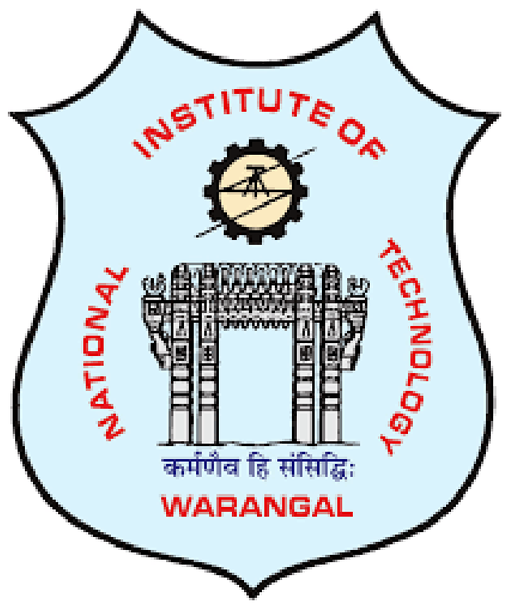
SUMMER INTERNSHIP

DOCUMENTATION ON MACHINE LEARNING

AT

NATIONAL INSTITUTE OF TECHNOLOGY

WARANGAL



Under the guidance of

Dr. J. Ravikumar sir

(Professor)

Submitted By

Gokeda Manibabu

s180169

Table of contents

Abstract

The Dress Code Detection project utilizes the ResNet50 convolutional neural network and the Adam Delta optimizer to develop a system that can determine whether a student's attire complies with the dress code guidelines. By analysing images, the system classifies clothing items as either allowed or not allowed based on predefined rules. The ResNet50 architecture is choosen for its excellent performance in image classification, while the Adam Delta optimizer ensures efficient training. The system is trained on a dataset of labelled images, learning to identify non-compliant clothing features such as offensive text or prohibited accessories. Once deployed, the system provides real-time feedback on attire

compliance, enabling objective enforcement of dress code policies. The project aims to improve efficiency, minimize subjective interpretation, and promote a respectful learning environment within educational institutions.

What is meant by Dress Code?

Dress code refers to a set of guidelines or rules that dictate appropriate clothing and appearance in specific settings. It varies depending on the context, such as schools, workplaces, or formal events. Dress codes aim to promote professionalism, maintain decorum, ensure safety, and reflect the values of the organization. They

specify aspects like clothing types, colors, patterns, lengths, accessories, and grooming standards. Dress codes are enforced through policies and compliance is expected to maintain a professional or appropriate appearance in the given setting.

✓ YES

Business Suit w/ a belt or
Blazer & Dress Pants w/ a belt
(with collared dress shirt, and necktie/ bow tie)

Dress Pants w/ a belt
(black, blue, brown, gray, plaid or print pants with collared dress shirt, and necktie/ bow tie)

Collared Dress Shirt – Tucked In
(Long or Short Sleeve with necktie/ bow tie)

Necktie or Bow Tie

Dress Shoes/ Dress Boots and socks

✗ NO

Cargo Pants, Sweatpants,
Athletic Pants, or Shorts

Polo/Golf Shirt, Henley Shirt,
Hawaiian Shirt, Flannel Shirt, or T-Shirt

Sperry's or Boat Shoes, Cowboy Boots,
Hiking Boots, Sneakers or Athletic
Footwear, Slides, or Vans

Why Dress Code is necessary?

Dress codes are necessary for various reasons. They promote professionalism, create a positive image, and ensure safety and security in different settings. Dress

codes maintain decorum, prevent discrimination, and reflect organizational values. They also provide guidance for specific occasions and contribute to a cohesive and unified appearance. Overall, dress codes help establish a suitable environment, align with organizational goals, and maintain a professional image.

What is the purpose of dress code detection project?

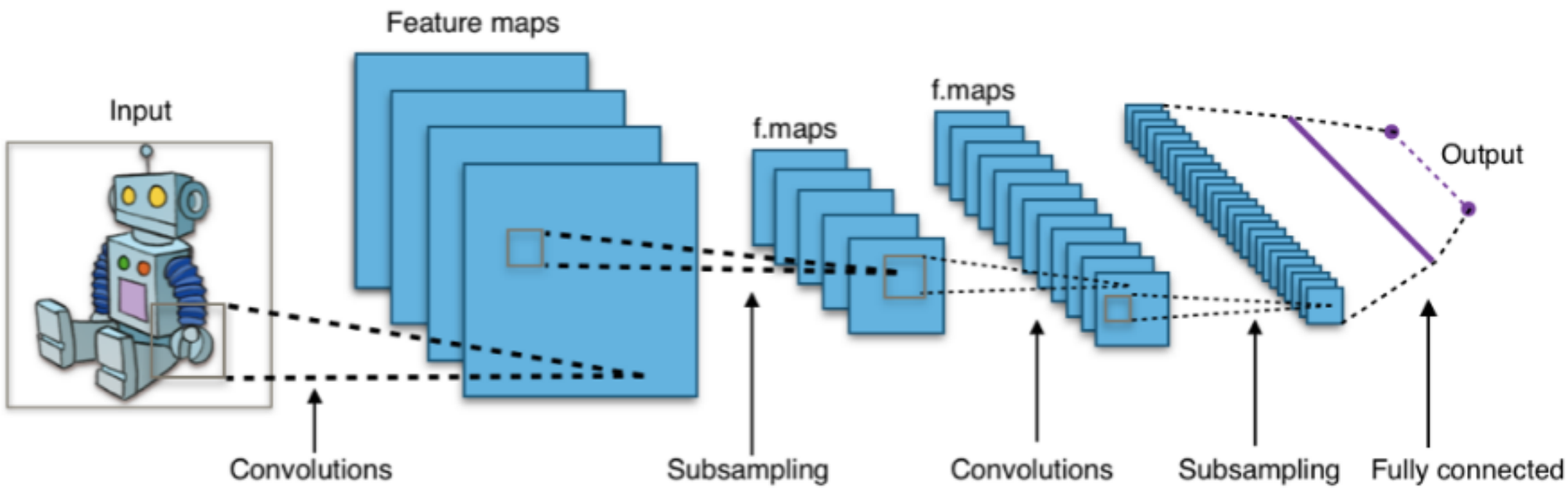
The Dress Code Detection project aims to automate the process of enforcing dress code policies in educational institutions. By utilizing computer vision and deep learning techniques, the project develops a system that can objectively analyze images of students' attire and determine if it complies with the dress code guidelines. The goal is to streamline monitoring, reduce subjective interpretation, and ensure consistent implementation of the dress code rules. The project aims to improve efficiency, save resources, and create a respectful and compliant learning environment through automated dress code enforcement.

Which algorithm we are used for Dress Code Detection?

We are using CNN algorithm for Dress Code Detection. And we are using ResNet50 model architecture with AdaDelta Optimizer.

What is CNN?

CNN, or Convolutional Neural Network, is a deep learning algorithm designed for processing visual data like images and videos. Its architecture consists of convolutional, pooling, and fully connected layers. Convolutional layers extract features by applying filters to the input image. Pooling layers down sample the feature maps. Fully connected layers enable classification or regression based on learned features. CNNs learn and extract relevant features automatically, without explicit feature engineering. They excel in various computer vision tasks and have revolutionized image processing and analysis. CNNs are widely used for image classification, object detection, and image segmentation, among other tasks. They play a crucial role in the field of deep learning for understanding and processing visual information.



Why we are using CNN for Dress Code Detection?

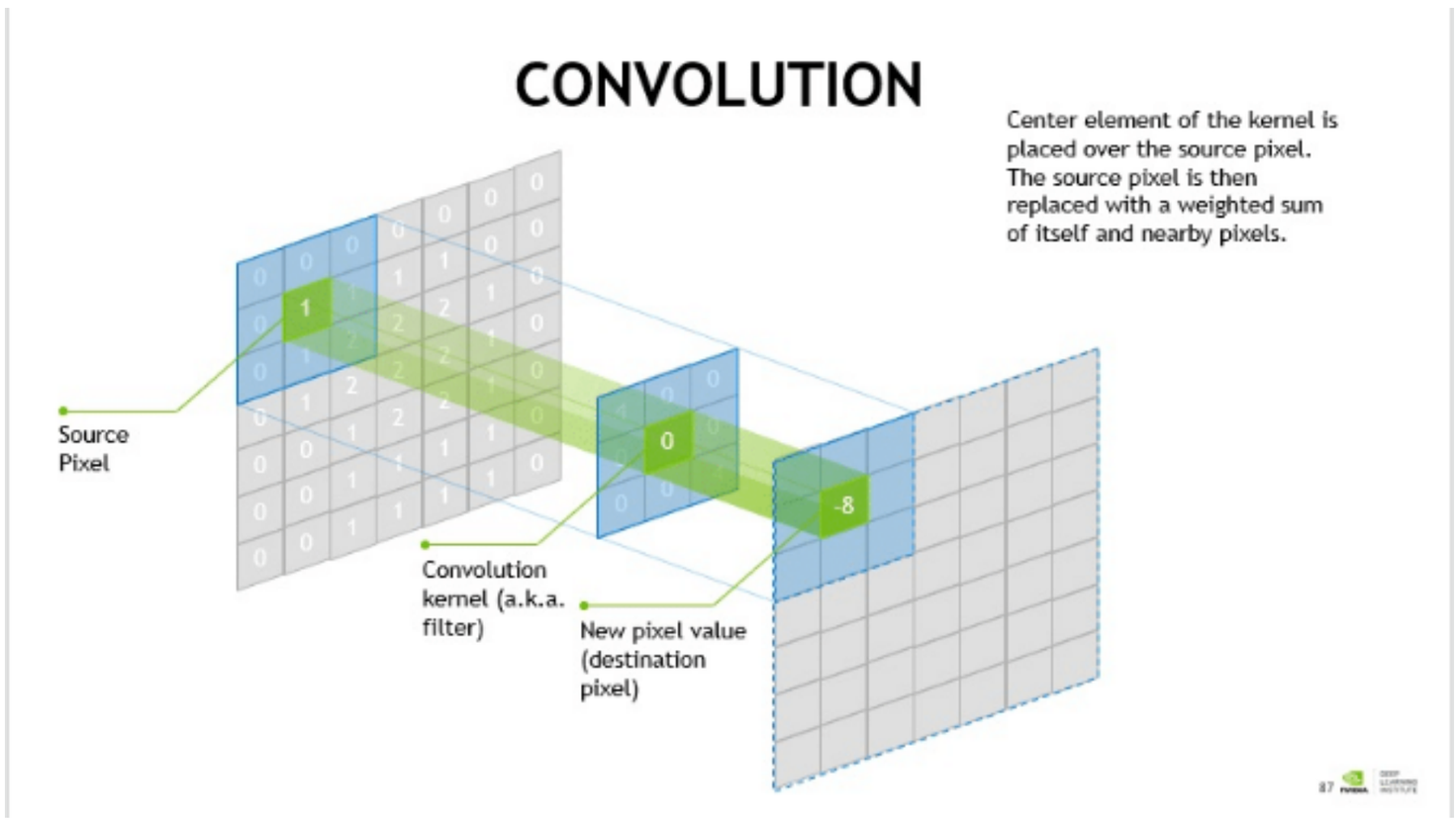
Dress code detection involves collecting a diverse dataset of images representing different dress codes and annotating them with corresponding labels. The dataset is preprocessed, including resizing, normalization, and augmentation. An image classification model is then trained using deep learning frameworks and architectures like VGG, ResNet, or Inception. The model is fine-tuned on the dress code dataset using appropriate loss functions and optimization techniques. Evaluation on a separate validation set helps adjust hyperparameters and network architecture. Once a trained and validated model is obtained, it can be deployed to predict dress codes in new images by preprocessing the test image and obtaining predictions from the model. Real-world challenges, such as occlusion, lighting

variations, and complex dress styles, may require additional techniques like object detection or semantic segmentation for improved performance.

Layers in CNN

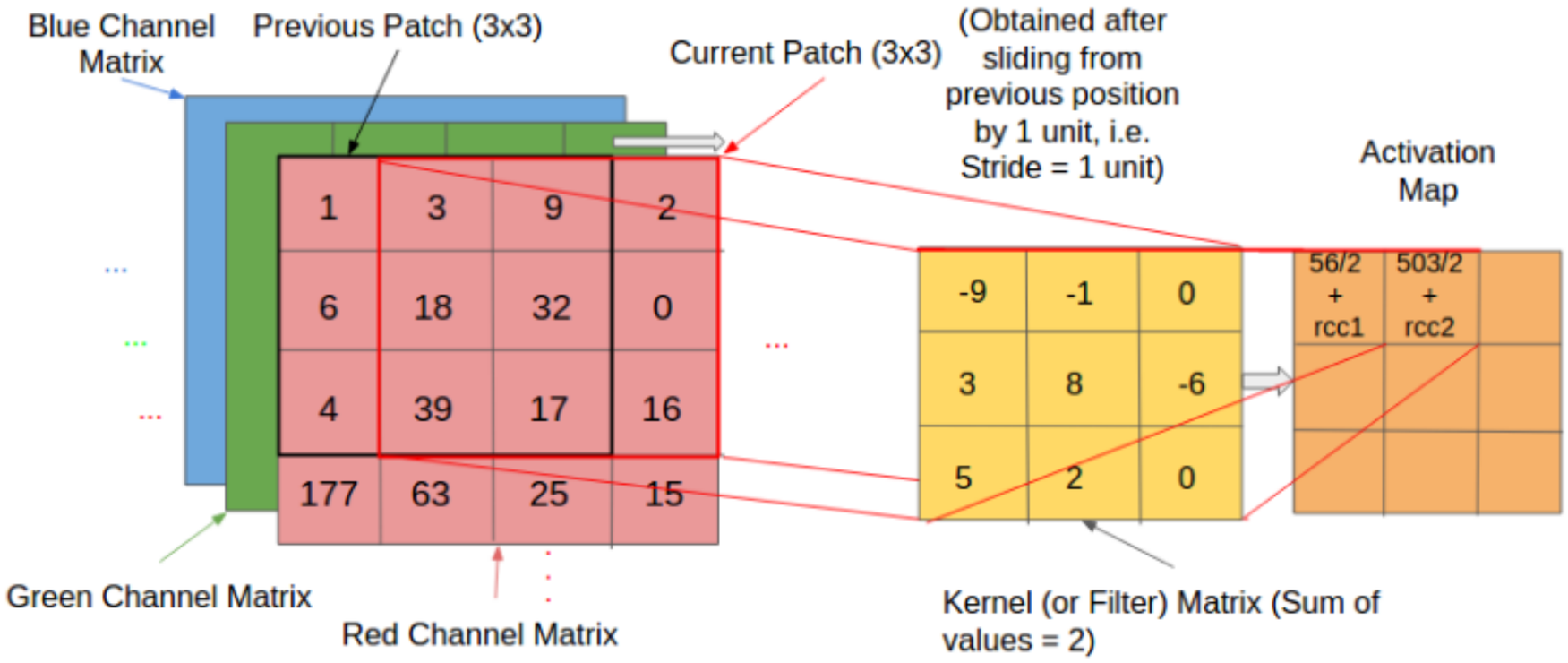
Convolutional Layer

The convolutional layer applies filters or kernels to the input image, convolving them across the image's spatial dimensions. This operation extracts relevant features and creates feature maps.



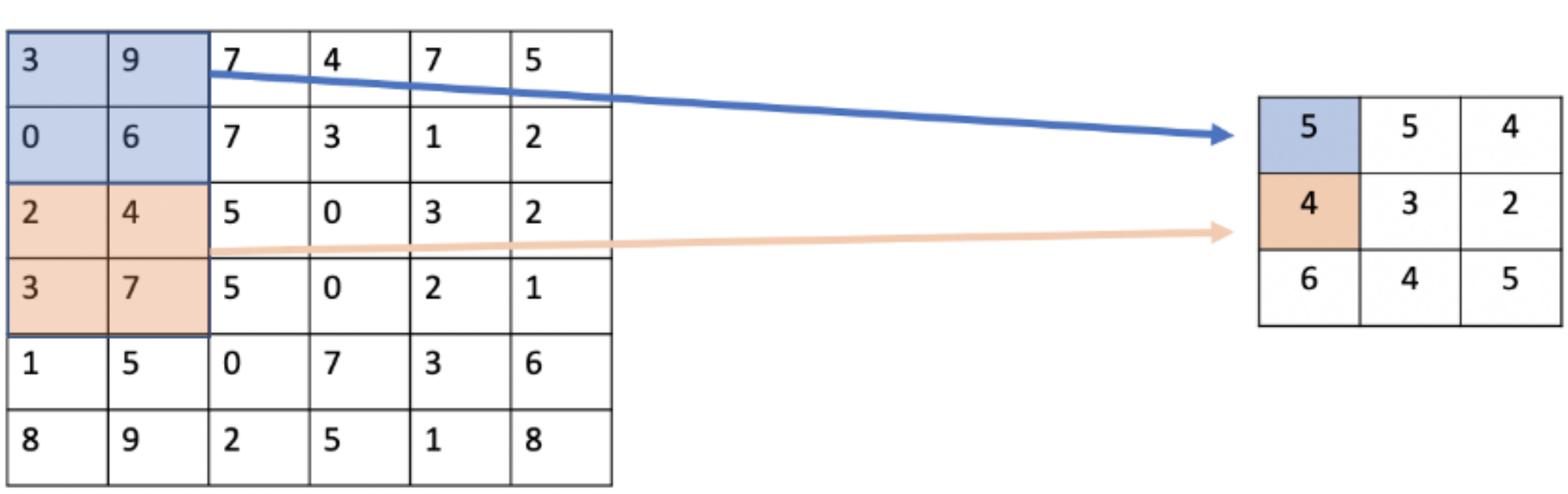
Activation Layer

An activation layer introduces non-linearity to the network by applying an activation function, such as ReLU (Rectified Linear Unit), to the output of the previous layer. It enhances the network's ability to learn complex patterns.



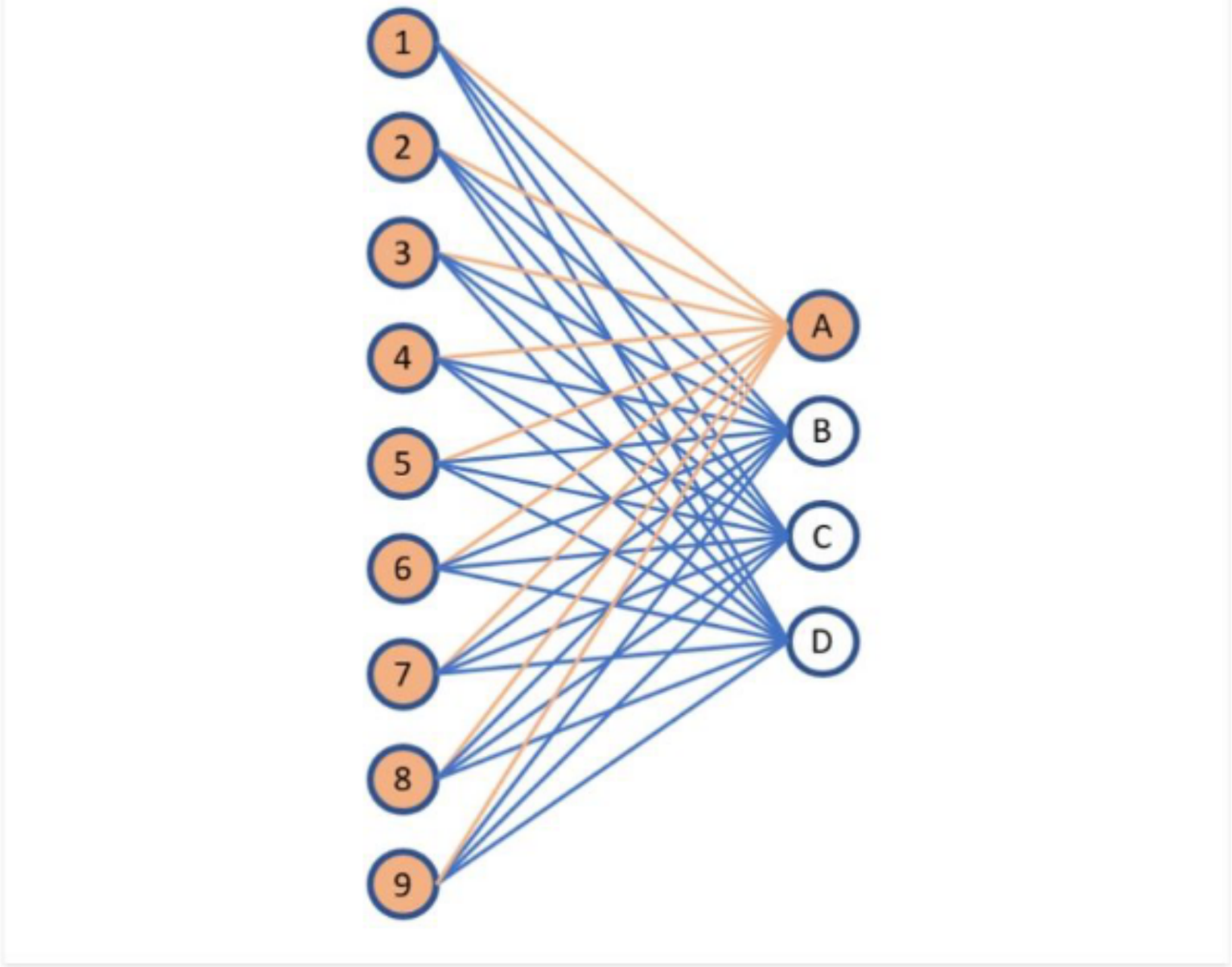
Pooling Layer

Pooling layers down sample the feature maps by reducing their spatial dimensions. Common pooling operations include max pooling or average pooling, which retain the most prominent features or compute the average value within each pooling region.



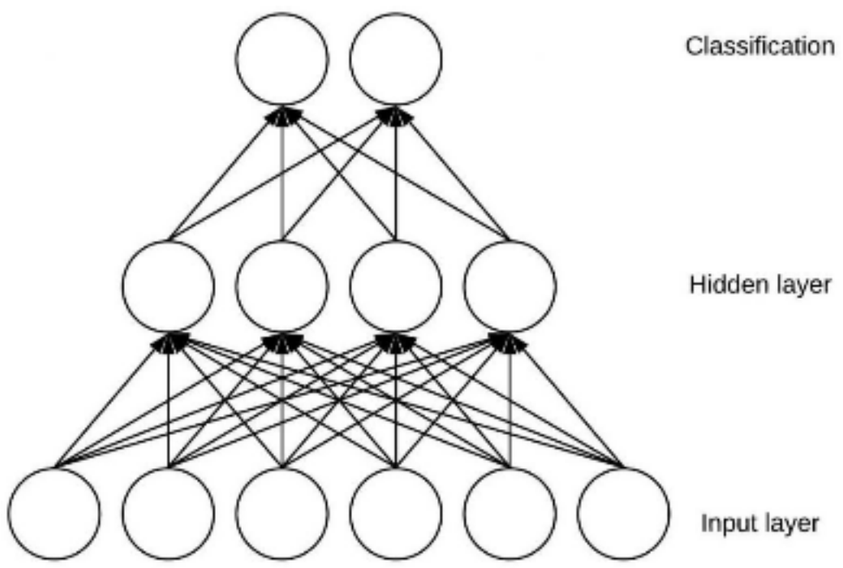
Fully Connected Layer

Fully connected layers connect every neuron from the previous layer to the subsequent layer. They perform high-level feature extraction and enable classification or regression based on the learned features. Fully connected layers are typically found in the final stages of the network.

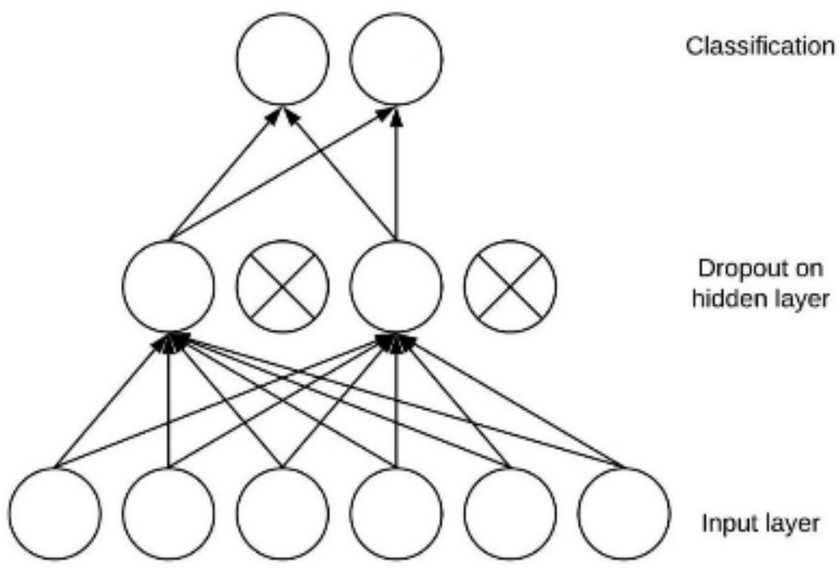


Dropout Layer

Dropout layers are used to reduce overfitting by randomly deactivating a fraction of the neurons during training. This helps prevent the network from relying too heavily on specific features.



Without Dropout

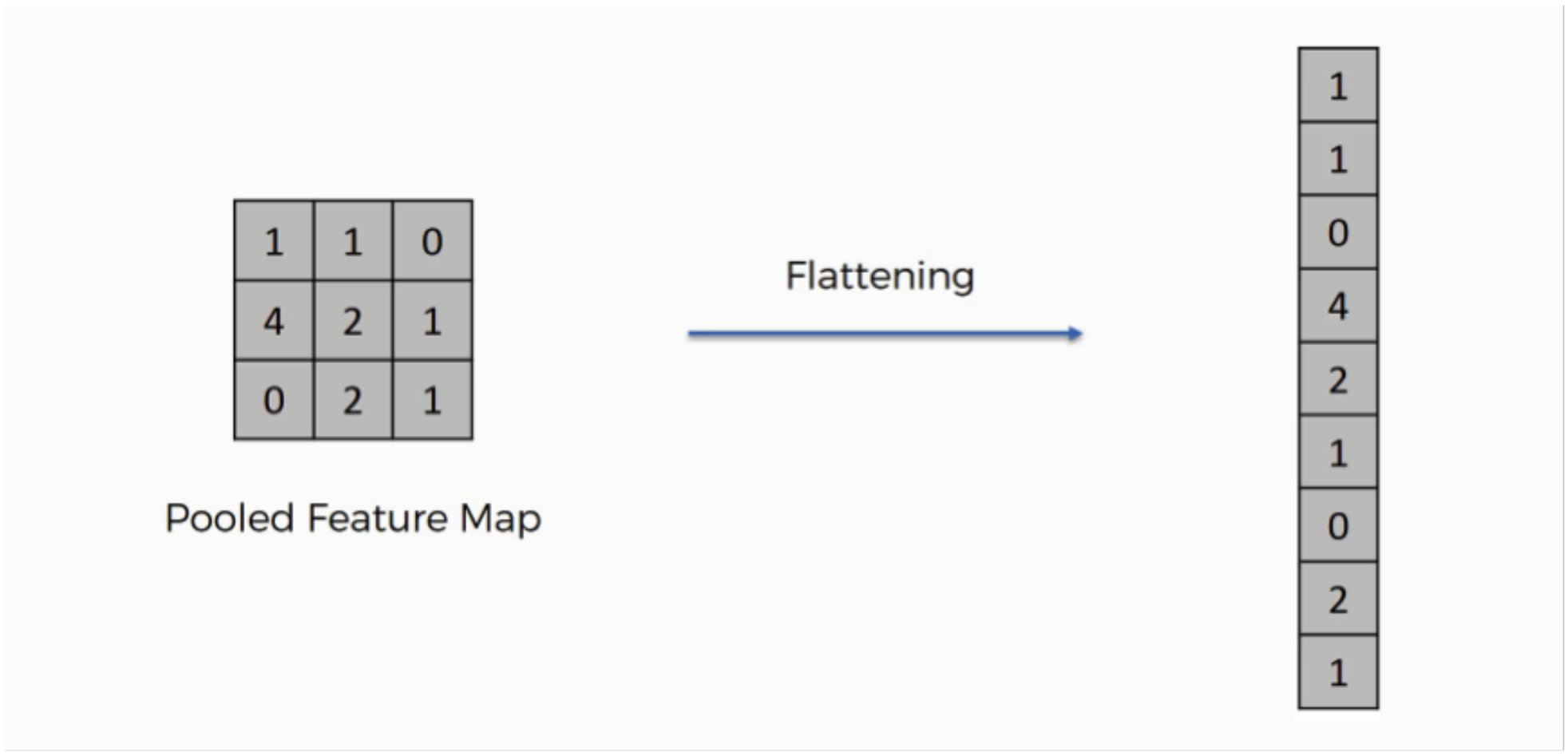


With Dropout

Flatten Layer

A flatten layer is used to convert the multi-dimensional feature maps into a one-dimensional vector, enabling compatibility with fully connected layers. These layers are stacked together to form the architecture of a CNN. The specific

arrangement and number of layers may vary depending on the network design and task requirements.



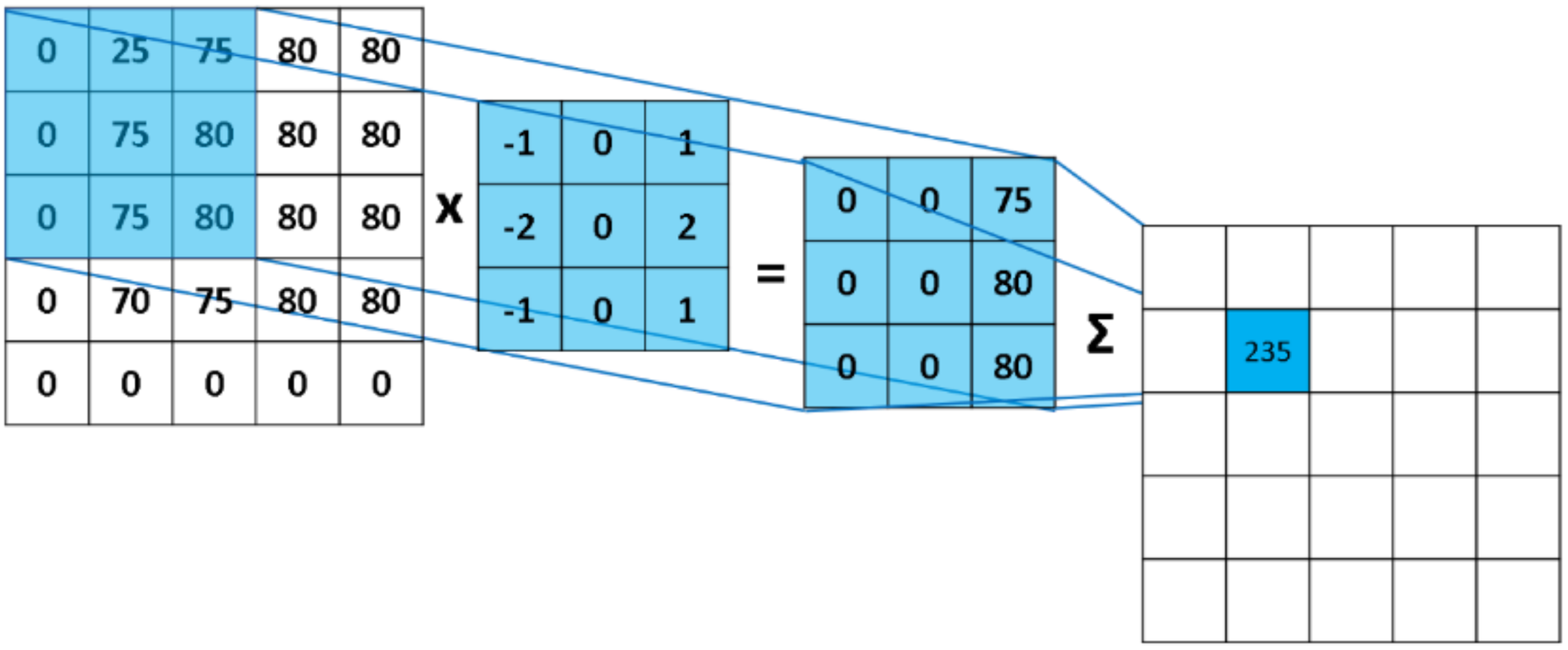
What is ResNet50?

ResNet50 is a deep convolutional neural network architecture consisting of 50 layers. It is part of the ResNet family developed by Microsoft Research, known for addressing the problem of vanishing gradients in deep networks. ResNet50 incorporates residual connections, allowing the network to learn residual mappings and train deep models effectively. It has been pre-trained on large image datasets like ImageNet, enabling it to extract meaningful features and classify objects accurately. ResNet50 is widely used in computer vision tasks due to its impressive performance and is commonly employed as a backbone architecture in state-of-the-art models for image recognition and analysis.

Layers in ResNet50

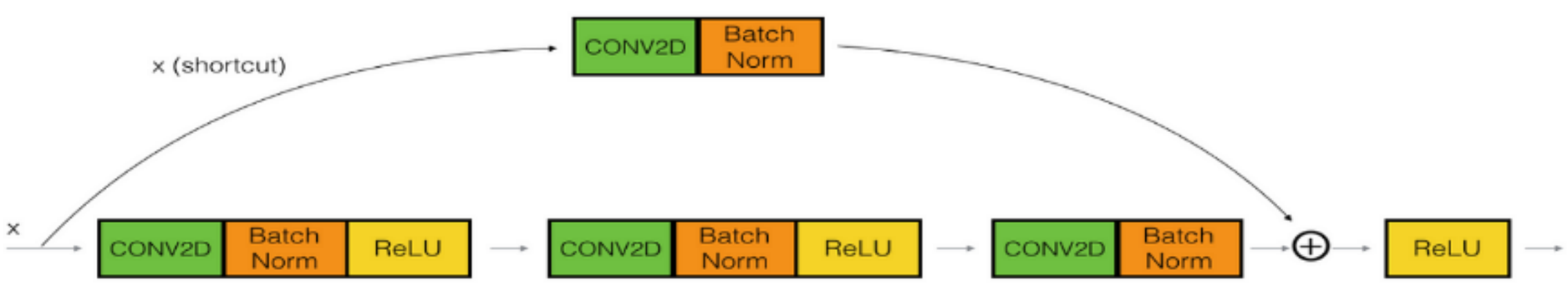
1. Convolutional Layer:

The initial layer performs a convolution operation on the input image, extracting low-level features such as edges and textures.



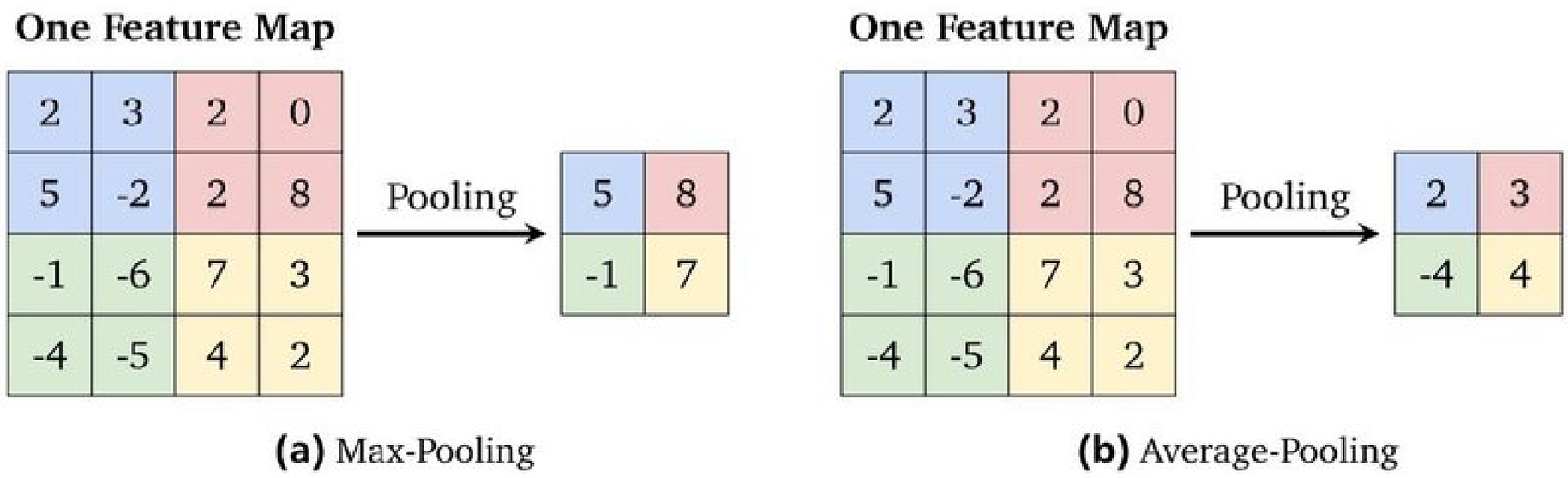
2. Residual Blocks:

ResNet50 consists of 16 residual blocks, with each block containing multiple convolutional layers. The residual blocks are the key building blocks of ResNet, and they enable the training of very deep networks by addressing the problem of vanishing gradients.



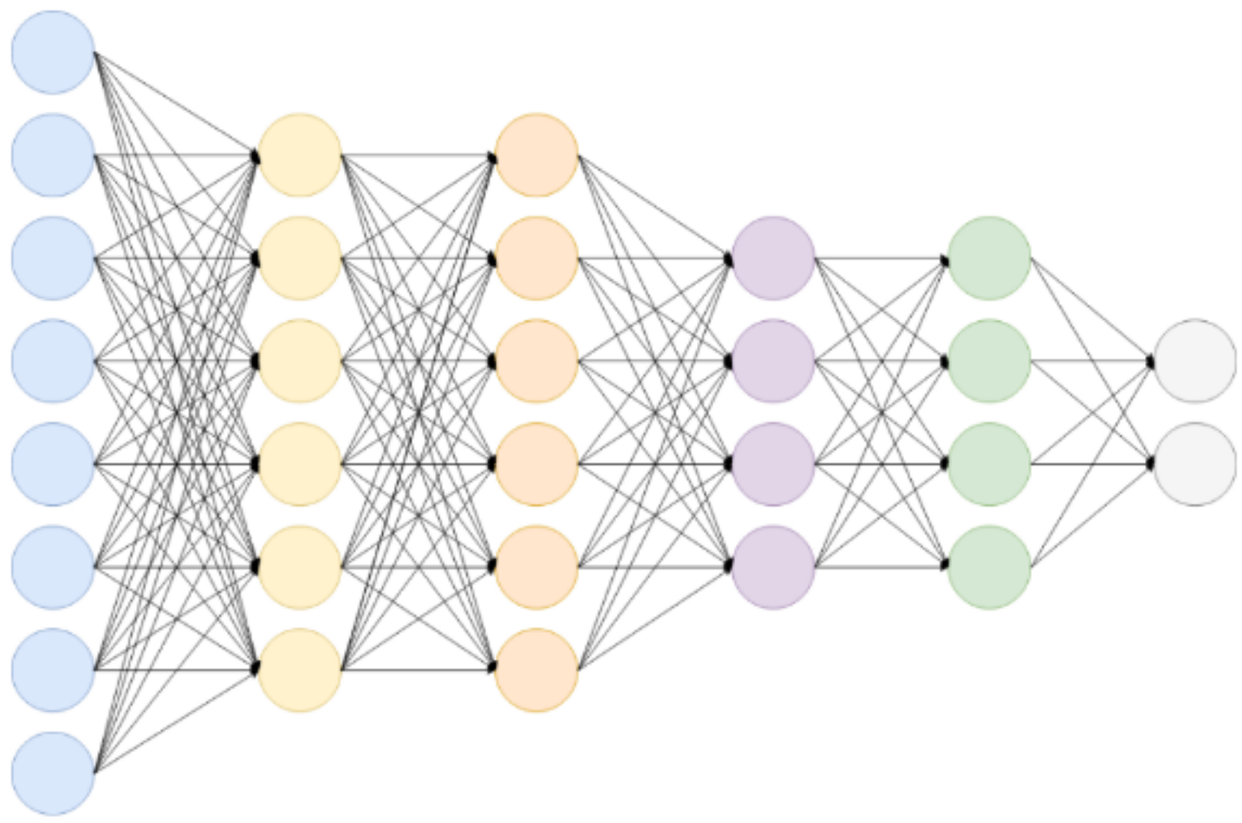
3. Global Average Pooling Layer:

Towards the end of the network, a global average pooling layer is applied, which reduces the spatial dimensions of the feature maps to a fixed size. This allows for a compact representation of the features extracted by the previous layers.



4. Fully Connected Layers:

Finally, ResNet50 includes fully connected layers that perform classification based on the learned features. These layers map the extracted features to the desired output classes, enabling the network to classify input images. Overall, ResNet50 consists of multiple convolutional layers, residual blocks with shortcut connections, a global average pooling layer, and fully connected layers. This architecture enables ResNet50 to learn and extract intricate features from input images, making it a powerful model for tasks like image classification and object recognition.

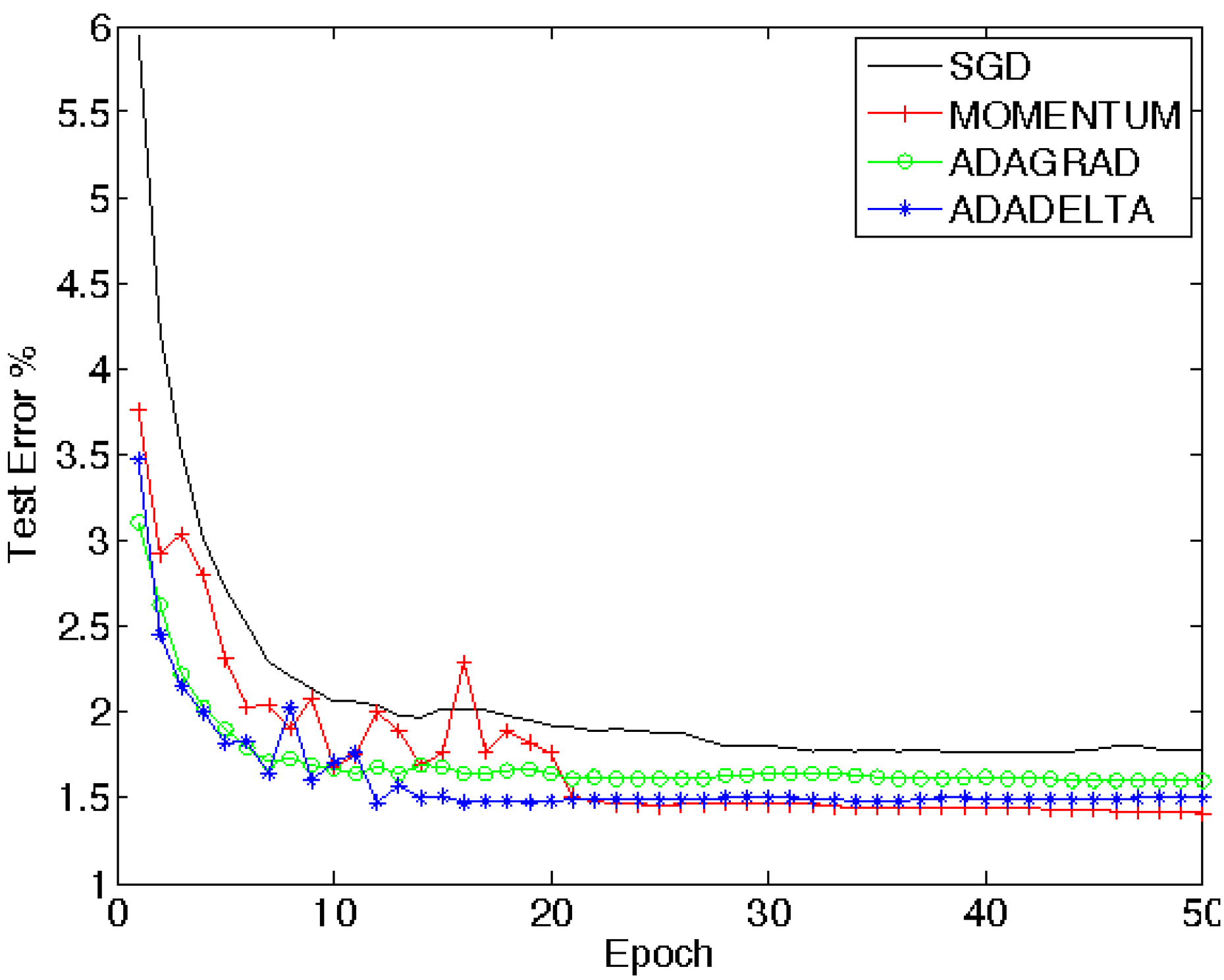


Model architecture code

```
input_shape = (256, 256, 3)
input_tensor = layers.Input(shape=input_shape)

# Stage 1
x = layers.Conv2D(64, (7, 7), strides=(2, 2),
padding='same')(input_tensor)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

# Stage 2
shortcut = x

x = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.Conv2D(256, (1, 1), strides=(1, 1), padding='same')(x)
x = layers.BatchNormalization()(x)

shortcut = layers.Conv2D(256, (1, 1), strides=(1, 1),
padding='same')(shortcut)
shortcut = layers.BatchNormalization()(shortcut)

x = layers.add([x, shortcut])
x = layers.Activation('relu')(x)

# Add more stages (Stage 3, 4, and 5) following a similar pattern...

# Output layer
x = layers.Flatten()(x)
output = layers.Dense(n_classes, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output)
```

What is AdaDelta?

AdaDelta is an optimization algorithm used for training deep neural networks. It dynamically adjusts the learning rate based on the historical information of gradients. Unlike AdaGrad, it accumulates a running average of squared gradients and discards older information. AdaDelta introduces an "accumulator" to track previous updates, which determines the learning rate for each parameter. It eliminates the need for manual learning rate setting and adapts to the optimization problem's characteristics. AdaDelta is widely used in deep learning, providing efficient and adaptive updates to model parameters.



Hand Calculation for AdaDelta

Instead of considering all of the gradients that were calculated, ADADELTA considers only the last we gradients.

Since storing w previous squared gradients is inefficient, our methods implement this accumulation as an exponentially decaying average of the squared gradients. Assume at time t this running average is E[g2]t then we compute:

E[g2]t=ρE[g2]t-1+(1-ρ)g2t)

where ρ is a decay constant [...]. Since we require the square root of this quantity in the parameter updates, this effectively becomes the RMS of previous squared gradients up to time t:

RMS [g]t=Eg2t+ϵ

where a constant ϵ is added to better condition the denominator

Similarly:

E[Δ x2]t-1=ρE[Δ x2]t-2++1-ρΔ x2t-1)

RMS [Δ x]t-1=E[Δ x2]t-1+ϵ

And finally:

[...] approximateΔ xt by compute the exponentially decaying RMS over a window of size of w previous Δ x to give the ADADELTA method:

Δ xt=RMS [Δ x]t-1RMS [g]tgt

where the same constant ϵ is added to the numerator RMS as well. This constant serves the purpose both to start off the first iteration where Δx0=0 and to ensure progress continues to be made even if previous updates become small.
[...]

The numerator acts as an acceleration term, accumulating previous gradients over a window of time [...]

if the gradient in step r isgr=arbrcr and Δ xr=irjrkr, then:

$$\Delta x_t = -\mathrm{RMS}[\Delta x]_{t-1}\mathrm{RMS}[g]_t g_t = -E[\Delta x^2]_{t-1} + \epsilon Eg^2_t + \epsilon g_t =$$

$$-\rho E[\Delta x^2]_{t-2} + (1-\rho)\Delta x^2_{t-1} + \epsilon\rho Eg^2_{t-1} + 1-\rho g^2_t + \epsilon g_t =$$

$$-\rho\rho E\Delta x^2_{t-3} + 1-\rho\Delta x^2_{t-2} + 1-\rho\Delta x^2_{t-1} + \epsilon\rho(\rho Eg^2_{t-2} + 1-\rho g^2_{t-1}) + 1-\rho g^2_t + \epsilon g_t =$$

$$-\rho^2 E\Delta x^2_{t-3} + p11-\rho\Delta x^2_{t-2} + p01-\rho\Delta x^2_{t-1} + \epsilon\rho^2 Eg^2_{t-2} + p11-\rho g^2_{t-1}) + p01-\rho g^2_t + \epsilon g_t =$$

$$-\rho^{t-1} E\Delta x^2_0 + r=1^{t-1}\rho^{t-1-r}1-\rho x^2_r + \epsilon\rho^{t-1} Eg^2_1 + r=2^t \rho^{t-r}1-\rho g^2_r + \epsilon g_t =$$

ρ is a decay constant, so we choose it such that $\rho \in (0,1) \in (0,1)$ (typically $\rho \geqslant 0.9$).
Therefore, multiplying by a high power of ρ results in a very small number.
Let w be the lowest exponent such that we deem the product of multiplying sane values by ρ^w negligible.
Now, we can approximate Δx_t by dropping negligible terms:

$$\Delta x_t \approx -r=t-w^{t-1}\rho^{t-1-r}1-\rho\Delta x^2_r + \epsilon r=t+1-w t\rho^{t-r}1-\rho g^2_r + \epsilon g_t = -r=t-w^{t-1}\rho^{t-1-r}1-\rho i^2_r j^2_r k^2_r, + \epsilon r=t+1-w^{t-1}\rho^{t-r}1-\rho a^2_r b^2_r c^2_r, + \epsilon a_t b_t c_t$$

$$\Delta x_t \approx r=t-w^{t-1}\rho^{t-1-r}1-\rho i^2_r, + \epsilon r=t+1-w^{t-1}\rho^{t-r}1-\rho a^2_r, + \epsilon + a_t r=t-w^{t-1}\rho^{t-1-r}1-\rho j^2_r, + \epsilon r=t+1-w^{t-1}\rho^{t-r}1-\rho b^2_r, + \epsilon + b_t r=t-w^{t-1}\rho^{t-1-r}1-\rho k^2_r, + \epsilon r=t+1-w^{t-1}\rho^{t-r}1-\rho c^2_r, + \epsilon + c_t$$

Dress Code Detection Using CNN (ResNet)

Dataset

We are creating our data set consists of more than 2098 images. Which consists of both allowed and not allowed images .

We are creating a folder called new. It contains sub folders Allowed and Not Allowed.

Allowed folder contains the images which are suitable to wear for schools, collages, etc.

Eg:

Not Allowed folder contains the images which are suitable to wear for schools, collages, etc.

Eg:

All the images are reshaped into 840*480

Implementation of code

Step1: Import Libraries

```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
from keras.layers import Dropout
```

Step2: Mount the google drive

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Output:

```
Mounted at /content/gdrive
```

Step3: Import the image folder path

```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
"/content/gdrive/MyDrive/new",labels='inferred',shuffle=True)
```

Output:

```
Found 2096 files belonging to 2 classes.
```

Step4: Define the class names

```
class_names = dataset.class_names
class_names
```

Output:

```
['allowed', 'notallowed']
```

Step5: Represent the shape and dimentions of image batch

```
for image_batch, labels_batch in dataset.take(1):
 print(image_batch.shape)
 print(labels_batch.numpy())
```

Output:

```
(32, 256, 256, 3)
[1 0 0 1 1 1 1 1 0 1 1 1 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 1 0 0 0 0]
```

Step6: Print random images with labels

```
plt.figure(figsize=(15, 6))
for image_batch, labels_batch in dataset.take(1):
 for i in range(12):
  ax = plt.subplot(3, 4, i + 1)
  plt.imshow(image_batch[i].numpy().astype("uint8"))
  plt.title(class_names[labels_batch[i]])
  plt.axis("on")
```

Output:

Step7: Divide the data into training and testing

```
def get_dataset_partitions_tf(ds, train_split=0.8,  test_split=0.2,
shuffle=True, shuffle_size=10):
    assert (train_split + test_split ) == 1
    ds_size = len(ds)
    if shuffle:
      ds = ds.shuffle(shuffle_size, seed=12)
    train_size = int(train_split * ds_size)
    train_ds = ds.take(train_size)
    test_ds = ds.skip(train_size)
    return train_ds, test_ds
train_ds,test_ds = get_dataset_partitions_tf(dataset)
```

Step8: Shuffle the images in training and testing data

```
train_ds =
train_ds.cache().shuffle(10).prefetch(buffer_size=tf.data.AUTOTUNE)
test_ds =
test_ds.cache().shuffle(10).prefetch(buffer_size=tf.data.AUTOTUNE)
```

Step9: Resize the images

```
resize_and_rescale = tf.keras.Sequential([
layers.experimental.preprocessing.Resizing(256,256),
layers.experimental.preprocessing.Rescaling(1./255),
])
```

Step10: Preprocess the data

```
data_augmentation = tf.keras.Sequential([
layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
layers.experimental.preprocessing.RandomRotation(0),
])
train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y)
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

Step11: Initialize the inputs

```
input_shape = (32,256,256, CHANNELS)
n_classes = 2
```

Step12: Import the model architecture

```
import keras
from keras.layers import Input, Conv2D, MaxPooling2D,
BatchNormalization, Activation, Flatten, Dense
from tensorflow.keras import layers, models

input_shape = (256, 256, 3)
input_tensor = layers.Input(shape=input_shape)

# Stage 1
x = layers.Conv2D(64, (7, 7), strides=(2, 2),
padding='same')(input_tensor)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

# Stage 2
shortcut = x

x = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.Conv2D(256, (1, 1), strides=(1, 1), padding='same')(x)
```

```
x = layers.BatchNormalization()(x)

shortcut = layers.Conv2D(256, (1, 1), strides=(1, 1),
padding='same')(shortcut)
shortcut = layers.BatchNormalization()(shortcut)

x = layers.add([x, shortcut])
x = layers.Activation('relu')(x)

# Add more stages (Stage 3, 4, and 5) following a similar pattern...

# Output layer
x = layers.Flatten()(x)
output = layers.Dense(n_classes, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output)
```

Step13: Save the h5 file

```
model.save('resNet50.h5')
```

Output:

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics
have yet to be built. `model.compile_metrics` will be empty until you
train or evaluate the model.
```

Step14: Get the summary of the model

```
model.summary()
```

Output:

```
Model: "model"
_____________________________
 Layer (type)                Output Shape              Param #
Connected to
==================================================================
==========================
 input_1 (InputLayer)        [(None, 256, 256, 3  0           []
                             )]

 conv2d (Conv2D)             (None, 128, 128, 64  9472
['input_1[0][0]']
                             )

 batch_normalization (BatchNorm  (None, 128, 128, 64  256
['conv2d[0][0]']
 alization)                  )
```

```
 activation (Activation)        (None, 128, 128, 64  0
['batch_normalization[0][0]']
                                )

 max_pooling2d (MaxPooling2D)   (None, 64, 64, 64)   0
['activation[0][0]']

 conv2d_1 (Conv2D)              (None, 64, 64, 64)   4160
['max_pooling2d[0][0]']

 batch_normalization_1 (BatchNo  (None, 64, 64, 64)  256
['conv2d_1[0][0]']
 rmalization)

 activation_1 (Activation)      (None, 64, 64, 64)   0
['batch_normalization_1[0][0]']

 conv2d_2 (Conv2D)              (None, 64, 64, 64)   36928
['activation_1[0][0]']

 batch_normalization_2 (BatchNo  (None, 64, 64, 64)  256
['conv2d_2[0][0]']
 rmalization)

 activation_2 (Activation)      (None, 64, 64, 64)   0
['batch_normalization_2[0][0]']

 conv2d_3 (Conv2D)              (None, 64, 64, 256)  16640
['activation_2[0][0]']

 conv2d_4 (Conv2D)              (None, 64, 64, 256)  16640
['max_pooling2d[0][0]']

 batch_normalization_3 (BatchNo  (None, 64, 64, 256)  1024
['conv2d_3[0][0]']
 rmalization)

 batch_normalization_4 (BatchNo  (None, 64, 64, 256)  1024
['conv2d_4[0][0]']
 rmalization)

 add (Add)                      (None, 64, 64, 256)  0
['batch_normalization_3[0][0]',

'batch_normalization_4[0][0]']

 activation_3 (Activation)      (None, 64, 64, 256)  0
['add[0][0]']

 flatten (Flatten)              (None, 1048576)      0
['activation_3[0][0]']

 dense (Dense)                  (None, 2)            2097154
['flatten[0][0]']

==================================================================
============================
Total params: 2,183,810
Trainable params: 2,182,402
Non-trainable params: 1,408
```

Step15: Initialize the optimizer

```
model.compile(
    optimizer='AdaDelta',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

Step16: Calculate the accuracy by epochs

```
history=model.fit(
    train_ds,
    batch_size=32,
    verbose=1,
    epochs=30,
)
```

Output:

```
Epoch 1/30
52/52 [==============================] - 15s 278ms/step - loss: 0.0967 -
 accuracy: 0.9856
Epoch 2/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0892 -
 accuracy: 0.9904
Epoch 3/30
52/52 [==============================] - 15s 281ms/step - loss: 0.0810 -
 accuracy: 0.9886
Epoch 4/30
52/52 [==============================] - 14s 277ms/step - loss: 0.0805 -
 accuracy: 0.9874
Epoch 5/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0788 -
 accuracy: 0.9904
Epoch 6/30
52/52 [==============================] - 14s 278ms/step - loss: 0.0771 -
 accuracy: 0.9910
Epoch 7/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0710 -
 accuracy: 0.9898
Epoch 8/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0652 -
 accuracy: 0.9922
Epoch 9/30
52/52 [==============================] - 15s 281ms/step - loss: 0.0618 -
 accuracy: 0.9934
Epoch 10/30
52/52 [==============================] - 15s 281ms/step - loss: 0.0605 -
 accuracy: 0.9934
Epoch 11/30
```

```
52/52 [==============================] - 15s 281ms/step - loss: 0.0620 -
 accuracy: 0.9922
Epoch 12/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0586 -
 accuracy: 0.9916
Epoch 13/30
52/52 [==============================] - 15s 279ms/step - loss: 0.0536 -
 accuracy: 0.9934
Epoch 14/30
52/52 [==============================] - 15s 279ms/step - loss: 0.0511 -
 accuracy: 0.9964
Epoch 15/30
52/52 [==============================] - 14s 277ms/step - loss: 0.0497 -
 accuracy: 0.9940
Epoch 16/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0474 -
 accuracy: 0.9946
Epoch 17/30
52/52 [==============================] - 14s 276ms/step - loss: 0.0456 -
 accuracy: 0.9964
Epoch 18/30
52/52 [==============================] - 14s 275ms/step - loss: 0.0456 -
 accuracy: 0.9958
Epoch 19/30
52/52 [==============================] - 14s 277ms/step - loss: 0.0416 -
 accuracy: 0.9970
Epoch 20/30
52/52 [==============================] - 15s 282ms/step - loss: 0.0411 -
 accuracy: 0.9970
Epoch 21/30
52/52 [==============================] - 15s 281ms/step - loss: 0.0386 -
 accuracy: 0.9970
Epoch 22/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0367 -
 accuracy: 0.9976
Epoch 23/30
52/52 [==============================] - 15s 281ms/step - loss: 0.0354 -
 accuracy: 0.9976
Epoch 24/30
52/52 [==============================] - 15s 283ms/step - loss: 0.0337 -
 accuracy: 0.9988
Epoch 25/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0313 -
 accuracy: 0.9976
Epoch 26/30
52/52 [==============================] - 15s 279ms/step - loss: 0.0304 -
 accuracy: 1.0000
Epoch 27/30
52/52 [==============================] - 15s 280ms/step - loss: 0.0318 -
 accuracy: 0.9958
Epoch 28/30
52/52 [==============================] - 15s 278ms/step - loss: 0.0290 -
 accuracy: 0.9988
Epoch 29/30
52/52 [==============================] - 15s 283ms/step - loss: 0.0276 -
 accuracy: 0.9988
Epoch 30/30
```

```
52/52 [==============================] - 15s 286ms/step - loss: 0.0267 -
 accuracy: 1.0000
```

Step17: Calculate the accuracy of training data

```
scores = model.evaluate(train_ds)
```

Output:

```
52/52 [==============================] - 13s 248ms/step - loss: 0.0243 -
 accuracy: 1.0000
```

Step18: Calculate the accuracy of testing data

```
scores = model.evaluate(test_ds)
```

Output:

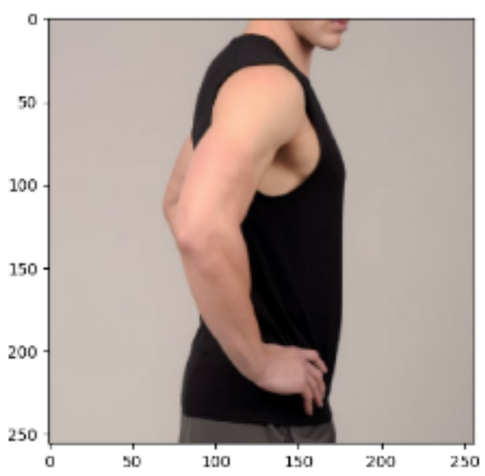
```
14/14 [==============================] - 0s 33ms/step - loss: 0.0405 -
accuracy: 0.9954
```

Step19: Predict the label of the image

```
import numpy as np
for images_batch, labels_batch in test_ds.take(1):
   first_image = images_batch[9].numpy().astype('uint8')
   first_label = labels_batch[9].numpy()
   print("first image to predict")
   plt.imshow(first_image)
   print("actual label:",class_names[first_label])
   batch_prediction = model.predict(images_batch)
   print("predicted
label:",class_names[np.argmax(batch_prediction[9])])
```

Output:

```
first image to predict
actual label: notallowed
1/1 [==============================] - 0s 37ms/step
predicted label: notallowed
```



Step20: Predict the random images

```
def predict(model, img):
    img_array =
tf.keras.preprocessing.image.img_to_array(images[i].numpy())
    img_array = tf.expand_dims(img_array, 0)
    predictions = model.predict(img_array)
    predicted_class = class_names[np.argmax(predictions[0])]
    confidence = round(100 * (np.max(predictions[0])), 2)
    return predicted_class, confidence
plt.figure(figsize=(15, 15))

for images, labels in test_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        predicted_class, confidence = predict(model, images[i].numpy())
        actual_class = class_names[labels[i]]
        plt.title(f"Actual: {actual_class},\n Predicted:
{predicted_class}.\n Confidence: {confidence}%")
        plt.axis("off")
```

Output:

```
1/1 [==============================] - 0s 277ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
```

Actual: allowed,
Predicted: allowed.
Confidence: 99.76%

Actual: notallowed,
Predicted: notallowed.
Confidence: 98.56%

Actual: notallowed,
Predicted: notallowed.
Confidence: 99.94%

Actual: allowed,
Predicted: allowed.
Confidence: 99.52%

Actual: allowed,
Predicted: allowed.
Confidence: 98.38%

Actual: notallowed,
Predicted: notallowed.
Confidence: 95.89%

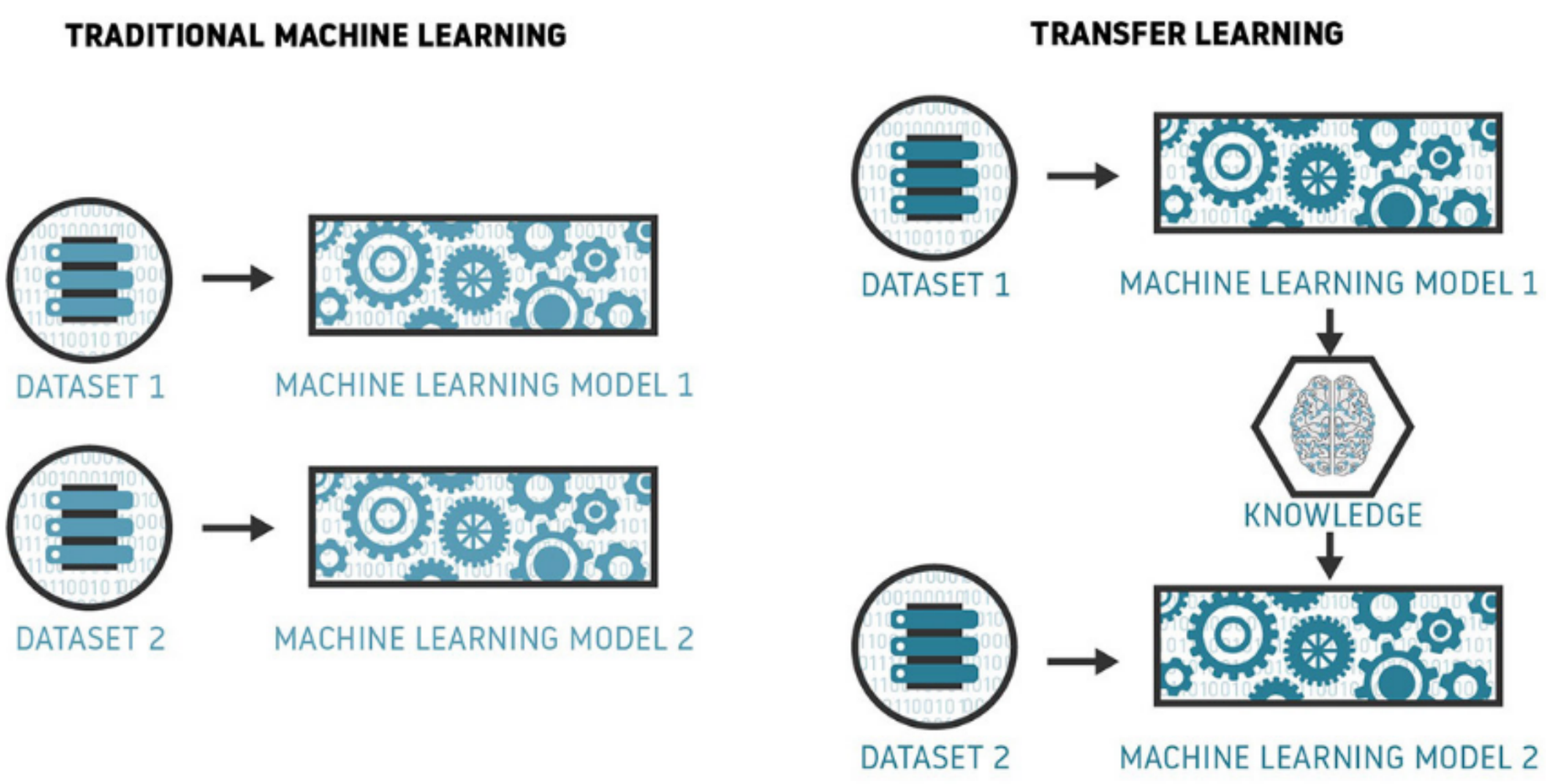
Actual: notallowed,
Predicted: notallowed.
Confidence: 85.19%

Actual: notallowed,
Predicted: notallowed.
Confidence: 98.47%

Actual: allowed,
Predicted: allowed.
Confidence: 100.0%

Transfer Learning

Transfer learning is a technique where a pre-trained model is used as a starting point for a new task. By leveraging the knowledge and learned features of the pre-trained model, the top layers are removed or frozen, new task-specific layers are added, and the model is trained on a new dataset. This approach allows for faster convergence, better performance with limited data, and the ability to leverage well-established pre-trained models. Fine-tuning can be performed by unfreezing some earlier layers for further adaptation. Transfer learning is widely used in computer vision and other domains, enabling practitioners to benefit from existing models and reduce the resources and time required for training.



Why Transfer Learning is necessary?

Transfer learning is necessary because it allows us to leverage pre-trained models and their learned representations to solve new tasks efficiently. It addresses the limitations of limited labeled data and reduces the computational and time requirements of training models from scratch. By utilizing the pre-trained model's knowledge and hierarchical features, we can achieve better performance and generalize well to new tasks. Transfer learning is a practical and effective approach

that accelerates the development and deployment of machine learning solutions by leveraging existing knowledge and representations.

Examples for Transfer Learning

Transfer Learning using VGG16

Step1: Initialize libraries

```
import torch
from torch import nn
from torchvision import models, transforms, datasets
```

Step2: Mount Google Drive

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Step3: Import pre-trained model

```
model = models.vgg16(pretrained=True)
```

Step4: Initialize the parameters

```
num_classes = 2
in_features = model.classifier[6].in_features
model.classifier[6] = nn.Linear(in_features, num_classes)
for param in model.features.parameters():
   param.requires_grad = False
for param in model.classifier[6].parameters():
   param.requires_grad = True
```

Step5: Data Transformation

```
data_transforms = transforms.Compose([
   transforms.Resize((256,256)),
   transforms.ToTensor(),
   transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
])
```

Step6: Initialize the data directory

```
data_dir = '/content/drive/MyDrive/fix'
dataset = datasets.ImageFolder(data_dir, transform=data_transforms)
```

Step7: Train and Test the data

```
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset,
[train_size, test_size])
batch_size = 16
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=batch_size, shuffle=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
for epoch in range(num_epochs):
   train_loss = 0.0
   train_accuracy = 0.0
 model.train()
 for images, labels in train_loader:
      images = images.to(device)
      labels = labels.to(device)

      optimizer.zero_grad()
      outputs = model(images)
      loss = loss_fn(outputs, labels)

      loss.backward()
      optimizer.step()

      train_loss += loss.item() * images.size(0)
      _, predicted = torch.max(outputs, 1)
      train_accuracy += torch.sum(predicted == labels).item()

train_loss = train_loss / len(train_loader.dataset)
train_accuracy = train_accuracy / len(train_loader.dataset)
```

Step8: Calculate the accuracy

```
model.eval()
test_loss = 0.0
test_accuracy = 0.0

with torch.no_grad():
   for images, labels in test_loader:
```

```
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = loss_fn(outputs, labels)

        test_loss += loss.item() * images.size(0)
        _, predicted = torch.max(outputs, 1)
        test_accuracy += torch.sum(predicted == labels).item()
```

```
test_loss = test_loss / len(test_loader.dataset)
test_accuracy = test_accuracy / len(test_loader.dataset)
```

Step9: Print the Test-Accuracy

```
print(test_accuracy)
```

Output:

```
0.9903614457831326
```

Transfer Learning using InceptionNet

Step1: Initialize libraries

```
import torch
from torch import nn
from torch.optim import Adam
from torch.utils.data import DataLoader
from torchvision import datasets, models, transforms
from sklearn.model_selection import train_test_split
from google.colab import drive
```

Step2: Mount Google Drive

```
drive.mount('/content/gdrive')
```

Step3: Data Transformation

```
data_transforms = {
   'train': transforms.Compose([
      transforms.RandomResizedCrop(299),
      transforms.RandomHorizontalFlip(),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225]),
   ]),
   'val': transforms.Compose([
      transforms.Resize(299),
      transforms.CenterCrop(299),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225]),
   ]),
}
```

Step4: Import Data Directory

```
data_dir = '/content/gdrive/MyDrive/fix'
```

Step5: Train and Test the data

```
dataset = datasets.ImageFolder(data_dir, data_transforms['train'])
class_names = dataset.classes
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(dataset,
[train_size, val_size])
```

```
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
```

Step6: Import pre-trained model

```
model = models.inception_v3(pretrained=True)
num_features = model.fc.in_features
```

Output:

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:20
8: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and
may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:22
3: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future.
The current behavior is equivalent to passing
`weights=Inception_V3_Weights.IMAGENET1K_V1`. You can also use
`weights=Inception_V3_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
Downloading:
"https://download.pytorch.org/models/inception_v3_google-0cc3c7bd.pth"
to /root/.cache/torch/hub/checkpoints/inception_v3_google-0cc3c7bd.pth
100%|██████████| 104M/104M [00:00<00:00, 277MB/s]
```

```
model.fc = nn.Linear(num_features, len(class_names))
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters(), lr=learning_rate)
```

Step7: Calculate the accuracy

```
for epoch in range(num_epochs):
   model.train()
   train_loss = 0.0
   train_corrects = 0
for inputs, labels in train_loader:
   inputs = inputs.to(device)
```

```
    labels = labels.to(device)

    optimizer.zero_grad()

    outputs, aux_outputs = model(inputs)
    _, preds = torch.max(outputs, 1)
    loss = criterion(outputs, labels)

    loss.backward()
    optimizer.step()

    train_loss += loss.item() * inputs.size(0)
    train_corrects += torch.sum(preds == labels.data)

    train_loss = train_loss / len(train_dataset)
    train_acc = train_corrects.double() / len(train_dataset)

      print(f"Epoch [{epoch + 1}/{num_epochs}] - "
        f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
```

Output:

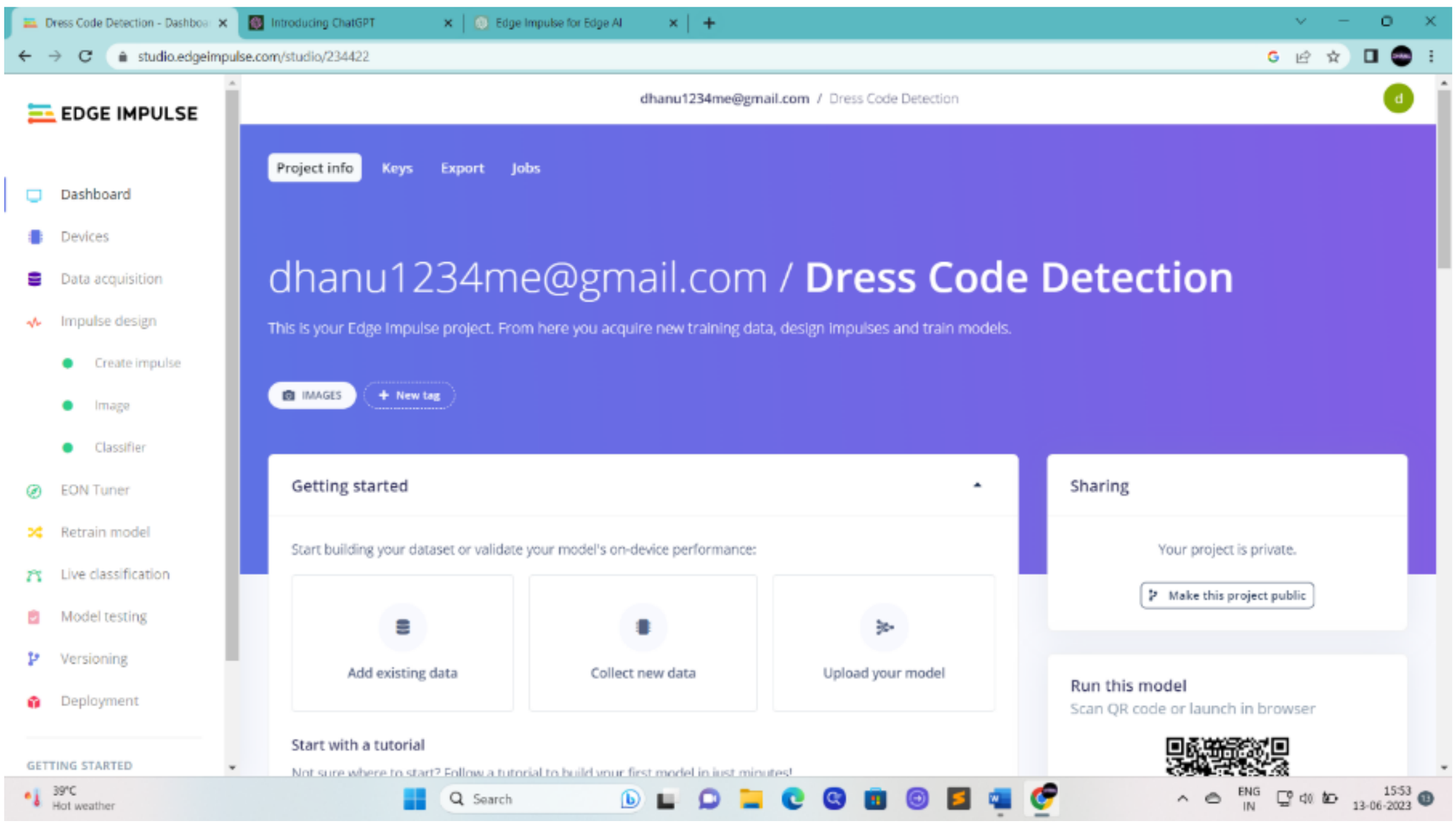
```
Epoch [10/10] - Train Loss: 0.3942, Train Acc: 0.8352
```

Edge Impulse

Edge Impulse is a platform that helps developers create intelligent edge devices by building and deploying machine learning models directly on embedded devices. It simplifies data acquisition, model development, and deployment processes. With Edge Impulse, developers can collect and preprocess sensor data, build models using popular frameworks, deploy models on edge devices, optimize for resource constraints, and monitor and manage deployed models remotely. The platform enables local processing and inference, providing benefits such as lower latency and increased privacy.

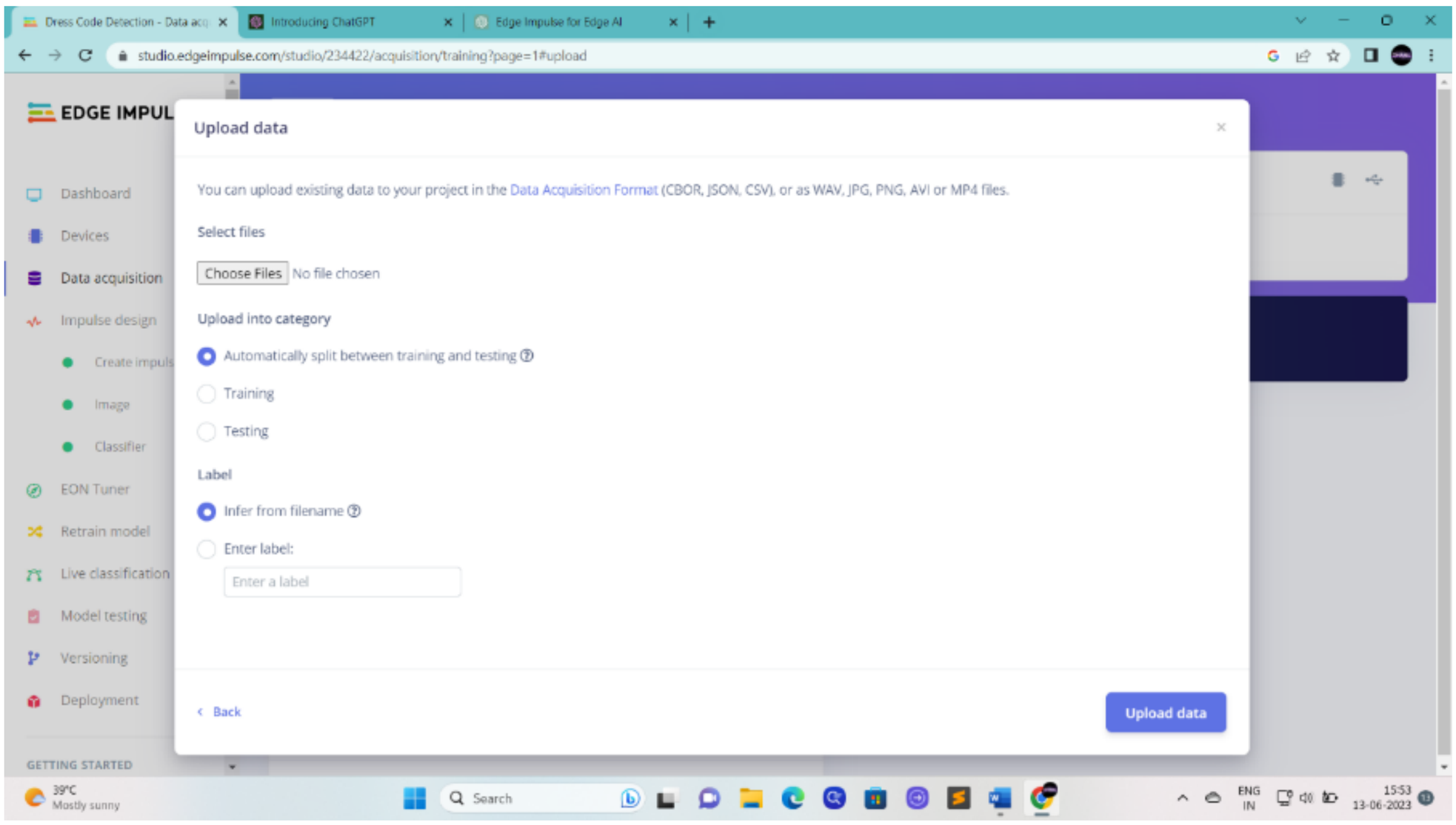
Dress code detection using Edge Impulse

Step1: Create the project repository with the project name.

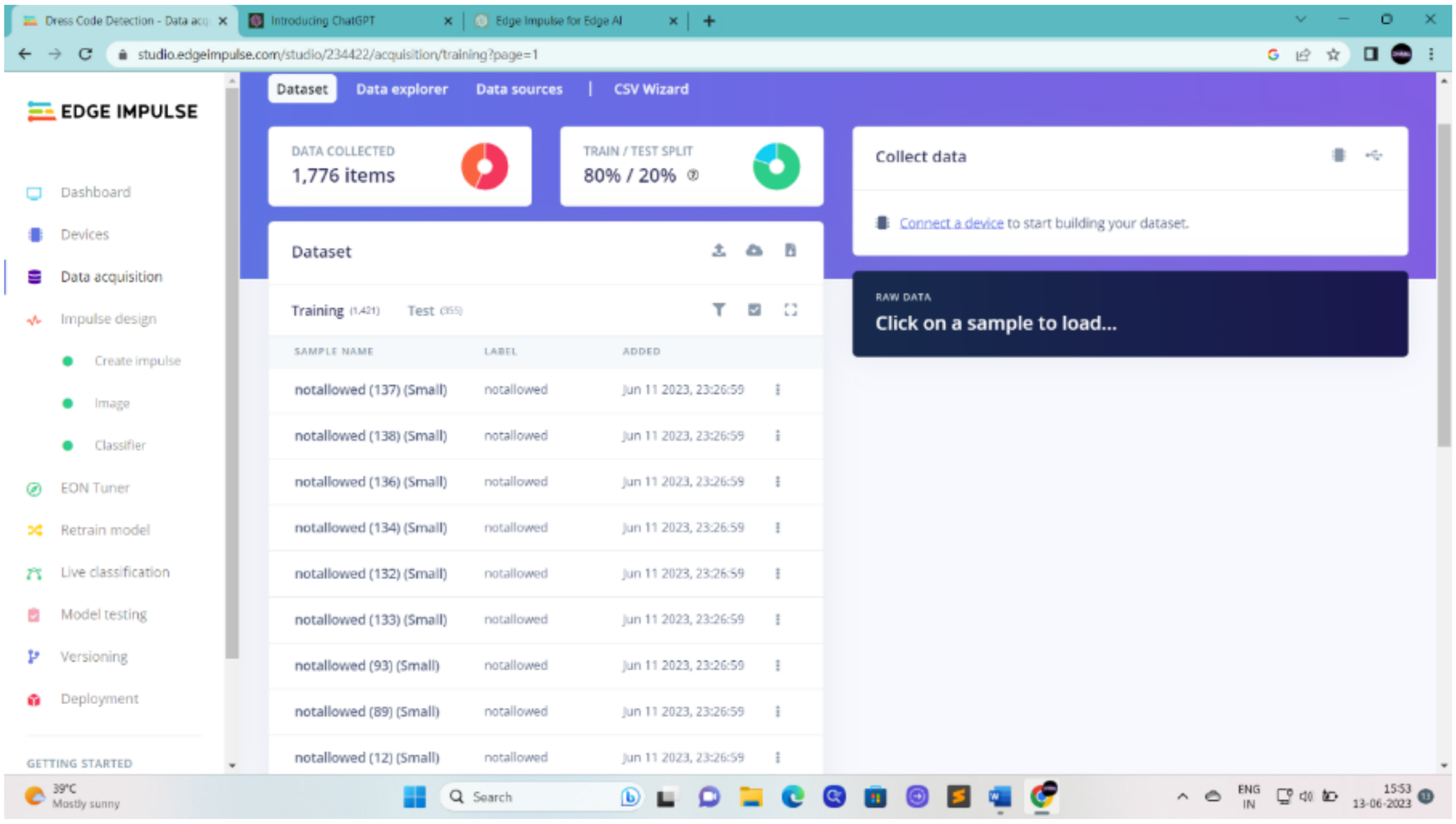


Step2: Data acquisition

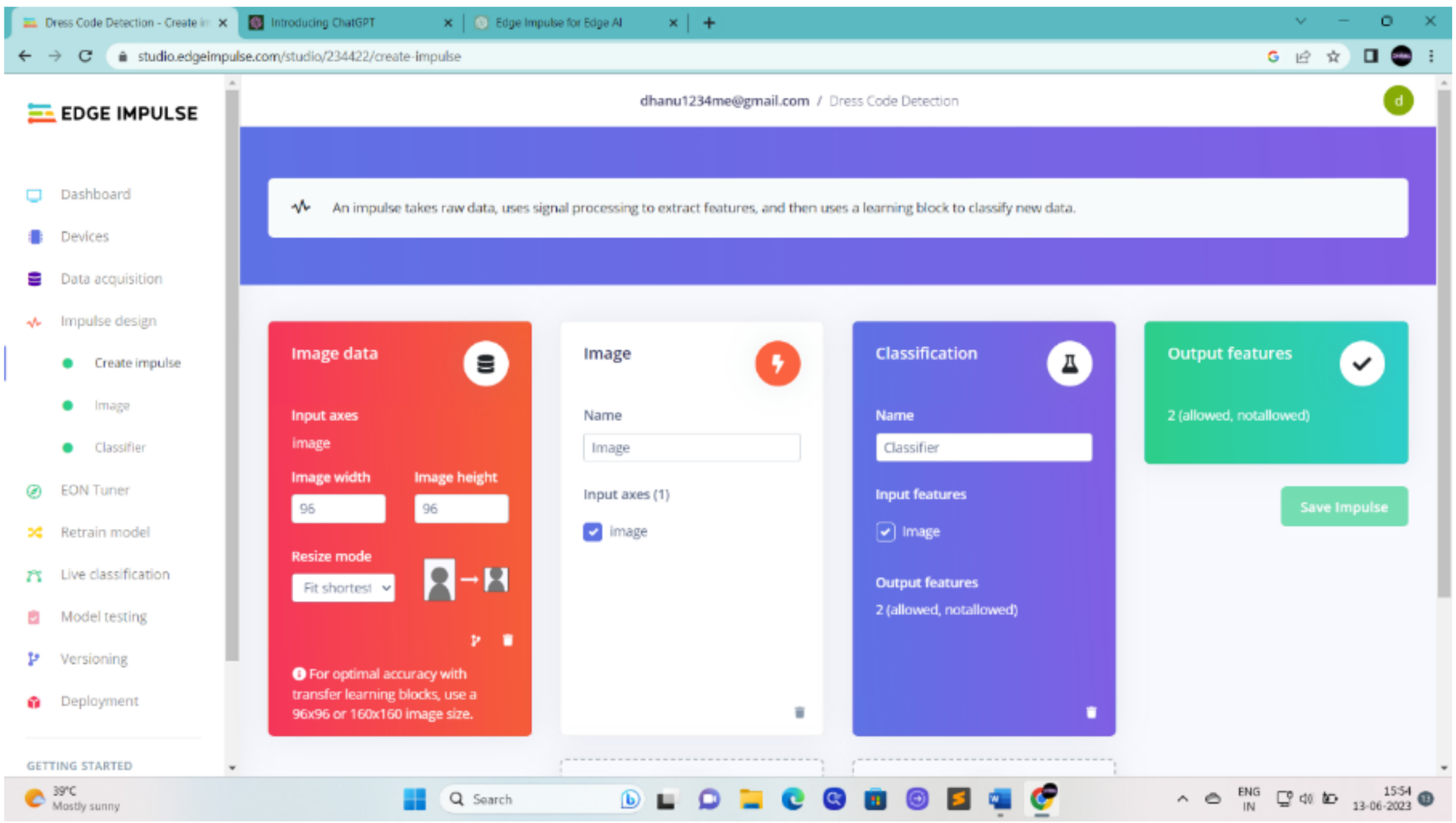
2.1 upload the data with lables and divide the data into training and testing



2.2 After uploading data

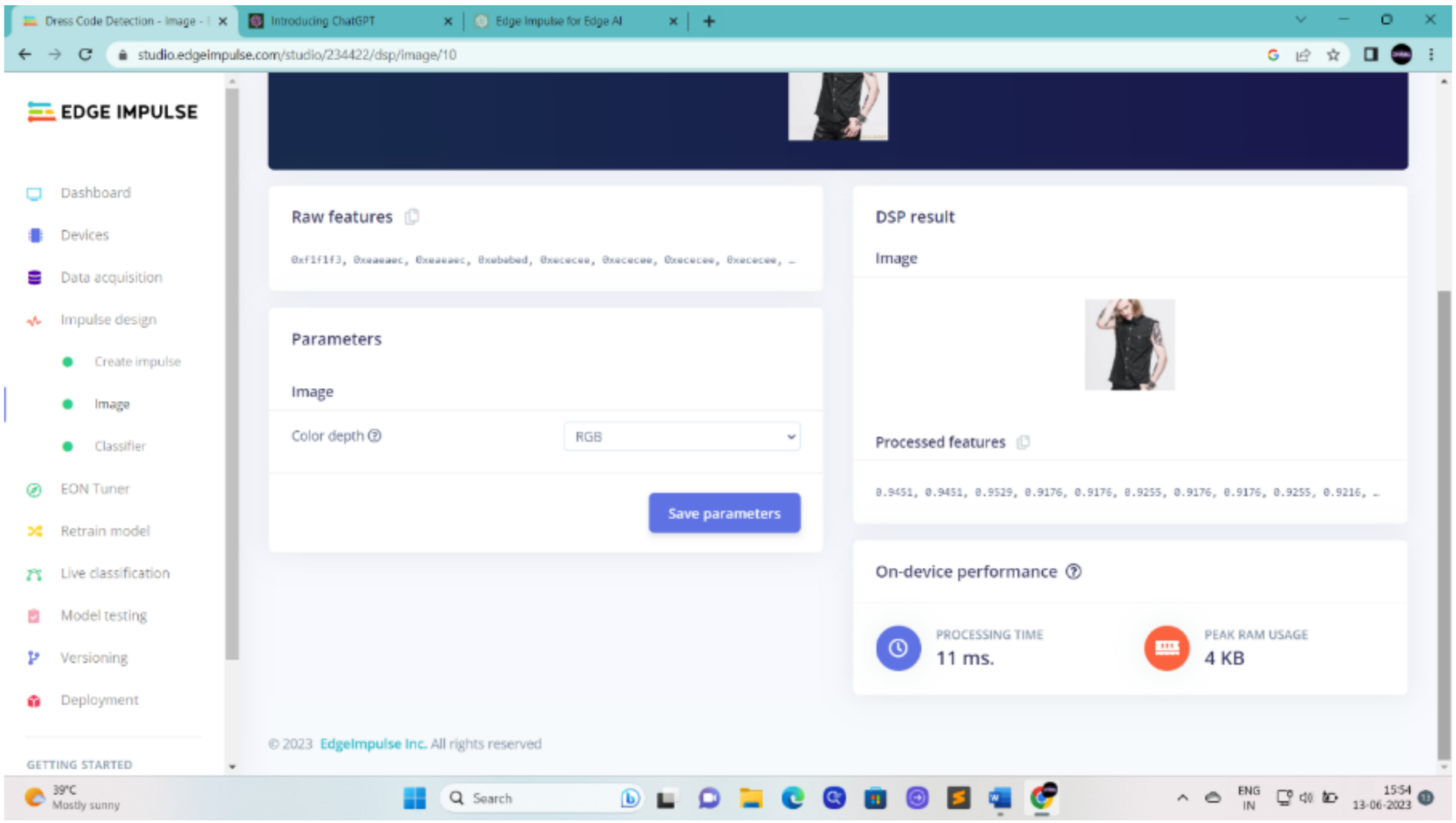


Step3: Create impulse in impulse design

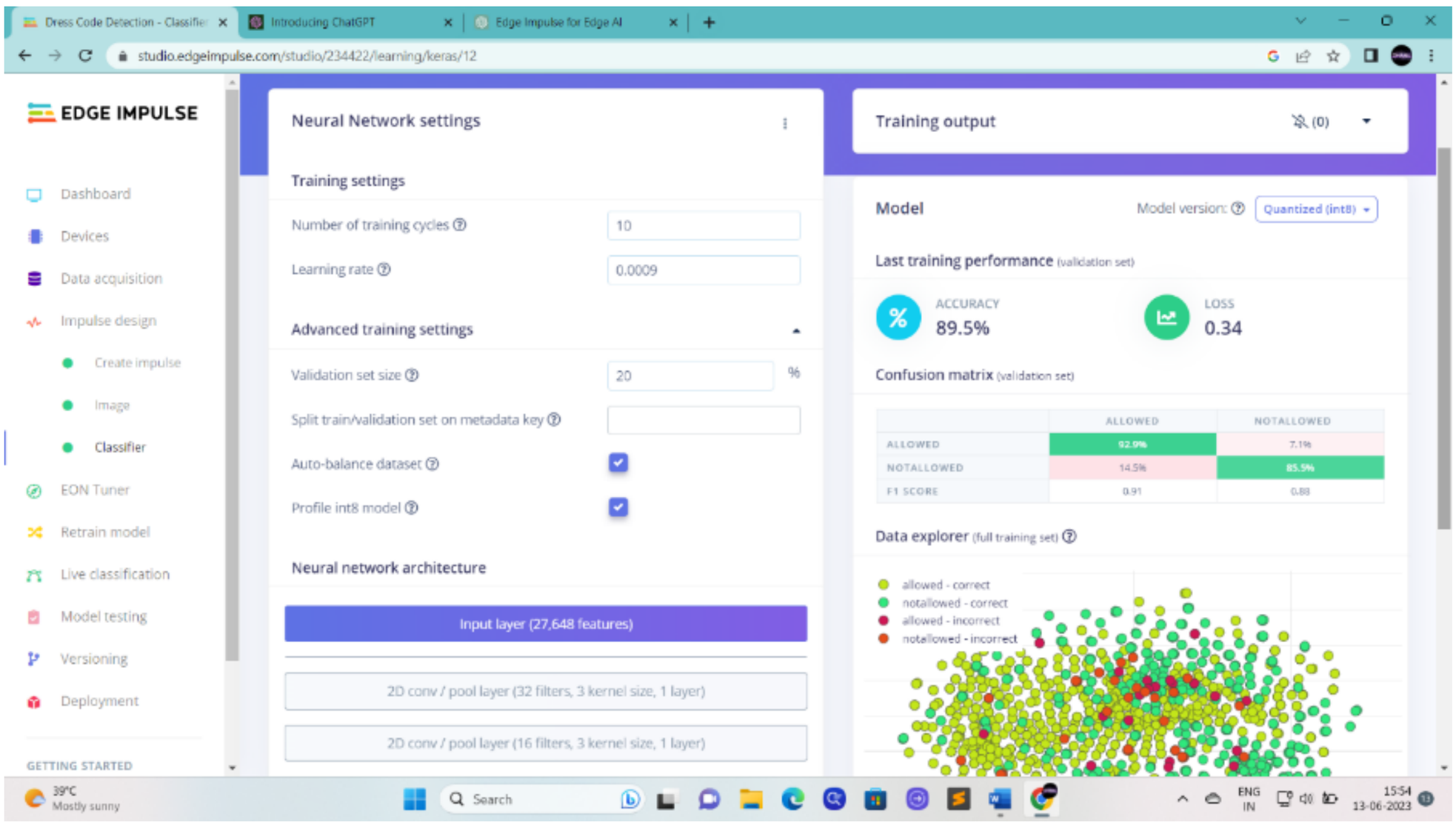


Here, we are defining image size and name of the classification.

Step4: Feature generation of images



Step5: Classification of images



Here we are getting 89.5% accuracy