COPPE/UFRJ (17/NOV/2019)
Redes Neurais sem Peso

Trabalho I – Analise da Base NMINST com WISARD e ClusWISARD

Nome: Luiz Marcio Faria de Aquino Viana
CPF: 024.723.347-10
RG: 08855128-8 IFP-RJ

1. INTRODUCAO

O objetivo deste trabalho é realizar uma analise de desempenho dos modelos de redes neurais sem peso WISARD e ClusWISARD no reconhecimento de números manuscritos. Para este estudo utilizamos como referencia a base NMINST e efetuamos uma analise comparativa entre o modelo de rede neural sem peso WISARD e o modelo de rede neural convolucional profunda, com o objetivo de avaliar o tempo de treinamento, o tempo de classificação, e o desmpenho final de cada um dos modelos.

Neste experimento, verificamos uma das características da rede neural WISARD que é a rapidez no treinamento dos dados. Esta característica permite que padrões de imagens sejam aprendidos muito rapidamente. Durante os experimentos, a rede neural WISARD efetuou o aprendizado de 60.000 imagens da base NMINST em apenas 2,1732 segundos em media para um endereçamento de 3 bits, enquanto a rede neural convolucional profunda analisada efetuou o mesmo treinamento em 135,5393 segundos. Isto é, o tempo gasto pela rede neural WISARD no treinamento dos dados foi de 1,6% do tempo gasto pela rede neural convolucional profunda avaliada.

Verificamos também que quando o espaço de endereçamento é pequeno, por exemplo de 3 bits, ocorrem muitos conflitos na classificação dos dados. Para estes casos a rede neural convolucional profunda apresenta um desempenho melhor.

O tempo médio de classificação do modelo WISARD com 3 bits de endereçamento foi de 97,5255 segundos, porém quando o espaço de endereçamento foi ampliado para 28 bits ou mais, o tempo médio para classificação dos dados foi reduzido para menos que 1,1053 segundos, enquanto o algoritmo utilizado pela rede neural convolucional profunda obteve um tempo médio de 4,3903 segundos.

Além disso, as redes neurais WISARD apresentaram um resultado pior em relação as redes convolucionais profundas quando comparamos a acurácia dos dois modelos. No experimento, a rede neural convolucional profunda analisada obteve um resultado de 98,3% de acerto, enquanto a rede neural WISARD obteve resultados que variaram de 69,80% à 92,33%.

Desta forma, identificamos que o desempenho das redes neurais WISARD e ClusWISARD são fortemente influenciados pelo tamanho de espaço de endereçamento, e que para as bases NMINST obtivemos os melhores desempenhos com endereçamentos entre 32 e 37 bits.

2. CONCEITOS DE REDES NEURAIS CONVOLUCIONAIS PROFUNDAS

As redes neurais profundas são modelos de redes neurais fortemente conectadas que apresentam uma camada de entrada, duas ou mais camadas ocultas e uma camada de saída.

As redes neurais convolucionais são modelos de redes neurais profundas que são largamente utilizadas na classificação de imagens. Este modelo de rede neural procura separar em cada camada as características que definem as imagens.

Quando estudamos imagens com maior profundidade de cores, como imagens RGB, podemos separar os elementos que formam a imagem em três vetores, correspondendo as intensidades de vermelho, verde e azul presentes na imagem. Desta forma, as características das imagens são melhor preservadas e o resultado obtido com as redes neurais convolucionais nestas tarefas, são melhores do que os resultados obtidos por outros modelos de redes neurais.

2.1. PARÂMETROS CONSIDERADOS NO EXPERIMENTO

No experimento realizado utilizamos uma rede convolucional profunda com uma camada de entrada, duas camadas convolucionais ocultas, com 64 e 32 features, e uma camada de saída softmax com 10 elementos, representando cada um dos algarismos aprendidos. Esta rede neural convolucional profunda foi treinada através de 1 época de treinamento com uma base de 60.000 imagens de caracteres manuscritos.

Apos o treinamento o modelo de rede neural convolucional profunda foi aplicado na classificação de uma base de teste com 10.000 imagens de caracteres manuscritos.

3. CONCEITOS DE REDES NEURAIS SEM PESO

3.1. REDES NEURAIS SEM PESO - WISARD

As redes neurais sem peso procuram mapear os bits da imagem em um vetor de endereçamento de memoria que armazena o aprendizado obtido durante a fase de treinamento da rede neural, e neste experimento utilizamos uma rede neural WISARD com 3, 28, 32, 37 e 64 bits de endereçamento.

Desta forma, em uma rede neural WISARD com um endereçamento com 3 bits, podemos mapear três pixels aleatórios da imagem nos três bits de endereçamento, permitindo que ate $2^3 = 8$ posições de memoria sejam mapeadas por cada elemento. Para um endereçamento de 5 bits, podemos mapear até $2^5 = 32$ posiçoes de memória e assim por diante.

3.2. REDES NEURAIS SEM PESO – ClusWISARD SUPERVISED

As redes neurais sem peso ClusWISARD Supervised *(Figura 1)*, procuram efetuar o treinamento de uma base de dados de entrada, separando os dados em uma classe inicial com os valores referentes aos *Labels* fornecidos como entrada, que no caso da base NMINST, são valores entre 0 e 9.

Em uma rede neural sem peso ClusWISARD Supervised, as classes geradas a partir dos *Labels* de entrada, são separadas em subclasses definidas pelos múltiplos discriminadores associados a elas. Esta separação ocorre em função da similaridade encontrada nas imagens do NMINST para a representação de cada *Label* de entrada. O número de subclasses criadas é dado pelo atributo *Discriminator Limit*, que determina a quantidade máxima de subclasses que podem ser criadas pelo modelo.

Figura 1: *Modelo de rede neural sem peso ClusWISARD Supervised*

3.3. REDES NEURAIS SEM PESO – ClusWISARD UNSUPERVISED

As redes neurais sem peso ClusWISARD Unsupervised *(Figura 2)*, procuram efetuar o treinamento de uma base de dados de entrada, separando os dados em uma única classe raiz, que são separadas em subclasses definidas pelos múltiplos discriminadores associados a elas. Esta separação ocorre em função da similaridade encontrada nas imagens do NMINST para a representação de cada *Label* de entrada. O número de subclasses criadas é dado pelo atributo *Discriminator Limit*, que determina a quantidade máxima de subclasses que podem ser criadas pelo modelo.

Figura 2: *Modelo de rede neural sem peso ClusWISARD Unsupervised*

3.4. PARÂMETROS CONSIDERADOS NO EXPERIMENTO

Para permitir este mapeamento, a imagem foi preprocessada, e a profundidade de cores reduzida para uma imagem em preto e branco, *Figura 3*. Em seguida, a rede neural WISARD foi treinada com uma base de treinamento com 60.000 imagens de caracteres manuscritos.

