Lab 3 - BCC406

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Regressão Logística e Rede Neural

Prof. Eduardo e Prof. Pedro

Objetivos:

- *Overfitting*
- Regularização

Data da entrega : 26/08

- 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 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/HIGG
```

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).

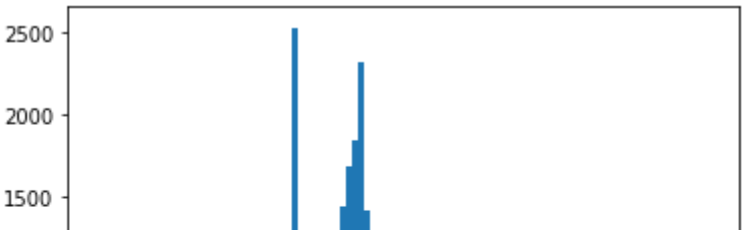
O TensorFlow é mais eficiente ao operar em grandes lotes de dados. Portanto, em vez de reempacotar cada linha individualmente, criaremos um novo conjunto de Dataset que receba lotes de 10.000 exemplos, aplique a função `pack_row` a cada lote e, em seguida, divida os lotes em registros individuais:

```
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

Configurando o treinamento de modelos

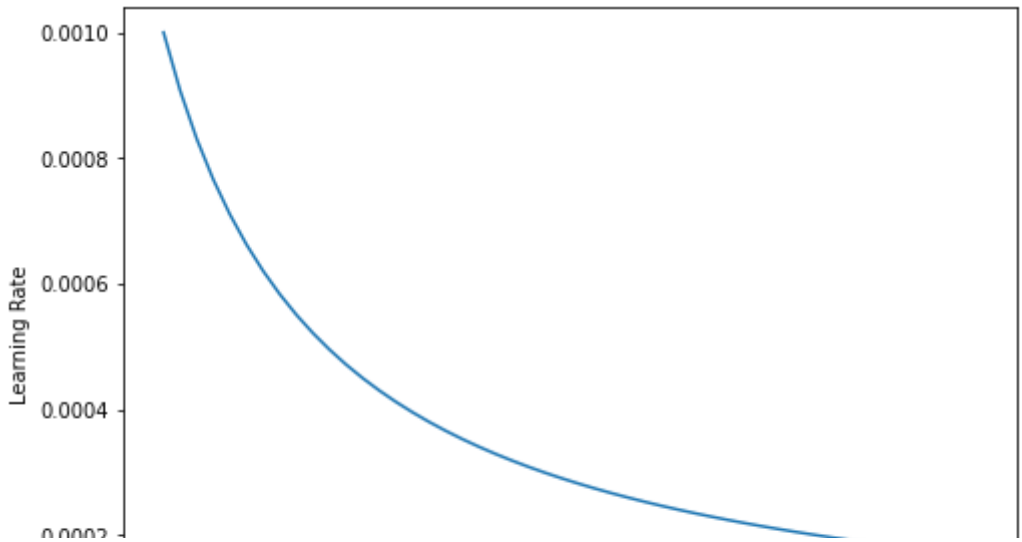
Nesta prática iremos avaliar diversos modelos. Para que a análise seja facilitada, usaremos a mesma configuração para todos os modelos.

- 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()
```

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

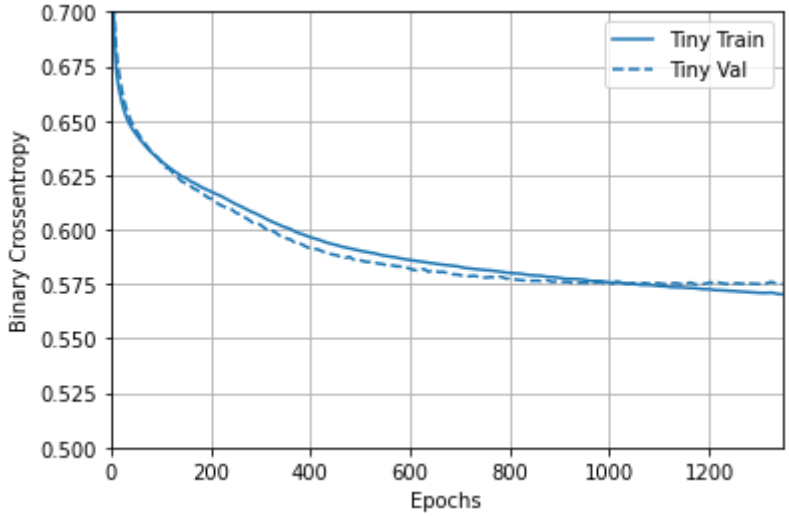
```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_20 (Dense)            (None, 16)                464

 dense_21 (Dense)            (None, 1)                 17

=================================================================
Total params: 481
Trainable params: 481
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.4969,  binary_crossentropy:0.9298,  loss:0.9298,  val_accuracy:
.....................................................................................
Epoch: 100, accuracy:0.5918,  binary_crossentropy:0.6316,  loss:0.6316,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6181,  binary_crossentropy:0.6179,  loss:0.6179,  val_accurac
.....................................................................................
Epoch: 300, accuracy:0.6347,  binary_crossentropy:0.6060,  loss:0.6060,  val_accurac
.....................................................................................
Epoch: 400, accuracy:0.6470,  binary_crossentropy:0.5961,  loss:0.5961,  val_accurac
```

