CS 182/282A Deep Neural Networks **Fall 2022**

Lecture 16: Self-Supervision and Autoencoders (cont.)

Oct 18, 2022

Instructor: Anant Sahai *Scribes: Xingyi Yang, Sayan Seal*

16.1 Autoencoder Style Training

In Lecture 15, we learnt the structure of Autoencoder, which involves an Encoder, a Decoder, and a reconstruction loss for the training of parameters. Given a dataset of unlabeled data $\{\vec{x}_i\}$, we can train the Autoencoder, so that the Encoder transforms the input $\vec{x}_i$ into a new latent representation, denoted as $\vec{l}_i$, and the Decoder reconstructs the input $\vec{x}_i$ from the transformed representation $\vec{l}_i$. Then, the question is, how does an Autoencoder help practical machine learning tasks? When do we need an Autoencoder and how do we use it?

The key task is performed on the Encoder side. By training an Autoencoder, we get an Encoder which can transform a data point $\vec{x}_i$ into a transformed representation $\vec{l}_i$. Since we can recover much information of $\vec{x}_i$ from $\vec{l}_i$ through the Decoder, and $\vec{l}_i$ often is lower-dimensional, we assume that $\vec{l}_i$ keeps the essence of $\vec{x}_i$, while dropping some noisy information. The goal here is to learn some underlying pattern implicitly such that this will be helpful for other downstream tasks. The basic structure of an Autoencoder is shown in Figure 16.1. The part in the dotted box is important as it performs some encoding to appropriately distill the pattern. The remaining part serves as a *surrogate task*, which helps in the passage of gradients during training, but is trivial compared to the Encoder. In self-supervision, conceptually we create a surrogate task, where the target is a function of the input.

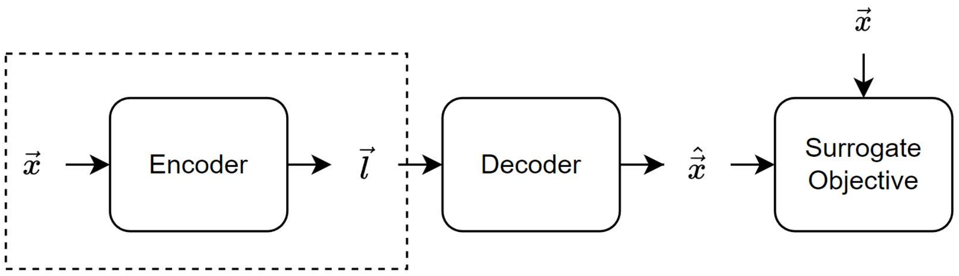

Figure 16.1: The basic form of an Autoencoder. We focus on the Encoder part in the following process.

Consider we are given a task with a set of labeled data $\{(x_i, y_i)\}$, and we are going to train a supervised model which provides a mapping from $\vec{x}_i$ to y_i. Normally, we train a deep neural network which maps input $\vec{x}_i$ to output $\hat{y}_i$ directly, and minimize the divergence between $\hat{y}_i$ and the ground-truth label y_i, as shown in Figure 16.2. However, if we are interested in classification, we do not optimize on the probability of misclassification, but on other forms of loss (surrogate loss) like squared error, one-hot encoding, cross-entropy loss or hinge loss. This pattern works for many tasks, but it may fail sometimes. The neural network may not train well if there are too little labeled data to support the training of a complex deep neural network of $\vec{x} \to y$.

Then comes the idea of using Autoencoder. Now that $\vec{l}_i$ provides a conciser representation of $\vec{x}_i$ itself, while

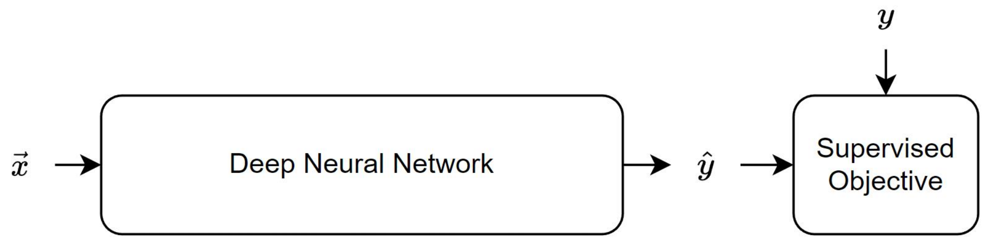

Figure 16.2: The common pattern for learning a supervised task using neural network.

keeping the most essential information, we can freeze the Encoder and learn a neural network which maps $\vec{l}_i$ to y_i, instead of $\vec{x}_i$. This way, the Encoder provides a refined representation of data, from which we can train a simpler neural network using gradient descent with less parameters. In this process, the Encoder takes a part of the representing work of neural networks, so that it is easier to train a neural network on top of it. The big picture of this idea in shown in Figure 16.3. Alternatively, the entire network can be learned in an end-to-end manner, with necessary fine-tuning of the encoder.

