http://bit.ly/2Kg7nmX

Is It a Cat or a Dog? A Neural Network Application in OpenCV

Who is the good boy?

From time to time, a website named Kaggle hosts several competitions in the fields of Data Science and Computer Vision. One of those competitions was the Dogs vs. Cats challenge, where the objective was "to create an algorithm to distinguish dogs from cats". Although this particular challenge already has been finished, I thought that it'd give me a pretty good material to a tutorial. Let's learn how to solve this problem together using OpenCV!

Here's a live demo:

SETUP ENVIRONMENT

I'll assume that you already have OpenCV 3.0 configured in your machine (if you don't, you can do it here). Also, I'll use the Boost library to read files in a directory (you can perhaps skip it and replace my code by dirent.h. It should work in the same way). You can download Boost here. Those are the only two external libraries that I'm going to use in this tutorial.

Ok, ok, let's start by downloading the training and test sets. Click here and download the test1.zip (271.15mb). You may need to register first. After downloading, extract them to a folder of your preference. The training set will be used to adjust the parameters of our neural network (we will talk in details later), while the test set will be

http://bit.ly/2Kg7nmX

used to check the performance of our neural network (how good it is at generalizing unseen examples). Unhappily, the provided test set by Kaggle is not labeled, so we will split the training set (in the provided link) and use a part of it as our test set.

READING TRAINING SAMPLES

Let's start coding! First, let's start by reading the list of files within the training set directory:

```
#include <vector>
#include <algorithm>
#include <functional>
#include <map>
#include <set>
#include <fstream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/ml/ml.hpp>

#include <boost/filesystem.hpp>

namespace fs = boost::filesystem;

/**
* Get all files in directory (not recursive)
* @param directory Directory where the files are contained
* @return A list containing the file name of all files inside given directory
**/
std::vector<std::string> getFilesInDirectory(const std::string& directory)
{
	std::vector<std::string> files;
	fs::path root(directory);
	fs::directory_iterator it_end;
	for (fs::directory_iterator it(root); it != it_end; ++it)
	{
		if (fs::is_regular_file(it->path()))
		{
			files.push_back(it->path().string());
		}
	}
	return files;
}


int main(int argc, char** argv)
{
	if (argc != 4)
	{
		std::cerr << "Usage: <IMAGES_DIRECTORY>  <NETWORK_INPUT_LAYER_SIZE> 
		<TRAIN_SPLIT_RATIO>" << std::endl;
		exit(-1);
	}
	std::string imagesDir = argv[1];
	int networkInputSize = atoi(argv[2]);
	float trainSplitSize = atof(argv[3]);

	std::cout << "Reading training set..." << std::endl;
	double start = (double)cv::getTickCount();
	std::vector<std::string> files = getFilesInDirectory(imagesDir);
	std::random_shuffle(files.begin(), files.end());

}
```

The function `getFilesInDirectory` expects as input a directory and returns a list of filenames within this directory. In our `main`, we expect to receive three parameters from command line: The directory where our training set is stored, the size of our network input layer and the ratio of our training set (i.e., 0.75 indicates that 75% of the images within the training set will be used to train our neural network while the remaining 25% will be used to test it). We then shuffle the list of filenames (in order to prevent *bias*). Pretty straight-forward until here, aye? :)

Now we are going to iterate over each filename inside `files` and read the image associated to it. Since we will do it twice (one during the training step and another during the test step), let's create a function apart in order to modularize our code.

```
typedef std::vector<std::string>::const_iterator vec_iter;

/**
* Read images from a list of file names and returns, for each read image,
* its class name and its local descriptors
*/
void readImages(vec_iter begin, vec_iter end, std::function<void(const std::string&,
const cv::Mat&)> callback)
{
	for (auto it = begin; it != end; ++it)
	{
		std::string filename = *it;
		std::cout << "Reading image " << filename << "..." << std::endl;
		cv::Mat img = cv::imread(filename, 0);
		if (img.empty())
		{
			std::cerr << "WARNING: Could not read image." << std::endl;
			continue;
		}
		std::string classname = getClassName(filename);
		cv::Mat descriptors = getDescriptors(img);
		callback(classname, descriptors);
	}
}
```

http://bit.ly/2Kg7nmX

There it is. The `readImages` function expect to receive as input two vector iterators (one for the start of our vector and another for the end, indicating the range from which we will iterate over). Finally, it expects another third parameter, a lambda function called "callback" (lambda functions are only available on C++11, so enable it on compiler by adding the `-std=c++11` flag or `-std=c++0x` on old compilers). Now let's look more carefuly on what's happening inside this function.

We use a `for` to iterate over each filename between the limiters `begin` and `end`. For each filename, we read its associated image through the OpenCV `imread` function. The second parameter passed to `imread` indicates the color space (0 = gray scale. We don't need the color information in this example. You'll find the explanation later). After calling `imread`, we check if we could really read the image (through the `empty` method). If don't, we skip to the next filename. Otherwise, we get the class name and the descriptors associated to the read image and return them to the "callback" function. Now let's implement the `getClassName` and `getDescriptors` functions.

If you look at the files inside the training set you extracted, you will find out that they are named as "dog.XXXXX.jpg" or "cat.XXXXX.jpg". The first three letters is always the class name, where the remaining is only an identifier. So let's get those three first letters!

```
/**
* Extract the class name from a file name
*/
inline std::string getClassName(const std::string& filename)
{
	return filename.substr(filename.find_last_of('/') + 1, 3);
}
```

Now what should the `getDescriptors` function looks like? Let's figure out on the next topic.

http://bit.ly/2Kg7nmX

EXTRACTING FEATURES

There are several approaches here. We could use the **color histogram**, or perhaps the **histogram of oriented gradients**, etc., … However, I'm going through a different approach. I'm going to use the **KAZE** algorithm to extract local features from the image. Since we can't submit local features to a neural network (because the number of descriptors varies), I'm also going to use the **Bag of words** strategy in order to address this problem, turning all set of descriptors into a single **histogram of visual words**, and THAT will be used as input to our neural network. Got it? Excellent! So let's implement the getDescriptors to extract the KAZE features from an image, and later, after all KAZE features had been extracted, we'll apply the Bag of Words technique.

```
/**
* Extract local features for an image
*/
cv::Mat getDescriptors(const cv::Mat& img)
{
	cv::Ptr<cv::KAZE> kaze = cv::KAZE::create();
	std::vector<cv::KeyPoint> keypoints;
	cv::Mat descriptors;
	kaze->detectAndCompute(img, cv::noArray(), keypoints, descriptors);
	return descriptors;
}
```

Ok, now let's go back to our main:

```
struct ImageData
{
	std::string classname;
	cv::Mat bowFeatures;
};

int main(int argc, char** argv)
{
	if (argc != 4)
	{
		std::cerr << "Usage: <IMAGES_DIRECTORY>  <NETWORK_INPUT_LAYER_SIZE> 
<TRAIN_SPLIT_RATIO>" << std::endl;
		exit(-1);
	}
	std::string imagesDir = argv[1];
	int networkInputSize = atoi(argv[2]);
	float trainSplitRatio = atof(argv[3]);
```

http://bit.ly/2Kg7nmX

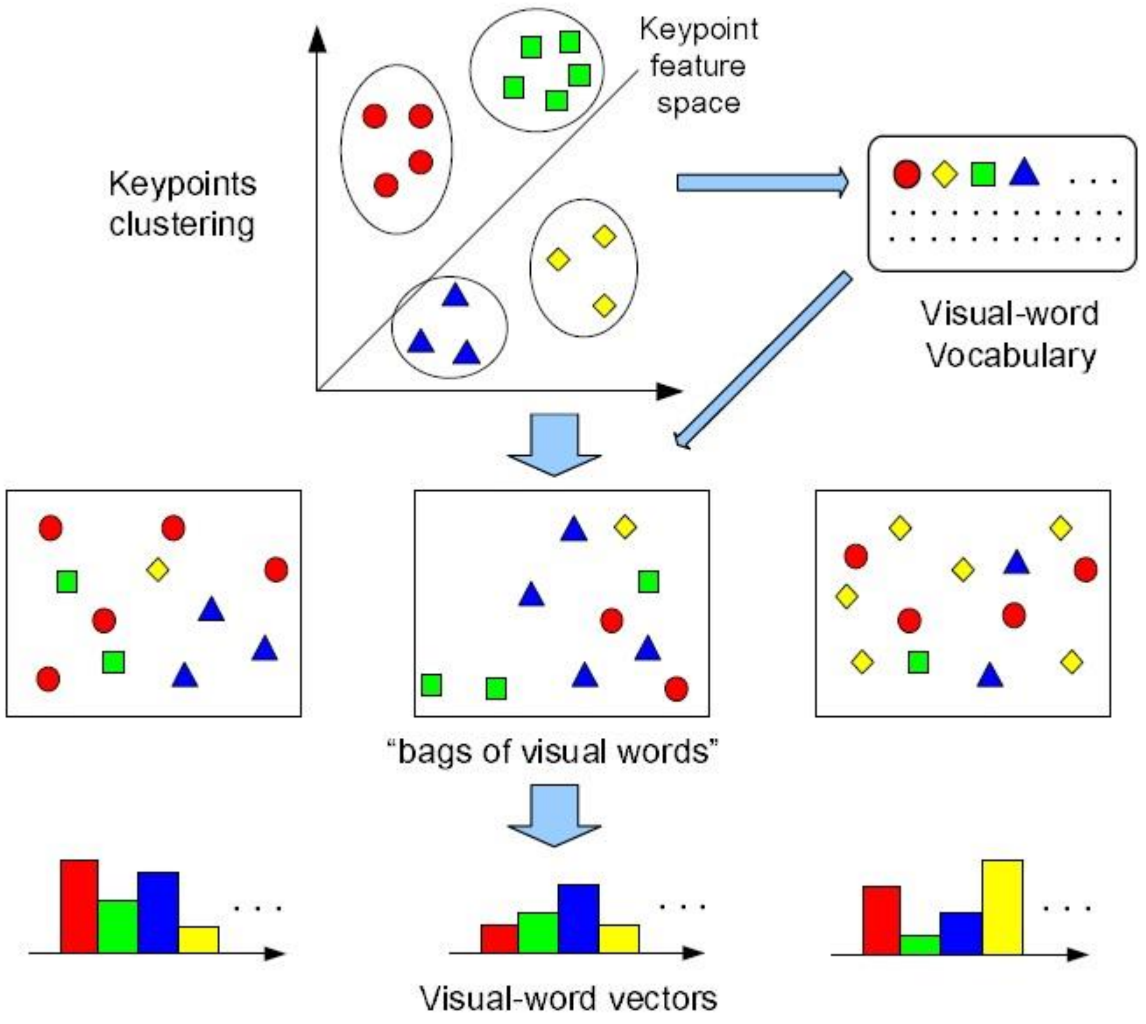
```
	std::cout << "Reading training set..." << std::endl;
	double start = (double)cv::getTickCount();
	std::vector<std::string> files = getFilesInDirectory(imagesDir);
	std::random_shuffle(files.begin(), files.end());

	cv::Mat descriptorsSet;
	std::vector<ImageData*> descriptorsMetadata;
	std::set<std::string> classes;
	readImages(files.begin(), files.begin() + (size_t)(files.size() *
trainSplitRatio),
		[&](const std::string& classname, const cv::Mat& descriptors) {
		// Append to the set of classes
		classes.insert(classname);
		// Append to the list of descriptors
		descriptorsSet.push_back(descriptors);
		// Append metadata to each extracted feature
		ImageData* data = new ImageData;
		data->classname = classname;
		data->bowFeatures = cv::Mat::zeros(cv::Size(networkInputSize, 1), CV_32F);
		for (int j = 0; j < descriptors.rows; j++)
		{
			descriptorsMetadata.push_back(data);
		}
	});
	std::cout << "Time elapsed in minutes: " << ((double)cv::getTickCount() - start) /
cv::getTickFrequency() / 60.0 << std::endl;
}
```

I created a struct named `ImageData`, with two fields: `classname` and `bowFeatures`. Before calling the `readImages` function, I instanciated three variables: `descriptorsSet` (the set of descriptors of all read images), `descriptorsMetadata` (a vector of the struct we previously created. It's being filled in such way that it has the same number of elements as the number of rows of `descriptorsSet`. That way, the i-th row of `descriptorsSet` can also be used to access its metadata (the class name, for instance)). And, for last, the `classes` variables (a set containing all found classes).

http://bit.ly/2Kg7nmX

TRAINING THE BAG OF WORDS

Now that we have the whole set of descriptors stored in the `descriptorsSet` variable, we can apply the Bag of words strategy. The Bag of Words algorithm is really simple: First we use a clustering algorithm (such as **k-means**) to obtain k centroids. Each centroid representates a **visual word** (the set of visual words is often called **vocabulary**). For each image, we create a histogram of size M, where M is the number of visual words. Now, for each extracted descriptor from the image, we measure its distance to all visual words, obtaining the index of the nearest one. We use that index to increment the position of histogram corresponding to that index, obtaining, that way, **a histogram of visual words**, that can later be submitted to our neural network.



Source: http://www.ifp.illinois.edu/~yuhuang/sceneclassification.html

http://bit.ly/2Kg7nmX

```
int main()
{
	...

		std::cout << "Creating vocabulary..." << std::endl;
	start = (double)cv::getTickCount();
	cv::Mat labels;
	cv::Mat vocabulary;
	// Use k-means to find k centroids (the words of our vocabulary)
	cv::kmeans(descriptorsSet, networkInputSize, labels,
cv::TermCriteria(cv::TermCriteria::EPS +
		cv::TermCriteria::MAX_ITER, 10, 0.01), 1, cv::KMEANS_PP_CENTERS,
vocabulary);
	// No need to keep it on memory anymore
	descriptorsSet.release();
	std::cout << "Time elapsed in minutes: " << ((double)cv::getTickCount() - start) /
cv::getTickFrequency() / 60.0 << std::endl;

	// Convert a set of local features for each image in a single descriptors
	// using the bag of words technique
	std::cout << "Getting histograms of visual words..." << std::endl;
	int* ptrLabels = (int*)(labels.data);
	int size = labels.rows * labels.cols;
	for (int i = 0; i < size; i++)
	{
		int label = *ptrLabels++;
		ImageData* data = descriptorsMetadata[i];
		data->bowFeatures.at<float>(label)++;
	}
}
```

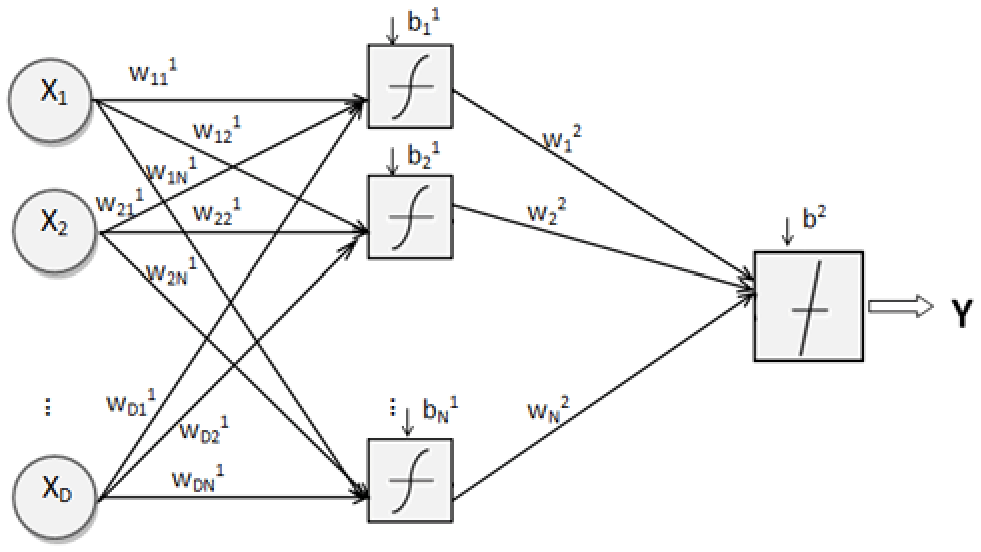
We use OpenCV `k-means` function to obtain k centroids (where k is the size of our network input layer, since the size of our histogram must be compatible with it), stored in the `vocabulary` variable. We also pass an additional parameter, `labels`, indicating the index of the nearest cluster for each descriptor, so we don't need to computer it twice. Now, iterating over each element of `labels`, we fill our histograms, the `bowFeatures` field of our `ImageData` struct. The strategy of filling the `descriptorsMetadata` to make its number of elements as the number of rows of `descriptorsSet` seemed to be very convenient here, as we can directly access the histogram associated to each descriptor.

http://bit.ly/2Kg7nmX

TRAINING THE NEURAL NETWORK

Now that we have the histogram of visual words for each image, we can finally supply them to our neural network. But, before that, we need to tell to our neural network the expected output for each image. The reason for that is simple: A neural network, or more precisely, the variation of neural network that we are interested in using, called **Multilayer perceptron**, is a **supervised learning algorithm**. A supervised learning algorithm is one that tries to estimate a function H(x) (called **hypothesis function**) that correctly maps inputs to outputs (for instance, we are considering as input the images and as output the class associated to each image - cat or dog).

So we need to supply the class name associated to each image (or, more precisely, to each histogram of visual words) in order to enable it to "learn" the pattern. However, a neural network doesn't understand categorical data. It works by showing numbers in the input layer and numbers in the output layer, and then it will try to adjust its weights in order that a function (called **activation function**) applied to the input numbers results in the output numbers. This process is shown in the image below.



Since the activation function generally outputs values between 0 and 1, it's usual to encode the classes as a sequence of zeros where only one bit is set to one. This bit is different for each class. For example, consider the example of number of classes = 4. We would then have four codifications:

Class A = 1000
Class B = 0100
Class C = 0010
Class D = 0001

http://bit.ly/2Kg7nmX

As we only have two classes, our codification will be:

Cat = 10
Dog = 01

```
int main()
{
	...
		// Filling matrixes to be used by the neural network
		std::cout << "Preparing neural network..." << std::endl;
	cv::Mat trainSamples;
	cv::Mat trainResponses;
	std::set<ImageData*> uniqueMetadata(descriptorsMetadata.begin(), 
descriptorsMetadata.end());
	for (auto it = uniqueMetadata.begin(); it != uniqueMetadata.end(); )
	{
		ImageData* data = *it;
		cv::Mat normalizedHist;
		cv::normalize(data->bowFeatures, normalizedHist, 0, data->bowFeatures.rows, 
cv::NORM_MINMAX, -1, cv::Mat());
		trainSamples.push_back(normalizedHist);
		trainResponses.push_back(getClassCode(classes, data->classname));
		delete *it; // clear memory
		it++;
	}
	descriptorsMetadata.clear();
}
```

Notice the use of the `getClassCode`. It’s a function that turns a class name into its binary codification. Also, pay attention to the `cv::normalize` function. We normalize the histogram of visual words in order to remove the bias of number of descriptors.\

```
/**
* Transform a class name into an id
*/
int getClassId(const std::set<std::string>& classes, const std::string& classname)
{
	int index = 0;
	for (auto it = classes.begin(); it != classes.end(); ++it)
	{
		if (*it == classname) break;
		++index;
	}
	return index;
}

/**
* Get a binary code associated to a class
*/
```

http://bit.ly/2Kg7nmX

```
cv::Mat getClassCode(const std::set<std::string>& classes, const std::string& classname)
{
	cv::Mat code = cv::Mat::zeros(cv::Size((int)classes.size(), 1), CV_32F);
	int index = getClassId(classes, classname);
	code.at<float>(index) = 1;
	return code;
}
```

And now we have the inputs and outputs for our neural network! We are finally able to train it!

```
/**
* Get a trained neural network according to some inputs and outputs
*/
cv::Ptr<cv::ml::ANN_MLP> getTrainedNeuralNetwork(const cv::Mat& trainSamples, const
cv::Mat& trainResponses)
{
	int networkInputSize = trainSamples.cols;
	int networkOutputSize = trainResponses.cols;
	cv::Ptr<cv::ml::ANN_MLP> mlp = cv::ml::ANN_MLP::create();
	std::vector<int> layerSizes = { networkInputSize, networkInputSize / 2,
		networkOutputSize };
	mlp->setLayerSizes(layerSizes);
	mlp->setActivationFunction(cv::ml::ANN_MLP::SIGMOID_SYM);
	mlp->train(trainSamples, cv::ml::ROW_SAMPLE, trainResponses);
	return mlp;
}

int main()
{
	...
		// Training neural network
		std::cout << "Training neural network..." << std::endl;
	start = cv::getTickCount();
	cv::Ptr<cv::ml::ANN_MLP> mlp = getTrainedNeuralNetwork(networkInputSize,
		trainSamples, trainResponses);
	std::cout << "Time elapsed in minutes: " << ((double)cv::getTickCount() - start) /
cv::getTickFrequency() / 60.0 << std::endl;

	// We can clear memory now
	trainSamples.release();
	trainResponses.release();
}
```

The getTrainedNeuralNetwork function expects to receive as input the size of training samples and training outputs. Inside the function, I first set two variables: networkInputSize, that is the number of columns (features) of our training samples and networkOutputSize, that is the number of columns of our training outputs. I

http://bit.ly/2Kg7nmX

then set layerSizes, that defines the number of layers and number of nodes for each layer of our network. For instance, I'm creating a network that only have one hidden layer (with size `networkInputSize / 2`), since I think it'll be enough for our task. If you want improved accuracy, we can increase it, at cost of performance.

EVALUATING OUR NETWORK

And now the training step is DONE! Let's use our trained neural network to evaluate our test samples and measure how good it is. First, let's train a FLANN model from the vocabulary, so we can calculate the histogram of visual words for each test sample much faster:

```
int main()
{
	...

		// Train FLANN
		std::cout << "Training FLANN..." << std::endl;
	start = cv::getTickCount();
	cv::FlannBasedMatcher flann;
	flann.add(vocabulary);
	flann.train();
	std::cout << "Time elapsed in minutes: " << ((double)cv::getTickCount() - start) /
	cv::getTickFrequency() / 60.0 << std::endl;
}
```

Now let's read the test samples:

```
int main()
{
	...
		// Reading test set
		std::cout << "Reading test set..." << std::endl;
	start = cv::getTickCount();
	cv::Mat testSamples;
	std::vector<int> testOutputExpected;
	readImages(files.begin() + (size_t)(files.size() * trainSplitRatio), files.end(),
		[&](const std::string& classname, const cv::Mat& descriptors) {
		// Get histogram of visual words using bag of words technique
		cv::Mat bowFeatures = getBOWFeatures(flann, descriptors, networkInputSize);
```

http://bit.ly/2Kg7nmX

```
              cv::normalize(bowFeatures, bowFeatures, 0, bowFeatures.rows,
cv::NORM_MINMAX, -1, cv::Mat());
              testSamples.push_back(bowFeatures);
              testOutputExpected.push_back(getClassId(classes, classname));
       });
       std::cout << "Time elapsed in minutes: " << ((double)cv::getTickCount() - start) /
cv::getTickFrequency() / 60.0 << std::endl;
}
```

We instanciated two variables: `testSamples` (set of histogram of visual words for each test samples) and `testOutputExpected` (the output expected for each test sample. We are using a number that correspond to the id of the class, obtained through the `getClassId` previously defined). We then get the Bag of Words features through the `getBOWFeatures` function and normalize it. What we still didn't define is the `getBOWFeatures` function, that turns a set of local KAZE features into a histogram of visual words. Let's do it

```
* Turn local features into a single bag of words histogram of
* of visual words (a.k.a., bag of words features)
*/
cv::Mat getBOWFeatures(cv::FlannBasedMatcher& flann, const cv::Mat& descriptors,
       int vocabularySize)
{
       cv::Mat outputArray = cv::Mat::zeros(cv::Size(vocabularySize, 1), CV_32F);
       std::vector<cv::DMatch> matches;
       flann.match(descriptors, matches);
       for (size_t j = 0; j < matches.size(); j++)
       {
              int visualWord = matches[j].trainIdx;
              outputArray.at<float>(visualWord)++;
       }
       return outputArray;
}
```

It uses the FLANN `match` method to calculate the nearest visual word for each descriptor. It then fill a histogram with the number of occurrences for each visual word. Pretty simple, right?

Now that we have the inputs and outputs for the test samples, let's calculate a **confusion matrix**.

```
/**
```

http://bit.ly/2Kg7nmX

```
* Receives a column matrix contained the probabilities associated to
* each class and returns the id of column which contains the highest
* probability
*/
int getPredictedClass(const cv::Mat& predictions)
{
	float maxPrediction = predictions.at<float>(0);
	float maxPredictionIndex = 0;
	const float* ptrPredictions = predictions.ptr<float>(0);
	for (int i = 0; i < predictions.cols; i++)
	{
		float prediction = *ptrPredictions++;
		if (prediction > maxPrediction)
		{
			maxPrediction = prediction;
			maxPredictionIndex = i;
		}
	}
	return maxPredictionIndex;
}

/**
* Get a confusion matrix from a set of test samples and their expected
* outputs
*/
std::vector<std::vector<int> > getConfusionMatrix(cv::Ptr<cv::ml::ANN_MLP> mlp,
	const cv::Mat& testSamples, const std::vector<int>& testOutputExpected)
{
	cv::Mat testOutput;
	mlp->predict(testSamples, testOutput);
	std::vector<std::vector<int> > confusionMatrix(2, std::vector<int>(2));
	for (int i = 0; i < testOutput.rows; i++)
	{
		int predictedClass = getPredictedClass(testOutput.row(i));
		int expectedClass = testOutputExpected.at(i);
		confusionMatrix[expectedClass][predictedClass]++;
	}
	return confusionMatrix;
}

/**
* Print a confusion matrix on screen
*/
void printConfusionMatrix(const std::vector<std::vector<int> >& confusionMatrix,
	const std::set<std::string>& classes)
{
	for (auto it = classes.begin(); it != classes.end(); ++it)
	{
		std::cout << *it << " ";
	}
	std::cout << std::endl;
	for (size_t i = 0; i < confusionMatrix.size(); i++)
	{
		for (size_t j = 0; j < confusionMatrix[i].size(); j++)
		{
			std::cout << confusionMatrix[i][j] << " ";
		}
```

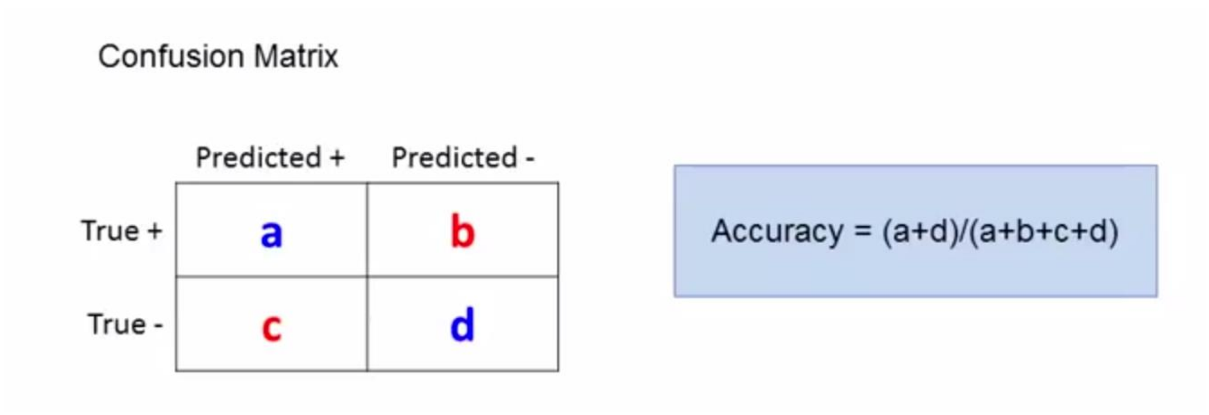
http://bit.ly/2Kg7nmX

```
              std::cout << std::endl;
       }
}

/**
* Get the accuracy for a model (i.e., percentage of correctly predicted
* test samples)
*/
float getAccuracy(const std::vector<std::vector<int> >& confusionMatrix)
{
       int hits = 0;
       int total = 0;
       for (size_t i = 0; i < confusionMatrix.size(); i++)
       {
              for (size_t j = 0; j < confusionMatrix.at(i).size(); j++)
              {
                     if (i == j) hits += confusionMatrix.at(i).at(j);
                     total += confusionMatrix.at(i).at(j);
              }
       }
       return hits / (float)total;
}


int main()
{
       ...
              // Get confusion matrix of the test set
              std::vector<std::vector<int> > confusionMatrix = getConfusionMatrix(mlp,
                     testSamples, testOutputExpected);

       // Get accuracy of our model
       std::cout << "Confusion matrix: " << std::endl;
       printConfusionMatrix(confusionMatrix, classes);
       std::cout << "Accuracy: " << getAccuracy(confusionMatrix) << std::endl;
}
```

OK, a lot happened here. Let's check it step by step. First, in the `getConfusionMatrix`, I use the MLP `predict`method to predict the class for each test sample. It returns a matrix with the same number of columns as our number of classes, where on each column lies a "probability" of the sample belong to class corresponding to that column. We use than a function called `getPredictedClass`, which is called over each row of the output of `predict` method and return the column index with highest "probability". Now that we have the predicted and expected classes, we can construct our confusion matrix by simplying incrementing the index composed by the tuple (expected, predicted).

In possess of the confusion matrix, we can easily calculate the **accuracy**, that is the ratio of correctly predicted samples, by simplying summing the diagonal of our confusion matrix (number of correct predictions) and diving by the sum of our cells of our confusion matrix (number of test samples).

http://bit.ly/2Kg7nmX



SAVING MODELS

Finally, let's save our models, so we can use it later on a production environment:

```
/**
* Save our obtained models (neural network, bag of words vocabulary
* and class names) to use it later
*/
void saveModels(cv::Ptr<cv::ml::ANN_MLP> mlp, const cv::Mat& vocabulary,
       const std::set<std::string>& classes)
{
       mlp->save("mlp.yaml");
       cv::FileStorage fs("vocabulary.yaml", cv::FileStorage::WRITE);
       fs << "vocabulary" << vocabulary;
       fs.release();
       std::ofstream classesOutput("classes.txt");
       for (auto it = classes.begin(); it != classes.end(); ++it)
       {
              classesOutput << getClassId(classes, *it) << "\t" << *it << std::endl;
       }
       classesOutput.close();
}

int main()
{
       ...

              // Save models
              std::cout << "Saving models..." << std::endl;
       saveModels(mlp, vocabulary, classes);

       return 0;
}
```

http://bit.ly/2Kg7nmX

The MLP object that its own saving function called `save` (it also has a `load` method that can later be used to load a trained neural network from a file). We save the vocabulary (since we need it in order to convert the local features into a histogram of visual words) into a file named “vocabulary.yaml”. And, finally, we also save the class names associated to each id (so we can map the output of neural network to a name). That’s it! The full code can be found below.

Compile it by calling:

```
g++ opencv_ann.cpp -std=c++0x  -I/usr/local/include/opencv -I/usr/local/include/boost -
I/usr/local/include -L/usr/local/lib -lopencv_shape -lopencv_stitching -lopencv_objdetect -
lopencv_superres -lopencv_videostab -lopencv_calib3d -lopencv_features2d -
1lopencv_highgui -lopencv_videoio -lopencv_imgcodecs -lopencv_video -lopencv_photo -
lopencv_ml -lopencv_imgproc -lopencv_flann -lopencv_core -lopencv_hal -
lboost_filesystem -lboost_system -o mlp
```

For instance, here’s the result I got from the Kaggle’s training set (using networkInputSize = 512, trainSplitRatio = 0.7)

```
Confusion matrix:
cat dog
2669 1097
1053 2681
Accuracy: 0.713333
```

Not bad! Not bad at all, considering the difficulty of some images! ;)

Source: https://picoledelimao.github.io/blog/2016/01/31/is-it-a-cat-or-dog-a-neural-network-application-in-opencv/