UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Projeto de Dissertação de Mestrado

ANDRÉ FELIPE ZANELLA

Geração de Código Reduzido via Machine Learning

Maringá
2020

ANDRÉ FELIPE ZANELLA

Geração de Código Reduzido via Machine Learning

Projeto de dissertação apresentado ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Anderson Faustino da Silva

Maringá
2020

Geração de Código Reduzido via Machine Learning

RESUMO

lipsun.

Palavras-chave: words.

A definir

ABSTRACT

lipsun.

Keywords: words.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

ANN: *Artificial Neural Network*

SUMÁRIO

1

Introdução

Compiladores são peças centrais na ciência da computação, uma vez que estes funcionam como pontes entre o software e o hardware. Compiladores modernos apresentam otimizações que podem ser aplicadas em diferentes fases da compilação. Tais otimizações tem como objetivo gerar código semanticamente equivalente ao código fonte original, porém, maximizando ou minimizando alguns atributos do código gerado, com o objetivo de produzir código de melhor qualidade.

Alguns compiladores consolidados, como o GCC (GNU Compiler Collection) 2 e LLVM 3 (Low Level Virtual Machine), disponibilizam otimizações pré-definidas pelo projetista do compilador em forma de diretivas de compilação, chamados, O0, O1, O2, O3, onde cada um aplica uma sequência de otimizações em uma ordem definida. No entanto, nem sempre tais planos geram o melhor código possível, ocasionando na necessidade de criação por humanos de planos de compilação especializados. Tal fato deu origem a técnica de compilação iterativa. Nesta abordagem, sequências de transformações são sucessivamente aplicadas a um programa e sua performance é medida através de sua execução a fim de determinar o melhor plano.

Um sistema de *autotuning*, é um sistema que emprega compilação iterativa acompanhado de um algoritmo gerador de sequências. Estes modelos apesar de encontrarem boas sequências, são custosos, pois necessitam de compilação e execução. Além disso, o grande espaço de busca de transformações torna a tarefa mais complexa. Sistemas modernos de autotuning utilizam diversas técnicas de aprendizagem de máquina para minimizar este problema, selecionando de forma inteligente os pontos que mais otimizam a tarefa de busca. Redes neurais artificiais são uma classe técnicas de aprendizagem que tem se mostrado uma solução generalizada para problemas de classificação, tendo

exito em diversas áreas clássicas de inteligencia computacional. Em sistemas de alto desempenho redes neurais conseguem ser mais eficientes que sistemas de autotuning, pois eliminam a necessidade de executar a aplicação e são adaptáveis para qualquer problema de otimização.

Aprendizagem de máquina pode auxiliar na geração de um código de qualidade superior ao código que o compilador tradicionalmente é capaz de produzir, uma vez que algoritmos são capazes tomar decisões que tornam o processo de otimização eficiente. Este fato ajudou a estabelecer uma direção de pesquisa voltada para aplicações destas técnicas na construção de compiladores. Outra questão que impulsionou o campo é o fato de exclusão de expertes na criação de planos de compilação, substituindo a intuição e experimentação manual destes, pela avaliação empírica de dados guiada por heurísticas.

A tarefa essencial da compilação consiste na escolha de uma boa sequência de transformações. Com este foco, o processo de criar e avaliar um sistema de compilação guiado por aprendizagem de máquina pode ser estruturado em fases. A fase inicial consiste em definir as características qualitativas que irão representar o programa, estas são chamadas de *features*. Uma boa engenharia de *features* é essencial para determinar a qualidade do sistema, uma vez que o modelo de aprendizagem faz uso destes dados para determinar o comportamento do programa. A próxima fase é o treinamento do modelo utilizando um algoritmo de aprendizagem. Existem diversos algoritmos para esta tarefa, cada um especializado para uma classe de problemas. Por exemplo, algoritmos de clusterização são utilizados para agrupar programas considerados similares, enquanto que técnicas de aprendizagem supervisionada, como redes neurais, são adequadas para problemas de classificação, como estimativas de performance. Por fim, o modelo deve ser capaz de tomar boas decisões de otimização para novos programas não observados.

Diversos estudos posteriores se dedicaram a melhoria de sistemas de *autotuning* incorporando técnicas de aprendizagem de máquina a fim de minimizar a busca por pontos no espaço. Muitos sub-campos de aprendizagem de máquina existem e podem ser utilizados no contexto de geração de código e otimização. Revisões sistemáticas são apresentas por Ashouri *et al* (Ashouri et al., 2018) e Wang (Wang e O'Boyle). Nesta dissertação estudamos dois paradigmas de aprendizagem que vem ganhando popularidade em sistemas de computação de alto desempenho, são estes: Aprendizagem Profunda (Deep Learning) e Aprendizagem por Reforço Profundo para *autotuning* de código.

Redes Neurais Artificiais (ANNs) são os componentes chave que definem Aprendizagem Profunda. Estas redes, quando conectadas em diversas camadas, são poderosos modelos de aprendizagem para problemas de classificação e regressão, pois são capazes de aprender relações complexas entre variáveis e promovendo avanços em áreas como

visão computacional, processamento de linguagem natural e mais recentemente vem ganhando popularidade dentro das áreas de pesquisa de linguagens de programação e compiladores (Nielsen, 2018). Teoricamente, qualquer função pode ser aproximada com precisão arbitrária por uma ANN (Goodfellow et al., 2016).

Aprendizagem por Reforço Profundo (ARP) é o modelo de aprendizagem adotado pelo famoso *AlphaGo*, o primeiro algoritmo a vencer uma pessoa no clássico jogo Go, e o recente *AlphaaFold*, responsável pela solução do problema de desdobramento de proteínas, respectivamente. O que torna esta abordagem suscetível a aplicação em problemas complexos é a sua capacidade de tomada de decisão visando a recompensa a longo prazo. Trabalhos recentes utilizam ARP para solução de problemas de Computação de Alto Desempenho (Haj-Ali et al., 2019).

Nesta dissertação buscamos utilizar as técnicas citadas para mitigação do clássico problema de predição em compilação. Diversas pesquisas buscam otimizar o desempenho da aplicação com relação ao tempo de execução (Ashouri et al., 2018), no entanto, com o avanço da Internet das Coisas (IoT) e sistemas embarcados, a preocupação com o tamanho dó código gerado não pode ser mais ignorada. Neste trabalho buscamos utilizar Aprendizagem Profunda e Aprendizagem por Reforço profundo aplicadas ao problema de prever a capacidade de redução de código que uma sequência de otimizações pode prover. Desenvolvemos dois *frameworks* de aprendizagem e trabalhamos com duas representações de código distintas, em um conjunto de 30 *benchmarks* embarcados para treino. Com esta abordagem, procuramos trazer novos *insights* na área de compilação preditiva. Por fim, comparamos nossa metodologia com dois trabalhos estados da arte em redução de código e compilação preditiva.preditiva. Por fim, comparamos nossa metodologia com dois trabalhos estados da arte em redução de código e compilação preditiva.

1.1 Desafios

A pesquisa por estratégias de compilação é dividida em dois principais problemas: o *Problema de Seleção de Fase* (PSF)[1], que busca encontrar o melhor conjunto de transformações para um dado programa, sem levar em consideração a ordem em que são aplicadas, e o *Problema de Ordenação de Fase* (POF), que busca, além de se encontrar o melhor conjunto de otimizações, encontrar a melhor ordem com que estas são aplicadas. O restante dessa seção discorre a respeito dos desafios existentes para responder estas duas questões.

[1]Fase é um sinônimo de sequência, porém, neste caso a palavra *sequência* não indica uma ordem.

Escassez de Dados e Inferência de programas. O desafio mais eminente que limita a performance de modelos de otimização de programas é qualidade dos dados de treino. Apesar de existirem inúmeros conjuntos de *benchmarks* disponíveis estes não são comparáveis com a diversidade de programas que um compilador usualmente tem que lidar. Este fato tem efeitos diretos no treinamento de modelos, uma vez que o conjunto de treino não é capaz representar significativamente o espaço de programas, levando a questões de como prever o comportamento de um programa baseado em um certo conjunto utilizado pelo modelo.

Aprendizagem por transferência é um método de aprendizagem de máquina que consiste em armazenar conhecimento adquirido ao solucionar um problema e utiliza-ló para solução de outro problema. Está técnica foi utilizada por Cummins *et al* (Cummins et al., 2017) para solucionar dois problemas de otimização de domínios diferentes: mapeamento de hardware e granularidade de threads. Redes neurais profundas tem a habilidade de criar abstrações e relações representativas de programas podendo ser capaz de criar abstrações de um domínio ou conjunto de programas e generalizar para outro, no entanto, um modelo que possa ser utilizado para todos os domínios de otimização e programas contínua um desafio de pesquisa.

Representação de Programas. Algoritmos de aprendizagem de máquina aprendem a correlacionar variáveis que alimentam o motor de aprendizagem. Para uma caracterização precisa de um programa é necessário um conjunto de características qualitativas que representem o seu comportamento semântico. Existem inúmeras features diferentes que podem ser utilizadas. Estas geralmente se enquadram em duas categorias: *estáticas*, que são estruturas de dados extraídas do código fonte ou representação intermediária do programa, e *dinâmicas*, obtidas durante a execução execução da aplicação.

Os problemas enfrentados na tarefa de engenharia de features para otimização de código são diversos; alguns deles são: Escolha da representação estática, dinâmica ou uma combinação das mesmas. Além disso, estas podem ser representadas por estruturas de grafo ou vetores. Redução de dimensionalidade. vetores podem ter alta dimensionalidade e por isso, conter informações irrelevantes. Analise de componentes principais (PCA) ou coeficientes de similaridade podem auxiliar na exclusão de características irrelevantes.

Além dos problemas de engenharia de features, outra questão levada em consideração é se a escolha das features deve ser guiada por expertes ou pelo próprio modelo, isto é, a partir de uma representação sintática do código, como *tokens* da linguagem fonte ou representação intermediária por exemplo, o modelo de aprendizagem descobre quais são as features mais representativas, automatizando tarefas como identificação de features muito similares e não sendo dependente do problema em questão. Esta técnica só é

possível graças a redes neurais profundas que são capazes de identificar relações entre códigos de programação (Allamanis et al., 2014).

Combinação de modelos Talvez um dos maiores desafios da compilação guiada por aprendizagem de máquina seja a combinação de diferentes modelos de aprendizagem. Em aprendizagem por reforço o algoritmo tenta aprender como maximizar a recompensa de uma função de valor, isto é, dado um input, o algoritmo precisa aprender quais as melhores decisões a serem tomadas. Este modelo de aprendizagem é muito intuitivo e se assemelha com o aprendizado por seres humanos, podendo criar modelos com excelente precisão, no entanto, aprendizagem por reforço depende do ambiente e se estive possuir muitas possibilidades de decisão pode tornar o treinamento extremamente custoso. Esta abordagem diverge de redes neurais em que os dados de entrada e saída são conhecidos. Combinar aprendizagem por reforço com aprendizagem profunda para resolver o problema de compilação é uma questão em aberto.

Aprendizagem ativa é um método de aprendizagem que pode ser utilizado para minimizar o custo do treinamento. Técnicas de aprendizagem profunda necessitam de grandes volumes de dados (LECUN et al., 2015) tornando o processo de compilação e execução grandes consumidores de tempo. Trabalhos como de Rosário *et al* (Rosário *et al., 2020*) utilizam 30 mil planos de compilação diferentes e 300 programas para treinamento. Aprendizagem ativa foi explorado no trabalho de (Ogilvie et al., 2017) para minimizar o custo da compilação iterativa, no entanto, nenhum trabalho que combine aprendizagem ativa e aprendizagem profunda para atacar o problema de compilação foi encontrado.

O foco deste trabalho esta no uso de redes neurais profundas para conduzir a criação de código otimizado

1.2 Objetivos e Contribuições

O objetivo desta dissertação é apresentar técnicas baseadas em aprendizagem de máquina para auxílio de criação de heurísticas de compilação focadas em redução de código. As principais contribuições são as seguintes.

- Um abordagem inédita capaz de prever o comportamento de planos de compilação com relação ao tamanho de código, a partir do código fonte da aplicação, sem necessidade de avaliação de uma representação intermediária, focando apenas na árvore sintática do programa (AST).

- Um estudo comparativo entre o uso de uma representação de código extraída a partir da AST e uma representação a partir da Grafo de Fluxo de Controle e Dados por Redes Recorrentes para criação de heurísticas de compilação.
- Demostração do potencial uso de Aprendizagem por Reforço Profundo para geração de código reduzido, superando planos de compilação padrão de compiladores consolidados.
- Avaliação do desempenho do nosso sistema em um conjunto de dados de mais de 30 mil programas gerados a partir de 30 *benchmarks* específicos para sistemas embarcados e profundamente embarcados e 1290 sequências de compilação especializadas em redução de código.
- Disponibilização do sistema como uma plataforma aberta.

