Guilherme Lira Ribeiro - 231013538

Trabalho Computacional 2: Perceptron Multicamada no problema MNIST

Este caderno tem como objetivo o uso de Multi Layer Perceptron (MLP) para o problema de classificação de dígitos manuscritos MNIST. Foram testados alguns modelos que diferiram na quantidade de camadas, na função de ativação, e também nas configurações de otimização. Além disso, foram apresentadas imagens de exemplos que foram classificados erroneamente pelo modelo. Ao final, a matriz de confusão do melhor classificador, no questo de acurácia, foi construída usando o módulo `sklearn.metrics`.

1. Importação de dependências e definição de classes

Para treinar os modelos, foi usada a biblioteca `pytorch`, bem como recursos da biblioteca disponibilizada pelo livro Dive into Deep Learning (`d2l`), a biblioteca `matplotlib` para a construção de gráficos e `sklearn` para a métrica de matriz de confusão.

In [1]:
```
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l
from torch import nn
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Aproveitando as classes fornecidas pelo módulo `d2l`, serão definidas as classes `MNIST` e `MLP`. A subclasse `MNIST` de `d2l.DataModule` será responsável pelo encapsulamento do carregamento e pré-processamento do *dataset* MNIST. Esta classe é responsável pela lógica de leitura dos dados, redimensionamento das imagens e conversão para tensores PyTorch.

In [2]:
```
class MNIST(d2l.DataModule):  #@save
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()

        trans = transforms.Compose([
            transforms.Resize(resize),  # Imagens do tamanho 28x28
            transforms.ToTensor()       # Converte de PIL.Image para tensor
        ])

        self.train = torchvision.datasets.MNIST(
            root=self.root, train=True, transform=trans, download=True)

        self.val = torchvision.datasets.MNIST(
            root=self.root, train=False, transform=trans, download=True)
```

In [3]:
```
dataset = MNIST()
print(len(dataset.train), len(dataset.val))
print(dataset.train.data.shape)
```

```
60000 10000
torch.Size([60000, 28, 28])
```

Como pode ser observado na saída da célula anterior, o dataset MNIST consiste de 60000 exemplos de treino e 10000 exemplos de validação. O tensor de treinamento é tridimensional, com cada um dos 60000 exemplos sendo uma matriz de 28 x 28.

In [4]:
```
@d2l.add_to_class(MNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

Esta célula adiciona o método `get_dataloader` à classe `MNIST`. É responsável por retornar o `DataLoader` adequado para o treinamento ou validação. Organiza os dados em lotes (batches) e permite que sejam carregados de forma eficiente durante o processo de aprendizado. O argumento train define se o loader será do conjunto de treino ou validação, o que é usado internamente pelas funções `train_dataloader` e `val_dataloader`.

In [5]:
```
X, y = next(iter(dataset.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 28, 28]) torch.float32 torch.Size([64]) torch.int64
```

Na célula anterior pode-se verificar o tamanho do batch, sendo de 64 imagens, cada uma com 1 canal de cores, na escala de cinza, e dimensão 28 por 28.

1.2 MLP

In [6]:
```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, hidden_sizes, lr, af='sigmoid'):
        super().__init__()
        self.save_hyperparameters()

        if af == 'sigmoid':
            activation = nn.Sigmoid
        elif af == 'relu':
            activation = nn.ReLU
        else:
            raise ValueError(f"função de ativação '{af}' não suportada.")

        layers = [nn.Flatten()]
        for size in hidden_sizes:
            layers.append(nn.LazyLinear(size))
            layers.append(activation())

        layers.append(nn.LazyLinear(num_outputs))

        self.net = nn.Sequential(*layers)
```

A classe MLP encapsula o modelo de perceptron multicamada que será usado como classificador para o MNIST. Como modelos com diferentes configurações serão testados, o construtor da classe recebe alguns parâmetros, como `num_outputs`, que no caso sempre é 10 neste caderno, tendo em vista que é o número de classes do problema MNIST (dígitos de 0 a 9). `hidden_sizes`, que é uma lista de inteiros contendo o tamanho das camadas ocultas, e `af`, abreviação para *activation function*, que pode ser "sigmoid" ou "relu".

2. Funções auxiliares

Algumas funções auxiliares são úteis, como a `evaluate_accuracy`, para fornecer a métrica de acurácia dos diferentes modelos a serem treinados, e também a função `visualize_errors` e `show_confusion_matrix`.

In [7]:
```
def evaluate_accuracy(model, data_iter):
    model.eval()
    metric = d2l.Accumulator(2)
    with torch.no_grad():
        for X, y in data_iter:
            metric.add(d2l.accuracy(model(X), y), y.numel())
    model.train()
    return metric[0] / metric[1]
```

A função `evaluate_accuracy`, definida na célula anterior, calcula a acurácia do modelo sobre um conjunto de dados, desativando o modo de treinamento (model.eval()) para garantir comportamento consistente (como desativar dropout). Ela acumula o número total de acertos e divide pelo número total de exemplos, retornando a proporção de classificações corretas.

In [8]:
```
def visualize_errors(model, dataset, max_errors=10):
    model.eval()
    errors_found = 0
    images_error = []
    correct_labels = []
    predicted_labels = []

    with torch.no_grad():
        for X, y in dataset.val_dataloader():
            preds = model(X).argmax(axis=1)
            error_mask = preds != y

            if error_mask.any():
                X_errors = X[error_mask]
                y_true = y[error_mask]
                y_pred = preds[error_mask]

                for img, true_label, pred_label in zip(X_errors, y_true, y_pred):
                    images_error.append(img)
                    correct_labels.append(int(true_label))
                    predicted_labels.append(int(pred_label))
                    errors_found += 1

                    if errors_found >= max_errors:
                        break

            if errors_found >= max_errors:
                break

    fig, axes = plt.subplots(1, len(images_error), figsize=(2 * len(images_error), 2))
    if len(images_error) == 1:
        axes = [axes]

    for ax, img, true_label, pred_label in zip(axes, images_error, correct_labels, predicted_labels):
        ax.imshow(img.squeeze(), cmap='gray')
        ax.set_title(f"Verdadeiro: {true_label}\nPrevisto: {pred_label}")
        ax.axis('off')

    plt.tight_layout()
    plt.show()
```

Para entender melhor os padrões que a rede falha em classificar corretamente, foram extraídas e visualizadas algumas amostras do conjunto de validação em que as previsões foram incorretas, por meio da função `visualize_errors`. A função percorre os minibatches até coletar um número fixo de erros (ex: 10), tratando corretamente casos em que um minibatch não contém erros.

Essa visualização nos permite verificar se os erros são compreensíveis também para um humano (por exemplo, 5 confundido com 6, 1 confundido com 2), o que indica limites intrínsecos dos dados e não necessariamente falhas graves do modelo.

In [9]:
```
def show_confusion_matrix(model, dataset):
    y_true_all = []
    y_pred_all = []

    for X, y in dataset.val_dataloader():
        preds = model(X).argmax(axis=1)
        y_true_all.extend(y.numpy())
        y_pred_all.extend(preds.numpy())

    cm = confusion_matrix(y_true_all, y_pred_all)

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.xlabel('Predito')
    plt.ylabel('Verdadeiro')
    plt.title('Matriz de Confusão')
    plt.show()

    return cm
```

A matriz de confusão revela onde ocorrem os principais erros de classificação. Ela mostra quantas vezes cada classe verdadeira foi confundida com outra. Portanto, a função `show_confusion_matrix` é tão útil quanto `visualize_errors` para verificar se os erros estão mais relacionados ao modelo ou a limitações intrínsecas dos dados de entrada, isto é, seres humanos também poderiam razoavelmente cometer os mesmos erros de classificação.

3. Testando diferentes modelos

3.1 Uma camada oculta, sigmoide como ativação

O primeiro modelo, chamado de `model_a`, consiste em uma instância de `MLP` com uma única camada oculta, de 128 neurônios, e a função de ativação sendo a sigmoide.

In [10]:
```
model_a = MLP(num_outputs=10,    # 10 classes do MNIST (Dígitos 0-9)
              hidden_sizes=[128], # 128 neurônios na camada escondida
              lr=0.1)             # learning rate pequeno

data = MNIST(batch_size=64)
```

```
/home/guilherme/miniconda3/envs/d2l/lib/python3.9/site-packages/torch/nn/modules/lazy.py:180: UserWarning: Lazy modules are a new feature under heavy development so changes to the API or functionality can happen at any moment.
  warnings.warn('Lazy modules are a new feature under heavy development '
```

In [11]:
```
trainer_a = d2l.Trainer(max_epochs=10)
trainer_a.fit(model_a, data)
```

In [12]:
```
print('Acurácia de validação para modelo A: ', evaluate_accuracy(model_a, data.val_dataloader()))
```

```
Acurácia de validação para modelo A:  0.9645
```

Como visto nas células anteriores, o treinamento se deu em dez épocas, e o resultado obtido foi satisfatório para o primeiro modelo, com uma acurácia de 94,4%.

3.2: Uma camada oculta, otimização ADAM

Nesta variação, a arquitetura básica do modelo anterior (MLP com uma camada escondida de 128 neurônios e ativação sigmoide), foi mantida, mas substituiu-se a otimização por descida de gradiente simples pelo otimizador Adam, que realiza ajustes adaptativos do *learning rate* para cada parâmetro. Essa troca visa acelerar a convergência e potencialmente melhorar o desempenho do modelo, mesmo sem alterações na estrutura da rede.

In [13]:
```
@d2l.add_to_class(MLP)
def configure_optimizers(self):
    return torch.optim.Adam(self.parameters(), lr=self.lr)
```

In [14]:
```
model_b = MLP(num_outputs=10, hidden_sizes=[128], lr=0.001)
```

A célula anterior inicializa o modelo `model_b`, com uma única diferença nos parâmetros, o learning rate de 0.001, que é um valor amplamente considerado adequado para o otimizador Adam, uma vez que valores altos, como o usado no `model_a`, podem causar instabilidade no treinamento.

In [15]:
```
trainer_b = d2l.Trainer(max_epochs=10)
trainer_b.fit(model_b, data)
```

In [16]:
```
print('Acurácia de validação para modelo B: ', evaluate_accuracy(model_b, data.val_dataloader()))
```

```
Acurácia de validação para modelo B:  0.9742
```

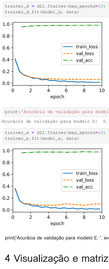
Percebe-se que o modelo b é melhor que o modelo a, com uma acurácia de 97,4%, portanto, erra cerca da metade do primeiro modelo, indicando que o Adam foi mais eficaz em ajustar os pesos da rede. Essa melhora reflete a capacidade do Adam de adaptar dinamicamente a taxa de aprendizado, o que torna o treinamento mais eficiente.

3.3 Uma camada oculta, função de ativação ReLU

Neste terceiro modelo, manteve-se a arquitetura com uma única camada oculta, mas a função de ativação passou de sigmoide pela ReLU (*Rectified Linear Unit*). Essa alteração visa melhorar a eficiência do treinamento e a capacidade de aprendizado, pois a ReLU costuma evitar problemas comuns da sigmoide, como o gradiente evanescente.

In [17]:
```
model_c = MLP(num_outputs=10, hidden_sizes=[128], lr=0.001, af='relu')
```

In [18]:
```
trainer_c = d2l.Trainer(max_epochs=10)
trainer_c.fit(model_c, data)
```



In [19]:
```
print('Acurácia de validação para modelo C: ', evaluate_accuracy(model_c, data.val_dataloader()))
```

```
Acurácia de validação para modelo C:  0.9781
```

Como visto na célula anterior, o `model_c` atingiu a maior acurácia, embora a melhora em relação ao `model_b` tenha sido sutil. Isso reflete que as vantagens da ReLU sobre a sigmoide costumam ser mais evidentes em redes mais profundas.

3.4 Mais neurônios e camadas

Com os ajustes anteriores resultando em melhora da acurácia, o passo natural para tentar melhorar o modelo é o uso de uma camada oculta com mais neurônios, ou mesmo múltiplas camadas ocultas. A seguir, foram testados o `model_d` com uma camada oculta de 256 neurônios, e o `model_e` com duas camadas de 256 neurônios. No tocante as demais características, serão mantidas as do `model_c`, ou seja, ReLU como função de ativação, e o otimizador Adam.

In [20]:
```
model_d = MLP(num_outputs=10, hidden_sizes=[256], lr=0.001, af='relu')
model_e = MLP(num_outputs=10, hidden_sizes=[256, 256], lr=0.001, af='relu')
```

In [21]:
```
trainer_d = d2l.Trainer(max_epochs=10)
trainer_d.fit(model_d, data)
```

In [22]:
```
print('Acurácia de validação para modelo D: ', evaluate_accuracy(model_d, data.val_dataloader()))
```

```
Acurácia de validação para modelo D:  0.9794
```

In [23]:
```
trainer_e = d2l.Trainer(max_epochs=10)
trainer_e.fit(model_e, data)
```

print('Acurácia de validação para modelo E: ', evaluate_accuracy(model_e, data.val_dataloader()))

4 Visualização e matriz de confusão

Com base na saída de avaliação de acurácia, percebe-se que o melhor modelo treinado foi o modelo E, que consiste em uma rede de duas camadas ocultas de 256 neurônios, com ReLU como função de ativação, e usando otimização Adam. Esse modelo forneceu uma acurácia de 98%. Para aprofundar a análise dessa rede, agora são úteis as funções `visualize_errors` e `show_confusion_matrix`, definidas na primeira seção deste caderno. A célula a seguir exibirá 50 erros cometidos pelo melhor classificador.

In [24]:
```
visualize_errors(model_e, data, max_errors=50)
```

Como visto na célula anterior, a maioria dos erros são de números que de fato, sobretudo manuscritos, possuem característica semelhantes e confundíveis mesmo para seres humanos, com destaque para os números 4 e 9. A variabilidade da caligrafia também é um fator contribuinte para tais confusões, mais notavelmente a terceira imagem, onde o dígito 2 está escrito em de forma bastante minimalista, o que o torna razoável classificá-lo como dígito 7.

In [25]:
```
show_confusion_matrix(model_e, data)
```


Out[25]:
```
array([[ 963,    1,    1,    0,    1,    0,    8,    2,    1,    3],
       [   0, 1124,    5,    2,    0,    1,    2,    0,    1,    0],
       [   1,    0, 1013,    6,    1,    0,    1,    5,    5,    0],
       [   0,    0,    3,  999,    0,    0,    0,    2,    2,    4],
       [   0,    0,    2,    1,  942,    1,    5,    0,    0,   31],
       [   1,    0,    0,   22,    2,  856,    5,    0,    4,    4],
       [   2,    3,    1,    1,    3,    3,  943,    0,    1,    1],
       [   0,    2,    9,    6,    0,    0,    0, 1003,    1,    7],
       [   0,    1,    8,   12,    1,    1,    2,    2,  940,    7],
       [   0,    3,    1,    5,    2,    4,    1,    1,    0,  992]])
```

Com a matriz de confusão gerada pela função `show_confusion_matrix`, é possível obter uma visualização mais ampla dos principais erros cometidos pelo classificador. Observa-se que o maior erro foi a classificação de "4" como "9", 16 vezes, o que é razoável, pois os dois dígitos possuem uma forma fechada no canto superior esquerdo, embora o "9" seja mais arredondado e o "4" com formas mais retas. Formas retas são levemente mais demoradas de se escrever que formas arredondadas, o que pode indicar que parte do erro é devido à características dos dados, ou seja, o "4" manuscrito é naturalmente mais parecido com o "9" do que o dígito. O "9" também é classificado como "9", 12 vezes, sendo o segundo erro mais comum do classificador. `model_e`, e também é razoável, pois a quina única distinção das formas é que o "9" possui uma forma fechada no canto superior esquerdo, e é relativamente fácil um "3" manuscrito parecer ter uma forma fechada na parte superior.