Project Documentation

AriadneDL

Deep Learning for Ariadne

Mirco De Marchi

Contents

Chapter 1

Introduction

Ariadne is a tool for reachability analysis and model checking of hybrid systems. Additionally, Ariadne is a framework for rigorous computation featuring arithmetic, linear algebra, calculus, geometry, algebraic and differential equations, and optimization solvers.

AriadneDL wants to be a library that extends Ariadne functionalities in order to provide online estimations of the running system to improve its execution through Machine Learning and Deep Learning techniques. In particular, AriadneDL focuses on 2 main use case:

- Task execution time estimation;
- Adaptation of the dynamic part of a hybrid system model;

The implementation has to be done in C++ and it can be based on a library for Machine Learning and Deep Learning, with strict compatibility for the major operating systems. As alternative, the library can be developed from zero, in order to build an ad-hoc implementation. The library has to be compatible with MacOS, Ubuntu, Debian and Windows, in particular the library has to be provided by the following package manager: Homebrew, Aptitude and vcpkg. The libraries discussed will be the following:

- Tensorflow: github.com/tensorflow, /www.tensorflow.org;
- Caffe: github.com/caffe, caffe.org;
- Cognitive Toolkit (CNTK): github.com/CNTK, docs.microsoft.com/cognitive-toolkit;
- dynet: github.com/dynet, dynet.io;
- shogun: github.com/shogun, www.shogun.ml;

- FANN: github.com/fann, leenissen.dk/fann;
- Shark library: github.com/Shark, www.shark-ml.org;
- OpenNN: github.com/OpenNN, www.opennn.net;
- mlpack: github.com/mlpack, www.mlpack.org;
- Boost: github.com/boost, www.boost.org;
- Eigen3: gitlab.com/eigen, eigen.tuxfamily.org;
- Armadillo: gitlab.com/armadillo, arma.sourceforge.net

The following list provides some useful links related to the implementation and other resources:

- Ariadne main repository: github.com/ariadne-cps/ariadne;
- AriadneDL extension repository: github.com/mircodemarchi/AriadneDL;

Chapter 2

Requirements

2.1 Task execution time estimation

The first Ariadne integration is the **estimation of task execution time**. Ariadne can execute in parallel multiple tasks, that are represented as numeric integrations of a set that evolves over time. The tasks amount executed during the system evolution before the global integration finishes, is quite a large number. The objective is to find the best schedule of the task with different configurations, in order to obtain the best result in terms of execution time. This implementation can be useful in Ariadne, because different tasks could have really different execution time and the predicted execution time of a specified task configuration can be used to improve the threads scheduling.

Let assume to take in input a vector of bounded integer values, that represents the task parameters configuration for the numeric integration, the output has to be the estimated execution time of input task in microseconds. The task takes another input, in addition to the numeric configuration, that is the initial state of the set. The set is represented as a Taylor expansion with dozen of terms related with the continuous variables of the set space, that is more difficult to represent in the training model of a neural network. The set state has to be taken in consideration during the evolution of system tasks, because the task execution time tends to vary during the temporal evolution of the set, according to its complexity.

The deep learning model can be initially trained with a sequence of task parameters configurations labelled with their execution time. Then the estimator model has to take in input a task configuration and produce the predicted execution time. Even if the system has to perform multiple task in concurrence, the estimator model is interested in each single task execution. For this reason the most suitable solution for this integration would be a simple Feedforward Neural Network (FNN), without too many layers, in order to avoid an excessive storage of training data.

The prediction model can be improved over time, because when tasks finish their execution, Ariadne keeps track of real execution time. Furthermore, the deep learning model has to work well for the last evolution time interval rather than for all evolution, therefore it has to loose memory of oldest executions during online learning and overfitting has to be avoided. To implement this, the estimator has to provide an online learning, in order to give more importance to the latest execution time results obtained.

2.2 Parametric dynamic system adaptation

The second Ariadne integration is the **adaptation of a parametric function that represents the dynamic part of a hybrid system model**. Given a vector of samples, defined on continuous variables, that represent the states of a real system, and an associated hybrid model, we want to get the best accuracy of the hybrid model compared to the real system. Since time evolution is faster than real system evolution, we can use the hybrid model to predict in advance the points that the real system will reach, in order to catch dangerous situations. However, if the hybrid model is not so accurate, the performed prediction will be inaccurate.

The accuracy of this hybrid model can be improved through some configurable parameters that are given to the functions associated with the model dynamic at each location. The parameters can be defined as point values or, more generally, intervals, and allow to include the samples evolution. In fact, the hybrid model evolution can be represented as a flow channel that is the time evolution of a non-punctual set. The narrower the flow channel, the more accurate the model prediction and the greater the likelihood of including samples of real evolution in the flow channel defined by the sequence of chosen parameters.

The hybrid model evolution is periodically repeated starting from the last sample received by the real system. Therefore, the hybrid model can learn how to improve the sequence of chosen parameter through its errors. The proposed solution is to integrate a deep learning model inside the hybrid model that periodically takes as input the real system state, defined as a sequence of continuous variable samples, and it has to give as output the parameters values for the parametric functions of each dynamic model location that allow the hybrid model to best return a set of points that includes the future real system samples. The output has to be given as an interval, that is a minimum-maximum pair.

The deep learning model can be trained with a sequence of samples produced by the real system and a related sequence of simulated values produced by the hybrid model of the real system, with a defined parametric function for each hybrid model locations. In addition, the estimator has to be able to improve periodically the accuracy during system execution, for this reason the model has to perform an online learning. Ideally, in a real system that perform over a loop, the learning phase should be executed along all the samples collected in the loop. However, since the loop period is not easily findable, the number of samples on which perform the learning phase is not defined. Finally, this solution needs to define a quality metric of the estimator.

Furthermore, the prediction could be improved with an inference performed along a sequence of samples, because a future state can be found more accurately if the learning model had some of the past sequence of states available. For example a past sequence of states might suggest a direction of the states evolution, if the states evolution is located around a limited area or if it spreads over a large area. For this reason the most suitable solution would be a Recurrent Neural Network (RNN), because it suits really well with temporal sequences. However, the RNN learning model works very well if the real system future state depends on a sequence of previous states, but if the evolution states of a real system depends only on the current state, the usefulness of RNN decreases.

Since there is not a strict necessity to avoid overfitting and the complexity of this use case, compared to the first one, is higher, a good solution could be to implement a Feedforward Neural Network (FNN) with a larger amount of layers, in order to manage better a difficult problem. Therefore, if the RNN does not provide good results then you could opt for the FNN solution.