```
.....................................................................................
Epoch: 500, accuracy:0.6549,  binary_crossentropy:0.5902,  loss:0.5902,  val_accurac
.....................................................................................
Epoch: 600, accuracy:0.6598,  binary_crossentropy:0.5864,  loss:0.5864,  val_accurac
.....................................................................................
Epoch: 700, accuracy:0.6656,  binary_crossentropy:0.5826,  loss:0.5826,  val_accurac
.....................................................................................
Epoch: 800, accuracy:0.6755,  binary_crossentropy:0.5802,  loss:0.5802,  val_accurac
.....................................................................................
Epoch: 900, accuracy:0.6752,  binary_crossentropy:0.5777,  loss:0.5777,  val_accurac
.....................................................................................
Epoch: 1000, accuracy:0.6754,  binary_crossentropy:0.5758,  loss:0.5758,  val_accura
.....................................................................................
Epoch: 1100, accuracy:0.6808,  binary_crossentropy:0.5741,  loss:0.5741,  val_accura
.....................................................................................
Epoch: 1200, accuracy:0.6783,  binary_crossentropy:0.5725,  loss:0.5725,  val_accura
.....................................................................................
Epoch: 1300, accuracy:0.6826,  binary_crossentropy:0.5710,  loss:0.5710,  val_accura
.........................................
```

Verificando o modelo

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

```
(0.5, 0.7)
```



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)
])
```

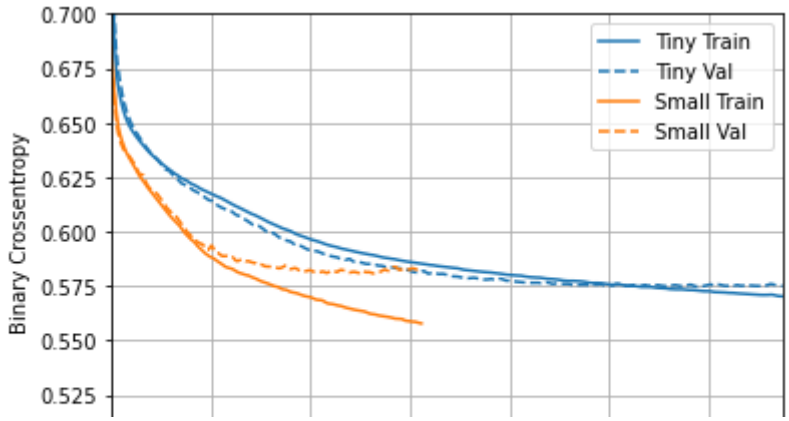
```
size_histories['Small'] = compile_and_fit(small_model, 'sizes/Small')
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_22 (Dense)            (None, 16)                464

 dense_23 (Dense)            (None, 16)                272

 dense_24 (Dense)            (None, 1)                 17

=================================================================
Total params: 753
Trainable params: 753
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.4870,  binary_crossentropy:0.7106,  loss:0.7106,  val_accuracy:
.....................................................................................
Epoch: 100, accuracy:0.6261,  binary_crossentropy:0.6121,  loss:0.6121,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6618,  binary_crossentropy:0.5878,  loss:0.5878,  val_accurac
.....................................................................................
Epoch: 300, accuracy:0.6771,  binary_crossentropy:0.5776,  loss:0.5776,  val_accurac
.....................................................................................
Epoch: 400, accuracy:0.6864,  binary_crossentropy:0.5705,  loss:0.5705,  val_accurac
.....................................................................................
Epoch: 500, accuracy:0.6954,  binary_crossentropy:0.5635,  loss:0.5635,  val_accurac
.....................................................................................
Epoch: 600, accuracy:0.6945,  binary_crossentropy:0.5590,  loss:0.5590,  val_accurac
.......................
```

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)
])
```

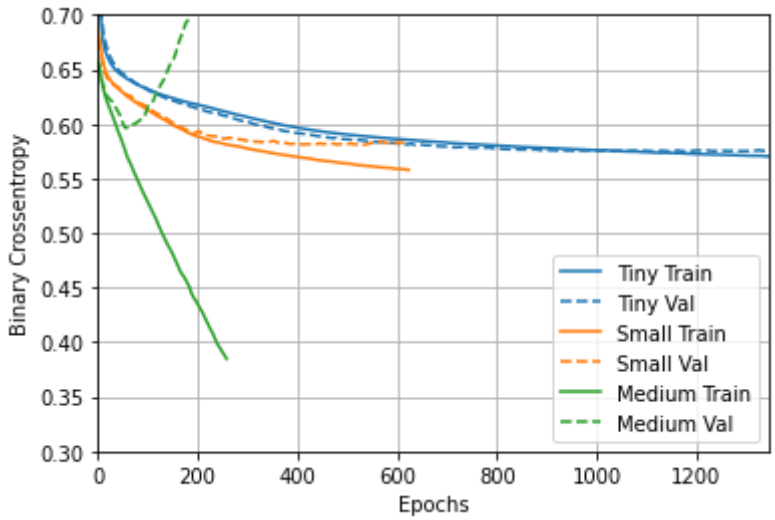
```
size_histories['Medium']  = compile_and_fit(medium_model, "sizes/Medium")
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_25 (Dense)            (None, 64)                1856

