Lab 5 - BCC406/PCC177

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Regressão Logística e Rede Neural

Prof. Eduardo e Prof. Pedro

Objetivos:

- *Overfitting*
- Regularização

Data da entrega : 14/11

- Complete o código (marcado com ToDo) e quando requisitado, escreva textos diretamente nos notebooks. Onde tiver *None*, substitua pelo seu código.
- Execute todo notebook e salve tudo em um PDF **nomeado** como "NomeSobrenome-LabX.pdf"
- Envie o PDF e o .ipynb via google FORM.

Overfitting e *Underfitting*

Importando os pacotes e funções auxiliares

```
import numpy as np
import pathlib
import shutil
import tempfile

from  IPython import display
from matplotlib import pyplot as plt

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import regularizers
```

```
prop_cycle = plt.rcParams['axes.prop_cycle']
COLOR_CYCLE = prop_cycle.by_key()['color']

def _smooth(values, std):
  """Smooths a list of values by convolving with a Gaussian distribution.
  Assumes equal spacing.
  Args:
    values: A 1D array of values to smooth.
    std: The standard deviation of the Gaussian distribution. The units are
      array elements.
  Returns:
    The smoothed array.
  """
  width = std * 4
  x = np.linspace(-width, width, min(2 * width + 1, len(values)))
  kernel = np.exp(-(x / 5)**2)

  values = np.array(values)
  weights = np.ones_like(values)

  smoothed_values = np.convolve(values, kernel, mode='same')
  smoothed_weights = np.convolve(weights, kernel, mode='same')

  return smoothed_values / smoothed_weights

class HistoryPlotter(object):
  """A class for plotting a named set of Keras-histories.
  The class maintains colors for each key from plot to plot.
  """

  def __init__(self, metric=None, smoothing_std=None):
    self.color_table = {}
    self.metric = metric
    self.smoothing_std = smoothing_std

  def plot(self, histories, metric=None, smoothing_std=None):
    """Plots a {name: history} dictionary of Keras histories.
    Colors are assigned to the name-key, and maintained from call to call.
    Training metrics are shown as a solid line, validation metrics dashed.
    Args:
      histories: {name: history} a dictionary of Keras histories.
      metric: which metric to plot from all the histories.
      smoothing_std: the standard deviation of the smoothing kernel applied
        before plotting. The units are in array-indices.
    """
    if metric is None:
      metric = self.metric
    if smoothing_std is None:
      smoothing_std = self.smoothing_std

    for name, history in histories.items():
      # Remember name->color associations.
      if name in self.color_table:
        color = self.color_table[name]
      else:
        color = COLOR_CYCLE[len(self.color_table) % len(COLOR_CYCLE)]
        self.color_table[name] = color

      train_value = history.history[metric]
      val_value = history.history['val_' + metric]
      if smoothing_std is not None:
        train_value = _smooth(train_value, std=smoothing_std)
        val_value = _smooth(val_value, std=smoothing_std)

      plt.plot(
          history.epoch,
          train_value,
          color=color,
          label=name.title() + ' Train')
      plt.plot(
          history.epoch,
          val_value,
          '--',
          label=name.title() + ' Val',
          color=color)

    plt.xlabel('Epochs')
    plt.ylabel(metric.replace('_', ' ').title())
    plt.legend()

    plt.xlim(
        [0, max([history.epoch[-1] for name, history in histories.items()])])
```

```
    plt.grid(True)

class EpochDots(tf.keras.callbacks.Callback):
  """A simple callback that prints a "." every epoch, with occasional reports.
  Args:
    report_every: How many epochs between full reports
    dot_every: How many epochs between dots.
  """

  def __init__(self, report_every=100, dot_every=1):
    self.report_every = report_every
    self.dot_every = dot_every

  def on_epoch_end(self, epoch, logs):
    if epoch % self.report_every == 0:
      print()
      print('Epoch: {:d}, '.format(epoch), end='')
      for name, value in sorted(logs.items()):
        print('{}:{:0.4f}'.format(name, value), end=',  ')
      print()

    if epoch % self.dot_every == 0:
      print('.', end='', flush=True)
```

Importando os dados e algumas constantes

Algumas constantes também podem ajudar:

```
FEATURES = 28
BATCH_SIZE = 500
N_VALIDATION = int(1e3)
N_TRAIN = int(1e4)
```

Iremos trabalhar com o conjunto de dados de Higgs. O objetivo não é fazer física de partículas ou se preocupar com detalhes do conjunto de dados. O importante de entender é que ele contém 11.000.000 amostras, cada um com 28 características (`FEATURES`) e um rótulo de classe binária.

```
gz = tf.keras.utils.get_file('HIGGS.csv.gz', 'http://mlphysics.ics.uci.edu/data/higgs/HIGGS.csv.gz')
```

A classe `tf.data.experimental.CsvDataset` pode ser usada para ler registros csv diretamente de um arquivo gzip sem etapa de descompactação intermediária.

```
ds = tf.data.experimental.CsvDataset(gz,[float(),]*(FEATURES+1), compression_type="GZIP")
```

Essa classe de leitor de csv retorna uma lista de escalares para cada registro. A função a seguir reempacota essa lista de escalares em um par (feature_vector, label).

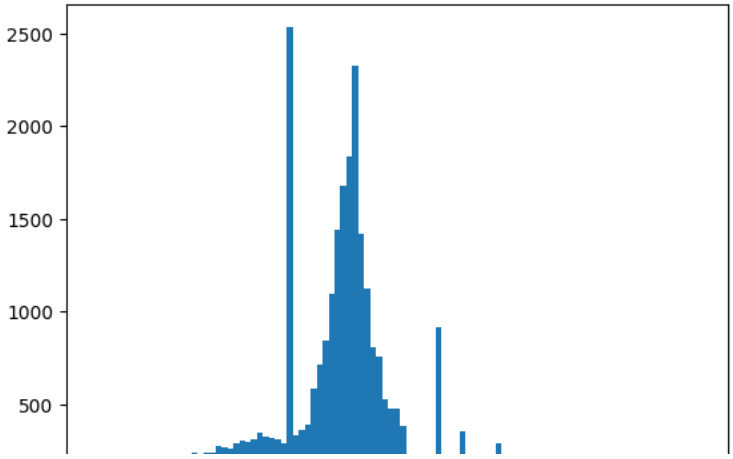
O TensorFlow é mais eficiente ao operar em grandes lotes de dados. Portanto, em vez de reempacotar cada linha individualmente, criaremos um novo conjunto de Dataset que receba lotes de 10.000 exemplos, aplique a função `pack_row` a cada lote e, em seguida, divida os lotes em registros individuais:

```
def pack_row(*row):
  label = row[0]
  features = tf.stack(row[1:],1)
  return features, label

packed_ds = ds.batch(10000).map(pack_row).unbatch()
```

Analisando os dados lidos

```
for features,label in packed_ds.batch(1000).take(1):
  print(features[0])
  plt.hist(features.numpy().flatten(), bins = 101)
```

```
tf.Tensor(
[ 0.8692932  -0.6350818   0.22569026  0.32747006 -0.6899932   0.75420225
 -0.24857314 -1.0920639   0.          1.3749921  -0.6536742   0.9303491
  1.1074361   1.1389043  -1.5781983  -1.0469854   0.          0.65792954
 -0.01045457 -0.04576717  3.1019614   1.35376     0.9795631   0.97807616
  0.92000484  0.72165745  0.98875093  0.87667835], shape=(28,), dtype=float32)
```



11.000.000 de amostras é um número elevado de amostras para treino. Para essa prática, usaremos as 1.000 amostras para validação e as próximas 10.000 para treinamento.

Usaremos os métodos `Dataset.skip` e `Dataset.take` para facilitar esse processo.

```
validate_ds = packed_ds.take(N_VALIDATION).cache()
train_ds = packed_ds.skip(N_VALIDATION).take(N_TRAIN).cache()
```

Overfitting (sobreajuste)

A maneira mais simples de evitar o *overfitting* é começar com um modelo pequeno: um modelo com um pequeno número de parâmetros (que é determinado pelo número de camadas e o número de unidades por camada). No aprendizado profundo, o número de parâmetros que podem ser aprendidos em um modelo é geralmente chamado de "capacidade" do modelo.

Intuitivamente, um modelo com mais parâmetros terá mais "capacidade de memorização" e, portanto, poderá aprender facilmente um mapeamento perfeito do tipo dicionário entre amostras de treinamento e seus alvos, um mapeamento sem nenhum poder de **generalização**, mas isso seria inútil ao fazer previsões em dados inéditos.

Sempre tenha isso em mente: os modelos de aprendizado profundo tendem a ser bons em se ajustar aos dados de treinamento, mas o verdadeiro desafio é a **generalização**, não o ajuste.

Por outro lado, se a rede tiver recursos de memorização limitados, ela não conseguirá aprender o mapeamento com tanta facilidade. Para minimizar sua perda, ele terá que aprender representações compactadas que tenham mais poder preditivo. Ao mesmo tempo, se você tornar seu modelo muito pequeno, ele terá dificuldade em se ajustar aos dados de treinamento. Há um equilíbrio entre "capacidade demais" e "capacidade de menos".

Infelizmente, não existe uma fórmula mágica para determinar o tamanho certo ou a arquitetura do seu modelo (em termos de número de camadas ou o tamanho certo para cada camada). Você terá que experimentar usando uma série de arquiteturas diferentes.

Para encontrar um tamanho de modelo apropriado, é melhor começar com relativamente poucas camadas e parâmetros e, em seguida, começar a aumentar o tamanho das camadas ou adicionar novas camadas até ver retornos decrescentes na perda de validação.

Comece com um modelo simples usando apenas `layers.Dense` como linha de base, depois crie versões maiores e compare-as.

