

Introduction

This project is mainly about the product line management. we focus on computer vision-based monitoring. It means that we would like to identify the timestamp when a certain part is missing from the product. We worked on annotating the datasets of our product. Moreover, I trained the dataset using different classification algorithms. This project features deep machine learning approaches. It means that a classification algorithm is run periodically in order to know which parts are to be detected.

Annotation

Using Roboflow software, I have participated in annotating over a hundred images from our dataset.

Moreover, I have annotated 420 images classified into (empty and complete) set.



Using single label tool in Roboflow, I could annotate images of the empty breaks box, and images of the complete breaks box.

Furthermore, I have used the multi label tool to extract each element of the breaks set into a different class of its own. In order for us to be able to identify which element is missing.

Complete set:



Empty set:



One element:



Machine Learning Classifiers

What is classification?

Classification is the process of predicting the class of given data points. Classes are sometimes called as targets/ labels or categories. Classification predictive modeling is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y).

For example, spam detection in email service providers can be identified as a classification problem. This is a binary classification since there are only 2 classes as spam and not spam. A classifier utilizes some training data to understand how given input variables relate to the class. In this case, known spam and non-spam emails have to be used as training data. When the classifier is trained accurately, it can be used to detect an unknown email.

Classification belongs to the category of supervised learning where the targets are also provided with the input data. There are many applications in classification in many domains such as in credit approval, medical diagnosis, target marketing etc.

There are two types of learners in classification as lazy learners and eager learners.

Lazy learners

Lazy learners simply store the training data and wait until a testing data appear. When it does, classification is conducted based on the most related data in the stored training data. Compared to eager learners, lazy learners have less training time but more time in predicting.

Ex. k-nearest neighbor, Case-based reasoning

2. Eager learners

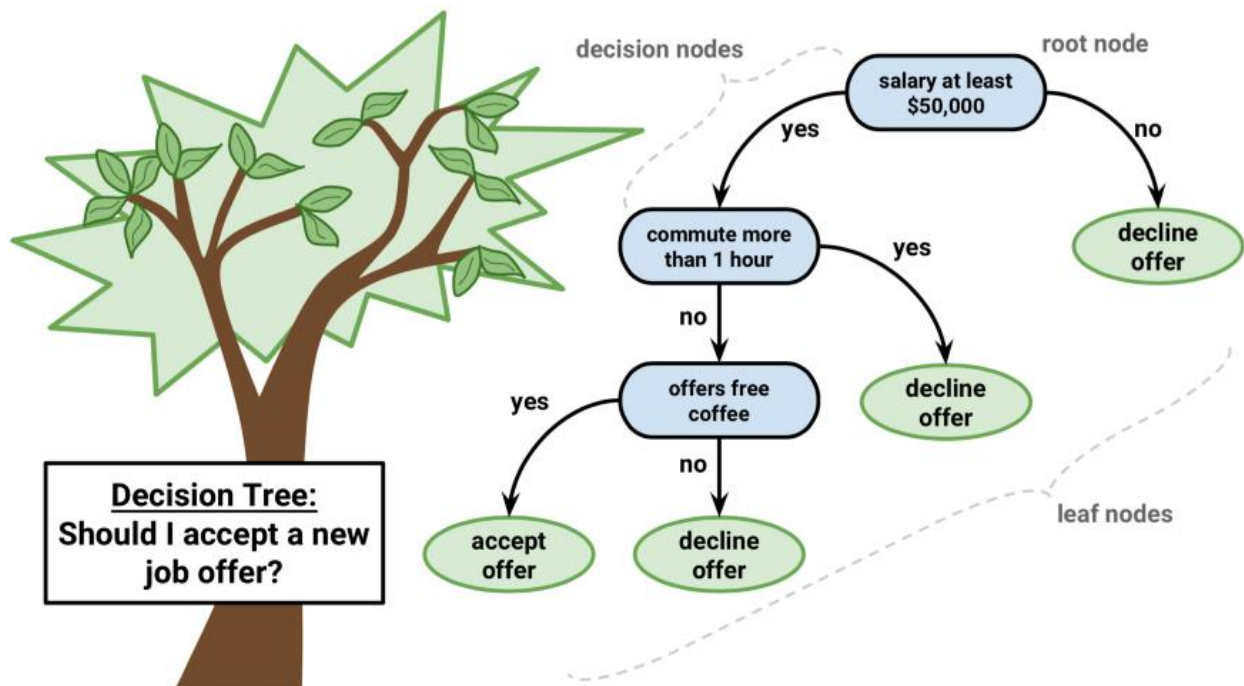
Eager learners construct a classification model based on the given training data before receiving data for classification. It must be able to commit to a single hypothesis that covers the entire instance space. Due to the model construction, eager learners take a long time for train and less time to predict.

Ex. Decision Tree, Naive Bayes, Artificial Neural Networks

Classification algorithms

There is a lot of classification algorithms available now but it is not possible to conclude which one is superior to other. It depends on the application and nature of available data set. For example, if the classes are linearly separable, the linear classifiers like Logistic regression, Fisher's linear discriminant can outperform sophisticated models and vice versa.

Decision Tree



Decision tree builds classification or regression models in the form of a tree structure. It utilizes an if-then rule set which is mutually exclusive and exhaustive for classification. The rules are learned sequentially using the training data one at a time. Each time a rule is learned, the tuples covered by the rules are removed. This process is continued on the training set until meeting a termination condition.

The tree is constructed in a top-down recursive divide-and-conquer manner. All the attributes should be categorical. Otherwise, they should be discretized in advance. Attributes in the top of the tree have more impact towards in the classification and they are identified using the information gain concept.

A decision tree can be easily over-fitted generating too many branches and may reflect anomalies due to noise or outliers. An over-fitted model has a very poor performance on the unseen data even though it gives an impressive performance on training data. This can be avoided by pre-pruning which halts tree construction early or post-pruning which removes branches from the fully grown tree.

Naive Bayes

Naive Bayes is a probabilistic classifier inspired by the Bayes theorem under a simple assumption which is the attributes are conditionally independent.

