Universidade da Beira Interior
Departamento de Informática

Nº 341 - 2021: *Visualização do Processo de Treino de uma Rede Neuronal*

Elaborado por:

Francisco Rua

Orientador:

Professor Doutor João Neves

4 de julho de 2021

Agradecimentos

Ao meu orientador, Professor Doutor João Neves, pela oportunidade de poder realizar este trabalho e adquirir conhecimentos numa área que me era de todo desconhecida. Apesar dos percalços que presenciei nesta fase da minha vida académica, incetivou-me a não desistir e isso aumentou a minha moral e deu-me forças para que a conclusão deste projeto fosse possível.

Também agradeço o facto de me ter dado a conhecer uma área que me agradou imenso e que espero poder trabalhar no futuro.

Aos meus colegas e amigos, por toda a paciência que tiveram comigo e por toda a ajuda prestada. Falo em particular da Joana Marques, do André Salgado e do Guilherme Lopes. Sem eles, o meu percurso seria claramente diferente.

À minha família, que apesar de nunca entender bem o que estava propriamente a fazer, sempre fez esforços para o entender.

Conteúdo

Lista de Figuras

Lista de Excertos de Código

Acrónimos

API *Application Programming Interfaces*

CNN *Convolutional Neural Network*

CPU *Central Processing Unit*

GPU *Graphic Processing Unit*

IA Inteligência Artificial

IDE *Integrated Development Environment*

MNIST *Modified National Institute of Standards and Technology*

PCA *Principal Component Analysis*

SGD *Stochastic Gradient Descent*

T-SNE *t-Distributed Stochastic Neighbor Embedding*

Capítulo

1

Introdução

1.1 Enquadramento

O projeto desenvolvido envolve o estudo sobre redes neuronais com consequente implementação de uma aplicação que permite visualizar a forma como as mesmas organizam os dados após o processo de treino, explicando assim o impacto que diferentes estratégias de aprendizagem têm na organização dos elementos de conjunto de treino. A Linguagem *Python* aliada à plataforma *Tensorflow* foi essencial no desenvolvimento desta aplicação, bem como os conhecimentos adquiridos em Inteligência Artificial e Álgebra Linear, aprofundados nos cursos online da *Coursera*, *"Neural Networks and Deep Learning"* e *"Convolutional Neural Networks"*, adquirindo, também, conhecimentos em Visão Computacional.

O desenvolvimento deste projeto decorreu no âmbito da unidade curricular Projeto, Licenciatura em Engenharia Informática.

1.2 Motivação

As Redes Neuronais têm sido uma das maiores áreas de investigação no que diz respeito à Inteligência Artificial. É uma forma de fazer *Machine Learning*, no qual um agente aprende a realizar uma tarefa através da análise de um conjunto de treino e de um processo de aprendizagem, oferecendo suporte a várias vertentes, como a visão computacional, reconhecimento de voz, processamento de linguagem, vídeo-jogos, diagnósticos médicos, entre outras.

No entanto, mesmo com todo o sucesso alcançado pelas redes neuronais nos mais diversos tipos de problemas, é difícil compreender o processo de

decisão/previsão destes modelos, dificultando a capacidade de os investigadores proporem melhorias nas estratégias de aprendizagem.

Sendo assim, a ideia deste projeto é, através da implementação de diversas estratégias de aprendizagem, visualizar como os modelos organizaram os dados após o processo de treino, fornecendo assim informação relativa ao impacto que diversas técnicas têm no treino de uma rede neuronal para o *dataset* em questão, neste caso, o *Modified National Institute of Standards and Technology* (MNIST) *dataset* (base de dados de dígitos escritos manualmente).

1.3 Objetivos

O foco do projeto seria a implementação de um sistema que permitisse o treino de modelos de redes neuronais, segundo opções escolhidas pelo utilizador, e a visualização de como os dados do conjunto de treino seriam organizados após o processo de aprendizagem, percebendo assim o impacto que as diferentes escolhas teriam, recorrendo também a dados estatísticos.

Posto isto, para a conclusão do projeto ser possível, os objetivos propostos foram os seguintes:

1. Realização de um curso *online* sobre redes neuronais;
2. Aprender como treinar redes neuronais usando a plataforma *Tensorflow*;
3. Desenvolvimento de código para o treino customizado de uma rede neuronal usando a plataforma *Tensorflow*;
4. Desenvolvimento da aplicação para visualização do espaço topológico criado pela rede, em função de parâmetros de entrada dados pelo utilizador;
5. Testes e depuramento.

1.4 Etapas de Desenvolvimento

Em seguida, são apresentadas as etapas realizadas para o desenvolvimento do projeto.

1. Implementação de funções que permitem o carregamento do conjunto de treino e conjunto de validação e respetivo tratamento de dados;
2. Implementação de diversos modelos de redes neuronais;

3. Implementação de funções que permitem treinar o modelo escolhido pelo utilizador e a visualização de dados estatísticos produzidos após o processo de treino;

4. Implementação de funções que permitem a visualização do espaço topológico criado pelo processo de aprendizagem de uma rede neuronal escolhida pelo utilizador;

5. Implementação de um menu principal e de um menu secundário de forma que o utilizador possa interagir com o sistema e obter resultados;

6. Realização de testes e depuramentos.

1.5 Organização do Documento

A enumeração seguinte revela a forma como foi organizado este documento:

1. O primeiro capítulo – **Introdução** – Enquadramento do projeto, motivação para a sua proposta, objetivos e etapas de desenvolvimento.

2. O segundo capítulo – **Estado da Arte** – Apresentação de trabalhos prévios que se relacionam com a ideia a produzir.

3. O terceiro capítulo – **Especificação** – Visão teórica sobre a matéria em que se enquadra o trabalho.

4. O quarto capítulo – **Tecnologias e Ferramentas Utilizadas** – Apresentação dos recursos utilizados para a execução do projeto.

