Short developer's tutorial on Deep Learning and Neural Networks with PyTorch

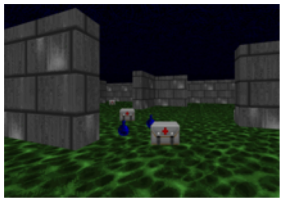
Christian Wolf

December 12th, 2019

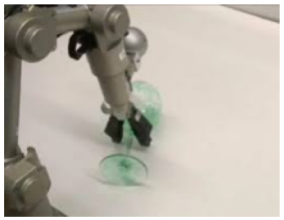
Decision making

→ {dog, **cat**, bike, avocado, blender, …}

→ {0, 1, … 26, 27, **28**…, 98, 99, …}



→ {Left, right, ahead, back, pick up, …}

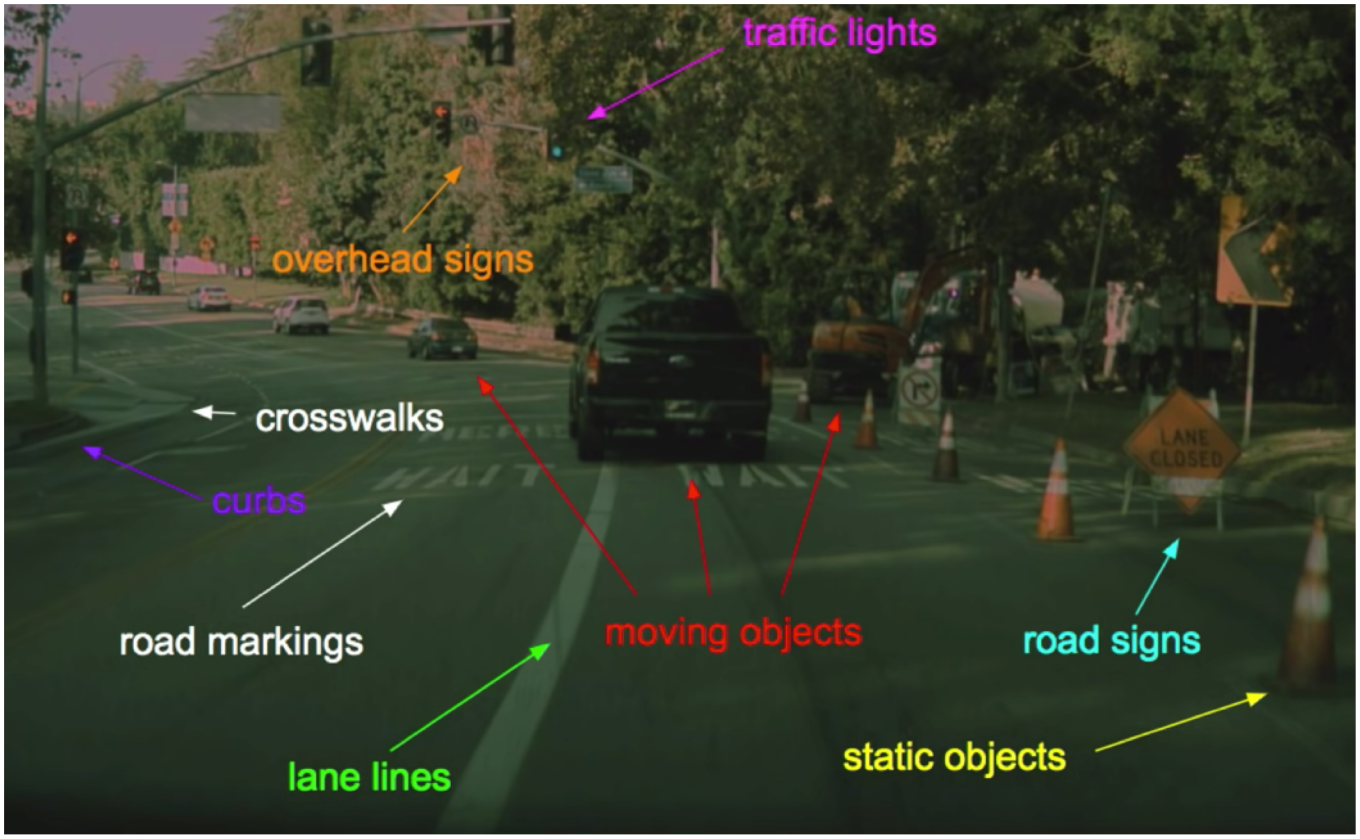


→ Motor commands

"A blue parrot with a yellow belly sitting on a branch in a forest" →

Deep Learning @ Tesla

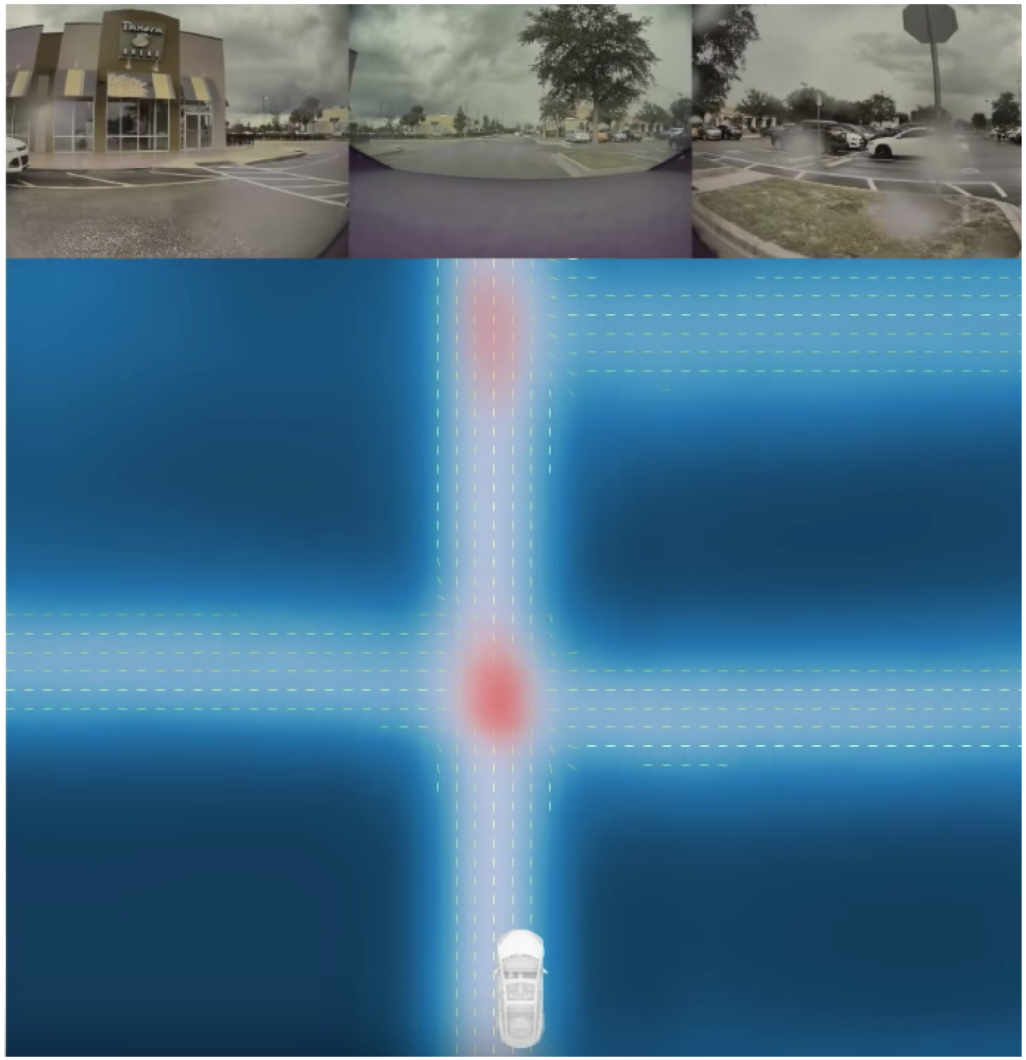
https://www.youtube.com/watch?v=oBklltKXtDE



- (3, 960, 1280) input images
- “ResNet-50 like” dilated backbones
- FPN / DeepLabV3 / UNet -like heads
- ~15 tasks => “prototypes framework”

Deep Learning @ Tesla

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



We would like to learn to predict a value y from observed input x

$$y = h(x, \theta)$$

Handcrafted from domain knowledge

Learned from data or interactions

Fully handcrafted

Fully Learned

The 3 fundamental problems of ML

1. Expressivity
   - What is the complexity of functions my model can represent?

2. Trainability
   - How easy is it to fit my model to my training data?

3. Generalization
   - Does my model generalize to unseen data?
   - In presence of shifts in distributions?

(After Eric Jang & Jascha Sohl-Dickstein)

