Chapter 10
Machine Learning Tricks

Machine learning has a beautiful story to tell. Given a set of examples of a real world problem (in our case, translated sentence pairs), an algorithm automatically builds a model that represents the model so it can then make predictions. Deep learning promises that we do not even need to worry much about the specific properties of the problem (for instance, that sentences have verbs and nouns that play different roles). No, deep learning automatically discovers these and does away with the task of feature engineering. And all this just by the simple method of error back-propagation via gradient descent.

The unfortunate reality is that deep learning for complex problems such as machine translation requires a bag of tricks that addresses the many ways the basic learning algorithm may get off track. In this chapter, I first survey common pitfalls of machine learning and then go over methods that today's neural machine translation models use to address them.

This chapter reminds me of a conversation long ago with one of my graduate students who ventured deep into machine learning and whose thesis draft had a chapter on various machine learning tricks. I asked him why he listed all of those obscure and often counterintuitive methods, and he answered: "Because I use them all." The same is true today.

10.1 Failures in Machine Learning

machine learning failures

We painted a picture of gradient descent as walking down the hillside to a valley of minimum error. Even in this simple world—which ignores that we operated in a space with thousands, maybe even millions of

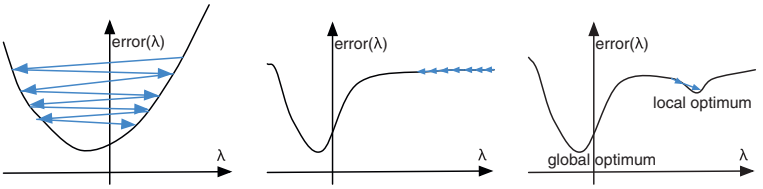

Figure 10.1 Problems with gradient descent training that motivate some of the refinements detailed in this chapter. (a) A too high learning rate may lead to too drastic parameter updates, overshooting the optimum. (b) Bad initialization may require many updates to escape a plateau. (c) The existence of local optima trap training.

dimensions—a lot can go wrong. Some of the basic failures are pictured in Figure 10.1.

learning rate

Learning Rate

The learning rate is a hyper parameter that defines how much the gradients are scaled to perform weight updates. In its simplest form, this is a fixed value that has to be set by hand. Setting the learning rate too high leads to updates that overshoot the optimum (Figure 10.1a). Conversely, a too low learning rate leads to slow convergence.

There are additional considerations about setting the learning rate. At the beginning of training, all the weights are far from optimal values, so we want to change them a lot. At later stages, updates are more nuanced, and we do not expect major changes. So, we may want to initially set the learning rate high, but decrease it over time. This general idea is called **annealing**. Later in this chapter, we will explore more sophisticated methods that adjust the learning rate.

annealing

weight initialization

Initialization of Weights

At the beginning of training, all weights are initialized to random values. Randomness is important so that the different hidden nodes start with different values and hence are able to learn different roles to play. If these values are very far from an optimum, we may need many update steps to reach it (Figure 10.1b). This is especially a problem with activation functions like sigmoid, which have only a short interval of significant change, and even more so with rectified linear units, which have gradients of zero for some interval. We obviously do not have a crystal ball to randomly set weights close to their optimal values, but at least they should be in a range of values that are easy to adjust, i.e., where changes to their values have an impact.

Local Optima

local optimum

Figure 10.1c gives a simple depiction of the problem of a local optimum. Instead of reaching the deep valley of the global optimum, training gets stuck in a little dip. Moving left or right would mean going uphill, so we will not leave it. The existence of local optima lead the search to get trapped and miss the global optimum.

Note that the depiction is a vast simplification of the problem. We usually operate with activation functions that are convex, i.e., they have a clear optimum that can be reached. However, we are operating with many parameters that span a highly dimensional space, and we also process many training examples (or batches of training examples) for which the error surface looks different every time.

Vanishing and Exploding Gradients

A specific problem of deep neural networks, especially recurrent neural networks, is the problem of vanishing and exploding gradients. A characteristics of these networks is a long path of computations that connects the input to the output of the computation. The error is measured at the output, but it has to be propagated all the way to the first parameters that operate on the input.

Recall that this happens by multiplication of the gradients of the computations along the path. If all of these gradients are above 1, we end up with a large number at the end (called **exploding gradients**). If all of these gradients are below 1, we end up close to zero (called **vanishing gradients**; see Figure 10.2). Hence we make either too extreme or practically no updates to the parameters for the early computations in the path.

exploding gradient

vanishing gradient

Note that this problem is especially acute for recurrent neural networks. Here, the long path of computations from input to output is packed with loops over the same calculations. Any increase or decrease of gradient values is accelerated in these loops.

