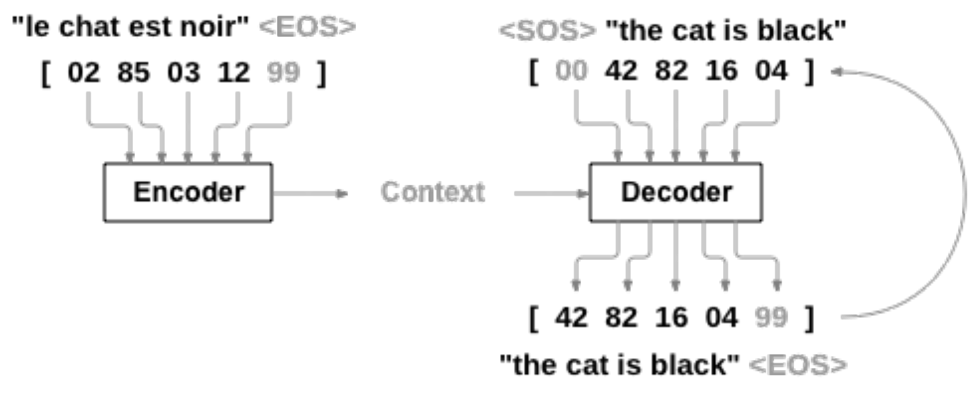
Lab4 : Conditional Sequence-to-Sequence VAE

contributed by < `tl32rodan` >

Introduction

Sequence-to-Sequence Network

- In natural language processing (NLP) (https://en.wikipedia.org/wiki/Natural_language_processing), translation between languages can be achieved by the simple but powerful idea of the sequence to sequence network, in which two **recurrent neural networks (RNNs)** work together to transform one sequence to another. An encoder network condenses an input sequence into a vector, and a decoder network unfolds that vector into a new sequence.



  - Reference: https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html (https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html)

- In this lab, we use conditional variational autoencoder (VAE) to implement sequence to sequence model, and do ***English tense conversion*** and ***text generation***
- For example, when we input the input word "access" with the tense (the **condition**) "simple present" to the encoder, it will generate a latent vector z. Then, we take z with the tense "present progressive" as the input for the decoder and we expect that the output word should be "accessing"

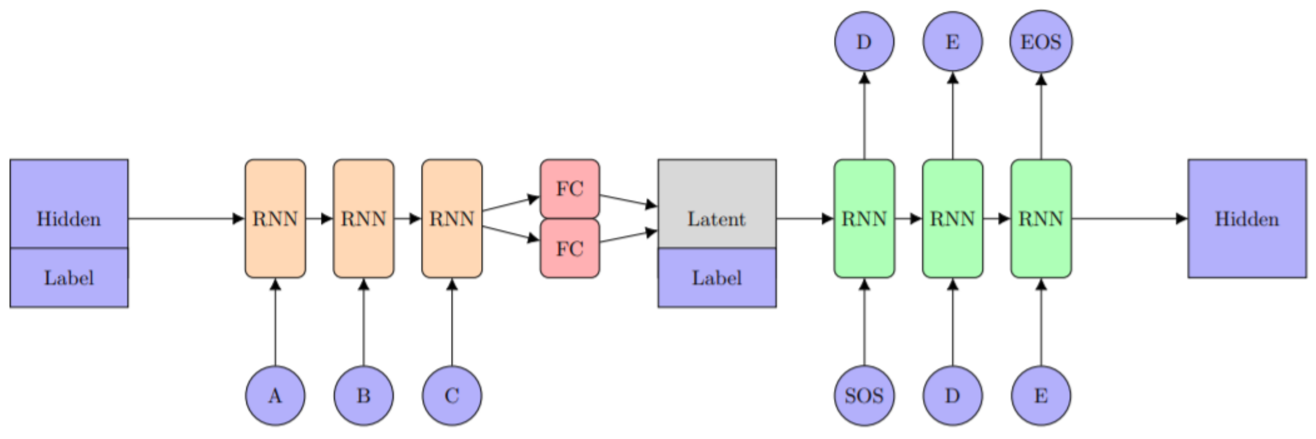


Figure 2: The illustration of sequence-to-sequence VAE architecture.

- Further Reading: Understanding Variational Autoencoders (VAEs) (https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73)

Derivation of CVAE

- To dirivate CVAE, we start from the solution
$$\theta^* = \arg\max_\theta p(X|c;\theta)$$
- The chain rule:
$$\log p(X|c;\theta) = \log p(X,Z|c;\theta) - \log p(Z|X,c;\theta)$$
- Integral both side with arbitrary distrubution $q(Z|c)$ over Z
$$\begin{aligned}&\int q(Z|c)\log p(X|c;\theta)dZ \\ &= \int q(Z|c)\log p(X,Z|c;\theta)dZ - \int q(Z|c)\log p(Z|X,c;\theta)dZ \\ &= \underbrace{\int q(Z|c)\log p(X,Z|c;\theta)dZ - \int q(Z|c)\log q(Z|c)dZ}_{\mathcal{L}(X|c,q,\theta)} \\ &+ \underbrace{\int q(Z|c)\log q(Z|c)dZ - \int q(Z)\log p(Z|X,c;\theta)dZ}_{KL(q(Z|c)||\log p(Z|X,c;\theta))}\end{aligned}$$
- Since the KL divergence is non-negative, $KL(q||p) \geq 0$, it follows that $\log p(X|c;\theta) \geq L(X|c,q,\theta)$
  - This implies that $L(X|c,q,\theta)$ is a lower bound of $\log p(X|c;\theta)$
- If we choose $q(Z|c)$ deliberately $q(Z|c) = p(Z|X,c;\theta^{old})$ and apply EM algorithm (https://hackmd.io/mzYvTR48R6eS2VDsBe3LXg) we have:
$$\begin{aligned}\log p(X|c;\theta^{new}) &= \underbrace{\int q(Z|c)\log p(X,Z|c;\theta^{new})dZ - \int q(Z|c)\log q(Z|c)dZ}_{(1)} \\ &+ \underbrace{\int q(Z|c)\frac{\log q(Z|c)}{\log p(Z|X,c;\theta^{new})}dZ}_{\geq 0} \\ &\geq \underbrace{\int q(Z|c)\log p(X,Z|c;\theta^{old})dZ - \int q(Z|c)\log q(Z|c)dZ}_{(1)'} \\ &+ \underbrace{\int q(Z|c)\frac{\log q(Z|c)}{\log p(Z|X,c;\theta^{old})}dZ}_{=0} = \log p(X|c,\theta^{old})\end{aligned}$$
  - $(1) \geq (1)'$ is from M step
- So that the increase in $\log p(X|c;\theta)$ is at least as much as $\mathcal{L}(X|c,q,\theta)$. That is, instead of **maximizing** $p(X|c;\theta)$ **directly**, we can **maximize** $\mathcal{L}$ as another option

- A rearrangement gives

$$\log p(X|c;\theta) - KL(q(Z|c) \| \log p(Z|X,c;\theta)) = \mathcal{L}(X|c,q,\theta)$$

- We can model $q(Z|c)$ using a neural network $q(Z|X,c;\theta^{'})$

$$\log p(X|c;\theta) - KL(q(Z|X,c;\theta^{'}) \| \log p(Z|X,c;\theta)) = \mathcal{L}(X|c,q,\theta)$$

- The right hand side can be spell out as

$$\begin{aligned}\mathcal{L}(X|c,q,\theta) = {} & E_{Z\sim q(Z|X,c;\theta')} \log p(X|Z,c;\theta) \\ & + E_{Z\sim q(Z|X,c;\theta')} \log p(Z|c) - E_{Z\sim q(Z|X,c;\theta')} q(Z|X,c;\theta^{'}) \\ & = E_{Z\sim q(Z|X,c;\theta')} \log p(X|Z,c;\theta) \\ & - KL(q(Z|X,c;\theta^{'}) \| p(Z|c))\end{aligned}$$

- Now, instead of directly maximizing the intractable $p(X;\theta)$, we attempt to maximize $\mathcal{L}(X|c,q,\theta)$, which amounts to maximizing $\underbrace{E_{Z\sim q(Z|X,c;\theta')} \log p(X|Z,c;\theta)}_{\text{Reconstruction}} - \underbrace{KL(q(Z|X,c;\theta^{'}) \| p(Z|c))}_{Regularization}$

Implementation Details

Encoder & Decoder

- VAE (or CVAE) usually composes of one encoder and one decoder, with a latent variable , which is generated by a Gaussian, between them
- Using embeddings to represent words or characters is common in NLP. To represent all alphabets and SOS, EOS token, we use `nn.Embedding` (https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html) to stores embeddings of them.
- We use `nn.LSTM` as RNNs of encoder and decoder
- Encoder:

```
#Encoder
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)

    def forward(self, input, hidden):

        output = self.embedding(input).view(1, 1, -1)
        output, hidden = self.lstm(output, hidden)

        return output, hidden
```

- Decoder:

```
#Decoder
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):

        output = self.embedding(input).view(1, 1, -1)
        output, hidden = self.lstm(output, hidden)
        output = self.out(output[0])
        return output, hidden
```

- Note that `hidden` of `nn.LSTM` should be a tuple because LSTM has both ***hidden state*** and ***memory state***

VAE & CVAE

- With encoder and decoder, we can constructure our VAE and CVAE

VAE

- First, we design our VAE

```
# VAE model
class VAE(nn.Module):
    def __init__(self,input_size, hidden_size, output_size, teache
        super(VAE, self).__init__()

        self.encoder   = EncoderRNN(input_size, hidden_size)
        self.fc_mu     = nn.Linear(hidden_size, hidden_size)
        self.fc_logvar = nn.Linear(hidden_size, hidden_size)
        self.decoder   = DecoderRNN(hidden_size, output_size)
        self.teacher_forcing_ratio = teacher_forcing_ratio

        ...
    def forward(self, x, hidden, use_teacher_forcing = False, targe
        # Encode
        mu, logvar = self.encode(x, hidden)

        # Reparameterize
        hidden = self.reparameterize(mu, logvar)

        result = self.decode(hidden, use_teacher_forcing, target_te

        return result, mu, logvar
```

- Forwarding is simple, just concate encoder & decoder with intermediate latent variable. Note that the latent is generated by reparameterization trick, which will be intruduce later.

