On Reinforcement Learning for Deep Neural Architectures : Conditional computation with stochastic computation policies

Emmanuel Bengio

Computer Science
McGill University, Montreal

October 26, 2016

Acknowledgements

I would like to start by thanking all those who have fostered my creativity throughout my life, and gave me the freedom to explore the world as well as my own mind. It is a gift which I cherish.

I would also like to thank Joelle Pineau and Doina Precup for guiding me through my master's and keeping me focused on the tasks at hand, and Pierre-Luc Bacon for the initial idea about this whole project as well as for his enlightening insights on reinforcement learning.

Abstract

Deep learning methods have recently started dominating the machine learning world as they offer state-of-the-art performance in many applications. Unfortunately, the learning and evaluation of deep models is time-consuming, sometimes prohibitive, on resource-constrained devices such as phones and mobile hardware. Conditional computation attempts to alleviate this problem by selectively computing only parts of some model at a time.

This thesis investigates the use of reinforcement learning algorithms as a tool to learn the computation selection strategies inside deep neural network models. Efficient ways of structuring these deep models are proposed, as well as parameterizations for activation-dependent computation policies. A learning scheme is then proposed, motivated by sparsity and computation speed as well as the desire to retain prediction accuracy. By using a regularized policy gradient algorithm, policies are learned which allow stable performance at a lower computational cost. Encouraging empirical results are presented, suggesting that the general framework presented in this work is a sensible basis for the problem of conditional computation.

Résumé

Les méthodes d'apprentissage profond ont récemment commencé à dominer le monde de l'apprentissage automatique car elles offrent des performances à l'état de l'art dans de nombreuses applications. Malheureusement, l'apprentissage et l'utilisation de ces modèles profonds est très demandant en ressources computationelles, voir même impossible sur des plateformes avec des ressources limitées, tels que les cellulaires et autres appareils mobiles. Le calcul conditionel tente de règler ce problème en sélectionnant activement certaines parties du modèles à calculer.

Cette thèse explore l'utilisation d'algorithmes d'apprentissage par renforcement comme outil pour apprendre des stratégies de calcul conditionel à l'intérieur de réseaux de neurones profonds. Des manières efficaces de structurer ces modèles profonds sont proposées, ainsi que des paramétrisations pour des politiques de calcul conditionel dépendantes des activations neuronales. Une méthode d'apprentissage est proposée, motivée par le calcul éparse et le temps de calcul, ainsi que le désir de maintenir la précision des prédictions. En utilisant des algorithmes de *policy gradient* régularisés, les politiques de calcul apprises permettent d'obtenir des performances stables à un coût de calcul moindre. Des résultats empiriques encourageants sont présentés, suggérant que les algorithmes présentés dans ce travail sont une base sensible pour le problème du calcul conditionel.

Contents

List of Figures

List of Algorithms

1 Introduction

Even before the advent of computers and microcomputers, several ideas relating computation and biological neurons made their way into science. From the demonstration of logical formalisms through neurons by McCulloch and Pitts (1943) to the works of Hebb (1949), Rosenblatt (1958) and many others, the idea of *artificial neural networks* began to flourish.

Artificial neural networks are a class of functions that compute their output values using structures that are inspired by mammalian brains, made of interconnected networks of neurons each doing some part of the computations that give rise to intelligence.

It was only later in the 1970s and 1980s that the modern form of artificial neural networks began to be used. By reusing simple ideas from modern calculus (Newton, 1671; Leibniz, circa 1674) the idea of learning the configuration of artificial neural networks through gradients and most notably through the backpropagation algorithm, popularized by Rumelhart et al. (1988), increased the appeal of artificial neural networks.

Although appealing at that time, the failure of this so-called connectionism to deliver many promising results beating other contemporary machine learning methods apparently led the community to regard any neural based work with much skepticism.

In the early 2000s, publications based on artificial neural networks started gaining traction, with highly cited works works such as Hochreiter and Schmidhuber (1997), LeCun et al. (1998a), Hinton and Salakhutdinov (2006), *inter alia.* Artificial neural networks are now the state-of-the-art for many machine learning problems; because of their capacity for end-to-end learning, i.e. their intrinsic ability to reason about any given level of abstraction, they can be successfully applied to many problems beyond the reach of traditional machine learning algorithms.

The same era saw the advent of reinforcement learning. Concerned with temporal decision making, reinforcement learning found its origin in the 1950s with Dynamic Programming and the work of Bellman (1956). From trial-and-error methods in bandits (Robbins, 1952; Gittins, 1979), the field evolved to use methods based in temporal difference (Samuel, 1959; Barto et al., 1983), leading to many popular successes such as Tesauro (1995)'s TD-Gammon AI.

Reinforcement learning regroups many powerful learning algorithms and provides a framework to understand and learn how agents can take actions in their environment. There are many possible goals in such a framework. For example, given some agent's behaviours, it is possible to learn what reward model motivated it; or given reward signals, it is possible to learn which behaviours maximize that reward.

When combined with powerful learning methods such as artificial neural networks, reinforcement learning algorithms can achieve groundbreaking results in many domains, such as games (Mnih et al., 2013) and robotics (Levine and Abbeel, 2014).

Another direction which this alliance between artificial neural networks and reinforcement learning can take is to learn to make decisions *inside* neural networks (Mnih et al., 2014; Zaremba and Sutskever, 2015), which normally compute real numbers with differentiable functions, and do not allow for intermediary computations to be discrete (e.g. a yes/no decision). Such discrete decisions can be useful, but require techniques such as reinforcement learning to be learned.

This work is concerned with such a setting, where decisions on whether to perform computations will be learned using reinforcement learning, in the context of learning artificial neural networks. Taking such decisions based on the input in order to reduce computation time is called **conditional computation**.

In large artificial neural networks, it is common that most neurons are not activated at any given time. In fact, as hardware is getting better, state-of-the-art models get overly large and expensive to compute, and finding efficient ways of using these models is becoming a important research and application concern. As such, it is useful, in neural networks, to predict which parts are activated and necessary to compute, which this work attempts to solve under the framework of conditional computation. This gives rise to several problems: how to learn these activation patterns? how to group neurons together? how to organize computation in a hardware-efficient manner?

In the second and third chapter we introduce artificial neural networks and reinforcement learning, and explain how they allow models to learn from the world. In the fourth chapter, we introduce the notion of conditional computation in artificial neural networks and how it can be framed as a reinforcement learning problem. Finally, in the fifth chapter we show experimental results demonstrating the potential of this approach, describing future work and concluding in the sixth chapter.

2 Artificial Neural Networks

2.1 Supervised learning with function approximators

In this section, we define many basic concepts related to machine learning that are necessary to understand this work. A comprehensive and in-depth survey of these concepts is available in Murphy (2012).

2.1.1 Functions of data distributions

At the core of machine learning is the idea that there exist some natural data-generating distributions, typically denoted $\mathcal{D}$, from which we can collect samples, and which have some predictability.

A simple and interesting case is when samples from these distributions come in the form of pairs $(X, Y) \sim \mathcal{D}$, where $Y \in \mathcal{Y}$ is some desirable representation, or *label*, for an accompanying $X \in \mathcal{X}$. $\mathcal{X}$ and $\mathcal{Y}$ are sets representing the possible values that are generated by the data. In most cases, these sets can be reduced to *vectors* in $\mathbb{R}^n$.

For example, we can generate images using a camera and then later on label each image with a set of tags denoting the presence or absence of objects, say, we could distinguish between humans, dogs, cats, or cars. In this case, the space of Xs, $\mathcal{X}$, would be the pixel space, and the space of Ys, $\mathcal{Y}$, would be binary vectors of length 4, with each component denoting the presence (or absence) of each type of object.

In such a scenario, we say that there exists a function $f : \mathcal{X} \to \mathcal{Y}$ that maps each fixed sized *input* X to some representation Y such that for every actual sample (x, y) from $\mathcal{D}$ we have $f(x) = y$.

In fact, there exists a whole family of functions that map from $\mathcal{X}$ to $\mathcal{Y}$. Unfortunately, most of them do not have the latter property, where $f(x) = y$ for any sample of $\mathcal{D}$.

Finding a "good" function in this set of functions when we are given a number of (x, y) samples from $\mathcal{D}$ is what is typically called **supervised learning**, since the ys allow us to explicitly know the right answer and directly *supervise* learning.

2.1.1.1 The I.I.D. assumption

In supervised learning, an important assumption is made with regards to *how* the *data* is collected, before being fed to a learning algorithm. It is assumed that each example X, or example pair (X, Y), is collected from *independent and identically distributed* random variables.

What this means is that examples have no relation to one another, except for being drawn from the same distribution. They have no temporal relation and no causal relation (an example doesn't follow another).

This allows us to make several other assumptions which greatly facilitates modelling the data. A simple example is approximating expectations: given N i.i.d. data points $x_i \sim X$, say we want to estimate $\mathbb{E}\{X\}$, then (since the points are i.i.d.) the mean of the points is an unbiased estimate of the expectation.

2.1.2 Function approximators

When given some small $\mathcal{X}$ and some $\mathcal{Y}$, e.g. $\mathcal{X} = \{1, 2, 3\}$, it is easy to simply store all the possible answers. Unfortunately this scenario rarely ever happens in practice.

When faced with a large discrete input space, there is an incredibly large number of functions to chose from. Worse, in the continuous case, the set of functions becomes uncountably infinite. In any case, there are many more functions than what can fit in a computer, or be tested using a computer in less than the age of the universe.

As such, one solution to this problem is to use function approximators.

Instead of explicitly mapping each possible x to some y, function approximators use rules and algorithms to determine y given some x. It is common to make a function approximator f depend on a set of **parameters** $\theta = \{\theta_1, ..., \theta_n\}$ which are used to compute y given x. Given many pairs of (x, y) we might be able to find the value of each θ_i which *fits* the best.

Let us note a few things. First, even though the number of functions of the form $f(x; \theta)$ is not strictly smaller than the number of functions $f : \mathbb{R} \to \mathbb{R}$ (we are in an uncountable infinite setting), the number of significantly different configurations a computer has to try is effectively now much smaller (for a reasonable number of parameters).

Second, it is very probable that some random distribution $\mathcal{D}$ will not be *fit* very well by $f(x; \theta)$, if at all. We will revisit *fitness*, but in general, it is a measure of how well one function approximates another.

Last, we now have a sensible way of *exploring* this subspace of functions of the form $f(x; \theta)$. All that is necessary is to pick a value for each θ_i and see how it performs.

Using function approximators can be seen as a trade-off between finding exactly correct functions, and the number of functions that we have to try before finding the right one. It is especially true in the continuous input space case.

2.1.3 Linear models

Linear models are among the simplest models to learn functions from $\mathbb{R}^n$ to $\mathbb{R}$, or to $\mathbb{R}^m$. In the case of $f : \mathbb{R}^n \rightarrow \mathbb{R}$ they are typically denoted as:

$$f(x) = x^T w,$$

where $x \in \mathbb{R}^n$ is an *input vector* and $w \in \mathbb{R}^n$ is a parameter vector, or weight vector. It is common to also add a bias term:

$$f(x) = x^T w + \beta,$$

where $\beta \in \mathbb{R}$ is a scalar parameter. We denote the set of parameters as $\theta = \{w, \beta\}$.

In the case of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ we typically have this notation:

$$f(x) = x^T W + b,$$

where $x \in \mathbb{R}^{n\times 1}$ is the input, $W \in \mathbb{R}^{n\times m}$ is a *weight matrix*, and $b \in \mathbb{R}^m$ is a *bias vector*.

2.1.4 Learning linear models

Given some collection of n_D data samples $D = \{(x_1, y_1), ..., (x_{n_D}, y_{n_D})\}$ we may want to find which model performs the best according to some measure.

A common and easy-to-understand measure of *loss*, or *error*, is the so called mean-squared-error (MSE):

$$\mathcal{L} = \frac{1}{n_D} \sum_{i=1}^{n_D} \|f(x_i) - y_i\|_2^2$$

MSE is a proxy for the *fitness* of a function, meaning that as the error gets lower, the function is more apt to make predictions about unseen (x, y) pairs [1].

[1]This is only true if we make several assumptions, most importantly, the assumption that each (x, y) pair was sampled from *identically and independently distributed* (i.i.d.) random variables. The i.i.d. assumption is important in supervised learning, and is not always true, which we will see in particular in chapter 3.

Such proxies for error allow us to define *optimal* parameters, θ^*, as the parameters that minimize the given error measure $\mathcal{L}$ with respect to the given data D:

$$\theta^* = \text{argmin}_\theta \ \mathcal{L}(D)$$

In the case of linear models, finding such parameters is relatively trivial, since it is possible to find θ^* in *closed form*, for example in Ordinary Least Squares (see Dismuke and Lindrooth (2006)) the optimal W can be found by analytically finding the minimizing value:

$$W^* = (X^T X)^{-1} X^T Y, \tag{2.1}$$

where X is the matrix of x_is and Y the matrix of y_is.

Such a solution might not be stable, e.g. if X is ill-conditioned, making the inverse numerically hard to compute. An alternative way to find an optimal W is presented in Section 2.1.7.

Linear models are simple and fail to capture more complex distributions. For example a linear model cannot accurately perform regression on data generated from quadratic or exponential functions. Linear models can only separate two regions in the input space by a hyperplane, which again fails to capture more complex scenarios.

Most models that are able to capture additional complexity cannot be solved analytically. Their closed form solution (i.e. to compute θ^*) is either intractable or inexistant, and they require other techniques to find "optimal" (or reasonably good) parameters.

2.1.4.1 Feature engineering

One way to get around having to use more complex models is, given a problem, to transform input vectors into "meaningful" vector representations using hand-coded algorithms (Harris, 1954; Lowe, 2004). This is called feature engineering, and has for many years been vital to most machine learning applications.

For example, when using pixel-space inputs, each pixel gives very little information; it is only the combination of them that means something. As such, we could decide to extract the position of objects in the input image using computer vision algorithms (Bay et al., 2006), and then represent these as vectors for our linear model. We call these new vectors *features* and often denote them as $\phi(x)$.

Designing relevant features is hard. One of the explicit goals of machine learning research is to design methods that can greatly minimize this engineering effort, and thus save many hours of experts hand-designing features. If a computer can discover these features by itself, then we can spend more time designing the next level of complexity.

2.1.5 Non-linearities

It often happens that the output domain $\mathcal{Y}$ can be restricted to domains such as $[0,1]^n$ or $[-1,1]^n$. In this case it is often useful to "restrict" the possible output values of our function approximators as well.

The *logistic* function (which is a type of *sigmoid* function, but is often referred to as *the* sigmoid function in the literature, see LeCun et al. (1998b)) is a function from $\mathbb{R}$ to $[0,1]$, and is often denoted as $\sigma(x)$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Applying it on every element of a vector $\mathbf{x} \in \mathbb{R}^n$ effectively restricts the range of the outputs to $[0,1]^n$, and is also typically denoted as $\sigma(\mathbf{x})$.