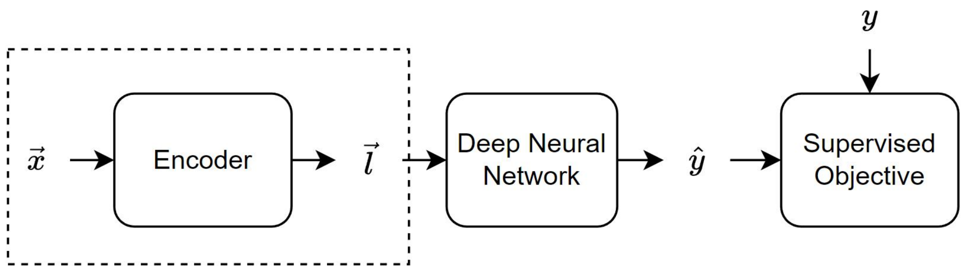

Figure 16.3: The big picture of using an Autoencoder to help a supervised task.

A big advantage of using Autoencoder is that we can use a larger set of unlabeled data for training of Autoencoder and leverage the knowledge for downstream tasks which come with a small set of labeled data. More specifically, consider a downstream task of mapping $\vec{x}_i$ to y_i, where there is a dataset $D = \{(x_i, y_i)\}$ of n pieces of data. And there is another unlabeled dataset $D_u = \{x_i\}$ of size N, where $N \gg n$. For the Autoencoder, we can use both D and D_u to train a better Encoder, because Autoencoder is an unsupervised model. This serves as a surrogate task. Then, for the downstream task, though we cannot use the dataset D_u directly, we manage to leverage the knowledge behind D_u by using the encoded representation $\vec{l}_i$ of Autoencoder. Figure 16.4 demonstrates this intuition.

This concept is very close to pre-trained models. However, the key difference between this type of approach and pre-trained models is that, here the surrogate task is not particularly interesting, but is designed to just learn the underlying pattern. In the case of pre-trained models, both the pre-training task to learn some pattern, and the actual task, which uses these patterns, are interesting. For example, pre-training a classifier on ImageNet, and then replacing the final classification layers to train on a smaller different dataset, such as classification of cancerous and non-cancerous tissues, are both interesting tasks.

16.2 Different Autoencoders based on Surrogate Tasks

There are multiple choices of surrogate tasks when we try to learn an Autoencoder. In terms of pre-process of input, there are three common tasks available: *vanilla autoencoding*, *denoising autoencoding*, and *masked*

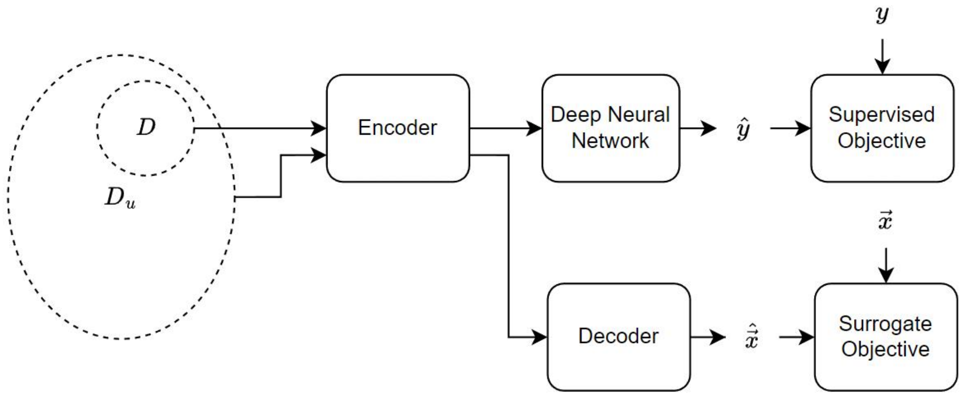

Figure 16.4: The demonstration of how an Autoencoder can help the downstream task by leveraging more training data in a self-supervised manner.

autoencoding. In the following parts, we denote $E(\cdot)$ as the Encoder function and $D(\cdot)$ as the Decoder function. The surrogate objective is denoted as $\mathcal{L}(\vec{x}, \hat{\vec{x}})$. The surrogate objective $\mathcal{L}$ is typically L2 distance between $\vec{x}$ and $\hat{\vec{x}}$.