1 Neural Networks & learning

2 Optimization / gradient descent

3 A simple linear example in PyTorch

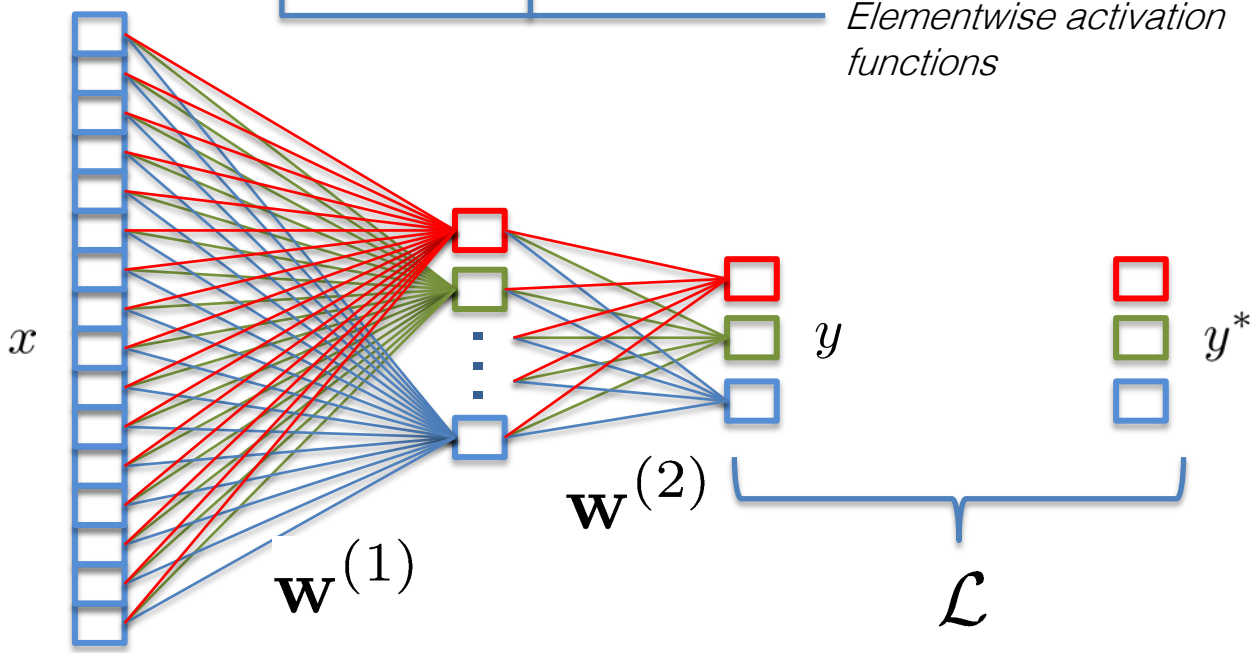
4 A multi-layer model in PyTorch

5 Convolutional networks

Deep neural networks

$$y_k(x) = \boxed{\sigma}\left(\sum_j w_{kj}^{(2)} \boxed{\zeta}\left(\sum_i w_{ji}^{(1)} x_i\right)\right)$$

Elementwise activation functions

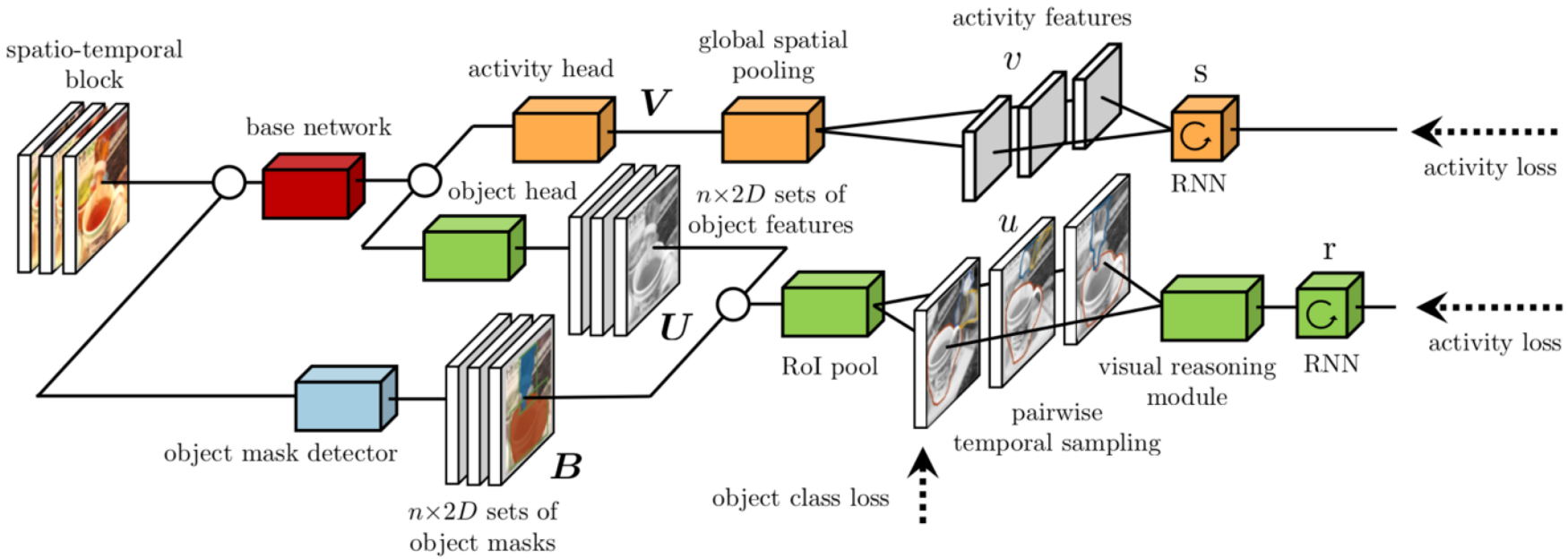


Input layer

Hidden layer

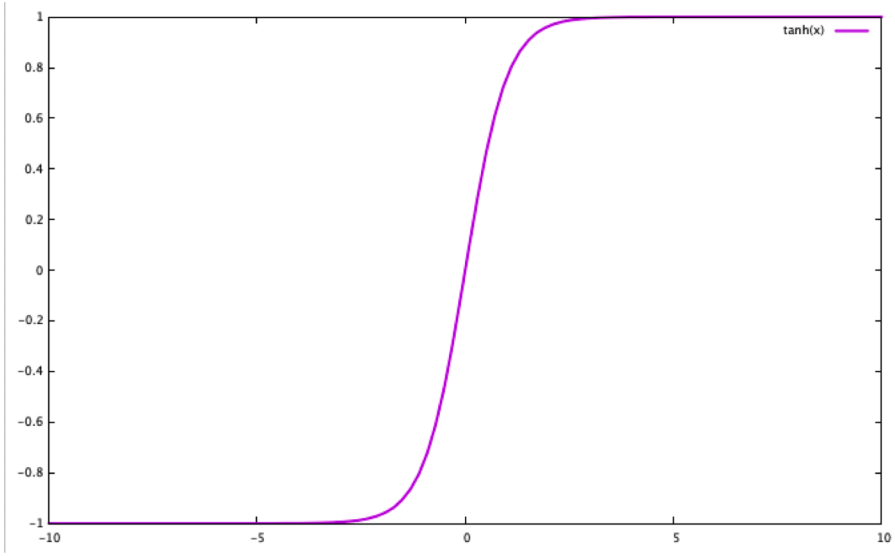
Output layer

Complex networks

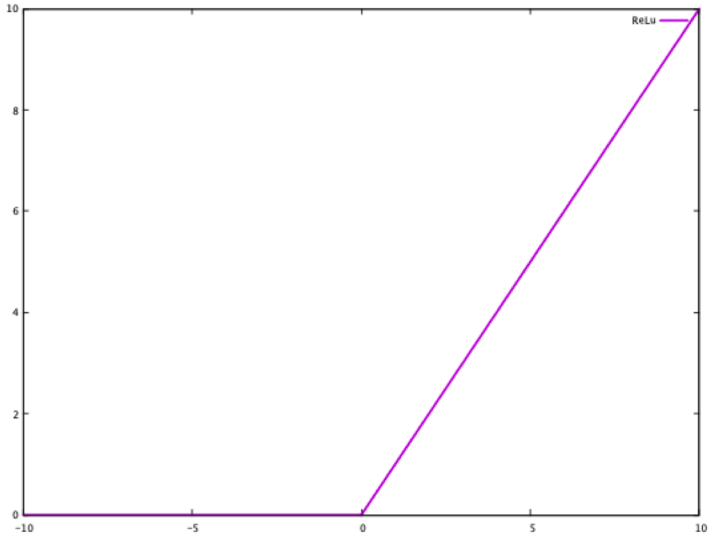


[Baradel, Neverova, Wolf, Mille, Mori, ECCV 2018]

