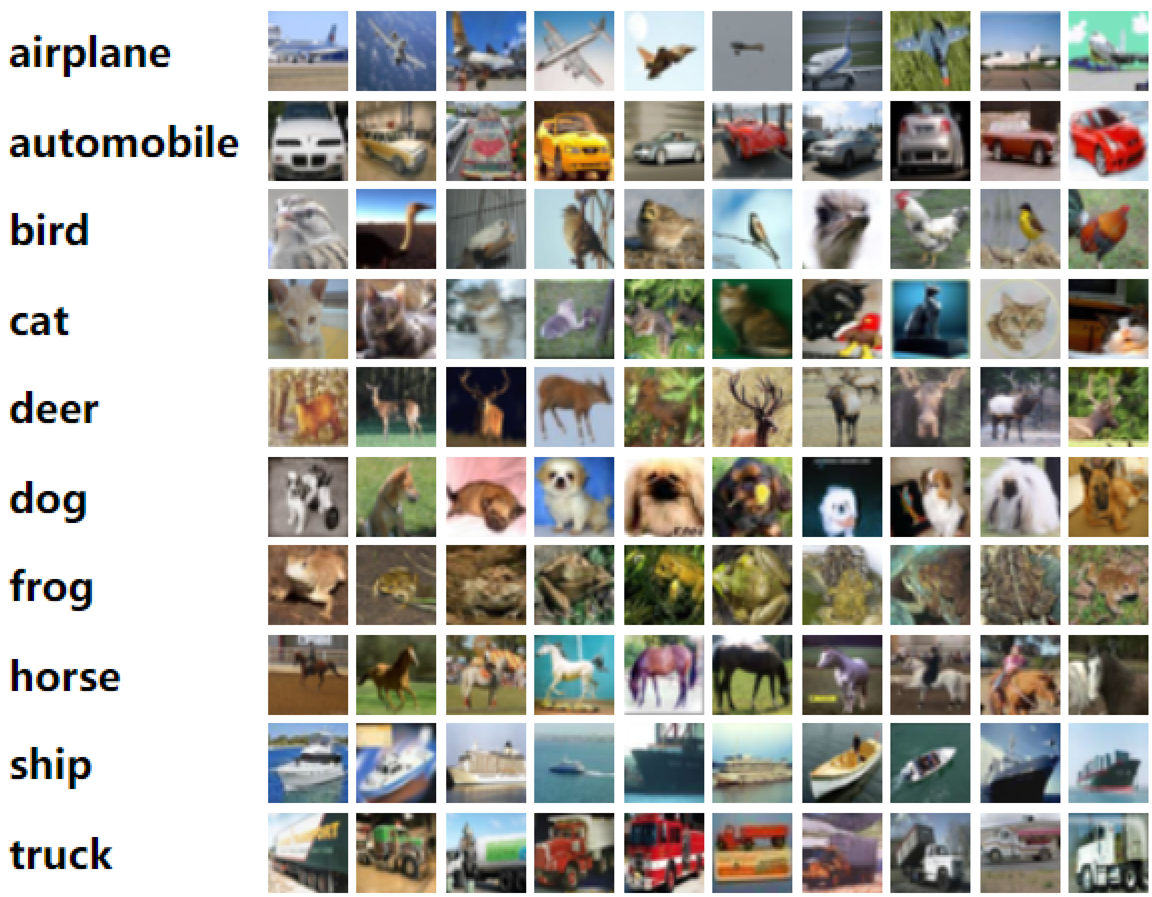
Convolutional Neural Networks for image classification

In this assignment you will create your own Convolutional Neural Networks (CNN) model. You should train the network so that it use at least 3 classes and at most 10 classes where your dataset should have at least 1500 images per class.

> **Be aware:** Advanced neural networks are often trained on high performance (super) computers. our hardware is limited in memory and performance, and more suited for deployment of these kind of networks then for training. But this doesn't mean we can't train on it, we should only be aware that if we want better results and more complex networks you should consider more advanced hardware.

There are several datasets available that are usable for image classification, one of them is the cifar10 dataset, which has 6000 images per class. The Cifar10 dataset, classifies objects like cats, cars, airplanes, etc.



The Cifar100 dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

Other wideley used datasets are https://www.image-net.org/ or https://cocodataset.org/

You can use these datasets but are also allowed to find your own dataset or to even create your own custom dataset.

Use the following website that takes you trought all the steps of development.
https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/

For this assignment:

- You should use at least 3 classes and at most 10 classes where such dataset should have at least 1500 images per class. (more images should lead to better detection performance)
- Show the output of the different (training) steps and the resulting classification and answer the related questions in the subsections below

Initialization

load all needed libraries and functions, check the previous tutorial how to correctly load keras and other modules

Import the needed libraries for this assignment.

In [14]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os

print("Current Tensorflow version used is: " + tf.__version__)
```

```
Current Tensorflow version used is: 2.11.0
```

Functions created for this assignment

In [15]:
```
def plotSample(X, y , classes):
    plt.imshow(X)
    plt.colorbar()
    plt.xlabel(classes[y])
    plt.show()

def plotAccuracyVsEpoch(history):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.show()


def plotLossVsEpoch(history):
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.show()

def plot_image(i, predictions_array, true_label, img, class_name):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
```

```
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_name[predicted_label],
                                100*np.max(predictions_array),
                                class_name[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')

  thisplot[true_label].set_color('blue')
```

Check if a GPU is detected or if CPU must be used to train the TensorFlow model.

In [16]:
```
physical_devices = tf.config.list_physical_devices('GPU')

if (len(physical_devices) > 0):
    details = tf.config.experimental.get_device_details(physical_devices[0])
    print("GPU detected!")
    print("Num GPUs:", len(physical_devices))
    print("GPU Type:", details["device_name"])
    print("Compute Capability:", details["compute_capability"])
else:
    print("No physical devices")
    print("Using CPU to train the model.")
```

```
No physical devices
Using CPU to train the model.
```

Create a variable to trigger training of the model or not.

This is done because the whole notebook can be run at once. If a model is trained already, it would be time consuming to create another model.

In [17]:
```
TrainModel = True
```

Load dataset & Plot a subset

load your dataset and show a plot of the subset of your data

> Just remember that you must use at least 3 classes and at most 10 classes, so, in the case of the cifar10, if you decide to use 5 classes, then get rid of the other 5 to save space. In other words, choose a dataset, check the images (amount, size in pixels) and implement the steps needed shown in the provided notebook.

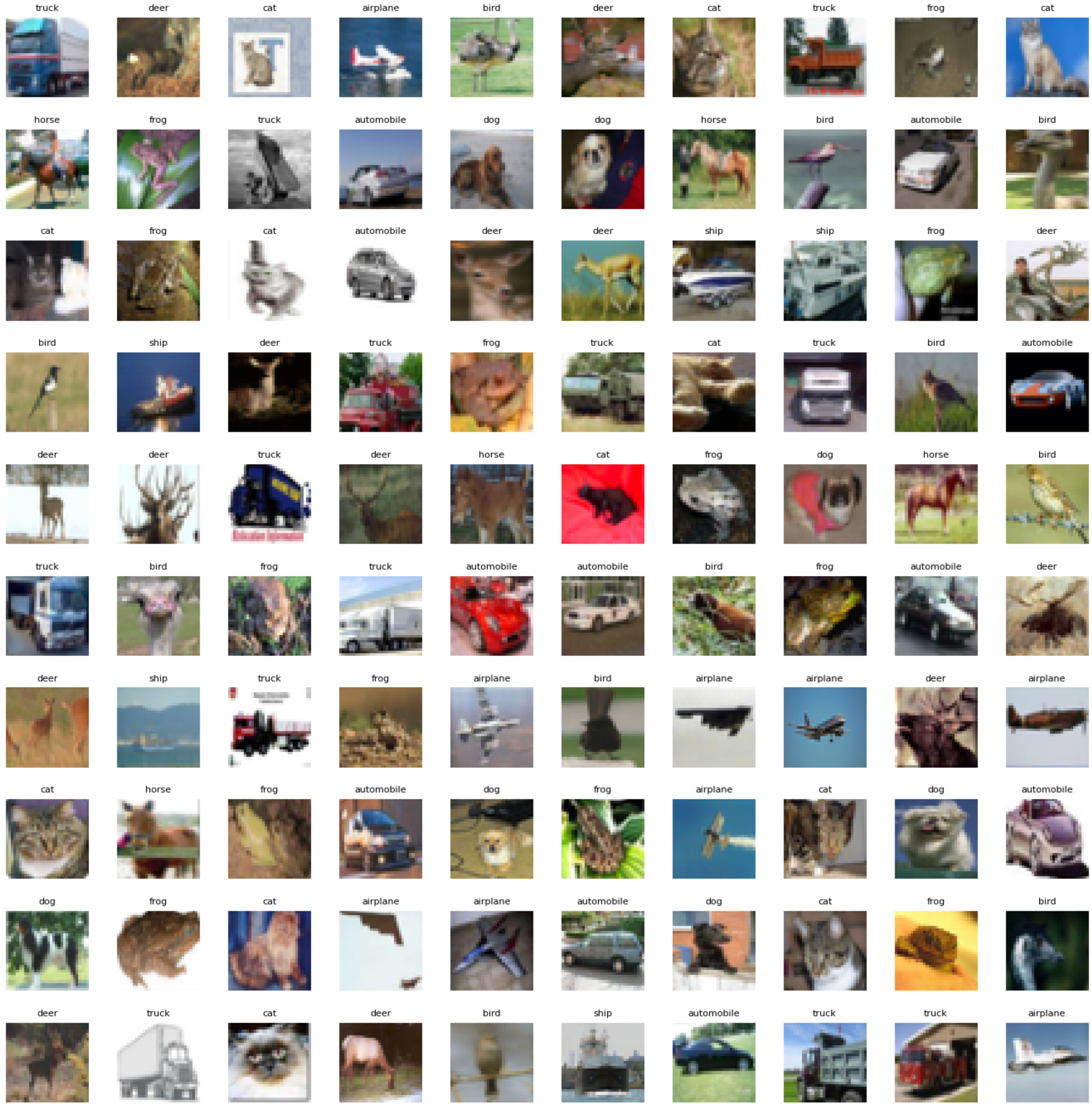
In [18]:
```
cifar10 = tf.keras.datasets.cifar10
(x_train, y_train) , (x_test, y_test) = cifar10.load_data()
class_name = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "
```

Show the train and test shape of the dataset.

In [19]:
```
print("The shape of x_train is: " + str(x_train.shape))
print("The shape of y_train is: " + str(y_train.shape))
print("The shape of x_test is: " + str(x_test.shape))
print("The shape of y_test is: " + str(y_test.shape))
```

```
The shape of x_train is: (50000, 32, 32, 3)
The shape of y_train is: (50000, 1)
The shape of x_test is: (10000, 32, 32, 3)
The shape of y_test is: (10000, 1)
```

In [20]:
```
# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']

# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 10
L_grid = 10

# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
# we can use the axes object to plot specific figures at various locations

fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))

axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array

n_train = len(x_train) # get the length of the train dataset

# Select a random number from 0 to n_train
for i in np.arange(0, W_grid * L_grid): # create evenly spaces variables

    # Select a random number
    index = np.random.randint(0, n_train)
    # read and display an image with the selected index
    axes[i].imshow(x_train[index,1:])
    label_index = int(y_train[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')

    plt.subplots_adjust(hspace=0.4)
```



The training and test labels are reshaped to a 1D array

In [21]:
```
y_train = np.reshape(y_train,(50000))
y_test = np.reshape(y_test,(10000))
```

Prepare Pixel Data

pre-process your raw input data... rescale... normalize....

The pictures in the dataset are already in a 32x32 pixel format. In the next steps, the x_train and x_test (Images) will be normalized by dividing them with 255.0. This results in output data ranging from 0 to 1 instead of 0 to 255. Which is easier to train for the model.

In [23]:
```
x_train[5]
```

```
array([[[159, 102, 101],
```

Out[23]:
```
         [150,  91,  95],
         [153,  95,  97],
         ...,
         [ 91,  71,  56],
         [ 74,  63,  55],
         [ 76,  58,  55]],

        [[142,  75,  68],
         [146,  72,  66],
         [155,  76,  65],
         ...,
         [127, 105,  71],
         [122, 111,  93],
         [ 86,  69,  61]],

        [[109,  67,  75],
         [ 99,  58,  60],
         [105,  59,  52],
         ...,
         [137, 112,  80],
         [163, 132, 105],
         [ 93,  72,  71]],

        ...,

        [[244, 129,  70],
         [240, 123,  65],
         [241, 122,  65],
         ...,
         [156,  42,  15],
         [179,  59,  26],
         [200,  73,  36]],

        [[246, 133,  74],
         [243, 128,  72],
         [243, 127,  70],
         ...,
         [162,  44,  14],
         [178,  56,  22],
         [192,  65,  27]],

        [[246, 139,  82],
         [243, 133,  78],
         [244, 132,  77],
         ...,
         [166,  47,  14],
         [173,  51,  17],
         [182,  57,  19]]], dtype=uint8)
```

In [24]:
```
x_train = x_train / 255.0
x_test = x_test / 255.0
```

In [25]:
```
x_train[100]
```

Out[25]:
```
array([[[0.83529412, 0.89803922, 0.94901961],
        [0.82745098, 0.89019608, 0.94117647],
        [0.82745098, 0.89019608, 0.94117647],
        ...,
        [0.59215686, 0.68235294, 0.80784314],
        [0.59215686, 0.68235294, 0.80784314],
        [0.58431373, 0.6745098 , 0.8       ]],

       [[0.83921569, 0.89803922, 0.94509804],
        [0.83137255, 0.89019608, 0.9372549 ],
        [0.83137255, 0.89019608, 0.9372549 ],
```

```
        ...,
        [0.59607843, 0.68627451, 0.81176471],
        [0.59607843, 0.68627451, 0.81176471],
        [0.59215686, 0.68235294, 0.80392157]],

       [[0.84705882, 0.89803922, 0.9372549 ],
        [0.83921569, 0.89019608, 0.92941176],
        [0.83529412, 0.89019608, 0.92941176],
        ...,
        [0.6       , 0.69019608, 0.80784314],
        [0.6       , 0.69019608, 0.80784314],
        [0.59215686, 0.68235294, 0.8       ]],

       ...,

       [[0.56862745, 0.62352941, 0.64705882],
        [0.53333333, 0.58039216, 0.60392157],
        [0.56078431, 0.59607843, 0.61960784],
        ...,
        [0.84705882, 0.85098039, 0.80784314],
        [0.76862745, 0.77254902, 0.74901961],
        [0.71764706, 0.71764706, 0.71372549]],

       [[0.54509804, 0.6       , 0.62352941],
        [0.50588235, 0.55686275, 0.58039216],
        [0.50588235, 0.54509804, 0.56862745],
        ...,
        [0.89019608, 0.89411765, 0.85882353],
        [0.8745098 , 0.87843137, 0.85882353],
        [0.81960784, 0.81960784, 0.81960784]],

       [[0.5372549 , 0.59607843, 0.61568627],
        [0.56078431, 0.60784314, 0.63137255],
        [0.53333333, 0.56862745, 0.59607843],
        ...,
        [0.81960784, 0.81960784, 0.79607843],
        [0.85098039, 0.85098039, 0.83529412],
        [0.89411765, 0.89411765, 0.88627451]]]])
```

Make a one hot value instead of integer value for the labels

In [26]:
```
y_train_onehot = tf.keras.utils.to_categorical(y_train, 10)
y_test_onehot = tf.keras.utils.to_categorical(y_test, 10)
```

Define your Model

This is the crucial part of the assignment!

We do not expect that you can/should develop your own network model, so you can take the suggested model as decribed on the given website.....but

NOTE:

If you run into memory and processing limitations you can reduce the amount of convolutions and dense layers, you can reduce the amount of classes, you can reduce the amount of input images, or the input images size. With a scaled down network the accuracy will be lower then with a more complex network.

- How is your model constructed, how many trainable parameters does it have, and where are they located?

CNN

The model is constructed using 4 convolution layers, 3 max pooling layers, 3 dropout layers, 1 flatten layer, 2 batch normalization layer and 2 dense layers. So there are 15 layers. Where 366,442 trainable parameters are located in the convolutional layers

In [39]:
```
model = tf.keras.models.Sequential()

# Convolutional Layer
model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), input_shape=(32, 32, 3)
model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), input_shape=(32, 32, 3)
# Dropout layers
model.add(tf.keras.layers.Dropout(0.25))

model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=(3, 3), input_shape=(32, 32, 3)
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.25))

model.add(tf.keras.layers.Conv2D(filters=128, kernel_size=(3, 3), input_shape=(32, 32, 3
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.25))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

METRICS = [
    'accuracy',
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall')
]
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=METRICS)
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 32, 32, 32)        896

 max_pooling2d_3 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 16, 16, 32)        9248

 dropout_3 (Dropout)         (None, 16, 16, 32)        0

 conv2d_6 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization_2 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 dropout_4 (Dropout)         (None, 8, 8, 64)          0

 conv2d_7 (Conv2D)           (None, 8, 8, 128)         73856

 batch_normalization_3 (Batc  (None, 8, 8, 128)        512
 hNormalization)
```

```
 max_pooling2d_5 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 dropout_5 (Dropout)         (None, 4, 4, 128)         0

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 128)               262272

 dense_3 (Dense)             (None, 10)                1290

=================================================================
Total params: 366,826
Trainable params: 366,442
Non-trainable params: 384
_________________________________________________________________
```

Fit the Model

Fitting the model is the time consuming part, this depend on the complexity of the model and the amount of training data. In the fitting process the model is first build up in memory with all the tunable parameters and intercomnnects (with random start values). This is also the limitation of some systems, all these parameters are stored in memory (or when not fitting in a swap file)

TIP: do not start the first time with training a lot of epochs, first see if this and all following steps in your system work and when you are sure that all works train your final model.

- Which batch size and how many epochs give a good result?

A batch size of 24 and 20 epochs give a very good result

In [40]:
```
if TrainModel == True:
    #es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', mode='max', patience=5,

    reduce_learningrate = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                              factor=0.2,
                              patience=3,
                              verbose=1,
                              min_delta=0.0001)

    callbacks_list = [reduce_learningrate]

    history = model.fit(x_train, y_train_onehot, epochs=20, batch_size=24, validation_da
```

```
Epoch 1/20
2084/2084 [==============================] - 50s 23ms/step - loss: 1.5601 - accuracy: 0.
4391 - precision: 0.6487 - recall: 0.2377 - val_loss: 1.3083 - val_accuracy: 0.5286 - va
l_precision: 0.6714 - val_recall: 0.3870 - lr: 0.0010
Epoch 2/20
2084/2084 [==============================] - 48s 23ms/step - loss: 1.1351 - accuracy: 0.
5978 - precision: 0.7358 - recall: 0.4574 - val_loss: 1.0879 - val_accuracy: 0.6194 - va
l_precision: 0.7262 - val_recall: 0.5170 - lr: 0.0010
Epoch 3/20
2084/2084 [==============================] - 49s 23ms/step - loss: 0.9831 - accuracy: 0.
6530 - precision: 0.7694 - recall: 0.5452 - val_loss: 1.1693 - val_accuracy: 0.6253 - va
l_precision: 0.7037 - val_recall: 0.5580 - lr: 0.0010
Epoch 4/20
2084/2084 [==============================] - 48s 23ms/step - loss: 0.8840 - accuracy: 0.
6916 - precision: 0.7893 - recall: 0.5978 - val_loss: 1.2149 - val_accuracy: 0.6063 - va
l_precision: 0.6741 - val_recall: 0.5411 - lr: 0.0010
```

```
Epoch 5/20
2084/2084 [==============================] - 49s 23ms/step - loss: 0.8205 - accuracy: 0.
7128 - precision: 0.8048 - recall: 0.6285 - val_loss: 0.8020 - val_accuracy: 0.7261 - va
l_precision: 0.8055 - val_recall: 0.6593 - lr: 0.0010
Epoch 6/20
2084/2084 [==============================] - 46s 22ms/step - loss: 0.7621 - accuracy: 0.
7354 - precision: 0.8148 - recall: 0.6610 - val_loss: 0.9057 - val_accuracy: 0.6996 - va
l_precision: 0.7834 - val_recall: 0.6339 - lr: 0.0010
Epoch 7/20
2084/2084 [==============================] - 46s 22ms/step - loss: 0.7167 - accuracy: 0.
7497 - precision: 0.8229 - recall: 0.6817 - val_loss: 0.7853 - val_accuracy: 0.7358 - va
l_precision: 0.8152 - val_recall: 0.6646 - lr: 0.0010
Epoch 8/20
2084/2084 [==============================] - 50s 24ms/step - loss: 0.6761 - accuracy: 0.
7658 - precision: 0.8358 - recall: 0.7033 - val_loss: 0.7854 - val_accuracy: 0.7397 - va
l_precision: 0.8012 - val_recall: 0.6916 - lr: 0.0010
Epoch 9/20
2084/2084 [==============================] - 50s 24ms/step - loss: 0.6429 - accuracy: 0.
7763 - precision: 0.8419 - recall: 0.7211 - val_loss: 0.8214 - val_accuracy: 0.7186 - va
l_precision: 0.7939 - val_recall: 0.6586 - lr: 0.0010
Epoch 10/20
2083/2084 [============================>.] - ETA: 0s - loss: 0.6123 - accuracy: 0.7857 -
precision: 0.8482 - recall: 0.7308
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.
2084/2084 [==============================] - 55s 27ms/step - loss: 0.6122 - accuracy: 0.
7858 - precision: 0.8482 - recall: 0.7308 - val_loss: 0.8368 - val_accuracy: 0.7319 - va
l_precision: 0.7865 - val_recall: 0.6896 - lr: 0.0010
Epoch 11/20
2084/2084 [==============================] - 53s 25ms/step - loss: 0.4899 - accuracy: 0.
8298 - precision: 0.8777 - recall: 0.7861 - val_loss: 0.6140 - val_accuracy: 0.7929 - va
l_precision: 0.8420 - val_recall: 0.7586 - lr: 2.0000e-04
Epoch 12/20
2084/2084 [==============================] - 48s 23ms/step - loss: 0.4617 - accuracy: 0.
8387 - precision: 0.8837 - recall: 0.7993 - val_loss: 0.6240 - val_accuracy: 0.7889 - va
l_precision: 0.8352 - val_recall: 0.7572 - lr: 2.0000e-04
Epoch 13/20
2084/2084 [==============================] - 49s 23ms/step - loss: 0.4367 - accuracy: 0.
8468 - precision: 0.8881 - recall: 0.8093 - val_loss: 0.6269 - val_accuracy: 0.7961 - va
l_precision: 0.8395 - val_recall: 0.7654 - lr: 2.0000e-04
Epoch 14/20
2084/2084 [==============================] - ETA: 0s - loss: 0.4209 - accuracy: 0.8519 -
precision: 0.8920 - recall: 0.8177
Epoch 14: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
2084/2084 [==============================] - 48s 23ms/step - loss: 0.4209 - accuracy: 0.
8519 - precision: 0.8920 - recall: 0.8177 - val_loss: 0.6354 - val_accuracy: 0.7963 - va
l_precision: 0.8357 - val_recall: 0.7647 - lr: 2.0000e-04
Epoch 15/20
2084/2084 [==============================] - 48s 23ms/step - loss: 0.3969 - accuracy: 0.
8613 - precision: 0.8988 - recall: 0.8296 - val_loss: 0.6019 - val_accuracy: 0.8038 - va
l_precision: 0.8434 - val_recall: 0.7754 - lr: 4.0000e-05
Epoch 16/20
2084/2084 [==============================] - 48s 23ms/step - loss: 0.3931 - accuracy: 0.
8633 - precision: 0.8991 - recall: 0.8306 - val_loss: 0.6053 - val_accuracy: 0.8042 - va
l_precision: 0.8420 - val_recall: 0.7762 - lr: 4.0000e-05
Epoch 17/20
2084/2084 [==============================] - 48s 23ms/step - loss: 0.3884 - accuracy: 0.
8630 - precision: 0.8995 - recall: 0.8319 - val_loss: 0.6057 - val_accuracy: 0.8043 - va
l_precision: 0.8404 - val_recall: 0.7757 - lr: 4.0000e-05
Epoch 18/20
2083/2084 [============================>.] - ETA: 0s - loss: 0.3800 - accuracy: 0.8667 -
precision: 0.9028 - recall: 0.8367
Epoch 18: ReduceLROnPlateau reducing learning rate to 8.000000525498762e-06.
2084/2084 [==============================] - 48s 23ms/step - loss: 0.3800 - accuracy: 0.
8666 - precision: 0.9027 - recall: 0.8367 - val_loss: 0.6056 - val_accuracy: 0.8053 - va
l_precision: 0.8402 - val_recall: 0.7778 - lr: 4.0000e-05
Epoch 19/20
```

```
2084/2084 [==============================] - 50s 24ms/step - loss: 0.3753 - accuracy: 0.
8684 - precision: 0.9035 - recall: 0.8398 - val_loss: 0.6060 - val_accuracy: 0.8056 - va
l_precision: 0.8397 - val_recall: 0.7786 - lr: 8.0000e-06
Epoch 20/20
2084/2084 [==============================] - 52s 25ms/step - loss: 0.3737 - accuracy: 0.
8685 - precision: 0.9033 - recall: 0.8376 - val_loss: 0.6048 - val_accuracy: 0.8058 - va
l_precision: 0.8408 - val_recall: 0.7788 - lr: 8.0000e-06
```

Evaluate Model

Show the model accuracy after the training process ...

- How accurate is your final model?

In [76]:
```
if TrainModel:
    val_loss, val_acc, val_precision, val_recall = model.evaluate(x_test, y_test_onehot,
    print(f"Validated loss: {val_loss} , Validated Accuracy: {val_acc}")
```

```
10000/10000 [==============================] - 13s 1ms/step - loss: 0.6047 - accuracy:
0.8058 - precision: 0.8408 - recall: 0.7788
Validated loss: 0.6047483682632446 , Validated Accuracy: 0.8058000206947327
```

The final validated accuracy is 80%

learning curves

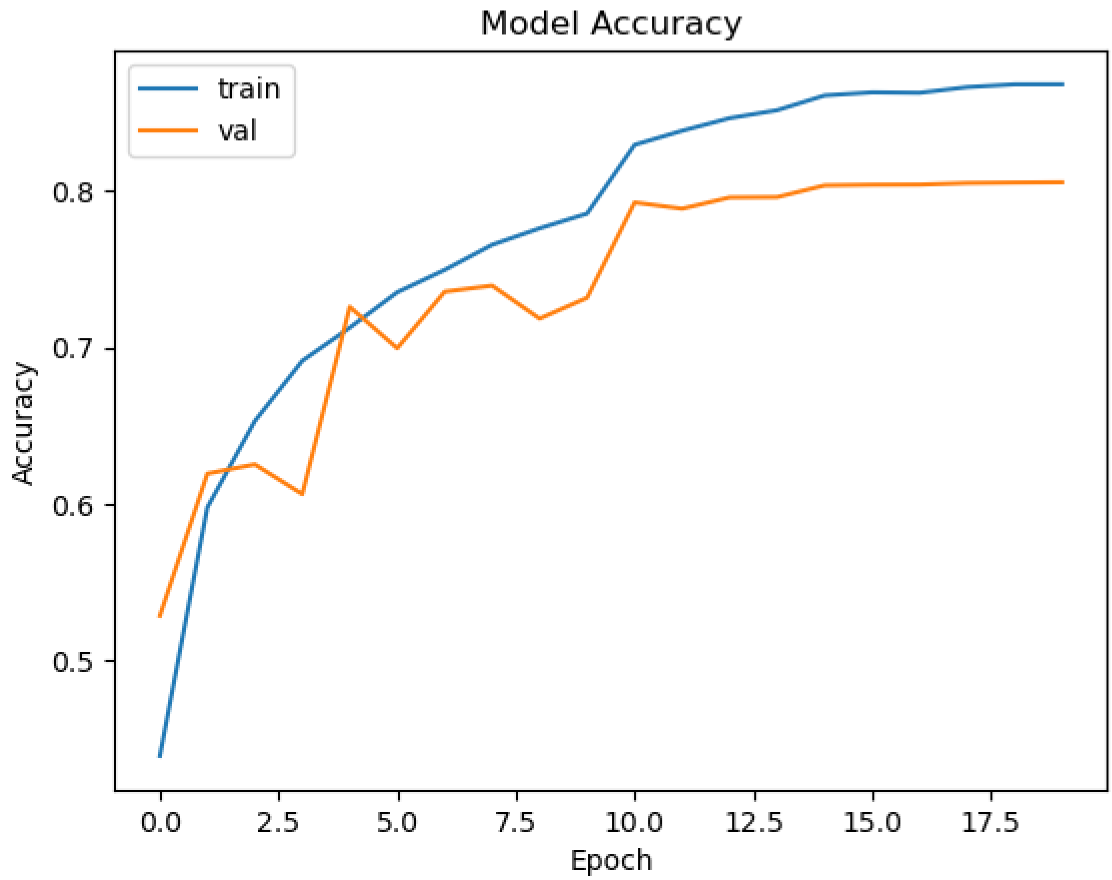
Show the learning curves of your training sequence, of accuracy, value_accuracy and loss, value_loss

- Explain what the difference is between the therms accuracy and value_accuracy? (what do they represent)

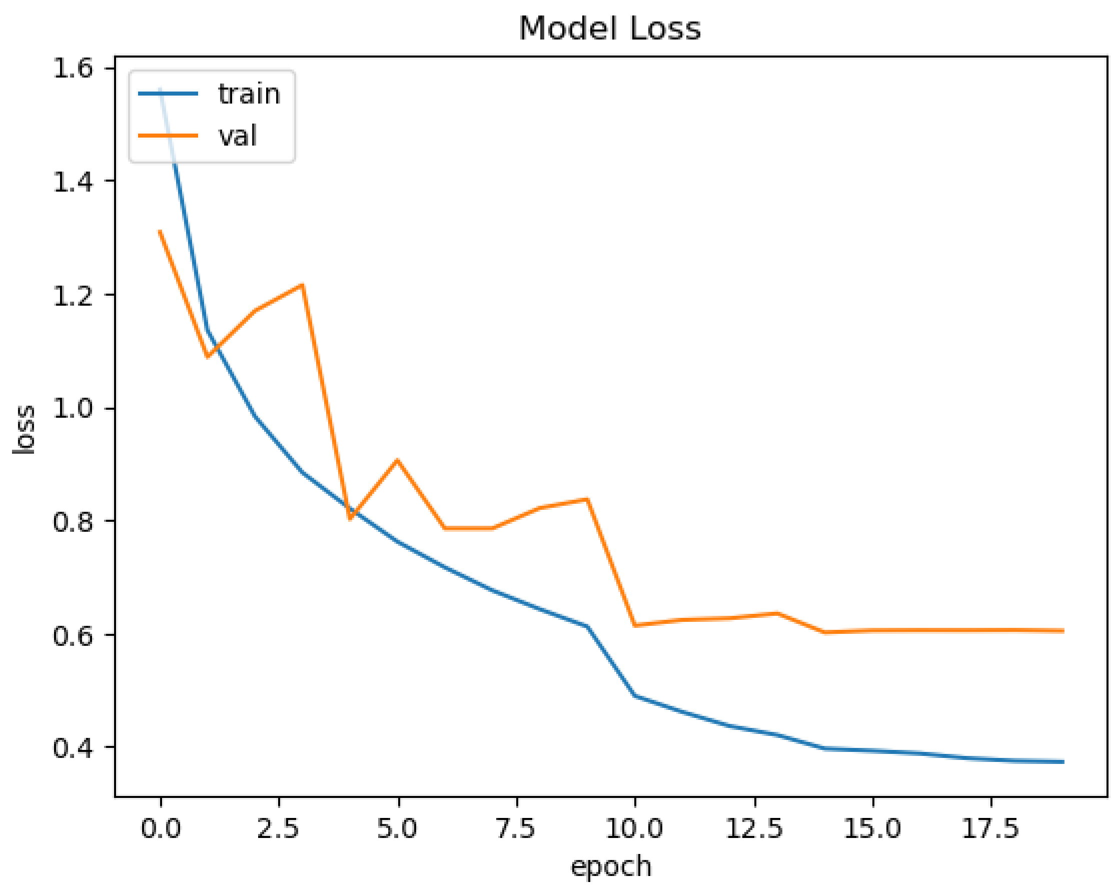
Accuracy and validated accuracy are two different things. Since accuracy is referring to the accuracy of the training data. And validated accuracy is the test data validated on the model which is trained using the training data.

A plot of the accuracy and validated accuracy per epoch is made, and the second plot shows the loss and validated loss per epoch.

In [41]:
```
plotAccuracyVsEpoch(history)
```



In [42]:
```
plotLossVsEpoch(history)
```



Save model

Save the model for later usage

In [43]:
```
if TrainModel:
    model.save('saved_models/model'+ str("9"))
```

```
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compile
d_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _update_st
ep_xla while saving (showing 5 of 5). These functions will not be directly callable afte
r loading.
INFO:tensorflow:Assets written to: saved_models/model8\assets
INFO:tensorflow:Assets written to: saved_models/model8\assets
```

Evaluate Final Model

After training and saving the model you can deploy this model on any given input image. You can start a new application in where you import this model and apply it on any given imput images, so you can just load the model and don't need the timeconsuming training anymore.

Importing own dataset

The model is first loaded into a test_model. We also created a probability model with a softmax function to simplify the prediction process.

In [62]:
```
test_model = tf.keras.models.load_model('./saved_models/model8')
probability_model = tf.keras.Sequential([test_model, tf.keras.layers.Softmax()])
```

Then the images are loaded out of a dataset which we created to verify that the data is correct.

In [63]:
```
import pathlib
import PIL
data_dir = pathlib.Path("./testDataset")
```

In [64]:
```
image_count = len(list(data_dir.glob('*/*.png')))
print(image_count)
```

```
100
```

In [65]:
```
car = list(data_dir.glob('automobile/*'))
PIL.Image.open(str(car[3]))
```

Out[65]:

Then we checked if the pictures were all of a .png or .jpg file. Luckily we did this because there were some corrupt images.

In [66]:
```
from pathlib import Path
```

```python
import imghdr

data_dir = "./testDataset"
image_extensions = [".png", ".jpg"]  # add there all your images file extensions

img_type_accepted_by_tf = ["bmp", "gif", "jpeg", "png"]
for filepath in Path(data_dir).rglob("*"):
    if filepath.suffix.lower() in image_extensions:
        img_type = imghdr.what(filepath)
        if img_type is None:
            print(f"{filepath} is not an image")
        elif img_type not in img_type_accepted_by_tf:
            print(f"{filepath} is a {img_type}, not accepted by TensorFlow")
```

Then we load in the data using a tensorflow function to create a dataset in which we specify the batch size, image height and width and a seed for random shuffling.

In [67]:
```python
batch_size = 100
img_height = 32
img_width = 32


test_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

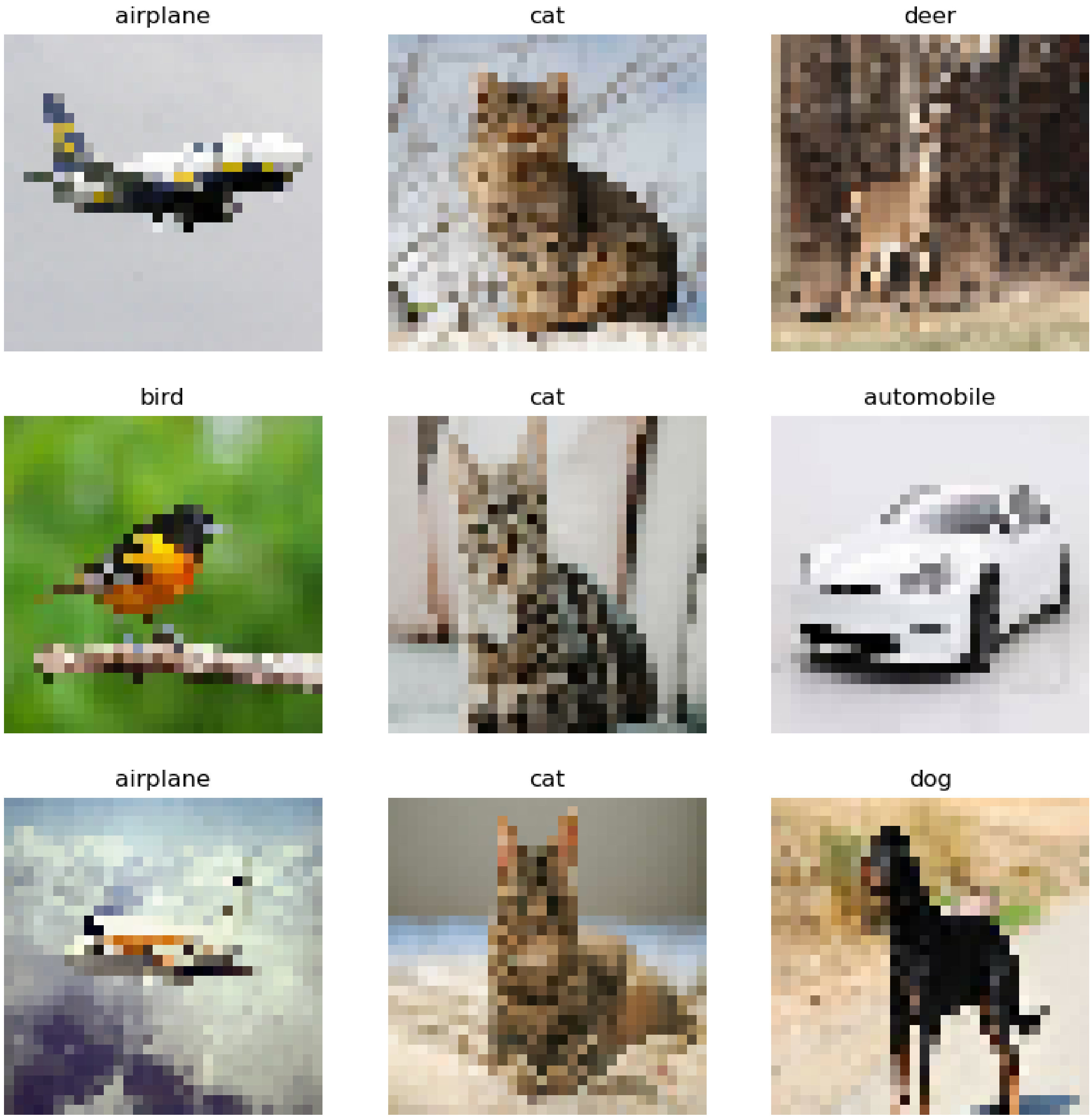
```
Found 100 files belonging to 10 classes.
```

In [68]:
```python
class_names = test_ds.class_names
print(class_names)
```

```
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truc
k']
```

We wanted to verify that the data looks similar to the original dataset and this is the case.

In [69]:
```python
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in test_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```

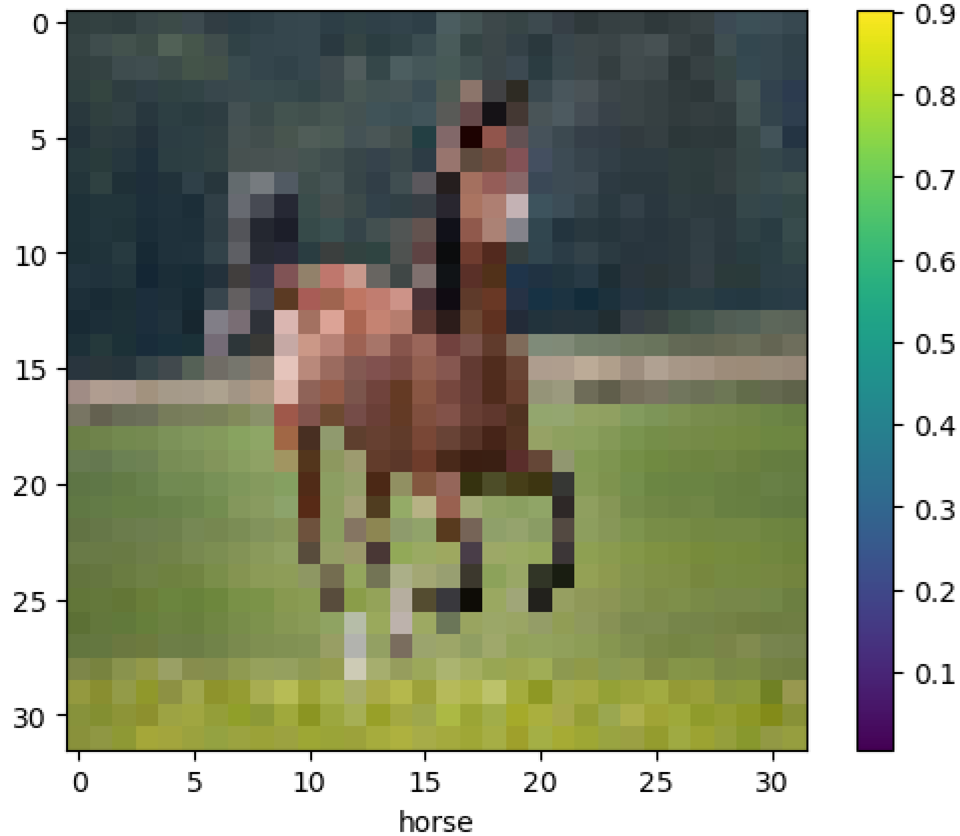


Then we extract the image and labels from the dataset and preprocess them by converting them to an numpy array and dividing the image data through 255. This is so the image values are between 0 and 1.

In [70]:
```
for image_batch, labels_batch in test_ds:
    x_validation = image_batch
    y_validation = labels_batch

y_validation_onehot = tf.keras.utils.to_categorical(y_validation)

x_validation = np.array(x_validation) / 255.0
y_validation = np.array(y_validation)
```

In [71]:
```
index = 22

plt.figure()
plt.imshow(x_validation[index])
plt.colorbar()
plt.grid(False)
```

```
plt.xlabel(class_names[y_validation[index]])
plt.show()
```



Make Prediction

We can use our saved model to make a prediction on new images that are not trained on... make sure the input images receive the same pre-processing as the images you trained on.

So fetch some images from the internet (similar classes, but not from your dataset), prepare them to fit your network and classify them. Do this for **10 images per class** and show the results!

- How good is the detection on you real dataset? (show some statistics)

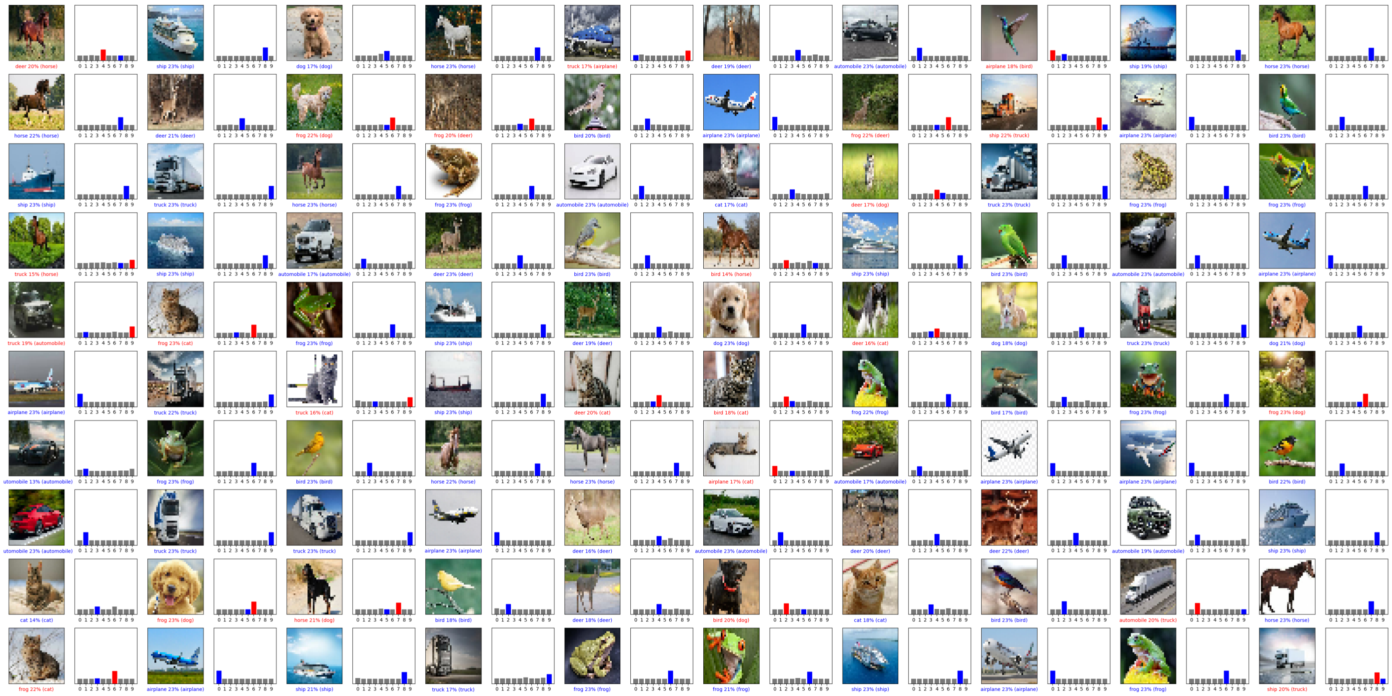
Then we predict using the predict_on_batch function so we can predict the whole dataset at once.

In [72]:
```
predictSource = x_validation
realLabels = y_validation

predictions = probability_model.predict_on_batch(predictSource)
predictionLabels = np.argmax(predictions, axis=1)
```

In the plot can be seen which labels were predicted by the model and the certainty of the prediction. A blue bar indicates a correct prediction and a red bar indicates a incorrect prediction.

In [73]:
```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 10
num_cols = 10
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))

for i in range(num_images):
```

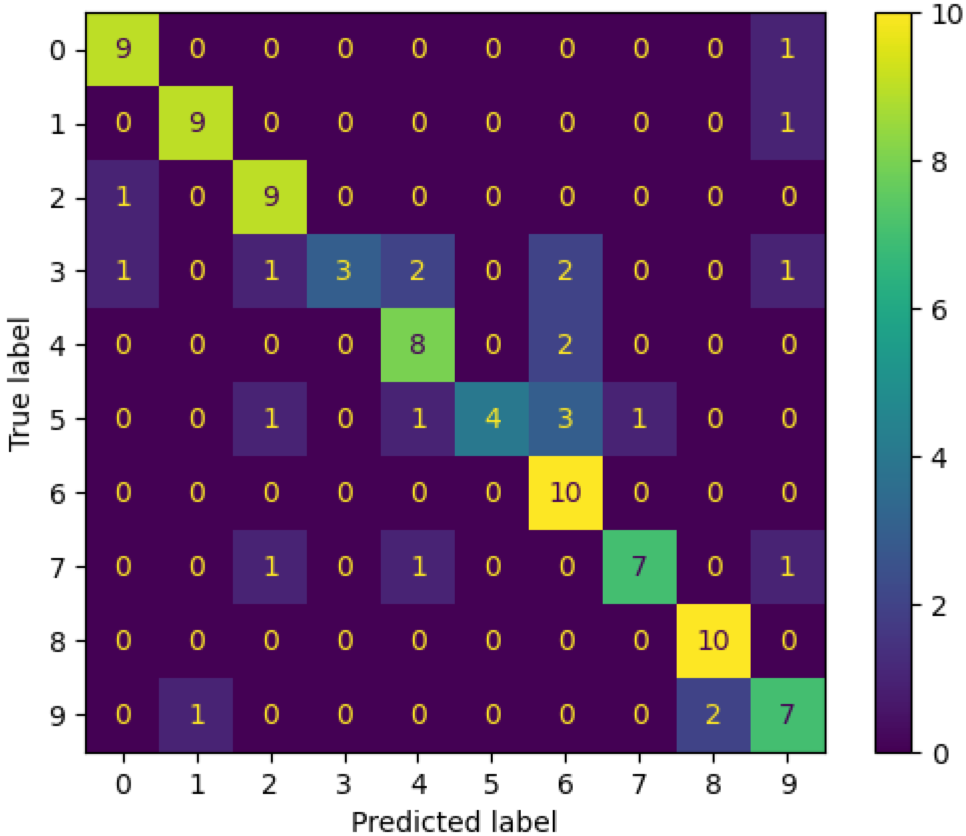
```
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], realLabels, predictSource, class_names)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions[i], realLabels)
plt.tight_layout()
plt.show()
```



Then we made a confusion matrix to see if there were any strange things happening in terms of True Positive, False Negative etc. For own data we are pretty satisfied with the result because the last step also shows that our own dataset has a validated accuracy of 76%.

In [74]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(realLabels, predictionLabels)
ConfusionMatrixDisplay(cm).plot()
```

Out[74]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x27431689430>
```



In [75]:
```
val_loss, val_acc, val_prec, val_rec = test_model.evaluate(predictSource, tf.keras.utils
print(f"Validated loss: {val_loss} , Validated Accuracy: {val_acc}")
```

```
100/100 [==============================] - 0s 2ms/step - loss: 1.0010 - accuracy: 0.7600
- precision: 0.7629 - recall: 0.7400
Validated loss: 1.0009955167770386 , Validated Accuracy: 0.7599999904632568
```