# University of Tehran
# Department of ECE
# Neural Networks & Deep Learning
# MP1


| | |
|---|---|
| Members: | Milad Mohammadi & Tohid Abdi |
| IDs: | 810100462 & 810100410 |
| Date: | April 24, 2022 |

## Questions List Table

# 1 CNN Classification

import essential libraries:

```
[1]: from keras.datasets import cifar10
from sklearn.model_selection import train_test_split
import random
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras.layers import BatchNormalization
import pandas as pd
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support
import keras.utils.vis_utils
from importlib import reload
import pydot
reload(keras.utils.vis_utils)
keras.utils.vis_utils.pydot = pydot
from tensorflow.keras.utils import to_categorical
```

import cifar10:

```
[2]: (X_train_full, y_train_full), (X_test, y_test) = cifar10.load_data()
```

normalize data:

```
[3]: X_train_normal  = X_train_full/255
X_test_normal  = X_test/255
```

train-test split:

```
[4]: X_train, X_val, y_train, y_val = train_test_split(X_train_normal, y_train_full,␣
 ↪test_size=0.2, random_state = 20)
```

transform dependent variables to one hot:

```
[5]: y_train_onehot = to_categorical(y_train, 10)
y_test_onehot = to_categorical(y_test, 10)
y_val_onehot = to_categorical(y_val, 10)
```

set random seed:

```
[6]: np.random.seed(0)
tf.random.set_seed(0)
```

make call back for early stopping:

[7]:
```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=4,␣
↪restore_best_weights=True)
```

define functions to compiling data, plotting accuracis and losses, evaluating test data and drawing model architecture:

[8]:
```
def model_compile(loss, optimizer, batch_s):

    model.compile(loss = loss,
                  optimizer = optimizer,
                  metrics = ['accuracy'])

    global model_history
    model_history = model.fit(X_train, y_train_onehot, epochs=500, batch_size =␣
↪batch_s, validation_data=(X_val, y_val_onehot), callbacks = [callback])
```

[9]:
```
def plot(x):

    loss = model_history.history.copy()
    loss.pop('accuracy')
    loss.pop('val_accuracy')
    acc = model_history.history.copy()
    acc.pop('loss')
    acc.pop('val_loss')

    pd.DataFrame(loss).plot(figsize=(8,5))
    plt.grid(True)
    plt.show()

    pd.DataFrame(acc).plot(figsize=(8,5))
    plt.grid(True)
    plt.ylim(0,1)
    plt.show()
```

[10]:
```
def test_evaluate(x):

    test_loss = x.evaluate(X_test_normal, y_test_onehot)[0]
    test_accuracy = x.evaluate(X_test_normal, y_test_onehot)[1]
    y_prob = x.predict(X_test_normal)
    y_pred = y_prob.argmax(axis=-1)
    cnf = confusion_matrix(y_test, y_pred)
    prf = precision_recall_fscore_support(y_test, y_pred, average = 'macro')


    print('test loss: ' + str(test_loss) + '\n\ntest accuracy: ' +␣
↪str(test_accuracy) + '\n\nconfusion matrix: \n' + str(cnf) + '\n\nf1-score: '␣
↪+ str(prf[2]) + '\n\nrecall: ' + str(prf[1]) + '\n\nprecision: ' + str(prf[0]))
```

```
[11]: def model_architecture(x):
    return tf.keras.utils.plot_model(
    x,
    show_shapes=True,
    show_dtype=True,
    show_layer_names=False,
    rankdir="LR",
    expand_nested=True,
    dpi=96,
    layer_range=None,
    show_layer_activations=True,
)
```

## 1.1 Convolutional Layers

model with solely convolution layers:

```
[12]: model = keras.models.Sequential()
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), strides=1, ␣
 ↪activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), strides=1, ␣
 ↪kernel_initializer='he_uniform', activation='relu', padding='same'))

model.add(layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, ␣
 ↪activation='relu', padding='same'))
model.add(layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, ␣
 ↪activation='relu', padding='same'))

model.add(layers.Conv2D(filters=64, kernel_size=(3,3), strides=1,␣
 ↪activation='relu', padding='same'))
model.add(layers.Conv2D(filters=64, kernel_size=(3,3), strides=1,␣
 ↪activation='relu', padding='same'))

model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(keras.layers.Dense(10, activation = 'softmax'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 16)        448

 conv2d_1 (Conv2D)           (None, 32, 32, 16)        2320
```

```
 conv2d_2 (Conv2D)           (None, 32, 32, 32)        4640

 conv2d_3 (Conv2D)           (None, 32, 32, 32)        9248

 conv2d_4 (Conv2D)           (None, 32, 32, 64)        18496

 conv2d_5 (Conv2D)           (None, 32, 32, 64)        36928

 flatten (Flatten)           (None, 65536)             0

 dense (Dense)               (None, 64)                4194368

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 4,267,098
Trainable params: 4,267,098
Non-trainable params: 0
_________________________________________________________________
```

[13]:
```
model_compile('categorical_crossentropy', 'adam', 32)
plot(model_history.history)
test_evaluate(model)
model_architecture(model)
```

```
Epoch 1/500
1250/1250 [==============================] - 252s 201ms/step - loss: 1.5978 -
accuracy: 0.4211 - val_loss: 1.2737 - val_accuracy: 0.5414
...
Epoch 9/500
1250/1250 [==============================] - 255s 204ms/step - loss: 0.1287 -
accuracy: 0.9543 - val_loss: 2.1546 - val_accuracy: 0.6258
```

