This commutative diagram is all you need to know to train convnets (if you already know multilayer perceptron and convolution).

x is an image squished into a vector.
$sqr()$ reshapes such vector back into a square matrix.
$K * S$ stands for a convolution operation of a kernel K with a matrix S.
$vec()$ turns a matrix into a vector $vec(sqr(x)) = x$.
$M \cdot x$ is a dot product.

It can be somewhat tricky to figure out how to get M from K and vice versa. Took me a day to code it, but most of the time has been wasted on searching for a bug that ended up being caused by numpy's automatic datatype conversion.

$$
\begin{tikzcd}
x \arrow[r, "sqr(x)"] \arrow[d, "M \cdot x"'] & S \arrow[d, "K * S"] \\
y & Z \arrow[l, "vec(Z)"]
\end{tikzcd}
$$

Training

1. Compute y from x and some K by going to the right-down-left way.
2 Find an equivalent $M \cdot x$ operation.
3. Differentiate it as you would normally do in a multilayer perceptron.
4. Do gradient descent on M.
5. Get a new K from an updated M, go to step 1.

How to find M from K and vice versa?

It involves some linear algebra which I had to google how to solve. I wrote all of it without internet but then got stuck at solving a not super nice system of linear equations. Fortunately numpy has a lot of great features built in. A good intuition is that both dot products and convolutions involve summs of products of elements.
Honestly I don't know a good way to describe the algorithm to my past self such that it would be easier to understand. So the best advice I can give is to just derive it yourself. If you are good at linear algebra you'll solve it in a few minutes, if you suck at it as much as I do it might take a while.
A lazy reader that just wants to believe can check out the `kernel(x, y)` and the `matrix(x, y)` functions in the ipython notebook.
Not for all M there is a K. But turns out gradient descend on M keeps all the properties of M enough for it to still be equivalent to some K. I am a pure math dropout and so don't give a fuck about why anymore.