Chapter 3

C++ Libraries

3.1 Deep Learning libraries

The most famous deep learning libraries are: TensorFlow, Caffe and Cognitive Toolkit. These libraries are complete, really optimized, with every tool that you need and easy to use because based on a simple and abstract interface. However, they are not easy install, and not available on all operating system and package manager. In particular, TensorFlow is a Google library, really used, mostly in Python, for mobile and IoT applications. The following are the main features:

- Programming languages support: Python, C++, Javascript, Java, Go, Swift;
- CUDA support;
- Installation: pip, docker or homebrew;

Caffe is an efficient, speed and modular library with also a command line interface. The following are Caffe's main features:

- Programming languages support: C++, Python, Matlab;
- CUDA support;
- Installation: docker, apt, homebrew, and, for Windows, you need to build the project from source in the windows branch.
- Math libraries dependencies: BLAS and BOOST;

Cognitive Toolkit is a library developed by Microsoft that allows the user to easily realize and combine popular model types. The following are CNTK's main features:

- Programming languages support: C++, Python, C# or its own model description language (BrainScript);
- CUDA support on multiple GPUs and servers;
- Installation: docker, pip, but generally it is installed from source through an installation script. It is not supported for MacOS.

After that, there are some libraries, not very well known, but quite optimized and specific that could be interesting: DyNet, Shogun, FANN, Shark Library, OpenNN and mlpack. DyNet is a C++ library that works well with networks that have dynamic structures that change for every training instance. These kinds of networks are particularly important in natural language processing tasks. The following are DyNet's main features:

- Programming languages support: C++, Python;
- CUDA support;
- Installation: pip, homebrew.
- Math libraries dependencies: Eigen;

Shogun is a library not really specialized in deep learning models, but it offers mostly a wide range of efficient and unified machine learning methods. The following are Shogun's main features:

- Programming languages support: C++, Python, Octave, R, Java/Scala, Lua, C#, Ruby;
- No CUDA support;
- Installation: in the official installation website you can find the instructions for apt, homebrew, docker and pip package managers. Shogun natively compiles under Windows using MSVC.

FANN is a easy to use, versatile, well documented, and fast library, which implements multilayer artificial neural networks in C with support for both fully connected and sparsely connected networks. The following are FANN's main features:

- Programming languages support: C++, Python, Octave, R, Java/Scala, Lua, C#, Ruby, Matlab, Perl, PHP, Javascript and others;
- No CUDA support;

- Installation: vcpkg install support in Windows, but only from source for Linux systems. See official github repository for instructions.

Shark Library is a fast, modular, general open-source C++ machine learning library, with support for Feedforward Neural Network and Autoencoders. This library is not so big and it can be useful to use as a source from start to build a new custom ad-hoc library for our use cases. The following are Shark's main features:

- Programming languages support: C++;
- No CUDA support (only experimental OpenCL support on earlier releases through BLAS library);
- Installation: works on Windows, MacOS X, and Linux but only from source.
- Math libraries dependencies: Boost and BLAS;

OpenNN is a high performance library in terms of execution speed and memory allocation, with neural networks and machine learning algorithms. As in the case of Shark Library, also OpenNN library is easy to read from source and it can be used as starting point for developing a new ad-hoc library. The following are OpenNN's main features:

- Programming languages support: C++ and Python;
- support CPU parallelization by means of OpenMP and GPU acceleration with CUDA;
- Installation: only from source.
- Math libraries dependencies: Eigen;

Mlpack is an intuitive, fast, and flexible C++ machine learning library. This is the only library that provides more compatibilities for operating systems and package managers. The following are Mlpack's main features:

- Programming languages support: C++, Python, Julia, Go, R and provides a command line interface;
- No native CUDA support, but indirectly inherits CUDA support through Armadillo library dependency;

- Installation: in the official get stared guide there are all the informations to install mlpack on MacOS and Linux (Debian), with support for homebrew, Pkg.jl, apt, pip, docker or from source. Install from source or through vcpkg for Windows.
- Math libraries dependencies: BLAS, Armadillo and Boost;

Finally, a good solution could be the one to develop a custom ad-hoc library. A custom ad-hoc library would result lighter and more efficient, because specific for the task to solve. On the other hand, a third-party library would result in faster, easier developing and a wide choice availability of machine learning algorithms that can be used also for future implementations. The following is a list of useful reference from which to start developing a custom ad-hoc C++ library:

- github.com/3ammor/SimpleNeuralNet
- github.com/huangzehao/SimpleNeuralNetwork
- github.com/jeremyong/cpp_nn_in_a_weekend
- github.com/Whiax/NeuralNetworkCpp

3.2 Math libraries

A custom ad-hoc implementation could be developed with a library that supports mathematical operations along huge arrays and matrices. The following are libraries that can be used in C++ for machine learning and deep learning applications: Boost, Eigen3 and Armadillo.

Boost is a set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. Boost library is not built to support CUDA internally but from CUDA 2.2 you can use Boost functionalities inside CUDA kernels. Boost supports the main installation package managers: homebrew, apt, vcpkg.

Eigen3 is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Similarly to Boost library, Eigen3 doesn't support CUDA internally but the library functionalities can be partially used in CUDA kernels. Eigen3 supports the main installation package managers: homebrew, apt, but it doesn't support vcpkg.

Armadillo is a high quality linear algebra library for the C++ language, aiming towards a good balance between speed and ease of use. Armadillo is a library of higher abstraction level than Boost or Eigen3 libraries, in fact it uses BLAS library as dependency. In order to obtain GPU speed up on large matrix multiplications, you can link Armadillo with NVBLAS, which is a GPU-accelerated implementation in CUDA of BLAS. Armadillo supports the main installation package managers: homebrew, apt, vcpkg.

Chapter 4

Implementation

4.1 FNN: Forward Neural Network

4.1.1 Background

Suppose we have a task we would like a machine learning model to complete (e.g. recognizing handwritten digits). At a high level, we need to perform the following tasks:

1. First, we must conceptualize the task as a “function” such that the inputs and outputs of the task can be described in a concrete mathematical sense (amenable for programmability).

2. Second, we need a way to quantify the degree to which our model is performing poorly against a known set of correct answers. This is typically denoted as the *loss* or *objective* function of the model.

3. Third, we need an *optimization strategy* which will describe how to adjust the model after feedback is provided regarding the model’s performance as per the loss function described above.

4. Fourth, we need a *regularization strategy* to address inadvertently tuning the model with a high degree of specificity to our training data, at the cost of generalized performance when handling inputs not yet encountered.

5. Fifth, we need an *architecture* for our model, including how inputs are transformed into outputs and an enumaration of all the adjustable parameters the model supports.

6. Finally, we need a robust *implementation* that executes the above within memory and execution budgets, accounting for floating-point stability, reproducibility, and a number of other engineering-related matters.

Deep learning is distinct from other machine learning models in that the architecture is heavily over-parameterized and based on simpler *building blocks* as opposed to bespoke components. The building blocks used are neurons, or particular arrangements of neurons, typically organized as layers. Over the course of training a deep learning model, it is expected that *features* of the inputs are learned and manifested as various parameter values in these neurons. This is in contrast to traditional machine learning, where features are not learned, but implemented directly.

4.1.2 Categorical Cross-EntropyLoss

More concretely, the task at hand is to train a model to recognize a 28 by 28 pixel handwritten greyscale digit. For simplicity, our model will interpret the data as a flattened 784-dimensional vector. Instead of describing the architecture of the model first, we'll start with understanding what the model should output and how to assess the model's performance. The output of our model will be a 10-dimensional vector, representing the probability distribution of the supplied input. That is, each element of the output vector indicates the model's estimation of the probability that the digit's value matches the corresponding element index. For example, if the model outputs:

$$M(\mathbf{I}) = [0, 0, 0.5, 0.5, 0, 0, 0, 0, 0, 0]$$

for some input image $\mathbf{I}$, we interpret this to mean that the model believes there is an equal chance of the examined digit to be a 2 or a 3.

Next, we should consider how to quantify the model's loss. Suppose, for example, that the image $\mathbf{I}$ actually corresponded to the digit "7" (our model made a horrible prediction!), how might we penalize the model? In this case, we know that the *actual* probability distribution is the following:

$$[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$$

This is known as a "one-hot" encoded vector, but it may be helpful to think of it as a probability distribution given a set of events that are mutually exclusive (a digit cannot be both a "7" *and* a "3" for instance).

Fortunately, information theory provides us some guidance on defining an easy-to-compute loss function which quantifies the dissimilarities between two probability distributions. If the probability of of an event E is given as $P(E)$, then the *entropy* of this event is given as $-\log P(E)$. The negation ensures that this is a positive quantity, and by inspection, the entropy increases as an event becomes less likely. Conversely, in the limit as $P(E)$ approaches 1, the entropy shrinks to 0. While several interpretations of entropy are possible, the pertinent interpretation here is that entropy is a *measure of the information conveyed when a particular event occurs.* That the "sun rose this morning" is a fairly mundane observation but being told "the sun exploded" is sure to pique your attention. Because we are reasonably certain that the sun rises each morning (with near 100% confidence), that "the sun rises" is an event that conveys little additional information when it occurs.

Let's consider next entropy in the context of a probability distribution. Given a discrete random variable X which can take on values $x_0, \ldots, x_{n-1}$ with probabilities $p(x_0), \ldots, p(x_{n-1})$, the entropy of the random variable X is defined as:

$$H(X) = -\sum_{x \in X} p(x) \log p(x)$$

For example, suppose W is a binary random variable that represents today's weather which can either be "sunny" or "rainy" (a binary random variable). The entropy $H(W)$ can be given as:

$$H(W) = -S \log S - (1 - S) \log(1 - S)$$

where S is the probability of a sunny day, and hence $1-S$ is the probability of a rainy day. As a binary random variable, the summation over weighted entropies expands to only two terms. What does this quantity mean? If we were to describe it in words, each term of the sum in the entropy calculation corresponds to the information of a particular event, weighted by the probability of the event. Thus, the entropy of the distribution is literally the *expected amount of information contained in an event* for a given distribution. If we plot $-S \log S - (1 - S) \log(1 - S)$ as a function of S, we will see something like this:

As a minor note, while $\log 0$ is an undefined quantity, information theorists accept that $\lim_{p \to 0} p \log p = 0$ by convention. Intuitively, the expected entropy should be unaffected by the set of impossible events.

As you might expect, when the distribution is 50-50, the uncertainty of a binary is maximal, and by extension the amount of information contained in each event is maximized too. Put another way, if you lived in an area where it was always sunny, you wouldn't *learn anything* if someone told you

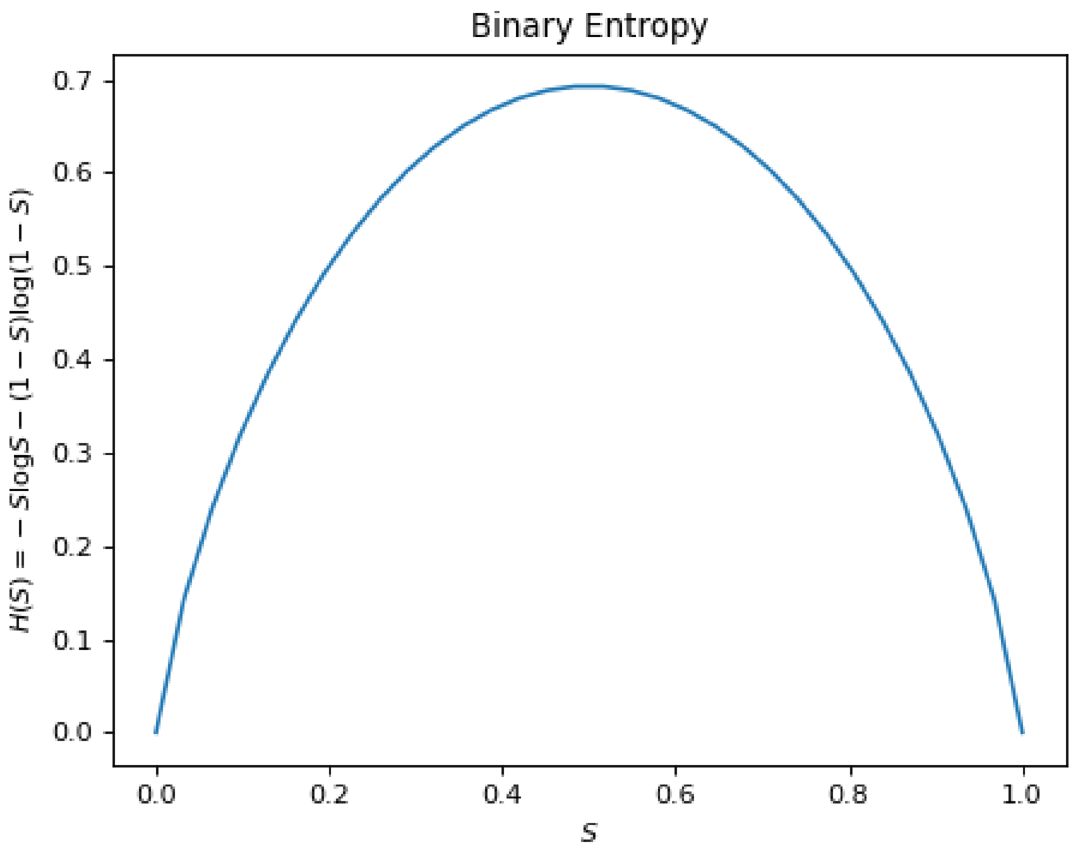


Figure 4.1: Cross Entropy matching distribution

it was sunny today. However, in a tropical region characterized by capricious weather, information conveyed about the weather is far more meaningful.

