# Final Project – ResNet Architecture for Image Classification

Team Member
Nitin Kulkarni: 50337029
Sai Charith: 50320452
Xiao Zhang: 50312346

Our aim is to build a state of the art ResNet model and compare the results with older models such AlexNet. The aim is to understand how the different architectures affect the performance parameters such as training time, classification accuracy etc.

The dataset we are going to use is ImageNet, which is an image database organized according to the WordNet hierarchy, in which each node of the hierarchy is depicted by hundreds and thousands of images. It contains an average of 500 images per class and we use 30 classes which contain around 1000 pictures per class.

We first build the image downloader which can drag images from the ImageNet Dataset.

Please check how the image downloader works. All the explanations are listed below the code, which have the line marks which you can find corresponding notes in the code.

```
def main():
    parser = argparse.ArgumentParser(description='ImageNet image scraper')
    parser.add_argument('-scrape_only_flickr', default=True, type=lambda x: (str(x).lower() == 'true')) #23
    parser.add_argument('-number_of_classes', default=20, type=int) # How many class you want.
    parser.add_argument('-images_per_class', default=750, type=int) # The minimum number of images per class.
    parser.add_argument('-data_root', default='./images', type=str) # where to store the images.
    parser.add_argument('-use_class_list', default=True, type=lambda x: (str(x).lower() == 'true')) # what class can you
pick.
    parser.add_argument('-class_list', # Actually class code. For example:
http://www.image-net.org/api/text/imagenet.synset.geturls?wnid=n00006484
                                default=['n00006484', 'n00007846', 'n00017222', 'n00021265', 'n03902125',
                                         'n00451635', 'n03082979', 'n01605630', 'n01741943', 'n01877134',
                                         'n01887787', 'n01910747', 'n02131653', 'n02437136', 'n02676938',
                                         'n02694662', 'n02686568', 'n04192698', 'n02773037', 'n04226826'], nargs='*')
    parser.add_argument('-debug', default=False, type=lambda x: (str(x).lower() == 'true'))
    parser.add_argument('-multiprocessing_workers', default=50, type=int) # Do the things in parallel.
    args, args_other = parser.parse_known_args()
    if args.debug:
        logging.basicConfig(filename='imagenet_scarper.log', level=logging.DEBUG)
    if len(args.data_root) == 0:
        logging.error("-data_root is required to run downloader!")
        exit()

    def imagenet_api_wnid_to_urls(wnid):
        return f'http://www.image-net.org/api/text/imagenet.synset.geturls?wnid={wnid}' #102
    current_folder = os.path.dirname(os.path.realpath(__file__))
    class_info_json_filename = 'imagenet_class_info.json'
    class_info_json_filepath = os.path.join(current_folder, class_info_json_filename)
    with open(class_info_json_filepath) as class_info_json_f:
        class_info_dict = json.load(class_info_json_f)
    classes_to_scrape = []  #111
    if args.use_class_list: #112
        for item in args.class_list:
            classes_to_scrape.append(item) #114
            if item not in class_info_dict:
                logging.error(f'Class {item} not found in ImageNete')
                exit()
    elif not args.use_class_list:
        potential_class_pool = [] #120
        for key, val in class_info_dict.items():
            if args.scrape_only_flickr:
                if int(val['flickr_img_url_count']) * 0.9 > args.images_per_class: #123
                    potential_class_pool.append(key)
```

```
            else:
                if int(val['img_url_count']) * 0.8 > args.images_per_class: #126
                    potential_class_pool.append(key)
        if len(potential_class_pool) < args.number_of_classes:
            logging.error(
                f"With {args.images_per_class} images per class there are {len(potential_class_pool)} to choose from.")
            logging.error(f"Decrease number of classes or decrease images per class.")
            exit() #133
        picked_classes_idxes = np.random.choice(len(potential_class_pool), args.number_of_classes, replace=False)
        for idx in picked_classes_idxes: #135
            classes_to_scrape.append(potential_class_pool[idx])
    print("Picked the following clases: \nCount: %s" % len(classes_to_scrape))  #137
    print([class_info_dict[class_wnid]['class_name'] for class_wnid in classes_to_scrape])
    if not os.path.isdir(args.data_root):
        os.mkdir(args.data_root)
    def add_debug_csv_row(row):
        with open('stats.csv', "a") as csv_f:
            csv_writer = csv.writer(csv_f, delimiter=",")
            csv_writer.writerow(row)
    class MultiStats:
        def __init__(self):
            self.lock = Lock()
            self.stats = dict(
                all=dict(
                    tried=Value('d', 0),
                    success=Value('d', 0),
                    time_spent=Value('d', 0),
                ),
                is_flickr=dict(
                    tried=Value('d', 0),
                    success=Value('d', 0),
                    time_spent=Value('d', 0),
                ),
                not_flickr=dict(
                    tried=Value('d', 0),
                    success=Value('d', 0),
                    time_spent=Value('d', 0),
                )
            )
        def inc(self, cls, stat, val):
            with self.lock:
                self.stats[cls][stat].value += val
        def get(self, cls, stat):
            with self.lock:
                ret = self.stats[cls][stat].value
            return ret
    multi_stats = MultiStats()
    if args.debug:
        row = [
            "all_tried",
            "all_success",
            "all_time_spent",
            "is_flickr_tried",
            "is_flickr_success",
            "is_flickr_time_spent",
            "not_flickr_tried",
            "not_flickr_success",
            "not_flickr_time_spent"
        ]
