**ImageNet Classification with Deep Convolutional Neural Networks**

**Abstract**

*deep CNN to classify*: 1.2 million high-resolution (256x256) images in the ImageNet LSVRC-2010 (1000 different classes).

*CNN:* 60 million parameters and 650,000 neurons, 5 convolutional layers, some of which are followed by max-pooling layers, and 3 fully-connected layers with a final 1000-way softmax[[1]](#footnote-1).

*To make training faster:* **non-saturating neurons[[2]](#footnote-2)** and GPU.

*To reduce* ***overfitting****[[3]](#footnote-3)*: in the fully-connected layers - “dropout[[4]](#footnote-4)” proved to be very effective.

On the test data, we achieved top-1 and top-5 **error rates** of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

**Introduction**

*improve performance object detection*: collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting.

*Drawback:* 10.000 images (NORB, Caltech-101/256, and CIFAR-10/100). Simple recognition can be solved quite well if they are augmented with label-preserving transformations: MNIST digit-recognition task (<0.3%) **approaches** human performance ⇒ necessary to use much larger training sets.

*Engine:*LabelMe ~ hundreds of thousands of fully-segmented images, ImageNet > 15 million labeled high-resolution images in over 22,000 categories ⇒ need a model with a large learning capacity.

*Solve:* Convolutional neural networks (CNNs) constitute one such class of models [16, 11, 13, 18, 15, 22, 26].

***Contribution***:

- trained one of the largest convolutional neural networks to date on the subsets of ImageNet used in the ILSVRC-2010 and ILSVRC-2012 competitions and achieved by far the best results ever reported on these datasets.

- public a highly-optimized GPU implementation of 2D convolution and all the other operations inherent in training convolutional neural networks.

- improve its performance and reduce its training time

- used several effective techniques for preventing overfitting

- final network contains 5 convolutional and 3 fully-connected layers, and this depth seems to be important: we found that removing any convolutional layer (each of which contains no more than 1% of the model’s parameters) resulted in inferior performance.

**The Dataset**

ImageNet (>15 million labeled high-resolution images belonging to roughly 22,000 categories). The images were collected from the web and labeled by human labelers using Amazon’s Mechanical Turk crowd-sourcing tool.

Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. (*In all, there are roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images*).

ILSVRC-2010 is the only version of ILSVRC for which the test set labels are available, so this is the version on which we performed most of our experiments.

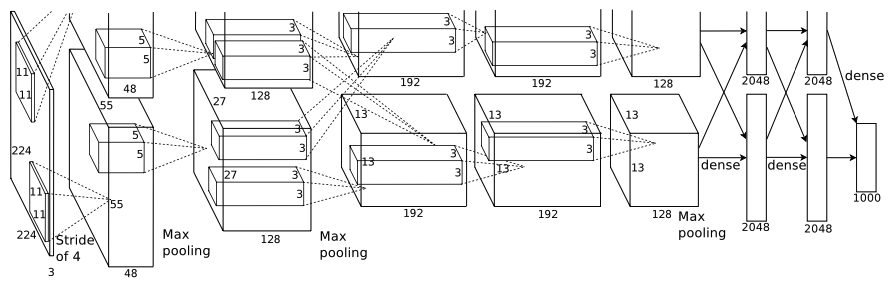
Since we also entered our model in the ILSVRC-2012 competition, in Section 6 we report our results on this version of the dataset as well, for which test set labels are unavailable.

On ImageNet, it is customary to report two error rates: top-1 and top-5[[5]](#footnote-5).

**In case variable-resolution images**: while Alexnet requires a constant input dimensionality. Therefore, we down-sampled the images to a fixed resolution of 256 × 256. Given a rectangular image, we first rescaled the image such that the shorter side was of length 256, and then cropped out the central 256×256 patch from the resulting image.



**The Architecture**

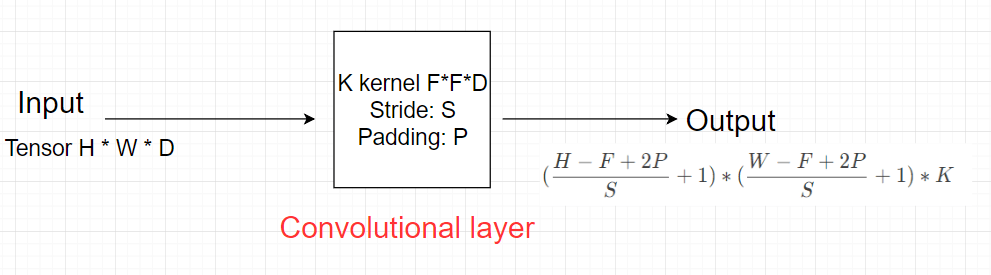


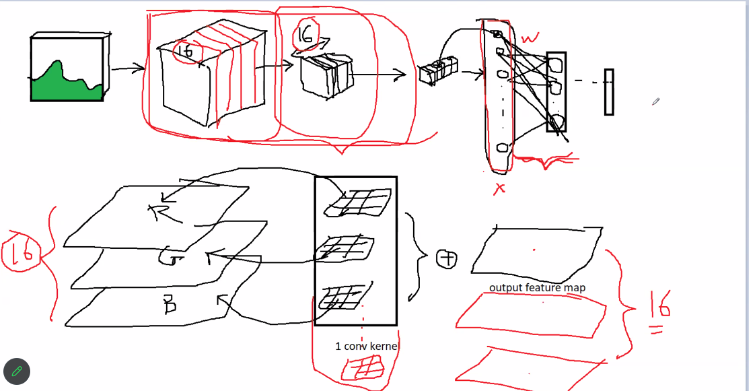
3 FC

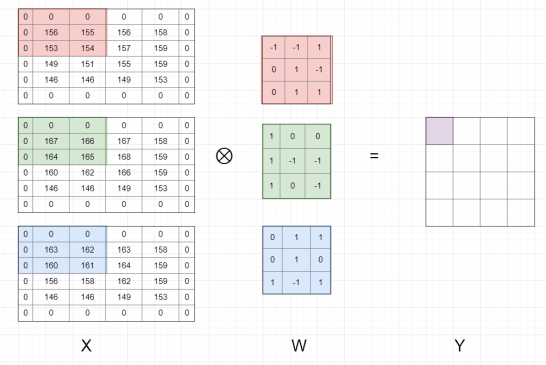
5 Conv

The net contains 8 layers with weights; the first five are convolutional and the remaining three are fully- connected. The output of the last fully-connected layer is fed to a 1000-way softmax (1000 classes).

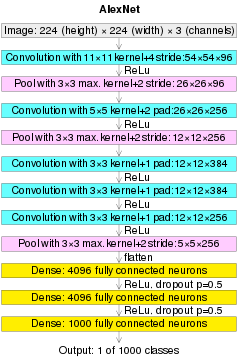
**1st convol:** In a CNN, the input is a tensor with shape: (number of inputs) × (input height) × (input width) × (input channels). The network’s input is 150,528-dimensional (224x224x3).







After passing through a convolutional layer (output), the image becomes abstracted to a feature map, also called an activation map, with shape:

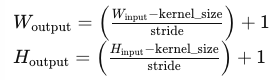


(AlexNet image size should be 227×227×3, instead of 224×224×3, so the math will come out right. The original paper said different numbers, but Andrej Karpathy, the head of computer vision at Tesla, said it should be 227×227×3 (he said Alex didn't describe why he put 224×224×3). The next convolution should be 11×11 with stride 4: 55×55×96 (instead of 54×54×96). It would be calculated, for example, as: [(input width 227 - kernel width 11) / stride 4] + 1 = [(227 - 11) / 4] + 1 = 55. Since the kernel output is the same length as width, its area is 55×55.)

The output of the convolutional layer will satisfy the activation function before becoming the input of the next convolutional layer.

After convolution with 96 kernel 3x3: 55x55x96 (290400 neurons)

Then, maxpooling (3x3, stride 2),



outpt: ? (

ReLU:

**convolutional 2:**

After convolution\_2 with 256 kernel 5x5: 26x26x256

ReLU:

Then, maxpooling (3x3, stride 2), outpt: (

**convolutional 3:**

After convolution\_3 with 384 kernel 3x3, pading 0: 13x13x384

ReLU:

**convolutional 4:**

After convolution\_4 with 384 kernel 3x3, pading 0: 13x13x384

ReLU:

**convolutional 5:**

After convolution\_4 with 384 kernel 3x3, pading 0: 13x13x256

ReLU:

Then, maxpooling (3x3, stride 2), outpt:

Flaten: 6\*6\*256 = 9216

Linear/Dense: 4096

Dropout = 0.5: 4096

Linear/Dense[[6]](#footnote-6): 1000

Model: "sequential\_4"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_23 (Conv2D) (None, 55, 55, 96) 34944

batch\_normalization\_20 (BatchNormalization) (None, 55, 55, 96) 384

max\_pooling2d\_12 (MaxPooling2D) (None, 27, 27, 96) 0

conv2d\_24 (Conv2D) (None, 27, 27, 256) 614656

batch\_normalization\_21 (BatchNormalization) (None, 27, 27, 256) 1024

max\_pooling2d\_13 (MaxPooling2D) (None, 13, 13, 256) 0

conv2d\_25 (Conv2D) (None, 13, 13, 384) 885120

batch\_normalization\_22 (BatchNormalization) (None, 13, 13, 384) 1536

conv2d\_26 (Conv2D) (None, 13, 13, 384) 1327488

batch\_normalization\_23 (BatchNormalization) (None, 13, 13, 384) 1536

conv2d\_27 (Conv2D) (None, 13, 13, 256) 884992

batch\_normalization\_24 (BatchNormalization) (None, 13, 13, 256) 1024

max\_pooling2d\_14 (MaxPooling2D) (None, 6, 6, 256) 0

flatten\_4 (Flatten) (None, 9216) 0

dense\_8 (Dense) (None, 4096) 37752832

dropout\_4 (Dropout) (None, 4096) 0

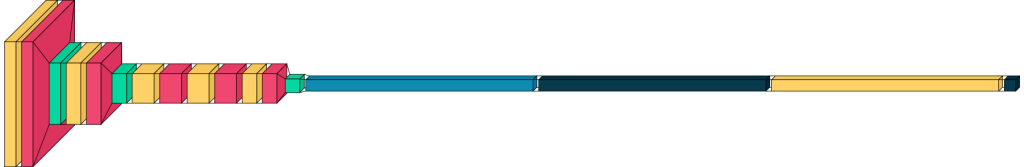
dense\_9 (Dense) (None, 1000) 4097000

=================================================================

Total params: 41,546,506

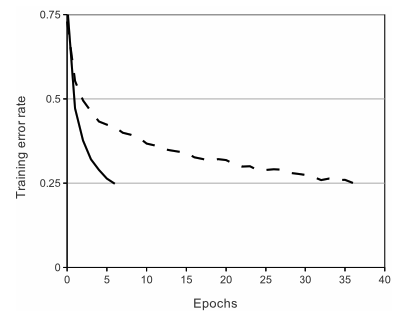
Trainable params: 41,543,754

Non-trainable params: 2,752



**ReLU Nonlinearity**

The standard way to model a neuron’s output f as a function of its input x is with f(x) = tanh(x) or (sigmoid) f(x) = (1 + e−x)−1. In terms of training time with gradient descent, these saturating nonlinearities are much slower than the non-saturating nonlinearity f(x) = max(0,x). Following Nair and Hinton, we refer to neurons with this nonlinearity as Rectified Linear Units (ReLUs).

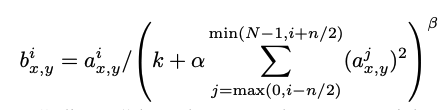


*A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons (dashed line).*

We are not the first to consider alternatives to traditional neuron models in CNNs. For example, Jarrett et al. claim that the nonlinearity f (x) = |tanh(x)| works particularly well with their type of contrast normalization followed by local average pooling on the Caltech-101 dataset.