Reparameterization

- Reparameterization trick is an important concern if we want to train VAE end-to-end
- Recall that the loss function of VAE
$$\underbrace{E_{z\sim q(Z|X;\theta')}\log p(X|Z;\theta)}_{\text{Reconstruction}} - \underbrace{KL(q(Z|X;\theta')\|p(Z|X;\theta))}_{\text{Regularization}}$$
where $q(Z|X;\theta')$ is considered as encoder and $p(X|Z;\theta)$ as decoder
- When we do regularization, $q(Z|X;\theta')$ is trained to approximate a Gaussian, that is $p(Z|X;\theta)$. But we can just sample from q to evaluate the KL divergence, so we're not able to transfer the gradient from decoder to encoder, that is, we can't train the model end-to-end
- We can make q to **generate it's mean and variance**, and combine with $N(0, I)$, so we can train VAE end-to-end

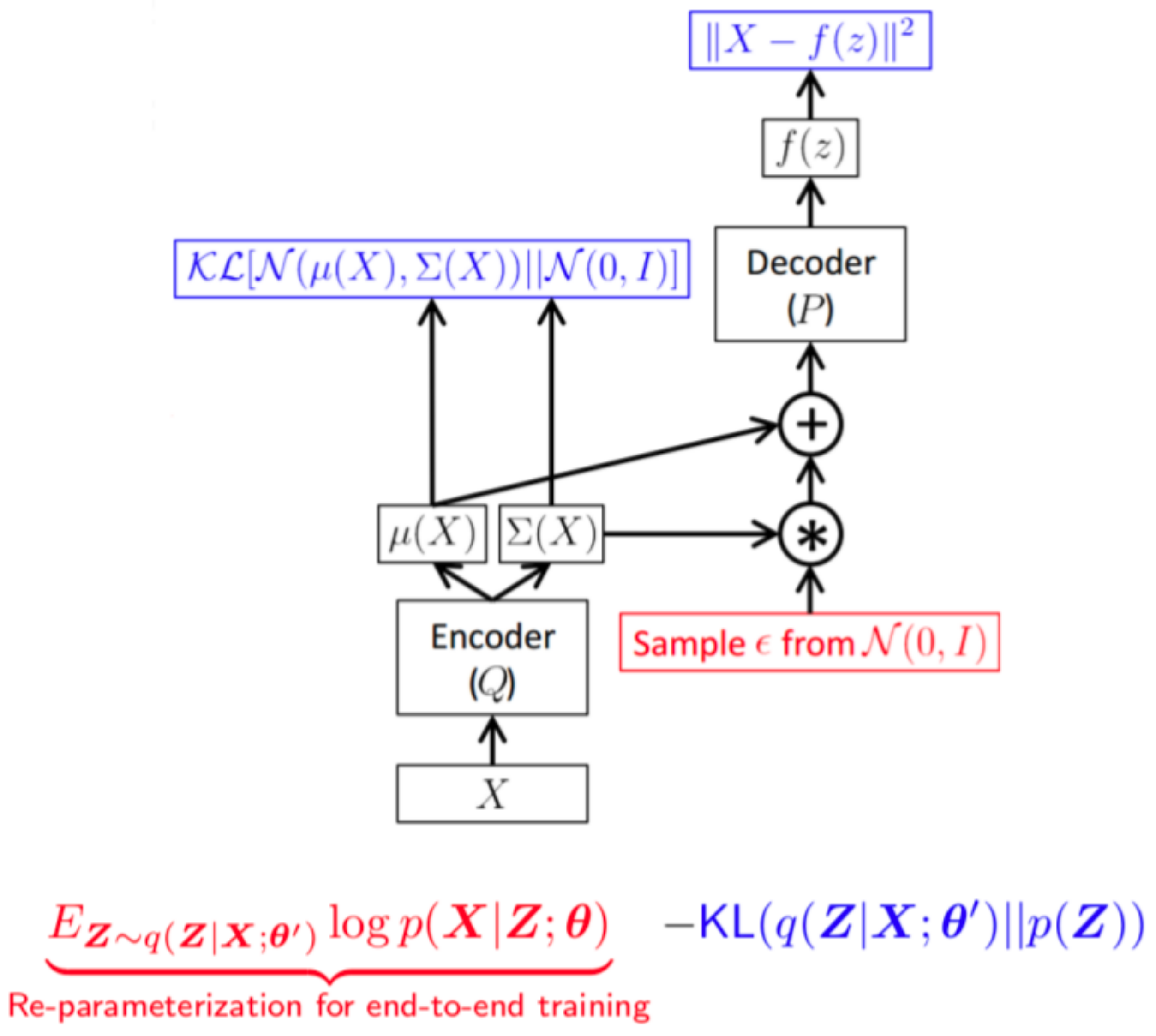

$\|X - f(z)\|^2$
$f(z)$
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X))\|\mathcal{N}(0, I)]$
Decoder (P)
+
$\mu(X)$
$\Sigma(X)$
*
Encoder (Q)
Sample ϵ from $\mathcal{N}(0, I)$
X
$E_{\boldsymbol{Z}\sim q(\boldsymbol{Z}|\boldsymbol{X};\boldsymbol{\theta}')}\log p(\boldsymbol{X}|\boldsymbol{Z};\boldsymbol{\theta})$ $-\mathrm{KL}(q(\boldsymbol{Z}|\boldsymbol{X};\boldsymbol{\theta}')\|p(\boldsymbol{Z}))$
Re-parameterization for end-to-end training

Figure 3: The illustration of reparameterization trick.

> Note: In practice, the generated variance should be **log variance** instead of variance, so that we can force the network to have the output range of the natural numbers rather than just positive values. This allows for smoother representations for the latent space.

- Implementation

```
def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)

        # eps is generated from N(0,I)
        eps = torch.randn_like(std)

        return mu + eps*std
```

CVAE

- With VAE, we can just convert English tense from one to another
- To cover all tense convertion, we can add condition to our VAE

```
# Conditional VAE model
class CondVAE(nn.Module):
    def __init__(self,input_size, hidden_size, output_size, teache
        super(CondVAE, self).__init__()
        self.hidden_size = hidden_size
        # Condition matters
        self.condition_embedding_size = 8
        self.num_conditions = 4
        ...
        self.encoder_condition_embedding = nn.Embedding(self.num_c
        self.decoder_condition_embedding = nn.Embedding(self.num_c
        ...
```

- In CVAE, we add condition embedding to implement it. Note that we should add these embedding to hidden units before encoding and decoding
- Furthermore, we can generate text using gaussian noise and condition

KL Annealing

- Problem: KL vanishing
    - When we train VAE, We can quantify the degree to which our model learns global features by looking at the variational lower bound objective
      $$\mathcal{L}(X,q,\theta) = \underbrace{E_{z\sim q(Z|X;\theta')} \log p(X|Z;\theta)}_{\text{Reconstruction}} - \underbrace{KL(q(Z|X;\theta')||p(Z|X;\theta))}_{\text{Regularization}}$$
    - To maximize the variational lower bound objective, the reconstruction term should be maximize, while the KL divergence should be minimize i.e. approach to `0`
    - A model that encodes useful information in the latent variable z will have a nonzero KL divergence term and a relatively small cross entropy term . Straightforward implementations of VAE will fail to learn this behavior: except in vanishingly rare cases, most training runs with most hyperparameters yield models that consistently set $q(Z|X)$ equal to the prior $p(z)$, bringing the KL divergence term of the cost function to zero
        - Reference: Generating Sentences from a Continuous Space (https://arxiv.org/abs/1511.06349)
- One solution: KL Annealing
    - Cyclical Annealing Schedule: A Simple Approach to Mitigating KL Vanishing (https://arxiv.org/abs/1903.10145) proposed a method call **KL annealing**, tried to solve the KL vanishing problem

- The annealing techniques are straight forward: add weight to KL term, and increase the weight as time goes by, So that the VAE will have enough time to encode information to z more with little KL penalty

