CNN for Abnormality detection in mammography

Stefano Fiori

April 2018

1 Introduction

Making screening mammography more effective has been on radiology's to-do list for several years. One of the most commonly used tools to improve diagnostic sensitivity and specificity has been computer-aided detection. Over the past decade, technology has made significant in-roads, but questions remain about how certain features should be used.

When paired with mammography, CAD helps radiologists identify any abnormal areas in breast tissue. To date, it's been used mainly with 2D imaging, but work is underway to extend its utility to 3D imaging, as well.

Despite its maturation and widespread use, however, there isn't yet a consensus about whether CAD is truly useful. In fact, a 2011 study published in the Journal of the National Cancer Institute found that CAD, when used in community radiology practices, can reduce specificity and provides no improvement to cancer detection.

2 The State of the art

2.1 CAD Today

CAD has evolved over the past decade to be a nearly indispensable technology for the practicing radiologist.

When used in conjunction with mammograms, CAD identifies and marks suspicious lesions and masses in breast tissue that radiologists could miss. The tool's sensitivity substitutes for the human eye's inability to focus on every pixel in an image. Consequently, CAD catches the sub-millimeter calcifications radiologists can miss.

CAD's biggest benefit comes with correctly and objectively identifying and measuring breast density. This ability is significant because dense breast tissue can easily hide malignancies, leaving them undiscovered until they are untreatable.

CAD's precision is invaluable. Not only does the accuracy minimize errors, but it also reduces needless fear among patients and unnecessary supplemental imaging.

Nowadays, CAD systems utilize image analysis and deep learning technologies. Information from thousands of mammography images are incorporated into the algorithms, enabling the product to distinguish between characteristics of cancerous and normal tissue) These CAD systems clearly identify lesion candidates in both 2D mammograms and 3D tomosynthesis volumes.

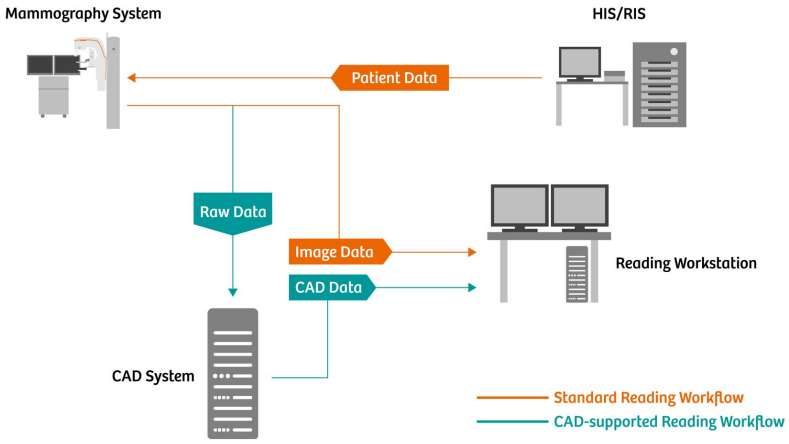


Figure 1: CAD Workflow

This is what a CAD workflow in screening might look like:

2.2 Sources and papers

A great contribution to my experience come from the article:
Classification of breast cancer histology images using incremental boosting convolution networks(1). Even if I didn't follow directly their idea, it given me a lot of help, I wonder to follow their implementation one day. Another main source has been the *Keras Documentation*(2). Here we can find what we are really looking for. I spent a lot of time by looking for some information everywhere, and at the end it was in the online doc of keras(feature extraction, ensemle, etc.)

3 Project

3.1 Scratch CNN experience

When trying to tune the Convolution Neural Network layers, I was gone through several empirical rules.

3.1.1 My first time

As first, I tried with a big model, with an already augmented data-set.
The main problem with my approach was that I was trying to obtain a good model, by starting with a huge problem. Indeed I had already applied data-

augmentation at this step, and every change to my model resulted in no difference on output accuracy.

Lesson Even if the approach was wrong I realized two empirical rules:

- activation function of the neurons: relu. Because, after the initial research, relu is a really good function for these reasons, tested by a lot of researchers in the Computer Vision field.
- add a MaxPooling layer after each Conv layer: because really decrease the size of the input to next layer in the net.

3.1.2 Second try

At this step I decided to start from a really simple model with only three layer: first one was composed by 32 feature, second 64, and third 128. With no data-augmentation I reached an accuracy of **90%** with no over-fit on validation set. I start to tune this simple model, and each time I tried to tune my simple model, the results were worse.

Lesson Meanwhile I was learning other notable lessons:

1. **fundamental**, for all other steps, is to have a **good data-set**: this means to have a big enough set, where data are pre-processed to work well with our network, remember the GIGO concept(3). In this step, there is also data augmentation. I spent a lot of time trying to improve my model, and when I finally tried to augment my train data I resolved a lot of my problems.
2. once we are sure data-set is good, it's good practice to start from a small model. If we notice weird behaviours in the training procedure, such as scattering results, starving accuracy, or increasing loss on the validation set, this means we have to modify the model to obtain better results.
3. try to increase the number of neurons on later layers. This typically yields to more stable results on training procedure, although sometimes it introduces overfitting. When we notice overfitting is better to return to the previous version, or if the overfitting is not so fast trying to introduce some regularization technique
4. Dropout layers, batch normalization layers, weight regularization, are all good for fight overfitting. I usually try first dropout technique. But remember that if the overfitting is too much evidence, it's better to change the model architecture rather than fight it with this technique.

Doubts But why does the model get worse every time I was changing its? Is it better to have a really simple model, with only a small data-set? Why everyone struggle to have a big, efficient model, when the answer lies in a small model, with few data as input?

Answers there was one last problem, I did notice really too late. My accuracy on train set was too low. It's good practice to make the accuracy on the training set reach values close to 100%; indeed, having a low accuracy on training set means to have either a low net capacity or a low learning rate or both. To test if my model had a low capacity i tried to run the train for 200 epochs, to see if it would ever reach high values on train accuracy.

(a) Accuracy over training epochs

(b) Accuracy over training epochs

As we can see after about 50 epochs the accuracy on training set starts to oscillate and never exceed 90%.

3.1.3 Third try

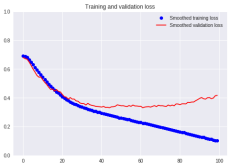
By only adding another earlier layer, of the same capacity of firsts layer, we reach **95% of accuracy** on training set

But this second model over-fit as well, if we augment our train set. At this time, I understand what *adding layer and adding neurons really means.* **The ideal model is one that stands right at the border between under-fitting and over-fitting; between under-capacity and over-capacity** // The figures below show how the loss is affected when doubles a layer.

3.2 Final Model

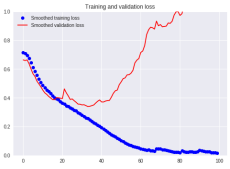
```
model = models.Sequential()
# LAYER 1
model.add(layers.Conv2D(32, (3, 3), activation='relu',
        input_shape=(IMAGE_SIZE[0], IMAGE_SIZE[1], 3)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
# LAYER 2
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

(a) double first layer of 32 feature



(b) double second layer of 64 feature

(c) double third layer of 128 feature



(d) a further layer (4th) of 256 features

Figure 3: loss behavior by doubling layers

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
# LAYER 3
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
#  LAYER 4
model.add(layers.Dropout(rate=0.10))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
# LAYER 5
model.add(layers.Dropout(rate=0.10))
model.add(layers.Conv2D(512, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
# DENSE
model.add(layers.Flatten())
model.add(layers.Dropout(rate=0.25))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dropout(rate=0.25))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dropout(rate=0.25))
# OUTPUT
model.add(layers.Dense(NUM_CLASSES, activation='softmax',
    name='softmax'))
```

3.2.1 Hyperparameters

The learning rate Learning rate points out the fraction of the step in the decreasing direction when updating the weights. By changing this parameter we can have mainly two behaviors.

High learning rate If we choose an high learning rate is highly probable the error doesn't decrease, because the learning algorithm overstep the solution or, even worse, goes off in the wrong direction (in such a case we will see the accuracy decrease). So whether our model is good (has enough capacity, correct structure, etc), and the dataset is correctly pre-processed, but the loss is not decreasing, this is likely due to an high learning rate.