16.2.1 Vanilla Autoencoder

The most basic idea is to directly use $\vec{x}_i$ as the input, and expect the Autoencoder to reconstruct the input $\vec{x}_i$ after decoding. Specifically, it minimizes $\mathcal{L}(\vec{x}_i, D(E(\vec{x}_i))$ directly.

It is the most naive approach to train an Autoencoder.

16.2.2 Denoising Autoencoder

We can also add some noise to the input $\vec{x}_i$ to train the model ability of denoising. That is, given an input vector $\vec{x}_i$, we add noise $\vec{n}$ to input $\vec{x}_i$ and give $\vec{x}_i + \vec{n}$ as input to the Encoder, where $\vec{n} \overset{\text{iid}}{\sim} N(0, \sigma^2)$. The surrogate target of the Decoder is still $\vec{x}_i$. Specifically, it minimizes $\mathcal{L}(\vec{x}_i, D(E(\vec{x}_i + \vec{n}))$. Then, to minimize the surrogate objective, the Autoencoder has to specify and diminish the noise part in $\vec{x}_i + \vec{n}$ when encoding. Thus the learnt Encoder is more robust to noise on the input. We can consider this approach as a kind of data augmentation. That is, by adding the noise term, more training data are generated to train the model. Moreover, this approach does not suffer from the problem of learning the identity mapping, while the Vanilla Autoencoder does. This is because, in this case, the input is different from the target, and the identity mapping is not the optimal solution.

16.2.3 Masked Autoencoder

Another approach of data augmentation is to mask some parts of each input vector $\vec{x}$. For example, given an input $\vec{x} = [x_1, x_2, x_3, x_4, x_5]^T$, we may randomly drop some of them, say, let the input to the Encoder be $\vec{x}' = [x_1, ?, x_3, x_4, ?]^T$. Specifically, it minimizes $\mathcal{L}(\vec{x}, D(E(\vec{x}'))$. The surrogate target of the Decoder is still $\vec{x}$. Then, to minimize the surrogate objective, the Autoencoder has to recover the missing information of the masked input $\vec{x}'$ according to the other parts. This approach can also be thought of as another kind of data augmentation, with the special symbol ?, but dealing with this symbol is a non-trivial task.

As there are several parameterization approaches introduced in Lecture 15, masked Autoencoder works differently with these approaches, as shown below.

1. The first parameterization approach is given by Equation 16.1

$$\begin{aligned} E(\vec{x}) &= A\vec{x} = \sum_{i=1}^{m} x[i]\vec{a}_i, \\ D(\vec{l}) &= B\vec{l}, \end{aligned} \tag{16.1}$$

   where the dimension of $\vec{x}$ is m and $A = [\vec{a}_1\, \vec{a}_2 \cdots \vec{a}_m]$. The architecture for this is shown in Figure 16.5. It involves two parameter matrices, A and B. For this approach, we simply change the masked part of $\vec{x}$ into value 0. This approach is like adding a dropout layer before the Encoder layer. So, just like dropout layer, we need to compensate for the loss of size of $\vec{l}$ when we drop some inputs. That is, if we randomly drop inputs by probability $1-p$, we need to multiply the transformation matrix A by $\frac{1}{p}$. The limitation to this approach is that we always use the same transformation matrix A and B, whatever part we mask. However, we may want the model to have different parameters when different part of the input is masked, because intuitively we may want to use different strategies to recover the information of the input when we know which part of the input is missing.

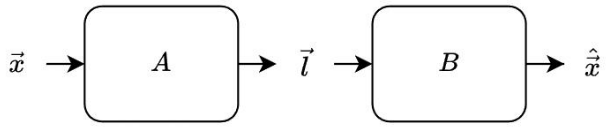

Figure 16.5: The demonstration of the first parameterization.