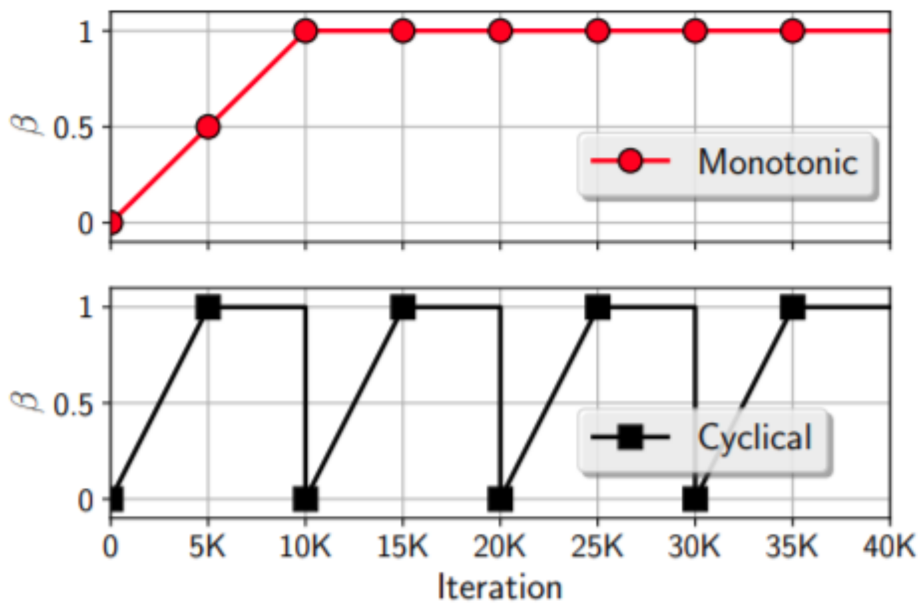


Figure 2: Comparison between (a) traditional monotonic and (b) proposed cyclical annealing schedules.In this figure, $M = 4$ cycles are illustrated, $R = 0.5$ is used for increasing within each cycle.

- Code implementation

```
## Use KL annealing
def KL_annealing(current_iter, policy = 'mono', mono_reach_max
                cycl_reach_max = 50000, cycl_period = 2000):
    if policy == 'mono':
        beta = 1 if current_iter >= mono_reach_max else (currer
    elif policy == 'cyclical':
        beta = 1 if current_iter%cycl_period >= cycl_reach_max
    else:
        raise ValueError

    return beta
```

Dataloader

- Since the data is text, the dataloader is simple in this lab

```
def load_data(filename):
    vocab = []
    with open(filename) as f:
        for line in iter(f):
            line = line[:-1].split(' ')
            vocab.append(line)
        f.close()

    return vocab
```

Text Generation

- As mentioned above, we can generate text with Gaussian and condition

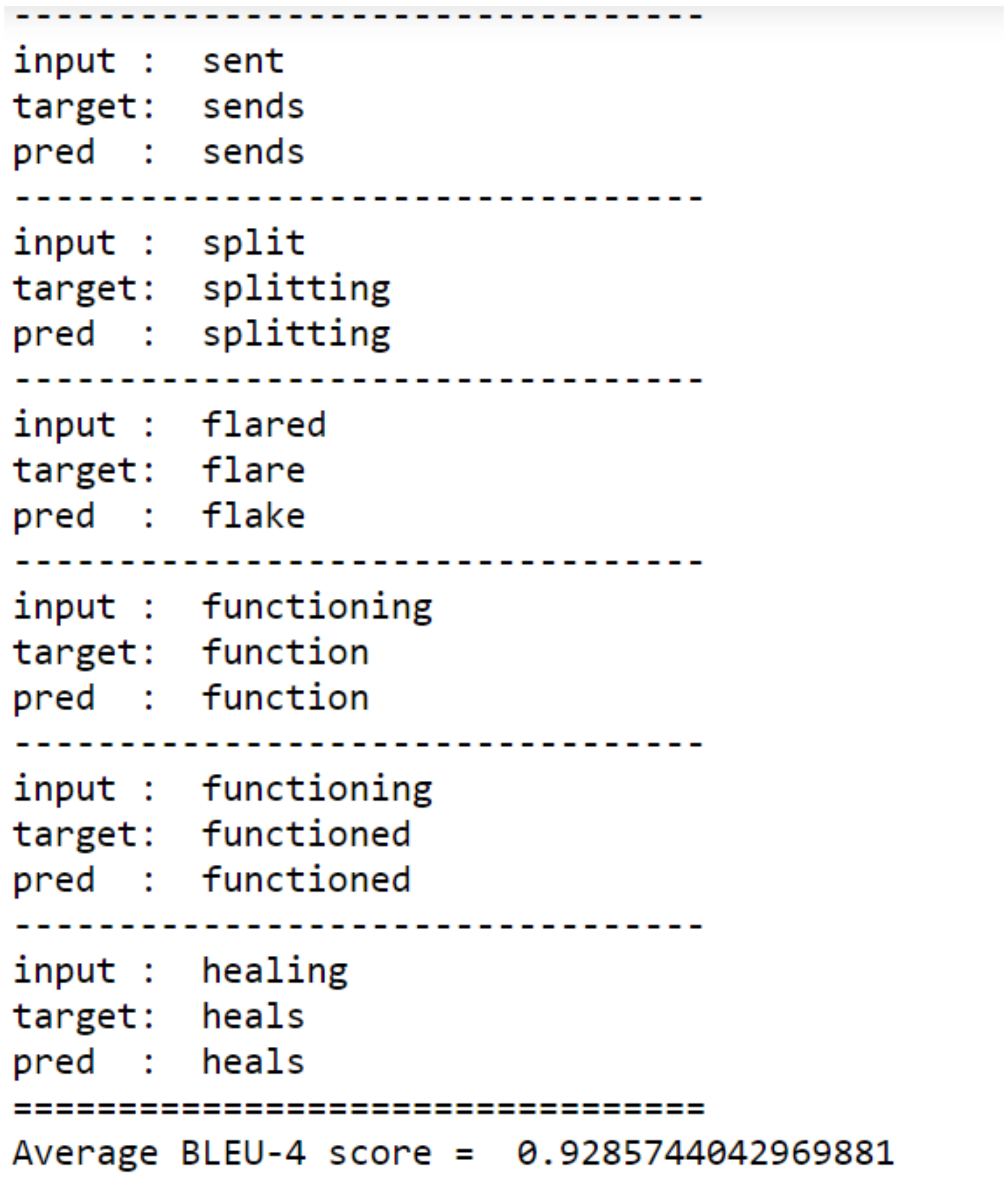
```python
def cal_gaussian(model, latent_size=32):
    result = []
    model.eval()
    for i in range(100):
        pred_tuple = []
        # Random latent from Gaussian
        latent = torch.randn(1, 1, latent_size, device=model.device

        # Run decoder with the latent, generate texts with 4 tense
        for cond in range(4):
            pred_seq = model.decode(model.fc_extend_latent(latent),
            pred_seq = str_from_tensor(pred_seq)
            # Remove EOS
            pred_tuple.append(pred_seq[:-1])

        result.append(pred_tuple)
    # Calculate score
    score = Gaussian_score(result)

    return score
```

Hyperparameters

- Learning Rate: `0.05`
  - Learning Rate Scheduler: `nn.optim.StepLR(gamma=0.8)`
- Teacher Forcing Ratio: `0.7`
- Number of Epochs: `5000`
- Hidden Size: `256`
- Latent Size: `32`

Results

Tense Conversion



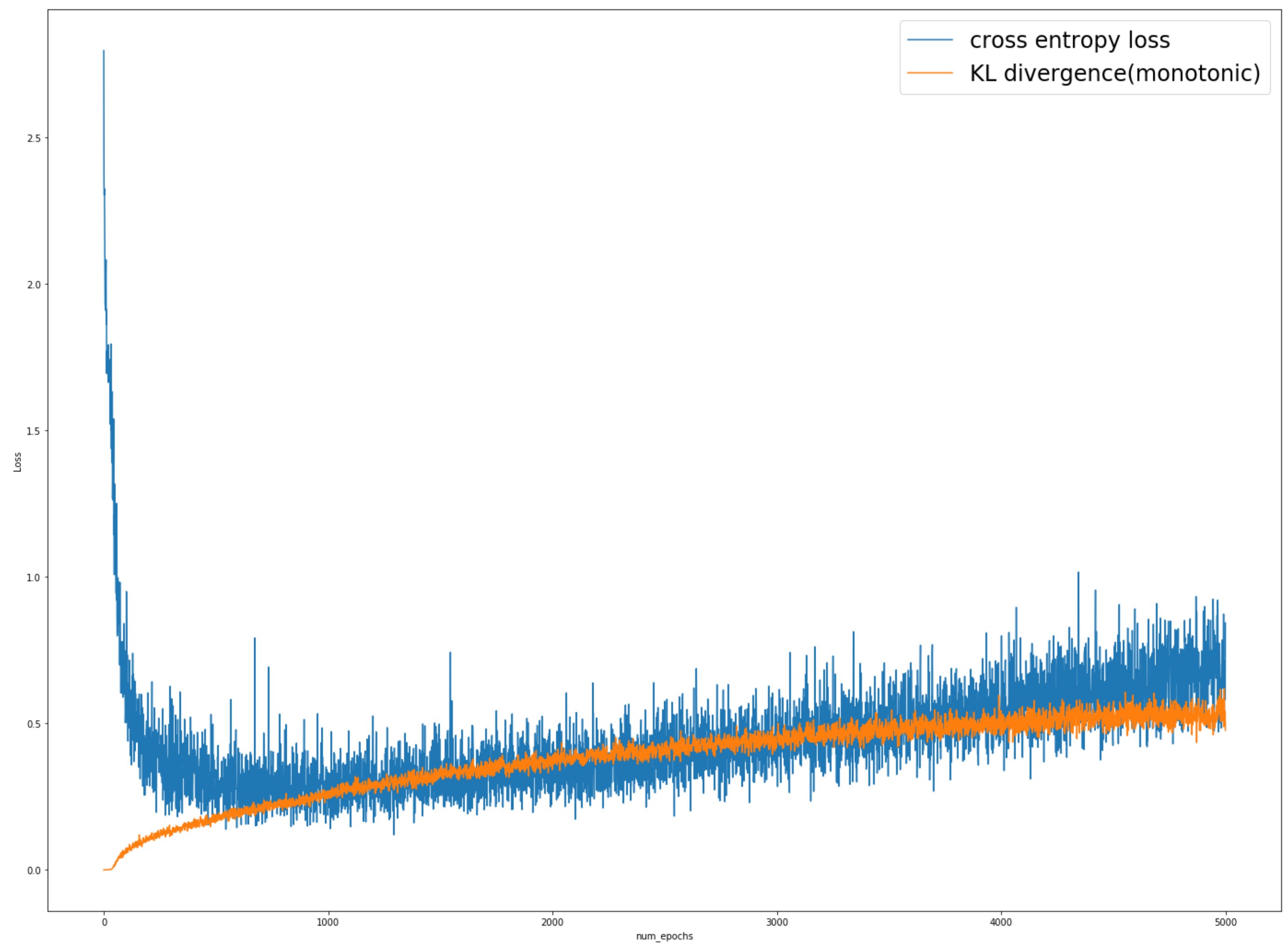
```
--------------------------------
input :  sent
target:  sends
pred  :  sends
--------------------------------
input :  split
target:  splitting
pred  :  splitting
--------------------------------
input :  flared
target:  flare
pred  :  flake
--------------------------------
input :  functioning
target:  function
pred  :  function
--------------------------------
input :  functioning
target:  functioned
pred  :  functioned
--------------------------------
input :  healing
target:  heals
pred  :  heals
================================
Average BLEU-4 score =  0.9285744042969881
```

Text Generation

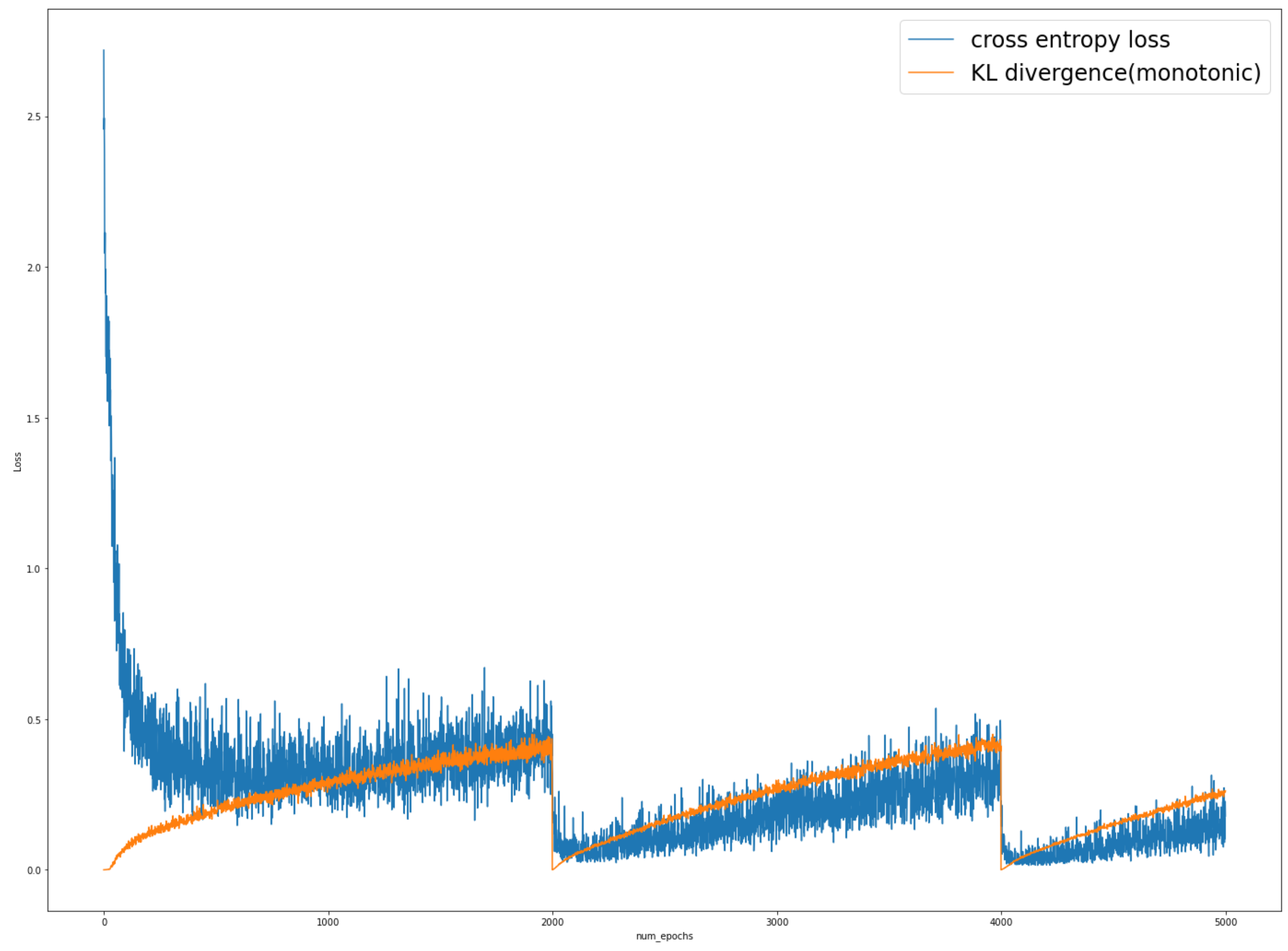
```
['repair', 'repairs', 'repairing', 'repeated']
['barl', 'barlies', 'barling', 'barled']
['require', 'requires', 'requiring', 'required']
['stam', 'stams', 'stamming', 'stagged']
['defire', 'defires', 'defiring', 'defired']
['mate', 'mates', 'matting', 'mated']
['reflect', 'reflects', 'reflecting', 'reflected']
['defend', 'defends', 'defending', 'defed']
['teach', 'teaches', 'teaching', 'taught']
['bhank', 'bhanks', 'bhanking', 'bhanked']
['seek', 'seeks', 'seeking', 'spent']
['assail', 'assails', 'assailing', 'assailed']
['snugg', 'snugs', 'snugging', 'snuggled']
['counter', 'counters', 'countering', 'countered']
Gaussian score:  0.32
```

