IA048 - Aprendizado de Máquina

Atividade 3 - Redes Neurais

Turma A - Primeiro semestre de 2024
Aluno: Ronald Gabriel Ferreira da Silva
RA: 195815

Introdução:

Este relatório aborda a aplicação prática das redes neurais em um contexto de grande interesse: a classificação de tipos sanguíneos. Para isto analisamos utilizando a base de dados BloodMNIST. Esta base de dados contém 17.092 imagens microscópicas coloridas de células sanguíneas, distribuídas em diversas classes, cada uma representando um tipo específico de célula. As imagens têm uma resolução de 28 × 28 pixels e são compostas por três canais de cor, facilitando a utilização de técnicas de processamento de imagem e aprendizado profundo.

Para realizar o estudo e tratamento dos dados, utilizaremos as seguintes bibliotecas do Python:

1) **numpy (np**): O NumPy é utilizado para operações numéricas eficientes em arrays e matrizes, sendo útil para cálculos estatísticos e manipulações de dados.

2) **matplotlib.pyplot (plt)**: Essa biblioteca é amplamente utilizada para a visualização de dados, permitindo criar gráficos e plots para análise exploratória dos dados.

3) **TensorFlow + Keras**: TensorFlow é uma biblioteca de código aberto para aprendizado de máquina, e Keras é uma API de alto nível que roda sobre TensorFlow. Juntas, essas bibliotecas são usadas para construir, treinar e avaliar modelos de aprendizado profundo. Especificamente:

- **Sequential** é usado para criar modelos lineares camada por camada;
- **Dense** e **Flatten** são camadas básicas para redes neurais densas;
- **to_categorical** é uma função de utilidade que converte rótulos inteiros em um formato de codificação one-hot, necessário para problemas de classificação multiclasse.

4) **Scikit-learn:** é uma biblioteca poderosa para aprendizado de máquina em Python que, neste caso, serão usadas para avaliar a performance do modelo:

- **Confusion_matrix** para calcular a matriz de confusão, que mostra a contagem de previsões corretas e incorretas divididas por classe;

- **Accuracy_score** para calcular a acurácia global do modelo, que é a proporção de previsões corretas.

5) **Seaborn:** Seaborn, assim como Matplotlib é usado para criar visualizações estatísticas atraentes e informativas, como mapas de calor (heatmaps).

6) **BloodMnist e INFO**: **BloodMNIST** e **INFO** são usados para carregar e obter informações sobre o conjunto de dados BloodMNIST, respectivamente.

7) Torchvision + Transforms: é usado para transformar e pré-processar as imagens antes de passá-las para o modelo.

Essas e outras bibliotecas que serão citadas durante o texto fornecem uma base sólida para realizar análises de dados, modelagem estatística e avaliação de modelos, o que nos permitirá explorar e compreender melhor a série temporal do tráfego aéreo.

Estudo dos dados com uma rede MLP com uma camada intermediária:

Primeiramente foi feita a preparação dos dados iniciando as bibliotecas e determinando o conjunto de dados para teste e para treino.

```
# Definir transformações
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean=[.5], std=[.5])])

# Carregar dados BloodMNIST
train_dataset = BloodMNIST(split='train', transform=transform, download=True)
test_dataset = BloodMNIST(split='test', transform=transform, download=True)

# Extrair dados e rótulos
x_train, y_train = train_dataset.imgs, train_dataset.labels
x_test, y_test = test_dataset.imgs, test_dataset.labels

# Normalizar os dados
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convertendo os rótulos para one-hot encoding
y_train = to_categorical(y_train, num_classes=8)
y_test = to_categorical(y_test, num_classes=8)
```

Após a preparação dos dados, definimos a arquitetura a ser utilizada neste primeiro momento. Para isso definimos uma arquitetura simples composta por uma camada Flatten e duas camadas densas (Fully Connected e Saída). A camada Flatten foi útil para transformar a entrada de um formato multidimensional (28x28x3, que é a dimensão de uma imagem colorida de 28x28 pixels com 3 canais de cor) em um vetor unidimensional. Essa transformação é necessária para converter a estrutura da imagem em um formato que possa ser processado por camadas densas.

A sua grande vantagem é que ela simplifica a entrada da imagem para ser utilizada em camadas densas, mantendo a informação necessária para a classificação.

A camada densa Fully Connected, por sua vez, contém 128 neurônios e cada neurônio está completamente conectado a todos os neurônios da camada anterior. A função de ativação Relu é aplicada para introduzir não-linearidade no modelo.

A vantagem desta camada densa é que permite que o modelo aprenda representações complexas dos dados de entrada. A ativação Relu ajuda a acelerar a convergência durante o treinamento e a combater o problema de gradientes que desaparecem.

Por último, a camada densa de Saída contém 8 neurônios, correspondendo ao número de classes no problema de classificação (neste caso, 8 tipos diferentes de células sanguíneas). A função de ativação softmax é utilizada para converter os valores de saída em probabilidades que somam 1, facilitando a interpretação dos resultados como probabilidades de classificação.

A grande vantagem desta camada é que a softmax é adequada para problemas de classificação multiclasse, pois fornece uma distribuição de probabilidade sobre as classes.

Para a compilação do modelo foi utilizada os seguintes métodos:

Optimizer (adam**):** O otimizador Adam combina as melhores propriedades dos algoritmos AdaGrad e RMSProp para fornecer um método eficiente e eficaz para otimização estocástica. Ele ajusta a taxa de aprendizado durante o treinamento, o que muitas vezes resulta em uma convergência mais rápida.

Loss (categorical_crossentropy**):** A função de perda categorical_crossentropy é usada para medir a performance do modelo na classificação multiclasse. Ela calcula a diferença entre as distribuições de probabilidade previstas e as reais.

Metrics (accuracy**):** A acurácia é usada como uma métrica para monitorar a proporção de previsões corretas durante o treinamento e a avaliação do modelo.

Por fim, utilizamos o método "fit" com 100 épocas de treinamento usando 32 amostras de treinamento por bath para treinar o modelo.

```
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 3)))
model.add(Dense(128, activation='relu'))
model.add(Dense(8, activation='softmax'))

# Compilar o modelo
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Após o treinamento do modelo utilizamos o método predct e confusion_matrix para, a partir dos dados preditos pelo modelo temos o resultado da matriz de confusão e por meio dela analisar a eficiência do nosso modelo.

