IMPLEMENTANDO UMA MÁQUINA VIRTUAL DIFERENCIÁVEL MÍNIMA EM REDES NEURAIS RECORRENTES

Felipe Borda Carregosa

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Gerson Zaverucha
Aline Marins Paes Carvalho

Rio de Janeiro
Março de 2018

IMPLEMENTANDO UMA MÁQUINA VIRTUAL DIFERENCIÁVEL MÍNIMA EM REDES NEURAIS RECORRENTES

Felipe Borda Carregosa

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Gerson Zaverucha, Ph.D.

Prof. Aline Marins Paes Carvalho, D.Sc.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Marley Maria Bernardes Rebuzzi Vellasco, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2018

Carregosa, Felipe Borda

Implementando uma máquina virtual diferenciável mínima em redes neurais recorrentes/Felipe Borda Carregosa. – Rio de Janeiro: UFRJ/COPPE, 2018.

XIII, 79 p.: il.; 29, 7cm.

Orientadores: Gerson Zaverucha

Aline Marins Paes Carvalho

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 71 – 75.

1. Redes Neurais Recorrentes. 2. Indução Neural de Programas. 3. Máquina Virtual Diferenciável. I. Zaverucha, Gerson *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A todos que me suportaram nesse processo, em especial à minha família e aos meus orientadores.

Agradecimentos

Gostaria de agradecer aos meus orientadores, Gerson Zaverucha e Aline Marins Paes Carvalho, pela contínua orientação apesar de meus atrasos repetidos lhes causando mais trabalho. Agradeço também a todos os professores e funcionários do PESC pelo conhecimento e suporte em todas as etapas. E finalmente agradeço à minha família por sempre me suportarem e prezarem pela minha educação.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTANDO UMA MÁQUINA VIRTUAL DIFERENCIÁVEL MÍNIMA EM REDES NEURAIS RECORRENTES

Felipe Borda Carregosa

Março/2018

Orientadores: Gerson Zaverucha
Aline Marins Paes Carvalho

Programa: Engenharia de Sistemas e Computação

Nos últimos anos, novas técnicas em redes neurais produziram excepcionais resultados em diversos domínios. Produzir redes neurais em que se é possível observar a lógica por trás de seu processo de decisão ainda é muito difícil, especialmente quando se deseja que também tenha desempenho competitivo com os modelos já existentes. Um passo nessa direção é o desenvolvimento recente dos programadores neurais. Nesta dissertação, propõe-se um programador neural comparativamente simples, com uma máquina virtual diferenciável bastante extensível, que pode ser facilmente integrada em arquiteturas de redes neurais de múltiplas camadas existentes, fornecendo módulos com um raciocínio mais transparente aos modelos atuais. Permite-se também adicionar a capacidade de se aprender a produzir e executar algoritmos com as mesmas ferramentas para treino e execução das redes neurais. Os testes realizados com a rede proposta sugerem que ela tem o potencial de induzir algoritmos, mesmo sem qualquer tipo de otimização especial, com resultados competitivos com as atuais arquiteturas de redes neurais recorrentes.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

IMPLEMENTING A MINIMAL DIFFERENTIABLE VIRTUAL MACHINE IN RECURRENT NEURAL NETWORKS

Felipe Borda Carregosa

March/2018

Advisors: Gerson Zaverucha
Aline Marins Paes Carvalho

Department: Systems Engineering and Computer Science

Deep Learning techniques have achieved impressive results in many domains over the last few years. However, it's still difficult to produce understandable models that clearly show the embedded logic behind the decision process while still having competitive performance. One step in this direction is the recent development of neural programmers. In this work, it's proposed a very simple neural programmer with an extensible differentiable virtual machine that can be easily integrated in existing deep learning architectures, providing modules with more transparent reasoning to current models. At the same time it enables neural networks to learn to write and execute algorithm within the same training environment. Tests conducted with the proposed network suggests that it has the potential to induce algorithms even without any kind of special optimization and being competitive with current recurrent neural networks architectures.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Álgebra, lógica e algoritmos em geral são conhecimentos e práticas antigas e fundamentais da construção não só do conhecimento humano mas de toda infra-estrutura moderna. Essas ferramentas permitem escapar do simples reconhecimento de padrão no qual o cérebro é excepcional e trabalhar em um nível mais profundo e abstrato, construindo regras universais que são muitas vezes independentes dos agentes e contexto no mundo real.

Esse foi o foco das pesquisas em Inteligência Artificial durante as décadas de 70 e 80, quando diversos sistemas de raciocínio usando uma base de conhecimento (um conjunto de regras) e um sistema de inferência (que recebem fatos e através da base de conhecimento geram novos fatos) foram desenvolvidos. Havia grandes problemas, no entanto, em particular na criação da base de conhecimento. Na maioria das vezes essas regras tinham que ser escritas com cuidado por especialistas, e a enorme complexidade de qualquer entidade do mundo real tornava os sistemas impraticáveis para aplicação em domínios que não sejam consideravelmente restritos.

Desde essa época, já existia o conceito de *Redes Neurais*, o perceptron [1], porém limitações dos algoritmos existentes na época, em particular a falta de mecanismos para o treino de redes com múltiplas camadas, a falta de poder de processamento e limitada quantidade de amostras para o treino supervisionado acabou levando seu potencial a ser desprezado por muitas décadas. Mesmo muito mais tarde, na década de 90, com algoritmos de treino como o *backpropagation* e processadores muito mais capazes, essas técnicas acabaram no segundo plano, comparadas com técnicas mais eficientes em termos de número de amostras como por exemplo *Máquina de Vetores de Suporte* [2].

A grande mudança de perspectiva ocorreu no início da década de 2010, graças ao acúmulo de aprimoramentos ao longo dos anos: novas técnicas de otimização, em

particular o uso de GPUs para acelerar o processamento, o aparecimento de bases de dados com um número muito superior de amostras, novas técnicas para melhorar a convergência e melhores ferramentas de desenvolvimento. Surgiu também uma nova abordagem em conflito com o pensamento e a teoria vigente, conhecida como *Aprendizado Profundo* ou *Deep Learning*, que envolve o uso de redes com grande número de camadas e parâmetros para solucionar problemas que antes usava-se modelos bem mais simples. A partir daí, Redes Neurais, conseguiram se estabelecer como estado da arte em diversos problemas nas áreas como o processamento de imagens, voz, texto e até mesmo em jogos como Xadrez e Go [3].

Na interpretação de texto, em particular, o grande avanço foi graças a formas eficientes de se produzir representações distribuídas tanto de palavras (tais como via word2vec [4]) como de expressões e frases (por meio de redes recorrentes). Também viu-se avanços na área de raciocínio sobre texto, em áreas como *textual entailment* [5] e *question aswering* [6], em que se destacam o uso de redes recorrentes e recursivas utlizando fortemente mecanismos de atenção [7], aprendendo a discernir elementos semânticos do texto, por exemplo, *textual entailment*, se uma frase é consequência lógica da outra, se são independentes ou se contradizem.

Apesar do avanço enorme na capacidade de criar associações e reconhecer padrões altamente complexos, uma área com comparativo menor progresso diz respeito a incluir formas de raciocínio as quais computadores já possuem tradicional facilidade em resolver, tais como os mencionados raciocínio lógico e algébrico. Usando as técnicas convencionais, Redes Neurais têm dificuldade de aprender, com boa generalização, mesmo algoritmos simples como somas binárias. Como essas redes são efetivamente caixas-pretas, adicionar diretamente informação *a priori* para facilitar esse processo, visto que muitas vezes já se sabe o algoritmo, ainda é um desafio sem solução.

Uma abordagem que vem sendo explorada recentemente trata da criação de *máquinas virtuais diferenciáveis*, também chamadas de programadores neurais ou programadores diferenciáveis, nas quais programas arbitrários são representados como uma função contínua, de forma que programas que solucionam cada problema podem ser encontrada eficientemente utilizando técnicas de otimização baseadas no cálculo do gradiente. Dessa forma, um programador humano pode adicionar o comportamento desejável para o modelo e as lacunas deixadas podem ser diretamente aprendidas supervisionadamente com pares de entradas e saídas esperadas para o programa. Exemplos são o *TerpreT* [8] e *Differentiable Forth* [9].

Alternativamente, outros modelos deixam todo o comportamento em aberto, porém permitem à rede acessar um conjunto de módulos com operações relevantes pré-definidas e que transformam elementos da sua memória ou estado, livrando a rede da necessidade de aprender o comportamento em si, apenas quais, quando e a

quais dados devem ser aplicadas. Esses modelos incluem a *Programmer Network* [10], a *Programmer Network Interpreter* [11], a *Neural Random Access Machine* [12] e o modelo proposto nessa dissertação.

No entanto, esses sistemas buscam resolver problemas específicos de ponta a ponta, requerendo, inclusive, bases de dados e formatos de entrada e saída específicos para a classe de problema que resolvem. Isso dificulta integrá-los em qualquer sistema de aprendizado profundo, o que adicionaria uma nova capacidade aos modelos existentes em vez de buscar substituí-los.

1.2 Objetivo da dissertação

O objetivo da dissertação é propor um novo modelo que cubra essa deficiência de máquinas virtuais diferenciáveis, de permitir fácil integração aos modelos preexistentes que utilizem redes neurais recorrentes.

O teste da viabilidade do modelo será na sua capacidade de significativamente reduzir o erro para programas muito longos, com a partir de 50 instruções sequenciais, em que mesmo com apenas 2 instruções e 2 argumentos a se escolher por passo, seriam $(2 * 2)^{50}$, ou mais de 10^{33}, programas possíveis. O enorme espaço de busca torna a solução extremamente difícil de se encontrar para modelos de buscas sem nenhuma informação e sem a possibilidade de testar todos os modelos simultaneamente usando as propriedades de modelos diferenciáveis.

1.3 Contribuições

Como contribuições dessa dissertação, podemos citar:

1. Introdução de uma nova estratégia para construir máquinas virtuais diferenciáveis, a partir da arquitetura de uma das redes neurais recorrentes mais populares na atualidade.

2. O novo modelo não depende de nenhum tipo de entrada ou saída especial. Pode, então, ser utilizado diretamente em qualquer modelo que usaria uma rede recorrente no lugar, oferecendo uma lógica mais fácil de observar e manipular além de poder resolver classes de problemas difíceis para redes neurais comuns.

3. Comparado com os programadores neurais preexistentes, o novo modelo é capaz de construir algoritmos similares com um número de parâmetros consideravelmente menor e menos operações que podem causar instabilidade numérica, permitindo um custo menor para o seu treino.

4. Apesar de uma base simples, o modelo pode ser estendido para construir algoritmos mais complexos, devido a sua fácil integração em redes profundas, e através de extensões como testes condicionais e gerenciamento de memória externa.

5. Uso de conceitos da lógica difusa para estender a funcionalidade da máquina virtual de forma completamente diferenciável, incluindo um novo método para seleção de operações agregadoras e um novo método de implementação de operadores condicionais.

1.4 Organização do texto

O capítulo 2 introduz os conceitos fundamentais utilizados no trabalho, apresentando uma perspectiva da evolução histórica das redes neurais relacionadas até o presente. O capítulo 3 introduz conceitos chaves relacionados a programadores neurais e máquinas virtuais diferenciáveis, com alguns exemplos das implementações mais conhecidas na literatura. No capítulo 4 é apresentado o modelo proposto juntamente com uma discussão sobre as possíveis instruções que podem ser implementadas, e como o modelo pode ser estendido. O capítulo 5 possui os resultados dos testes com o modelo, e o capítulo 6 a conclusão da dissertação. O anexo A apresenta brevemente uma implementação do modelo.

Capítulo 2

Background

Nesse capítulo serão introduzidos os conceitos básicos de Redes Neurais, incluindo aspectos chaves para o modelo, como as redes recorrentes e mecanismos de atenção. Também serão brevemente explicados elementos de Lógica Difusa que serão aplicados adiante.

2.1 Redes Neurais

2.1.1 *Perceptron*

Uma das primeiras Redes Neurais Artificiais foi o *Perceptron* [13], criado no fim da década de 50. Ele é composto de um, ou múltiplos para gerar mais de uma saída, do simples módulo visto na figura 2.1. O modelo, inspirado nas pesquisas sobre o cérebro, consiste em um "neurônio"conectado a múltiplas entradas multiplicadas por pesos, que emulam os dendritos. Os pesos podem ser treinados, de forma que pode-se aprender a prioridade de cada entrada na ativação do neurônio. O neurônio, por sua vez, ao receber uma entrada combinada igual ou maior que um limiar, muda sua saída de desativado (zero) para ativado (um). O sistema de equações do *perceptron* é:

$$f(x) = \begin{cases} 1 & \text{se } w \cdot x + b > 0 \\ 0 & \text{caso contrário} \end{cases}$$

Em que N é o número de entradas, $w \in \mathcal{R}^N$ é um vetor representando os pesos, que pertencem ao conjunto dos números reais, $x \in \mathcal{R}^N$ é um vetor representando as entradas, $w \cdot x$ é um produto escalar que produz uma média ponderada para introduzir ao neurônio e b é o limiar ou viés (*bias*), a variável que define o quanto o neurônio precisa ser excitado para ativar (quanto $w \cdot x$ precisa ser para o valor ultrapassar zero). Uma forma alternativa de escrever a equação é:

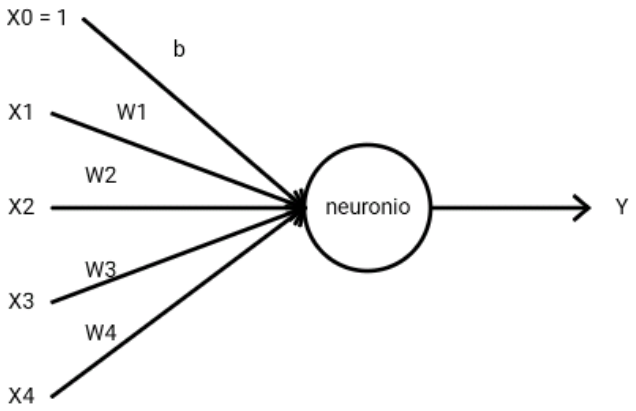

Figura 2.1: Um *perceptron* com 4 entradas, 4 pesos e um viés alimentando o neurônio. Quando as entradas ponderadas pelos respectivos pesos ultrapassa o limiar definido pelo viés, a saída se torna um, caso contrário zero.

$$H(w \cdot x + b) \tag{2.1}$$

Em que $H(...)$ faz o papel da chamada função de ativação, e no caso particular a Função de Heaviside, popularmente conhecida como função degrau (que retorna zero para entradas negativas, um para saída positivas e tipicamente $1/2$ para saídas igual a zero). A função $a = w \cdot x + b$ aparece frequentemente e é conhecida como função ou transformação afim. Alternativamente, pode-se adicionar uma entrada fixa em um, e adicionar o viés aos pesos para simplificar a notação para apenas $w' \cdot x'$.

A função afim define um hiperplano no espaço, transladado da origem pelo viés, dividindo o espaço em dois, e a função degrau classifica o que está acima e o que está abaixo dessa divisão. Assim, o *perceptron* é um classificador linear.

Um algoritmo simples para achar os pesos (e o limiar) é descrito a seguir.

1. Inicializar os pesos e o limiar em zero ou um número aleatório.
2. Calcular a saída atual.
3. Adicionar ao peso atual a diferença entre a saída calculada e a saída esperada, ou seja, o erro, multiplicada pela entrada, como escala.
4. Repetir a partir do segundo passo até convergir.

A idéia é que a cada iteração o hiperplano se mova na direção da resposta desejada. O algoritmo é garantido de convergir se as amostras de treino forem linearmente separáveis. Caso não seja, como no exemplo do notável crítico dos *perceptrons*,

Marvin Minsky, o XOR, uma convergência nunca será obtida e o algoritmo sempre continuará tentando melhorar sua classificação sem sucesso. Nesses casos, um aprimoramento simples é guardar e retornar apenas o melhor resultado, mesmo que não haja uma solução perfeita. Esse algoritmo é conhecido como *Pocket Algorithm* [14].

2.1.2 *Multi-Layer Perceptron* (MLP)

A grande fraqueza dos *perceptrons* é serem capazes de apenas criar separações lineares no espaço, limitando sua aplicação e popularidade até a segunda metade da década de 80. Nessa década introduziram uma nova técnica que permitiria o treinamento de múltiplas camadas, ou seja, ligar as saídas do *perceptron* a novos *perceptrons*. Nessa arquitetura existe um conjunto de camadas intermediárias manipulando suas entradas, chamadas de camadas ocultas, e no fim uma camada de saída que produz os resultados de acordo com a figura 2.2. Com isso, a rede passa a ser capaz de adaptar qualquer função existente mesmo que tenha apenas uma camada oculta, desde que suficientemente larga (a explicação intuitiva é que a camada oculta infinitamente larga pode operar como uma *Look Up Table* que mapeia todas as entradas existentes a um estado oculto único, e mapeia esse estado oculto único para qualquer saída possível na camada de saída).

Cada neurônio recebe uma transformação afim de toda entrada, de onde vem o nome de redes, ou camadas, densas para diferenciar de outras redes nas quais nem todas as entradas são conectadas em todos os neurônios, e aplica uma função de ativação não linear. Historicamente, as transformações mais comuns são a logística e a tangente hiperbólica. A necessidade de uma transformação não linear é porque uma cadeia de transformações afim pode ser representada por uma única transformação afim, logo sistemas puramente lineares têm profundidade máxima efetiva de um. Isso pode ser visto facilmente no seguinte conjunto de equações representando duas camadas sem função de ativação, em que W_1, W_2 são matrizes de pesos de cada camada, b_1, b_2 são os viés e x a entrada:

$$W_2(W_1x + b_1) + b_2$$

$$(W_2W_1)x + (W_2b_1 + b_2)$$

$$W_3x + b_3$$

Em que $W_3 = (W_2W_1)$ e $b_3 = (W_2b_1 + b_2)$. Logo, a rede neural linear de duas camadas é equivalente à de apenas uma camada.

O uso de sigmoidais como função não linear traz várias vantagens:

1. Aproximam a função degrau, sendo adequadas para classificação.

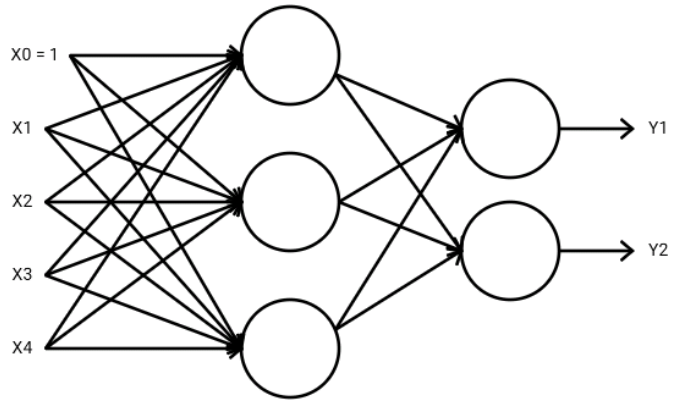

Figura 2.2: Um *Multi-Layer Perceptron* com 4 entradas, 1 camada oculta com 3 neurônios e dois neurônios na camada de saída.

2. A função logística em particular restringe o espaço em [0,1] sendo adequado para regressões cujo resultado seja uma probabilidade.

3. São diferenciáveis em toda sua extensão.

4. A derivada de ambas são função da própria, facilitando os cálculos para a regressão.

No entanto, elas também possuem uma desvantagem importante: para valores distantes da origem mesmo grandes variações da entrada produzem variações muito pequenas, visível na figura 2.3, o que causa um dos principais problemas de redes neurais com muitas camadas, o desaparecimento do gradiente (*vanishing gradient*) [15]. Isso ocorre porque a logística e a tangente hiperbólica transformam entradas de $[-\infty, \infty]$ em variáveis de $[0, 1]$ e $[-1, 1]$, respectivamente, e pelo seu formato sigmoidal, valores distantes do eixo central podem ter grandes variações sem provocar uma diferença significativa de valores na saída dessas funções de ativação. Dessa forma, fica difícil não só estimar uma variação adequada dos pesos da própria camada, já que variações no peso não mudam significamente a saída, como também de estimar o erro para toda camada anterior. E através de muitas camadas esse erro vai se compondo, somado também pela restrição de computadores trabalharem eficientemente com números muito pequenos.

Uma alternativa recente é o uso da ReLU (*Rectifier Linear Unit*) [16], que utiliza a função $max(0, x)$, que retorna o maior entre os dois argumentos, trazendo benefícios não só em eficiência (apenas uma comparação) como em minimizar o problema de desaparecimento de gradiente acima tendo saída $[0, \infty]$ e gradientes sempre igual a zero ou um, sendo adequada para redes com muitas camadas. O problema, no entanto, é que se muitos neurônios tiverem entradas negativas boa parte da rede pode ficar inativa (gradiente igual a zero), impedindo o cálculo dos pesos.

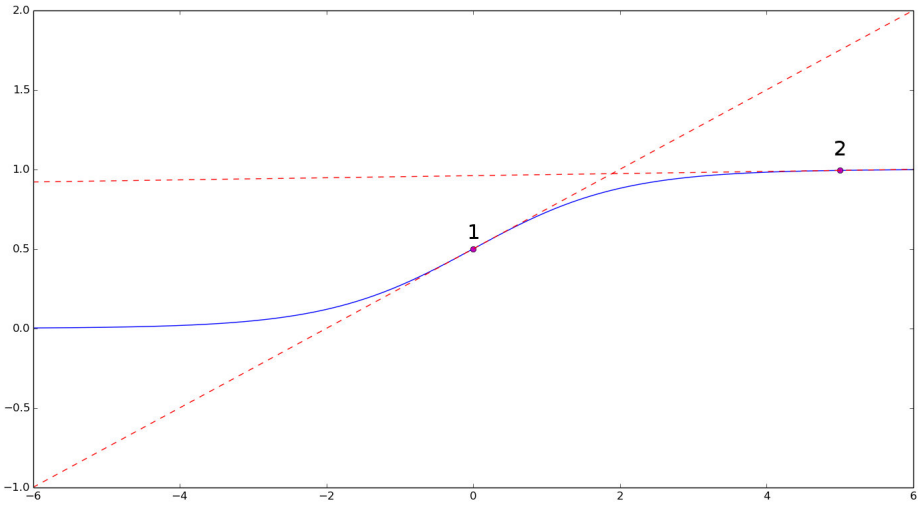

Figura 2.3: Uma função sigmóide. Quanto mais o valor se afasta do centro (2), mais a função tende a se tornar paralela ao eixo x, e portanto sua derivada tende a zero.

Um exemplo comum de MLP para regressão linear (a rede produz um número na saída em vez de uma classificação zero ou um) usado até hoje é:

$$o = W_2 \cdot \tanh(W_1 \cdot x + b) + b_2 \tag{2.2}$$

Para classificação ou regressão logística (retornar uma probabilidade) pode-se usar $f(o)$, em que f é a função logística.

