A Appendix

A.1 Word Dropout as a Special Case

Here, we derive word dropout as an instance of our framework. First, let us introduce a new token, $\langle\text{null}\rangle$, into both the source vocabulary and the target vocabulary. $\langle\text{null}\rangle$ has the embedding of a all-0 vector and is never trained. For a sequence x of words in a vocabulary with $\langle\text{null}\rangle$, we define the *neighborhood* $N(x)$ to be:

$$N(x) = \left\{x' : \left|x'\right| = |x| \text{ and } x'_i \in \{x_i, \langle\text{null}\rangle\}\right\}$$

In other words, $N(x)$ consists of x and all the sentences obtained by replacing a few words in x by $\langle\text{null}\rangle$. Clearly, all augmented sentences $\widehat{x}$ that are sampled from x using word dropout fall into $N(x)$.

In (4), the augmentation policy $\mathbf{q}^*(\widehat{x}, \widehat{y}|x, y)$ was decomposed into two independent terms, one of which samples the augmented source sentence $\widehat{x}$ and the other samples the augmented target sentence $\widehat{y}$

$$\mathbf{q}^*(\widehat{x}, \widehat{y}|x, y) = \underbrace{\frac{\exp\{r_x(\widehat{x}, x)/\tau_x\}}{\sum_{\widehat{x}'}\exp\{r_x(\widehat{x}', x)/\tau_x\}}}_{\mathbf{q}(\widehat{x}|x)} \times \underbrace{\frac{\exp\{r_y(\widehat{y}, y)/\tau_y\}}{\sum_{\widehat{y}'}\exp\{r_y(\widehat{y}', y)/\tau_y\}}}_{\mathbf{q}(\widehat{y}|y)}$$

Word dropout is an instance of this decomposition, where r_y takes the same form with r_x, given by:

$$r_x(\widehat{x}, x) = \begin{cases} -\text{HammingDistance}(\widehat{x}, x) & \text{if } \widehat{x} \in N(x) \\ -\infty & \text{otherwise} \end{cases}, \tag{5}$$

where $\text{HammingDistance}(\widehat{x}, x) = \sum_{i=1}^{|x|} \mathbf{1}[\widehat{x}_i \neq x_i]$. To see this is indeed the case, let h be the Hamming distance for $\widehat{x} \in N(x)$ and set $\lambda_{\text{word}} = \exp\{-1/\tau_x\}$, then we have:

$$\exp\{r_x(\widehat{x}, x)/\tau_x\} = \exp\{-h/\tau_x\} = \exp\left\{-h \cdot \log\frac{1}{\lambda_{\text{word}}}\right\} = \exp\{h \cdot \log\lambda_{\text{word}}\} = {\lambda_{\text{word}}}^h, \tag{6}$$

which is precisely the probability of dropping out h words in x, where each word is dropped with the distribution $\text{Bernoulli}(\lambda_{\text{word}})$.

The difference between word dropout and SwitchOut comes in the fact that $N(x)$ is much smaller than the support of $\widehat{x}$ that SwitchOut can sample from, which is $V^{|x|}$ where V is the vocabulary. Word dropout concentrates all augmentation probability mass into $N(x)$ while SwitchOut spreads the mass into a larger support, leading to a larger entropy. Meanwhile, both word dropout and SwitchOut are exponentially less likely to diverge a way from x, ensuring the smoothness desiderata of a good data augmentation policy, as we discussed in Section 2.3.

A.2 RAML as a Special Case

Here, we present a detailed description of how RAML is a special case of our proposed framework. For each empirical observation $(x, y) \sim \widehat{\mathbf{p}}$, RAML defines a reward aware target distribution $\mathbf{p}_{\text{RAML}}(Y|x, y)$ for the model distribution $\mathbf{p}_\theta(Y \mid x)$ to match. Concretely, the target distribution in RAML has the form

$$\mathbf{p}_{\text{RAML}}(\widehat{y}|x, y) = \frac{\exp\{r(\widehat{y}; y)/\tau\}}{\sum_{\widehat{y}'}\exp\{r(\widehat{y}'; y)/\tau\}},$$

where r is the task reward function. With this definition, RAML amounts to minimizing the expected KL divergence between $\mathbf{p}_{\text{RAML}}$ and $\mathbf{p}_\theta$, *i.e.*

