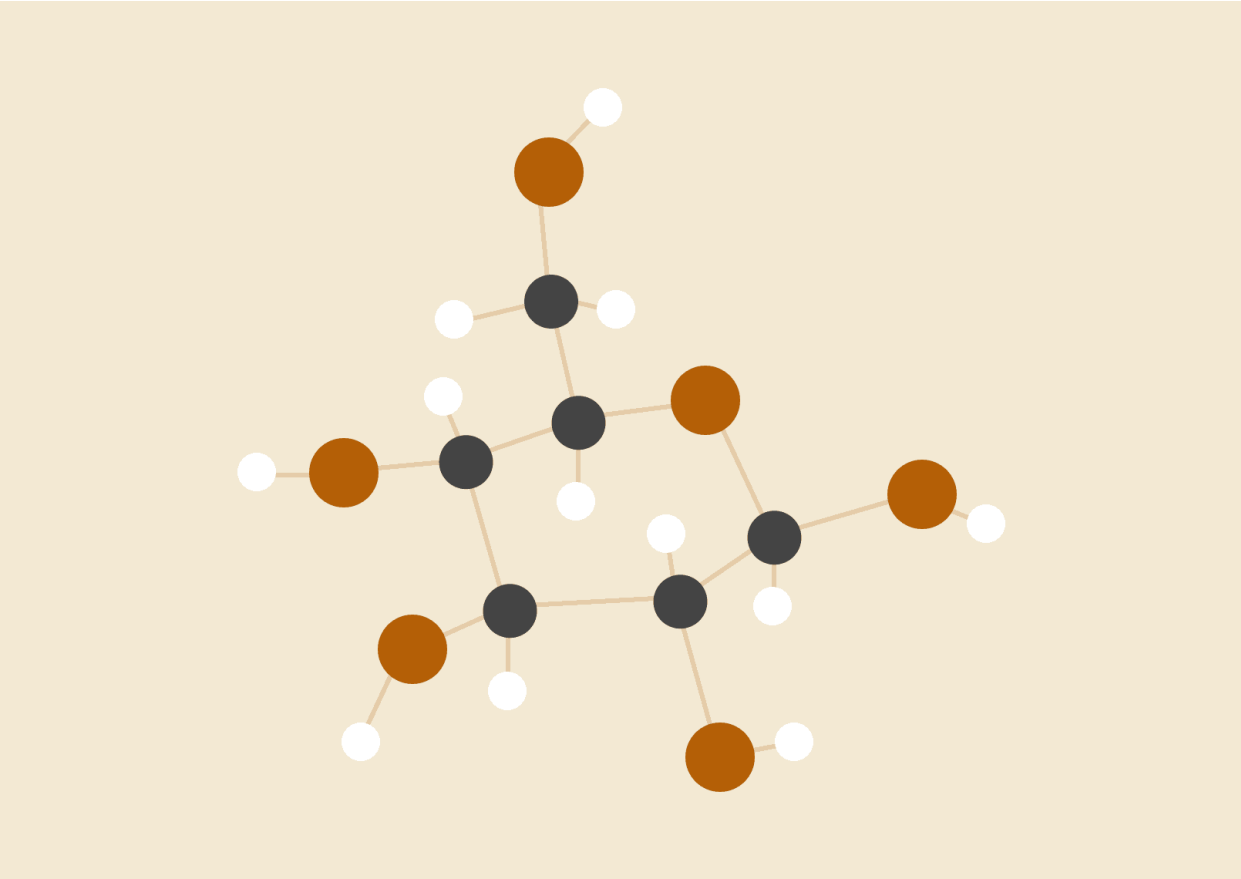
Solar Storm recognition Report



02/09/2020

INTRODUCTION

This project is a computer vision task that aims to perform sunspot group classification using deep learning algorithms, 2 types of images are used in classification: Magnetogram and continuum images, each type is split into 3 classes: Alpha, Beta and Betax.

METHODS

Since it's a computer vision task, we used **CNN (Convolutional neural network)** that is widely used in image classification, object detection and face recognition, other architectures like RNN (Recurrent neural networks) or MLP (Multi layer perceptron) can be used, but CNNs have proved to be a better architecture for computer vision tasks.

Machine learning algorithms like SVM (Support vector machine), Decision trees or KNN (k-nearest neighbors) can also be used in image classification, but CNN gives better results.

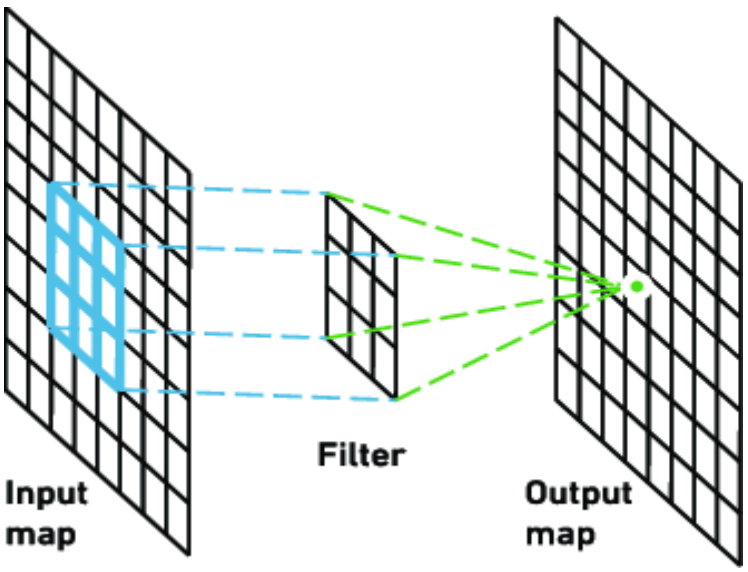
In order to improve CNN performance, we used **transfer learning** that consists of using a **pre-trained** model and modifying it in order to classify our data. In this project we used the Resnet model for feature extraction and we added our own classifier, steps will be explained below.

Convolutional neural Networks

I. **Description:** CNN is a special architecture of neural networks, it's composed of neurons (nodes) that has learnable weights and biases and that performs a dot product followed by a nonlinear function, CNN is composed of 3 type of layers:

1. **Convolutional layers:** it's the core building block of CNNs, their main goal is to extract relevant data from images (edges, colors, lines… etc) by using a group of filters (kernels), each kernel is responsible for detecting a certain characteristic that is going to be used in order to classify the image.

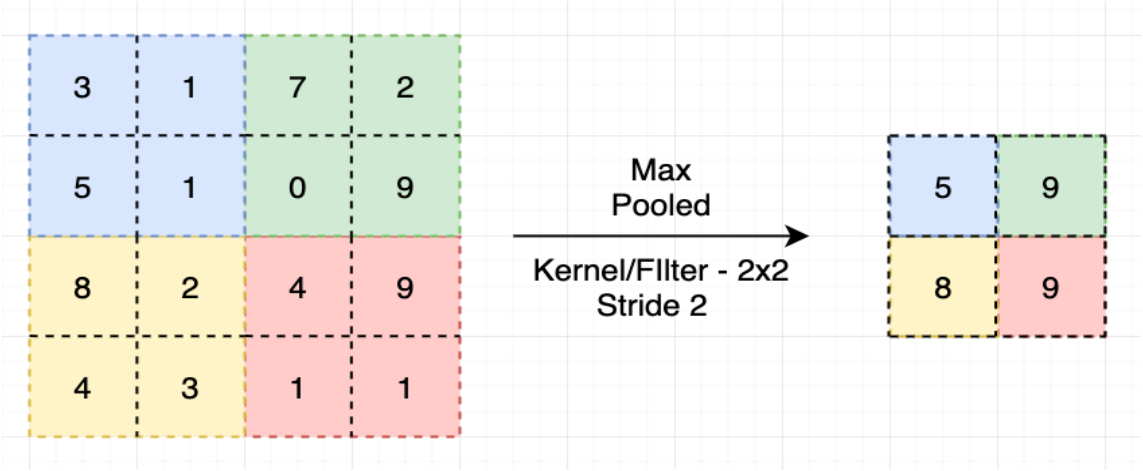
   A kernel represents a matrix of trainable values (parameters) of 3x3 or 5x5 size, each kernel will be passed through the image starting from top left (0,0) and will perform a dot product with the covered region, then it makes another step to the next region, this step is called **Stride** (note that stride can be greater than 1), this process will continue until we cover the whole image, and we will get another image that contains a specific characteristic (Can be lines, colors, shapes ...etc) as output.



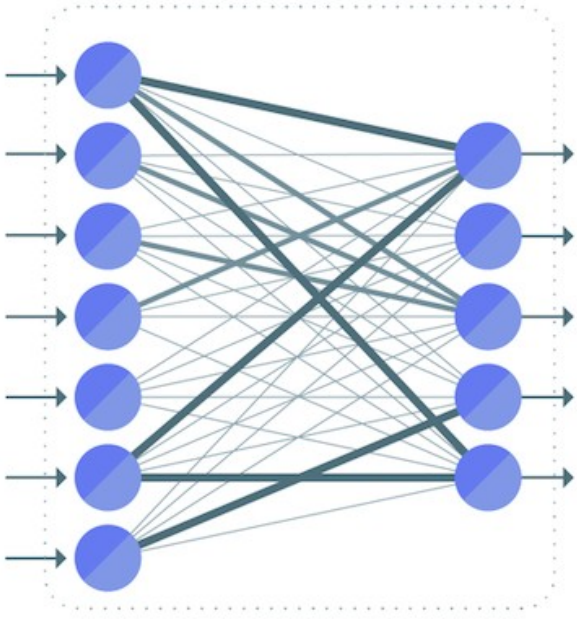
2. **Pooling layer:** they are used in order to reduce the spatial size of convolutional layers output, as we mentioned before, the output of our convolutional layer is an image, and in most cases we have a lot of kernels, therefore, a lot of output images (since each kernel is going to output an image).

   The number of images is a problem since it consumes more resources and needs a lot of computation power in order to process them, that's where Pooling layers interfere, they reduce the size of output images (maps) and keep the relevant data only, therefore, they reduce computation and help prevent overfitting.

   Pooling layers perform pooling operations, it means that a box with certain size (2x2 generally) is passed through the whole map (kernel's output images) and in each region, it performs an operation that depends on the type of pooling, the most common types of pooling are **Max pooling** and **Average pooling**, if we use Max pooling we take the maximum value of the covered region, but if we use Average pooling we take the average instead.



3. **Fully connected layers:** these layers are responsible for classification, their main task is to take the images created by kernels in previous layers, flatten them and pass them through fully connected nodes, these nodes are also trainable nodes that performs a dot product followed by a nonlinear function, at the end we will have a number of nodes corresponding to the number of classes that we have, let's say for example that we have 3 classes, the number of nodes in the final layer will be 3, these nodes will output the probability of each class.



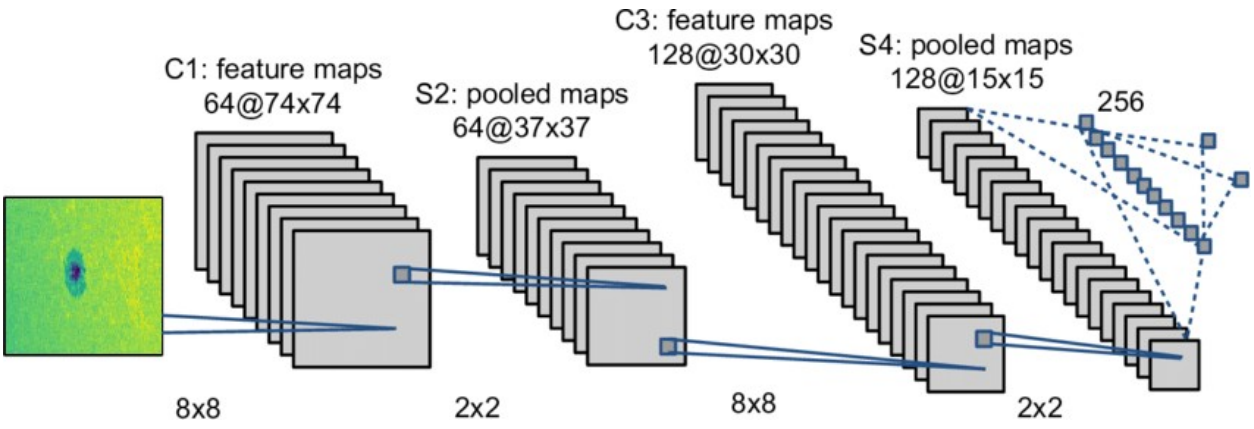
II. **CNN working:** To sum up the last section, CNN is composed of Convolutional layers that have multiple Kernels (Filters), each kernel is a matrix responsible for detecting a specific characteristic in the image, the values of kernel's matrix are trainable, it means that each time we train the network, kernel's values are updated in order to make it better in characteristics detection, these kernels will output maps (images) that contains the specific characteristic only

(horizontal edges, vertical edges, colors, shapes… etc), so for example, if we have 16 kernel (16 filter) we will get 16 images that contains a certain characteristic, these images are all **stacked** and passed to the next layer.

After each convolutional layer, there is a pooling layer that reduces the size of stacked output maps (output images) in order to reduce computation, the last pooling layer will pass the stacked images (after dimensionality reduction) to the fully connected layers.

Finally, there's fully connected layers, they start by flattening the images so they can fit their nodes, it means that they convert images (presented as matrices) to vectors, these vectors will be passed through each layer of fully connected nodes and will output the final predictions of each class.

The whole process is called **forward propagation**.



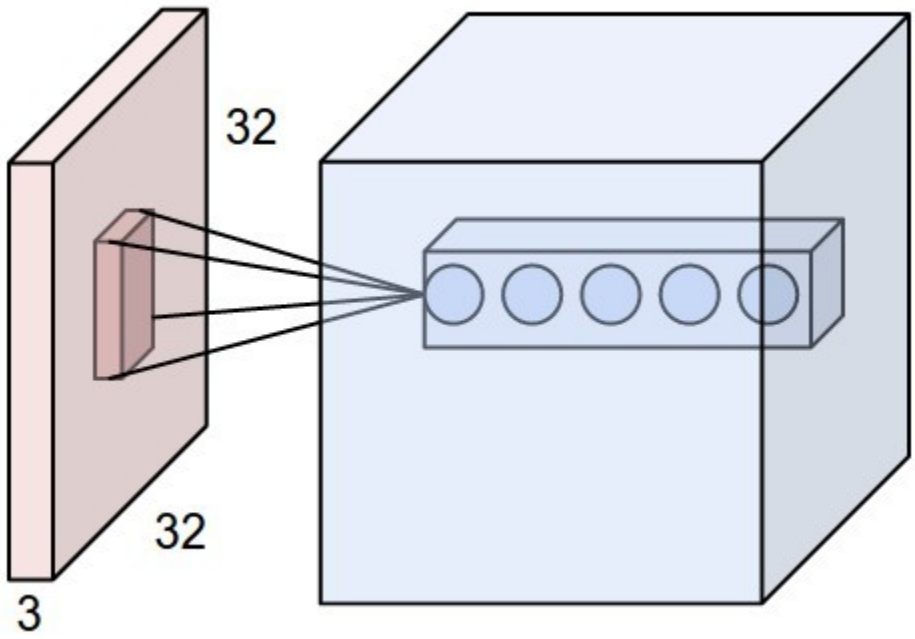
III. **Backpropagation:** Forward propagation isn't enough to train our CNN since it only gives prediction results of a certain image, so in order to train our model we need to use forwarward propagation output to find better parameters and get a better result, and this is the job of backpropagation, it works as follows:

1. **Calculating cost function:** cost function (sometimes called loss function when applied on 1 element of dataset) is a mathematical function used in order to measure the difference between the ground truth value (real value) and predicted value (result of forward propagation), the closer it's to 0 the better the result is (predicted value is close to ground truth)
2. **Back propagating :** Using the result of cost function, the model will update the weights and biases of each node and kernel, the aim here is to find weights and biases values that reduces the cost function
3. **Repeating the process:** once we finish back propagating, we repeat a forward propagation followed by backpropagation until our cost function converges, i.e. until it gets closer to 0, in other words, we try to make our prediction closer to ground truth values.

So to sum up, training in CNN is in fact performing a forward propagation using our dataset, followed by backpropagation that will update weights, biases, and kernel values (considered as weights also) and repeating these steps until our cost function converges.

IV. **CNN advantage:** the main advantage of using CNN is the reduced number of weights compared to MLP (Multi layer perceptron), this reduction is mainly due to node's **local connectivity**, in contrary to simple Neural network, each node is connected to a certain number of nodes or a certain region (in MLP each node is connected to all other nodes) this means that in CNN we have less weights than MLP, therefore, less time and computation power are needed.

Each node covers a region called **Receptive field**, the size of the receptive field is determined by the size of the kernel, so for example, if we have a kernel size of 3x3, each node will cover a region of 3x3.



Project details

I. **Technology used:**
   - Python
   - Pytorch
   - scikit-learn

II. **Neural network description:** In this project, transfer learning is used in order to improve model's accuracy, we used Resnet-152 model for feature extraction, followed by 2 fully connected layers with 512 and 128 nodes respectively, and an output layer of 3 nodes, Resnet is a good choice when we need to train deep neural networks, since it solves vanishing gradient problem and gives good results.

III. **Project packages and files description:** the project contains 3 main packages as follows:

   - **network:** Neural network architecture is defined as a class in this package, it contains all the methods necessary for training and predicting.
   - **data:** Contains Data class that is responsible for Loading train, validation and test sets, performing data augmentation on the data set (Normalization, resizing, horizontal flip, cropping), and then creating Batches of images and storing them in data loaders.
   - **CustomImageLoader:** This class is a subclass of pytorch ImageFolder class, the main difference is that CustomImageLoader returns the path to each image (in addition to the image itself and its label), this helps in visualization only, and will have no effect on training or testing result.

In addition to these packages, there's 2 other folders named dataset (data set should be inside this folder) and models (training models will be saved on this folder), and another python file 'classifier.py' that contains our main function.

IV. **Detailed description:**

- Neural network: our neural network is defined inside 'network' package, precisely inside 'Network' class that is a subclass of nn.Module which is the base class of all neural networks in pytorch, once an instance of Network class is created, the program will load resnet-152 pretrained model with its weights, it means that all kernels (filters) in resnet-152 are trained, and they can be used in order to extract features (edges, shapes, colors… etc) from our images, but since resnet-152 is trained with another type of images, the fully connected layers can't be used for solar storm recognition, so we changed these layers with custom fully connected layers that has the following characteristics:
  - Linear layer with 512 nodes followed by ReLU activation function and a dropout with a probability of 0.2 in order to reduce overfitting
  - Linear layer with 128 nodes followed by ReLU activation function and a dropout with a probability of 0.2
  - Output layer with 3 nodes that represents our classes followed by LogSoftmax activation function
  - Negative log likelihood cost function (NLLLoss)
  - Adam optimizer

  Before replacing resnet fully connected layers with our custom layers, we need to make sure that we disable Autograd which is responsible for creating a directed acyclic graph that records all operations used for predicting our output, this graph is used by the optimizer (We used Adam optimizer on this project) in order to update weights in a way that gives a better result and help cost function converge, the main reason for disabling autograd is to avoid training the whole CNN since convolutional layers are already trained on a huge amount of data, instead, we train our classifier (fully connected layers) only, this method will guarantee less time and computation power, and more accuracy.

- Data: we used 3 partitions of datasets, train set, validation set and test set, train set is used for training our model, and test set for testing it on new images, as to validation set, its main task is to prevent overfitting, the approach that we used is as follows:

  1) Initialize a variable that contains the minimum validation cost (it should be initialized to $+\infty$)
  2) After each training epoch, we predict the output for each image in validation set (without backpropagation)

3) We calculate the cost function of our validation set using the calculated output, if our new validation cost is less than the minimum validation cost, we save our model and update the minimum validation cost.
4) If the minimum validation cost doesn't decrease after a certain number of epochs, but the training cost keeps on converging, we can say that our model is overfitting, and we stop training.

In addition to preventing overfitting, this method can be useful in case if we stop training for any reason, we can use the saved models as checkpoints to continue our training instead of starting all over again.

- Data Augmentation: since we have a small number of data, augmentation can be a good choice in order to increase the diversity of our data, which will reduce overfitting, we used the following transformations on Training and validation sets:
  - Random resizing and cropping
  - Random rotations of 30 degrees
  - Random horizontal flipping and normalization.
- F1 score: In order to get an F1 score for each class, we used scikit learn metrics package that provides easy tools for such tasks.

V. **Experiment and results:** We tested 2 architectures in order to train our model, we started with a small neural network that has 1 fully connected layer of 256 node and an output layer of 3 nodes, and then we tested with the architecture that we mentioned before with 512, 128, 3 respectively, we used in both architecture 100 training epoch, batch size of 32, a learning rate of 0.0003, a dropout with 0.2 probability, 20% of training data was used for validation set and 80% for training.

We used a GTX 1080 GPU to accelerate training, note that GPU acceleration will work only on GPUs that support cuda, if the program doesn't detect any such GPU, the training will be on CPU which will take much more time.

Two models were created for Magnetogram and continuum images, so they were trained separately.

We noticed the following results after training on training set, and testing on test set:

Continuum	Small Architecture (256 node)	Larger Architecture (512,128 nodes)
Training Loss	0.6075	0.51
Testing Accuracy	65.4%	72.7%
F1 score for each class	[0.70548332 0.66794626 0.56078707]	[0.8081714 , 0.73421819, 0.57610475]
Macro F1 score	0.644	0.706

- **256 Architecture screenshot:**

```
Model's Accuracy is : 65.40678405761719
Calculating F1 score
Classes F1 score is respectively: [0.70548332 0.66794626 0.56078707]
Average F1 score (macro) is 0.6447388835602799
```

- **512 Architecture screenshot:**

```
Model's Accuracy is : 72.74099731445312
Calculating F1 score
Classes F1 score is respectively: [0.8081714  0.73421819 0.57610475]
Average F1 score (macro) is 0.7061647800024407
```

Since the larger architecture proved to be better for this task, we used it directly with magnetogram image, and results were as follows:

Magnetogram	Larger Architecture (512,128 nodes)
Training Loss	0.327
Testing Accuracy	75,8%
F1 score for each class	[0.82614874, 0.77874456, 0.54398382]
Macro F1 score	0.716

```
Model's Accuracy is : 75.81521606445312
Calculating F1 score
Classes F1 score is respectively: [0.82614874 0.77874456 0.54398382]
Average F1 score (macro) is 0.7162923761842971
```

VI. **Discussion:** Based on Macro F1 score, Class F1 score and training loss, we can say that a larger architecture gives a lower loss, and a higher Test accuracy for our task. We must mention that F1 score doesn't always determine the best model since it gives equal weight to recall and precision, for example, let's say that we want to classify whether a person is sick or not, if we chose a model based on recall, we may classify a sick person as healthy, which can cause a great danger, but if we chose a model based on precision, we can classify a healthy person as sick, which is less dangerous, that's why it's not always good to choose our model based on F1 score.

We can also notice that the F1 score of the Betax class is lower than Alpha and Beta classes, and that can be due to the low number of images in that class compared to other classes (imbalance), or the quality of the images.

In order to improve our model's F1 score, the following measures can be taken:

- Using a Deeper neural network, this can be achieved by adding more layers to the model, but this will take more time and power to train. We must mention that sometimes it's better to decrease the number of layers, but in our case, increasing the number of layers gave better results.
- Using another pretrained model like ResNext.
- Using Class weights in order to balance data (note that we used class weights in our project, but it didn't improve the accuracy a lot).
- Using smaller training batches since it has a regularization effect.
- Finding a better Dataset, that has a better image quality, and more balance.

Useful links

1. https://cs231n.github.io/convolutional-networks/#conv
2. https://pytorch.org/docs/stable/index.html
3. https://arxiv.org/abs/1412.6980
4. https://en.wikipedia.org/wiki/Activation_function
5. https://en.wikipedia.org/wiki/Vanishing_gradient_problem