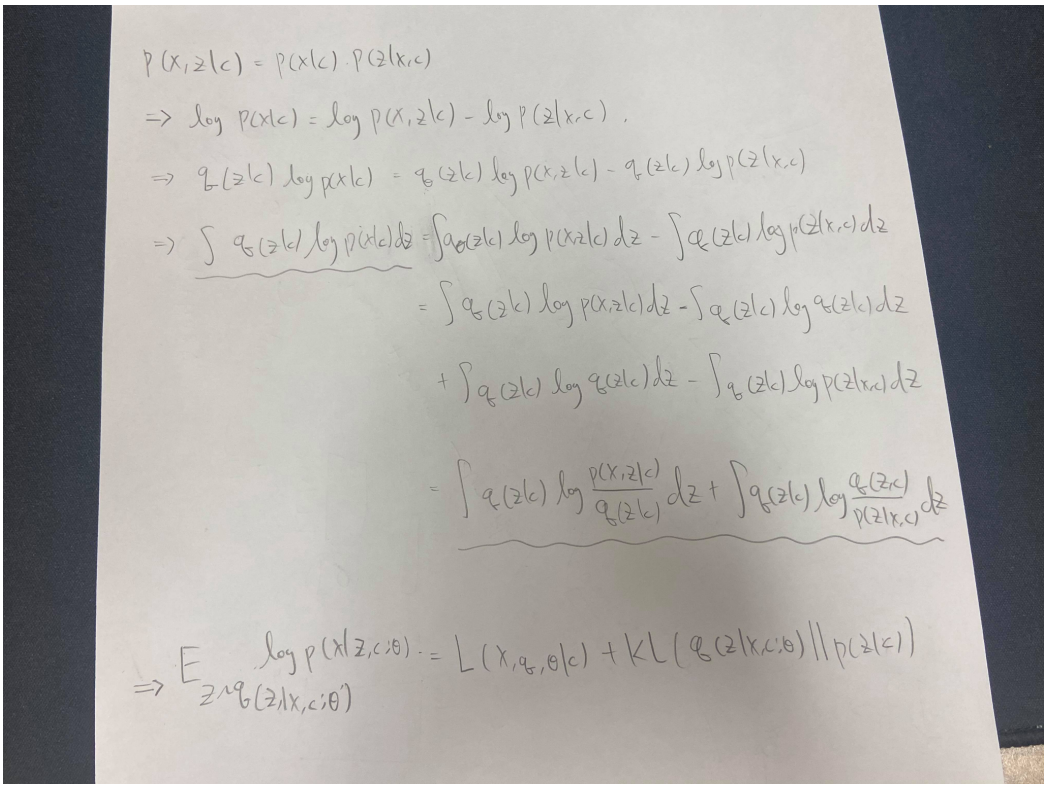
# 309554042 數據所 李政毅
# Report

• Introduction (5%)

藉由用LSTM的RNN框架實做CVAE, Task是要處理英文單字的時態轉換問題, 例如: abandon(sp) -> abandoned(p), condtion有4種時態, 分別為simple present(sp), third person (tp), present progressive(pg), simple past(p)。並在decoder上透過Gaussaian noise生成100個有4個時態的單字, 再用Gaussain score來計算分數, 以及透過Bleu score計算test data的分數。

• Derivation of CVAE (5%)




• Implementation details (15%)

CVAE的架構是由三個所組成, 可以分為encoder -> middle sample -> decoder.