In the previous example, we weighted the event entropies according to the event's probability distribution. What would happen if, instead, we used weights corresponding to a *different* probability distribution? This is known as the *cross entropy*:

$$H(p,q) = -\sum_{x \in X} p(x) \log q(x)$$

To get some intuition about this, first, we note that if $p(x) = q(x), \forall x \in X$, the cross entropy trivially matches the self-entropy. Let's go back to our binary entropy example and visualize what it looks like if we chose a completely *incorrect* distribution. Specifically, suppose we computed the cross entropy where if the probability of a sunny day is S, we weight the entropy with $1 - S$ instead of S as in the self-entropy formula.

If you compare the values with the previous figure, you'll see that the cross entropy diverges from the self-entropy everywhere except 0.5, where $S = 1 - S$. The difference between the cross entropy $H(p,q)$ and entropy $H(p)$ provides then, a *measure of error* between the presumed distribution q and the true distribution p. This difference is also known as the Kullback-Leibler divergence or KL divergence for short.

Given that the entropy of a given probability distribution p is constant, then $H(p)$ must be constant as well. This is why in practice, we will generally seek to minimize the cross entropy between p and a predicted distribution q, which by extension will minimize the Kullback-Leibler divergence as well.

Now, we have the tools to know if our model is succeeding or not! Given an estimation of a sample's label as before:

$$M(\mathbf{I}) = [0, 0, 0.5, 0.5, 0, 0, 0, 0, 0, 0]$$

we will treat our model's output as a predicted probability distribution of the sample digit's classification from 0 to 9. Then, we compute the cross entropy between this predction and the true distribution, which will be in the form of a one-hot vector. Supposing the actual digit is 3 in this particular case ($P(7) = 1$):

$$\sum_{x \in \{0,\ldots,9\}} -P(x) \log Q(x) = -P(3) \log(Q(3)) = \log(0.5) \approx 0.301$$

Let's make a few observations before continuing. First, for a one-hot vector, the entropy is 0 (can you see why?). Second, by pretending the correct