Anotação pessoal:

"Intuitivamente, um modelo com mais parâmetros terá mais "capacidade de memorização" e, portanto, poderá aprender facilmente um mapeamento perfeito do tipo dicionário entre amostras de treinamento e seus alvos, um mapeamento sem nenhum poder de generalização, mas isso seria inútil ao fazer previsões em dados inéditos. Por outro lado, se a rede tiver recursos de memorização limitados, ela não conseguirá aprender o mapeamento com tanta facilidade. Para minimizar sua perda, ele terá que aprender representações compactadas que tenham mais poder preditivo. "

Q: É possível treinar um modelo pequeno, para obter essas representações compactas, e então alimentar esse modelo a um modelo maior que tenha mais capacidade de armazenamento? quais seriam as consequências?

R: Sim, é possível treinar um modelo menor para aprender representações compactas e, em seguida, alimentar essas representações a um modelo maior com mais capacidade de armazenamento. Esse processo é conhecido como **"transferência de aprendizado"** ou **"pré-**

treinamento".

A ideia é que o modelo menor, devido às suas limitações de capacidade, seja forçado a aprender características mais gerais e representações mais compactas das entradas. Essas representações podem ser utilizadas como entradas para um modelo maior, que pode ter mais capacidade de memorização. Isso pode oferecer algumas vantagens:

1. **Generalização melhorada:** As representações compactas aprendidas pelo modelo menor podem capturar padrões mais universais e generalizáveis, o que pode ajudar o modelo maior a generalizar melhor para novos dados.
2. **Treinamento mais eficiente:** O treinamento do modelo menor pode ser mais rápido e eficiente, já que ele lida com uma representação mais simplificada do problema.
3. **Menor risco de overfitting:** Como o modelo menor é menos propenso a memorizar os dados de treinamento de maneira exata, há menos risco de overfitting nessa etapa. O modelo maior, ao receber representações mais generalizáveis, pode ser treinado para se ajustar aos dados específicos sem capturar ruídos irrelevantes.

No entanto, há algumas considerações a ter em mente:

1. **Compatibilidade de representações:** É importante garantir que as representações aprendidas pelo modelo menor sejam compatíveis com o problema que o modelo maior está tentando resolver. Caso contrário, a transferência de aprendizado pode não ser eficaz.
2. **Ajuste fino:** Após a pré-treinamento, geralmente é necessário realizar um ajuste fino (fine-tuning) do modelo maior nas tarefas específicas que você deseja resolver. Isso envolve treinar o modelo maior com os dados específicos da tarefa para otimizar ainda mais suas capacidades.
3. **Escolha apropriada dos modelos:** Nem sempre um modelo maior é necessário ou benéfico. Em alguns casos, um modelo menor com técnicas de regularização pode ser suficiente para aprender representações úteis sem a necessidade de transferência de aprendizado.

Em resumo, a transferência de aprendizado de um modelo menor para um modelo maior pode ser uma estratégia eficaz.

Q: É possível treinar um modelo pequeno, para obter essas representações compactas, e então alimentar esse modelo a um modelo maior que tenha mais capacidade de armazenamento, e então repetir esse processo? quais seriam as consequências?

R: Sim, é possível realizar um processo iterativo de treinamento, onde um modelo menor é treinado para aprender representações compactas e, em seguida, essas representações são alimentadas a um modelo maior, repetindo o processo. Esse tipo de abordagem é conhecido como **treinamento em camadas** ou **treinamento em cascata**. No entanto, há algumas considerações e consequências a serem observadas:

1. **Perda de Informação:** A cada camada do processo iterativo, pode haver uma perda gradual de informações específicas aos dados originais, já que o modelo menor tende a capturar características mais gerais e abstratas. Isso pode resultar em uma representação final que pode não preservar todos os detalhes dos dados de treinamento.
2. **Diminuição da Interpretabilidade:** À medida que você aumenta a complexidade do modelo, pode se tornar mais difícil interpretar e entender o que está sendo aprendido em cada camada. Modelos mais complexos podem aprender representações altamente não lineares e abstratas, o que pode dificultar a interpretação dos resultados.
3. **Custo Computacional:** Treinar modelos maiores é geralmente mais computacionalmente intensivo. Se o objetivo é ter um modelo grande o suficiente para lidar com dados mais complexos, você pode precisar de recursos computacionais significativos.
4. **Risco de Overfitting:** À medida que você aumenta a capacidade do modelo, há um risco potencial de overfitting aos dados de treinamento, especialmente se não houver cuidado na regularização e controle do treinamento.
5. **Ajuste Fino Necessário:** Assim como na transferência de aprendizado, pode ser necessário realizar ajuste fino do modelo maior nas tarefas específicas para as quais você deseja usá-lo.
6. **Projeto Cuidadoso:** A eficácia desse processo dependerá muito do projeto cuidadoso das arquiteturas dos modelos, do tamanho dos conjuntos de dados e da seleção de hiperparâmetros.

Em resumo, treinar um modelo pequeno e transferir suas representações para um modelo maior pode ser uma estratégia interessante.

Q: "Perda de Informação: A cada camada do processo iterativo, pode haver uma perda gradual de informações específicas aos dados originais, já que o modelo menor tende a capturar características mais gerais e abstratas. Isso pode resultar em uma representação final que pode não preservar todos os detalhes dos dados de treinamento." Seria possível mitigar esse (e outros) downsides do aprendizado transferido re-introduzindo de alguma forma ao modelo mais complexo os dados originais em conjunto às representações obtidas pelo modelo menor?

R: Sim, é possível mitigar a perda de informação reintroduzindo os dados originais em conjunto com as representações obtidas pelo modelo menor. Essa abordagem é conhecida como "**conexão residual**" ou "**skip connection**" e tem sido utilizada para ajudar a mitigar problemas de degradação do desempenho em redes mais profundas.

A ideia básica é adicionar conexões diretas (residuais) entre camadas, permitindo que as informações originais fluam mais facilmente através da rede. Isso ajuda a preservar informações importantes durante o treinamento, mesmo quando camadas adicionais estão sendo empilhadas. A arquitetura de rede resultante é muitas vezes chamada de "residual network" ou "ResNet".

A aplicação dessa ideia ao seu contexto pode envolver a concatenação ou combinação de representações do modelo menor com os dados originais antes de alimentar o modelo maior. Essa abordagem pode ajudar a preservar detalhes importantes dos dados originais, ao mesmo tempo em que permite que o modelo maior aprenda representações mais complexas.

No entanto, é importante ajustar cuidadosamente como as informações são combinadas, pois a simples concatenação pode aumentar a dimensionalidade dos dados e exigir considerações adicionais. Além disso, você pode querer considerar estratégias adicionais, como técnicas de regularização e ajuste fino, para controlar o treinamento do modelo maior.

Em resumo, a reintrodução dos dados originais em conjunto com as representações aprendidas pelo modelo menor pode ser uma estratégia eficaz para mitigar a perda de informação durante o aprendizado transferido em camadas.

Q: Mas é possível fazer isso com os dados originais, ao invés de apenas conexões residuais às camadas anteriores?

R: Sim, é possível reintroduzir os dados originais diretamente em um ponto específico da arquitetura, ao invés de apenas adicionar conexões residuais entre camadas anteriores. Essa abordagem é conhecida como "**conexão direta**" ou "**skip connection**" diretamente aos dados originais.

Ao fazer isso, você efetivamente cria um caminho direto para os dados originais fluírem através da rede e se combinarem com as representações aprendidas pelo modelo menor. Essa técnica tem sido usada em várias arquiteturas de redes neurais e pode ajudar a preservar informações importantes durante o treinamento do modelo maior.

Por exemplo, ao treinar um modelo menor para aprender representações compactas, você pode então alimentar essas representações diretamente para um ponto intermediário do modelo maior e, ao mesmo tempo, fornecer os dados originais a esse ponto. A combinação dessas informações pode ajudar a manter detalhes importantes dos dados originais.

A implementação prática dessa abordagem pode variar dependendo da arquitetura específica da rede neural que você está usando. Em muitos casos, isso envolve a concatenação ou combinação direta dos dados originais com as saídas de uma camada específica. Essa camada específica pode ser escolhida com base na complexidade da representação desejada.

Configurando o treinamento de modelos

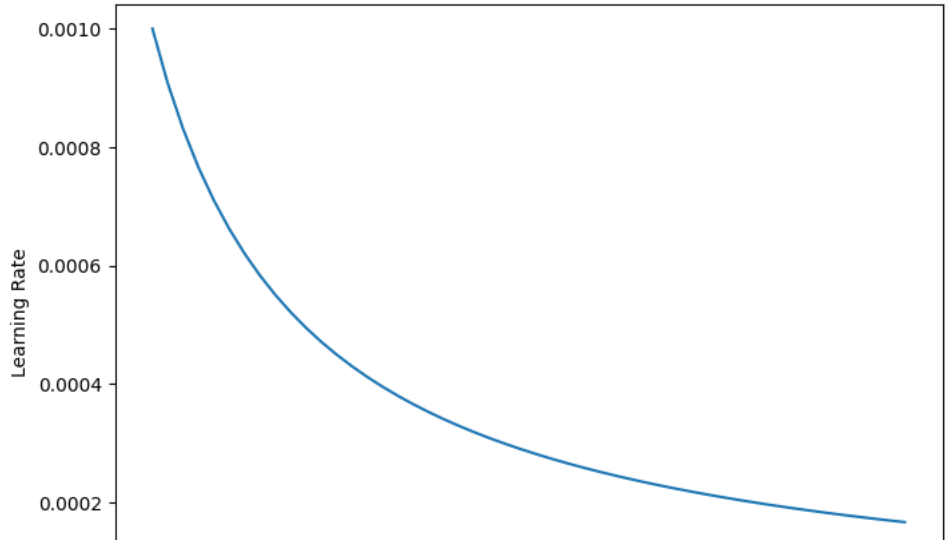
Nesta prática iremos avaliar diversos modelos. Para que a análise seja facilitada, usaremos a mesma configuração para todos os modelos.