5. O quinto capítulo – **Implementação e Testes** – Descrição do processo de implementação, desde a implementação de elementos descritos na Especificação à implementação do menu que permite a interação do utilizador com o sistema, e demonstração visual do funcionamento do sistema.

6. O sexto capítulo – **Conclusões e Trabalho Futuro** – Conclusões principais e referência a um possível trabalho futuro.

Capítulo

2

Estado da Arte

2.1 Introdução

Neste capítulo serão revelados trabalhos que tenham um propósito semelhante a este projeto ou que estejam relacionados com o mesmo.

Através do trabalho de pesquisa efetuado, foi possível encontrar artigos que falam sobre a forma como mudanças nos modelos de treino revelam diferentes resultados e outros que analisam a visualização de modelos de treino, onde são explicadas formas de representação dos dados podendo assim relacionar tudo e implementar o sistema tal como era pretendido

2.2 Trabalhos Prévios

Sendo que o primeiro objetivo do trabalho seria o treino customizado de uma rede neuronal recorrendo à plataforma *Tensorflow* e, portanto, aplicar diversas estratégias de aprendizagem, o artigo [1] revela formas de otimizar a rede neuronal com o intuito de atingir a melhor performance possível, aplicando várias técnicas.

O artigo [2] e o tutorial [3] demonstram como alterações em parâmetros da rede neuronal podem levar a diferentes resultados no processo de aprendizagem, desde resultados não desejáveis a resultados de grande nível.

Os artigos [4], [5] e [6] revelam os métodos essenciais para a visualização do conjunto de dados e respetiva explicação, bem como formas de aplicar esses métodos a um modelo de aprendizagem ou até mesmo a uma camada do modelo em específico.

2.3 Conclusões

Analisando todos os artigos e o tutorial referenciados, é claro que nenhum demonstra/implementa na totalidade o que é pretendido para a elaboração deste projeto. Ainda assim, são trabalhos que estão relacionados com o que era pedido e, por conseguinte, foram um guia essencial para que a realização deste projeto fosse possível.

Capítulo

3

Especificação

3.1 Introdução

É neste capítulo que é feita uma visão geral sobre os conceitos teóricos que estão relacionados com o projeto, bem como uma explicação teórica sobre a área ao qual este trabalho se concentra.

3.2 Conceitos Gerais

São mencionados alguns conceitos gerais relacionados com o trabalho.

Inteligência Artificial -– ciência/engenharia de criar/desenvolver máquinas inteligentes, especialmente programas de computador inteligentes. Está relacionada com a utilização de computadores para entender a inteligência humana, no entanto, esta área não tem a necessidade de se limitar a métodos que são biologicamente observáveis.

Machine learning – ramo da Inteligência Artificial (IA) e da ciência da computação que, através do uso de dados e algoritmos, mede esforços para imitar a forma como um humano aprende, melhorando de forma gradual a sua precisão.

Redes Neuronais — conjunto de algoritmos que imitam as operações do cérebro humano de modo a reconhecer as relações entre grandes conjuntos de dados.

Deep Learning (em português: aprendizagem profunda) – parte integrante do *machine learning*. Na sua definição mais resumida, *deep learning* é essencialmente redes neuronais com 3 ou mais camadas.

De um modo geral, os conceitos supramencionados estão relacionados como se fossem Matrioscas (bonecas russas), ou seja, a ordem de conceitos seria inteligência artificial > *machine learning* > redes neuronais > *deep learning*.

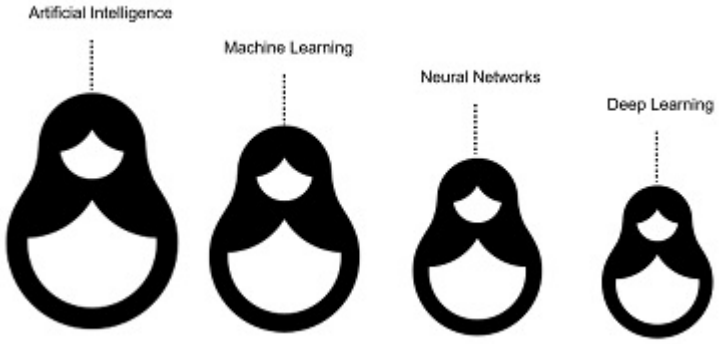

Figura 3.1: Matrioscas e relação com os conceitos gerais.

3.3 Redes Neuronais

Esta é uma área de grande interesse para os investigadores da ciência dos dados ou da IA, pois há uma grande esperança em compreender o comportamento do cérebro humano e a maneira como aprende, o que levou a grandes avanços na aprendizagem automática recentemente, combinando conjuntos de dados enormes e técnicas de *deep learning*.

Este último conceito (*deep learning* ou aprendizagem profunda) é caracterizado por certas técnicas de aprendizagem automática, onde unidades de processamento simples, os neurónios (nodos que contêm valores, importantes para o processo de aprendizagem), formam camadas e estas, por sua vez, estão interligadas, sendo que uma entrada, que é fornecida ao sistema, passará por todas, uma de cada vez.

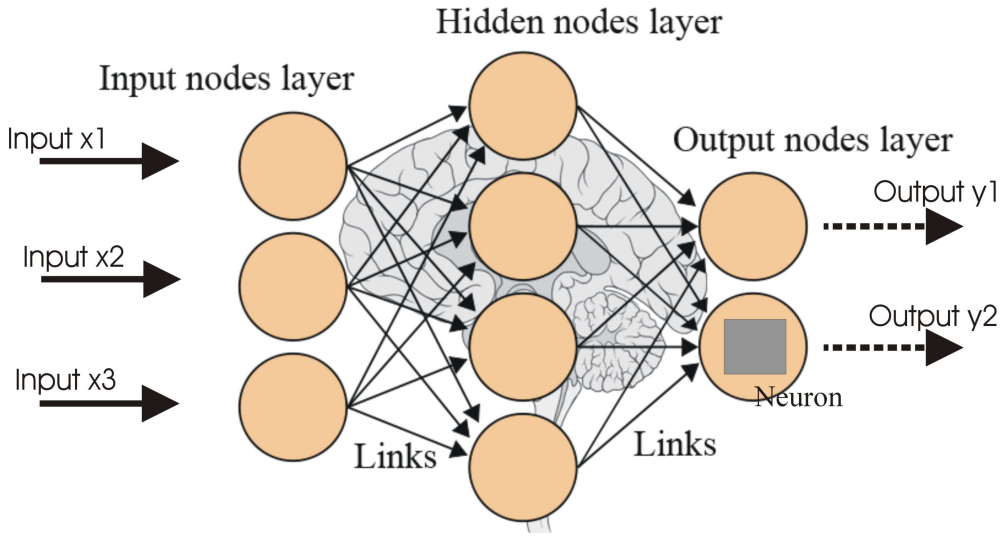

Figura 3.2: Rede Neuronal.

