# 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.

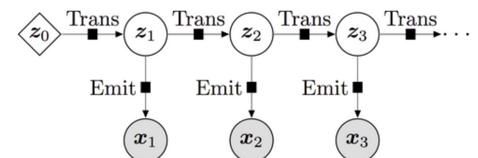


Figure 1: An illustration of a DMM. Each of the black squares represent an RNNs that determine the probability of emission or transmission. Image replicated from Pyro documentation [? ]

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 [? ].

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) [? ]. 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 [? ].

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

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.

We found that using the entirety of the PTB dataset to be challenging for our language model. The longer the sentence used in the character level language model, the more difficult the model was to optimize. We found that limiting ourselves to sentences shorter than 50 characters significantly improved results. However, this would mean that the semantic quality of generated sentences would be reduced as the length of the sentence increases.

## *Results & Discussion*

Figure  illustrates the negative log likelihood learning curve, showing that the model converges.

In order to see if our implementation is reasonable we compared our results to the numbers reported in [? ] in Table 1

| NLL Validation/Test Loss | |
|---|---|
| LSTM | 1.38 |
| AWD-LSTM | 1.18 |
| IAF | 1.42 |
| DMM | 6.82 |

Table 1: This table compares the various results of state of the art character level language models. The results for the LSTM, AWD-LSTM and IAF come from [? ]

The LSTM, AWD-LSTM and IAF are state of the art language models. These langauge models were able to train for much longer durations and were able to use a larger portion of the PTB dataset because they could use long sentences. Our results are fairly close to the others in terms of NLL loss, which appears to be the standard performance metric in character-level language modelling tasks.

Performance of the DMM might be improve by using a different method for KL annealing, which can improve stability during training. Furthermore, we use a Monte Carlo estimate 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 the way we resolved this was to train on shorter text sequences, which it a bit of a hacky solution.

## *Conclusion*

In conclusion, we were able to successfully train a DMM as a character-level language model and achieve strong performance. However, though more research is needed to improve DMMs for NLP tasks.

## *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?

```

```
48         return probs
49
50
51 class GatedTransition(nn.Module):
52     """
53     parameterizes the gaussian latent transition probability p(z_t | z_{t-1})
54     """
55
56     def __init__(self, z_dim, transition_dim):
57         super().__init__()
58         """
59         initilize the fcns used in the network
60         """
61         self.lin_gate_z_to_hidden = nn.Linear(z_dim, transition_dim)
62         self.lin_gate_hidden_to_z = nn.Linear(transition_dim, z_dim)
63         self.lin_proposed_mean_z_to_hidden = nn.Linear(z_dim, transition_dim)
64         self.lin_proposed_mean_hidden_to_z = nn.Linear(transition_dim, z_dim)
65         self.lin_sig = nn.Linear(z_dim, z_dim)
66         self.lin_z_to_loc = nn.Linear(z_dim, z_dim)
67
68         self.lin_z_to_loc.weight.data = torch.eye(z_dim)
69         self.lin_z_to_loc.bias.data = torch.zeros(z_dim)
70
71         self.relu = nn.ReLU()
72         self.softplus = nn.Softplus()
73
74         pass
75
76     def forward(self, z_t_1):
77         """
78         Given the latent z_{t-1} we return the mean and scale vectors that parameterize the
79         (diagonal) gaussian distribution p(z_t | z_{t-1})‘
80         """
81         # compute the gating function
82         _gate = self.relu(self.lin_gate_z_to_hidden(z_t_1))
83         gate = torch.sigmoid(self.lin_gate_hidden_to_z(_gate))
84         # compute the ’proposed mean’
85         _proposed_mean = self.relu(self.lin_proposed_mean_z_to_hidden(z_t_1))
86         proposed_mean = self.lin_proposed_mean_hidden_to_z(_proposed_mean)
87         # assemble the actual mean used to sample z_t, which mixes a linear transformation
88         # of z_{t-1} with the proposed mean modulated by the gating function
89         loc = (1 - gate) * self.lin_z_to_loc(z_t_1) + gate * proposed_mean
90         # compute the scale used to sample z_t, using the proposed mean from
91         # above as input the softplus ensures that scale is positive
92         scale = self.softplus(self.lin_sig(self.relu(proposed_mean)))
93         # return loc, scale which can be fed into Normal
94         return loc, scale
95
96
97 class Combiner(nn.Module):
98     """
```

```
    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,
        emission_dim=100,
        transition_dim=200,
        rnn_dim=600,
        num_layers=1,
        dropout=0.0,
    ):
        super().__init__()
        """
```

```
150         instantiate modules used in the model and guide
151         """
152         self.emitter = Emitter(input_dim, z_dim, emission_dim)
153         self.transition = GatedTransition(z_dim, transition_dim)
154         self.combiner = Combiner(z_dim, rnn_dim)
155 
156         if num_layers == 1:
157             rnn_dropout = 0.0
158         else:
159             rnn_dropout = dropout
160 
161         self.rnn = nn.RNN(
162             input_size=input_dim,
163             hidden_size=rnn_dim,
164             nonlinearity="relu",
165             batch_first=True,
166             bidirectional=False,
167             num_layers=num_layers,
168             dropout=rnn_dropout,
169         )
170         """
171         define learned parameters that define the probability distributions P(z_1) and q(z_1) and hidden
      state of rnn
172         """
173         self.z_0 = nn.Parameter(torch.zeros(z_dim))
174         self.z_q_0 = nn.Parameter(torch.zeros(z_dim))
175         self.h_0 = nn.Parameter(torch.zeros(1, 1, rnn_dim))
176 
177         self.cuda()
178 
179         pass
180 
181     def model(self, batch, reversed_batch, batch_mask, batch_seqlens, kl_anneal=1.0):
182         """
183         the model defines p(x_{1:T}|z_{1:T}) and p(z_{1:T})
184         """
185         # maximum duration of batch
186         Tmax = batch.size(1)
187 
188         # register torch submodules w/ pyro
189         pyro.module("dmm", self)
190 
191         # setup recursive conditioning for p(z_t|z_{t-1})
192         z_prev = self.z_0.expand(batch.size(0), self.z_0.size(0))
193 
194         # sample conditionally indepdent text across the batch
195         with pyro.plate("z_batch", len(batch)):
196             # sample latent vars z and observed x w/ multiple samples from the guide for each z
197             for t in pyro.markov(range(1, Tmax + 1)):
198 
199                 # compute params of diagonal gaussian p(z_t|z_{t-1})
```

```
                z_loc, z_scale = self.transition(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(reversed_batch, h_0_contig)

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

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

```
250
251         with pyro.plate("z_batch", len(batch)):
252
253             for t in pyro.markov(range(1, Tmax + 1)):
254
255                 z_loc, z_scale = self.combiner(z_prev, rnn_output[:, t - 1, :])
256
257                 z_dist = dist.Normal(z_loc, z_scale)
258                 assert z_dist.event_shape == ()
259                 assert z_dist.batch_shape[-2:] == (len(batch), self.z_q_0.size(0))
260
261                 # sample z_t from distribution z_dist
262                 with pyro.poutine.scale(scale=kl_anneal):
263                     z_t = pyro.sample(
264                         "z_%d" % t, z_dist.mask(batch_mask[:, t - 1 : t]).to_event(1)
265                     )
266
267                 # set conditional var for next time step
268                 z_prev = z_t
269
270         pass
```

The code below is `train.py`, the main code used to perform training and evaluation.

```
"""train
"""
import os
import random
import time

import numpy as np
import pyro
import torch

import torchtext

from torch import nn

from pyro.infer import SVI, Trace_ELBO
from pyro.optim import ClippedAdam
from torch.autograd import Variable

import utils
import datahandler
from dmm import DMM


class Trainer(object):
    """
    trainer class used to instantiate, train and validate a network
    """

    def __init__(self, args):

        # argument gathering
        self.rand_seed = args.rand_seed
        self.dev_num = args.dev_num
        self.cuda = args.cuda
        self.n_epoch = args.n_epoch
        self.batch_size = args.batch_size
        self.lr = args.lr
        self.beta1 = args.beta1
        self.beta2 = args.beta2
        self.wd = args.wd
        self.cn = args.cn
        self.lr_decay = args.lr_decay
        self.kl_ae = args.kl_ae
        self.maf = args.maf
        self.dropout = args.dropout
        self.ckpt_f = args.ckpt_f
        self.load_opt = args.load_opt
        self.load_model = args.load_model
        self.save_opt = args.save_opt
        self.save_model = args.save_model
```

```
        self.maxlen = args.maxlen
        # setup logging
        self.log = utils.get_logger(args.log)
        self.log(args)

    def _validate(self, val_iter):
        """
        freezes training and validates on the network with a validation set
        """
        # freeze training
        self.dmm.rnn.eval()
        val_nll = 0
        for ii, batch in enumerate(iter(val_iter)):

            batch_data = Variable(batch.text[0].to(self.device))
            seqlens = Variable(batch.text[1].to(self.device))

            # transpose to [B, seqlen, vocab_size] shape
            batch_data = torch.t(batch_data)
            # compute one hot character embedding
            batch_onehot = nn.functional.one_hot(batch_data, self.vocab_size).float()
            # flip sequence for rnn
            batch_reversed = utils.reverse_seq(batch_onehot, seqlens)
            batch_reversed = nn.utils.rnn.pack_padded_sequence(
                batch_reversed, seqlens, batch_first=True
            )
            # compute temporal mask
            batch_mask = utils.generate_batch_mask(batch_onehot, seqlens).cuda()
            # perform evaluation
            val_nll += self.svi.evaluate_loss(
                batch_onehot, batch_reversed, batch_mask, seqlens
            )

        # resume training
        self.dmm.rnn.train()

        loss = val_nll / self.N_val_data

        return loss

    def _train_batch(self, train_iter, epoch):
        """
        process a batch (single epoch)
        """
        batch_loss = 0
        epoch_loss = 0
        for ii, batch in enumerate(iter(train_iter)):

            batch_data = Variable(batch.text[0].to(self.device))
            seqlens = Variable(batch.text[1].to(self.device))