- Muitos modelos treinam melhor se você reduzir gradualmente a taxa de aprendizado durante o treinamento. Uma forma de utilizar isso em TensorFlow é por meio dos `optimizers.schedules`, os quais variam a taxa de aprendizado ao longo do tempo. Nesta prática utilizaremos o `InverseTimeDecay`.
- Utilizaremos como base o otimizador `tf.keras.optimizers.Adam`, o qual usará o `InverseTimeDecay` para regular a taxa de aprendizado.

```
STEPS_PER_EPOCH = N_TRAIN // BATCH_SIZE # Total de amostras de treino / Batch size

lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.001,
  decay_steps=STEPS_PER_EPOCH*1000,
  decay_rate=1,
  staircase=False)

def get_optimizer():
  return tf.keras.optimizers.Adam(lr_schedule)
```

O código acima define um `schedules.InverseTimeDecay` para diminuir hiperbolicamente a taxa de aprendizado para 1/2 da taxa básica em 1.000 épocas, 1/3 em 2.000 épocas e assim por diante.

```
step = np.linspace(0,100000)
lr = lr_schedule(step)
plt.figure(figsize = (8,6))
plt.plot(step/STEPS_PER_EPOCH, lr)
plt.ylim([0,max(plt.ylim())])
plt.xlabel('Epoch')
_ = plt.ylabel('Learning Rate')
```



Outras Callbacks são necessárias:

- O treinamento para desta prática é executada por muitas épocas curtas. Para reduzir o ruído de log, use o `tfdocs.EpochDots` que simplesmente imprime `.` para cada época e um conjunto completo de métricas a cada 100 épocas.
- Para evitar tempos de treinamento longos e desnecessários, pode-se utilizar a `callbacks.EarlyStopping`. Observe que esse retorno de chamada é definido para monitorar o `val_binary_crossentropy`, não o `val_loss`. Essa diferença será importante mais tarde.
- Por fim, a `callbacks.TensorBoard` para gerar logs do TensorBoard para o treinamento.

```
logdir = pathlib.Path(tempfile.mkdtemp())/"tensorboard_logs"
shutil.rmtree(logdir, ignore_errors=True)

def get_callbacks(name):
  return [
    EpochDots(),
    tf.keras.callbacks.EarlyStopping(monitor='val_binary_crossentropy', patience=200),
    tf.keras.callbacks.TensorBoard(logdir/name),
  ]
```

Por fim, iremos definir uma função para treinar e compilar os modelos.

```
def compile_and_fit(model, name, max_epochs=10000):
    optimizer = get_optimizer()
    model.compile(optimizer=optimizer,
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                  metrics=[tf.keras.losses.BinaryCrossentropy(
                              from_logits=True,
                              name='binary_crossentropy'),
                           'accuracy'])

    model.summary()

    print(f"Antes do treinamento - Dimensões dos dados de treinamento: {treino_x.shape}")
    print(f"Antes do treinamento - Dimensões dos dados de teste: {teste_x.shape}")

    history = model.fit(
        train_ds,
        steps_per_epoch=STEPS_PER_EPOCH,
        epochs=max_epochs,
        validation_data=validate_ds,
        callbacks=get_callbacks(name),
        verbose=0
    )

    print(f"Após o treinamento - Dimensões dos dados de treinamento: {treino_x.shape}")
    print(f"Após o treinamento - Dimensões dos dados de teste: {teste_x.shape}")

    return history
```

Treinando um modelo minúsculo

```
tiny_model = tf.keras.Sequential([
    layers.Dense(16, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(1)
])
```

Criando uma variável para guardar a história dos modelos treinados.

```
size_histories = {}
```

Treinando o nosso pequeno modelo.

```
validate_ds = validate_ds.batch(BATCH_SIZE)
train_ds = train_ds.shuffle(int(1e4)).repeat().batch(BATCH_SIZE)

size_histories['Tiny'] = compile_and_fit(tiny_model, 'sizes/Tiny')
```

```
Model: "sequential_24"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_72 (Dense)            (None, 16)                464

 dense_73 (Dense)            (None, 1)                 17

=================================================================
Total params: 481 (1.88 KB)
Trainable params: 481 (1.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Antes do treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Antes do treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)
```

Verificando o modelo

```
plotter = HistoryPlotter(metric = 'binary_crossentropy', smoothing_std=10)
plotter.plot(size_histories)
plt.ylim([0.5, 0.7])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-35-a9e7ae8c1aed> in <cell line: 2>()
      1 plotter = HistoryPlotter(metric = 'binary_crossentropy', smoothing_std=10)
----> 2 plotter.plot(size_histories)
      3 plt.ylim([0.5, 0.7])

NameError: name 'size_histories' is not defined
```

SEARCH STACK OVERFLOW

As linhas sólidas mostram a `loss` de treinamento e as linhas tracejadas mostram a `loss` de validação (lembre-se: uma `loss` de validação menor indica um modelo melhor).

Treinando um modelo pequeno

Uma forma de tentar superar o desempenho do modelo minúsculo é treinando progressivamente alguns modelos maiores.

No modelo minúsculo usamos somente uma camada de 16 neurônios. Vamos experimentar com duas camadas ocultas com 16 unidades.

```
small_model = tf.keras.Sequential([
    layers.Dense(16, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(16, activation='elu'),
    layers.Dense(1)
])
```

```
size_histories['Small'] = compile_and_fit(small_model, 'sizes/Small')
```

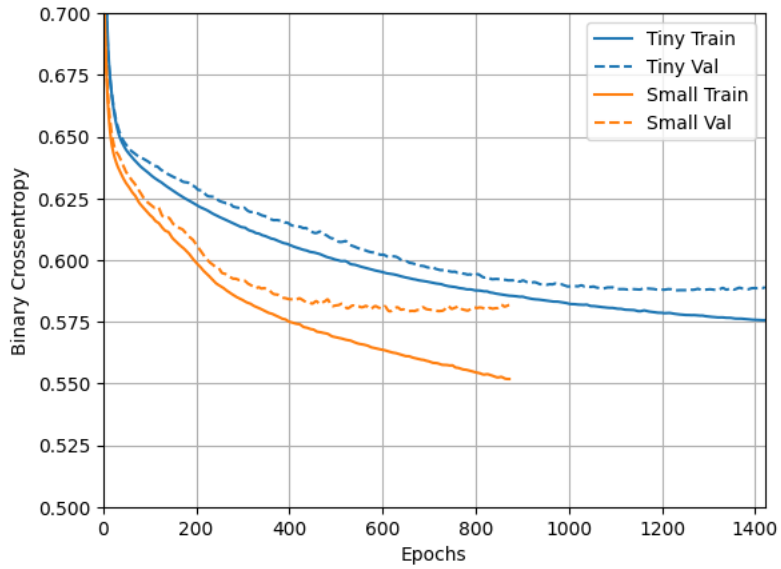
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 16)                464

 dense_5 (Dense)             (None, 16)                272
```

```
 dense_6 (Dense)             (None, 1)                 17

=================================================================
Total params: 753 (2.94 KB)
Trainable params: 753 (2.94 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch: 0, accuracy:0.4742,  binary_crossentropy:0.7889,  loss:0.7889,  val_accuracy:0.4830,  val_binary_crossentropy:0.7380,  val_lo
.....................................................................................................
Epoch: 100, accuracy:0.6178,  binary_crossentropy:0.6181,  loss:0.6181,  val_accuracy:0.5930,  val_binary_crossentropy:0.6215,  val_
.....................................................................................................
Epoch: 200, accuracy:0.6524,  binary_crossentropy:0.5991,  loss:0.5991,  val_accuracy:0.6320,  val_binary_crossentropy:0.6068,  val_
.....................................................................................................
Epoch: 300, accuracy:0.6702,  binary_crossentropy:0.5834,  loss:0.5834,  val_accuracy:0.6480,  val_binary_crossentropy:0.5901,  val_
.....................................................................................................
Epoch: 400, accuracy:0.6810,  binary_crossentropy:0.5751,  loss:0.5751,  val_accuracy:0.6580,  val_binary_crossentropy:0.5845,  val_
.....................................................................................................
Epoch: 500, accuracy:0.6852,  binary_crossentropy:0.5689,  loss:0.5689,  val_accuracy:0.6670,  val_binary_crossentropy:0.5806,  val_
.....................................................................................................
Epoch: 600, accuracy:0.6889,  binary_crossentropy:0.5636,  loss:0.5636,  val_accuracy:0.6800,  val_binary_crossentropy:0.5793,  val_
.....................................................................................................
Epoch: 700, accuracy:0.6949,  binary_crossentropy:0.5585,  loss:0.5585,  val_accuracy:0.6730,  val_binary_crossentropy:0.5793,  val_
.....................................................................................................
Epoch: 800, accuracy:0.6962,  binary_crossentropy:0.5544,  loss:0.5544,  val_accuracy:0.6760,  val_binary_crossentropy:0.5795,  val_
.............................................................
```

Verificando ambos modelos treinados

```
plotter = HistoryPlotter(metric = 'binary_crossentropy', smoothing_std=10)
plotter.plot(size_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```



Treinando um modelo médio

Vamos tentar agora um modelo com 3 camadas e 64 neurônios por camada (quatro vezes mais).

```
medium_model = tf.keras.Sequential([
    layers.Dense(64, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(64, activation='elu'),
    layers.Dense(64, activation='elu'),
    layers.Dense(1)
])
```

```
size_histories['Medium']  = compile_and_fit(medium_model, "sizes/Medium")
```