```python
# Avaliação no conjunto de teste
loss, accuracy = model.evaluate(x_test, y_test)
print(f'Accuracy: {accuracy}')

# Prever as classes do conjunto de teste
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

# Matriz de confusão
conf_matrix = confusion_matrix(y_true, y_pred_classes)

# Plotando a matriz de confusão
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=range(8), yticklabels=range(8))
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

O resultado final do modelo foi o seguinte:

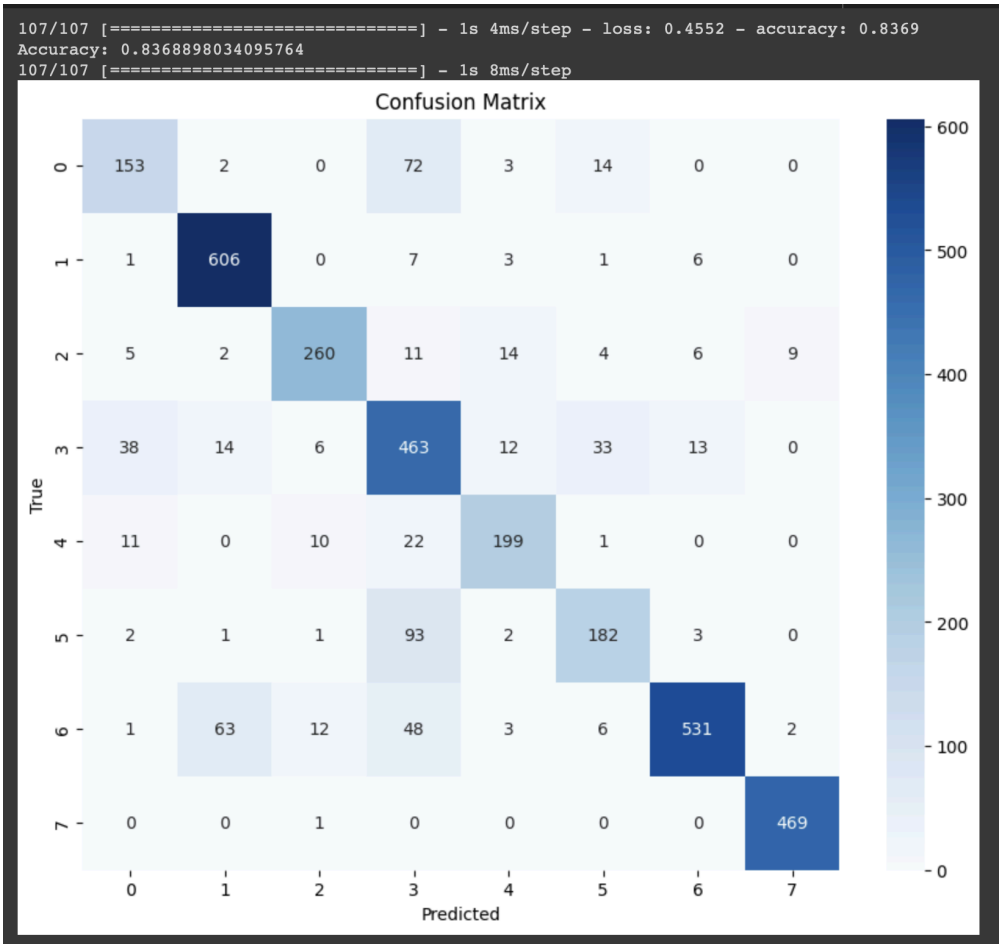

Figura 1. Matriz de confusão e acurácia do modelo MPL

O resultado da acurácia de 83,69%(valor logo acima da matriz) pode ser observado na matriz de confusão.

Estudo dos dados com uma CNN:

O estudo dos dados com a CNN foi feito usando (i) uma camada convolucional usando uma função de ativação não-linear, (ii) uma camada de Pooling, (iii) uma camada de saída do tipo Softmax.

Para isso, primeiramente definimos uma função que constrói e compila uma rede neural simples.

```
def create_model(num_kernels, kernel_size):
    model = Sequential([
        Conv2D(num_kernels, kernel_size=kernel_size, activation='relu', input_shape=(28, 28, 3)),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(8, activation='softmax')
    ])

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

A função create_model contém a camada de convolução dependente do número e tamanho de Kernels, contém a camada de Pooling. Nesta camada a janela de 2x2 reduz a entrada para metade de sua dimensão original em cada eixo preservando as características mais importantes e fornecendo invariância a pequenas traduções na imagem.

Além disso, também é adicionada uma camada Flatten e uma camada Densa com a função de ativação softmax.

Após isso, fazemos o treinamento da rede neural com diferentes números e tamanhos de kernels com o objetivo de encontrar o melhor modelo possível.

```
# Parâmetros para testar
kernel_numbers = [8, 16, 32]
kernel_sizes = [(3, 3), (5, 5), (7, 7)]

# Função para treinar e avaliar o modelo
def train_and_evaluate(num_kernels, kernel_size):
  model = create_model(num_kernels, kernel_size)
  model.fit(x_train, y_train, epochs=40, batch_size=32, validation_data=(x_test, y_test))
  loss, accuracy = model.evaluate(x_test, y_test)
  return accuracy

# Testar diferentes combinações de parâmetros
results = {}
for num_kernels in kernel_numbers:
    for kernel_size in kernel_sizes:
        accuracy = train_and_evaluate(num_kernels, kernel_size)
        results[(num_kernels, kernel_size)] = accuracy
        print(f'Num Kernels: {num_kernels}, Kernel Size: {kernel_size}, Accuracy: {accuracy}')

# Exibir os resultados
for params, accuracy in results.items():
    print(f'Kernels: {params[0]}, Kernel Size: {params[1]} => Accuracy: {accuracy}')
```

Observe que o treinamento do modelo foi feito com apenas 40 épocas, um número bem inferior ao usado no modelo anterior.

O resultado obtido pode ser observado logo abaixo:

```
Num Kernels: 32, Kernel Size: (7, 7), Accuracy: 0.8918445110321045
Kernels: 8, Kernel Size: (3, 3) => Accuracy: 0.8798596858978271
Kernels: 8, Kernel Size: (5, 5) => Accuracy: 0.8731365203857422
Kernels: 8, Kernel Size: (7, 7) => Accuracy: 0.8766442537307739
Kernels: 16, Kernel Size: (3, 3) => Accuracy: 0.8801519870758057
Kernels: 16, Kernel Size: (5, 5) => Accuracy: 0.8684595227241516
Kernels: 16, Kernel Size: (7, 7) => Accuracy: 0.8871675133705139
Kernels: 32, Kernel Size: (3, 3) => Accuracy: 0.9038292765617371
Kernels: 32, Kernel Size: (5, 5) => Accuracy: 0.8880444169044495
Kernels: 32, Kernel Size: (7, 7) => Accuracy: 0.8918445110321045
```

Observe que a acurácia do modelo tende a aumentar com o número de kernels utilizado.Mas, escolhido um número de kernels, a acurácia varia muito pouco com relacao ao tamanho dos mesmos.

Avaliação do melhor modelo encontrado:

Após a análise da relação entre número e tamanho dos kernels pode-se encontrar e avaliar o melhor modelo de CNN.

