18 Algorithms for MT 2: Parameter Optimization Methods

In this chapter we re-visit the problem of optimizing our parameters for sequence-to-sequence models.

18.1 Error Functions and Error Minimization

Up until this point, most of the models we have encountered have been learned using some variety of maximum likelihood estimation. However, when actually using a translation model, we aren't interested in how much probability the model gives to good translations, but whether the translation that it generates is actually good or not. Thus, we would like a method that tunes the parameters of a machine translation system to *actually increase translation accuracy*.

To state this formally, we know that our system will be generating a translation

$$\hat{E} = \operatorname*{argmax}_{\tilde{E}} P(\tilde{E} \mid F). \tag{168}$$

Given a corpus of translations $\hat{\mathcal{E}}$ and references $\mathcal{E}$, we can calculate an **error function**

$$\text{error}(\mathcal{E}, \hat{\mathcal{E}}). \tag{169}$$

The error function is a measure of how *bad* the translation is, and is often chosen to be something like $1 - \text{BLEU}(\mathcal{E}, \hat{\mathcal{E}})$ for translation, or whatever other appropriate measure we can come up with for the task at hand. Thus, instead of training the parameters to maximize the likelihood, we would like to train the parameters to minimize this error, improving the quality of the results generated by our model.

However, directly optimizing this error function is difficult for a couple of reasons. The first reason is that there are a *myriad of possible translations* $\hat{\mathcal{E}}$ that the system could produce depending on what parameters we choose. It is generally not feasible to enumerate all these possible outputs, so it is necessary to come up with a method that allows us to work over a subset of the actual potential translations. The second reason why direct error minimization is difficult is because the argmax function in Equation 168, and by corollary the error function in Equation 169, is *not continuous*. The result of the argmax will not change unless the highest-scoring hypothesis changes, and thus tiny changes in the parameters will often not make a difference in the error because they don't result in the change in the most probable hypothesis. As a result, the error function is **piecewise constant**, in most places its gradient is zero, and in some places (where the best-scoring hypothesis suddenly changes), its gradient is undefined. Readers with good memory will remember that the step function in Section 5.3 had the exact same problem, which made it difficult to optimize.

In order to overcome these difficulties, there are a number of methods to *approximate the hypothesis space* and create more *easily calculable loss functions*, which we describe in the following sections.

18.2 Minimum Error Rate Training

One example of a method that makes it computationally feasible to minimize the error for arbitrary evaluation measures is the minimum error rate training (MERT) framework of [13]. This gets around the problems stated above in three ways: (1) it assumes that we are dealing

with a linear model where the scores of hypotheses are the linear combination of multiple features, like the log-linear models described in Section 4 or Section 14.6, (2) it works over only a subset of the hypotheses that can be produced by a translation system, and (3) it uses an efficient line-search method to iteratively find the best value for a single parameter for each parameter to be optimized. We will give a conceptual overview of the procedure here.

To re-iterate, this method is concerned with linear models, which express the probability of a sentence according to a linear combination of feature values:[55]

$$\begin{aligned}\log P(F,E) &\propto S(F,E),\\ &= \sum_i \lambda_i \phi_i(F,E),\\ &= \boldsymbol{\lambda}\cdot\boldsymbol{\phi}_i(F,E),\end{aligned} \tag{170}$$

where $S(F,E)$ is a function expressing a score proportional to the log probability. At the beginning of the procedure, we start with an initial set of weights $\boldsymbol{\lambda}$ for our linear model, initialized to some value (for example $\lambda = 1$ for all values). Given source and target training corpora $\mathcal{F}$ and $\mathcal{E}$, we perform an iterative procedure (the *outer loop*) of:

Generating hypotheses: For source corpus $\mathcal{F}$, we generate n-best hypotheses $\hat{\mathcal{E}}$ according to the current value of $\boldsymbol{\lambda}$. This hypothesis generation step can be done using beam search, as has been covered in previous sections. For the ith sentence in $\mathcal{F}$, F_i, we will express the n-best list as $\hat{\boldsymbol{E}}_i$ and the jth hypothesis in this n-best list as $\hat{E}_{i,j}$.

Adjusting parameters: We start with our initial estimate of $\boldsymbol{\lambda}$, and try to adjust it to reduce the error. To define this formally, we first define $\hat{\mathcal{E}}^{(\boldsymbol{\lambda})}$ to be the highest-scoring hypothesis for each of the sentences in the corpus given $\boldsymbol{\lambda}$:

$$\hat{\mathcal{E}}^{(\boldsymbol{\lambda})} = \{\hat{E}_1^{(\boldsymbol{\lambda})}, \hat{E}_2^{(\boldsymbol{\lambda})}, \ldots, \hat{E}_{|\mathcal{E}|}^{(\boldsymbol{\lambda})}\} \tag{171}$$

where

$$\hat{E}_i^{(\boldsymbol{\lambda})} = \operatorname*{argmax}_{\tilde{E}\in\hat{\boldsymbol{E}}_i} S(F_i, \tilde{E}; \boldsymbol{\lambda}). \tag{172}$$

Then, we attempt to find the lambda that minimizes our error:

$$\hat{\boldsymbol{\lambda}} = \operatorname*{argmin}_{\boldsymbol{\lambda}} \text{error}(\mathcal{E}, \hat{\mathcal{E}}^{(\boldsymbol{\lambda})}). \tag{173}$$

Because hypotheses can be generated using standard beam search, the main difficulty in MERT is how to go from our n-best list and initial parameters to $\hat{\boldsymbol{\lambda}}$. [13]'s method for MERT proposes an elegant solution using **line search**, which explores all the possible parameter vectors $\boldsymbol{\lambda}$ that fall along a particular line in parameter space, and finding the parameters that minimize the error along this line. This second iterative process (the *inner loop*) consists of the following two steps:

[55] More precisely, this equation would include derivation D as noted before, but we omit it here for conciseness of notation.

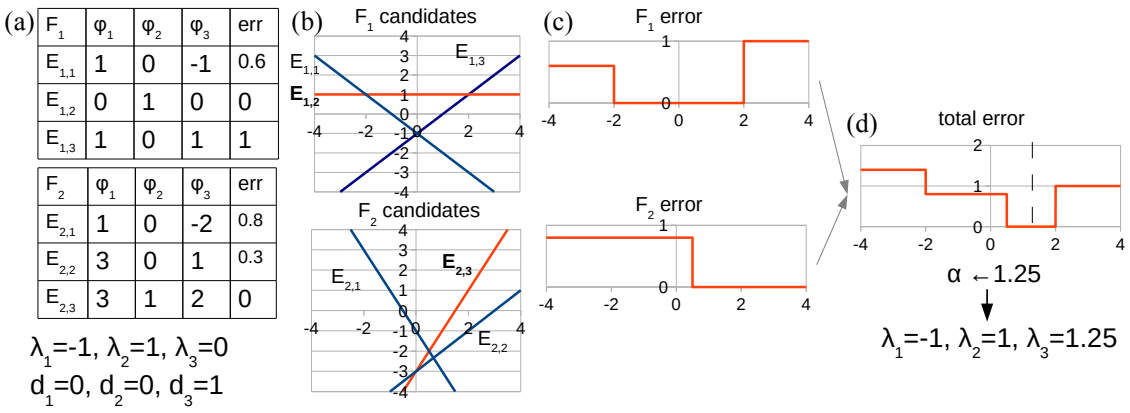

Figure 56: An example of line search in minimum error rate training (MERT).

Picking a direction: We pick a direction in parameter space to explore, expressed by a vector $\boldsymbol{d}$ of equal size as the parameter vector. Some options for this vector include a one-hot vector, where a single parameter is given a value of 1 and the rest are given a value of zero, a random vector, or a vector calculated according to gradient-based methods such as the minimum-risk method described in Section 18.3 [4].

Finding the optimal point along this direction: We then perform a line search, detailed below to find the parameters along this direction that minimize the error. Formally, this can be thought of defining new parameters $\boldsymbol{\lambda}_\alpha := \boldsymbol{\lambda} + \alpha\boldsymbol{d}$, and finding the optimal α:

$$\hat{\boldsymbol{\lambda}_\alpha} = \operatorname*{argmin}_\alpha \text{error}(\mathcal{E}, \hat{\mathcal{E}}^{(\boldsymbol{\lambda}_\alpha)}). \tag{174}$$

We then update $\boldsymbol{\lambda} \leftarrow \hat{\boldsymbol{\lambda}}_\alpha$ and repeat the loop until no further improvements in error can be made.

Figure 56 demonstrates this procedure on two questions with answers and their corresponding features in Figure 56 (a). First note that for any hypothesis $\hat{E}_{i,j}$ for source F_i, the score $\boldsymbol{\lambda}_\alpha \cdot \boldsymbol{\phi}(F_i, \hat{E}_{i,j})$ can be decomposed into the part affected by $\boldsymbol{\lambda}$ and the part affected by $\alpha\boldsymbol{d}$:

$$\begin{aligned} S(F_i, E_{i,j}) &= (\boldsymbol{\lambda} + \alpha\boldsymbol{d}) \cdot \boldsymbol{\phi}(F_i, E_{i,j}) \\ &= \boldsymbol{\lambda} \cdot \boldsymbol{\phi}(F_i, E_{i,j}) + \alpha(\boldsymbol{d} \cdot \boldsymbol{\phi}(F_i, E_{i,j})) \\ &= b + c\alpha. \end{aligned} \tag{175}$$

The final equation in this sequence emphasizes that the score can be thought of as a line, where b is the intercept, c is the slope, and α is the variable that defines the x-axis.

Figure 56 (b) plots Equation 175 as in this linear equation form. These plots demonstrate for which values of α candidate $E_{i,j}$ will be assigned a particular score, with the highest line being the one chosen by the system. For F_1, the chosen answer will be $\hat{E}_{1,1}$ for $\alpha < -2$, to $\hat{E}_{1,2}$ for $-2 < \alpha < 2$, and to $\hat{E}_{1,3}$ for $2 < \alpha$ as indicated by the range where the answer's corresponding line is greater than the others. These ranges can be found by a simple algorithm

called a **line sweep algorithm**, which sorts the lines in order of ascending slope, and process them one by one, finding where this line and the next line intersects.

Next, we take the information of answers chosen in Figure 56 (b) and, given the information about the error incurred by each answer, convert this into a graph as Figure 56 (c). The ranges for each question are then combined into a single graph indicating the total error for each value of α across the entire corpus (Figure 56 (d)). We then choose a point in the center of the region with the minimal error, and uses this as our value of α.

By iteratively performing the outer loop of generating hypotheses, then the inner loop of gradually moving along lines in the direction that will reduce the error, we can effectively find a local optimum in the error surface, even though it is highly discontinuous. There are a few other tricks to MERT that are worth mentioning as well:

Random restarts: One thing to be aware of is that this process is quite prone to falling into local optima, and thus it is common to re-run the process several times (e.g. 10) from independent random starting points in the parameter space, then take the best final point achieved by the several random restarts.

Corpus-level measures: It also should be noted that while the previous example assumed that we were using a sentence-level error, it is also possible to use MERT with corpus level metrics like standard BLEU. This is done by adding up the sufficient statistics used to calculate BLEU (n-gram counts and match counts) across all the sentences, then calculating BLEU after these statistics have been aggregated across the whole corpus.

18.3 Minimum Risk Training

The method described in the previous section allows us to perform search to directly minimize error in linear models. However, this method is not applicable to non-linear models such as neural MT models, and has trouble scaling to large numbers of features due to its non-differentiable nature. **Minimum risk** (MinRisk) training [16, 15] is a method that is very similar to MERT, but has the desirable property that it results in a loss function that is differentiable and conducive to optimization through gradient-based methods.

Specifically, the risk is defined as the expected value of the error according to a probabilistic model, which can be defined as below (assuming that we have a sentence-level measure of error):

$$\text{risk}(F, E, \theta) = \sum_{\tilde{E}} P(\tilde{E} \mid F; \theta)\text{error}(E, \tilde{E}). \tag{176}$$

This objective function looks very similar to the error surface itself, but with smooth transitions between the values. The leftmost graph of Figure 57 demonstrates this for the same error surface that we calculated using MERT, and we can see it contains no jagged transitions caused by abruptly switching from one hypothesis to the next. While this is a simple change – instead of using the argmax we use the sum over all candidates – this function is now differentiable, allowing us to use standard methods such as stochastic gradient descent.

There are a few things to be careful of here though. First, because it is still intractable to sum over the entire space of all possible translations, like MERT, we must approximate the sum in Equation 176. This can be done by choosing a subset of hypotheses S and summing

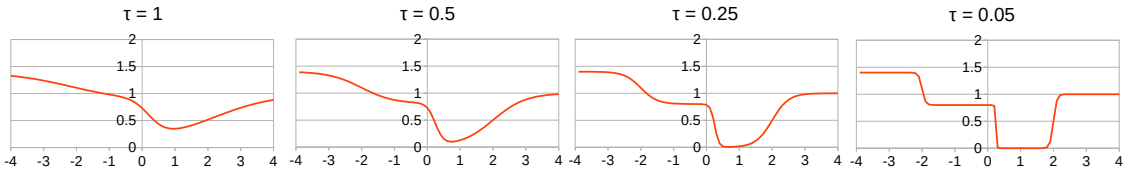

Figure 57: An example of a MinRisk loss surface for different temperature values.

over the hypotheses in this subset. This subset can be obtained either searching for the n-best candidates according to the current model as we did in MERT, or randomly sampling hypotheses. In general, the more hypotheses we have the more accurate our estimate of the gradient will be, but this will also incur a certain amount of computational cost.

The second thing that we should not forget is that MinRisk training is not actually optimizing the error, which is what we are finally interested in. This is obvious by noting that the MinRisk error surface on the left side of Figure 57 is only a very rough approximation of the MERT error surface on the right side of Figure 56. To resolve this difference, it is common for MinRisk training methods introduce a **temperature** parameter τ, which modifies the risk calculation as follows:

$$\text{risk}(F, E, \theta, \tau) = \sum_{\tilde{E}} \frac{P(\tilde{E} \mid F; \theta)^{1/\tau}}{Z} \text{error}(E, \hat{E}), \tag{177}$$

where

$$Z := \sum_{E'} P(E' \mid F; \theta)^{1/\tau}. \tag{178}$$

Put simply, this temperature is modifying the "smoothness" of the probability distribution:

- When $\tau = 1$, this is the regular probability distribution.
- When $\tau > 1$, the distribution becomes "smoother", assigning probability more uniformly across all of the hypotheses in the space. As $\tau \to \infty$, probability will be assigned uniformly across all hypotheses.
- When $\tau < 1$, the distribution becomes "sharper", assigning more probability to the hypotheses with the highest probability in the space. As $\tau \to 0$, all of the probability will be assigned to the one-best hypothesis. At this point, the objective will be equivalent to the MERT objective.

An example of the MinRisk objective at different temperatures is shown in Figure 57, and it can be seen that as τ approaches zero, the distribution becomes more sharp and closer to the error surface.

The question then becomes: how do we choose our temperature τ. In order to do so, it is common to use a strategy called **annealing**, where we gradually decrease τ from a high value to zero as training progresses [16].[56] This allows us to start with a smooth, easy-to-optimize

[56]This term comes from the annealing process in metalworking, where the temperature of molten metal is gradually reduced until it solidifies in the desired shape.

error surface (e.g. on the left of the figure), and gradually progress to the bumpier less easy-to-optimize error surface (e.g. on the left of the figure) that is nonetheless closer to our final objective function.

18.4 Optimization Through Search

Up until now, we talked about how to find parameters that minimize the error or risk with respect to subset of hypotheses S. However, we mustn't forget that in sequence-to-sequence models, simply searching for the highest scoring hypothesis is not an easy task, which necessitated the advanced search techniques such as beam search in Section 7. One other way to optimize our parameters is to explicitly optimize the parameters so we do a good job of searching for the best hypothesis.

Before getting into these specific methods, let us first go over a method called the **structured perceptron** [3, 8], which will give some helpful preliminaries. First, in the context of log-linear models using features $\boldsymbol{\phi}(\cdot)$ and weights $\boldsymbol{\lambda}$, the structured perceptron can be summarized as a simple 2-step process iterated for each training example:

1. Search for the highest scoring hypothesis $\hat{E}$.
2. If this hypothesis is not the reference E, penalize the weights of $\boldsymbol{\phi}(F, \hat{E})$ and boost the weights of $\boldsymbol{\phi}(F, E)$.

This is stated explicitly in Algorithm 8.