        add_debug_csv_row(row)
    def add_stats_to_debug_csv():
        row = [
            multi_stats.get('all', 'tried'),
            multi_stats.get('all', 'success'),
            multi_stats.get('all', 'time_spent'),
            multi_stats.get('is_flickr', 'tried'),
            multi_stats.get('is_flickr', 'success'),
            multi_stats.get('is_flickr', 'time_spent'),
```

```
                multi_stats.get('not_flickr', 'tried'),
                multi_stats.get('not_flickr', 'success'),
                multi_stats.get('not_flickr', 'time_spent'),
        ]
        add_debug_csv_row(row)
    def print_stats(cls, print_func):
        actual_all_time_spent = time.time() - scraping_t_start.value
        processes_all_time_spent = multi_stats.get('all', 'time_spent')
        if processes_all_time_spent == 0:
                actual_processes_ratio = 1.0
        else:
                actual_processes_ratio = actual_all_time_spent / processes_all_time_spent
        print_func(f'STATS For class {cls}:')
        print_func(f' tried {multi_stats.get(cls, "tried")} urls with'
                           f' {multi_stats.get(cls, "success")} successes')
        if multi_stats.get(cls, "tried") > 0:
                print_func(
                    f'{100.0 * multi_stats.get(cls, "success") / multi_stats.get(cls, "tried")}% '
                    f'success rate for {cls} urls ')
        if multi_stats.get(cls, "success") > 0:
                print_func(
                    f'{multi_stats.get(cls, "time_spent") * actual_processes_ratio / multi_stats.get(cls, "success")}'
                    f' seconds spent per {cls} successful image download')
    lock = Lock()
    url_tries = Value('d', 0)
    scraping_t_start = Value('d', time.time())
    class_folder = ''
    class_images = Value('d', 0)
    def get_image(img_url): #232
        def check():
                with lock:
                        cls_imgs = class_images.value
                if cls_imgs >= args.images_per_class:
                        return True
        if len(img_url) <= 1:
                return
        if check():
                return
        logging.debug(img_url)
        cls = ''
        if 'flickr' in img_url:
                cls = 'is_flickr'
        else:
                cls = 'not_flickr'
                if args.scrape_only_flickr:
                        return
        t_start = time.time()
        def finish(status):
                t_spent = time.time() - t_start
                multi_stats.inc(cls, 'time_spent', t_spent)
                multi_stats.inc('all', 'time_spent', t_spent)
                multi_stats.inc(cls, 'tried', 1)
                multi_stats.inc('all', 'tried', 1)
                if status == 'success':
                        multi_stats.inc(cls, 'success', 1)
                        multi_stats.inc('all', 'success', 1)
                elif status == 'failure':
                        pass
                else:
                        logging.error(f'No such status {status}!!')
                        exit()
                return
        with lock:
                url_tries.value += 1
                if url_tries.value % 250 == 0:
                        print(f'\nScraping stats:')
                        print_stats('is_flickr', print)
                        print_stats('not_flickr', print)
```

```
                    print_stats('all', print)
                    if args.debug:
                        add_stats_to_debug_csv()
            try:
                img_resp = requests.get(img_url, timeout=1) #289
            except ConnectionError:
                logging.debug(f"Connection Error for url {img_url}")
                return finish('failure')
            except ReadTimeout:
                logging.debug(f"Read Timeout for url {img_url}")
                return finish('failure')
            except TooManyRedirects:
                logging.debug(f"Too many redirects {img_url}")
                return finish('failure')
            except MissingSchema:
                return finish('failure')
            except InvalidURL:
                return finish('failure')
            if 'content-type' not in img_resp.headers:
                return finish('failure')
            if 'image' not in img_resp.headers['content-type']:
                logging.debug("Not an image")
                return finish('failure')
            if len(img_resp.content) < 1000:
                return finish('failure')
            logging.debug(img_resp.headers['content-type'])
            logging.debug(f'image size {len(img_resp.content)}')
            img_name = img_url.split('/')[-1]
            img_name = img_name.split("?")[0]
            if len(img_name) <= 1:
                return finish('failure')
            img_file_path = os.path.join(class_folder, img_name)
            logging.debug(f'Saving image in {img_file_path}')
            if check():
                return
            with open(img_file_path, 'wb') as img_f:
                img_f.write(img_resp.content) #330
                with lock:
                    class_images.value += 1
                logging.debug(f'Scraping stats')
                print_stats('is_flickr', logging.debug)
                print_stats('not_flickr', logging.debug)
                print_stats('all', logging.debug)
                return finish('success')
    print(f"Multiprocessing workers: {args.multiprocessing_workers}") #341
    for class_wnid in tqdm(classes_to_scrape):
        class_name = class_info_dict[class_wnid]["class_name"] #344
        url_urls = imagenet_api_wnid_to_urls(class_wnid) #345
        time.sleep(0.05)
        resp = requests.get(url_urls) #348
        class_folder = os.path.join(args.data_root, class_name)
        if not os.path.exists(class_folder):
            os.mkdir(class_folder)
        class_images.value = 0
        urls = [url.decode('utf-8') for url in resp.content.splitlines()] #356
        with ThreadPool(processes=args.multiprocessing_workers) as p: #358
            p.map(get_image, urls) #359
if __name__ == '__main__':
    main()
```