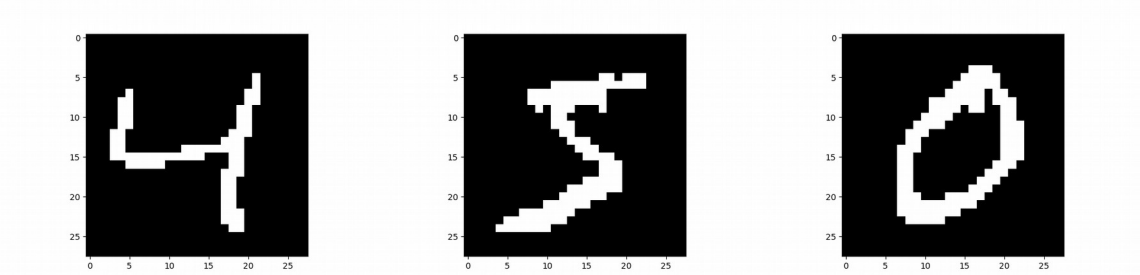

Figura 3: *Exemplo de imagem em preto e branco obtida apos o preprocessamento dos dados*

Apos o treinamento o modelo de rede neural WISARD foi aplicado na classificação de uma base com 10.000 imagens de caracteres manuscritos. A *Figura 4*, apresenta dois exemplos de imagens mentais geradas durante o aprendizado da rede neural WISARD.

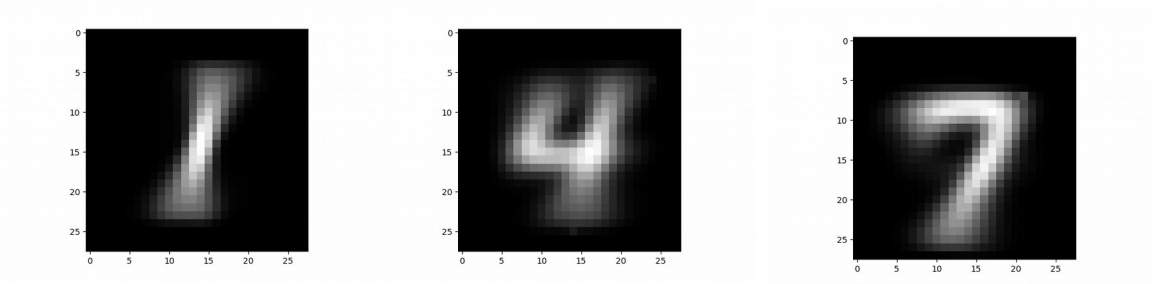

Figura 4: *Exemplo de imagem mental obtida apos o treinamento da rede neural WISARD*

A mesma base de treinamento e teste foram aplicadas aos modelos de rede neural WISARD e de rede neural convolucional profunda analisados.

4. ANÁLISE DOS RESULTADOS

4.1. RESULTADOS OBTIDOS COM REDES NEURAIS CONVOLUCIONAIS PROFUNDAS

Para realização do experimento, o modelo de rede neural convolucional profunda foi executado 5 vezes para uma base de treinamento com 60.000 imagens de caracteres manuscritos e uma base de teste com 10.000 imagens de caracteres manuscritos.

Em seguida, o tempo de treinamento, o tempo de classificação e o resultado foram registrados na *Tabela 1*.

	Rede Neural Convolucional Profunda		
	Treinamento	**Classificacao**	**Resultado**
Teste 1	136.2262	4.3991	0.9857
Teste 2	135.2238	4.3792	0.9826
Teste 3	134.5444	4.3672	0.9813
Teste 4	135.2767	4.4176	0.9827
Teste 5	136.4254	4.3885	0.9841
Media =	135.5393	4.3903	0.9833

Tabela 1: *Resultados obtidos no treinamento de uma rede neural convolucional profunda L=2 e M=64*

Podemos observar na *Tabela 1*, que a rede neural convolucional profunda obtém excelente resultado na classificação dos dados, com uma taxa de acerto de 98,33%. Entretanto, o tempo de aprendizado deste modelo de rede neural é bastante lento, sendo que para 1 época de treinamento, foram necessários 136,4254 segundos de processamento.

4.2. RESULTADOS OBTIDOS COM REDES NEURAIS SEM PESO WISARD

Para realização do experimento, foi usado um modelo de rede neural WISARD com espaço de endereçamento de 3 bits, e este modelo foi executado 5 vezes para a mesma base de treinamento com 60.000 imagens de caracteres manuscritos e a mesma base de teste com 10.000 imagens de caracteres manuscritos.

Em seguida, o tempo de treinamento, o tempo de classificação e o resultado foram registrados na *Tabela 2*.

	WISARD		
	Treinamento	**Classificacao**	**Resultado**
Teste 1	2.1707	97.4596	70.8900
Teste 2	2.1686	102.8278	69.9500
Teste 3	2.1699	97.1135	70.8500
Teste 4	2.1772	98.4594	69.5900
Teste 5	2.1794	91.7673	72.1200
Media =	2.1732	97.5255	70.6800

Tabela 2: *Resultados obtidos no treinamento de uma rede neural WISARD com 3 bits de mapeamento*

Podemos observar na *Tabela 2*, que a rede neural WISARD obtém excelente resultado no tempo de treinamento dos dados. Obtendo um resultado médio de 2,1732 segundos para o treinamento da base. Este desempenho representa apenas 1,6% do tempo de aprendizado de 1 época da rede neural convolucional profunda analisada.

Entretanto, a taxa de acerto da rede neural WISARD com 3 bits de endereçamento, foi de apenas 70,68%, sendo este valor menor do que o registrado pela rede neural convolucional profunda analisada.

Também observamos neste experimento, que o tempo de classificação da base de teste com 10.000 imagens manuscritas usando a rede neural WISARD com 3 bits de endereçamento, foi pior que o tempo registrado pela rede neural convolucional profunda analisada.

Desta forma identificamos um possível problema no desempenho das redes WISARD na classificação dos dados, e consideramos inicialmente que este problema é resultado de uma fraca otimização da biblioteca Wisardpkg para a linguagem Python, e desta forma decidimos analisar mais profundamente este problema.

4.3. ANÁLISE DE DESMPENHO NA CLASSIFICAÇÃO DOS DADOS EM REDES NEURAIS WISARD

Na *Listagem 1*, observamos um trecho de código da biblioteca Wisardpkg. Este trecho de código se refere a etapa de classificação de uma entrada em uma rede neural WISARD que foi previamente treinada. Neste código existe um *looping* do tipo **do…while** que finaliza somente após todas as possíveis ambiguidades na classificação de uma imagem seja resolvida.

```
do{
  for(std::map<std::string,std::vector<int>>::iterator i=allvotes.begin(); i!=allvotes.end(); ++i){
    labels[i->first] = 0;
    for(unsigned int j=0; j<i->second.size(); j++){
      if(i->second[j] >= bleaching){
        labels[i->first]++;
      }
    }
  }
  if(!bleachingActivated) break;
  bleaching++;
  ambiguity = isThereAmbiguity(labels, confidence);
}while( std::get<0>(ambiguity) && std::get<1>(ambiguity) > 1 );
```

Listagem 1: *Trecho de código da biblioteca Wisardpkg que apresenta a resolução de conflitos na classificação.*

Para poder avaliar o quanto a fase de resolução de conflitos influencia no tempo de processamento na classificação dos dados de uma rede neural WISARD, foi incluído um novo parametro durante a compilação da biblioteca, __BLEACHING_LIMIT__, que limitou o número de iterações deste *looping (Listagem 2)*.

```
int n_exec = 0;
do{
  if(n_exec >= __BLEACHING_LIMIT__) break;
  n_exec++;

  for(std::map<std::string,std::vector<int>>::iterator i=allvotes.begin(); i!=allvotes.end(); ++i){
    labels[i->first] = 0;
    for(unsigned int j=0; j<i->second.size(); j++){
      if(i->second[j] >= bleaching){
        labels[i->first]++;
      }
    }
  }
  if(!bleachingActivated) break;
  bleaching++;
  ambiguity = isThereAmbiguity(labels, confidence);
}while( std::get<0>(ambiguity) && std::get<1>(ambiguity) > 1 );
```

Listagem 2: *Trecho de código da biblioteca Wisardpkg que foi modificado para incluir um limite no looping.*

Em seguida, realizamos testes com variações nos valores do parâmetro __BLEACHING_LIMIT__ entre 1 e 1000 iterações, obtendo os resultados apresentados na *Tabela 3*. Estes resultados mostram o aumento na acurácia e no tempo de processamento em função do aumento do limite de iterações para resolução de conflitos (*Figuras 5a e 5b)*.

WISARD – Bleaching Limit Analisys

Bleaching Limit	Train	Classify	Accuracy
1	2,2568	1,8623	18,15
2	2,2624	2,8518	20,40
3	2,3077	3,8155	21,93
4	2,2659	4,7681	23,79
5	2,2629	5,5273	25,55
10	2,3212	10,1477	32,57
15	2,2850	14,1412	36,72
20	2,2490	17,6235	39,52
30	2,2074	24,5481	44,53
40	2,2053	31,0703	47,83
50	2,2736	37,8108	50,23
100	2,2627	58,2265	58,36
200	2,2197	75,9489	64,84
300	2,6653	94,9329	65,51
400	2,2542	96,4372	68,66
500	2,3056	95,5665	68,99
515	2,2671	98,2168	68,03
531	2,2557	96,8540	71,28
562	2,2472	96,3647	71,27
625	2,3850	92,1897	70,31
750	2,3776	92,1897	72,32
1000	2,2342	98,7079	70,73

Tabela 3: *Resultados de uma rede neural WISARD com 3 bits e variações no limite do bleaching.*

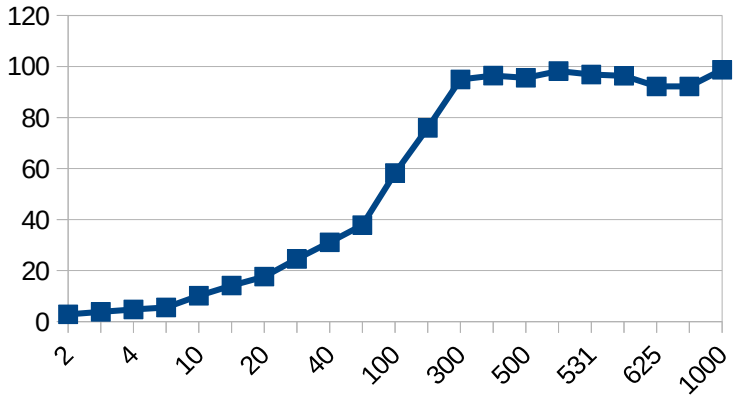

Figura 5a: *Aumento do tempo de classificação com a variação no limite do bleaching.*

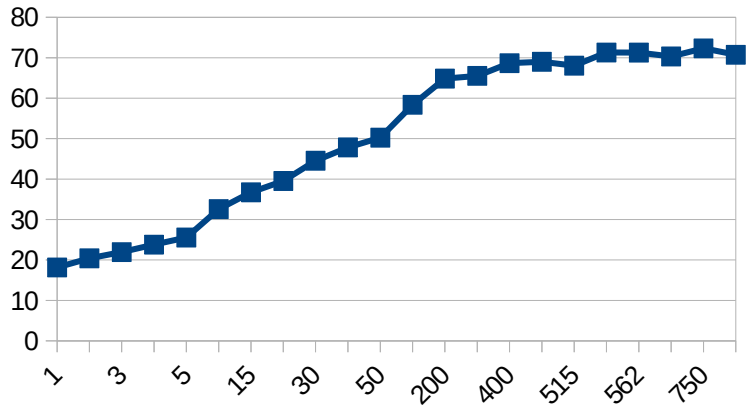

Figura 5b: *Aumento da acurácia com a variação no limite do bleaching.*

Nas *Figuras 5a e 5b*, podemos observar que para a rede neural WISARD com 3 bits de endereçamento que analisamos, o tempo de processamento se estabiliza após o limite de 400 iterações para resolução de conflitos, e que após este limite o ganho de acurácia é de cerca de 2,07% para 1000 iterações. Isto ocorre, porque a maioria dos conflitos são resolvidos em até 400 iterações.

Embora esta análise não tenha sido realizada com outros tamanhos de endereçamento da rede neural WISARD, podemos observar que para algumas aplicações pode ser interessante permitir uma pior acurácia na classificação dos dados em detrimento de uma maior velocidade nesta operação.

4.4. ANÁLISE DAS REDES NEURAIS SEM PESO COM VARIAÇÃO DO ENDEREÇAMENTO

Após a análise do problema de grande número de iterações na resolução de conflitos para classificação de dados das redes neurais WISARD, começamos a avaliar o resultado obtido em função do número de bits de endereçamento. A *Tabela 4a, 4b e 4c*, apresentam os resultados obtidos no treinamento, classificação e acurácia dos três modelos de redes neurais sem peso analisados.

WISARD

Bits	Train	Classify	Accuracy
3	2,2860	98,0546	69,80%
28	2,2624	1,1053	91,10%
32	2,3571	1,0580	92,05%
37	2,3373	1,0738	92,33%
64	2,2804	0,8080	83,06%

Tabela 4a: *Resultado das Redes Neurais WISARD com a variação no endereçamento.*

ClusWISARD – Supervised

Bits	Train	Classify	Accuracy
3	6,1491	273,9805	76,04%
28	3,9050	5,1869	91,34%
32	3,7024	4,4622	92,62%
37	3,4373	3,7153	91,92%
64	2,8086	2,0045	83,55%

Tabela 4b: *Resultado das Redes Neurais ClusWISARD Supervised com a variação no endereçamento.*

ClusWISARD – Unsupervised

Bits	Train	Classify	Accuracy
3	4,3210	46,6163	N.A.
28	3,4219	0,6954	N.A.
32	3,5135	0,6745	N.A.
37	3,2314	0,6247	N.A.
64	2,6572	0,4501	N.A.

Tabela 4c: *Resultado das Redes Neurais ClusWISARD Unsupervised com a variação no endereçamento.*

Neste experimento utilizamos uma rede neural ClusWISARD com até 5 discriminadores para cada classe, e com 3, 28, 32, 37 e 64 bits de endereçamento. Nas *Figuras 4a e 4b*, podemos verificar a redução no tempo de treinamento e de classificação de uma rede neural ClusWISARD, com o aumento do endereçamento.

Isto ocorre, porque com um maior número de discriminadores criados para cada classe, é necessário um maior número de iterações para treinar e classificar os dados, e com o aumento no endereçamento das redes neurais ClusWISARD ocorrem menos conflitos, reduzindo o tempo de treinamento e classificação.