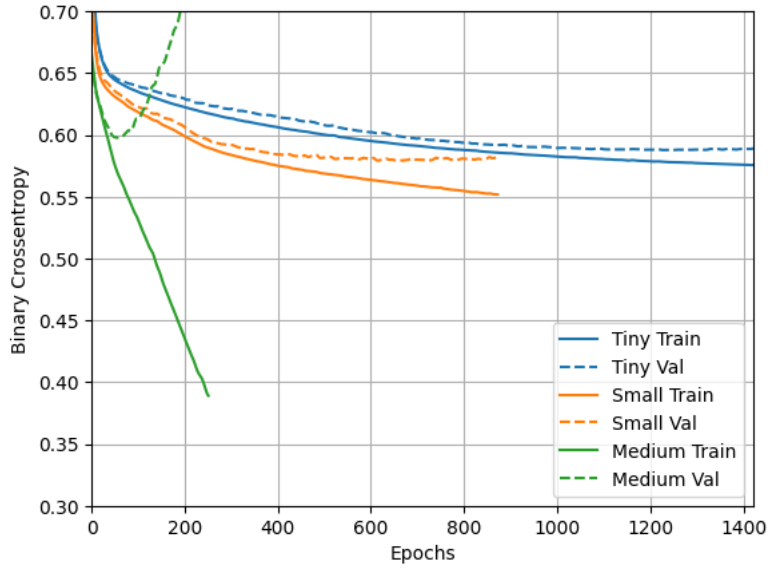
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_7 (Dense)             (None, 64)                1856
```

```
 dense_8 (Dense)             (None, 64)                4160

 dense_9 (Dense)             (None, 64)                4160

 dense_10 (Dense)            (None, 1)                 65

=================================================================
Total params: 10241 (40.00 KB)
Trainable params: 10241 (40.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch: 0, accuracy:0.4917,  binary_crossentropy:0.6975,  loss:0.6975,  val_accuracy:0.4830,  val_binary_crossentropy:0.6798,  val_lo
.....................................................................................................
Epoch: 100, accuracy:0.7197,  binary_crossentropy:0.5310,  loss:0.5310,  val_accuracy:0.6580,  val_binary_crossentropy:0.6087,  val_
.....................................................................................................
Epoch: 200, accuracy:0.7846,  binary_crossentropy:0.4366,  loss:0.4366,  val_accuracy:0.6380,  val_binary_crossentropy:0.7189,  val_
..................................................
```

Verificando os modelos treinados

```
plotter = HistoryPlotter(metric = 'binary_crossentropy', smoothing_std=10)
plotter.plot(size_histories)
plt.ylim([0.3, 0.7])
```

```
(0.3, 0.7)
```



Treinando um modelo grande

Você pode criar um modelo ainda maior e verificar a rapidez com que ele começa a fazer overfitting. O novo modelo possui mais camadas e mais neurônios por camada (512).

```
large_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(512, activation='elu'),
    layers.Dense(512, activation='elu'),
    layers.Dense(512, activation='elu'),
    layers.Dense(1)
])
```

```
size_histories['large'] = compile_and_fit(large_model, "sizes/large")
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_11 (Dense)            (None, 512)               14848

 dense_12 (Dense)            (None, 512)               262656

 dense_13 (Dense)            (None, 512)               262656

 dense_14 (Dense)            (None, 512)               262656
```

```
 dense_15 (Dense)            (None, 1)                 513

=================================================================
Total params: 803329 (3.06 MB)
Trainable params: 803329 (3.06 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch: 0, accuracy:0.5113,  binary_crossentropy:0.8193,  loss:0.8193,  val_accuracy:0.4690,  val_binary_crossentropy:0.6993,  val_lo
.....................................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0022,  loss:0.0022,  val_accuracy:0.6530,  val_binary_crossentropy:1.7998,  val_
.....................................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0001,  loss:0.0001,  val_accuracy:0.6500,  val_binary_crossentropy:2.4604,  val_
.................
```

Verificando os modelos treinados

Embora a construção de um modelo maior lhe dê mais poder, se esse poder não for restringido de alguma forma, ele pode facilmente se ajustar ao conjunto de treinamento.

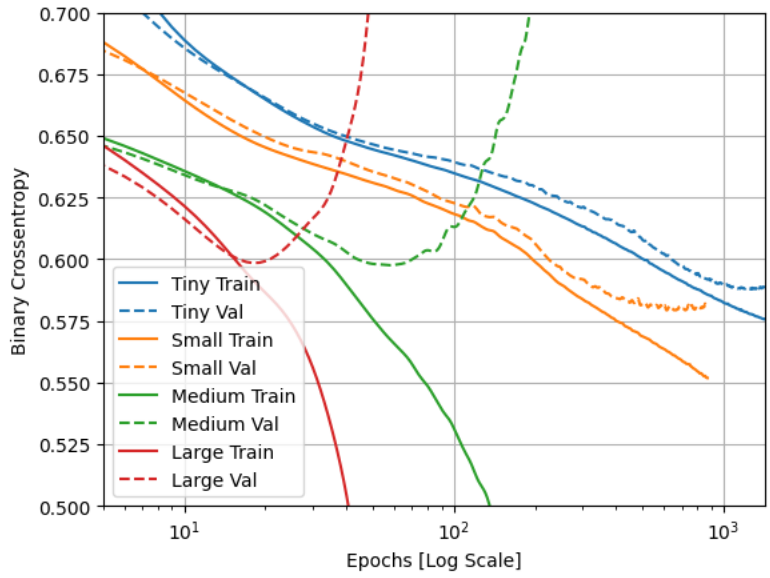
Neste exemplo, normalmente, apenas o modelo minúsculo (`"Tiny"`) consegue evitar completamente o overfitting, e cada um dos modelos maiores superajusta os dados mais rapidamente. Isso se torna tão grave para o modelo grande (`"Large"`) que você precisa mudar o gráfico para uma escala logarítmica para realmente descobrir o que está acontecendo.

Isso fica aparente se você plotar e comparar as métricas de validação com as métricas de treinamento.

- É normal que haja uma pequena diferença.
- Se ambas as métricas estiverem se movendo na mesma direção, está tudo bem.
- Se a métrica de validação começar a estagnar enquanto a métrica de treinamento continua a melhorar, você provavelmente está perto do overfitting.
- Se a métrica de validação estiver indo na direção errada, o modelo está claramente superajustado.

```
plotter.plot(size_histories)
a = plt.xscale('log')
#Sets the x-axis limits, starting from epoch 5 (excluding earlier epochs) up to the maximum epoch value in the plot. This is done to avo
plt.xlim([5, max(plt.xlim())])
plt.ylim([0.5, 0.7])
plt.xlabel("Epochs [Log Scale]")
```

```
Text(0.5, 0, 'Epochs [Log Scale]')
```



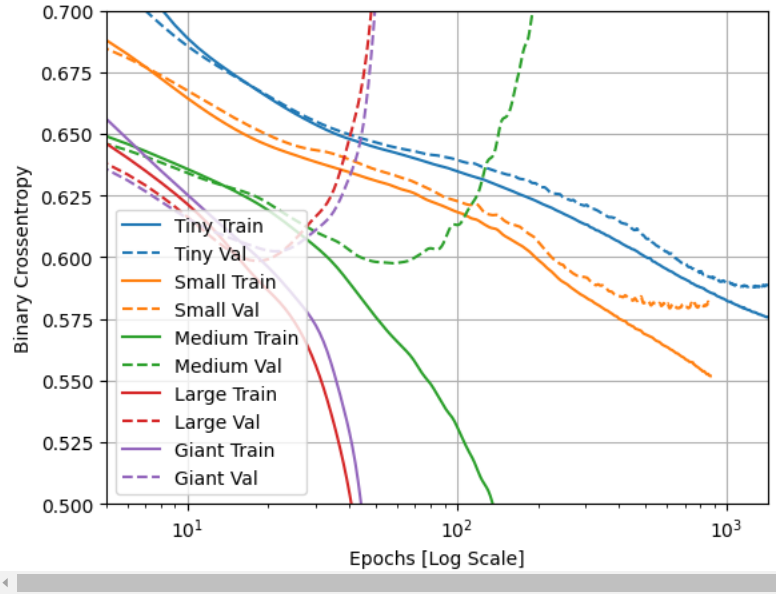
Todas as execuções de treinamento acima usaram o `callbacks.EarlyStopping` para encerrar o treinamento, uma vez que ficou claro que o modelo não estava progredindo.

ToDo: Avaliando um modelo gigante (10pt)

Adicione a esse *benchmark* uma rede que tenha muito mais capacidade, muito mais do que o problema precisa.

```
### Seu código aqui
giant_model = tf.keras.Sequential([
    layers.Dense(1024, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(1024, activation='elu'),
    layers.Dense(1024, activation='elu'),
    layers.Dense(1024, activation='elu'),
    layers.Dense(1)
])

size_histories['giant'] = compile_and_fit(giant_model, "sizes/giant")

plotter.plot(size_histories)
a = plt.xscale('log')
plt.xlim([5, max(plt.xlim())])
plt.ylim([0.5, 0.7])
plt.xlabel("Epochs [Log Scale]")
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 1024)              29696

