Lecture 19

Deep Learning and Computer Vision

Who Am I?

- Hi, I'm Shubhang Desai
  - BS + MS from Stanford CS
  - Moved to Seattle ~2.5 years ago

- Applied Scientist at Microsoft, on Ink AI Team
  - Spearheaded deep learning handwriting recognizer (HWR)
  - Working on HWR, ink analysis, Copilot features
  - Both image and sequence modelling tasks

- Passionate about teaching
  - CS 131 (Comp. Vision), CS 230 (Deep Learning), CS 21SI (AI + Social Good) @ Stanford
  - CSE 493G1 (Deep Learning) @ UW

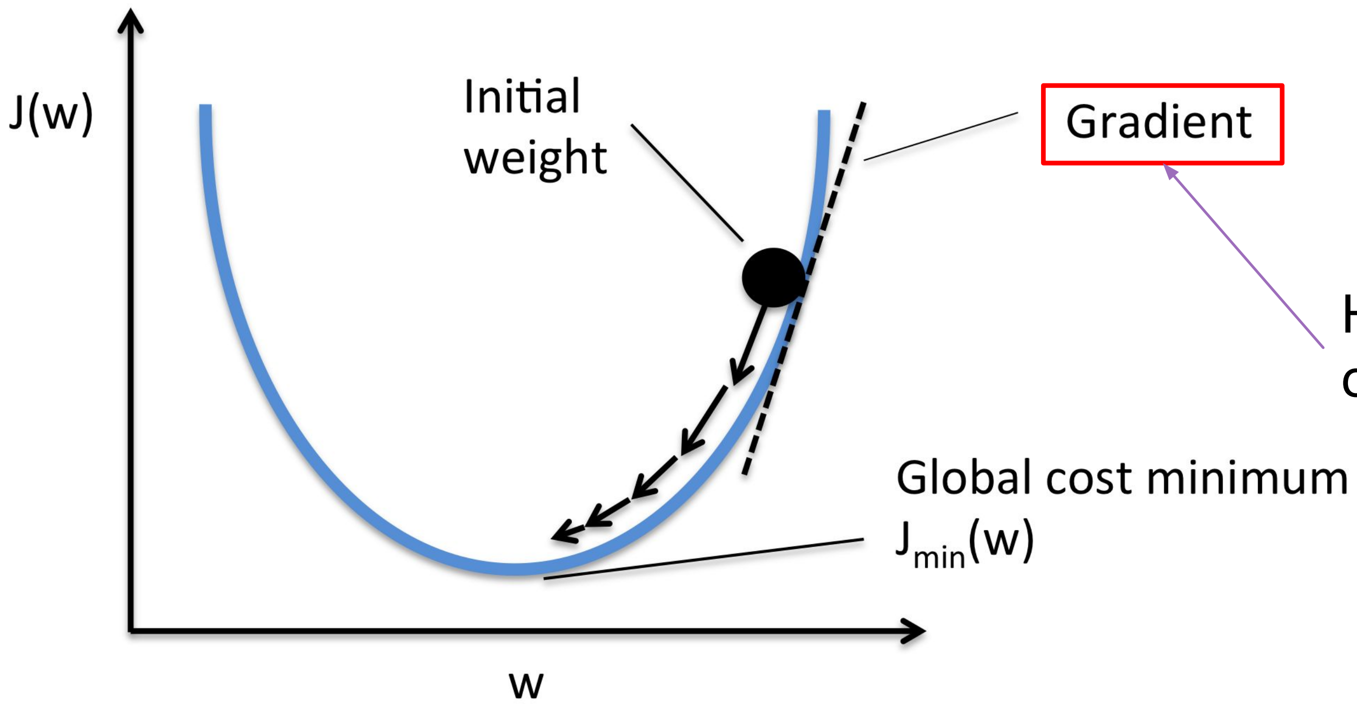
Plan for Today

- Deep dive on backpropagation
- Convolution-based classification algorithm
- Deep convolutional neural networks
- Vision transformers

Backpropagation Deep Dive

Refresher: Gradient Descent

Iteratively moving neural network weights in the direction of the gradient to minimize loss:



How do we actually compute this?

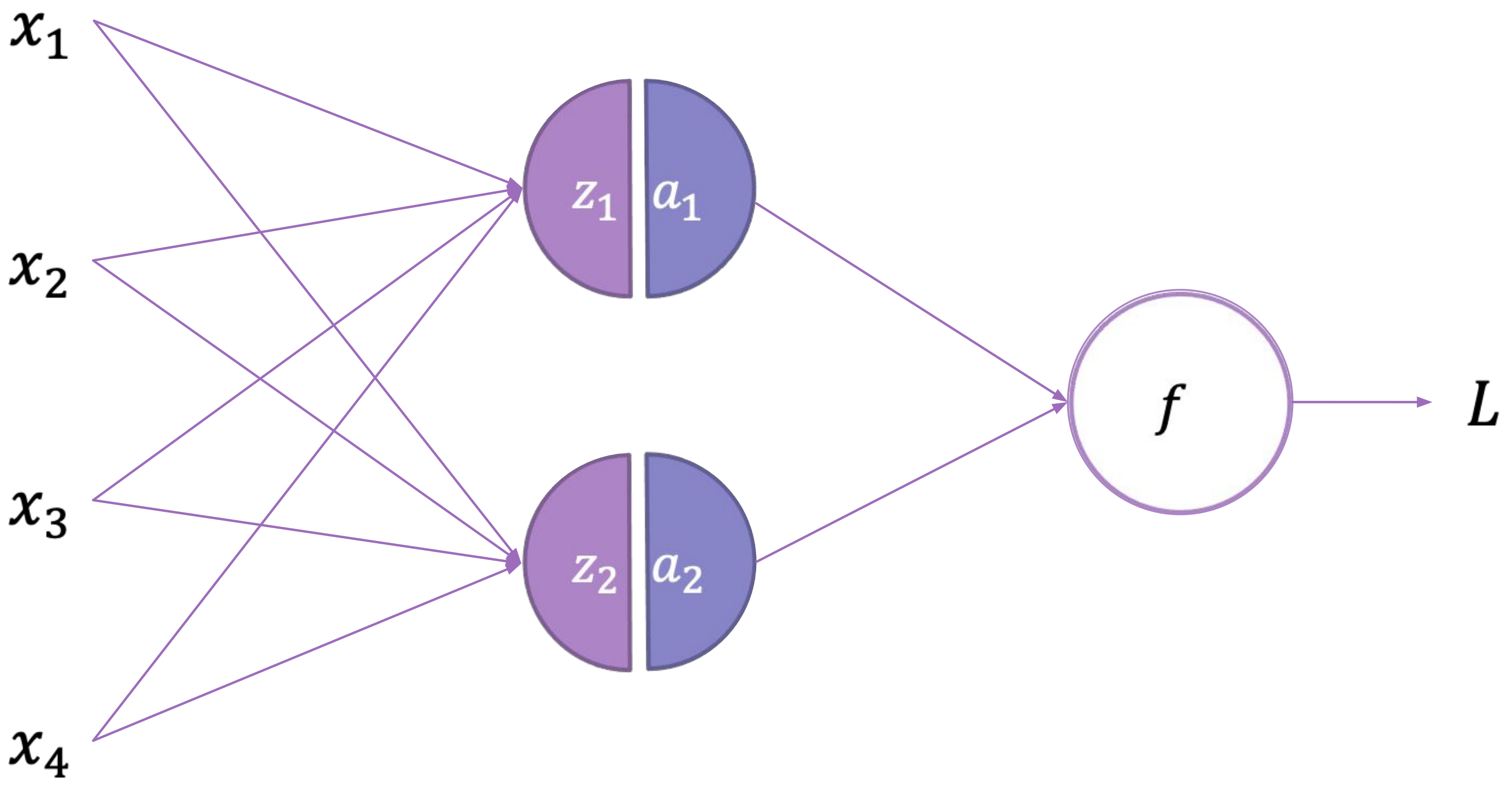
Refresher: Chain Rule

"Derivative of outside of inside equals derivative of outside times derivative of inside"

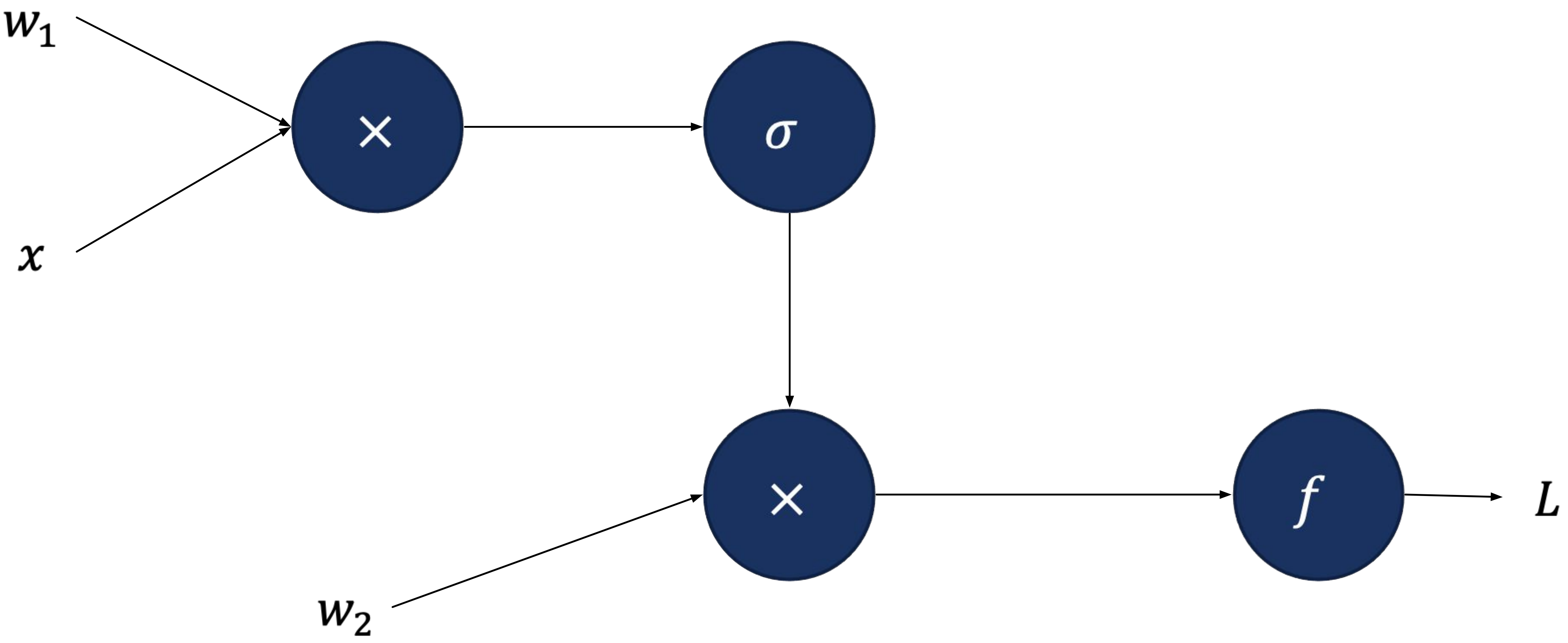
$$\frac{d}{dx}f\left(g(x)\right) = \frac{d}{dg(x)}f\left(g(x)\right) \times \frac{d}{dx}g(x)$$

-

2-Layer MLP



$\hat{y} = w_2\sigma(w_1 x)$ and $L = f(\hat{y})$



2-Layer MLP Equations and Gradients

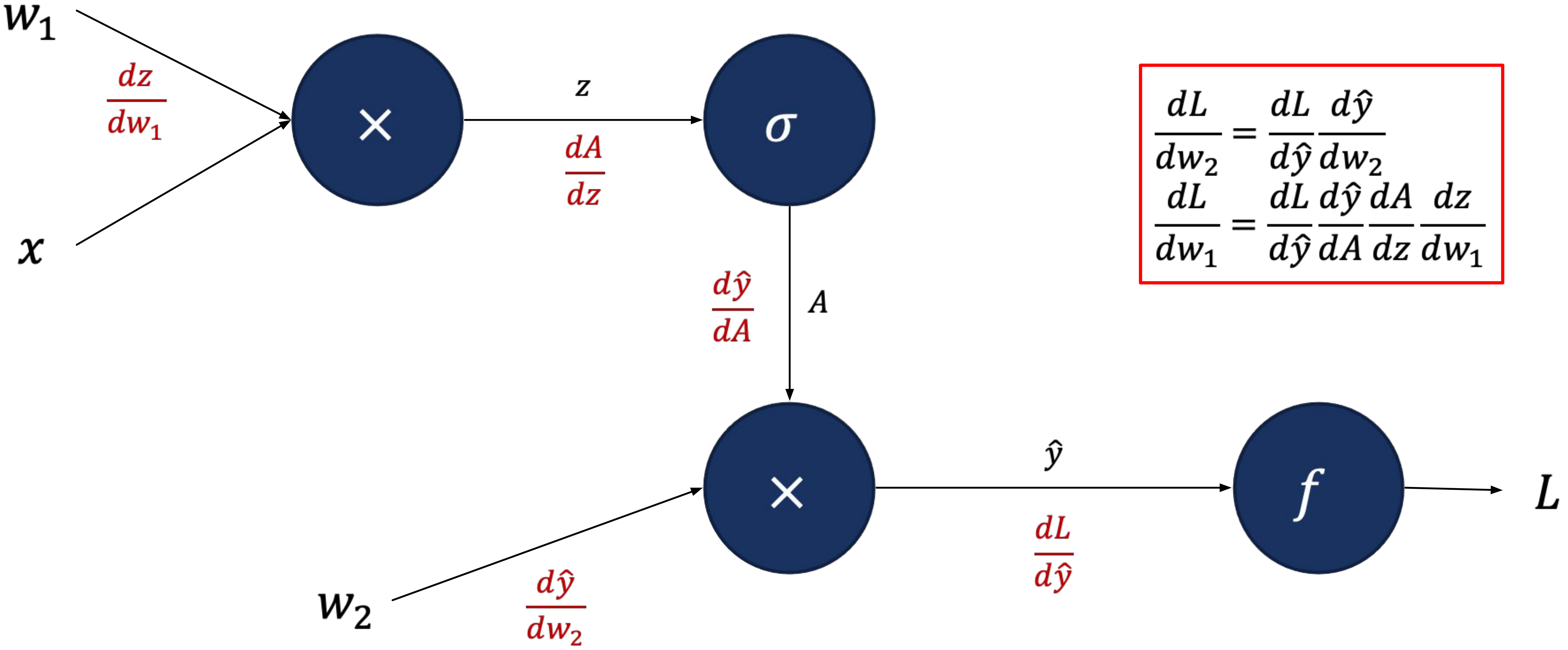
$L = f(\hat{y})$, where $\hat{y} = w_2 A$, $A = \sigma(z)$, and $z = w_1 x$

$$\frac{dL}{dw_1} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dw_1} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dA}\frac{dA}{dw_1} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dA}\frac{dA}{dz}\frac{dz}{dw_1}$$

$$\frac{dL}{dw_2} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dw_2}$$

Notice that this value is used in both gradient computations!

$\hat{y} = w_2\sigma(w_1 x)$ and $L = f(\hat{y})$



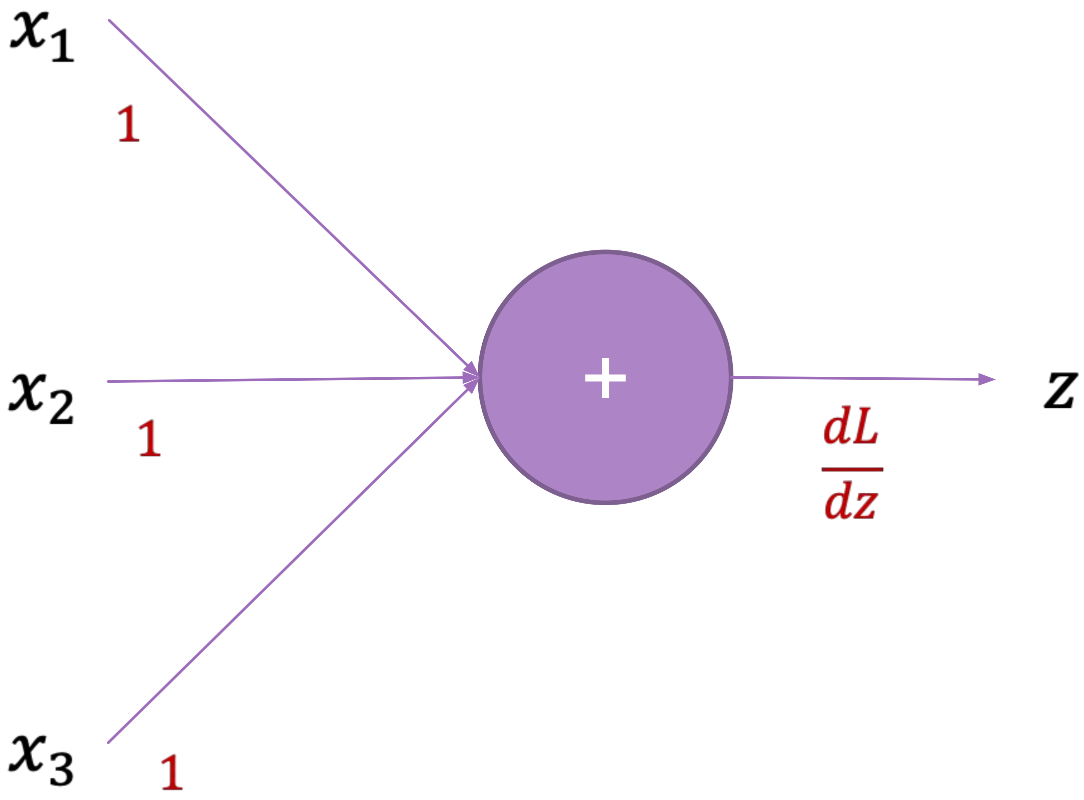
$$\frac{dL}{dw_2} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dw_2}$$
$$\frac{dL}{dw_1} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dA}\frac{dA}{dz}\frac{dz}{dw_1}$$

The Backpropagation Algorithm

- Treat entire network as a computational graph, each computation as a node
- We can independently compute local gradient at each node given node inputs
- Accumulate gradients from back (loss) to front (weights) using chain rule (simple multiplication!)

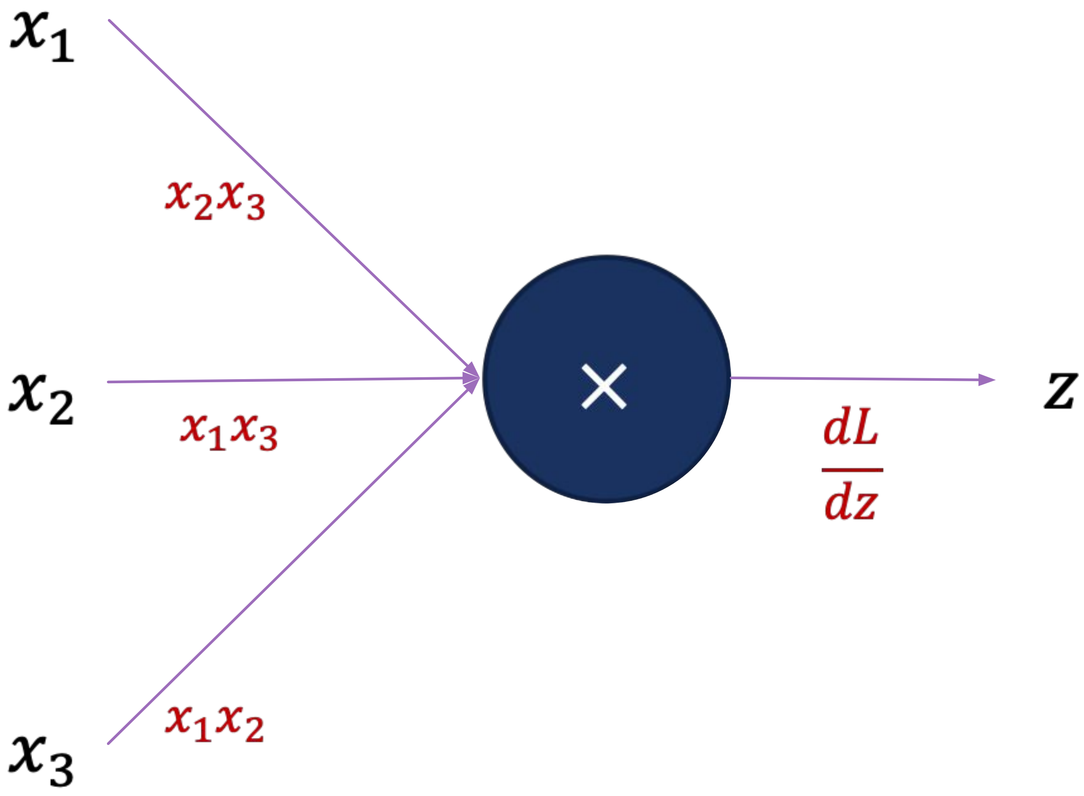
Summation

- $$z = \sum_i x_i \,, L = f(Z)$$
  $$\frac{\partial z}{\partial x_i} = 1$$
  $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z}$$



Multiplication

- $$z = \prod_i x_i \,, L = f(Z)$$
  $$\frac{\partial z}{\partial x_i} = \frac{z}{x_i}$$
  $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z}\frac{z}{x_i}$$

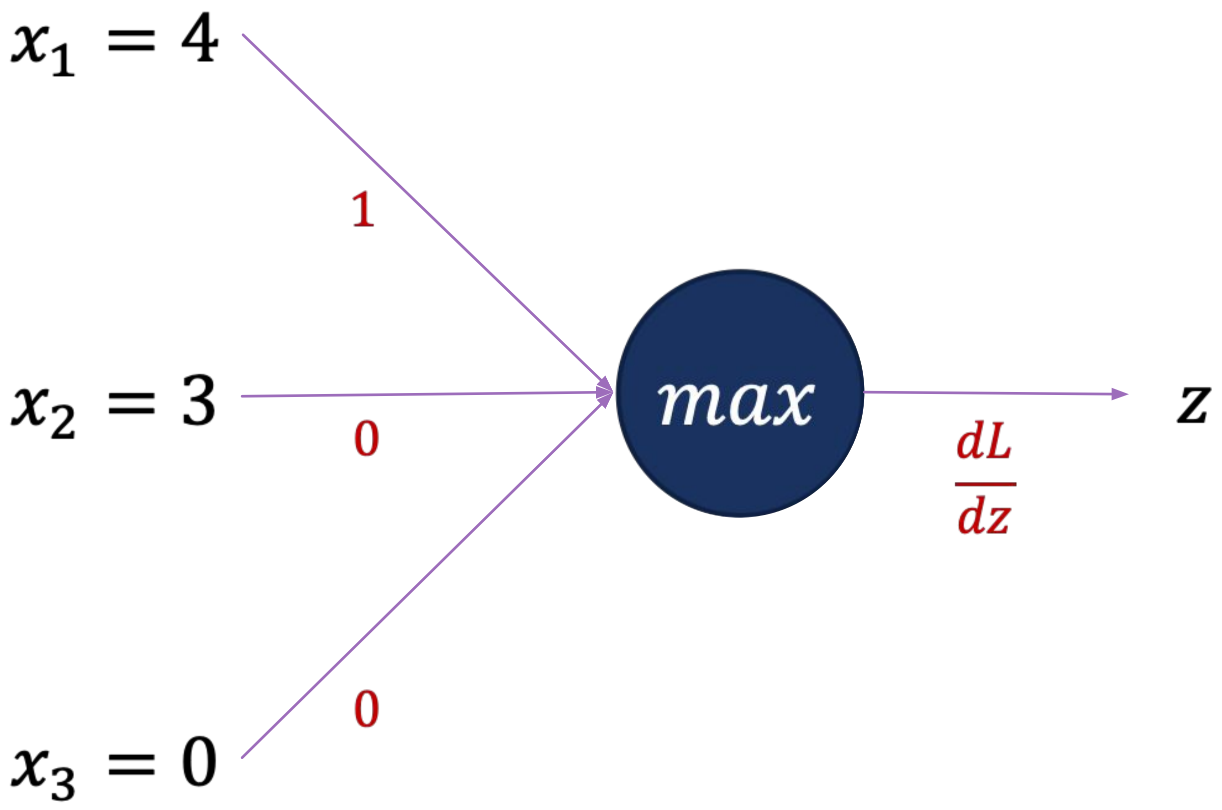


Min/Max

$$z = \max_i x_i \, , L = f(Z)$$

$$\frac{\partial z}{\partial x_i} = \mathbf{1}[x_i = z]$$

- $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z}\mathbf{1}[x_i = z]$$

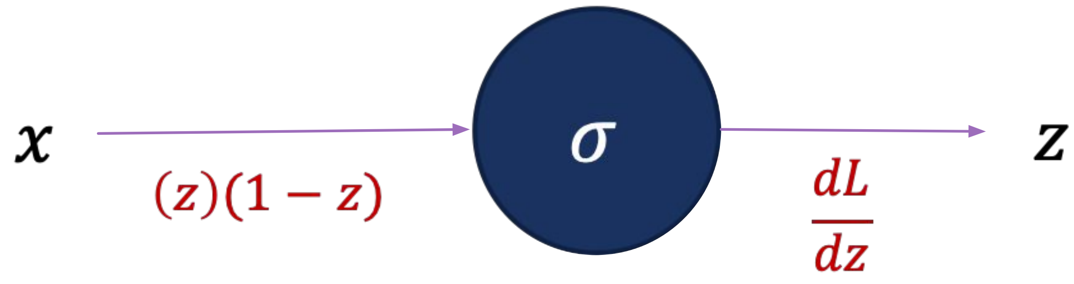


Sigmoid

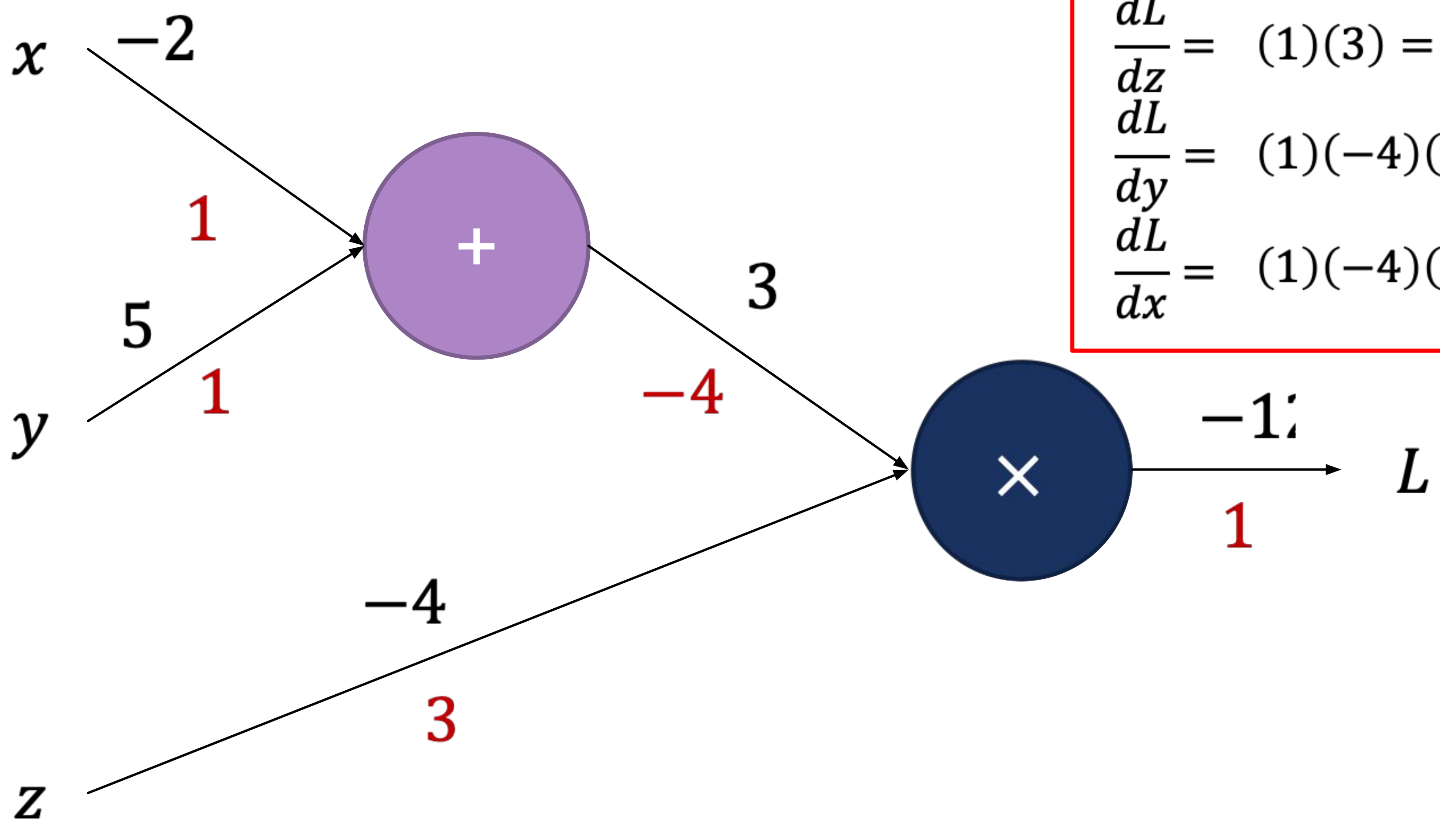
$$z = \sigma(x), L = f(Z)$$

$$\frac{\partial z}{\partial x_i} = z(1 - z)$$

- $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z} z(1 - z)$$



$$L = (x + y)z$$

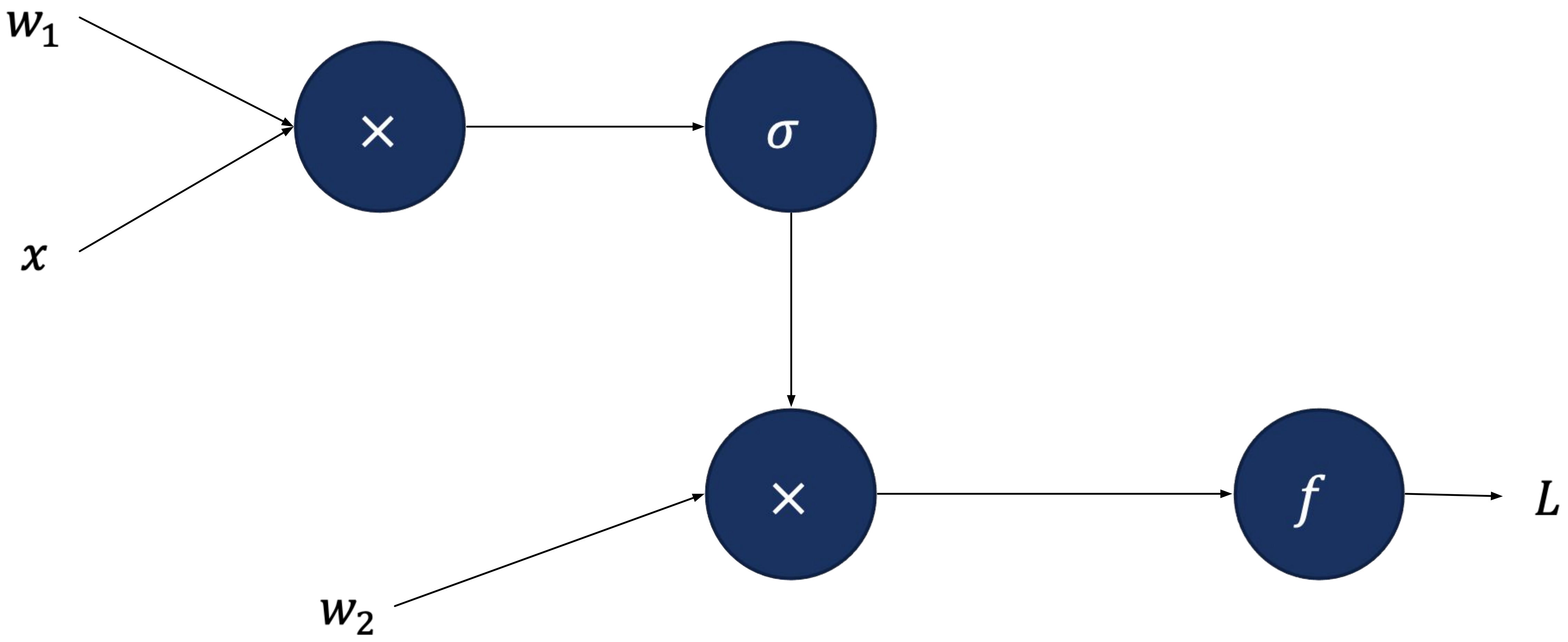


$$\frac{dL}{dz} = (1)(3) = 3$$

$$\frac{dL}{dy} = (1)(-4)(1) = -4$$

$$\frac{dL}{dx} = (1)(-4)(1) = -4$$

$\hat{y} = w_2\sigma(w_1x)$ and $L = f(\hat{y})$



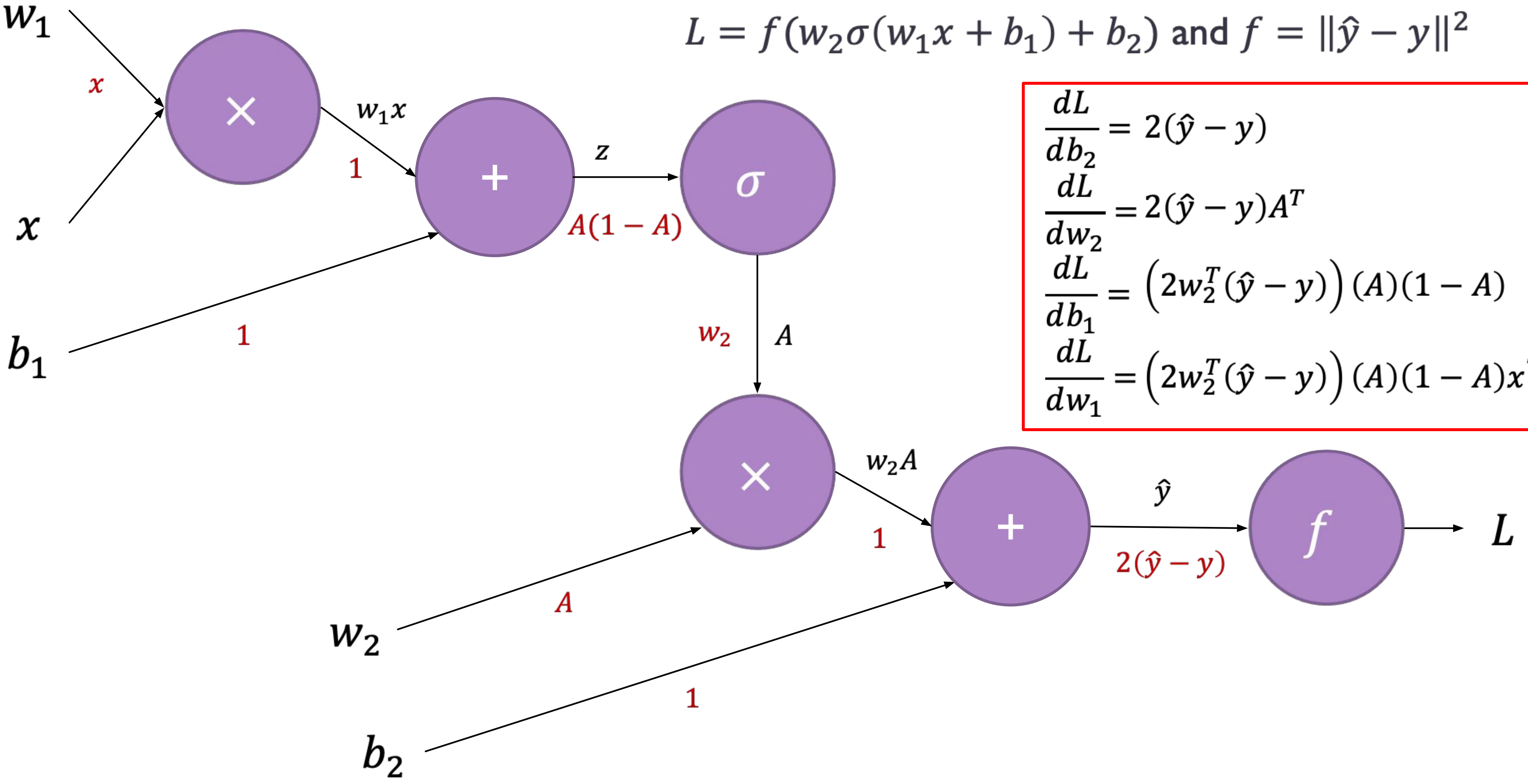
$$L = f(w_2\sigma(w_1x + b_1) + b_2) \text{ and } f = \|\hat{y} - y\|^2$$

$$\frac{dL}{db_2} = 2(\hat{y} - y)$$

$$\frac{dL}{dw_2} = 2(\hat{y} - y)A^T$$

$$\frac{dL}{db_1} = \left(2w_2^T(\hat{y} - y)\right)(A)(1 - A)$$

$$\frac{dL}{dw_1} = \left(2w_2^T(\hat{y} - y)\right)(A)(1 - A)x^T$$



Convolution-Based Classifier

Recall convolutions…

12	**3**	**19**
25	10	1
9	7	17

$*$

1	**2**
3	4

$=$

?	**?**
?	?

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Recall convolutions…

12	3	19
25	10	1
9	7	17

$*$

1	2
3	4

$=$

133	?
?	?

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Recall convolutions…

12	**3**	**19**
25	10	1
9	7	17

$*$

1	**2**
3	4

$=$

133	**75**
?	?

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Recall convolutions…

12	3	19
25	10	1
9	7	17

*

1	2
3	4

=

133	75
100	?

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Recall convolutions…

12	3	19
25	10	1
9	7	17

$*$

1	2
3	4

$=$

133	75
100	101

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Recall convolutions…

12	3	19
25	10	1
9	7	17

$*$

1	2
3	4

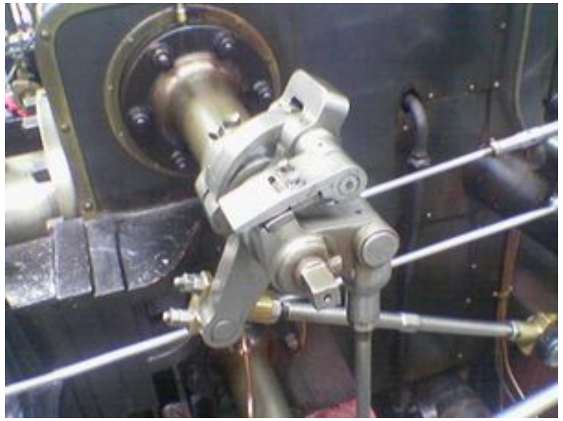
$=$

133	75
100	101

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Why they are useful

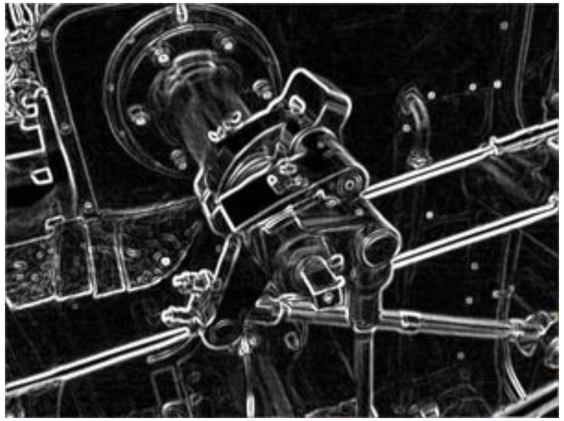
Allow us to find **interesting insights/features** from images!



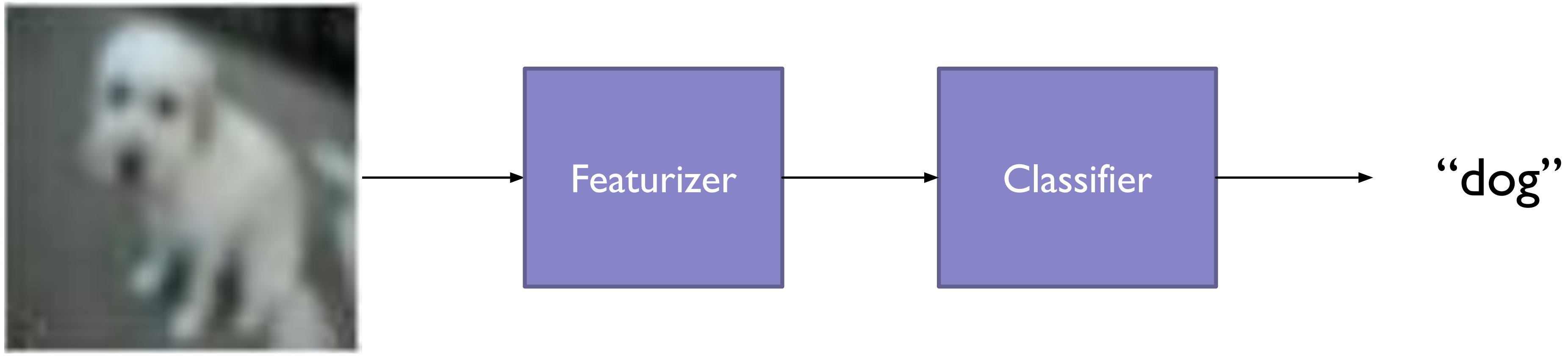
*

0	**-½**	**0**
0	0	0
0	½	0

=



Recall Image Classification…



Allow us to use features to put **images in categories**!

Convolution-Based Classification System

Convolution = Image -> Features

Classification Algorithm = Features -> Category

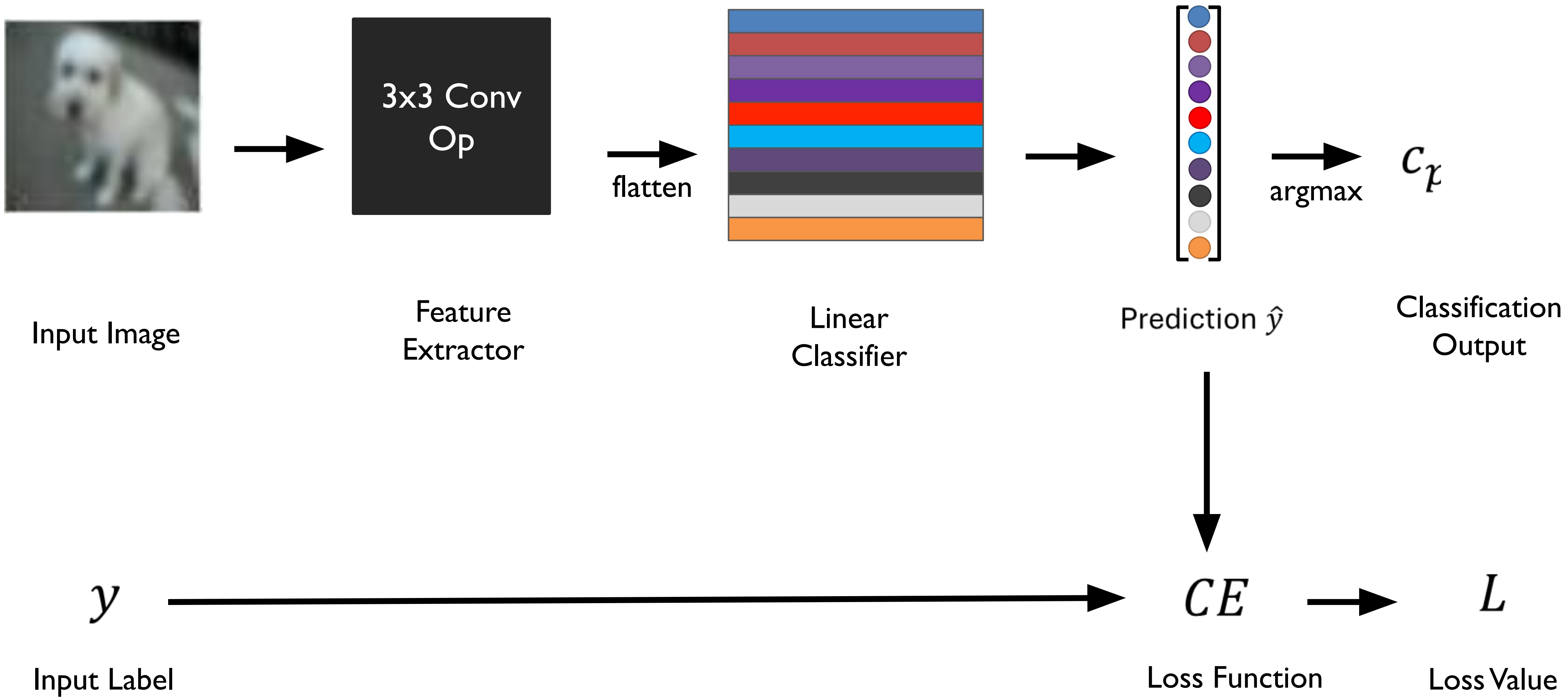
Convolution-Based Classification System

Convolution = Image -> Features

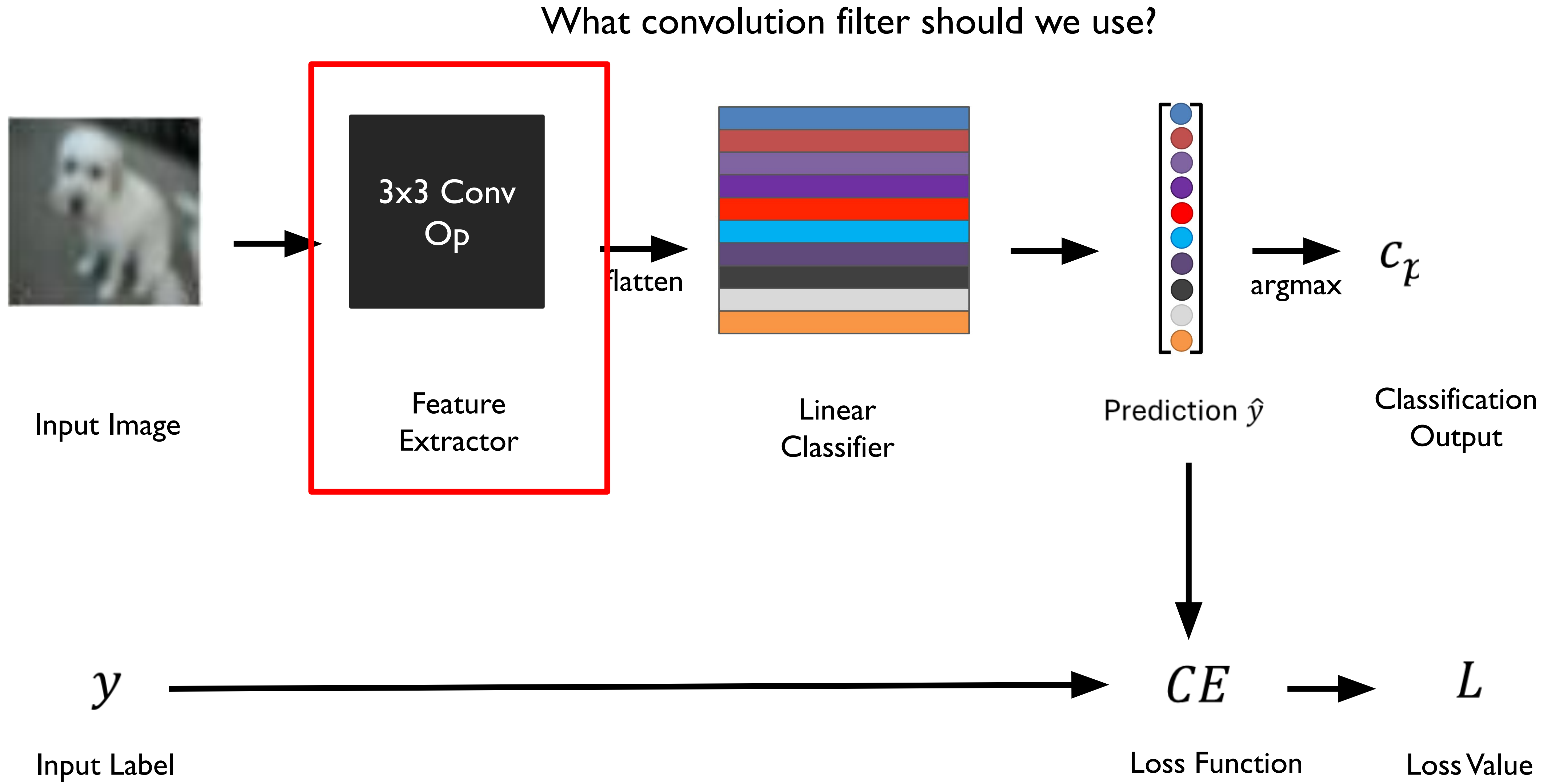
Classification Algorithm = Features -> Category

Can we put them together?

Convolution + Learned Linear Layer

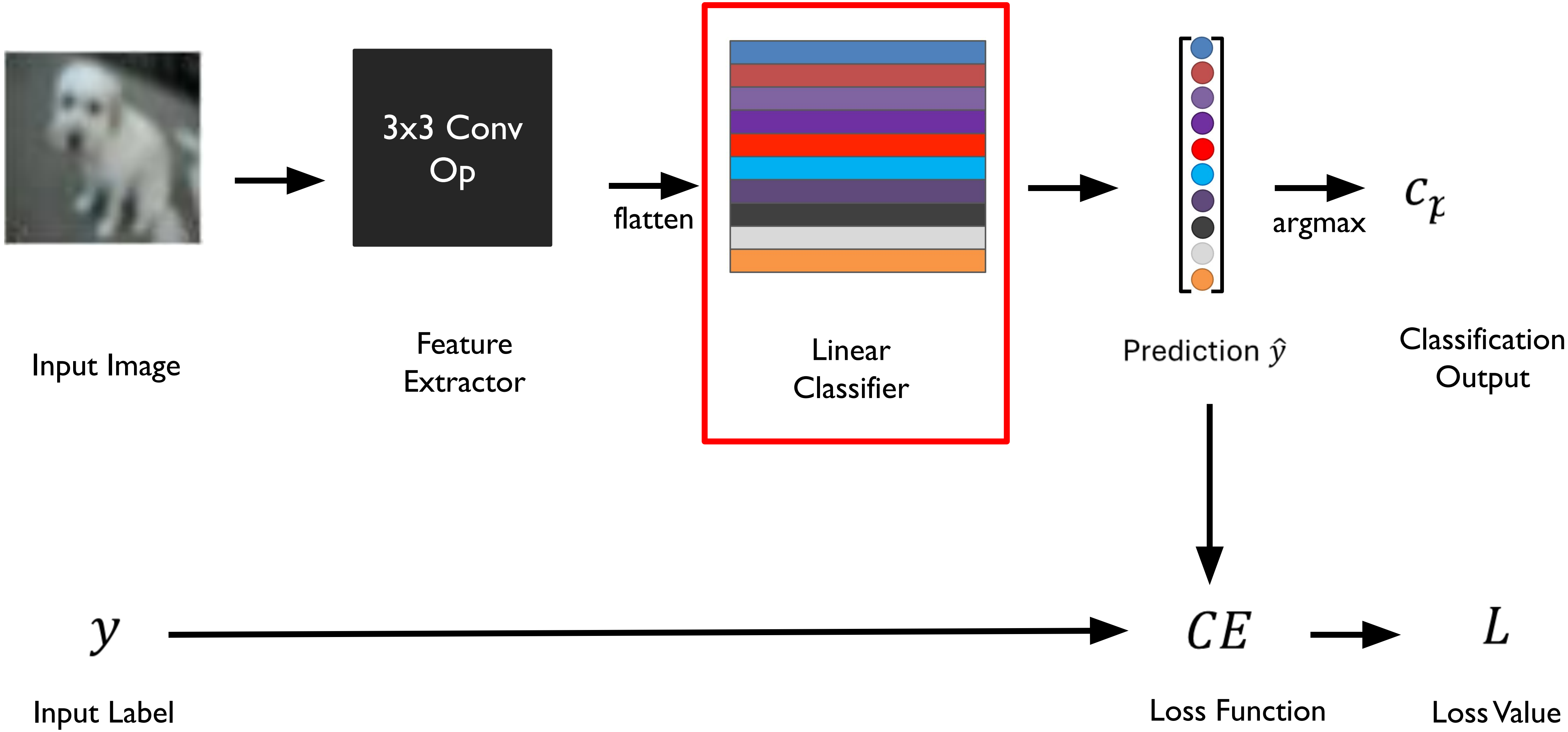


Convolution + Learned Linear Layer

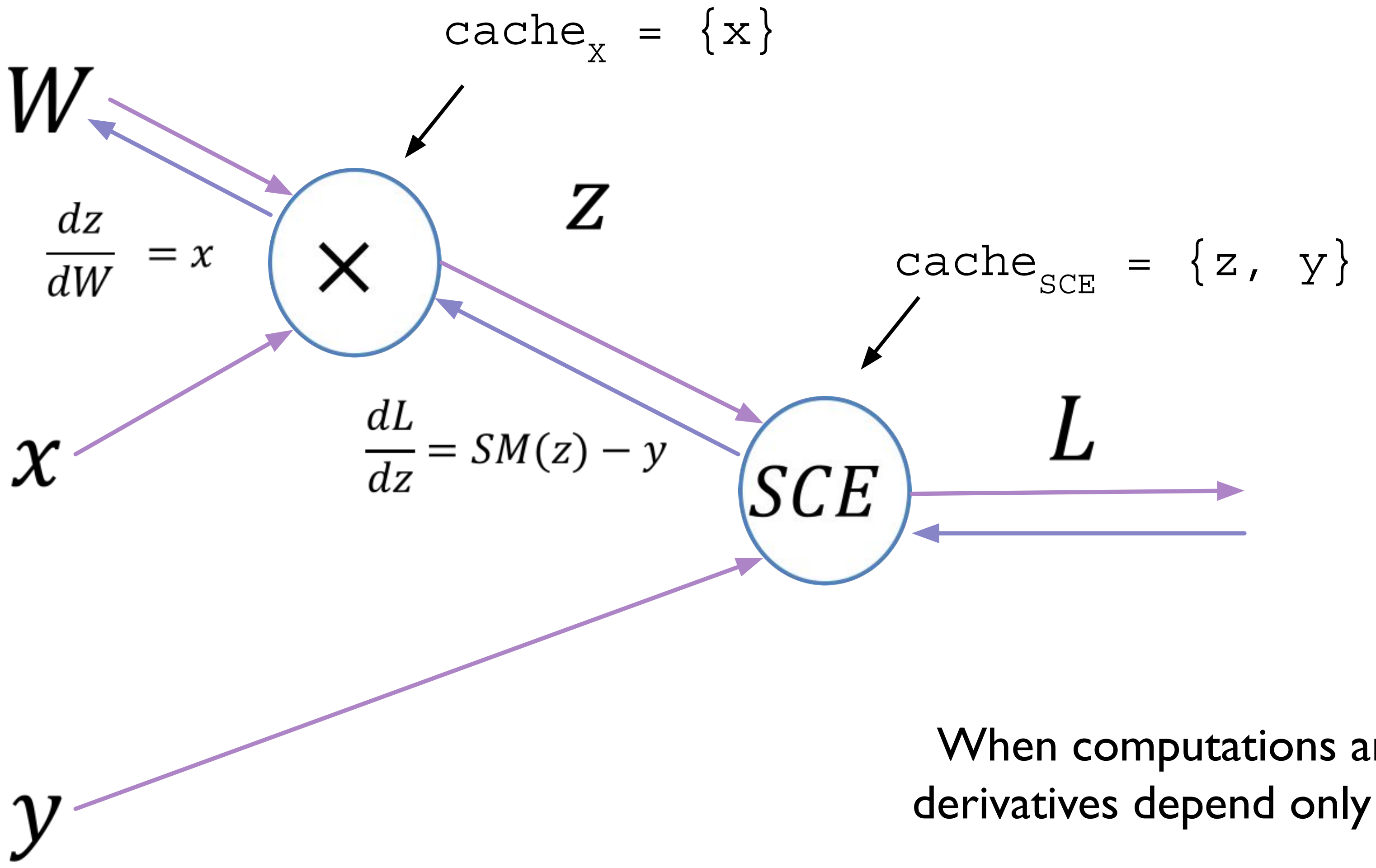


(Learned?) Convolution + Learned Linear Layer

Can we learn it like we learn the linear layer?



Calculating Local Gradients



$$z = Wx$$
$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

When computations are treated as nodes, all derivatives depend only on inputs to that node.

So, we can cache the initial computation and reuse!

Backprop Graph to Layer-Based Pseudocode

```
class Linear:
   def __init__(self):
      self.cache = {}

   def forward(self, W, x):
      self.cache['x'] = x

      return np.dot(W, x)

   def backward(self, dout):
      x = self.cache['x']

      return np.matmul(dout, x.T)
```

```
class SCELoss:
   def __init__(self):
      self.cache = {}

   def forward(self, z, y):
      self.cache['z'] = z
      self.cache['y'] = y

      return sce(z, y)

   def backward(self):
      z = self.cache['z']
      y = self.cache['y']

      return sm(z) - y
```

Model Definition + Training Pseudocode

```
model = Sequential([Linear, SCELoss])

for i in {0,…,num_epochs}:
    for X, y in data:
        L = model.forward(X, y)
            gradients = model.backprop(L)
            model.update_weights(gradients)
```

Model Definition + Training Pseudocode

How about a learnable conv layer right here?!

```
model = Sequential([Linear, SCELoss])

for i in {0,…,num_epochs}:
    for X, y in data:
        L = model.forward(X)
            gradients = model.backprop(L)
            model.update_weights(gradients)
```

It's Just a Linear Layer in a Double For-Loop!

12	**3**	**19**
25	10	1
9	7	17

$*$

1	**2**
3	4

$=$

133	**75**
100	101

$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l]\, h[n-k, m-l]$$

Conv Layer Pseudocode

```
class Conv:
   def __init__(self):
      self.cache = {}

   def forward(self, W, x):
      self.cache['x'] = x
      out = np.zeroes(out_shape)
      for i in range(out.shape[0]):
         for j in range(out.shape[1]):
            out[i, j] = np.dot(W, x[i-3:i+3, j-3:j+3])

   def backward(self, dout):
      x = self.cache['x']
      d = np.zeros(W_shape)

      for i in range(dout.shape[0]):
         for j in range(dout.shape[1]):
            d += dout[i, j] * x[i-3:i+3, j-3:j+3]
```

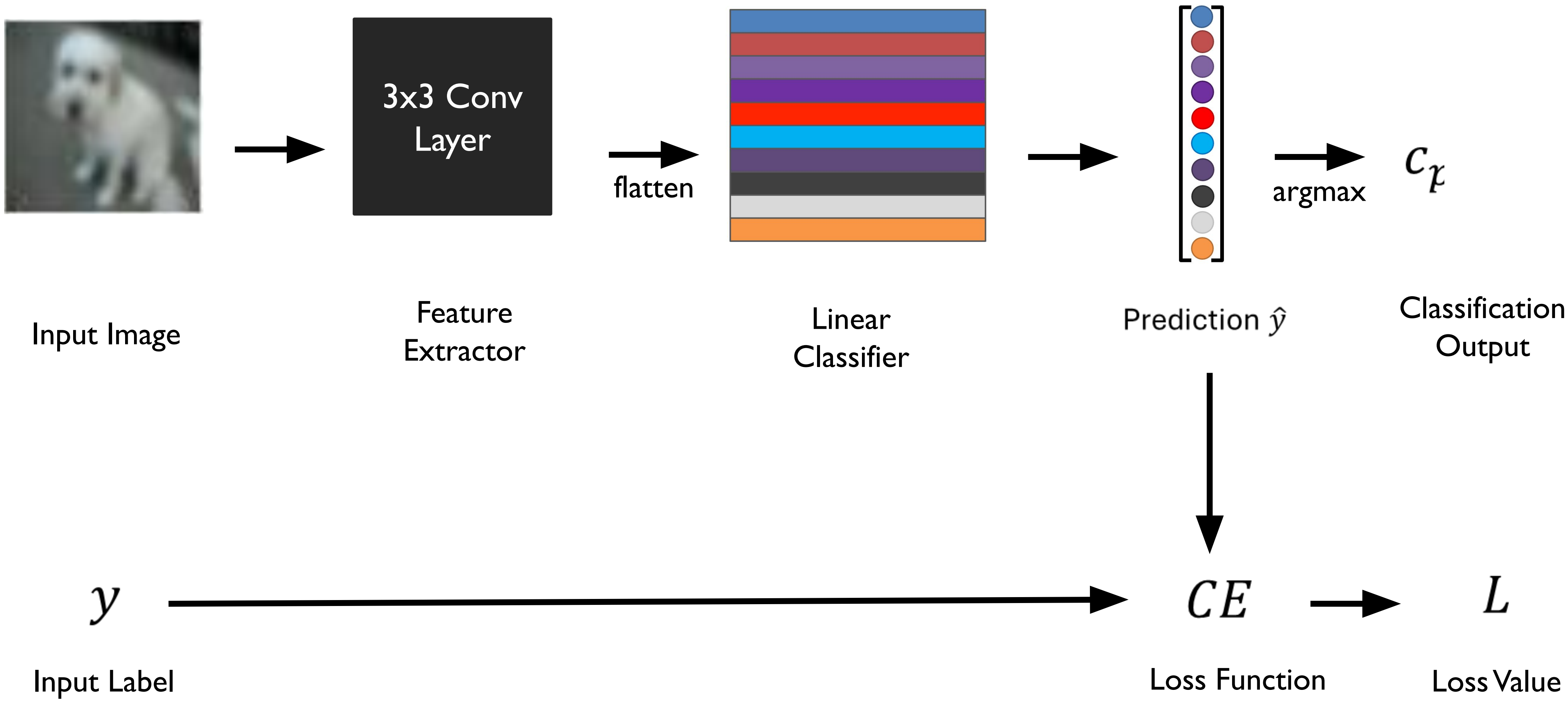
Adding Conv Layer to Model Definition

```
layers = [Linear, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```

Adding Conv Layer to Model Definition

```
layers = [Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
            gradients = model.backprop(L)
            model.update_weights(gradients)
```

Learned Convolution + Learned Linear Layer



Deep Convolutional Networks

Stacking Conv Layers in Model Definition

Why just one convolutional layer?

```
layers = [Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
            gradients = model.backprop(L)
            model.update_weights(gradients)
```

Stacking Conv Layers in Model Definition

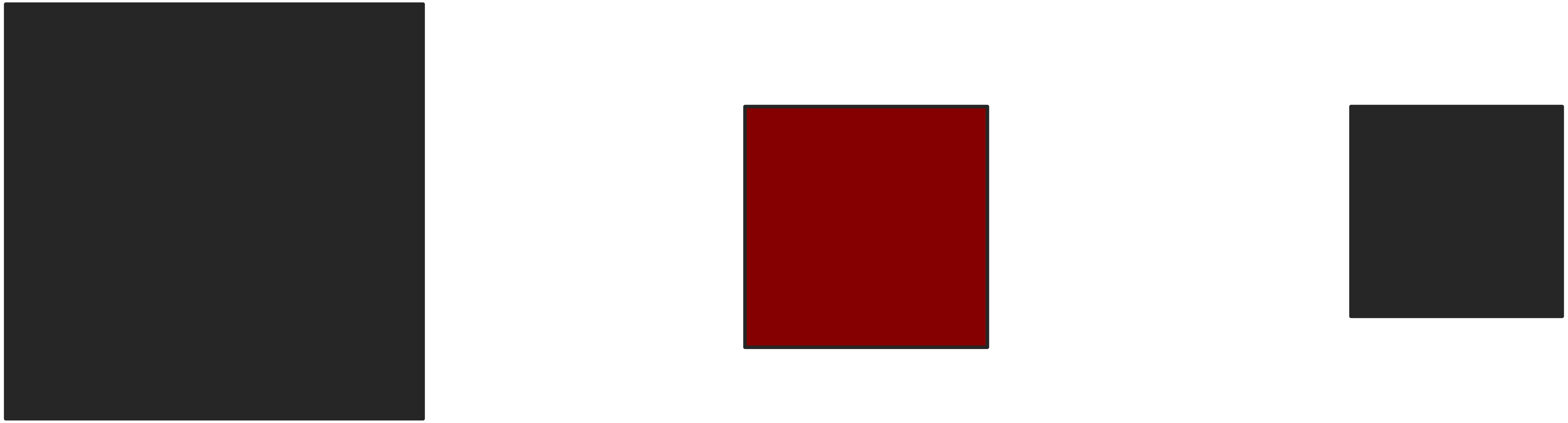
```
layers = [Conv, Conv, Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
            gradients = model.backprop(L)
            model.update_weights(gradients)
```

Stacking Conv Layers in Model Definition

And how much control do we have over what happens in each conv layer?

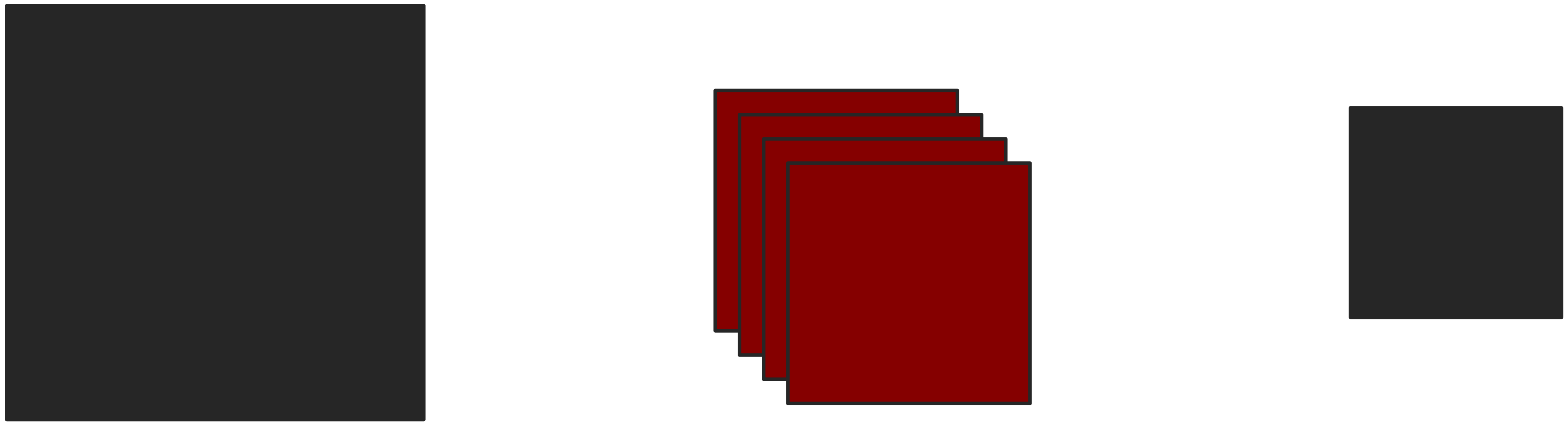
```
layers = [Conv, Conv, Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
            gradients = model.backprop(L)
            model.update_weights(gradients)
```

Right Now: One Filter Per Conv Layer



28×28×3 Image   15×15×3 Filter   14×14×1 Output

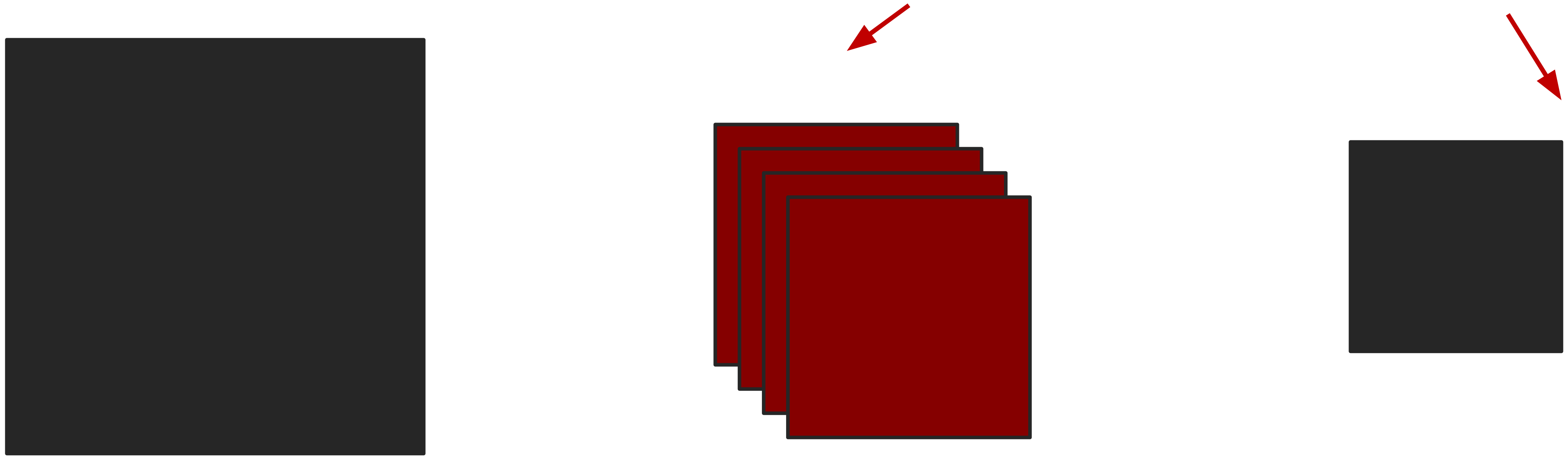
New Idea: Multiple Filters Per Conv Layer!



28×28×3 Image | 15×15×3×4 Filter | 14×14×4 Output

New Idea: Multiple Filters Per Conv Layer!

more output channels
= more filters
= more features we can learn!

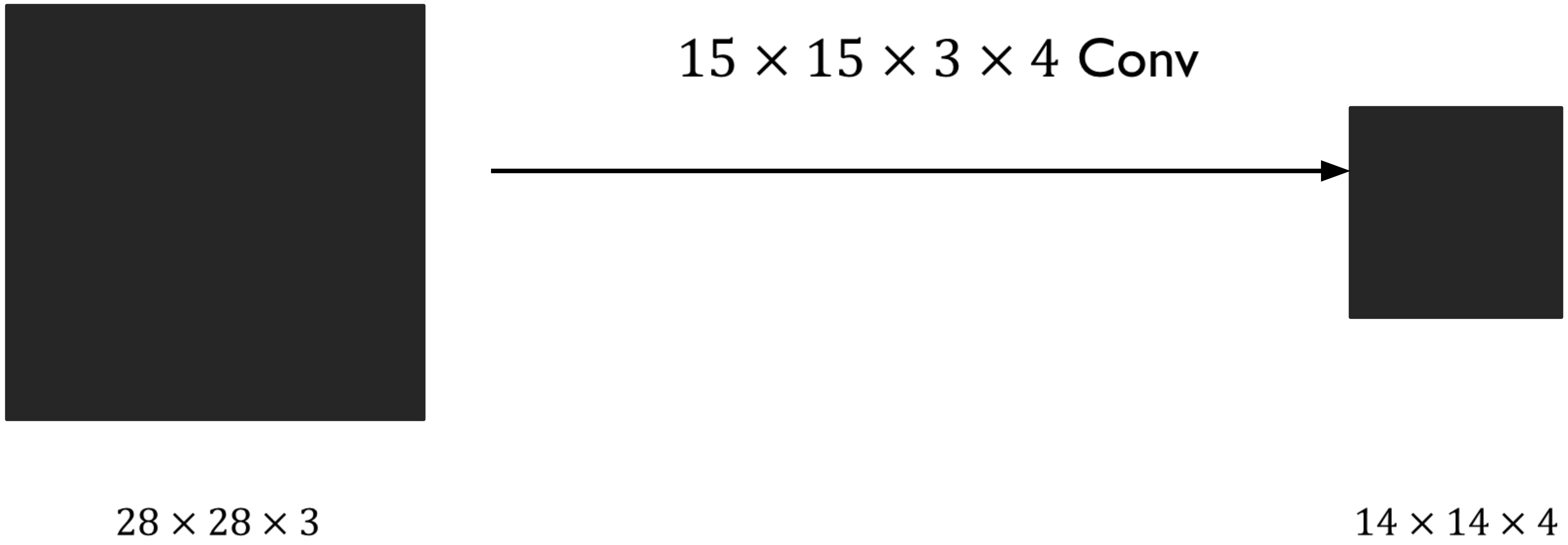


28×28×3 Image

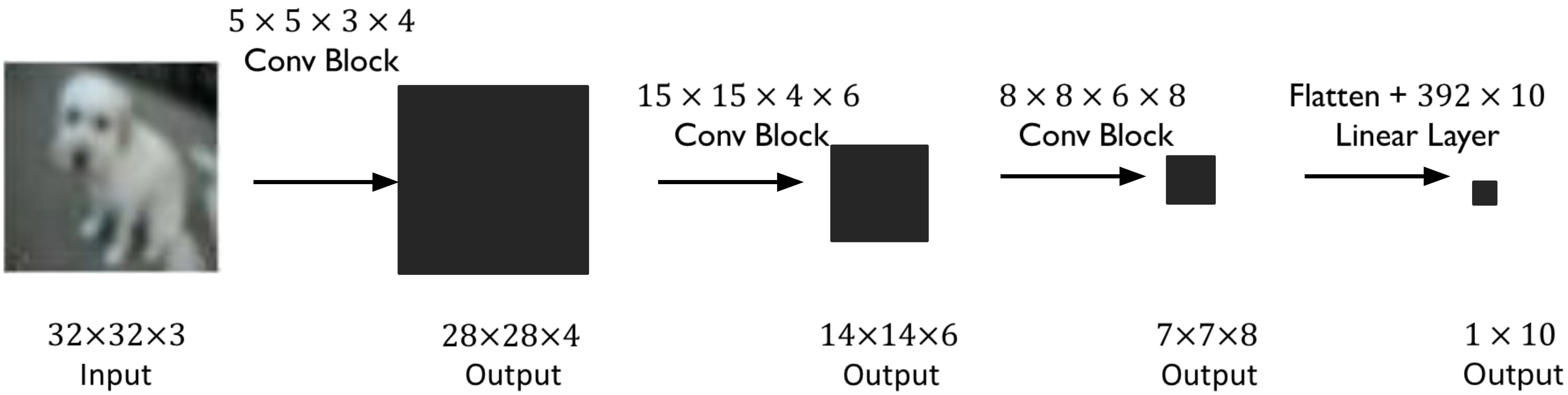
15×15×3×4 Filter

14×14×4 Output

Simplified Notation



Stacking Wide Convolutions



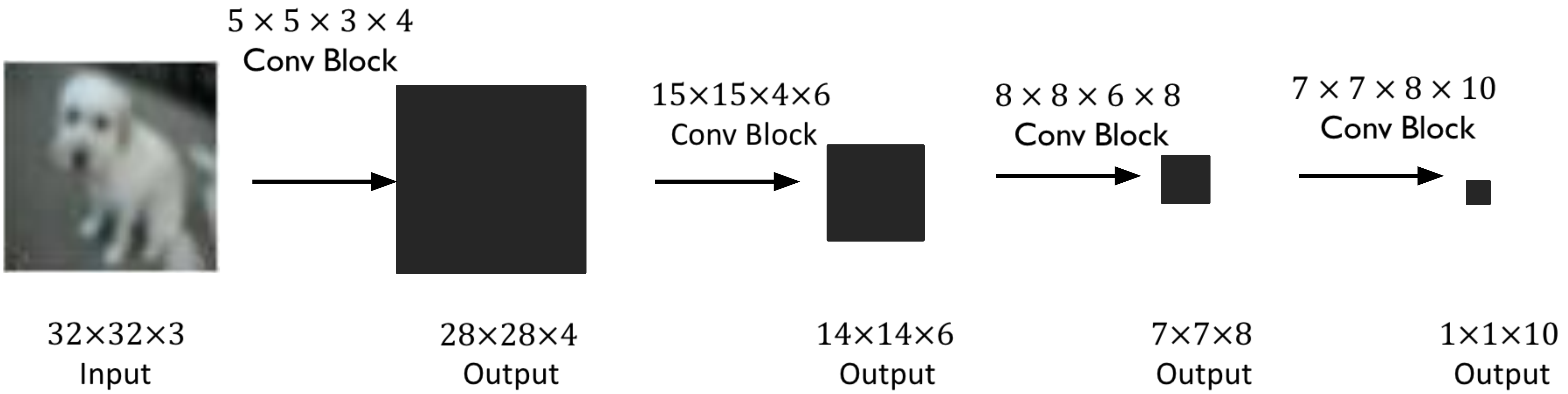
Multiple Filters in Conv Layer Pseudocode

```
class Conv:
   def __init__(self):
      self.cache = {}

   def forward(self, W, x):
      self.cache['x'] = x
      out = np.zeroes(out_shape)
      for i in range(out.shape[0]):
         for j in range(out.shape[1]):
            out[i, j] = np.dot(W, x[i-3:i+3, j-3:j+3])

   def backward(self, dout):
      x = self.cache['x']
      d = np.zeros(W_shape)

      for i in range(dout.shape[0]):
         for j in range(dout.shape[1]):
            d += dout[i, j] * x[i-3:i+3, j-3:j+3]
```

Multiple Filters in Conv Layer Pseudocode

```
class Conv:
   def __init__(self):
      self.cache = {}

   def forward(self, W, x):
      self.cache['x'] = x
      out = np.zeroes(out_shape)
      for i in range(out.shape[0]):
         for j in range(out.shape[1]):
            for f in range(out.shape[2]):
               out[i, j, f] = np.dot(W[f], x[i-3:i+3, j-3:j+3])

   def backward(self, dout):
      x = self.cache['x']
      d = np.zeros(W_shape)

      for i in range(dout.shape[0]):
         for j in range(dout.shape[1]):
            for f in range(dout.shape[2]):
               d[f] += dout[i, j, f] * x[i-3:i+3, j-3:j+3]
```

Fully-Convolutional Neural Network



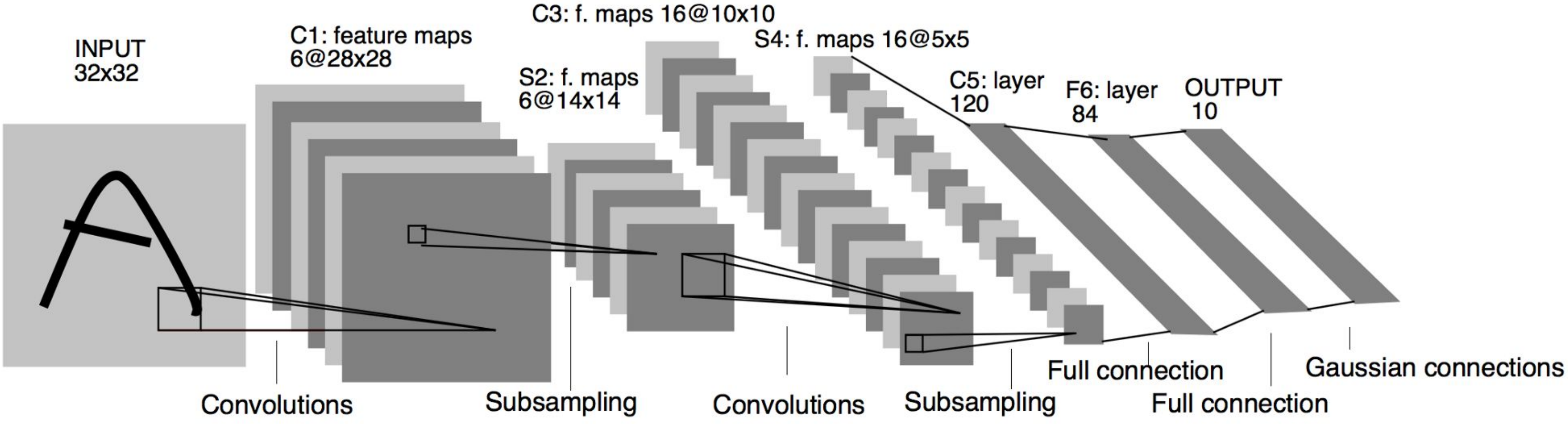
Fully-Convolutional Neural Network

```
layers = [Conv, Conv, Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
                gradients = model.backprop(L)
                model.update_weights(gradients)
```

Fully-Convolutional Neural Network

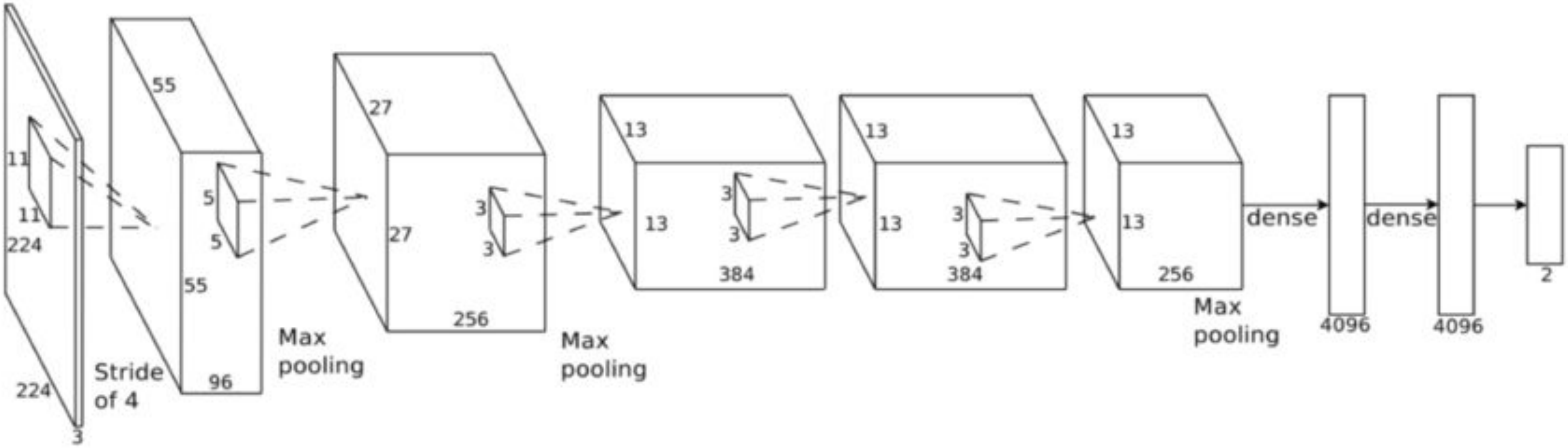
```
layers = [Conv, Conv, Conv, Conv, SCELoss]
model = Sequential(layers)

for i in {0,…,num_epochs}:
    for x, y in data:
        L = model.forward(X)
                gradients = model.backprop(L)
                model.update_weights(gradients)
```

History of ConvNets



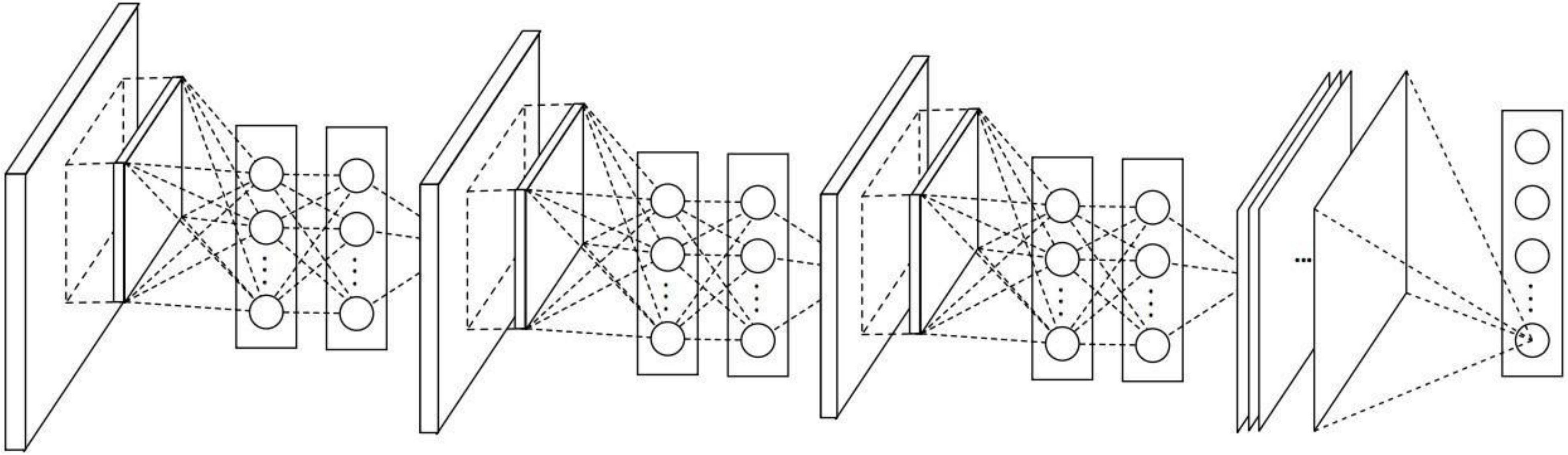
LeNet – 1998

History of ConvNets



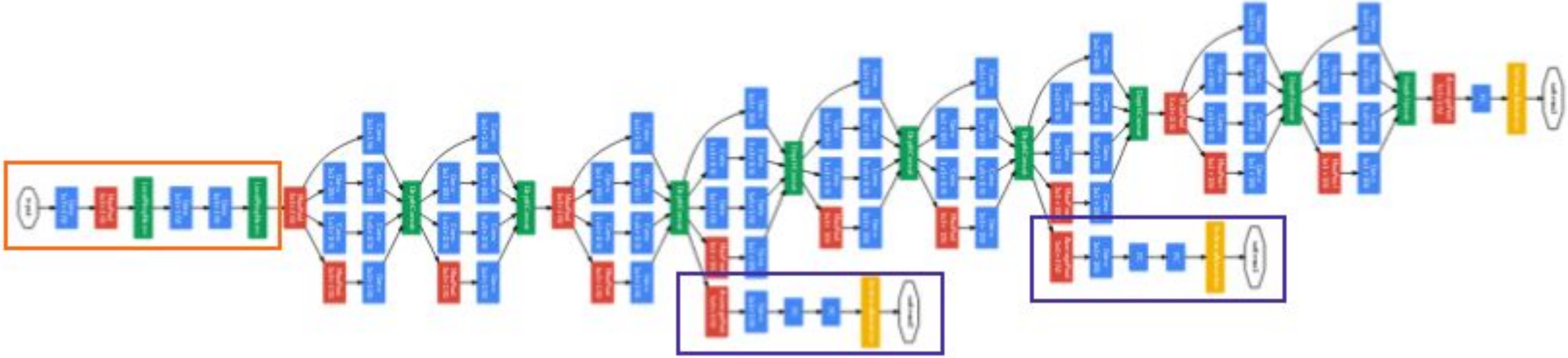
AlexNet – 2012

History of ConvNets



NiN – 2013

History of ConvNets



Inception Network – 2015

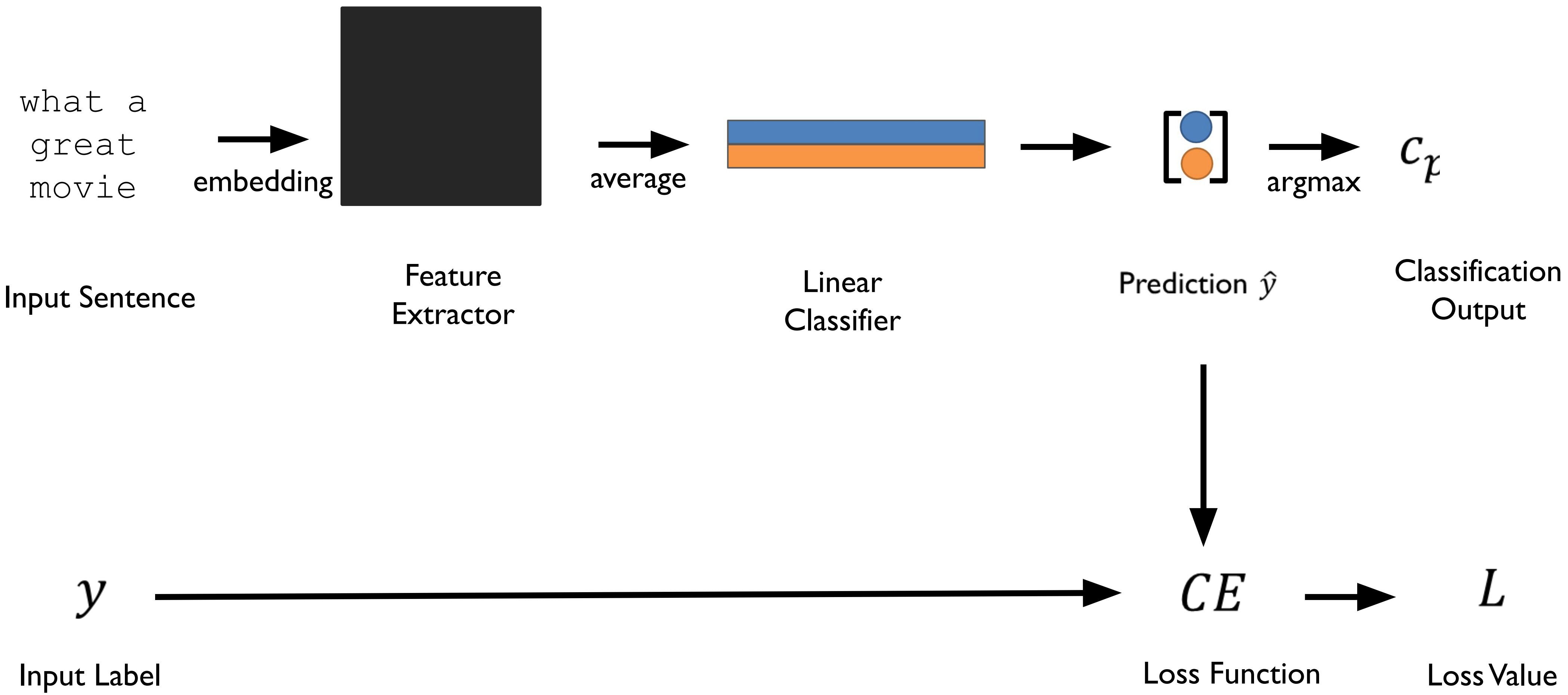
Vision Transformers

Self-Attention: An Alternative to Convolution

- Until now, we have been using convolution as the operation to gain deeper understanding of images
  - Good theoretical grounding: inspired by filters in classical CV
  - Efficient, stackable, and maintains spatial relationships

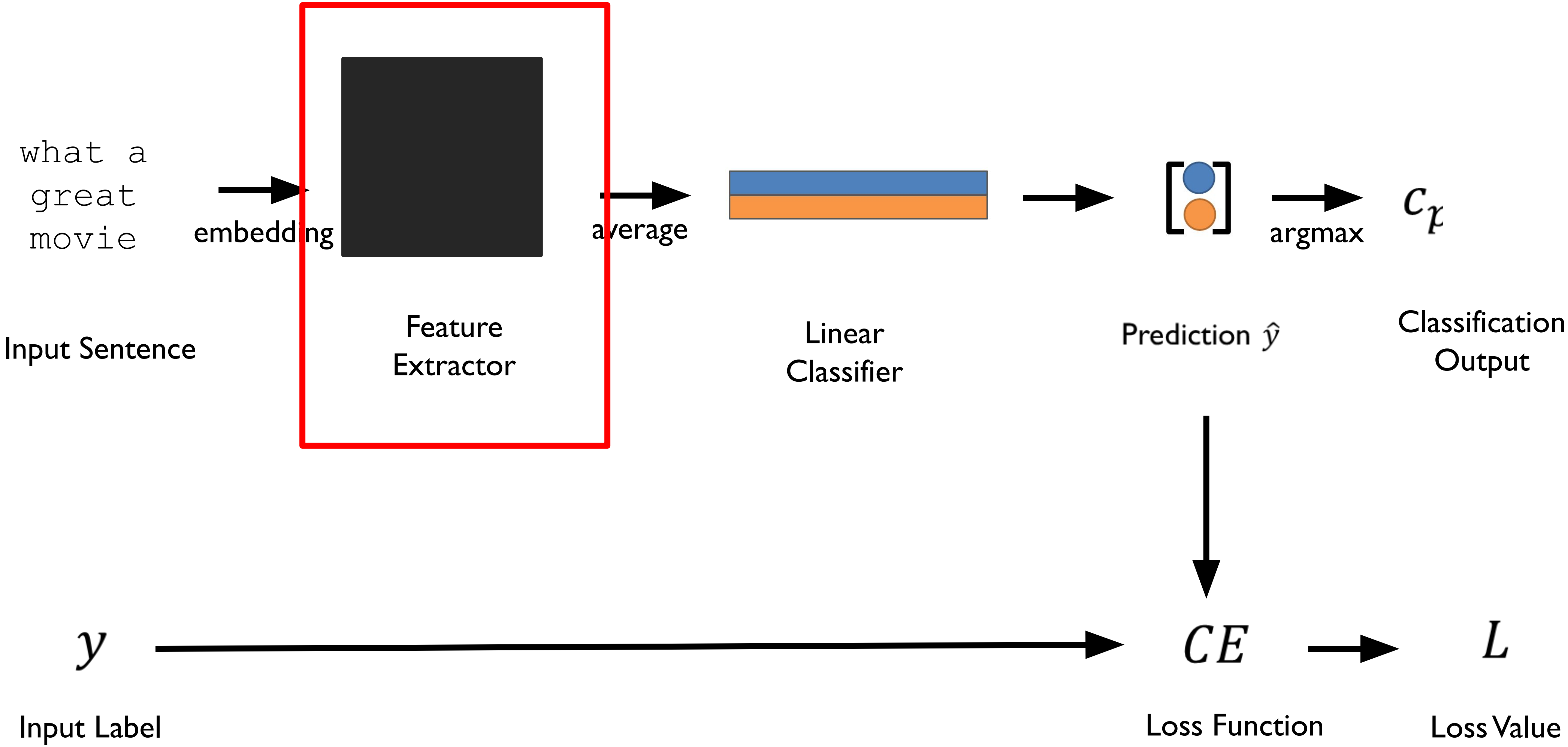
- Another operation which can efficiently learn spatial features is called self-attention
  - First applied to NLP tasks, eventually applied to CV
  - At each layer, we learn alignments between image patches i.e. how “relevant” is one patch to another
  - We can stack self-attn as as featurizer for arbitrary downstream CV tasks

Self-Attention Background: Sentiment Classifier

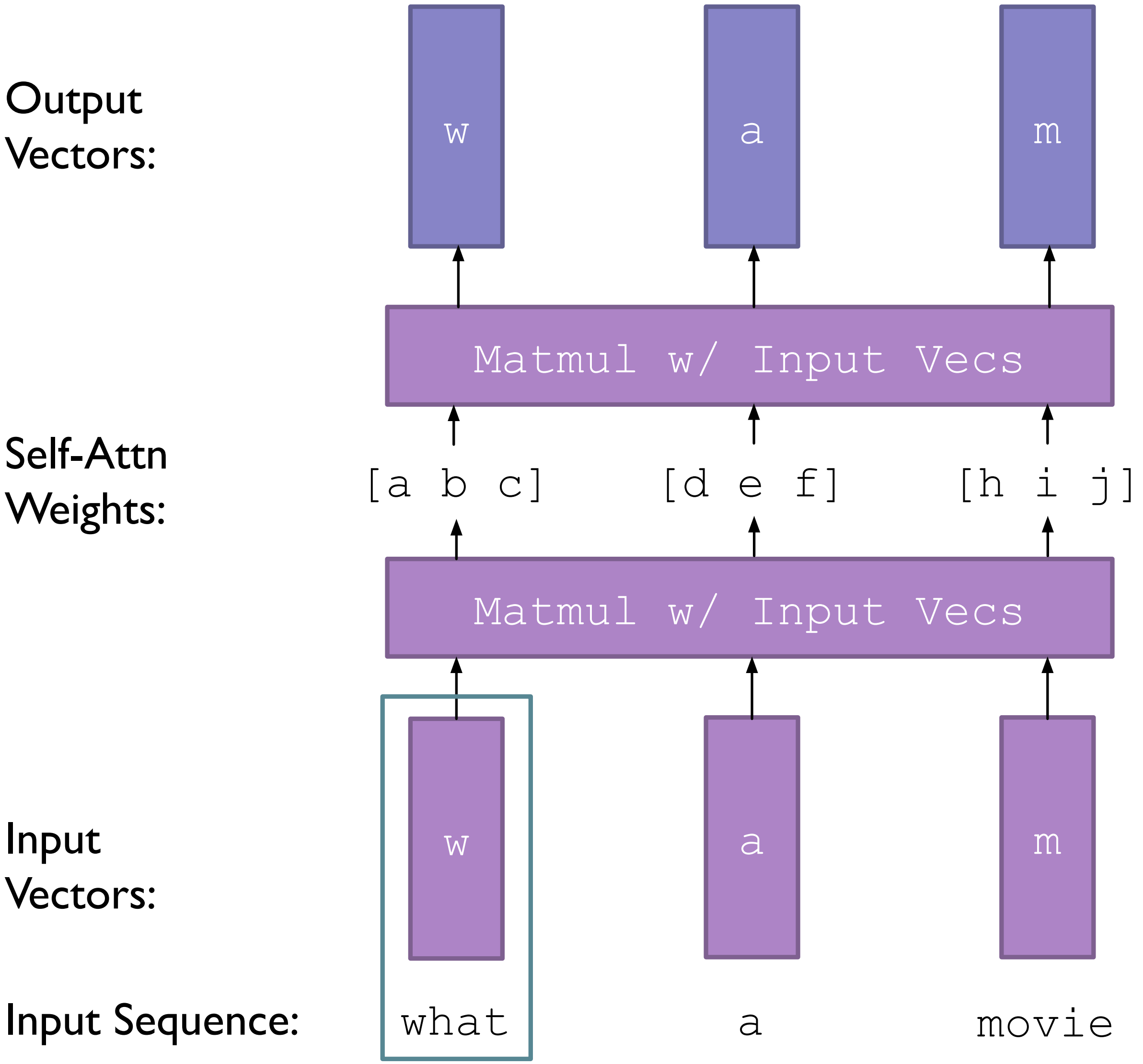


Self-Attention Background: Sentiment Classifier

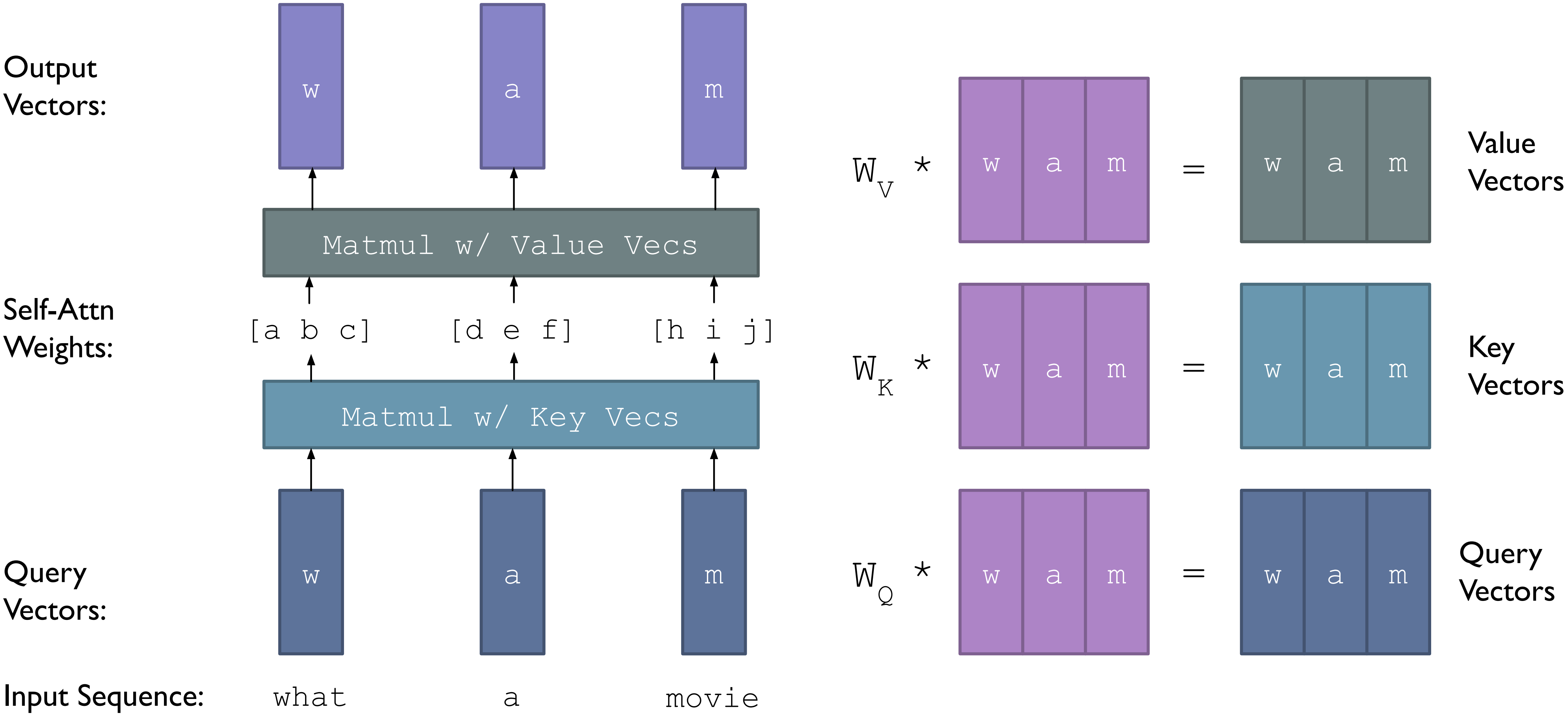
What (learnable) operation should we put here?



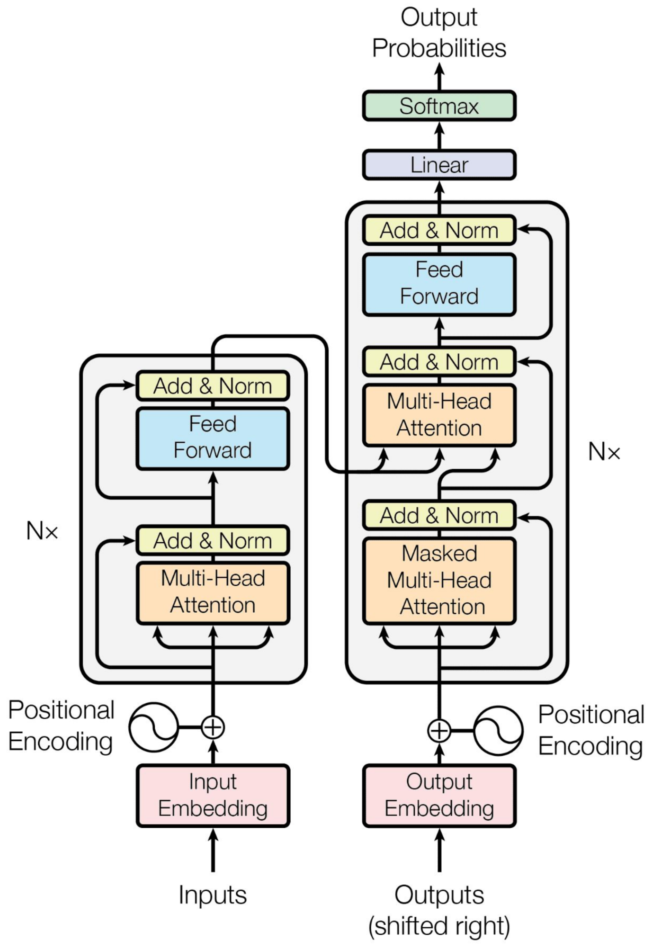
Visual Intuition of Self-Attention



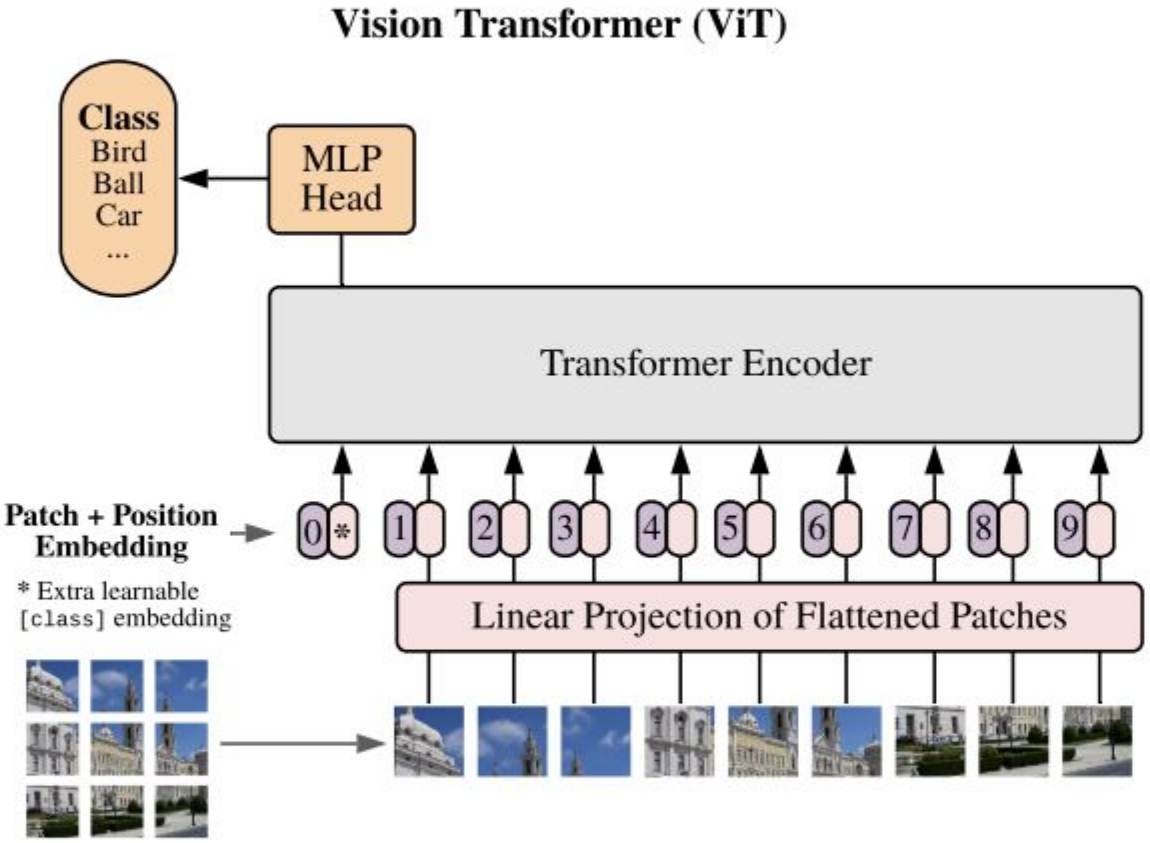
Visual Intuition of Self-Attention (Reality)



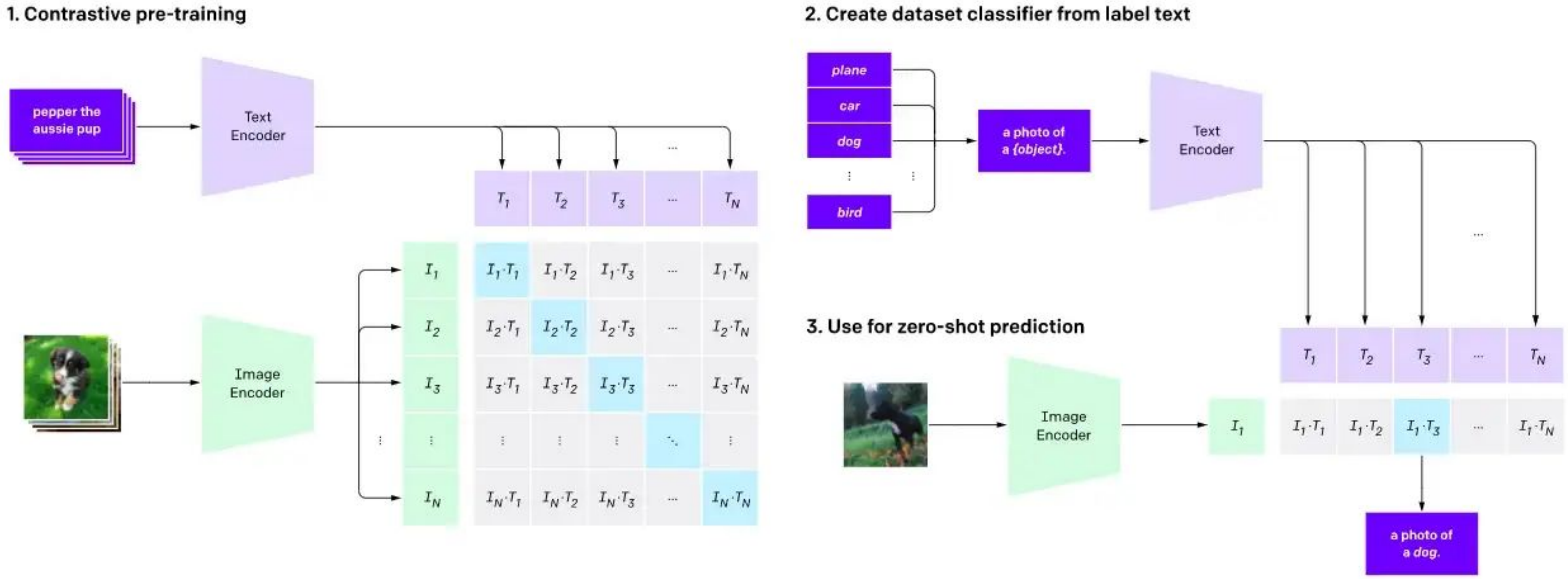
Transformer: Stack of Self-Attn (+more) Layers



Vision Transformer: Self-Attn on Img Patches



CLIP: Learning Multi-Model Embedding Space



Summary

- The popular method to do so in deep learning is backpropagation algorithm
  - Greedily use chain rule to accumulate local gradients from "back to front"
- We can use this principle to create computation "nodes" (layers), including a learnable convolutional layer
  - Foundational idea: create a classifier with a filter-based featurizer
- Convolution is not the only operation in our playbook: we can use an operation borrowed from NLP called self-attention to build a Transformer