Common activation functions



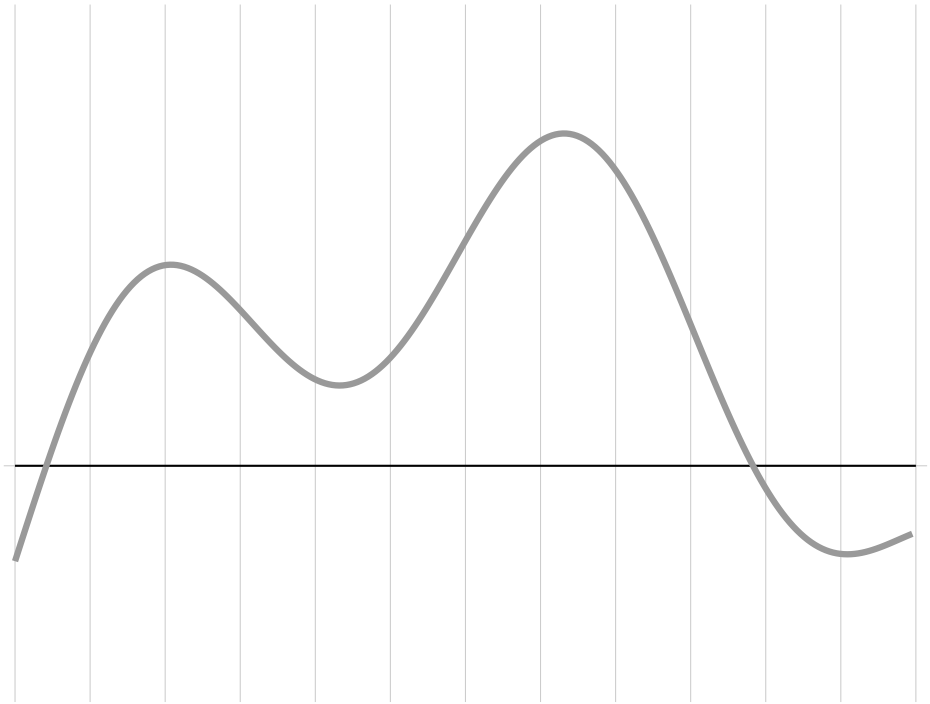
Hyperbolic tangent
tanh



Rectified Linear Unit
ReLU

Universal approximation

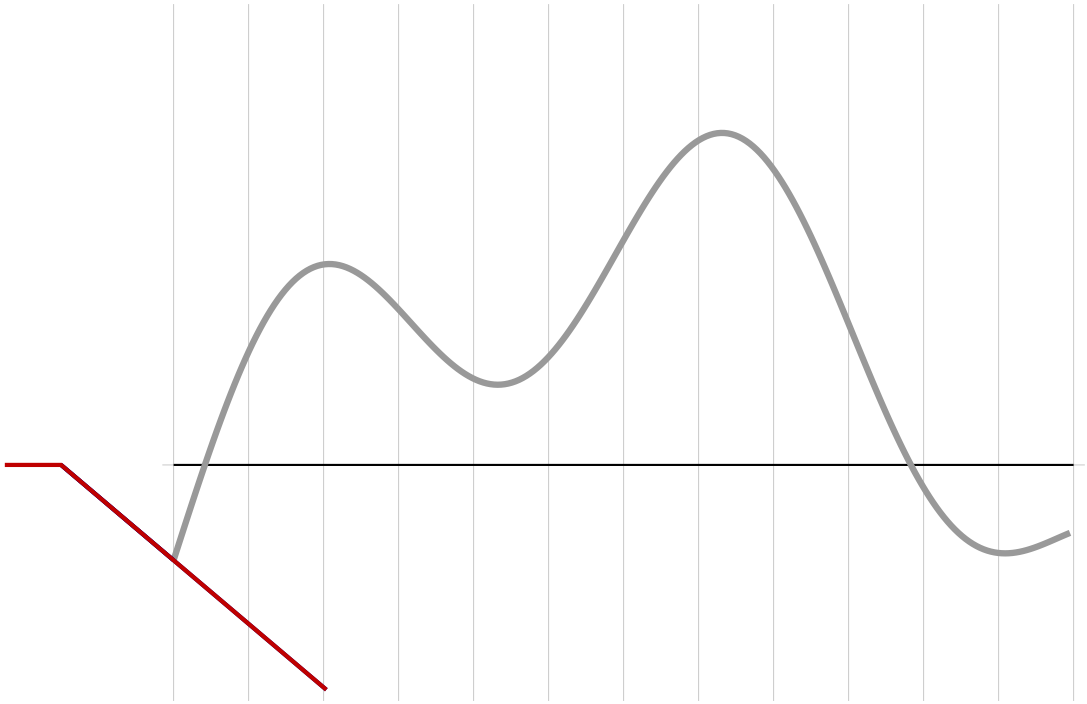
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

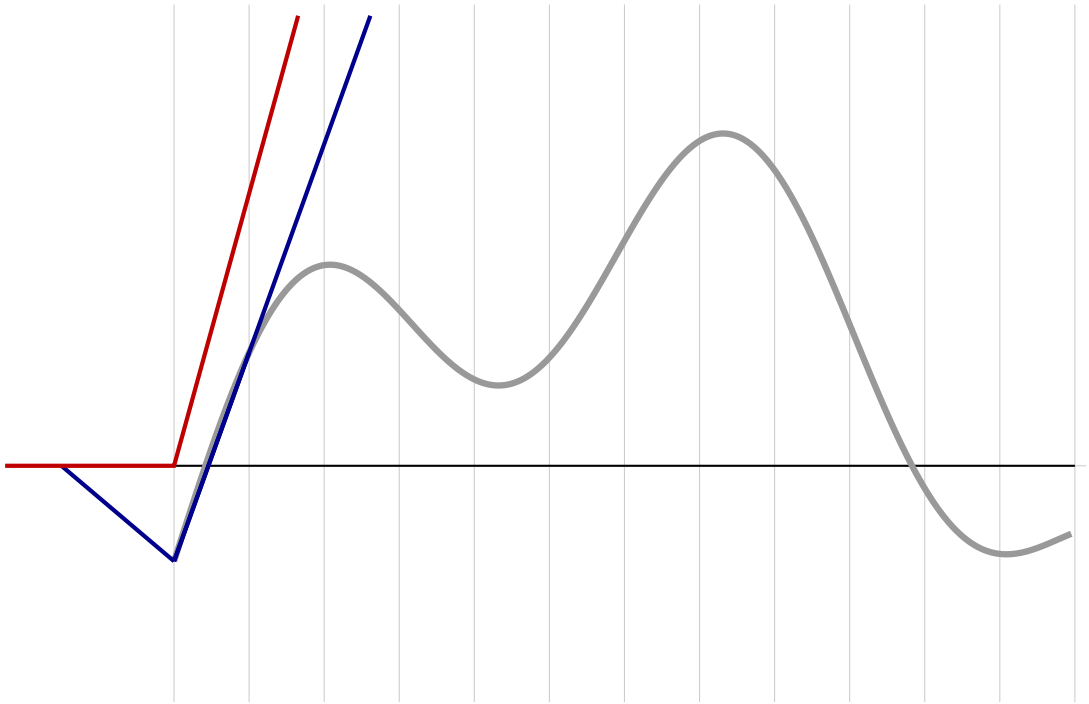
$$f(x) = \sigma(w_1 x + b_1)$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

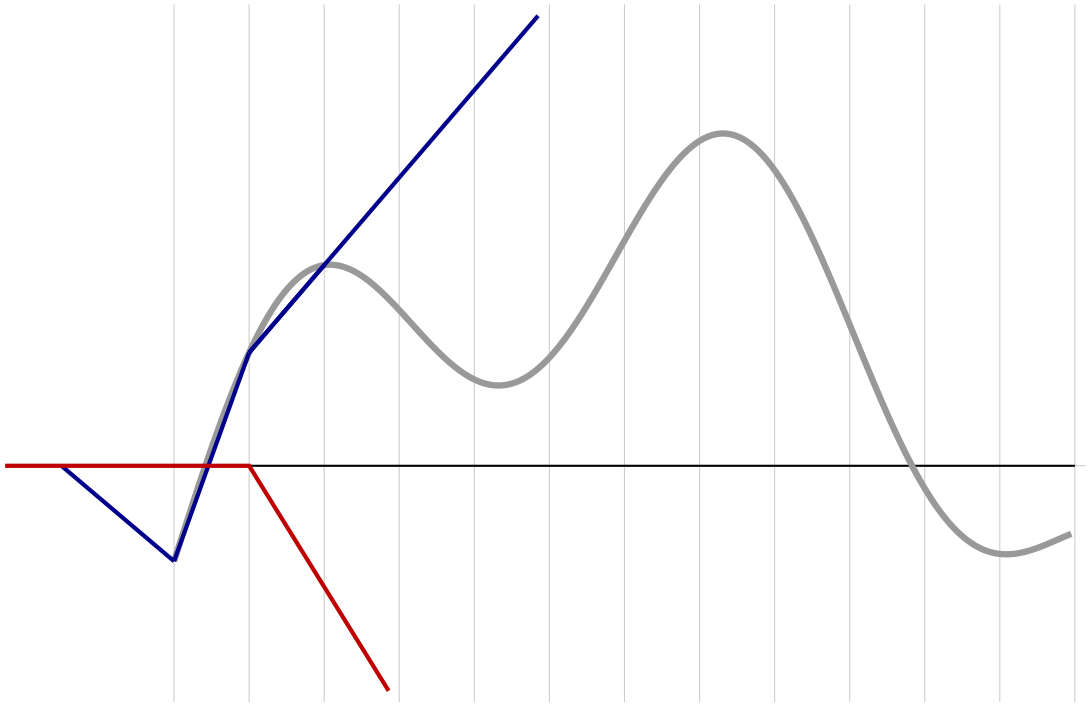
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

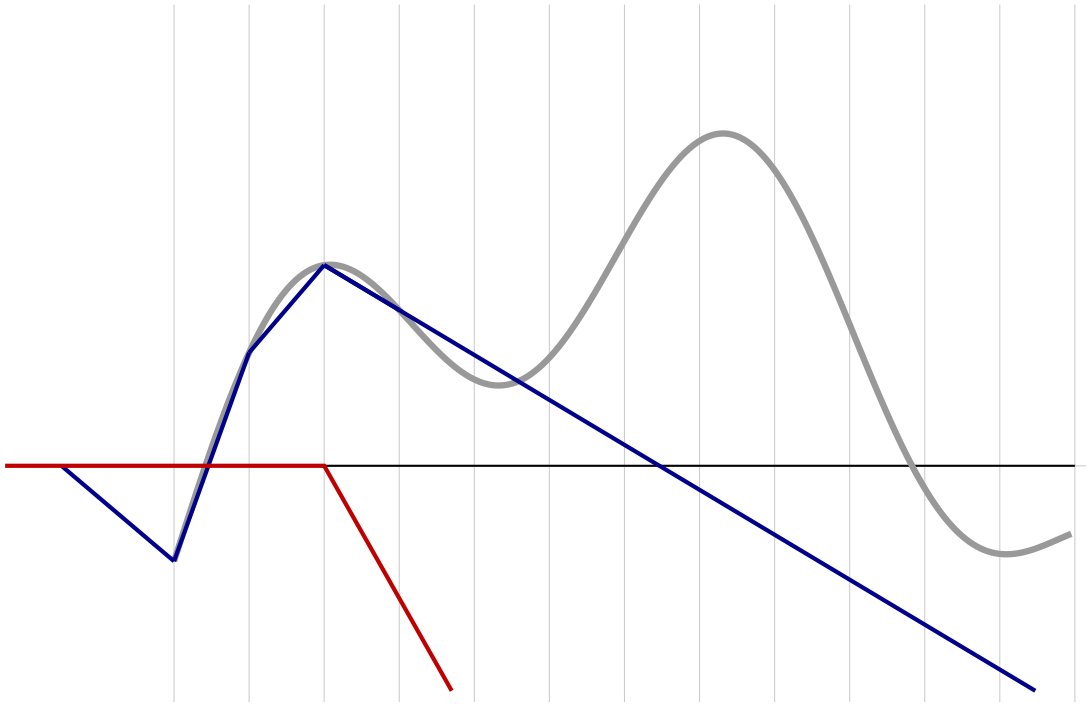
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3)$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