Encoder:

一次只處理一個字母, 一開始把這個字母根據char2idx 的dictionary轉成idx, 共有28種可能, 分別為[SOS,EOS,a-z], 轉成idx後丟入encoder, 透過embedding layer轉成hidden_size=256大小的vector, 這是用來描述這個字母的vector, 之後再丟入LSTM跑, 最後輸出output, hidden, cell。

```
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(CVAE.Encoder,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_state, cell_state):
        embedded = self.embedding(input).view(1,1,-1)  # view(1,1,-1) due to input of lstm must be (seq_len,batch,vec_dim)
        output, (hidden_state, cell_state) = self.lstm(embedded, (hidden_state, cell_state))
        return output, hidden_state, cell_state
```

middle sample:
接著把輸出的hidden透過fc生成latent_size=32的 logvar 跟mean, 有了logvar跟mean後就可以透過reparameterization trick去sample一個大小是latent_size=32的vector, 接著把這個大小32的sample vector和conditional_size=8的condition vector concate在一起後, 再透過fc轉成hidden_size=256的vector, 並作為接著decoder的hidden。之所以用logvar的原因是因為這邊的logvar是透過神經網路去學的, variance一定大於零, 但網路的output可能是負值 , 因此取log避免這個問題。

```
"""
middle sample
"""
#calculate mean & logvar
mean=self.hidden2mean(encoder_hidden)
logvar=self.hidden2logvar(encoder_hidden)
# use mean & logvar sample
latent=self.reparameterization(mean,logvar)
decoder_hidden = self.latentcondition2hidden(torch.cat((latent, condition), dim=-1))
decoder_cell = self.decoder.init_c0()
decoder_input = torch.tensor([[SOS_token]], device=device)    #the begin of decoder default as SOS_token
```

decoder:
把默認是EOS的input,middle sample 轉換過得到的 hidden, 初始化的cell,傳入decoder, 丟到LSTM裡面跑, 最後decoder會把生成hidden_size = 256的output轉回跟input大小相同的28, 藉由比對input跟output來計算loss。

```
# Decoder
class Decoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(CVAE.Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.hidden2input = nn.Linear(hidden_size, input_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, cell):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, (hidden, cell) = self.lstm(output, (hidden, cell))
        output = self.softmax(self.hidden2input(output[0]))
        return output, hidden, cell
```

reparameterization trick:
我們希望透過x ~ N(0,1) -> y = ax + b -> y ~ N(b,a^2)的方式去做sample, 所以我們需要標準差以及平均數。而logvar實際上是 log sigma^2  = log sigma + log sigma , 所以事先乘上0.5後及得到log sigma 再取exp後就能得到真正的stdandard deviation。

```
def reparameterization(self,mean,logvar):
    """
    x ~ N(0,1)
    y = ax + b
    y ~ N(b,a^2)
    """
    std=torch.exp(0.5*logvar)
    eps=torch.randn_like(std)
    latent=mean+eps*std
    return latent
```

dataloader:
透過ascii code查到 a是從97開始，以此為依據去建立 idx2char char2idx 的dictionary，以方便之後再做字母轉idx , idx 轉字母的行為，以及根據不同的四個時態建立對應的dictionary.

```
class MyDataLoader:
    def __init__ (self, path, train):

        #build dictionary
        self.char2idx = self.build("c2i")
        self.idx2char = self.build("i2c")
        self.tense2idx= {'sp':0, 'tp':1, 'pg':2, 'p':3 }
        self.idx2tense= { 0:'sp', 1:'tp', 2:'pg', 3:'p'}
        self.train = train

        self.words, self.tenses = self.get_dataset(path,self.train)

        assert len(self.words)==len(self.tenses),'word list is not compatible with tense list'

    def build(self,type):

        if(type=="c2i"):
            #first two
            dictionary={'SOS':0,'EOS':1}
            #alhpabets
            dictionary.update([(chr(i+97),i+2) for i in range(0,26)])
            return dictionary
        if(type=="i2c"):
            #first two
            dictionary={0:'SOS',1:'EOS'}
            #alphabets
            dictionary.update([(i+2,chr(i+97)) for i in range(0,26)])
        return dictionary
```

並依據readme告知讀入的資料是train.txt以及test.txt的時態是不同的序列，而去做不同的行為的資料讀取。train.txt是依據sp, tp,pg,p的順序建立，test.txt則不規則，則依據readme的時態去取tense。

```
    def get_dataset(self,path,train):
        words=[]
        tenses=[]
        with open(path,'r') as file:
            if train:
                #Deal with train.txt
                for line in file:
                    words.extend(line.split('\n')[0].split(' '))
                    # with the tense order
                    tenses.extend(range(0,4))
            else:
                #Deal with test.txt
                for line in file:
                    words.append(line.split('\n')[0].split(' '))
                test_tenses=[['sp','p'],['sp','pg'],['sp','tp'],['sp','tp'],['p','tp'],['sp','pg'],['p','sp'],['pg','sp'],['pg','p'],['pg','tp']]
                for test_tense in test_tenses:
                    tenses.append([self.tense2idx[tense] for tense in test_tense])
        return words,tenses

    def __getitem__(self, idx):

        if self.train:
            # return training data
            return self.string2tensor(self.words[idx],EOS=True),self.tense2tensor(self.tenses[idx])
        else:
            # return testing data and it's tense
            return self.string2tensor(self.words[idx][0],EOS=True),self.tense2tensor(self.tenses[idx][0]),\
                   self.string2tensor(self.words[idx][1],EOS=True),self.tense2tensor(self.tenses[idx][1])
```

text generation is produced by Gaussian noise:
這個部份是由兩個function所構成的，首先generate藉由把1*32的latent vector 及1*8的condtion vector 給concate起來後，藉由decorder去產生對應的對應的predict 。

```python
def generate(self,latent,tense):
    tense_tensor=torch.tensor([tense]).to(device)
    #convert to dim = 8
    c=self.tense_embedding(tense_tensor).view(1, 1, -1)
    #concate
    decoder_hidden = self.latentcondition2hidden(torch.cat((latent, c), dim=-1))
    decoder_cell=self.decoder.init_c0()
    decoder_input = torch.tensor([[SOS_token]], device=device)

    predict_output = None
    for di in range(self.max_length):
        output, decoder_hidden, decoder_cell = self.decoder(decoder_input, decoder_hidden,decoder_cell)
        predict_class = output.topk(1)[1]
        predict_output = torch.cat((predict_output, predict_class)) if predict_output is not None else predict_class

        if predict_class.item() == EOS_token:
            break
        decoder_input = predict_class

    return predict_output
```

而這個latent vector是由Gaussian_generate透過torch.randn隨機生成的，根據pytorch document得到是sample from N(0,1)，也就是藉由高斯雜訊所產生，接著再依據這個高斯雜訊生成的latent vector去產生4個不同時態的字來作為Gassian score的判斷標準。

## TORCH.RANDN

`torch.randn(*size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)` → Tensor

Returns a tensor filled with random numbers from a normal distribution with mean *0* and variance *1* (also called the standard normal distribution).

$$\text{out}_i \sim \mathcal{N}(0, 1)$$

```python
def Gaussian_generate(CVAE,latent_size,tensor2string):
    CVAE.eval()
    words=[]

    with torch.no_grad():
        # 100 words with 4 tenses
        for i in range(100):
            latent = torch.randn(1, 1, latent_size).to(device)
            line = []
            for tense in range(4):
                word = tensor2string(CVAE.generate(latent, tense))
                line.append(word)

            words.append(line)
    return words
```

Hyperparameters:
input_size = 28
hidden_size = 256
latent_size = 32
conditional_size = 8
lr = 0.05
epochs = 250
Teacher forcing ratio: 從1開始，隨著epochs數線性遞減 到 0
KL weight :

cyclic: 前半從0開始用epochs/2 以sigmoid遞增，後半再從0開始 用剩下的epochs/2 再sigmoid遞增
linear:從0開始根據epochs線性遞增
cyclic跟linear都分別做了遞增到 1 、0.3、0.1的版本，0.1的版本使cyclic真正作用。

• Results and discussion (25%)

Bleu test result: 0.71 (把main中的load改成1做evaluation)

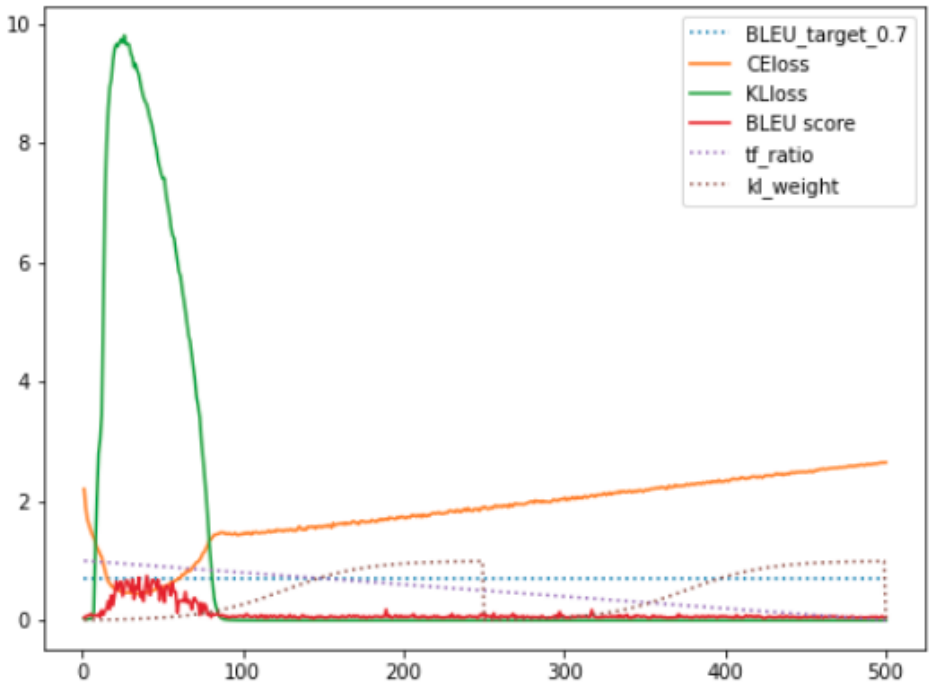
```
input: split
target: splitting
prediction: splitting

input: flared
target: flare
prediction: flatten

input: functioning
target: function
prediction: function

input: functioning
target: functioned
prediction: functioned

input: healing
target: heals
prediction: heaves

avg BLEUscore 0.71
```

Gaussian test result: 0.43

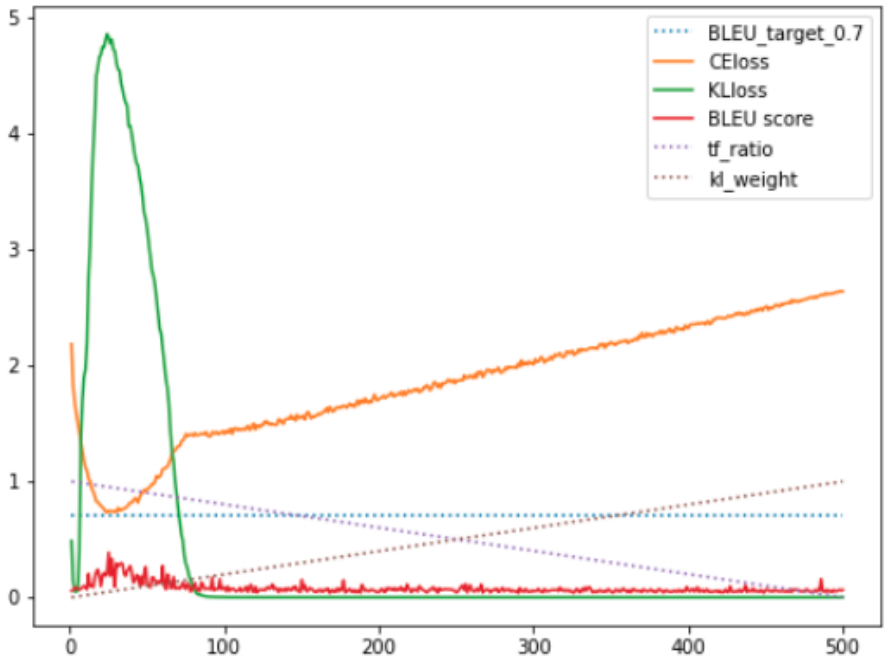
```
['snuggle', 'snuggles', 'snuggling', 'snuggled'], ['penetrate', 'penetrates', 'penetrating', 'penetrated'], ['exile', 'en
vies', 'exiling', 'exiled'], ['harass', 'harasses', 'harassing', 'harassed'], ['father', 'fattens', 'fattening', 'fattene
d'], ['suspect', 'suspects', 'sucking', 'summoned'], ['function', 'flees', 'functioning', 'functioned'], ['frown', 'forew
arns', 'foregoing', 'forewarned'], ['allay', 'allays', 'allaying', 'allayed'], ['command', 'coasts', 'commanding', 'comma
nded'], ['nod', 'nods', 'nodding', 'nodded'], ['convene', 'convenes', 'convening', 'convened'], ['revolve', 'revolves',
'revolving', 'revolved'], ['fash', 'fashions', 'fashioning', 'fashioned'], ['survey', 'surveys', 'surviving', 'surged'],
['resound', 'escapes', 'escaping', 'escaped'], ['forewarn', 'forewarns', 'forewarning', 'forewarned'], ['sift', 'sings',
'sifting', 'sifted'], ['flash', 'flashes', 'flashing', 'flashed'], ['resign', 'resigns', 'resigning', 'reminded'], ['bac
k', 'backslides', 'backing', 'backed'], ['toss', 'testifies', 'testifying', 'tossed'], ['hot', 'hots', 'horsewhiping', 'h
ot']]

avg Gaussianscore 0.43
```

KL-weight 遞增到1的結果

KL-weight , 用sigmoid做cyclic遞增到1兩次



KL-weight , linear遞增到1



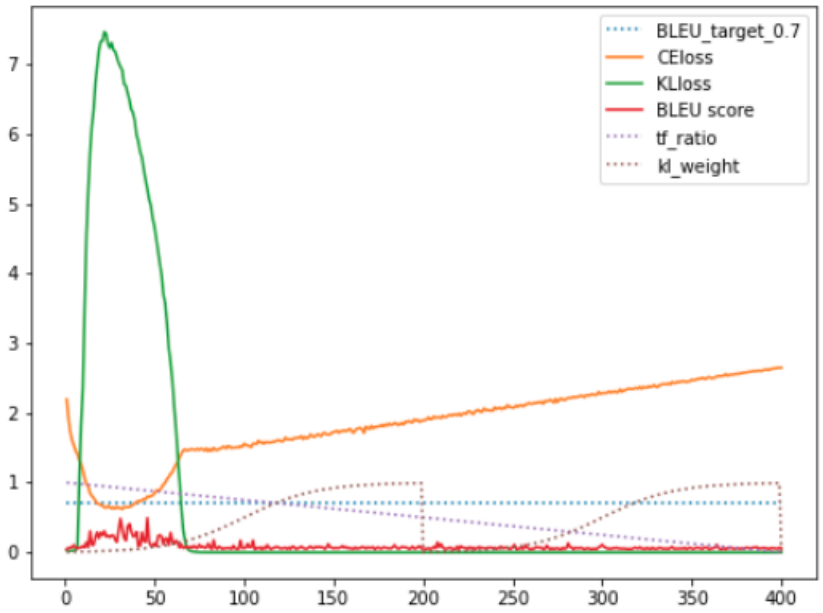

1.可以發現前期的時候因為KL-weight很小，因此前期主宰Loss的是CEloss，所以前期CEloss有效的被降低了，但也因為KL-weight小的關係KLloss上升了。

