Using Deep Reinforcement Learning for Online Machine Translation

Harsh Satija

Computer Science
McGill University, Montreal

May 8, 2018

Acknowledgements

I would like to thank and express my deepest gratitude to my advisor, Joelle Pineau, for providing me the opportunities and freedom, and at the same time providing encouragement and support thorough out my graduate studies. I highly appreciate the time and energy spent on teaching me how to do good science.

Additional thanks go to He He and Hal Hal Daumé III for the initial discussions on the project and sharing their perspective on real-time machine translation.

Finally, I would like to thank my parents for all the support they have provided me over the years.

Abstract

We present a Deep Reinforcement Learning based approach for the task of real time machine translation. In the traditional machine translation setting, the translator system has to 'wait' till the end of the sentence before 'committing' any translation. However, real-time translators or 'interpreters' have to make a decision at every time step either to wait and gather more information about the context or translate and commit the current information. The goal of interpreters is to reduce the delay for translation without much loss in accuracy.

We formulate the problem of online machine translation as a Markov Decision Process and propose a unified framework which combines reinforcement learning techniques with existing neural machine translation systems. A training scheme for learning policies on the transformed task is proposed. We empirically show that the learnt policies can be used to reduce the end to end delay in translation process without drastically dropping the quality. We also show that the policies learnt by our system outperform the monotone and the batch translation policies while maintaining a delay-accuracy trade-off.

Résumé

Nous présentons une approche basée sur l'apprentissage par renforcement profond pour la tâche de traduction automatique en temps réel. Dans le cadre traditionnel de la traduction automatique, le système de traduction doit 'attendre' jusqu'à la fin de la phrase avant de 'valider' toute traduction. Cependant, les traducteurs en temps réel ou les 'interprètes' doivent décider à chaque moment s'ils doivent attendre et recueillir plus d'informations sur le contexte ou traduire et valider l'information disponible actuellement. Le but des interprètes est de réduire le délai de traduction sans perte de précision.

Nous formulons le problème de traduction automatique 'simultanée' comme processus de décision markovien et proposons un cadre unifié qui joint des techniques d'apprentissage par renforcement avec des systèmes neuronaux existants de traduction automatique. Un schéma d'entraînement pour les politiques d'apprentissage sur la tâche transformée est proposé. Nous montrons empiriquement que les politiques apprises peuvent être utilisées pour réduire le retard de bout en bout dans le processus de traduction sans pour autant réduire radicalement la qualité. Nous montrons également que les politiques apprises par notre système surpassent les politiques monotones de traduction et celles de traduction par lots tout en maintenant un compromis entre précision et retard.

Contents

List of Figures

List of Algorithms

1

Introduction

A desirable trait of an intelligent system will be its ability to communicate with humans. Humans communicate with each other primarily through the means of a language. However, there is no universal language and humans in different parts of the world have developed many different languages (there are roughly 7000 languages in the modern society (Anderson, 2004)). In order for machines to communicate with humans across the world, not only should they understand different languages, they should also have the ability to translate the information from one language to another. The field of training the machines to translate from one language to the other is called Machine Translation. The trained machines can also help humans to communicate with humans who speak other languages by removing the need to learn their language first or relying on an intermediary.

While a lot of work had been done to improve the quality of the translation systems over the past few decades (Sheridan, 1955, Brown et al., 1993, Och and Ney, 2003, Bahdanau, Cho, and Y. Bengio, 2014, Sutskever, Vinyals, and Le, 2014, Wu et al., 2016), the entire process is not human-like. So far, machine translation is only being done at sentence level, where every text is split into multiple sentences, each of which is treated independently. However, this is not the case in reality as humans neither process the meaning of text nor translate at the sentence level. Humans are able to capture the complex relationships and organization of various components within the

sentences (Koehn, Och, and Marcu, 2003, Och and Ney, 2003) as well as between a sequence of sentences (Mann and Thompson, 1988).

Online machine translation (or Real-time MT) is defined as producing a partial translation of a sentence before the entire input sentence is completed. It is often used in interactive multi-lingual scenarios, such as diplomatic meetings, where the translators or 'interpreters' have to produce a coherent partial translation before the sentence ends with minimum effect on the actual speed of conversation. Interpreters have to make complex decisions in real-time to either wait for more words to gather information about the context or translate the current partial context. This task becomes even more difficult when the source and target languages have different word orders like English (Subject-Verb-Object) and German (Subject-Object-Verb) (Grissom II et al., 2014).

Reinforcement Learning (RL) is a paradigm associated with sequential decision making. It provides a framework to train an agent that has to take a series of decisions (actions) to maximize some long-term goal. RL has been shown to be successful in a variety of tasks such as learning in games (Tesauro, 1995), networking (Boyan and Littman, 1994), inventory management (Crites and Barto, 1996, Simao et al., 2009) and robotics (Peters, Vijayakumar, and Schaal, 2003, Abbeel et al., 2007).

Deep Learning (DL) is a framework for learning good representations for a task. Given an objective, using Deep Learning we can hope to learn good representations which help to achieve that objective with minimal domain knowledge in an end-to-end learning fashion. As a result, Deep Learning has helped machine learning achieve state-of-the-art performance on a variety of tasks such as image classification (Krizhevsky, Sutskever, and G. E. Hinton, 2012), machine translation (Wu et al., 2016) and speech processing (G. Hinton, Deng, et al., 2012).

Deep Reinforcement Learning (DRL) combines Deep Learning with Reinforcement Learning. Reinforcement Learning helps with temporal decision making and provides the objective while Deep Learning provides a mechanism for learning representations

to achieve that objective. This allows the system to learn the optimal behavior directly from the raw input in an end-to-end manner. This combination has allowed to build agents that can perform better than humans on tasks which were previously considered very challenging (V. Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015, Silver et al., 2016).

1.1 Problem Statement and Objectives

Returning to the problem of machine translation, we consider a scenario where the text is being revealed one word at a time and we want a system that produces the translation in real-time. One way to address this task is to wait until the end of the input sentence before translating anything, but it adds a delay to the entire procedure and does not provide a natural element. The other option is to translate every word as it is being revealed, but the quality of translation produced will potentially be affected as word level translation systems cannot capture the relationships across different elements of a sentence. They also cannot be applied to cases where the sentence structure differs vastly.

We are interested in building a system that is able to make decisions regarding when to wait and gather more information about the input sequence or to start translating the current given sequence. The goal is to reduce delay in translation process without drastically dropping the quality. We want the framework to be flexible enough to generalize to new language translation tasks with minimal human supervision. We also want the system to exhibit a trade-off in delay and quality that can be controlled by the user. This allows the user to change the behavior of the real-time translation system depending on the kind of task at hand.

1.2 Contributions

Prior work in simultaneous machine translation is dominated by rule and parse-based approaches (Ryu, Matsubara, and Inagaki, 2006, He et al., 2015) or word segmentation based approaches (Oda et al., 2014). Our work is built on the earlier work by Grissom II et al. (2014), where they cast real-time machine translation as reinforcement learning task. However, their work only deals with verb final sentences and is limited to smaller datasets.

We extend the work by Grissom II et al. (2014) and present a framework which uses the existing neural machine translation systems to function as a simultaneous machine translation system. Our framework combines the tasks of learning translation and decision-making into a single architecture. This eliminates the need of any feature engineering and allows the model to be applied to real-time translation tasks in other languages with minimal previous knowledge.

Our results show that we can effectively trade-off translation accuracy and delay using RL approach on English to French, English to German and Japanese to English benchmark datasets. The work presented in this thesis has been published in the Abstraction in Reinforcement Learning Workshop, International Conference on Machine Learning (Satija and Pineau, 2016). In the thesis, we go into more detail on the individual components of the system. We also change and simplify the reward function formulation and present a more thorough analysis of the results.

The work is limited to texts, i.e., we do not experiment in audio or speech. We also ignore the computational delay in translating and generating a word. The work is highly dependent on the amount of training data available and models are likely to not perform well on the smaller datasets.

Source	ich	bin	mit	dem	Zug	nach	Ulm	gefahren.	
Word-level	I	am	with	the	train	to	Ulm	traveled.	
Sentence-level	(..... *wait till end* .....)								I traveled by train to Ulm.
Optimal	I	(..... *waiting* .....)						traveled	by train to Ulm.

Table 1.1: An example inspired by Grissom II et al., 2014 which demonstrate the difference between various type of translation procedures on an example from German to English.
The top row ("source") represents the original sentence in German.
The second row ("Word-level") represents a word level translation system which produces the translation of a word immediately, without any delay.
The third row ("Sentence-level") represents a sentence level translation system which waits till the end of sentence before producing a translation.
The last row ("Optimal") represents how a human translator generates a translation, producing the translation of the first word immediately (ich $\rightarrow$ I) but then deciding to wait till encountering the final verb before resuming the translation.

2

Supervised Learning

The goal of this chapter is to familiarize the reader with the essential machine learning techniques on which the work is built. For more thorough introduction to Machine Learning (ML) we advise the reader to look up textbooks by Bishop (2006) and Murphy (2012).

The core machine learning problems and algorithms can be broadly divided into three categories, based on the kind of information available at the time of learning:

- **Supervised learning**

  In this scenario, the algorithm has access to labelled observations in the form of <observation, label> pairs. The labels are given by an expert (usually humans) who is assumed to know the correct solutions. The goal of the learning problem is to learn this mapping of observations to labels. Example of these class of problems are face recognition (Taigman et al., 2014), cancer detection (Cruz and Wishart, 2006), machine translation (Koehn, 2009), etc.

  The amount of labelled data we have plays an important role in deciding the complexity of the learning problem. If one has close to an infinite amount of data, then it is possible to learn the most probable label for each of the observation in the observation space. At the same time, when there is limited data the need to exploit the pattern in the data becomes even more prominent.

- **Reinforcement learning**

In this scenario, the learning agent (or algorithm) can interact with its environment and the environment provides a sequence of observations and a signal in the form of a numerical value (reward/penalty). The goal of the learning problem is to learn a behaviour which maximizes the total reward the agent can acquire from the environment.

Most of the sequential decision making problems, where an algorithm has to take a series of decisions (actions) to maximize the long-term goal, fall into this category. Some example applications of the problems in this domain are game playing agents (V. Mnih, Kavukcuoglu, Silver, Graves, et al., 2013) and trading agents (Lee, 2001).

- **Unsupervised learning**

  In this setting the agent does not get any feedback from the environment or has access to any labelled data. The learning problem is to discover some intrinsic structure in the data. Examples include problems like clustering, dimensionality reduction (Tenenbaum, De Silva, and Langford, 2000, G. E. Hinton and Ghahramani, 1997).

The rest of this chapter is focused on introducing the supervised learning methods and techniques that will be used extensively in the subsequent chapters and as such are necessary to understand this work.

2.1 Preliminaries

The basic mathematical notation used exhaustively in rest of the work is as follows:

- $\mathbb{N}$ denotes the set of natural numbers: $\mathbb{N} : \{0, 1, 2, \dots\}$.
- $\mathbb{R}$ denotes the set of real numbers.

- Any vector denoted by v represents a column vector. v^T represents the transpose of a vector. A vector in d-dimensional vector space will be denoted by $v \in \mathbb{R}^d$.
- If $f = f(\theta, x)$, i.e. f is a function of x and θ, then $\frac{\partial}{\partial\theta} f$ denotes the partial derivative with respect to θ.
- If $f = f(\theta_0, \theta_1, \theta_2, \ldots, \theta_n)$, then the gradient represents a vector containing all the partial derivatives i.e.

$$\nabla f(\theta_0, \theta_1, \theta_2, \ldots, \theta_n) =< \frac{\partial f}{\partial \theta_0}, \frac{\partial f}{\partial \theta_1}, \ldots, \frac{\partial f}{\partial \theta_n} >$$

- If P is a probability distribution then $X \sim P$ means that X is a random variable sampled from P.
- Indicator event Ω will be denoted by $\mathbb{I}_{\{\Omega\}}$, i.e. $\mathbb{I}_{\{\Omega\}} = 1$, when Ω is true and $\mathbb{I}_{\{\Omega\}} = 0$ when Ω is false.
- $\sim$ refers to the sampling operation. Eg. $x \sim \mathcal{N}(0, I)$ means that x is sampled from a normal distribution with mean 0 and fixed identical co-variance (I).

2.2 Basic methods

As mentioned earlier, for supervised learning problems we have <observation, label> pairs and we want to learn a mapping between them. More formally, let $\mathcal{X}$ denote the observation or **input space**. If the input space is d-dimensional then a point input space is denoted by $x \in \mathbb{R}^d$. Often the dimension of the input space is also referred as its **feature space**. Similarly, let $\mathcal{Y}$ denote the label or **target space**. The amount of <input, target> the algorithm has access to is called a **dataset**, denoted by $D \subset \mathcal{X} \times \mathcal{Y}$. The goal in supervised learning is to find a function that learns the mapping from input to target space, $f : \mathcal{X} \to \mathcal{Y}$. Sometimes this is also called the predictor or hypothesis. If $\mathcal{Y}$ can take values in $\mathbb{R}$ then the problem is referred to as

regression and if $\mathcal{Y}$ can only take values within a finite discrete set then the problem is referred to as **classification**. Note that in this scenario we assume that samples are independent and identically distributed (i.i.d.). With this assumptions there is no relation between the elements of the dataset (except that they are drawn from the same underlying distribution).

A function $f : \mathcal{X} \to \mathcal{Y}$ with parameters w can be denoted by either $f(x; w)$. When $w \in \mathbb{R}^c$, $y \in \mathbb{R}^t$ and $x \in \mathbb{R}^d$ and $f(x; w)$ is a linear function, we have:

$$f(x; w) = b + w_1 * x_1 + \dots$$

This can also be represented in matrix form as $f(x; w) = xW + b$, where x represents the $n \times d$ matrix and W is a $d \times t$ matrix and $b \in \mathbb{R}^t$.

2.2.1 Learning Objective

In order to quantify how well the function is able to learn this mapping we need to define a surrogate loss (or cost function). This loss function is used in the optimization procedure to select the hypothesis function which has the minimum loss on the dataset. One of the commonly used loss function is the **Least Mean Squares (LMS)**:

$$J(\theta) = \frac{1}{N} \sum_i ||y_i - (x_i W + b)||^2.$$

In the above equation N represents the number of samples in the dataset and θ represents all the parameters of the function, in this case W and b.

2.2.2 Optimization of Linear Functions

When the size of the dataset is small we can solve for the exact solution directly by equating the gradient of the loss to zero with respect to the parameters. The computational complexity is polynomial in the size of the dataset. When the error cannot be solved in closed form (or is computationally expensive), but we can compute

∇J easily, we can use a gradient descent optimization procedure (Arfken, 1985). The main idea here is to treat the gradient as the slope of the loss function and move direction towards the negative slope with the aim to reach the minima, where the slope will be 0. We start with randomly initialized parameters and in each iteration we move towards the minima with some small step size (or **learning rate** α). The optimization procedure is shown in algorithm 1:

Input: learning rate: α, threshold: ϵ, initial parameters: w^0

Output: last value of w

while $\left\|w^{k+1} - w^k > \epsilon\right\|_1$ **do**

$w^{k+1} = w^k - \alpha \nabla J(w^k)$

end

return w^{k+1}

Algorithm 1: Gradient Descent optimization procedure

Note that we are chasing the minima of the function here with respect to where we start the procedure. If the function is convex, the procedure will converge to the global minima if the learning rate satisfies the Robbins-Monroe conditions (Robbins and Monro, 1951). Other often used optimization methods are Newton's method and second order methods like Newton-Rhapson method (Luenberger, Ye, et al., 1984).

The gradient descent optimization can also be applied for non-convex functions, but there is no guarantee that the procedure will always converge to the global minima. It is possible for the optimization procedure to converge to the local minima as different start parameters may give different solutions. One of the strategies to avoid selecting a local minima is to do many runs with different initial parameters and then select the parameters which have the smallest loss.