 dense_17 (Dense)            (None, 1024)              1049600

 dense_18 (Dense)            (None, 1024)              1049600

 dense_19 (Dense)            (None, 1024)              1049600

 dense_20 (Dense)            (None, 1)                 1025

=================================================================
Total params: 3179521 (12.13 MB)
Trainable params: 3179521 (12.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch: 0, accuracy:0.5133,  binary_crossentropy:0.9554,  loss:0.9554,  val_accuracy:0.5490,  val_binary_crossentropy:0.6788,  val_lo
.....................................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0009,  loss:0.0009,  val_accuracy:0.6620,  val_binary_crossentropy:1.7867,  val_
.....................................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0001,  loss:0.0001,  val_accuracy:0.6610,  val_binary_crossentropy:2.2171,  val_
.....................Text(0.5, 0, 'Epochs [Log Scale]')
```



ToDo: Avalie o seu modelo treinado conforme foi feito nos exemplos anteriores (10pt)

- No início do treinamento, a acurácia é modesta, em torno de 50%, o que é esperado para um modelo inicializado aleatoriamente. Ao longo das épocas, a acurácia de treinamento melhora gradualmente e atinge 100% após (apenas) 100 épocas. A acurácia na validação tem uma melhoria pequena, atingindo cerca de 66% após também 100 épocas.
- O modelo "Gigante" segue uma tendência semelhante ao modelo "Grande" em termos de overfitting, atingindo 100% de acurácia no treinamento de forma bem rápida. No entanto, a acurácia na validação do modelo "Gigante" é marginalmente melhor em comparação com o modelo "Grande". Apesar de começar marginalmente pior, o modelo "Gigante" consegue melhorar sua acurácia na validação ao longo do treinamento.

Embora ambos os modelos ("Gigante" e "Grande") sofram de overfitting, o modelo "Gigante" pode, paradoxicalmente, ter uma vantagem marginal *devido* à sua capacidade extra.

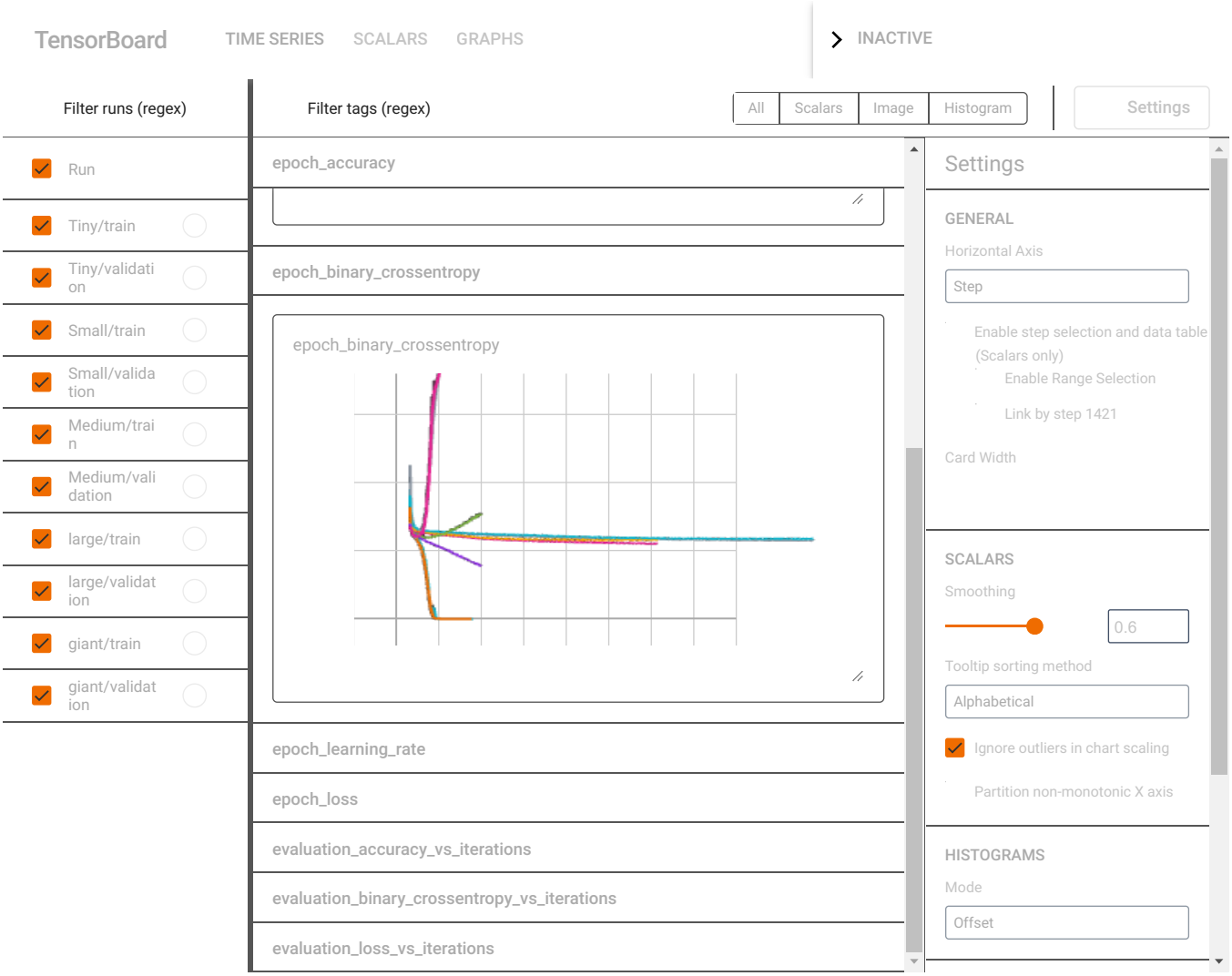
No entanto, é importante destacar que ambos os modelos ainda enfrentam desafios de generalização para novos dados e, apesar de ser interessante checar se essa anormalidade no padrão estabelecido - de modelos maiores sofrerem *overfitting* mais rapidamente - talvez seja melhor explorar outras estratégias para mitigar o overfitting.

Avaliando os resultados no *TensorBoard*

Salvamos os logs do TensorBoard durante o treinamento de todos os modelos treinados.

Podemos abrir um visualizador TensorBoard incorporado em um notebook.

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
#%reload_ext tensorboard

# Open an embedded TensorBoard viewer
%tensorboard --logdir {logdir}/sizes

#!kill 35913
```



ToDo: Análises (10pt)

Quais análises você pode fazer sobre o TensorBoard?

Aparenta ser uma ferramenta bem útil para analise comparativa dos modelos e seus resultados. A facilidade de navegar entre visualizações de diferentes métricas permitiu análises mais rápidas e intuitivas do quê está acontecendo ali.

Estratégias para previnir *overfitting* (sobreajuste)

Antes de entrar no conteúdo desta seção, copie os logs de treinamento do modelo minúsculo (`"Tiny"`) acima, para usar como linha de base para comparação.

Iremos comparar os logs de treinamento do modelo minúsculo (`"Tiny"`) acima, por isso iremos copiar os logs.

```
shutil.rmtree(logdir/'regularizers/Tiny', ignore_errors=True)
shutil.copytree(logdir/'sizes/Tiny', logdir/'regularizers/Tiny')
```

```
PosixPath('/tmp/tmp2sz7h20o/tensorboard_logs/regularizers/Tiny')
```

```
regularizer_histories = {}
regularizer_histories['Tiny'] = size_histories['Tiny']
```

Adicionando estratégias de regularização ao modelo

Você pode estar familiarizado com o princípio da Navalha de Occam: dadas duas explicações para algo, a explicação mais provável de ser correta é a **mais simples**, aquela que faz a menor quantidade de suposições. Isso também se aplica aos modelos aprendidos pelas redes neurais: dados alguns dados de treinamento e uma arquitetura de rede, existem vários conjuntos de valores de pesos (múltiplos modelos) que podem explicar os dados, e modelos mais simples são menos propensos a sobreajustar do que os complexos.

Um "modelo simples" neste contexto é um modelo onde a distribuição de valores de parâmetros tem menos entropia (ou um modelo com menos parâmetros, como demonstrado na seção acima). Assim, uma maneira comum de mitigar o *overfitting* é colocar restrições na complexidade de uma rede, forçando seus pesos apenas a assumir valores pequenos, o que torna a distribuição de valores de peso mais "regular". Isso é chamado de "regularização de peso", e é feito adicionando à função de perda da rede um custo associado a ter grandes pesos. Este custo vem em dois sabores:

- Regularização L1, onde o custo adicionado é proporcional ao valor absoluto dos coeficientes dos pesos (ou seja, ao que é chamado de "norma L1 "dos pesos).
- Regularização L2, onde o custo adicionado é proporcional ao quadrado do valor dos coeficientes dos pesos (ou seja, ao que é chamado de quadrado "norma L2" dos pesos). A regularização L2 também é chamada de decaimento de peso no contexto de redes neurais. Não deixe que o nome diferente o confunda: a redução de peso (*weight decay*) é matematicamente igual à regularização L2.

A regularização L1 empurra os pesos para zero, incentivando um modelo esparso. A regularização de L2 penalizará os parâmetros de pesos sem torná-los esparsos, já que a penalidade vai para zero para pesos pequenos - uma razão pela qual L2 é mais comum.

Em `tf.keras`, a regularização de peso é adicionada passando instâncias do regularizador de peso para camadas como argumentos de palavras-chave. Adicione regularização de peso L2:

```
l2_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', kernel_regularizer=regularizers.l2(0.001), input_shape=(FEATURES,)),
    layers.Dense(512, activation='elu', kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu', kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu', kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1)
])

regularizer_histories['l2'] = compile_and_fit(l2_model, "regularizers/l2")
```

```
Model: "sequential_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_47 (Dense)            (None, 512)               14848

 dense_48 (Dense)            (None, 512)               262656

 dense_49 (Dense)            (None, 512)               262656

 dense_50 (Dense)            (None, 512)               262656

 dense_51 (Dense)            (None, 1)                 513

=================================================================
Total params: 803329 (3.06 MB)
Trainable params: 803329 (3.06 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch: 0, accuracy:0.5034,  binary_crossentropy:0.8060,  loss:2.3243,  val_accuracy:0.4890,  val_binary_crossentropy:0.6790,  val_lo
.....................................................................................
Epoch: 100, accuracy:0.6572,  binary_crossentropy:0.5961,  loss:0.6190,  val_accuracy:0.6750,  val_binary_crossentropy:0.5791,  val_
.....................................................................................
Epoch: 200, accuracy:0.6736,  binary_crossentropy:0.5807,  loss:0.6031,  val_accuracy:0.6650,  val_binary_crossentropy:0.5768,  val_
.....................................................................................
Epoch: 300, accuracy:0.6823,  binary_crossentropy:0.5724,  loss:0.5948,  val_accuracy:0.6730,  val_binary_crossentropy:0.5823,  val_
```

```
.....................................................................................
Epoch: 400, accuracy:0.6898,  binary_crossentropy:0.5641,  loss:0.5882,  val_accuracy:0.6930,  val_binary_crossentropy:0.5712,  val_
.....................................................................................
Epoch: 500, accuracy:0.6960,  binary_crossentropy:0.5554,  loss:0.5807,  val_accuracy:0.6910,  val_binary_crossentropy:0.5766,  val_
.....................................................................................
Epoch: 600, accuracy:0.7057,  binary_crossentropy:0.5469,  loss:0.5732,  val_accuracy:0.6790,  val_binary_crossentropy:0.5829,  val_
.....................................................................................
Epoch: 700, accuracy:0.7061,  binary_crossentropy:0.5457,  loss:0.5733,  val_accuracy:0.6780,  val_binary_crossentropy:0.5838,  val_
.........................
```