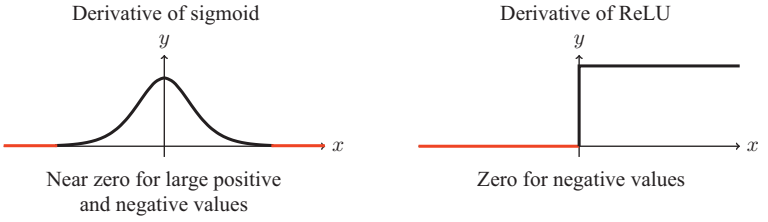

Figure 10.2 Vanishing gradients. If gradients are close to zero, no updates are applied to parameters, also for parameters upstream in the computation graph.

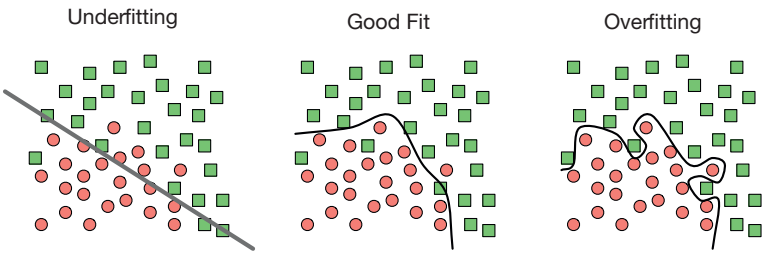

Figure 10.3 Underfitting and overfitting. With too few trainable parameters (low capacity), the model cannot match the data well (underfitting). Having too many parameters (high capacity) leads to memorizing the training data and insufficient generalization (overfitting).

Overfitting and Underfitting

The goal of machine learning is to learn the general principles of a problem based on representative examples. It does so by setting the values of the model parameters for a provided model architecture. The complexity of the model has to match the complexity of the problem. If the model has too many parameters and training runs long enough, the model will memorize all the training examples, but do poorly on unseen test examples. This is called **overfitting**. On the other hand, if the number of parameters is too small, the model will not be able to represent the problem. This is called **underfitting**.

overfitting

underfitting

Figure 10.3 shows a two-dimensional classification task. A too simple model that can only draw a line between the red circles and the green squares will not be able to learn the true curve separating the two classes. Having too much freedom to draw the curve leads to insufficient generalization. The degree of freedom is called the **capacity** of the model.

capacity

10.2 Ensuring Randomness

randomness

The ideal view of machine learning is that we are starting in some random point in parameter space, encounter randomly **independent and identically distributed** training examples (meaning that they are drawn independently from the same underlying true probability distribution), and move closer to a model that resembles the truth. To come close to this ideal, it is important to avoid undue structure in the training data and the initial weight setting. There is a whole machine learning approach called **maximum entropy training** that is based on the principle that absent concrete evidence, the model should be as random as possible (i.e., have maximum entropy).

maximum entropy training

10.2.1 Shuffling the Training Data

shuffling training data

The training data used for neural machine translation typically come from several sources, say the European Parliament Proceedings or a collection of subtitle translations. These individual corpora have very specific characteristics that drive training into different directions. Each corpus may also have internally different characteristics, for instance a chronological sorted corpus that draws training data initially from older text and ends with the latest additions.

Since we are updating the weights one batch of training data at a time, they will be more affected by the last training examples seen. If the last part of the training data is all from one of the corpora, this biases the model unfairly to it. Note also that we typically stop training once performance on a validation set does not improve—measured by cross-entropy or translation quality such as the BLEU score. If parts of the training data are more helpful than others (or just more similar to the validation set), these performance measures may differ over the course of training, and training may be prematurely stopped just because a harmful stretch of training examples is encountered.

To balance out these effects, the training data are randomly shuffled at the beginning of training. It is common to reshuffle the training data each epoch (each full pass through the training data), but this may not be necessary due to the very large corpora used in neural machine translation.

10.2.2 Weight Initialization

initialization

Before training starts, weights are initialized to random values. The values are chosen from a uniform distribution. We prefer initial weights that lead to node values that are in the transition area for the activation function and not in the low or high shallow slope where it would take a long time to push toward a change. For instance, for the sigmoid activation function, feeding values in the range of, say, $[-1; 1]$ to the activation function leads to activation values in the range of [0.269;0.731].

For the sigmoid activation function, commonly used formulas for weights to the final layer of a network are

$$\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], \tag{10.1}$$

where n is the size of the previous layer. For hidden layers, we chose weights from the range

$$\left[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}\right], \tag{10.2}$$

where n_j is the size of the previous layer, n_{j+1} size of next layer.

These formulas were first suggested by Glorot and Bengio (2010).

10.2.3 Label Smoothing

label smoothing

The predictions of the neural machine translation models are surprisingly confident. Often almost all the probability mass is assigned to a single word, with word prediction probabilities of over 99%. These peaked probability distributions are a problem for both decoding and training. In decoding sensible alternatives are not given enough credence, preventing successful beam search. In training, overfitting is more likely.

A common strategy for combating peaked distributions during decoding is to smooth them (Chorowski and Jaitly, 2017). As I described it so far, the prediction layer produces numbers for each word that are converted into probabilities using the softmax:

$$p(y_i) = \frac{\exp s_i}{\sum_j \exp s_j}. \tag{10.3}$$

temparature

The softmax calculation can be smoothed with what is commonly called a **temperature** T:

$$p(y_i) = \frac{\exp s_i/T}{\sum_j \exp s_j/T}. \tag{10.4}$$

With higher temperature, the distribution becomes smoother, i.e., less probability is given to the most likely choice.

But the problem of peaked distributions is rooted in training where the truth assigns all probability mass to a single word, so the training objective is to optimize toward such distributions. To remedy this, we can present as truth not this perfectly peaked distribution but a smoothed distribution that spreads out some of the probability mass (say, 10% of it) to other words. This may be done uniformly (assigning all words the same probability), but may also take unigram word probabilities into account (relative counts of each word in the target side of the training data).

10.3 Adjusting the Learning Rate

adjusting the learning rate

Gradient descent training implies a simple weight update strategy: just follow the gradient downhill. Since the actual gradients have fairly large values, we scale the gradient with a learning rate—typically a very low

number such as 0.001. Moreover, we may want to change the learning rate over time (starting with larger updates, and then refining weights with smaller updates), or adjust it for other reasons.

A simple **learning rate schedule** may reduce the learning rate after each epoch, maybe cutting it in half. But more sophisticated method have been proposed and are in common use.

learning rate schedule

10.3.1 Momentum Term

Consider the case where a weight value is far from its optimum. Even if most training examples push the weight value in the same direction, it may still take a while for each of these small updates to accumulate until the weight reaches its optimum. A common trick is to use a **momentum term** to speed up training. This momentum term m_t gets updated at each time step t (i.e., for each training example). We combine the previous value of the momentum term m_{t-1} with the current raw weight update value Δw_t and use the resulting momentum term value to update the weights.

momentum term

For instance, with a decay rate of 0.9, the update formula changes to

$$\begin{aligned} m_t &= 0.9 m_{t-1} + \Delta w_t \\ w_t &= w_{t-1} - \mu\, m_t. \end{aligned} \tag{10.5}$$

10.3.2 Adapting Learning Rate per Parameter

A common training strategy is to reduce the learning rate μ over time. At the beginning the parameters are far away from optimal values and have to change a lot, but in later training stages we are concerned with nuanced refinements, and a large learning rate may cause a parameter to bounce around an optimum.

Adagrad

Different parameters may be at different stages on the path to their optimal values, so a different learning rate for each parameter may be helpful. One such method, called **Adagrad**, records the gradients that were computed for each parameter and accumulates their square values over time, and uses this sum to adjust the learning rate.

Adagrad

The Adagrad update formula is based on the sum of gradients of the error E with respect to the weight w at all time steps t, i.e., $g_t = \frac{\partial E_t}{\partial w}$. We divide the learning rate μ for this weight by the accumulated sum of gradients:

$$\Delta w_t = \frac{\mu}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}}\, g_t. \tag{10.6}$$

Intuitively, big changes in the parameter value (corresponding to big gradients g_t), lead to a reduction of the learning rate of the weight parameter.

Adam

Combining the idea of momentum term and adjusting parameter update by their accumulated change is the inspiration of **Adam**, another method to transform the raw gradient into a parameter update.

Adam

First, there is the idea of momentum, which is computed as in Equation 10.5:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t. \tag{10.7}$$

Then, there is the idea of the squares of gradients (as in Adagrad) for adjusting the learning rate. Since raw accumulation runs the risk of becoming too large, and hence permanently depressing the learning rate, Adam uses exponential decay, just like for the momentum term:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \tag{10.8}$$

The hyper parameters β_1 and β_2 are set typically close to 1, but this also means that early in training the values for m_t and v_t are close to their initialization values of 0. To adjust for that, they are corrected for this bias:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{10.9}$$

With increasing training time steps t, this correction goes away: $\lim_{t \to \infty} \frac{1}{1-\beta^t} \to 1$.

Having these pieces in hand (learning rate μ, momentum $\hat{m}_t$, accumulated change $\hat{v}_t$), the weight update per Adam is computed as:

$$\Delta w_t = \frac{\mu}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \tag{10.10}$$

Common values for the hyper parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

There are various other adaptation schemes. This is an active area of research. For instance, the second-order derivative gives some useful information about the rate of change (the square matrix with second derivatives is called **Hessian matrix**). However, it is often expensive to compute, so other shortcuts are taken.

Hessian matrix

10.3.3 Batched Gradient Updates

batched gradient updates

One way to convert gradients into weight updates is to first process all the training examples, add up all the computed gradient values, and the use this sum to update parameters. However, this requires many passes over the training data for training to converge.