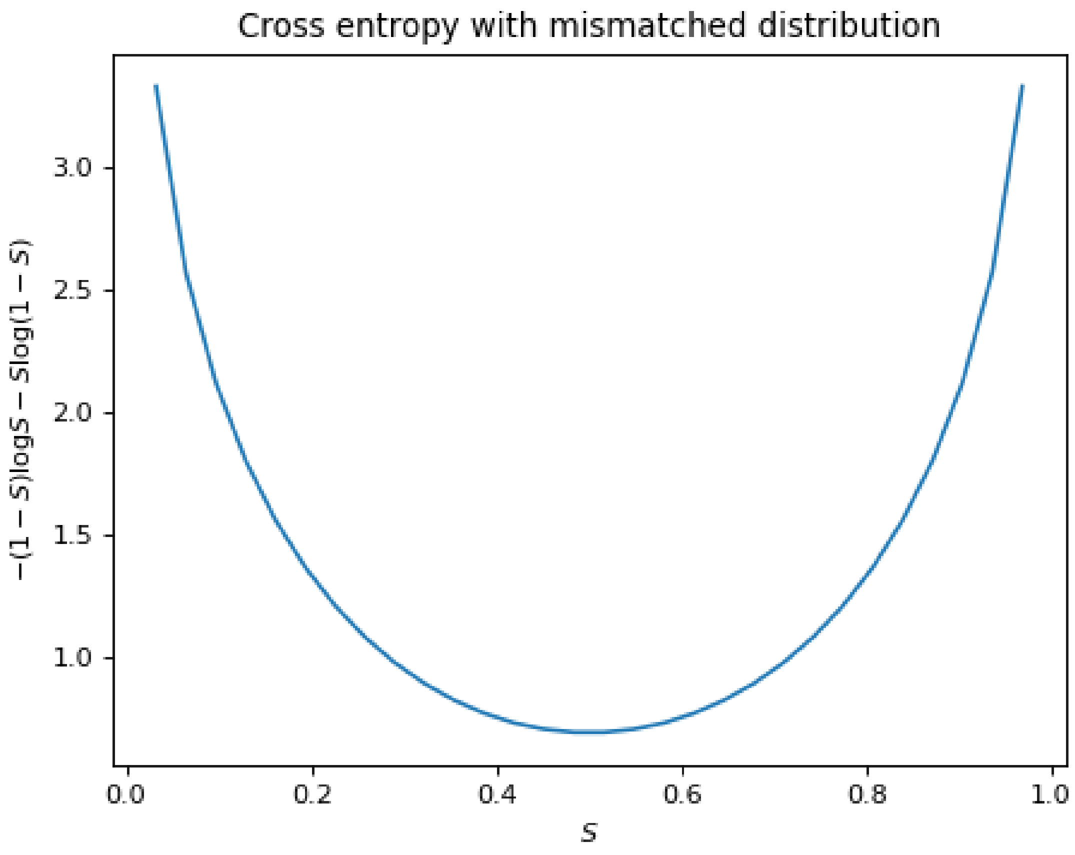


Figure 4.2: Cross Entropy mismatching distribution

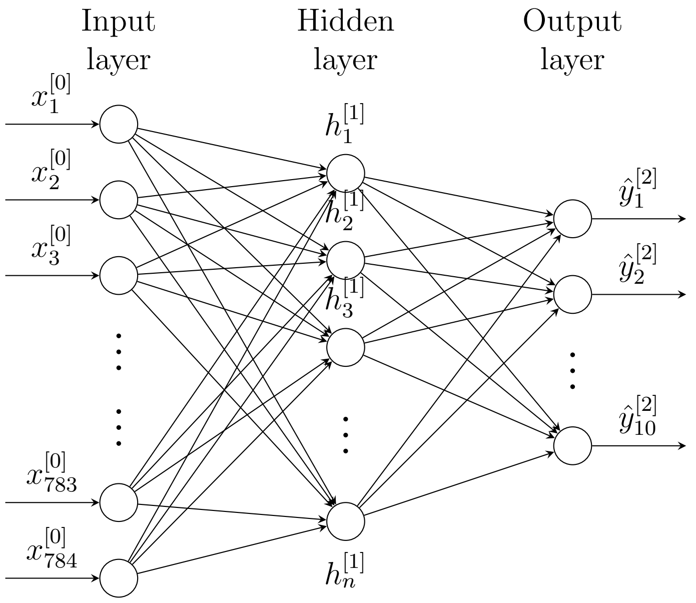
digit above is 3 and not, say, 7, we conveniently avoided $\log 0$ showing up in the final expression. A common method to avoid this is to add a small ϵ to the log argument to avoid this singularity, but we'll discuss this in more detail later.

4.1.3 Approximation Function with a Neural Network

Now that we know how to evaluate our model, we'll need to decide how to go about making predictions in the form of a probability distribution. Our model will need to take as inputs, 28x28 images (which as mentioned before, will be flattened to 784x1 vectors for simplicity). Let's enumerate the properties our model will need:

1. Parameterization - our model will need parameters we can adjust to "fit" the model to the data
2. Nonlinearity - it is assuredly not the case that the probability distribution can be modeled with a set of linear equations
3. Differentiability - the gradient of our model's output with respect to any given parameter indicates the *impact* of that parameter on the final result

There are an infinite number of functions that fit this criteria, but here, we'll use a simple feedforward network with a single hidden layer.



A few quick notes regarding notation: a superscript of the form $[i]$ is used to denote the ith layer. A subscript is used to denote a particular element within a layer or vector. The vector $\mathbf{x}$ is usually reserved for training samples, and the vector $\mathbf{y}$ is typically reserved for sample labels (i.e. the desired "answer" for a given sample). The vector $\hat{\mathbf{y}}$ is used to denote a model's predicted labels for a given input.

On the far left, we have the input layer with 784 nodes corresponding to each of the 28 by 28 pixels in an individual sample. Each $x_i^{(0)}$ is a floating point value between 0 and 1 inclusive. Because the data is encoded with 8 bits of precision, there are 256 possible values for each input. Each of the 784 input values fan out to each of the nodes in the hidden layer without modification.

In the center hidden layer, we have a variable number of nodes that each receive all 784 inputs, perform some processing, and fan out the result to the output nodes on the far right. That is, each node in the hidden layer transforms a $\mathbb{R}^{784}$ vector into a scalar output, so as a whole, the n nodes collectively need to map $\mathbb{R}^{\rightarrow}\mathbb{R}^{n}$. The simplest way to do this is with an $n \times 784$ matrix (treating inputs as column vectors). Modeling the hidden layer this way, each of the n nodes in the hidden layer is associated with a single row in our $\mathbb{R}^{n\times 784}$ matrix. Each entry of this matrix is referred to as a *weight*.

We still have two issues we need to address however. First, a matrix provides a linear mapping between two spaces, and linear maps take 0 to 0 (you can visualize such maps as planes through the origin). Thus, such fully-connected layers typically add a *bias* to each output node to turn the map into an affine map. This enables the model to respond zeroes in the input. Thus, the hidden layer as a whole has now both a weight matrix, and also a bias vector. A linear mapping with a constant bias is commonly referred to as an *affine map*.

The second issue is that our hidden layer's now-affine mapping still scales linearly with the input, and one of our requirements for our approximation function was nonlinearity (a strict prerequisite for universality). Thus, we perform one final non-linear operation the result of the affine map. This is known as the *activation function*, and an infinite number of choices present itself here. In practice, the *rectifier function*, defined below, is a perennial choice.

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

The rectifier is popular for having a number of desirable properties.

1. Easy to compute

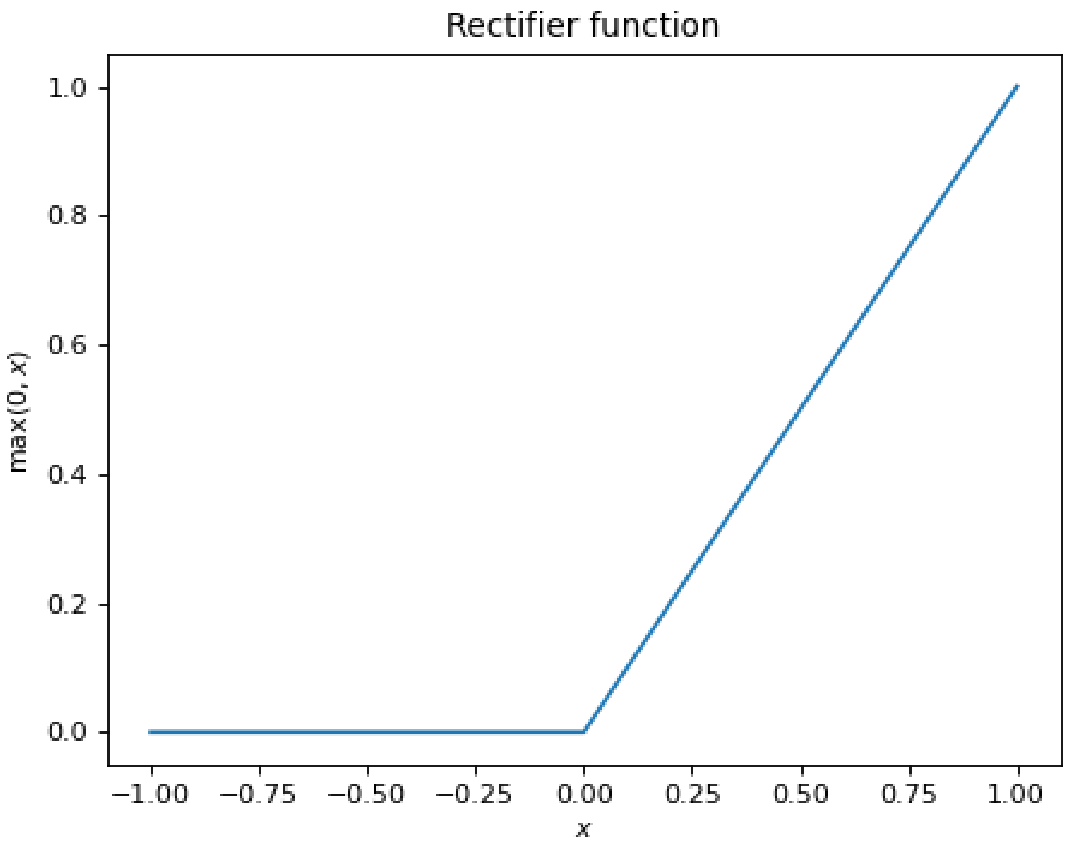


Figure 4.3: ReLU: Rectifier Linear Unit

2. Easy to differentiate (except at 0, which has not been found to be a problem in practice)

3. Sparse activation, which aids in addressing model overfitting and "unlearning" useful weights

As our hidden layer units will use this rectifier just before emitting its final output to the next layer, our hidden units may be called *rectified linear units* or ReLUs for short.

Summarizing our hidden layer, the output of each unit in the layer can be written as:

$$a_i^{[1]} = \max(0, W_i^{[1]} \cdot \mathbf{x}^{[0]} + b_i^{[1]})$$

It's common to refer to the final activated output of a neural network layer as the vector $\mathbf{a}$, and the result of the internal affine map $\mathbf{z}$. Using this notation and considering the output of the hidden layer as a whole as a vector quantity, we can write:

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= \max(\mathbf{0}, \mathbf{z}^{[1]}) \\
\mathbf{a}^{[1]}, \mathbf{b}^{[1]} &\in \mathbb{R}^n \\
\mathbf{W}^{[1]} &\in \mathbb{R}^{n\times 784} \\
\mathbf{x}^{[0]} &\in \mathbb{R}^{784}
\end{aligned}$$

The last layer to consider is the output layer. As with the hidden layer, we need a dimensionality transform, in this case, taking vectors in $\mathbb{R}^n$ and mapping them to vectors in $\mathbb{R}^{10}$ (corresponding to the 10 possible digits in the target output). As before, we will use an affine map with the appropriately sized weight matrix and bias vector. Here, however, the rectifier isn't suitable as an activation function because we want to emit a probability distribution. To be a valid probability distribution, each output of the hidden layer must be in the range $[0, 1]$, and the sum of all outputs must equal 1. The most common activation function used to achieve this is the *softmax function*:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Given a vector input z, each component of the softmax output (as a vector quantity) is given as per the expression above. The exponential functions conveniently map negative numbers to positive numbers, and the denominator ensures that all outputs will be between 0 and 1, and sum to 1 as desired. There are other reasons why an exponential function is used here, stemming from our choice of a loss function (based on the underpinning notion of maximum-likelihood estimation), but we won't get into that in too much detail here (consult the further reading section at the end to learn more). Suffice it to say that an additional benefit of the exponential function is its clean interaction with the logarithm used in our choice of loss function, especially when we will need to compute gradients in the next section.

Summarizing our neural network architecture, with two weight matrices and two bias vectors, we can construct two affine maps which map vectors in $\mathbb{R}^{784}$ to $\mathbb{R}^{n}$ to $\mathbb{R}^{10}$. Prior to forwarding the results of one affine map as the input of the next, we employ an activation function to add non-linearity to the model. First, we use a linear rectifier and second, we use a softmax function, ensuring that we end up with a nice discrete probability distribution with 10 possible events corresponding to the 10 digits.

Our network is small enough that we can actually write out the entire process as a single function using the notation we've built so far:

$$f(\mathbf{x}^{[0]}) = \mathbf{y}^{[2]} = \text{softmax}\left(\mathbf{W}^{[2]}\left(\max\left(\mathbf{0}, \mathbf{W}^{[1]}\mathbf{x}^{[0]} + \mathbf{b}^{[1]}\right)\right) + \mathbf{b}^{[1]}\right)$$

One thing to keep in mind here is that this implementation is *not* the most efficient implementation possible for a softmax layer feeding to a cross-entropy loss function by any stretch. The code and derivation here is completely general for arbitrary sample probability distributions. If, however, we can assume that the target distribution is one-hot encoded, then all gradients in this node will either be 0 or $-1/\hat{y}_k$ where k is the active label in the one-hot target. Upon substitution in the previous layer, it should be clear that important cancellations are possible that dramatically simplify the gradient computations in the softmax layer. Here's the simplification, again assuming that the kth index is the correct label:

$$\begin{aligned}
\frac{\partial J_{CE}}{\partial \text{softmax}(\mathbf{z})_i} &= \frac{\partial J_{CE}}{\partial a_i} \sum_j \begin{cases} \text{softmax}(\mathbf{z})_i\,(1 - \text{softmax}(\mathbf{z})_i) & i = j \\ -\text{softmax}(\mathbf{z})_i \text{softmax}(\mathbf{z})_j & i \neq j \end{cases} \\
&= \begin{cases} -\dfrac{\text{softmax}(\mathbf{z})_k(1 - \text{softmax}(\mathbf{z}_k))}{\text{softmax}(\mathbf{z})_k} & i = k \\ \dfrac{\text{softmax}(\mathbf{z})_i \text{softmax}(\mathbf{z})_k}{\text{softmax}(\mathbf{z})_k} & i \neq k \end{cases} \\
&= \begin{cases} \text{softmax}(\mathbf{z})_k - 1 & i = k \\ \text{softmax}(\mathbf{z})_i & i \neq k \end{cases}
\end{aligned}$$

When following the computation above, remember that $\partial J_{CE}/\partial a_i$ is 0 for all $i \neq k$. Thus, the only term in the sum that survives is the term corresponding to $j = k$, at which point we break out the differentation depending on whether $i = k$ or $i \neq k$.

This is an elegant result! Essentially, the gradient of a the loss with respect to an emitted probability $p(x)$ is simply $p(x)$ if x was not the correct label, and $p(x) - 1$ if it was. Considering the effect of gradient descent, this should check out with our intuition. The optimizer seeks to suppress probabilities predicted that should have been 0, and increase probabilities predicted that should have been 1. Check for yourself that after gradient descent is performed, the gradients derived here will nudge the model in the appropriate direction.

This sort of optimization highlights an important observation about backpropagation, namely, that backpropagation does not guarantee any sort of optimality beyond a worst-case performance ceiling. Several production neural networks have architectures that employ heuristics to identify optimizations such as this one, but the problem of generating a perfect computational strategy is NP and so not covered here. The code provided here will remain in the general form, despite being slower in the interest of maintaining generality and not adding complexity, but you are encouraged to consider abstractions to permit this type of optimization in your own architecture (a useful keyword to aid your research is *common subexpression elimination* or *CSE* for short).