2. The second parameterization (Figure 16.6) fixes the limitation. It parameterizes the encoder as Equation 16.2

$$E(\vec{x}) = \begin{cases} (B^TB)^{-1}B^T\vec{x}, & \text{if } \dim(l) < \dim(x) \\ & \text{(reduction of parameters using least squares)} \\ B^T(BB^T)^{-1}\vec{x}, & \text{otherwise,} \\ & \text{(reduction of parameters using min norm)} \end{cases} \tag{16.2}$$

   where $B = [\vec{b}_1{}^T\, \vec{b}_2{}^T \cdots \vec{b}_m{}^T]^T$ and $B^T = [\vec{b}_1\, \vec{b}_2 \cdots \vec{b}_m]$, and $D(\vec{l}) = B\vec{l}$. The second case of Equation 16.2 yields Equation 16.3

$$D(E(\vec{x})) = BB^T(BB^T)^{-1} = I, \tag{16.3}$$

   which is not desirable, and hence this case is not used. This parameterization involves one parameter matrix, B. When masking the input for it, we drop the corresponding rows of B because these parts are not used by the input. Specifically, if there is an input vector $\vec{x} = [x_1, x_2, x_3, x_4, x_5]$ and we mask it as $\vec{x}' = [x_1, 0, x_3, x_4, 0]$, then we can remove the second and the last rows in B and get $\tilde{B}$. And then we have Equation 16.4.

$$E(\vec{x}) = (\tilde{B}^T\tilde{B})^{-1}B^T\vec{x}. \tag{16.4}$$

   The advantage of this approach is that when different part of $\vec{x}$ is masked, the model changes accordingly, because $\tilde{B}$ is changed accordingly. For each possibility of masking, the resulting parameter $\tilde{B}$ is different, which intuitively provides different strategies for recovering information from the masked input. Also, in the decoder, $D(\vec{l}) = B\vec{l}$, where $\vec{l}$ is being learned, and we have an actual target $\vec{x}$. So, we get gradients during reconstruction, and not from the encoder, to set the rows of B which were previously dropped.

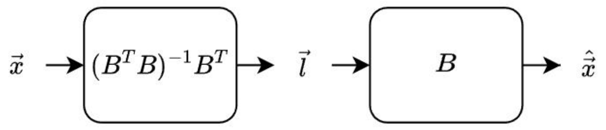

Figure 16.6: The demonstration of the second parameterization.

3. The third parameterization approach considers the encoder as an RNN style network with weight sharing (Figure 16.7). The gradient descent step is given by Equation 16.5.

$$\begin{aligned} \vec{l}_0 &= \vec{0}, \\ \vec{l}_{t+1} &= \vec{l}_t + \eta B^T(\vec{x} - B\vec{l}_t). \end{aligned} \tag{16.5}$$

   Running this block for d steps signifies gradient descent with early stopping, which can learn something meaningful. This is because, if B is such that most of its singular values are in the direction of the underlying pattern, and the other singular values are small, taking a few steps of gradient descent along that direction will project onto the pattern, with very little energy going in other directions. This is effectively denoising despite projecting onto a larger space. Hence, it is expected that deeper architectures for Autoencoders can learn interesting inductive biases.

   In this approach, we first compute $\vec{x} - B\vec{l}_t$. But some of the inputs are masked, and hence we need to first figure out what $? - *$ means ($*$ denotes some real number or vector). Two natural choices for this are either to use ? or 0, which give the same final result. This is because, in the next step, when we multiply ηB^T to $\vec{x} - B\vec{l}_t$, the multiplication of some quantity with 0 will yield 0, as will the multiplication with ? following the strategy used in the second parameterization. Hence $? - *$ is clamped as 0, which makes the residual 0. Treating ? directly as 0 is not a valid choice as $? - *$ will then be $-*$, denoting a residual. In this case, the model will try to update the parameters to get rid of the residual, which does not make sense since nothing should be present at the masked locations to drive the update at the encoder. Decoder is designed to deal with this issue, while the main task of the encoder is to find a good latent representation.

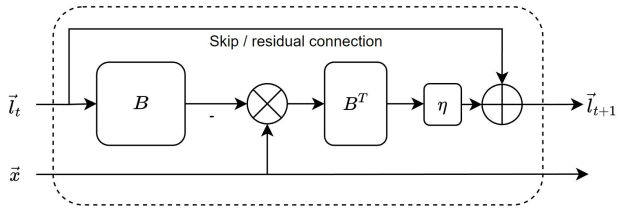

Figure 16.7: The demonstration of the parameterization of RNN style network with weight sharing.

