Convolution Neural Networks

Topics Covered

1. Efficient Convolution Algorithms
2. Dropout Optimization and Batch Normalization
3. Regularization For CNN
4. Interpreting And Explainability With CNN
5. Best Practices With CNN
6. Summary

Efficient Convolution Algorithms

In some problem settings, performing convolution as pointwise multiplication in the frequency domain can provide a speed up as compared to direct computation. This is a result from the property of convolution:

$$F\{f \star g\} = F\{f\} \times F\{g\}$$

Convolution in the source domain is multiplication in the frequency domain. F is the transformation operation

$$f \star g = F^{-1}\{F\{f \star g\}\} = F^{-1}\{F\{f\} \times F\{g\}\}$$

When a **d**-dimensional kernel can be broken into the outer product of **d** vectors, the kernel is said to be separable. The corresponding convolution operations are more efficient when implemented as **d** 1-dimensional convolutions rather than a direct **d**-dimensional convolution. Note however, it may not always be possible to express a kernel as an outer product of lower dimensional kernels.

This is not to be confused with **depthwise separable convolution**.This method restricts convolution kernels to operate on only one input channel at a time followed by 1x1 convolutions on all channels of the intermediate output.

Devising faster ways of performing convolution or approximate convolution without harming the accuracy of the model is an active area of research.

Dropout Optimization and Batch Normalization

Dropout Optimization

Dropout is actually a form of regularization that aims to help prevent overfitting by increasing testing accuracy, perhaps at the expense of training accuracy. For each mini-batch in our training set, dropout layers, with probability *p*, randomly disconnect inputs from the preceding layer to the next layer in the network architecture.

Figure visualizes this concept where we randomly disconnect with probability *p* = 0.5 the connections between two FC(fully connected) layers for a given mini-batch. Again, notice how half of the connections are severed for this mini-batch. After the forward and backward pass are computed for the mini-batch, we re-connect the dropped connections, and then sample another set of connections to drop.

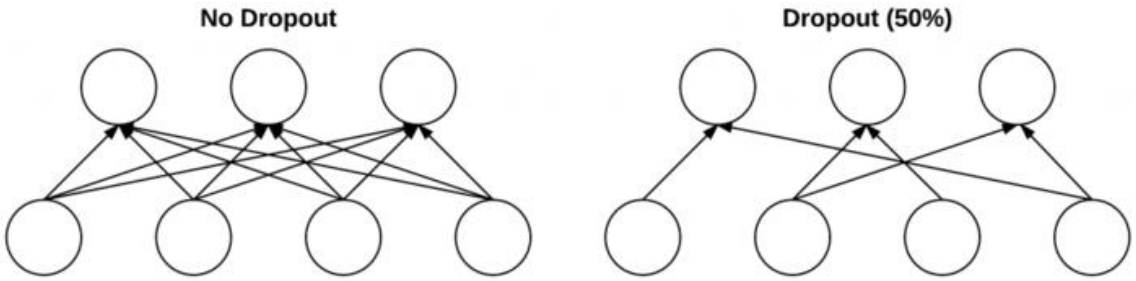


Figure *Left:* Two layers of a neural network that are fully connected with no dropout. *Right:* The same two layers after dropping 50% of the connections.

The reason we apply dropout is to reduce overfitting by explicitly altering the network architecture at training time. Randomly dropping connections ensures that no single node in the network is responsible for "activating" when presented with a given pattern. Instead, dropout ensures there are multiple, redundant nodes that will activate when presented with similar inputs — this, in turn, helps our model to generalize.

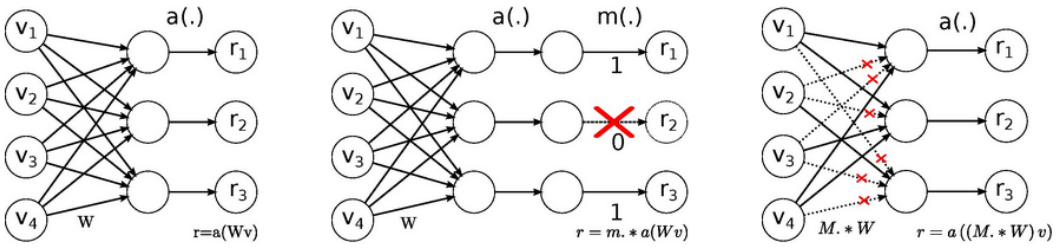
It is most common to place dropout layers with *p* = 0.5 *in-between* FC layers of an architecture where the final FC layer is assumed to be our softmax classifier:
However, we may also apply dropout with smaller Convolutional Neural Networks (CNNs) and Layer Types

... CONV => RELU => POOL => FC => DO => FC => DO => FC

Probabilities (i.e., *p* = 0.10–0.25) in earlier layers of the network as well (normally following a down sampling operation, either via max pooling or convolution).

Drop Connect

A generalization of Dropout for regularizing large fully-connected layers within neural networks. When training with Dropout, a randomly selected subset of activations are set to zero within each layer. DropConnect instead sets a randomly selected sub-set of weights within the network to zero. Each unit thus receives input from a random subset of units in the previous layer. We derive a bound on the generalization performance of both Dropout and DropConnect. We then evaluate DropConnect on a range of datasets, comparing to Dropout, and show state-of-the-art results on several image recognition benchmarks by aggregating multiple DropConnect-trained models.



No-Drop Network DropOut Network DropConnect Network

Batch Normalization

First introduced by *Ioffe and Szegedy in their 2015 paper,* Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, batch normalization layers (or BN for short), as the name suggests, are used to normalize the activations of a given input volume before passing it into the next layer in the network.

If we consider x to be our mini-batch of activations, then we can compute the normalized $\hat{x}$ via the following equation:

$$\hat{x_i} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}$$

During training, we compute the μ_β and σ_β over each mini-batch β, where:

$$\mu_\beta = \frac{1}{M}\sum_{i=1}^{m} x_i \qquad \sigma_\beta^2 = \frac{1}{M}\sum_{i=1}^{m}(x_i - \mu_\beta)^2$$

We set ϵ equal to a small positive value such as 1e-7 to avoid dividing by zero. Applying this equation implies that the activations leaving a batch normalization layer will have approximately zero mean and unit variance (i.e., zero-centered).

Batch normalization has been shown to be extremely effective at reducing the number of epochs it takes to train a neural network. Batch normalization also has the added benefit of helping "stabilize" training, allowing for a larger variety of learning rates and regularization strengths. Using batch normalization doesn't alleviate the need to tune these parameters of course, but it *will* make your life easier by making learning rate and regularization less volatile and more straightforward to tune. You'll also tend to notice lower final loss *and a* more stable loss curve when using batch normalization in your networks.

At *testing* time, we replace the mini-batch μ_β and σ_β with *running averages* of μ_β and σ_β computed during the training process. This ensures that we can pass images through our network and still obtain accurate predictions without being biased by the μ_β and σ_β from the final mini-batch passed through the network at training time.

The biggest drawback of batch normalization is that it can actually slow down the wall time it takes to train your network (even though you'll need fewer epochs to obtain reasonable accuracy) by 2-3x due to the computation of per-batch statistics and normalization.

That said, we recommend using batch normalization in nearly every situation as it does make a significant difference. Applying batch normalization to our network architectures can help us prevent overfitting and allows us to obtain significantly higher classification accuracy in fewer epochs compared to the same network architecture *without* batch normalization.

According to the original paper by Ioffe and Szegedy**,** they placed their batch normalization (BN) *before* the activation:

We add the BN transform immediately before the nonlinearity, by normalizing $x = Wu + b$.

Using this scheme, a network architecture utilizing batch normalization would look like this:

Convolutional Neural Networks (CNNs) and Layer Types

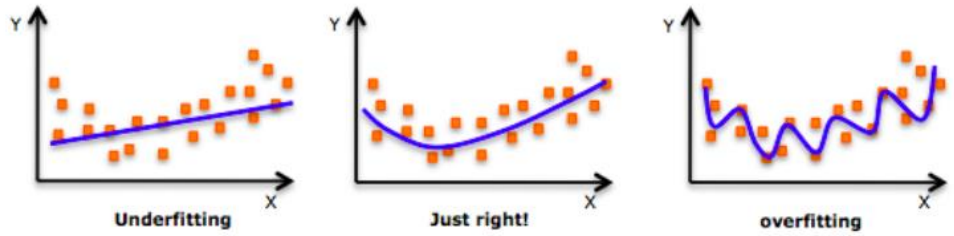
INPUT => CONV => BN => RELU ...

However, this view of batch normalization doesn't make sense from a statistical point of view. In this context, a BN layer is normalizing the distribution of features coming out of a CONV layer. Some of these features may be negative, in which they will be clamped (i.e., set to zero) by a nonlinear activation function such as ReLU.

Placing the BN after the RELU yields slightly higher accuracy and lower loss. That said, take note of the word *"nearly"* — there have been a *very small* number of situations where placing the BN before the activation worked better, which implies that you should default to placing the BN after the activation, but may want to dedicate (at most) one experiment to placing the BN before the activation and noting the results.

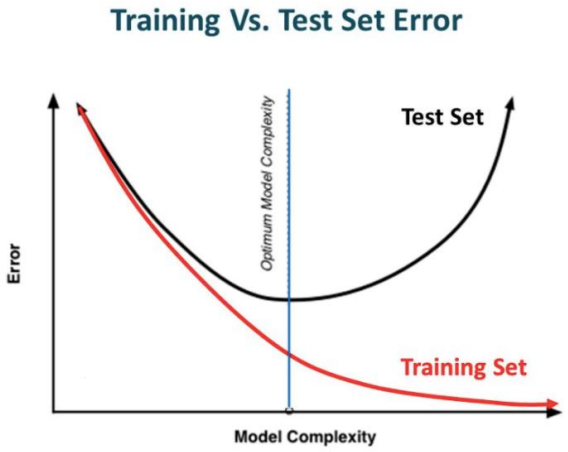
Regularization for CNN

Before we deep dive into the topic, take a look at this image:

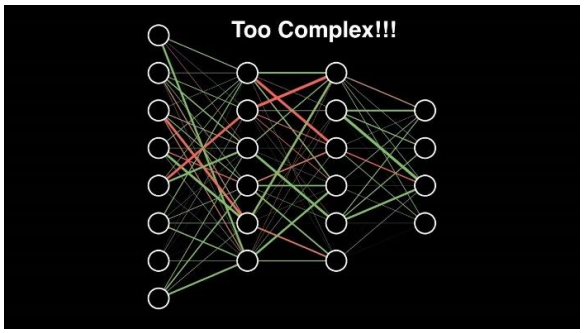


Have you seen this image before? As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen data.

In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't. This is shown in the image below.



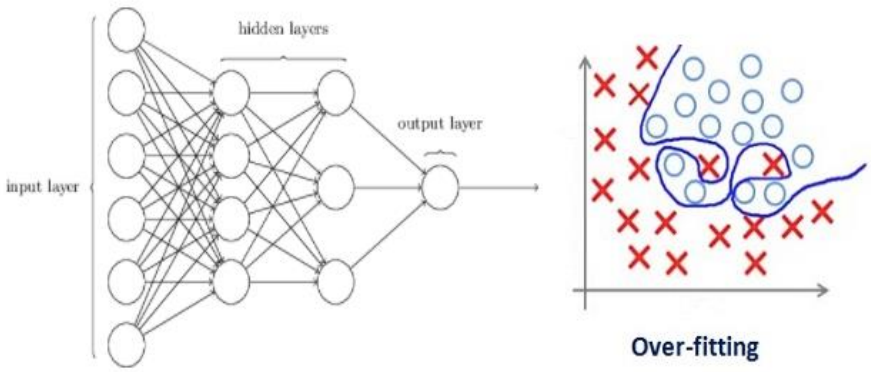
If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting.



Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

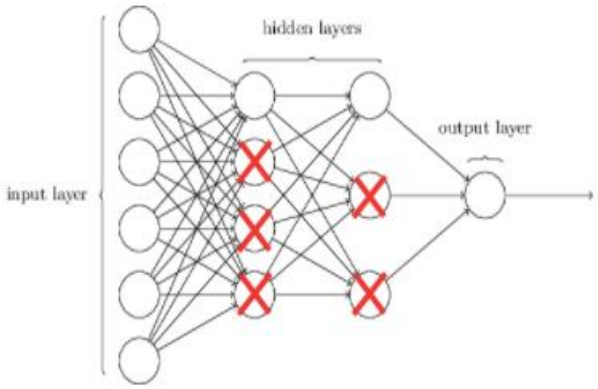
How does Regularization help reduce Overfitting?

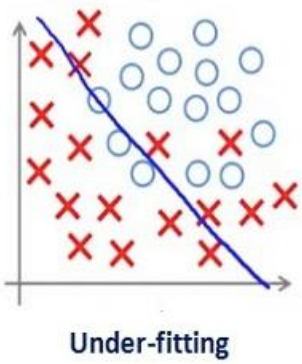
Let's consider a neural network which is overfitting on the training data as shown in the image below.



If you have studied the concept of regularization **in machine learning, you will have a fair idea that regularization penalizes the coefficients. In deep learning, it actually penalizes the weight matrices of the nodes.**

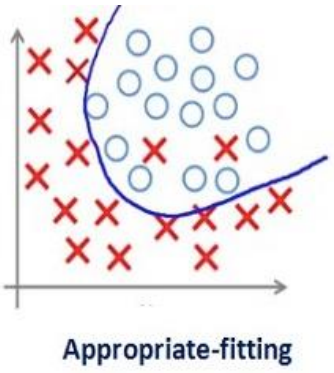
Assume that our regularization coefficient is so high that some of the weight matrices are nearly equal to zero.





This will result in a much simpler linear network and slight underfitting of the training data.

Such a large value of the regularization coefficient is not that useful. We need to optimize the value of regularization coefficient in order to obtain a well-fitted model as shown in the image below.



Different Regularization Techniques in Deep Learning

Now that we have an understanding of how regularization helps in reducing overfitting, we'll learn a few different techniques in order to apply regularization in deep learning.

L2 & L1 regularization

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

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

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

However, this regularization term differs in L1 and L2.

In L2, we have:

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

Here, **lambda** is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as *weight decay* as it forces the weights to decay towards zero (but not exactly zero).

In L1, we have:

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

In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. **Hence, it is very useful when we are trying to compress our model. Otherwise, we usually prefer L2 over it.**

In *keras*, we can directly apply regularization to any layer using the regularizers.

Below is the sample code to apply L2 regularization to a Dense layer.

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
```

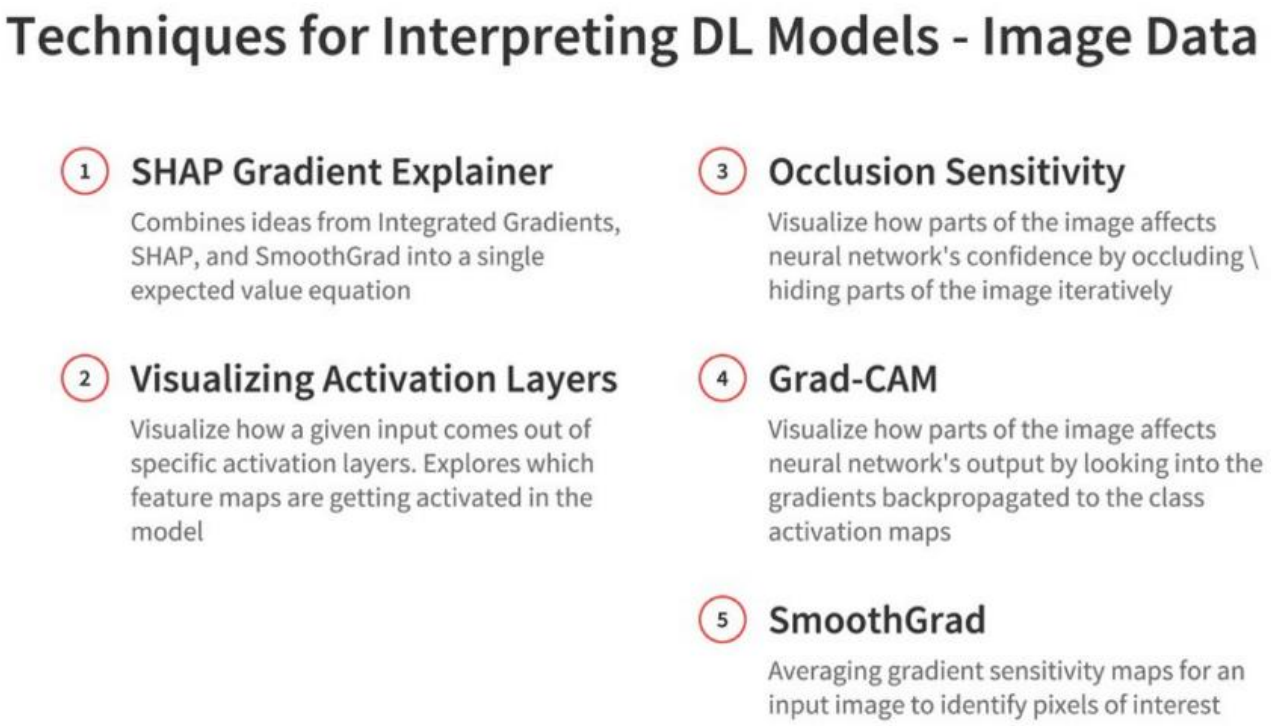
```
                kernel_regularizer=regularizers.l2(0.01)
```

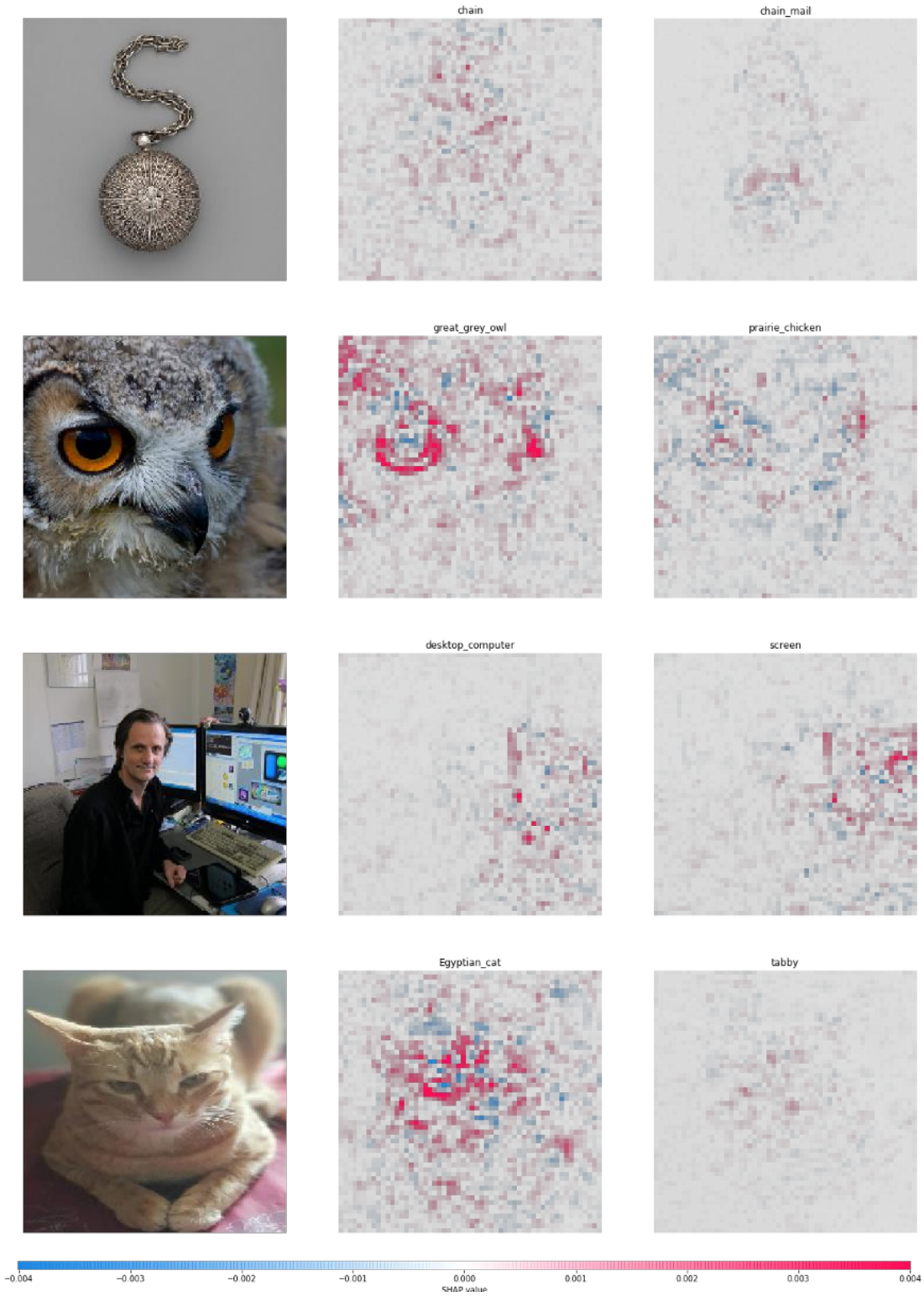
Note: Here the value 0.01 is the value of regularization parameter, i.e., lambda, which we need to optimize further. We can optimize it using the grid-search method.

Interpreting and explainability with CNN

Convolutional neural networks learn abstract features and concepts from raw image pixels. With convolutional neural networks, the image is fed into the network in its raw form (pixels). The network transforms the image many times. First, the image goes through many convolutional layers. In those convolutional layers, the network learns new and increasingly complex features in its layers. Then the transformed image information goes through the fully connected layers and turns into a classification or prediction.

- The first convolutional layer(s) learn features such as edges and simple textures.
- Later convolutional layers learn features such as more complex textures and patterns.
- The last convolutional layers learn features such as objects or parts of objects.
- The fully connected layers learn to connect the activations from the high-level features to the individual classes to be predicted.




chain
chain_mail
great_grey_owl
prairie_chicken
desktop_computer
screen
Egyptian_cat
tabby
−0.004
−0.003
−0.002
−0.001
0.000
0.001
0.002
0.003
0.004
SHAP value

Best practices with CNN

Convolutional Neural Network — a pillar algorithm of deep learning — has been one of the most influential innovations in the field of computer vision. They have performed a lot better than traditional computer vision algorithms. These neural networks have proven to be successful in many different real-life case studies and applications, like:

- Image classification, object detection, segmentation, face recognition
- Classification of crystal structure using a convolutional neural network
- Self driving cars that leverage CNN based vision systems

Below topics will give you certain ideas to lift the performance of CNN.

The list is divided into 4 topics

1. Tune Parameters
2. Image Data Augmentation
3. Deeper Network Topology
4. Handel Overfitting and Underfitting problem

1. Tune Parameters

To improve CNN model performance, we can tune parameters like epochs, learning rate etc. Number of epochs definitely affect the performance. For large number of epochs, there is improvement in performance. But need to do certain experimentation for deciding epochs, learning rate. We can see after certain epochs there is not any reduction is training loss and improvement in training accuracy. Accordingly, we can decide number of epochs. Also, we can use dropout layer in the CNN model. As per the application, need to decide proper optimizer during compilation of model. We can use various optimizer e.g SGD,rmsprop etc. There is need to tune model with various optimizers. All these things affect the performance of CNN.

2. Image Data Augmentation

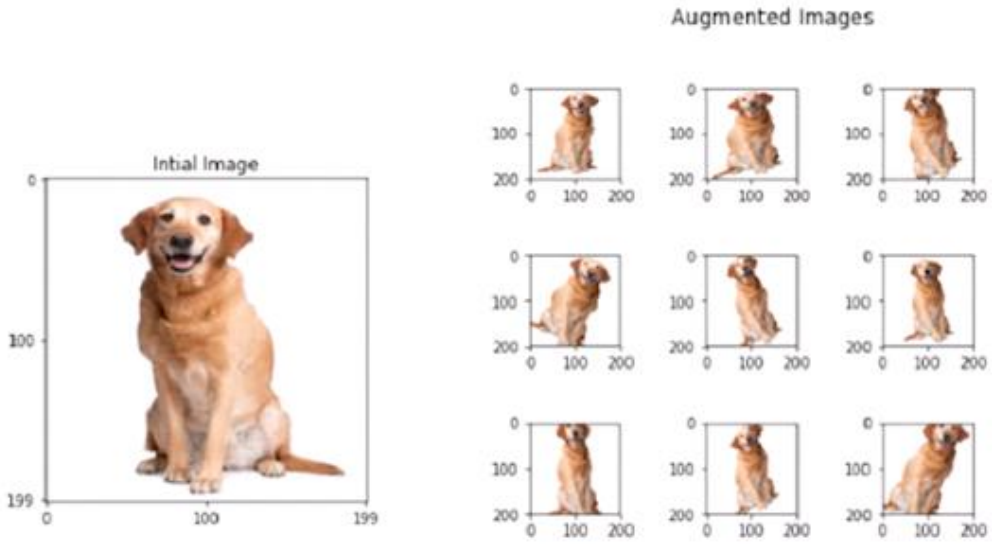
"Deep learning is only relevant when you have a huge amount of data". It's not wrong. CNN requires the ability to learn features automatically from the data, which is generally only possible when lots of training data is available.

If we have less training data available. what to do?

Solution is to use Image Augmentation

Image augmentation parameters that are generally used to increase the data sample count are zoom, shear, rotation, preprocessing function and so on. Usage of these parameters results in generation of images

having these attributes during training of Deep Learning model. Image samples generated using image augmentation, in general existing data samples increased by the rate of nearly 3x to 4x times.



One more advantage of data augmentation is as we know CNN is not rotation invariant, using augmentation we can add the images in the dataset by considering rotation. Definitely it will increase the accuracy of system.

3. Deeper Network Topology

Now let's start to talk on wide network vs deep network! A wide neural network is possible to train with every possible input value. Hence, these networks are very good at good at memorization, but not so good at generalization. There are, however, a few difficulties with using an extremely wide, shallow network. Though, wide neural network is able to accept every possible input value, in the practical application we won't have every possible value for training.

Deeper networks capture the natural "hierarchy" that is present everywhere in nature. See a convnet for example, it captures low level features in first layer, a little better but still low level features in the next layer and at higher layers object parts and simple structures are captured. The advantage of multiple layers is that they can learn features at various levels of abstraction.

So that explains why you might use a deep network rather than a very wide but shallow network.
But why not a very deep, very wide network?

The answer is we want our network to be as small as possible to produce good results. The wider network will take longer time to train. Deep networks are very computationally expensive to train. Hence, make them wide and deep enough that they work well, but no wider and deeper.

4. Handel Overfitting and Underfitting problem

In order to talk on overfitting and underfitting, let's start with simple concept e.g Model. What is model? It is a system which maps input to output. e.g we can generate a model of image classification which takes test input image and predicts class label for it. It's interesting!

To generate a model we divide dataset into training and testing set. We train our model with classifier e.g CNN on training set . Then we can use trained model for predicting output of test data.

Let's start with Underfitting

The Example of underfiiting is your model is giving 50% accuracy on train data and 80% accuracy on test data? Its the worst problem..

Why it occurs?

The answer is Underfitting occurs when a model is too simple — informed by too few features or regularized too much — which makes it inflexible in learning from the dataset.

Solution

we would suggest if there is underfitting, focus on the level of deepness of the model. You may need to add layers.. as it will give you more detailed features. As we discussed above you need to tune parameters to avoid Underfitting.

Overfitting

The Example of overfitting is your model is giving 99% accuracy on train data and 60% accuracy on test data? Overfitting is a common problem in machine learning..

There are certain solutions to avoid overfitting

1. Train with more data
2. Early stopping
3. Cross validation

Summary

- Performing convolution as pointwise multiplication in the frequency domain can provide a speed up as compared to direct computation.
- The motivation of InceptionNet comes from the presence of sparse features Salient parts in the image that can have a large variation in size
- Batch normalization has been shown to be extremely effective at reducing the number of epochs it takes to train a neural network.
- Placing the BN after the RELU yields slightly higher accuracy and lower loss.
- DropConnect sets a randomly selected sub-set of weights within the network to zero. Each unit thus receives input from a random subset of units in the previous layer.
- To improve CNN model performance, we can tune parameters like epochs, learning rate etc
- Image augmentation parameters that are generally used to increase the data sample count are zoom, shear, rotation, preprocessing function.