 dense_26 (Dense)            (None, 64)                4160

 dense_27 (Dense)            (None, 64)                4160

 dense_28 (Dense)            (None, 1)                 65

=================================================================
Total params: 10,241
Trainable params: 10,241
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5008,  binary_crossentropy:0.6839,  loss:0.6839,  val_accuracy:
................................................................................
Epoch: 100, accuracy:0.7120,  binary_crossentropy:0.5288,  loss:0.5288,  val_accurac
................................................................................
Epoch: 200, accuracy:0.7811,  binary_crossentropy:0.4325,  loss:0.4325,  val_accurac
................................................
```

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_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_29 (Dense)            (None, 512)               14848

 dense_30 (Dense)            (None, 512)               262656

 dense_31 (Dense)            (None, 512)               262656

 dense_32 (Dense)            (None, 512)               262656
```

```
 dense_33 (Dense)            (None, 1)                 513

=================================================================
Total params: 803,329
Trainable params: 803,329
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5051,  binary_crossentropy:0.7577,  loss:0.7577,  val_accuracy:
................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0018,  loss:0.0018,  val_accurac
................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0001,  loss:0.0001,  val_accurac
......................
```

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.

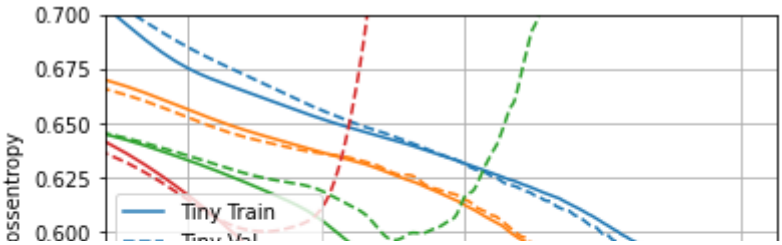
Neste exemplo, normalmente, apenas o modelo minúsculo (`"Tiny"`) consegue evitar completamente o overfitting, e cada um dos modelos maiores superajusta os dados mais rapidamente. Isso se torna tão grave para o modelo grande (`"Large"`) que você precisa mudar o gráfico para uma escala logarítmica para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

```
big_large_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(1024, activation='elu'),
    layers.Dense(1)
])
```

```
size_histories['big_large'] = compile_and_fit(big_large_model, "sizes/big_large")
```

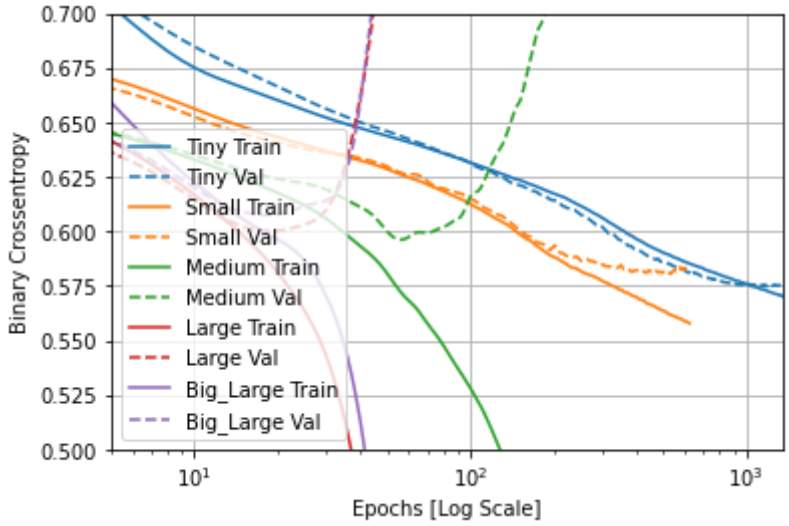
```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_40 (Dense)            (None, 1024)              29696

 dense_41 (Dense)            (None, 1024)              1049600

 dense_42 (Dense)            (None, 1024)              1049600

 dense_43 (Dense)            (None, 1024)              1049600

 dense_44 (Dense)            (None, 1024)              1049600

 dense_45 (Dense)            (None, 1)                 1025

=================================================================
Total params: 4,229,121
Trainable params: 4,229,121
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5056,  binary_crossentropy:1.0540,  loss:1.0540,  val_accuracy:
.....................................................................................
```

```
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0003,  loss:0.0003,  val_accurac
.....................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0000,  loss:0.0000,  val_accurac
.............................
```

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

```
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]")
```