Another commonly used non-linearity is the hyperbolic tangent function, tanh, which has range $[-1,1]$ and which also has this nice S-shape. In Neural Networks these non-linear functions are commonly referred to as *activation functions*.

The reason why such functions are preferable to simply clamping the output of our function approximators to $[0,1]$ or $[-1,1]$ is that they make the function **differ-**

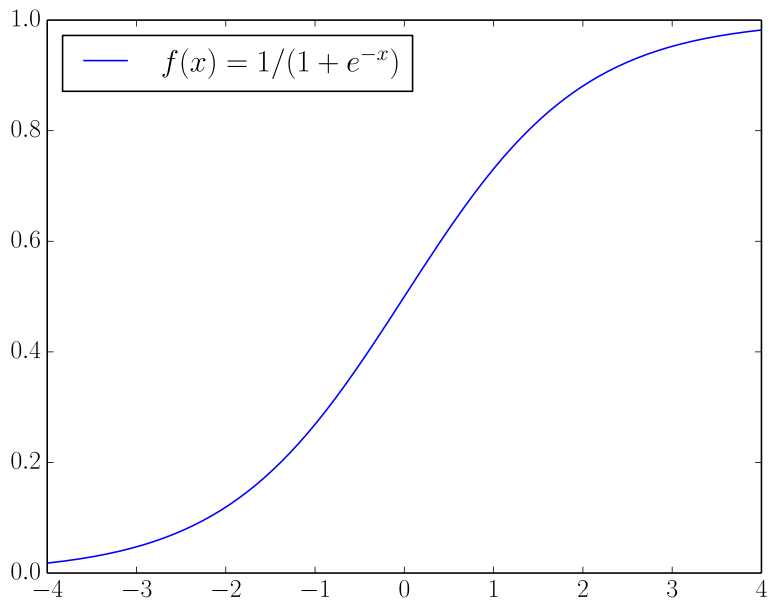


Figure 2.1: The logistic function

entiable. Combined with a differentiable loss or cost function, they allow *learning* to take place, as we will see in the next two sections.

Another commonly used function is the *softmax* function[2] . This function is useful when the output vector of our function approximators, Y, represents the probabilities of *mutually exclusive events*:

$$\mathrm{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} = p_i$$

where $\mathrm{softmax}(\mathbf{x})_i = p_i$ is the ith element of the output of the softmax, and where x_i is the ith element of the input of the softmax. Since $\sum_i p_i = 1$, each p_i can be interpreted as a probability, which can be very useful in many contexts including classification.

[2]which should really have been named soft-arg-max, since the max will have its associated value closest to 1 and the min closest to 0. This retains some (soft) measure of relatively how much the max really is the max, and tells us *which* value is the **argmax**, and has nothing to do with the *value* which is the "max" (in fact that information is lost through this transformation).

2.1.6 Gradient Descent

A common algorithm used in optimization is *gradient descent*, summarized in Algorithm 1. In its simplest incarnation, it is a procedure that iteratively finds some value x that minimizes (or maximizes, in gradient ascent) some value $f(x)$ by following the first order derivative. When x is a vector in $\mathbb{R}^n$, the vector of partial derivativatives is denoted $\nabla_x f = \left[\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n}\right]$ and is called the *gradient* of f with respect to x. Since it is not possible to take infinitesimal steps in a computer, these derivatives are "followed" at a certain rate α (with typically $\alpha < 1$).

```
initialize $x$ to some initial guess (can be random);
while $x$ has not converged do
    $x \leftarrow x - \alpha \nabla_x f(x)$
end
```

Algorithm 1: Gradient descent

If f is a convex function and in each iteration α is decreased in the limit to 0, x is guaranteed to be the global minimum of f. In the non-convex case no such guarantees are available. Fortunately, by using assumptions about f, such as Lipschitz continuity, it is possible to reach interesting local minima using gradient descent (see Bottou (2010)).

Note that here *convergence* is meant in the broad sense of "x does not change by more than some ϵ". In practice, there are more precise measures of when and how to stop gradient descent, even when x might still be changing (see for example Yao et al. (2007)).

2.1.7 Learning differentiable models

The unfortunate side effect of using non-linearities (or any more complex models), is that they make closed-form solutions impractical or even non-existent. Fortunately, when these non-linear models are differentiable, they can still be learned from data.

Given a loss function such as the MSE, it is easy to find the gradient of the loss:

$$\mathcal{L} = \frac{1}{n_D} \sum_{x,y \in D} \sum_i (f(x)_i - y_i)^2$$

with respect to the **parameters**, W and b that form $f(x)$:

$$f(x) = \sigma(x^T W + b)$$

for example in this particular case, the gradient of $\mathcal{L}$ with respect to W, denoted $\nabla_W \mathcal{L}$ has the form:

$$\nabla_W \mathcal{L} = \frac{2}{n_D} \sum_{x,y} \left[(f(x) - y) \odot f(x) \odot (1 - f(x)) \right] x^T$$

Where $\odot$ represents the element-wise matrix multiplication, also known as the Hadamard product.

When σ is replaced with the identity function, we have a linear model, with the following gradient:

$$\nabla_W \mathcal{L} = \frac{2}{n_D} \sum_{x,y} (f(x) - y) x^T \tag{2.2}$$

By using gradient descent, as defined in Algorithm 1 above, we can follow these gradients in order to *minimize* the loss function $\mathcal{L}$. In this setting α is some small number, typically well below 1, and is often referred to as the **learning rate**.

Note that by following the gradient in (2.2), the same solution as (2.1) is obtained in convergence, without the problems of ill-conditioned matrices.

By following this gradient, we minimize the loss function $\mathcal{L}$ and thus increase the fitness of our model. This simple method, using gradient descent, is used to learn a wide variety of models, simple and complex, which have the property of being differentiable.

2.1.8 The bias-variance trade-off

Given a complex enough problem, a complex model of the data is necessary. This creates a new problem, which is to choose the appropriate model complexity.

A common approach when designing machine learning algorithms, and selecting the best model out of many, is to separate the available data into three sets: **training** set, **validation** set, and **testing** set.

The training set is used to train the models and their parameters (it is the data that the learning algorithm sees). When all the models are trained, we can compare their performance on unseen examples from the validation set to see how well they generalize. Doing this many times might bias our selection; as such, it is necessary to report the performance of the best selected model on the last set, the test set.

We can also characterize this problem in terms of bias and variance: given a dataset $D = \{(x_1, y_1), (x_2, y_2), ...\}$, and a parameterized model $f_\theta : \mathcal{X} \to \mathcal{Y}$, if $f(x_i)$ is very close to y_i for every example, we say that the model has **low bias**. The predictions that it makes are exact, at least for the examples that it knows. If on the other hand $f(x_i)$ is on average rather far from y_i, then the model has **high bias**.

Consider that we remove a some examples from the training dataset, and put them in another set V (the **validation set**). If the resulting function changes drastically when removing these examples (in V), then we say that it has **high variance**.

Models with high variance can be in the regime of **overfitting**. If changing the dataset slightly completely alters the function, it means that the function probably learned each example by heart, and will generalize poorly to unseen examples.

Models with high bias can be in the regime of **underfitting**. One can see this as the model generalizing too much, and making broad assumptions about the data when in reality the input distribution is more complex.

Note that the complexity of a model is typically controlled by various values, e.g. the number of neurons in a neural network. Such values are called **hyperparameters** because they cannot be directly optimized by the learning algorithms, and as such are typically compared by retraining the model with several different values and comparing the resulting validation errors, as explained earlier.

2.1.9 Regularization

Regularization is key in many machine learning approaches. It is often easier (mainly for optimization reasons) to build and learn models with potentially high variance, but to **regularize** them.

The goal of regularization is to push optimization in a certain direction that makes the learned model have certain desirable properties. For example it can be desirable to prevent a model from learning some input-output (x, y) pairs by heart, since we want models to generalize. It can be desirable to force a model not to use all of its input features if we suspect that some of them are not indicative of the output (but don't know which ones).

Regularization is a widely studied subject in statistical estimation; the most common one that is found in machine learning is weight decay, also known as L1 (Tibshirani, 1996) or L2 regularization (also known as Tikhonov regularization, see Ng (2004)). L2 regularization consists in adding a term to the loss which penalizes the squared Frobenius norm of the weight vector:

$$\mathcal{L}_{L2} = \sum_i \theta_i^2$$

while L1 regularization consists in the sum of the absolute values of every θ_i:

$$\mathcal{L}_{L1} = \sum_i |\theta_i|$$

Optimizing a regularized model is typically described as minimizing such as loss function (in this case L2 for MSE):

$$\mathcal{L} = \sum_{i=1}^{n} \|f(x_i;\theta) - y_i\|_2^2 + \lambda \sum_{i=1}^{n_\theta} \theta_i^2$$

where λ is a hyperparameter which decides the strength of the regularization.

One effect of these parameter norm regularizations is that it becomes harder for the model to learn extreme weight values and overfit to the data, i.e. fitting (x, y) pairs in the training set exactly at the expense of predicting extreme values for xs that are not in the training set, effectively learning output values by heart. Choosing λ is not obvious, if λ is too small and the supervised loss outweighs the regularization loss, then the model will always learn to use the available complexity, if λ is too large, the model might not learn anything.

2.2 Depth and distributed representations

The function space of models defined with a single matrix multiplication and non-linearity is quite restricted. One of the great machine learning contributions of the 1990s, and more significantly the first decade of the 2000s, was the effort towards making artificial neural networks a mainstream technique, developing methods for them which became known as **Deep Learning**.

2.2.1 The Multilayer Perceptron

The initial intuition behind artificial neural networks (and a manifestation of the model that we have seen in Section 2.1.3) was originally known as the Perceptron (Rosenblatt, 1958). The Perceptron is a simple two-class classifier, with the form:

$$f(x) = \text{class}(x) = \begin{cases} 1, & \text{when } x^T\theta + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

accompanied by a simple learning rule:

$$\theta_i \leftarrow \theta_i - \alpha(y - f(x))x_i$$

which, if we look closely, is the same learning rule as a linear model (with a single output unit) when using the gradient of the MSE, as in (2.2).

Such a model is commonly depicted as a series of units, representing each dimension of the input vector x, linked to a single *output unit* representing the resulting computation $f(x)$, as in Figure 2.2.

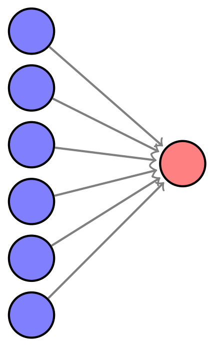


Figure 2.2: The Perceptron, here depicted with 6 input units and a single output unit.

The problem with such a simple model, is that it can only linearly separate the input space $\mathcal{X}$, and cannot solve many problems, such as the infamous XOR problem. There is no parameterization of θ than can correctly classify the XORing of two bits (i.e. predict 0 for [0;0] and [1;1], and predict 1 for [0;1] and [1;0]).

A solution to this problem is to add *intermediate layers*, or **hidden layers**, to the model. This idea was already discussed in the early 1960s, and notably made its way in machine learning through the works of Werbos (1974) and Rumelhart et al. (1988).

For example, a single hidden layer network is formulated as:

$$h(x) = \sigma(x^T W_h + b_h)$$
$$f(x) = \sigma(h(x)^T W_o + b_o)$$

$h(x)$ is typically referred to as *hidden units*, hidden activations, *feature vectors*, intermediate representations, or inner representations. Note that the set of all the parameters (all W_i matrices and b_i vectors) is often represented as a single vector θ. Such an architecture is commonly referred to as multilayer perceptrons (MLP), and as fully-connected architectures (due to the dense nature of matrix multiplications, each neuron is connected to all other neurons in its following layer).

It is possible to add as many intermediate layers as desired, but of course more layers are *harder* to learn (see Section 2.3.4; Hochreiter (1991); Bengio et al. (1994)), at least without many tricks (some of which we expose in Section 2.3). This is equivalent to composing many functions, for example a n layered model:

$$f(x) = (f_n \circ ... \circ f_2 \circ f_1)(x)$$
$$\text{where } f_i(h) = g_i(h^T W_i + b_i)$$
$$g_i \text{ is the activation function of layer } i$$

The computation of $f(x)$, done by first computing the activations of the first layer, then the next up until the last, is called **feed-forward propagation**.

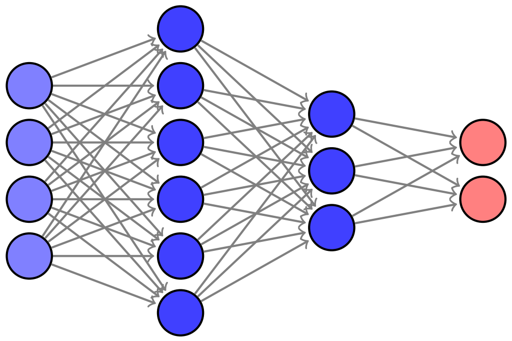


Figure 2.3: A multilayer perceptron with 2 hidden layers, one of 6 units, one of 3 units. The output layer has 2 units.

This model is useful for many reasons. First, it can solve the XOR problem very simply. Second, it is capable of representing a wide variety of functions, as proved by Cybenko (1989)'s universal approximation theorem (also Hornik (1991)), many orders of magnitude more complex than the XOR function. Finally, it is *efficient* because it can learn distributed representations.

These multi-layered models give rise to *depth*, and the techniques that are associated with them are regrouped under the umbrella of *Deep Learning* (LeCun et al., 2015; Goodfellow et al., 2016).

2.2.2 Convolutional Neural Networks

Another popular type of layer that is used for any image-related task (and some signal related tasks, even language) is **convolutional** operations (Fukushima, 1980; LeCun et al., 1998b; Krizhevsky et al., 2012). The convolution between a k-channeled image I and a set W of n k-channeled filters of size $w \times h$ is defined as:

$$(I * W)_{nxy} = \sum_{c=0}^{k} \sum_{i=0}^{w} \sum_{j=0}^{h} I_{cuv} W_{ncij}$$

$$\text{where } u = x + i - \lfloor \frac{w}{2} \rfloor \quad v = y + j - \lfloor \frac{h}{2} \rfloor$$

There are many reasons why this computational structure (illustrated in Figure 2.4) is strong, especially when dealing with images. Notably, it imposes a strong prior on the neural network: patterns repeat themselves across images. As such, there is no need to relearn the same feature detector to detect the same pattern at different locations in an image since, by its nature, the convolutional operator will allow the model to detect a pattern whatever its location. There are also many clear mathematical properties of convolution that make such a structure elegant when dealing with signals. In addition, there are many computational advantages to using convolutions since the computation is very dense, and well suited for modern hardware to take advantage of.

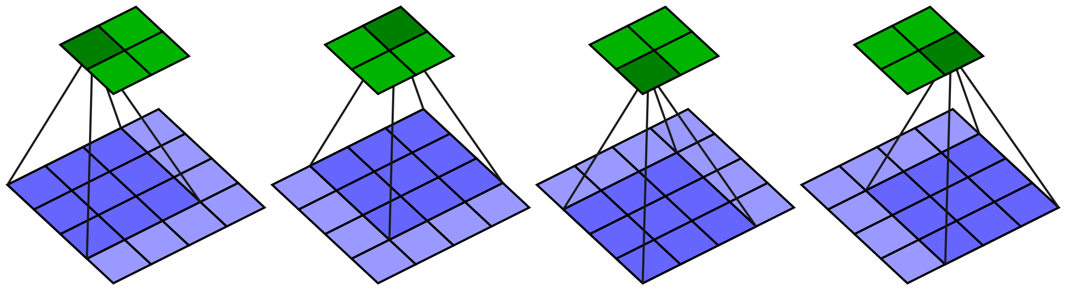


Figure 2.4: Illustration of the computations in a convolutional network, with $n = 1$, $k = 1$, $w = 3$, $h = 3$, and from left to right $(x, y) \in \{(0, 0), (1, 0), (0, 1), (1, 1)\}$. See Dumoulin and Visin (2016) for a complete reference.

2.2.3 Distributed Representations

Consider, in a multi-layer perceptron, that a given hidden unit h_j receives all its input units through the vector of weights $W_{:j}$, and that upon seeing its input it takes a value that is either positive or negative. In a sense this unit "detects" whether the input matches $W_{:j}$ or not. This behaviour is commonly described as the unit being a **pattern detector**.

By having n_1 hidden units connected to the input, we can detect n_1 different *independent*[3] patterns (of which there are 2^{n_1} possible combinations). As such the **representation** for the input (which is the vector h obtained) is **distributed** among each component of the vector.

Consider a hidden unit in the next layer. By the same reasoning it will detect a "pattern of patterns". Intuitively, we can "choose k" from n_1 patterns. With n_2 hidden units in this second layer, one can imagine that we can roughly identify $O(n_1 n_2)$ patterns from the input space. In fact for every layer we add, we gain exponentially in the number of input patterns we can "detect".

This intuition was recently formalized by Montufar et al. (2014) in terms of the number of separable linear regions in the input space. They showed that for a deep model with L layers of width n, the maximal number of linear regions per parameter grows exponentially fast:

$$\Omega\left(\left(n/n_0\right)^{n_0(L-1)} \frac{n^{n_0-2}}{L} \right)$$

Another intuition for representations is the following. At each layer, the network combines certain components of these distributed representations to create a new representation. This is akin to increasing levels of abstractions. As we go deeper in the network, we go from low-level *features* (e.g. in image space horizontal or diagonal lines) to high-level, abstract features (e.g. eyes, smiles, chairs, etc.).

2.2.4 Learning Deep Neural Network models

Learning of Deep Neural Network (DNN) models is typically done with gradient descent, and variations of it. Most importantly, in order to be efficient, it is done

[3]Here independence is not meant in the probabilistic sense, but in the sense that each pattern captures a different factor of variation of the data (and factors of variation are often correlated with one another, but can be disentangled nonetheless).

with the backpropagation algorithm Rumelhart et al. (1988).

Backpropagation simply arises from the chain rule in derivatives, which is that:

$$\frac{\partial a}{\partial b} = \frac{\partial a}{\partial x_1}\frac{\partial x_1}{\partial b} = \frac{\partial a}{\partial x_1}\frac{\partial x_1}{\partial x_2}..\frac{\partial x_{n-1}}{\partial x_n}\frac{\partial x_n}{\partial b}$$

Consider the loss $\mathcal{L}$ in a DNN that depends on the input and the weights W_i, b_i of each layer. To find $\frac{\partial \mathcal{L}}{\partial W_i}$ we can use the chain rule to decompose it in terms of

$$\frac{\partial \mathcal{L}}{\partial h_i}\frac{\partial h_i}{\partial W_i}$$

Using the same reasoning we can decompose $\frac{\partial \mathcal{L}}{\partial h_i}$ in terms of the partial derivatives of the following layers $j > i$.

As such if there are L layers, one starts by computing $\frac{\partial \mathcal{L}}{\partial h_L}$, then $\frac{\partial \mathcal{L}}{\partial h_{L-1}}$, down to the first layers. This is called backwards propagating through the network, or **backpropagation**, because the "error signal", the gradient, is computed from top (layer L) to bottom (input layer), in the inverse order in which the feed-forward propagation happened.

Using such a technique is much more efficient than explicitly computing each gradient term separately, because most computations can be reused as one goes down the layers. Additionally, it is more efficient because computing the full product of partial derivatives when done from left to right (i.e. in backpropagation order) uses matrix-vector operations, while computing it right to left would require matrix-matrix operations, making it n times more costly for exactly the same result.

Once we have the gradient terms $\nabla_W \mathcal{L}$, we can apply the gradient descent algorithm to learn the optimal parameters from data:

$$W \leftarrow W - \alpha \nabla_W \mathcal{L}$$

2.2.4.1 Minibatch stochastic gradient descent

In the previous section we considered a loss such as the Mean Squared Error, which is the average error over *every* example in our dataset D.

If D is very large (thousands or millions of examples), it is impractical and inefficient to compute the updates in gradient descent over every example in D, and only then apply an update. This technique is called *batch gradient descent*.

Instead, one can randomly pick a single example in D, compute the loss according to that example and then do the updates. This is known as **stochastic gradient descent** (SGD), and has many advantageous properties Bottou (1998).

It is computationally inexpensive to compute the updates for a single example versus the whole dataset, and the resulting gradient is still representative of the "right direction" (mainly because the example is from a distribution and not noise).

Learning incrementally with different –i.i.d. sampled– examples, has the effect of adding noise to the weight exploration, and fortunately this noise "smooths" the non-convex error curve, making it easier for SGD to find locally optimum solutions.

In terms of computation time until convergence, SGD is much faster than batch GD (see Bottou (2010)), mainly because even though the batch GD gradient is a better approximation of the expected gradient, it is only relevant information near the current weights. Since we do not know the curvature, i.e. the second-order derivative, the gradient will most likely not remain the same as we move away from the current weights, even if by very little. As such, one cannot take larger steps than a certain threshold, and so it is wasteful to do batch GD since we could not compensate longer computation times of a single update by a larger learning rate.

An interesting observation is that, in SGD, after a few iterations the weights often do not change by a lot. As such, it is possible to compute the SGD update for many examples at a time (using the same weights) in parallel. This transforms the normal vector-matrix operations into matrix-matrix operations, which are much faster with modern hardware. This is called **minibatch SGD**.

With a minibatch of size k, this method roughly does the equivalent of k SGD updates, but in much less time than k SGD updates. k is typically a small power of 2, from 16 to 128, depending on the size of the input, so as to maximize the throughput of matrix-matrix computations. There is a certain limit after which, for the same reasons as above with batch GD, one cannot increase the learning rate and k, as it becomes wasteful to compute the gradient for too many examples at a time.

2.2.4.2 Local minima and saddle points

For a long time, DNN approaches (and non-convex optimization problems in general) were perceived to have a very large weakness. When doing gradient descent in a non-convex setting, it seems intuitive the one would get stuck in local minima (Baldi and Hornik, 1989; Gori and Tesi, 1992), valleys in the mountains of the error curve, far away from the global minimum, where in every direction the error seemed to go up. This was disproved recently by Dauphin et al. (2014).

In a deep model setting there are roughly $N = Ln^2$ parameters, L the number of layers and n their width, which for reasonable settings quickly sums up to millions of parameters.

Consider the following intuition: gradient descent navigates in $\mathbb{R}^N$, and each point in this space is associated with a scalar loss. Additionally this space is rather smooth, two neighbouring points often have very similar loss values.

SGD navigates this space in a semi-random way. In our 3-dimensional intuition, it is easy to picture a valley which would be a local minimum, in fact if we picture a mountainous terrain (such as in Figure 2.5), often it is mostly that. In millions of dimension, this mental image is mostly wrong. What is the probability that for *every* dimension going left or right will only increase the error? In very high dimension it is close to zero.

In fact what Dauphin et al. (2014) have found is that SGD in neural networks mostly converges not to local minima, but saddle points, or flat regions in the error-parameter space. Choromanska et al. (2015) also showed similar results for functions related to neural networks. Fortunately, knowing this, there are many techniques to get past these flat regions (also addressed in Dauphin et al. (2014)).

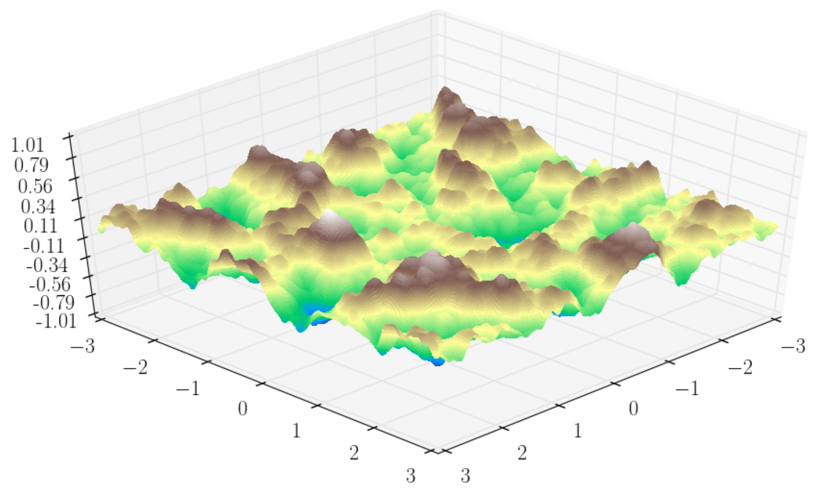


Figure 2.5: A 3-dimensional landscape

2.3 Modern Tricks in Deep Learning

One of the main propellants of Deep Learning was the increasing power of hardware, and the massive availability of data, two very important ingredients to learn complex models that generalize well to unseen situations.

Another important part of the success of deep models is the set of techniques that guide SGD into better solutions. Although many of those techniques do not have proper theoretical foundations, there is often a strong intuition as to why they work.

2.3.1 Dropout

Hinton et al. (2012) introduced **dropout**. This method consists in randomly dropping (multiplying by 0) activations with some probability p (which may be different for every layer) during the feed-forward pass during training only, as to make the model more robust at test time.

The effect that this has on a DNN is that two given units cannot *coordinate* or *co-adapt* during training. This effectively adds redundancy in the model, but also heavily prevents it from learning by heart. Using dropout seems to allow building larger models. More neurons are necessary for redundancy, but at the same time bigger networks can model more complex input distributions, all without overfitting.

Dropout is a regularization mechanism that allowed several models to beat state-of-the-art results (Hinton et al., 2012; Wager et al., 2013; Srivastava et al., 2014) because it allowed larger models to be built (able to capture more complexity in the data) that would not overfit, and is still heavily used in Deep Learning (He et al., 2015b).

2.3.2 Autoencoders and pretraining

An autoencoder (Ballard, 1987; Bengio et al., 2007; Vincent et al., 2008) is a function $f : \mathcal{X} \to \mathcal{X}$ which given some input x reconstructs it into $f(x) = \hat{x}$ such that x and $\hat{x}$ resemble each other. This is typically done in two steps of computation, first an encoder $f : \mathcal{X} \to \mathcal{H}$ encodes x into a hidden representation h, and then a decoder $g : \mathcal{H} \to \mathcal{X}$ attemps to rebuild x from h

Achieving such a model is interesting when the inner representation h of the model is a bottleneck: i.e., it has many less dimensions than the input. It means that the model managed to learn a representation of the input that is smaller than the input. In a sense, it is doing compression.

It is often the case in machine learning that there is an abundance of data, but very little of that data is *labeled*, meaning that we have an abundance of xs, but little (x, y) pairs.

As such, it is interesting to train models that are both able to do the prediction *and* the reconstruction of the unlabelled data, because forcing the model to also reconstruct other data means it will have access to more samples and will be much less likely to overfit. Instead of directly using the inputs to predict, we can reuse the encoder (or share weights) and build a function $f_{pred} : \mathcal{H} \to \mathcal{Y}$ that takes as input the hidden representation of the autoencoder, and predicts a value y.

Another possibility is to first train an autoencoder, and then reuse the weights of the autoencoder to train another predictive model. This is called **pretraining**, and can be a powerful regularization mechanism as well, but is rarely used anymore. Indeed, training a model with both the reconstruction cost and a prediction cost at the same time (without pretraining) can by itself lead to impressive results even with very few labelled examples (Rasmus et al., 2015).

2.3.3 Momentum and adaptive gradient descent

Momentum (Polyak, 1964) is a variant of the SGD algorithm. The idea is to maintain, for each parameter, the overall direction in which gradient descent leads them. This is usually maintained through an exponential moving average, μ.

Momentum gradient descent is as follows:

$$\begin{aligned}
\mu_0 &= \mathbf{0} \\
\mu_t &= \gamma \nabla_\theta \mathcal{L} + (1-\gamma)\mu_{t-1} \\
\theta_t &= \theta_{t-1} - \alpha \mu_t
\end{aligned}$$

where t represents the time of each minibatch, and $\gamma \in (0, 1]$ represents how much each new gradient update counts towards the average.

Another aspect of gradient descent that is changed in modern methods is the learning rate. It can be desirable to adapt the learning rate on a per-parameter basis, instead of globally. One such algorithm is RMSProp (Tieleman and Hinton, 2012).

RMSProp maintains an exponential moving average μ of the root mean squared gradients, and divides the current gradient by it (as such it is akin to a learning rate).

$$\begin{aligned}
\mu_0 &= \mathbf{1} \\
\mu_t &= \gamma (\nabla_\theta \mathcal{L})^2 + (1-\gamma)\mu_{t-1} \\
\theta_t &= \theta_{t-1} - \alpha \frac{\nabla_\theta \mathcal{L}}{\sqrt{\mu_t + \epsilon}}
\end{aligned}$$

where ϵ is a small number that prevents division by 0.

As for momentum, there are many variants of adaptative learning rate methods (see Schaul et al. (2013) and citations within), but they all revolve around these main ideas.

Both momentum and RMSProp address the problem of the saddle point, as exposed in Section 2.2.4.2. When the optimization gets "stuck" in a saddle point, for a small enough γ momentum will simply continue in the same direction because of its inertia, while RMSProp will quickly increase the learning rate of the "stuck" parameters, allowing them to move away from these saddle points (Ruder, 2016).