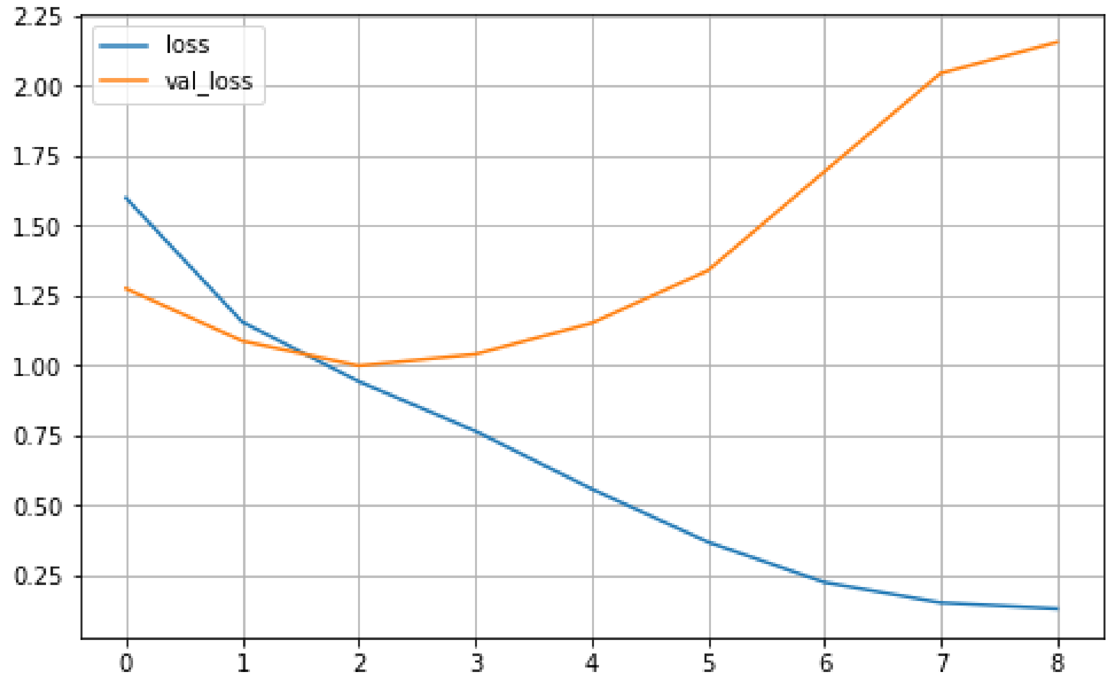


Figure 1: Loss and validation loss for model with convolutional layers

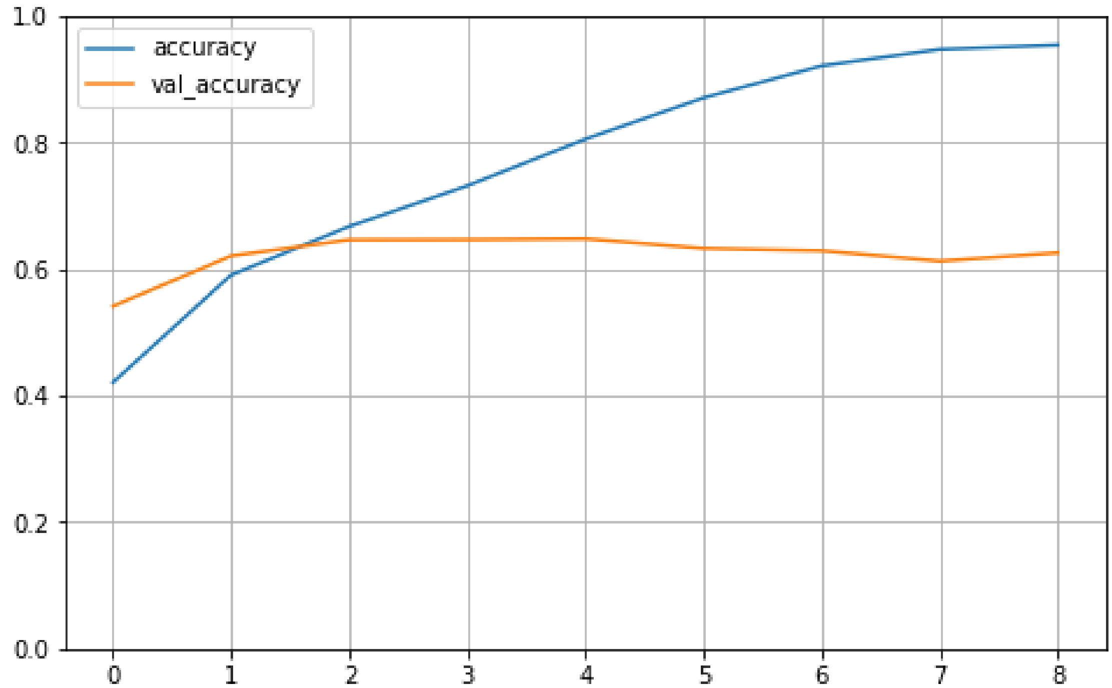


Figure 2: Accuracy and validation accuracy for model with convolutional layers

```
313/313 [==============================] - 11s 36ms/step - loss: 1.1573 -
accuracy: 0.6429
313/313 [==============================] - 11s 36ms/step - loss: 1.1573 -
accuracy: 0.6429
test loss: 1.1573235988616943

test accuracy: 0.6428999900817871

confusion matrix:
[[728  24  47  26  21   6   3  13  98  34]
 [ 41 800   4  13   6   3   3   7  36  87]
 [109  11 467  82 128  63  41  61  29   9]
 [ 41  31  60 464  80 146  50  73  33  22]
 [ 34   3  88  78 594  42  33 100  23   5]
 [ 25  11  45 182  57 535  23  98  19   5]
 [ 19  18  79  99  68  28 635  22  21  11]
 [ 19   3  29  36  73  56   1 753  13  17]
 [127  50   7  13  12   5   1  13 730  42]
 [ 64 109   5  20   7   6   0  23  43 723]]

f1-score: 0.6413638859917244
```

```
recall: 0.6428999999999999

precision: 0.6453779023882527
```

[13]:

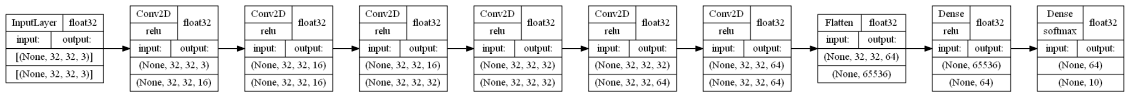

Figure 3: Architecture of model with convolutional layers

## 1.2 Batch Normalization and Pooling

**batch normalization:** Batch-Normalization (BN) is an algorithmic method which makes the training of Deep Neural Networks (DNN) faster and more stable. It consists of normalizing activation vectors from hidden layers using the first and the second statistical moments (mean and variance) of the current batch. This normalization step is applied right before (or right after) the nonlinear function.

**pooling:** Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

implementing model with batch normalization and pooling:

[14]:
```
model = keras.models.Sequential()
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), strides=1, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D(2, 2))

model.add(layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
```

```python
model.add(layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D(2, 2))

model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(BatchNormalization())
model.add(keras.layers.Dense(10, activation = 'softmax'))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 32, 32, 16)        448

 conv2d_7 (Conv2D)           (None, 32, 32, 16)        2320

 batch_normalization (BatchN  (None, 32, 32, 16)       64
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 16, 16, 16)       0
 )

 conv2d_8 (Conv2D)           (None, 16, 16, 32)        4640

 conv2d_9 (Conv2D)           (None, 16, 16, 32)        9248

 batch_normalization_1 (Batc  (None, 16, 16, 32)       128
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_10 (Conv2D)          (None, 8, 8, 64)          18496

 conv2d_11 (Conv2D)          (None, 8, 8, 64)          36928

 batch_normalization_2 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten_1 (Flatten)         (None, 1024)              0
```

```
 dense_2 (Dense)             (None, 64)                65600

 batch_normalization_3 (Batc  (None, 64)               256
 hNormalization)

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 139,034
Trainable params: 138,682
Non-trainable params: 352
_________________________________________________________________
```

[15]:
```
model_compile('categorical_crossentropy', 'adam', 32)
plot(model_history.history)
test_evaluate(model)
model_architecture(model)
```

```
Epoch 1/500
1250/1250 [==============================] - 74s 58ms/step - loss: 1.3735 -
accuracy: 0.5133 - val_loss: 1.2330 - val_accuracy: 0.5631
...
Epoch 17/500
1250/1250 [==============================] - 68s 54ms/step - loss: 0.1683 -
accuracy: 0.9406 - val_loss: 1.0347 - val_accuracy: 0.7447
```

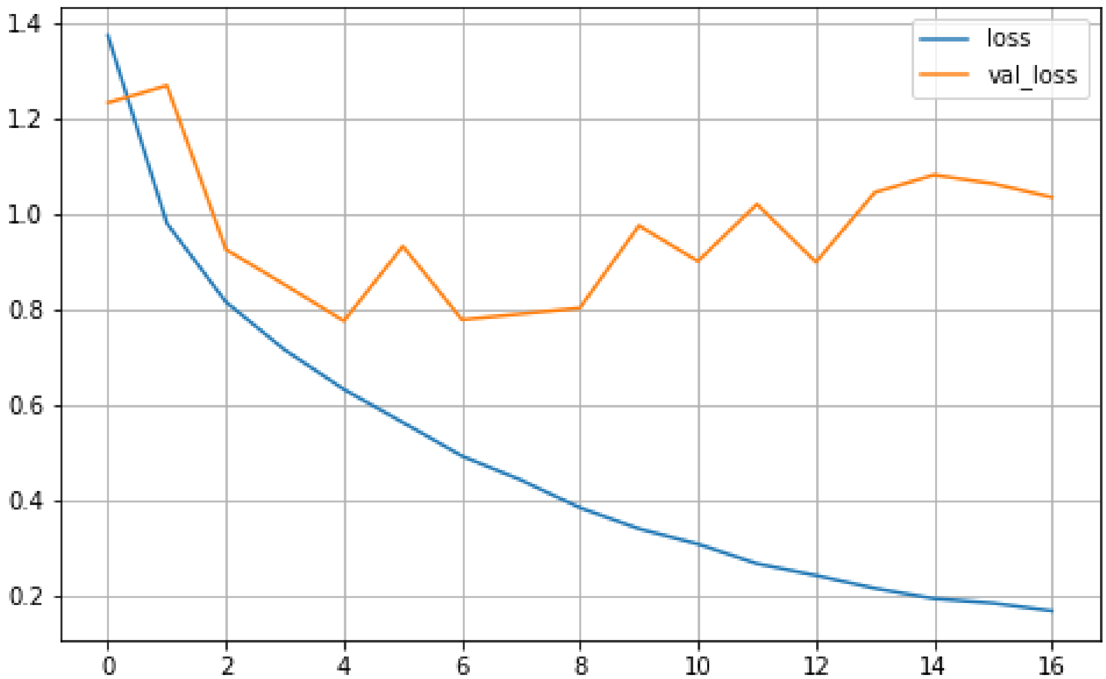


Figure 4: Loss and validation loss for CNN model with batch normalization and Max pooling

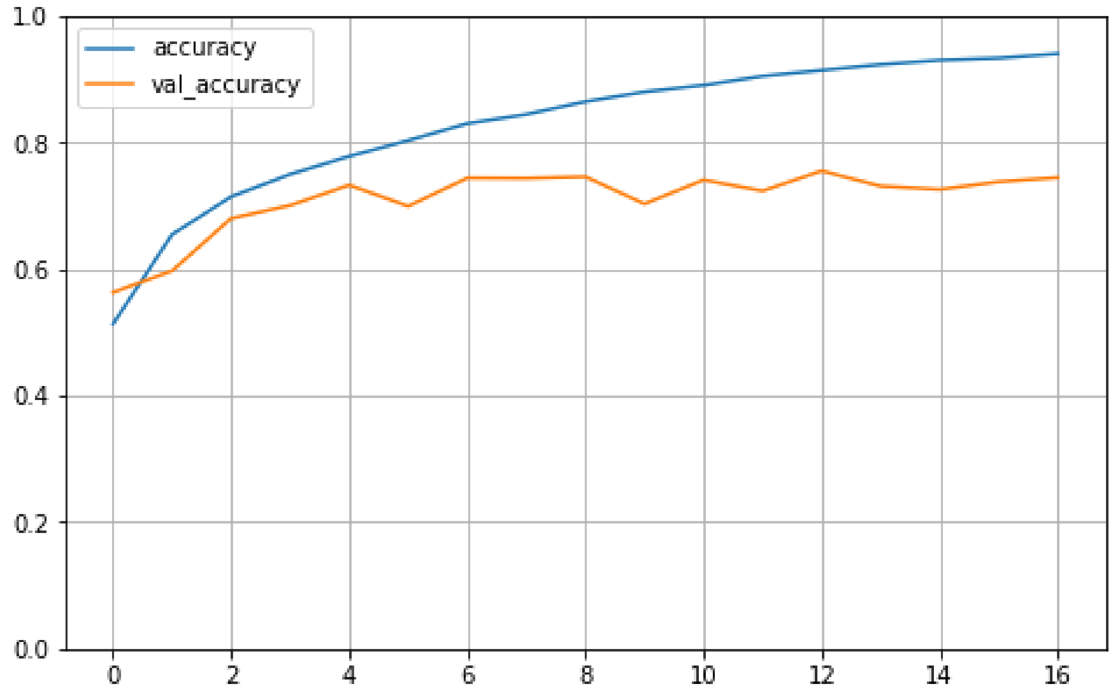


Figure 5: Accuracy and validation accuracy for CNN model with batch normalization and Max pooling

```
313/313 [==============================] - 4s 13ms/step - loss: 0.9102 -
accuracy: 0.7541
313/313 [==============================] - 4s 13ms/step - loss: 0.9102 -
accuracy: 0.7541
test loss: 0.9102030992507935

test accuracy: 0.7541000247001648

confusion matrix:
[[805  18  20  18  10  17  14  21  43  34]
 [ 15 877   1  17   3   7  13   6  17  44]
 [ 70   4 533  68  97  75  93  45   9   6]
 [ 18   3  30 547  58 210  78  41  11   4]
 [ 20   1  28  67 720  42  55  57   5   5]
 [ 11   1  22 144  40 713  24  42   3   0]
 [  5   2  15  56  26  32 853   6   3   2]
 [  7   0  11  29  37  62  10 831   5   8]
 [ 62  25   6  13   5   6   8   8 841  26]
 [ 23  75   5  14   3  18  12  18  11 821]]
```

```
f1-score: 0.7527738682082805

recall: 0.7541

precision: 0.7588957071962327
```

[15]:

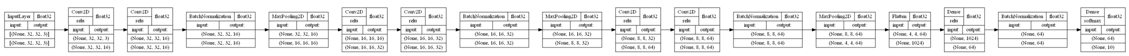

Figure 6: Architecture of CNN model with batch normalization and Max pooling

using batch normalization will increase our model's accuracy and decrease the loss for both train and validation data. also, using pooling significantly decreae training time by reducing parameters numbers.

