IMAGE CLASSIFIER

A MINI PROJECT REPORT

IT5039 – Deep Learning

Submitted by

ADITYA TARUN S (2020506007)

MADHUMITHA R (2020506049)

NEHA M (2020506058)

DEPARTMENT OF INFORMATION TECHNOLOGY

MADRAS INSTITUTE OF TECHNOLOGY CAMPUS

ANNA UNIVERSITY :: CHENNAI 600 044

Abstract - In recent year, with the speedy development in the digital contents identification, automatic classification of the images became most challenging task in the fields of computer vision. Automatic understanding and analysing of images by system is difficult as compared to human visions. Several research have been done to overcome problem in existing classification system, but the output was narrowed only to low level image primitives. However, those approach lack with accurate classification of images. In this paper, our system uses deep learning algorithm to achieve the expected results in the area like computer visions. Our system present Transfer learning, a deep learning technique is being used for automatic classification of the images. Our system uses the ImageNet data set in MobileNet architecture for classification of grayscale images. The grayscale images in the data set used for training which require more computational power for classification of images. The results show the effectiveness of deep learning based image classification using MobileNet.

I. Introduction

In recent years, the field of computer vision has been greatly impacted by the rapid advancements in digital content identification. Automatic image classification has become one of the most challenging tasks in this field, as it is more difficult for machines to understand and analyze images compared to human vision. Despite numerous research efforts, the current classification systems are limited in their capabilities and often produce inaccurate results.

To overcome these limitations, our system employs a deep learning algorithm to achieve superior results in the area of computer vision. Specifically, we use transfer learning, a powerful technique in deep learning, for automatic image classification. Our system utilizes the universal data set of the MobileNet architecture to classify grayscale images. Grayscale images are particularly challenging to classify, as they require significant computational power to analyze and understand.

The results of our system demonstrate the effectiveness of deep learning-based image classification using the MobileNet architecture. Our approach improves upon existing methods by providing more accurate classification of images and expanding the capabilities of current systems. Furthermore, the use of transfer learning allows for faster and more efficient training of the model, making it more practical for real-world applications.

In addition to the above, The paper also highlights the limitations of the current image classification systems, which are mostly focused on low-level image primitives and lack the ability to accurately classify images. Moreover, The paper also emphasizes the need for more powerful computational resources and advanced deep learning techniques like transfer learning to enhance the performance of image classification systems.

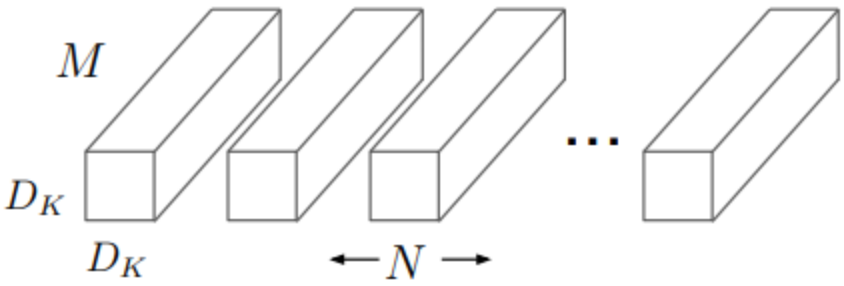
Furthermore, the paper also suggests that, this research is significant as it has the potential to greatly impact various industries and fields such as self-driving cars, medical imaging, surveillance, and security. The system can be applied to a wide range of applications where automatic image classification is needed.

.

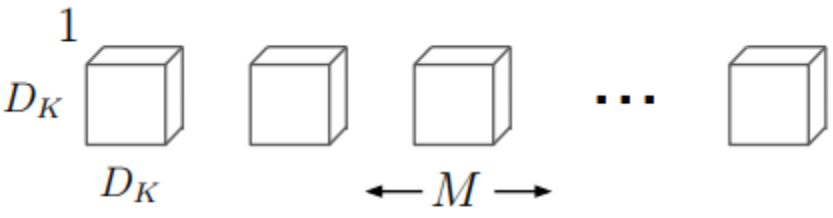
II. Background literature

MobileNet architecture:

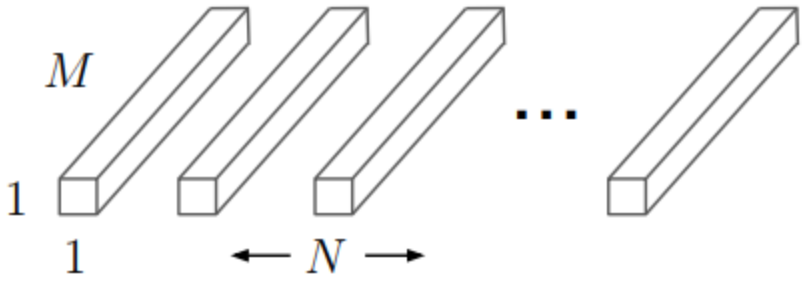
Our system uses MobileNet as it is lightweight in its architecture. It uses depth wise separable convolutions which basically means it performs a single convolution on each colour channel rather than combining all three and flattening it. This has the effect of filtering the input channels. For MobileNet the depthwise convolution applies a single filter to each input channel. The pointwise convolution then applies a 1×1 convolution to combine the outputs the depth wise convolution. A standard convolution both filters and combines inputs into a new set of outputs in one step. The depth wise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining. This factorization has the effect of drastically reducing computation and model size.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

So the overall architecture of the Mobilenet is as follows, having 30 layers with

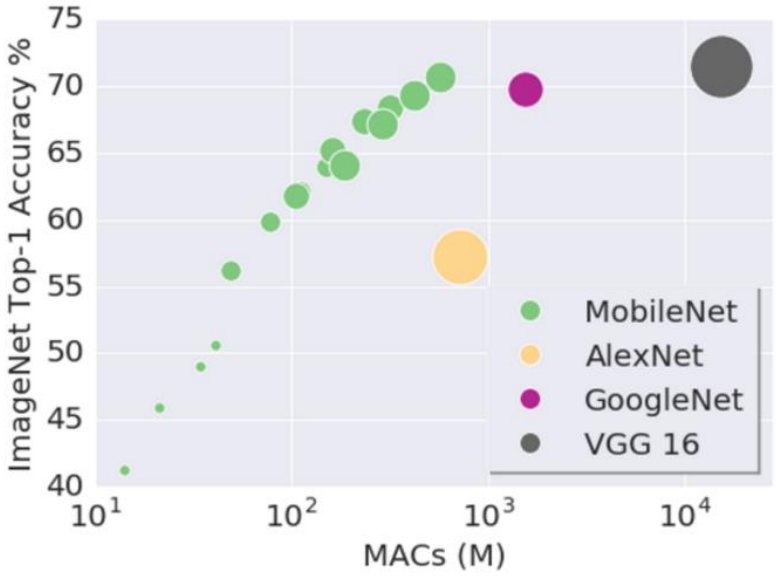
1. convolutional layer with stride 2
2. depthwise layer
3. pointwise layer that doubles the number of channels
4. depthwise layer with stride 2
5. pointwise layer that doubles the number of channels etc.

It is also very low maintenance thus performing quite well with high speed. There are also many flavours of pre-trained models with the size of the network in memory and on disk being proportional to the number of parameters being used. The speed and power consumption of the network is proportional to the number of MACs (Multiply-Accumulates) which is a measure of the number of fused Multiplication and Addition operations.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Mobilenet full architecture



ImageNet dataset:

The **ImageNet** dataset contains 14,197,122 annotated images according to the WordNet hierarchy. Since 2010 the dataset is used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a benchmark in image classification and object detection. The publicly released dataset contains a set of manually annotated training images. A set of test images is also released, with the manual annotations withheld. ILSVRC annotations fall into one of two categories: (1) image-level annotation of a binary label for the presence or absence of an object class in the image, e.g., "there are cars in this image" but "there are no tigers," and (2) object-level annotation of a tight bounding box and class label around an object instance in the image, e.g., "there is a screwdriver centered at position (20,25) with width of 50 pixels and height of 30 pixels". The ImageNet project does not own the copyright of the images, therefore only thumbnails and URLs of images are provided.

- Total number of non-empty WordNet synsets: 21841
- Total number of images: 14197122
- Number of images with bounding box annotations: 1,034,908
- Number of synsets with SIFT features: 1000
- Number of images with SIFT features: 1.2 million

III Proposed Work

App.js

The script starts by defining a few variables and constants. The constraints variable is used to set options for the video stream, in this case it only requests video with the facingMode set to "user" (so the front facing camera is used) and audio is disabled. The track variable is initially set to null, but will later be used to store the camera's video stream. The showingResults variable is initially set to false and will be used to track if the image has been taken or not.

The script then defines several constants such as cameraView which will be used to display the video stream in the HTML, cameraOutput is used to show the image after it's taken, cameraSensor used to get the size of the video stream and cameraTrigger used to trigger the function when the user clicks the button. The rest of the constants are used to display the prediction, show status and current task.

When the cameraStart function is called, it uses the navigator.mediaDevices.getUserMedia() method to access the camera and stream the video to the cameraView element.

When the user clicks the cameraTrigger button, the onclick function is fired. In the function, it first checks whether the results are showing or not. If not, it sets the size of the cameraSensor, takes the photo and assigns it to the cameraOutput element. It also applies style and shows the status of the task by calling applyStyleAndShowStatus(true) and runs predictModal() function. The function returns a prediction which is then passed to the showResults(prediction) function to format and display it. After that, the changeCurrentTask(false) function is called to change the icon of the current task and set showingResults to true so the cameraTrigger won't work again unless the user closes the results.

If showingResults is true, it calls applyStyleAndShowStatus(false) to hide the status and removePhotoFromFrame() function to remove the photo from the frame. The changeCurrentTask(true) function is also called to change the icon of the current task and set showingResults to false so the user can take another photo.

The predictModal() function is an async function that returns the prediction for the image that was passed to the model.It is worth noting that the model is not part of this script and it should be loaded first before the predictModal() function is called, the code for loading the model is missing from this script.

Code:

```
// Set constraints for the video stream
let constraints = { video: { facingMode: "user" }, audio: false };
let track = null;

let showingResults = false;

// Define constants
const cameraView = document.querySelector("#camera--view"),
    cameraOutput = document.querySelector("#camera--output"),
    cameraSensor = document.querySelector("#camera--sensor"),
    cameraTrigger = document.querySelector("#camera--trigger"),
    predictionSpan = document.querySelector("#prediction"),
    statusSpan = document.querySelector("#statusSpan"),
    currentTaskImage = document.querySelector("#currentTask");

// Access the device camera and stream to cameraView
function cameraStart() {
    navigator.mediaDevices
        .getUserMedia(constraints)
        .then(function(stream) {
            track = stream.getTracks()[0];
            cameraView.srcObject = stream;
        })
        .catch(function(error) {
            console.error("Oops. Something is broken.", error);
        });
}
// Take a picture when cameraTrigger is tapped
cameraTrigger.onclick = async function() {

  if(!showingResults) {
    cameraSensor.width = cameraView.videoWidth;
    cameraSensor.height = cameraView.videoHeight;
    cameraSensor.getContext("2d").drawImage(cameraView, 0, 0);
    cameraOutput.src = cameraSensor.toDataURL("image/webp");
    cameraOutput.classList.add("taken");
    applyStyleAndShowStatus(true);
    let prediction = await predictModal();
```

```
    statusSpan.classList.remove("statusSpanClass");
    statusSpan.innerHTML = '';
    predictionSpan.classList.add("prediction");
    predictionSpan.innerHTML = showResults(prediction);

    changeCurrentTask(false);
    showingResults = true;
  }
  else {
    applyStyleAndShowStatus(false);
    removePhotoFromFrame();
    changeCurrentTask(true);
    showingResults = false;
  }
   };
function applyStyleAndShowStatus(show) {
  if(show) {
    statusSpan.classList.add("statusSpanClass");
    statusSpan.innerHTML = "<img class='searching'
src='/images/searching.gif' alt='Scanning'>";
  }
  else {
    predictionSpan.classList.remove("prediction");
    predictionSpan.innerHTML = "";
  }
}
function removePhotoFromFrame () {
 requestAnimationFrame(function() {
    cameraOutput.classList.add("leaveFrame");
  });
    updateClass();
}
async function updateClass() {
  // cameraOutput.classList.remove("taken");
  setTimeout(function() {
    cameraOutput.src = "";
    cameraOutput.classList.remove("taken");
    cameraOutput.classList.remove("leaveFrame");
  }, 750);
}
function changeCurrentTask (ready) {
  if(ready) {
    currentTaskImage.src = "/images/search.png";
  }
  else {
```

```
    currentTaskImage.src = "/images/cancel.png";
  }
}
function showResults(props) {
    let html = "<div style='margin-bottom: 0.5rem;'>Picture of
a:</div><ul class='predictions'>"
    props.forEach(element => {
      html  += "<li>" +
                  "<div class='predictionDiv'>" + element.className +
"</div>"
                  + "<div class='predictionPercentage'>" +
getPercentage(parseFloat(element.probability)) + "</div>"
               + "</li>";  });
    html += "</ul>";
    return html;
}
// function to conver the string input 0.5909343 to a percentage
function getPercentage (number) {
    return (number * 100).toFixed(2) + "%";
}
async function predictModal () {
  let prediction;
  let img = document.getElementById('camera--output');

// Load the model.
  await mobilenet.load().then(async model => {
  // Classify the image.
      await model.classify(img).then(predictions => {
          prediction = predictions;
      });
  });
  return prediction;
}
// Start the video stream when the window loads
window.addEventListener("load", cameraStart, false);
if('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('serviceworker.js')
      .then((reg) => console.log('Success: ', reg.scope))
      .catch((err) => console.log('Failure: ', err));
  })
}
```

Serviceworker.js:

This is a Service Worker script, which is a JavaScript file that runs separately from your web page and can be used to control the caching of assets and manage network requests. Service Workers allow web pages to work offline, or on low-quality networks, by providing a way to cache resources and provide a fallback experience if the network is unavailable.

The script starts by defining some constants such as CACHE which is the name of the current cache, OFFLINE which is the URL to offline HTML document, and AUTO_CACHE which is an array of URLs of assets to immediately cache.

When the service worker is first installed, the install event is fired, and the self.addEventListener("install" function is called. The function will open the cache named CACHE and add all the URLs in the AUTO_CACHE array to it. It will also call self.skipWaiting() function which tells the browser that the new version of the service worker should be activated immediately, rather than waiting for all tabs to be closed.

The activate event is fired when the service worker is activated and the self.addEventListener("activate" function is called. This function will iterate over all the cache names, filter out the caches with the name other than CACHE and delete them. It will also call self.clients.claim() which tells the browser to activate the service worker for the current page.

The fetch event is fired whenever a network request is made from the page, and the self.addEventListener("fetch" function is called. This function checks whether the request URL starts with the origin of the current page and the method is GET, if not it simply returns.

The function then use the browser's built-in fetch API to make a request to the server for the requested resource. If the request is successful, it will update the cache with the new version, clone the response and display it.

If the request fails, it checks the cache for a version of the resource. If it finds one, it will use it, otherwise it will display an offline HTML document.

It is worth noting that similar to the previous script this service worker script is not complete and it should be registered and used in conjunction with the web app code in order to take full advantage of the service worker capabilities.

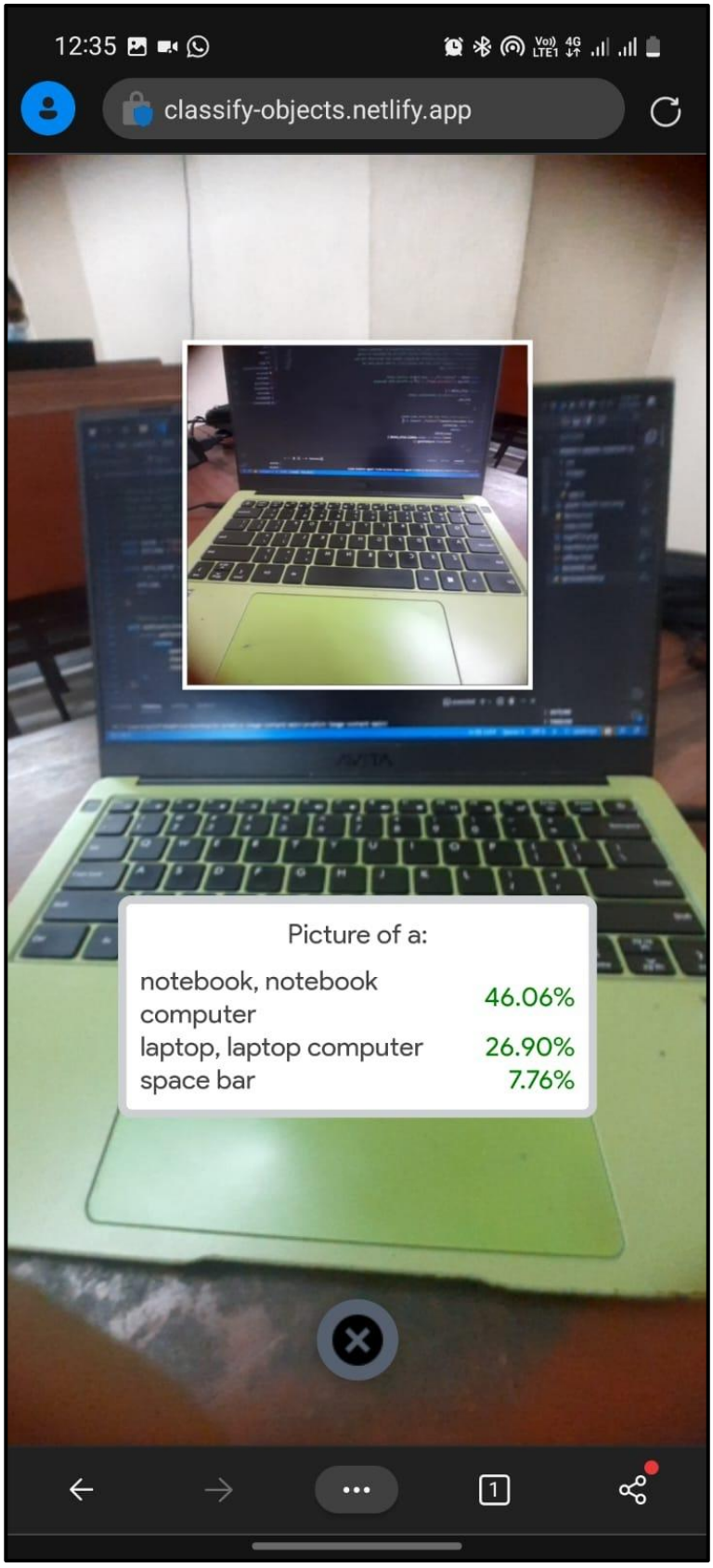
Code:

```
/**
 * Unlike most Service Workers, this one always attempts to download assets
 * from the network. Only when network access fails do we fallback to using
 * the cache. When a request succeeds we always update the cache with the new
 * version. If a request fails and the result isn't in the cache then we
 * display an Offline page.
 */
const CACHE = "content-v1"; // name of the current cache
const OFFLINE = "/offline.html"; // URL to offline HTML document

const AUTO_CACHE = [
    // URLs of assets to immediately cache
    OFFLINE,
    "/" ];
// Iterate AUTO_CACHE and add cache each entry
self.addEventListener("install", (event) => {
    event.waitUntil(
        caches
            .open(CACHE)
            .then((cache) => cache.addAll(AUTO_CACHE))
            .then(self.skipWaiting())
    );
});
// Destroy inapplicable caches
self.addEventListener("activate", (event) => {
    event.waitUntil(
        caches
            .keys()
            .then((cacheNames) => {
                return cacheNames.filter((cacheName) => CACHE !== cacheName);
            })
            .then((unusedCaches) => {
                console.log("DESTROYING CACHE", unusedCaches.join(","));
                return Promise.all(
                    unusedCaches.map((unusedCache) => {
                        return caches.delete(unusedCache);
                    })
                );
```

```
            })
            .then(() => self.clients.claim()) );
});
 self.addEventListener("fetch", (event) => {
    if (
        !event.request.url.startsWith(self.location.origin) ||
        event.request.method !== "GET"
    ) {
        // External request, or POST, ignore
        return void event.respondWith(fetch(event.request));
    }
     event.respondWith(
        // Always try to download from server first
        fetch(event.request)
            .then((response) => {
                // When a download is successful cache the result
                caches.open(CACHE).then((cache) => {
                    cache.put(event.request, response);
                });
                // And of course display it
                return response.clone();
            })
            .catch((_err) => {
                // A failure probably means network access issues
                // See if we have a cached version
                return caches.match(event.request).then((cachedResponse) => {
                    if (cachedResponse) {
                        // We did have a cached version, display it
                        return cachedResponse;
                    }
                    // We did not have a cached version, display offline page
                    return caches.open(CACHE).then((cache) => {
                        const offlineRequest = new Request(OFFLINE);
                        return cache.match(offlineRequest);
                    });
                });
            })
    );
});
```


12:36
classify-objects.netlify.app
Picture of a:
pool table, billiard table, snooker table 21.20%
prison, prison house 21.13%
ping-pong ball 12.46%


12:35
classify-objects.netlify.app
Picture of a:
notebook, notebook computer 46.06%
laptop, laptop computer 26.90%
space bar 7.76%


12:35
classify-objects.netlify.app
Picture of a:
safety pin 10.16%
spotlight, spot 5.97%
quill, quill pen 5.13%

IV Conclusion

In conclusion, building an image classifier using deep learning and transfer learning is a powerful way to accurately classify images. By using a pre-trained model, such as MobileNet, as the base architecture and fine-tuning it on a new dataset, you can leverage the knowledge learned from a large dataset and apply it to a new problem with a smaller dataset. This approach can save a significant amount of time and resources compared to training a model from scratch.

The architecture of MobileNet is well-suited for deployment on mobile and embedded devices due to its small size and efficient use of computation. By using this architecture and the transfer learning technique, it allows you to use the image classifier on mobile devices with limited computation power.

Moreover, by allowing the user to take an image and predict it in real-time, you are providing a user-friendly and interactive experience. This approach allows users to quickly and easily access the classification results, which can be useful in a variety of applications such as object recognition, automated image tagging, and many more.

It's important to keep in mind that when making a conclusion, it's also important to identify any limitations or shortcomings of the model and the dataset that it was trained on. In future, it would be a good idea to test your model with a larger, more diverse dataset to further improve its accuracy and performance.

However, overall, the image classifier you've created is a powerful tool that has the potential to make image classification more accessible and user-friendly