2.3.4 Batch normalization

Batch normalization (Ioffe and Szegedy, 2015) also addresses an optimization problem, yet in a different way. One important problem in deep models is that in backpropagation, gradient information typically becomes weaker and weaker (Hochreiter, 1991; Bengio et al., 1994) as one goes further away from the source of the error (i.e. the top layer).

Batch normalization uses minibatch SGD to readjust the values of each activated unit so that it is normally distributed with respect to the other activations of the same unit in the minibatch. For a hidden layer computed as

$$H = XW + b$$

where X is the matrix of the k examples in a minibatch, or the activations of a previous layer, and H is the resulting matrix of n activated units, H is corrected so that H_{ij} becomes:

$$H'_{ij} = \frac{H_{ij} - \frac{1}{k}\sum_u^k H_{uj}}{\sqrt{\mathrm{Var}_u H_{uj}}}$$

Intuitively, this allows the activations of each layer to be nicely distributed, and thus for the gradient to propagate better than if the values of each layer were not 0-centered and with a constant standard deviation.

In practice, using batch normalization speeds up learning, which is a potential side effect of having “better” gradients.

2.3.5 Rectifiers

Rectifying linear units (Glorot et al., 2011), or ReLUs, are also another tool that allow for better gradient propagation, they are an alternative to previously popular activation functions such as sigmoids and hyperbolic tangents. Rectifiers are units with the following activation function:

$$r(x) = \max(0, x)$$

While strange at first, this activation function became popular as models using them achieved much lower errors Glorot et al. (2011); He et al. (2015b). It is strange, since it is essentially a linear unit (thus we lose the smoothness of sigmoidal functions), but with a cutoff at zero.

One reason why ReLUs might be performing so well is that in the positive regime, the slope (the partial gradient) of this activation unit is always 1 (as opposed to sigmoid and tanh, where it would gradually become 0 as the activation got larger). In backpropagation computations ReLUs remove a multiplication by the slope of the activation, which is typically a small number, and thus deep models with rectifier units also possibly learn faster (and better) because gradients tend not to vanish as much with depth (He et al., 2015b).

2.4 Deep Learning

Deep models learn progressively more abstract features, in depth, by stacking varied kinds of simple layers. They are well suited to extract simple representations from complex inputs, making them an appealing learning framework and active area of research.

Deep Learning is a young and fast evolving field. This chapter only mentions the few concepts necessary to understand the rest of this work, but there is a wide variety of architectures, loss functions, regularizations, optimization methods, and theoretical analyses that revolve around Deep Learning. A comprehensive survey of these facets of Deep Learning can be found in Goodfellow et al. (2016).

Notably, even though this work has some relation to Dropout, we will not cover its more theoretical aspects (Gal and Ghahramani, 2015). There are also many more optimization strategies available to artificial neural networks (Kingma and Ba, 2014), but since these might interfere with our analysis of this work, they are not used.

Finally, there is a vast litterature concerning Convolutional Neural Networks (Krizhevsky et al., 2012; He et al., 2015b; Huang et al., 2016) and what they learn (Zeiler and Fergus, 2014; Li et al., 2015).

3 Reinforcement Learning

3.1 The Reinforcement Learning Setting

Reinforcement Learning (RL) is concerned about decision making, both in immediate and temporal settings. Note that the material presented in this section is but a small subset of RL methods, necessary to understand the rest of the work. Thorough explanations of the material are found in the works of Sutton and Barto (1998) and Szepesvári (2010).

RL is commonly viewed as reducing problems to the agent-environment setting, illustrated in Figure 3.1. An **agent**, which can for example be a collection of sensors and actuators or a game playing program, resides in an **environment**. At each discrete time-step t, the agent receives information about its state. From that information, it makes an action, which will (possibly) influence the environment, resulting in a new state observation, along with a reward.

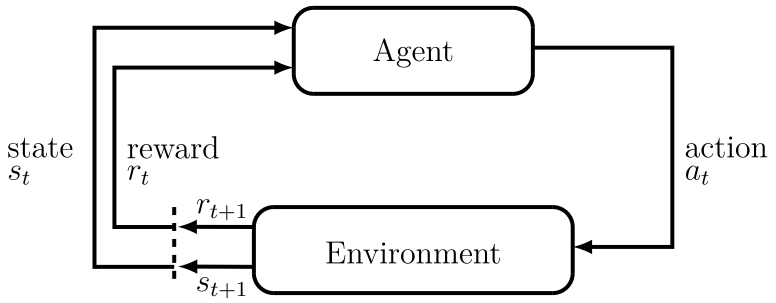


Figure 3.1: The RL Environment diagram

Often, the goal will be for the agent to learn to maximize its reward. To do so it may have to overcome several challenges, such as delayed rewards, partially observable states, or vast state and action spaces.

3.2 The Markov Decision Process Setting

Markov decision processes (Bellman, 1956) are a formal tool used to describe problems in reinforcement learning, as they are well suited to describe total-information stochastic (or deterministic) environments.

A Markov decision process (MDP) M is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$:

- $\mathcal{S}$ is the (possibly infinite) set of states;
- $\mathcal{A}$ is the (possibly infinite) set of actions; $\mathcal{A}$ can also be state-dependent;
- $\mathcal{P}(s, s', a)$ is the transition probability from s to s' given that action a is taken;
- $\mathcal{R}(s, s', a)$ is the reward received from action a when going from s to s'.

Markovian processes are interesting because they follow this rule:

$$P(s_t|s_{t-1}, a_{t-1}) = P(s_t|s_{t-1}, a_{t-1}, s_{t-2}, ..., s_0, a_0)$$

which is that the next state only depends on the current state. An agent living in a Markovian environment only needs to know about the most recent state in order to know about the future. It does not require memory (at least in principle). In this setting, a sequence of state-action-reward tuples is described as a *trajectory* τ.

3.2.1 Policies and Stochastic Policies

Consider a memory-less agent which at each moment in time receives the current state information, and gets to choose an action. This behaviour can be formalized as a **policy** $\pi : \mathcal{S} \rightarrow \mathcal{A}$, which maps states to actions, indicating what to do at each moment. In a more general way, we can formulate a **stochastic policy** as a conditional probability distribution $\pi(a|s)$; given a state there is a distribution over each action. We will assume that policies are stochastic for the rest of this work.

It is often the goal of reinforcement learning methods to find a policy which is optimal with respect to the environment that is given, or to find a policy which matches the policy of an expert agent (e.g., a human). A policy which is as good as or better in expectation than all other policies, in terms of accumulating reward, is called an *optimal* policy.

In a stochastic –or simply unexplored– environment it can be desirable to have a stochastic policy, that is, to take actions according to some state-dependent probability distribution $\pi(a|s)$. A simple case of such a policy is a Bernoulli policy, where one of two actions is chosen at random according to some probability p. A slightly better policy might be to make this probability input-dependent, and compute it as a function of the state. Stochastic policies make deriving some learning methods harder, and others simpler, simply due to the nature and properties of random variables.

3.2.2 Value Functions

It is an interesting problem, and often a necessary step, to be able to judge *how good* a given state is. The **value** of a state is defined recursively as:

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

It is the expected cumulative reward (also known as *return*) that can be received in s and in all following states when following the policy π.

It is also possible to measure *discounted reward*, and instead of seeing infinitely far in the future in terms of reward, prioritizing nearer states by an exponential factor γ:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} \mathcal{P}(s,s',a)\left[\mathcal{R}(s,s',a) + \gamma V^\pi(s')\right]$$

One can also express the **action-value** function, the expected cumulative reward when in state s and taking action a

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}(s,s',a)\left[\mathcal{R}(s,s',a) + \gamma V^{\pi}(s')\right] \tag{3.1}$$

It is desirable to learn or measure the value function of an MDP, as it can allow an agent to chose which state to visit next if it wants to maximize reward. Unfortunately, in complex environments this can be a hard, or even intractable problem.

Under the optimal policy, the value functions are called *optimal* value functions, V^* and Q^*, since they represent choosing the optimal action at each time-step.

$$Q^*(s,a) = \sum_{s'} \mathcal{P}(s,s',a)\left[\mathcal{R}(s,s',a) + \gamma V^*(s')\right]$$

$$V^*(s) = \max_a Q^*(s,a)$$

The simplest policy that can be used in conjunction with a value function is the **ϵ-greedy policy**, which simply consists in taking the action which leads to the state with highest value function, but take a uniformly random action with probability ϵ.

In the case where $\epsilon = 0$, it is simply the greedy policy.

3.3 Learning Value Functions

There are many algorithms that can learn value functions (Sutton and Barto, 1998), most of which we will not cover. As described in Section 2.1.2, it is possible to learn a tabular value function if the number of states (and possibly the number of actions) is small enough, or is discretizable into a small set.

In the other case, if the number of states is continuous or too large, it is also possible to use function approximation to learn $V(s)$ or $Q(s,a)$. Doing so poses a major problem: when going through an MDP we can collect *trajectories* – sequences of states, actions and rewards – as a data set, but while one can consider trajectories to be i.i.d., the tuples (s, a, r, s') of which they are made *are not i.i.d.*, which makes them unsuitable for typical function approximation learning methods.

Fortunately, there are ways to mitigate this problem, such as using experience replay (Lin, 1992), which makes it possible to learn function approximators on large input spaces (Mnih et al., 2015).

The Q-Learning (Watkins, 1989) algorithm is an iterative way of learning Q functions. An intuitive derivation of the algorithm follows.

Consider having a parameterized estimate of the Q function $Q_\theta(s, a)$, and consider that we are under the ϵ-greedy policy. We can now rewrite (3.1) as:

$$\begin{aligned}
Q^\pi(s,a) =& \sum_{s'} \mathcal{P}(s,s',a)\left[\mathcal{R}(s,s',a) + \gamma V^\pi(s')\right] \quad (3.1)\\
Q_\theta(s,a) =& \sum_{s'} \mathcal{P}(s,s',a)\mathcal{R}(s,s',a) + \sum_{s'} \mathcal{P}(s,s',a)\gamma V^\pi(s')\\
& \text{under a greedy policy } \pi,\ V_\theta^\pi(s') = \max_a Q_\theta(s',a)\\
=& \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{P}(s,s',a) \max_a Q_\theta(s',a)
\end{aligned}$$

Now consider that we are constantly sampling trajectories from the environment. The transition probability $\mathcal{P}(s, s', a)$ is "naturally" taken into account from the sampling. For a given sample (s_t, a_t, r_t, s_{t+1}), where $r_t \sim \mathcal{R}(s_t, a_t)$, the following should hold true (assuming θ is reasonably good):

$$Q_\theta(s_t, a_t) \approx r_t + \gamma \max_a Q_\theta(s_{t+1}, a)$$

Since at the beginning of training Q_θ will be mostly wrong, this will in fact lead to an error term:

$$\delta = |r_t + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t)|$$

To reduce the error, one can update Q_θ to explicitly minimize δ using the previously seen supervised learning methods[1], or in the tabular case, update $Q(s, a)$:

$$Q^{k+1}(s_t, a_t) \leftarrow Q^k(s_t, a_t) + \alpha[r_t + \max_a Q^k(s_{t+1}, a) - Q^k(s_t, a_t)]$$

[1]This is not as simple as it may look, mainly due to the data distribution used in learning **not** being i.i.d., and due to the bootstrapping problem. There are various methods to make this work, see Mnih et al. (2015) for example.

where α is the learning rate, at which the values of Q are updated.

3.4 Learning Policies with Policy Gradient Methods

In some systems, it can be advantageous to learn policies *directly*, instead of first learning value functions and selecting optimal actions. Just as for value functions, if the state and action spaces are discrete (or discretizable) and small enough, it is possible to learn an explicit mapping $\mathcal{S} \to \mathcal{A}$ or as a stochastic mapping $\pi(a|s)$.

In many cases, most commonly when the state space is too large, it is necessary to use function approximation. For this, it is necessary to have a parameterization of the policies either as a deterministic $f_\theta : \mathcal{S} \to \mathcal{A}$ or as a stochastic function $\pi_\theta(a|s)$.

Learning policies directly has a few advantages, notably when the value functions are hard to approximate or when a stochastic policy is desired. In some cases, when the end-goal is to learn an optimal policy, it can simply be more convenient, and reduce the number of parameters that one has to learn.

Just as it is possible to learn function approximators in the supervised case with gradient descent, similar methods can be applied to parameterized policies, albeit with a few major differences. I the problems that are relevant here, actions are typically “hard” decisions, and there does not exist a derivative of some loss (or reward) with respect to having taken the action (or not). As such, it is not trivial to obtain the gradient of the parameters that goes in the direction of maximizing the rewards. Fortunately, there are methods that compute approximations of these gradients. Using these methods to search among the space of policy functions is known as **policy search**.

3.4.1 REINFORCE

REINFORCE (Williams, 1992), also known as the likelihood ratio method (Glynn, 1987), is a policy search algorithm. It computes an unbiased estimate of the gradient, and uses it to perform policy search in the direction that minimizes loss.

The objective function of a parameterized policy π_θ for the cumulative return (the total reward received) of a trajectory τ of length T is described as:

$$J(\theta) = \mathbb{E}_\tau^{\pi_\theta} \left\{ \sum_{t=1}^{T} r(S_t, A_t) \middle| S_0 = s_0 \right\}$$

where s_0 is the initial state of the trajectory. Let $R(\tau)$ denote the return for trajectory τ. The gradient of the objective with respect to the parameters of the policy is:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_\tau^{\pi_\theta} \{R(\tau)\} \\ &= \nabla_\theta \int_\tau \mathbb{P}\{\tau|\theta\} R(\tau) d\tau \\ &= \int_\tau \nabla_\theta \left[\mathbb{P}\{\tau|\theta\} R(\tau)\right] d\tau \qquad (3.2) \end{aligned}$$

Note that the interchange in (3.2) is only valid under some assumptions (see Silver et al. (2014)).

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_\tau \nabla_\theta \left[\mathbb{P}\{\tau|\theta\} R(\tau)\right] d\tau \\ &= \int_\tau \left[R(\tau) \nabla_\theta \mathbb{P}\{\tau|\theta\} + \nabla_\theta R(\tau) \mathbb{P}\{\tau|\theta\}\right] d\tau \qquad (3.3) \\ &= \int_\tau \left[\frac{R(\tau)}{\mathbb{P}\{\tau|\theta\}} \nabla_\theta \mathbb{P}\{\tau|\theta\} + \nabla_\theta R(\tau)\right] \mathbb{P}\{\tau|\theta\} d\tau \\ &= \mathbb{E}_\tau^{\pi_\theta} \left\{R(\tau) \nabla_\theta \log \mathbb{P}\{\tau|\theta\} + \nabla_\theta R(\tau)\right\} \qquad (3.4) \end{aligned}$$

The product rule of derivatives is used in (3.3), and the derivative of a log in (3.4). Since $R(\tau)$ does not depend on θ directly, the gradient $\nabla_\theta R(\tau)$ is zero. We end up with this gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau^{\pi_\theta} \left\{R(\tau) \nabla_\theta \log \mathbb{P}\{\tau|\theta\}\right\} \qquad (3.5)$$

Without knowing the transition probabilities, we cannot compute the probability of our trajectories $\mathbb{P}\{\tau|\theta\}$, or their gradient. Fortunately, we are in a MDP setting, and we can make use of the Markov property of the trajectories to compute the gradient:

$$\begin{aligned}\nabla_\theta \log \mathbb{P}\{\tau|\theta\} &= \nabla_\theta \log \left[p(s_0) \prod_{t=1}^{T} \mathbb{P}\{s_{t+1}|s_t, a_t\} \pi_\theta(a_t|s_t) \right] \\ &= \nabla_\theta \log p(s_0) + \sum_{t=1}^{T} \nabla_\theta \log \mathbb{P}\{s_{t+1}|s_t, a_t\} + \nabla_\theta \log \pi_\theta(a_t|s_t) \qquad (3.6) \\ &= \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\end{aligned}$$

In (3.6), $p(s_0)$ does not depend on θ, so the gradient is zero. Similarly, $\mathbb{P}\{s_{t+1}|s_t, a_t\}$ does not depend on θ (not in a differentiable way at least), so the gradient is also zero. We end up with the gradient of the log policy, which is easy to compute.

In the case where the trajectories only have a single step, the summation disappears and the gradient is found by taking the log of the probability of the sampled action:

$$\nabla_\theta R(s) = \mathbb{E}\left\{R(s) \nabla_\theta \log \pi_\theta(a|s)\right\} \qquad (3.7)$$

3.4.1.1 Reducing variance in REINFORCE

Just as in stochastic gradient descent, it is only useful (and computationally tractable) to use stochastic samples of the data in order to approximate the full gradient in (3.7).

A problem can arise if the length of the trajectories is large or if the number of possible actions per-step is large. Since a particular trajectory or action sample has an intrinsic low probability (simply because there are many possibilities) its gradient's distance to the full expected gradient will be large. This gives rise to a high variance estimator.

There are many ways of reducing variance in policy gradient methods, some work by incurring bias (Sutton et al., 1999), while some manage to reduce the variance without adding bias (Greensmith et al., 2004; Weaver and Tao, 2001).

A simple way to reduce variance without adding bias in REINFORCE is by modifying the reward signal by subtracting a *baseline* b.

$$\nabla_\theta R(s) = \mathbb{E}\left\{(R(s) - b)\nabla_\theta \log \pi_\theta(a|s)\right\} \tag{3.8}$$

There are many ways to formulate the baseline, the simplest being to use the average reward. In this case an interesting interpretation of REINFORCE can be made, where if an action has a higher reward than the average, its probability will be increased, while if it is lower, its probability will be reduced.

4

Conditional Computation in Deep Networks through Policies

Conditional computation (Bengio et al., 2013; Davis and Arel, 2013) is concerned with reducing the amount of computation made by some models, it does so by actively selecting parts of it that are to be computed given the current input. This problem can be viewed as a sequential decision making problem that takes place inside inside a traditional machine learning algorithm.

Having presented artificial neural networks and reinforcement learning, this chapter explains how the two are used in conjunction in order to create efficient conditional computation models.

State-of-the-art models are geting larger and larger every year, and even with the massive increase in computation via distributed systems and massively parallel hardware, it is non-trivial to *deploy* such models such that they can be used in real-time, since most target hardware does not have access to the same kind of massive computation hardware (e.g. mobile phones). As such, there is a need for a computation scheme that speeds up such models. The proposed approach to conditional computation is a step in this direction.

4.1 Defining Conditional Computation

The problem of conditional computation can be expressed as follows: *how can a model choose to perform as few computations as possible, and still make correct predictions?* Let us start by giving a definition of computation.

In a function approximator, the prediction $y = f(x;\theta)$ is typically done using many steps of computation, computing intermediate values up to the final result. Sometimes these steps can be done in parallel, sometimes one step depends on a previous one.

Let us define a computation graph $\langle V, E\rangle$. We denote the set of intermediate values as $V = \{v_0, v_1, ..., v_n\} \cup \{x_0, ..., x_d\}$, and the set of dependencies as directed edges $E = \{(v_i, v_j) \mid \text{computing } v_j \text{ requires } v_i\}$. Conditional computation can act on the edges of this computation graph. Consider the function $a : \mathcal{X} \to \{0,1\}^{|E|}$. Given an input x, c outputs a binary vector, associating each edge (v_a, v_b) to either 0 or 1. On a 1, the normal computation occurs. On a 0, v_b is computed while ignoring the value of v_a and considering it to be either 0 or some other predefined value (either a constant, or a ancestor node in the graph). If for all edges (v_a, v_j), the associated $a(x)_{v_a,v_j}$ is 0, then v_a does not need to be computed.

This last phenomenon can be encoded in the function $a : \mathcal{X} \to \{0,1\}^{|V|}$. Given an input x, $a(x)_{v_i} = 0$ means that v_i does not need to be computed. Now consider dividing V into disjoint subsets ν_i. Instead of associating to each intermediate value a conditional computation flag $a(x)_{v_i}$, we can associate to each subset a single flag $a(x)_{\nu_i}$.

The success of conditional computation relies on the idea of sparse (and thus faster) computation. If the number of subsets is substantially smaller than the size of V, then the computation of $a(x)_\nu$ will only incur a slight overhead. Additionally, if the number of non-zero $a(x)_i$s is small, then most intermediate values will not need to be computed, *resulting in fewer computations being needed.*

In neural network models a very natural computation graph arises from the struc-

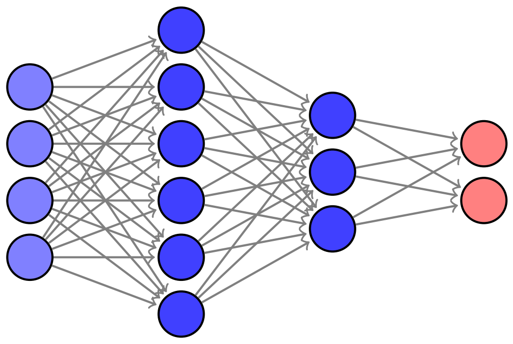


Figure 2.3: A multilayer perceptron with 2 hidden layers, one of 6 units, one of 3 units. The output layer has 2 units.

ture of the network. Consider Figure 2.3 (reproduced here), at layer i each node $h_j^{(i)}$ is computed as:

$$h_j^{(i)} = \sum_k W_{kj}^{(i)} h_k^{(i-1)}$$

There are at least two "computation-skipping" mechanisms that we can use in such a computation when the computation flag is 0: akin to dropout (Hinton et al., 2012), "drop" the unit $h_j^{(i)} = 0$; or, akin to residual networks (He et al., 2015a), "skip" the unit $h_j^{(i)} = h_j^{(i-1)}$.

In the same spirit, there are two natural grouping mechanisms which one can use. Since many units are often required to disentangle features, it is logical that within a layer some features will often activate together, as observed by Li et al. (2015). As such we can divide hidden layers into groups of hidden units, that are all dropped or activated altogether.

One can also group a whole layer into a computation subset, and conditionally skip layers. This especially makes sense in the computer vision setting, where applying a specific convolutional layer in a very deep network extracts a particular set of visual patterns which might not be present, in which case skipping the layer is justified.

4.2 Conditional Computation as a Policy

This work is an attempt to illustrate that conditional computation in a deep neural network model can naturally be viewed as sequential decision making problem. At each layer, given information on the current state of the computation (what has been computed so far), one has to decide which units to compute.

There is also a natural, delayed reward measure, given by the error at the output of the network; one can further impose rewards that encourage sparsity in the computation, or a fast computation time. This view makes this problem a natural fit for reinforcement learning algorithms.

In the previous section we defined a computation function $a : \mathcal{X} \to \{0,1\}^n$. This function can naturally be seen as a policy $\pi_\theta(a|x)$, where the state space is $\mathcal{X}$ and the action space is this computation flag vector space $\{0,1\}^n$.The sequential case also takes the same form, where at each layer, decisions about the next layer's computations are made depending on the state of the computation, thus creating a temporal delayed-reward setting with a policy that maps some intermediate computations $\mathcal{H}$ to a binary vector $\{0,1\}^k$.

Finally, in the context of *learning* a conditional computation policy, it is undesirable to have a deterministic policy. For two close deterministic policies, the expected gradient would change greatly between one and the other (since we are always taking the same action for some input), making learning unstable. On the other hand, for two close stochastic policies, the probabilities of actions will be close to one another, and so their *expected* gradients will be similar, making learning more stable. As such, we will learn and use stochastic Bernoulli policies to achieve conditional computation.

Learning Conditional Computation policies

Now that the problem of conditional computation is framed as a reinforcement learning problem with stochastic policies, this work shows it is possible to learn these models and their policies using the frameworks seen in sections 2.2.4 and 3.4.1. Note that we will refer to the principal neural network (e.g. the classifier or regressor) as *target*, or target model, and neural network outputting the policy as the *policy model*, or simply *policy*.

In this particular case, the negative reward signal of the policy received after taking actions can be set to be the loss of the target model $\mathcal{L}$, as for example in Section 2.1.4. As such we perform gradient descent on the policy parameters using the REINFORCE estimator in the following form:

$$\nabla_\theta \mathcal{L}(x) = \mathbb{E}_{x\sim X}\left\{(\mathcal{L}(x) - b)\nabla_\theta \log \pi_\theta(a|x)\right\}$$

Here $\pi_\theta(a|x)$ is the probability of the sampled as. We assume there is some parameterization of the policy model $f_\pi(x;\theta)$, possibly a neural network itself, that outputs a vector in $[0,1]^n$, and a is sampled as a vector of Bernoullis with probabilities $f_\pi(x;\theta) = [\sigma_1\ \sigma_2\ ...\ \sigma_n]^T$. Its log probability is simply:

$$\log \prod_i \left[a_i\sigma_i + (1-a_i)(1-\sigma_i)\right] = \sum_i \log\left[a_i\sigma_i + (1-a_i)(1-\sigma_i)\right]$$

Note that the learning of the target model must occur simultaneously to the learning of the policy: consider the scenario where blocks of hidden units are dropped (the same reasoning applies in other scenarios). If the target model is learned first, then the units that activate together must be grouped correctly[1], otherwise the computation policy will not be of any help; this creates a new, possibly harder, problem of grouping units correctly. On the other hand if the policy is learned first (with the target model at its initial random weights), then the policy has no way of effectively reducing the error, since the target model outputs mostly random and unstructured values (as it has yet to learn).

[1]As is suggested by litterature concerning dropout, units often coadapt to form some particular computational structure. Even with dropout training, this still occurs, only much less. As such, grouping units arbitrarily post-training has a high chance of perturbing these structures and negatively affecting accuracy.

4.3 Guiding Policy Search

As mentioned in the previous section:

- the aim of conditional computation is to perform few computations;
- it is easy to impose additional constraints to minimize computations.

Such constraints can take two forms. The first, most straightforward one, is to modify the reward signal to include some computation cost, either as a proxy (i.e. the number of activated nodes) or as a direct measure (i.e. elapsed time). Unfortunately, such a direct modification could add variance to an already high-variance estimator. The second possible modification is to use regularization. Regularization has the advantage of being differentiable, and as such smoother (it varies consistently with the gradient).

This work introduces three differentiable regularization terms that act on the Bernoulli probabilities $[\sigma_1 \ \sigma_2 \ ... \ \sigma_n]^T$ of the stochastic policies, rather than on the action samples. The first two terms, L_b and L_e force the policies to achieve some sparsity hyperparameter target ratio τ.

$$L_b = \sum_j^n \|\mathbb{E}_X\{\sigma_j\} - \tau\|_2^2$$
$$L_e = \mathbb{E}_X\{\|(\frac{1}{n}\sum_j^n \sigma_j) - \tau\|_2^2\}$$

The first term L_b penalizes each σ_j independently, pushing σ_j to be τ in expectation over the data. The second term L_e penalizes σ_js together, forcing that for a given example only some σ_js can be high while others are low.

The third term is optional, but speeds up learning, as it forces policies to be "varied" and thus to be more input dependent:

$$L_v = -\sum_j^n \mathrm{var}_i\{\sigma_{ij}\}$$

These regularization terms can be optimized via gradient descent, simultaneously to the training of the target model and the policy. The additional loss term can be expressed as:

$$\mathcal{L}_{reg} = \lambda_s(L_b + L_e) + \lambda_v(L_v)$$

where λ_s and λ_v are hyperparameters that dictate the desired strength of the regularization.

4.4 Structure of conditional computation policies in deep models

Conditional computation can be added to models through all kinds of parameterizations. There are a few criteria that must be met in order to ensure that applying conditional computation is useful:

- there should be a natural and hardware-friendly way to distribute computation;
- the number of computation subsets should be at least an order of magnitude smaller than the number of parameters;
- the overhead of computing the policy should be minimal;
- each computation subset should be "capable" of discriminating a subset of the input space.

We proposed and investigated the performance of two such models, in the context of training Deep Neural Networks.

4.4.1 Fully connected architectures

A common observation in fully connected neural architectures is that their activation is often very sparse: i.e., most of the neurons tend to be zero, especially when using activation functions such as rectifying linear units.

Another observation, now empirically verified (Li et al., 2015), is that neurons tend to detect the presence of multiple patterns *together*, and as such form natural relevant groups of computation (which we can assume are conditioned on input subspaces).

As such, a natural way of grouping computation in this architecture is, for a given layer, to divide neurons into groups, which we well refer to as blocks. For this division to be hardware friendly, we impose for all groups to have the same size.

With these groupings, a way of parameterizing the policy is to associate a Bernoulli unit to each group, and make this unit dependent on the activations of the previous layer. This is illustrated in Figure 4.1.

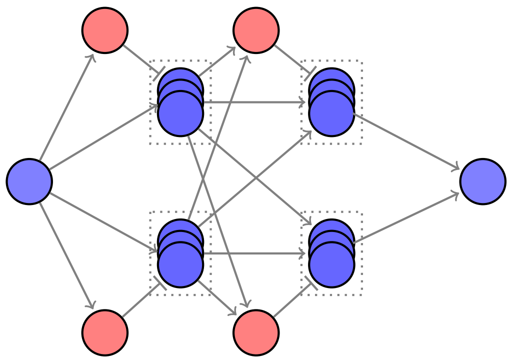


Figure 4.1: A fully-connected block conditional computation architecture. In blue, the target model, divided in blocks of units. In red, the computation policy, calculated from the previous layer's activation.

