Erik Hallin-Logo Classification

In our project, we will use artificial neural networks to perform logo classification on images scraped from social media, to find current and potential customers. To create our model, we have a training dataset obtained from Flickr 27. This dataset contains 1070 images depicting 27 different logos. Since we are interested in doing logo classification for a few, competing companies, we have focused on Nike, Adidas, and Puma by combining the remaining logos into a category named "other." We have also added more images from other datasets, such as Flickr 32, Flickr 47, and BelgaLogo. These images, in combination with images scraped from Google, constitute our dataset, where we use around 3000 images, spread across our four categories: Adidas, Nike, Puma, and other. The label for each image is stored in a separate text file, so to create our training dataset, we will use a simple for-loop to add the logo-label to each corresponding image by matching the correct label with the correct image-id.

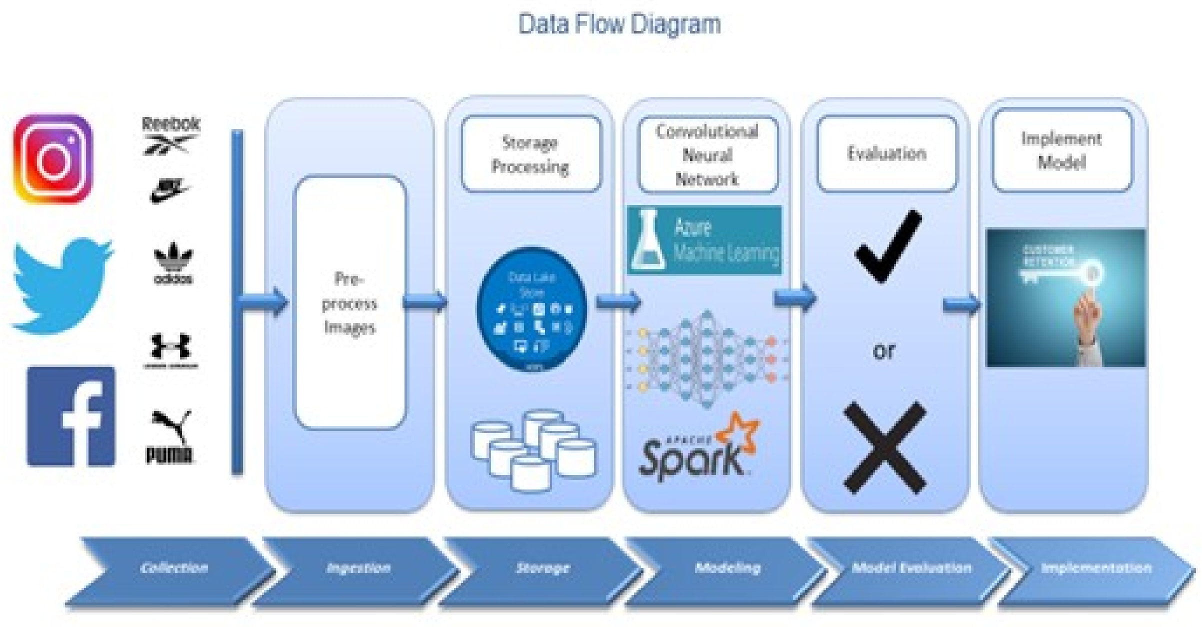
The reason we keep each label in a separate text file, instead of just using the logo name as the image-id, is that each file needs to have a unique id, which would have been problematic if the image-ids consisted of the logo name. Also, by storing the labels in a separate text file, we can with ease manipulate the labels to combine the image labels from the different datasets into the "other"-category with a simple line of code, rather than having to change each image-id manually.

Once we have created our model, we can start collecting data to employ our model for logo classification. We will collect our data from Twitter, Instagram, and Facebook through their highly used public APIs to obtain these images, as well as the users' usernames when we employ the model. To use these APIs to download images, we will have to have access to a user account for each of these social media platforms. With these accounts, we can create an app where we obtain a consumer key, consumer secret, access token, and access secret. These are the credentials we require to scrape the images of the platforms, and then store them in our database. The environment in which we will do the scraping through the APIs are a simple Jupyter Notebook, using Python.

Since our data is in the form of images, the data will be represented by an array with the dimensions of the images' height and width, where each pixel contains an array of three elements consisting of the three primary colors red, blue, and green to determine the color of that specific pixel. We will, thus, not perform traditional data cleansing, as we are not dealing with missing values, extreme outliers, or such. Instead, our focus lies on pre-processing the images to prepare the dataset for analysis. Such pre-processing entails scaling pixels, filtering the images by removing glare, brightness and replacing pixels above a certain intensity. Also, in order to maximize the accuracy of our model, we will create an image generator to manipulate each picture. Image generators are a common tool when working with image recognition in order to create a robust model, by applying elements such as flipping the image, rotating the image, moving the center of the image, and so forth. In our image generator, we will, therefore, multiply each of our images by a coefficient of 10, to create slightly different variations of our images. Therefore, even though our dataset might not be that large, it will be processed as if it was 10 times larger.

As insinuated from the above discussion, the major issue we are facing with our data is not dirty data. Instead, the issues of preprocessing the images into the right dimension and scaling will be more prevalent. It is also important to highlight the size issue with working with data. Since a single image usually contains thousands of pixels, which each represent more than one data point through its position in the picture, along with the colors, images take up more space than a simple row in a structured dataframe. Therefore, even though we are working with a conceptual storing principle, in reality we are limited to only using my laptop, which therefore restricts the number of images we can use for the project, simply due to storage space and processing capabilities.

The detailed dataflow diagram, depicting how our model will go from ingestion, to pre-processing, model creation, evaluation, and implementation, can be seen below:



Since we are assuming that this project is for a large, multinational company, we assume that we will need a data storage solution that will be scalable to the point to support large enterprises. Also, as our data is in the form of images, which is considered unstructured data, we will not use a relational database to store our data. Instead, we will be looking to a NoSQL database for our data-storage. Since we will be using Keras on top of tensorflow to build convolutional neural networks for our logo classification, we need a database that works well with a platform that supports Keras and neural networks. Thus, we decided on Azure Data Lake Store and its machine learning platform, Azure ML, as this solution provides support for what this project is trying to accomplish.

As mentioned above, we need a data storage solution that is highly scalable, which Azure Data Lake Store is, as it has no limit regarding total size, account size, or file size. Azure Data Lake Store also uses Hadoop Insight Cluster, HDInsight, to fetch data from the database. HDInsight uses parallel processing and partition to speed up the processing of the data, which is vital since a traditional data-processing system would not be able to process the expected large amounts of data that this project would require in a timely manner. Also, since HDInsight uses parallel processing and partitioning, it enables us to ensure fault tolerance, as partitions of the data is stored across the cluster, which is key to building a reliable infrastructure.

Furthermore, as mentioned, since our data is in the form of images, we needed a NoSQL-like database that can handle unstructured data. However, we will also have some structured data, since we will be scraping data in the form of text in usernames, and would thus require a rather flexible database. Azure Data Lake fits this description as the database can handle any type of data, which enables us to use it for storing our images. Due to this flexibility, Azure Data Lake store is the optimal database for our model.

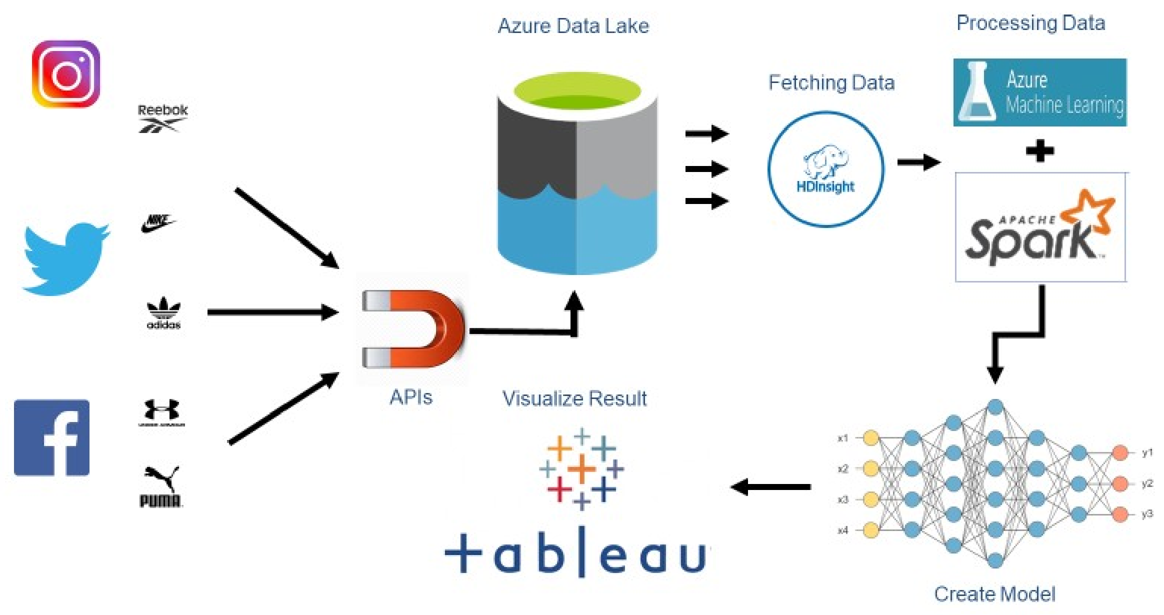
Another reason why Azure Data Lake Store fits into our solution is that is has an integrated function in Azure Machine Learning, which is the platform in which we will build our convolutional neural network. Using HDInsight, we are able to fetch the data from Azure Data Lake Store and transfer into our programming environment, Azure ML. In Azure ML, we use the libraries provided through this environment by building a convolutional neural network with Keras, on top of Tensorflow. Through the flexibility of Azure ML, we have the option to build our model in a variety of languages, such as Python, R, and Spark. To access the parallel processing of cluster computing, the optimal solution for our project would be to build the model using Spark.

Furthermore, since Azure Data Lake Store is a cloud-service, each data scientist working on this project would access the data with ease regardless of geographical location. This solution, thus, enables us to create virtual teams for this project, which would have been substantially harder if the project would have used a traditional, local database. Since localizing and fetching data from a local database can be time consuming, and thus costly, a cloud storage solution like Azure Data Lake Store is therefore optimal for our project.

Lastly, the reason why we choose Azure Data Lake Store for storing our data is due to its incredible cost efficiency and ease of setting up. Since Azure offers this storage as an IaaS (Infrastructure as a Service), our organization would not have to build this infrastructure ourselves. Instead, we will simply pay Azure a small portion of the amount that it would cost to build this infrastructure ourselves, and they will provide us this service. Also, as Azure Data Lake Store is a cloud service, we will only pay for the storage capabilities that we are using. Furthermore, since this infrastructure is already established and provided by Azure, it would only take an instance to set it up and keep it running for implementing our model. Should there be any issues with our storage unit, we will have access to support through Azure ML's customer support, which is something we should not have had if we were to build this infrastructure ourselves and keep the maintenance inhouse.

Once the model has been created and evaluated, we will use Tableau to visualize our result, in the form of image labeling and potential customer that we might want to target as our current customers. The reason why we will use Tableau is that one can use this tool to build interactive dashboard that are easy to understand and esthetically pleasing, and that it enables us to show the results of our models without having to show any code, as many business professionals with a non-technical background might find code intimidating and, thus, lose concentration on the model.

A more detailed visualization of the infrastructure of our solution is attached below.

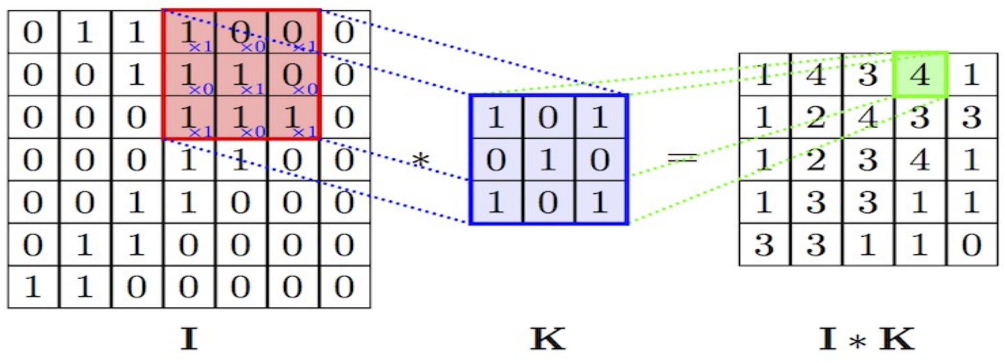


As mentioned above, we will be using neural networks to create our model for logo classification. More specifically, we will be employing a convolutional neural network to classify the images according to the logos they contain. Convolutional neural networks, also called CNNs, are a distinct kind of neural networks that are specifically efficient at image recognition and classification, even though CNNs can be used for other purposes as well. What foremost makes CNNs unique is the use of convolutional layers in these models. Each of these layers specializes on a specific task, such as identifying edges and shapes, to more advanced objects, such as eyes and ears.

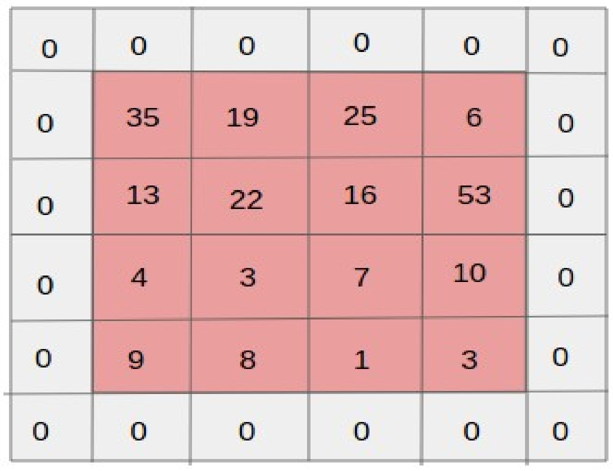
Regular Neural Networks simply process the input data through a series of one-dimensional hidden layers consisting of sets of neurons, where each layer is fully connected to all neurons in the layer before. The last layer in this architecture – called the connected layer — is also fully connected to the previous layer and it produces the predictions.

Convolutional Neural Networks, on the other hand, differ from this architecture. CNNs use layers structured into three dimensions: width, height, and depth. Also in contrast to regular neural networks, in CNNs, the neurons are structured in layers which do not connect to all neurons in the adjacent layer, but rather only to a small portion of it.

To dive deeper into the details of CNNs, CNNs create a feature map of the input image. The CNN constructs the feature map by using a filter, called a kernel, sliding over the image. At every location of the image, the dot product is calculated for the pixels within the kernel, and these dot products then constitute the feature map, which thus creates sort of an abstract replica of the original image. Below is an image that depicts this idea:



Evidently, the feature map will then be smaller than the original image if this procedure is left unregulated. Hence, we are required to pad the feature map, to make sure it remains the same size as the original image. This is done simply by adding a layer of zero-value pixels around the feature map. In addition to maintaining the original image size for the feature map, this padding increases performance and makes sure the kernel and stride size will fit in the input. An example of zero-padding of a feature map is shown below:



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

These operations of creating a feature map and then padding the feature map are called convolutions, hence the name convolutional neural network, and the convolutions occur in the convolutional layers. Between each convolutional layer, one usually adds a pooling layer. The most common technique used in pooling layers is max pooling, which simply collects the largest value in each kernel location. The pooling layers, thus, reduce the dimensions of the models by reducing the number of parameters and computation in the network. The effect of this reduction is shortened training time, and reducing the risk of overfitting.

Once the data has passed through the convolutional- and pooling layers, the model will classify the image. Prior to the classification, our data is transformed from three-dimensional to one-dimensional, since the few, fully connected layers that are responsible for the classification can only handle one-dimensional data. This dimension reduction is done through a dense-layer, and as its name indicates, it increases the density of our data by reducing its dimensions. Thus, the output of the model, which is then used as support for classifying the image, is a one-dimensional vector. A code example written in Python on how a CNN is constructed is attached below:

In [16]:

```
cnn = Sequential()
cnn.add(Conv2D(16, kernel_size=(3,3),input_shape=(64,64,3), activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(32, kernel_size=(3,3),activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(64, kernel_size=(3,3),activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(128, kernel_size=(3,3),activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Flatten())
cnn.add(Dense(64,activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(4,activation='softmax'))
cnn.compile(optimizer=sgd,loss='sparse_categorical_crossentropy',metrics=['accuracy'])
print(cnn.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 16)        448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 29, 29, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 4, 128)         73856
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 2, 2, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                32832
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 260
=================================================================
Total params: 130,532
Trainable params: 130,532
Non-trainable params: 0
_________________________________________________________________
None
```

The reason why we employ CNNs in our project is, as stated above, that CNNs are optimal to use for complicated image classification. There are, evidently other techniques that one can employ for image recognition, such as simple KNN-algorithms to neural networks, but CNNs have historically proven again and again to be the best option for image recognition. With large amounts of data, CNNs can achieve tremendous results and since our project aims to create a scalable model to be employed by a large enterprise, CNNs fit our requirement. The only downside to CNNs is that they are rather hard to grasp and it is rather hard to tune the parameters of the network, as one has to truly understand the aspect of each parameter. Nevertheless, CNNs are, therefore, optimal for our project from a learning perspective, since one has to understand the CNN-technique in order to employ such a solution.

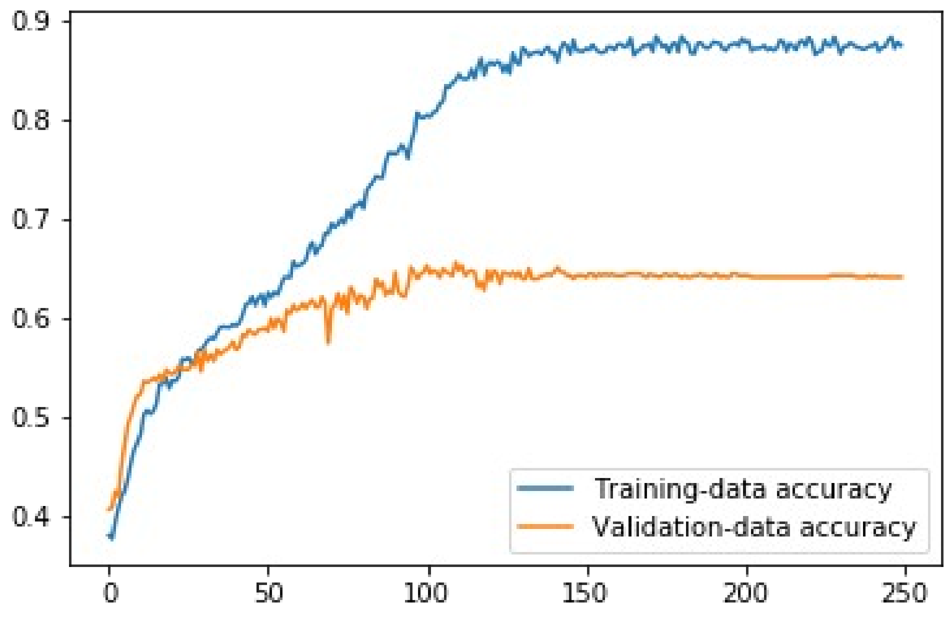
To apply an example CNN to our specific solution, we feed the model our training data in the form of our X_train-variable, which is an array of each image with its pixels, along with our Y_train-variable, which is the label of each image. The CNN-model then takes this input, and for each convolutional layer, creates an abstraction of the images. The model then learns patterns in these abstraction by recognizing the shape and edges of each image containing a logo. Through learning these patterns, our model establishes rules for each logo-category. The model then uses the given labels to test itself to see if it is learning from the images and, thus, calculates a training accuracy.

When the images have passed through the entire network, we then give it the test data in form of our X_test-variable, which is also an array of each image with its pixels. The model then applies these rules to the images in order to predict what category the image belongs to, dependent on what logo it contains. After the model has made its predictions, it compares these prediction with our Y_test-variable, which contains the true labels of each test image. Through this comparison, we get a validation accuracy.

In order to utilize the power of CNNs, I will create two models: one simple model created from scratch and a model using the concept of transfer learning. To begin with, looking at the initial, simple model, we created the model using the following code:

```
In [16]:
cnn = Sequential()
cnn.add(Conv2D(16, kernel_size=(3,3),input_shape=(64,64,3), activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(32, kernel_size=(3,3),activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(64, kernel_size=(3,3),activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Conv2D(128, kernel_size=(3,3),activation='relu'))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Flatten())
cnn.add(Dense(64,activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(4,activation='softmax'))
cnn.compile(optimizer=sgd,loss='sparse_categorical_crossentropy',metrics=['accuracy'])
print(cnn.summary())
```

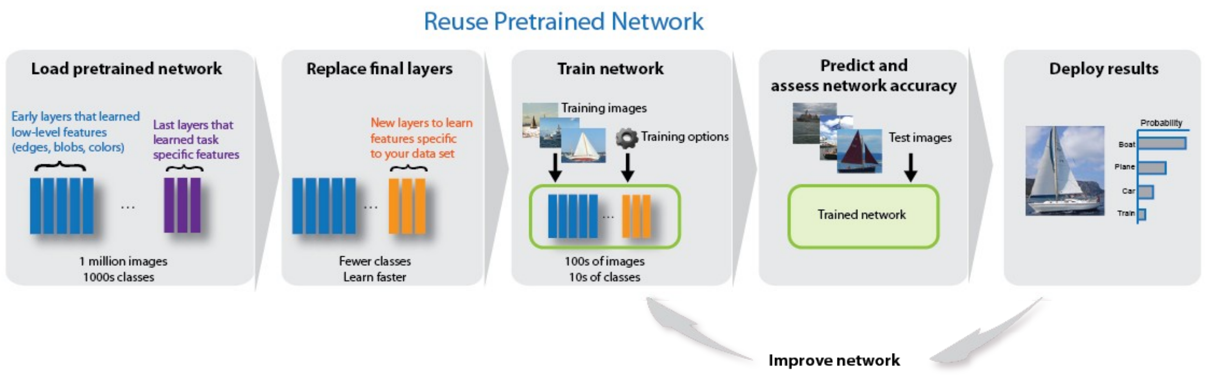
The model thus consists of four convolutional layers, four pooling layers, two dense layers, a dropout layer, and a flattening layer. Employing this model, we get the following accuracies:



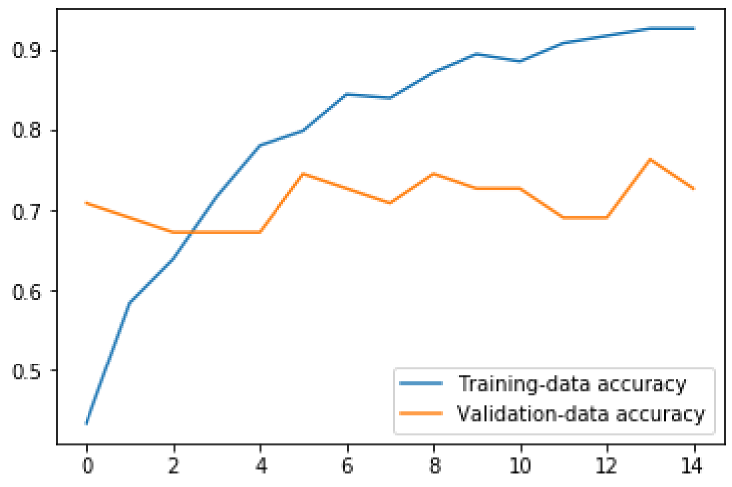
As we can see, there is a gap between the training accuracy and the validation accuracy that continues to grow the more epochs we pass through. This indicates overfitting, as the model is doing well in training but not in testing. A common way to handle overfitting is lowering the learning rate to make sure that we are doing equally as well on the validation set as on the training set, or increasing the dropout rate. However, bearing in mind that we are working with an extremely small dataset for a neural network solution, we must accept that there will be some overfitting, as there is not enough data for the network to create robust, general rules. This is especially true considering that we have adjusted the learning rate and the dropout rate, along with many other parameters, to lower the overfitting and maximize the validation accuracy. Nevertheless, considering the small dataset, the validation accuracy of over 66% that we are able to achieve should be considered acceptable, at least. The argument for viewing the model acceptable is even further strengthened when discussing the business application of our model: finding those who post images of Adidas-products. Looking at this detail, we can see that the recall for images containing Adidas images is over 83% and the precision is over 71%. The reason why we have this high recall and precision for Adidas is most likely due to Adidas being the category with the most images. Below is a confusion matrix displaying true label and predicted labels for each class:

	Adidas	Nike	Other	Puma	True Total
Adidas	203	4	24	13	244
Nike	29	21	28	7	85
Other	19	18	142	9	188
Puma	31	17	17	19	84
Predicted Total	282	60	211	48	0

Furthermore, we also created a model using transfer learning. Transfer learning is a concept of neural networks in which we use an already pre-trained network, excluding the top layers. This pre-trained network is used to find shapes and edges in our images. On top of this pre-trained model we customize new layers that are specific to what we want to predict. A visualization of how transfer learning works is attached below:



In our case, we use ResNet50 as our pre-trained network, and we use a virtual image-set named ImageNet, which consists of roughly 12 million images. On top of ResNet50 we add two convolutional layers and a dense layer to perform the predictions. Using our specific model, we get the following accuracies:



Yet again, we can see that there is some overfitting occurring. This is can be explained by the small dataset we use for this model –only 270 images. The reason why we use so few images is that ImageNet contains enough images for us to have an acceptable performance, and should we increase our image-count, we would need more powerful processing units. Nevertheless, as we can see, this model provides an accuracy of above 76%, and looking closer at each prediction, we can see that the model has a precision above 77% and a recall of 90% when it comes to prediction Adidas-logos. Below is the confusion matrix for the model's predictions:

	Adidas	**Nike**	**Other**	**Puma**	**True Total**
Adidas	27	0	3	0	30
Nike	3	0	2	1	6
Other	0	0	13	0	13
Puma	5	0	1	0	6
Predicted Total	35	0	19	1	0

What is interesting to notice is that our model is unable to classify any of the Nike-images in our validation set. This might indicate that Nike is especially hard to predict, as its products differ highly from each other.

These are the evaluation metrics for each of the two models:

	CNN built using transfer learning	CNN built from scratch
Validation accuracy	76.4%	65.6%
Precision	77.1%	72.0%
Recall	90%	83.2%

While comparing these two models, we can, thus, see that using transfer learning is more effective than building a convolutional neural network from scratch. The explanation of this occurrence is that it is uncommon to have access to a dataset large and versatile enough to build an adequate convolutional neural network. Therefore, transfer learning is usually preferable above building convolutional neural networks from scratch, and it is a technique that we will employ in this project, since we then are able to build an accurate and robust model will a very small dataset.

In conclusion, the business issue one would be able to solve with our proposed solution is the global decline in customer retention. By using our models on images scraped from social media, we would be able to identify users who post pictures of our products. Thus, we can target these customers in our advertisement as current customers in order to increase our customer retention. This would be significant for any organization, since according to Harvard Business School, increasing customer retention rates by 5% increases profits by 25%-95%. Therefore, our solution has a clear business issue that could generate substantial profit for many organizations across many industries.