```python
best_params = max(results, key=results.get)
best_model = create_model(best_params[0], best_params[1])
best_model.fit(x_train, y_train, epochs=40, validation_data=(x_test, y_test))

# Avaliar o modelo final
loss, accuracy = best_model.evaluate(x_test, y_test)
print(f'Melhor configuração - Num Kernels: {best_params[0]}, Kernel Size: {best_params[1]}, Accuracy: {accuracy}')
```

Para isso usamos a função max para encontrar a chave do dicionário results que tem o maior valor. Após isso, guardamos o valor da tupla retornado por esta função na variável best_params. Ou seja, armazenamos o número de kernels e o tamanho dos kernels do melhor modelo.

Após isso, criamos novamente uma CNN mas agora com os melhores hiperparâmetros e treinamos este modelo com o mesmo número de épocas usado para encontrá-lo. O mesmo número de épocas foi utilizado para que a acurácia deste modelo seja próximo ao que ele resultou anteriormente na análise dos kernels.

Após isso re-avaliamos a performance do modelo final.

```
Melhor configuração - Num Kernels: 32, Kernel Size: (3, 3), Accuracy: 0.8830751180648804
```

Apesar de destoar da acurácia anterior em, aproximadamente, 0.02 o valor ainda continua com uma acurácia bem elevada mesma para um número de época de treinamento muito mais baixa que a utilizada pelo modelo anterior o que mostra o poder que tem este rede neural em relação à rede MLP.

É possível também analisar a matriz de confusão para analisar o comportamento dessa rede neural. Utilizando os mesmo métodos que utilizamos na última análise de matriz de confusão chegamos ao seguinte resultado:

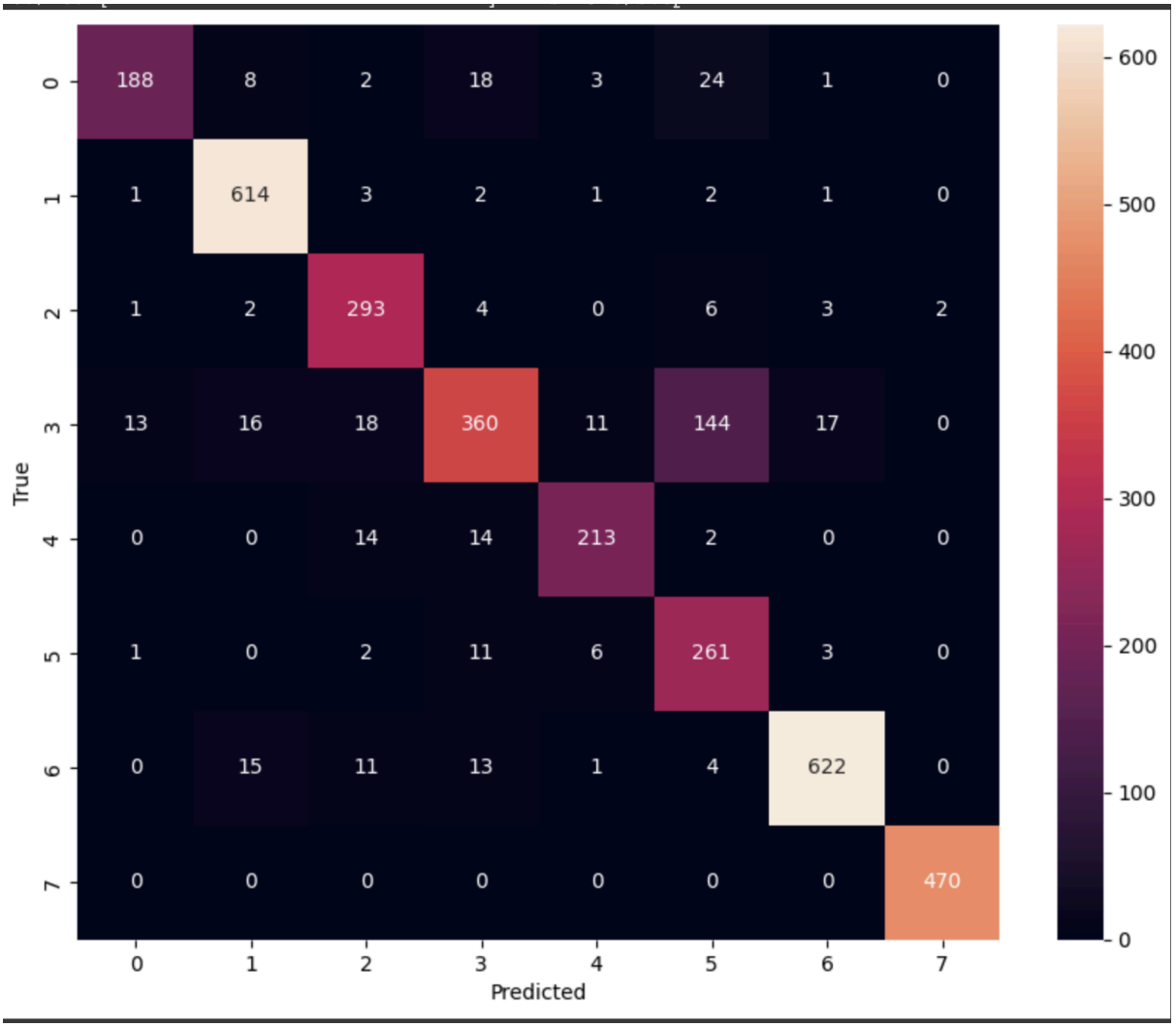

Figura 2. Matriz de confusão e acurácia do modelo CNN

	precision	recall	f1-score	support
0	0.92	0.77	0.84	244
1	0.94	0.98	0.96	624
2	0.85	0.94	0.90	311
3	0.85	0.62	0.72	579
4	0.91	0.88	0.89	243
5	0.59	0.92	0.72	284
6	0.96	0.93	0.95	666
7	1.00	1.00	1.00	470
accuracy			0.88	3421
macro avg	0.88	0.88	0.87	3421
weighted avg	0.90	0.88	0.88	3421

Figura 3. Dados estatísticos em relação a eficiência do modela para cada uma das 8 labels

Segue abaixo exemplos de casos classificados incorretamente pela rede CNN com os resultados reais e os resultados do modelo:

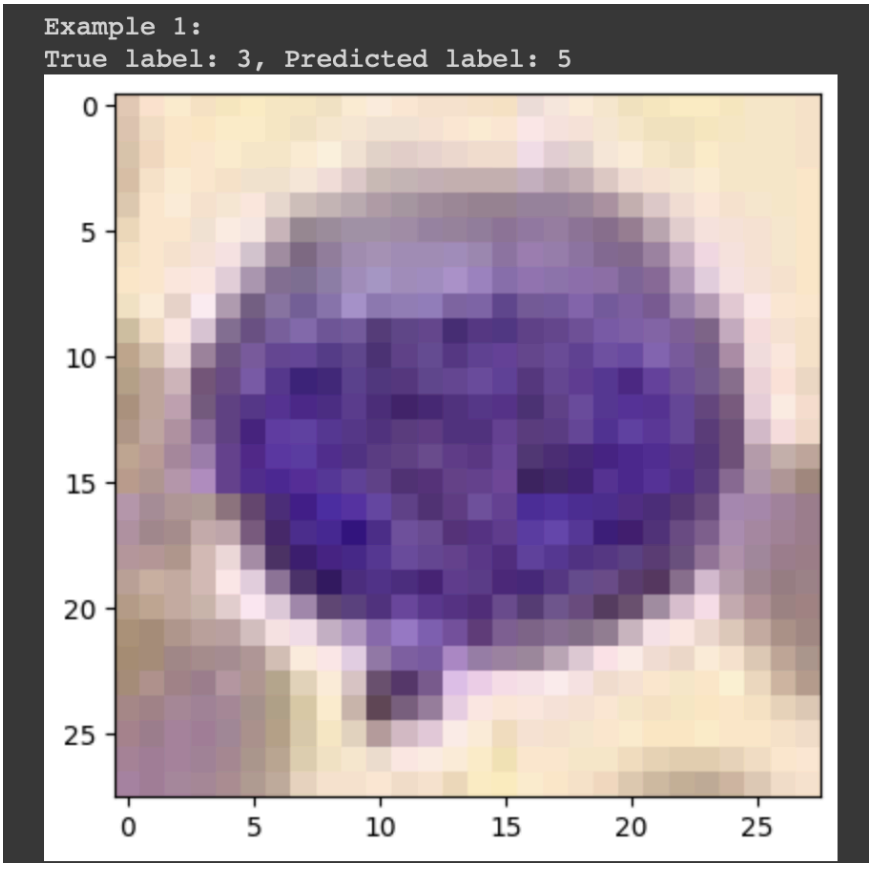

Figura 4. Exemplo 1 de resultado incorreto do modelo

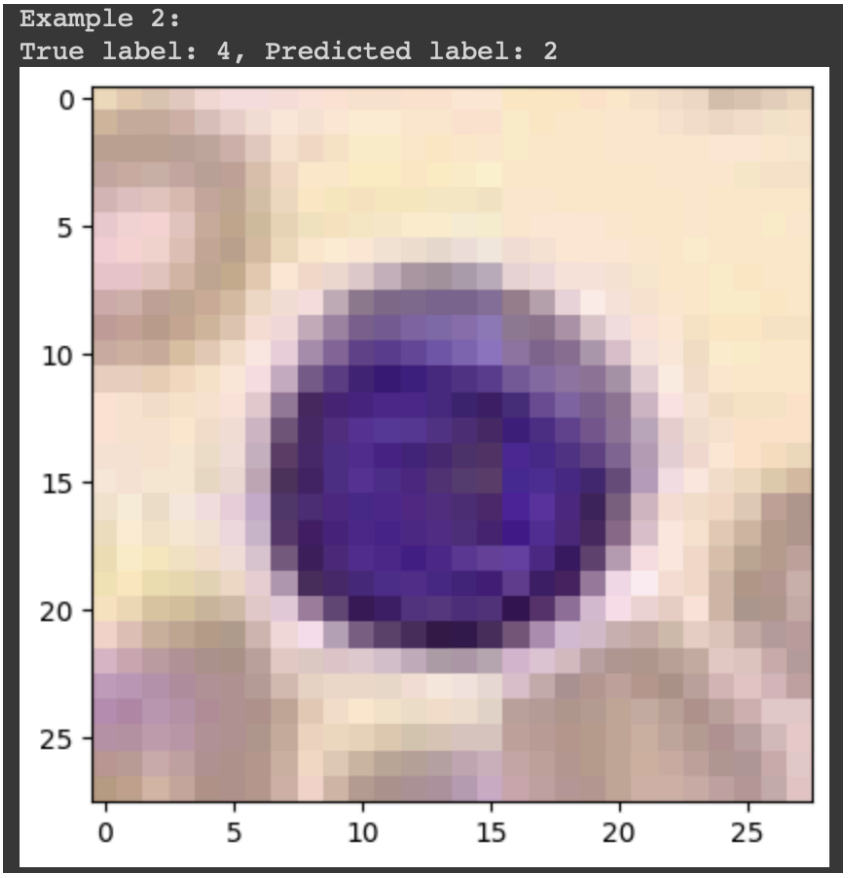

