HW4: All about Attention

Jiafeng Chen Yufeng Ling Francisco Rivera

April 5, 2019

1 Introduction

In this writeup we consider natural language inference–given a premise and a hypothesis, can we determine the entailment and contradiction relationship between them. The key to the model is the attention architecture, which serves to decompose the problem into aligned subphrases. Not only does this design make training parellelizable, it also significantly reduces the number of parameters while delivering state-of-the-art results.

2 Problem Description

In this writeup, we consider the problem of natural language inference. Let

$$\boldsymbol{a} = (a_1, \dots, a_{\ell_a}) \tag{1}$$

be the premise of length ℓ_a and let

$$\boldsymbol{b} = (b_1, \dots, b_{\ell_b}) \tag{2}$$

be the hypothesis of length ℓ_b. Each $a_i, b_j \in \mathbb{R}^d$ is a word embedding vector of dimension d. Our goal is to, given the input pair $\boldsymbol{a}, \boldsymbol{b}$, predict the relationship $y \in \{y_1, \dots, y_C\}$ where C is the number of output classes.

3 Model and Algorithms

3.1 Decomposable Attention Model

This section follows the architecture of Parikh et al. (2016).

3.1.1 Vanilla model

We first look at the most basic approach that is the foundation of this architecture. We start by setting the inputs $\bar{\mathbf{a}}, \bar{\mathbf{b}}$ to be the premise and hypothesis $\mathbf{a}, \mathbf{b}$ themselves. We generate attention weight matrix by softmaxing over

$$e_{ij} := F'(\bar{a}_i, \bar{b}_j) = F(\bar{a}_i)^\intercal F(\bar{b}_j). \tag{3}$$

Note that we made the simplification of setting F' to be the dot product of $\bar{a}_i$ and $\bar{b}_j$ through the same feed-forward neural network, which reduces the number of operations from $O(\ell_a \times \ell_b)$ to $O(\ell_a + \ell_b)$. The attended phrases are then

$$\begin{aligned}\beta_i &:= \sum_{j=1}^{\ell_b} \frac{\exp(e_{ij})}{\sum_{k=1}^{\ell_b} \exp(e_{ik})} \bar{b}_j, \\ \alpha_j &:= \sum_{i=1}^{\ell_b} \frac{\exp(e_{ij})}{\sum_{k=1}^{\ell_a} \exp(e_{kj})} \bar{a}_i. \end{aligned} \tag{4}$$

A key implementation detail is that we need to apply masking to the $\{e_{ij}\}$ matrix before softmaxing to avoid putting attention on padding.

Next, we compare the aligned attended phrases to the original ones by concatenating them and applying a feed-forward neural network G.

$$\begin{aligned}\mathbf{v}_{1,i} &:= G([\bar{a}_i, \beta_i]) \quad \forall i \in [1, \ldots, \ell_a], \\ \mathbf{v}_{2,j} &:= G([\bar{b}_j, \alpha_j]) \quad \forall j \in [1, \ldots, \ell_b]. \end{aligned} \tag{5}$$

We then apply sum-over-time pooling, with padding masked out, to generate the penultimate vectors

$$\mathbf{v}_1 = \sum_{i=1}^{\ell_a} \mathbf{v}_{1,i}, \qquad \mathbf{v}_2 = \sum_{j=1}^{\ell_b} \mathbf{v}_{2,j}. \tag{6}$$

Finally, we apply a feed-forward neural network H to the concatenated vectors to generate the unnormalized predictions for each class

$$\widehat{y} = H([\mathbf{v}_1, \mathbf{v}_2]) \in \mathbb{R}^C. \tag{7}$$

The prediction is $\widehat{y} = \arg\max_i \widehat{\mathbf{y}}_i$. In the training of this model, we use the multi-class cross-entropy loss as the loss function.

$$L(\theta_F, \theta_G, \theta_H) = \frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} y_c^{(n)} \log \frac{\exp(\widehat{y}_c)}{\sum_{c'=1}^{C} \exp(\widehat{y}_{c'})}. \tag{8}$$

3.1.2 Intra-Sentence Attention

We can improve the model by incorporating intra-sentence attention. Instead of having $(\bar{\mathbf{a}}, \bar{\mathbf{b}}) = (\mathbf{a}, \mathbf{b})$, we add self-attention to the input. We let the unnormalized attention weights be

$$f_{ij} = F_{\text{intra}}(a_i)^\intercal F_{\text{intra}}(a_j), \tag{9}$$

where F_{intra} is a feed-forward network. We then create the self-aligned phrases

$$a'_i = \sum_{j=1}^{\ell_a} \frac{\exp(f_{ij} + d_{i-j})}{\sum_{k=1}^{\ell_a} \exp(f_{ik} + d_{i-k})} a_j. \tag{10}$$

In the above equation, $d_{i-j} \in \mathbb{R}$ is the bias term based on distance, which is shared throughout sentences. Moreover, we bucket the terms such that all distances greater than 10 have the same bias. In the end, we use $\bar{a}_i = [a_i, a'_i]$ and $\bar{b}_j = [b_j, b'_j]$ as inputs.

