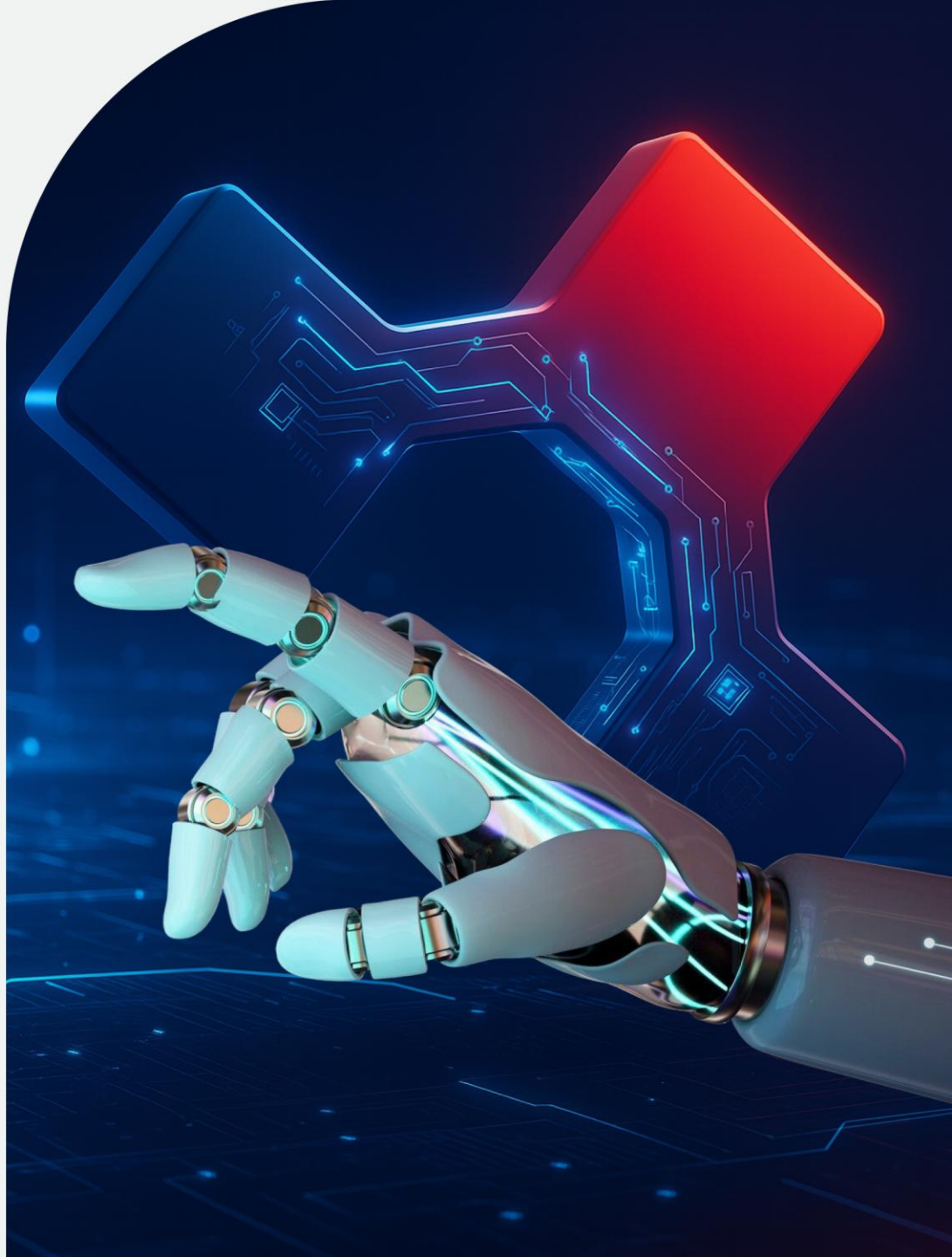
NCIA

Núcleo de Capacitação em Inteligência Artificial



NCIA | FOXCONN FPF tech

Using the Subclassing API to Build Dynamic Models

Tanto a API sequencial quanto a API funcional são declarativas. Isso significa que você primeiro declara quais camadas quer usar e como elas se conectam, e só depois começa a alimentar o modelo com dados para treinamento ou inferência.

Essa abordagem tem várias vantagens: o modelo pode ser facilmente salvo, copiado e compartilhado, sua estrutura pode ser visualizada e analisada, e o framework consegue inferir dimensões e tipos, permitindo que erros sejam detectados antes mesmo de qualquer dado passar pelo modelo. O modelo é essencialmente um grafo estático de camadas, o que facilita a depuração.

Using the Subclassing API to Build Dynamic Models

Tanto a API sequencial quanto a API funcional são declarativas. Você começa declarando quais camadas quer usar e como elas se conectam, e só depois alimenta o modelo com dados para treinamento ou inferência.

Essa abordagem tem muitas vantagens: o modelo pode ser facilmente salvo, copiado e compartilhado, sua estrutura pode ser visualizada e analisada, e o framework consegue inferir formas e checar tipos, permitindo detectar erros antes mesmo de qualquer dado passar pelo modelo. O modelo é um grafo estático de camadas, facilitando bastante a depuração.