Figura 5. Exemplo 2 de resultado incorreto do modelo

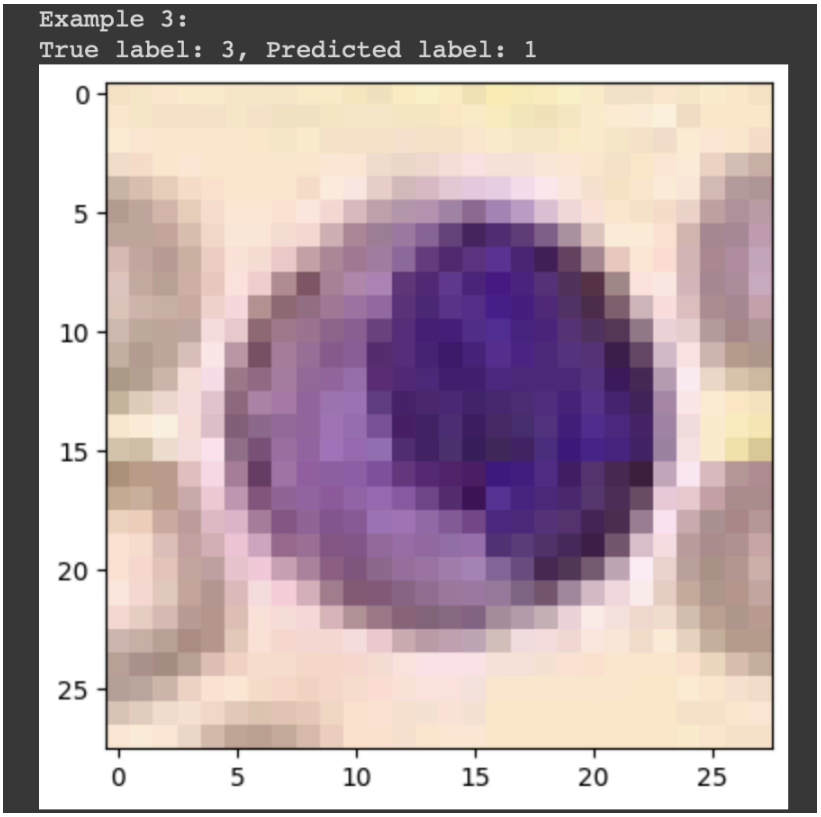

Figura 6. Exemplo 3 de resultado incorreto do modelo

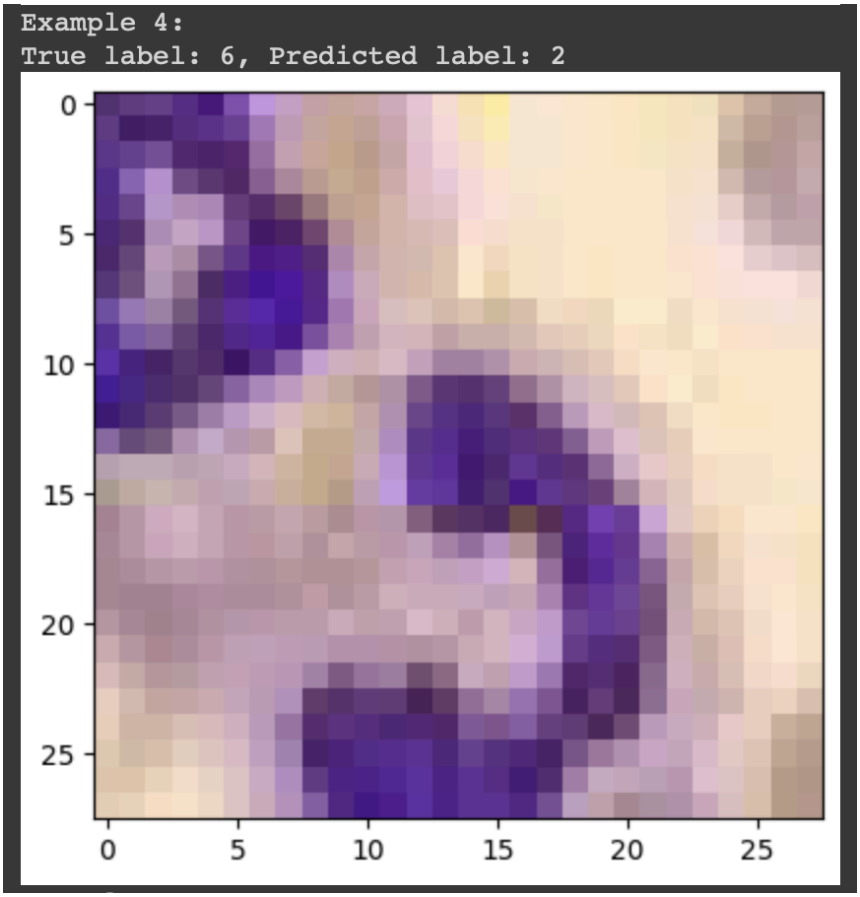

Figura 7. Exemplo 4 de resultado incorreto do modelo

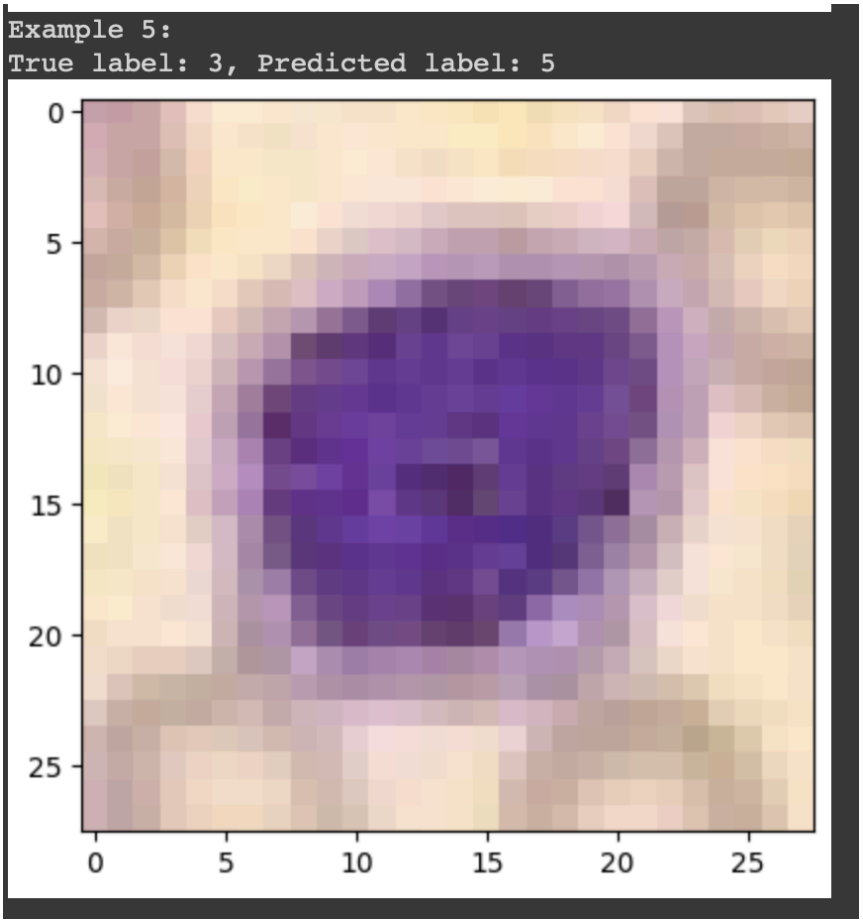

Figura 8. Exemplo 5 de resultado incorreto do modelo

Estudo dos dados com uma Resnet:

Neste último tópico faremos o estudo dos dados por meio de um modelo mais profundo usando uma CNN Resnet. Para isto iremos utilizar as bibliotecas Tensorflow e Keras para construir a Resnet.

Primeiramente definimos um bloco básico de convolução com batch normalization e uma ativação ReLU opcional.

O Batch Normalization tem um papel fundamental nesta camada, pois ele normaliza as ativações da camada anterior para uma distribuição com média zero e variância um. Isso ajuda a estabilizar o treinamento e permite que o modelo aprenda mais rapidamente. Além disso, ele reduz o problema do Vanishing Gradiente ajudando a manter os gradientes em uma faixa útil, tornando o treinamento de redes profundas mais viável. Dessa forma, o ele torna o

aprendizado mais rápido e eficiente.