3.2 Latent Variable Mixture Model

One way to try to get increased performance out of the Decomposable Attention Model model is to ensemble multiple copies of the model with different weights. In particular, we explore two variants, one with an "exact" ensemble where every model is queried by the ensemble to get a marginal likelihood, and one where we use an inference network and an ELBO to simplify the training step.

3.2.1 Exact Ensemble Model

For our ensemble, we will use K vanilla decomposable attention models. Each one of these models gives us a distribution over the classes,

$$p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta_k).$$

We introduce a uniform discrete latent variable which represents which model to listen to,

$$c \sim \text{Unif}(1, \dots, K).$$

Then, the marginal likelihood will be given by marginalizing over c, giving us

$$p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta) = \sum_{c=1}^{K} p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta_c).$$

The K models can be trained simultaneously through backprop using this equation for the ensemble likelihood.

3.2.2 Latent Variable Mixture Model

A problem with the above model is that training time scales linearly with K. Because we are propagating gradients through all K models for every training example, training it takes K times as long as training an individual model. We wish for each model $c \in \{1, \dots, K\}$ to specialize in some of the problems. To this end, we create an inference network $q(c \mid y, \boldsymbol{a}, \boldsymbol{b})$. This network follows a similar architecture to the aligned attention vanilla model, except it outputs weights for each of the K models rather than the 4 labels. We sample from the distribution output by the inference network in order to avoid evaluating all models, using the ELBO,

$$\log p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta) \geq E_{c \sim q(c \mid y, \boldsymbol{a}, \boldsymbol{b})} \log p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta_c) - \text{KL}(q(c \mid y, \boldsymbol{a}, \boldsymbol{b}) \mid\mid p(c))$$

where the KL term can be evaluated analytically and penalizes our inference network diverging from a uniform choice of model. The random variable c is discrete so we cannot employ the reparameterization trick to differentiate the expectation, so instead we use REINFORCE, giving us the gradient to update on as,

$$
\begin{aligned}
&= \nabla E_{c \sim q(c \mid y, \boldsymbol{a}, \boldsymbol{b})} \log p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta_c) \\
&= E_{c \sim q(c \mid y, \boldsymbol{a}, \boldsymbol{b})} \left[\nabla \log p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta_c) + \log p(y \mid \boldsymbol{a}, \boldsymbol{b}; \theta_c) \nabla \log q(c \mid y, \boldsymbol{a}, \boldsymbol{b}) \right]
\end{aligned}
$$

When testing, we can enumerate through the models as in the exact ensemble model.

4 Experiments

4.1 Set-up and hyper-parameters

For each of our four models, we train each one on a Google Collab instance until the instance times out. This means that we train a variable number of epochs depending on each model's epoch training time as well as some variability from the instance's longevity.

For the vanilla model, we follow the hyper-parameter recommendations from the paper. This means we embed into a 200 dimensions, and use feed-forward neural networks with two layers and ReLU activation. We similarly employ dropout with probability 0.2 of dropping out. For the intra-attention variant, we use the same hyperparameters and treat all distances bigger than 10 the same.

The ensemble models use the vanilla decomposable attention model. Our exact ensemble uses 3 models, while the VAE approximation uses 4. We change these numbers because the VAE approximation can handle more models with less of a cost on training time.

For training, we use Adam with a learning rate of $1e^{-4}$ for the vanilla model and $1e^{-3}$ for the other models. We arrive at this optimizer through experimentation. We found more success with using Adam than Adagrad as recommended in Parikh et al. (2016), as well as in using smaller learning rates than suggested by that paper.

model name	Validation loss	Validation accuracy
`Vanilla`	0.612	74.44%
`Intra-Attn`	0.745	62.06%
`Exact-Ensemble`	0.650	71.65%
`VAE Ensemble`	0.783	66.01%

Table 1: Performance metrics for different models

4.2 Results

The performance of our trained models appears in Table 1. These numbers merit some discussion. The most successful model is the vanilla. We attribute this outperformance to the model's simplicity which allowed it to train for more epochs. We trained for a relatively small number of epochs relative to those reported in the literature, so we fully expect that if trained for longer, the more complex models would outperform.

Between the exact ensemble and the VAE ensemble, there are two interesting stylized facts to draw. The first is that the VAE ensemble's speed advantage fell short of preliminary expectations because each model had to be evaluated on a subset of a batch. Thus, while each training example was only evaluated by one model, the batching advantage was reduced. We increased the batch size by a factor of two to combat this problem, but GPU memory constraints made it unfeasible to increase it further.

The second stylized point is that the VAE model seems to have a higher validation accuracy than might be expected from its validation loss (e.g. the intra-attention model has a better validation loss, but worse accuracy). We attribute this to model specialization. For instance, if for a particular example, one model is sure of the correct label, and the other models have no predictive power (i.e. return a uniform distribution over labels), the ensemble will pick the correct answer but have a moderately high loss because the probabilities given by the specialized model will be averaged with uniforms given by the other models.

Our belief in specialization is strengthened by inspection of the inference network q. We find that the recommendations that q makes for which model to can vary moderately from the uniform that we would get under no specialization.

However, manual inspection of which models get recommended for which sentences did not yield any obvious patterns. It is possible that the model is specializing in a way that is difficult to articulate, but more work would be required to better understand the qualitative aspects of this specialization.

4.3 Visualizing attention

We can also introspect on the vanilla attention model by looking at the attention weights the model outputs for certain examples. To do this, we visualize both the log attention weights as well as the softmaxes over both possible dimensions.

For example, we can compare the sentences "a snowboarder jumps off the snow" and "A skier jumps off the snow" in Figure 1. We use different color-maps to emphasize the left-most graph is in log-space, whereas the other two are in probability space. The weights exhibit intuitively desirable behavior. The subjects of both sentences are being compared (snowboarder versus skier) as well as their actions ("jumps" in both) and location ("snow").

We also include two other examples for reference in Figure 2 and Figure 3.

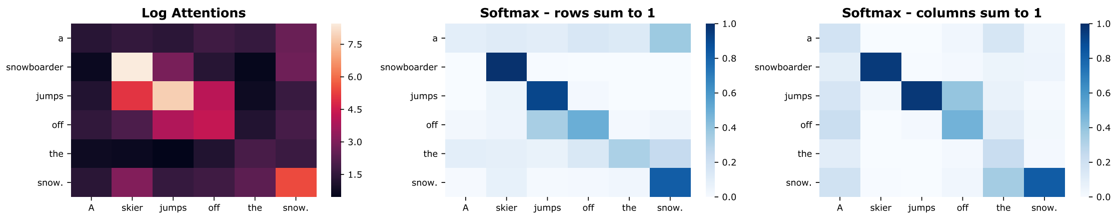

Figure 1: Attention weights for ski/snow sentences.

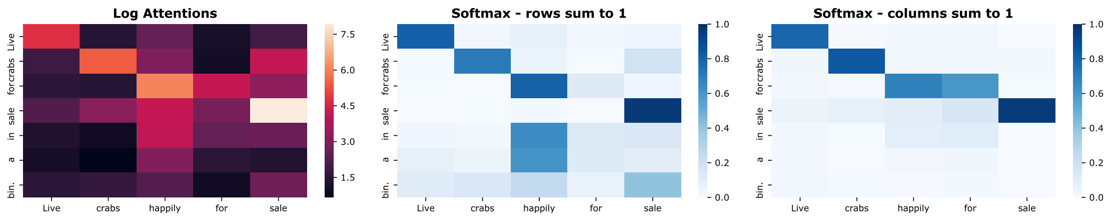

Figure 2: Attention weights for crabs example

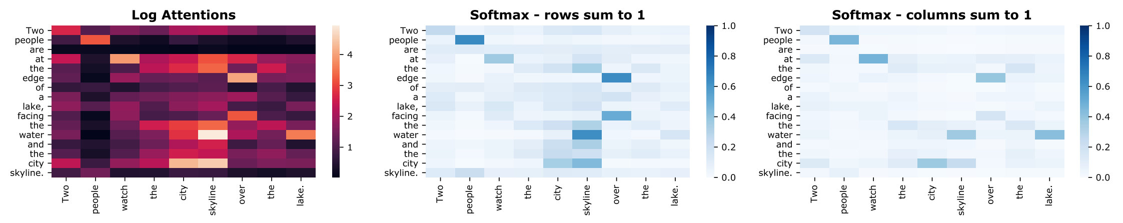

Figure 3: Attention weights for skyline example

References

Parikh, A. P., Täckström, O., Das, D., and Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.

A Model implementations

Listing 1: Decomposable Attention

```
from namedtensor import ntorch
from namedtensor.nn import nn as nnn

class FeedFwd(nnn.Module):
    def __init__(self, d_in, d_out, name_in, name_out,
                 dropout_p=.2, hidden_n=200):
        super().__init__()
        self.w1 = nnn.Linear(d_in, hidden_n).spec(name_in, "hidden")
        self.w2 = nnn.Linear(hidden_n, d_out).spec("hidden", name_out)
        self.drop = nnn.Dropout(p=dropout_p)

    def forward(self, x):
        x = ntorch.relu(self.w1(x))
        x = self.drop(ntorch.relu(self.w2(x)))
        return x


class DecompAttn(nnn.Module):
    def __init__(
            self,
            TEXT,
            LABEL,
            embed_dim=200,
            input_dim=None,
            dropout=0.2):
        super().__init__()

        padding_idx = TEXT.vocab.stoi['<pad>']
        self.padding_idx = padding_idx
        original_embed_dim = TEXT.vocab.vectors.size('embedding')
        num_classes = len(LABEL.vocab)
        self.embed_dim = embed_dim

        # this doesn't get updated
        self.embed = nnn.Embedding(TEXT.vocab.vectors.size('word'), embed_dim,
                                   padding_idx=padding_idx) \
            .from_pretrained(TEXT.vocab.vectors.values)

        # self.embed.weight.requires_grad = True

        # project the unchanged embedding into something smaller
        self.embed_proj = nnn.Linear(original_embed_dim, embed_dim, bias=False) \
            .spec('embedding', 'embedding')

        if input_dim is None:
            input_dim = embed_dim

        self.attn_w = FeedFwd(input_dim, embed_dim,
                              'embedding', 'attnembedding', dropout_p=dropout)

        self.match_w = FeedFwd(input_dim * 2, embed_dim,
                               'embedding', 'matchembedding', dropout_p=dropout)
        self.classifier_w = FeedFwd(embed_dim * 2, num_classes,
                                    'matchembedding', 'classes', dropout_p=0)

    def process_input(self, sentence, seqlen_dim):
        return self.embed_proj(self.embed(sentence))
        # return self.embed(sentence)

    def forward(self, hypothesis, premise, debug=False):
        attn_w, match_w, classifier_w = (
            self.attn_w, self.match_w, self.classifier_w)
        premise = premise.rename('seqlen', 'premseqlen')
        hypothesis = hypothesis.rename('seqlen', 'hypseqlen')

        premise_mask = (premise != self.padding_idx).float()
        hypothesis_mask = (hypothesis != self.padding_idx).float()

        log_mask = (1 - premise_mask * hypothesis_mask) * (-1e3)

        # Embedding the premise and the hypothesis
        premise_embed = self.process_input(premise, 'premseqlen')
        hypothesis_embed = self.process_input(hypothesis, 'hypseqlen')

        # Attend
        premise_keys = (attn_w(premise_embed))
        hypothesis_keys = (attn_w(hypothesis_embed))

        log_alignments = (
            ntorch.dot('attnembedding', premise_keys, hypothesis_keys)
             + log_mask)

        premise_attns = (log_alignments).softmax(
            'hypseqlen').dot('hypseqlen', hypothesis_embed)
        hypothesis_attns = (log_alignments).softmax(
            'premseqlen').dot('premseqlen', premise_embed)
        premise_concat = ntorch.cat(
            [premise_embed, premise_mask * premise_attns], 'embedding')
        hypothesis_concat = ntorch.cat(
            [hypothesis_embed, hypothesis_mask * hypothesis_attns], 'embedding')

        # Compare
        compare_premise = premise_mask * match_w(premise_concat)
        compare_hypothesis = hypothesis_mask * match_w(hypothesis_concat)

        # Aggregate
        result_vec = ntorch.cat([
            compare_premise.sum('premseqlen'),
            compare_hypothesis.sum('hypseqlen')],
            'matchembedding')

        if debug:
            return classifier_w(result_vec), log_alignments
        return classifier_w(result_vec)


class DecompAttnWithIntraAttn(DecompAttn):
    def __init__(
            self,
            TEXT,
            LABEL,
            intra_dropout=.2,
            embed_dim=200,
            max_distance=10,
            **kwargs):
        super().__init__(TEXT, LABEL, embed_dim=embed_dim,
            input_dim=2 * embed_dim, **kwargs)
        self.max_distance = max_distance
        self.distance_embed = nnn.Embedding(num_embeddings=max_distance + 1,
                                            embedding_dim=1)

        self.intra_attn_w = FeedFwd(embed_dim, embed_dim,
            'embedding', 'embedding', dropout_p=intra_dropout)

    def process_input(self, sentence, seqlen_dim):
        embedded = super().process_input(sentence, seqlen_dim)
        other_dim = seqlen_dim + "2"
        other_embedded = embedded.rename(seqlen_dim, other_dim)

        embedded_mask = (sentence != self.padding_idx).float()
        embedded_mask = embedded_mask * \
            embedded_mask.rename(seqlen_dim, other_dim)

        distances = (
            (ntorch.arange(embedded.size(seqlen_dim), names=seqlen_dim,
                device=embedded.values.device) -
             ntorch.arange(embedded.size(seqlen_dim), names=other_dim,
                device=embedded.values.device))
            .abs().clamp(max=self.max_distance))
        d_mat = self.distance_embed(distances)[{'embedding': 0}]

        f_embedded = self.intra_attn_w(embedded)
        f_embedded_other = f_embedded.rename(seqlen_dim, other_dim)

        log_alignments = (
            f_embedded.dot("embedding", f_embedded_other)
            + d_mat + (1 - embedded_mask) * (-1e3))

```

```python
        embedded_attns = log_alignments.softmax(
            other_dim).dot(other_dim, other_embedded)
        return ntorch.cat([embedded, embedded_attns], "embedding")
```

Listing 2: Exact Ensemble

```python
from namedtensor import ntorch
from namedtensor.nn import nn as nnn

class ExactEnsemble(nnn.Module):
    def __init__(self, models):
        super().__init__()
        self.models = nnn.ModuleList(models)

    def forward(self, hypothesis, premise):
        log_preds = ntorch.stack([
            model(hypothesis, premise) for model in self.models
        ], 'model')

        return log_preds.softmax('classes').mean('model').log()
```

Listing 3: VAE Ensemble

```python
import torch
from torch.distributions import Categorical, kl_divergence

from namedtensor import NamedTensor
from namedtensor import ntorch
from namedtensor.nn import nn as nnn

def logsumexp(named_tensor, dim_name):
    names = list(named_tensor.shape.keys())
    dim_num = names.index(dim_name)
    names.pop(dim_num)
    return NamedTensor(torch.logsumexp(named_tensor.values, dim_num, keepdim=False),
               names=names)

class VAEEnsemble(nnn.Module):
    def __init__(self, models, q, num_classes=4):
        super().__init__()
        self.models = nnn.ModuleList(models)
        self.q = q
        self.ce_loss = nnn.CrossEntropyLoss(reduction='none').spec('classes')
        self.num_classes = num_classes
        self.unif = Categorical(torch.ones(len(models),
                                           device=next(self.parameters()).device))

    def forward(self, hypothesis, premise, y=None):
        if self.training:
            assert(y is not None)
            weights = self.q(hypothesis, premise).softmax('classes')
            m = Categorical(weights.values)
            models = NamedTensor(m.sample(), names=('batch',))

            global_log_probs = ntorch.zeros(hypothesis.size('batch'),
                                            self.num_classes,
                                            names=('batch', 'classes'),
                                            device=hypothesis.values.device)

            for i in range(len(self.models)):
               is_model = models == i
               if is_model.sum().item() == 0:
                   continue
               model_batches = is_model.nonzero(names=('batch', 'extra'))[{'extra':
                   0}]
               model_hypothesis = hypothesis[{'batch': model_batches}]
               model_premise = premise[{'batch': model_batches}]

               log_probs = self.models[i](model_hypothesis, model_premise)

               global_log_probs[{'batch': model_batches}] = log_probs

            loss = -m.log_prob(models.values) * \
               self.ce_loss(global_log_probs, y).values + \
               kl_divergence(m, self.unif).sum()

            return loss.sum()

        else:
            log_preds = ntorch.stack([
               model(hypothesis, premise) for model in self.models
            ], 'model')

            unnorm_preds = logsumexp(log_preds, 'model')
            normalizing_factor = logsumexp(unnorm_preds, 'classes')
            return (unnorm_preds - normalizing_factor)
            #return ntorch.log(log_preds.softmax('classes').mean('model'))
```