1.3 Organização do texto

No Capítulo 2 é apresentado uma breve revisão da literatura dos conceitos utilizados nesta dissertação. O Capítulo 3 reúne os principais trabalhos relacionados a aprendizagem de máquina e compiladores. No capítulo 4 apresentamos nossa proposta, a formulação do problema, a metodologia e os materiais utilizados. No capítulo 5 discutimos os resultados encontrados a partir dos experimentos propostos, e por fim, o Capítulo 6 apresenta as considerações finais e possíveis futuras direções de trabalho.

2

Fundamentos Teóricos

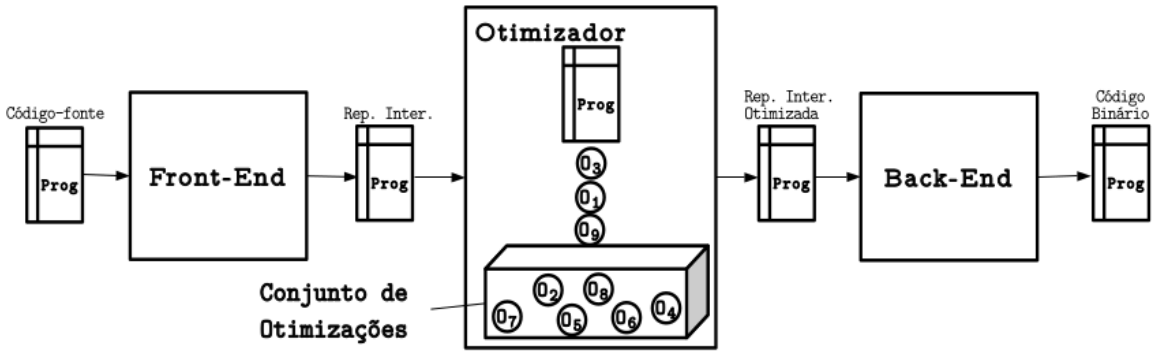
Neste capítulo é fornecida uma visão intuitiva e global do funcionamento de sistemas de compilação otimizantes, algoritmos de aprendizagem e as técnicas de avaliação utilizadas nesta dissertação.

2.1 Compilador Otimizante

Segundo Souza (2014) uma *transformação* ou *otimização* é um algoritmo aplicado pelo compilador, que transforma o programa fonte em um programa alvo semanticamente equivalente, porém mais eficiente. Transformações podem ser classificadas de duas formas: transformações de *análise* e *otimizantes*. A primeira tem como objetivo apenas coletar e armazenar informações a respeito do programa, enquanto que a segunda modifica o código e pode fazer uso das informações armazenadas para otimizar a aplicação. Uma *sequência* ou *fase* é um subconjunto do conjunto de transformações presentes em um compilador.

Um *compilador otimizante* é um compilador que apresenta um conjunto de transformações de análise e otimização para serem utilizadas durante o processo de compilação. O processo de compilação é composto pelas seguintes etapas: *Front-End*, e *Back-End*. O *Front-End* representa a fase de análise do código fonte, enquanto que no *Back-End*, ocorre a síntese do código fonte para código binário executável para a arquitetura-alvo. No entanto, compiladores modernos não realizam essa tradução diretamente. Entre as duas fases descritas acima, existe um gerador de *Código Intermediário* e um *Otimizador* (TORCZON e COOPER, 2011). A Figura - 2.1 apresenta a arquitetura de um compilador otimizante.

Figura 2.1: Arquitetura de um compilador moderno.



Fonte: (XAVIER, 2014).

Uma representação intermediária (IR) é uma estrutura de dados que representa uma abstração da linguagem de máquina, que é independente da arquitetura-alvo em questão, facilitando assim, a portabilidade da linguagem. Um otimizador tem a função de aplicar uma sequência de transformações fornecidas pelo usuário, ou uma estratégia já definida no compilador, a fim de melhorar o desempenho de uma aplicação. Transformações podem ser aplicadas ao programa em qualquer fase de compilação, no entanto, a grande maioria é feita no código IR. Compiladores modernos, como o GCC, e a LLVM, possuem otimizadores e suas próprias representações intermediárias de código (JUNIOR, 2016; XAVIER, 2014).

O espaço de busca de transformações é extremamente grande, uma vez que um compilador como o GCC, por exemplo, possui centenas de transformações. O problema de seleção de fase (PSF) é um problema de otimização, que busca encontrar um conjunto de transformações de compilação que proporcionem o maior desempenho para um programa. O desempenho de um programa pode ser definido de acordo com os objetivos que se deseja minimizar, podendo ser o tempo de execução ou tamanho de código gerado, por exemplo.

Para ilustrar a complexidade do espaço de busca, suponha que O seja o conjunto de todas as transformações de um compilador C, onde $O = \{o_1, o_2, \ldots, o_n\}$ tal que o_i, representa uma transformação. O conjunto de partes (conjunto potência) $P(O)$ contém todos os possíveis subconjuntos de transformações de O e possui cardinalidade 2^n, ou seja, possui complexidade exponencial. A fim de definir a sequência ótima, o compilador precisa avaliar todas as possibilidade. Existem ferramentas que procuram encontrar a forma canônica de uma aplicação, isto é, a sequência ótima. Tais sistemas são chamados de *Superotimizadores* (JUNIOR, 2016; MASSALIN, 1987).

Estudos anteriores mostram que as interdependências e a interação entre habilitar e desabilitar transformações em uma sequência pode alterar drasticamente o desempenho

de um programa (AGAKOV et al., 2006). Quando levada em consideração a ordem de aplicação das transformações em um conjunto, se define o problema de ordenação de fase (POF), que pode ter uma complexidade maior que 2^n, uma vez que selecionado um subconjunto com m transformações, tal que $m \leq n$, existem $m!$ arranjos diferentes para este subconjunto.

Transformações de código permitem que o programa se beneficie de melhorias sem alteração do código fonte original. No entanto, devido aos motivos já apresentados, definir manualmente uma sequência para um programa em específico não é uma tarefa viável. A próxima seção apresenta os métodos utilizados para encontrar as melhores transformações automaticamente.

2.2 Redes Neurais Atificiais

Técnicas de aprendizagem supervisionada e não-supervisionada tem sido exploradas na otimização de compiladores Aprendizagem supervisionada é um modelo de aprendizagem de máquina no qual é estabelecida uma relação entra os valores do vetor de entrada e saída correspondente. Este modelo é muito utilizado na resolução de problemas de regressão e classificação, pois correlações entre os dados de entrada e as decisões de otimização tomadas para se obter a melhor performance são estabelecidas. Em aprendizagem não-supervisionada não se tem conhecimento a respeito do valor de saída de cada entrada.

Deep learning é um poderoso framework para aprendizagem supervisionada que tem mostrando bons resultados nas mais diversas áreas do conhecimento pois é capaz de aprender relações complexas entre os dados. Este modelo de aprendizagem pode ser utilizado como uma ferramenta efetiva para extrair tal conhecimento do espaço de otimização de código. As próximas subseções tratam dos algoritmos utilizados nesta dissertação.

2.2.1 Redes Múltiplas Camadas Alimentadas Adiante

Redes feed-forward também conhecidas como multilayer perceptrons são redes neurais compostas por múltiplas camadas alimentadas adiante, isto é, a rede consiste de uma camada de entrada e uma mais camadas ocultas de nós, onde o sinal ou informação é propagado camada por camada, constituindo um mapeamento de um conjunto de variáveis para uma resposta. A Figura - 2.2 mostra uma arquitetura de uma rede alimentada a diante.

Estas redes são consideradas aproximadores universais, pois definem um mapeamento $y = f(x;\theta)$ que aproxima alguma função f^*, onde θ são os parâmetros de aproximação aprendidos pela rede. Estas servem como base para oturos modelos de redes neurais, devido as seguintes características:

- Cada neurônio na rede inclui uma função de ativação não-linear $\varphi(v)$ para descrever o sinal de cada neurônio.
- Contém uma ou mais camadas de neurônios ocultos que capacitam a rede a aprender tarefas complexas.
- Exibe um alto grau de conectividade, isto é, um neurônio pode mapear sua saída para todos os neurônios da camada subsequente.

Um vetor de entrada n-dimensional $\overline{x}$ pode ser transformado em uma resposta de saída utilizando as seguintes equações:

$$\overline{h}_1 = \varphi(W_1^T \overline{x}) \tag{2.1}$$

$$\overline{h}_{p+1} = \varphi(W_{p+1}^T \overline{h}_p) \tag{2.2}$$

$$\overline{o} = \varphi(W_{k+1}^T \overline{h}_k) \tag{2.3}$$

onde (2.1) é a equação *Entrada para Camada Oculta*, (2.2) *Oculta para Camada oculta* e (2.3) *Oculta para Camada de Saída*, W_p é a matriz de pesos, $\overline{h}_p$ é o vetor coluna de sinais da camada p.

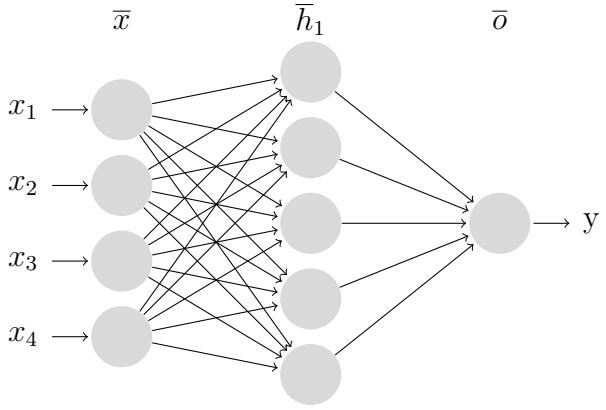


Figura 2.2: Rede neural artificial alimentada adiante composta por uma única camada oculta de 5 neurônios, ou nós, e uma camada de saída com um único neurônio onde. Cada aresta representa a saída de informação de um neurônio que serve como entrada para o neurônio da cada subsequente.

Algumas funções de ativação comumente utilizadas na literatura são as seguintes: Função *Limiar* 2.4

$$\varphi(v) = \begin{cases} 1, & \text{se } v \geq 1 \\ 0, & \text{se } v < 0 \end{cases} \tag{2.4}$$

Função *Sigmóide* 2.5. Esta é definida como uma função estritamente crescente com um comportamento balanceado entre linear e não-linear.

$$\varphi = \frac{1}{1 + exp(-av)} \tag{2.5}$$

Variando o parâmetro a se obtém funções sigmoides com diferentes inclinações.
Função *ReLU (Unidade Linear Retificada)* 2.6.

$$\varphi(v) = max(0, v) \tag{2.6}$$

Esta função largamente utilizada para redes multi-camada devido sua eficiente computação e sua ativação esparsa, por exemplo, para valores aleatórios no intervalo $[-1, 1]$, apenas 50% dos neurônios com ativação ReLU serão ativados.

2.2.2 Aprendizagem Baseada em Descida de Gradiente

Redes neurais usualmente são treinadas utilizando métodos iterativos, baseados em descida de gradiente, pois são capazes de derivar a função de custo a um valor muito pequeno. Funções de custo em redes com multiplas camadas são definidas como composições de funções de peso de camadas anteriores. Para minizar este problema o algoritmo de retropropagação é utilizado extensivamente na literatura (Nielsen, 2018).

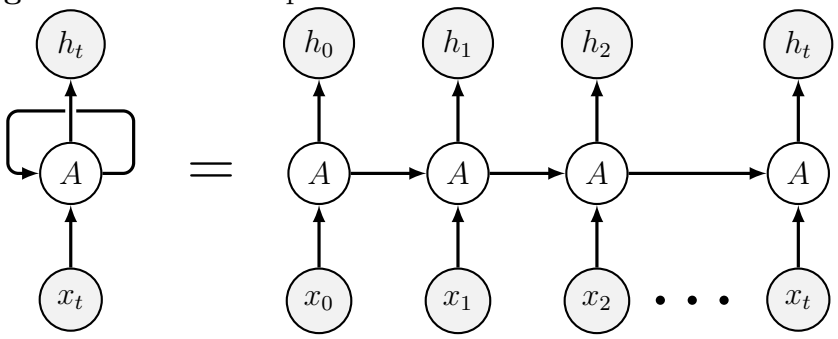
Redes alimentadas adiante são inicializadas com pequenos valores de peso aleatórios e com bias nulo ou um pequeno valor positivo. Inicialmente, a computação inicialmente acorre de forma *avante*, isto é, sequencialmente através das camadas até a camada de saída. O resultado final $\hat{y}$ é comparado com o valor da instância de treino y e computado a perda $\mathcal{L}(\hat{y}, y)$. Após este passo, se da inicio a fase *retrograda* que consiste em computar o gradiente da função de custo com respeito aos diferentes pesos de todas as camadas, iniciando o processo pela camada de saída. Os gradientes são utilizados para atualizar os valores dos pesos.

