Neural networks

**Hello nets!**

You're going to build a simple neural network to get a feeling of how quickly it is to accomplish this in Keras.

You will build a network that **takes two numbers as an input**, passes them through **a hidden layer of 10 neurons**, and finally **outputs a single non-constrained number**.

A **non-constrained output can be obtained by avoiding setting an activation function in the output layer**. This is useful for problems like regression, when we want our output to be able to take any non-constrained value.

Diagram, schematic

Description automatically generated

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Import the Sequential model from keras.models and the Denselayer from keras.layers.

Create an instance of the Sequential model.

Add a 10-neuron hidden Dense layer with an input\_shape of two neurons.

Add a final 1-neuron output layer and summarize your model with summary()

# Import the Sequential model and Dense layer

from keras.models import Sequential

from keras.layers import Dense

# Create a Sequential model

model = Sequential()

# Add an input layer and a hidden layer with 10 neurons

model.add(Dense(10, input\_shape=(2,), activation="relu"))

# Add a 1-neuron output layer

model.add(Dense(1))

# Summarise your model

model.summary()

**Hello nets!**

You're going to build a simple neural network to get a feeling of how quickly it is to accomplish this in Keras.

You will build a network that **takes two numbers as an input**, passes them through **a hidden layer of 10 neurons**, and finally **outputs a single non-constrained number**.

A **non-constrained output can be obtained by avoiding setting an activation function in the output layer**. This is useful for problems like regression, when we want our output to be able to take any non-constrained value.

Diagram, schematic

Description automatically generated

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Import the Sequential model from keras.models and the Denselayer from keras.layers.

Create an instance of the Sequential model.

Add a 10-neuron hidden Dense layer with an input\_shape of two neurons.

Add a final 1-neuron output layer and summarize your model with summary().

# Import the Sequential model and Dense layer

from keras.models import Sequential

from keras.layers import Dense

# Create a Sequential model

model = Sequential()

# Add an input layer and a hidden layer with 10 neurons

model.add(Dense(10, input\_shape=(2,), activation="relu"))

# Add a 1-neuron output layer

model.add(Dense(1))

# Summarise your model

model.summary()

# Counting parameters

You've just created a neural network. But you're going to create a new one now, taking some time to think about the weights of each layer. The Keras Dense layer and the Sequential model are already loaded for you to use.

This is the network you will be creating:

Diagram

Description automatically generated

#### Question

Given the model you just built, which answer is correct regarding the number of weights (parameters) in the **hidden layer**?

##### Possible Answers



There are 20 parameters, 15 from the connections of our inputs to our hidden layer and 5 from the bias weight of each neuron in the hidden layer.

# Import the Sequential model and Dense layer

from keras.models import Sequential

from keras.layers import Dense

# Create a Sequential model

model = Sequential()

# Add an input layer and a hidden layer with 10 neurons

model.add(Dense(5, input\_shape=(3,), activation="relu"))

# Add a 1-neuron output layer

model.add(Dense(1))

# Summarise your model

model.summary()

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_1 (Dense) (None, 5) 20

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 1) 6

=================================================================

Total params: 26

Trainable params: 26

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Build as shown!**

You will take on a final challenge before moving on to the next lesson. Build the network shown in the picture below. Prove your mastered Keras basics in no time!

Shape, polygon

Description automatically generated

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Instantiate a Sequential model.

Build the input and hidden layer.

Add the output layer.

from keras.models import Sequential

from keras.layers import Dense

# Instantiate a Sequential model

model = Sequential()

# Build the input and hidden layer

model.add(Dense(3,input\_shape=(2,),activation='relu'))

# Add the ouput layer

model.add(Dense(1))

**Specifying a model**

You will build a simple regression model to predict the orbit of the meteor!

Your training data consist of measurements taken at time steps from **-10 minutes before the impact region to +10 minutes after**. Each time step can be viewed as an X coordinate in our graph, which has an associated position Y for the meteor orbit at that time step.

*Note that you can view this problem as approximating a quadratic function via the use of neural networks.*

Chart

Description automatically generated

This data is stored in two numpy arrays: one called time\_steps , what we call *features*, and another called y\_positions, with the *labels*. Go on and build your model! It should be able to predict the y positions for the meteor orbit at future time steps.

Keras Sequential model and Dense layers are available for you to use.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Instantiate a Sequential model.

Add a Dense layer of 50 neurons with an input shape of 1 neuron.

Add two Dense layers of 50 neurons each and 'relu' activation.

End your model with a Dense layer with a single neuron and no activation.

# Instantiate a Sequential model

model = Sequential()

# Add a Dense layer with 50 neurons and an input of 1 neuron

model.add(Dense(50, input\_shape=(1,), activation='relu'))

# Add two Dense layers with 50 neurons and relu activation

model.add(Dense(50, activation='relu'))

model.add(Dense(50, activation='relu'))

# End your model with a Dense layer and no activation

model.add(Dense(1))

**Training**

You're going to train your first model in this course, and for a good cause!

Remember that **before training your Keras models you need to compile them**. This can be done with the .compile() method. The .compile() method takes arguments such as the optimizer, used for weight updating, and the loss function, which is what we want to minimize. Training your model is as easy as calling the .fit() method, passing on the *features*, *labels* and a number of *epochs* to train for.

The regression model you built in the previous exercise is loaded for you to use, along with the time\_steps and y\_positions data. Train it and evaluate it on this very same data, let's see if your model can learn the meteor's trajectory.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Compile your model making use of the 'adam' optimizer and 'mse' as your loss function.

Fit your model using the features and labels for 30 epochs.

Evaluate your model with the .evaluate() method, passing the features and labels used during training.

# Compile your model

model.compile(optimizer  = 'adam', loss = "mse")

print("Training started..., this can take a while:")

# Fit your model on your data for 30 epochs

model.fit(time\_steps,y\_positions, epochs = 30)

# Evaluate your model

print("Final loss value:",model.evaluate(time\_steps,y\_positions))

# Predicting the orbit!

You've already trained a model that approximates the orbit of the meteor approaching Earth and it's loaded for you to use.

Since you trained your model for values between -10 and 10 minutes, your model hasn't yet seen any other values for different time steps. You will now visualize how your model behaves on unseen data.

If you want to check the source code of plot\_orbit, paste show\_code(plot\_orbit) into the console.

Hurry up, the Earth is running out of time!

Remember *np.arange(x,y)* produces a range of values from ***x*** to ***y-1***. That is the *[x, y)* interval.

##### Instructions 1/2

**50 XP**

Use the model's .predict() method to predict from -10 to 10 minutes.

###### Incorrect Submission

Check your call of np.arange(). Did you correctly specify the second argument? Expected 11, but got 9.

Did you find this feedback helpful?

**YesNo**

Use the model's .predict() method to predict from -40 to 40 minutes.

# Predict the twenty minutes orbit

twenty\_min\_orbit = model.predict(np.arange(-10, 11))

# Plot the twenty minute orbit

plot\_orbit(twenty\_min\_orbit)

**Exploring dollar bills**

You will practice building classification models in Keras with the **Banknote Authentication** dataset.

Your goal is to distinguish between real and fake dollar bills. In order to do this, the dataset comes with 4 features: variance,skewness,kurtosis and entropy. These features are calculated by applying mathematical operations over the dollar bill images. The labels are found in the dataframe's class column.

A close-up of a dollar bill

Description automatically generated with medium confidence

A pandas DataFrame named banknotes is ready to use, let's do some data exploration!

**Instructions**

**100 XP**

Import seaborn as sns.

Use seaborn's pairplot() on banknotes and set hue to be the name of the column containing the labels.

Generate descriptive statistics for the banknotes authentication data.

Count the number of observations per label with .value\_counts().

# Show the plot

plt.show()

# Describe the data

print('Dataset stats: \n', banknotes.describe())

# Count the number of observations per class

print('Observations per class: \n', banknotes['class'].value\_counts())

Shape

Description automatically generated

<script.py> output:

Dataset stats:

variance skewness curtosis entropy

count 96.000000 96.000000 96.000000 96.000000

mean -0.057791 -0.102829 0.230412 0.081497

std 1.044960 1.059236 1.128972 0.975565

min -2.084590 -2.621646 -1.482300 -3.034187

25% -0.839124 -0.916152 -0.415294 -0.262668

50% -0.026748 -0.037559 -0.033603 0.394888

75% 0.871034 0.813601 0.978766 0.745212

max 1.869239 1.634072 3.759017 1.343345

Observations per class:

real 53

fake 43

Name: class, dtype: int64

**A binary classification model**

Now that you know what the **Banknote Authentication** dataset looks like, we'll build a simple model to distinguish between real and fake bills.

You will perform binary classification by using a single neuron as an output. The input layer will have 4 neurons since we have 4 features in our dataset. The model's output will be a value constrained between 0 and 1.

We will interpret this output number as the probability of our input variables coming from a fake dollar bill, with 1 meaning we are certain it's a fake bill.

