Advanced Logistic Regression in TensorFlow 2.0

Learning Objectives

1. Load a CSV file using Pandas
2. Create train, validation, and test sets
3. Define and train a model using Keras (including setting class weights)
4. Evaluate the model using various metrics (including precision and recall)
5. Try common techniques for dealing with imbalanced data: Class weighting and Oversampling

Introduction

This lab how to classify a highly imbalanced dataset in which the number of examples in one class greatly outnumbers the examples in another. You will work with the Credit Card Fraud Detection dataset hosted on Kaggle. The aim is to detect a mere 492 fraudulent transactions from 284,807 transactions in total. You will use Keras to define the model and class weights to help the model learn from the imbalanced data.

PENDING LINK UPDATE: Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

Start by importing the necessary libraries for this lab.

In [52]:

```
# You can use any Python source file as a module by executing an import statement in so
# The import statement combines two operations; it searches for the named module, then
# results of that search to a name in the local scope.
import tensorflow as tf
from tensorflow import keras

import os
import tempfile

# Use matplotlib for visualizing the model
import matplotlib as mpl
import matplotlib.pyplot as plt
# Here we'll import Pandas and Numpy data processing libraries
import numpy as np
import pandas as pd
# Use seaborn for data visualization
import seaborn as sns

# Scikit-learn is an open source machine learning library that supports supervised and
import sklearn
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

print("TensorFlow version: ",tf.version.VERSION)
```

```
TensorFlow version:  2.1.0
```

In the next cell, we're going to customize our Matplot lib visualization figure size and colors. Note that each time Matplotlib loads, it defines a runtime configuration (rc) containing the default styles for every plot element we create. This configuration can be adjusted at any time using the plt.rc convenience routine.

In [53]:

```
# Customize our Matplot lib visualization figure size and colors
mpl.rcParams['figure.figsize'] = (12, 10)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

Data processing and exploration

Download the Kaggle Credit Card Fraud data set

Pandas is a Python library with many helpful utilities for loading and working with structured data and can be used to download CSVs into a dataframe.

Note: This dataset has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group of ULB (Université Libre de Bruxelles) on big data mining and fraud detection. More details on current and past projects on related topics are available here and the page of the DefeatFraud project

In [54]:

```
file = tf.keras.utils
# pandas module read_csv() function reads the CSV file into a DataFrame object.
raw_df = pd.read_csv('https://storage.googleapis.com/download.tensorflow.org/data/credi
# `head()` function is used to get the first n rows of dataframe
raw_df.head()
```

Out[54]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739

5 rows × 31 columns

Now, let's view the statistics of the raw dataframe.

In [4]:

```
# describe() is used to view some basic statistical details
raw_df[['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V26', 'V27', 'V28', 'Amount', 'Class']].
```

Out[4]:

	Time	V1	V2	V3	V4	V5

	Time	V1	V2	V3	V4	V5	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.84
mean	94813.859575	1.165980e-15	3.416908e-16	-1.373150e-15	2.086869e-15	9.604066e-16	1.6
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	4.8
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.6(
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-3.2
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-5.2
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	2.4
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	3.51

Examine the class label imbalance

Let's look at the dataset imbalance:

In [55]:

```
# Numpy bincount() method is used to obtain the frequency of each element provided insi
neg, pos = np.bincount(raw_df['Class'])
total = neg + pos
print('Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total, pos, 100 * pos / total))
```

```
Examples:
    Total: 284807
    Positive: 492 (0.17% of total)
```

This shows the small fraction of positive samples.

Clean, split and normalize the data

The raw data has a few issues. First the `Time` and `Amount` columns are too variable to use directly. Drop the `Time` column (since it's not clear what it means) and take the log of the `Amount` column to reduce its range.

In [56]:

```
cleaned_df = raw_df.copy()

# You don't want the `Time` column.
cleaned_df.pop('Time')

# The `Amount` column covers a huge range. Convert to log-space.
eps=0.001 # 0 => 0.1¢
cleaned_df['Log Ammount'] = np.log(cleaned_df.pop('Amount')+eps)
```

Split the dataset into train, validation, and test sets. The validation set is used during the model fitting to evaluate the loss and any metrics, however the model is not fit with this data. The test set is completely unused during the training phase and is only used at the end to evaluate how well the model generalizes to new data. This is especially important with imbalanced datasets where overfitting is a significant concern from the lack of training data.

In [57]:

```
# TODO 1
# Use a utility from sklearn to split and shuffle our dataset.
# train_test_split() method split arrays or matrices into random train and test subsets
train_df, test_df = train_test_split(cleaned_df, test_size=0.2)
train_df, val_df = train_test_split(train_df, test_size=0.2)

# Form np arrays of labels and features.
train_labels = np.array(train_df.pop('Class'))
bool_train_labels = train_labels != 0
val_labels = np.array(val_df.pop('Class'))
test_labels = np.array(test_df.pop('Class'))

train_features = np.array(train_df)
val_features = np.array(val_df)
test_features = np.array(test_df)
```

Normalize the input features using the sklearn StandardScaler. This will set the mean to 0 and standard deviation to 1.

Note: The `StandardScaler` is only fit using the `train_features` to be sure the model is not peeking at the validation or test sets.

In [58]:

```
scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)

val_features = scaler.transform(val_features)
test_features = scaler.transform(test_features)

# `np.clip()` clip (limit) the values in an array.
train_features = np.clip(train_features, -5, 5)
val_features = np.clip(val_features, -5, 5)
test_features = np.clip(test_features, -5, 5)


print('Training labels shape:', train_labels.shape)
print('Validation labels shape:', val_labels.shape)
print('Test labels shape:', test_labels.shape)

print('Training features shape:', train_features.shape)
print('Validation features shape:', val_features.shape)
print('Test features shape:', test_features.shape)
```

```
Training labels shape: (182276,)
Validation labels shape: (45569,)
Test labels shape: (56962,)
Training features shape: (182276, 29)
Validation features shape: (45569, 29)
Test features shape: (56962, 29)
```

Caution: If you want to deploy a model, it's critical that you preserve the preprocessing calculations. The easiest way to implement them as layers, and attach them to your model before export.

Look at the data distribution

Next compare the distributions of the positive and negative examples over a few features. Good questions to ask yourself at this point are:

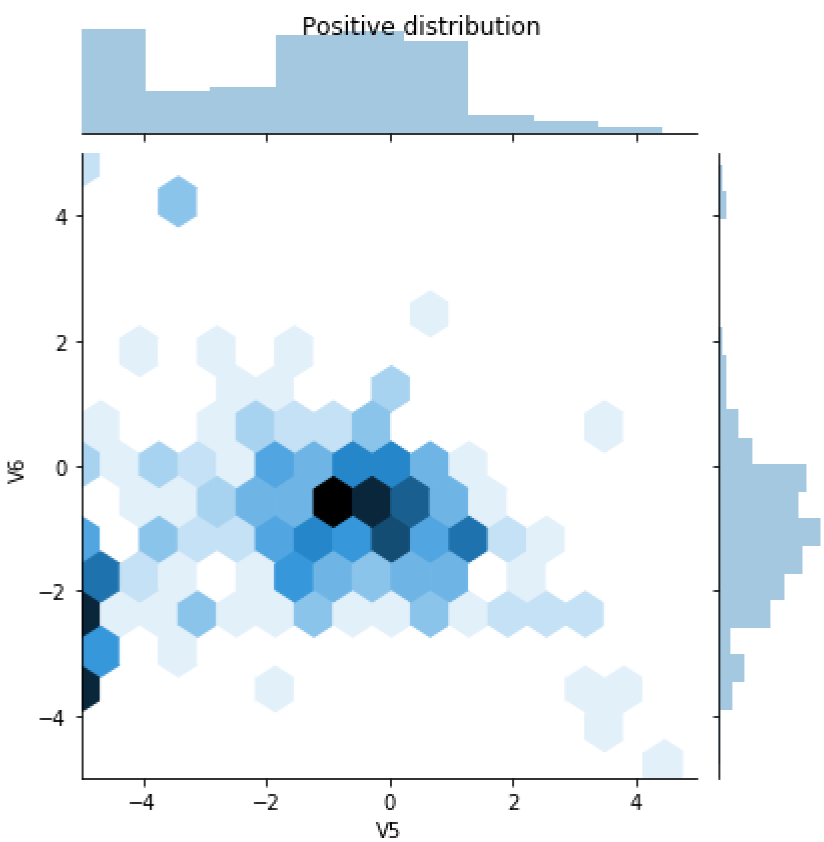
- Do these distributions make sense?
    - Yes. You've normalized the input and these are mostly concentrated in the `+/- 2` range.
- Can you see the difference between the ditributions?
    - Yes the positive examples contain a much higher rate of extreme values.

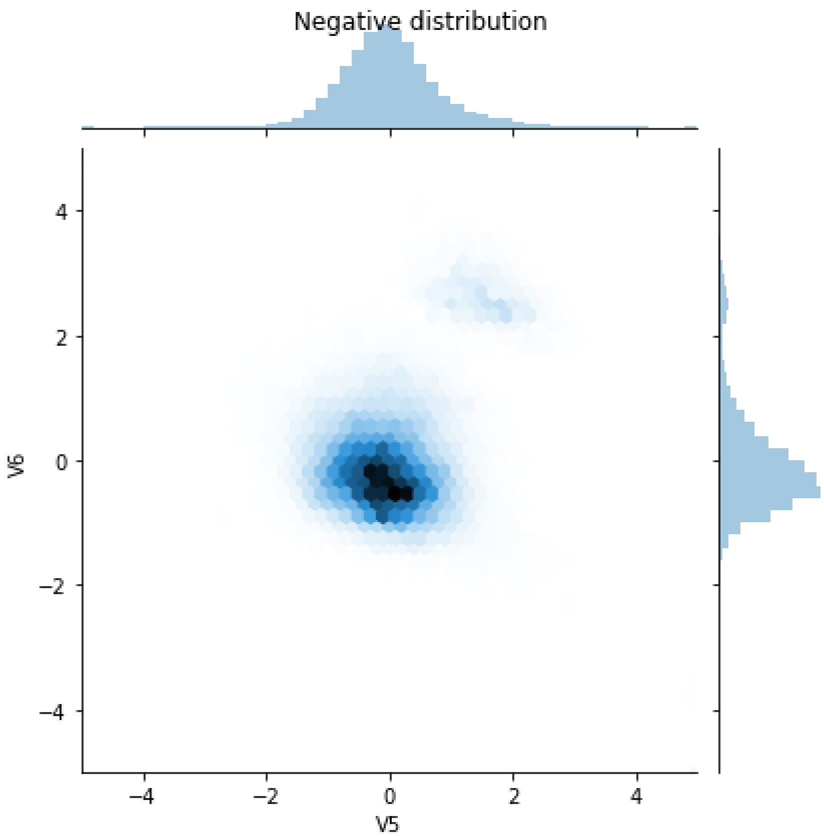
In [59]:

```
# pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular d
pos_df = pd.DataFrame(train_features[ bool_train_labels], columns = train_df.columns)
neg_df = pd.DataFrame(train_features[~bool_train_labels], columns = train_df.columns)

# Seaborn's jointplot displays a relationship between 2 variables (bivariate) as well a
sns.jointplot(pos_df['V5'], pos_df['V6'],
              kind='hex', xlim = (-5,5), ylim = (-5,5))
# The suptitle() function in pyplot module of the matplotlib library is used to add a t
plt.suptitle("Positive distribution")

sns.jointplot(neg_df['V5'], neg_df['V6'],
              kind='hex', xlim = (-5,5), ylim = (-5,5))
_ = plt.suptitle("Negative distribution")
```





Define the model and metrics

Define a function that creates a simple neural network with a densly connected hidden layer, a dropout layer to reduce overfitting, and an output sigmoid layer that returns the probability of a transaction being fraudulent:

In [60]:
```
METRICS = [
      keras.metrics.TruePositives(name='tp'),
      keras.metrics.FalsePositives(name='fp'),
      keras.metrics.TrueNegatives(name='tn'),
      keras.metrics.FalseNegatives(name='fn'),
      keras.metrics.BinaryAccuracy(name='accuracy'),
      keras.metrics.Precision(name='precision'),
      keras.metrics.Recall(name='recall'),
      keras.metrics.AUC(name='auc'),
]

def make_model(metrics = METRICS, output_bias=None):
  if output_bias is not None:
# `tf.keras.initializers.Constant()` generates tensors with constant values.
    output_bias = tf.keras.initializers.Constant(output_bias)
# TODO 1
# Creating a Sequential model
  model = keras.Sequential([
      keras.layers.Dense(
          16, activation='relu',
          input_shape=(train_features.shape[-1],)),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(1, activation='sigmoid',
                         bias_initializer=output_bias),
```

```
  ])

# Compile the model
  model.compile(
      optimizer=keras.optimizers.Adam(lr=1e-3),
      loss=keras.losses.BinaryCrossentropy(),
      metrics=metrics)

  return model
```

Understanding useful metrics

Notice that there are a few metrics defined above that can be computed by the model that will be helpful when evaluating the performance.

- **False** negatives and **false** positives are samples that were **incorrectly** classified
- **True** negatives and **true** positives are samples that were **correctly** classified
- **Accuracy** is the percentage of examples correctly classified

  > $\frac{\text{true samples}}{\text{total samples}}$

- **Precision** is the percentage of **predicted** positives that were correctly classified

  > $\frac{\text{true positives}}{\text{true positives + false positives}}$

- **Recall** is the percentage of **actual** positives that were correctly classified

  > $\frac{\text{true positives}}{\text{true positives + false negatives}}$

- **AUC** refers to the Area Under the Curve of a Receiver Operating Characteristic curve (ROC-AUC). This metric is equal to the probability that a classifier will rank a random positive sample higher than than a random negative sample.

Note: Accuracy is not a helpful metric for this task. You can 99.8%+ accuracy on this task by predicting False all the time.

Read more:

- True vs. False and Positive vs. Negative
- Accuracy
- Precision and Recall
- ROC-AUC

Baseline model

Build the model

Now create and train your model using the function that was defined earlier. Notice that the model is fit using a larger than default batch size of 2048, this is important to ensure that each batch has a decent chance of containing a few positive samples. If the batch size was too small, they would likely have no fraudulent transactions to learn from.

Note: this model will not handle the class imbalance well. You will improve it later in this tutorial.

In [61]:

```
EPOCHS = 100
BATCH_SIZE = 2048

# Stop training when a monitored metric has stopped improving.
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_auc',
    verbose=1,
    patience=10,
    mode='max',
    restore_best_weights=True)
```

In [62]:

```
# Display a model summary
model = make_model()
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_16 (Dense)             (None, 16)                480       
_________________________________________________________________
dropout_8 (Dropout)          (None, 16)                0         
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 17        
=================================================================
Total params: 497
Trainable params: 497
Non-trainable params: 0
_________________________________________________________________
```

Test run the model:

In [13]:

```
# use the model to do prediction with model.predict()
model.predict(train_features[:10])
```

Out[13]:

```
array([[0.89924395],
       [0.7323974 ],
       [0.9322966 ],
       [0.8881701 ],
       [0.88115484],
       [0.6485833 ],
       [0.79132897],
       [0.7073316 ],
       [0.8343261 ],
       [0.8008822 ]], dtype=float32)
```

Optional: Set the correct initial bias.

These are initial guesses are not great. You know the dataset is imbalanced. Set the output layer's bias to reflect that (See: A Recipe for Training Neural Networks: "init well"). This can help with initial convergence.

With the default bias initialization the loss should be about `math.log(2) = 0.69314`

In [14]:
```
results = model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE, verbose=0
print("Loss: {:0.4f}".format(results[0]))
```

```
Loss: 1.7441
```

The correct bias to set can be derived from:

$$ p_0 = pos/(pos + neg) = 1/(1+e^{-b_0}) $$$$ b_0 = -log_e(1/p_0 - 1) $$$$ b_0 = log_e(pos/neg)$$

In [15]:
```
# np.log() is a mathematical function that is used to calculate the natural logarithm.
initial_bias = np.log([pos/neg])
initial_bias
```

Out[15]:
```
array([-6.35935934])
```

Set that as the initial bias, and the model will give much more reasonable initial guesses.

It should be near: `pos/total = 0.0018`

In [16]:
```
model = make_model(output_bias = initial_bias)
model.predict(train_features[:10])
```

Out[16]:
```
array([[0.00196099],
       [0.00737071],
       [0.00182639],
       [0.00342294],
       [0.00442886],
       [0.00714428],
       [0.0061818 ],
       [0.00631511],
       [0.0088356 ],
       [0.01214694]], dtype=float32)
```

With this initialization the initial loss should be approximately:

$$-p_0log(p_0)-(1-p_0)log(1-p_0) = 0.01317$$

In [17]:
```
results = model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE, verbose=0
print("Loss: {:0.4f}".format(results[0]))
```

```
Loss: 0.0275
```

This initial loss is about 50 times less than if would have been with naive initilization.

This way the model doesn't need to spend the first few epochs just learning that positive examples are unlikely. This also makes it easier to read plots of the loss during training.

Checkpoint the initial weights

To make the various training runs more comparable, keep this initial model's weights in a checkpoint file, and load them into each model before training.

In [18]:
```
initial_weights = os.path.join(tempfile.mkdtemp(),'initial_weights')
```

```
model.save_weights(initial_weights)
```

Confirm that the bias fix helps

Before moving on, confirm quick that the careful bias initialization actually helped.

Train the model for 20 epochs, with and without this careful initialization, and compare the losses:

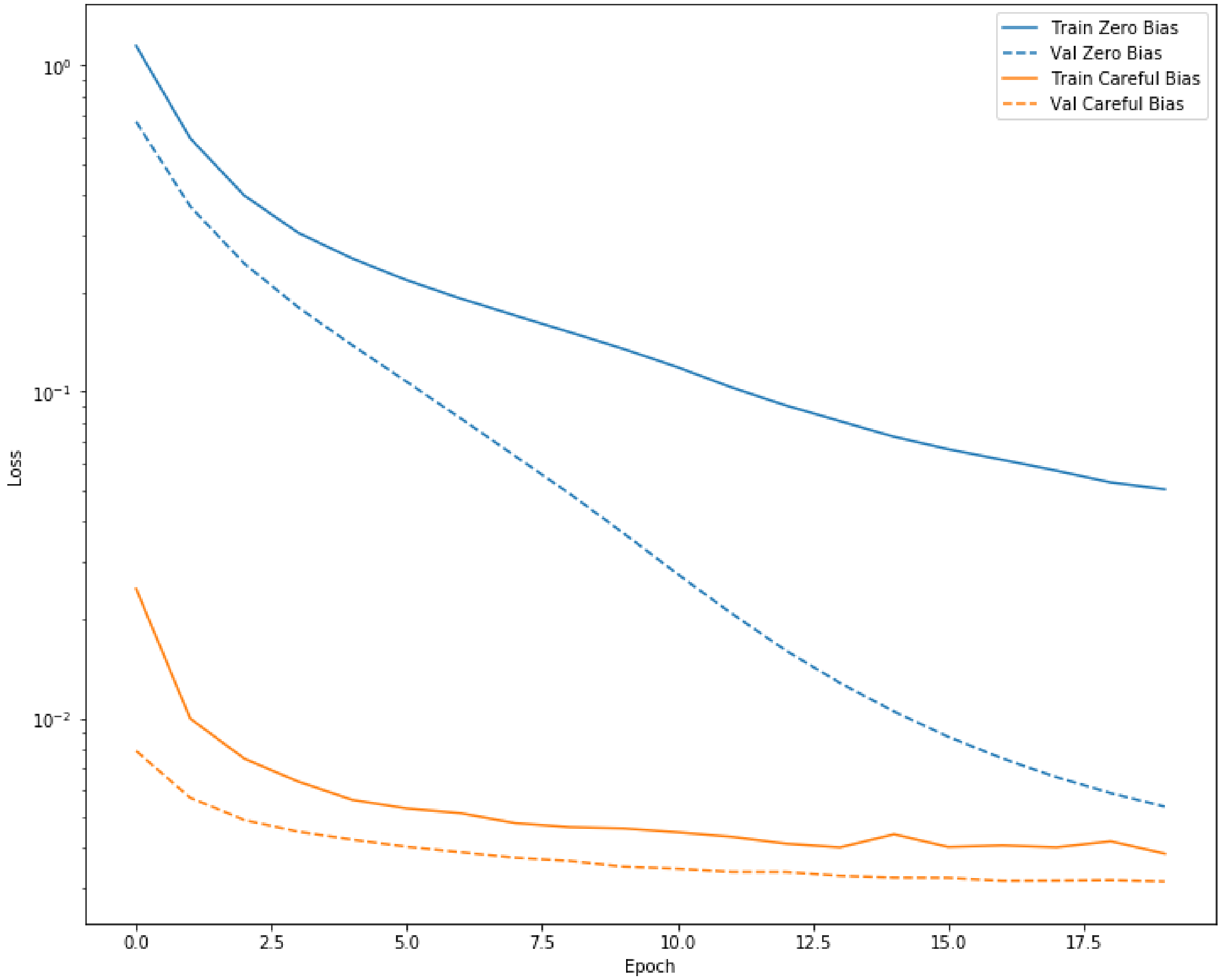
In [19]:
```
model = make_model()
model.load_weights(initial_weights)
model.layers[-1].bias.assign([0.0])
# Fit data to model
zero_bias_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

In [20]:
```
model = make_model()
model.load_weights(initial_weights)
careful_bias_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

In [21]:
```
def plot_loss(history, label, n):
  # Use a log scale to show the wide range of values.
  plt.semilogy(history.epoch,  history.history['loss'],
               color=colors[n], label='Train '+label)
  plt.semilogy(history.epoch,  history.history['val_loss'],
          color=colors[n], label='Val '+label,
          linestyle="--")
  plt.xlabel('Epoch')
  plt.ylabel('Loss')

  plt.legend()
```

In [22]:
```
plot_loss(zero_bias_history, "Zero Bias", 0)
plot_loss(careful_bias_history, "Careful Bias", 1)
```



The above figure makes it clear: In terms of validation loss, on this problem, this careful initialization gives a clear advantage.

Train the model

In [23]:

```
model = make_model()
model.load_weights(initial_weights)
# Fit data to model
baseline_history = model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks = [early_stopping],
    validation_data=(val_features, val_labels))
```

```
Train on 182276 samples, validate on 45569 samples
Epoch 1/100
182276/182276 [==============================] - 3s 16us/sample - loss: 0.0256 - tp: 64.
0000 - fp: 745.0000 - tn: 181227.0000 - fn: 240.0000 - accuracy: 0.9946 - precision: 0.0
791 - recall: 0.2105 - auc: 0.8031 - val_loss: 0.0079 - val_tp: 17.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 66.0000 - val_accuracy: 0.9984 - val_precision: 0.7083 -
val_recall: 0.2048 - val_auc: 0.9377
Epoch 2/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0100 - tp: 111.
0000 - fp: 131.0000 - tn: 181841.0000 - fn: 193.0000 - accuracy: 0.9982 - precision: 0.4
587 - recall: 0.3651 - auc: 0.8758 - val_loss: 0.0056 - val_tp: 40.0000 - val_fp: 7.0000
```

```
- val_tn: 45479.0000 - val_fn: 43.0000 - val_accuracy: 0.9989 - val_precision: 0.8511 -
val_recall: 0.4819 - val_auc: 0.9422
Epoch 3/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0075 - tp: 148.
0000 - fp: 57.0000 - tn: 181915.0000 - fn: 156.0000 - accuracy: 0.9988 - precision: 0.72
20 - recall: 0.4868 - auc: 0.9206 - val_loss: 0.0048 - val_tp: 52.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 31.0000 - val_accuracy: 0.9992 - val_precision: 0.8814 -
val_recall: 0.6265 - val_auc: 0.9382
Epoch 4/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0065 - tp: 157.
0000 - fp: 48.0000 - tn: 181924.0000 - fn: 147.0000 - accuracy: 0.9989 - precision: 0.76
59 - recall: 0.5164 - auc: 0.9210 - val_loss: 0.0045 - val_tp: 52.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 31.0000 - val_accuracy: 0.9992 - val_precision: 0.8814 -
val_recall: 0.6265 - val_auc: 0.9387
Epoch 5/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0058 - tp: 172.
0000 - fp: 43.0000 - tn: 181929.0000 - fn: 132.0000 - accuracy: 0.9990 - precision: 0.80
00 - recall: 0.5658 - auc: 0.9246 - val_loss: 0.0042 - val_tp: 51.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 32.0000 - val_accuracy: 0.9991 - val_precision: 0.8793 -
val_recall: 0.6145 - val_auc: 0.9390
Epoch 6/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0054 - tp: 169.
0000 - fp: 28.0000 - tn: 181944.0000 - fn: 135.0000 - accuracy: 0.9991 - precision: 0.85
79 - recall: 0.5559 - auc: 0.9210 - val_loss: 0.0039 - val_tp: 56.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 27.0000 - val_accuracy: 0.9993 - val_precision: 0.8889 -
val_recall: 0.6747 - val_auc: 0.9391
Epoch 7/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0054 - tp: 167.
0000 - fp: 33.0000 - tn: 181939.0000 - fn: 137.0000 - accuracy: 0.9991 - precision: 0.83
50 - recall: 0.5493 - auc: 0.9224 - val_loss: 0.0038 - val_tp: 60.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 23.0000 - val_accuracy: 0.9993 - val_precision: 0.8955 -
val_recall: 0.7229 - val_auc: 0.9392
Epoch 8/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0050 - tp: 182.
0000 - fp: 28.0000 - tn: 181944.0000 - fn: 122.0000 - accuracy: 0.9992 - precision: 0.86
67 - recall: 0.5987 - auc: 0.9215 - val_loss: 0.0038 - val_tp: 62.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 21.0000 - val_accuracy: 0.9994 - val_precision: 0.8986 -
val_recall: 0.7470 - val_auc: 0.9332
Epoch 9/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0047 - tp: 186.
0000 - fp: 36.0000 - tn: 181936.0000 - fn: 118.0000 - accuracy: 0.9992 - precision: 0.83
78 - recall: 0.6118 - auc: 0.9238 - val_loss: 0.0036 - val_tp: 63.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 20.0000 - val_accuracy: 0.9994 - val_precision: 0.9000 -
val_recall: 0.7590 - val_auc: 0.9332
Epoch 10/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0048 - tp: 176.
0000 - fp: 33.0000 - tn: 181939.0000 - fn: 128.0000 - accuracy: 0.9991 - precision: 0.84
21 - recall: 0.5789 - auc: 0.9208 - val_loss: 0.0036 - val_tp: 63.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 20.0000 - val_accuracy: 0.9994 - val_precision: 0.9000 -
val_recall: 0.7590 - val_auc: 0.9332
Epoch 11/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0045 - tp: 180.
0000 - fp: 32.0000 - tn: 181940.0000 - fn: 124.0000 - accuracy: 0.9991 - precision: 0.84
91 - recall: 0.5921 - auc: 0.9341 - val_loss: 0.0035 - val_tp: 64.0000 - val_fp: 7.0000
- val_tn: 45479.0000 - val_fn: 19.0000 - val_accuracy: 0.9994 - val_precision: 0.9014 -
val_recall: 0.7711 - val_auc: 0.9331
Epoch 12/100
169984/182276 [==========================>...] - ETA: 0s - loss: 0.0045 - tp: 175.0000 -
fp: 30.0000 - tn: 169674.0000 - fn: 105.0000 - accuracy: 0.9992 - precision: 0.8537 - re
call: 0.6250 - auc: 0.9306Restoring model weights from the end of the best epoch.
182276/182276 [==============================] - 1s 4us/sample - loss: 0.0045 - tp: 188.
0000 - fp: 31.0000 - tn: 181941.0000 - fn: 116.0000 - accuracy: 0.9992 - precision: 0.85
84 - recall: 0.6184 - auc: 0.9326 - val_loss: 0.0034 - val_tp: 63.0000 - val_fp: 6.0000
- val_tn: 45480.0000 - val_fn: 20.0000 - val_accuracy: 0.9994 - val_precision: 0.9130 -
```

```
val_recall: 0.7590 - val_auc: 0.9332
Epoch 00012: early stopping
```

Check training history

In this section, you will produce plots of your model's accuracy and loss on the training and validation set. These are useful to check for overfitting, which you can learn more about in this tutorial.

Additionally, you can produce these plots for any of the metrics you created above. False negatives are included as an example.

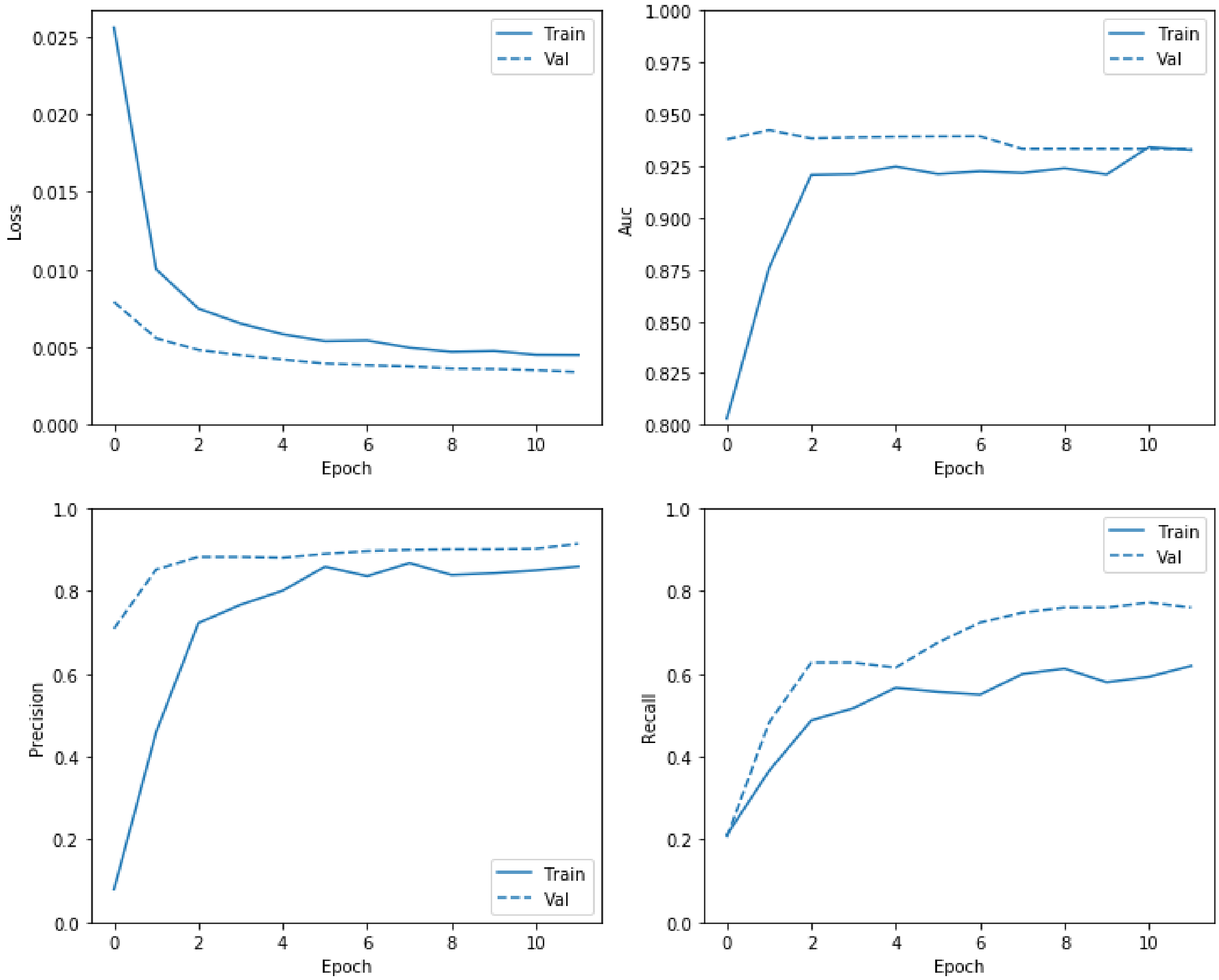
In [24]:

```
def plot_metrics(history):
  metrics =  ['loss', 'auc', 'precision', 'recall']
  for n, metric in enumerate(metrics):
    name = metric.replace("_"," ").capitalize()
# subplots() which acts as a utility wrapper and helps in creating common layouts of su
    plt.subplot(2,2,n+1)
    plt.plot(history.epoch,  history.history[metric], color=colors[0], label='Train')
    plt.plot(history.epoch, history.history['val_'+metric],
             color=colors[0], linestyle="--", label='Val')
    plt.xlabel('Epoch')
    plt.ylabel(name)
    if metric == 'loss':
      plt.ylim([0, plt.ylim()[1]])
    elif metric == 'auc':
      plt.ylim([0.8,1])
    else:
      plt.ylim([0,1])

    plt.legend()
```

In [25]:

```
plot_metrics(baseline_history)
```



Note: That the validation curve generally performs better than the training curve. This is mainly caused by the fact that the dropout layer is not active when evaluating the model.

Evaluate metrics

You can use a confusion matrix to summarize the actual vs. predicted labels where the X axis is the predicted label and the Y axis is the actual label.

In [26]:
```
# TODO 1
train_predictions_baseline = model.predict(train_features, batch_size=BATCH_SIZE)
test_predictions_baseline = model.predict(test_features, batch_size=BATCH_SIZE)
```

In [27]:
```
def plot_cm(labels, predictions, p=0.5):
  cm = confusion_matrix(labels, predictions > p)
  plt.figure(figsize=(5,5))
  sns.heatmap(cm, annot=True, fmt="d")
  plt.title('Confusion matrix @{:.2f}'.format(p))
  plt.ylabel('Actual label')
  plt.xlabel('Predicted label')

  print('Legitimate Transactions Detected (True Negatives): ', cm[0][0])
  print('Legitimate Transactions Incorrectly Detected (False Positives): ', cm[0][1])
  print('Fraudulent Transactions Missed (False Negatives): ', cm[1][0])
```

```
  print('Fraudulent Transactions Detected (True Positives): ', cm[1][1])
  print('Total Fraudulent Transactions: ', np.sum(cm[1]))
```

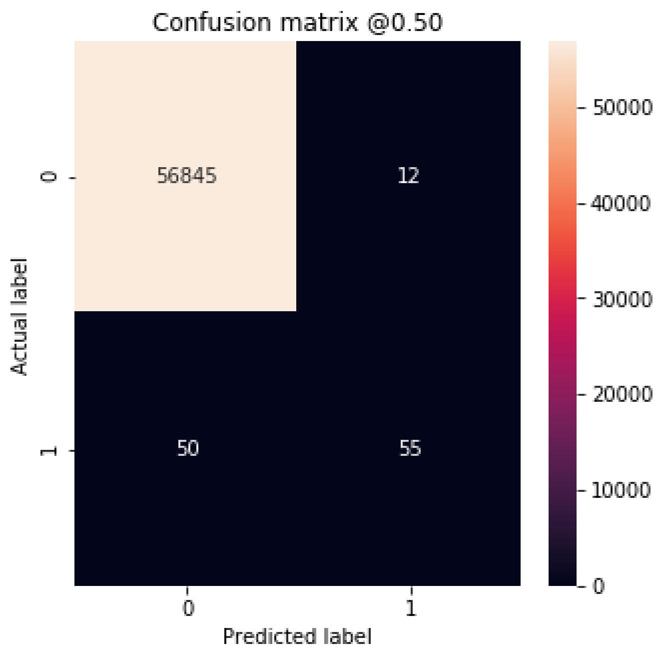
Evaluate your model on the test dataset and display the results for the metrics you created above.

In [28]:
```
baseline_results = model.evaluate(test_features, test_labels,
                                  batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(model.metrics_names, baseline_results):
  print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_baseline)
```

```
loss :  0.005941324691873794
tp :  55.0
fp :  12.0
tn :  56845.0
fn :  50.0
accuracy :  0.99891156
precision :  0.8208955
recall :  0.52380955
auc :  0.9390888

Legitimate Transactions Detected (True Negatives):  56845
Legitimate Transactions Incorrectly Detected (False Positives):  12
Fraudulent Transactions Missed (False Negatives):  50
Fraudulent Transactions Detected (True Positives):  55
Total Fraudulent Transactions:  105
```



If the model had predicted everything perfectly, this would be a diagonal matrix where values off the main diagonal, indicating incorrect predictions, would be zero. In this case the matrix shows that you have relatively few false positives, meaning that there were relatively few legitimate transactions that were incorrectly flagged. However, you would likely want to have even fewer false negatives despite the cost of increasing the number of false positives. This trade off may be preferable because false negatives would allow fraudulent transactions to go through, whereas false positives may cause an email to be sent to a customer to ask them to verify their card activity.

Plot the ROC

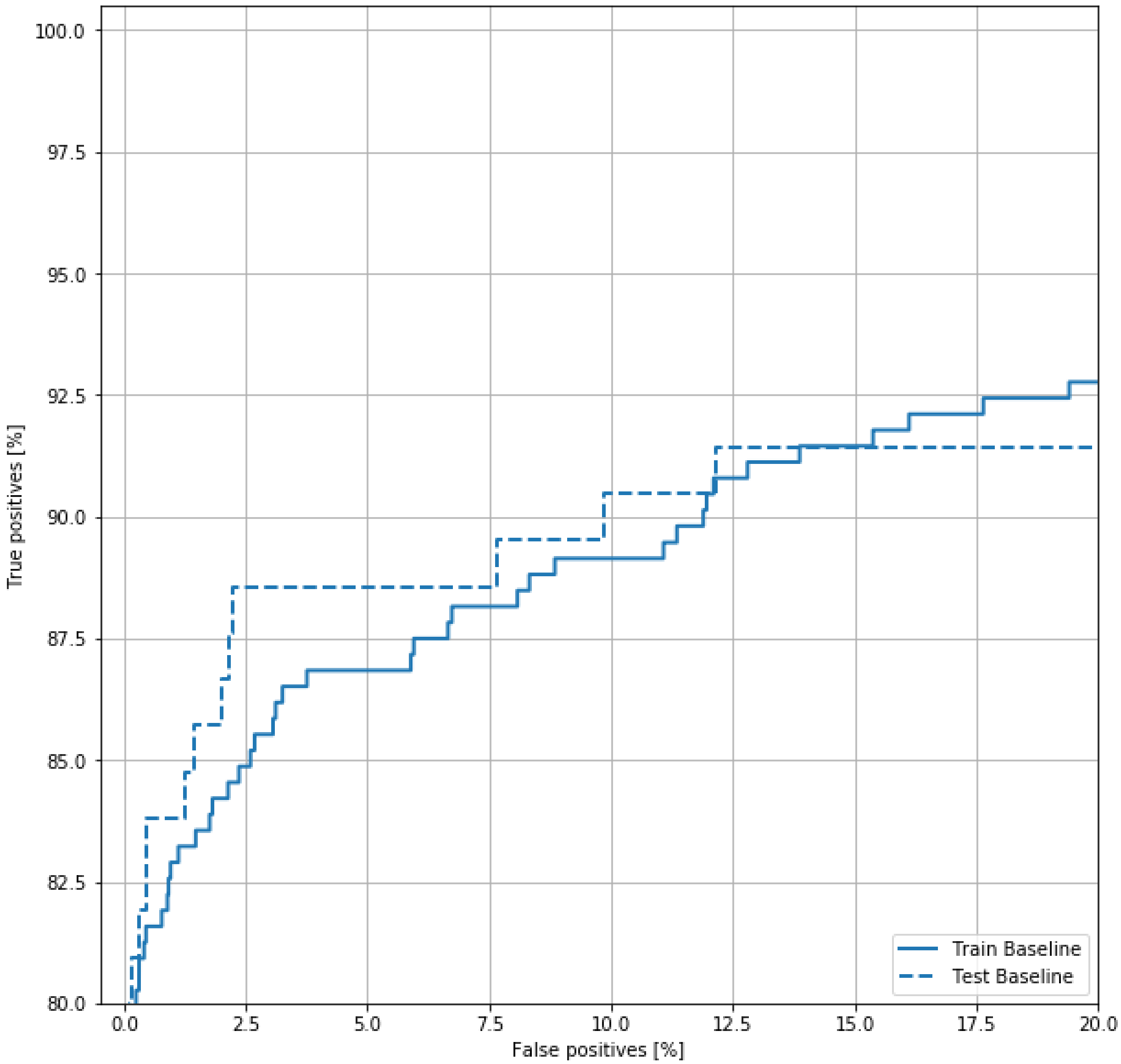
Now plot the ROC. This plot is useful because it shows, at a glance, the range of performance the model can reach just by tuning the output threshold.

In [29]:
```
def plot_roc(name, labels, predictions, **kwargs):
# Plot Receiver operating characteristic (ROC) curve.
  fp, tp, _ = sklearn.metrics.roc_curve(labels, predictions)

  plt.plot(100*fp, 100*tp, label=name, linewidth=2, **kwargs)
  plt.xlabel('False positives [%]')
  plt.ylabel('True positives [%]')
  plt.xlim([-0.5,20])
  plt.ylim([80,100.5])
  plt.grid(True)
  ax = plt.gca()
  ax.set_aspect('equal')
```

In [30]:
```
plot_roc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], line
plt.legend(loc='lower right')
```

Out[30]: <matplotlib.legend.Legend at 0x7f5f28134cf8>



It looks like the precision is relatively high, but the recall and the area under the ROC curve (AUC) aren't as high as you might like. Classifiers often face challenges when trying to maximize both precision and recall, which is especially true when working with imbalanced datasets. It is important to consider the costs of different types of errors in the context of the problem you care about. In this example, a false negative (a fraudulent transaction is missed) may have a financial cost, while a false positive (a transaction is incorrectly flagged as fraudulent) may decrease user happiness.

Class weights

Calculate class weights

The goal is to identify fradulent transactions, but you don't have very many of those positive samples to work with, so you would want to have the classifier heavily weight the few examples that are available. You can do this by passing Keras weights for each class through a parameter. These will cause the model to "pay more attention" to examples from an under-represented class.

In [31]:
```
# Scaling by total/2 helps keep the loss to a similar magnitude.
```

```
# The sum of the weights of all examples stays the same.
# TODO 1
weight_for_0 = (1 / neg)*(total)/2.0
weight_for_1 = (1 / pos)*(total)/2.0

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

```
Weight for class 0: 0.50
Weight for class 1: 289.44
```

Train a model with class weights

Now try re-training and evaluating the model with class weights to see how that affects the predictions.

Note: Using `class_weights` changes the range of the loss. This may affect the stability of the training depending on the optimizer. Optimizers whose step size is dependent on the magnitude of the gradient, like `optimizers.SGD`, may fail. The optimizer used here, `optimizers.Adam`, is unaffected by the scaling change. Also note that because of the weighting, the total losses are not comparable between the two models.

In [32]:

```
weighted_model = make_model()
weighted_model.load_weights(initial_weights)

weighted_history = weighted_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks = [early_stopping],
    validation_data=(val_features, val_labels),
    # The class weights go here
    class_weight=class_weight)
```

```
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train on 182276 samples, validate on 45569 samples
Epoch 1/100
182276/182276 [==============================] - 3s 19us/sample - loss: 1.0524 - tp: 13
8.0000 - fp: 2726.0000 - tn: 179246.0000 - fn: 166.0000 - accuracy: 0.9841 - precision:
0.0482 - recall: 0.4539 - auc: 0.8321 - val_loss: 0.4515 - val_tp: 59.0000 - val_fp: 43
2.0000 - val_tn: 45054.0000 - val_fn: 24.0000 - val_accuracy: 0.9900 - val_precision: 0.
1202 - val_recall: 0.7108 - val_auc: 0.9492
Epoch 2/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.5537 - tp: 216.
0000 - fp: 3783.0000 - tn: 178189.0000 - fn: 88.0000 - accuracy: 0.9788 - precision: 0.0
540 - recall: 0.7105 - auc: 0.9033 - val_loss: 0.3285 - val_tp: 69.0000 - val_fp: 514.00
00 - val_tn: 44972.0000 - val_fn: 14.0000 - val_accuracy: 0.9884 - val_precision: 0.1184
- val_recall: 0.8313 - val_auc: 0.9605
```

```
Epoch 3/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.4178 - tp: 238.
0000 - fp: 4540.0000 - tn: 177432.0000 - fn: 66.0000 - accuracy: 0.9747 - precision: 0.0
498 - recall: 0.7829 - auc: 0.9237 - val_loss: 0.2840 - val_tp: 69.0000 - val_fp: 570.00
00 - val_tn: 44916.0000 - val_fn: 14.0000 - val_accuracy: 0.9872 - val_precision: 0.1080
- val_recall: 0.8313 - val_auc: 0.9669
Epoch 4/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.3848 - tp: 247.
0000 - fp: 5309.0000 - tn: 176663.0000 - fn: 57.0000 - accuracy: 0.9706 - precision: 0.0
445 - recall: 0.8125 - auc: 0.9292 - val_loss: 0.2539 - val_tp: 71.0000 - val_fp: 622.00
00 - val_tn: 44864.0000 - val_fn: 12.0000 - val_accuracy: 0.9861 - val_precision: 0.1025
- val_recall: 0.8554 - val_auc: 0.9709
Epoch 5/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.3596 - tp: 254.
0000 - fp: 6018.0000 - tn: 175954.0000 - fn: 50.0000 - accuracy: 0.9667 - precision: 0.0
405 - recall: 0.8355 - auc: 0.9323 - val_loss: 0.2363 - val_tp: 72.0000 - val_fp: 713.00
00 - val_tn: 44773.0000 - val_fn: 11.0000 - val_accuracy: 0.9841 - val_precision: 0.0917
- val_recall: 0.8675 - val_auc: 0.9725
Epoch 6/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.3115 - tp: 255.
0000 - fp: 6366.0000 - tn: 175606.0000 - fn: 49.0000 - accuracy: 0.9648 - precision: 0.0
385 - recall: 0.8388 - auc: 0.9477 - val_loss: 0.2243 - val_tp: 72.0000 - val_fp: 768.00
00 - val_tn: 44718.0000 - val_fn: 11.0000 - val_accuracy: 0.9829 - val_precision: 0.0857
- val_recall: 0.8675 - val_auc: 0.9728
Epoch 7/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.3179 - tp: 258.
0000 - fp: 6804.0000 - tn: 175168.0000 - fn: 46.0000 - accuracy: 0.9624 - precision: 0.0
365 - recall: 0.8487 - auc: 0.9435 - val_loss: 0.2165 - val_tp: 72.0000 - val_fp: 812.00
00 - val_tn: 44674.0000 - val_fn: 11.0000 - val_accuracy: 0.9819 - val_precision: 0.0814
- val_recall: 0.8675 - val_auc: 0.9739
Epoch 8/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2880 - tp: 260.
0000 - fp: 6669.0000 - tn: 175303.0000 - fn: 44.0000 - accuracy: 0.9632 - precision: 0.0
375 - recall: 0.8553 - auc: 0.9530 - val_loss: 0.2122 - val_tp: 72.0000 - val_fp: 783.00
00 - val_tn: 44703.0000 - val_fn: 11.0000 - val_accuracy: 0.9826 - val_precision: 0.0842
- val_recall: 0.8675 - val_auc: 0.9769
Epoch 9/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2676 - tp: 262.
0000 - fp: 6904.0000 - tn: 175068.0000 - fn: 42.0000 - accuracy: 0.9619 - precision: 0.0
366 - recall: 0.8618 - auc: 0.9594 - val_loss: 0.2056 - val_tp: 72.0000 - val_fp: 855.00
00 - val_tn: 44631.0000 - val_fn: 11.0000 - val_accuracy: 0.9810 - val_precision: 0.0777
- val_recall: 0.8675 - val_auc: 0.9750
Epoch 10/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2498 - tp: 266.
0000 - fp: 6833.0000 - tn: 175139.0000 - fn: 38.0000 - accuracy: 0.9623 - precision: 0.0
375 - recall: 0.8750 - auc: 0.9593 - val_loss: 0.2001 - val_tp: 73.0000 - val_fp: 840.00
00 - val_tn: 44646.0000 - val_fn: 10.0000 - val_accuracy: 0.9813 - val_precision: 0.0800
- val_recall: 0.8795 - val_auc: 0.9761
Epoch 11/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2681 - tp: 262.
0000 - fp: 6845.0000 - tn: 175127.0000 - fn: 42.0000 - accuracy: 0.9622 - precision: 0.0
369 - recall: 0.8618 - auc: 0.9559 - val_loss: 0.1964 - val_tp: 73.0000 - val_fp: 865.00
00 - val_tn: 44621.0000 - val_fn: 10.0000 - val_accuracy: 0.9808 - val_precision: 0.0778
- val_recall: 0.8795 - val_auc: 0.9768
Epoch 12/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2406 - tp: 268.
0000 - fp: 7070.0000 - tn: 174902.0000 - fn: 36.0000 - accuracy: 0.9610 - precision: 0.0
365 - recall: 0.8816 - auc: 0.9646 - val_loss: 0.1940 - val_tp: 73.0000 - val_fp: 848.00
00 - val_tn: 44638.0000 - val_fn: 10.0000 - val_accuracy: 0.9812 - val_precision: 0.0793
- val_recall: 0.8795 - val_auc: 0.9771
Epoch 13/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2285 - tp: 269.
0000 - fp: 6976.0000 - tn: 174996.0000 - fn: 35.0000 - accuracy: 0.9615 - precision: 0.0
371 - recall: 0.8849 - auc: 0.9680 - val_loss: 0.1930 - val_tp: 73.0000 - val_fp: 857.00
00 - val_tn: 44629.0000 - val_fn: 10.0000 - val_accuracy: 0.9810 - val_precision: 0.0785
```

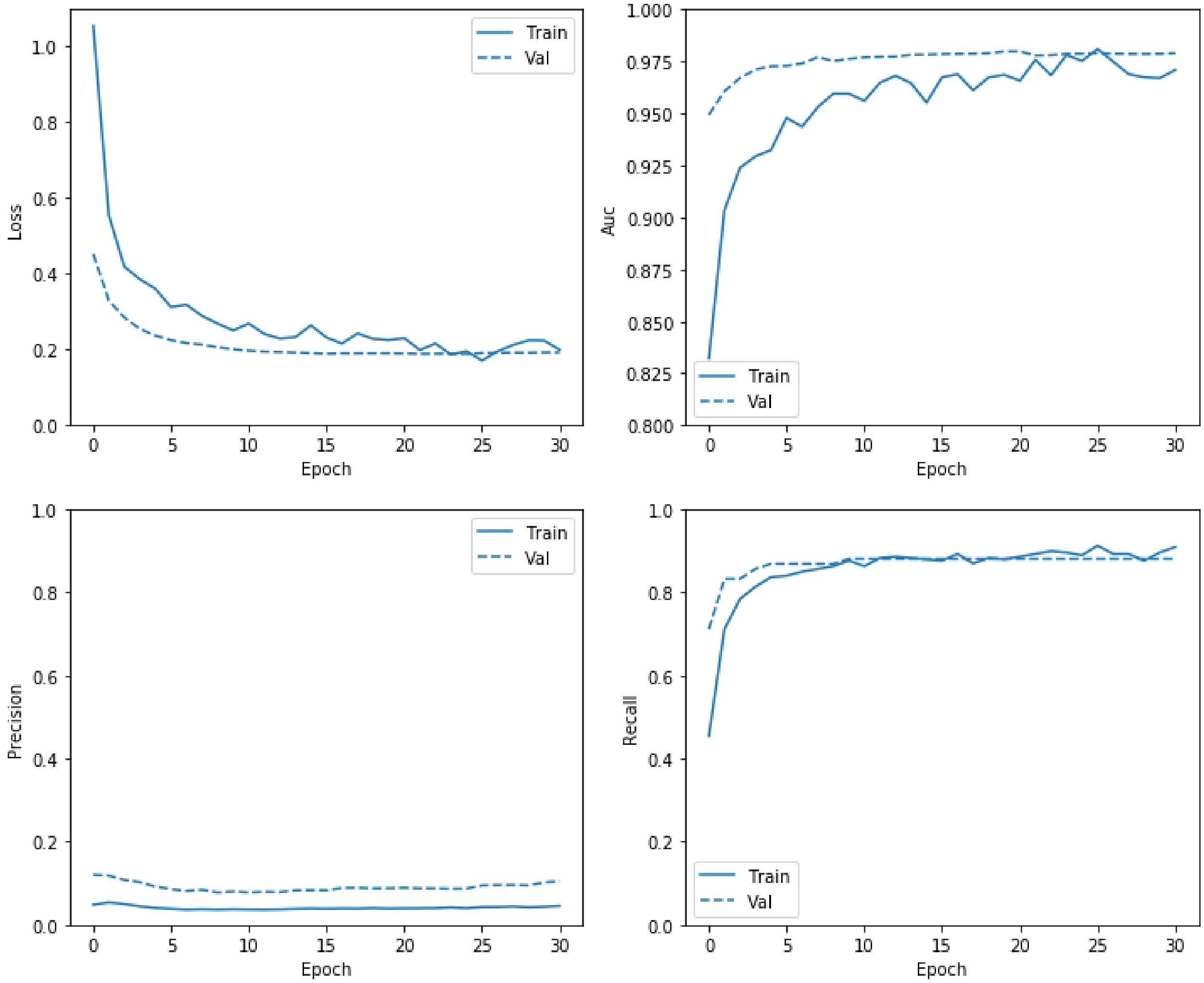
```
- val_recall: 0.8795 - val_auc: 0.9772
Epoch 14/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2322 - tp: 268.
0000 - fp: 6718.0000 - tn: 175254.0000 - fn: 36.0000 - accuracy: 0.9629 - precision: 0.0
384 - recall: 0.8816 - auc: 0.9644 - val_loss: 0.1915 - val_tp: 73.0000 - val_fp: 808.00
00 - val_tn: 44678.0000 - val_fn: 10.0000 - val_accuracy: 0.9820 - val_precision: 0.0829
- val_recall: 0.8795 - val_auc: 0.9781
Epoch 15/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2631 - tp: 267.
0000 - fp: 6578.0000 - tn: 175394.0000 - fn: 37.0000 - accuracy: 0.9637 - precision: 0.0
390 - recall: 0.8783 - auc: 0.9551 - val_loss: 0.1900 - val_tp: 73.0000 - val_fp: 803.00
00 - val_tn: 44683.0000 - val_fn: 10.0000 - val_accuracy: 0.9822 - val_precision: 0.0833
- val_recall: 0.8795 - val_auc: 0.9781
Epoch 16/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2314 - tp: 266.
0000 - fp: 6644.0000 - tn: 175328.0000 - fn: 38.0000 - accuracy: 0.9633 - precision: 0.0
385 - recall: 0.8750 - auc: 0.9672 - val_loss: 0.1882 - val_tp: 73.0000 - val_fp: 806.00
00 - val_tn: 44680.0000 - val_fn: 10.0000 - val_accuracy: 0.9821 - val_precision: 0.0830
- val_recall: 0.8795 - val_auc: 0.9784
Epoch 17/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2152 - tp: 271.
0000 - fp: 6663.0000 - tn: 175309.0000 - fn: 33.0000 - accuracy: 0.9633 - precision: 0.0
391 - recall: 0.8914 - auc: 0.9687 - val_loss: 0.1895 - val_tp: 73.0000 - val_fp: 754.00
00 - val_tn: 44732.0000 - val_fn: 10.0000 - val_accuracy: 0.9832 - val_precision: 0.0883
- val_recall: 0.8795 - val_auc: 0.9785
Epoch 18/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2420 - tp: 264.
0000 - fp: 6535.0000 - tn: 175437.0000 - fn: 40.0000 - accuracy: 0.9639 - precision: 0.0
388 - recall: 0.8684 - auc: 0.9610 - val_loss: 0.1895 - val_tp: 73.0000 - val_fp: 749.00
00 - val_tn: 44737.0000 - val_fn: 10.0000 - val_accuracy: 0.9833 - val_precision: 0.0888
- val_recall: 0.8795 - val_auc: 0.9786
Epoch 19/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2279 - tp: 268.
0000 - fp: 6443.0000 - tn: 175529.0000 - fn: 36.0000 - accuracy: 0.9645 - precision: 0.0
399 - recall: 0.8816 - auc: 0.9672 - val_loss: 0.1895 - val_tp: 73.0000 - val_fp: 763.00
00 - val_tn: 44723.0000 - val_fn: 10.0000 - val_accuracy: 0.9830 - val_precision: 0.0873
- val_recall: 0.8795 - val_auc: 0.9788
Epoch 20/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2247 - tp: 267.
0000 - fp: 6596.0000 - tn: 175376.0000 - fn: 37.0000 - accuracy: 0.9636 - precision: 0.0
389 - recall: 0.8783 - auc: 0.9684 - val_loss: 0.1896 - val_tp: 73.0000 - val_fp: 760.00
00 - val_tn: 44726.0000 - val_fn: 10.0000 - val_accuracy: 0.9831 - val_precision: 0.0876
- val_recall: 0.8795 - val_auc: 0.9797
Epoch 21/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2296 - tp: 269.
0000 - fp: 6562.0000 - tn: 175410.0000 - fn: 35.0000 - accuracy: 0.9638 - precision: 0.0
394 - recall: 0.8849 - auc: 0.9656 - val_loss: 0.1889 - val_tp: 73.0000 - val_fp: 750.00
00 - val_tn: 44736.0000 - val_fn: 10.0000 - val_accuracy: 0.9833 - val_precision: 0.0887
- val_recall: 0.8795 - val_auc: 0.9797
Epoch 22/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.1982 - tp: 271.
0000 - fp: 6583.0000 - tn: 175389.0000 - fn: 33.0000 - accuracy: 0.9637 - precision: 0.0
395 - recall: 0.8914 - auc: 0.9756 - val_loss: 0.1879 - val_tp: 73.0000 - val_fp: 764.00
00 - val_tn: 44722.0000 - val_fn: 10.0000 - val_accuracy: 0.9830 - val_precision: 0.0872
- val_recall: 0.8795 - val_auc: 0.9777
Epoch 23/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2154 - tp: 273.
0000 - fp: 6552.0000 - tn: 175420.0000 - fn: 31.0000 - accuracy: 0.9639 - precision: 0.0
400 - recall: 0.8980 - auc: 0.9682 - val_loss: 0.1882 - val_tp: 73.0000 - val_fp: 762.00
00 - val_tn: 44724.0000 - val_fn: 10.0000 - val_accuracy: 0.9831 - val_precision: 0.0874
- val_recall: 0.8795 - val_auc: 0.9779
Epoch 24/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.1861 - tp: 272.
0000 - fp: 6248.0000 - tn: 175724.0000 - fn: 32.0000 - accuracy: 0.9655 - precision: 0.0
417 - recall: 0.8947 - auc: 0.9779 - val_loss: 0.1885 - val_tp: 73.0000 - val_fp: 772.00
```

```
00 - val_tn: 44714.0000 - val_fn: 10.0000 - val_accuracy: 0.9828 - val_precision: 0.0864
- val_recall: 0.8795 - val_auc: 0.9785
Epoch 25/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.1953 - tp: 270.
0000 - fp: 6501.0000 - tn: 175471.0000 - fn: 34.0000 - accuracy: 0.9641 - precision: 0.0
399 - recall: 0.8882 - auc: 0.9751 - val_loss: 0.1877 - val_tp: 73.0000 - val_fp: 768.00
00 - val_tn: 44718.0000 - val_fn: 10.0000 - val_accuracy: 0.9829 - val_precision: 0.0868
- val_recall: 0.8795 - val_auc: 0.9786
Epoch 26/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.1704 - tp: 277.
0000 - fp: 6215.0000 - tn: 175757.0000 - fn: 27.0000 - accuracy: 0.9658 - precision: 0.0
427 - recall: 0.9112 - auc: 0.9808 - val_loss: 0.1903 - val_tp: 73.0000 - val_fp: 698.00
00 - val_tn: 44788.0000 - val_fn: 10.0000 - val_accuracy: 0.9845 - val_precision: 0.0947
- val_recall: 0.8795 - val_auc: 0.9788
Epoch 27/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.1946 - tp: 271.
0000 - fp: 6036.0000 - tn: 175936.0000 - fn: 33.0000 - accuracy: 0.9667 - precision: 0.0
430 - recall: 0.8914 - auc: 0.9748 - val_loss: 0.1908 - val_tp: 73.0000 - val_fp: 692.00
00 - val_tn: 44794.0000 - val_fn: 10.0000 - val_accuracy: 0.9846 - val_precision: 0.0954
- val_recall: 0.8795 - val_auc: 0.9786
Epoch 28/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2115 - tp: 271.
0000 - fp: 5873.0000 - tn: 176099.0000 - fn: 33.0000 - accuracy: 0.9676 - precision: 0.0
441 - recall: 0.8914 - auc: 0.9688 - val_loss: 0.1914 - val_tp: 73.0000 - val_fp: 691.00
00 - val_tn: 44795.0000 - val_fn: 10.0000 - val_accuracy: 0.9846 - val_precision: 0.0955
- val_recall: 0.8795 - val_auc: 0.9785
Epoch 29/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2237 - tp: 266.
0000 - fp: 6047.0000 - tn: 175925.0000 - fn: 38.0000 - accuracy: 0.9666 - precision: 0.0
421 - recall: 0.8750 - auc: 0.9672 - val_loss: 0.1909 - val_tp: 73.0000 - val_fp: 698.00
00 - val_tn: 44788.0000 - val_fn: 10.0000 - val_accuracy: 0.9845 - val_precision: 0.0947
- val_recall: 0.8795 - val_auc: 0.9784
Epoch 30/100
182276/182276 [==============================] - 1s 4us/sample - loss: 0.2232 - tp: 272.
0000 - fp: 5990.0000 - tn: 175982.0000 - fn: 32.0000 - accuracy: 0.9670 - precision: 0.0
434 - recall: 0.8947 - auc: 0.9668 - val_loss: 0.1919 - val_tp: 73.0000 - val_fp: 642.00
00 - val_tn: 44844.0000 - val_fn: 10.0000 - val_accuracy: 0.9857 - val_precision: 0.1021
- val_recall: 0.8795 - val_auc: 0.9785
Epoch 31/100
178176/182276 [============================>.] - ETA: 0s - loss: 0.2022 - tp: 273.0000 -
fp: 5659.0000 - tn: 172216.0000 - fn: 28.0000 - accuracy: 0.9681 - precision: 0.0460 - r
ecall: 0.9070 - auc: 0.9705Restoring model weights from the end of the best epoch.
182276/182276 [==============================] - 1s 4us/sample - loss: 0.1989 - tp: 276.
0000 - fp: 5796.0000 - tn: 176176.0000 - fn: 28.0000 - accuracy: 0.9680 - precision: 0.0
455 - recall: 0.9079 - auc: 0.9708 - val_loss: 0.1920 - val_tp: 73.0000 - val_fp: 626.00
00 - val_tn: 44860.0000 - val_fn: 10.0000 - val_accuracy: 0.9860 - val_precision: 0.1044
- val_recall: 0.8795 - val_auc: 0.9788
Epoch 00031: early stopping
```

Check training history

In [33]:

```
plot_metrics(weighted_history)
```



Evaluate metrics

In [34]:
```
# TODO 1
train_predictions_weighted = weighted_model.predict(train_features, batch_size=BATCH_SI
test_predictions_weighted = weighted_model.predict(test_features, batch_size=BATCH_SIZE
```

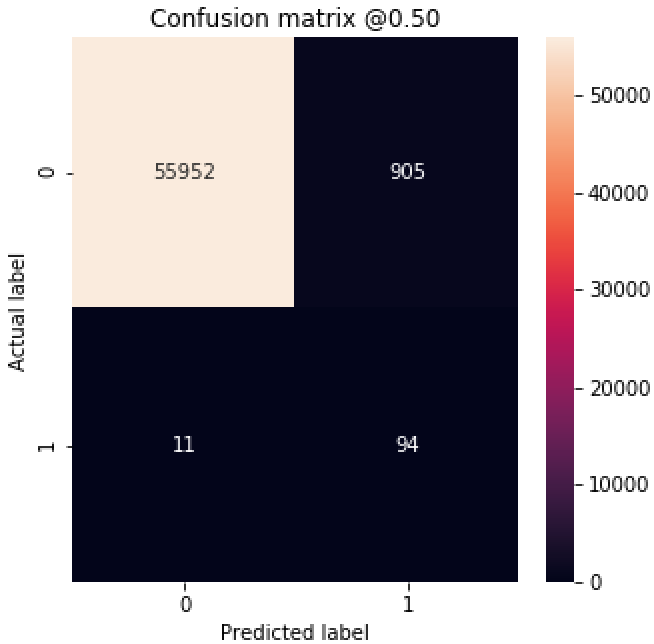
In [35]:
```
weighted_results = weighted_model.evaluate(test_features, test_labels,
                                           batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(weighted_model.metrics_names, weighted_results):
  print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_weighted)
```

```
loss :  0.06950428275801711
tp :  94.0
fp :  905.0
tn :  55952.0
fn :  11.0
accuracy :  0.9839191
precision :  0.0940941
recall :  0.8952381
auc :  0.9844724

Legitimate Transactions Detected (True Negatives):  55952
```

```
Legitimate Transactions Incorrectly Detected (False Positives):  905
Fraudulent Transactions Missed (False Negatives):  11
Fraudulent Transactions Detected (True Positives):  94
Total Fraudulent Transactions:  105
```



Here you can see that with class weights the accuracy and precision are lower because there are more false positives, but conversely the recall and AUC are higher because the model also found more true positives. Despite having lower accuracy, this model has higher recall (and identifies more fraudulent transactions). Of course, there is a cost to both types of error (you wouldn't want to bug users by flagging too many legitimate transactions as fraudulent, either). Carefully consider the trade offs between these different types of errors for your application.

Plot the ROC

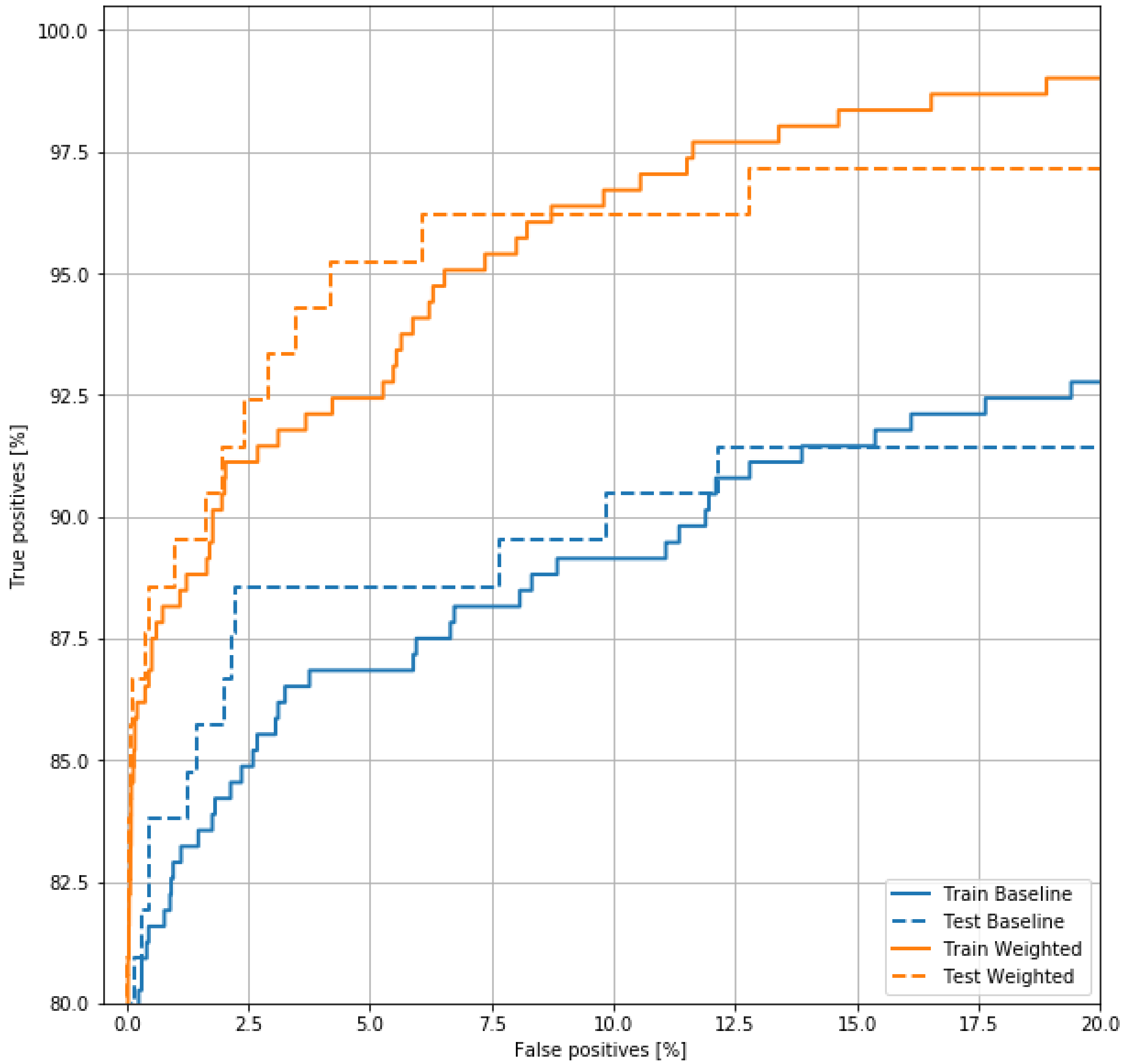
In [36]:

```
plot_roc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], line

plot_roc("Train Weighted", train_labels, train_predictions_weighted, color=colors[1])
plot_roc("Test Weighted", test_labels, test_predictions_weighted, color=colors[1], line


# Function legend() which is used to Place a legend on the axes
plt.legend(loc='lower right')
```

Out[36]: <matplotlib.legend.Legend at 0x7f5f2017f7b8>



Oversampling

Oversample the minority class

A related approach would be to resample the dataset by oversampling the minority class.