Definida a estrutura, vejamos como é possível achar os pesos adequados. O objetivo é a minimização do erro, ou seja da diferença entre o valor que a rede gera para uma entrada e o valor que se espera a partir da base de dados, ou seja, é um problema de otimização. Essa forma de treino em que se tem os dados desejados se chama treinamento supervisionado. Sendo a rede diferenciável em todos os pontos, pode-se também calcular o gradiente em todos os pontos. A generalização da derivada para múltiplas variáveis em um ponto é um vetor que aponta para a direção de maior crescimento, e portanto o inverso dela aponta para a direção de maior redução. Pode-se então somar esse vetor aos pesos atuais, multiplicado por uma constante que determina o tamanho do passo (a taxa de aprendizado) para atualizá-lo para uma posição mais próxima da solução que minimiza o problema. Com isso, pode-se escolher um percurso muito mais eficiente até um mínimo adequado. Essa técnica é conhecida como *Gradiente Descendente* [17], e a atualização dos pesos ocorre da seguinte forma:

$$w^{(t+1)} = w^t - \eta * \frac{\partial E}{\partial w^t} \tag{2.3}$$

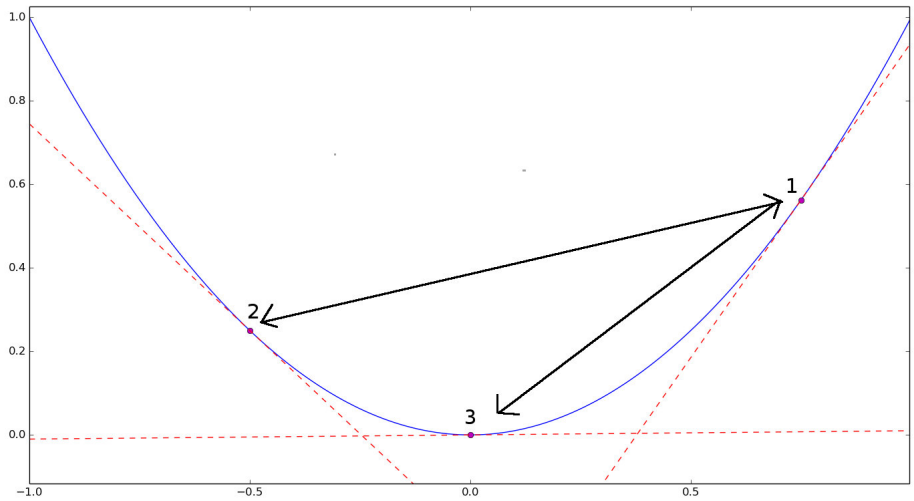

Figura 2.4: O processo do gradiente descendente em uma dimensão. A cada passo o gradiente revela a direção que reduz o valor da função. Caso a taxa de aprendizado seja muito grande, o modelo pode ser incapaz de achar a solução em 3, ficando permanentemente saltando demais entre valores em volta como 1 e 2.

Onde w é o peso que queremos atualizar, t a iteração atual, η a taxa de aprendizado e $\frac{\partial E}{\partial w^t}$ a taxa de variação do erro em função dos pesos. Uma taxa de aprendizado muito pequena pode implicar um tempo maior para convergir, pois os passos ficarão reduzidos, enquanto uma taxa grande pode tornar o modelo incapaz de convergir por ter um passo mais longo que a distância entre o ponto original e o objetivo. Um exemplo do processo está na figura 2.4.

O grande avanço que permitiu a estrutura de múltiplas camadas foi o algoritmo do *Backpropagation* [18]. Esse algoritmo é uma aplicação da regra da cadeia para múltiplas variáveis de cálculo. O erro diz a variação desejada na camada de saída, e com isso podemos calcular o efeito da variação de cada peso (em proporção ao valor de sua entrada e ao erro do neurônio de saída). Se parasse aí e a ativação fosse a função degrau, somando aos pesos atuais teríamos um algoritmo igual ao *perceptron*, movendo os pesos e, portanto, o hiperespaço na direção que minimiza o número de amostras classificadas erradas.

O primeiro passo portanto é calcular o erro E, no caso abaixo o erro quadrático:

$$E = \frac{1}{2}\sum_{i=1}^{M}(a_i - y_i)^2 \tag{2.4}$$

Em que $\frac{1}{2}$ é tipicamente usada para facilitar as contas, M o número de neurônios na saída e a a saída de 2.2 quando a função de ativação é linear e y o valor desejado para cada entrada x. A função de erro é tipicamente um par da função de ativação na camada de saída. Para saídas lineares tenta-se minimizar o erro quadrático.

Para regressão logística, por exemplo, tenta-se minimizar a distância entre as duas distribuições probabilísticas através da entropia cruzada.

Calculado o erro, no processo conhecido como *forward pass*, precisamos corrigir os pesos para aproximar-se da resposta desejado segundo 2.3, o chamado *backward pass*. O gradiente em função dos pesos da camada de saída é:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a}\frac{\partial a}{\partial o}\frac{\partial o}{\partial w} \tag{2.5}$$

Onde $o = W \cdot x + b$ é a saída do neurônio, com x sendo sua entrada, seja a entrada da rede ou a saída da camada anterior, e W e b seus pesos, e $a = f(o)$ é a saída da função de ativação, e E o erro.

Mas como temos múltiplas camadas precisamos corrigir os pesos também nas camadas anteriores. A saída do neurônio na camada N é $f(W^{(N)}a^{(N-1)} + b^{(N)})$, e já ajustamos $W^{(N)}$ e $b^{(N)}$) através das equações 2.3 e 2.5, então falta ajustar a saída anterior $a^{(N-1)}$, o que será possível através dos pesos e do viés $W^{(N-1)}$ e $b^{(N-1)}$). Da mesma forma que o efeito do peso na resposta é proporcional ao valor da entrada e ao erro, a da saída da camada anterior é proporcional ao peso e ao erro nos neurônios da camada a frente. Mas como tipicamente um neurônio na camada $N-1$ está ligado a todos os neurônios na camada N, o quanto ele deve ser mudado, ou seja seu erro, é proporcional ao somatório de todos os pesos e erros de neurônios na camada à frente.

$$\frac{\partial E}{\partial a^{(N-1)}} = \sum \frac{\partial E}{\partial o^{(N)}}\frac{\partial o^{(N)}}{\partial a^{(N)}}\frac{\partial a^{(N)}}{\partial w}\frac{\partial w}{\partial a^{(N-1)}} \tag{2.6}$$

Onde w é o peso que está ligado nos respectivos neurônios sendo somado, e N a camada. Sabendo o erro nessa camada de ativação, calcula-se o passo anterior só que nessa camada usando o novo erro para a camada de ativação, processo que é chamado de fazer o *backpropagation* do erro, realizando o *backward pass*. Feito isso repete-se o processo a partir de um novo *forward pass*.

Um último aspecto é quantas amostras são utilizadas para gerar o gradiente. O algoritmo tradicional utiliza uma média dos gradientes de todas as amostras da base de dados, constituindo uma atualização por época, que é o nome dado a uma iteração completa pelo banco de dados. Dessa forma, o gradiente vai apontar na melhor direção para a solução do problema para a base de treino. Porém, como redes neurais são modelos altamente não convexos essa estratégia determinística muitas vezes se torna incapaz de escapar de um mínimo local quando para sair é preciso em algum momento aumentar o erro. Além disso, possui alto custo em memória para bases de dados grandes, tais como imagem, especialmente em dispositivos limitados em memória como GPUs.

O outro extremo usado é atualizar a cada amostra, a variante conhecida como

Stochastic Gradient Descent. Dessa forma, como o gradiente será muito diferente a cada amostra, muitas vezes a otimização decidirá por piorar o resultado, especialmente quando a entrada tem ruído, ou o modelo não é expressivo o suficiente para modela-la. Isso permite que não se fique preso permanentemente em insatisfatórios mínimos locais enquanto, em média, ainda tenderá para a direção correta, pois as amostras na totalidade apontam para a direção correta. Por outro lado, esse caminhar aleatório causa a otimização a demorar muito mais tempo para convergir.

A técnica mais comum usada na atualidade é uma combinação das duas abordagens, usando apenas um subgrupo pequeno de todas as amostras, o que recebe o nome de *Mini-batch Gradient Descent*, ou Gradiente Descendente em Lote. Dessa forma, direciona-se melhor o gradiente para a solução correta comparado ao *Stochastic Gradient Descent*. Ao mesmo tempo acelera a convergência ao manter parte da probabilidade de tomar decisões erradas (quando todo o subgrupo aponta em uma direção que aumenta o erro como um todo). Isso permite escapar de mínimos locais melhor que o *Gradient Descent*, além de manter o custo em memória relativamente baixo.

2.1.3 Redes Neurais Recorrentes (RNN)

Embora MLPs sejam extremamente populares hoje em dia, elas não são capazes de eficientemente lidar com qualquer tipo de entrada. Olhando sua estrutura, vemos que a entrada, dada pelo vetor x, não pode ter seu tamanho alterado sem se modificar o tamanho da matriz W da primeira camada, e portanto necessitando a rede a ser retreinada para lidar com o novo conjunto de pesos.

Não só isso, mas a posição da entrada tem muito efeito no processamento, por exemplo recebendo como entrada uma lista de palavras ["ele", "comeu", "pizza",] e ["ele", "comeu", "a", "pizza"] "pizza"apesar de ser a mesma palavra com o mesmo significado e no mesmo contexto será recebida por um conjunto de pesos diferente na primeira camada, tendo que se treinar agora os dois conjuntos diferentes para produzir a representação adequada para "pizza"nas duas situações. Isso dificulta o treinamento, já que todos os pesos terão que aprender a lidar com todas as palavras, mesmo quando a base de dados não possui as palavras em todas as posições possíveis.

Para lidar com esses dois casos existe uma classe de redes neurais focadas no processamento de listas como entradas. Essa tarefa é vital em áreas como processamento de linguagem natural, voz, sinais e séries temporais de eventos em geral, tais como mercado de ações e históricos do consumo de redes elétricas até servidores web.

A figura 2.5 representa a estrutura "desenrolada"de uma *Rede Neural Recorrente* (*Recurrent Neural Network* ou RNN). A cada camada ela recebe um novo elemento

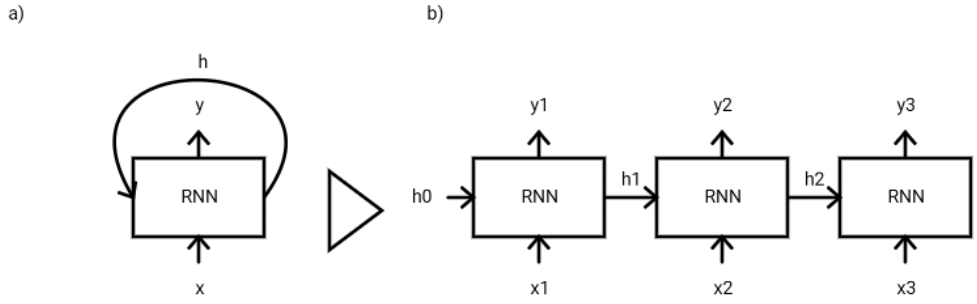

Figura 2.5: a) Uma rede neural recorrente. A camada oculta é realimentada como entrada para o passo seguinte. b) a rede neural desdobrada no tempo, com x1-x3 e y1-y3 sendo as entradas e saídas nos passos 1-3.

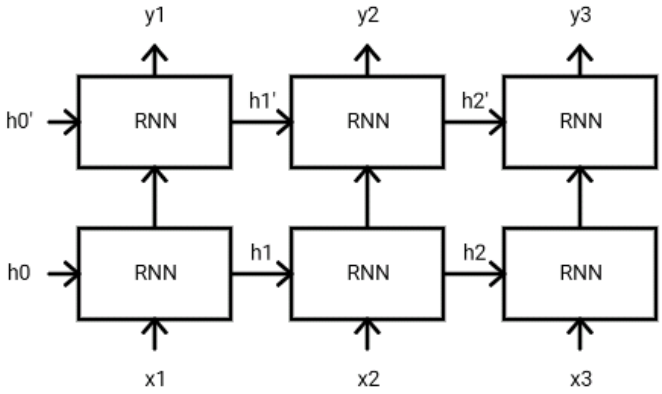

Figura 2.6: Uma rede recorrente com duas camadas, as entradas da segunda camada são as saídas da primeira. Dessa forma aumenta-se a expressividade da rede a cada passo.

da lista de entradas e o resultado da camada anterior, ou seja, para cada entrada X_t e H_{t-1}, a saída Y_t é dada por uma função $f(X_t, H_{t-1}) = f(X_t, f(X_{t-1}, H_{t-2})) = ... = f(X_t; X_{t-1}, X_{t-2}, ...X_0)$ em que t é o número da camada desenrolada, comumente chamado de passo no contexto de redes recorrentes. Na última camada temos uma representação de todas as entradas.

A segunda mudança na comparação com MLPs é que os respectivos pesos de cada camada são amarrados, ou seja, $W_{i,j}^{(N)} = W_{i,j}^{(N-1)} = ... = W_{i,j}^{(0)}$, em que i é a entrada da camada e j o campo do estado oculto seguinte, definindo juntamente cada peso de uma camada, e N o número de camadas ou passos. Portanto, podemos representar o modelo como se a saída de cada camada fosse realimentada nos pesos da própria camada, da onde vem o nome de rede recorrente.

Assim como em MLPs pode-se adicionar camadas facilmente, o mesmo é possível

com RNNs simplesmente conectando a sua saída a cada passo à entrada da camada superior, como na figura 2.6. Aumenta-se assim a sua capacidade de aprendizado a cada passo, aumentando o número de parâmetros livres (não amarrados) e o tamanho do estado oculto contendo a informação de todos os passos prévios.

Essas redes são treinadas também com *Backpropagation*. Dada a estrutura desenrolada, faz-se o *backpropagation* da mesma forma que no MLP e, ao se atualizar os pesos com o gradiente no final, usa-se a soma das correções de cada peso correspondente de cada camada gerando uma variação comum aos pesos amarrados. Esse processo de desenrolar e ajustar os pesos amarrados recebe o nome específico de *Backpropagation Through Time* (BPTT) [19]. Naturalmente, modificando os pesos de forma amarrada cria saltos caóticos e com maior dificuldade de alcançar um erro aceitável comparado a redes MLP com plena liberdade de ação, porém na prática redes recorrentes têm tido grande sucesso em suas aplicações.

Uma limitação imediata das redes recorrentes é que o tamanho da lista de entradas define o número de passos a serem executados e, portanto, o número de saídas, que são sempre geradas a cada passo. Não é possível ter mais saídas que entradas, embora parte da lista de saída pode ser ignorada, em particular quando só se interessa pelo resultado no último passo. Um exemplo de problema que essa limitação pode causar é na tradução entre línguas, em que a língua destino pode ter mais palavras que a de origem.

Para resolver esse problema, uma estrutura comum é uma camada da rede processar toda a entrada e gerar um resultado final que vai ser a entrada de cada passo de uma segunda RNN que poderá gerar uma saída de qualquer tamanho. Como a primeira RNN codifica a entrada, ela é chamada de *encoder* ou codificador, e como a segunda RNN decodifica sua entrada e gera uma saída desejada é chamada de *decoder* ou decodificador, e a estrutura, portanto, chamada de *Encoder-Decoder* [20]. Esse modelo é exemplificado na figura 2.7, com um encoder codificando uma lista de 3 entradas, e o decodificador produzindo 2 saídas a partir do processamento final do codificador.

Sua criação foi importante para a área de tradução por máquina, em que línguas diferentes não só envolvem número de palavras diferentes como mencionado, como também o conhecimento completo da frase é necessária antes de começar a traduzir a primeira palavra, mas sua aplicação rapidamente se expandiu para muitos outros problemas.

2.1.4 Long Short-Term Memory (LSTM)

Vendo a forma desenrolada fica evidente que uma rede neural recorrente com muitos passos sofre dos mesmos problemas de redes neurais com muitas camadas. O

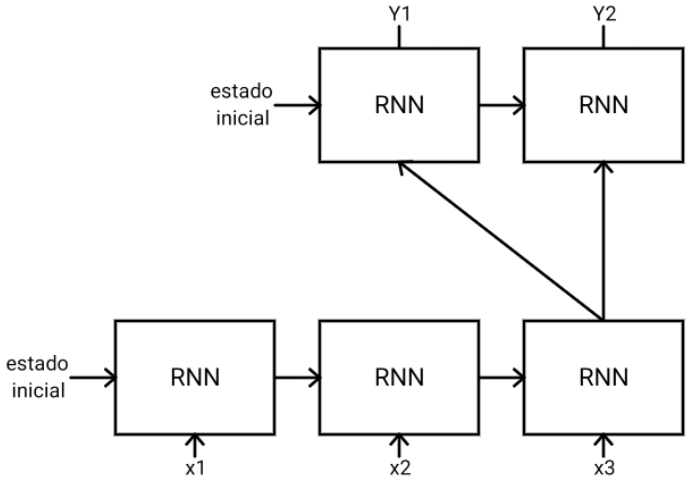

Figura 2.7: Uma arquitetura de *Encoder-Decoder*. A RNN inferior codifica a entrada completa em uma representação após o último passo, que é então usada pelo decodificador para produzir a série de saídas do tamanho desejado.

principal problema é o mencionado desaparecimento do gradiente e o seu oposto, a explosão do gradiente quando várias etapas com valores maiores que 1 causam o gradiente a crescer exponencialmente através das camadas.

Dessa forma, as camadas inferiores (passos mais antigos) tenderiam, graças a múltiplas instâncias do problema acima, a receber um gradiente local próximo de zero (ou alto demais), de forma que os pesos teriam ajustes muito pouco significativo, independente da taxa de aprendizado, para minimização do erro global. Ou seja, a rede recorrente na prática só é capaz de propriamente ajustar pesos minimizando os erros dos pares (X_t,Y_t) mais recentes nas listas de entrada e saída.

Uma solução proposta foi acoplar uma memória à RNN, de forma que a mesma se tornasse capaz de, através dos mesmos mecanismos de aprendizado, escolher e armazenar informações da entrada importante na memória, escolher e deletar informações não mais relevantes da memória e escolher e ler informações da memória para inserir no estado oculto e na saída. Esse modelo é a *Long Short-Term Memory* (LSTM) [21].

Em uma LSTM, a camada que se repete ao longo do tempo possui três módulos que trabalham em conjunto para gerar o novo estado oculto, em vez de gerá-la em um única etapa, como na RNN discutida acima. Cada um desses módulos tem uma função especial no processo de decidir quais informações devem ser mantidas, quais devem ser transmitidas para as seguintes etapas e quais informações devem ser descartadas, utilizando para tal informações usuais provenientes da entrada, da saída da camadas oculta anterior e da chamada *Célula de Memória*. Esses módulos

são chamados de portas (*gates*). Um *gate* é composto por uma função de ativação sigmóide e uma operação de multiplicação elemento a elemento (operação também conhecida de forma geral como produto de Hadamard). Além dos *gates*, as LSTMs também empregam uma camada oculta regular de uma MLP que recebe a concatenação da entrada atual com o estado oculto anterior, assim como as redes neurais recorrentes originais.

O primeiro *gate*, o *forget gate*, tem a responsabilidade de decidir quais elementos da célula de memória serão jogadas fora e quais poderão continuar. Ele realiza essa tarefa produzindo um vetor de valores entre zero e um e multiplicando a memória por esses valores, de forma que aqueles valores multiplicados por números próximos de zero são esquecidos, e aqueles multiplicados por valores próximos a um são mantidos. Essa tarefa é realizada, a cada passo, a partir do estado oculto anterior da rede e da entrada atual. A equação do *forget gate* é dada, portanto, por 2.7:

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f) \tag{2.7}$$

Em que $[h_{t-1}, x_t]$ é a concatenação do estado oculto do passo anterior e a entrada atual, e substitui a entrada x na transformação afim usada para a MLP. W_f e b_f são os parâmetros a serem aprendidos para o *forget gate*, e σ a função de ativação logística que "achata" os valores para [0,1].

Depois de esquecer dados julgados irrelevantes, a rede então gera um novo estado para a memória, processo conhecido como criação do candidato $\widetilde{C}_t$ da memória, usando uma função afim e uma função de ativação da mesma forma que a RNN tradicional conforme a equação 2.8, que utiliza tipicamente a tangente hiperbólica ($tanh$) como ativação:

$$\widetilde{C}_t = tanh(W_C.[h_{t-1}, x_t] + b_C) \tag{2.8}$$

Em que W_C e b_C são os pesos e o viés utilizados para gerar o novo candidato a memória $\widetilde{C}_t$. Tendo, então, um novo candidato a memória, entra em funcionamento o segundo *gate*, o *input gate*:

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i) \tag{2.9}$$

Que também possui seu conjunto de parâmetros W_i e b_i, e também gera um vetor de valores entre [0,1], só que o mesmo será multiplicado ao candidato da nova memória $\widetilde{C}_t$, de forma a selecionar quais campos vão ser guardados na memória e quais serão imediatamente esquecidos. Esse novo candidato a memória é então somado à célula de memória após ter tido campos apagados pelo *forget gate*, segundo a equação 2.10:

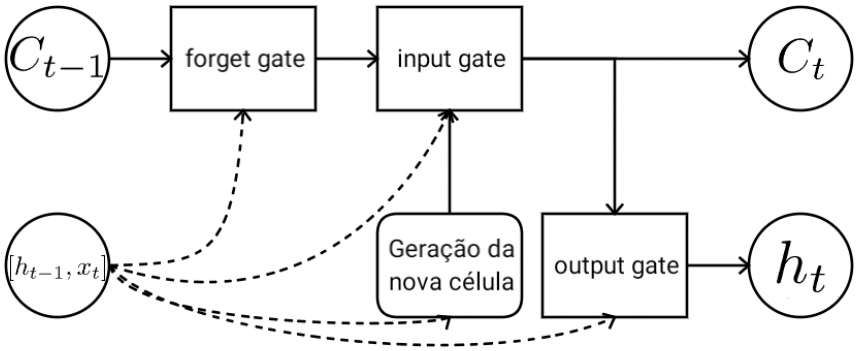

Figura 2.8: A *Long Short Term Memory*. Linhas pontilhadas são as entradas das portas (*gates*). O *forget gate* apaga informações da célula de memória prévia, o *input gate* combina o candidato a novo estado com o estado prévio e o *update gate* gera o estado oculto a partir da célula de memória atual.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{2.10}$$

Em que o primeiro operando $f_t * C_{t-1}$ representa o processo de esquecimento de dados não mais pertinentes da memória e o segundo operando $i_t * \widetilde{C}_t$ a introdução de novos valores na memória. Com isso gera-se a memória do passo t, C_t.

Finalmente, constrói-se o novo estado oculto e a nova saída, que são o mesmo vetor para a LSTM, e para tanto usamos o quarto *gate*, o *output gate*, dado pela equação 2.11

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o) \tag{2.11}$$

Em que W_o e b_o são mais um conjunto independente de pesos e viés para o modelo. O vetor o_t é então multiplicado pela memória atual para gerar a saída e estado oculto h_t, selecionando quais informações na memória devem ser usados para gerar a saída desse passo, conforme a equação 2.12

$$h_t = o_t * tanh\,(C_t) \tag{2.12}$$

O resultado de todo esse processo é um novo valor para a célula de memória e um novo estado oculto, ambos podendo ser usados pelo próximo passo da rede recorrente juntamente com a nova entrada, e uma saída que pode ser usada diretamente para o erro ou predição, ou ligada a uma nova camada para realizar mais processamentos. Um diagrama simplificado está representado na figura 2.8.

2.1.5 Gated Recurrent Unit (GRU)

LSTMs tem sido a rede neural recorrente mais utilizada desde a sua criação, devido ao seu consistente sucesso em diversos domínios que requerem lidar com sequências de dados. No entanto, um modelo mais simples foi criado recentemente, a Gated Recurrent Unit (GRU) [22, 23] capaz de produzir resultados similares e portanto vem ganhando popularidade.

Como as LSTMs, GRUs também têm *gates* para decidir qual informação será lembrada e qual informação será esquecida. No entanto, GRUs só tem dois desses módulos, combinando a *input gate* e a *forget gate* em um único *update gate*. O outro *gate* é chamado *reset gate*. O estado oculto e a memória são representados por um único vetor na GRU também.

Os *gates* da GRU também são compostos por uma função afim com uma matriz de pesos e um viés e uma função de ativação sigmóide gerando valores de [0,1], tal como nas equações seguintes:

$$r_t = \sigma(W_r.[h_{t-1}, x_t] + b_r) \tag{2.13}$$

$$u_t = \sigma(W_u.[h_{t-1}, x_t] + b_u) \tag{2.14}$$

Em que W_r e W_u são os pesos do *reset* e *update gates*, respectivamente, b_r e b_u são os vieses do *reset* e *update gates*, respectivamente e $[h_{t-1}, x_t]$ é a concatenação do estado oculto prévio e o valor de entrada atual.

