Universidade Federal de Pernambuco
Centro de Informática
Curso de Bacharelado em Engenharia da Computação

Análise da eficiência do algoritmo *Adjustable Step Decay*

Erick Calado de Carvalho Filho

Recife
2018

Centro de Informática
Curso de Bacharelado em Engenharia da Computação

Erick Calado de Carvalho Filho

Análise da eficiência do algoritmo *Adjustable Step Decay*

Monografia apresentada ao Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Engenharia da Computação, orientada pelo professor Cleber Zanchettin.

Recife
2018
Universidade Federal de Pernambuco

Dedico esse trabalho a todos aqueles que me apoiaram
nessa longa jornada.

Resumo

Na área de *deep learning*, um dos fatores mais importantes para o bom funcionamento dos algoritmos é o ajuste dos muitos parâmetros do modelo. No contexto supervisionado, normalmente são utilizados algoritmos *Stochastic Gradient Descent* para esse processo de ajuste. Para garantir a convergência do algoritmo, a taxa de aprendizagem deve ser ajustado com o tempo; valores muito baixos retardam a convergência, e valores muito altos podem fazer a função divergir ou flutuar ao redor do ponto esperado. Uma proposta para ajuste baseado nas épocas de treinamento, chamada de *Adjustable Step Decay*, foi sugerida em um trabalho não finalizado e apresentou resultados positivos em um conjunto de bases de dados, melhorando os resultados e acelerando a convergência. Esse trabalho pretende realizar uma avaliação para analisar a eficácia do *Adjustable Step Decay* em outras bases de dados.

Palavras-chave: ajuste de parâmetros, *Stochastic Gradient Descent*, *deep learning*, taxa de aprendizagem.

Conteúdo

Lista de Figuras

Lista de Tabelas

1. Introdução

Redes *deep learning* são redes neurais artificiais com múltiplas camadas, que têm como objetivo o aprendizado de uma função. Essas redes são inspiradas nos neurônios e suas sinapses. Um neurônio é uma estrutura celular com um conjunto de dendritos para recepção de estímulos, um corpo celular para integrar esses estímulos, e axônios que transmitem o sinal emitido pelo corpo celular para outros neurônios. Um neurônio artificial recebe diversas entradas e produz uma saída baseada nos valores dessas entradas, nos pesos de cada uma delas e na sua função de ativação. Uma camada de rede é composta de um ou vários destes neurônios em paralelo. A quantidade de camadas determina a profundidade da rede. As redes *deep learning* são chamadas assim por terem muitas camadas (*deep* significa profundo em inglês), embora sejam classificadas assim tendo ao menos duas camadas escondidas.

Aprendizagem de máquina usando *deep learning* requer que os dados sejam representados por muitos níveis de abstração, gerando uma hierarquia de características com representações em níveis mais próximo da saída sendo definidas por representações em níveis mais próximo da entrada. Esse grande número de camadas faz com que o treinamento das redes seja mais complexo, aumentando a quantidade de pesos da rede e tornando complexo o processo de ajuste dos parâmetros.

Para o ajuste dos parâmetros, normalmente são utilizados algoritmos que empregam a primeira ou segunda derivada da função de perda (Goodfellow et al., 2016). Os que usam a primeira derivada (gradiente) são chamados de algoritmos de otimização de primeira ordem, e aqueles que utilizam o gradiente e o Hessiano são chamados de algoritmos de otimização de segunda ordem (Nocedal and Wright, 2006).

Em teoria, algoritmos de segunda ordem são melhores que os de primeira ordem. No entanto, os cálculos para casos gerais são extremamente custosos. Um exemplo é o método de Newton, que necessita da matriz inversa do Hessiano. Essa matriz é incalculável para a maioria das redes reais. Métodos quasi-Newton, como BFGS, usam uma aproximação da inversa do Hessiano. Porém, tais aproximações requerem que o conjunto de testes seja avaliado por completo antes de

uma atualização, fazendo com que o treinamento seja mais lento. Isso faz com que algoritmos de primeira ordem sejam mais utilizados (Goodfellow et al., 2016).

Os algoritmos de primeira ordem, em sua maioria, utilizam gradiente descendente estocástico (em inglês, *Stochastic Gradient Descent*) e ajustam os parâmetros de acordo com uma estimativa do gradiente da função de custo. Essa estimativa é calculada com base em uma amostra dos dados (Goodfellow et al., 2016). Apesar de serem mais rápidos (já que analisam apenas uma parte dos dados de treinamento, e não o conjunto completo) esses métodos precisam de ajustes na taxa de aprendizagem. Isso ocorre por que a amostragem aleatória insere um ruído no gradiente, e este ruído não desaparece nem mesmo no mínimo local Goodfellow et al. (2016), diferente do gradiente real, que chega a zero quando o mínimo é atingido. Os principais métodos para o ajuste do taxa de aprendizagem são os ajustes exponencial, linear e por passo (Goodfellow et al., 2016). Nos ajustes exponencial e linear, o taxa de aprendizagem é multiplicado por um termo dependente da época. No ajuste por passo (*step decay*), o taxa de aprendizagem é multiplicado por um fator fixo cada vez que se passarem um certo número de épocas.

Muitas alterações foram propostas com o intuito de melhorar a eficiência do SGD, como os algoritmos de momento (Polyak (1964) e Nesterov (1983)), que se baseiam no conceito físico de momento e adicionam um hiperparâmetro ao sistema, e algoritmos de taxa de aprendizagem adaptativa ((Kingma and Ba, 2014), (Hinton et al., 2012) e (Duchi et al., 2011)), que definem um valor de taxa de aprendizagem para cada peso da rede.

1.1 Objetivos

O algoritmo proposto, *Adjustable Step Decay*, é uma modificação do algoritmo *step decay* original. Essa modificação, melhor explicada no capítulo 3, visa reduzir o tempo necessário para treinar a rede a partir da diminuição do número de épocas necessário para a convergência. Para atingir esse objetivo, o *Adjustable Step Decay* altera o valor da taxa de aprendizagem em pontos proporcionais ao número de épocas do treinamento.

Essa redução é importante por impactar diretamente no tempo de treinamento da rede. Quanto mais rápida for a convergência, menor o número de épocas necessário para o treinamento da rede. Porém, uma convergência rápida só pode ser obtida com um ajuste correto da taxa de aprendizagem .

Testes foram realizados na proposição original, utilizando diferentes estruturas de rede e bases de dados. Esses testes geraram resultados positivos, onde a convergência foi atingida mais rapidamente. Os experimentos realizados neste trabalho têm como objetivo analisar a eficácia do *Adjustable Step Decay*, através da comparação dos resultados obtidos com o algoritmo com aqueles obtidos por

outras técnicas.

1.2 Estrutura do documento

No capítulo 2, será realizada a revisão da literatura, mostrando as principais técnicas para atualização do taxa de aprendizagem. No capítulo 3, o algotitmo *Adjustable Step Decay* será apresentado em mais detalhes. O capítulo 4 mostra os experimentos realizados e seus resultados. No capítulo 5 apresenta as conclusões.

