

10-701: Introduction to Machine Learning

Lecture 11 - Convolutional Neural Networks

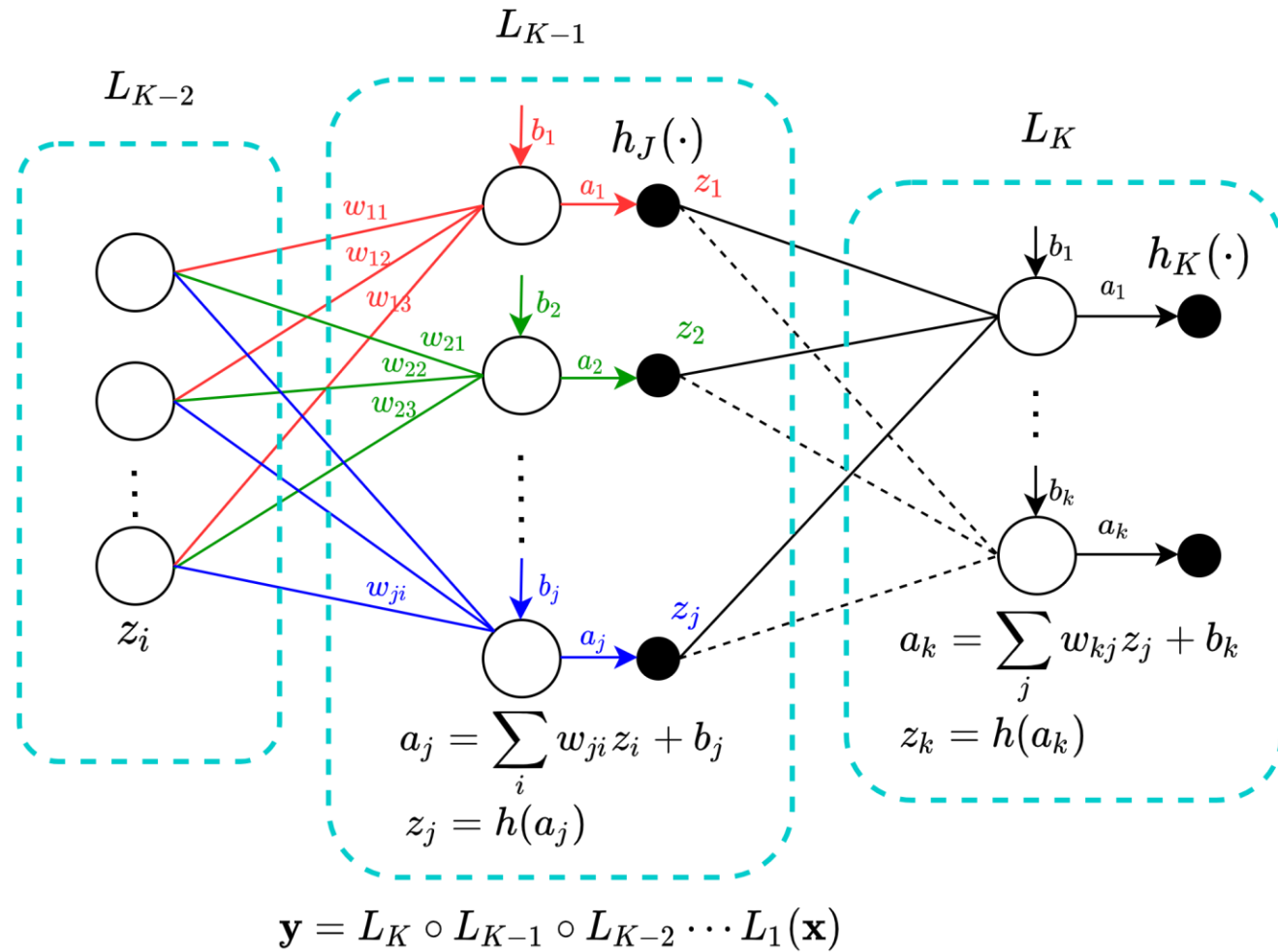
Hoda Heidari

* Slides adopted from F24 offering of 10701 by Henry Chai.

Recall: Stochastic Gradient Descent for Learning

- Input: $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N, \eta^{(0)}$
- Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0$
- While TERMINATION CRITERION is not satisfied
 - For $i \in \text{shuffle}(\{1, \dots, N\})$
 - For $l = 1, \dots, L$
 - Compute $G^{(l)} = \nabla_{W^{(l)}} \ell^{(i)}(W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)})$
 - Update $W^{(l)}$: $W_{(t+1)}^{(l)} = W_{(t)}^{(l)} - \eta_0 G^{(l)}$
 - Increment t : $t = t + 1$
- Output: $W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)}$

Back-propagation



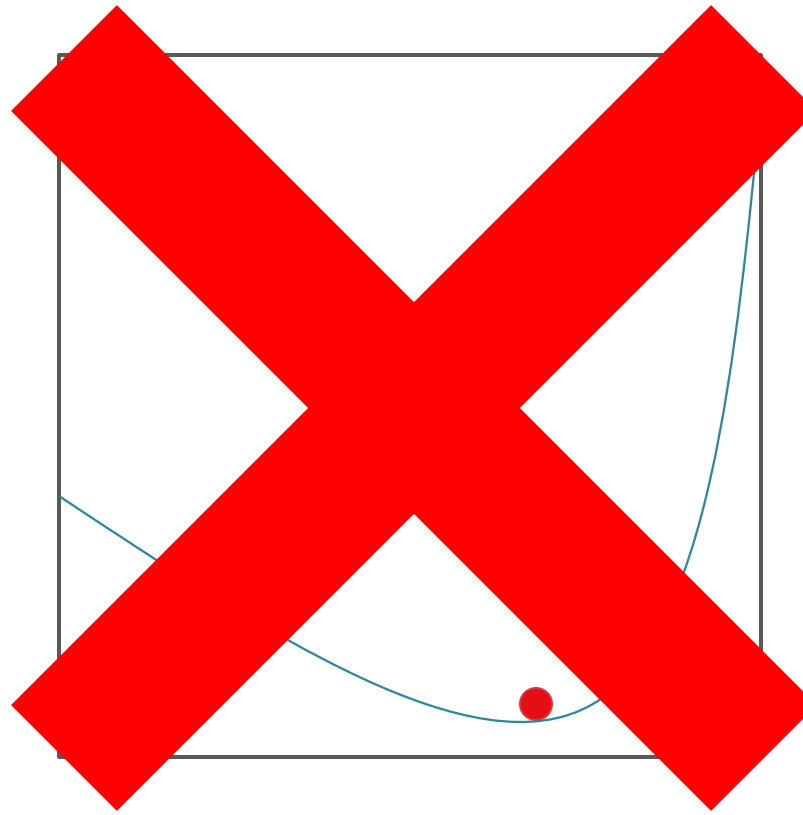
<https://towardsdatascience.com/error-backpropagation-5394d33ff49b/>

Back-propagation

- Input: $W^{(1)}, \dots, W^{(L)}$ and $(\mathbf{x}^{(i)}, y^{(i)})$
- Run forward propagation with $\mathbf{x}^{(i)}$ to get $\mathbf{o}^{(1)}, \dots, \mathbf{o}^{(L)}$
- (Optional) Compute $\ell^{(i)} = (o^{(L)} - y^{(i)})^2$
- Initialize: $\delta^{(L)} = 2(o_1^{(L)} - y^{(i)})$
- For $l = L - 1, \dots, 1$
 - Compute $\delta^{(l)} = W^{(l+1)T} \delta^{(l+1)} \odot (1 - \mathbf{o}^{(l)} \odot \mathbf{o}^{(l)})$
 - Compute $G^{(l)} = \delta^{(l)} \mathbf{o}^{(l-1)T}$
- Output: $G^{(1)}, \dots, G^{(L)}$, the gradients of $\ell^{(i)}$ w.r.t $W^{(1)}, \dots, W^{(L)}$

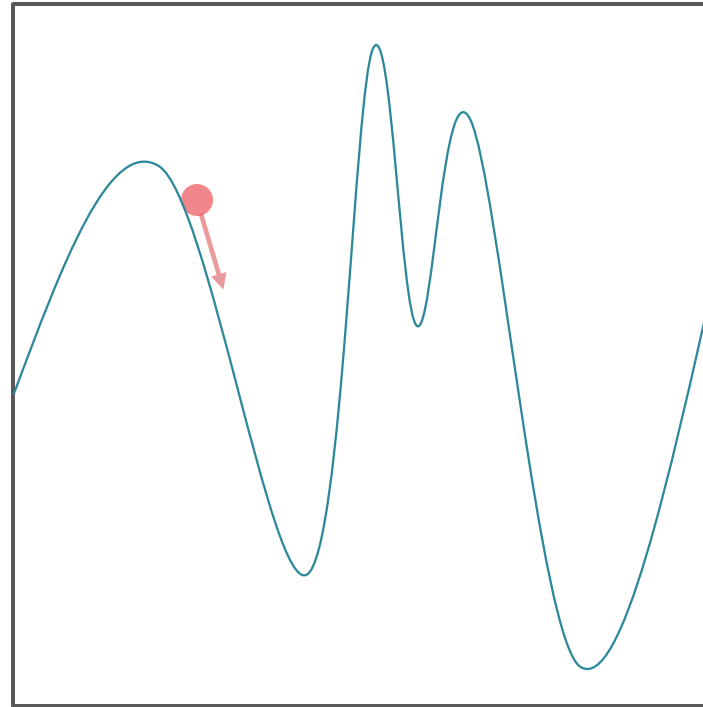
Recall: Gradient Descent

- Iterative method for minimizing functions
- Requires the gradient to exist everywhere



Non-convexity

- Gradient descent is not guaranteed to find a global minimum on non-convex surfaces



Stochastic Gradient Descent for Learning

- Input: $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N, \eta^{(0)}$
- Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0$
- While TERMINATION CRITERION is not satisfied
 - For $i \in \text{shuffle}(\{1, \dots, N\})$
 - For $l = 1, \dots, L$
 - Compute $G^{(l)} = \nabla_{W^{(l)}} \ell^{(i)}(W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)})$
 - Update $W^{(l)}$: $W_{(t+1)}^{(l)} = W_{(t)}^{(l)} - \eta_0 G^{(l)}$
 - Increment t : $t = t + 1$
- Output: $W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)}$

Mini-batch Stochastic Gradient Descent for Learning

- Input: $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N, \eta_{MB}^{(0)}, B$
- 1. Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0$
- 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from $\mathcal{D}, \{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient w.r.t. the sampled *batch*,
$$G^{(l)} = \frac{1}{B} \sum_{b=1}^B \nabla_{W^{(l)}} \ell^{(b)} \left(W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)} \right) \forall l$$
 - c. Update $W^{(l)}: W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \eta_{MB}^{(0)} G^{(l)} \forall l$
 - d. Increment $t: t \leftarrow t + 1$
- Output: $W_t^{(1)}, \dots, W_t^{(L)}$

Momentum

- **Idea:** Add a "velocity" term": a weighted average of previous gradients, to guide the weight updates.
 - It helps the optimizer to overcome local minima,
 - speed up convergence in flat regions of the loss landscape,
 - and dampen oscillations in regions with high curvature by smoothing the update path.