Quando os valores calculados pelo *reset gate* são próximos de zero, os respectivos campos do estado oculto também são zerado e sua informação é ignorada para efeitos de criação do novo estado oculto. Isso é feito modificando, em comparação a LSTM, o módulo de criação do candidato a novo estado oculto $\widetilde{h}_t$ para receber o estado oculto já multiplicado pelo *reset gate*, de forma que os campos "resetados"nesse passo não estarão presentes no novo estado gerado. A seguir essa equação do candidato a novo estado oculto para a GRU:

$$\widetilde{h}_t = \tanh(W.[r_t * h_{t-1}, x_t] + b) \tag{2.15}$$

O *update gate*, por sua vez, controla o quanto de informação do estado oculto prévio deve ser mantido sobre o candidato a novo estado oculto, ajudando na manutenção de informação de longo prazo, ao permitir que partes do estado oculto sejam mantidas inalteradas por um número de passos indefinido. O valor do estado oculto atual é computado como a equação 2.16 mostra, onde r_t e u_t são o *reset* e *update gates* calculados acima.

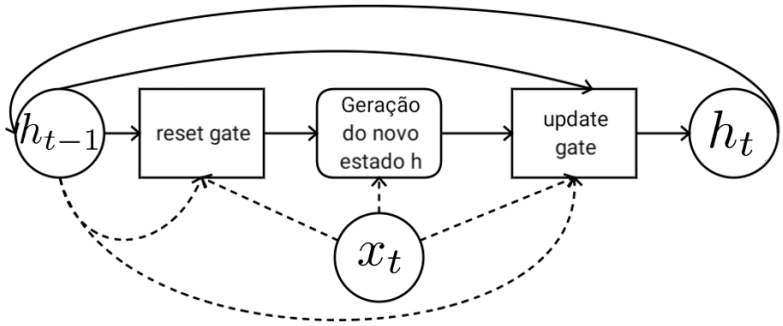

Figura 2.9: A *Gated Recurrent Unit*. Linhas pontilhadas são as entradas das portas (*gates*), linhas normais são o percurso do estado oculto.

$$h_t = (1 - u_t) * h_{t-1} + u_t * \tilde{h}_t \tag{2.16}$$

Onde o operando $(1 - u_t) * h_{t-1}$ seleciona as partes do estado oculto prévio a se manter, e $u_t * \tilde{h}_t$ as partes do candidato a novo estado oculto a se guardar em seu lugar.

O processo pode ser observado na figura 2.9, em que o estado oculto é filtrado pelo *reset gate*, selecionando os dados que devem ser processados nesse passo, que é utilizado junto com a entrada para gerar o novo estado. O *update gate* seleciona o que vai guardar do novo estado oculto e o que será reutilizado do estado passado. O atalho entre o antigo estado oculto e o novo estado oculto sem passar por nenhuma função de ativação, produzido pelo *update gate*, é o que permite a rede lembrar fatos antigos e evitar o problema do desaparecimento do gradiente.

2.1.6 Aprendizado Profundo

A área popularmente conhecida como *Aprendizado Profundo* ou *Deep Learning* representou uma quebra de paradigma apesar de usar essencialmente as mesmas técnicas utilizadas acima. O grande motor desse processo foi o crescimento no poder de processamento, na qualidade e tamanho das bases de dados e na melhora das ferramentas, em particular no renascimento da técnica de diferenciação automática. Através dela, basta o pesquisador definir as variáveis que devem ser treinadas e um conjunto de transformações que gera a saída (o *forward pass*), que as ferramentas transformam o modelo em um grafo de operações básicas (somas, produtos, sigmóides) e tendo implementado apenas a derivada (aproximada) de cada operação calculam a regra da cadeia através de todos os caminhos do grafo entre a saída e cada parâmetro. Dessa forma o pesquisador fica livre para testar qualquer modelo sem ter de se preocupar em escrever o processo de *backward pass* envolvendo o gradiente, facilitando muito o desenvolvimento de modelos menos convencionais.

A maior parte da teoria em Aprendizado de Máquina, como por exemplo a conclusão da dimensão de Vapnik–Chervonenkis (VC dimension) [24], prevê que modelos com muitos parâmetros têm uma capacidade tão grande de aprender a base de dados que chega ao ponto de memorizar as respostas todas em vez de aprender regras ocultas mais simples, e portanto não seriam capazes de generalizar para amostras não presentes no treino. Esse é o conceito conhecido como *sobreajuste* (*overfitting*), e foi um dos motivadores para o uso de modelos mais simples, como separadores lineares que são naturalmente incapazes de perfeitamente separar todas as amostras em bases de dados não triviais, em vez de redes neurais.

Apesar disso, resultados práticos recentes contrariaram a crença, com as redes neurais mais complexas sendo capazes de generalizar em muitos casos de forma melhor que os modelos mais simples. Isso não contraria a literatura no entanto, pois esses são baseados no conceito em probabilidade de desigualdades, que prevêem um limite superior à capacidade de generalização (a diferença entre o erro na base de treino e no mundo real, estimado pela base de testes), o que não diz que modelos complexos não são capazes de generalizar, mas apenas que a confiança em sua capacidade de generalizar é menor dado apenas alguns fatores comuns entre todos os modelos possíveis [25].

Existem muitas tentativas de criar limiares para essas desigualdades mais próximos da realidade prática, porém as hipóteses disponíveis ainda requerem mais trabalho. Enquanto os teóricos ainda estão em busca de explicações, os resultados práticos foram suficientes para mover uma enorme quantidade de recursos humanos e financeiros para área, que se desenvolveu mais em poucos anos que em décadas.

O problema do sobreajuste ainda é real e comum para Redes Neurais, no entanto. São utilizadas também formas de reduzir os graus de liberdade dos pesos das redes neurais, adicionando critérios que restringem o valor combinado dos pesos (como a regularização L2, ou *weight decay*), ou critérios que garantam a esparsidade (como a regularização L1) ou mesmo mecanismos que, durante o treino, removem partes da rede para que nenhum módulo seja capaz de plenamente depender do outro (*Dropout* [26]). A técnica mais comum no entanto é manter uma base de dados de validação separada da base de treino e a cada época, ou número fixo de épocas, testa-se quão bem o modelo prediz a base de dados de validação (sem atualizar os pesos), e caso piore a qualidade da predição finaliza-se o treino, pois a rede já está sobreajustando para o treino. A distribuição entre treino e validação é uma escolha do pesquisador, tendo inclusive técnicas para melhor uso dos dados, incluindo fazer médias com diferentes grupos de validação e treino, como a validação cruzada. E possivelmente mais importante de tudo, muitos dos problemas que só modelos de aprendizado profundo conseguem tratar, tal como dirigir carros a partir da entrada de câmeras, recebem bases de dados tão grandes e os dados são tão complexos que

simplesmente memorizar os casos de treino estão além da capacidade de aprendizado, até de modelos que seriam considerados demasiadamente complexos no passado.

Existindo no mercado Redes Neurais com números massivos de parâmetros, algumas vezes nas casas de bilhões, capazes de superar o problema do sobreajuste, a tendência foi a criação de redes progressivamente mais longas tais como [27] e que realizam progressivamente maior parte do processamento. Originalmente um especialista teria de selecionar os atributos (*features*) importantes para alimentar o modelo. Por exemplo, palavras em uma frase seriam representadas por regras que permitem diferenciá-las, tais como se a palavra começa com maiúscula, quantas letras possui, se possui algum número, se acaba com "s"(indicando pluralidade), etc. Dessa forma os modelos fariam uma estimativa da importância de cada atributo para definir cada classe, por exemplo começar com maiúscula teria um peso grande para definir nomes próprios. Atualmente, no entanto, deixa-se a rede receber as palavras diretamente (ou convertida previamente por uma outra rede neural para uma representação mais compacta) e decidir por conta própria o que é importante para cada palavra, ou seja, automatizando a construção e extração dos atributos.

Essa evolução causou os pesquisadores atuais a favorecerem os chamados modelos treinados ponta a ponta (*end-to-end*), em que se recebe dados com o mínimo possível de processamento (e nenhum processamento baseado em seu significado), apenas o par de entradas e saídas (tais como lista de imagens de animais e o nome dos animais, respectivamente).

Da mesma forma que a rede cria autonomamente uma representação interna (conjunto de atributos relevantes) das entradas para poder trabalhar, cada camada cria novas representações de suas respectivas entradas, de forma que ao longo da rede as representações se tornam cada vez mais próximas do domínio da pergunta final. No exemplo comum de reconhecimento de faces, a primeira camada poderia aprender formas simples como bordas e listras a partir de píxeis, a segunda camada poderia juntar essas bordas em formas geométricas simples tais como circunferências e quadrados, a terceira poderia juntar formas geométricas simples em mais complexas tais como olhos e narizes, a quarta poderia juntar as partes da face em face e finalmente pode-se ter um classificador que trabalha diretamente com separação de faces, o que seria complicado de analisar diretamente a partir de conjuntos de píxeis.

Apesar de uma rede neural densa de uma camada ser considerada como aproximadora universal, a criação de representações intermediárias se mostra como uma técnica que melhora o resultado em muitos casos. Porém, redes demasiadamente profundas ainda sofrem de problemas para treinar, tais como o desaparecimento ou explosão do gradiente, além de não estar claro o número ótimo de camadas, ou graus de abstração, para resolver cada problema mais eficientemente.

2.1.7 Representações Distribuídas, Transferência de Aprendizado e *Softmax*

Foi mencionado na subseção anterior que palavras tradicionalmente eram representadas por atributos definidos por especialistas e posteriormente foram substituídas por representações aprendidas. Portanto, para redes neurais que só conseguem processar números precisa-se de uma forma de representar palavras em números sem que se adicione qualquer juízo de valor. Para redes neurais, contínuas, simplesmente numerar as palavras pela ordem do dicionário teria um efeito sobre o processamento, pois a palavra de número x e a palavra de número $x + 1$ são tão próxima em um contínuo de possivelmente centenas de milhares de palavras que seriam consideradas como praticamente iguais na rede, independente de seu significado ser ou não ser quase igual.

Portanto, a única forma de não adicionar um juízo é ter uma entrada tão grande quanto o número de palavras no dicionário, ou seja, se a rede tiver cem mil palavras que seja capaz de processar, então sua entrada deve ser um vetor de 100 mil elementos que será um para a palavra alvo e zero para todo resto (e dessa forma cada palavra é tangente a todas as demais palavras). Essa representação é chamada de *one-hot*. O problema claro é que em um *Multi-Layer Perceptron*, isso significa que para cada neurônio na primeira camada oculta será necessário cem mil parâmetros, de modo que a rede precisará de um conjunto grande de amostras apenas para treinar essa camada, antes mesmo de poder criar as representações mais complexas.

Um dos grandes possibilitadores do processamento de texto foi o algoritmo conhecido como *word2vec* [4]. O *word2vec* é uma rede neural densa simples de uma camada oculta que resolve um problema que apesar de supervisionado não precisa de entradas manualmente etiquetadas. Em uma de suas duas modalidades originais, chamada *Continuous Bag-of-Words* (CBOW), a partir de texto bruto, a entrada é composta por n palavras vizinhas a uma palavra central, e tenta-se prever a tal palavra central na saída. Esse modelo pode ser observado na figura 2.10. A intuição é de que a camada oculta precisa representar o contexto de cada palavra de entrada para que a camada de saída possa adequadamente selecionar a palavra mais provável para um contexto, ou seja, palavras que estão frequentemente no mesmo contexto terão uma representação similar na camada oculta pois elas frequentemente levam a camada de saída a prever a mesma palavra. Por exemplo palavras como "pizza"e "bolo"frequentemente estão próximas das palavras "comida"e "deliciosa"em frases como "a pizza é minha comida favorita"ou "o bolo estava delicioso", enquanto a palavra "circuito"dificilmente aparecerá junto com essas palavras já que é difícil de imaginar alguém descrevendo circuito como delicioso. Logo a representação de "pizza"e "bolo"serão similares entre si pois a camada de saída precisa prever pala-

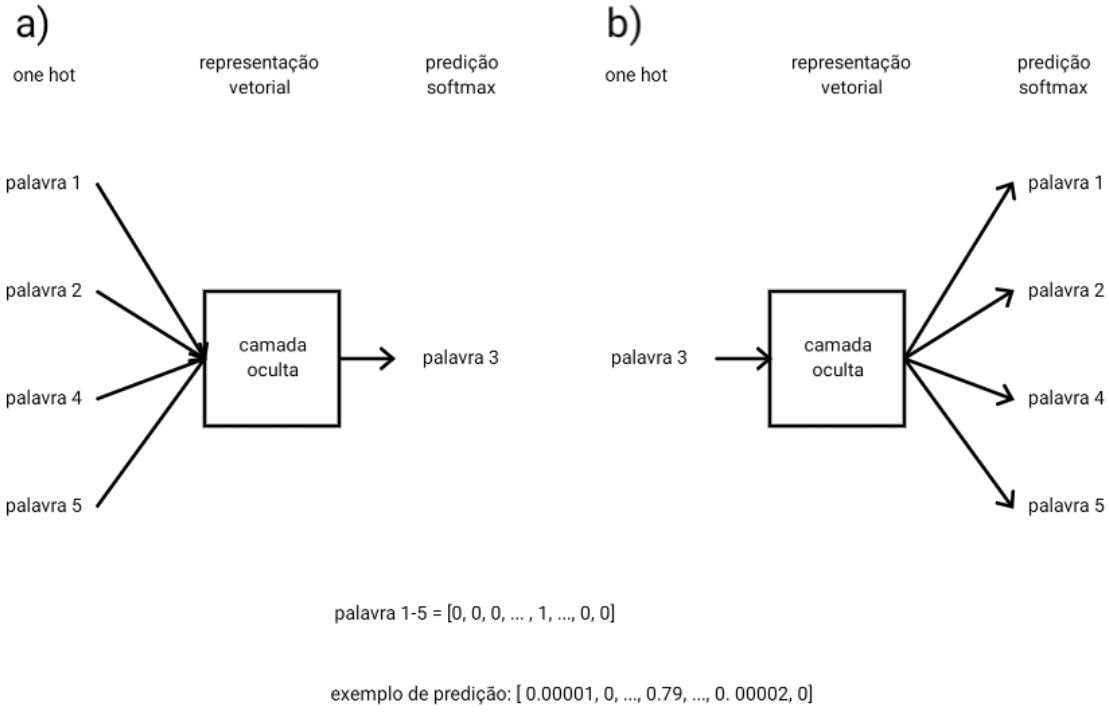

Figura 2.10: A estrutura do *word2vec* na modalidade de a) *Continuous Bag-of-Words* e b) *Skip-Gram*. As entradas e a saída estão codificadas em *one-hot*, e a predição *softmax* tenta aproximá-la gerando um vetor de probabilidades normalizado em um para cada palavra do dicionário. Os pesos que transformam as palavras em *one-hot* na camada oculta, que é uma representação simplificada que é usada pelo classificador na última camada, são reaproveitados para outros domínios.

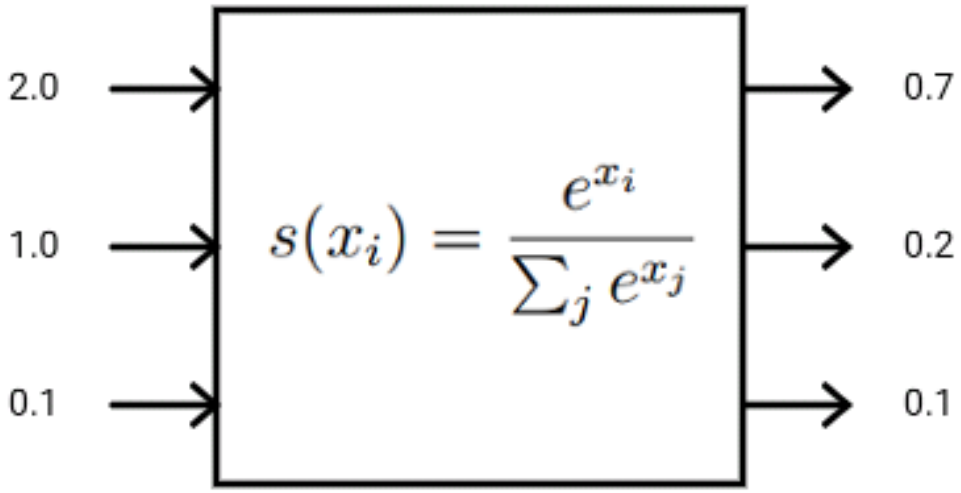

Figura 2.11: Uma operação de *softmax* que calcula uma distribuição de probabilidade entre três possíveis classes.

vras similares, enquanto "circuito"será bem diferente para não prever quase nenhuma palavra em comum.

A outra modalidade funcionalmente similar do *word2vec*, também representada na figura 2.10, é o *Skip-Gram* que possui apenas uma palavra na entrada e tenta-se prever todas as palavras em volta. CBOW é mais rápido para treinar e tem melhor resultados para palavras comuns, enquanto *Skip-Gram* consegue representar palavras raras melhor.

Definindo o tamanho da camada oculta, define-se então o número de atributos aprendidos para cada palavra, de forma que inclusive pode-se aprender múltiplos contextos que a palavra apresentou no texto (como contexto de gênero, contexto de número, contexto dos sentimentos positivos, negativos ou neutros além do contexto principal da semântica).

Note que a camada de saída tem de escolher uma palavra entre todas as palavras do dicionário, o que é feito através de uma técnica chamada regressão multinomial, e uma camada chamada *softmax*. Essa camada faz uma regressão multinomial para cada palavra do dicionário, prevendo a probabilidade de cada uma dessas palavras e normalizando em função de todas as palavras (o somatório da probabilidade de cada palavra deve ser um). Essa técnica é exemplificada na figura 2.11. Calculando a potência de cada número garante que o valor sempre será positivo, e o denominador garante que a soma de todas as saídas será zero.

Além do problema do tamanho da rede, já que existe uma saída para cada palavra, e cada saída conectada via pesos independentes a todos os neurônios da

camada oculta anterior, a normalização faz com que cada palavra seja uma função de todas as palavras do dicionário. Esse processo final é extremamente ineficiente para treinar com dicionários grandes, com centenas de milhares de palavras, e o que garantiu a viabilidade do *word2vec* foram métodos mais eficientes de realizar essa operação. A mais simples delas é conhecida como *Negative Sampling* [28] que envolve treinar o modelo para detectar para cada entrada tanto amostras positivas (palavras realmente no contexto da entrada) quanto negativas (palavras tipicamente fora do contexto da entrada) geradas pelo modelo, e atualizar apenas as saídas nessas amostras em vez de todas as palavras no dicionário para cada entrada. Isso se torna muito mais eficiente se o número de amostras falsas for consideravelmente menor que o tamanho do dicionário.

Para os modelos que requerem processar palavras utiliza-se os pesos que transformam a representação *one-hot* para a camada oculta do modelo do *word2vec* para criar uma representação compacta da palavra com atributos baseados na semântica contextual da palavra, reduzindo de centenas de milhares de campos se for processar qualquer palavra do dicionário para tipicamente algumas centenas de campos da entrada.

Treinada a rede *word2vec*, para se obter então uma representação compacta (apenas algumas centenas de campos) com informação contextual de uma palavra, basta inseri-la na camada de entrada codificada em *one-hot* e extrair os valores dos neurônios da camada oculta. Esse valor extraído é a chamada *Representação Distribuída*. Com isso o treino da rede neural a ser aplicada no domínio final pode ocorrer mais eficientemente pois palavras similares, incluindo sinônimos, podem ser treinadas juntas na rede por possuírem uma representação bastante similar.

Alternativamente, pode-se continuar recebendo uma representação *one-hot* e utilizar os pesos da rede do *word2vec* como valores iniciais do peso, em uma espécie de pré-treino, de forma que a rede já comece com resultados bons na criação de representações para palavras, mas seja capaz de melhorar essa representação com as amostras usadas para o domínio real da aplicação. Esse processo de utilizar o aprendizado em um problema ou domínio e usar como base para acelerar ou melhorar o aprendizado em outro problema ou domínio é conhecido como *Transferência de Aprendizado* (*Transfer Learning*) [29], e é uma área de bastante foco na atualidade.

2.1.8 Mecanismos de Atenção e Seleção

A função *softmax* permite redes neurais fazer seleções entre diversas opções, e pode ser aplicada a outras seleções além de múltiplas categorias na saída. Uma aplicação foi um aprimoramento do modelo *Encoder-Decoder* apresentado anteriormente. No *Encoder-Decoder* o codificador é responsável por criar uma representação de toda a

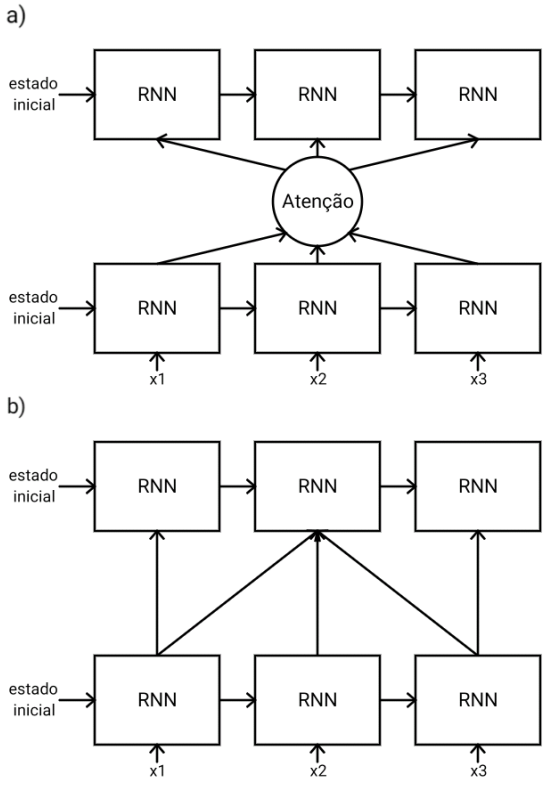

Figura 2.12: a) o *Encoder-Decoder* utilizando um mecanismo de atenção, em que é calculada uma distribuição através de uma operação *softmax*, que é então multiplicada às entradas permitindo que a rede aprenda a selecionar as entradas que precisa usar. b) Um exemplo, com o primeiro e último passo do decodificador lendo apenas os dados do primeiro e último passo do decodificador respectivamente, e o segundo passo lendo parte de todos os 3 passos

entrada em um único vetor de tamanho fixo, o que se torna problemático quando as entradas possuem tamanhos variáveis. Frases muito longas, especialmente mais longas que aquelas com a qual a rede foi treinada, são difíceis de codificar propriamente nesse modelo.

Uma melhoria seria permitir que o decodificador selecione quais resultados parciais do codificador são relevantes para cada tarefa. Como exemplo, traduzindo "Gosto de pizza"para "I like pizza", para gerar "I"a única palavra relevante seria "Gosto", para "like"novamente apenas "Gosto"possui toda a informação necessária, e "pizza"em inglês só depende de "pizza"em português. Dessa forma o codificador não precisa mais ser capaz de representar toda a informação da frase em apenas um vetor fixo final, e o decodificador pode trabalhar com informação mais relevante para sua tarefa imediata a cada passo. Esse exemplo é demonstrado na figura 2.12.

Esse mecanismo foi inspirado nos *gates* do LSTM. Tipicamente, usa-se duas redes recorrentes no codificador, uma lendo normal e outra de trás para frente e

concatenando suas saídas, de forma que sempre se tenha informação completa da frase, mas com a palavra que se deseja traduzir tendo sido processada sempre por último. Esse mecanismo que envolve aprender a selecionar uma entre várias saídas a cada passo se chama de atenção, e o modelo citado é comumente chamado de *Encoder-Decoder with Attention* [20]

Outra aplicação do mecanismo de atenção envolve outra deficiência das LSTMs: apesar de possuir memória e ser capaz de lidar com informações de passos mais distante no passado que as RNNs tradicionais, essa memória é normalmente muito pequena e pouco compartimentada. Aumentar essa memória significa aumentar a camada oculta e o número de pesos de todos os *gates*, o que é ineficiente. Para desacoplar o tamanho da memória do número de parâmetros da rede, passou-se a utilizar o mecanismo de atenção para selecionar campos para leitura e escrita em memórias externas à LSTM. O modelo mais famoso a implementar esse acesso a memórias externas foi a *Neural Turing Machine* (NTM) [30].