![A picture containing chart

Description automatically generated](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgAAD/4RDsRXhpZgAATU0AKgAAAAgABAE7AAIAAAALAAAISodpAAQAAAABAAAIVpydAAEAAAAWAAAQzuocAAcAAAgMAAAAPgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEJlbGxhZG9ubmEAAAAFkAMAAgAAABQAABCkkAQAAgAAABQAABC4kpEAAgAAAAM0MgAAkpIAAgAAAAM0MgAA6hwABwAACAwAAAiYAAAAABzqAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMjAxOTowNDoyMCAxNjozMzo1NwAyMDE5OjA0OjIwIDE2OjMzOjU3AAAAQgBlAGwAbABhAGQAbwBuAG4AYQAAAP/hCx1odHRwOi8vbnMuYWRvYmUuY29tL3hhcC8xLjAvADw/eHBhY2tldCBiZWdpbj0n77u/JyBpZD0nVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkJz8+DQo8eDp4bXBtZXRhIHhtbG5zOng9ImFkb2JlOm5zOm1ldGEvIj48cmRmOlJERiB4bWxuczpyZGY9Imh0dHA6Ly93d3cudzMub3JnLzE5OTkvMDIvMjItcmRmLXN5bnRheC1ucyMiPjxyZGY6RGVzY3JpcHRpb24gcmRmOmFib3V0PSJ1dWlkOmZhZjViZGQ1LWJhM2QtMTFkYS1hZDMxLWQzM2Q3NTE4MmYxYiIgeG1sbnM6ZGM9Imh0dHA6Ly9wdXJsLm9yZy9kYy9lbGVtZW50cy8xLjEvIi8+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczp4bXA9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8iPjx4bXA6Q3JlYXRlRGF0ZT4yMDE5LTA0LTIwVDE2OjMzOjU3LjQyMDwveG1wOkNyZWF0ZURhdGU+PC9yZGY6RGVzY3JpcHRpb24+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczpkYz0iaHR0cDovL3B1cmwub3JnL2RjL2VsZW1lbnRzLzEuMS8iPjxkYzpjcmVhdG9yPjxyZGY6U2VxIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpsaT5CZWxsYWRvbm5hPC9yZGY6bGk+PC9yZGY6U2VxPg0KCQkJPC9kYzpjcmVhdG9yPjwvcmRmOkRlc2NyaXB0aW9uPjwvcmRmOlJERj48L3g6eG1wbWV0YT4NCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgPD94cGFja2V0IGVuZD0ndyc/Pv/bAEMABwUFBgUEBwYFBggHBwgKEQsKCQkKFQ8QDBEYFRoZGBUYFxseJyEbHSUdFxgiLiIlKCkrLCsaIC8zLyoyJyorKv/bAEMBBwgICgkKFAsLFCocGBwqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKv/AABEIAdECcwMBIgACEQEDEQH/xAAfAAABBQEBAQEBAQAAAAAAAAAAAQIDBAUGBwgJCgv/xAC1EAACAQMDAgQDBQUEBAAAAX0BAgMABBEFEiExQQYTUWEHInEUMoGRoQgjQrHBFVLR8CQzYnKCCQoWFxgZGiUmJygpKjQ1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4eLj5OXm5+jp6vHy8/T19vf4+fr/xAAfAQADAQEBAQEBAQEBAAAAAAAAAQIDBAUGBwgJCgv/xAC1EQACAQIEBAMEBwUEBAABAncAAQIDEQQFITEGEkFRB2FxEyIygQgUQpGhscEJIzNS8BVictEKFiQ04SXxFxgZGiYnKCkqNTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqCg4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2dri4+Tl5ufo6ery8/T19vf4+fr/2gAMAwEAAhEDEQA/APpGiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKgvXuo7KVtPhhnuQv7uOeYxIx9C4ViB77TQBPRXF/D/xrq/jWO+uLzQbXS7SzupbMuuotPI8sbAHC+Uo28nndnjp3rtKPMVwooooGFFFU4tWs5tauNKimV7y2hSaaNTny1ckLn0J2mgC5RWdr17qGnaLcXek2MF9cQqX8ie5MCsoGThwj8+gxj3FY3w98Wan418Mwa7faPb6XaXab7ZI743EjDJB3Dy0C9OME9e1C1v5A9LHVUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRUF5BJdWcsEF3NZyOuFuIAheM+oDqy5+oIoA8s+HUYl+F3jBWZ1/wCJtqZBjdkIIJ7gg1ky6VbWnwk8B+JIfNGsQTaWsd4ZWLhJHRHTrjYVYgr0PU85Ndvo3wuh0LR9U0yw8U6+LbVGkknDtasRJIQXdT5GQSAR6cnABwQk/wALLa48Iab4bbxNry2GmyJJAVa2D/uypjBbyeQhXI+vOeMKN1b/ALc/8l3FbRrvzfjsZ2o6Paat8dptP1B55bObw4JJLY3DhJD9pIIIz04GQMA4Gc1xup6Fa2PhT4p+FZIPtejaFAl5pK3P737HJJbs5WNmyRtPT/ePqa3b+70q7/aFhsB4pe01GHw+tol1BPCJTP55JjZWUxlypB2FPcAcV6Jb+ENKi0LUNLmjkuo9U3/b5p2zJdM67WZmGOcYAwAAAAAAKLXp6dU//SnY0ulUd/7vy91X/wAvv7HE+JtPs/7U+GOo/ZYftv2+GD7T5Y8zy/s0jbN3XbnnHTNTaH4d0V/jx4ouH0ewae3tLG4hlNqhaOVvN3OpxkMcDJHJxWwnww0prXSIL7U9Zv8A+x7hZ7SS4vPnTapVVyoXgA9RhjgZJxWs3hCx/wCExbxHFc3sNzJbpBPBHNiGcISULjGSVye4BzyDVtpyuu7f3r/MxSfLZ9kvuf8Akaerf8gW9/695P8A0E1x/wAFP+SL+Gv+vU/+htXWa3pb6zpM1gmo3enCYbWns/L8zaeCAXRgMjvjPoRXNaJ8OpvDuh2+kaN408Q21jbKUhi2WT7ASTjc1sT1J71K05vO34X/AMymrtPtf8bf5DvHMdhf3+h2l7PJeKbqQnRIYlk/tIiM/K4YhQiZDnd8vTuRXO+C/D1nrun+OvC2s6fDHpa6r5UWno++O1VoY3Aj4G3DHcAAAD09a6i7+Hmn3dvo3/Ez1WC90d3eDUYp1FxIZP8AWeYSpVt/f5R7Yoi0XQvh9/bviaW8v47e4AuL3zZXnVNqhS4UAsSQBknd7Y5pWSvfz/T/ACKu3a39b/5nE+F7HTbzwGvw11HTrT7bY6h9hvIRCoDxofN+049XjA+brvatTUobnWvH2qeHYdA0bVdN0uwtRBp+pXjW8UYffl0jWCQMflCg8bduB1NdD4atdN1rxRfeM7G2Ki8tYrO2uJImjaaJSWL7WAIBLADIGQmemKk8UfD/AEzxPqltqjXeo6TqtsnlJqGlXPkTGMnJjY4IZc9iKp305vn91v8Ag+WwlbW3y++//APP9Z0LXNE+G+j6br+oo91b+JbSKBrK8kl8mB51KRszKu9lBxll6AV6poPhrS/DUd3Ho8MkK3lwbmYPO8m6QgAn5ycZxWJrvw4sdc0Ox0ptY1izgs51ut8E6PLPMpDLJJJKjsxBGeoHtgDHV20TwWsUUs8ly8aBWmlCh5CB947QFyevAA9AKFon6/hZfqri7f11f+ZLRRRSGFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUyaaO3head1jjRSzMxwAPWgG7K7H1kal4p0fS2KXN4rSjrHF87D646fjXDeJfG9zqUj22ms1vaZxuBw8v19B7fnXJ1yzr20ifM4vPFCXLh1fze3yPSpPiVpobEdndMPUhR/Wp7b4iaNMwE63Fv/tOgI/Qk/pXl1FZe3meYs7xad7r7j3Sy1C01GHzbG4jnTuUbOPqO1WK8Ks7250+4WeymeGVejIcf/rr07wp4vj1tRa3YWK+UZwPuyj1Hv7V0U6ylo9z3sDm9PEy9nUXLL8GdPRRRW57YUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFQ3V3b2UJmu544Yx1aRgBWN4n8UweH4AiKJryQZSLPAH95vb+deW6jql5qt0Z7+dpX7AnhR6AdhWFSsoaLc8bHZrTwr5Irml+XqekXfxC0W3YrD59yR3jjwP/HiKqp8S9OLfPZXQHqNp/rXm1Fc/t5ngSzvFt3TS+R7Bp/jDRdRYJHdiKQ9EnGw/n0/WtyvA66Pw74xvNFkWGdmubLoY2PKD1U/06VpDEdJHoYXPby5cQrea/yPWaKhs7yC/s47m0kEkUgyrCpq6z6dNSV0FFFFAwooooAKKKKACiiigAooooAKKKKACiiigAopskiQxNJKwREBZmY4AA715n4l8cXGoSPa6U7W9oCQZBw8v+A9v/1VnOooLU4sZjaWEhzT36LudxqXijSNKYpdXimUdY4/nYfXHT8aw5PiVpqtiK0unHqQo/rXmtFcrxEnsfL1c8xMn7iSX3nqFt8RdHmYCdLm3/2mQMP0JP6V0dlqNnqUPm2NzHOvfY2SPqOorwyprS8uLG4WezmeGVejIcGnHESW5pQz2tF/vUmvuZ7tRXK+FPGKa1izvgsV6B8pHCy/T0PtXVV1xkpK6PqqGIp4imqlN3QUUUVRuFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFYfiXxPB4ftgMCW7kH7uLP/jx9v50m1FXZlVqwowc6jska9zdQWcJmu5o4Y16vIwArnLv4g6LbsVhM9yR3ijwPzbFeb6lqt7q1yZ7+dpW/hB+6o9AO1U65JYh/ZPlcRn1Ru1GNl57npK/EvTi3z2V0B6jaf61raf4y0XUWCJdiGQ9EnGz9en615BRUqvNbmFPPMVF+9Z/I98Bz0oryTw94vvdEkWKQtcWfQxMeUH+ye306V6nY31vqNnHdWcgkikGVI/kfeuqnUU9j6bBZhSxkfd0a3RYooorQ9AKKKKACvPviHrrNMuj2z4RQHuMdz1C/wBfxFegkhVJPAHJrwvUbtr/AFK4u3OTNIz89sngVz15WjbueDneIdKgqcftfkivRRRXCfFhRRRQAU+CeW2uEmgcxyxsGVl6gimUUAm07o9q0DVl1rRYbxcByNsij+Fx1/x/GtKvPvhneET3tkT8rKJlHoQcH+Y/KvQa9KnLmimfomAxDxGGjUe/X5BRRRWh3BRRRQAUUUUAFFFFABRRRQAUUUUAFVdTv4tL0ye9n+5Cm7Hqew/E4FWq4r4lXhi0u0tFOPPkLt7hR/iw/KonLli2cuMr/V8PKqui/HocBf30+pX0t3dPvllbJ9vQD2FV6KK83c/OJScm5PdhRRRSEFFFFAHXeAdday1MabO5+z3R+QE/ck7fn0/KvTq8FjkaKVJIztdGDKR2Ir3OyuReWFvcrwJolkH4jNduHldWPsMixDnTlRl9nb0ZPRRRXSfRBRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAHA/EPXWDJpFs5AwHuCD19F/r+VcDVvVbxtQ1a6u2OfOlZh7DPA/LFVK8ypLmlc/OcbiHia8qj26egUUUVBxhRRRQA+GaS3mSaFykkbBlZTyCO9ey+HdYXW9Ehu+BJ9yVR2cdfz6/jXi9dx8NLwre3lkT8rxiVR6EHB/9CH5VvQlaVu57eS4h0sR7PpL8+h6JRRRXefbhRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAFfUL2LTtPnvJz+7hQsff0H4nivFNR1CfVNQlu7pt0kjZ9lHYD2Feg/Ei8MOjW1qpx9oly3uFHT8yK81rixEry5T47PMQ5VlRW0fzf/ACiiiuY+eCiiigArq/Aeutp+qiwmf8A0a6bAB/hk7H8en5VylKjtG6uhKspyCOxqoycXdG+Hryw9WNSPQ97oqtp119t0y2uv+e0Sv8ATIzVmvU3P0qMlJJrqFFFFAyG8BNjOF6mNsflXhNe+V4hrFi2m6zdWjDHlSEL7r1B/LFcmIWzPl+IIO1OfTVfkUqKKK5D5UKKKKACiiigDrPhwCfEsuOgtmz/AN9LXqFcF8NLEhby/YYDYhQ+vdv/AGWu9r0KKtA+7yaDhg4363YUUUVseuFFFFABRRRQAUUUUAFFFFABRRRQAV558TQftenntsfH5ivQ64/4j2LXGiQ3aDJtpPm9lbj+YWsqyvBnmZrBzwc0vX7meZ0UUV5x8AFFFFABRRRQAV7T4bBHhjTt3X7Mn8q8atreS6uoreEbpJXCKPcnFe6W0C21rFBH92JAi/QDFdWHWrZ9Nw/B8859LJElFFFdh9YFFFFABRRRQAUUUUAFFFFABRRRQAUycE28gXqVOPyp9FAnqjwOir+u2Labrt3akYCSnZ7qeR+hFUK8pqzsfmE4OEnF7oKKKKRIUUUUAFdV8OwT4oOOgt3z+Yrla734aWLb7y/YYXAhQ+vdv/Za1pK80ejlkHPFwS73+47+iiivRP0EKKKKACiiigAooooAKKKKACiiigAooooA8/8AicD5mmntiTH/AI7XB16f8RLE3OgR3KDJtZct/utwf1xXmFefWVpnwmcwccZJvrZ/hb9AooorE8gKKKKACiinwQvcXEcMQ3SSMEUepJwKASbdkex+FwR4V07d18ha1qhs7dbOxgtk+7DGsY/AYqavVirKx+m0YuFOMX0SCiiimahXEfEDw+1zCNWtEy8K7Z1Hdezfh/L6V29BAIIIyD1BqZxUlZnNisPDE0nSl1PA6K73xL4Bbe93oSgg8vbZxj/d/wAPy9K4WaGW3laKeNo5FOGR1wR+FedKEoOzPgMThK2Fny1F8+jGUUUVByhViwsbjUr6K0tE3yyHAHp7n2qzpOg6hrUwSxt2Zc4aVuEX6n/JrtNFsR4J1XytSEcltfbUi1ALjyn/AOeTegJ5B/P22p0nN+R6uAy2pipXeke/+R1ek6bFpOlwWUHKxLgt/ePc/nVyiiu9KysfdxioRUY7IKKKKZQUUUUAFFFFABRRRQAUUUUAFFFFABUN3axXtnLbXC7opUKMPY1NRQJpNWZ4jrOkz6Lqclpcj7pyj9nXsRVGvaNd0C01+z8m6G2ReY5VHzIf6j2ry3WfDOo6JIftMJeDPyzxjKn6+n0NefUpOLutj4XMMsqYaTlBXh+XqZNFFFYnkBRSojSOEjUszHACjJNdn4c8BT3Mi3OtK0EA5EHR3+v90fr9KuMXJ2R04fC1cTPlpq/5Il+H3h9pLj+17pMRpkW4P8TdC30HT6/SvQ6bHGkMSxxIERAFVVGAB6U6vQhBQjY+9weFjhaKpx+fmwoooqzsCiiigAooooAKKKKACiiigAooooAKKKKAOL8f+H2vLddUtE3SwLtmUdWT1/D+X0rzevfDz1rgvE3gIvI95oSjnl7bOPxX/D8vSuWtSbfNE+YzbLJTk69FX7r9UcBRT54JbaZoriN4pFOGR1II/CmVxnyjTTswoorQ0rQ9Q1mYJYW7OucNIeEX6mmk27IqEJVJcsFdlaysp9QvYrW1TfLK21R/X6V7Po2mRaPpMFlDz5a/M395j1P51Q8N+F7bw/CWB867cYkmI6ey+g/nW7XdRp8iu9z7bKsveFi51Pif4IKKKK3PaCiiigAooooAKKKKACiiigAooooAKKKKAIrm3ju7WW3nXdHKpRh6g14xrejz6JqklpOCQDmN8cOvY17ZXI+JIl8U6gmiaeiF7Zw91ekZFsP7g9Xb07D9MatPnWm55WZYBYyCcdJLb/I8xorY1rwxqWiSMbiIyQZ+WeMZU/X0/GseuBpp2Z8NUpTpS5ZqzCiilVWdgqKWYnAAGSaRmJXbfD/w+090NXuUxDESIAf4m6Z+g/n9Ki8O+Ari8dbjWVa3t+oh6O/1/uj9a9Ihhjt4UhgRY40AVVUYAFdVGk780j6XKsrm5qvWVktkPooorsPrQooooAKKKKACq15ptlqCbb61inA6eYgJH0ParNFG5MoqStJXRz0ngXQJGyLNk9llf/Gp7bwfoVqwaPT43Yd5SX/Qkitqio5I9jnWDwyd1TX3IREWNAkahFUYCqMAVFeWdvqFlLaXkSywTLtdGHBFTUVZ1HM6VeXOg6lHoWsStLBIcadeuf8AWD/nk5/vjt6j3rpqqanplrq+nyWd9Hvif0OCp7MD2I9ayNG1W5stR/sHXn3XYUm1uiMC7jH/ALOO4/GgDoqKKKACiiigAooooAKKKKACiiigAooooAKKKKACggMCCMg8EHvRRQBj3fhTQ71i02nRBj1MeU/9BxVVPAmgK2TaM3sZn/xroqKjki+hyyweHk7umvuRytjYWmg+Ozb20CRQajZ7ou5EkRwwH1Vwfwrqq57xePstnZawvDaZdpK5A58pvkkH/fLZ/CuhBz0q9jojGMVaKsFFFFBQUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAHMeIraDWPE2jaVLEkiIZLy4yOQiDao+hZx+VTSeBtAkbIs2T/dlb/GjQv9P8Sa1qp5VJFsICR0WMZfHsXY/lXQ1LjF7oxqYejV/iRT9UYlt4P0G1YMmnxuw/56sX/QnFbKRpFGEjVURRgKowBTqKailsOnRp0tKcUvRBRRRTNQooooAKKKKACiiigAooooAKKKKACiiigAoorD17W5raeLStHRZtWuhmNTysCd5H9h2Hc0AR67qt1NeroegsP7QlXdNPjK2cZ/iP8AtHsPxrT0nSrbRtOSzs1O1eWdjlpGPVmPcmo9F0aHRrMxxs008reZcXEnLzOerH+g7Vo0ABAIIIyD1BrIu/CuiXrFp9OiDHqY8p/6Diteik0nuZ1KVOorTin6nOr4E0BWybR29jM/+Natjo+nab/x42cMLf3lX5j+PWrtFJRitkZ08NQpu8IJP0QUUUVR0BRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABWfrWjW+t2PkXBaORGDwzpw8Ljoyn1rQooAw9E1m4e6fSNbCxarAu7cowlynaRP6jsa3Kydf0NdZtUMUptr62bzLW6XrE/9QehHem6Drb6islnqMYttVtMC5g7H0dPVT+nSgDYooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKAIL+zj1DTrizn/1dxE0TY9GGP61m+EruS78M2ouf+Pm3BtpxnkPGdhz9cZ/Gtmue0r/iX+MdX07pHdqmoQgep+ST/wAeVT/wKgDoaKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAqnq+oJpWjXd/JjFvC0mD3IHA/E8VcrnvFX+mS6Vow5F9dhpVPeKL9436hR+NAFzwzp76X4asrabPnCPfMT18xjuf/AMeJrVoooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiis3W9ah0WzV2Rp7mZvLtraP78z9gPb1PYUAR67rf9mLFbWcX2rUrolba2B+8e7N6KOpNGhaGNKjlnupftWpXR33VyR98/3R6KOgFQ+H9Ems5JdT1eRZ9Wux+9cfdiXtGnoo/U81uUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAVja/or6gsV7psgttVtMm3n7MO8b+qn9OtbNFAGVoGuJrVm/mRG2vbdvLurV/vRP/UHqD3rVrC1zRrhrpNY0MrHqsC4Kk4S6j7xv/Q9j+l3RdZt9c0/7Rbho3Vik0DjDwuOqsPWgDQooooAKKKKACiiigAooooAKKKKACiiigAooooAK5/xH/oWraNqw4WG4NrMe3lyjbk/RwldBWdr+nf2t4fvbEffmiIjPo45U/gwBoA0aKz9B1H+1tAsr7+KaFWcejdGH4EEVoUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABXP2X/Ew8c6hddYtOt0tI/Qu/zufqBsFblxPHa20s8x2xxIXc+gAyaxvB8Dp4djurhcXGoSPey/WQ7h+S7R+FAG7RRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUVV1LUrXSdPlvb+URwRDLHufQAdyfSgCPWNXttE017y7JIBCpGgy0rnoijuTWboOk3Ul02ua8oOpTLiOLOVtI/wC4vv6mo9I0651bUE17XozHIAfsNk3S2Q/xN/00Pf06fTpKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigArnNY0u60/Ujr+gR77naBeWYOBdxj0/wBsdj+FdHRQBU0vVLXWNPjvbGTfFIO4wVPdSOxHpVuuX1azuPD2pya9pETS20pzqVkg++P+eqD++O47j35rorO8t9Qs4ruylWaCZdyOvQigCaiiigAooooAKKKKACiiigAooooAKKKKACiiigDn/Df+haprOkHhYLn7TCP+mco3YH0beK6Cuf1P/iX+MtKv+kd4j2Ep9/vx/qGH410FABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQBgeMXaXRo9NiJEmp3CWgI6hWOXP4IGreRFjjVEAVVGAB2FYEn/ABMfH0SdYtJtTIfaWU4H/jit/wB9V0FABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRTZJEijaSVlREBZmY4CgdSTQAy6uoLK1kubuVYoYlLO7HAUVzdhaTeKdRh1nVYmi06Bt+n2bjlj2mcev8AdHYfrHAj+NNRS7nUroFq+beJhj7a4/5aMP7g7Dv1PpXW0AFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFcpeQv4OvpNSsUZ9FuH3Xtsgz9mY/8tkH93+8Px+nV0jKHUqwDKRggjgigBsMsdxCk0DrJHIoZHU5DA8gg0+uT+fwRefxP4duJPr9gdj/6LJP4E/n1asGUMpBBGQR3oAWiiigAooooAKKKKACiiigAooooAKKKKAMXxbaSXXhq5a3Gbi123UGOu+Mhx+eMfjWnZXcd/YQXcBzHPGsi/QjNTkAggjIPUGuf8IE21jd6Q5+bTLp4VB6+WfnjP/fLAfhQB0FFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFBOBk8CisbxZdyWnhm6FucXFwBbQf78h2D+efwoAg8Ij7VZ3mrsPm1O6eZCf8AnmvyRj/vlc/jXQVBY2ken6fb2cAxHbxLGv0AxU9ABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABXJu7eNr54ImZfD9s+JZFOPt0gP3Qf8AnmD1Pc9PUOv7ifxXfy6RpkjRaXA2zULxDgynvCh/9CPbp9eltraGztY7e1jWKGJQqIowFA7UASIixxqkahEUAKqjAA9AKWiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAZNDHcQvDOiyRSKVdGGQwPUEVy9nNJ4Ovo9MvnZ9FuH22V05ybZj/yxc/3f7pP0+nV1BeWdvqFnLaXsSzQTLtdG6Ef570AT0VzGlXlx4f1KPQdYlaW3lONNvZD/rB/zxc/3x2P8Q9+K6egAooooAKKKKACiiigAooooAKKKKACuff/AIl3j6NukWrWpQ/9dYjkfmjH/vmugrA8YoYtIi1OMZk0y4S6+qA4cf8AfJagDfopEdZI1dCGVhkEdxS0AFFFFABRRRQAUUUUAFFFFABRRRQAVz+q/wDEw8YaTp45jtFe/mHuPkj/AFZj+FdBXPeHP9O1bWdXPKzXH2WE/wDTOIbcj6sWoA6GiiigAooooAKKKKACiiigAooooAKKKKACua1fULnWtQk0DQpWi2/8hC+T/l3U/wACn/nof0+vSTXNVup70aFoDD+0JV3T3GMrZRn+I+rH+FfxPHXU0nSbXRdOSzsUIReWZjlpGPVmPcn1oAk0+wttL0+GysYhFBCu1EH8/c981ZoooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAqappdrrGnSWV/HvikHbgqezA9iPWsjRdTurHUBoGvyb7sKWtLs8C8jH/s47j8feuirP1rR7fW9PNtcFo3VhJDPGcPDIPuup7EUAaFFYWhazcPcyaPrgWPVbdd25RhbqPtIn9R2P6btABRRRQAUUUUAFFFFABRRRQAVHc26XVrLbzDMcyFHHqCMGpKKAMPwfcSSeHIre4JNxYu1nLn1jO0H8Rg/jW5XPWP/ABLvHOo2nAi1CBLyMf7a/I//ALKa6GgAooooAKKKKACiiigAooooAKKKKAM/XtR/snQL2+HLQxEoB3boo/Mik0DT/wCyvD9lZNy8UQDk93PLH8yaz/En+napo2kDBWe5+0zD/pnF82D9W2iuhoAKKKKACiiigAooooAKKKKACiiigArD17WZ7eaLStGRZtWulJjVvuwJ0Mr+ijt6ngZqXXtbOlpFbWUX2rU7slLW2B+8e7N6KOpNLoOif2TDLNcy/atRumD3d0w5kbsB6KOgHYUASaJosGiWJijdpp5W8y4uZPvzyHqx/wAO1aVFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAZeu6Ims2sZSU217bN5lpdoPmhf+qnoV6EfhUeg62+oebZajEttqtpgXEAPyt6SJ6of06H1OxWPruiPqHlXunSra6raZNtORwfWN/VD+nUehANiisvQtaTWLaQSRG2vrZvLu7Rz80L/1U9Qw4I/GtSgAooooAKKKKACiiigAooooA57xT/oVxpWsrwLO6Ecxx0il+RvyJU/hXQ1S1nT11XRLyxf/AJeIWQH0OOD+BxUHhrUG1Tw3ZXUn+tMYSUE9HX5W/UGgDUooooAKKKKACiiigAooooAKKKgvruOw0+4u5jiOCNpG57AZoAxdL/4mHjPVb88x2aJYQnHf78n6lR+FdDWJ4RtJLXwzbNcDFxdbrqbP9+Q7j/MD8K26ACiiigAooooAKKKKACiiigArN1vWotFs1cxtcXMzeXbWsZ+ed+wHoPU9hUmr6tbaLpzXd2WIyEjjQZeVz91FHdj/AJ4FZ+iaRctdtrevBTqUy7Y4QcpZx/8APNfVv7zdzwOOoBJoOiy2by6lq0i3Gr3YHnSqPliXtFH6KP1PJ7VtUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAGHrmj3ElzHq+iFY9Vt124Y4W5j6mJ/6Hsau6NrFvrVgLi3DRurGOaGQYeGQfeRh2Iq/XP6xplzZ6gde0JN12FC3dqDhbyMdv98fwt+HSgDoKKqaZqdtq+nx3lk++KQdCMFT3UjsR6VboAKKKKACiiigAooooAK53QP9A8Ra3pJ4UyrewDH8Mg+bH0dT+ddFXO63/wAS/wAVaLqY4SZmsJzns/zJ/wCPL+tAHRUUUUAFFFFABRRRQAUUUUAFc94xY3NjZ6Qhw+p3SQsB18sHc5/75XH410Nc6n/Ex+IMjdYtJtAg56Sy8n/xxR+dAHRABVAUYA4AHaiiigAooooAKKKKACiiigAqtqOo2ulafLe38oighXLMf5D1J6YqS6uobK1kubqVYoYlLO7HAUCue0+1m8SahDrWqRNHZQNv06zcd+07j+8f4R/CPfmgCTSdOutT1Bdd12IxygEWVm3/AC6oe5H/AD0Pf06V0VFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAHN6pZ3OhahJrmjRNNDIc6jYp1lH/PVB/wA9B3H8Q9+a3rO8t9Qs4ruylWaCZdyOvQj/AD2qauYvIpPCd5LqdmjPpEzb762QZNux6zIPT+8B9fWgDp6KZDNHcQJNBIskUihkdDkMD0INPoAKKKKACiiigArH8V2D6h4YvIof9fGnnQn0dDuX9RitiigCppV+mqaTa30WNtxEsmB2yOlW653wj/oa6lozZH9nXbCMH/nlJ86fzI/CuioAKKKKACiiigAooooAR3WONnchVUEsT2Fc/wCDEabSJtUlGJNUuXuvohOEH/fIH51J4yuJIvDU1tbki4v3Wzhx/ekO0/pk/hWxa28dpZw20IxHDGsaD0AGBQBLRRRQAUUUUAFFFFABTZJEijaSVlREBZmY4CgdSTTunWuWZz4zuzHESPD9u+Hccfb3B+6P+mYPU/xHgdKAC3jfxhfJeXKFNCt33WsDDBvXHSVx/cH8K9+p9K6mkVQihUAVVGAAMACloAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACggEYIyDRRQBy+0+DbolcnQLh8kf8+Dseo/6Zkn/gJPoa6cEMoZSCCMgjvSSRpNE8cqK8bqVZWGQwPUEVzNo0nhG9SwupGfRJ322k7nJtGPSJz/cP8LHp0PY0AdRRRRQAUUUUAFFFFAHOXX/Et+IFnPwItUtmt35/5aR/OpP1UsK6OsDxpC//AAj5vrdcz6bMl5GAOuw/MP8AvktW5DMlxbxzQtujkUOp9QRkUAPooooAKKKKACiiigDnNQ/4mPjvTbPgxadA97IM/wAbfIg+o+Y10dc54V/0661fWjyL26McJx1ii+RSPqdxro6ACiiigAooooAKKK5zVry51q+fQ9GmaFE41C9TrCp/5Zof+ehHf+Ee9ADL2aTxTey6VYyMmlQtsv7lDgzHvCh/9CI+ldHBBFbW8cFvGsUUahURBgKB0AFR2Vlb6dYxWdlEsMEK7URew/z371PQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAVFdWsF7ayW13EssMqlXRhwwqWigDmtPvJ/D+pRaLqsjS2kx26deufvf9MXP98dj/EPeulqtqOnW2q2ElnfR+ZDIMEdCD2IPYjqDWTpWpXGn6guh65JvnIJs7tuBdoOx/wCmg7jv1oA36KKKACiiigBk0STwPDKu6ORSrA9wRg1heC5XHh/7BOxabTJns3J7hD8p/wC+StdBXOW3/Et+IN3CeItVtVnT/rrH8rAfVSp/CgDo6KKKACiiigArK8Tai2leGr67iz5qxFYsdd7fKv6kVq1zniH/AImHiLRNJHK+cb6ceixD5QfYuw/KgDV0TTl0nQ7OwXH+jwqhI7tjk/icmr1FFABRRRQAUUVi61rE0VzHpWjqsuqXC7hu5W3ToZX9h2Hc8UARazql1c6gNC0JsXrqGubnGVs4z/EfVz/Cv4nitTTNNttJsEtLNCsackscs7HqzHuSepqPR9Ig0ezMMJaSSRjJPPJy8znqzH1/lV+gAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACqGsaRba3p7Wt1uXDB4pUOHicfddT2Iq/RQBiaLqtx9obSNb2rqUK5WQDC3Uf/PRff1HY+1bdZut6PHrFmqiRre6hbzLa5T70L+o9R2I7ioNB1iW982x1SNbfVbTAniH3XHaRPVT+nQ0AbNFFFABXOeL/APQ49O1peP7Nu1aVvSF/kf8ARgfwro6q6nYpqek3VjL924iaMn0yMZoAtUVj+E757/wvZST/AOvjTyJgTyHQ7Gz+Iz+NbFABRRRQAVzmif8AEw8Wa1qh5SBl0+A+mz5pP/H2/StnVL5NM0m6vpfu28LSEE9cDOPx6VR8J2L6f4Xsop8meRPOmJ6l3O5s/icfhQBsUUUUAFFFZut6ymj2ilYmuLudvLtrZD80z+nsB1J7CgBmt6w2nrHa2MQudSusrbW+ePd29FHc/hS6Foi6RBK8spub66bzLq6Ycyt/RR0A7Co9C0eWxEl7qci3Gq3WDcTAfKo7Rp6KP16mtigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKytb0Y6isVzZS/ZdStctbXGOnqjDup7itWigDK0PW11aGWKeI21/atsurZjzG3qPVT1B71q1i63o801xHqujssWqW64UtwtwneN/Y9j2NWtH1iDWbMyxK0UsbGOe3k4eFx1Vh/XvQBoUUUUAc5ov/Eu8W61ph4S4K6hAPXf8sn/AI8v610dc74g/wBA8QaJqwyF842U+O6Sj5SfYOq/nXRUAFFFFAHOeL/9Mj03RRz/AGldqJV9YY/nf/0ED8a6Oudtv+Jl8QLuY5MWlWq26Z6ebJ87Ef8AAQo/GuioAKKKq6lqVtpNhJeX0myKMemSx7ADuT6UARaxrFvotgbm53OzMEihQZeZz0VR3Jqnouk3AuW1fWir6nMu0Ipytqn/ADzX+p7motJ025v9QXXNcj2XGCLS0PItEPr/ALZ7nt0roKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACsDWtLuoL4a5oS5vo123FvnC3cY/hP+0P4T+Fb9FAFPS9TttX0+O7s2yjcFWGGRh1Vh2I9KuVzmrWVzo18+u6LEZA3N/ZJ/y3UfxqP74/UVt2F/banYxXllKJYJV3Iw/z1oAp+JdObVfDV9aR5814i0WOokX5l/8AHgKm0XUV1fQ7O/XH+kQq7AdmxyPwORV6ud8Lf6Fdavox4FndmWEY4EUvzqB9DvH4UAdFTJZUgheWZgkcalmY9gOSafWB4zlc+HzYwMVm1OZLJCB03n5v/HA1AB4Mic+H/t867ZtTme9cem8/KP8AvkLW/TIYkggSGJQscahVUdgBgCiWWOCF5ZnWONFLMzHAUDvQAy7u4LG0luruVYoIlLO7HgCsDTLS41/Uo9c1aNoraI506zcfc/6auP7x7DsPeo7OOTxdepqN4jJo0D7rO3YY+0sP+Wrj+7/dH411NABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFcxe28vha9l1XTo2k0yZt99aIMmI95kH/oQ/Gunrh/iB4ubS4jpWnPi7lXMsg/5ZKew9z+g+tbUaMq01CJjXrQoQc5lvXPiHpWmRAWDDUJ3UMoib5ACOCW/oOfpXnN/4v1e91aTUI7j7JLJEISLf5QUBJA9c8nmsQnJyeTSV9LRwNGktVd+Z8tXzCvVejsuyNA69rBbcdVvS3r9of8AxqxD4p1iO9tLmW8kuWtHLxLcHeASpU5zz0NY9FdMqNOStKKOWNerF3jJnrfh/wCI9jqTrb6qq2Nw3Afd+7Y/X+H8fzq383jS64yvh+B/p9vcH/0WD/30a8Zr0b4e+L2Eseiak5ZW4tZG7H+4f6fl6V42My9Ri6lL7j3MFmTnJU63yf8AmekKoVQqgAAYAA6UtFFeIe6FFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQBBfXcdhYT3c3+rgjaRvoBmvn+/vZtR1Ce8uW3SzOXY/XtXsXxAnMHgm92nBkKJn2LjP6V4rX0GVU0oSn52PnM3qN1I0+iVwooor2TwwooooAKcjtHIrxsVZSCrA8g02igD3zw7qn9teH7S+ON8iYkA7OOG/UGtOuJ+Fk5k8N3ETHPl3Jx7Aqv8AXNdtXxuJpqnWlFdz7fDVHUoxm+qCiiisDoCiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigDnPH1ubjwVfBRlowsn4Bhn9M14nX0Vc28d3aTW043RzIUceoIwa8A1TTptJ1S4sbkYkhcrnH3h2P0Iwa9/Kqi5JU/mfOZxTanGp02KlFFFe0eGFFFFABRRUkEMlzcRwQIXlkYKijqxPAFGw9z1b4XW5i8MTTMMedcsV9wFA/nmu1qhoemLo+h2lgmD5MYDEd2PLH8yav18biKiqVZTXVn2+GpulRjB9EFFFFYG4UUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFch468InXbUXlgo+3wrjHTzl/u/X0/KuvorWlVlSmpx3Mq1KFaDhPZnzlJG8MjRyoyOpwysMEH0Ipte4+IPB+meIR5lwhhusYFxFw34joa4DUPhlrVqxNk0N6nba2xvybj9a+ko5hRqL3nZ+Z8vXy2vSfurmXl/kcbRW6fBXiINt/sqbPsRj881o2Pw2166YfaUhs07mWQMfyXP8ASumWJoxV3JfecscLXk7KD+45IDJwOTXp/gDwa9iV1fVYts5H+jwsOYwf4j7+3b+Wx4e8CaZoTrcPm8u15EsgwFP+yvb68munrxcZmCqRdOlt3PcwWWunJVKu/YKKKK8c9sKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigD//Z)

**Instructions**

**100 XP**

Import the Sequential model and Dense layer from Keras.

Create a sequential model.

Add a 4 neuron input layer with the input\_shape parameter and a 1 neuron output layer with sigmoid activation.

Compile your model using sgd as an optimizer.

# Import the sequential model and dense layer

from keras.models import Sequential

from keras.layers import Dense

# Create a sequential model

model = Sequential()

# Add a dense layer

model.add(Dense(1, input\_shape=(4,), activation='sigmoid'))

# Compile your model

model.compile(loss='binary\_crossentropy', optimizer='sgd', metrics=['accuracy'])

# Display a summary of your model

model.summary()

<script.py> output:

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_1 (Dense) (None, 1) 5

=================================================================

Total params: 5

Trainable params: 5

Non-trainable params: 0

**Is this dollar bill fake ?**

You are now ready to train your model and check how well it performs when classifying new bills! The dataset has already been partitioned into features: X\_train & X\_test, and labels: y\_train & y\_test.

**Instructions**

**100 XP**

Train your model for 20 epochs calling .fit(), passing in the training data.

Check your model accuracy using the .evaluate() method on the test data.

Print accuracy.

# Train your model for 20 epochs

model.fit(X\_train, y\_train, epochs = 20)

# Evaluate your model accuracy on the test set

accuracy = model.evaluate(X\_test, y\_test)[1]

# Print accuracy

print('Accuracy:', accuracy)

**A multi-class model**

You're going to build a model that predicts who threw which dart only based on where that dart landed! (That is the dart's x and y coordinates on the board.)

This problem is a multi-class classification problem since each dart can only be thrown by one of 4 competitors. So classes/labels are mutually exclusive, and therefore we can build a neuron with as many output as competitors and use the softmax activation function to achieve a total sum of probabilities of 1 over all competitors.

Keras Sequential model and Dense layer are already loaded for you to use.

**Instructions**

**70 XP**

**Instructions**

**70 XP**

Instantiate a Sequential model.

Add 3 dense layers of 128, 64 and 32 neurons each.

Add a final dense layer with as many neurons as competitors.

Compile your model using categorical\_crossentropy loss.

# Instantiate a sequential model

model = Sequential()

# Add 3 dense layers of 128, 64 and 32 neurons each

model.add(Dense(128, input\_shape=(2,), activation='relu'))

model.add(Dense(64, activation='relu'))

model.add(Dense(32, activation='relu'))

# Add a dense layer with as many neurons as competitors

model.add(Dense(4, activation='softmax'))

# Compile your model using categorical\_crossentropy loss

model.compile(loss='categorical\_crossentropy',

              optimizer='adam',

              metrics=['accuracy'])

**Prepare your dataset**

In the console you can check that your labels, darts.competitor are not yet in a format to be understood by your network. They contain the names of the competitors as strings. You will first turn these competitors into unique numbers,then use the to\_categorical() function from keras.utils to turn these numbers into their one-hot encoded representation.

This is useful for multi-class classification problems, since there are as many output neurons as classes and for every observation in our dataset we just want one of the neurons to be activated.

The dart's dataset is loaded as darts. Pandas is imported as pd. Let's prepare this dataset!

**Instructions 1/2**

**50 XP**

[1](javascript:void(0))

[2](javascript:void(0))

Use the Categorical() method from pandas to transform the competitor column.

Assign a number to each competitor using the cat.codes attribute from the competitor column.

# Transform into a categorical variable

darts.competitor = pd.Categorical(darts.competitor)

# Assign a number to each category (label encoding)

darts.competitor = darts.competitor.cat.codes

# Print the label encoded competitors

print('Label encoded competitors: \n',darts.competitor.head())

Import to\_categorical from keras.utils.

Apply to\_categorical() to your labels.

# Transform into a categorical variable

darts.competitor = pd.Categorical(darts.competitor)

# Assign a number to each category (label encoding)

darts.competitor = darts.competitor.cat.codes

# Import to\_categorical from keras utils module

from keras.utils import to\_categorical

coordinates = darts.drop(['competitor'], axis=1)

# Use to\_categorical on your labels

competitors = to\_categorical(darts.competitor)

# Now print the one-hot encoded labels

print('One-hot encoded competitors: \n',competitors)

# Training on dart throwers

Your model is now ready, just as your dataset. It's time to train!

The coordinates features and competitors labels you just transformed have been partitioned into coord\_train,coord\_test and competitors\_train,competitors\_test.

Your model is also loaded. Feel free to visualize your training data or model.summary() in the console.

Let's find out who threw which dart just by looking at the board!

##### Instructions

**100 XP**

Train your model on the training data for 200 epochs.

Evaluate your model accuracy on the test data.

# Fit your model to the training data for 200 epochs

model.fit(coord\_train,competitors\_train,epochs=200)

# Evaluate your model accuracy on the test data

accuracy = model.evaluate(coord\_test, competitors\_test)[1]

# Print accuracy

print('Accuracy:', accuracy)

32/640 [>.............................] - ETA: 0s - loss: 0.3403 - acc: 0.8125

544/640 [========================>.....] - ETA: 0s - loss: 0.5323 - acc: 0.7960

640/640 [==============================] - 0s 102us/step - loss: 0.5114 - acc: 0.8016

32/160 [=====>........................] - ETA: 0s

160/160 [==============================] - 0s 363us/step

Accuracy: 0.84375

# Softmax predictions

Your recently trained model is loaded for you. This model is generalizing well!, that's why you got a high accuracy on the test set.

Since you used the softmax activation function, for every input of 2 coordinates provided to your model there's an output vector of 4 numbers. Each of these numbers encodes the probability of a given dart being thrown by one of the 4 possible competitors.

When computing accuracy with the model's .evaluate() method, your model takes the class with the highest probability as the prediction. np.argmax() can help you do this since it returns the index with the highest value in an array.

Use the collection of test throws stored in coords\_small\_test and np.argmax()to check this out!

##### Instructions 1/2

**50 XP**

[1](javascript:void(0))

[2](javascript:void(0))

Predict with your model on coords\_small\_test.

Print the model predictions.

# Predict on coords\_small\_test

preds = model.predict(coords\_small\_test)

# Print preds vs true values

print("{:45} | {}".format('Raw Model Predictions','True labels'))

for i,pred in enumerate(preds):

  print("{} | {}".format(pred,competitors\_small\_test[i]))

Raw Model Predictions | True labels

[0.34438723 0.00842557 0.63167274 0.01551455] | [0. 0. 1. 0.]

[0.0989717 0.00530467 0.07537904 0.8203446 ] | [0. 0. 0. 1.]

[0.33512568 0.00785374 0.28132284 0.37569773] | [0. 0. 0. 1.]

[0.8547263 0.01328656 0.11279515 0.01919206] | [1. 0. 0. 0.]

[0.3540977 0.00867271 0.6223853 0.01484426] | [0. 0. 1. 0.]

# Predict on coords\_small\_test

preds = model.predict(coords\_small\_test)

# Print preds vs true values

print("{:45} | {}".format('Raw Model Predictions','True labels'))

for i,pred in enumerate(preds):

  print("{} | {}".format(pred,competitors\_small\_test[i]))

# Extract the position of highest probability from each pred vector

preds\_chosen = [np.argmax(pred) for pred in preds]

# Print preds vs true values

print("{:10} | {}".format('Rounded Model Predictions','True labels'))

for i,pred in enumerate(preds\_chosen):

  print("{:25} | {}".format(pred,competitors\_small\_test[i]))

<script.py> output:

Raw Model Predictions | True labels

[0.34438723 0.00842557 0.63167274 0.01551455] | [0. 0. 1. 0.]

[0.0989717 0.00530467 0.07537904 0.8203446 ] | [0. 0. 0. 1.]

[0.33512568 0.00785374 0.28132284 0.37569773] | [0. 0. 0. 1.]

[0.8547263 0.01328656 0.11279515 0.01919206] | [1. 0. 0. 0.]

[0.3540977 0.00867271 0.6223853 0.01484426] | [0. 0. 1. 0.]

Rounded Model Predictions | True labels

2 | [0. 0. 1. 0.]

3 | [0. 0. 0. 1.]

3 | [0. 0. 0. 1.]

0 | [1. 0. 0. 0.]

2 | [0. 0. 1. 0.]

# An irrigation machine

You're going to automate the watering of farm parcels by making an intelligent irrigation machine. Multi-label classification problems differ from multi-class problems in that each observation can be labeled with zero or more classes. So classes/labels are not mutually exclusive, you could water all, none or any combination of farm parcels based on the inputs.

To account for this behavior what we do is have an output layer with as many neurons as classes but this time, unlike in multi-class problems, each output neuron has a sigmoid activation function. This makes each neuron in the output layer able to output a number between 0 and 1 independently.

Keras Sequential() model and Dense() layers are preloaded. It's time to build an intelligent irrigation machine!

Diagram

Description automatically generated

##### Instructions

**100 XP**

##### Instructions

**100 XP**

Instantiate a Sequential() model.

Add a hidden layer of 64 neurons with as many input neurons as there are sensors and relu activation.

Add an output layer with as many neurons as parcels and sigmoidactivation.

Compile your model with the adam optimizer and binary\_crossentropy loss.

# Instantiate a Sequential model

model = Sequential()

# Add a hidden layer of 64 neurons and a 20 neuron's input

model.add(Dense(64, input\_shape=(20,), activation='relu'))

# Add an output layer of 3 neurons with sigmoid activation

model.add(Dense(3, activation='sigmoid'))

# Compile your model with binary crossentropy loss

model.compile(optimizer = 'adam',

              loss = 'binary\_crossentropy',

              metrics = ['accuracy'])

model.summary()

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_1 (Dense) (None, 64) 1344

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 3) 195

=================================================================

Total params: 1,539

Trainable params: 1,539

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Training with multiple labels

An output of your multi-label model could look like this: [0.76 , 0.99 , 0.66 ]. If we round up probabilities higher than 0.5, this observation will be classified as containing all 3 possible labels [1,1,1]. For this particular problem, this would mean watering all 3 parcels in your farm is the right thing to do, according to the network, given the input sensor measurements.

You will now train and predict with the model you just built. sensors\_train, parcels\_train, sensors\_test and parcels\_test are already loaded for you to use.

Let's see how well your intelligent machine performs!

##### Instructions

**100 XP**

##### Instructions

**100 XP**

Train the model for 100 epochs using a validation\_split of 0.2.

Predict with your model using the test data.

Round up your preds with np.round().

Evaluate your model's accuracy on the test data.

# Train for 100 epochs using a validation split of 0.2

model.fit(sensors\_train, parcels\_train, epochs = 100, validation\_split = 0.2)

# Predict on sensors\_test and round up the predictions

preds = model.predict(sensors\_test)

preds\_rounded = np.round(preds)

# Print rounded preds

print('Rounded Predictions: \n', preds\_rounded)

# Evaluate your model's accuracy on the test data

accuracy = model.evaluate(sensors\_test, parcels\_test)[1]

# Print accuracy

print('Accuracy:', accuracy)

32/600 [>.............................] - ETA: 0s

600/600 [==============================] - 0s 21us/step

Accuracy: 0.9061111267407735

# The history callback

The history callback is returned by default every time you train a model with the .fit() method. To access these metrics you can access the history dictionary parameter inside the returned h\_callback object with the corresponding keys.

The irrigation machine model you built in the previous lesson is loaded for you to train, along with its features and labels now loaded as X\_train, y\_train, X\_test, y\_test. This time you will store the model's historycallback and use the validation\_data parameter as it trains.

You will plot the results stored in history with plot\_accuracy() and plot\_loss(), two simple matplotlib functions. You can check their code in the console by pasting show\_code(plot\_loss).

Let's see the behind the scenes of our training!

##### Instructions

**100 XP**

##### Instructions

**100 XP**

Train your model on X\_train and y\_train, validate each epoch on X\_test and y\_test.

Use plot\_lossextracting lossand val\_loss from h\_callback.

Use plot\_accuracyextracting accand val\_acc from h\_callback.

# Train your model and save its history

h\_callback = model.fit(X\_train, y\_train, epochs = 50,

validation\_data=(X\_test, y\_test))

# Plot train vs test loss during training

plot\_loss(h\_callback.history['loss'], h\_callback.history['val\_loss'])

# Plot train vs test accuracy during training

plot\_accuracy(h\_callback.history['acc'], h\_callback.history['val\_acc'])

Chart

Description automatically generated

# Early stopping your model

The early stopping callback is useful since it allows for you to stop the model training if it no longer improves after a given number of epochs. To make use of this functionality you need to pass the callback inside a list to the model's callback parameter in the .fit() method.

The model you built to detect fake dollar bills is loaded for you to train, this time with early stopping. X\_train, y\_train, X\_test and y\_test are also available for you to use.

##### Instructions

**100 XP**

Import the EarlyStoppingcallback from keras.callbacks.

Define a callback, monitor 'val\_acc' with a patience of 5 epochs.

Train your model using the early stopping callback.