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



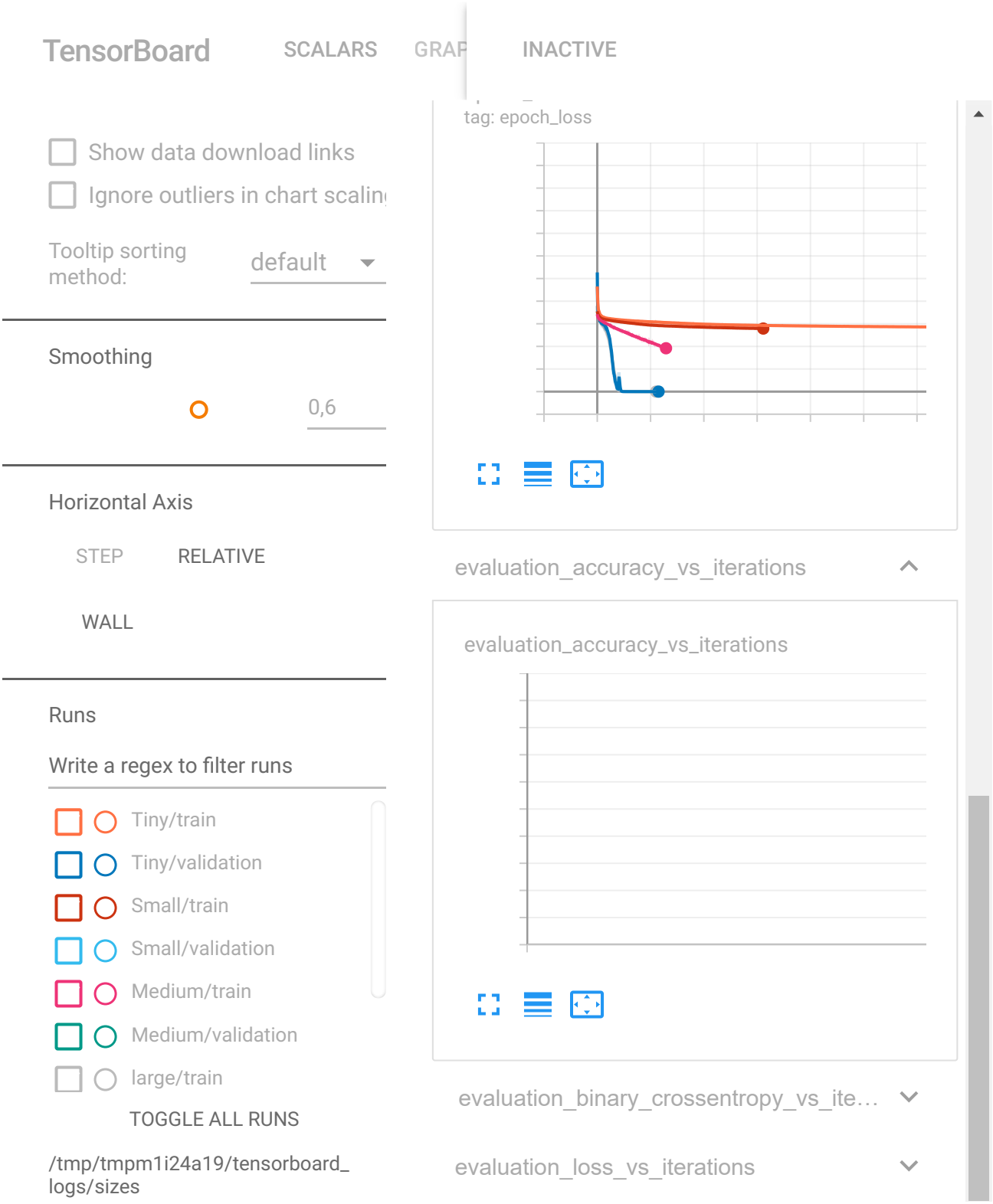
Percebe-se que o modelo Big_Large também sobreajustou aos dados. Portanto, só o modelo Tiny evitou esse sobreajuste.

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

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



ToDo: Análises (10pt)

Quais análises você pode fazer sobre o TensorBoard?

Modelos com maiores acurácias no Treino em ordem decrescente: large, big_large, medium, small, tiny. Modelos com maiores acurácias na Validação em ordem decrescente: big_large, tiny, small, large e medium. Analisando a binary_crossentropy: todos os modelos, exceto o Tiny sobreajustou os dados (overfitting), pois a curva de custo de treinamento e de validação seguiram na mesma direção. E como a diferença entre as

acurácias é muito pequena na validação, o modelo Tiny, apesar de ser pequeno, é o melhor modelo para fazer a predição dos dados em relação aos outros modelos treinados

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/tmpm1i24a19/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_s
    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_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_46 (Dense)            (None, 512)               14848

 dense_47 (Dense)            (None, 512)               262656

 dense_48 (Dense)            (None, 512)               262656

 dense_49 (Dense)            (None, 512)               262656

 dense_50 (Dense)            (None, 1)                 513

=================================================================
Total params: 803,329
Trainable params: 803,329
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.4936,  binary_crossentropy:0.8380,  loss:2.3554,  val_accuracy:
.....................................................................................
Epoch: 100, accuracy:0.6613,  binary_crossentropy:0.5951,  loss:0.6185,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6697,  binary_crossentropy:0.5840,  loss:0.6058,  val_accurac
.....................................................................................
Epoch: 300, accuracy:0.6798,  binary_crossentropy:0.5779,  loss:0.5995,  val_accurac
.....................................................................................
Epoch: 400, accuracy:0.6825,  binary_crossentropy:0.5721,  loss:0.5943,  val_accurac
.....................................................................................
Epoch: 500, accuracy:0.6846,  binary_crossentropy:0.5662,  loss:0.5876,  val_accurac
.....................................................................................
Epoch: 600, accuracy:0.6954,  binary_crossentropy:0.5602,  loss:0.5828,  val_accurac
.........
```

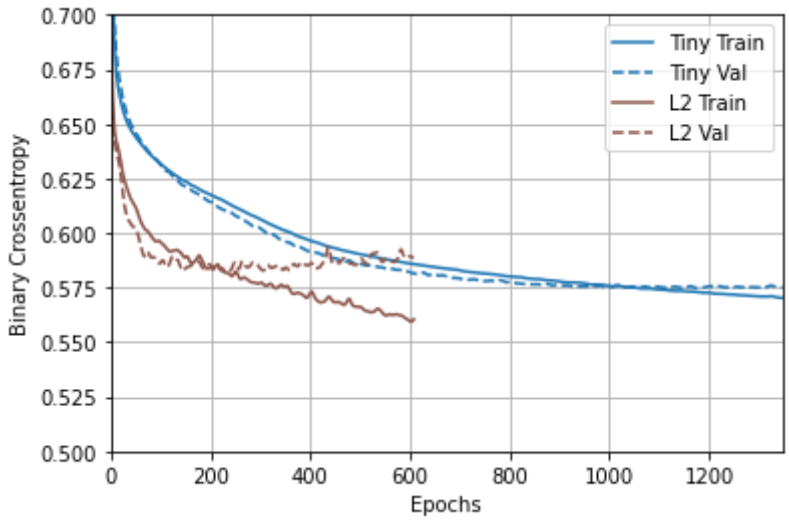
`l2(0,001)` significa que cada coeficiente na matriz de peso da camada adicionará `0,001 * weight_coeficiente_value**2` ao total de **perda** da rede.

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:

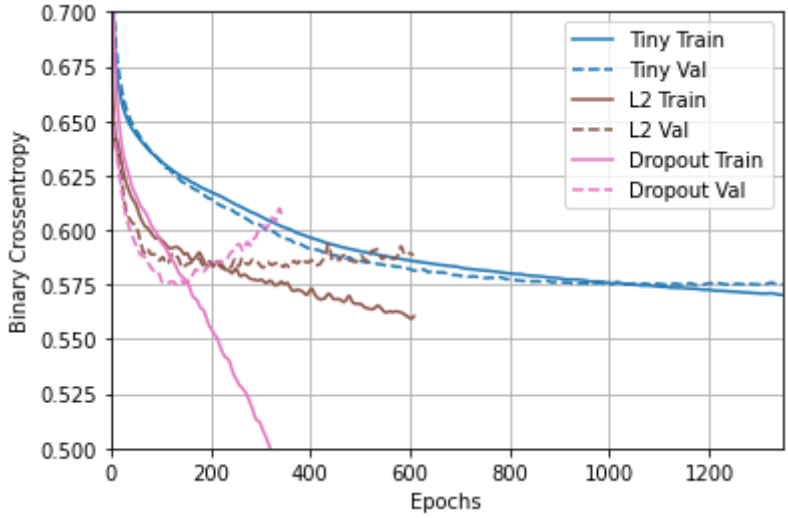
```
dropout_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),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

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

```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_51 (Dense)            (None, 512)               14848

 dropout (Dropout)           (None, 512)               0

 dense_52 (Dense)            (None, 512)               262656

 dropout_1 (Dropout)         (None, 512)               0

 dense_53 (Dense)            (None, 512)               262656

 dropout_2 (Dropout)         (None, 512)               0

 dense_54 (Dense)            (None, 512)               262656

 dropout_3 (Dropout)         (None, 512)               0

 dense_55 (Dense)            (None, 1)                 513

=================================================================
Total params: 803,329
Trainable params: 803,329
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5102,  binary_crossentropy:0.7869,  loss:0.7869,  val_accuracy:
```

```
.....................................................................................
Epoch: 100, accuracy:0.6610,  binary_crossentropy:0.5958,  loss:0.5958,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6900,  binary_crossentropy:0.5556,  loss:0.5556,  val_accurac
.....................................................................................
Epoch: 300, accuracy:0.7179,  binary_crossentropy:0.5121,  loss:0.5121,  val_accurac
........................................
```

```
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*

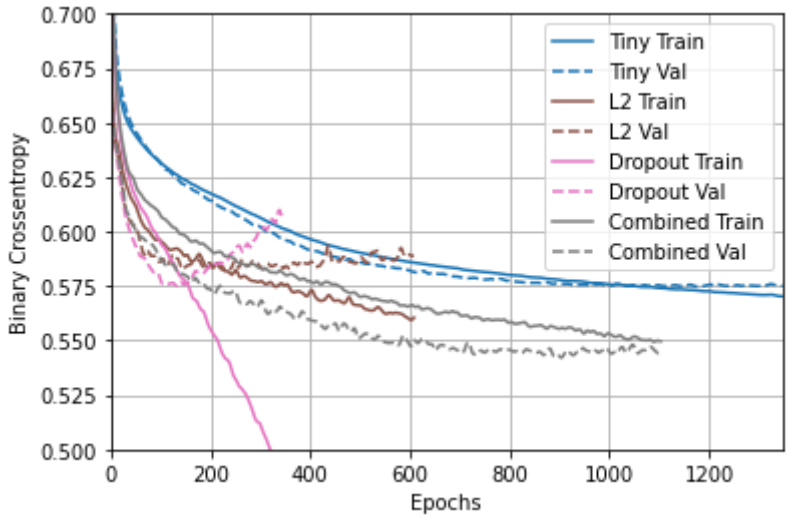
```
combined_model = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
   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

```
Model: "sequential_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_56 (Dense)            (None, 512)               14848

 dropout_4 (Dropout)         (None, 512)               0

 dense_57 (Dense)            (None, 512)               262656

 dropout_5 (Dropout)         (None, 512)               0

 dense_58 (Dense)            (None, 512)               262656

 dropout_6 (Dropout)         (None, 512)               0

 dense_59 (Dense)            (None, 512)               262656

 dropout_7 (Dropout)         (None, 512)               0

 dense_60 (Dense)            (None, 1)                 513

=================================================================
Total params: 803,329
Trainable params: 803,329
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5035,  binary_crossentropy:0.8138,  loss:0.9722,  val_accuracy:
.....................................................................................
Epoch: 100, accuracy:0.6463,  binary_crossentropy:0.6059,  loss:0.6352,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6553,  binary_crossentropy:0.5936,  loss:0.6183,  val_accurac
.....................................................................................
Epoch: 300, accuracy:0.6655,  binary_crossentropy:0.5847,  loss:0.6127,  val_accurac
.....................................................................................
Epoch: 400, accuracy:0.6746,  binary_crossentropy:0.5822,  loss:0.6121,  val_accurac
.....................................................................................
Epoch: 500, accuracy:0.6834,  binary_crossentropy:0.5705,  loss:0.6025,  val_accurac
.....................................................................................
Epoch: 600, accuracy:0.6797,  binary_crossentropy:0.5668,  loss:0.6008,  val_accurac
.....................................................................................
Epoch: 700, accuracy:0.6910,  binary_crossentropy:0.5622,  loss:0.5974,  val_accurac
.....................................................................................
Epoch: 800, accuracy:0.6928,  binary_crossentropy:0.5579,  loss:0.5942,  val_accurac
.....................................................................................
Epoch: 900, accuracy:0.6954,  binary_crossentropy:0.5530,  loss:0.5906,  val_accurac
.....................................................................................
Epoch: 1000, accuracy:0.6897,  binary_crossentropy:0.5534,  loss:0.5921,  val_accura
.....................................................................................
Epoch: 1100, accuracy:0.6997,  binary_crossentropy:0.5470,  loss:0.5869,  val_accura
....
```

```
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.

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?

Modelos com maiores acurácias no Treino em ordem decrescente: dropout, combined, l2, Tiny. Modelos com maiores acurácias na Validação em ordem decrescente: combined, dropout, l2, Tiny. Analisando a binary_crossentropy: todos os modelos, exceto o Tiny e Combined sobreajustou os dados (overfitting), pois a curva de custo de treinamento e de validação seguiram na mesma direção. Sendo que o modelo Combined foi o melhor modelo, pois além de não sobreajustar os dados, possui um maior valor de acurácia tanto no Treino quanto na Validação, além de assumir um menor valor na função de custo(loss e binary_crossentropy).

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)