A NTM utiliza uma memória externa com a forma similar à memória RAM, ou seja uma lista de elementos diretamente endereçáveis. Para o acesso cria-se uma distribuição normalizada, e aplica-se uma série de transformações, em particular para criar uma nova distribuição para acesso direto ou para utilizar uma distribuição antiga deslocada para esquerda ou para a direita para simular acesso em listas encadeadas ou vetores em linguagem de programação. Com isso, essas redes são capazes de inferir algoritmos simples tais como copiar, ordenar, e também responder perguntas simples de associação com informação passada.

Uma alternativa aos NTMs, são as *Stack-RNN*s [31] que, ao invés de uma memória contígua com acesso direto, utiliza uma pilha, em que o gerenciamento se dá através de três operações, PUSH, POP e NO-OP, operando como um autômato de pilha, e teoricamente obtendo a expressividade de uma máquina de Turing com múltiplas pilhas a custo de ter de aprender mais complexas interações entre pilhas.

As *Stack-RNN*s já utilizam a função *softmax* para selecionar uma dentre múltiplas operações, de forma similar ao mecanismo dos programadores neurais como o apresentado nessa dissertação. No entanto, a própria rede neural faz o processamento dos dados, assim como na LSTM e NTM, em vez de construir algoritmos explícitos em uma máquina virtual como nos programadores neurais.

2.1.9 Limitações

Métodos de aprendizado usando redes neurais, especialmente, mas não somente, sobre a filosofia do aprendizado profundo [32] têm emergido como estado da arte em um grande número de tarefas. Porém, as arquiteturas de redes neurais mais usadas ainda são caixas pretas, no sentido que ainda é difícil entender a lógica interna que

aprendem e seguem para tomar cada decisão. Por exemplo, estudos como [33] tem mostrado que adicionar leves ruídos específicos, imperceptíveis para seres humanos, em fotos pode causar significativas mudanças em como a rede classifica a entrada. Essa falta de segurança pode impedir a aplicação dos modelos em tarefas em que prestações de conta são uma necessidade [34].

Outra limitação é que, sem observabilidade, a manipulação direta da lógica da rede não é possível. Portanto, a adição de um conhecimento preexistente a uma rede cujo aprendizado é um mistério é uma tarefa muito difícil, com a maioria das tentativas focando no pré-treino em um domínio diferente porém relacionado, transferindo assim parte desse conhecimento para a tarefa presente [35, 36]. O uso dessas técnicas de transferência de conhecimento pode acelerar o processo de aprendizado e a qualidade dos modelos finais, mesmo na presença de poucas amostras para o treino. De qualquer forma, modelos possuindo maior transparência permitiriam formas mais simples e diretas de transferir, e portanto, adicionar conhecimento prévio.

Uma abordagem que melhora a transparência de redes neurais, e permitem a elas enfrentar problemas mais difíceis, incluindo aqueles envolvendo aritmética complexa e raciocínio lógico, é o do programador diferenciável neural [10, 37]. Esse tipo de rede, em vez de aplicar transformações diretamente na entrada, escolhe uma sequência de transformações de um conjunto predefinido, produzindo um algoritmo para resolver o problema. Dessa forma, conseguem produzir um raciocínio observável para cada inferência. E como se gera um algoritmo com suas próprias entradas de dados independentes da parte neural da rede, permite-se produzir um raciocínio mais genérico em relação aos dados sendo processados. Isso torna o modelo mais próximo da programação tradicional, em que é uma decisão explícita quais dados de entrada podem afetar qual código será executado, normalmente feito através de testes condicionais.

Contudo, modelos atuais de programadores neurais focam em soluções ponta a ponta para problemas e contextos específicos, em vez de criar uma ferramenta que pode ser integrada diretamente no contexto de redes neurais de muitas camadas. Caso contrário, elas permitiriam a fácil coexistência de raciocínio transparente e explícito e a capacidade de reconhecimento de padrões e aprendizado das tradicionais redes neurais em caixa preta.

Essa dissertação propõe uma técnica de construção de uma máquina virtual diferenciável em rede neural recorrente que pode ser facilmente integrada em qualquer modelo que usa uma rede neural recorrente, permitindo então potencialmente resolver os problemas acima ao permitir que modelos possam integrar elementos de lógica e aritmética customizadas para cada aplicação.

2.2 Lógica Difusa

Um outro aspecto pertinente à nossa máquina virtual diferenciável em particular é a teoria da *Lógica Difusa* (*Fuzzy Logic* [38]). A lógica Booleana considera apenas duas possibilidades, verdadeiro ou falso, um ou zero. Ou seja, o sistema não é definido para nenhum valor exceto um e zero, tornando todos os operadores não diferenciáveis em toda sua extensão.

A lógica difusa tem sua principal aplicação em sistemas de controle, tais como sistemas que envolvem estados subjetivos tais como a percepção da temperatura ambiente. Na lógica Booleana, ou o ambiente está quente ou está frio, e a mudança é instantânea (por exemplo, sendo 25° o limiar, $24,9°$ seria frio e $25,1°$ seria quente apesar de subjetivamente não fazer diferença). A lógica difusa, no entanto, trabalha com graus de verdade, ou pertinência. Por exemplo, só seria totalmente quente em 30° e só é totalmente frio em 15°, e todos os estados intermediários seriam uma combinação dos múltiplos estados, ou seja meio quente e meio frio.

O interesse particular dessa dissertação é na lógica difusa de norma triangular (ou lógica difusa de t-norma). T-norma diz respeito à generalização da conjunção (o operador "E"ou "AND") na lógica Booleana para o espaço contínuo de [0,1]. A T-norma possui seu par na conorma triangular (*T-conorm*), a generalização da disjunção (o operador "OU"ou "OR"). Existem diversas formas que essas generalizações podem ser descritas, como por exemplo:

$$x \text{ E } y = x \wedge y = x * y \tag{2.17}$$

$$x \text{ OU } y = x \vee y = x + y - x * y \tag{2.18}$$

$$\text{NÃO } x = \neg x = 1 - x \tag{2.19}$$

Essa conjunção se chama norma triangular de produto (*product t-norm*) e sua dual para disjunção se chama soma probabilística (*probabilistic sum*). Uma alternativa é a norma triangular mínima (*minimum t-norm*) para conjunção e a conorma triangular máxima (*maximum t-conorm*) para a disjunção como nas equações abaixo.

$$x \wedge y = \text{MIN}(x, y) \tag{2.20}$$

$$x \vee y = \text{MAX}(x, y) \tag{2.21}$$

$$\neg x = 1 - x \tag{2.22}$$

Pode-se facilmente ver que em ambos os modelos os casos bases da lógica booleana são válidos substituindo x e y pelos valores um e zero e comparando o resultado com a tabela verdade das operações booleanas. A conversão entre a respectiva t-norma para t-conorma e vice-versa usando a negação, a generalização das leis de

De Morgan, também é possível, assim como as diversas outras propriedades dos operadores lógicos booleanos.

A partir desses operadores lógicos difusos é possível implementar qualquer sistema proposicional de forma diferenciável.

Capítulo 3

Máquinas Virtuais e Redes Neurais

3.1 Problema

Considere o simples pseudo-código:

Exemplo 1 $V \leftarrow account.current//50$
$V \leftarrow V + 20$
for $i = 1$ *to* 2 ***do***
$V \leftarrow V - 30$
end for
$PRINT(V)$

Assumindo que tal pseudocódigo tenha uma sintaxe similar às linguagens de programação tradicionais, não seria difícil construir um interpretador capaz de ler tal código e imprimir o resultado esperado, no caso 10. Agora considere um outro exemplo:

Exemplo 2 *Pegue o valor da minha conta, ponha 20 reais de crédito e compre com ela 2 produtos de 30 reais. Quanto sobra?*

Não é difícil ver que o novo problema e o anterior são resolvidos através do exato mesmo algoritmo, mas agora foi escrita em uma linguagem muito mais complexa (não mais representável por uma gramática livre de contexto considerando a hierarquia de Chomsky [39]). Essa nova língua possui um vocabulário muito mais amplo (potencialmente todo o dicionário da língua deveria ter de ser aceito, incluindo potenciais erros ortográficos em ambientes práticos), é carregada de contexto implícito (o que significa crédito ou qual o efeito de "compre"sobre o cálculo? Onde encontramos a informação de "minha conta"?), possui complexas correferências (O produto

custa cada um 30 reais e consiste de duas unidades) e possíveis ambiguidades. Não é mais possível construir um interpretador convencional capaz de consistentemente produzir a resposta correta para a infinitude de formas de representar o mesmo algoritmo.

De fato, não só uma interpretação de texto em língua natural do nível de uma pessoa comum é ainda um sonho possivelmente distante para os atuais algoritmos e técnicas, o problema é possivelmente IA-completo [40], o termo usado para uma tarefa que para resolver requer um modelo tão inteligente quanto o próprio ser humano. Porém, as Redes Neurais, especialmente as recorrentes, vêm conseguido resultados significativos quando se restringe o domínio, como análise de sentimentos, sumarização, tradução e respostas a perguntas [41] mesmo que extrair o sentido em sua totalidade ainda seja impensável.

O grande potencial das redes neurais é trabalhar com dados de entrada sem um formato restrito, rígido e predefinido. Enquanto isso, os interpretadores e compiladores têm uma capacidade enorme já presente através de todas as bibliotecas para facilmente lidar com todo tipo de problema complexo. A natureza de caixa-preta das redes neurais demanda delas aprender sempre do zero, e para o problema acima a rede não só precisa aprender a interpretar o texto, ela também precisa aprender todas as operações presentes no algoritmo, como soma, produtos e até gerenciamento da memória e como elas devem ser integradas para resolver o problema. Para dificultar ainda mais, mesmo operações básicas, tais como soma e subtração, são difíceis para as redes neurais aprenderem mesmo separadamente com a precisão de uma calculadora simples.

Através das redes recorrentes e dos mecanismos de atenção e de seleção, recentemente começaram a surgir as primeiras formas de bibliotecas (listas de operações ou algoritmos já desenvolvidas para tarefas comuns) capazes de rodar no mesmo ambiente das redes neurais. Dessa forma, cria-se a possibilidade de que desenvolvedores de redes neurais sejam capazes de utilizar e compartilhar bibliotecas que adicionem novas funcionalidades aos seus modelos da mesma forma que interpretadores e compiladores fazem desde a sua origem. Assim, podem surgir modelos que aprendem apenas as peculiaridades do seu domínio, a partir da escolha de subrotinas preparadas para tarefas comuns.

3.2 Estrutura Geral

A figura 3.1 mostra a estrutura geral utilizada para integrar redes neurais e máquinas virtuais, com no mínimo o codificador da entrada sendo neural. Só isso permitiria que o comando de entrada seja qualquer entidade que uma rede neural possa processar, ou seja, qualquer entidade que possa ser codificada em um tensor (qualquer

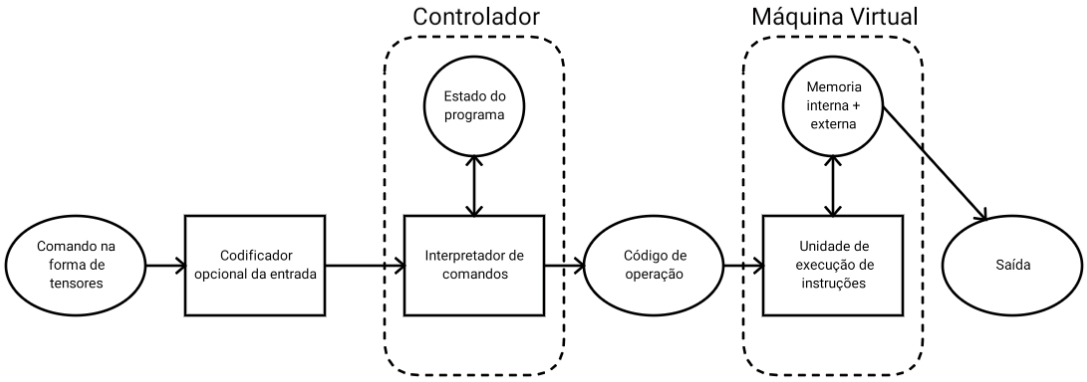

Figura 3.1: Estrutura geral da integração entre máquinas virtuais e redes neurais. O codificador transforma o comando em uma representação vetorial. O controlador define a instrução a ser executada para cada comando através do código de operação. A máquina virtual executa a instrução determinada pelo código de operação, salvando o resultado em sua memória para os passos seguintes ou para a saída.

array de números com n dimensões/índices) ou lista de tensores. Isso inclui:

- Texto: Normalmente codificado como uma lista de palavras formando uma frase ou oração ou um *n-grams* (grupos de n elementos consecutivos extraídos do texto), ou ainda uma árvore descrevendo a análise sintática da oração.
- Som: Similar a texto, especialmente quando se refere a reconhecimento de voz. Considera-se uma série temporal de eventos, tipicamente fonemas para voz, em que cada elemento é traduzido como uma série de atributos tais como um conjunto de amplitudes e frequências.
- Imagens: Imagens são tipicamente tensores de três dimensões: coordenada horizontal, coordenada vertical e canal (por exemplo a cor em imagens codificadas em RGB). Imagens 3D, produzidas por exemplo por tomografias, adicionam uma coordenada para profundidade.
- Vídeos: Vídeos são similares a imagens adicionando uma coordenada temporal.
- Lista de atributos: Ocasionalmente opta-se por não ter uma arquitetura completamente ponta a ponta e um especialista escolhe uma lista de parâmetros que descrevem as partes importantes do problema.
- Saída de outras redes neurais: Redes neurais de múltipla camada criam uma série de representações intermediárias de complexidade progressivamente maiores, e cada módulo pode ser integrado em qualquer parte da rede, de modo que as redes anteriores buscarão criar uma representação adequada para que o novo módulo consiga realizar sua tarefa e aproximar-se da solução.

Como lidar com todos esses formatos de entrada é o resultado de muitas décadas de trabalho no desenvolvimento dos conhecimentos e técnicas em redes neurais e aprendizado de máquina, e, portanto, a capacidade de uma máquina virtual diferenciável de facilmente se integrar com todos esses modelos existentes é o que garante sua versatilidade. Essas entradas são processadas pelos seguinte módulos, até produzir uma saída que é o resultado da execução de um algoritmo criado:

- Codificador da entrada: Dentro do conceito de aprendizado profundo e redes de múltiplas camadas, o codificador é o módulo que cria uma representação mais compacta e eficaz da entrada para os módulos adiante. Por exemplo codificar uma pergunta (uma série de palavras) em uma representação vetorial de forma similar à arquitetura de *Encoder-Decoder*. No caso de entradas suficientemente simples o próprio interpretador pode fazer sua codificação.

- Interpretador de comandos: É o responsável por gerar os programas, produzindo o *Código de Operação* (ou *opcode*) a partir de cada comando de entrada e de uma representação do contexto atual do programa na forma do *Estado do programa*. Esse código de operação define unicamente a tarefa que a máquina virtual irá executar, incluindo qual operação (ex: soma, comparação, copiar dados...), quais seus argumentos e onde guardar os resultados (manipulação da memória, entrada e saída) e qualquer controle de fluxo (condicionais, laços). No contexto de redes neurais, esse módulo é chamado de *Controlador*.

- Estado do programa: É um estado mantido pelo controlador que indica o contexto atual do programa. É normalmente formado por uma combinação da codificação das escolhas anteriores do controlador, das entradas passadas e em alguns modelos algumas informações da máquina virtual. Esse estado não possui, no entanto, acesso aos dados na memória da máquina virtual, pois deseja-se que sejam criados algoritmos genéricos que não dependam dos dados específicos (por exemplo um somador não deveria se preocupar com os valores dos números que está somando desde que sejam válidos).

- Unidade de execução das instruções: É o módulo responsável por manipular a memória da máquina virtual, executando cada correspondente operação sobre os correspondentes setores da memória. Ao contrário dos demais módulos, a máquina virtual tipicamente não possui qualquer peso a ser treinado, porém é necessário que ela seja diferenciável para a criação de um modelo de ponta a ponta (que pode ser treinado recebendo apenas as entradas e a saída do programa desejado).

- Memória Interna + Externa: A manutenção do estado da máquina virtual, similar aos registradores e memória de um processador ou às variáveis em uma

linguagem de programação. Ela pode ser completamente interna ao modelo, com todos os valores gerados pela própria rede, ou externa, sendo parte da entrada como por exemplo um vetor de números para se ordenar. Essa entrada é diferente da entrada do controlador: essa é uma descrição do programa, enquanto aquela são seus argumentos. A saída dos modelos é tipicamente o estado final da memória.

Usando o exemplo 2, o codificador seria responsável por, primeiramente, transformar cada palavra em sua representação vetorial via, por exemplo, *word2vec*. Em seguida deve combinar todas as representações vetoriais de palavras em uma representação única para cada comando (nesse exemplo cada comando deve ser separado do resto do texto através de uma etapa de pré-processamento, pois existem vários comandos em uma só frase). A etapa de gerar uma representação para o comando poderia ser feito de forma similar ao codificador do *encoder-decoder* com ou sem atenção. A partir da representação do comando, e do estado do programa do controlador (tipicamente o seu próprio estado oculto sendo uma rede recorrente), o interpretador de comandos gera o código de operação para a máquina virtual.

O código de operação define qual operação a máquina virtual irá executar e com quais argumentos. No exemplo de comando "ponha 20 reais de crédito", a operação deve ser adição e as variáveis "saldo atual", mantida na sua memória interna ou externa, e "20", extraído do comando. possivelmente durante o pré-processamento. Após a máquina virtual atualizar sua memória com o resultado da operação, o processo se reinicia para o próximo comando, ou a saída final é extraída da memória no caso de término do algoritmo.

Ter só o codificador na figura 3.1 como uma Rede Neural significa utilizar modelos neurais apenas para processar a entrada e usar as ferramentas tradicionais para trabalhar com essa nova entrada. Por exemplo construir uma representação das palavras via *word2vec* e usar algoritmos comuns ou ferramentas matemáticas para processá-los. Apesar de úteis, elas requerem que já se saiba a solução do problema para cada caso, já que não há aprendizado quanto à construção do algoritmo que resolve o problema.

Ter até controlador como rede neural já entra no domínio dos *Programadores Neurais*, Redes Neurais que produzem algoritmos observáveis através de códigos de operação, ao contrário das redes recorrentes tradicionais que também produzem algoritmos inteiramente dentro de uma caixa-preta. Os programadores neurais podem ser divididos em dois grupos: aqueles que treinam o controlador separado da máquina virtual e aqueles que treinam o sistema de ponta a ponta.

3.2.1 Treinar a rede neural e a máquina virtual separadamente.

Da forma mais simples, podemos ter a rede neural sendo treinada de forma completamente independente da máquina virtual. Para tanto treina-se a rede neural de forma supervisionada, através de uma base de dados manualmente produzida com a entrada sendo os comandos em sua forma original não estruturada e a saída com um formato compatível com a máquina virtual a ser utilizada.

A vantagem dessa abordagem é que as ferramentas utilizadas não possuem nenhuma restrição de execução: programas de manipulação algébrica, programação lógica e até banco de dados e buscas na web são viáveis, já que a máquina virtual não é parte da rede a ser treinada. Para sistemas que não podem ser escritos de forma diferenciável essa é a única opção.

A desvantagem é a necessidade de construir um banco de exemplos para treino novo, e tipicamente com muitas amostras para qualquer problema não trivial, que ficará sempre restrito a uma ferramenta específica. Qualquer mudança na interface da ferramenta exigirá que o banco de exemplos seja reescrito e que a rede seja treinada novamente. Cada amostra do banco também será consideravelmente mais complexa pois quem criará precisará ter domínio da ferramenta, e em casos com múltiplas instruções é necessário escrever um programa inteiro para cada exemplo em vez da saída final.

3.2.2 Treinar o sistema completo ponta a ponta

O sistema treinado ponta a ponta (*end-to-end*) tem como vantagem a maior flexibilidade. As bases de exemplos não mais precisam explicar como realizar a tarefa, mas apenas dizer qual o resultado desejável e deixar para a rede decidir a melhor forma de resolvê-lo. Dessa forma os exemplos não só são mais simples, mas também não dependem de que quem os crie saiba resolver a tarefa com as ferramentas escolhidas. Assim, é possível inclusive que a rede produza novas soluções para problemas que os próprios desenvolvedores das ferramentas nunca planejaram. Torna-se possível também integrar diretamente o sistema a uma rede neural ainda mais profunda, que utiliza a resposta para processamentos ainda mais complexos e treinar o processo todo junto em uma única base de dados. E se o usuário quiser mudar a máquina virtual basta retreinar na mesma base de dados.

Dentro dessa abordagem, existem ainda duas possibilidades no que diz respeito a como treinar a rede. A primeira forma é o treinamento por reforço. O treinamento por reforço tem a vantagem da primeira abordagem em termos de não ter restrições de implementação para a máquina virtual, podendo utilizar até uma pesquisa na internet por exemplo. A rede neural irá produzir um código de operação que será

repassado à máquina virtual, que por sua vez retornará o resultado de seu processamento. Se o resultado for correto, os pesos da rede serão ajustados para favorecer essa instrução. Caso contrário. para rejeitar essa instrução. O problema no entanto é que sistemas de reforço são extremamente lentos de treinar, especialmente se o espaço de busca for muito grande (como em programas relativamente longos) e as soluções corretas muito raras em comparação, forçando a rede a errar muitas vezes antes de ter qualquer progresso.

A segunda forma, e a que nosso modelo foca, é o treinamento supervisionado. Como explicado, usa-se um algoritmo de otimização como gradiente descendente para minimizar o erro na saída. Como o gradiente descendente utiliza o gradiente, é necessário que a função seja diferenciável em toda sua extensão, o que restringe consideravelmente a implementação da máquina virtual. Felizmente, boa parte das operações algébricas básicas são diferenciáveis (somas, produtos, exponenciação, e aproximações de demais na forma de séries), enquanto operações lógicas mais complexas, tais como predicados, podem ser implementadas de forma aproximada.

3.3 Programadores Diferenciáveis

Programação diferenciável é a técnica que tenta combinar o reconhecimento de padrões à natureza de aproximador universal das redes neurais com a série de operações discretas de algoritmos tradicionais, produzindo redes neurais capazes de construir algoritmos mais explícitos. Fundamentalmente, redes neurais são simplesmente uma cadeia de transformações geométricas, e achar uma transformação geométrica que generaliza cada operação aritmética e lógica é difícil mesmo com uma grande quantidade de exemplos. Construir uma cadeia dessas operações, mesmo antes de considerar aspectos importantes como controle de fluxo, se torna exponencialmente mais difícil. Por exemplo, mesmo uma simples soma ou produto de inteiros é uma tarefa não trivial para uma rede neural aprender, especialmente considerando a distorção causada pela transformação não linear que ocorre a cada passo.

Integrar aspectos de algoritmos a redes neurais tradicionais não é uma exclusividade desses modelos. Os mecanismos de atenção já vistos permitem à rede aprender de forma completamente diferenciável quais dados ele deseja acessar a cada passo. Através desses mecanismos são efetivamente testados todas as possibilidades em cada passo, produzindo uma distribuição discreta da probabilidade de acesso de cada entrada, que será multiplicada pelos valores. Se escolher apenas uma entrada minimiza o erro na saída, então essa distribuição vai tender a apontar para apenas uma das opções.

Programadores neurais dão um passo a mais e não somente aplicam a seleção à entrada a ser processada mas também à operação que transformará essa entrada,

fazendo isso recorrentemente. Para tanto, possuem uma seleção de operações diferenciáveis que, selecionadas através de mecanismos de seleção a cada passo, induzem um algoritmo capaz de transformar a entrada na saída desejável. A operação de seleção normalmente tem a forma:

$$result = oplist(args)^T softmax(opcode) \tag{3.1}$$

Em que *oplist* é um vetor com tamanho N com cada campo sendo uma operação tal como soma ou multiplicação, e o *opcode*, o código de operação, um vetor com N valores, tipicamente gerados por uma rede neural recorrente a cada passo. Basicamente, a seleção significa utilizar a função *softmax* para gerar uma lista de pesos normalizados para cada operação, e então multiplicar o resultado de cada operação pelo seu respectivo peso. Assim, ao final do treino, se uma operação levar à resposta correta, essa operação será multiplicada por aproximadamente um, e todas as demais multiplicadas por aproximadamente zero, e no somatório será como se apenas a operação escolhida existisse.

Dessa forma, os programadores neurais são capazes de desacoplar os dados de entrada da máquina virtual do processo de decidir qual transformação aplicar a eles, o que é feito agora pelo controlador, garantindo ao modelo a capacidade de criar soluções mais gerais. E como os dados são processados diretamente por uma máquina virtual diferenciável, a rede pode lidar com problemas que seriam difíceis de lidar usando apenas a tradicional série de transformações geométricas. A seguir estão algumas das mais conhecidas redes programadoras.

3.3.1 Neural Programmer Interpreter (NPI)

O primeiro exemplo é o modelo mais famoso em que se treina o controlador separado da máquina virtual, o *Neural Programmer Interpreter* (NPI) [11], com sua arquitetura na figura 3.2. O NPI é um modelo composto primariamente por uma única LSTM conectado a três seletores (módulos com saída *softmax* a partir o estado oculto da LSTM) que decidem a próxima instrução. Uma decide a próxima função ou subrotina a se executar, uma decide se a função terminou e uma decide os argumentos da função. Além disso há um codificador para produzir uma representação do estado do ambiente que a rede está manipulando.

Por apenas predizer o passo seguinte, e não um programa inteiro de uma vez, tendo portanto um espaço de busca menor e similaridade maior com os classificadores *softmax* já amplamente em uso, produz comparativamente bons resultados com um número reduzido de amostras. No entanto a base de dados é consideravelmente mais difícil de criar por precisar de um programa completo (cada instrução de entrada com sua instrução seguinte como saída) em vez de apenas o par de entrada e saída. Dessa

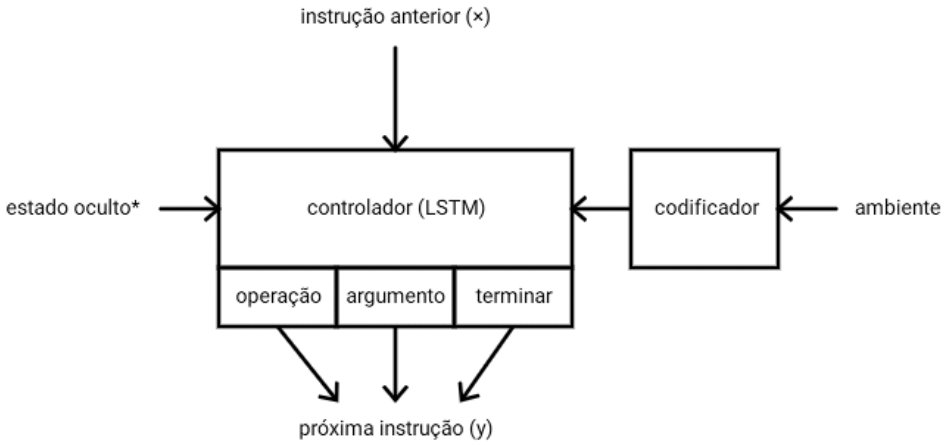

Figura 3.2: O *Neural Programmer Interpreter*. O modelo recebe a instrução anterior e uma representação do ambiente para gerar a instrução seguinte. O estado oculto é mantido em uma pilha para emular chamadas a funções.

forma também dificulta que os modelos aprendam a construir algoritmos inéditos para resolver problemas sem solução preexistente. Como ponto positivo no entanto, e como visto acima, a máquina virtual não precisa ser diferenciável, pois ela não será treinada junto ao modelo para se gerar a saída.

Uma outra propriedade interessante é que, diferente dos outros modelos que são primariamente imperativos, o NPI tem uma estrutura primariamente recursiva, com a rede neural podendo chamar subprogramas a cada passo, recebendo um estado oculto novo zerado, e ao terminar o subprograma recuperando o estado oculto anterior de uma pilha de contextos. Essa estrutura recursiva é possível por se ter a estrutura de execução já predefinida por cada exemplo na base de treino.

3.3.2 Neural Programmer (NP)

O *Neural Programmer* [10], na figura 3.3 é um modelo de perguntas a uma tabela, similar a um banco de dados simples com uma só tabela e uma linguagem de consulta não estruturada. Recebendo uma pergunta em linguagem natural e uma tabela de entrada, seleciona uma operação, dentre um conjunto predefinido de operações, e colunas para aplicar a operação. O treinamento envolve encontrar a série de instruções usadas que minimizam o erro na célula de saída.

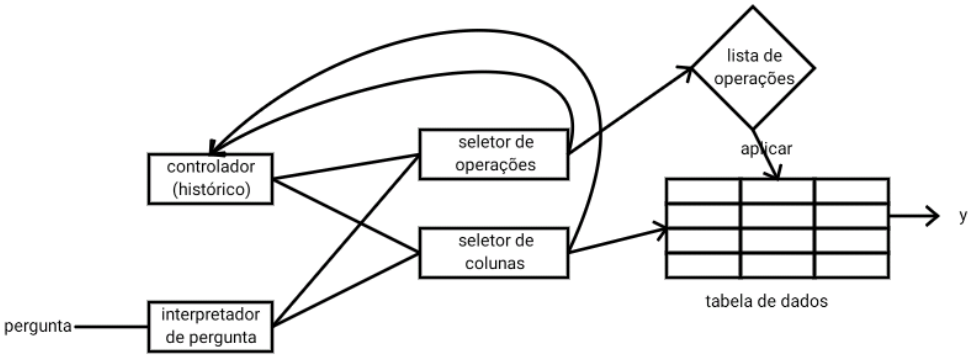

Figura 3.3: O *Neural Programmer*. O módulo da pergunta e da história criam juntamente uma representação de suas entradas. Esses dados são utilizados por seletores de operação e de coluna que definem respectivamente qual será a transformação e o que será transformado na tabela de entrada.

As operações envolvem o somatório de uma coluna, maior, menor e operadores lógicos. No caso das operações de comparação, utiliza-se um pré-processamento para extrair todos os números da pergunta, e utiliza-se um mecanismo de atenção para decidir qual número usar na comparação com cada elemento da coluna. Esse tipo de pré-processamento faz com que o modelo deixe de ser completamente diferenciável ponta a ponta, mas o pré-processamento é simples o suficiente para não ser um problema.

O modelo consiste de quatro componentes: uma rede recorrente, tipicamente uma LSTM, que guarda as informações das últimas seleções de operações e colunas, chamada de *History RNN*, uma RNN que produz uma representação distribuída da pergunta (o codificador da entrada), chamada *Question Selector*, um componente que faz duas operações de *softmax*, um para selecionar a operação e um para selecionar a coluna e finalmente uma lista de operações.

Através de múltiplos ciclos controlados pelo *History RNN*, induz-se um algoritmo completo.

3.3.3 Neural Random Access Machine (NRAM)

O *Neural Random Access Machine* [12], representado na figura 3.4, é um programador que manipula sequências em listas. Tem como dados em todos os registradores da máquina virtual um ponteiro, na forma de uma distribuição de probabilidade ou um vetor de atenção com o mesmo tamanho da lista a ser processada. Esses vetores podem ser transformados ou combinados através de operações lineares que somam e multiplicam os campos sem mudar sua soma total de um, implementadas de forma

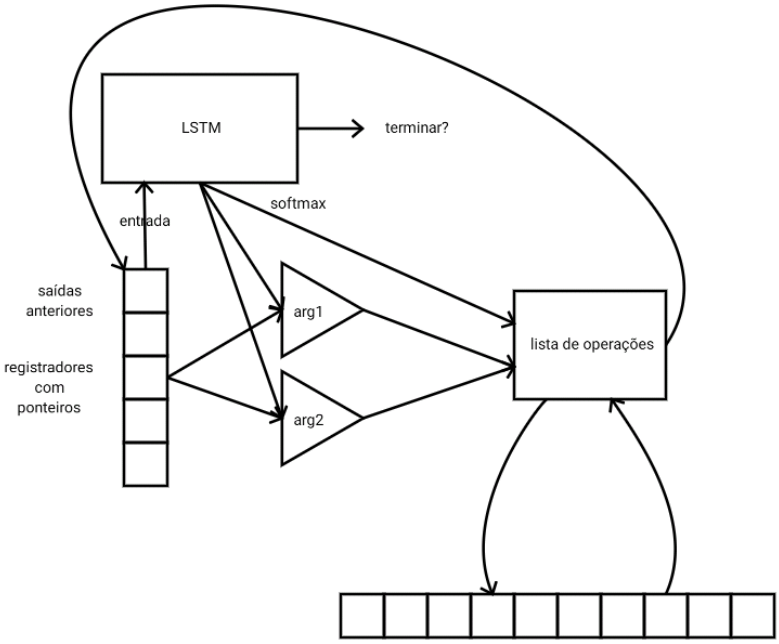

Figura 3.4: O *Neural Random Access Machine*. Uma LSTM que atua como controlador utiliza seleção via *softmax* para os dois argumentos e para as operações cujo resultado devem ser armazenados. Todos os registradores mantém ponteiros que são usado pelos operadores para operações de leitura e escrita. O controlador recebe apenas a informação de quais registradores estão vazios, e também produz o comando de parada.

similar a *Look up Tables*. Cada ponteiro também pode ser usado para ler ou escrever de uma fita de memória, dando sua capacidade de manipulação avançada de listas.

O controlador é uma LSTM que seleciona dois vetores entre resultados anteriores e registradores, e também seleciona os resultados dos operadores que serão guardados para uso futuro nos registradores.

A entrada do modelo é uma fita de memória, e a saída uma nova fita de memória com as transformações aplicadas. Para desacoplar os dados sendo processados dos algoritmos sendo criados, o controlador recebe apenas um vetor com a probabilidade de que cada registrador seja zero, e não o seu conteúdo. Dessa forma que o controlador não sabe os dados sendo processados, a menos que os vetores guardados representem lógica booleana (sabendo o que é zero ou falso sabe-se qual é verdadeiro).

3.3.4 Differentiable Forth

O *Differentiable Forth* [42] recebe um programa em uma linguagem baseada em *Forth*, uma linguagem de programação com sintaxe em notação polonesa reversa, com um número de partes faltantes, ou *slots*, com anotações para serem usados pelo mecanismo de inferência.

A máquina virtual é similar ao interpretador real da linguagem *Forth*, que é tido como um dos mais simples entre linguagens de programação com nível de abstração acima da de máquina. Possui duas pilhas (uma para os dados e uma para o retorno), ambas com um ponteiro para ser modificados pelas operações como PUSH e PULL, um *heap*, assim como outras linguagens de programação, e uma lista de operações implementadas.

Os *slots* requerem uma descrição de uma arquitetura de rede neural do tipo *Encoder-Decoder*, que irá ocupar a falta de uma instrução. O codificador permite, por exemplo, a seleção de dados a observar para corretamente decidir qual ação executar. O decodificador, por outro lado, é relacionado à operação a ser executada, como a escolha de uma operação dentre uma lista fornecida na anotação, de forma similar às outras redes programadoras, ou a permuta de elementos listados na memória. Esse modelo mostra como máquinas diferenciáveis tão complexas quanto linguagens de programação podem ser criadas e treinadas através de gradiente descendente em pares de entrada e saída.

3.4 Limitações atuais e a proposta de novo modelo

Além da dificuldade de convergência, possivelmente por causa da falta de estudo sobre esses modelos ainda novos, os modelos criados buscaram resolver classes de problemas diretamente e de ponta a ponta. Faltam modelos capazes de facilmente se integrar a modelos existentes, proporcionando as propriedades adicionais em transparência e indução de uma lógica mais estruturada às arquiteturas já amplamente utilizadas.

Buscando construir essa ponte, introduzo nesse trabalho uma nova arquitetura de programador diferenciável focando em baixo custo de treino e execução, e fácil integração com os modelos neurais comuns em produção. Enquanto simples em sua base, é flexível para cobrir muitos dos casos de uso dos algoritmos acima. Além disso, mesmo estendido possui menos parâmetros que cada um dos modelos acima, tendo um pouco menos parâmetros que uma GRU com mesmo tamanho do estado oculto, o que já é aproximadamente três quartos do número de parâmetros de uma única LSTM nas mesmas condições.

Também não requer nenhum tipo de estrutura complexa de entrada ou saída tanto no treino quanto na execução, como tabelas, fitas de memória, código de execução de um algoritmo ou programas completos com partes faltantes anotadas, e pode diretamente trabalhar em qualquer tipo de problema que uma rede recorrente trabalha, desde que o conjunto de instrução seja expressivo o suficiente. E, apesar do fato de não requerer nenhum tipo de memória externa, elas podem ser adicionadas e acessadas tanto através da parte neural (controlador) como da máquina virtual.

Adicionalmente, ao contrário dos modelos prévios, suas instruções podem ter diferentes números de argumentos, Isso permite tanto as funções agregadoras do *Neural Programmer* como as operações com dois argumentos do *Neural Random Access Machine* coexistirem na mesma máquina virtual, aproximando a flexibilidade da máquina diferenciável de *Forth* com uma arquitetura muito mais simples.

Finalmente, comparando com as abordagens comumente usadas de atenção, o modelo não requer, embora permita, nenhuma operação de *softmax*, o que pode potencialmente permitir melhor estabilidade numérica (por não requerer operações de divisão, usado pelo *softmax* na normalização) e otimização.

Capítulo 4

Gated Recurrent Programmer Unit (GRPU)

Nessa seção iremos explicar o modelo base da *Gated Recurrent Programmer Unit* (GRPU) desenvolvido nesse trabalho, incluindo como criar as instruções que ele é capaz de executar e como integrá-lo em modelos profundos.

4.1 A arquitetura

A GRPU é construído sobre a bem compreendida e testada estrutura da *Gated Recurrent Unit* (GRU). Não somente pode ser facilmente utilizado em qualquer modelo em que uma GRU pode ser usada, adicionando os benefícios na observabilidade e controle, como também pode ser implementado com apenas algumas linhas de código a partir da GRU. A diferença fundamental entre os dois modelos é a forma como o novo estado é produzido, mas essa pequena diferença também afeta como todos os elementos da rede são interpretados.

Na GRPU, a transformação afim que é seguida pela transformação não linear na GRU, mostrada na equação 2.15, é substituída por uma *Unidade Lógica e Aritmética*

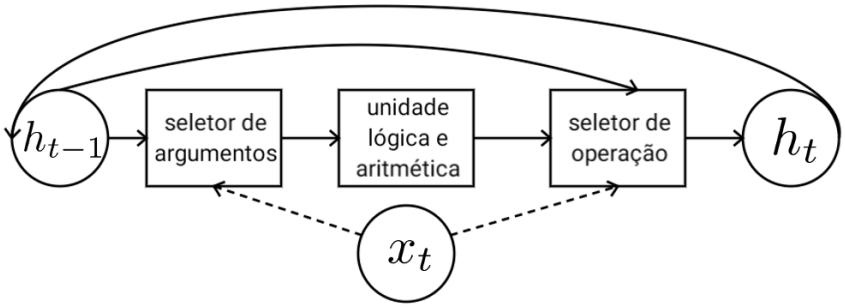

Figura 4.1: O *Gated Recurrent Programmer Unit* básico. As linhas tracejadas são as entradas das portas (gates), as linhas normais são o percurso do estado oculto.

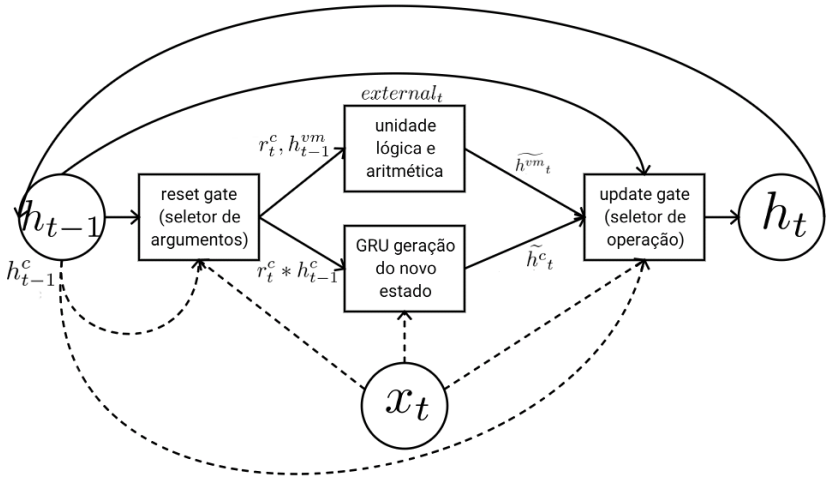

Figura 4.2: O *Gated Recurrent Programmer Unit* completo. h^{vm} se refere ao estado da máquina virtual, enquanto h^c é o estado do controlador. A parte inferior é o controlador, que é responsável junto com a entrada por controlar os *gates*, enquanto a parte superior é a máquina virtual, que executa as instruções de acordo com as seleções feitas pelos *gates* (seletores).

(*Arithmetic and Logic Unit* ou ALU), que executa o papel da Unidade de execução de instruções. Para tanto, a ALU executa uma operação para cada campo (ou grupo de campos) do *estado do programador*, ou estado da máquina virtual, para produzir os valores do próximo estado. Isso também faz com que o tamanho do estado do programador seja fixo, amarrado ao número de operações disponíveis.

A figura 2.9 representa a GRU tradicional, enquanto a figura 4.1 representa a GRPU. O *seletor de argumentos* e o *seletor de operadores* são estruturalmente iguais ao *reset* e *update gate* respectivamente, com a diferença sendo que o estado oculto específico à máquina virtual não é usado como seus parâmetros. Na GRPU, a ALU recebe o estado prévio da máquina virtual $h^{vm} \in \mathcal{R}^N$, em que N é tanto o número de operações como o número de campos de argumentos, e retorna um novo candidato para o próximo estado criado a partir do resultado de cada operação. O *reset gate* nesse contexto opera como seletor de argumentos, responsável por determinar quais argumentos serão alimentados para a ALU, tornando zero aqueles que devem ser ignorados. E o *update gate* define quais operações terão os resultados mantidos e quais serão ignorados. No último caso, os valores prévios do estado da máquina virtual são restaurados, e a operação é definida como um NOP (*No Operation*). O algoritmo é produzido computando cada instrução $[u, r]_t$ a cada passo t baseado nas entradas, o que é equivalente ao código de operação $[instruction, operands]_t$. Calculando todos os passos o modelo gera o algoritmo final, como o exemplo simples na figura 4.3. O sistema de equações desse modelo é:

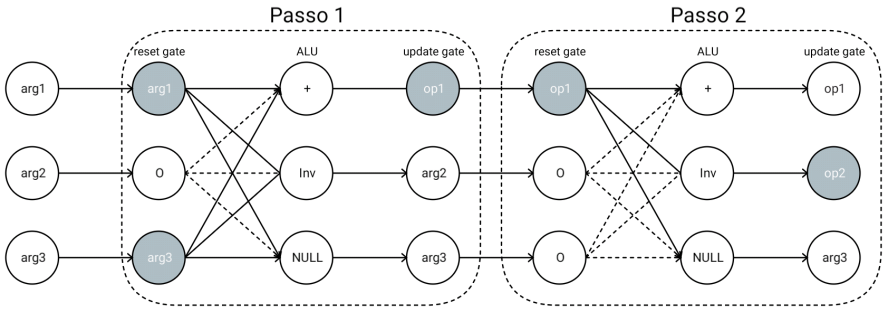

Figura 4.3: Exemplo de algoritmo de dois passos: -(arg1+arg3). Cada linha é tanto o argumento quanto a operação ao longo de dois passos da recorrência. O *reset gate* seleciona os argumentos para as operações da ALU (acinzentado na imagem com linhas sólidas), enquanto o *update gate* seleciona quais operações terão os seus resultados guardados (resultados de operação acinzentados).

$$r_t = \sigma(W_r.x_t + b_r) \tag{4.1}$$

$$u_t = \sigma(W_u.x_t + b_u) \tag{4.2}$$

$$\widetilde{h_t}[i] = ALU(r_t, h_{t-1}, external_t, operation[i]) \tag{4.3}$$

$$h_t = (1 - u_t) * h_{t-1} + u_t * \tilde{h}_t \tag{4.4}$$

As equações 4.1 e 4.2 são similares as da GRU, equações 2.13 e 2.14 respectivamente, com a única diferença sendo que na GRPU básica os *gates* recebem como entrada apenas x_t. Como é comum com programadores diferenciáveis, queremos desacoplar os dados transformados pelo algoritmo das decisões envolvidas ao gerar o algoritmo, de modo que o modelo possa aprender algoritmos mais simples e genéricos que funcionam com qualquer valor para os argumentos. Como tal, o estado da máquina virtual não é usado para definir o código da operação, de forma que apenas as entradas atuais são usadas para o controle dos seletores. A maior diferença entre a GRU e a GRPU é a equação 4.3 que representa o novo módulo adicionado, a ALU. A ALU é uma função que recebe o estado da máquina virtual (argumentos), a seleção dos argumentos, qualquer entrada externa e uma lista de operações para se aplicar aos argumentos, e retorna o novo estado oculto com os resultados de todas as operações, dados os argumentos selecionados. A equação 4.4 é idêntica a equação da GRU 2.16, e permite que o modelo selecione entre o estado prévio e os resultados da ALU do passo atual, segundo o vetor produzido pelo *update gate*.

Segundo o modelo atual, nenhuma entrada ou instrução do passado é considerada ao fazer cada passo do algoritmo, produzindo, então, um mapeamento direto entre a entrada e a respectiva instrução. Enquanto esse comportamento em algumas situações é apropriado, gostaríamos que o modelo usasse todas as informações do

presente e do passado da mesma forma que a GRU, e, por essa razão, adicionamos uma unidade adicional, que atua em paralelo à GRPU básica, denominado de controlador, e que tem a mesma estrutura da GRU. O modelo completo é representado na Figura 4.2, e representado pelo seguinte conjunto de equações a seguir:

$$r_t = \sigma(W_r.[h^c_{t-1}, x_t] + b_r) \tag{4.5}$$

$$u_t = \sigma(W_u.[h^c_{t-1}, x_t] + b_u) \tag{4.6}$$

$$\widetilde{h^c}_t = \tanh(W.[r^c_t * h^c_{t-1}, x_t] + b) \tag{4.7}$$

$$\widetilde{h^{vm}}_t[i] = ALU(r^{vm}_t, h^{vm}_{t-1}, external_t, operation[i]) \tag{4.8}$$

$$h_t = (1 - u_t) * h_{t-1} + u_t * \tilde{h}_t \tag{4.9}$$

Onde vm define a seção da máquina virtual e c a seção do controlador do estado oculto e dos seletores, $h_t = [h^{vm}_t, h^c_t]$, $r_t = [r^{vm}_t, r^c_t]$ e $u_t = [u^{vm}_t, u^c_t]$ são o estado oculto, *reset gate* (que assume a tarefa do seletor de argumentos para o estado da máquina virtual) e *update gate* (que assume a tarefa de seletor de operação para o estado da máquina virtual), respectivamente. Esse modelo implementa ao mesmo tempo a GRU na seção do controlador, e a GRPU básica acima na seção da máquina virtual, permitindo que o controlador (a GRU) mantenha seu próprio estado, através do qual cada comando usa informação de todos os comandos e decisões anteriores para gerar o código de operação para ser executado na máquina virtual (a GRPU básica). A máquina virtual também possui sua própria memória independente, armazenando os argumentos e resultados das operações, implementando assim todos os módulos do programador diferenciável geral representado na figura 3.1.

4.2 A unidade lógica e aritmética

Um importante aspecto a se considerar ao definir o conjunto de instruções pertencentes a ALU é o número de argumentos que cada operação permite, juntamente com a forma pela qual esses argumentos são selecionados. Definido isso podemos exemplificar várias operações para cada uma das categorias.

4.2.1 Operações agregadoras

As instruções mais naturais para a GRPU envolvem as operações agregadoras ou de aridade completa n, que são qualquer operação que recebe qualquer número de argumentos de zero até todos os campos do estado da máquina virtual. Para essas operações, argumentos são selecionados diretamente através do seletor de argumentos.

Tabela 4.1: Entrada desejada quando aceitando ou rejeitando o argumento.

entrada (i)	selector (r)	neutro 0	neutro 1
0	0	0	1
0	1	0	0
x	0	0	1
x	1	x	x

Quanto às operações agregadoras, deve ser considerado o elemento neutro na operação. O seletor de argumentos causa a rejeição de um argumento multiplicando-os por zero. Este comportamento não influencia operações que possuem elemento neutro zero, como a soma (qualquer número somado a zero é zero) ou o "ou" lógico, ou disjunção, (qualquer constante "ou"zero é a própria constante, sendo zero = falso e um = verdadeiro), ou seja a quantidade de zeros não afeta o resultado. Em outros casos, tais como o produto (qualquer número multiplicado por zero é zero) ou o "e" lógico, ou conjunção, (qualquer constante "e"zero é zero), um único valor rejeitado faria com que o resultado seja zero ou Falso, respectivamente, independentemente dos argumentos que foram de fato selecionados.

Para resolver esse problema, introduz-se uma transformação que faça com que os argumentos rejeitados (em que $r_t[i] = 0$) tenham valor um em vez de zero, e os argumentos selecionados continuem tendo o seu valor original, que pode incluir tanto o valor zero quanto o valor um. A tabela 4.1 possui as saídas que gostaríamos que cada caso tivesse. Mas uma complicação é que os valores do seletor de argumentos, que são gerados através da saída da função de ativação logística, não estão restrito a escolhas binárias, mas ao contrário, cobrem todo o espaço entre zero e um. Para lidar com casos que estendem além das duas possíveis constantes booleanas, utilizamos a lógica difusa. Em particular estamos selecionando a seguinte forma generalizada para as operações básicas:

$$x \text{ E } y = x \wedge y = x * y \tag{4.10}$$

$$x \text{ OU } y = x \vee y = x + y - x * y \tag{4.11}$$

$$\text{NÃO } x = \neg x = 1 - x \tag{4.12}$$

Considerando que a 1 é o único valor não-zero permitindo na lógica booleana, consideramos $x = 1$ para efeito a tabela verdade Booleana. A primeira tabela verdade é a coluna com o elemento neutro zero em função de i e r, que na lógica Booleana é facilmente representada $i \wedge r$, e, portanto nos operadores generalizados definidos:

$$i * r \tag{4.13}$$

Tabela 4.2: Exemplos de operações agregadoras

operação	implementação	elemento neutro
adição	sum(args)	zero
ou lógico/max	sum(args)/sum(r*)	zero
média	sum(args)/sum(r*)	zero
produto	product(args)	um
e lógico	min(args)	um

Que representa a operação básica do *reset gate* sem modificação, que já era válida para as operações com elemento neutro zero. Convertendo a tabela verdade 4.1, para o elemento neutro um em função de i e r, para uma expressão Booleana temos:

$$(i \wedge r) \vee (i \wedge \neg r) \vee (\neg i \wedge \neg r) \tag{4.14}$$

Que é a representação da tabela verdade na forma normal canônica disjuntiva, ou a soma de produtos, ou ainda *soma de mintermos*. $(i \wedge r)$ é o mintermo que representa a primeira linha da tabela, $(i \wedge \neg r)$ a terceira e $(\neg i \wedge \neg r)$ a quarta. Apenas as linhas cujo resultado não é 0 são representadas na soma de mintermos, e a expressão produzida é logicamente equivalente à tabela verdade desejada.

A forma de soma de mintermos pode ser simplificada usando a álgebra Booleana através dos seguinte passos:

$$(i \wedge r) \vee (i \wedge \neg r) \vee (\neg i \wedge \neg r) \text{ (Fatorando } \neg r \text{ nos últimos dois termos)}$$

$$\neg r \wedge (i \vee \neg i) \vee i \wedge r \text{ (Identidade } i \vee \neg i = 1)$$

$$\neg r \vee i \wedge r \tag{4.15}$$

Então, substituindo os operadores booleanos pelos operadores difusos, permitindo então generalizar para além de 0 e 1, temos:

$$(1 - r) \vee (i * r)$$

$$(1 - r) + (i * r) - (1 - r) * (i * r)$$

$$1 - r + i * r - i * r + i * r^2$$

$$1 - r + i * r^2 \tag{4.16}$$

Avaliando essa equação vemos que se o seletor de argumentos r for 0 teremos $1 - (0) + i * (0)^2 = 1$, e se o seletor for 1 teremos $1 - 1 + i * (1)^2 = i$, cobrindo corretamente os casos bases da tabela 4.1 para elemento neutro um.

Dessa forma, para qualquer operador que possua elemento neutro zero, usamos

Tabela 4.3: Exemplos de operações unárias

operação	implementação
contador	contador+1
alternador	1-alternador
deslocamento decimal	max(args)*10
inverter	-max(args)
copiar	softmax(u_t^{vm})*args

a equação 4.13 como entrada, e para qualquer operação com elemento neutro um usamos a equação 4.16 como entrada. Exemplos de operações agregadoras estão na tabela 4.2

4.2.2 Operações fixas

O oposto do exemplo acima são os operadores que não dependem de nenhum argumento, tais como RESET $= 0 e SET = 1$, assim como qualquer constante que se queira introduzir a máquina virtual.

Um exemplo mais complexo que começa a introduzir o conceito de acesso à dados externos seria uma função READ que lê uma entrada externa fixa, permitindo alimentar a ALU com dados diretamente a cada passo. Esses dados podem vir de listas de entrada paralelas, extraídos da entrada como utilizado no *Neural Programmer*, ou mesmo através de uma memória externa cujo controle de acesso é feito pelo controlador, aos moldes do utilizado pela *Neural Turing Machine*, brevemente descrito na subseção 2.1.8.

4.2.3 Operações unárias

Quando se decide um número de argumentos restrito, um método de seleção de argumentos é necessário. Algumas formas de seleção de argumentos para operações unárias são:

- Seleção fixa: a operação recebe o argumento de um campo predefinido do estado da máquina virtual. Por exemplo, uma operação que alterna entre zero e um sempre que selecionada pode ser implementada aplicando $(1 - h[i])$ em que $h[i]$ é o resultado da própria operação no passo anterior, e quando esse for um torna-se zero e vice e versa.

- Agregador máximo (com módulo para valores negativos): Um método simples para se obter uma operação unária é escolher o argumento com maior valor. Argumentos rejeitados tendem a ser zero (com o elemento neutro zero), então quando todos os argumentos menos um forem zero, a entrada da operação tenderá a ser o argumento selecionado.

- *Softmax*: o seletor de argumentos tradicional em programadores neurais é a forma mais versátil de se selecionar um ou mais argumentos para uma operação.

- Endereçamento indireto: Se um dos campos armazena uma distribuição (ponteiro) com o tamanho correto, esse ponteiro pode ser usado para acessar qualquer campo no estado do processador ou mesmo em uma memória externa. É o formato de dados utilizado pela *Neural Random Access Machine*.

Combinando uma operação unária com uma operação fixa podemos construir um modelo de gerenciamento de memória similar ao *Stack-RNN* para adicionar uma memória externa mais complexa do que aquela usada pelo READ. Adiciona-se uma lista ao estado geral do modelo que é modificada por duas operações. Uma fixa, POP, que causa com que todos os campos sejam movidos para cima quando o correspondente seletor u_t for um, e não mover nada se for zero. A operação unária é o PUSH, que recebe um argumento para empilhar e move todos os demais valores para baixo na pilha de acordo com seu próprio seletor u_t. O equivalente do NOP é ambos seletores de operações serem zero.

Uma lista de exemplos de operações unárias está na tabela 4.3

4.2.4 Operações binárias

Para operações de aridade dois, usa-se uma combinação de seleção de argumentos visto acima. Todas as operações agregadoras podem ser realizadas com seleções de dois argumentos, além do operador de subtração.

Um aspecto das operações que não foi aprofundado é que operações não necessariamente são compostas de apenas um campo no estado da máquina virtual. Elas podem possuir qualquer número de campos, desde que todos esses campos sejam selecionados pelo mesmo seletor de argumentos e o resultado, com o mesmo número de campos, selecionado pelo mesmo campo do seletor de operações. O exemplo já dado foram ponteiros para endereçamento indireto de tanto o estado da máquina virtual quanto de qualquer memória externa, desde que o número de campos de ambos seja igual.

Além de ponteiros, um outro tipo de argumento composto são representações vetoriais, que podem ser qualquer entidade codificada por uma rede neural, como palavras processadas pelo *word2vec*, frases processadas pelo *Encoder-Decoder*, imagens, entre outros. Representações vetoriais podem ser transformadas através de operações com vetores, tais como a lei dos cossenos, mas mais interessante é usar o próprio poder de redes neurais como aproximadores universais para representar qualquer operação tal como em [43], em que se avalia predicados como ("Pablo Picaso", nacionalidade, "Espanha"). Para tanto, carrega-se os argumentos usando a

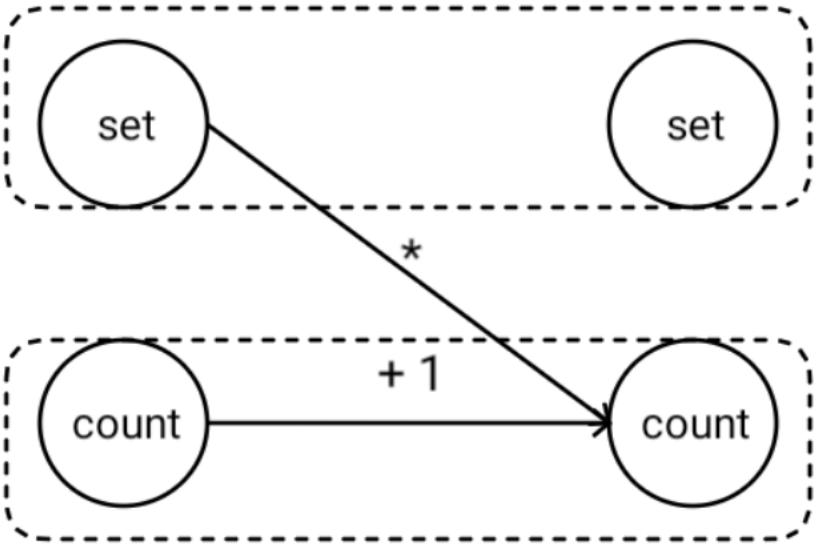

Figura 4.4: Operação composta: contador com *reset*.

operação *read* e uma nova entrada ao modelo, e cada predicado é uma operação binária que retorna um número para processamento posterior através de por exemplo os operadores lógicos já descritos.

4.2.5 Operações compostas

Para se criar máquinas ainda mais complexas, existe ainda a possibilidade de criar múltiplas operações que atuam como apenas uma instrução mais complexa.

Por exemplo na tabela 4.3 temos a operação contador, que a cada passo, se selecionado, aumenta seu próprio valor em um. Digamos que se queira reiniciar o valor. Se o seletor de argumentos for zero e o seletor de operações um, seu valor volta para um e não zero, e se o seletor de operações for zero então nenhuma mudança será efetivada.

Para resolver isso podemos utilizar uma estrutura como na figura 4.4, em que o resultado é multiplicado por uma outra operação (SET). Se o seletor de argumento do SET for um, então o funcionamento da operação ocorre da mesma forma que antes, porém se o seletor de argumento do *SET* for zero e o seletor de operação do contador for um, então o seu valor retornará a zero para o próximo passo.

Indo um pouco além, e se quisermos que a operação conte em *one-hot* em vez de apenas um número? Para tanto, podemos combinar operações compostas e argumentos compostos da forma na figura 4.5, em que se constrói uma simples máquina de estado que a cada ativação da operação move todos os elementos um campo abaixo. Quando o seletor do argumento de SET for um, a contagem começa, e

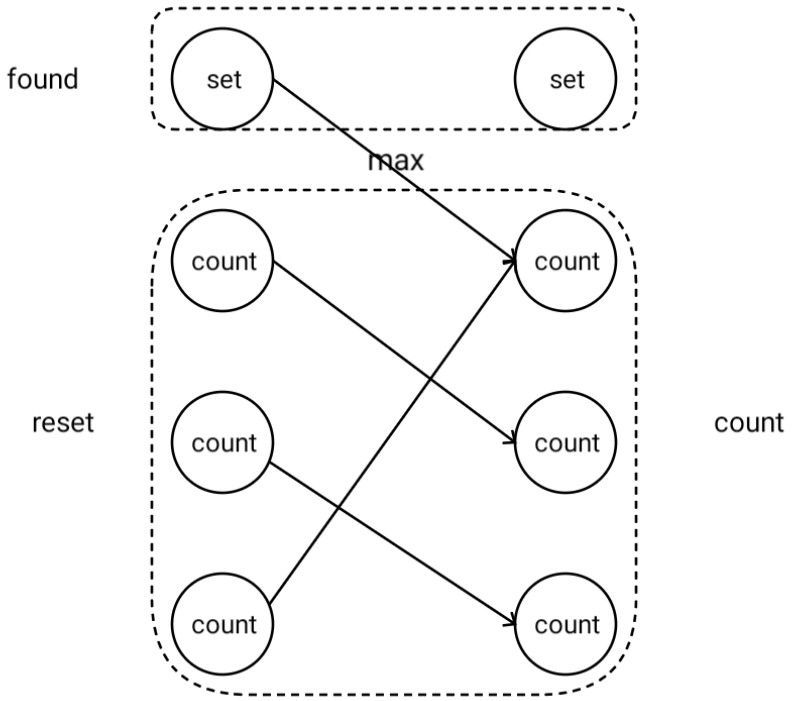

Figura 4.5: Operação composta: contador *one-hot*.

múltiplas contagens podem ser feitas ao mesmo tempo, permitindo que a instrução opere por exemplo como um ponteiro para criar uma seleção das entradas.

4.2.6 Comparadores e condicionais

Até esse ponto, os programas implementados foram completamente sequencias, o que já provê bastante funcionalidade, e foram o foco de todos os programadores diferenciáveis descrito no capítulo anterior, embora a máquina virtual do *Differentiable Forth* suporte essas operações. O controle de fluxo, elemento essencial de algoritmos, até esse ponto é feito exclusivamente através do controlador, que tem o poder de selecionar as operações a cada passo, podendo inclusive fazer com que o passo seja pulado, produzindo um vetor de zeros para o seletor de operações. O controlador, porém, não tem acesso às variáveis sendo processadas pela máquina virtual, e muitas vezes queremos que uma decisão seja tomada de acordo com esses dados, e, portanto, seria interessante apresentar essa possibilidade à máquina virtual.

Primeiramente, vamos definir as operações de comparação. Operações de comparação tipicamente possuem aridade dois (tais como igual, diferente, menor que, maior que) ou aridade um quando um dos argumentos é predefinido (igual a zero, diferente de zero, verdadeiro, falso...) e retorna um se a condição for verdadeira, ou zero caso contrário. Uma forma de implentar o *diferente* diferenciável (e o *igual* sim-

Tabela 4.4: Saída desejada para as operações condicionais.

u_{cond}	$\widetilde{h}_{cond}$	$\widetilde{u}_{op}$	u_{op}
0 (-)	0 (-)	0 (-)	0 (NOP)
0 (-)	0 (-)	1 (exec OP)	1 (OP)
0 (-)	1 (-)	0 (-)	0 (NOP)
0 (-)	1 (-)	1 (exec OP)	1 (OP)
1 (se)	0 (falso)	0 (-)	0 (NOP)
1 (se)	0 (falso)	1 (exec OP)	0 (NOP)
1 (se)	1 (verdadeiro)	0 (-)	0 (NOP)
1 (se)	1 (verdadeiro)	1 (exec OP)	1 (OP)

plesmente o subtraindo de 1) é tendo $|arg1 - arg2|/(|arg1 - arg2| + \epsilon)$ onde ϵ é uma constante para evitar divisão por zero. *Maior* ou *menor* podem ser implementadas com uma sigmóide/logística deslocada pelo valor que queremos comparar, aproximando a função degrau de Heaviside, que representa a comparação não-diferenciável. *Menor ou igual* e *Maior ou igual* não tem como ser representado de forma diferente do *menor* que e *maior* que nesse modelo.

A partir dos operadores de comparação, podemos implementar então o primeiro elemento de controle de fluxo na máquina virtual diferenciável, o operador condicional. Ele faz com que a instrução atual só seja executada caso a condição determinada pela comparação, ou uma combinação de comparações e operadores lógicos, seja verdadeira, e caso contrário toda a instrução será rejeitada. Isso é implementado modificando a operação de seleção de acordo com a tabela 4.4, em que $\widetilde{u}_{cond}$ é o valor do seletor de operações (update gate) para a operação condicional, $\widetilde{u}_{op}$ é o valor do seletor de operação para cada operação não condicional, h_{cond} o resultado da operação de comparação usado pela condicional, e u_{op} os valores finais de cada operação dado o efeito das condicionais (o valor da operação ou um NOP, ou *No Operation*, equivalente ao *update gate* rejeitando a operação. A tabela verdade 4.4, similarmente ao que foi feito para o elemento neutro um, pode ser representado por uma soma de mintermos:

$$(\neg u_{cond} \wedge \neg \widetilde{h}_{cond} \wedge \widetilde{u}_{op}) \vee (\neg u_{cond} \wedge \widetilde{h}_{cond} \wedge \widetilde{u}_{op}) \vee (u_{cond} \wedge \widetilde{h}_{cond} \wedge \widetilde{u}_{op})$$

Colocando $\widetilde{u}_{op}$ em evidência, temos:

$$\widetilde{u}_{op} \wedge ((\neg u_{cond} \wedge \neg \widetilde{h}_{cond}) \vee (\neg u_{cond} \wedge \widetilde{h}_{cond}) \vee (u_{cond} \wedge \widetilde{h}_{cond}))$$

Note que $(\neg u_{cond} \wedge \neg \widetilde{h}_{cond}) \vee (\neg u_{cond} \wedge \widetilde{h}_{cond}) \vee (u_{cond} \wedge \widetilde{h}_{cond})$ é equivalente a $\neg(u_{cond} \wedge \neg \widetilde{h}_{cond})$, pois com 2 variáveis temos 4 possíveis combinações, e se uma tabela verdade com 2 variáveis possui 3 combinações verdadeiras, a negação dela possui

apenas 1 (a única que originalmente era falsa). Aplicando a lei de De Morgan sobre $\neg(u_{cond} \wedge \neg\widetilde{h}_{cond})$ temos $(\neg u_{cond} \vee \widetilde{h}_{cond})$, de forma que temos uma implementação mínima da tabela verdade 4.4 através da expressão 4.17:

$$u_{op} = \widetilde{u}_{op} \wedge ((\neg\widetilde{u}_{cond}) \vee \widetilde{h}_{cond}) \tag{4.17}$$

E aplicando os mesmos operadores de lógica difusa discutidos anteriormente, chegamos na forma contínua e diferenciável que implementa a tabela acima no nosso modelo:

$$u_{op} = u_{op} * (1 + \widetilde{u}_{cond} * (\widetilde{h}_{cond} - 1)) \tag{4.18}$$

E para integrar no modelo de equações, com u_t sendo a saída final do *update gate* para usar na equação 4.9, $\widetilde{u}_t^c$ a seção do controlador e $\widetilde{u}_t^{vm}$ a seção da máquina virtual do *update gate* calculado em 4.6:

$$u_t = [\widetilde{u}_t^c, \widetilde{u}_t^{vm} * (1 + u_{cond} * (\widetilde{h}_{cond} - 1))] \tag{4.19}$$

Se a condição de rejeição ocorre, então a seção completa da máquina virtual do *update gate* é multiplicado pelo escalar zero, e o novo estado da máquina virtual se torna h_{t-1}^{vm} e o algoritmo não produz nenhum efeito no passo correspondente.

4.3 Integrando o modelo em redes profundas

Redes recorrentes podem ter sua expressividade estendida de forma simples, e a GRPU também pode de forma similar. A figura 4.6 mostra como o empilhamento ocorre mantendo uma única sequência de execução. Nesse caso, o estado do programador da última camada é o estado prévio do programador da primeira camada do passo seguinte, e os demais recebem o estado do programador da camada abaixo. Dessa forma a cada passo são executadas N instruções, em que N é o número de camadas, e cada passo continua a execução completa do passo anterior.

Essa estrutura permite que a máquina, através da composicionalidade de instruções, construa instruções muito mais complexas para lidar com cada entrada, inclusive combinações não planejadas durante o projeto da máquina virtual. O ponto negativo no entanto é que o cálculo do gradiente envolve retropropagar por todas as N camadas vezes o número de passos, aumentando as complicações envolvendo redes neurais muito profundas tais como explosão e desaparecimento do gradiente.

A figura 4.7 mostra uma outra forma de empilhar GRPUs, com o foco no processamento paralelo em que cada passo realiza operações nas suas entradas de forma

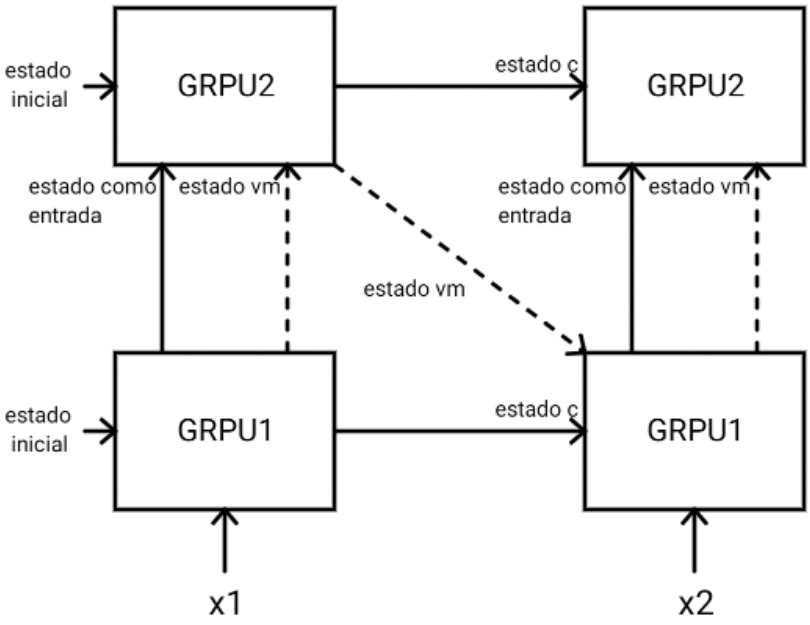

Figura 4.6: Empilhamento sequencial, estado vm se refere ao estado da máquina virtual, enquanto estado c se refere ao estado do controlador. O estado oculto do controlador é usada como entrada do controlador na camada acima.

