim, z_dim)

        self.lin_z_to_loc.weight.data = torch.eye(z_dim)
        self.lin_z_to_loc.bias.data = torch.zeros(z_dim)

        self.relu = nn.ReLU()
        self.softplus = nn.Softplus()

        pass

    def forward(self, z_t_1):
        """
        Given the latent z_{t-1} we return the mean and scale vectors that parameterize the
        (diagonal) gaussian distribution p(z_t | z_{t-1})`
        """
        # compute the gating function
        _gate = self.relu(self.lin_gate_z_to_hidden(z_t_1))
        gate = torch.sigmoid(self.lin_gate_hidden_to_z(_gate))
        # compute the 'proposed mean'
        _proposed_mean = self.relu(self.lin_proposed_mean_z_to_hidden(z_t_1))
        proposed_mean = self.lin_proposed_mean_hidden_to_z(_proposed_mean)
        # assemble the actual mean used to sample z_t, which mixes a linear transformation
        # of z_{t-1} with the proposed mean modulated by the gating function
        loc = (1 - gate) * self.lin_z_to_loc(z_t_1) + gate * proposed_mean
        # compute the scale used to sample z_t, using the proposed mean from
        # above as input the softplus ensures that scale is positive
        scale = self.softplus(self.lin_sig(self.relu(proposed_mean)))
        # return loc, scale which can be fed into Normal
        return loc, scale


class Combiner(nn.Module):
    """
```

```
    parameterizes q(z_t | z_{t-1}, x_{t:T}), which is the basic building block
    of the guide (i.e. the variational distribution). The dependence on x_{t:T} is
    through the hidden state of the RNN
    """

    def __init__(self, z_dim, rnn_dim):
        super().__init__()
        """
        initilize the fcns used in the network
        """
        self.lin_z_to_hidden = nn.Linear(z_dim, rnn_dim)
        self.lin_hidden_to_loc = nn.Linear(rnn_dim, z_dim)
        self.lin_hidden_to_scale = nn.Linear(rnn_dim, z_dim)
        self.tanh = nn.Tanh()
        self.softplus = nn.Softplus()

        pass

    def forward(self, z_t_1, h_rnn):
        """
        Given the latent z_{t-1} at at a particular time as well as the hidden
        state of the RNN h(x_{t:T}) we return the mean and scale vectors that
        parameterize the (diagonal) gaussian distribution q(z_t | z_{t-1}, x_{t:T})
        """
        # combine the rnn hidden state with a transformed version of z_t_1
        h_combined = 0.5 * (self.tanh(self.lin_z_to_hidden(z_t_1)) + h_rnn)
        # use the combined hidden state to compute the mean used to sample z_t
        loc = self.lin_hidden_to_loc(h_combined)
        # use the combined hidden state to compute the scale used to sample z_t
        scale = self.softplus(self.lin_hidden_to_scale(h_combined))
        # return loc, scale which can be fed into Normal
        return loc, scale


class DMM(nn.Module):
    """
    module for the model and the guide (variational distribution) for the DMM
    """

    def __init__(
        self,
        input_dim=52,
        z_dim=100,
        emissions_dim=100,
        transition_dim=200,
        rnn_dim=600,
        num_layers=1,
        dropout=0.0,
    ):
        super().__init__()
        """
```

```
        instantiate modules used in the model and guide
        """
        self.emitter = Emitter(intput_dim, z_dim, emission_dim)
        self.transition = GatedTransition(z_dim, transition_dim)
        self.combiner = Combiner(z_dim, rnn_dim)

        # TODO: alter dropout scheme
        if num_layers == 1:
            rnn_dropout = 0.0
        else:
            rnn_dropout = dropout

        # TODO: add option for bidirectional rnn?
        self.rnn = nn.RNN(
            input_size=input_dim,
            hidden_size=rnn_dim,
            nonlinearity="relu",
            batch_first=True,
            bidirectional=False,
            num_layers=num_layers,
            dropout=rnn_drouput,
        )
        """
        define learned parameters that define the probability distributions P(z_1) and q(z_1) and hidden state of rnn
        """
        self.z_0 = nn.Parameter(torch.zeros(z_dim))
        self.z_q_0 = nn.Parameter(torch.zeros(z_dim))
        self.h_0 = nn.Parameter(torch.zeros(1, 1, rnn_dim))

        pass

    def model(self, batch, reversed_batch, batch_mask, batch_seqlens, kl_anneal=1.0):
        """
        the model defines p(x_{1:T}|z_{1:T}) and p(z_{1:T})
        """
        # maximum duration of batch
        Tmax = batch.size(1)

        # register torch submodules w/ pyro
        pyro.module("dmm", self)

        # setup recursive conditioning for p(z_t|z_{t-1})
        z_prev = self.z_0.expand(batch.size(0), self.z_0.size(0))

        # sample conditionally indepdent text across the batch
        with pyro.plate("z_batch", len(batch)):
            # sample latent vars z and observed x w/ multiple samples from the guide for each z
            for t in pyro.markov(range(1, Tmax + 1)):

                # compute params of diagonal gaussian p(z_t|z_{t-1})
```

```
                z_loc, z_scale = self.trans(z_prev)

                # sample latent variable
                with poutine.scale(scale=kl_anneal):
                    z_t = pyro.sample(
                        "z_%d" % t,
                        dist.Normal(z_loc, z_scale)
                        .mask(batch_mask[:, t - 1 : t])
                        .to_event(1),
                    )

                # compute emission probability from latent variable
                emission_prob = self.emitter(z_t)

                # observe x_t according to the Categorical distribution defined by the emitter
    probability
                pyro.sample(
                    "obs_x_%d" % t,
                    dist.OneHotCategorical(emission_prob)
                    .mask(batch_mask[:, t - 1 : t])
                    .to_event(1),
                    obs=batch[:, t - 1, :],
                )

                # set conditional var for next time step
                z_prev = z_t
        pass

    def guide(self, batch, reversed_batch, batch_mask, batch_seqlens, kl_anneal=1.0):
        """
        the guide defines the variational distribution q(z_{1:T}|x_x{1:T})
        """
        # maximum duration of batch
        Tmax = batch.size(1)

        # register torch submodules w/ pyro
        pyro.module("dmm", self)

        # to parallelize, we broadcast rnn into continguous gpu memory
        h_0_contig = self.h_0.expand(
            1, batch_size(0), self.rnn.hidden_size
        ).contiguous()

        # push observed sequence through rnn
        rnn_output, _ = self.rnn(batch_reversed, h_0_contig)

        # reverse and unpack rnn output
        rnn_output = utils.pad_reverse(rnn_output, batch_seqlens)

        # setup recursive conditioning
        z_prev = self.z_q_0.expand(batch.size(0), self.z_q_0.size(0))
```

```

        with pyro.plate("z_batch", len(mini_batch)):

            for t in pyro.markov(range(1, Tmax + 1)):

                z_loc, z_scale = self.combiner(z_prev, rnn_output[:, t - 1, :])

                z_dist = dist.Normal(z_loc, z_scale)

                assert z_dist.event_shape == ()
                assert z_dist.batch_shape[-2:] == len(batch) == self.z_q_0.size(0)

                # sample z_t from distribution z_dist
                with pyro.poutine.scale(scale=kl_anneal):
                    z_t = pyro.samle(
                        "z_%d" % t, z_dist.mask(batch[:, t - 1 : t]).to_event(1)
                    )

                # set conditional var for next time step
                z_prev = z_t

        pass
```