2.因為CEloss控制的是input of encoder跟 out of decoder的接近程度，所以可以發現降低CEloss，代表讓兩者越接近的時候可以有效提高Bleu score，可以發現大約CEloss小於1的時候Bleu表現會變好。

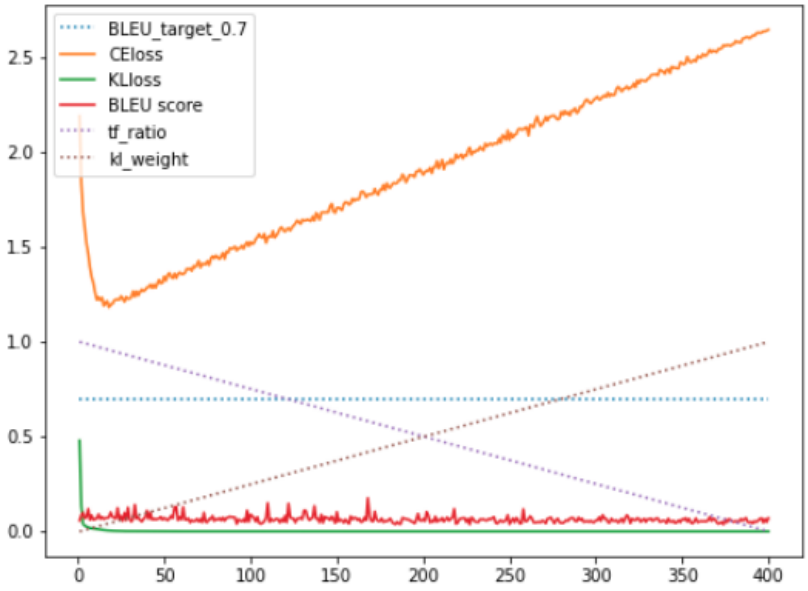
3.但直到KL-weight越來越大後導致KL見見主宰整個loss function，因此為了降低Loss所以得把KLloss給降低，但網路花越多心力去把KL降低時會導致CEloss沒有辦法這麼有效被限制住，所以可以從圖上看到當KL降低時CEloss開始逐漸提高超過1，且Bleu的分數開始變得很差。

4.可以發現KL weight遞增到1還是太大了，可以看到圖上在訓練的前期KL大幅降低後，剩下訓練大部分的時間都在讓網路學會讓KLloss很小，且CEloss可以稍微大的這種走向，所以即便是cyclic的版本時，在中期ecpohs=250時已經把kl-weight縮成零了，也調整不回來CEloss跟KLloss的距離了。

KL-weight遞增到1 但epochs數減少為400
KL-weight sigmoid遞增到1 , cyclic , epochs 500 -> 400

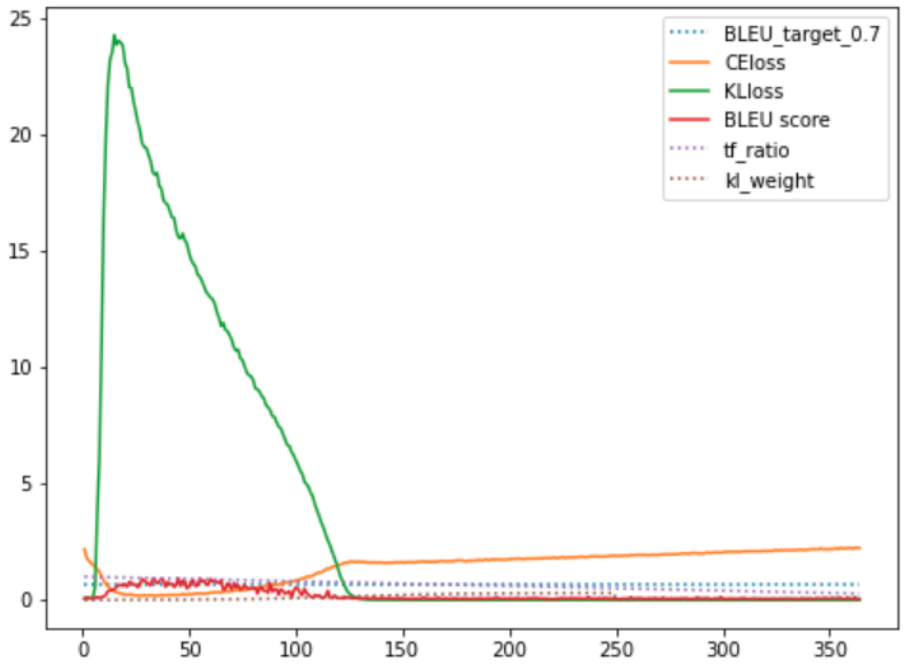



KL-weight linear 遞增到1, epochs 500 -> 400

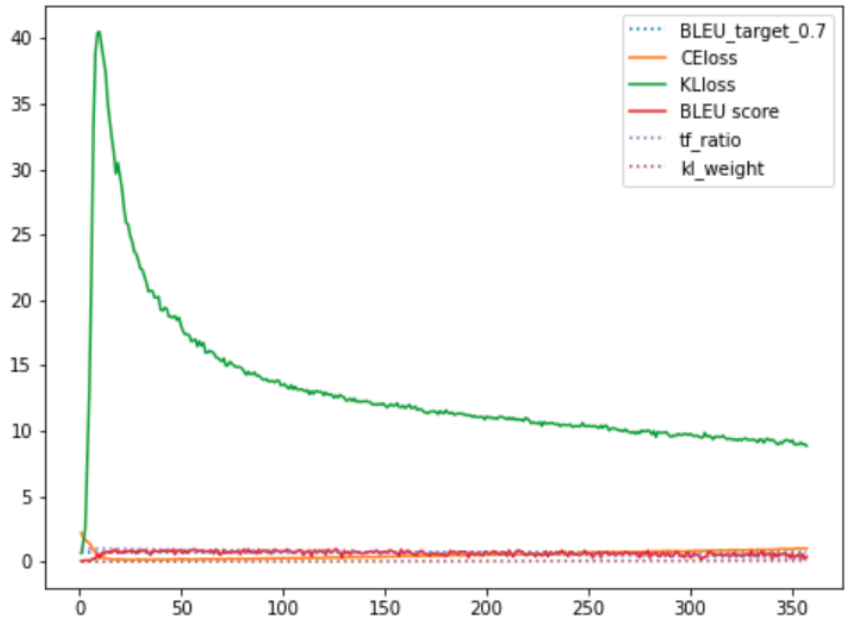



對epochs數目減少的情況下觀察，可以很明顯看到cyclic的版本KL loss有先上升後再下降，即便epochs數減少了，但是因為cyclic的遞增是用sigmoid的遞增，所以前期KL-weight上升會比linear的慢一點，所以這時候的KL-weight還是足夠小，能夠限制CEloss的，因此這時候的Bleu的分數還是有變好。但很明顯的看到linear遞增的版本，因為epochs數目減少了，上升的速度會比500的再快一點（500/400=1.25倍），但這個速度很明顯太快了，讓CE loss還來不及下降，KL就已經下降到底了，所以導致CEloss沒有辦法被降低，而Bleu score的表現很糟糕。

KL-weight 遞增到0.3的結果:
KL-weight sigmoid遞增到0.3 , cyclic

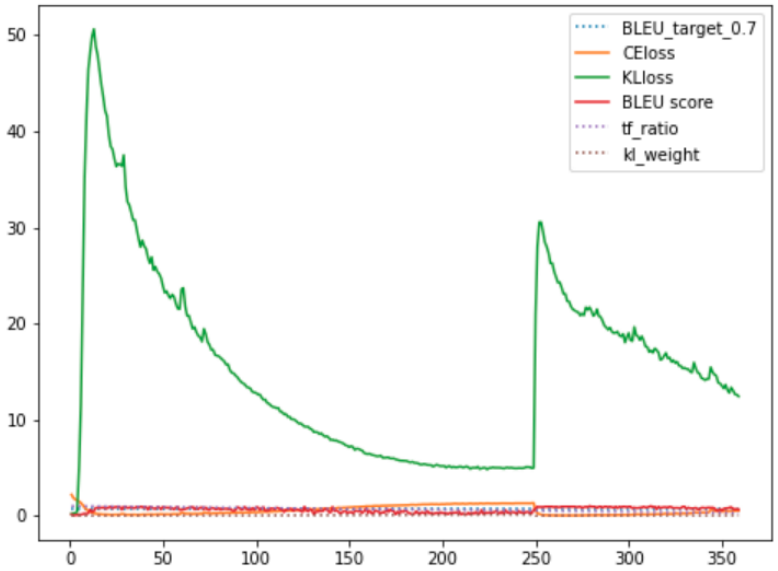



KL-weight linear 遞增到0.3

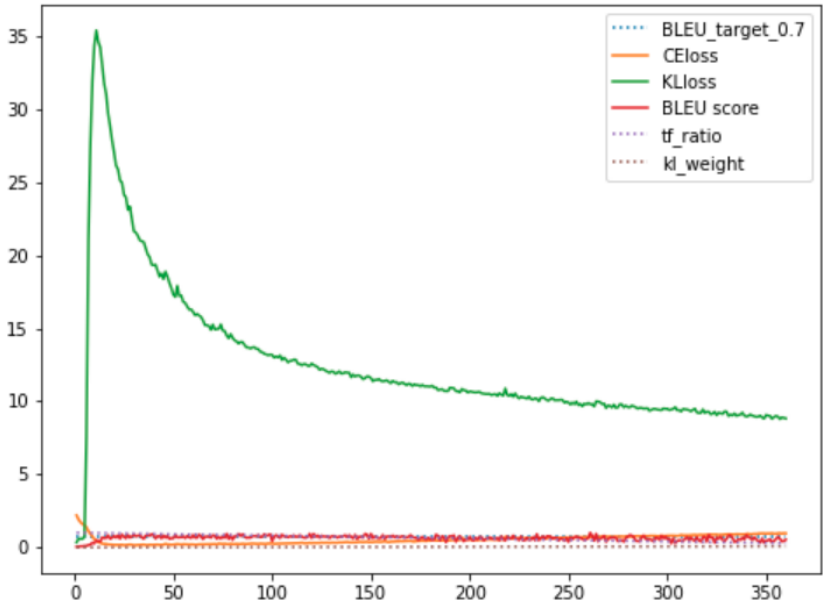



降低kl-weight後可以發現，cyclic的部份Bleu表現好的部份變成100多個epochs了，且KLloss起初的時候更高，可以看到當KLloss大 CEloss小的前面120個epoch,Bleu的表現都比較好，但到面KL-weight又太大讓KLloss直接逼近零後，CEloss開始大過1，Bleu score就降低了，而這邊cyclic的問題跟上面的一樣，在250的時候KLloss已經逼近零了，所以沒辦法再改變它們的距離。而反觀linear的KL-weight更小一些，所以可以看到KLloss沒有逼近零在12左右，CEloss也沒有超過1開始往上遞增，維持著很好的平衡，所以這邊的Bleuscore都很不錯(藍色虛線是0.7)。

KL-weight 遞增到0.1的結果:
KL-weight sigmoid遞增到0.1，cyclic




KL-weight linear 遞增到0.1




把KL-weight再縮的更小到0.1，這邊linear跟上面到0.3的一樣，值得注意的是cyclic的版本，由於KL-weight更小了，因此KLloss下降的速度比較慢，可以看到它在epochs=250時 KLloss還沒有降到0，所以這時候縮小KL-weight可以有效制止KLloss直接到0，CEloss開始遞增這件事，所以可以看到KLloss再次升高，CEloss再次下降，Bleu score也再次提高了。