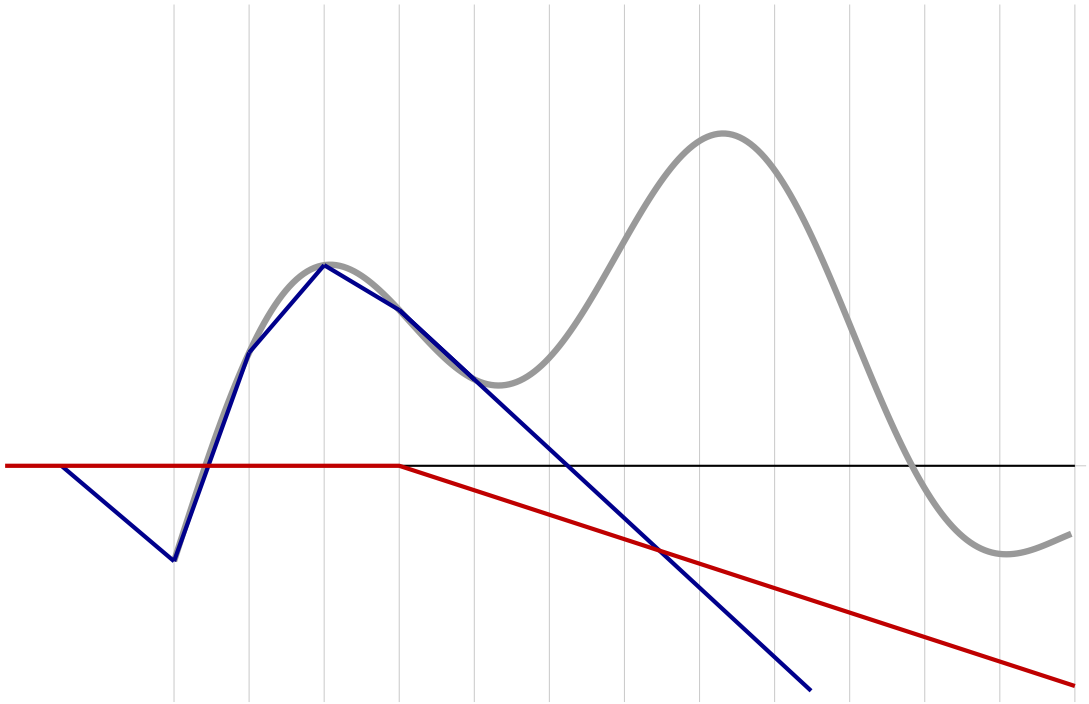
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

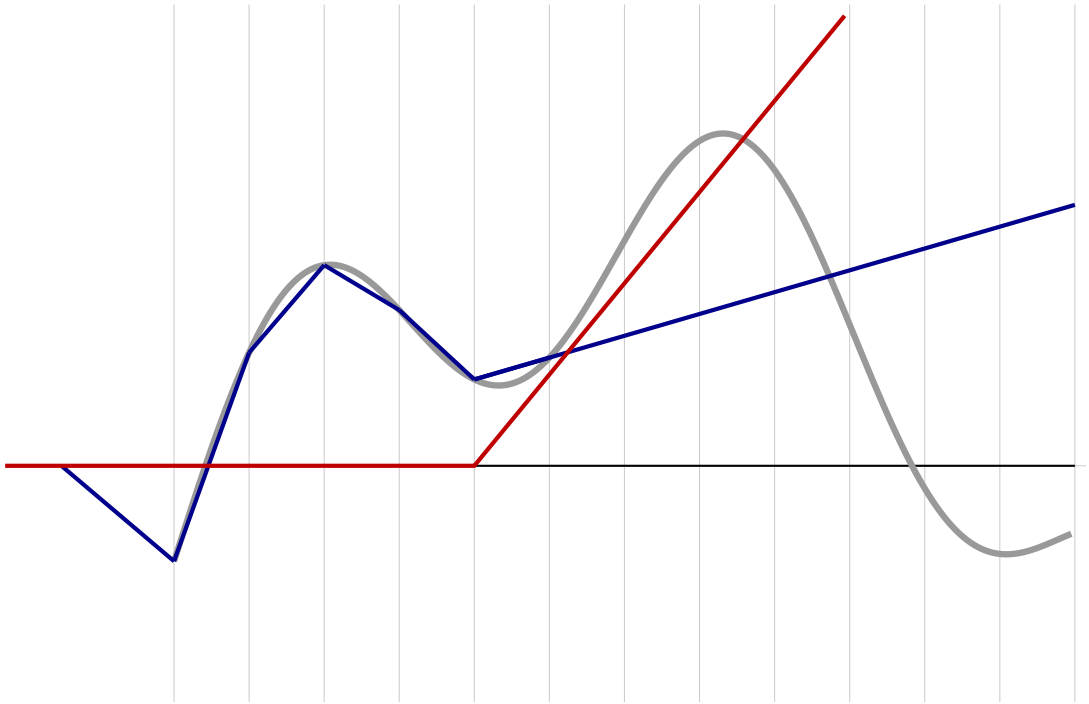
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

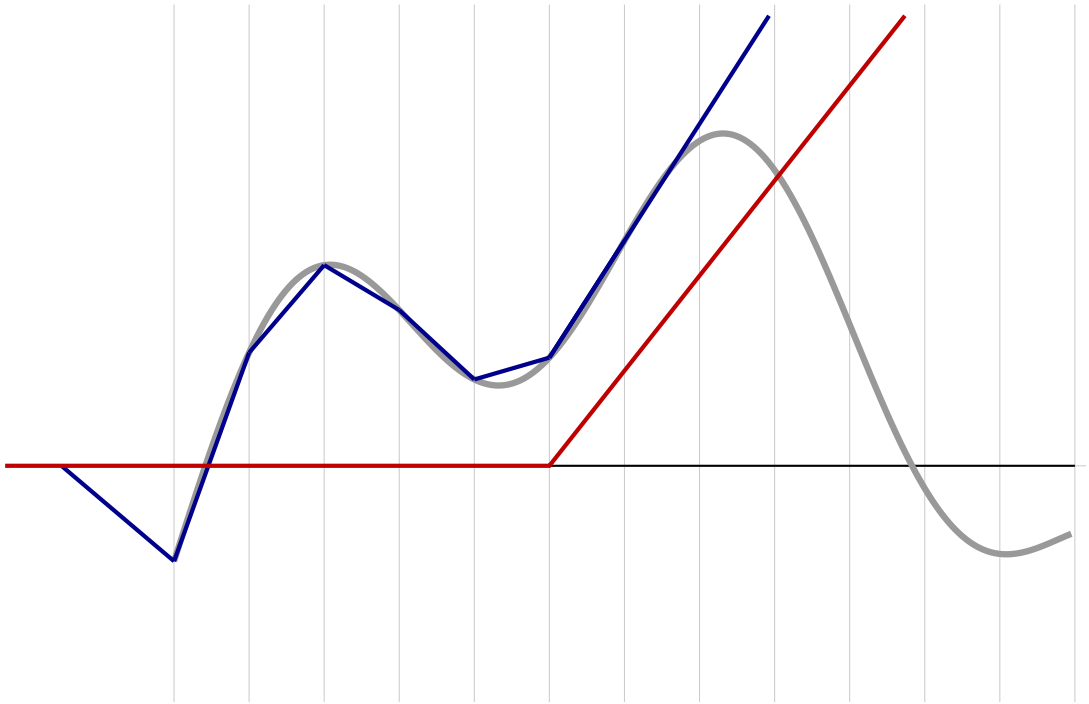
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