Cada neurónio consegue processar e armazenar uma informação diferente, ou seja, grandes quantidades de informação estão a ser processadas e armazenadas em simultâneo.

Este processamento em paralelo leva à importante utilização dos *Graphic Processing Unit* (GPU), que possuem esta valência.

A característica de profundidade que estas redes possuem permite-lhes aprender estruturas complexas através de um conjunto finito de dados e gerar respostas que interessam aos investigadores.

3.4 Comportamento de uma Rede Neuronal

O neurónio artificial ou nodo tem como base o comportamento do neurónio humano. Para um nodo ser "ativado", o estímulo é uma entrada de valores que precisam de ser processados. A partir daí, toda a rede será ativada, de forma sequencial, através de conexões entre as camadas constituintes, dando resposta/saída a esse estímulo/entrada, sendo que os valores de entrada vão sendo processados ao longo do "caminho".

Portanto, os neurónios decidem se há informação suficiente a ser enviada para os próximos através de cálculos matemáticos, verificando se o limite mínimo suficiente para os restantes neurónios serem ativados é atingido.

O aumento do número de camadas intermédias/ocultas permite que os computadores tenham mais capacidade, através de um processo de aprendizagem, simular comportamentos humanos com precisão.

Quando os dados são inseridos no modelo, a camada de entrada vai comunicar com as camadas ocultas, local onde acontece todo o processamento através das conexões que têm os pesos ajustados conforme o processo de treino efetuado previamente. Cada entrada é multiplicada pelo peso da conexão relativa a cada nodo. É feita a soma dos valores resultantes e, para cada nodo, a soma respetiva gera o potencial de ativação que servirá de parâmetro para uma função matemática de ativação não linear, limitando esse valor a um determinado intervalo (dependendo do resultado pretendido), determinando qual o caminho a seguir pela rede até afetar o resultado final. Por fim, as camadas ocultas ligam-se à camada de saída que permite a obtenção de resultados.

3.4.1 Processo de Aprendizagem

O processo de treino de uma rede neuronal ocorre segundo uma aprendizagem supervisionada, onde são fornecidas respostas corretas. É utilizado um conjunto de treino rotulado para este processo. Através de um vetor de entrada, correspondente a um elemento do conjunto de treino, é gerado um vetor de saída e consequente resposta. A resposta é comparada com a resposta que é pretendida, originando um erro. Esse erro vai ser utilizado para ajustar os pesos da rede. Através desse ajuste, a intenção é esse erro ser minimizado.

O ajuste é feito segundo o algoritmo de *Back-Propagation* [7], que utiliza a derivada parcial do erro em relação à camada de saída, seguida de derivada parcial do erro em função do potencial de ativação (da camada anterior), derivada parcial do erro em função dos pesos (da camada anterior) e assim sucessivamente até chegar à primeira camada da rede, de modo otimizar o processo de treino, sendo que, a função de custo (que calcula o erro) será reduzida ao máximo, gradualmente, método muito comum denominado de descida de gradiente.

Este processo é iterativo e acontece até serem atingidos valores de erro aceitáveis.

A redução do erro originará melhores respostas.

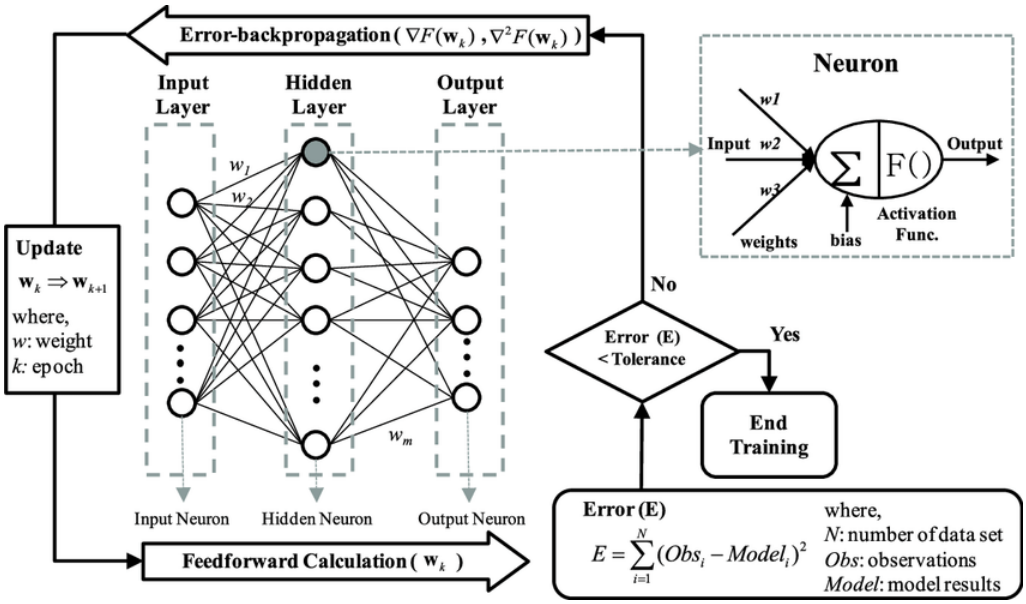

Figura 3.3: Back-Propagation.

3.5 Rede Neuronal Convolucional

Uma rede neuronal convolucional ou uma *Convolutional Neural Network* (CNN) é uma classe das redes neuronais, especializada no processamento de dados que possuem uma topologia semelhante a uma grelha, especialmente as imagens, sendo estas uma representação binária de dados visuais, dados esses a que chamamos de pixéis.

Ao visualizarmos uma imagem, uma enorme quantidade de informação é processada pelo ser humano. Cada campo recetivo é trabalhado por um neurónio que está conectado a outros, igualmente com a sua própria função, cobrindo assim todo o campo de visão.

Numa CNN o sistema é semelhante. Cada nodo processa os dados de acordo com a sua própria função. As primeiras camadas de nodos detetam os padrões mais simples (formas, linhas, etc) e ao longo da rede o processo vai evoluindo até ser possível detetar algo mais complexo, como rostos ou objetos.

3.5.1 Arquitetura de uma CNN

As CNN são tipicamente, no seu modelo mais simples, compostas por uma *convolution layer*, uma *pooling layer* e uma *fully connected layer*.

Convolution layer — é nesta camada que é processada a principal carga computacional da rede. É calculado um produto escalar entre a matriz que uma parte restrita do campo recetivo e a matriz que contém os parâmetros aprendíveis, tipicamente nomeda de *kernel*. O *kernel* é espacialmente (em altura e em largura) menor que uma imagem mas mais profundo, tendo em conta que uma imagem tem três canais visuais (RGB). Ao longo do processo, o *kernel* "deslizará" pela largura e pela altura, gerando a representação da imagem correspondente à região específica da imagem que está a ser tratada. Essa representação é bidimensional e é conhecida como mapa de ativação, que é a resposta que o kernel dá em cada posição espacial da imagem. O comprimento do "deslize" é chamado de passo ou *stride*.

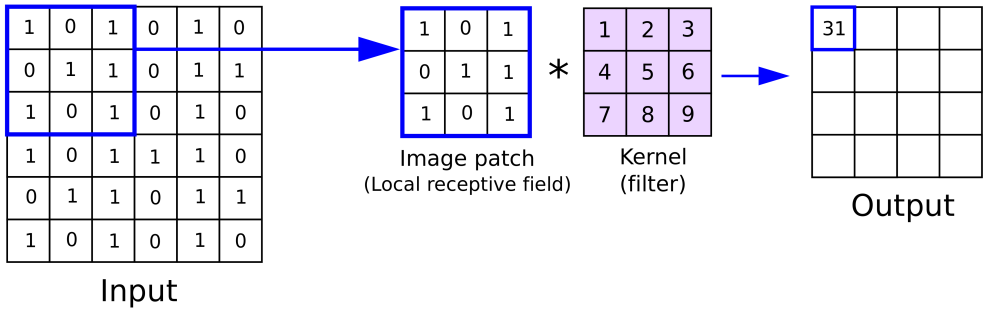

Figura 3.4: Convolution layer.

Pooling layer – nesta camada é reduzido o tamanho espacial da representação, aliviando a rede em termos computacionais. O método mais comum é o de *max pooling* em que em cada passo do *kernel* é escolhido o valor máximo, valor este que representará todos os outros na camada seguinte.

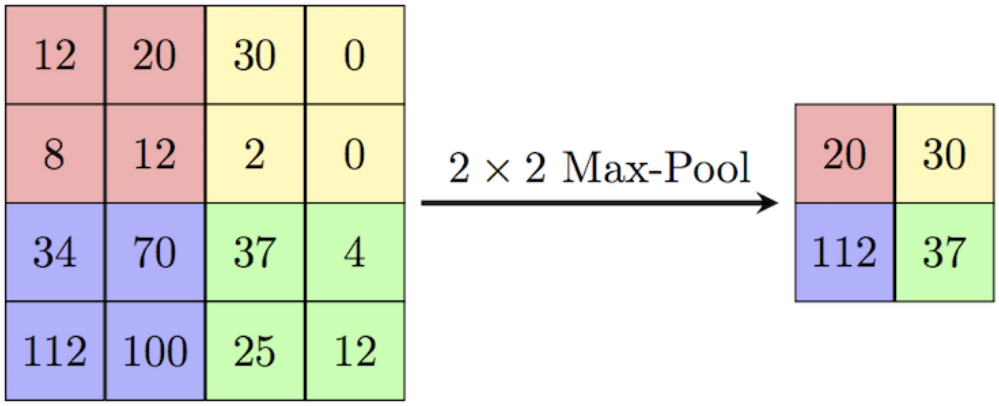

Figura 3.5: Max Pooling layer.

Fully Connected layer – nesta camada todos os neurónios têm conexão com os da camada anterior e posterior, mapeando a representação entre a entrada e a saída.

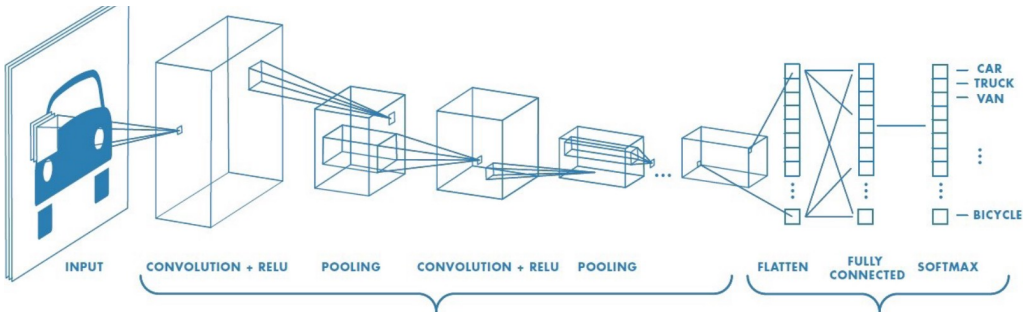

Figura 3.6: Convolutional Neural Network.

Como a convolução é uma operação linear e as imagens estão longe de ser lineares, é frequente colocar camadas não lineares logo após a camada convolucional ou *convolution layer* para introduzir a não linearidade no mapa de ativação.

Essas camadas contêm funções de ativação. As mais populares são:

- **Sigmoid** — função que transforma um valor para outro que esteja contido num intervalo entre 0 e 1.

- **Tanh** — função muito semelhante à anterior, no entanto, o intervalo é entre -1 e 1.

- **ReLU** (*Rectified Linear Unit*) – função que escolhe o valor máximo entre 0 e um limite passado como parâmetro. Se esse limite for negativo, o valor dessa função será 0 e o neurónio em questão não será ativado.

3.6 *t-Distributed Stochastic Neighbor Embedding* (T-SNE)

Este conceito é relativo a um algoritmo de redução de dimensionalidade não linear utilizado para analisar conjuntos de grandes dimensões e a forma como as redes os organizam, mapeando dados multidimensionais para duas ou mais dimensões adequadas para observação humana.

Portanto, redução de dimensionalidade é a técnica de representar dados que têm uma correlação entre si, em "*plots*" de 2 ou mais dimensões. A ideia é apresentar estes dados de forma que o observador perceba que elementos estão mais relacionados nesse conjunto através da distância entre eles.

O algoritmo *Principal Component Analysis* (PCA) está relacionado com o algoritmo T-SNE. É um algoritmo linear, ou seja, não tem a capacidade de interpretar relações polinomiais complexas entre os dados/características. Concentra-se em colocar pontos de dados que são diferentes bem separados, o que leva a um grande problema quando se trata de dados de grandes dimensões visto que este algoritmo não consegue que pontos de dados que são semelhantes sejam representados próximos (apenas os dados diferentes).

Sendo que é importante visualizar uma estrutura global dos dados, o algoritmo T-SNE tem um papel fundamental nesta matéria.

O algoritmo pode ser analisado no seguinte artigo [8].

De modo resumido, este algoritmo de redução de dimensionalidade não linear encontra padrões, baseando-se na similaridade, no conjunto de dados, dados estes que possuem múltiplas características. Organiza os dados segundo esses padrões e após esse processo é possível visualizar o mapeamento efetuado em que pontos são dados analisados e quanto mais próximos estão uns dos outros mais semelhantes serão esses dados.

3.7 Conclusões

Neste capítulo foi feita uma visão mais teórica sobre a matéria em que se enquadra o projeto desenvolvido centrando o foco nas redes neuronais convolucionais, transmitindo uma base de conhecimentos para o design de um modelo de aprendizagem deste tipo, e na função de redução de dimensiona-

lidade T-SNE, que permite obter uma visão de como é possível representar a forma como esses modelos organizam os dados.

Capítulo

4

Tecnologias e Ferramentas Utilizadas

4.1 Introdução

Neste capítulo é apresentado o *Integrated Development Environment* (IDE) *PyCharm*, a plataforma *Tensorflow* e outras ferramentas cruciais para que a realização do projeto fosse possível.

4.2 *PyCharm*

O *PyCharm* foi desenvolvido pela *JetBrains*, companhia que tem desenvolvido IDEs para diferentes linguagens de programação.

Sendo que neste trabalho a intenção é utilizar a Linguagem *Python*, o *PyCharm* é o IDE ideal. Possui deteção de erros, funcionalidade de completar código e correções de código automáticas, bem como um recurso de pesquisa inteligente que pode saltar para qualquer classe, ficheiro, função, etc.

Este IDE contém várias ferramentas úteis como o depurador integrado, o executador de teste e o terminal integrado. Oferece suporte ao *Anaconda*, permitindo trabalhar com um interpretador *tensorflow* e integra pacotes científicos essenciais para o projeto como o *Matplotlib*, *NumPy*, *Tensorflow* e *Sklearn*.

4.3 *Tensorflow*

Tensorflow é uma plataforma de código aberto dedicada ao *machine learning*. Possui várias ferramentas, bibliotecas e recursos, fornecendo máximo proveito aos investigadores/desenvolvedores da área.

Esta plataforma foi desenvolvida por quem, na altura, trabalhava na equipa da *Google Brain*, organizada para conduzir investigações em *machine learning* e *deep learning*.

O *TensorFlow* fornece *Application Programming Interfaces* (API) *Python* e *C++* estáveis, bem como uma API compatível com versões anteriores não garantidas para outras linguagens.

Com esta biblioteca não é necessária uma codificação complexa para preparar uma rede neuronal, visto que maior parte dos requisitos já estão implementados.

O processo de treino de uma rede neuronal exige muito poder computacional. O grande tamanho de dados a processar, os cálculos matemáticos necessários, os processos iterativos e as multiplicações de matrizes levam a que este processo seja demorado. Executando tudo isto com uma *Central Processing Unit* (CPU) normal, o processo ainda será mais demorado.

As GPUs são populares no contexto de vídeo-jogos, e foram criadas, precisamente, para esse propósito (alta resolução da tela e da imagem). No entanto, o uso destas para aplicações de *deep learning* é essencial.

Deste modo, o *Tensorflow* oferece suporte tanto para GPUs como para CPUs.

4.3.1 *Keras*

Keras é uma API, desenvolvida em *Python* para o *deep learning*, executada na plataforma *Tensorflow*. A criação desta API teve como foco permitir a experimentação rápida, permitindo ir da ideia ao resultado o mais rápido possível.

O *Keras* tem bastantes recursos que possibilitam trabalhar com redes neuronais, como camadas, funções de custo, funções de ativação, optimizadores e ainda suporte para redes neuronais convolucionais, importante para o trabalhado desenvolvido.

4.4 *Scikit-Learn, NumPy e Matplotlib*

A *Scikit-Learn* é uma biblioteca dedicada ao *machine learning* que contém diversos algoritmos, um dos quais importante para a implementação deste trabalho que é o algoritmo de redução de dimensionalidade T-SNE.