Loss Visualization

- Monotonic

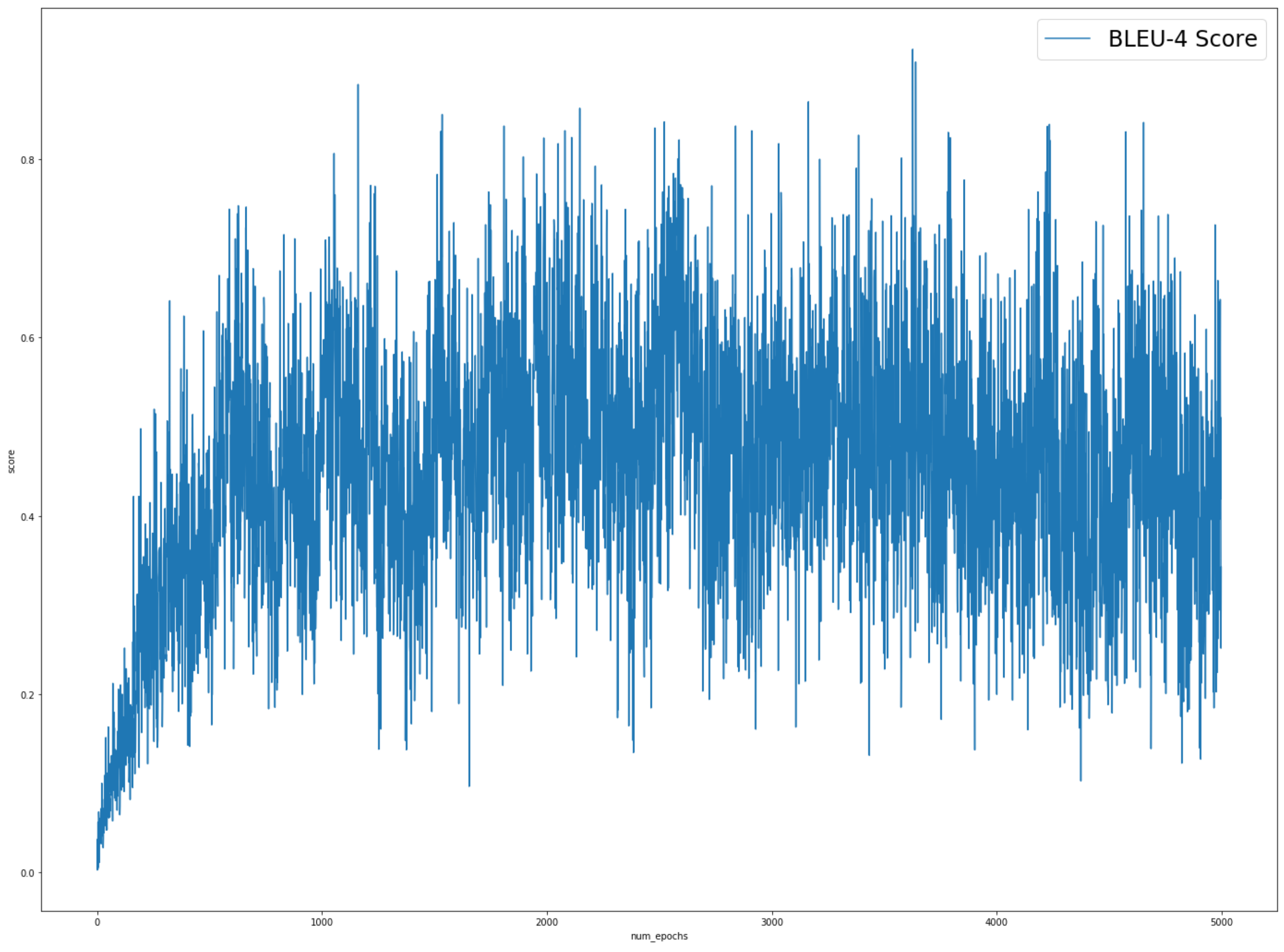


- Cyclical

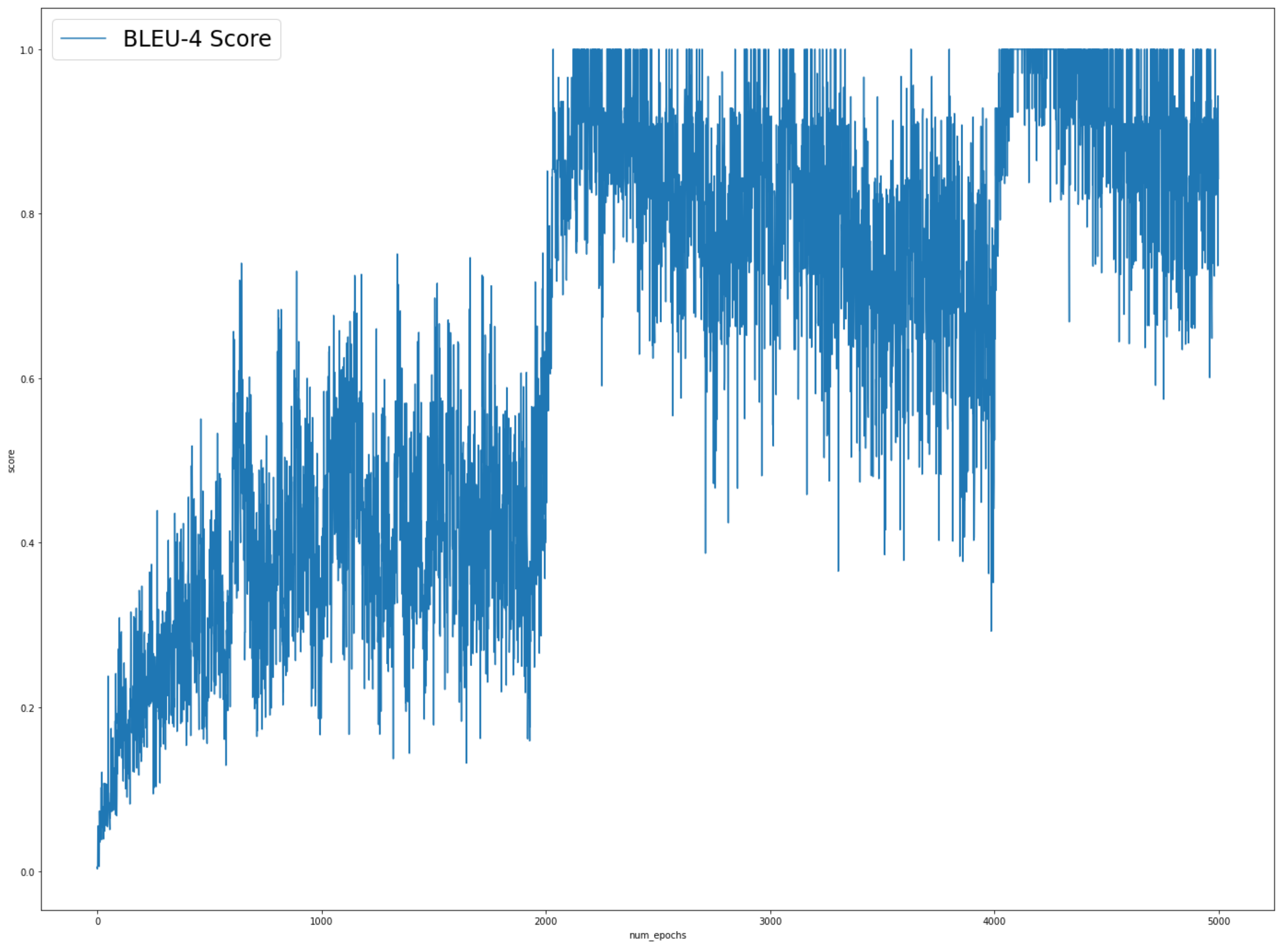


BLEU-4 Score Visualization

- Monotonic



- Cyclical

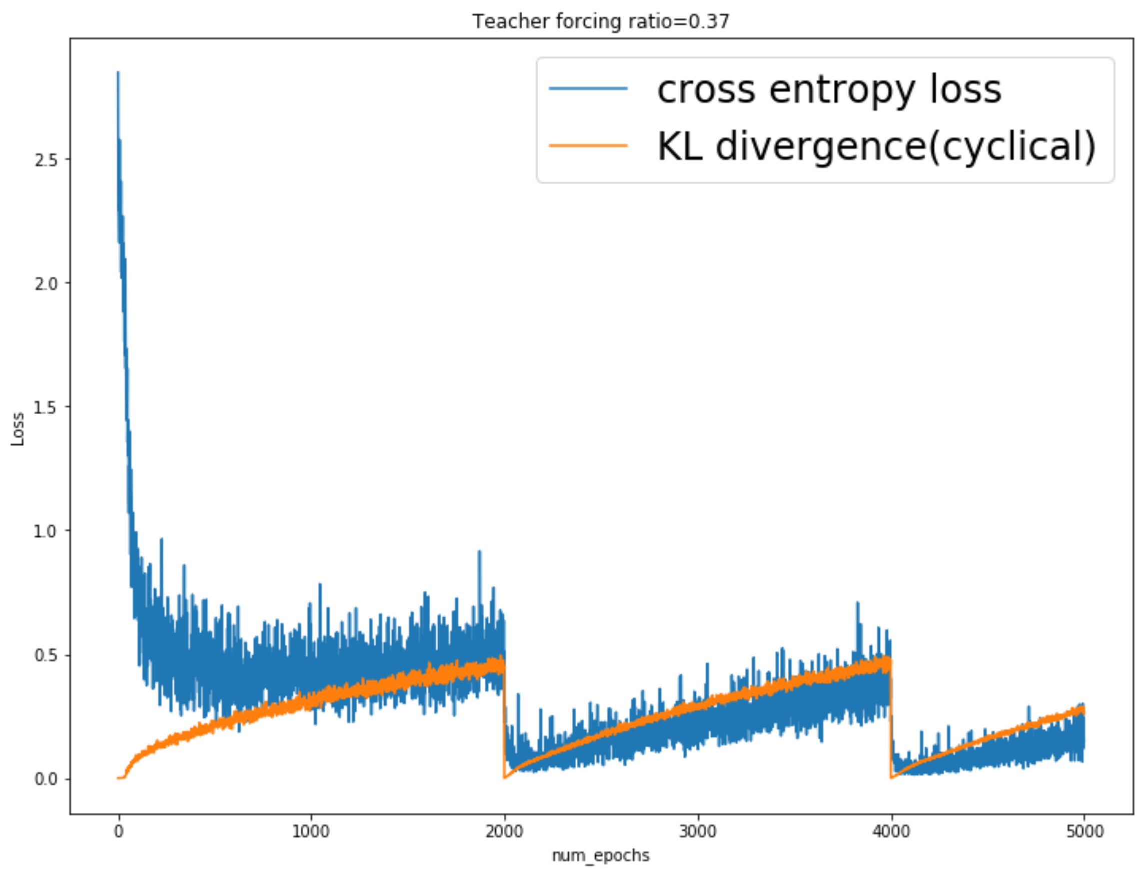


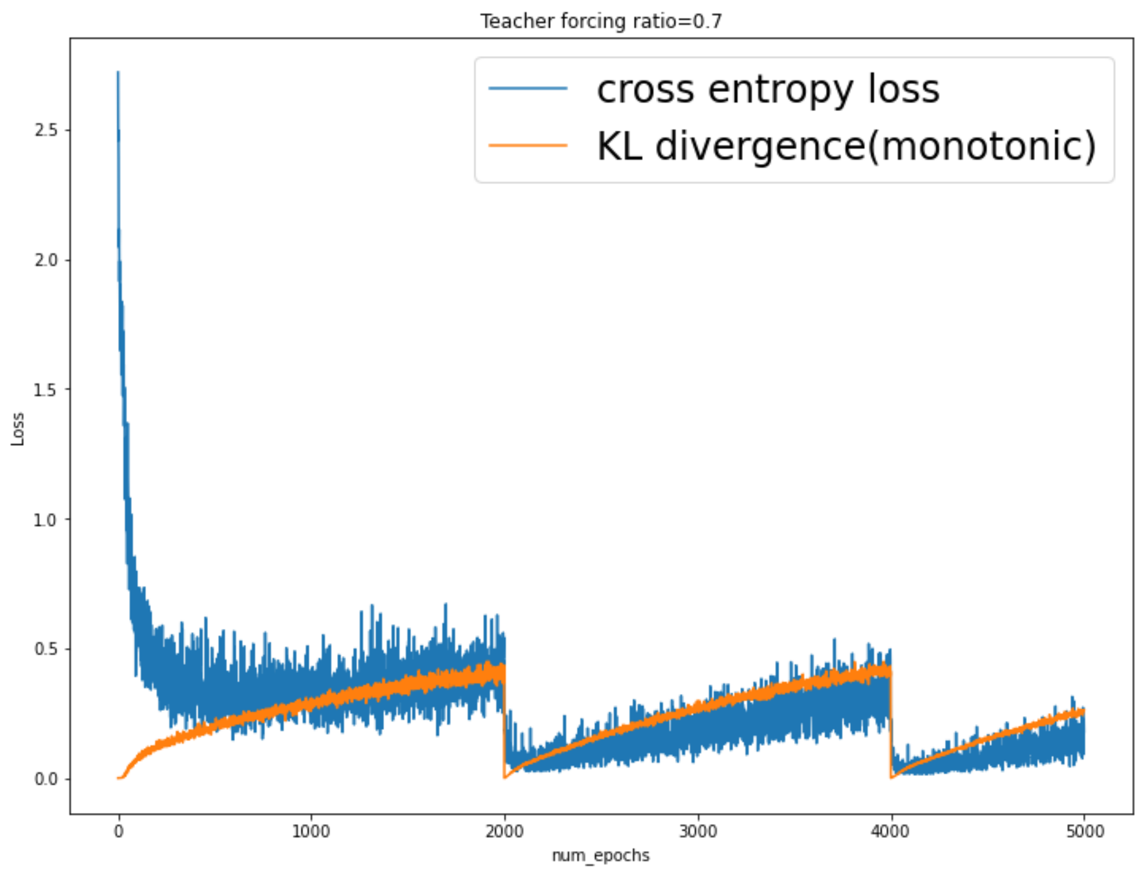
Discussions