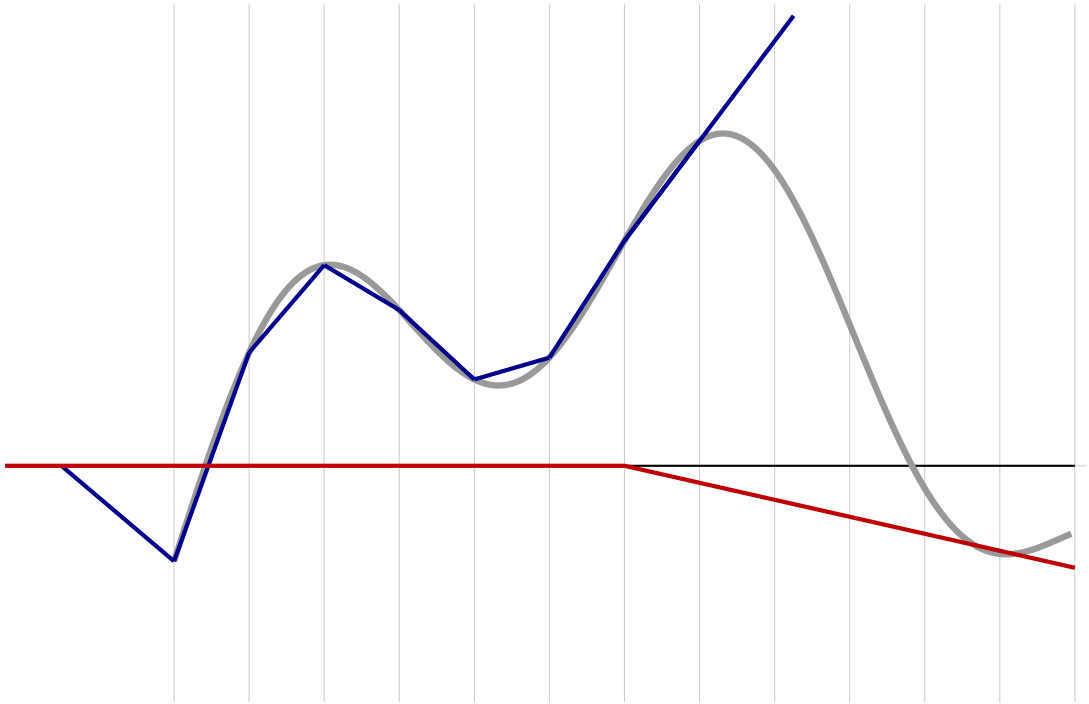
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

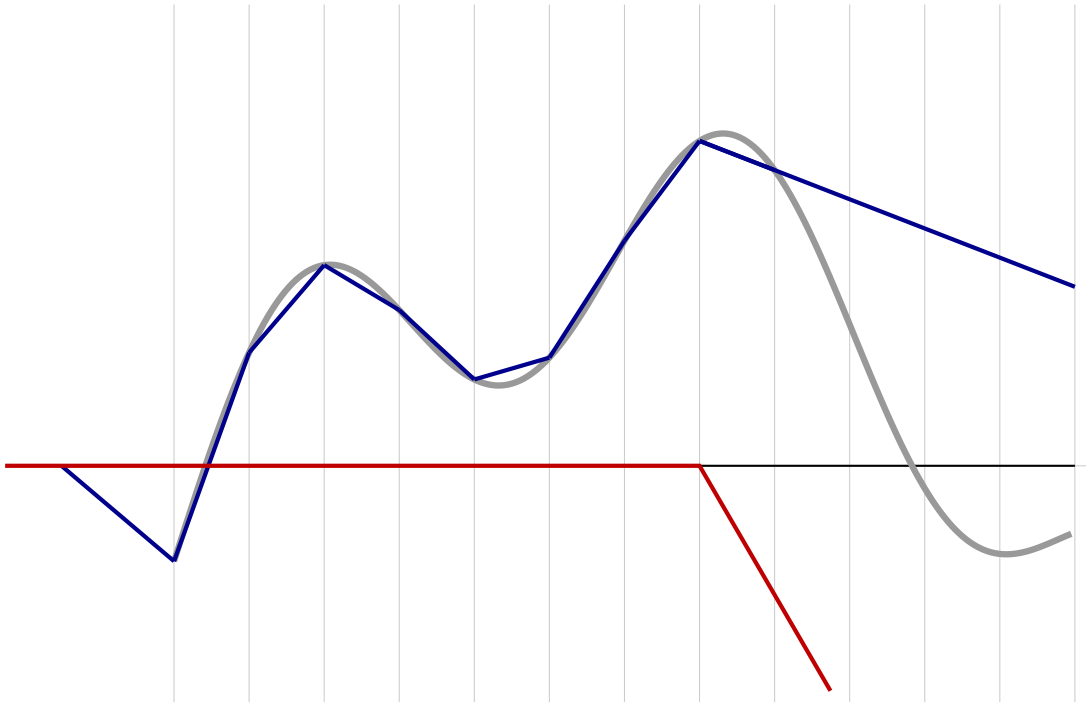
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

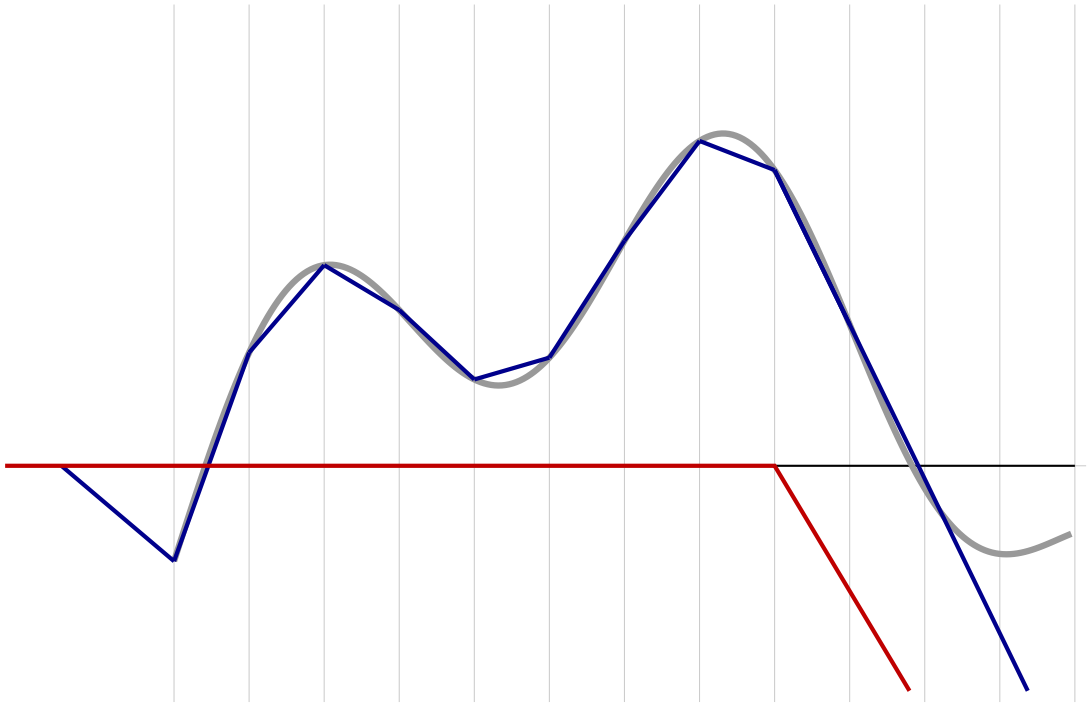
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

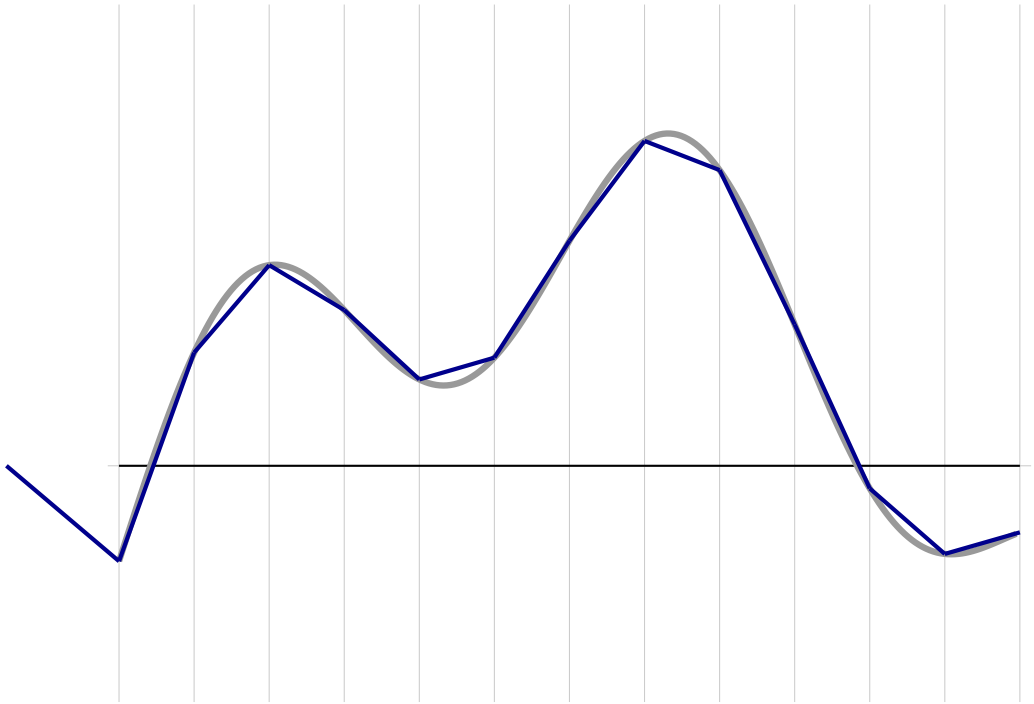
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