NumPy é mais uma das bibliotecas de computação científica do *Python* que permite a manipulação de vetores e matrizes multidimensionais, contendo múltiplas funções matemáticas que permitem trabalhar com estas estruturas.

A *Matplotlib* é uma extensão da *NumPy*, permitindo a criação de gráficos e visualizações de dados. Através do módulo *Pyplot*, esta biblioteca permite trabalhar em *Python* como se de *MATLAB* tratasse, ferramenta essencial para o propósito do Projeto.

4.5 Conclusões

As secções anteriores revelam as tecnologias e ferramentas essenciais para a implementação do trabalho proposto. O facto de a Linguagem *Python* ser moderna e muito globalizada, permite a existência de inúmeras e vantajosas ferramentas de trabalho, simplificando a vida dos desenvolvedores e reduzindo o processo cognitivo. A plataforma *Tensorflow* aliada à extensão *Keras* e ainda as bibliotecas *Scikit-Learn, NumPy e Matplotlib* foram cruciais para toda a implementação do trabalho no IDE dedicado ao *Python*, o *PyCharm*.

Capítulo

5

Implementação e Testes

5.1 Introdução

O capítulo atual revela o procedimento de implementação do trabalho. Aqui serão descritos os modelos construídos e respetivo processo de aprendizagem, bem como a forma como foi possível a visualização dos dados e estatísticas após o processo de treino. Finalmente, é revelada uma breve demonstração do funcionamento do sistema.

5.2 MNIST *dataset*

De maneira a cumprir o propósito do projeto, o MNIST *dataset*, também conhecido como o "*Hello World*" da Visão Computacional, parece ser o conjunto de dados que melhor se adequa. Trata-se de um conjunto de 60000 imagens de dígitos escritos à mão. É fácil identificar características comuns entre dígitos, pelo que qualquer utilizador poderá analisar o porquê de as redes reunirem os dados seja da forma que for.

Para o uso desta base de dados no sistema, foi necessário a implementação de funções que permitem descarregar o MNIST e tratar as respetivas imagens de forma a poderem ser usadas para treino das redes.

```
def load_dataset():
    (trainX, trainY), (testX, testY) = mnist.load_data()
    # redefinir canal de forma a ter um canal  nico
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY
```

```
def prep_dados(train, test):
    # converter inteiros para floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalizar para um intervalo entre 0 e 1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0

    return train_norm, test_norm
```

Excerto de Código 5.1: Tratamento do conjunto de dados.

5.3 Modelos de Aprendizagem

Para ser possível a visualização de como as redes organizam os dados após o processo de aprendizagem, convém ter diferentes modelos de treino de modo a existir termo de comparação.

Sendo assim, foram implementados 6 modelos de CNN, uns mais eficientes que outros.

A ideia foi começar por um modelo mais básico e ir implementando algumas técnicas de forma a verificar as mudanças que podem ocorrer. A verdade é que uma simples mudança pode mudar o comportamento da rede.

Portanto o modelo inicial é composto por uma única camada convolucional com 32 filtros 3x3, seguida de uma camada *Max Pooling*. O facto de haver 10 dígitos possíveis de identificar, requer uma camada de saída composta por 10 nodos de modo a prever a distribuição de probabilidade de uma imagem pertencente a cada uma das 10 classes (dígitos). Esse facto também exigirá o uso de uma função de ativação final ***softmax***, da família ***sigmoid*** mas em vez de retornar valores no intervalo [0, 1], retorna valores entre [0, número máximo de classes – 1].

Antes da camada de saída foi introduzida uma camada que extrai as características finais (`Flatten()`) e uma camada densa como 100 nodos (`Dense()`) que interpreta as caraterísticas.

A função de ativação ***ReLU*** foi usada em todas as camadas bem como o esquema de inicialização de pesos '`He uniform`', práticas ambas recomendadas por Jason Brownlee, especialista em *machine learning*.

A configuração utilizada para o otimizador foi a *Stochastic Gradient Descent* (SGD) *optimizer* com *learning rate* de 0.001 e *momentum* de 0.9, valor conservador. A função de custo foi a '`categorical_crossentropy`', muito adequada para problemas que envolvem várias classes, e monitoraremos a

métrica de precisão da classificação, apropriada quando se tem o mesmo número de exemplos para cada classe.

```
def basemodel(learningrate):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='
        he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(Dense(10, activation='softmax'))

    o = SGD(lr=learningrate, momentum=0.9)
    model.compile(optimizer=o, loss='categorical_crossentropy', metrics
        =['accuracy'])
    return model
```

Excerto de Código 5.2: Modelo base.

Com esse modelo, o utilizador pode variar o learning rate do otimizador bem como o número de epochs. De seguida, foi feita uma pequena alteração na rede, alterando o otimizador para *Adam*, podendo variar igualmente o *learning rate*:

```
o = Adam(lr=learningrate)
```

Este é um otimizador moderno, muito utilizado hoje em dia pelos investigadores, sendo que estes referem o *Adam* como substituto do SGD.

O próximo passo foi tornar o modelo mais "profundo". Foi criado então outra rede em que se adicionou mais uma camada convolucional com 32 filtros, duas camadas convolucionais com 64 filtros seguidas de mais uma camada *Max Pooling* bem como a troca da camada densa de 100 nodos por três camadas densas de 256, 128 e 84 nodos respetivamente. Um modelo mais profundo e mais denso permite que características mais complexas dos dígitos manuscritos sejam aprendidas.

```
def deepmodel():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='
        he_uniform', input_shape=(28, 28, 1)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='
        he_uniform'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='
        he_uniform'))
```

```
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='
        he_uniform'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(256, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(Dense(128, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(Dense(84, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(Dense(10, activation='softmax'))

    o = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=o, loss='categorical_crossentropy', metrics
        =['accuracy'])
    return model
```

Excerto de Código 5.3: Modelo mais profundo e denso.