2.3 Redes Neurais Recorrentes

Redes Neurais Artificias (**ANN**) se tornaram um importante campo de pesquisa em aprendizagem de máquina, uma vez que estas redes apresentam bons resultados em diversas áreas da inteligência artificial. Grandes avanços científicos foram obtidos com a utilização de redes neurais profundas (Senior et al., 2020) (Silver et al., 2016), e mais recentemente estas vem desempenhando um papel essencial na otimização de compiladores e sistemas de alto desempenho (Wang and O'Boyle, 2018). Uma revisão básica dos principais conceitos sobre Redes Neurais é descrito no Apêndice A.

Redes Neurais Recorrentes (**RNN**) formam uma classe especial de ANNs cujo as conexões entre as unidades formam um ciclo direcionado. Estes ciclos permitem uma simulação dinâmica temporal, isto é, possuem uma memória interna para o processamento sequencial de dados. Graças a esta natureza, os dados de entrada e saída trabalham de forma dependente neste tipo de rede, fazendo destas, muito eficientes em tarefas de aprendizado de associação temporal em sequências para classificação e predição.

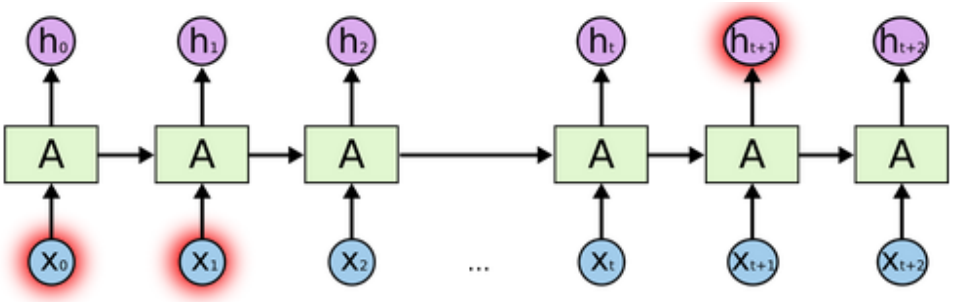
Figura 2.3: Fluxo de processamento de uma unidade recorrente.



2.3.1 O Problema das Dependências de Longo Prazo

Muitas aplicações importantes em inteligência artificial requerem o conhecimento de dependências à longo prazo em eventos de uma sequência. Teoricamente este problema pode ser solucionado com a técnica de Descida de Gradiente e *Backpropagation Through Time* (BPTT). No entanto, computar a descida de gradiente com BPTT quando se deseja aprender dependências de longo prazo em muitos casos pode levar a dissipação ou explosão do gradiente durante o treino (Trinh et al., 2018). A Figura - 2.4 ilustra a relação dependência entre termos os termos em uma unidade recorrente.

Figura 2.4: Dependência entre termos em RNNs.[2]

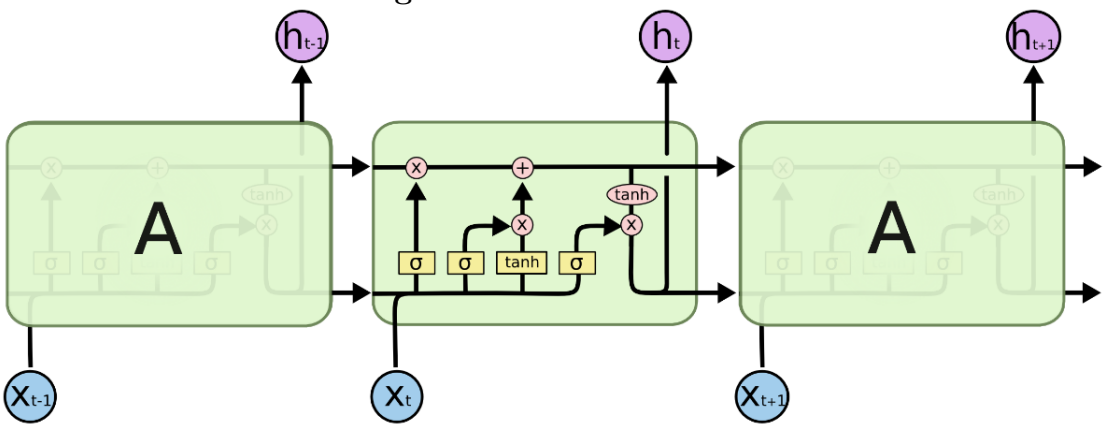


Diversos métodos promissores para mitigar o problema de dependências de longo prazo foram propostos. Estas técnicas são capazes de reter ou "relembrar"informações por longos períodos de tempo e não apresentam a dificuldade de aprendizagem de redes RNN básicas. Não seção 2.3.2 e 2.3.3 apresentamos duas destas abordagens.

2.3.2 *Redes Long Short Term Memory*

Existem diferentes abordagens para lidar com o problema de dependências de longo prazo, a mais popular delas são as Redes *Long Short Term Memory* (**LSTM**), um tipo especial de RNN capaz de aprender estas depedências. Redes LSTM foram criadas em 1997 por Hochreiter e Schimdhuber, e são compostas por uma célula de memória e três portas multiplicativas, denominadas como porta de *entrada*, *saída* ou *inferência* e *esquecimento*.

Figura 2.5: Células LSTM



Assim como em RNNs, LSTMs também possuem recorrência, porém, ao invés de uma única camada, LSTMs possuem quarto, e interagem de maneira diferente. Na Figura - 2.5,

[2]Esta figura e a figura 2.4 foram retiradas da página web: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

cada linha representa um vetor de entrada ou saída. Os círculos de cor rosa, representam operações vetoriais, e os retângulos amarelos são camada de rede que foram aprendidas. Linhas mescladas denotam concatenação, enquanto que linhas com bifurcações indicam que o conteúdo esta sendo copiado e propagado para locais diferentes.

- **Porta de Esquecimento**: a função $f_t = \sigma(W_f.[h_{t-1}, x_t + b_f])$ representa qual informação será descartada pela célula.
- **Porta de Entrada**: através de uma função *Sigmoid* $i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$ decide quais valores serão atualizados. Uma camada *tanh* cria um vetor de candidatos $\hat{C}_t = tang(W_C.[h_{t-1}, x_t + b_C])$ que poderia ser adicionado ao estado da célula, e este é atualizado $C_t = f_t \odot C_{T-1} + i_t \odot \hat{C}_t$.
- **Porta de saída**: uma camada sigmoid $o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$ e o estado da célula, decidem quais partes da serão propagadas para saída $h_t = o_t \odot tanh(C_t)$.

2.3.3 *Gated Recurrent Unit*

Gated Recurrent Unit (**GRU**) foram propostas por Cho *et al* (2014). Este modelo, assim como LSTM, faz uso de componentes de portas. GRU combina as portas de esquecimento e entrada em uma única porta e introduzem a porta de *update* u e a porta *reset* r.

A porta r controla quais partes do estado são utilizadas para computar o próximo estado, adicionando um efeito não-linear na relação de estado passado e estado futuro. A porta u atua como um integrador condicional, podendo optar por copiar a informação ou ignorá-la completamente.

2.4 Aprendizagem por Reforço

Aprendizagem por Reforço (**RL**) é uma das técnicas promissoras em Aprendizagem de Máquina (). RL é uma classe de problemas de decisão que envolvem agentes de *software* tomando decisões em um ambiente, geralmente modelado como um Processo de Decisão Markoviano (MDP), com o objetivo de maximizar a recompensa a longo prazo. MDPs são uma formalização matemática que modelam o processo sequencial de tomada de decisões, isto é, um mapeamento de decisões tomadas pelo agente, baseadas no estado atual do ambiente, para a melhor recompensa esperada.

Um MDP é definido como uma 4-tupla $(S, A, P(.), R(.))$, onde S é o conjunto de estados do ambiente, A é o conjunto de ações, $P(s_{t+1} = s^{'}, r_{t+1} = 1 | s_t = s, a_t = a)$ é a probabilidade de transição de estados que especifica a dinâmica do modelo, e $R(s_t, a_t) =$

$\mathbb{E}[r_t|s_t = s, a_t = a]$ é a função recompensa. Tomando a ação a no estado s é esperado a recompensa r_t.

O objetivo do RL é encontrar uma política decisões π que maximize a recompensa a longo prazo. Dado um *timestep* $t \in T$, o agente recebe um subconjunto de estados $s_t \in S$ e seleciona um subconjunto de ações $a_t \in A$. O mapemanto do estado s_t para a ação é dado pela distribuição $\pi(a_t|s_t) = p(A_t = a|S_t = s)$. Como consequência de suas ações, o agente recebe uma recompensa $r_{t+1} \in R$ e se desloca para o estado s_{t+1}. Dessa forma, o agente o MDP dão origem a sequência $S_0, A_0, R_1, S_1, A_1, R_2 \ldots$. O *horizonte* é definido como o número de *timesteps*. Em um horizonte finito, dizemos que o MDP é finito. A **??** descreve o ambiente de interação em um Processo Markoviano.

Definimos o *retorno* do modelo como sendo a soma descontada das recompensas a partir de t até o horizonte, isto é, $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \ldots$, onde $\gamma \in [0, 1]$ é o fator de desconto. Este fator quantifica a importância que damos ao modelo para recompensas a longo prazo, por exemplo, um fator $\gamma = 0$, leva em consideração apenas a recompensa imediata. Por fim, a função *valor* é definida como o retorno esperado no estado s com a política π, e é definida da seguinte forma: $V(s) = \mathbb{E}[G_t|s_t]$.

Dessa forma o problema de Aprendizagem por Reforço é a melhoria da política de decisão, a fim de maximizar o retorno esperado. A política ótima é definida como: $\pi^* = argmax_\pi \mathbb{E}[R|\pi]$.

2.4.1 Aprendizagem por Reforço Profundo

O paradigma de Aprendizagem por Reforço Profundo (**DRL**) utiliza Redes Neurais Artificiais para aproximar qualquer um dos seguintes elementos de RL: política de decisões $\pi_\theta(a|s)$, função valor $\hat{v}_\theta(s)$ ou função-Q, e o modelo com relação a transição de estados e função recompensa. Neste contexto os parâmetros θ são os pesos da Rede Neural Artificial.

Nesta dissertação direcionamos nossos interesses em estudar a aproximação para a política de decisões e a função-Q, introduzida na seção **??**.

2.4.2 Políticas de Gradiente

Métodos de Política de Gradiente **(PG)** são direcionados para a modelagem e otimização direta da política de decisão. A política é geralmente modelada através de uma função parametrizada a respeito de θ, $\pi_\theta(a|s)$. A função valor da recompensa depende da política

e vários algoritmos podem ser utilizados para otimizar θ para a melhor recompensa. A função recompensa é definida da seguinte forma:

$$J(\theta) = \sum_{s \in S} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \pi_{\theta}(a|s) Q^{\pi}(s,a), \tag{2.7}$$

onde $d^{\pi}(s)$ é a distribuição estacionária da Cadeia de Markov para a política π_{θ}, e $Q^{\pi}(s,a)$ é a função Ação-Valor que tem como *output* o retorno esperado de um par (s,a) sob a política π. A política é atualizada através de diferenciação do termo $\nabla_{\theta} J(\theta)$ e atualizando os parâmetros da Rede Neural conforme a direção do gradiente:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J, \tag{2.8}$$

Dezenas de algoritmos foram propostos para computar o gradiente e apresentar todos estes esta fora do escopo desta dissertação. Focamos em discorrer apenas a respeito de um deles, devido ao fato deste ter sido a opção adotado por trabalhos recentes na área de Otimização de Sistemas.

Política de Otimização Proximal (PPO) é um algoritmo PG utilizado para um comportamento mais determinístico, estável e robusto, através da limitação das atualizações e garantindo que o desvio da política anterior não seja grande. PPO foi testando em um conjunto de *benchmarks* te sem provado eficiente na produção de bons resultados com uma grande simplicidade.

Diferente de uma abordagem de um algoritmo PG *vanilha*, PPO habilita atualizações de *mini-batchs* em múltiplas épocas, melhorando a complexidade da amostra. A técnica consistem em amostrar os dados por meio de interação com o ambiente e otimização a função objetivo utilizando subida de gradiente estocástica. O algoritmo executa atualizações que maximizam a função recompensa enquanto garante que o desvio da política anterior é pequeno através do uso de uma função objetivo substituta.

A função de perda em PPO é definida da seguinte forma:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.9}$$

onde $r_t(\theta)$ é definido com a razão probabilística $\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta}(a_{t_{old}}|s_t)}$, tal que r. Este termo penaliza a atualização da política de $r(\theta_{old})$ para $r_t(\theta)$ $\hat{A}_t$ denota a vantagem estimada que aproxima o quão bom a_t é em comparação com a média. O segundo termo na função *min* atua como um desincentivo para mover r_t do intervalo $[1-\epsilon, 1+\epsilon]$.

