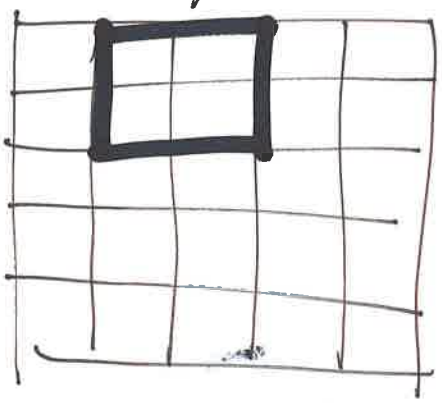
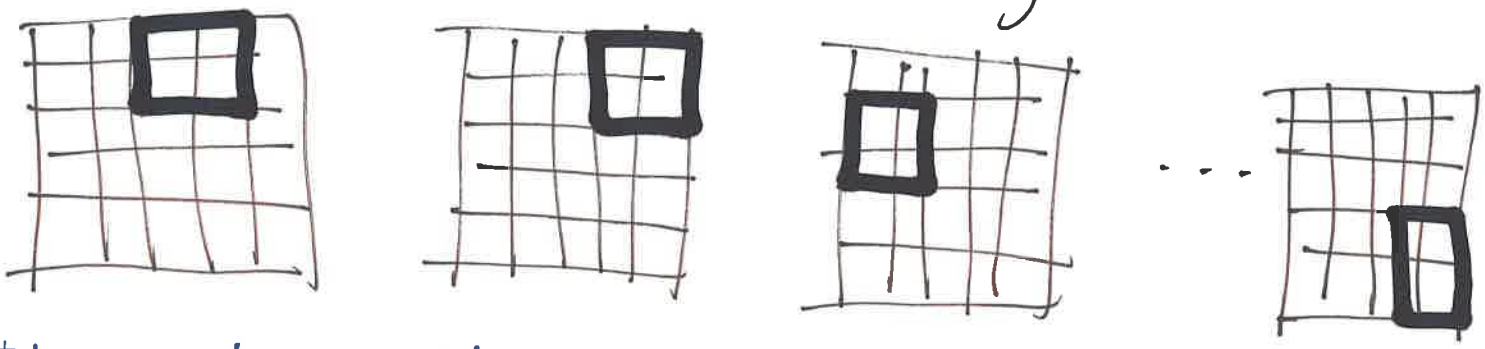
Then, we slide the filter one unit to the right and get the second output.



$$2 \cdot 1 + 0 \cdot (-1) + (-1) \cdot 0 + 2 \cdot 2 = 4.$$

The sequence of calculations can go like this.



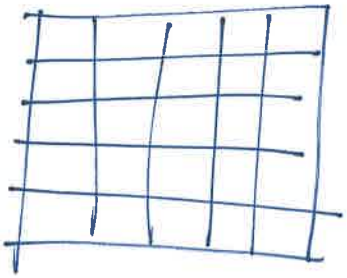
The end result is a 4 x 4 feature map consisting of all the 16 numbers we have calculated by sliding the filter:

-3	4	...	...
...	..		

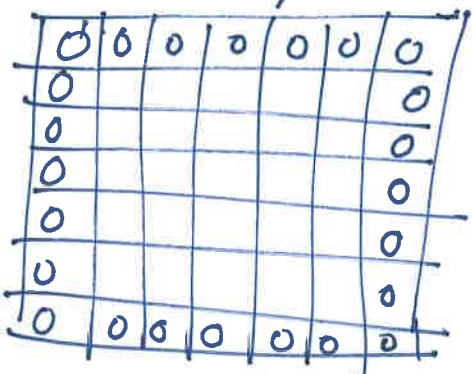
You can also imagine that we have 16 neurons that share the weights [1 −1 0 2]

Each neuron is only connected to a subset of the inputs called its local receptive field.

* We almost always have more than one filter to extract multiple features. In this case, one creates multiple feature maps in a convolutional layer. Each filter is allowed to have different weights.
* Several variations of the filtering idea exists, such as zero padding. Instead of



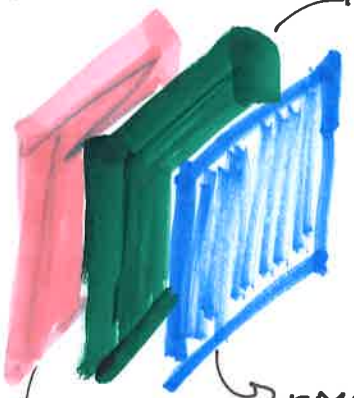
we can apply convolutions to the zero padded version.



* One can also define a "stride", i.e how many squares to "jump" when sliding the filter.
* The sliding manner in which local fields of neurons are calculated is analogous to signal convolution. This is also why the networks are called convolutional networks.
* Convolutions may be followed by any activation function like Relu, tanh, etc.

Handling colored inputs.

In this case, the image is decomposed to R, G, and B channels. Each channel is a separate image. We can stack the channels on top of each other to get a multichannel image that looks like as follows:



A filter to be applied to this "3D" input should now also be 3D. For example, if the image is 4×4, and the filter is 2×2; the first neuron local field would be:

$1 \cdot 1 + 1 \cdot -1 + 2 \cdot 1 + 0 \cdot 2$
$+ 1 \cdot 1 + (-1) \cdot 0 + 0 \cdot 0 + 1 \cdot 6$
$+ 4 \cdot 3 + 1 \cdot 2 + 2 \cdot 1 + 3 \cdot 0$
$=$ whatever.

3-channel input

1	1	2	1
2	0	-1	1
-1	1	0	-1
0	1	2	3

1	-1	1	-1
0	1	0	2
2	3	0	1
1	4	3	1

4	1	4	2
2	3	1	3
1	2	3	1
0	1	2	2

1	-1
1	2

1	0
0	6

3	2
1	0

3-channel filter.

Handling multi-channel inputs

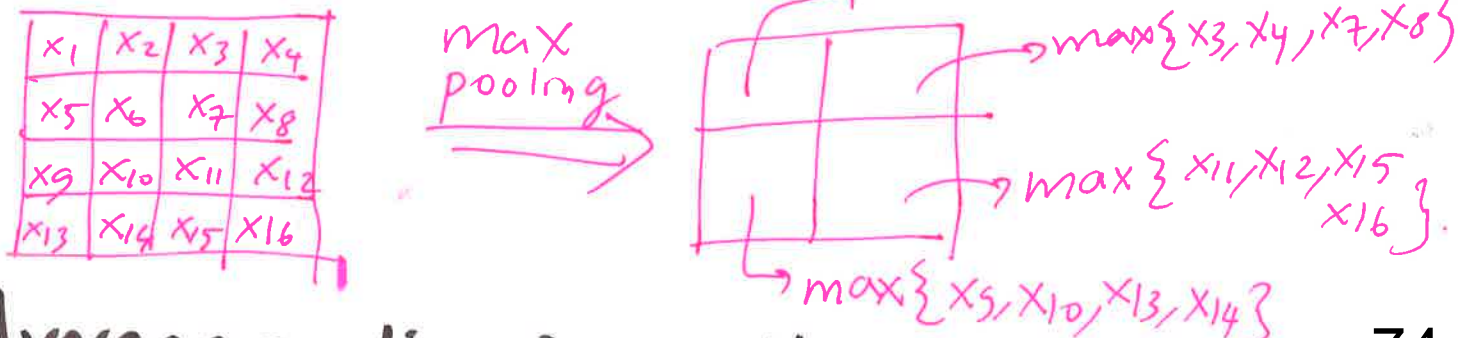
Are done in the same manner.

Suppose you applied a convolutional layer and got 10 feature maps. You can follow this by another convolutional layer. The filters of this new layer may act on all 10 channels of the previous layer (or a subset of them).

Pooling layers.

Convolutional layers are typically followed by pooling layers to reduce the dimensions of the feature maps. A commonly used method is max-pooling.

Consider, eg, a stride of 2:



Average pooling is another possibility.

A typical CNN architecture
(From LeCun et.al. "Gradient-Based Learning Applied to Document Recognition").

11

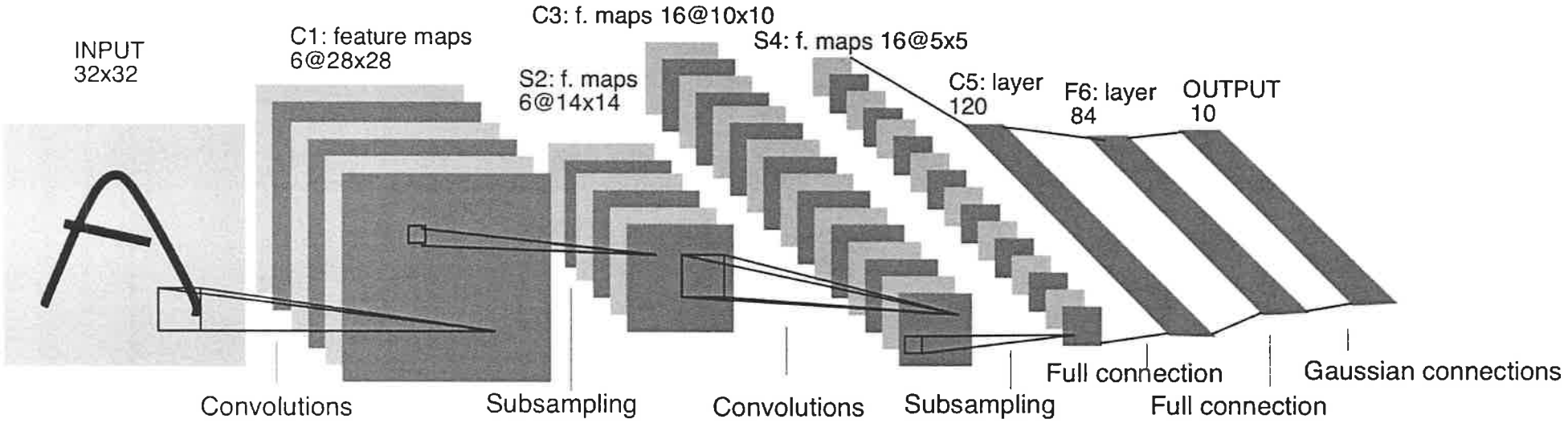


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

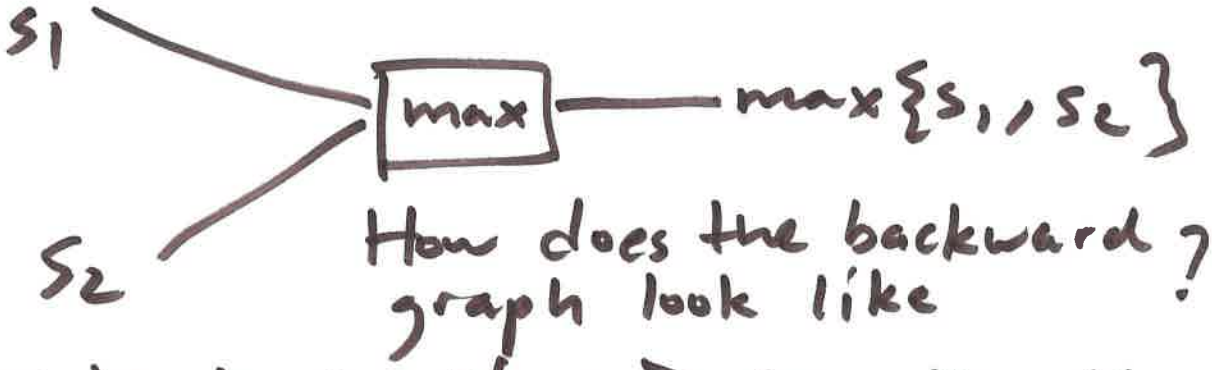
13

Fig. 3. Initial parameters of the output RBFs for recognizing the full ASCII set.

Training a CNN

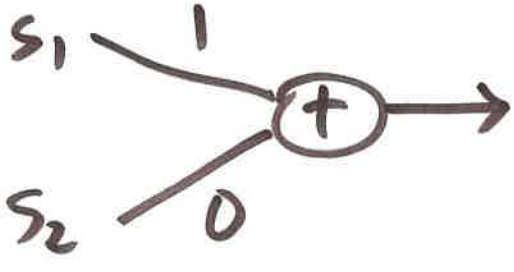
Is accomplished through the Backpropagation algorithm (similar to any multilayer NN).

- Should take care of issues like:
  - * Weight sharing
    (Not a big problem; just accumulate gradients / add them up).
  - * max. pooling:

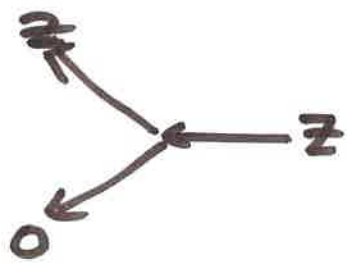


How does the backward graph look like?

Note that, if $s_1 \geq s_2$, then the above graph is equivalent to:



, so that the backward graph should be:



Similarly the case $s_1 \leq s_2$ is handled.

L13

ECE/CS 559 Lecture 13 T 10/8

Last time: Regularization

- Idea: Avoid overexplaining by limiting to reasonable explanations: prior knowledge / constraints on weights / biases
- $\min_w R(w)$ for $\|w\|_2 \le C$ $\iff$ $\min_w R(w) + \lambda \|w\|_2^2$ (L_2)
- During gradient descent simply add: $\nabla_w(R(w) + \lambda\|w\|_2^2) = \nabla_w R + 2\lambda w$ (L_1: $\|w\|_1 = |w_1| + \dots + |w_n|$, other choice; $\sim \text{sign}(w)$)
  - regularized / penalized risk
- Dropout: Randomly prune weights during training (with prob. p). Include all weights (scaled by p) at test time.

① Optimization

ⓐ Stochastic Gradient Descent

Empirical risk $R(w) = \frac{1}{n}\sum_{(x,y)\in Data} \ell(y, f(x;w))$

"True" risk $R'(w) = \mathbb{E}[\ell(Y, f(X;w))]$ — what we really want to minimize, behavior on future points.

Note: $R(w)$ is an unbiased estimate of $R'(w)$: $\mathbb{E}[R(w)] = R'(w)$

This means the gradients are unbiased: $\mathbb{E}[\nabla_w R(w)] = \nabla_w R'(w)$

random/stochastic gradients

- But... this remains true for even a single point $\mathbb{E}[\nabla_w \ell(Y, f(x;w))] = \nabla_w R(w)$ or a mini batch of points $\mathbb{E}\left[\nabla_w \sum_{(x,y)\in B} \ell(y, f(x;w))\right] = \nabla_w R(w)$
- What changes? The variance. # points ↓ variance ↑ noisiness ↑

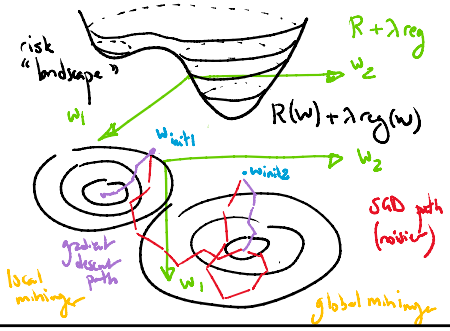
Then why don't we always use all the points?

- Reasons for minibatch SGD:
  - Computation: It's faster to get an update with small batches
  - Statistics: Homogeneous data ⇒ small batches give a glimpse of the whole
  - Optimization: The variance can be helpful to get out of local minima.

(choose batch size to trade off this part with convergence speed.)

- General form of SGD:

```
for epoch in #of epochs:
    t=0 ; Σ∇ = 0
    For (x,y) in Data:
        Forward(x); Backward(y) → gives ∇R + λ∇reg ; t += 1
        Σ∇ += ∇R + λ∇reg
        if (t+1) % B == 0:
            W += −η Σ∇
            Σ∇ = 0
```



ⓑ Choice of η, the "learning rate"

- Because SGD introduces noise, η should be adjusted
  - Otherwise, performance can worsen/improve/worsen (oscillate)
- Typically: Reduce η by a factor (e.g. 90%) if sustained worsening.
- Advanced: Adapt to the landscape (e.g. flatness, steepness)
- Vary η by layer: Later layers ∇_w ↑, make η ↓
  by neuron: # Inputs ↑, make η ↓ $\propto 1/\sqrt{\#\text{ inputs}}$ (heuristic)

ⓒ Controlling instability:

- Gradients are repeatedly multiplied during back propagation.
  - As a result, they could vanish ($\times 0.9 \times 0.9 \times \dots$) or explode ($\times 1.1 \times 1.1 \times \dots$)
  - Solution: normalize gradients (across a batch or a layer), clip gradients

② Heuristics

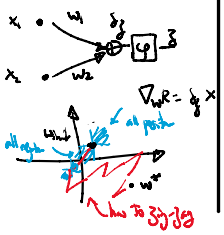
The "craft" of training neural networks.

ⓐ Choosing the activation function:

- For the longest time: Sigmoid σ, hyperbolic tangent tanh
  - Issue: $\varphi(v)$ saturates at +1 / −1 — saturation slows training
  - Instead: ReLU $\varphi(v)$ — no saturation there; hack + fix; ... but still an issue too

ⓑ Preprocessing inputs:

- Ideally, input coordinates are uncorrelated/centered
  - Otherwise, may slow down learning
  - Example: if x_i's are always + or − all together
- Preprocess to make them have 0 mean, unit variance, 0 covariance: "whitened"



- How? $\mu = \frac{1}{|Data|}\sum_{x\in Data} x$ (mean (vector)) $\quad C = \frac{1}{|Data|}\sum_{x\in Data}(x-\mu)(x-\mu)^T$ (covariance (matrix))

C is positive semidefinite: can be diagonalized $C = UDU^T$, $UU^T = U^TU = I$

Let $x_{new} = D^{-\frac{1}{2}}U^T(x-\mu)$ ← has 0 mean, covariance = I

ⓒ Initializing the weights:

- Intuition: maintain similar statistics from layer to layer.
- Ignoring activation, assuming whitened inputs, covariance I, next layer's inputs is:

$$C_i = \frac{1}{|Data|}\sum_{x\in Data}(Wx)(Wx)^T = WW^T = n\left[\ \ \right]n\left[\ \ \right]$$

Ideally, we want to make $\mathbb{E}[WW^T] = I$

- If w have 0 mean + uncorrelated, then off-diagonals are 0 ✓
- On the diagonal, we'd be adding n, $\mathbb{E}[w^2]$, so we get $n\,\text{var}(w)$.
- To make it identity, we just have to choose $\text{var}(w) = 1$
- E.g., choose w to be i.i.d. $\mathcal{N}\left(0, \frac{1}{\sqrt{\#\text{ inputs}}}\right)$