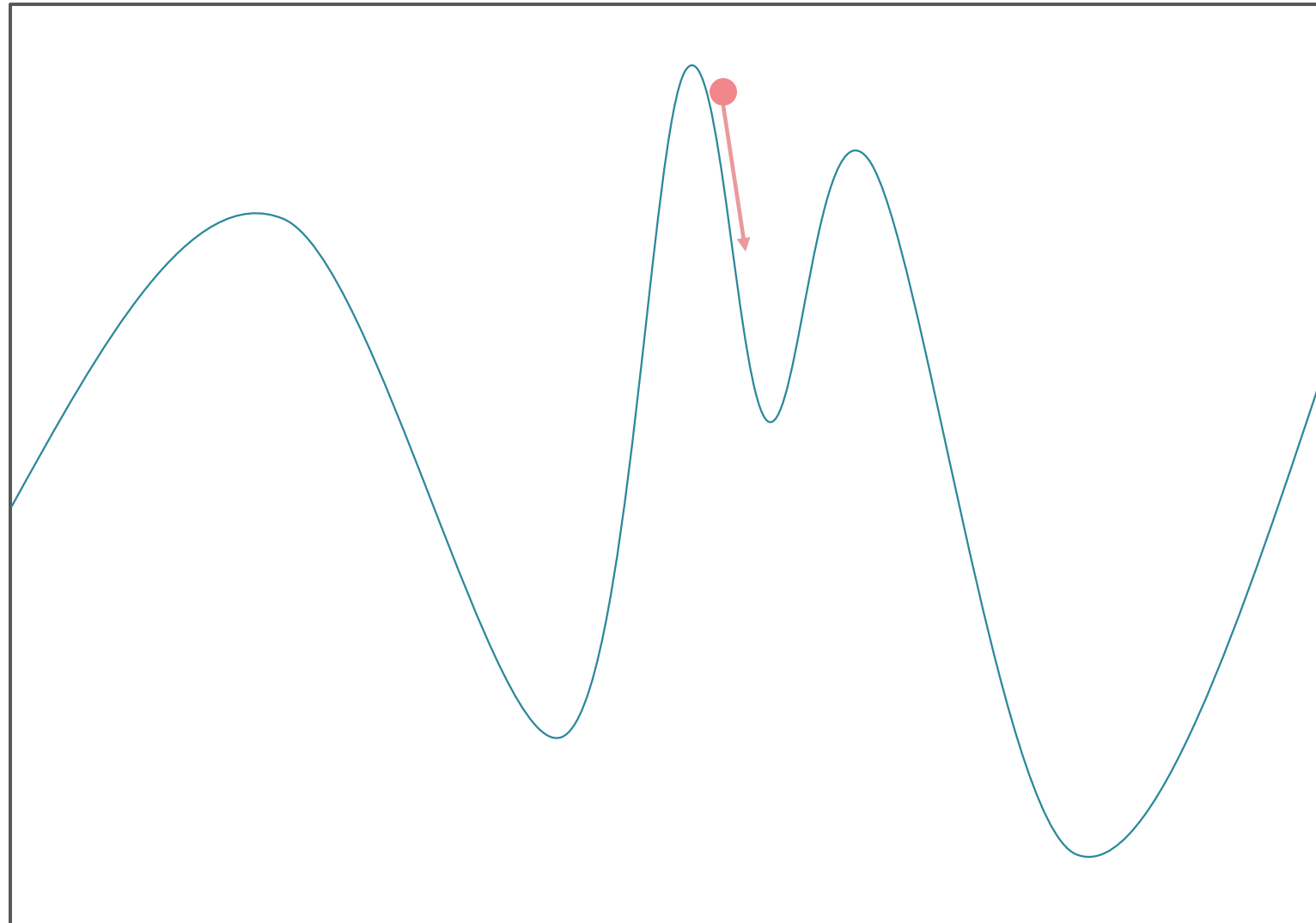
Regular SGD: A ball that's easily deflected by every small bump on the surface.

SGD with Momentum: The ball gains momentum, and once rolling, it's less affected by small bumps and continues in the general direction of the slope, helping it reach the bottom faster and more smoothly.

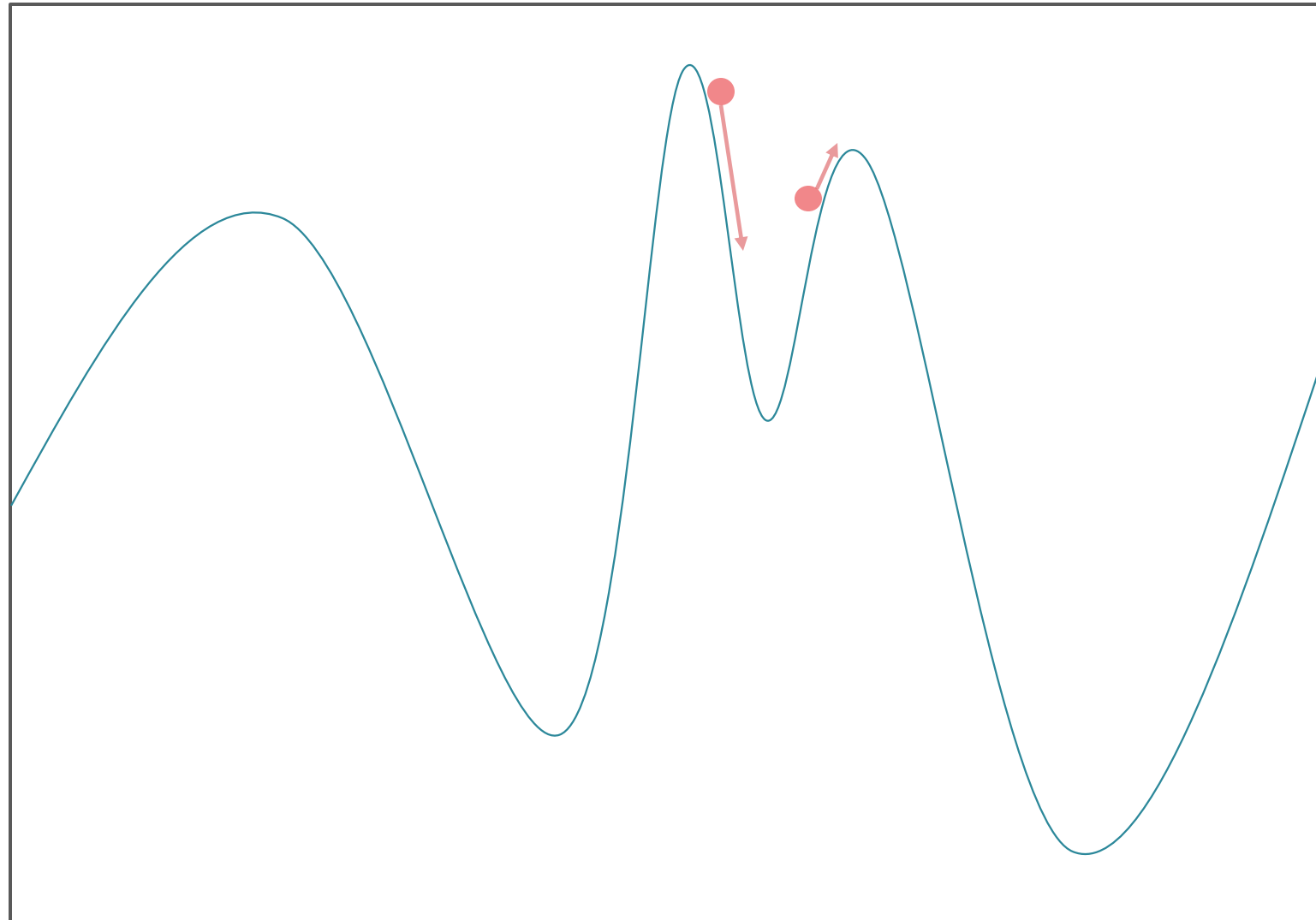
Mini-batch Stochastic Gradient Descent with Momentum for Learning

- Input: $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N, \eta_{MB}^{(0)}, B, \text{momentum parameter } \beta$
- 1. Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0, G_{-1}^{(l)} = 0 \odot W^{(l)} \forall l = 1, \dots, L$
- 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from $\mathcal{D}, \{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient w.r.t. the sampled *batch*,
$$G_t^{(l)} = \frac{1}{B} \sum_{b=1}^B \nabla_{W^{(l)}} \ell^{(b)} \left(W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)} \right) \forall l$$
 - c. Update $W^{(l)}: W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \eta_{MB}^{(0)} \left(\beta G_{t-1}^{(l)} + G_t^{(l)} \right) \forall l$
 - d. Increment $t: t \leftarrow t + 1$
- Output: $W_t^{(1)}, \dots, W_t^{(L)}$

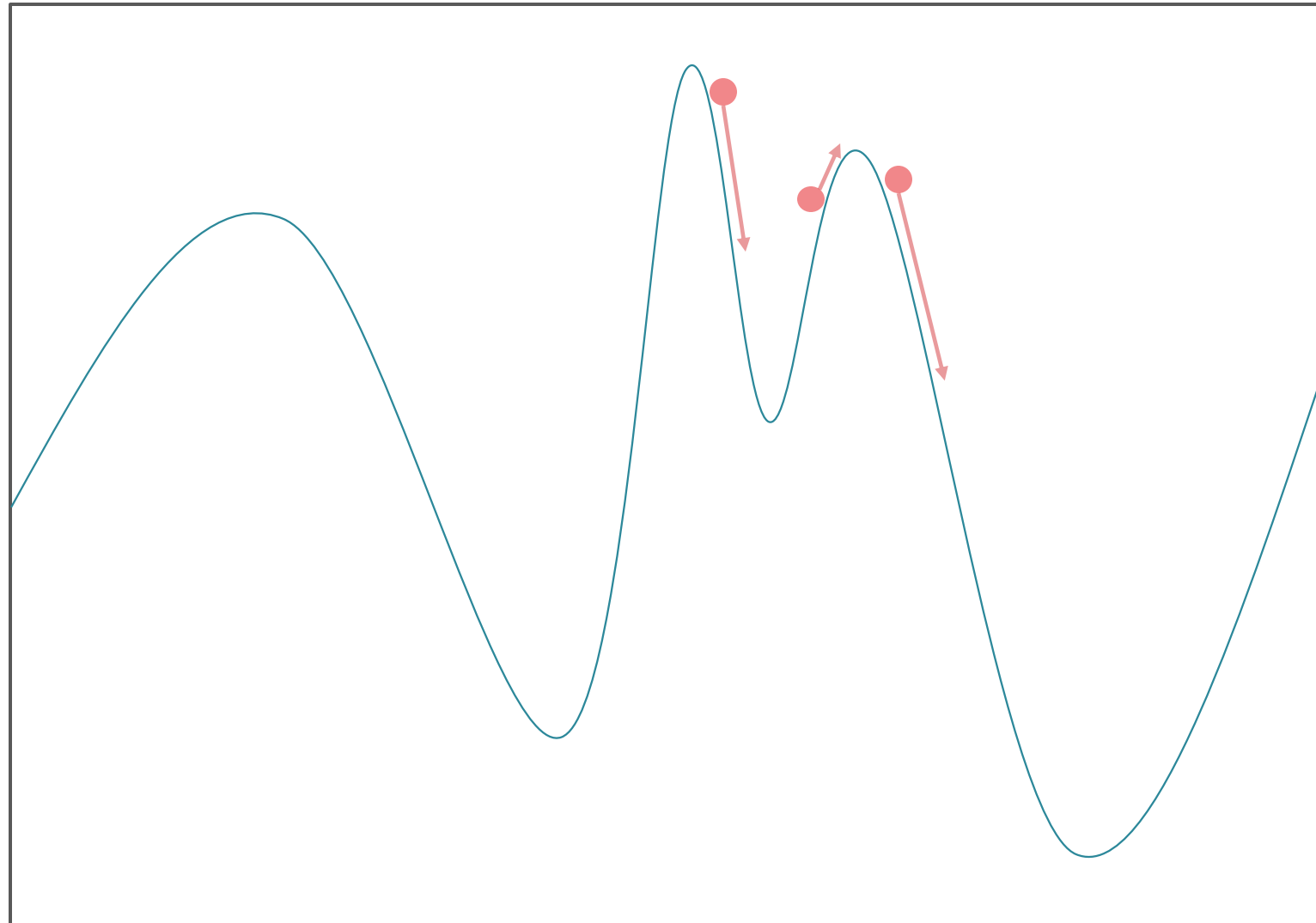
Mini-batch Stochastic Gradient Descent with Momentum for Learning



Mini-batch Stochastic Gradient Descent with Momentum for Learning



Mini-batch Stochastic Gradient Descent with Momentum for Learning



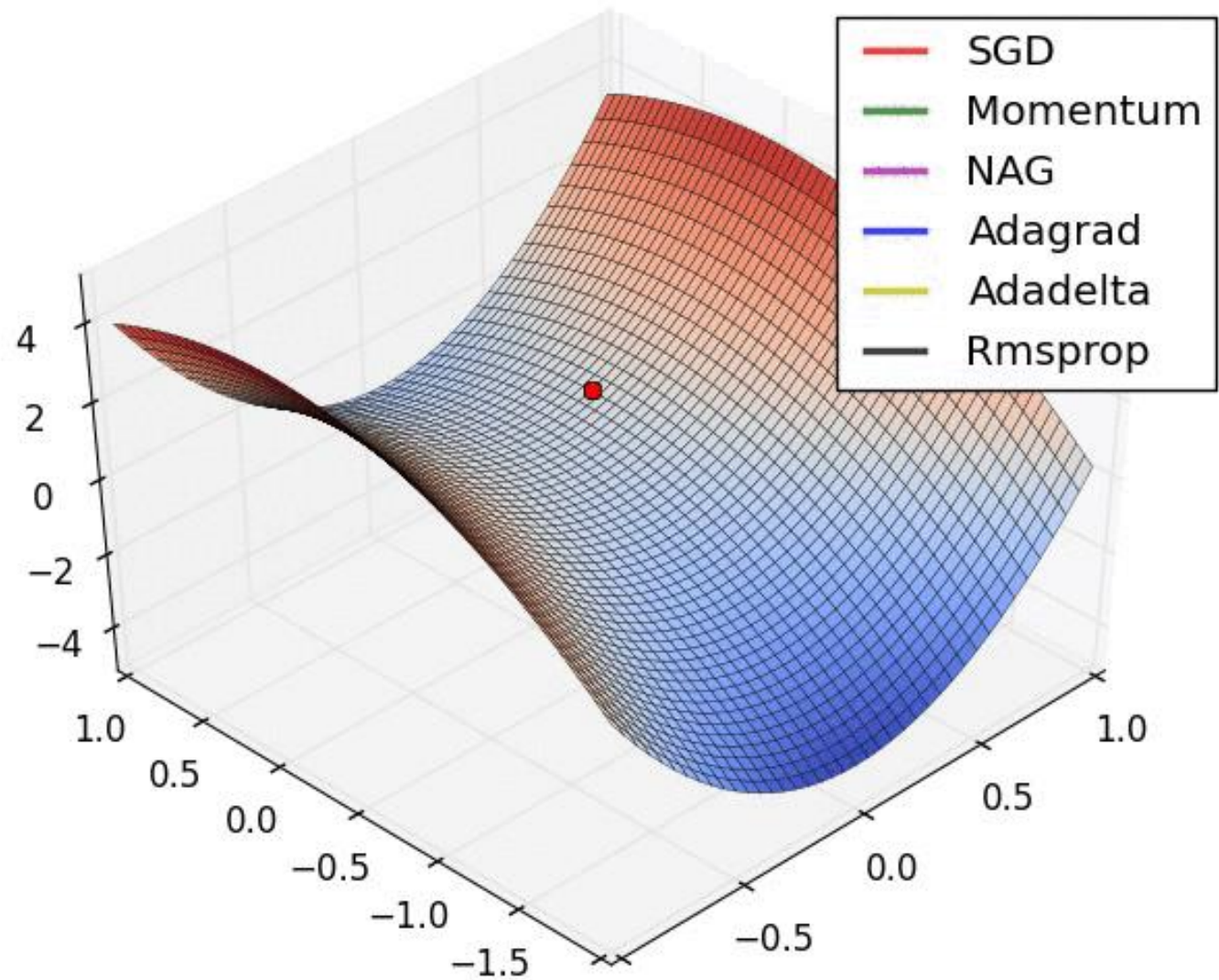
Adaptive Learning Rates

- SGD has a *constant* learning rate, which can cause oscillations in steep loss function areas and slow convergence in flatter regions.
- **Root Mean Square Propagation (RMSProp)** maintains a decaying average of the squared gradients:
 - scaling down the learning rate in directions with large, frequent gradients (flat optimization surface)
 - scaling it up in directions with small, infrequent gradients (steep surface, prevents leaping over the optimum)

Mini-batch Stochastic Gradient Descent with Root Mean Square Propagation (RMSProp)

- Input: $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N, \eta_{MB}^{(0)}, B$, decay parameter β
- 1. Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0, S_{-1}^{(l)} = 0 \odot W^{(l)} \forall l = 1, \dots, L$
- 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from $\mathcal{D}, \{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient w.r.t. the sampled *batch*,
$$G_t^{(l)} = \frac{1}{B} \sum_{b=1}^B \nabla_{W^{(l)}} \ell^{(b)} \left(W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)} \right) \forall l$$
 - c. Update the scaling factor: $S_t = \beta S_{t-1} + (1 - \beta)(G_t \odot G_t)$
 - d. Update $W^{(l)}$: $W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \frac{\gamma}{\sqrt{S_t}} \odot G_t$
 - e. Increment t : $t \leftarrow t + 1$
- Output: $W_t^{(1)}, \dots, W_t^{(L)}$

Mini-batch Stochastic Gradient Descent with Root Mean Square Propagation (RMSProp)



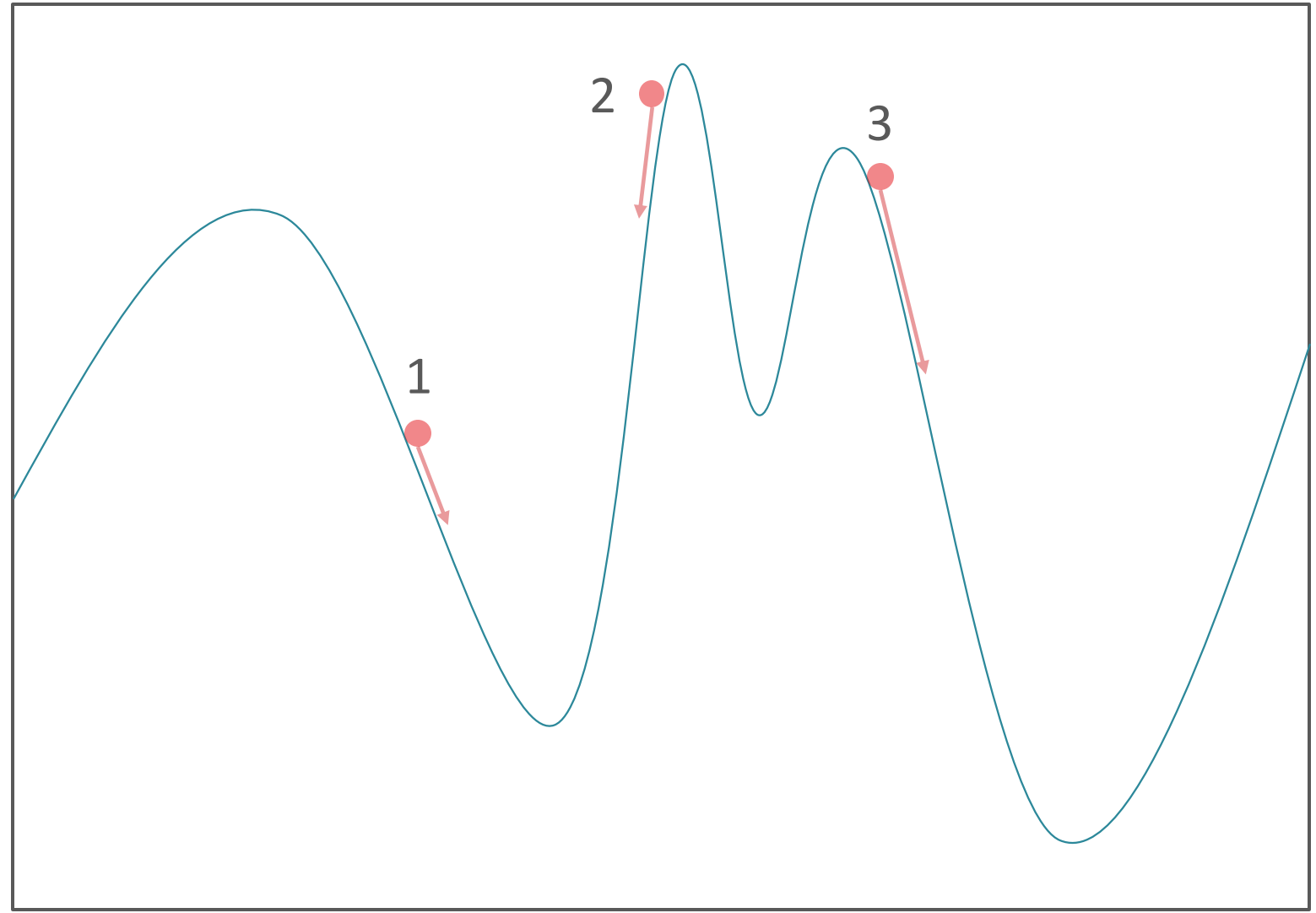
Adam (Adaptive Moment Estimation) = SGD + Momentum + RMSProp

- Input: $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N, \eta_{MB}^{(0)}, B$, decay parameters β_1 and β_2
- 1. Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0, M_{-1} = S_{-1} = 0 \odot W^{(l)} \forall l = 1, \dots, L$
- 2. While TERMINATION CRITERION is not satisfied
 - a. Randomly sample B data points from $\mathcal{D}, \{(\mathbf{x}^{(b)}, y^{(b)})\}_{b=1}^B$
 - b. Compute the gradient (G_t), momentum and scaling factor
$$M_t = \beta_1 M_{t-1} + (1 - \beta_1) G_t$$
$$S_t = \beta_2 S_{t-1} + (1 - \beta_2) (G_t \odot G_t)$$
 - c. Update $W^{(l)}: W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \frac{\eta}{\sqrt{S_t/(1-\beta_2^t)}} \odot (M_t/(1-\beta_1^t))$
 - d. Increment $t: t \leftarrow t + 1$
- Output: $W_t^{(1)}, \dots, W_t^{(L)}$

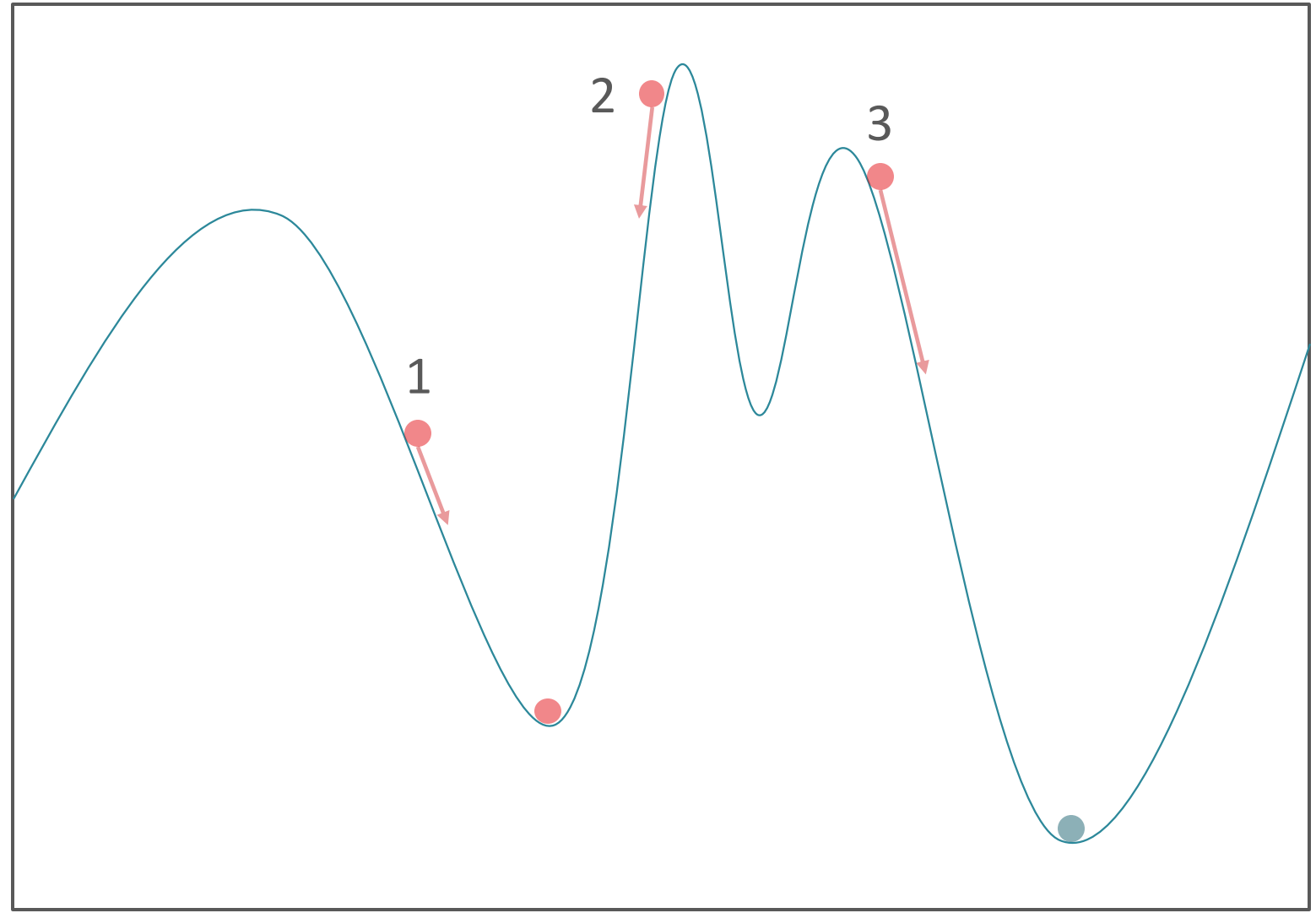
Random Restarts

- Run mini-batch gradient descent (with momentum & adaptive gradients) multiple times, each time starting with a ***different, random*** initialization for the weights.
- Compute the training error of each run at termination and return the set of weights that achieves the lowest training error.

Random Restarts

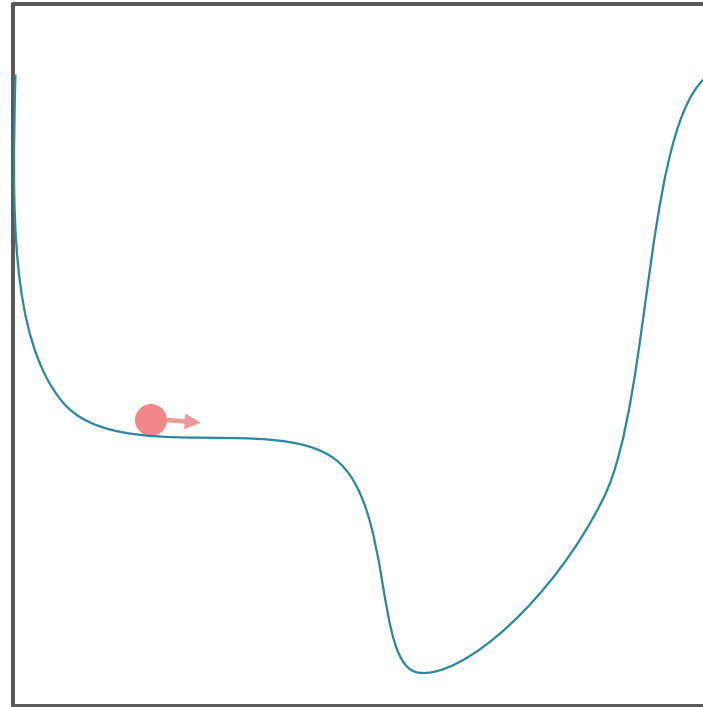


Random Restarts



Terminating Gradient Descent

- For non-convex surfaces, the gradient's magnitude is often not a good metric for proximity to a minimum



Terminating Gradient Descent “Early”

- For non-convex surfaces, the gradient's magnitude is often not a good metric for proximity to a minimum
- Combine multiple termination criteria e.g. only stop if enough iterations have passed and the improvement in error is small
- Alternatively, terminate early by using a validation data set: if the validation error starts to increase, just stop!
 - Early stopping asks like regularization by **limiting how much of the hypothesis set** is explored

Neural Networks and Regularization

- Minimize $\ell_{AUG}^{(i)}(W^{(1)}, \dots, W^{(L)}, \lambda_c)$
 $= \ell^{(i)}(W^{(1)}, \dots, W^{(L)}) + \lambda_c r(W^{(1)}, \dots, W^{(L)})$

e.g. L2 regularization

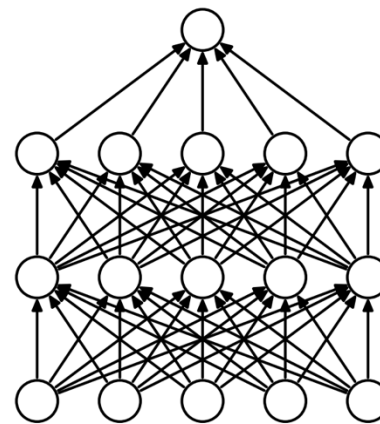
$$r(W^{(1)}, \dots, W^{(L)}) = \sum_{l=1}^L \sum_{i=0}^{d^{(l-1)}} \sum_{j=1}^{d^{(l)}} (w_{j,i}^{(l)})^2$$

Neural Networks and “Strange” Regularization (Bishop, 1995)

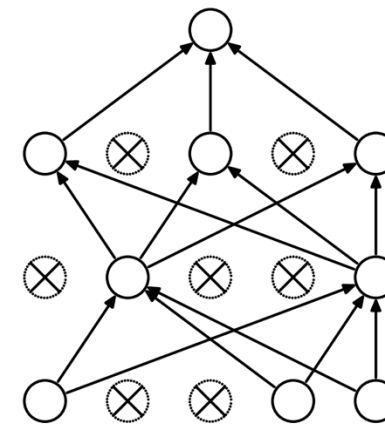
- Jitter
 - In each iteration of gradient descent, add some random noise or “jitter” to each training data point
 - Instead of computing the gradient w.r.t. $(\mathbf{x}^{(i)}, y^{(i)})$, use $(\mathbf{x}^{(i)} + \boldsymbol{\epsilon}, y^{(i)})$ where $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2 I)$
 - Makes neural networks resilient to input noise
 - Has been proven to be equivalent to using a certain kind of regularizer r for some error metrics

Neural Networks and “Strange” Regularization (Srivastava et al., 2014)

- Dropout
 - In each iteration of gradient descent, randomly remove some of the nodes in the network
 - Compute the gradient using only the remaining nodes
 - The weights on edges going into and out of “dropped out” nodes are not updated



(a) Standard Neural Net



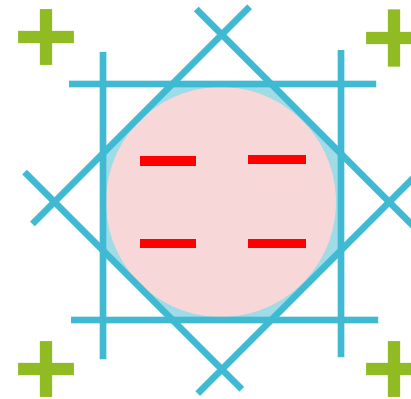
(b) After applying dropout.

Key Takeaways

- The loss function for neural networks is non-convex!
 - Momentum can help break out of local minima
 - Adaptive gradients help when parameters behave differently w.r.t. step sizes
 - Random restarts can improve the chances of finding better local minima
 - Jitter & dropout act like regularization for neural networks by preventing them fitting the training dataset perfectly

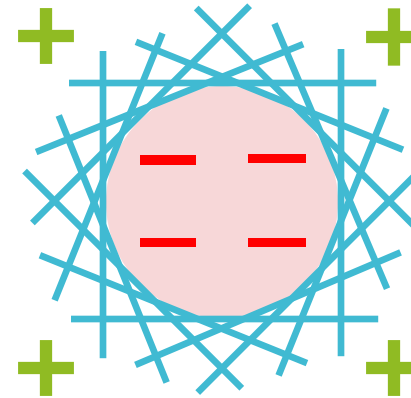
MLPs as Universal Approximators

- Theorem: any function that can be decomposed into perceptrons can be modelled exactly using a 3-layer MLP
- Any smooth decision boundary can be approximated to an arbitrary precision using a finite number of perceptrons



MLPs as Universal Approximators

- Theorem: any function that can be decomposed into perceptrons can be modelled exactly using a 3-layer MLP
- Any smooth decision boundary can be approximated to an arbitrary precision using a finite number of perceptrons



- Theorem: Any smooth decision boundary can be approximated to an arbitrary precision using a 3-layer MLP

NNs as Universal Approximators (Cybenko, 1989 & Hornik, 1991)

- Theorem: Any bounded, continuous function can be approximated to an arbitrary precision using a 2-layer (1 hidden layer) feed-forward NN if the activation function, θ , is continuous, bounded and non-constant.

NNs as Universal Approximators (Cybenko, 1988)

- Theorem: Any function can be approximated to an arbitrary precision using a 3-layer (2 hidden layers) feed-forward NN if the activation function, θ , is continuous, bounded and non-constant.

Are Shallow NNs All We Need In Practice?

- A shallow network may need an *exponentially large number of neurons* to represent certain functions.
- Deeper networks can represent the same functions with *far fewer parameters* by reusing and composing features across layers.
- Depth enables **layered feature abstraction**:
 - Early layers learn low-level patterns (e.g., edges in images).
 - Middle layers capture mid-level structures (shapes).
 - Higher layers encode high-level concepts (objects).

Deep Learning

- From Wikipedia's page on Deep Learning...

Definition [\[edit \]](#)

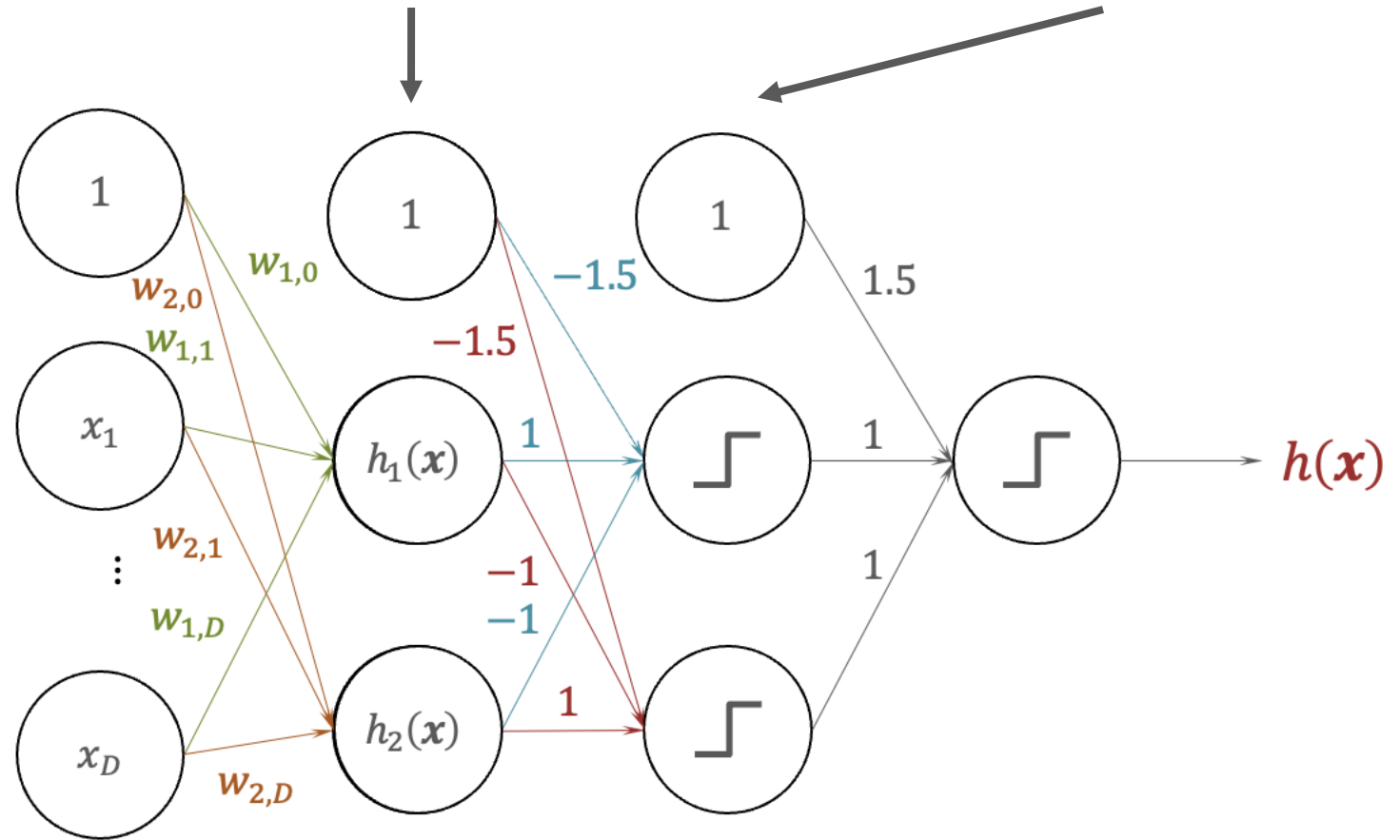
Deep learning is a class of [machine learning algorithms](#) that^[11](pp199–200) uses multiple layers to progressively extract higher level features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.

- Deep learning = more than one layer

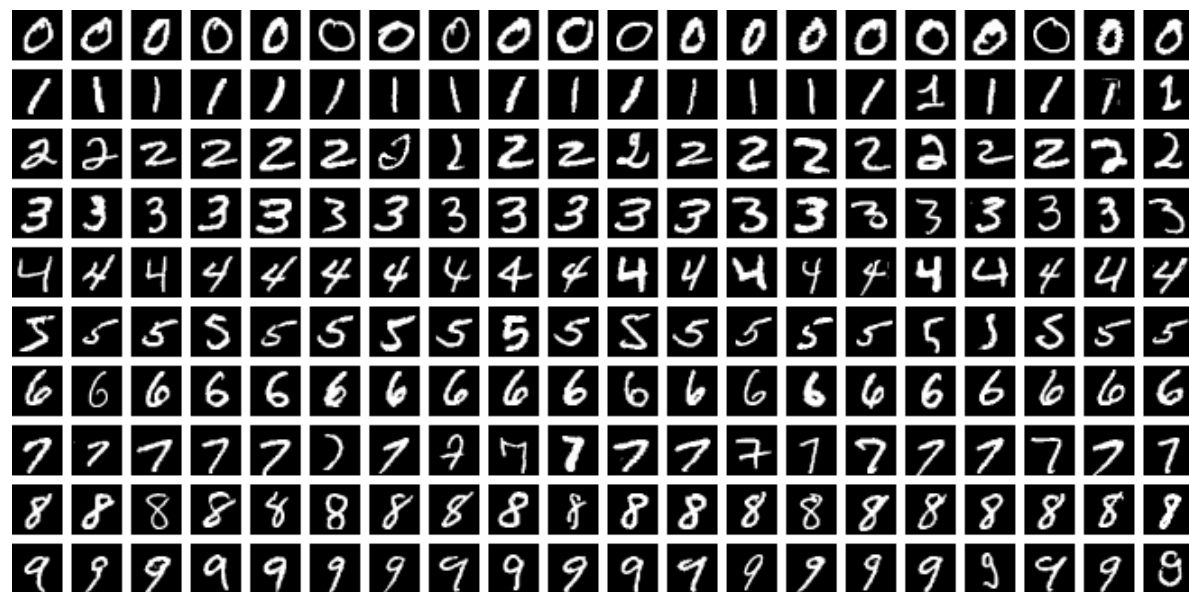
Deep Learning

First layer: computes the perceptrons' predictions

Second layer: combines lower-level components



Neural Networks for Image Recognition



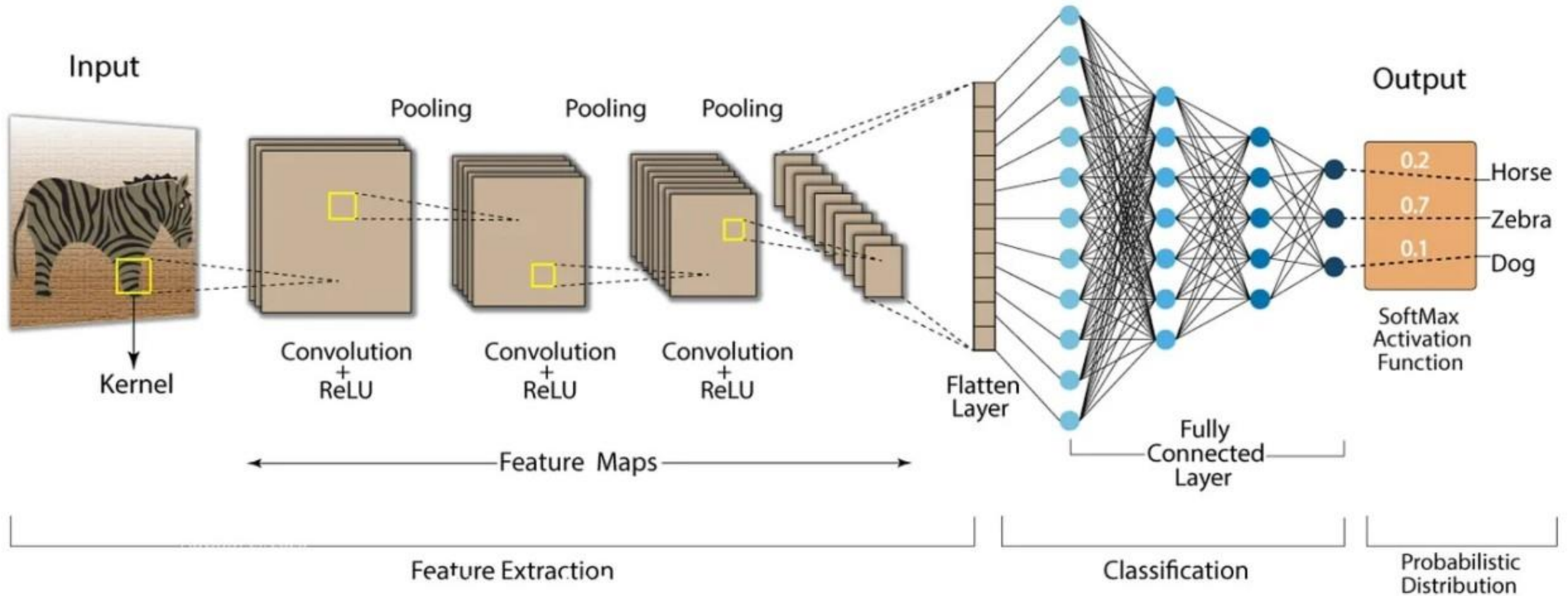
Why Not Fully Connected Feed Forward NNs?

- **Poll:** Suppose we are using a fully connected feed forward NN to **detect edges** in an image. How many parameters are there if the input and output images are both 100x100 pixels?

Convolutional Neural Networks

- Neural networks are frequently applied to inputs with some inherent spatial structure, **e.g., images**
- Idea: use the first few layers to identify relevant macro-features, **e.g., edges**
- Insight: for spatially-structured inputs, many useful macro-features are shift or location-invariant, **e.g., an edge in the upper left corner of a picture looks like an edge in the center**
- Strategy: learn a *filter* for macro-feature detection in a small window and apply it over the entire image

Convolution Neural Network (CNN)



Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A filter is just a small matrix that is convolved with same-sized sections of the image matrix

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

 $*$

0	1	0
1	-4	1
0	1	0

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A filter is just a small matrix that is convolved with same-sized sections of the image matrix

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

 $*$

0	1	0
1	-4	1
0	1	0

 $=$

0			

$$(0 * 0) + (0 * 1) + (0 * 0) + (0 * 1) + (1 * -4) + (2 * 1) + (0 * 0) + (2 * 1) + (4 * 0) = 0$$

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A filter is just a small matrix that is convolved with same-sized sections of the image matrix

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

 $*$

0	1	0
1	-4	1
0	1	0

 $=$

0	-1		

$$(0 * 0) + (0 * 1) + (0 * 0) + (1 * 1) + (2 * -4) + (2 * 1) + (2 * 0) + (4 * 1) + (4 * 0) = -1$$

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A **filter/kernel** is just a small matrix that is convolved with same-sized sections of the image matrix

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





 $*$

0	1	0
1	-4	1
0	1	0




 $=$

0	-1	-1	0
-2	-5	-5	-2
2	-2	-1	3
-1	0	-5	0

Convolutional Filters

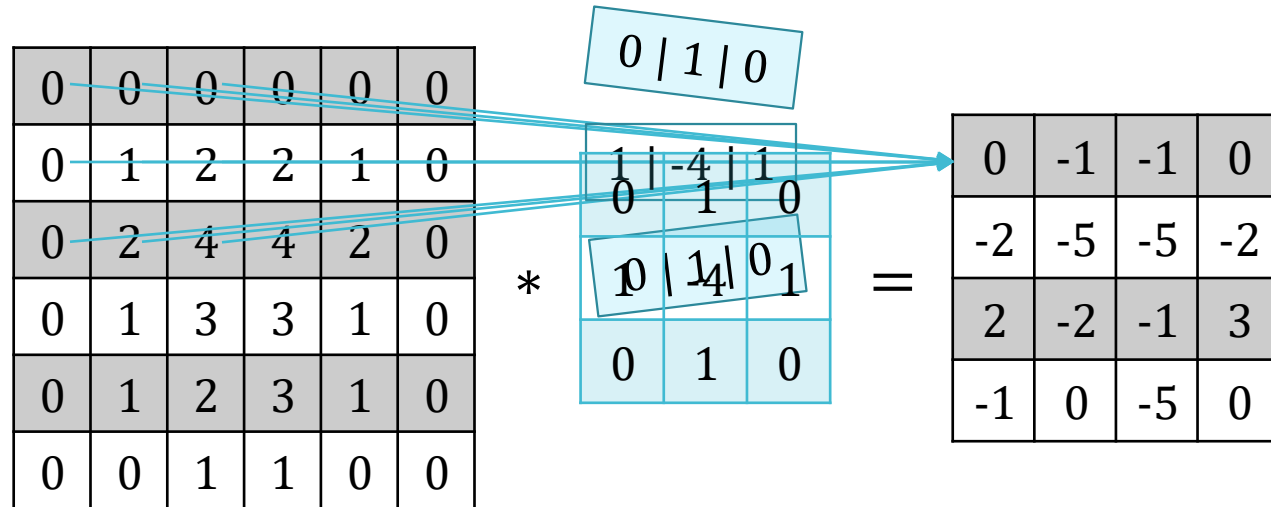
Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	

More Filters

Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

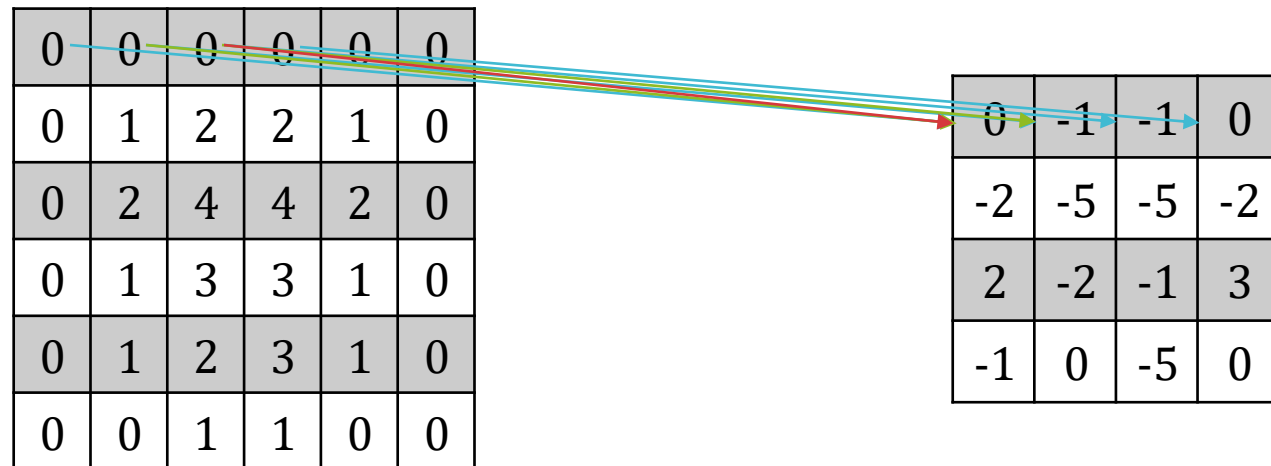
Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A filter is just a small matrix that is convolved with same-sized sections of the image matrix



Convolutional Filters

- Convolutions can be represented by a feed forward neural network where:
 1. Nodes in the input layer are only connected to some nodes in the next layer but not all nodes.
 2. Many of the weights have the same value.



- Many fewer weights than a fully connected layer!
- **Convolution weights are learned using gradient descent/backpropagation, not prespecified**

Convolutional Filters: Padding

- What if relevant features exist at the border of our image?
- Add zeros around the image to allow for the filter to be applied “everywhere” e.g. a *padding* of 1 with a 3x3 filter preserves image size and allows every pixel to be the center

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

 $*$

0	1	0
1	-4	1
0	1	0

 $=$

0	1	2	2	1	0
1	0	-1	-1	0	1
2	-2	-5	-5	-2	2
1	2	-2	-1	3	1
1	-1	0	-5	0	1
0	2	-1	0	2	0

Downsampling

- **Idea:** reduce the spatial size of the feature maps to
 - cut down the number of parameters and computations in later layers
 - reduce the risk of overfitting
 - make the model less sensitive to small shifts in input

Downsampling: Stride

- Only apply the convolution to some subset of the image
e.g., every other column and row = a *stride* of 2

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

 $*$

0	1
1	-2

 $=$

-2		

Downsampling: Stride

- Only apply the convolution to some subset of the image
e.g., every other column and row = a *stride* of 2

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

 $*$

0	1
1	-2

 $=$

-2	-2	

Downsampling: Stride

- Only apply the convolution to some subset of the image
e.g., every other column and row = a *stride* of 2

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

 $*$

0	1
1	-2

 $=$

-2	-2	1

Downsampling: Stride

- Only apply the convolution to some subset of the image
e.g., every other column and row = a *stride* of 2

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

 $*$

0	1
1	-2

 $=$

-2	-2	1
0		

Downsampling: Stride

- Only apply the convolution to some subset of the image
e.g., every other column and row = a *stride* of 2

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

 $*$

0	1
1	-2

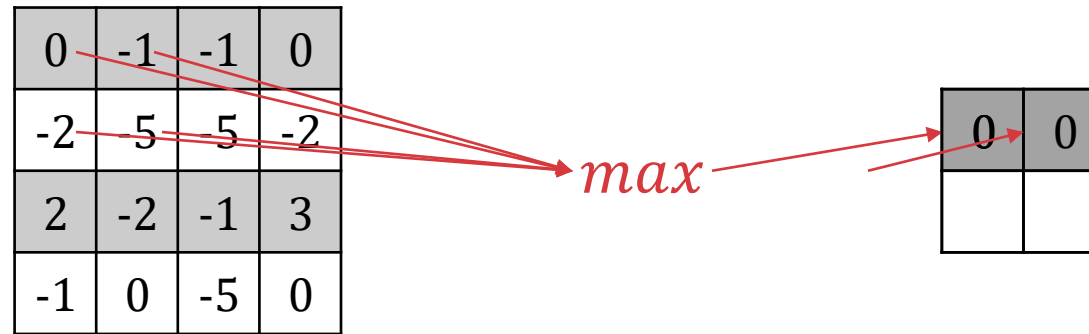
 $=$

-2	-2	1
0	1	1
1	2	0

- Reduces the dimensionality of the input to subsequent layers and thus, the number of weights to be learned
- Many relevant macro-features will tend to span large portions of the image, so taking strides with the convolution tends not to miss out on too much

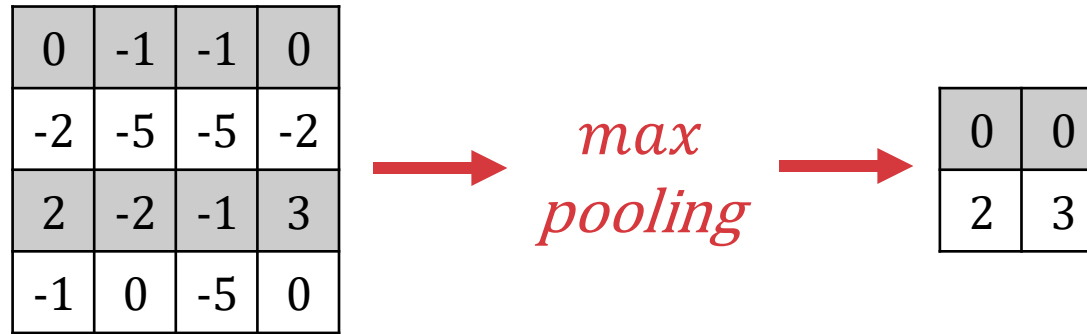
Downsampling: Pooling

- Combine multiple adjacent nodes into a single node

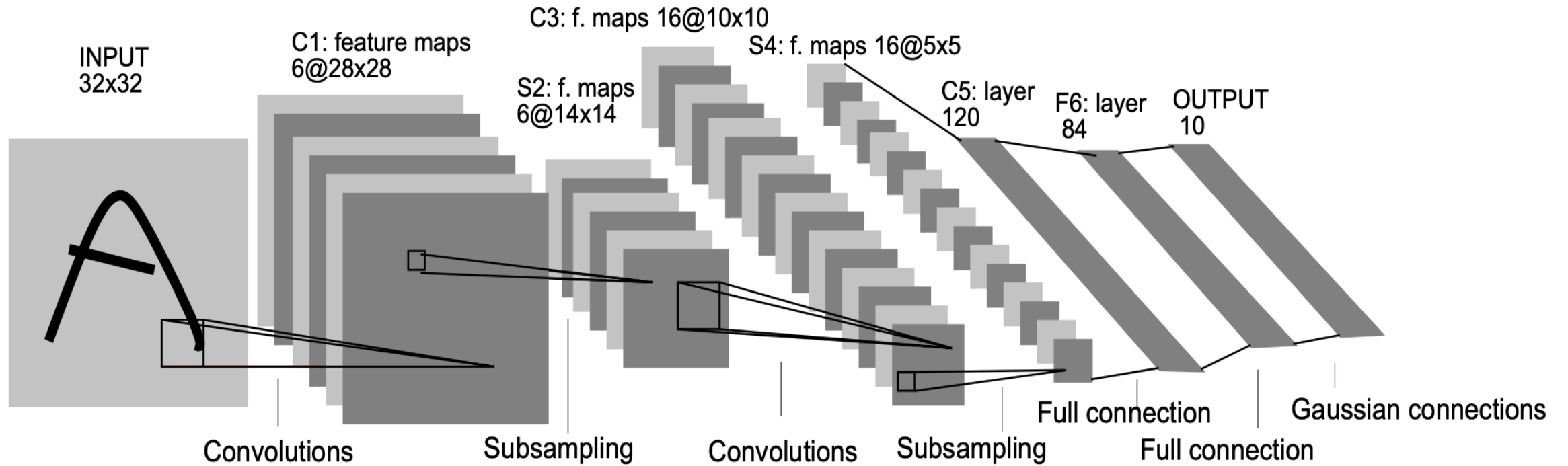


Downsampling: Pooling

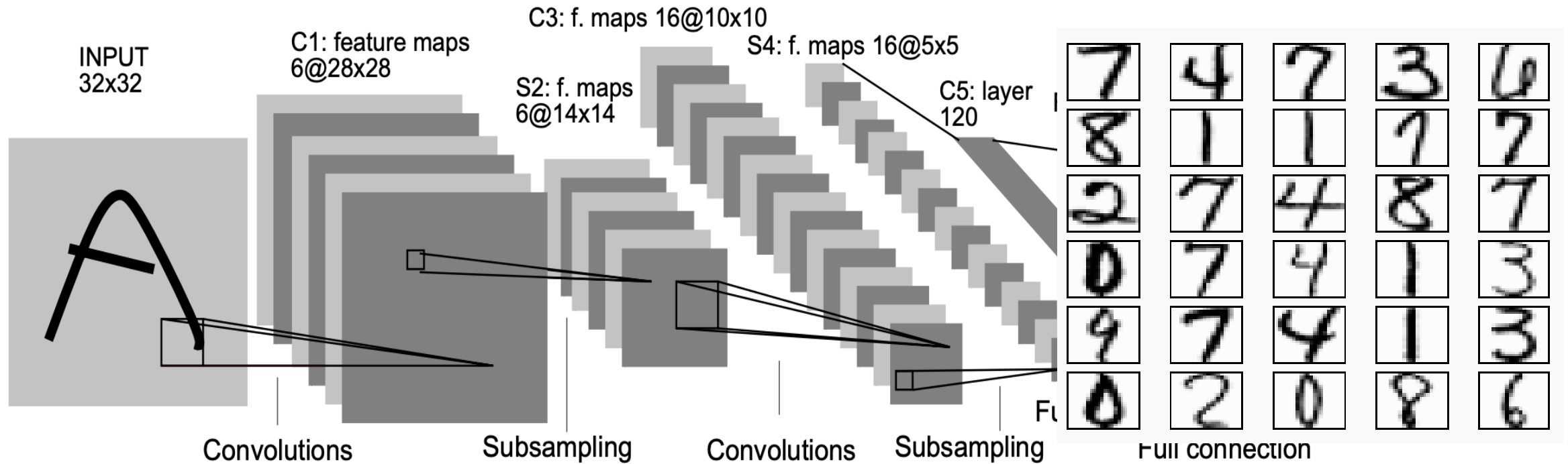
- Combine multiple adjacent nodes into a single node



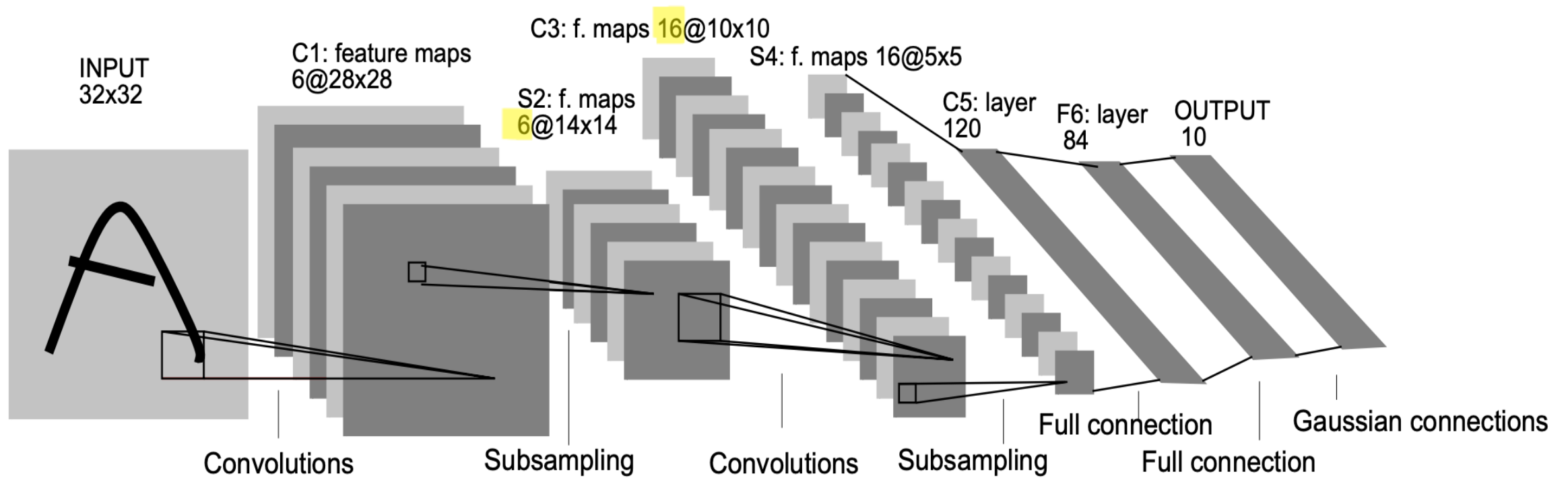
- **Max Pooling** keeps the strongest activation in each region, focusing on the most prominent features.
- **Average Pooling** computes the average of the region, providing a smoother, more generalized representation.



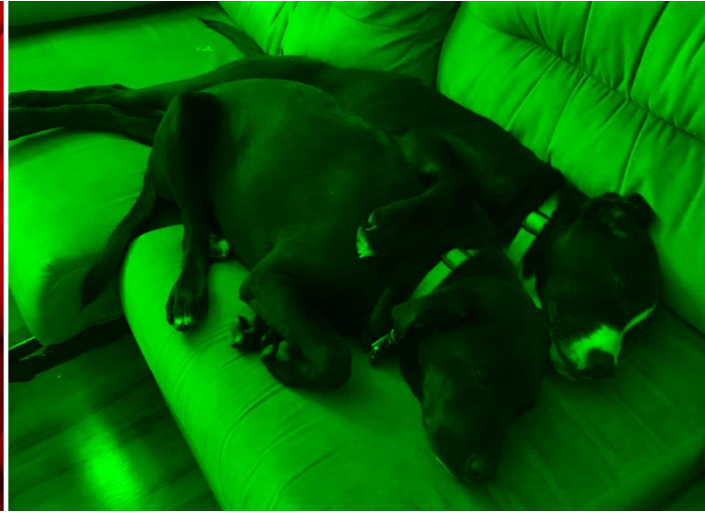
LeNet (LeCun et al., 1998)



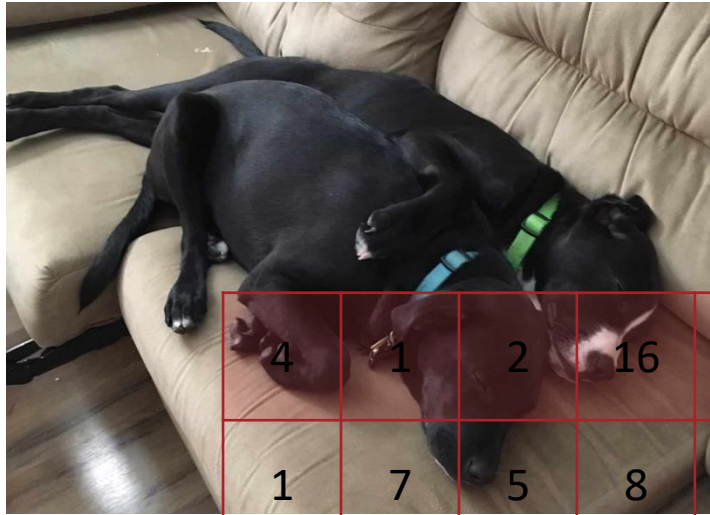
- One of the earliest, most famous deep learning models – achieved remarkable performance at handwritten digit recognition (< 1% test error rate on MNIST)
- Used sigmoid (or logistic) activation functions between layers and mean-pooling, both of which are pretty uncommon in modern architectures



Wait how did we go from 6 to 16?



Channels



4	1	2	16	3	6
1	7	5	8	19	27
5	2	5	12	17	8
0	4	9	9	6	11

5	2	6	14	15	8
26	3	6	8	4	9
0	15	24	6	1	8
7	4	9	5	24	17

4	6	8	9	5	3
16	5	2	8	2	1
5	2	14	11	7	8
15	2	5	0	9	8

- An image can be represented as the sum of red, green and blue pixel intensities
- Each color corresponds to a *channel*



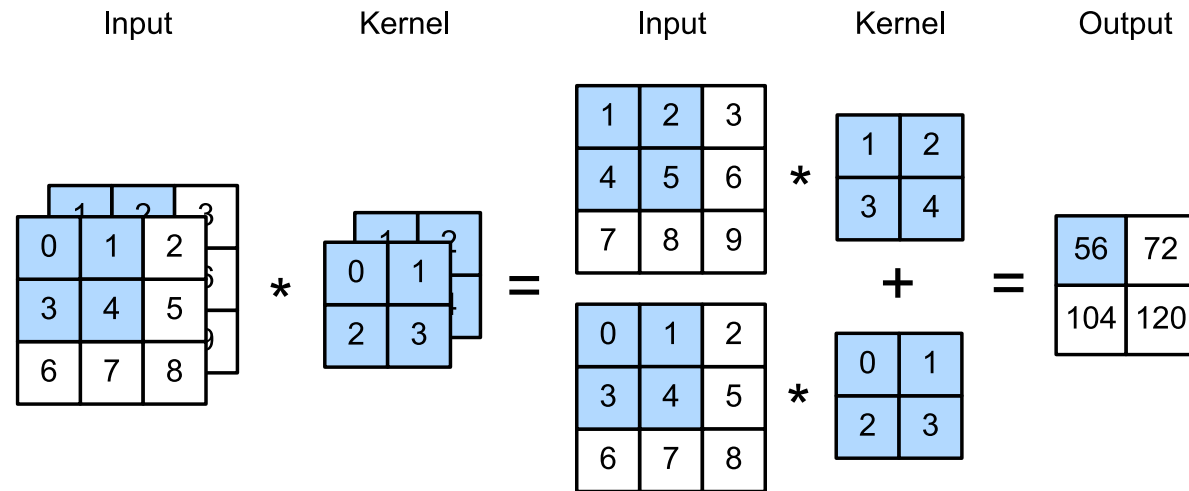
Example: $3 \times 4 \times 6$ tensor

4	1	2	16	3	6				
1		5	2	6	14	15	8		
5		26							
0		0		4	6	8	9	5	3
		7		16	5	2	8	2	1
				5	2	14	11	7	8
				15	2	5	0	9	8

- An image can be represented as a *tensor* or multidimensional array

Convolutions on Multiple Input Channels

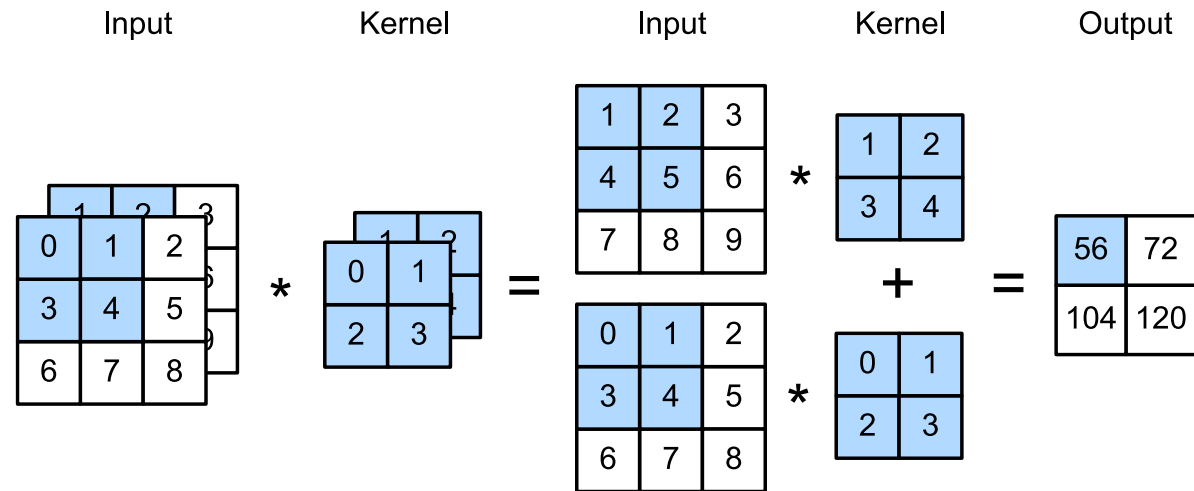
- Given multiple input channels, we can specify a filter for each one and sum the results to get a 2-D output tensor



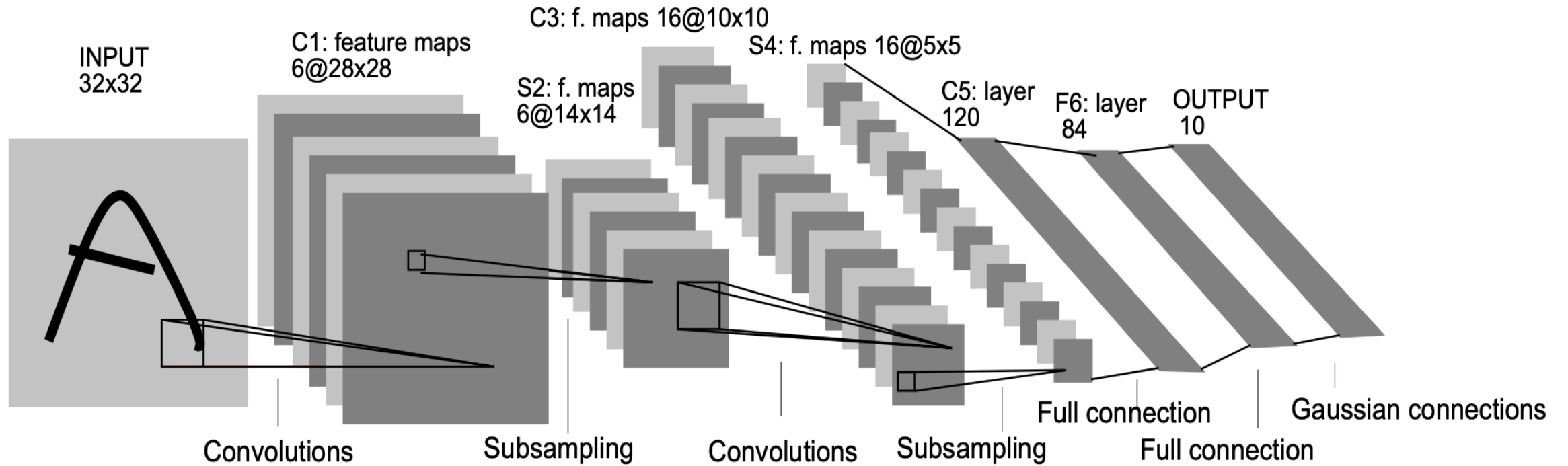
- For c channels and $h \times w$ filters, we have $c(hw + 1)$ learnable parameters (each filter has a bias term)

Convolutions on Multiple Input Channels

- Given multiple input channels, we can specify a filter for each one and sum the results to get a 2-D output tensor



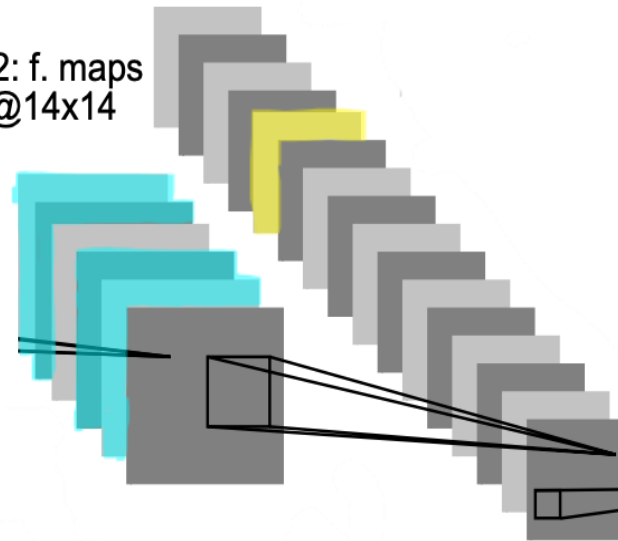
- Questions:
 - Why might we want a different filter for each channel?
 - Why do we combine them together into a single output channel?



- Channels in hidden layers correspond to different macro-features, which we might want to manipulate differently → one filter per channel

C3: f. maps 16@10x10

S2: f. maps
6@14x14



	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	X

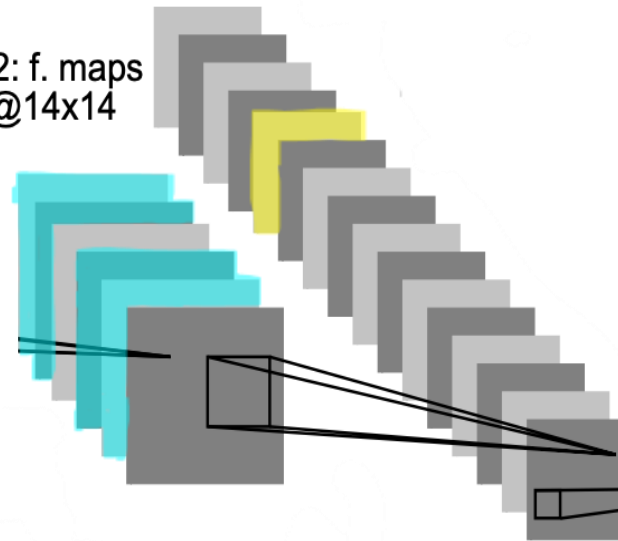
TABLE I

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

- We can combine these macro-features into a new, interesting, “higher-level” feature
 - But we don’t always need to combine all of them!
 - Different combinations → multiple output channels
 - Common architecture: more output channels and smaller outputs in deeper layers

C3: f. maps 16@10x10

S2: f. maps
6@14x14



	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

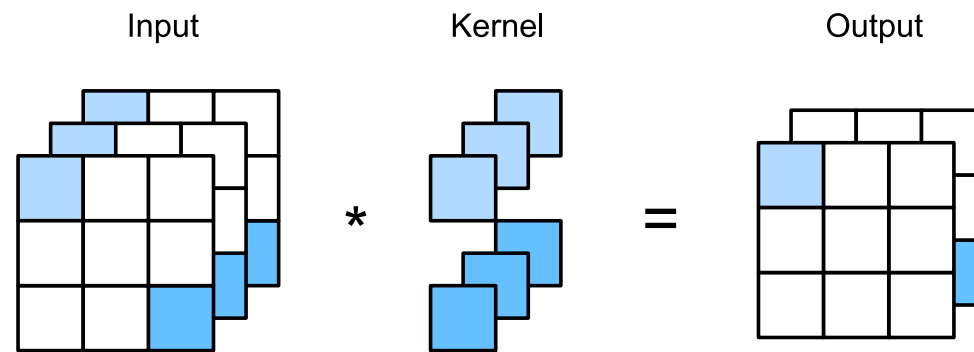
TABLE I

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

Okay, but what if our layers become too big in the channel dimension?

Downsampling: 1×1 Convolutions

- Convolutional layers can be represented as 4-D tensors of size $c_o \times c_i \times h \times w$ where c_o is the number of output channels and c_i is the number of input channels
- Layers of size $c_o \times c_i \times 1 \times 1$ can condense many input channels into fewer output channels (if $c_o < c_i$)



- Practical note: 1×1 convolutions are typically followed by a nonlinear activation function; otherwise, they could simply be folded into other convolutions

Downsampling : 1×1 Convolutions

- Suppose your input feature map is $32 \times 32 \times 256$ (width \times height \times channels).
- A single 1×1 filter will look at one pixel across all 256 channels at once, and compute a weighted sum of them.
- If you use 64 such filters, the output becomes $32 \times 32 \times 64$.

Key Takeaways

- Convolutional neural networks use convolutions with filters to learn macro-features
 - Can be thought of as slight modifications to the fully-connected feed-forward neural network
 - Can still be learned using SGD
 - Padding is used to preserve spatial dimensions while pooling, stride and 1×1 convolutions are used to downsample intermediate representations