The policy units themselves are computed as normal sigmoid units, with a different set of weights for each layer (each predicting the policy for all the layer's blocks):

$$\sigma^{(i)} = s(W^{\pi_i}h + b^{\pi_i})$$

where s is the sigmoid function (see Section 2.1.5, here σ instead denotes the policy probabilities).

4.4.2 Convolutional architectures

In the convolutional setting, there are many ways in which we could turn off parts of the target model, some more expensive than others. For example we could choose which channels we want to use at each layer, but that would require the policy to be complex and expensive.

A simple alternative, inspired by Huang et al. (2016), is to skip layers in a residual manner. If layer i is skipped, then the input of layer $i+1$ is simply the activations of layer $i-1$ (which can itself be the result of a skip). In Huang et al. (2016) layers are randomly skipped during training, in a similar fashion as dropout (Hinton et al., 2012).

In this case, we would like to use conditional computation to choose which layers to skip as a function of the input. As such the policy only has to have a single unit per layer. This is illustrated in Figure 4.2.

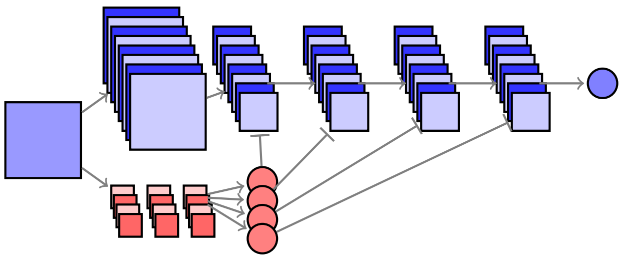


Figure 4.2: A convolutional layer-skipping architecture

4.5 Related Work

Ba and Frey (2013) proposed a learning algorithm called *standout* for computing an input-dependent dropout distribution at every node. As opposed to our layer-wise method, standout computes a one-shot dropout mask over the entire network, conditioned on the input to the network. Additionally, masks are unit-wise, while our approach is more general and can handle groups of units, for example it can use masks that span blocks of units. Bengio et al. (2013) introduced Stochastic Times Smooth neurons as gaters for conditional computation within a deep neural network. STS neurons are highly non-linear and non-differentiable functions learned using estimators of the gradient obtained through REINFORCE. They allow a sparse binary gater to be computed as a function of the input, thus reducing computations in the then sparse activation of hidden layers.

Stollenga et al. (2014) recently proposed to learn a sequential decision process over the filters of a convolutional neural network (CNN). As in our work, a direct policy search method was chosen to find the parameters of a control policy. Their problem formulation differs from ours mainly in the notion of decision "stage". In their model, an input is first fed through a network, the activations are computed during forward propagation then they are served to the next decision stage. The goal of the policy is to select relevant filters from the previous stage to improve the decision accuracy on the current example. They also use a gradient-free evolutionary algorithm, in contrast to our gradient-based method.

The Deep Sequential Neural Network (DSNN) model of Denoyer and Gallinari (2014) is possibly closest to our approach. The control process is carried over the layers of the network and uses the output of the previous layer to compute actions. The REINFORCE algorithm is used to train the policy with the reward/cost function being defined as the loss at the output in the base network. DSNN considers the general problem of choosing between different type of mappings (weights) in a composition of functions. However, they test their model on datasets in which different modes are prominent, making it easy for a policy to distinguish between them.

Another point of comparison for our work are the recent attention models (Mnih et al., 2014; Gregor et al., 2015; Xu et al., 2015). These models typically learn a policy, or a form of policy, that allows them to selectively attend to parts of their input *sequentially*, in a visual 2D environment. Both attention and conditional computation share a goal of reducing computation times and maintaining accuracy. While attention aims to perform dense computations on subsets of the inputs, our conditional computation approach aims to be more general, since the policy can target subsets of computation throughout the network architecture. As such, it can lead to more distributed computation at various levels of feature resolution. It should be possible to combine these two approaches, since one acts on the input space and the other acts on the representation space, although the required policy space to do this may well be more complex, and thus harder to train.

5 Empirical Evaluation

In this chapter we cover both implementation details and empirical results of experiments made to illustrate the capabilities of the conditional computation framework. For our experiments we made extensive use of the Theano library (Theano Development Team, 2016), and implement naive sparse computation algorithms.

5.1 Sparse Matrix Operations

In Section 4.4.1 we showed how to build a fully-connected block architecture, and in Section 4.3 we showed how to regularize such architectures so that they are sparse.

If the computation policies are sparse enough, then it becomes advantageous to explicitly *not* compute zero'ed units (which would otherwise be zeroed via a multiplicative mask).

We implemented such an algorithm on both CPU (in C) and GPU (in CUDA), as Theano Ops (building blocks of the library's computation graphs).

In the convolutional case, the model described in Section 4.4.2 has a very simple conditional structure. If a layer's flag is 0, it is simply not computed and its input is passed on to the next layer. As such there is no need for any specialized implementation.

5.1.1 Structure of sparse blocks

Consider two layers, the input layer $i-1$ and output layer i, for each of these layers the policy was sampled into a binary vector, which we will refer to as mask. In the minibatch setting, the many vectors are joined into mask matrices $M^{(i-1)}$ and $M^{(i)}$. As such the computation (implicitly taking into account bias and activation function) of the output of layer i is:

$$H^{(i)} = (H^{(i-1)} \times W^{(i)}) \odot M^{(i)} \tag{5.1}$$

where in fact $H^{(i-1)} = H^{(i-1)} \odot M^{(i-1)}$. The structure of the non-zero entries in this computation is illustrated in Figure 5.1.

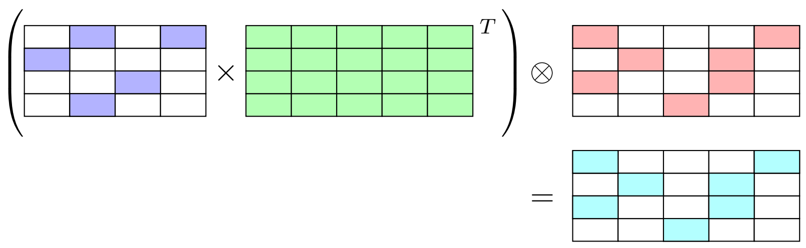


Figure 5.1: A sparse matrix-matrix product

5.1.2 CPU implementation

A very natural algorithm follows from this sparse structure.

For every non-zero entry in $M^{(i)}$, compute the dot product at that location, say x, y, only taking into account $W^{(i)}_{xk}$ and $H^{(i-1)}_{ky}$ where $M^{(i-1)}_{ky}$ is non-zero. Since each non-zero entry of the masks spans several units (because they are grouped in blocks), the full masks need not to be represented explicitly, and this computation happens somewhat locally densely (in terms of memory local to that block).

Implementing this algorithm in the C programming language, we can measure the impact of using the algorithm versus doing the full computation (which in Theano is done using BLAS's gemm, GEneral Matrix Multiplication operation).

Figure 5.2 illustrates the obtained speedup for a particular instance, 32 blocks of size 32, with a minibatch size of 32. We can observe that below 30% sparsity, using such a sparse dot product implementation is faster.

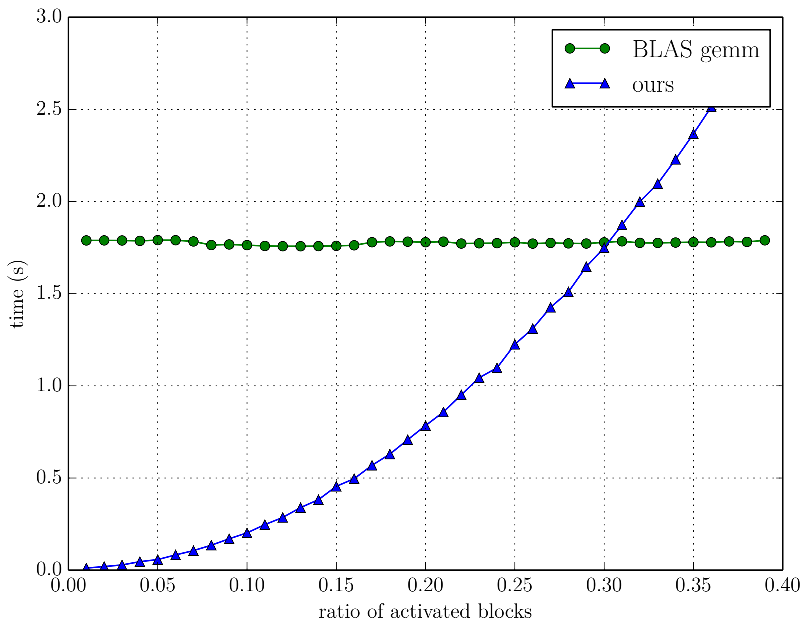


Figure 5.2: Timing of a sparse matrix-matrix product for different sparsity rates.

We also measure how this speedup evolves with respect to the size of blocks. This is illustrated in Figure 5.3. We observe that having too small blocks hinders performance. At the same time, it is not necessarily enviable, from the target model's point of view, to have very large blocks, thus a trade-off is necessary.

5.1.3 Anatomy of a GPGPU

General Purpose Graphical Processing Units, GPGPUs, are specialized hardware mainly used to generate rasterized images, but which can also be used to achieve significant computation gains.

GPGPUs are a very advantageous hardware to use when doing parallel computations, because they are inherently massively parallel. Such computations include matrix multiplication, and convolutions.

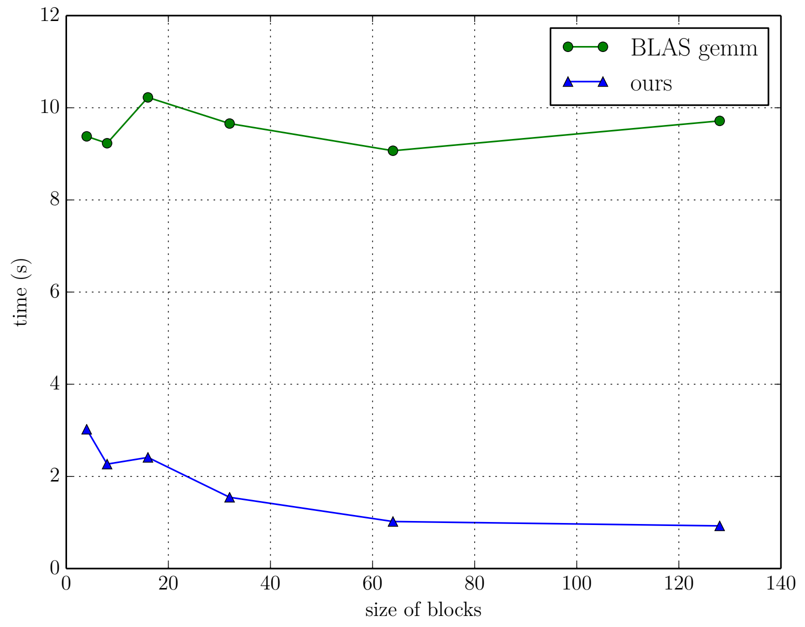


Figure 5.3: Timing of a sparse matrix-matrix product for different block sizes, averaged over different sparsity rates.

GPGPUs are made of hundreds, sometimes thousands of cores. They have a fundamental difference with typical CPU coreswhich allows many cores to function at the same time. They are grouped together in clusters, and within each cluster every core *has to be executing the same instruction*. GPGPU programs in which this is not the case will cause the execution to branch. If two cores in the same cluster branch off (e.g. as the result of an `if`), simultaneous execution is no longer possible: the true branch of the `if` will be executed, and then the false branch (which can cause massive slowdown of execution if it happens often).

5.1.4 GPU implementation

Simply rewriting the CPU implementation in CUDA (for NVIDIA GPGPUs) would not work, since as explained in the last section, conditioning execution of each block on whether its associated flag is 0 or 1 will catastrophically break parallelism.

As such the GPU implementation is done in 2 passes, where the first pass is determining which blocks are non-zero, and the second pass is actually doing the computations for these non-zero blocks only. As such the "kernel" (parallel subprogram which is launched for each core) is only launched for blocks where there is computation, so that no branching happens.

Figure 5.4 illustrates the obtained speedup for a particular instance on GPGPU, again 32 blocks of size 32, with a minibatch size of 32. We can observe that below 17% sparsity, using this implementation is faster.

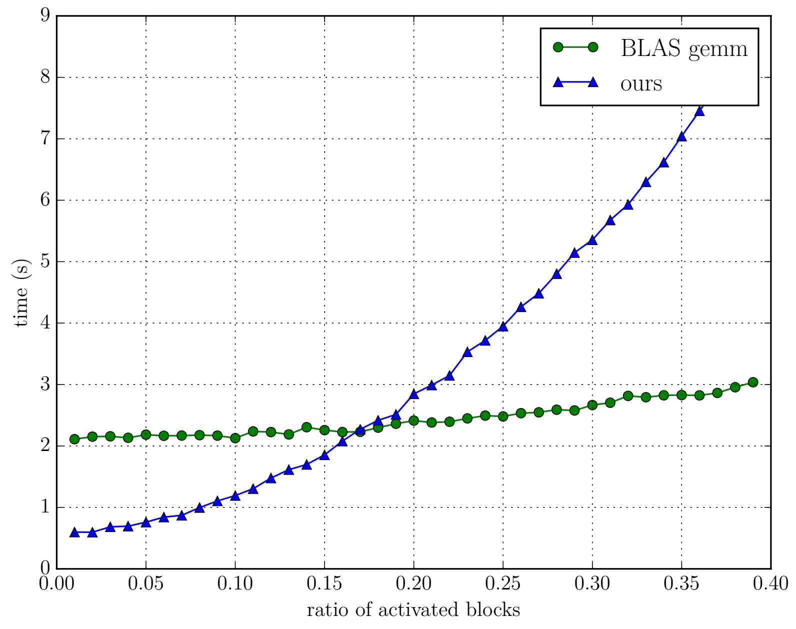


Figure 5.4: Timing of a sparse product for different sparsity rates on the GPU.

Comparing figures 5.2 and 5.4, we see that the GPU implementation is much faster than the CPU implementation, but less amenable to sparsity, as the GPGPU cuBLAS implementation of the matrix product is also much faster.

5.2 Image Experiments

We experiment on 3 datasets, MNIST (LeCun et al., 1998a), CIFAR-10 (Krizhevsky and Hinton, 2009), and SVHN (Netzer et al., 2011). We compare neural networks trained with a conditional computation mechanism to those without, and see whether they can achieve similar accuracy while reducing the time required to perform computations.

5.2.1 MNIST

MNIST (LeCun et al., 1998a) is a simple classification dataset consisting of 28×28 grayscale images of handwritten digits. The task is to indentify the class of a digit among 10 possible classes. It has long been regarded as a standard benchmark in the Deep Learning community, and is now commonly used as a sanity check when creating new architectures.

Figure 5.5: MNIST samples

For this task, we hand-pick sensible hyperparameters to conduct sanity checks, and we use a target model with a single hidden layer of 16 blocks of 16 units (256 units total), with a target sparsity rate of $\tau = 6.25\% = 1/16$, learning rates of 10^{-3} for the neural network and 5×10^{-5} for the policy, $\lambda_v = \lambda_s = 200$ and $\lambda_{L2} = 0.005$. Under these conditions, a test error of around 2.3% is achieved. A normal (without any conditional computation policy) neural network with the same number of hidden units achieves a test error of around 1.9%, while a normal neural network with a similar *amount* of computation (multiply-adds) being made (32 hidden units) achieves a test error of around 2.8%.

While not beating state-of-the-art, which is not our intent here, we obtain low error and similar results to a normal neural network. Since the computation performed in our model is sparse, one could hope that it achieves this performance with less computation time, yet we consistently observe that models that deal with MNIST are too small to allow our specialized sparse implementation (Section 5.1) to make a substantial difference.

5.2.1.1 Learned policies

What is interesting for MNIST is analyzing the policies of conditional computation that were learned for this model.

Figure 5.6 illustrates the activation of the policy units. Each unit, associated with a block of units in the target model, has a particular index on the x axis. On the y axis the policy probabilities, σ_is, are plotted as a transparent dot for each sample.

In Figure 5.6a, only examples of class 0 are plotted, showing that only some of the blocks are used by the policy. In Figure 5.6b, only examples of class 1 are plotted, again showing that only some of the blocks are used, but they are different blocks than for class 0. Figure 5.6c shows the plot for all classes, showing that across its input space, across classes, the policy has learned to balance which blocks are being used given an input, always achieving some fair amount of sparsity.

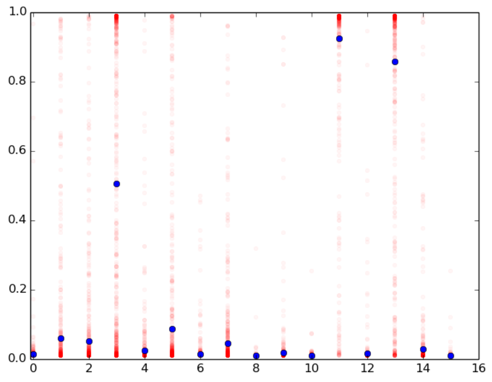
Note that regularization is required for such results, but that λ_v seems to mostly help to speed up (sometimes drastically) convergence.

These results show that this framework for conditional computation is capable of learning sensible policies that separate the input space and specialize computation to a limited number of computation subsets.

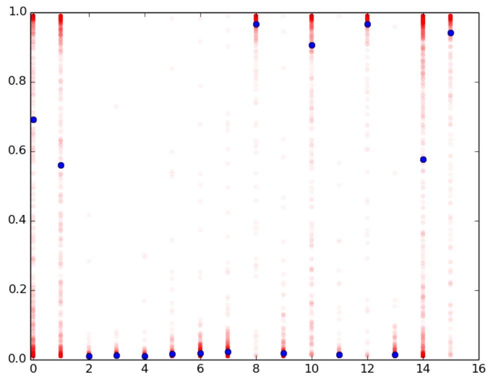
5.2.1.2 Is learning the policy helpful?

As mentioned in Section 2.3.1, randomly dropping units in a neural network has proved to be an effective method for learning models. Considering this, we compare using random uniform policies, similarly to the dropout scenario (Hinton et al., 2012), to learning conditional computation policies.

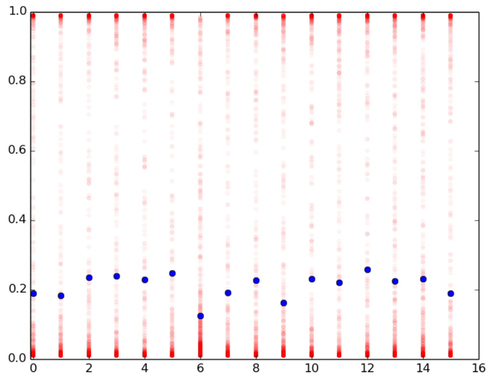
The results can be see in Figure 5.7, where measured sparsity rates are plotted against validation error. As sparsity is increased, the performance of a uniform policy



(a) Policy activation for samples of class 0



(b) Policy activation for samples of class 1



(c) Policy activation for samples of all classes

Figure 5.6: MNIST, (a,b,c), probability distribution of the policy, each example's probability (y axis) of activating each unit (x axis) is plotted as a transparent red dot. Redder regions represent more examples falling in the probability region. Plot (a) is for class '0', (b) for class '1', (c) for all classes.

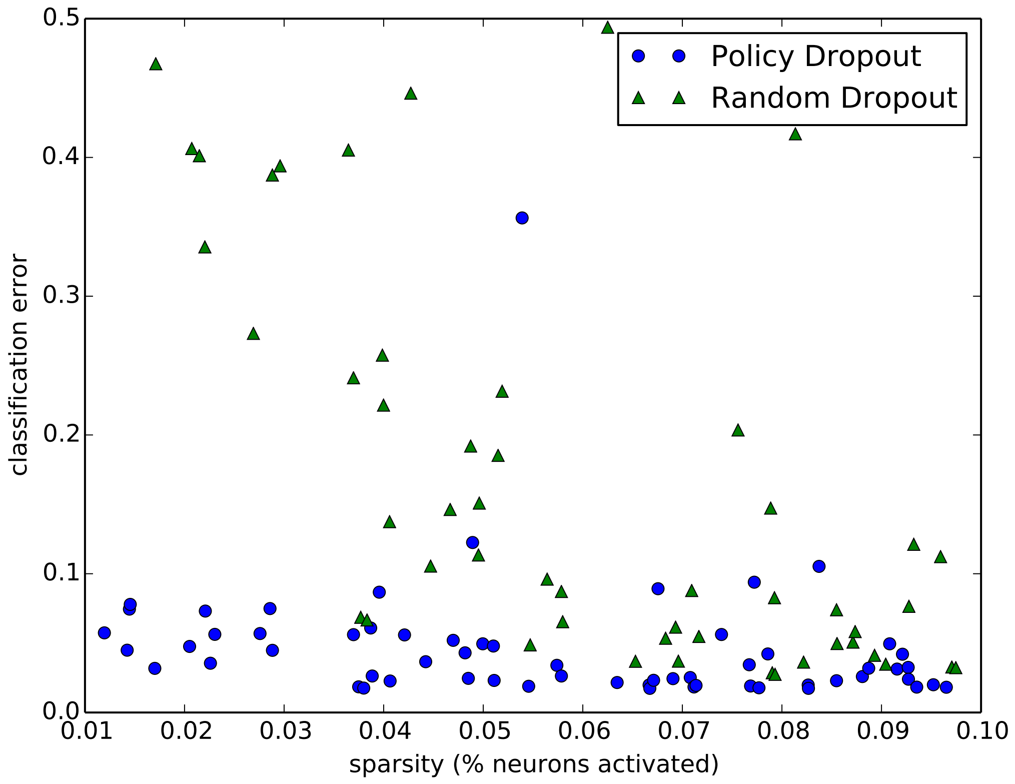


Figure 5.7: Varying sparsity rates, comparing learned policies to uniform policies

on MNIST gets worse and worse, while the performance of a learned computation policy stays stable, even in the most extreme cases.

5.2.2 SVHN

The Street View House Numbers (SVHN, Netzer et al. (2011)) is a classification dataset consisting of 32×32 RGB images of digits, extracted from images of street building addresses. It is similar to the MNIST dataset, but much harder due to the greater variation in digit orientation and additional texture noise, as well as the confounding nearby digits.

To test the robustness of our framework, we performed a hyperparameter exploration over fully-connected architectures, which is summarized by Figure 5.9.

We see that using conditional computation, with a specialized sparse operation,

Figure 5.8: Street View House Numbers dataset samples

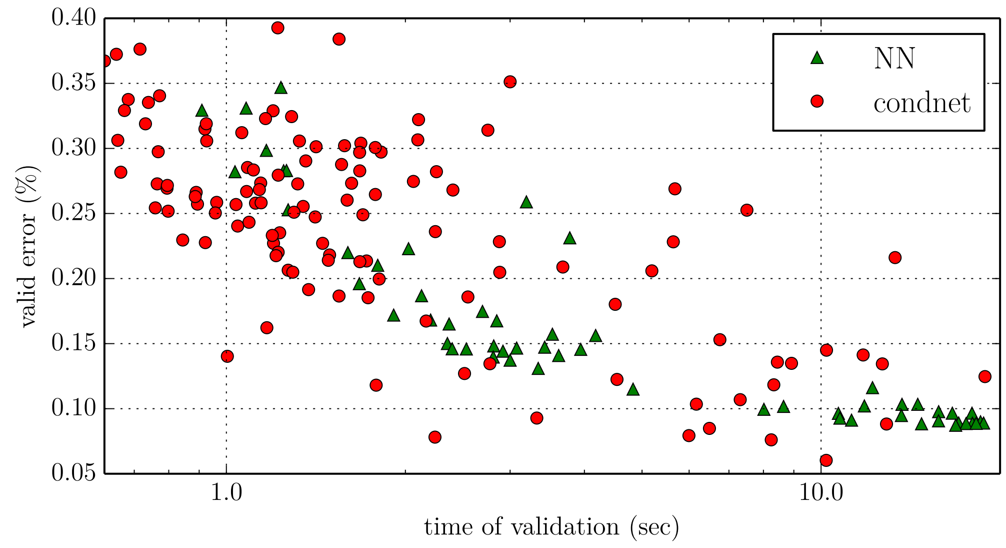


Figure 5.9: SVHN, each point is an experiment. The x axis is the time required to do a full pass over the valid dataset (log scale, lower is better). NN is a normal neural network, condnet is the present approach.

achieves better results than similar fully-connected architectures, at a much lower computational cost. Specific results are selected and shown in Figure 5.10.

It seems that by varying hyperparameters, it is possible to manually control the trade-off between high-accuracy and low computation time. In the next section on CIFAR-10, we design experiments that specifically highlight this fact.

5.2.3 CIFAR-10

The CIFAR-10 dataset is a classification dataset of 32×32 natural RGB images of 10 different classes of objects: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

It is a much harder dataset than SVHN, and it is hard to get less than 50% test

model	error	τ	#blocks	k	test time
condnet	.183	1/11	13,8	16	1.5s(1.4×)
condnet	.139	.04,1/7	27,7	16	2.8s (1.6×)
condnet	**.073**	1/22	25,22	32	10.2s(1.4×)
NN	.116	-	288,928	1	4.8s
NN	.100	-	800,736	1	10.7s
NN	.091	-	1280,1056	1	16.8s

Figure 5.10: SVHN results. condnet: the present approach, NN: Neural Network without the conditional activations. k is the block size. “test time” is the time required to do a full pass over the test dataset; in parenthesis is the speedup factor compared to run-time without the specialized implementation.

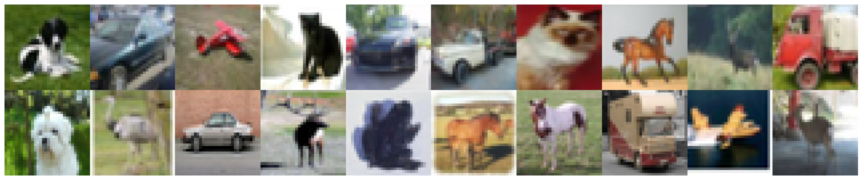


Figure 5.11: CIFAR-10 dataset samples

error without using convolutional models. As such we test both fully-connected and convolutional architectures on this dataset.

5.2.3.1 The speed-performance trade-off

Simply by controlling how much sparsity is enforced through the hyperparameter λ_s (see Section 4.3), it is possible to control the trade-off between having a fast model, and having a low-error model.

This can be seen in Figure 5.12, where we retrained the same fully-connected condnet architecture with several different values of λ_s. We plot both the resulting run-time of each model, as well as its validation error, and effectively observe the strong trade-off control effect of λ_s.

5.2.3.2 Regularizing early policies

By controlling the hyperparameter λ_v (see Section 4.3), it is possible to encourage policies not to be uniform. This can be beneficial early in training, as it pushes

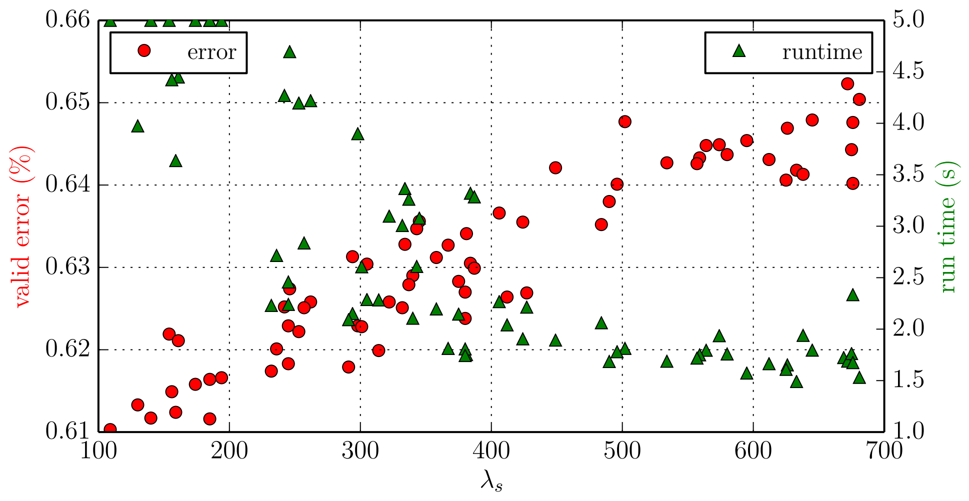


Figure 5.12: The effect of λ_s when training fully-connected conditional computation architectures

the policies to somehow separate the input space and force the model's computation subsets to specialize.

This can be seen in Figure 5.13. As λ_v gets bigger, the error goes down more rapidly, although it is to be noted that in convergence all the models reach the same error (some faster than others).

