- PROJECT 3 - CONVOLUTIONAL NEURAL NETWORK - CLASSIFY CAT AND DOG

STUDENT INFORMATION

Name	Student ID	Task	Percentage
Đỗ Vương Phúc	19127242	Template coding. Research about optimizers. Transfer learning and VGG. Loss and sliding window.	100%
Hoàng Như Thanh	19127067	Research about CNN layers. Activation function. How to choose hyperparameters.	100%
Bùi Ngọc Chính	19127109	Research about overfit, underfit and justfit. How to prevent overfit, what technique can we use. The different kinds of dataset.	100%

TABLE OF CONTENTS

1 PROBLEM 1

1.1 CONVOLUTIONAL NEURAL NETWORK

1.1.1 USAGE

Convolutional Neural Network (CNN) is typically used when the input of the neural network is an image as it works well in sharpenning, blurring, recognizing patterns and detecting edges in images. CNNs make the unequivocal supposition that the sources are pictures, which permits encoding certain properties into the architecture. These then make the forward works more proficient to carry out and lessen the parameters in the neural network.

To detect the feature of an image, CNNs take advantage of filters. There are operations of a CNN that we will discuss later in this document:

- Convolution
- ReLU
- Pooling
- Dense Layer (Fully-connected layer)

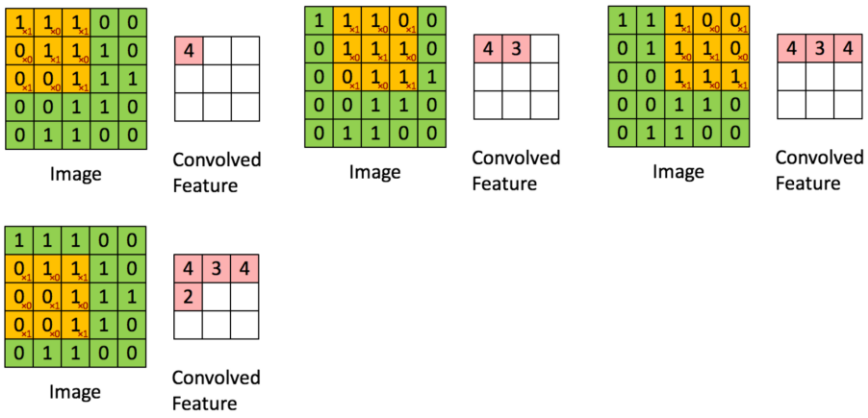
1.1.2 CONVOLUTIONAL LAYER

Convolutional layers are crucial parts of a CNN. But before jumping into it we better know about local connectivity. With high-dimensional images, it is not practical for a neuron to be connected with all neurons of the previous layer; connection of pixels that are next to each other tells more than connected pixels that are far away. The spatial of that connectivity is equivalent to the filter size.

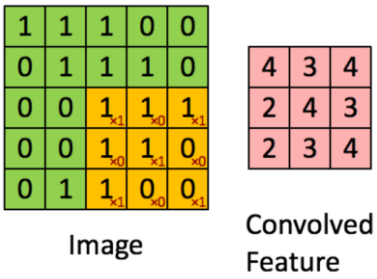
Now what is the filter for, we can see below. For example I decide a filter size 3*3 like this:

1	0	1
0	1	0
1	0	1

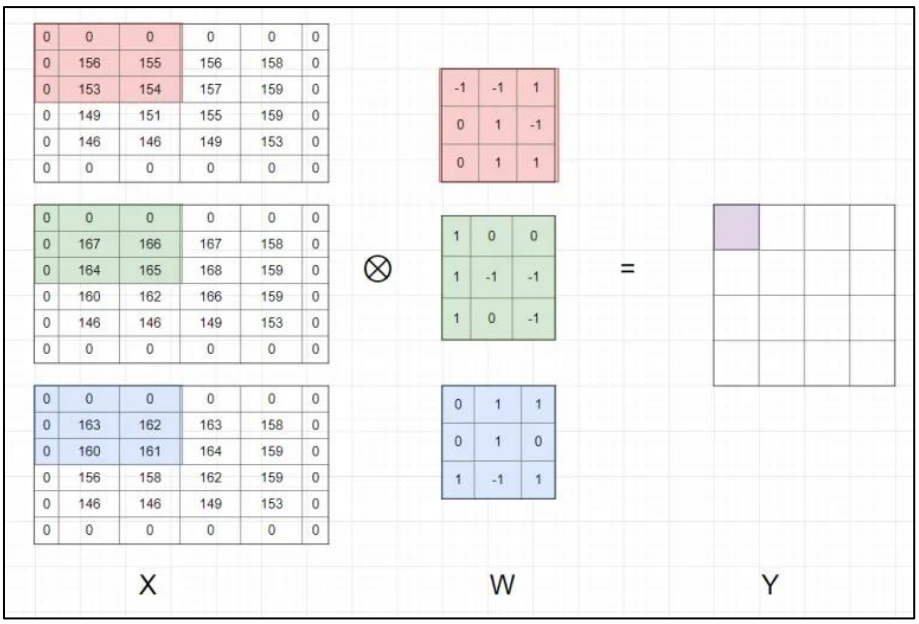
The filter shifts to the right with a stride value that we will discuss later, until it traverses the whole image, each time it will calculate the dot product between the filter-sized patch of the input and the filter and output to the matrix colored with pink. For example I have stride = 1 so it performs like below:



It keeps shifting until it finish:

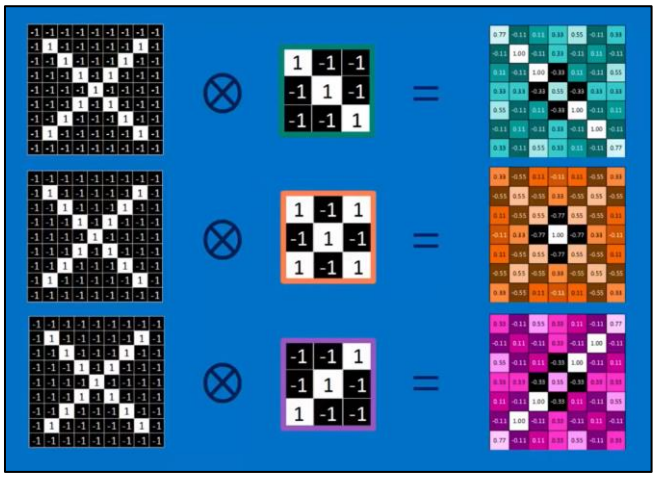


If the input is 3D (image with height, width and depth) so the filter must have the same depth as the input. Therefore, when the filter traverses through an image it finally forms a 2D matrix, not a 3D one. Our input is an image with 3 channels Red, Green, Blue means that depth = 3. So we can just select a filter with the size k*k*3 for example like below:



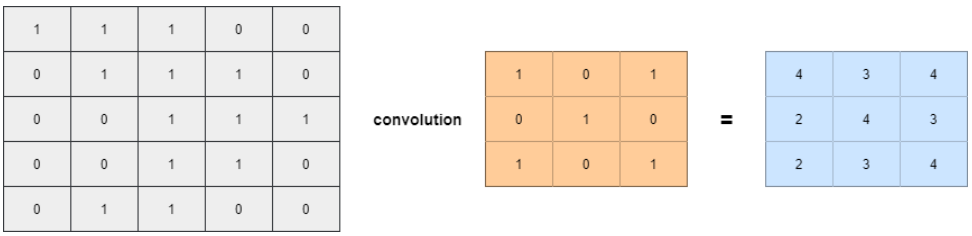
It just calculates the dot product from filter-sized patches from each channel of X with filter W and then sum them up, do not forget to add bias together, finally output to Y.

By that way, when an image goes through a convolution layer, we will have a new matrix representing the detected patterns of the image being highlighted. To be easier to visualize just have a look at the following figure.

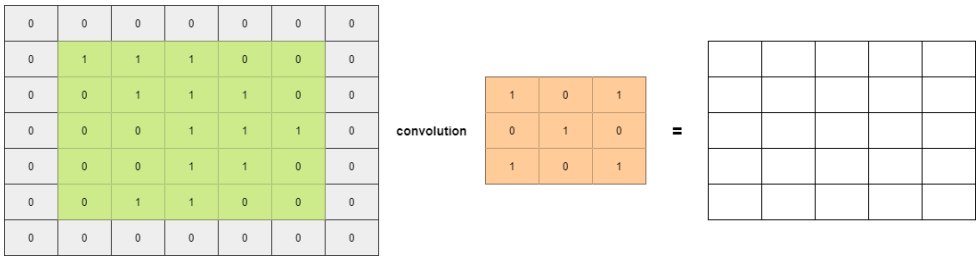


1.1.3 ZERO-PADDING TECHNIQUE

Generally, zero-padding is a technique to create an output matrix whose dimension is similar to the input one. We can see that, if we apply a 3x3x1 filter to a 5x5x1 input the result matrix will be in 3x3x1, which means the same dimension with the filter.



In case we want the output to have the same dimension with the input one, just wrap the input matrix with the zeros, like the figure below. With padding, after doing the convolution we will have the result matrix with 5x5x1, same as the input one.



1.1.4 MAX-POOLING LAYER

Max pooling is a type of operation that is typically added into the CNN following each convolutional layer. Max pooling helps reduce the dimensionality of images by reducing the number of pixels in the output of the previous convolutional layer. Hence, reduce the amount of parameters in the network.

Before getting to know why we would like to add max pooling into our CNN, let us have a look on how max pooling performs.

Firstly, we have to decide a hyperparameter to work with max pooling. For example I have an input like below and choose a hyperparameter with filter 2x2 and stride = 2.

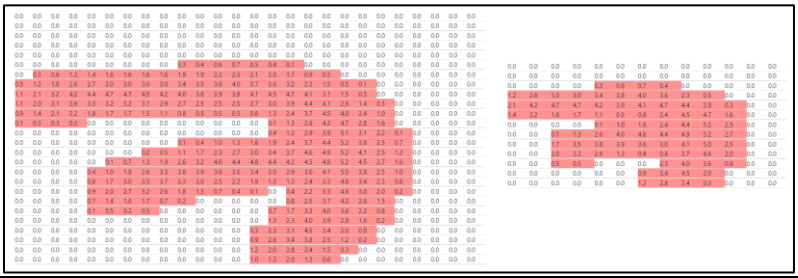
9	8	7	6
1	5	3	2
4	7	4	6
8	2	1	2

Because the filter is 2x2, when first applied the filter to the input, max pooling tries to select out the maximum value among those in the blue patch. Then because the value of stride is 2, the filter right shifts 2 pixels to the yellow patch and does the same, and keeps going on until it traverses the whole input. When max pooling finish we have the result like follow:

9	7
7	6

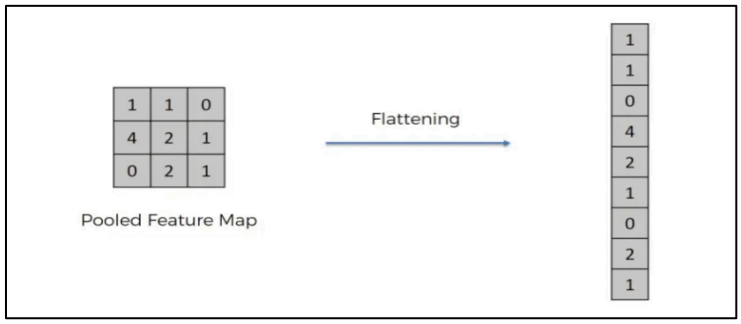
So now we would like to discuss why max pooling is necessary for CNN. Firstly it lessens the resolution of an output, as we can see the output is just 2x2 while the input is 4x4, which means it reduces factoring by 2. More importantly, for a particular image, the network will try to extract some particular features for example edges, curves, etc. From the output of the convolutional layer, max pooling takes the pixels that have highest values as the ones which are the most activated, the most efficient values for representing the features. As max pooling traverses over individual regions from the convolutional output, it is able to pick out the most activated pixels and preserve them in the output to go forward.

Here we can see the more practical work of max pooling in the process of analyzing hand-writing numbers. The dimensionality is reduced 2 times but the particular features stay.



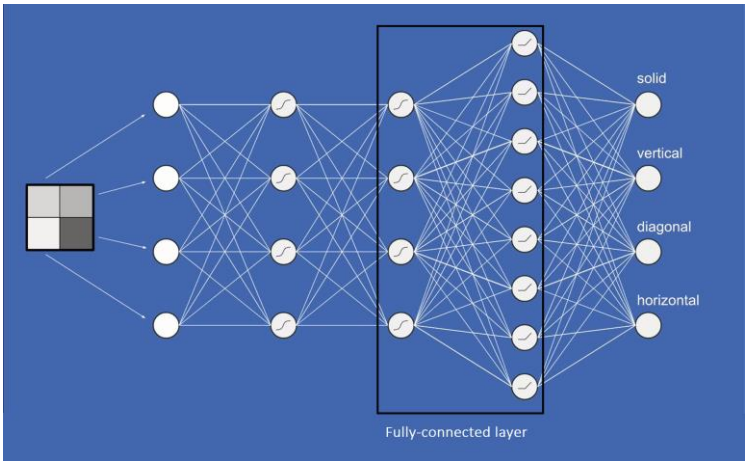
1.1.5 FLATTEN LAYER

After the image is passed through many convolutional layers and pooling layers, the model has learned the relative features of the image (eyes, nose, face frame, etc) then the tensor of the output of the last layer will be converted to a vector. This step is a preparation for the last layers which are fully-connected layers.



1.1.6 DENSE LAYER (FULLY-CONNECTED LAYER)

After the flatten step, each neuron which was just flattened will be connected to every neuron of the dense layer. This layer helps combine the features of the image to get the output of the model, the concept of a fully-connected layer in CNN can be visualized as below.



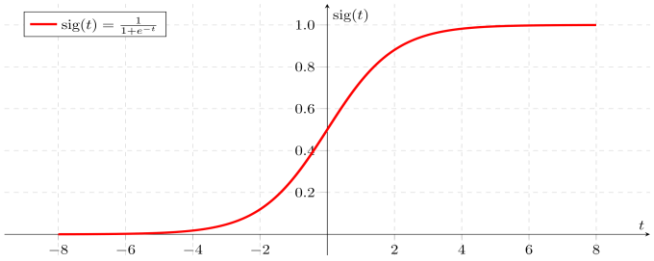
We can observe that after the fully-connected layer takes place, the CNN can decide what the input represents.

1.2 ACTIVATION FUNCTION

1.2.1 SIGMOID

The sigmoid function is crucial in the context of logistic regression, it is to predict the outcome of binary classification problems. In our case, the sigmoid function plays the role of an activation function, it takes the weighted sum of the input features as an input and outputs the probability values [0-1]. If the probability value ≥ 0.5, we assign the value 1 to the prediction label, otherwise we assign 0.

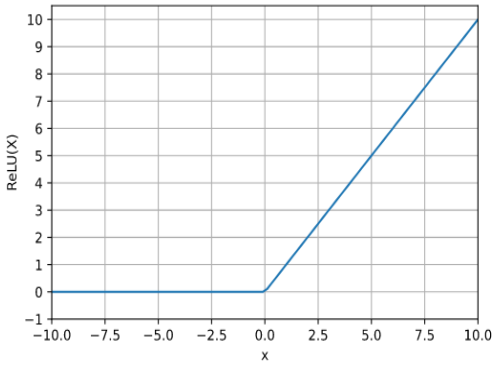
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



1.2.2 RELU

ReLU is one of the most widely used activation functions today. ReLU transforms the input to the maximum of either 0 or the input itself. Therefore if the input is less than 0 or equal to 0 then ReLU will transpose that input into 0, and if the input is greater than 0, the output is an input. The more positive a neuron, the more activated it is.

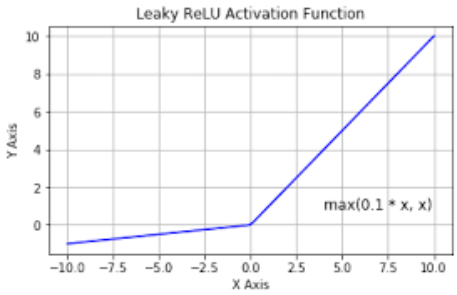
$$f(x) = max(0, x)$$



The ReLU function helps the model with accounting for non-linearities and interaction effects.

ReLU has some variants but the most widely used among all is Leaky ReLU, which attempts to fix the "dying ReLU" problem, instead of outputting 0 when $x < 0$Leaky ReLU has a small positive slope.

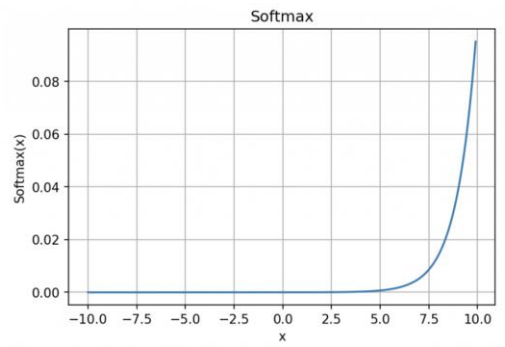
$$f(x) = max(0.01 * x, x)$$



1.2.3 SOFTMAX

Softmax function as known as normalized exponential function calculates the probability distribution of a vector. In our case, softmax function is an activation function to normalize the output of our neural network to a probability distribution over predicted output class. Therefore, it is usually used in multiple class classification and categorical cross entropy.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} for\ i = 1, \ldots, K\ and\ z = (z_1, \ldots, z_K) \in R^K$$



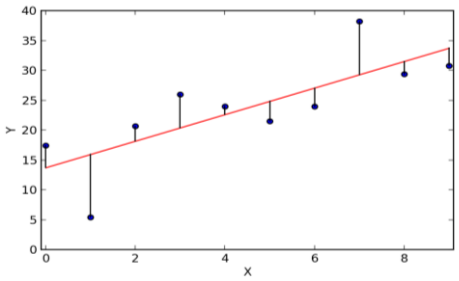
The calculated probability helps decide the correct class for an input value. Sum of probabilities equal to 1 is the most highlighted property of softmax function.

$$\begin{bmatrix} P(\text{cat}) \\ P(\text{dog}) \end{bmatrix} = \sigma\left(\begin{bmatrix} 1.2 \\ 0.3 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{1.2}}{e^{1.2}+e^{0.3}} \\ \frac{e^{0.3}}{e^{1.2}+e^{0.3}} \end{bmatrix} = \begin{bmatrix} 0.71 \\ 0.29 \end{bmatrix}$$

1.3 LOSS FUNCTION

So, to make the machine learnable, we must have some way to measure the error between the prediction and the target. When we work with machine learning, we must choose a loss function to measure the loss. One famous loss function we use in linear regression is MSE (Mean square error). The MSE function calculate the average values of squared the difference

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$$



However, in different fields, we have different loss functions to use. In classification categories, we usually use categorical cross-entropy and binary cross-entropy which are discussed in this section.

1.3.1 CATEGORICAL CROSS-ENTROPY

First, we need to understand some basic concepts.

The multiple-class classification is the task that each sample belongs to only one class. For example, a fruit can be apple, or banana, or a lemon but not both at the same time.

The multiple-label classification is the task that each sample can belong to many target labels. For example, the weather of a day can be both rainy and cloudy, or only sunny, or only cloudy without rain.

Categorical cross-entropy loss is the class used in multiple-class classification. The formula of the loss is:

$$Loss = -\sum_{i=1}^{n} y_i . log\,\widehat{y_i}$$

With the $\hat{y}_i$ is the prediction value and y_i is the target. We can see that, in the multiple-class classification, the total sum of y_i is 1 because there must be exactly one event occurring. In this loss function, we must ensure that the predicted value must be positive.

The softmax activation function we discussed above is the only recommended to use with this loss function. This loss function can be used in or number recognition either cat and dogs recognition. To use the categorical cross-entropy, you need to use one-hot encoding. The one-hot encoding will encode the class into the vector. For instance, the class cat will be [1 0] and dog will be [0 1]. (Notice that the sum of the vector must equal 1).

However, in this project we want the output of the prediction to be only a binary single number (e.g. [1] for cat and [0] for dog). So in the project, we use Binary Cross-entropy.

1.3.2 BINARY CROSS-ENTROPY

Unlike the categorical cross-entropy, binary cross-entropy is the function which is used for the multiple-label classification. However, each classification task must be a binary classification task (Yes or no answer). Let's take the weather example above, the task is to answer whether it rains or not; it is sunny or not; it is cloudy or not. We can see that each task is a binary answer. If a day is both cloudy and rainy, we can encode it as a vector [1 0 1].

In this kind of problem, we can see it does not require the sum is 1. However, it still requires the value of the prediction to be between 0 and 1 (as the probability).

The loss formula is:

$$Loss = -\frac{1}{n}\sum_{i=1}^{n}[y_i.log(\hat{y_i}) + (1-y_i).log(1-\hat{y_i})]$$

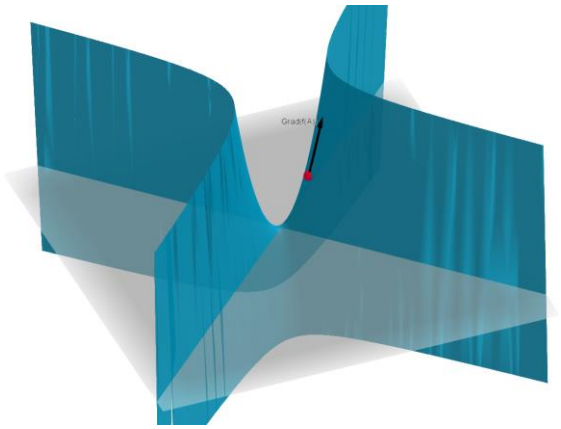
This is equivalent to the average result of the categorical cross entropy for many independent classification problems. The sigmoid activation function is the only function compatible with this loss function. In this project we use this loss function so the last layer must use the sigmoid activation function.

1.4 OPTIMIZERS

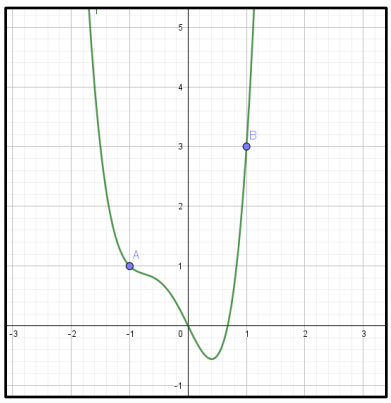
After we can measure the loss, we must have some algorithm to help the machine improve itself. There are various algorithms to do this, they are called the optimizer. The target of the optimizer is to reduce the loss of the prediction. In this section, we will explore the most basic optimizer: Gradient Descent and its variants.

1.4.1 GRADIENT DESCENT

For easier to write, in this section, we call the loss function is $L(w)$ with w is the set of weight. The idea of gradient descent is that after the prediction (forward propagation), we know the loss and the weight at that time. So if we take the gradient $\nabla_w L(w)$ (the sign is called nabla) it will give us the direction vector which increases the loss if we move by that vector from w. To decrease the loss, we will move in the opposite direction of that gradient. For example, in the function 2 variable $f(x, y) = x^2 - y^2$ we calculate the gradient vector at point $A = (2,0)$ is $\nabla f(A) = (\nabla_x f(A), \nabla_y f(A)) = (4,0)$ and it is the same as image below:



Take an easier problem, begin with the 1 variable function $f(x) = 2x^4 + 3x^3 - 2x$ and given two points $A = (-1,1)$ and $B = (1,3)$. The derivative of the function is $f'(x) = 8x^3 + 9x^2 - 2$. So the direction of those given points are $f'(A) = (-1)$ and $f'(B) = 15$:



We can see that if the point A moves to the left (decrease of x-axis) it will increase the value of our function. Likewise, the point B moves right will also increase the value of our function. To decrease the function, we will move negatively to the value of the gradient (this is why we call gradient descent - descent means go in the reverse way).

So at each iteration, we will update the variable x with a value which is proportional to the negative of the gradient vector:

$$x = x - \alpha . \nabla_x f(x)$$

The value α is called learning rate. In the same way for multiple variables, we just update the parameters as:

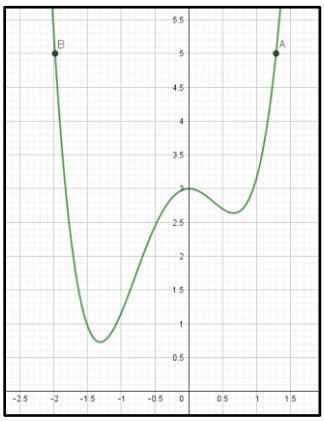
$$w = w - \alpha \nabla_w L(w)$$

However, in this algorithm, there are still many problems.

One problem that we can easily see is the initial point. With two points above, point B will take less time to reach the local minimum value than point A does (usually called as converged).

The second problem is the learning rate is too large which causes the overshot. For example, with the learning rate $\alpha = 1$, the point B after the first iteration, it will be at $B' = (-14{,}85036)$; the second iteration will be at $B'' = (20176, f(20176))$. This means that it cannot converge.

Another problem is, what if our function has many local minimum values? We may not find the global minimum value because we have been stopped at the local one (gradient vector is 0). For example, the function $f(x) = x^4 + x^3 + 4cos(x) - 1$ has two local minimums:



We can see that point A may not be able to reach the global minimum, or point B may overshot the local minimum. To improve those problems, we have many variants of gradient descent (GD) we will discuss below.

1.4.2 MOMENTUM

From the GD algorithm, we can imagine that we have a ball and we drop it from a point in the valley then it will roll down. However, in real life, the ball can go through the local minimum position as its residual velocity. From this idea, another algorithm is developed in order to improve the disadvantages of GD.

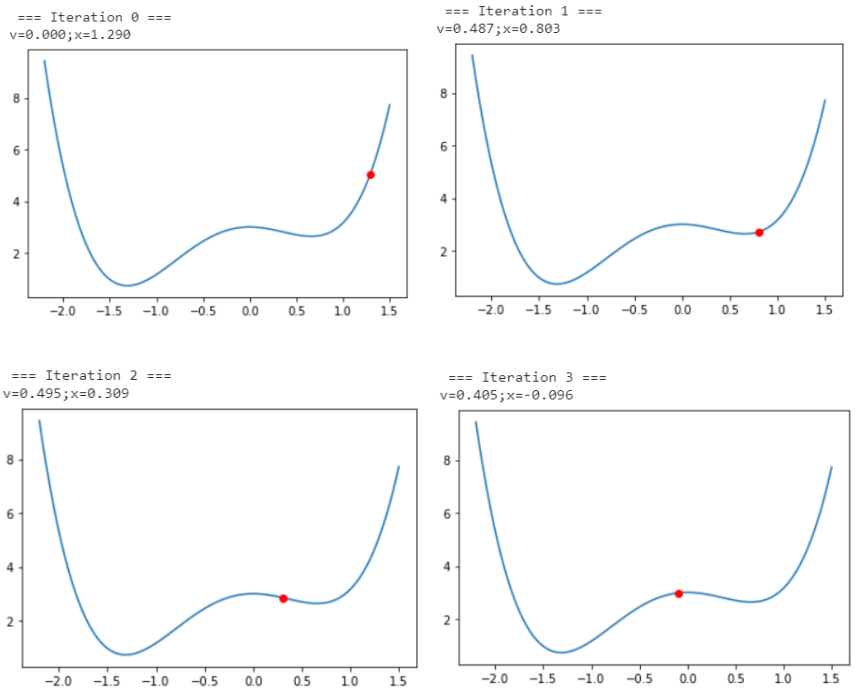
In physics, this is called momentum. When we use GD, the weight is updated after each iteration by calculating the gradient. However, in momentum, we treat the amount of updating as the velocity:

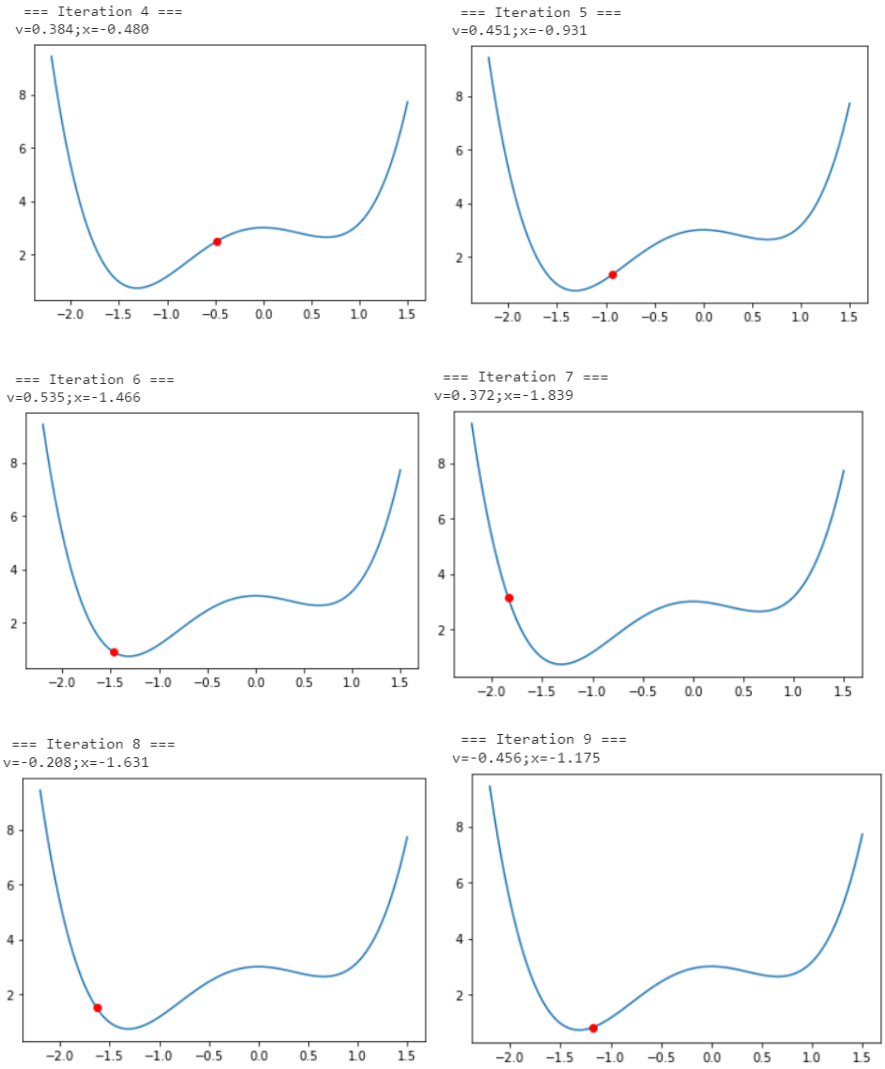
$$w = w - v_t$$

The negative sign is to represent the "descent" of the gradient (move in opposite to the direction of gradient). Now, we just need to construct the formula for the velocity. We know that the velocity at a moment must contain the information from the velocity at the previous moment and the vector gradient. Indeed, in real life the velocity is lossy because of many difference element (such as friction), so our formula should be like:

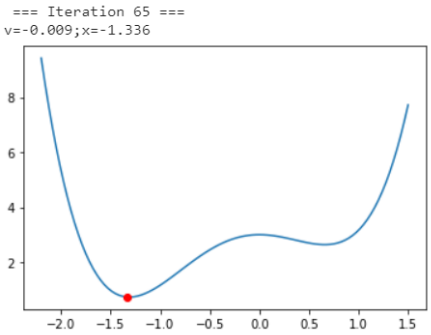
$$v_t = \gamma v_{t-1} + \alpha \nabla_w L(w)$$

Usually, people choose the $\gamma = 0.9$. In physics, the gradient part takes the role of being the accelerator. Let try for the problem about using the learning rate $\alpha = 0.05$, I have visualize the position of the point after each iteration below:





From the example above, we can see that it can improve the disadvantages of traditional GD. However, it still takes a very long time to converge. If we continue to run the process above, it will converge at iteration 65:



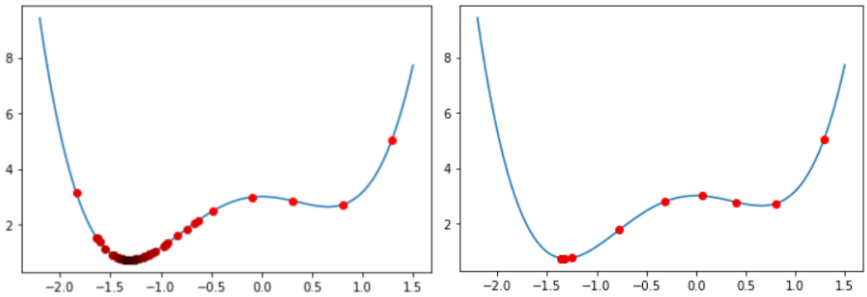
1.4.3 NAG

As we see at the momentum algorithm, it still has the pros. Another method to help our problem converge faster is NAG (Nesterov accelerated gradient).

The idea is we will "predict" the future. It means that from the momentum, we can estimate approximately the next position as $(w - \gamma . v_{t-1})$ then we use the gradient vector of the "future" position:

$$v_t = \gamma . v_{t-1} + \alpha \nabla_w L(w - \gamma . v_{t-1})$$

So now, let's take a comparison between momentum and NAG. On the left image is the visualize of the position our point run after each iteration using momentum and on the right is using NAG:



Unlike momentum, the NAG algorithm takes only 9 iterations to converge instead of 65 iterations as momentum does.

1.4.4 SGD (STOCHASTIC GRADIENT DESCENT)

The gradient descent (GD) algorithm above is also called Batch gradient descent. Batch here means that each time we calculate the gradient and update the weight, we use all the data we got.

This approach is not very good for the dataset which is big (such as google, facebook, etc). The cost we must pay to calculate the loss and do the optimization are very large. In online learning, whose data increases instantaneously, we can see that this algorithm is not effective. In practice, we do not use gradient descent for this problem, we can use its variants SGD (Stochastic gradient descent).

In the SGD algorithm, we update the weight using "only one" data point instead of using the whole dataset. Each time we go through the whole dataset is called one epoch. So we can understand that the GD algorithm updates the weight once per epoch whereas SGD update N times per epoch (N is the number of data points in the dataset).

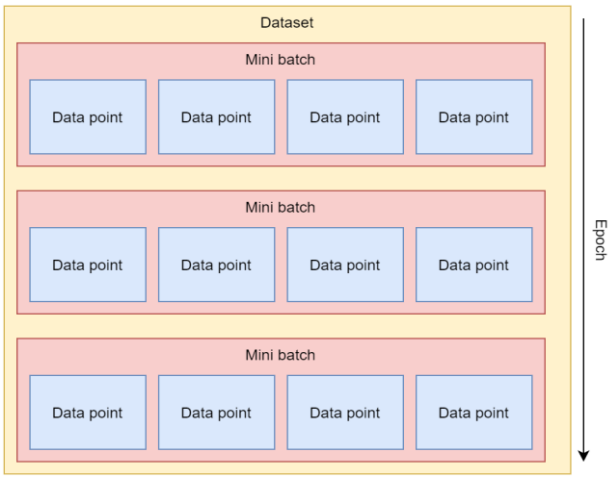
However, we must notice that we must shuffle the dataset after each epoch to ensure the randomness.

1.4.5 MINI-BATCH GRADIENT DESCENT

In deep learning, people usually use another algorithm called mini-batch GD. This is similar to stochastic GD, at the beginning of each epoch, we also shuffle the dataset and then we divide the data into mini

batches. The difference is in stochastic GD, we update the weight for each data point while the weight in mini-batch GD is updated for each batch. In other words, the stochastic GD is the mini-batch GD with the batch size is 1.

Usually, the batch size is in the range from 50 to 100 data. People usually choose the batch size 32 and 64 - as the power of two (for fast computing). Both stochastic GD and mini-batch GD are able to be applied with momentum and NAG method.

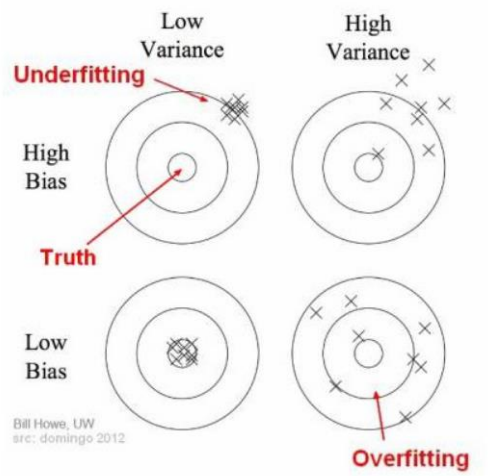


In this project, my model will use mini-batch GD with both momentum and NAG method.

1.5 OVERFIT PROBLEM

1.5.1 BIAS - VARIANCE

1. The bias is error from erroneous assumptions in the learning algorithm (or we can simply understand it like a stat that shows up the inaccuracy of our trained model). So that, the high bias can cause underfitting because of the low accuracy of our model.
2. The variance is error from sensitivity to small fluctuations in the training set (the difference between the predict of our model and the test dataset, more difference more invalidation). In other words, it means that the high variance can lead our models to the overfitting problem by the low valid accuracy.
3. Relative of bias-variance:



1.5.2 WHAT IS OVERFITTING, UNDERFITTING?

1.5.2.1 UNDERFITTING

Underfitting happens when the statistical model or machine learning algorithm cannot capture the underlying trend of data. When this problem occurs, it simply means that our model or algorithm does not fit the data enough.

It can be avoided by using more data and reducing the features by feature selection. Techniques to reduce underfitting:

- Increase model complexity.
- Increase the number of features and perform feature engineering.
- Remove noise from the data.
- Increase the number of epochs or increase the duration of training.

1.5.2.2 OVERFITTING

Overfitting happens when we train a model with a lot of data and it starts learning from the noise and inaccurate data entries in our data set. This problem makes our model categorize the data incorrectly because of too many details and noise.

We can avoid it by using linear algorithm with linear data or using the parameters like the maximal depth if we are using decision trees. Techniques to reduce overfitting:

- Increase training data, reduce model complexity.
- Early stopping during the training phrase.
- Ridge Regularization and Lasso Regularization.
- Use dropout for neural networks to tackle overfitting.

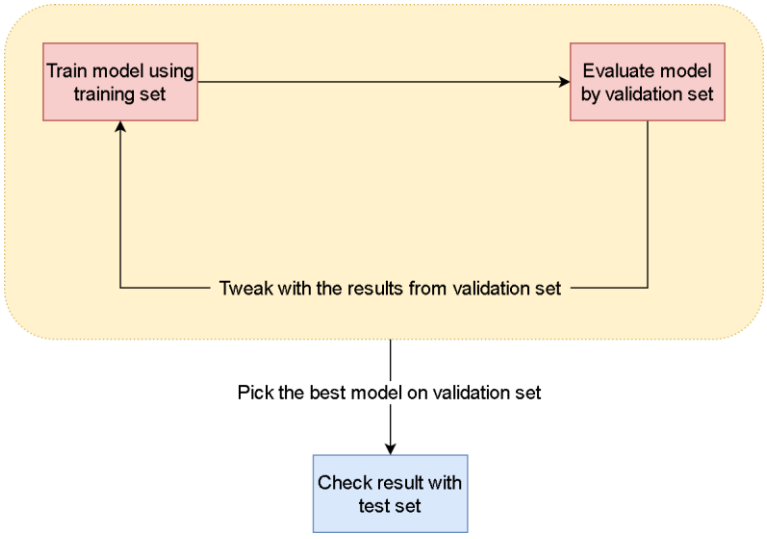
1.5.3 DATA VALIDATION AND TESTING

- Training set: used for training model.
- Test set: used for testing the trained model.
- Validation set: used for evaluating the result of training set.

The test set must follow these requirements:

- Is large enough to yield statistically meaningful results.
- Is suitable with the training set.
- Is different from training set.

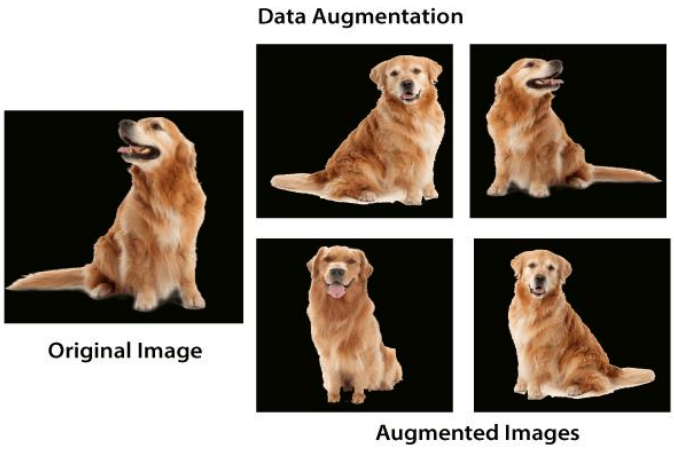
Workflow:



1.5.4 DATA AUGMENTATION

Data augmentation is a way to redece overfitting. In data augmentation, we create new data based on modifications of our existing data. (Augmented data is created based on training set to create more samples).

Usually we can: rotating image, shifting the width & height, zooming, varying color, horizontally flipping the image. For instance:



1.5.5 VANISHING AND EXPLODING WEIGHT

The vanishing gradients problem occurs when we train the model with gradient-based methods and backpropagation but in some cases, there are some problems with the update weights of neural network, it will be vanishingly small. This problem prevents the weight from changing value.

The exploding gradients problem occurs when the model has to update large gradients, it leads to large updates to weight, turns the model into an unstable model. In worst case, it can lead to overflow.

To handle this problem, we can use the kernel initialization technique to create the initial weight.

1.5.6 REGULARIZATION

Regulariztion is a technique which makes changes to our model to such that the model generalizes better. It also improves the model's performance on the unseen data as well.

By penalizing the coefficients, Regularization helps us to optimize the value of its coefficient in order to obtain a well-fitted model.

Before we go into the L2 and L1 regularization, we should understand *the Cost function* which is used by L2 and L1.

$$Cost\ function\ =\ Loss(say, binary\ cross\ entropy)\ +\ Regularization\ term$$

When the Regularization term increases, the Cost function also increases, it makes the values of weight matrices decrease. So that it will also reduce overfitting to quite an extent.

1.5.6.1 L2 REGULARIZATION

Is a neat little way to ensure that our training model is not overfitted. When our data size is relatively small, **L2** can improve predictions made from new data (like reduce Variance) by making the predictions less sensitive to the Training Data.

$$Cost\ function\ =\ Loss\ +\ \frac{\lambda}{2m}\ *\ \sum \|w\|^2$$

λ is the regularization parameter, is the hyperparameter whose value is optimized for better results. **L2 regularization** is also knows as weight decay as it forces the weights to decay towards zero (but not exactly zero). In linear regression, this is **the ridge regression**.

1.5.6.2 L1 REGULARIZATION

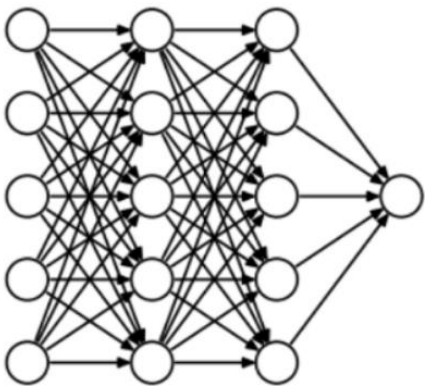
Is similar to **L2 regularization**, but there is a huge difference between them. While the L2 penalty the square of the weight, L1's coefficient the absolute value of weight to it.

$$Cost\ function\ =\ Loss\ +\ \frac{\lambda}{2m}\ *\ \sum \|w\|$$

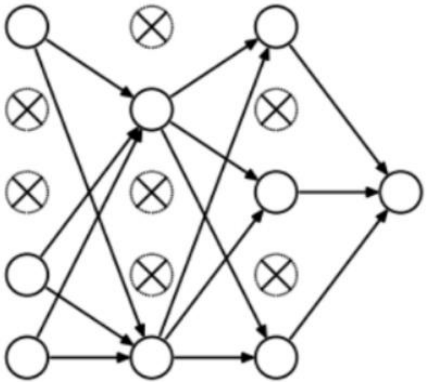
Unlike **L2 regularization,** the weights may be reduced to zero here. Hence, it is useful when we're trying to compress our model. Otherwise, we usually prefer **L2** over it.

1.5.7 DROPOUT

This technique is usually preferred when we have a large neural network structure in order to introduce more randomness. By randomly remove all incoming and outgoing connections of some nodes in each iteration, it makes our model more simpler and effiecient. To make it easier to understand, let see the shown images below:



Assume that it is our neural network structure, and the below image is our structure after drop out:



We can see it more simpler by remove some connections of nodes. So each iteration has a different set of nodes and this results in a different set of outputs. It can also be thought of as an ensemble technique in ML.

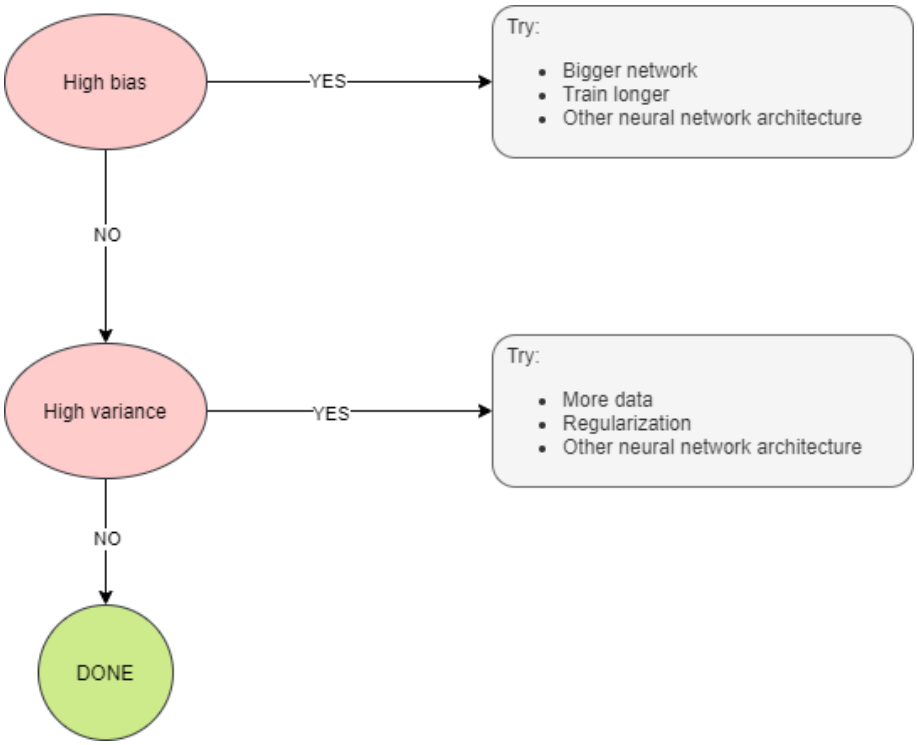
1.6 CHOOSING LAYER FOR CNN ARCHITECTURE

As discussed above, looking at training results and validation results can help us diagnose whether the network has a bias or variance problem or maybe both. With that information we can systematically improve our network performance.

When working with neural networks, we may have to make lots of decisions like how many layers will the network have, what is the learning rate and what is the activation function and so on. At the first time the neural networks come to work it is impossible to get all these values to be correct so we have to build up by and by.

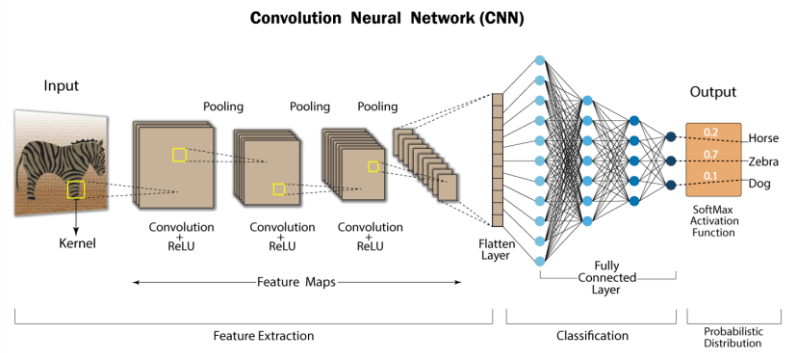
Firstly, start with a simple neural network and after training the initial model we will first look at the training set to decide if our model has high bias. If our model is suffering from high bias we can try a bigger network by adding more layers and hidden units, or we can train for longer, etc. Secondly, when we have managed to fit the data well, we continue on checking the variance problem. In order to due with high variance we can consider getting more data or work with regularization.

We will keep considering the two above steps until both bias and variance problems are solved. So with these two steps we can see that the neural network can add more layers or get more data, that is how we achieve the big neural network as a result.



Now let us talk about which layer to select and how to choose appropriate hyperparameters. Since machine learning has gone through years of researching now we are granted trends for building up a CNN. According to the trend recipe, we know that it is good to follow these sequence of layers: Conv-Pool-Conv-Pool or Conv-Conv-Pool-Conv-Conv-Pool, and surely the last layer will be a dense layer. For trend in numbers of channels we can use 32–64–128 or 32–32-64–64. With filter size, we often use odd numbers like 3x3, 5x5, 7x7. With max-pooling filter size we use 2x2, 3x3 with stride = 2, larger filter size and stride is used when we want to shrink a large image down to the moderate size. Last but not least, try using zero-padding if the image's borders are important.

We can have a look at the figure below to be more referenced.



1.7 RESULT

Because this project is using only three kinds of layers: Convolution layer, Max pooling and dense layer (fully connected), we begin with the small network with VGG-style.

First, we improve all the libraries we need to use. To load up and preprocess the image, our team used the opencv2 library. After we successfully load those images, we use the os library (operating system) to get the files name and create the data frame using pandas library. Because of using the mini-batch GD, we need to divide the data into batches so we pass the data frame to the ImageDataGenerator of the Keras library to do the preprocess for us. In addition, to visualize the image and the graph, we also use the pyplot module in the matlibplot library.

Our dataset has a total 25.000 images, we divide it into two sets: train set (80%) and validation set (20%). About the test set, we can use the validation accuracy to evaluate the model or we can test it using Kaggle.

```
Found 20000 validated image filenames belonging to 2 classes.
Found 5000 validated image filenames belonging to 2 classes.
```

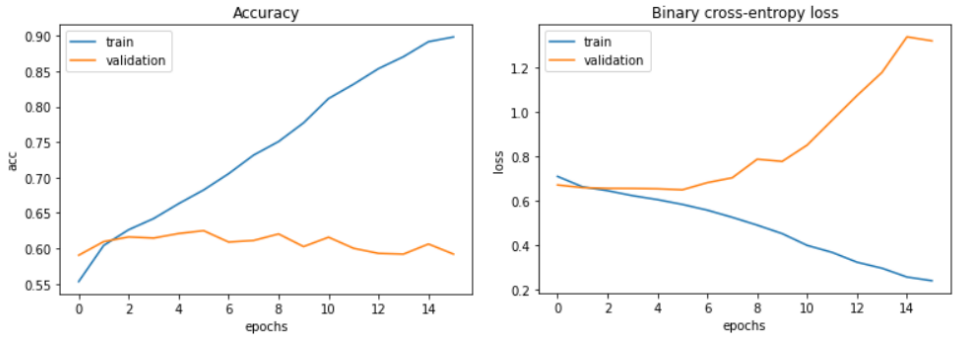
On the first step, we just rescale all the pixel by the factor of $\frac{1}{255}$ to map those values to [0,1], resize the image into (224, 224) size and we do not use the data augmentation. As we already mentioned, in this project we use the mini-batch GD so we divide the train set into batches of 64 data and validation set into batches of 32 data. Of course that we just only need to shuffle the train set to ensure the randomness. Shuffling the validation set does not give us anything, so we do not need to shuffle the validation set.

As we already said in the “How to choose CNN layer” section, we first start with the most simple CNN architecture using VGG-style. The VGG model is a model which achieved top performance in the ILSVRC 2014 competition as it is easy to understand and implement. The model is stacked using convolution layers with a 3x3 max pooling layer.

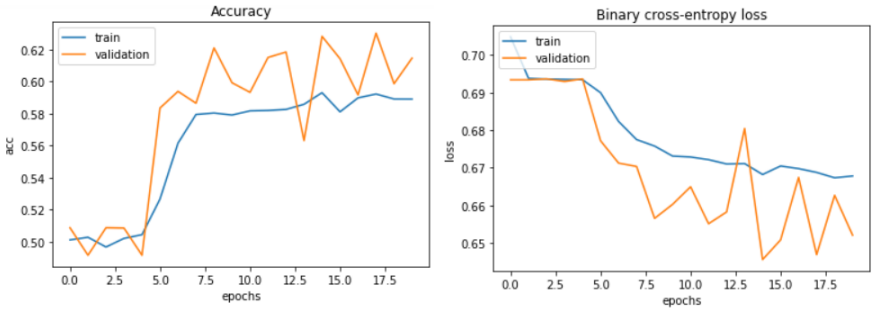
So the first model is look like:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 32)      0
_________________________________________________________________
flatten (Flatten)            (None, 394272)            0
_________________________________________________________________
dense (Dense)                (None, 512)               201867776
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 201,869,185
Trainable params: 201,869,185
Non-trainable params: 0
_________________________________________________________________
{'name': 'SGD',
 'learning_rate': 0.05,
 'decay': 0.0,
 'momentum': 0.9,
 'nesterov': True}
```

To measure the loss, we use the binary cross entropy. Because the training process costs a lot of resources and time, we define a checkpoint callback to save the best model whose validation accuracy is highest. We choose to train the model with 50 epochs and use an early stop to detect the convergence whenever there is no improvement after 20 epochs (patience). And we have the result of:



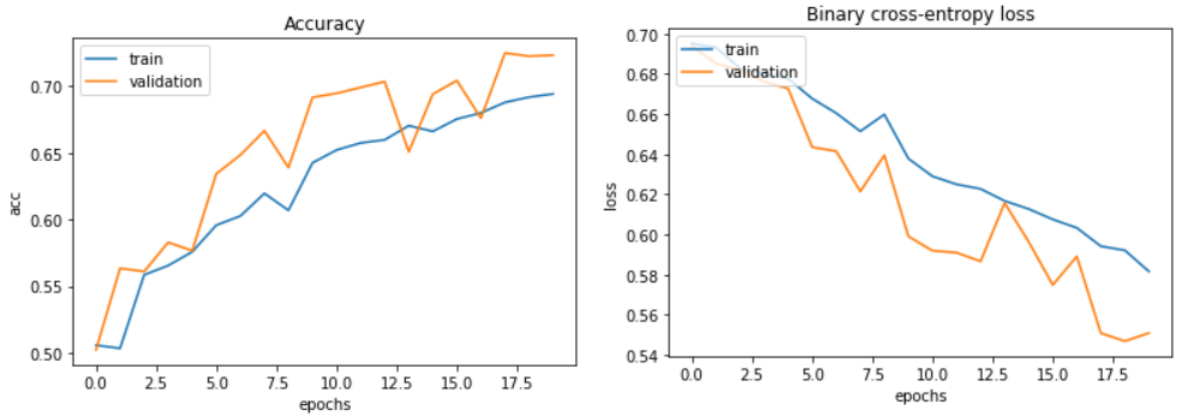
We can see that the model has very high accuracy (90%). However, after we use the validation set to validate, the accuracy is only around 62%. This means that the model is **overfitting**. So we try to use the **data augmentation** to generalize the model. The result now have some differences:



Now the model is no longer being overfitted. However, the model is still low accuracy (62%), which mean high bias, so we increase the complexity by using 2-block VGG model:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 186624)            0
_________________________________________________________________
dense (Dense)                (None, 512)               95552000
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 95,571,905
Trainable params: 95,571,905
Non-trainable params: 0
_________________________________________________________________
{'name': 'SGD',
 'learning_rate': 0.05,
 'decay': 0.0,
 'momentum': 0.9,
 'nesterov': True}
```

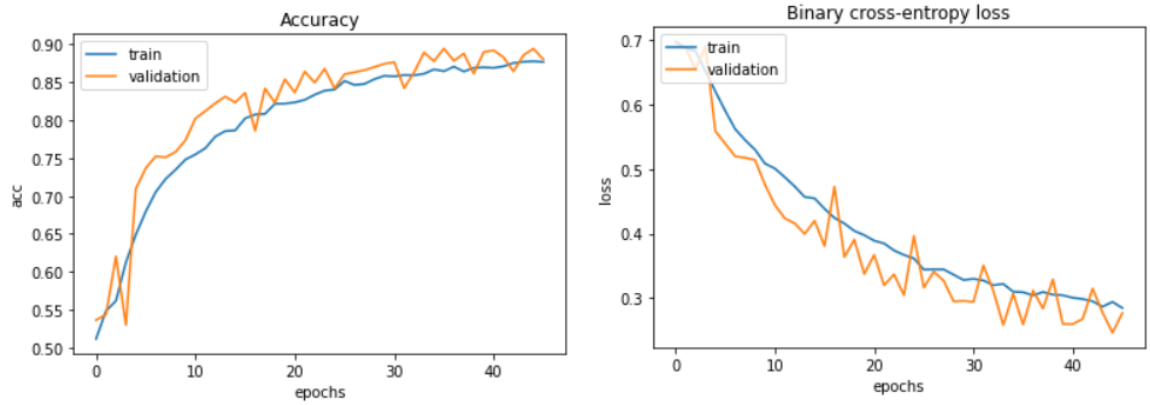
This is the result we achieved from 2-block VGG model, the accuracy now is increased:



Now the accuracy of validation data is approximately 73%. However, we want to reach a better prediction, so we try with 3-block VGG model:

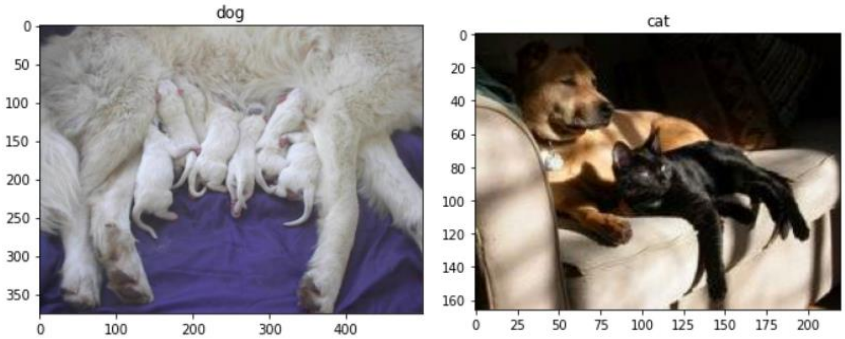
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 86528)             0
_________________________________________________________________
dense (Dense)                (None, 512)               44302848
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 44,396,609
Trainable params: 44,396,609
Non-trainable params: 0
_________________________________________________________________
{'name': 'SGD',
 'learning_rate': 0.05,
 'decay': 0.0,
 'momentum': 0.9,
 'nesterov': True}
```

The result is much better (89%) though it learn still slow:

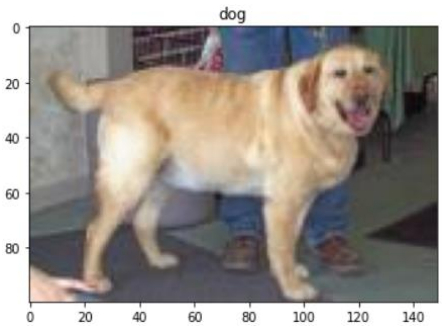


Now let's try to use this model to predict some image. I will load up randomly 50 image from the test set given by Kaggle to see how accurate the prediction is:

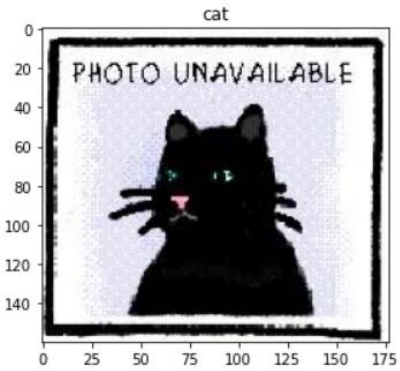
Over 50 images we tested, there are 45 images that the model predicts correctly. However, there still exist some wrong predictions. In our test set there are some special image that may hard to predict because the training set may not have:



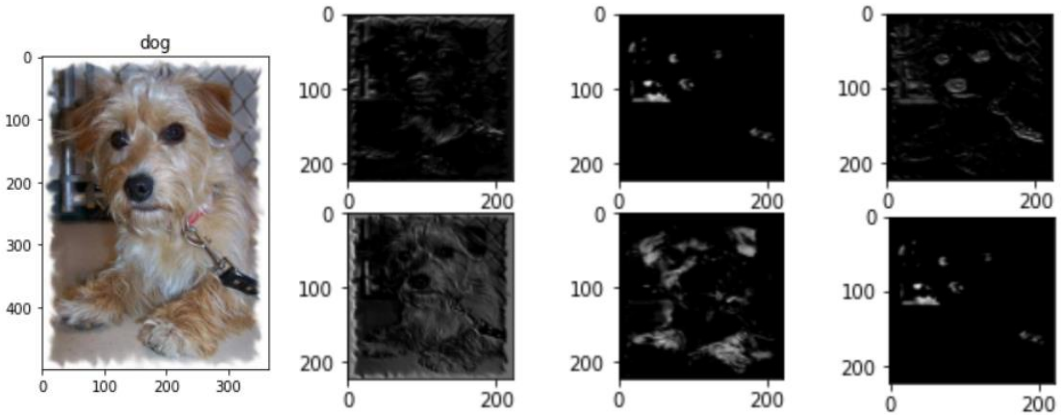
Or some image with very low resolution:



And some hand drawing image:



To improve the model, we have some solutions which will be discussed in the next section. Because the convolution layer is doing such an incredible thing, we already visualized the output image from the convolution layer to see what it has done. For instance, we take one random image:



The right image is the output of the convolution layer in the first block, we can see that it can recognize the eye and nose of the dog.

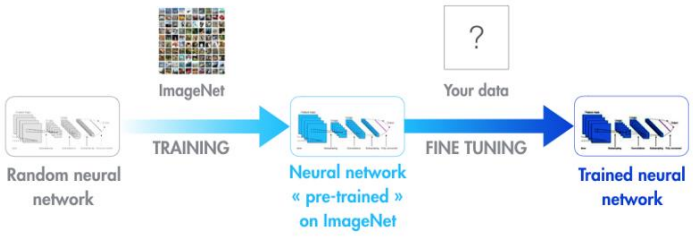
1.8 FURTHER STUDY

1.8.1 TRANSFER LEARNING

One of the methods people can use to improve the prediction is transfer learning. Transfer learning is the method that you use the result or knowledge you achieved from one problem (source) to solve another related problem (destination). For example, we can use the knowledge we achieved in the calculus course to solve the problem in the Artificial Intelligent course.

In our problem, we can use some pre-trained model such as VGG, Resnet, InceptionNet to get the pre-prediction result. In the next part, I will try to use VGG and modify its top layer.

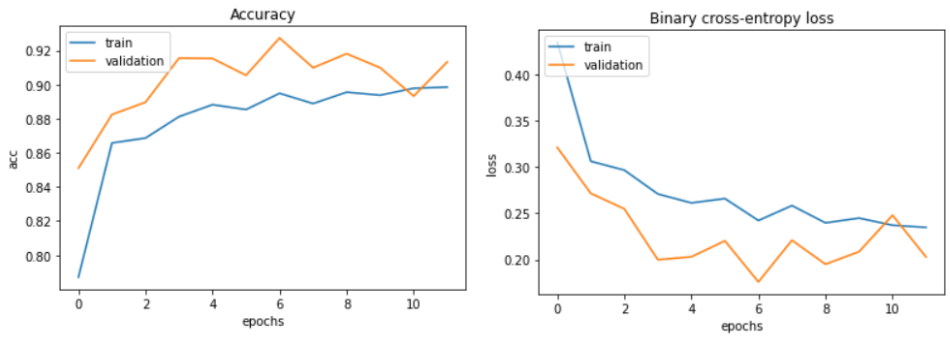
And there are many ways to use the transfer learning in Deep Learning like warming the network by another big dataset (such as ImageNet) then train with our dataset. There are many more which we will not discuss deeply in this document.



1.8.2 VGG

We can try to use the pre-trained model VGG16 and modify it a little bit. In this project, we have loaded the VGG16 model without the top layer then freeze all the weights. After that we add two dense layers on the top and we choose the learning rate 0.001 with momentum 0.9:

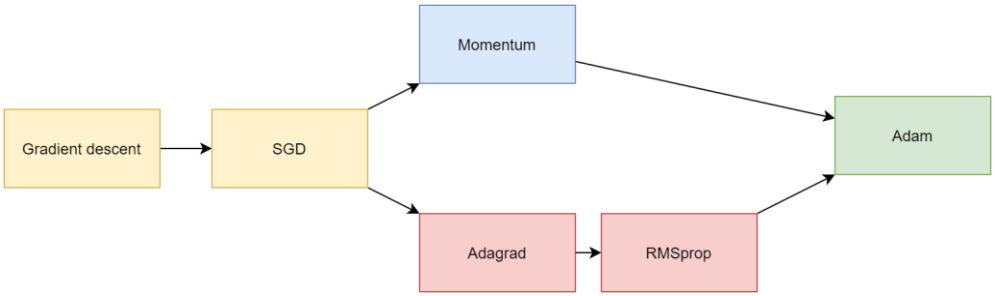
And after trained the result is impressive, the validation accuracy is very high 92% and on the first epoch it learn very fast:



Not only VGG16, there is another deeper version which is VGG19 we can try in future.

1.8.3 ADAM

In this project, we use the mini-batch gradient descent with momentum and NAG technique as an optimizer. In practice, people usually use another optimizer called “Adam”. There are many optimizers used in Deep Learning like Adagrad, RMSprop. Each one has their properties, such as Adagrad can “learn” the hyper parameter learning rate also. Adam is an optimizer which is created from the Momentum and RMSprop:



2 PROBLEM 2

2.1 SLIDING WINDOW

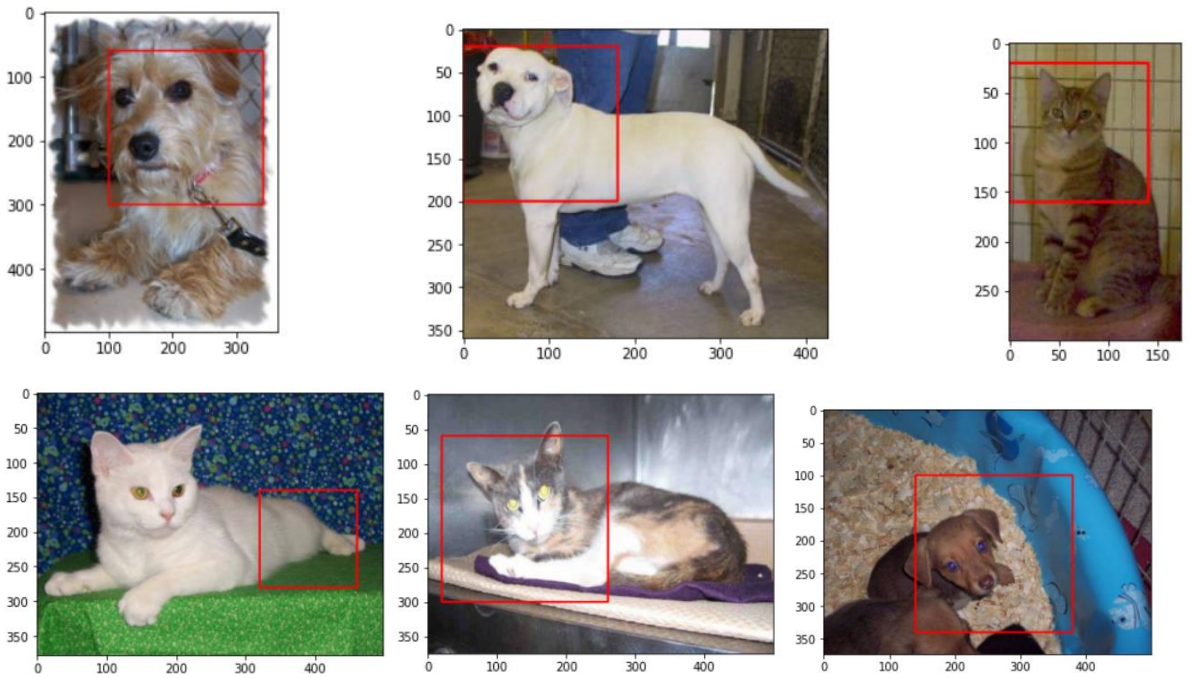
In the second problem, we will detect where the object (dog and cat) is sitting on the image. To do this, we have many solutions such as training a new model using pascal dataset; using RCNN or fast-RCNN; combine the model with the sliding window. In this document we will use the sliding window.

The idea is using a square window (frame) and sliding it on the image. With each position, we will have a cropped image. We use our model to predict the cropped image and choose the best window whose probability is highest.

However, there are problems we need to consider. First, how big is our window? Because in the image, the object can be either big or small, we must try different window sizes. Second, how many pixels do we need to move between two windows? The perfect solution for this question is 1 pixel, we will try all the cases. However, choose the step as 1 pixel will make our algorithm run very slow.

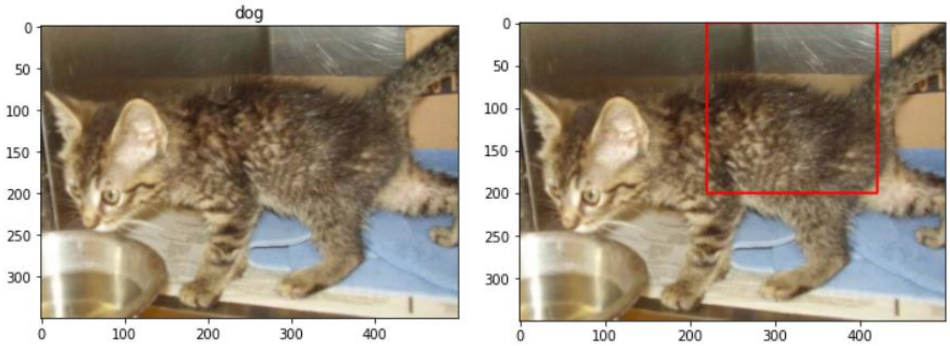
So in this problem, my team chose to try the window size from 100 to 240 and after each run we increase the window size by 20 pixels. For the step, we choose 10 pixels because moving 1 pixel does not make much difference.

2.2 RESULT



2.3 FURTHER STUDY

We can see that the detect work seems well but it still has many problems. First, the sliding window depends on the first prediction. If the prediction (cat or dog) is wrong, surely that detection will be wrong too:



Another problem is the performance, because we must try different window sizes and different positions for each one, the time to detect the object is very slow (Especially with the high resolution image).

So in future, to improve the performance we can try to use RCNN and fast R-CNN (Region-based Convolutional Neural Networks). This is one kind of neural network which is mainly used for object detection.

3 REFERENCES

- Introduction to Keras - Thomas Daniels - Code project - 22 Jun 2020
- An overview of gradient descent optimization algorithms - S.Ruder - 19 Jan 2016
- Basic about machine learning - Vũ Hữu Tiệp - 12 Jan 2017
- How to Classify Photos of Dogs and Cats (with 97% accuracy) - J.Brownlee - 17 May 2019
- ML Practicum: Image Classification - Google Machine Learning course - 25 Apr 2021
- Machine Learning Crash Course - Google course - 18 Aug 2021
- Very Deep Convolutional Networks for Large-Scale Image Recognition - K.Simonyan, A.Zisserman - 4 Sep 2014
- Fully-connected and Convolutional Nets - CS231n - Spring 2021 assignment