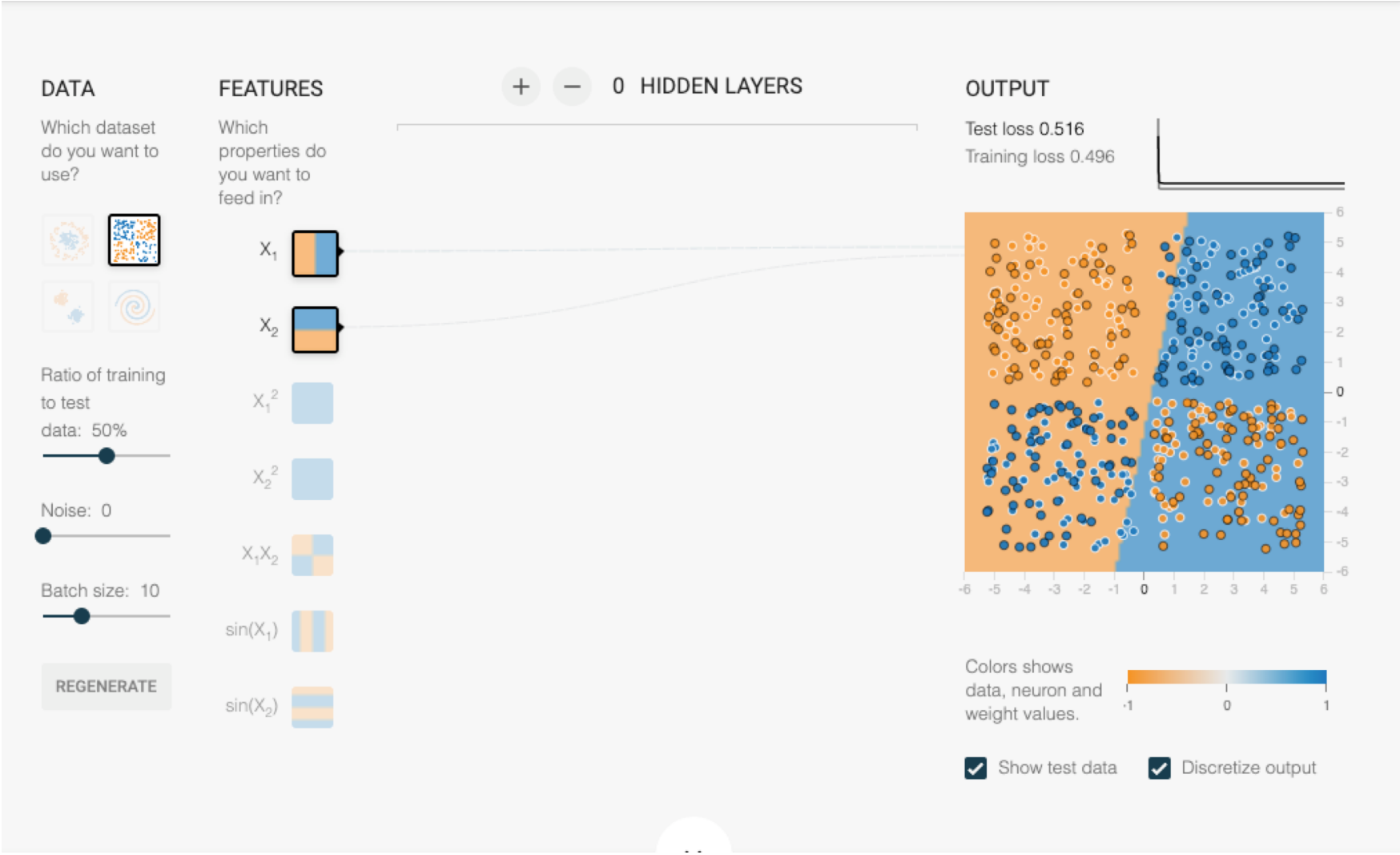
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



This is true for other activation functions u

Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/

Demo: Tensorflow Playground



https://playground.tensorflow.org

1 Neural Networks & learning

2 Optimization / gradient descent

3 A simple linear example in PyTorch

4 A multi-layer model in PyTorch

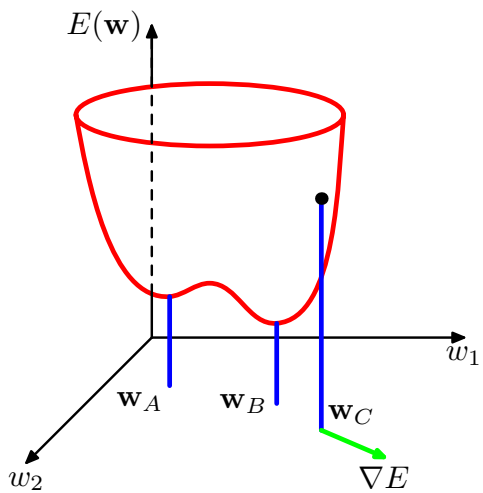
5 Convolutional networks

Learning by gradient descent

Iterative minimisation through gradient descent:

$$\theta^{[t+1]} = \theta^{[t]} + \boxed{\nu} \nabla \mathcal{L}(h(x, \theta), y^*)$$

Learning rate



[Figure: C. Bishop, 2006]

Can be blocked in a local minimum (not that it matters much …)

Gradient descent

One optimizer step:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}\left(h(x, \theta), y^*\right)$$

The gradient is a vector of partial derivatives:

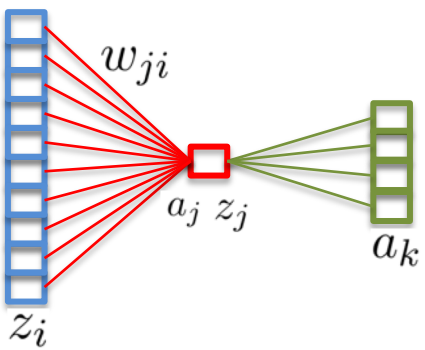
$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_0} \\ \\ \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \\ \vdots \\ \\ \frac{\partial \mathcal{L}}{\partial \theta_N} \end{bmatrix}$$

Differentiate a multi-layer network

Consider a multi-layer network, in particular an arbitrary unit indexed by j and receiving inputs from units index by i, providing outputs z_i.

Its gradient is:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \frac{\partial \mathcal{L}}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$
$$= \delta_j z_i$$



δ_j can be obtained with a recurrence:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

The full backpropagation algorithm

1. **Forward pass**: stimulate the model with input x, calculate all a_j and z_j up to the output y.
2. Calculate the δ_j for the output units using the derivative of the loss function.
3. **Backward pass**: calculate all δ_j with

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

4. Calculate the gradients with

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \delta_j z_i$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]

Backpropagation in PyTorch

Autograd

In PyTorch (and some other frameworks), `Autograd` performs automatic differentiation through a sequence of tensor instructions of an imperative language.

Let's consider a simple linear operation:

$$w = [\ 5\ 3\ ], \quad x = [\ 7\ 2\ ], \quad y = wx^T$$

The gradient of y w.r.t to x is given as

$$\nabla = \left[\frac{\partial y}{\partial x_i}\right] = \left[\begin{array}{c} 5 \\ 3 \end{array}\right]$$

The gradient of y w.r.t to w is given as

$$\nabla = \left[\frac{\partial y}{\partial w_i}\right] = \left[\begin{array}{c} 7 \\ 1 \end{array}\right]$$

Autograd

In PyTorch, we will first create the tensors:

```
w = torch.tensor([5, 3], dtype=float, requires_grad=True)
x = torch.tensor([7, 1], dtype=float, requires_grad=True)
```

The `requires_grad` flag ensures that all calculations are tracked. We perform the linear operation:

```
y = torch.dot(w,x)
```

Since the tensor y has been calculated as result of operations on tracked tensors, it has a gradient function:

```
print (y)
```

```
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```

Autograd

We now run a backward pass on the variable y, which calculates gradients w.r.t. to all involved tensors:

```
y.backward()
```

The gradients are attached to each variable:

```
print (x.grad)
print (w.grad)
```

```
tensor([5., 3.], dtype=torch.float64)
tensor([7., 1.], dtype=torch.float64)
```

1 Neural Networks & learning

2 Optimization / gradient descent

3 A simple linear example in PyTorch

4 A multi-layer model in PyTorch

5 Convolutional networks

Example: Wisconsin breast-cancer

Id	Cl.thickness	Cell.size	Cell.shape	Marg.adhesion	Epith.c.size	Bare.nuclei	Bl.cromatin	Normal.nucleoli	Mitoses	Class
1000025	5	1	1	1	2	1	3	1	1	benign
1002945	5	4	4	5	7	10	3	2	1	benign
1015425	3	1	1	1	2	2	3	1	1	benign
1016277	6	8	8	1	3	4	3	7	1	benign
1017023	4	1	1	3	2	1	3	1	1	benign
1017122	8	10	10	8	7	10	9	7	1	malignant
1018099	1	1	1	1	2	10	3	1	1	benign
1018561	2	1	2	1	2	1	3	1	1	benign
1033078	2	1	1	1	2	1	1	1	5	benign
1033078	4	2	1	1	2	1	2	1	1	benign
1035283	1	1	1	1	1	1	3	1	1	benign
1036172	2	1	1	1	2	1	2	1	1	benign
1041801	5	3	3	3	2	3	4	4	1	malignant

```
chris pytorch $ head breast-cancer-wisconsin.csv
1000025,5,1,1,1,2,1,3,1,1,2
1002945,5,4,4,5,7,10,3,2,1,2
1015425,3,1,1,1,2,2,3,1,1,2
1016277,6,8,8,1,3,4,3,7,1,2
1017023,4,1,1,3,2,1,3,1,1,2
1017122,8,10,10,8,7,10,9,7,1,4
1018099,1,1,1,1,2,10,3,1,1,2
1018561,2,1,2,1,2,1,3,1,1,2
1033078,2,1,1,1,2,1,1,1,5,2
1033078,4,2,1,1,2,1,2,1,1,2
```

https://www.machinelearningplus.com/machine-learning/logistic-regression-tutorial-examples-r/

Data loading and conversion

