# Image Classification and CNN

Matias Giannoni

MIT

April 17, 2020

## Neural Networks

- Despite the *hype* around them, they are just nonlinear models - Hastie et al., (2009) have a nice introduction.
- We can think about them as a classification (Quant III logit/multilogit models) problem with a "hidden" layer. For $m$ hidden units and $k$ categories (and one hidden layer) we have[1]:

$$\begin{aligned} Z_m &= \sigma(\alpha_m' X) \\ T_k &= \beta' Z \\ f_k(X) &= g_k(T) \end{aligned}$$

- $\sigma$ is an activation function (typically logistic, or more commonly the rectified linear unit which seems to perform better on CNNs)

[1] Intercept (or bias) in X and Z, omitted for simplicity.

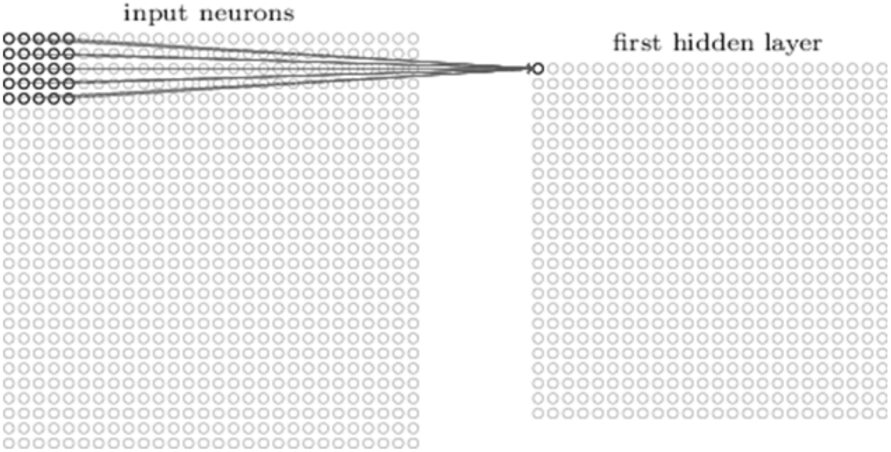
## Neural Networks

- In a regression problem, $g_k(T)$ is just the identity, while in a classification problem with $k$ classes is the "softmax" (the one we use in multinomial logit: $g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$)
- As usual, this is generally solved by minimizing the loss function (cross-entropy[2] for classification) through gradient descent (or some variant of it).

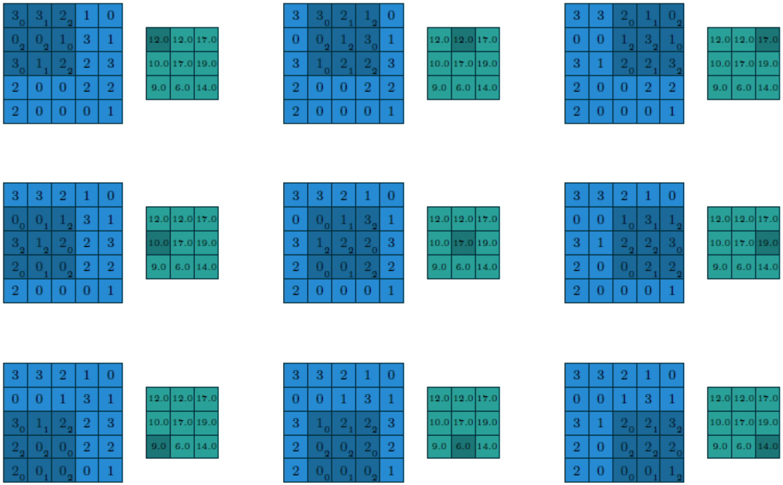
[2]That is, $R(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} log[f_k(x_i)]$, same as when we maximize the likelihood.

## Convolutional Neural Networks

- We want to classify images (or objects within images) but images have an important property: nearby pixels are more strongly correlated than more distant pixels (Bishop, 2006). We loose a lot of information if we just flatten the image into a vector.
- As in NN, in a CNN we have a hidden layer, but the weights instead have the form of a matrix of dimensions smaller than that of the image.
- A convolutional NN convolves (graphically, slides the matrix and takes the product) the matrix of weights over the original image (take the element-wise product and sum the results).
- We start with a matrix of a certain size and after the convolution we have another matrix (whose dimensions are a function of how many times the matrix of weights can "move" over the image).



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map


3 3 2 1 0
0 0 1 3 1
3 1 2 2 3
2 0 0 2 2
2 0 0 0 1
12.0 12.0 17.0
10.0 17.0 19.0
9.0 6.0 14.0

## Key words

- Kernel: matrix of weights (we define its dimensions).
- Filter: a filter is a concatenation of kernels (i.e., define third dimension). No particular reason to choose the number, more filters means you think there are more distinct features to capture in any given region.
- Stride: How many units at a time does the filter moves over the image? It can be just one, but if you increase it the hidden layer becomes smaller.
- Padding: Since the kernel is smaller than the image, the output layer will be smaller. But we might want to preserve the size (at least in the initial layers) for the kernel to cover more space. So padding adds zero valued pixels to the borders: a padding of two on a 32x32 image would transform it into a 36x36 image and the kernel can convolve over those borders.

## Key words

- Feature map/output image: the image (a layer) that is produced after the convolution.
- Pooling layer (or downsampling): It is a layer in which instead of doing a convolution by taking a product we just slide over a filter that takes one value by doing some operation (generally keeping the maximum, but also the average or the norm) out of those in the region to create the next layer.
- Dropout: For a given layer, we set to zero a random sample of activations in that layer.

## Stochastic Gradient Descent / Mini-Batches

- Usually (as in MLE) we would optimize the weights based on the gradient from the entire dataset.
- That's generally unfeasible with really big data (specially images and deep networks with many layers) due to memory limitations.
- So SGD passes observations through the network one by one (pass, update weights, move to the next and so on). It can be very slow but takes little memory.
- The standard in CNNs is to use a mini batch: not one (too unstable) but not all the images at once (classical gradient descent), guarantees a smooth descent but most memories won't handle it and it risks overfitting.

## Deep networks

- An image is a set of pixels $x \in \mathbb{R}^{m \times n \times c}$ that is, a 3D tensor defined by the height ($m$ of pixel rows), width ($n$ pixel columns) and channels (three channels $c$ in a digital coloured image) - (Pelt and Sethian, 2018).
- The problem of image classification is best understood as finding a function that, given an image $x$ produces another image $y$ with different dimensions ($n'$, $m'$ and $c'$), which in the case of a binary classification problem (where we expect an output in the form of a single probability) the dimensions after applying the function over $x$ become $n' = m' = c' = 1$.

- Each layer $i$ produces an image $z_i \in \mathbb{R}^{m_i \times n_i \times c_i}$. So it is generally the same as in a NN but with a convolution step (call $z_{i-1}$ the previous/input image, for the channel $j$):

$$z_i^j = \sigma(g_{ij}(z_{i-1}) + b_{ij})$$

- Where $g_{ij}(z_{i-1})$ is the result of the convolution of the input image with a filter $h$ and the pixel by pixel sum of its input channels:

$$g_{ij}(z_{i-1}) = \sum_{k=0}^{c_{i-1}} C_{h_{ijk}} z_{i-1}^k$$

- The convolution is just a linear operation (as in a standard NN) but with less parameters than a fully connected network on a flattened image (we are re-using the filter over the image).

- Since the convolution is linear, the activation function (generally ReLU) introduces non-linearity to the network.
- For an input of size $W$, a filter of size $K$, a padding $P$ and a stride $S$, the size of the output will be:

$$O = \frac{(W - K + 2P)}{S} + 1$$

- In a **deep** network we play around with this since we want to get at many subtle features. The first layers will get some basic features (lines, geometries), the middle ones some abstract ones, while layers closer to the output interpret the features in the context of the task.
- We can end up with millions of parameters by stacking more and more layers.
- The results from some of the filters from the richest models are really scary: GoogleNet examples

- But eventually we want to reduce all this to a fully connected layer and finally to some probability/number that solves our classification/regression problem.
- As in any machine learning problem, we also want to avoid overfitting.
- That is where pooling (helps both to reduce dimensions and overfitting) and dropout (helps reduce overfitting) come in.
- In any architecture, since in the end we just want a vector of probabilities (classification) or a regression output, the final steps will be to flatten the last layer (transform the last matrix/tensor to a long vector) and have a final fully connected (standard NN) layer.[3]

[3] We can "stop" before this if we wanted to "see" what the machine sees, which may or may not make sense to our brains but it is for example useful in adversarial attacks: `https://www.youtube.com/watch?v=i1sp4X57TL4`

## Keras

- Beauty of Keras: an API with almost no coding, just decide the layers and number of parameters and build the architecture in just a few lines
- It's an API for three deep learning libraries (Google's Tensorflow, Microsoft's CNTK, and Theano - Université de Montréal; the later two are deprecated)
- You could code the architecture directly in Tensorflow (i.e., Tensorflow example). It's just more work for something that it's super simple in Keras
- It's easy to use in Python but it also exists in R (where layers are stacked up with the `dplyr` pipes syntax)

## Image preprocessing

- As in any problem, more data is better. The “good” thing about images is that we can artificially “increase” our data.
- A dog from different perspectives is still a dog, so we want to zoom the image and move it around to increase the size of our training set.
- Keras makes this easy thanks to its image preprocessing class.

## CUDA

- It is still possible to fit CNNs without a GPU, but depending on the architecture and data size it will be painfully slow.
- If you have a GPU, it's good to take advantage of it because training will be very fast.
- CUDA allows for the use of powerful GPU's for parallel computing.

## Transfer Learning

- In general, we wouldn't start training a model from the scratch. There are many features (shapes, color patterns) that repeat in any real world image regardless of the classification problem.
- In other words, there are pre-trained models that can give us really good *priors* on which are the weights ("parameters") for many of our filters.
- Pre-trained models decrease the training time and in theory they can reduce the generalization error. Many good models are readily available in Keras. You can also save your own models and re-use them later.
- But for things that look too different from the data used to train those models, our models can sometimes perform worse than if we train from the scratch. It is important to figure out how much we can "unfreeze" (you can unfreeze the entire model but it means learning 20+ millions of parameters with far less data).

## Example in Keras

- A very simple problem: I scraped all the images with a certain hashtag referring to a protest (Yellow Vests) but I had to get rid of all the random stuff that people uploads (memes, cats, news, weird stuff)