```
def resnet_block(inputs, filters, kernel_size, strides, use_activation=True):
  x = Conv2D(filters, kernel_size=kernel_size, strides=strides, padding='same')(inputs)
  x = BatchNormalization()(x)
  if use_activation:
      x = Activation('relu')(x)
  return x
```

Após a definição do primeiro bloco, definimos então o bloco residual, onde irá permitir a rede neural a aprender com seus resíduos. Esta camada é crucial para o funcionamento da Resnet.

Esta camada será composta por 2 camadas de convolução sequenciais onde a primeira camada convolucional serve para extrair características básicas dos dados de entrada, como bordas e texturas simples e a segunda camada convolucional, operando sobre as características extraídas pela primeira camada, pode capturar padrões mais complexos e combinações de características simples.

Após isso, foi criado uma variável shortcut com uma convolução 1x1 para igualar as dimensões, uma função Add() para somar a saída da convolução principal com a saída do atalho e uma função de ativação ReLU aplicada à soma.

```
def residual_block(inputs, filters, kernel_size, strides):
  x = resnet_block(inputs, filters, kernel_size, strides)
  x = resnet_block(x, filters, kernel_size, 1, use_activation=False)
  shortcut = Conv2D(filters, kernel_size=(1, 1), strides=strides, padding='same')(inputs)
  shortcut = BatchNormalization()(shortcut)
  x = Add()([x, shortcut])
  x = Activation('relu')(x)
  return x
```

Por último, definimos um método para a criação do modelo a ser aplicado na classificação de imagens. O método recebe imagens e o número de classes como parâmetros. Neste método usamos:

- A função resnet_block para criar uma camada convolucional com 64 filtros, tamanho de filtro 7x7 e passo 2;
- Um MaxPoolin2D para reduzir a dimensão espacial da saída, uma sequência de blocos residuais com diferentes números de filtros e tamanhos de filtro, reduzindo a dimensão espacial e aumentando a profundidade,
- Um GlobalAveragePooling2D que reduz a dimensão espacial final para um vetor, preparado para a camada densa de saída;
- Uma camada densa de saída com número de neurônios igual ao número de classes, com ativação softmax para classificação;
- O método compile que compila o modelo com o otimizador Adam e a perda de entropia cruzada categórica.

```python
def resnet_block(inputs, filters, kernel_size, strides, use_activation=True):
  x = Conv2D(filters, kernel_size=kernel_size, strides=strides, padding='same')(inputs)
  x = BatchNormalization()(x)
  if use_activation:
      x = Activation('relu')(x)
  return x

def residual_block(inputs, filters, kernel_size, strides):
  x = resnet_block(inputs, filters, kernel_size, strides)
  x = resnet_block(x, filters, kernel_size, 1, use_activation=False)
  shortcut = Conv2D(filters, kernel_size=(1, 1), strides=strides, padding='same')(inputs)
  shortcut = BatchNormalization()(shortcut)
  x = Add()([x, shortcut])
  x = Activation('relu')(x)
  return x

def create_resnet(input_shape, num_classes):
  inputs = Input(shape=input_shape)
  x = resnet_block(inputs, 64, (7, 7), 2)
  x = MaxPooling2D(pool_size=(3, 3), strides=2, padding='same')(x)

  x = residual_block(x, 64, (3, 3), 1)
  x = residual_block(x, 64, (3, 3), 1)

  x = residual_block(x, 128, (3, 3), 2)
  x = residual_block(x, 128, (3, 3), 1)

  x = residual_block(x, 256, (3, 3), 2)
  x = residual_block(x, 256, (3, 3), 1)

  x = residual_block(x, 512, (3, 3), 2)
  x = residual_block(x, 512, (3, 3), 1)

  x = GlobalAveragePooling2D()(x)
  outputs = Dense(num_classes, activation='softmax')(x)

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

# Criar a ResNet
input_shape = (28, 28, 3)
num_classes = 8
resnet_model = create_resnet(input_shape, num_classes)
```

Após a definição da Resnet, foi feito o treinamento do modelo e então chegamos ao valor da acurácia global da nossa Rede neural profunda.

```python
# Treinar o modelo ResNet
resnet_model.fit(x_train, y_train, epochs=20, validation_data=(x_test, y_test))

# Avaliar o modelo ResNet
loss, accuracy = resnet_model.evaluate(x_test, y_test)
print(f'Acurácia global da ResNet: {accuracy}')
```

```
Acurácia global da ResNet: 0.8757672905921936
```

Figura 9. Acurácia global do modelo Resnet

Por fim analisamos analisamos a matriz de confusão para analisar o comportamento da rede neural ResNet utilizando os mesmo métodos que utilizamos na última análise de matriz de confusão o resultado pode ser visto abaixo, assim como os exemplos:

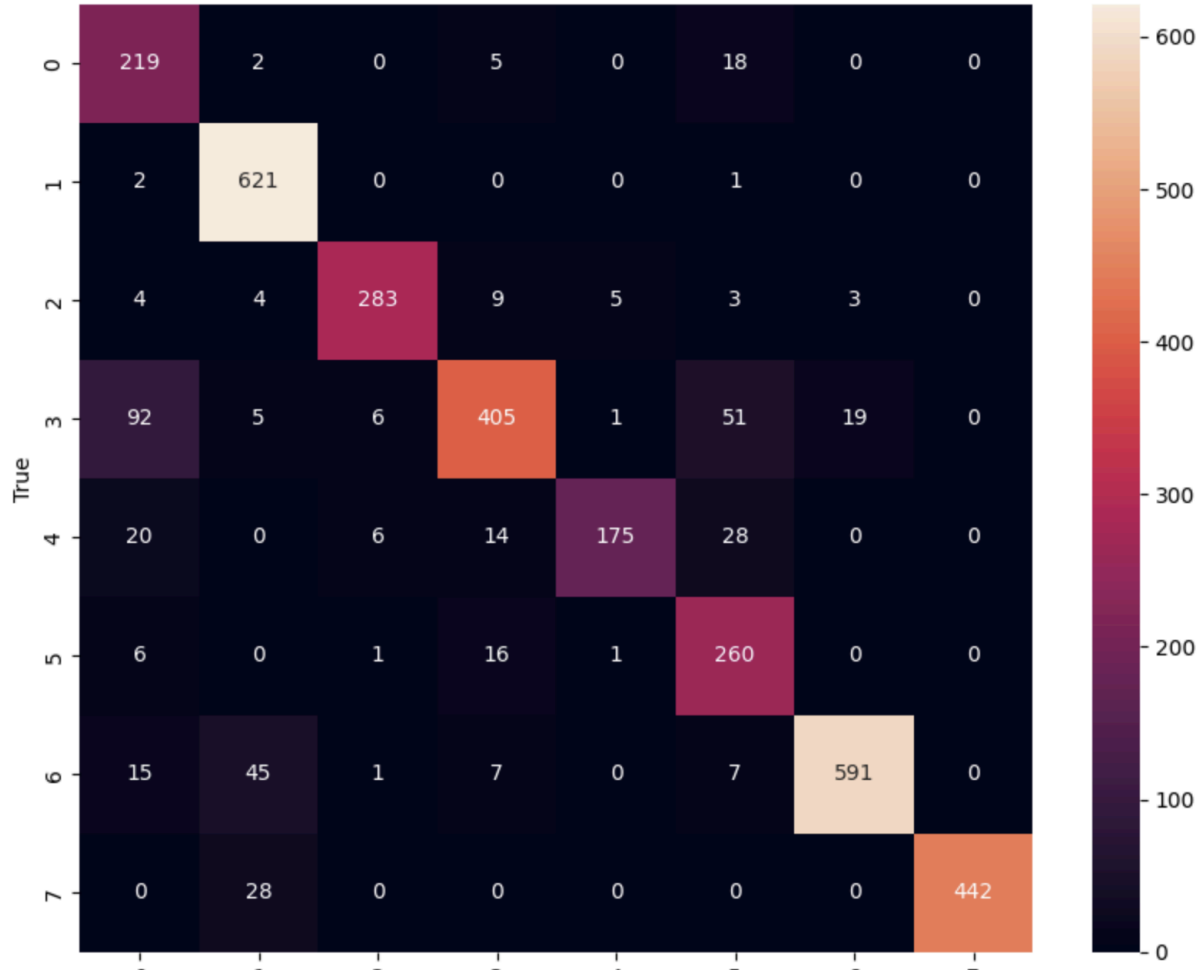

Figura 10. Matriz de confusão do modelo Resnet

	precision	recall	f1-score	support
0	0.61	0.90	0.73	244
1	0.88	1.00	0.93	624
2	0.95	0.91	0.93	311
3	0.89	0.70	0.78	579
4	0.96	0.72	0.82	243
5	0.71	0.92	0.80	284
6	0.96	0.89	0.92	666
7	1.00	0.94	0.97	470
accuracy			0.88	3421
macro avg	0.87	0.87	0.86	3421
weighted avg	0.89	0.88	0.88	3421

Figura 11. Dados estatísticos em relação a eficiência do modelo Resnet para cada uma das 8 labels

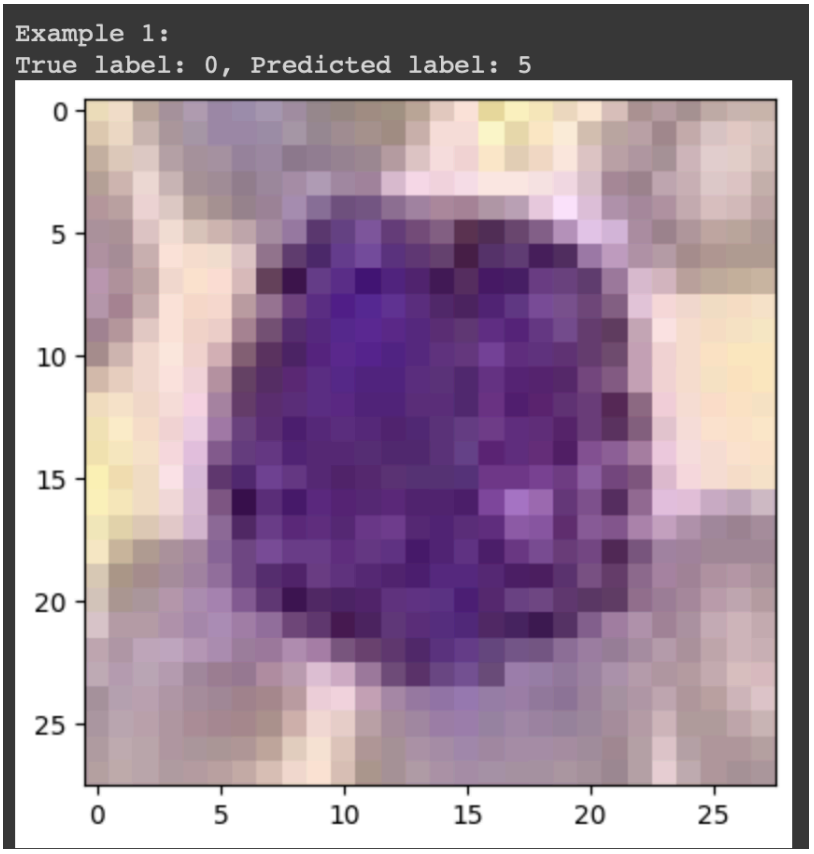

Figura 12. Exemplo 1 de resultado incorreto do modelo Resnet

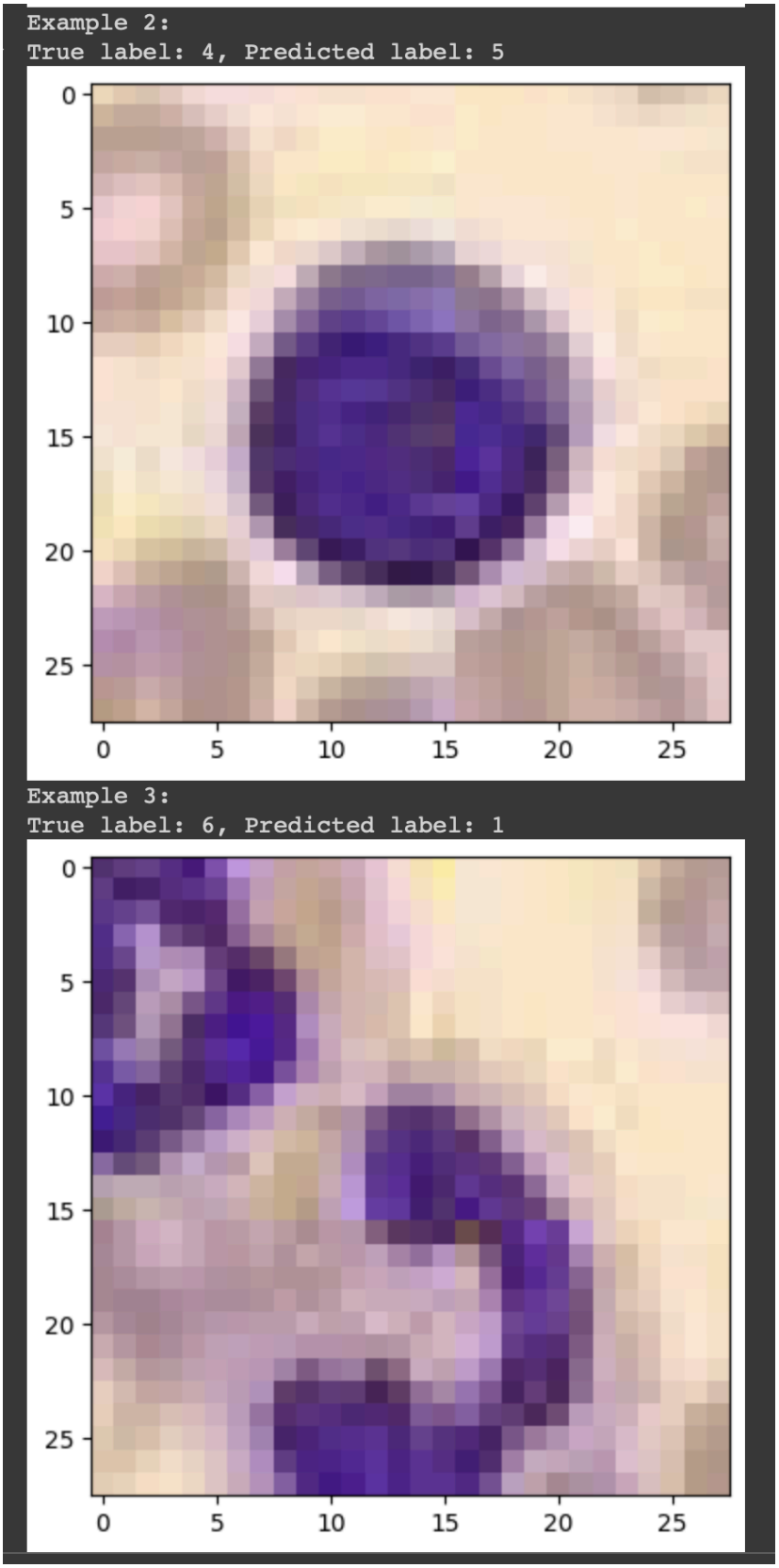