```
# Para Google Colab: é necessário fazer o upload dos arquivos no drive e montá-lo
import h5py
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.
```

```
# Lendo os dados (gato/não-gato)
def load_dataset():

  """train_ds = h5py.File('train_catvnoncat.h5', "r")
  train_ds = zip(train_ds["train_set_x"][:], train_ds["train_set_y"][:])  # your train set


  validate_ds = h5py.File('test_catvnoncat.h5', "r")
  validate_ds = zip(validate_ds["test_set_x"][:], validate_ds["test_set_y"][:]) # your tes

  return train_ds, validate_ds
  """

  train_dataset = h5py.File('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('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


# Lendo os dados (gato/não-gato)
"""train_ds, validate_ds = load_dataset()"""
treino_x_orig, treino_y, teste_x_orig, teste_y, classes = load_dataset()
```

```
# Vetorizando a imagem

m_treino = len(treino_x_orig)
m_teste = len(teste_x_orig)
num_px = teste_x_orig[1].shape[1]


#  Vetorizando as imagens de treinamento e teste

### Início do código ###
treino_x = treino_x_orig.reshape((m_treino, num_px*num_px*3)) # dica : utilize reshape par
### Fim do código ###

### Início do código ###
# Normalize os dados para ter valores de recurso entre 0 e 1.
teste_x = teste_x_orig.reshape((m_teste, num_px*num_px*3)) # dica : utilize reshape para m

### Fim do código ###


train_ds = tf.data.Dataset.from_tensor_slices((treino_x, treino_y.reshape(-1)))
validate_ds = tf.data.Dataset.from_tensor_slices((teste_x, teste_y.reshape(-1)))


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()

  history = model.fit(
    treino_x, treino_y.reshape(-1),
    epochs=max_epochs,
    validation_data=(teste_x, teste_y.reshape(-1)),
    callbacks=get_callbacks(name),
    verbose=0)
  return history
```

ToDo: Treinando um modelo pequeno (10pt)

```
modelo_pequeno = tf.keras.Sequential([
    ## Seu código aqui
    layers.Dense(16, activation='elu', input_shape=(12288,)),
    layers.Dense(16, activation='elu'),
    layers.Dense(1)
])
```

```
cat_histories['Pequeno'] = compile_and_fit(modelo_pequeno, 'sizes/Pequeno')
```

```
Model: "sequential_48"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_168 (Dense)           (None, 16)                196624

 dense_169 (Dense)           (None, 16)                272

 dense_170 (Dense)           (None, 1)                 17

=================================================================
Total params: 196,913
Trainable params: 196,913
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5311,  binary_crossentropy:32.6600,  loss:34.3835,  val_accurac
................................................................................
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0004,  loss:0.0004,  val_accurac
................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0002,  loss:0.0003,  val_accurac
...........
```

ToDo: Treinando um modelo médio (10pt)

```
modelo_medio = tf.keras.Sequential([
    layers.Dense(64, activation='elu', input_shape=(12288,)),
    layers.Dense(64, activation='elu'),
    layers.Dense(64, activation='elu'),
    layers.Dense(1)
])
cat_histories['Medio'] = compile_and_fit(modelo_medio, 'sizes/Medio')
```

