Intelligent Systems Lab

Michele Baldassini
Department of Information Engineering – University of Pisa

 michele.baldassini@unifi.it

Convolutional Neural Networks

Structure

- The convolutional network consists of multiple layers. Each layer has a specific purpose. The layers may be repeated with different parameters as part of the convolutional network. The layer types we will use are

1. imageInputLayer
2. convolution2dLayer
3. batchNormalizationLayer
4. reluLayer
5. maxPooling2dLayer
6. fullyConnectedLayer
7. softmaxLayer
8. classificationLayer

imageInputLayer

- This tells the network the size of the images. For example:

layer = imageInputLayer([28 28 3]);

says the image is RGB and 28 by 28 pixels.

convolution2dLayer (1/5)

- Convolution is the process of highlighting expected features in an image. This layer applies sliding convolutional filters to an image to extract features. You can specify the filters and the stride. Convolution is a matrix multiplication operation. You define the size of the matrices and their contents. For most images, like images of faces, you need multiple filters. Some types of filters are:

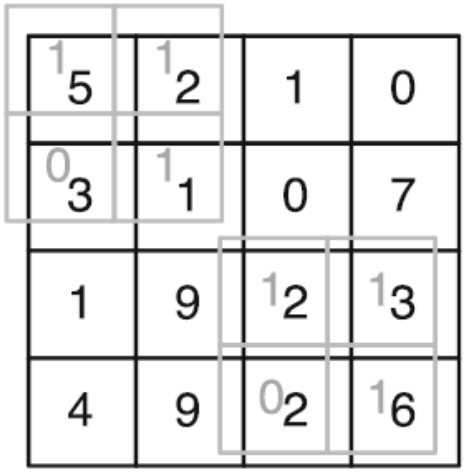
1. Blurring filter	**ones(3,3)/9**
2. Sharpening filter	**[0 -1 0;-1 5 -1;0 -1 0]**
3. Horizontal Sobel filter for edge detection	**[-1 -2 -1; 0 0 0; 1 2 1]**
4. Vertical Sobel filter for edge detection	**[-1 0 1;-2 0 2;-1 0 1]**

convolution2dLayer (2/5)

- We create an n-by-n mask that we apply to an m-by-m matrix of data where m is greater than n. We start in the upper left corner of the matrix. We multiply the mask by the corresponding elements in the input matrix and do a sum, that is the first element of the convolved output. We then move it column by column until the highest column of the mask is aligned with the highest column of the input matrix. We then return to the first column and increment the row. We continue until we have traversed the entire input matrix and our mask is aligned with the maximum row and maximum column. The mask represents a feature. In effect we are seeing if the feature appears in different areas of the image. Here is an example.

convolution2dLayer (3/5)

Input Matrix



Mask

Convolution Matrix

8	3	8
13	3	10
19	13	11

convolution2dLayer (4/5)

```
%% Demonstrate convolution
filter = [1 0;1 1];
image  = [0 0 0 0 0 0; ...
          0 0 0 0 0 0; ...
          0 0 1 0 0 0; ...
          0 0 1 1 0 0; ...
          0 0 0 0 0 0];
out = zeros(3,3);
for k = 1:4
  for j = 1:4
    g = k:k+1;
    f = j:j+1;
    out(k,j) = sum(sum(filter.*image(g,f)));
  end
end
disp(out);
```

```
The 3 appears where the 'L' is in the image.
```

convolution2dLayer (5/5)

- We can have multiple masks. There is one weight for each element of the mask and there is one bias for each feature. In this case, the convolution works on the image itself. Convolutions can also be applied to the output of other convolutional layers or pooling layers. Pooling layers further condense the data. In deep learning, the masks are determined as part of the learning process. Convolution should be highlighting important features in the data. Subsequent convolution layers narrow down features. The MATLAB function has two inputs: the *filterSize*, specifying the height and width of the filters as either a scalar or an array of *[h w]*, and *numFilters*, the number of filters.

batchNormalizationLayer

- A batch normalization layer normalizes each input channel across a mini-batch. It automatically divides up the input channel into batches. This reduces the sensitivity to the initialization.

reluLayer (1/2)

- reluLayer is a layer that uses the rectified linear unit activation function.

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

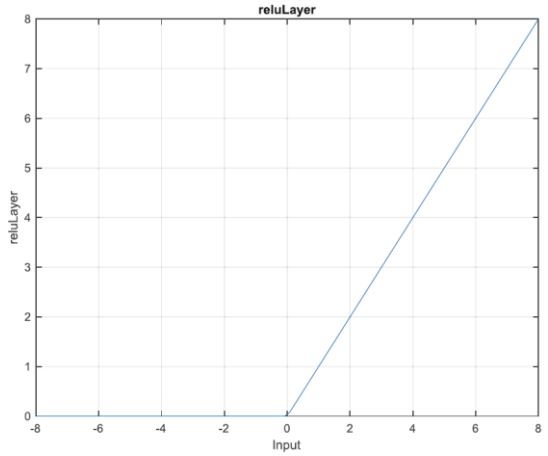
- Its derivative is

$$\frac{df}{dx} = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

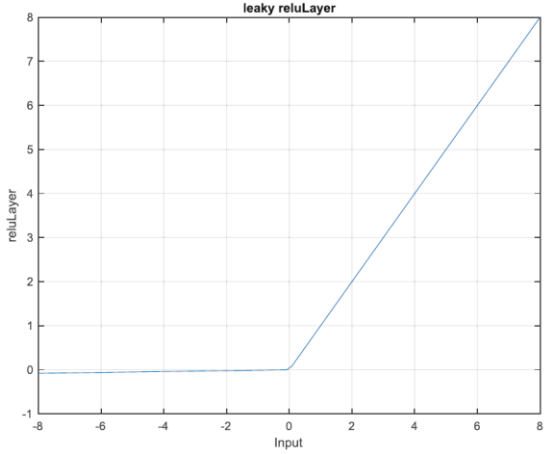
- This is very fast to compute. It says that the neuron is only activated for positive values, and the activation is linear for any value greater than zero. You can adjust the activation point with a bias.

reluLayer (2/2)

```
filter = [1 0;1 1];
x = linspace(-8,8);
y = x;
y(y<0) = 0;
plot(x, y);
```




```
filter = [1 0;1 1];
x = linspace(-8,8);
y = x;
y(y<0) = 0.01*x(y<0);
plot(x, y);
```



- An alternative is a leaky reluLayer where the value is not zero below zero. A leaky Relu layer solves the dead Relu problem where the network stops learning because the inputs to the activation function are below zero, whatever the threshold might be. It should let you worry a bit less on how you initialize the network.

maxPooling2dLayer

- maxPooling2dLayer creates a layer that breaks the 2D input into rectangular pooling regions and outputs the maximum value of each region. The input *poolSize* specifies the width and height of a pooling region. *poolSize* can have one element (for square regions) or two for rectangular regions. This is a way to reduce the number of inputs that need to be evaluated. Typical images have to be more than a mega-pixel in size, and it is not practical to use all as inputs. Furthermore, most images, or two-dimensional entities of any sort, don't really have enough information to require finely divided regions. You can experiment with pooling and see how it works for your application. An alternative is *averagePooling2dLayer*.

fullyConnectedLayer

- The fully connected layer connects all of the inputs to the outputs with weights and biases. For example:

layer = fullyConnectedLayer(10);

creates ten outputs from any number of inputs. You don't have to specify the inputs. Effectively, this is the equation:

$$y = ax + b$$

If there are m inputs and n outputs, b is a column bias matrix of length n and a is n by m.

softmaxLayer

- softmax finds a maximum of a set of values using the logistic function. The softmax is the maximum value of the set

$$p_k = \frac{e^{q_k}}{\sum e^{q_k}}$$

```
q = [1,2,3,5,1,2,3]
d = sum(exp(q));
p = exp(q)/d;
```

In this case, the maximum is element in position 4. This is just a method of smoothing the inputs. Softmax is used for multiclass classification because it guarantees a well-behaved probability distribution. Well behaved means that the sum of the probabilities is 1.

classificationLayer (1/2)

- A classification layer computes the cross-entropy loss for multiclass classification problems with mutually exclusive classes. Let us define loss. Loss is the sum of the errors in training the neural net. It is not a percentage. For classification the loss is usually the negative log likelihood, which is

$$L(y) = -\log(y)$$

where y is the output of the softmax layer.
For regression it is the residual sum of squares. A high loss means a bad fit.

classificationLayer (2/2)

- Cross-entropy loss means that an item being classified can only be in one class. The number of classes is inferred from the output of the previous layer. In the following problem, we have only two classes, circle or ellipse, so the number of outputs of the previous layer must be 2. Cross-entropy is the distance between the original probability distribution and what the model believes it should be. It is defined as

$$H(y,p) = -\sum_{i} y_i \log(p_i)$$

where i is the index for the class. It is a widely used replacement for mean squared error. It is used in neural nets where softmax activations are in the output layer.

Structuring the Layers (1/2)

- For our first net to identify circles, we will use the following set of layers. The first layer is the input layer, for the 32x32 images. These are relatively low-resolution images. You can visually determine which are ellipses or circles so we would expect the neural network to be able to do the same. Nonetheless, the size of the input images is an important consideration. In our case, our images are tightly cropped around the shape. In a more general problem, the subject of interest, a cat, for example, might be in a general setting. We use a *convolution2dLayer*, *batchNormalizationLayer*, and *reluLayer* in sequence, with a pool layer in between. There are three sets of convolution layers, each with an increasing number of filters. The output set of layers consists of a *fullyConnectedLayer*, *softmaxLayer*, and finally, the *classificationLayer*.

Structuring the Layers (2/2)

```
% Define the layers for the net
% This gives the structure of the convolutional neural net
layers = [
    imageInputLayer(size(img))
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
];
```

Training (1/7)

- Once we have the data, we need to create the training and testing sets. We have 200 files with each label (0 or 1, for an ellipse or circle). We create a training set of 80% of the files and reserve the remaining as a test set using splitEachLabel. Labels could be names, like 'circle' and 'ellipse'. You are generally better off with descriptive 'labels'. After all, a 0 or 1 could mean anything. The MATLAB software handles many types of labels.

```
imds = imageDatastore('Ellipses','labels',t);
% Split the data into training and testing sets
fracTrain                = 0.8;
[imdsTrain,imdsTest]   = splitEachLabel(imds,fracTrain,'randomize');
```

Training (2/7)

- The next step is training. The *trainNetwork* function takes the data, set of layers, and options, runs the specified training algorithm, and returns the trained network. This network is then invoked with the *classify* function, as shown later. This network is a series network. The network has other methods which you can read about in the MATLAB documentation.

Training (3/7)

```
%% Training
% The mini-batch size should be less than the data set size; the mini-batch
% is used at each training iteration to evaluate gradients and update the
% weights.
options = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.01, ...
    'MiniBatchSize', 16, ...
    'MaxEpochs', 5, ...
    'Shuffle', 'every-epoch', ...
    'ValidationData', imdsTest, ...
    'ValidationFrequency', 2, ...
    'Verbose', false, ...
    'Plots', 'training-progress');

net = trainNetwork(imdsTrain,layers,options);
```

Training (4/7)

- The training options need explanation. This is a subset of the parameter pairs available for *trainingOptions*. The first input to the function, 'sgdm', specifies the training method. There are three to choose from:

1. 'sgdm' — Stochastic gradient descent with momentum
2. 'adam' — Adaptive moment estimation (ADAM)
3. 'rmsprop' — Root mean square propagation (RMSProp)

Training (6/7)

- The 'InitialLearnRate' is the initial speed of learning. Higher learn rates mean faster learning, but the training may get stuck in a suboptimal point. The default rate for 'sgdm' is 0.01.
- 'MaxEpochs' is the maximum number of epochs to be used in the training. In each epoch, the training sees the entire training set, in batches of 'MiniBatchSize'. The number of iterations in each epoch is therefore determined by the amount of data in the set and the 'MiniBatchSize'. We are using a smaller dataset so we reduce the 'MiniBatchSize' from the default of 128 to 16, which will give us 10 iterations per epoch.
- 'Shuffle' tells the training how often to shuffle the training data. If you don't shuffle, the data will always be used in the same order. Shuffling should improve the accuracy of the trained neural network.
- 'ValidationFrequency' is how often, in number of iterations, 'ValidationData' is used to test the training. This validation data will be the data we reserved for testing when using splitEachLabel. The default frequency is every 30 iterations. We can use a validation frequency for our small problem of one, two, or five iterations.
- 'Verbose' means print out status information to the command window.
- 'Plots' only has the option 'training-progress' (besides 'none').
- 'Padding' in the *convolution2dLayer* means that the output size is ceil(*inputSize*/*stride*), where *inputSize* is the height and width of the input.

Training (7/7)

- The training window runs in real time with the training process. Our network starts with a 50% accuracy since we only have two classes, circles and ellipses. The loss plot shows how well we are doing. The lower the loss, the better the neural net. The loss plot approaches zero as the accuracy approaches 100%. In this case the validation data loss and the training data loss are about the same. This indicates good fitting of the neural net with the data. If the validation data loss is greater than the training data loss, the neural net is overfitting the data. Overfitting happens when you have an overly complex neural network. You can fit the training data, but it may not perform very well with new data, such as the validation data. For example, if you have a system which really is linear, and you fit it to a cubic equation, it might fit the data well but doesn't really model the real system. If the loss is greater than the validation data loss, your neural net is underfitting. Underfitting happens when your neural net is too simple. The goal is to make both zero.

Testing

- Finally, we test the net. Remember that this is a classification problem. An image is either an ellipse or a circle. We therefore use *classify* to implement the network. *predLabels* is the output of the net, that is, the predicted labels for the test data. This is compared to the truth labels from the datastore to compute an accuracy.

```
%% Test the neural net
predLabels  = classify(net,imdsTest);
testLabels  = imdsTest.Labels;

accuracy = sum(predLabels == testLabels)/numel(testLabels);
fprintf('Accuracy is %8.2f%%\n',accuracy*100)
```

Training (different activation functions)

- We can try different activation functions. We replaced reluLayer with leakyReluLayer. The output is similar, but in this case, learning was achieved even faster than before

```
layers = [
    imageInputLayer(size(img))

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer

    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
];
```

Training (fewer layers)

- We can try fewer layers (only one set of layers).

```
layers = [
    imageInputLayer(size(img))

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
];
```

The results with only one set of layers is still pretty good.
This shows that you need to try different options with your net architecture as well.
With this size of a problem, multiple layers are not buying very much.

Analyze Network

- The one-set network is short enough that the whole thing can be visualized inside the window of *analyzeNetwork*. This function will check your layer architecture before you start training and alert you to any errors. The size of the activations and 'Learnables' is displayed explicitly.

- We trained a neural net to classify features in our images! It is critical to carefully examine your training and test data to ensure it contains the features you wish to identify. You should be prepared to experiment with your layers and training parameters as you develop nets for different problems.

Image Classification with AlexNet (1/5)

- Image classification can be done with pretrained networks. MATLAB makes it easy to access and use these networks.
- First we need to download the support packages with the *Add-On Explorer*. If you attempt to run *alexnet* or *googlenet* without having them installed, you will get a link directly to the package in the *Add-On Explorer*. You will need your MathWorks password.
- *AlexNet* is a pretrained convolutional neural network (CNN) that has been trained on approximately 1.2 million images from the ImageNet data set (http://image-net.org/index). The model has 23 layers and can classify images into 1000 object categories. It can be used for all sorts of object identification. However, if an object was not in the training, it won't be able to identify the object.

Image Classification with AlexNet (2/5)

```
%% Load the network
% Access the trained model. This is a SeriesNetwork.
net = alexnet;
net
% See details of the architecture
net.Layers
```

- There are many layers in this convolutional network. *ReLU* and *Softmax* are the activation functions. In the first layer, 'zerocenter' normalization is used. This means the images are normalized to have a mean of zero and a standard deviation of 1. Two layers are new, cross-channel normalization and grouped Convolution. Filter groups, also known as grouped convolution, were introduced with *AlexNet* in 2012. You can think of the output of each filter as a channel and filter groups as groups of the channels. Filter groups allowed more efficient parallelization across GPUs. They also improved performance. Cross-channel normalization normalizes across channels, instead of one channel at a time. The weights in each filter are determined during training. Dropout is a layer that ignores nodes, randomly, when training the weights. This prevents interdependencies between nodes. For our first example, we load an image that comes with MATLAB, of a set of peppers. We use the top left corner as input to the net. Note that each pretrained network has a fixed input image size that we can determine from the first layer.

Image Classification with AlexNet (3/5)

```
%% Load a test image and classify it
% Read the image to classify
I = imread('peppers.png');

% Adjust size of the image to the net's input layer
sz = net.Layers(1).InputSize;
I = I(1:sz(1),1:sz(2),1:sz(3));

% Classify the image using AlexNet
[label, scorePeppers] = classify(net, I);

% Show the image and the classification results
figure;
ax = gca;
imshow(I);
title(ax,label);

figure;
plot(1:length(scorePeppers),scorePeppers);
```

To learn more about this network, we print out the categories that had next highest scores, sorted from high to low.
The categories are stored in the last layer of the net in its Classes.

Image Classification with AlexNet (4/5)

```
% What other categories are similar?
disp('Categories with highest scores for Peppers:')
kPos = find(scorePeppers>0.01);
[vals,kSort] = sort(scorePeppers(kPos),'descend');
for k = 1:length(kSort)
  fprintf('%13s:\t%g\n',net.Layers(end).Classes(kPos(kSort(k))),vals(k));
end
```

The results show that the net was considering all fruits and vegetables! The Granny Smith had the next highest scores, followed by cucumber, while the fig and lemon had much smaller scores. This makes sense since Granny Smiths and cucumbers are also usually green.

Image Classification with AlexNet (5/5)

- We also have two of our own test images. One is of a cat and one of a metal box.

For the cat the selected label is tabby. It is clear that the net can recognize that the photo is of a cat, as the other highest scored categories are also kinds of cats. Although what a tiger cat might be, as distinguished from a tabby, we can't say...
The metal box proves the biggest challenge to the net. In this case, the hard disk is by far the highest score, but the score is much lower than that of the tabby cat—roughly 0.3 vs. 0.8.

Image Classification with GoogleNet (1/5)

- GoogLeNet is a pretrained model that has also been trained on a subset of the ImageNet database which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The model is trained on more than a million images, has 144 layers (a lot more than the AlexNet), and can classify images into 1000 object categories. First we load the pretrained network as before.

```
%% Load the pretrained network
net = googlenet;
net  % display the 144 layer network
```

Image Classification with GoogleNet (2/5)

- Next we test it on the image of peppers.

```
% Read the image to classify
I = imread('peppers.png');
sz = net.Layers(1).InputSize;
I = I(1:sz(1),1:sz(2),1:sz(3));
[label, scorePeppers] = classify(net, I);
imshow(I);
title(label);
% What other categories are similar?
disp('Categories with highest scores for Peppers:')
kPos = find(scorePeppers>0.01);
[vals,kSort] = sort(scorePeppers(kPos),'descend');
for k = 1:length(kSort)
  fprintf('%13s:\t%g\n',net.Layers(end).Classes(kPos(kSort(k))),vals(k));
end
```

Image Classification with GoogleNet (3/5)

- As before, the image is correctly identified as having a bell pepper, and the score is similar to AlexNet. However, the remaining categories are a little different. In this case, the cucumber, and for some reason, a maraca, scored higher than a Granny Smith. Maracas are also round and oblong.

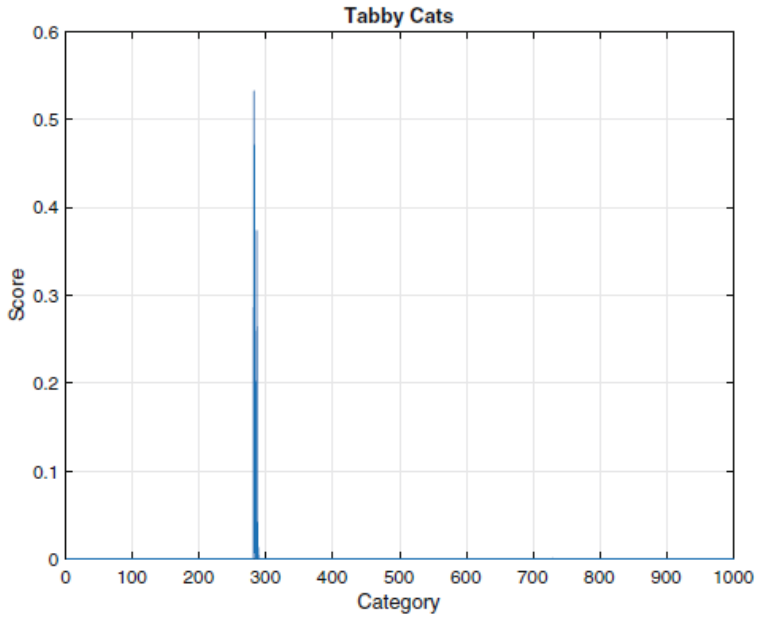
```
% Adjust size of the image
I0 = imread('Cat.png');
I  = imresize(I0,[224 224]);

% Classify the image using GoogleNet
[label, scoreCat] = classify(net, I);

% Show the image and the classification results
figure;
ax = gca;
imshow(I);
title(ax,label);

plot(1:length(scoreCat),scoreCat);


% What other categories are similar?
disp('Categories with highest scores for Cat:')
kPos = find(scoreCat>0.01);
[vals,kSort] = sort(scoreCat(kPos),'descend');
for k = 1:length(kSort)
  fprintf('%20s:\t%g\n',net.Layers(end).Classes(kPos(kSort(k))),vals(k));
end
```

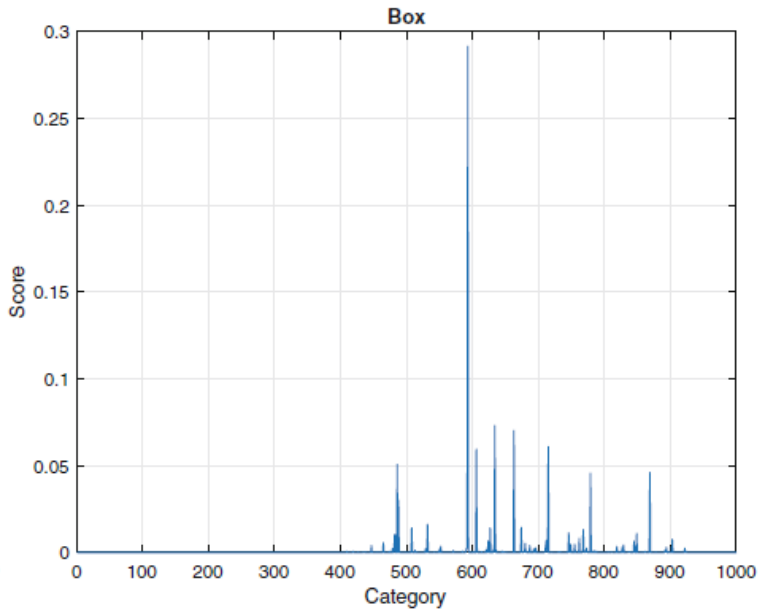
Image Classification with GoogleNet (4/5)

- We also test this net on the images of the cat and box. The image size for this network is 224x224. The categories for the cat are the same, with the addition of a lynx, and note that the tabby score is significantly lower than for AlexNet.
- The box scores prove the most interesting, and while hard disk is among the highest scores, in this case the net returns iPod. A cellular telephone is added to the mix this time. The net clearly knows that it is a rectangular metal object, but beyond that there is no clear evidence for one category over another.

Image Classification with GoogleNet (5/5)

- The box scores are visibly spread all over the place. This reinforces that the choice of 'ipod' is less certain than the pepper or cat. This shows that even highly trained networks are not necessarily reliable if the input strays too far from the test set.





Fine-Tuning (1/2)

```
% Extracting all layers except last 3.
layersTransfer = net.Layers(1:end-3);

numClasses = numel(categories(imdsTrain.Labels));

layers = [
    layersTransfer
    fullyConnectedLayer(numClasses,
        'WeightLearnRateFactor',20,'BiasLearnRateFactor',20)
    softmaxLayer
    classificationLayer];
```

Fine-Tuning (2/2)

```
numClasses = numel(categories(imdsTrain.Labels));

layers(23) = fullyConnectedLayer(numClasses,'Name','NewfullyConnectedLayer');
layers(25) = classificationLayer('Name','NewClassificationLayer');

%% Setup learning rates for fine-tuning (bump up learning rate for last layers)
layers(end-2).WeightLearnRateFactor = 10;
layers(end-2).BiasLearnRateFactor = 20;
```

For fine-tuning, we want to changed the network slightly. How much a network is changed during training is controlled by the learning rates. Here we do not modify the learning rates of the original layers, i.e. the ones before the last 3. The rates for these layers are already pretty small so they don't need to be lowered further. You could even freeze the weights of these early layers by setting the rates to zero.
Instead we boost the learning rates of the new layers we added, so that they change faster than the rest of the network. This way earlier layers don't change that much and we quickly learn the weights of the newer layer. We want the last layer to train faster than the other layers because bias the training process to quickly improve the last layer and keep the other layers relatively unchanged