# Import the early stopping callback

from keras.callbacks import EarlyStopping

# Define a callback to monitor val\_acc

monitor\_val\_acc = EarlyStopping(monitor='val\_acc',

                       patience=5)

# Train your model using the early stopping callback

model.fit(X\_train, y\_train,

           epochs=1000, validation\_data=(X\_test,y\_test),

           callbacks= [monitor\_val\_acc])

960/960 [==============================] - 0s 54us/step - loss: 0.2111 - acc: 0.9531 - val\_loss: 0.2136 - val\_acc: 0.9417

Epoch 26/1000

32/960 [>.............................] - ETA: 0s - loss: 0.2096 - acc: 0.9688

960/960 [==============================] - 0s 53us/step - loss: 0.2091 - acc: 0.9531 - val\_loss: 0.2116 - val\_acc: 0.9417

# A combination of callbacks

Deep learning models can take a long time to train, especially when you move to deeper architectures and bigger datasets. Saving your model every time it improves as well as stopping it when it no longer does allows you to worry less about choosing the number of epochs to train for. You can also restore a saved model anytime and resume training where you left it.

The model training and validation data are available in your workspace as X\_train, X\_test, y\_train, and y\_test.

Use the EarlyStopping() and the ModelCheckpoint() callbacks so that you can go eat a jar of cookies while you leave your computer to work!

##### Instructions

**100 XP**

##### Instructions

**100 XP**

Import both the EarlyStopping and ModelCheckpoint callbacks from keras.

Create monitor\_val\_acc as an EarlyStopping callback that will monitor 'val\_acc', with a patience of 3 epochs.

Create modelCheckpoint as a ModelCheckpointcallback, save the best model as best\_banknote\_model.hdf5.

Fit your model providing a **list** with the defined callbacks and X\_test and y\_test as validation data.

# Import the EarlyStopping and ModelCheckpoint callbacks

from keras.callbacks import EarlyStopping , ModelCheckpoint

# Early stop on validation accuracy

monitor\_val\_acc = EarlyStopping(monitor = 'val\_acc', patience = 3 )

# Save the best model as best\_banknote\_model.hdf5

modelCheckpoint = ModelCheckpoint('best\_banknote\_model.hdf5', save\_best\_only = True)

# Fit your model for a stupid amount of epochs

h\_callback = model.fit(X\_train, y\_train,

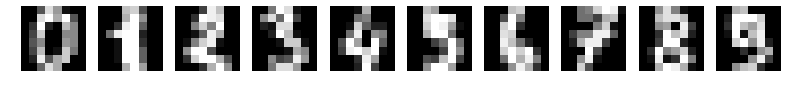
                    epochs = 1000000000000,

                    callbacks = [monitor\_val\_acc, modelCheckpoint],

                    validation\_data = (X\_test, y\_test))

**Learning the digits**

You're going to build a model on the **digits dataset**, a sample dataset that comes pre-loaded with scikit learn. The **digits dataset** consist of **8x8 pixel handwritten digits from 0 to 9**:



You want to distinguish between each of the 10 possible digits given an image, so we are dealing with **multi-class classification**.

The dataset has already been partitioned into X\_train, y\_train, X\_test, and y\_test, using 30% of the data as testing data. The labels are already one-hot encoded vectors, so you don't need to use Keras to\_categorical() function.

Let's build this new model!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Add a Dense layer of 16 neurons with relu activation and an input\_shape that takes the **total number of pixels** of the 8x8 digit image.

Add a Dense layer with 10 outputs and softmax activation.

Compile your model with adam, categorical\_crossentropy, and accuracy metrics.

Make sure your model works by predicting on X\_train.

# Instantiate a Sequential model

model = Sequential()

# Input and hidden layer with input\_shape, 16 neurons, and relu

model.add(Dense(16, input\_shape = (64,), activation = 'relu'))

# Output layer with 10 neurons (one per digit) and softmax

model.add(Dense(10, activation='softmax'))

# Compile your model

model.compile(optimizer = 'adam', loss = 'categorical\_crossentropy', metrics = ['accuracy'])

# Test if your model is well assembled by predicting before training

print(model.predict(X\_train))

<script.py> output:

[[1.57801419e-01 3.13342916e-08 1.17609663e-04 ... 2.88670161e-03

1.75133277e-08 9.27261251e-04]

[9.17966962e-01 4.87130869e-08 1.09600009e-08 ... 1.81080788e-04

8.53955407e-06 9.01037129e-05]

[9.99938369e-01 1.82372684e-09 9.08111347e-12 ... 2.19022222e-05

2.59289088e-15 4.20937489e-08]

...

[5.37219822e-01 5.52924506e-09 1.57055577e-10 ... 1.38584892e-05

4.47214532e-09 1.09312405e-05]

[2.70578653e-01 5.34917831e-07 8.48428527e-08 ... 1.55824000e-05

4.48651798e-03 3.27920467e-02]

[4.90147155e-03 2.87994535e-05 1.48348074e-04 ... 1.64761033e-04

2.08042213e-04 1.32970810e-01]]

# Is the model overfitting?

Let's train the model you just built and plot its learning curve to check out if it's overfitting! You can make use of the loaded function plot\_loss() to plot training loss against validation loss, you can get both from the history callback.

If you want to inspect the plot\_loss() function code, paste this in the console: show\_code(plot\_loss)

##### Instructions 1/2

**50 XP**

[1](javascript:void(0))

[2](javascript:void(0))

##### Instructions 1/2

**50 XP**

[1](javascript:void(0))

[2](javascript:void(0))

Train your model for 60 epochs, using X\_test and y\_test as validation data.

Use plot\_loss() passing loss and val\_loss as extracted from the history attribute of the h\_callback object.

# Train your model for 60 epochs, using X\_test and y\_test as validation data

h\_callback = model.fit(X\_train , y\_train, epochs = 60, validation\_data = (X\_test , y\_test), verbose=0)

# Extract from the h\_callback object loss and val\_loss to plot the learning curve

plot\_loss(h\_callback.history['loss'], h\_callback.history['val\_loss'])

#### Question

Just by looking at the picture, do you think the learning curve shows this model is overfitting after having trained for 60 epochs?

##### Possible Answers



Yes, it started to overfit since the test loss is higher than the training loss.



No, the test loss is not getting higher as the epochs go by.

Graphical user interface

Description automatically generated

**Do we need more data?**

It's time to check whether the **digits dataset** model you built benefits from more training examples!

In order to keep code to a minimum, various things are already initialized and ready to use:

The model you just built.

X\_train,y\_train,X\_test, and y\_test.

The initial\_weights of your model, saved after using model.get\_weights().

A pre-defined list of training sizes: training\_sizes.

A pre-defined early stopping callback monitoring loss: early\_stop.

Two empty lists to store the evaluation results: train\_accs and test\_accs.

Train your model on the different training sizes and evaluate the results on X\_test. End by plotting the results with plot\_results().

The full code for this exercise can be found on the slides!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Get a fraction of the training data determined by the size we are currently evaluating in the loop.

Set the model weights to the initial\_weights with set\_weights() and train your model on the fraction of training data using early\_stop as a callback.

Evaluate and store the accuracy for the training fraction and the test set.

Call plot\_results() passing in the training and test accuracies for each training size.

for size in training\_sizes:

    # Get a fraction of training data (we only care about the training data)

    X\_train\_frac, y\_train\_frac = X\_train[:size], y\_train[:size]

    # Reset the model to the initial weights and train it on the new training data fraction

    model.set\_weights(initial\_weights)

    model.fit(X\_train\_frac, y\_train\_frac, epochs = 50, callbacks = [early\_stop])

    # Evaluate and store both: the training data fraction and the complete test set results

    train\_accs.append(model.evaluate(X\_train\_frac, y\_train\_frac)[1])

    test\_accs.append(model.evaluate(X\_test, y\_test)[1])

# Plot train vs test accuracies

plot\_results(train\_accs, test\_accs)

Chart, line chart

Description automatically generated

**Different activation functions**

The sigmoid(),tanh(), ReLU(), and leaky\_ReLU() functions have been defined and ready for you to use. Each function receives an input number *X* and returns its corresponding *Y* value.

Which of the statements below is **false**?

**Instructions**

**50 XP**

**Possible Answers**



The sigmoid() takes a value of *0.5* when *X = 0* whilst tanh() takes a value of *0*.

###### Incorrect Submission

This is true! The sigmoid() at X = 0 takes 0.5 the midpoint value between 0 and 1. The tanh takes 0 for X = 0, the midpoint value between -1 and 1.



The leaky\_ReLU() takes a value of *-0.01* when *X = -1* whilst ReLU() takes a value of *0*.

###### Incorrect Submission

This is true! The leaky\_ReLU() allows for negative values in X to also have a negative Y, this can be useful since that way different negative values have different Y responses. The ReLU() function produces the same output, 0, for any negative value.



The sigmoid() and tanh() both take values close to *-1* for big negative numbers.

**Comparing activation functions**

Comparing activation functions involves a bit of coding, but nothing you can't do!

You will try out different activation functions on the **multi-label model** you built for your farm irrigation machine in chapter 2. The function get\_model('relu') returns a copy of this model and applies the 'relu' activation function to its hidden layer.

You will loop through several activation functions, generate a new model for each and train it. By storing the history callback in a dictionary you will be able to visualize which activation function performed best in the next exercise!

X\_train, y\_train, X\_test, y\_test are ready for you to use when training your models.

**Instructions**

**100 XP**

Fill up the activation functions array with relu,leaky\_relu, sigmoid, and tanh.

Get a new model for each iteration with get\_model() passing the current activation function as a parameter.

Fit your model providing the train and validation\_data, use 20 epochs and set verbose to 0.

# Activation functions to try

activations = ['relu','leaky\_relu', 'sigmoid','tanh']

# Loop over the activation functions

activation\_results = {}

for act in activations:

  # Get a new model with the current activation

  model = get\_model(act)

  # Fit the model and store the history results

  h\_callback = model.fit(X\_train, y\_train, epochs=20,validation\_data = (X\_test , y\_test), verbose=0)

  activation\_results[act] = h\_callback

**Comparing activation functions II**

What you coded in the previous exercise has been executed to obtain theactivation\_results variable, this time **100 epochs were used instead of 20**. This way you will have more epochs to further compare how the training evolves per activation function.

For every h\_callback of each activation function in activation\_results:

The h\_callback.history['val\_loss'] has been extracted.

The h\_callback.history['val\_acc'] has been extracted.

Both are saved into two dictionaries: val\_loss\_per\_function and val\_acc\_per\_function.

Pandas is also loaded as pd for you to use. Let's plot some quick validation loss and accuracy charts!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Use pd.DataFrame()to create a new DataFrame from the val\_loss\_per\_function dictionary.

Call plot() on the DataFrame.

Create another pandas DataFrame from val\_acc\_per\_function.

Once again, plot the DataFrame.

Chart

Description automatically generated

**Changing batch sizes**

You've seen models are usually trained in batches of a fixed size. The smaller a batch size, the more weight updates per epoch, but at a cost of a more unstable gradient descent. Specially if the batch size is too small and it's not representative of the entire training set.

Let's see how different batch sizes affect the accuracy of a simple binary classification model that separates red from blue dots.

You'll use a batch size of one, updating the weights once per sample in your training set for each epoch. Then you will use the entire dataset, updating the weights only once per epoch.

**Instructions 1/2**

**50 XP**

Use get\_model() to get a new, already compiled, model, then train your model for 5 epochs with a batch\_size of 1.

