Try yourself: Effect of adding dropout layers:

In this short notebook, we will look at the effects of adding dropout to a simple classification problem. Howver, this notebook will not run correctly, until you fill in the areas that says "TODO" of the code. Often the TODO's offer hints / explaination of what you are suppposed to do.

- For simplicity, just use "ctrl+f" to find all "TODOs" in the text.

If you get stuck, read up on the the paper, or look at Prof. Brownlee's example:

Paper on dropout.

- https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

This exercise is inspired by the work of Prof. Jason Brownlee:

- https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/

In [1]:
```
# Requirements
import pandas as pd
from tensorflow.keras.callbacks import Callback
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
```

The dataset

The dataset is from UCI: https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks), and is a collection of raw sonar data. Sonar was bounced of the surface of rocks, as well as metal cylinders, and our task is to classify the two.

Each datapoint is an array of 60 sonar reading, in the range 0-1, representing the energy of which the sonar reading was recorded. Not a trivial task!

In [7]:
```
# Load dataset
dataframe = pd.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
x_train = dataset[:,0:60].astype(float)
y_train = dataset[:,60]
# Encode class values as integers
encoder = LabelEncoder()
encoder.fit(y_train)
y_train = encoder.transform(y_train)
# Standard scale
std_scale = StandardScaler().fit(x_train)
x_scaled = std_scale.transform(x_train)
# Print the shapes
print("Training set shape:", x_scaled.shape, "\tTraining labels shape:", y_train.shape)
```

```
Training set shape: (208, 60)   Training labels shape: (208,)
```

Some util functions, don't really focus too much on these

In [8]:
```
""" Some util functions """
class PrintDot(Callback):
    """Replaces the normal verbose with a simple dot printer."""
    def on_epoch_end(self, epoch, logs):
        if epoch % 100 == 0: print('')
        print('.', end='')

def plot_history(hist):
    """Plots the loss and val loss of a training history (pandas format)"""
    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.plot(hist['epoch'], hist['loss'], label='Train Error')
    plt.plot(hist['epoch'], hist['val_loss'], label = 'Val Error')
    plt.legend()
```

Baseline model

Before doing anything with dropout layers, lets create a baseline model to compare to.

The model is a basic sequential model with two dense layers, and one output layer with a .... activation function for classification.

The loss will be a binary cross entropy.

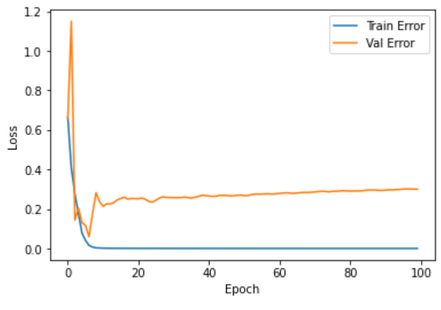
In [12]:
```
# Create baseline_model
baseline_model = Sequential()
baseline_model.add(Dense(60, input_shape=(60,), activation='relu'))
baseline_model.add(Dense(30,  activation='relu'))
# TODO-START: What activation function should we use for binary classifcation?
baseline_model.add(Dense(1, activation=...))
# TODO-END.
# Compile baseline_model
sgd = SGD(learning_rate=0.1, momentum=0.9)
baseline_model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
```

Training and plotting the results.

In [13]:
```
history = baseline_model.fit(x=x_scaled, y=y_train,
    epochs=100,
    shuffle=True,
    validation_split=0.1,
    verbose=0,
    callbacks=[PrintDot()])

hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
display(hist.tail(1))

plot_history(hist)
```

```
....................................................................................................
```

	loss	accuracy	val_loss	val_accuracy	epoch
99	0.000086	1.0	0.299625	0.809524	99



You should expect a result like this: with smooth loss-curves, and maybe you can spot how the val loss increases as we slowly overfit the model more and more.