## 1.3 Dropout

adding dropout to model:

[16]:
```
model = keras.models.Sequential()
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), strides=1, activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D(2, 2))
model.add(layers.Dropout(0.2))

model.add(layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D(2, 2))
model.add(layers.Dropout(0.2))

model.add(layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(layers.Conv2D(filters=64, kernel_size=(3,3), strides=1, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D(2, 2))
model.add(layers.Dropout(0.2))
```

```python
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(BatchNormalization())
model.add(layers.Dropout(0.2))
model.add(keras.layers.Dense(10, activation = 'softmax'))

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_12 (Conv2D)          (None, 32, 32, 16)        448

 conv2d_13 (Conv2D)          (None, 32, 32, 16)        2320

 batch_normalization_4 (Batc  (None, 32, 32, 16)       64
 hNormalization)

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

 dropout (Dropout)           (None, 16, 16, 16)        0

 conv2d_14 (Conv2D)          (None, 16, 16, 32)        4640

 conv2d_15 (Conv2D)          (None, 16, 16, 32)        9248

 batch_normalization_5 (Batc  (None, 16, 16, 32)       128
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 dropout_1 (Dropout)         (None, 8, 8, 32)          0

 conv2d_16 (Conv2D)          (None, 8, 8, 64)          18496

 conv2d_17 (Conv2D)          (None, 8, 8, 64)          36928

 batch_normalization_6 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 64)         0
 2D)
```

```
 dropout_2 (Dropout)         (None, 4, 4, 64)          0

 flatten_2 (Flatten)         (None, 1024)              0

 dense_4 (Dense)             (None, 64)                65600

 batch_normalization_7 (Batc  (None, 64)               256
 hNormalization)

 dropout_3 (Dropout)         (None, 64)                0

 dense_5 (Dense)             (None, 10)                650

=================================================================
Total params: 139,034
Trainable params: 138,682
Non-trainable params: 352
_________________________________________________________________
```

[17]:

```
model_compile('categorical_crossentropy', 'adam', 32)
plot(model_history.history)
test_evaluate(model)
model_architecture(model)
```

```
Epoch 1/500
1250/1250 [==============================] - 75s 60ms/step - loss: 1.6013 -
accuracy: 0.4311 - val_loss: 1.2272 - val_accuracy: 0.5618
...
Epoch 17/500
1250/1250 [==============================] - 87s 69ms/step - loss: 0.5453 -
accuracy: 0.8105 - val_loss: 0.6458 - val_accuracy: 0.7771
```

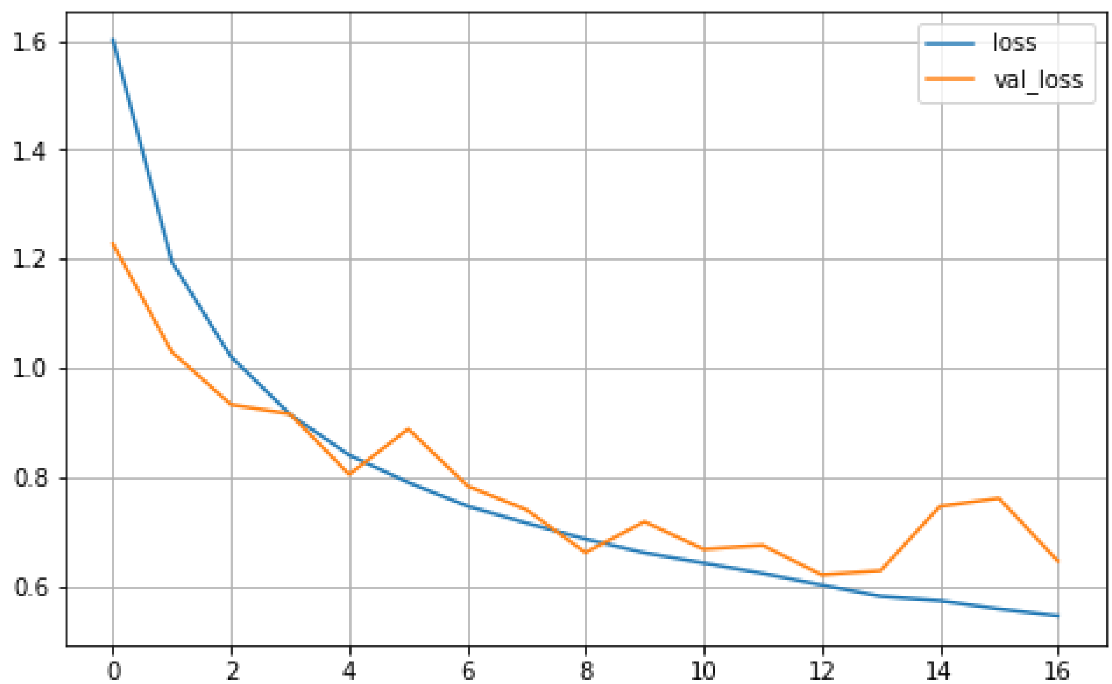


Figure 7: Loss and validation loss for CNN model with batch normalization, Max pooling and dropout

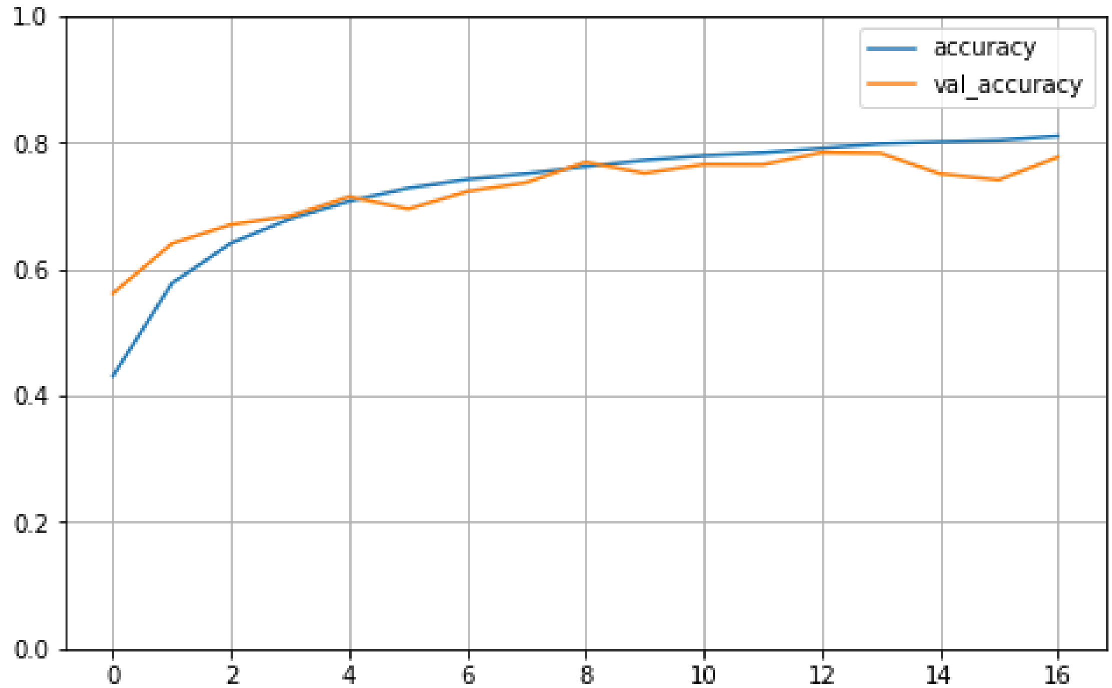


Figure 8: Accuracy and validation accuracy for CNN model with batch normalization, Max pooling and dropout

```
313/313 [==============================] - 4s 14ms/step - loss: 0.6282 -
accuracy: 0.7842
313/313 [==============================] - 4s 14ms/step - loss: 0.6282 -
accuracy: 0.7842
test loss: 0.6282299757003784

test accuracy: 0.7842000126838684

confusion matrix:
[[891   7  24   9   6   0   2  12  33  16]
 [ 31 854   4   3   1   2   5   1  29  70]
 [ 87   4 696  27  57  48  50  24   6   1]
 [ 42   3  90 544  38 175  60  28  13   7]
 [ 39   2  76  43 717  29  40  48   6   0]
 [ 25   2  76  78  31 733  21  29   5   0]
 [  9   1  47  41  22  18 846  11   5   0]
 [ 15   0  37  18  27  60   2 837   1   3]
 [ 80  10   8   9   2   3   2   3 869  14]
 [ 50  43   5   6   4   1   5   7  24 855]]
```

```
f1-score: 0.7832141967852961

recall: 0.7842

precision: 0.7874265187529428
```

[17]:

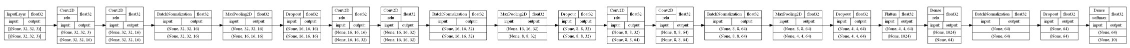

Figure 9: Architecture of CNN model with batch normalization, Max pooling and dropout

using dropout in our model increased the accuracy and decreased the loss value.

The main advantage of dropout method is that it prevents all neurons in a layer from synchronously optimizing their weights. This adaptation, made in random groups, prevents all the neurons from converging to the same goal, thus decorrelating the weights.

A second property discovered for the application of dropout is that the activations of the hidden units become sparse, which is also a desirable characteristic.