2.2.3 Probabilistic View of Linear Methods

We now assume that the labels, y_i, are generated from a hypothesis $h(x;\theta)$, where:

$$y_i = h(x_i;\theta) + \epsilon_i$$

where,

$$\epsilon_i \sim \mathcal{N}(0,\sigma)$$
$$x_i \sim P(X). \qquad \text{(i.i.d. assumption)}$$

The goal of the learning algorithm is to find the hypothesis (which is defined by parameters θ) that is able to explain the observed data most likely. Let the space of hypothesis be $\mathcal{H}$, that is defined by the all the valid configurations of θ, then the most probable hypothesis is:

$$\begin{aligned} h^* &= \arg\max_{h\in\mathcal{H}} P(h|D) \\ &= \arg\max_{h\in\mathcal{H}} P(D|h)P(h) && \text{(bayes rule and P(D) is independent of } h) \\ &= \arg\max_{h\in\mathcal{H}} P(D|h). && \text{(if all hypothesis are equally likely a priori)} \end{aligned}$$

This is also known as **Maximum Likelihood Estimation (MLE)** (Fisher, 1925). The probability on the right hand side can be written as:

$$L(h) = P(D|h) = P(< x_1, y_1 >, < x_2, y_2 >, \ldots |h).$$

This function is called the **likelihood**, which we now want to maximize. As the x_i's are independent and identically distributed, we have

$$\begin{aligned} L(h) &= \prod_i P(< x_i, y_i > |h) \\ &= \prod_i P(y_i|x_i;h)P(x_i). \end{aligned}$$

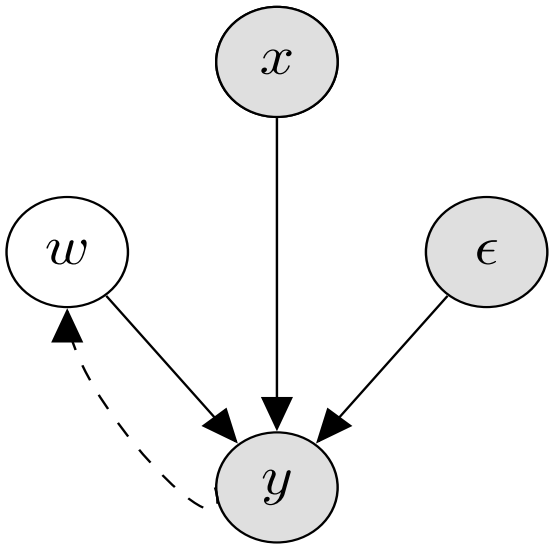


Figure 2.1: Probabilistic View

The right hand term is a product of probabilities, which is hard to optimize, so we apply the log operation on both side and reduce it to a sum:

$$\log L(h) = \sum_i \log P(y_i|x_i; h) + \sum_i \log P(x_i).$$

The second term in the equation is independent of h and does not affect the optimization, so we can remove that. The final maximization objective becomes:

$$\log L(h) = \sum_i \log P(y_i|x_i; h).$$

$P(y_i|x_i; h)$ can be modelled in different ways: one can assume it fits in a family of distributions (with parameters h) or can be assumed it is some mixture of a deterministic function and random noise.

2.2.4 Over-fitting

Over-fitting is the phenomena when the hypothesis function is powerful enough to fit the given dataset but does not generalize to other samples generated from the same

underlying distribution. It can also be thought as if the learner is memorizing the data samples. One of the ways to evaluate the performance of a learned hypothesis with respect to loss function on a dataset, is to divide the original dataset into two parts: the **training set**, on which we will learn the function's parameters and the other, **test set**, which we will use to measure how well our learned function generalizes to the samples which it did not have access in training phase. If the learnt function is over-fitting, then on the training set the error will be low but on the test set the error will be high. If the function has less capacity and it is not able to learn on training set then we refer to it as **under-fitting**. One of the ways to fix is to further divide the training dataset into two sets: training and validation. The error on the validation set is used as an estimate of the error on the test set. The goal is to find a hypothesis such that it has the least error on both the training and validation set. This procedure is also called **cross-validation**.

2.2.5 Bias and Variance

The expected mean squared error, when $x \sim P(X)$ and $h \in H$, can be written as:

$$\begin{aligned}\mathbb{E}_P[(y - h(x))^2|x] &= \mathbb{E}_P[(h(x)^2 + y^2 - 2h(x)y|x] \\ &= \mathbb{E}_P[h(x)^2|x] + \mathbb{E}_P[y^2|x] - \mathbb{E}_P[2h(x)y|x].\end{aligned}$$

If the data is being generated from a true function $f(x)$ with some added noise $\epsilon \sim \mathcal{N}(0, 1)$, then $\mathbb{E}_p[y|x] = \mathbb{E}_P[f(x) + \epsilon] = f(x)$. Let $\bar{h}(x) = \mathbb{E}_P[h(x)|x]$ denote the mean prediction of h. Then we have:

$$\mathbb{E}_P[(h(x))^2|x] = \mathbb{E}_P[(h(x) - \bar{h}(x))^2|x] + \mathbb{E}_P[(\bar{h}(x))^2]$$

$$\mathbb{E}_P[y^2|x] = \mathbb{E}_P[(y - f(x))^2|x] + \mathbb{E}_P[f(x)^2].$$

Using all the above three terms together, we get:

$$\begin{aligned}\mathbb{E}_P[(y-h(x))^2|x] &= \mathbb{E}_P[(h(x)-\bar{h}(x))^2|x] + \mathbb{E}_P[(\bar{h}(x))^2] - 2\bar{h}(x)f(x) \\ &\quad + \mathbb{E}_P[(y-f(x))^2|x] + \mathbb{E}_P[f(x)^2] \\ &= \mathbb{E}_P[(h(x)-\bar{h}(x))^2|x] + \mathbb{E}_P[(y-f(x))^2|x] + (f(x)-\bar{h(x)})^2.\end{aligned}$$

The first term $\mathbb{E}_P[(h(x)-\bar{h}(x))^2|x]$, represents the error due to the prediction of mean hypothesis from the current hypothesis trained on a sample of $X \sim P$ and represents the **variance** of the hypothesis. This represents that error which can be caused due to changing the configuration of the dataset (if it is too high, that means the learned function only works for the current batch of data and fails to generalize to unseen samples).

The second term $\mathbb{E}_P[(y-f(x))^2|x]$ is the inherent noise present in the true function generating the data. Most of the times that is usually a property of the dataset and can not be avoided.

The third term $(f(x)-\bar{h}(x))^2$ is the **bias** associated with the class of functions with respect to the true function. If the bias is high it generally means that the current hypothesis class is too simple with respect to the true function.

On a related note, over-fitting can be associated with having high variance but low bias, and under-fitting can be characterized by having high bias but low variance.

2.3 Non-linear function approximators

In general the relation between $\mathcal{X}$ and $\mathcal{Y}$ might not be linear in nature. We will now look into methods which assume $f_\theta(x)$ is a non-linear mapping. One of the ways to achieve such mapping is to pass the original x through a series of K non-linear transformations defined by ϕ_k (ϕ_k is also sometimes referred as basis functions in the literature, Bishop, 2006). We can then apply the linear methods defined above on this non-linear transformation of x, defined by the vector $< \phi_0(x), \ldots, \phi_k(x) >$.

The basis functions are given as input to the learning algorithm (can also be thought of as feature engineering) by the human. A natural step to extend this approach is to introduce parametric basis functions and include them also as the part of the learning problem. In the most simplest terms, *neural networks* can be thought of as a sequential application of such parametric basis functions, where after each application of component basis function we get a more rich and non-linear feature vector (also referred as *layer*), which is fed into the next basis function, until the end where we have regression or classification loss. In the next section, we will go into more detail of the each component of the neural networks and talk about the techniques and developing methods which are referred to as *Deep Learning* (LeCun, Y. Bengio, and G. Hinton, 2015).

2.3.1 Neural Network

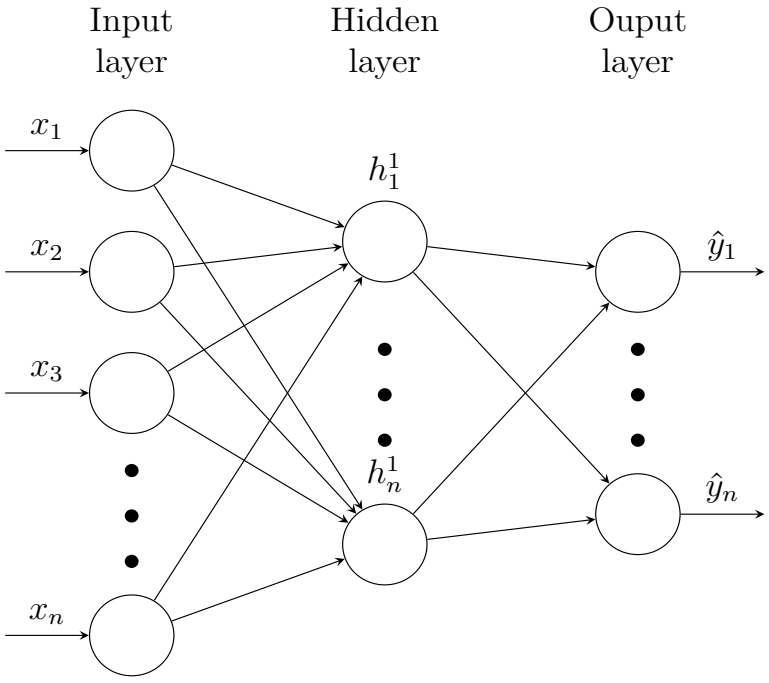
Multi-Layer Perceptron by Rosenblatt (1958) introduced Perceptron that forms the basic component of the artificial neural network[1]. A perceptron is a simple binary classifier with the form:

$$h(x; w) = sgn(b + w^T x) = \begin{cases} +1, & \text{if } (b + w^T \geq 0 \\ -1, & \text{otherwise.} \end{cases}$$

The perceptron has a linear decision boundary and thus even using a linear combination of multiple such perceptrons we can only have linear decision boundaries. In order to get nonlinear decision boundaries we can use a network of perceptron where a **layer** (a set of perceptron units) feeds its output to the next layer, with the logistic loss at the end (in case of classification). However, the gradient based optimization

[1]The name neuron and neural network because the inspiration stems from neuroscience

Figure 2.2: A Feed-forward Neural Network with one hidden layer



methods cannot be as the function is now non-differentiable (because of the step function, sgn). We now review a modified version of perceptron, which is is also called the **artificial neuron** (or network unit):

$$h(x; w) = g(b + w^T x).$$

Here, $g()$ is a differentiable function and is called the **activation function** (w is called the connection weights and b is neuron's bias). In the original Multi Layer Perceptron, layers of perceptron units with sigmoid non-linearity as the activation function, were stacked to create a network. In general, an intermediate layer (sometimes called as **hidden layer**), in a neural network with L layers can be defined as:

$$h^k(x) = g(W^k h^{k-1}(x) + b^k) \qquad (\forall k \in [1, L])$$

$$h^0(x) = x \qquad \text{(input layer)}$$

$$h^{L+1}(x) = l(W^{L+1} h^L(x) + b^{L+1}) = f(x). \qquad \text{(output layer)}$$

In the above formulation, for the last layer (*output layer*), we typically have a more specific activation function. For eg, in the case of multi-class classification, l is usually substituted with the $softmax$ function, which helps to convert the output to probabilities which can then be easily combined with the final loss function.

This kind of network architecture, where output of one hidden layer acts as the input for the next layer, is called as **feed-forward neural networks**. If all units in a layer feed into the all the units of the next layer, then the architecture is called as *fully connected* feed-forward neural network. Such multi-layered models are termed under **Deep Learning** (LeCun, Y. Bengio, and G. Hinton, 2015, Ian Goodfellow and Courville, 2016) (as the number of hidden layers increases, the network also becomes deeper).

2.3.2 Activation Functions

The most commonly used activation functions are:

- **Sigmoid** (LeCun, Bottou, et al., 1998)

  A sigmoid activation (or non-linearity) squashes the output of a neuron between 0 and 1. It is defined as:

  $$\sigma(x) = sigmoid(x) = \frac{1}{1 + \exp(-x)}.$$

  It is monotonically increasing and always positive. The partial derivative of the sigmoid function is:

  $$\sigma'(x) = \frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)).$$

- **tanh** (Collobert, 2004)

  The hyperbolic tangent (or tanh) is also monotonically increasing but instead

bounds the output between -1 and 1.

$$tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = \frac{\exp(2x) - 1}{\exp(2x) + 1}$$
$$tanh'(x) = \frac{\partial tanh(x)}{\partial x} = (1 - tanh(x)^2).$$

- **ReLU** (Glorot, Bordes, and Y. Bengio, 2011)

  The rectified linear unit (or ReLU is monotonically increasing and bounded below by 0 but has no upper bound. Using ReLUs has shown to result in sparse activations (Glorot, Bordes, and Y. Bengio, 2011).

$$ReLU(x) = \max(0, x) RELU'(x) \quad = \frac{\partial ReLU(x)}{\partial x} = \begin{cases} +1, & \text{if} x > 0 \\ 0, & \text{otherwise.} \end{cases}$$

- **softmax**

  We mentioned it briefly earlier that softmax is typically used as the output activation for the multi-class classification problems. A softmax activation results in multiple output where each output is strictly positive and sums to 1. Here, input x is a vector with K classes ($< x_1, \ldots, x_k >$), then:

$$softmax(x) = \left[\frac{\exp(x_1)}{\sum_k \exp(x_k)} \cdots \frac{\exp(x_K)}{\sum_k \exp(x_k)} \right].$$

  This has a nice interpretation where each class output can be considered as a probability for a class.

It is worth mentioning the *Universal approximation theorem* (Hornik, 1991) here, which states that a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough units. The result is also true for all the above activation functions.

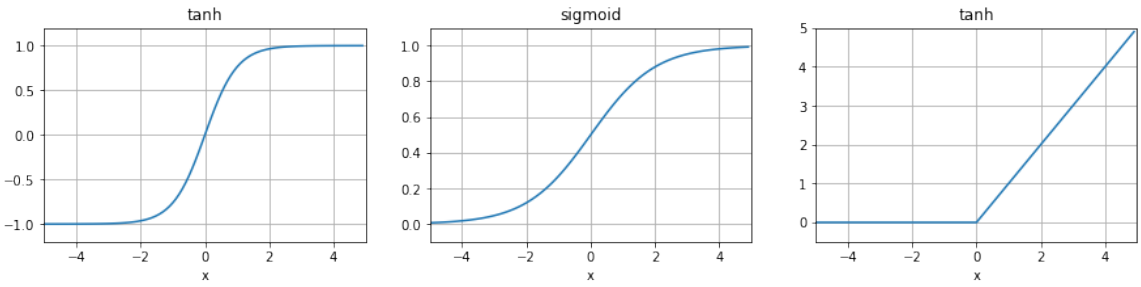


Figure 2.3: Most popular activation functions. From left to right: hyperbolic tangent ($tanh$), sigmoid (σ) and Rectified Linear Units ($ReLU$). Note that the y-axis are different for each function.

2.3.3 Learning in Neural Networks

The entire neural network can be thought of as a single function with parameters W denoting the weights and biases of all the neurons in the network. Since, the network is differentiable we can apply the gradient based update rule to learn the networks however since there are complex dependencies between different neurons we just cannot the partial derivative w.r.t. each neuron. **Back-Propagation** (Rumelhart, G. E. Hinton, Williams, et al., 1988) makes it possible to update each neuron's parameters, by taking advantage of the structure of the network, and computing gradient efficiently by applying the chain rule iteratively from last to first. The chain rule, simply states that if a variable z is dependent on variable y, which is in-turn dependent on variable x, then the partial derivative of z w.r.t. x, can be expressed as:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}.$$

Consider a loss function $\mathcal{L}$, we can now calculate the gradient of each layer's parameters (denoted by h_i), by just applying the chain rule recursively, starting from the last layer. We first take the gradient of the loss w.r.t. to last layer, $\frac{\partial \mathcal{L}}{\partial h_L}$, directly as there are no dependencies no any other variables. For the second last layer, we can get the gradient as $\frac{\partial \mathcal{L}}{\partial h_L}\frac{\partial h_L}{\partial h_{L-1}}$. Note that, we have already calculated the first term in the previous computation, and over here we are just adjusting the gradient to only account for the effect of the current layer. In this way, we can propagate

the gradient with respect to the loss function backwards through the network in a computationally efficient manner, hence the name, back-propagation. Once, we have the gradient for each layer, $\frac{\partial \mathcal{L}}{\partial h_i}$, we can apply the gradient descent to update the parameters independently.

2.3.4 Stochastic Gradient Descent

We earlier talked about Gradient Descent, but in that case we were taking the gradient w.r.t. all the samples in our dataset and then updating the weights. This kinds of gradient descent algorithm is called **Batch Gradient Descent**. As, one might think this kind of update rule becomes highly computationally inefficient as we move to bigger datasets.

On the other end of this spectrum, we can take the gradients, with respect to just a single sample and use to it update the parameters. This is known as **Stochastic Gradient Descent** (SGD) (LeCun, Bottou, et al., 1998). SGD is quite inexpensive compared to the batch and is still valid because of the i.i.d. assumption, as each sample is representative of the dataset and can provide a sample of the right signal for weight update. However, there will be now higher variance in the gradients over the updates, which can cause the weights to take a longer time to converge. As a result, there exists another update rule, which instead of doing updates on either single data-point or the entire dataset, samples k data-points form the dataset, and updates the parameters based on this mini-batch. This is known as **Mini Batch Gradient Descent** (G. Hinton, Srivastava, and Swersky, 2012).