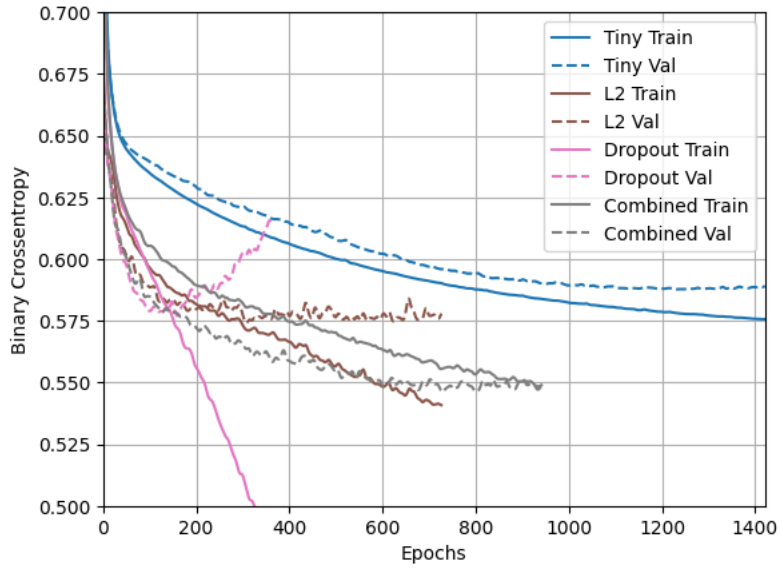
`l2(0,001)` significa que cada coeficiente na matriz de peso da camada adicionará `0,001 * weight_coeficiente_value**2` ao total de **perda** da rede.

É por isso que estamos monitorando o `binary_crossentropy` diretamente. Porque não tem esse componente de regularização misturado.

Então, esse mesmo modelo `"Large"` com uma penalidade de regularização `L2` tem um desempenho muito melhor:

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```



Conforme demonstrado, o modelo regularizado `"L2"` agora é muito mais competitivo com o modelo `"Tiny"`. Este modelo `"L2"` também é muito mais resistente ao overfitting do que o modelo `"Large"` no qual foi baseado, apesar de ter o mesmo número de parâmetros.

Adicionando *dropout*

Dropout é uma das técnicas de regularização mais eficazes e mais utilizadas para redes neurais, desenvolvida por Hinton e seus alunos da Universidade de Toronto.

A explicação intuitiva para o dropout é que, como os nós individuais na rede não podem contar com a saída dos outros, cada nó deve produzir recursos que sejam úteis por conta própria.

O dropout, aplicado a uma camada, consiste em "descartar" aleatoriamente (ou seja, definir como zero) um número de recursos de saída da camada durante o treinamento. Por exemplo, uma determinada camada normalmente retornaria um vetor `[0.2, 0.5, 1.3, 0.8, 1.1]` para uma determinada amostra de entrada durante o treinamento; após aplicar dropout, este vetor terá algumas entradas zero distribuídas aleatoriamente, por exemplo. `[0, 0,5, 1,3, 0, 1,1]`.

A "taxa de abandono" é a fração dos recursos que estão sendo zerados; geralmente é definido entre 0,2 e 0,5. No momento do teste, nenhuma unidade é descartada e, em vez disso, os valores de saída da camada são reduzidos por um fator igual à taxa de abandono, de modo a equilibrar o fato de que mais unidades estão ativas do que no tempo de treinamento.

No Keras, você pode introduzir dropout em uma rede através da camada `tf.keras.layers.Dropout`, que é aplicada à saída da camada imediatamente anterior.

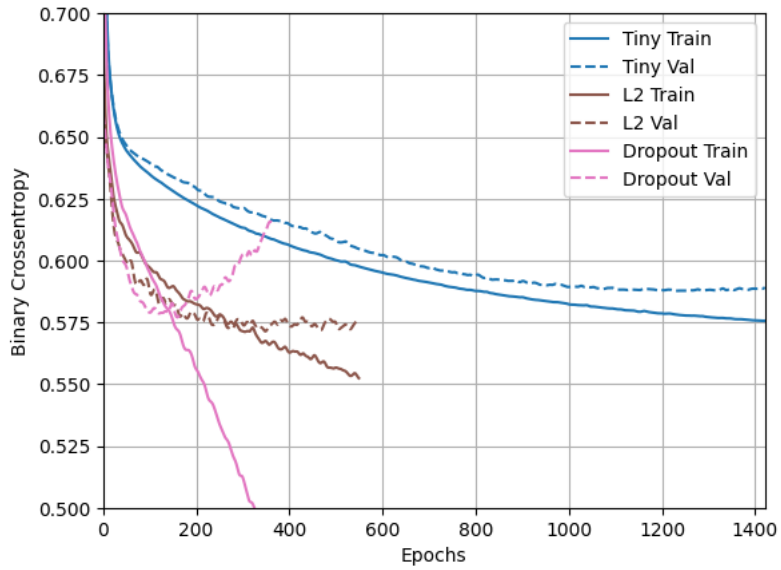
Adicione duas camadas de dropout à sua rede para verificar o desempenho delas na redução do overfitting:

```
dropout_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['dropout'] = compile_and_fit(dropout_model, "regularizers/dropout")
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_26 (Dense)            (None, 512)               14848

 dropout (Dropout)           (None, 512)               0

 dense_27 (Dense)            (None, 512)               262656

 dropout_1 (Dropout)         (None, 512)               0

 dense_28 (Dense)            (None, 512)               262656

 dropout_2 (Dropout)         (None, 512)               0

 dense_29 (Dense)            (None, 512)               262656

 dropout_3 (Dropout)         (None, 512)               0

 dense_30 (Dense)            (None, 1)                 513

=================================================================
Total params: 803329 (3.06 MB)
Trainable params: 803329 (3.06 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Epoch: 0, accuracy:0.5112,  binary_crossentropy:0.7937,  loss:0.7937,  val_accuracy:0.5070,  val_binary_crossentropy:0.6661,  val_lo
.....................................................................................................
Epoch: 100, accuracy:0.6520,  binary_crossentropy:0.5967,  loss:0.5967,  val_accuracy:0.6710,  val_binary_crossentropy:0.5797,  val_
.....................................................................................................
Epoch: 200, accuracy:0.6855,  binary_crossentropy:0.5590,  loss:0.5590,  val_accuracy:0.6860,  val_binary_crossentropy:0.5867,  val_
.....................................................................................................
Epoch: 300, accuracy:0.7107,  binary_crossentropy:0.5155,  loss:0.5155,  val_accuracy:0.6860,  val_binary_crossentropy:0.6082,  val_
.............................................................
```

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```



Fica claro neste gráfico que ambas as abordagens de regularização melhoram o comportamento do modelo grande (`"Large"`). Mas isso ainda não supera nem mesmo a linha de base `"Tiny"` .

Naturalmente, o próximo passo é testar os dois juntos.

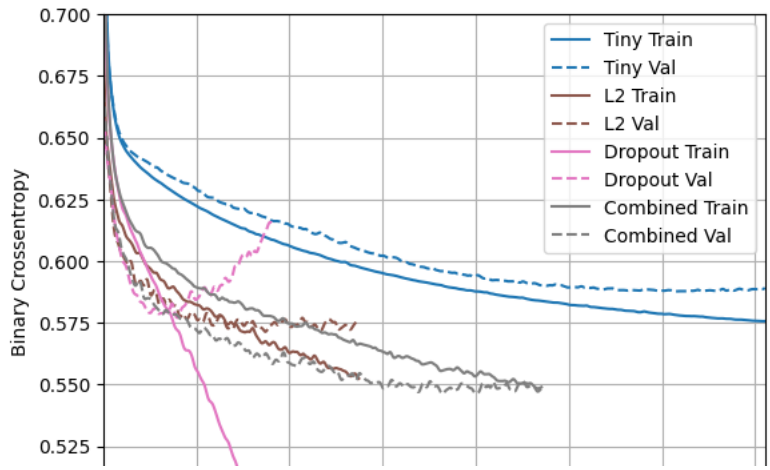
Combinando L2 + *dropout*

```
combined_model = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['combined'] = compile_and_fit(combined_model, "regularizers/combined")
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_31 (Dense)            (None, 512)               14848

 dropout_4 (Dropout)         (None, 512)               0

 dense_32 (Dense)            (None, 512)               262656

 dropout_5 (Dropout)         (None, 512)               0

 dense_33 (Dense)            (None, 512)               262656

 dropout_6 (Dropout)         (None, 512)               0

 dense_34 (Dense)            (None, 512)               262656

 dropout_7 (Dropout)         (None, 512)               0

 dense_35 (Dense)            (None, 1)                 513

=================================================================
Total params: 803329 (3.06 MB)
Trainable params: 803329 (3.06 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch: 0, accuracy:0.4980,  binary_crossentropy:0.8227,  loss:0.9810,  val_accuracy:0.5520,  val_binary_crossentropy:0.6909,  val_lo
.....................................................................................
Epoch: 100, accuracy:0.6450,  binary_crossentropy:0.6062,  loss:0.6369,  val_accuracy:0.6570,  val_binary_crossentropy:0.5847,  val_
.....................................................................................
Epoch: 200, accuracy:0.6617,  binary_crossentropy:0.5926,  loss:0.6186,  val_accuracy:0.6870,  val_binary_crossentropy:0.5637,  val_
.....................................................................................
Epoch: 300, accuracy:0.6686,  binary_crossentropy:0.5826,  loss:0.6114,  val_accuracy:0.6850,  val_binary_crossentropy:0.5649,  val_
.....................................................................................
Epoch: 400, accuracy:0.6812,  binary_crossentropy:0.5732,  loss:0.6047,  val_accuracy:0.6740,  val_binary_crossentropy:0.5671,  val_
.....................................................................................
Epoch: 500, accuracy:0.6796,  binary_crossentropy:0.5723,  loss:0.6060,  val_accuracy:0.6710,  val_binary_crossentropy:0.5536,  val_
.....................................................................................
Epoch: 600, accuracy:0.6848,  binary_crossentropy:0.5635,  loss:0.5998,  val_accuracy:0.6900,  val_binary_crossentropy:0.5470,  val_
.....................................................................................
Epoch: 700, accuracy:0.6851,  binary_crossentropy:0.5585,  loss:0.5966,  val_accuracy:0.7030,  val_binary_crossentropy:0.5531,  val_
.....................................................................................
Epoch: 800, accuracy:0.6997,  binary_crossentropy:0.5558,  loss:0.5954,  val_accuracy:0.6970,  val_binary_crossentropy:0.5441,  val_
.....................................................................................
Epoch: 900, accuracy:0.6975,  binary_crossentropy:0.5497,  loss:0.5903,  val_accuracy:0.7020,  val_binary_crossentropy:0.5544,  val_
.....................................
```