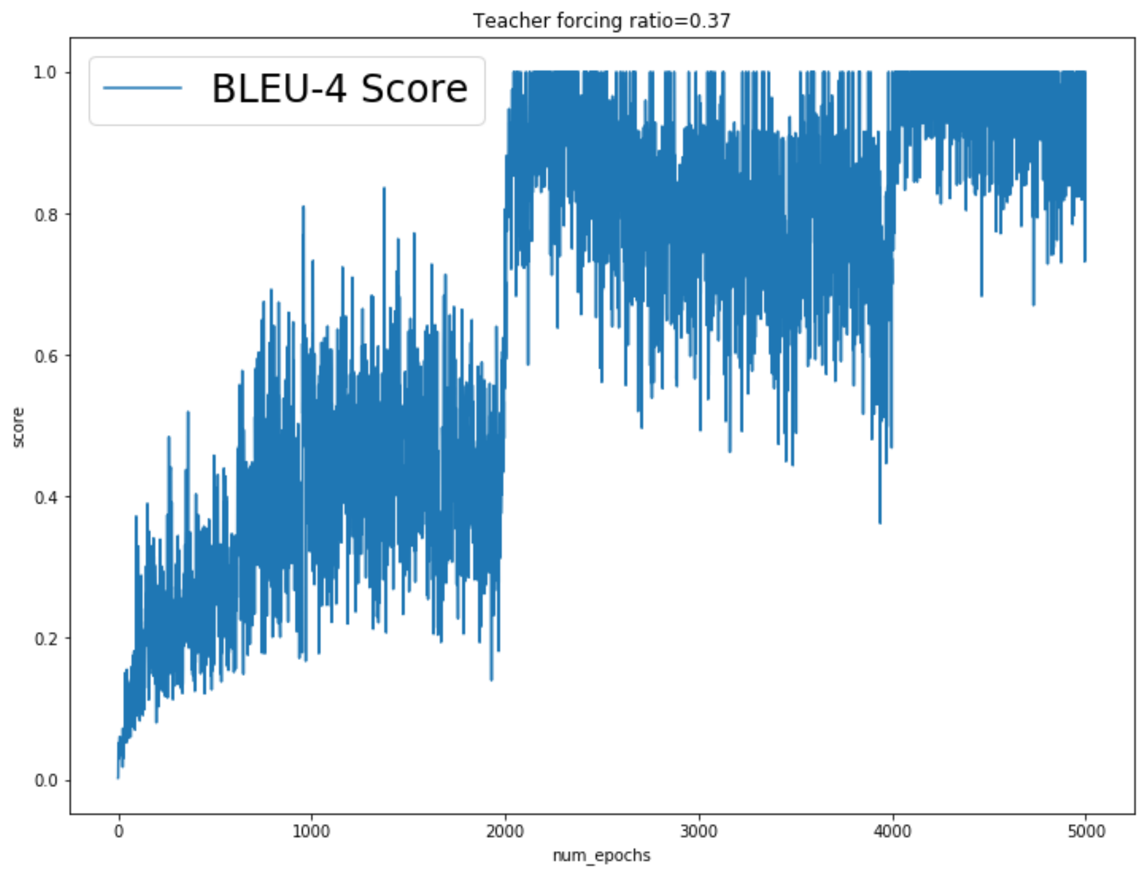
Impact of Teacher Forcing Ratio

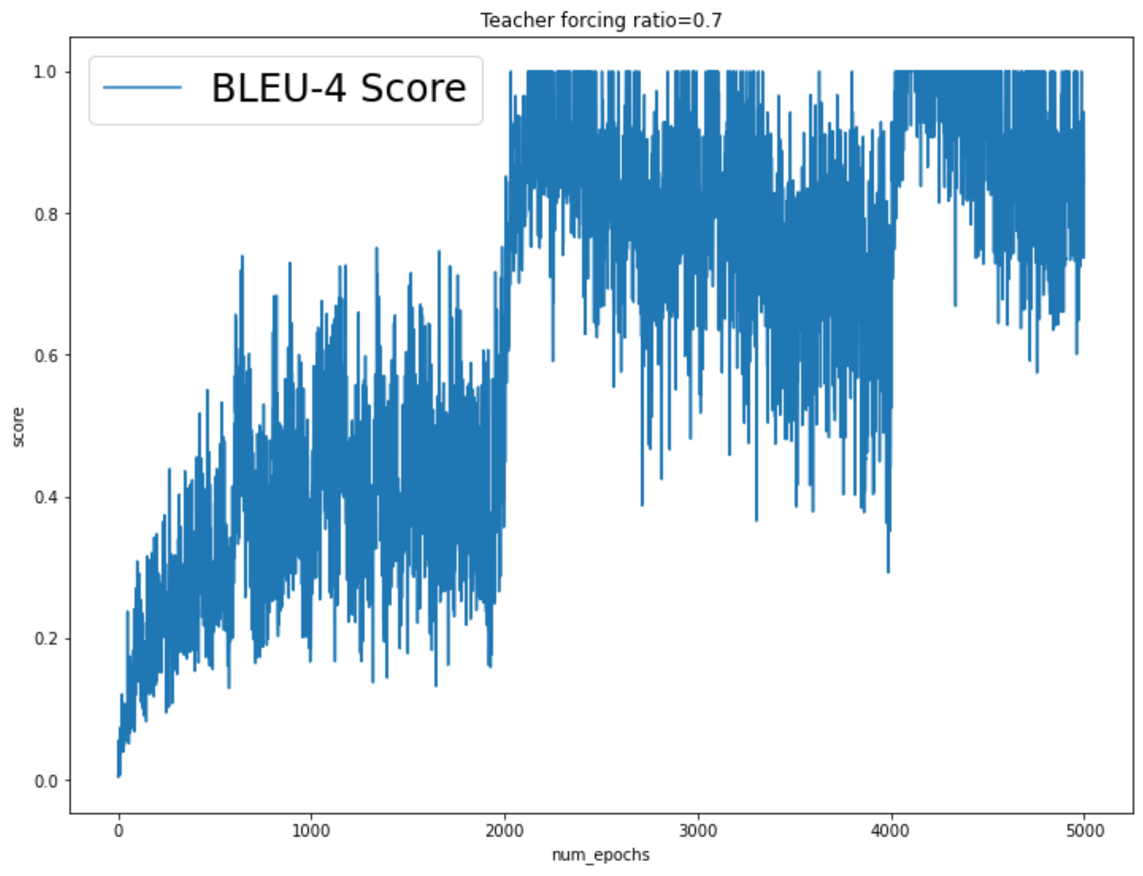
> Models that have recurrent connections from their outputs leading back into the model may be trained with teacher forcing.
> – Page 382, *Deep Learning* (https://www.deeplearningbook.org/), 2016

- Recursive output-as-input process can be used when training the model, but it can result in problems such as:
  - Slow convergence
  - Model instability

- Poor skill
- Teacher forcing is an approach to improve model skill and stability when training these types of models
    - Reference: What is Teacher Forcing for Recurrent Neural Networks? (https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/)
- Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$
- Teaching forcing has problem, such as open-loop issue
    - Open-loop issue: inputs seen at training and test time are different
    - To resolve this, it is common to train with both teacher-forced inputs and free-running inputs (generated by the output-to-input paths), or randomly choose between them
    - Experiments:
        - We conduct experiemnts to discuss the impact of teaching ratio
        - Loss
        - 



- BLEU
  - 



- From the experiments, we can see that the loss variance of training with lower teacher forcing ratio is higher; however, lower teaching forcing ratio performs better when evaluating (with BLEU-4)

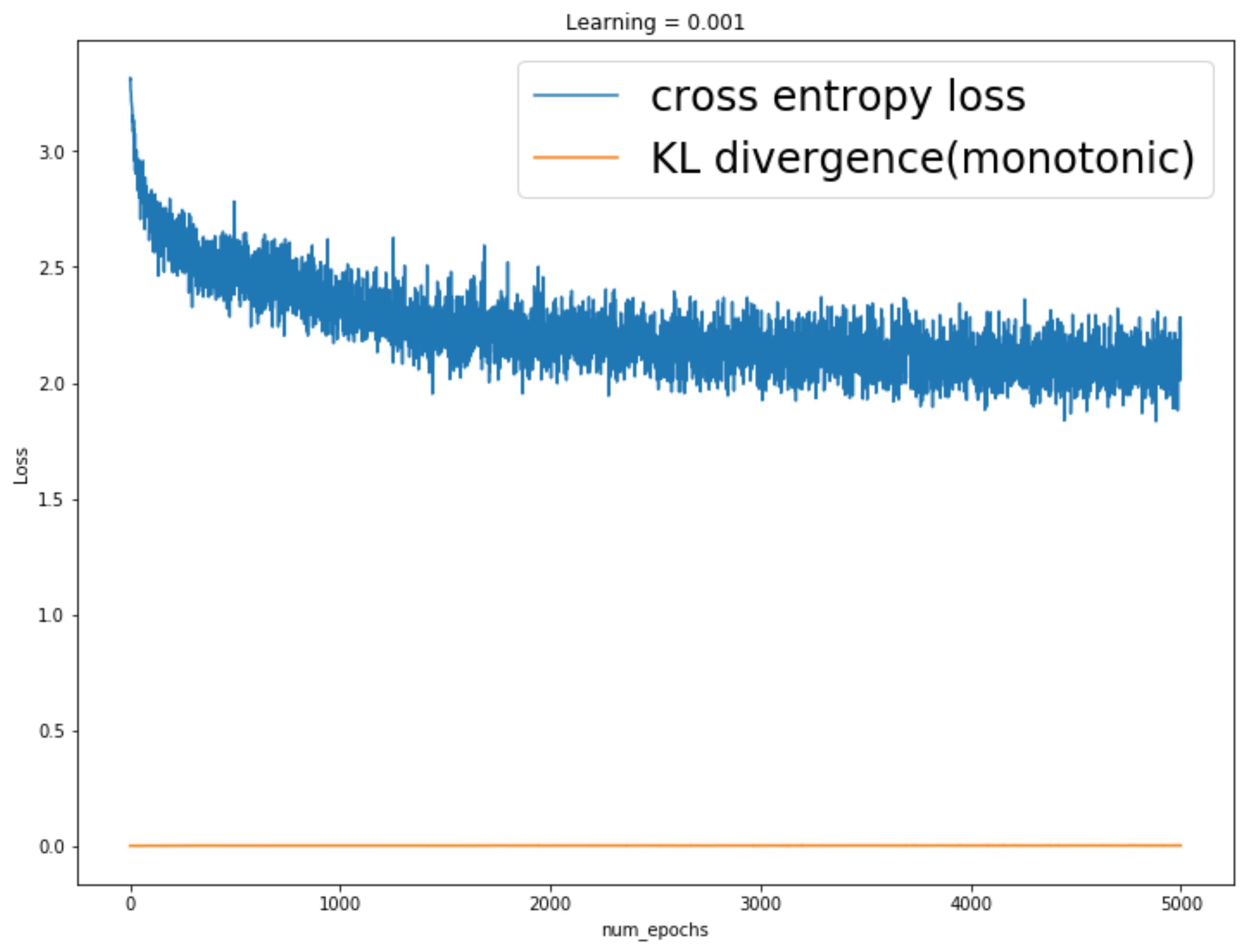
Impact of KL Weight

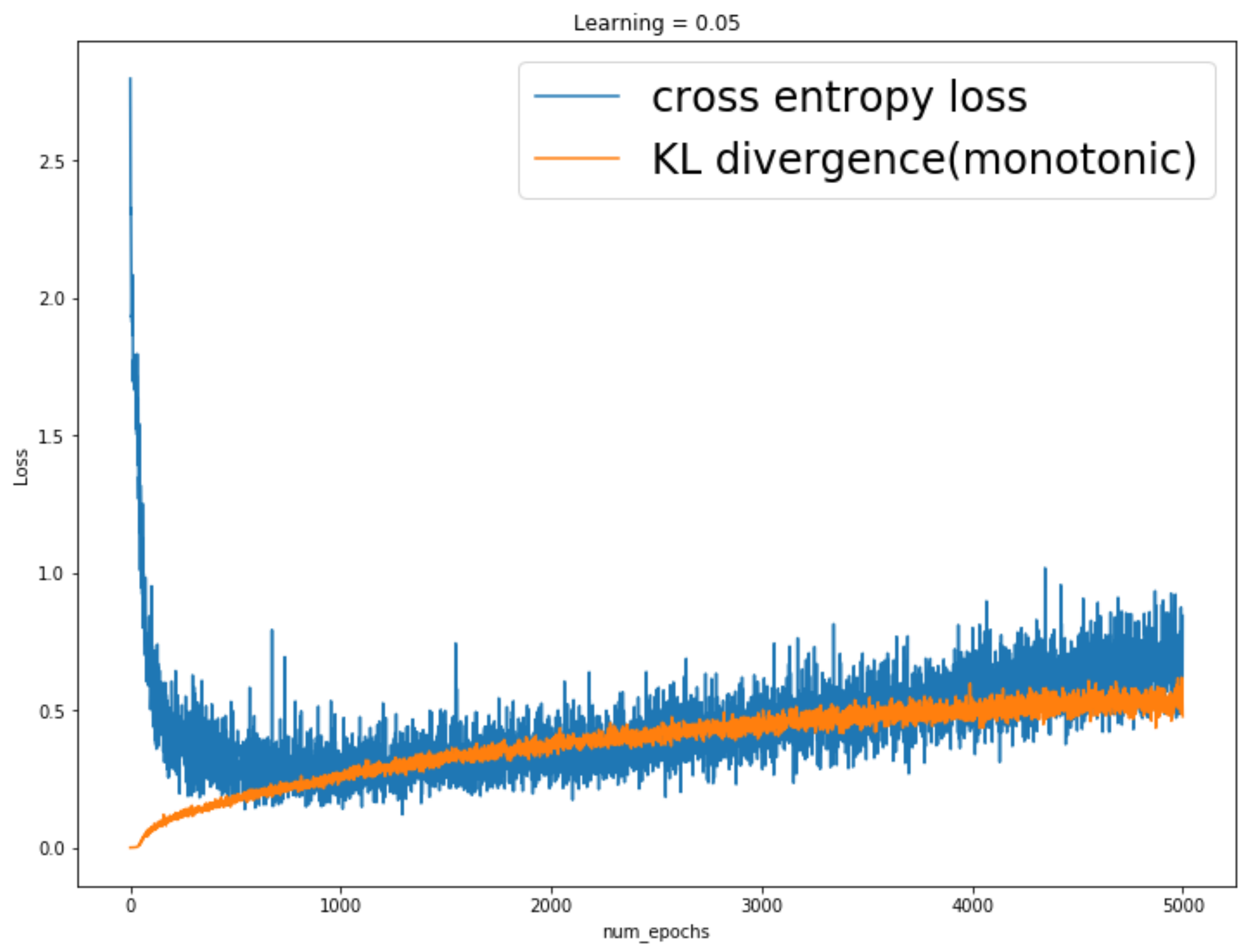
- As the visualization result in here (https://hackmd.io/MdPUWN2GQ4CnOCMh52MtLg?view#Loss-Visualization), we can see that monotonically increase the KL weight can avoid KL vanishing problem, while cross-entropy loss learn something but not to ignore the latent variable z
- We can see that using cyclical annealing will make cross entropy loss drop significantly when the weight resets. This may not be what we want, but we can improve it by waiting for cross entropy loss becoming more stable
- In the process of experiment, we find that starting with low KL weight (like `1e-4`) can actually make encoder more robust on tense conversion, but very weak on text generation (almost all get `0` on Gaussian score)

