# NOIA 2/2024 - Trabalho 1

## Perceptron Multicamada no problema MNIST

### 1. Introdução e Preparação

Faremos uso do MLP para o problema de classificação de dígitos manuscritos MNIST. O código apresentado nestas instruções deve ser visto como sugestão, mas o trabalho deve necessariamente estar em python, na forma de um "notebook", e implementar o perceptron multicamada com as características dadas para resolver o problema dado.

Além das sugestões dadas aqui, observe o caderno de implementação do MLP no problema Fashion_MNIST que estudamos em sala.

Começamos com os módulos necessários:

```
import tensorflow as tf
from d2l import tensorflow as d2l
d2l.use_svg_display()
import pdb
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
```

Não é necessário desta vez usar um arquivo em disco, porque a base de dados MNIST está disponível como um dos conjuntos de dados do submódulo keras.datasets do módulo tensorflow.

```
class MNIST(d2l.DataModule):  #@save
    """The MNIST dataset."""
    def __init__(self, batch_size=64):
        super().__init__()
        self.save_hyperparameters()
        self.train, self.val = tf.keras.datasets.mnist.load_data()
```

Há 60000 exemplos de treinamento e 10000 de validação. A entrada, em cada exemplo, é uma matriz 28x28 de pixels. As imagens originais manuscritas foram pré-processadas (normalizadas em tamanho e centralizadas).

```
data = MNIST()
print(len(data.train[0]), len(data.val[0]))
print(data.train[0].shape)
```

```
60000 10000
(60000, 28, 28)
```

Abaixo um código, do d2l, para ler um minibatch de um dado tamanho. Em cada chamada, ele nos dá o tensor de entrada X e a classe desejada y. Ele também embaralha os dados no momento do treinamento. Finalmente, observe que ele divide os valores de pixels por 255, para obter um float entre 0 e 1.

```
@d2l.add_to_class(MNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    process = lambda X, y: (tf.expand_dims(X, axis=3) / 255,
                            tf.cast(y, dtype='int32'))
    resize_fn = lambda X, y: (tf.image.resize_with_pad(X,
*self.resize), y)
    shuffle_buf = len(data[0]) if train else 1
    return tf.data.Dataset.from_tensor_slices(process(*data)).batch(
        self.batch_size).shuffle(shuffle_buf)
```

Vamos usar isso com um minibatch e ver as dimensões

```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
(64, 28, 28, 1) <dtype: 'float32'> (64,) <dtype: 'int32'>
```

Você pode estranhar a princípio as dimensões de X. São 64 imagens 28x28, mas por que a dimensão unitária adicional? Lembre-se que, em geral, imagens têm 3 canais de cor. Estas imagens são em tons de cinza.

## 2. Modelos e Arquiteturas

Para a implementação, baseie-se na classe MLP definida no caderno que vimos em sala de aula. A primeira camada deve ser do tipo "Flatten" como no outro exemplo. Ela transforma a matriz 28x28 em um vetor unidimensional de tamanho 784. Neste trabalho, vocês farão algumas variações de arquitetura, funções de ativação e algoritmos de treinamento, indo das técnicas usadas nas primeiras aplicações do Perceptron nos anos 80/90 às atuais. Para cada uma delas, avalie a acurácia, sobre o arquivo de validação, do modelo treinado. Você pode usar a função `test_acc = d2l.evaluate_accuracy(model, data.val_dataloader())`.

Eu não estou sugerindo que cada modificação introduzida vai necessariamente melhorar o modelo em todas as ocasiões. Lembre-se também que um valor melhor em um único treinamento pode não ter significância estatística. Haverá bonificação se essas análises foram feitas para mais de uma rodada de treinamento (com valores de inicialização aleatória diferentes).

### 2a) Perceptron com uma camada escondida, função de ativação logística, função custo SSE e otimização por descida de gradiente.

Comecemos com um MLP com uma camada escondida de 128 neurônios. Usaremos sigmoides nas duas camadas (escondida e saída). Passe o parâmetro `activation='sigmoid'` na função que cria a camada. Defina erro quadrático como função de custo (parâmetro `loss_fn=tf.keras.losses.MeanSquaredError` para o construtor da classe MLP), e a

descida simples de gradiente como otimizador (parâmetro `optimizer=tf.keras.optimizers.SGD` para o construtor da classe MLP).

Estas eram opções comuns nos primeiros anos de uso do perceptron. Vamos avaliar a mudança de desempenho com redes maiores e parâmetros mais otimizados.

### 2b) Saída do tipo softmax, custo "entropia cruzada"

A simples logística na saída não representa corretamente o fato de que queremos uma decisão "one-hot", isto é, quando uma saída é alta, as outras devem ser baixas. Esta restrição é melhor representada pelo uso do softmax na saída. Use `activation=softmax` na última camada. Mantenha a logística nas demais camadas.

Esta mudança também nos aproxima de medidas de probabilidade, e neste caso o uso de um custo como entropia cruzada, em vez do simples erro de saída, também é um avanço significativo. Use `loss_fn=tf.keras.losses.SparseCategoricalCrossentropy`. Esta função de custo de entropia cruzada é usada quando a saída é o número da classe correta (como aqui), e não um vetor "one-hot". Retreine e reavalie.

### 2c) Otimizador Adam

O gradiente simples tem um inconveniente grave: sabemos que melhoramos o desempenho do classificador se acompanharmos o gradiente em um "pequeno passo", mas não é fácil determinar que passo deveria ser esse. E sabemos que ele deveria variar ao longo do treinamento. Várias propostas para uma adaptação da taxa de aprendizado foram feitas, mas a que acabou se tornando padrão é o chamado otimizador Adam (Kingma and Ba, 2014). O nome vem de "Estimativa adaptativa de momentos". Acompanhando a média e variância do gradiente entre atualizações, ele consegue adaptar a taxa de aprendizado, e torná-la diferente para cada parâmetro.

Use `optimizer=tf.keras.optimizers.Adam`. Retreine e reavalie.

### 2d) A função Relu

Um desenvolvimento um tanto surpreendente foi a descoberta de que uma simples função retificadora (0 se a soma é negativa, função identidade se é positiva) nos dá não-linearidade suficiente para o MLP. Use na camada escondida `activation=relu`. Retreine e reavalie.

### 2e) Redes maiores

Com esses desenvolvimentos, e a melhoria geral na capacidade de computação, aos poucos foi possível trabalhar com redes cada vez maiores. Experimente uma camada escondida com 256 neurônios, e também uma rede com duas camadas escondidas. Retreine e reavalie.

## 3. Visualização

Gostaríamos agora de observar alguns dos exemplos mal-classificados. Você pode se basear na função `data.visualize` do exemplo visto em sala, mas adaptações são necessárias. Em primeiro lugar, naquela ocasião usamos apenas os erros de um minibatch. Como o desempenho daquele classificador foi relativamente fraco, havia erros em todos os "minibatches". O MLP

(especialmente os maiores) deve ter um número pequeno de erros, e é possível que não haja nenhuma classificação errada em um dado minibatch. Isso produziria um erro no seu código.

Além disso, queremos talvez visualizar o erro não apenas no último minibatch. Faça um código que passe por todos os minibatches (por exemplo, `for X, y in data.val_dataloader()`) e identifique exemplos de classificações incorretas. Ou então, trabalhe diretamente com o objeto MNIST "data". Neste caso, lembre-se que data.train e data.val são ambas "duplas", contendo as entradas e as saídas de todos os exemplos. Usar o iterador como `data.val.dataloader()` torna-se essencial quando usarmos grandes bases de dados, e não pudermos trazer alocar tudo em memória de uma vez.

Há um outro problema na implementação feita em sala. `data.visualize` espera uma lista de imagens, e não vai funcionar com uma imagem única. Assim como pode haver minibatches sem erros, pode haver minibatches com um erro a visualizar apenas. Nestes casos talvez as funções `imshow` e `show` do módulo matplotlib.pyplot podem ser úteis.

Seja como for, mostre alguns padrões erradamente classificados (juntamente com os rótulos esperados e obtidos, como vimos no exemplo em sala de aula). Você acha que é razoável que um humano cometesse esse tipo de erro?

## 4. Matriz de confusão

Queremos agora observar a matriz de confusão do seu melhor classificador. A função confusion_matrix do módulo `sklearn.metrics(y,preds)` faz esse cálculo, mas temos de novo o problema que essa função espera receber todos as classes corretas (y) e todas as predições (preds) de uma vez. De novo, faça isso para cada minibatch usando o iterador `data.val_dataloader()` e depois combine-as corretamente, ou então trabalhe diretamente com as duplas no objeto data.

Obtenha a matriz de confusão (10x10). Observe o resultado. Quais são as "confusões" mais comuns para cada classe (ou seja, para cada classe i qual a classe j é a atribuição errada mais comum? Isso faz sentido?