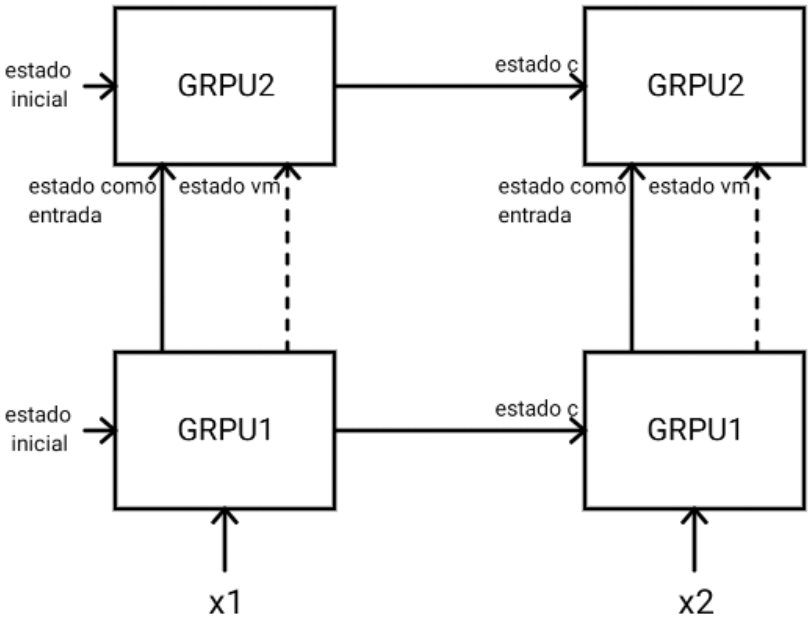

Figura 4.7: Empilhamento paralelo, estado vm se refere ao estado do programador, enquanto estado c se refere ao estado do controlador. O estado oculto do controlador é usada como entrada do controlador na camada acima.

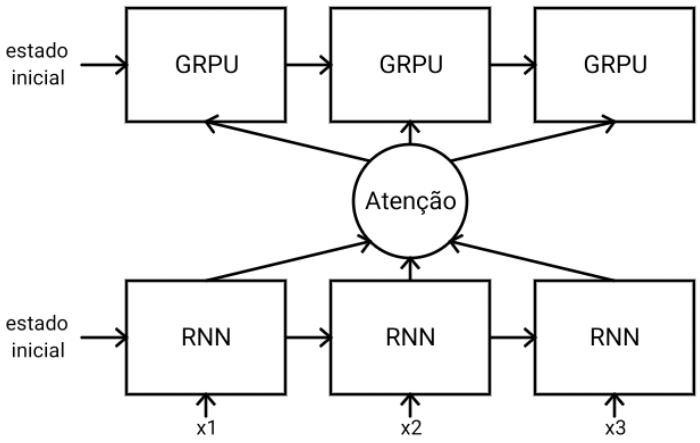

Figura 4.8: Modelo de atenção com GRPU.

independente. Em termos de programação, essa arquitetura permite implementar a função de alta ordem conhecida como *Map*, aplicando uma transformação independente a cada elemento da lista. Essa técnica pode ser particularmente eficiente em termos de gradiente se utilizar o modelo inicial apresentado sem estado do controlador, fazendo com que a rede opere como uma forma de convolução unitária da entrada, com a profundidade da retropropagação do gradiente sendo independente do número de passos.

Outras combinações também são possíveis, como passar o estado do processador do último passo da camada primeira camada para o primeiro passo da segunda camada, implementado um laço *for* em linguagem de programação, com uma iteração por camada.

Um modelo ainda mais interessante no entanto é a modalidade de *Encoder-Decoder* com atenção na figura 4.8, que é obtido com uma rede recorrente no codificador (que pode estar codificando instruções em linguagem natural, imagens ou qualquer outro tipo de entrada que redes neurais são capazes de trabalhar) e deixando a GRPU no decodificador. A capacidade do modelo de aprender a selecionar a entrada a cada passo permite a rede aprender seu próprio sequenciamento através das entradas, permitindo ao modelo efetivamente decidir um completo fluxo de controle para o programa de forma autônoma. Essa arquitetura também permite o desacoplamento do tamanho do programa com o tamanho da entrada, já que frequentemente mais de uma entrada, mesmo não consecutivas, podem representar uma única instrução, enquanto alguma entrada pode representar mais de uma instrução.

4.4 Aplicando a GRPU

A GRPU pode utilizar as mesmas entradas e saídas de uma Rede Neural Recorrente. A cada passo recebe uma entrada para o controlador e retorna um vetor de saída, então a princípio a substituição é imediata, dado que se foi projetada uma ALU adequada para o problema.

Por exemplo, considere uma rede simples que recebe uma série de comandos em linguagem natural com um número e uma operação ou soma ou produto (por exemplo: "adicione sete", "multiplique por 4", "some três unidades"), codificados em uma representação vetorial pelas camadas prévias da rede. A saída é o valor que se obtém executando todos os comandos segundo o algoritmo em pseudo-código a seguir:

```
acumulador <- 0
para comando em lista_de_comandos faça:
    operação, valor <- interpretar(comando)
    se operação == soma
     acumulador <- acumulador + valor
    se operação == produto
     acumulador <- acumulador * valor
retorne acumulador
```

Uma possível ALU que resolve esse problema (usando duas instruções por entrada, através do empilhamento sequencial) contém um conjunto de operações similar ao SET (operação fixa), mas cada um retornando um dos valores possíveis para a entrada (por exemplo 10 operações retornando de 0 a 9). Além disso contém a operação de soma, o produto e uma operação que simplesmente copia um valor para si (o acumulador). As instruções serão aprendidas pelo controlador via gradiente descendente, utilizando os pares de entrada (comandos) e saídas (resultado final das operações).

A primeira instrução de cada passo poderia ser selecionar a operação SET com o valor adequado, aprendendo a reconhecer números escritos em linguagem natural, realizando tanto a soma como o produto do mesmo com o acumulador. A segunda instrução poderá então reconhecer a operação presente no comando (soma ou produto), copiando apenas o resultado da operação correta para o acumulador. Essa é apenas uma das soluções possíveis com essa ALU, e apenas uma ALU entre várias que podem resolver o problema. A transparência da rede nos permite saber qual campo da camada oculta (da máquina virtual) estará o resultado (no acumulador), que será usado como saída do modelo.

No entanto, caso tenhamos uma forma externa de extrair o número da entrada, o problema pode ser drasticamente simplificado. Agora a rede terá uma entrada

além do padrão da GRU, que representa o $external_t$ na equação da ALU em 4.8. Com isso podemos substituir todos os SETs por uma só instrução READ, que lê esse valor específico a cada passo, e agora a única decisão que a rede precisa tomar é qual operação utilizar, e o argumento pode ser qualquer número real.

Uma outra possibilidade seria adicionar uma memória externa com um só campo (e que ao contrário do exemplo acima não é sobrescrita externamente para servir de entrada), que é automaticamente lida como entrada para a soma e o produto, e escrita por um dos dois, de acordo com o *update gate*. Dessa forma pode-se reduzir o número de instruções por passo para apenas uma cortando a necessidade de gerenciar manualmente o acumulador. Essa memória também é parte do $external_t$, sendo equivalente para a perspectiva da GRPU, embora não mais dependa de uma entrada extra, podendo ser simplesmente inicializada com zero antes do primeiro passo. A GRPU têm portanto, até duas entradas, a entrada do controlador, idêntica a da GRU, e a entrada da máquina virtual, que pode ser usada para repassar dados externos para máquina virtual, ou permitir gerenciamento mais sofisticado de variáveis.

As decisões relativas ao uso da GRPU são, portanto, escolher uma ALU com expressividade suficiente para resolver o problema, decidir o fluxo de execução via empilhamento e atenção, decidir as entradas da máquina virtual e decidir pelo uso de uma memória (com instruções da ALU que manipulam de alguma forma essa memória).

4.5 A Máquina Virtual da GRPU e de outros Programadores Neurais

Várias vantagens da GRPU já foram mencionadas, como baixo número de parâmetros, fácil integração em modelos neurais profundos e não depender de entradas e saídas incomuns. Essa seção compara exclusivamente a expressividade de sua máquina virtual em realizar os algoritmos dos outros modelos citados.

A GRPU permite uso de ponteiros para operações, incluindo para endereçamento indireto. Usando operações binárias e ponteiros como campos, e tendo uma lista como entrada do modelo, e operações de escrita e leitura na lista de forma similar ao PUSH e POP descritos, é possível replicar a máquina virtual do *Neural Random Access Machine*. Assim, a GRPU pode obter o mesmo grau de expressividade quando se trata de processamento de listas, acrescida de suas funcionalidades não presentes na NRAM como condicionais e manipulação de outros dados além de ponteiros ou representações *one-hot*.

O *Neural Programmer* depende de executar transformações em colunas e salvar

nas próprias colunas, algo que a GRPU tem dificuldade pois o resultado de cada operação sempre ocorre na saída da respectiva operação e não no argumento selecionado. Requer então duas instruções, uma para a operação e uma para copiar a coluna para o lugar original. A tabela é adicionada no modelo através do estado oculto inicial, e as operações nesse campo são para copiar (usando *softmax*). Alternativamente, modifica-se o controlador para gerenciar uma tabela externa independente da máquina virtual, que lê da tabela através de uma operação sem argumentos READ_COLUMN e salva ou usando WRITE_COLUMN ou através do próprio controlador usando o seu seletor de operações para extrair o resultado e salvar no mesmo lugar.

A máquina virtual do *Neural Programmer Interpreter* é primariamente recursiva, enquanto a GRPU é imperativa sem mecanismos para chamadas de função (manutenção de uma pilha de contextos e um mecanismo de retorno), então não é possível construir algoritmos comuns. Porém é possível treinar a GRPU utilizando o programa diretamente, com as saídas das bases de treino sendo o *opcode*, ou seja, a saída dos seletores de argumentos e operações. Essa forma de treino é mais eficaz, porém com as desvantagens vistas para modelos que não são ponta a ponta.

O *Differentiable Forth* usa uma abordagem consideravelmente diferente com substituições em linha, mas as operações da máquina virtual em si podem ser replicadas mesmo que com múltiplas instruções como no *Neural Programmer* pois envolvem primariamente operações aritméticas e gerenciamento de memória com pilhas. Os laços, existentes em *Forth*, são implementados através do empilhamento de GRPUs visto na seção anterior. Essa implementação não seria prática no entanto, pois seria uma máquina virtual extremamente complexa, e uma das principais forças da GRPU é sua simplicidade.

Capítulo 5

Experimentos

Infelizmente, sendo uma área ainda muito nova, não existem bases de treinos de qualidade para efeito comparativo de programadores neurais. Isso ocorre especialmente pelo fato de os programadores neurais até então resolverem seus próprios domínios de problemas específicos, com entradas e saídas específicas, sem interseção comum para realizar comparações.

Os experimentos focaram na capacidade do modelo de substituir facilmente redes recorrentes, sem nenhuma modificação especial como regularização e ajustes finos de hiperparâmetros. Foi usada uma variante do gradiente descendente chamado Adam [44] com taxa de aprendizado 1^{-4}. Estamos usando uma variante do problema da adição introduzido primeiramente no artigo original da LSTM [21] como base dos testes.

Esse problema é importante porque mostra que a rede proposta é capaz de produzir programas muito longos, um programa com 50 a 55 instruções para produzir a saída, mesmo que simples. Com apenas essas duas instruções, o modelo busca em um espaço de até 4^{55} combinações de instruções, ou aproximadamente 10^{33} programas possíveis. Isso torna métodos mais tradicionais de busca complicados de se utilizar, com o gradiente descendente sendo mais eficaz ao testar todos os possíveis programas em paralelo, ajustando-se sempre na direção de maior melhora do *mini-batch* a cada passo.

5.1 O problema da adição

A variante do problema da adição usado foi o apresentado no artigo de uma variante da GRU chamada *Minimal Gated Unit* [45], com o mesmo tamanho do *mini-batch* e mesmo tamanho da base de dados. No artigo é apresentado a performance para a GRU bidirecional, um modelo com o dobro de parâmetros que o nosso. O problema envolve uma lista com 50 a 55 duplas de dados (valor, controle), em que o valor é um número a ser adicionado e o controle é um valor entre -1, 0 e 1. Se o controle for

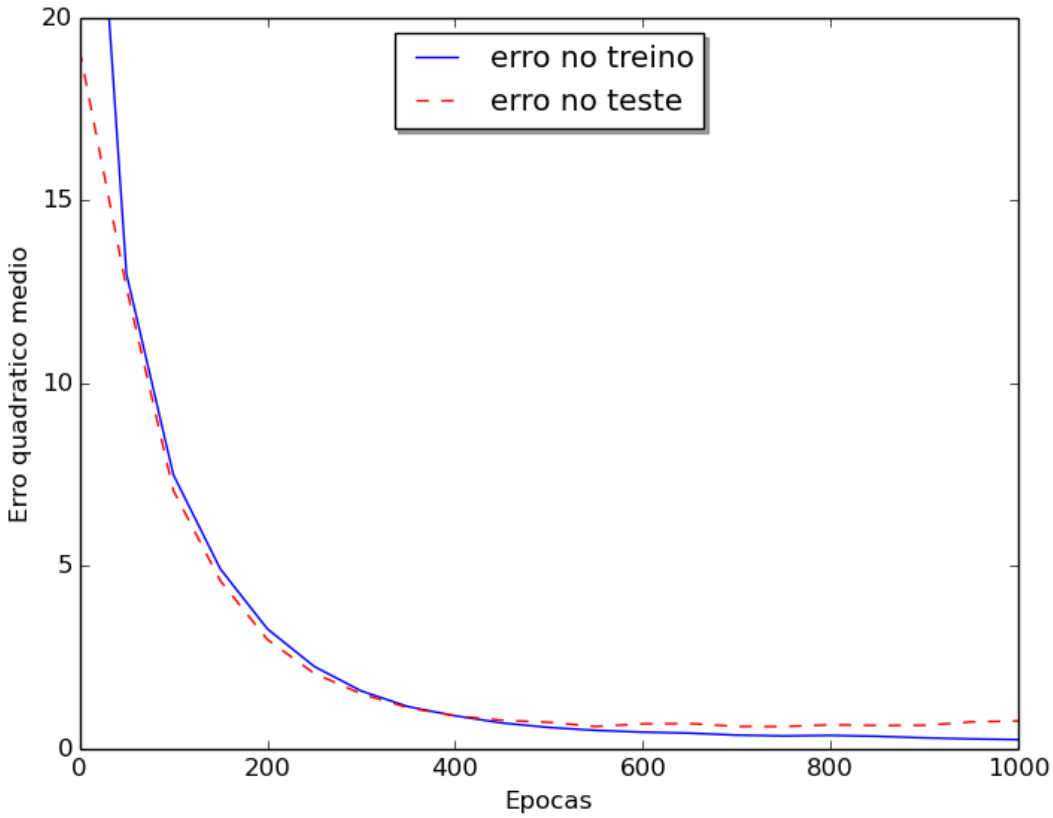

Figura 5.1: Convergência do problema da adição.

1 soma-se o valor, se for 0 ou -1 ignora-se o valor. Apenas dois números são somados a cada amostra. A base de dados é gerada aleatoriamente baseado nessas regras.

Alimenta-se apenas o controlador com o vetor de controle para evitar a dependência entre o programa sendo induzido e os dados sendo processados. Os dados a serem somados são lidos por uma operação de READ na máquina virtual, e o resultado da última soma é utilizado como saída do sistema, para ser treinado minimizando o erro quadrático com a saída. Apenas READ e o somatório foram utilizados na máquina virtual.

A figura 5.1 mostra o erro quadrático dentro dos conjuntos de treinamento e teste ao longo das épocas. O erro na época final foi de $0,247$ para o conjunto de treinamento sem sinais de atingir um mínimo local, mostrando que, mesmo com uma arquitetura simples, a convergência é significativamente mais lenta do que a GRU bidirecional do artigo referenciado acima, que em 1000 épocas alcança o erro na base de dados de teste de 0.0041. O erro no teste da GRPU foi de 0.759 mostrando um sobreajuste.

A velocidade de treinamento em cada época também é muito próxima de uma GRU com o mesmo número de parâmetros.

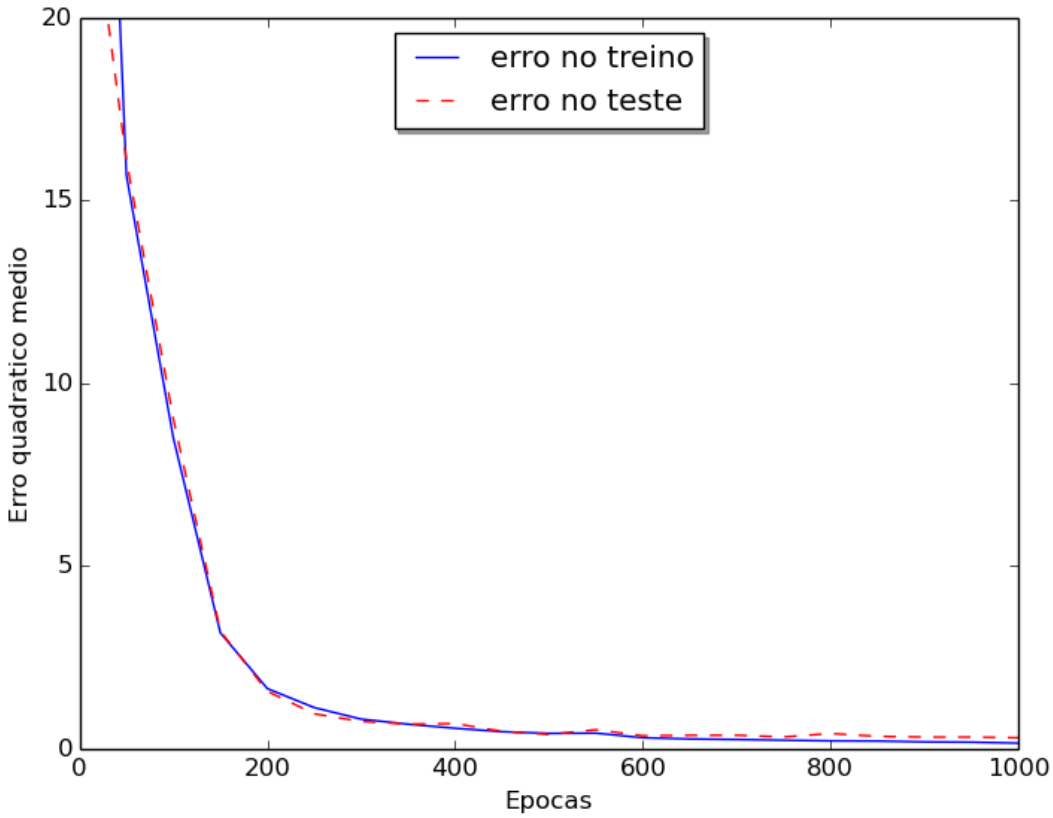

Figura 5.2: Convergência do problema da multiplicação.

Nós ainda experimentamos esse problema com um conjunto de dados maior e um tamanho de lote menor: o conjunto de dados foi aumentado por 32 e o tamanho do lote foi reduzido para 32. Neste caso, o erro quadrático atingiu $0,00890$ e $0,00699$ para treinamento e teste respectivamente, implicando que a convergência e a generalização podem ser fortemente melhoradas com conjuntos de dados maiores.

5.2 O problema da multiplicação

Para testar o problema de multiplicação, as seguintes modificações na rede foram feitas comparadas ao anterior:(1) os intervalos de entrada foram mudados para [0,5, 1.5] (para evitar que os valores fiquem muito próximos de zero); (2) adicionamos um novo operador à GRPU (o produto), atingindo um total de três operadores, onde o novo operador é a saída a ser usada na função de erro; e (4) o elemento neutro para a multiplicação é um, como explicado previamente.

Multiplicações através de tantos passos provocam um efeito exponencial no estado do programador, então é recomendado definir um limite do valor para evitar *overflow* nos valores em pontos flutuantes utilizados, dificultando o aprendizado.

Qualquer valor acima de 100, já muito distante de qualquer resposta possível é substituído por exatamente 100. A rede se comportou de forma semelhante ao problema de adição com convergência lenta, e ainda não conseguindo alcançar um o mínimo local após 1000 épocas como mostrado na figura 5.2.

Após 1000 épocas o modelo alcança 0,150 e 0,298 nos erros de treinamento e teste respectivamente. Após 5000 épocas, os erros são 0,0125 e 0,139.

Testou-se também como um experimento alternativo usando um método simplificado de aprendizagem curricular [46], em que se começa com problemas mais simples e progressivamente se avança para problemas mais longos e mais complexos. Porém usou-se número de épocas fixas para cada passo, em vez de uma condição de aprovação no teste mais simples para avançar para o mais complicado. Aumentou-se o tamanho da lista de entrada progressivamente de 10-15 até 50-55 dentro das mesmas 1000 épocas (200 épocas com 10-15, 200 com 20-25, 200 com 30-35, 200 com 40-45, 200 com 50-55). O resultado foi que não houve qualquer melhoria visível no erro de treinamento, em 0.193, mas o sobreajuste foi possivelmente reduzido com um novo erro de teste de 0.137, e o tempo de execução também foi reduzido ao usar menos passos para a maioria das amostras.

5.3 O problema da soma com condicionais.

Também foi feito um teste com uma variação do primeiro teste com todos os dados sendo alimentados ao programador e nenhum ao controlador, tornando o último incapaz de selecionar argumentos e operandos com base na entrada. Também foi introduzido um operador condicional que executa o passo atual apenas se a entrada for o alvo a ser somado (entrada de controle igual a '1').

A Figura 5.3 mostra que esse modelo converge rapidamente para um intervalo de erro comparativamente baixo em 0.600 com perda de teste 0.456 até a época número 200 e depois converge muito devagar, exigindo 800 épocas para chegar ao erro de treino 0.504 e 0.436 erro de teste. Ao inverter a comparação, ou seja apenas poder executar a instrução caso o controle não seja "1", tornando o programador não expressivo o suficiente para resolver o problema, a rede se torna completamente incapaz de convergir, ficando presa já nas épocas iniciais.

5.4 Interpretação dos resultados e testes adicionais

Os resultados iniciais encontraram problemas semelhantes às tentativas anteriores de desenvolvimento de programadores neurais no entanto: a convergência de tais

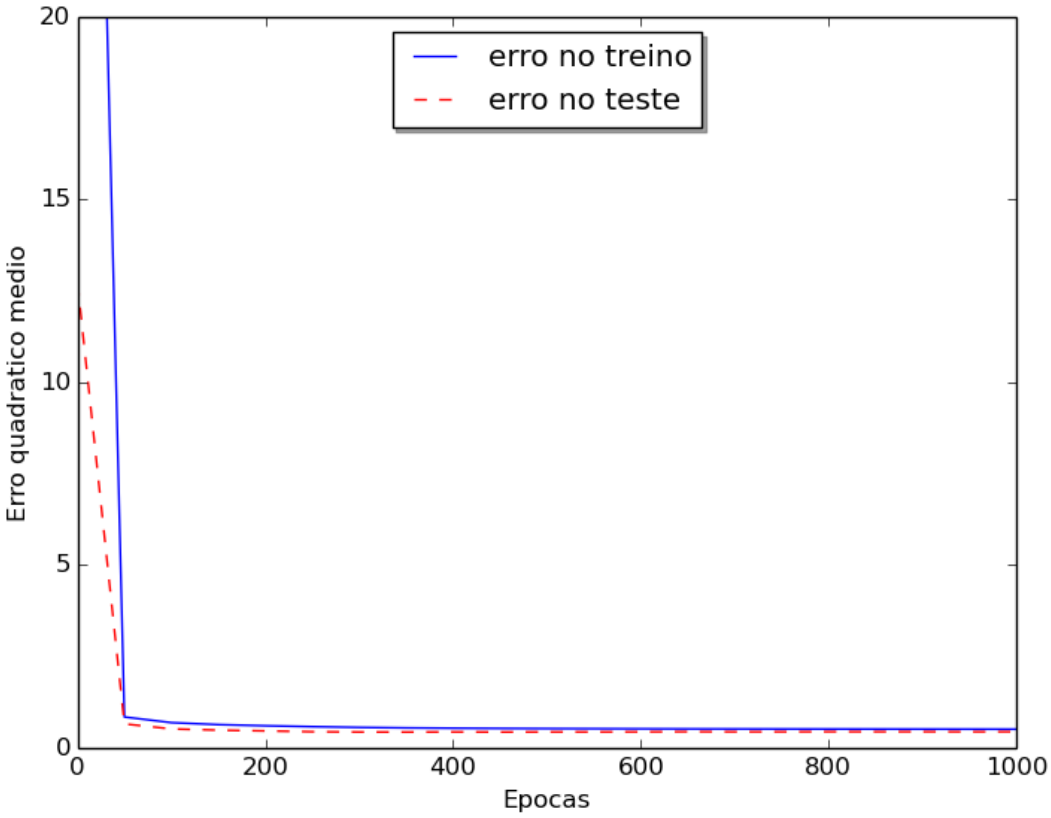

Figura 5.3: Convergência do problema da multiplicação com condicionais.