As the number of parameters increases, the effectiveness of the above gradient update algorithms decreases as they do not take into account any information about the previously calculated gradients. The popular adaptive learning update rules which are essential for training deep networks are:

- **Momentum**

The core idea of momentum (Polyak, 1964) is that each parameter should preserve the sense of which direction is being updated in and should not deviate drastically from it, from one update to the next. More formally, each parameter maintains a momentum variable (which is initialized to 0), and updates its weight now as a exponential moving average of the momentum and the gradient.

$$\mu_0 = 0$$
$$\mu_t = \beta \nabla_\theta \mathcal{L} + (1 - \beta)\mu_{t-1}$$
$$\theta_t = \theta_{t-1} - \alpha \mu_t.$$

Here, β, denotes the decay of the effect of the momentum.

- **RMSProp**

  In RMSProp (Tieleman and G. Hinton, 2012) the gradients are normalized by the square root of the exponential moving average of the squared gradients.

$$\mu_0 = 1$$
$$\mu_t = \beta (\nabla_\theta \mathcal{L})^2 + (1 - \beta)\mu_{t-1}$$
$$\theta_t = \theta_{t-1} - \alpha \frac{\nabla_\theta \mathcal{L}}{\sqrt{\mu_t + \epsilon}}.$$

  Here, *beta* is the decay parameter as in the above case, and ϵ is a very small number (close to 0), added to prevent the denominator going to 0.

- **Adam**

  In Adam (Kingma and Ba, 2014), the idea is similar to RMSprop, however instead of exponential moving average we use the cumulative sum.

$$\mu_0 = 1$$
$$\mu_t = (\nabla_\theta \mathcal{L})^2 + \mu_{t-1}$$
$$\theta_t = \theta_{t-1} - \alpha \frac{\nabla_\theta \mathcal{L}}{\sqrt{\mu_t + \epsilon}}.$$

2.3.5 Computation Graphs

A **computational graph** (Pollack, 1990) is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Each node in the graph represents a variable. The variable can be a constant, scalar, vector, matrix or tensor. An operation is a function of one or more variables and returns a single output variable. Operations link the nodes in the graphs and are denoted by directed edges. For eg, if a variable y is the result of application of operation z to a variable x, then this can be represented as directed edge from x to y, with the label z, $x \xrightarrow{z} y$. A complex function can be described as a graph by composing many operations together, with the input nodes having no incoming edges (start of the computation) and the output nodes containing no outgoing edges (end of the computation).

In a Feed-forward network each layer can be considered as the node in the computation graph with the last layer being the final output node that computes the function represented by the network. A **forward** pass computes the output of the given input and goes from bottom to top (input nodes to output nodes). The **backward** pass computes the gradient given the loss for learning. This is back-propagation and goes from top to bottom. In this pass the gradient of each layer is reverse-composed to compute the gradient of the whole model by automatic differentiation [2].

2.4 Deep Learning

Deep learning refers to the machine learning methods that are based on learning data representations. A **Deep Neural Network** (DNN) is a feed-forward neural network with multiple hidden layers between the input and output layers. As with the feed-forward networks a DNN can also capture complex non-linear relationships and

[2] A good introduction to computational graphs and back-propagation can be found here: http://colah.github.io/posts/2015-08-Backprop/

given enough capacity, has the ability to represent any function. The extra layers in a DNN enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network (Y. Bengio et al., 2009).

In the next sections, we cover the topics that are relevant to this work with regard to deep learning techniques and architectures, but we advise the reader consult the Deep Learning textbook (Ian Goodfellow and Courville, 2016), for a thorough introduction and review of these techniques.

2.4.1 Auto-Encoders

Auto-encoders (Vincent et al., 2008), provide a way to learn some compressed representation of the data, where the network should learn to map the input data on some smaller dimensional manifold, from which it can reconstruct the original data. Essentially, an auto-encoder is a feed-forward neural network which consists of two functions:

- **Encoder**: The first part of the auto-encoder, is a feed-forward neural network which maps $f_{enc} : \mathcal{X} \rightarrow \mathcal{H}$, where dimensionality of $\mathcal{H}$ is lower than $\mathcal{X}$. This function essentially *encodes* (or compresses) the information about the original data sample in a lesser space.

- **Decoder**: The second part, is a function $f_{dec} : \mathcal{H} \rightarrow \mathcal{X}$ which learns to *decode* the representation in $\mathcal{H}$ to the original space.

The loss function in this case is the reconstruction error between the original sample x and the reconstructed sample $\hat{x}$ (it can be either mean square error for real valued data or cross-entropy for binary data). Both encoder and decoder can be feed-forward networks with multiple hidden layers. An interesting aspect of auto-encoders is that, once we have successfully learnt an encoding of the input distribution, we can

take the learned encoder and reuse it as the initial layer for the subsequent tasks. Since the encoder is able to project the input data to a lower dimensional space, it should be easier for the rest of the network to work with this already learned lower dimensional representation (Rasmus et al., 2015, Y. Bengio, Courville, and Vincent, 2013).

2.4.2 Recurrent Neural Networks

Recurrent Neural Networks or (RNNs, Rumelhart, G. E. Hinton, Williams, et al., 1988) are neural networks for processing sequential data. Unlike Feed-forward networks, which are cyclic, RNNs have (at least one) cyclic path or dependencies between the units. As we are now dealing with sequences, the data has a temporal dimension to it. The dataset $\mathcal{X}$ is now composed of i.i.d. sequences, where each sequence can have a variable length and consists of dependent datapoints, denoted by x_t. As we are dealing with a sequential task, we need a mechanism to keep track of the sequence we have observed so far, some sort of memory or register. One of the ways to keep the information about the previous observations is use the notion of parameter sharing across time. This not only helps to generalize across different length sequences but also becomes important when a specific piece of information can occur at any position within the sequence. As a result, the state of RNN can be defined using the following equation:

$$h_t = f(h_{t-1}, x_t; \theta).$$

The state h_t is the value of the hidden units of the network and can also be thought of as some non-linear function of the observations so far, $h_t = F(x_0, x_1, \ldots, x_t)$. The recurrent nature of the RNNs also makes them powerful generative models. They can also be thought of as directed graphical models which are conditioned on their observations. Some examples of design architectures for recurrent neural networks include:

- RNNs that produce output at each time step and have recurrent connections between hidden units with adjacent time steps only. Eg, a sequence labelling task, where each element of the sequence needs to be assigned a label, can be modelled efficiently using this type of RNN architecture.

- RNNs that read the entire sequence and produces output only at end (eg for sequence classification tasks).

- RNNs that read the entire sequence and then generate a new sequence which is conditioned on the entire previous sequence. These are also called a *Sequence-to-sequence* models and used in extensively machine translation. More details about such architecture is covered in Chapter 4.

2.4.3 Back Propagation Through Time

The parameter sharing across hidden units in time makes the computational graph cyclic in nature as the output of the recurrent operation is the input node itself, both being the state of the RNN. Assuming the length of a sequence is τ, we can unfold the graph in time, by expanding the recurrent equation:

$$\begin{aligned} h_t &= f(h_{t-1}, x_t; \theta) \\ &= f(f(h_{t-2}, x_{t-1}; \theta), x_t; \theta) \\ &\vdots \\ &= f(f(\ldots(f(h_0, x_1; \theta)), \ldots, x_{t-1}; \theta), x_t; \theta). \end{aligned}$$

This new unfolded graph can now be considered as a fixed length directed acyclic graph. Gradients can now be obtained by applying chain rule on unrolled graph and **back-propagated through time** (BPTT, Rumelhart, G. E. Hinton, Williams, et al., 1988). (Williams and Zipser, 1989b). Assuming length of the sequence is τ, the forward pass does computation forward in time, $t = 0 \rightarrow \tau$, and the back-propagation goes backwards in time, $t = \tau \rightarrow 0$.

The above method becomes ineffective for very long sequences or in the case of online learning, where there is just a stream of incoming data without any end. In such scenarios, we divide the sequence in chunks of a fixed size, k, and perform forward and backward pass only on those chunks. It is not equivalent to considering them as independent sequences because the hidden layers between the chunks are still dependent on the previous chunks. Since the gradients are not allowed to go backwards all the way, this is also referred as **Truncated Back Propagation Through Time** (Williams and Peng, 1990). There are other ways to the update weights in RNNs, most notably, Real Time Recurrent Learning (RTRL) (Williams and Zipser, 1989b).

The optimization is still based on the Maximum Likelihood objective (just like in the case of a feed-forward network). However, in cases where the hidden step at time-step t is dependent on the previous label/prediction y_t, we employ a procedure called **Teacher Forcing** (Williams and Zipser, 1989a). During training, the actual label from the previous time-step y_{t-1} is fed to hidden state h_t, however during test/generation time the prediction from the previous time-step $\hat{y}_{t-1}$ is fed the hidden state. This can in practice lead to compounding errors and is an active area of research resulting in few techniques such as Scheduled sampling (S. Bengio et al., 2015) to counter it.

One of the bigger challenges of optimizing RNNs is the **vanishing gradient problem**, i.e., the back-propagated gradients in RNNs tend to either vanish ($\rightarrow 0$) or explode ($\rightarrow \infty$) as the sequence length increases (Y. Bengio, Simard, and Frasconi, 1994). This makes optimization in RNNs significantly harder than regular deep networks (we cannot have layer-wise re-normalization as the parameters are shared). The vanishing gradient problem arises because the gradient at any step is a multiplicative function of the gradient of the time-steps in the future (because of the chain rule). A number of techniques have been advised to prevent this, the major ones being:

- *Gradient clipping*: This is a technique to avoid exploding gradients (Pascanu,

Mikolov, and Y. Bengio, 2013) by defining a upper bound on the gradient value according to some threshold defined as a hyper-parameter. A common way to perform gradient clipping is to normalize the gradients of a parameter vector when its L2 norm exceeds a certain threshold, $gradients^{(clipped)} = gradients \times \frac{threshold}{\|gradients\|_2}$

- *Gated Loops*: These includes introducing specialized gated units, like Long-Short Term Memory (LSTMs, Hochreiter and Schmidhuber, 1997) and Gated Recurrent Units (GRUs, Cho, Van Merriënboer, Gulcehre, et al., 2014), as a mechanism that learns the control on the information flow through the RNN state.

- *Skip connections*: The idea is to add direct connections from variables from the past to variables in the present (T. Lin et al., 1996). This helps to retain more information across time-steps and capture longer dependencies.

2.4.4 Gated Recurrent Units

LSTMs (Hochreiter and Schmidhuber, 1997) introduce the notion of gates for RNNs, where a gate creates a control over the paths through which the gradients can flow over time, conditioned on a context. The gates control the state of the RNN, and determine, for any given context, if the information from the corresponding time-step is worth adding to the current state or not. In simplest terms, gates have explicit control units which work to create paths through which relevant gradient can be propagated. These control units over the hidden state can also be thought of as update or write operations over the state. LSTMs have shown to be successful in preserving long-term dependencies across varieties of tasks (Ian Goodfellow and Courville, 2016).

In this work, instead of working with LSTMs, we work with a simpler variant of LSTMs called the Gated Recurrent Unit or GRU (Cho, Van Merriënboer, Gulcehre, et al., 2014). A GRU has two main gates, update gate, u_t, to update (replace or copy)

the state of the cell and reset gate, r_t, to select which parts of the state get used to compute the next target state. Both the gates are conditioned on the input (x_t) and the hidden state (h_t). The gates are defined as follows:

$$u_t = \sigma(b_u + W_u h_t + U_u x_t)$$
$$r_t = \sigma(b_r + W_r h_t + U_r x_t).$$

In the above equations b_u, W_u, U_u represents the weights of the update gate and similarly b_r, W_r, U_r represent reset gate parameters. The cell state is updated using both the update and reset gates simultaneously, within a single operation as:

$$h_t = u_{t-1} h_{t-1} + (1 - u_{t-1})\sigma(b + U x_{t-1} + W r_{t-1} h_{t-1}).$$

The first term in the above equation selects which part of the previous state to preserve. The second term selects the parts to be overwritten and updates them according to another non-linearity based on the region of hidden state selected by the reset state and the input.

The reason we chose to work with GRUs instead of LSTMs is because GRUs have fewer parameters compared to the LSTMs which in turn leads to fewer computations. Unlike LSTMs, in GRUs we directly update the hidden state, which also results in fewer computations. All these factors can lead to a higher training speed without having much impact in loss of information.

2.4.5 Dropout

Srivastava et al., 2014 introduced the idea of dropout as a regularization technique to prevent the neural networks from over-fitting. The core idea of dropout is to hamper the ability of a neural network to memorize (over-fit perfectly without learning any generalizable representation) by stochastically dropping the activation of hidden units. Each unit's activation is set to 0 with some probability p. This forces the hidden unit

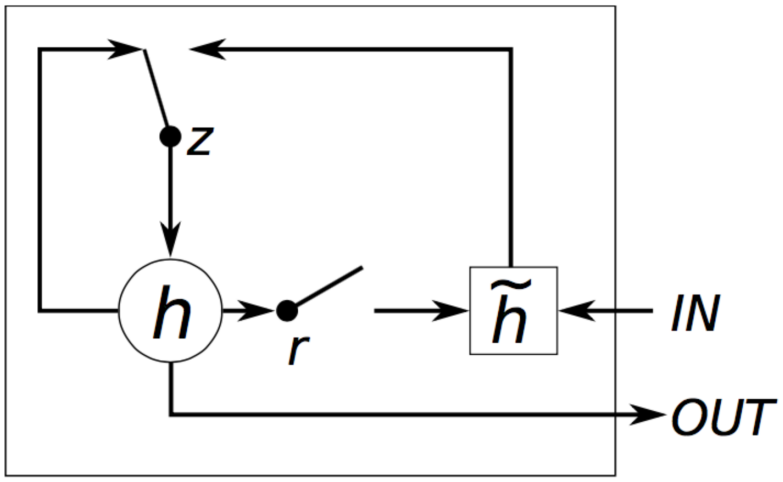


Figure 2.4: GRU gating architecture. r and z are the reset and update gates, and h and $\tilde{h}$ are the current activation and the candidate activation. IN refers to the current input to the cell and OUT refers to current prediction. Figure taken from Chung et al., 2014

.

to take into account another hidden unit's information and adapt according to that. Each layer within a network can have a different dropout probability at the training time. Although this adds some redundancy in the network, as now it has to be more robust, but at the same time it also prevents any dependency on some specific units and forces the network to learn a more distributed representation.

2.4.6 Automatic Differentiation

Automatic differentiation packages provide a programming framework which allow us to represent a network as a symbolic computation graph. The module can then take the gradient with respect to a loss function automatically by applying back-propagation. We use Theano (Bergstra et al., 2010, Bastien et al., 2012) in this work, but there now exists other popular frameworks such as TensorFlow (Abadi et al., 2016) and Torch (Collobert, Kavukcuoglu, and Farabet, 2011). Most of time when we are optimizing the neural networks, we end up doing essentially a lot of matrix multiplications. CPUs (Central Processing Unit) are fast and good for sequential tasks,

however we can parallelize most of these matrix multiplications on a specialized hardware such as GPU (Graphical Processing Unit). It has been shown that using a GPU with specialized library can lead up to $50 - 100\times$ boost for training deep networks[3]. An added advantage of using these libraries is that they provide an abstraction over lower level hardware libraries such as CUDA (Nvidia, 2010) and CuDNN (Chetlur et al., 2014).

[3]https://github.com/jcjohnson/cnn-benchmarks

3

Reinforcement Learning

Reinforcement Learning (RL) refers to the learning problem where an agent should learn a behaviour to optimize some numerical performance measure that expresses the long term goal. The predictions of the agent also have the effect of changing the future state of the environment (the observed data). Thus, there is a temporal relation between the observed data which is turn dependent on the learner's predictions.

A typical RL setting is depicted in figure 3.1. At any given time-step, t, the **agent** (also called learner or controller) is in some state of **environment** (or system) which it can observe, s_t, and interact with by taking an action a_t. Once the agent takes the action it receives feedback from the environment in terms of numerical value called reward, r_t. The environment then makes a transition to a new state, s_{t+1}, and the cycle is repeated until some terminal state is reached. The task is to learn to interact with the environment in a way that maximizes the total reward over time.

In this chapter we will only cover the necessary RL algorithms that the reader needs to know to understand this work. For a proper introduction and extensive understanding we advise the reader to refer to the textbooks by Sutton and Barto (1998) and Szepesvári (2010)[1].