2. Revisão da literatura

Como citado no capítulo anterior, existem diversas técnicas utilizadas para aumentar a eficiência do *Stochastic Gradient Descent*, sendo as principais os algoritmos de momento e os algoritmos de taxa de aprendizagem adaptativa, como Adam Kingma and Ba (2014), RMSProp Hinton et al. (2012) e AdaGrad Duchi et al. (2011), que serão abordados em mais detalhes nas seções 2.3 e 2.4.

No entanto, existem outras técnicas. AdaDelta (Zeiler, 2012) é uma variante do AdaGrad que limita o acúmulo dos gradientes anteriores a uma janela fixa de tamanho w. AdaMax (Kingma and Ba, 2014) é variante do Adam onde o momento de segunda ordem é substituído pelo momento de ordem infinita. Nadam (Dozat, 2016) é um algoritmo que combina o momento de Nesterov com o Adam.

2.1 Redes neurais convolucionais

Redes neurais convolucionais são similares às redes neurais comuns. No entanto, essas redes são construídas especificamente para trabalhar com imagens. Assim, é possível reduzir o número de parâmetros da rede. As camadas escondidas das redes convolucionais nem sempre estão totalmente conectadas às camadas anteriores. Existem três tipos principais de camadas usadas na construção de Redes Neurais Convolucionais: camadas convolucionais, camadas de *pooling* e camadas totalmente conectadas.

2.1.1 Camadas convolucionais

As camada convolucionais são fundamentais na construção de Redes Neurais Convolucionais. Essas camadas aplicam operações de convolução à imagem, e são compostas por uma série de filtros pequenos. Os neurônios dessas camadas se conectam apenas a uma parte da entrada, representada por um hiperparâmetro chamado de *campo de recepção*, reduzindo o número de conexões totais da camada e, por consequência, o número de parâmetros. Embora essas conexões sejam limitadas a uma área pequena no comprimento e largura, elas se extendem por toda

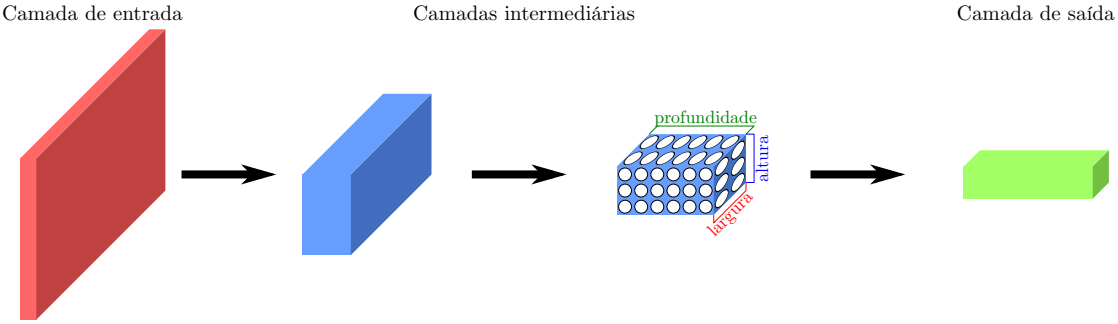

Figura 2.1: Uma rede neural convolucional com duas camadas intermediárias. A camada de entrada tem dimensões $W \times H \times C$, onde W e H são respectivamente a largura e a altura da imagem de entrada e C é o número de canais da imagem (3 para imagens coloridas). A cada camada, uma entrada em três dimensões é convertida em uma saída em três dimensões.

a profundidade da imagem. O tamanho do volume da saída é controlado por três outros hiperparâmetros: o *volume*, que é o número de filtros usados; o *passo*, que representa quantos *pixels* o filtro irá se mover, e o *preenchimento com zeros*, que representa a quantidade de zeros adicionados à borda da imagem. Além disso, os parâmetros são compartilhados entre unidades de mesma profundidade (que tem apenas o viés diferente), o que reduz o número de parâmetros únicos.

2.1.2 Camadas de *pooling*

Camadas de *pooling* são responsáveis pela diminuição espacial da representação, chamada de *downsampling*. Isso é importante para reduzir os parâmetros e ajudar a controlar o sobreajuste. Essas camadas operam utilizando o operador **MAX**, que escolhe o máximo entre as entradas. Essa camada não altera a profundidade, e não introduz novos parâmetros na rede porque calcula uma função fixa. Alguns trabalhos, como Springenberg et al. (2014), propõem a não utilização de camadas de *pooling*, substituindo por camadas convolucionais com passos maiores.

2.1.3 Camadas totalmente conectadas

As camadas totalmente conectadas são aquelas em que cada unidade está conectada a todas as unidades da camada anterior. Isso significa que cada neurônio tem um parâmetro para cada conexão mais um viés, tornando o ajuste dessas camadas mais complexo que as camadas convolucionais.

As camadas convolucionais podem ser convertidas em camadas totalmente conectadas. Nesse caso, a maioria dos parâmetros seria nulo, e os valores não-nulos teriam repetições (devido ao compartilhamento de parâmetros). Também é

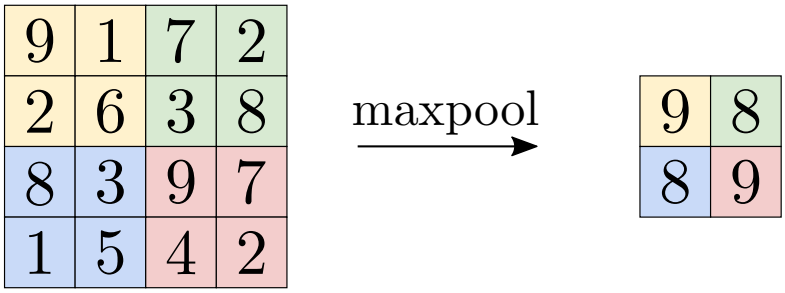

Figura 2.2: Representação da operação de *pooling* (*maxpool*) para redução de dimensionalidade. Cada cor representa um filtro, e esse filtro escolhe o maior valor. O resultado é uma saída com 75% dos dados da entrada.

possível converter camadas totalmente conectadas em convolucionais, calculando os hiperparâmetros necessários.

2.1.4 Sequência de camadas

A forma mais comum de se ordenarem camadas em uma Rede Neural Convolucional é usar uma combinação de uma ou mais camadas convolucionais, seguidas de uma camada de *pooling*. Essa combinação é repetida uma ou mais vezes, até que a imagem tenha sido processada o suficiente para seguir para uma camada totalmente conectada.

É preferível o uso de múltiplas camadas convolucionais com campo de recepção pequeno do que o uma camada com campo de recepção maior. Isso acontece por dois motivos. O primeiro é porque a redução no número de parâmetros. Suponha uma única camada com campo de recepção 7×7 e uma sequência de três camadas com campo de recepção 3×3, ambas com C canais. A camada 7×7 teria $C \times (7 \times 7 \times C) = 49C^2$ parâmetros, enquanto a sequência de camadas 3×3 teria $3 \times C \times (3 \times 3 \times C) = 27C^2$ parâmetros. O segundo motivo é que as não-linearidades da sequência de camadas podem tornar as características da entrada mais expressivas. Um ponto negativo é que mais memória é necessária para os cálculos de ajuste de parâmetros por retropropagação.