In [37]:

```
# TODO 1
pos_features = train_features[bool_train_labels]
neg_features = train_features[~bool_train_labels]

pos_labels = train_labels[bool_train_labels]
neg_labels = train_labels[~bool_train_labels]
```

Using NumPy

You can balance the dataset manually by choosing the right number of random indices from the positive examples:

In [38]:

```
# np.arange() return evenly spaced values within a given interval.
ids = np.arange(len(pos_features))
# choice() method, you can get the random samples of one dimensional array and return t
choices = np.random.choice(ids, len(neg_features))

res_pos_features = pos_features[choices]
res_pos_labels = pos_labels[choices]

res_pos_features.shape
```

Out[38]: (181972, 29)

In [39]:

```
# numpy.concatenate() function concatenate a sequence of arrays along an existing axis.
resampled_features = np.concatenate([res_pos_features, neg_features], axis=0)
resampled_labels = np.concatenate([res_pos_labels, neg_labels], axis=0)

order = np.arange(len(resampled_labels))
# numpy.random.shuffle() modify a sequence in-place by shuffling its contents.
np.random.shuffle(order)
resampled_features = resampled_features[order]
resampled_labels = resampled_labels[order]

resampled_features.shape
```

Out[39]: (363944, 29)

Using `tf.data`

If you're using `tf.data` the easiest way to produce balanced examples is to start with a `positive` and a `negative` dataset, and merge them. See the tf.data guide for more examples.

In [40]:

```
BUFFER_SIZE = 100000

def make_ds(features, labels):
# With the help of tf.data.Dataset.from_tensor_slices() method, we can get the slices o
# by using tf.data.Dataset.from_tensor_slices() method.
  ds = tf.data.Dataset.from_tensor_slices((features, labels))#.cache()
  ds = ds.shuffle(BUFFER_SIZE).repeat()
  return ds

pos_ds = make_ds(pos_features, pos_labels)
neg_ds = make_ds(neg_features, neg_labels)
```

Each dataset provides `(feature, label)` pairs:

In [41]:

```
for features, label in pos_ds.take(1):
  print("Features:\n", features.numpy())
  print()
  print("Label: ", label.numpy())
```

```
Features:
 [-2.46955933  3.42534191 -4.42937043  3.70651659 -3.17895499 -1.30458304
 -5.          2.86676917 -4.9308611  -5.          3.58555137 -5.
  1.51535494 -5.          0.01049775 -5.         -5.         -5.
  2.02380731  0.36595419  1.61836304 -1.16743779  0.31324117 -0.35515978
 -0.62579636 -0.55952005  0.51255883  1.15454727  0.87478003]
```

```
Label:  1
```

Merge the two together using `experimental.sample_from_datasets` :

In [42]:
```
# Samples elements at random from the datasets in `datasets`.
resampled_ds = tf.data.experimental.sample_from_datasets([pos_ds, neg_ds], weights=[0.5
resampled_ds = resampled_ds.batch(BATCH_SIZE).prefetch(2)
```

In [43]:
```
for features, label in resampled_ds.take(1):
  print(label.numpy().mean())
```

```
0.48974609375
```

To use this dataset, you'll need the number of steps per epoch.

The definition of "epoch" in this case is less clear. Say it's the number of batches required to see each negative example once:

In [44]:
```
# `np.ceil()` function returns the ceil value of the input array elements
resampled_steps_per_epoch = np.ceil(2.0*neg/BATCH_SIZE)
resampled_steps_per_epoch
```

Out[44]:
```
278.0
```

Train on the oversampled data

Now try training the model with the resampled data set instead of using class weights to see how these methods compare.

Note: Because the data was balanced by replicating the positive examples, the total dataset size is larger, and each epoch runs for more training steps.

In [45]:
```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).cache()
val_ds = val_ds.batch(BATCH_SIZE).prefetch(2)

resampled_history = resampled_model.fit(
    resampled_ds,
    epochs=EPOCHS,
    steps_per_epoch=resampled_steps_per_epoch,
    callbacks = [early_stopping],
    validation_data=val_ds)
```

```
Train for 278.0 steps, validate for 23 steps
Epoch 1/100
278/278 [==============================] - 13s 48ms/step - loss: 0.4624 - tp: 267186.000
0 - fp: 124224.0000 - tn: 160439.0000 - fn: 17495.0000 - accuracy: 0.7511 - precision:
0.6826 - recall: 0.9385 - auc: 0.9268 - val_loss: 0.3299 - val_tp: 79.0000 - val_fp: 282
```

```
5.0000 - val_tn: 42661.0000 - val_fn: 4.0000 - val_accuracy: 0.9379 - val_precision: 0.0
272 - val_recall: 0.9518 - val_auc: 0.9799
Epoch 2/100
278/278 [==============================] - 11s 39ms/step - loss: 0.2362 - tp: 264077.000
0 - fp: 26654.0000 - tn: 257570.0000 - fn: 21043.0000 - accuracy: 0.9162 - precision: 0.
9083 - recall: 0.9262 - auc: 0.9708 - val_loss: 0.1926 - val_tp: 75.0000 - val_fp: 1187.
0000 - val_tn: 44299.0000 - val_fn: 8.0000 - val_accuracy: 0.9738 - val_precision: 0.059
4 - val_recall: 0.9036 - val_auc: 0.9779
Epoch 3/100
278/278 [==============================] - 11s 40ms/step - loss: 0.1887 - tp: 263490.000
0 - fp: 12935.0000 - tn: 271381.0000 - fn: 21538.0000 - accuracy: 0.9395 - precision: 0.
9532 - recall: 0.9244 - auc: 0.9804 - val_loss: 0.1373 - val_tp: 75.0000 - val_fp: 1064.
0000 - val_tn: 44422.0000 - val_fn: 8.0000 - val_accuracy: 0.9765 - val_precision: 0.065
8 - val_recall: 0.9036 - val_auc: 0.9778
Epoch 4/100
278/278 [==============================] - 11s 41ms/step - loss: 0.1605 - tp: 263933.000
0 - fp: 10513.0000 - tn: 274505.0000 - fn: 20393.0000 - accuracy: 0.9457 - precision: 0.
9617 - recall: 0.9283 - auc: 0.9866 - val_loss: 0.1078 - val_tp: 75.0000 - val_fp: 1070.
0000 - val_tn: 44416.0000 - val_fn: 8.0000 - val_accuracy: 0.9763 - val_precision: 0.065
5 - val_recall: 0.9036 - val_auc: 0.9783
Epoch 5/100
278/278 [==============================] - 11s 39ms/step - loss: 0.1423 - tp: 265715.000
0 - fp: 9592.0000 - tn: 275145.0000 - fn: 18892.0000 - accuracy: 0.9500 - precision: 0.9
652 - recall: 0.9336 - auc: 0.9901 - val_loss: 0.0928 - val_tp: 75.0000 - val_fp: 1051.0
000 - val_tn: 44435.0000 - val_fn: 8.0000 - val_accuracy: 0.9768 - val_precision: 0.0666
- val_recall: 0.9036 - val_auc: 0.9762
Epoch 6/100
278/278 [==============================] - 11s 40ms/step - loss: 0.1297 - tp: 267181.000
0 - fp: 8944.0000 - tn: 275445.0000 - fn: 17774.0000 - accuracy: 0.9531 - precision: 0.9
676 - recall: 0.9376 - auc: 0.9920 - val_loss: 0.0847 - val_tp: 75.0000 - val_fp: 1077.0
000 - val_tn: 44409.0000 - val_fn: 8.0000 - val_accuracy: 0.9762 - val_precision: 0.0651
- val_recall: 0.9036 - val_auc: 0.9748
Epoch 7/100
278/278 [==============================] - 11s 39ms/step - loss: 0.1203 - tp: 267440.000
0 - fp: 8606.0000 - tn: 276459.0000 - fn: 16839.0000 - accuracy: 0.9553 - precision: 0.9
688 - recall: 0.9408 - auc: 0.9933 - val_loss: 0.0775 - val_tp: 75.0000 - val_fp: 1003.0
000 - val_tn: 44483.0000 - val_fn: 8.0000 - val_accuracy: 0.9778 - val_precision: 0.0696
- val_recall: 0.9036 - val_auc: 0.9742
Epoch 8/100
278/278 [==============================] - 11s 40ms/step - loss: 0.1132 - tp: 268799.000
0 - fp: 8165.0000 - tn: 276260.0000 - fn: 16120.0000 - accuracy: 0.9573 - precision: 0.9
705 - recall: 0.9434 - auc: 0.9941 - val_loss: 0.0716 - val_tp: 75.0000 - val_fp: 927.00
00 - val_tn: 44559.0000 - val_fn: 8.0000 - val_accuracy: 0.9795 - val_precision: 0.0749
- val_recall: 0.9036 - val_auc: 0.9713
Epoch 9/100
278/278 [==============================] - 11s 40ms/step - loss: 0.1074 - tp: 269627.000
0 - fp: 7971.0000 - tn: 276559.0000 - fn: 15187.0000 - accuracy: 0.9593 - precision: 0.9
713 - recall: 0.9467 - auc: 0.9947 - val_loss: 0.0670 - val_tp: 75.0000 - val_fp: 880.00
00 - val_tn: 44606.0000 - val_fn: 8.0000 - val_accuracy: 0.9805 - val_precision: 0.0785
- val_recall: 0.9036 - val_auc: 0.9713
Epoch 10/100
278/278 [==============================] - 11s 39ms/step - loss: 0.1017 - tp: 270359.000
0 - fp: 7590.0000 - tn: 277311.0000 - fn: 14084.0000 - accuracy: 0.9619 - precision: 0.9
727 - recall: 0.9505 - auc: 0.9952 - val_loss: 0.0629 - val_tp: 75.0000 - val_fp: 848.00
00 - val_tn: 44638.0000 - val_fn: 8.0000 - val_accuracy: 0.9812 - val_precision: 0.0813
- val_recall: 0.9036 - val_auc: 0.9717
Epoch 11/100
276/278 [============================>.] - ETA: 0s - loss: 0.0977 - tp: 269672.0000 - f
p: 7408.0000 - tn: 274621.0000 - fn: 13547.0000 - accuracy: 0.9629 - precision: 0.9733 -
recall: 0.9522 - auc: 0.9955Restoring model weights from the end of the best epoch.
278/278 [==============================] - 11s 39ms/step - loss: 0.0978 - tp: 271609.000
0 - fp: 7474.0000 - tn: 276625.0000 - fn: 13636.0000 - accuracy: 0.9629 - precision: 0.9
732 - recall: 0.9522 - auc: 0.9955 - val_loss: 0.0615 - val_tp: 75.0000 - val_fp: 841.00
00 - val_tn: 44645.0000 - val_fn: 8.0000 - val_accuracy: 0.9814 - val_precision: 0.0819
```

```
- val_recall: 0.9036 - val_auc: 0.9637
Epoch 00011: early stopping
```

If the training process were considering the whole dataset on each gradient update, this oversampling would be basically identical to the class weighting.