Outra técnica de aprendizagem aplicada foi a introdução de uma camada `BatchNormalization()` após a primeira camada convolucional e após a camada densa de 100 nodos (considerando o Modelo Base).

```
model.add(BatchNormalization())
```

O propósito foi estabilizar a rede e fazer o algoritmo de aprendizagem convergir mais rápido e permitir que as camadas aprendam mais por si mesmas de forma independente, visto que normalizar as ativações reduz o movimento entre as camadas ocultas.

De forma a reduzir a variância do modelo significativamente, procedeu-se à aplicação da técnica de *Data Augmentation*, procedendo à rotação aleatória e zoom aleatório de imagens do conjunto de treino. Esta é adicionada antes da primeira camada convolucional.

Em paralelo, a implementação da técnica de *dropout* a seguir à camada de *pooling* e à camada densa de 100 nodos, que desativa alguns neurónios (neste caso 25%), permitindo que a rede generalize melhor não dependendo absolutamente de nenhum nodo em particular para produzir uma saída, contribuiu ainda mais para a regularização do modelo de aprendizagem, impedindo, em grande parte, o sobre-ajuste, fenómeno que acontece quando os modelos aprendem com ruídos ou detalhes indesejados de exemplos de treino, prejudicando negativamente o desempenho do modelo em novos exemplos.

```
data_augmentation = keras.Sequential(
  [
    layers.experimental.preprocessing.RandomFlip("horizontal",
        input_shape=(28, 28, 1)),
    layers.experimental.preprocessing.RandomRotation(0.1),
    layers.experimental.preprocessing.RandomZoom(0.1),
  ]
)


def model_Drop_and_DataAug():
    model = Sequential()
    model.add(data_augmentation)
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='
        he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(Dropout(0.25))
    model.add(Dense(10, activation='softmax'))

    o = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=o, loss='categorical_crossentropy', metrics
        =['accuracy'])
    return model
```

Excerto de Código 5.4: *Data Augmentation e Dropout.*

Por último, surgiu a ideia de criar um modelo que implemente todas as alterações supracitadas.

```
def bestmodel():
    model = Sequential()
    model.add(data_augmentation)
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='
        he_uniform', input_shape=(28, 28, 1)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='
        he_uniform'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='
        he_uniform'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='
        he_uniform'))
    model.add(BatchNormalization())
```

```
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(256, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(BatchNormalization())
    model.add(Dense(128, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(BatchNormalization())
    model.add(Dense(84, activation='relu', kernel_initializer='
        he_uniform'))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))
    model.add(Dense(10, activation='softmax'))

    o = Adam(lr=0.001)
    model.compile(optimizer=o, loss='categorical_crossentropy', metrics
        =['accuracy'])
    return model
```

Excerto de Código 5.5: Modelo com todas as alterações.

5.4 Processo de treino e apresentação de dados estatísticos e visuais

Tendo os modelos previamente implementados, o passo seguinte será o treino destes recorrendo às ferramentas disponibilizadas pela *Tensorflow*, bem como a respetiva análise de dados, de forma estatística e visual, através da biblioteca *Matplotlib* e da *Sklearn* respetivamente, onde são importados o *pyplot* e a *T-SNE function*. Todas estas ações ocorrem num menu principal implementado de forma que o utilizador possa interagir com o sistema.

Finalmente, um menu secundário é apresentado, onde o utilizador pode testar o modelo escolhido com imagens respetivas a cada dígito, sendo feito uma previsão de qual será o dígito introduzido como *input*.

A seguir é apresentado um exemplo das imagens geradas após o processo de aprendizagem do Modelo Base:

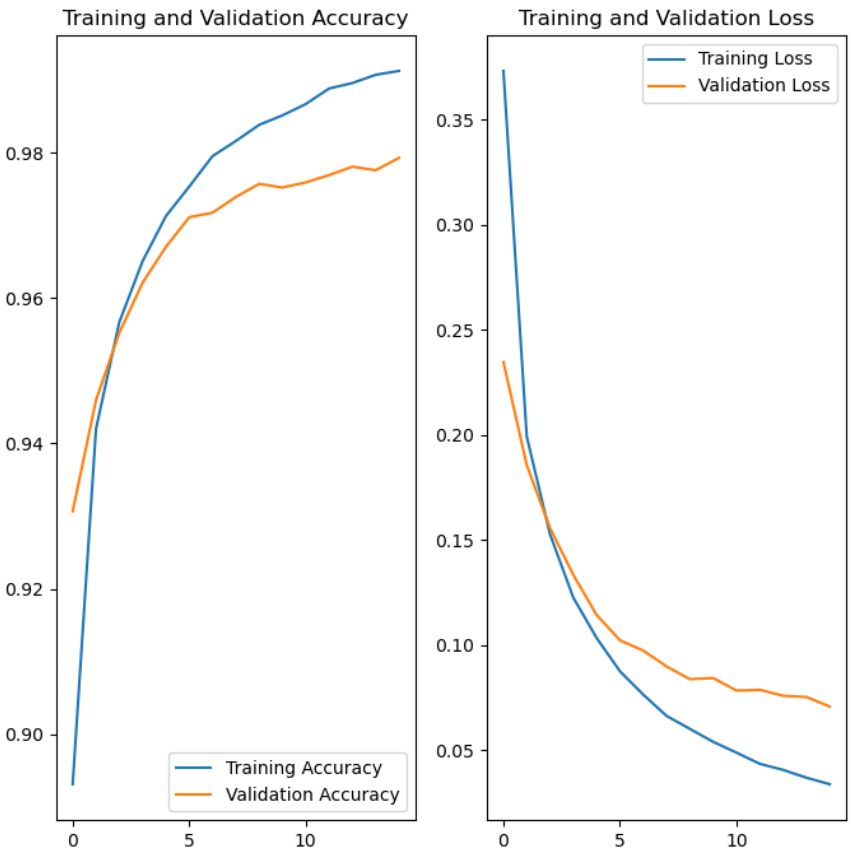

Figura 5.1: Gráficos gerados pelo Modelo Base.