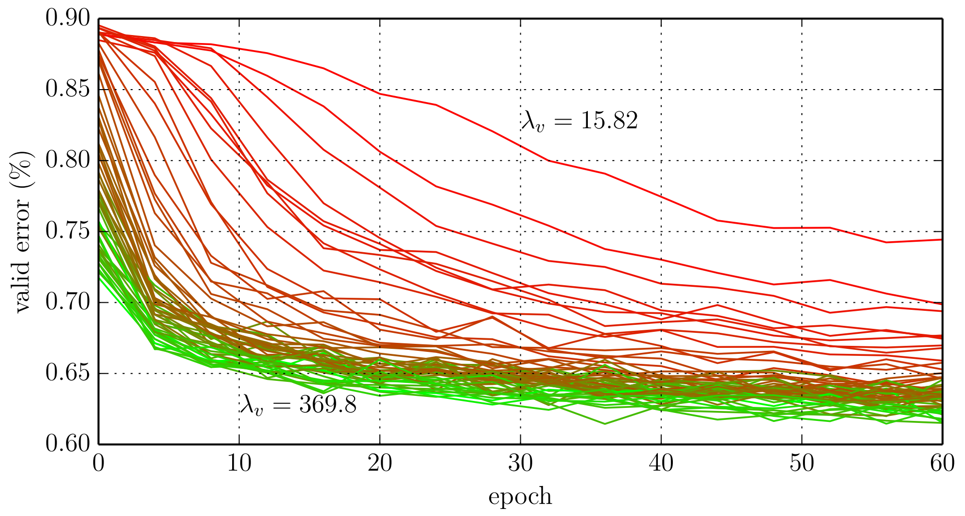


Figure 5.13: The effect of λ_v when training fully-connected conditional computation architectures

model	test error	# of layers	τ	test time
conv-condnet	**.157**	12	0.5	1.03s
conv-condnet	.167	12	0.3	0.84s
conv-condnet	.176	12	0.2	0.66s
conv-condnet	.173	6	0.5	**0.58s**
conv-NN	.159	12	-	1.07s

Figure 5.14: CIFAR-10 convolutional model results. conv-condnet is our approach, conv-NN is a convolutional network with no layers dropped out. Time measured on a NVIDIA Titan X GPU.

5.2.3.3 Training convolutional architectures

We also trained convolutional architectures as shown in Section 4.4.2. In the convnet case, it is easy architecturally to obtain speedups, but it seems much harder to train policies that correctly separate the input space.

Instead, regardless of complexity of depth, most of the learned policies seemed to divide computation in two, sometimes three groups, and assigned layers to each of those groups, making the policy a much weaker in terms of helping to minimize loss. We show some results in Figure 5.14.

5.3 Empirical Observations on Learning Policies

5.3.1 Time, computation and energy consumption

Throughout experiments, wall time of computing the forward pass over the entire validation set in minibatches is the primary measure of how well sparse computations are doing. Time is not necessarily the best measure, but is for now the one that best depicts reality in terms of deployment of Machine Learning models.

Counting the theoretical number of operations is the simplest comparison measure. In fact, it is so simple that it is not representative of modern hardware, and how it operates.

In the fully connected case, for two successive sparse layers of n units with a sparsity rate of τ, the complexity of the forward pass is in $O(n^3\tau^2)$.

In the layer-skipping case, it is simply linear in τ, the less layers are active, the less computation happens.

5.3.2 Streaming data in the fully-connected case

A use case where fully-connected conditional computation as presented in this work (and sparse computation implementations) could provide a massive advantage is in the case of streaming data; when each example is passed through the model one at a time, instead of in minibatches of many examples. This is a typical setting in deployment of machine learning models, for example on a mobile phone, where only one example at a time is to be treated.

In the streaming case, the advantages provided by dense computations in neural networks is lost since most matrix-matrix operations become matrix-vector operations, which cannot obtain the same gains.

Repeating measurements for figures 5.2 and 5.9 gives a very different impression in the case of streaming data. Figure 5.15 shows the same measurement as 5.2– Figure 5.16 shows the same as 5.9 – with a minibatch of size 1, i.e. the streaming case.

We see in the first case that the threshold at which sparse computations are advantageous is now much higher, more than 70%, compared to 30% in the minibatch case. In the second case, we see that almost all models beat their dense fully-connected counterparts, in terms of time against accuracy. Some well performing models are about 10 times faster than fully-connected models with the same accuracy.

5.3.3 Measuring power consumption

Power consumption is a different metric than time, achieving a computation slowly but without as much power as a faster alternative can prove beneficial to any battery-

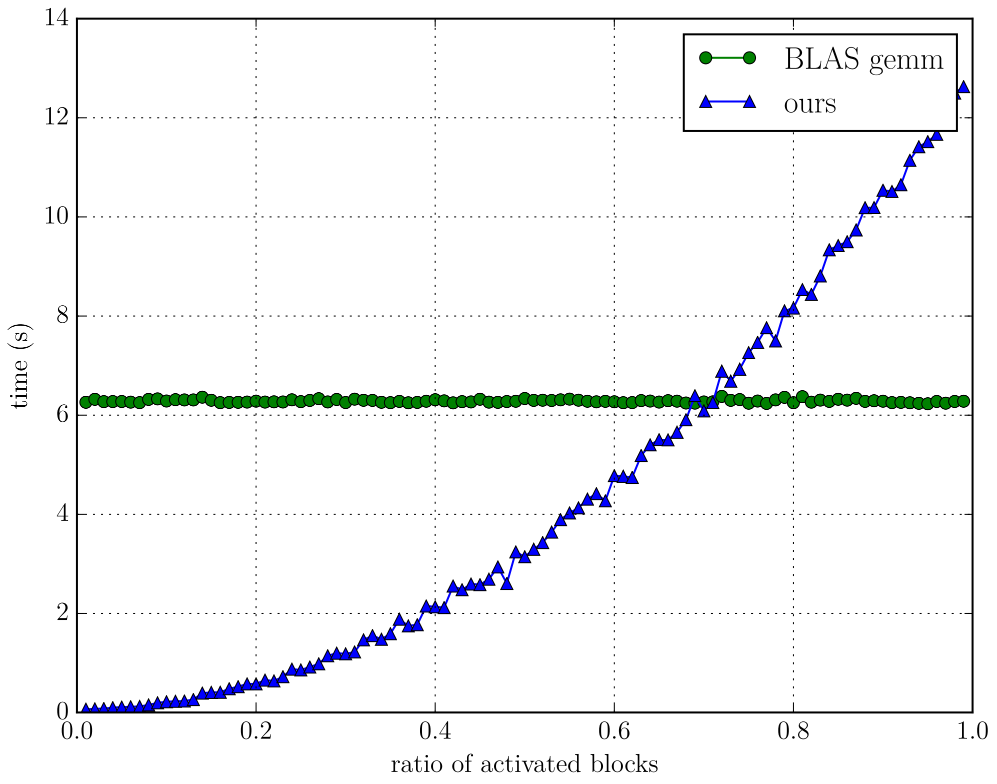


Figure 5.15: Timing of a sparse matrix-matrix product for different sparsity rates and a minibatch size of 1.

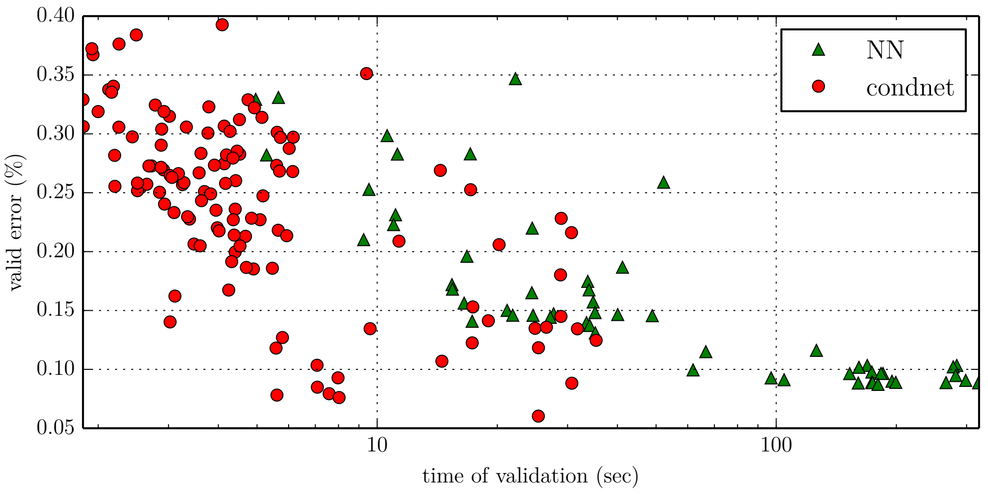


Figure 5.16: SVHN, each point is an experiment. The x axis is the time required to do a full pass over the valid dataset in a streaming setting (log scale, lower is better).

connected computing hardware.

We measured power consumption on a portable x86-64 computer, specifically on a i5-2467M CPU chip, and found that it unfortunately correlates linearly with time. A more interesting hardware to run such an experiment would be ARM CPUs, or even in the extreme case FPGAs (although the bottleneck in this case is memory transfer rather than computation).

5.4 Discussion

Overall our results show that using Reinforcement Learning for conditional computation is a sensible choice. Given the right parameterizations, it is possible to learn policies that separate the input space and specialize subsets of the model parameters. Given the right sparsity regularizations, it is possible to run these models much faster than their dense counterparts.

New hyperparameters add complexity to the algorithm, but each has a clear interpretation, and their effect can easily be monitored by visualizing policies during training (such as in Figure 5.6). Although sensible to these hyperparameters, the optimization remains relatively robust and it is possible, with little additional effort, to get as good (or sometimes better) results than with plain deep neural networks.

6

Conclusion and future work

Sparse computations seem like an inherently hard problem with modern architectures, which perform so well for dense computations. Nonetheless, leveraging Reinforcement Learning methods to reason about the structure of computations allow us to take advantage of the structure of Neural Networks, retaining strong accuracy and still managing to decrease computation time by using sparse algorithms.

Despite the optimization problems that learning such models poses, using strong regularization terms encouraging sparsity and varied activations enables them to perform just as well as their dense counterparts. The hyperparameters for such regularization can be tuned easily, and nicely correlate with the trade-off between precision and computation time. As such, it is possible to manually control this trade-off by varying these hyperparameters.

Using specialized algorithms that perform sparse computations based on the conditional policies, it is possible to get large speedups both on CPU and GPGPU architectures. Depending on the size of the data, the parameters, the minibatch, and the sparsity rates, sparse methods can slow down computations on one end, while on the other end, large speedups of more than 10 times can be obtained, despited using hardware unfavorable to sparse memory accesses.

There remain many challenges and possible improvements to the presented framework. It is still unclear whether the Reinforcement Learning methods that are used here are optimal. Even though they perform better than a random baseline, alternative methods taking into account the temporality of the sequence of computations that are performed in Neural Networks could perform better. Optimization also remains an issue, with training times being two to three times longer than normal networks. Similarly, the combination of the current Reinforcement Learning technique, REINFORCE, and the block structure of computation that is designed in Neural Networks might not be the most amenable to partial computation. Structures such as trees or recurrent models might offer different but more powerful conditional computation architecures, although at an additional optimization cost. It is also possible that entirely novel architectures of computation could be even more amenable to sparse computations.

There are also many aspects of this framework that need to be understood. There is an interplay during learning between what policy is currently used and what direction the main model parameters tend to. It should be possible to caracterize this interaction, understand whether it is beneficial to the resulting model, or whether it could be improved or accomplished differently.

It would also be interesting to explore how different such models are from normal models, how robust they are to changes in policy, or whether they learn different features. With high sparsity, many less parameters are changed at each learning iteration, and this might affect neural network problems such as the vanishing gradient, possibly amplifying it; it may as well be interesting to see the effect of momentum methods on such optimization, as these methods could compensate for the mostly sparse gradient signals.

Overall, there are good empirical reasons to believe that using Reinforcement Learning to take hard decisions in Neural Networks is a sensible choice. The results presented in this work support that such a choice is a sensible framework for Conditional Computation.

Bibliography

J. Ba and B. Frey. Adaptive dropout for training deep neural networks. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3084–3092. Curran Associates, Inc., 2013.

P. Baldi and K. Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58, 1989.

D. H. Ballard. Modular learning in neural networks. In *AAAI*, pages 279–284, 1987.

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

R. Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

L. Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

A. Davis and I. Arel. Low-rank approximations for conditional feedforward computation in deep neural networks. *arXiv preprint arXiv:1312.4461*, 2013.

L. Denoyer and P. Gallinari. Deep sequential neural network. *CoRR*, abs/1410.0510, 2014.

C. Dismuke and R. Lindrooth. Ordinary least squares. *Methods and Designs for Outcomes Research*, 93:93–104, 2006.

V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4): 193–202, 1980.

Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. *arXiv preprint arXiv:1506.02142*, 2015.

J. C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, 41(2):148–177, 1979. ISSN 00359246.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

P. W. Glynn. Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM, 1987.

I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. Book in preparation for MIT Press, 2016. URL `http://www.deeplearningbook.org`.

M. Gori and A. Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, 1992.

E. Greensmith, P. L. Bartlett, and J. Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004.

K. Gregor, I. Danihelka, A. Graves, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015a.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015b.

D. O. Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 1949.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen*, 1991.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/0893-6080(91)90009-T.

G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Weinberger. Deep networks with stochastic depth. *arXiv preprint arXiv:1603.09382*, 2016.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images, 2009.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998a.

Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. 1998b.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

G. W. Leibniz. Various writings, circa 1674.

S. Levine and P. Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

Y. Li, J. Yosinski, J. Clune, H. Lipson, and J. Hopcroft. Convergent learning: Do different neural networks learn the same representations? *arXiv preprint arXiv:1511.07543*, 2015.

L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, N. Heess, A. Graves, and k. kavukcuoglu. Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2204–2212. Curran Associates, Inc., 2014.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

K. P. Murphy. *Machine Learning: A Probabilistic Perspective.* The MIT Press, 2012. ISBN 0262018020, 9780262018029.

Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5. Granada, Spain, 2011.

I. Newton. *The Method of Fluxions and Infinite Series: With Its Application to the Geometry of Curve-lines.* Unknown, 1671.

A. Y. Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko. Semi-supervised learning with ladder networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3532–3540. Curran Associates, Inc., 2015.

H. Robbins. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, pages 169–177. Springer, 1952.

F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

T. Schaul, I. Antonoglou, and D. Silver. Unit tests for stochastic optimization. *arXiv preprint arXiv:1312.6055*, 2013.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 387–395, 2014.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

M. F. Stollenga, J. Masci, F. Gomez, and J. Schmidhuber. Deep networks with internal selective attention through feedback connections. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3545–3553. Curran Associates, Inc., 2014.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0262193981.

R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

C. Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

G. Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3): 58–68, Mar. 1995. ISSN 0001-0782. doi: 10.1145/203330.203343.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:2, 2012.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

S. Wager, S. Wang, and P. S. Liang. Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems*, pages 351–359, 2013.

C. J. C. H. Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge England, 1989.

L. Weaver and N. Tao. The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 538–545. Morgan Kaufmann Publishers Inc., 2001.

P. Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences.* PhD thesis, Harvard University, 1974.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992. ISSN 0885-6125. doi: 10.1007/BF00992696.

K. Xu, J. Ba, R. Kiros, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.

Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007. ISSN 1432-0940.

W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 362, 2015.

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.