2.5 Modelos de Representação de Código

Algoritmos de Aprendizagem de Máquina dependem de um conjunto de característica quantitativas, ou *features*, para representar programas. Existe uma enorme variedade de *features* que podem variar entre estáticas ou dinâmicas e podem ser extraídas manualmente ou pro alguma metodologia que automatize o processo (Wang and O'Boyle, 2018). Diversos problemas devem ser mitigados afim de uma seleção de *features* apropriadas para o problema em questão. Para uma seleção adequada, questões como a classe a qual estas pertencem (estática, dinâmica), a estrutura de dados (vetor, grafo), técnicas de correlação e redução de dimensionalidade devem ser exploradas.

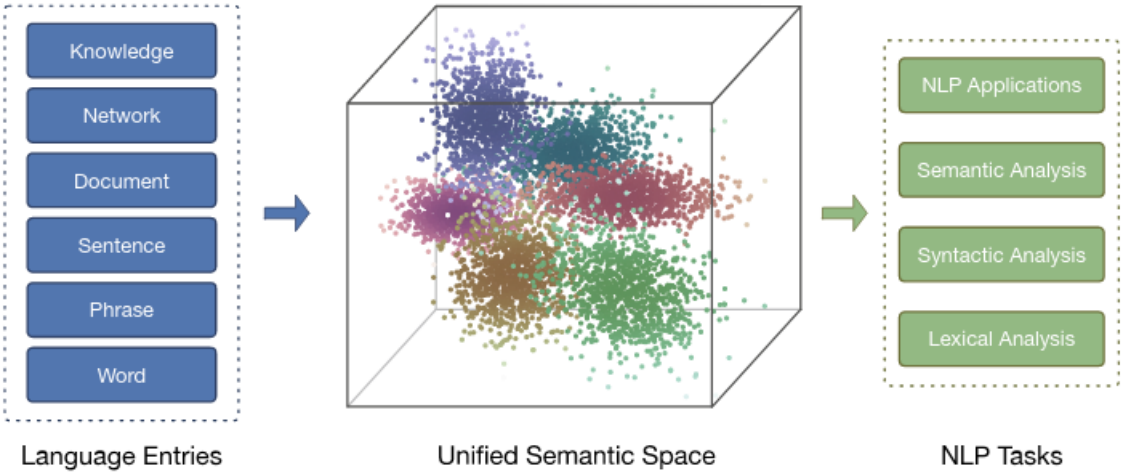
Uma alternativa a esta abordagem de coleta de características, é utilizar uma *Representação Distribuída* de código é deixar nas mãos do modelo a descoberta das *features* mais relevantes.

2.5.1 Representações Distribuídas

Representações Distribuídas ou *embeddings* são largamente utilizadas em PLN para codificar elementos de linguagem natural. Neste paradigma de representação, todos os objetos de interesse são projetados em um espaço semântico unificado de baixa dimensão, isto é, cada entidade é representada por um padrão de ativação distribuído sob múltiplos elementos (Liu et al., 2020).

Representações distribuídas tem se mostrado eficientes, pois, usualmente possuem dimensões baixas que preveem problemas de esparses de dados, sendo mais compactas que abordagens *one-hot* (Liu et al., 2020). Modelos probabilísticos que utilizam vetores de representação distribuída aprendem uma função da forma $c \rightarrow \mathbb{R}^D$ que mapeiam elementos do código para vetores D-dimensionais de números reais (Allamanis et al., 2018). A Figura - 2.6 representa este mapeamento.

Figura 2.6: Representação Distribuída de entidades para tarefas de PLN.

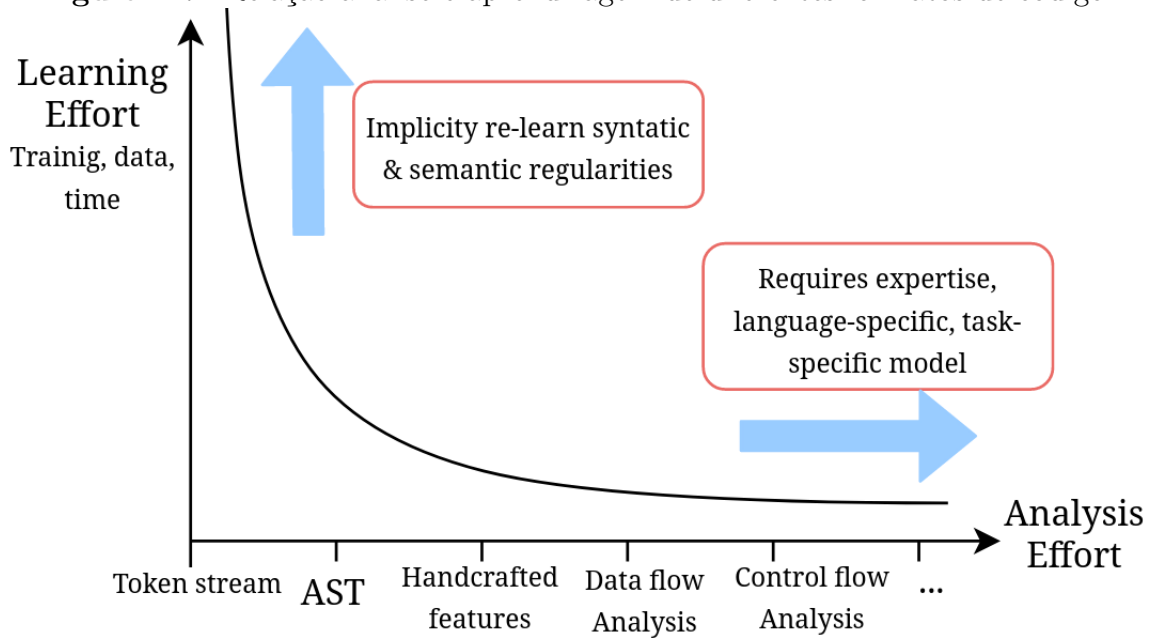


Fonte: (Liu et al., 2020).

Inspirado no sucesso de técnicas de *Processamento de Linguagem Natural* (PNL), novos trabalhos em linguagens de programação buscam explorar o uso de vocabulários e *embeddings*. Este último é um mapeamento de variáveis discretas ou categóricas para números contínuos. No contexto de PNL, *embeddings* são vetores reais de baixas dimensões, onde cada valor desde vetor representa uma palavra ou frase. Esta metodologia de criação de vetores representando programas também é conhecida como Representação Distribuída de Programas.

Diferente de linguagem natural, linguagens de programação são bem estruturadas, pois são formalizadas em termos de sintaxe e semântica. A complexidade de representação destas linguagens emerge, uma vez que o processo de compilação geralmente envolve diversas representações intermediárias de código, independente ou dependente de arquitetura alvo, até a geração final do binário. Portanto, existem diversas maneiras de criação de vocabulários e *embeddings*. A Figura - 2.7 apresenta a relação código, complexidade de aprendizagem e análise em diferentes etapas da compilação. A Figura - 2.7 apresenta a relação código, complexidade de aprendizagem e análise em diferentes etapas da compilação.

Figura 2.7: Relação análise e aprendizagem de diferentes formatos de código.



Fonte: (Alon et al., 2018).

2.6 Técnicas de Avaliação

Esta seção descreve as técnicas utilizadas para avaliar os métodos de aprendizagem de máquina descritos nas seções anteriores

2.6.1 Método de Validação Cruzada *k-fold*

Validação cruzada é um método estatístico utilizado para avaliar a capacidade de generalização de um sistema através da divisão dos dados em dois seguimentos, um para treinar e outro para validar. Idealmente, com um número suficiente de dados, não seria necessário validação. Quando não possível, a validação cruzada deve buscar, em rodadas sucessivas, deve tentar validar todos os pontos do conjunto de dados.

O método *k-fold* particiona o conjunto de dados em k conjuntos mutualmente exclusivos de mesma cardinalidade, a partir daí, um subconjunto é utilizado para teste e os $k-1$ conjuntos, para estimação de parâmetros. O processo é realizado k vezes, alternando de forma circular, os conjuntos para teste.

2.6.2 Matriz de Confusão

Uma matriz de confusão (Kohavi e Provost, 1998) contem informações a respeito dos valores previstos e dos valores esperados obtidos por um sistema de classificação. É uma matrix de tamanho LxL, onde l é o número de valores rotulados.

2.6.3 Precisão e Erro Absoluto Médio

Afim de definirmos a métricas de precisão devemos definir primeiramente os seguinte conceitos:

- **Verdadeiro Positivo (VP)**: Quantidade de instâncias positivas verdadeiramente classificadas.
- **Verdadeiros Negativos (VN)**: Quantidade de instâncias negativas corretamente classificadas.
- **Falsos Positivos (FP)**: Quantidades de instâncias negativas erroneamente classificadas.
- **Falsos Negativos (FN)**: Quantidade de instâncias positivas erroneamente classificadas.

  A acurácia calcula a proporção entre a quantidade de correta de acertos feitos e a quantidade de predições realizadas. A equação 2.10 define a acurácia.

$$Ac = \frac{VP + VN}{VP + VN + FP + FN} \tag{2.10}$$

  Por fim, o Erro Absoluto Médio (EAM) é uma métrica que calcula a média da distância absoluta entre os pares previstos e os valores verdadeiros. A equação 2.11 definem o EAM.

$$EAM = \frac{1}{N}\sum_{i=1}^{N} = |y_i - \hat{y}_i| \tag{2.11}$$

3

Trabalhos Relacionados

Este capítulo discute trabalhos relacionados com relação aos temas abordados nesta dissertação. Seção 3.1 apresenta trabalhos prévios em otimização para compiladores guiada por aprendizagem de máquina. Na seção 3.2 é discutido um duas classes de aplicações, predição de desempenho e criação de heurísticas de compilação. Finalmente, a seção 3.2.1 apresenta a principal literatura em Aprendizagem Profunda para compiladores.

3.1 Autotuning e Compilação Iterativa

Sistemas de *Autotuning* exploram o espaço de buscam de possíveis implementações para uma aplicação para geração e otimização automática de código para diferentes cenários e arquiteturas. Um algoritmo de busca guiado por diversas estratégias navega neste espaço compilando e quando necessário executando aplicações com diferentes configurações, a fim de encontrar pontos que maximizem o ganho do sistema; este processo é conhecido como compilação iterativa (Ashouri et al., 2018). Dependendo do propósito do sistema, isto é, do objetivo de minimização ou maximização, a medição do desempenho pode ter um alto custo, pois a aplicação em questão pode ter um elevado tempo de execução ou compilação. A fim de minimizar o custo de busca, técnicas de aprendizagem de máquina são amplamente utilizadas na literatura especializada. Esta seção providencia uma breve revisão da literatura envolvendo sistemas de *autotuning* e compilação iterativa.

3.1.1 Compilação Iterativa Inteligente

Compilação Iterativa se provou uma estratégia eficiente para investigação empírica do comportamento de programas no espaço de transformações (). Chen *et al* (Chen et al., 2010) mostram que a metodologia também é aplicável para qualquer domínio de programas, avaliando 10,000 *benchmarks* e obtendo resultados com 2x de ganho de *speedup* com relação ao plano -O3 do GCC, consolidando ainda mais a técnica.

Muitas pesquisas focam em minimizar o custo da compilação iterativa. O trabalho de Bodin *et al* (Bodin et al., 1998) foi pioneiro ao utilizar compilação iterativa de forma inteligente com busca aleatória, que explora menos que 2% do espaço. Agakov*et al* (Agakov et al., 2006) também inova ao utilizar modelagem preditiva parar otimizar o espaço de busca de transformações. Filho *et al* (J. F. Filho, 2018) apresentam um sistema com diferentes estratégias para redução do espaço de busca, guiado por aprendizagem contínua e um motor que emprega raciocínio baseado em casos. Ashouri *et al* (Ashouri et al., 2016) propõe um sistema de *autotuning* misto de aprendizagem de máquina e compilação iterativa para o problema de seleção de transformações. Outro trabalho que otimiza a tarefa de compilação iterativa é apresentado por Ogilvie *et al* (Ogilvie et al., 2017). Os autores utilizam técnicas de aprendizagem ativa para minimizar o custo de compilação iterativa. Sendo capaz de reduzir até aproximadamente pela metade o número de execuções de uma aplicação.

A compilação iterativa também aparece como complemento para outras técnicas, como é o caso de aprendizagem contínua, e paralelismo de *loops*. (Tartara and Crespi Reghizzi, 2013) propõe um algoritmo que combina aprendizagem contínua com compilação iterativa para construção de heurísticas de compilação. O algoritmo utiliza o conhecimento adquirido pela heurística atual para aprimorara próxima decisão, isto é, a heurística evolui a cada compilação. (Ganser et al., 2017) utilizam copilação iterativa no modelo poliédrico de otimização dentro do espaço de transformações de *loop*. Problemas como superotimização (Bunel et al., 2017) e ordenação de sequências (Newell and Pupyrev, 2020) também fazem uso da técnica.

