HW2: Language Modeling

Jiafeng Chen Francisco Rivera

February 20, 2019

1 Introduction

In this write-up, our main focus is language modeling. That is, given words in a sentence, can we predict the word that follows? We implemented a Trigram model, a embedding neural network model, an LSTM, and a few extensions—including pre-trained embeddings, ensemble models, and multi-head attention decoders.

2 Problem Description

To tackle language modeling, we start with sequences of words $w \in \mathcal{V}$ in some vocabulary $\mathcal{V}$ and aim to predict the last word in the sequence which we cannot observe. We can do this probabilistically by attempting to estimate,

$$p(w_t \mid w_1, \ldots, w_{t-1}) \tag{1}$$

that is, the conditional distribution over the last word conditional on the words leading up to it.

In particular, there will be a special word, $w_{\text{unk}} \in \mathcal{V}$ which represent an unknown word; we use this whenever we encounter a token we have not previously seen.

In some models, we represent words with dense embeddings. That is, each word gets assigned a vector $v \in \mathbb{R}^d$ where d is the embedding dimension. These embeddings are trained as part of the model, but can also be initialized to pre-trained values.

3 Model and Algorithms

3.1 Trigram model

In our trigram model, we aim to estimate the probability written in Equation 1. This conditional probability is intractable itself because it's likely that we've never seen the exact sequence of words $w_1, \ldots, w_{t-1}$. However, we can gain tractability by dropping words toward the beginning of the sequence, hoping that they don't affect the probability too much. That is, we hope that,

$$p(w_t \mid w_1, \ldots, w_{t-1}) \stackrel{?}{\approx} p(w_t \mid w_{t-2}, w_{t-1}).$$

Having replaced our first probability with a simpler one which conditions on less information, we can estimate the latter by its empirical sample estimator. In other words, we can take all the times in our training set when we've seen words w_{t-2}, w_{t-1} adjacent to each other, and consider the empirical distribution of the word that follows them. We represent this sample approximation as $\widehat{p}$ and write,

$$p(w_t \mid w_{t-2}, w_{t-1}) \approx \widehat{p}(w_t \mid w_{t-2}, w_{t-1}).$$

By doing this, we've solved most of the intractability of conditioning on the entire sentence $w_1, \ldots, w_{t-1}$, but we still have some of the same problems. Namely, it's possible that in our training set, we either haven't seen words w_{t-2} and w_{t-1} together before, or we've seen them only a very small number of times such that the empirical probability distribution becomes a poor approximation. (To avoid division by zero errors, we adopt the convention that empirical probabilities are all 0 if we haven't seen the words being conditioned on before.) We can fix this by also considering the probabilities,

$$p(w_t) \text{ and } p(w_t \mid w_{t-1})$$

which give us the unconditional probability of a word and the probability conditional on only the previous word. These have the benefit of being more tractable to estimate and the drawback of losing information. In the end, we calculate a blend of these three approximations:

$$\alpha_1 \widehat{p}(w_t \mid w_{t-2}, w_{t-1}) + \alpha_2 \widehat{p}(w_t \mid w_{t-1}) + (1 - \alpha_1 - \alpha_2)\widehat{p}(w_t).$$

Training the weights (α_1, α_2) loads up most of our weight on α_1 which suggests the latter two probabilities are better used as "tie-breakers" when conditioning on the previous bi-gram yields a small number of possibilities. In our final model, we use $(\alpha_1, \alpha_2) = (0.9, 0.05)$.

We conclude this section with discussion on implementation. In particular, one easy way to keep track of all the conditional probabilities would be to keep a three dimensional $|\mathcal{V}| \times |\mathcal{V}| \times |\mathcal{V}|$ tensor which keeps track of the trigram counts. However, vocabulary size is large, and this data structure can get prohibitively large very quickly. Moreover, we don't expect most trigrams to have any counts. Therefore, we use a sparse tensor to store these counts which reduces our memory usage to the number of distinct trigrams in the training dataset.

3.2 Neural Network Language Model

Following Bengio et al. (2003), we implement a neural network language model (NNLM). We model (1) by first assuming limited dependence:

$$p(w_i \mid w_1, \ldots, w_{i-1}) = p(w_i \mid w_{i-1}, \ldots, w_{i-k}),$$

i.e., the current word only depends on the past k words, a useful restriction motivated by n-gram models. Next, we convert input tokens $w_{i-1}, \ldots, w_{i-k}$ into embedding vectors $\{v_{i-t}\}_{t=1}^{k}$ and concatenate them into a dk vector $\boldsymbol{v}_{i-k:i-1}$. We then pass this vector into a multilayer perceptron network with a softmax output into $|\mathcal{V}|$ classes. We implement this efficiently across a batch by using a convolution operation, since convolution acts like a moving window of size k. This way we can generate $T - k + 1$ predictions for a sentence of length T. We depict the convolution trick in Figure 1.

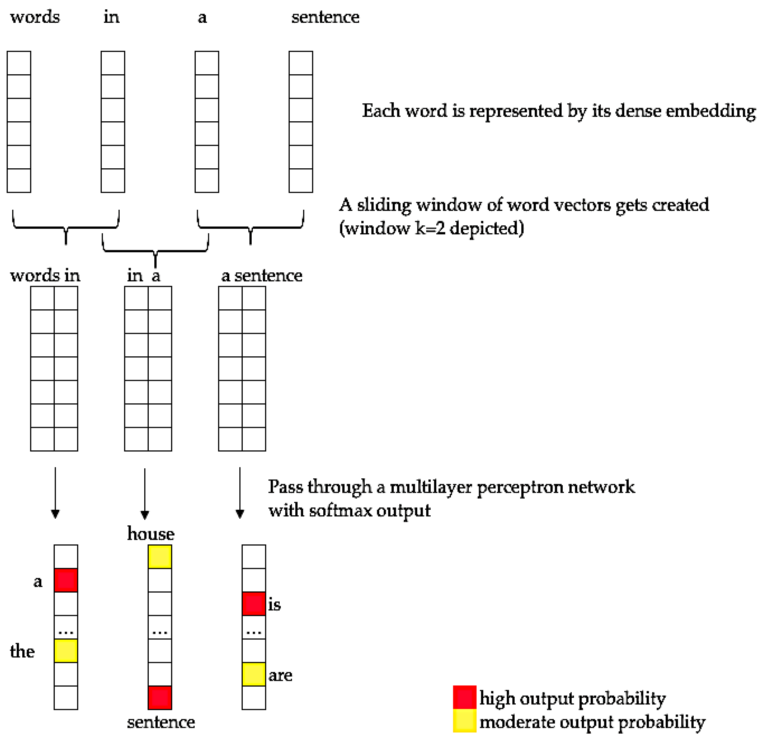

Figure 1: Diagram for Neural Network Language Model

3.3 LSTM

A concern with our trigram model is that it completely ignores words more than two positions before the word we wish to predict. To the extent we believe these words are predictive (personal experience with language suggest that they should be!), the trigram model has an inherent limitation in its ability to model that dependence.

One way to combat this is with an LSTM, the architecture of which is depicted in Figure 2. At a high level, the LSTM functions by keeping track of three vectors: v_t, C_t, and h_t. The first of these vectors is simply a dense embedding for the word w_t. The h_t and C_t vectors are state representations of the model, which are dependent on previous words and give the model a "memory." The LSTM can thus theoretically condition on all previous text, and in practice exhibits long-term memory through its architecture on C_t which encourages only intentional changes over time.

The LSTM is formally characterized by the following equations which determine the evolution of these vectors,

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{3}$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{4}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{5}$$
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o) \tag{6}$$
$$h_t = o_t \odot \tanh(C_t) \tag{7}$$

Roughly speaking, their intuitions are as follows: $\tilde{C}_t$ represents the new information that might be relevant for encoding into long-term memory. f_t captures the information that needs to be deleted from long-term memory, and i_t captures the places where information needs to be added. Then, these are combined to determine the new C_t. The hidden state roughly captures a filtered version (captured by the multiplication with o_t) of the cell-state.

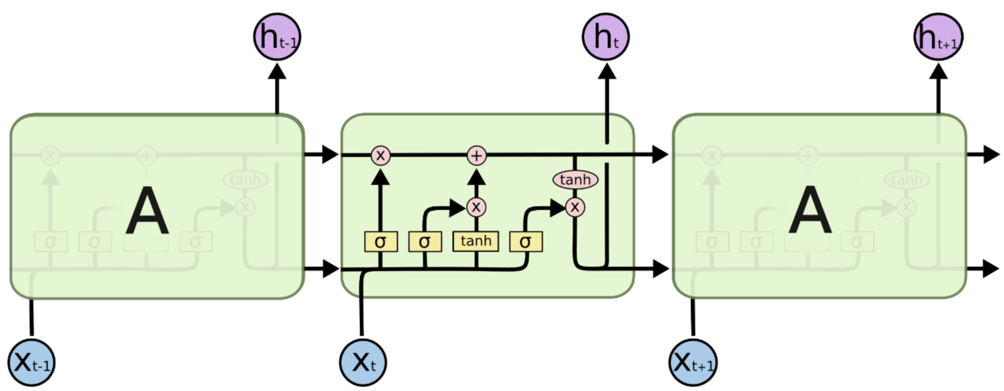

Figure 2: Depiction of LSTM inner-workings (Olah, 2015)

When creating an LSTM, we have two main hyper-parameters to decide on, the embedding dimension and the hidden dimension. We experiment with various choices for these hyper-parameters, finding that 300 for both maximizes our performance on the validation set using early-stopping as a regularization technique when we train.

3.4 Multi-head Attention

Following Vaswani et al. (2017), we implement a variant of the multi-head attention decoder network. Instead of passing the last hidden state from an LSTM h_t to predict h_{t+1}, we use a concatenation of h_t and a *context vector* $c_t = a_1 h_1 + \cdots + a_{t-1} h_{t-1}$ for *attention values* $a_1, \ldots, a_{t-1}$ residing in the unit simplex. Following Vaswani et al. (2017), we use the *scaled attention* mechanism, where

$$\boldsymbol{a} = \mathsf{Softmax}\left(\left\{\frac{h_i^T h_t}{\sqrt{\dim(h_t)}}\right\}_{i=1}^{t-1}\right).$$

In *multi-head attention*, we repeat the attention mechanism above on different linear projections of $h_1, \ldots, h_t$, with the motivation being that we wish to capture similarity on different projections of words—one attention layer could be capturing grammar, another semantics, a third pronoun association, etc. We depict the architecture in Figure 3, for $t = 3$ and predicting $t + 1$.

Certain computational tricks need to be employed for efficient utilization of the GPU. Unlike the encoding attention network, the decoder cannot condition on future information when predicting the future. As a result, each attention layer can only look at past hidden states. We parallelize the procedure and take advantage of GPU hardware by applying attention as usual, computing every inner product $h_i^T h_j$ for all i, j, and use a mask that sets entries with $h_i^T h_j$ to $-\infty$ if $j \leq i$ (which correspond to the forbidden attention links by looking ahead) before applying the softmax.

4 Experiments

We train a few models and document hyperparameters in Table 1. Results are displayed in Table 2.

For a simple model with only three undetermined parameters,[1] tri-grams performs exceptionally well. We conjecture that this is because training set sentences are not very long, so that most of the information is captured by the immediate

[1] Of course, the training data is being memorized to make for a much larger model complexity, but there is only one canonical way to do this.

predecessors of the word. In the limit, in a sentence with only three words, a trigram model conditions on all the information.

The NNLM appears to underperform the trigram model by a bit, even when we condition on the past seven words rather than the past two. We believe the network architecture is not flexible enough to efficiently learn the regression function without significantly overfitting on a relatively small data set.

LSTMs out-perform the previous two models, which we attribute to their ability to not only condition on local information (as do the previous models) but also on longer dependences. Unlike an NNLM where if we increase the window we look at, the model becomes too prone to overfitting, the LSTM architecture encourages "memorizing" only relevant features that get mostly propagated through the long-term memory "high-way." We also discover that initializing word embeddings to pre-trained values increases performance by 10 perplexity points—something we attribute to the pre-trained word embeddings capturing deeper relationships before we start training, and training reinforcing the useful ones which it might not have discovered had it been randomly initialized.

Of the pre-trained embeddings, we find most success with word2vec. However, we find that different embeddings result in different model predictions. This suggests an ensemble model where we combine the predictions of LSTMs initialized to different embeddings. Such an ensemble gives us further predictive power.

- Trigram is decent (3-parameters).
- NN language model is bad.
- LSTM is good
- Initiate to word2vec is good
- Attention is bad because training set is small and short
- Ensemble is very good since it's ensemble; though the LSTM outputs are fairly correlated

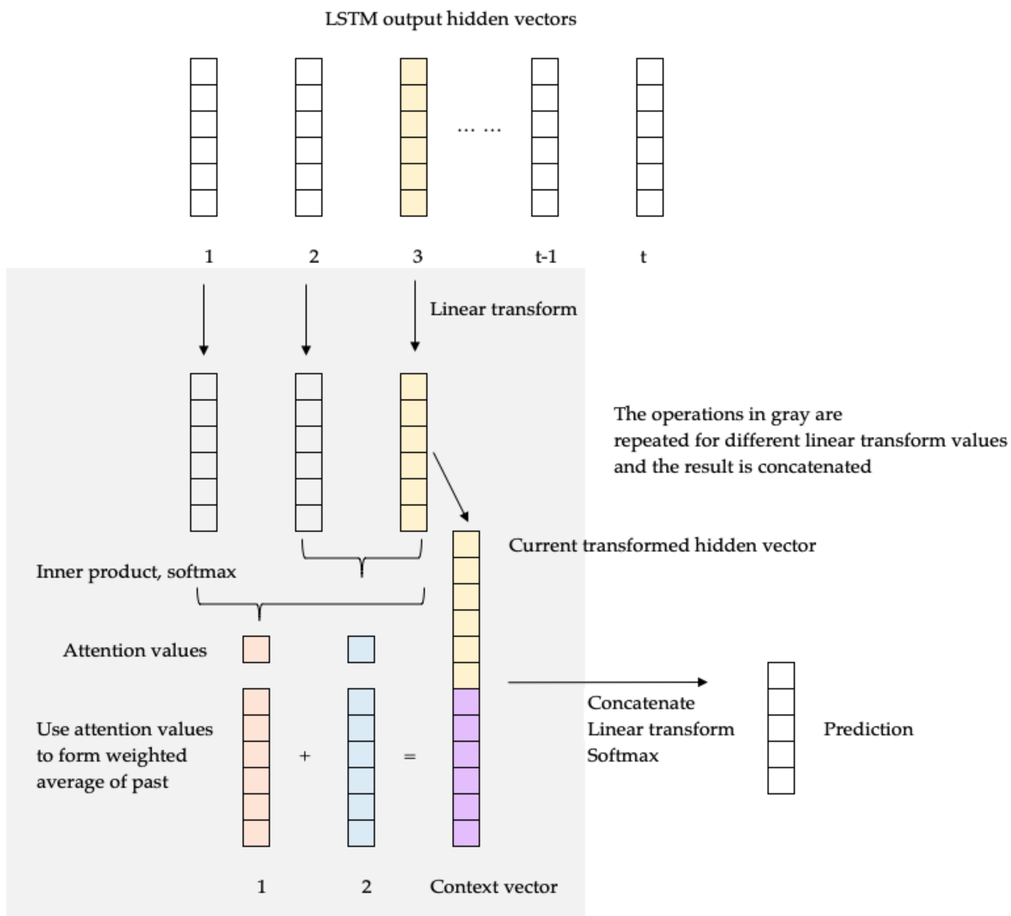

Figure 3: Diagram for Multi-head Attention. In this diagram, we predict the fourth word in the sentence by conditioning on the first three. We compute attention values of h_3 with h_2 and h_1 and concatenate h_3 with a context vector $a_1 h_1 + a_2 h_2$, where a_i are the attention values. We pass the resulting vector through a linear transformation and softmax output. In multi-head attention, we repeat the process for different projections of h_1, h_2, h_3 and concatenate the results.

model name	specifications
`attention_300.200.0.1dropout`	Multihead attention (3 heads), 300 embedding, 200 hidden, 0.1 dropout on linear transform
ensemble	Ensembling (equal weights) all LSTM/attention based models
`lstm_charngram.100d`	100 embedding, 100 hidden, initiate to charngram
`lstm_fasttext.en.300d`	300 embedding, 300 hidden, initiate to fasttext
`lstm_glove.42B.300d`	300 embedding, 300 hidden, initiate to GloVe-42B
`lstm_glove.twitter.27B.200d`	200 embedding, 200 hidden, initiate to GloVe-Twitter
`lstm_word2vec.300d`	300 embedding, 300 hidden, initiate to word2vec
`nnlang.300.7.300`	NNLM, 300 embedding, 300 hidden, $k = 7$
Trigram	$\alpha_1 = 0.05 = \alpha_2, \alpha_3 = 0.9$

Table 1: Hyperparameter specifications for models that we train. All LSTM models are one-layer (we experimented with two layers and dropout, but did not achieve gains). All models are trained with Adam and learning rate 10^{-3} *over 3 epochs (which is usually when validation error starts to increase). The trigram probabilities are somewhat arbitrarily chosen: We initially trained the trigram weights over the training set and observe that the gradient direction is always in the direction of increasing the weight for the trigram term,* α_3*; we then decided to use a* $0.9, 0.05, 0.05$ *split, putting most weight on trigrams.*

model name	Loss	MAP@20	Perplexity
ensemble	4.656	0.350	105.248
attention_300.200.0.1dropout	4.977	0.328	145.081
lstm_charngram.100d	4.931	0.329	138.500
lstm_fasttext.en.300d	4.896	0.328	133.750
lstm_glove.42B.300d	4.879	0.331	131.564
lstm_glove.twitter.27B.200d	4.917	0.331	136.595
lstm_word2vec.300d	4.862	0.332	129.220
nnlang.300.7.300	5.691	0.254	296.330

Table 2: Performance metrics for different models

5 Conclusion

References

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Olah, C. (2015). Understanding LSTM networks.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

A Model implementation

Listing 1: Trigram model implementation

```
import torch
from torch import Tensor
from torch import sparse as sp
from tqdm import tqdm_notebook as tqdm
import pandas as pd
import numpy as np
from torch import nn

```

```python
def sparse_select(dims, indices, t):
    """
    Select sparse tensor t on on dimensions dims and indices chosen by indices.
    Equivalent to t[i_0, i_1, ...] where i_d = : if d is not in dims and
    i_(dims[j]) = indices[j] for all j
    """
    if type(dims) is not list:
        dims = [dims]
    if type(indices) is not list:
        indices = [indices]

    t_indices = t._indices()
    t_values = t._values()
    selector = torch.ones(t_indices.shape[-1]).byte()
    for dim, index in zip(dims, indices):
        selector = selector & (t._indices()[dim, :] == index)
    remaining_dimensions = list(filter(lambda x: x not in dims,
                               range(t_indices.shape[0])))

    indices_selected = t_indices[:, selector][remaining_dimensions, :]
    values_selected = t_values[selector]
    new_shape = torch.Size(t.shape[d] for d in remaining_dimensions)

    out = sp.FloatTensor(indices_selected, values_selected, new_shape)
    return out



class Trigram:
    def __init__(self, TEXT):
        self.log_weights = torch.zeros(3, requires_grad=True)
        self.TEXT = TEXT
        self.V = len(TEXT.vocab)

    def get_probabilities(self, train_iter):
        TEXT = self.TEXT
        V = self.V
        unigram = sp.FloatTensor(V)
        bigram = sp.FloatTensor(V,V)
        trigram = sp.FloatTensor(V,V,V)

        for batch in tqdm(train_iter):

            i = batch.text.values.flatten().unsqueeze(0)
            unigram_counts = sp.FloatTensor(
                i, torch.ones(i.shape[1]), torch.Size([V])
            )

            unigram += unigram_counts

            ii = torch.stack([batch.text.values[:-1,:], batch.text.values[1:,
                :]]).view(2, -1)
            bigram_counts = sp.FloatTensor(
                ii, torch.ones(ii.shape[-1]), torch.Size([V, V])
            )
            bigram += bigram_counts

            iii = torch.stack([batch.text.values[:-2,:], batch.text.values[1:-1, :],
                batch.text.values[2:, :]]).view(3, -1)
```

```
            trigram_counts = sp.FloatTensor(
                iii, torch.ones(iii.shape[-1]), torch.Size([V, V, V])
            )
            trigram += trigram_counts

        unigram = unigram.coalesce()
        unigram = unigram / sp.sum(unigram)

        bigram = bigram.coalesce()
        trigram = trigram.coalesce()

        bigram_df = pd.DataFrame(np.hstack([bigram.indices().numpy().T,
                                bigram.values().numpy()[:, np.newaxis]]),
                    dtype=int, columns=['word1', 'word2', 'counts'])

        trigram_df = pd.DataFrame(np.hstack([trigram.indices().numpy().T,
                                trigram.values().numpy()[:, np.newaxis]]),
                    dtype=int, columns=['word1', 'word2', 'word3', 'counts'])

        bigram_df['prob'] = ((bigram_df['counts'] / bigram_df.groupby(['word1'])
                              .transform('sum')['counts']))

        bigram_ind = torch.from_numpy(bigram_df[['word1', 'word2']].values.T)
        bigram_val = torch.from_numpy(bigram_df['prob'].values)
        bigram = torch.sparse.FloatTensor(bigram_ind, bigram_val, bigram.shape)

        trigram_df['prob'] = (trigram_df['counts'] / trigram_df.groupby(['word1', 'word2'])
                              .transform('sum')['counts'])

        trigram_ind = torch.from_numpy(trigram_df[['word1', 'word2', 'word3']].values.T)
        trigram_val = torch.from_numpy(trigram_df['prob'].values)
        trigram = torch.sparse.FloatTensor(trigram_ind, trigram_val, trigram.shape)

        self.unigram = unigram.float()
        self.bigram = bigram.float()
        self.trigram = trigram.float()


    def predict(self, past_two_words):
        weights = torch.softmax(self.log_weights, dim=0)
        output_batch = torch.zeros(len(past_two_words), self.V)
        for i, pair in enumerate(past_two_words):
            bi = sparse_select(0, pair[-1], self.bigram)
            tri = sparse_select([0,1], pair.tolist(), self.trigram)
            uni = self.unigram
            output_batch[i, :] = (
                tri.to_dense() * weights[0]
                + bi.to_dense() * weights[1]
                + uni.to_dense() * weights[2])
        return output_batch

    def __call__(self, batch_text):
        packaged = torch.stack([batch_text.values[:-1,:], batch_text.values[1:,:]]).view(2, -1).t()
        return self.predict(packaged)

cross_entropy_loss = nn.CrossEntropyLoss()
```

```
def trigram_loss_fn(model, batch):
    pred = model(batch.text.values)
    labels = batch.target[1:,:].flatten()
    loss = cross_entropy_loss(pred, labels)
    return loss
```

Listing 2: NNLM

```
import torch
from torch import nn
import namedtensor
from namedtensor.nn import nn as namednn


class NNLangModel(namednn.Module):
    def __init__(self, TEXT, embedding_dim, kernel_size, hidden, dropout=.5):
        super().__init__()
        V = len(TEXT.vocab)
        pad_idx = TEXT.vocab.stoi['<pad>']

        self.embed = namednn.Embedding(num_embeddings=V,
                                       embedding_dim=embedding_dim,
                                       padding_idx=pad_idx)
        self.conv = namednn.Conv1d(embedding_dim, embedding_dim,
                                   kernel_size=kernel_size).spec('embedding', 'seqlen')

        self.w1 = namednn.Linear(embedding_dim, hidden).spec('embedding', 'hidden')
        self.w2 = namednn.Linear(hidden, hidden).spec('hidden', 'hidden2')
        self.w3 = namednn.Linear(hidden, V).spec('hidden2', 'classes')
        self.dropout = namednn.Dropout(dropout)

    def forward(self, batch_text):
        embedded = self.embed(batch_text)
        conved = self.conv(embedded)
        h1 = self.w1(conved).tanh()
        h2 = self.w2(self.dropout(h1)).tanh()
        out = self.w3(self.dropout(h2))
        return out

nn_lang_loss = namednn.CrossEntropyLoss().spec('classes')
def nn_lang_loss_fn(model, batch):
    output = model(batch.text)
    size = output.shape['seqlen']
    target_size = batch.target.size('seqlen')
    target = (batch.target[{'seqlen' : slice(target_size-size, target_size)}])
    return nn_lang_loss(output, target)
```

Listing 3: LSTM

```
from namedtensor.nn import nn as nnn

class LSTM(nnn.Module):
    """
    LSTM implementation for sentence completion.
    """
    def __init__(self, TEXT,
                 embedding_dim=100,
                 hidden_dim=150,
```

```
                 num_layers=1,
                 dropout=0):
        super().__init__()

        pad_idx = TEXT.vocab.stoi['<pad>']

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embedding_dim,
                                   padding_idx=pad_idx)

        self.lstm = nnn.LSTM(input_size=embedding_dim,
                             hidden_size=hidden_dim,
                             num_layers=num_layers,
                             dropout=dropout) \
                        .spec("embedding", "seqlen")

        self.w = nnn.Linear(in_features=hidden_dim,
                            out_features=len(TEXT.vocab)) \
                        .spec("embedding", "classes")

    def forward(self, batch_text):
        embedded = self.embed(batch_text)
        hidden_states, _ = self.lstm(embedded)
        log_probs = self.w(hidden_states)

        return log_probs


ce_loss = nnn.CrossEntropyLoss().spec('batch')

def lstm_loss(model, batch):
    """
    Calculate loss of the model on a batch.
    """
    return ce_loss(model(batch.text), batch.target)
```

Listing 4: LSTM-attention

```
from namedtensor.nn import nn as nnn
from namedtensor import ntorch
import torch
from namedtensor import NamedTensor
from numpy import inf


class MaskedAttention(nnn.Module):
    def __init__(self, cuda=True):
        super().__init__()
        self.cuda_enabled = cuda

    def forward(self, hidden):
        dotted = (hidden * hidden.rename("seqlen", "seqlen2")).sum("embedding")
        mask = torch.arange(hidden.size('seqlen'))
        mask = (NamedTensor(mask, names='seqlen') < NamedTensor(mask,
            names='seqlen2')).float()
        mask[mask.byte()] = -inf
        if self.cuda_enabled:
            attn = ((dotted + mask.cuda()) / (hidden.size("embedding") **
                .5)).softmax('seqlen2')
```

```
        else:
            attn = ((dotted + mask) / (hidden.size("embedding") **
                .5)).softmax('seqlen2')
        return (attn * hidden.rename('seqlen', 'seqlen2')).sum('seqlen2')

class LSTM_att(nnn.Module):
    """
    LSTM implementation for sentence completion.
    """
    def __init__(self, TEXT,
                 embedding_dim=100,
                 hidden_dim=150,
                 num_layers=1,
                 dropout=0,
                 nn_dropout=.5,
                 **kwargs):
        super().__init__()

        pad_idx = TEXT.vocab.stoi['<pad>']

        self.embed = nnn.Embedding(num_embeddings=len(TEXT.vocab),
                                   embedding_dim=embedding_dim,
                                   padding_idx=pad_idx)

        self.lstm = nnn.LSTM(input_size=embedding_dim,
                             hidden_size=hidden_dim,
                             num_layers=num_layers,
                             dropout=dropout) \
                        .spec("embedding", "seqlen")


        self.w1 = (nnn.Linear(in_features=hidden_dim, out_features=hidden_dim)
                   .spec("embedding", "embedding"))
        self.w2 = (nnn.Linear(in_features=hidden_dim, out_features=hidden_dim)
                   .spec("embedding", "embedding"))
        self.w3 = (nnn.Linear(in_features=hidden_dim, out_features=hidden_dim)
                   .spec("embedding", "embedding"))


        self.lins = [self.w1, self.w2, self.w3]
        self.attn = MaskedAttention(**kwargs)

        h_len = len(self.lins) + 2
        self.w = nnn.Linear(in_features=hidden_dim * h_len,
                            out_features=len(TEXT.vocab)) \
                        .spec("embedding", "classes")
        self.dropout = nnn.Dropout(nn_dropout)


    def forward(self, batch_text):
        embedded = self.embed(batch_text)
        H, _ = self.lstm(embedded)
        joint = ntorch.cat([H, self.attn(H)] + [self.attn(l(H)) for l in self.lins],
            "embedding")
        log_probs = self.w(self.dropout(joint))
        return log_probs


ce_loss = nnn.CrossEntropyLoss().spec('classes')
```

```python

def lstm_loss(model, batch):
    """
    Calculate loss of the model on a batch.
    """
    return ce_loss(model(batch.text), batch.target)
```

Listing 5: Ensemble

```python
from namedtensor import ntorch

class Ensemble:
    def __init__(self, *models):
        self.models = models

    def __call__(self, batch_text):
        return ntorch.stack(
            [model(batch_text) for model in self.models], "model").mean("model")
```