Universidade Federal Fluminense

TÓPICOS ESP. SIST. INFORMAÇÃO

Prof.ªLeila Weitzel

3. Treinamento de redes

Taxa de aprendizado

- **Taxa de aprendizado (learning rate):**
  - A taxa de aprendizado (learning rate) em uma rede neural é um hiperparâmetro que controla o tamanho dos ajustes feitos nos pesos da rede durante o treinamento.
  - Em outras palavras, ela determina o quão rápido ou devagar a rede neural aprende com os dados.

Taxa de aprendizado

- Uma taxa de aprendizado muito alta pode levar a oscilações, enquanto uma taxa muito baixa pode resultar em treinamento lento ou estagnado.
- Para entender melhor, imagine que você está descendo uma montanha em direção a um vale, e o objetivo é chegar ao ponto mais baixo.
- A taxa de aprendizado seria a sua velocidade de descida. Se você descer muito rápido, corre o risco de passar direto pelo ponto mais baixo e não conseguir parar.
- Por outro lado, se você descer muito devagar, pode levar muito tempo para chegar ao ponto mais baixo.

Taxa de aprendizado

- Na prática, uma taxa de aprendizado muito alta pode resultar em oscilações nos ajustes dos pesos, dificultando a convergência do modelo.
- A escolha da taxa de aprendizado adequada é crucial para o sucesso do treinamento da rede neural.
- Ela geralmente é um valor pequeno, como 0.1, 0.01, ou até mesmo menor, dependendo do problema e da arquitetura da rede.

Taxa de aprendizado

- Em muitos casos, é útil usar técnicas como "schedulding" da taxa de aprendizado, onde a taxa é reduzida ao longo do tempo para permitir uma busca mais refinada pelos melhores pesos.
  - **Decaimento exponencial**: Neste método, a taxa de aprendizado é reduzida multiplicando-a por um fator constante a cada determinado número de épocas.
  - **Decaimento linear**: Neste método, a taxa de aprendizado é reduzida subtraindo um valor constante dela a cada determinado número de épocas.

Taxa de aprendizado

- A taxa de aprendizado é chamada dentro da função compile

```
# Compile o modelo com a função de perda MAE
model.compile(optimizer=Adam(learning_rate=0.001), loss='mae')
```

Função de perda (loss)

- O objetivo do aprendizado de máquina e do aprendizado profundo é reduzir a diferença entre a saída prevista e a saída real, também é chamado de Função de Perda.
- Mede a discrepância entre as previsões feitas pelo modelo e os valores reais (rótulos) dos dados de treinamento.
- Ela quantifica o erro cometido pelo modelo e fornece um valor escalar que representa o “custo” associado a essa discrepância.

Função de perda (loss)

- **Exemplos de Funções de Perda:**
  - **Classificação**:
    - Para problemas de classificação, uma função de perda comum é a **verossimilhança logarítmica negativa** (ou **entropia cruzada**).
    - Ela compara as probabilidades previstas pelo modelo com as probabilidades reais (one-hot encoding) e penaliza erros.
  - **Regressão**:
    - Para problemas de regressão, a função de perda frequentemente é a **soma dos quadrados residuais**.
    - Ela mede a diferença entre as previsões contínuas do modelo e os valores reais.

Função de perda (loss)

- A escolha da função de perda depende do tipo de problema que está sendo abordado.

1. Mean Squared Error (MSE)

Descrição: Calcula a média dos quadrados das diferenças entre as previsões e os valores reais.

Uso: Comumente usada em problemas de regressão.

Funcionalidade: Penaliza grandes erros mais severamente, o que pode ser útil para modelos que precisam ser precisos em suas previsões.

2. Mean Absolute Error (MAE)

Descrição: Calcula a média das diferenças absolutas entre as previsões e os valores reais.

Uso: Também usada em problemas de regressão.

Funcionalidade: Penaliza todos os erros de forma linear, o que pode ser mais robusto a outliers do que o MSE.

3. Binary Crossentropy

Descrição: Mede a diferença entre duas distribuições de probabilidade para problemas de classificação binária.

Uso: Usada em problemas de classificação binária.

Funcionalidade: Penaliza previsões que estão longe das classes reais, ajudando o modelo a melhorar a precisão das previsões de classes binárias.

4. Categorical Crossentropy

Descrição: Similar à Binary Crossentropy, mas usada para problemas de classificação com múltiplas classes.

Uso: Usada em problemas de classificação multiclasse.

Funcionalidade: Penaliza previsões incorretas de forma proporcional à probabilidade prevista, ajudando a melhorar a precisão em problemas de classificação multiclasse.

5. Sparse Categorical Crossentropy

Descrição: Uma variação da Categorical Crossentropy que é mais eficiente para grandes conjuntos de dados com muitas classes.

Uso: Usada em problemas de classificação multiclasse onde as classes são representadas por inteiros.

Funcionalidade: Reduz a necessidade de converter rótulos de classe em vetores one-hot, economizando memória e tempo de computação.

6. Huber Loss

Descrição: Combina as vantagens do MSE e do MAE, sendo menos sensível a outliers do que o MSE.

Uso: Usada em problemas de regressão onde há outliers.

Funcionalidade: Penaliza erros quadráticos para pequenos erros e erros lineares para grandes erros, proporcionando um equilíbrio entre MSE e MAE.

7. Kullback-Leibler Divergence (KL Divergence)

Descrição: Mede a diferença entre duas distribuições de probabilidade.

Uso: Usada em problemas de aprendizado de máquina onde a distribuição de saída prevista deve ser comparada com uma distribuição alvo.

Funcionalidade: Ajuda a ajustar o modelo para que a distribuição prevista se aproxime da distribuição alvo.

Função de perda (loss)

- O hiperparâmetro Função de perda é chamada dentro da função compile

```
# Compile o modelo com a função de perda MAE
model.compile(optimizer=Adam(learning_rate=0.001), loss='mae')
```

Gradiente descendente

- é um algoritmo de otimização usado para minimizar a função de perda (loss function) em modelos de machine learning, incluindo redes neurais.
- Ele funciona ajustando iterativamente os pesos da rede em direção ao mínimo da função de perda, seguindo o gradiente negativo da função.
- É o principal algoritmo usado para atualizar os pesos durante o treinamento.
- Ele permite que a rede aprenda a partir dos dados, ajustando os pesos para reduzir a diferença entre as previsões do modelo e os valores reais.

Regularização

- Antes de falarmos em regularização devemos ter em mente o que é overfitting e underfitting.
- Overfitting e underfitting são problemas comuns em redes neurais e em outros modelos de aprendizado de máquina