$$\begin{aligned} & \min_\theta \mathbb{E}_{x,y\sim\widehat{\mathbf{p}}}\left[\text{KL}(\mathbf{p}_{\text{RAML}}(Y|x, y)\|\mathbf{p}_\theta(Y \mid x)\right] \\ \iff & \max_\theta \mathbb{E}_{x,y\sim\widehat{\mathbf{p}}}\left[\mathbb{E}_{\widehat{y}\sim\mathbf{p}_{\text{RAML}}(Y|x,y)}\left[\log\mathbf{p}_\theta(\widehat{y} \mid x)\right]\right] \\ \iff & \max_\theta \mathbb{E}_{\widehat{y}\sim\mathbf{p}_{\text{RAML}}(Y)}\left[\log\mathbf{p}_\theta(\widehat{y} \mid x)\right], \end{aligned}$$

where $\mathbf{p}_{\text{RAML}}(Y)$ is the marginalized target distribution, *i.e.* $\mathbf{p}_{\text{RAML}}(Y) = \mathbb{E}_{x,y\sim\widehat{\mathbf{p}}}\left[\mathbf{p}_{\text{RAML}}(Y|x,y)\right]$. Now, notice that $\mathbf{p}_{\text{RAML}}(Y)$ is a member of the augmentation distribution family in consideration (*c.f.* Section 2.2). Specifically, it is equivalent to a data augmentation distribution where

$$
\begin{aligned}
& \mathbf{q}(\widehat{x},\widehat{y}\mid x,y) = \mathbf{1}[\widehat{x}=x]\cdot\mathbf{p}_{\text{RAML}}(\widehat{y}|x,y) \\
\iff \quad & \frac{\exp\left\{s(\widehat{x},\widehat{y};x,y)/\tau\right\}}{\sum_{\widehat{x}',\widehat{y}'}\exp\left\{s(\widehat{x}',\widehat{y}';x,y)/\tau\right\}} = \mathbf{1}[\widehat{x}=x]\cdot\frac{\exp\left\{r(\widehat{y};y)/\tau\right\}}{\sum_{\widehat{y}'}\exp\left\{r(\widehat{y}';y)/\tau\right\}} \\
\iff \quad & s(\widehat{x},\widehat{y};x,y) = \begin{cases} r(\widehat{y};y), & \widehat{x}=x \\ -\infty, & \widehat{x}\neq x \end{cases}.
\end{aligned} \tag{7}
$$

The last equality reveals an immediate connection between RAML and our proposed framework. In summary, RAML can be seen as a special case of our data augmentation framework, where the similarity function is defined by (7). Practically, this means RAML only consider pairs with source sentences from the empirical set for data augmentation.

A.3 Datasets Descriptions

	vocab (K)		**#sents**		
	src	**tgt**	**train**	**dev**	**test**
en-vi	17.2	7.7	133.3K	1.6K	1.3K
de-en	32.0	22.8	153.3K	7.0K	6.8K
en-de	50.0	50.0	4.5M	2.7K	2.2K

Table 3: Statistics of the datasets.

Table 3 summarizes the statistics of the datasets in our experiments. The WMT 15 en-de dataset is one order of magnitude larger than the IWSLT 16 de-en dataset and the IWSLT 15 en-vi dataset. For the en-vi task, we use the data pre-processed by Luong and Manning (2015). For the en-de task, we use the data pre-processed by Luong et al. (2015), with *newstest2014* for validation and *newstest2015* for testing. For the de-en task, we use the data pre-processed by Ranzato et al. (2016).

A.4 Hyper-parameters

Task	n_{layers}	n_{heads}	$d_{\text{k}}, d_{\text{v}}$	d_{model}	d_{inner}	init	clip	λ_{drop}	τ_x^{-1}	τ_y^{-1}	λ_{word}
en-de	8	6	64	512	1024	0.04	25.0	0.10	1.00	0.80	0.1
de-en	8	5	64	288	507	0.035	25.0	0.25	0.95	0.90	0.1
en-vi	4	4	64	256	384	0.035	20.0	0.15	1.00	0.90	0.1

Table 4: Hyper-parameters for our experiments.

The hyper-parameters used in our experiments are in Table 4. All models are initialized uniformly at random in the range as reported in Table 4. All models are trained with Adam (Kingma and Ba, 2015). Gradients are clipped at the threshold as specified in Table 4. For the WMT en-de task, we use the legacy learning rate schedule as specified by Vaswani et al. (2017). For the de-en task and the en-vi task, the learning rate is initially 0.001, and is decreased by a factor of 0.97 for every 1000 steps, starting at step 8000. All models are trained for 100,000 steps, during which one checkpoint is saved for each 2500 steps and the final evaluation is performed on the checkpoint with lowest perplexity on the dev set.