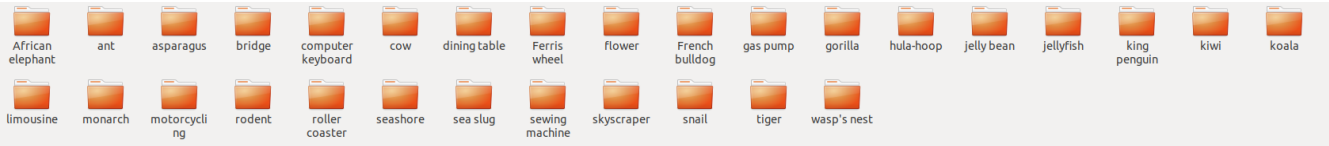
## Note:

**Rules of reading the notes**

# 'X' means X is a variable defined in the program
# [$ XXXX] means what you input in terminal
# "X" means X is a name or address
# Function[XXXXX] means the function name is XXXXX
# ction*[XXXXX] means XXXXis a package

```
    Start:
    # First check what is f-string formatting -> #101 (this is line number. Please jump to see)
    # Reference: https://www.datacamp.com/community/tutorials/f-string-formatting-in-python
    # Example: f'xx_{shit}_xx' shit = 'fat_fuck' is equivalent to xx_fat_fuck_xx, remember to remove '{}'.


    How the data flows:
    # if input [$ use_class_list true] in command line or doesn't input anything(default true) -> #111
    # then read what your input to parser 'class_list' (should look like n00006484), assign it to 'item' -> #112
    # Then check if 'item' in json file (it store all the class codes in image net). You can't input class which doesn't exist -> #114
    # if you set [$ use_class_list false]. Then read the code of image classes from json file. -> #120
    # The jason file is a dictionary which looks like {"n00004475": {"img_url_count": 8, "flickr_img_url_count": 6, "class_name":
"organism"}
    # n00004475 is one of the 'key' of the dic. It is the numeric code for class "organism".
    # read the value of the 'key' (its number of images belong to that class). if big enough then put 'key' into a set
'potential_class_pool' -> #123, 126
    # randomly pick codes from 'potential_class_pool', store them to 'classes_to_scrape'. The number of codes you pick is -> #135
    # defined by 'args.number_of_classes', (default 20). -> #23
    # 'picked_classes_idxes' is used to store random index. -> #133
    # till now the codes are stored in 'classes_to_scrape' -> #135
    # And you print it out -> #137
    # jump jump jump many lines -> #341
    # read codes from 'classes_to_scrape' and assign them to 'class_wnid' -> #341
    # tqdm is bar function it shows how many codes(class) has been processed. -> #10
    # send 'class_wnid' to function Function(imagenet_api_wnid_to_urls) -> #344
    # inside the function it inserts the string 'class_wnid' (codes) to a URL -> #102
    # http://www.image-net.org/api/text/imagenet.synset.geturls?wnid={'class_wnid'} -> #102
    # remember it is f-string formatting so remove the {}
    # it actually looks like:
    # http://www.image-net.org/api/text/imagenet.synset.geturls?wnid=n00004475  (You can click it to see)
    # yeah it has "text" in the url path so the website above has no pictures but the url of the pictures.
    # Now the paths(their corresponding websites contain the paths of all images) are store in 'url_urls' -> #345
    # send request to these website by Function*(requests('url_urls')) and get all the text documents (many url) -> #348
    # store these text in 'resp' -> #348
    # decode the url text and store in 'urls' -> #356
    # look the code here
    #         with ThreadPool(processes=args.multiprocessing_workers) as p: -> #358
    #             p.map(get_image, urls) -> #359
    # parallel processing: send parameter 'urls' to function[get_image] and process them simultaneously -> #359
    # Inside the function, parameter 'urls' is assigned to 'img_url' -> #232
    # use Function*[requests('img_url')] again to send requests to the website -> #289
    # The last time we get url. This time is real images.
    # store images in 'img_resp' -> #289
    # open the folder and write  'img_resp' into it. -> #330
    # All the others are locking in parallel processing and logging system.
```

With this downloader we get a image set of 30 different classes, which contains more than 1000 pictures per class.




The labels are ["African elephant", "ant", "asparagus", "bridge", "computer keyboard","cow","dining table","Ferris wheel","flower","French bulldog","gas pump","gorilla","hula-hoop","jelly bean","jellyfish","king penguin","kiwi","koala","limousine","monarch","motorcycling","rodent","roller coaster","seashore","sea slug","sewing machine","skyscraper","snail","tiger","wasp's nest"], as you can see in the screenshot.

Then we built the pipe-line to install and process these images. The files' structure is listed below:
*downloader.py: The downloader which helps to download the images you want from ImageNet*

*augmentation.py: It is used to do the data augmentation. All the numbers of different classes of images will increase to exact 1000 after augmentation.*

*imagenet_class_info.json: It is a dictionary store the class names and their corresponding code.*

*image_links: The url store the image set we used in the project.*

*Alex_Net_Final.ipynb: Build AlexNet.*
*ResNet18.ipynb: Build ResNet18*
*ResNet34.ipynb: Build ResNet34*
ResNet50.ipynb: *Build ResNet50, the pipeline of the above 3 models are independent. One does not need to run the files in a order.*

And let me introduce the data processing pipe-line in the models (it is the same in all four models).

```
physical_devices = tf.config.experimental.list_physical_devices('GPU')
print("physical_devices----------", len(physical_devices))
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

First we enable the GPU version of TF to deal with the challenge. Function *tf.config.experimental.set_memory_growth* is used to prevent the kernel run out of all the memory of the GPU.

Then, we perform the data augmentation. We can import the data first and do the augmentation in the Jupyter notebook. However this will significantly slower the processing and we have to do the augmentation over and over again if we rerun the kernel. Thus we use a separate augmentation file to generate more pictures and store them in the local folder.

```
cv_img = []
count = 0
for img in glob.glob(".\images/tiger\*.jpg"):
    n= cv2.imread(img)
    cv_img.append(n)
    count += 1
    if count > 149:
        break
print('Length Images:', len(cv_img)) # first test the size of the image class
# cv_img = np.array(cv_img)
# cv_img = np.reshape(cv_img, (len(cv_img), 224, 224, 3))
def flip_horizontally(image): # flip the image manually
    """This function flips a given image vertically."""
    vertically_flipped_image = copy.deepcopy(image)
    center = int(len(image[0]) / 2)
    for i, row in enumerate(image):
        for j in range(center):
            vertically_flipped_image[i][j] = image[i][(len(image[0]) - 1) - j]
            vertically_flipped_image[i][(len(image[0]) - 1) - j] = image[i][j]
    return vertically_flipped_image
i = 851
for image in cv_img:
```

```
    horizontally_flipped_img = flip_horizontally(image)
    cv2.imwrite(f'{i}.jpg', horizontally_flipped_img)
    i += 1
```

Then we get the document path with function os.listdir and store all the picture names and their corresponding folder name(the folder name is also the images' label) in *x_train* and *y_train.* The images and their labels are stored separately but in the same order.

```
filenames= os.listdir ('./images')
x_train = []
y_train = []
for folder in filenames: # Read the file name.
    for image in os.listdir(f'./images/{folder}'):# Use folder as parameter.
        img = read_image(f'./images/{folder}/{image}') # Read the files in each folder.
        x_train.append(img) # Append them (the file and label) in exact same order.
        y_train.append(folder)
```

Then,

```
c = list(zip(x_train, y_train))
random.shuffle(c)
a, b = zip(*c)
```

What we do here is to *zip* the *x_train* and *y_train* together, which means we attached the elements in x and y which have the same index together. And then we shuffle the integrated set and then do the reverse to split the integrated set back to two separate lists, which contain images and their labels correspondingly. Finally we get a random ordered object list and label list.

Then we launched a encoder. The motivation is that Keras cannot deal with 'string' type labels, we have to convert the string labels to numbers.
We first define the encoder.