The issue with the parameterizations discussed above is that the rows of the matrices corresponding to the masked entries in the encoder will have no update.

For each input, some probability metric is used to determine the position of the masks. The final goal here is to learn the underlying pattern, so that the model can learn to complete things. However, due to this masking, some of the important features may not get recovered, if those masked features cannot be extracted from the other unmasked features. We can expect that the different masks help in learning the implicit patterns on average.

Binary flags, instead of ?, can also be passed to the network to denote masking. The implementation depends on whether we want the model to learn how to use the binary flag or do some pre-processing with the flag.

Masked autoencoders can be seen as another way of learning low dimensional structure, apart from denoising autoencoders. The models tend to generalize well. For example, the task of adding noise in language models is difficult, and hence masking can be used. Even though blanks can be in a critical position, the model can learn something about the pattern so that some unlikely predictions can be discarded.

Even though with low dimensional latent space, there is no fear of learning the identity mapping, sometimes we want to have the freedom of learning larger l. In deep learning, we have seen many times that if we allow more parameters in the model, we tend to have better training, and achieve better performance. If l is big, for parameterization one, we have more rows of A with more parameters for initialization, and hence more room for a lucky guess to get a good row of A. Moreover, it is possible to have a low dimensional structure embedded in a high dimensional space that is still being purified in some way. The pure invertibility perspective of a matrix is not that useful since in matrix inversion, no information is lost. But the singular value perspective of a matrix helps to see the purification process in some way. It does not allow everything to go in the same way, but might shrink things in some direction and expand things in other directions.

16.3 Aside: Beam Search

A *language model* is a sequential model with access to some kind of memory. For RNN or LSTM, memory of the past is constrained to be the state, but attention mechanism is also used for language models. The general picture of a language model is shown in Figure 16.8.

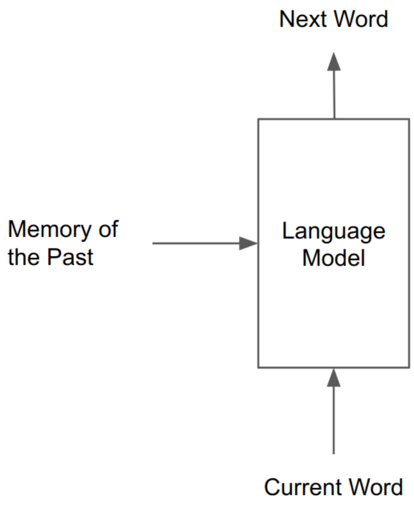

Figure 16.8: General Picture of a Language Model.

An example of self-supervised training of a language model is shown in Figure 16.9.

Words (represented using some vector embedding) can be considered as a discrete set of objects, and hence this self-supervision task is kind of a generalized classification problem. The output of the model at each step is not a single object, but a set of learned scores or probabilities corresponding to the different words. Self-supervised training over millions of sentences can make a model learn to put high scores on words that appear frequently, and lower the scores of words that do not appear. For example, if we have a sentence *"The man ate the ___"*, the probability of the next word being *carrot* should be much more compared to the word *of*.

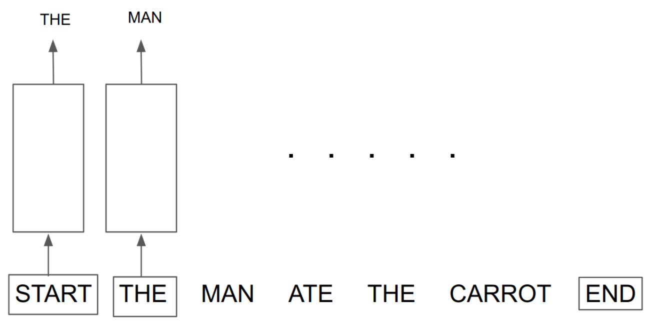

Figure 16.9: Self-supervised training of a Language Model where at each step, the model is predicting the next word conditioned on the previous words.

We can use the trained model on some task, for example, to generate a sentence. Filling in just one blank using the model is not a difficult task, as the model has already been trained on that. So, we can take the word corresponding to the highest output probability, or use random sampling. For the sentence *"The man ate the ___"*, if we want a single output, we can just get the output corresponding to the highest probability (suppose *carrot* in this case). But for a variable length output, different outcomes such as *"The man ate the banana."*, and *"The man ate the banana with the fork."* are possible. This task of completing a sentence is challenging. Exhaustively searching the tree of all $O(M^T)$ possible sequences, where M is the size of the vocabulary, and T is the maximum length of a sequence, to determine the true most likely sequence can be intractable.