Figura 13. Exemplo 2 e 3 de resultado incorreto do modelo Resnet

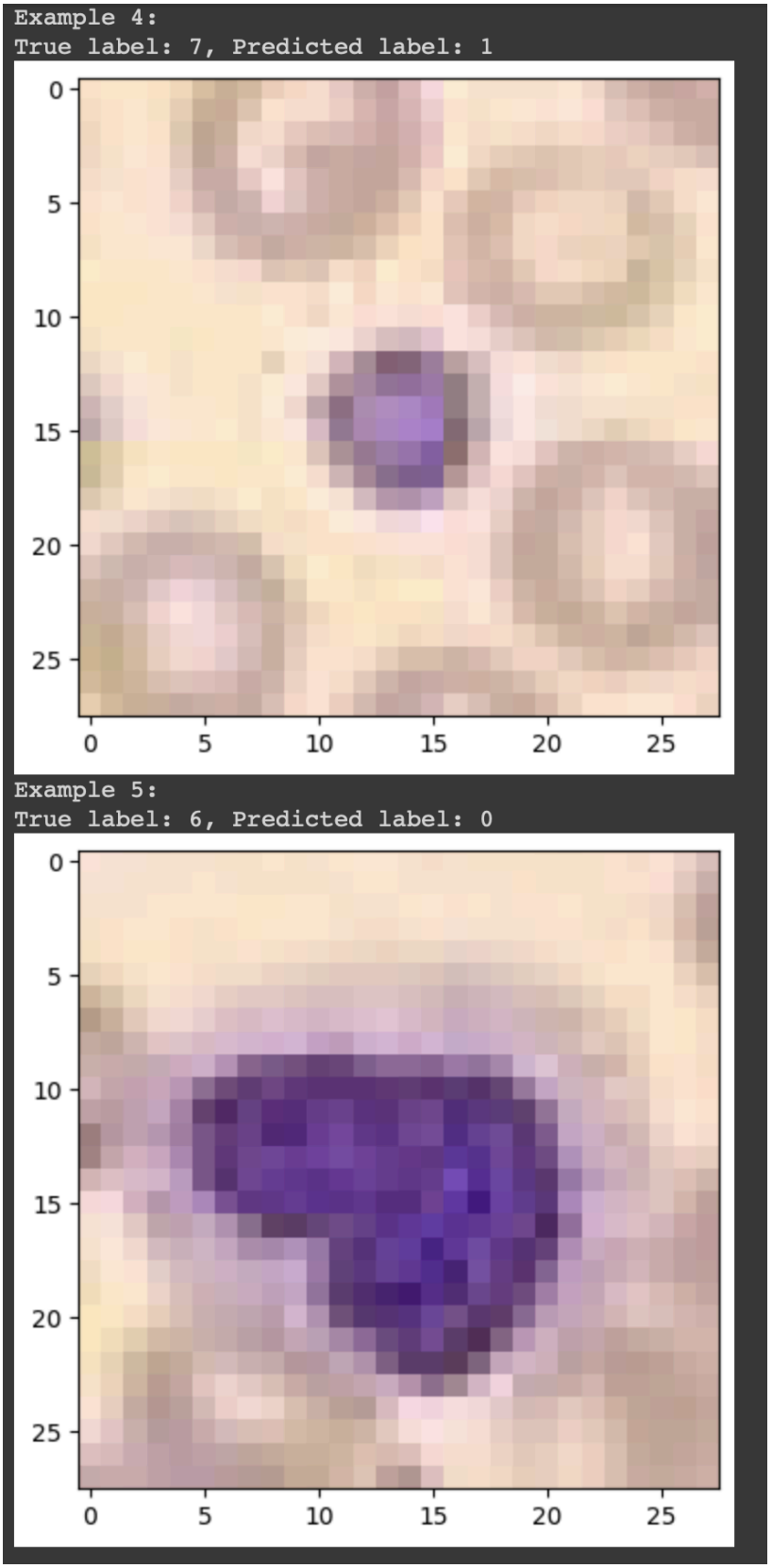

Figura 13. Exemplo 4 e 5 de resultado incorreto do modelo Resnet

Conclusão:

A partir do conjunto de dados oferecido pela biblioteca BloodMnist foi possível analisar 3 modelos de redes neurais(MLP, CNN e Resnet) e analisar as suas diferenças. Neste caso, foi possível verificar o impacto que a profundidade da rede neural possui em seus resultados no caso da classificação de imagens. A partir deste estudo foi possível verificar qual a relação da profundidade da rede com a velocidade de treinamento, esforço computacional, velocidade e assertividade da classificação. A conclusão foi que quanto mais profunda for a rede neural, mais assertiva são as suas classificações. Porém, maior será o esforço computacional para treiná-las também, mesmo utilizando de ferramentas de bibliotecas especializadas. Em contrapartida, quanto maior for a profundidade da rede neural, mais rápida e assertiva será a sua classificação.