Redes Neurais - Backpropagation
Trabalho 2 – INF01017 Aprendizado de Máquina – 2019/1

Bruno S. M. de Lima, Lucas N. Alegre e Pedro S. Perrone

Junho de 2019

1 Introdução

Este trabalho tem como objetivo a documentação da implementação e da avaliação de desempenho de uma rede neural treinada via *backpropagation* para tarefas de classificação em aprendizado supervisionado. Foram avaliadas as performances de diferentes arquiteturas com diferentes otimizadores e metaparâmetros em 4 datasets disponibilizados pelos professores.

2 Implementação

Nossa implementação foi feita com a linguagem Python3. Usamos a bilioteca **pandas** para armazenamento e manipulações básicas dos datasets, a biblioteca **numpy** para armazenamento e operações de matrizes e vetores, a biblioteca **tqdm** para visualizar o progresso e o tempo de treinamento e a biblioteca **seaborn** para gerar os gráficos aqui demonstrados. Todas as funcionalidades (incluindo as opcionais) foram implementadas pelo grupo.

2.1 Organização da Implementação

Abaixo, demonstramos como estruturamos os módulos da implementação. O código completo está disponível no GitHub (`https://github.com/LucasAlegre/backpropagation`).

```
.
├── backpropagation          Módulo principal
│   ├── __init__.py
│   ├── nn.py                Implementação da classe da Rede Neural (NN)
│   └── util.py              Funções auxiliares (métricas, normalização de datasets, etc)
├── backpropagation.py       Interface via linha de comando
├── datasets                 Datasets (arquivos CSV) utilizados
├── docs
├── initial_weights.txt
├── logs                     Logs das performances de treinamento
│   ├── plot.py              Geração de gráficos
├── network.txt
├── README.md
└── requirements.txt         Bibliotecas usadas
```

2.2 Interface via linha de comando

É possível executar o script *backprogation.py* de forma parametrizada conforme demonstrado abaixo:

Listing 1: Interface via linha de comando

```
Backpropagation - Aprendizado de Maquina 2019/1 UFRGS
optional arguments:
```

```
-h, --help            show this help message and exit
-s SEED               The random seed. (default: None)
-d DATA               The dataset .csv file. (default: datasets/wine.csv)
-c CLASS_COLUMN       The column of the .csv to be predicted. (default:
                      class)
-sep SEP              .csv separator. (default: ,)
-k NUM_FOLDS          The number of folds used on cross validation.
                      (default: 10)
-e EPOCHS             Amount of epochs for training the neural network.
                      (default: 100)
-mb BATCH_SIZE        Mini-batch size used for training. (default: None)
-drop DROP [DROP ...]
                      Columns to drop from .csv. (default: [])
-nn NN [NN ...]       Neural Network structure. (default: None)
-w WEIGHTS            Initial weights. (default: None)
-alpha ALPHA          Learning rate. (default: 0.001)
-beta BETA            Efective direction rate used on the Momentum Method.
                      (default: 0.9)
-regularization REGULARIZATION
                      Regularization factor. (default: 0.0)
-numerical            Calculate the gradients numerically. (default: False)
-opt OPT              Optimizer [SGD, Momentum, Adam]. (default: SGD)
-log                  Generate log file. (default: False)
```

Para visualizar a verificação numérica dos cálculos dos gradientes para os exemplos 1 e 2 fornecidos na especificação do trabalho, os seguintes comandos devem ser executados:

```
$ python3 backpropagation.py -d datasets/teste1.txt -nn network.txt
-w initial_weights.txt -numerical -e 1

$ python3 backpropagation.py -d datasets/teste2.txt -nn network2.txt
-w initial_weights2.txt -numerical -e 1
```

2.3 Estruturas de Dados

O conjunto de instâncias de teste e treinamento foram armazenadas numa tabela representada por um *DataFrame* da biblioteca **pandas**, que fornece uma implementação eficiente para manipulação de grandes quantidades de dados.