2.1.5 Tamanho das camadas

O tamanho das camadas também segue certos padrões. A camada de entrada, por exemplo, deve ser divisível por 2 múltiplas vezes; valores como 32, 64, 96 ou 512 são comuns.

Camadas convolucionais devem usar filtros de tamanho pequeno (como 3×3 ou 5×5), passo 1 e preenchimento suficiente apenas para que a camada não altere

o tamanho da entrada (em geral, o preenchimento pode ser calculado como $P = (F-1)/2$). Filtros maiores são comuns apenas na primeira camada convolucional (a que se liga à entrada).

Camadas de *pooling* tem, normalmente, campo de recepção 2×2 com passo 2, o que descarta 75% da entrada. Também é possível usar campos de recepção 3×3, mas isso torna o *downsampling* muito agressivo e reduz a perfomance da rede.

2.2 Métodos de redução da taxa de aprendizagem

Os métodos de redução da taxa de aprendizagem são aqueles em que a taxa é ajustado por um fator a cada conjunto de épocas. Dentro desse grupo, podemos destacar o ajuste linear, o ajuste exponencial e o ajuste por passo (*step decay*).

Ajuste linear O ajuste linear da taxa de aprendizagem é feito pela multiplicação da taxa de aprendizagem original por um termo linear dependente da época. A taxa de aprendizagem é calculada como

$$\eta_t = \eta_0 \cdot \frac{1}{t+1}, \tag{2.1}$$

onde η é a taxa de aprendizagem , e t é a época.

Ajuste exponencial O ajuste exponencial da taxa de aprendizagem é feito pela multiplicação da taxa de aprendizagem original por um termo exponencial dependente da época. A taxa de aprendizagem é calculado como

$$\eta_t = \eta_0 \cdot \mathrm{e}^{-\beta \cdot t}, \tag{2.2}$$

onde η é a taxa de aprendizagem , t é a época e β é um termo maior que zero que multiplica a época.

Ajuste por passo a ajuste da taxa de aprendizagem por passo é feito multiplicando a taxa de aprendizagem por uma taxa de decaimento cada vez que passa um certo número de épocas. Assim, a atualização da taxa de aprendizagem fica

$$\eta_t = \eta_0 \cdot k^{\frac{t}{d}}, \tag{2.3}$$

onde η é o taxa de aprendizagem , k é a taxa de decaimento, t é a época e d é o intervalo entre atualizações.

Em todos esses algoritmos, o ajuste é feito apenas com a redução da taxa de aprendizagem em intervalos definidos. No entanto, os hiperparâmetros devem ser definidos previamente. Além disso, para dados esparsos, o ideal é que o ajuste da taxa de aprendizagem seja feito para cada parâmetro, e não único (Lau, 2017). Para esses casos, os algoritmos de momento e de taxa de aprendizagem adaptativa são mais indicados.

2.3 Algoritmos de momento

Algoritmos de momento se baseiam no conceito físico de momento, também chamado de quantidade de movimento. Nesses algoritmos, há um conceito de *velocidade acumulada*, que representa a direção atual para onde a solução está caminhando. O novo passo é calculado com base no gradiente e na velocidade, fazendo com que o passo leve a solução para mais próximo do resultado.

2.3.1 Método de momento de Polyak

O método de momento proposto em Polyak (1964) se baseia na velocidade acumulada e no gradiente da função a ser minimizada. O valor da velocidade no tempo $t + 1$ é dado por

$$\mathbf{v}_{t+1} = \mu \mathbf{v}_t + g_t, \tag{2.4}$$

onde $\mathbf{v}$ é a velocidade, μ é o coeficiente de momento, η é a taxa de aprendizagem e g_t é o gradiente da função de custo no ponto atual. O novo parâmetro $\boldsymbol{\theta}_{t+1}$ é calculado como a soma do parâmetro atual com a velocidade calculada na equação 2.4, ou seja,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1}, \tag{2.5}$$

onde $\boldsymbol{\theta}$ é o vetor de parâmetros e $\mathbf{v}$ é a velocidade.

Como especificado em Sutskever et al. (2013), direções mais planas tem menor gradiente e, por isso, tendem a ser mantidas e amplificadas pelo método. Polyak (1964) também mostra que a convergência para um mínimo local é acelerada, requerendo menos iterações para atingir o mesmo nível de acurácia que o método de gradiente descendente.

2.3.2 Método de momento de Nesterov

O método proposto em Nesterov (1983) difere levemente do citado na subseção anterior: enquanto no método de Polyak a solução calcula a correção da rota *antes* de se mover, no método de Nesterov a correção é calculada *como se o movimento*

tivesse ocorrido. Assim, a velocidade pode ser calculada como

$$\mathbf{v}_{t+1} = \mu\mathbf{v}_t - \eta\nabla J(\boldsymbol{\theta}_t + \mu\mathbf{v}_t), \tag{2.6}$$

onde $\mathbf{v}$ é a velocidade, μ é o coeficiente de momento, η é a taxa de aprendizagem e o termo $\nabla J(\boldsymbol{\theta}_t + \mu\mathbf{v}_t)$ é o gradiente calculado no ponto para onde a solução seria originalmente direcionada. A atualização de $\boldsymbol{\theta}$ ocorre como mostrado na equação 2.5. A figura 2.3 mostra a diferença entre os dois métodos. No momento de Polyak, o gradiente (seta azul) é calculado no ponto $\boldsymbol{\theta}_t$, enquanto que no momento de Nesterov o cálculo é feito no ponto $\boldsymbol{\theta}_t + \mu\mathbf{v}_t$. Isso dá ao momento de Nesterov uma vantagem: ela é capaz de "desacelerar" antes que o gradiente comece a subir novamente (Ruder, 2016).

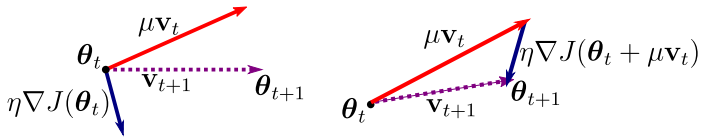

Figura 2.3: Representação visual do cálculo de v_{t_1} em cada método de momento. À esquerda, o momento de Polyak; à direita, o momento de Nesterov.

2.4 Algoritmos de taxa de aprendizagem adaptativa

Os algoritmos de taxa de aprendizagem adaptativa são aqueles em que o valor da taxa de aprendizagem é modificado por um termo definido em cada algoritmo. Nessa categoria, podemos destacar três algoritmos: Adagrad, RMSProp e Adam. Além destes, existem outros como Adamax (Kingma and Ba, 2014), AdaDelta (Zeiler, 2012) e Nadam (Dozat, 2016).

2.4.1 Adagrad

Adagrad, proposto por Duchi et al. (2011), é um algoritmo que ajusta a taxa de aprendizagem de acordo com os a frequência dos parâmetros. A taxa de aprendizagem dos parâmetros mais frequentes é atualizada mais vezes, porém com valores menores; para os parâmetros menos frequentes, a taxa de aprendizagem é atualizada com valores e intervalos maiores. Essa característica o torna ideal para trabalhar com dados esparsos (Ruder, 2016). A atualização do vetor de parâmetros $\boldsymbol{\theta}$ segue a fórmula

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t, \tag{2.7}$$

onde η é a taxa de aprendizagem, g_t é o gradiente da função de custo no tempo t e ϵ é um termo que serve para evitar a divisão por zero, sendo normalmente da ordem de 10^{-8}. G_t é uma matriz diagonal dada por

$$G_t = \begin{vmatrix} \sum_{T=0}^{t} (g_{T,1})^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sum_{T=0}^{t} (g_{T,i})^2 \end{vmatrix}, \tag{2.8}$$

onde $g_{T,i}$ é o gradiente da função de custo no tempo T com relação ao i-ésimo parâmetro do vetor $\boldsymbol{\theta}$.

A principal vantagem do Adagrad é que não há necessidade de ajustar a taxa de aprendizagem manualmente (Ruder, 2016), sendo o valor de 0.01 é o mais encontrado e recomendado. Outra vantagem é que a taxa de aprendizagem é ajustada para cada parâmetro, o que é útil quando a matriz de gradientes é esparsa (Duchi et al., 2011). Já a maior desvantagem é o acúmulo dos quadrados dos gradientes no denominador: como os valores da diagonal da matriz G_t se tornam muito grandes, o segundo termo da equação 2.7 é reduzido para muito próximo de zero, reduzindo a taxa de aprendizagem . Isso faz com que o modelo não consiga mais aprender, estagnando ou até reduzindo a acurácia para um número grande de iterações.

2.4.2 RMSProp

RMSProp é um algoritmo proposto em Hinton et al. (2012)[1]. Ele é similar ao Adagrad, mas substitui a soma dos quadrados dos gradientes anteriores por uma média móvel E dos quadrados dos gradientes anteriores. Essa média móvel é definida como

$$E\left[g^2\right]_t = 0.9 \cdot E\left[g^2\right]_{t-1} + 0.1 \cdot g_t^2, \tag{2.9}$$

onde g é o gradiente da função de custo, g_t é o gradiente da função de custo no tempo t, e $E[g^2]_x$ é a média móvel dos quadrados dos gradientes no tempo x. A equação 2.7 é reescrita como

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t, \tag{2.10}$$

onde η é a taxa de aprendizagem (com valor sugerido de 0.001), g_t é o gradiente da função de custo no tempo t, $E[g^2]_t$ é a média móvel dos quadrados dos gradientes no tempo t e ϵ é um termo que serve para evitar a divisão por zero. O uso da

[1]Não publicado. A proposição do algoritmo se deu em um material de aula de uma disciplina no Coursera.

média móvel diminui o impacto da soma dos quadrados no denominador, evitando o problema da estagnação do Adagrad.

Embora o RMSProp tenha bons resultados, ainda não há entendimento prático sobre seu sucesso (Dauphin et al., 2015).

2.4.3 Adam

Adam é outro método para ajuste de parâmetros, proposto em Kingma and Ba (2014). Esse método utiliza as médias móveis dos gradientes e dos quadrados dos gradientes para atualizar os parâmetros. Essas médias são inicializadas como zero, e são calculadas como

$$\mathbf{m}_t = \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot g_t \tag{2.11}$$

$$\mathbf{v}_t = \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot g_t^2, \tag{2.12}$$

onde $\mathbf{m}_t$ é o vetor das médias dos gradientes no tempo t, $\mathbf{v}_t$ é o vetor das médias dos quadrados dos gradientes no tempo t, g_t é o gradiente da função de custo, β_1 é p peso atribuído à media dos gradientes (com valor sugerido de 0.9) e β_2 é o peso atribuído à media dos quadrados dos gradientes (com valor sugerido de 0.999).

No entanto, esses valores de $\mathbf{m}_t$ e $\mathbf{v}_t$ são enviesados e tendem a mover a solução em direção ao zero, principalmente no início e quando os valores de β_1 e β_2 são próximos de 1 (Ruder, 2016). A razão para isso é que os passos iniciais tendem a direcionar $\mathbf{m}_t$ e $\mathbf{v}_t$ para $\mathbf{m}_{t-1}$ e $\mathbf{v}_{t-1}$ (inicialmente nulos) devido aos altos valores de β_1 e β_2. Isso torna o crescimento de $\mathbf{m}_t$ e $\mathbf{v}_t$ lento. Para evitar isso, esses valores são corrigidos de acordo com as equações

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t}, \tag{2.13}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t}, \tag{2.14}$$

onde $\hat{\mathbf{m}}_t$ é o vetor corrigido das médias dos gradientes, $\hat{\mathbf{v}}_t$ é o vetor corrigido das médias dos quadrados dos gradientes, $\mathbf{m}_t$ é o vetor das médias dos gradientes, $\mathbf{v}_t$ é o vetor das médias dos quadrados dos gradientes e β_x^t é o peso (β_1 ou β_2) elevado ao número de épocas. A explicação para essa correção é dada em Kingma and Ba (2014), seção 3.

A atualização dos parâmetros segue uma ideia similar às equações 2.7 e 2.10, substituindo G_t ou $E\left[g^2\right]_t$ por $\hat{\mathbf{v}}_t$ e g_t por $\hat{\mathbf{m}}_t$, resultando na equação

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \cdot \hat{\mathbf{m}}_t, \tag{2.15}$$

onde $\boldsymbol{\theta}$ é o vetor de pesos, η é a taxa de aprendizagem , $\hat{\mathbf{m}}_t$ é o vetor corrigido das médias dos gradientes no tempo t, $\hat{\mathbf{v}}$ é o vetor corrigido das médias dos quadrados dos gradientes no tempo t e ϵ serve para evitar a divisão por zero.

Os autores mostram que o algoritmo funciona bem na prática, tendo eficiência igual ou superior ao Adagrad e ao RMSProp nos experimentos realizados (Kingma and Ba, 2014). Nos experimentos realizados pelo autor, o Adam convergiu consideravelmente mais rápido que Adagrad para redes neurais convolucionais.

3. Adjustable Step Decay

O algoritmo *Adjustable Step Decay*, objetiva melhorar o *step decay* original. No algoritmo original, a alteração se dá em pontos fixos pré-definidos. A alteração proposta é que a alteração ocorra em pontos relativos à quantidade de épocas, fazendo com que os ajustes de taxa de aprendizagem sejam automaticamente adaptados ao número de épocas. Na figura 3.1, as linhas vermelhas representam o ajuste em pontos fixos (após 3, 6 e 9 épocas), enquanto que as linhas verdes representam o ajuste em pontos proporcionais ao total (após 30%, 60% e 90% do total de épocas). No lado esquerdo da figura, as linhas se sobrepõem porque os pontos onde ocorre a atualização coincidem. No lado direito da figura, a linha vermelha decresce nos mesmos pontos em que no lado esquerdo, enquanto que a linha verde se adaptou ao novo número de épocas.

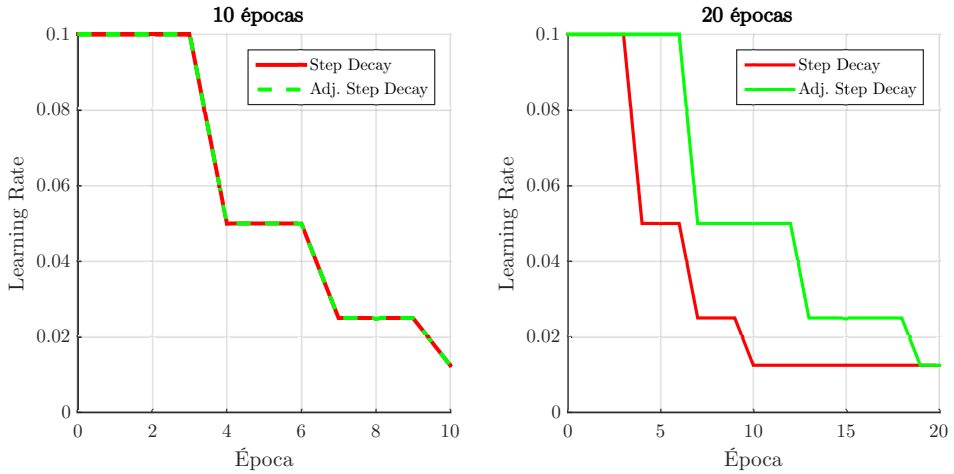

Figura 3.1: Ajuste fixo (vermelho) e proporcional (verde).

A redução da taxa de aprendizagem é importante para a convergência no *Stochastic Gradient Descent*. Nas etapas iniciais, o uso de passos maiores ajuda a acelerar a convergência. No entanto, à medida que o algoritmo avança, é necessária a redução do passo, para que seja possível ajustes mais finos (Senior et al., 2013).

Em Shwartz-Ziv and Tishby (2017), os autores defendem a existência de duas etapas distintas do *Stochastic Gradient Descent*, com relação à informação mútua (a quantidade relevante de *bits* da entrada que contém informações sobre a saída) das camadas. Na primeira etapa, chamada de *drift phase*, os gradientes são maiores do que os desvios padrão, e a informação mútua sobre a saída é incrementada com a diminuição do erro empírico causada pela redução dos gradientes. Já na segunda etapa, denominada *diffusion phase*, os gradientes passam a ser menores que os desvios-padrão. Essa etapa adiciona ruído aleatório aos gradientes, minimizando a informação mútua da entrada a cada camada e comprimindo a representação da informação, priorizando as características mais importantes.

Considerando essas duas fases, é possível perceber a importância de se alterar a taxa de aprendizagem durante o treinamento. O valor da taxa de aprendizagem deve ser mantido alto durante a primeira fase do treinamento, onde os gradientes são maiores e passos grandes são desejados para atingir mais rapidamente a região próxima do mínimo. Já durante a segunda fase, a taxa de aprendizagem deve ser reduzido para permitir que sejam feitos ajustes mais finos.

Com base nessas observações, a proposição do algoritmo sugere que a taxa de aprendizagem seja mantido alto durante a maior porcentagem do número de épocas do treinamento, aproveitando a *drift phase* para que a solução se mova mais rapidamente em direção ao ótimo; e que as reduções da taxa de aprendizagem ocorram durante a segunda fase de forma rápida (grandes reduções) e proporcional ao total de épocas, garantindo ajuste automático ao problema.

4. Experimentos

Os experimentos foram realizados comparando o algoritmo *Adjustable Step Decay* com três outros algoritmos: Adam, RMSProp e Adagrad. Esses algoritmos foram escolhidos por serem os que possuem melhores resultados entre os algoritmos, e seus parâmetros foram ajustados de acordo com o recomendado pelos autores. Esses parâmetros estão listados na tabela 4.1.

Tabela 4.1: Valores padrão dos algoritmos usados para comparação.

Algoritmo	***Learning Rate***	$\boldsymbol{\alpha}$	$\boldsymbol{\beta_1}$	$\boldsymbol{\beta_2}$	$\boldsymbol{\epsilon}$
Adagrad	0.01	-	-	-	-
RMSProp	0.001	0.9	-	-	-
Adam	0.001	-	0.9	0.99	10^{-8}

Experimentos originais Os experimentos da proposta original também compararam o *Adjustable Step Decay* com os algoritmos citados na tabela 4.1, porém com parâmetros diferentes para o RMSProp. Foram utilizados dois conjuntos de dados (CIFAR-10 e CIFAR-100) e três arquiteturas de rede (LeNet5, VGG19 e ResNet-101, esta última apenas com treinamento em 10 épocas). Os resultados foram similares aos apresentados neste trabalho, com o *Adjustable Step Decay* tendo maior acurácia em todos os casos.

4.1 Configurações do experimento

Os códigos usados nos experimentos foram implementados em Python (usando a biblioteca PyTorch), e foram executados em um computador com as configurações mostradas na tabela 4.2. Os experimentos utilizaram, para todos os algoritmos, duas arquiteturas de rede diferentes: LeNet5 (Le Cun et al., 1989) e VGG19 (Simonyan and Zisserman, 2014). Essas redes têm, respectivamente, 5 e 19 camadas, e foram escolhidas para representar diferentes profundidades de rede.

Tabela 4.2: Configurações do computador usado para a execução dos experimentos

Sistema Operacional	Linux Mint 18.2 *Sonya*
Processador	Intel Core I5-7400 @ 3.0GHz
Memória	16GB
GPU	NVidia GeForce GTX 1060 3GB

A arquitetura LeNet5 consiste em uma camada de entrada, três camadas escondidas (denominadas $H1$, $H2$ e $H3$) e uma camada de saída. A camada de entrada tem 256 unidades. A camada $H1$ é composta de 768 unidades, agrupados em 12 mapas de características 8×8 com 64 unidades cada. A camada $H2$ é composta de 192 unidades organizadas em 12 mapas de características 4×4 com 16 unidades cada. Todas as unidades de um mesmo mapa de características tem o mesmo conjunto de pesos, mas com um viés diferente. A camada $H3$ tem 30 unidades, e a camada de saída tem 10 unidades; nessas camadas, cada unidade tem seus próprios pesos e viés.

A arquitetura VGG-19 possui 16 camadas convolucionais (com largura variando entre 64 e 256 canais), 3 camadas totalmente conectadas (duas com 4096 canais e uma com 1000 canais), 5 camadas de *pooling* e uma camada *soft-max*. Uma característica dessa rede é o uso de convoluções 3×3 em sequência no lugar de convoluções maiores, como modo de reduzir o número de parâmetros. Mesmo com essas reduções, esta rede tem 144 milhões de parâmetros.

A função de ativação utilizada foi a ReLu (Glorot et al., 2011), definida por

$$f(x) = \max(0, x). \tag{4.1}$$

Essa função acelera a convergência do gradiente (Krizhevsky et al., 2012), e não precisa de operações de alto custo computacional.

Após experimentos, foi encontrado um conjunto de hiperparâmetros que parece funcionar bem em diferentes cenários, tornando o ajuste desses hiperparâmetros quase desnecessário. Os hiperparâmetros estão descritos na tabela 4.3. Também foram usados o momento de Nesterov e esquema de ajuste de pesos baseado na norma L2.

Tabela 4.3: Hiperparâmetros do *Adjustable Step Decay*.

Parâmetro	**Valor**
Momento	0.9
taxa de aprendizagem inicial	0.1
Decaimento	0.005
Pontos de atualização	60%, 80% e 90% do total de épocas

O treinamento do algoritmo *Adjustable Step Decay* foi realizado em 10, 30 e 100 épocas, como forma de verificar a diminuição da performance para casos onde não há interesse em gastar muito tempo com treinamento. Foram utilizados três conjuntos de dados, explicados na tabela 4.4.

Cada experimento consiste na em 10 execuções de cada um dos algoritmos. Isso serve para que se possa fazer uma análise da média dos resultados, uma vez que existem componentes aleatórios envolvidos.

Tabela 4.4: Descrição das bases de dados utilizadas nos experimentos.

Base de dados	**Classes**	**Itens (Treino)**	**Itens (Teste)**	**Resolução da imagem**
CIFAR-10	10	50000	10000	32x32x3
CIFAR-100	100	50000	10000	32x32x3
MNIST	10	60000	10000	28x28x1

4.2 Resultados

4.2.1 Experimento 1: MNIST, LeNet5

Na figura 4.1 é apresentado o gráfico da acurácia em relação à época para o conjunto de teste, e na tabela 4.5 estão os valores finais da acurácia para o experimento com a arquitetura LeNet5 e o conjunto de dados MNIST. Todos os algoritmos alcançam muito rapidamente valores de acurácia altos, e com diferença mínima. É interessante notar que o melhor resultado, ainda que por pouco, é obtido pelo *Adjustable Step Decay* com o treinamento em 30 épocas, e o treinamento em 100 e 10 épocas apresentaram resultados similares. Isso significa que o treinamento da rede pode ser acelerado sem perda significativa de acurácia. Esse experimento foi executado em aproximadamente 6 horas.

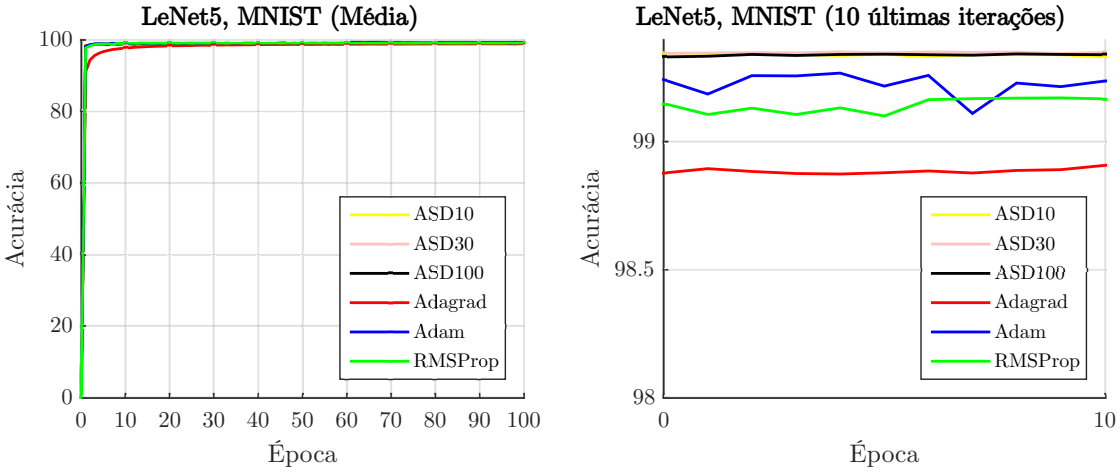

Figura 4.1: Gráficos de acurácia para a arquitetura LeNet5 com o banco de dados CIFAR-10. À esquerda, o gráfico completo; à direita, as 10 últimas iterações. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

Tabela 4.5: Resultados dos experimentos com a arquitetura LeNet5 e a base de dados MNIST. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

	Acurácia (Treinamento)		**Acurácia (Teste)**	
Algoritmo	**Média**	**Desvio padrão**	**Média**	**Desvio padrão**
ASD30	99.969	0.004	99.355	0.034
ASD10	99.831	0.016	99.347	0.061
ASD100	99.993	0.002	99.342	0.065
Adam	99.990	0.013	99.239	0.122
RMSProp	99.988	0.005	99.169	0.095
Adagrad	99.557	0.295	98.909	0.142

4.2.2 Experimento 2: MNIST, VGG19

Na figura 4.2 é apresentado o gráfico da acurácia em relação à época para o conjunto de teste, e na tabela 4.6 estão os valores finais da acurácia para o experimento com a arquitetura VGG19 e o conjunto de dados MNIST. Novamente, os algoritmos alcançam muito rapidamente valores altos e com diferença mínima. O *Adjustable Step Decay* teve performance superior aos outros algoritmos em todos os casos. Esse experimento foi executado em aproximadamente 6 horas.

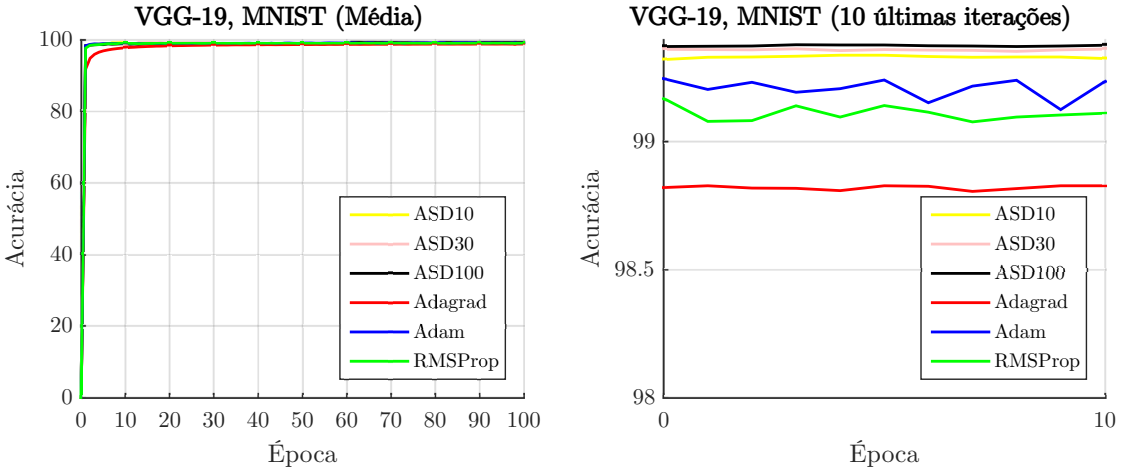

Figura 4.2: Gráficos de acurácia para a arquitetura VGG19 com o banco de dados MNIST. À esquerda, o gráfico completo; à direita, as 10 últimas iterações. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

Tabela 4.6: Resultados dos experimentos com a arquitetura VGG19 e a base de dados MNIST. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

	Acurácia (Treinamento)		**Acurácia (Teste)**	
Algoritmo	**Média**	**Desvio padrão**	**Média**	**Desvio padrão**
ASD100	99.993	0.002	99.379	0.053
ASD30	99.970	0.004	99.364	0.048
ASD10	99.835	0.014	99.353	0.057
Adam	99.962	0.081	99.237	0.056
RMSProp	99.988	0.004	99.113	0.068
Adagrad	99.598	0.271	98.829	0.155

4.2.3 Experimento 3: CIFAR10, VGG19

Na figura 4.3 é apresentado o gráfico da acurácia em relação à época para o conjunto de teste, e na tabela 4.7 estão os valores finais da acurácia para o experimento com a arquitetura VGG19 e o conjunto de dados CIFAR-10. É possível observar que o *Adjustable Step Decay* com 100 épocas de treinamento apresentou maior acurácia. Também vale a pena ressaltar que o resultado do algoritmo com apenas 10 épocas de treinamento tem desempenho comparável com o RMSProp e o Adam. Esse experimento foi executado em aproximadamente 60 horas.

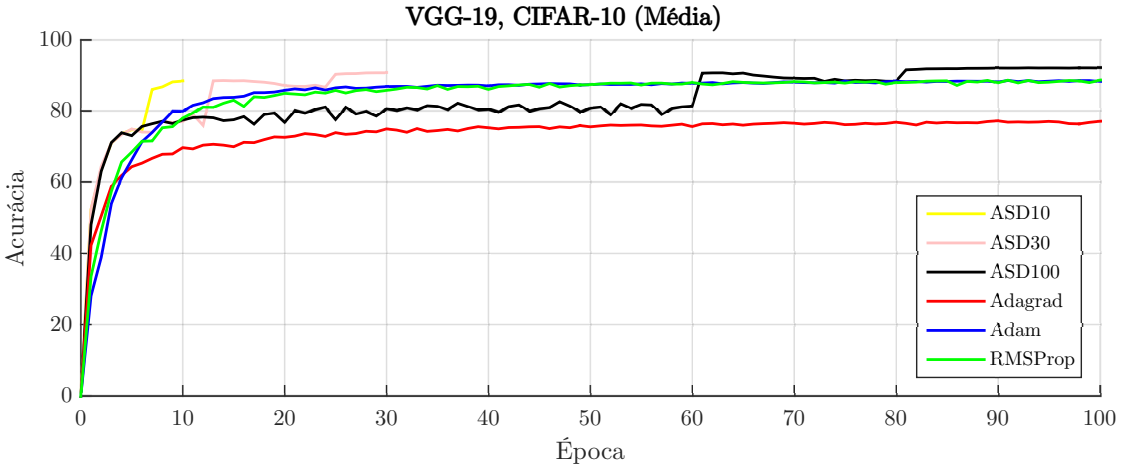

Figura 4.3: Gráficos de acurácia para a arquitetura VGG19 com o banco de dados CIFAR-10. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

Tabela 4.7: Resultados dos experimentos com a arquitetura VGG-19 e a base de dados CIFAR-10. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

	Acurácia (Treinamento)		**Acurácia (Teste)**	
Algoritmo	**Média**	**Desvio padrão**	**Média**	**Desvio padrão**
ASD100	99.996	0.003	92.066	0.213
ASD30	99.828	0.026	90.707	0.131
ASD10	93.905	0.128	88.377	0.239
RMSProp	99.284	0.035	88.586	0.431
Adam	99.465	0.065	88.252	0.450
Adagrad	99.833	0.064	77.090	4.919

4.2.4 Experimento 4: CIFAR10, LeNet5

Na figura 4.4 é apresentado o gráfico da acurácia em relação à época para o conjunto de teste, e na tabela 4.8 estão os valores finais da acurácia para o experimento com a arquitetura LeNet5 e o conjunto de dados CIFAR-10. Os resultados mostram que o *Adjustable Step Decay* foi melhor em todos os casos, com uma diferença de apenas 0.708 pontos percentuais entre as execuções com treinamento em 100 e 30 épocas. Novamente, o resultado indica que o treinamento pode ser realizado em menos tempo sem perda significativa de acurácia. Esse experimento foi executado em aproximadamente 60 horas.

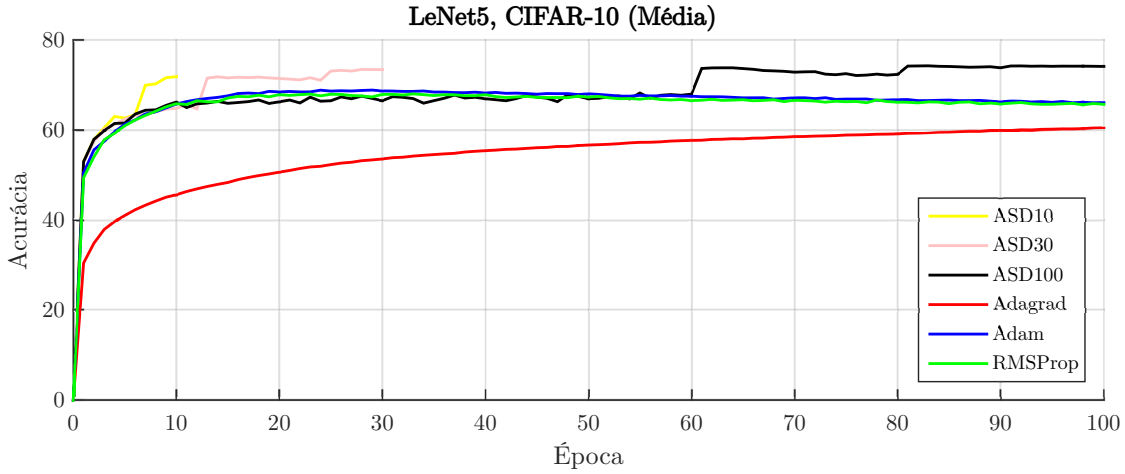

Figura 4.4: Gráficos de acurácia para a arquitetura LeNet5 com o banco de dados CIFAR-10. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

Tabela 4.8: Resultados dos experimentos com a arquitetura LeNet5 e a base de dados CIFAR-10. ASDX é o *Adjustable Step Decay* com X épocas de treinamento.

	Acurácia (Treinamento)		**Acurácia (Teste)**	
Algoritmo	**Média**	**Desvio padrão**	**Média**	**Desvio padrão**
ASD100	93.158	0.653	74.149	0.375
ASD30	85.826	0.568	73.441	0.553
ASD10	76.891	0.674	71.872	0.508
Adam	90.509	1.009	66.128	0.820
RMSProp	92.030	1.127	65.763	0.980
Adagrad	62.474	5.051	60.503	3.909

4.2.5 Experimento 5: CIFAR-100, VGG19

Na figura 4.5 é apresentado o gráfico da acurácia em relação à época para o conjunto de teste, e na tabela 4.9 estão os valores finais da acurácia para o experimento com a arquitetura VGG19 e o conjunto de dados CIFAR-100. Nesse experimento, o *Adjustable Step Decay* foi executado apenas com treinamento em 100 épocas. Os resultados mostram que o *Adjustable Step Decay* apresentou uma acurácia 12.49 pontos percentuais acima do segundo melhor, e quase o dobro da acurácia do último. Esse experimento foi executado em aproximadamente 60 horas.

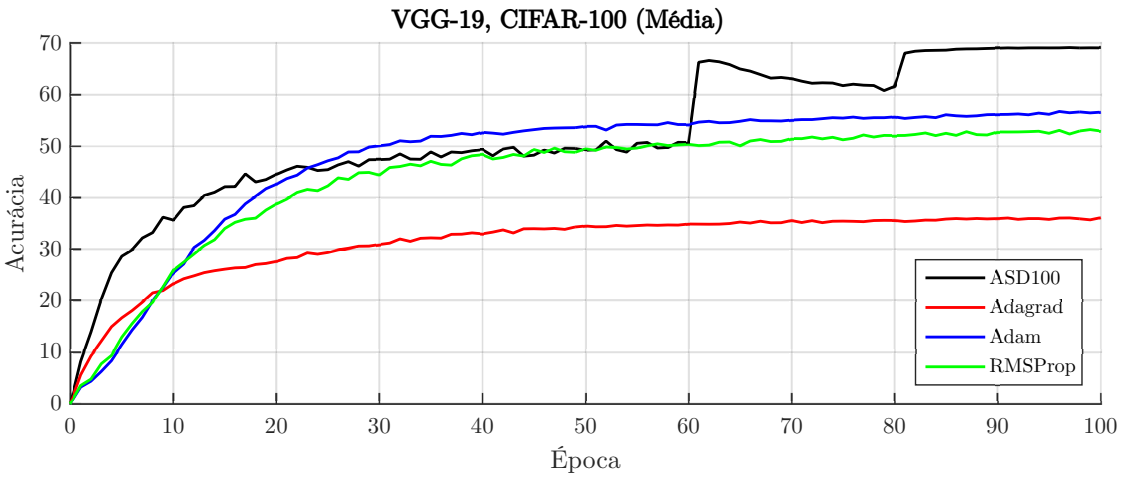

Figura 4.5: Gráfico de acurácia para a arquitetura VGG19 com o banco de dados CIFAR-100. ASD100 é o *Adjustable Step Decay* com 100 épocas de treinamento.

Tabela 4.9: Resultados dos experimentos com a arquitetura VGG19 e a base de dados CIFAR-100. ASD100 é o *Adjustable Step Decay* com 100 épocas de treinamento.

	Acurácia (Treinamento)		**Acurácia (Teste)**	
Algoritmo	**Média**	**Desvio padrão**	**Média**	**Desvio padrão**
ASD100	99.970	0.005	69.127	0.291
Adam	96.919	0.393	56.628	0.985
RMSProp	95.724	0.523	52.968	1.267
Adagrad	99.733	0.440	36.108	4.752

5. Conclusões

Com este trabalho, concluímos que o algoritmo *Adjustable Step Decay* apresenta bons resultados quando comparado com os algoritmos selecionados. Em um dos experimentos, o algoritmo apresentou um ganho de mais de 30 pontos percentuais em relação ao Adagrad. Um fato interessante é que os resultados dos treinamentos em 30 e 10 épocas ficaram próximos (menos de 5 pontos percentuais) dos obtidos com o treinamento em 100 épocas, inclusive superando em um dos cenários. Isso possibilita a redução do tempo de treinamento, com perda mínima de acurácia.

A tabela 5.1 mostra a classificação dos algoritmos em cada experimento. O *Adjustable Step Decay* ficou sempre com a melhor colocação, tendo também a segunda nos experimentos em que as três variações foram executadas. A variante com treinamento em 100 épocas foi o melhor em 80% dos cenários, perdendo apenas em um dos experimentos.

Outro ponto a ser notado é a curva característica do *Adjustable Step Decay*. Logo após as alterações na taxa de aprendizagem, há um incremento na acurácia, mais facilmente notada na primeira alteração (em 60% do total de épocas). Isso reforça a importância do ajuste do valor da taxa de aprendizagem para o treinamento. Com o valor fixo, a acurácia fica estagnada ou começa a cair; logo após o ajuste, a acurácia aumenta. No entanto, o motivo para esse comportamento da curva de acurácia ainda não é claro.

Tabela 5.1: Classificação dos algoritmos em cada experimento.

	Ex.1	**Ex.2**	**Ex.3**	**Ex.4**	**Ex.5**
ASD10	4°	*3°*	-	*2°*	*3°*
ASD30	*2°*	*2°*	-	*1°*	*2°*
ASD100	*1°*	*1°*	*1°*	*3°*	*1°*
Adam	5°	4°	*2°*	4°	4°
Adagrad	6°	6°	4°	6°	6°
RMSProp	*3°*	5°	*3°*	5°	5°

Trabalhos futuros podem analisar o *Adjustable Step Decay* em mais bases de dados e com outras arquiteturas de rede (principalmente arquiteturas mais profun-

das), como forma de verificar se o resultado apresenta as mesmas características. Também podem ser feitos experimentos comparando com outros algoritmos, como os citados no capítulo 2.

Bibliografia

Dauphin, Y. N., de Vries, H., Chung, J., and Bengio, Y. (2015). Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390.

Dozat, T. (2016). Incorporating nesterov momentum into adam.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

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

Hinton, G., Srivastava, N., and Swersky, K. (2012). Neural networks for machine learning. lecture 6e. *BIBLIOGRAPHY BIBLIOGRAPHY*.

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

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

Lau, S. (2017). Learning rate schedules and adaptive learning rate methods for deep learning.

Le Cun, Y., Jackel, L., Boser, B., Denker, J., Graf, H., Guyon, I., Henderson, D., Howard, R., and Hubbard, W. (1989). Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46.

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, 2nd edition.

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

Ruder, S. (2016). An overview of gradient descent optimization algorithms.

Senior, A., Heigold, G., Ranzato, M., and Yang, K. (2013). An empirical study of learning rates in deep neural networks for speech recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Vancouver, CA.

Shwartz-Ziv, R. and Tishby, N. (2017). Opening the black box of deep neural networks via information. *CoRR*, abs/1703.00810.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA. PMLR.

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.