On the other extreme, as I described, we may process one training example at a time, and use its gradient immediately to update weights. This variant is called **stochastic gradient descent**, since it operates on a randomly sampled training example. This converges faster but has the disadvantage that the last seen training examples have a disproportionately higher impact on the final parameter values.

stochastic gradient descent

Given the efficiency gains of parallelization with as much computation as possible on modern GPU hardware, we typically batch together several sentence pairs, compute all their gradients for all their word predictions, and then use the sum over each batch to update parameters.

Note that it is not only the specifics of GPU design that leads us to process training examples in batches. We may also want to distribute training over several machines. To avoid communication overhead, each machine processes a batch of training examples and communicates gradients back to a parameter server for carry out the updates. This raises the problems that some machines are faster at processing their batch for various reasons, most having to do with underlying systems issues such as load on machines. Only when all machines have reported back their results, updates to the model are applied and new batches are issued.

In **asynchronous**, instead of waiting for all machines to report back their updates, each update is immediately applied and a new batch is issued. This training set up creates the problem that some machines operate on rather old parameter values. In practice, however, that does not seem to cause much of a problem. An alternative is not to wait for the stragglers and to discard their updates (Chen et al., 2016a).

asynchronous training

10.4 Avoiding Local Optima

avoiding local optima

The hardest problem for designing neural network architectures and optimization methods is to ensure that the model converges to the global optimum, or at least to a set of parameter values that give results close to this optimum on unseen test data. There is no real solution to this problem. It requires experimentation and analysis that is more craft than science. Still, this section presents a number of methods that generally help avoiding getting stuck in local optima.

10.4.1 Regularization

regularization

Large-scale neural machine translation models have hundreds of millions of parameters. But these models are also trained on hundreds of millions of training examples (individual word predictions). There are no hard rules for the relationship between these two numbers, but there is a general sense that having too many parameters and too few training examples leads to overfitting. Conversely a model with too few parameters and many training examples does not have enough capacity to learn the properties of the problem.

A standard technique in machine learning is **regularization**, which adds a term to the cost function to keep the model *simple*. By simple we mean parameter values that do not take on extreme values but rather have values close to zero, unless it is really necessary otherwise. A simple model also may not use all its parameters, i.e., sets their value to zero. There is a general philosophy behind the idea of keeping models simple.

Occam's razor

In this discussion **Occam's razor** is frequently invoked: if there are multiple explanations, prefer the simple one.

The complexity of a model is commonly measured with the L2 norm over the parameters. We add the L2 norm to the cost function, i.e., the sum of the squares of all parameters. So, the training objective is not only to match the training data as closely as possible but also to keep the parameters values small. Consider what happens when we compute the gradient of the L2 norm. We obtain a number relative to its current value. Gradient descent then reduces the parameter, unless there is evidence otherwise. So, adding the L2 norm can be understood as **weight decay**, where parameters that do not show any benefit are pushed towards zero.

weight decay

Deep learning methods for machine translation typically may not include a regularization term in the cost function, but many techniques can be understood as a form of regularization by other means.

10.4.2 Curriculum Learning

The sentence pairs in the parallel corpus are presented to the learning algorithm in random fashion. There are good arguments for this randomness, as laid out above. But there is also a line of research that explores how to present the training data in different ways. This idea is called **curriculum learning**. Just students over their time in a elementary school or university are not bombarded with facts in random order, they start with the easiest concepts first, and then move to more complex issues.

curriculum learning

For our problem of machine translation, this means that we may want to sort the training examples by easy to hard. First run an epoch

on the easy examples, then add harder ones, and only in the final epochs run on the full training set. This is a simple idea, but there are a lot of difficult details.

First, how do we measure the difficulty of sentence pairs? We could take simple guidance such as preferring short sentences, maybe even creating artificial training data by extracting smaller segments from sentence pairs (similar to phrase pair extraction in statistical machine translation). We may use a pretrained neural machine translation system to score sentence pairs in the hope to discard outliers and poorly matched sentences.

Second, how exactly do we set up the curriculum? How many easy examples in the first epochs? How many epochs should we spend on each level of difficulty? These are decisions that are decided by experimentation to establish some rules of thumb, but the answers may differ a lot for different data conditions.

There are many open questions here at the time of writing, but we should note that the idea of training on different data sets in subsequent epochs is a common idea for adaptation, which I discuss at length in Chapter 13.

10.4.3 Dropout

While the last two sections discussed methods that help get training started, we now consider a different concern. What if training is well under way, but the current set of parameters are stuck in a region where some of the properties of the task have been learned, but the discovery of other properties would take it too far out of its comfort zone.

In machine translation, a vivid example of this is a model that has learned the language model aspects of the task, but is oblivious to the role of the input sentence and hence the translation aspects. Such a model produces beautiful output language, however it is completely unrelated to the input. Further training steps may just refine the model to make better use of distant words in the output sentence, but keep ignoring the input.