```
Model: "sequential_49"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_171 (Dense)           (None, 64)                786496

 dense_172 (Dense)           (None, 64)                4160

 dense_173 (Dense)           (None, 64)                4160

 dense_174 (Dense)           (None, 1)                 65

=================================================================
Total params: 794,881
Trainable params: 794,881
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5598,  binary_crossentropy:135.9922,  loss:142.9003,  val_accur
```

```
.....................................................................................
Epoch: 100, accuracy:0.9952,  binary_crossentropy:0.0095,  loss:0.0101,  val_accurac
.....................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0000,  loss:0.0000,  val_accurac
..........
```

ToDo: Treinando um modelo grande (10pt)

```
modelo_grande = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(12288,)),
    layers.Dense(512, activation='elu'),
    layers.Dense(512, activation='elu'),
    layers.Dense(512, activation='elu'),
    layers.Dense(1)
])
cat_histories['Grande'] = compile_and_fit(modelo_grande, 'sizes/Grande')
```

```
Model: "sequential_50"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_175 (Dense)           (None, 512)               6291968

 dense_176 (Dense)           (None, 512)               262656

 dense_177 (Dense)           (None, 512)               262656

 dense_178 (Dense)           (None, 512)               262656

 dense_179 (Dense)           (None, 1)                 513

=================================================================
Total params: 7,080,449
Trainable params: 7,080,449
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5407,  binary_crossentropy:537.4789,  loss:558.7018,  val_accur
.....................................................................................
Epoch: 100, accuracy:0.9856,  binary_crossentropy:0.0387,  loss:0.0374,  val_accurac
.....................................................................................
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0001,  loss:0.0001,  val_accurac
.............................................................
```

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

```
modelo_pequeno_regularizado = tf.keras.Sequential([
    ## Seu código aqui
    layers.Dense(16, kernel_regularizer=regularizers.l2(0.0001), activation='elu', input_s
    layers.Dropout(0.5),
```

```
    layers.Dense(16, kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

cat_histories['pequeno_regularizado'] = compile_and_fit(modelo_pequeno_regularizado, 'size
```

```
Model: "sequential_51"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_180 (Dense)           (None, 16)                196624

 dropout_15 (Dropout)        (None, 16)                0

 dense_181 (Dense)           (None, 16)                272

 dropout_16 (Dropout)        (None, 16)                0

 dense_182 (Dense)           (None, 1)                 17

=================================================================
Total params: 196,913
Trainable params: 196,913
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.4545,  binary_crossentropy:118.7528,  loss:127.0737,  val_accur
.....................................................................................
Epoch: 100, accuracy:0.6077,  binary_crossentropy:0.8056,  loss:0.8084,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6364,  binary_crossentropy:0.7195,  loss:0.7231,  val_accurac
......
```

```
modelo_medio_regularizado = tf.keras.Sequential([
    layers.Dense(64, kernel_regularizer=regularizers.l2(0.0001), activation='elu', input_s
    layers.Dropout(0.5),
    layers.Dense(64, kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(64,kernel_regularizer=regularizers.l2(0.0001), activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])
cat_histories['medio_regularizado'] = compile_and_fit(modelo_medio_regularizado, 'sizes/me
```

```
Model: "sequential_52"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_183 (Dense)           (None, 64)                786496

 dropout_17 (Dropout)        (None, 64)                0

 dense_184 (Dense)           (None, 64)                4160
```

```
 dropout_18 (Dropout)        (None, 64)                0

 dense_185 (Dense)           (None, 64)                4160

 dropout_19 (Dropout)        (None, 64)                0

 dense_186 (Dense)           (None, 1)                 65

=================================================================
Total params: 794,881
Trainable params: 794,881
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5167,  binary_crossentropy:491.7466,  loss:488.1154,  val_accur
................................................................................
Epoch: 100, accuracy:0.6220,  binary_crossentropy:0.7790,  loss:0.8123,  val_accurac
................................................................................
Epoch: 200, accuracy:0.5933,  binary_crossentropy:0.8334,  loss:0.8653,  val_accurac
...
```

```
modelo_grande_regularizado = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001), activation='elu', input_
    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)
])
cat_histories['grande_regularizado'] = compile_and_fit(modelo_grande_regularizado, 'sizes/
```

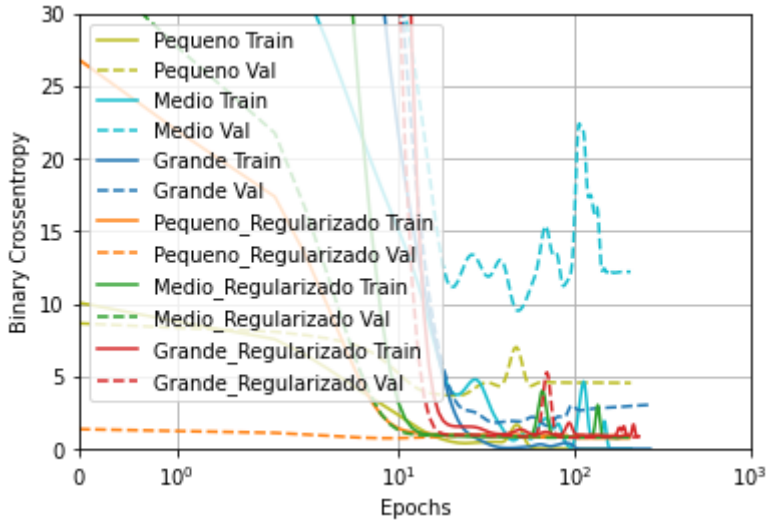
```
Model: "sequential_53"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_187 (Dense)           (None, 512)               6291968

 dropout_20 (Dropout)        (None, 512)               0

 dense_188 (Dense)           (None, 512)               262656

 dropout_21 (Dropout)        (None, 512)               0

 dense_189 (Dense)           (None, 512)               262656

 dropout_22 (Dropout)        (None, 512)               0

 dense_190 (Dense)           (None, 512)               262656

 dropout_23 (Dropout)        (None, 512)               0

 dense_191 (Dense)           (None, 1)                 513
```

```
=================================================================
Total params: 7,080,449
Trainable params: 7,080,449
Non-trainable params: 0
_________________________________________________________________

Epoch: 0, accuracy:0.5024,  binary_crossentropy:778.6417,  loss:724.5026,  val_accur
.....................................................................................
Epoch: 100, accuracy:0.6268,  binary_crossentropy:0.8975,  loss:1.1668,  val_accurac
.....................................................................................
Epoch: 200, accuracy:0.6172,  binary_crossentropy:0.8599,  loss:1.1126,  val_accurac
................................
```

Resultados do treinamento

```
plotter.plot(cat_histories)
a = plt.xscale('symlog')
plt.ylim([0, 30])
plt.xlim([0, 1000])
plt.xlabel("Epochs")
```

```
Text(0.5, 0, 'Epochs')
```



É possível perceber que os modelos regularizados são os que mais generalizam os dados, evitando o overfitting.

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.

O que você pode inferir analisando os resultados dos treinamentos para a base de gatos/não-gatos.

Observando as métricas abaixo é possível perceber que os modelos não possuem um taxa de acurácia boa. Todos os modelos estão chutando como não gato.

```
from sklearn.metrics import classification_report


y_pred = modelo_pequeno.predict(teste_x, verbose=1)
y_pred_bool = np.argmax(y_pred,·axis=1)

print(list(y_pred_bool))
print(list(teste_y.reshape(-1)))

print(classification_report(teste_y.reshape(-1), y_pred_bool))
```

```
2/2 [==============================] - 0s 7ms/step
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
[1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0,
              precision    recall  f1-score   support

           0       0.34      1.00      0.51        17
           1       0.00      0.00      0.00        33

    accuracy                           0.34        50
   macro avg       0.17      0.50      0.25        50
weighted avg       0.12      0.34      0.17        50

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

```
y_pred = modelo_medio.predict(teste_x, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(list(y_pred_bool))
print(list(teste_y.reshape(-1)))

print(classification_report(teste_y.reshape(-1), y_pred_bool))
```

```
2/2 [==============================] - 0s 11ms/step
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
[1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0,
              precision    recall  f1-score   support

           0       0.34      1.00      0.51        17
           1       0.00      0.00      0.00        33

    accuracy                           0.34        50
   macro avg       0.17      0.50      0.25        50
weighted avg       0.12      0.34      0.17        50
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

```
y_pred = modelo_grande.predict(teste_x, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(list(y_pred_bool))
print(list(teste_y.reshape(-1)))

print(classification_report(teste_y.reshape(-1), y_pred_bool))
```

```
2/2 [==============================] - 0s 16ms/step
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
[1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0,
              precision    recall  f1-score   support

           0       0.34      1.00      0.51        17
           1       0.00      0.00      0.00        33

    accuracy                           0.34        50
   macro avg       0.17      0.50      0.25        50
weighted avg       0.12      0.34      0.17        50

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

```
y_pred = modelo_pequeno_regularizado.predict(teste_x, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(classification_report(teste_y.reshape(-1), y_pred_bool))
```

```
2/2 [==============================] - 0s 8ms/step
              precision    recall  f1-score   support

           0       0.34      1.00      0.51        17
           1       0.00      0.00      0.00        33

    accuracy                           0.34        50
   macro avg       0.17      0.50      0.25        50
weighted avg       0.12      0.34      0.17        50

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

```
y_pred = modelo_medio_regularizado.predict(teste_x, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(classification_report(teste_y.reshape(-1), y_pred_bool))
```

```
WARNING:tensorflow:5 out of the last 24 calls to <function Model.make_predict_functi
2/2 [==============================] - 0s 11ms/step
              precision    recall  f1-score   support

           0       0.34      1.00      0.51        17
           1       0.00      0.00      0.00        33

    accuracy                           0.34        50
   macro avg       0.17      0.50      0.25        50
weighted avg       0.12      0.34      0.17        50

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

```
y_pred = modelo_grande_regularizado.predict(teste_x, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(list(y_pred_bool))
print(list(teste_y.reshape(-1)))

print(classification_report(teste_y.reshape(-1), y_pred_bool))
```

```
2/2 [==============================] - 0s 14ms/step
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
[1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0,
              precision    recall  f1-score   support

           0       0.34      1.00      0.51        17
           1       0.00      0.00      0.00        33

    accuracy                           0.34        50
   macro avg       0.17      0.50      0.25        50
weighted avg       0.12      0.34      0.17        50

/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

Produtos pagos do Colab - Cancele os contratos aqui

✓ 0 s concluído à(s) 19:43