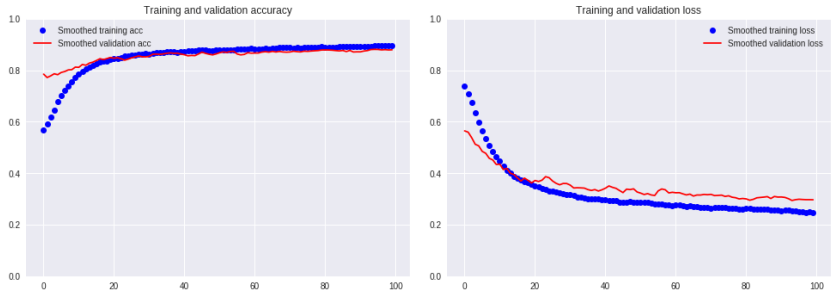
Low learning rate On the other hand, very small steps may go in the correct direction, but may require too many iteration. A small low learning results in a lot of epochs with a tiny increase of the performance each step; this behavior is not good especially when we are trying to find out a good model. Indeed imagine we modify our model, then train and look at results, than again: modify, train, compare results, and so on. Whit a slow learning rate this steps are really slows. My advice is to start with an high learning rate, and decrease it until we see in the fit phase, that the loss function is starting decrease. At this point if the loss function is scattered among the epochs we can further decrease the learning a little bit until the loss function is more stable.

3.3 Pre-trained CNN experience

Starting from an already trained Model has brought better early results than scratch CNN. The first architecture was not so good compared to the last ones. All that I have already learned from scratch CNN experience was still valid at this point. The questions on which I focused on a pre-trained model was:

- which layers of the pre-trained model reuse?
- which layers fine-tune? which do layers freeze?
- what if I add some convolutional layer on top?

After some research on the web, I realized VGG16 model was good for my scope. So starting from a VGG16 without the top Dense layer, first try was to add a Dense layer on top and see the results.



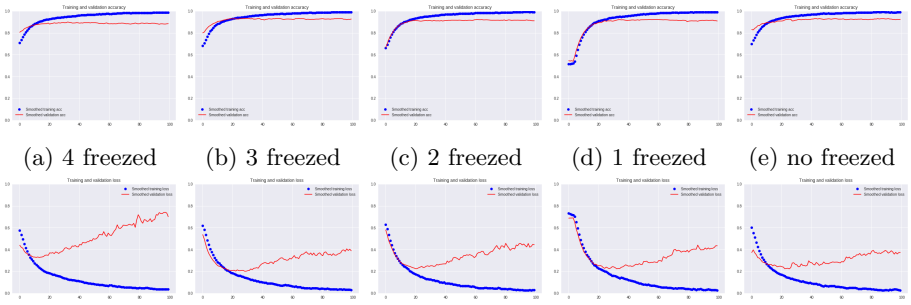
(a) Caption (b) Caption

As we can see good results are reached faster than the scratch experience. We are only training a layer of 1024 neurons on top of VGG16 that's why we reach good results so fast. Besides the speed, we can see is possible to reach an accuracy of about **86%**. It is a good result until now.

3.3.1 Freezing

The VGG16 model is divided in block; each block have three or more hidden layers of the same size. Between a block and another one, there is a max pooling layer, that halves the output of the previous block.

Freezing firsts layers My curiosity here was: *How the behaviour changes by freezing different blocks?*.
The graphs below show the results from such an experience.



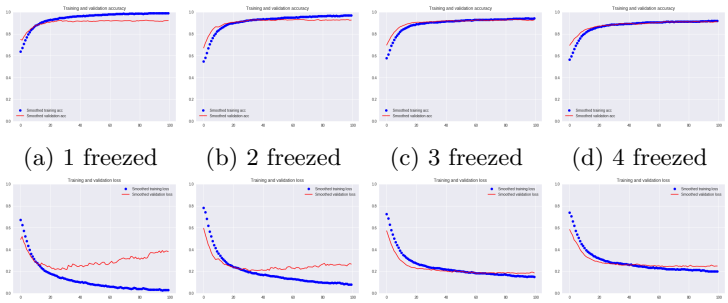
(a) 4 freezed (b) 3 freezed (c) 2 freezed (d) 1 freezed (e) no freezed

(f) 4 freezed (g) 3 freezed (h) 2 freezed (i) 1 freezed (j) no freezed

Figure 5: accuracy and loss functions by freezing the **firsts N** blocks

As we ca see, by increasing the number of *free* blocks is possible to obtain best results, but there is a limit. Indeed freezing only the first layer is better than no freeze any layer at all. In such a tests we can see the net has start to over-fit, but I decide to deal with this problem later. At this step I have some others doubt to remove.

Freezing lasts layers *What if I freeze the lasts layers rather than the first ones?*



(a) 1 freezed (b) 2 freezed (c) 3 freezed (d) 4 freezed

(e) 1 freezed (f) 2 freezed (g) 3 freezed (h) 4 freezed

Figure 6: accuracy and loss functions by freezing the **lass N** blocks

From the latter experiment we can see that the net is less sensitive to freezing last layers. The previous experiments suggest us to leave free(i.e. to fine tune) layers from two to five. Then I will try to remove the last layer from the VGG16 and freeze only the first layer, in order to fit the net capacity to our problem. If this bring us a good result, it will likely be the final net.

3.3.2 Final Model

My final model is a VGG16 without the last block of convolutional layer. On top of it there are two layer of 512 fully connected neurons, among which there are Dropout layer with a 0.4 probability.

The code:

```
# freeze firsts layers
for layer in vgg16.layers[:FREEZE_LAYERS]:
    layer.trainable = False
for layer in vgg16.layers[FREEZE_LAYERS:]:
    layer.trainable = True

net_final = models.Sequential()
# drop last layers
last_net_layer = len(vgg16.layers)-DROP_LAYERS
for layer in vgg16.layers[:last_net_layer]:
    net_final.add(layer)

net_final.add(layers.Flatten())
net_final.add(layers.Dropout(0.4))
net_final.add(layers.Dense(512, activation="relu"))
net_final.add(layers.Dropout(0.4))
net_final.add(layers.Dense(512, activation="relu"))
net_final.add(layers.Dropout(0.4))
net_final.add(layers.Dense(NUM_CLASSES, activation='softmax',
            name='softmax'))
```

Examples of hyperparameters are:

Number of layers and hidden units: affects the capacity of the model; Learning rate: determines the step size in learning procedure; Learning rate decay strategy; Mini-batch size; Number of epochs of training and stop criterion; Weights initialization strategy; Preprocessing strategy; and possibly many others

3.4 Ensemble

Ensembling is the technique to combine predictions, coming from different models, in order to obtain a more accurate prediction. Indeed, combining more than one model lower the error rate on predictions. **Keras** provide the programmer with layers that can merge outputs coming from models in different ways. We can simply concatenate outputs or having more complex computation on outputs. For my ensemble i decide to use the ***Average*** *Keras merge Layer.*

The code:

```
# ENSEMBLE
```

```
from keras import Input
from keras.models import Model
from keras.layers import Average

input_tensor = Input(shape=(IMAGE_SIZE[0],IMAGE_SIZE[1],3),
                     name='input_tensor', dtype='float32')
output_tensors = []
for i,m in enumerate(mymodels):
    m.name = MODELS[i]['name']
    m.trainable = False
    output_tensor = m(input_tensor)
    output_tensors.append(output_tensor)

output_tensor = Average()(output_tensors)
ensemble_model = models.Model(input_tensor, output_tensor,
                              name='ensemble')
```

References

[1] S.-W. Duc My Vo, Ngoc-Quang Nguyen, "Classification of breast cancer histology images using incremental boosting convolution networks," *Information Sciences*, vol. 482, pp. 123–138, 2019.

[2] Keras, "Keras online documentation." [Online]. Available: https://keras.io/

[3] Wikipedia contributors, "Garbage in, garbage out — Wikipedia, the free encyclopedia," 2004, [Online; accessed 22-July-2004]. [Online]. Available: https://en.wikipedia.org/wiki/Garbage_in,_garbage_out