Algorithm 8 The structured perceptron for linear featurized models.

```
procedure StructuredPerceptron(F, E)
    for ⟨F, E⟩ ∈ ⟨F, E⟩ do
        Ê = argmax_Ẽ λ · φ(F, Ẽ)
        if Ê ≠ E then
            λ ← λ + φ(F, E) − φ(F, Ê)
        end if
    end for
end procedure
```

An alternative view of the structured perceptron is that it is simply stochastic gradient descent minimizing the following loss function:

$$\ell_{\text{percep}} = S(F, \hat{E}) - S(F, E). \tag{179}$$

We can see this in the log-linear case by taking the derivative:

$$\frac{d\ell_{\text{percep}}}{d\boldsymbol{\lambda}} = \frac{d}{d\boldsymbol{\lambda}} \left(\boldsymbol{\lambda} \cdot \boldsymbol{\phi}(F, E) - \boldsymbol{\lambda} \cdot \boldsymbol{\phi}(F, \hat{E}) \right), \tag{180}$$

$$= \boldsymbol{\phi}(F, E) - \boldsymbol{\phi}(F, \hat{E}). \tag{181}$$

When $E = \hat{E}$, the feature vectors will be equal and thus no update will be performed. Once we see the equivalence between the procedure in Algorithm 8 and SGD using the loss function

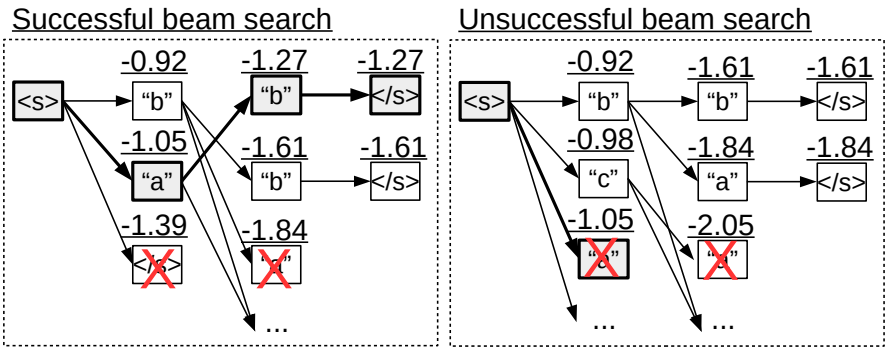

Figure 58: An example of where the best hypothesis “a b” does not fall out of the beam (left) and does (right).

in Equation 179, it becomes clear that we can also use the structured perceptron to optimize any other variety of model (including non-linear neural models) by doing SGD with a similar objective.

Now, moving back to search, there is a variety of the structured perceptron that uses a technique called **early stopping** [3]. This is a method that applies to any method that generates output one step at a time, which is true for both neural models (which generate one word at a time), and symbolic models (which generate ouput one word, phrase, or rule at a time). The concept behind early stopping is very simple: instead of unrolling all the way to the end of the hypothesis $\hat{E}$, the moment we perform an action producing an output that is inconsistent with the reference E. For example, if the word at time-step t fails to be generated properly, then we would perform an update based on the loss function

$$\ell_{\text{early-percep}} = S(F, \hat{e}_1^t) - S(F, e_1^t). \tag{182}$$

This can be seen as penalizing the model as soon as it makes a mistake in search, and thus brings us a step closer to models that consider the search process in parameter optimization.

Finally, consider the case of beam search. When we are performing beam search there is a chance that even at a particular time step the score $S(F, \hat{e}_1^t) > S(F, e_1^t)$, by the time we get to the end of the translation, the reference will have recovered and achieved the highest score. For example, if we re-use the search example of Figure 23, we can see in the left side of Figure 58 that just because the best path has temporarily fallen from the first-place position does not necessarily mean that it will not recover and score the best in the long run.

Search-aware tuning [9] and **beam-search optimization** [22] are two methods that attempt to exploit this fact by adapting methods very similar to the early-stopping perceptron to beam search. Both methods are based on the idea that we want to adjust the score of hypotheses in the intermediate search steps. Search-aware tuning does so by giving a bonus to hypotheses at each time step that get lower error at the end of the search process. Beam-search optimization does so by applying a perceptron-style penalty at each time step where the best hypothesis falls off the beam (i.e., it would apply a penalty at the first step in the example on the right of Figure 58, but not at all in the left example).

18.5 Margin-based Loss-augmented Training

One thing that we should note is that in our description above, we did not explicitly mention the error incurred by each hypothesis, which was an essential element of MERT and MinRisk training. One common way to incorporate this type of information into perceptron-based or search-based algorithms is through the use of a **margin**. Taking the example of the vanilla structured perceptron, the idea of a margin is basically that we not only want the score of the best hypothesis to exceed that of others

$$S(F, E) > S(F, \hat{E}), \tag{183}$$

where $E \neq \hat{E}$), but want it to exceed it by a margin M

$$S(F, E) > S(F, \hat{E}) + M. \tag{184}$$

Forcing the MT system to have a margin of error of M ensures that it will have some breathing room in its predictions, making it more robust in the face of new unseen inputs.

To incorporate the error into this margin, it is common to use **loss-augmented** training, where we further multiply by the margin by the error incurred by that particular hypothesis:

$$S(F, E) > S(F, \hat{E}) + M * \text{err}(E, \hat{E}). \tag{185}$$

This is essentially saying that we need to have a larger margin for very bad hypotheses, and a smaller margin for hypotheses that are only off by a little. A wide variety of methods incorporating this intuition have been proposed for sequence-to-sequence tasks, and have been used with some success [17, 19].

18.6 Optimization as Reinforcement Learning

Finally, another way to think about incorporating loss into optimization for machine translation systems is by treating it as as a form of **reinforcement learning**. Reinforcement learning is a general class of machine learning algorithms where a learner takes a sequence of **actions**, and after a while receives a **reward**. Machine translation fits nicely into this paradigm, as we can view each word selection as an action, and the final evaluation score (e.g. BLEU) of the produced sentence $\text{eval}(E, \hat{E})$ as the reward.

One simple method of reinforcement learning that is now one of the methods of choice for neural MT models is **policy gradient** methods, and specifically the REINFORCE algorithm [21, 14]. This method can be best explained by starting from the maximum likelihood objective and slowly building into the full REINFORCE objective. First, the negative log likelihood loss for a particular reference sentence E is calculated as:

$$\ell_{\text{nll}}(E) = \sum_{t=1}^{|E|} -\log P(e_t | F, e_1^{t-1}). \tag{186}$$

Now let's say we randomly sample a sentence $\hat{E}$ from the current model, we could also optimize our model to maximize the probability of this sentence:

$$\ell_{\text{nll}}(\hat{E}) = \sum_{t=1}^{|E|} -\log P(\hat{e}_t | F, \hat{e}_1^{t-1}). \tag{187}$$

This method is called **self-training** [18, 11], and can be a useful method for semi-supervised learning when we don't have access to the reference E, but if we have access to the reference it seems difficult to argue for the merits of a method that optimizes towards a randomly sampled and potentially erroneous translation instead of the gold reference. However, let's say we make the following change, weighting the objective with the value of the evaluation function, giving us the REINFORCE objective:

$$\ell_{\text{reinforce}}(\hat{E}, E) = \text{eval}(E, \hat{E}) \sum_{t=1}^{|E|} -\log P(\hat{e}_t | F, \hat{e}_1^{t-1}). \tag{188}$$

Now things make a bit more sense; the higher the evaluation score, the stronger we will push the gradients towards the sampled sequence, which means that high-scoring samples will have more effect on the outcome than low-scoring samples.

However, the story is not quite this simple. If we get a really easy sentence where we would normally expect a BLEU score of basically 1.0, we will be very disappointed if we get 0.6, but if we get a hard sentence where the expectation is that we'll be able to get basically nothing right, we may very well be happy if we get a 0.4. To capture this intuition, we can make the addition of a **baseline** function $\text{base}(F, \hat{e}_1^{t-1})$ as follows:

$$\ell_{\text{reinforce+base}}(\hat{E}, E) = \sum_{t=1}^{|E|} -(\text{eval}(E, \hat{E}) - \text{base}(F, \hat{e}_1^{t-1})) \log P(\hat{e}_t | F, \hat{e}_1^{t-1}). \tag{189}$$

This baseline function captures our expectation of how easy the sentence will be to translate, and is trained as a separate model by predicting the value of the evaluation based on the current hidden state using a regression model. This helps reduce the variance of the reward and make training more stable.

The above method for reinforcement learning is called policy-based, as the probability distribution over the next word can be viewed as a probabilistic "policy" for taking the next action. In contrast, **value-based reinforcement learning** tries to predict learn a **value function** or Q **function** $Q(H, a)$, where H is our history of actions up to this point, and a is the next action. In the translation scenario the history is the input sentence and the words we have translated up to this point ($H = \langle F, e_1^{t-1} \rangle$), and the action is the next word ($a = e_t$).

One common way to learn this value function is through **actor-critic** methods [6, 1]. In these methods, we have an *actor*, which is essentially a policy that performs actions and samples a set of actions leading to a translation $\hat{E}$. The *critic* then attempts to predict the final evaluation score of the sentence given the current state and next action, estimating the value function $Q(\cdot)$. At test time, we can then generate translations by calculating the value function $Q(F, e_1^{t-1}, e_t)$ for each word e_t in the vocabulary, and choose the e_t that has the highest value according to this function.

18.7 Further Reading

[12] presents an extensive survey of parameter optimization techniques within the context of symbolic log-linear models.

Evaluation measures for optimization: One thing that was glossed over in the rest of the section was how we decide our evaluation metrics. The default metric for tuning our

systems is BLEU score due to its prevalence in evaluation of systems, but it's possible that tuning with a more effective metric would result in systems that generate results that are better for human consumption [2, 5]. For example, some work has found that using an embedding-based trainable model of semantic similarity between sentences is easier to optimize and often results in better human evaluation results [20].

Efficient data structures and algorithms for optimization: One of the common threads of the optimization algorithms above is that they cannot enumerate the entire space of hypotheses, and thus need to optimize over a subset of n-best hypotheses. There are also efficient algorithms for symbolic models that make it possible to perform optimization over lattices or forests, enumerating a far larger number of hypotheses than can be covered by an n-best list [10, 7].

18.8 Exercise

As an exercise, try to implement minimum risk training for either a symbolic or neural machine translation model. This will involve:

- Implementing code to sample a subset of hypotheses given the current parameters.
- Calculating the loss function using your BLEU calculation code.
- Calculating the gradients of the parameters, and updating them.

References

[1] Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086*, 2016.

[2] Daniel Cer, Christopher Manning, and Daniel Jurafsky. The best lexical metric for phrase-based statistical MT system optimization. In *NAACL HLT*, 2010.

[3] Michael Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1–8, 2002.

[4] Michel Galley, Chris Quirk, Colin Cherry, and Kristina Toutanova. Regularized minimum error rate training. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1948–1959, 2013.

[5] Bushra Jawaid, Amir Kamran, Miloš Stanojević, and Ondřej Bojar. Results of the wmt16 tuning shared task. In *Proceedings of the 1st Conference on Machine Translation (WMT)*, pages 232–238, 2016.

[6] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. volume 13, pages 1008–1014, 1999.

[7] Zhifei Li and Jason Eisner. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 40–51, 2009.

[8] Percy Liang, Alexandre Bouchard-Côté, Dan Klein, and Ben Taskar. An end-to-end discriminative approach to machine translation. In *Proceedings of the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 761–768, 2006.

[9] Lemao Liu and Liang Huang. Search-aware tuning for machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1942–1952, 2014.

[10] Wolfgang Macherey, Franz Och, Ignacio Thayer, and Jakob Uszkoreit. Lattice-based minimum error rate training for statistical machine translation. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2008.

[11] David McClosky, Eugene Charniak, and Mark Johnson. Effective self-training for parsing. In *Proceedings of the 2006 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, pages 152–159, 2006.

[12] Graham Neubig and Taro Watanabe. Optimization for statistical machine translation: A survey. *Computational Linguistics (CL)*, 42(1):1–54, 2016.

[13] Franz Josef Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 160–167, 2003.

[14] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*, 2015.

[15] Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. Minimum risk training for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1683–1692, 2016.

[16] David A. Smith and Jason Eisner. Minimum risk annealing for training log-linear models. In *Proceedings of the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 787–794, 2006.

[17] Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. *Proceedings of the 17th Annual Conference on Neural Information Processing Systems (NIPS)*, 16, 2003.

[18] Nicola Ueffing. Self-training for machine translation. In *NIPS workshop on Machine Learning for Multilingual Information Access*, 2006.

[19] Taro Watanabe, Jun Suzuki, Hajime Tsukada, and Hideki Isozaki. Online large-margin training for statistical machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 764–773, 2007.

[20] John Wieting, Taylor Berg-Kirkpatrick, Kevin Gimpel, and Graham Neubig. Beyond BLEU: Training neural machine translation with semantic similarity. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[21] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[22] Sam Wiseman and Alexander M. Rush. Sequence-to-sequence learning as beam-search optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1296–1306, 2016.