THEORY REVIEW

Noise Contrastive Estimation

Introduction

Neural Language Probabilistic Language Model specifies the distribution for the target words given the a sequence of words h from context. w is typically the text word in sentence while h is the sequence of words precede w.
Given a context h, the NPLM defines the distribution for the word using the scoring function $s_\theta(w, h)$ that quantifies the **compatibility** between the context and candidate target word.The scores are converted to probabilities by soft normalization:

$$P_\theta^h(w) = \frac{e^{s_\theta(w,h)}}{\sum_w e^{s_\theta(w,h)}}$$

This equation is intractable because it requires normalizing over the entire vocabulary, and three solutions are presented:

1. tree structured vocabulary $\Rightarrow$ nontrivial
2. importance sampling $\Rightarrow$ unstable
3. NCE

Scalable Log-Bilinear Models

LBL, unlike traditional NPLMs, does not have a hidden layer and words by performing linear prediction in the word feature vector space. The model has two sets of words representations:

1. target words q_w for word w
2. context words r_w for word w

Given a sequence of context words $h = w_1, \ldots, w_n$, the model computes the **predicted representation** for the target word by

$$\hat{q} = \sum c_i \odot \gamma_w$$

> c_i weight vector for the context

The scoring function then computes

$$s_\theta(w,h) = \hat{q}_h^T q_w + b_w$$

This can be made simpler by

$$\hat{q} = \frac{1}{n}\sum^{n} r_{w_i}$$

As our main concern is learning word representations, we are free to move away from the paradigm of predicting the target from context and do the reverse. This is motivated by the distributed hypothesis:

> **Distributed hypothesis**
> words with similar meaning often occur in similar contexts.

Thus we'll be looking for word representations that capture the context distributions. Unfortunately, predicting n-word context requires modeling the joint distribution of n-words. This is considerably harder than modeling one of the words. We can make this trackable by assuming words in different context positions are **conditionally independent**, given current word w, the inverse LBL, or ivLBL:

$$P_\theta^w(h) = \prod P_{i,\theta}^w(w_i)$$

$$s_{i,\theta}(w_i, w) = (c_i \odot \gamma_w)^T q_{w_i} + b_{w_i}$$

$$s_{i,\theta}(w_i, w) = \gamma_w^T q_{w_i} + b_{w_i}$$

> $P_{i,\theta}^w(w_i)$ vector LBL models conditioned on the current word
> c_i optional, position-specific weight

Noise Contrastive Estimation

NCE is a method for fitting unnormalized methods, based on the **reduction of density estimation to probability binary classification**. The basic idea is to train a logistic regression classifier to discriminate between samples from the data distribution and samples from noise distribution. This is done based on the ration of $P_{x\sim model}$ and $P_{x\sim noise}$. NCE allows us to fit models that are not explicitly normalized, making the training time effectively independent of vocabulary size. As a result, we will be able to drop the normalizing factor during training

$$P_\theta^h(w) = \frac{e^{s_\theta(w,h)}}{\sum_w e^{s_\theta(w,h)}} \Rightarrow e^{s_\theta(w,h)}$$

Suppose we want to learn the distribution of words for some specific context h, or $P^h(w)$. To do that, we create an auxiliary binary classification problem, treating data as positive, and samples from a noise distribution $P_n(w)$ as negative. In the original research, authors use *global unigram distribution* of training data as noise distribution.

If we assume that noise samples are k times more frequent than data samples, the probability that given sample come from data

$$P^h(D=1|h) = \frac{P_{data}^h(w)}{P_{data}^h(w) + kP_n(w)}$$

We can estimate this probability using the model distribution in place of P_{data}^h, that is:

$$P^h(D=1|w,\theta) = \frac{P_\theta^h(w)}{P_\theta^h(w) + kP_n(w)} = \sigma(\Delta s_\theta(w,h)$$

> $P_\theta^h(w)$ model distribution parameterized by θ
> $\sigma(\cdot)$ logistic function
> $\Delta s_\theta(w,h) = s_\theta(w,h) - log(kP_n(w))$ the difference in the scores of word w under model distribution and scaled noise distribution. Note that the equation used $s_\theta(w,h)$ instead of $logP_\theta^h(w)$, ignoring the normalization term since we are dealing with unnormalized models. Remember that NCE model's objective encourages the model to be appropriately normalized and recovers a perfectly normalized model if the model class contains the data distribution.