Por fim, compilação iterativa também é utilizada para aprimorar algoritmos de otimização de código pelo compilador, como Reordenação de Blocos Básicos (Newell and Pupyrev, 2020). Esta otimização é importante para principalmente em sistemas de larga escala, que tendem a conter grandes quantidades de código. A técnica de reordenação pode otimizar a utilização da cache, previsão de desvios otimização de instruções.

3.1.2 Infraetruturas de Autotuning e Exploração

Os benefícios proporcionados por aprendizagem de máquina motivou a construção de infraestruturas de compilação que permitem explorar o espaço de otimizações, integração de estratégias e reprodução de experimentos. *Milepost GCC* (Fursin et al., 2011) foi o primeiro projeto que visou a criação de um compilador de produção guiado por aprendizagem de máquina. Este possui uma interface de extração de *features* estáticas e controle de sequencias de transformações. O sistema de autotuning utiliza 32 programas do conjunto *cBench*. *OpenTuner* é um *framework* de *autotuning* multi-objetivo com técnicas de busca independente do domínio. (Ansel et al., 2014). BOAT é um *framework* desenvolvido para criação de *auto-tuners* a partir de otimização bayseana. . *YaCoS* () é uma infraestrutura para exploração e design de sequencias de transformações, assim como Milepost, porém mais robusta. O sistema possui um motor de busca que disponibiliza diversos algoritmos de aprendizagem e também possibilita a caracterização de programas estaticamente ou dinamicamente e diversas métricas para medir a distância entre programas. YaCoS utiliza LLVM como base de compilação e disponibiliza um conjunto de mais de 5,000 mil C *benchmarks*. Um *frameworks* para compilação iterativa que oferecem abstrações para o processo de compilação iterativa para programas OpenCL é apresentado por Cummins *et al* (Nugteren and Codreanu, 2017).

Outros *frameworks* direcionam seus esforços para o designe de modelos de predição, como é o caso da ferramenta FBNet, um *framework* para design de ConvNets para mobiles onde os autores propõem uma metodologia de busca de arquiteturas neurais que não necessitam de muitos recursos. Também existe uma grande preocupação na geração de programas tensoriais para execução eficiente de redes neurais. Uma vez que existem muitas arquiteturas alvos, esta tarefa demanda um grande esforço de engenharia. (Zheng et al., 2020) explora de forma combinatorial, as otimizações do espaço de busca, utilizando um algoritmo genético, a fim de gerar programas tensoriais eficientes, tornando esta metodologia independente da plataforma alvo.

Uma classe de otimizadores automatizados que também recebe grande atenção são os *Superotimizadores*, que buscam a forma canônica para blocos do programa, isto é, a sequência ótima. Superotimizadores usualmente utilizam busca exaustiva (Bansal and Aiken, 2006), aleatória (Schkufza et al., 2012) ou síntese (Sasnauskas et al., 2017).

3.2 Predição de Desempenho

Modelos de aprendizagem de máquina tem a habilidade de prever um resultado para um conjunto de dados baseado em dados anteriores. No contexto de sistemas de alto desempenho, predição de desempenho significa compreender o comportamento que uma aplicação pode obter em um cenário específico. Este comportamento pode ser formulado como um problema de classificação, ou regressão, atuando em valores contínuos ou discretos, respectivamente.

Modelos de predição para valores contínuos encontram aplicações em tarefas como determinação de *speedup* e consumo de energia por exemplo. Em geral preditores recebem como entrada uma tupla (P, T) onde T é a sequencia de transformações, e P o programa de teste e como saída o valor de predição desejado. Diversos trabalhos utilizam algoritmos de ajuste de curva para estimar ganho de *speedup* programas sequências (Vaswani et al., 2007) (Lee and Brooks, 2006) e OpenMP (Curtis-Maury et al., 2008). Luk *et al* (Luk et al., 2009) estima o tempo de execução para CPU e GPU de acordo com a entrada de programas OpenCL. (Wen et al., 2014) estendem o trabalho anterior para agendamento de tarefas em tempo de execução. O trabalho anterior de Brewer *et al.* (Brewer, 1995) propõe um modelo de regressão baseado em um *solver* de equações diferencias parciais para prever tempo de execução de *layout* de dados. Para predição de energia Benedict (Benedict et al., 2015) utilizam a abordagem de floresta aleatória para estudar o comportamento de programas OpenMP, Rejitha(R.S. et al., 2017) preveem o consumo de energia de código CUDA para GPGPU por métodos de regressão dinâmicos.

No domínio de predição de valores discretos tarefas como mapeamento de hardware, desdobramento de *loops*, granularidade de *threads* são domínios amplamente estudados e de alta complexidade. Para predição de desdobramento de *loops* técnicas supervisionadas como, árvores de decisão (Monsifrot et al., 2002), florestas aleatórias (Zacharopoulos et al., 2018), K-viznhos próximos (KNN) e máquinas de vetores de suporte (SVM) (Stephenson and Amarasinghe, 2005) são utilizadas. Quando programas OpenCL são o foco da otimização, dois principais problemas são estudados: mapeamento de CPU/GPU que consiste em classificar um problema como CPU ou GPU dependendo em qual plataforma de hardware este tem execução mais rápida e fator de granularidade de *threads*, similar a desdobramento de *loops* porém no contexto de ambientes paralelo. Wen *et al* (Wen et al., 2014) utilizam SVM para a tarefa. Ardalani *et al* (Ardalani et al., 2015) utilizam regressão *stepwise* seguido pela meta-heurística de agregação via *bootstrap*. Moren e Göhringer (Moren and Göhringer, 2018) utilizam regressão por floresta aleatória. Fan *et al* (Fan et al., 2019) prevem o *speedup* que diz a respeito da frequência entre número de núcleos

e memória para programas OpenCL utilizando diversos regressores lineares como LASSO e SVR. Sarkar e Mitra (Sarkar and Mitra, 2014) propõe um método não baseado em aprendizagem capaz de prever o tempo gasto por um programa em diferentes partes do seu código. A vantagem de tal metodologia é que esta proporciona a capacidade de identificar quais partes do código consomem mais recursos, e portanto limitam o desempenho em plataformas com GPU. Redes neurais artificiais apresentam resultados promissores para o problema de *threads* (Magni et al., 2014) e (Cummins et al., 2017).

Estratégias que buscam encontrar bons planos de compilação para o PSF utilizando uma abordagem probabilística baseada em regressão logística pode ser visto no trabalho de Cavazos et al (Cavazos et al., 2007) (Cavazos and O'Boyle, 2006). Ashouri *et al* (Ashouri et al., 2014) também faz uso de probabilidades com um rede Bayesiana para encontrar planos de compilação que maximizam a performance da aplicação alvo. Por outro lado, Xavier e Silva (Xavier e Silva, 2018) utilizam técnicas não supervisionadas de clusterização e meta-heurísticas para além de encontrar bons planos de compilação e também a melhor ordem de aplicação. Ashouri *et al* (Ashouri et al., 2016) fazem uso de utilizadas redes Bayesianas para derivar uma distribuição das melhores transformações e então aplicam compilação iterativa sob esta distribuição. Os autores relatam melhores resultados em comparação a utilização exclusiva de compilação iterativa ou aprendizagem de máquina. Ashouri *et al* (Ashouri et al., 2017) utilizam redes neurais e diversos outros algoritmos de regressão linear para mitigação do POF. Park (PARK, 2015) utiliza SVMs em uma predição denominada como predição por torneio, onde uma classificação binária é realizada entre duas sequências.

O trabalho de Cooper *et al* (1999) (Cooper et al., 1999) foi pioneiro ao utilizar uma abordagem evolucionária para redução de código em sistemas embarcados, os autores obtiveram resultados superiores a buscas aleatórias. Desde então, uma série de pesquisas focaram esforços na redução de código para arquiteturas *Very Long Instruction Word* (VLIW). No entanto, com o avanço de sistemas de armazenamento, este problema começou a ser negligenciado pela academia (Ashouri et al., 2018). Porém, recentes trabalhos trazem novamente a tona esta problemática, principalmente com o surgimento da IoT, e a ascensão do *open-source* e a crescente complexidade de evolução dos sistemas de software, onde muitas vezes, estes acabam se tornando imensos projetos com milhares de linhas de código não otimizado e redundante.

O trabalho de Heo *et al* (2018) (Heo et al., 2018) busca através de aprendizagem por reforço, "desinflar"estes sistemas de software. O sistema dos autores recebem como entrada o programa a ser reduzido e uma especificação alto-nível de sua funcionalidade. A saída do sistema é um programa reduzido que é comprovadamente correto com a sua especificação.

Da Silva *et al,* (2020) direcionam esforços para encontrar pequenas sequências de código que superam os planos base LLVM Oz e Os. Rocha *et al,* (2019) desenvolvem uma otimização de redução de código a partir da fusão de funções. Os autores desenvolvem sua metodologia utilizando o algoritmo de Needleman-Wunsch para alinhamento de sequências de DNA.

3.2.1 Redes Neurais para Compiladores e Sistemas

Esta seção providencia um sumário a respeito dos trabalhos realizados em computação de alto desempenho que fazem uso de redes neurais como proposta de solução. A seção 3.3 apresenta algumas representações de programas utilizadas em redes neurais. Estas diferem em sua maioria das representações apresentadas na seção 3.3.1 pois são inspiradas pelas técnicas de processamento de linguagem natural.

Redes de Alimentação Adiante. O modelo base de redes multicamadas é muito utilizado para criação de modelos de predição em ambientes de alta performance. Curtis-Maury *et al* (Curtis-Maury et al., 2007) fazem uso destas redes para prever o consumo de energia de aplicações OpenMP em sistemas multinúcleo. Por sua vez Magni *et al* (Magni et al., 2014) utilizam o modelo cascada de redes neurais para prever o melhor fator de granularidade de *threads* para GPUs. Kulkarni e Cavazos (Kulkarni and Cavazos, 2012) empregam redes neurais para mitigar o problema de ordenação de fase (POF); os autores preveem a performance que a ordem das transformações pode proporcionar para o código sendo otimizado. Ïpek *et al* (Ïpek et al., 2006) utilizam redes neurais para previsão de performance no contexto de exploração do espaço de design arquitetural. Os autores reduzem o número de pontos necessários para simulação. O cálculo de desempenho ocorre através da simulação de um subconjunto de pontos previstos a partir de valores de parâmetros de microarquitetura. Lee *et al* (Lee et al., 2007) realizam um estudo comparativo entre modelos de regressão não linear, como clusterização e ANNs na tarefa de predição de performance no contexto de parâmetros de entradas variáveis. Os resultados indicam que cada método possui vantagens em determinados cenários, no entanto, o processo de treinamento é significativamente mais simplificado quando utilizado ANNs. Dubach *et al* (Dubach et al., 2007) prevem a performance de um programa otimizado utilizando características estáticas e dinâmicas coletadas a partir da execução da aplicação com uma única *thread*.

Redes Neurais Recorrentes. Diferente de redes multicamadas regulares, RNNs são capazes estimar distribuições baseadas em n observações anteriores, tornando esta classe de redes adequada para problemas de predição de séries temporais. Cummins *et al*

(Cummins et al., 2017) utilizam redes recorrentes LSTM para criar um modelo de predição de granularidade de threads para programas OpenCL, como também o mapeamento em CPU ou GPU. Os autores também propõe um método para representar programas baseado no apenas nas características sintáticas do programa, inspirado por técnicas de processamento de linguagem natural. Ben-Nun *et al* (Ben-Nun et al., 2018) também modelam a própria representação, porém ao contrário de Cummins *et al*, a representação dos autores ocorre com código intermediário LLVM. Redes LSTM foram avaliadas em duas tarefas de predição similar ao trabalho anterior: mapeamento de hardware e fator de granularidade ótimo de *threads*. Ambos os trabalhos obtiveram melhores resultados com suas representações em comparação a *features* extraídas manualmente, e resultados similares quando comparados entre si.

Redes Neurais Baseadas em Grafo. Esta classe de redes neurais utilizam estruturas de grafos como entrada. A representação em forma de grafo, em contraste com o modelo sequencial de vetor, é mais enriquecedora para o modelo, uma vez que podem ser quantificadas informações a respeito do fluxo de controle de um programa, e não somente instruções. Brauckmann *et al* (Brauckmann et al., 2020) atacam os mesmos problemas do trabalho de Cummins e Ben-Nun e demonstram que o ganho de desempenho em utilizar grafos é similar ou melhor a abordagem sequencial. O trabalho de Rosário *et al* () é similar ao trabalho de Brauchmann ao utilizar GNNs como mecanismo de aprendizagem, porém, o autor utiliza estas para prever o tempo de execução de programas dado uma sequência de otimizações.

3.2.2 Aprendizagem por Reforço para Compiladores e Sistemas

Muitos problemas de otimização de sistemas de alto desempenho possuem uma natureza baseada em recompensas sequências, onde a soma das recompensar a longo termo é mais importante que a recompensa imediata, por exemplo, o agendamento de tarefas em um *cluster* computacional. Esta característica faz do Aprendizado por Reforço uma abordagem valiosa para otimização de sistemas.