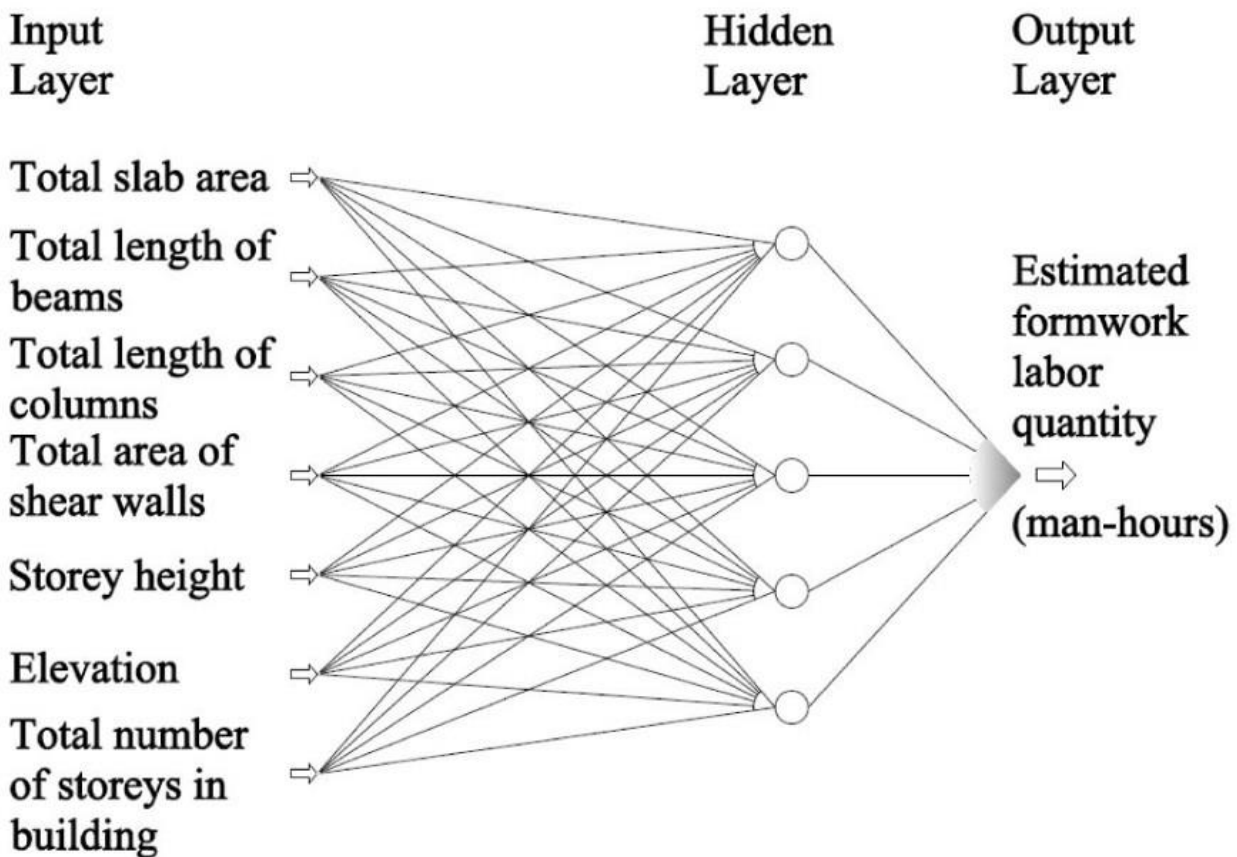
$$P(\mathbf{X} | C_i) = \prod_{k=1}^n P(x_k | C_i) = P(x_1 | C_i) \times P(x_2 | C_i) \times \dots \times P(x_n | C_i)$$

The classification is conducted by deriving the maximum posterior which is the maximal $P(C_i | X)$ with the above assumption applying to Bayes theorem. This assumption greatly reduces the computational cost by only counting the class distribution. Even though the assumption is not valid in most cases since the attributes are dependent, surprisingly Naive Bayes has able to perform impressively.

Naive Bayes is a very simple algorithm to implement and good results have obtained in most cases. It can be easily scalable to larger datasets since it takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

Naive Bayes can suffer from a problem called the zero-probability problem. When the conditional probability is zero for a particular attribute, it fails to give a valid prediction. This needs to be fixed explicitly using a Laplacian estimator.

Artificial Neural Networks



Artificial Neural Network is a set of connected input/output units where each connection has a weight associated with it started by psychologists and neurobiologists to develop and test computational analogs of neurons. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples.

There are many network architectures available now like Feed-forward, Convolutional, Recurrent etc. The appropriate architecture depends on the application of the model. For most cases feed-forward models give reasonably accurate results and especially for image processing applications, convolutional networks perform better.

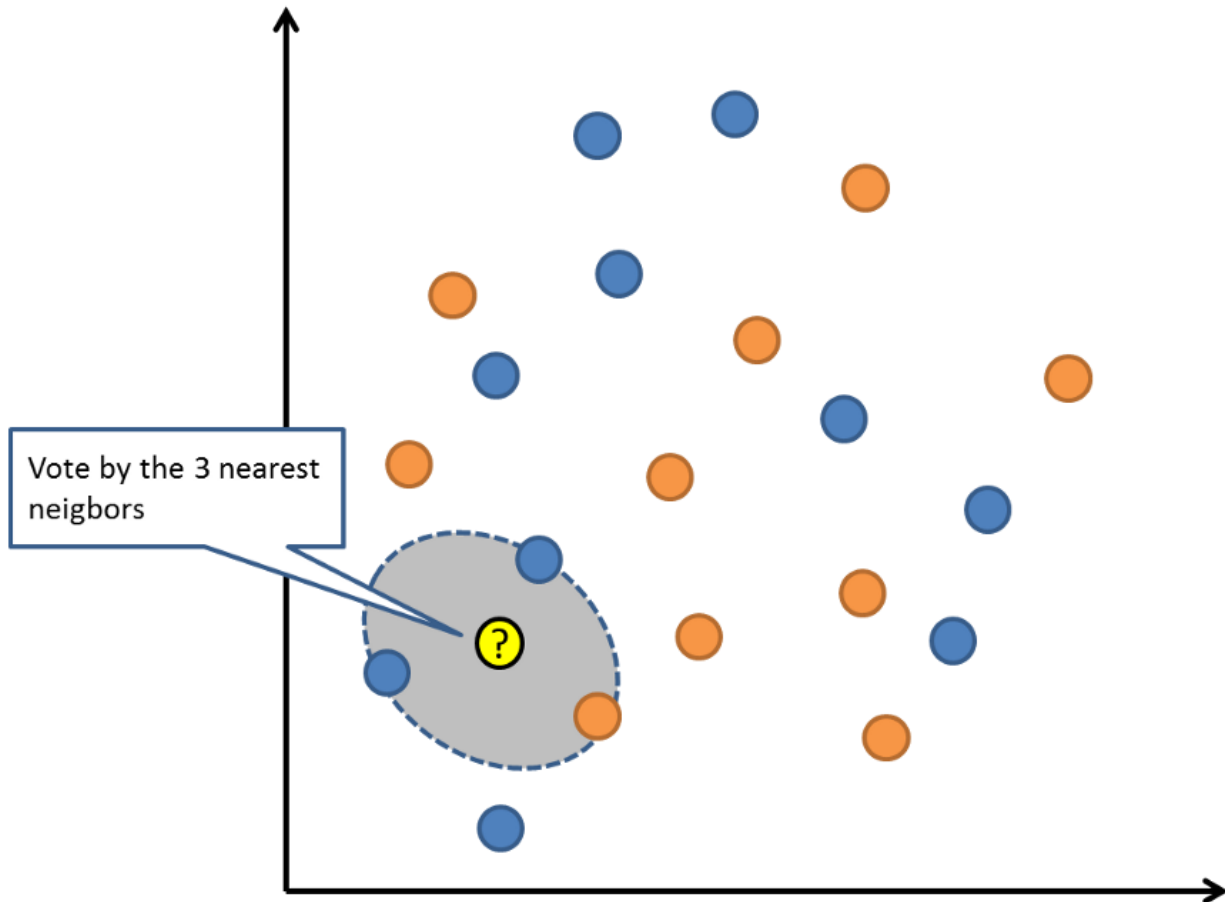
There can be multiple hidden layers in the model depending on the complexity of the function which is going to be mapped by the model. Having more hidden layers will enable to model complex relationships such as deep neural networks.

However, when there are many hidden layers, it takes a lot of time to train and adjust weights. The other disadvantage of is the poor interpretability of model compared to other models like Decision Trees due to the unknown symbolic meaning behind the learned weights.

But Artificial Neural Networks have performed impressively in most of the real world applications. It is high tolerance to noisy data and able to classify untrained patterns. Usually, Artificial Neural Networks perform better with continuous-valued inputs and outputs.

All of the above algorithms are eager learners since they train a model in advance to generalize the training data and use it for prediction later.

k-Nearest Neighbor (KNN)



k-Nearest Neighbor is a lazy learning algorithm which stores all instances correspond to training data points in n-dimensional space. When an unknown discrete data is received, it analyzes the closest k number of instances saved (nearest neighbors) and returns the most common class as the prediction and for real-valued data it returns the mean of k nearest neighbors.

In the distance-weighted nearest neighbor algorithm, it weights the contribution of each of the k neighbors according to their distance using the following query giving greater weight to the closest neighbors.

$$w \equiv \frac{1}{d(x_q, x_i)^2}$$

Distance calculating query

Usually, KNN is robust to noisy data since it is averaging the k-nearest neighbors.

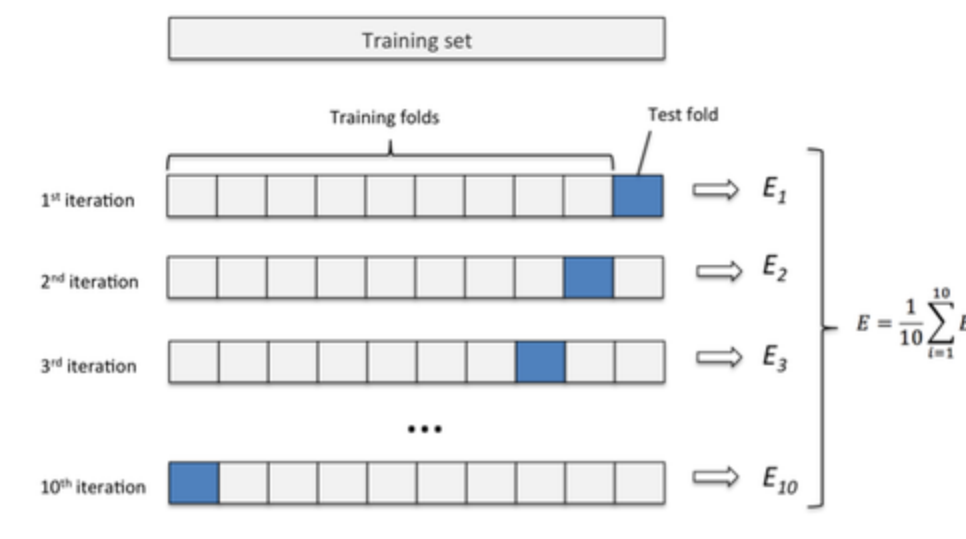
Evaluating a classifier

After training the model the most important part is to evaluate the classifier to verify its applicability.

Holdout method

There are several methods exists and the most common method is the holdout method. In this method, the given data set is divided into 2 partitions as test and train 20% and 80% respectively. The train set will be used to train the model and the unseen test data will be used to test its predictive power.

Cross-validation



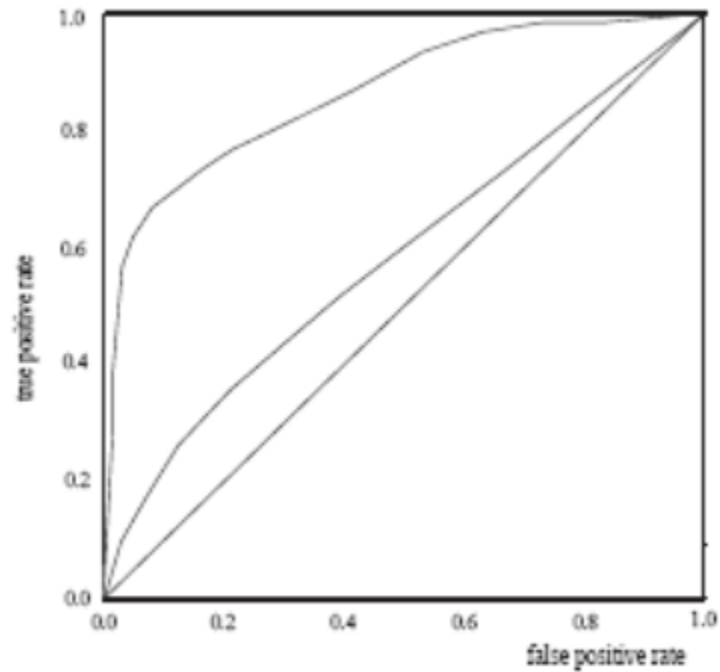
Over-fitting is a common problem in machine learning which can occur in most models. k-fold cross-validation can be conducted to verify that the model is not over-fitted. In this method, the data-set is randomly partitioned into k mutually exclusive subsets, each approximately equal size and one is kept for testing while others are used for training. This process is iterated throughout the whole k folds.

Precision and Recall

Precision is the fraction of relevant instances among the retrieved instances, while recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances. Precision and Recall are used as a measurement of the relevance.

ROC curve (Receiver Operating Characteristics)

ROC curve is used for visual comparison of classification models which shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve is a measure of the accuracy of the model. When a model is closer to the diagonal, it is less accurate and the model with perfect accuracy will have an area of 1.0



ROC Curve

How to train YOLOv5-Classification on a Custom Dataset

Setting Up the Environment

First, we need to clone the Ultralytics YOLOv5 repo and install all its dependencies.

```
1 !git clone https://github.com/ultralytics/yolov5 # clone
2 %cd yolov5
3 %pip install -qr requirements.txt # install
4
5 import torch
6 import utils
7 display = utils.notebook_init() # checks
```


Prepare a Custom Dataset for Classification

in order to train YOLOv5 with a custom dataset, you'll need to gather a dataset, label the data, and export the data in the proper format for YOLOv5 to understand your annotated data

in this project I'm following two types of classification

first part I annotated the empty and the full brakes and in second part I did each specific part of the brakes in a certain class.

Create Project

Labeler Projects /  New Public Project

Project Name

Plant Disease

License

CC BY 4.0

Project Type

✓ Object Detection (Bounding Box)

Single-Label Classification

Multi-Label Classification

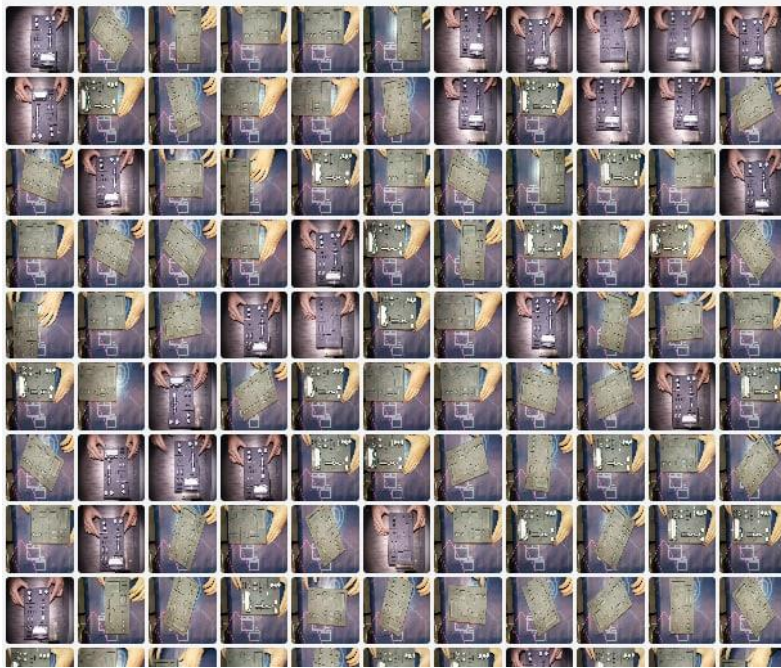
Instance Segmentation

Semantic Segmentation

Keypoint Detection

Other

Here I am chsing single label classification for the empty and full dataset



As can be seen empty and full sets are annotated separately

Annotations

Group: trainbrake-part

CLASSES

LAYERS

complete

0

UNUSED CLASSES

Unlabeled

empty

1 complete

2 empty

3 Unlabeled

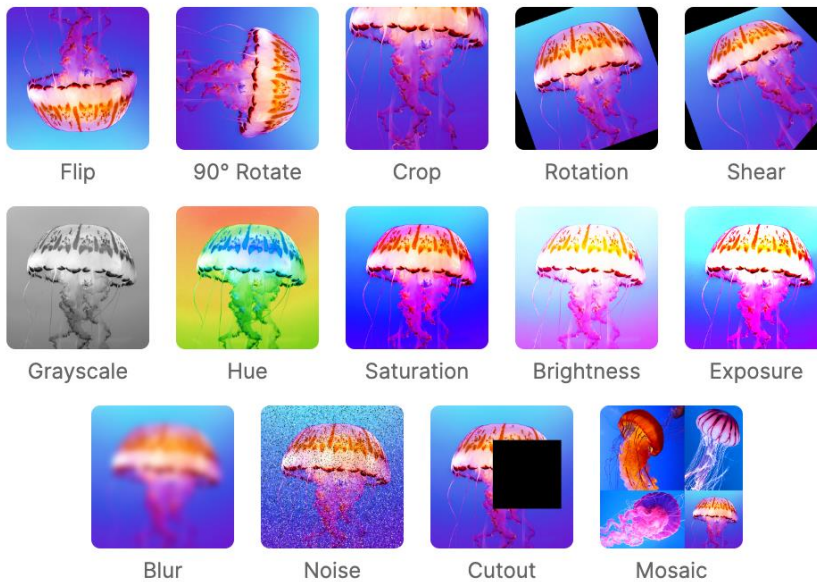
A close-up image of a person's hands holding a black metal component, likely a train brake part, which has several silver-colored bolts and a central silver-colored cylinder.

let's use Auto-Orient and Resize because the brakes can be in any position and we have different image sizes in our dataset.

Augmentation Options

Augmentations create new training examples for your model to learn from.

IMAGE LEVEL AUGMENTATIONS

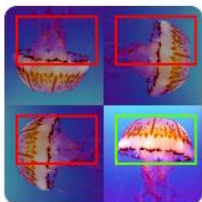


We can apply augmentations that could be possible in real world scenarios. Flipping, rotating, and shear will help when our objects can be in multiple orientations so we will apply those. Blur, noise, and cutout can help when anything is blocking the full view of our object or if other objects might enter our images.

Preprocessing Options



Preprocessing can decrease training time and increase inference speed.



Auto-Orient



Static Crop



Resize



Grayscale



Auto-Adjust
Contrast



Tile



Filter Null

Cancel

4

Augmentation

? What can augmentation do?

Create new training examples for your model to learn from by generating augmented versions of each image in your training set.

Flip Horizontal, Vertical	Edit	×
90° Rotate Clockwise, Counter-Clockwise, Upside Down	Edit	×
Rotation Between -11° and +11°	Edit	×
Shear ±15° Horizontal, ±15° Vertical	Edit	×
Blur Up to 3px	Edit	×
Noise Up to 5% of pixels	Edit	×
Cutout 3 boxes with 16% size each	Edit	×
+ Add Augmentation Step		

Continue

Once we've applied our preprocessing and augmentations, we can generate our dataset which is the final step in creating a dataset to train the model.

After clicking generate, our dataset will be prepared and can then be exported for training in our custom YOLOv5 classification notebook pipeline.

Return to the [YOLOv5 Classification Colab Notebook](#) and paste this generated code snippet. Be sure to paste the code snippet (which includes our Workspace, Project name, and priv

2022-08-19 7:39pm
Version 3 Generated Aug 19, 2022

ExportEdit

TRAINING OPTIONS

Use Roboflow Train
Let us train your model and get results within 24 hours along with a hosted API endpoint for making predictions. [Learn More >>](#)

Start Training

Available Credits: 35

ate API key) to below `%cd ../datasets/` so that the data downloads to the right place.

```
1 !pip install roboflow
2
3 %cd ../datasets/
4
5 from roboflow import Roboflow
6 rf = Roboflow(api_key="YOUR API KEY")
7 project = rf.workspace("workspace-name").project("project-name")
8 dataset = project.version(1).download("folder")
```

In the notebook, we'll also set an environment variable equal to our dataset_name so we can reference this dataset when we call the custom training script below.

Train YOLOv5 For Classification on a Custom Dataset

Before training on our own custom dataset, we'll need to download some pre-trained weights so we aren't starting from scratch. This will accelerate the training process and improve the quality of our results.

```

1 %cd ../yolov5
2 from utils.downloads import attempt_download
3
4 p5 = ['n', 's', 'm', 'l', 'x'] # P5 models
5 cls = [f'{x}-cls' for x in p5] # classification models
6
7 for x in cls:
8     attempt_download(f'weights/yolov5{x}.pt')

```

Then, we are ready to train! We can follow the same steps as above with some minor modifications to the input parameters.

```
!python classify/train.py --model yolov5s-cls.pt --data $DATASET_NAME --epochs 25 --img 128 --
pretrained weights/yolov5s-cls.pt
```

Test and Validate the Custom Model

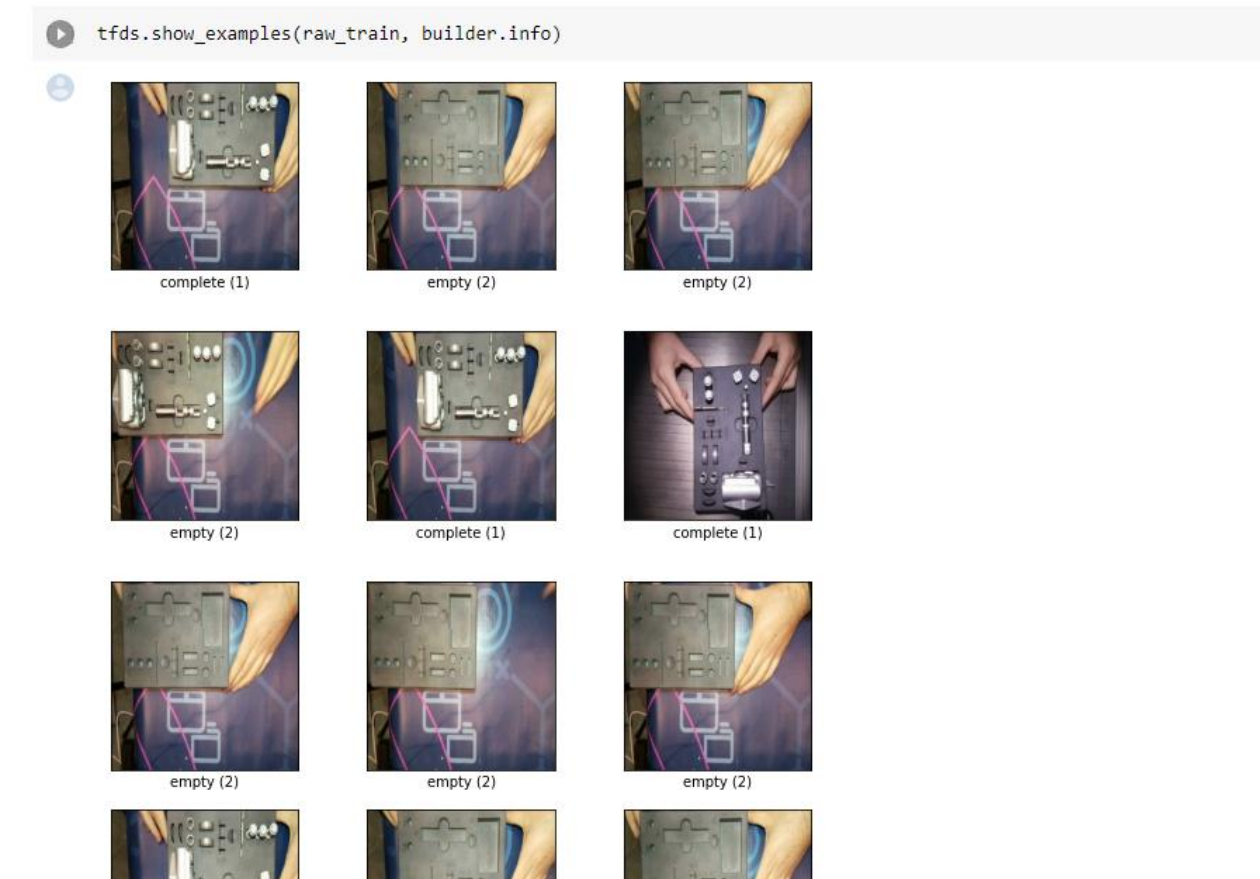
We can test and validate with our newly trained custom model. This script reports the results of the classification training results for each class.

```
!python classify/val.py --weights runs/train-cls/exp/weights/best.pt --data ../datasets/$DATASET_NAME
```

Next, we can try out inferring with our custom model! We'll pass an example image. In the notebook, note that we set a `TEST_IMAGE_PATH` to be one of the images that comes in the test set folder that comes from the Roboflow data loader. This ensures that we're truly testing on an image our model did not see in training.

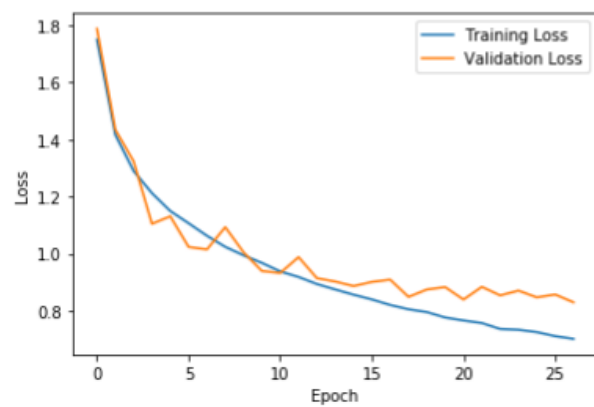
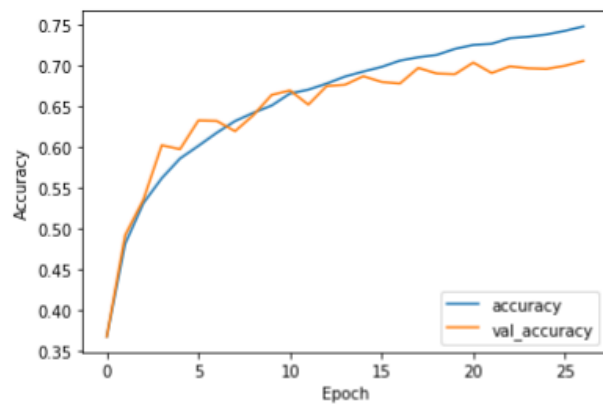
```
!python classify/predict.py --weights runs/train-cls/exp/weights/best.pt --source $TEST_IMAGE_PATH
```


These are examples of our annotations



You can reach code by link below

https://colab.research.google.com/drive/1rGTSeYa6iqKfeo04oPMj798bwmZ_5Evu



Test accuracy: 0.71 / Test loss: 0.83

Binary Classification with MobileNetV2

Data preprocessing

Data download

We'll download our dataset from Roboflow. To download the dataset, use the "Folder Structure" export format.

Format

Folder Structure

Export

```
[ ] !curl -L "https://app.roboflow.com/ds/TsV8JU92v?key=mEG5BhA5Tl" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
    %cd /content/
    %mkdir images/
    %mv train images/train
    %mv test images/test
    %mv valid images/valid
```

We need to turn this dataset into a Tensorflow Dataset format. Fortunately, Tensorflow provides the ImageFolder dataset structure which is compatible with the format we downloaded the data in.

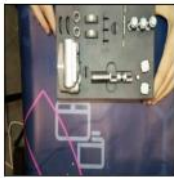
We will then use the builder to build the raw versions of our train, test, and validation data.

CodeText

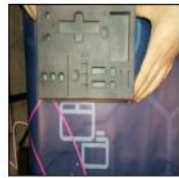
```
[ ] import tensorflow_datasets as tfds
    builder = tfds.folder_dataset.ImageFolder('images/')
    print(builder.info)
    raw_train = builder.as_dataset(split='train', shuffle_files=True)
    raw_test = builder.as_dataset(split='test', shuffle_files=True)
    raw_valid = builder.as_dataset(split='valid', shuffle_files=True)
```

Show images and labels from the training set:

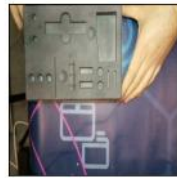
```
[ ] tfds.show_examples(raw_train, builder.info)
```



complete (1)



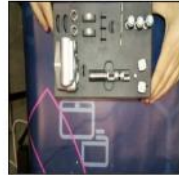
empty (2)



empty (2)



empty (2)



complete (1)



complete (1)



empty (2)



empty (2)



empty (2)



complete (1)



empty (2)



empty (2)

Format the Data

We can use the `tf.image` module to format the images for the task.

Resize the images to a fixed input size, and rescale the input channels to a range of `[-1,1]`

```
[ ] IMG_SIZE = 160 # All images will be resized to 160x160

def format_example(pair):
    image, label = pair['image'], pair['label']
    image = tf.cast(image, tf.float32)
    image = (image/127.5) - 1
    image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
    return image, label
```

Apply this function to each item in the dataset using the map method:

```
[ ] train = raw_train.map(format_example)
    validation = raw_valid.map(format_example)
    test = raw_test.map(format_example)
```

Now shuffle and batch the data.

```
[ ] BATCH_SIZE = 32
    SHUFFLE_BUFFER_SIZE = 1000

[ ] train_batches = train.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
    validation_batches = validation.batch(BATCH_SIZE)
    test_batches = test.batch(BATCH_SIZE)
```

Inspect a batch of data:

```
[ ] for image_batch, label_batch in train_batches.take(1):
    pass

    image_batch.shape

TensorShape([32, 160, 160, 3])
```

Create the base model from the pre-trained convnets

You will create the base model from the MobileNet V2 model developed at Google. This is pre-trained on the ImageNet dataset, a large dataset consisting of 1.4M images and 1000 classes. ImageNet is a research training dataset with a wide variety of categories like jackfruit and syringe. This base of knowledge will help us classify cats and dogs from our specific dataset.

First, you need to pick which layer of MobileNet V2 you will use for feature extraction. The very last classification layer (on "top", as most diagrams of machine learning models go from bottom to top) is not very useful. Instead, you will follow the common practice to depend on the very last layer before the flatten operation. This layer is called the "bottleneck layer". The bottleneck layer features retain more generality as compared to the final/top layer.

First, instantiate a MobileNet V2 model pre-loaded with weights trained on ImageNet. By specifying the `include_top=False` argument, you load a network that doesn't include the classification layers at the top, which is ideal for feature extraction.

```
[ ] IMG_SHAPE = (IMG_SIZE, IMG_SIZE, 3)

# Create the base model from the pre-trained model MobileNet V2
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                              include_top=False,
                                              weights='imagenet')

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet\_v2/mobilenet\_v2\_weights\_tf\_dim\_ordering\_tf\_kernels\_1.0\_160\_no\_top.h5
9406464/9406464 [-----] - 0s 0us/step
```

This feature extractor converts each 160x160x3 image into a 5x5x1280 block of features. See what it does to the example batch of images:

```
[ ] feature_batch = base_model(image_batch)
print(feature_batch.shape)

(32, 5, 5, 1280)
```

Add a classification head

To generate predictions from the block of features, average over the spatial 5x5 spatial locations, using a `tf.keras.layers.GlobalAveragePooling2D` layer to convert the features to a single 1280-element vector per image.

```
[ ] global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
print(feature_batch_average.shape)

(32, 1280)
```

Apply a `tf.keras.layers.Dense` layer to convert these features into a single prediction per image. You don't need an activation function here because this prediction will be treated as a `logit`, or a raw prediction value. Positive numbers predict class 1, negative numbers predict class 0.

```
[ ] prediction_layer = tf.keras.layers.Dense(1)
prediction_batch = prediction_layer(feature_batch_average)
print(prediction_batch.shape)

(32, 1)
```

Now stack the feature extractor, and these two layers using a `tf.keras.Sequential` model:

```
[ ] model = tf.keras.Sequential([
    base_model,
    global_average_layer,
    prediction_layer
])
```

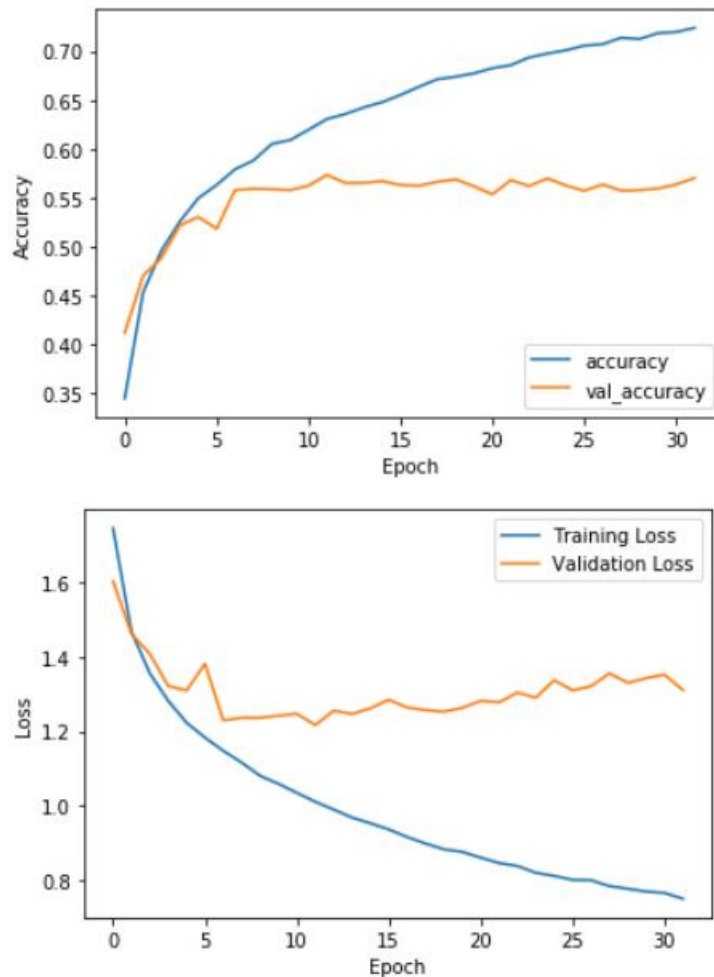
Let's take a look at the learning curves of the training and validation accuracy/loss when using the MobileNet V2 base model as a fixed feature extractor.

```
[ ] acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.ylabel('Accuracy')
plt.ylim([min(plt.ylim()), 1])
plt.title('Training and Validation Accuracy')

plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.ylabel('Cross Entropy')
plt.ylim([0, 1.0])
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```



Test loss: 1.25/

Test accuracy: 0.58

The training accuracy keeps rising but the validation accuracy plateaus very early. Hence, the model is very overfitting, despite many Dropout layers.

Summary:

Using a pre-trained model for feature extraction: When working with a small dataset, it is a common practice to take advantage of features learned by a model trained on a larger dataset in the same domain. This is done by instantiating the pre-trained model and adding a fully-connected classifier on top. The pre-trained model is "frozen" and only the weights of the classifier get updated during training. In this case,

the convolutional base extracted all the features associated with each image and you just trained a classifier that determines the image class given that set of extracted features.

SSD MOBILR NET V2

<https://colab.research.google.com/drive/1bOzVaDQo8h6Ngstb7AcfzC35OihpHspt>

model summary:

```
base_model.summary()
```

Model: "mobilenetv2_1.00_160"

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 160, 160, 3)]	0	[]
Conv1 (Conv2D)	(None, 80, 80, 32)	864	['input_2[0][0]']
bn_Conv1 (BatchNormalization)	(None, 80, 80, 32)	128	['Conv1[0][0]']
Conv1_relu (ReLU)	(None, 80, 80, 32)	0	['bn_Conv1[0][0]']
expanded_conv_depthwise (DepthwiseConv2D)	(None, 80, 80, 32)	288	['Conv1_relu[0][0]']
expanded_conv_depthwise_BN (BatchNormalization)	(None, 80, 80, 32)	128	['expanded_conv_depthwise[0][0]']
expanded_conv_depthwise_relu (ReLU)	(None, 80, 80, 32)	0	['expanded_conv_depthwise_BN[0][0]']
expanded_conv_project (Conv2D)	(None, 80, 80, 16)	512	['expanded_conv_depthwise_relu[0][0]']
expanded_conv_project_BN (BatchNormalization)	(None, 80, 80, 16)	64	['expanded_conv_project[0][0]']
block_1_expand (Conv2D)	(None, 80, 80, 96)	1536	['expanded_conv_project_BN[0][0]']
block_1_expand_BN (BatchNormalization)	(None, 80, 80, 96)	384	['block_1_expand[0][0]']
block_1_expand_relu (ReLU)	(None, 80, 80, 96)	0	['block_1_expand_BN[0][0]']
block_1_pad (ZeroPadding2D)	(None, 81, 81, 96)	0	['block_1_expand_relu[0][0]']

Now I will use CNN for classifying my dataset

This model is based on the Convolutional Neural Network (CNN) of TensorFlow (33), with adjustments. There are three convolutional layers, after each is a max pooling layer, and two dropout layers with the dropout rate of 0.3 added to prevent overfitting. After that, there are two dense layers, with 256 and 10 units respectively (10 is the number of classes for classification). Between the two dense layers there is a dropout layer with the dropout rate of 0.2. The batch size is 32 and the number of epochs is 32. The optimizer is Adam with the learning rate of 0.0001. Here are the code and the summary of the model:

```
In [0]: model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))

In [0]: model.add(layers.Flatten())

model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.2))

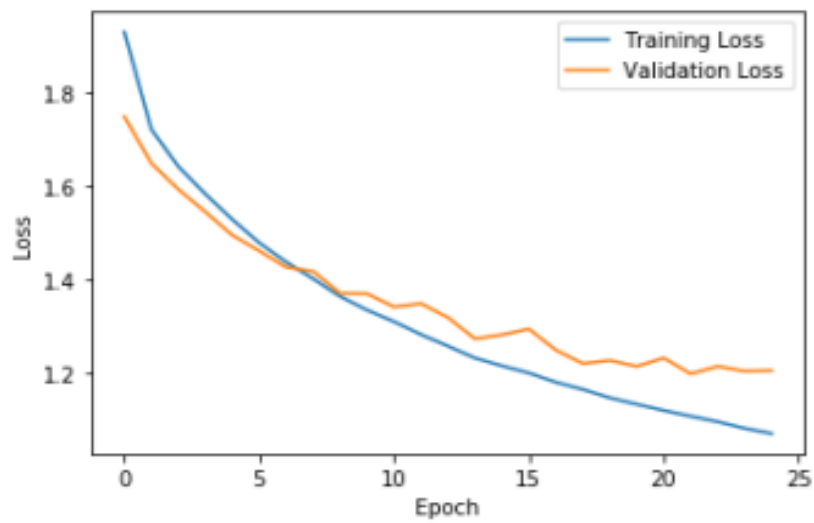
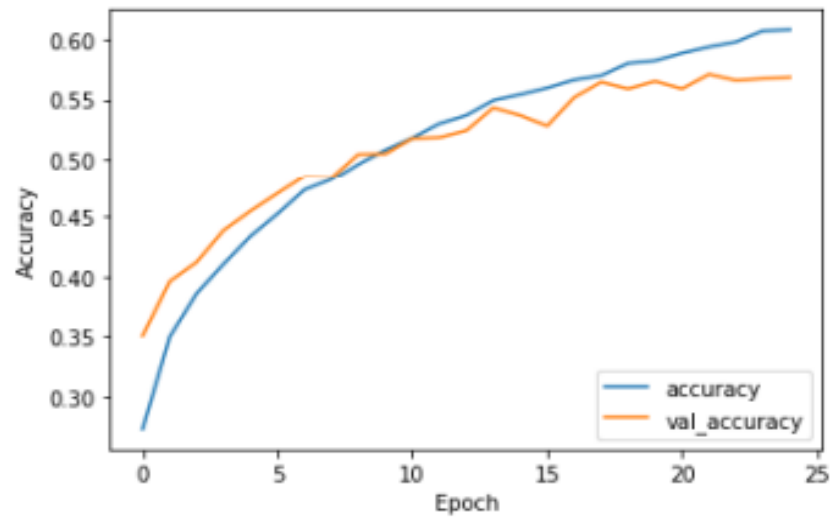
model.add(layers.Dense(10, activation='softmax'))
```

```
model.summary()
```

Model: "sequential_47"

Layer (type)	Output Shape	Param #
conv2d_155 (Conv2D)	(None, 32, 32, 32)	896
max_pooling2d_111 (MaxPoolin	(None, 16, 16, 32)	0
dropout_140 (Dropout)	(None, 16, 16, 32)	0
conv2d_156 (Conv2D)	(None, 16, 16, 64)	18496
max_pooling2d_112 (MaxPoolin	(None, 8, 8, 64)	0
conv2d_157 (Conv2D)	(None, 8, 8, 128)	73856
max_pooling2d_113 (MaxPoolin	(None, 4, 4, 128)	0
dropout_141 (Dropout)	(None, 4, 4, 128)	0
flatten_46 (Flatten)	(None, 2048)	0
dense_111 (Dense)	(None, 256)	524544
dropout_142 (Dropout)	(None, 256)	0
dense_112 (Dense)	(None, 10)	2570
Total params: 620,362		
Trainable params: 620,362		
Non-trainable params: 0		

And the output is as



Test loss: 1.16

Test accuracy: 0.58

The model is not as overfitting, but the accuracy is not high enough (just over 60%).

Comparisons

According to the above graphs, Yolov5 showed most accurate results I could get among the three different algorithms I used.

Moreover, I also i tried to train by <https://colab.research.google.com/drive/1U3fkRu6-hwj7wWlpq-iyIL2u5T9t7rr> by faster R CNN algorithm. However, I faced some issued with the training.