The naive strategy is to pick the highest probability for the next word in a greedy manner, then add the word to the resulting sequence, and repeat the process using the updated sequence. But this may lead to a dead end, such as loops. Backtracking approach is not commonly used for language models, since it is difficult to interpret which situation is bad, so that the model can return to the last stable state. The data is always collected from a dataset containing good examples (i.e., from the pattern), and the model learns on that dataset. We do not have data that does not come from the pattern, or comes from almost the pattern.

Hence, a better strategy is not to commit to one particular thing, but instead keep a bag of current best possibilities in order to generate the most likely next sequence based on high likelihood. This is the main idea behind *Beam Search*. At each step, keep k best-so-far continuations based on the output probabilities. For each of the k continuations, predict the next output, which will generate k^2 possibilities, pick the k best ones, and repeat the process. The pseudocode of this strategy (taken from question 5 of hw6) is given in Algorithm 1:

Algorithm 1 Beam Search

```
for each time step $t$ do
    for each hypothesis $y_{1:t-1,i}$ that are being tracked do
        find the top $k$ tokens $y_{t,i,1}, \cdots, y_{t,i,k}$
    end for
    sort the resulting $k^2$ length $t$ sequences based on the total log-probability
    store the top $k$ sequences
    advance each hypothesis to time step t + 1
end for
```

A simple visualization of this strategy as a tree search problem is shown in Figure 16.10, where $k = 2$. The leftmost node can be considered as the START token, or one of the results so far. At each next step, based on the probabilities, 2 (red nodes) out of the 4 most probable nodes (red and blue nodes) are chosen for the most likely paths. For a more concrete example, please refer to question 5 of hw6.

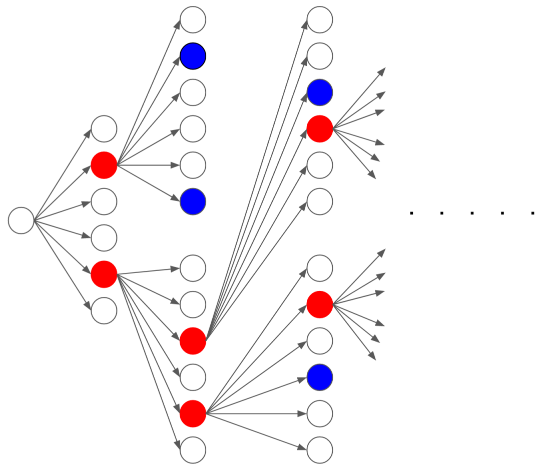

Figure 16.10: Simple visualization of the Beam Search strategy as a tree search problem ($k = 2$).

However, multiplication of the probabilities along the predicted sequence might create biases towards very short sentences. If appropriate normalization is used to tackle the issue, biases might be created for very long sentences. There are numerous tricks involved to get this strategy working properly.

16.4 What we wish this lecture also had to make things clearer?

1. It was briefly mentioned in the lecture that in the context of language models, adding noise is a difficult task, for which masking is used. Even though we believe that the topic will be covered in more details in the subsequent lectures, it would have been really helpful if some specific examples of such models using masking techniques were provided.

2. For all the parameterizations, the bottleneck case where $\dim(l) < \dim(x)$ does not have the issue of learning the trivial identity map. This issue might arise if $\dim(l) \geq \dim(x)$, and under this condition, learning the identity map is always the case for parameterization 2. However, it was mentioned in the lecture that sometimes, having larger latent space can be beneficial for learning better models (for the other parameterization cases). The manner in which the different relative latent space dimensions may be useful (or harmful), can be summarized for clear understanding. Also, in the case where $\dim(l) \geq \dim(x)$, the importance of dense and sparse representations can be mentioned to provide a complete picture. More figures might be included for explaining the different autoencoder ideas.

3. The overview of Beam Search was briefly covered in the lecture. But, some of the important concepts such as stopping criterion, or normalization techniques to deal with different lengths of the sequences, or multiplication of probabilities along the sequence were not covered. It would have been helpful to get some information on these concepts, along with a concrete example.