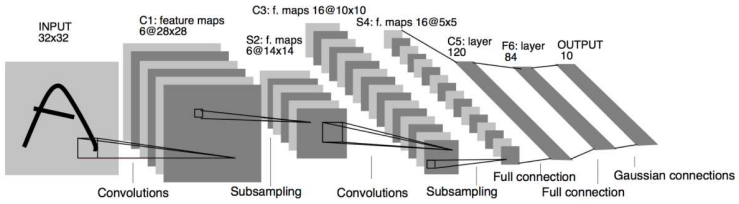
Deep Learning - LeNet

Bernhard Kainz

LeNet-5

Gradient-Based Learning Applied to Document Recognition

YANN LECUN, MEMBER, IEEE, LÉON BOTTOU, YOSHUA BENGIO, AND PATRICK HAFFNER



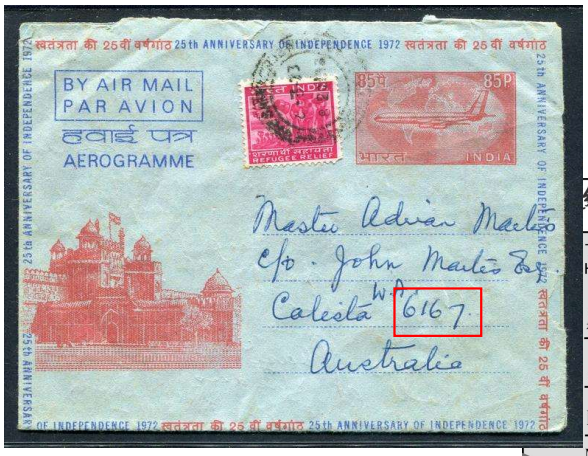
LeCun et al. 1998

leNet was a network proposed in the 90s so LeNet-5 is around 1995 by Yann LeCun and his team.
In the simplest version this was engineered for low resolution black and white object recognition.
The inputs were 32 by 32 bit image size images
Those images were then convolved to turn into six channels of 28 by 28 pixels which were then reduced through average pooling to 14 by 14

Remember the channel size is the same because pooling is applied to layers individually and not across layers.
This was then followed by another convolution, which reduced it to 10 by 10 images with 16 channels
The resolution gets halfed again by average pooling, so you get 16 5 by 5
This is in the end followed by 120 fully connected units, then by 84 fully connected units
In this case this was a Gaussian RBF network to map feature maps into 10 classes

At the time this was a serious engineering effort and it probably took about between three months and half a year to implement including all the tooling that was needed

Handwritten digit recognition



Why would people have care about it 95?

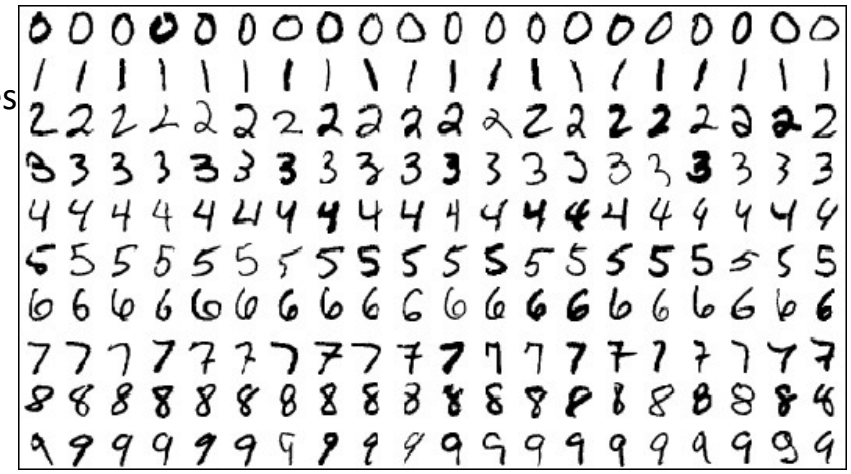
At&T at the time had a project about handwritten digit recognition.
Handwritten characters for postal codes on letters and also amounts on checks

Basically banks ant post offices wanted to automate the to this point manual transcription of handwritten documents into computer readable digits.

Obviously that's not quite so trivial, object recognition is required to find the right area to make a prediction

MNIST

- Cantered and scaled
- 50.000 training samples
- 10.000 test samples
- 28 x 28 images
- 10 classes



Deep Learning – Bernhard Kainz

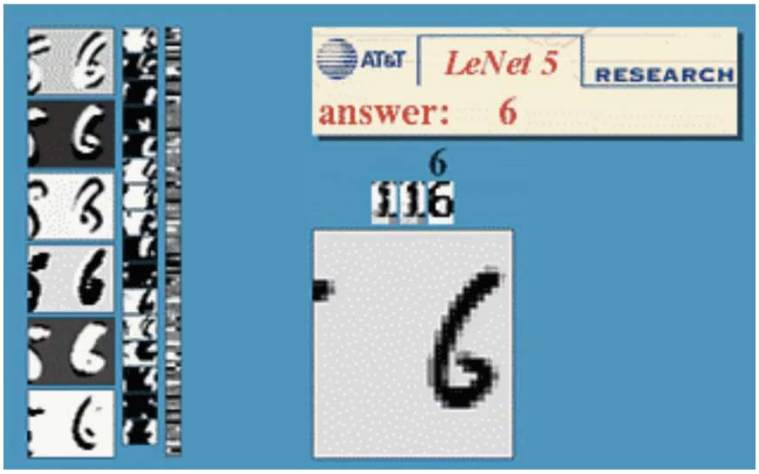
MNIST was a data set that was engineered specifically for the purpose of handwritten digit recognition. It consisted of centered and scaled images.

50,000 training data and 10,000 tests data at the resolution of 28 by 28 pixels.

There were 10 classes because, well, there are 10 digits and these digits were realistic digits as obtained by looking at letters and
actually segmenting them appropriately.

Demo from 1995

https://www.youtube.com/watch?v=yxuRnBEczUU



To see how this worked, let's have a look at the demo of LeNet. So here you can see digits scanning through the network.
In the network outputting its estimate of what it thinks that digit would be

Even for different shifts it still recognizes a five is a five

Furthermore you can see on the left hand side the activations of the various layers.
In the first layer after the convolutions you pretty much just get edge detector, vertical edges, horizonal edges, things that enhance contrast and so on.

on the next layer you can see how this now turns into higher level features but still sort of spatially related

then beyond that there are activations of the fully connected layers and you can see that this is now much more diverse

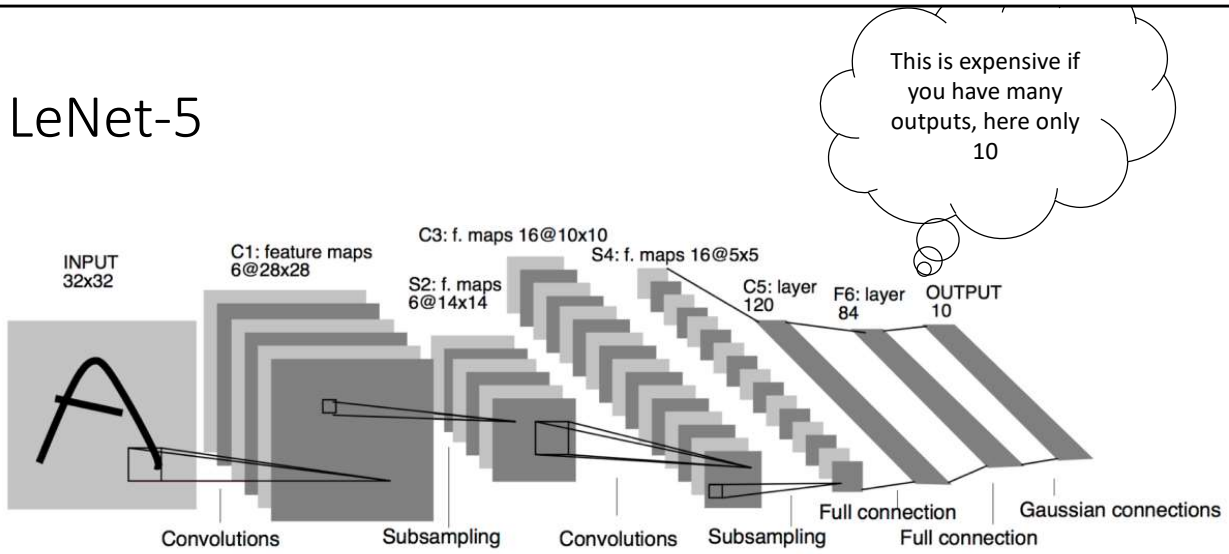
in the end this is converted into an estimate of a particular digit as a vector of probabilities.

so the paper that documents this from 1998 is a really landmark paper

I strongly recommend reading this. It is also part of the notes.

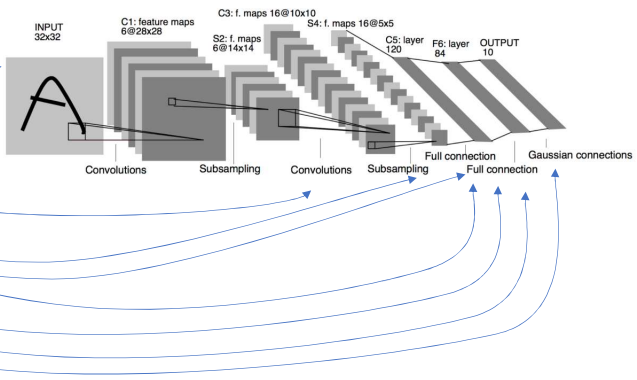
there's a lot of detail in there also on graph transducers which I don't think are fully appreciated even nowadays.

LeNet-5



fully connected layers.
These fully connected layers can be quite expensive if we have many outputs, so for ten classes it's not a big deal.
Once we will move to maybe a thousand classes for imagenet this will actually become the dominant factor in our network design and we'll have to find ways around it.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```



https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

– Bernhard Kainz

Let's have a look at how to actually implement that. We'll do this in pytorch and nowadays it is not too complicated to do this.
This fits nicly on a slide but In 1995 this would not have been so trivial to implement.

Here, we include torch.nn.functional to have direct access to network functions. We do not need class wrappers here, but there are several ways to define these functions in pytorch.
We have to define a network as child class of nn.Module to be acceptable for the optimiser and then we can simply define all we need.
The data is passed through the network in the overloaded forward function. Backward() is done automatically for you by pytorch for any differential functions.

In the forward pass I want to have a sequential composition of layers in that network and then I just add a convolution average pooling another

convolution another average pooling operation and then two dense layers in the end