Implementamos o algoritmo de *backpropagation* na sua forma vetorizada. Desse modo, as matrizes e vetores de pesos, gradientes, deltas e ativações dos neurônios foram armazenadas em objetos da biblioteca **numpy** para maior eficiência. Abaixo mostramos um exemplo de como as matrizes de pesos e de gradientes são inicializadas.

Listing 2: Inicialização das matrizes de pesos e gradientes

```
def init_random_weights(self):
    self.weights = np.array([np.random.normal(size=(self.architecture[layer+1],
    self.architecture[layer]+1)) for layer in range(self.num_layers-1)])

def init_grads(self):
    self.grads = np.array([np.zeros((self.architecture[layer+1],
    self.architecture[layer]+1)) for layer in range(self.num_layers-1)])
```

Adicionalmente, o grupo tomou cuidado para realizar as operações vetoriais *in-place*, evitando alocações de matrizes temporárias desnecessárias que potencialmente poderiam aumentar significativamente o tempo de execução. Foram avaliados diferentes otimizadores, arquiteturas

2.4 Funcionalidades Principais

Demonstramos nessa seção duas das principais funcionalidades implementadas: a predição de uma nova instância e a validação cruzada estratificada. É possível visualizar as outras funcionalidades detalhadamente no código completo da implementação.

2.4.1 Predição

Abaixo mostramos como a predição de uma nova instância acontece. Ao receber esta instancia, que já foi previamente normalizada, este método propaga a mesma para gerar as ativações da rede. A partir disso, o método retorna a classe relacionada à saída que gerou o maior valor de ativação.

Listing 3: Predição de uma nova instância

```
def propagate(self, x):
    np.copyto(self.activations[0], np.append(1.0, x).reshape(-1,1))
    for layer in range(1, self.num_layers-1):
        np.dot(self.weights[layer-1], self.activations[layer-1],
               out=self.activations[layer][1:])
        self.activations[layer] = sigmoid(self.activations[layer])
        self.activations[layer][0][0] = 1.0
    np.dot(self.weights[self.num_layers-2], self.activations[self.num_layers-2],
           out=self.activations[self.num_layers-1])
    self.activations[self.num_layers-1] = sigmoid(self.activations[self.num_layers-1])
    return self.activations[self.num_layers-1]

def predict(self, instance):
    instance = instance.drop(labels=[self.class_column]).values
    instance = np.array(instance, dtype='float64')
    activations = list(self.propagate(instance))
    max_index = activations.index(max(activations))
    return self.class_values[max_index]
```

2.4.2 Validação Cruzada Estratificada

Abaixo podemos ver o código utilizado para realizar a validação cruzada estratificada. Inicialmente, é gerado um novo dataframe para cada classe possível, contendo todas as suas intâncias. A partir disso, cada dataframe gerado é subdividido em k subconjuntos de mesmo tamanho, que posteriormente são combinados com subconjuntos de outras classes para gerar um fold estratificado.

Listing 4: Validação cruzada estratificada

```
def stratified_k_cross_fold(data, class_column, k=10):
    data = data.sample(frac=1).reset_index(drop=True) # Shuffle DataFrame rows

    folds_per_class = []
    for _, g in data.groupby(class_column):
        folds_per_class.append(k_cross_fold(g, k=k))  # create fold for each class

    for _ in range(k):
        train = pd.DataFrame()
        test = pd.DataFrame()
        for c in range(len(folds_per_class)):         # append the next fold for each class
            train_c, test_c = next(folds_per_class[c])
            train = train.append(train_c)
            test = test.append(test_c)
        yield train, test
```

3 Parâmetros

- **architecture:** número de camadas e neurônios por camada da rede;
- **epochs:** quantidade de épocas nas quais a rede neural é treinada;
- **minibatch_size:** tamanho do mini-batch usado durante o treinamento da rede. Caso o valor especificado seja 1, realiza treinamento estocástico. Caso o valor não seja especificado, reliza treinamento em batch. Caso o valor especificado for maior que 1, realiza treinamento em mini-batch;
- **weights:** pesos iniciais da rede. Caso não seja especificado, os pesos são amostrados de uma distribuição Gaussiana $N(0, 1)$;