The model is fit by maximizing the expected log posterior probability of correct label D:

$$J^h(\theta) = \mathbb{E}_{w\sim P_d^h}[logP^h(D=1|w,\theta)] + k\mathbb{E}_{w\sim P_n}logP^h(D=0|w,\theta)]$$

$$= \mathbb{E}_{w\sim P_d^h}\left[log\,\sigma\Big(\Delta s_\theta(w,h)\Big)\right] + k\mathbb{E}_{w\sim P_n}\left[log\Bigg(1-\sigma\Big(\Delta s_\theta(w,h)\Big)\Bigg)\right]$$

where the expectation over the noise distribution is approximated by sampling. Thus we estimate the contribution of word-context pair to the gradient of J by generating k noise samples and computing:

$$\frac{\partial}{\partial\theta}J^{h,w}(\theta) = (1-\sigma(\Delta s_\theta(w,h)))\frac{\partial}{\partial\theta}logP_\theta^h(w) - \sum^k[\sigma(\Delta s_\theta(x_i,h)\frac{\partial}{\partial\theta}logP_\theta^h(x_i)]$$

Note that the gradient involves a sum over k noise samples, instead of a sum over the entire vocabulary, making the training time of NCE **linear** in the number of noise samples and independent of vocabulary size. As we increase the number of noise samples k, this estimate approaches the likelihood gradient of the normalized model, allowing us to trade off computation cost against estimation efficiency

Evaluating Word Embeddings

We can answer the question $a : b \to c :?$ by finding the word d^* with the representation closest to $\vec{b} - \vec{a} + \vec{c}$ acording to cosine similarity: the word with the representation most similar to $\vec{b}, \vec{c}$ and dissimilar to $\vec{a}$

$$d^* = \arg\max \frac{\vec{b}^T \vec{x} - \vec{a}^T \vec{x} + \vec{c}^T \vec{x}}{||\vec{b} - \vec{a} + \vec{c}||}$$

Word2Vec

The Skip-Gram Model

The training objective of the Skip-Gram model is to find word representations that are useful for predicting the surrounding words. Given a sequence of training words $w_1, w_2, ..., w_T$, the objective of the Skip-Gram model is to maximize the expected log probability:

$$\frac{1}{T} \sum^{T} \sum_{j \in context} log\, p(w_t + j | w_t)$$

The basic SKip-Gram defines p using the softmax function:

$$p(w_O | w_I) = \frac{exp(v'_{w_o}{}^T v_{w_I})}{\sum_w^W exp(v'_w{}^T v_{w_I})}$$

> v_w, v'_w input, output vector representations of word w
> W number of words in the vocabulary

This formulation is practical because the cost of computing $\nabla log\, p(w_O | w_I)$ is proportional to W. Following are computationally efficient approximation of the full softmax

Hierarchical Softmax

Evaluating probability distribution only takes $log_2(W)$ nodes. Hierarchical softmax uses a binary tree representation of the output layer with W words as leaves. For each node, it explicitly represents the relative probabilities of its child nodes. These define a random walk that assigns probabilities to words. More precisely, each word w can be reached by an approximate path from the root of the tree. The Hierarchical softmax defines $p(w_O|w_I)$ as follows:

$$p(w|w_I) = \prod_{j=1}^{L(w)-1} \sigma(\mathbb{I}[n(w,j+1) = ch(n(w,j))] \cdot {v'_{n(w,j)}}^T v_{w_I})$$

> $n(w,j)$ j-th node on the path frm root to w
> $L(w)$ length of the path $\rightarrow$ $n(w,1) =$ root, $n(w, L(w)) = w$
> $ch(n)$ arbitray fixed child of inner node n

Hierarchical softmax has one representation for eacb word and one representation for every inner node n of the binary tree, unlike the standard Skip-Gram which assigns assigns two representations to each word w

Negative Sampling

NCE posits that a good model should be able to differentiate data from noise by mens of logistic regression, this is similar to hinge loss when training models by ranking the data above noise. While NCE can be shown to approximately maximize the log probability, Skip-Gram is only concerned with learning high quality vector representation. So we are free to simply NCE as long as the vector representation retain their quality. The NCE objective is defined by:

$$log\,\sigma({v'_{w_O}}^T v_{w_I}) + \sum^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[log\,\sigma(-{v'_{w_i}}^T v_{w_i}) \right]$$

Note how this is derived from the original equation:

$$\mathbb{E}_{w \sim P_d^h} \left[log\,\sigma\Big(\Delta s_\theta(w,h)\Big) \right] + k\mathbb{E}_{w \sim P_n} \left[log\Big(1 - \sigma\Big(\Delta s_\theta(w,h)\Big)\Big) \right]$$

$$\Rightarrow log\,\sigma\Big(\Delta s_\theta(w,h)\Big) + \sum^{k} \mathbb{E}_{w \sim P_n} \left[log\Big(1 - \sigma\Big(\Delta s_\theta(w,h)\Big)\Big) \right]$$

$$\Rightarrow log\,\sigma\left(s_\theta(w,h) - log\Big(kP_n(w)\Big)\right) + \sum^{k} \mathbb{E}_{w \sim P_n} \left[log\left(1 - \sigma\Big(s_\theta(w,h) - log\Big(kP_n(w)\Big)\Big)\right) \right]$$

$$\Rightarrow log\,\sigma\Bigg(s_\theta(w,h)\Bigg)+\sum^k\mathbb{E}_{w\sim P_n}\Bigg[log\,\sigma\Big(-s_\theta(w,h)\Big)\Bigg]$$

$$\Rightarrow log\,\sigma\Bigg(\gamma_w^T q_{w_i}+b_{w_i}\Bigg)+\sum^k\mathbb{E}_{w\sim P_n}\Bigg[log\,\sigma\Big(-\gamma_w^T q_{w_i}-b_{w_i}\Big)\Bigg]$$

$$\Rightarrow log\,\sigma\Bigg(\gamma_w^T q_{w_i}\Bigg)+\sum^k\mathbb{E}_{w\sim P_n}\Bigg[log\,\sigma\Big(-\gamma_w^T q_{w_i}\Big)\Bigg]$$

$$\Rightarrow log\,\sigma({v'_{w_O}}^T v_{w_I})+\sum^k\mathbb{E}_{w_i\sim P_n(w)}\Big[log\,\sigma(-{v'_{w_i}}^T v_{w_i})\Big]$$

This is used to replace every $logP(w_O|w_I)$ in the Skip-Gram objective. Thus the task is to distinguish the target word w_O from draws from the noie distribution $P_n(w)$ using logistic regression, where there are k negative samples for each data sample.

The major difference between Negative Sampling and NCE is that **NCE needs both samples and numerical probabilities of the noise distribution**, while **Negative Sampling uses only samples**. And while NCE approximately maximizes the log probability of the softmax, this property is not important for the application. The experiment from the original research indicated that $U(w)^{3/4}/Z$ outperformed significantly the unigram and uniform distributions

Subsampling of Frequent Words

In very large corpora, the most frequent can easily occur hundred of millions of times. Such words usually provide less information value. For example, the Skip-Gram benefits much less from observing the co-ocurrance of "France" and "the". The idea can be applied in the opposite direction:

> the vector representations of frequent words do not change significantly after training on several million examples

To counter the imbalance between the rare and frequent words, we can use a simple subsampling approach: **each words w_i in the training set is discarded with probability**

$$P(w_i)=1-\sqrt{\frac{t}{f(w_i)}}$$

> $f(w_i)$ frequency of word w_i
> t threshold, usually 10^{-5}

This aggressively subsamples frequent words (above the threshold) while preserving the rank of frequencies. It was found to accelerates learning and improves the accuracy of learned vectors of rare words.

Code

```
class Word2Vec(object):
    def __init__(self, options, session):
        self._options = options
        self._session = session
        self._word2id = {}
        self._id2word = []
        self.build_graph()
        self.eval_graph()
        self.save_vocab()
    def read_analogies(self):
        # skip for now
```

Forward

Softmax

A generalization of logistic regression to handle multiple classes.

$$P(y_i = k|x_i;\theta) = \frac{P(y_i = k, x; \theta)}{P(x;\theta)} = \frac{exp({\theta_k}^T x^i)}{\sum_k^K exp({\theta_k}^T x_i)}$$

Here we define

$$p(w_O|w_I) = \frac{p(w_O, w_I)}{p(w_I)} = \frac{exp({v^{'}_{w_o}}^T v_{w_I} + b_{w_O})}{\sum_w^W exp({v^{'}_w}^T v_{w_I} + b_w)}$$

$V = \{v_1, ..., v_W\} \in \mathbb{R}^{W \times emb}$ the weight vector,
$B = \{b_1, ..., b_W\} \in \mathbb{R}^{W \times 1}$ the bias vector

```
# Declare all variables we need.
# Softmax weight: [vocab_size, emb_dim].T
# Softmax bias: [vocab_size]
sm_w_t = tf.Variable(tf.zeros([opts.vocab_size, opts.emb_dim]), name = 'sw_w_t')
sm_b = tf.Variable([tf.zeros(opts.vocab_size)], name='sm_b')
```

Negative Sampling

Here we will replace $logP(w_O|w_I)$ with NCE objective $\Rightarrow$ to distinguish the target word w_O from noie distribution $P_n(w)$ using logistic regression.

$$logP(w_O|w_I) = log\left[\frac{exp({v'_{w_o}}^T v_{w_I} + b_{w_O})}{\sum_w^W exp({v'_w}^T v_{w_I} + b_w)}\right]$$

$$\Rightarrow \mathbb{E}_{w\sim P_d^h}\left[log\, \sigma\Big(\Delta s_\theta(w,h)\Big)\right] + k\mathbb{E}_{w\sim P_n}\left[log\Big(1-\sigma\Big(\Delta s_\theta(w,h)\Big)\Big)\right]$$

$$\Rightarrow log\, \sigma({v'_{w_O}}^T v_{w_I}) + \sum^k \mathbb{E}_{w_i\sim P_n(w)}\left[log\, \sigma(-{v'_{w_i}}^T v_{w_i})\right]$$

```
# Nodes to compute the nce loss w/ candidate sampling.
labels_matrix = tf.reshape(
        tf.cast(labels, dtype=tf.int64),
        [opts.batch_size, 1]
    )
# Embedding: [vocab_size, emb_dim]
emb = tf.Variable(
        tf.random_uniform(
            [opts.vocab_size, opts.emb_dim], -init_width, init_width
        ), name="emb"
    )
# Embeddings for examples: [batch_size, emb_dim]
example_emb = tf.nn.embedding_lookup(emb, examples) # examples are input
```

Sample $w_i \sim P_{noise}(w)$

```
sampled_ids, _, _ = (tf.nn.fixed_unigram_candidate_sampler(
        true_classes=labels_matrix,
        num_true=1,
        num_sampled=opts.num_samples,
        unique=True, # without replacement
        range_max=opts.vocab_size,
        distortion=0.75,
        unigrams=opts.vocab_counts.tolist()
    ))
```

Compute

$$logits = log\, \sigma({v'_{w_O}}^T v_{w_I} + b_{w_O})$$

$$logits \in \mathbb{R}^{batch,1}, v_{w_O} \in \mathbb{R}^{batch\times emb}, b_{w_O} \in \mathbb{R}^{batch\times 1}$$

```python
# Weights for labels: [batch_size, emb_dim]
true_w = tf.nn.embedding_lookup(sm_w_t, labels)
# Biases for labels: [batch_size, 1]
true_b = tf.nn.embedding_lookup(sm_b, labels)
# True logits: [batch_size, 1]
true_logits = tf.reduce_sum(tf.multiply(example_emb, true_w), 1) + true_b
```

Compute

$$logits = \sum^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[log\ \sigma(-v_{w_i}^{'}{}^{T} v_{w_i} + b_w) \right]$$

$$logits \in \mathbb{R}^{batch \times k}, V = \{v_1, ..., v_k \in \mathbb{R}^{emb}\} \in \mathbb{R}^{k \times emb}$$

```python
# labels are inputs
# Weights for sampled ids: [num_sampled, emb_dim]
sampled_w = tf.nn.embedding_lookup(sm_w_t, sampled_ids)
# Biases for sampled ids: [num_sampled, 1]
sampled_b = tf.nn.embedding_lookup(sm_b, sampled_ids)
# Sampled logits: [batch_size, num_sampled]
# We replicate sampled noise labels for all examples in the batch
# using the matmul.
sampled_b_vec = tf.reshape(sampled_b, [opts.num_samples])
sampled_logits = tf.matmul(example_emb,
                           sampled_w,
                           transpose_b=True) + sampled_b_vec
```

Tensorflow Operation

Candidate Sampling

> Training a large model with a full Softmax is slow, since all of the classes are evaluated for every training example. Candidate Sampling training algorithms can speed up the step times by only considering a small ramdomly-chosen subset of constrastive classes, *candidates*, for each batch of training examples

`tf.nn.fixed_unigram_candidate_samplier`

> Samples a set of classes using base distribution.
> This operation samples a tensor of sampled classes `sampled_candidates` from the range of integers `[0, range_max]`, set `unique=True` will draw `sampled_candidates` without replacement.
> The base distribution if read from a file or passed in as an in-memory array. This operation returns tensors `true_expected_count` and `sampled_expected_count` representing the number of times each of the

> target classes `true_classes` and the sampled classes `sampled_candidates` is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)`.

####`tf.nn.embedding_lookup`

> Looks up `ids` in a list of embedding tensors.
> This function is used to perform parallel lookups on the list of tensors in `params`.

```
    def forward(self, examples, labels):
        opts = self._options
        # embedding: [vocab_size, emb_dim]
        init_width = .5 / opts.emb_dim
        emb = tf.Variable(
            tf.random_uniform(
                [opts.vocab_size, opts.emb_dim], -init_width, init_width
            ), name = 'emb')
        self._emb = emb

        # Softmax weight: [vocab_size, emb_dim].T
        # Softmax bias
        sm_w_t = tf.Variable(tf.zeros([opts.vocab_size, opts.emb_dim]), name = 's
w_w_t')
        sm_b = tf.Variable([tf.zeros(opts.vocab_size)], name='sm_b')

        # Global step
        self.global_step = tf.Variable(0, name='global_step')

        # Nodes to compute the nce loss w/ candidate sampling
        label_matrix = tf.reshape(
            tf.casts(labels, dtype=tf.int64),
            [opts.batch_size, 1])

        # Negative Sampling
        sampled_ids, _ = (tf.nn.fixed_unigram_candidate_sampler(
            true_classes=labels_matrix,
            num_true=1,
            num_sampled = opts.num_samples
            unique=True,
            range_max=opts.vocab_size,
            distortion=.75
            unigrams=opts.vocab_counts.tolost()
        ))

        # Embeddings for examples: [batch_size, emb_dim]
        example_emb = tf.nn.embedding_lookup(emb, examples)

        # Weights for labels: [batch_size, emb_dim]
        # Biases for sampled ids: [batch_size, 1]
        true_w = tf.nn.embedding_lookup(sm_w_t, sampled_ids)
        true_b = tf.nn.embedding_loopup(sm_b, labels)

        # Weights for sampled ids: [num_sampled, emb_dim]
        # Biases for sampled ids: [num_sampled, 1]
        sampled_w = tf.nn.embedding_lookup(sm_w_t, sampled_ids)
        sampled_b = tf.nn.embedding_lookup(sm_b, sampled_ids)

        # True logits: [batch_size, 1]
        true_logits = tf.reduce_sum(tf.multiply(example_emb, true_w), 1)+ true_b

        # Sampled logits: [batch_size, num_sampled]
        sampled_b_vec = tf.reshape(sampled_b, [opts.num_samples])
        sampled_logits = tf.matmul(example_emb, sampled_w,
```

```
                                     transpose_b=True) + sampled_b_vec


        return true_logits, sampled_logits
```

NCE Loss

Cross Entropy between two probability distributions q and p over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set, if a coding scheme is used that is optimized for an "unnatural" probability distribution q, rather than the true probability distribution p.

$$H(p,q) = -\sum_{x} p(x) log\ q(x)$$

`tf.zeros_like (tensor, dtype, name, optimize)`

> given a single tensor, this operation returns a tensor of the same tyoe and shape.

```
def nce_loss(self, true_logits, sample_logits):

    # cross entropy (logits, labels)
    opts = self._options
    true_xent = tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.ones_like(true_logits), logits=true_logits)
    sampled_xent = tf.nn.sigmoid_cross_entropy_with_logits(
        labels=tf.zeros_like(sampled_logits), logits=sampled_logits)

    # nce loss is the sum of true and noise
    # contributions, averaged
    nce_loss_tensor = (tf.reduce_sum(true_xent) +
                       tf.reduce_sum(sampled_xent)) /
                       opts.babatch_size
    return nce_loss_tensor
```

optimize

```
def optimize(self, loss):
    # optimizer nodes
    # linear learning rate decay
    opts = self._options
    words_to_train = float(opts.words_per_epoch * opts.epoches_to_train)

    lr = opts.learning_date * tf.maximum(
        0.0001,
        1.0 - tf.cast(self._words, tf.float32) / words_to_train
    )

    self._lr = lr
    optimizer = tf.train.GradientDescentOptimizer(lr)
    train = optimizer.minimize(loss,
                               global_step=self.global_step,
                               gate_gradients=optimizer.GATE_NONE)
```

Build Graphs

$$d^* = \arg\max \frac{\vec{b}^T\vec{x} - \vec{a}^T\vec{x} + \vec{c}^T\vec{x}}{||\vec{b} - \vec{a} + \vec{c}||}$$

`tf.nn.l2_normalize (x, dim, epsilon, name)`

> normalizes along dimenion using L2 norm

`tf.nn.l2_normalize (x, dim, epsilon, name)`

> gather slices from parameters according to indices

```python
def build_eval_graph(self):
    analogy_a = tf.placeholder(dtype=tf.int32)
    analogy_b = tf.placeholder(dtype=tf.int32)
    analogy_c = tf.placeholder(dtype=tf.int32)

    # normalized embeddings
    nemb = tf.nn.l2_normalize(self._emb, 1)

    a_emb = tf.gather(nemb, analogy_a)
    b_emb = tf.gather(nemb, analogy_b)
    c_emb = tf.gather(nemb, analogy_c)

    target = c_emb + b_emb - a_emb

    # cosine distance
    dist = tf.matmul(target, nemb, transpose_b=True)
    _, pred_idx = tf.nn.top_k(dist, 4)

    nearby_word = tf.placeholder(dtype=tf.int32)
    nearby_emb = tf.gather(nemb, nearby_word)
    nearby_dist = tf.matmul(nearby_emb, nemb, transpose_b=True)
    nearby_val, nearby_idx = tf.nn.top_k(nearby_dist,
                                        min(1000,
                                            self._options.vocab_size)
                                        )
    self._analogy_a = analogy_a
    self._analogy_b = analogy_b
    self._analogy_c = analogy_c

    self._analogy_pred_idx = pred_idx
    self._nearby_word - nearby_word
    self._nearby_val = nearby_val
    self._nearby_idx = nearby_idx

def build_graph(self):
    opts = self._options
    (words,
        counts,
        words_per_epoch,
        self._epoch,
        self._words,
        examples,
        labels) = word2vec.skipgram_word2vec(
                                        filename=opts.train_data,
                                        batch_size=opts._batch_size
                                        window_size=opts.window_size,
                                        min_count=opts.min_count,
                                        subsample=opts.subsample)
    (opts.vocab_words,
        opts.vocab_counts,
        pts.words_per_epoch) = self._session.run([
                                                words,
                                                counts,
                                                words_per_epoch])
```

```
    opts.vocab_size = len(opts.vocal_words)
    self._examples = examples
    self._labels = labels
    self._id2word = opt.vocab_words

    for i,w in enumerate(self._id2word):
        self._word2id[w] = i

    true_logits, sampled_logits = self.forward(examples, labels)
    loss = self.nce_loss(true_logits, sampled_logits)
    tf.summary_scalar('NCE loss', loss)
    self._loss = loss
    self.optimize(loss)

    tf.global_varibales_initializer().run()
    self.saver = tf.train.Saver()

def train(self):
    opts = opt._options

    initial_epoch, initial_word = self._session.run([self._epoch, self._words])
    summary_op = tf.summary.merge_all()
    summary_writer = tf.summary.FileWriter(opts.save_path, self._session.graph)

    worders = []

    for _ in range(opts.concurrent_steps):
        t = threading.Thread(target=self._train_thread_body)
        t.start()
        worders.append(t)

    last_word, last_time, last_summary_time = initial_words, time.time(), 0
    last_checkpoint_time = 0

    while True:
        time.sleep(opts.statistics_interval) # report progress
        (epoch, step, loss, words, lr) = sess._session.run(
            [self._epoch, self.global_step, self._loss, self._words, self._lr])
        last_words, last_time, rate = words, now, (words - last_words) / (now - l
ast_time)

        sys.stdout.flush()

        if now - last_summary_time > opts.summary_interval:
            summary_str = self._session.run(summary_op)
            summary_writer.add_summary(summary_str, step)
            last_summary_time = now
        if now - last_checkpoint_time > opts.checkpoint_interval:
            self.saver.save(self._session,
                            os.path.join(opts.save_path, 'model.ckpt'),
                            global_step=step.astype(int))
            last_checkpoint_time = now

        if epoch != initial_epoch:
            breakk
```

```
    for t in workers:
        t.join()

    return epoch
```