```
import numpy as np
from numpy import genfromtxt
import torch
from torch.nn import functional as F

# Import the text file into a numpy array
n = genfromtxt('breast-cancer-wisconsin-cleaned.csv',
    delimiter=',')
D = torch.tensor(n, dtype=torch.float32)
N_samples = D.size(0)

# The input is the full matrix without first and
# last column, plus the 1 column for the bias
X=D[:,1:-1]
X = torch.cat ((X, torch.ones((X.size(0),1))),1)

# The targets. Change all 2->0 and 4->1
T=D[:,-1:]
T[T==2]=0
T[T==4]=1
```

The model

```python
class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()

        # The linear layer (input dim, output dim)
        # It also contains a weight matrix
        # (here single output-> vector)
        self.fc1 = torch.nn.Linear(10, 1)

    # The forward pass of the network. x is the input
    def forward(self, x):
        return F.sigmoid(self.fc1(x))
```

Set up the environment

```
# Instantiate the model
model = LogisticRegression()

# The loss function: binary cross-entropy
criterion = torch.nn.BCELoss()

# Set up the optimizer: stochastic gradient descent
# with a learning rate of 0.01
optimizer = torch.optim.SGD(model.parameters(), lr
    =0.01)
```

Iterative training

```
# 1 epoch = 1 pass over the full dataset
for epoch in range(200):
    print ("Starting epoch", epoch, " ",end='')
    calcAccuracy()

    for sample in range(N_samples):
        # model -> train mode, clear gradients
        model.train()
        optimizer.zero_grad()

        # Forward pass (stimulate model with inputs)
        y = model(X[sample,:])

        # Compute Loss
        loss = criterion(y, T[sample])

        # Backward pass: calculate the gradients
        loss.backward()

        # One step of stochastic gradient descent
        optimizer.step()
```

Evaluation

Calculate the accuracy (in percent) at each epoch:
Proportion of correctly classified samples.
Random performance = 50% on a binary task.

```
def calcAccuracy():

    # model -> eval mode
    model.eval()
    correct = 0.0
    for sample in range(N_samples):

      # threshold the output probability
      y = 1 if model(X[sample,:]) > 0.5 else 0
      correct += (y == T[sample]).numpy()

    print ("Accuracy = ", 100.0*correct/N_samples)
```

Results

```
Starting epoch 0  Accuracy =  [34.69985359]
Starting epoch 1  Accuracy =  [91.21522694]
Starting epoch 2  Accuracy =  [93.99707174]
Starting epoch 3  Accuracy =  [95.16837482]
Starting epoch 4  Accuracy =  [95.75402635]
Starting epoch 5  Accuracy =  [95.60761347]
Starting epoch 6  Accuracy =  [95.75402635]
Starting epoch 7  Accuracy =  [95.75402635]
Starting epoch 8  Accuracy =  [96.33967789]
Starting epoch 9  Accuracy =  [96.48609078]
Starting epoch 10  Accuracy =  [96.63250366]
Starting epoch 11  Accuracy =  [96.63250366]
Starting epoch 12  Accuracy =  [96.63250366]
Starting epoch 13  Accuracy =  [96.63250366]
Starting epoch 14  Accuracy =  [96.63250366]
```

(...)

```
Starting epoch 197  Accuracy =  [97.07174231]
Starting epoch 198  Accuracy =  [97.07174231]
Starting epoch 199  Accuracy =  [97.07174231]
```

What is missing?

- The model is simpler than deep neural networks, but sufficient for the task.
- We did not use batch processing, i.e. using more than one sample for a given gradient update
- We calculated performance on the training set. We might overfit.

1 Neural Networks & learning

2 Optimization / gradient descent

3 A simple linear example in PyTorch

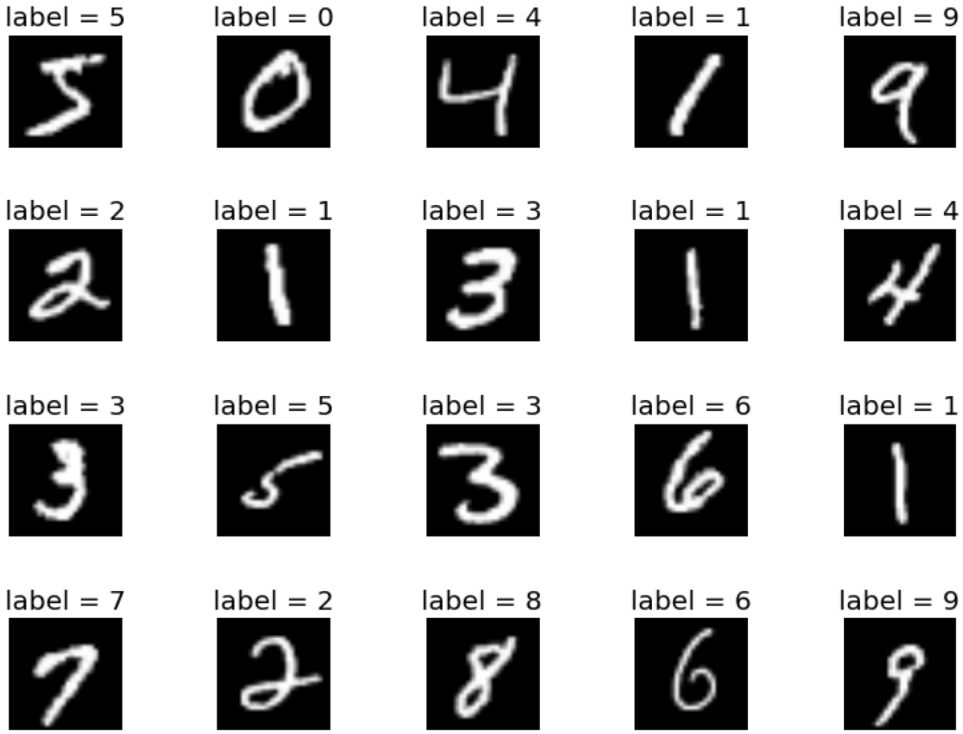
4 A multi-layer model in PyTorch

5 Convolutional networks

Example: the MNIST dataset

A dataset of handwritten digits introduced by Yann LeCun in 1999 with 60 000 training images and 10 000 test images.

One image is of size 28x28 pixels.



http://yann.lecun.com/exdb/mnist/

Writing Data Access

We subclass the `Dataset` class and implement the methods `__getitem__`, `__len__()` (and `__init__`)!). This class will be used by actual PyTorch Dataloader, `DataLoader`.

```
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

class MNISTDataset(Dataset):
    def __init__(self, dir, transform=None):
        # Read in the files from the directory and
        # store them in self.images and self.labels
        # See MOODLE for the full code of this method
        (...)

    # The access is _NOT_ shuffled. The PyTorch
    # Dataloader will need to do this.
    def __getitem__(self, index):
        return self.images[index], self.labels[index]
    # Return the dataset size
    def __len__(self):
        return self.no_images
```

Writing Data Access

Let's recall training through gradient descent:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}(h(x, \theta), y^*)$$

The gradient is rarely (never?) taken over the whole dataset, but over a single sample, or batches (mini-batches) of a certain size. These batches are sample randomly from the dataset.

The actual shuffling and batching is performed by a built-in PyTorch `DataLoader` class, which uses an instance of our `Dataset` subclass:

```
dataset = MNISTDataset ("MNIST-png/testing",
    transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))]))

loader = torch.utils.data.DataLoader(dataset,
    batch_size=50, shuffle=True)
```

We passed a set of image transforms to the Dataset class, which applies them to each image.

Tensor dimension conventions

PyTorch functions operate on multi-dimensional tensors and follow conventions on the order of dimensions.

- The first dimension is the batch dimension
  - Use 1 if you don't use batches (= batches of size 1).
  - Losses are reduced (sum or mean) over samples in a batch
- the second dimension is the channel dimension
  - Use 1 if you don't use channels (= single channels).
  - Channel arithmetics will be explained in detail in the section on convolutions.
- the following dimensions are application dependant, e.g. rows, columns in images.

The 2 layer model

The model is similar to our linear example. Differences:

- A hidden layer with 300 units and relu activation.
- A forward pass deals with a full batch.

```
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # input size to 300 units
        self.fc1 = torch.nn.Linear(28*28, 300)
        # 300 units to 10 output classes
        self.fc2 = torch.nn.Linear(300, 10)

    def forward(self, x):
        # Reshape from a 3D tensor (batchsize, 28, 28)
        # to a flattened (batchsize, 28*28)
        # 1 sample = 1 vector
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Setup model, loss, optimizer

This time we have more than 2 classes, we use the Cross-entropy loss.

```
# Instantiate the model
model = MLP()

# This criterion combines LogSoftMax and NLLLoss
# in one single class.
crossentropy = torch.nn.CrossEntropyLoss()

# Set up the optimizer: stochastic gradient descent
# with a learning rate of 0.01
optimizer = torch.optim.SGD(model.parameters(), lr
    =0.01)

# Init some statistics
running_loss = 0.0
running_correct = 0
running_count = 0
```

Cycling through batches and samples

```
# Cycle through epochs
for epoch in range(100):

    # Cycle through batches. One batch is a set of
    # images and a set of ground truth labels
    for batch_idx, (data, labels) in enumerate(loader):

        optimizer.zero_grad()

        # We calculate a prediction on the full batch
        y = model(data)

        # Loss of the full batch, summed
        loss = crossentropy(y, labels)

        # Calculate gradients of the full batch
        loss.backward()

        # One gradient update
        optimizer.step()
```