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```



Este modelo com as regularizações combinadas (`"Combined"`) é obviamente o melhor até agora.

0 200 400 600 800 1000 1200 1400

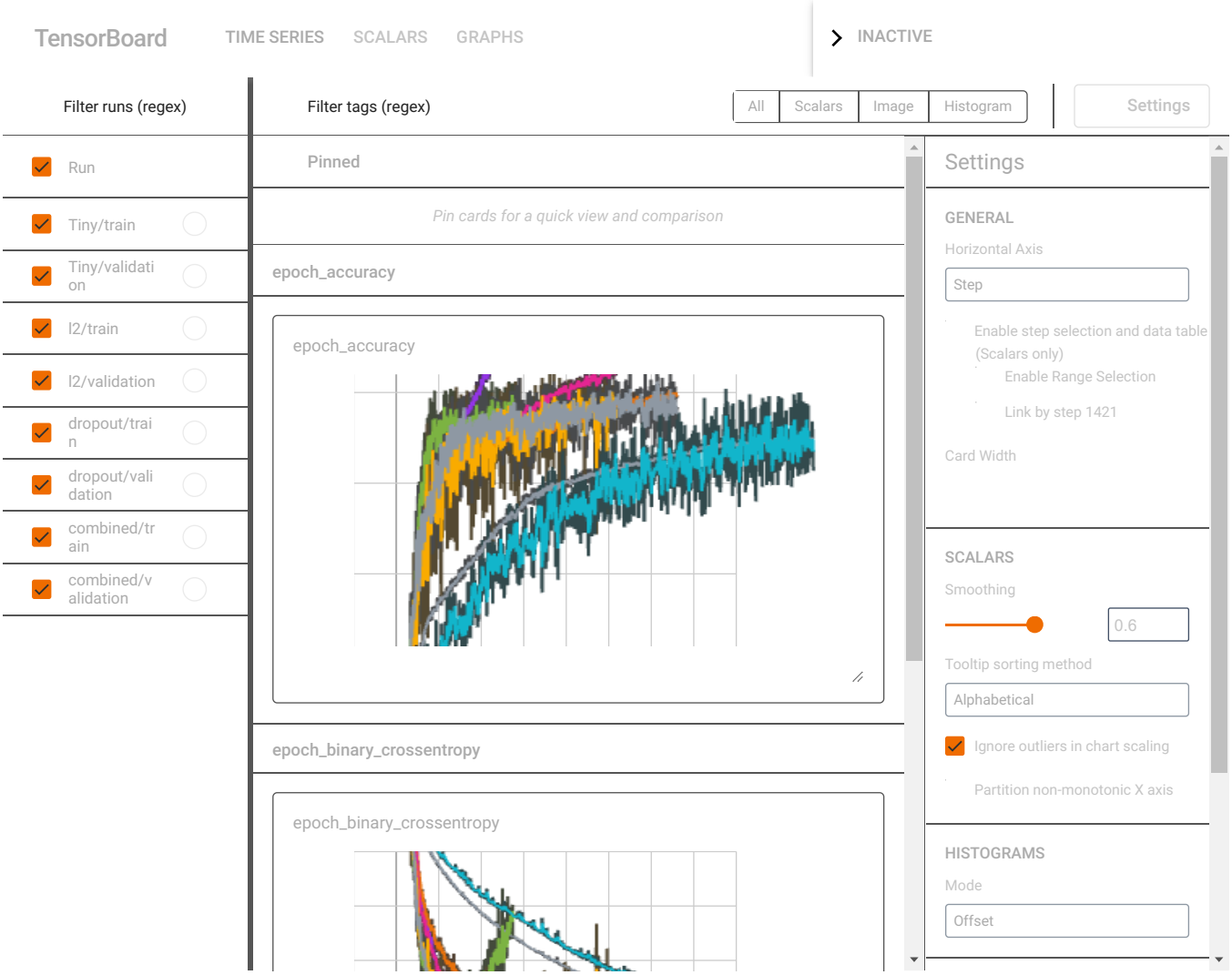
Avaliando os resultados no *TensorBoard*

Esses modelos também registraram logs do TensorBoard.

Para abrir um visualizador de tensorboard incorporado em um notebook, copie o seguinte em uma célula de código:

```
%tensorboard --logdir {logdir}/regularizers
```

%tensorboard --logdir {logdir}/regularizers



ToDo: Análise dos resultados (10pt)

O que você pode inferir analisando os resultados apresentados no TensorBoard?

` Escreva sua resposta aqui

Analisando os resultados apresentados, podemos inferir:

Modelo Inicial ("Tiny"): Este modelo, com uma arquitetura simples, mostrou ser eficaz para evitar o overfitting. No entanto, sua capacidade preditiva é limitada, resultando em uma acurácia modesta.

Modelos Maiores ("Medium", "Large", "Giant"): Modelos com capacidade significativamente maior mostraram um rápido overfitting aos dados de treinamento. Eles alcançaram alta acurácia no treinamento, mas essa alta performance não se traduziu igualmente na validação, indicando dificuldades de generalização.

Estratégias de Regularização (L2 e Dropout): A introdução de estratégias de regularização, como L2 e Dropout, mostrou-se eficaz para mitigar o overfitting nos modelos maiores. Especificamente, o modelo regularizado com L2 teve um desempenho muito melhor em comparação com o modelo sem regularização.

Combinação de Regularização (L2 + Dropout): A combinação de regularização L2 e dropout em um modelo mostrou ser uma abordagem ainda mais eficaz para evitar o overfitting. Este modelo combinado conseguiu resistir ao overfitting, apresentando um desempenho competitivo em comparação com o modelo inicial "Tiny".

Necessidade de Ajustes Adicionais: Embora a combinação de L2 e dropout seja uma estratégia eficaz, pode ser útil explorar outras técnicas e ajustes para melhorar ainda mais o desempenho e a generalização do modelo, especialmente ao lidar com conjuntos de dados mais complexos.

Importância da Análise Gráfica: A análise gráfica das métricas de treinamento e validação ao longo do tempo foi crucial para identificar overfitting, entender o desempenho do modelo e avaliar a eficácia das estratégias de regularização.

Essas conclusões destacam a importância de ajustar cuidadosamente a complexidade do modelo e incorporar estratégias de regularização para garantir um bom desempenho em dados de validação e evitar overfitting. `

Avaliando estrátegias de *overfitting* e regularização para a base de gato/não-gato

Para essa próxima tarefa, avalie três modelos (semelhante ao que foi feito para a base de Higgs). Você deve treinar:

- Um modelo pequeno.
- Um modelo médio.
- Um modelo grande.

Criando uma variável para guardar a história dos modelos treinados.

```
cat_histories = {}
```

ToDo: Lendo os dados da base de gatos/não-gatos (10pt)

```
### Seu código aqui
import h5py
from google.colab import drive
drive.mount("/content/drive", force_remount=True)
# Função para ler os dados (gato/não-gato)
def load_dataset():
    def _load_data():
      train_dataset = h5py.File('drive/MyDrive/datasets/train_catvnoncat.h5', "r")
      train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
      train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

      test_dataset = h5py.File('drive/MyDrive/datasets/test_catvnoncat.h5', "r")
      test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
      test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

      classes = np.array(test_dataset["list_classes"][:]) # the list of classes
      train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
      test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

      return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes

    def _preprocess_dataset(_treino_x_orig, _teste_x_orig):
      # Formate o conjunto de treinamento e teste dados de treinamento e teste para que as imagens
      # de tamanho (num_px, num_px, 3) sejam vetores de forma (num_px * num_px * 3, 1)
      _treino_x_vet = _treino_x_orig.reshape(_treino_x_orig.shape[0], 64, 64, 3) # Corrija as dimensões aqui
      _teste_x_vet = _teste_x_orig.reshape(_teste_x_orig.shape[0], 64, 64, 3) # Corrija as dimensões aqui

      # Normalize os dados (colocar no intervalo [0.0, 1.0])
      _treino_x = _treino_x_vet / 255. # ToDo: normalize os dados de treinamento aqui
      _teste_x = _teste_x_vet / 255. # ToDo: normalize os dados de teste aqui
      return _treino_x, _teste_x

    treino_x_orig, treino_y, teste_x_orig, teste_y, classes = _load_data()
    treino_x, teste_x = _preprocess_dataset(treino_x_orig, teste_x_orig)
    return treino_x, treino_y, teste_x, teste_y, classes
# Lendo os dados (gato/não-gato)
treino_x, treino_y, teste_x, teste_y, classes = load_dataset()
```

```
Mounted at /content/drive
```

```
#É necessário alterar a implementação da função compile_and_fit para os novos dados
#O erro ocorreu durante o treinamento do modelo, e a mensagem de erro indica que há uma incompatibilidade nas dimensões dos dados de ent
#O modelo espera dados de entrada com formato (None, 64, 64, 3), mas recebeu dados com formato (None, 28).
#Ao analisar o código, parece que o problema está relacionado ao uso do conjunto de dados train_ds e validate_ds no método model.fit
#Substituimos então o uso desses conjuntos de dados pelos arrays treino_x e teste_x, que são os dados de treinamento e teste carregados
def compile_and_fit(model, name, max_epochs=10000):
    optimizer = get_optimizer()
    model.compile(optimizer=optimizer,
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                  metrics=[tf.keras.losses.BinaryCrossentropy(
                              from_logits=True,
                              name='binary_crossentropy'),
                           'accuracy'])

    model.summary()

    print(f"Antes do treinamento - Dimensões dos dados de treinamento: {treino_x.shape}")
    print(f"Antes do treinamento - Dimensões dos dados de teste: {teste_x.shape}")


    history = model.fit(
      treino_x,  # Altere para o conjunto de dados de treinamento
      treino_y.flatten(),  # Flatten da matriz de rótulos
      steps_per_epoch=STEPS_PER_EPOCH,
      epochs=max_epochs,
      validation_data=(teste_x, teste_y.flatten()),  # Altere para o conjunto de dados de teste
      callbacks=get_callbacks(name),
      verbose=0
    )

    print(f"Após o treinamento - Dimensões dos dados de treinamento: {treino_x.shape}")
    print(f"Após o treinamento - Dimensões dos dados de teste: {teste_x.shape}")

    return history
```

ToDo: Treinando um modelo pequeno (10pt)

```
# Definindo o modelo pequeno
modelo_pequeno = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(64, 64, 3)),  # Camada de flatten para transformar a imagem em vetor
    tf.keras.layers.Dense(128, activation='relu'),  # Camada densa com ativação ReLU
    tf.keras.layers.Dense(1, activation='sigmoid')  # Camada de saída com ativação sigmoid para classificação binária
])
cat_histories['Pequeno'] = compile_and_fit(modelo_pequeno, 'sizes/Pequeno')
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 12288)             0

 dense_4 (Dense)             (None, 128)               1572992

 dense_5 (Dense)             (None, 1)                 129

=================================================================
Total params: 1573121 (6.00 MB)
Trainable params: 1573121 (6.00 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Antes do treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Antes do treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)
/usr/local/lib/python3.10/dist-packages/keras/src/backend.py:5820: UserWarning: "`binary_crossentropy` received `from_logits=True`,
  output, from_logits = _get_logits(

Epoch: 0, accuracy:0.5864,  binary_crossentropy:1.6619,  loss:1.6619,  val_accuracy:0.4000,  val_binary_crossentropy:0.9943,  val_lo
....................................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0109,  loss:0.0109,  val_accuracy:0.6800,  val_binary_crossentropy:1.6194,  val_
....................................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0014,  loss:0.0014,  val_accuracy:0.7000,  val_binary_crossentropy:2.0691,  val_
..Após o treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Após o treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)
```

ToDo: Treinando um modelo médio (10pt)

```
# Definindo o modelo médio
modelo_medio = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(64, 64, 3)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
cat_histories['Medio'] = compile_and_fit(modelo_medio, 'sizes/Medio')
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_3 (Flatten)         (None, 12288)             0

 dense_10 (Dense)            (None, 256)               3145984

 dense_11 (Dense)            (None, 256)               65792

 dense_12 (Dense)            (None, 256)               65792

 dense_13 (Dense)            (None, 1)                 257

=================================================================
Total params: 3277825 (12.50 MB)
Trainable params: 3277825 (12.50 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Antes do treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Antes do treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)

Epoch: 0, accuracy:0.5591,  binary_crossentropy:1.1890,  loss:1.1890,  val_accuracy:0.6400,  val_binary_crossentropy:0.6093,  val_lo
....................................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0001,  loss:0.0001,  val_accuracy:0.6800,  val_binary_crossentropy:3.5523,  val_
....................................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0000,  loss:0.0000,  val_accuracy:0.6800,  val_binary_crossentropy:4.7553,  val_
............Após o treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Após o treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)
```

ToDo: Treinando um modelo grande (10pt)

```
# Definindo o modelo grande
modelo_grande = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(64, 64, 3)),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
cat_histories['Grande'] = compile_and_fit(modelo_grande, 'sizes/Grande')
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_4 (Flatten)         (None, 12288)             0

 dense_14 (Dense)            (None, 1024)              12583936

 dense_15 (Dense)            (None, 1024)              1049600

 dense_16 (Dense)            (None, 1024)              1049600

 dense_17 (Dense)            (None, 1024)              1049600

 dense_18 (Dense)            (None, 1)                 1025

=================================================================
Total params: 15733761 (60.02 MB)
Trainable params: 15733761 (60.02 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Antes do treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Antes do treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)

Epoch: 0, accuracy:0.5545,  binary_crossentropy:1.5507,  loss:1.5507,  val_accuracy:0.3400,  val_binary_crossentropy:1.3063,  val_lo
.....................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0000,  loss:0.0000,  val_accuracy:0.7600,  val_binary_crossentropy:6.5094,  val_
.....................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0000,  loss:0.0000,  val_accuracy:0.7600,  val_binary_crossentropy:7.8080,  val_
..................Após o treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Após o treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)
```

ToDo: Avalie a adição de regularização aos modelos (10pt)

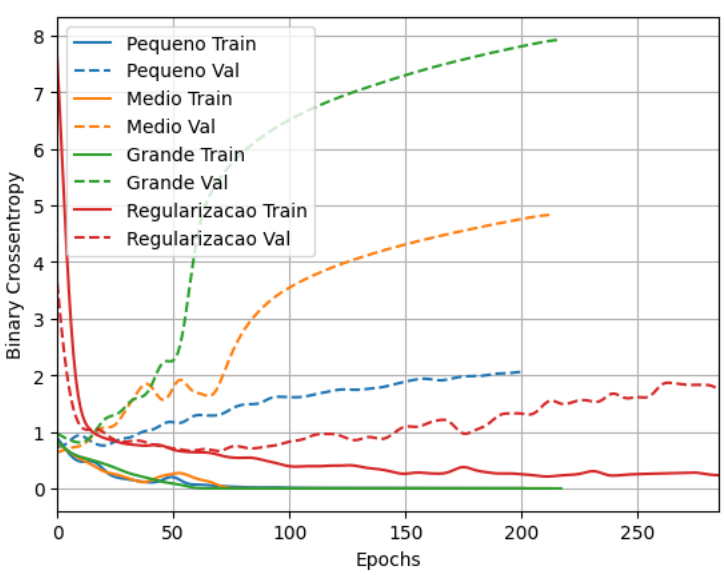
```
# Aplicando regularização (L2 + dropout) ao modelo grande
modelo_com_regularizacao = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(64, 64, 3)),
    tf.keras.layers.Dense(1024, kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])
cat_histories['Regularizacao'] = compile_and_fit(modelo_com_regularizacao, 'sizes/Regularizacao')
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_6 (Flatten)         (None, 12288)             0

 dense_24 (Dense)            (None, 1024)              12583936

 dropout_4 (Dropout)         (None, 1024)              0

 dense_25 (Dense)            (None, 1024)              1049600

 dropout_5 (Dropout)         (None, 1024)              0

 dense_26 (Dense)            (None, 1024)              1049600

 dropout_6 (Dropout)         (None, 1024)              0

 dense_27 (Dense)            (None, 1024)              1049600

 dropout_7 (Dropout)         (None, 1024)              0

 dense_28 (Dense)            (None, 1)                 1025
```

```
=================================================================
Total params: 15733761 (60.02 MB)
Trainable params: 15733761 (60.02 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Antes do treinamento - Dimensões dos dados de treinamento: (209, 64, 64, 3)
Antes do treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)

Epoch: 0, accuracy:0.5455,  binary_crossentropy:16.6295,  loss:17.1431,  val_accuracy:0.6600,  val_binary_crossentropy:13.1231,  val
................................................................................................
Epoch: 100, accuracy:0.7955,  binary_crossentropy:0.3941,  loss:0.8516,  val_accuracy:0.5000,  val_binary_crossentropy:1.0096,  val_
................................................................................................
Epoch: 200, accuracy:0.9000,  binary_crossentropy:0.2488,  loss:0.6728,  val_accuracy:0.7600,  val_binary_crossentropy:1.0890,  val_
.....................................................................................Após o treinamento - Dimensões dos dados de tr
Após o treinamento - Dimensões dos dados de teste: (50, 64, 64, 3)
```

Resultados do treinamento

```
plotter.plot(cat_histories)
# plt.ylim([0.5, 0.7])
```



ToDo: Análise dos resultados (10pt)

Avalia os modelos treinados quanto a Acurácia, F1-score, Precisão e revocação.

Dica: utilize a função `classification_report` da sklearn.

```
from sklearn.metrics import classification_report

# Predições no conjunto de teste
test_predictions = modelo_com_regularizacao.predict(teste_x)

# Convertemos as probabilidades em rótulos binários (0 ou 1)
binary_predictions = (test_predictions > 0.5).astype(int)

# Ajuste para garantir a compatibilidade das dimensões
teste_y_flat = teste_y.flatten()

# Relatório de classificação
report = classification_report(teste_y_flat, binary_predictions)

# Exibindo o relatório
print(report)
```

```
2/2 [==============================] - 0s 37ms/step
              precision    recall  f1-score   support
```