Final project
Introduction to Machine Learning
CSC 59929
Professor: Dr. Erik Grimmelmann
Student: Marina Orzechowski

Pooling Methods in Convolutional Neural Networks on Quick, Draw! Dataset.

Introduction

In this project I will be implementing different pooling methods with different strides using the Quick, Draw! dataset and comparing the loss, accuracy, and time taken to train the models.

The motivation for this project was the fact that there are different opinions about the use of overlapping pooling windows and different strides.

For example, Geoffrey Hinton said "If the pools do not overlap, pooling loses valuable information about where things are. We need this information to detect precise relationships between the parts of an object." Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, who worked together on the research paper "ImageNet Classification with Deep Convolutional Neural Networks" use overlapping pooling in their model and mention "We generally observe during training that models with overlapping pooling find it slightly more difficult to overfit".

At the same time, Dominik Scherer in his work "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition" comes to the conclusion that "using smoother, overlapping pooling windows does not improve recognition rates" [1].

I decided to test both methods on the Quick, Draw! dataset and see if overlapping pooling improves model's performance on this data.

Pooling layers and their importance, pooling methods.

Convolutional layer applies different filters and results in a stack of feature maps where we have one feature map for each filter. A complicated dataset with many different object categories will require many filters, each responsible for finding a pattern in the image. More filters mean a bigger stack, which means that the dimensionality of our convolutional layers can get very large. Higher dimensionality means, we will need to use more parameters which can lead to over-fitting. We need a method for reducing this dimensionality without losing much information. That is why we need a pooling layer.

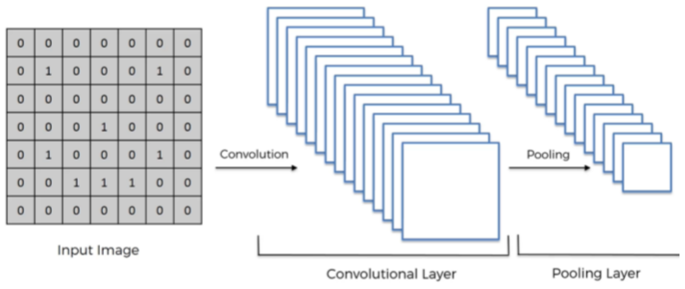


Figure 1. Pooling Layer Takes a Stack of Feature Maps as Input and Reduces Dimensionality.

A pooling layer will take a stack of feature maps as input. Then, it will implement a function according to the pooling method to each feature map independently. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs [4].

Max pooling

$$Y_{\max}[n_1,n_2]=\sum_{k_1=0}^{F-1}\sum_{k_2=0}^{F-1}\max\left(X[n_1S+k_1,n_2S+k_2]\right)$$

Average pooling

$$Y_{avg}[n_1,n_2]=\frac{1}{F^2}\sum_{k_1=0}^{F-1}\sum_{k_2=0}^{F-1}X[n_1S+k_1,n_2S+k_2]$$

Sum pooling

$$Y_{sum}[n_1,n_2]=\sum_{k_1=0}^{F-1}\sum_{k_2=0}^{F-1}X[n_1S+k_1,n_2S+k_2]$$

L2 pooling

$$Y_{L2}[n_1,n_2]=\sqrt{\sum_{k_1=0}^{F-1}\sum_{k_2=0}^{F-1}X^2[n_1S+k_1,n_2S+k_2]}$$

Figure 2. Formulas Explaining Different Pooling Methods.

There are different pooling methods (figure 2), but the most common are *max pooling* (reports the maximum output within a rectangular neighborhood), *mean pooling* (the average output within a rectangular neighborhood), *sum pooling*, L^2 *pooling* (the square root of the sum of the squares of the activations within a rectangular neighborhood). There is no universal best pooling method or straightforward guidelines on which method to use, as the result depends on data we are trying to analyze and a goal we are trying to achieve.

Many sources say that max pooling is the most commonly applied pooling operation because it focuses on the most responsive features in the image. Therefore, in theory, max pooling is the best candidate for image recognition and classification purposes [5].

However, we can see average (mean) pooling method used in applications for which smoothing an image is preferable. Average pooling has the effect to smooth images and the difference between original images and smoothing images can embody some outstanding information [6].

Figure 3 shows that there are many successful CNN architectures which use max or mean pooling, and some of them use a combination of both methods.

Models which use max pooling	**Models which use mean pooling**
AlexNet (2012) VGG-16 (2014) Inception models ResNet-50 (2015)	LeNet-5 (1998) Inception models ResNet-50 (2015) (global average pooling)

Figure 3. Usage of Max and Mean Pooling by some Successful CNN Architectures.

We can control behavior of a pooling layer by specifying the *window size* and a *stride*. The stride is just an amount by which the window slides over the feature map. In the example below we chose a window of size 2x2 and a stride 2x2.

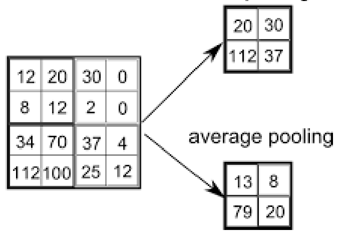


Figure 4. Pooling with window size 2x2 and a stride 2x2 on 4x4 image.

If the stride is equal to the pooling size, then the pooling is non-overlapping. If the stride is less than the pooling size, then the polling is overlapping.

Just as we predicted the size of the feature map after applying a filter to an image, we can predict the dimensions after applying pooling methods using the following formula.

$$o_i = \left\lfloor \frac{n_i + 2p_i - f_i}{s_i} \right\rfloor + 1$$

for i = 1, 2, where n_i is the side of the feature map, p_i is padding in the ith direction, f_i is a size of the pooling window, s_i is a stride in the ith direction.