[1]The notation used in this chapter is borrowed from http://videolectures.net/deeplearning2017_pineau_reinforcement_learning/ and http://videolectures.net/deeplearning2017_abbeel_policy_search/

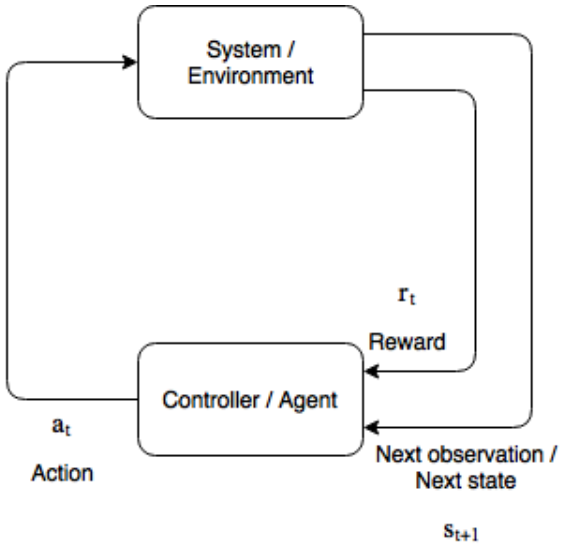


Figure 3.1: RL framework

3.1 Markov Decision Process

We assume the environment is stochastic and has markovian property, i.e. the distribution over the future states depends only on the current state and action. Markov Decision Processes (MDPs, Bellman, 1957) provide a framework to model such kind of stochastic problems. A MDP $\mathcal{M}$ is defined by:

- $\mathcal{S}$: the set of states (can be infinite).
- $\mathcal{A}$: the set of actions which the agent can take (can be infinite).
- $\mathcal{R}(s, a)$: the reward function ($r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$) which determines the reward that the agent gets on executing action a in state s.
- $\mathcal{P}(s, a, s')$: the transition function which defines the probability of next state, s', when the agent is in state s and takes action a, i.e. $P(s'|s, a)$. It is also called the dynamics model of the environment.
- γ: the discount factor, $\gamma \in [0, 1]$, which tells how much importance is given to the rewards in the future vs the present.

- $\mu(s)$: the initial state distribution, $p(s_0)$, in which the agent can be at time-step $t = 0$.

Because of the markovian property:

$$P(s_t|s_{t-1}, a_{t-1}) = P(s_t|s_{t-1}, a_{t-1}, \ldots, s_1, a_1, s_0).$$

A state is said to be *terminal* if the MDP ends when the agent lands in that state. If such terminal states are present, the MDP (or task) is said to be *Episodic*. In the absence of terminal states, the task is referred to as *Continuing* (or continual learning). A sequence of < state, action, reward > tuples is described as **Trajectory**, τ. The goal of the problem is to maximize the total reward over the trajectory.

The discount factor can be thought of as the probability of the agent dying, i.e., with $(1 - \gamma)$ probability the episode can end or the agent dies. The case $\gamma = 1$ denotes that all the rewards (short-term or long-term) have equal importance because the agent never dies and thus every reward is equally important. When γ is close to 0, then the agent pays more importance to short-term rewards as there is a larger probability it might die in the future. The goal of the problem is to maximize the total reward over the trajectory.

3.2 Policies

The behavior of the agent is defined by a **policy**, π, which represents the distribution over actions the agent can take in a given state.

$$\pi(a|s) = P(a_t = a|s_t = s).$$

If the agent samples the action from this distribution then the agent is said to be following a *stochastic* policy. If the policy is *deterministic*, i.e., the agent will always take a particular action in a particular state, then the policy can also be thought of as function that maps states to actions, $\pi : \mathcal{S} \rightarrow \mathcal{A}$. If Π represents the set of all

possible policies, then the goal is to find the policy which gets the maximum total reward among all the possible policies.

3.3 Value Functions

In order to evaluate how good a particular state is under a policy, we define the value function ($V^\pi : \mathcal{S} \to \mathbb{R}$), which represents the expected return when the policy is followed from that state:

$$V^\pi(s) = \mathbb{E}_\pi[r_t + r_{t+1} + \cdots + r_T | s_t = s].$$

The cumulative expected reward is also sometimes called the *return*. Since the policy does not change over the course of the trajectory, the value function can be recursively defined using Bellman's equation (Bellman, 1957):

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_t] + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s') \\ &= \sum_{a \in \mathcal{A}} \pi(s,a) \mathcal{R}(s,a) + \gamma \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') V^\pi(s') \\ &= \sum_{a \in \mathcal{A}} \pi(s,a) \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') V^\pi(s') \right]. \end{aligned}$$

The **state-action value function** ($Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is also defined in a similar way, but assumes that a particular action is taken at the first state.

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') \left[\sum_{a \in \mathcal{A}} \pi(s',a') Q^\pi(s',a') \right].$$

The **optimal value**, $V^*(s)$, of state $s \in \mathcal{S}$ gives the highest achievable expected return when the process is started from the state s. In similar fashion, one can also define the **optimal state-action value**, $Q^*(s,a)$, which defines the maximum expected return possible if the process starts at state s, and the first action chosen is

a. They are connected as follows:

$$V^*(s) = \sup_{a \in \mathcal{A}} Q^*(s, a)$$

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') V^*(s').$$

The optimal value function is:

$$V^*(s) = \sup_{\pi \in \Pi} V^\pi(s).$$

An action which maximizes $Q(s, .)$ for some s is called greedy with respect to Q in state s. A policy that selects greedy actions with respect to Q in all states is defined as a **greedy policy** with respect to Q. A policy which selects the greedy action w.r.t. Q with probability $(1 - \epsilon)$ and selects a random action with probability ϵ in all states is called an **ϵ-greedy policy**. Any policy that achieves V^* is called the **optimal** policy.

Note that a greedy policy with respect to Q^* is optimal. Similarly, knowing $V^*, \mathcal{R}, \gamma, \mathcal{R}$, we can get the optimal policy, π^*:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} (\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') V^*(s')).$$

The next obvious question is how to get the optimal V^* or π^*. From the recurrent nature of the Bellman equation mentioned above, one can directly try to forumlate it as a dynamic programming problem (Bellman, 1957). Two of the popular dynamic programming based MDP solver algorithms are **Value Iteration** and **Policy Iteration**. For actual details and theoretical results of these methods (and more such techniques for solving the MDPs) we advise the reader to refer to the textbook by Puterman, 2014.

3.4 Learning Value Functions

The classical MDP solver algorithms assume that we know the transition and reward functions for the process. However in many domains, the agent does not have access

to the underlying dynamics of the environment. We want the agent to explore the environment, adjust its behavior to exploit the environment in order to maximize the reward. We still want to use the notion of value functions and policy in order to define the behavior and utility, but now we want to learn them without any information about the environment. Since there is not any information about the environment, *exploration* plays an important role and the agent needs to include it along with the *exploitation* objective. A typical agent behavior can be described as a continuous loop of acting based on current estimates, observing the outcomes and updating the current beliefs based on the observation.

3.4.1 Monte Carlo Methods

The value function is defined as the expectation of returns when the process is started from a given state. One way to estimate this value is to compute the average over multiple runs starting from the given state by using the *Monte Carlo (MC, Metropolis and Ulam, 1949)* estimates of the independent runs to estimate the value function. If $U(s_t)$ is the empirical estimate of the value function, then the value function estimator can be updated in a gradient based iterative update rule style as:

$$V^{k+1}(s) = V^k(s_t) + \alpha(U(s_t) - V^k(s_t)).$$

One of the major limitation of the MC methods is that the variance of the returns can be very high because of sampling which leads to poor estimates. These methods also cannot be applied in a closed loop system (i.e. task needs to be episodic). Though MC methods provide an unbiased estimate but most of the times it is characterized by high variance.

3.4.2 Temporal Difference Learning

Temporal Difference Learning (Sutton, 1988) provides an alternate view of learning value functions using predictions of estimates as targets during the training as a form

of *bootstrapping*. Using the Bellman equation, we can write the value function at time-step t, when the agent goes from state s to s_{t+1} by taking $a_t \sim \pi(a_t|s_t)$ and observes reward, r_t, and next state, s_{t+1} as:

$$V^{\pi}(s_t) = r_t + \gamma V^{\pi}(s_{t+1}).$$

Since we are now estimating both $V^{\pi}(s_t)$ and $V^{\pi}(s_{t+1})$, there will be some estimation error on both sides of this equation. We denote this difference between values of states corresponding to successive time steps as the *Temporal Difference (TD)* error (δ_t). The error is only considered over next immediate step predictions, and as a result it is also known as the TD(0) error.

$$\delta_t = |r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)|.$$

As the error depends on the estimated value function, the algorithm uses bootstrapping, using its predicted estimates to improve the estimate predictor. The main difference with the MC style learning is that now the agent can now also optimize to minimize the TD error in addition to just estimating the value function. The new update rule takes a form similar to:

$$V^{k+1}(s_t) = V^{k}(s_t) + \alpha\delta_t, \quad \forall t = 0, 1, \ldots$$

There also exists a multi-step version of TD learning, called TD(λ), which unifies both TD(0) and MC methods (Sutton, 1988).

3.4.3 Function Approximation

Until now, we made no assumptions on the structure of value function or policy, as we were in a tabular setting, i.e., we were calculating the value and policy for each state and action and storing it in memory. When the number of states is too big (or continuous), it becomes impossible to keep track of every possible state and update them. In such a scenario, instead of tracking values for each state, we can

learn a parametric function to give an approximate estimate of the true value function ($V^{\pi}(s) \approx V^{\pi}(s;\theta)$). The function used for modelling can be either linear or non-linear.

3.4.4 Q - Learning

Q-Leaning (Watkins and Dayan, 1992) is a method to directly estimate the optimal state-action values using the TD error. On observing transitions of the form $< s_t, a_t, s_{t+1} >$, the TD error variant for Q-Learning is defined as:

$$\delta_t = r_t + \max_{a \in \mathcal{A}} \gamma Q(s_{t+1}, a) - Q(s_t, a_t).$$

Note that over here the samples are collected under some policy, but we are evaluating the state-action values using a greedy policy. This type of setting, when we are learning about one policy (greedy w.r.t. optimal Q-values), while following another (often more exploratory scheme) is called **off-policy learning**.

The update rule for the Q-values takes the form:

$$Q^{k+1}(s_t, a_t) = Q^{k}(s_t, a_t) + \alpha(\delta_t), \quad \forall t = 0, 1, \ldots$$

In the above equation, α denotes the learning rate and δ_t denotes the TD error for Q-value estimates as defined above.

3.4.5 Deep Q-Learning Network

Q-Learning can be used with non-linear function approximation by having a parametric estimate of the optimal state-action value function ($Q(s, a; \theta) \approx Q^{*}(s, a)$). Deep Q-Networks (DQNs, V. Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015) combine deep learning based function approximator with a variant of Q-Learning to successfully learn policies directly from the pixel space and achieve human level control on Atari games (Bellemare et al., 2013). V. Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015 use

a neural network with parameters θ to estimate the optimal state-value function and referred to it as Q-Network[2].

Unlike in the regular supervised training for learning for deep networks, now our data is temporally correlated. It does not follow the i.i.d. assumption anymore and as a result we cannot use the methods from the supervised learning setting anymore without employing some additional techniques. One of the ways to break the correlation in the observations is to use **Experience Replay** (L.-J. Lin, 1993). For experience replay, the agent's experiences at each time step, $e_t =< s_t, a_t, r_t, s_{t+1} >$, are stored in a fixed size memory $\mathcal{D}$, also called the experience buffer. During learning, samples (or mini-batches) of experiences are uniformly sampled from the the experience buffer.

Another technique used in DQNs is the *target networks* (sometimes also referred as frozen networks). The term $r_t + \max_{a'} Q(s_{t+1}, a')$, in the TD error is called the target. Compared to the supervised learning case, where the targets are fixed, the targets in this case depend on the network weights. As a result, the parameters from the previous iteration are held fixed for the target estimation when optimizing the current Q-network. The target network are only periodically updated, thereby reducing the correlations in the targets.

The Q-learning update at an iteration i can be represented as minimizing the TD error, which results in the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2\right].$$

In the above equation θ_i represents the parameters of the Q-network at iteration i, and θ_i^- represents the parameters of the target network. The target parameters are updated with Q-network parameters every fixed number of steps and are held

[2] In their work, they used Convolutional Neural Networks (CNNs, LeCun, Bottou, et al., 1998) as the function approximator. CNNs are translation-invariant neural networks to learn the spatial features in the images. Since we are not using CNNs in this work, we will not be going into much detail. A nice introduction to CNNs can be found here: http://cs231n.github.io/convolutional-networks/

constant between the updates. The gradient of the above loss then becomes:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)\right].$$

The algorithm for Q-learning with experience replay and target networks is described in Algorithm 2.

Initialize Replay buffer $\mathcal{D}$ to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$

for *episode*=$1, \ldots, M$ **do**

Get the initial state the agent is in, s_1

for $t = 1, \ldots, T$ **do**

With probability ϵ select a random action a_t

else, $a_t = \arg\max_a Q(s_t, a; \theta)$

Execute a_t and observe s_{t+1} and r_t

Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$

Sample mini-batch of experiences (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$

Set $y_j = \begin{cases} r_j, & \text{if episode ends at } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-), & \text{otherwise.} \end{cases}$

Perform gradient descent with the loss $(y_j - Q(s_{j+1}, a_j; \theta))^2$ with respect to θ

Every K steps, update $\theta^- = \theta$

end

end

Algorithm 2: Deep Q-learning with experience replay and target networks

3.5 Learning Policies

Sometimes, we might be directly interested in learning an optimal policy without the need for learning a V or Q function first and then selecting actions based on that[3]. If the policy is parameterized by θ (often as a stochastic function, $f_\theta : \mathcal{S} \rightarrow \mathcal{A}$, denoted by $\pi_\theta(a|s)$), we are interested in finding the parameters which maximize the expected return.

$$\max_\theta J(\theta) = \max_\theta \mathbb{E}[\sum_t r_t | \pi_\theta]$$

where,

$$a_t \sim \pi_\theta(s_t)$$
$$r_t = \mathcal{R}(s_t, a_t).$$

This can also be thought of as searching for the optimal policy in the family of policies characterized by θ, and hence such methods are also called **Policy Search** methods.

3.5.1 Likelihood Ratio Policy Gradient

Unlike supervised learning, we cannot take the gradient of the objective directly with respect to policy parameters as the final objective is not deterministic because of the sampling operation. **REINFORCE** or Likelihood ratio policy gradient (Williams, 1992) is a policy search method which allows to compute an unbiased estimate of the gradient and uses it for searching for the policy in the direction of the objective. Let τ denote the trajectory (of length T) which the agent takes under the policy π_θ, the

[3] In some scenarios, learning a policy might be easier than learning the value function. In case of Q function this is true when the action space is very big (continuous). V function on the other hand does not provide direct information about actions.

objective can be written as:

$$\begin{aligned} J(\theta) &= \mathbb{E}\left[\sum_{t=0}^{T} \mathcal{R}(s_t, a_t); \pi_\theta\right] \\ &= \sum_{\tau} \mathcal{P}(\tau;\theta)\mathcal{R}(\tau). \end{aligned}$$

Taking gradient with respect to θ :

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} \mathcal{P}(\tau;\theta)\mathcal{R}(\tau) \\ &= \sum_{\tau} \nabla_\theta \mathcal{P}(\tau;\theta)\mathcal{R}(\tau) \\ &= \sum_{\tau} \mathcal{P}(\tau;\theta)\frac{\nabla_\theta \mathcal{P}(\tau;\theta)}{\mathcal{P}(\tau;\theta)}\mathcal{R}(\tau) \\ &= \sum_{\tau} \mathcal{P}(\tau;\theta)\nabla_\theta \log \mathcal{P}(\tau;\theta)\mathcal{R}(\tau). \end{aligned}$$

The Monte Carlo estimate of the gradient for m trajectories under policy π_{theta} can be written as:

$$\nabla_\theta J(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log \mathcal{P}(\tau^{(i)};\theta)\mathcal{R}(\tau^{(i)}).$$

For a trajectory i, the gradient with respect to the dynamics model can be represented as:

$$\begin{aligned} \nabla_\theta \log \mathcal{P}(\tau^{(i)};\theta) &= \nabla_\theta \log\left[\prod_t \mathcal{P}(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)})\pi_\theta(a_t^{(i)}|s_t^{(i)})\right] \\ &= \nabla_\theta\left[\sum_t \log \mathcal{P}(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) + \sum_t \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\right] \\ &= \nabla_\theta \sum_t \log \pi_\theta(a_t^{(i)}|s_t^{(i)}). \end{aligned}$$

The final gradient estimate over m samples now becomes:

$$\nabla_\theta J(\theta) \approx \frac{1}{m}\sum_{i=1}^{m}\left(\sum_t \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})\right)\left(\sum_t r_t^{(i)}\right).$$

The gradient now does not depend on either the reward function or the dynamics function. This means that we do not need to any make assumption about $\mathcal{R}$ (it can be non-differentiable or discontinuous) and we can still use the observed rewards

directly. Since the gradient is multiplied with the observed rewards, it can also be seen as if the gradient is scaled with the value of the rewards observed over a path. This increases the probability of trajectories that have positive rewards and vice versa for negative rewards. Note that this method just changes the likelihood ratio of the paths observed according to rewards obtained over them, hence the name.

