Explicação das Etapas

1. Normalização dos Dados

No código, as imagens são carregadas do dataset MNIST e normalizadas para o intervalo de [0, 1] dividindo os valores dos pixels por 255.

python

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

A **normalização** é fundamental porque ajuda a rede neural a aprender de forma mais eficiente, pois os valores de entrada ficam em uma faixa padrão.

2. Camadas Convolucionais (Conv2D) e Pooling

O modelo usa camadas **convolucionais** para extrair características das imagens e **camadas de pooling** para reduzir a dimensionalidade dos dados, mantendo as características importantes. Isso ajuda a rede a generalizar melhor.

- **Conv2D**: Camada que aplica filtros para detectar padrões nas imagens (ex: bordas, formas).
- **MaxPooling2D**: Camada de pooling que reduz o tamanho das imagens para diminuir a carga computacional.

3. Função de Ativação Softmax

A camada final do modelo utiliza a função **Softmax**, que converte as saídas da rede em probabilidades para cada uma das 10 classes (dígitos de 0 a 9).

python

```
layers.Dense(10, activation='softmax')
```

A função **Softmax** é utilizada para problemas de classificação multiclasse, como neste caso, onde temos 10 categorias (dígitos). Ela transforma as saídas da rede para que somem 1, representando uma distribuição de probabilidade.

4. Função de Perda Cross-Entropy

A **cross-entropy** é usada como função de perda para problemas de classificação multiclasse.

python

```
loss='sparse_categorical_crossentropy'
```

A função de perda calcula a diferença entre as previsões da rede e as classes reais, penalizando previsões incorretas e ajudando a rede a melhorar.

5. Backpropagation e Algoritmo de Otimização Adam

O algoritmo de **backpropagation** ajusta os pesos da rede com base no erro da previsão. Para otimizar esse processo, usamos o **Adam** como algoritmo de otimização.

python

optimizer='adam'

Adam ajusta a taxa de aprendizado para cada peso da rede com base em gradientes passados, tornando o treinamento mais rápido e estável.

6. Dropout

A técnica de **Dropout** é usada para evitar o **overfitting** (sobreajuste) durante o treinamento, desativando aleatoriamente uma fração das unidades da rede.

python

layers.Dropout(0.2)

Isso impede que a rede se torne muito dependente de alguns neurônios, ajudando a generalizar melhor para dados não vistos.

7. Avaliação do Modelo

Após o treinamento, o modelo é avaliado nos dados de teste para verificar sua precisão.

python

test_loss, test_acc = model.evaluate(x_test, y_test)

A **precisão** (test_acc) é a porcentagem de classificações corretas, enquanto a **perda** (test_loss) mede o erro entre a previsão do modelo e os valores reais.

Resultados Esperados no Colab

Ao rodar o código no Google Colab, você verá o seguinte:

- **Acurácia de Treinamento**: A precisão do modelo em cada época.
- **Acurácia de Validação**: A precisão do modelo nos dados de validação (dados de teste não vistos durante o treinamento).
- **Loss**: A perda do modelo durante o treinamento e teste.
- **Gráfico**: O gráfico exibirá a acurácia do modelo em função das épocas (número de vezes que o modelo foi treinado com todos os dados).

O treinamento deve resultar em uma alta **precisão de teste**, próxima de 98-99%, dependendo das condições do treinamento.

O gráfico mostra como a **precisão** melhora com cada época, e a **perda** diminui ao longo do treinamento.