Lab 05 Report

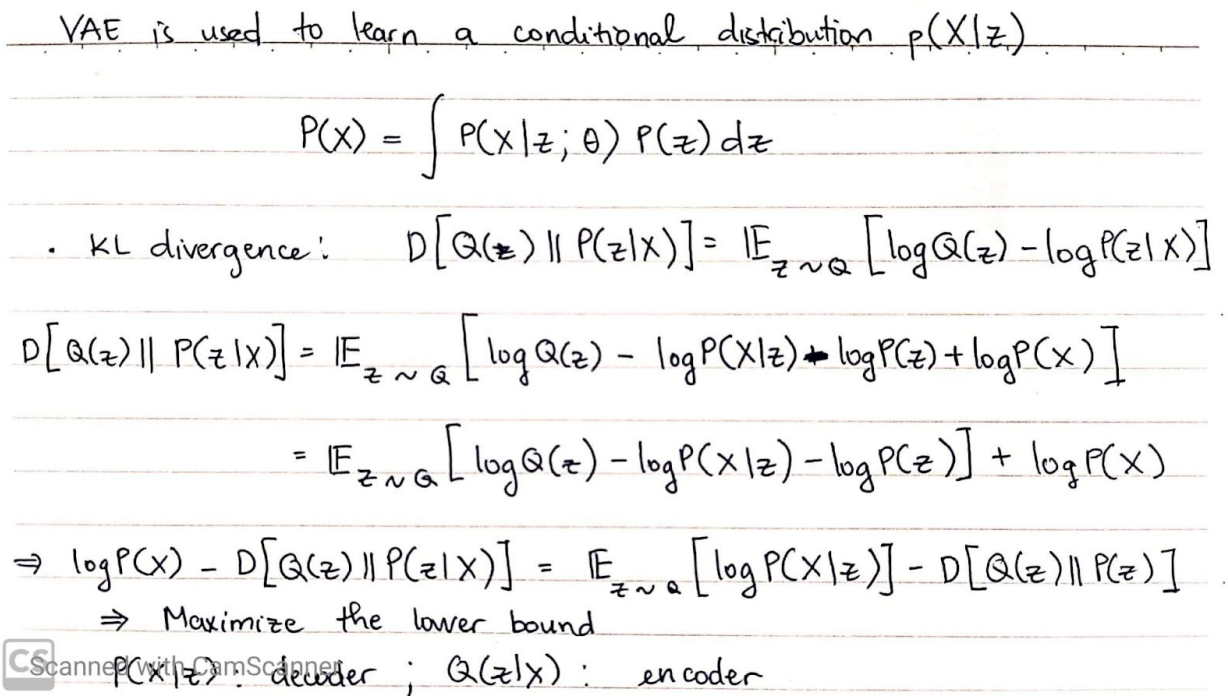
1. Introduction

In this lab, I will implement the conditional seq2seq VAE for English tense conversion and generation.

By experiment with various configurations, the highest result so far for word conversion is 0.82. On the other hand, the Gaussian score for 100-word generation is (sadly) 0.02 with 4 tenses as conditions.

2. Derivation of CVAE

The variational lower bound of VAE



In CVAE, both Q and P distribution is conditioned in other variable (another input data). Then the variational lower bound becomes

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

Now both the encoder q(Z|X, c) and decoder p(X|Z, c) need to take c as part of their input. At test time, samples can be generated by drawing Z ~ p(Z|c) and pass to decoder p(X|Z, c). In training time, the conditional prior P(Z|c) is learned.

3. Implementation details

A. Model implemetation (encoder, decoder, dataloader)

Encoder

We have embedding, LSTM and 2 linear layers for mean and (log) variance of Gaussian distribution.

```
class Encoder(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, condition_dim, latent_dim):
        super(Encoder, self).__init__()
        self.hidden_dim = hidden_dim
        self.embedding_dim = embedding_dim
        self.latent_dim = latent_dim
        self.condition_dim = condition_dim

        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.lstm = nn.LSTM(self.embedding_dim, self.hidden_dim + condition_dim)
        self.mu = nn.Linear(hidden_dim + condition_dim, latent_dim)
        self.sigma = nn.Linear(hidden_dim + condition_dim, latent_dim)
```

Encoder

The LSTM receives input from embedding layer and its hidden tensor is concatenation of hidden and condition (which is also fed from input). After forwarding to mean and variance layer for Gaussian, we sample latent from that Gaussian distribution to pass to decoder.

```
def forward(self, word, condition):
    c_0 = torch.zeros((1, 1, self.hidden_dim + self.condition_dim), device=Const.device)
    h_0 = torch.zeros((1, 1, self.hidden_dim), device=Const.device)
    h_0 = torch.cat((h_0, condition), dim=2)

    embedded = self.embedding(word).view(-1, 1, self.embedding_dim)
    outputs, (hidden, cell) = self.lstm(embedded, (h_0, c_0))

    mu = self.mu(hidden)
    sigma = self.sigma(hidden)

    z = self.sample_z() * torch.exp(sigma / 2) + mu

    return z, mu, sigma

def sample_z(self):
    return torch.normal(
        torch.FloatTensor([0] * self.latent_dim),
        torch.FloatTensor([1] * self.latent_dim)
    ).to(Const.device)
```

Decoder

Like encoder, we need embedding layer for input character, LSTM for learning hidden state and linear layer for distribution of predicted character.

In decoder, we need to concat the latent from encoder with the condition (tense) of the corresponding output word.

```
class Decoder(nn.Module):
    def __init__(self, output_dim, embedding_dim, hidden_dim, condition_dim, latent_dim):
        super().__init__()

        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.condition_dim = condition_dim

        self.latent2hidden = nn.Linear(latent_dim + condition_dim, hidden_dim + condition_dim)
        self.embedding = nn.Embedding(output_dim, embedding_dim)
        self.lstm = nn.LSTM(self.embedding_dim, self.hidden_dim + condition_dim)
        self.out = nn.Linear(hidden_dim + condition_dim, output_dim)
```

The forward pass of decoder similar to encoder, except it output the predicted character each time we feed an input into the LSTM.

```
def forward(self, input, hidden):
    embedded = self.embedding(input).view(1, 1, self.embedding_dim)

    output, hidden = self.lstm(embedded, hidden)
    output = self.out(output).view(-1, self.output_dim)

    return output, hidden

def initHidden(self, z, condition):
    return (
        self.latent2hidden(torch.cat((z, condition), dim=2)),
        torch.zeros((1, 1, self.hidden_dim + self.condition_dim), device=Const.device)
    )
```

Conditional VAE

Conditional seq2seq VAE is a combined of Encoder and Decoder. We also includes a dataset (word_processing) for easier to implement.

```
class EncoderDecoder(nn.Module):
    def __init__(self, io_dim, embedding_dim, hidden_dim, condition_dim, latent_dim):
        super().__init__()
        self.encoder = Encoder(io_dim, embedding_dim, hidden_dim, condition_dim, latent_dim)
        self.decoder = Decoder(io_dim, embedding_dim, hidden_dim, condition_dim, latent_dim)
        self.condition_embedding = nn.Embedding(4, condition_dim)
        self.word_processing = dataset.WordProcessing()
```

Each forward pass into CVAE model is a full word prediction. Firstly, we embedding both input and output conditions. Then feed the input into encoder to get the latent tensor, use that latent with output condition for decoder's input.