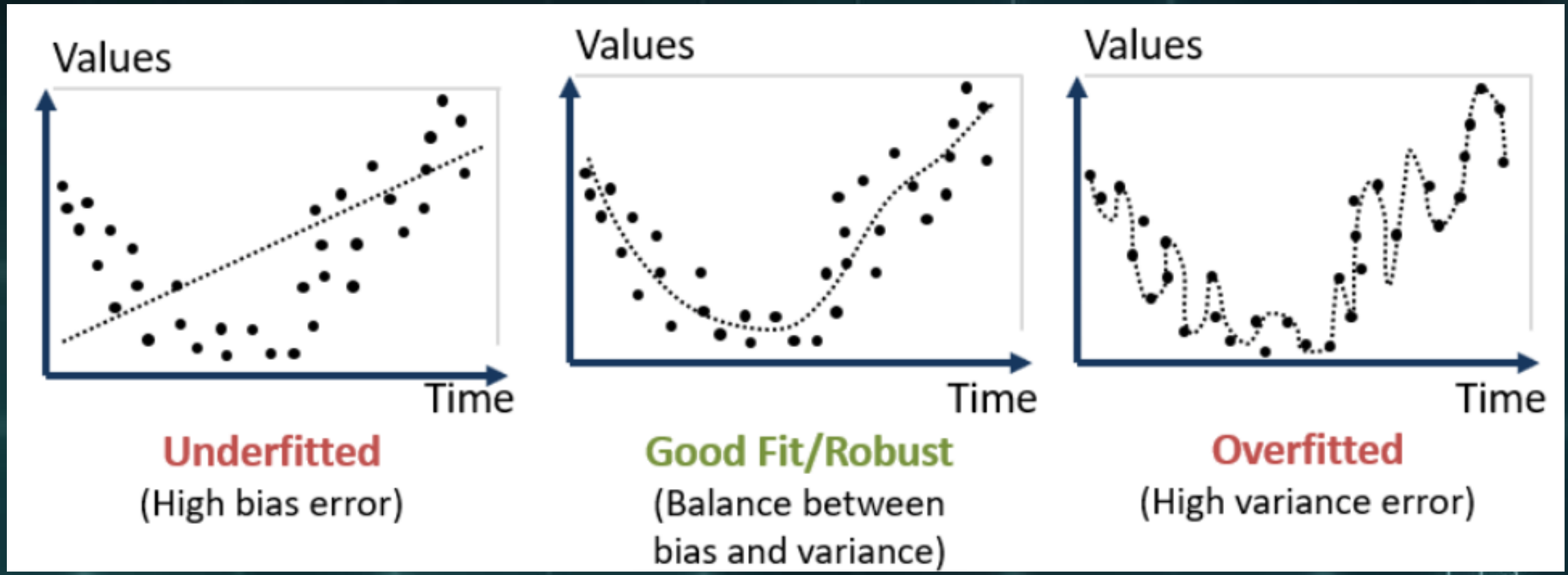
Overfitting

- **Definição:** Ocorre quando o modelo se ajusta excessivamente aos dados de treinamento, capturando ruídos e detalhes específicos que não são relevantes para a generalização.
  - Exemplo:
    - Imagine um modelo de classificação de gatos e cachorros treinado com poucos exemplos de cada classe. Se o modelo memorizar esses exemplos em vez de aprender padrões gerais, ele pode ter um desempenho ruim em novos dados.
- **Soluções:**
  - Regularização: Adicione termos de regularização (como L1 ou L2) à função de perda para penalizar pesos grandes.
  - Redução da complexidade do modelo: Use menos camadas ou neurônios.
  - Mais dados de treinamento: Aumente o tamanho do conjunto de treinamento.

Underfitting

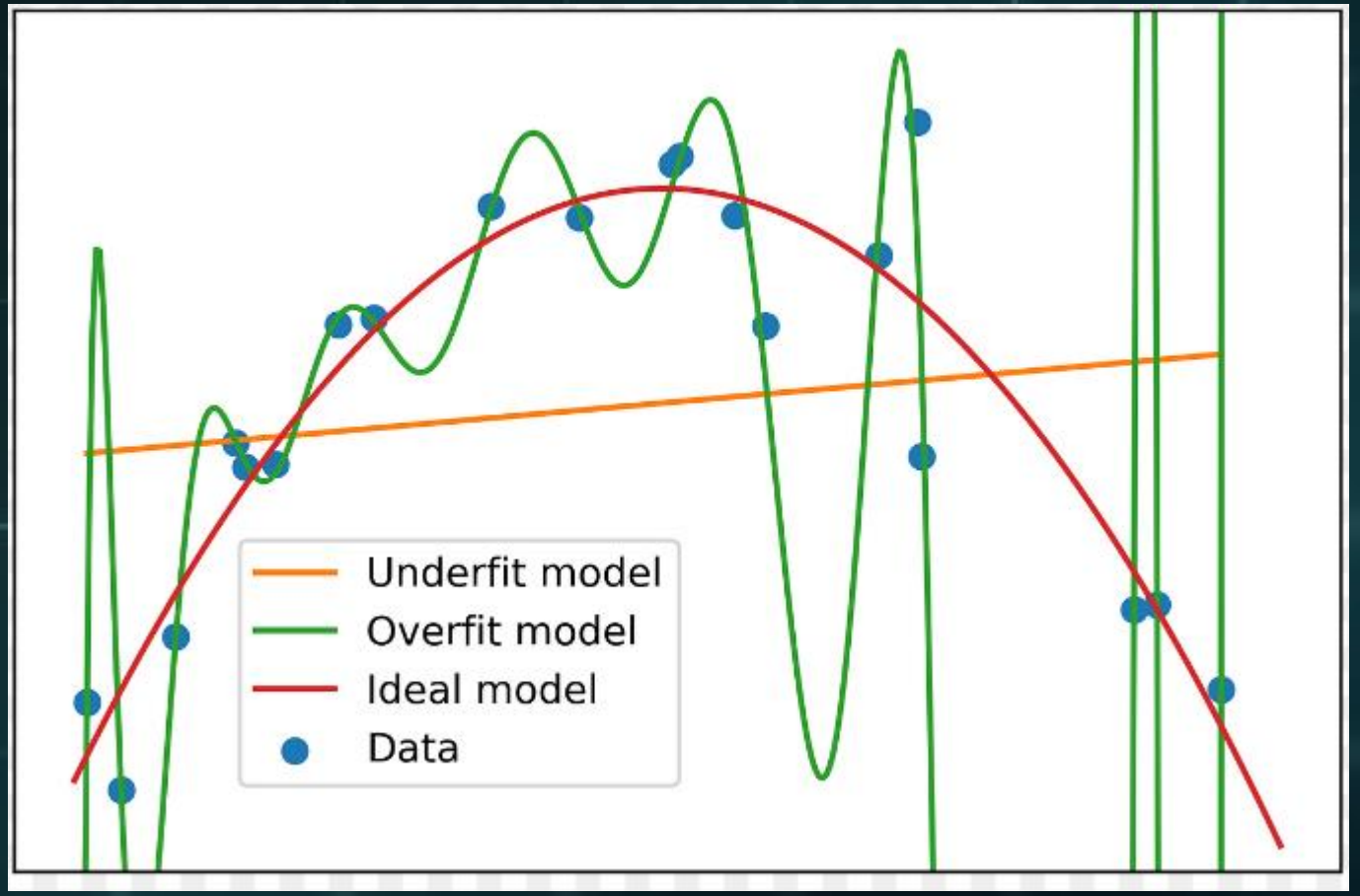
- **Definição:** Ocorre quando o modelo é muito simples para capturar os padrões nos dados de treinamento.
  - **Exemplo**:
    - Um modelo linear tentando prever a altura de uma pessoa com base apenas na idade terá um desempenho insatisfatório, pois a relação não é linear.
- **Soluções:**
  - **Aumento da complexidade do modelo**: Adicione mais camadas ou neurônios.
  - **Mais recursos**: Use mais atributos relevantes nos dados.
  - **Treinamento mais longo**: Aumente o número de épocas de treinamento.

Over e under fitting



Over e under fitting

- Resultado de um treinamento da rede.



Regularização

- A regularização em redes neurais é uma técnica utilizada para evitar **overfitting**, que ocorre quando o modelo se ajusta muito bem aos dados de treinamento, mas tem um desempenho ruim em dados não vistos.
- A regularização adiciona termos à função de perda durante o treinamento para penalizar pesos grandes, incentivando assim que a rede neural aprenda padrões mais simples e generalize melhor para novos dados.

Regularização

- **Tipos:**
  - **L1 Regularization (Lasso):** Adiciona o valor absoluto dos pesos à função de perda. Isso leva à sparseness, ou seja, muitos pesos se tornam exatamente zero, o que pode ajudar na seleção de features.
  - **L2 Regularization (Ridge):** Adiciona o quadrado dos pesos à função de perda. Isso penaliza pesos grandes sem torná-los exatamente zero, promovendo a suavização dos pesos.
  - **Dropout:** Durante o treinamento, aleatoriamente "desliga" (seta para zero) um percentual dos neurônios em uma camada. Isso ajuda a prevenir o overfitting, pois força a rede a não depender muito de nenhum neurônio específico.
  - **Data Augmentation:** Aumenta artificialmente o tamanho do conjunto de dados de treinamento aplicando transformações como rotações, zooms, e inversões nas imagens. Isso ajuda a melhorar a capacidade de generalização da rede.
  - **Batch Normalization:** Normaliza as ativações de cada camada, reduzindo a covariância entre as features e ajudando a regularizar o modelo.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers

# Criando o modelo com regularização L2
model = Sequential([
    Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.01), input_shape=(784,)),
    Dense(64, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    Dense(10, activation='softmax')
])

# Compilando o modelo
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Treinando o modelo
model.fit(x_train, y_train, epochs=10, validation_data=(x_val, y_val))
```

- Neste exemplo, kernel_regularizer=regularizers.l2(0.01) é usado para aplicar regularização L2 com um fator de penalização de 0.01 às camadas densas da rede neural. Este é apenas um exemplo e a escolha da técnica de regularização depende do problema específico e da arquitetura da rede.

Regularização

- **Tipos que veremos**
  - –Dropout
  - –Batch Normalization

Dropout

- é uma técnica de regularização usada em redes neurais durante o treinamento para reduzir o overfitting.
- A ideia principal do dropout é aleatoriamente "desligar" (ou "zerar") um percentual de unidades (neurônios) em uma camada durante cada passagem de treinamento.
- Isso impede que unidades específicas se tornem muito dependentes umas das outras, o que ajuda a melhorar a generalização do modelo.
- O objetivo do dropout é forçar a rede neural a aprender representações mais robustas e distribuídas dos dados, em vez de depender fortemente de um conjunto específico de unidades para fazer previsões.

```
# Crie um modelo sequencial
model = Sequential()

# Adicione uma camada densa com 64 neurônios e ativação ReLU
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))

# Adicione uma camada Dropout com taxa de 20%
model.add(Dropout(0.2))

# Adicione outra camada densa com 32 neurônios e ativação ReLU
model.add(Dense(32, activation='relu'))

# Adicione uma camada de saída com 1 neurônio (para um problema de regressão, por exemplo)
model.add(Dense(1, activation='linear'))
```

- Neste exemplo, o dropout é aplicado após a primeira camada densa com uma taxa de 20% (0.2), o que significa que durante cada época de treinamento, 20% das unidades daquela camada serão "desligadas".

Batch_normalization

- É uma técnica usada para treinar redes neurais profundas que visa normalizar as entradas para cada camada em cada mini-lote durante o treinamento.
- Isso ajuda a estabilizar o processo de aprendizado e reduzir o número de épocas necessárias para treinar redes profundas.

Batch_normalization

- **Problema de Treinamento em Redes Profundas:**
  - Treinar redes neurais profundas com muitas camadas é desafiador.
  - Um dos problemas é que a distribuição das entradas para camadas profundas pode mudar após cada mini-lote quando os pesos são atualizados.
  - Isso é chamado de “deslocamento interno covariante”.

Batch_normalization

- **Benefícios do Batch Normalization:**
  - Estabilização: Ajuda a estabilizar o processo de treinamento, permitindo que os gradientes fluam mais suavemente.
  - Redução de Épocas: Pode reduzir significativamente o número de épocas necessárias para treinar a rede.
  - Regularização: Também atua como uma forma leve de regularização, melhorando a generalização

```
# Crie um modelo sequencial
model = Sequential()

# Adicione uma camada densa com 32 neurônios
model.add(Dense(32, input_shape=(64,)))

# Adicione uma camada BatchNormalization
model.add(BatchNormalization())

# Adicione outra camada densa com 32 neurônios
model.add(Dense(32))

# Adicione uma camada de saída com ativação softmax
model.add(Dense(10, activation='softmax'))
```

- Neste exemplo:
- Criamos um modelo com duas camadas densas e uma camada de saída.
- A camada BatchNormalization é adicionada entre as camadas densas para normalizar as ativações.

Batch_normalization

- A normalização em lote (Batch Normalization) tem parâmetros que podem ser ajustados durante o treinamento da rede neural.
  - **Média e Variância Móveis:**
    - A normalização em lote calcula a média e a variância das ativações para cada mini-lote durante o treinamento.
    - Essas médias e variâncias são usadas para normalizar as ativações.
    - Os parâmetros relacionados a isso são:
      - momentum: Controla a taxa de atualização das médias e variâncias móveis.
      - epsilon: Um valor pequeno adicionado à variância para evitar divisão por zero.

Batch_normalization

- **Gamma e Beta:**
  - Após a normalização, as ativações são escalonadas e deslocadas usando os parâmetros gamma e beta.
  - Esses parâmetros permitem que o modelo aprenda a escala e o deslocamento ideais para cada camada.
  - Eles são inicializados como 1 e 0, respectivamente, mas são ajustados durante o treinamento.
- **Camadas de Ativação:**
  - A normalização em lote pode ser aplicada antes ou depois das camadas de ativação.
  - Alguns frameworks (como o Keras) permitem escolher onde aplicar a normalização.

```
# Crie um modelo sequencial
model = Sequential()

# Adicione uma camada densa com 64 neurônios
model.add(Dense(64, input_shape=(input_dim,)))

# Adicione uma camada BatchNormalization com momentum ajustado
model.add(BatchNormalization(momentum=0.9))  # Experimente diferentes valores para o momentum

# Adicione outra camada densa com 32 neurônios
model.add(Dense(32))

# Adicione uma camada de saída com ativação softmax
model.add(Dense(10, activation='softmax'))

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

- Uso:
  - Adicione a camada BatchNormalization após as camadas densas ou convolucionais.
  - Ajuste os parâmetros conforme necessário para o seu problema específico.

Momentum

- Valor de Momentum:
  - O valor de **0.9** é uma escolha comum para o momentum.
  - Significa que, a cada iteração, as médias e variâncias móveis são atualizadas em 90% da direção da média e variância calculadas no mini-lote atual.
  - Isso ajuda a suavizar as atualizações e estabilizar o processo de treinamento.
- Ajuste do Momentum:
  - O valor do momentum pode ser ajustado conforme necessário para o seu problema específico.
  - Valores mais altos (próximos a 1) tornam as atualizações mais suaves, mas podem atrasar a adaptação a mudanças nos dados.
  - Valores mais baixos (próximos a 0) permitem atualizações mais rápidas, mas podem ser sensíveis a ruídos nos dados.

```
# Adicione uma camada BatchNormalization com momentum ajustado
model.add(BatchNormalization(momentum=0.9))  # Experimente diferentes valores para o momentum
```


THANK YOU
FOR LISTENING

See You
Next time