- **alpha:** taxa de aprendizado (α) usado no treinamento;
- **beta:** taxa da direção efetiva (β) utilizada no método do momento;
- **regularization:** fator de regularização (λ) da rede;
- **optimizer:** indica o otimizador a ser utilizado no treinamento da rede (SGD, Momentum ou Adam);

4 Experimentos e Resultados

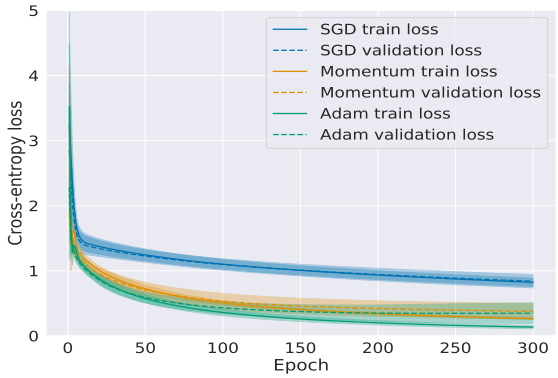
Nos experimentos apresentados a seguir, o método de validação cruzada estratificada foi usada com $k = 10$ folds. Os gráficos relativos ao valor da função de custo apresentam a média nos folds de treinamento e de validação para cada umas das k execuções, onde a área sombreada representa o desvio padrão. Os gráficos do tipo boxplot relativos a F1-measure apresentam o F1 score obtido no fold de validação para cada uma das k execuções.

Foi feito o treinamento em *mini-batch* com batches de tamanho igual a 32, valor que obteve os melhores resultados em avaliações empíricas. Todos os datasets foram normalizados com a normalização min-max, escalando as features para o intervalo [0-1]. Ainda, todos os experimentos foram executados com a semente aleatória igual a 42 para garantir a reprodutibilidade dos resultados.

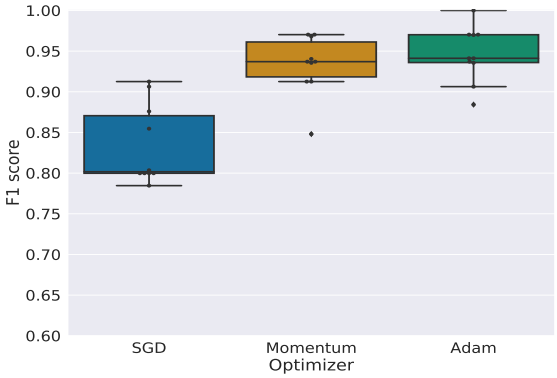
4.1 Otimizadores

Além dos métodos Stochastic Gradient Descent (SGD) e Momentum, o grupo achou interessante realizar a comparação com um otimizador estado-da-arte. Desse modo, implementamos e analisamos a performance do otimizador Adam (`https://arxiv.org/abs/1412.6980`). Escolhemos realizar essa análise no dataset *ionosphere* a partir da melhor arquitetura encontrada para esse dataset (uma camada oculta com 30 neurônios e sem regularização).

Para a taxa de aprendizado foi atribuído o valor $\alpha = 0.01$ para os três otimizadores. Para o método Momentum foi usado $\beta = 0.9$, e para o método Adam foi usado $\beta_1 = 0.9$ e $\beta_2 = 0.999$, conforme é sugerido na literatura.



(a) Valores médios da função de custo nos folds de treinamento e validação.



(b) F1 score nos folds de validação obtidos por cada otimizador.