3.5.2 Variance Reduction

The gradient estimate defined above is unbiased but has high variance because the estimate is collected over multiple trajectories and over all the time-steps within a trajectory. The variance further increases as the length of the trajectory increases or the action space increases. This is because the effect of a particular action or trajectory on the expected gradient will decrease as the number of possibilities increases. For example, if the reward from the environment is always positive, then the algorithm will try to increase the probabilities of any path which has non-zero reward. One way to reduce the variance of the estimator is by introducing a control variate (Greensmith, Bartlett, and Baxter, 2004). Williams, 1992 introduced a simple control variate in the form of **baseline** b:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log \mathcal{P}(\tau^{(i)}; \theta)(\mathcal{R}(\tau^{(i)}) - b)$$

where,

$$\begin{aligned} b &= \mathbb{E}[\mathcal{R}(\tau)] \\ &\approx \frac{1}{m} \sum_{i=1}^{m} \mathcal{R}(\tau^{(i)}). \end{aligned}$$

As long as $\nabla_\theta b = 0$, the gradient estimates remain unbiased. Therefore we can use any function which is not dependent on the actions. For eg, value function is also another popular choice of baseline, which results in:

$$b(s_t) = \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \cdots + r_T].$$

Now, the gradient has the term $\mathcal{R}(s_t, a_t) - b(s_t)$, which increases the log probability of an action with regards to how much better its returns are compared to the expected return. The final REINFORCE algorithm is given in Algorithm 3.

Initialize policy π_θ with random weights θ

Initialize baseline b

for *iteration*=$1, 2, \ldots$ **do**

Collect a set of trajectories by executing the current policy (π_θ)

for *each time-step* t **do**

compute return, $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ a_t

compute return with baseline, $\hat{A}_t = R_t - b(s_t)$

end

Update the baseline by minimizing $\|b(s_t) - R_t\|^2$ averaged over all time steps and trajectories

Update the policy by using the policy gradient estimate, $\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t$ averaged over all time steps and trajectories

end

Algorithm 3: REINFORCE (Williams, 1992)

There are other policy gradient methods which have not considered in our work, most notably the *Actor Critic* methods (Sutton and Barto, 1998), which learn policies like REINFORCE but also maintain another estimator for the value function using TD learning.

4

Machine Translation

The task of building a system that can translate from one language to the other is called Machine Translation (MT). Machine Translation started in the 1950s with the word-to-word replacement using bilingual dictionaries (Sheridan, 1955). By 1990s, corpus-based MT systems were being developed which use a dataset containing parallel sentences of sentence pair in both the languages (Brown et al., 1993) to learn the statistical translation models without using bilingual dictionaries or pre-defined set of rules. In the machine learning context, we are interested in a system that can learn a mapping of sentences from a source language to a target language. The dataset for learning consists of sequences in source language mapped to sequences in target language:

$$\mathcal{D} = \{(X^{(1)}, Y^{(1)}), \ldots, (X^{(N)}, Y^{(N)})\}$$

where, $\forall i$

$$X^{(i)} = \{x_1^{(i)}, x_2^{(i)}, \ldots, x_{T^{(i)}}^{(i)}\}$$
$$Y^{(i)} = \{y_1^{(i)}, y_2^{(i)}, \ldots, y_{T'^{(i)}}^{(i)}\}.$$

Vocabulary V of a language is a set of all words defined in the language. The total number of words in a language is denoted by the size of the vocabulary, $|V|$. Each element in a sequence for a language (x_j) can only take one value from that language's vocabulary V_x. Each sequence in the dataset represents a sentence in the

corresponding language. An element in a vocabulary (each word) can be defined by a unique identifier which is called *token*. Words are converted to labels or tokens in a pre-processing step and a dictionary of token to word mapping and vice versa is maintained.

As there can be many valid translations for a given sentence, the function we are interested in learning should return a probability distribution over sentences. The function should be able to handle sequences of variable length as the sentences in the source and target languages can be of different lengths. To denote the end of sequence, an end of sequence token (*<eos>*) is also added to all sequences as another pre-processing step.

4.1 Language Modelling

Language Modeling (LM) refers to building a model that can estimate how likely is a given sentence. Given a dataset of sentences, a Natural Language Model learns the statistics of how likely a sequence of words are to go together. This allows the model to generate sentences that have similar structure as the training dataset. Given a sentence $(x_1, x_2, \ldots, x_T)$, we are interested in estimating the probability distribution over this sentence.

4.1.1 N-gram language models

As the set of number of unique sentences possible is exponential in the size of the vocabulary, a valid assumption would be to consider that a current word is only dependent on the previous words, i.e.,

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_{i<t}).$$

Classical statistical language models employ the n-th order Markovian assumption, i.e., the current word is only dependent on the past n words in the sequence (Rosenfeld,

2000). This allows us to write the probability of a sentence as:

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_{t-1}, \ldots, x_{t-n}).$$

Due to the markov assumption these kind of language models are also known as n-gram language models. Consider a 2-gram language model, where the assumption is that the current word is dependent on only the previous two words. In order to build such a model, we need to first collect the statistics of all the triplets of the words occurring together in the dataset and store them. The biggest disadvantage of these models is that as n increases, the data sparsity problem becomes accute. Another disadvantage is that these models often fail to generalize because they treat each word as a separate entity and as such cannot consider relations between similar words.

4.1.2 Neural Language Model

The core idea of a Neural Language Model (Y. Bengio, Ducharme, et al., 2003, Morin and Y. Bengio, 2005, A. Mnih and G. E. Hinton, 2009) is to use a parametric estimator along with distributed representations of words (Mikolov et al., 2013). Instead of treating each word as a unique labelled entity, the goal is to learn a distributed vector representation that captures syntactic and semantic word relationships and use them for predicting the probability of the next word.

$$p(x_t | x_{t-1}, \ldots, x_{t-n}) = f(x_t | x_{t-1}, \ldots, x_{t-n}; \theta).$$

Each word in the n-gram context is represented in a one-hot encoding vector, $x_i \in \mathbb{R}^{|\mathbb{V}|}$, where V is the size of the vocabulary. This one-hot representation is then passed through an **embedding layer**, which is essentially a matrix that projects the one-hot representation in a low-dimensional continuous space ($\theta_{emb} \in \mathbb{R}^{d_{emb} \times |V|}$). This continuous representation is then passed as input to non-linear hidden layers, which in the end produce the probabilities over the vocabulary using the softmax activation. The entire network is differentiable and can be trained using back-propagation to minimize the cross-entropy loss.

The Markovian assumption is still a limiting factor in the above case, where we are taking n previous words and concatenating them for generating the next word probability. In order to remove the markovian assumption, Y. Bengio, Ducharme, et al., 2003 propose using an RNN to map a variable length context to a fixed smaller dimensional space. If the input to the RNN is a sequence of words, the state of the RNN at time-step t should be able to capture the information until that time-step, which can be used to build a language model as follows:

$$p(x_{t+1}|x_1, \ldots, x_t) = f(h_t; \theta_{out})$$

where,

$$h_t = g(h_{t-1}, x_t; \theta_{rec}).$$

Here g is a non-linear activation function like $tanh$ and f is the softmax activation function. The parameters of the entire network are $[\theta_{emb}, \theta_{rec}, \theta_{out}]$. The entire network consists of three components:

- **Embedding Layer**, θ_{emb} : Transforms the one-hot encoded words into continuous smaller dimensional space for learning distributed representations.
- **Encoder**, θ_{rec} : This component encodes the information about the sequence so far by updating the state of the RNN. The state acts as a memory and should be able to capture any relevant information from the previous time-steps.
- **Decoder**, θ_{out} : This part of the network uses the state of the RNN to learn the probability distribution over the next word. The next word can be sampled from this learnt distribution.

Given a set of N sentences $\{(x_1^{(1)}, \ldots, x_{T^{(1)}}^{(1)}), \ldots, (x_1^{(N)}, \ldots, x_{T^{(N)}}^{(N)})\}$, with lengths $\{T^{(1)}, \ldots, T^{(N)}\}$, the optimization objective is to maximize the negative log-likelihood

or minimize the cross-entropy loss:

$$\begin{aligned}\min J(\theta_{emb}, \theta_{rec}, \theta_{out}) &= \min_{\theta_{emb},\theta_{rec},\theta_{out}} \frac{1}{N}\sum_{i=1}^{N} -\log p(x_1^{(i)}, \ldots, x_{T^{(i)}}^{(i)}) \\ &= \min_{\theta_{emb},\theta_{rec},\theta_{out}} \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T^{(i)}} -\log p(x_t^{(i)}|x_{<t}^{(i)}).\end{aligned}$$

The entire network is fully differentiable and the weights can be learned with stochastic mini-batch gradient descent.

4.2 Neural Machine Translation

Given a set of parallel sentences in different languages $x^{(1)}, \ldots, x^{(n)}$ and $y_{(1)}, \ldots, y^{(n)}$ where $x_{(i)}$s is in the source language and $y_{(i)}$ is in the target language, Machine Translation can be seen as a language model that is conditioned on an additional sentence in other language.

$$p(y_1^{(i)}, \ldots, y_t^{(i)}, x_1^{(i)}, \ldots, x_T^{(i)}) = p(y_t^{(i)}|y_{<t}^{(i)}, x_1^{(i)}, \ldots, x_T^{(i)}).$$

Earlier statistical MT models were trained in a similar way to the statistical language models and suffered from similar issues: translation model was limited to markovian assumption and was often not generalizable.

Neural Machine Translation (NMT) takes the motivation from neural language models to create an end-to-end translation system which can model the entire MT process with just the parallel corpora with minimal domain knowledge (Kalchbrenner and Blunsom, 2013, Sutskever, Vinyals, and Le, 2014, Cho, Van Merriënboer, Gulcehre, et al., 2014). The core idea of the NMT system is to have a encoder which learns the representation for the source sentence h, based on which the decoder generates the translation, one target word at a time.

$$\begin{aligned}p(y^{(i)}, x^{(i)}) &= p(y_t^{(i)}|y_{<t}^{(i)}, h^{(i)}) \\ &= f(y_{<t}^{(i)}, h^{(i)}; \theta).\end{aligned}$$

Borrowing the notation from Cho, 2015, the entire neural network can be described into two major components: Encoder and Decoder.

4.2.1 Encoder

The Encoder is a parametric non-linear recurrent function for learning the representation of sentences in the source language in a small continuous space. The input to the encoder is a one-hot encoding of the words, which are passed to the embedding layer (similar to NLG models, section 4.1.2) to produce a continuous state representation of words:

$$\begin{aligned} s_t &= W_{emb}^T x_t && (W_{emb} \in \mathbb{R}^{|V| \times d}) \\ &= f_{emb}(x_t; W_{emb}). \end{aligned}$$

The word embeddings are then passed as an input to the RNN, which updates its hidden state recursively as:

$$h_t = f_{enc}(h_{t-1}, s_t; W_{enc}).$$

The hidden state the RNN can be considered as a representation for the sentence and as such is also called *summary vector*. Sutskever, Vinyals, and Le, 2014 showed that the summary vectors preserve the underlying semantics and syntax structure by projecting the state of the RNNs for different sentences in 2D space and showing similar sentences are grouped together in the summary space (Figure 4.1).

4.2.2 Decoder

The decoder is also a parametric non-linear recurrent function, which is used to compute the next word probability distribution given the summary vector and the target words generated so far. The hidden state of the decoder is updated as:

$$z_{t'} = f_{dec}(z_{t'-1}, u_{t'-1}, h_T; W_{dec}).$$

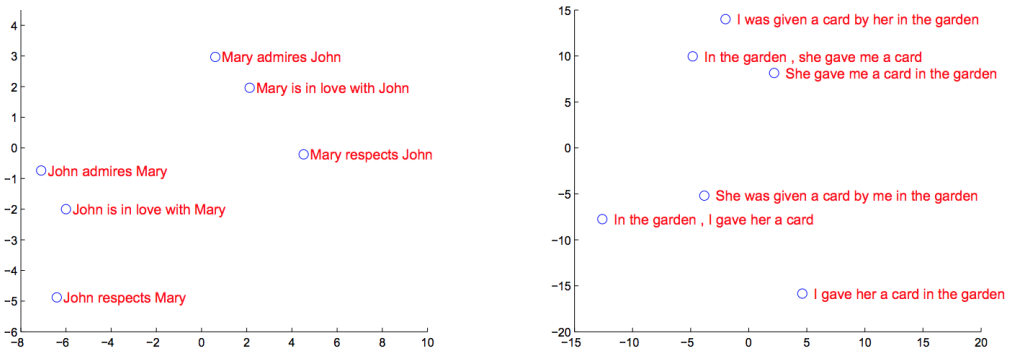


Figure 4.1: 2D PCA projection of the hidden state of the encoder after processing the phrases as shown in Sutskever, Vinyals, and Le, 2014. The phrases are clustered according to similar semantic and syntactic structure.

Here, $u_{t'}$ represents the target word generated at time t'.

$$p(u_{t'}|u_{<t'}) = f_{gen}(z_{t'}; W_{gen})$$

$$u_{t'} \sim p(u_{t'}|u_{<t'}).$$

The probabilities are calculated using the softmax activation. One of the ways to generate the target sentence is to directly sample words from the resulting probability distribution and consider this sampled target word as input to the decoder for the next time step. This approach is also known as **Sampling** based generation (Cho, Van Merriënboer, Gulcehre, et al., 2014). In order to get good translations, we need to sample repeatedly and choose the target sequence with the highest probability. Estimating the probability of each generated sentence can be computationally exhaustive and at the same time suffer from high variance.

An alternate technique is to consider all the possible target sentences which can be generated and computing the log-probability of each the sentence starting from the first word probability distribution. This approach is computationally intractable as it grows exponentially with the size of vocabulary along the length of the sentences ($|V|^T$). Therefore, an approximate search over the space of candidates is done based on the log-probability. One of the ways is to use a **Greedy Decoding** (Cho, 2015),

where at each time step the word with highest log-probability is selected, conditioned on previous generated words:

$$u_{t'} = \arg\max_{w' \in V} \log p(u_{t'} = w' | u_{<t'}).$$

Instead of doing greedy decoding for the most probable word at every time step, **Beam Search** keeps several hypothesis candidates ("beams") in the memory and chooses the best hypothesis based on highest log-probability. If there are K beams (or candidate sentences), then for each candidate, the next target is generated conditioned on the beam which results in $K \times |V|$ new possible candidates. Among these new candidates only the top K candidates are retained based on log-probabilities (Cho, 2015). Beam search typically gives better translations compared to greedy search as it performs a broader search over the space of possible candidates at the cost of higher computation time.

4.2.3 Automatic Evaluation Metric

Though Maximum Likelihood Estimation is used to train the MT system, the evaluation of such system on unseen sentence pairs is often done with approximate translation quality metrics, most notably **BLEU** (Papineni et al., 2002). This is because there can be many valid translations to a given sentence. Unlike classification, there is no absolute measure for the quality of the sentence generated. The main idea behind BLEU is to calculate the ratio between the n-grams in translation generated by the MT system and the n-grams in the actual human translations. If $\mathcal{D}$ represents the dataset, and S is the set of all unique n-grams in one sentence in $\mathcal{D}$, then n-gram precision of a translation, p_n, is :

$$p_n = \frac{\sum_{S \in \mathcal{D}} \sum_{\text{n-gram} \in S} \hat{c}(\text{n-gram})}{\sum_{S \in \mathcal{D}} \sum_{\text{n-gram} \in S} c(\text{n-gram})}$$

where,

$$c(\text{n-gram}) = \text{count of n-gram}$$
$$c_{ref}(\text{n-gram}) = \text{count of the n-gram in reference training sentences}$$
$$\hat{c}(\text{n-gram}) = \min(c(\text{n-gram}), c_{ref}(\text{n-gram})).$$

There is an additional penalty to account for the length of the translation, called Brevity Penalty (BP). If c is the length of the candidate translation and r is the length of the corresponding translation in the training set:

$$BP = \begin{cases} 1, & \text{if } c \geq r \\ \exp^{(1-r)/c}, & \text{otherwise.} \end{cases}$$

The final BLEU score is the product of the Brevity penalty multiplied with the geometric mean of N-grams[1]:

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n).$$

The core appeal of the BLEU score lies with its simplicity and correlation to the human scores. One of the caveats for using BLEU is that increase in BLEU score does not necessarily indicates improvement in translation quality (C.-Y. Lin and Och, 2004). Though there exist alternative metrics to BLEU such as METEOR (Denkowski and Lavie, 2014), BLEU is still used widely in the MT community.