Aprendizagem por Reforço em combinação com Redes Neurais para otimizar o processo de tomada de decisões é referida como Aprendizagem por Reforço Profundo (ARP). Haj-Ali *et al,* (2019) (Haj-Ali et al., 2019) apresentam uma revisão sistemática dos principais trabalhos que utilizam esta metodologia para otimizar sistemas de alto desempenho, incluindo a definição de estados, ações, recompensa, objetivo e o algoritmo utilizado por cada trabalho.

Dentro do contexto de compilação, três trabalhos recentes ganham destaque pois utilizam abordagens modernas de Aprendizagem por Reforço Profundo. NeuroVectorizer: uma abordagem *End-to-End* para vetorização de *loops*, as ações são definidas como a escolha do fator de vetorização e intercalação de *loops*, de ordem $2^n, n \in \mathbb{N}$ (Haj-Ali et al., 2020). AutoPhase: mitigação de problema de ordenação de fase em síntese de alto nível, onde o espaço de ações é definido como $a \in \mathbb{Z} : a \in [0, K)$, onde k é o número total de transformações de compilação. (Huang et al., 2020). AutoCtk: uma abordagem para design de circuitos analógicos. O modelo apresentou resultados melhores com os trabalhos comparados em um espaço de ações de complexidade 10^{11} (Settaluri et al., 2020). Todos os três trabalhos utilizam o algoritmo *Proximal Policy Optimization*.

3.3 Representação de Programas

Abordagens de aprendizagem de máquina dependem propriedades qualitativas, ou *features*, para caracterizar o fenômeno observado. Este fato não é diferente para programas, um bom conjunto de características que descrevam a semântica do programa sendo avaliado é um, se não o mais importante passo da modelagem.

Para caracterizar programas, existem diversas formas. Duas classes de *features* são utilizadas: estáticas ou dinâmicas. Também pode ser utilizado uma combinação das mesmas. Características estáticas são geralmente coletadas da representação intermediária do programa, ou do código fonte, enquanto que, características dinâmicas são coletadas durante a execução da aplicação via *profiling*. Estas características podem ser extraídas manualmente e alimentadas diretamente no modelo, ou a partir de uma representação distribuída de código, permitir que o modelo descubra e utilize as *features* que parecerem mais promissoras.

3.3.1 Representações por Features

Modelos de aprendizagem de máquina utilizam *features* para representar as características dos dados a serem processados; Levando em consideração otimização de código, e o número ilimitado de representações possíveis para um programa a escolha das características mais representativas passa a ser um problema difícil.

Diferentes tipos de características para representar programas tem sido utilizadas, usualmente classificadas entre estáticas, geralmente extraídas da representação intermediária do programa e dinâmicas, obtidas a partir da execução. Cavazos et al (Cavazos et al., 2007) propõem o uso de contadores de performance, um conjunto compacto de informações

que representam o comportamento dinâmico da aplicação. Nestes dados, são incluídos chace misses e utilizações de unidade de ponto flutuante, por exemplo. Por outro lado, Namolaru *et al* (Namolaru et al., 2010) avaliam um conjunto de 56 *features* estáticas obtidas através de programação lógica em relações. Estas sumarizam informações a respeito de funções, instruções, operandos, variáveis, tipos, constantes, blocos básicos e *loops*. Este conjunto de características tem se provado uma representação superior quando comparado com os dados estatísticos fornecidos pelo passe `-stats` presente no otimizador LLVM ().

Métodos que automatizam a extração de *features* incluem o sistema HERCULES (Park et al., 2014). Os autores desenvolvem uma metodologia de caracterização estática de programas guiada por um motor que utiliza programação lógica para busca de padrões em código a fim de efetuar à extração de características de *loops*. Muitas destas informações dizem a respeito de dependências de dados e dependências carregadas por *loops*. Leather *et al* (Leather et al., 2014) apresentam um método automático de selecionar *features* através programação genética em um espaço de características definido por uma gramática. Devido ao tamanho do espaço de busca, uma gramática de 160kB foi escrita.

Usualmente, técnicas de aprendizagem de máquina utilizam vetores para estruturar características. Park *et al* (Park et al., 2012) apresentam uma abordam única baseada em estruturas de grafo para representatividade de programas ao invés de estruturas de vetores. Resultados superiores a representações estáticas utilizando vetores e contadores de performance foram obtidas utilizando Máquinas de Vetor Suporte (SVM). No entanto, o treinamento se torna mais custoso uma vez que é necessário realizar travessias em grafos. Cummins *et al* (200) apresentam PROGRAML , uma representação multigrafo construída em cima da LLVM IR, que captura relações de controle, dados, e chamadas, e indicam instruções, tipos de operandos e ordenação (Cummins et al., 2020).

Todas os trabalhos e técnicas apresentadas possuem um fator em comum: todas as *features* são selecionadas manualmente. Este processo é sujeito a má representabilidade, pois depende da escolha do designer a respeito das características que mais afetam o desempenho e requer conhecimento especializado do sistema mesmo quando a técnica é automatizada. Técnicas que buscar contornar este problemas são apresentadas na seção.

A tabela Tabela - 3.1 apresenta um sumário de todos os trabalhos abordados nesta seção.

Tabela 3.1: Sumário dos trabalhos apresentados na subseção e sua classificação em relação a linguagem, método de extração de características, e estrutura de dados.

Modelo	**Linguagem**	**Extração**	**Estrutura**
Cavazos *et al*	`C,C++`	Dinâmico	Vetor
Namolaru	GCC IR	Estática	Vetor
HERCULES	LLVM IR	Estática	Vetor
Park *et al*	LLVM IR	Estática	Grafo
ProGraML	LLVM IR	Estática	Grafo

3.3.2 Representações Distribuídas

Redes neurais não necessitam de conhecimento preliminar fornecido pela base de conhecimento. Nesta seção discutimos alguns trabalhos que não utilizam *features* pré-definidas e permitem que o próprio modelo aprenda e selecione as *features* mais apropriadas. Nestas configurações a rede é alimentada por uma estrutura de dados que codifica o vocabulário da linguagem para uma representação numérica, esta caracterização é conhecida como *code embedding*.

Prever sequências de texto, utilizando uma uma abordagem de codificação-decodificação é um problema muito estudado dentro do contexto de Processamento de Linguagem Natural. Recentemente esta técnica começou a ser utilizada para representação de linguagens de programação. Allamanis *et, al* (Allamanis et al., 2016) utilizam uma abordagem de atenção em redes convolucionais para sumarização de fragmentos de código, neste caso, previsão de nomes de métodos em programas `C#`. (Iyer et al., 2016) utilizam uma abordagem similar, porém, com redes neurais recorrentes LSTM. Cummins *et al* (Cummins et al., 2017) como já mencionado na seção 3.2.1 utilizam um vocabulário a partir da representação sintática do código, isto é, do código fonte. Esta se da por meio de uma sequência de *tokens* de um vocabulário, onde cada *token* é representado por um número inteiro. Dessa forma, uma sequência de *embedding vectors*, isto é, vetores numéricos do vocabulário que representam o programa, são processados por redes recorrentes, resultando em um único vetor para o código processado.

Diferente dos trabalhos citados acima, Alon *et al* (2018) constroem uma ferramenta CODE2VEC , similar a abordagem WORD2VEC que gera *embedding vectors*, porém, utilizando caminhos da árvore sintática abstrata gerada pelo compilador, ao invés de tokens da sintaxe da linguagem. Este trabalho é estendido e os autores apresentam a ferramenta CODE2SEC . Nesta abordagem os vetores são processados por redes recorrentes LSTM e tem como saída uma sequência de vetores utilizados para previsão (Alon et al., 2019).

Ben-Nun *et al* (Ben-Nun et al., 2018) por outro lado, constroem uma representação independente do código fonte, em cima da representação intermediária da infraestrutura LLVM, a esta técnica, os autores a nomeiam INST2VEC . Este modelo além de quantificar informações a respeito dos dados também leva informações sobre o fluxo de controle do programa. Ambas as propostas apresentam resultados similares quando testados em problemas de classificação no sentido de que em um certo domínio ou certos *benchmarks* um modelo é mais performático que outro.

Deixando de lado vetores, e adotando estruturas de grafos podemos citar o trabalho de Cvitkovic *et al* (Cvitkovic et al., 2018) que propõe um modelo baseado em árvores sintáticas abstratas, porém também carregando informação semática através de anotações. Cada elemento do vocabulário é representado como um nó de um grafo; o resultado final é uma AST anotada com informações a respeito dos dados e fluxo de controle pronta para ser processada utilizando Redes Neurais de Grafos. Brauckmann *et al* (Brauckmann et al., 2020) também utilizam estruturas de grafo e definem definem duas representações para código: AST+DF: uma AST enriquecida com informações de fluxo de dados; CDFG+CALL+MEM: um grafo de fluxo de controle enriquecido com informações de chamada e nós de memória rotulados com instruções. Os dois modelos são processados por Redes Neurais de Grafos e comprados com DeepTune (Cummins et al., 2017) e INST2VEC (Ben-Nun et al., 2018). Os resultados a respeito da melhor representação foram inconclusivos. A representação PROGRAML apresenta um formato CDFG em que nada nó possui um atributo do vocabulário INST2VEC , podendo então ser procecssado por Redes Neurais par Grafos (Cummins et al., 2020).

A tabela Tabela - 3.2 apresenta um sumário de todos os trabalhos abordados nesta seção.

Tabela 3.2: Sumário dos trabalhos apresentados na subseção e sua classificação em relação a linguagem, representação de código, e estrutura de dados.

Modelo	**Linguagem**	**Representação**	**Estrutura**
Allamis *et al*	`C#`	Sequência de Tokens	Vetor
Magni *et al*	`C#`	Sequência de Tokens	Vetor
Cummins *et al*	`OpenCL`	Sequência de Tokens	Vetor
code2vec	`Java`	AST	Vetor
code2seq	`Java`	AST	Vetor
inst2vec	LLVM IR	Sequência de Tokens IR	Vetor
Cvitcovic *et al*	`C`	AST	Grafo
AST+DF	`C`	AST	Grafo
CDFG+CALL+MEM	LLVM IR	LLVM IR Grafo IR	Grafo
ProGraMl *et al*	LLVM IR	LLVM IR Grafo IR	Grafo

4

Conjuntos de Dados

4.1 Construção do Conjunto de Dados

O trabalho proposto nessa dissertação norteia a questão de como utilizar aprendizagem de máquina para seleção de sequências de transformações que reduzem o código de maneira mais efetiva que os planos `Oz` e `Os`. Para avaliar esta questão, elaboramos diversos experimentos, a partir dos paradigmas de Aprendizagem Profunda e Aprendizagem por Reforço Profundo. A tabela 4.1 especifica os experimentos propostos para avaliar nossa hipótese.

Tabela 4.1: Tabela de categorias de experimentos.

Modelo	Linguagem	Embedding	Ação
RNN	C	code2seq	-
RNN	LLVM IR	inst2vec	-
DRL	LLVM IR	inst2vec	Múltiplos passes
DRL	LLVM IR	inst2vec	Um passe

A fim de avaliar empiricamente os algoritmos propostos, foi necessário a construção de um conjunto de dados para avaliação. A construção do conjunto consiste em I) Construir equências de otimização, II) Selecionar benchmarks, III) Criar representações de programas para os algoritmos. As subseções a seguir descrevem estes passos

4.1.1 Sequências de Otimização

4.1.2 Conjuntos de Benchmarks

As principais plataformas que se beneficiam com código reduzido são sistemas embarcados. Devido a este fato, os *benchmarks* utilizados para treino, teste e validação, foram projetados para testar a performance em tais sistemas.

Para treinamento dos algoritmos de aprendizagem, foram utilizados os *benchmarks* `Angha` e a coleção de *benchmarks* disponíveis pela plataforma de compilação `YACOS`. *Benchmarks* `Angha` são compostos por programas C compiláveis, gerados a partir de mineração de código fonte de repositórios de código aberto. Cada programa é composto por uma única função extraída do conjunto de programas coletados. Para garantir a compilação de cada função, a ferramenta reconstrói declarações necessárias.

A partir do conjunto `Angha`, foram selecionadas três suítes de *benchmarks*: $angha_{1K}$, $angha_{15K}$, $angha_{W}$. O conjunto $angha_{W}$ é formada por todos os 15.264 programas extraídos dos repositórios. O conjunto $angha_{15K}$ é composto pelas 15.000 maiores funções (número de instruções LLVM) do conjunto de 1 milhão de funções geradas pela metodologia Angha. Por fim, a metodologia de construção da suíte $angha_{1K}$ constituiu em clusterizar o conjunto $angha_{15K}$ em 1.500 grupos utilizando o algoritmo *K-Means* em vetores `msf`. Cada benchmark do conjunto $angha_{1K}$ é o centroide de um dos grupos.

A infraestrutura de exploração `YACOS` disponibiliza em um único repositório de código aberto com mais de 40 conjuntos de *benchmarks*, somando um total de mais de 300 *benchmarks* C e C++. Estes conjuntos não são especializados para redução de código ou sistemas embarcados. No entanto, seu uso se faz necessário a fim de se obter um parâmetro comparativo entre os conjuntos com esta finalidade. Denominados o conjunto de programas C utilizado neste trabalho como $yacos_{c}$. A lista completa dos *benchmarks* se encontra no apêndice [].

Cada uma dos quatro conjuntos distintos foram utilizados separadamente para treinamento dos algoritmos.

Tabela 4.2: Conjuntos de treinamento.

	`Angha`			`YACOS`
Conjunto	$angha_{1K}$	$angha_{15K}$	$angha_{W}$	$yacos_{c}$
Tamanho	1.500	15.000	15.264	300

As suítes utilizadas para teste contemplam quatro conjuntos de benchmarks, são estes:

- **CoreMark**[1] é um *benchmark* destinado a avaliar a performance de um único processador. Multiplicação de matrizes, máquina de estados e lista linkada são os algoritmos avaliados.
- **CoreMark-Pro**[2] foi desenvolvido com o objetivo de estressar todos os núcleos da CPU através de uma combinação de *workloads* de pontos flutuantes e inteiros.
- **EMBENCH**[3] direcionam seus *benchmarks* para sistemas profundamente embarcados, isto é, sistemas que não assumem a presença de um sistema operacional, possuem uma biblioteca C mínima e sem fluxo de saída de dados.
- **CSiB**[4] é um conjunto de *benchmarks* utilizados para monitorar o tamanho de código gerado para o compilador.

Os *benchmarks CoreMark* e *CoreMark-Pro* fazem parte suíte EEMBC[5]. Esta é uma suíte com diversos conjuntos de *benchmarks*. Cada conjunto é especializado em testar performance em uma tarefa específica, como computação heterogênea, performance de um processadores de um único núcleo, sistemas simétricos *multi-core*, *telefones*, *tablets*, e IoT. A **??** mostra a divisão dos *benchmarks* entre treino e teste.

Tabela 4.3: Conjuntos de teste.

	Conjuntos			
	Coremark	Coremark-pro	Embench-iot	CSiB
Tamanho	1	9	19	2

Usualmente, redes neurais necessitam de milhares de dados para treino. Com 29 *benchmakrs* e 1290 sequências de transformações, o espaço de treino apresenta 37410 pontos.

4.1.3 Trabalho Comparativo

Uma métrica padrão de avaliação para compilação preditiva é a comparação com os planos usuais de compilação `-O3` para tempo de execução, e `-Os -Oz` para tamanho de código. Estes planos estão presentes nos compiladores GCC e LLVM. No entanto, existem trabalhos com metodologias que superam estes planos. A fim de avaliar a eficácia

[1] https://github.com/eembc/coremark
[2] https://github.com/eembc/coremark-pro
[3] https://www.embench.org/
[4] http://szeged.github.io/csibe/
[5] https://www.eembc.org/

do modelo proposto com as melhores técnicas atuais, comparamos nossos resultados com o trabalho de (Rocha et al., 2019). O autores implementam uma técnica de fusão de funções baseadas em alinhamento de sequências de DNA, onde estes relatam superiores aos trabalhos comparados em até 25% com os *benchmarks* C/C++ SPEC CPU2006 e MiBench.

Nossa proposta também é compara com a proposta SalSSA de (Rocha et al., 2020). Os autores estendem o trabalho anterior e criar uma ferramenta implementada em LLVM para suportar código em formato SSA, e obtém $2x$ mais redução de código sob os *benchmarks* SPEC 2006 e 2017 em comparação com trabalhos estado da arte.

4.1.4 Questões de Avaliação

A avaliação dos nossos modelos de predição é norteada pelas questões descritas a seguir:

- Quais as vantagens e desvantagens dos modelos propostos com relação ao problema abordado?
- Qual representação distribuída aumenta a acurácia da predição?
- Qual o erro médio do desempenho das abordagens propostas?
- Dentre os modelos avaliados, em qual cenário cada um apresenta a predição mais precisa ?

As respostas a tais questões são discutidas na Seção 5.

5

Metodologia Experimental

Neste Capítulo é descrito a proposta da dissertação. Isto inclui a motivação, os algoritmos utilizados para mitigar o problema de geração de código reduzido, e os materiais e métodos utilizados para obtenção dos resultados.

5.1 Motivação

Tamanho de código nunca foi o alvo principal objetivo de otimização visado pelos desenvolvedores de compiladores padrões. Tal fato pode ser comprovado quando observamos os planos de compilação `-Oz` e `-Os` providenciados pela LLVM. Estes planos não desabilitam as transformações *loop unrolling* e *inlining*, que são transformações notórias pelo aumento do tamanho de código (llvm dev, 2020). Os planos mais bem consolidados são os clássicos `-O1`, `-O2`, e `-O3`, focados em otimizar tempo de execução.

No entanto, geração de código pequeno tem grande valor para sistemas profundamente embarcados, e juntamente com a ascensão da IoT e arquiteturas *open-source*, como RISC-V, onde a memória implica em um grande custo de produção, tem se tornado uma necessidade crescente. Além de código embarco, aplicações *WebAssembly*, aplicativos para *mobile* e sistemas de tempo real se beneficiam de código reduzido (llvm dev, 2020).

Partimos da premissa de que programas devem ser estudados sob a mesma perspectiva de estudo de linguagem natural, portanto utilizando métodos similares a PNL. Em adição, programas contam com uma estrutura de grafo/árvore que pode representar a ordem do fluxo de dados e controle. Conduzimos um estudo com duas abordagens. Primeiramente, tratando com Redes Recorrentes LSTM programas tratados de forma sequencial similar

a um documento, onde cada instrução é codificada com um embedding pré treinado, e realizamos estudos com a mesma classe de redes LSTM, porém utilizando estruturas de dados em forma de grafo, especificamente, grafos de fluxo de dados e controle.

5.2 Aprendizagem Profunda para Redução de Código

Técnicas de aprendizagem de máquina tem mostrado resultados satisfatórios em diversas tarefas de compilação (Wang and O'Boyle, 2018). Trabalhos que fazem uso de aprendizagem não supervisionada, como *clustering* e algoritmos genéticos datam o início dos anos 2000 e foram estudados principalmente para os problemas de seleção e ordenação de fase. Na última década, em especial, técnicas de aprendizagem profunda e aprendizagem por reforço ganharam destaque em otimização de sistemas. Este trabalho propõe avaliar o problema de seleção de transformações para redução de código utilizando as técnicas estado da arte destes dois modelos de aprendizado citados.

Se tratando de aprendizagem supervisionada, a maior parte das pesquisas envolvendo aprendizagem profunda na mitigação de problemas de compilação, se concentram no uso de redes neurais recorrentes e vetores numéricos como estrutura representativa de código. No entanto, em especial nos últimos dois anos, redes neurais de grafos e estruturas de dados de grafo vem ganhando grande atenção da comunidade de compiladores (Rosário *et al*, 2020). Nossa proposta é avaliar o problema de seleção de fase em redes clássicas recorrentes, como redes LSTM e GRU, e também avaliar redes neurais em grafos.

Outra contribuição que este trabalho trás é o uso de Aprendizagem por Reforço Profundo. A técnica de aprendizagem por reforço tem se mostrado uma das mais promissoras em aprendizagem de máquina, devido ao seu sucesso em robótica, jogos clássicos e capacidades super-humanas. Neste cenário, um agente aprende através da iteração contínua com o ambiente. Em conjunto com uma rede neural que é utilizada para aprender a política de decisão do agente, esta técnica é denominada Aprendizagem por Reforço Profundo, e neste trabalho buscamos utiliza-lá para otimizar a tarefa de compilação.

Além da questão da técnica de aprendizagem e escolha do modelo, outro problema que abrange a compilação iterativa é a representação de código adequada para o objetivo em questão. Em contraste com técnicas de extração manual, com características definidas empiricamente, representação baseadas em linguagem natural, que buscam criar um vocabulário para a linguagem, tem ganhado atenção nos últimos anos. Neste trabalho, apresentamos uma abordagem inédita para predição de tamanho de código, onde o modelo olha apenas para o código fonte do programa e uma sequência, e é capaz de prever a

quantidade de instruções que esta pode reduzir. Também realizamos uma abordagem de predição baseada na representação intermediária do código.

5.2.1 Representação de Código

Neste trabalho optamos por utilizar representações distribuídas de código. A escolha se deu devido a grande integração destas representações com Redes Neurais. Outro fator, é a possibilidade dos modelos aprenderem automaticamente quais as características mais relevantes, evitando o trabalho da coleta manual de *features* do código.

O processo de compilação de uma linguagem geralmente envolve a tradução para diversas outras linguagens intermediárias, neste trabalho utilizamos duas representações distribuídas em dois formatos de código diferentes: a representação intermediária (IR) LLVM, e o próprio código fonte. Para o primeiro, utilizamos a metodologia *Neural Code Comprehesion*, ou INST2VEC [1], um *embedding* do Grafo de Fluxo de Controle e Dados da LLVM IR. Para o código fonte, utilizamos o *embedding* CODE2SEC [2]. Esta representação diferente das representações de tokens dos métodos clássicos de *Neural Machine Translation*, e utilizam conjuntos de caminhos da Árvore Sintática Abstrata para realizar o mapeamento. A escolha de linguagens diferentes possibilita a análise dos fatores positivos e negativos de cada representação.

5.2.2 Arquiteturas de Redes Neurais

Nossa proposta é utilizar Redes Neurais Recorrentes para mitigar o problema de Predição de Tamanho de Código. Nosso modelo faz uso das camadas recorrentes LSTM e GRU. O modelo será avaliado com duas representações de código distintas. Uma representação distribuída próxima ao código fonte, a partir da Árvore Sintática Abstrata, e outra avaliação com uma representação da Linguagem Intermediária. Ambas estruturas são grafos, porém, estes são linearizados em vetores para serem processados por Redes Recorrentes. A Rede Neural foi implementada utilizando o *framework* de Aprendizagem Profunda *Keras*[3].

A arquitetura de rede é similar para os dois experimentos. Para o primeiro experimento, o modelo tem como entrada uma tupla (P_j, s_i), onde o primeiro elemento é um vetor numérico extraído com a ferramenta CODE2SEC que representa um *embending* da AST programa fonte, o segundo elemento é a i-ésima sequência de otimização a ser

[1] https://github.com/spcl/ncc
[2] https://code2seq.org/
[3] https://keras.io/

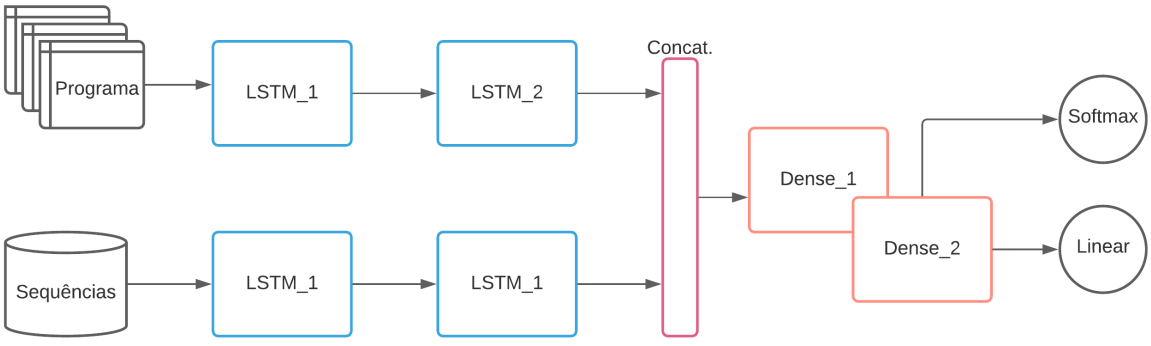
avaliada do conjunto $S = \{s_1, s_2, \ldots, s_{1290}\}$. Para cada programa P_j, a rede é alimentada pela sequência de entradas $(P_j, s_1), (P_j, s_2), (P_j, s_3), \ldots, (P_j, s_{1290})$. Cada elemento da tupla de entrada é processado por duas camadas recorrentes e então concatenados em um único vetor, como ilustra a Figura - 5.1. Este serve como entrada para uma rede densa de 3 camadas que tem como saída o número estimado de instruções que a sequência proporciona, e a categoria de redução desta sequência. A Tabela Tabela - 5.1 apresenta as 4 categorias de classificação de redução.

Tabela 5.1: Categorias de redução de código.

	Redução Baixa	Redução Insignificante	Redução Relativa	Redução Alta
Porcentagem de Redução (cr)	10% <cr <5%	5% <cr <0%	10%<cr <50%	cr >50%

O modelo é treinado com Descida Estocástica de Gradiente, utilizando o otimizador *Adam*. A fim de reduzir o tempo de treino, os dados são separados em múltiplos *batchs* e fornecidos a Rede Neural simultaneamente, reduzindo a frequência com que os pesos são atualizados durante a retro-propagação. As sequências são codificadas em vetores de tamanho fixo $s = max\{S\}$ e caso $s_i < s$, a sequência é completada com 0's. Para gerar um *embedding* das sequências, cada transformação é uma frase que faz parte do conjunto total de transformações presentes nas 1290 sequências, e estas são processadas pela ferramenta *doc2vec*.

Figura 5.1: Modelo RNN para predição de desempenho.



Fonte: Autor, 2020.

Além de camadas LSTM, também avaliamos o desempenho em camadas GRU. Estas são relativamente novas em relação a LSTM e controlam a informação de modo similar,

porém sem a necessidade de uma unidade de memória. Em muitos casos redes GRU são tão eficientes quanto LSTM.

No segundo experimento, avaliamos a rede utilizando uma representação provinda do Grafo de Fluxo de Controle (CDFG), utilizando a ferramenta INST2VEC . Diferente da abordagem anterior, em que um grafo representava o programa e este era avaliado para todas as sequências, agora cada sequência gera um código diferente, e a sequência de entradas para o programa P_j, e da forma: $(CDFG_1^j, s_1), (CDFG_2^j, s_2), (CDFG_3^j, s_3), \ldots, (CDFG_{1290}^j, s_{1290})$, onde $CDFG_i^j$ é o Grafo de Fluxo de Controle e Dados gerado pela aplicação da sequência s_i no programa P_j.

5.2.3 Redes Neurais Tree-LSTM

biblioteca python DGL (Deep Graph Library) (Wang et al., 2019)

Broadly, tree-LSTM is a recurrent neural network (RNN) [37], designed to perform feature extraction from arbitrary length sequence data via the recursive application

6

Proposta

6.1 Comparação de Resultados

Uma métrica padrão de avaliação para compilação preditiva é a comparação com os planos usuais de compilação `-O3` para tempo de execução, e `-Os -Oz` para tamanho de código. Estes planos estão presentes nos compiladores GCC e LLVM. No entanto, existem trabalhos com metodologias que superam estes planos. A fim de avaliar a eficácia do modelo proposto com as melhores técnicas atuais, comparamos nossos resultados com o trabalho de (Rocha et al., 2019). O autores implementam uma técnica de fusão de funções baseadas em alinhamento de sequências de DNA, onde estes relatam superiores aos trabalhos comparados em até 25% com os *benchmarks* C/C++ SPEC CPU2006 e MiBench.

Nossa proposta também é compara com a proposta SalSSA de (Rocha et al., 2020). Os autores estendem o trabalho anterior e criar uma ferramenta implementada em LLVM para suportar código em formato SSA, e obtém $2x$ mais redução de código sob os *benchmarks* SPEC 2006 e 2017 em comparação com trabalhos estado da arte.

6.2 Reprodutibilidade

7

Conclusão e Perspectivas Futuras

REFERÊNCIAS

AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMPSON, J., TOUSSAINT, M., and WILLIAMS, C. K. I. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA. IEEE Computer Society.

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11 pp.–305.

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2014). Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA. Association for Computing Machinery.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. 51(4).

Allamanis, M., Peng, H., and Sutton, C. (2016). A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100.

Alon, U., Brody, S., Levy, O., and Yahav, E. (2019). code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2018). A general path-based representation for predicting program properties. PLDI 2018, page 404–419, New York, NY, USA. Association for Computing Machinery.

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., and Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 303–315.

Ardalani, N., Lestourgeon, C., Sankaralingam, K., and Zhu, X. (2015). Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737, New York, NY, USA. Association for Computing Machinery.

Ashouri, A. H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., and Cavazos, J. (2017). Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3).

Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., and Silvano, C. (2018). A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5).

Ashouri, A. H., Mariani, G., Palermo, G., Park, E., Cavazos, J., and Silvano, C. (2016). Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2).

Ashouri, A. H., Mariani, G., Palermo, G., and Silvano, C. (2014). A bayesian network approach for compiler auto-tuning for embedded processors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 90–97.

Bansal, S. and Aiken, A. (2006). Automatic generation of peephole superoptimizers. ASPLOS XII, page 394–403, New York, NY, USA. Association for Computing Machinery.

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. (2018). Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336.

Benedict, S., Rejitha, R. S., Gschwandtner, P., Prodan, R., and Fahringer, T. (2015). Energy prediction of openmp applications using random forest modeling approach. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 1251–1260.

Bodin, F., Kisuki, T., Knijnenburg, P., O' Boyle, M., and Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France.

Brauckmann, A., Goens, A., Ertel, S., and Castrillon, J. (2020). Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201–211, New York, NY, USA. Association for Computing Machinery.

Brewer, E. A. (1995). High-level optimization via automated statistical modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 80–91, New York, NY, USA. Association for Computing Machinery.

Bunel, R., Desmaison, A., Kumar, M. P., Torr, P. H. S., and Kohli, P. (2017). Learning to superoptimize programs.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, page 185–197, USA. IEEE Computer Society.

Cavazos, J. and O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. *SIGPLAN Not.*, 41(10):229–240.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 448–459, New York, NY, USA. Association for Computing Machinery.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. *SIGPLAN Not.*, 34(7):1–9.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. (2020). Programl: Graph-based deep learning for program optimization and analysis.

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232.

Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. (2008). Prediction models for multi-dimensional power-performance optimization on many cores. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 250–259.

Curtis-Maury, M., Singh, K., McKee, S. A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. (2007). Identifying energy-efficient concurrency levels using machine learning. In *2007 IEEE International Conference on Cluster Computing*, pages 488–495.

Cvitkovic, M., Singh, B., and Anandkumar, A. (EasyChair, 2018). Deep learning on code with an unbounded vocabulary. EasyChair Preprint no. 466.

Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., and Temam, O. (2007). Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, page 131–142, New York, NY, USA. Association for Computing Machinery.

Fan, K., Cosenza, B., and Juurlink, B. (2019). Predictable gpus frequency scaling for energy and performance. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA. Association for Computing Machinery.

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C. K. I., and O'Boyle, M. (2011). Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39:296–327.

Ganser, S., Grösslinger, A., Siegmund, N., Apel, S., and Lengauer, C. (2017). Iterative schedule optimization for parallelization in the polyhedron model. *ACM Trans. Archit. Code Optim.*, 14(3).

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Haj-Ali, A., Ahmed, N. K., Willke, T., Gonzalez, J., Asanovic, K., and Stoica, I. (2019). A view on deep reinforcement learning in system optimization.

Haj-Ali, A., Ahmed, N. K., Willke, T., Shao, Y. S., Asanovic, K., and Stoica, I. (2020). Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 242–255, New York, NY, USA. Association for Computing Machinery.

Heo, K., Lee, W., Pashakhanloo, P., and Naik, M. (2018). Effective program debloating via reinforcement learning. CCS '18, page 380–394, New York, NY, USA. Association for Computing Machinery.

Huang, Q., Haj-Ali, A., Moses, W., Xiang, J., Stoica, I., Asanovic, K., and Wawrzynek, J. (2020). Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning.

İpek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M. (2006). Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 195–206, New York, NY, USA. Association for Computing Machinery.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

J. F. Filho, L. G. A. Rodriguez, A. F. d. S. (2018). Yet another intelligent code-generating system: A flexible and low-cost solution. *Journal of Computer Science and Technology*, 33(5):940.

JUNIOR, N. L. Q. (2016). *Uma solução hibrida para mitigação do problema de seleção de otimizações*. Mestrado, Universidade Estadual de Maringa.

Kulkarni, S. and Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 147–162, New York, NY, USA. Association for Computing Machinery.

Leather, H., Bonilla, E., and O'boyle, M. (2014). Automatic feature generation for machine learning–based optimising compilation. 11(1).

Lee, B. C. and Brooks, D. M. (2006). Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGOPS Oper. Syst. Rev.*, 40(5):185–194.

Lee, B. C., Brooks, D. M., de Supinski, B. R., Schulz, M., Singh, K., and McKee, S. A. (2007). Methods of inference and learning for performance modeling of parallel

applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 249–258, New York, NY, USA. Association for Computing Machinery.

Liu, Z., Lin, Y., and Sun, M. (2020). *Representation Learning and NLP*, pages 1–11. Springer Singapore, Singapore.

llvm dev (2020). LLVM 2020 code size bof minutes.

Luk, C., Hong, S., and Kim, H. (2009). Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55.

Magni, A., Dubach, C., and O'Boyle, M. (2014). Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, page 455–466, New York, NY, USA. Association for Computing Machinery.

MASSALIN, H. (1987). Superoptimizer: A look at the smallest program. *SIGARCH Comput. Archit. News*, 15(5):122–126.

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, page 41–50, Berlin, Heidelberg. Springer-Verlag.

Moren, K. and Göhringer, D. (2018). Automatic mapping for opencl-programs on cpu/gpu heterogeneous platforms. In Shi, Y., Fu, H., Tian, Y., Krzhizhanovskaya, V. V., Lees, M. H., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science – ICCS 2018*, pages 301–314, Cham. Springer International Publishing.

Namolaru, M., Cohen, A., Fursin, G., Zaks, A., and Freund, A. (2010). Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, page 197–206, New York, NY, USA. Association for Computing Machinery.

Newell, A. and Pupyrev, S. (2020). Improved basic block reordering. *IEEE Transactions on Computers*, 69(12):1784–1794.

Nielsen, M. A. (2018). *Neural Networks and Deep Learning*. Determination Press.

Nugteren, C. and Codreanu, V. (2017). Cltune: A generic auto-tuner for opencl kernels. *CoRR*, abs/1703.06503.

Ogilvie, W. F., Petoumenos, P., Wang, Z., and Leather, H. (2017). Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256.

Park, E., Cavazos, J., and Alvarez, M. A. (2012). Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 196–206, New York, NY, USA. Association for Computing Machinery.

Park, E., Kartsaklis, C., and Cavazos, J. (2014). Hercules: Strong patterns towards more intelligent predictive modeling. In *2014 43rd International Conference on Parallel Processing*, pages 172–181.

PARK, E. J. (2015). *Automatic selection of compiler optimizations using program characterization and machine learning*. Phd, University of Delaware.

Rocha, R. C. O., Petoumenos, P., Wang, Z., Cole, M., and Leather, H. (2019). Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 149–163.

Rocha, R. C. O., Petoumenos, P., Wang, Z., Cole, M., and Leather, H. (2020). Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 854–868, New York, NY, USA. Association for Computing Machinery.

R.S., R., Benedict, S., Alex, S., and Infanto, S. (2017). Energy prediction of cuda application instances using dynamic regression models. *Computing*, 99.

Sarkar, S. and Mitra, S. (2014). Execution profile driven speedup estimation for porting sequential code to gpu. In *Proceedings of the 7th ACM India Computing Conference*, New York, NY, USA. Association for Computing Machinery.

Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Taneja, J., and Regehr, J. (2017). Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422.

Schkufza, E., Sharma, R., and Aiken, A. (2012). Stochastic superoptimization. *CoRR*, abs/1211.0557.

Senior, A. W., Evans, R., Jumper, J., Kirkpatrick, J., Sifre, L., Green, T., Qin, C., Žídek, A., Nelson, A. W., Bridgland, A., et al. (2020). Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710.

Settaluri, K., Haj-Ali, A., Huang, Q., Hakhamaneshi, K., and Nikolic, B. (2020). Autockt: Deep reinforcement learning of analog circuit designs. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 490–495.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Stephenson, M. and Amarasinghe, S. (2005). Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*, pages 123–134.

Tartara, M. and Crespi Reghizzi, S. (2013). Continuous learning of compiler heuristics. *ACM Trans. Archit. Code Optim.*, 9(4).

Trinh, T. H., Dai, A. M., Luong, M.-T., and Le, Q. V. (2018). Learning longer-term dependencies in rnns with auxiliary losses.

Vaswani, K., Thazhuthaveetil, M. J., Srikant, Y. N., and Joseph, P. J. (2007). Microarchitecture sensitive empirical models for compiler optimizations. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 131–143.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. (2019). Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*.

Wang, Z. and O'Boyle, M. (2018). Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901.

Wen, Y., Wang, Z., and O'Boyle, M. F. P. (2014). Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10.

XAVIER, T. C. d. S. (2014). *Solução Integrada para os Problemas de Seleção e Ordenação de Fase*. Mestrado, Universidade Estadual de Maringa.

Zacharopoulos, G., Barbon, A., Ansaloni, G., and Pozzi, L. (2018). Machine learning approach for loop unrolling factor prediction in high level synthesis. In *2018 International Conference on High Performance Computing Simulation (HPCS)*, pages 91–97.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. (2020). Ansor : Generating high-performance tensor programs for deep learning.