UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

VERY DEEP CONVOLUTIONAL NEURAL NETWORKS FOR FACE IDENTIFICATION

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Alessandro Luca Vilardi

In partial fulfilment

of the requirements for the degree of

MASTER IN ELECTRONICS ENGINEERING

Advisor: Elisa Sayrol

Co-Advisor: Josep Ramon Morros

Barcelona, July 2016

Title of the thesis: Very deep convolutional neural networks for face identification

Author: Vilardi Alessandro Luca

Advisor: Sayrol Elisa, Morros Rubió Josep Ramon

Abstract

The goal of this thesis is to evaluate the face identification problem using very deep convolutional neural networks. In recent years, the use of CNN, with a large amount of images in databases, have made the deep learning technique very performant. The problems in training a network from scratch, such as having sufficient hardware resources and large databases, can be overcome using the finetune technique on pretrained models. This thesis evaluate the performance in finetuning for face classification the most recent CNN architectures which have obtained the best results at ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in the last years, in particular VGG, GoogLeNet and ResNet. All the pre-trained models of the CNNs were downloaded from the MatConvNet website. VGG-16 has shown best results in face classification which was followed with ResNet-101 and GoogLeNet that are the matter of this thesis.

Acknowledgements

I would like to thank Professors Elisa Sayrol and Ramon Morros for giving me the opportunity to write a thesis in this field, and for giving me always supported during these months of learning. I thank them for their availability, professionalism and sympathy.
I would also like to thank Albert Gil, simply the best in its field, always ready to respond quickly and with openness regarding the possible computer problems I encountered.

Revision history and approval record

Revision	Date	Purpose
0	01/06/2016	Document creation
1	01/07/2016	Document revision

Written by:		Reviewed and approved by:	
Date	04/07/2016	Date	04/07/2016
Name	Alessandro Vilardi	Name	Elisa Sayrol Josep Ramon Morros
Position	Project Author	Position	Project Supervisor

Table of contents

1. Introduction

Face recognition is one of the Computer vision problems that has arisen in recent years. The high possible variance in images, caused by illumination, obstructions, different facial expressions and face positions make it difficult to have an algorithm which is invariant to these problems involving in a good classification. In previous years the Eigenfaces and Fischerfaces were some of the algorithms developed to treat with face problems. In last years, several classification problems in computer vision have boosted its performance by using Deep Learning techniques, in particular Convolutional Neural Networks (CNNs). The topic of this research project focus in face classification and explores the state of the art in deep learning architectures such as VGG, GoogleNet, and ResNet, which have recently shown to perform better in the ImageNet Challenge than previous networks.
To evaluate the performance of these very deep CNNs, pre-trained models of these architectures have been used, with a re-training step to adapt them to different face databases. The MatConvNet Deep-Learning platform, a library freely released in 2014 developed for the Matlab envinronment was used in this work.

The starting point was to learn how to use the new Matlab-MatConvNet library, so first we considered the simpler very deep architecture VGG-16 pretrained with face images (VGG-face) released by the MatConvNet group, and finetune it for face classification using the database Google-face-UPC. At the beginning, the main goal of the thesis was to consider the GoogLeNet structure, and eventually apply its 'Inception modules' architecture to VGG-face. The not so satisfactory results led to another idea of combine the two networks VGG-face and GoogLeNet-ImageNet, instead of applying directly the inception modules to VGG.
To make a consistent performance comparison of VGG-16 and GoogLeNet, both networks pretrained with ImageNet were finetuned using the face database Google-face-UPC, obtained the accuracy results in face classification.
Lastly, for a complete overview about very deep CNNs, the ResNe-t101 model, just released few months ago (available as MatConvNet model pretrained with ImageNet), was used to have a comparison of the three very deep convolutional neural networks VGG-16, GoogLeNet and ResNet-101.
The different hyperparameters to set at the training have taken some time in finding a good configuration for finetune each network. The problem of not obtaining good results in first instance in finetuning ResNet had recommended that maybe using a large publicly available face database, like FaceScrub, with many more images than the database Google-face-UPC, could lead to obtain better results. Finally, a comparison of VGG-16, GoogLeNet and ResNet-101 was made using the FaceScrub dataset. Only at the end of the work, it was found the correct way in using ResNet, and so this time finetune was successfully also with the smaller database Google-face-UPC.
The MatConvNet training graphs that don't appear in the thesis might be found in the Annex.

2. State of the art of the technology used or applied in this thesis

In this Chapter, there is first a basic description about the convolutional neural network: how each computational blocks that compose the network works, and what is the algorithm that allows to reach the pattern classification. After this, there is an architectural description of the CNNs used in this project: VGG-16, GoogLeNet and ResNet-101. Finally a brief introduction to MatConvNet, which is an open-source library developed for a Matlab envinronment, that is suitable for training a CNN and powerful for its code flexibility and GPU capability.

2.1. Convolutional Neural Networks

From the inspiration of studies about the functionality of the visual cortex, considering the results of Hubel and Wiesel works, a convolutional neural network is an artificial neural network that presents a sparse connectivity, on biological inspiration, and it is applied with success in many computer vision problems.
In the classification problem, the structure of a convolutional neural network is a series of computational blocks (or layers) stacking together and ending with a classifier, that make a labelled class prediction about an input image. Typical layers are: Convolution, Activation function, Pooling, Normalization, Classifier. All these computational layers take inspiration to the functionality of the visual cortex in recognising objects, people, and.. simply everything. Just one of the reason that led to the development of many mathematical models of visual cortex was:

> ‘‘The mechanism of pattern recognition in the brain is little known, and it seems to be almost impossible to reveal it only by conventional physiological experiments. So, we take a slightly different approach to this problem. If we could make a neural network model which has the same capability for pattern recognition as a human being, it would give us a powerful clue to the understanding of the neural mechanism in the brain.’’ Kunihiko Fukushima,1980 [18]

So a convolutional neural network can be an idea of how the brain works in recognition, and it has reached very good results in recent years, especially from 2011, in some cases also outperforming the human accuracy.

2.1.1 Neuron model

A biological neuron is the unit cell of the nervous system. It can receive and propagate signals from its connections. In particular a neuron presents many input connections (dendrites) and a single output (axon) that can be connected to other neurons or to biological tissue. The signals propagate from one neuron cell to another via synaptic process. The ensemble of cells connection form a neural network. Synaptic signals may be excitatory or inhibitory. If the input excitations are large enough, an impulse signal, also called action potential, activate the neuron cell and propagates through the axon.

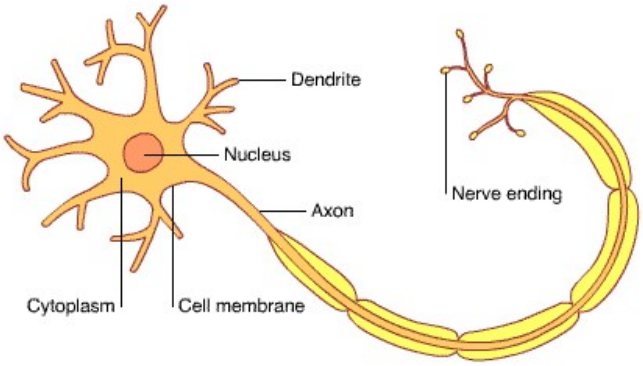


Fig.1 A biological neuron structure.[1]

Inspired by the biology, a single neuron model or perceptron (Fig.2), can be view as a cell with many inputs and a single output. Each input is weighted. When the sum of inputs reach some threshold of the activation function, the output of the neuron is active and can propagate to other neurons.

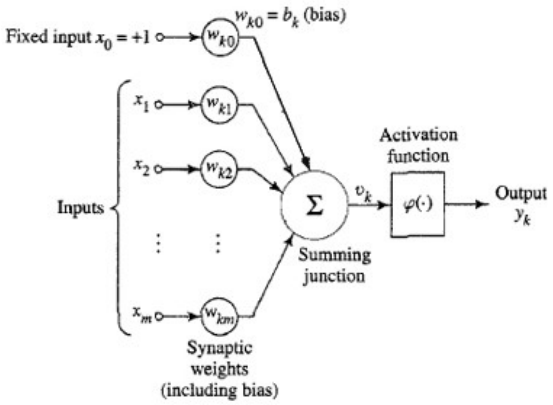


Fig.2 This picture explain the matematical model of a neuron functionality, so a sum of products of different input with different weights, and finally an activation function that activate in case, the output of the neuron. The bias input is like an offset, and can be present or not in a neural network model.

The inputs x_i of a neuron are combined using a weighted linear combination (1).

(1) $$f(x) = \sum_{i=1}^{N} (w_{ij} \cdot x_i).$$

[1] Image from http://www.frankswebspace.org.uk

The result is then modified by a nonlinear activation function φ that generate the output y,

(2) $$y = \varphi(f(x))$$,

where φ can be tanh, sigmoid, ReLU, or other activation functions.

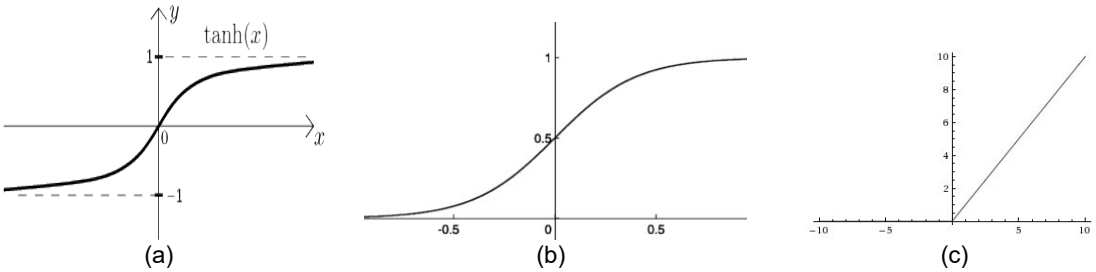


Fig. 3 Different activation functions of a neuron (a) Hyperbolic tangent function: *y=tanh(x)* (b) Sigmoid function: $y=1/1+e^{-x}$ (c) ReLU function: *y=max(0,x)*.

The studies about the visual cortex have shown a regular structure of neuron connections. From this assumption, a neural network structure is composed by different layers, each one is made by a huge amount of neurons. A typical multi-layer structure stacks more layers of neurons, where the neurons of a layer receive the input only from the previous layer, and feed the output to the next one. The layers between the input and the output of a network are called hidden layers.
The output layer in a classification problem is composed by N neurons, where N is equal to the number of classes to predict.

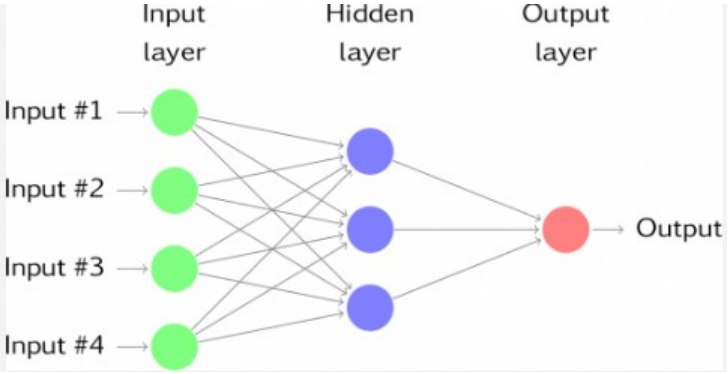
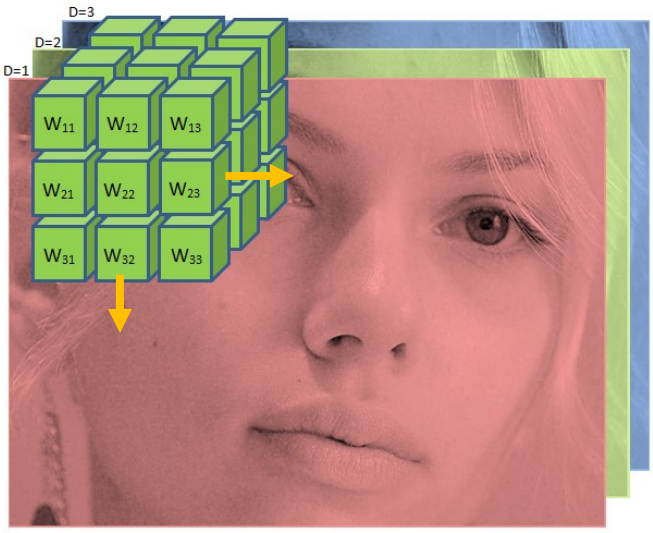


Fig.4 A simple network of neuron layers, where each layer of neurons receives inputs from the previous layer. The outputs of one layer are inputs to the next one.

2.1.2 Convolution layer

The convolutional operation is the basic function that brought to the development of the convolutional neural network. It works on an image, also in 3-channel RGB, without vectorising it in a single vector as for example the Eigen-face technique does, but it process the image in the original dimension WxHxD or simply on the resized image to fit a given input size for a multilayer CNN.
The convolution operation is a sum of dot product in space between an input volume and a kernel, made by a cubic matrix of weights, that slides along the input volume to produce an output feature map (fig.6). The kernel works like a filter of the input and extracts some visual information *f(w,x)* from the image.



$$f(w,x) = \sum_{d}^{D} \sum_{j=1}^{k} \sum_{i=1}^{k} (w_{ji,d} \cdot x_{ji,d}) \quad (3)$$

Fig.5 Convolution operation of a 3x3 kernel on a RGB image.

The kernel must have size KxK less or equal to the input, but always with the same depth, so for a input RGB image, the kernel must have a depth of 3. The values inside the kernel are the weights w that filter the input simply multiplying them with the respective value of the image. The values of the input image have to be set in a range of 8 bit (0-255, or zero mean -128,+127).
The weights present in a neural networks are the parameters to be learned, and, at the end of a training process, these learned parameters will characterized the network for a computer vision problem for a specific training database.
A convolution layer has more than one filter, so each output layer is composed by a block of different feature maps (fig.7), each one derived applying different kernels, and so different weights, to the input volume. The total number of learned parameters in a convolution layer can be calculated as KxKxNxM, where N is the number of channel depth of the input and M is the number of output feature maps.

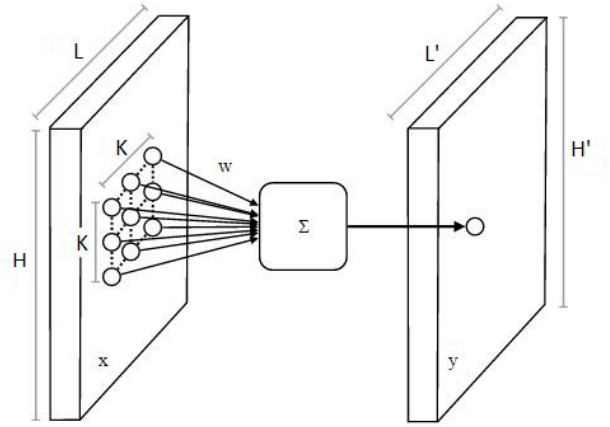


Fig.6 Convolution operation producing an output feature map

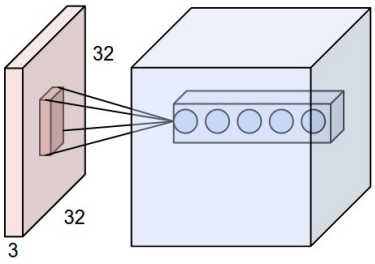


Fig.7 An input layer of dimension 32x32x3, and the output convolutional layer with five different feature maps.

Other parameters in a convolution operation are the stride, padding and the kernel dimension.
The stride determines how many position the kernel shifts along the image in horizontal and in vertical way. Typical values are [1 1], [2 2], [4 4].
Padding means adding zero values on the sides of a image to obtain for example an output matrix of the same dimension of the input, in case of convolution, and so maintain constant the dimension of input-output matrix, or it is also used to permit to the kernel to cover all the image in case that the size of the image it's not a multiple of the kernel.

2.1.3 ReLU layer

The Rectified Linear Unit has become very popular in the last few years as activation function [5]. This layer is placed always after a convolution layer. It computes the function

(8) $f(x)=\max(0,x)$.

In other words, the activation is thresholded at zero.

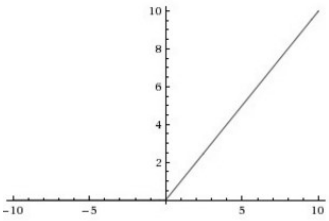


Fig.8 The ReLU function (8).

There are some positive and negative aspects in using ReLU as activation function.
Instead of using more complex functions such as tanh or sigmoid, ReLU simply make a threshold activation at zero. Its linear form for x>0 also performs a better propagation than a saturated non-linear function and so accelerates the training convergence using the stochastic gradient descent.
On the other side, a large gradient value, back propagated through a ReLU during a training could cause weights update in a way that a neuron will never be active, independently on the inputs values, and so that neuron will be useless. If this happens, then the gradient flowing through the unit will forever be zero from that point on. Setting the learning rate[2] high can cause that a huge part of the neural network is not active at all. So it's important to set properly the learning rate and also to use an appropriate weights initialization, such as the He initialization (He et al. 2015)
In the literature there are some modifications to this activation function, for example the Leaky ReLU function f(x)= max(0.01x,x). It doesn't present a zero output when input is less than zero, but presents a small negative slope. This function wants to address the problem of 'diying neurons'. Also the slope can be a parameter of the activation function as the Parametric ReLU (PReLU) function does. The advantages of using different ReLU types are not so studied in literature.
A question can be why use activation function after each convolution. It is because without an activation function a convolutional neural network has the simply capacity of a linear classifier, and so can learn efficiently only if data are linearly separable, but with activation it has more complex capacity and can represent non-linear functions and so solve different complex tasks.
In this project only used the simpler ReLU (fig.8) is used as an activation function for the CNNs. The ReLU function has to be considered in the initialization of the weights, using a specific function like the He_initialization, to make work the whole training process. Xavier_initialization make weights in deeper layers to have a very low standard deviation, so all near the same values during the training process, and classification doesn't work.

[2] The learning rate is a parameter that during the training determine the amount of weights update

2.1.4 Pooling layer

The pooling operation makes a downsampling of an input volume spatially, reducing the dimensionality in width and height, without reducing the depth (fig.9). A subsampling operation reduce the resolution of the input feature map and it is essential to solve the spatial invariant problem in computer vision. Pooling operation has no learned parameters, but it makes a compression of the data, and it is important to obtain translation-invariant features [3].

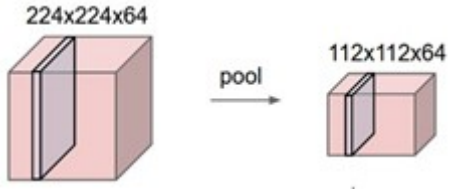


Fig.9 The effect of pooling operation: the input is downpampled, so the output has the same number of feature maps, but the half of the planar size. In this example the input volume of size [224x224x64] is pooled with filter size=2, stride 2 into output volume of size [112x112x64].

Most common pooling opreations are the max and the average pooling. In the paper [3], it is declared that ‘‘empirical results show that a maximum pooling operation significantly outperforms subsampling operations. Despite their shift-invariant properties, overlapping pooling windows show no significant improvement over non-overlapping pooling windows.’’ So, usually, it is common to find max-pooling layers with 2x2 kernel dimension and stride 2 to not overlap the sampling, and it is also common to insert this subsampling operation after a convolution layer instead of using directly a convolution with stride equal to 2 to downsample the input.

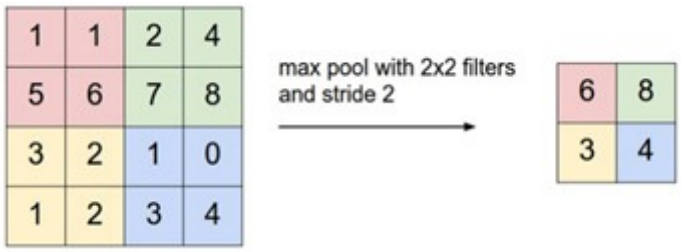


Fig.10 The most common downsampling operation is max-pooling, that calculate the maximum value of k x k input square, and reported it to the output.

2.1.5 Batch normalization layer

The batch normalization is a technique introduced in 2015 that permits to train very deep network in a very fast and efficient way, and resolve the problem of taking really care about weights initialization of a neural network. The intuition of BN is to prevent model

explosion as the gradient increased with large learning rate. This layer should be insert between a convolution layer and its activation function.
Batch normalization is divided in two operations: first normalize the input data with zero mean and unit variance within a batch of training images, then BN will optimally apply scale and sift parameters γ and β, to preserve the network capacity.

Why Batch normalization is useful:

‘‘Training Deep Neural Networks is complicated by the fact that the distribution of each layer’s inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. This phenomenon is the so called ‘internal covariate shift’, and can be addressed by normalizing layer inputs. This method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch.

Batch Normalization allows to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin.
Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, we make sure that the transformation inserted in the network can represent the identity transform. To accomplish this, we introduce, for each activation x(k), a pair of parameters γ(k), β(k), which scale and shift the normalized value:

(4) $$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

These parameters are learned along with the original model parameters, and restore the representation power of the network. ’’ (Ioffe and Szegedy, 2015) [4]

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Fig.11 Batch normalization computational steps

If γ is equal to the variance and β equal to the mean of the mini-batch, the original input x is restored. This parameters can be learned using the standard Stochastic Gradient Descent, as all the other parameters of the convolution neural network.

At the inference time, the mean and variance of each feature must be the ones calculated from the whole training set, so during the training, the batch normalization layer also take care in averaging the mean and variance of every mini-batch group of images, and then at inference apply the linear transformation using also the learned parameter γ and β:

(5) $$\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$, average mean of the whole training set

(6) $$\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma^2_{\mathcal{B}}]$$, average variance

(7) $$y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$$, linear transformation applyed by the batch normalization layer at inference

2.1.6 Dropout

The dropout layer is useful to avoid the overfitting of the data during the training process and it is applied to the fully connected layers at the end of a neural network. In practise it drops out randomly an amount of neuron units proportional to its setting parameter. For example a dropout of 0.5 connects only 50% of the units. Applying this during training makes that different units are trained randomly for each iteration. This can prevent to overfit the data. At inference, all units are present in the network, but the output weights are scaled by the probability of its presents during training, so in the example $0.5*w_i$. for other details take reference to the paper [14].

2.1.7 Softmax, Loss and regularization

The *Softmax*, also called multinomial logistic regression, is a math function that normalize the predicted scores of the classes. The scores are the output values of a convolutional neural network. They are placed in a vector form as outputs of the classifier, where each position corresponds to a class, and so in an inference test the maximum value of the score vector should correspond to the correct output class.
The function f(z) is the softmax, where z_j is the score value of the correct class and z_k are the score values of each k class:

(9) $$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

The softmax function is used after a classifier, as computing the probabilities of each class. The Softmax take the scores and squashes it to a vector of values between zero and one. So the outputs of a softmax can be interpret as a vector of probabilities of a label in a classification task. During the training, the goal is to maximize the probability of the correct class. So, what is done is to minimize the negative log likelihood of the correct class.

The Loss expression in a Sotmax classifier is:

(10) $$L_i = -\log\left(e^{s_{y_i}} \Big/ \sum_j e^{s_j}\right)$$,

where s_{y_i} is the score value of the correct class, and with s_j the scores of all the classes. The Loss function is also called cost function or objective, and quantifies the goodness of the prediction, based on the scores, along the training data. The lower is the value of the loss, the better is the classification of the training data. At the initialization step all weights are near zero, so also all the class scores, and the loss should be of value –log(1/Nclasses). This is useful as sanity check at the beginning of the training process, to see for example if there's something wrong in the developed code, or if the loss is implemented in a good way.

The role of regularization:

Unfortunately, the solution in finding optimal parameters can be not unique, so it is important to add a regularization in the loss formula. This regularization term, permits to generalize better the data and have less test errors. This occurs because regularization fights against the data loss, penalizing unbalanced weights, so large weights and zero weights in a layer, and increase the generalization property spreading out better the weights values around zero, avoiding that a particular input influence most the output than others. So, in practise, the regularization takes care about having better weights, whereas the data loss looks only to the class score predictions.

This is the L2 formula for the regularization, which is the most common used:

(11) $$R(W) = \sum_k \sum_l W_{k,l}^2 \ ,$$

where W are the weights of the classifier.

So the Loss function for N training examples, considering also the regularization term is:

(12) $$L = \frac{1}{N}\sum_i L_i + \frac{\lambda}{2} R(W),$$

The regularization term is weighted by a factor λ, which is an hyperparameter of the training process.

Biases do not have the same effect since, they do not control the strength of influence of an input. It is common to only regularize the weights W but not the biases b, but however, in practice, this often turns out to have a negligible effect. Note that due to the regularization loss, the cost function can not achieve zero in all examples, because this would only be possible in the pathological setting of weigths=0. If the hyperparameter λ is high, the weights W would lead to smaller weights, and so smaller scores. The probabilities computed by the softmax would be more diffuse along classes, because of the exponential, and so less selective label prediction. So also the hyperparameter λ can affect the training results.

2.1.8 Optimization

The optimization problem is to find the best weight values for all the network that will make a good classification of the data. At the starting point the weights to be learned are usually randomly initialized and so the score output presents incorrect prediction. To learn correct weight values, the optimization algorithm looks at the loss values and tries to minimize this loss to fit the data. A loss, or objective, can be thought as a multidimensional function with a global minimum (in real cases there can be probably many local minimum), and the goal is to update the weights towards the minimum loss value. The loss, using a softmax classifier, is computed as in (10).
To minimize the loss function the Stochastic Gradient Descent method with back-propagation is used. The loss derivative respect to each weight dimension is computed, and then the gradient is back propagate to the other layers of the convolution neural network using the chain rule of the derivatives. Finally the weigths are updated in the negative gradient direction through this formulation:

(13) $W_i := W_i - \alpha \Delta W_i$

The parameter α is called *learning rate* and determine the step size of weight updates. ΔW is the gradient of the weights. The learning rate is a critical parameter of the training process because it can lead to an oscillating learning behaviour or to the explosion of the model due to high weights update.
Taking a mini-batch of images and computing the gradient on it is more computational efficient than looking to a single image or the entire training set, so it is actually more convenient to select a random mini-batch and pass this batch to the GPU to a faster computation. So the size of the mini-batch is usually determined by the memory capacity of the GPU available.
After the mini-batch computation, the weights are updated taking the mean of the gradients ΔW in the mini-batch.
The *weight decay* training parameter is derived from the presence of the regularization in the loss formula (12). In particular the weigths update with regularization is:

(14) $$W_i := W_i - \alpha \frac{dL}{dw_i} - \alpha \lambda W_i$$,

where the weight decay is defined as

(15) $$w_d \stackrel{def}{=} \alpha * \lambda$$.

To increase the speed of the parameters optimization in a training process there are a lot of algorithms that modify the SGD formula (13) to have an update of the weights that decreases the loss faster. One of these algorithms is the momentum update defined as:

(16) $v = mu * v - \alpha * \Delta w$,

(17) $W_i := W_i + v$.

In this formulation *v* is a parameter that take in account of the speed of the gradient descent and it is initialized to zero. The momentum parameter *mu* scale the amount of the velocity factor. It is usually set to 0.5, 0.9, 0.99. Using this weights update with momentum (17) instead of the stochastic gradient descent allow to reach the minimum loss in a faster way. Sometimes it could happen that using a too high momentum can

make the network to not converge for the higher weights update, so it is a hyperparameter that the user has to take in consideration in the training setting.
So, finally we can call the optimization setting parameters as hyperparameters, and these are the learning rate, the weight decay and the momentum. The weights that have to be updated are instead the parameters of the network.

2.2 State of the art models

In recent years the topic of face recognition has been one of the most studied problems in computer vision, and so many different algorithms have been developed to address it. The performance has increased mostly in few years, with the use of the deep learning techniques. In particular the convolutional neural networks models that are the state of the art in face classification are DeepFace and FaceNet. DeepFace [19] is a system developed by Facebook Research that reaches results close to the human performance in face recognition. The network architecture consists on a 3D alignment step based on fiducial face points which can be applied to RGB images. Using this system, it is therefore not necessary to use several layers of convolutions to reach results near human accuracy. This system uses the deep learning technique with a nine-layer deep neural network with 120 millions of parameters. The 3D alignment was made by taking as a reference a general 3D face and then use the previous 2D alignment on faces to warp it and having the final 3D shape.

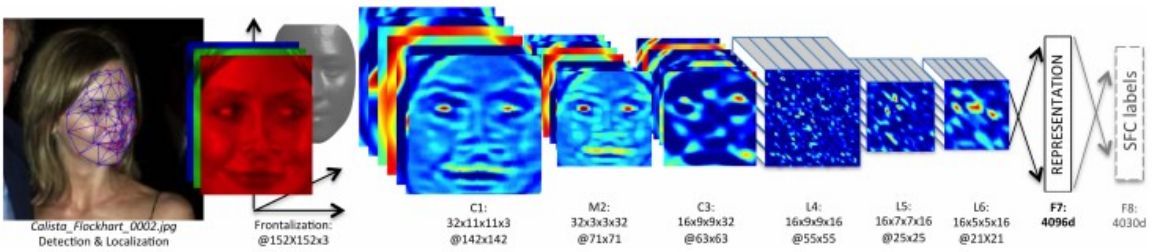


Fig.12 Outline of the DeepFace architecture

In the competition between Facebook and Google, in 2015 the FaceNet [17] system was developed by Google for solving the problem of facial verification. This system was tried on a Zeiler&Fergus based architecture and on another architecture that uses Inception modules. FaceNet learns a mapping from face images and then uses a compact Euclidean space to measure the face similarity instead of a simple classification layer. The output was trained using a triplet-based loss function that works with three input images, where two belong to the same person and the other to a different one. The triplet loss works to minimize the distance between images of the same identity, and maximizes the distance with a different identity. The trained dataset consisted on 8 millions of different people with a total amount of 200 millions of images.

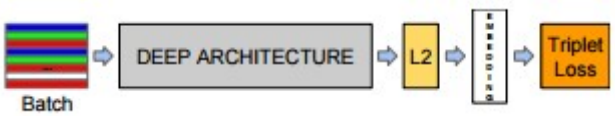
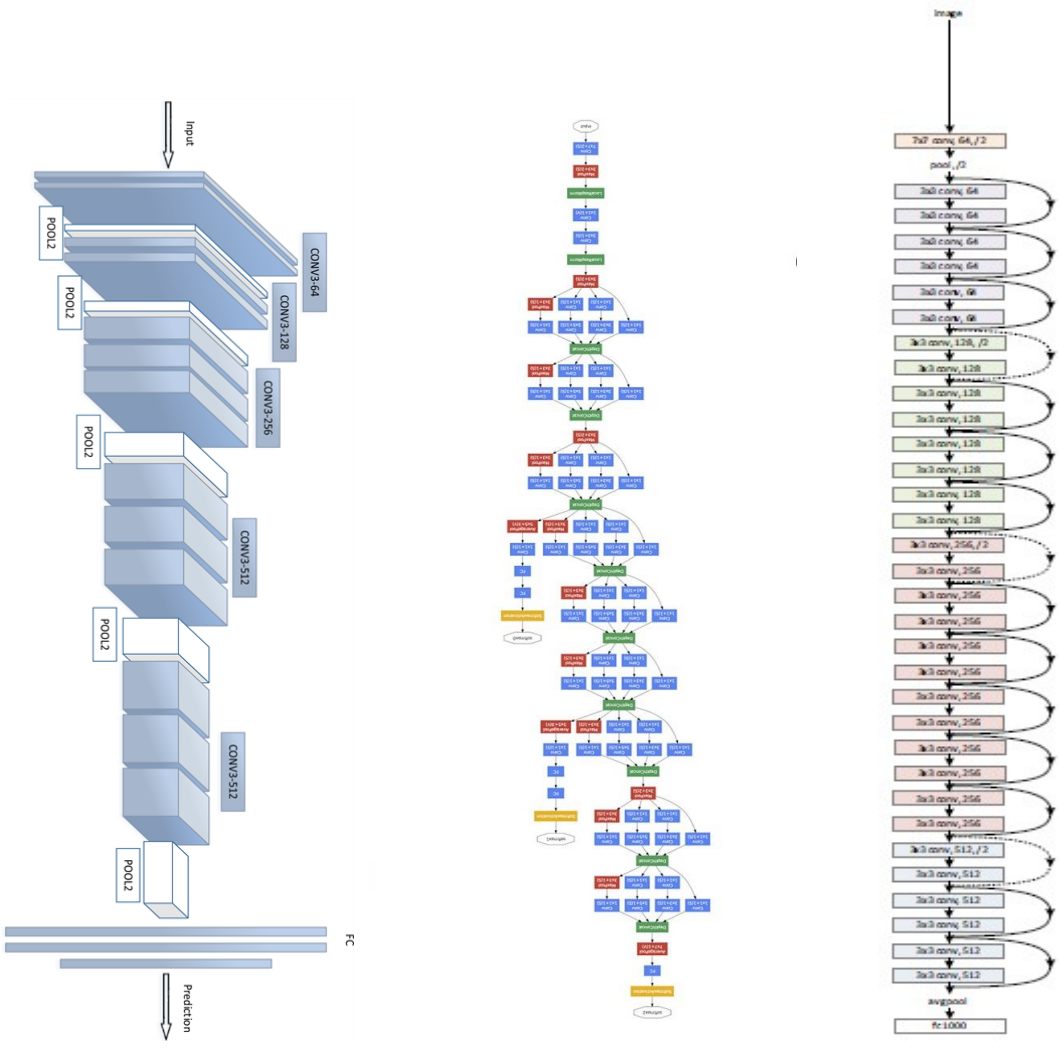


Fig.13 FaceNet model structure. It consists of a batch input layer and a deep CNN followed by L2 normalization, which results in the face embedding. During training this is followed by the triplet loss function.

This method was evaluated on Labelled Faces in the Wild (LFW) and YouTube faces dataset for verification obtaining 98.87% on LFW with fixed center crop images and 99.63% using extra face alignment, which is the highest accuracy result ever reached in LFW.

2.3 Very deep CNN models

For this thesis we want to evaluate the performance of the most powerful CNNs of the last years for face classification. So, the focus is about three different very-deep CNN: VGG-typeD (16 weight layers), GoogLeNet (9 Inception layer), and ResNet (evaluating the model with 101 weight layers). For these networks we consider the pre-trained models with ImageNet.
For VGG, there is also a model trained using a private database of faces [12], called VGG-face.



(a) (b) (c)

Fig.14 Very deep convolution neural networks structures (a) VGG-D, 16 convolution layers (b) GoogLeNet 9 Inception layer (c) Resnet-34 (in this thesis it is used ResNet-101, which architectural details are explained later in section 2.2.3)

Model	#Weights (without classifier)	#FLOPS	File .m size (MB)
VGG-D	134.248.128	15 billion	507
GoogLeNet	5.971.648	1.5 billion	29.5
ResNet-101	42.357.952	7.6 billion	167

Table 1. Computation and memory comparison of the CNNs analized

Model	Inference speed (s)
VGG-D	1.125
GoogLeNet	0.25
ResNet-101	0.62

Table 2. Speed computation comparison. The inference speed was computed on cpu using tic-toc matlab functions, making a mean over 100 inference images. It include the time to read the image from disk, pass through the network and extract the maximum output value, that correspond to the predicted class.

2.3.1 VGG

Made by the Visual Geometry Group, Department of Engineering Science at University of Oxford, VGG was born by an investigation of the effect in accuracy in increasing the depth of a convolutional network, using mostly 3x3 convolution filters. This work showed an important improvement on the prior-art architectures increasing the depth to 16 and 19 weight layers. They also declare in the paper [7] that these structures generalise well to other datasets, where they achieve state-of-the-art results. They reached the second place in ImageNet challenge 2014 in image classification.

ConvNet Configuration						
A	A-LRN	B	C	D	E	blocks
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers	
input (224 × 224 RGB image)						
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	1
maxpool						
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	2
maxpool						
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**	3
maxpool						
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**	4
maxpool						
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**	5
maxpool						
FC-4096						6
FC-4096						7
FC-1000						classifier
soft-max						

Table 3. VGG A-E different configurations. The pointed out VGG-D is the configuration used in this project. The blocks are divided by the fact that convolution operate on different input size dimension, due to the pooling subsample. Inside each block, VGG uses padding to keep constant the output size of a convolution layer.

Choosing a small kernel dimension of 3x3 was decided with the intention of reducing the amount of parameters. For example a single 5x5 kernel has the same receptive field of stacking two 3x3 convolution layer. Assuming fixed the input and output number of feature maps to F, the number of parameter of a single 5x5 kernel will be of $25F^2$. Using two 3x3 kernel instead have $18F^2$ parameters in total. Another advantage can be the presence of more non-linear functions in stacking layers, instead of one. This can increase the capacity of the network to be more selective.

VGG Network	A	A-LRN	B	C	D	E
Number of parameters (in millions)	133		133	134	138	144

Table 4. Number of parameters (in millions), from the paper [7].

The layer with the majority of learned parameters is the fully connected layer in block 6. It is a 7x7x512x4096 FC layer, so more than 102 millions parameters in this layer.

2.3.2 GoogLeNet

The GoogLeNet [8] architecture takes the practical issue from the Network in Network by Lin [15], and the theory of Hebbian principle and Arora [16] works. Google introduces a new form of CNN structure called 'Inception module' (fig.13) and stack these modules to increase the depth of the neural network.
The final architecture has a series of nine 'Inception modules', which is the implementation that Google researchers made to go sparser, with an optimized computational efficiency, and so less parameters and faster computation. Less parameter are reached by inserting 1x1 convolution layers in the Inception module as parameters reducers. A faster computation, instead, comparing with the classical stack of layers, is reached by the structure of the Inception module. Stacking convolution layers one after another, may increase a lot the computation, and, if a lot of weights are close to zero this leads to computational inefficiency. Arora says that 'optimal network topology can be constructed layer after layer by analyzing the correlation statistics of the preceding layer activations and clustering neurons with highly correlated outputs' (Arora et al., 2013), so a mathematical analysis near the Hebbian principle 'neurons that fire together, wire together'. Go sparser instead of using fully connected layers mimic biological systems and resolve inefficient computation.
But, in contrast with this assumption, considering the realistic computation on CPU or GPU, going sparser, with also non-uniform sparse models, don't drive towards efficient parallel computation, and a lot of cache misses occurs. So Google has studied how to increase the computational efficiency in hardware, also using a sparse neural network model. The inception module is a way to use dense matrix for the system, and so go faster, but using a not-uniform and sparse structure. However their first architecture with the inception modules without the 1x1 reduction layers, was very computational expensive and with high number of parameters. The second Inception module (in fig.15), with the 1x1 convolutions was instead a high efficient structure, with low memory consumption.

GoogLeNet has won the international competition ILSVRC 2014, becoming one of the best neural network at the state-of-art. The efficiency of this architecture, especially for the power and memory use, make it portable in mobile and embedded computing systems. The number of computational sum-of-products at inference time is about 1.5 billion.

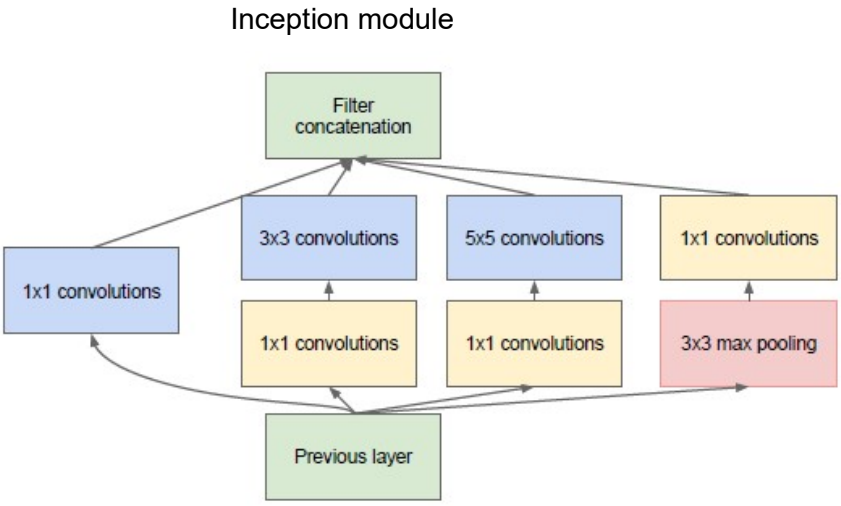


Fig.15 An inception module with dimensionality reduction

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 5. GoogLeNet architecture table [8]. The number of feature maps for each layer of the inception modules, the number of parameters and the amount of operations are presented in this table.

2.3.3 ResNet

ResNet is a Convolutional Neural Network, developed by the MSRA (Microsoft Research Asia) in 2015 (Kaiming He et al.) [9]. ResNet has won the ImageNet Challenge (ILSVRC 2015) in every task.

This architecture addresses the problem of training deeper convolutional neural network with a residual learning structure. This CNN has reached better accuracy results by increasing the depth of the architecture and it is at the same time easy to optimize, due to the structure of the residual block (fig. 17).
The problem of vanishing or exploding gradient in stacking more layers was solved by applying the batch normalization after each convolution layer. But MSRA researchers focus also on another problem called *degradation*:

‘‘..when the depth of a network is increasing, the accuracy can saturate and then degrades rapidly. This is not caused by overfitting, but by adding more layers the model reaches higher training error. The degradation of training accuracy indicates that not all systems are similarly easy to optimize.’’ (He et al., 2015)

The solution they found in dealing this problem was to modify the architecture leaving the network to learn only a residual mapping, and add the input to this output to recover the original mapping (fig.15).

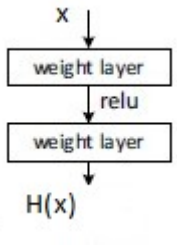


Fig.16 H(x) is any desired mapping

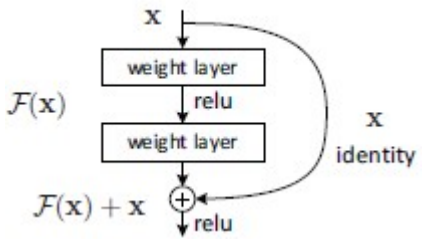


Fig.17 Residual block. F(x) represents the residual mapping, x is the identity mapping. The original mapping presents in figure 14, here become H(x)=F(x)+x.

With this structure there are two possible cases during learning in the optimization of weights: if the identity is the optimal mapping is easy to set weights to zero values, or if optimal mapping is closer to identity, it’s easier to find small perturbations. This makes a good optimization of weights also with a deeper architecture.
As practical design to go deeper, one residual block in Resnet-101 (fig.18), contains three stacked layer: the first is a 1x1 convolution layer to make a dimensionality reduction (lower number of feature maps), then one 3x3 convolution layer, and finally a 1x1 convolution that recovers the original dimensionality, to make appropriate the sum with the identity map. When there is a change in dimensions, for example going from 256 to 512 feature maps, in the identity shortcut is introduced a 1x1 convolution layer to match the output dimensionality.
Applying the dimensionality reduction in the residual blocks, takes under control the number of parameters. For example the model ResNet-152 has lower complexity than VGG-16/19.

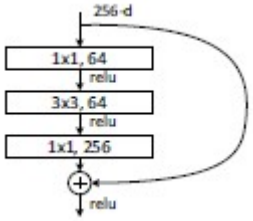


Fig.18 A Residual block example in ResNet-101. It figure out the three stacked convolutional layers in each block, with dimensional reduction and recover.

For the ResNet model, going deeper reach better accuracy results. For example using ImageNet, they declare an error rate described in Table 6.

Model	Top-1 error	Top-5 error
ResNet-34	21.84	5.71
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	19.38	4.49

Table 6. Error rates single model on ImageNet validation, reported in the paper [9].

In this thesis, the performance of ResNet-101 (fig.19) for face classification is evaluated.

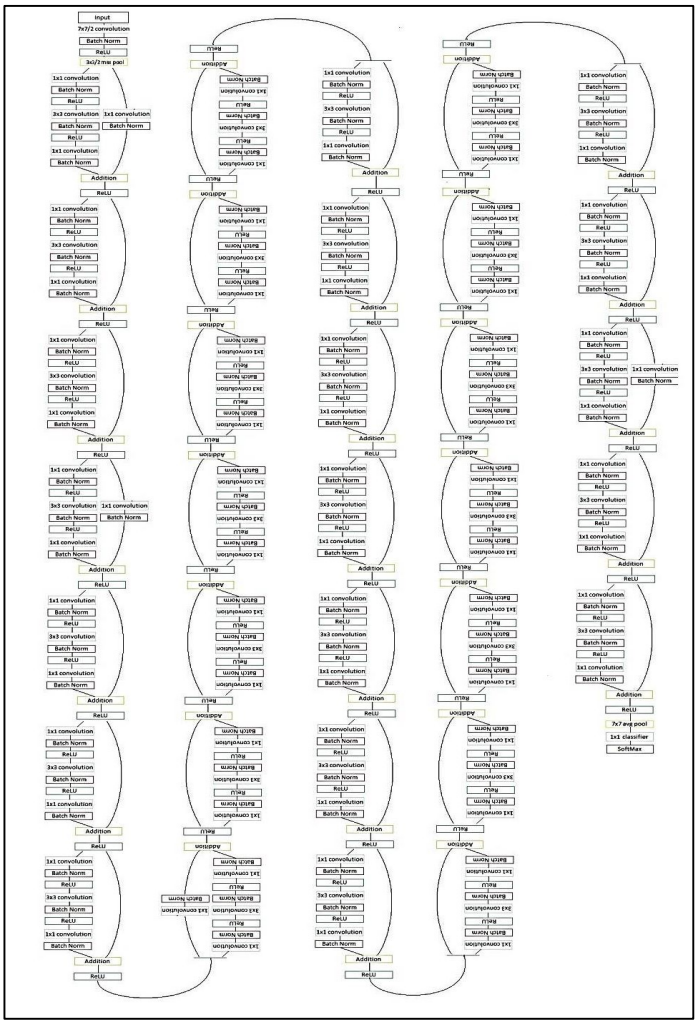


Fig.19 ResNet-101 architecture.

ResNet-101 architecture details:

Each branch of ResNet-101 is composed by 3 convolution layers (1x1, 3x3, 1x1). Then, in cases of change in dimensionality, the input is not connected directly with the sum layer, but is adjusted in dimension by a 1x1 convolution layer. At the beginning there is first a 7x7 convolutional layer with stride 2 and a max-pooling 3x3 also with stride 2, so the same starting layers of GoogLeNet. Then there are 3 Residual Blocks with depth of 256, 4 Residual Blocks with depth of 512, 23 Residual Blocks with depth dimension 1024, and finally 3 Residual Blocks with depth 2048. At every change of dimensionality there is a subsample made by the stride 2 of the convolution layer, so no pooling is used, but only halve the input size increasing the stride and, at the same time, doubling the number of output feature maps at the same layer. At the end a 7x7 average pooling and a Softmax classifier close the network.

2.4 Matconvnet

Matconvnet is a new library for Matlab, realized by Andrea Vedaldi and Karel Lenc, first released on Github on December 2014, that can be used to implement Convolutional Neural Networks (CNN) in Matlab environment [4]. This library can be found and freely downloaded from https://github.com/vlfeat/matconvnet.
MatConvNet has simple MATLAB commands for building CNN blocks such as convolution, normalisation and pooling that can then be combined to create CNN architectures. Matconvnet commands are also optimized for a CPU and GPU computation, with code implementation written in C++ and CUDA. It is possible to develop new building blocks as well as to implement other architectures. To install MatConvNet only MATLAB is needed and a compatible C++ compiler. Using the GPU code requires the freely-available CUDA DevKit and a suitable NVIDIA GPU.
Several examples and several standard pre-trained networks can be downloaded and used in applications.
Matconvnet provides two wrappers suitable for neural networks: SimpleNN and DagNN. SimpleNN is the older wrapper, instead Dag (Directly Acyclic Graph) is the more complex and more flexible one, so usually now it is used to convert to DagNN a previous network description or writing directly new network in DagNN. So it is what I've done, working with pretrained CNNs.
The bases of a CNN in Matconvnet are the building blocks (fig.20) to develop every kind of possible CNN architecture. Each block has its parameters and options[3], such as for a convolutional block (layer) there are the size, stride, padding, input, output and learning parameters.

[3] See Appendix 1 for MatConvNet building blocks code instantiation.

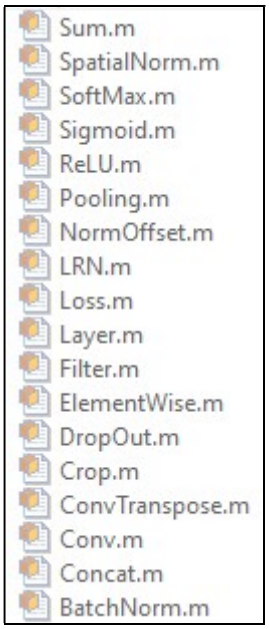


Fig.20 Matconvnet building blocks, version beta-18

The main function in MatConvNet that permits to train a network is *'cnn_train_dag.m'*. Its code is placed in the 'example' folder after downloading the library. This is a complex function that receives as input the train options that the user can set (learning rate, momentum, weight decay, number of GPUs), the network structure, and the images for training and validation. It produces also a training graph (fig.21) in run time useful for the user to check the error and objective values for each training epoch[4].

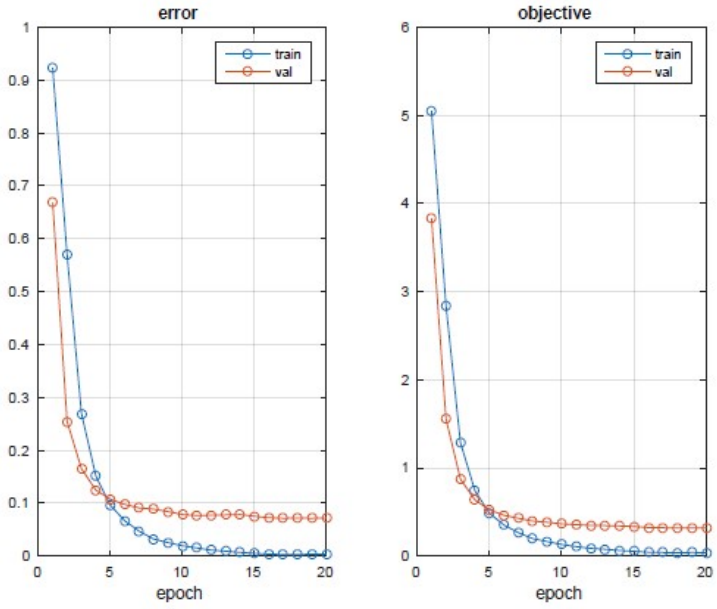


Fig.21 A typical training graph, checking the decrease of the error on the left side and the objective on the right side for the training and validation data along epochs. Different shapes of this graph depend on many elements like the setting of training options, the network, the data, and the amount of data choosing for training and for validation.

[4] An epoch is the one time computation of the whole database in the training process.

The MatConvNet code of the networks VGG-16, GooLeNet and ResNet-101 used in this thesis for the training process are described in Appendices 2 to 7.

3 Methodology / project development

In this Chapter it is explained how to use a pretrained model to adapt it for different databases using the finetuning technique, and the hardware resources used for this goal. At the end, there is an overview on the databases ImageNet, Google-face-UPC, Facescrub and Youtube-face that had been considered in this project.

3.1 Finetuning technique

Training a whole very deep CNN can be problematic in terms of time, resources and large database disposal. In a full training, the network parameters (the weights of the different layers) are initialized randomly and learned using the gradient descent algorithm. If the number of parameters is large, a large number of images are needed to learn these parameters. A full training may not be feasible in applications where the training set is small. This is true especially for a very deep CNN, which should be trained with millions of images to obtain very good results. So, a limited size dataset can limit strongly the performance of the network. When the dataset is not so large, starting from a pretrained model and then finetuning it, can be the good choice to obtain better results with a small dataset. This is mean to build up a new neural network, taking the pre-trained learned parameters as initialization, and initialize randomly the parameters of the classifier, to classify the correct amount of labels. It is also possible to initialize only few layers of the neural network with the pretrained parameters, and use a random initialization for some of the last layers.
Taking a pretrained model as starting point can solve many resources problems and save time. Having already this good weights initialization, the network should be trained easily, adapting it to classify another database. Among the possible choices about the weights initialization in finetuning a pre-trained network, in this thesis only the weights of the classifier were initialized randomly, and all other layer parameters were initialized with the pre-trained model weights.
Finetuning can be done on only few layers (usually, the last ones). So it is possible with this method to not modify the parameters of the firsts layers, and leave only the last layers to change their parameters to adapt the network for another dataset. In training, MatConvNet allows to choose the learning rate for each layer, so it is possible to set a very low learning rate, or even to zero for the layers that we don't want to change the weight values. On the other hand higher learning rate for the layers, as the classifier, that need to update the weights may be set. In adapting a network for different databases, the minimum adjustment is to train only the classifier, setting its weights randomly and adapting the output to the number of classes of the other dataset. It is important, in training more layers, to set a low learning rate because the weight values should be already near the optimum.
Finetuning more layers should reach better and better results, until you could have some worst case accuracy, because for example the low number of images in the dataset with respect to set millions of parameters of the network. However in this work I did not arrive

to finetune such number of layers of a network to find a worst accuracy results than tuning less layers. Also following the proposal of [13] (Yosinski et al.,2014) it is convenient to not freeze the parameters if the pretrained database was of different kind with respect to the finetune data.
For a comparison between the networks VGG, GoogLeNet and ResNet, in first instance it was preferred to 'freeze' the weights of the layers that we don't want to train, setting their learning rate to zero. This was to evaluate and compare the efficiency as feature extractor of the studied very deep neural networks, also if they have a different CNN structure. When Finetuning a network, 'freezing' most of the parameters should also require less time and memory consumption for the training process.
Some disadvantages of finetuning can be that if the dataset is very small and very similar to the trained database, the network could reach overfitting of the data. It is suggested to train only the classifier, or using a dropout layer at the end of the network to reduce the overfitting. Another point is that, at the state of art, the best accuracy results for a classification problem, if the database is sufficiently large, come in training a network from scratch, with respect to only finetune a pretrained network. So using pretrained models with Imagenet and finetune them for face classification can lead to worse accuracy results.

3.2 Software and hardware resources

For the development of this project, Image Processing Group servers were used to develop and run codes in Matlab and to access shared databases. In particular Matlab 2015b was used, with compiled MatConvNet-beta18 library. The pre-trained models were downloaded from the Matconvnet website [2], and also some codes in training and testing CNNs were taken from the examples of MatConvNet.
For the training/finetuning of CNN models, it was always used one of the GPUs (Table 7) available for the Image Processing Group, selected automatically by the system.

Models	#GPUs	Cores	RAM	Arch	Capability
GeForce GTX 980	2	2048	4GB	Maxwell	5.2
Tesla K20m	1	2496	5GB	Kepler	3.5
GeForce GTX Titan Black	2	2880	6GB	Kepler	3.5
GeForce GTX Titan X	5	3072	12GB	Maxwell	5.2
GeForce GTX Titan Z	2	5760	12GB	Kepler	3.5

Table 7. List of GPUs available in the servers of UPC IPG

Training speed:

The training speed depends not only on the architecture of a network but also on better setup of the training hyperparameters and weights initialization. However I can compare approximately the speed relatively to VGG-16, which is the slower of the three networks to train. Using one Gpu, with MatconvNet, VGG-16 takes 1.5s per iteration. ResNet was a little bit faster, approximately 10% faster than VGG-16. GoogLeNet is the fastest one, approximately the double train speed of VGG-16.

Training memory:

It's quite difficult to say how much memory RAM occurs for a training process, because it can depend mainly on the code implementation. However in this experience, 20 GB of RAM dedicated to the Matlab application is a safe amount for training a network in MatConvNet with a quite huge database. Note that the batch of images have to be read from disk every time, because not all the database can stay in the RAM space in one time. Using one GPU of 4 GB had permitted to select batches of 30 images to be computed on GPU. Using a GPU with higher memory RAM capacity allows to select a higher number for the mini-batch size, and so increase the computational efficiency of the training process.

3.3 Databases

Different databases exist and are used in neural network researchers to compare results and performance of an architecture. So, many of them are free to download directly from a website. Sometimes there is only a text file that contains a list of links to the images, or in other cases, such as for Facebook or Google, the databases are private.
Let's start to see in details the databases that had been considered influence in this project.

3.3.1 ImageNet

ImageNet is an international database used for different kind of purposes. It is used in the yearly competition ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that evaluates algorithms and neural network architectures for object detection and image classification at large scale [1].
The ImageNet database is composed by 1000 classes organized in hierarchy. It is composed by 1.2 million images for training, 50000 for validation and 100000 for testing. The database has not been changed since the year 2012. For the challenge of ILSVRC classification, the performance of the challengers neural network is evaluated in top-1 and top-5 accuracy rate. In top-5, it is considered correct if the correct class is at least in one of a list of five predictions. The challenge uses the top-5 error rate for ranking purposes.
In this thesis, I have used the networks VGG-16, GoogLeNet and ResNet-101, pretrained with ImageNet for classification (1000 categories). Each pretrained network was downloaded from the Matconvnet website. In Table 7 the accuracy on ImageNet declared in the papers is described.

In Table 8 there is instead the accuracy on the ImageNet 2012 validation data of the released models by MatConvNet. The accuracy of the released models is a little higher than the accuracy obtained in the ILSVRC.
In Table 8 there's also a reference of the top-1 error of these CNN models, which is instead the our evaluating error taken in consideration in this thesis for a performance models comparison.

Team	Year	Ensemble	Top-5 error
VGG	2014	7	7,32%
VGG	(not submitted in ILSVRC)	1	7,1%
VGG	(not submitted in ILSVRC)	2	6.8%
GoogLeNet	2014	1	7,89%
GoogLeNet	2014	7	6,67%
Resnet-101	2015	1	4,60%
Resnet	2015	6	3,57%

Table 8. Classification results on ImageNet for a single or ensamble networks. For the Resnet, which had reached the best result, the team had combined six models of different depth to form the ensemble (with two 152-layer networks in the ensemble) [9].

Model	Introduced year	top-1 err.	top-5 err.
Resnet-101-dag	2015	23.4	7.0
MatConvNet-VGG-verydeep-16	2014	28.3	9.5
GoogLeNet-dag	2014	34.2	12.9

Table 9. Values from the Matconvnet website [2] of the performance of these neural network models on the ILSVRC 2012 validation data. The MatConvNet model was trained using MatConvNet (beta17) and batch normalization. The GoogLeNet model performance is a little lower than expected (the model should be on par or a little better than VGG-VD). This network was imported in MatConvNet from the Princeton version of GoogLeNet, not by the Google team, so the difference might be due to parameter setting during training.

3.3.2 Google-face-UPC

Google-face-UPC is a face database available at the Image Processing Group, Universitat Politècnica de Catalunya, Barcelona. I have manually check the errors present in this database of 263 identities, since it was made by an automatic algorithm of searching, downloading and cropping images on Google search. At the end of this check, I have divided the database into 'training and validation', which is composed by 13056 images, and 'test', which is composed by 6288 images. This test set, is also harder to classify correctly because contains many face obstructions.

The number of images per person in this database is not constant, so this number can vary from 20 images for one person to 90 images for another one; approximately there is an average of 50 images per person. It isn't used any type of face alignment; the images of faces were only cropped form original images. There is a lot of variability in these face images such as different face positions, facial expressions and different lighting.

(a) (b)

Fig.22 Cropped face examples in Google-face-UPC database. The cropped images include only the face and not some other parts of the human head, such as hair, ears, chin. (a) Single images of different people. (b) Face images of one person in the database. There are a lot of variability in positions and facial expressions.

3.3.3 FaceScrub

FaceScrub is one of the largest public database of faces. It contains 106863 photos of 530 celebrities, 265 whom are male (55306 images), and 265 female (51557 images) with about 200 images per person [10]. Available full frame and cropped version in UPC servers of IPG.

The images were taken from Internet, and finally checking manually the errors. So it is a very 'free from errors' face database respect to others, and all images were taken from real situations, and so uncontrolled conditions. If it is necessary, name and gender annotations are included.

Fig.23 Face examples in FaceScrub database

3.3.4 YouTube Faces

The YouTube faces data set [11] contains 621126 face pictures of 1595 identities, taken from 3,425 YouTube videos.
Images come from videos so there is not a lot of variability between them for each person, and also images are not in very high resolution. There is an average of 2.15 videos for each person from which the frames are extracting. The minimum amount of images for a person is 48, and the maximum is 6070. The average quantity of images for each one is 181.3 images.

Fig.24 Youtube Faces examples images

Number of videos	1	2	3	4	5	6
Number of people	591	471	307	167	51	8

Table 10. Number of videos per person in YouTube faces

In this project, cropped faces from YouTube Faces database are used for training GoogLeNet[5].

[5] Section 4.7

4 Results

In this Chapter, I will report the results I have obtained in face classification using the three very deep convolutional neural networks VGG-16, GoogLeNet and ResNet-101. In the Appendices 2 to 7 there is the Matlab code necessary for the training in MatConvNet.
First I will report finetuning results using the Google-face-UPC database with the VGG-face pretrained model. I start first in training only the classifier, and then finetune more layers of the networks to see the improvements. Then there is a test of combining VGG-face with inception modules, and after this an experiment of combine VGG-face with GoogLeNet-ImageNet to try to find some improvements in face classification with these models. After this, there are the results in finetuning also VGG-16, GoogLeNet and ResNet-101 ImageNet pretrained models using Google-face-UPC and Facescrub.
The option parameters, such as the learning rate, momentum, weight decay were chosen first referencing to the training parameters described in the respectively papers of the models, and then by observing the training process iterations, to find better hyper-parameters to train.
To initialize the weights, I always use the He initialization, that is good in networks that use the ReLU activation function. With this initialization, weight values are taken from a random normal distribution, multiplied by a constant that depends on the square root of two over the input size of the kernel of parameters of each layer. In evaluating the accuracy, we use the correct predictions over a test set of images; note that the test accuracy in Google-face-UPC was always a little lower (1-1.5%) than the validation. We usually evaluate the test phase only with the model that has shown best result in the same trial, so sometimes we refers first to the validation error, and then evaluate the test on the best model. For deriving the validation accuracy we take complementar value of the validation error.

4.1 Finetune VGG-face using Google-face-UPC

The network VGG-face is a VGG-16 model pretrained with a face dataset of 2622 identities, 2,6 millions of images [12] manually checked. It was trained by the Visual Geometry Group on NVIDIA Titan Black GPUs with 6GB of memory, using four GPUs together, taking seven days. This model was developed to reach the state-of-the art results in face identification using VGG. In comparison with the models of FaceNet and DeepFace, VGG-face uses a smaller database than state-of-the art models but regardless of this, it has competitive results with Labeled faces in the wild (LFW) and Youtube Faces database.
In this section we have used this VGG-face pretrained model for finetuning using Google-face-UPC. As training setup it was set a fixed learning rate of 0.001 and zero momentum. It was tried to finetune from different layers in different trials, from the bottom one (the FC classifier) to the layer conv5_1, and evaluate the test accuracy in each trial (Table 11).

Finetuning Layer	Learn_rate/W_decay/momentum	Test Accuracy (%)
FC Classifier	0.001 /0.0002/ 0	89.23
FC conv7	0.001 /0.0002/ 0	89.04
FC conv6	0.001 /0.0002/ 0	89.68
Conv5-3	0.001 /0.0002/ 0	90.62
Conv5-2	0.001 /0.0002/ 0	90.82
Conv5-1	0.001 /0.0002/ 0	91.38

Table 11. Accuracy results in finetuning VGG-face using Google-face-UPC

Training graphs:

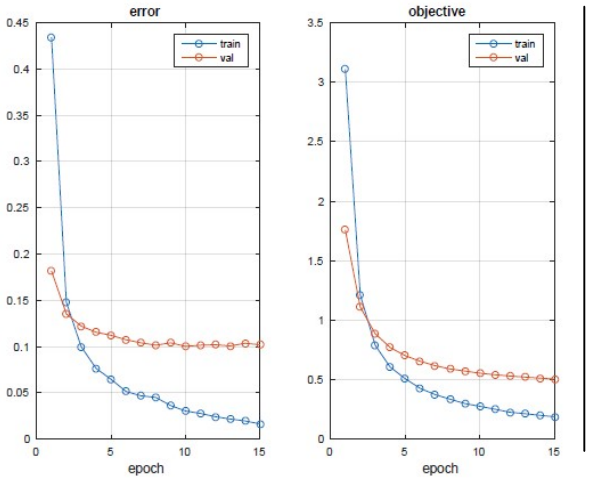


Fig.25 Finetune FC Classifier

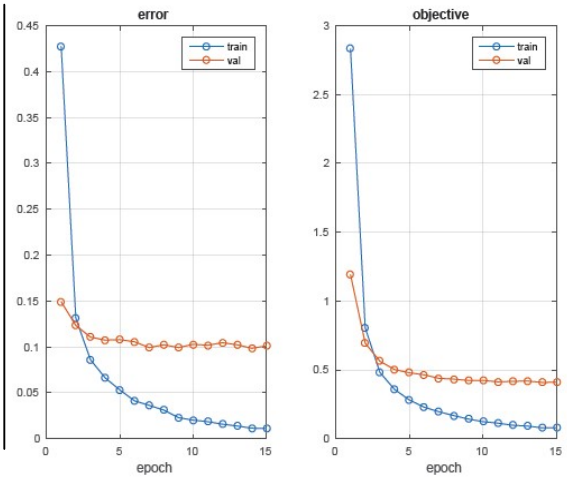


Fig.26 Finetune from layer conv7

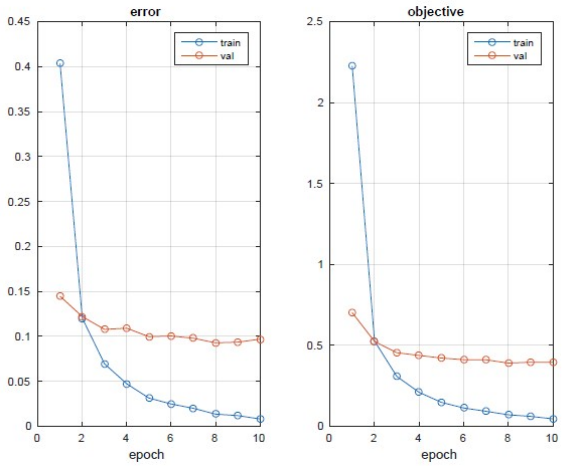


Fig.27 Finetune from layer conv6

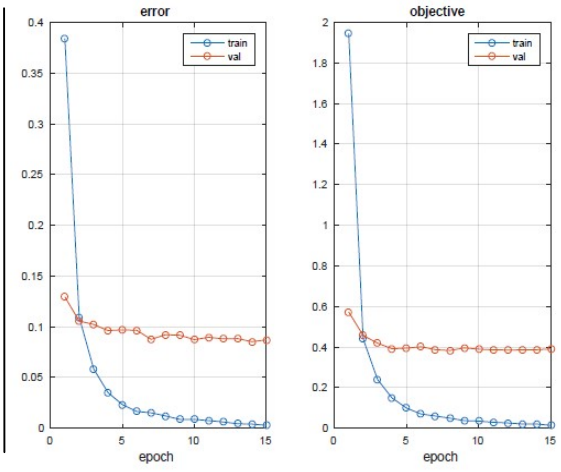


Fig.28 Finetune from conv5_3

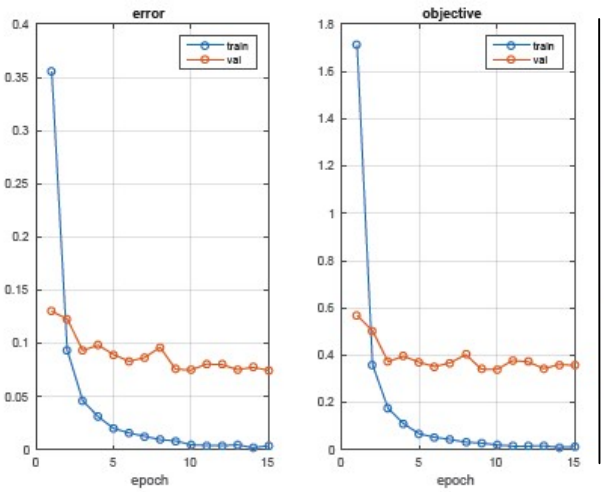


Fig.29 Finetune from conv5_2

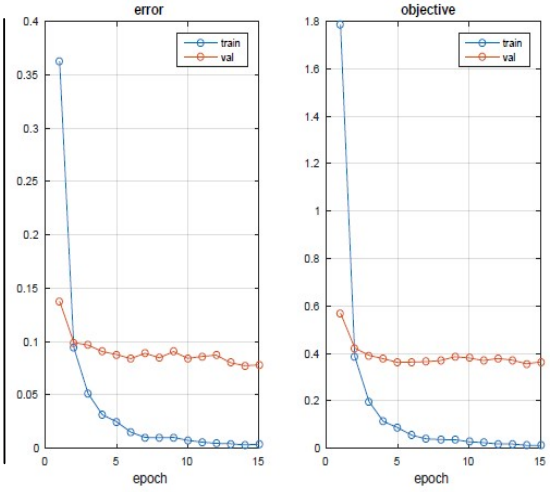


Fig.30 Finetune from conv5_1

Considerations about graphs:

Finetuning layers from more hidden ones, makes a faster convergence and a better fitting of training data. In particular, looking at these graphs, the objective in the first epoch starts from lower points, and reach the zero after 15 epochs. Instead, finetuning only lastest layers, it take more epochs for the loss to go to zero. However the validation error is approximately constant during the training along the epochs, and decreases its value significantly only in training more hidden layers.

4.2 Combine VGG-face with inception modules

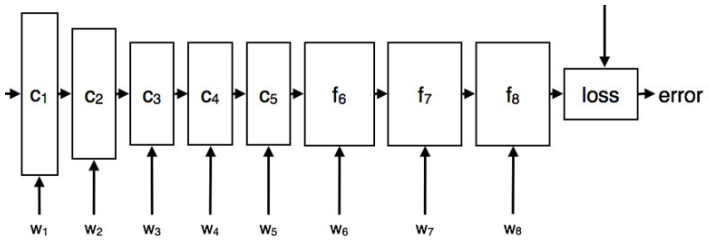


Fig.31 VGG-16 Convolutional Layers[6]. f8 is the classifier, which output is used to calculate the loss in the training process. For my experiment I used VGG-face pretrained in Matconvnet with 2622 face identities.

The VGG-face performance:

To compare the difference in performance between leave last the convolutional layers or substitute some of them with inception modules I want first to evaluate the classical VGG architecture performances. So, in first instance, I focused in the three fully connected convolution layers f6, f7 and f8 making a training of them using the Google-face-UPC

[6] Image from http://www.robots.ox.ac.uk/~VGG/practicals/cnn/

database, and leave fixed the pretrained parameters of the other layers. The parameters of layers f6, f7, and f8 were initialized randomly. The test accuracy after the training was of 92,78%.

First experiment:

Now, knowing the performance of VGG-face with Google-face-UPC database, it was tried to replace the convolutional layers f6 and f7 with two Inception modules. This require no padding adjustment because the output from C5 (fig.32) is a 7x7 feature map, and the last two ICP modules in GoogLeNet receive exactly the same input dimension. So only adjust the depth of the features maps and apply the Inceptions to VGG-face architecture.

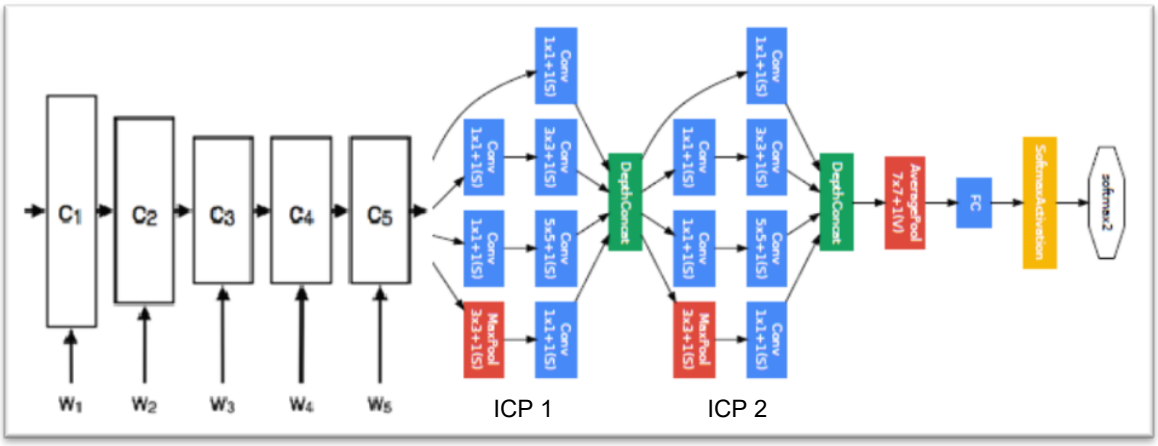


Fig.32 VGG-face modified with two Inception modules at the end of the structure

The choice of the number of feature maps of the layers in the two inception modules was based taking as reference the last two layers in GoogLeNet (ICP8, ICP9), respecting the relationship between the input and output number of feature maps[7].
The original last three convolutional layers of VGG-face were f6, f7 and the classifier, so we can evaluate the amount of parameters of these layers. The exact amount of learning parameters, including the classifier for the 263 identities of Google-face-UPC is described in Table 12.

Layer	Size	#Params
f6	7x7x512x4096	102.760.488
f7	1x1x4096x4096	16.777.216
Classifier	1x1x4096x263	1.077.248

Table 12. Amount of learning parameters of last three layers in VGG-face

Replacing f6 and f7 with two inception modules we have a **30 times less** learning parameters, described in Table 13.

[7] See Appendix 3 for the inception modules code implementation.

Layer	#Params
Icp1	1.156.232
Icp2	2.586.511
Classifier	321.386

Table 13. Amount of parameters of two inception modules in this experiment of combine VGG-face with inception modules.

In the training, the two inception modules were initialized randomly, and use a weight update with momentum 0.9 and learning rate of 0.001. The layers from C1 to C5 were initialized with the pretrained VGG-face weights, and set a zero learning rate, freezing these parameters.
At the test phase, this VGG with Inceptions have an accuracy of **87,31%**, with 5490 correct predictions, and 798 errors; it was therefore 5,47% less than the accuracy reached by the standard VGG-face architecture. This can be due to a poor choice of the number of feature maps in the inception modules, that lead to few learning parameters, so, in a second experiment, it was increased the number of feature maps in the inception modules to see if there was an improvement.

Second experiment[8]*:*

In increasing the number of feature maps of the ICP modules, it is expected to have better results than the previous experiment, or at least the same results.

Layer	#Params
Icp1	1.146.880
Icp2	6.356.992
Classifier	471.296

Table 14. Parameters amount in second experiment

The training graph[9] in this second experiment shows that it is easier for the system to find a convergence, but however the validation error at the end of training is about 0.110, so it was approximately the same as in first experiment. In every case the original VGG-face has lower validation error (0.1), and so better accuracy. In conclusion, using inception modules as ending layers of VGG works, has lower learning parameters, but has higher error.

4.3 Finetune GoogLeNet-ImageNet using Google-face-UPC

This section is about finetuning GoogLeNet model, pretrained with ImageNet, released from Princeton, and imported to MatConvNet by the MatConvNet group. This time for finetuning it, Google-face-UPC database was used with flipping image augmentation. It was first finetune only the classifier, and then went up in the network, training every Inception module to find better results. It was used different learning rates (0,01-0,001)

[8] See Appendix 4 for the code implementation of the second experiment

[9] Training graph in Annex 2.

along the epochs in this experiment to see the behaviour of the training process in changing this hyperparameter, and try to reach better results; no momentum was used.

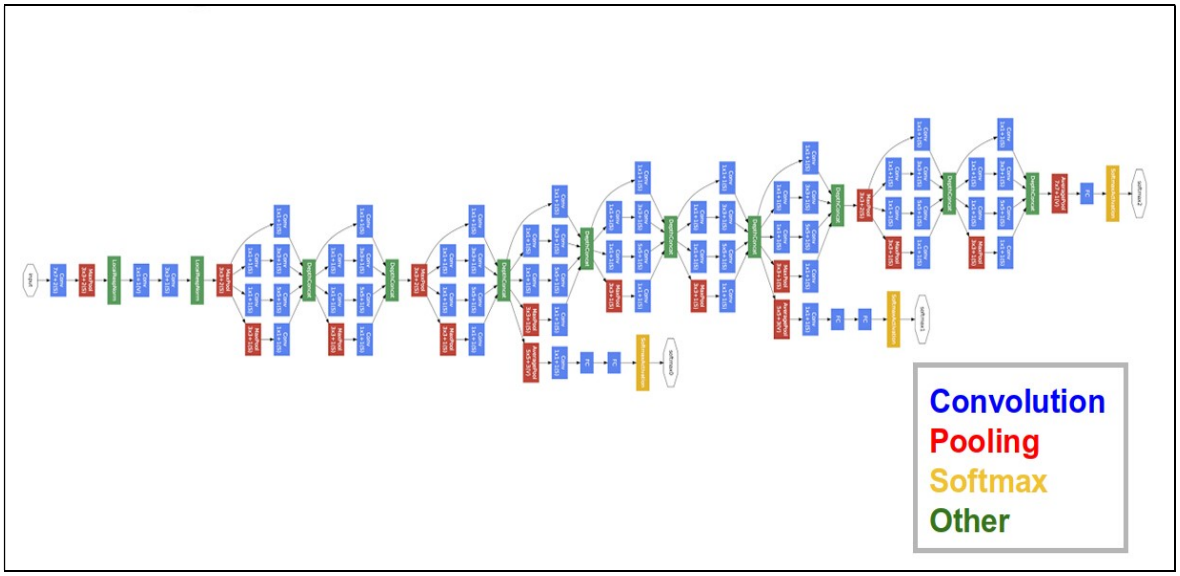


Fig.33 GoogLeNet architecture, 9 inception modules

Finetuning only the classifier of GoogLeNet-ImageNet using Google-face-UPC didn't converge.it requires probably a lot of epochs and an appropriate learning rate.
Finetining from Inception module 9 (ICP9) was better and took 40 epochs to reach convergence. The validation accuracy is still not so good because the validation error is around 0.44.
Finetuning from ICP7 shows better results in validation error reaching a value of 0.33 when the loss converges.
Finetuning from ICP4 has a better validation error (around 0.25) after 18 epochs, so a validation accuracy of **75%** with the Google-face-UPC database.
We can expected that finetuning more inception modules can reach a lower validation error. All the training graphs with the respective learning rate setting are in the Annex 3.

Finetuning layer	Validation error
Classifier	0.6
ICP 9	0.45
ICP 8	0.43
ICP 7	0.33
ICP 6	0.31
ICP 5	0.28
ICP 4	0.25

Table 15. It show the validation error in finetuning from different Inception modules of GoogLeNet-ImageNet using Google-face-UPC.

4.4 Finetune the ensemble VGG-face + GoogLeNet-ImageNet

Different networks can learn different features, so maybe concatenate them can result in better classification. Here there is an experiment of combine two different networks, both finetuned previously with the database Google-face-UPC.

Network used:

- VGG-face finetuned from layer conv5_1
- GoogLeNet finetuned from ICP4

The joining of this two architectures is made by concatenate the outputs of both neural networks, adding one fully connected layer and a dropout of 0.5, and finally the classifier with 263 outputs (fig.34). A flip data augmentation is applied to the images. The mini-batch size was of 10, to fits images in the GPU Tesla K20m. The learning rate was 0.001 for first 10 epochs, and then 0.0001 for other 8 epochs.

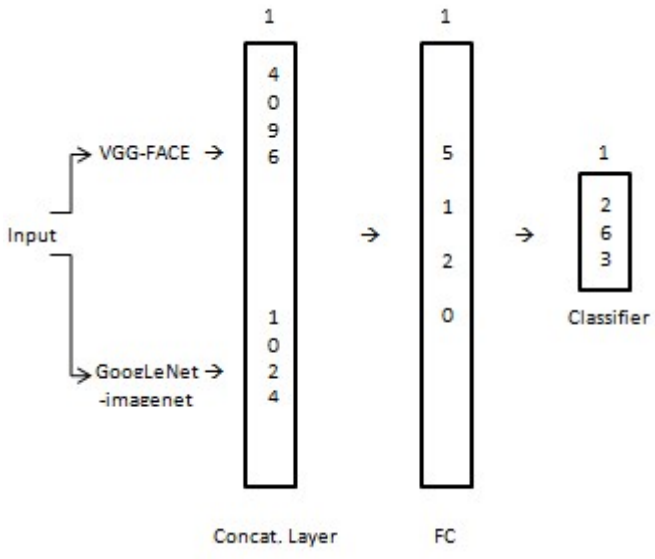


Fig.34 Concatenate VGG-face and GoogLeNet-ImageNet

From the training graph (Fig.35) we can see a validation error in VGG-face+Googlenet-ImageNet of about 0.066. Instead, the only VGG-face finetuned from layer conv5_1 has a validation error about 0.075 (Fig.36). So there's a slightly better performance of 0.009 in concatenating the two very deep CNN.

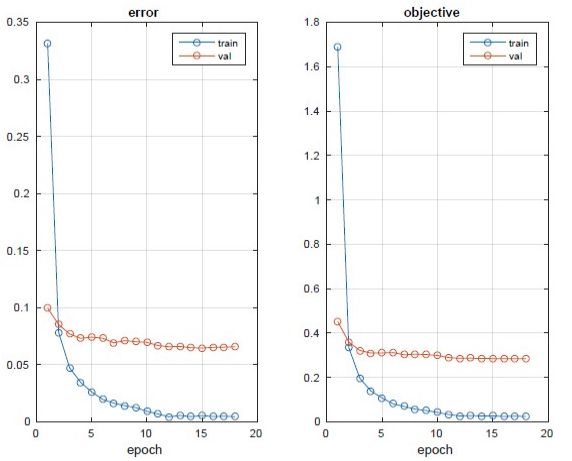


Fig.35 Finetune graph VGG-face+GoogLeNet

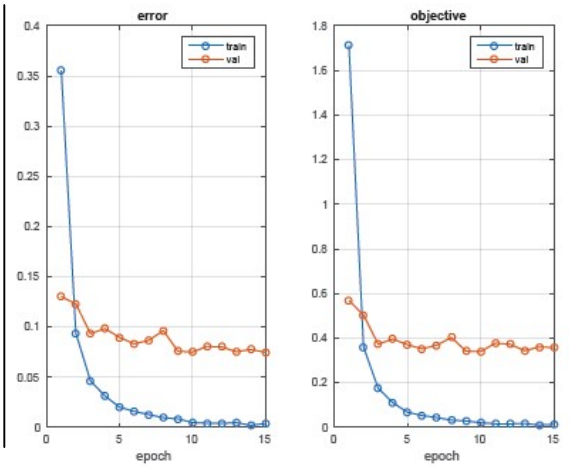


Fig.36 Finetune graph VGG-face from conv5_1[10]

4.5 Finetune VGG-ImageNet using Google-face-UPC

Using the pretrained model VGG-16 with ImageNet, we want to finetune it using the face database Google-face-UPC to compare the performance of finetuning this architecture with the other two very deep network GoogLeNet and ResNet-101. Also here, the starting point is to finetune the classifier, and then more layers.

Finetuning only the Classifier with learning rate of 0.01 and momentum 0.5 doesn't make a good training process; also after 30 epochs the Loss was far away from convergence. Maybe a higher learning rate could be set if the system doesn't diverge, or use more processing epochs to find a better performance. The test accuracy experiment in finetuning the classifier was of 51,42%.

Setting a momentum of 0.5 was good to accelerate the learning process because with momentum equal to 0 the system doesn't train well using this database and pretrained model. Setting momentum of 0.9 makes the loss to diverge, so an intermediate setting was good to accelerate the process of finding a minimum loss.

Finetuning block L5 gave a validation error of 0.21 after 20 epochs, so it was expected near 80% of test accuracy.

Finetuning from layer L4_1 gave a good test accuracy of **83,13%**, as it is shown in Table 16.

Finetuning layer	Validation error	Test accuracy
Classifier	0.47	51,42%
Conv5_1	0.21	-
Conv4_1	0.16	83,13%

Table 16. Errors and accuracy performance in finetuning VGG16-ImageNet using the database Google-face-UPC.

[10] From Section 4.1, Figure 30

Considerations:

VGG-16 architecture is good in feature extraction, and can reach high accuracy also with different datasets, finetuning a pretrained model with another database. Its disadvantages are the high number of learned parameters and Sum-of-Products. That is, training and also the inference require high computational memory and high computational time respect to GoogLeNet.

4.6 Finetune ResNet-101-ImageNet using Google-face-UPC

After several experiments to set the learning rate in the batch normalization layers of ResNet-101, a finetuning experiment with Google-face-UPC database finally had success reaching good results. Finetuning only the classifier with this architecture gave lower validation error than with VGG-16 or GoogLeNet (Table 17).

Finetuning Classifier	
Model	Validation error
ResNet-101	0.39
VGG-16	0.46
GoogLeNet	0.6

Table 17. Comparison results in finetuning the classifier of the architectures ResNet101, VGG-16 and GoogLeNet pretrained with ImageNet, using Google-face-UPC.

When finetuning of the last three Residual Blocks a minimum validation error of 0.325 was reached after 30 epochs. The learning rate used for finetuning was of 0.005 in the first 20 epochs and then 0.001 for the following 10 epochs. A momentum of 0.7 was used. Finally in finetuning all the layers of the network ResNet-101 (using Google-face-UPC) with pretrained weights initialization, using ImageNet, an **82,35%** of test accuracy was obtained (Table 18).

Finetune layers	Validation error	Test accuracy
Classifier	0.39	-
Last 3 residual blocks	0.325	67,10%
All layers	0.17	82,35%

Table 18. Validation error and test accuracy in finetuning ResNet-101-ImageNet

4.7 Train GoogLeNet 6 ICP-BN using Youtube Faces

Searching for a GoogLeNet pretrained with faces on the Internet, there wasn't any model of this type. It exists in literature only FaceNet [17] that uses the GoogLeNet architecture trained with hundreds of millions of images for the face identification problem, but this is not a free released model. So, to evaluate the problem of training from scratch a very deep convolutional neural network, a smaller GoogLeNet, with six Inception layers (fig.37) was trained from scratch using the YouTube faces database, that is one of the largest database public released. The idea was first to train the network, and then finetune it using Google-face-UPC images.

At the immediately starting point I have seen that the standard structure of GoogLeNet, also simplified by taking only 6 inception layers, was far from making the loss to decrease, also because of the initialization of the parameters which is quite critical in training from scratch. Modifying the architecture by inserting the Batch Normalization layers finally helped a lot for a successful training.

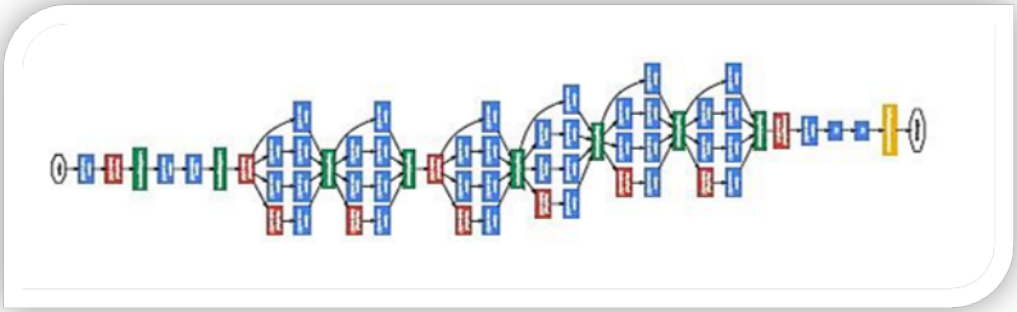


Fig.37 Googlenet 6 ICP architecture

Therfore the network architecture is the same structure of GoogLeNet, only taking the first 6 Inception modules, and adding batch normalization after each convolutional layer.
The Youtube-faces database was divided into 497493 training images, 12456 validation images and 111.100 test images, selected randomly. Single cropped face images were used for the training.
A mini-batch of 100 images, a learning rate of 0.01 and momentum of 0.9 was used. We obtained the error for 10 epochs. The results were satisfactory also choosing fewer epochs. To train ten epochs on Googlenet_6ICP with Yotube-faces it took two days using one GPU GeForce® GTX TITAN Black.

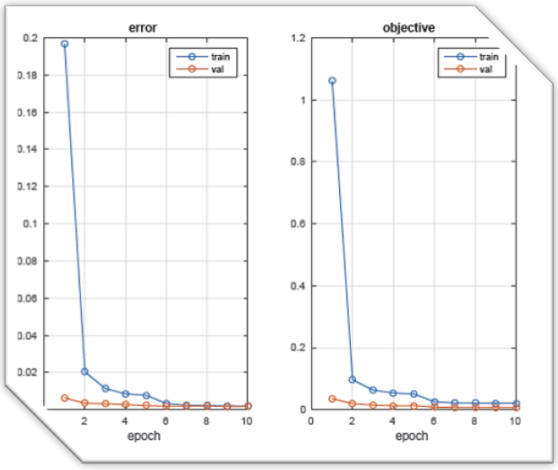


Fig.38 Training graph GoogLeNet 6ICP with Youtube faces

The validation error in figure 38 had reached a bottom of 0.003 after two epochs, so it was expecting a test accuracy around more than 99%. In fact, the test results of this model at the end of the training was of **99,867%** of accuracy, with only 148 errors in the whole test set composed by 111.100 images.
Afterwards, his model was finetuned it using Google-face-UPC.

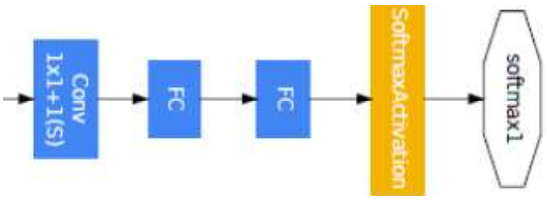


Fig.39 Last three layers in Googlenet 6ICP that will be finetuned in first instance using Google-face-UPC

Finetuning the layers in fig. 39 gave a validation error of 0.45, so the same as finetuning GoogLeNet-ImageNet from icp9.
Finetuning this GoogLeNet 6ICP BN from ICP4 reach a minimum validation error of 0.3, with 69,23% of test accuracy. So, no improvement was reached by this trained network with respect to finetuning the pretrained model GoogLeNet-ImageNet. This is probably because images in the Youtube faces database have few variability, so easy for the network to classify, but not to make a model that can generalize well with other databases.

Considerations:

As we have seen before in Section 4.3, with the same Google-face-UPC database and with Googlenet 9ICP, finetuning more inception modules is good to obtain better results, but not to reach results as a GoogLeNet trained from scratch. This is probably because this architecture distribute more the main characteristics of a database along all the network. In particular, if we focus on a single inception module, its output is made by a simple concatenation of outputs from a 1x1 conv, 3x3 conv of a 1x1 conv, a 5x5 conv of a 1x1 conv, and a 1x1 conv of a 3x3 maxpool (see Fig.15). Concatenate these outputs means to concatenate feature maps belonging to different layers. Stacking the inception modules causes to this system a more complex propagation of activations and so maybe this system adapts a little better the weights to the trained database with respect to VGG

or ResNet. To have a very good model in GoogLeNet that can generalize well and can be used in large scale for face identification, a very big dataset of images to train this model is necessary.

4.8 Finetune ResNet-ImageNet using FaceScrub

To finetune FaceScrub, the database was divided into training, validation, and test images. In particular there were 73369 images for training, 4477 images for validation, and 13866 images for testing. The total amount of images were a little less than the amount declared in section 3.3.3 maybe due to downloading images problems.

A first test in finetuning the classifier of ResNet-101 freezing pretrained parameters using FaceScrub has shown better results than with Google-face-UPC. After 20 epochs the validation error was about 0.25 as shown in Table 19. The huge amount of images of Facescrub makes this training very slow with only one GPU of 4 GB, so it was decided to stop the training after 20 epochs (the accuracy could be further be better) and start the finetuning this time without freezing all the parameters, but setting a learning rate of the hidden layers to 0.1 times the learning rate of the classifier, and so allow all the parameters to change a little bit to reach good results.

Database	Validation error
Google-face-UPC	0.39
FaceScrub	0.25

Table 19. Comparison of validation error in training ResNet-101 classifier with freezing parameters, except for the classifier, using Google-face-UPC and Facescrub.

Finetuning ResNet-101 without freezing parameters had a better behave in the training, and finally the validation error reached a value of 0.095. The test accuracy was of **91,06%**.

4.9 Finetune VGG-ImageNet and GoogleNet-ImageNet using Facescrub

To make a comparison with the results of ResNet in finetuning FaceScrub, the same setting was made. So initialize last layer randomly, and other layers with pretrained weights. The learning rate of the classifier was set to 0.005 and then 0.001 after some epochs. Other layers had a learning rate 0.1 times the learning rate of the classifier.
Also with Facescrub, VGG-16 was still the network with less validation error, also less than Resnet, and reached a validation accuracy of 94,8% (Table 20). With this experiment we can see that using a larger database as Facescrub allows to obtain better results in finetuning starting from a pretrained Imagenet model. The gain was about +10% of accuracy, with respect to using the database Google-face-UPC.

Model	Validation Accuracy (%)
VGG-16	94,8
GoogLeNet	90,5
ResNet-101	91,5

Table 20. Results in finetuning using FaceScrub

4.10 Finetune GoogLeNet-ImageNet with different learning rates

After seeing several experiments, we have shown always a better improvement in finetuning more hidden layers. The difference from using freezing pretrained parameters or leave instead more and more the parameters to change during the training was evaluated. For evaluate this comparison, we took GoogLeNet pretrained with ImageNet which was the faster network to train, but also the worst in accuracy performance with freezing parameters.
Google-face-UPC database was used here for finetuning. We called CFLR the Classifier learning rate, which was set for the training to 0.005 for 15 epochs and then to 0.001 for others 15 epochs.

Learning rate hidden layers	Validation error
0	0.64
0.003*CFLR	0.53
0.1*CFLR	0.29
CFLR	0.2

Table 21. Validation error in increasing the learning rate of all the layers of GoogLeNet, respected to the learning rate of the classifier (CFLR).

Leaving the weights of all the layers able to change, allows to obtain better results instead of freezing the parameters. The Test accuracy over 6288 images was finally of **79,52%**.

4.11 Finetune all layers of VGG-face using Google-face-UPC

In Section 4.1 we stop finetuning VGG-face from the convolution layer 5_1. Now we would like to evaluate what happen in finetuning all the layers using Google-face-UPC. I set the learning rate this time to 0.005 for 20 epochs, then 0.001 for 10 epochs.
In the training, the validation error decreased further from the experiment in finetuning from conv5_1, and at the test phase over 6288 images this model gave a **93,62%** of accuracy.
This is the best result obtained in finetuning Google-face-UPC database with very deep models. The gap here in accuracy in training only the classifier is about 4.4%. The gap with the pretrained ImageNet model is much larger. There is an amount of about 10% of accuracy gap between the best finetuned VGG-face model, with respect to the model finetuned taking the pretrained ImageNet VGG-16 model (from Table 24).

4.12 Summary of results

In this Section we sum up the accuracy[11] results obtained with VGG, GoogLeNet and ResNet.

Models	Database	Finetuned layer	Test Accuracy (%)
VGG-face	Google-face-UPC	L5_1	91,38
VGG-16-Imagenet	Google-face-UPC	L5_1	~79

Table 22. Comparison of performance in finetuning VGG-face, VGG-face with Inception modules and VGG-Imagenet using Google-face-UPC, finetuning the parameters from layers L5_1 or L6, and freezing the parameters of the other layers.

Models	Database	Finetuned layer	Initialization parameters	Test Accuracy (%)
VGG-face	Google-face-UPC	L6	Pre-trained	89,68
VGG-face	Google-face-UPC	L6	Random	92,78
VGG-face with ICP	Google-face-UPC	L6	Random	87,31

Table 23. VGG models finetuned with Google-face-UPC, using a different parameter initializations for the finetuned layers. The other layers of the networks were initialized with pretrained weights and freezing them during the training.

[11] In Test accuracy we evaluate the models as top-1 correct predictions in a test set. The percentages with the symbol ~ take reference to the validation accuracy, which is usually a little higher (usually +1%) in the Google-face-database with respect to the test accuracy.

Models	Database	Finetuned layer	Test Accuracy (%)
VGG-face	Google-face-UPC	Classifier	89,23%
VGG-face	Google-face-UPC	All layers	93,62%
VGG-16-Imagenet	Google-face-UPC	L4_1	83,13%

Table 24. Test accuracy comparison on finetuning VGG using Google-face-UPC. Training only the classifier of VGG-face obtains higher accuracy than finetuning more layers of VGG-Imagenet.

Models	Database	Finetuned layer	Test Accuracy (%)
VGG-face	Google-face-UPC	L5_1	91,38
GoogLeNet-Imagenet	Google-face-UPC	ICP4	~ 75
VGG-face+GoogLeNet	Google-face-UPC	FC+Classifier	92,49

Table 25. This table shows the test accuracy results in finituning the single models of VGG-face and GoogLeNet-Imagenet, with respect to combine them and train an added fully connected layer and a classifier.

Models	Database	Finetuning layer	Test Accuracy (%)
GoogLeNet-Imagenet	Google-face-UPC	ICP4	~ 75
GoogLeNet-6ICP-BN-YouTubeFaces	Google-face-UPC	ICP4	69,23

Table 26. A GoogLeNet-Imagenet model and the GoogLeNet-6ICP-BN –YoutubeFaces model are compared in finetuning using the Google-face-UPC database.

Models	Database	Test Accuracy (%)
VGG-face	Google-face-UPC	93,62
VGG-16-ImageNet	Google-face-UPC	83,13
GoogLeNet-Imagenet	Google-face-UPC	79,52
ResNet-Imagenet	Google-face-UPC	82,35
VGG-face+GoogLeNet	Google-face-UPC	92,49[13]

Table 27. Maximum accuracy obtained in finetuning different models using Google-face-UPC.

[13] This results was done taking not the best VGG-face and GoogLeNet-Imagenet finetuned models, so the test accuracy can be higher if those models were used.

Models	Database	Validation Accuracy (%)
VGG-16-ImageNet	FaceScrub	94,8
GoogLeNet-Imagenet	FaceScrub	90,5
ResNet-Imagenet	FaceScrub	91,5

Table 28. Results in finetuning all hidden layers at 1/10 of the learning rate of the classifier

5 Budget

To evaluate the budget, it is possible to make a simulation of the computational cost of the work using the Amazon Elastic Compute Cloud (Amazon EC2). In this cloud it is possible to use a 4 GB GPU selecting one of the instance of G2. The prize is calculated in €/hr. The price increases if the software used is not open source, such as Matlab.

G2 Instance on Amazon EC2

Modello	GPU	vCPU	Memoria Ram(GiB)	Storage SSD(GB)	Prize
g2.2xlarge	1	8	15	1x60	$0.65/hr
g2.8xlarge	4	32	60	2x120	$2.6/hr

I didn't find a Matlab price on the http://aws.amazon.com/, so I've searched for another software platform for training a neural network with GPU, like it is Caffe. These are the prices of usage for software and hardware with Caffe with a GPU instance:

g2.2xlarge	**$0.716/hr**
g2.8xlarge	**$2.86/hr**

The amount of hours of this project is about **650 hours**, so we can estimate the computational price in EC2 Cloud in **468 €.** To this amount of hours I have to add the time needed for developing codes and error checking in the image databases, measured in approximately 50 hours. Then, one hour weekly meeting with a senior engineer for 20 weeks.

	€/hour	Work hour	Price(€)
EC2 Amazon cloud	0.72	650	468
Engineer junior	12	720 h	8640
Engineer senior	50	20 h	1000
		Total	10.108

6 Conclusions and future development

After this experiment with VGG-16, GoogLeNet and ResNet-101, we can say that, at the state-of-art, the results of finetuning are still quite dependent by the training database model. Essentially, if you want to reach high accuracy (more than 90%) in face classification, you have to finetune a pretrained-with-faces images network model or to train from scratch these deep CNNs with a large database. Use an ensemble of convolutional neural networks, also of different models, can help to reach better results as the experiment VGG+GoogLeNet has shown.
VGG-16 model, pretrained with faces, or also pretrained with ImageNet, reached the best results using Google-face-UPC. So, this more classical structure of subsequent convolution layers, is maybe a little better than GoogLeNet and ResNet as features extractor. GoogLeNet-ImageNet is also good as features extractor, but with lower accuracy than VGG-16, especially when a freezing parameters during finetuning. Over the three CNNs only VGG-16 and ResNet-101, pretrained with ImageNet, have reached more than 80% of accuracy in face classification with Google-face-UPC.
The accuracy in finetuning a pretrained model with ImageNet and a pretrained model with faces, using Google-face-UPC has still a gap of at least 10%: finetuning VGG-face has obtained a maximum test accuracy value of 93.62%, while VGG-16-Imagenet stopped at 83.13%. This gap is reduced if we use a different and larger database such as FaceScrub, where VGG-16 pretrained with Imagenet had obtained a high 94.8% of accuracy on validation data.
GoogLeNet remain the fastest very deep CNN. Its speed at inference and low memory consumption makes this network the fastest and with less parameters very deep neural network. Training a GoogLeNet with six inception modules and batch normailization, using YouTube Faces has reached the higher results of 99.86% of test accuracy, but this database has few face images variability to recognise really a person ‘in the wild’, so it was only an experiment about training a GoogLeNet model from scratch. In literature there is the paper FaceNet system [17], proposed by Google group, that uses GoogLeNet for face verification. They obtained a 99.63% of accuracy with Labeled Faces in the Wild (LFW) and 95.12% on YoutubeFaces.
ResNet can be sometimes quite tricky in training setting to find the best results due to the high number of hyperparameters that can be set in this architecture, considering also the batch normalization. However, ResNet has shown results close to VGG-16, with a more easy optimization of weights, looking at the training graphs. Nevertheless VGG-16 has shown during all this thesis the best accuracy results in face classification.

Future development

Other explorations about these three very deep CNN can be done to try to obtain better results. For instance, combining the results on multiple crops of the face images could increase the accuracy in face classification. Another possibility could be to concatenate vectors from VGG-face and ResNet-ImageNet, because ResNet has better results than GoogLeNet. Improvements in training, for instance collecting a larger face database for training GoogLeNet could also increase the classification accuracy.

Bibliography

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, S. Huang, A. Karpathy, A. Khosla,M. Bernstein, A.C. Berg, and F.F. Li. *ImageNet large scale visual recognition challenge.* IJCV,2015.

[2] The MatConvNet Team. http://www.vlfeat.org/matconvnet/pretrained/

[3] Dominik Scherer, Andreas Muller, Sven Behnke. *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*. 20th International Conference on Artificial Neural Networks (ICANN), Thessaloniki, Greece, September 2010.

[4] Sergey Ioffe, Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.*

[5] Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). *Deep sparse rectifier neural networks.*

[6] Andrea Vedaldi, Karel Lenc. *MatConvNet - Convolutional Neural Networks for MATLAB*. ArXiv, 15 Dec 2014 (v1), last revised 5 May 2016

[7] Karen Simonyan, Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition.*

[8] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. *Going Deeper with Convolutions.*

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition.* ArXiv:1512.03385v1 [cs.CV] 10 Dec 2015

[10] http://vintage.winklerbros.net/facescrub.html

[11] Lior Wolf, Tal Hassner and Itay Maoz. Face Recognition in Unconstrained Videos with Matched Background Similarity. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), 2011.

[12] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman. *Deep Face Recognition.* http://www.robots.ox.ac.uk/~VGG/data/VGG_face/

[13] Yosinski J, Clune J, Bengio Y, and Lipson H. *How transferable are features in deep neural networks?* In Advances in Neural Information Processing Systems 27 (NIPS '14), NIPS Foundation, 2014.

[14] Nitish Srivastava Geoffrey Hinton Alex Krizhevsky Ilya Sutskever Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of Machine Learning Research 15 (2014)

[15] Min Lin, Qiang Chen, Shuicheng Yan. *Network In Network*. ArXiv, 4 Mar 2014.

[16] Sanjeev Arora, Aditya Bhaskara, Rong Ge, Tengyu Ma. *Provable Bounds for Learning Some Deep Representations.* Arxiv, 23 Oct 2013.

[17] Florian Schroff, Dmitry Kalenichenko, James Philbin. *FaceNet: A Unified Embedding for Face Recognition and Clustering.* ArXiv, 17 Jun 2015.

[18] Kunihiko Fukushima. *Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position*

[19] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, Lior Wolf. *DeepFace: Closing the Gap to Human-Level Performance in Face Verificatio.* Conference on Computer Vision and Pattern Recognition (CVPR) June 24, 2014.

Appendices

1. MatConvNet building blocks instantiation

To build a neural network in MatConvNet using a DagNN wrapper, first instantiate one empty network with:

```
net = dagnn.DagNN() ;
```

then use addlayer and build the network layer by layer, setting all the options for each one. Here some layers instantiation:

CONVOLUTIONAL LAYER

```
net.addLayer('name', dagnn.Conv('size', [11 11 3 96], 'hasBias', true, 'stride', [4, 4], 'pad', [0 0 0 0]), {‘input’}, {‘output’},  {'paramsfilter'  'paramsbias'});
```

RELU ACTIVATION FUNCTION

```
net.addLayer(' name ', dagnn.ReLU(), {' input'}, {' output '}, {});
```

POOLING

```
net.addLayer(' name ', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 0 0 0]), {'input'}, {' output '}, {});
```

BATCH NORMALIZATION

```
net.addLayer(' name ', dagnn.BatchNorm('numChannels', Numchannels), {' input '}, {' output '}, {'paramf', 'paramb', 'param_m'});
```

LOCAL RESPONSE NORMALIZATION

```
net.addLayer(' name ', dagnn.LRN('param', [5 1 2.0000e-05 0.7500]), {' input '}, {' output '}, {});
```

CONCATENATION

```
net.addLayer(' name ', dagnn.Concat(),{' input1' 'input2' 'input3' 'input4'}, {'output'});
```

DROPOUT

```
net.addLayer(' name ', dagnn.DropOut('rate', 0.4), {'input'}, {'output'}, {});
```

SOFTMAX

```
net.addLayer('prob', dagnn.SoftMax(), {' input '}, {'prob'}, {});
```

LOSS

```
net.addLayer('objective', dagnn.Loss('loss', 'log'), {'prob', 'label'}, {'objective'}, {});
```

ERROR

```
net.addLayer('error', dagnn.Loss('loss', 'classerror'), {'prob','label'}, 'error') ;
```

2. VGG-D network code implementation

```
function [net, info] = VGGD_train(imdb, expDir, varargin)
        vl_setupnn;
        % some common options
        opts.train.batchSize = 31;
        opts.train.numEpochs = 60 ;
        opts.train.continue = true ;
        opts.train.gpus = [1] ;
        opts.train.learningRate = [15e-3*ones(1, 30), 1e-3*ones(1,20)];
        opts.train.weightDecay = 5e-4;
        opts.train.momentum = 0.7;
        opts.train.expDir = expDir;
        opts.train.numSubBatches = 1;
        % getBatch options
        bopts.useGpu = numel(opts.train.gpus) >  0 ;
        opts.optMethod = 'gradient'; % ['adagrad', 'gradient']

    if(numel(varargin) > 0)

        netPre = varargin{1};

   end

net = dagnn.DagNN() ;

   net.addLayer('conv1_1', dagnn.Conv('size', [3 3 3 64], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]), {'input'},
{'conv1'},  {'conv1_1f'  'conv1_1b'});
        net.addLayer('relu1_1', dagnn.ReLU(), {'conv1'}, {'relu1'}, {});
        net.addLayer('conv1_2', dagnn.Conv('size', [3 3 64 64], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'relu1'}, {'conv2'},  {'conv1_2f'  'conv1_2b'});
        net.addLayer('relu1_2', dagnn.ReLU(), {'conv2'}, {'relu2'}, {});
        %net.addLayer('lrn1', dagnn.LRN('param', [5 1 0.0001/5 0.75]), {'relu1'}, {'lrn1'}, {});
        net.addLayer('pool1', dagnn.Pooling('method', 'max', 'poolSize', [2, 2], 'stride', [2 2], 'pad', [0 0 0 0]),
{'relu2'}, {'pool1'}, {});

   net.addLayer('conv2_1', dagnn.Conv('size', [3 3 64 128], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'pool1'}, {'conv3'},  {'conv2_1f'  'conv2_1b'});
        net.addLayer('relu2_1', dagnn.ReLU(), {'conv3'}, {'relu3'}, {});
        net.addLayer('conv2_2', dagnn.Conv('size', [3 3 128 128], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1
1]), {'relu3'}, {'conv4'},  {'conv2_2f'  'conv2_2b'});
        net.addLayer('relu2_2', dagnn.ReLU(), {'conv4'}, {'relu4'}, {});
        %net.addLayer('lrn2', dagnn.LRN('param', [5 1 0.0001/5 0.75]), {'relu2'}, {'lrn2'}, {});
        net.addLayer('pool2', dagnn.Pooling('method', 'max', 'poolSize', [2, 2], 'stride', [2 2], 'pad', [0 0 0 0]),
{'relu4'}, {'pool2'}, {});

   net.addLayer('conv3_1', dagnn.Conv('size', [3 3 128 256], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'pool2'}, {'conv5'},  {'conv3_1f'  'conv3_1b'});
        net.addLayer('relu3_1', dagnn.ReLU(), {'conv5'}, {'relu5'}, {});
   net.addLayer('conv3_2', dagnn.Conv('size', [3 3 256 256], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'relu5'}, {'conv6'},  {'conv3_2f'  'conv3_2b'});
        net.addLayer('relu3_2', dagnn.ReLU(), {'conv6'}, {'relu6'}, {});
        net.addLayer('conv3_3', dagnn.Conv('size', [3 3 256 256], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1
1]), {'relu6'}, {'conv7'},  {'conv3_3f'  'conv3_3b'});
        net.addLayer('relu3_3', dagnn.ReLU(), {'conv7'}, {'relu7'}, {});
        %net.addLayer('lrn3', dagnn.LRN('param', [5 1 0.0001/5 0.75]), {'relu2'}, {'lrn2'}, {});
        net.addLayer('pool3', dagnn.Pooling('method', 'max', 'poolSize', [2, 2], 'stride', [2 2], 'pad', [0 0 0 0]),
{'relu7'}, {'pool3'}, {});
```

```
   net.addLayer('conv4_1', dagnn.Conv('size', [3 3 256 512], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'pool3'}, {'conv8'},  {'conv4_1f'  'conv4_1b'});
            net.addLayer('relu4_1', dagnn.ReLU(), {'conv8'}, {'relu8'}, {});
            net.addLayer('conv4_2', dagnn.Conv('size', [3 3 512 512], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1
1]), {'relu8'}, {'conv9'},  {'conv4_2f'  'conv4_2b'});
            net.addLayer('relu4_2', dagnn.ReLU(), {'conv9'}, {'relu9'}, {});
            net.addLayer('conv4_3', dagnn.Conv('size', [3 3 512 512], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1
1]), {'relu9'}, {'conv10'},  {'conv4_3f'  'conv4_3b'});
            net.addLayer('relu4_3', dagnn.ReLU(), {'conv10'}, {'relu10'}, {});
            %net.addLayer('lrn4', dagnn.LRN('param', [5 1 0.0001/5 0.75]), {'relu2'}, {'lrn2'}, {});
            net.addLayer('pool4', dagnn.Pooling('method', 'max', 'poolSize', [2, 2], 'stride', [2 2], 'pad', [0 0 0 0]),
{'relu10'}, {'pool4'}, {});

   net.addLayer('conv5_1', dagnn.Conv('size', [3 3 512 512], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'pool4'}, {'conv11'},  {'conv5_1f'  'conv5_1b'});
            net.addLayer('relu5_1', dagnn.ReLU(), {'conv11'}, {'relu11'}, {});
            net.addLayer('conv5_2', dagnn.Conv('size', [3 3 512 512], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1
1]), {'relu11'}, {'conv12'},  {'conv5_2f'  'conv5_2b'});
            net.addLayer('relu5_2', dagnn.ReLU(), {'conv12'}, {'relu12'}, {});
            net.addLayer('conv5_3', dagnn.Conv('size', [3 3 512 512], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1
1]), {'relu12'}, {'conv13'},  {'conv5_3f'  'conv5_3b'});
            net.addLayer('relu5_3', dagnn.ReLU(), {'conv13'}, {'relu13'}, {});
            net.addLayer('pool5', dagnn.Pooling('method', 'max', 'poolSize', [2, 2], 'stride', [2 2], 'pad', [0 0 0 0]),
{'relu13'}, {'pool5'}, {});

            net.addLayer('conv6', dagnn.Conv('size', [7 7 512 4096], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool5'}, {'conv14'},  {'conv6f'  'conv6b'});
            net.addLayer('relu6', dagnn.ReLU(), {'conv14'}, {'relu14'}, {});
            net.addLayer('drop6', dagnn.DropOut('rate', 0.5), {'relu14'}, {'drop6'}, {});

            net.addLayer('conv7', dagnn.Conv('size', [1 1 4096 4096], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'drop6'}, {'conv15'},  {'conv7f'  'conv7b'});
            net.addLayer('relu7', dagnn.ReLU(), {'conv15'}, {'relu15'}, {});
            net.addLayer('drop7', dagnn.DropOut('rate', 0.5), {'relu15'}, {'drop7'}, {});

            net.addLayer('classifier', dagnn.Conv('size', [1 1 4096 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'drop7'}, {'classifier'},  {'conv8f'  'conv8b'});
            net.addLayer('prob', dagnn.SoftMax(), {'classifier'}, {'prob'}, {});
            net.addLayer('objective', dagnn.Loss('loss', 'log'), {'prob', 'label'}, {'objective'}, {});
            net.addLayer('error', dagnn.Loss('loss', 'classerror'), {'prob','label'}, 'error') ;
            % -- end of the network
% initialization of the weights (CRITICAL!!!!)
   if(numel(varargin) > 0)
                  initNet_FineTuning(net, netPre);

            else
                   initNet_He(net);
            end

   %train
   info = cnn_train_dag(net, imdb, @(i,b) getBatchDisk(bopts,i,b), opts.train, 'val', find(imdb.images.set ==
2)) ;
end

function initNet_FineTuning(net, netPre)
            net.initParams();

            ind = 1;
            for l=1:length(net.layers)-4
                  % is a convolution layer?
```

```
                if(strcmp(class(net.layers(l).block), 'dagnn.Conv'))
                        f_ind = net.layers(l).paramIndexes(1);
                        b_ind = net.layers(l).paramIndexes(2);

        net.params(f_ind).value = netPre.params(ind).value
                        ind = ind + 1;
                        net.params(b_ind).value = netPre.params(ind).value
                        ind = ind + 1;
                         if l<= length(net.layers)-4
                        net.params(f_ind).learningRate = 0;
                        net.params(f_ind).weightDecay = 1;
                        net.params(b_ind).learningRate = 0;
                        net.params(b_ind).weightDecay = 1;

                           else
                        net.params(f_ind).learningRate = 1;
                        net.params(f_ind).weightDecay = 1;
                        net.params(b_ind).learningRate = 1;
                        net.params(b_ind).weightDecay = 1;
                            end
                end
        end

        for l=length(net.layers)-6:length(net.layers)
                % is a convolution layer?
                if(strcmp(class(net.layers(l).block), 'dagnn.Conv'))
                        f_ind = net.layers(l).paramIndexes(1)
                        b_ind = net.layers(l).paramIndexes(2)

                        [h,w,in,out] = size(net.params(f_ind).value)
                        he_gain = 1e-2*sqrt(2/(h*w*in)) % sqrt(1/fan_in)
                        net.params(f_ind).value = (1/5)*he_gain*randn(size(net.params(f_ind).value),
'single');
                        net.params(f_ind).learningRate = 1;
                        net.params(f_ind).weightDecay = 1;

                        net.params(b_ind).value = zeros(size(net.params(b_ind).value), 'single');
                        net.params(b_ind).learningRate = 0.5;
                        net.params(b_ind).weightDecay = 1;

                end
        end
end


% getBatch for IMDBs that are too big to be in RAM
function inputs = getBatchDisk(opts, imdb, batch)
        images = zeros(224, 224, 3, 'single');
for i=1:numel(batch)
        im = imread(imdb.images.filenames{batch(i)}) ;
   im_ = single(im) ; im_=imresize(im_, [224 224]);
   im_= im_ - imdb.images.data_mean;
        images(:,:,:, i) = im_;
end
clear im; clear im_;
        labels = imdb.images.labels(1,batch) ;
        if opts.useGpu > 0
                images = gpuArray(images) ;
        end
```

```
        inputs = {'input', images, 'label', labels} ;
end
```

3. VGG-face with inception, first experiment

```
% ICP1
 net.addLayer('icp1_reduction1', dagnn.Conv('size', [1 1 512 256], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool5'}, {'icp1_reduction1'},  {'conv1_icp1_red1f'  'conv1_icp1_red1b'});
        net.addLayer('relu_icp1_reduction1', dagnn.ReLU(), {'icp1_reduction1'}, {'icp1_reduction1x'}, {});
        net.addLayer('icp1_reduction2', dagnn.Conv('size', [1 1 512 42], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'pool5'}, {'icp1_reduction2'},  {'conv1_icp1_red2f'  'conv1_icp1_red2b'});
        net.addLayer('relu_icp1_reduction2', dagnn.ReLU(), {'icp1_reduction2'}, {'icp1_reduction2x'}, {});
   net.addLayer('icp1_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'pool5'}, {'icp1_pool'}, {});

   net.addLayer('icp1_out0', dagnn.Conv('size', [1 1 512 172], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'pool5'}, {'icp1_out0'},  {'conv2_icp1_out0f'  'conv2_icp1_out0b'});
        net.addLayer('relu_icp1_out0', dagnn.ReLU(), {'icp1_out0'}, {'icp1_out0x'}, {});

   net.addLayer('icp1_out1', dagnn.Conv('size', [3 3 256 340], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp1_reduction1x'}, {'icp1_out1'},  {'conv2_icp1_out1f'  'conv2_icp1_out1b'});
        net.addLayer('relu_icp1_out1', dagnn.ReLU(), {'icp1_out1'}, {'icp1_out1x'}, {});

   net.addLayer('icp1_out2', dagnn.Conv('size', [5 5 42 84], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp1_reduction2x'}, {'icp1_out2'},  {'conv2_icp1_out2f'  'conv2_icp1_out2b'});
        net.addLayer('relu_icp1_out2', dagnn.ReLU(), {'icp1_out2'}, {'icp1_out2x'}, {});

   net.addLayer('icp1_out3', dagnn.Conv('size', [1 1 512 86], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp1_pool'}, {'icp1_out3'},  {'conv2_icp1_out3f'  'conv2_icp1_out3b'});
        net.addLayer('relu_icp1_out3', dagnn.ReLU(), {'icp1_out3'}, {'icp1_out3x'}, {});
   net.addLayer('icp2_in', dagnn.Concat(),{'icp1_out0x' 'icp1_out1x' 'icp1_out2x' 'icp1_out3x'}, {'icp2_in'});


   % ICP2
   net.addLayer('icp2_reduction1', dagnn.Conv('size', [1 1 682 341], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp2_in'}, {'icp2_reduction1'},  {'conv1_icp2_red1f'  'conv1_icp2_red1b'});
        net.addLayer('relu_icp2_reduction1', dagnn.ReLU(), {'icp2_reduction1'}, {'icp2_reduction1x'}, {});
        net.addLayer('icp2_reduction2', dagnn.Conv('size', [1 1 682 86], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp2_in'}, {'icp2_reduction2'},  {'conv1_icp2_red2f'  'conv1_icp2_red2b'});
        net.addLayer('relu_icp2_reduction2', dagnn.ReLU(), {'icp2_reduction2'}, {'icp2_reduction2x'}, {});
   net.addLayer('icp2_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp2_in'}, {'icp2_pool'}, {});

   net.addLayer('icp2_out0', dagnn.Conv('size', [1 1 682 341], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_in'}, {'icp2_out0'},  {'conv2_icp2_out0f'  'conv2_icp2_out0b'});
        net.addLayer('relu_icp2_out0', dagnn.ReLU(), {'icp2_out0'}, {'icp2_out0x'}, {});

   net.addLayer('icp2_out1', dagnn.Conv('size', [3 3 341 455], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp2_reduction1x'}, {'icp2_out1'},  {'conv2_icp2_out1f'  'conv2_icp2_out1b'});
        net.addLayer('relu_icp2_out1', dagnn.ReLU(), {'icp2_out1'}, {'icp2_out1x'}, {});

   net.addLayer('icp2_out2', dagnn.Conv('size', [5 5 86 256], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp2_reduction2x'}, {'icp2_out2'},  {'conv2_icp2_out2f'  'conv2_icp2_out2b'});
        net.addLayer('relu_icp2_out2', dagnn.ReLU(), {'icp2_out2'}, {'icp2_out2x'}, {});

   net.addLayer('icp2_out3', dagnn.Conv('size', [1 1 682 170], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_pool'}, {'icp2_out3'},  {'conv2_icp2_out3f'  'conv2_icp2_out3b'});
        net.addLayer('relu_icp2_out3', dagnn.ReLU(), {'icp2_out3'}, {'icp2_out3x'}, {});
   net.addLayer('icp2_out', dagnn.Concat(),{'icp2_out0x' 'icp2_out1x' 'icp2_out2x' 'icp2_out3x'}, {'icp2_out'});
```

```
   net.addLayer('cls3_pool', dagnn.Pooling('method', 'avg', 'poolSize', [7, 7], 'stride', [1 1], 'pad', [0 0 0 0]),
{'icp2_out'}, {'cls3_pool'}, {});
   net.addLayer('drop_cls3', dagnn.DropOut('rate', 0.4), {'cls3_pool'}, {'drop_cls3'}, {});


net.addLayer('classifier', dagnn.Conv('size', [1 1 1222 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'drop_cls3'}, {'classifier'},  {'conv8f'  'conv8b'});
            net.addLayer('prob', dagnn.SoftMax(), {'classifier'}, {'prob'}, {});
            net.addLayer('objective', dagnn.Loss('loss', 'log'), {'prob', 'label'}, {'objective'}, {});
            net.addLayer('error', dagnn.Loss('loss', 'classerror'), {'prob','label'}, 'error') ;
% -- end of the network
```

4. VGG-face with inception, second experiment

ICP1

```
net.addLayer('icp1_reduction1', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool5'}, {'icp1_reduction1'},  {'conv1_icp1_red1f'  'conv1_icp1_red1b'});
            net.addLayer('relu_icp1_reduction1', dagnn.ReLU(), {'icp1_reduction1'}, {'icp1_reduction1x'}, {});
            net.addLayer('icp1_reduction2', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'pool5'}, {'icp1_reduction2'},  {'conv1_icp1_red2f'  'conv1_icp1_red2b'});
            net.addLayer('relu_icp1_reduction2', dagnn.ReLU(), {'icp1_reduction2'}, {'icp1_reduction2x'}, {});
   net.addLayer('icp1_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'pool5'}, {'icp1_pool'}, {});

   net.addLayer('icp1_out0', dagnn.Conv('size', [1 1 512 256], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'pool5'}, {'icp1_out0'},  {'conv2_icp1_out0f'  'conv2_icp1_out0b'});
            net.addLayer('relu_icp1_out0', dagnn.ReLU(), {'icp1_out0'}, {'icp1_out0x'}, {});

   net.addLayer('icp1_out1', dagnn.Conv('size', [3 3 128 384], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp1_reduction1x'}, {'icp1_out1'},  {'conv2_icp1_out1f'  'conv2_icp1_out1b'});
            net.addLayer('relu_icp1_out1', dagnn.ReLU(), {'icp1_out1'}, {'icp1_out1x'}, {});

   net.addLayer('icp1_out2', dagnn.Conv('size', [5 5 64 256], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp1_reduction2x'}, {'icp1_out2'},  {'conv2_icp1_out2f'  'conv2_icp1_out2b'});
            net.addLayer('relu_icp1_out2', dagnn.ReLU(), {'icp1_out2'}, {'icp1_out2x'}, {});

   net.addLayer('icp1_out3', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp1_pool'}, {'icp1_out3'},  {'conv2_icp1_out3f'  'conv2_icp1_out3b'});
            net.addLayer('relu_icp1_out3', dagnn.ReLU(), {'icp1_out3'}, {'icp1_out3x'}, {});
   net.addLayer('icp2_in', dagnn.Concat(),{'icp1_out0x' 'icp1_out1x' 'icp1_out2x' 'icp1_out3x'}, {'icp2_in'});

   % add icp2
   net.addLayer('icp2_reduction1', dagnn.Conv('size', [1 1 1024 512], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0
0 0]), {'icp2_in'}, {'icp2_reduction1'},  {'conv1_icp2_red1f'  'conv1_icp2_red1b'});
            net.addLayer('relu_icp2_reduction1', dagnn.ReLU(), {'icp2_reduction1'}, {'icp2_reduction1x'}, {});
            net.addLayer('icp2_reduction2', dagnn.Conv('size', [1 1 1024 128], 'hasBias', true, 'stride', [1, 1],
'pad', [0 0 0 0]), {'icp2_in'}, {'icp2_reduction2'},  {'conv1_icp2_red2f'  'conv1_icp2_red2b'});
            net.addLayer('relu_icp2_reduction2', dagnn.ReLU(), {'icp2_reduction2'}, {'icp2_reduction2x'}, {});
   net.addLayer('icp2_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp2_in'}, {'icp2_pool'}, {});

   net.addLayer('icp2_out0', dagnn.Conv('size', [1 1 1024 256], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_in'}, {'icp2_out0'},  {'conv2_icp2_out0f'  'conv2_icp2_out0b'});
            net.addLayer('relu_icp2_out0', dagnn.ReLU(), {'icp2_out0'}, {'icp2_out0x'}, {});

   net.addLayer('icp2_out1', dagnn.Conv('size', [3 3 512 768], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp2_reduction1x'}, {'icp2_out1'},  {'conv2_icp2_out1f'  'conv2_icp2_out1b'});
```

```
        net.addLayer('relu_icp2_out1', dagnn.ReLU(), {'icp2_out1'}, {'icp2_out1x'}, {});

  net.addLayer('icp2_out2', dagnn.Conv('size', [5 5 128 512], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp2_reduction2x'}, {'icp2_out2'},  {'conv2_icp2_out2f'  'conv2_icp2_out2b'});
        net.addLayer('relu_icp2_out2', dagnn.ReLU(), {'icp2_out2'}, {'icp2_out2x'}, {});

  net.addLayer('icp2_out3', dagnn.Conv('size', [1 1 1024 256], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_pool'}, {'icp2_out3'},  {'conv2_icp2_out3f'  'conv2_icp2_out3b'});
        net.addLayer('relu_icp2_out3', dagnn.ReLU(), {'icp2_out3'}, {'icp2_out3x'}, {});
  net.addLayer('icp2_out', dagnn.Concat(),{'icp2_out0x' 'icp2_out1x' 'icp2_out2x' 'icp2_out3x'}, {'icp2_out'});

  net.addLayer('cls3_pool', dagnn.Pooling('method', 'avg', 'poolSize', [7, 7], 'stride', [1 1], 'pad', [0 0 0 0]),
{'icp2_out'}, {'cls3_pool'}, {});
        net.addLayer('drop_cls3', dagnn.DropOut('rate', 0.4), {'cls3_pool'}, {'drop_cls3'}, {});
        net.addLayer('classifier', dagnn.Conv('size', [1 1 1792 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'drop_cls3'}, {'classifier'},  {'conv8f'  'conv8b'});
```

5. GoogLeNet code implementation

```
function [net, info] = googLeNet_train(imdb, expDir, varargin)
        vl_setupnn;

        % some common options
        opts.train.batchSize = 20;
        opts.train.numEpochs = 30 ;
        opts.train.continue = true ;
        opts.train.gpus = [1] ;
        opts.train.learningRate = [1e-2*ones(1, 20),  1e-3*ones(1, 10), 1e-3*ones(1,40)];
        opts.train.weightDecay = 5e-4;
        opts.train.momentum = 0.;
        opts.train.expDir = expDir;
        opts.train.numSubBatches = 1;
        % getBatch options
        bopts.useGpu = numel(opts.train.gpus) >  0 ;
        opts.optMethod = 'gradient'; % ['adagrad', 'gradient']

    if(numel(varargin) > 0)
                netPre = varargin{1};
  end
% network definition!
net = dagnn.DagNN() ;

  net.addLayer('conv1', dagnn.Conv('size', [7 7 3 64], 'hasBias', true, 'stride', [2, 2], 'pad', [3 3 3 3]), {'input'},
{'conv1'},  {'conv1f'  'conv1b'});
        net.addLayer('relu1', dagnn.ReLU(), {'conv1'}, {'conv1x'}, {});
        net.addLayer('pool1', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'conv1x'}, {'pool1'}, {});
  net.addLayer('norm1', dagnn.LRN('param',[5 1 0.0001/5 0.75]), {'pool1'}, {'norm1'}, {});
  net.addLayer('reduction2', dagnn.Conv('size', [1 1 64 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'norm1'}, {'reduction2'},  {'conv1_redf'  'conv1_redb'});
        net.addLayer('relu1_reduction2', dagnn.ReLU(), {'reduction2'}, {'reduction2x'}, {});

  net.addLayer('conv2', dagnn.Conv('size', [3 3 64 192], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'reduction2x'}, {'conv2'},  {'conv2f'  'conv2b'});
        net.addLayer('relu2', dagnn.ReLU(), {'conv2'}, {'conv2x'}, {});
  net.addLayer('norm2', dagnn.LRN('param', [5 1 0.0001/5 0.75]), {'conv2x'}, {'norm2'}, {});
  net.addLayer('pool2', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'norm2'}, {'pool2'}, {});
```

```
    net.addLayer('icp1_reduction1', dagnn.Conv('size', [1 1 192 96], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool2'}, {'icp1_reduction1'},  {'conv1_icp1_red1f'  'conv1_icp1_red1b'});
            net.addLayer('relu_icp1_reduction1', dagnn.ReLU(), {'icp1_reduction1'}, {'icp1_reduction1x'}, {});
            net.addLayer('icp1_reduction2', dagnn.Conv('size', [1 1 192 16], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'pool2'}, {'icp1_reduction2'},  {'conv1_icp1_red2f'  'conv1_icp1_red2b'});
            net.addLayer('relu_icp1_reduction2', dagnn.ReLU(), {'icp1_reduction2'}, {'icp1_reduction2x'}, {});
    net.addLayer('icp1_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'pool2'}, {'icp1_pool'}, {});

    net.addLayer('icp1_out0', dagnn.Conv('size', [1 1 192 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'pool2'}, {'icp1_out0'},  {'conv2_icp1_out0f'  'conv2_icp1_out0b'});
            net.addLayer('relu_icp1_out0', dagnn.ReLU(), {'icp1_out0'}, {'icp1_out0x'}, {});

    net.addLayer('icp1_out1', dagnn.Conv('size', [3 3 96 128], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp1_reduction1x'}, {'icp1_out1'},  {'conv2_icp1_out1f'  'conv2_icp1_out1b'});
            net.addLayer('relu_icp1_out1', dagnn.ReLU(), {'icp1_out1'}, {'icp1_out1x'}, {});

    net.addLayer('icp1_out2', dagnn.Conv('size', [5 5 16 32], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp1_reduction2x'}, {'icp1_out2'},  {'conv2_icp1_out2f'  'conv2_icp1_out2b'});
            net.addLayer('relu_icp1_out2', dagnn.ReLU(), {'icp1_out2'}, {'icp1_out2x'}, {});

    net.addLayer('icp1_out3', dagnn.Conv('size', [1 1 192 32], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp1_pool'}, {'icp1_out3'},  {'conv2_icp1_out3f'  'conv2_icp1_out3b'});
            net.addLayer('relu_icp1_out3', dagnn.ReLU(), {'icp1_out3'}, {'icp1_out3x'}, {});
    net.addLayer('icp2_in', dagnn.Concat(),{'icp1_out0x' 'icp1_out1x' 'icp1_out2x' 'icp1_out3x'}, {'icp2_in'});

    % add icp2
    net.addLayer('icp2_reduction1', dagnn.Conv('size', [1 1 256 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp2_in'}, {'icp2_reduction1'},  {'conv1_icp2_red1f'  'conv1_icp2_red1b'});
            net.addLayer('relu_icp2_reduction1', dagnn.ReLU(), {'icp2_reduction1'}, {'icp2_reduction1x'}, {});
            net.addLayer('icp2_reduction2', dagnn.Conv('size', [1 1 256 32], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp2_in'}, {'icp2_reduction2'},  {'conv1_icp2_red2f'  'conv1_icp2_red2b'});
            net.addLayer('relu_icp2_reduction2', dagnn.ReLU(), {'icp2_reduction2'}, {'icp2_reduction2x'}, {});
    net.addLayer('icp2_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp2_in'}, {'icp2_pool'}, {});

    net.addLayer('icp2_out0', dagnn.Conv('size', [1 1 256 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_in'}, {'icp2_out0'},  {'conv2_icp2_out0f'  'conv2_icp2_out0b'});
            net.addLayer('relu_icp2_out0', dagnn.ReLU(), {'icp2_out0'}, {'icp2_out0x'}, {});

    net.addLayer('icp2_out1', dagnn.Conv('size', [3 3 128 192], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp2_reduction1x'}, {'icp2_out1'},  {'conv2_icp2_out1f'  'conv2_icp2_out1b'});
            net.addLayer('relu_icp2_out1', dagnn.ReLU(), {'icp2_out1'}, {'icp2_out1x'}, {});

    net.addLayer('icp2_out2', dagnn.Conv('size', [5 5 32 96], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp2_reduction2x'}, {'icp2_out2'},  {'conv2_icp2_out2f'  'conv2_icp2_out2b'});
            net.addLayer('relu_icp2_out2', dagnn.ReLU(), {'icp2_out2'}, {'icp2_out2x'}, {});

    net.addLayer('icp2_out3', dagnn.Conv('size', [1 1 256 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_pool'}, {'icp2_out3'},  {'conv2_icp2_out3f'  'conv2_icp2_out3b'});
            net.addLayer('relu_icp2_out3', dagnn.ReLU(), {'icp2_out3'}, {'icp2_out3x'}, {});
    net.addLayer('icp2_out', dagnn.Concat(),{'icp2_out0x' 'icp2_out1x' 'icp2_out2x' 'icp2_out3x'}, {'icp2_out'});


    %ADD icp3
    net.addLayer('icp3_in', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'icp2_out'}, {'icp3_in'}, {});

    net.addLayer('icp3_reduction1', dagnn.Conv('size', [1 1 480 96], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp3_in'}, {'icp3_reduction1'},  {'conv1_icp3_red1f'  'conv1_icp3_red1b'});
```

```
        net.addLayer('relu_icp3_reduction1', dagnn.ReLU(), {'icp3_reduction1'}, {'icp3_reduction1x'}, {});
        net.addLayer('icp3_reduction2', dagnn.Conv('size', [1 1 480 16], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp3_in'}, {'icp3_reduction2'},  {'conv1_icp3_red2f'  'conv1_icp3_red2b'});
        net.addLayer('relu_icp3_reduction2', dagnn.ReLU(), {'icp3_reduction2'}, {'icp3_reduction2x'}, {});
    net.addLayer('icp3_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp3_in'}, {'icp3_pool'}, {});

    net.addLayer('icp3_out0', dagnn.Conv('size', [1 1 480 192], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp3_in'}, {'icp3_out0'},  {'conv2_icp3_out0f'  'conv2_icp3_out0b'});
        net.addLayer('relu_icp3_out0', dagnn.ReLU(), {'icp3_out0'}, {'icp3_out0x'}, {});

    net.addLayer('icp3_out1', dagnn.Conv('size', [3 3 96 208], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp3_reduction1x'}, {'icp3_out1'},  {'conv2_icp3_out1f'  'conv2_icp3_out1b'});
        net.addLayer('relu_icp3_out1', dagnn.ReLU(), {'icp3_out1'}, {'icp3_out1x'}, {});

    net.addLayer('icp3_out2', dagnn.Conv('size', [5 5 16 48], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp3_reduction2x'}, {'icp3_out2'},  {'conv2_icp3_out2f'  'conv2_icp3_out2b'});
        net.addLayer('relu_icp3_out2', dagnn.ReLU(), {'icp3_out2'}, {'icp3_out2x'}, {});

    net.addLayer('icp3_out3', dagnn.Conv('size', [1 1 480 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp3_pool'}, {'icp3_out3'},  {'conv2_icp3_out3f'  'conv2_icp3_out3b'});
        net.addLayer('relu_icp3_out3', dagnn.ReLU(), {'icp3_out3'}, {'icp3_out3x'}, {});
    net.addLayer('icp3_out', dagnn.Concat(),{'icp3_out0x' 'icp3_out1x' 'icp3_out2x' 'icp3_out3x'}, {'icp3_out'});

%
%     net.addLayer('cls1_pool', dagnn.Pooling('method', 'avg', 'poolSize', [5, 5], 'stride', [3 3], 'pad', [0 2 0 2]),
{'icp3_out'}, {'cls1_pool'}, {});
%     net.addLayer('cls1_reduction', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0
0 0]), {'cls1_pool'}, {'cls1_reduction'},  {'conv1_cls1f'  'conv1_cls1b'});
%         net.addLayer('relu_cls1_reduction', dagnn.ReLU(), {'cls1_reduction'}, {'cls1_reductionx'}, {});
%     net.addLayer('cls1_fc1', dagnn.Conv('size', [1 1 128 1024], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'cls1_reductionx'}, {'cls1_fc1'},  {'conv2_cls1f'  'conv2_cls1b'});
%         net.addLayer('relu_cls1_fc1', dagnn.ReLU(), {'cls1_fc1'}, {'cls1_fc1x'}, {});
%     net.addLayer('drop_cls1', dagnn.DropOut('rate', 0.7), {'cls1_fc1x'}, {'drop_cls1'}, {});
%     net.addLayer('cls1_fc2', dagnn.Conv('size', [1 1 1024 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'cls1_fc1x'}, {'cls1_fc2'},  {'conv3_cls1f'  'conv3_cls1b'});
%         net.addLayer('prob_cls1', dagnn.SoftMax(), {'cls1_fc2'}, {'prob_cls1'}, {});
%         net.addLayer('objective_cls1', dagnn.Loss('loss', 'log'), {'prob_cls1', 'label_cls1'}, {'objective_cls1'},
{});
%

    %add icp4
    net.addLayer('icp4_reduction1', dagnn.Conv('size', [1 1 512 112], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp3_out'}, {'icp4_reduction1'},  {'conv1_icp4_red1f'  'conv1_icp4_red1b'});
        net.addLayer('relu_icp4_reduction1', dagnn.ReLU(), {'icp4_reduction1'}, {'icp4_reduction1x'}, {});
        net.addLayer('icp4_reduction2', dagnn.Conv('size', [1 1 512 24], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp3_out'}, {'icp4_reduction2'},  {'conv1_icp4_red2f'  'conv1_icp4_red2b'});
        net.addLayer('relu_icp4_reduction2', dagnn.ReLU(), {'icp4_reduction2'}, {'icp4_reduction2x'}, {});
    net.addLayer('icp4_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp3_out'}, {'icp4_pool'}, {});

    net.addLayer('icp4_out0', dagnn.Conv('size', [1 1 512 160], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp3_out'}, {'icp4_out0'},  {'conv2_icp4_out0f'  'conv2_icp4_out0b'});
        net.addLayer('relu_icp4_out0', dagnn.ReLU(), {'icp4_out0'}, {'icp4_out0x'}, {});

    net.addLayer('icp4_out1', dagnn.Conv('size', [3 3 112 224], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp4_reduction1x'}, {'icp4_out1'},  {'conv2_icp4_out1f'  'conv2_icp4_out1b'});
        net.addLayer('relu_icp4_out1', dagnn.ReLU(), {'icp4_out1'}, {'icp4_out1x'}, {});
```

```
    net.addLayer('icp4_out2', dagnn.Conv('size', [5 5 24 64], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp4_reduction2x'}, {'icp4_out2'},  {'conv2_icp4_out2f'  'conv2_icp4_out2b'});
            net.addLayer('relu_icp4_out2', dagnn.ReLU(), {'icp4_out2'}, {'icp4_out2x'}, {});

    net.addLayer('icp4_out3', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp4_pool'}, {'icp4_out3'},  {'conv2_icp4_out3f'  'conv2_icp4_out3b'});
            net.addLayer('relu_icp4_out3', dagnn.ReLU(), {'icp4_out3'}, {'icp4_out3x'}, {});
    net.addLayer('icp4_out', dagnn.Concat(),{'icp4_out0x' 'icp4_out1x' 'icp4_out2x' 'icp4_out3x'}, {'icp4_out'});

    %add icp5
    net.addLayer('icp5_reduction1', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp4_out'}, {'icp5_reduction1'},  {'conv1_icp5_red1f'  'conv1_icp5_red1b'});
            net.addLayer('relu_icp5_reduction1', dagnn.ReLU(), {'icp5_reduction1'}, {'icp5_reduction1x'}, {});
            net.addLayer('icp5_reduction2', dagnn.Conv('size', [1 1 512 24], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp4_out'}, {'icp5_reduction2'},  {'conv1_icp5_red2f'  'conv1_icp5_red2b'});
            net.addLayer('relu_icp5_reduction2', dagnn.ReLU(), {'icp5_reduction2'}, {'icp5_reduction2x'}, {});
    net.addLayer('icp5_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp4_out'}, {'icp5_pool'}, {});

    net.addLayer('icp5_out0', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp4_out'}, {'icp5_out0'},  {'conv2_icp5_out0f'  'conv2_icp5_out0b'});
            net.addLayer('relu_icp5_out0', dagnn.ReLU(), {'icp5_out0'}, {'icp5_out0x'}, {});

    net.addLayer('icp5_out1', dagnn.Conv('size', [3 3 128 256], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp5_reduction1x'}, {'icp5_out1'},  {'conv2_icp5_out1f'  'conv2_icp5_out1b'});
            net.addLayer('relu_icp5_out1', dagnn.ReLU(), {'icp5_out1'}, {'icp5_out1x'}, {});

    net.addLayer('icp5_out2', dagnn.Conv('size', [5 5 24 64], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp5_reduction2x'}, {'icp5_out2'},  {'conv2_icp5_out2f'  'conv2_icp5_out2b'});
            net.addLayer('relu_icp5_out2', dagnn.ReLU(), {'icp5_out2'}, {'icp5_out2x'}, {});

    net.addLayer('icp5_out3', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp5_pool'}, {'icp5_out3'},  {'conv2_icp5_out3f'  'conv2_icp5_out3b'});
            net.addLayer('relu_icp5_out3', dagnn.ReLU(), {'icp5_out3'}, {'icp5_out3x'}, {});
    net.addLayer('icp5_out', dagnn.Concat(),{'icp5_out0x' 'icp5_out1x' 'icp5_out2x' 'icp5_out3x'}, {'icp5_out'});

    %add icp6
    net.addLayer('icp6_reduction1', dagnn.Conv('size', [1 1 512 144], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp5_out'}, {'icp6_reduction1'},  {'conv1_icp6_red1f'  'conv1_icp6_red1b'});
            net.addLayer('relu_icp6_reduction1', dagnn.ReLU(), {'icp6_reduction1'}, {'icp6_reduction1x'}, {});
            net.addLayer('icp6_reduction2', dagnn.Conv('size', [1 1 512 32], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp5_out'}, {'icp6_reduction2'},  {'conv1_icp6_red2f'  'conv1_icp6_red2b'});
            net.addLayer('relu_icp6_reduction2', dagnn.ReLU(), {'icp6_reduction2'}, {'icp6_reduction2x'}, {});
    net.addLayer('icp6_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp5_out'}, {'icp6_pool'}, {});

    net.addLayer('icp6_out0', dagnn.Conv('size', [1 1 512 112], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp5_out'}, {'icp6_out0'},  {'conv2_icp6_out0f'  'conv2_icp6_out0b'});
            net.addLayer('relu_icp6_out0', dagnn.ReLU(), {'icp6_out0'}, {'icp6_out0x'}, {});

    net.addLayer('icp6_out1', dagnn.Conv('size', [3 3 144 288], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp6_reduction1x'}, {'icp6_out1'},  {'conv2_icp6_out1f'  'conv2_icp6_out1b'});
            net.addLayer('relu_icp6_out1', dagnn.ReLU(), {'icp6_out1'}, {'icp6_out1x'}, {});

    net.addLayer('icp6_out2', dagnn.Conv('size', [5 5 32 64], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp6_reduction2x'}, {'icp6_out2'},  {'conv2_icp6_out2f'  'conv2_icp6_out2b'});
            net.addLayer('relu_icp6_out2', dagnn.ReLU(), {'icp6_out2'}, {'icp6_out2x'}, {});

    net.addLayer('icp6_out3', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp6_pool'}, {'icp6_out3'},  {'conv2_icp6_out3f'  'conv2_icp6_out3b'});
```

```
            net.addLayer('relu_icp6_out3', dagnn.ReLU(), {'icp6_out3'}, {'icp6_out3x'}, {});
    net.addLayer('icp6_out', dagnn.Concat(),{'icp6_out0x' 'icp6_out1x' 'icp6_out2x' 'icp6_out3x'}, {'icp6_out'});
%
%     net.addLayer('cls2_pool', dagnn.Pooling('method', 'avg', 'poolSize', [5, 5], 'stride', [3 3], 'pad', [0 2 0 2]),
{'icp6_out'}, {'cls2_pool'}, {});
%     net.addLayer('cls2_reduction', dagnn.Conv('size', [1 1 528 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0
0 0]), {'cls2_pool'}, {'cls2_reduction'},  {'conv1_cls2f'  'conv1_cls2b'});
%           net.addLayer('relu_cls2_reduction', dagnn.ReLU(), {'cls2_reduction'}, {'cls2_reductionx'}, {});
%     net.addLayer('cls2_fc1', dagnn.Conv('size', [1 1 128 1024], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'cls2_reductionx'}, {'cls2_fc1'}, {'conv2_cls2f'  'conv2_cls2b'});
%           net.addLayer('relu_cls2_fc1', dagnn.ReLU(), {'cls2_fc1'}, {'cls2_fc1x'}, {});
%     net.addLayer('drop_cls2', dagnn.DropOut('rate', 0.7), {'cls2_fc1x'}, {'drop_cls2'}, {});
%     net.addLayer('cls2_fc2', dagnn.Conv('size', [1 1 1024 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'drop_cls2'}, {'cls2_fc2'},  {'conv3_cls2f'  'conv3_cls2b'});
%
%     net.addLayer('prob_cls2', dagnn.SoftMax(), {'cls2_fc2'}, {'prob_cls2'}, {});
%           net.addLayer('objective_cls2', dagnn.Loss('loss', 'log'), {'prob_cls2', 'label_cls2'}, {'objective_cls2'},
{});

    %add icp7
    net.addLayer('icp7_reduction1', dagnn.Conv('size', [1 1 528 160], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp6_out'}, {'icp7_reduction1'},  {'conv1_icp7_red1f'  'conv1_icp7_red1b'});
            net.addLayer('relu_icp7_reduction1', dagnn.ReLU(), {'icp7_reduction1'}, {'icp7_reduction1x'}, {});
            net.addLayer('icp7_reduction2', dagnn.Conv('size', [1 1 528 32], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp6_out'}, {'icp7_reduction2'},  {'conv1_icp7_red2f'  'conv1_icp7_red2b'});
            net.addLayer('relu_icp7_reduction2', dagnn.ReLU(), {'icp7_reduction2'}, {'icp7_reduction2x'}, {});
    net.addLayer('icp7_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp6_out'}, {'icp7_pool'}, {});

    net.addLayer('icp7_out0', dagnn.Conv('size', [1 1 528 256], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp6_out'}, {'icp7_out0'},  {'conv2_icp7_out0f'  'conv2_icp7_out0b'});
            net.addLayer('relu_icp7_out0', dagnn.ReLU(), {'icp7_out0'}, {'icp7_out0x'}, {});

    net.addLayer('icp7_out1', dagnn.Conv('size', [3 3 160 320], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'icp7_reduction1x'}, {'icp7_out1'},  {'conv2_icp7_out1f'  'conv2_icp7_out1b'});
            net.addLayer('relu_icp7_out1', dagnn.ReLU(), {'icp7_out1'}, {'icp7_out1x'}, {});

    net.addLayer('icp7_out2', dagnn.Conv('size', [5 5 32 128], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'icp7_reduction2x'}, {'icp7_out2'},  {'conv2_icp7_out2f'  'conv2_icp7_out2b'});
            net.addLayer('relu_icp7_out2', dagnn.ReLU(), {'icp7_out2'}, {'icp7_out2x'}, {});

    net.addLayer('icp7_out3', dagnn.Conv('size', [1 1 528 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp7_pool'}, {'icp7_out3'},  {'conv2_icp7_out3f'  'conv2_icp7_out3b'});
            net.addLayer('relu_icp7_out3', dagnn.ReLU(), {'icp7_out3'}, {'icp7_out3x'}, {});
    net.addLayer('icp7_out', dagnn.Concat(),{'icp7_out0x' 'icp7_out1x' 'icp7_out2x' 'icp7_out3x'}, {'icp7_out'});

    %add icp8
    net.addLayer('icp8_in', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'icp7_out'}, {'icp8_in'}, {});

    net.addLayer('icp8_reduction1', dagnn.Conv('size', [1 1 832 160], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp8_in'}, {'icp8_reduction1'},  {'conv1_icp8_red1f'  'conv1_icp8_red1b'});
            net.addLayer('relu_icp8_reduction1', dagnn.ReLU(), {'icp8_reduction1'}, {'icp8_reduction1x'}, {});
            net.addLayer('icp8_reduction2', dagnn.Conv('size', [1 1 832 32], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp8_in'}, {'icp8_reduction2'},  {'conv1_icp8_red2f'  'conv1_icp8_red2b'});
            net.addLayer('relu_icp8_reduction2', dagnn.ReLU(), {'icp8_reduction2'}, {'icp8_reduction2x'}, {});
    net.addLayer('icp8_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp8_in'}, {'icp8_pool'}, {});
```

```
    net.addLayer('icp8_out0', dagnn.Conv('size', [1 1 832 256], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'icp8_in'}, {'icp8_out0'},  {'conv2_icp8_out0f'  'conv2_icp8_out0b'});
            net.addLayer('relu_icp8_out0', dagnn.ReLU(), {'icp8_out0'}, {'icp8_out0x'}, {});

    net.addLayer('icp8_out1', dagnn.Conv('size', [3 3 160 320], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]), {'icp8_reduction1x'}, {'icp8_out1'},  {'conv2_icp8_out1f'  'conv2_icp8_out1b'});
            net.addLayer('relu_icp8_out1', dagnn.ReLU(), {'icp8_out1'}, {'icp8_out1x'}, {});

    net.addLayer('icp8_out2', dagnn.Conv('size', [5 5 32 128], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]), {'icp8_reduction2x'}, {'icp8_out2'},  {'conv2_icp8_out2f'  'conv2_icp8_out2b'});
            net.addLayer('relu_icp8_out2', dagnn.ReLU(), {'icp8_out2'}, {'icp8_out2x'}, {});

    net.addLayer('icp8_out3', dagnn.Conv('size', [1 1 832 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'icp8_pool'}, {'icp8_out3'},  {'conv2_icp8_out3f'  'conv2_icp8_out3b'});
            net.addLayer('relu_icp8_out3', dagnn.ReLU(), {'icp8_out3'}, {'icp8_out3x'}, {});
    net.addLayer('icp8_out', dagnn.Concat(),{'icp8_out0x' 'icp8_out1x' 'icp8_out2x' 'icp8_out3x'}, {'icp8_out'});

    %add icp9
    net.addLayer('icp9_reduction1', dagnn.Conv('size', [1 1 832 192], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'icp8_out'}, {'icp9_reduction1'},  {'conv1_icp9_red1f'  'conv1_icp9_red1b'});
            net.addLayer('relu_icp9_reduction1', dagnn.ReLU(), {'icp9_reduction1'}, {'icp9_reduction1x'}, {});
            net.addLayer('icp9_reduction2', dagnn.Conv('size', [1 1 832 48], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'icp8_out'}, {'icp9_reduction2'},  {'conv1_icp9_red2f'  'conv1_icp9_red2b'});
            net.addLayer('relu_icp9_reduction2', dagnn.ReLU(), {'icp9_reduction2'}, {'icp9_reduction2x'}, {});
    net.addLayer('icp9_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]), {'icp8_out'}, {'icp9_pool'}, {});

    net.addLayer('icp9_out0', dagnn.Conv('size', [1 1 832 384], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'icp8_out'}, {'icp9_out0'},  {'conv2_icp9_out0f'  'conv2_icp9_out0b'});
            net.addLayer('relu_icp9_out0', dagnn.ReLU(), {'icp9_out0'}, {'icp9_out0x'}, {});

    net.addLayer('icp9_out1', dagnn.Conv('size', [3 3 192 384], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]), {'icp9_reduction1x'}, {'icp9_out1'},  {'conv2_icp9_out1f'  'conv2_icp9_out1b'});
            net.addLayer('relu_icp9_out1', dagnn.ReLU(), {'icp9_out1'}, {'icp9_out1x'}, {});

    net.addLayer('icp9_out2', dagnn.Conv('size', [5 5 48 128], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]), {'icp9_reduction2x'}, {'icp9_out2'},  {'conv2_icp9_out2f'  'conv2_icp9_out2b'});
            net.addLayer('relu_icp9_out2', dagnn.ReLU(), {'icp9_out2'}, {'icp9_out2x'}, {});

    net.addLayer('icp9_out3', dagnn.Conv('size', [1 1 832 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'icp9_pool'}, {'icp9_out3'},  {'conv2_icp9_out3f'  'conv2_icp9_out3b'});
            net.addLayer('relu_icp9_out3', dagnn.ReLU(), {'icp9_out3'}, {'icp9_out3x'}, {});
    net.addLayer('icp9_out', dagnn.Concat(),{'icp9_out0x' 'icp9_out1x' 'icp9_out2x' 'icp9_out3x'}, {'icp9_out'});

    % classification
    net.addLayer('cls3_pool', dagnn.Pooling('method', 'avg', 'poolSize', [7, 7], 'stride', [1 1], 'pad', [0 0 0 0]), {'icp9_out'}, {'cls3_pool'}, {});
            net.addLayer('drop_cls3', dagnn.DropOut('rate', 0.4), {'cls3_pool'}, {'drop_cls3'}, {});
    net.addLayer('classifier', dagnn.Conv('size', [1 1 1024 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]), {'drop_cls3'}, {'classifier'},  {'conv_cls3f'  'convcls3b'});
            net.addLayer('prob', dagnn.SoftMax(), {'classifier'}, {'prob'}, {});
            net.addLayer('objective', dagnn.Loss('loss', 'log'), {'prob', 'label'}, {'objective'}, {});
            net.addLayer('error', dagnn.Loss('loss', 'classerror'), {'prob','label'}, 'error') ;
            % -- end of the network
% initialization of the weights (CRITICAL!!!!)
    if(numel(varargin) > 0)
                    initNet_FineTuning(net, netPre);

            else
                     initNet_He(net);
```

```
        end

    %train
    info = cnn_train_dag(net, imdb, @(i,b) getBatchDisk(bopts,i,b), opts.train, 'val', find(imdb.images.set ==
2)) ;
end

function initNet_FineTuning(net, netPre)
        net.initParams();

        ind = 1;
        for l=1:length(net.layers)-4
                % is a convolution layer?
                if(strcmp(class(net.layers(l).block), 'dagnn.Conv'))
                        f_ind = net.layers(l).paramIndexes(1);
                        b_ind = net.layers(l).paramIndexes(2);

      net.params(f_ind).value = netPre.params(ind).value;
                        ind = ind + 1;
                        net.params(b_ind).value = netPre.params(ind).value;
                        ind = ind + 1;
                         if l<= length(net.layers)-4
                        net.params(f_ind).learningRate = 1e-5;
                        net.params(f_ind).weightDecay = 1;
                        net.params(b_ind).learningRate = 1e-5;
                        net.params(b_ind).weightDecay = 1;

                          else
                         net.params(f_ind).learningRate = 1;
                        net.params(f_ind).weightDecay = 1;
                        net.params(b_ind).learningRate = 1;
                        net.params(b_ind).weightDecay = 1;
                         end
                end
        end

        for l=length(net.layers)-3:length(net.layers)
                % is a convolution layer?
                if(strcmp(class(net.layers(l).block), 'dagnn.Conv'))
                        f_ind = net.layers(l).paramIndexes(1)
                        b_ind = net.layers(l).paramIndexes(2)


                        [h,w,in,out] = size(net.params(f_ind).value)
                        he_gain = 1e-2*sqrt(2/(h*w*in)); % sqrt(1/fan_in)
                        net.params(f_ind).value = 2*he_gain*randn(size(net.params(f_ind).value), 'single');
                        net.params(f_ind).learningRate = 1;
                        net.params(f_ind).weightDecay = 1;

                        net.params(b_ind).value = zeros(size(net.params(b_ind).value), 'single');
                        net.params(b_ind).learningRate = 0.5;
                        net.params(b_ind).weightDecay = 1;

                end
        end
end


% getBatch for IMDBs that are too big to be in RAM
function inputs = getBatchDisk(opts, imdb, batch)
```

```
        images = zeros(224, 224, 3, 'single');
for i=1:numel(batch)
        im = imread(imdb.images.filenames{batch(i)}) ;
   im_ = single(im) ; im_=imresize(im_, [224 224]);
   im_= im_ - imdb.images.data_mean;
        images(:,:,:, i) = im_;
end
clear im; clear im_;
        labels = imdb.images.labels(1,batch) ;
        if opts.useGpu > 0
                images = gpuArray(images) ;
        end

        inputs = {'input', images, 'label', labels} ;
end
```

6. GoogLeNet 6 ICP BN

```
net = dagnn.DagNN() ;
        net.addLayer('bn0', dagnn.BatchNorm('numChannels', 3), {'input'}, {'bn0'}, {'bn0f', 'bn0b', 'bn0m'});

   net.addLayer('conv1', dagnn.Conv('size', [7 7 3 64], 'hasBias', true, 'stride', [2, 2], 'pad', [3 3 3 3]), {'bn0'},
{'conv1'},  {'conv1f'  'conv1b'});
        net.addLayer('relu1', dagnn.ReLU(), {'conv1'}, {'conv1x'}, {});
      net.addLayer('bn1', dagnn.BatchNorm('numChannels', 64), {'conv1x'}, {'bn1'}, {'bn1f', 'bn1b', 'bn1m'});

        net.addLayer('pool1', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'bn1'}, {'pool1'}, {});
   net.addLayer('norm1', dagnn.LRN('param',[5 1 0.0001/5 0.75]), {'pool1'}, {'norm1'}, {});
   net.addLayer('reduction2', dagnn.Conv('size', [1 1 64 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'norm1'}, {'reduction2'},  {'conv1_redf'  'conv1_redb'});
        net.addLayer('relu1_reduction2', dagnn.ReLU(), {'reduction2'}, {'reduction2x'}, {});
        net.addLayer('bn2', dagnn.BatchNorm('numChannels', 64), {'reduction2x'}, {'bn2'}, {'bn2f', 'bn2b',
'bn2m'});

   net.addLayer('conv2', dagnn.Conv('size', [3 3 64 192], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]), {'bn2'},
{'conv2'},  {'conv2f'  'conv2b'});
        net.addLayer('relu2', dagnn.ReLU(), {'conv2'}, {'conv2x'}, {});
        net.addLayer('bn3', dagnn.BatchNorm('numChannels', 192), {'conv2x'}, {'bn3'}, {'bn3f', 'bn3b',
'bn3m'});

   net.addLayer('norm2', dagnn.LRN('param', [5 1 0.0001/5 0.75]), {'bn3'}, {'norm2'}, {});
   net.addLayer('pool2', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'norm2'}, {'pool2'}, {});

   net.addLayer('icp1_reduction1', dagnn.Conv('size', [1 1 192 96], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool2'}, {'icp1_reduction1'},  {'conv1_icp1_red1f'  'conv1_icp1_red1b'});
        net.addLayer('relu_icp1_reduction1', dagnn.ReLU(), {'icp1_reduction1'}, {'icp1_reduction1x'}, {});
                net.addLayer('bn4', dagnn.BatchNorm('numChannels', 96), {'icp1_reduction1x'}, {'bn4'},
{'bn4f', 'bn4b', 'bn4m'});

   net.addLayer('icp1_reduction2', dagnn.Conv('size', [1 1 192 16], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool2'}, {'icp1_reduction2'},  {'conv1_icp1_red2f'  'conv1_icp1_red2b'});
        net.addLayer('relu_icp1_reduction2', dagnn.ReLU(), {'icp1_reduction2'}, {'icp1_reduction2x'}, {});
                net.addLayer('bn5', dagnn.BatchNorm('numChannels', 16), {'icp1_reduction2x'}, {'bn5'},
{'bn5f', 'bn5b', 'bn5m'});

   net.addLayer('icp1_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'pool2'}, {'icp1_pool'}, {});
```

```
    net.addLayer('icp1_out0', dagnn.Conv('size', [1 1 192 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'pool2'}, {'icp1_out0'},  {'conv2_icp1_out0f'  'conv2_icp1_out0b'});
            net.addLayer('relu_icp1_out0', dagnn.ReLU(), {'icp1_out0'}, {'icp1_out0x'}, {});
                    net.addLayer('bn6', dagnn.BatchNorm('numChannels', 64), {'icp1_out0x'}, {'bn6'}, {'bn6f',
'bn6b', 'bn6m'});


    net.addLayer('icp1_out1', dagnn.Conv('size', [3 3 96 128], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'bn4'}, {'icp1_out1'},  {'conv2_icp1_out1f'  'conv2_icp1_out1b'});
            net.addLayer('relu_icp1_out1', dagnn.ReLU(), {'icp1_out1'}, {'icp1_out1x'}, {});
               net.addLayer('bn7', dagnn.BatchNorm('numChannels', 128), {'icp1_out1x'}, {'bn7'}, {'bn7f', 'bn7b',
'bn7m'});

    net.addLayer('icp1_out2', dagnn.Conv('size', [5 5 16 32], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'bn5'}, {'icp1_out2'},  {'conv2_icp1_out2f'  'conv2_icp1_out2b'});
            net.addLayer('relu_icp1_out2', dagnn.ReLU(), {'icp1_out2'}, {'icp1_out2x'}, {});
               net.addLayer('bn8', dagnn.BatchNorm('numChannels', 32), {'icp1_out2x'}, {'bn8'}, {'bn8f', 'bn8b',
'bn8m'});

    net.addLayer('icp1_out3', dagnn.Conv('size', [1 1 192 32], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp1_pool'}, {'icp1_out3'},  {'conv2_icp1_out3f'  'conv2_icp1_out3b'});
            net.addLayer('relu_icp1_out3', dagnn.ReLU(), {'icp1_out3'}, {'icp1_out3x'}, {});
              net.addLayer('bn9', dagnn.BatchNorm('numChannels', 32), {'icp1_out3x'}, {'bn9'}, {'bn9f', 'bn9b',
'bn9m'});

    net.addLayer('icp2_in', dagnn.Concat(),{'bn6' 'bn7' 'bn8' 'bn9'}, {'icp2_in'});

    % add icp2
    net.addLayer('icp2_reduction1', dagnn.Conv('size', [1 1 256 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp2_in'}, {'icp2_reduction1'},  {'conv1_icp2_red1f'  'conv1_icp2_red1b'});
            net.addLayer('relu_icp2_reduction1', dagnn.ReLU(), {'icp2_reduction1'}, {'icp2_reduction1x'}, {});
                      net.addLayer('bn10', dagnn.BatchNorm('numChannels', 128), {'icp2_reduction1x'},
{'bn10'}, {'bn10f', 'bn10b', 'bn10m'});

    net.addLayer('icp2_reduction2', dagnn.Conv('size', [1 1 256 32], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp2_in'}, {'icp2_reduction2'},  {'conv1_icp2_red2f'  'conv1_icp2_red2b'});
            net.addLayer('relu_icp2_reduction2', dagnn.ReLU(), {'icp2_reduction2'}, {'icp2_reduction2x'}, {});
               net.addLayer('bn11', dagnn.BatchNorm('numChannels', 32), {'icp2_reduction2x'}, {'bn11'}, {'bn11f',
'bn11b', 'bn11m'});

    net.addLayer('icp2_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp2_in'}, {'icp2_pool'}, {});

    net.addLayer('icp2_out0', dagnn.Conv('size', [1 1 256 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_in'}, {'icp2_out0'},  {'conv2_icp2_out0f'  'conv2_icp2_out0b'});
            net.addLayer('relu_icp2_out0', dagnn.ReLU(), {'icp2_out0'}, {'icp2_out0x'}, {});
               net.addLayer('bn12', dagnn.BatchNorm('numChannels', 128), {'icp2_out0x'}, {'bn12'}, {'bn12f',
'bn12b', 'bn12m'});

    net.addLayer('icp2_out1', dagnn.Conv('size', [3 3 128 192], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'bn10'}, {'icp2_out1'},  {'conv2_icp2_out1f'  'conv2_icp2_out1b'});
            net.addLayer('relu_icp2_out1', dagnn.ReLU(), {'icp2_out1'}, {'icp2_out1x'}, {});
               net.addLayer('bn13', dagnn.BatchNorm('numChannels', 192), {'icp2_out1x'}, {'bn13'}, {'bn13f',
'bn13b', 'bn13m'});

    net.addLayer('icp2_out2', dagnn.Conv('size', [5 5 32 96], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'bn11'}, {'icp2_out2'},  {'conv2_icp2_out2f'  'conv2_icp2_out2b'});
            net.addLayer('relu_icp2_out2', dagnn.ReLU(), {'icp2_out2'}, {'icp2_out2x'}, {});
```

```
            net.addLayer('bn14', dagnn.BatchNorm('numChannels', 96), {'icp2_out2x'}, {'bn14'}, {'bn14f',
'bn14b', 'bn14m'});

   net.addLayer('icp2_out3', dagnn.Conv('size', [1 1 256 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp2_pool'}, {'icp2_out3'},  {'conv2_icp2_out3f'  'conv2_icp2_out3b'});
          net.addLayer('relu_icp2_out3', dagnn.ReLU(), {'icp2_out3'}, {'icp2_out3x'}, {});
            net.addLayer('bn15', dagnn.BatchNorm('numChannels', 64), {'icp2_out3x'}, {'bn15'}, {'bn15f',
'bn15b', 'bn15m'});

   net.addLayer('icp2_out', dagnn.Concat(),{'bn12' 'bn13' 'bn14' 'bn15'}, {'icp2_out'});


   %ADD icp3
   net.addLayer('icp3_in', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'icp2_out'}, {'icp3_in'}, {});

   net.addLayer('icp3_reduction1', dagnn.Conv('size', [1 1 480 96], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp3_in'}, {'icp3_reduction1'},  {'conv1_icp3_red1f'  'conv1_icp3_red1b'});
          net.addLayer('relu_icp3_reduction1', dagnn.ReLU(), {'icp3_reduction1'}, {'icp3_reduction1x'}, {});
                        net.addLayer('bn16', dagnn.BatchNorm('numChannels', 96), {'icp3_reduction1x'},
{'bn16'}, {'bn16f', 'bn16b', 'bn16m'});

   net.addLayer('icp3_reduction2', dagnn.Conv('size', [1 1 480 16], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp3_in'}, {'icp3_reduction2'},  {'conv1_icp3_red2f'  'conv1_icp3_red2b'});
          net.addLayer('relu_icp3_reduction2', dagnn.ReLU(), {'icp3_reduction2'}, {'icp3_reduction2x'}, {});
                    net.addLayer('bn17', dagnn.BatchNorm('numChannels', 16), {'icp3_reduction2x'}, {'bn17'},
{'bn17f', 'bn17b', 'bn17m'});

   net.addLayer('icp3_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp3_in'}, {'icp3_pool'}, {});

   net.addLayer('icp3_out0', dagnn.Conv('size', [1 1 480 192], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp3_in'}, {'icp3_out0'},  {'conv2_icp3_out0f'  'conv2_icp3_out0b'});
          net.addLayer('relu_icp3_out0', dagnn.ReLU(), {'icp3_out0'}, {'icp3_out0x'}, {});
                    net.addLayer('bn18', dagnn.BatchNorm('numChannels', 192), {'icp3_out0x'}, {'bn18'},
{'bn18f', 'bn18b', 'bn18m'});

   net.addLayer('icp3_out1', dagnn.Conv('size', [3 3 96 208], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'bn16'}, {'icp3_out1'},  {'conv2_icp3_out1f'  'conv2_icp3_out1b'});
          net.addLayer('relu_icp3_out1', dagnn.ReLU(), {'icp3_out1'}, {'icp3_out1x'}, {});
                    net.addLayer('bn19', dagnn.BatchNorm('numChannels', 208), {'icp3_out1x'}, {'bn19'},
{'bn19f', 'bn19b', 'bn19m'});

   net.addLayer('icp3_out2', dagnn.Conv('size', [5 5 16 48], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'bn17'}, {'icp3_out2'},  {'conv2_icp3_out2f'  'conv2_icp3_out2b'});
          net.addLayer('relu_icp3_out2', dagnn.ReLU(), {'icp3_out2'}, {'icp3_out2x'}, {});
                    net.addLayer('bn20', dagnn.BatchNorm('numChannels', 48), {'icp3_out2x'}, {'bn20'},
{'bn20f', 'bn20b', 'bn20m'});

   net.addLayer('icp3_out3', dagnn.Conv('size', [1 1 480 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp3_pool'}, {'icp3_out3'},  {'conv2_icp3_out3f'  'conv2_icp3_out3b'});
          net.addLayer('relu_icp3_out3', dagnn.ReLU(), {'icp3_out3'}, {'icp3_out3x'}, {});
                    net.addLayer('bn21', dagnn.BatchNorm('numChannels', 64), {'icp3_out3x'}, {'bn21'},
{'bn21f', 'bn21b', 'bn21m'});

   net.addLayer('icp3_out', dagnn.Concat(),{'bn18' 'bn19' 'bn20' 'bn21'}, {'icp3_out'});


    %add icp4
```

```
    net.addLayer('icp4_reduction1', dagnn.Conv('size', [1 1 512 112], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp3_out'}, {'icp4_reduction1'},  {'conv1_icp4_red1f'  'conv1_icp4_red1b'});
            net.addLayer('relu_icp4_reduction1', dagnn.ReLU(), {'icp4_reduction1'}, {'icp4_reduction1x'}, {});
                                    net.addLayer('bn23', dagnn.BatchNorm('numChannels', 112),
{'icp4_reduction1x'}, {'bn23'}, {'bn23f', 'bn23b', 'bn23m'});

            net.addLayer('icp4_reduction2', dagnn.Conv('size', [1 1 512 24], 'hasBias', true, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'icp3_out'}, {'icp4_reduction2'},  {'conv1_icp4_red2f'  'conv1_icp4_red2b'});
            net.addLayer('relu_icp4_reduction2', dagnn.ReLU(), {'icp4_reduction2'}, {'icp4_reduction2x'}, {});
                                    net.addLayer('bn24', dagnn.BatchNorm('numChannels', 24), {'icp4_reduction2x'},
{'bn24'}, {'bn24f', 'bn24b', 'bn24m'});

    net.addLayer('icp4_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp3_out'}, {'icp4_pool'}, {});

    net.addLayer('icp4_out0', dagnn.Conv('size', [1 1 512 160], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp3_out'}, {'icp4_out0'},  {'conv2_icp4_out0f'  'conv2_icp4_out0b'});
            net.addLayer('relu_icp4_out0', dagnn.ReLU(), {'icp4_out0'}, {'icp4_out0x'}, {});
                                    net.addLayer('bn25', dagnn.BatchNorm('numChannels', 160), {'icp4_out0x'},
{'bn25'}, {'bn25f', 'bn25b', 'bn25m'});

    net.addLayer('icp4_out1', dagnn.Conv('size', [3 3 112 224], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'bn23'}, {'icp4_out1'},  {'conv2_icp4_out1f'  'conv2_icp4_out1b'});
            net.addLayer('relu_icp4_out1', dagnn.ReLU(), {'icp4_out1'}, {'icp4_out1x'}, {});
                                    net.addLayer('bn26', dagnn.BatchNorm('numChannels', 224), {'icp4_out1x'},
{'bn26'}, {'bn26f', 'bn26b', 'bn26m'});

    net.addLayer('icp4_out2', dagnn.Conv('size', [5 5 24 64], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'bn24'}, {'icp4_out2'},  {'conv2_icp4_out2f'  'conv2_icp4_out2b'});
            net.addLayer('relu_icp4_out2', dagnn.ReLU(), {'icp4_out2'}, {'icp4_out2x'}, {});
                                    net.addLayer('bn27', dagnn.BatchNorm('numChannels', 64), {'icp4_out2x'},
{'bn27'}, {'bn27f', 'bn27b', 'bn27m'});

    net.addLayer('icp4_out3', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp4_pool'}, {'icp4_out3'},  {'conv2_icp4_out3f'  'conv2_icp4_out3b'});
            net.addLayer('relu_icp4_out3', dagnn.ReLU(), {'icp4_out3'}, {'icp4_out3x'}, {});
                                    net.addLayer('bn28', dagnn.BatchNorm('numChannels', 64), {'icp4_out3x'},
{'bn28'}, {'bn28f', 'bn28b', 'bn28m'});

    net.addLayer('icp4_out', dagnn.Concat(),{'bn25' 'bn26' 'bn27' 'bn28'}, {'icp4_out'});

    %add icp5
    net.addLayer('icp5_reduction1', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp4_out'}, {'icp5_reduction1'},  {'conv1_icp5_red1f'  'conv1_icp5_red1b'});
            net.addLayer('relu_icp5_reduction1', dagnn.ReLU(), {'icp5_reduction1'}, {'icp5_reduction1x'}, {});
                                    net.addLayer('bn29', dagnn.BatchNorm('numChannels', 128),
{'icp5_reduction1x'}, {'bn29'}, {'bn29f', 'bn29b', 'bn29m'});

    net.addLayer('icp5_reduction2', dagnn.Conv('size', [1 1 512 24], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp4_out'}, {'icp5_reduction2'},  {'conv1_icp5_red2f'  'conv1_icp5_red2b'});
            net.addLayer('relu_icp5_reduction2', dagnn.ReLU(), {'icp5_reduction2'}, {'icp5_reduction2x'}, {});
                                    net.addLayer('bn30', dagnn.BatchNorm('numChannels', 24), {'icp5_reduction2x'},
{'bn30'}, {'bn30f', 'bn30b', 'bn30m'});

    net.addLayer('icp5_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp4_out'}, {'icp5_pool'}, {});

    net.addLayer('icp5_out0', dagnn.Conv('size', [1 1 512 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp4_out'}, {'icp5_out0'},  {'conv2_icp5_out0f'  'conv2_icp5_out0b'});
            net.addLayer('relu_icp5_out0', dagnn.ReLU(), {'icp5_out0'}, {'icp5_out0x'}, {});
```

```
                net.addLayer('b31', dagnn.BatchNorm('numChannels', 128), {'icp5_out0x'},
{'bn31'}, {'bn31f', 'bn31b', 'bn31m'});

   net.addLayer('icp5_out1', dagnn.Conv('size', [3 3 128 256], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'bn29'}, {'icp5_out1'},  {'conv2_icp5_out1f'  'conv2_icp5_out1b'});
          net.addLayer('relu_icp5_out1', dagnn.ReLU(), {'icp5_out1'}, {'icp5_out1x'}, {});
                net.addLayer('bn32', dagnn.BatchNorm('numChannels', 256), {'icp5_out1x'},
{'bn32'}, {'bn32f', 'bn32b', 'bn32m'});

   net.addLayer('icp5_out2', dagnn.Conv('size', [5 5 24 64], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'bn30'}, {'icp5_out2'},  {'conv2_icp5_out2f'  'conv2_icp5_out2b'});
          net.addLayer('relu_icp5_out2', dagnn.ReLU(), {'icp5_out2'}, {'icp5_out2x'}, {});
                net.addLayer('bn33', dagnn.BatchNorm('numChannels', 64), {'icp5_out2x'},
{'bn33'}, {'bn33f', 'bn33b', 'bn33m'});

   net.addLayer('icp5_out3', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp5_pool'}, {'icp5_out3'},  {'conv2_icp5_out3f'  'conv2_icp5_out3b'});
          net.addLayer('relu_icp5_out3', dagnn.ReLU(), {'icp5_out3'}, {'icp5_out3x'}, {});
                net.addLayer('bn34', dagnn.BatchNorm('numChannels', 64), {'icp5_out3x'},
{'bn34'}, {'bn34f', 'bn34b', 'bn34m'});

   net.addLayer('icp5_out', dagnn.Concat(),{'bn31' 'bn32' 'bn33' 'bn34'}, {'icp5_out'});

%     %add icp6
   net.addLayer('icp6_reduction1', dagnn.Conv('size', [1 1 512 144], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp5_out'}, {'icp6_reduction1'},  {'conv1_icp6_red1f'  'conv1_icp6_red1b'});
          net.addLayer('relu_icp6_reduction1', dagnn.ReLU(), {'icp6_reduction1'}, {'icp6_reduction1x'}, {});
                net.addLayer('bn35', dagnn.BatchNorm('numChannels', 144),
{'icp6_reduction1x'}, {'bn35'}, {'bn35f', 'bn35b', 'bn35m'});

   net.addLayer('icp6_reduction2', dagnn.Conv('size', [1 1 512 32], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'icp5_out'}, {'icp6_reduction2'},  {'conv1_icp6_red2f'  'conv1_icp6_red2b'});
          net.addLayer('relu_icp6_reduction2', dagnn.ReLU(), {'icp6_reduction2'}, {'icp6_reduction2x'}, {});
                net.addLayer('bn36', dagnn.BatchNorm('numChannels', 32), {'icp6_reduction2x'},
{'bn36'}, {'bn36f', 'bn36b', 'bn36m'});

   net.addLayer('icp6_pool', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [1 1], 'pad', [1 1 1 1]),
{'icp5_out'}, {'icp6_pool'}, {});

   net.addLayer('icp6_out0', dagnn.Conv('size', [1 1 512 112], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp5_out'}, {'icp6_out0'},  {'conv2_icp6_out0f'  'conv2_icp6_out0b'});
          net.addLayer('relu_icp6_out0', dagnn.ReLU(), {'icp6_out0'}, {'icp6_out0x'}, {});
                net.addLayer('bn37', dagnn.BatchNorm('numChannels', 112), {'icp6_out0x'},
{'bn37'}, {'bn37f', 'bn37b', 'bn37m'});

   net.addLayer('icp6_out1', dagnn.Conv('size', [3 3 144 288], 'hasBias', true, 'stride', [1, 1], 'pad', [1 1 1 1]),
{'bn35'}, {'icp6_out1'},  {'conv2_icp6_out1f'  'conv2_icp6_out1b'});
          net.addLayer('relu_icp6_out1', dagnn.ReLU(), {'icp6_out1'}, {'icp6_out1x'}, {});
                net.addLayer('bn38', dagnn.BatchNorm('numChannels', 288), {'icp6_out1x'},
{'bn38'}, {'bn38f', 'bn38b', 'bn38m'});

   net.addLayer('icp6_out2', dagnn.Conv('size', [5 5 32 64], 'hasBias', true, 'stride', [1, 1], 'pad', [2 2 2 2]),
{'bn36'}, {'icp6_out2'},  {'conv2_icp6_out2f'  'conv2_icp6_out2b'});
          net.addLayer('relu_icp6_out2', dagnn.ReLU(), {'icp6_out2'}, {'icp6_out2x'}, {});
                net.addLayer('bn39', dagnn.BatchNorm('numChannels', 64), {'icp6_out2x'},
{'bn39'}, {'bn39f', 'bn39b', 'bn39m'});

   net.addLayer('icp6_out3', dagnn.Conv('size', [1 1 512 64], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'icp6_pool'}, {'icp6_out3'},  {'conv2_icp6_out3f'  'conv2_icp6_out3b'});
          net.addLayer('relu_icp6_out3', dagnn.ReLU(), {'icp6_out3'}, {'icp6_out3x'}, {});
```

```
                net.addLayer('bn40', dagnn.BatchNorm('numChannels', 64), {'icp6_out3x'},
{'bn40'}, {'bn40f', 'bn40b', 'bn40m'});

   net.addLayer('icp6_out', dagnn.Concat(),{'bn37' 'bn38' 'bn39' 'bn40'}, {'icp6_out'});

   net.addLayer('cls2_pool', dagnn.Pooling('method', 'avg', 'poolSize', [5, 5], 'stride', [3 3], 'pad', [0 2 0 2]),
{'icp6_out'}, {'cls2_pool'}, {});
   net.addLayer('cls2_reduction', dagnn.Conv('size', [1 1 528 128], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0
0]), {'cls2_pool'}, {'cls2_reduction'},  {'conv1_cls2f'  'conv1_cls2b'});
        net.addLayer('relu_cls2_reduction', dagnn.ReLU(), {'cls2_reduction'}, {'cls2_reductionx'}, {});
                net.addLayer('bn41', dagnn.BatchNorm('numChannels', 128), {'cls2_reductionx'},
{'bn41'}, {'bn41f', 'bn41b', 'bn41m'});

   net.addLayer('cls2_fc1', dagnn.Conv('size', [4 4 128 1024], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'bn41'}, {'cls2_fc1'}, {'conv2_cls2f'  'conv2_cls2b'});
        net.addLayer('relu_cls2_fc1', dagnn.ReLU(), {'cls2_fc1'}, {'cls2_fc1x'}, {});
   net.addLayer('drop_cls2', dagnn.DropOut('rate', 0.7), {'cls2_fc1x'}, {'drop_cls2'}, {});
   net.addLayer('cls2_fc2', dagnn.Conv('size', [1 1 1024 263], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'drop_cls2'}, {'cls2_fc2'},  {'conv3_cls2f'  'conv3_cls2b'});

   net.addLayer('prob', dagnn.SoftMax(), {'cls2_fc2'}, {'prob'}, {});
        net.addLayer('objective', dagnn.Loss('loss', 'log'), {'prob', 'label'}, {'objective'}, {});
   net.addLayer('error', dagnn.Loss('loss', 'classerror'), {'prob','label'}, 'error') ;
```

7. ResNet-101 code implementation

```
vl_setupnn;

        % some common options
        opts.train.batchSize = 20;
        opts.train.numEpochs = 30 ;
        opts.train.continue = true ;
        opts.train.gpus = [1] ;
        opts.train.learningRate = [5e-3*ones(1, 20),  1e-3*ones(1,10), 1e-3*ones(1,5)];
        opts.train.weightDecay = 1e-4;
        opts.train.momentum = 0.7;
        opts.train.expDir = expDir;
        opts.train.numSubBatches = 1;
        % getBatch options
        bopts.useGpu = numel(opts.train.gpus) >  0 ;
        opts.optMethod = 'gradient'; % ['adagrad', 'gradient']

    if(numel(varargin) > 0)
                netPre = varargin{1};
   end
% network definition!
net = dagnn.DagNN() ;

   net.addLayer('conv1', dagnn.Conv('size', [7 7 3 64], 'hasBias', false, 'stride', [2, 2], 'pad', [3 3 3 3]), {'input'},
{'conv1'},  {'conv1f'});
        net.addLayer('bn_conv1', dagnn.BatchNorm('numChannels', 64), {'conv1'}, {'bn1'}, {'bn1f', 'bn1b',
'bn1m'});
   net.addLayer('conv1_relu', dagnn.ReLU(), {'bn1'}, {'conv1x'}, {});

   net.addLayer('pool1', dagnn.Pooling('method', 'max', 'poolSize', [3, 3], 'stride', [2 2], 'pad', [0 1 0 1]),
{'conv1x'}, {'pool1'}, {});

   net.addLayer('res2a_branch1', dagnn.Conv('size', [1 1 64 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0
0]), {'pool1'}, {'res2a_branch1'},  {'res2a_branch1f'});
```

```
        net.addLayer('bn_res2a_branch1', dagnn.BatchNorm('numChannels', 256), {'res2a_branch1'}, {'res2a_branch1x'}, {'res2a_branch1xf', 'res2a_branch1xb', 'res2a_branch1xm'});

    net.addLayer('res2a_branch2a', dagnn.Conv('size', [1 1 64 64], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'pool1'}, {'res2a_branch2a'},  {'res2a_branch2af'});
        net.addLayer('bn_res2a_branch2a', dagnn.BatchNorm('numChannels', 64), {'res2a_branch2a'}, {'res2a_branch2ax'}, {'res2a_branch2axf', 'res2a_branch2axb', 'res2a_branch2axm'});
    net.addLayer('res2a_branch2a_relu', dagnn.ReLU(), {'res2a_branch2ax'}, {'res2a_branch2axxx'}, {});

    net.addLayer('res2a_branch2b', dagnn.Conv('size', [3 3 64 64], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res2a_branch2axxx'}, {'res2a_branch2b'},  {'res2a_branch2bf'});
        net.addLayer('bn_res2a_branch2b', dagnn.BatchNorm('numChannels', 64), {'res2a_branch2b'}, {'res2a_branch2bx'}, {'res2a_branch2bxf', 'res2a_branch2bxb', 'res2a_branch2bxm'});
    net.addLayer('res2a_branch2b_relu', dagnn.ReLU(), {'res2a_branch2bx'}, {'res2a_branch2bxxx'}, {});

    net.addLayer('res2a_branch2c', dagnn.Conv('size', [1 1 64 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res2a_branch2bxxx'}, {'res2a_branch2c'},  {'res2a_branch2cf'});
        net.addLayer('bn_res2a_branch2c', dagnn.BatchNorm('numChannels', 256), {'res2a_branch2c'}, {'res2a_branch2cx'}, {'res2a_branch2cxf', 'res2a_branch2cxb', 'res2a_branch2cxm'});

        net.addLayer('res2a', dagnn.Sum(), {'res2a_branch1x', 'res2a_branch2cx'}, {'res2a'});
    net.addLayer('res2a_relu', dagnn.ReLU(), {'res2a'}, {'res2ax'}, {});


    net.addLayer('res2b_branch2a', dagnn.Conv('size', [1 1 256 64], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res2ax'}, {'res2b_branch2a'},  {'res2b_branch2af'});
        net.addLayer('bn_res2b_branch2a', dagnn.BatchNorm('numChannels', 64), {'res2b_branch2a'}, {'res2b_branch2ax'}, {'res2b_branch2axf', 'res2b_branch2axb', 'res2b_branch2axm'});
    net.addLayer('res2b_branch2a_relu', dagnn.ReLU(), {'res2b_branch2ax'}, {'res2b_branch2axxx'}, {});

    net.addLayer('res2b_branch2b', dagnn.Conv('size', [3 3 64 64], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res2b_branch2axxx'}, {'res2b_branch2b'},  {'res2b_branch2bf'});
        net.addLayer('bn_res2b_branch2b', dagnn.BatchNorm('numChannels', 64), {'res2b_branch2b'}, {'res2b_branch2bx'}, {'res2b_branch2bxf', 'res2b_branch2bxb', 'res2b_branch2bxm'});
    net.addLayer('res2b_branch2b_relu', dagnn.ReLU(), {'res2b_branch2bx'}, {'res2b_branch2bxxx'}, {});

    net.addLayer('res2b_branch2c', dagnn.Conv('size', [1 1 64 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res2b_branch2bxxx'}, {'res2b_branch2c'},  {'res2b_branch2cf'});
        net.addLayer('bn_res2b_branch2c', dagnn.BatchNorm('numChannels', 256), {'res2b_branch2c'}, {'res2b_branch2cx'}, {'res2b_branch2cxf', 'res2b_branch2cxb', 'res2b_branch2cxm'});

        net.addLayer('res2b', dagnn.Sum(), {'res2ax', 'res2b_branch2cx'}, {'res2b'});
    net.addLayer('res2b_relu', dagnn.ReLU(), {'res2b'}, {'res2bx'}, {});


    net.addLayer('res2c_branch2a', dagnn.Conv('size', [1 1 256 64], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res2bx'}, {'res2c_branch2a'},  {'res2c_branch2af'});
        net.addLayer('bn_res2c_branch2a', dagnn.BatchNorm('numChannels', 64), {'res2c_branch2a'}, {'res2c_branch2ax'}, {'res2c_branch2axf', 'res2c_branch2axb', 'res2c_branch2axm'});
    net.addLayer('res2c_branch2a_relu', dagnn.ReLU(), {'res2c_branch2ax'}, {'res2c_branch2axxx'}, {});

    net.addLayer('res2c_branch2b', dagnn.Conv('size', [3 3 64 64], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res2c_branch2axxx'}, {'res2c_branch2b'},  {'res2c_branch2bf'});
        net.addLayer('bn_res2c_branch2b', dagnn.BatchNorm('numChannels', 64), {'res2c_branch2b'}, {'res2c_branch2bx'}, {'res2c_branch2bxf', 'res2c_branch2bxb', 'res2c_branch2bxm'});
    net.addLayer('res2c_branch2b_relu', dagnn.ReLU(), {'res2c_branch2bx'}, {'res2c_branch2bxxx'}, {});

    net.addLayer('res2c_branch2c', dagnn.Conv('size', [1 1 64 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res2c_branch2bxxx'}, {'res2c_branch2c'},  {'res2c_branch2cf'});
```

```
        net.addLayer('bn_res2c_branch2c', dagnn.BatchNorm('numChannels', 256), {'res2c_branch2c'}, {'res2c_branch2cx'}, {'res2c_branch2cxf', 'res2c_branch2cxb', 'res2c_branch2cxm'});

        net.addLayer('res2c', dagnn.Sum(), {'res2bx', 'res2c_branch2cx'}, {'res2c'});
    net.addLayer('res2c_relu', dagnn.ReLU(), {'res2c'}, {'res2cx'}, {});

    net.addLayer('res3a_branch1', dagnn.Conv('size', [1 1 256 512], 'hasBias', false, 'stride', [2, 2], 'pad', [0 0 0 0]), {'res2cx'}, {'res3a_branch1'},  {'res3a_branch1f'});
        net.addLayer('bn_res3a_branch1', dagnn.BatchNorm('numChannels', 512), {'res3a_branch1'}, {'res3a_branch1x'}, {'res3a_branch1xf', 'res3a_branch1xb', 'res3a_branch1xm'});

    net.addLayer('res3a_branch2a', dagnn.Conv('size', [1 1 256 128], 'hasBias', false, 'stride', [2, 2], 'pad', [0 0 0 0]), {'res2cx'}, {'res3a_branch2a'},  {'res3a_branch2af'});
        net.addLayer('bn_res3a_branch2a', dagnn.BatchNorm('numChannels', 128), {'res3a_branch2a'}, {'res3a_branch2ax'}, {'res3a_branch2axf', 'res3a_branch2axb', 'res3a_branch2axm'});
    net.addLayer('res3a_branch2a_relu', dagnn.ReLU(), {'res3a_branch2ax'}, {'res3a_branch2axxx'}, {});

    net.addLayer('res3a_branch2b', dagnn.Conv('size', [3 3 128 128], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res3a_branch2axxx'}, {'res3a_branch2b'},  {'res3a_branch2bf'});
        net.addLayer('bn_res3a_branch2b', dagnn.BatchNorm('numChannels', 128), {'res3a_branch2b'}, {'res3a_branch2bx'}, {'res3a_branch2bxf', 'res3a_branch2bxb', 'res3a_branch2bxm'});
    net.addLayer('res3a_branch2b_relu', dagnn.ReLU(), {'res3a_branch2bx'}, {'res3a_branch2bxxx'}, {});

    net.addLayer('res3a_branch2c', dagnn.Conv('size', [1 1 128 512], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res3a_branch2bxxx'}, {'res3a_branch2c'},  {'res3a_branch2cf'});
        net.addLayer('bn_res3a_branch2c', dagnn.BatchNorm('numChannels', 512), {'res3a_branch2c'}, {'res3a_branch2cx'}, {'res3a_branch2cxf', 'res3a_branch2cxb', 'res3a_branch2cxm'});

        net.addLayer('res3a', dagnn.Sum(), {'res3a_branch1x', 'res3a_branch2cx'}, {'res3a'});
    net.addLayer('res3a_relu', dagnn.ReLU(), {'res3a'}, {'res3ax'}, {});


    net.addLayer('res3b1_branch2a', dagnn.Conv('size', [1 1 512 128], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res3ax'}, {'res3b1_branch2a'},  {'res3b1_branch2af'});
        net.addLayer('bn_res3b1_branch2a', dagnn.BatchNorm('numChannels', 128), {'res3b1_branch2a'}, {'res3b1_branch2ax'}, {'res3b1_branch2axf', 'res3b1_branch2axb', 'res3b1_branch2axm'});
    net.addLayer('res3b1_branch2a_relu', dagnn.ReLU(), {'res3b1_branch2ax'}, {'res3b1_branch2axxx'}, {});

    net.addLayer('res3b1_branch2b', dagnn.Conv('size', [3 3 128 128], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res3b1_branch2axxx'}, {'res3b1_branch2b'},  {'res3b1_branch2bf'});
        net.addLayer('bn_res3b1_branch2b', dagnn.BatchNorm('numChannels', 128), {'res3b1_branch2b'}, {'res3b1_branch2bx'}, {'res3b1_branch2bxf', 'res3b1_branch2bxb', 'res3b1_branch2bxm'});
    net.addLayer('res3b1_branch2b_relu', dagnn.ReLU(), {'res3b1_branch2bx'}, {'res3b1_branch2bxxx'}, {});

    net.addLayer('res3b1_branch2c', dagnn.Conv('size', [1 1 128 512], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res3b1_branch2bxxx'}, {'res3b1_branch2c'},  {'res3b1_branch2cf'});
        net.addLayer('bn_res3b1_branch2c', dagnn.BatchNorm('numChannels', 512), {'res3b1_branch2c'}, {'res3b1_branch2cx'}, {'res3b1_branch2cxf', 'res3b1_branch2cxb', 'res3b1_branch2cxm'});

        net.addLayer('res3b1', dagnn.Sum(), {'res3ax', 'res3b1_branch2cx'}, {'res3b1'});
    net.addLayer('res3b1_relu', dagnn.ReLU(), {'res3b1'}, {'res3b1x'}, {});


    net.addLayer('res3b2_branch2a', dagnn.Conv('size', [1 1 512 128], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res3b1x'}, {'res3b2_branch2a'},  {'res3b2_branch2af'});
        net.addLayer('bn_res3b2_branch2a', dagnn.BatchNorm('numChannels', 128), {'res3b2_branch2a'}, {'res3b2_branch2ax'}, {'res3b2_branch2axf', 'res3b2_branch2axb', 'res3b2_branch2axm'});
    net.addLayer('res3b2_branch2a_relu', dagnn.ReLU(), {'res3b2_branch2ax'}, {'res3b2_branch2axxx'}, {});
```

```
    net.addLayer('res3b2_branch2b', dagnn.Conv('size', [3 3 128 128], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res3b2_branch2axxx'}, {'res3b2_branch2b'},  {'res3b2_branch2bf'});
            net.addLayer('bn_res3b2_branch2b', dagnn.BatchNorm('numChannels', 128), {'res3b2_branch2b'},
{'res3b2_branch2bx'}, {'res3b2_branch2bxf', 'res3b2_branch2bxb', 'res3b2_branch2bxm'});
    net.addLayer('res3b2_branch2b_relu', dagnn.ReLU(), {'res3b2_branch2bx'}, {'res3b2_branch2bxxx'}, {});

    net.addLayer('res3b2_branch2c', dagnn.Conv('size', [1 1 128 512], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res3b2_branch2bxxx'}, {'res3b2_branch2c'},  {'res3b2_branch2cf'});
            net.addLayer('bn_res3b2_branch2c', dagnn.BatchNorm('numChannels', 512), {'res3b2_branch2c'},
{'res3b2_branch2cx'}, {'res3b2_branch2cxf', 'res3b2_branch2cxb', 'res3b2_branch2cxm'});

            net.addLayer('res3b2', dagnn.Sum(), {'res3b1x', 'res3b2_branch2cx'}, {'res3b2'});
        net.addLayer('res3b2_relu', dagnn.ReLU(), {'res3b2'}, {'res3b2x'}, {});

         net.addLayer('res3b3_branch2a', dagnn.Conv('size', [1 1 512 128], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res3b2x'}, {'res3b3_branch2a'},  {'res3b3_branch2af'});
            net.addLayer('bn_res3b3_branch2a', dagnn.BatchNorm('numChannels', 128), {'res3b3_branch2a'},
{'res3b3_branch2ax'}, {'res3b3_branch2axf', 'res3b3_branch2axb', 'res3b3_branch2axm'});
    net.addLayer('res3b3_branch2a_relu', dagnn.ReLU(), {'res3b3_branch2ax'}, {'res3b3_branch2axxx'}, {});

    net.addLayer('res3b3_branch2b', dagnn.Conv('size', [3 3 128 128], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res3b3_branch2axxx'}, {'res3b3_branch2b'},  {'res3b3_branch2bf'});
            net.addLayer('bn_res3b3_branch2b', dagnn.BatchNorm('numChannels', 128), {'res3b3_branch2b'},
{'res3b3_branch2bx'}, {'res3b3_branch2bxf', 'res3b3_branch2bxb', 'res3b3_branch2bxm'});
    net.addLayer('res3b3_branch2b_relu', dagnn.ReLU(), {'res3b3_branch2bx'}, {'res3b3_branch2bxxx'}, {});

    net.addLayer('res3b3_branch2c', dagnn.Conv('size', [1 1 128 512], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res3b3_branch2bxxx'}, {'res3b3_branch2c'},  {'res3b3_branch2cf'});
            net.addLayer('bn_res3b3_branch2c', dagnn.BatchNorm('numChannels', 512), {'res3b3_branch2c'},
{'res3b3_branch2cx'}, {'res3b3_branch2cxf', 'res3b3_branch2cxb', 'res3b3_branch2cxm'});

            net.addLayer('res3b3', dagnn.Sum(), {'res3b2x', 'res3b3_branch2cx'}, {'res3b3'});
        net.addLayer('res3b3_relu', dagnn.ReLU(), {'res3b3'}, {'res3b3x'}, {});

    net.addLayer('res4a_branch1', dagnn.Conv('size', [1 1 512 1024], 'hasBias', false, 'stride', [2, 2], 'pad', [0 0
0 0]), {'res3b3x'}, {'res4a_branch1'},  {'res4a_branch1f'});
            net.addLayer('bn_res4a_branch1', dagnn.BatchNorm('numChannels', 1024), {'res4a_branch1'},
{'res4a_branch1x'}, {'res4a_branch1xf', 'res4a_branch1xb', 'res4a_branch1xm'});

    net.addLayer('res4a_branch2a', dagnn.Conv('size', [1 1 512 256], 'hasBias', false, 'stride', [2, 2], 'pad', [0 0
0 0]), {'res3b3x'}, {'res4a_branch2a'},  {'res4a_branch2af'});
            net.addLayer('bn_res4a_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4a_branch2a'},
{'res4a_branch2ax'}, {'res4a_branch2axf', 'res4a_branch2axb', 'res4a_branch2axm'});
    net.addLayer('res4a_branch2a_relu', dagnn.ReLU(), {'res4a_branch2ax'}, {'res4a_branch2axxx'}, {});

    net.addLayer('res4a_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4a_branch2axxx'}, {'res4a_branch2b'},  {'res4a_branch2bf'});
            net.addLayer('bn_res4a_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4a_branch2b'},
{'res4a_branch2bx'}, {'res4a_branch2bxf', 'res4a_branch2bxb', 'res4a_branch2bxm'});
    net.addLayer('res4a_branch2b_relu', dagnn.ReLU(), {'res4a_branch2bx'}, {'res4a_branch2bxxx'}, {});

    net.addLayer('res4a_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res4a_branch2bxxx'}, {'res4a_branch2c'},  {'res4a_branch2cf'});
            net.addLayer('bn_res4a_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4a_branch2c'},
{'res4a_branch2cx'}, {'res4a_branch2cxf', 'res4a_branch2cxb', 'res4a_branch2cxm'});

            net.addLayer('res4a', dagnn.Sum(), {'res4a_branch1x', 'res4a_branch2cx'}, {'res4a'});
        net.addLayer('res4a_relu', dagnn.ReLU(), {'res4a'}, {'res4ax'}, {});
```

```
    net.addLayer('res4b1_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res4ax'}, {'res4b1_branch2a'},  {'res4b1_branch2af'});
        net.addLayer('bn_res4b1_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b1_branch2a'}, {'res4b1_branch2ax'}, {'res4b1_branch2axf', 'res4b1_branch2axb', 'res4b1_branch2axm'});
    net.addLayer('res4b1_branch2a_relu', dagnn.ReLU(), {'res4b1_branch2ax'}, {'res4b1_branch2axxx'}, {});

    net.addLayer('res4b1_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res4b1_branch2axxx'}, {'res4b1_branch2b'},  {'res4b1_branch2bf'});
        net.addLayer('bn_res4b1_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b1_branch2b'}, {'res4b1_branch2bx'}, {'res4b1_branch2bxf', 'res4b1_branch2bxb', 'res4b1_branch2bxm'});
    net.addLayer('res4b1_branch2b_relu', dagnn.ReLU(), {'res4b1_branch2bx'}, {'res4b1_branch2bxxx'}, {});

    net.addLayer('res4b1_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res4b1_branch2bxxx'}, {'res4b1_branch2c'},  {'res4b1_branch2cf'});
        net.addLayer('bn_res4b1_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b1_branch2c'}, {'res4b1_branch2cx'}, {'res4b1_branch2cxf', 'res4b1_branch2cxb', 'res4b1_branch2cxm'});

        net.addLayer('res4b1', dagnn.Sum(), {'res4ax', 'res4b1_branch2cx'}, {'res4b1'});
    net.addLayer('res4b1_relu', dagnn.ReLU(), {'res4b1'}, {'res4b1x'}, {});

    net.addLayer('res4b2_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res4b1x'}, {'res4b2_branch2a'},  {'res4b2_branch2af'});
        net.addLayer('bn_res4b2_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b2_branch2a'}, {'res4b2_branch2ax'}, {'res4b2_branch2axf', 'res4b2_branch2axb', 'res4b2_branch2axm'});
    net.addLayer('res4b2_branch2a_relu', dagnn.ReLU(), {'res4b2_branch2ax'}, {'res4b2_branch2axxx'}, {});

    net.addLayer('res4b2_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res4b2_branch2axxx'}, {'res4b2_branch2b'},  {'res4b2_branch2bf'});
        net.addLayer('bn_res4b2_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b2_branch2b'}, {'res4b2_branch2bx'}, {'res4b2_branch2bxf', 'res4b2_branch2bxb', 'res4b2_branch2bxm'});
    net.addLayer('res4b2_branch2b_relu', dagnn.ReLU(), {'res4b2_branch2bx'}, {'res4b2_branch2bxxx'}, {});

    net.addLayer('res4b2_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res4b2_branch2bxxx'}, {'res4b2_branch2c'},  {'res4b2_branch2cf'});
        net.addLayer('bn_res4b2_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b2_branch2c'}, {'res4b2_branch2cx'}, {'res4b2_branch2cxf', 'res4b2_branch2cxb', 'res4b2_branch2cxm'});

        net.addLayer('res4b2', dagnn.Sum(), {'res4b1x', 'res4b2_branch2cx'}, {'res4b2'});
    net.addLayer('res4b2_relu', dagnn.ReLU(), {'res4b2'}, {'res4b2x'}, {});

    net.addLayer('res4b3_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res4b2x'}, {'res4b3_branch2a'},  {'res4b3_branch2af'});
        net.addLayer('bn_res4b3_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b3_branch2a'}, {'res4b3_branch2ax'}, {'res4b3_branch2axf', 'res4b3_branch2axb', 'res4b3_branch2axm'});
    net.addLayer('res4b3_branch2a_relu', dagnn.ReLU(), {'res4b3_branch2ax'}, {'res4b3_branch2axxx'}, {});

    net.addLayer('res4b3_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1 1 1]), {'res4b3_branch2axxx'}, {'res4b3_branch2b'},  {'res4b3_branch2bf'});
        net.addLayer('bn_res4b3_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b3_branch2b'}, {'res4b3_branch2bx'}, {'res4b3_branch2bxf', 'res4b3_branch2bxb', 'res4b3_branch2bxm'});
    net.addLayer('res4b3_branch2b_relu', dagnn.ReLU(), {'res4b3_branch2bx'}, {'res4b3_branch2bxxx'}, {});

    net.addLayer('res4b3_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0 0 0]), {'res4b3_branch2bxxx'}, {'res4b3_branch2c'},  {'res4b3_branch2cf'});
        net.addLayer('bn_res4b3_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b3_branch2c'}, {'res4b3_branch2cx'}, {'res4b3_branch2cxf', 'res4b3_branch2cxb', 'res4b3_branch2cxm'});

        net.addLayer('res4b3', dagnn.Sum(), {'res4b2x', 'res4b3_branch2cx'}, {'res4b3'});
    net.addLayer('res4b3_relu', dagnn.ReLU(), {'res4b3'}, {'res4b3x'}, {});
```

```
    net.addLayer('res4b4_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b3x'}, {'res4b4_branch2a'},  {'res4b4_branch2af'});
            net.addLayer('bn_res4b4_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b4_branch2a'},
{'res4b4_branch2ax'}, {'res4b4_branch2axf', 'res4b4_branch2axb', 'res4b4_branch2axm'});
    net.addLayer('res4b4_branch2a_relu', dagnn.ReLU(), {'res4b4_branch2ax'}, {'res4b4_branch2axxx'}, {});

    net.addLayer('res4b4_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4b4_branch2axxx'}, {'res4b4_branch2b'},  {'res4b4_branch2bf'});
            net.addLayer('bn_res4b4_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b4_branch2b'},
{'res4b4_branch2bx'}, {'res4b4_branch2bxf', 'res4b4_branch2bxb', 'res4b4_branch2bxm'});
    net.addLayer('res4b4_branch2b_relu', dagnn.ReLU(), {'res4b4_branch2bx'}, {'res4b4_branch2bxxx'}, {});

    net.addLayer('res4b4_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b4_branch2bxxx'}, {'res4b4_branch2c'},  {'res4b4_branch2cf'});
            net.addLayer('bn_res4b4_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b4_branch2c'},
{'res4b4_branch2cx'}, {'res4b4_branch2cxf', 'res4b4_branch2cxb', 'res4b4_branch2cxm'});

            net.addLayer('res4b4', dagnn.Sum(), {'res4b3x', 'res4b4_branch2cx'}, {'res4b4'});
        net.addLayer('res4b4_relu', dagnn.ReLU(), {'res4b4'}, {'res4b4x'}, {});

    net.addLayer('res4b5_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b4x'}, {'res4b5_branch2a'},  {'res4b5_branch2af'});
            net.addLayer('bn_res4b5_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b5_branch2a'},
{'res4b5_branch2ax'}, {'res4b5_branch2axf', 'res4b5_branch2axb', 'res4b5_branch2axm'});
    net.addLayer('res4b5_branch2a_relu', dagnn.ReLU(), {'res4b5_branch2ax'}, {'res4b5_branch2axxx'}, {});

    net.addLayer('res4b5_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4b5_branch2axxx'}, {'res4b5_branch2b'},  {'res4b5_branch2bf'});
            net.addLayer('bn_res4b5_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b5_branch2b'},
{'res4b5_branch2bx'}, {'res4b5_branch2bxf', 'res4b5_branch2bxb', 'res4b5_branch2bxm'});
    net.addLayer('res4b5_branch2b_relu', dagnn.ReLU(), {'res4b5_branch2bx'}, {'res4b5_branch2bxxx'}, {});

    net.addLayer('res4b5_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b5_branch2bxxx'}, {'res4b5_branch2c'},  {'res4b5_branch2cf'});
            net.addLayer('bn_res4b5_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b5_branch2c'},
{'res4b5_branch2cx'}, {'res4b5_branch2cxf', 'res4b5_branch2cxb', 'res4b5_branch2cxm'});

            net.addLayer('res4b5', dagnn.Sum(), {'res4b4x', 'res4b5_branch2cx'}, {'res4b5'});
        net.addLayer('res4b5_relu', dagnn.ReLU(), {'res4b5'}, {'res4b5x'}, {});

    net.addLayer('res4b6_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b5x'}, {'res4b6_branch2a'},  {'res4b6_branch2af'});
            net.addLayer('bn_res4b6_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b6_branch2a'},
{'res4b6_branch2ax'}, {'res4b6_branch2axf', 'res4b6_branch2axb', 'res4b6_branch2axm'});
    net.addLayer('res4b6_branch2a_relu', dagnn.ReLU(), {'res4b6_branch2ax'}, {'res4b6_branch2axxx'}, {});

    net.addLayer('res4b6_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4b6_branch2axxx'}, {'res4b6_branch2b'},  {'res4b6_branch2bf'});
            net.addLayer('bn_res4b6_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b6_branch2b'},
{'res4b6_branch2bx'}, {'res4b6_branch2bxf', 'res4b6_branch2bxb', 'res4b6_branch2bxm'});
    net.addLayer('res4b6_branch2b_relu', dagnn.ReLU(), {'res4b6_branch2bx'}, {'res4b6_branch2bxxx'}, {});

    net.addLayer('res4b6_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b6_branch2bxxx'}, {'res4b6_branch2c'},  {'res4b6_branch2cf'});
            net.addLayer('bn_res4b6_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b6_branch2c'},
{'res4b6_branch2cx'}, {'res4b6_branch2cxf', 'res4b6_branch2cxb', 'res4b6_branch2cxm'});

            net.addLayer('res4b6', dagnn.Sum(), {'res4b5x', 'res4b6_branch2cx'}, {'res4b6'});
        net.addLayer('res4b6_relu', dagnn.ReLU(), {'res4b6'}, {'res4b6x'}, {});
```

```
    net.addLayer('res4b7_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b6x'}, {'res4b7_branch2a'},  {'res4b7_branch2af'});
            net.addLayer('bn_res4b7_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b7_branch2a'},
{'res4b7_branch2ax'}, {'res4b7_branch2axf', 'res4b7_branch2axb', 'res4b7_branch2axm'});
    net.addLayer('res4b7_branch2a_relu', dagnn.ReLU(), {'res4b7_branch2ax'}, {'res4b7_branch2axxx'}, {});

    net.addLayer('res4b7_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4b7_branch2axxx'}, {'res4b7_branch2b'},  {'res4b7_branch2bf'});
            net.addLayer('bn_res4b7_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b7_branch2b'},
{'res4b7_branch2bx'}, {'res4b7_branch2bxf', 'res4b7_branch2bxb', 'res4b7_branch2bxm'});
    net.addLayer('res4b7_branch2b_relu', dagnn.ReLU(), {'res4b7_branch2bx'}, {'res4b7_branch2bxxx'}, {});

    net.addLayer('res4b7_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b7_branch2bxxx'}, {'res4b7_branch2c'},  {'res4b7_branch2cf'});
            net.addLayer('bn_res4b7_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b7_branch2c'},
{'res4b7_branch2cx'}, {'res4b7_branch2cxf', 'res4b7_branch2cxb', 'res4b7_branch2cxm'});

            net.addLayer('res4b7', dagnn.Sum(), {'res4b6x', 'res4b7_branch2cx'}, {'res4b7'});
        net.addLayer('res4b7_relu', dagnn.ReLU(), {'res4b7'}, {'res4b7x'}, {});

    net.addLayer('res4b8_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b7x'}, {'res4b8_branch2a'},  {'res4b8_branch2af'});
            net.addLayer('bn_res4b8_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b8_branch2a'},
{'res4b8_branch2ax'}, {'res4b8_branch2axf', 'res4b8_branch2axb', 'res4b8_branch2axm'});
    net.addLayer('res4b8_branch2a_relu', dagnn.ReLU(), {'res4b8_branch2ax'}, {'res4b8_branch2axxx'}, {});

    net.addLayer('res4b8_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4b8_branch2axxx'}, {'res4b8_branch2b'},  {'res4b8_branch2bf'});
            net.addLayer('bn_res4b8_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b8_branch2b'},
{'res4b8_branch2bx'}, {'res4b8_branch2bxf', 'res4b8_branch2bxb', 'res4b8_branch2bxm'});
    net.addLayer('res4b8_branch2b_relu', dagnn.ReLU(), {'res4b8_branch2bx'}, {'res4b8_branch2bxxx'}, {});

    net.addLayer('res4b8_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b8_branch2bxxx'}, {'res4b8_branch2c'},  {'res4b8_branch2cf'});
            net.addLayer('bn_res4b8_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b8_branch2c'},
{'res4b8_branch2cx'}, {'res4b8_branch2cxf', 'res4b8_branch2cxb', 'res4b8_branch2cxm'});

            net.addLayer('res4b8', dagnn.Sum(), {'res4b7x', 'res4b8_branch2cx'}, {'res4b8'});
        net.addLayer('res4b8_relu', dagnn.ReLU(), {'res4b8'}, {'res4b8x'}, {});

    net.addLayer('res4b9_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b8x'}, {'res4b9_branch2a'},  {'res4b9_branch2af'});
            net.addLayer('bn_res4b9_branch2a', dagnn.BatchNorm('numChannels', 256), {'res4b9_branch2a'},
{'res4b9_branch2ax'}, {'res4b9_branch2axf', 'res4b9_branch2axb', 'res4b9_branch2axm'});
    net.addLayer('res4b9_branch2a_relu', dagnn.ReLU(), {'res4b9_branch2ax'}, {'res4b9_branch2axxx'}, {});

    net.addLayer('res4b9_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res4b9_branch2axxx'}, {'res4b9_branch2b'},  {'res4b9_branch2bf'});
            net.addLayer('bn_res4b9_branch2b', dagnn.BatchNorm('numChannels', 256), {'res4b9_branch2b'},
{'res4b9_branch2bx'}, {'res4b9_branch2bxf', 'res4b9_branch2bxb', 'res4b9_branch2bxm'});
    net.addLayer('res4b9_branch2b_relu', dagnn.ReLU(), {'res4b9_branch2bx'}, {'res4b9_branch2bxxx'}, {});

    net.addLayer('res4b9_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad', [0
0 0 0]), {'res4b9_branch2bxxx'}, {'res4b9_branch2c'},  {'res4b9_branch2cf'});
            net.addLayer('bn_res4b9_branch2c', dagnn.BatchNorm('numChannels', 1024), {'res4b9_branch2c'},
{'res4b9_branch2cx'}, {'res4b9_branch2cxf', 'res4b9_branch2cxb', 'res4b9_branch2cxm'});

            net.addLayer('res4b9', dagnn.Sum(), {'res4b8x', 'res4b9_branch2cx'}, {'res4b9'});
        net.addLayer('res4b9_relu', dagnn.ReLU(), {'res4b9'}, {'res4b9x'}, {});
```

```
    net.addLayer('res4b10_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b9x'}, {'res4b10_branch2a'},  {'res4b10_branch2af'});
            net.addLayer('bn_res4b10_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b10_branch2a'}, {'res4b10_branch2ax'}, {'res4b10_branch2axf', 'res4b10_branch2axb',
'res4b10_branch2axm'});
    net.addLayer('res4b10_branch2a_relu', dagnn.ReLU(), {'res4b10_branch2ax'}, {'res4b10_branch2axxx'},
{});

    net.addLayer('res4b10_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b10_branch2axxx'}, {'res4b10_branch2b'},  {'res4b10_branch2bf'});
            net.addLayer('bn_res4b10_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b10_branch2b'}, {'res4b10_branch2bx'}, {'res4b10_branch2bxf', 'res4b10_branch2bxb',
'res4b10_branch2bxm'});
    net.addLayer('res4b10_branch2b_relu', dagnn.ReLU(), {'res4b10_branch2bx'}, {'res4b10_branch2bxxx'},
{});

    net.addLayer('res4b10_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b10_branch2bxxx'}, {'res4b10_branch2c'},  {'res4b10_branch2cf'});
            net.addLayer('bn_res4b10_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b10_branch2c'}, {'res4b10_branch2cx'}, {'res4b10_branch2cxf', 'res4b10_branch2cxb',
'res4b10_branch2cxm'});

            net.addLayer('res4b10', dagnn.Sum(), {'res4b9x', 'res4b10_branch2cx'}, {'res4b10'});
        net.addLayer('res4b10_relu', dagnn.ReLU(), {'res4b10'}, {'res4b10x'}, {});

    net.addLayer('res4b11_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b10x'}, {'res4b11_branch2a'},  {'res4b11_branch2af'});
            net.addLayer('bn_res4b11_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b11_branch2a'}, {'res4b11_branch2ax'}, {'res4b11_branch2axf', 'res4b11_branch2axb',
'res4b11_branch2axm'});
    net.addLayer('res4b11_branch2a_relu', dagnn.ReLU(), {'res4b11_branch2ax'}, {'res4b11_branch2axxx'},
{});

    net.addLayer('res4b11_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b11_branch2axxx'}, {'res4b11_branch2b'},  {'res4b11_branch2bf'});
            net.addLayer('bn_res4b11_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b11_branch2b'}, {'res4b11_branch2bx'}, {'res4b11_branch2bxf', 'res4b11_branch2bxb',
'res4b11_branch2bxm'});
    net.addLayer('res4b11_branch2b_relu', dagnn.ReLU(), {'res4b11_branch2bx'}, {'res4b11_branch2bxxx'},
{});

    net.addLayer('res4b11_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b11_branch2bxxx'}, {'res4b11_branch2c'},  {'res4b11_branch2cf'});
            net.addLayer('bn_res4b11_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b11_branch2c'}, {'res4b11_branch2cx'}, {'res4b11_branch2cxf', 'res4b11_branch2cxb',
'res4b11_branch2cxm'});

            net.addLayer('res4b11', dagnn.Sum(), {'res4b10x', 'res4b11_branch2cx'}, {'res4b11'});
        net.addLayer('res4b11_relu', dagnn.ReLU(), {'res4b11'}, {'res4b11x'}, {});

    net.addLayer('res4b12_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b11x'}, {'res4b12_branch2a'},  {'res4b12_branch2af'});
            net.addLayer('bn_res4b12_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b12_branch2a'}, {'res4b12_branch2ax'}, {'res4b12_branch2axf', 'res4b12_branch2axb',
'res4b12_branch2axm'});
    net.addLayer('res4b12_branch2a_relu', dagnn.ReLU(), {'res4b12_branch2ax'}, {'res4b12_branch2axxx'},
{});

    net.addLayer('res4b12_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b12_branch2axxx'}, {'res4b12_branch2b'},  {'res4b12_branch2bf'});
```

```
        net.addLayer('bn_res4b12_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b12_branch2b'}, {'res4b12_branch2bx'}, {'res4b12_branch2bxf', 'res4b12_branch2bxb',
'res4b12_branch2bxm'});
    net.addLayer('res4b12_branch2b_relu', dagnn.ReLU(), {'res4b12_branch2bx'}, {'res4b12_branch2bxxx'},
{});

    net.addLayer('res4b12_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b12_branch2bxxx'}, {'res4b12_branch2c'},  {'res4b12_branch2cf'});
        net.addLayer('bn_res4b12_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b12_branch2c'}, {'res4b12_branch2cx'}, {'res4b12_branch2cxf', 'res4b12_branch2cxb',
'res4b12_branch2cxm'});

        net.addLayer('res4b12', dagnn.Sum(), {'res4b11x', 'res4b12_branch2cx'}, {'res4b12'});
    net.addLayer('res4b12_relu', dagnn.ReLU(), {'res4b12'}, {'res4b12x'}, {});

    net.addLayer('res4b13_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b12x'}, {'res4b13_branch2a'},  {'res4b13_branch2af'});
        net.addLayer('bn_res4b13_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b13_branch2a'}, {'res4b13_branch2ax'}, {'res4b13_branch2axf', 'res4b13_branch2axb',
'res4b13_branch2axm'});
    net.addLayer('res4b13_branch2a_relu', dagnn.ReLU(), {'res4b13_branch2ax'}, {'res4b13_branch2axxx'},
{});

    net.addLayer('res4b13_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b13_branch2axxx'}, {'res4b13_branch2b'},  {'res4b13_branch2bf'});
        net.addLayer('bn_res4b13_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b13_branch2b'}, {'res4b13_branch2bx'}, {'res4b13_branch2bxf', 'res4b13_branch2bxb',
'res4b13_branch2bxm'});
    net.addLayer('res4b13_branch2b_relu', dagnn.ReLU(), {'res4b13_branch2bx'}, {'res4b13_branch2bxxx'},
{});

    net.addLayer('res4b13_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b13_branch2bxxx'}, {'res4b13_branch2c'},  {'res4b13_branch2cf'});
        net.addLayer('bn_res4b13_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b13_branch2c'}, {'res4b13_branch2cx'}, {'res4b13_branch2cxf', 'res4b13_branch2cxb',
'res4b13_branch2cxm'});

        net.addLayer('res4b13', dagnn.Sum(), {'res4b12x', 'res4b13_branch2cx'}, {'res4b13'});
    net.addLayer('res4b13_relu', dagnn.ReLU(), {'res4b13'}, {'res4b13x'}, {});

    net.addLayer('res4b14_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b13x'}, {'res4b14_branch2a'},  {'res4b14_branch2af'});
        net.addLayer('bn_res4b14_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b14_branch2a'}, {'res4b14_branch2ax'}, {'res4b14_branch2axf', 'res4b14_branch2axb',
'res4b14_branch2axm'});
    net.addLayer('res4b14_branch2a_relu', dagnn.ReLU(), {'res4b14_branch2ax'}, {'res4b14_branch2axxx'},
{});

    net.addLayer('res4b14_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b14_branch2axxx'}, {'res4b14_branch2b'},  {'res4b14_branch2bf'});
        net.addLayer('bn_res4b14_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b14_branch2b'}, {'res4b14_branch2bx'}, {'res4b14_branch2bxf', 'res4b14_branch2bxb',
'res4b14_branch2bxm'});
    net.addLayer('res4b14_branch2b_relu', dagnn.ReLU(), {'res4b14_branch2bx'}, {'res4b14_branch2bxxx'},
{});

    net.addLayer('res4b14_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b14_branch2bxxx'}, {'res4b14_branch2c'},  {'res4b14_branch2cf'});
```

```
        net.addLayer('bn_res4b14_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b14_branch2c'}, {'res4b14_branch2cx'}, {'res4b14_branch2cxf', 'res4b14_branch2cxb',
'res4b14_branch2cxm'});

        net.addLayer('res4b14', dagnn.Sum(), {'res4b13x', 'res4b14_branch2cx'}, {'res4b14'});
    net.addLayer('res4b14_relu', dagnn.ReLU(), {'res4b14'}, {'res4b14x'}, {});

  net.addLayer('res4b15_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b14x'}, {'res4b15_branch2a'},  {'res4b15_branch2af'});
        net.addLayer('bn_res4b15_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b15_branch2a'}, {'res4b15_branch2ax'}, {'res4b15_branch2axf', 'res4b15_branch2axb',
'res4b15_branch2axm'});
  net.addLayer('res4b15_branch2a_relu', dagnn.ReLU(), {'res4b15_branch2ax'}, {'res4b15_branch2axxx'},
{});

  net.addLayer('res4b15_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b15_branch2axxx'}, {'res4b15_branch2b'},  {'res4b15_branch2bf'});
        net.addLayer('bn_res4b15_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b15_branch2b'}, {'res4b15_branch2bx'}, {'res4b15_branch2bxf', 'res4b15_branch2bxb',
'res4b15_branch2bxm'});
  net.addLayer('res4b15_branch2b_relu', dagnn.ReLU(), {'res4b15_branch2bx'}, {'res4b15_branch2bxxx'},
{});

  net.addLayer('res4b15_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b15_branch2bxxx'}, {'res4b15_branch2c'},  {'res4b15_branch2cf'});
        net.addLayer('bn_res4b15_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b15_branch2c'}, {'res4b15_branch2cx'}, {'res4b15_branch2cxf', 'res4b15_branch2cxb',
'res4b15_branch2cxm'});

        net.addLayer('res4b15', dagnn.Sum(), {'res4b14x', 'res4b15_branch2cx'}, {'res4b15'});
    net.addLayer('res4b15_relu', dagnn.ReLU(), {'res4b15'}, {'res4b15x'}, {});

  net.addLayer('res4b16_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b15x'}, {'res4b16_branch2a'},  {'res4b16_branch2af'});
        net.addLayer('bn_res4b16_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b16_branch2a'}, {'res4b16_branch2ax'}, {'res4b16_branch2axf', 'res4b16_branch2axb',
'res4b16_branch2axm'});
  net.addLayer('res4b16_branch2a_relu', dagnn.ReLU(), {'res4b16_branch2ax'}, {'res4b16_branch2axxx'},
{});

  net.addLayer('res4b16_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b16_branch2axxx'}, {'res4b16_branch2b'},  {'res4b16_branch2bf'});
        net.addLayer('bn_res4b16_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b16_branch2b'}, {'res4b16_branch2bx'}, {'res4b16_branch2bxf', 'res4b16_branch2bxb',
'res4b16_branch2bxm'});
  net.addLayer('res4b16_branch2b_relu', dagnn.ReLU(), {'res4b16_branch2bx'}, {'res4b16_branch2bxxx'},
{});

  net.addLayer('res4b16_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b16_branch2bxxx'}, {'res4b16_branch2c'},  {'res4b16_branch2cf'});
        net.addLayer('bn_res4b16_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b16_branch2c'}, {'res4b16_branch2cx'}, {'res4b16_branch2cxf', 'res4b16_branch2cxb',
'res4b16_branch2cxm'});

        net.addLayer('res4b16', dagnn.Sum(), {'res4b15x', 'res4b16_branch2cx'}, {'res4b16'});
    net.addLayer('res4b16_relu', dagnn.ReLU(), {'res4b16'}, {'res4b16x'}, {});

  net.addLayer('res4b17_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b16x'}, {'res4b17_branch2a'},  {'res4b17_branch2af'});
```

```
        net.addLayer('bn_res4b17_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b17_branch2a'}, {'res4b17_branch2ax'}, {'res4b17_branch2axf', 'res4b17_branch2axb',
'res4b17_branch2axm'});
    net.addLayer('res4b17_branch2a_relu', dagnn.ReLU(), {'res4b17_branch2ax'}, {'res4b17_branch2axxx'},
{});

    net.addLayer('res4b17_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b17_branch2axxx'}, {'res4b17_branch2b'},  {'res4b17_branch2bf'});
        net.addLayer('bn_res4b17_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b17_branch2b'}, {'res4b17_branch2bx'}, {'res4b17_branch2bxf', 'res4b17_branch2bxb',
'res4b17_branch2bxm'});
    net.addLayer('res4b17_branch2b_relu', dagnn.ReLU(), {'res4b17_branch2bx'}, {'res4b17_branch2bxxx'},
{});

    net.addLayer('res4b17_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b17_branch2bxxx'}, {'res4b17_branch2c'},  {'res4b17_branch2cf'});
        net.addLayer('bn_res4b17_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b17_branch2c'}, {'res4b17_branch2cx'}, {'res4b17_branch2cxf', 'res4b17_branch2cxb',
'res4b17_branch2cxm'});

        net.addLayer('res4b17', dagnn.Sum(), {'res4b16x', 'res4b17_branch2cx'}, {'res4b17'});
    net.addLayer('res4b17_relu', dagnn.ReLU(), {'res4b17'}, {'res4b17x'}, {});

    net.addLayer('res4b18_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b17x'}, {'res4b18_branch2a'},  {'res4b18_branch2af'});
        net.addLayer('bn_res4b18_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b18_branch2a'}, {'res4b18_branch2ax'}, {'res4b18_branch2axf', 'res4b18_branch2axb',
'res4b18_branch2axm'});
    net.addLayer('res4b18_branch2a_relu', dagnn.ReLU(), {'res4b18_branch2ax'}, {'res4b18_branch2axxx'},
{});

    net.addLayer('res4b18_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b18_branch2axxx'}, {'res4b18_branch2b'},  {'res4b18_branch2bf'});
        net.addLayer('bn_res4b18_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b18_branch2b'}, {'res4b18_branch2bx'}, {'res4b18_branch2bxf', 'res4b18_branch2bxb',
'res4b18_branch2bxm'});
    net.addLayer('res4b18_branch2b_relu', dagnn.ReLU(), {'res4b18_branch2bx'}, {'res4b18_branch2bxxx'},
{});

    net.addLayer('res4b18_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b18_branch2bxxx'}, {'res4b18_branch2c'},  {'res4b18_branch2cf'});
        net.addLayer('bn_res4b18_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b18_branch2c'}, {'res4b18_branch2cx'}, {'res4b18_branch2cxf', 'res4b18_branch2cxb',
'res4b18_branch2cxm'});

        net.addLayer('res4b18', dagnn.Sum(), {'res4b17x', 'res4b18_branch2cx'}, {'res4b18'});
    net.addLayer('res4b18_relu', dagnn.ReLU(), {'res4b18'}, {'res4b18x'}, {});

    net.addLayer('res4b19_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b18x'}, {'res4b19_branch2a'},  {'res4b19_branch2af'});
        net.addLayer('bn_res4b19_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b19_branch2a'}, {'res4b19_branch2ax'}, {'res4b19_branch2axf', 'res4b19_branch2axb',
'res4b19_branch2axm'});
    net.addLayer('res4b19_branch2a_relu', dagnn.ReLU(), {'res4b19_branch2ax'}, {'res4b19_branch2axxx'},
{});

    net.addLayer('res4b19_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b19_branch2axxx'}, {'res4b19_branch2b'},  {'res4b19_branch2bf'});
```

```
            net.addLayer('bn_res4b19_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b19_branch2b'}, {'res4b19_branch2bx'}, {'res4b19_branch2bxf', 'res4b19_branch2bxb',
'res4b19_branch2bxm'});
    net.addLayer('res4b19_branch2b_relu', dagnn.ReLU(), {'res4b19_branch2bx'}, {'res4b19_branch2bxxx'},
{});

    net.addLayer('res4b19_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b19_branch2bxxx'}, {'res4b19_branch2c'},  {'res4b19_branch2cf'});
            net.addLayer('bn_res4b19_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b19_branch2c'}, {'res4b19_branch2cx'}, {'res4b19_branch2cxf', 'res4b19_branch2cxb',
'res4b19_branch2cxm'});

            net.addLayer('res4b19', dagnn.Sum(), {'res4b18x', 'res4b19_branch2cx'}, {'res4b19'});
        net.addLayer('res4b19_relu', dagnn.ReLU(), {'res4b19'}, {'res4b19x'}, {});

    net.addLayer('res4b20_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b19x'}, {'res4b20_branch2a'},  {'res4b20_branch2af'});
            net.addLayer('bn_res4b20_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b20_branch2a'}, {'res4b20_branch2ax'}, {'res4b20_branch2axf', 'res4b20_branch2axb',
'res4b20_branch2axm'});
    net.addLayer('res4b20_branch2a_relu', dagnn.ReLU(), {'res4b20_branch2ax'}, {'res4b20_branch2axxx'},
{});

    net.addLayer('res4b20_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b20_branch2axxx'}, {'res4b20_branch2b'},  {'res4b20_branch2bf'});
            net.addLayer('bn_res4b20_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b20_branch2b'}, {'res4b20_branch2bx'}, {'res4b20_branch2bxf', 'res4b20_branch2bxb',
'res4b20_branch2bxm'});
    net.addLayer('res4b20_branch2b_relu', dagnn.ReLU(), {'res4b20_branch2bx'}, {'res4b20_branch2bxxx'},
{});

    net.addLayer('res4b20_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b20_branch2bxxx'}, {'res4b20_branch2c'},  {'res4b20_branch2cf'});
            net.addLayer('bn_res4b20_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b20_branch2c'}, {'res4b20_branch2cx'}, {'res4b20_branch2cxf', 'res4b20_branch2cxb',
'res4b20_branch2cxm'});

            net.addLayer('res4b20', dagnn.Sum(), {'res4b19x', 'res4b20_branch2cx'}, {'res4b20'});
        net.addLayer('res4b20_relu', dagnn.ReLU(), {'res4b20'}, {'res4b20x'}, {});

    net.addLayer('res4b21_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b20x'}, {'res4b21_branch2a'},  {'res4b21_branch2af'});
            net.addLayer('bn_res4b21_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b21_branch2a'}, {'res4b21_branch2ax'}, {'res4b21_branch2axf', 'res4b21_branch2axb',
'res4b21_branch2axm'});
    net.addLayer('res4b21_branch2a_relu', dagnn.ReLU(), {'res4b21_branch2ax'}, {'res4b21_branch2axxx'},
{});

    net.addLayer('res4b21_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b21_branch2axxx'}, {'res4b21_branch2b'},  {'res4b21_branch2bf'});
            net.addLayer('bn_res4b21_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b21_branch2b'}, {'res4b21_branch2bx'}, {'res4b21_branch2bxf', 'res4b21_branch2bxb',
'res4b21_branch2bxm'});
    net.addLayer('res4b21_branch2b_relu', dagnn.ReLU(), {'res4b21_branch2bx'}, {'res4b21_branch2bxxx'},
{});

    net.addLayer('res4b21_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b21_branch2bxxx'}, {'res4b21_branch2c'},  {'res4b21_branch2cf'});
```

```
        net.addLayer('bn_res4b21_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b21_branch2c'}, {'res4b21_branch2cx'}, {'res4b21_branch2cxf', 'res4b21_branch2cxb',
'res4b21_branch2cxm'});

        net.addLayer('res4b21', dagnn.Sum(), {'res4b20x', 'res4b21_branch2cx'}, {'res4b21'});
    net.addLayer('res4b21_relu', dagnn.ReLU(), {'res4b21'}, {'res4b21x'}, {});

  net.addLayer('res4b22_branch2a', dagnn.Conv('size', [1 1 1024 256], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b21x'}, {'res4b22_branch2a'},  {'res4b22_branch2af'});
        net.addLayer('bn_res4b22_branch2a', dagnn.BatchNorm('numChannels', 256),
{'res4b22_branch2a'}, {'res4b22_branch2ax'}, {'res4b22_branch2axf', 'res4b22_branch2axb',
'res4b22_branch2axm'});
  net.addLayer('res4b22_branch2a_relu', dagnn.ReLU(), {'res4b22_branch2ax'}, {'res4b22_branch2axxx'},
{});

  net.addLayer('res4b22_branch2b', dagnn.Conv('size', [3 3 256 256], 'hasBias', false, 'stride', [1, 1], 'pad', [1
1 1 1]), {'res4b22_branch2axxx'}, {'res4b22_branch2b'},  {'res4b22_branch2bf'});
        net.addLayer('bn_res4b22_branch2b', dagnn.BatchNorm('numChannels', 256),
{'res4b22_branch2b'}, {'res4b22_branch2bx'}, {'res4b22_branch2bxf', 'res4b22_branch2bxb',
'res4b22_branch2bxm'});
  net.addLayer('res4b22_branch2b_relu', dagnn.ReLU(), {'res4b22_branch2bx'}, {'res4b22_branch2bxxx'},
{});

  net.addLayer('res4b22_branch2c', dagnn.Conv('size', [1 1 256 1024], 'hasBias', false, 'stride', [1, 1], 'pad',
[0 0 0 0]), {'res4b22_branch2bxxx'}, {'res4b22_branch2c'},  {'res4b22_branch2cf'});
        net.addLayer('bn_res4b22_branch2c', dagnn.BatchNorm('numChannels', 1024),
{'res4b22_branch2c'}, {'res4b22_branch2cx'}, {'res4b22_branch2cxf', 'res4b22_branch2cxb',
'res4b22_branch2cxm'});

        net.addLayer('res4b22', dagnn.Sum(), {'res4b21x', 'res4b22_branch2cx'}, {'res4b22'});
    net.addLayer('res4b22_relu', dagnn.ReLU(), {'res4b22'}, {'res4b22x'}, {});


  net.addLayer('res5a_branch1', dagnn.Conv('size', [1 1 1024 2048], 'hasBias', false, 'stride', [2, 2], 'pad', [0 0
0 0]), {'res4b22x'}, {'res5a_branch1'},  {'res5a_branch1f'});
        net.addLayer('bn_res5a_branch1', dagnn.BatchNorm('numChannels', 2048), {'res5a_branch1'},
{'res5a_branch1x'}, {'res5a_branch1xf', 'res5a_branch1xb', 'res5a_branch1xm'});

  net.addLayer('res5a_branch2a', dagnn.Conv('size', [1 1 1024 512], 'hasBias', false, 'stride', [2, 2], 'pad', [0 0
0 0]), {'res4b22x'}, {'res5a_branch2a'},  {'res5a_branch2af'});
        net.addLayer('bn_res5a_branch2a', dagnn.BatchNorm('numChannels', 512), {'res5a_branch2a'},
{'res5a_branch2ax'}, {'res5a_branch2axf', 'res5a_branch2axb', 'res5a_branch2axm'});
  net.addLayer('res5a_branch2a_relu', dagnn.ReLU(), {'res5a_branch2ax'}, {'res5a_branch2axxx'}, {});

  net.addLayer('res5a_branch2b', dagnn.Conv('size', [3 3 512 512], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res5a_branch2axxx'}, {'res5a_branch2b'},  {'res5a_branch2bf'});
        net.addLayer('bn_res5a_branch2b', dagnn.BatchNorm('numChannels', 512), {'res5a_branch2b'},
{'res5a_branch2bx'}, {'res5a_branch2bxf', 'res5a_branch2bxb', 'res5a_branch2bxm'});
  net.addLayer('res5a_branch2b_relu', dagnn.ReLU(), {'res5a_branch2bx'}, {'res5a_branch2bxxx'}, {});

  net.addLayer('res5a_branch2c', dagnn.Conv('size', [1 1 512 2048], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res5a_branch2bxxx'}, {'res5a_branch2c'},  {'res5a_branch2cf'});
        net.addLayer('bn_res5a_branch2c', dagnn.BatchNorm('numChannels', 2048), {'res5a_branch2c'},
{'res5a_branch2cx'}, {'res5a_branch2cxf', 'res5a_branch2cxb', 'res5a_branch2cxm'});

        net.addLayer('res5a', dagnn.Sum(), {'res5a_branch1x', 'res5a_branch2cx'}, {'res5a'});
    net.addLayer('res5a_relu', dagnn.ReLU(), {'res5a'}, {'res5ax'}, {});

  net.addLayer('res5b_branch2a', dagnn.Conv('size', [1 1 2048 512], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res5ax'}, {'res5b_branch2a'},  {'res5b_branch2af'});
```

```
        net.addLayer('bn_res5b_branch2a', dagnn.BatchNorm('numChannels', 512), {'res5b_branch2a'},
{'res5b_branch2ax'}, {'res5b_branch2axf', 'res5b_branch2axb', 'res5b_branch2axm'});
    net.addLayer('res5b_branch2a_relu', dagnn.ReLU(), {'res5b_branch2ax'}, {'res5b_branch2axxx'}, {});

    net.addLayer('res5b_branch2b', dagnn.Conv('size', [3 3 512 512], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res5b_branch2axxx'}, {'res5b_branch2b'},  {'res5b_branch2bf'});
        net.addLayer('bn_res5b_branch2b', dagnn.BatchNorm('numChannels', 512), {'res5b_branch2b'},
{'res5b_branch2bx'}, {'res5b_branch2bxf', 'res5b_branch2bxb', 'res5b_branch2bxm'});
    net.addLayer('res5b_branch2b_relu', dagnn.ReLU(), {'res5b_branch2bx'}, {'res5b_branch2bxxx'}, {});

    net.addLayer('res5b_branch2c', dagnn.Conv('size', [1 1 512 2048], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res5b_branch2bxxx'}, {'res5b_branch2c'},  {'res5b_branch2cf'});
        net.addLayer('bn_res5b_branch2c', dagnn.BatchNorm('numChannels', 2048), {'res5b_branch2c'},
{'res5b_branch2cx'}, {'res5b_branch2cxf', 'res5b_branch2cxb', 'res5b_branch2cxm'});

        net.addLayer('res5b', dagnn.Sum(), {'res5ax', 'res5b_branch2cx'}, {'res5b'});
    net.addLayer('res5b_relu', dagnn.ReLU(), {'res5b'}, {'res5bx'}, {});

    net.addLayer('res5c_branch2a', dagnn.Conv('size', [1 1 2048 512], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res5bx'}, {'res5c_branch2a'},  {'res5c_branch2af'});
        net.addLayer('bn_res5c_branch2a', dagnn.BatchNorm('numChannels', 512), {'res5c_branch2a'},
{'res5c_branch2ax'}, {'res5c_branch2axf', 'res5c_branch2axb', 'res5c_branch2axm'});
    net.addLayer('res5c_branch2a_relu', dagnn.ReLU(), {'res5c_branch2ax'}, {'res5c_branch2axxx'}, {});

    net.addLayer('res5c_branch2b', dagnn.Conv('size', [3 3 512 512], 'hasBias', false, 'stride', [1, 1], 'pad', [1 1
1 1]), {'res5c_branch2axxx'}, {'res5c_branch2b'},  {'res5c_branch2bf'});
        net.addLayer('bn_res5c_branch2b', dagnn.BatchNorm('numChannels', 512), {'res5c_branch2b'},
{'res5c_branch2bx'}, {'res5c_branch2bxf', 'res5c_branch2bxb', 'res5c_branch2bxm'});
    net.addLayer('res5c_branch2b_relu', dagnn.ReLU(), {'res5c_branch2bx'}, {'res5c_branch2bxxx'}, {});

    net.addLayer('res5c_branch2c', dagnn.Conv('size', [1 1 512 2048], 'hasBias', false, 'stride', [1, 1], 'pad', [0 0
0 0]), {'res5c_branch2bxxx'}, {'res5c_branch2c'},  {'res5c_branch2cf'});
        net.addLayer('bn_res5c_branch2c', dagnn.BatchNorm('numChannels', 2048), {'res5c_branch2c'},
{'res5c_branch2cx'}, {'res5c_branch2cxf', 'res5c_branch2cxb', 'res5c_branch2cxm'});

        net.addLayer('res5c', dagnn.Sum(), {'res5bx', 'res5c_branch2cx'}, {'res5c'});
    net.addLayer('res5c_relu', dagnn.ReLU(), {'res5c'}, {'res5cx'}, {});


    net.addLayer('pool5', dagnn.Pooling('method', 'avg', 'poolSize', [7, 7], 'stride', [1 1], 'pad', [0 0 0 0]),
{'res5cx'}, {'pool5'}, {});

    net.addLayer('classifier', dagnn.Conv('size', [1 1 2048 530], 'hasBias', true, 'stride', [1, 1], 'pad', [0 0 0 0]),
{'pool5'}, {'classifier'},  {'conv_clsf'  'conv_clsb'});

    net.addLayer('prob', dagnn.SoftMax(), {'classifier'}, {'prob'}, {});
        net.addLayer('objective', dagnn.Loss('loss', 'log'), {'prob', 'label'}, {'objective'}, {});
    net.addLayer('error', dagnn.Loss('loss', 'classerror'), {'prob','label'}, 'error') ;
% initialization of the weights
    if(numel(varargin) > 0)
                    initNet_FineTuning(net, netPre);

        else
                     initNet_He(net);
      %initNet_xavier(net);
        end

    %train
    info = cnn_train_dag(net, imdb, @(i,b) getBatchDisk(bopts,i,b), opts.train, 'val', find(imdb.images.set ==
2)) ;
```

```
end

function initNet_FineTuning(net, netPre)
          net.initParams();
          ind = 1;
          for l=1:length(net.layers)-4
                    % is a convolution layer?
                    if(strcmp(class(net.layers(l).block), 'dagnn.Conv'))
        f_ind = net.layers(l).paramIndexes(1);
                              %b_ind = net.layers(l).paramIndexes(2);
         net.params(f_ind).value = netPre.params(f_ind).value;
                              %ind = ind + 1;
%                             net.params(b_ind).value = netPre.params(b_ind).value;
%                             ind = ind + 1;

          if l<= length(net.layers)-4
                              net.params(f_ind).learningRate = 1e-1;
                              net.params(f_ind).weightDecay = 1;

        else
          net.params(f_ind).learningRate = 1;
                              net.params(f_ind).weightDecay = 1;

         end
      end
      if(strcmp(class(net.layers(l).block), 'dagnn.BatchNorm'))
        f_indbn = net.layers(l).paramIndexes(1);
                              b_indbn = net.layers(l).paramIndexes(2);
                              m_indbn = net.layers(l).paramIndexes(3);

         net.params(f_indbn).value = netPre.params(f_indbn).value;
                              %ind = ind + 1;
                              net.params(b_indbn).value = netPre.params(b_indbn).value;
                              %ind = ind + 1;
          net.params(m_indbn).value = netPre.params(m_indbn).value;
                              %ind = ind + 1;

          net.params(f_indbn).learningRate = 1;
                              net.params(f_indbn).weightDecay = 1;
          net.params(b_indbn).learningRate = 1;
                              net.params(b_indbn).weightDecay = 1;
          net.params(m_indbn).learningRate = 0.01;
                              net.params(m_indbn).weightDecay = 1;
      end
           end

          for l=length(net.layers)-3:length(net.layers)
                    % is a convolution layer?
                    if(strcmp(class(net.layers(l).block), 'dagnn.Conv'))
                              f_ind = net.layers(l).paramIndexes(1);
                              b_ind = net.layers(l).paramIndexes(2);


                              [h,w,in,out] = size(net.params(f_ind).value);
                              he_gain = 1e-2*sqrt(2/(h*w*in)); % sqrt(1/fan_in)
                              net.params(f_ind).value = 2*he_gain*randn(size(net.params(f_ind).value), 'single');
                              net.params(f_ind).learningRate = 1;
                              net.params(f_ind).weightDecay = 1;

                              net.params(b_ind).value = zeros(size(net.params(b_ind).value), 'single');
```

```
                net.params(b_ind).learningRate = 1;
                net.params(b_ind).weightDecay = 1;

            end
    end
end


% getBatch for IMDBs that are too big to be in RAM
function inputs = getBatchDisk(opts, imdb, batch)
        images = zeros(224, 224, 3, 'single');
for i=1:numel(batch)
        im = imread(imdb.images.filenames{batch(i)}) ;
    im_ = single(im) ; im_=imresize(im_, [224 224]);
    im_= im_ - imdb.images.data_mean;
        images(:,:,:, i) = im_;
end
clear im; clear im_;

        labels = imdb.images.labels(1,batch) ;
        if opts.useGpu > 0
            images = gpuArray(images) ;
        end

        inputs = {'input', images, 'label', labels} ;
end
```

8. Test code

```
function [class, score, correct, errors] = test_inference(imdb, net)
test_images = find(imdb.images.set == 3);
size(test_images)
correct=0; errors=0;
j=1;
for i=1:length(test_images)
    im=imread(imdb.images.filenames{test_images(i)});
    im_ = single(im) ; im_=imresize(im_, [224 224]);
        im_ = im_ - imdb.images.data_mean;
        i
        % run the CNN
        net.conserveMemory = false;
         net.mode='test';
        net.eval({'input', im_});

        % obtain the CNN otuput
        scores = net.vars(net.getVarIndex('prob')).value;
        scores = squeeze(gather(scores));

        % show the classification results
        [score, class] = max(scores);
PersonID=imdb.images.labels(test_images(i));
if PersonID==class
   correct=correct+1;
else
       errors=errors+1;
end
%       %figure() ; clf ; imshow(uint8(im));
        %title(sprintf('%s (%d), score %.3f', net.meta.classes.description{class}, class, score));
 end
```

```
end
```

Glossary

BN= Batch normalzation

CFLR= learning rate classifier

Epoch: a training computation of the whole database

ICP= Inception module

IPG= Image Processing Group

LR= learning rate

w_d=weight decay