Analisando as *Figuras 6a* e 6*b*, podemos observar que com o aumento do endereçamento da rede neural sem peso WISARD e ClusWISARD, ocorre uma aproximação no tempo de treinamento e de classificação, isto ocorre porque há menos similaridade entre os padrões identificados e os resultados são obtidos mais rapidamente.

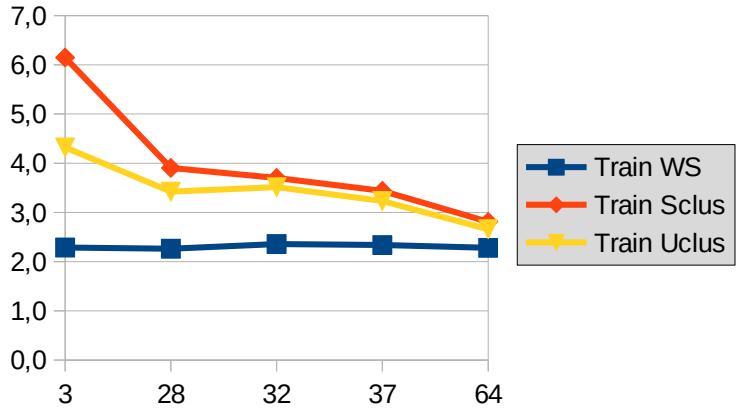

Figura 6a: *Redução no tempo de treinamento com o aumento do endereçamento.*

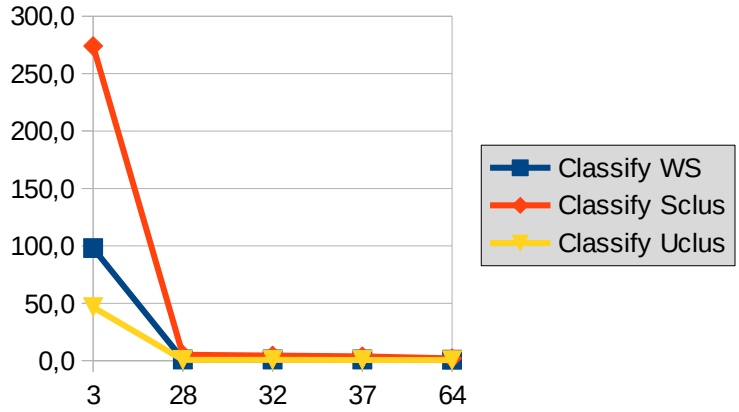

Figura 6b: *Redução no tempo de classificação com o aumento do endereçamento.*

Na *Figuras 6c*, podemos verificar um aumento na acurácia das redes neurais ClusWISARD, com o aumento do endereçamento. Analisando os gráfico da *Figura 6c*, observamos que com o aumento do endereçamento da rede neural sem peso WISARD e ClusWISARD, ocorre uma aproximação na acurácia, porque há menos similaridade entre os padrões identificados, o que aumenta a precisão do resultado.

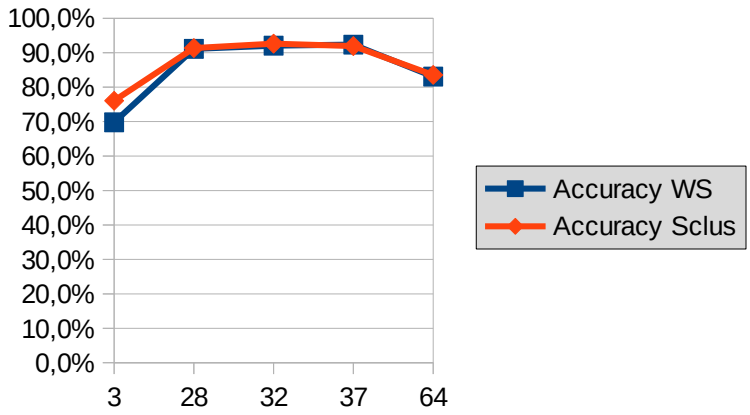

Figura 6c: *Aumento da acurácia com o aumento do endereçamento.*

4.5. RESULTADOS OBTIDOS COM REDES NEURAIS SEM PESO APÓS APLICAR DETECÇÃO DE BORDA

Após analisarmos o processamento das redes neurais WISARD e ClusWISARD com diferentes tamanhos de endereçamento, procuramos melhorar o desempenho destes modelos de redes neurais incluindo um pre-processamento das imagens da base NMINST com um filtro Sobel para detecção de borda. A *Figura 7*, apresentam um exemplo de imagem com este filtro aplicado.

Figura 7: *Resultado das redes neurais WISARD após detecção de borda.*

As Tabelas 5a e 5b, apresentam os resultados do processamento dos modelos de redes neurais sem peso WISARD e ClusWISARD após a aplicação do filtro de detecção de borda. Podemos observar que há uma melhora pequena no resultado, que chega no máximo há 1%, acredito que isto ocorreu porque as imagens da base NMINST são bastante simples e o pre-processamento dos dados não oferece muito ganho e ainda gera um tempo adicional para o processamento das imagens.

WISARD w/Border Detection

Bits	Train	Classify	Accuracy
3	2,2913	97,6225	70,01%
28	2,2825	1,0761	91,46%
32	2,3912	0,9966	91,48%
37	2,3199	0,9554	91,70%
64	2,2726	0,7551	84,49%

Tabela 5a: *Resultado das redes neurais WISARD após detecção de borda.*

Supervised ClusWISARD w/Boder Detection

Bits	Train	Classify	Accuracy
3	5,4328	280,6459	75,63%
28	3,9041	5,2500	92,07%
32	3,7626	4,9276	92,05%
37	3,5131	3,9587	92,60%
64	2,8333	2,0799	83,77%

Tabela 5b: *Resultado das redes neurais ClusWISARD Supervised após detecção de borda.*

Podemos observar novamente nas Figuras 8a e 8b, que o aumento do endereçamento da rede neural sem peso WISARD e ClusWISARD é mais importante do que o pré-processamento da imagem para a redução do tempo de treinamento e de classificação.

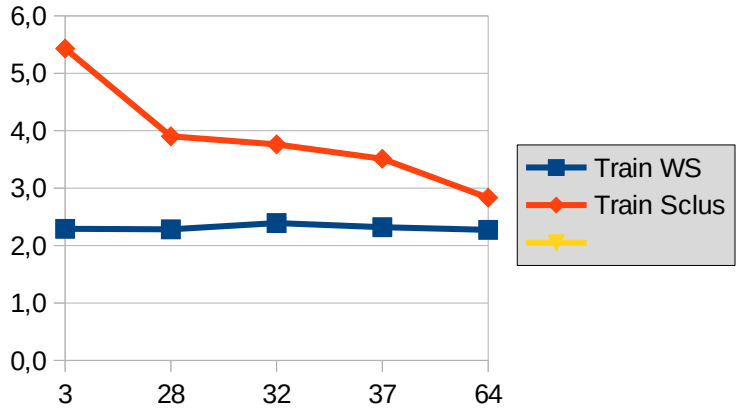

Figura 8a: *Redução no tempo de treinamento das redes neurais WISARD após detecção de borda.*

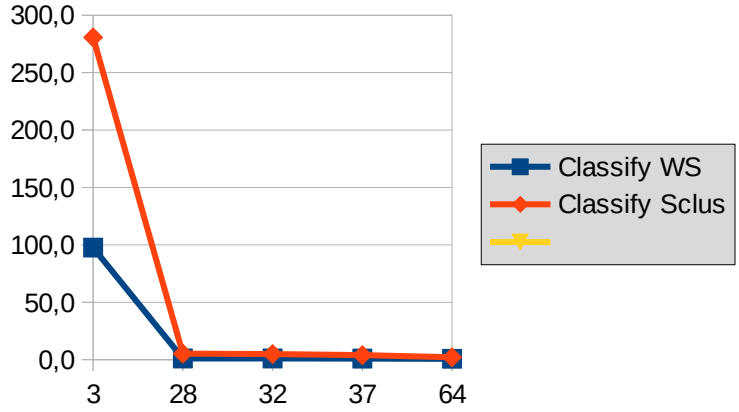

Figura 8b: *Redução no tempo de classificação das redes neurais WISARD após detecção de borda.*

Observamos na *Figuras 8c*, que é preferível aumentar a acurácia das redes neurais WISARD e ClusWISARD com o aumento do endereçamento do que com o pré-processamento da imagem.

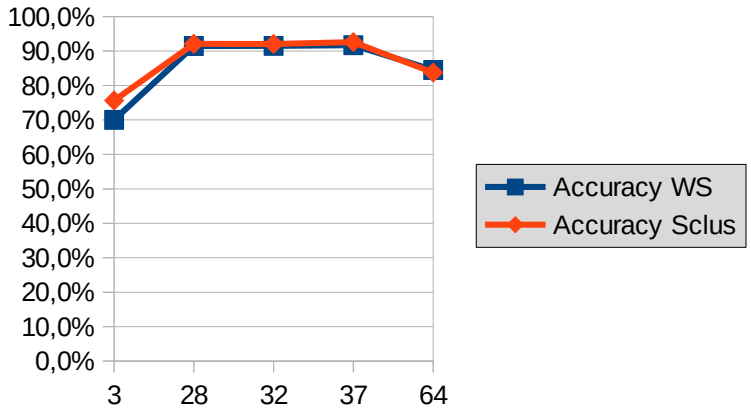

Figura 8c: *Aumento da acurácia das redes neurais WISARD após detecção de borda.*

5. CONCLUSÃO

Concluímos neste experimento que a aplicação da rede neural WISARD oferece uma grande vantagem no treinamento dos dados, com um tempo de processamento que é 1,6% do tempo registrado por uma rede convolucional profunda.

Podemos concluir também, que se efetuarmos uma otimização na implementação da biblioteca Wisardpkg para a linguagem Python, podemos também melhorar bastante o desempenho no treinamento e classificação dos dados, e isto pode ser um fator importante para uso destes modelos de redes neurais em ambientes que exijam maior rapidez no processamento mesmo que haja uma perda na acurácia da classificação dos dados.

Concluímos também neste experimento que o aumento no número de bits de endereçamento das redes neurais sem peso WISARD e ClusWISARD produzem uma grande melhora no tempo de treinamento no modelo ClusWISARD, e uma grande melhora no tempo de classificação e na acurácia de ambos os modelos.

Neste experimento o melhor resultado obtido com o modelo WISARD foi usando um endereçamento de 37 bits, onde o tempo de treinamento foi de 2,3373 segundos para um conjunto de treinamento com 60.000 imagens, e de classificação 1,0738 segundos, com uma acurácia de 92,33%, para uma base de teste com 10.000 imagens.

O melhor resultado obtido com o modelo ClusWISARD Supervised foi usando um endereçamento de 32 bits, onde o tempo de treinamento foi de 3,7024 segundos para um conjunto de treinamento com 60.000 imagens, e de classificação 4,4622 segundos, com uma acurácia de 92,62%, para uma base de teste com 10.000 imagens.

Podemos observar que com o aumento do endereçamento os dois modelos de redes neurais tendem a convergir em termos de acurácia, mas o tempo gasto no treinamento e classificação do modelo WISARD ainda é significativamente menor que o tempo gasto pelo modelo ClusWISARD quando analisamos as imagens da base NMINST.

ANEXO I – CÓDIGO REDE NEURAL CONVOLUCIONAL PROFUNDA (L=2 / M=64 / EPOCH=1)

```
import datetime
import csv
import numpy as np
import matplotlib.pyplot as plt
import keras
import keras.callbacks

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, Conv2D
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist
from keras.utils import to_categorical
from sklearn import datasets, svm
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from scipy.sparse import coo_matrix

class LossHistory(keras.callbacks.Callback):
	def on_train_begin(self, logs={}):
		self.losses = []
		self.acc = []
		self.list_X = []
		self.curr_X = 0
		self.list_Epoch = []
		self.list_Epoch.append(0.0)

	def on_batch_end(self, batch, logs={}):
		self.curr_X = self.curr_X + 1
		self.losses.append(logs.get('loss'))
		self.acc.append(logs.get('acc'))
		self.list_X.append(self.curr_X)

	def on_epoch_end(self, epoch, logs={}):
		self.list_Epoch.append(self.curr_X)

def calc_test(arr1, arr2):
	n = 0
	n_win = 0
	for v1 in arr1:
		v2 = arr2[n]
		n = n + 1
		if v1 == v2:
			n_win = n_win + 1
	result = (n_win / n * 100.0)
	return result

def plot_learning_curve(title, train_scores, train_sizes, train_epoch):
	plt.figure()
	plt.title(title)
	plt.xlabel("Training epoch")
	plt.ylabel("Score")
	plt.grid()
	plt.plot(train_sizes, train_scores)
	plt.legend(loc="best")
	n = 0
	for val_x in train_epoch:
		val_str = "Epoch " + str(n)
		plt.text(val_x, 0, val_str)
		n = n + 1
	return plt

startTime = datetime.datetime.now()
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255
X_test /= 255

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

n_train = len(X_train)
n_test = len(X_test)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Data Preparation - Elapsed Time")
print(elapsedTime)

model = Sequential()
model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

train_hist = LossHistory()

startTime = datetime.datetime.now()

hist = model.fit(X_train, y_train, batch_size=32, nb_epoch=1, verbose=1, callbacks=[train_hist])

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Train - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()

accuracy = model.evaluate(X_test, y_test)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Classify and Test - Elapsed Time")
print(elapsedTime)

print("Result")
print(accuracy)

plot_learning_curve(
        "L_2-M_64-Epoch_10-Losses",
        train_hist.losses,
        train_hist.list_X,
        train_hist.list_Epoch)

plot_learning_curve(
        "L_2-M_64-Epoch_10-Acc",
        train_hist.acc,
        train_hist.list_X,
        train_hist.list_Epoch)

plt.show()
```