4.1.4 Optimizing our network

We now have a model given above which can turn our 784 dimensional inputs into a 10-element probability distribution, *and* we have a way to evaluate how accuracy of each prediction. Next, we need a reliable way to improve the model based on the feedback provided by our loss function. This is known as function *optimization*, and most methods of model optimization are based on the principle of *gradient descent*.

The idea is quite simple. Given a function with a set of parameters which we'll denote $\boldsymbol{\theta}$, the partial derivative of that function with respect to a given parameter $\theta_i \in \boldsymbol{\theta}$ tells us the overall *impact* of θ_i on the final result. In our model, we have many parameters; each weight and bias constitutes an individually tunable parameter. Thus, our strategy should be, given a set of input samples, compute the loss our model produces for each sample. Then, compute the partial derivatives of that loss with respect to *every parameter* in our model. Finally, adjust each parameter in proportion to its impact on the final loss. Mathematically, this process is described below (note that the superscript (i) is used to denote the i-th sample):

$$\text{Total Loss} = \sum_i J(\mathbf{x}^{(i)}; \boldsymbol{\theta})$$

$$\text{Compute } \sum_i \frac{\partial J(\mathbf{x}^{(i)})}{\partial \theta_j} \;\forall\; \theta_j \in \boldsymbol{\theta}$$

$$\text{Adjust } \theta_j \to \theta_j - \eta \sum_i \frac{\partial J(\mathbf{x}^{(i)})}{\partial \theta_j} \;\forall\; \theta_j \in \boldsymbol{\theta}$$

Here, there is some flexibility in the choice of η, often referred to as the *learning rate.* A small η promotes more conservative and accurate steps, but at the cost of our model being more costly to update. A large η on the other hand results in larger updates to our model per training cycle, but may result in instability. Updating in the above fashion should adjust the model such that it will produce a smaller loss given the same inputs.

In practice, the size of the input set may be very large, rendering it intractable to evaluate the model on every single training sample in the sum above before adjusting parameters. Thus, a common strategy is to use *stochastic gradient descent* (abbrev. SGD) and perform loss-gradient-based adjustments after evaluating smaller batches of samples. Concretely, the MNIST handwritten digits database contains 60,000 training samples.

SGD is very similar, but the batch size can be much smaller than the amount of training data available. This enables the model to get more frequent updates and waste fewer cycles especially at the start of training when the model is likely wildly inaccurate.

When it comes time to compute the gradients, we are fortunate to have made the prescient choice of constructing our model solely with elementary functions in a manner conducive to relatively painless differentiation. However, we still must exercise care as there is plenty of bookkeeping involved. We will evaluate loss-gradients with respect to individual parameters when we walkthrough the implementation later, but for now, let's establish a few preliminary results.

Recall that our choice of loss function was the categorical cross entropy function, reproduced below:

$$J_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log \hat{y}_i$$

The index i is enumerated over the set of possible outcomes (i.e. the set of digits from 0 to 9). The quantities y_i are the elements of the one-hot label corresponding to the correct outcome, and $\hat{\mathbf{y}}$ is the discrete probability distribution emitted by our model. We compute $\partial J_{CE}/\partial \hat{y}_i$ like so:

$$\frac{\partial J_{CE}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

Notice that for a one-hot vector, this partial derivative vanishes whenever i corresponds to an incorrect outcome.

Working backwards in our model, we next provide the partial derivative of the softmax function:

$$\begin{aligned}
\text{softmax}(\mathbf{z})_i &= \frac{\exp z_i}{\sum_j \exp z_j} \\
\frac{\partial\left(\text{softmax}(\mathbf{z})_i\right)}{\partial z_k} &= \begin{cases} \dfrac{\left(\sum_j \exp z_j\right)\exp z_i - \exp 2z_i}{\left(\sum_j \exp z_j\right)^2} & i = k \\ \dfrac{-\exp z_i \exp z_k}{\left(\sum_j \exp z_j\right)^2} & i \neq k \end{cases} \\
&= \begin{cases} \text{softmax}(\mathbf{z})_i\left(1 - \text{softmax}(\mathbf{z})_i\right) & i = k \\ -\text{softmax}(\mathbf{z})_i \text{softmax}(\mathbf{z})_k & i \neq k \end{cases}
\end{aligned}$$

The last set of equations follow from factorizing and rearranging the expressions preceding it. It's often confusing to newer practitioners that the partial derivative of softmax needs this unique treatment. The key observation is that softmax is a vector-function. It accepts a vector as an input and emits a vector as an output. It also "mixes" the input components, thereby imposing a functional dependence of *every output component* on *every input component*. The lone $\exp z_i$ in the numerator of the softmax equation creates an asymmetric dependence of the output component on the input components.

Finally, let's consider the partial derivative of the linear rectifier.

$$\begin{aligned}
\text{ReLU}(z) &= \max(0, z) \\
\frac{\partial \text{ReLU}(z)}{\partial z} &= \begin{cases} 0 & z < 0 \\ \text{undefined} & z = 0 \\ z & z > 0 \end{cases}
\end{aligned}$$

While the partial derivative *exactly* at 0 is undefined, in practice, the derivative is simply assigned to 0. Why the non-differentiability at 0 isn't an issue has been a subject of practical debate for a long time. Here is a simple line of thinking to justify the apparent issue. Consider a rectifier function that is nudged *ever so slightly* to the right such that the inflection point is $\epsilon/2$, where ϵ is the smallest positive floating point number the machine can represent. In this case, the model will never produce a value that sits directly on this inflection point, and as far as the computer is concerned, we never encounter a point where this function is non-differentiable. We can even imagine an infinitesimal curve that smooths out the function at that inflection point if we want. Either way, experimentally, the linear rectifier remains one of the most effective activation functions for reasons mentioned, so we have no reason to discredit it over a technicality.

Now that we can compute partial derivatives of all the nonlinear functions in our neural network (and presumably the linear functions as well), we are prepared to compute loss gradients with respect to any parameter in the network. Our tool of choice is the venerable chain rule of calculus:

$$\left.\frac{\partial f(g(x))}{\partial x}\right|_x = \left.\frac{\partial f}{\partial g}\right|_{g(x)} \left.\frac{\partial g}{\partial x}\right|_x$$

This gives us the partial derivative of a composite function $f \circ g$ evaluated at a particular value of x. Our model itself is a series of composite functions, and as we can now compute the partials of each individual component in the model, we are ready to begin implementation in the next section.

4.1.5 Regularization

Regularization will not be implemented as part of this self-contained neural network, but it is such a fundamental part of most deep learning frameworks that we'll discuss it here.

Often, the dimensionality of our model will be much higher than what is stricly needed to make accurate predictions. This stems from the fact that we seldom no a priori how many features are needed for the model to be successful. Thus, the likelihood of overfitting increases as more training data is fed into the model. The primary tool to combat overfitting is *regularization*. Loosely speaking, regularization is any strategy employed to restrict the hypothesis space of fit-functions the model can occcupy to prevent overfitting.

What is meant by restricting the hypothesis space, you might ask? The idea is to consider the entire family of functions possible spanned by the model's entire parameter vector. If our model has 10000 parameters (many networks will easily exceed this), each unique 10000-dimensional vector corresponds to a possible solution. However, we know it's unlikely that certain parameters should be vastly greater in magnitude than others in a theoretically *optimal* condition. Models with "strange" parameter vectors that are unlikely to be the optimal solution are likely converged on as a result of overfitting. Therefore, it makes sense to consider ways to constrain the space this parameter vector may occupy.

The most common approach to achieve this is to add an initial penalty term to the loss function which is a function of the weight. For example, here is the cross-entropy loss with the so-called L^2 regularizer (also known as the ridge regularizer) added:

$$-\sum_{x\in X} y_x \log \hat{y}_x + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

In a slight abuse of notation, $\mathbf{w}$ here corresponds to a vector containing every weight in our network. The factor λ is a constant we can choose to adjust the penalty size. Note that when a regularizer is used, we *expect training loss to increase.* The tradeoff is that we simultaneously *expect test loss to decrease.* Tuning the regularization speed λ is a routine problem for model fitting in the wild.

By modifying the loss function, in principal, all loss gradients must change as well. Fortunately, as we've only added a quadratic term to the loss, the only change to the gradient will be an additional linear additive term $\lambda\mathbf{w}$. This means we don't have to add a ton of code to modify all the gradient calculations thus far. Instead, we can simply *decay* the weight based on a percentage of the weight's magnitude when we adjust the weight after each batch is performed. You will often here this type of regularization referred to as simply *weight decay* for this reason.

To implement L^2 regularization, simply add a percentage of a weight's value to its loss gradient. Crucially, do not adjust bias parameters in the same way. We only wish to penalize parameters for which increased magnitude corresponds with more complex models. Bias parameters are simply scalar offsets, regardless of their value and do not scale the inputs. Thus, attempting to regularize them will likely increase *both* training and test error.

4.2 RNN: Recurrent Neural Network

A recurrent neural network is a neural network that is specialized for processing a sequence of data $x(t) = x(1), ..., x(\phi)$ with the time step index t ranging from 1 to ϕ. For tasks that involve sequential inputs, such as speech and language, it is often better to use RNNs. In a NLP problem, if you want to predict the next word in a sentence it is important to know the words before it. RNNs are called recurrent because they perform the same task for

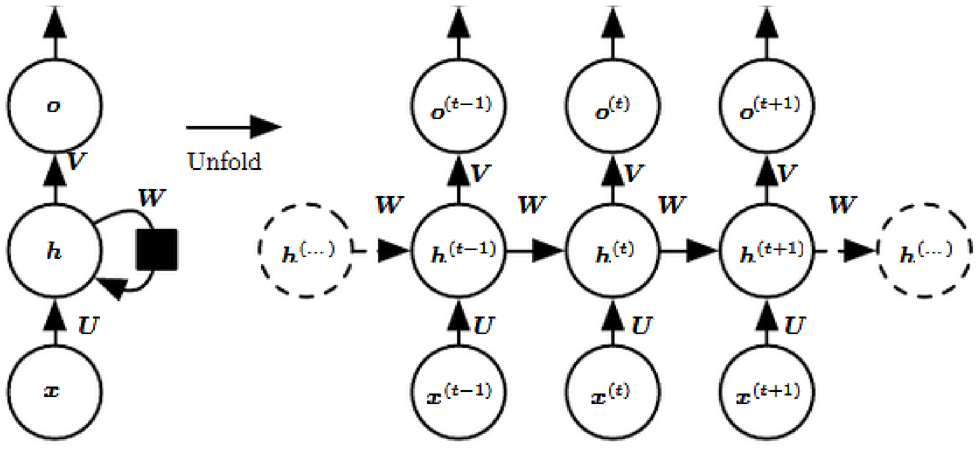


Figure 4.4: Architecture of a Recurrent Neural Network (RNN)

every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a “memory” which captures information about what has been calculated so far.

The left side of the diagram in 4.4 shows a notation of an RNN and on the right side an RNN being unrolled (or unfolded) into a full network. By unrolling we mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 3 words, the network would be unrolled into a 3-layer neural network, one layer for each word.

- Input: $x(t)$ is taken as the input to the network at time step t. For example, $x1$, could be a one-hot vector corresponding to a word of a sentence.

- Hidden state: $h(t)$ represents a hidden state at time t and acts as ”memory” of the network. $h(t)$ is calculated based on the current input and the previous time step's hidden state: $h(t) = f(U * x(t) + W * h(t-1))$. The function f is taken to be a non-linear transformation such as tanh, ReLU.

- Weights: The RNN has input to hidden connections parameterized by a weight matrix U, hidden-to-hidden recurrent connections parameterized by a weight matrix W, and hidden-to-output connections parameterized by a weight matrix V and all these weights (U, V, W) are shared across time.

- Output: $o(t)$ illustrates the output of the network. In the figure 4.4 just put an arrow after $o(t)$ which is also often subjected to non-linearity, especially when the network contains further layers downstream.

The gradient computation involves performing a forward propagation pass moving left to right through the graph shown above followed by a backward propagation pass moving right to left through the graph. The runtime is $O(\phi)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\phi)$. The back-propagation algorithm applied to the unrolled graph with $O(\phi)$ cost is called back-propagation through time (BPTT). Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.

Chapter 5

Conclusion

The correct behavior of a neural network during the training phase is given by the loss. In a given training step, if the loss is less than or equal to the loss of the previous step then the neural network is working correctly.

The implementation produced was tested with a small dataset and on small neural networks, in order to keep track of the variation of the weights during the training phase. With the passage of the benches and the epochs during the training phase, the loss of the neural networks is lowered or remain constant consequently the training works. In addition, in the most cases, 100% accuracy is achieved on the training data. In particular, the best results are produced by the RNN.

Further details of testing, profiling and examples will be reported here.