Various methods have been proposed to get training out of these local optima. One currently popular method is called **drop-out**. It sounds a bit simplistic and wacky. For each batch of training examples, some of the nodes in the network are randomly chosen to be ignored. Their values are set to 0, and their associated parameters are not updated. These dropped-out nodes may account for as much as 10%, 20% or even 50% of all the nodes. Training resumes for some number of iterations without the nodes (maybe just for just one batch), and then a different set of drop-out nodes are selected (Figure 10.4).

drop-out

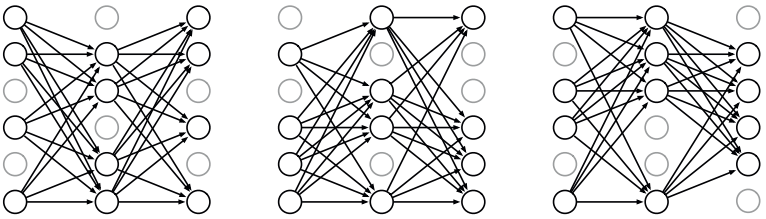

Figure 10.4 Dropout. For each batch of training examples, a different random set of nodes is removed from training, their values are set to 0 and their weights are not updated.

Obviously, the dropped-out nodes played some useful role in the model trained up to the point when they are ignored. But after that, other nodes have to pick up the slack. The end result is a more robust model where several nodes share similar roles.

One way to make sense of drop-out, is to view it as a form of ensemble learning. We can typically achieve large gains by not only training one model, but having multiple training runs, each producing a model and then merging their predictions during inference (see Section 9.2). Models make different mistakes, but when they agree they are more likely to be right. Removing nodes from the network creates effectively a different model that is trained on its own—at least for a while. So by masking out different subsets of nodes, we simultaneously train multiple models. While these share a lot of parameters and we do not explicitly merge multiple predictions, we avoid a lot of overhead by having a smaller set of parameters and fewer computation than when training a real ensemble.

10.5 Addressing Vanishing and Exploding Gradients

vanishing gradient
exploding gradient

Since gradients are computed over a long path of computations in the computation graph, these may misbehave, either becoming too small or too large. This is especially a problem for deep models with many layers or recurrent steps. When developing a neural network architecture, the behavior of gradients is important to keep in mind. There are also some general techniques that help address this problem.

10.5.1 Gradient Clipping

gradient clipping

If gradients become too large, a simple method is to just reduce their value. This method is called gradient clipping. The sum of all gradients for a parameter vector limited to a specified threshold. Typically, the L2 norm is used.

Formally, we define a hyper parameter τ for the threshold and check if the L2 norm of the gradient values for a parameter tensor (typically a weight matrix) exceeds it. If it does, we scale each gradient value by the ratio between the threshold τ and the L2 norm over all gradient values g_j. Thus each gradient g_i of the tensor, is adjusted by

$$g'_i = g_i \times \frac{\tau}{\max(\tau, L2(g))} = g_i \times \frac{\tau}{\max\left(\tau, \sqrt{\sum_j g_j^2}\right)}. \tag{10.11}$$

Instead of choosing a fixed threshold, we may also dynamically detect unusual gradient values and discard these updates. In **adaptive gradient clipping**, we keep track of the mean and variance of gradient values, which allows us to detect gradient values that lie well outside the normal distribution. This is typically done by updating a moving average and moving standard deviation (Chen et al., 2018).

adaptive gradient clipping

10.5.2 Layer Normalization

layer normalization

Layer normalization addresses a problem that arises especially in the deep neural networks that we are using in neural machine translation, where computing proceeds through a large sequence of layers. For some training examples, average values at one layer may become very large, which feed into the following layer, also producing large output values, and so on. This is a problem with activation functions that do not limit the output to a narrow interval, such as rectified linear units.

The opposite problem is that for other training examples the average values at the same layers may be very small. This causes a problem for training. Recall from Equation 5.31 that gradient updates are strongly affected by node values. Too small node values lead to vanishing gradients and too large node values lead to exploding gradients.

To remedy this, the idea is to normalize the values on a per-layer basis. This is done by adding additional computational steps to the neural network. Recall that a feed-forward layer l consists of the the matrix multiplication of the weight matrix W with the node values from the previous layer h^{l-1}, resulting in a weighted sum s^l, followed by an activation function such as sigmoid:

$$\begin{aligned} s^l &= W\, h^{l-1} \\ h^l &= \text{sigmoid}(s^l). \end{aligned} \tag{10.12}$$

We can compute the mean μ^l and variance σ^l of the values in the weighted sum vector s^l by

$$\begin{aligned} \mu^l &= \frac{1}{H}\sum_{i-1}^{H} s_i^l \\ \sigma^l &= \sqrt{\frac{1}{H}\sum_{i-1}^{H}(s_i^l - \mu^l)^2}. \end{aligned} \tag{10.13}$$

Using these values, we normalize the vector s^l using two additional bias vectors g and b:

$$\hat{s}^l = \frac{g}{\sigma^l}(s^l - \mu^l) + b, \tag{10.14}$$

where the difference subtracts the scalar average from each vector element.

The formula first normalizes the values in s^l by shifting them against their average value, hence ensuring that their average afterward is 0. The resulting vector is then divided by the variance σ^l. The additional bias vectors give some flexibility. They may be shared across multiple layers of the same type, such as multiple time steps in a recurrent neural network.

10.5.3 Shortcuts and Highways

shortcut connection
highway connection

The *deep* in deep learning implies models that go through several stages of computation (typically a sequence of layers). Obviously, passing the error information through these many stages is a challenge for the back-propagation algorithm. It is quite remarkable, that this works at all, considering the many computations that an input value has to travel to produce a output value, which is then matched against the true target value. All the parameters involved in this chain have to be adjusted. Given that we start with random values for all of them, it is amazing that training gets off the ground at all.

But deeper architecture may stretch this amazing ability too far. To avoid this problem, a common method is to add shortcuts to the architecture of these models. Instead of forcing input values though, say, 6 layers of processing, we add a connection that connects the input directly to the last layer. In early iterations of training, we expect the model to focus on these simpler paths (reflecting the learning of with simpler model architecture) but in later iterations to exploit the true power of the deep models.

residual connection
skip connection

These shortcuts have many names: **residual connections** and **skip connections** are commonly used. In their simplest form, a feed-forward layer,

$$y = f(x), \tag{10.15}$$

is extended with just passing through the input (skipping the pass through the function f):

$$y = f(x) + x. \tag{10.16}$$

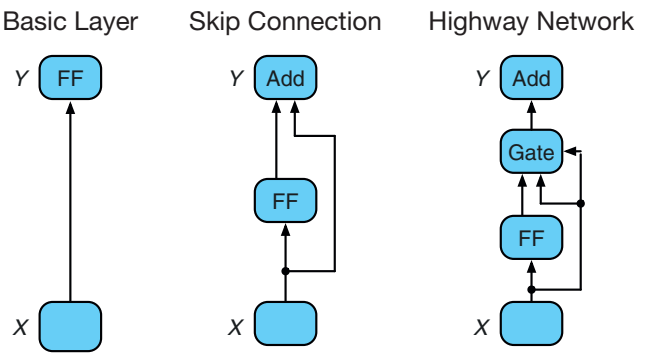

Figure 10.5 Shortcuts and highways. Left: basic connection between layer *X* and *Y*. Middle: in residual or skip connection, the input is added to the output of the processing between layers *X* and *Y*. Right: in highway networks, a gate value is learned that balances between processing

Note the effect on gradient propagation. The gradient for this computation is

$$y' = f'(x) + 1. \tag{10.17}$$

The constant 1 implies that the gradient is passed through unchanged. So, even in a deep network with several layers, there is a direct path for error propagation from the last layer all the way to the input.

Taking this idea one step further, we may regulate how much information from $f(x)$ and x should impact the output y. In **highway networks**, this is regulated with a gate $t(x)$ (typically computed by a feed-forward layer).

$$y = t(x)\, f(x) + (1 - t(x))\, x. \tag{10.18}$$

Figure 10.5 shows these different designs for shortcut connections.

10.5.4 LSTM and Vanishing Gradients

vanishing gradient

I introduced long short-term memory cells in Section 7.5. One of the core motivations for these type of neural network elements is the vanishing gradient problem. Recurrent neural networks are especially prone to vanishing gradients, since the error is back-propagated through each step of the sequence that was processed. LSTM cells allow for shortcuts through the network, in the spirit of what I discussed in the previous section.

Specifically, the pass-through of memory connects the time steps of sequence processing. Recall Equation 7.12:

$$\text{memory}^t = \text{gate}_{\text{input}} \times \text{input}^t + \text{gate}_{\text{forget}} \times \text{memory}^{t-1}. \tag{10.19}$$

For nodes for which the forget gate has values close to 1, the gradient is passed through to earlier steps nearly unchanged.

10.6 Sentence-Level Optimization

sentence-level optimization

The cost function that we use to drive parameter updates so far has been defined at the word level. For each output word given in the parallel data, we first make a model prediction in the form of a distribution over output words. Then we compare how much probability was given to the given correct output word.

While this allows for many parameter updates, machine translation is a sequence prediction task. There is concern that predicting one word at a time, given the correct prefix of correct previous output words, is not a realistic learning objective for test time, when an entire output sequence has to be generated without the safety net of the correct prefix. In machine learning, this is called **exposure bias**, the problem that conditions seen at test time are never encountered during training. Should we not also at training time predict an output sequence and then compare that against the correct sequence?

exposure bias

10.6.1 Minimum Risk Training

minimum risk training

When predicting an output sequence, word-by-word comparison against the given correct output sequence is not useful. If an additional word was generated at the beginning of the sequence, then all words are off by one position and none match, while the proposed sequence may be still a fine sentence translation. Also, consider the case of reordering—any deviation of the word order of the correct output sequence would be severely punished.

Comparing sentence translations for the purpose of evaluation has been extensively studied. This is essentially the long-standing problem of sentence-level automatic machine translation metrics. The oldest and still dominant metric is the BLEU score (see Section 4.3.1). It matches words and short word sequences between system output and reference translation, and includes a brevity penalty. But there are also other metrics, such as translation edit rate.

We previously used as cost function (or loss) cross-entropy on word predictions. We defined this as the negative of the predicted probability (according to the distribution t_i) assigned to the correct word y_i, as presented in Equation 8.5:

$$\text{loss} = -\log t_i[y_i]. \tag{10.20}$$

Another way to look at this, is that the model makes various predictions y for the ith output word, and we assign a loss value to each prediction, which happens to be 1 for the correct word and 0 for the incorrect words.

$$\text{loss} = -\log \sum_{y \in V} \delta(y = y_i)\ t_i[y]. \tag{10.21}$$

This formulation allows us to use other quality measures beyond a 0/1 loss, which is what we have in mind for sentence-level optimization. Staying with word-level optimization for the moment, we may want to employ loss functions that give partial credits to near misses, such as morphological variants or synonyms. So, instead of the Kronecker delta δ that fires only when there is a match between y and y_i, we may use any other quality metric:

$$\text{loss} = -\log \sum_{y \in V} \text{quality}(y, y_i)\ t_i[y]. \tag{10.22}$$

Since the quality score is weighted by the probability distribution t_i, the resulting loss function can be considered a computation of the **expected loss**. The method that optimizes to minimize the expected loss is called **minimum risk training**.

expected loss

minimum risk training

Now, we are predicting the full sequence $\mathbf{y}$. Mathematically, this does not much change our definition of the loss:

$$\text{loss} = -\log \sum_{\mathbf{y'} \in V*} \text{quality}(\mathbf{y'}, \mathbf{y})\ t(\mathbf{y'}). \tag{10.23}$$

We can now use the BLEU score or any other evaluation metric as the quality score.

There are a number of computational challenges with this formulation. As written, the space of possible sequences $\mathbf{y'} \in V*$ is infinite but even with reasonable length constraints, the number of sequences is exponential with respect to the length. To keep matters manageable, we have to resort to sampling sequences that, combined, account for the bulk of the probability mass.

Sampling full-sequence translations can be done with beam search. While this aims to find the most probably translations, it suffers from lack of diversity. An alternative is **Monte Carlo search**. This works like repeated greedy search, with one distinction. Instead of choosing the most probable word from the distribution each time, we now choose the output word given its predicted probability.

Monte Carlo decoding

To give an example: if the model predicts the word *cat* with 17% probability, and the word *tiger* with 83% probability, we now roll a die and if the number 1 comes up, we choose cat, otherwise we choose tiger. This process generates each possible output sequence according to its sequence probability, given the model. We repeat this search multiple times and obtain multiple sentence translations. In the limit, the relative

frequency of each output sequence approximates its probability—hence the required distribution t in Equation 10.23.

Note that while sampling the space of possible translations makes sentence-level optimization feasible, it is still more computationally expensive than traditional word-level optimization. We have to carry out a beam search or repeated Monte Carlo searches to collect one data point for computing back-propagation values. Previously, a single word prediction sufficed.

But there are clear benefits to sentence-level optimization. It more closely mirrors decoding during test time, when a sequence is predicted and scored. It also allows methods such as generative adversarial training.

10.6.2 Generative Adversarial Training

generative adversarial training

generator

discriminator

Coming from computer game playing research, generative adversarial networks (GANs) frame a task such as machine translation as a game between two players: the **generator** and the **discriminator**. The generator makes predictions for the task, and the discriminator attempts to distinguish these predictions from actual training examples.

Applied to machine translation (Wu et al., 2017; Yang et al., 2018b), the generator is a traditional neural machine translation model that generates translations for input sentences and the discriminator attempts to tell them apart from real translations. Specifically, given a sentence pair (x, y) from the training data, the generator (i.e., the neural machine translation model) takes the input sentence x and produces a translation t for it. The discriminator aims to detect this translation pair (x, t) as coming from a machine, while classifying the sentence pair from the training data (x, y) as coming from a human.

During training, the discriminator receives a positive sample (x, y) and a negative sample (x, t) to improve its prediction ability. The ability to make the right decision for both of them is used as error signal for back-propagation training.

But the point of the discriminator is to also improve training of the generator. Typically, we train the generator to predict each word of the output sequence. Now, we combine this training objective with the objective to fool the discriminator. So, we have two objective functions. The importance of each objective is balanced with some weight λ, a hyper parameter, which has to be chosen by hand.

What makes this setup complex is that the signal from the discriminator can be computed only for a completed sentence translation t, while typical neural machine translation training is based on making correct individual word predictions. In machine learning, this type of problem

is framed as a **reinforcement learning** problem. Reinforcement learning is defined as a setup where the error signal for training is obtained only after a sequence of decisions. Good examples are games such as chess and walking through a maze to avoid monsters and find the gold. In the chess example, we only know if our moves are correct when the game ended in our favor, but we do not get individual feedback for each step.

reinforcement learning

In the language of reinforcement learning, the generator is called a **policy**, and the ability to fool the discriminator is called the **reward**. A common solution to this problem is to sample different translations based on our current neural machine translation model, as described in the previous section. Given a set of sampled translations, we can compute the reward, i.e., how well we can fool the discriminator, for each of them. We use this as the training objective to update the model.

policy

reward

10.7 Further Readings

A number of key techniques that have been recently developed have entered the standard repertoire of neural machine translation research. Ranges for the random initialization of weights need to be carefully chosen (Glorot and Bengio, 2010). To avoid overconfidence of the model, label smoothing may be applied, i.e., optimization toward a target distribution that shifts probability mass away from the correct given target word toward other words (Chorowski and Jaitly, 2017). Distributing training over several GPUs creates the problem of synchronizing updates. Chen et al. (2016a) compare various methods, including asynchronous updates. Training is made more robust by methods such as drop-out (Srivastava et al., 2014), where during training intervals a number of nodes are randomly masked. To avoid exploding or vanishing gradients during back-propagation over several layers, gradients are typically clipped (Pascanu et al., 2013). Chen et al. (2018) present briefly adaptive gradient clipping. Layer normalization (Lei Ba et al., 2016) has similar motivations, by ensuring that node values are within reasonable bounds.

Adjusting the Learning Rate

adjusting the learning rate

An active topic of research is optimization methods that adjust the learning rate of gradient descent training. Popular methods are Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), and currently Adam (Kingma and Ba, 2015).

Sequence-Level Optimization

sentence-level optimization

Shen et al. (2016) introduce minimum risk training that allows for sentence-level optimization with metrics such as the BLEU score. A set

of possible translation is sampled, and their relative probability is used to compute the expected loss (probability-weighted BLEU scores of the sampled translations). They show large gains on a Chinese–English task. Neubig (2016) also showed gains when optimizing toward smoothed sentence-level BLEU, using a sample of 20 translations. Hashimoto and Tsuruoka (2019) optimize toward the GLEU score (a variant of the BLEU score) and speed by training by vocabulary reduction. Wiseman and Rush (2016) use a loss function that penalizes the gold standard of falling off the beam during training. Ma et al. (2019c) also consider the point where the gold standard falls of the beam but record the loss for this initial sequence prediction and then reset the beam to the gold standard at that point. Edunov et al. (2018b) compare various word-level and sentence-level optimization techniques but see only small gains by the best-performing sentence-level minimum risk method over alternatives. Xu et al. (2019) use a mix of gold-standard and predicted words in the prefix. They use an alignment component to keep the mixed prefix and the target training sentence in sync. Zhang et al. (2019b) gradually shift from matching toward ground truth toward so-called word-level oracle obtained with Gumbel noise and sentence-level oracles obtained by selecting the BLEU-best translation from the *n*-best list obtained by beam search.

faster training

Faster Training

Ott et al. (2018c) improve training speed with 16-bit arithmetic and larger batches that lead to less idle time due to less variance in processing batches on different GPU. They scale up training to 128 GPUs.

right-to-left training

Right-to-Left Training

Several researchers report that translation quality for the right half of the sentence is lower than for the left half of the sentence and attribute this to the exposure bias: during training a correct prefix (also called teacher forcing) is used to make word predictions, while during decoding only the previously predicted words can be used. Wu et al. (2018) show that this imbalance is to a large degree due to linguistic reasons: it happens for right-branching languages like English and Chinese, but the opposite is the case for left-branching languages like Japanese.

adversarial training

Adversarial Training

Wu et al. (2017) introduce adversarial training to neural machine translation, in which a discriminator is trained alongside a traditional machine translation model to distinguish between machine translation output and human reference translations. The ability to fool the discriminator is used as an additional training objective for the machine translation

model. Yang et al. (2018b) propose a similar setup, but add a BLEU-based training objective to neural translation model training. Cheng et al. (2018) employ adversarial training to address the problem of robustness, which they identify by the evidence that 70% of translations change when an input word is changed to a synonym. They aim to achieve more robust behavior by adding synthetic training data where one of the input words is replaced with a synonym (neighbor in embedding space) and by using a discriminator that predicts from the encoding of an input sentence if it is an original or an altered source sentence.

Knowledge Distillation

knowledge distillation

There are several techniques that change the loss function not only to reward good word predictions that closely match the training data but also to closely match predictions of a previous model, called the teacher model. Khayrallah et al. (2018a) use a general domain model as teacher to avoid overfitting to in-domain data during domain adaptation by fine-tuning. Wei et al. (2019) use the models that achieved the best results during training at previous checkpoints to guide training.