Baseline

As expected, over time our model will begin overfitting, even if the train loss is constant/decreasing, the validation error stagnate / increase. The training accuracy should be around 100%, with a validation accuracy in the range 70-90%.

Adding dropout to input layer

Now we will try to see the effect of adding a dropout layer - to the input layer!

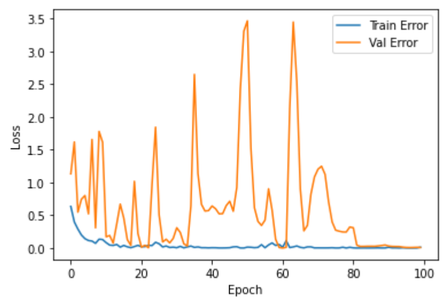
Dropout in keras works like this: adding a "dropout layer" will make the previous layer drop nodes at random.

- So if you want to drop inputs, we need to add dropout as our first layer in the sequential model.

In [14]:
```
from tensorflow.keras.layers import Dropout

# Define model
p = 0.1 # Dropout rate
model = Sequential()
# TODO-START: add a dropout layer to the input layer.
# HINT: remember input_shape=(60,)
# HINT2: look at https://keras.io/api/layers/regularization_layers/dropout/
...
...
...
# TODO-END.
model.add(Dense(60, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
sgd = SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])

# Train model
history = model.fit(x=x_scaled, y=y_train,
    epochs=100,
    shuffle=True,
    validation_split=0.1,
    verbose=0,
    callbacks=[PrintDot()])


hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
display(hist.tail(1))
plot_history(hist)
```

```
....................................................................................................
```

	loss	accuracy	val_loss	val_accuracy	epoch
99	0.012008	0.994652	0.008295	1.0	99



IF you have done it correctly you should see an output something like this:

Right away we should notice that both the training loss and validation loss is a lot more "spikey":

TODO: EXPLAIN why you think we see the spikey pattern now that we use dropout on the input data?

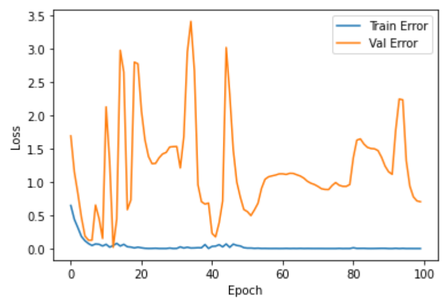
Add dropout layers to hidden layers

Next we will try to add dropout layers in all hidden layers.

In [15]:
```
from tensorflow.keras.layers import Dropout

# Define model
p = 0.1 # Dropout rate
model = Sequential()
model.add(Dense(60, input_shape=(60,), activation='relu'))
# TODO-START: Add dropout layer here
...
# TODO-END.
model.add(Dense(30, activation='relu'))
# TODO-START: Add dropout layer here
...
# TODO-END.
model.add(Dense(1, activation='sigmoid'))

# Compile model
sgd = SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])

# Train model
history = model.fit(x=x_scaled, y=y_train,
    epochs=100,
    shuffle=True,
    validation_split=0.1,
    verbose=0,
    callbacks=[PrintDot()])


hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
display(hist.tail(1))
plot_history(hist)
```

.....

	loss	accuracy	val_loss	val_accuracy	epoch
99	0.000151	1.0	0.703509	0.857143	99



Again we notice the same spikey patterns, even as all input data is available for all training samples in this case.

TODO: Why do you think we get all these spikes now, even if we do not drop any of the input data?

Unlocking the full potential of dropout

First of all we will limit the norms of the weights in the dense layers, to 3 - this is due to a suggestion in the paper.

- This is done to counter some of the most extreme "spikey" behaviour in the net. If a node has an extremely large weight attached to it, it means it relies / is relied upon a lot, and will most likely cause a large spike when dropped.