```
label_encoder = preprocessing.LabelEncoder()
```

```
label_encoder.fit(["African elephant", "ant", "asparagus", "bridge", "computer keyboard","cow","dining table","Ferris wheel","flower","French bulldog","gas pump","gorilla","hula-hoop","jelly bean","jellyfish","king penguin","kiwi","koala","limousine","monarch","motorcycling","rodent","roller coaster","seashore","sea slug","sewing machine","skyscraper","snail","tiger","wasp's nest"])
```

Then we feed in the data

```
b_label = np.copy(b)
b_label = label_encoder.transform(b_label)
```

```
print(b_label)
```

and we can see the encoding: [11 19 11 ... 9 15 10]
The mapping between numeric code and string label looks like the following:

```
In [19]: list(label_encoder.inverse_transform([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,
```

```
Out[19]: ['African elephant',
          'Ferris wheel',
          'French bulldog',
          'ant',
          'asparagus',
          'bridge',
          'computer keyboard',
          'cow',
          'dining table',
          'flower',
          'gas pump',
          'gorilla',
          'hula-hoop',
          'jelly bean',
          'jellyfish',
          'king penguin',
          'kiwi',
          'koala',
          'limousine',
          'monarch',
          'motorcycling',
          'rodent',
          'roller coaster',
          'sea slug',
          'seashore',
          'sewing machine',
          'skyscraper',
          'snail',
          'tiger',
          "wasp's nest"]
```

And then we use *train_test_split* from *sklearn* and *cv2.normalize()* to split the dataset and perform normalization.

Finally we should do a one-hot encoding to transfer the label (now it is a numeric number refers to its corresponding image class) into a 1-dimension vector.

```
x_label_OneHot = to_categorical(training_label, 30)
y_label_OneHot = to_categorical(test_label, 30)
```

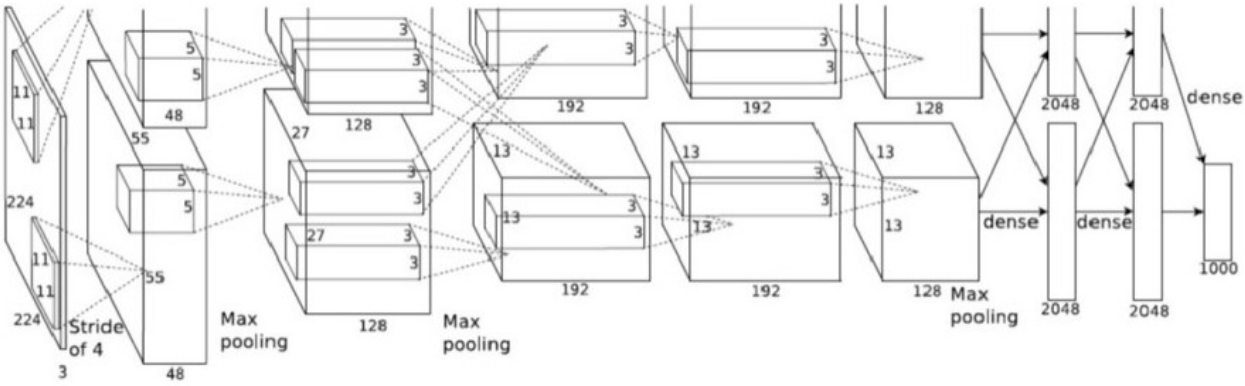
We randomly print one and see how it works.

```
print(x_label_OneHot[1])
```

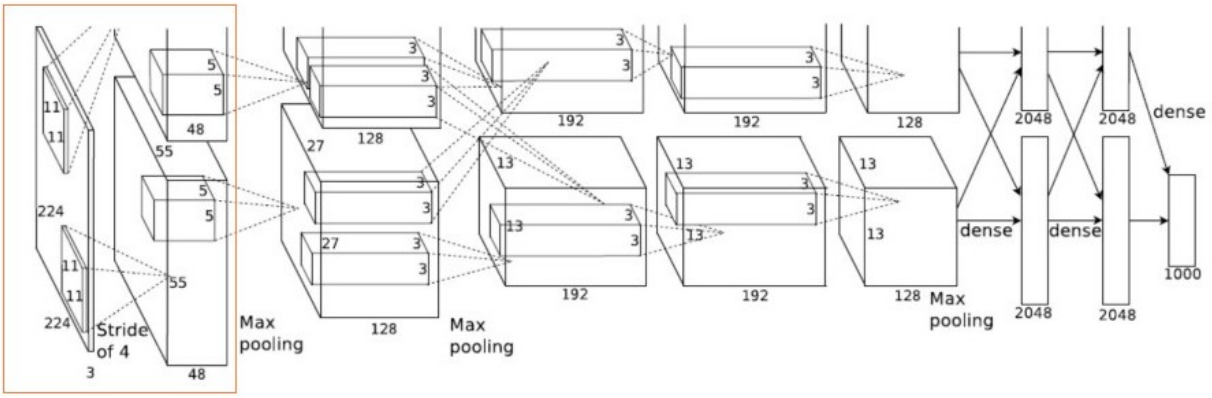
```
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
```

It means the numeric code for the first image in the training set is 8, which refers to ‘dining table’.

After all these are settled we are going to launch our model.
For the *AlexNet*, it was build exactly the same with what mentioned in the paper.



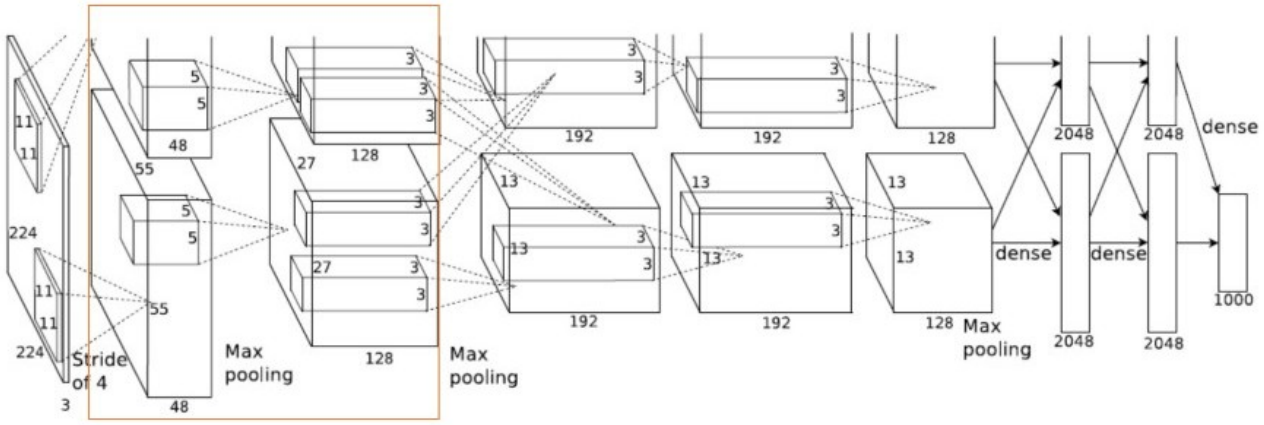

It has 8 layers, The first 5 are ConV layers and the last 3 are dense layers.
For layer1 we have:




**Layer 1** is a Convolution Layer,

- Input Image size is – 227 x 227 x 3
- Number of filters – 96
- Filter size – 11 x 11 x 3
- Stride – 4
- Layer 1 Output
- 224/4 x 224/4 x 96 = 55 x 55 x 96 (because of stride 4)
- Split across 2 GPUs – So 55 x 55 x 48 for each GPU

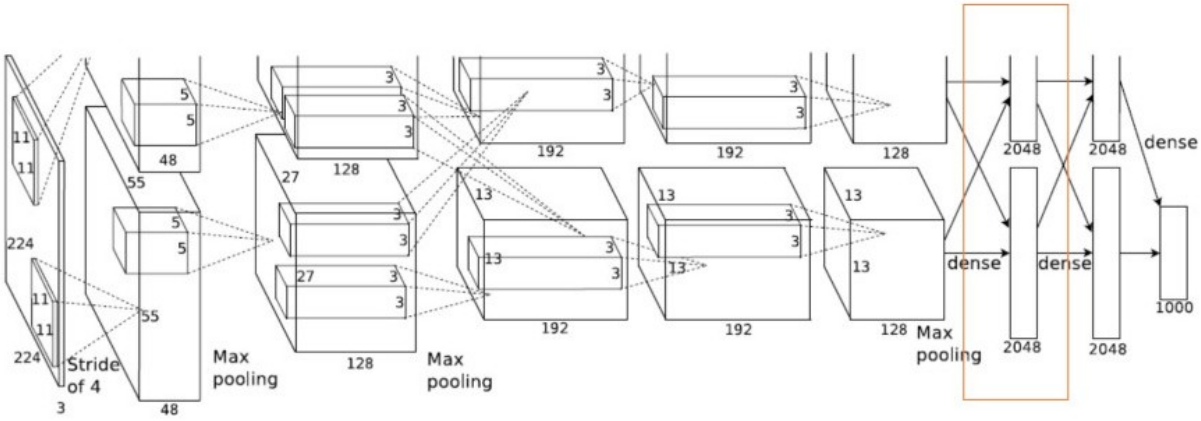
**Layer 2** is a Max Pooling Followed by Convolution




• Input – 55 x 55 x 96
• Max pooling – 55/2 x 55/2 x 96 = 27 x 27 x 96
• Number of filters – 256
• Filter size – 5 x 5 x 48
• Layer 2 Output
• 27 x 27 x 256
• Split across 2 GPUs – So 27 x 27 x 128 for each GPU

And **layer 3,4,5** all have similar features.

Finally we have **layer6.**




**Layer 6** is fully connected.

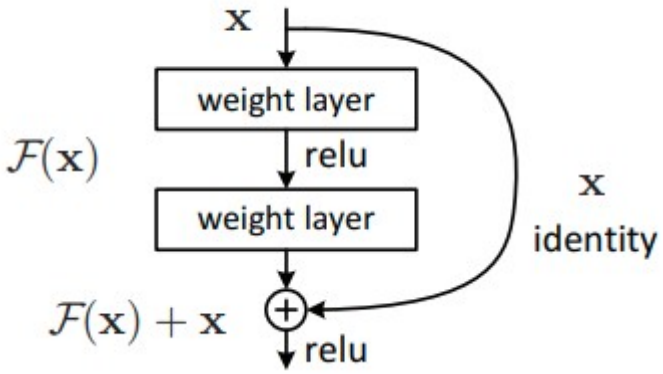
• Input – 13 x 13 x 128 – > is transformed into a vector
• And multiplied with a matrix of the following dim – (13 x 13 x 128) x 2048
• GEMV(General Matrix Vector Multiply) is used here:

And **layer 7, 8** all have similar features.

For the ResNet,
We tried **ResNet 18, 34, and 50.** Their structures are listed below.

| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride 2 | | | | |
| conv2_x | 56×56 | 3×3 max pool, stride 2 | | | | |
| | | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ |
| conv3_x | 28×28 | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$ |
| conv4_x | 14×14 | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$ |
| conv5_x | 7×7 | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ |
| | 1×1 | average pool, 1000-d fc, softmax | | | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

The most import part of doing ResNet is to make the **identity mapping** have the same shape with the direct output of the Conv layers. We can achieve this either by zero padding or 1x1 convolutional layers.




In the ResNet18, we stack the layers together and do the shortcut manually.

```
x = keras.Input((224, 224, 3))

# conv_1 layer
conv1 = keras.layers.Conv2D(filters=32, kernel_size=7, strides=1, **params_conv2d)(x)
conv1 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv1)

# conv_2_x layer
max_pool = keras.layers.MaxPool2D(padding="SAME")(conv1)
conv2_1 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(max_pool)
conv2_1 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv2_1)
conv2_2 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(conv2_1)
conv2_2 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv2_2)
# Add the two parts manually.
skip2_1 = keras.layers.BatchNormalization()(keras.layers.add([max_pool, conv2_2]))
```

```
conv2_3 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(conv2_2)
conv2_3 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv2_3)
conv2_4 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=2, **params_conv2d)(conv2_3)
conv2_4 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv2_4)

resize2 = keras.layers.Conv2D(filters=32, kernel_size=1, strides=2, padding="SAME")(skip2_1)
# Add the two parts manually.
skip2_2 = keras.layers.BatchNormalization()(keras.layers.add([resize2, conv2_4]))

# conv_3_x layer
conv3_1 = keras.layers.Conv2D(filters=64, kernel_size=3, strides=1, **params_conv2d)(skip2_2)
conv3_1 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv3_1)
conv3_2 = keras.layers.Conv2D(filters=64, kernel_size=3, strides=1, **params_conv2d)(conv3_1)
conv3_2 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv3_2)

resize3_1 = keras.layers.Conv2D(filters=64, kernel_size=1, strides=1, padding="SAME")(skip2_2)
# Add the two parts manually.
skip3_1 = keras.layers.BatchNormalization()(keras.layers.add([resize3_1, conv3_2]))

conv3_3 = keras.layers.Conv2D(filters=64, kernel_size=3, strides=1, **params_conv2d)(skip3_1)
conv3_3 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv3_3)
conv3_4 = keras.layers.Conv2D(filters=64, kernel_size=3, strides=2, **params_conv2d)(conv3_3)
conv3_4 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv3_4)

resize3_2 = keras.layers.Conv2D(filters=64, kernel_size=1, strides=2, padding="SAME")(skip3_1)
# Add the two parts manually.
skip3_2 = keras.layers.BatchNormalization()(keras.layers.add([resize3_2, conv3_4]))

# conv_4_x layer
conv4_1 = keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, **params_conv2d)(skip3_2)
conv4_1 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv4_1)
conv4_2 = keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, **params_conv2d)(conv4_1)
conv4_2 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv4_2)

resize4_1 = keras.layers.Conv2D(filters=128, kernel_size=1, strides=1, padding="SAME")(skip3_2)
# Add the two parts manually.
skip4_1 = keras.layers.BatchNormalization()(keras.layers.add([resize4_1, conv4_2]))

conv4_3 = keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, **params_conv2d)(skip4_1)
conv4_3 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv4_3)
conv4_4 = keras.layers.Conv2D(filters=128, kernel_size=3, strides=2, **params_conv2d)(conv4_3)
conv4_4 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv4_4)

resize4_2 = keras.layers.Conv2D(filters=128, kernel_size=1, strides=2, padding="SAME")(skip4_1)
# Add the two parts manually.
skip4_2 = keras.layers.BatchNormalization()(keras.layers.add([resize4_2, conv4_4]))

# conv_5_x layer
conv5_1 = keras.layers.Conv2D(filters=256, kernel_size=3, strides=1, **params_conv2d)(skip4_2)
conv5_1 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv5_1)
conv5_2 = keras.layers.Conv2D(filters=256, kernel_size=3, strides=1, **params_conv2d)(conv5_1)
conv5_2 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv5_2)

resize5_1 = keras.layers.Conv2D(filters=256, kernel_size=1, strides=1, padding="SAME")(skip4_2)
# Add the two parts manually.
skip5_1 = keras.layers.BatchNormalization()(keras.layers.add([resize5_1, conv5_2]))

conv5_3 = keras.layers.Conv2D(filters=256, kernel_size=3, strides=1, **params_conv2d)(skip5_1)
```

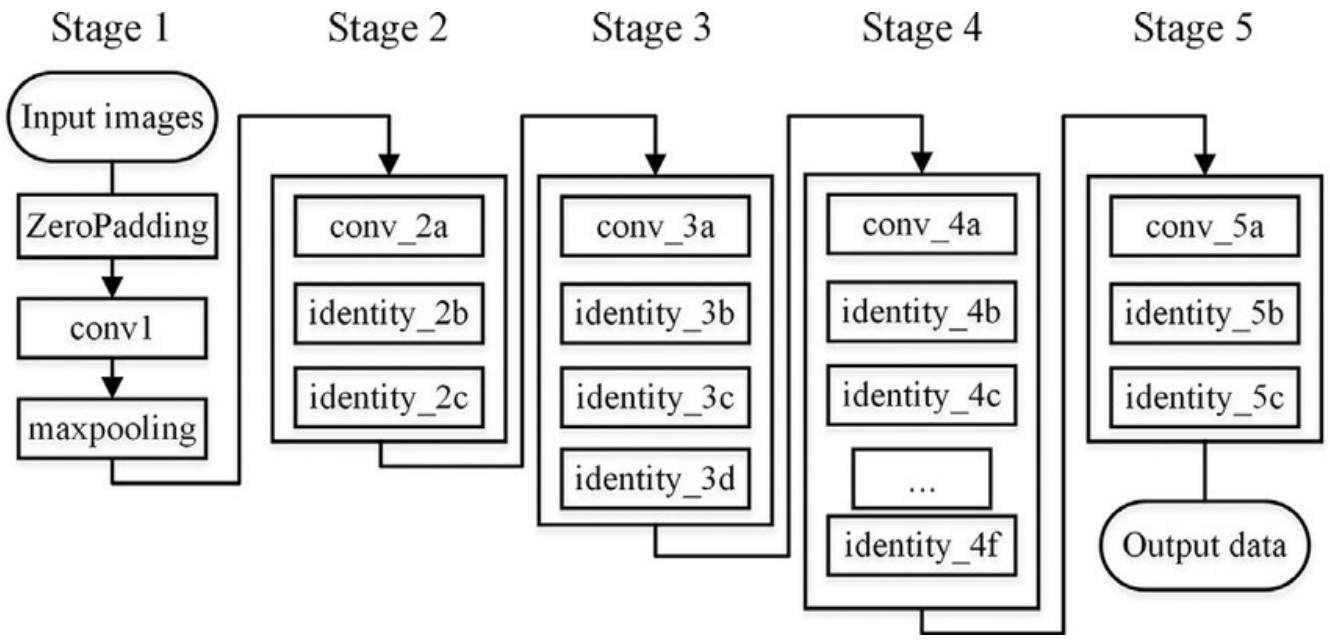
```
conv5_3 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv5_3)
conv5_4 = keras.layers.Conv2D(filters=256, kernel_size=3, strides=2, **params_conv2d)(conv5_3)
conv5_4 = keras.layers.SpatialDropout2D(.5, data_format='channels_last')(conv5_4)

resize5_2 = keras.layers.Conv2D(filters=256, kernel_size=1, strides=2, **params_conv2d)(skip5_1)
# Add the two parts manually.
skip5_2 = keras.layers.BatchNormalization()(keras.layers.add([resize5_2, conv5_4]))

avg_pool = keras.layers.AvgPool2D(strides=2)(skip5_2)
flat = keras.layers.Flatten()(avg_pool)
dense10 = keras.layers.Dense(30)(flat)
softmax = keras.layers.Softmax()(dense10)

model = keras.Model(inputs=x, outputs=softmax)
```

In the ResNet34 we try to do this with **two helper functions.**

```
# Standard block.
def Conv2d_BN(x, nb_filter, kernel_size, strides=(1,1), padding='same', name=None): # Name block can be removed.
    if name is not None: # Add operation name to distinguish this process in tensorboard and model summary.
        bn_name = name + '_bn'
        conv_name = name + '_conv'
    else:
        bn_name = None
        conv_name = None

    x = Conv2D(nb_filter, kernel_size, padding=padding, strides=strides, activation='relu', name=conv_name)(x)
    x = BatchNormalization(axis=3, name=bn_name)(x) # Batch_Normalization
    return x

# shortcut
def Conv_Block(inpt, nb_filter, kernel_size, strides=(1,1), with_conv_shortcut=False):
    x = Conv2d_BN(inpt, nb_filter=nb_filter, kernel_size=kernel_size, strides=strides, padding='same') # layer1
    x = Conv2d_BN(x, nb_filter=nb_filter, kernel_size=kernel_size, padding='same') # layer2
    if with_conv_shortcut:
        # if shortcut then run 1 layer of Conv2D without padding (since the size is the same).
        # Also there is no need to add batchnormalization. Check # Batch_Normalization for comparison.
        shortcut = Conv2d_BN(inpt, nb_filter=nb_filter, strides=strides, kernel_size=kernel_size)
        x = add([x, shortcut])
        return x
    else:
        x = add([x, inpt]) # if not shortcut then run 2 layers of Conv2D with padding. See layer1 and layer2.
        return x
```

In ResNet50 we even need to define different name blocks to help deal with the complicated structure.

**Reference:** *Shi, Zaifeng & Liu, Minghe & Cao, Qingjie & Ren, Huizheng & Tao, Luo. (2019). A data augmentation method based on cycle-consistent adversarial networks for fluorescence encoded microsphere image analysis. Signal Processing. 161. 10.1016/j.sigpro.2019.02.028.* (https://reader.elsevier.com/reader/sd/pii/S0165168419300854?token=86A5841D2FB82632751DC85F99B4BC0EDA339D1121CDE6B31FE45C1B014E4F1BBFB3CE0CAD6A0A4B5C7964C917109C88)



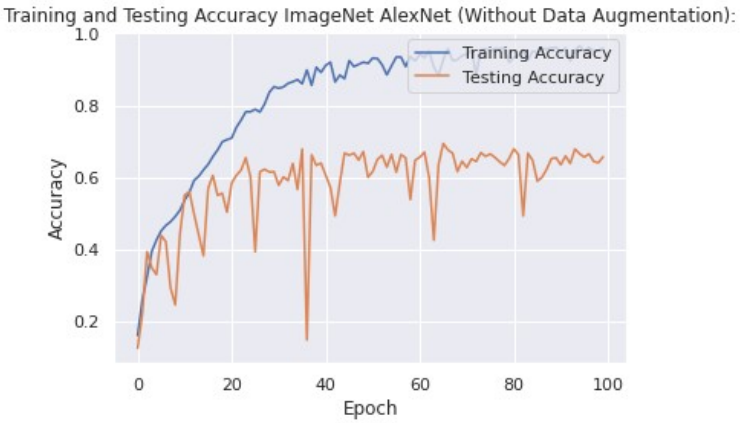

```
# ResNet model
def ResNet50(input_shape= (224,224,3), classes=30):
    X_input = Input(input_shape)
    X = ZeroPadding2D(padding= (3,3))(X_input)

    #stage 1
    X = Conv2D(64, (7,7), strides= (2,2), name= 'conv1', kernel_initializer=glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3, name= 'bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3,3), strides= (2,2))(X)

    #stage 2
    X = convolutional_block(X, f=3, filters= [64,64,256], stage=2, block= 'a', s=1)
    X = identity_block(X, f=3, filters= [64,64,256], stage=2, block='b')
    X = identity_block(X, f=3, filters= [64,64,256], stage=2, block='c')

    #stage 3
    X = convolutional_block(X, f=3, filters= [128,128,512], stage=3, block= 'a', s=2)
    X = identity_block(X, f=3, filters= [128,128,512], stage=3, block='b')
    X = identity_block(X, f=3, filters= [128,128,512], stage=3, block='c')
    X = identity_block(X, f=3, filters= [128,128,512], stage=3, block='d')

    #stage 4
    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], stage=4, block='a', s = 2)
    X = identity_block(X, f= 3, filters= [256, 256, 1024], stage=4, block='b')
    X = identity_block(X, f= 3, filters= [256, 256, 1024], stage=4, block='c')
    X = identity_block(X, f= 3, filters= [256, 256, 1024], stage=4, block='d')
    X = identity_block(X, f= 3, filters= [256, 256, 1024], stage=4, block='e')
    X = identity_block(X, f= 3, filters= [256, 256, 1024], stage=4, block='f')

    #stage 5
    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], stage=5, block='a', s = 2)
    X = identity_block(X, f= 3, filters= [512, 512, 2048], stage=5, block='b')
    X = identity_block(X, f= 3, filters= [512, 512, 2048], stage=5, block='c')

    X = AveragePooling2D()(X)
```

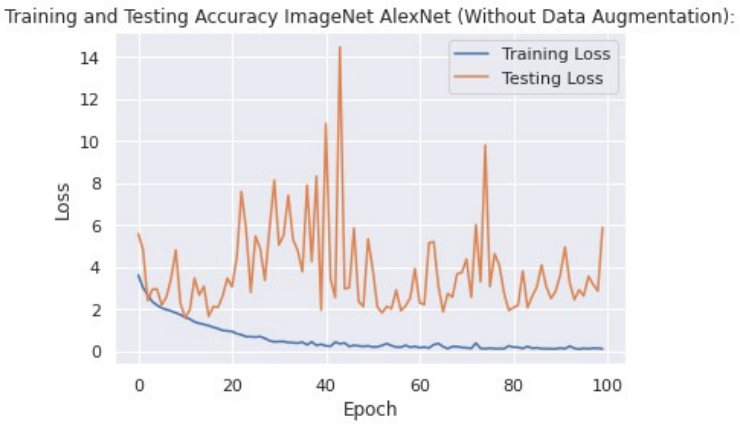
```
    X = Flatten()(X)
    X = Dense(classes, activation= 'softmax', name='fc' + str(classes), kernel_initializer= glorot_uniform(seed= 0))(X)

    model = Model(inputs = X_input, outputs = X, name='ResNet50')
    return model
```

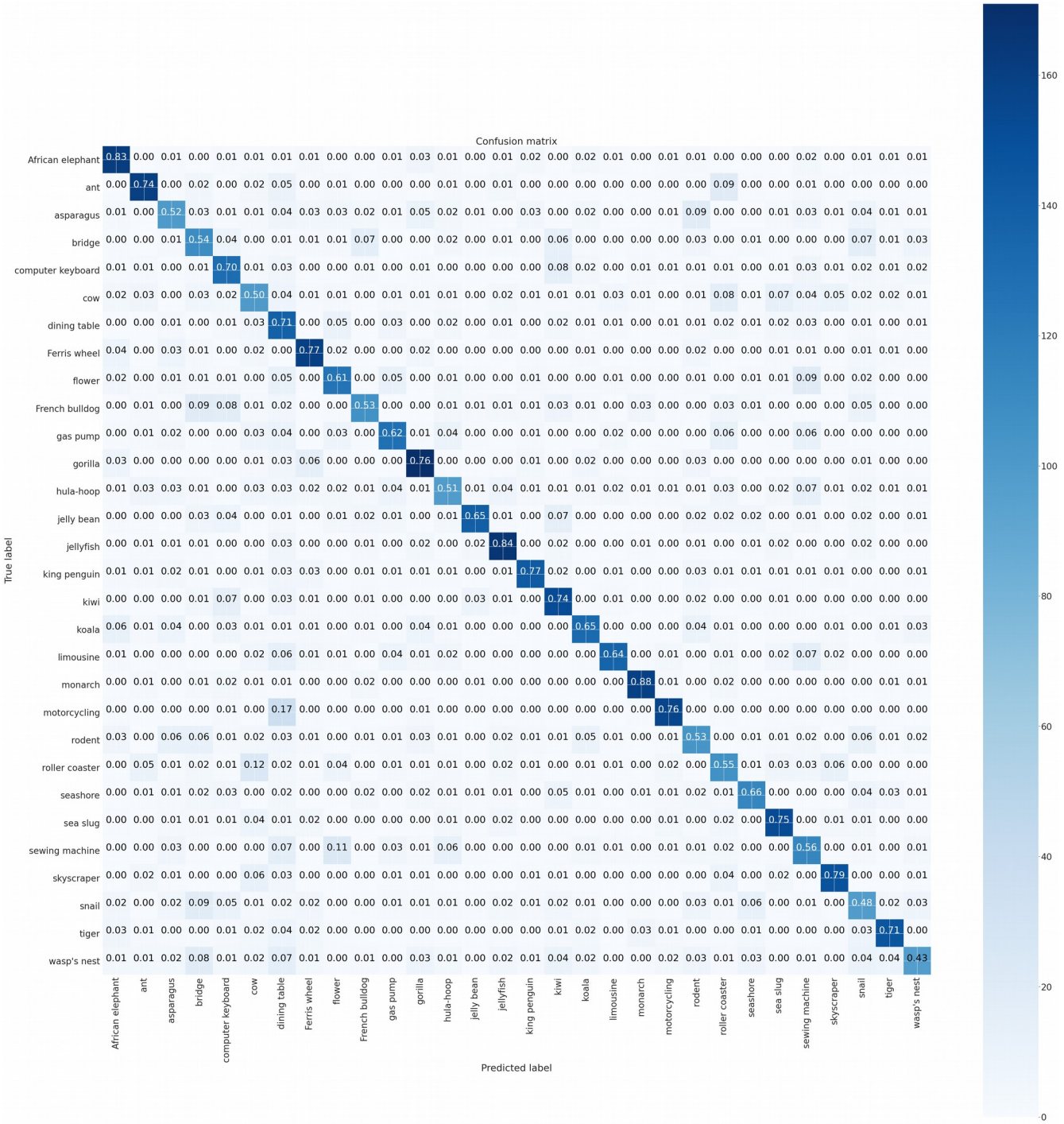
It also has two helper functions which are convolution block and identity block(short cut block). You can check it in the Jupyter file.

And finally we get the results.

For the AlexNet,
AlexNet Accuracy on ImageNet without data-augmentation: **0.6568333506584167 %**

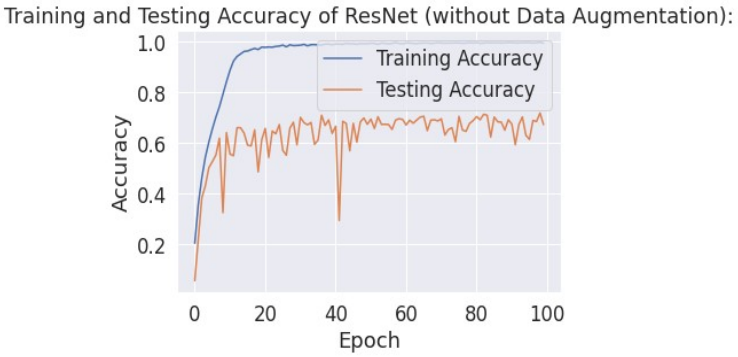


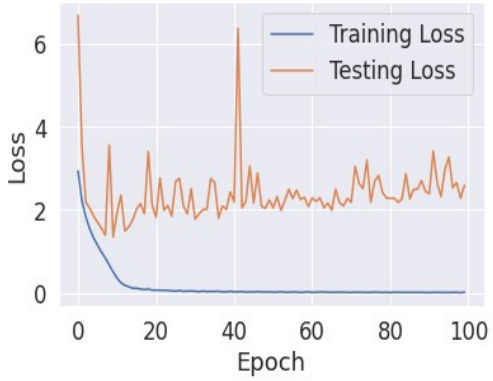



For the ResNet34,

ResNet Accuracy on ImageNet without data-augmentation: **0.6725000143051147 %**



Training and Testing Loss of ResNet (without Data Augmentation):




For the ResNet50 and ResNet18, we have:
loss: 5.3707e-04 - accuracy: 1.0000 - val_loss: 2.4976 - val_accuracy: **0.6288** – **ResNet50**
loss: 2.1113 - accuracy: 0.5185 - val_loss: 2.1482 - val_accuracy: **0.5210** – **ResNet18**

We notice that ResNet generally behave much better than AlexNet. You may see here AlexNet records 65.5% compared 67.2% of ResNet34. The difference seems not significant. But the reality is that I fit the model in AlexNet for multiple times to reach its limit, while all the ResNets get trained for only one single time (it takes too much time so I can't fit the model for more times). There are two other important discoveries:

**1.** ResNet18 and ResNet50 all perform worse then ResNet34. We can see that the training accuracy of ResNet50 reaches **100%**, while validation accuracy is only **62.8%**. We can safely assume there is a over-fitting here. On the other hand, the ResNet18 only get **51.8%** in training accuracy, which indicates a classic under-fitting. ResNet50 is too much; RestNet18 is too less; and ResNet34 is perfect for our image data size. We can learn from here that we should never blindly pick complex or simple model for our data set. The model must exactly match each others.

**2.** The training time doesn't grow linearly with the parameters number. We can see the parameters number in ResNet34 is **22,684,830**, while ResNet50 has **24,087,582**. ResNet50 only records **6%** more parameters than ResNet34 but the training time increases like crazy, which is **81s/epoch** Vs **144s/epoch**. We must always consider the time consumption when we plan to use deeper model, since the time consumption growth may be out of control.