

AUTODIFF & DESIGNING NNS

TODAY'S CLASS

GOALS

1. Autodifferentiation
2. Designing neural networks for specific problems

BACKPROP

BACKWARD PASS

Using the results of the forward pass, apply the chain rule to compute the derivatives for each layer

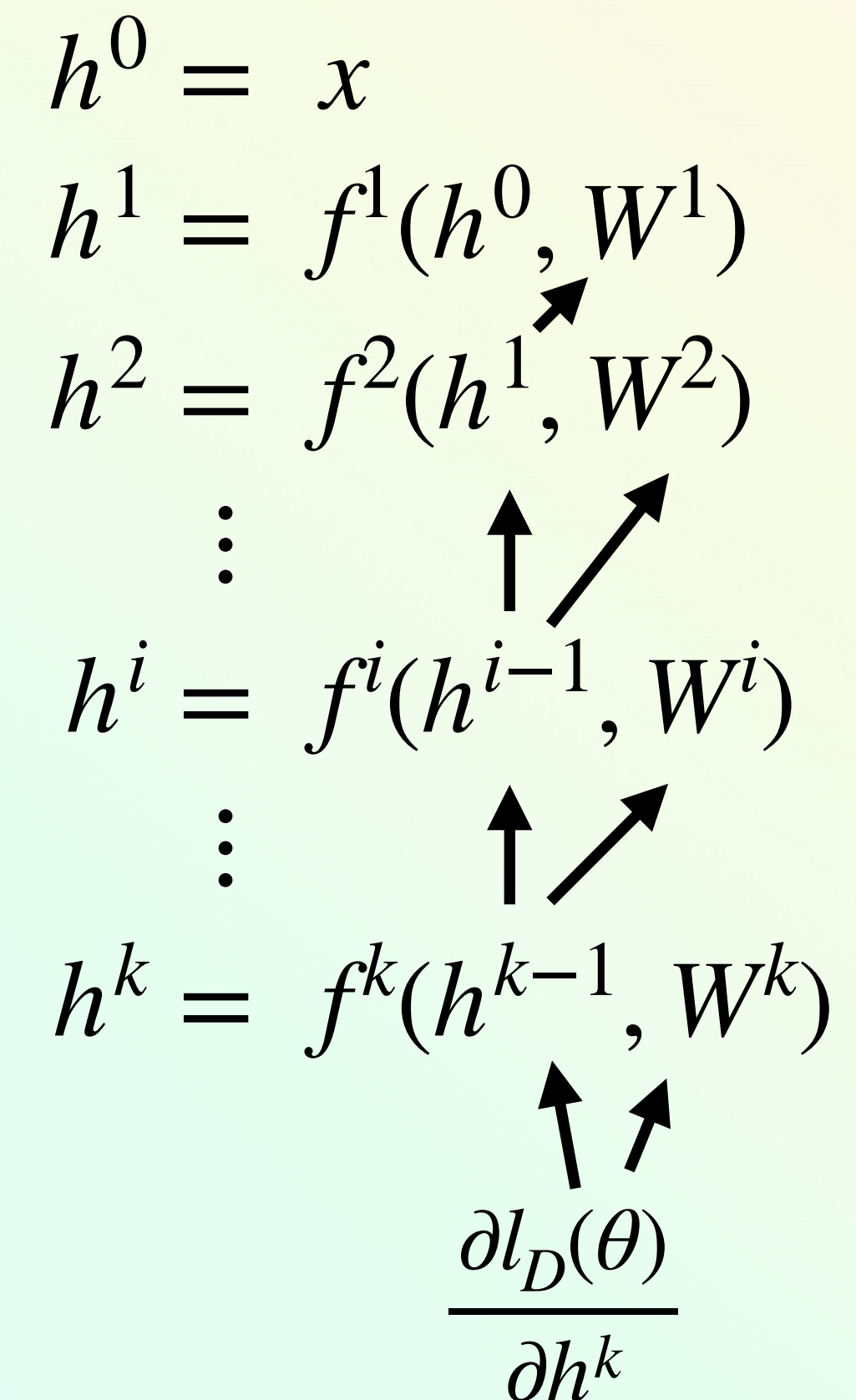
Compute $\frac{\partial l_D(\theta)}{\partial h^k}$

Then compute $\frac{\partial l_D(\theta)}{\partial W^k}$ and $\frac{\partial l_D(\theta)}{\partial h^{k-1}}$

Repeat till W^1

$$\begin{array}{l} h^0 = x \\ h^1 = f^1(h^0, W^1) \\ h^2 = f^2(h^1, W^2) \\ \vdots \\ h^i = f^i(h^{i-1}, W^i) \\ \vdots \\ h^k = f^k(h^{k-1}, W^k) \end{array}$$

$\frac{\partial l_D(\theta)}{\partial h^k}$



MANUAL BACKPROP

WRITING YOUR OWN GRADIENTS

Create each function f^i and the functions for the derivatives

$$f^i(\partial_{h^i}, h^{i-1}, W^i), \text{ which return } \frac{\partial l(\theta)}{\partial W^i} \text{ and } \frac{\partial l(\theta)}{\partial h^{i-1}}.$$

Create loss function $g(h^k, y)$ and derivative $\frac{\partial g(h^k, y)}{\partial h^k}$

1. Compute each h^i
2. Compute $g(h^k, y)$ and set $\partial_{h^k} \leftarrow \frac{\partial g(h^k, y)}{\partial h^k}$
3. For i in $\{k, k-1, \dots, 1\}$
 - $\partial_{h^{i-1}}, \partial_{W^i} \leftarrow f^i(\partial_{h^i}, h^{i-1}, W^i)$
4. Return $\{\partial_{W^1}, \dots, \partial_{W^k}\}$

MANUAL BACKPROP

WRITING YOUR OWN GRADIENTS

If we change the network or loss function, we must write and test new functions for the derivatives.

Pros:

- We can write very efficient code for the derivatives

Cons:

- A lot of work, prone to errors, cannot try many things easily

AUTO DIFFERENTIATION

AN APPROXIMATION

$$\frac{df(x)}{dx} \doteq \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta) - f(x)}{\Delta}$$

Instead of knowing the rules of differentiation, we can approximate the derivative.

AUTO DIFFERENTIATION

AN APPROXIMATION

$$\frac{df(x)}{dx} \doteq \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta) - f(x)}{\Delta}$$

Instead of knowing the rules of differentiation, we can approximate the derivative.

Method of Finite Differences: for some small Δ

$$\frac{df(x)}{dx} \approx \frac{f(x + \Delta) - f(x)}{\Delta}$$

AUTO DIFFERENTIATION

AN APPROXIMATION

$$\frac{df(x)}{dx} \doteq \lim_{\Delta \rightarrow 0} \frac{f(x + \Delta) - f(x)}{\Delta}$$

Instead of knowing the rules of differentiation, we can approximate the derivative.

Method of Finite Differences: for some small Δ

$$\frac{df(x)}{dx} \approx \frac{f(x + \Delta) - f(x)}{\Delta}$$

Centered estimate:
$$\frac{df(x)}{dx} \approx \frac{f(x + \Delta) - f(x - \Delta)}{2\Delta}$$

FINITE DIFFERENCES

MULTIPLE DIMENSIONS

$$\frac{\partial f(x)}{\partial x_i} \doteq \lim_{\Delta \rightarrow 0} \frac{f(x + [0, \dots, \Delta, \dots, 0]^\top) - f(x)}{\Delta}$$

FINITE DIFFERENCES

MULTIPLE DIMENSIONS

$$\frac{\partial f(x)}{\partial x_i} \doteq \lim_{\Delta \rightarrow 0} \frac{f(x + [0, \dots, \Delta, \dots, 0]^\top) - f(x)}{\Delta}$$

Apply finite differences on each element of x

$$\begin{bmatrix} \partial_{x_1} f(x) \\ \partial_{x_2} f(x) \\ \vdots \\ \partial_{x_n} f(x) \end{bmatrix} \approx \begin{bmatrix} \frac{f(x + [\Delta, 0, \dots, 0]^\top) - f(x)}{\Delta} \\ \frac{f(x + [0, \Delta, \dots, 0]^\top) - f(x)}{\Delta} \\ \vdots \\ \frac{f(x + [0, 0, \dots, \Delta]^\top) - f(x)}{\Delta} \end{bmatrix}$$

FINITE DIFFERENCES

MULTIPLE DIMENSIONS

$$\frac{\partial f(x)}{\partial x_i} \doteq \lim_{\Delta \rightarrow 0} \frac{f(x + [0, \dots, \Delta, \dots, 0]^\top) - f(x)}{\Delta}$$

Apply finite differences on each element of x

Requires $n + 1$ function evaluations.

Neural networks have thousands to billions of parameters

Too expensive

$$\begin{bmatrix} \partial_{x_1} f(x) \\ \partial_{x_2} f(x) \\ \vdots \\ \partial_{x_n} f(x) \end{bmatrix} \approx \begin{bmatrix} \frac{f(x + [\Delta, 0, \dots, 0]^\top) - f(x)}{\Delta} \\ \frac{f(x + [0, \Delta, \dots, 0]^\top) - f(x)}{\Delta} \\ \vdots \\ \frac{f(x + [0, 0, \dots, \Delta]^\top) - f(x)}{\Delta} \end{bmatrix}$$

AUTO DIFFERENTIATION

USING KNOWLEDGE OF DIFFERENTIATION RULES

Additive Rule: $\frac{\partial f(x) + g(x)}{\partial x} = \frac{\partial f(x)}{\partial x} + \frac{\partial g(x)}{\partial x}$

Multiplicative Rule: $\frac{\partial f(x)g(x)}{\partial x} = \frac{\partial f(x)}{\partial x}g(x) + f(x)\frac{\partial g(x)}{\partial x}$

Power Rule: $\frac{\partial x^n}{\partial x} = nx^{n-1}$ Exponential Rule: $\frac{\partial e^x}{\partial x} = e^x$ Chain Rule: $\frac{\partial g(f(x))}{\partial x} = \frac{\partial g(f(x))}{\partial f(x)} \frac{\partial f(x)}{\partial x}$

Use these rules to create functions for the derivatives automatically

AUTO DIFFERENTIATION

USING KNOWLEDGE OF DIFFERENTIATION RULES

Additive Rule: $\frac{\partial f(x) + g(x)}{\partial x} = \frac{\partial f(x)}{\partial x} + \frac{\partial g(x)}{\partial x}$

Multiplicative Rule: $\frac{\partial f(x)g(x)}{\partial x} = \frac{\partial f(x)}{\partial x}g(x) + f(x)\frac{\partial g(x)}{\partial x}$

Power Rule: $\frac{\partial x^n}{\partial x} = nx^{n-1}$ Exponential Rule: $\frac{\partial e^x}{\partial x} = e^x$ Chain Rule: $\frac{\partial g(f(x))}{\partial x} = \frac{\partial g(f(x))}{\partial f(x)} \frac{\partial f(x)}{\partial x}$

Use these rules to create functions for the derivatives automatically

AUTO DIFFERENTIATION

USING KNOWLEDGE OF DIFFERENTIATION RULES

Two modes: Forward mode and reverse mode (backprop)

AUTO DIFFERENTIATION

BREAKING DOWN DERIVATIVE INTO MATRIX PRODUCT

Abuse notation:

$$W^i \in \mathbb{R}^{p_i}, \quad p_i = n_i n_{i-1}$$

$$h^i \in \mathbb{R}^{n_i}$$

$$\frac{\partial h_i}{\partial h_{i-1}} \in \mathbb{R}^{n_{i-1} \times n_i}, \quad \frac{\partial h_i}{\partial W^i} \in \mathbb{R}^{p_i \times n_i}$$

AUTO DIFFERENTIATION

BREAKING DOWN DERIVATIVE INTO MATRIX PRODUCT

$$\frac{\partial l(\theta)}{\partial W^{k-2}} = \frac{\partial h^{k-2}}{\partial W^{k-2}} \frac{\partial h^{k-1}}{\partial h^{k-2}} \frac{\partial h^k}{\partial h^{k-1}} \frac{\partial l(\theta)}{\partial h^k}$$

AUTO DIFFERENTIATION

BREAKING DOWN DERIVATIVE INTO MATRIX PRODUCT

$$\frac{\partial l(\theta)}{\partial W^{k-2}} = \underbrace{\frac{\partial h^{k-2}}{\partial W^{k-2}}}_{p_{k-2} \times n_{k-2}} \underbrace{\frac{\partial h^{k-1}}{\partial h^{k-2}}}_{n_{k-2} \times n_{k-1}} \underbrace{\frac{\partial h^k}{\partial h^{k-1}}}_{n_{k-1} \times n_k} \underbrace{\frac{\partial l(\theta)}{\partial h^k}}_{n_k \times 1}$$

AUTO DIFFERENTIATION

BREAKING DOWN DERIVATIVE INTO MATRIX PRODUCT

Forward mode:

$$\begin{array}{c}
 \xrightarrow{\hspace{10em}} \\
 \frac{\partial l(\theta)}{\partial W^{k-2}} = \underbrace{\underbrace{\frac{\partial h^{k-2}}{\partial W^{k-2}}}_{p_{k-2} \times n_{k-2}} \underbrace{\frac{\partial h^{k-1}}{\partial h^{k-2}}}_{n_{k-2} \times n_{k-1}} \underbrace{\frac{\partial h^k}{\partial h^{k-1}}}_{n_{k-1} \times n_k}}_{p_{k-2} \times n_{k-1}} \underbrace{\frac{\partial l(\theta)}{\partial h^k}}_{n_k \times 1} \\
 \underbrace{\hspace{10em}}_{p_{k-2} \times n_k} \\
 \underbrace{\hspace{10em}}_{p_{k-2} \times 1}
 \end{array}$$


Cost
 $nm(2p - 1)$ operations
 for matrix-matrix multiplication with dims
 $n \times p$ and $p \times m$

Forward mode cost scales with p_{k-2}

AUTO DIFFERENTIATION

BREAKING DOWN DERIVATIVE INTO MATRIX PRODUCT

Reverse mode:



$$\frac{\partial l(\theta)}{\partial W^{k-2}} = \underbrace{\underbrace{\underbrace{\frac{\partial h^{k-2}}{\partial W^{k-2}}}_{p_{k-2} \times n_{k-2}} \underbrace{\frac{\partial h^{k-1}}{\partial h^{k-2}}}_{n_{k-2} \times n_{k-1}} \underbrace{\frac{\partial h^k}{\partial h^{k-1}}}_{n_{k-1} \times n_k}}_{n_{k-1} \times 1}}_{n_{k-2} \times 1}}_{p_{k-2} \times 1} \underbrace{\frac{\partial l(\theta)}{\partial h^k}}_{n_k \times 1}$$

Cost
 $n(2p - 1)$ operations
 for matrix-vector multiplication with dims
 $n \times p$ and $p \times 1$

Reverse mode performs matrix-vector
 multiplication not matrix-matrix

AUTODIFF EXAMPLE

CODE

$$l(x, y) = 3(xy + y)$$

```
def add(x,y):
```

```
    z = x + y
```

```
    return z
```

```
def d_add(dz,x,y):
```

```
    return dz, dz
```

```
def prod(x,y):
```

```
    z= x * y
```

```
    Return z
```

```
def d_prod(dz, x,y):
```

```
    return dz*y, dz*x
```


AUTODIFF EXAMPLE

CODE

$$l(x, y) = 3(xy + y)$$

```
def l(x,y):
```

```
    P1 = prod(x,y)
```

```
    P2 = add(P1, y)
```

```
    P3 = prod(3, P2)
```

```
    return P3
```

```
def d_l(dz, x,y):
```

```
    P1 = prod(x,y)
```

```
    P2 = add(P1, y)
```

```
    P3 = prod(3, P2)
```

```
    _, DP3P2 = d_prod(dz, 3, P2)
```

```
    DP2P1, DP2y = d_add(DP3P2, P1, y)
```

```
    DP1x, DP1y = d_prod(DP2P1,x,y)
```

```
    Dx = DP1x
```

```
    Dy = DP1y + DP2y
```

```
    Return Dx, Dy
```


AUTODIFF EXAMPLE

CODE

$$l(x, y) = 3(xy + y)$$

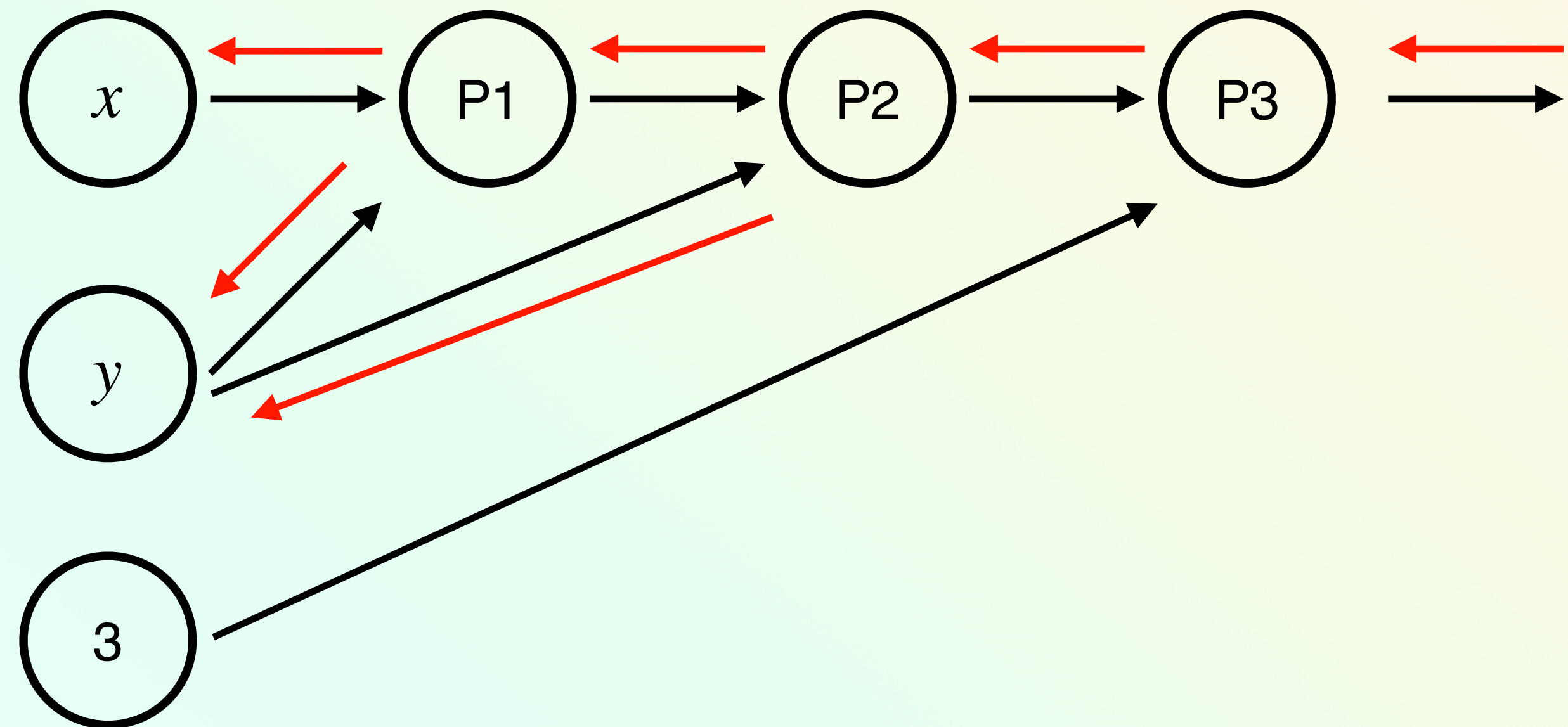
```
def l(x,y):
```

```
    P1 = prod(x,y)
```

```
    P2 = add(P1, y)
```

```
    P3 = prod(3, P2)
```

```
    return P3
```



AUTODIFF EXAMPLE

IMPLEMENTATION

Ahead of time:

Take the computation graph of the code and create derivative code

Return function output and function for the derivatives of the inputs

AUTODIFF EXAMPLE

CODE

```
def add_(x,y):
    Return x+y, dz: d_add(dz, x,y)
def prod_(dz, x,y):
    Return x*y, dz: d_prod(dz, x,y)
Def l_(x,y):
    P1,DP1 = prod_(x,y)
    P2,DP2 = add_(P1,y)
    P3,DP3 = prod_(3,P2)
    Def d_l(dz):
        __, DP3P2 = DP3(dz)
        DP2P1, DP2y = DP2(DP3P2)
        DP1x, DP1y = DP1(P2P1)
        return DP1x, DP2y + DP1y
    return P3, d_l
```

“Compiler” automatically generates d_l based on known rules

If an operation does not have a derivative defined, the “compiler” throws an error.

Can create a neural network using only simple scalar rules.

More efficient: define custom derivatives for common functions (e.g., layers)

AUTODIFF EXAMPLE

TRACER MODE

Each variable is an object:

data — the value of the object

grad — gradient for the data

backward — function to compute the derivatives of the inputs

Children — list of variable objects used to construct this one

`X = Var(1.0)`

`X.data` is 1

`X.grad` is None

`X.backward()`

`X.grad` is 1.0

AUTODIFF LIMITATIONS

Derivatives apply on a per-operation level

$$\begin{aligned}\frac{\partial y \ln \sigma(x) + (1 - y) \ln(1 - \sigma(x))}{\partial x} &= y \frac{\partial \ln \sigma(x)}{\partial x} + (1 - y) \frac{\partial \ln(1 - \sigma(x))}{\partial x} \\ &= y \frac{\partial \ln \sigma(x)}{\partial \sigma(x)} \frac{\partial \sigma(x)}{\partial x} + (1 - y) \frac{\partial \ln(1 - \sigma(x))}{\partial 1 - \sigma(x)} \frac{\partial 1 - \sigma(x)}{\partial x}\end{aligned}$$

However, we computed this derivative by hand we found

$$\frac{\partial y \ln \sigma(x) + (1 - y) \ln(1 - \sigma(x))}{\partial x} = y - \sigma(x)$$

AUTODIFF LIMITATIONS

Backprop cannot discover more efficient derivatives

Numerical instability is a common problem due to extra multiplications/divisions

We need to be smart about how to compute functions so differentiation code is fast and reliable

INDUCTIVE BIASES

GOAL

We motivated neural networks as a way not to have to specify good basis functions

However, we know that having a good basis function makes learning easy

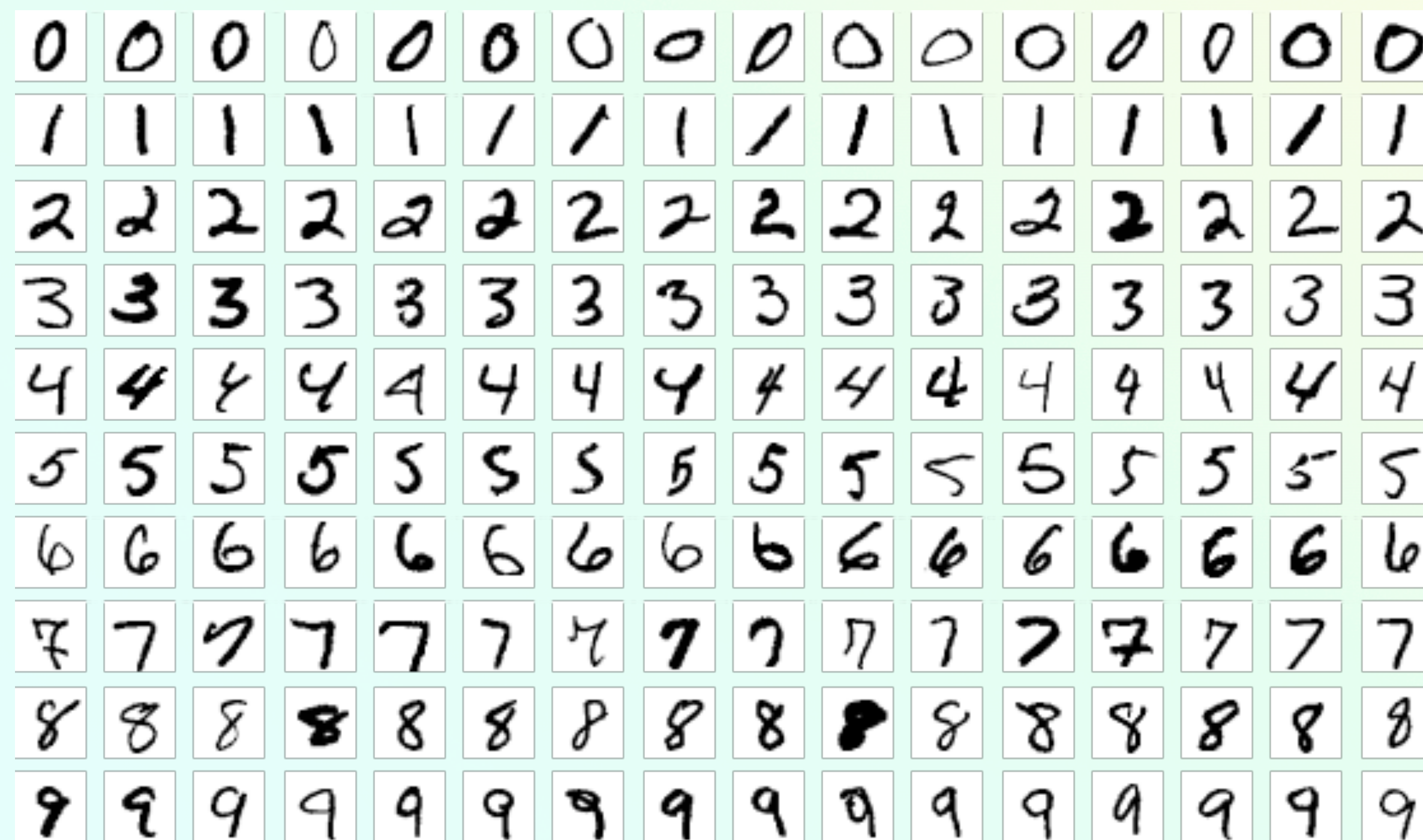
Neural networks are not immune to this problem.

- Need good initialization
- Need a good network structure

Choose network structure so the neural network is likely to learn the right properties of the data

MLPS

EXAMPLE — HANDWRITTEN DIGIT CLASSIFICATION



MLPS

EXAMPLE — HANDWRITTEN DIGIT CLASSIFICATION



Represent as a vector of features

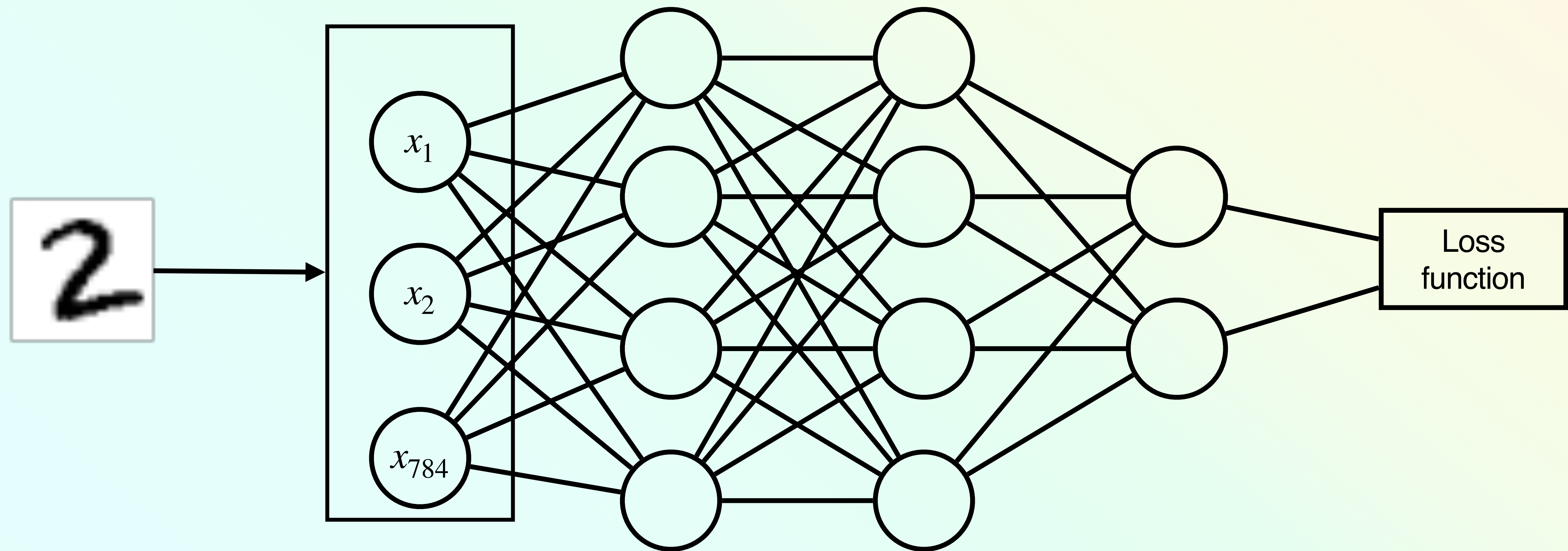
The image is a 28x28 pixel greyscale image (a matrix with values $[0, 255]$)

Map it to a 784-dimensional vector with each featuring being the intensity of a pixel

Rescale $[0, 255] \rightarrow [0, 1]$.

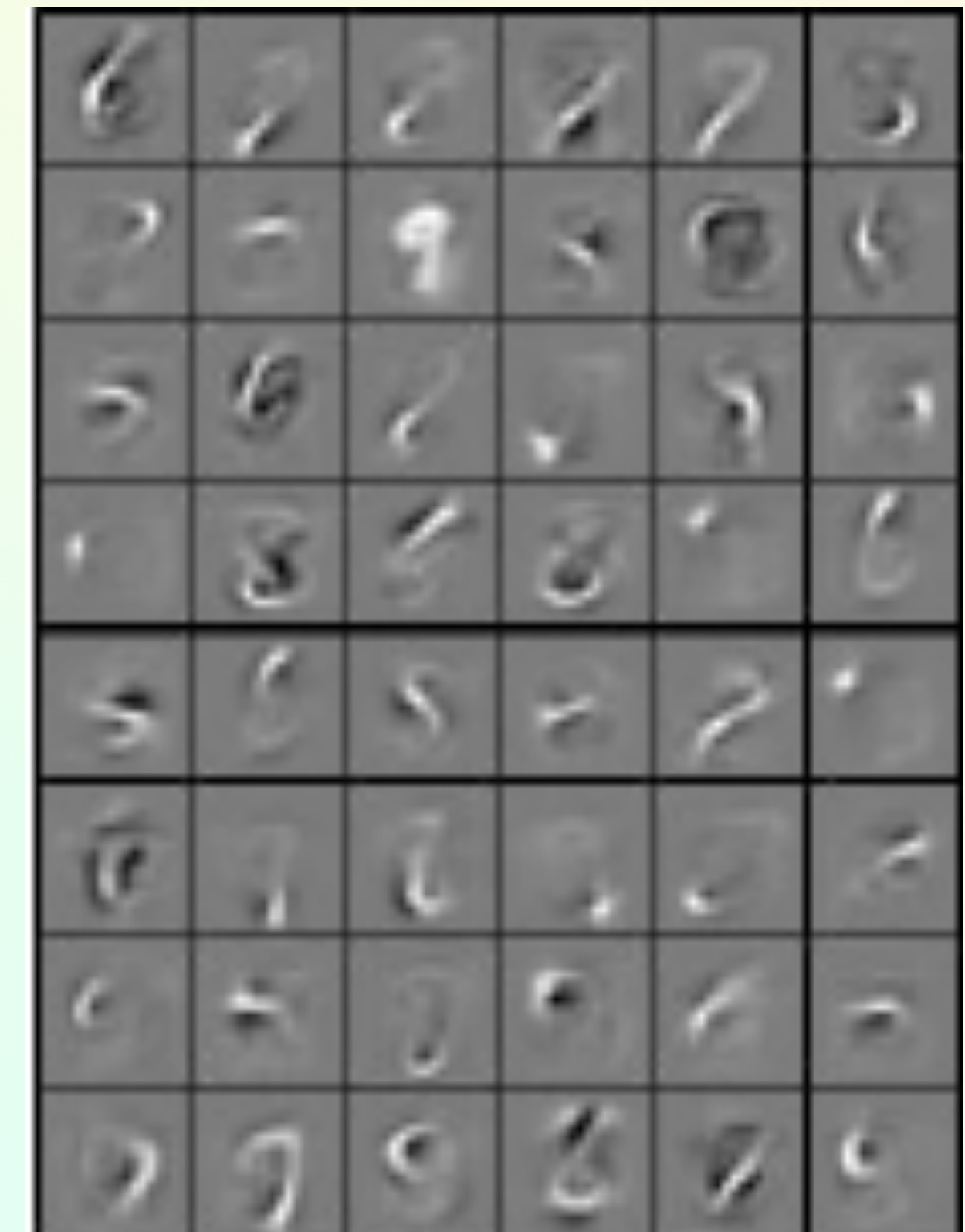
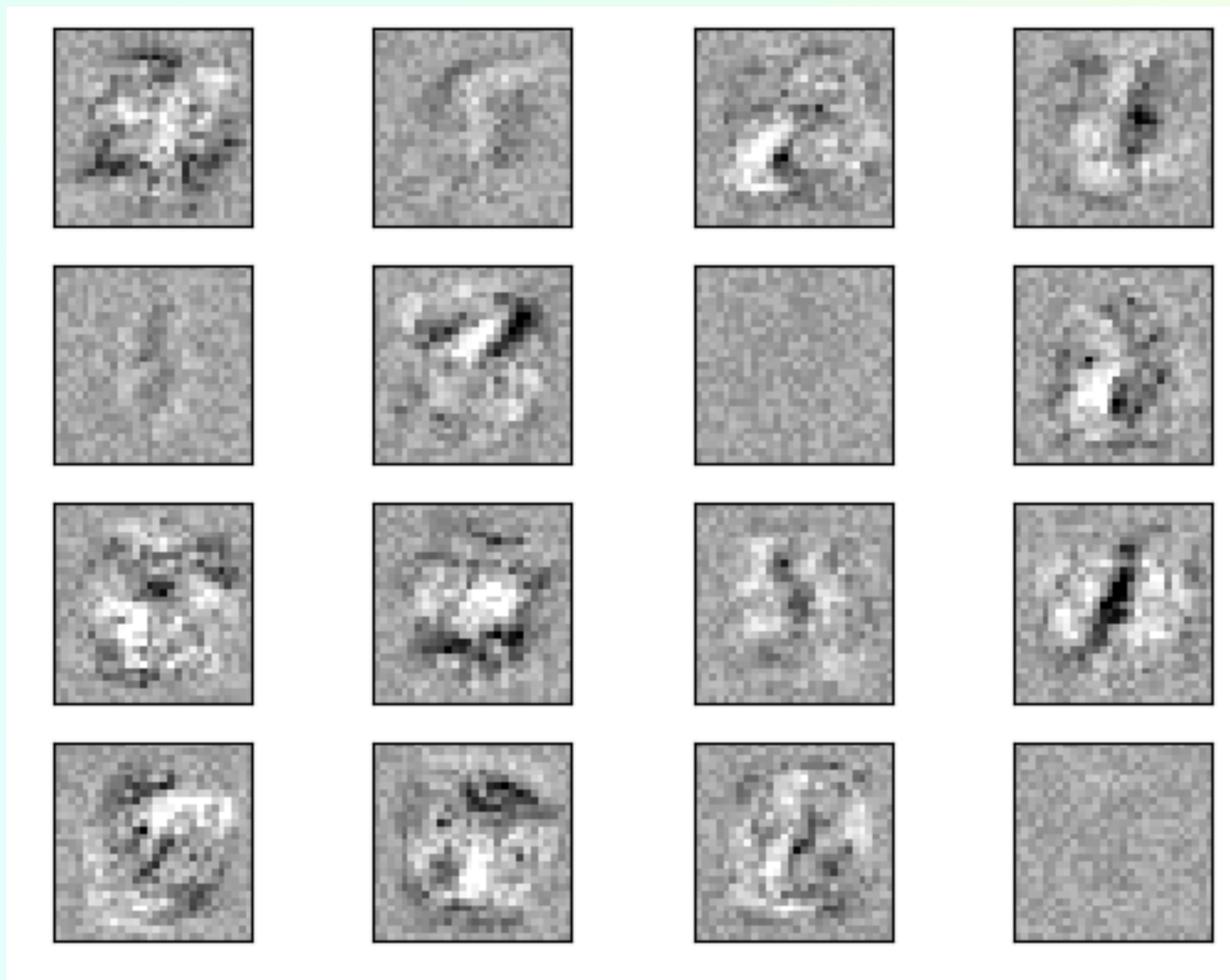
MLPS

EXAMPLE — HANDWRITTEN DIGIT CLASSIFICATION



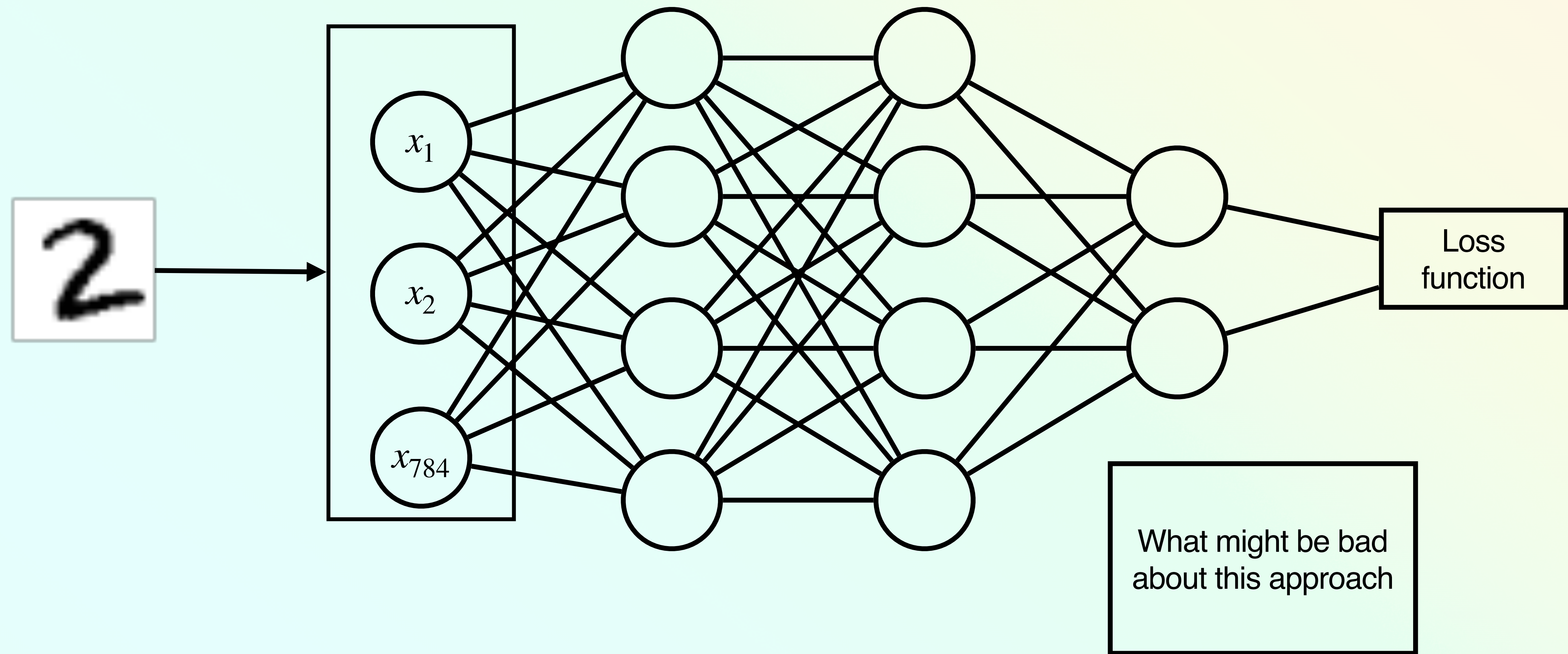
LEARNED FEATURES

MNIST — FIRST LAYER FEATURES



MLPS

EXAMPLE — HANDWRITTEN DIGIT CLASSIFICATION



MLPS

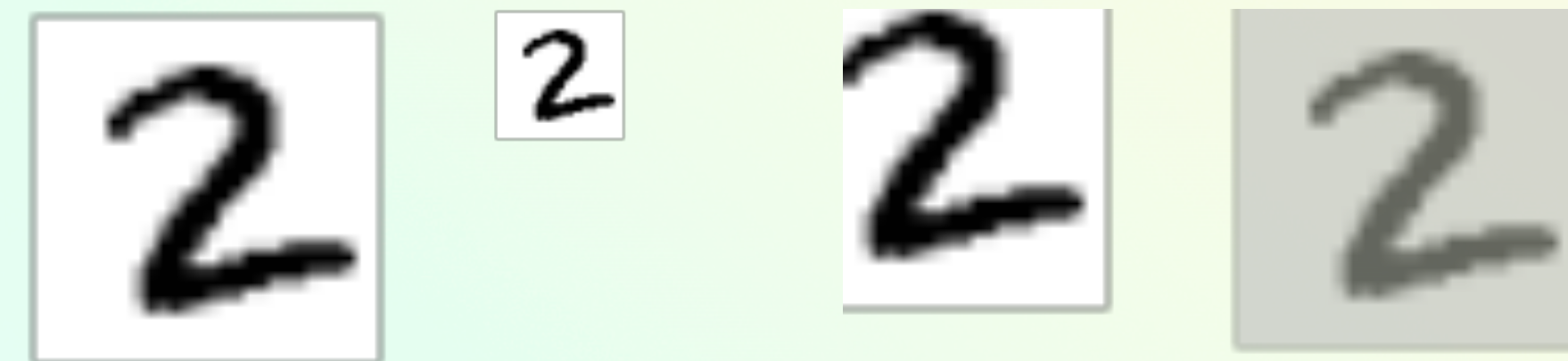
EXAMPLE — HANDWRITTEN DIGIT CLASSIFICATION

The concept of two does not depend on specific pixels being active and others not

Can shift the image,

Can rescale the image,

Change the brightness,



All of these transformations would still have the same concept of a 2

The MLP needs many units to detect all possible feature transformations

Needs to be trained on all these variations. (True for most NNs)

CONVOLUTIONAL NEURAL NETWORKS

EXPLOITING SPATIAL INVARIANCE

If the concept of the class is invariant to shifts in the image, then

We should have a neural network that is also invariant to shifts

Convolution:

- Instead of having a weight for every pixel

- Have weights for a small image patch

- Process each image patch with the weights

- Repeat with different weights to create multiple features for each patch

CONVOLUTIONAL NEURAL NETWORKS

EXPLOITING SPATIAL INVARIANCE

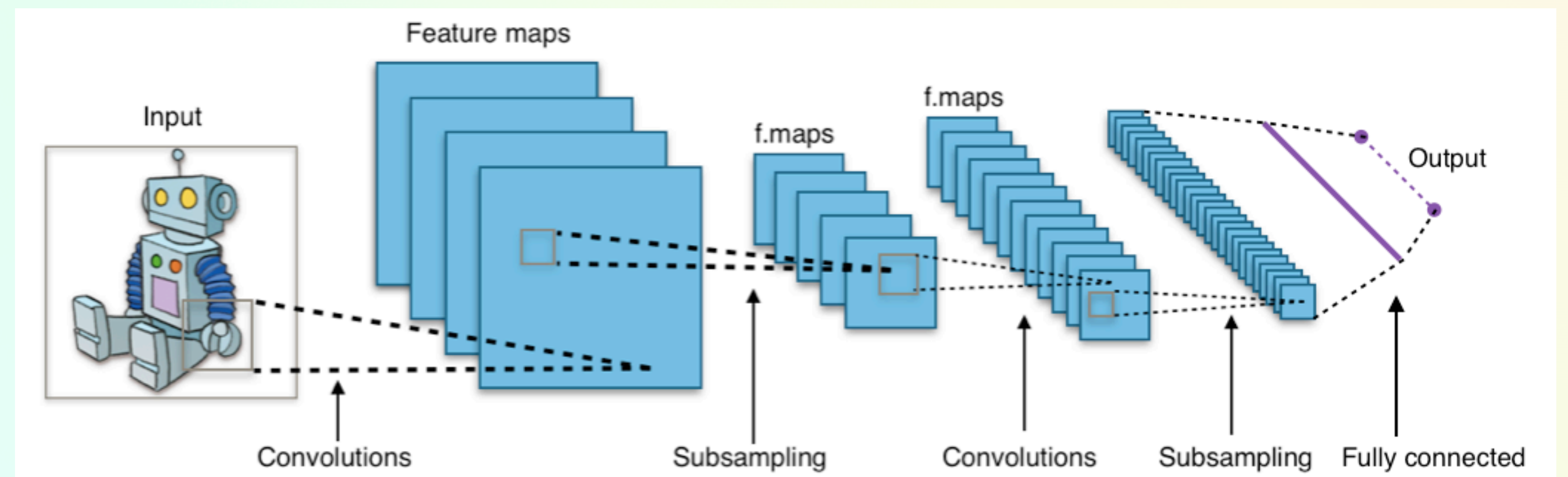
Each convolution map detects a different feature.

Applied to the whole image creates a feature map

Subsample to reduce dimensions

Repeat

Represent as a vector and pass to an MLP (not necessary)



DESIGNING NEURAL ARCHITECTURES

What properties does your data have that you can exploit:

Spatial invariance —> convolution

Graphical structure —> graph neural networks

Data comes from a differential equation —> add a differential equation layer

DESIGNING NEURAL ARCHITECTURES

Do you know something about the structure of the answer?

$$f_*(x) = \beta_1 x_1 + \beta_2 x_2 + g_*(x)$$

Create a neural network with model parameters

$$f(x, \theta, \beta_1, \beta_2) = \beta_1 x_1 + \beta_2 x_2 + g(x, \theta)$$

$$\nabla l(\theta, \beta_1, \beta_2) = \nabla \mathbf{E}[(f(X, \theta, \beta_1, \beta_2) - Y)^2]$$

NEXT CLASS

Representing language with neural networks