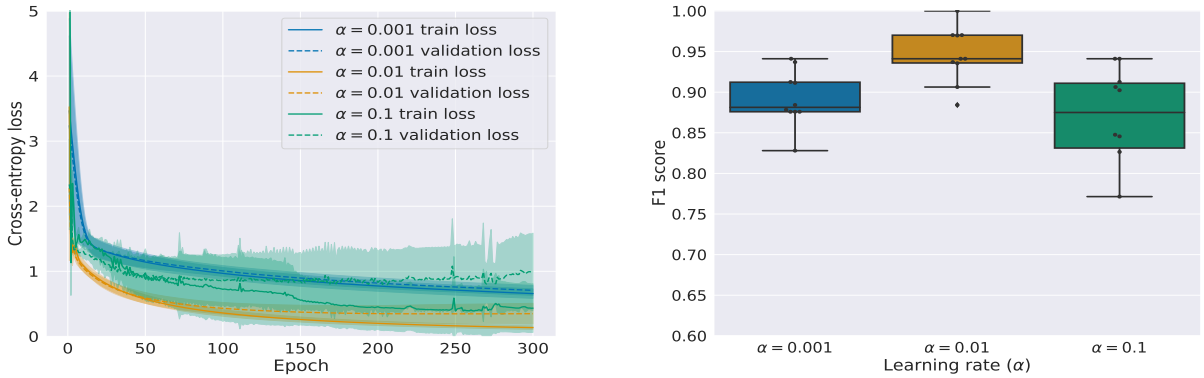
Figura 1: Comparação dos otimizadores SGD, Momentum e Adam no dataset *ionosphere*.

A Figura 1a mostra que as redes neurais treinadas com o SGD tenderam a ficar presas em um mínimo local. Já o método Momentum introduziu melhoria significativa na redução do valor da função de custo tanto no treinamento quanto na validação em relação ao SGD, enquanto o método Adam foi levemente superior ao Momentum. O mesmo comportamento se mantém se analisamos o F1 score na Figura 1b.

O otimizador Adam, portanto, foi usado em todos os experimentos posteriores.

4.2 Taxa de aprendizado

A partir dos parâmetros utilizados no experimento anterior e utilizando o otimizador Adam, que obteve os melhores resultados, avaliamos o impacto de diferentes taxas de aprendizado na performance do modelo. Os valores testados foram $\alpha = 0.001$, $\alpha = 0.01$ e $\alpha = 0.1$.



(a) Valores médios da função de custo nos folds de treinamento e validação.

(b) F1 score obtido em cada fold de validação.

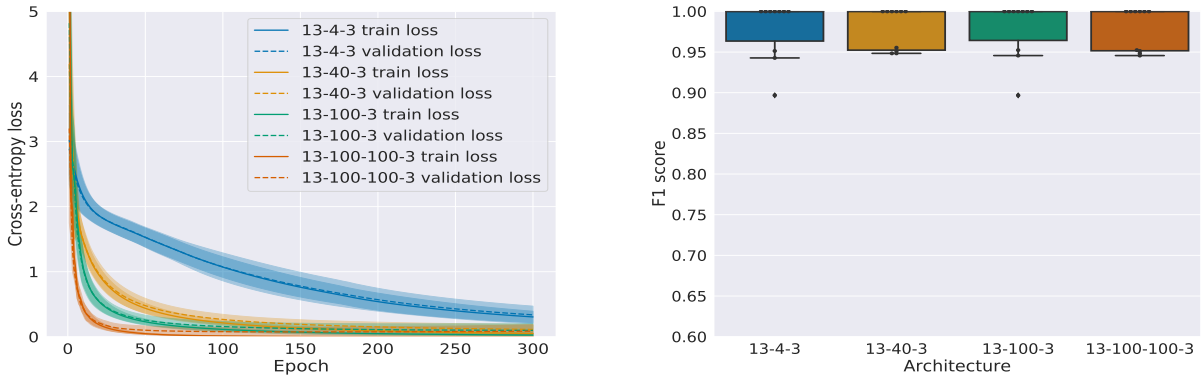
Figura 2: Comparação de diferentes taxas de aprendizado (α) no dataset *ionosphere*.

A Figura 2a demonstra a importância de selecionarmos um valor adequado para o parâmetro. Um valor muito baixo ($\alpha = 0.001$) levou os pesos da rede para um mínimo local. Já um valor muito alto ($\alpha = 0.1$) produziu o efeito de *overshooting*, levando à instabilidade e alta variância na performance das redes neurais (note o alto desvio padrão nas Figuras 2a e 2b). Um valor intermediário ($\alpha = 0.01$), entretanto, obteve os melhores resultados.

O valor $\alpha = 0.01$ foi usado nos experimentos seguintes, pois também mostrou-se um bom valor para os outros datasets e arquiteturas testadas.

4.3 Wine dataset

O *wine dataset* contém 13 atributos que representam características de 178 instâncias de vinhos que estão divididos em três tipos. Os gráficos da figura 3 mostram os resultados do treinamento de diferentes configurações de uma rede neural.



(a) Valores médios da função de custo nos folds de treinamento e validação.

(b) F1 score obtido em cada fold de validação.

Figura 3: Comparação entre diferentes arquiteturas no dataset *wine*.

Ao observar a Figura 3a, podemos perceber que quanto maior a arquitetura utilizada na rede, mais

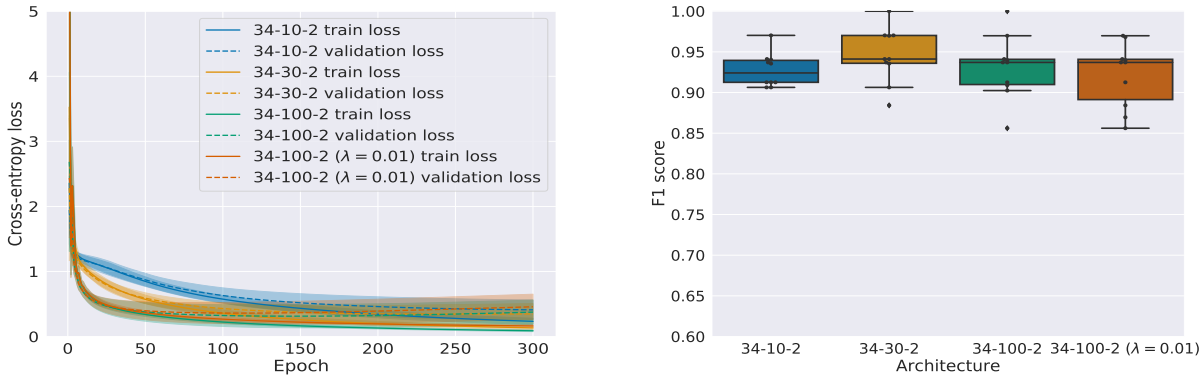
rápido o treinamento convergiu para uma boa solução. Além disso, todas as arquiteturas, com exceção da *13-4-3*, convergiram para um valor de perda muito próximo de 0.

Este benefício de utilizar arquiteturas muito grandes normalmente viria acompanhado de dois principais problemas: um grande aumento no tempo de treinamento por época e *overfitting*. Como este trabalho foi implementado fazendo o uso de vetorização durante o treinamento da rede, o tempo de execução acabou não se tornando um problema tão significativo. Além disso, devido à simplicidade deste dataset, o problema de *overfitting* também não aconteceu para arquiteturas muito grandes.

Outro ponto interessante de se observar é a performance final das redes treinadas, mostrada na Figura 3b. Como a arquitetura *13-4-3* demonstrou maior perda ao final do treinamento quando comparada à outras arquiteturas, seria intuitivo pensar que esta também demonstraria um desempenho pior no *cross-validation*. No entanto, esta arquitetura demonstrou obter resultados tão bons quanto as arquiteturas mais complexas. Isso se deve ao fato de que, diferente da medida *loss*, a *F1-measure* avalia o modelo considerando somente sua predição final, e não o valor esperado das ativações da rede.

4.4 Ionosphere dataset

O *ionosphere dataset* contém 351 instâncias com 34 atributos sendo classificados em duas classes. Os gráficos da figura 4 exibem os resultados dos experimentos.



(a) Valores médios da função de custo nos folds de treinamento e validação.

(b) F1 score obtido em cada fold de validação.