ANEXO II – CODIGO REDE NEURAL WISARD (3 BITS)

```
def calc_test(arr1, arr2):
	n = 0
	n_win = 0
	for v1 in arr1:
		v2 = arr2[n]
		n = n + 1
		if v1 == v2:
			n_win = n_win + 1
	result = (n_win / n * 100.0)
	return result

def convert_to_list(arr):
	lst = []
	for arr_row in arr:
		lst_row = []
		for arr_col in arr_row:
			lst_row.append(arr_col)
		lst.append(lst_row)
	return lst

def convert_to_list2(arr):
	lst = []
	for arr_col in arr:
		lst.append(arr_col)
	return lst

def plot_learning_curve(title, train_scores, train_sizes, train_epoch):
	plt.figure()
	plt.title(title)
	plt.xlabel("Training epoch")
	plt.ylabel("Score")
	plt.grid()
	plt.plot(train_sizes, train_scores)
	plt.legend(loc="best")
	n = 0
	for val_x in train_epoch:
		val_str = "Epoch " + str(n)
		plt.text(val_x, 0, val_str)
		n = n + 1
	return plt

def show_mental_image(s):
	img = np.array(s, dtype='float')
	pixels = img.reshape((28, 28))
	plt.imshow(pixels, cmap='gray')
	plt.show()

import datetime
import wisardpkg as wp
import tensorflow as tf
import numpy as np
import math as m
import matplotlib.pyplot as plt

from array import *
from matplotlib import pyplot as plt
from synthesizer import Player, Synthesizer, Waveform

startTime = datetime.datetime.now()

(X, y), (X_t, y_t) = tf.keras.datasets.mnist.load_data()

X = X.reshape(60000,784)
X_t = X_t.reshape(10000,784)

X = X / 128
X_t = X_t / 128

X = X.astype('int')
y = y.astype('str')
```

```
X_t = X_t.astype('int')
y_t = y_t.astype('str')

X1 = []
y1 = []

X_t1 = []
y_t1 = []

X1 = convert_to_list(X)
y1 = convert_to_list2(y)

X_t1 = convert_to_list(X_t)
y_t1 = convert_to_list2(y_t)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Data Preparation - Elapsed Time")
print(elapsedTime)

show_mental_image(X[0])

wsd = wp.Wisard(3, ignoreZeros=False, verbose=False)

startTime = datetime.datetime.now()

wsd.train(X1, y1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Train - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()

out = wsd.classify(X_t1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Classify - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()

rst = calc_test(out, y_t1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Test - Elapsed Time")
print(elapsedTime)

print("Result")
print(rst)

mentalImages = wsd.getMentalImages()
s0 = mentalImages["0"]
show_mental_image(s0)
```

ANEXO III – CODIGO REDE NEURAL ClusWISARD SUPERVISED (3 BITS)

```
def calc_test(arr1, arr2):
	n = 0
	n_win = 0
	for v1 in arr1:
		v2 = arr2[n]
		n = n + 1
		if v1 == v2:
			n_win = n_win + 1
	result = (n_win / n * 100.0)
	return result

def convert_to_list(arr):
	lst = []
	for arr_row in arr:
		lst_row = []
		for arr_col in arr_row:
			lst_row.append(arr_col)
		lst.append(lst_row)
	return lst

def convert_to_list2(arr):
	lst = []
	for arr_col in arr:
		lst.append(arr_col)
	return lst

def plot_learning_curve(title, train_scores, train_sizes, train_epoch):
	plt.figure()
	plt.title(title)
	plt.xlabel("Training epoch")
	plt.ylabel("Score")
	plt.grid()
	plt.plot(train_sizes, train_scores)
	plt.legend(loc="best")
	n = 0
	for val_x in train_epoch:
		val_str = "Epoch " + str(n)
		plt.text(val_x, 0, val_str)
		n = n + 1
	return plt

def show_mental_image(s):
	img = np.array(s, dtype='float')
	pixels = img.reshape((28, 28))
	plt.imshow(pixels, cmap='gray')
	plt.show()

def show_cluster_mental_image(clus):
	for index, discriminator in enumerate(clus):
		#print(index, discriminator)
		show_mental_image(discriminator)

import datetime
import wisardpkg as wp
import tensorflow as tf
import numpy as np
import math as m
import matplotlib.pyplot as plt

from array import *
from matplotlib import pyplot as plt
from synthesizer import Player, Synthesizer, Waveform

startTime = datetime.datetime.now()

(X, y), (X_t, y_t) = tf.keras.datasets.mnist.load_data()

X = X.reshape(60000,784)
X_t = X_t.reshape(10000,784)

X = X / 128
```

```
X_t = X_t / 128

X = X.astype('int')
y = y.astype('str')

X_t = X_t.astype('int')
y_t = y_t.astype('str')

X1 = []
y1 = []

X_t1 = []
y_t1 = []

X1 = convert_to_list(X)
y1 = convert_to_list2(y)

X_t1 = convert_to_list(X_t)
y_t1 = convert_to_list2(y_t)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Data Preparation - Elapsed Time")
print(elapsedTime)

show_mental_image(X[0])
show_mental_image(X[1])
show_mental_image(X[2])
#show_mental_image(X[3])
#show_mental_image(X[4])
#show_mental_image(X[5])

addressSize = 3
minScore = 0.1
threshold = 10
discriminatorLimit = 5

wsd = wp.ClusWisard(
  addressSize,              # required
  minScore,                 # required
  threshold,                # required
  discriminatorLimit,       # required
  bleachingActivated=True,  # optional
  ignoreZero=False,         # optional
  completeAddressing=True,  # optional
  verbose=False,            # optional
  indexes=[],               # optional
  base=2,                   # optional

  ## types of return of classify
  returnActivationDegree=False, # optional
  returnConfidence=False,       # optional
  returnClassesDegrees=False    # optional
)

startTime = datetime.datetime.now()

wsd.train(X1, y1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Train - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()

out = wsd.classify(X_t1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
```

```
print("Classify - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()

rst = calc_test(out, y_t1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Test - Elapsed Time")
print(elapsedTime)

print("Result")
print(rst)

mentalImages = wsd.getMentalImages()
s0 = mentalImages["0"]

show_cluster_mental_image(s0)
```

ANEXO IV – CODIGO REDE NEURAL ClusWISARD UNSUPERVISED (3 BITS)

```
def calc_test(arr1, arr2):
	n = 0
	n_win = 0
	for v1 in arr1:
		v2 = arr2[n]
		n = n + 1
		if v1 == v2:
			n_win = n_win + 1
	result = (n_win / n * 100.0)
	return result

def convert_to_list(arr):
	lst = []
	for arr_row in arr:
		lst_row = []
		for arr_col in arr_row:
			lst_row.append(arr_col)
		lst.append(lst_row)
	return lst

def convert_to_list2(arr):
	lst = []
	for arr_col in arr:
		lst.append(arr_col)
	return lst

def plot_learning_curve(title, train_scores, train_sizes, train_epoch):
	plt.figure()
	plt.title(title)
	plt.xlabel("Training epoch")
	plt.ylabel("Score")
	plt.grid()
	plt.plot(train_sizes, train_scores)
	plt.legend(loc="best")
	n = 0
	for val_x in train_epoch:
		val_str = "Epoch " + str(n)
		plt.text(val_x, 0, val_str)
		n = n + 1
	return plt

def show_mental_image(s):
	img = np.array(s, dtype='float')
	pixels = img.reshape((28, 28))
	plt.imshow(pixels, cmap='gray')
	plt.show()

def show_cluster_mental_image(clus):
	for index, discriminator in enumerate(clus):
		print(index, discriminator)
		show_mental_image(discriminator)

import datetime
import wisardpkg as wp
import tensorflow as tf
import numpy as np
import math as m
import matplotlib.pyplot as plt

from array import *
from matplotlib import pyplot as plt
from synthesizer import Player, Synthesizer, Waveform

startTime = datetime.datetime.now()

(X, y), (X_t, y_t) = tf.keras.datasets.mnist.load_data()

X = X.reshape(60000,784)
X_t = X_t.reshape(10000,784)

X = X / 128
```

```
X_t = X_t / 128

X = X.astype('int')
y = y.astype('str')

X_t = X_t.astype('int')
y_t = y_t.astype('str')

X1 = []
y1 = []

X_t1 = []
y_t1 = []

X1 = convert_to_list(X)
y1 = convert_to_list2(y)

X_t1 = convert_to_list(X_t)
y_t1 = convert_to_list2(y_t)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Data Preparation - Elapsed Time")
print(elapsedTime)

show_mental_image(X[0])
show_mental_image(X[1])
show_mental_image(X[2])

addressSize = 3
minScore = 0.1
threshold = 10
discriminatorLimit = 5

wsd = wp.ClusWisard(
  addressSize,              # required
  minScore,                 # required
  threshold,                # required
  discriminatorLimit,       # required
  bleachingActivated=True,  # optional
  ignoreZero=False,         # optional
  completeAddressing=True,  # optional
  verbose=False,            # optional
  indexes=[],               # optional
  base=2,                   # optional

  ## types of return of classify
  returnActivationDegree=False, # optional
  returnConfidence=False,       # optional
  returnClassesDegrees=False    # optional
)

startTime = datetime.datetime.now()

wsd.trainUnsupervised(X1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Train - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()

out = wsd.classifyUnsupervised(X_t1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Classify - Elapsed Time")
print(elapsedTime)
```

ANEXO V – CODIGO REDE NEURAL WISARD C/ DETECÇÃO DE BORDA (3 BITS)

```
def calc_test(arr1, arr2):
	n = 0
	n_win = 0
	for v1 in arr1:
		v2 = arr2[n]
		n = n + 1
		if v1 == v2:
			n_win = n_win + 1
	result = (n_win / n * 100.0)
	return result

def convert_to_list(arr):
	lst = []
	for arr_row in arr:
		lst_row = []
		for arr_col in arr_row:
			lst_row.append(arr_col)
		lst.append(lst_row)
	return lst

def convert_to_list2(arr):
	lst = []
	for arr_col in arr:
		lst.append(arr_col)
	return lst

def plot_learning_curve(title, train_scores, train_sizes, train_epoch):
	plt.figure()
	plt.title(title)
	plt.xlabel("Training epoch")
	plt.ylabel("Score")
	plt.grid()
	plt.plot(train_sizes, train_scores)
	plt.legend(loc="best")
	n = 0
	for val_x in train_epoch:
		val_str = "Epoch " + str(n)
		plt.text(val_x, 0, val_str)
		n = n + 1
	return plt

def show_mental_image(s):
	img = np.array(s, dtype='float')
	pixels = img.reshape((28, 28))
	plt.imshow(pixels, cmap='gray')
	plt.show()

def detect_edge(arrImg):
	lst = []
	for img in arrImg:
		pixels = img.reshape((28,28))
		img_out = filters.sobel(pixels)
		img_out = img_out.reshape(784)
		lst.append(img_out)
	return lst

import datetime
import wisardpkg as wp
import tensorflow as tf
import numpy as np
import math as m
import matplotlib.pyplot as plt

from skimage import filters
from array import *
from matplotlib import pyplot as plt
from synthesizer import Player, Synthesizer, Waveform

startTime = datetime.datetime.now()

(X, y), (X_t, y_t) = tf.keras.datasets.mnist.load_data()
```

```
X = X.reshape(60000,784)
X_t = X_t.reshape(10000,784)

Xb = detect_edge(X)
Xb_t = detect_edge(X_t)

Xb = X / 128
Xb_t = X_t / 128

Xb = Xb.astype('int')
y = y.astype('str')

Xb_t = Xb_t.astype('int')
y_t = y_t.astype('str')

X1 = []
y1 = []

X_t1 = []
y_t1 = []

X1 = convert_to_list(Xb)
y1 = convert_to_list2(y)

X_t1 = convert_to_list(Xb_t)
y_t1 = convert_to_list2(y_t)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Data Preparation - Elapsed Time")
print(elapsedTime)

show_mental_image(X[0])
wsd = wp.Wisard(3, ignoreZeros=False, verbose=False)
startTime = datetime.datetime.now()

wsd.train(X1, y1)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Train - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()
out = wsd.classify(X_t1)
endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Classify - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()
rst = calc_test(out, y_t1)
endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Test - Elapsed Time")
print(elapsedTime)

print("Result")
print(rst)

mentalImages = wsd.getMentalImages()
s0 = mentalImages["0"]
show_mental_image(s0)
```

ANEXO VI – CODIGO REDE NEURAL ClusWISARD SUPERVISED C/ DETECÇÃO DE BORDA (3 BITS)

```
def calc_test(arr1, arr2):
    n = 0
    n_win = 0
    for v1 in arr1:
        v2 = arr2[n]
        n = n + 1
        if v1 == v2:
            n_win = n_win + 1
    result = (n_win / n * 100.0)
    return result

def convert_to_list(arr):
    lst = []
    for arr_row in arr:
        lst_row = []
        for arr_col in arr_row:
            lst_row.append(arr_col)
        lst.append(lst_row)
    return lst

def convert_to_list2(arr):
    lst = []
    for arr_col in arr:
        lst.append(arr_col)
    return lst

def plot_learning_curve(title, train_scores, train_sizes, train_epoch):
    plt.figure()
    plt.title(title)
    plt.xlabel("Training epoch")
    plt.ylabel("Score")
    plt.grid()
    plt.plot(train_sizes, train_scores)
    plt.legend(loc="best")
    n = 0
    for val_x in train_epoch:
        val_str = "Epoch " + str(n)
        plt.text(val_x, 0, val_str)
        n = n + 1
    return plt

def show_mental_image(s):
    img = np.array(s, dtype='float')
    pixels = img.reshape((28, 28))
    plt.imshow(pixels, cmap='gray')
    plt.show()

def show_cluster_mental_image(clus):
    for index, discriminator in enumerate(clus):
        print(index, discriminator)
        show_mental_image(discriminator)

def detect_edge(arrImg):
    lst = []
    for img in arrImg:
        pixels = img.reshape((28,28))
        img_out = filters.sobel(pixels)
```

```
            img_out = img_out.reshape(784)
            lst.append(img_out)
        return lst

import datetime
import wisardpkg as wp
import tensorflow as tf
import numpy as np
import math as m
import matplotlib.pyplot as plt

from skimage import filters
from array import *
from matplotlib import pyplot as plt
from synthesizer import Player, Synthesizer, Waveform

startTime = datetime.datetime.now()

(X, y), (X_t, y_t) = tf.keras.datasets.mnist.load_data()

X = X.reshape(60000,784)
X_t = X_t.reshape(10000,784)

Xb = detect_edge(X)
Xb_t = detect_edge(X_t)

Xb = X / 128
Xb_t = X_t / 128

Xb = Xb.astype('int')
y = y.astype('str')

Xb_t = Xb_t.astype('int')
y_t = y_t.astype('str')

X1 = []
y1 = []

X_t1 = []
y_t1 = []

X1 = convert_to_list(Xb)
y1 = convert_to_list2(y)

X_t1 = convert_to_list(Xb_t)
y_t1 = convert_to_list2(y_t)

endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Data Preparation - Elapsed Time")
print(elapsedTime)

show_mental_image(X[0])

addressSize = 3
minScore = 0.1
threshold = 10
discriminatorLimit = 5
```

```
wsd = wp.ClusWisard(
   addressSize,              # required
   minScore,                 # required
   threshold,                # required
   discriminatorLimit,       # required
   bleachingActivated=True,  # optional
   ignoreZero=False,         # optional
   completeAddressing=True,  # optional
   verbose=False,            # optional
   indexes=[],               # optional
   base=2,                   # optional

   ## types of return of classify
   returnActivationDegree=False, # optional
   returnConfidence=False,       # optional
   returnClassesDegrees=False    # optional
)

startTime = datetime.datetime.now()
wsd.train(X1, y1)
endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Train - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()
out = wsd.classify(X_t1)
endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Classify - Elapsed Time")
print(elapsedTime)

startTime = datetime.datetime.now()
rst = calc_test(out, y_t1)
endTime = datetime.datetime.now()

elapsedTime = endTime - startTime
print("Test - Elapsed Time")
print(elapsedTime)

print("Result")
print(rst)

mentalImages = wsd.getMentalImages()
s0 = mentalImages["0"]
show_cluster_mental_image(s0)
```

ANEXO VII – MODIFICAÇÃO APLICADA NO WISARDPKG (src/commom/bleaching.cc)

```
#define     __BLEACHING_LIMIT__     10

class Bleaching{
public:
  static std::map<std::string, int> make(std::map<std::string,std::vector<int>>& allvotes, const bool bleachingActivated, bool
searchBestConfidence=false, int confidence=1) {
    if(searchBestConfidence){
      return Bleaching::makeWithConfidence(allvotes, bleachingActivated, confidence);
    }
    else{
      return Bleaching::makeConfidenceless(allvotes, bleachingActivated, confidence);
    }
  }

  static std::map<std::string, int> makeWithConfidence(std::map<std::string,std::vector<int>>& allvotes, const bool bleachingActivated, const
int confidence) {
    return Bleaching::makeConfidenceless(allvotes, bleachingActivated, confidence);
  }

  static std::map<std::string, int> makeConfidenceless(std::map<std::string,std::vector<int>>& allvotes, const bool bleachingActivated, const
int confidence) {
    std::map<std::string, int> labels;
    int bleaching = 1;
    std::tuple<bool,int> ambiguity;

    int n_exec = 0;
    do{
      if(n_exec >= __BLEACHING_LIMIT__) break;
      n_exec++;

      for(std::map<std::string,std::vector<int>>::iterator i=allvotes.begin(); i!=allvotes.end(); ++i){
        labels[i->first] = 0;
        for(unsigned int j=0; j<i->second.size(); j++){
          if(i->second[j] >= bleaching){
            labels[i->first]++;
          }
        }
      }
      if(!bleachingActivated) break;
      bleaching++;
      ambiguity = isThereAmbiguity(labels, confidence);
    }while( std::get<0>(ambiguity) && std::get<1>(ambiguity) > 1 );

    return labels;
  }

  static std::string getBiggestCandidate(std::map<std::string,int>& candidates) {
    std::string label = "";
    int biggest = 0;
    for(std::map<std::string,int>::iterator i=candidates.begin(); i != candidates.end(); ++i){
      if(i->second >= biggest){
        biggest = i->second;
        label = i->first;
      }
    }
    return label;
  }

  static float getConfidence(std::map<std::string,int>& candidates, int biggest) {
    float secondBiggest = 0;
    for(std::map<std::string,int>::iterator i=candidates.begin(); i != candidates.end(); ++i){
      if(i->second >= secondBiggest && i->second < biggest){
        secondBiggest = i->second;
      }
    }
    return (biggest-secondBiggest)/biggest;
  }

private:
  static std::tuple<bool, int> isThereAmbiguity(std::map<std::string,int>& candidates, int confidence) {
    int biggest = 0;
```

```
    bool ambiguity = false;
    for(std::map<std::string,int>::iterator i=candidates.begin(); i != candidates.end(); ++i){
      if(i->second > biggest){
        biggest = i->second;
        ambiguity = false;
      }
      else if( (biggest - i->second) < confidence ){
        ambiguity = true;
      }
    }
    std::tuple<bool, int> ambiguityAndHighest = std::make_tuple(ambiguity, biggest);
    return ambiguityAndHighest;
  }
};

class BBleaching{
public:
  static std::map<std::string, int>& make(std::map<std::string,std::vector<int>>& allvotes, const bool bleachingActivated) {
    std::map<std::string, int>* labels = new std::map<std::string, int>;

    std::tuple<bool,int,int> ambiguity;
    int biggest = getBiggestValue(allvotes);
    int steps = 1;
    int piece = biggest/(int)std::pow(2,steps);
    int bleaching = piece;

    int n_exec = 0;
    do{
      if(n_exec >= __BLEACHING_LIMIT__) break;
      n_exec++;

      for(std::map<std::string,std::vector<int>>::iterator i=allvotes.begin(); i!=allvotes.end(); ++i){
        (*labels)[i->first] = 0;
        for(unsigned int j=0; j<i->second.size(); j++){
          if(i->second[j] >= bleaching){
            (*labels)[i->first]++;
          }
        }
      }
      if(!bleachingActivated) break;

      ambiguity = isThereAmbiguity(*labels);
      steps++;
      piece = biggest/(int)pow(2,steps);
      if(piece == 0) break;

      if(std::get<1>(ambiguity) > 0){
        bleaching = bleaching + piece;
      }
      else{
        bleaching = bleaching - piece;
      }
    }while( std::get<0>(ambiguity) && std::get<1>(ambiguity) > 0 );

    return *labels;
  }
private:
  static int getBiggestValue(std::map<std::string,std::vector<int>>& allvotes){
    int biggest = 0;
    for(std::map<std::string,std::vector<int>>::iterator i=allvotes.begin(); i!=allvotes.end(); ++i){
      for(unsigned int j=0; j<i->second.size(); j++){
        if(i->second[j]>biggest){
          biggest = i->second[j];
        }
      }
    }
    return biggest;
  }

  static std::tuple<bool, int, int> isThereAmbiguity(std::map<std::string,int>& candidates) {
    int biggest = 0;
    bool ambiguity = false;
    for(std::map<std::string,int>::iterator i=candidates.begin(); i != candidates.end(); ++i){
```

```
      if(i->second > biggest){
        biggest = i->second;
        ambiguity = false;
      }
      else if(i->second == biggest){
        ambiguity = true;
      }
    }
    std::tuple<bool, int, int> ambiguityAndHighest = std::make_tuple(ambiguity, biggest, 0);
    return ambiguityAndHighest;
  }
};
```