Multiple GPUs are used for each experiment. For the de-en and the en-vi experiments, if we use n GPUs, where $n \in \{1, 2, 4\}$, then we only perform $10^5/n$ updates to the models' parameters. We find that this is sufficient to make the models converge.

A.5 Source Code for Sampling in TensorFlow

Hamming distance sampling in TensorFlow

```tensorflow
import tensorflow as tf
def hamming_distance_sample(sents, tau, bos_id, eos_id, pad_id, vocab_size):
  """Sample a batch of corrupted examples from sents.

  Args:
    sents: Tensor [batch_size, n_steps]. The input sentences.
    tau: temperature.
    vocab_size: to create valid samples.

  Returns:
    sents: Tensor [batch_size, n_steps]. The corrupted sentences.
  """

  # mask
  mask = [
      tf.equal(sents, bos_id),
      tf.equal(sents, eos_id),
      tf.equal(sents, pad_id),
  ]
  mask = tf.stack(mask, axis=0)
  mask = tf.reduce_any(mask, axis=0)

  # first, sample the number of words to corrupt for each sentence
  batch_size, n_steps = tf.unstack(tf.shape(sents))
  logits = -tf.range(tf.to_float(n_steps), dtype=tf.float32) * tau
  logits = tf.expand_dims(logits, axis=0)
  logits = tf.tile(logits, [batch_size, 1])
  logits = tf.where(mask,
                    x=tf.fill([batch_size, n_steps], -float("inf")), y=logits)

  # sample the number of words to corrupt at each sentence
  num_words = tf.multinomial(logits, num_samples=1)
  num_words = tf.reshape(num_words, [batch_size])
  num_words = tf.to_float(num_words)

  # <bos> and <eos> should never be replaced!
  lengths = tf.reduce_sum(1.0 - tf.to_float(mask), axis=1)

  # sample corrupted positions
  probs = num_words / lengths
  probs = tf.expand_dims(probs, axis=1)
  probs = tf.tile(probs, [1, n_steps])
  probs = tf.where(mask, x=tf.zeros_like(probs), y=probs)
  bernoulli = tf.distributions.Bernoulli(probs=probs, dtype=tf.int32)

  pos = bernoulli.sample()
  pos = tf.cast(pos, tf.bool)

  # sample the corrupted values
  val = tf.random_uniform(
      [batch_size, n_steps], minval=1, maxval=vocab_size, dtype=tf.int32)
  val = tf.where(pos, x=val, y=tf.zeros_like(val))
  sents = tf.mod(sents + val, vocab_size)

  return sents
```

A.6 Source Code for Sampling in PyTorch

Hamming distance sampling in Pytorch

```pytorch
  """
  Sample a batch of corrupted examples from sents.

  Args:
      sents: Tensor [batch_size, n_steps]. The input sentences.
```

```
        tau: Temperature.
        vocab_size: to create valid samples.
    Returns:
        sampled_sents: Tensor [batch_size, n_steps]. The corrupted sentences.
    """

    mask = torch.eq(sents, bos_id) | torch.eq(sents, eos_id) | torch.eq(sents, pad_id)
    lengths = mask.float().sum(dim=1)
    batch_size, n_steps = sents.size()
    # first, sample the number of words to corrupt for each sentence
    logits = torch.arange(n_steps)
    logits = logits.mul_(-1).unsqueeze(0).expand_as(
      sents).contiguous().masked_fill_(mask, -float("inf"))
    logits = Variable(logits)
    probs = torch.nn.functional.softmax(logits.mul_(tau), dim=1)
    num_words = torch.distributions.Categorical(probs).sample()

    # sample the corrupted positions.
    corrupt_pos = num_words.data.float().div_(lengths).unsqueeze(
      1).expand_as(sents).contiguous().masked_fill_(mask, 0)
    corrupt_pos = torch.bernoulli(corrupt_pos, out=corrupt_pos).byte()
    total_words = int(corrupt_pos.sum())
    # sample the corrupted values, which will be added to sents
    corrupt_val = torch.LongTensor(total_words)
    corrupt_val = corrupt_val.random_(1, vocab_size)
    corrupts = torch.zeros(batch_size, n_steps).long()
    corrupts = corrupts.masked_scatter_(corrupt_pos, corrupt_val)
    sampled_sents = sents.add(Variable(corrupts)).remainder_(vocab_size)

    return sampled_sents
```