**Local Response Normalization[[7]](#footnote-7)**

ReLUs have the desirable property that they do not require input normalization to prevent them from saturating.



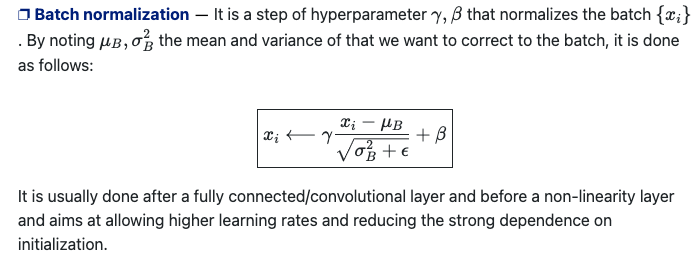
: the activity of a neuron computed by applying kernel i at position (x, y) and then applying the ReLU nonlinearity.

n “adjacent” kernel maps

N is the total number of kernels in the layer

The constants k, n, α, and β are hyper-parameters

Response normalization reduces our top-1 and top-5 error rates by 1.4% and 1.2%, respectively.



**Overlapping Pooling:**

Pooling layers in CNNs summarize the outputs of neighboring groups of neurons in the same kernel map. Traditionally, the neighborhoods summarized by adjacent pooling units do not overlap.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| s=3, z=3 | | | | | |  | s=2, z=3 | | | | | |

To be more precise, a pooling layer can be thought of as consisting of a grid of pooling units spaced s pixels apart, each summarizing a neighborhood of size z × z centered at the location of the pooling unit.

If we set s = z: traditional local pooling as commonly employed in CNNs.

If we set s < z: overlapping pooling.

This scheme reduces the top-1 and top-5 error rates by 0.4% and 0.3%, respectively, as compared with the non-overlapping scheme s = 2, z = 2, which produces output of equivalent dimensions.

**Reducing Overfitting**

The first form: image translations and horizontal reflections. We do this by extracting random 224 × 224 patches (and their horizontal reflections) from the 256×256 images.

The second form perform PCA on the set of RGB pixel values throughout the ImageNet training set.

**Dropout**

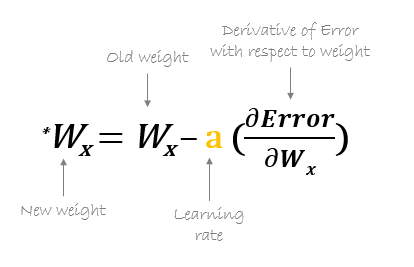
consists of setting to zero the output of each hidden neuron with probability 0.5. The neurons which are “dropped out” in this way do not contribute to the forward pass and do not participate in back- propagation.

**Details of learning**

batch: 128

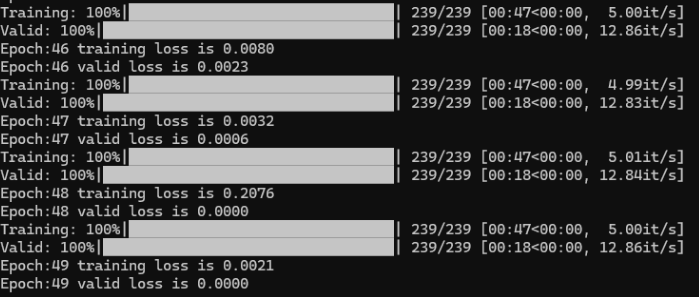
w: 0.9

lr=0.0005



from torch import nn  
  
# Image Classifier Neural Network  
class ImageClassifier(nn.Module):  
 def \_\_init\_\_(self,num\_class):  
 super().\_\_init\_\_()  
 self.model = nn.Sequential(  
 #### Convolutional Layers ####  
 #input: 224\*224\*3  
 # Layer 1:  
 nn.Conv2d(3, 96, kernel\_size=(11, 11), stride=(4, 4)), # Change 1 to 3 for RGB images: output = 32  
 #output: W = (224-11+2\*0)/4 + 1=54, H = (224-11+2\*0)/4 + 1=54  
 nn.ReLU(), # output:96, W=H=54 do co cung chieu dai moi canh sau conv  
 nn.BatchNorm2d(96),  
 nn.MaxPool2d(2, 2),  
 # after pooling: ((W=H=54-kernel=2)/stride=2) + 1= 27; nn.Flatten(), nn.Linear(96\*(27)\*(27), 53),  
  
 # Layer 2  
 nn.Conv2d(96, 256, kernel\_size=(5, 5), padding=(2, 2)), # input: 96, output: 256  
 #output: W=(27 - 5 + 2 \* 2) / 1 + 1 = 26, H=(27 - 5 + 2 \* 2) / 1 + 1 = 26  
 nn.ReLU(), # output:256, W=H=26 do co cung chieu dai moi canh sau conv  
 nn.BatchNorm2d(256),  
 nn.MaxPool2d(3, 2),  
 # after pooling: ((W=H=26-kernel=3)/stride=2)+1 = 12; nn.Flatten(), nn.Linear(256\*(12)\*(12), 53),  
  
 # Layer 3  
 nn.Conv2d(256, 384, kernel\_size=(3, 3), padding=(1, 1)), # input: 256, output: 384  
 # output: W=(12 - 3 + 2 \* 1) / 1 + 1 = 12, W=(12 - 3 + 2 \* 1) / 1 + 1 = 12  
 nn.ReLU(), # output:384, W=H=12 do co cung chieu dai moi canh sau conv  
 nn.BatchNorm2d(384),  
  
 # Layer 4  
 nn.Conv2d(384, 384, kernel\_size=(3, 3), padding=(1, 1)), # input: 384, output: 384  
 # output: W=(12 - 3 + 2 \* 1) / 1 + 1 = 12, W=(12 - 3 + 2 \* 1) / 1 + 1 = 12  
 nn.ReLU(), # output:384, W=H=12 do co cung chieu dai moi canh sau conv  
 nn.BatchNorm2d(384),  
  
 # Layer 5  
 nn.Conv2d(384, 256, kernel\_size=(3, 3), padding=(1, 1)), # input: 384, output: 256  
 # output: W=(12 - 3 + 2 \* 1) / 1 + 1 = 12, W=(12 - 3 + 2 \* 1) / 1 + 1 = 12  
 nn.ReLU(), # output:256, W=H=12 do co cung chieu dai moi canh sau conv  
 nn.BatchNorm2d(256),  
 nn.MaxPool2d(3, 2),  
 # after pooling: ((W=H=12-kernel=3)/stride=2)+1 = 6; nn.Flatten(), nn.Linear(256\*(6)\*(6), 53),  
  
 #### Fully-Connected Layer ####  
 nn.Flatten(),  
 nn.Dropout(0.5),  
 nn.Linear(256\*6 \*6, num\_class), #tinh sao ra bang 6??????  
 )  
 #Truyen tham so vao  
 self.num\_class = num\_class  
  
 def forward(self, x):  
 return self.model(x)

**TRAIN: 50 Epoch**

****



01.jpg: ace of clubs

02.jpg: ace of diamonds

03.jpg: ace of hearts

04.jpg: ace of spades

05.jpg: two of clubs

06.jpg: two of diamonds

07.jpg: two of hearts

08.jpg: two of spades

09.jpg: three of clubs

10.jpg: three of diamonds

11.jpg: three of hearts

12.jpg: three of spades

13.jpg: four of clubs

14.jpg: four of diamonds

15.jpg: four of hearts

16.jpg: four of spades

17.jpg: five of clubs

18.jpg: five of diamonds

19.jpg: five of hearts

20.jpg: five of spades

21.jpg: six of diamonds

22.jpg: six of hearts

24.jpg: six of spades

25.jpg: seven of clubs

26.jpg: seven of diamonds

27.jpg: seven of hearts

28.jpg: seven of spades

29.jpg: eight of clubs

30.jpg: eight of diamonds

31.jpg: eight of hearts

32.jpg: eight of spades

33.jpg: nine of clubs

34.jpg: nine of spades

35.jpg: nine of diamonds

36.jpg: nine of hearts

37.jpg: ten of clubs

38.jpg: ten of diamonds

39.jpg: ten of hearts

40.jpg: ten of spades

41.jpg: jack of clubs

42.jpg: jack of diamonds

43.jpg: jack of hearts

44.jpg: jack of spades

45.jpg: queen of clubs

46.jpg: queen of diamonds

47.jpg: two of hearts

48.jpg: queen of spades

49.jpg: king of clubs

50.jpg: king of diamonds

51.jpg: king of hearts

52.jpg: king of spades

53.jpg: joker

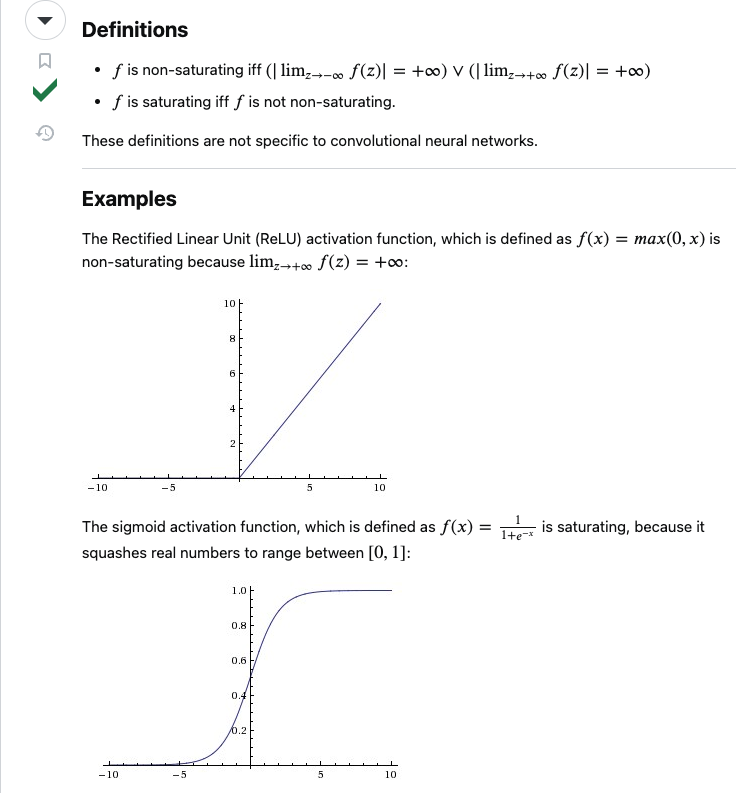
54.jpg: joker

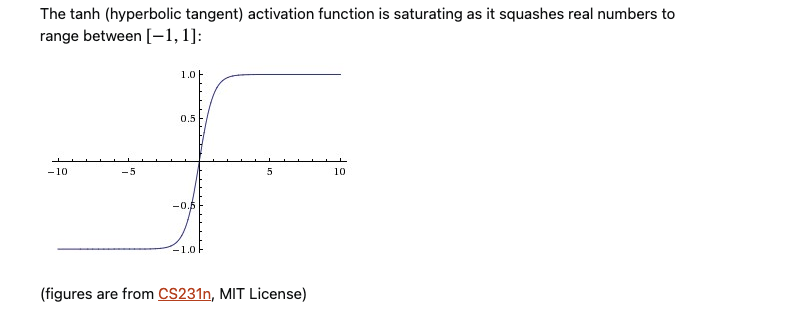
55.jpg: joker

56.jpg: joker

57.jpg: joker

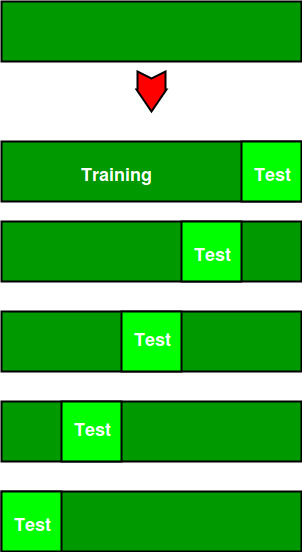
**non-saturating neurons (neuron không bão hoà)**





**overfitting**

In machine learning, overfitting occurs when an algorithm fits too closely or even exactly to its training data, resulting in a model that can’t make accurate predictions or conclusions from any data other than the training data.

*How to detect overfit models?*

K-fold cross-validation is one of the most popular techniques to assess accuracy of the model. In k-folds cross-validation, data is split into k equally sized subsets, which are also called “folds.” One of the k-folds will act as the test set, also known as the holdout set or validation set, and the remaining folds will train the model. This process repeats until each of the fold has acted as a holdout fold. After each evaluation, a score is retained and when all iterations have completed, the scores are averaged to assess the performance of the overall model.

(https://www.geeksforgeeks.org/cross-validation-machine-learning/). The diagram below shows an example of the training subsets and evaluation subsets generated in k-fold cross-validation. Here, we have total 25 instances. In first iteration we use the first 20 percent of data for evaluation, and the remaining 80 percent for training ([1-5] testing and [5-25] training) while in the second iteration we use the second subset of 20 percent for evaluation, and the remaining three subsets of the data for training ([5-10] testing and [1-5 and 10-25] training), and so on.

*How to avoid overfitting?*

**Early stopping**: As we mentioned earlier, this method seeks to pause training before the model starts learning the noise within the model. This approach risks halting the training process too soon, leading to the opposite problem of underfitting. Finding the “sweet spot” between underfitting and overfitting is the ultimate goal here.

**Train with more data**: Expanding the training set to include more data can increase the accuracy of the model by providing more opportunities to parse out the dominant relationship among the input and output variables. That said, this is a more effective method when clean, relevant data is injected into the model. Otherwise, you could just continue to add more complexity to the model, causing it to overfit.

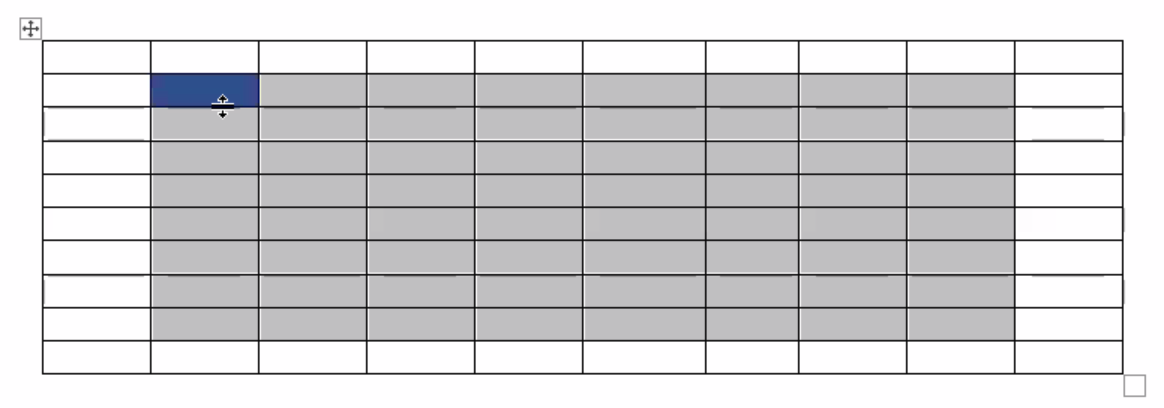
**Data augmentation**: While it is better to inject clean, relevant data into your training data, sometimes noisy data is added to make a model more stable. However, this method should be done sparingly.

**Feature selection**: When you build a model, you’ll have a number of parameters or features that are used to predict a given outcome, but many times, these features can be redundant to others. Feature selection is the process of identifying the most important ones within the training data and then eliminating the irrelevant or redundant ones. This is commonly mistaken for dimensionality reduction, but it is different. However, both methods help to simplify your model to establish the dominant trend in the data.

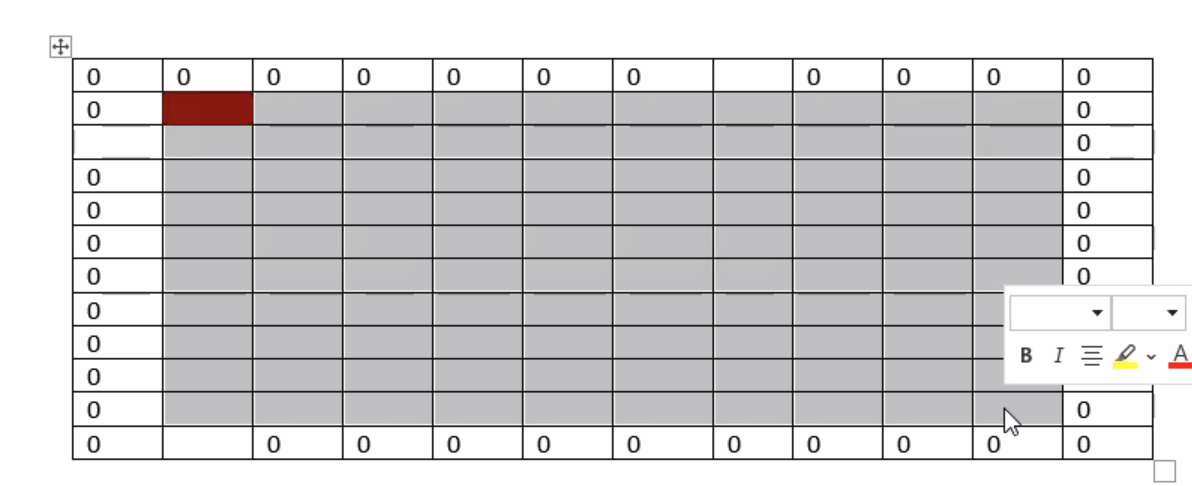
**Regularization**: If overfitting occurs when a model is too complex, it makes sense for us to reduce the number of features. But what if we don’t know which inputs to eliminate during the feature selection process? If we don’t know which features to remove from our model, regularization methods can be particularly helpful. Regularization applies a “penalty” to the input parameters with the larger coefficients, which subsequently limits the amount of variance in the model. While there are a number of regularization methods, such as lasso regularization, ridge regression and dropout, they all seek to identify and reduce the noise within the data.

**Ensemble methods**: Ensemble learning methods are made up of a set of classifiers—e.g. decision trees—and their predictions are aggregated to identify the most popular result. The most well-known ensemble methods are bagging and boosting. In bagging, a random sample of data in a training set is selected with replacement—meaning that the individual data points can be chosen more than once. After several data samples are generated, these models are then trained independently, and depending on the type of task—i.e. regression or classification—the average or majority of those predictions yield a more accurate estimate. This is commonly used to reduce variance within a noisy dataset.

Padding = 0, -2 pixel (10x10) -> 8x8



padding 1 (add boundary pixel 0): 10x10 output



**padding = kernel\_size/2 (floor)**

**kernel = trainable weight**

1. softmax: The output of the model is a vector with 1000 elements. The ith element of the vector represents the probability that the image belongs to the ith class. Therefore, the sum of the elements in the vector is 1. [↑](#footnote-ref-1)
2. ReLU (Rectified Linear Unit): https://stats.stackexchange.com/questions/174295/what-does-the-term-saturating-nonlinearities-mean [↑](#footnote-ref-2)
3. [What is Overfitting? | IBM](https://www.ibm.com/topics/overfitting) [↑](#footnote-ref-3)
4. reduce trainable weight, turn off neurons randomly => prevent over fiting [↑](#footnote-ref-4)
5. where the top-5 error rate is the fraction of test images for which the correct label is **not among** the five labels considered **most probable** by the model. [↑](#footnote-ref-5)
6. https://stackoverflow.com/questions/66626700/difference-between-tensorflows-tf-keras-layers-dense-and-pytorchs-torch-nn-lin [↑](#footnote-ref-6)
7. Another: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks [↑](#footnote-ref-7)