### 1.4 Early Stop

A significant challenge when training a machine learning model is deciding how many epochs to run. Too few epochs might not lead to model convergence, while too many epochs could lead to overfitting. Early stopping is an optimization technique used to reduce overfitting without compromising on model accuracy. The main idea behind early stopping is to stop training before a model starts to overfit.

**parameters of early stop:**

- **monitor:** Quantity to be monitored.
- **min_delta:** Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
- **patience:** Number of epochs with no improvement after which training will be stopped.
- **verbose:** Verbosity mode, 0 or 1. Mode 0 is silent, and mode 1 displays messages when the callback takes an action.
- **mode:** One of {"auto", "min", "max"}. In min mode, training will stop when the quantity monitored has stopped decreasing; in "max" mode it will stop when the quantity monitored has stopped increasing; in "auto" mode, the direction is automatically inferred from the name of the monitored quantity.
- **baseline:** Baseline value for the monitored quantity. Training will stop if the model doesn't show improvement over the baseline.
- **restore_best_weights:** Whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used. An epoch will be restored regardless of the performance relative to the baseline. If no epoch improves on baseline, training will run for patience epochs and restore weights from the best epoch in that set.

we already used early stop callback in our models.

# 2 Transfer Learning

Student IDs: 810100462 & 810100410

selected model: 2 -> SqueezeNet

## 2.1 Model Properties

**model architecture:** The SqueezeNet architecture is comprised of "squeeze" and "expand" layers. A squeeze convolutional layer has only $1 \times 1$ filters. These are fed into an expand layer that has a mix of $1 \times 1$ and $3 \times 3$ convolution filters.

The authors of the paper use the term "fire module" to describe a squeeze layer and an expand layer together.

An input image is first sent into a standalone convolutional layer. This layer is followed by 8 "fire modules" which are named "fire2-9", according to Strategy One above.

Following Strategy Two, the filters per fire module are increased with "simple bypass." Lastly, SqueezeNet performs max-pooling with a stride of 2 after layers conv1, fire4, fire8, and conv10. According to Strategy Three, pooling is given a relatively late placement, resulting in SqueezeNet with a "complex bypass".

Below are the details of other parameters used in the network: * The ReLU activation is applied between all the squeeze and expand layers inside the fire module. * Dropout layers are added to reduce overfitting, with a probability of 0.5 after the fire9 module. * There are no fully connected layers used in the network. This design choice was inspired by the Network In Netowork (NIN) architecture proposed by (Lin et al, 2013). * SqueezeNet was trained with a learning rate of 0.04, which is linearly decreased throughout the training process. * The batch size for training is 32, and the network used an Adam Optimizer.

**pros and cons:** SqueezeNet makes the deployment process easier due to its small size.

SqueezeNet has two disadvantages: low classification accuracy and high computational complexity.

**initial preprocess:** All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean and std.

## 2.2 Transfer Learning

**Transfer Learning** is a machine learning method where we reuse a pre-trained model as the starting point for a model on a new task.

To put it simply—a model trained on one task is repurposed on a second, related task as an optimization that allows rapid progress when modeling the second task.

By applying transfer learning to a new task, one can achieve significantly higher performance than training with only a small amount of data.

[1]:
```
import torch
```

[2]:
```
model = torch.hub.load('pytorch/vision:v0.10.0', 'squeezenet1_0',␣
↪pretrained=True)
model.eval()
```

```
Downloading: "https://github.com/pytorch/vision/archive/v0.10.0.zip" to
/root/.cache/torch/hub/v0.10.0.zip
Downloading: "https://download.pytorch.org/models/squeezenet1_0-b66bff10.pth" to
/root/.cache/torch/hub/checkpoints/squeezenet1_0-b66bff10.pth

  0%|          | 0.00/4.78M [00:00<?, ?B/s]
```

[2]:
```
SqueezeNet(
  (features): Sequential(
    (0): Conv2d(3, 96, kernel_size=(7, 7), stride=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (3): Fire(
      (squeeze): Conv2d(96, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (4): Fire(
      (squeeze): Conv2d(128, 16, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (5): Fire(
      (squeeze): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (6): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (7): Fire(
      (squeeze): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
```

```
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (8): Fire(
      (squeeze): Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (9): Fire(
      (squeeze): Conv2d(384, 48, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(48, 192, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (10): Fire(
      (squeeze): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
    (11): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=True)
    (12): Fire(
      (squeeze): Conv2d(512, 64, kernel_size=(1, 1), stride=(1, 1))
      (squeeze_activation): ReLU(inplace=True)
      (expand1x1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
      (expand1x1_activation): ReLU(inplace=True)
      (expand3x3): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (expand3x3_activation): ReLU(inplace=True)
    )
  )
  (classifier): Sequential(
```

```
    (0): Dropout(p=0.5, inplace=False)
    (1): Conv2d(512, 1000, kernel_size=(1, 1), stride=(1, 1))
    (2): ReLU(inplace=True)
    (3): AdaptiveAvgPool2d(output_size=(1, 1))
  )
)
```

## 2.3 Implementation

our model can classify animals.

[7]:
```python
import urllib
url, filename = ("https://github.com/pytorch/hub/raw/master/images/dog.jpg",␣
 ↪"dog.jpg")
try: urllib.URLopener().retrieve(url, filename)
except: urllib.request.urlretrieve(url, filename)
```

[5]:
```python
from PIL import Image
import matplotlib.pyplot as plt
dog = Image.open('dog.jpg')
imgplot = plt.imshow(dog)
```

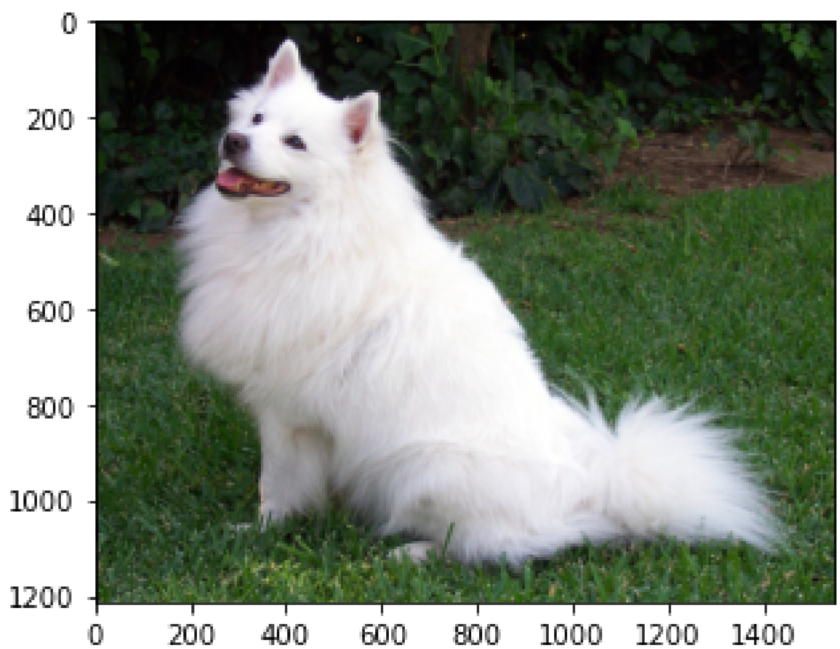


Figure 10: a custom photo of recognizable categories

[8]:
```python
from PIL import Image
from torchvision import transforms

input_image = Image.open(filename)
preprocess = transforms.Compose([
    transforms.Resize(299),
    transforms.CenterCrop(299),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the
 ↪model

# move the input and model to GPU for speed if available
if torch.cuda.is_available():
    input_batch = input_batch.to('cuda')
    model.to('cuda')

with torch.no_grad():
  output = model(input_batch)


probabilities = torch.nn.functional.softmax(output[0], dim=0)
```

[9]:
```python
!wget https://raw.githubusercontent.com/pytorch/hub/master/imagenet_classes.txt

# Read the categories
with open("imagenet_classes.txt", "r") as f:
    categories = [s.strip() for s in f.readlines()]
```

```
--2022-04-15 09:09:56--
https://raw.githubusercontent.com/pytorch/hub/master/imagenet_classes.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10472 (10K) [text/plain]
Saving to: ‘imagenet_classes.txt.1’

imagenet_classes.tx 100%[===================>]  10.23K  --.-KB/s    in 0s

2022-04-15 09:09:56 (74.9 MB/s) - ‘imagenet_classes.txt.1’ saved [10472/10472]
```

[10]:
```
_, indices = torch.topk(probabilities, 3)
for index in indices:
    print('Object {} with probability {}'.format(categories[index],␣
 ↪probabilities[index]))
```

```
Object Samoyed with probability 0.8526738882064819
Object Pomeranian with probability 0.03457362577319145
Object West Highland white terrier with probability 0.02477547898888588
```

# 3 Segmentation

**FCN:** * remove any dense(fully connected) layers and only use convolution layers. * downsampling and then do upsampling to get pixel wise prediction to do segmentation. * also introduce lateral connections, which is combining downsampling feature map and upasmpled feature map on same level. * when combining lateral connections, the paper simply adds the two. * upsampling is a layer which is initialized as bilinear interpolation but allow the params to be learned.

**UNet:** * down and upsampling architecture. * only the final upsampled feature map is utilized. * designed for segmentation.

import essential libraries:

[1]:
```
import tensorflow_datasets as tfds
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers
```

import 'oxford_iiit_pet' dataset for training:

[2]:
```
dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)
```

resize the images and masks to 128x128:

[3]:
```
def resize(input_image, input_mask):
   input_image = tf.image.resize(input_image, (128, 128), method="nearest")
   input_mask = tf.image.resize(input_mask, (128, 128), method="nearest")
   return input_image, input_mask
```

create a function to augment the dataset by flipping them horizontally:

[4]:
```
def augment(input_image, input_mask):
   if tf.random.uniform(()) > 0.5:
       # Random flipping of the image and mask
       input_image = tf.image.flip_left_right(input_image)
       input_mask = tf.image.flip_left_right(input_mask)
   return input_image, input_mask
```

normalize the dataset by scaling the images to the range of [-1, 1] and decreasing the image mask by 1:

[5]:
```
def normalize(input_image, input_mask):
   input_image = tf.cast(input_image, tf.float32) / 255.0
   input_mask -= 1
   return input_image, input_mask
```

functions to preprocess the training and test datasets:

[6]:
```
def load_image_train(datapoint):
    input_image = datapoint["image"]
    input_mask = datapoint["segmentation_mask"]
    input_image, input_mask = resize(input_image, input_mask)
    input_image, input_mask = augment(input_image, input_mask)
    input_image, input_mask = normalize(input_image, input_mask)
    return input_image, input_mask
def load_image_test(datapoint):
    input_image = datapoint["image"]
    input_mask = datapoint["segmentation_mask"]
    input_image, input_mask = resize(input_image, input_mask)
    input_image, input_mask = normalize(input_image, input_mask)
    return input_image, input_mask
```

build an input pipeline:

[7]:
```
train_dataset = dataset["train"].map(load_image_train, num_parallel_calls=tf.
 ↪data.AUTOTUNE)
test_dataset = dataset["test"].map(load_image_test, num_parallel_calls=tf.data.
 ↪AUTOTUNE)
```

making dataset ready for training:

[8]:
```
BATCH_SIZE = 64
BUFFER_SIZE = 1000
train_batches = train_dataset.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).
 ↪repeat()
train_batches = train_batches.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
validation_batches = test_dataset.take(3000).batch(BATCH_SIZE)
test_batches = test_dataset.skip(3000).take(669).batch(BATCH_SIZE)
```

visualize a random sample image and its mask from the training dataset:

[14]:
```
def display(display_list):
 plt.figure(figsize=(15, 15))
 title = ["Input Image", "True Mask", "Predicted Mask"]
 for i in range(len(display_list)):
   plt.subplot(1, len(display_list), i+1)
   plt.title(title[i])
   plt.imshow(tf.keras.utils.array_to_img(display_list[i]))
   plt.axis("off")
 plt.show()
sample_batch = next(iter(train_batches))
random_index = np.random.choice(sample_batch[0].shape[0])
sample_image, sample_mask = sample_batch[0][random_index],␣
 ↪sample_batch[1][random_index]
display([sample_image, sample_mask])
```


Figure 11: Sample input image and mask

building blocks:

[15]:
```
def double_conv_block(x, n_filters):
    # Conv2D then ReLU activation
    x = layers.Conv2D(n_filters, 3, padding = "same", activation = "relu",␣
↪kernel_initializer = "he_normal")(x)
    # Conv2D then ReLU activation
    x = layers.Conv2D(n_filters, 3, padding = "same", activation = "relu",␣
↪kernel_initializer = "he_normal")(x)
    return x
```

[16]:
```
def downsample_block(x, n_filters):
    f = double_conv_block(x, n_filters)
    p = layers.MaxPool2D(2)(f)
    p = layers.Dropout(0.3)(p)
    return f, p
```

[17]:
```
def upsample_block(x, conv_features, n_filters):
    # upsample
    x = layers.Conv2DTranspose(n_filters, 3, 2, padding="same")(x)
    # concatenate
    x = layers.concatenate([x, conv_features])
    # dropout
    x = layers.Dropout(0.3)(x)
    # Conv2D twice with ReLU activation
```

```
    x = double_conv_block(x, n_filters)
    return x
```

Unet model:

[13]:
```
def build_unet_model():
    # inputs
    inputs = layers.Input(shape=(128,128,3))

    # encoder: contracting path - downsample
    # 1 - downsample
    f1, p1 = downsample_block(inputs, 64)
    # 2 - downsample
    f2, p2 = downsample_block(p1, 128)
    # 3 - downsample
    f3, p3 = downsample_block(p2, 256)
    # 4 - downsample
    f4, p4 = downsample_block(p3, 512)

    # 5 - bottleneck
    bottleneck = double_conv_block(p4, 1024)

    # decoder: expanding path - upsample
    # 6 - upsample
    u6 = upsample_block(bottleneck, f4, 512)
    # 7 - upsample
    u7 = upsample_block(u6, f3, 256)
    # 8 - upsample
    u8 = upsample_block(u7, f2, 128)
    # 9 - upsample
    u9 = upsample_block(u8, f1, 64)

    # outputs
    outputs = layers.Conv2D(3, 1, padding="same", activation = "softmax")(u9)

    # unet model with Keras Functional API
    unet_model = tf.keras.Model(inputs, outputs, name="U-Net")

    return unet_model
```

[14]:
```
unet_model = build_unet_model()
```

compile and train Unet:

[15]:
```
unet_model.compile(optimizer=tf.keras.optimizers.Adam(),
                  loss="sparse_categorical_crossentropy",
                  metrics="accuracy")
```

[16]:
```python
NUM_EPOCHS = 10

TRAIN_LENGTH = info.splits["train"].num_examples
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE

VAL_SUBSPLITS = 5
TEST_LENTH = info.splits["test"].num_examples
VALIDATION_STEPS = TEST_LENTH // BATCH_SIZE // VAL_SUBSPLITS

model_history = unet_model.fit(train_batches,
                               epochs=NUM_EPOCHS,
                               steps_per_epoch=STEPS_PER_EPOCH,
                               validation_steps=VALIDATION_STEPS,
                               validation_data=test_batches)
```

```
Epoch 1/10
57/57 [==============================] - 162s 2s/step - loss: 2.1894 - accuracy:
0.5593 - val_loss: 0.9181 - val_accuracy: 0.5864
Epoch 2/10
57/57 [==============================] - 130s 2s/step - loss: 0.8967 - accuracy:
0.6057 - val_loss: 0.8755 - val_accuracy: 0.5864
Epoch 3/10
57/57 [==============================] - 120s 2s/step - loss: 0.7823 - accuracy:
0.6693 - val_loss: 0.7104 - val_accuracy: 0.7190
Epoch 4/10
57/57 [==============================] - 121s 2s/step - loss: 0.6857 - accuracy:
0.7217 - val_loss: 0.6645 - val_accuracy: 0.7264
Epoch 5/10
57/57 [==============================] - 121s 2s/step - loss: 0.6481 - accuracy:
0.7387 - val_loss: 0.6407 - val_accuracy: 0.7337
Epoch 6/10
57/57 [==============================] - 120s 2s/step - loss: 0.6147 - accuracy:
0.7520 - val_loss: 0.6235 - val_accuracy: 0.7424
Epoch 7/10
57/57 [==============================] - 121s 2s/step - loss: 0.5955 - accuracy:
0.7619 - val_loss: 0.6218 - val_accuracy: 0.7432
Epoch 8/10
57/57 [==============================] - 121s 2s/step - loss: 0.5597 - accuracy:
0.7775 - val_loss: 0.5804 - val_accuracy: 0.7636
Epoch 9/10
57/57 [==============================] - 121s 2s/step - loss: 0.5139 - accuracy:
0.7976 - val_loss: 0.4846 - val_accuracy: 0.8100
Epoch 10/10
57/57 [==============================] - 121s 2s/step - loss: 0.4784 - accuracy:
0.8130 - val_loss: 0.4545 - val_accuracy: 0.8228
```

Unet prediction:

[54]:
```python
def create_mask(pred_mask):
 pred_mask = tf.argmax(pred_mask, axis=-1)
 pred_mask = pred_mask[..., tf.newaxis]
 return pred_mask[0]

def show_predictions_unet(dataset=None, num=1):
 if dataset:
   for image, mask in dataset.take(num):
     pred_mask = unet_model.predict(image)
     display([image[0], mask[0], create_mask(pred_mask)])
 else:
   display([sample_image, sample_mask,
             create_mask(model.predict(sample_image[tf.newaxis, ...]))])

count = 0
for i in test_batches:
   count +=1
print("number of batches:", count)
```

```
number of batches: 11
```

[18]:
```python
show_predictions_unet(test_batches)
```


Figure 12: U-Net model prediction on test sample

fcn model:

[48]:
```python
def build_fcn_model():
   # inputs
   inputs = layers.Input(shape=(128,128,3))

   # encoder: contracting path - downsample
```

```
    # 1 - downsample
    f1, p1 = downsample_block(inputs, 128)
    # 2 - downsample
    f2, p2 = downsample_block(p1, 128)
    # 3 - downsample
    f3, p3 = downsample_block(p2, 128)
    # 4 - downsample
    f4, p4 = downsample_block(p3, 128)

    # 5 - bottleneck
    bottleneck = double_conv_block(p4, 512)

    # decoder: expanding path - upsample
    # 6 - upsample
    u6 = upsample_block(bottleneck, f4, 128)
    # 7 - upsample
    u7 = upsample_block(u6, f3, 128)
    # 8 - upsample
    u8 = upsample_block(u7, f2, 128)
    # 9 - upsample
    u9 = upsample_block(u8, f1, 128)

    # outputs
    outputs = layers.Conv2D(3, 1, padding="same", activation = "softmax")(u9)

    # fcn model with Keras Functional API
    fcn_model = tf.keras.Model(inputs, outputs, name="FCN")

    return fcn_model
```

[49]:
```
fcn_model = build_fcn_model()
```

compile and train fcn model:

[50]:
```
fcn_model.compile(optimizer=tf.keras.optimizers.Adam(),
                  loss="sparse_categorical_crossentropy",
                  metrics="accuracy")
```

[51]:
```
NUM_EPOCHS = 10

TRAIN_LENGTH = info.splits["train"].num_examples
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE

VAL_SUBSPLITS = 5
TEST_LENTH = info.splits["test"].num_examples
VALIDATION_STEPS = TEST_LENTH // BATCH_SIZE // VAL_SUBSPLITS
```

```
model_history = fcn_model.fit(train_batches,
                              epochs=NUM_EPOCHS,
                              steps_per_epoch=STEPS_PER_EPOCH,
                              validation_steps=VALIDATION_STEPS,
                              validation_data=test_batches)
```

```
Epoch 1/10
57/57 [==============================] - 278s 5s/step - loss: 0.9992 - accuracy:
0.5656 - val_loss: 0.8899 - val_accuracy: 0.5864
Epoch 2/10
57/57 [==============================] - 180s 3s/step - loss: 0.8151 - accuracy:
0.6151 - val_loss: 0.7467 - val_accuracy: 0.6691
Epoch 3/10
57/57 [==============================] - 173s 3s/step - loss: 0.6820 - accuracy:
0.7131 - val_loss: 0.6134 - val_accuracy: 0.7567
Epoch 4/10
57/57 [==============================] - 173s 3s/step - loss: 0.6216 - accuracy:
0.7513 - val_loss: 0.5687 - val_accuracy: 0.7751
Epoch 5/10
57/57 [==============================] - 173s 3s/step - loss: 0.5564 - accuracy:
0.7818 - val_loss: 0.5354 - val_accuracy: 0.7920
Epoch 6/10
57/57 [==============================] - 173s 3s/step - loss: 0.5262 - accuracy:
0.7964 - val_loss: 0.5607 - val_accuracy: 0.7842
Epoch 7/10
57/57 [==============================] - 173s 3s/step - loss: 0.4887 - accuracy:
0.8114 - val_loss: 0.4386 - val_accuracy: 0.8287
Epoch 8/10
57/57 [==============================] - 173s 3s/step - loss: 0.4328 - accuracy:
0.8335 - val_loss: 0.4242 - val_accuracy: 0.8349
Epoch 9/10
57/57 [==============================] - 173s 3s/step - loss: 0.4170 - accuracy:
0.8399 - val_loss: 0.3844 - val_accuracy: 0.8511
Epoch 10/10
57/57 [==============================] - 172s 3s/step - loss: 0.4100 - accuracy:
0.8420 - val_loss: 0.3853 - val_accuracy: 0.8527
```

prediction by fcn:

[52]:
```
def show_predictions_fcn(dataset=None, num=1):
  if dataset:
    for image, mask in dataset.take(num):
      pred_mask = fcn_model.predict(image)
      display([image[0], mask[0], create_mask(pred_mask)])
  else:
    display([sample_image, sample_mask,
             create_mask(model.predict(sample_image[tf.newaxis, ...]))])
```

[55]:
```
show_predictions_fcn(test_batches)
```


Figure 13: FCN model prediction on test sample

the accuracy and validation accuracy for fcn model is greater than model for nearly 3 percent.

# 4 Object Detection

## 4.1 YOLO v1 and v2

**The changes from YOLO to YOLO v2:**

- **Batch Normalization:** YOLO V2 normalise the input layer by altering slightly and scaling the activations. Batch normalization decreases the shift in unit value in the hidden layer and by doing so it improves the stability of the neural network. By adding batch normalization to convolutional layers in the architecture MAP (mean average precision) has been improved by 2%. It also helped the model regularise and overfitting has been reduced overall.

- **Higher Resolution Classifier:** the input size in YOLO v2 has been increased from 224*224 to 448*448. The increase in the input size of the image has improved the MAP (mean average precision) upto 4%. This increase in input size is been applied while training the YOLO v2 architecture DarkNet 19 on ImageNet dataset.

- **Anchor Boxes:** one of the most notable changes which can visible in YOLO v2 is the introduction the anchor boxes. YOLO v2 does classification and prediction in a single framework. These anchor boxes are responsible for predicting bounding box and this anchor boxes are designed for a given dataset by using clustering(k-means clustering).

- **Fine-Grained Features:** one of the main issued that has to be addressed in the YOLO v1 is that detection of smaller objects on the image. This has been resolved in the YOLO v2 divides the image into 13*13 grid cells which is smaller when compared to its previous version. This enables the yolo v2 to identify or localize the smaller objects in the image and also effective with the larger objects.*

- **Multi-Scale Training:** on YOLO v1 has a weakness detecting objects with different input sizes which says that if YOLO is trained with small images of a particular object it has issues detecting the same object on image of bigger size. This has been resolved to a great extent in YOLO v2 where it is trained with random images with different dimensions range between 320*320 to 608*608. This allows the network to learn and predict the objects from various input dimensions with accuracy.

- **Darknet 19:** YOLO v2 uses Darknet 19 architecture with 19 convolutional layers and 5 max pooling layers and a softmax layer for classification objects. Darknet is a neural network framework written in Clanguage and CUDA. It's really fast in object detection which is very important for predicting in real-time.

With the advancements in several categories in YOLO v2 is better, faster, and stronger. With Multi-Scale Training now the network is able to detect and classify objects with different configurations and dimensions. YOLO v2 has seen a great improvement in detecting smaller objects with much more accuracy which it lacked in its predecessor version.

## 4.2 YOLO 4 & YOLO 5:

As a modified version of YOLOv3, YOLO4. uses Cross Stage Partial Network (CSPNet) in Darknet, creating a new feature extractor backbone called CSPDarknet53. The convolution architecture is based on modified DenseNet. It transfers a copy of feature map from the base layer to the next layer through dense block. The advantages of using DenseNet include the diminishing gradient vanishing problems, boosting backpropagation, removal of the computational bottleneck, and improved learning. Neck is composed of spatial pyramid pooling (SPP) layer and PANet path aggregation. SPP layer and PANet path aggregation are used for feature aggregation to improve the receptive field and short out important features from the backbone. In addition, the head is composed of YOLO layer. First, the image is fed to CSPDarknet53 for feature extraction and then fed to path aggregation network PANet for fusion. Finally, YOLO layer generates the results, similar to YOLOv3 YOLOv4 uses bag of freebies and bag of specials to improve the algorithm performance. Bag of freebies includes Complete IOU loss (CIOU), drop block regularization and different augmentation techniques. Bags of specials includes mish activation, Diou-NMS and modified the path aggregation networks.

However, YOLOv5 is different from the previous releases. It utilizes PyTorch instead of Darknet. It utilizes CSPDarknet53 as backbone. This backbone solves the repetitive gradient information in large backbones and integrates gradient change into feature map that reduces the inference speed, increases accuracy, and reduces the model size by decreasing the parameters. It uses path aggregation network (PANet) as neck to boost the information flow. PANet adopts a new feature pyramid network (FPN) that includes several bottom ups and top down layers. This improves the propagation of low level features in the model. PANet improves the localization in lower layers, which enhances the localization accuracy of the object. In addition, the head in YOLOv5 is the same as YOLOv4 and YOLOv3 which generates three different output of feature maps to achieve multi scale prediction. It also helps to enhance the prediction of small to large objects efficiently in the model. The image is fed to CSPDarknet53 for feature extraction and again fed to PANet for feature fusion. Finally, the YOLO layer generates the results.

### 4.3 Two-stage vs One-stage Detectors:

In two-stage Detectors, first, the model proposes a set of regions of interests by select search or regional proposal network. The proposed regions are sparse as the potential bounding box candidates can be infinite. Then a classifier only processes the region candidates. But in one-stage Detectors Single convolutional network predicts the bounding boxes and the class probabilities for these boxes.

**Example of two-stage Detectors:** R-CNN family

**Examples of one-stage Detectors:** YOLO & SSD

### 4.4 YOLO v5 implementation:

Install Dependencies:

[ ]:
```
# clone YOLOv5 repository
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5
!git reset --hard 886f1c03d839575afecb059accf74296fad395b6
```

```
fatal: destination path 'yolov5' already exists and is not an empty directory.
/content/yolov5
HEAD is now at 886f1c0 DDP after autoanchor reorder (#2421)
```

```
[ ]: # install dependencies as necessary
!pip install -qr requirements.txt  # install dependencies (ignore errors)
import torch

from IPython.display import Image, clear_output  # to display images
from utils.google_utils import gdrive_download  # to download models/datasets

# clear_output()
print('Setup complete. Using torch %s %s' % (torch.__version__, torch.cuda.
 ↪get_device_properties(0) if torch.cuda.is_available() else 'CPU'))
```

```
Setup complete. Using torch 1.10.0+cu111 _CudaDeviceProperties(name='Tesla K80',
major=3, minor=7, total_memory=11441MB, multi_processor_count=13)
```

```
[ ]: %cd /content
!curl -L "https://app.roboflow.com/ds/F2Eo4hPmah?key=8ZdNTyoAP8" > roboflow.zip;␣
 ↪unzip roboflow.zip; rm roboflow.zip
```

```
/content
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   896  100   896    0     0    746      0  0:00:01  0:00:01 --:--:--   746
100 5698k  100 5698k    0     0  2279k      0  0:00:02  0:00:02 --:--:--  245M
Archive:  roboflow.zip
 extracting: README.roboflow.txt
 extracting: data.yaml
   creating: test/
   creating: test/images/
 extracting: test/images/cap_100_jpg.rf.04d2e5f139a204fe67ba4a6e318b9352.jpg
 ...
 extracting: valid/labels/cap_91_jpg.rf.f4c440ea8c8a9f0304073f9b6336bba2.txt
```

```
[ ]: %cat data.yaml
```

```
train: ../train/images
val: ../valid/images

nc: 1
names: ['Caps']
```

```
[ ]: import yaml
with open("data.yaml", 'r') as stream:
    num_classes = str(yaml.safe_load(stream)['nc'])
```

```
[ ]: %cat /content/yolov5/models/yolov5s.yaml
```

```
# parameters
nc: 80  # number of classes
```

```yaml
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, C3, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, C3, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, C3, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 3, C3, [1024, False]],  # 9
  ]

# YOLOv5 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, C3, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, C3, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, C3, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],
   [[-1, 10], 1, Concat, [1]],  # cat head P5
   [-1, 3, C3, [1024, False]],  # 23 (P5/32-large)

   [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
  ]
```

[ ]:
```python
from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))
```

[ ]:
```
%%writetemplate /content/yolov5/models/custom_yolov5s.yaml

# parameters
nc: {num_classes}  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 3, BottleneckCSP, [1024, False]],  # 9
  ]

# YOLOv5 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, BottleneckCSP, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, BottleneckCSP, [256, False]],  # 17 (P3/8-small)
```

```
  [-1, 1, Conv, [256, 3, 2]],
  [[-1, 14], 1, Concat, [1]],  # cat head P4
  [-1, 3, BottleneckCSP, [512, False]],  # 20 (P4/16-medium)

  [-1, 1, Conv, [512, 3, 2]],
  [[-1, 10], 1, Concat, [1]],  # cat head P5
  [-1, 3, BottleneckCSP, [1024, False]],  # 23 (P5/32-large)

  [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
 ]
```

Train Custom YOLOv5 Detector:

[ ]:
```
%%time
%cd /content/yolov5/
!python train.py --img 416 --batch 16 --epochs 100 --data '../data.yaml' --cfg ./
 ↪models/custom_yolov5s.yaml --weights '' --name yolov5s_results  --cache
```

```
/content/yolov5
github: WARNING: code is out of date by 1006 commits. Use 'git
pull' to update or 'git clone https://github.com/ultralytics/yolov5' to download
latest.
YOLOv5 v4.0-126-g886f1c0 torch 1.10.0+cu111 CUDA:0 (Tesla K80, 11441.1875MB)

Namespace(adam=False, batch_size=16, bucket='', cache_images=True,
cfg='./models/custom_yolov5s.yaml', data='../data.yaml', device='', entity=None,
epochs=100, evolve=False, exist_ok=False, global_rank=-1,
hyp='data/hyp.scratch.yaml', image_weights=False, img_size=[416, 416],
linear_lr=False, local_rank=-1, log_artifacts=False, log_imgs=16,
multi_scale=False, name='yolov5s_results', noautoanchor=False, nosave=False,
notest=False, project='runs/train', quad=False, rect=False, resume=False,
save_dir='runs/train/yolov5s_results', single_cls=False, sync_bn=False,
total_batch_size=16, weights='', workers=8, world_size=1)
wandb: Install Weights & Biases for YOLOv5 logging with 'pip
install wandb' (recommended)
Start Tensorboard with "tensorboard --logdir runs/train", view at
http://localhost:6006/
hyperparameters: lr0=0.01, lrf=0.2, momentum=0.937,
weight_decay=0.0005, warmup_epochs=3.0, warmup_momentum=0.8, warmup_bias_lr=0.1,
box=0.05, cls=0.5, cls_pw=1.0, obj=1.0, obj_pw=1.0, iou_t=0.2, anchor_t=4.0,
fl_gamma=0.0, hsv_h=0.015, hsv_s=0.7, hsv_v=0.4, degrees=0.0, translate=0.1,
scale=0.5, shear=0.0, perspective=0.0, flipud=0.0, fliplr=0.5, mosaic=1.0,
mixup=0.0

                 from  n    params  module
arguments
  0                -1  1      3520  models.common.Focus                     [3,
32, 3]
```

```
  1                -1  1     18560  models.common.Conv                      [32,
64, 3, 2]
  2                -1  1     19904  models.common.BottleneckCSP             [64,
64, 1]
  3                -1  1     73984  models.common.Conv                      [64,
128, 3, 2]
  4                -1  1    161152  models.common.BottleneckCSP
[128, 128, 3]
  5                -1  1    295424  models.common.Conv
[128, 256, 3, 2]
  6                -1  1    641792  models.common.BottleneckCSP
[256, 256, 3]
  7                -1  1   1180672  models.common.Conv
[256, 512, 3, 2]
  8                -1  1    656896  models.common.SPP
[512, 512, [5, 9, 13]]
  9                -1  1   1248768  models.common.BottleneckCSP
[512, 512, 1, False]
 10                -1  1    131584  models.common.Conv
[512, 256, 1, 1]
 11                -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 12           [-1, 6]  1         0  models.common.Concat                    [1]
 13                -1  1    378624  models.common.BottleneckCSP
[512, 256, 1, False]
 14                -1  1     33024  models.common.Conv
[256, 128, 1, 1]
 15                -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 16           [-1, 4]  1         0  models.common.Concat                    [1]
 17                -1  1     95104  models.common.BottleneckCSP
[256, 128, 1, False]
 18                -1  1    147712  models.common.Conv
[128, 128, 3, 2]
 19          [-1, 14]  1         0  models.common.Concat                    [1]
 20                -1  1    313088  models.common.BottleneckCSP
[256, 256, 1, False]
 21                -1  1    590336  models.common.Conv
[256, 256, 3, 2]
 22          [-1, 10]  1         0  models.common.Concat                    [1]
 23                -1  1   1248768  models.common.BottleneckCSP
[512, 512, 1, False]
 24      [17, 20, 23]  1     16182  models.yolo.Detect                      [1,
[[10, 13, 16, 30, 33, 23], [30, 61, 62, 45, 59, 119], [116, 90, 156, 198, 373,
326]], [128, 256, 512]]
/usr/local/lib/python3.7/dist-packages/torch/functional.py:445: UserWarning:
torch.meshgrid: in an upcoming release, it will be required to pass the indexing
argument. (Triggered internally at
```

```
../aten/src/ATen/native/TensorShape.cpp:2157.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
Model Summary: 283 layers, 7255094 parameters, 7255094 gradients, 16.8 GFLOPS

Scaled weight_decay = 0.0005
Optimizer groups: 62 .bias, 70 conv.weight, 59 other
train: Scanning '../train/labels' for images and labels... 140
found, 0 missing, 0 empty, 0 corrupted: 100% 140/140 [00:00<00:00, 2087.96it/s]
train: New cache created: ../train/labels.cache
train: Caching images (0.1GB): 100% 140/140 [00:00<00:00,
429.69it/s]
val: Scanning '../valid/labels' for images and labels... 40 found,
0 missing, 0 empty, 0 corrupted: 100% 40/40 [00:00<00:00, 1733.99it/s]
val: New cache created: ../valid/labels.cache
val: Caching images (0.0GB): 100% 40/40 [00:00<00:00, 283.62it/s]
Plotting labels...

autoanchor: Analyzing anchors... anchors/target = 5.36, Best
Possible Recall (BPR) = 0.9931
Image sizes 416 train, 416 test
Using 2 dataloader workers
Logging results to runs/train/yolov5s_results
Starting training for 100 epochs...

     Epoch   gpu_mem       box       obj       cls     total   targets  img_size
      0/99     1.79G    0.1086    0.0208         0    0.1294        26
416: 100% 9/9 [00:08<00:00,  1.06it/s]
               Class      Images     Targets           P           R      mAP@.5
mAP@.5:.95: 100% 2/2 [00:02<00:00,  1.40s/it]
                 all          40          41      0.0051       0.976     0.00565
0.000752

...

     Epoch   gpu_mem       box       obj       cls     total   targets  img_size
     99/99     1.82G   0.03599   0.01685         0   0.05284        15
416: 100% 9/9 [00:03<00:00,  2.36it/s]
               Class      Images     Targets           P           R      mAP@.5
mAP@.5:.95: 100% 2/2 [00:01<00:00,  1.63it/s]
                 all          40          41       0.769       0.975       0.818
0.448
Optimizer stripped from runs/train/yolov5s_results/weights/last.pt, 14.7MB
Optimizer stripped from runs/train/yolov5s_results/weights/best.pt, 14.7MB
100 epochs completed in 0.143 hours.

CPU times: user 5.96 s, sys: 882 ms, total: 6.84 s
Wall time: 8min 56s
```

Evaluate Custom YOLOv5 Detector Performance:

[ ]:
```
%load_ext tensorboard
%tensorboard --logdir runs
```

```
&lt;IPython.core.display.Javascript object&gt;
```

[ ]:
```
from utils.plots import plot_results  # plot results.txt as results.png
Image(filename='/content/yolov5/runs/train/yolov5s_results/results.png',␣
↪width=1000)  # view results.png
```

[ ]:

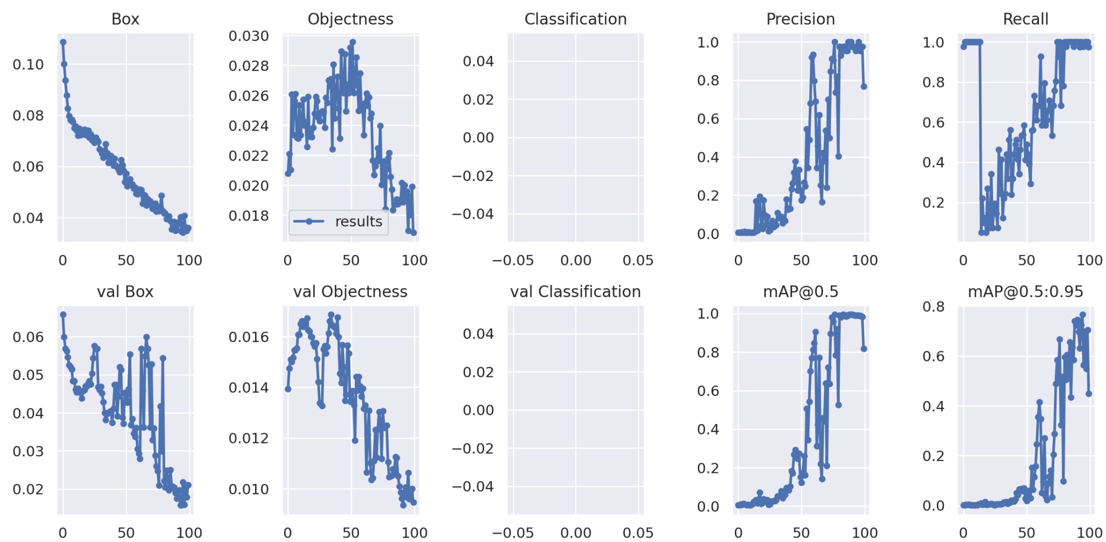


Figure 14: YOLO v5 detector evaluation

Visualize Our Training Data with Labels:

[ ]:
```
# first, display our ground truth data
print("GROUND TRUTH TRAINING DATA:")
Image(filename='/content/yolov5/runs/train/yolov5s_results/test_batch0_labels.
↪jpg', width=900)
```

```
GROUND TRUTH TRAINING DATA:
```

[ ]:


Figure 15: Ground truth training data samples with labels

[ ]:
```
# print out an augmented training example
print("GROUND TRUTH AUGMENTED TRAINING DATA:")
Image(filename='/content/yolov5/runs/train/yolov5s_results/train_batch0.jpg',␣
 ↪width=900)
```

```
GROUND TRUTH AUGMENTED TRAINING DATA:
```

[ ]:

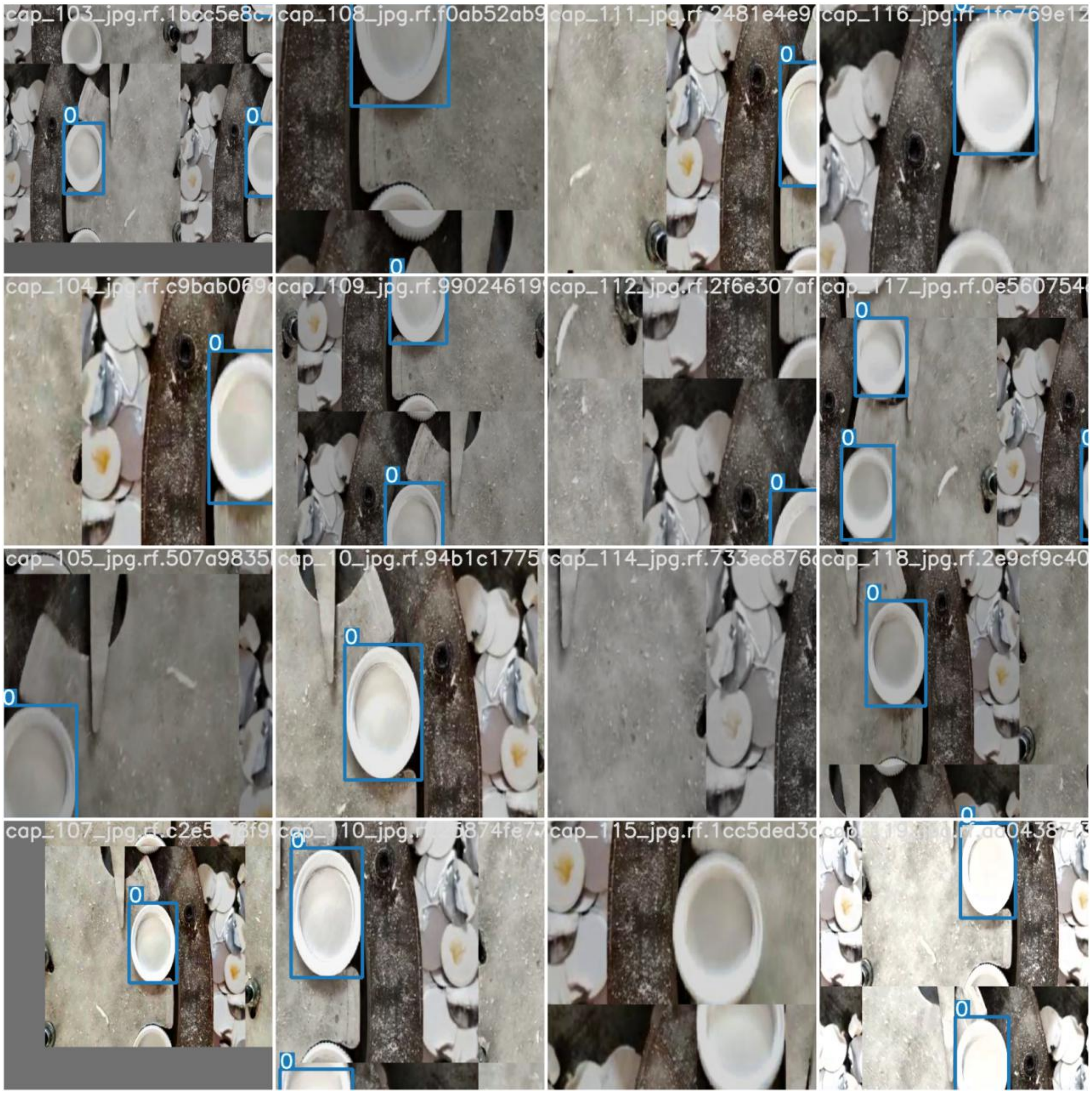


Figure 16: Augmented ground truth training data samples with labels

[ ]:
```
%ls runs/
```

```
train/
```

[ ]:
```
%ls runs/train/yolov5s_results/weights
```

```
best.pt   last.pt
```

[ ]:
```
%cd /content/yolov5/
!python detect.py --weights runs/train/yolov5s_results/weights/best.pt --img 416␣
↪--conf 0.4 --source ../test/images
```

```
/content/yolov5
Namespace(agnostic_nms=False, augment=False, classes=None, conf_thres=0.4,
device='', exist_ok=False, img_size=416, iou_thres=0.45, name='exp',
project='runs/detect', save_conf=False, save_txt=False, source='../test/images',
update=False, view_img=False,
weights=['runs/train/yolov5s_results/weights/best.pt'])
YOLOv5 v4.0-126-g886f1c0 torch 1.10.0+cu111 CUDA:0 (Tesla K80, 11441.1875MB)

Fusing layers...
/usr/local/lib/python3.7/dist-packages/torch/functional.py:445: UserWarning:
torch.meshgrid: in an upcoming release, it will be required to pass the indexing
argument. (Triggered internally at
../aten/src/ATen/native/TensorShape.cpp:2157.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
Model Summary: 232 layers, 7246518 parameters, 0 gradients, 16.8 GFLOPS
image 1/20 /content/yolov5/../test/images/cap_100_jpg.rf.04d2e5f139a204fe67ba4a6
e318b9352.jpg: 416x416 1 Caps, Done. (0.033s)
image 2/20 /content/yolov5/../test/images/cap_101_jpg.rf.bcafe407a27e06e31019d28
02a5f267a.jpg: 416x416 1 Caps, Done. (0.033s)
image 3/20 /content/yolov5/../test/images/cap_102_jpg.rf.67265b95ec5339ff91158ab
2ea35e9ed.jpg: 416x416 1 Caps, Done. (0.033s)
image 4/20 /content/yolov5/../test/images/cap_113_jpg.rf.85dd06988ea675614c4bb36
98937e9e6.jpg: 416x416 1 Caps, Done. (0.032s)
image 5/20 /content/yolov5/../test/images/cap_121_jpg.rf.99cc4b1767233b440b7397b
fe1f77bdf.jpg: 416x416 1 Caps, Done. (0.033s)
image 6/20 /content/yolov5/../test/images/cap_125_jpg.rf.1ba0709ce7f0a493a741cfd
b1c2f996d.jpg: 416x416 1 Caps, Done. (0.032s)
image 7/20 /content/yolov5/../test/images/cap_12_jpg.rf.3188210d6200a6602a3a0cef
13a85418.jpg: 416x416 1 Caps, Done. (0.033s)
image 8/20 /content/yolov5/../test/images/cap_14_jpg.rf.6b8857c5e479723cd280dedf
8a98ad0b.jpg: 416x416 1 Caps, Done. (0.033s)
image 9/20 /content/yolov5/../test/images/cap_151_jpg.rf.43f8a7c779e09a08879da99
fd13ac9cc.jpg: 416x416 1 Caps, Done. (0.032s)
image 10/20 /content/yolov5/../test/images/cap_154_jpg.rf.604207823d49f08e5b013f
ffa6ff2110.jpg: 416x416 1 Caps, Done. (0.032s)
image 11/20 /content/yolov5/../test/images/cap_165_jpg.rf.dd42c9c02ac812af1e415f
0f31119a2a.jpg: 416x416 1 Caps, Done. (0.033s)
image 12/20 /content/yolov5/../test/images/cap_189_jpg.rf.000b2581f1304c8f69805f
e09986ebcb.jpg: 416x416 1 Caps, Done. (0.033s)
image 13/20 /content/yolov5/../test/images/cap_25_jpg.rf.5055e576f507088c0986dc5
765961294.jpg: 416x416 1 Caps, Done. (0.032s)
image 14/20 /content/yolov5/../test/images/cap_53_jpg.rf.03389dd7ec7c331a6fe6eaf
2e0adbb4a.jpg: 416x416 1 Caps, Done. (0.032s)
```

```
image 15/20 /content/yolov5/../test/images/cap_68_jpg.rf.781215a4e29d4f6d03858e3
f983e7fb8.jpg: 416x416 Done. (0.032s)
image 16/20 /content/yolov5/../test/images/cap_72_jpg.rf.e2c94bdfc62bd0895a30066
fb847ede4.jpg: 416x416 1 Caps, Done. (0.032s)
image 17/20 /content/yolov5/../test/images/cap_79_jpg.rf.e6b2975dc8af5a4724007d5
161342f6a.jpg: 416x416 1 Caps, Done. (0.032s)
image 18/20 /content/yolov5/../test/images/cap_83_jpg.rf.d337aa4233ab8ffcce1668a
55bb761ab.jpg: 416x416 1 Caps, Done. (0.031s)
image 19/20 /content/yolov5/../test/images/cap_93_jpg.rf.3b41aa7c94cab22004cf90f
9cab53f95.jpg: 416x416 1 Caps, Done. (0.031s)
image 20/20 /content/yolov5/../test/images/cap_98_jpg.rf.28d796ab0d219b3ab51ca37
cf6cc2945.jpg: 416x416 1 Caps, Done. (0.031s)
Results saved to runs/detect/exp
Done. (0.843s)
```

[ ]:
```
import glob
from IPython.display import Image, display

for imageName in glob.glob('/content/yolov5/runs/detect/exp/*.jpg'): #assuming␣
 ↪JPG
    display(Image(filename=imageName))
    print("\n")
```


Caps 0.56


Caps 0.54


Caps 0.63


Caps 0.49


Caps 0.68


Caps 0.62


Caps 0.54


Caps 0.57


Caps 0.66


Caps 0.42


Caps 0.56


Caps 0.48


Caps 0.61


Caps 0.53


Caps 0.52


Caps 0.47


Caps 0.53


Figure 17: Detected objects with label and probability