4.3 NMT with Attention

The major limitation of the architecture described in the previous section 4.2, is that the model aims to encode the meaning of the entire sentence into a single vector, the state of the encoder. This problem becomes more prominent as the length of sentences increases (Cho, Van Merriënboer, Bahdanau, et al., 2014, Sutskever, Vinyals, and Le,

[1] (N is usually 4)

2014). Bahdanau, Cho, and Y. Bengio (2014) proposed a new architecture with the following two major techniques to exploit a variable-length context representation that learns better translations over longer sequences.

4.3.1 Bi-directional Encoder

Instead of encoding the sentence to a single vector as in the previous section 4.2, the source sentence is encoded as a set of state vectors for each time-step from two recurrent networks. If the length of a sequence is T, the first recurrent network reads the sentence in left to right manner, from $t = 0 \rightarrow T$, and generates the state for each time-step $\overrightarrow{h_t}$. This is the same as the NMT encoder described in the above section 4.2. The second recurrent network reads the sentence backwards, i.e., from $t = T \rightarrow 0$, and generates the state for each step, $\overleftarrow{h_t}$. For each time step, the final state is the concatenation from the forward and backward RNNs, h_t:

$$h_t = \begin{bmatrix} \overrightarrow{h_t} \\ \overleftarrow{h_t} \end{bmatrix}$$

The state at every time-step now has the context about the entire sentence with emphasis around the current word. This is because the forward RNN's hidden state, $\overrightarrow{h_t}$, contains information from beginning of the sentence to the current time-step and the backward RNN's state, $\overleftarrow{h_t}$, contains information from end of sentence to the current time-step. The concatenated state, h_t, is called the *context-dependent word representation* as it is encoding both the representation of the entire sentence and the current word. Such an architecture, where there is a forward and backward RNN, is also called a bi-directional RNN.

4.3.2 Attention based Decoding

The state of the decoder ($z_{t'}$) contains information about the target words generated so far. As all the encoder's context vectors cannot be given as input to the decoder for

each time-step, a score to decide how relevant each context vector is for translating the next word is calculated. The score for context vector h_j is based on the previous state of the decoder $z_{t'-1}$ (which contains information about the target words generated up to time-step $t'-2$), the previously generated target word $u_{t'-1}$, and the context vector itself.

$$e_j = f_{score}(z_{t'-1}, u_{t'-1}, h_j) \qquad (j = 0, 1, \ldots, T)$$

After calculating the scores for each h_j, the score is then normalized with a softmax function:

$$\alpha_j = \frac{\exp(e_j)}{\sum_{i=1}^{T} \exp(e_i)}.$$

The normalized score is called **attention score**, as it measures which context vectors the decoder is paying attention to. The final context vector is the weighted sum of the encoder's context vectors:

$$c_t = \sum_{j=1}^{T} \alpha_j h_j.$$

This final context vector is used to update the hidden state of the decoder and generate the next word:

$$z_{t'} = f_{dec}(z_{t'-1}, u_{t'-1}, c_t; W_{dec}).$$

The final architecture with both the above techniques is able to discover the word and phrase mappings between two languages. Using a NMT with attention mechanism is able to achieve an improvement of up to 60% BLEU score compared to a regular NMT without attention (Bahdanau, Cho, and Y. Bengio, 2014).

5

Real time machine translation

Online Machine Translation (or real-time translation) is defined as producing a partial translation of a sentence before the entire input sentence is completed. As mentioned briefly in the Introduction, the aim is to build a translation agent that can reduce the delay in the translation process by making decisions as to when to wait and gather more information about the input sequence or to start translating the current given sequence. Instead of using heuristics such as waiting till the end of the input sentence before generating the translation or translating each word as it is observed, the goal is to take into account the temporal nature of the problem and learn behaviors that can trade-off between translation quality and delay. Prior work in simultaneous machine translation is dominated by rule and parse-based approaches (Ryu, Matsubara, and Inagaki, 2006) or word segmentation based approaches (Oda et al., 2014). We extend the work by Grissom II et al. (2014) and convert the task into a Markov Decision Process so that the methods from the RL literature can be applied directly to learn optimal policies.

5.1 MDP formulation

In this section the real-time machine translation task is formulated in a Reinforcement Learning setting, by defining an environment, states, actions and a reinforce reward

or feedback from the environment.

5.1.1 Environment

In a traditional machine translation setting, the entire sentence is available to the system for translation but in a real-time setting the source emits only one word at a time. The environment acts as the source and provides a word in the source language at each time-step, denoted by, $x_1, \ldots, x_t$. The agent has to make a decision regarding which action to take and gets a feedback accordingly in the form of a reward (described in the next sections: 5.1.4, 5.1.3). At next the time-step, the environment emits another word from the source sentence sequence, x_{t+1}. Once the environment emits the last word from the original sentence sequence x_T it reaches a terminal state and episode ends. The environment then starts another episode and begins emitting words from a new sentence.

5.1.2 States

The state of the agent represents its current view of the environment. It is a function of the observed word sequence $x_1, \ldots, x_t$ and the translation emitted so far $y_1, \ldots, y_{t'}$. Formally, current state of the agent can be represented as the concatenation of the following vectors:

- The current context of the encoder of NMT system. It is essentially a representation of the observed words $x_1, \ldots, x_t$ in the source language.
- The current decoder-context conditioned on the translation emitted so far. This is a representation of the current translation $y_1, \ldots, y_{t'-1}$ in the target language.
- The last word emitted by the agent $y_{t'}$ in the target language.

5.1.3 Actions

At any particular time-step, the agent must decide whether to wait for more information or to generate the translation. The following two actions are defined to capture such behaviour:

- **WAIT**

  This action represents that the agent is not ready to translate yet and hence must "wait". On executing this action, the agent does not produces any output. This allows the agent to receive more input so if it decides to translate later, the translation is based on more information.

- **COMMIT**

  This action defines that the agent will generate a word in the target language and update the translation. On executing this action, the agent produces the next translation word $y_{t'+1}$ which is generated from the decoder of the NMT system using the current state. This takes advantage of the NMT system and gives the agent the ability to both translate from the existing input sequence as well as predict the next word in the output sequence.

5.1.4 Reward Function

At the end of each episode, the agent gets a reward based on the BLEU (Papineni et al., 2002)[1] score of the translation it produced with respect to the reference translation. Two additional modifications are made to the above reward function formulation :

- First, in the above formulation the agent does not have any penalty for taking WAIT actions. This can lead to a pathological behaviour where the agent waits till the very end of the source sentence and then start generating the translation.

[1]The implementation of BLEU used in this work is taken from the NLTK(Bird, Klein, and Loper, 2009) package: http://www.nltk.org/_modules/nltk/align/bleu.html

In order to prevent this behaviour, we give a small constant negative reward (penalty) to the agent, denoted by β, whenever the agent takes a WAIT action.

- Another issue with the current reward function is that even when the agent starts producing poorer translations, it will only receive the feedback for them at the end of the sequence. We use the partial BLEU score to address this issue. Let $\mathcal{S}$ be the score function for a partial translation, X_t the sequentially revealed source words $x_1, x_2, ..., x_t$ from time step 1 to t , and $\hat{Y}_t$ the partial translations $\hat{y}_1, \hat{y}_2, ..., \hat{y}_{t'}$ till time-step t. Each incremental translation $\hat{Y}_t$ has a BLEU score with respect to a reference Y. The partial BLEU score for time-step t is defined as,

$$\mathcal{S}(X_t, \hat{Y}_t) = \text{BLEU}(\hat{Y}_t, Y) - \text{BLEU}(\hat{Y}_{t-1}, Y)$$

  Using the partial reward at each time-step instead of awarding the whole score at the last step does not change the optimal policy (Ng, Harada, and S. Russell, 1999). This formulation provides training signal for the agent at each time-step and ensures that the agent knows it is drifting towards producing poorer translations.

Therefore, if the agent takes the WAIT action, it gets a penalty denoted by β. When the agent chooses the COMMIT action, it gets the partial BLEU score depending on the difference of current translation and previous time-step's translation.

5.2 Learning

The problem is now transformed into a classical RL setting where the agent can interact with the environment at every time-step, receive a feedback reward, and the goal is to maximize the expected return. In the new formulation, the state space is very large (the hidden states of the encoder and decoder concatenated with the

word embedding of the last generated word) but the action space is extremely small (two actions: WAIT and COMMIT). As a result, Deep Reinforcement Learning based algorithms such as DQNs (V. Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015) and Deep Policy Gradients (Lillicrap et al., 2015) can be used to learn policies that maximize the expected return.

5.2.1 Learning a Q Network

Policies are learned using Q-learning with function approximation in the same manner as described in Section DQN (3.4.5). The NMT provides the state representation which is fed into the DQN. A high-level architecture of the system is represented in Figure 5.1

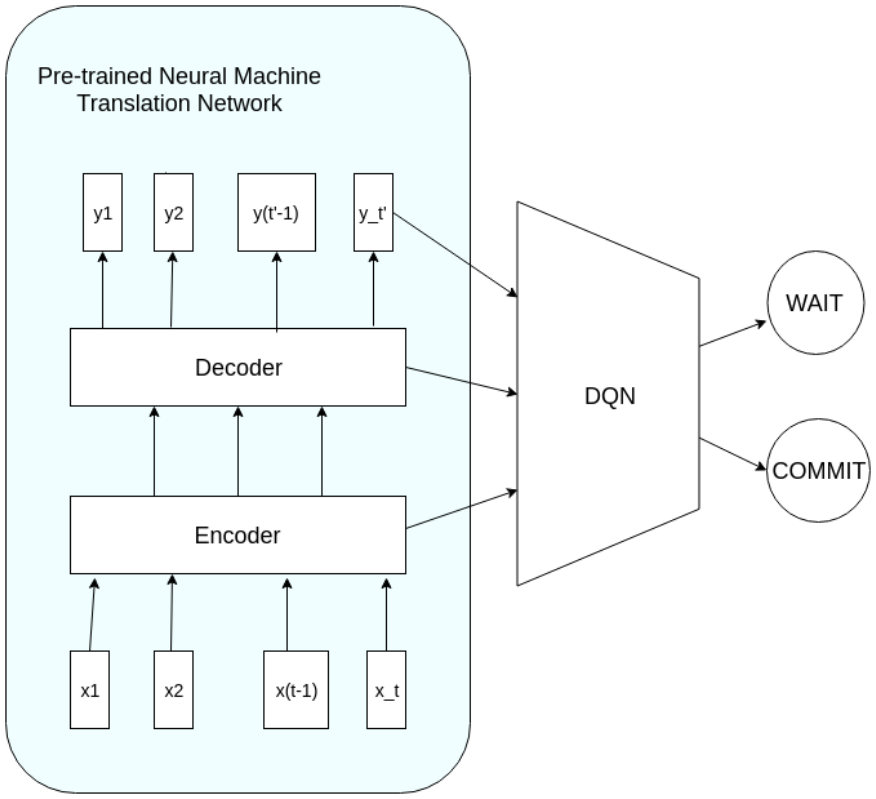


Figure 5.1: Architecture of the SMT system

The training is done in two phases. In the first phase, the aim is to learn a good batch translation system (a regular NMT system) using a bilingual corpus. This pre-

trained NMT system can now be used to get feature representation for the states. As described in section 5.1.2 the state of an agent includes context from encoder, context from decoder and the last emitted word. These are concatenated together to form a feature vector represented as s_t.

In the second phase the parameters of the DQN are learned to maximize the expected return using Q-Learning. Given a source sentence, at time-step t the state representation from the NMT system s_t is fed as input to the DQN which produces the Q-values for the available actions for that state. The agent follows the greedy policy with respect to the Q-values and selects the action, a_t, with the largest return and executes that. It gets a corresponding reward, r_t from the environment and also the next word from the source sequence, which changes the state to s_{t+1}. This transition of $< s_t, a_t, r_t, s_{t+1} >$ is stored in an experience buffer. The agent keeps on executing in such manner until it reaches the terminal state (the end of sentence, represented by *<eos>* token). Once the agent reaches the terminal state, WAIT is no longer an available action and hence it performs a series of COMMIT actions until it produces the *<eos>* token for the target sentence, at which stage the episode ends.

6

Empirical Evaluation

6.1 Setup

To empirically study the proposed real-time translation model, we ran experiments on three different language translation tasks: translating sentences from English to French (En→Fr), translating sentences from English to German (En→De) and translating sentences from Japanese to English (Jp→En)[1].

For En→Fr and En→De translation tasks, the datasets consisting of bilingual parallel sentences are taken from the WMT 2014 translation task[2]. The Europarl parallel corpus (Koehn, 2005) is used as the training set for training the NMT system as well as the RL agent. The Europarl parallel corpus is extracted from the proceedings of the European Parliament. We use the news-test-2014 development dataset as the validation set and evaluate the models on the news-test-2011 dataset from WMT 2014 task. Both news-test-2014 and news-test-2011 consist of approximately 3000 sentence pairs which were not present in the training data.

The dataset, for Jp→En translation task, is taken from the Kyoto Free Translation Task (Neubig, 2011). The dataset consists of parallel sentences of Wikipedia articles related to Kyoto[3]. Details for each dataset can be found in Table 6.1.

[1]We chose the above language pairs to test our results to cover a variety of language pairs with different syntactic structure.

[2]http://www.statmt.org/wmt14/

[3]The dataset is already partitioned into train, validation and test sets by the Kyoto Free Trans-

Table 6.1: Dataset details

Language pair	Sentences	Source language words	Target language words
English→French	2M	50M	51M
English→German	1.9M	47M	44.5M
Japanese→English	440k	12M	11.5M

Table 6.2: NMT network parameters

Parameter	Value
Source-vocabulary size	30,000
Target-vocabulary size	30,000
Word embedding dimension	1024
Optimizer	Adam
Learning rate	0.0001
GRU state dimension	500
Training Epochs	5000

Each word in the dataset is given a unique identifier, also called the *token*. Words are converted to tokens using the tokenizing script provided by the Moses Statistical MT system (Koehn, Hoang, et al., 2007). After the tokenization the size of the vocabulary is fixed due to the computationally intensive nature of the softmax over vocabulary operation in the NMT system. If $|V|$ denotes the size of vocabulary, then only $|V|$ most frequent words in the language are selected[4]. Any word not in the vocabulary is replaced by a special token, $<UNK>$, denoting an unknown word. To denote the end of sentence, an end of sequence token ($<eos>$) is also added to all sequences as another pre-processing step. We do not apply any other special pre-processing to the data.

The network architecture and other major parameters for training our NMT system and DQN are described in Table 6.2 and Table 6.3 respectively.

lation Task organizers.

[4]Most NMT system limit their vocabularies to be the top 30K-80K most frequent words in each language.

Table 6.3: DQN network parameters

Parameter	Value
Discount factor (γ)	0.99
Hidden units (Layer 1 - Layer 2- Layer 3)	512 - 64 - 2
Replay memory size	1000000
Optimizer	RMSProp
Learning rate	0.00025
Training Epochs	300
Training steps per epoch	25000
Target network update frequency	1000
Exploration end rate	0.1
Exploration decay steps	200000
Wait Penalty (β)	-0.005

6.2 Results

We now present the results of the proposed real-time translation system. We compare the policies learnt by our system against monotone and batch translation policies. **Batch** translation is defined as when the agent waits till the end of sentence before starting to translate (traditional machine translation). This can be reproduced by the agent via taking a sequence of WAIT actions until all input is observed and then taking a sequence of COMMIT actions. A **monotone** translation is one where the agent translates each word as soon as it is observed. The agent can reproduce this by taking a COMMIT action at every time-step. Note that for language pairs which have different word orders, the monotone translation will potentially provide a poorer translation. The policy learnt by the agent in the RL framework will be referred to as **Adaptive** policy.

Quantitative results comparing the expected return by different policies can be found in Table 6.7. However, the notion of comparing policies based on the expected return is not fair as the reward function is biased against batch and monotone policies. An alternate way to interpret the results can be to compare the trade-off between delay and translation quality for different policies. We define the following metrics to capture the quality and delay:

- **Quality**

  The translation quality of the generated sentences is denoted by calculating the BLEU score on the entire evaluation dataset.

- **Delay**

  Delay can be represented by the number of time-steps when agent is laying idle and is not producing any translation. Thus, we can define delay as average number of WAIT actions the agent takes.

The results based on the new metrics for two different values of WAIT action penalty (β) are shown in the Figure 6.1. The learning curve during the training for a sample run is shown in Figure 6.2.

Some examples of the translation generated by the agent are demonstrated in the tables: Table 6.4, Table 6.5 and Table 6.6. In these tables, rows represent the progression in time, with the top row representing $t = 0$, the second row denoting time-step $t = 1$ and so on. The first column shows the source sentence observed by the agent (Input) and the translation produced by the agent (Translation). The second column shows the action taken by the agent at that corresponding time-step (Action). The reference translation used for calculating the BLEU score is shown in the last row as Original translation.

6.3 Analysis

In this section, we examine some examples to gather insights into the behavior of the system. We can make a few observations from the plot of delay vs translation quality (Figure 6.1) :

- The batch policy (which waits till the end of entire input sequence) has much higher delay but also has the best translation score. This is expected behaviour

Table 6.4: Example of a translation generated by the trained agent for English→French translation task.

Observed sentence and translation	Action
Input: The Translation: La	COMMIT
Input: The European Translation: La Commission	COMMIT
Input: The European Union Translation: La Commission	WAIT
Input: The European Union must Translation: La Commission	WAIT
Input: The European Union must be Translation: La Commission	WAIT
Input: The European Union must be able Translation: La Commission	WAIT
Input: The European Union must be able to Translation: La Commission européenne	COMMIT
Input: The European Union must be able to act Translation: La Commission européenne	WAIT
Input: The European Union must be able to act UNK Translation: La Commission européenne doit	COMMIT
Input: The European Union must be able to act UNK . Translation: La Commission européenne doit	WAIT
Input: The European Union must be able to act UNK . <eos> Translation: La Commission européenne doit pouvoir agir de façon UNK .	Sequence of COMMITs
Original translation: L' Union européenne doit pouvoir UNK de façon UNK .	

as the batch policy has access to the entire input sequence and thus has the highest potential to produce the best translation.

- The monotone policy (which translates at each time-step) has no delay but has the worst translation quality when compared to other policies. Again this is perhaps not surprising as the agent is forced to produce a translated word at each time-step and thus has to make translations without entire knowledge of the sequence which results in poorer quality.

- The policies learnt by the agent have less delay than the batch but better translation quality than the monotone policy and by setting the value for the WAIT penalty we can control the trade-off between delay and accuracy.

- The gap between monotone and batch translation scores is dependent on word-order for language pairs. The gap is largest for Japanese → English task and

Table 6.5: Example of a translation generated by the trained agent for Japanese→English translation task.

Observed sentence and translation	Action
Input: 真宗 Translation:	WAIT
Input: 真宗 佛光 Translation:	WAIT
Input: 真宗 佛光 寺 Translation: Shinshu	COMMIT
Input: 真宗 佛光 寺 派 Translation: Shinshu sect	COMMIT
Input: 真宗 佛光 寺 派 （ Translation: Shinshu sect	WAIT
Input: 真宗 佛光 寺 派 （ しんしゅう Translation: Shinshu sect	WAIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶ Translation: Shinshu sect	WAIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ Translation: Shinshu sect	WAIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ Translation: Shinshu sect	WAIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は Translation: Shinshu sect	WAIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） Translation: Shinshu sect Bukko-ji	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と Translation: Shinshu sect Bukko-ji school	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は Translation: Shinshu sect Bukko-ji school is	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 Translation: Shinshu sect Bukko-ji school is a	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 浄土 Translation: Shinshu sect Bukko-ji school is a Pure	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 浄土 真宗 Translation: Shinshu sect Bukko-ji school is a Pure Land	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 浄土 真宗 の Translation: Shinshu sect Bukko-ji school is a Pure Land sect	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 浄土 真宗 の 一派 Translation: Shinshu sect Bukko-ji school is a Pure Land sect school	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 浄土 真宗 の 一派 。 Translation: Shinshu sect Bukko-ji school is a Pure Land sect school ,	COMMIT
Input: 真宗 佛光 寺 派 （ しんしゅう ぶっ こうじ は ） と は 、 浄土 真宗 の 一派 。 <eos> Translation: Shinshu sect Bukko-ji school is a Pure Land sect school , one of the Jodo Shinshu (the True Pure Land Sect of Buddhism) .	Sequence of COMMITs
Original translation: The Bukko-ji (UNK) School of Shin Sect is a school of Jodo Shinshu (the True Pure Land Sect of Buddhism) .	

smallest for English → French translation task.

6.3.1 Comparison with random policies

We compare the policies learnt by the agent in the previous section with the random policies. In Figure 6.3, random policies are represented as solid lines and the probability for a WAIT action increase from left to right on the x-axis. Since, there are two actions, this corresponds to probability of COMMIT action decreasing from left to right. At the far left we have a policy with <probability(WAIT)=0, probability(COMMIT)=1>, that is equivalent to the monotone policy. On the extreme right, we have policy with <probability(COMMIT)=0, probability(WAIT)=1> which is the same as the batch policy. In the middle of the x-axis, we have a policy with <probability(COMMIT)=probability(WAIT)=0.5>.

Table 6.6: Example of a translation generated by the trained agent for English→German translation task.

Observed sentence and translation	Action
Input: It Translation: Er	COMMIT
Input: It is Translation: Er ist	COMMIT
Input: It is precisely Translation: Er ist eben	COMMIT
Input: It is precisely for Translation: Er ist eben f‘ijr	COMMIT
Input: It is precisely for this Translation: Er ist eben f‘ijr	WAIT
Input: It is precisely for this reason Translation: Er ist eben f‘ijr	WAIT
Input: It is precisely for this reason that Translation: Er ist eben f‘ijr	WAIT
Input: It is precisely for this reason that the Translation: Er ist eben f‘ijr	WAIT
Input: It is precisely for this reason that the idea Translation: Er ist eben f‘ijr	WAIT
Input: It is precisely for this reason that the idea of Translation: Er ist eben f‘ijr diesen	COMMIT
Input: It is precisely for this reason that the idea of a Translation: Er ist eben f‘ijr diesen	WAIT
Input: It is precisely for this reason that the idea of a review Translation: Er ist eben f‘ijr diesen	WAIT
Input: It is precisely for this reason that the idea of a review of Translation: Er ist eben f‘ijr diesen	WAIT
Input: It is precisely for this reason that the idea of a review of the Translation: Er ist eben f‘ijr diesen Grund	COMMIT
Input: It is precisely for this reason that the idea of a review of the financial Translation: Er ist eben f‘ijr diesen Grund der	COMMIT
Input: It is precisely for this reason that the idea of a review of the financial UNK Translation: Er ist eben f‘ijr diesen Grund der Idee	COMMIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is Translation: Er ist eben f‘ijr diesen Grund der Idee	WAIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not Translation: Er ist eben f‘ijr diesen Grund der Idee	WAIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not being Translation: Er ist eben f‘ijr diesen Grund der Idee einer ‘IJberpr‘ijfung der UNK nicht im Moment .	WAIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not being considered Translation: Er ist eben f‘ijr diesen Grund der Idee einer	COMMIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not being considered at Translation: Er ist eben f‘ijr diesen Grund der Idee einer ‘IJberpr‘ijfung	COMMIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not being considered at present Translation: Er ist eben f‘ijr diesen Grund der Idee einer ‘IJberpr‘ijfung	WAIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not being considered at present . Translation: Er ist eben f‘ijr diesen Grund der Idee einer ‘IJberpr‘ijfung	WAIT
Input: It is precisely for this reason that the idea of a review of the financial UNK is not being considered at present . <eos> Translation: Er ist eben f‘ijr diesen Grund der Idee einer ‘IJberpr‘ijfung der UNK nicht im Moment .	sequence of COMMITs
Original translation: Aus diesem Grund wird UNK eine UNK der UNK UNK noch nicht UNK .	

Table 6.7: Quantitative results on English→German

Policy	Average Reward
Monotone	0.5498
Batch	0.5847
Learnt	0.6123

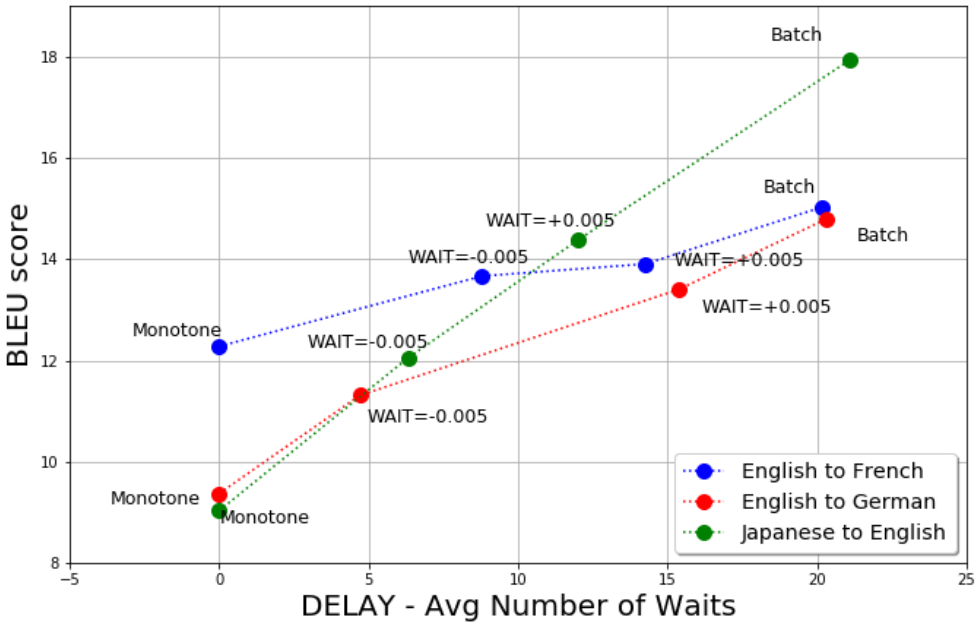


Figure 6.1: Delay vs Translation Quality for En→Fr, En→De and Jp→En translation tasks for two different Wait penalty (β). WAIT=-0.005 represents the RL agent trained on $\beta = -0.005$ and vice versa.

From the above plot (Figure 6.3), we can observe that the policy learnt by the agent is as good as a random policy. Moreover, random policies are also able to achieve delay-accuracy trade-off. A possible hypothesis is that taking a few WAIT actions allows the agent to have a preview of next words in the sequence for the remaining episode. For example, after taking 5 WAIT actions, the agent now always observes at least 5 more words in the source sequence. This can also be seen as the inability of the agent to catch up to the source sentence as the agent can only generate one word at every time-step.

We expected the random policies to not perform well on Japanese to English translation task as this pair of languages has a very different syntax order. However, we found the results to be similar in the sense that random policies work as well as the learnt policies. An interesting observation worth mentioning is that the learnt policy is not purely random in nature, for example, the policy learns the set of Japanese

characters that are supposed to appear together (such as kanjis for dates and numbers). This results in a learnt policy that waits for the phrase before generating the translation. A random policy cannot capture this behavior but still empirically shows the same performance as the learnt policy.

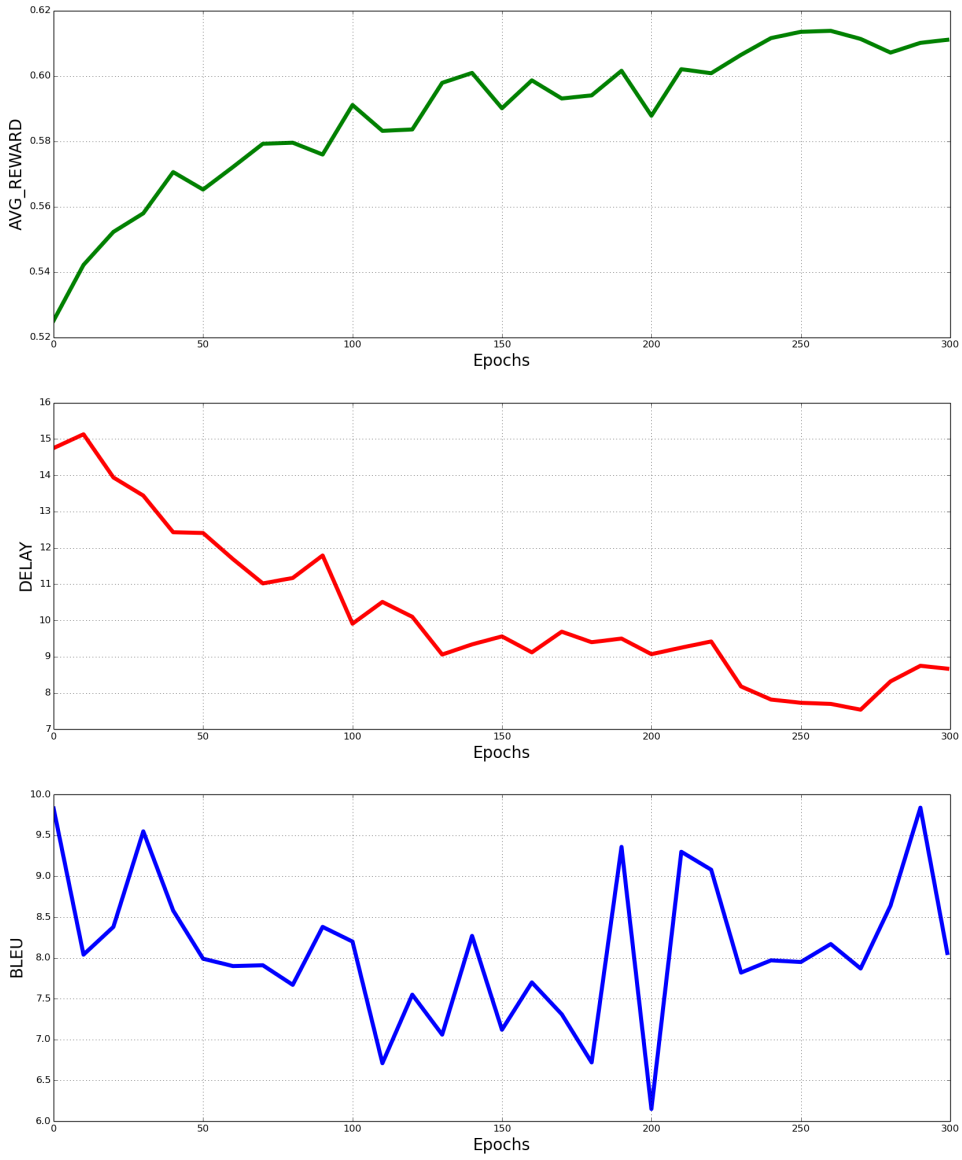


Figure 6.2: Learning Curves during training for English→German translation task for the validation set. The x-axis represents the training epochs. The top plot denotes the expected reward which the agent is trying to maximize using RL. The middle plot denotes delay (average number of WAIT steps). The bottom plot denotes the translation quality (BLEU score).

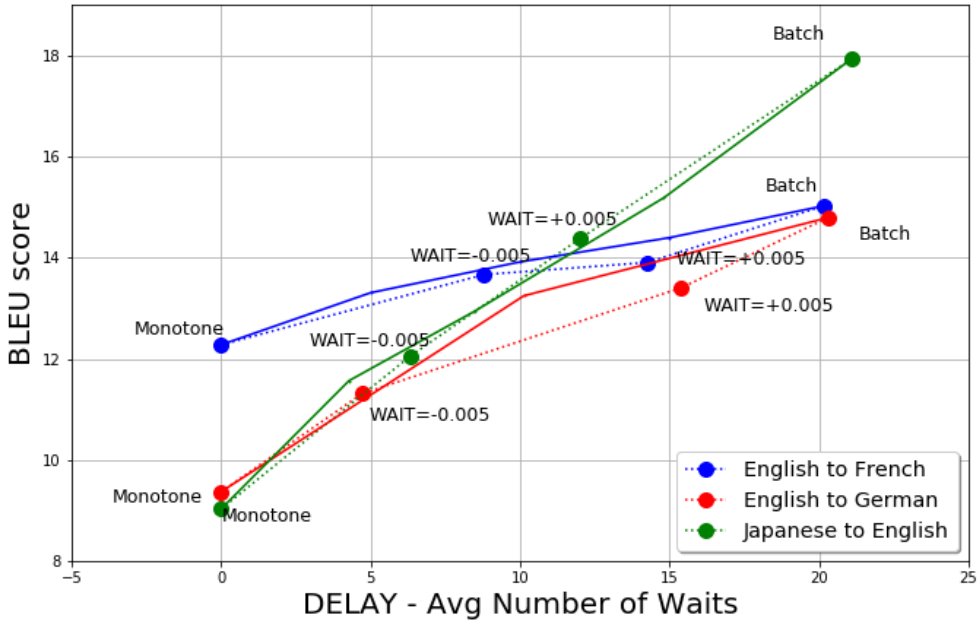


Figure 6.3: Policy comparison with random policies. The solid lines represent random policies.

7

Conclusion and Future work

In this work, we present for the first time, a unified framework for real-time machine translation task that combines translation and decision-making into a single architecture. We have empirically demonstrated that our agent is able to learn better policies than the standard batch and monotone policies on two language pairs. We also demonstrated that the policies learnt in this framework exhibit a delay-accuracy trade-off. The framework is generalizable as it can be extended to any language pair without any prior knowledge. Any advances to the existing NMT systems in the future can be easily integrated in the system.

The presented framework has many challenges and possible avenues of improvements. The foremost task that needs to be understood is the behavior of random policies as mentioned in Section 6.3.1. Since our work, there has already been a rise in work on real-time machine translation (Gu, Neubig, et al., 2017, Cho and Esipova, 2016), that proposes alternate metrics for evaluating the performance on real-time machine translation task. However, none of them compare the performance to random policies baselines. The relation between random policy and reward penalty needs to be studied in more detail. Also, we have not completely explored other Reinforcement Learning methods for this task, particularly the family of on-policy methods.

In the present framework, the agent can only perform primitive actions. One of the limitations of the current actions is that the agent needs to make decisions at the word

level. The agent should ideally also have actions like COMMIT-PHRASE that can be used to generate entire phrases. This can be done in either a neural language model based decoding style (Gu, Cho, and Li, 2017) or using hierarchical reinforcement learning algorithms such as the options framework (Sutton, Precup, and Singh, 1999). Recent work by Gu, Neubig, et al. (2017) builds upon the approach mentioned in this work and introduces a greedy decoder based COMMIT action mechanism that can consecutively decode multiple steps and generate an entire phrase. This allows the agent to generate variable length translations on taking COMMIT action and not lag behind the source sentence. However, they do not give any comparisons with the random policy based baselines.

One of weaknesses of the current approach is that the NMT system is trained on sentence level but then it is employed for decision making for incomplete sentences. One way to address this is to fine-tune the NMT system on either phrases or incomplete sentences. Another way to address this can be to construct task specific datasets for real-time machine translation. The data from human translators (Shimizu et al., 2014) can be helpful to fine tune the translation system and learn further strategies for real-time translation. However, much less data of this type is available.

An important aspect that needs to be studied is the impact of the reward formulation on the task. The current system heavily depends on the reward formulation and penalty for WAIT action. Instead of manually specifying the reward and penalty, another area of interest will be to learn better reward function directly from human translators' data. This should be possible using the techniques from the Inverse Reinforcement Learning literature (Ng, S. J. Russell, et al., 2000). The translation data can also be seen as demonstrations by the humans, and as such it should also be possible to use Imitation Learning (Argall et al., 2009, Billard et al., 2008) in this domain.

This work is a first small step towards building a fully generalizable and usable real-time machine translation system. There are shortcomings in the present framework

that need to be addressed before it can be used in any real setting. In spite of that, we demonstrate that it is possible to build end-to-end online translation system using reinforcement learning. As the research in NMT systems, online translation evaluation metrics and datasets, and reinforcement learning advances, our framework can also improve.

Bibliography

Abadi, Martin, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. (2016). "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467*.

Abbeel, Pieter, Adam Coates, Morgan Quigley, and Andrew Y Ng (2007). "An application of reinforcement learning to aerobatic helicopter flight". In: *Advances in neural information processing systems*, pp. 1–8.

Anderson, Stephen R (2004). "How many languages are there in the world". In: *Linguistic Society of America*.

Arfken, George (1985). "The method of steepest descents". In: *Mathematical methods for physicists* 3, pp. 428–436.

Argall, Brenna D, Sonia Chernova, Manuela Veloso, and Brett Browning (2009). "A survey of robot learning from demonstration". In: *Robotics and autonomous systems* 57.5, pp. 469–483.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473*.

Bastien, Fr‘Ĭd‘Ĭric, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Ben-

gio (2012). “Theano: new features and speed improvements”. In: *arXiv preprint arXiv:1211.5590*.

Bellemare, Marc G, Yavar Naddaf, Joel Veness, and Michael Bowling (2013). “The Arcade Learning Environment: An evaluation platform for general agents.” In: *J. Artif. Intell. Res.(JAIR)* 47, pp. 253–279.

Bellman, Richard (1957). “A Markovian decision process”. In: *Journal of Mathematics and Mechanics*, pp. 679–684.

Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer (2015). “Scheduled sampling for sequence prediction with recurrent neural networks”. In: *Advances in Neural Information Processing Systems*, pp. 1171–1179.

Bengio, Yoshua et al. (2009). “Learning deep architectures for AI”. In: *Foundations and trends‘ő in Machine Learning* 2.1, pp. 1–127.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013). “Representation learning: A review and new perspectives”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1798–1828.

Bengio, Yoshua, R‘ľjean Ducharme, Pascal Vincent, and Christian Jauvin (2003). “A neural probabilistic language model”. In: *Journal of machine learning research* 3.Feb, pp. 1137–1155.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi (1994). “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5.2, pp. 157–166.

Bergstra, James, Olivier Breuleux, Fr‘ľd‘ľric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio (2010). “Theano: A CPU and GPU math compiler in Python”. In: *Proc. 9th Python in Science Conf*, pp. 1–7.

Billard, Aude, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal (2008). “Robot programming by demonstration”. In: *Springer handbook of robotics*. Springer, pp. 1371–1394.

Bird, Steven, Ewan Klein, and Edward Loper (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc."

Bishop, Christopher M (2006). *Pattern recognition and machine learning*. springer.

Boyan, Justin A and Michael L Littman (1994). "Packet routing in dynamically changing networks: A reinforcement learning approach". In: *Advances in neural information processing systems*, pp. 671–678.

Brown, Peter F, Vincent J Della Pietra, Stephen A Della Pietra, and Robert L Mercer (1993). "The mathematics of statistical machine translation: Parameter estimation". In: *Computational linguistics* 19.2, pp. 263–311.

Chetlur, Sharan, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer (2014). "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759*.

Cho, Kyunghyun (2015). "Natural Language Understanding with Distributed Representation". In: *CoRR* abs/1511.07916.

Cho, Kyunghyun and Masha Esipova (2016). "Can neural machine translation do simultaneous translation?" In: *arXiv preprint arXiv:1606.02012*.

Cho, Kyunghyun, Bart Van Merri‘ńnboer, Dzmitry Bahdanau, and Yoshua Bengio (2014). "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259*.

Cho, Kyunghyun, Bart Van Merri‘ńnboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078*.

Chung, Junyoung, ‘Ğaglar G‘ijl‘ğehre, KyungHyun Cho, and Yoshua Bengio (2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555.

Collobert, Ronan (2004). "Large Scale Machine Learning". In:

Collobert, Ronan, Koray Kavukcuoglu, and Cl‘ľment Farabet (2011). “Torch7: A matlab-like environment for machine learning”. In: *BigLearn, NIPS Workshop*. EPFL-CONF-192376.

Crites, Robert H and Andrew G Barto (1996). “Improving elevator performance using reinforcement learning”. In: *Advances in neural information processing systems*, pp. 1017–1023.

Cruz, Joseph A and David S Wishart (2006). “Applications of machine learning in cancer prediction and prognosis”. In: *Cancer informatics* 2, p. 59.

Denkowski, Michael and Alon Lavie (2014). “Meteor universal: Language specific translation evaluation for any target language”. In: *Proceedings of the ninth workshop on statistical machine translation*, pp. 376–380.

Fisher, Ronald Aylmer (1925). “Theory of statistical estimation”. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 22. 5. Cambridge University Press, pp. 700–725.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). “Deep sparse rectifier neural networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323.

Greensmith, Evan, Peter L Bartlett, and Jonathan Baxter (2004). “Variance reduction techniques for gradient estimates in reinforcement learning”. In: *Journal of Machine Learning Research* 5.Nov, pp. 1471–1530.

Grissom II, Alvin, He He, Jordan Boyd-Graber, John Morgan, and Hal Daum‘ĺ III (2014). “Don‘Ă‘Źt until the final verb wait: Reinforcement learning for simultaneous machine translation”. In: *Proceedings of the 2014 Conference on empirical methods in natural language processing (EMNLP)*, pp. 1342–1352.

Gu, Jiatao, Kyunghyun Cho, and Victor OK Li (2017). “Trainable greedy decoding for neural machine translation”. In: *arXiv preprint arXiv:1702.02429*.

Gu, Jiatao, Graham Neubig, Kyunghyun Cho, and Victor OK Li (2017). "Learning to translate in real-time with neural machine translation". In: *Proc. EACL, Valencia, Spain, April 2017*.

He, He, Alvin Grissom II, John Morgan, Jordan Boyd-Graber, and Hal Daum'ł III (2015). "Syntax-based rewriting for simultaneous machine translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 55–64.

Hinton, Geoffrey E and Zoubin Ghahramani (1997). "Generative models for discovering sparse distributed representations". In: *Philosophical Transactions of the Royal Society of London B: Biological Sciences* 352.1358, pp. 1177–1190.

Hinton, Geoffrey, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. (2012). "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal Processing Magazine* 29.6, pp. 82–97.

Hinton, Geoffrey, Nitish Srivastava, and Kevin Swersky (2012). "Lecture 6a overview of mini–batch gradient descent". In: *Coursera Lecture slides https://class. coursera. org/neuralnets-2012-001/lecture,[Online*.

Hochreiter, Sepp and J'ijrgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2, pp. 251–257.

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016). "Deep Learning". Book in preparation for MIT Press.

Kalchbrenner, Nal and Phil Blunsom (2013). "Recurrent Continuous Translation Models." In: *EMNLP*. Vol. 3. 39, p. 413.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Koehn, Philipp (2005). "Europarl: A parallel corpus for statistical machine translation". In: *MT summit*. Vol. 5, pp. 79–86.

— (2009). *Statistical machine translation*. Cambridge University Press.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. (2007). "Moses: Open source toolkit for statistical machine translation". In: *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics, pp. 177–180.

Koehn, Philipp, Franz Josef Och, and Daniel Marcu (2003). "Statistical phrase-based translation". In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for Computational Linguistics, pp. 48–54.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444.

LeCun, Yann, L'ĺon Bottou, Yoshua Bengio, and Patrick Haffner (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Lee, Jae Won (2001). "Stock price prediction using reinforcement learning". In: *Industrial Electronics, 2001. Proceedings. ISIE 2001. IEEE International Symposium on*. Vol. 1. IEEE, pp. 690–695.

Lillicrap, Timothy P, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2015). "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*.

Lin, Chin-Yew and Franz Josef Och (2004). "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics". In:

Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, p. 605.

Lin, Long-Ji (1993). *Reinforcement learning for robots using neural networks*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Lin, Tsungnan, Bill G Horne, Peter Tino, and C Lee Giles (1996). "Learning long-term dependencies in NARX recurrent neural networks". In: *IEEE Transactions on Neural Networks* 7.6, pp. 1329–1338.

Luenberger, David G, Yinyu Ye, et al. (1984). *Linear and nonlinear programming*. Vol. 2. Springer.

Mann, William C and Sandra A Thompson (1988). "Rhetorical structure theory: Toward a functional theory of text organization". In: *Text-Interdisciplinary Journal for the Study of Discourse* 8.3, pp. 243–281.

Metropolis, Nicholas and Stanislaw Ulam (1949). "The monte carlo method". In: *Journal of the American statistical association* 44.247, pp. 335–341.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean (2013). "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*, pp. 3111–3119.

Mnih, Andriy and Geoffrey E Hinton (2009). "A scalable hierarchical distributed language model". In: *Advances in neural information processing systems*, pp. 1081–1088.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013). "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Morin, Frederic and Yoshua Bengio (2005). "Hierarchical Probabilistic Neural Network Language Model." In: *Aistats*. Vol. 5, pp. 246–252.

Murphy, Kevin P (2012). *Machine learning: a probabilistic perspective*. MIT press.

Neubig, Graham (2011). *The Kyoto Free Translation Task*. http://www.phontron.com/kftt.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping". In: *International Conference on Machine Learning*. Vol. 99, pp. 278–287.

Ng, Andrew Y, Stuart J Russell, et al. (2000). "Algorithms for inverse reinforcement learning." In: *International Conference on Machine Learning*, pp. 663–670.

Nvidia, CUDA (2010). *Programming guide*.

Och, Franz Josef and Hermann Ney (2003). "A systematic comparison of various statistical alignment models". In: *Computational linguistics* 29.1, pp. 19–51.

Oda, Yusuke, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura (2014). "Optimizing Segmentation Strategies for Simultaneous Speech Translation." In: *ACL (2)*, pp. 551–556.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu (2002). "BLEU: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, pp. 311–318.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning*, pp. 1310–1318.

Peters, Jan, Sethu Vijayakumar, and Stefan Schaal (2003). "Reinforcement learning for humanoid robotics". In: *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pp. 1–20.

Pollack, Jordan B (1990). "Recursive distributed representations". In: *Artificial Intelligence* 46.1, pp. 77–105.

Polyak, Boris T (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17.

Puterman, Martin L (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Rasmus, Antti, Mathias Berglund, Mikko Honkala, Harri Valpola, and Tapani Raiko (2015). "Semi-supervised learning with ladder networks". In: *Advances in Neural Information Processing Systems*, pp. 3546–3554.

Robbins, Herbert and Sutton Monro (1951). "A stochastic approximation method". In: *The annals of mathematical statistics*, pp. 400–407.

Rosenblatt, Frank (1958). "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Rosenfeld, Ronald (2000). "Two decades of statistical language modeling: Where do we go from here?" In: *Proceedings of the IEEE* 88.8, pp. 1270–1278.

Rumelhart, David E, Geoffrey E Hinton, Ronald J Williams, et al. (1988). "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3, p. 1.

Ryu, Koichiro, Shigeki Matsubara, and Yasuyoshi Inagaki (2006). "Simultaneous English-Japanese spoken language translation based on incremental dependency parsing and transfer". In: *Proceedings of the COLING/ACL on Main conference poster sessions*. Association for Computational Linguistics, pp. 683–690.

Satija, Harsh and Joelle Pineau (2016). "Simultaneous machine translation using deep reinforcement learning". In: *Abstraction in Reinforcement Learning Workshop, International Conference on Machine Learning 2016*.

Sheridan, Peter (1955). "Research in language translation on the IBM type 701". In: *IBM Technical Newsletter* 9, pp. 5–24.

Shimizu, Hiroaki, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura (2014). "Collection of a Simultaneous Translation Corpus for Comparative Analysis." In: *LREC*, pp. 670–673.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587, pp. 484–489.

Simao, Hugo P, Jeff Day, Abraham P George, Ted Gifford, John Nienow, and Warren B Powell (2009). "An approximate dynamic programming algorithm for large-scale fleet management: A case application". In: *Transportation Science* 43.2, pp. 178–197.

Srivastava, Nitish, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of machine learning research* 15.1, pp. 1929–1958.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*, pp. 3104–3112.

Sutton, Richard S (1988). "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1, pp. 9–44.

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge.

Sutton, Richard S, Doina Precup, and Satinder Singh (1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1-2, pp. 181–211.

Szepesv'ąri, Csaba (2010). "Algorithms for reinforcement learning". In: *Synthesis lectures on artificial intelligence and machine learning* 4.1, pp. 1–103.

Taigman, Yaniv, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf (2014). "Deepface: Closing the gap to human-level performance in face verification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1701–1708.

Tenenbaum, Joshua B, Vin De Silva, and John C Langford (2000). "A global geometric framework for nonlinear dimensionality reduction". In: *science* 290.5500, pp. 2319–2323.

Tesauro, Gerald (1995). "Td-gammon: A self-teaching backgammon program". In: *Applications of Neural Networks*. Springer, pp. 267–285.

Tieleman, Tijmen and Geoffrey Hinton (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2, pp. 26–31.

Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol (2008). "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 1096–1103.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8.3-4, pp. 279–292.

Williams, Ronald J (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4, pp. 229–256.

Williams, Ronald J and Jing Peng (1990). "An efficient gradient-based algorithm for on-line training of recurrent network trajectories". In: *Neural computation* 2.4, pp. 490–501.

Williams, Ronald J and David Zipser (1989a). "A learning algorithm for continually running fully recurrent neural networks". In: *Neural computation* 1.2, pp. 270–280.

— (1989b). "Experimental analysis of the real-time recurrent learning algorithm". In: *Connection Science* 1.1, pp. 87–111.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. (2016). "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144*.