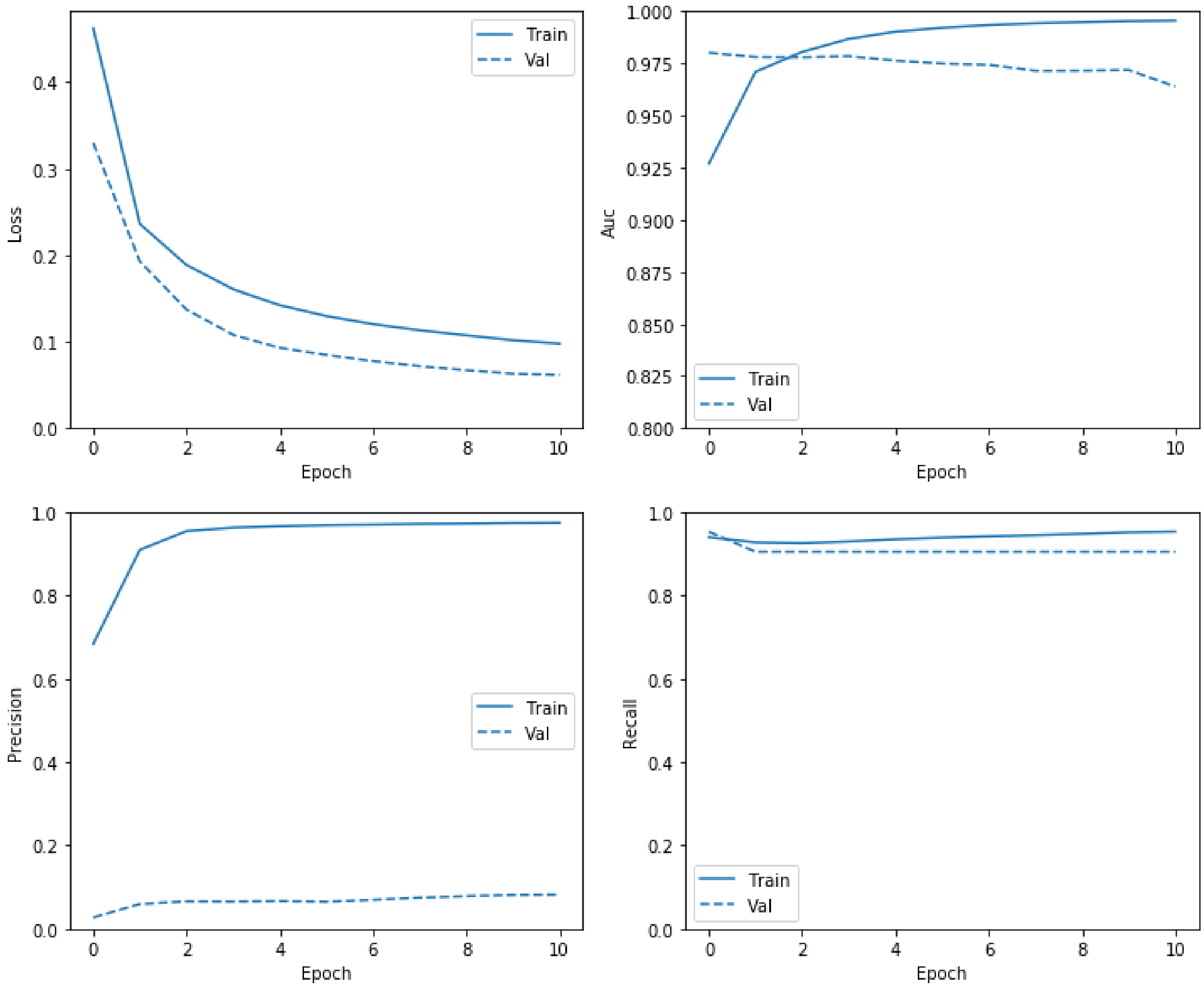
But when training the model batch-wise, as you did here, the oversampled data provides a smoother gradient signal: Instead of each positive example being shown in one batch with a large weight, they're shown in many different batches each time with a small weight.

This smoother gradient signal makes it easier to train the model.

Check training history

Note that the distributions of metrics will be different here, because the training data has a totally different distribution from the validation and test data.

In [46]:
```
plot_metrics(resampled_history )
```



Re-train

Because training is easier on the balanced data, the above training procedure may overfit quickly.

So break up the epochs to give the `callbacks.EarlyStopping` finer control over when to stop training.

In [47]:

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

resampled_history = resampled_model.fit(
    resampled_ds,
    # These are not real epochs
    steps_per_epoch = 20,
    epochs=10*EPOCHS,
    callbacks = [early_stopping],
    validation_data=(val_ds))
```

```
Train for 20 steps, validate for 23 steps
Epoch 1/1000
20/20 [==============================] - 4s 181ms/step - loss: 0.8800 - tp: 18783.0000 -
fp: 16378.0000 - tn: 4036.0000 - fn: 1763.0000 - accuracy: 0.5571 - precision: 0.5342 -
recall: 0.9142 - auc: 0.7752 - val_loss: 1.3661 - val_tp: 83.0000 - val_fp: 40065.0000 -
val_tn: 5421.0000 - val_fn: 0.0000e+00 - val_accuracy: 0.1208 - val_precision: 0.0021 -
val_recall: 1.0000 - val_auc: 0.9425
Epoch 2/1000
20/20 [==============================] - 1s 35ms/step - loss: 0.7378 - tp: 19613.0000 -
fp: 15282.0000 - tn: 5187.0000 - fn: 878.0000 - accuracy: 0.6055 - precision: 0.5621 - r
ecall: 0.9572 - auc: 0.8680 - val_loss: 1.1629 - val_tp: 83.0000 - val_fp: 36851.0000 -
val_tn: 8635.0000 - val_fn: 0.0000e+00 - val_accuracy: 0.1913 - val_precision: 0.0022 -
val_recall: 1.0000 - val_auc: 0.9580
Epoch 3/1000
20/20 [==============================] - 1s 39ms/step - loss: 0.6431 - tp: 19522.0000 -
fp: 13990.0000 - tn: 6558.0000 - fn: 890.0000 - accuracy: 0.6367 - precision: 0.5825 - r
ecall: 0.9564 - auc: 0.8950 - val_loss: 0.9853 - val_tp: 82.0000 - val_fp: 32268.0000 -
val_tn: 13218.0000 - val_fn: 1.0000 - val_accuracy: 0.2919 - val_precision: 0.0025 - val
_recall: 0.9880 - val_auc: 0.9660
Epoch 4/1000
20/20 [==============================] - 1s 39ms/step - loss: 0.5563 - tp: 19488.0000 -
fp: 12475.0000 - tn: 8032.0000 - fn: 965.0000 - accuracy: 0.6719 - precision: 0.6097 - r
ecall: 0.9528 - auc: 0.9135 - val_loss: 0.8430 - val_tp: 82.0000 - val_fp: 26633.0000 -
val_tn: 18853.0000 - val_fn: 1.0000 - val_accuracy: 0.4155 - val_precision: 0.0031 - val
_recall: 0.9880 - val_auc: 0.9713
Epoch 5/1000
20/20 [==============================] - 1s 37ms/step - loss: 0.4984 - tp: 19489.0000 -
fp: 11049.0000 - tn: 9377.0000 - fn: 1045.0000 - accuracy: 0.7047 - precision: 0.6382 -
recall: 0.9491 - auc: 0.9242 - val_loss: 0.7307 - val_tp: 82.0000 - val_fp: 20850.0000 -
val_tn: 24636.0000 - val_fn: 1.0000 - val_accuracy: 0.5424 - val_precision: 0.0039 - val
_recall: 0.9880 - val_auc: 0.9753
Epoch 6/1000
20/20 [==============================] - 1s 39ms/step - loss: 0.4463 - tp: 19305.0000 -
fp: 9622.0000 - tn: 10895.0000 - fn: 1138.0000 - accuracy: 0.7373 - precision: 0.6674 -
recall: 0.9443 - auc: 0.9336 - val_loss: 0.6405 - val_tp: 82.0000 - val_fp: 15843.0000 -
val_tn: 29643.0000 - val_fn: 1.0000 - val_accuracy: 0.6523 - val_precision: 0.0051 - val
_recall: 0.9880 - val_auc: 0.9773
Epoch 7/1000
20/20 [==============================] - 1s 40ms/step - loss: 0.4121 - tp: 19365.0000 -
fp: 8524.0000 - tn: 11931.0000 - fn: 1140.0000 - accuracy: 0.7641 - precision: 0.6944 -
recall: 0.9444 - auc: 0.9411 - val_loss: 0.5691 - val_tp: 82.0000 - val_fp: 11981.0000 -
val_tn: 33505.0000 - val_fn: 1.0000 - val_accuracy: 0.7371 - val_precision: 0.0068 - val
_recall: 0.9880 - val_auc: 0.9787
Epoch 8/1000
```

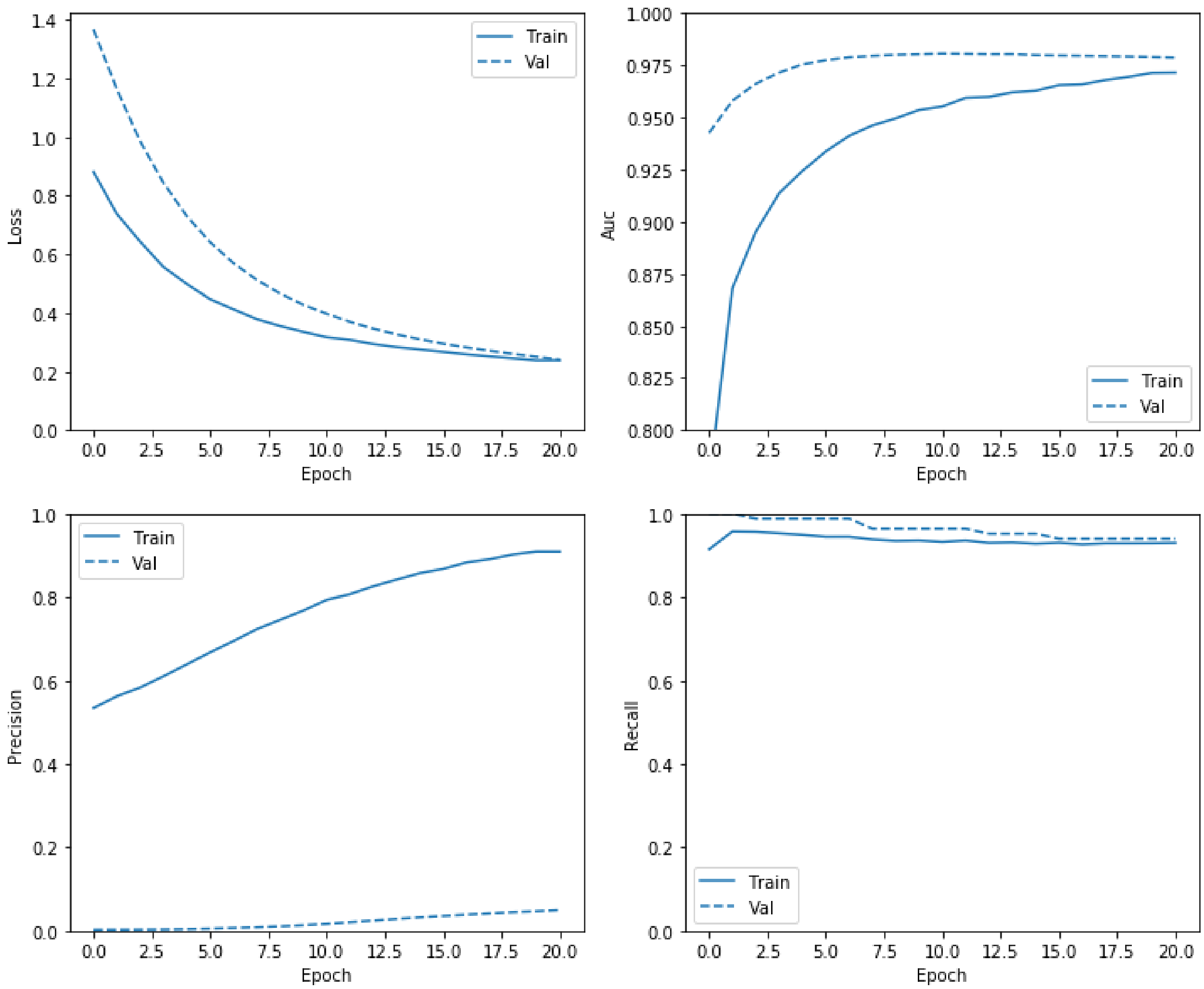
```
20/20 [==============================] - 1s 39ms/step - loss: 0.3784 - tp: 19242.0000 -
fp: 7375.0000 - tn: 13072.0000 - fn: 1271.0000 - accuracy: 0.7889 - precision: 0.7229 -
recall: 0.9380 - auc: 0.9461 - val_loss: 0.5120 - val_tp: 80.0000 - val_fp: 9309.0000 -
val_tn: 36177.0000 - val_fn: 3.0000 - val_accuracy: 0.7957 - val_precision: 0.0085 - val
_recall: 0.9639 - val_auc: 0.9794
Epoch 9/1000
20/20 [==============================] - 1s 45ms/step - loss: 0.3551 - tp: 19106.0000 -
fp: 6529.0000 - tn: 13989.0000 - fn: 1336.0000 - accuracy: 0.8080 - precision: 0.7453 -
recall: 0.9346 - auc: 0.9495 - val_loss: 0.4657 - val_tp: 80.0000 - val_fp: 7354.0000 -
val_tn: 38132.0000 - val_fn: 3.0000 - val_accuracy: 0.8386 - val_precision: 0.0108 - val
_recall: 0.9639 - val_auc: 0.9799
Epoch 10/1000
20/20 [==============================] - 1s 38ms/step - loss: 0.3350 - tp: 19149.0000 -
fp: 5794.0000 - tn: 14698.0000 - fn: 1319.0000 - accuracy: 0.8263 - precision: 0.7677 -
recall: 0.9356 - auc: 0.9535 - val_loss: 0.4275 - val_tp: 80.0000 - val_fp: 5832.0000 -
val_tn: 39654.0000 - val_fn: 3.0000 - val_accuracy: 0.8720 - val_precision: 0.0135 - val
_recall: 0.9639 - val_auc: 0.9802
Epoch 11/1000
20/20 [==============================] - 1s 40ms/step - loss: 0.3168 - tp: 19224.0000 -
fp: 5013.0000 - tn: 15322.0000 - fn: 1401.0000 - accuracy: 0.8434 - precision: 0.7932 -
recall: 0.9321 - auc: 0.9552 - val_loss: 0.3969 - val_tp: 80.0000 - val_fp: 4730.0000 -
val_tn: 40756.0000 - val_fn: 3.0000 - val_accuracy: 0.8961 - val_precision: 0.0166 - val
_recall: 0.9639 - val_auc: 0.9805
Epoch 12/1000
20/20 [==============================] - 1s 40ms/step - loss: 0.3077 - tp: 19028.0000 -
fp: 4564.0000 - tn: 16058.0000 - fn: 1310.0000 - accuracy: 0.8566 - precision: 0.8065 -
recall: 0.9356 - auc: 0.9593 - val_loss: 0.3695 - val_tp: 80.0000 - val_fp: 3819.0000 -
val_tn: 41667.0000 - val_fn: 3.0000 - val_accuracy: 0.9161 - val_precision: 0.0205 - val
_recall: 0.9639 - val_auc: 0.9804
Epoch 13/1000
20/20 [==============================] - 1s 40ms/step - loss: 0.2936 - tp: 19047.0000 -
fp: 4028.0000 - tn: 16444.0000 - fn: 1441.0000 - accuracy: 0.8665 - precision: 0.8254 -
recall: 0.9297 - auc: 0.9597 - val_loss: 0.3461 - val_tp: 79.0000 - val_fp: 3149.0000 -
val_tn: 42337.0000 - val_fn: 4.0000 - val_accuracy: 0.9308 - val_precision: 0.0245 - val
_recall: 0.9518 - val_auc: 0.9802
Epoch 14/1000
20/20 [==============================] - 1s 38ms/step - loss: 0.2829 - tp: 19087.0000 -
fp: 3596.0000 - tn: 16855.0000 - fn: 1422.0000 - accuracy: 0.8775 - precision: 0.8415 -
recall: 0.9307 - auc: 0.9619 - val_loss: 0.3266 - val_tp: 79.0000 - val_fp: 2691.0000 -
val_tn: 42795.0000 - val_fn: 4.0000 - val_accuracy: 0.9409 - val_precision: 0.0285 - val
_recall: 0.9518 - val_auc: 0.9803
Epoch 15/1000
20/20 [==============================] - 1s 39ms/step - loss: 0.2748 - tp: 19020.0000 -
fp: 3174.0000 - tn: 17283.0000 - fn: 1483.0000 - accuracy: 0.8863 - precision: 0.8570 -
recall: 0.9277 - auc: 0.9627 - val_loss: 0.3095 - val_tp: 79.0000 - val_fp: 2360.0000 -
val_tn: 43126.0000 - val_fn: 4.0000 - val_accuracy: 0.9481 - val_precision: 0.0324 - val
_recall: 0.9518 - val_auc: 0.9797
Epoch 16/1000
20/20 [==============================] - 1s 40ms/step - loss: 0.2666 - tp: 18890.0000 -
fp: 2889.0000 - tn: 17757.0000 - fn: 1424.0000 - accuracy: 0.8947 - precision: 0.8673 -
recall: 0.9299 - auc: 0.9653 - val_loss: 0.2945 - val_tp: 78.0000 - val_fp: 2101.0000 -
val_tn: 43385.0000 - val_fn: 5.0000 - val_accuracy: 0.9538 - val_precision: 0.0358 - val
_recall: 0.9398 - val_auc: 0.9796
Epoch 17/1000
20/20 [==============================] - 1s 38ms/step - loss: 0.2583 - tp: 18959.0000 -
fp: 2517.0000 - tn: 17973.0000 - fn: 1511.0000 - accuracy: 0.9017 - precision: 0.8828 -
recall: 0.9262 - auc: 0.9657 - val_loss: 0.2817 - val_tp: 78.0000 - val_fp: 1929.0000 -
val_tn: 43557.0000 - val_fn: 5.0000 - val_accuracy: 0.9576 - val_precision: 0.0389 - val
_recall: 0.9398 - val_auc: 0.9794
Epoch 18/1000
20/20 [==============================] - 1s 46ms/step - loss: 0.2511 - tp: 19104.0000 -
fp: 2344.0000 - tn: 18043.0000 - fn: 1469.0000 - accuracy: 0.9069 - precision: 0.8907 -
recall: 0.9286 - auc: 0.9678 - val_loss: 0.2704 - val_tp: 78.0000 - val_fp: 1787.0000 -
val_tn: 43699.0000 - val_fn: 5.0000 - val_accuracy: 0.9607 - val_precision: 0.0418 - val
_recall: 0.9398 - val_auc: 0.9793
```

```
Epoch 19/1000
20/20 [==============================] - 1s 40ms/step - loss: 0.2445 - tp: 19183.0000 -
fp: 2087.0000 - tn: 18215.0000 - fn: 1475.0000 - accuracy: 0.9130 - precision: 0.9019 -
recall: 0.9286 - auc: 0.9693 - val_loss: 0.2598 - val_tp: 78.0000 - val_fp: 1665.0000 -
val_tn: 43821.0000 - val_fn: 5.0000 - val_accuracy: 0.9634 - val_precision: 0.0448 - val
_recall: 0.9398 - val_auc: 0.9791
Epoch 20/1000
20/20 [==============================] - 1s 39ms/step - loss: 0.2373 - tp: 18995.0000 -
fp: 1906.0000 - tn: 18602.0000 - fn: 1457.0000 - accuracy: 0.9179 - precision: 0.9088 -
recall: 0.9288 - auc: 0.9712 - val_loss: 0.2500 - val_tp: 78.0000 - val_fp: 1587.0000 -
val_tn: 43899.0000 - val_fn: 5.0000 - val_accuracy: 0.9651 - val_precision: 0.0468 - val
_recall: 0.9398 - val_auc: 0.9788
Epoch 21/1000
19/20 [===========================>..] - ETA: 0s - loss: 0.2378 - tp: 18121.0000 - fp: 1
821.0000 - tn: 17599.0000 - fn: 1371.0000 - accuracy: 0.9180 - precision: 0.9087 - recal
l: 0.9297 - auc: 0.9714Restoring model weights from the end of the best epoch.
20/20 [==============================] - 1s 40ms/step - loss: 0.2376 - tp: 19083.0000 -
fp: 1918.0000 - tn: 18513.0000 - fn: 1446.0000 - accuracy: 0.9179 - precision: 0.9087 -
recall: 0.9296 - auc: 0.9714 - val_loss: 0.2401 - val_tp: 78.0000 - val_fp: 1485.0000 -
val_tn: 44001.0000 - val_fn: 5.0000 - val_accuracy: 0.9673 - val_precision: 0.0499 - val
_recall: 0.9398 - val_auc: 0.9785
Epoch 00021: early stopping
```

Re-check training history

In [48]:

```
plot_metrics(resampled_history)
```



Evaluate metrics

In [49]:

```
# TODO 1
train_predictions_resampled = resampled_model.predict(train_features, batch_size=BATCH_
test_predictions_resampled = resampled_model.predict(test_features, batch_size=BATCH_SI
```

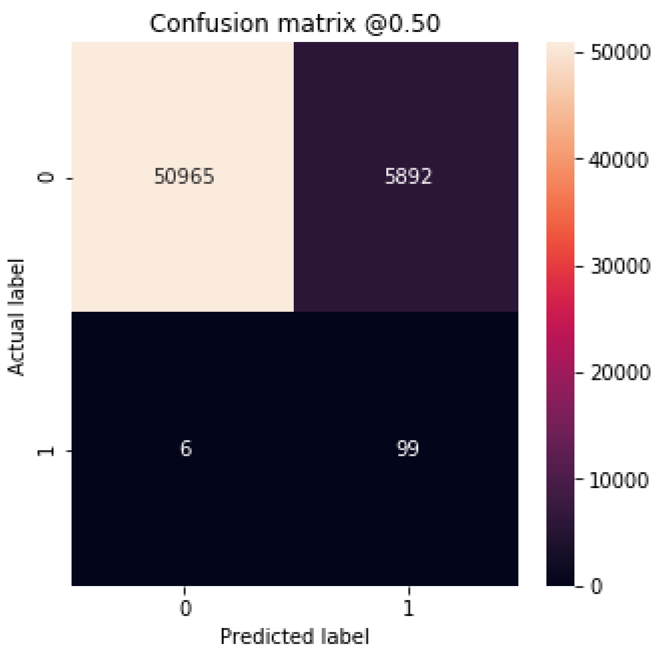
In [50]:

```
resampled_results = resampled_model.evaluate(test_features, test_labels,
                                             batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(resampled_model.metrics_names, resampled_results):
  print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_resampled)
```

```
loss :  0.3960801533448772
tp :  99.0
fp :  5892.0
tn :  50965.0
fn :  6.0
accuracy :  0.8964573
precision :  0.016524788
recall :  0.94285715
auc :  0.9804354

Legitimate Transactions Detected (True Negatives):  50965
Legitimate Transactions Incorrectly Detected (False Positives):  5892
Fraudulent Transactions Missed (False Negatives):  6
Fraudulent Transactions Detected (True Positives):  99
Total Fraudulent Transactions:  105
```



Plot the ROC

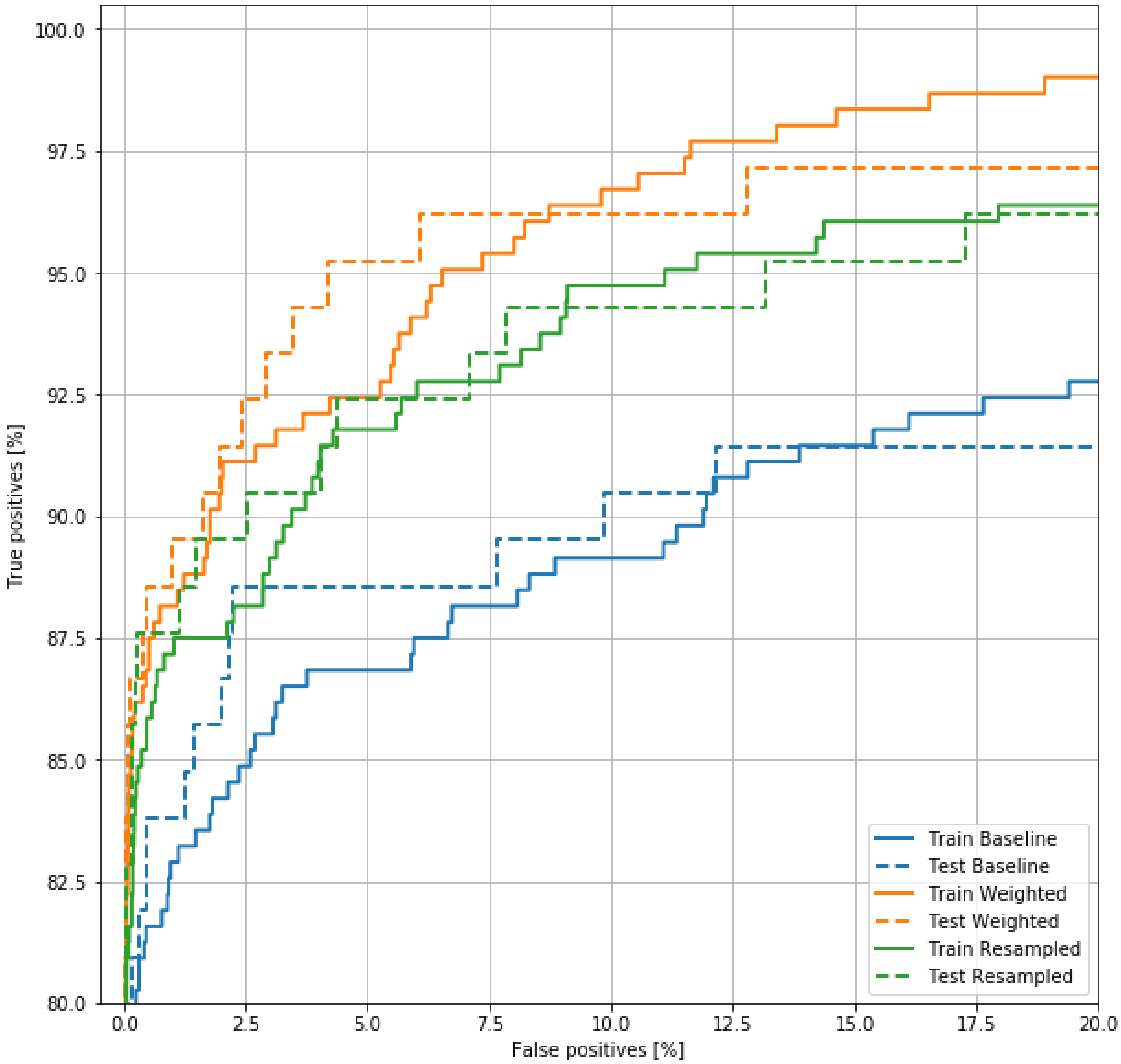
In [51]:

```
plot_roc("Train Baseline", train_labels, train_predictions_baseline, color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline, color=colors[0], line

plot_roc("Train Weighted", train_labels, train_predictions_weighted, color=colors[1])
plot_roc("Test Weighted", test_labels, test_predictions_weighted, color=colors[1], line
```

```
plot_roc("Train Resampled", train_labels, train_predictions_resampled,  color=colors[2]
plot_roc("Test Resampled", test_labels, test_predictions_resampled,  color=colors[2], l
plt.legend(loc='lower right')
```

Out[51]: <matplotlib.legend.Legend at 0x7f5eebd220b8>



Applying this tutorial to your problem

Imbalanced data classification is an inherantly difficult task since there are so few samples to learn from. You should always start with the data first and do your best to collect as many samples as possible and give substantial thought to what features may be relevant so the model can get the most out of your minority class. At some point your model may struggle to improve and yield the results you want, so it is important to keep in mind the context of your problem and the trade offs between different types of errors.