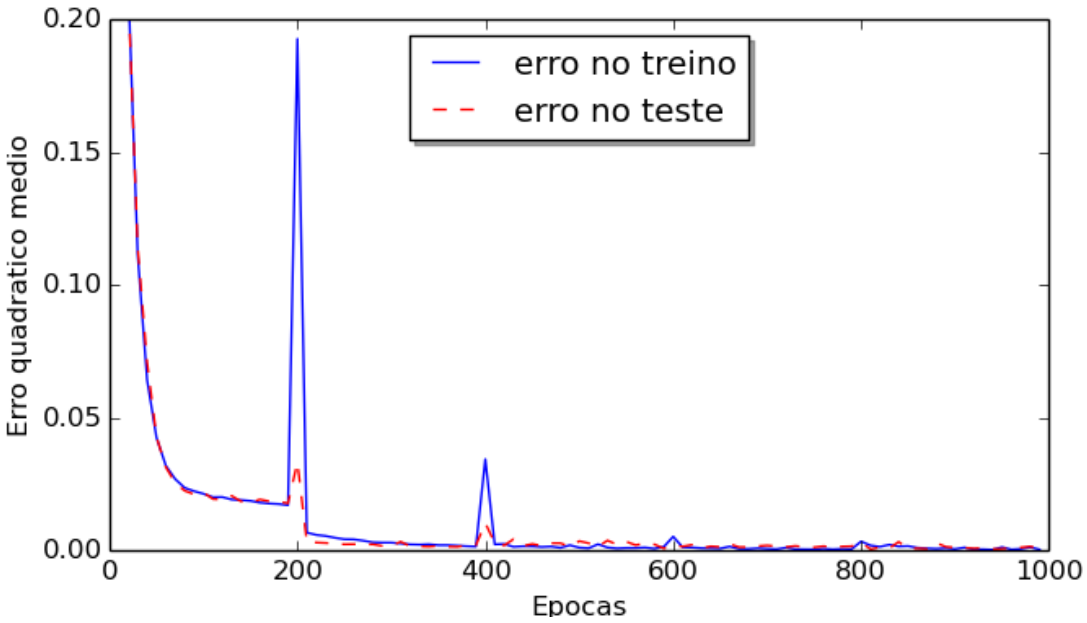

Figura 5.4: Convergência do novo experimento para adição.

modelos não é trivial. Avaliando as diferentes testes, vemos que o fator mais significativo para a convergência é o tamanho do *mini-batch*, sendo consideravelmente mais importante que o número de épocas executadas.

Uma possível razão para isso ocorrer é porque reduzindo o espaço de busca a apenas um vetor de tamanho fixo por passo (o código de operação), facilita o aparecimento de mínimos locais, enquanto se o espaço de busca se estende por toda a excepcionalmente alta dimensionalidade de redes neurais, encontra-se muito mais pontos de sela que mínimos locais. Diminuindo o *mini-batch* aumenta-se o fator probabilístico (o gradiente fica mais caótico, se aproximando do *Stochastic Gradient Descent*), permitindo a rede escapar melhor de mínimos locais.

Dessa forma, foi refeito o teste da adição com um *mini-batch* de apenas 10, com 3 operações (READ, soma e produto) e aprendizado curricular. Obteve-se então um resultado em 1000 épocas consideravelmente melhor que a GRU bidirecional no artigo de referência, atingindo um erro quadrático médio de 0.000426 e 0.000696 para treino e teste respectivamente. Os picos principais ocorrem quando o aprendizado por currículo muda para um problema mais difícil (adicionando 10 passos ao programa). Os picos menores são causados pelo fator probabilístico do gradiente ao usar *mini-batches* menores.

A variante da multiplicação está apresentada no gráfico 5.5. Atingiu 0.00616 como erro no treino, e 0.0166 de erro na base de testes. A variante com condicional manteve o mesmo padrão anterior, convergindo imediatamente para 0.06 de erro tanto para o treino quanto para o teste.

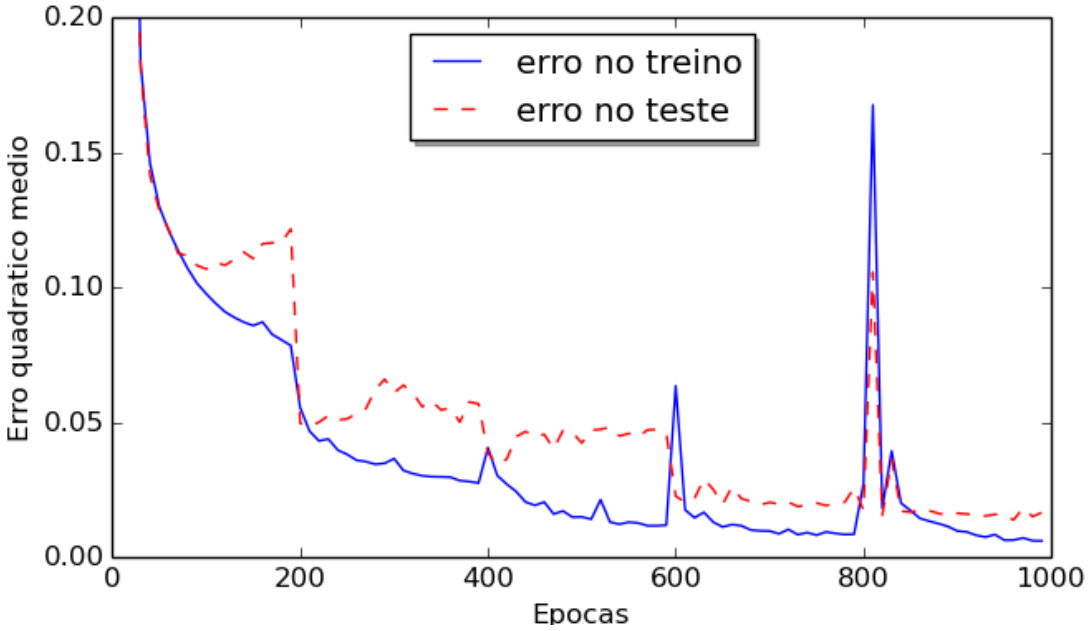

Figura 5.5: Convergência do novo experimento para multiplicação.

O resumo dos resultados finais, com a comparação com a GRU bidirecional, está presente na tabela 5.1, de forma que pode se observar que o modelo consegue alcançar, e até ultrapassar, um modelo mais complexo, porém restrito a um *mini-batch* menor.

Mais pesquisas nesta área ainda pouco explorada poderão fornecer melhores informações sobre a convergência e como melhorar os modelos em geral.

Tabela 5.1: Resultados dos testes após 1000 iterações e comparação com a GRU bidirecional.

Modelo	1000 Iterações (treino)	1000 iterações (teste)
GRU Bilateral – batch 100 – 1000 amostras*	N/A	0.0041
GRPU – batch 100 – 1000 amostras	0.247	0.759
GRPU – batch 32 – 3200 amostras	0.0089	0.00699
GRPU – batch 10 – 1000 amostras	0.0000387	0.00709
GRPU – batch 10 – 1000 amostras – Curriculum Learning	0.000426	0.000696
GRPU – batch 100 – 1000 amostras – Condicional	0.504	0.436
GRPU – batch 10 – 1000 amostras – Curriculum Learning – Condicional	0.06	0.06

Capítulo 6

Conclusão

Redes neurais atuais são caixas pretas, extremamente difíceis de se interpretar cada decisão. Ao criar modelos que, em vez de resolver o problema diretamente, gera as instruções para resolvê-lo, produz-se um modelo que pode ser interpretado e que é menos dependente dos dados específicos. Isso torna mais fácil a integração com redes neurais profundas, com a capacidade de rodar as máquinas virtuais inteiramente no ambiente diferenciável, permite-se a inclusão de algoritmos que simplificam o aprendizado da rede como bibliotecas facilitam escrever programas para domínios específicos.

A GRPU é um passo nessa direção, sendo um modelo conceitualmente simples, leve, extensível e fácil de ser integrado, tendo a mesma arquitetura de uma popular rede recorrente. Os testes iniciais revelam que é capaz de aprender mesmo programas longos, com um espaço de busca tão grande que se torna muito difícil para métodos que, além de não ter informação sobre a estrutura do programa, não usam o gradiente.

A GRPU ainda requer muitos trabalhos futuros para entender melhor seu potencial. Isso inclui:

- Seleção de hiperparâmetros: ou seja, quais os melhores parâmetros para se treinar, incluindo o tamanho do *mini-batch*, base de treino, taxa de aprendizado, formas de empilhamento e algoritmo de otimização.
- Diferentes configurações da ALU: incluindo como o aprendizado é afetado pela complexidade da máquina virtual;
- Formas de regularização: um fator essencial para convergência de redes neurais não foi avaliados, inclusive sugestões de outros autores de redes programadoras, como adicionar ruído aos pesos no caso do *Neural Programmer*;
- Transferência de aprendizado entre domínios: um dos grandes benefícios de máquinas virtuais diferenciáveis é compartilhar algoritmos entre domínios.

- Extração do código de operações: Criação de mecanismos que traduzem o código de operação para algoritmos em linguagem de programação;
- Criação de algoritmos discretos: O treino atual pode funcionar com pré-treino para modelos de aprendizado por reforço para aprender eficientemente algoritmos discretos complexos;
- Criação de bases de dados para programadores neurais: Sendo uma área muito nova, faltam bases de dados de qualidade que explorem as qualidades dos programadores neurais comparados a redes neurais tradicionais;

Além de integrar os novos conhecimentos e técnicas que surgirem, em uma área que teve inúmeros avanços em todos os anos recentes.

Referências Bibliográficas

[1] NORVIG, P., RUSSELL, S. *Inteligência Artificial: Tradução da 3a Edição*. Elsevier Brasil, 2017. ISBN: 9788535251418. Disponível em: <`https://books.google.com.br/books?id=BsNeAwAAQBAJ`>.

[2] CORTES, C., VAPNIK, V. "Support-vector networks", *Machine learning*, v. 20, n. 3, pp. 273–297, 1995.

[3] SILVER, D., HUBERT, T., SCHRITTWIESER, J., et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", *arXiv preprint arXiv:1712.01815*, 2017.

[4] MIKOLOV, T., CHEN, K., CORRADO, G., et al. "Efficient estimation of word representations in vector space", *arXiv preprint arXiv:1301.3781*, 2013.

[5] BOWMAN, S. R., ANGELI, G., POTTS, C., et al. "A large annotated corpus for learning natural language inference". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[6] RAJPURKAR, P., ZHANG, J., LOPYREV, K., et al. "Squad: 100,000+ questions for machine comprehension of text", *arXiv preprint arXiv:1606.05250*, 2016.

[7] WANG, S., JIANG, J. "Machine comprehension using match-lstm and answer pointer", *arXiv preprint arXiv:1608.07905*, 2016.

[8] GAUNT, A. L., BROCKSCHMIDT, M., SINGH, R., et al. "Terpret: A probabilistic programming language for program induction", *arXiv preprint arXiv:1608.04428*, 2016.

[9] BOSNJAK, M., ROCKTÄSCHEL, T., NARADOWSKY, J., et al. "Programming with a Differentiable Forth Interpreter". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, v. 70, *Proceedings of Machine Learning Research*, pp. 547–556. PMLR, 2017.

[10] NEELAKANTAN, A., LE, Q. V., SUTSKEVER, I. "Neural Programmer: Inducing Latent Programs with Gradient Descent", *CoRR*, v. abs/1511.04834, 2015. Disponível em: <http://arxiv.org/abs/1511.04834>.

[11] REED, S. E., DE FREITAS, N. "Neural Programmer-Interpreters", *CoRR*, v. abs/1511.06279, 2015. Disponível em: <http://arxiv.org/abs/1511.06279>.

[12] KURACH, K., ANDRYCHOWICZ, M., SUTSKEVER, I. "Neural Random Access Machines", *ERCIM News*, v. 2016, n. 107.

[13] HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. 2nd ed. Upper Saddle River, NJ, USA, Prentice Hall PTR, 1998. ISBN: 0132733501.

[14] GALLANT, S. I. "Perceptron-based learning algorithms", *IEEE Transactions on neural networks*, v. 1, n. 2, pp. 179–191, 1990.

[15] BENGIO, Y., SIMARD, P., FRASCONI, P. "Learning long-term dependencies with gradient descent is difficult", *IEEE transactions on neural networks*, v. 5, n. 2, pp. 157–166, 1994.

[16] GLOROT, X., BORDES, A., BENGIO, Y. "Deep sparse rectifier neural networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

[17] RUDER, S. "An overview of gradient descent optimization algorithms", *arXiv preprint arXiv:1609.04747*, 2016.

[18] LECUN, Y. "A theoretical framework for Back-Propagation". In: Mehra, P., Wah, B. (Eds.), *Artificial Neural Networks: concepts and theory*, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[19] WERBOS, P. J. "Backpropagation through time: what it does and how to do it", *Proceedings of the IEEE*, v. 78, n. 10, pp. 1550–1560, 1990.

[20] BAHDANAU, D., CHO, K., BENGIO, Y. "Neural machine translation by jointly learning to align and translate", *arXiv preprint arXiv:1409.0473*, 2014.

[21] HOCHREITER, S., SCHMIDHUBER, J. "Long short-term memory", *Neural computation*, v. 9, n. 8, pp. 1735–1780, 1997.

[22] CHO, K., VAN MERRIENBOER, B., GÜLÇEHRE, Ç., et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*, pp. 1724–1734. ACL, 2014.

[23] JOZEFOWICZ, R., ZAREMBA, W., SUTSKEVER, I. "An empirical exploration of recurrent network architectures". In: *International Conference on Machine Learning*, pp. 2342–2350, 2015.

[24] BLUMER, A., EHRENFEUCHT, A., HAUSSLER, D., et al. "Learnability and the Vapnik-Chervonenkis Dimension", *J. ACM*, v. 36, n. 4, pp. 929–965, out. 1989. ISSN: 0004-5411. doi: 10.1145/76359.76371. Disponível em: <http://doi.acm.org/10.1145/76359.76371>.

[25] VIDAL, R., BRUNA, J., GIRYES, R., et al. "Mathematics of Deep Learning", *arXiv preprint arXiv:1712.04741*, 2017.

[26] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., et al. "Dropout: A simple way to prevent neural networks from overfitting", *The Journal of Machine Learning Research*, v. 15, n. 1, pp. 1929–1958, 2014.

[27] SZEGEDY, C., LIU, W., JIA, Y., et al. "Going deeper with convolutions". .

[28] MIKOLOV, T., SUTSKEVER, I., CHEN, K., et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[29] PAN, S. J., YANG, Q. "A survey on transfer learning", *IEEE Transactions on knowledge and data engineering*, v. 22, n. 10, pp. 1345–1359, 2010.

[30] GRAVES, A., WAYNE, G., DANIHELKA, I. "Neural turing machines", *arXiv preprint arXiv:1410.5401*, 2014.

[31] JOULIN, A., MIKOLOV, T. "Inferring algorithmic patterns with stack-augmented recurrent nets". In: *Advances in neural information processing systems*, pp. 190–198, 2015.

[32] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep learning*. MIT Press, 2016.

[33] GOODFELLOW, I. J., SHLENS, J., SZEGEDY, C. "Explaining and harnessing adversarial examples", *arXiv preprint arXiv:1412.6572*, 2014.

[34] DOSHI-VELEZ, F., KIM, B. "Towards A Rigorous Science of Interpretable Machine Learning", *arXiv*, 2017. Disponível em: <https://arxiv.org/abs/1702.08608>.

[35] LONG, M., CAO, Y., WANG, J., et al. "Learning Transferable Features with Deep Adaptation Networks". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015*, v. 37, *JMLR Workshop and Conference Proceedings*, pp. 97–105. JMLR.org, 2015.

[36] YANG, W., LU, W., ZHENG, V. "A Simple Regularization-based Algorithm for Learning Cross-Domain Word Embeddings". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 2888–2894. Association for Computational Linguistics, 2017.

[37] VINYALS, O., FORTUNATO, M., JAITLY, N. "Pointer Networks". In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*, pp. 2692–2700, 2015.

[38] KLIR, G. J., YUAN, B. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1995. ISBN: 0131011715.

[39] CHOMSKY, N., SCHÜTZENBERGER, M. "The Algebraic Theory of Context-Free Languages*". In: Braffort, P., Hirschberg, D. (Eds.), *Computer Programming and Formal Systems*, v. 35, *Studies in Logic and the Foundations of Mathematics*, Elsevier, pp. 118 – 161, 1963. doi: https://doi.org/10.1016/S0049-237X(08)72023-8. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0049237X08720238>.

[40] SHAPIRO, S. C. *Encyclopedia of artificial intelligence second edition*. John, 1992.

[41] GOLDBERG, Y. "A Primer on Neural Network Models for Natural Language Processing", *CoRR*, v. abs/1510.00726, 2015. Disponível em: <http://arxiv.org/abs/1510.00726>.

[42] ROCKTÄSCHEL, T., RIEDEL, S. "End-to-end Differentiable Proving". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pp. 3791–3803, 2017.

[43] SOCHER, R., CHEN, D., MANNING, C. D., et al. "Reasoning with neural tensor networks for knowledge base completion". In: *Advances in neural information processing systems*, pp. 926–934, 2013.

[44] KINGMA, D. P., BA, J. “Adam: A method for stochastic optimization”, *arXiv preprint arXiv:1412.6980*, 2014.

[45] ZHOU, G.-B., WU, J., ZHANG, C.-L., et al. “Minimal gated unit for recurrent neural networks”, *International Journal of Automation and Computing*, v. 13, n. 3, pp. 226–234, 2016.

[46] BENGIO, Y., LOURADOUR, J., COLLOBERT, R., et al. “Curriculum learning”. In: *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.

[47] ABADI, M., AGARWAL, A., BARHAM, P., et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems”. 2015. Disponível em: <`https://www.tensorflow.org/`>. Software available from tensorflow.org.

Apêndice A

Implementação

A implementação do modelo foi feita em linguagem de programação python e com a biblioteca de manipulação de tensores Tensorflow [47] na versão 1.5. Essa biblioteca suporta diferenciação automática de forma que basta escrever o sistema de equação para fazer a predição que a biblioteca calcula os gradientes para cada peso.

Como dito, a GRPU é implementada modificando apenas algumas linhas da GRU, então comecemos com o código da GRU no Tensorflow.

```
class GRUCell(tf.nn.rnn_cell.RNNCell):

    def __init__(self, num_units):
        self._num_units = num_units

    @property
    def state_size(self):
        return self._num_units

    @property
    def output_size(self):
        return self._num_units

    def __call__(self, inputs, state, scope=None):
        with tf.variable_scope(scope or type(self).__name__):
            with tf.variable_scope("Gates"):
                ru = rnn_cell_impl._linear([inputs, state],
                                           2 * self._num_units, True)
                ru = tf.nn.sigmoid(ru)
                r, u = tf.split(1, 2, ru)
            with tf.variable_scope("Candidate"):
```

```
                c = tf.nn.tanh(tf.nn.rnn_cell._linear([inputs, r * state],
                                                      self._num_units, True))
            new_h = u * state + (1 - u) * c
        return new_h, new_h
```

Vamos focar nas diferenças entre a GRU e a GRPU. Primeiramente o estado oculto da GRPU é dividido em duas partes, uma para o programador e uma para o controlador que opera da mesma forma que a GRU normal. Segundo que como queremos uma operação READ para alimentar dados da entrada diretamente na máquina virtual temos que adicionar o elemento a entrada do sistema. Dessa forma a entrada de cada passo é *c_state*, *p_state*, *c_input* e *p_input*, além das respectivas divisões dos resultados do *update* e *reset gates*.

Todas as adições ocorrerão no processamento de p_state através da lista de operações e valores *update* e *reset gates*, na forma de uma função que implementa 4.8, chamada *_operations*.

```
class GRPUCell(tf.nn.rnn_cell.RNNCell):
    """Data Stream Unit"""

    def __init__(self, num_units):
        self._num_units = num_units
        self._num_opers = 3
        self._state_size = num_units+self._num_opers

    @property
    def state_size(self):
        return self._state_size

    @property
    def output_size(self):
        return self._state_size

    def _operations(self, prog_state, inp_select, prog_inp, gru_input):
        pre_opers = [prog_inp, #load input
                     tf.expand_dims(prog_state[:,1],1),
                     tf.expand_dims(prog_state[:,2],1)]
        prog_state = tf.concat(pre_opers,1,name="pre_state")

        neutral_zero = prog_state*inp_select
        neutral_one = 1-inp_select+2*prog_state*inp_select-
```

```
                    prog_state*inp_select*inp_select

        opers = [prog_inp
                 tf.reduce_prod(neutral_one,1, keep_dims=True), #prod
                 tf.reduce_sum(neutral_zero, 1, keep_dims=True)] #sum
        return tf.clip_by_value(tf.concat(opers,1,name="post_state"),-5E2,5E2)

    def __call__(self, inputs, state, scope=None):
        gru_state, prog_state = tf.split(state, [self._num_units,
                                 self._num_opers], 1)
        print(inputs.get_shape())
        c_input, p_input = tf.split(inputs, [1,1], 1)

        with tf.variable_scope(scope or type(self).__name__):
            with tf.variable_scope("Gates"):.
                ru = rnn_cell_impl._linear([c_input, gru_state],
                                           2 * (self._state_size), True)
                ru = tf.nn.sigmoid(ru)
                r_gru, r_prog, u = tf.split(ru, [self._num_units,
                       self._num_opers, self._state_size], 1)
            with tf.variable_scope("Candidate"):
                c_gru = tf.nn.tanh(rnn_cell_impl._linear(
                [c_input, r_gru * gru_state], self._num_units, True))

                c_prog = self._operations(prog_state, r_prog, p_input, c_gru)
                c = tf.concat([c_gru, c_prog],1)

            new_h = (1-u) * state + u * c
        return new_h, new_h
```

Nessa máquina virtual foi criada 3 instruções na ALU, a primeira carrega o dado da entrada do programador, o que é executado antes das outras operações para os dados estarem imediatamente disponíveis. A segunda é uma simples operação agregadora de soma, e a última uma agregadora de produto. A entrada de cada uma delas é uma função do estado anterior do programa e o seletor de argumentos, e essa função são as citadas para elemento neutro zero e elemento neutro um. Com essas modificações temos implementada uma variante da GRPU em Tensorflow.

Para implementar a condicional, basta adicionar a correspondente operação de comparação na lista de operações basta dividir também o *update gate* nas seções do programador e do controlador, aplicar as seguintes linhas antes de concatenar os

campos do *reset gate*:

```
u_prog_final = tf.multiply(u_prog,(1+tf.expand_dims(u_prog[:,COND_OPER],1)*
(tf.expand_dims(c_prog[:,COND_OPER],1)-1)))
u = tf.concat([u_gru, u_prog_final],1)
```

Que implementa a equação 4.19.