For our example in figure 4 the formula for both i = 1 and i = 2 will be as follows:

$$o_i = \left\lfloor \frac{4-2}{2} \right\rfloor + 1 = 2$$

Besides reducing dimensionality and helping to reduce overfitting pooling layer *makes the model approximately invariant to small translations of the input*. On the figure 5 we can see an image of a panda and same image rotated counterclockwise. We also can see the matrix image representation and a rotated version of it. When we implement pooling, we get the same result for the original and a rotated matrix.

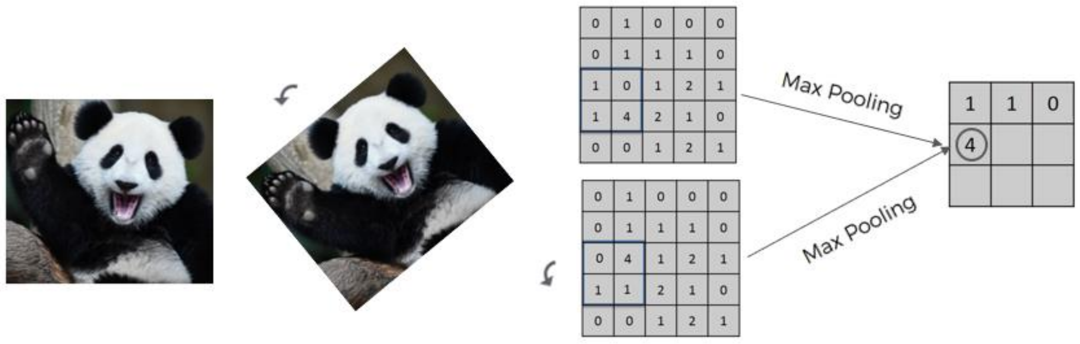


Figure 5. Pooling Makes the Model Approximately Invariant to Small Translations of the Input

Dataset description

The data was taken from the Quick, Draw! dataset [2]. It is a collection of 50 million drawings across 345 categories, contributed by players of the game Quick, Draw!. The drawings were captured as timestamped vectors, tagged with metadata including what the player was asked to draw and in which country the player was located.

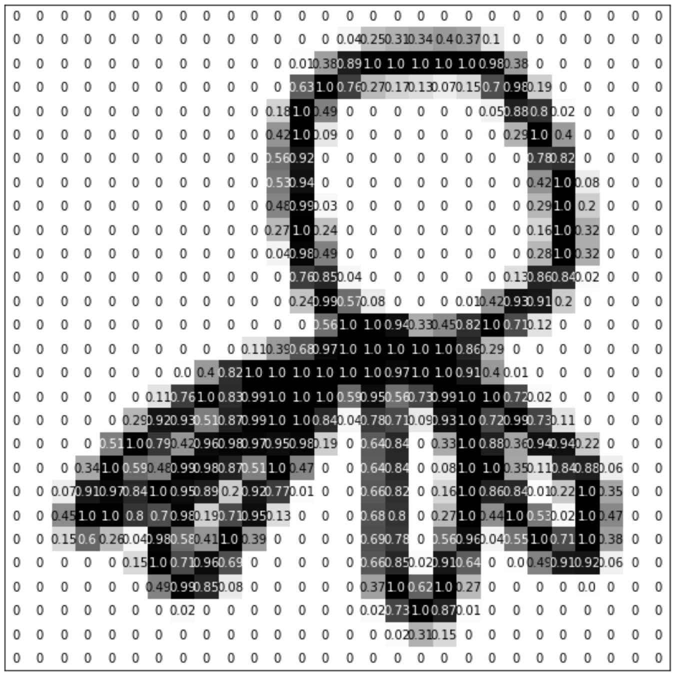


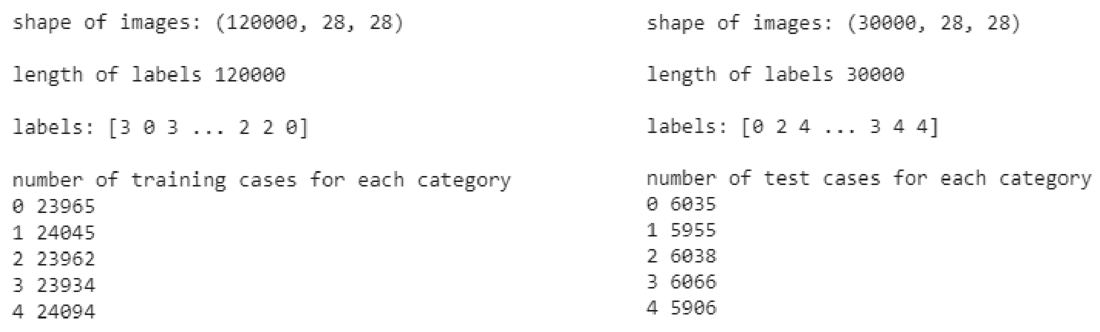
Figure 6. Image of an octopus with rescaled pixel values (reversed gray scale).

For my project I used a simplified preprocessed dataset in .npy format. Each file includes >100000 images of a specific category. The total dataset size is 37 Gb, which is very difficult to work with. Therefore, I reduced the number of images for each category to 30000. And the total dataset size to 7.7 Gb. However, Google Colab RAM of 25.5 Gb wasn't enough to process all 345 categories. So, my initial plan was to consider one hundred categories. It worked when I trained the model, and I achieved 89% accuracy. However, when I embedded the model in the web application, I realized that the accuracy is low, and the model makes too many errors.

All the computations were made for 5 categories `['car', 'ice cream', 'octopus', 'sheep', 'umbrella']`. But in the end, I was able to make the app work with 10 categories.

Building different models and comparing results.

Before building the models, I loaded previously normalized and pickled data and divided it into training (80%) and testing data (20%).

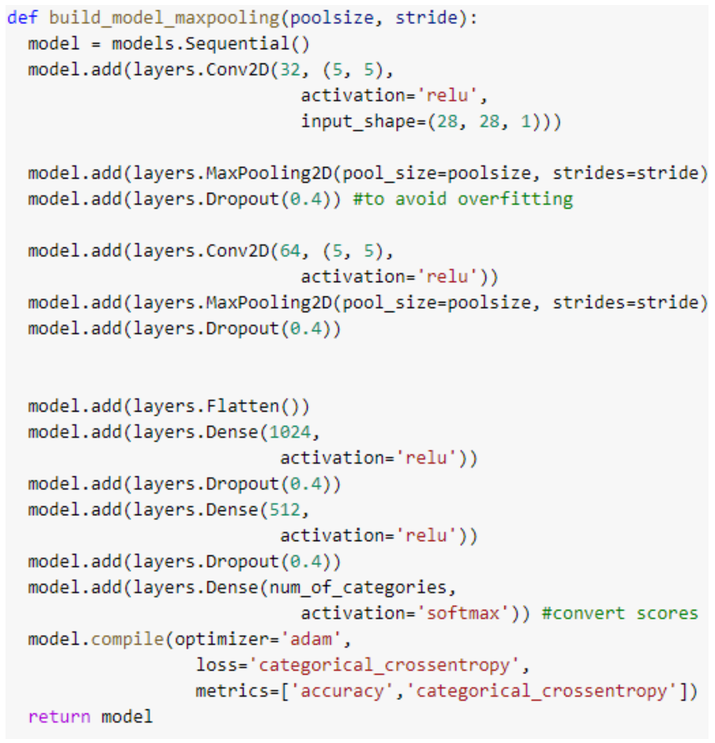


```
shape of images: (120000, 28, 28)

length of labels 120000

labels: [3 0 3 ... 2 2 0]

number of training cases for each category
0 23965
1 24045
2 23962
3 23934
4 24094
```

```
shape of images: (30000, 28, 28)

length of labels 30000

labels: [0 2 4 ... 3 4 4]

number of test cases for each category
0 6035
1 5955
2 6038
3 6066
4 5906
```

Figure 7. Examine the training (left) and testing (right) data

Then the data was reshaped, and the labels were converted from categorical to one-hot-encoded. Because I was planning to build around 18 models with different pooling methods, window sizes, and strides, I implemented two functions (for max and average pooling) which take pooling size and stride size as input and return the model.



```
def build_model_maxpooling(poolsize, stride):
  model = models.Sequential()
  model.add(layers.Conv2D(32, (5, 5),
                          activation='relu',
                          input_shape=(28, 28, 1)))

  model.add(layers.MaxPooling2D(pool_size=poolsize, strides=stride)
  model.add(layers.Dropout(0.4)) #to avoid overfitting

  model.add(layers.Conv2D(64, (5, 5),
                          activation='relu'))
  model.add(layers.MaxPooling2D(pool_size=poolsize, strides=stride)
  model.add(layers.Dropout(0.4))


  model.add(layers.Flatten())
  model.add(layers.Dense(1024,
                         activation='relu'))
  model.add(layers.Dropout(0.4))
  model.add(layers.Dense(512,
                         activation='relu'))
  model.add(layers.Dropout(0.4))
  model.add(layers.Dense(num_of_categories,
                          activation='softmax')) #convert scores
  model.compile(optimizer='adam',
                loss='categorical_crossentropy',
                metrics=['accuracy','categorical_crossentropy'])
  return model
```

Figure 8. Function to Build Max Pooling model

I used Adam optimizer because many sources claim that it is accurate and fast. I used 0.4 dropout to avoid overfitting (I also tried 0.2 and 0.5).

First, I built models using max pooling method with different pooling sizes and strides. Then I built same models but using average pooling. I trained them with batch size of 128 for 20 epochs. I called each model as “model_xx_yy”, where *xx* is a size of window, and *yy* is a stride.

Figure 9 shows that when the stride is small the model gets overfitted around the 5th epoch, and for non-overlapping models it gets overfitted after the 15th epoch (or much later).

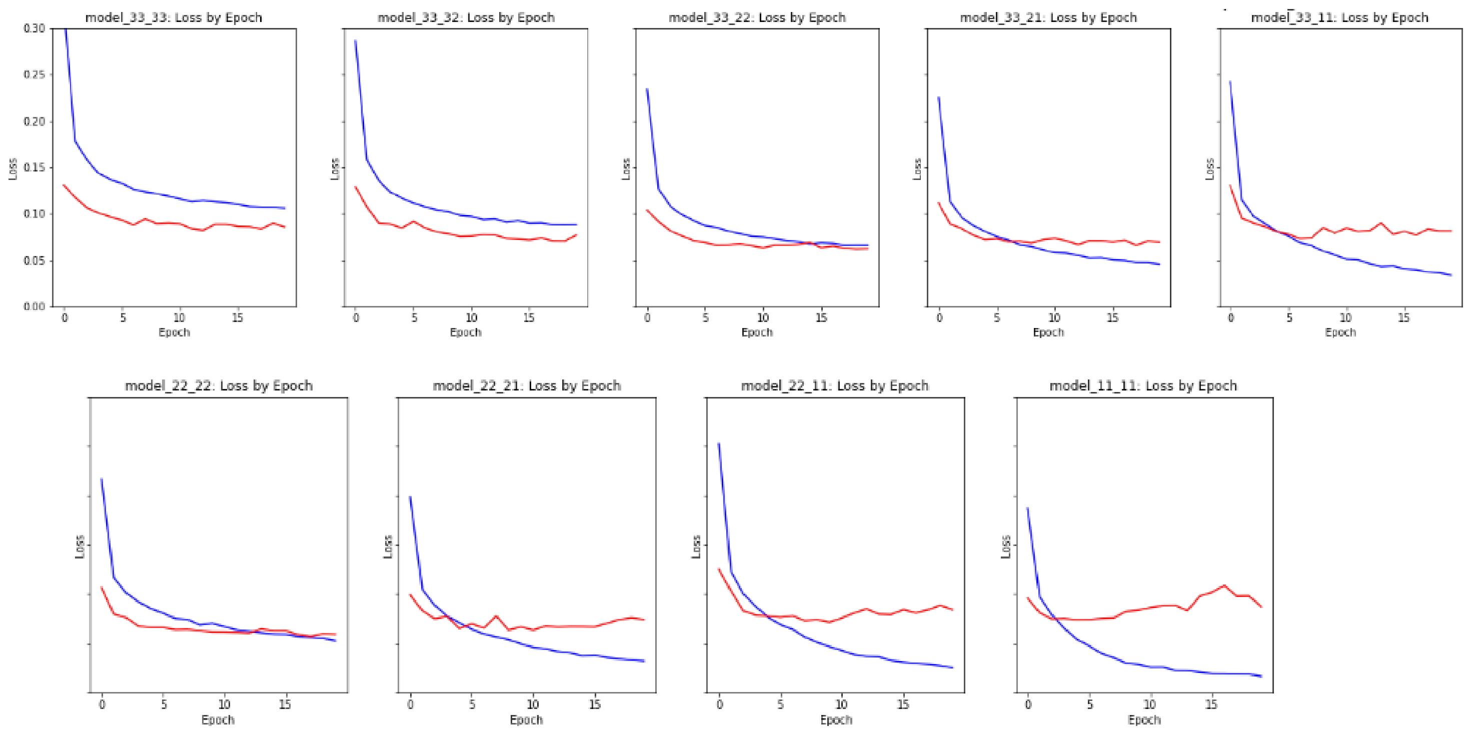
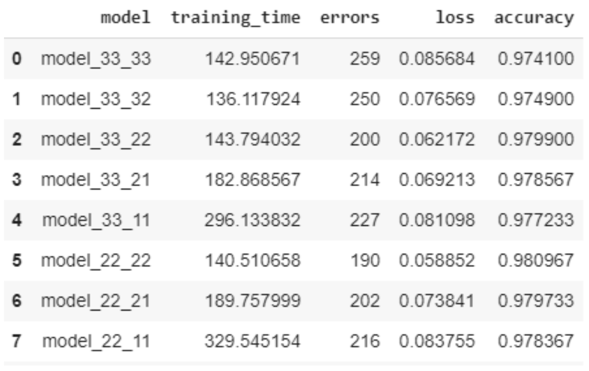


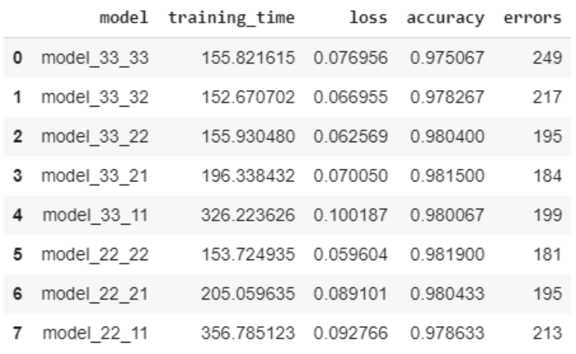
Figure 9. Maxpooling: loss

Figure 10 shows that model_33_33 gives the largest validation loss. However from figure 9 we can see that this model was undertrained. Models with stride 1x1 also resulted in a larger validation loss (they were overfitted). Non overlapping models gave less loss, but because it takes more epochs for them to get overfitted. Later, I will try to reduce overfitting by running some models for less epochs.

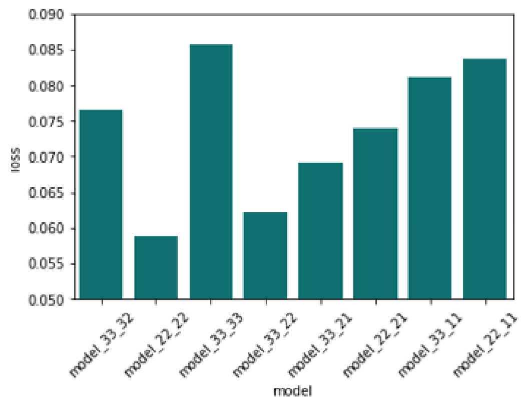
Figure 10 shows the summary of the results for the methods. We can see that results for average and max pooling are similar. However, average pooling takes a little longer than max pooling. And the accuracy for average pooling is very slightly higher.

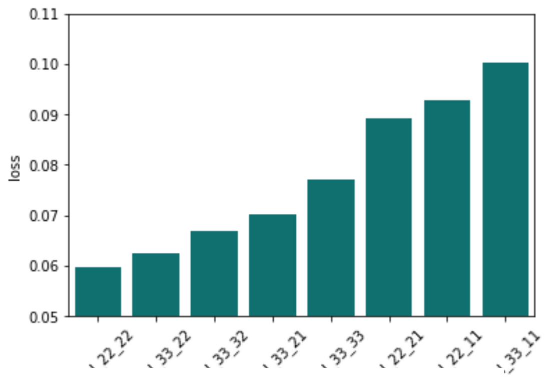


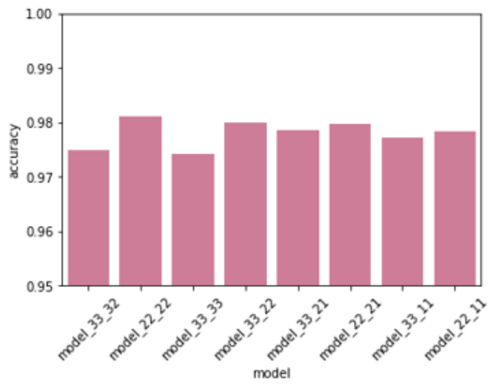
	model	training_time	errors	loss	accuracy
0	model_33_33	142.950671	259	0.085684	0.974100
1	model_33_32	136.117924	250	0.076569	0.974900
2	model_33_22	143.794032	200	0.062172	0.979900
3	model_33_21	182.868567	214	0.069213	0.978567
4	model_33_11	296.133832	227	0.081098	0.977233
5	model_22_22	140.510658	190	0.058852	0.980967
6	model_22_21	189.757999	202	0.073841	0.979733
7	model_22_11	329.545154	216	0.083755	0.978367

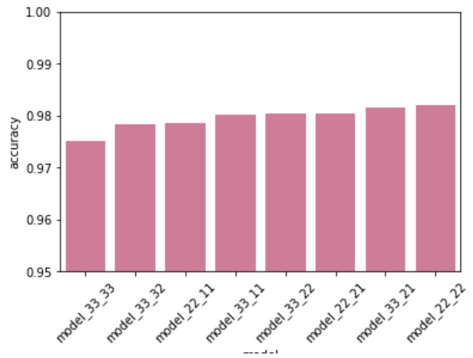


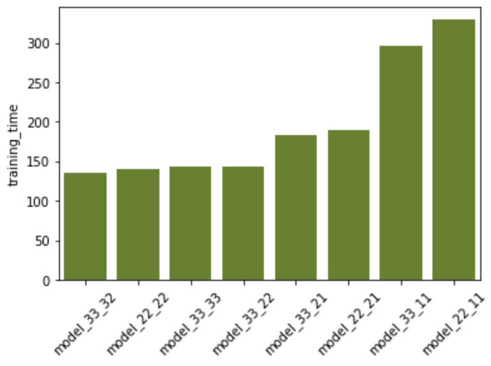
	model	training_time	loss	accuracy	errors
0	model_33_33	155.821615	0.076956	0.975067	249
1	model_33_32	152.670702	0.066955	0.978267	217
2	model_33_22	155.930480	0.062569	0.980400	195
3	model_33_21	196.338432	0.070050	0.981500	184
4	model_33_11	326.223626	0.100187	0.980067	199
5	model_22_22	153.724935	0.059604	0.981900	181
6	model_22_21	205.059635	0.089101	0.980433	195
7	model_22_11	356.785123	0.092766	0.978633	213











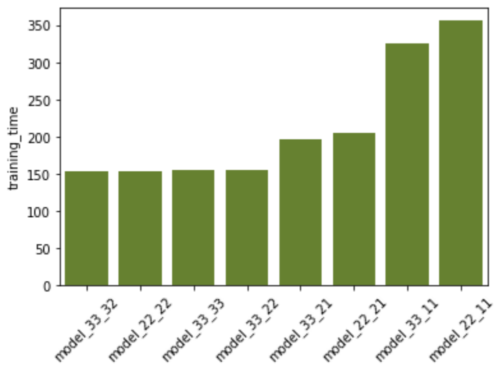


Figure10. Max pooling (left) and average pooling (right): loss, accuracy, training time

Figure 11 shows the pairwise comparison and correlation between pairs of features.

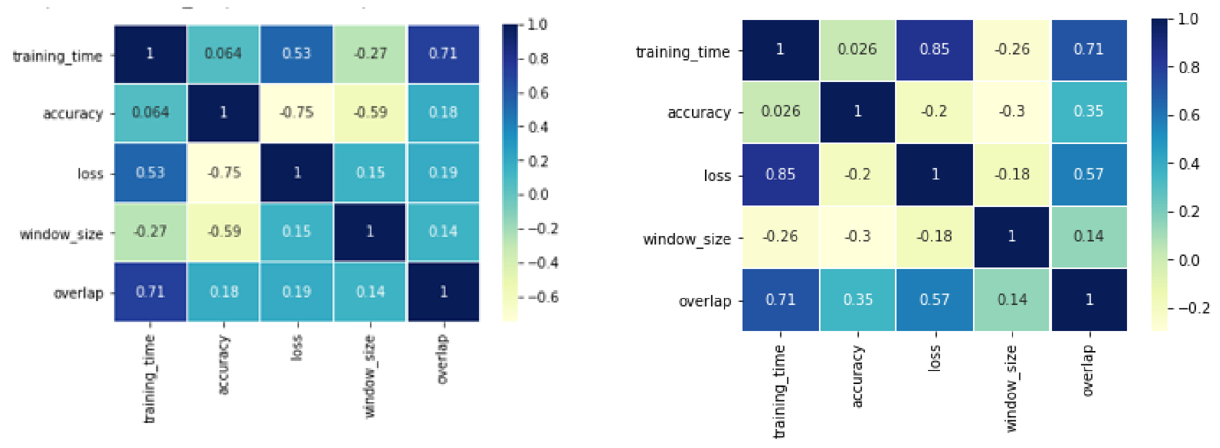
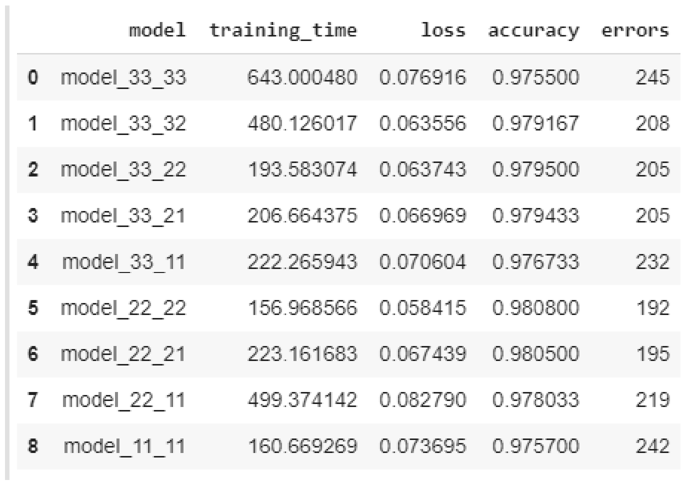
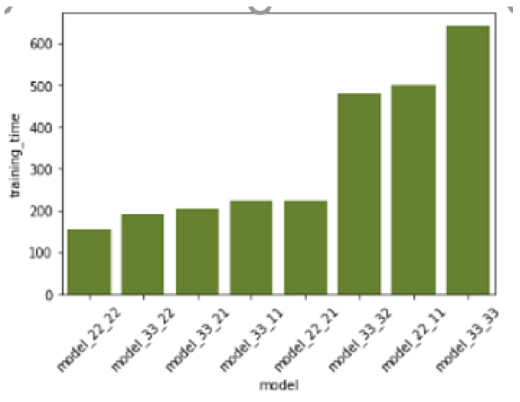


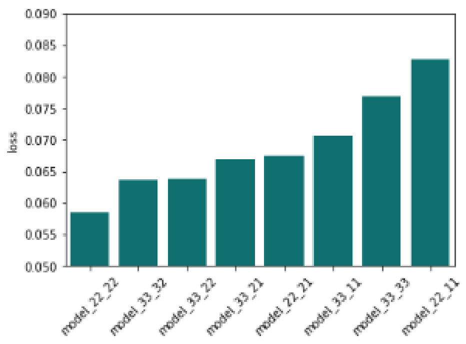
Figure 11. Correlation between pairs of features (left – max pooling, right – mean pooling).

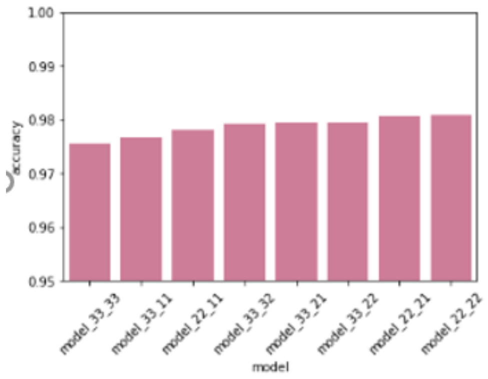
We see a strongest positive correlation between the overlapping and training time which is intuitive as we trained all models for the same number of epochs. There is a moderate positive correlation between training time and loss which can be explained by overfitting. So, overlapping didn't really influence anything except for training time. However, this experiment wasn't great because I let models with overlapping to overfit. Next, I repeated the experiment, but didn't let models to overfit, so I trained each of them for different number of epochs:



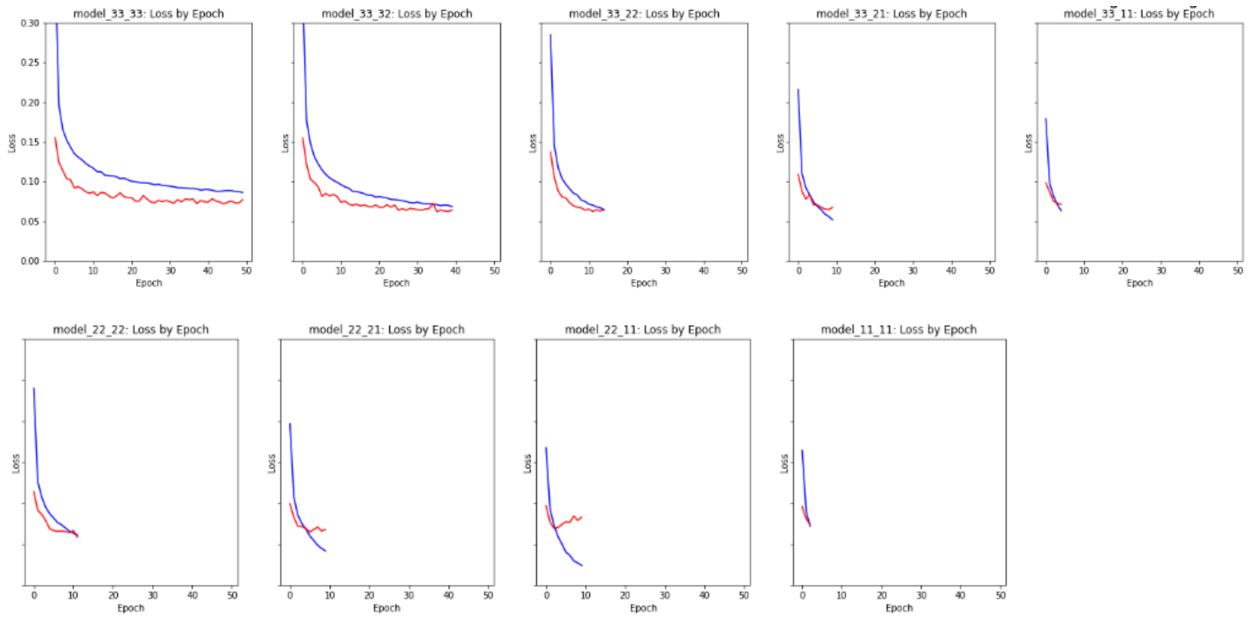
	model	training_time	loss	accuracy	errors
0	model_33_33	643.000480	0.076916	0.975500	245
1	model_33_32	480.126017	0.063556	0.979167	208
2	model_33_22	193.583074	0.063743	0.979500	205
3	model_33_21	206.664375	0.066969	0.979433	205
4	model_33_11	222.265943	0.070604	0.976733	232
5	model_22_22	156.968566	0.058415	0.980800	192
6	model_22_21	223.161683	0.067439	0.980500	195
7	model_22_11	499.374142	0.082790	0.978033	219
8	model_11_11	160.669269	0.073695	0.975700	242







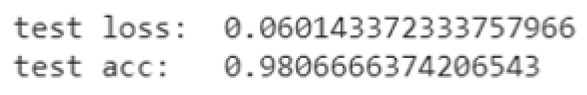
Again, model with window size 2x2 and stride 2x2s (non overlapping) gave the best accuracy, loss, and time. Model_33_33 is still undertrained even after 70 epochs. Models with 1x1 stride resulted in a worst validation loss and least accuracy even when not overfitted.



Application Draw&Learn

I really enjoyed working with this data and I was thinking how I could implement my model. My first idea was to create a “Guess a Movie” app, where a person thinks of a movie, illustrates words from its title, and computer guesses the movie. I started building an app and realized that it gives too many errors when I embed a model trained for more than 10 image categories. So, my idea failed.

Then I just wanted to make an application which recognizes some categories. I used a max pooling model with window size 2x2 and a stride 2x2 (non overlapping). I trained the model for 5 categories first. I was able to achieve 98% accuracy.



```
test loss:   0.060143372333757966
test acc:    0.9806666374206543
```

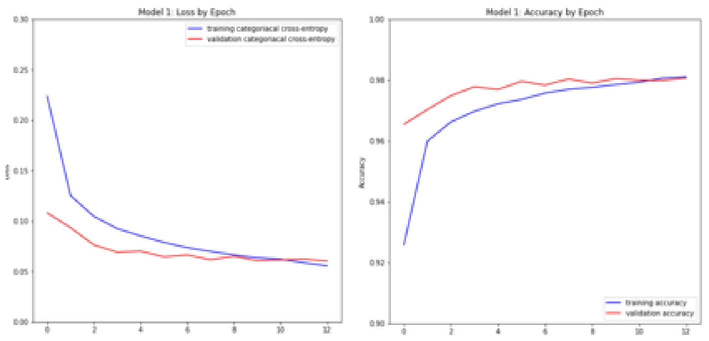
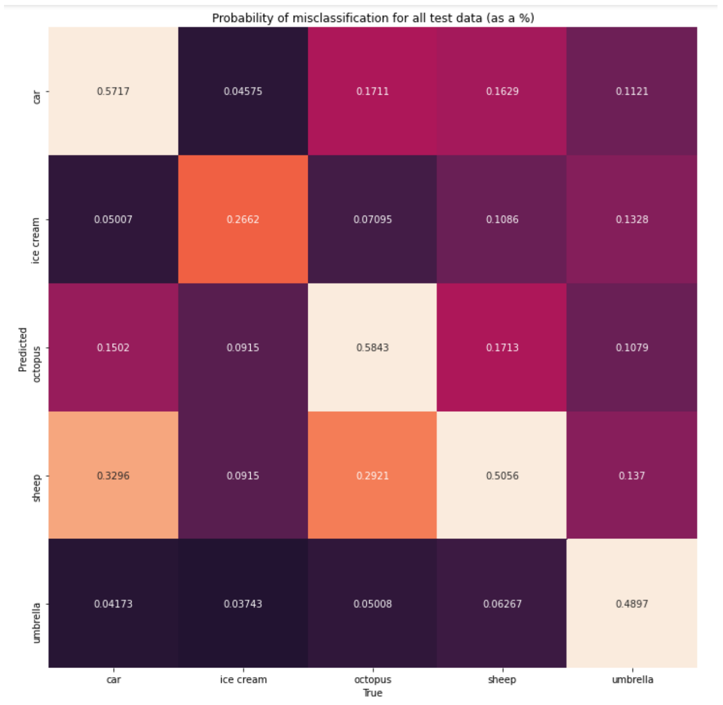


Figure 12. Max pooling with 2x2 window size and 2x2 stride

I was able to implement the app using Flask (what I demonstrated in my presentation).

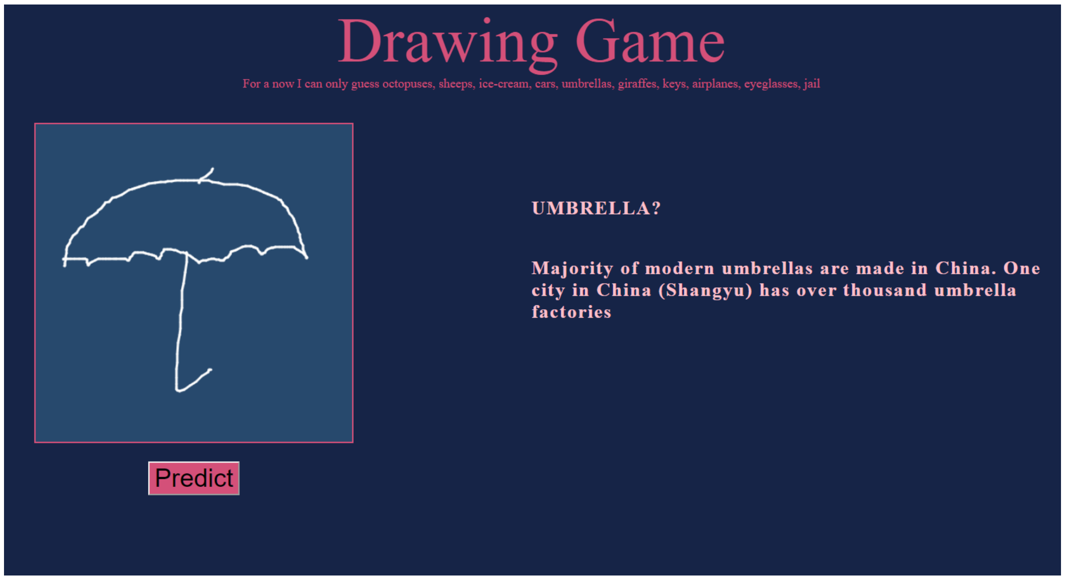
Figure 13 shows that the model still confuses a sheep and a car, and a sheep and an octopus quite often.



	car	ice cream	octopus	sheep	umbrella
car	0.5717	0.04575	0.1711	0.1629	0.1121
ice cream	0.05007	0.2662	0.07095	0.1086	0.1328
octopus	0.1502	0.0915	0.5843	0.1713	0.1079
sheep	0.3296	0.0915	0.2921	0.5056	0.137
umbrella	0.04173	0.03743	0.05008	0.06267	0.4897

Figure 13. Probability of misclassification for the data.

At that point I started working with my friend from CTP and our mentor gave us an idea – what if we print out interesting facts about the object when machine recognizes it? And this is the end result.

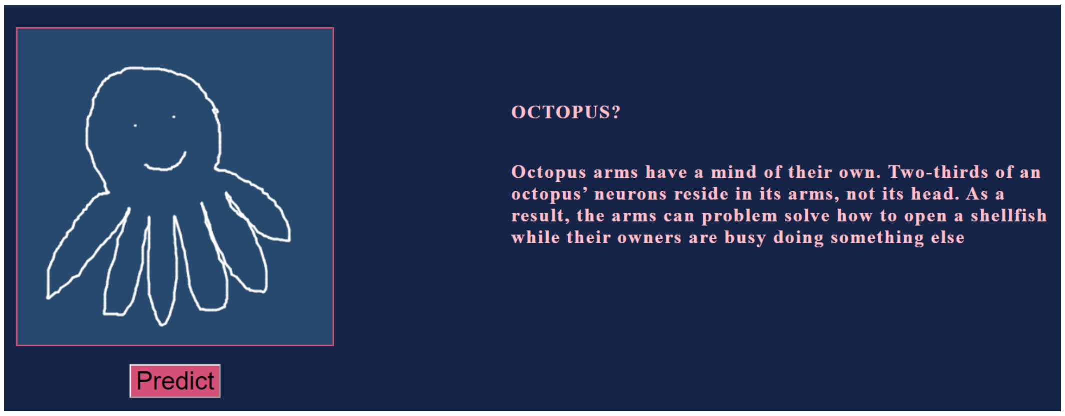


Drawing Game

For a now I can only guess octopuses, sheeps, ice-cream, cars, umbrellas, giraffes, keys, airplanes, eyeglasses, jail

UMBRELLA?

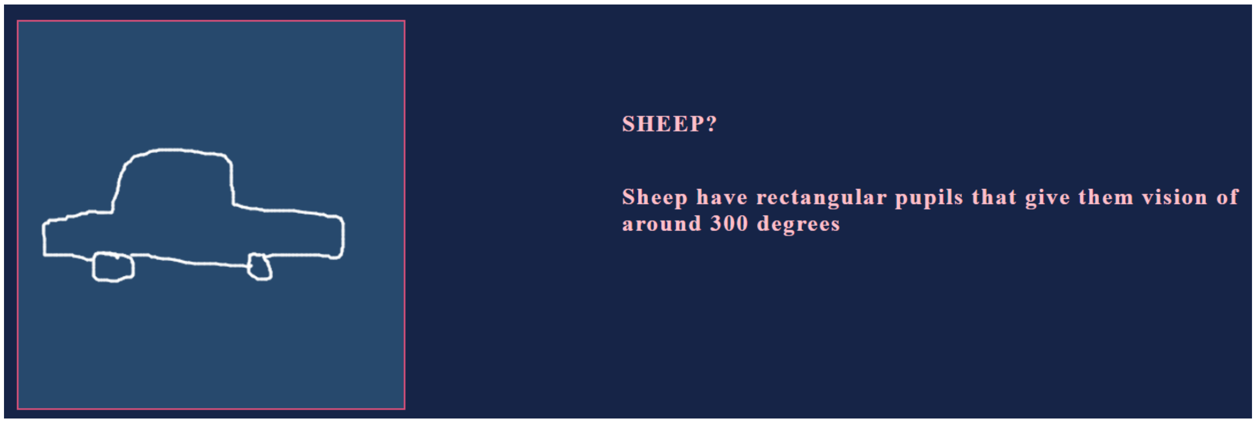
Majority of modern umbrellas are made in China. One city in China (Shangyu) has over thousand umbrella factories



OCTOPUS?

Octopus arms have a mind of their own. Two-thirds of an octopus' neurons reside in its arms, not its head. As a result, the arms can problem solve how to open a shellfish while their owners are busy doing something else

As predicted, the model confuses cars and sheep in 32% of cases:



SHEEP?

Sheep have rectangular pupils that give them vision of around 300 degrees

The next step for this application is to learn how to use original not simplified data and add more categories.

Conclusion

In this project I was trying to see how the size of the pooling window and a stride influence the model performance. I trained 18 models with different pooling methods, window sizes and strides.

Results showed that overlapping and small polling windows size lead to a faster overfitting. My mistake with these experiments was that I let all the models to train for same number of epochs, so some of them got overfitted very quickly. So, I trained the models for different number of epochs to avoid overfitting.

The experiments showed that small strides (severe overlapping) resulted in a worst validation loss and least accuracy even without overfitting.

Besides that, generally average pooling gave a slightly better accuracy than max pooling, however it took longer to train the model.

After working with this data, I was inspired to create a simple app which recognizes objects which the user draws.

The model with 2x2 window and 2x2 stride gave the best result, so I embedded this model in the application. It makes a correct guess in 98% cases among 10 categories. No doubt, there is room for improvement.

Citations

1. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. Dominik Scherer, Andreas Muller, and Sven Behnke University of Bonn, Institute of Computer Science VI, Autonomous Intelligent Systems Group, Bonn, Germany.
2. https://github.com/googlecreativelab/quickdraw-dataset
3. https://towardsdatascience.com/develop-an-interactive-drawing-recognition-app-based-on-cnn-deploy-it-with-flask-95a805de10c0
4. Ian Goodfellow, Yoshua Bengio, Aaron Courville - Deep Learning [pre-pub version]-MIT Press (2016).
5. John Hearty - Advanced Machine Learning with Python-Packt Publishing (2018).
6. Pattern Recognition and Computer Vision: First Chinese Conference, PRCV 2018, Guangzhou, China, November 23-26, 2018, Proceedings, Part I Jian-Huang LaiCheng-Lin, LiuXilin ChenJie ZhouTieniu TanNanning ZhengHongbin Zha.
7. Udacity, Deep Learning Nanodegree Program.