Using the Subclassing API to Build Dynamic Models

Na API de subclassing, você cria modelos herdando a classe Model do Keras. As camadas necessárias são definidas no construtor (__init__), e os cálculos e fluxos de dados são implementados no método call().

Essa abordagem permite total liberdade, possibilitando lógica dinâmica, múltiplas saídas e tratamento de entradas com formatos diferentes. É ideal para modelos que não podem ser representados como um grafo estático simples.

Using the Subclassing API to Build Dynamic Models

O modelo WideAndDeepModel combina duas entradas: uma wide e uma deep. A entrada wide passa por normalização e é mantida para concatenação, enquanto a entrada deep também é normalizada e depois processada por duas camadas densas escondidas.

Após o processamento, as saídas deep e wide são concatenadas para gerar a saída principal, e a saída auxiliar recebe apenas o processamento da deep.

O modelo é, portanto, flexível, permitindo múltiplas saídas, lógica dinâmica e diferentes formas de entrada.

Using the Subclassing API to Build Dynamic Models

```
class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)  # needed to support naming the model
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units, activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units, activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)
        concat = tf.keras.layers.concatenate([norm_wide, hidden2])
        output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return output, aux_output

model = WideAndDeepModel(30, activation="relu", name="my_cool_model")
```

Using the Subclassing API to Build Dynamic Models

Na API de subclassing, a grande diferença em relação à API funcional é que as camadas são criadas no construtor (__init__) e só são aplicadas no método call(), que define o fluxo de dados do modelo.

Não precisamos criar objetos Input, pois o argumento inputs do call() fornece todas as entradas necessárias. Depois de criar a instância do modelo, podemos compilá-lo, adaptar as camadas de normalização, treinar, avaliar e fazer previsões, exatamente como na API funcional.

Essa separação deixa o código mais organizado e facilita a reutilização de camadas em diferentes fluxos.

Using the Subclassing API to Build Dynamic Models

O método call() é onde a magia da API de subclassing acontece. Você pode colocar qualquer operação de TensorFlow dentro dele, incluindo loops, condicionais e cálculos complexos.

Essa flexibilidade é ideal para pesquisadores e quem quer experimentar novas ideias, permitindo criar modelos que seriam difíceis ou impossíveis de representar com APIs declarativas.

Em resumo, o call() transforma seu modelo em algo dinâmico, adaptável e altamente personalizável, sem limites fixos na estrutura ou no fluxo de dados.

Using the Subclassing API to Build Dynamic Models

Apesar de poderosa, a API de subclassing tem algumas limitações importantes:

- A arquitetura fica escondida dentro do método call(), então o Keras não consegue inspecionar facilmente como as camadas estão conectadas.
- Não é possível clonar o modelo com tf.keras.models.clone_model().
- O método summary() mostra apenas a lista de camadas, sem detalhar suas conexões.
- O framework não verifica tipos ou dimensões antecipadamente, o que aumenta o risco de erros.

Por isso, a menos que você precise de flexibilidade máxima, a API sequencial ou funcional continua sendo mais prática e segura.

Using the Subclassing API to Build Dynamic Models

Após treinar e avaliar sua rede neural, é essencial salvar o modelo para reutilização futura.

Salvar o modelo preserva tanto a arquitetura quanto os pesos, permitindo recuperar e continuar usando o modelo sem precisar treinar tudo novamente. Essa prática é importante tanto em projetos de pesquisa quanto em aplicações de produção.

Além disso, modelos salvos podem ser compartilhados com outros pesquisadores ou equipes, facilitando a colaboração.

Saving and Restoring a Model

Salvar um modelo treinado em Keras é extremamente simples. Basta usar o método save() na instância do modelo, informando o nome do arquivo ou diretório e o formato desejado.

```
model.save("my_keras_model", save_format="tf")
```

Ao definir save_format="tf", o Keras salva o modelo no formato SavedModel do TensorFlow, que é um diretório contendo vários arquivos e subdiretórios.

Saving and Restoring a Model

O diretório gerado pelo save() contém:

- saved_model.pb: contém a arquitetura e a lógica do modelo como um grafo de computação serializado, permitindo usar o modelo em produção sem precisar do código-fonte.
- keras_metadata.pb: informações extras necessárias para o Keras.
- variables/: contém todos os valores dos parâmetros, como pesos, vieses, estatísticas de normalização e parâmetros do otimizador. Se o modelo for grande, os arquivos podem ser múltiplos.
- assets/: pode conter arquivos extras, como exemplos de dados, nomes de features ou classes, mas geralmente está vazio por padrão.

Como o otimizador também é salvo, você pode continuar o treinamento após carregar o modelo.

Saving and Restoring a Model

Para carregar o modelo salvo e fazer previsões, basta usar load_model():

```
model = tf.keras.models.load_model("my_keras_model")
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

Depois disso, você pode avaliar ou fazer previsões com o modelo carregado, assim como faria logo após o treino.

Saving and Restoring a Model

Além de salvar o modelo completo, é possível salvar apenas os pesos usando save_weights() e carregá-los com load_weights().

Isso inclui pesos das conexões, vieses, estatísticas de pré-processamento e estado do otimizador. Os arquivos são normalmente gerados em formatos como my_weights.data-00004-of-00052 e um arquivo de índice my_weights.index.

Salvar apenas os pesos é mais rápido e ocupa menos espaço em disco, sendo ideal para checkpoints durante o treino, especialmente em modelos grandes que levam horas ou dias para treinar.

Saving and Restoring a Model

Se você está treinando um modelo grande, é fundamental salvar checkpoints regularmente para evitar perda de progresso em caso de falhas no computador.

Para isso, use callbacks, que permitem indicar ao método fit() quando e como salvar os pesos ou o modelo completo.

Dessa forma, você garante que o treinamento pode ser retomado de forma segura, sem precisar começar do zero.

Using Callbacks

O método fit() do Keras aceita o argumento callbacks, que é uma lista de objetos chamados em momentos específicos do treinamento: antes e depois de todo o treino, antes e depois de cada época, e até antes e depois de cada batch.

Um exemplo clássico é o ModelCheckpoint, que salva checkpoints do modelo em intervalos regulares, por padrão no final de cada época.

Using Callbacks

O método fit() do Keras aceita o argumento callbacks, que é uma lista de objetos chamados em momentos específicos do treinamento: antes e depois de todo o treino, antes e depois de cada época, e até antes e depois de cada batch.

Um exemplo clássico é o ModelCheckpoint, que salva checkpoints do modelo em intervalos regulares, por padrão no final de cada época.

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
                                                   save_weights_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

Using Callbacks

Ao usar um conjunto de validação durante o treino, podemos definir save_best_only=True no ModelCheckpoint.

Dessa forma, o modelo será salvo apenas quando o desempenho na validação for melhor do que qualquer desempenho anterior, evitando salvar modelos inferiores ou com overfitting.

Isso é uma forma de implementar early stopping implícito, garantindo que ao final do treino você tenha o melhor modelo da validação, mesmo sem interromper o treinamento.

Using Callbacks

O método fit() do Keras aceita o argumento callbacks, que é uma lista de objetos chamados em momentos específicos do treinamento: antes e depois de todo o treino, antes e depois de cada época, e até antes e depois de cada batch.

Um exemplo clássico é o ModelCheckpoint, que salva checkpoints do modelo em intervalos regulares, por padrão no final de cada época.

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
                                                   save_weights_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

Using Callbacks

Outra forma de parar o treinamento automaticamente é usar o EarlyStopping. Ele interrompe o treino quando não há progresso no conjunto de validação por um número definido de épocas (patience).

Se configurado com restore_best_weights=True, ele retorna ao modelo com melhor desempenho ao final do treino. Podemos combinar ModelCheckpoint e EarlyStopping para:

- Salvar checkpoints regulares, garantindo recuperação em caso de falha no computador.
- Interromper o treino cedo quando não há mais progresso, evitando desperdício de tempo e overfitting.

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                     restore_best_weights=True)
history = model.fit([...], callbacks=[checkpoint_cb, early_stopping_cb])
```

Using Callbacks

Ao usar EarlyStopping, podemos definir um número grande de épocas, pois o treino será interrompido automaticamente quando não houver progresso.

É importante apenas garantir que a taxa de aprendizado não seja muito baixa, para que o modelo não continue fazendo progresso muito lento até o final.

O EarlyStopping armazena as melhores pesos em memória RAM e os restaura automaticamente no final do treino.

Using Callbacks

Você também pode criar seus próprios callbacks personalizados, implementando métodos como:

- on_train_begin(), on_train_end()
- on_epoch_begin(), on_epoch_end()
- on_batch_begin(), on_batch_end()

Por exemplo, este callback exibe a relação entre a loss de validação e a loss de treino a cada época, ajudando a identificar overfitting:

```
class PrintValTrainRatioCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        ratio = logs["val_loss"] / logs["loss"]
        print(f"Epoch={epoch}, val/train={ratio:.2f}")
```

Using Callbacks

Callbacks não servem apenas para treino. Também podem ser usados durante:

- Avaliação: implementar on_test_begin(), on_test_end(), on_test_batch_begin() ou on_test_batch_end() que são chamados pelo evaluate().
- Previsão: implementar on_predict_begin(), on_predict_end(), on_predict_batch_begin() ou on_predict_batch_end() que são chamados pelo predict().

Isso permite usar callbacks para debug, monitoramento e métricas personalizadas mesmo fora do treino.

Using Callbacks

Uma ferramenta essencial para quem usa Keras é o TensorBoard, que permite:

- Visualizar métricas de treino e validação
- Monitorar perda, acurácia e aprendizado ao longo do tempo
- Observar gráficos de rede, histogramas de pesos e distribuições de ativações

TensorBoard é extremamente útil para acompanhar o treinamento e depurar modelos complexos.

Using TensorBoard for Visualization

TensorBoard é uma poderosa ferramenta interativa de visualização usada para acompanhar o treinamento de redes neurais.

Com ele, é possível:

- Visualizar curvas de aprendizado e comparar métricas entre diferentes execuções;
- Explorar o grafo computacional e estatísticas de treinamento;
- Analisar projeções em 3D de pesos aprendidos e dados multidimensionais;
- Monitorar o desempenho da rede, identificando gargalos e uso de recursos.

TensorBoard é instalado automaticamente com o TensorFlow.
Entretanto, para visualizar dados de *profiling*, é necessário instalar o plugin adicional.
Se estiver usando o Google Colab, execute:

```
%pip install -q -U tensorboard-plugin-profile
```

Using TensorBoard for Visualization

- Para usar o TensorBoard, é preciso configurar o programa para gravar os dados que serão exibidos — os chamados *event files*.

  Cada arquivo contém registros binários (*summaries*) monitorados pelo servidor do TensorBoard, que atualiza as visualizações automaticamente. A prática recomendada é criar uma pasta raiz (por exemplo, my_logs) e, a cada nova execução, gerar um subdiretório com data e hora, para manter os resultados organizados:

```
from pathlib import Path
from time import strftime

def get_run_logdir(root_logdir="my_logs"):
    return Path(root_logdir) / strftime("run_%Y_%m_%d_%H_%M_%S")

run_logdir = get_run_logdir()  # e.g., my_logs/run_2022_08_01_17_25_59
```

Using TensorBoard for Visualization

O Keras fornece um *callback* pronto que facilita o uso do TensorBoard. Ele cria automaticamente os diretórios de log, escreve os *event files* e mede as métricas de treino e validação (por exemplo, *MSE* e *RMSE*).

```
tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir,
                                                profile_batch=(100, 200))
history = model.fit([...], callbacks=[tensorboard_cb])
```

Nesse exemplo, o perfilamento ocorre entre os *batches* 100 e 200, pois o início do treino costuma ter ajustes iniciais (“aquecimento”) que podem distorcer a medição.

Using TensorBoard for Visualization

Ao modificar hiperparâmetros como a taxa de aprendizado e rodar novamente o modelo, novos diretórios são criados automaticamente, resultando em uma estrutura como:

```
my_logs
├── run_2022_08_01_17_25_59
│   ├── train
│   │   ├── events.out.tfevents.1659331561.my_host_name.42042.0.v2
│   │   ├── events.out.tfevents.1659331562.my_host_name.profile-empty
│   │   └── plugins
│   │       └── profile
│   │           └── 2022_08_01_17_26_02
│   │               ├── my_host_name.input_pipeline.pb
│   │               └── [...]
│   └── validation
│       └── events.out.tfevents.1659331562.my_host_name.42042.1.v2
└── run_2022_08_01_17_31_12
    └── [...]
```

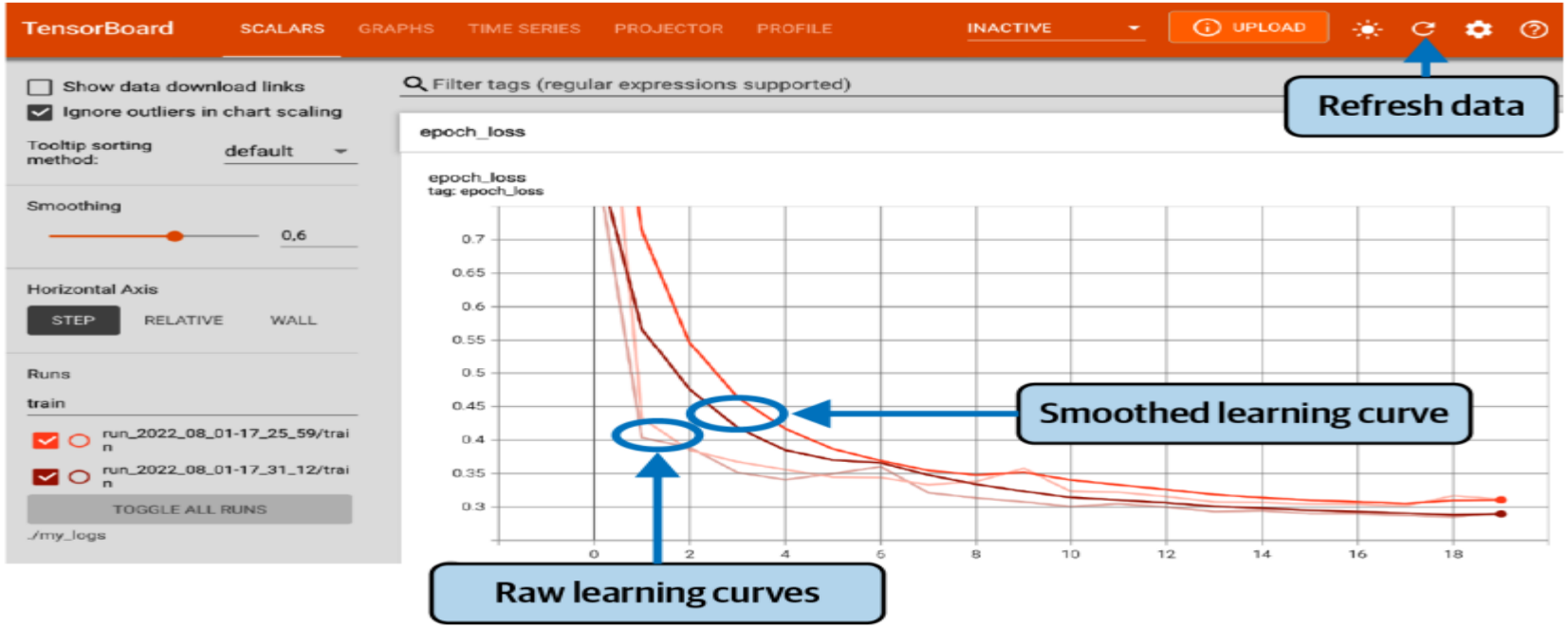
Using TensorBoard for Visualization

Use a extensão do Jupyter (pré-instalada no Colab) para iniciar o servidor e exibir a interface dentro do notebook. O servidor usa a primeira porta TCP disponível ≥ 6006 por padrão.

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs
```

Using TensorBoard for Visualization

Na aba SCALARS selecione os runs desejados e visualize métricas (p.ex. loss por época). Ao comparar dois runs você verá como mudanças de hiperparâmetros (ex.: learning rate) afetam a velocidade de convergência.



Using TensorBoard for Visualization

Além da aba **SCALARS**, o TensorBoard oferece outras visualizações importantes:

- **GRAPHS:** mostra o grafo completo de computação do modelo — útil para entender e depurar a arquitetura.
- **PROJECTOR:** exibe embeddings ou pesos em 3D, permitindo observar como o modelo organiza os dados em seu espaço interno.
- **PROFILE:** apresenta traces de desempenho, ajudando a identificar gargalos e otimizar o treinamento.

Use o botão **refresh (⟳)** para atualizar as métricas e ⚙ para ativar o **auto-refresh** durante o treinamento.

Using TensorBoard for Visualization

Para registrar conteúdo customizado (escalars, histogramas, imagens, áudio, texto), use tf.summary.create_file_writer() e o contexto writer.as_default() para emitir summaries que aparecem nas abas correspondentes do TensorBoard.

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(str(test_logdir))
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)

        data = (np.random.randn(100) + 2) * step / 100  # gets larger
        tf.summary.histogram("my_hist", data, buckets=50, step=step)

        images = np.random.rand(2, 32, 32, 3) * step / 1000  # gets brighter
        tf.summary.image("my_images", images, step=step)

        texts = ["The step is " + str(step), "Its square is " + str(step ** 2)]
        tf.summary.text("my_text", texts, step=step)

        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

Using TensorBoard for Visualization

Após executar e atualizar o TensorBoard, tabs como **IMAGES**, **AUDIO**, **DISTRIBUTIONS / HISTOGRAMS** e **TEXT** aparecerão. Use sliders para ver imagens/áudio em diferentes passos de treinamento. TensorBoard é útil até fora do contexto estrito de deep learning.

Using TensorBoard for Visualization

Você aprendeu sobre as origens das redes neurais, MLPs, como construir MLPs com a API Sequencial, e arquiteturas complexas com a API Funcional e Subclassing (Wide & Deep, múltiplas entradas/saídas). Aprendeu a salvar/recuperar modelos, usar callbacks (checkpoints, early stopping), e como usar o TensorBoard para monitorar e visualizar experimentos. A seguir: seleção de hiperparâmetros (número de camadas, neurônios, learning rate, etc).

Fine-Tuning Neural Network Hyperparameters

As redes neurais oferecem grande flexibilidade, mas isso também é uma desvantagem: há muitos hiperparâmetros a ajustar — número de camadas e neurônios, funções de ativação, inicialização de pesos, otimizador, taxa de aprendizado, tamanho do batch etc. Saber qual combinação é ideal para a tarefa é um desafio central no treinamento de redes neurais.

Fine-Tuning Neural Network Hyperparameters

Uma opção é converter o modelo Keras em um estimador Scikit-Learn e usar GridSearchCV ou RandomizedSearchCV. No entanto, uma abordagem mais moderna e integrada é usar o Keras Tuner, uma biblioteca projetada para ajuste automático de hiperparâmetros em modelos Keras. O Keras Tuner é flexível, personalizável e tem integração nativa com o TensorBoard.

Fine-Tuning Neural Network Hyperparameters

Se você está rodando localmente, o Keras Tuner já está instalado.
No Google Colab, basta executar:

```
%pip install -q -U keras-tuner
```

Depois, importa-se o pacote:

```
import keras_tuner as kt
```

Em seguida, cria-se uma função de construção do modelo (build_model) que define os hiperparâmetros e retorna um modelo Keras compilado.

Fine-Tuning Neural Network Hyperparameters

A função build_model(hp) usa um objeto kt.HyperParameters para definir os intervalos de busca.
Exemplo de hiperparâmetros:

- n_hidden: número de camadas ocultas (0–8)
- n_neurons: número de neurônios por camada (16–256)
- learning_rate: taxa de aprendizado (1e−4 a 1e−2, escala log)
- optimizer: escolha entre **SGD** e **Adam**

O modelo é criado, compilado e retornado conforme esses valores.

```python
import keras_tuner as kt

def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,
                             sampling="log")
    optimizer = hp.Choice("optimizer", values=["sgd", "adam"])
    if optimizer == "sgd":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    model.add(tf.keras.layers.Dense(10, activation="softmax"))
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

Fine-Tuning Neural Network Hyperparameters

Para iniciar a busca, criamos um RandomSearch tuner que:

- Gera combinações aleatórias de hiperparâmetros;
- Treina o modelo por um número fixo de épocas;
- Armazena os resultados e métricas de cada tentativa.

A função search() executa o processo.

```
random_search_tuner = kt.RandomSearch(
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)
random_search_tuner.search(X_train, y_train, epochs=10,
                           validation_data=(X_valid, y_valid))
```

Fine-Tuning Neural Network Hyperparameters

Durante a busca:

- O tuner chama build_model() uma vez para registrar os hiperparâmetros.
- Em seguida, executa diversos trials, amostrando combinações aleatórias dentro dos intervalos definidos.
- Cada modelo é treinado e avaliado, e o resultado é salvo.
- O parâmetro objective="val_accuracy" define que o objetivo é maximizar a acurácia de validação.

Fine-Tuning Neural Network Hyperparameters

Após a busca, podemos obter os melhores modelos e hiperparâmetros:

```
top3_models = random_search_tuner.get_best_models(num_models=3)
best_model = top3_models[0]
```

Ou apenas os melhores hiperparâmetros:

```
>>> top3_params = random_search_tuner.get_best_hyperparameters(num_trials=3)
>>> top3_params[0].values  # best hyperparameter values
{'n_hidden': 5,
 'n_neurons': 70,
 'learning_rate': 0.00041268008323824807,
 'optimizer': 'adam'}
```

Fine-Tuning Neural Network Hyperparameters

Após a busca, podemos obter os melhores modelos e hiperparâmetros:

```
top3_models = random_search_tuner.get_best_models(num_models=3)
best_model = top3_models[0]
```

Ou apenas os melhores hiperparâmetros:

```
>>> top3_params = random_search_tuner.get_best_hyperparameters(num_trials=3)
>>> top3_params[0].values  # best hyperparameter values
{'n_hidden': 5,
 'n_neurons': 70,
 'learning_rate': 0.00041268008323824807,
 'optimizer': 'adam'}
```

Fine-Tuning Neural Network Hyperparameters

Cada tuner é guiado por um oracle, responsável por decidir qual será o próximo conjunto de hiperparâmetros testado. No caso do RandomSearch, ele usa uma busca aleatória, mas existem outros tipos de oráculos (por exemplo, BayesianOptimization).
É possível inspecionar o melhor trial diretamente:

```
>>> best_trial = random_search_tuner.oracle.get_best_trials(num_trials=1)[0]
>>> best_trial.summary()
Trial summary
Hyperparameters:
n_hidden: 5
n_neurons: 70
learning_rate: 0.00041268008323824807
optimizer: adam
Score: 0.8736000061035156
```

Fine-Tuning Neural Network Hyperparameters

Após encontrar o melhor conjunto de hiperparâmetros, você pode:

- Continuar o treinamento do melhor modelo com todos os dados de treino;
- Avaliar no conjunto de teste;
- Salvar e implantar o modelo em produção.

```
>>> best_trial.metrics.get_last_value("val_accuracy")
0.8736000061035156
```

Fine-Tuning Neural Network Hyperparameters

Além dos hiperparâmetros do modelo, podemos ajustar parâmetros de pré-processamento e até argumentos do model.fit() (como o *batch size*).

Para isso, usamos uma técnica diferente: em vez de uma função build_model(), criamos uma subclasse de kt.HyperModel com dois métodos:

- build(): constrói e compila o modelo (igual ao build_model() anterior);
- fit(): recebe o modelo, dados e hiperparâmetros, e decide como ajustar o treinamento (por exemplo, se normaliza os dados ou não).

Fine-Tuning Neural Network Hyperparameters

Neste exemplo, o hiperparâmetro booleano "normalize" decide se os dados devem ser normalizados antes do treinamento.

```python
class MyClassificationHyperModel(kt.HyperModel):
    def build(self, hp):
        return build_model(hp)

    def fit(self, hp, model, X, y, **kwargs):
        if hp.Boolean("normalize"):
            norm_layer = tf.keras.layers.Normalization()
            X = norm_layer(X)
        return model.fit(X, y, **kwargs)
```

Fine-Tuning Neural Network Hyperparameters

Podemos usar essa classe diretamente em um tuner Hyperband, que otimiza a busca de forma mais eficiente do que a aleatória:

```
hyperband_tuner = kt.Hyperband(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_epochs=10, factor=3, hyperband_iterations=2,
    overwrite=True, directory="my_fashion_mnist", project_name="hyperband")
```

Fine-Tuning Neural Network Hyperparameters

O Hyperband é semelhante ao HalvingRandomSearchCV:

- Treina muitos modelos por poucas épocas;
- Elimina os piores e mantém apenas os melhores (ex: top 1/3);
- Repete o processo até sobrar um modelo vencedor.

O argumento max_epochs define o número máximo de épocas do melhor modelo, e hyperband_iterations controla quantas vezes repetir o processo.

Fine-Tuning Neural Network Hyperparameters

- root_logdir: diretório raiz dos logs. O Hyperband cria subdiretórios separados para cada teste.
- Callback do TensorBoard (tensorboard_cb): registra métricas, perdas e perfis em tempo real para cada teste.
- EarlyStopping callback (early_stopping_cb): interrompido o treinamento de um julgamento se a métrica de validação não melhorar por 2 temporadas consecutivas.
- hyperband_tuner.search(...): Treina múltiplos modelos com diferentes configurações de hiperparâmetros.
- Avaliar cada teste usando dados de validação (validation_data).

```
root_logdir = Path(hyperband_tuner.project_dir) / "tensorboard"
tensorboard_cb = tf.keras.callbacks.TensorBoard(root_logdir)
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=2)
hyperband_tuner.search(X_train, y_train, epochs=10,
                       validation_data=(X_valid, y_valid),
                       callbacks=[early_stopping_cb, tensorboard_cb])
```

Fine-Tuning Neural Network Hyperparameters

Ao abrir o TensorBoard no diretório my_fashion_mnist/hyperband/tensorboard, você verá:

- O progresso dos *trials* em tempo real;
- A aba **HPARAMS**, com as combinações de hiperparâmetros e suas métricas;
- Três modos de visualização:
  1. **Tabela** com os resultados;
  2. **Paralel coordinates** para filtrar bons resultados;
  3. **Scatterplot matrix** para relações entre parâmetros.

Dica: filtre por validation.epoch_accuracy para destacar os melhores modelos.

Fine-Tuning Neural Network Hyperparameters

O Keras Tuner também oferece o BayesianOptimization, que aprende quais regiões do espaço de hiperparâmetros são mais promissoras. Ele usa um processo gaussiano para estimar o desempenho esperado e ajustar a busca gradualmente.

```
bayesian_opt_tuner = kt.BayesianOptimization(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_trials=10, alpha=1e-4, beta=2.6,
    overwrite=True, directory="my_fashion_mnist", project_name="bayesian_opt")
bayesian_opt_tuner.search([...])
```

Os parâmetros:

- alpha: nível de ruído nas métricas;
- beta: equilíbrio entre exploração e aproveitamento das melhores regiões.

Fine-Tuning Neural Network Hyperparameters

A busca de hiperparâmetros ainda é um campo de pesquisa ativo.
Outras estratégias incluem:

- Algoritmos evolutivos para otimizar populações de modelos (DeepMind, 2017);
- Google AutoML / Vertex AI, que explora tanto hiperparâmetros quanto arquiteturas;
- Deep Neuroevolution (Uber, 2017), que substitui o gradiente descendente por evolução neural.

Mesmo com todas essas técnicas, compreender os hiperparâmetros continua essencial para restringir o espaço de busca e prototipar rapidamente.

Number of Hidden Layers

Para muitos problemas, é possível começar com apenas uma camada oculta e obter resultados razoáveis. Um MLP com uma camada oculta pode, teoricamente, modelar funções muito complexas, desde que tenha neurônios suficientes. No entanto, para problemas mais complexos, redes profundas (deep networks) são muito mais eficientes em termos de parâmetros do que redes rasas (shallow networks). Elas conseguem representar funções complexas usando exponencialmente menos neurônios, alcançando melhor desempenho com a mesma quantidade de dados de treinamento.

Number of Hidden Layers

Imagine que você precise desenhar uma floresta sem copiar e colar nada. Você teria que desenhar cada árvore, ramo e folha individualmente, o que seria muito demorado. Se você puder desenhar uma folha, copiá-la para formar um ramo, copiar o ramo para formar uma árvore e finalmente copiar a árvore para formar a floresta, o trabalho será muito mais rápido. Isso ilustra como dados do mundo real são hierárquicos, e redes profundas aproveitam isso automaticamente:

- Camadas ocultas inferiores: modelam estruturas de baixo nível (ex.: segmentos de linha).
- Camadas intermediárias: combinam essas estruturas para formar elementos de nível médio (ex.: quadrados, círculos).
- Camadas superiores e camada de saída: combinam elementos intermediários para formar estruturas de alto nível (ex.: rostos).

Number of Hidden Layers

A arquitetura hierárquica ajuda as redes profundas a:

- Convergirem mais rápido para boas soluções.
- Generalizarem melhor para novos conjuntos de dados.

Exemplo de Transfer Learning: se você treinou um modelo para reconhecer rostos e agora quer treinar outro para reconhecer penteados, pode reutilizar as camadas inferiores do primeiro modelo. Dessa forma, o novo modelo não precisa aprender do zero estruturas de baixo nível, apenas padrões de alto nível (como penteados).

Number of Hidden Layers

Para muitos problemas, uma ou duas camadas ocultas são suficientes. Exemplos:

- MNIST: uma camada oculta com algumas centenas de neurônios já alcança >97% de acurácia.
- MNIST: duas camadas ocultas com o mesmo número total de neurônios alcançam >98%, sem aumentar significativamente o tempo de treinamento.

Problemas mais complexos exigem aumentar gradualmente o número de camadas até começar a ocorrer overfitting. Tarefas muito complexas, como classificação de grandes imagens ou reconhecimento de fala, podem precisar de dezenas (ou até centenas) de camadas, normalmente não totalmente conectadas, e uma grande quantidade de dados. Treinar essas redes do zero é raro; é mais comum reutilizar partes de redes pré-treinadas de última geração, tornando o treinamento mais rápido e eficiente.

Number of Neurons per Hidden Layer

O número de neurônios nas camadas de entrada e saída depende do tipo de entrada e saída exigido pela tarefa. Por exemplo, para o MNIST, cada imagem tem 28 × 28 pixels, resultando em 784 neurônios de entrada, e há 10 classes de dígitos, portanto 10 neurônios de saída.

Number of Neurons per Hidden Layer

Antes, era comum dimensionar as camadas ocultas em formato de pirâmide, com cada camada sucessiva tendo menos neurônios. A ideia era que muitos recursos de baixo nível se combinassem em poucos recursos de alto nível. Por exemplo, uma rede típica para MNIST poderia ter três camadas ocultas: 300, 200 e 100 neurônios, respectivamente.

Hoje, essa prática caiu em desuso. Usar o mesmo número de neurônios em todas as camadas ocultas costuma ter desempenho igual ou melhor, além de reduzir o número de hiperparâmetros a serem ajustados. Em alguns casos, entretanto, a primeira camada oculta maior que as demais ainda pode ser útil.

Number of Neurons per Hidden Layer

Assim como o número de camadas, você pode:

- Aumentar gradualmente o número de neurônios até começar a ocorrer overfitting.
- Criar uma rede um pouco maior do que o necessário e usar técnicas de regularização, como early stopping, para evitar overfitting.
- Essa abordagem foi apelidada de “stretch pants” por Vincent Vanhoucke: use “calças grandes” que se ajustam ao tamanho certo, evitando camadas gargalo que limitem a capacidade de representação da rede.

Importante: Se uma camada tem poucos neurônios, pode perder informações importantes. Por exemplo, uma camada com 2 neurônios recebendo dados 3D não consegue preservar toda a informação. Mesmo que o resto da rede seja grande e poderoso, essa informação perdida não pode ser recuperada.

Learning Rate, Batch Size, and Other Hyperparameters

O número de camadas e neurônios é importante, mas não são os únicos hiperparâmetros que influenciam o desempenho de uma MLP.

Outros parâmetros, como **taxa de aprendizado, otimizador, tamanho do batch, função de ativação** e **número de iterações**, também afetam fortemente a performance e a velocidade de convergência.

Learning Rate, Batch Size, and Other Hyperparameters

A taxa de aprendizado é o hiperparâmetro mais importante.
Uma forma prática de escolher um bom valor é começar com uma taxa muito pequena (por exemplo, 10^{-5} (e aumentá-la gradualmente até um valor alto (como 10), multiplicando por um fator constante a cada iteração.

Ao plotar a função de perda em relação à taxa (em escala logarítmica), observa-se que a perda diminui até certo ponto e depois aumenta bruscamente.
O valor ideal é aproximadamente 10 vezes menor que o ponto onde a perda começa a subir.

Depois disso, reinicializa-se o modelo e treina-se normalmente com essa taxa ótima.
Figura: seria uma curva em forma de “U”, mostrando a perda mínima antes da divergência.

Learning Rate, Batch Size, and Other Hyperparameters

Usar um bom otimizador é essencial para melhorar a eficiência do treinamento. Algoritmos mais avançados, como **RMSProp**, **Adam** e **Nadam**, ajustam dinamicamente a taxa de aprendizado, acelerando a convergência e melhorando o desempenho. Esses otimizadores serão discutidos em mais detalhes nos capítulos seguintes.

Learning Rate, Batch Size, and Other Hyperparameters

O tamanho do batch influencia diretamente o tempo de treinamento e a qualidade do modelo. Batches grandes são mais eficientes em GPUs, pois processam mais exemplos por segundo. Entretanto, podem causar instabilidades e pior generalização, especialmente no início do treino. Pesquisas indicam que batches pequenos (2 a 32 amostras) muitas vezes produzem modelos mais robustos, enquanto outros estudos mostram que batches grandes (até 8192) funcionam bem se combinados com aquecimento da taxa de aprendizado (*learning rate warmup*). Na prática, recomenda-se testar ambos: começar com batch grande e reduzir se houver instabilidade ou baixo desempenho.

Learning Rate, Batch Size, and Other Hyperparameters

A função de ativação ReLU é o padrão mais indicado para camadas ocultas.
Na camada de saída, a escolha depende da tarefa — por exemplo, softmax para classificação ou linear para regressão. O número de iterações normalmente não precisa ser ajustado manualmente. Utiliza-se early stopping, interrompendo o treinamento automaticamente quando não há mais melhora na validação.

Learning Rate, Batch Size, and Other Hyperparameters

Encerramos a introdução às redes neurais artificiais e à sua implementação com Keras.
Nos próximos capítulos, estudaremos como treinar redes profundas de forma eficiente, criar modelos personalizados com a API de baixo nível do TensorFlow e usar o tf.data para carregar e pré-processar dados.
Também exploraremos arquiteturas avançadas como:

- CNNs (para imagens),
- RNNs e Transformers (para sequências e texto),
- Autoencoders (para aprendizado de representações),
- e GANs (para geração de dados).

FPFtech