```

```
102             # transpose to [B, seqlen, vocab_size] shape
103             batch_data = torch.t(batch_data)
104             # compute one hot character embedding
105             batch_onehot = nn.functional.one_hot(batch_data, self.vocab_size).float()
106             # flip sequence for rnn
107             batch_reversed = utils.reverse_seq(batch_onehot, seqlens)
108             batch_reversed = nn.utils.rnn.pack_padded_sequence(
109                 batch_reversed, seqlens, batch_first=True
110             )
111             # compute temporal mask
112             batch_mask = utils.generate_batch_mask(batch_onehot, seqlens).cuda()
113 
114             # compute kl-div annealing factor
115             if self.kl_ae > 0 and epoch < self.kl_ae:
116                 min_af = self.maf
117                 kl_anneal = min_af + (1 - min_af) * (
118                     float(ii + epoch * self.N_batches + 1)
119                     / float(self.kl_ae * self.N_batches)
120                 )
121             else:
122                 # default kl-div annealing factor is unity
123                 kl_anneal = 1.0
124 
125             # take gradient step
126             batch_loss = self.svi.step(
127                 batch_onehot, batch_reversed, batch_mask, seqlens, kl_anneal
128             )
129             batch_loss = batch_loss / (torch.sum(seqlens).float())
130             print("loss at iteration {0} is {1}".format(ii, batch_loss))
131         epoch_loss = epoch_loss+batch_loss
132 
133         return epoch_loss
134 
135     def train(self):
136         """
137         trains a network with a given training set
138         """
139         self.device = torch.device("cuda")
140         np.random.seed(self.rand_seed)
141         torch.manual_seed(self.rand_seed)
142 
143         train, val, test, vocab = datahandler.load_data("./data/ptb", self.maxlen)
144 
145         self.vocab_size = len(vocab)
146 
147         # make iterable dataset object
148         train_iter, val_iter, test_iter = torchtext.data.BucketIterator.splits(
149             (train, val, test),
150             batch_sizes=[self.batch_size, 1, 1],
151             device=self.device,
152             repeat=False,
```

```
153             sort_key=lambda x: len(x.text),
154             sort_within_batch=True,
155         )
156         self.N_train_data = len(train)
157         self.N_val_data = len(val)
158         self.N_batches = int(
159             self.N_train_data / self.batch_size
160             + int(self.N_train_data % self.batch_size > 0)
161         )
162 
163         self.N_train_data = len(train)
164         self.N_val_data = len(val)
165         self.N_batches = int(
166             self.N_train_data / self.batch_size
167             + int(self.N_train_data % self.batch_size > 0)
168         )
169         self.log("N_train_data: %d  N_mini_batches: %d" % (self.N_train_data, self.N_batches)    )
170 
171 
172         # instantiate the dmm
173         self.dmm = DMM(input_dim=self.vocab_size, dropout=self.dropout)
174 
175         # setup optimizer
176         opt_params = {
177             "lr": self.lr,
178             "betas": (self.beta1, self.beta2),
179             "clip_norm": self.cn,
180             "lrd": self.lr_decay,
181             "weight_decay": self.wd,
182         }
183         self.adam = ClippedAdam(opt_params)
184         # set up inference algorithm
185         self.elbo = Trace_ELBO()
186         self.svi = SVI(self.dmm.model, self.dmm.guide, self.adam, loss=self.elbo)
187 
188         val_f = 10
189 
190         print("training dmm")
191         times = [time.time()]
192         for epoch in range(self.n_epoch):
193 
194             if self.ckpt_f > 0 and epoch > 0 and epoch % self.ckpt_f == 0:
195                 self.save_ckpt()
196 
197             # train and report metrics
198             train_nll = self._train_batch(train_iter, epoch,)
199 
200             times.append(time.time())
201             t_elps = times[-1] - times[-2]
202             self.log(
203                 "epoch %04d -> train nll: %.4f \t t_elps=%.3f sec"
```

```
204                 % (epoch, train_nll, t_elps)
205             )
206 
207             if epoch % val_f == 0:
208                 val_nll = self._validate(val_iter)
209         pass
210 
211     def save_ckpt(self):
212         """
213         saves the state of the network and optimizer for later
214         """
215         self.log("saving model to %s" % self.save_model)
216         torch.save(self.dmm.state_dict(), self.save_model)
217         self.log("saving optimizer states to %s" % self.save_opt)
218         self.adam.save(self.save_opt)
219 
220         pass
221 
222     def load_ckpt(self):
223         """
224         loads a saved checkpoint
225         """
226         assert exists(args.load_opt) and exists(
227             args.load_model
228         ), "--load-model and/or --load-opt misspecified"
229         self.log("loading model from %s..." % self.load_model)
230         self.dmm.load_state_dict(torch.load(self.load_model))
231         self.log("loading optimizer states from %s..." % self.load_opt)
232         self.adam.load(self.load_opt)
233 
234         pass
```