# Get a fresh new model with get\_model

model = get\_model()

# Train your model for 5 epochs with a batch size of 1

model.fit(X\_train, y\_train, epochs=5, batch\_size=1)

print("\n The accuracy when using a batch of size 1 is: ",

      model.evaluate(X\_test, y\_test)[1])

625/700 [=========================>....] - ETA: 0s - loss: 0.2755 - acc: 0.9984

668/700 [===========================>..] - ETA: 0s - loss: 0.2731 - acc: 0.9985

700/700 [==============================] - 1s 1ms/step - loss: 0.2726 - acc: 0.9986

32/300 [==>...........................] - ETA: 0s

300/300 [==============================] - 0s 73us/step

The accuracy when using a batch of size 1 is: 0.9966666666666667

model = get\_model()

# Fit your model for 5 epochs with a batch of size the training set

model.fit(X\_train, y\_train, epochs=5, batch\_size=X\_train.shape[0])

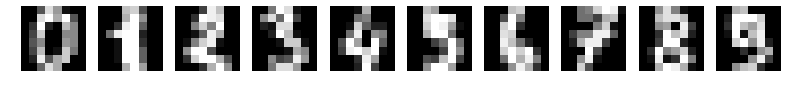
print("\n The accuracy when using the whole training set as batch-size was: ",

      model.evaluate(X\_test, y\_test)[1])

The accuracy when using the whole training set as batch-size was: 0.553333334128062

**Batch normalizing a familiar model**

Remember the **digits dataset** you trained in the first exercise of this chapter?



A multi-class classification problem that you solved using softmax and 10 neurons in your output layer.

You will now build a new deeper model consisting of 3 hidden layers of 50 neurons each, using batch normalization in between layers. The kernel\_initializer parameter is used to initialize weights in a similar way.

**Instructions**

**100 XP**

* Import BatchNormalization from keras layers.
* Build your deep network model, use **50 neurons for each hidden layer** adding batch normalization in between layers.
* Compile your model with stochastic gradient descent, sgd, as an optimizer.

# Import batch normalization from keras layers

from  keras.layers import BatchNormalization

# Build your deep network

batchnorm\_model = Sequential()

batchnorm\_model.add(Dense(50, input\_shape=(64,), activation='relu', kernel\_initializer='normal'))

batchnorm\_model.add(BatchNormalization())

batchnorm\_model.add(Dense(50, activation='relu', kernel\_initializer='normal'))

batchnorm\_model.add(BatchNormalization())

batchnorm\_model.add(Dense(50, activation='relu', kernel\_initializer='normal'))

batchnorm\_model.add(BatchNormalization())

batchnorm\_model.add(Dense(10, activation='softmax', kernel\_initializer='normal'))

# Compile your model with sgd

batchnorm\_model.compile(optimizer='sgd', loss='categorical\_crossentropy', metrics=['accuracy'])

**Batch normalization effects**

Batch normalization tends to increase the learning speed of our models and make their learning curves more stable. Let's see how two identical models with and without batch normalization compare.

The model you just built batchnorm\_model is loaded for you to use. An exact copy of it without batch normalization: standard\_model, is available as well. You can check their summary() in the console. X\_train, y\_train, X\_test, and y\_test are also loaded so that you can train both models.

You will compare the accuracy learning curves for both models plotting them with compare\_histories\_acc().

You can check the function pasting show\_code(compare\_histories\_acc) in the console.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Train the standard\_model for 10 epochs passing in train and validation data, storing its history in h1\_callback.
* Train your batchnorm\_model for 10 epochs passing in train and validation data, storing its history in h2\_callback.
* Call compare\_histories\_acc passing in h1\_callback and h2\_callback.

Chart, line chart

Description automatically generated

# Train your standard model, storing its history callback

h1\_callback = standard\_model.fit(X\_train, y\_train, validation\_data=(X\_test,  y\_test), epochs=10, verbose=0)

# Train the batch normalized model you recently built, store its history callback

h2\_callback = batchnorm\_model.fit(X\_train, y\_train, validation\_data=(X\_test,  y\_test), epochs=10, verbose=0)

# Call compare\_histories\_acc passing in both model histories

compare\_histories\_acc(h1\_callback, h2\_callback)

**Preparing a model for tuning**

Let's tune the hyperparameters of a **binary classification** model that does well classifying the **breast cancer dataset**.

You've seen that the first step to turn a model into a sklearn estimator is to build a function that creates it. The definition of this function is important since hyperparameter tuning is carried out by varying the arguments your function receives.

Build a simple create\_model() function that receives both a learning rate and an activation function as arguments. The Adam optimizer has been imported as an object from keras.optimizers so that you can also change its learning rate parameter.

**Instructions**

**100 XP**

* Set the learning rate of the Adam optimizer object to the one passed in the arguments.
* Set the hidden layers activations to the one passed in the arguments.
* Pass the optimizer and the binary cross-entropy loss to the .compile() method.
* # Creates a model given an activation and learning rate
* def create\_model(learning\_rate, activation):
* # Create an Adam optimizer with the given learning rate
* opt = Adam(lr = learning\_rate)
* # Create your binary classification model
* model = Sequential()
* model.add(Dense(128, input\_shape = (30,), activation = activation))
* model.add(Dense(256, activation = activation))
* model.add(Dense(1, activation = 'sigmoid'))
* # Compile your model with your optimizer, loss, and metrics
* model.compile(optimizer = Adam, loss = BinaryCrossentropy, metrics = ['accuracy'])
* return model

**Tuning the model parameters**

It's time to try out different parameters on your model and see how well it performs!

The create\_model() function you built in the previous exercise is ready for you to use.

Since fitting the RandomizedSearchCV object would take too long, the results you'd get are printed in the show\_results() function. You could try random\_search.fit(X,y) in the console yourself to check it does work after you have built everything else, but you will probably timeout the exercise (so copy your code first if you try this or you can lose your progress!).

You don't need to use the optional epochs and batch\_size parameters when building your KerasClassifier object since you are passing them as params to the random search and this works already.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import KerasClassifier from keras scikit\_learn wrappers.
* Use your create\_model function when instantiating your KerasClassifier.
* Set 'relu' and 'tanh' as activation, 32, 128, and 256 as batch\_size, 50, 100, and 200 epochs, and learning\_rate of 0.1, 0.01, and 0.001.
* Pass your converted model and the chosen params as you build your RandomizedSearchCV object.

# Import KerasClassifier from keras scikit learn wrappers

from keras.wrappers.scikit\_learn import KerasClassifier

# Create a KerasClassifier

model = KerasClassifier(build\_fn = create\_model)

# Define the parameters to try out

params = {'activation':    ['relu' , 'tanh'],

          'batch\_size':    [ 32, 128,  256],

          'epochs':        [50, 100, 200],

          'learning\_rate': [0.1, 0.01,  0.001]}

# Create a randomize search cv object passing in the parameters to try

random\_search = RandomizedSearchCV(model, param\_distributions = params, cv = KFold(3))

# Running random\_search.fit(X,y) would start the search,but it takes too long!

show\_results()

**Training with cross-validation**

Time to train your model with the best parameters found: **0.001** for the **learning rate**, **50 epochs**, **a 128 batch\_size** and **relu activations**.

The create\_model() function from the previous exercise is ready for you to use. X and y are loaded as features and labels.

Use the best values found for your model when creating your KerasClassifier object so that they are used when performing cross\_validation.

End this chapter by training an awesome tuned model on the **breast cancer dataset**!

**Instructions**

**100 XP**

* Import KerasClassifier from keras scikit\_learn wrappers.
* Create a KerasClassifier object providing the best parameters found.
* Pass your model, features and labels to cross\_val\_score to perform cross-validation with 3 folds.
* # Import KerasClassifier from keras wrappers
* from keras.wrappers.scikit\_learn import KerasClassifier
* # Create a KerasClassifier
* model = KerasClassifier(build\_fn =
* create\_model(
* learning\_rate = 0.001,
* activation = 'relu'),
* epochs = 50,
* batch\_size = 128,
* verbose = 0)
* # Calculate the accuracy score for each fold
* kfolds = cross\_val\_score(model, X, y, cv = 3)
* # Print the mean accuracy
* print('The mean accuracy was:', kfolds.mean())
* # Print the accuracy standard deviation
* print('With a standard deviation of:', kfolds.std())
* The mean accuracy was: 0.9718834066666666
* With a standard deviation of: 0.002448915612216046

**It's a flow of tensors**

If you have already built a model, you can use the model.layers and the keras.backend to build functions that, provided with a valid input tensor, return the corresponding output tensor.

This is a useful tool when we want to obtain the output of a network at an intermediate layer.

For instance, if you get the input and output from the first layer of a network, you can build an inp\_to\_out function that returns the result of carrying out forward propagation through only the first layer for a given input tensor.

So that's what you're going to do right now!

X\_test from the **Banknote Authentication** dataset and its model are preloaded. Type model.summary() in the console to check it.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import keras.backend as K.
* Use the model.layers list to get a reference to the input and output of the first layer.
* Use K.function() to define a function that maps inp to out.
* Print the results of passing X\_test through the 1st layer.

# Import keras backend

import keras.backend as K

# Input tensor from the 1st layer of the model

inp = model.layers[0].input

# Output tensor from the 1st layer of the model

out = model.layers[0].output

# Define a function from inputs to outputs

inp\_to\_out = K.function([inp], [out])

# Print the results of passing X\_test through the 1st layer

print(inp\_to\_out([X\_test]))

**Neural separation**

Put on your gloves because you're going to perform brain surgery!

Neurons learn by updating their weights to output values that help them better distinguish between the different output classes in your dataset. You will make use of the inp\_to\_out() function you just built to visualize the output of two neurons in the first layer of the **Banknote Authentication** model as it learns.

The model you built in chapter 2 is ready for you to use, just like X\_test and y\_test. Paste show\_code(plot) in the console if you want to check plot().

You're performing heavy duty, once all is done, click through the graphs to watch the separation live!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Use the previously defined inp\_to\_out() function to get the outputs of the first layer when fed with X\_test.
* Use the model.evaluate() method to obtain the validation accuracy for the test dataset at each epoch
* for i in range(0, 21):
* # Train model for 1 epoch
* h = model.fit(X\_train, y\_train,
* batch\_size = 16,
* epochs = 1,
* verbose = 0)
* if i%4==0:
* # Get the output of the first layer
* layer\_output = inp\_to\_out([X\_test])[0]
* # Evaluate model accuracy for this epoch
* test\_accuracy = model.evaluate( X\_test , y\_test)[1]
* # Plot 1st vs 2nd neuron output
* plot()

Chart, scatter chart

Description automatically generated

**Building an autoencoder**

Autoencoders have several interesting applications like anomaly detection or image denoising. They aim at producing an output identical to its inputs. The input will be compressed into a lower dimensional space, **encoded**. The model then learns to **decode** it back to its original form.

You will encode and decode the **MNIST** dataset of handwritten digits, the hidden layer will encode a 32-dimensional representation of the image, which originally consists of 784 pixels (28 x 28). The autoencoder will essentially learn to turn the 784 pixels original image into a compressed 32 pixels image and learn how to use that encoded representation to bring back the original 784 pixels image.

The Sequential model and Dense layers are ready for you to use.

Let's build an autoencoder!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Create a Sequential model.
* Add a dense layer with as many neurons as the encoded image dimensions and input\_shape the number of pixels in the original image.
* Add a final layer with as many neurons as pixels in the input image.
* Compile your autoencoder using adadelta as an optimizer and binary\_crossentropy loss, then summarise it.
* # Start with a sequential model
* autoencoder = Sequential()
* # Add a dense layer with input the original image pixels and neurons the encoded representation
* autoencoder.add(Dense(32, input\_shape=(784, ), activation="relu"))
* # Add an output layer with as many neurons as the orginal image pixels
* autoencoder.add(Dense(784, activation = "sigmoid"))
* # Compile your model with adadelta
* autoencoder.compile(optimizer = 'adadelta', loss = 'binary\_crossentropy')
* # Summarize your model structure
* autoencoder.summary()
* <script.py> output:
* Model: "sequential\_1"
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* Layer (type) Output Shape Param #
* =================================================================
* dense\_1 (Dense) (None, 32) 25120
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* dense\_2 (Dense) (None, 784) 25872
* =================================================================
* Total params: 50,992
* Trainable params: 50,992
* Non-trainable params: 0

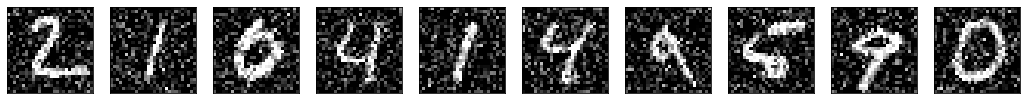
# De-noising like an autoencoder

Okay, you have just built an autoencoder model. Let's see how it handles a more challenging task.

First, you will build a model that encodes images, and you will check how different digits are represented with show\_encodings(). To build the encoder you will make use of your autoencoder, that has already being trained. You will just use the first half of the network, which contains the input and the bottleneck output. That way, you will obtain a 32 number output which represents the encoded version of the input image.

Then, you will apply your autoencoder to noisy images from MNIST, it should be able to clean the noisy artifacts.

X\_test\_noise is loaded in your workspace. The digits in this noisy dataset look like this:



Apply the power of the autoencoder!

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Build an encoder model with the first layer of your trained autoencoder model.
* Predict on X\_test\_noise with your encoder and show the results with show\_encodings().
* # Build your encoder by using the first layer of your autoencoder
* encoder = Sequential()
* encoder.add(autoencoder.layers[0])
* # Encode the noisy images and show the encodings for your favorite number [0-9]
* encodings = encoder.predict(X\_test\_noise)
* show\_encodings(encodings, number = 1)

Diagram

Description automatically generated

* Predict on X\_test\_noise with your autoencoder, this will effectively perform both the encoding and decoding.
* Plot noisy vs decoded images with compare\_plot().

# Build your encoder by using the first layer of your autoencoder

encoder = Sequential()

encoder.add(autoencoder.layers[0])

# Encode the noisy images and show the encodings for your favorite number [0-9]

encodings = encoder.predict(X\_test\_noise)

show\_encodings(encodings, number = 4)

# Predict on the noisy images with your autoencoder

decoded\_imgs = autoencoder.predict(X\_test\_noise)

# Plot noisy vs decoded images

compare\_plot(X\_test\_noise, decoded\_imgs)

**Building a CNN model**

Building a CNN model in Keras isn't much more difficult than building any of the models you've already built throughout the course! You just need to make use of convolutional layers.

You're going to build a shallow convolutional model that classifies the **MNIST** digits dataset. The same one you de-noised with your autoencoder! The images are 28 x 28 pixels and **just have one channel**, since they are black and white pictures.

Go ahead and build this small convolutional model!

**Instructions**

**100 XP**

* Import the Conv2D and Flatten layers and instantiate your model.
* Add a first convolutional layer with 32 filters of size 3x3 and the corresponding 3D tuple as input\_shape.
* Add a second convolutional layer with 16 filters of size 3x3 with relu activation.
* Flatten the previous layer output to create a one-dimensional vector.
* # Import the Conv2D and Flatten layers and instantiate model
* from keras.layers  import Conv2D , Flatten
* model = Sequential()
* # Add a convolutional layer of 32 filters of size 3x3
* model.add(Conv2D(
* filters=32,
* kernel\_size = 3,
* input\_shape = (28, 28, 1),
* activation = 'relu'))
* # Add a convolutional layer of 16 filters of size 3x3
* model.add(Conv2D(16, kernel\_size = 3, activation = 'relu'))
* # Flatten the previous layer output
* model.add(Flatten())
* # Add as many outputs as classes with softmax activation
* model.add(Dense(10, activation = 'softmax'))

**Looking at convolutions**

Inspecting the activations of a convolutional layer is a cool thing. You have to do it at least once in your lifetime!

To do so, you will build a new model with the Keras Model object, which takes in a list of inputs and a list of outputs. The output you will provide to this new model is the first convolutional layer outputs when given an **MNIST** digit as input image.

The convolutional model you built in the previous exercise has already been trained for you. It can now correctly classify **MNIST** handwritten images. You can check it with model.summary() in the console.

Let's look at the convolutional masks that were learned in the first convolutional layer of this model!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Obtain a reference to the outputs of the first convolutional layer in the model.
* Build a new model using the model's first layer input and the first\_layer\_output as outputs.
* Use this first\_layer\_model to predict on X\_test.
* Plot the activations of the first digit of X\_test for the **15th** and the **18th** neuron filter.
* # Obtain a reference to the outputs of the first layer
* first\_layer\_output = model.layers[0].output
* # Build a model using the model's input and the first layer output
* first\_layer\_model = Model(inputs = model.layers[0].input, outputs = first\_layer\_output)
* # Use this model to predict on X\_test
* activations = first\_layer\_model.predict(X\_test)
* # Plot the activations of first digit of X\_test for the 15th filter
* axs[0].matshow(activations[0,:,:,14], cmap = 'viridis')
* # Do the same but for the 18th filter now
* axs[1].matshow(activations[0,:,:,18], cmap = 'viridis')
* plt.show()

Chart, histogram

Description automatically generated

**Preparing your input image**

The original **ResNet50 model** was trained with images of size **224 x 224 pixels** and a number of preprocessing operations; like the subtraction of the mean pixel value in the training set for all training images. You need to pre-process the images you want to predict on in the same way.

When predicting on a single image you need it to fit the model's input shape, which in this case looks like this: (batch-size, width, height, channels),np.expand\_dims with parameter axis = 0 adds the batch-size dimension, representing that a single image will be passed to predict. This batch-size dimension value is 1, since we are only predicting on one image.

You will go over these preprocessing steps as you prepare this dog's (named Ivy) image into one that can be classified by **ResNet50**.

A dog sitting on a bed

Description automatically generated with medium confidence

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import image from keras.preprocessing and preprocess\_input from keras.applications.resnet50.
* Load the image with the right target\_size for your model.
* Turn it into an array with image.img\_to\_array().
* Pre-process img\_expanded the same way the original ResNet50 training images were processed with preprocess\_input().
* # Import image and preprocess\_input
* from keras.preprocessing import image
* from keras.applications.resnet50 import preprocess\_input
* # Load the image with the right target size for your model
* img = image.load\_img(img\_path, target\_size=(224, 224))
* # Turn it into an array
* img\_array = image.img\_to\_array(img)
* # Expand the dimensions of the image, this is so that it fits the expected model input format
* img\_expanded = np.expand\_dims(img\_array, axis = 0)
* # Pre-process the img in the same way original images were
* img\_ready = preprocess\_input(img\_expanded)

# Using a real world model

Okay, so Ivy's picture is ready to be used by **ResNet50**. It is stored in img\_ready and now looks like this:

A dog sitting on a bed

Description automatically generated

**ResNet50** is a model trained on the **Imagenet dataset** that is able to distinguish between 1000 different labeled objects. **ResNet50** is a deep model with 50 layers, you can check it in 3D [**here**](https://tensorspace.org/html/playground/resnet50.html).

ResNet50 and decode\_predictions have both been imported from keras.applications.resnet50 for you.

It's time to use this trained model to find out Ivy's breed!

##### Instructions

**0 XP**

##### Instructions

**0 XP**

* Instantiate a ResNet50 model, setting the weights parameter to be 'imagenet'.
* Use the model to predict on your processed image.
* Decode the first 3 predictions with decode\_predictions().

###### Hint

* Predict with this model using .predict() passing in the processed image.
* decode\_predictions() takes your predictions as it's first argument.
* # Instantiate a ResNet50 model with 'imagenet' weights
* model = ResNet50(weights='imagenet')
* # Predict with ResNet50 on your already processed img
* preds = model.predict(img\_ready)
* # Decode the first 3 predictions
* print('Predicted:', decode\_predictions(preds, top=3)[0])

**Text prediction with LSTMs**

During the following exercises you will build a toy LSTM model that is able to predict the next word using a small text dataset. This dataset consist of cleaned quotes from the **The Lord of the Ring** movies. You can find them in the text variable.

You will turn this text into sequences of **length 4** and make use of the Keras Tokenizer to prepare the features and labels for your model!

The Keras Tokenizer is already imported for you to use. It assigns a unique number to each unique word, and stores the mappings in a dictionary. This is important since the model deals with numbers but we later will want to decode the output numbers back into words.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Split the text into an array of words using .split().
* Make sentences of 4 words each, moving one word at a time.
* Instantiate a Tokenizer(), then fit it on the sentences with .fit\_on\_texts().
* Turn sentences into a sequence of numbers calling .texts\_to\_sequences().
* # Split text into an array of words
* words = text.split()
* # Make sentences of 4 words each, moving one word at a time
* sentences = []
* for i in range(4, len(words)):
* sentences.append(' '.join(words[i-4:i]))
* # Instantiate a Tokenizer, then fit it on the sentences
* tokenizer = Tokenizer()
* tokenizer.fit\_on\_texts(sentences)
* # Turn sentences into a sequence of numbers
* sequences = tokenizer.texts\_to\_sequences(sentences)
* print("Sentences: \n {} \n Sequences: \n {}".format(sentences[:5],sequences[:5]))

**Build your LSTM model**

You've already prepared your sequences of text. It's time to build your LSTM model!

Remember your sequences had 4 words each, your model will be trained on the first three words of each sequence, predicting the 4th one. You are going to use an Embedding layer that will essentially learn to turn words into vectors. These vectors will then be passed to a simple LSTM layer. Our output is a Dense layer with as many neurons as words in the vocabulary and softmax activation. This is because we want to obtain the highest probable next word out of all possible words.

The size of the vocabulary of words (the unique number of words) is stored in vocab\_size.

**Instructions**

**100 XP**

* Import the Embedding, LSTM and Dense layer from Keras layers.
* Add an Embedding() layer of the vocabulary size, that will turn words into 8 number vectors and receive sequences of length 3.
* Add a 32 neuron LSTM() layer.
* Add a hidden Dense() layer of 32 neurons and an output layer of vocab\_size neurons with softmax.
* # Import the Embedding, LSTM and Dense layer
* from keras.layers import Embedding, LSTM , Dense
* model = Sequential()
* # Add an Embedding layer with the right parameters
* model.add(Embedding(input\_dim = 44, input\_length = 3, output\_dim = 8, ))
* # Add a 32 unit LSTM layer
* model.add(LSTM(32))
* # Add a hidden Dense layer of 32 units and an output layer of vocab\_size with softmax
* model.add(Dense(32, activation='relu'))
* model.add(Dense(vocab\_size, activation='softmax'))
* model.summary()

**Decode your predictions**

Your LSTM model has already been trained (details in the previous exercise success message) so that you don't have to wait. It's time to **define a function** that decodes its predictions. The trained model will be passed as a default parameter to this function.

Since you are predicting on a model that uses the softmax function, numpy's argmax() can be used to obtain the index/position representing the most probable next word out of the output vector of probabilities.

The tokenizer you previously created and fitted, is loaded for you. You will be making use of its internal index\_word dictionary to turn the model's next word prediction (which is an integer) into the actual written word it represents.

You're very close to experimenting with your model!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Use texts\_to\_sequences() to turn the test\_text parameter into a sequence of numbers.
* Get the model's next word prediction by passing in test\_seq . The index/position representing the word with the highest probability is obtained by calling .argmax(axis=1)[0] on the numpy array of predictions.
* Return the word that maps to the prediction using the tokenizer's index\_word dictionary.
* def predict\_text(test\_text, model = model):
* if len(test\_text.split()) != 3:
* print('Text input should be 3 words!')
* return False
* # Turn the test\_text into a sequence of numbers
* test\_seq = tokenizer.texts\_to\_sequences([test\_text])
* test\_seq = np.array(test\_seq)
* # Use the model passed as a parameter to predict the next word
* pred = model.predict(test\_seq).argmax(axis = 1)[0]
* # Return the word that maps to the prediction
* return tokenizer.index\_word[pred]

**Test your model!**

The function you just built, predict\_text(), is ready to use. *Remember that the model object is already passed by default as the second parameter so you just need to provide the function with your 3 word sentences.*

Try out these strings on your LSTM model:

* 'meet revenge with'
* 'the course of'
* 'strength of the'

Which sentence could be made with the word output from the sentences above?

**Instructions**

**50 XP**

**Possible Answers**

* 

A **worthless** **gnome** is **king**

* 

**Revenge** is your **history** and **spirit**

* 

Take a **sword** and **ride** to **Florida**

**Submit Answer**

**Take Hint (-15 XP)**