We also use teacher forcing and dropout mechanism to enhance the performance of the model. Teacher forcing use the ground truth as the next input for decoder, which helps to prevent the error to propagate further. Word dropout randomly set a next input to UNKNOWN character. This technique weakens the decoder by randomly replacing the input character tokens with the unknown token.
After one forward pass, you get the predicted word and parameters of latent distribution which is Gaussian (mean, log variance).

```python
def forward(
        self, input_word, input_tense, target_word, target_tense,
        teacher_forcing_ratio=0.5, word_dropout_ratio=0.1
):
    input_condition_embedded = self.condition_embedding(input_tense).view(1, 1, -1)
    target_condition_embedded = self.condition_embedding(target_tense).view(1, 1, -1)

    batch_size = target_word.shape[0]
    max_len = target_word.shape[1]
    target_vocab_size = self.decoder.output_dim
    outputs = torch.zeros(max_len, batch_size, target_vocab_size).to(Const.device)

    z, mu, sigma = self.encoder(input_word[:, 1:-1], input_condition_embedded)

    hidden = self.decoder.initHidden(z, target_condition_embedded)
    input = target_word[:, 0]
    # print(input)
    teacher_force = random.random() < teacher_forcing_ratio

    for t in range(1, max_len):
        output, hidden = self.decoder(input, hidden)
        outputs[t] = output
        top1 = torch.argmax(output, dim=1)

        input = (target_word[:, t] if teacher_force else top1)
        word_dropout = random.random() < word_dropout_ratio
        if word_dropout:
            input = torch.tensor(
                self.word_processing.char2int(self.word_processing.UNK_TOKEN)
            ).to(Const.device)

    return outputs, mu, sigma
```

Reparameterization trick

After encoding, there are mean μ and variance σ. And in an original form, it can not backpropagation through the random node z which is drawn from a distribution $q_\theta(z|x)$. To solve this problem, it is required using reparameterization trick which introduces a ε drawn from Normal(0,1), so that we can sample following this formula: $z = \mu + \sigma \odot \varepsilon$

```
def forward(self, word, condition):
    c_0 = torch.zeros((1, 1, self.hidden_dim + self.condition_dim), device=Const.device)
    h_0 = torch.zeros((1, 1, self.hidden_dim), device=Const.device)
    h_0 = torch.cat((h_0, condition), dim=2)

    embedded = self.embedding(word).view(-1, 1, self.embedding_dim)
    outputs, (hidden, cell) = self.lstm(embedded, (h_0, c_0))

    mu = self.mu(hidden)
    sigma = self.sigma(hidden)

    z = self.sample_z() * torch.exp(sigma / 2) + mu

    return z, mu, sigma

def sample_z(self):
    return torch.normal(
        torch.FloatTensor([0] * self.latent_dim),
        torch.FloatTensor([1] * self.latent_dim)
    ).to(Const.device)
```

Data Loader

For data management, I create some utilities to convert between word and tensor, and use dataloader in pytorch for implementation. In training, we needs to input the same word and tense for both input and target. So after reading from 'train.txt' file, we reshape all words to 1 column. Then it is easy to implement __getitem__ function.As a requirement, every word before return to training and testing need to be converted in to a list of number (every character represent by a number). During the converting process, it also attaches a <SOS> and <EOS> to the beginning and the end of converted word as a signal of the start of string and end of string for the model. The tenses is represented by 4 number from 0 to 3.

```
def __len__(self):
    return len(self.data)

def __getitem__(self, index):
    if self.mode == Mode.TRAIN:
        return self.word_processing.word2tensor(self.data[index]), index % 4
    else:
        target_tenses = np.array([[0, 3], [0, 2], [0, 1], [0, 1], [3, 1], [0, 2], [3, 0], [2, 0], [2, 3], [2, 1]])
        return self.word_processing.word2tensor(self.data[index][0]), target_tenses[index][0], \
               self.word_processing.word2tensor(self.data[index][1]), target_tenses[index][1]
```

Dataloader implmentation

```python
def word2int(self, word):
    return np.array([self.char2int(c) for c in word])

def word2tensor(self, word):
    list_word = [self.SOS_TOKEN] + list(word) + [self.EOS_TOKEN]
    list_word = self.word2int(list_word)

    return torch.tensor(list_word).long()
```

Conversion from word to tensor

Generation

The latent is generated by function *sample_z()* , which is draw a random sample from latent distribution. After that, the latent is fed into decoder with each condition for word generation.

```python
for word_id in range(100):
    noise = model.encoder.sample_z()
    words = []
    for i in range(len(english_tense.tenses)):
        condition = torch.tensor([i]).long().to(Const.device)
        target_condition_embedded = model.condition_embedding(condition).view(1, 1, -1)
        outputs = decode_inference(model.decoder, noise, target_condition_embedded)
        outputs = outputs[1:].view(-1, outputs.shape[-1])
        prediction = torch.argmax(torch.softmax(outputs, dim=1), dim=1)
        prediction = word_processing.tensor2word(prediction)
        print('{:20s} : {}'.format(english_tense.tenses[i], prediction))
        words.append(prediction)
    generated.append(words)

print(Gaussian_score(generated))
```

B. Specify the hyperparameters (KL weight, learning rate, teacher forcing ratio, epochs, etc).

Teacher forcing 0.5
KL weight (use both monotonic and cyclic) from 0 to 0.002 (scaled)
Learning rate 0.01
Epochs 100

4. Results and Discussion

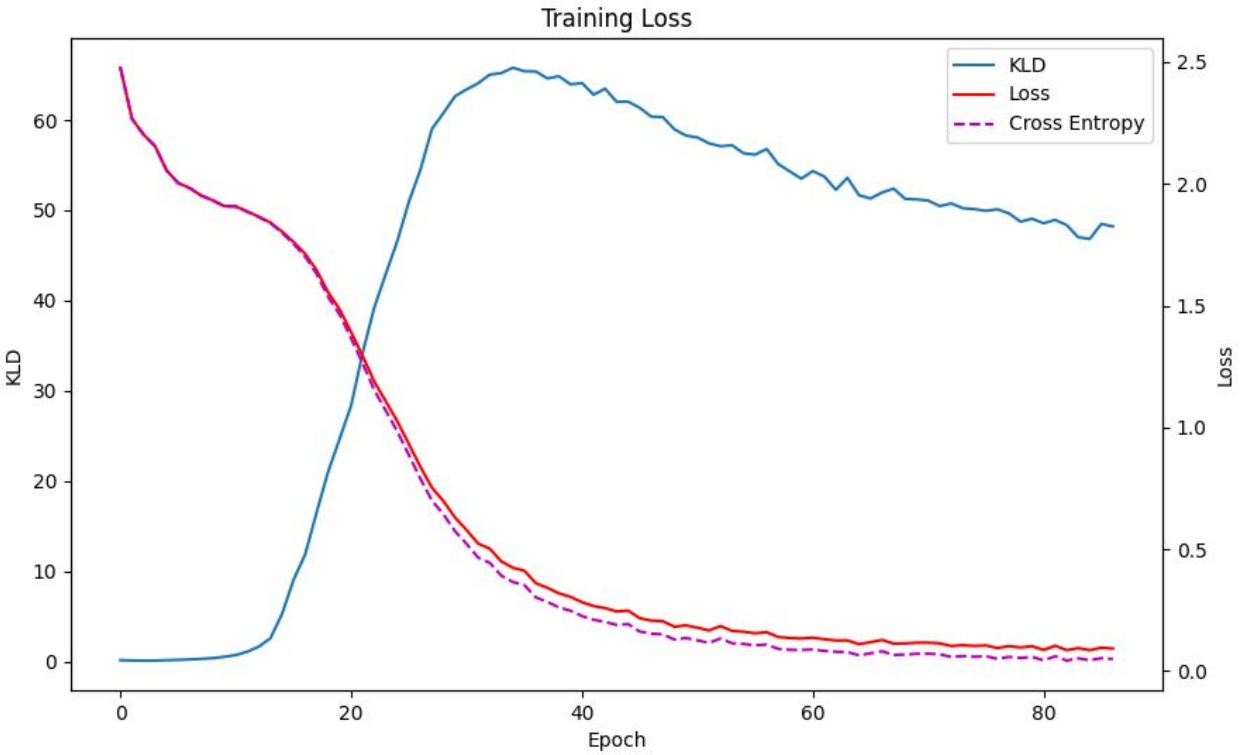
A. Results of tense conversion and generation

```
['fousid', 'sounds', 'sougming', 'sounsed']
['stide', 'tuifs', 'tuiring', 'tuifed']
['glick', 'tilch', 'tulching', 'twich']
['swober', 'bwobels', 'swobeling', 'bswired']
['linch', 'linches', 'linching', 'linched']
['intoore', 'inworms', 'inworming', 'inswormed']
['designe', 'designes', 'designing', 'designed']
['satagg', 'gannus', 'satgling', 'satgned']
2
0.02
```

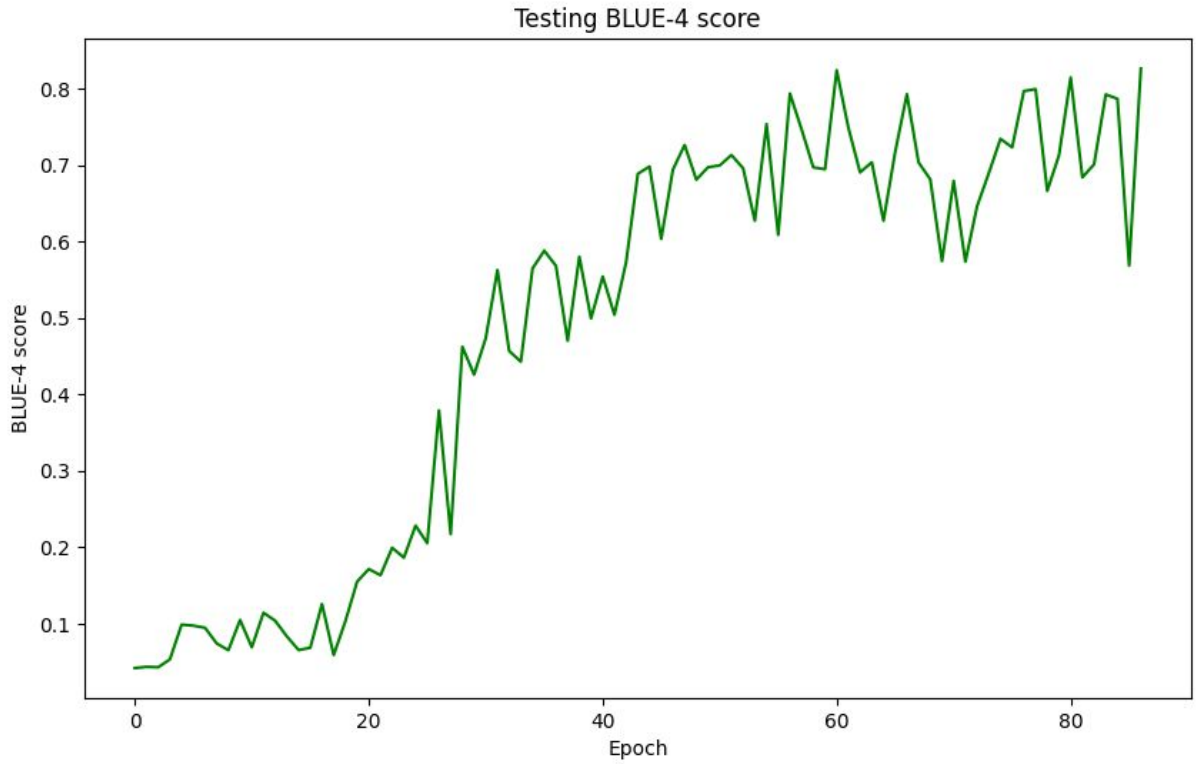
Word generation result

```
input:      abandon
target:     abandoned
prediction: abandon

input:      abet
target:     abetting
prediction: abeting

input:      begin
target:     begins
prediction: begins

input:      expend
target:     expends
prediction: expends

input:      sent
target:     sends
prediction: sents

input:      split
target:     splitting
prediction: spliting

input:      flared
target:     flare
prediction: flare

input:      functioning
target:     function
prediction: function

input:      functioning
target:     functioned
prediction: functiond

input:      healing
target:     heals
prediction: heals

BLEU score avg: 0.8264349603179582
```

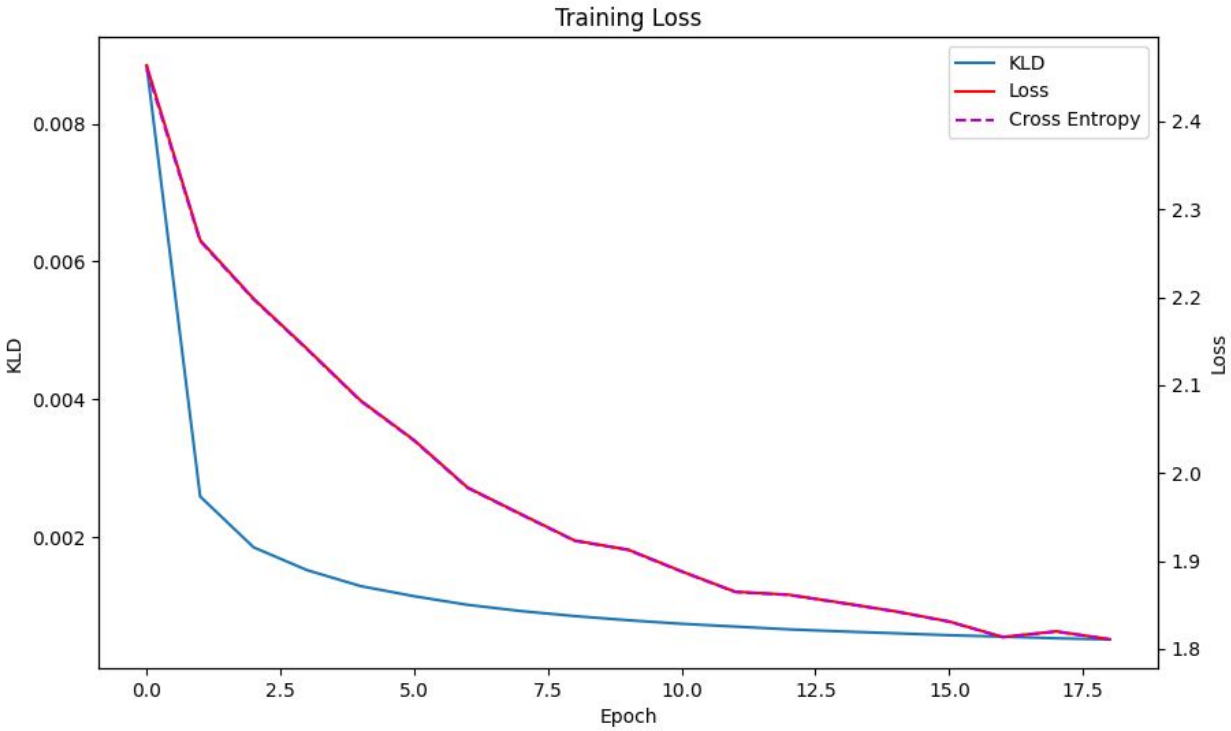
Word conversion result



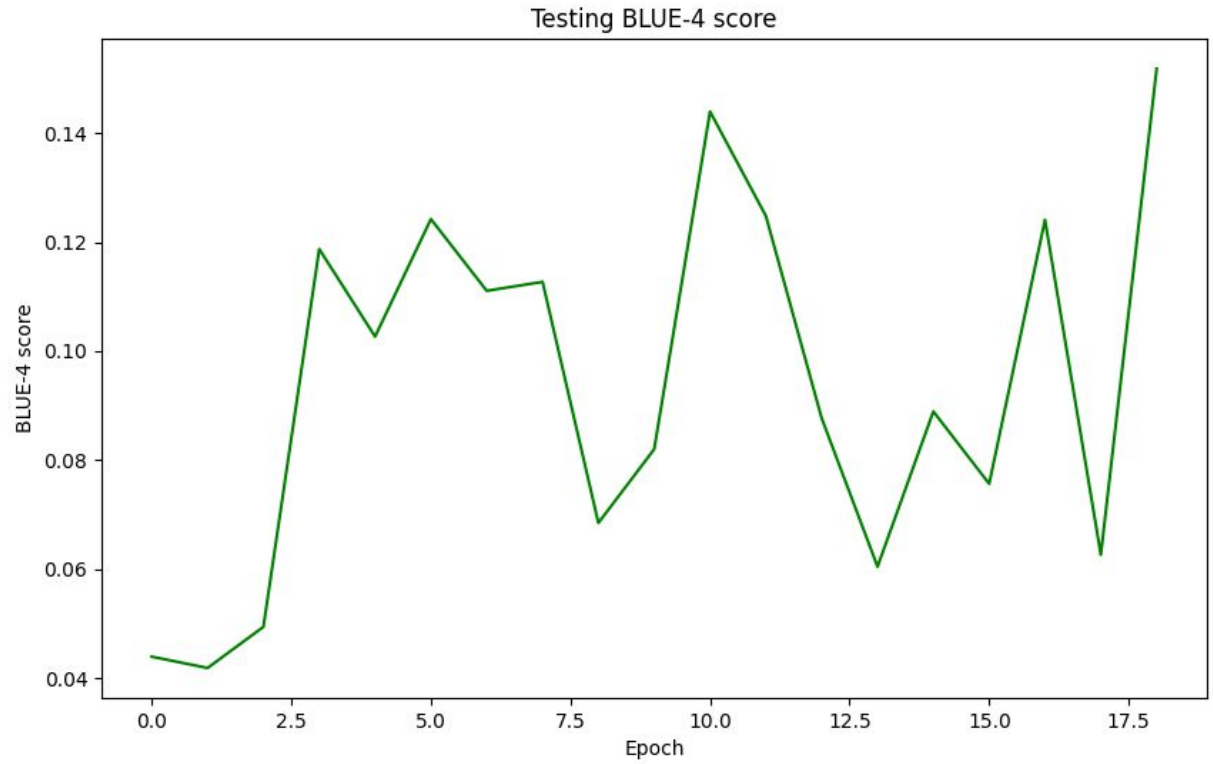
Teacher forcing 0.5; Word dropout 0.1; LR 0.01; KL weight cyclical



Teacher forcing 0.5; Word dropout 0.1; LR 0.01; KL weight cyclical



Teacher forcing 0.5; Word dropout 0.1; LR 0.01; KL weight monotonic



Teacher forcing 0.5; Word dropout 0.1; LR 0.01; KL weight monotonic

B. Discuss based on teacher forcing ratio, KL weight and learning rate.

Teacher forcing allow us to avoid back-propagation through time in models that lack hidden-to-hidden connections. The disadvantage of strict teacher forcing arises if the network outputs fed back as input. In this case, the kind of inputs that the network sees during training could be quite different from the kind of inputs that it will see at test time.

By trying different configuration, we can see that the teacher forcing and KL weight are significantly influence the performance.

When the KL term gets higher and higher in the first 30 epochs (in correct configuration) and it drops steadily while the total loss and cross entropy decrease gradually. It means at the beginning the model confuses about the tense of a word but it could recreate the same word without tense easily. In later, the KL term reduce show that the model is able to realize how to combine the word with the tense or condition.

A cyclical annealing shecdule allows the progressive learning of more meaningful latent codes, by leveraging the informative representations of previous cycles as warm restarts.