Multi-label Classification using a variation of VGGNet

By Sai Prashant Chintalapudi (1627482) and Keshav Kasichainula (1456090)

Part - 1: Survey

Convolutional Neural Networks

Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. The main difference is that CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

Regular neural nets don't scale well to full images. For example, an image of size 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would want to have several such hidden layers, so the parameters would add up quickly. This full connectivity is wasteful, and the huge number of parameters would quickly lead to overfitting.

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way: the neurons in a layer will only be connected to a small region of the layer before it, instead of all the neurons in a fully-connected manner. Unlike a regular Neural Network, the layers of a CNN have neurons arranged in three dimensions: width, height, depth. They perceive images as volumes; i.e. three-dimensional objects, rather than flat canvases to be measured only by width and height. That's because digital colour images have a red-blue-green (RGB) encoding, mixing those three colours to produce the colour spectrum humans perceive. A convolutional neural network ingests such images as three separate layers of colour stacked one on top of the other.

So, a convolutional network receives a normal colour image as a rectangular box whose width and height are measured by the number of pixels along those dimensions, and whose depth is three layers deep, one for each letter in RGB. Those depth layers are referred to as channels.

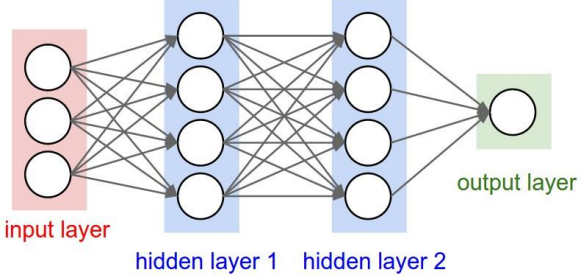

Fig 1. A regular 3-layer Neural Network.

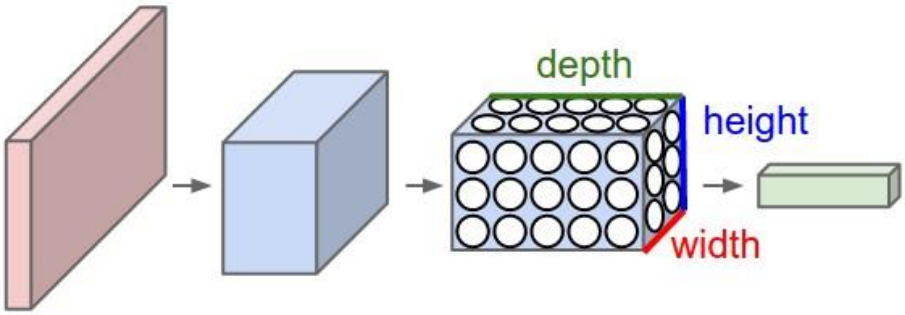

Fig 2. A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations.

Essentially, every image can be represented as a matrix of pixel values. An image from a standard digital camera will have three channels – red, green and blue. A grayscale image, on the other hand, has just one channel. The value of each pixel in the matrix will range from 0 to 255 – zero indicating black and 255 indicating white.

There are four main operations which are the basic building blocks of every CNN:

1) Convolution

CNNs derive their name from the "convolution" operator. The primary purpose of a convolution in case of a CNN is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data.

The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height) but extends through the full depth of the input volume. During the forward pass, we convolve each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2D activation map that gives the responses of that filter at every spatial position as shown in Fig 3. Now, we will have an entire set of filters in each CONV layer, and each of them will produce a separate 2D activation map. We will stack these activation maps along the depth dimension and produce the output volume.

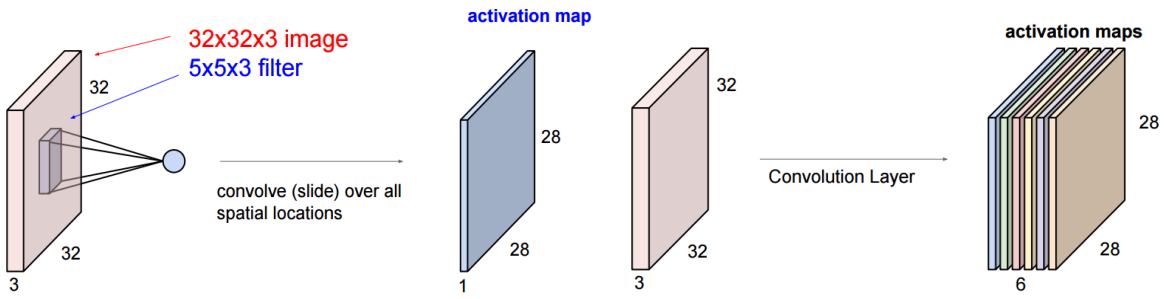

Fig 3. Stacking of activation maps as an output from a single convolutional layer.

In practice, a CNN learns the values of the filters on its own during the training process (although we still need to specify parameters such as number of filters, filter size,

architecture of the network etc. before the training process). The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

2) Non- Linearity (ReLU)

An additional operation called ReLU is used after every Convolution operation. ReLU stands for Rectified Linear Unit and is a non-linear operation. ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our CNN, since most of the real-world data we would want our CNN to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU). Other nonlinear functions such as **tanh** or **sigmoid** can also be used instead of ReLU, but ReLU has been found to perform better in most situations.

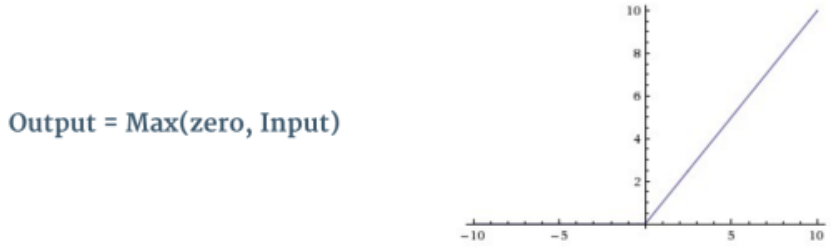

Fig 4. The Rectified Linear Unit (ReLU) function.

3) Pooling or Sub Sampling

Spatial Pooling (also called subsampling or down-sampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc. In case of Max Pooling, we define a spatial neighbourhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Pooling makes the input representations (feature dimension) smaller and more manageable. It reduces the number of parameters and computations in the network, therefore, controlling overfitting. It also makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighbourhood). It also helps us arrive at an almost scale invariant representation of our image. This is very powerful since we can detect objects in an image no matter where they are located.

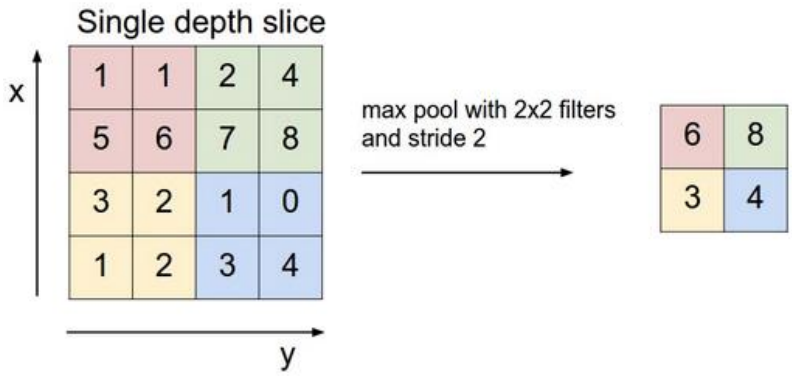

Fig 5. MaxPooling with a 2x2 filter.

4) Classification (Fully Connected Layer)

The Fully Connected layer is a traditional multi-layer perceptron that uses a SoftMax activation function in the output layer. The term "Fully Connected" implies that every neuron in the previous layer is connected to every neuron on the next layer. The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the fully connected layer is to use these features for classifying the input image into various classes based on the training dataset. The sum of output probabilities from the fully connected Layer is 1. This is ensured by using the SoftMax as the activation function in the output layer of the fully connected Layer. The SoftMax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one.

VGGNet

The VGG network architecture was introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Convolutional Networks for Large Scale Image Recognition*. They achieved state-of-the-art results in the ILSVRC-2014 challenge securing first and second in localization and classification tracks respectively.

This network is characterized by its simplicity, using only 3×3 receptive fields (which is the smallest size to capture the notion of left/right, up/down, center) stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. Two fully-connected layers, each with 4,096 nodes are then followed by a softmax classifier.

Rather than using relatively large receptive fields in the first convolution layers (e.g. 11×11 with stride 4 or 7×7 with stride 2) VGGNet uses very small 3×3 receptive fields throughout the whole net, which are convolved with the input at every pixel (with stride 1). It is easy to see that a stack of two 3×3 convolution layers (without spatial pooling in between) has an effective receptive field of 5×5; three such layers have a 7×7 effective receptive field. So, there are two advantages of using a stack of three 3×3 convolution layers instead of a single 7×7 layer. First, we incorporate three non-linear rectification layers instead of one, which makes the decision function more discriminative. Second, we decrease the number of parameters: assuming that both the input and the output of a three-layer 3×3 convolution stack has C channels, the stack is parametrised by $3(3^2C^2) = 27C^2$ weights; at the same time, a single 7×7 conv. layer would require $7^2C^2 = 49C^2$ parameters, i.e. 81% more. This can be seen as imposing a regularisation on the 7×7 convolution filters,

forcing them to have a decomposition through the 3×3 filters (with non-linearity injected in between).

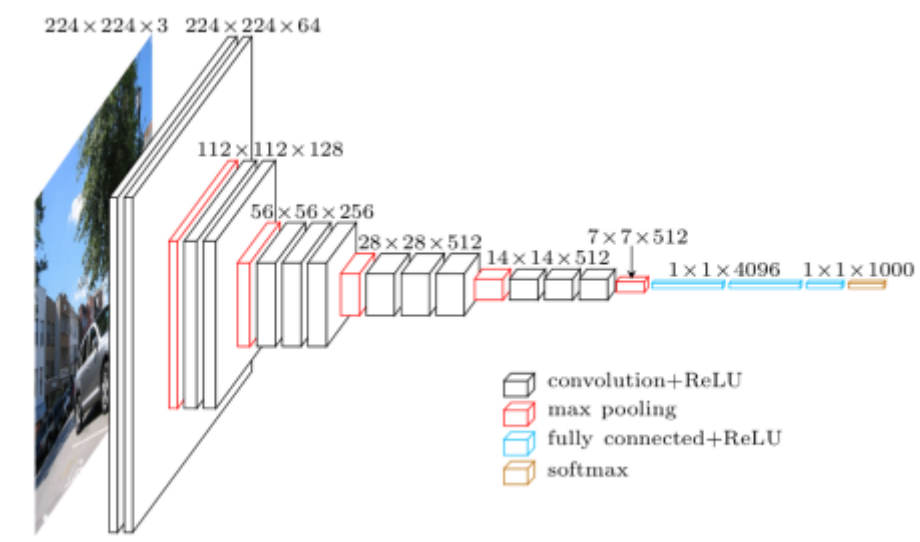

Fig. 6. A visualization of the VGG architecture.

Multi-Label Classification

Multilabel classification is a classification problem where multiple target labels can be assigned to each observation instead of only one in multiclass classification. It can be regarded as a special case of multivariate classification or multi-target prediction problems, for which the scale of each response variable can be of any kind, for example nominal, ordinal or interval.

Two different approaches exist for multilabel classification. On the one hand, there are algorithm adaptation methods that try to adapt multiclass algorithms, so they can be applied directly to the problem. On the other hand, there are problem transformation methods, which try to transform the multilabel classification into binary or multiclass classification problems.

In this project we use the Binary Relevance method. The binary relevance method (BR) is the simplest problem transformation method. BR learns a binary classifier for each label. Each classifier $C_1, \ldots, C_m$ is responsible for predicting the relevance of their corresponding label by a 0/1 prediction:

$$C_k: X \rightarrow \{0, 1\}, \quad k = 1, \ldots, m$$

These binary predictions are then combined to a multilabel target. An unlabelled observation $x^{(l)}$ is assigned the prediction $(C_1(x^{(l)}), C_2(x^{(l)}), \ldots, C_m(x^{(l)}))^T$. Hence, labels are predicted independently of each other and label dependencies are not considered. BR has linear computational complexity with respect to the number of labels and can easily be parallelized.

Part - 2: Development

As the title of our project suggests, we tried to solve a multi-label classification process using a variation of the VGGNet architecture.

The Dataset

The dataset we used for this task is the fashion dataset we collected over the internet by crawling google image search results and removing irrelevant results. It consists of 2,167 images divided into six categories:

- Black jeans (344 images)
- Blue dress (386 images)
- Blue jeans (356 images)
- Blue shirt (369 images)
- Red dress (380 images)
- Red shirt (332 images)

This means we have six possible labels – red, black, blue, shirt, dress and jeans for any given image. The images were stored in appropriately named subdirectories – all blue shirt images were store in a "blue_shirt" folder and so on. This makes it easier to assign class labels to an example during training. The goal of our model is to correctly predict both the colour and clothing type. Few examples from our dataset are:

Fig. 7. Examples of images used in our dataset.

The Neural Network Architecture

The architecture of our neural network is heavily inspired by that of VGGNet. We use the same concept of small 3x3 receptive fields with depth increasing after each layer, but the only difference is that of the depth of the network. The VGGNet that performed excellently at the ILSVRC-2014 challenge had 16 and 19 hidden layers each (VGG-16 and VGG-19). Training such a deep network would obviously take a lot of time. So, our version of the VGGNet, SmallerVGGNet was scaled down to be trained in a reasonable amount of time in the architecture available to us. The code for the neural network architecture can be found in "smallervggnet.py". Table 1 shows the details of all the layers of our neural network.

#	Layer Type	Description
0	Input Image	Input image of dimensions 96x96x3
1a	Convolution 2D	32 filters of size 3x3, “same” padding applied on the input vector, ReLU activation function with a Dropout of 25% and Batch Normalization applied along the depth
1b	MaxPooling 2D	Max pooling with a pool size of 3x3
2a	Convolution 2D	64 filters of size 3x3, “same” padding applied on the input vector, ReLU activation function and Batch Normalization applied along the depth
2b	Convolution 2D	64 filters of size 3x3, “same” padding applied on the input vector, ReLU activation function with a Dropout of 25% and Batch Normalization applied along the depth
2c	MaxPooling 2D	Max pooling with a pool size of 2x2
3a	Convolution 2D	128 filters of size 3x3, “same” padding applied on the input vector, ReLU activation function and Batch Normalization applied along the depth
3b	Convolution 2D	128 filters of size 3x3, “same” padding applied on the input vector, ReLU activation function with a Dropout of 25% and Batch Normalization applied along the depth
3c	MaxPooling 2D	Max pooling with a pool size of 2x2
4a	Fully Connected	A Dense fully connected layer with the flattened output vector of the previous layer fully connected with 1024 neurons at this layer. The activation function used is ReLU. A Dropout of 50% and Batch Normalization was also applied.
4b	Output Layer	The final layer of the architecture uses the sigmoid classifier and the number of neurons in this layer is the number of classes

Table 1. Architecture of our “SmallerVGGNet”

Training the Network

The code for training the network is available in “train_network.py”. We used the Keras library over TensorFlow to train our CNN and the matplotlib library to create the accuracy and loss graphs.

1) The first step is to initialize matplotlib in the backend so that it can save the values for the loss and accuracy graphs in the background and parse the arguments provided through the command line.

2) After importing all the required libraries and modules, we need to initialize our hyper – parameters: Learning rate, Epochs and Batch Size.

   Learning rate determines how fast we would want our error function to converge to a minimum. A lower than optimal learning rate would result in a sluggish pace of training, whereas a higher than optimal rate could miss the solution altogether. There is no way to know the optimal learning rate apriori. We used a learning rate of e^{-3}. The number of epochs determine how many times the entire dataset would be fed into the network. Higher the number of epochs, greater is the probability for the

network to learn intricate details about the dataset. However, one must be wary of overfitting the network. Overfitting causes the network to perform poorly on test images. Again, there is no method to decide the optimal number of epochs. We chose 150 epochs. We can't pass our entire dataset through the network in one pass; we must do it in batches. This is where Batch Size comes in. Batch Size is always initialized in powers of 2 i.e. 32, 64, 128 etc. Since we have a small dataset, we decided to use a batch size of 32; anything greater would result in a poor performance.

3) After initializing our hyper – parameters, we must load our training dataset. We load all our image paths into a dictionary and shuffle them.

   By shuffling the images and training on only a subset(batch) of them during a given iteration, the loss function changes with *every* iteration, and it is quite possible that no two iterations over the entire sequence of training iterations and epochs will be performed on the exact same loss function. This is important for us to "bounce" out of a local minimum.

4) After loading the dataset, we need to pre-process the images and assign class labels to them.

   We resize the images to 96x96x3, convert them into an array and store it in a list "data". The subdirectory in which an image exists dictates its label. We use the binary relevance method for multi-label classification. This means that if our classes are in the order ['red', 'blue', 'black', 'jeans', 'dress', 'shirt'], all red dresses would have a class label of [1 0 0 0 1 0] assigned to them and so on. The MultiLabelBinarizer class from sklearn's preprocessing module was used for this purpose.

5) The training data is then partitioned into training and validation sets with 80% for training and 20% for validation. Since we have a small dataset, we augment the number of images we have using "ImageDataGenerator" to boost the performance over our validation set.

6) Now, we build our model. Our model is a sequential one, we import our model from "smallervggnet.py" and use the sigmoid activation function in the final layer of our neural network. We use sigmoid instead of softmax because we are using the binary relevance method of multi-label classification in which we treat each class label as an independent binary classification.

7) After building our model, we can compile it. We compile the model using binary cross-entropy rather than categorical cross-entropy. This may seem counterintuitive for multi-label classification; however, the goal is to treat each output label as an independent Bernoulli distribution and we want to penalize each output node independently. The optimization function we used is the Adam optimization function.

8) We save the model and the multi-label binarizer to disk as we need them during testing. Finally, we can plot our Training Loss and Training Accuracy curves.

With set of hyperparameters reported we were able to achieve a training accuracy of **98.25%** and a validation accuracy of **99.11%**. Fig. 8 shows the training loss and accuracy graphs.

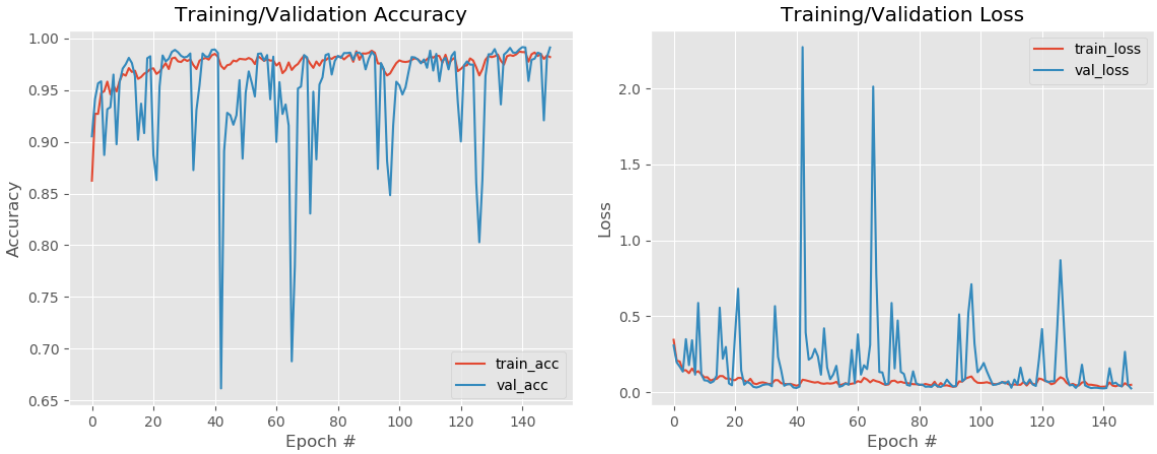

Fig. 8. Training/Validation Accuracy and Loss Graphs

Testing the Model

The code for using the model to classify test labels is present in "test_network.py". First, we parse the arguments and pre-process the image. Then we load the model and classes into memory and use the model to classify the image. The two labels with the highest confidence are written onto the image and the output is displayed. Fig. 9 shows the outputs of a few test images.

Fig. 8 Examples of model predictions on the test images.

More test image predictions can be found in the "examples" subdirectory.

Conclusions

In this project, we used the binary relevance method of solving multi-label classification problem by applying a small-scale VGGNet to our fashion dataset. We were able to achieve very good accuracy on our validation set. However, there are a few drawbacks to our approach. Binary relevance ignores label correlations and may suffer the class-imbalance problem. Because of this, we need training data for each combination of categories we would like to predict.

Just like a neural network cannot predict classes it was never trained on; our neural network cannot predict multiple class labels for combinations it has never seen. The reason for this behaviour is due to activations of neurons inside the network.

If our network is trained on examples of both (1) black pants and (2) red shirts and now we want to predict "red pants" (where there are no *"red pants"* images in your dataset), the neurons responsible for detecting *"red"* and *"pants"* will fire, but since the network has never seen this combination of data/activations before once they reach the fully-connected layers, our output predictions will very likely be incorrect (i.e., we may

encounter *"red"* or *"pants"* but very unlikely both). This is the drawback of binary relevance method.

The parameters used during training have scope for optimization. We just used the values we felt best from limited experimentation. These parameters include number of epochs, batch size, learning rate, training/testing split ratio and most importantly the model architecture. Training on a larger set of training images would significantly improve the performance of our CNN.

References

- ImageNet Classification with Deep Convolutional Neural Networks - Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
  https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf
- Visualizing and Understanding Convolutional Networks - Matthew D. Zeiler, Rob Fergus
  https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf
- Very Deep Convolutional Networks for Large-Scale Image Recognition – Karen Simonyan and Andrew Zisserman
  https://arxiv.org/pdf/1409.1556.pdf
- Multi-Label Classification: An Overview – Grigorios Tsoumakas, Ioannis Katakis
  http://lpis.csd.auth.gr/publications/tsoumakas-ijdwm.pdf
- Multilabel Classification with R Package mlr - Philipp Probst, Quay Au, Giuseppe Casalicchio, Clemens Stachl and Bernd Bisch
  https://journal.r-project.org/archive/2017/RJ-2017-012/RJ-2017-012.pdf
- Binary Relevance for Multi-Label Learning: An Overview – Min-Ling Zhang, Yu-Kun Li, Xu-Ying Liu, Xin Geng
  http://cse.seu.edu.cn/people/zhangml/files/FCS%2717.pdf