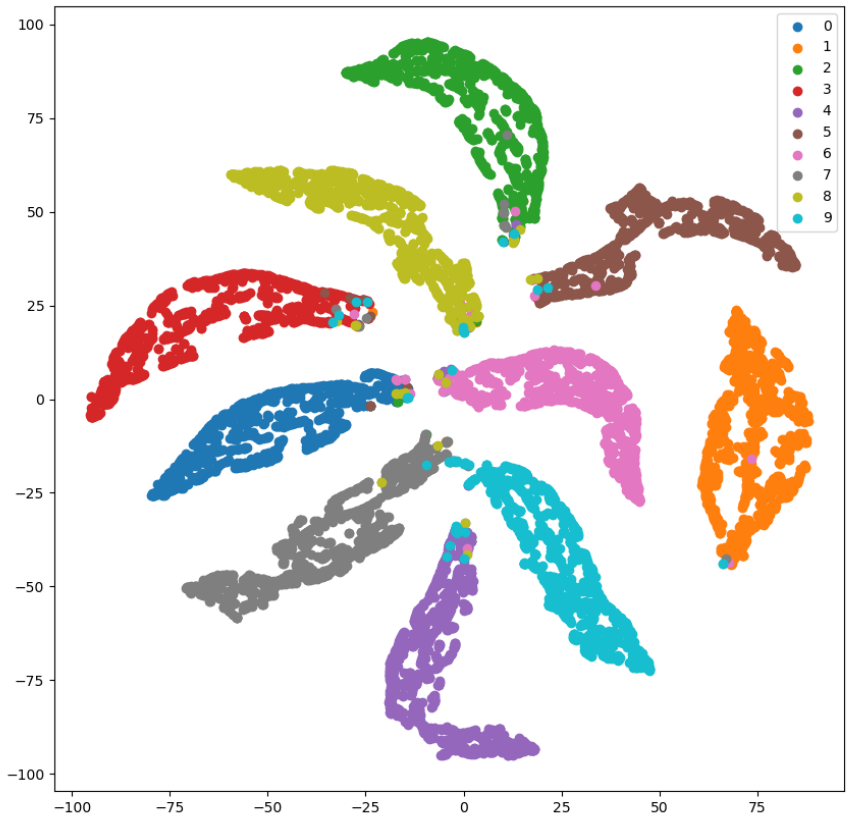

Figura 5.2: Agrupamento dos dados do conjunto de treino (Gerados pelo T-SNE).

5.5 Conclusões

Neste capítulo foram apresentadas as etapas de implementação do projeto e respetiva demonstração de como o sistema funciona. O principal foco foi demonstrar os modelos criados que vão provar que alterações num modelo base implica alterações na forma como a rede organizará os dados de treino.

Capítulo

6

Conclusões e Trabalho Futuro

6.1 Conclusões Principais

O objetivo principal deste trabalho seria perceber o impacto que diferentes estratégias de aprendizagem têm na organização dos elementos de conjunto de treino, em particular os mais difíceis de discernir pela rede. Com o recurso à função de redução de dimensionalidade T-SNE, o objetivo foi cumprido. Através dos diferentes modelos criados e treinados devidamente, seguida de aplicação da função supramencionada, foi possível a Visualização do Espaço Topológico das redes neuronais convolucionais.

Interagindo com o sistema, é fácil perceber que pequenas mudanças podem afetar a forma como uma rede neuronal aprende e organiza os dados.

Visualizando os dados que são apresentados, o utilizador poderá ter uma noção de qual seria a melhor forma de construir um modelo de aprendizagem para o *dataset* em questão.

6.2 Trabalho Futuro

O sistema foi implementado de forma que qualquer utilizador possa visualizar, de uma forma simples, a existência de diferenças através das opções escolhidas.

Numa versão mais avançada, seria interessante que o utilizador pudesse construir o seu próprio modelo através de uma interface gráfica, escolhendo todos os parâmetros e Hiper parâmetros livremente, e consequente visualização de resultados.

Consoante os resultados, o sistema fornecer sugestões para melhoria do modelo de aprendizagem, dando liberdade completa ao utilizador de tomar

as suas escolhas.

Assim, seria uma aplicação bastante mais intuitiva e poderia dar hipótese ao utilizador de otimizar ao máximo uma rede neuronal, sendo este o próprio autor da mesma.

No entanto, esta ideia de implementação levaria a que a máquina onde seria instalado o sistema tivesse um grande poder computacional, visto que o processo de treino de um modelo de aprendizagem exige bastante das GPUs.

Bibliografia

[1] Jay Gupta. Going beyond 99% — MNIST Handwritten Digits Recognition, 2020. [Online] https://towardsdatascience.com/going-beyond-99-mnist-handwritten-digits-recognition-cfff96337392.

[2] Vadim Smolyakov. Neural Network Optimization Algorithms, 2018. [Online] https://towardsdatascience.com/neural-network-optimization-algorithms-1a44c282f61d.

[3] Jason Brownlee. Understand the Impact of Learning Rate on Neural Network Performance, 2019. [Online] https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks

[4] Andre Violante. An Introduction to t-SNE with Python Example, 2018. [Online] https://medium.com/@violante.andre/an-introduction-to-t-sne-with-python-example-47e6ae7dc58f.

[5] Luuk Derksen. Visualising high-dimensional datasets using PCA and t-SNE in Python, 2016. [Online] https://towardsdatascience.com/visualising-high-dimensional-datasets-using-pca-and-t-sne-in-python-8ef87

[6] Siddhartha Banerjee. Visualizing Layer Representations in Neural Networks, 2017. [Online] https://becominghuman.ai/visualizing-representations-bd9b62447e38.

[7] David E. Rumelhart and James L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1987.

[8] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE, 2008. [Online] https://lvdmaaten.github.io/publications/papers/JMLR_2008.pdf.