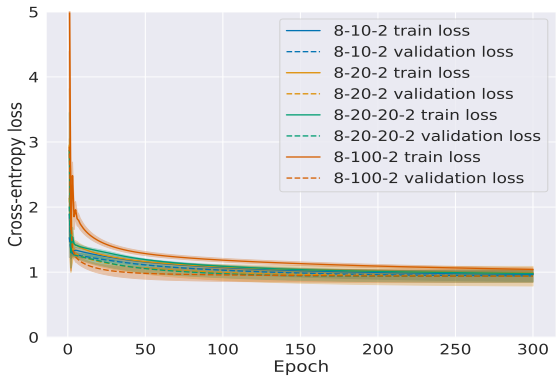
Figura 4: Comparação entre diferentes arquiteturas no dataset *ionosphere*.

A melhor arquitetura encontrada foi *34-30-2*. A versão com apenas 10 neurônios na camada oculta gerou um modelo mais simples e menos capaz de fazer classificações corretas. A versão com 100 neurônios na camada oculta gerou um modelo complexo demais, gerando *overfitting*. Isso se conclui ao ver no gráfico 4a que o erro para este modelo é muito baixo, ou seja, que ele aprendeu muito bem a classificar os dados de treinamento, mas o gráfico 4b mostra um desemepnho mais baixo na medida F1 para diferentes *folds*.

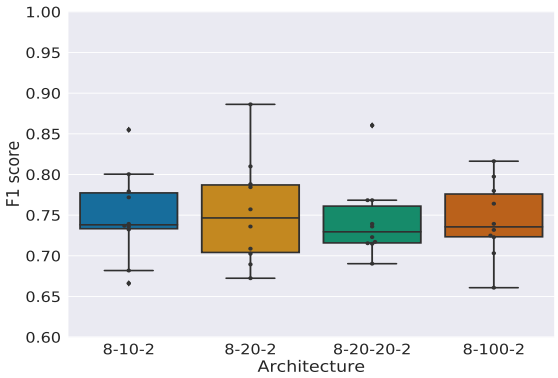
Por fim, notamos que a adição de regularização não foi capaz de aumentar o desempenho da rede nesse dataset.

4.5 Pima dataset

O *pima dataset* conta com 768 instâncias com oito atributos e duas classes. Os resultados dos testes com diferêntes parâmetros se encontram na Figura 5.



(a) Valores médios da função de custo nos folds de treinamento e validação.



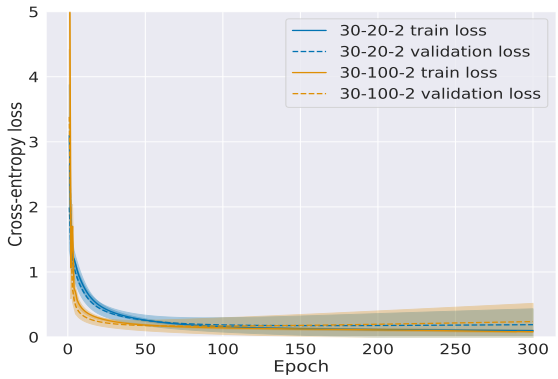
(b) F1 score obtido em cada fold de validação.

Figura 5: Comparação entre diferentes arquiteturas no dataset *pima*.

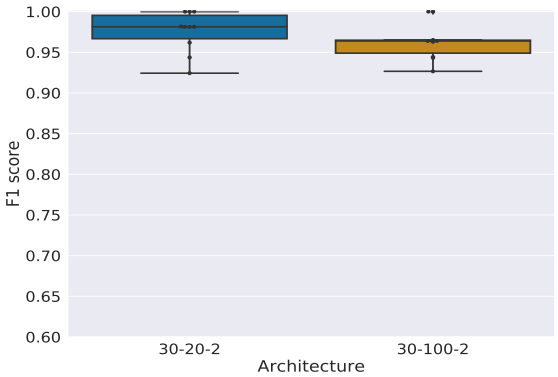
Este foi o conjunto de dados com o pior desempenho. Alterações na arquitetura e em outros parâmetros como a taxa de aprendizado e o fator de regularização não introduziram diferenças significativas na performance. O gráfico 5b evidencia que nem o aumento do número de neurônios e nem a adição de uma nova camada oculta foram capazes de trazer melhorias.

4.6 WDBC dataset

O *wdbc dataset* apresenta 32 atributos de 569 pacientes e indica a presença ou ausência de câncer. Os resultados dos experimentos se encontram nas Figuras 6 e 7.

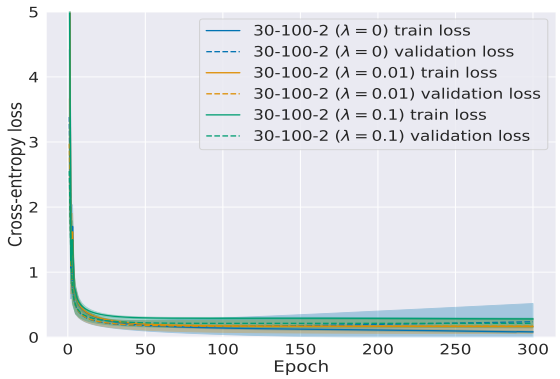


(a) Valores médios da função de custo nos folds de treinamento e validação.

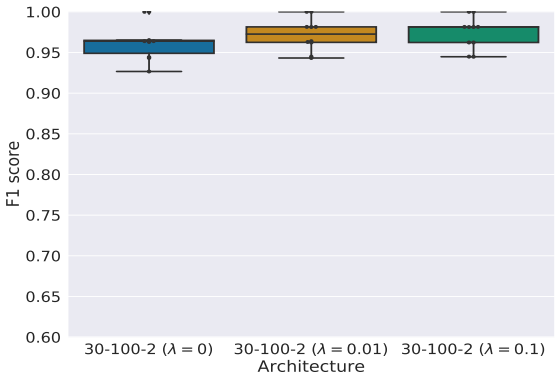


(b) F1 score obtido em cada fold de validação.

Figura 6: Comparação entre diferentes arquiteturas no dataset *wdbc*.



(a) Valores médios da função de custo nos folds de treinamento e validação.



(b) F1 score obtido em cada fold de validação.

Figura 7: Comparação entre diferentes valores para o fator de regularização no dataset *wdbc*.

Uma arquitetura simples com apenas 20 neurônios obteve resultados excelentes para esse dataset. Ao aumentar o número para 100 neurônios introduziu-se um pequeno *overfitting* (note o aumento do desvio padrão para a linha amarela).

Na Figura 7 avaliamos como a regularização pode reduzir o *overfitting* introduzido anteriormente. Com valores iguais a $\lambda = 0.01$ e $\lambda = 0.1$ a performance melhorou comparativamente ao não uso de regularização. É interessante notar que o aumento do desvio padrão ocorre apenas na linha azul, na qual não há regularização.

5 Conclusão

Redes neurais treinadas via *backpropagation* apresentam um excelente desempenho na tarefa de classificação. Os resultados obtidos foram superiores aos do primeiro trabalho, com o treinamento de florestas aletorias e o tempo de treinamento foi muito menor.

Um aspecto notável é como o método de aplicação do gradiente otimiza e acelera o treinamento da rede neural. A Figura 1a mostra que, após 50 epocas, o erro na rede treinada com o Adam era cerca de metade do erro da rede treinada com o Stochastic Gradient Descent. Já a imagem 1b mostra visualmente como os resultados obtidos são mais precisos com os métodos mais sofisticados.

Outro ponto relevante é a importância de experimentação: para cada dataset um conjunto de parâmetros obterá um melhor resultado. Para ilustrar a afirmação podemos ver o efeito do parâmetro de regularização nos diferentes *datasets* analisados.

Para os *datasets* utilizados neste trabalho não foi necessária a adição de muitas camadas devido a simplicidade dos conjuntos. Apesar de tal adição reduzir o erro, era gerado *overfitting*, o que reduzia o valor da medida F1.

O desempenho da implementação foi bastante satisfatório para a maioria dos datasets, nos quais puderam ser avaliados diferentes combinações de metaparâmetros e aplicados os conceitos de aprendizado de máquina aprendidos em aula.