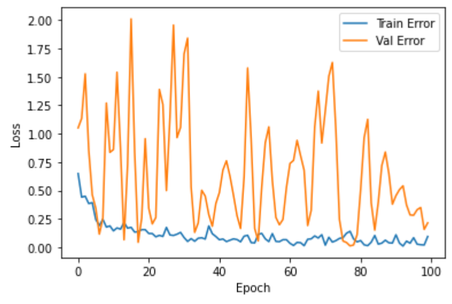
Second, we will increase the net size. A problem with dropout is that we are always training with a net that is reduced by 10-50% in size (depending on dropout rate), and so we should increase the net with the same amount to get as a precise net architecture.

We will also use dropout on both visible and hidden layers.

In [16]:
```
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm


# Define model
p = 0.2     # Dropout rate
max_norm=4  # Max weight norm
model = Sequential()
model.add(Dropout(p, input_shape=(60,)))
# TODO: Add the MAXNORM constraint to the dense layer
model.add(Dense(60, activation='relu', kernel_constraint=...))
model.add(Dropout(p))
# TODO: Add the MAXNORM constraint to the dense layer
model.add(Dense(60, activation='relu', kernel_constraint=...))
model.add(Dropout(p))
model.add(Dense(1, activation='sigmoid'))

# Compile model
sgd = SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])

# Train model
history = model.fit(x=x_scaled, y=y_train,
    epochs=100,
    shuffle=True,
    validation_split=0.1,
    verbose=0,
    callbacks=[PrintDot()])


hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
display(hist.tail(1))
plot_history(hist)
```

.....

	loss	accuracy	val_loss	val_accuracy	epoch
99	0.095551	0.957219	0.21563	0.904762	99



Is the dropout layers worth it?

- The dropout layer will cause a very "spikey" behaviour in loss performance, and sometimes the model in the final iteration will not impress with its val_loss.
- Generally, training a dropout layer is slower, and learns a less precise / more spartse solution.
  - **TODO**: Explain why this might be the case
- However, we might want to combine dropout with another regularization, for better performance:...

- This can be done using a callback function in keras (called "model checkpoint"), which will retrieve the model with the best validation accuracy.
  - Alternatively use the callback "early stopping" which has a function to restore the best weights to the current model

Early stopping / weight retrieval: **this part is voluntary**

In [35]:
```
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import EarlyStopping

# Callback 1: the model checkpoint - saves a model to be retrieved later
mcp_save = ModelCheckpoint('.best_model.hdf5', save_best_only=True, monitor='val_loss', mode='min')
# Callback 2: Stops the training if the val loss climbs over time, and saves best weigths to model.
early_stopping_monitor = EarlyStopping(
    monitor='val_loss', # What should be monitored by the stopper?
    min_delta=0,    # What is concidered a drop in validation data performance?
    patience=20,    # How long the stopper will wait before calling off the training?
    verbose=1,      #
    mode='auto',
    baseline=None,  # Could add baseline to compare with
    restore_best_weights=True # Restore best weights with regard to validation data, to the model
```

```
)

# Define model
p = 0.2 # Dropout rate
model = Sequential()
model.add(Dropout(p, input_shape=(60,)))
model.add(Dense(60, activation='relu', kernel_constraint=MaxNorm(4)))
model.add(Dropout(p))
model.add(Dense(60, activation='relu', kernel_constraint=MaxNorm(4)))
model.add(Dropout(p))
model.add(Dense(1, activation='sigmoid'))

# Compile model
sgd = SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])

# Train model
history = model.fit(x=x_scaled, y=y_train,
    epochs=100,
    shuffle=True,
    validation_split=0.1,
    verbose=0,
    # TODO: Add one or both of our new callback functions defined above
    callbacks=[PrintDot(), ..., ...])


hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
display(hist.tail(1))
plot_history(hist)
```

```
.................Restoring model weights from the end of the best epoch: 8.
Epoch 18: early stopping
```

	loss	accuracy	val_loss	val_accuracy	epoch
17	0.140995	0.962567	0.135012	0.952381	17

In [ ]: