code-final

May 12, 2022

0.1 Face Emotion Recognition from Facial Expressions using Deep leaning Techniques

0.2 Abstract

Facial Emotion Recognition is a basis of human interface technology with applications in society ranging from social robots to video games that modify difficulty levels based on the player's facial expressions. Many academics use deep learning approaches to develop models with higher accuracy, particularly Convolutional Neural Networks (CNN), which excel at pattern recognition and image processing. To improve the accuracy of the dataset, this study uses advanced Efficientnet architecture (FER2013). When compared to Resnet and the VGG16 model, Efficientnet, one of the proposed designs, has the highest accuracy.

0.3 Problem Definition

The stimulation of particular facial muscles allows humans to transmit their emotions through their faces. These expressions are sometimes accurate, but they can be difficult to grasp; signals in a face expression can contain a wealth of information about their emotional state. Humans are generally adept at recognizing and comprehending the emotions of others. "Anger,", "contempt, ","disgust, ", "fear, ","happiness,", "sadness, ", and "surprise," are among the seven emotions of human face that can be roughly classified. This project will use Python to implement the source code for a facial expression recognition system. We'll use the deep neural network to solve complicated issues like facial expression detection. In today's world of computer vision, facial expression detection systems are extremely important.

It helps to comprehend the underlying meaning of human-machine interaction, which is useful in a variety of applications such as diagnosing mental diseases, assessing mental states, detecting lies, and so on. The task of accurately recognising human expressions remains difficult. Deep learning techniques, which are particularly successful for image processing, are used here. The FER2013 dataset is used for analysis, and popular "Deep Learning (DL)" frameworks for example OpenCV and Efficientnet are employed to accurately detect facial emotions.

1 Introduction

As model and training data sizes grow bigger, deep network training performance gets extremely important. GPT-3 (Brown et al., 2020), for instance, demonstrates outstanding capabilities in few class classification from a much larger size input data and more training data, but still it takes many days of work with dozens of 'GPUs' that building problematic toward retraining. Consequently, training effectiveness have received a lot of attention. For illustration, NFNets (Brock et al.,

2021) intends to enhance subsequent revisions by trying to reduce the time-consuming convolution layers. Some researcher ((Srinivas et al., 2021) works to improve learning rate via incorporating responsiveness network layers into deep convolutional; and Vision Transformers (Dosovitskiy et al., 2021) utilizes Transformer components to enhance training efficiency on huge datasets. These methods, however, constantly come with many pitfalls.

Techniques frequently come from high cost whenever dealing with large feature sizes (b). On increase all learning rate and feature efficiency, designers integrate training-aware neural architecture search (NAS) alongside scalability inside this work. We start by systematically analyzing the learning barriers in EfficientNets, considering its feature efficiency (Tan & Le, 2019a). In EfficientNets, we noticed below points,

(1) Learning rate for large image sizes is slow,

(2) Depthwise convolution layers in previous layer are slow.

(3) Correspondingly speeding up every stage is expensive.

Findings from this study, we construct a search area which includes additional procedures like 'Fused-MBConv' method, then apply 'training-aware- NAS' then scale for optimize training model accuracy, speed of the training process, as well as input feature/parameter size simultaneously.

Researchers discovered systems, called EfficientNetV2, can train the model up to four times quicker than traditional methods, by using 6.8 times fewer features. They can speed it up our training with gradually raising the image size while doing it. Several past studies have utilized lower image sizes in training, like (Howard, 2018) progressive resizing, (Touvron et al., 2019) FixRes, as well as (Hoffer et al., 2019) Mix & Match approaches. However, they typically maintain the same training process for all image sizes, which leads to loss of accuracy. Researchers suggest that using same parameterization/ training process for various image sizes is not perfect: small image size result in smaller network capacity and so needs small training data; high image size, on either hand, requires stronger training data to prevent fitting problem. Researchers develop an enhanced dynamic learning method based on this understanding. They training the model with lesser input image sizes also thin normalization (for example, loss as well as data augmentation) in the initial training epochs, then gradually increase size of the image and add deeper regularization.

Their strategy relies on continuous scaling (Howard, 2018), but can increase speed training without sacrificing quality by constantly adjusting regularisation. The advanced EfficientNetV2 accomplishes robust performance on various image dataset such as CIFAR100, ImageNet, Cars, CIFAR-10 and Flowers models since for faster progressive learning. They achieved 85.7 % of accuracy on ImageNet despite training the model with 3 times to 9times faster and being 6.8x less than earlier models. Also it's easier to train systems on huge datasets with our EfficientNetV2 and progressive learning.

As instance, ImageNet21k (Russakovsky et al., 2015) is roughly 10 times greater as ImageNet ILSVRC2012, however the EfficientNetV2 could accomplish the train in couple of days only with 32 TPUv3 cores. The EfficientNetV2 obtains 87.3% first rank accuracy on 'ImageNet- ILSVRC2012' dataset, which following learning methodologies on the available ImageNet21k, beating the other method such as recent 'ViT-L/16' by 2.0 % error rate while training 5 times to 11 times faster.

2 We consider 3 contributions

• EfficientNetV2, a young family of shorter as well as faster models, is presented. EfficientNetV2 improves previous systems in terms of training time and feature optimization, due to its training-aware NAS and scaling.

• This work proposed an upgraded continuous active learning system, which changes parameterization and image size flexibly. We demonstrate that this really enhances accuracy even while accelerating training.

• We show up to 11 times faster training speed also up to 6.8 times improved feature optimization efficiency on existing dataset such ImageNet, CIFAR, Cars, and Flowers.

3 Design

Inside this section, we examine the EfficientNet's 'training constraints' and its 'training-aware NAS plus scaling' and 'EfficientNetV2' algorithms (Tan & Le, 2019a). This author established a collection of methods which are tuned for 'FLOPs' then feature/parameter optimization. This system exploits NAS technique to determine basis of 'EfficientNet-B0', which has a good balance of accuracy with 'FLOPs' model and the initial system is then enlarged using a complex strategy in scaling process to generate the B1-B7 paradigm group. Despite recent reports of big changes in training but rather test time, this system are regularly weaker than EfficientNet method in terms of features/parameters and effectiveness of FLOPs process. Our goal of this work is to improve learning rate simultaneously maintaining feature quality. Researchers evaluate the learning constraints of EfficientNet (Tan & Le, 2019a), that will be referred to as EfficientNetV1 from now on, as well as a few simple techniques for improving learning rate. Earlier studies have shown EfficientNet's (Radosavovic et al., 2020) the big image size causes foremost memory consumption. Researchers have to train such systems with smaller batches so because overall memory on GPU/TPU is limited, that significantly starts to slow down the training. FixRes is (Touvron et al., 2019) an easily enhancement network that utilizes a smaller size of input image for training process than for inference process. Low resolution image sizes lead to fewer calculations and enable for higher batch sizes, improving learning rate by up to 2.2 times. According to Brock et al. (2021), employing a lower size of the image for learning improves the accuracy significantly. We do not adjust any layers after training, unlike (Touvron et al., 2019).

EfficientNet's large depth - wise separable convolution layers are just another training issue (Sifre, 2014). While depth - wise separable convolution layers used less parameters and FLOPs over regular convolution layers, they can't usually make utilisation modern processors. To better leverage portable or server accelerators, Fused-MBConv suggested by Gupta & Tan (2019) and this method was used by Gupta & Akin (2020), Xiong et al., (2020) and Li et al., (2021). Sandler et al (2018) and Tan & Le, (2019) substitutes the depthwise conv3x3 and expansion conv1x1 with a single normal conv3x3 by MBConv method.

This work progressively update the old MBConv in EfficientNet-B4 by FusedMBConv to evaluate the two basic components. FusedMBConv can increase training learning rate with a small impact on input features and FLOPs when it is used in preliminary phase 1-3, but since all units are replaced by the Fused-MBConv model (stage 1-7), which considerably rises features/parameters as well as the slowing down the learning process by using FLOPs. Determine the best combination/hybrid of different basic components, MBConv and Fused-MBConv, is hard, that's why we adopted neural

network models investigation to find the optimum combination automatically.

Using only a simple composite scale rule, EfficientNet properly expands all stages. Whenever the depth coefficient is 2, for examples, the number of layers in all phases of the system increases. However, but not all of those stages make contributions to train rate and feature effectiveness. We'll employ a – anti scaling strategy inside this work to progressively add additional layers to later steps. EfficientNets often forcefully scale up size of the image that process primary reason for huge memory utilization also it slow the training process. This issues are solved by the scaling the rule significantly also confine the maximal size of the input image to a lower rate. They utilize similar compounds scaling as (Tan & Le, 2019a) to massively increase 'EfficientNetV2-S' to get 'EfficientNetV2-M/L', with some adjustments. This work gradually add additional network layers to final phases to boost network infrastructure without imposing much execution complexity.

They exhibit the following curves for EfficientNet, one can be trained (Tan & Le, 2019a) by both the normal inference size and then another one is trained by a slightly smaller size of image, comparable to (Touvron et al., 2019) EfficientNetV2 system and (Brock et al., 2021) NFNet method. With the exception of NFNets, that are trained using 360 epochs, most model can be trained using 350 epochs, culminating in an equal amount of training cycles. Amazingly, they discover so when properly trained, EfficientNets typically ensure a better achievement ratio. Most importantly, our suggested EfficientNetV2 system trains significantly faster than in other recent models due to its training-aware NAS & scaling. Those results correspond previous inference conclusions.

4 Emotion Recognition requirements and specification

- Jupyter is used to run the planned system, which includes Anaconda 3, OpenCV, and pytorch.

- The entire experiment is run on an 8GB of RAM with "Intel(R)- Core(TM)-i5-3230M processor" CPU machine with and a 64-bit Windows Operating System.

- Use pip install comments in -r requirements.txt for installing the all required packages for this work.

5 Objective of this project

The purpose of project work is to improve a DL- established scheme for understanding people's emotions that can take human facial images and recognize and classify them into seven different expressions. The main goal is to collect an image using FER and then use Deep leaning methods to detect the emotion of the person present from the input photographs.

- Create a solid framework for recognizing "happy," "sad," "surprise," "fear," "rage," "disgust," and neutral facial expressions automatically.

- To research and apply existing methods for identifying human face expressions, such as Resnet and VGG16.

- To use an advanced deep learning algorithm to identifying human face expressions with a high level of accuracy value and efficiency.

- To compare and assess the recognition rate of facial expressions utilizing existing and Efficientnet V2 approaches.

• Consequently, the general goal is to research and generate effective deep learning approaches for automatic face recognition.

5.1 Computer Vision and Face Emotions Recognition

5.2 Computer Vision

It is a key component of Artificial Intelligence (AI) approaches, which tries to create intelligent algorithms that can perceive images as if they were seen by a person. It is a way of converting input data from an image into a decision result in which the transformation outcomes have a desire to achieve a goal.

Computer vision is the ability of computers to comprehend digitally acquired images. Computer vision is an important technical solution for many complicated problems, but it is not without its challenges. It creates human-like vision abilities for numerous applications by integrating computer vision with DL systems. Computer vision now has an extensive variety of applications in the real world, including retail, finance, construction, sports, automobile, agriculture, insurance, and more. Some use cases include computer vision projects that have a positive impact on the world.

One of the most important aspects of recognizing human emotions is the ability to recognize facial expressions. This field is part of the human-computer interaction study that benefits society. This field research could lead to the development of humanoid robots that can interact with humans and respond to human emotion, such as those used in child daycare. It can also be used in healthcare to detect patients' emotions in order to determine their mental health status. It can also be employed in the entertainment sector, for example, in video games that modify the game's flow based on the person's facial gestures.

5.3 Methodology

It is critical to design advanced methods to address the aforementioned challenge in order to acquire the best solution. We are going to use Resnet VGG16 and Efficientnet_V2 for human facial expressions recognition to produce a valid solution in this project.

5.4 Data pre-processing:

FER datasets are available online, however size of input image, colour, and image format, also labelling and directory arrangement, and differ greatly. To resolve these disparities, we simply divided the input datasets into seven catalogues (one for each class). During training, we loaded images from disc in batches (to avoid memory overflow) and used data generators to resize and format the images automatically.

Because it comprises non-face shots, text images, drowsy faces, discernible profile photographs, and a large number of incorrectly labelled images, FER2013 suffers from severe crowdsourcing. As a result, utilizing this dataset, the overall accuracy of facial expression categorization is less than 65%. Data augmentation approaches were used to overcome this challenge by artificially creating training images using various processing methods such as random rotation, shifts, shear, and flips, among others.

5.5 Data Agumentation

Data augmentation is a set of approaches for artificially increasing the amount of data by creating more data points from current data. This can include making minor changes to data or using deep learning models to generate more data points. ImageDataGenerator enables it simple to create a real-time data augmentation image generator that generates batches of image data. The simplest basic code for creating and configuring ImageDataGenerator and training enhanced images on deep neural networks. It is a method of modifying existing data in order to generate new data for the model training process.

To overcome this problem, data augmentation techniques were employed to create training images artificially utilising various processing methods such as random rotation, "shifts,", "shear,", then ",flips",, among others. Data augmentation is a collection of approaches for insincerely cumulative the quantity of image/features data by creating more data-points after current features. This can include making minor changes to data or using deep learning models to generate more data points. ImageDataGenerator makes creating a better picture generator easier. ImageDataGenerator produces image data in batches and enhances it in real time. The most basic code for setting up ImageDataGenerator and training enhanced photos with deep neural networks. It's a technique for altering existing data in order to generate new data for model training. To put it another way, it's a method for artificially boosting the dataset available for deep learning model training. The following are the advantages of data augmentation methods:

1. Improving the accuracy of model predictions.
2. Size of Training data is increased in the DL models.
3. Data over-fitting (a statistical error in which a function is too closely related to a limited group of data points) and data variability are reduced.
4. Improving the model's capacity to generalize.
5. Assisting with concerns of categorization class imbalance.
6. Lowering the costs of data collecting and labelling.
7. Enables the prediction of uncommon events.

Image transformations, such as rotation, flipping, and cropping, are among the most basic image manipulations. The majority of these approaches directly change photographs and are simple to use.

6 Flipping

We can horizontally and vertically flip images. Vertical flips are not supported by all frameworks. A vertical flip, on the other hand, is the same as rotating an image 180 degrees before flipping it horizontally. Here are some examples of photos that have been flipped.

6.1 Rotation

It's vital to remember that following rotation, input image size/dimensions cannot be conserved. If your image is square, rotate it at right angles to maintain the same size. Turning it "180"degrees will maintain it the same size if it's a rectangle. As the image is rotated at finer angles, the final image size will change.

6.2 Scaling Ratio

It's conceivable to zoom in or out of the image. The final image size will be greater than the starting image size after scaling outward. Most image frameworks extract a chunk of the new image that is the same size as the old one. In the following session, we'll look into scaling inward, which reduces the image size and forces us to make assumptions about what is beyond the limit.

6.3 Noise injection

Noise injection is the process of injecting a matrix of random values, often from a Gaussian distribution. Moreno-Barea et al. evaluated noise injection on nine datasets from the UCI collection. CNNs can learn more robust features by adding noise to pictures. Geometric adjustments can successfully address positional biases in training data. Training data distributions may differ from testing data distributions due to a variety of choices. If positional biases exist, such as in a facial recognition dataset where every face is properly centred in the image, geometric modifications are an excellent alternative. Geometric transformations are useful not only for eliminating positional biases, but also because they are simple to use.

6.4 Color space

A tensor is used to store the dimensions (height-width-color channels) of digital image data. Another useful method is to perform augmentations in the colour channels space. Isolating a single colour channel, such as R, G, or B, can be used for colour augmentation.By isolating one matrix and adding two zero matrices from the other colour channels, a picture can be swiftly transformed into its representation in one colour channel.

Moreover, using basic matrix operations, the RGB values can be easily changed to enhance or lessen the image's brightness. More extensive colour augmentations can be achieved by creating a colour histogram that describes the image. The lighting effects produced by changing the intensity levels in these histograms are similar to those found in photo editing software.

7 Contrast Stretching

It is a technique that examines the distribution of pixel densities in an image. Contrast is defined as the difference between light and dark hues in an image. Control the amount of jitter in contrast with the contrast parameter, which ranges from 0 (no change) to 1. (Potentially large change). Contrast does not define whether the augmented image's contrast will be higher or lower, only the effect's possible strength.

8 Resize

The shortest edge is scaled to the specified size, while the longest edge is automatically adjusted to maintain the aspect ratio of the supplied image. It adjusts both height and width to the specified size, even if the aspect ratio is not preserved.

9 Cropping

Unlike scaling, we just sample a piece of the original input image on unsystematic, which is then resized to the original format of image size. Random cropping is the name given to this technique. Here are some random cropping instances. The difference between this approach and scaling can be seen if you look attentively.

10 Sharpening

Sharpening input images/photos for preprocessing/Data Augmentation may consequence in additional facts about objects of area being captured. The traditional methods of applying kernel filters on photos include sharpening and blurring.

11 Translation

During translation, the image is simply moved in the X or Y direction (or both). In the next example, we assume the image has a black background beyond its boundaries and translate it correctly. This form of augmentation is incredibly useful because most things can be found almost anywhere in the image. Your convolutional neural network is presently looking in every direction conceivable.

12 Use of ToTensor and normalization:

The process of converting the original values of picture pixels to a new set of values is known as image transformation. Converting a photo into a PyTorch tensor is one of the processes we conduct on photographs. The pixel values are scaled between 0.0 and 1.0 when converting an image to a PyTorch tensor. When using PyTorch, transforms have used to achieve this transition. ToTensor().

Normalizing images is a valuable practise when working with deep neural networks. Normalizing images includes converting them to values such as 0.0 and 1.0 for the image's mean and standard deviation, respectively. Remove the network mean vale from each of the input channel of the system and divide the outcome by using the number of input channels to achieve this.

12.1 Dataset Description

FERR2013 is a collection of 48x48 facial photos with a variety of expressions. Faces in the Fer2013 dataset were automatically collected. As a result, the captured human face is centred then occupies the similer amount of size in each frame. The results of each emotion's Google image search, as well as alternatives for the feelings, were compiled into this dataset.

Fer2013 datasets are used in a variety of applications that involve making decisions about facial expression detection tasks. Traditional methods, deep learning, pre-trained models, and ensemble neural networks techniques are used to handle these difficulties.

13 Deep Learning Algorithms

Furthermore, feature selection has a substantial impact on the performance of machine learning algorithms, potentially resulting in incorrect class separation. DL, unlike standard machine learning

methods, can automate the learning of feature sets for a wide range of tasks. With DL, learning and classification may be done in one step. DL has been an exceedingly popular type of ML approach in recent years because to the massive expansion and innovation of the field of big data. Its novel performance for a number of machine learning tasks is still under development, but it has streamlined the improvement of so many AI learning domains for example input picture/ digital image super-resolution, main object discovery, and input image gratitude. On tasks like image categorization, DL performance has recently surpassed human performance.

CNNs are a kind of DL that uses convolutional layers to filter meaningful information. The convolutional filters of the input image are used to calculate the yield of neurons associated to input image local area of interest to extract the spatial and temporal aspects of the image. In the convolutional layers of CNN, a weight sharing mechanism is used to lower the total number of input parameters. The CNN approach is often made up of three components:

- Use a convolutional layer to learn spatial and temporal features.

- A subsampling layer (max-pooling) to reduce or down sample the dimensionality of an input image.

- A fully connected (FC) layer for categorizing the input image into different classifications.

- Convolutional layer: Convolutional layers are created by applying a series of filters (also known as kernels) to an input image. The feature map created by using the convolutional layer is a representation of the input image with the filters applied. Convolutional layers can be stacked to build increasingly complicated models that can learn more complex features from photos.

- Pooling layer: In deep learning, pooling layers are a sort of convolutional layer. The spatial size of the input is reduced by pooling layers, making it easier to process and needing less memory. Pooling also minimises the amount of factors in the training process and speeds it up. Pooling can be divided into two types: maximal pooling and average pooling. The maximum value from each feature map is used in max pooling, while the average value is used in average pooling. After convolutional layers, pooling layers are typically used to minimise the amount of the input before it is delivered to a fully connected layer.

- Fully connected layer: These are one of the most fundamental types of layers of CNN method. Each neuron in this layer is entirely coupled to all other neuron of previous layer, as the name implies. This layer employed at the end of a CNN method when the goal is to use the features learned by the preceding layers to create predictions. If we were using a CNN to categorise photographs of animals, for example, the last fully connected layer could use the information learnt by the preceding layers to classify an image as including a dog, cat, bird, or other animal.

13.1 VGG (Visual Geometry Group) -16

The International ILSVRC is a "computer vision," challenge held every year. Teams fight for two jobs each year. The first step is to locate objects within the image using the 200 classes, which is referred to as the object's local action. Teams fight for two jobs each year. The first step is to locate objects within the image using the 200 classes, which is referred to as the object's local action.

13.2 Architecture:

Imaging partition is the next step, which entails sorting images into one of 1000 categories. In their paper "VERY DEEP COVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION," Karen Simonyan and Andrew Zisserman of Oxford University's Visual Geometry Group Lab suggested VGG 16 in 2014. In the 2014 ILSVRC event, this model took first and second place in the aforementioned categories.

Following Steps 2 and 2, there is a large integration layer that is similar to the prior layer. The filter size (3, 3) and filter 256 are converted using two layers. There are two sets of three convolution layers after that, followed by a maximum pool layer. Each has the same number of filters and pads (3, 3).

We employ size 3 * 3 filters instead of 11 * 11 on AlexNet and 7 * 7 on ZF-Net for these convolution layers and larger integration. It also changes the amount of input channels at different levels by using the 1 * 1 pixel. After each layer of rotation, a 1 pixel termination (parallel termination) is conducted to prevent the image's local element. We uncovered (7, 7, 512) a feature map after a lot of convolution and a max-pooling layer. We generate a component (1, 25088) vector by decrypting this output. There are three fully linked layers after that: the first subtracts the vector (1, 4096) from the input from the last element vector, the second extracts a vector size (1, 4096), and the third layer subtracts the vector (1, 4096) from the input after the last reduce the computational.

Following the testing of the five vector segment classifications. The unlock function for all concealed layers is ReLU. ReLU is mathematically sound since it speeds up learning while lowering the possibility of gradient collapse. The VGG-16 network was trained using image databases of various sizes. The VGG-16 network provides good accuracies even when the picture data sets are small because to its thorough training. The VGG-16 network has 16 convolution layers and a 33 percent receptive field. There are a total of 5 of these levels, with a Max pooling layer of size 22. After the last Max pooling layer, there are three fully connected layers. The next three layers are all linked together. The softmax classifer is used as the final layer. All buried layers receive ReLu activation. The VGG-16 architecture is depicted schematically.

The input image for VGG-19 in this work is 48x48 pixels in size. The input image is distributed over a series of convolution network layers, including 3x3 filters, 19 weight layers, 16 layers of convolution with 3x3 filter size, and 3 layers of completely connected, followed by a stack of convolutional layers. Each of the first two completely connected layers has 4096 channels, whereas the third uses a seven-way classification strategy and hence has seven channels (every class has a channel).

13.3 RESNET 18

Following AlexNet's historic victory in the LSVRC2012 segmentation competition, the deep Residual Network has definitely been a key milestone in the computer / social learning process of in-depth learning during the previous few years. You can train hundreds or thousands of layers with ResNet and still achieve amazing results. Many computer vision applications that do not require image classification, such as object detection and face recognition, benefit from its powerful representation capabilities.

Since it originally made news in 2015, many in the research community have looked into the secrets of ResNet's success, and several architectural enhancements have been devised. The first section of this essay will provide some background information for anyone unfamiliar with ResNet, and the second will address some recent publications I've read concerning the various types and

classifications. If a specified volume is given, a single-layer feedforward network can represent any function, according to the universal approximation theorem. However, the layer could be quite large, and the network could suffer from data overload. As a result, there is widespread agreement among scientists that our network design should be improved. According to AlexNet, CNN's present design is becoming increasingly sophisticated. VGG and GoogleNet (also known as Inception v1) both have 19 and 22 layers to convert, respectively, although AlexNet just has five. In contrast, increasing the network's depth accomplishes more than just layering layers.

The well-known gradient problem makes deep networks difficult to train: as the gradient is propagated back to previous layers, duplication can cause the gradient to decline endlessly. As a result, as the network goes deeper, its performance depletes or rapidly diminishes. There were a few approaches to dealing with gradient loss before ResNet, such as adding auxiliary losses to the middle layer as additional monitoring, but none of them seemed to solve the problem permanently. According to the authors, packaging layers should have no impact on network performance, according to the authors, because we can just accumulate identifiable mappings (the blank layer) in the existing network, and the final structures will do the same. As a result, a DL would not yield more training error rate than shallow models.

It believe that matching loaded model layers to the residual map is informal than permitting the exact layout map you want. And the last block indicated earlier clearly allows it to do so. In reality, Highway Network was the first to use gateway shortcut connections, not ResNet. These settings are controlled by the quantity of data that can be sent via the shortcut. The amount of information that travels to the next stage of the Short-Term Memory Cell (LSTM), which is based on the same idea, is determined by a parameter forgetting gate. As a result, ResNet might be considered a one-of-a-kind Highway Network story. ResNet50 is divided into two types: ResNet50 and SEResnet50. ResNet50 is a shortened version of a 50-layer residual network. Resnet50 is comparable. Resnet50 is similar to VGG-16 with the exception that it adds a supplementary individuality plotting ability.

14 EFFCIENTNET

15 EfficientNetV2

A revolutionary family of deep convolutional having quicker training speed and better parameter optimization than past versions, is presented in this work. We employ a hybrid of 'training-aware' neural network learning models for search as well as scale for enhancing learning rate and parameter optimization in these networks. The models were designed in a search space that were extended to include new events like Fused-MBConv. EfficientNetV2 networks train up to 6.8 times quicker than province models, as per our experiments. Researchers can speed up the training by steadily increasing the size of the image during it, although often this results in the loss of accuracy. To cover the loss of accuracy, we propose an improved progressive learning method which balances training data (e.g. data augmentation) and input image pixel density.

16 Transfer learning

Here, we use the knowledge of a model that has collected training for a particular task to solve a different but related task. The model can benefit from the lessons learned from the previous work so that the new one learns quickly. Let's make an example here and say you want to see the dogs

in the pictures. On the internet you find a model that can see cats. Since this is the same job enough, you take a few pictures of your dog and retrain the model from seeing the dogs.

There are actually two types of learning transfer, feature removal and fine adjustment. Both methods usually follow the same procedure: Launch a pre-trained model (the model we want to learn from) Rearrange the final layers to have the same output value as the category included in the new database Explain what layers we want to review Train to new database

17 Feature extraction

LConsider the creation of a convolutional neural network with a dense layer and a single output neuron as filters. The network has been taught to predict the presence of a cat in an image. We'll need a large data set (photos with and without cats), and the training process will take a lengthy time. This is referred to as "pre-training."

Then there's the enjoyable part. We retrain the network with a data set of a small image with dogs this time. All layers except the exit layer get "snow" during training. This means that throughout training, we do not renew. Following training, the network eliminates the possibility of the dog appearing in the image. This training will be quicker than the initial training. We can also "release" the last two levels, the output and the compact layer, voluntarily. This is dependent on how much data we have. If we just have a tiny amount of data, we may only train the final layer.

18 Transfer learning example in pytorch

This work differs from that of my previous example. A model to see which dogs and cats are in which photographs. For the code to work you will need to organize your data in the following format:

19 implementation

Step1: First, we need to install the required libraries for building the network.

Step2: Import the required machine learning libraries and pytorch frame work to evaluate the model efficiency.

Step3: Identifies the number of processor/thread available in the system.

Step4: Load the data from the folder and preprocess and augment the data based on the train/test/val. Batch size of the network can be decided based on the system GPU/CPU cache memory for our consideration batch size as 32. Based on the set name parameter the preprocessing and augmentation might be apply. After read, preprocess and augment the image we can shuffle the dataset by setting the parameter true.

We then create different lists of storing the testing and training image pixels. After this, we check if the pixel belongs to training then we append it into the training list & training labels. Similarly, for pixels belonging to the Public test, we append it to testing lists. After doing this we convert the training labels and testing labels into categorical ones. The code is given below.

Step5: Predict the trained model based on the input data and calculate the loss of the model using the criterion parameter.

Step6: Initialize the required parameter. Image size should be 48 x 48. Number of different class in face emotion detection consideration is 7. Dataset folder needs to be initialized in data_dir. Changing the trained_model_weight path can give you the different model accuracy results currently we provide the efficiennet v2 small model weights.

Step7: Import the dataset image into torch object

Step8: Import the effcientnet version2 small model base architecture and load the pretrained weights for the same model.

Step9: Import the VGG16 model base architecture and load the pretrained weights for the same model.

Step10: Import the RESNET model base architecture and load the pretrained weights for the same model.

Step11: Import the Efficientnet V2 model base architecture and load the pretrained weights for the same model.

Step12: Compare the results

20 Result and Discussion

All of our studies were run on a 64-bit Windows 10 OS with an Intel Core i5-7200U (7th Gen) CPU. The results achieved utilising the three pre-trained networks, viz. VGG-16, Resnet50, and efficientnetv2, as classifiers to categorise an image into the seven different states, viz. Anger, Contempt, Disgust, Fear, Happy, Sad, and Surprise, are discussed in this part. We compared the findings that they produced. Photos from the training set are used to train the networks, while images from the validation set are used to test them. The validation set consists of photos that have never been viewed previously by the network. To improve the network's training, the data was shuffled.

The method was tested on Google Colab with GPU acceleration enabled using PyTorch. Because of the magnitude of the training data and the lifespan of Google Colab, the researchers had to implement a run-and-pause mechanism every 10 epochs as part of the training mechanism to avoid an abrupt disengagement from the machine, which would cause the training process to end prematurely. The training and validation were completed in a single period. The test was repeated on the model every 10 epochs after each run-and-pause, and the results were recorded.

Many new models have emerged as a result of the application of deep learning for computer vision problems, which outperform previous models. Deep learning faces a significant problem in image categorization and recognition. And it has advanced significantly in recent years. The new EfficientNet models have appeared as a result of this. In image categorization, these models outperformed several previous deep learning models. We now have access to all of the EfficientNet models due to the recent release of PyTorch 1.10. We will use the PyTorch pretrained EfficientNet model to do picture classification in this project.

21 PyTorch Pretrained Models

We now have access to the pre-trained EfficientNet models with PyTorch version 1.10. Using PyTorch's torchvision module, we can access the models. In reality, PyTorch includes all models trained on the ImageNet dataset, from EfficientNetB0 to EfficientNetB7. This means that if our

requirements are similar to those of the pretrained models, we may immediately load and use these models for image classification jobs. We can also apply these to our own dataset for transfer learning and fine-tuning. We'll use EfficientNetB0 to do picture classification and compare the CPU and GPU forward pass times to ResNet and VGG16.

22 Intention of this model

The JAFFE, Cohn-Kanade Dataset (CK), Extended CohnKanade Dataset, KDEF, AffectNet, Static Facial Expression in the Wild (SFEW), and FER2013 are all popular datasets for facial expression recognition research. FER2013 was first presented at the International Conference on Machine Learning (ICML) and is now widely utilised in facial emotion recognition research. Based on a study of prior facial expression recognition research using the FER2013 dataset, we discovered that using ensembled CNN, the maximum accuracy was 76.82 percent, whereas research using a single model without additional data achieved 73.28 percent accuracy. Based on this study, the difficulty of improving the accuracy performance of the model, particularly on a single model, remains. Many experiments with FER2013 use a variety of models, and one prominent architecture is VGG, which has reached state of the art a few times.

Import the required machine learning library library and pytorch frame work to evaluate the model efficiency

[15]:
```
from copy import deepcopy
from datetime import datetime
import os
from os import makedirs
from os.path import join, isfile, isdir
import psutil
import torch
import torch.nn as nn
import torch.optim as optim

from sklearn.metrics import accuracy_score, classification_report
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import torchvision.models as models
import matplotlib.pyplot as plt

module_path = os.path.abspath(os.path.join('.'))
print(module_path)
import sys
if module_path not in sys.path:
    sys.path.append(module_path)
# from efficientnet_v2 import EfficientNetV2
```

```
C:\Users\exterro
```

[16]:
```python
from collections import OrderedDict
from math import ceil, floor

import torch
import torch.nn as nn
import torch.nn.functional as F
import collections.abc as container_abc
from torch.utils import model_zoo


def _pair(x):
    if isinstance(x, container_abc.Iterable):
        return x
    return (x, x)


def torch_conv_out_spatial_shape(in_spatial_shape, kernel_size, stride):
    if in_spatial_shape is None:
        return None
    # in_spatial_shape -> [H,W]
    hin, win = _pair(in_spatial_shape)
    kh, kw = _pair(kernel_size)
    sh, sw = _pair(stride)

    # dilation and padding are ignored since they are always fixed in␣
 ↪efficientnetV2
    hout = int(floor((hin - kh - 1) / sh + 1))
    wout = int(floor((win - kw - 1) / sw + 1))
    return hout, wout


def get_activation(act_fn: str, **kwargs):
    if act_fn in ('silu', 'swish'):
        return nn.SiLU(**kwargs)
    elif act_fn == 'relu':
        return nn.ReLU(**kwargs)
    elif act_fn == 'relu6':
        return nn.ReLU6(**kwargs)
    elif act_fn == 'elu':
        return nn.ELU(**kwargs)
    elif act_fn == 'leaky_relu':
        return nn.LeakyReLU(**kwargs)
    elif act_fn == 'selu':
        return nn.SELU(**kwargs)
    elif act_fn == 'mish':
        return nn.Mish(**kwargs)
    else:
```

```
        raise ValueError('Unsupported act_fn {}'.format(act_fn))


def round_filters(filters, width_coefficient, depth_divisor=8):
    """Round number of filters based on depth multiplier."""
    min_depth = depth_divisor

    filters *= width_coefficient
    new_filters = max(min_depth, int(
        filters + depth_divisor / 2) // depth_divisor * depth_divisor)
    # Make sure that round down does not go down by more than 10%.
    if new_filters < 0.9 * filters:
        new_filters += depth_divisor
    return int(new_filters)


def round_repeats(repeats, depth_coefficient):
    """Round number of filters based on depth multiplier."""
    return int(ceil(depth_coefficient * repeats))


class DropConnect(nn.Module):
    def __init__(self, rate=0.5):
        super(DropConnect, self).__init__()
        self.keep_prob = None
        self.set_rate(rate)

    def set_rate(self, rate):
        if not 0 <= rate < 1:
            raise ValueError(
                "rate must be 0<=rate<1, got {} instead".format(rate))
        self.keep_prob = 1 - rate

    def forward(self, x):
        if self.training:
            random_tensor = self.keep_prob + torch.rand([x.size(0), 1, 1, 1],
                                                        dtype=x.dtype,
                                                        device=x.device)
            binary_tensor = torch.floor(random_tensor)
            return torch.mul(torch.div(x, self.keep_prob), binary_tensor)
        else:
            return x


class SamePaddingConv2d(nn.Module):
    def __init__(self,
                 in_spatial_shape,
```

```
                in_channels,
                out_channels,
                kernel_size,
                stride,
                dilation=1,
                enforce_in_spatial_shape=False,
                **kwargs):
        super(SamePaddingConv2d, self).__init__()

        self._in_spatial_shape = _pair(in_spatial_shape)
        # e.g. throw exception if input spatial shape does not match␣
↪in_spatial_shape
        # when calling self.forward()
        self.enforce_in_spatial_shape = enforce_in_spatial_shape
        kernel_size = _pair(kernel_size)
        stride = _pair(stride)
        dilation = _pair(dilation)

        in_height, in_width = self._in_spatial_shape
        filter_height, filter_width = kernel_size
        stride_heigth, stride_width = stride
        dilation_height, dilation_width = dilation

        out_height = int(ceil(float(in_height) / float(stride_heigth)))
        out_width = int(ceil(float(in_width) / float(stride_width)))

        pad_along_height = max((out_height - 1) * stride_heigth +
                               filter_height + (filter_height - 1) *␣
↪(dilation_height - 1) - in_height, 0)
        pad_along_width = max((out_width - 1) * stride_width +
                              filter_width + (filter_width - 1) *␣
↪(dilation_width - 1) - in_width, 0)

        pad_top = pad_along_height // 2
        pad_bottom = pad_along_height - pad_top
        pad_left = pad_along_width // 2
        pad_right = pad_along_width - pad_left

        paddings = (pad_left, pad_right, pad_top, pad_bottom)
        if any(p > 0 for p in paddings):
            self.zero_pad = nn.ZeroPad2d(paddings)
        else:
            self.zero_pad = None
        self.conv = nn.Conv2d(in_channels=in_channels,
                              out_channels=out_channels,
                              kernel_size=kernel_size,
                              stride=stride,
```

```
                                dilation=dilation,
                                **kwargs)

        self._out_spatial_shape = (out_height, out_width)

    @property
    def out_spatial_shape(self):
        return self._out_spatial_shape

    def check_spatial_shape(self, x):
        if x.size(2) != self._in_spatial_shape[0] or \
                x.size(3) != self._in_spatial_shape[1]:
            raise ValueError(
                "Expected input spatial shape {}, got {} instead".format(self.
↪_in_spatial_shape,
                                                                         x.
↪shape[2:]))

    def forward(self, x):
        if self.enforce_in_spatial_shape:
            self.check_spatial_shape(x)
        if self.zero_pad is not None:
            x = self.zero_pad(x)
        x = self.conv(x)
        return x


class SqueezeExcitate(nn.Module):
    def __init__(self,
                 in_channels,
                 se_size,
                 activation=None):
        super(SqueezeExcitate, self).__init__()
        self.dim_reduce = nn.Conv2d(in_channels=in_channels,
                                    out_channels=se_size,
                                    kernel_size=1)
        self.dim_restore = nn.Conv2d(in_channels=se_size,
                                     out_channels=in_channels,
                                     kernel_size=1)
        self.activation = F.relu if activation is None else activation

    def forward(self, x):
        inp = x
        x = F.adaptive_avg_pool2d(x, (1, 1))
        x = self.dim_reduce(x)
        x = self.activation(x)
        x = self.dim_restore(x)
```

```
        x = torch.sigmoid(x)
        return torch.mul(inp, x)


class MBConvBlockV2(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 kernel_size,
                 stride,
                 expansion_factor,
                 act_fn,
                 act_kwargs={},
                 bn_epsilon=None,
                 bn_momentum=None,
                 se_size=None,
                 drop_connect_rate=None,
                 bias=False,
                 tf_style_conv=False,
                 in_spatial_shape=None):

        super().__init__()

        exp_channels = in_channels * expansion_factor

        self.ops_lst = []

        # expansion convolution
        if expansion_factor != 1:
            self.expand_conv = nn.Conv2d(
                in_channels=in_channels,
                out_channels=exp_channels,
                kernel_size=1,
                bias=bias)

            self.expand_bn = nn.BatchNorm2d(
                num_features=exp_channels,
                eps=bn_epsilon,
                momentum=bn_momentum)

            self.expand_act = get_activation(act_fn, **act_kwargs)
            self.ops_lst.extend(
                [self.expand_conv, self.expand_bn, self.expand_act])

        # depth-wise convolution
        if tf_style_conv:
            self.dp_conv = SamePaddingConv2d(
```

```
                in_spatial_shape=in_spatial_shape,
                in_channels=exp_channels,
                out_channels=exp_channels,
                kernel_size=kernel_size,
                stride=stride,
                groups=exp_channels,
                bias=bias)
            self.out_spatial_shape = self.dp_conv.out_spatial_shape
        else:
            self.dp_conv = nn.Conv2d(
                in_channels=exp_channels,
                out_channels=exp_channels,
                kernel_size=kernel_size,
                stride=stride,
                padding=1,
                groups=exp_channels,
                bias=bias)
            self.out_spatial_shape = torch_conv_out_spatial_shape(
                in_spatial_shape, kernel_size, stride)

        self.dp_bn = nn.BatchNorm2d(
            num_features=exp_channels,
            eps=bn_epsilon,
            momentum=bn_momentum)

        self.dp_act = get_activation(act_fn, **act_kwargs)
        self.ops_lst.extend(
            [self.dp_conv, self.dp_bn, self.dp_act])

        # Squeeze and Excitate
        if se_size is not None:
            self.se = SqueezeExcitate(exp_channels,
                                      se_size,
                                      activation=get_activation(act_fn,␣
↪**act_kwargs))
            self.ops_lst.append(self.se)

        # projection layer
        self.project_conv = nn.Conv2d(
            in_channels=exp_channels,
            out_channels=out_channels,
            kernel_size=1,
            bias=bias)

        self.project_bn = nn.BatchNorm2d(
            num_features=out_channels,
            eps=bn_epsilon,
```

```
            momentum=bn_momentum)

        # no activation function in projection layer

        self.ops_lst.extend(
            [self.project_conv, self.project_bn])

        self.skip_enabled = in_channels == out_channels and stride == 1

        if self.skip_enabled and drop_connect_rate is not None:
            self.drop_connect = DropConnect(drop_connect_rate)
            self.ops_lst.append(self.drop_connect)

    def forward(self, x):
        inp = x
        for op in self.ops_lst:
            x = op(x)
        if self.skip_enabled:
            return x + inp
        else:
            return x


class FusedMBConvBlockV2(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 kernel_size,
                 stride,
                 expansion_factor,
                 act_fn,
                 act_kwargs={},
                 bn_epsilon=None,
                 bn_momentum=None,
                 se_size=None,
                 drop_connect_rate=None,
                 bias=False,
                 tf_style_conv=False,
                 in_spatial_shape=None):

        super().__init__()

        exp_channels = in_channels * expansion_factor

        self.ops_lst = []

        # expansion convolution
```

```
        if expansion_factor != 1:
            if tf_style_conv:
                self.expand_conv = SamePaddingConv2d(
                    in_spatial_shape=in_spatial_shape,
                    in_channels=in_channels,
                    out_channels=exp_channels,
                    kernel_size=kernel_size,
                    stride=stride,
                    bias=bias)
            else:
                self.expand_conv = nn.Conv2d(
                    in_channels=in_channels,
                    out_channels=exp_channels,
                    kernel_size=kernel_size,
                    padding=1,
                    stride=stride,
                    bias=bias)

            self.expand_bn = nn.BatchNorm2d(
                num_features=exp_channels,
                eps=bn_epsilon,
                momentum=bn_momentum)

            self.expand_act = get_activation(act_fn, **act_kwargs)
            self.ops_lst.extend(
                [self.expand_conv, self.expand_bn, self.expand_act])

        # Squeeze and Excitate
        if se_size is not None:
            self.se = SqueezeExcitate(exp_channels,
                                      se_size,
                                      activation=get_activation(act_fn,␣
↪**act_kwargs))
            self.ops_lst.append(self.se)

        # projection layer
        kernel_size = 1 if expansion_factor != 1 else kernel_size
        stride = 1 if expansion_factor != 1 else stride
        if tf_style_conv:
            self.project_conv = SamePaddingConv2d(
                in_spatial_shape=in_spatial_shape,
                in_channels=exp_channels,
                out_channels=out_channels,
                kernel_size=kernel_size,
                stride=stride,
                bias=bias)
            self.out_spatial_shape = self.project_conv.out_spatial_shape
```

```
        else:
            self.project_conv = nn.Conv2d(
                in_channels=exp_channels,
                out_channels=out_channels,
                kernel_size=kernel_size,
                stride=stride,
                padding=1 if kernel_size > 1 else 0,
                bias=bias)
            self.out_spatial_shape = torch_conv_out_spatial_shape(
                in_spatial_shape, kernel_size, stride)

        self.project_bn = nn.BatchNorm2d(
            num_features=out_channels,
            eps=bn_epsilon,
            momentum=bn_momentum)

        self.ops_lst.extend(
            [self.project_conv, self.project_bn])

        if expansion_factor == 1:
            self.project_act = get_activation(act_fn, **act_kwargs)
            self.ops_lst.append(self.project_act)

        self.skip_enabled = in_channels == out_channels and stride == 1

        if self.skip_enabled and drop_connect_rate is not None:
            self.drop_connect = DropConnect(drop_connect_rate)
            self.ops_lst.append(self.drop_connect)

    def forward(self, x):
        inp = x
        for op in self.ops_lst:
            x = op(x)
        if self.skip_enabled:
            return x + inp
        else:
            return x


class EfficientNetV2(nn.Module):
    _models = {'s': {'num_repeat': [2, 4, 4, 6, 9, 15],
                     'kernel_size': [3, 3, 3, 3, 3, 3],
                     'stride': [1, 2, 2, 2, 1, 2],
                     'expand_ratio': [1, 4, 4, 4, 6, 6],
                     'in_channel': [24, 24, 48, 64, 128, 160],
                     'out_channel': [24, 48, 64, 128, 160, 256],
                     'se_ratio': [None, None, None, 0.25, 0.25, 0.25],
```

```
                    'conv_type': [1, 1, 1, 0, 0, 0],
                    'is_feature_stage': [False, True, True, False, True, True],
                    'width_coefficient': 1.0,
                    'depth_coefficient': 1.0,
                    'train_size': 300,
                    'eval_size': 384,
                    'dropout': 0.2,
                    'weight_url': 'https://api.onedrive.com/v1.0/shares/u!
↪aHR0cHM6Ly8xZHJ2Lm1zL3UvcyFBdGlRcHc5VGNjZmllbFF5VWJOZzd0cmhBbm8/root/
↪content',
                    'model_name': 'efficientnet_v2_s_21k_ft1k-dbb43f38.pth'},
             'm': {'num_repeat': [3, 5, 5, 7, 14, 18, 5],
                    'kernel_size': [3, 3, 3, 3, 3, 3, 3],
                    'stride': [1, 2, 2, 2, 1, 2, 1],
                    'expand_ratio': [1, 4, 4, 4, 6, 6, 6],
                    'in_channel': [24, 24, 48, 80, 160, 176, 304],
                    'out_channel': [24, 48, 80, 160, 176, 304, 512],
                    'se_ratio': [None, None, None, 0.25, 0.25, 0.25, 0.25],
                    'conv_type': [1, 1, 1, 0, 0, 0, 0],
                    'is_feature_stage': [False, True, True, False, True,␣
↪False, True],
                    'width_coefficient': 1.0,
                    'depth_coefficient': 1.0,
                    'train_size': 384,
                    'eval_size': 480,
                    'dropout': 0.3,
                    'weight_url': 'https://api.onedrive.com/v1.0/shares/u!
↪aHR0cHM6Ly8xZHJ2Lm1zL3UvcyFBdGlRcHc5VGNjZmllN1ZDazRFb0o1bnlyNUE/root/
↪content',
                    'model_name': 'efficientnet_v2_m_21k_ft1k-da8e56c0.pth'},
             'l': {'num_repeat': [4, 7, 7, 10, 19, 25, 7],
                    'kernel_size': [3, 3, 3, 3, 3, 3, 3],
                    'stride': [1, 2, 2, 2, 1, 2, 1],
                    'expand_ratio': [1, 4, 4, 4, 6, 6, 6],
                    'in_channel': [32, 32, 64, 96, 192, 224, 384],
                    'out_channel': [32, 64, 96, 192, 224, 384, 640],
                    'se_ratio': [None, None, None, 0.25, 0.25, 0.25, 0.25],
                    'conv_type': [1, 1, 1, 0, 0, 0, 0],
                    'is_feature_stage': [False, True, True, False, True,␣
↪False, True],
                    'feature_stages': [1, 2, 4, 6],
                    'width_coefficient': 1.0,
                    'depth_coefficient': 1.0,
                    'train_size': 384,
                    'eval_size': 480,
                    'dropout': 0.4,
```

```
                    'weight_url': 'https://api.onedrive.com/v1.0/shares/u!
↪aHR0cHM6Ly8xZHJ2Lm1zL3UvcyFBdGlRcHc5VGNjZmlmcmIyRHEtQTBhUTBhWVE/root/
↪content',
                    'model_name': 'efficientnet_v2_l_21k_ft1k-08121eee.pth'},
               'xl': {'num_repeat': [4, 8, 8, 16, 24, 32, 8],
                      'kernel_size': [3, 3, 3, 3, 3, 3, 3],
                      'stride': [1, 2, 2, 2, 1, 2, 1],
                      'expand_ratio': [1, 4, 4, 4, 6, 6, 6],
                      'in_channel': [32, 32, 64, 96, 192, 256, 512],
                      'out_channel': [32, 64, 96, 192, 256, 512, 640],
                      'se_ratio': [None, None, None, 0.25, 0.25, 0.25, 0.25],
                      'conv_type': [1, 1, 1, 0, 0, 0, 0],
                      'is_feature_stage': [False, True, True, False, True,␣
↪False, True],
                      'feature_stages': [1, 2, 4, 6],
                      'width_coefficient': 1.0,
                      'depth_coefficient': 1.0,
                      'train_size': 384,
                      'eval_size': 512,
                      'dropout': 0.4,
                      'weight_url': 'https://api.onedrive.com/v1.0/shares/u!
↪aHR0cHM6Ly8xZHJ2Lm1zL3UvcyFBdGlRcHc5VGNjZmlmVXQtRHJLa21taUkxWkE/root/
↪content',
                      'model_name': 'efficientnet_v2_xl_21k_ft1k-1fcc9744.pth'}}

   def __init__(self,
                model_name,
                in_channels=3,
                n_classes=1000,
                tf_style_conv=False,
                in_spatial_shape=None,
                activation='silu',
                activation_kwargs=None,
                bias=False,
                drop_connect_rate=0.2,
                dropout_rate=None,
                bn_epsilon=1e-3,
                bn_momentum=0.01,
                pretrained=False,
                progress=False,
                ):
       super().__init__()

       self.blocks = nn.ModuleList()
       self.model_name = model_name
       self.cfg = self._models[model_name]
```

```
        if tf_style_conv and in_spatial_shape is None:
            in_spatial_shape = self.cfg['eval_size']

        activation_kwargs = {} if activation_kwargs is None else␣
↪activation_kwargs
        dropout_rate = self.cfg['dropout'] if dropout_rate is None else␣
↪dropout_rate
        _input_ch = in_channels

        self.feature_block_ids = []

        # stem
        if tf_style_conv:
            self.stem_conv = SamePaddingConv2d(
                in_spatial_shape=in_spatial_shape,
                in_channels=in_channels,
                out_channels=round_filters(
                    self.cfg['in_channel'][0], self.cfg['width_coefficient']),
                kernel_size=3,
                stride=2,
                bias=bias
            )
            in_spatial_shape = self.stem_conv.out_spatial_shape
        else:
            self.stem_conv = nn.Conv2d(
                in_channels=in_channels,
                out_channels=round_filters(
                    self.cfg['in_channel'][0], self.cfg['width_coefficient']),
                kernel_size=3,
                stride=2,
                padding=1,
                bias=bias
            )

        self.stem_bn = nn.BatchNorm2d(
            num_features=round_filters(
                self.cfg['in_channel'][0], self.cfg['width_coefficient']),
            eps=bn_epsilon,
            momentum=bn_momentum)

        self.stem_act = get_activation(activation, **activation_kwargs)

        drop_connect_rates = self.get_dropconnect_rates(drop_connect_rate)

        stages = zip(*[self.cfg[x] for x in ['num_repeat', 'kernel_size',␣
↪'stride',
```

```
                                            'expand_ratio', 'in_channel',␣
↪'out_channel', 'se_ratio', 'conv_type', 'is_feature_stage']])

        idx = 0

        for stage_args in stages:
            (num_repeat, kernel_size, stride, expand_ratio,
             in_channels, out_channels, se_ratio, conv_type, is_feature_stage)␣
↪= stage_args

            in_channels = round_filters(
                in_channels, self.cfg['width_coefficient'])
            out_channels = round_filters(
                out_channels, self.cfg['width_coefficient'])
            num_repeat = round_repeats(
                num_repeat, self.cfg['depth_coefficient'])

            conv_block = MBConvBlockV2 if conv_type == 0 else FusedMBConvBlockV2

            for _ in range(num_repeat):
                se_size = None if se_ratio is None else max(
                    1, int(in_channels * se_ratio))
                _b = conv_block(
                    in_channels=in_channels,
                    out_channels=out_channels,
                    kernel_size=kernel_size,
                    stride=stride,
                    expansion_factor=expand_ratio,
                    act_fn=activation,
                    act_kwargs=activation_kwargs,
                    bn_epsilon=bn_epsilon,
                    bn_momentum=bn_momentum,
                    se_size=se_size,
                    drop_connect_rate=drop_connect_rates[idx],
                    bias=bias,
                    tf_style_conv=tf_style_conv,
                    in_spatial_shape=in_spatial_shape
                )
                self.blocks.append(_b)
                idx += 1
                if tf_style_conv:
                    in_spatial_shape = _b.out_spatial_shape
                in_channels = out_channels
                stride = 1

            if is_feature_stage:
                self.feature_block_ids.append(idx - 1)
```

```
        head_conv_out_channels = round_filters(
            1280, self.cfg['depth_coefficient'])

        self.head_conv = nn.Conv2d(
            in_channels=in_channels,
            out_channels=head_conv_out_channels,
            kernel_size=1,
            bias=bias)
        self.head_bn = nn.BatchNorm2d(
            num_features=head_conv_out_channels,
            eps=bn_epsilon,
            momentum=bn_momentum)
        self.head_act = get_activation(activation, **activation_kwargs)

        self.dropout = nn.Dropout(p=dropout_rate)

        self.avpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(head_conv_out_channels, n_classes)

        if pretrained:
            self._load_state(_input_ch, n_classes, progress, tf_style_conv)

        return

    def _load_state(self, in_channels, n_classes, progress, tf_style_conv):
        state_dict = model_zoo.load_url(
            self.cfg['weight_url'],
            progress=progress,
            file_name=self.cfg['model_name'])

        strict = True

        if not tf_style_conv:
            state_dict = OrderedDict([(k.replace('.conv.', '.'), v) if '.conv.'␣
↪in k else (
                k, v) for k, v in state_dict.items()])

        if in_channels != 3:
            if tf_style_conv:
                state_dict.pop('stem_conv.conv.weight')
            else:
                state_dict.pop('stem_conv.weight')
            strict = False

        if n_classes != 1000:
            state_dict.pop('fc.weight')
```

```
            state_dict.pop('fc.bias')
            strict = False

        self.load_state_dict(state_dict, strict=strict)
        print("Model weights loaded successfully.")

    def get_dropconnect_rates(self, drop_connect_rate):
        nr = self.cfg['num_repeat']
        dc = self.cfg['depth_coefficient']
        total = sum(round_repeats(nr[i], dc) for i in range(len(nr)))
        return [drop_connect_rate * i / total for i in range(total)]

    def get_features(self, x):
        x = self.stem_act(self.stem_bn(self.stem_conv(x)))

        features = []
        feat_idx = 0
        for block_idx, block in enumerate(self.blocks):
            x = block(x)
            if block_idx == self.feature_block_ids[feat_idx]:
                features.append(x)
                feat_idx += 1

        return features

    def forward(self, x):
        x = self.stem_act(self.stem_bn(self.stem_conv(x)))

        for block in self.blocks:
            x = block(x)

        x = self.head_act(self.head_bn(self.head_conv(x)))

        x = self.dropout(torch.flatten(self.avpool(x), 1))

        return self.fc(x)
```

Indentifies the number of processor/thread available in the system.

[17]:
```
num_workers = psutil.cpu_count()
```

Load the data from the folder and preprocess and augment the data based on the train/test/val. Batch size of the network can be decided based on the sytem GPU/CPU chache memory for our consideration batch size as 32. Based on the set name parameter the preprocessing and agumentation might be apply. After read, preprocess and augment the image we can shuffle the dataset by setting the parameter true.

We then create different lists of storing the testing and training image pixels. After this, we check if the pixel belongs to training then we append it into the training list & training labels. Similarly,

for pixels belonging to the Public test, we append it to testing lists. After doing this we convert the training labels and testing labels into categorical ones. The code is given below.

[18]:
```
def loaddata(data_dir, batch_size, set_name, shuffle):
    input_size = 48
    data_transforms = {
        'train': transforms.Compose([
            transforms.Resize(input_size),
            transforms.CenterCrop(input_size),
            transforms.RandomAffine(degrees=0, translate=(0.05, 0.05)),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'test': transforms.Compose([
            transforms.Resize(input_size),
            transforms.ToTensor(),
            #transforms.Normalize(mean=0.5,std=0.5),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'val': transforms.Compose([
            transforms.Resize(input_size),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    }
    image_datasets = {x: datasets.ImageFolder(data_dir, data_transforms[x]) for␣
↪x in [set_name]}

    dataset_loaders = {x: torch.utils.data.DataLoader(image_datasets[x],
                                                      batch_size=batch_size,
                                                      shuffle=shuffle,␣
↪num_workers=num_workers) for x in [set_name]}
    data_set_sizes = len(image_datasets[set_name])
    return dataset_loaders, data_set_sizes
```

Predict the trained model based on the input data and calculate the loss of the model using the criterion parameter.

[19]:
```
def eval_model(model, dataloader, device, criterion=None):
    loss_value = []
    y_pred = []
    y_true = []
    model.eval()
    with torch.no_grad():
        for xb, yb in dataloader:
            xb, yb = xb.to(device), yb.to(device)
            out = model(xb)
```

```
            if out.size(1) == 1:
                # regression, squeeze output of shape [N,1] to [N]
                out = torch.squeeze(out, 1)
            if criterion is not None:
                loss = criterion(out, yb)
                loss_value.append(loss.item())
            y_pred.append(out.detach().cpu())
            y_true.append(yb.detach().cpu())
    if criterion is not None:
        loss_value = sum(loss_value) / len(loss_value)
        return torch.cat(y_pred), torch.cat(y_true), loss_value
    else:
        return torch.cat(y_pred), torch.cat(y_true)
```

Calculate and display the accuracy and F1 score measure of each class in the trained model.

[20]:
```
def metric_fn(y_pred, y_true):
    _, y_pred = torch.max(y_pred, 1)
    print(classification_report(y_true,y_pred))
    return accuracy_score(y_pred, y_true)
```

Intiatize the required parameter. Image size should be 48 x 48. number of different class in face emotion detection considereation is 7. Dataset folder needs to be initialized in data_dir. Changing the trained_model_weight path can give you the different model accuracy results Currently we provide the efficiennet v2 small model weights.

[30]:
```
batch_size = 32
model_index = 's'
img_size = 48
n_classes = 7
pretrained = False
data_dir = r"./Dataset/FER/"
traind_effv2_model_weight = "./input/best_weights_effv2_s_Adam.pth"
traind_resnet_model_weight = "./input/best_weights_resnet_Adam.pth"
traind_vgg_model_weight = "./input/best_weights_VGG16_Adam.pth"
```

Import the dataset image into torch object

[31]:
```
dataloader_val_loader, validation_sizes = loaddata(data_dir=os.path.
↪join(data_dir, "test"), batch_size=batch_size, set_name='test',␣
↪shuffle=False)
dataloader_validation = dataloader_val_loader["test"]
print("Testing  data size : ",validation_sizes)

criterion = nn.CrossEntropyLoss()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
Testing  data size :  7066
```

Import the effcientnet version2 small model base architecture and load the pretrained weights for

the same model.

[32]:
```
model = models.resnet18(pretrained=False)
model.load_state_dict(torch.load(traind_resnet_model_weight,map_location=torch.
 ↪device('cpu')))
model.to(device)
```

[32]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

[33]:
```
model_effv2 = EfficientNetV2(model_index,
                     in_spatial_shape=img_size,
                     n_classes=n_classes,
                     pretrained=pretrained,
                     )
model_effv2.load_state_dict(torch.
 ↪load(traind_effv2_model_weight,map_location=torch.device('cpu')))
model_effv2.to(device)
```

[33]:
```
EfficientNetV2(
  (blocks): ModuleList(
```

```
    (0): FusedMBConvBlockV2(
      (project_conv): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (project_bn): BatchNorm2d(24, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (project_act): SiLU()
      (drop_connect): DropConnect()
    )
    (1): FusedMBConvBlockV2(
      (project_conv): Conv2d(24, 24, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (project_bn): BatchNorm2d(24, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (project_act): SiLU()
      (drop_connect): DropConnect()
    )
    (2): FusedMBConvBlockV2(
      (expand_conv): Conv2d(24, 96, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(96, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(96, 48, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(48, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
    )
    (3): FusedMBConvBlockV2(
      (expand_conv): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(192, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(48, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (4): FusedMBConvBlockV2(
      (expand_conv): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(192, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1),
bias=False)
```

```
      (project_bn): BatchNorm2d(48, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (5): FusedMBConvBlockV2(
      (expand_conv): Conv2d(48, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(192, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(48, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (6): FusedMBConvBlockV2(
      (expand_conv): Conv2d(48, 192, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(192, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(64, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
    )
    (7): FusedMBConvBlockV2(
      (expand_conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(64, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (8): FusedMBConvBlockV2(
      (expand_conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
```

```
      (project_bn): BatchNorm2d(64, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (9): FusedMBConvBlockV2(
      (expand_conv): Conv2d(64, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (expand_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (project_conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(64, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (10): MBConvBlockV2(
      (expand_conv): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), groups=256, bias=False)
      (dp_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(256, 16, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(16, 256, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(128, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
    )
    (11): MBConvBlockV2(
      (expand_conv): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=512, bias=False)
      (dp_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
```

```
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(128, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (12): MBConvBlockV2(
      (expand_conv): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=512, bias=False)
      (dp_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(128, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (13): MBConvBlockV2(
      (expand_conv): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=512, bias=False)
      (dp_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
```

```
        (dim_restore): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(128, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (14): MBConvBlockV2(
      (expand_conv): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=512, bias=False)
      (dp_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(128, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (15): MBConvBlockV2(
      (expand_conv): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=512, bias=False)
      (dp_bn): BatchNorm2d(512, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(512, 32, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(32, 512, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
```

```
      (project_conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(128, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (16): MBConvBlockV2(
      (expand_conv): Conv2d(128, 768, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(768, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(768, 768, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=768, bias=False)
      (dp_bn): BatchNorm2d(768, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(768, 32, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(32, 768, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
    )
    (17): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
```

```
      (drop_connect): DropConnect()
    )
    (18): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (19): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (20): MBConvBlockV2(
```

```
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (21): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (22): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
```

```
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (23): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (24): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 160, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(160, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (25): MBConvBlockV2(
      (expand_conv): Conv2d(160, 960, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(960, 960, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), groups=960, bias=False)
      (dp_bn): BatchNorm2d(960, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(960, 40, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(40, 960, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(960, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
    )
    (26): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
```

```
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (27): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (28): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
```

```
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (29): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (30): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
```

```
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (31): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (32): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
```

```
      (drop_connect): DropConnect()
    )
    (33): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (34): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (35): MBConvBlockV2(
```

```
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (36): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (37): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
```

```
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (38): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
    (39): MBConvBlockV2(
      (expand_conv): Conv2d(256, 1536, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (expand_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (expand_act): SiLU()
      (dp_conv): Conv2d(1536, 1536, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1), groups=1536, bias=False)
      (dp_bn): BatchNorm2d(1536, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (dp_act): SiLU()
      (se): SqueezeExcitate(
        (dim_reduce): Conv2d(1536, 64, kernel_size=(1, 1), stride=(1, 1))
        (dim_restore): Conv2d(64, 1536, kernel_size=(1, 1), stride=(1, 1))
        (activation): SiLU()
      )
      (project_conv): Conv2d(1536, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
      (project_bn): BatchNorm2d(256, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
      (drop_connect): DropConnect()
    )
  )
  (stem_conv): Conv2d(3, 24, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
  (stem_bn): BatchNorm2d(24, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
  (stem_act): SiLU()
  (head_conv): Conv2d(256, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (head_bn): BatchNorm2d(1280, eps=0.001, momentum=0.01, affine=True,
track_running_stats=True)
  (head_act): SiLU()
  (dropout): Dropout(p=0.2, inplace=False)
  (avpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=1280, out_features=7, bias=True)
)
```

[34]:
```
model_vgg16 = models.vgg16_bn(pretrained=False)
# Freeze training for all layers
for param in model_vgg16.features.parameters():
    param.require_grad = False

# Newly created modules have require_grad=True by default
num_features = model_vgg16.classifier[6].in_features
features = list(model_vgg16.classifier.children())[:-1] # Remove last layer
features.extend([nn.Linear(num_features, n_classes)]) # Add our layer with 4␣
 ↪outputs
model_vgg16.classifier = nn.Sequential(*features) # Replace the model classifier

model_vgg16.load_state_dict(torch.
 ↪load(traind_vgg_model_weight,map_location=torch.device('cpu')))
model_vgg16.to(device)
```

[34]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (39): ReLU(inplace=True)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=7, bias=True)
  )
)
```

Predict the dataset input data. Evaluate the loss of the model using cross entropy calculation. Measure the accuracy and F1 score metric for each class data

23 Testing and Measure Performance

FER2013 private data will be used to test the correctness of all models. A multiclass confusion matrix is utilised in addition to the overall accuracy computation to evaluate the performance of the models that have been constructed. The macro average is used to assess the model's performance, precision, recall, and F1-Score computations. The number of datasets for each label is represented by the confusion matrix's row elements. The confusion matrix's column elements, on the other hand, are numeric values that represent the projected outputs of a model on the dataset.

Precision, also known as positive predictive value, is a measure that compares the number of true positives to all positive data. The number of true positives compared to the sum of all true data is referred to as recall, also known as sensitivity. The harmonic mean of precision and recall is the F1-Score. Because the macro average treats all classes equally, the class with the least data is also important.

24 Performance Evaluation Metrics

In this research work, the performance of Deep Learning Models is assessed using various performance metrics such as accuracy (acc), precision (P), recall (R), specificity (S), and F1-score (F1). Precision: The proportion of expected positives that are true positives is measured by this parameter. As a result, true positive (TP) and false positive (FP) values are important. P= TP / (TP+FP) Recall: Recall that this is the ratio of true positives accurately classified by the model. Recall is calculated using TP and FN values. R = TP / (TP+FN) Specificity: Specificity is the proportion of true negatives (those not caused by pathology) accurately categorized by the model. Specificity is calculated using TN and FP values. S = TN / (TN+FP) F1-Score: This is a combined precision and recall measure of the model's accuracy. It's the ratio of the F1-score precision and recall metrics to their totals multiplied by two.

F1 = 2 × (P×R) / (P+R) Accuracy: The ratio of accurately classified by the model (TP+TN) to total numbers (TP+TN+FP+FN) represents an algorithm's accuracy.

ACC = (TN+TP) / (TN+FP+FN+TP)

- True Positives (TP)
- True Negatives (TN)
- False positives (FP)
- False Negatives (FN)

25 RESNET Experiment Result

[26]:
```
val_results = eval_model(model, dataloader_validation, device,␣
 ↪criterion=criterion)
y_pred, y_true, loss = val_results
Accuracy = metric_fn(y_pred, y_true)
print("Resnet Test accuaracy : ",Accuracy)
```

```
              precision    recall  f1-score   support

           0       0.60      0.56      0.58       960
           1       0.65      0.41      0.51       111
           2       0.57      0.45      0.50      1018
           3       0.87      0.89      0.88      1825
           4       0.62      0.66      0.64      1216
           5       0.53      0.62      0.57      1139
           6       0.78      0.77      0.78       797

    accuracy                           0.68      7066
   macro avg       0.66      0.62      0.64      7066
weighted avg       0.68      0.68      0.67      7066

Resnet Test accuaracy :  0.6769034814605152
```

25.1 Efficientnet V2 Experiment Result

[27]:
```
effv2_val_results = eval_model(model_effv2, dataloader_validation, device,␣
 ↪criterion=criterion)
effv2_pred, effv2_true, effv2_loss = effv2_val_results
effv2_Accuracy = metric_fn(effv2_pred, effv2_true)
print("Efficientnet V2 small Test accuaracy : ",effv2_Accuracy)
```

```
              precision    recall  f1-score   support

           0       0.70      0.68      0.69       960
           1       0.79      0.73      0.76       111
           2       0.67      0.58      0.62      1018
           3       0.92      0.93      0.93      1825
           4       0.71      0.79      0.75      1216
           5       0.65      0.66      0.66      1139
           6       0.85      0.85      0.85       797

    accuracy                           0.77      7066
   macro avg       0.76      0.75      0.75      7066
weighted avg       0.77      0.77      0.76      7066

Efficientnet V2 small Test accuaracy :  0.76648740447212
```

26 VGG16 Experiment Result

[28]:
```
val_results = eval_model(model_vgg16, dataloader_validation, device,␣
 ↪criterion=criterion)
y_pred, y_true, loss = val_results
Accuracy = metric_fn(y_pred, y_true)
print("VGG16 Test accuaracy : ",Accuracy)
```

```
              precision    recall  f1-score   support

           0       0.64      0.51      0.57       960
           1       0.61      0.37      0.46       111
           2       0.52      0.37      0.43      1018
           3       0.81      0.92      0.86      1825
           4       0.58      0.72      0.64      1216
           5       0.54      0.55      0.55      1139
           6       0.72      0.71      0.72       797

    accuracy                           0.66      7066
   macro avg       0.63      0.59      0.60      7066
weighted avg       0.65      0.66      0.65      7066

VGG16 Test accuaracy :  0.6579394282479479
```

Above results illustrates the performance of reported models on the FER2013 dataset and models trained on this experiment. All algorithms developed for this project outperform human performance estimates. This work's model reaches state-of-the-art performance, with accuracy 77 percent higher than the previous best single-based model published.

With an accuracy of 77%, the model developed in this experiment is the best single standalone model for the FER2013 dataset. Based on the results of earlier literature review experiments, the VGG model variation outperforms the other models for FER2013 data. This over-fitting issue arises from the fact that both models are complicated models that do not perform well on FER2013, owing to a lack of data as previously noted. VGG16 and RESNET show that the tiny model works effectively on FER2013.

26.1 Conclusion and Future work

We contributed to tackling the problem of improving facial expression recognition accuracy on FER2013 with our model, Efficientnet V2, which achieved the highest accuracy value. In the future, more effort might be done to improve results by training using the Vision Transformer model and capable hardware resources. Ensemble models, which include VGGSpinalNet and other models, can be employed in future research to increase accuracy even more.

27 References

"Application: Facial Expression Recognition" in Machine Learning in Computer Vision. Computational Imaging and Vision, Dordrecht: Springer, vol. 29, 2005.

G.-B. Duchenne, Mécanisme de la physionomie humaine ou analyse électro-physiologique de ses différents modes de l'expression, Paris:Archives générales de médecine, vol. 1, pp. 29-47, 1862.

C. Pramerdorfer and M. Kampel, Facial expression recognition using convolutional neural networks: State of the art, 2016, [online] Available:

Z. Zhang, P. Luo, C.-C. Loy and X. Tang, "Learning Social Relation Traits from Face Images", Proc. IEEE Int. Conference on Computer Vision (ICCV), pp. 3631-3639, 2015.

B.-K. Kim, S.-Y. Dong, J. Roh, G. Kim and S.-Y. Lee, "Fusing aligned and non-aligned face information for automatic affect recognition in the wild: A deep learning approach", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 48-57, 2016.

A. Raghuvanshi and V. Choksi, Facial Expression Recognition with Convolutional Neural Networks, 2016.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems,pages1097–1105,2012.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385,2015.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556,2014.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,pages 1–9,2015.

R. Srivastava, K. Greff and J. Schmidhuber. Training Very Deep Networks. arXiv preprint arXiv:1507.06228v2,2015.

S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Comput., 9(8):1735–1780, Nov. 1997.

O. Tutsoy, F. Göngör, D. Barkana, and H. Kose, "AN EMOTION ANALYSIS ALGORITHM AND IMPLEMENTATION TO NAO HUMANOID ROBOT," The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM), vol. 1, pp. 316–330, Jan. 2017.

F. Wang, H. Chen, L. Kong, and W. Sheng, "Real-time Facial Expression Recognition on Robot for Healthcare," Apr. 2018, pp. 402–406. doi: 10.1109/IISR.2018.8535710.

J. v. Moniaga, A. Chowanda, A. Prima, Oscar, and M. D. Tri Rizqi, "Facial Expression Recognition as Dynamic Game Balancing System," Procedia Computer Science, vol. 135, pp. 361–368, Jan. 2018, doi: 10.1016/J.PROCS.2018.08.185.

P. Ekman and W. v. Friesen, "Constants across cultures in the face and emotion," Journal of Personality and Social Psychology, vol. 17, no. 2, pp. 124–129, Feb. 1971, doi: 10.1037/H0030377

P. Lucey, J. F. Cohn, T. Kanade, J. Saragih, Z. Ambadar, and I. Matthews, "The extended Cohn-Kanade dataset (CK+): A complete dataset for action unit and emotion-specified expression," 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops, CVPRW 2010, pp. 94–101, 2010, doi: 10.1109/CVPRW.2010.5543262.

P. Ekman and E. L. Rosenberg, "What the Face Reveals: Basic and Applied Studies of Spontaneous Expression Using the Facial Action Coding System (FACS)," What the Face Reveals: Basic and Applied Studies of Spontaneous Expression Using the Facial Action Coding System (FACS), pp. 1–672, Mar. 2012, doi: 10.1093/ACPROF:OSO/9780195179644.0 01.0001.

L. Wright and N. Demeure, "Ranger21: a synergistic deep learning optimizer," Jun.2021, Accessed: Oct. 17, 2021. [Online]. Available:

M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," Information Processing & Management, vol. 45, no. 4, pp. 427–437, Jul. 2009, doi: 10.1016/J.IPM.2009.03.002.

D. Misra, "Mish: A Self Regularized NonMonotonic Activation Function," Aug. 2019, Accessed: Oct. 17, 2021. [Online]. Available: https://arxiv.org/abs/1908.08681v3

D. Misra, "Mish: A Self Regularized NonMonotonic Activation Function," Aug. 2019, Accessed: Oct. 17, 2021. [Online]. Available: https://arxiv.org/abs/1908.08681v3

M. R. Zhang, J. Lucas, G. Hinton, and J. Ba, "Lookahead Optimizer: k steps forward, 1 step back," Advances in Neural Information Processing Systems, vol. 32, Jul. 2019, Accessed: Oct. 17, 2021. [Online]. Available: https://arxiv.org/abs/1907.08610v1

X. Wang et al., "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11133 LNCS, pp. 63–79, Sep. 2018, Accessed: Oct. 23, 2021. P.-L. Carrier and A. Courville, "Challenges in Representation Learning: Facial Expression Recognition Challenge," Apr. 13, 2013. https://www.kaggle.com/c/challenges-inrepresentation-learning-facial-expressionrecognition-challenge/data (accessed Jan. 24, 2021).

[ ]: