# Convolutional Neural Networks

**Why not to use MLP NNs for computer vision problems:** Using MLP fully-connected feed-forward NNs with high resolution images to solve a computer vision problem (e.g. image recognition) is very prohibitive. For example, with 1000x1000x3 images the input dimension is 3M features and e.g. the kernel matrix of the first hidden layer will be Nx3M weights, where N the number of units in the hidden layer. Having such big vectors and/or matrixes **might be impossible to store** them in memory. Also, it is **computationally expensive** to train networks with large no. of trainable weights. Lastly, MLP NNs **tend to overfit** when we use a small no. of high-dimensional examples. The solution to this problem is using Convolutional Neural Networks (CNN). CNNs work with sparsity of connections and parameters sharing, thus they have fewer trainable parameters in comparison to MLP NNs.

**Edge detection having knowledge for dark-to-light or light-to-dark transition:** When detecting edges in an image using convolution the feature map has either negative or positive values in the detected edges area depending on whether the transition was from dark to light or light to dark pixels. In case we do not care about this type of information and we want to detect only the transition event we can just **take the absolute value of the feature map**.

**Translation invariance:** CNNs can support this type of geometrical transformation by default cause they search for features of interest in the whole image using the moving convolutional window.

**Learning the edge detectors:** While training CNNs we actually learn also the convolution kernels (we treat them as learnable parameters) instead of specifying them manually. That way we can discover filters that not only can detect horizontal/vertical edges (using e.g. Prewitt/Sobel/Scharr filters) but we can learn also filters that detect edges of e.g. 45$^o$, 70$^o$, 78$^o$ degrees. In general, on training a CNN we actually train feature extractors. The optimization algorithm learns whatever edge detectors are needed to minimize the loss.

**Output size after convolution:** The output size of a convolution operation is governed by the formula (p=padding, f=filter size, n=input size, s=stride). In a deep architecture we gradually shrink the width, height and increase the depth of the tensors (# of features extracted).

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \quad \times \quad \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

**How convolution w/ SAME padding can help: 1)** The **VALID convolution** (when p=0) operation uses no padding and shrinks always the input. So, in a deep NN with lots of convolutional layers the input shrinks too much unnecessary. In many cases we want to be able to use convolution but keep the dimension of input to the output and perform sub-sampling with other techniques (e.g. pooling) whenever we want to. One way to **avoid shrinking the input** and keep its size in output is using stride=1 with **SAME padding**, which adds as many as padding borders of zeros (zero-padding) are needed to the input before convolution so that the output has the same size (it depends only with the filter size: for a 3x3 convolution filter we use a single border p=1). **2)** When we use VALID convolution the **pixels in the border are not taken into account as much as the inner pixels**, thus the convolution throws away a lot of the information from the border of the image. So, by using SAME padding we take into account the border pixels more.

**Pooling usage:** It reduces the size of internal representation to speed the computations, memory requirements and makes features extracted more robust. Pooling layers most of the times use only s and f hyperparameters. The p hyperparameter is rarely used.

**Max pooling:** Subsampling the input tensor (features extracted from previous layers) and keep from each window (using a max filter) the most highly activated feature value for the output tensor.

**Average pooling:** Same as map pooling but instead of taking the max value it takes the average of window values. Also, can be used to collapse e.g. a 7x7x1000 to 1x1x1000 by taking the average value from each 7x7 feature map using f=7 and s=1.

**Cost is different than loss:** Usually the loss is used for calculating the error in a single example where cost a mean calculated from all the losses.

**1x1 convolutions (also known as Network in Network)**: It can help to shrink, increase or keep the same depth input dimension in the output without affecting the width and height of the tensors. It seems to be the opposite of pooling which affects the width and height but keeps the same depth in the output. It is used in Inception (GoogleNet) NN but it can be used in general.

**Visualizing what deep CNN are learning:** In order to visualize what a unit in a CNN has learned to detect is to find the image patches that make it to highly activate. Usually, we feed-forward the training dataset and search for the first N image patches for which the unit highly activates. By visualizing the N image patches we can have a clue of what the unit has learned. In CNN units from earlier layers learn to detect low-level features (e.g. edges, textures, colors) like the V1 primary visual cortex in human brain and are connected with a small image patch. However, units in deeper layers learn to detect high-level features (e.g. eyes, mouth, nose, face, tree, objects) and are connected with a large image patch. By visualizing the image patches that make units from different layers to highly activate we can have understand better what a CNN actually learns.

**CNN Architectures**

If a CNN architecture is applied on a computer vision task and works well it is a good practice to try to use the same architecture in a similar and different computer vision task. Some architectures: LeNet-5, AlexNet, VGG, ResNet, Inception (GoogleNet).

**LeNet-5**

Uses sigmoid/tanh activation functions instead of ReLU
Uses no padding (valid convolutions) and avg pooling instead of max pooling
Uses grayscale images instead of RGB
Conv+pool repetitive blocks with FC layers + softmax
Sigmoid non-linearity after pooling
10 classes
60K parameters

**AlexNet**

Bigger resolution of RGB images
Bigger conv filters and stride
Same convolutions
Relu is used
10-100M parameters
More FC layers + softmax
1000 classes

**VGG**

Use less hyperparameters (all conv filters are 3x3 /w s=1 and map pooling layers are 2x2 /w s=2)
Sequential convolutions
Blocks: conv-conv-pool
Fully connected layers + softmax
1000 classes
vgg-16/19 has 16/19 layers with trainable params (pooling layers are excluded)
138M parameters (downside large no. of parameters)
Uniform architecture
Doubling filters in blocks 64x64, 128x128, 256x256, 512x512

**ResNet**

**Difficulty in training deep NNs:** Very deep networks (either MLP or plain CNN) face the problem of **vanishing and/or exploding gradients**, thus they are very difficult to train. In theory when we increase the #layers the training error should be decreased since the model can fit the training data better. However, in practice after some value of #layers the training error either starts to increase or stops decreasing (due to vanishing/exploding gradients problems which starts to appear in deep networks). So, it is difficult to train very deep NNs.

A ResNet NN, using **residual blocks**, can be very deep (e.g. 100 or 1000 layers) and still be able to train without problems. **A residual block** contains some main path of consecutive layers (FC or CONV) and one **skip/shortcut connection** alternative path. The skip connection adds the input of the first layer to the linear input (before non-linearity) of the last layer inside the residual block. The idea of skip connection in residual block can be used to make the NN deeper and deeper without hurting the performance of the model. In backpropagation the gradient can be propagated back to earlier layers using the shortcut connection without the need to pass it back sequentially from all layers - which will make the gradients to either grow or shrink.

**Two types of residual blocks exist:** Identity block and Convolutional block.

**What is the intuition of ResNet?** When adding a residual block to an existing NN it either helps or keeps same performance. We can add residual blocks to make a NN deeper and be assured that at least we are not going to hurt its performance. A residual block it is very easy to learn to behave like the identity function (outputting its input). If a residual block cannot learn something useful that improves the NN's performance it will be very easily learn the identity function which at least doesn't hurt it.

(for the residual block to learn to behave like the identity function we need to use L2 regularization and ReLU activation functions)

**Inception (GoogleNet) – we need to go deeper**

When we are designing a layer in a CNN we need to decide if we should use pooling or convolution and more specifically what kind of convolution filter size, e.g. 3x3, 5x5, 7x7? The Inception NN uses **inception modules**. An inception module computes them all (pooling and convolutions) using appropriately "**same padding**" so that all the results can be **concatenated** (channel concat) to a single output volume. That way, the NN while training can decide what weights to learn for each filter and what is best to use to perform

well. However, the inception module is **computationally expensive (due to all the calculations)** and to reduce the no. of multiplications the inception module uses a **trick with 1x1 convolutions**. Before each convolution layer (3x3, 5x5, 7x7, etc) and after each pooling layer it uses 1x1 convolution to shrink the depth of the representation (this works as a dimensionality reduction operation). The decreased representation before the convolutions usually is called the **"bottleneck"** and doesn't hurt the performance of the NN.

**Data augmentation:** Use Mirroring, Cropping, Rotation, Translation, Shearing, Color Shifting (PCA color augmentation = changes mostly the color channels that dominate the image) and other to generate new training examples. Usually, we use a generator that loads from the disk some training examples, augment it (using CPU) and pass to the gradient descent algorithm a batch of training examples for training (using CPU/GPU). In case the dataset is small it can be loaded once from disk and used by the generator for generating the batches. All these data augmentation techniques have hyperparameters we might need to tune.

**Image classification:** Find the label of a single object in an image (the object is usually in the middle of the image and has some background).

**Image classification with localization:** Image classification along with estimating the bounding box of the object in the image.

**Object detection:** Recognize multiple objects with same or different label in an image and locate their bounding boxes.

**Classification with Localization (for 1 object)**

The ground-truth vector of a training example:

**$P_c$ :** 0 or 1 depending if there is an object or not in the image
**$b_x$, $b_y$, $b_w$, $b_h$ :** x/y (center point) and width/height of the object's bounding box
**$P_1$, $P_2$, $P_3$ :** one-hot encoding of the object's label

The output vector of the NN is consisted of:

**$P_c$ :** probability that an object is detected
**$b_x$, $b_y$, $b_w$, $b_h$ :** x/y (center point) and width/height of the object's bounding box
**$P_1$, $P_2$, $P_3$ :** softmax probabilities for the labels

The **$b_x$, $b_y$** are normalized in range [0, 1].

When calculating the loss of a single training example and the ground-truth $P_c$ is 0 then we just use a logistic regression loss for the binary output and we do not care about the rest of the output values. However, when the ground-truth $P_c$ is 1 then we use also extra losses for the rest of the values:

**$b_x$, $b_y$, $b_w$, $b_h$ :** sum of squared errors loss (used in regression problems)
**$P_1$, $P_2$, $P_3$ :** cross-entropy loss (used in classification problems)

Instead of having $P_1$, $P_2$, $P_3$ we could have a single number indicating the class, e.g. c=24.

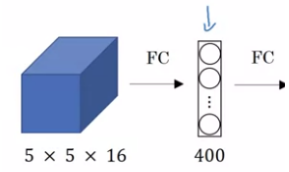
**Classification with landmark detection:** Train a NN using a labeled training set to not only recognize the label of the image's object but also detect any important **landmark points** of it. The

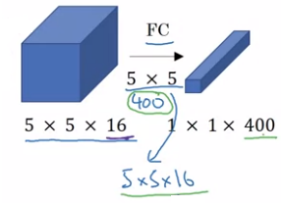
landmark points can be used as features for further processing in a ML pipeline. For example, we can train a binary face classifier to recognize if an image has a face or not and also detect its face landmark points (left eye, right eye, nose, etc) for detecting the emotion of the face or add special face effects to it (e.g. snapchat filters). Another application could be to extract body landmark points from a human body in order to detect its pose.

**Object Detection using Sliding Window Detection Algorithm:** We train an image classifier using a training set of **closely cropped images** containing only the object (centered with some background). After that, for an input image we use a moving/sliding window with a stride and for each image region we ask the classifier to recognize it (e.g. binary or multi-label classification recognition). We repeat this process using windows of different sizes and each image region is resized so that the classifier can accept it as input. **This method is slow/many places to look when the stride is small and fast/few regions to look when it is large.** If we want to use a small stride to do a better search we should use a simple classifier (e.g. logistic/softmax regression using hand-engineering features) because the model will be used in many image regions. In other words, if we want to use a deep CNN it is not recommended with a small stride cause it will be computationally expensive. **How to solve this problem trade-off?** The sliding window detection algorithm can be implemented convolutionally.

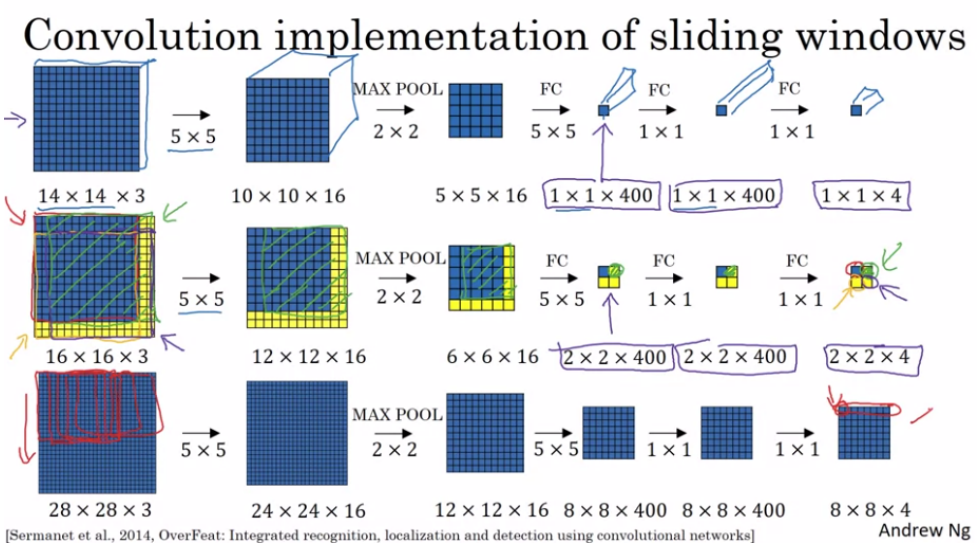
**Convolutional implementation of sliding window detection algorithm**

We first need to transform the trained CNN that uses FC layers to a CNN so that uses only convolutional layers. A FC layer can be transformed to a convolutional layer easily. For example:




After that, we change the input of the CNN to use the whole input image. This new CNN implements the sliding window detection algorithm convolutionally and instead of using the previous CNN multiple times in different regions of the input image we can use the new one once in the whole image. From the output volume, each 1x1xC (where C no. of labels) is a probabilities vector for a single window region of the input image. Using a single pass of input image we can calculate the prediction for each label for each rectangle of the image (using a specific filter size). If we want to use multiple window sizes we need to train and use multiple networks with different filter sizes.




**Not accurate bounding boxes:** Image classification w/ localization is able to recognize an object in an image and accurately estimate its bounding box. However, the algorithm can be used only for single object detection in an image. The sliding window detection algorithm can be used to detect multiple objects in a single image but has limitations on the accuracy of the bounding box prediction (bounding boxes with fixed aspect ratio). Is there an algorithm that accurately and fastly predict the label and bounding box of multiple objects in an image? Yes, it is YOLO (You Only Look Once). The YOLO algorithm looks only once the input image.

**YOLO algorithm**

Fast (real-time) object detection **with accurate bounding boxes (of any aspect ratio)** prediction. In YOLO an image is splitted in multiple parts using a grid and the algorithm learns to classify and localize the bounding box of a possible object in each of the KxK grid cells. The ground-truth y of a training example in the YOLO algorithm is a 3D KxKx(5+N) volume where for each grid cell we have the following information: $P_c$, $b_x$, $b_y$, $b_w$, $b_h$, $P_1$, $P_2$, $P_3$, …, $P_N$. If an object is spread in multiple grid cells only the grid cell that contains the center point of the object has $P_c = 1$. The YOLO model (deep CNN) is trained to map input images of WxHx3 resolution to KxKx(5+N) target output. For creating the ground-truth Y of training images we only need to know the bounding box and the label of each object. Using only this information we can create the Y ground-truth matrix MxKxKx(5+N) where M is the number of training examples, KxK the number of grids cells and N the number of labels. The values of $b_x$, $b_y$, $b_w$, $b_h$ in the YOLO algorithm can be calculated in various ways and one valid convention is to be relative to the grid cell containing the object.

**Accuracy metrics of bounding boxes:** The accuracy metric used for bounding boxes is IoU (**intersection over union**). IoU is a way of measuring how similar are two rectangles. A correct bounding box can be assumed when IoU >= 0.5. However, if we want to use a higher value <= 1 we can do it (e.g. 0.6 or 0.7 values).

**Non-max suppression:** The object detection algorithms (YOLO, sliding window detection algorithm) usually detect multiple bounding boxes for the same object in an image. As an example, in YOLO only the grid cells that contain the center point of an object are assigned to that object (in the ground-truth). However, when the algorithm is used **it predicts multiple bounding boxes for the same object in the image**. One way to resolve this problem is to use non-max suppression. This method runs per label and first discards all the detections with $P_c < 0.6$. Next, until all the bounding boxes related to a label are processed

(accepted or discarded) repeatedly finds the bounding box with the maximum probability, accepts it as an actual prediction, and discards the rest bounding boxes that overlap with it with IoU >= 0.5. This has to be run for each label's detected bounding boxes.

**How to encode in a single grid cell multiple overlapping objects?** With the KxKx(5+N) output the YOLO algorithm is unable to detect multiple objects that overlap and their center points appear in the same grid cell. For each grid cell we are able to detect and encode information only for one object. Center points of multiple objects can appear in the same grid cell mostly when the grid cells are large (small K value). However, when we are using large K number we have small grid cells and it is more unlikely for multiple objects to appear in the same grid cell. However, some times that might happen and using **anchor boxes** we can encode multiple objects in the same grid cell. Each object is encoded in a different location in the grid cell output depending on the anchor box that matches most the object's shape, using IoU similarity metric. In that case the output is KxKxLx(5+N) where L is the new dimension for the various anchor boxes. **How do we select the anchor boxes?** The anchor boxes are selected manually and carefully to represent the variety of shapes of objects that we want to detect. Usually, when the no. of different object shapes is very large, a clustering method (K-means algorithm) is used to group the various object shapes. From each group we can select some limited no. of representative object shapes to use as anchor boxes.

YOLO I/O: IMAGE (m, W, H, 3) -> DEEP CNN -> ENCODING (m, K, K, NL, 5+N)

**Limitations:** The YOLO algorithm can't handle cases where the no. of objects assigned with a grid cell is greater than L or more than 2 objects share the same shape and match with the same anchor box.

**R(egion)-CNN object detection**

**Region Of Interest (ROI) proposals:** Using a sliding window detection algorithm is very expensive computationally cause we blindly try to search the whole input one region at a time. The convolutional implementation of the sliding window algorithm is faster but still tries to search the whole input. The idea of **ROI proposals** is to extract only some interest regions and search only these. ROI proposal algorithms usually perform image segmentation on input and try to find region blobs of possible objects ignoring other parts of the image that do not seem to be objects. The proposed regions are used in a second phase by a convolutional sliding window algorithm for detection. Examples: a) **R-CNN**: Propose regions by segmenting the input image using a clustering algorithm. Classify (label + bounding box) each region using sliding window detection algorithm. Still slow algorithm. b) **Fast R-CNN**: Propose regions same as R-CNN. Use convolution sliding window to classify each proposed region at once. c) **Faster R-CNN**: Use a CNN instead of a clustering algorithm to generate the ROI proposals. In most cases even the Faster R-CNN is still a bit slower than the YOLO algorithm.

**Some history of object detection:** Models that perform image classification (recognize a single object in an image) or image classification w/ localization (recognize a single object in an image and its bounding box) are building blocks and can be used in larger algorithms that try to solve the object detection problem for recognizing and locating multiple objects in an image. For example, we have seen the sliding window detection algorithm that uses an image classifier to detect in an image the regions that contain an object of interest. However, we have seen that this algorithm is not efficient so we continued with a convolutional implementation of the sliding window detection algorithm that finds all the objects in an image with a single CNN pass. Instead of blindly search the whole input image we can use ROI proposals to limit the search and gain in computational time.

The family of R-CNN models are the most well-known using ROI proposals. However, all these implementations, cannot predict accurate bounding boxes for the object in the image and need to implement two separate phases (1: generate ROI proposals, 2: classify each proposed region). A solution to this problem is to use the YOLO algorithm which is a fast end-to-end algorithm capable of accurately predict bounding boxes of any aspect ratio.

How to build a face recognition system using only a database of single face images per person?

Many companies have such projects for security access control and build their dataset with single face images per person. Training a CNN with softmax output is not a good solution since it **will not be able to fit well because of a small no. of training examples**. Also, we would **need to re-train the CNN whenever we need to support a new person**. We only have a dataset of single training examples (face images) per person ID (label) and to implement the system we need to solve the problem of **One-shot Learning** for **Face Verification** and **Face Recognition**.

**One-shot Learning:** Training a model using only a single training example per class.

**Face Verification:** We ask the question "Is this the claimed person?" Input the face image of a person and an ID (e.g. face image from the dataset) and output the degree of difference between them as a scalar value. By thresholding this value we can transform the problem to binary classification.

**Face Recognition:** We ask the question "Who is this person?" Input the face image of a person and output the identity label or "unknown" in case the person in the person is unknown. This is a multi-label classification problem. Usually, the face recognition system uses the face verification system internally.

**How the face verification system works?** The face verification model is trained to operate as a **similarity function d(img1, img2)** which takes as input two face images and outputs the degree of difference as a scalar number. If the two input face images refer to the same person the output will be small and otherwise large. The face recognition system internally performs face verification of the new input face image with all the ID face images stored in the database and outputs the label of the one with the smallest difference. In case no face verification outputs a small difference (needs a threshold value) we should output the "unknown" label.

**How can we build and train the similarity function d(img1, img2)?**

1)

We usually build **d(img1, img2)** as a **Siamese CNN** that gets as input a face image x and outputs a **low-dimensional encoding vector** y=f(x). We train the CNN using backprop and update its parameters so that the loss(img1, img2) = l2norm(f(img1) – f(img2))^2 is small when the two images img1, img2 are the same person and large otherwise. In other words, the CNN is trained so that the encodings of img1, img2 are similar when the input images belong to the same label and different otherwise. **But, how we can perform such an optimization that needs both minimization and maximization?**

**Answer:** The actual loss function used to train Siamese CNNs is the **triplet loss function**: max (l2norm(f(A)-f(P))^2 – l2norm(f(A)-f(N))^2 + α, 0), which is designed for minimization with gradient descent. The loss needs a training triple pair of images (**A**nchor, **P**ositive, **N**egative). The related cost function is the sum of all triple losses. We actually want to solve l2norm(f(A)-f(P))^2 + α <= l2norm(f(A)-f(N))^2. The hyperparameter **α** is for

1) avoiding the trivial solutions [e.g. where f(x) is y for any x or f(P) = f(N)] and 2) supporting a **margin/gap** between l2norm(f(A)-f(P))^2 and l2norm(f(A)-f(N))^2. We usually set α large in order to find solutions where l2norm(f(A)-f(P))^2 is far different than l2norm(f(A)-f(N))^2.

**How to train well a Siamese CNN?** To train well a Siamese CNN we need to create triplets that are "hard" to learn. So, it is important our dataset to have triplets such that l2norm(f(A)-f(P))^2 is almost similar to l2norm(f(A)-f(N))^2 in order for gradient descent to try to separate them and make them far part at least at distance α.

**How such a model is used in One-short learning problems?** When we do not have such a dataset (with A and P images for the same person) to train the CNN we can use a pre-trained one. The pre-trained Siamese CNN can operate on new face images in a one-shot learning problem (single face images per person) to generate the degree of difference of input images.

2)

We usually build **d(img1, img2)** as a **Siamese CNN** that gets as input a face image x and outputs a low-dimensional encoding vector y=f(x). The CNN has a logistic regression unit in the last layer and the whole NN is trained using training pairs of images (img1 and img2) of same or different labels (y = {0, 1}) in order to learn to separate the pairs of images that belong to the same label from those that belong to a different label (as a binary classification problem). The logistic regression unit in the last layer is connected to a layer that calculates the encodings difference l1norm(f(img1) – f(img2)) of the input images.

Both the (1) and (2) algorithms work well and the encoding vectors for the face images from the dataset in the one-shot learning problem can be pre-computed and stored, for increasing efficiency in the face recognition system.

**Neural Style Transfer**

For style transfer to work we need to use transfer learning with a pre-trained CNN image classifier that has learned low-level as well as high-level features from various images. In style transfer we use the **content C as well as the style S** input images and try to **generate a new image G** which has some content from the content image C and some style from the style image S. The cost function $\mathbf{J(G) = a * J_{content}(C, G) + b * J_{style}(S, G)}$ is used for measuring how well the generated image G is. The $J_{content}$ measures how similar content the C, G images have and $J_{style}$ measures how similar style the S, G images have. By minimizing the cost function J(G) using gradient descent and by starting with a randomly initialized image tensor we can generate the image G. We actually calculate the gradients of J(G) w.r.t. to the image tensor and iteratively update its pixels using gradient descent. The image tensor gradually evolves to the generated image G.

**$J_{content}$(C, G)**

This cost component measures how similar content the images C and G have. Someone could suggest to use the MSE / PSNR metrics to measure the content difference. However, this metric operates on the lowest image features (pixels) and by optimizing such a metric we force the image G to be very similar to C. To avoid that we can measure the content difference of the two images using some extracted feature encodings from a pre-trained CNN image classifier. However, if we use the encodings from the earlier CNN layers we might slightly face the same problem of using the pixel features for measuring the content difference. Also, by using the encodings from the last

layers we lose the local spatial features since we compare two images in high-level features only (e.g. if there is a cat somewhere in the C image then there should be also in the generate image G somewhere). We usually, select a layer from the middle of the CNN that captures both some of the local spatial features and some of the high-level features. The difference of the encodings is calculated either with the Frobenius norm in case of matrixes or with the l2-norm in case of vectors.

**$J_{style}(S, G)$**

This cost component measures how similar style the images S and G have. The style of an image is reflected in the various CNN layers. **How do we measure the style of an image in a specific CNN layer?** It is measured as the **correlation matrix** (usually it is called 'style matrix') of the layer's activated feature maps. We measure how much correlated are the different activated feature maps. Two feature maps are highly correlated when their detected features appear together in the input image. The total style of an image is the collection of all the style matrixes (correlation matrixes) from all the CNN layers. The **style loss for a specific CNN layer** is measured as the Frobenius norm of the difference of the style matrixes of style image S and generated image G. The total style cost **$J_{style}(S, G)$** is the sum of all the style losses (from the difference CNN layers).

One important part of the style matrix is that the diagonal elements such as $G_{ii}$ also measures how active filter i is. For example, suppose filter i is detecting vertical textures in the image. Then $G_{ii}$ measures how common vertical textures are in the image as a whole: If $G_{ii}$ is large, this means that the image has a lot of vertical texture. By capturing the prevalence of different types of features ($G_{ii}$), as well as how much different features occur together ($G_{ij}$), the style matrix G measures the style of an image.