Impact of Learning Rate

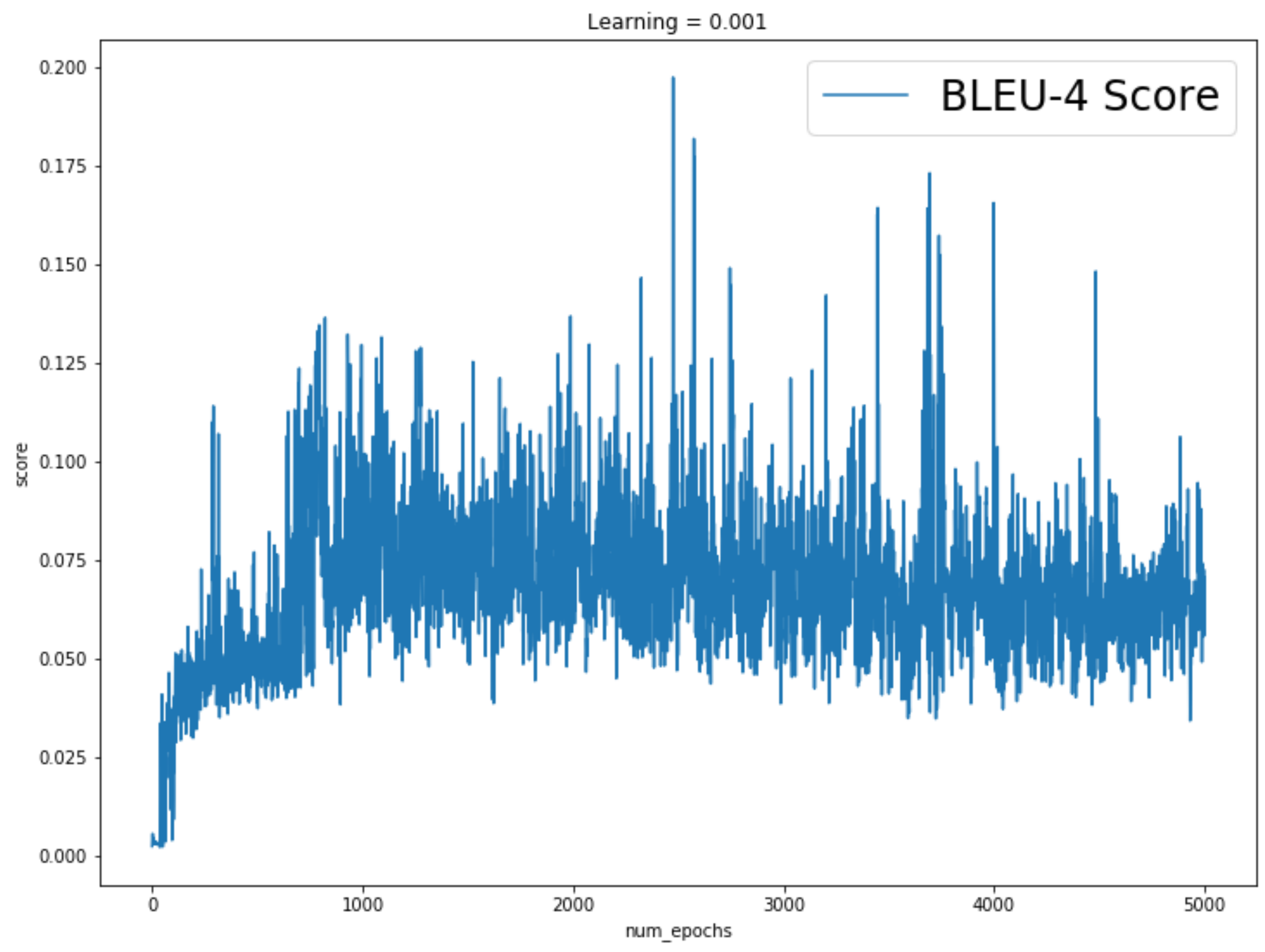
- We conduct experiments to explore the impact of learning rate on VAE

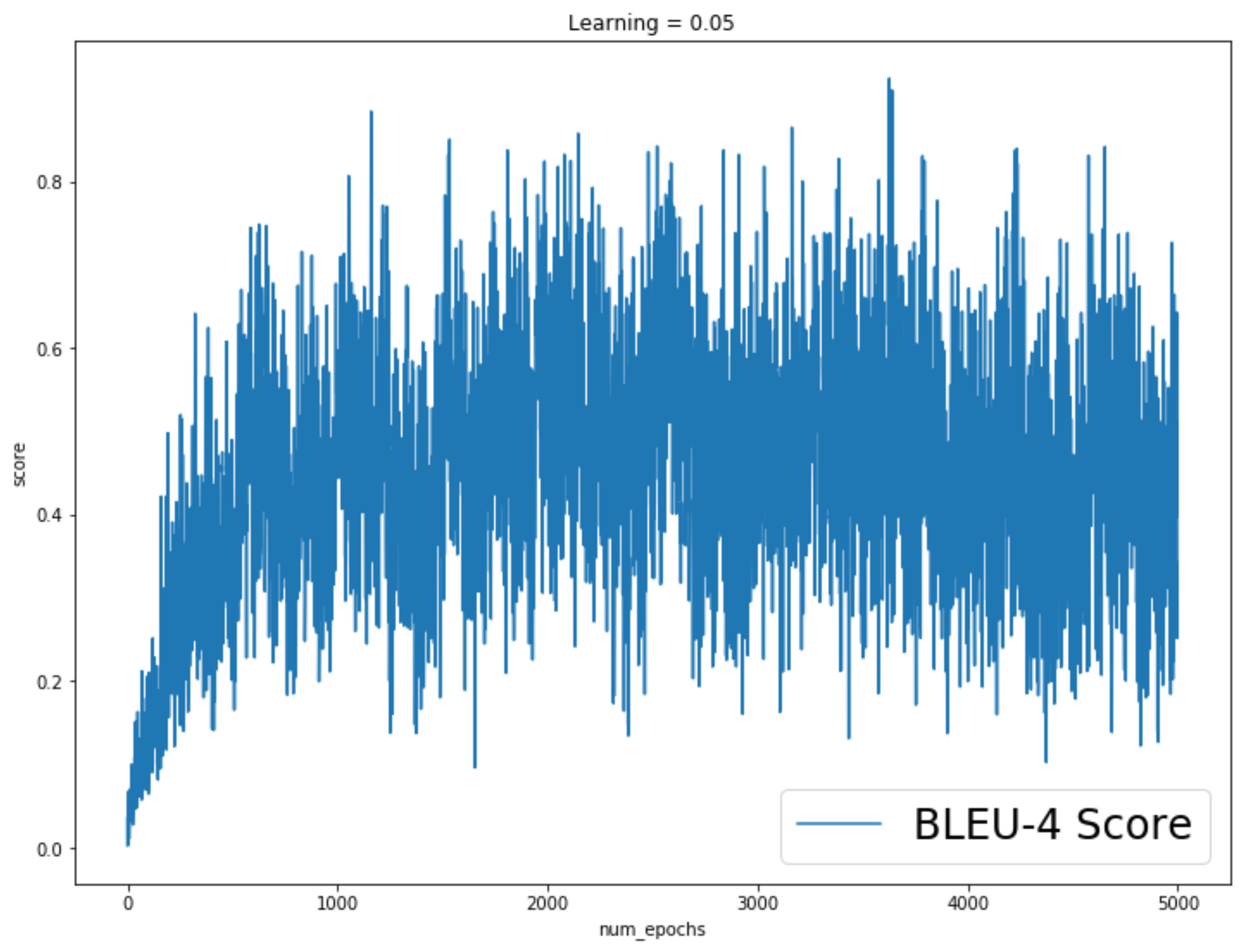
- Experiments:
    - Loss:




Learning = 0.05
cross entropy loss
KL divergence(monotonic)
Loss
num_epochs

- BLEU-4:


Learning = 0.001
BLEU-4 Score
score
num_epochs



- We can see that lower learning rate may converge slower