Track training error

```
    # Calculate the winner class
    _, predicted = torch.max(y.data, 1)

    # How many correct samples?
    running_correct += (predicted == labels).sum().item()
    running_count += BATCHSIZE

    # Every 100 batches, print statistics
    if (batch_idx % 100) == 0:

    train_err = 100.0*(1.0-running_correct / running_count)
    print ('Epoch: %d batch: %5d ' % (epoch + 1, batch_idx +
     1), end="")
    print ('train-loss: %.3f train-err: %.3f' % (
    running_loss / 100, train_err))
    running_loss = 0.0
    running_correct = 0.0
    running_count=0.0
```

Example output

```
Epoch: 1 batch:    1 train-loss: 0.024 train-err: 98.000
Epoch: 1 batch:  101 train-loss: 1.576 train-err: 36.140
Epoch: 1 batch:  201 train-loss: 0.747 train-err: 16.320
Epoch: 1 batch:  301 train-loss: 0.539 train-err: 12.880
Epoch: 1 batch:  401 train-loss: 0.481 train-err: 13.000
Epoch: 1 batch:  501 train-loss: 0.414 train-err: 11.280
Epoch: 1 batch:  601 train-loss: 0.386 train-err: 10.380
Epoch: 1 batch:  701 train-loss: 0.385 train-err: 10.900
Epoch: 1 batch:  801 train-loss: 0.363 train-err: 10.540
Epoch: 1 batch:  901 train-loss: 0.320 train-err: 9.120
Epoch: 1 batch: 1001 train-loss: 0.323 train-err: 8.920
Epoch: 1 batch: 1101 train-loss: 0.325 train-err: 9.400
Epoch: 2 batch:    1 train-loss: 0.304 train-err: 8.880
```

(...)

```
Epoch: 75 batch:  801 train-loss: 0.007 train-err: 0.000
Epoch: 75 batch:  901 train-loss: 0.007 train-err: 0.020
Epoch: 75 batch: 1001 train-loss: 0.008 train-err: 0.000
Epoch: 75 batch: 1101 train-loss: 0.009 train-err: 0.040
```

This is training error, not validation error, i.e. **NOT** representative of the performance of the model!

1 Neural Networks & learning

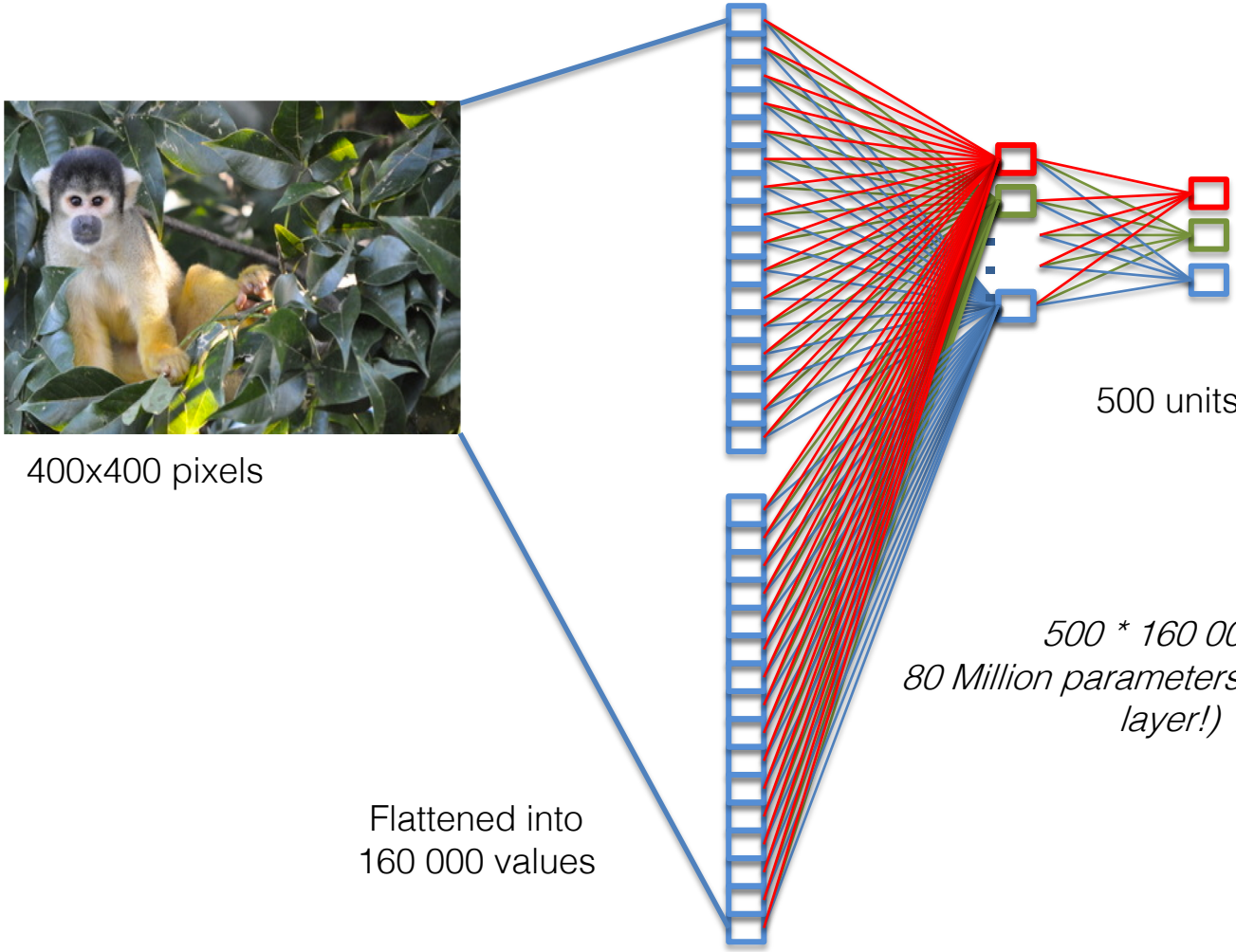
2 Optimization / gradient descent

3 A simple linear example in PyTorch

4 A multi-layer model in PyTorch

5 Convolutional networks

Fully connected layers might be harmful



*500 * 160 000 =*
80 Million parameters (for a single layer!)

Fully connected layers might be harmful

Parameters learned for one part of the image do not generalize to other parts of the image.

Shift-invariant linear operations

Classical MLPs are sequences of linear layers followed by point-wise non-linearities. What kind of linear and shift-invariant operators can there exist?

Let's consider an operator $\varnothing$ with following properties:

Linearity:

$$\phi\left(\alpha f + \beta f'\right) = \alpha\phi(f) + \beta\phi\left(f'\right)$$

Shift-invariance:

$$\phi\left({}^{m,n}S(f)\right) = {}^{m,n}S(\phi(f))$$

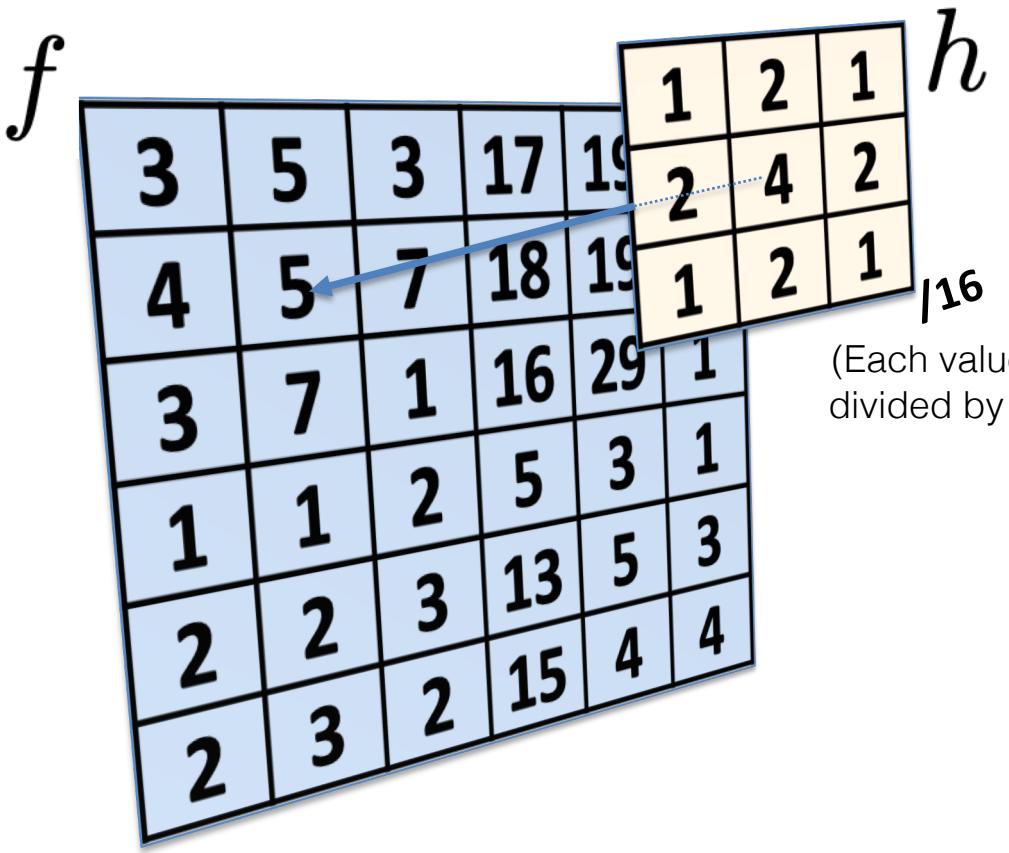
where ${}^{m,n}S(f)$ shifts a signal by ^{m,n}.

Impulse response h:

$$h = \phi({}^{0,0}p)$$

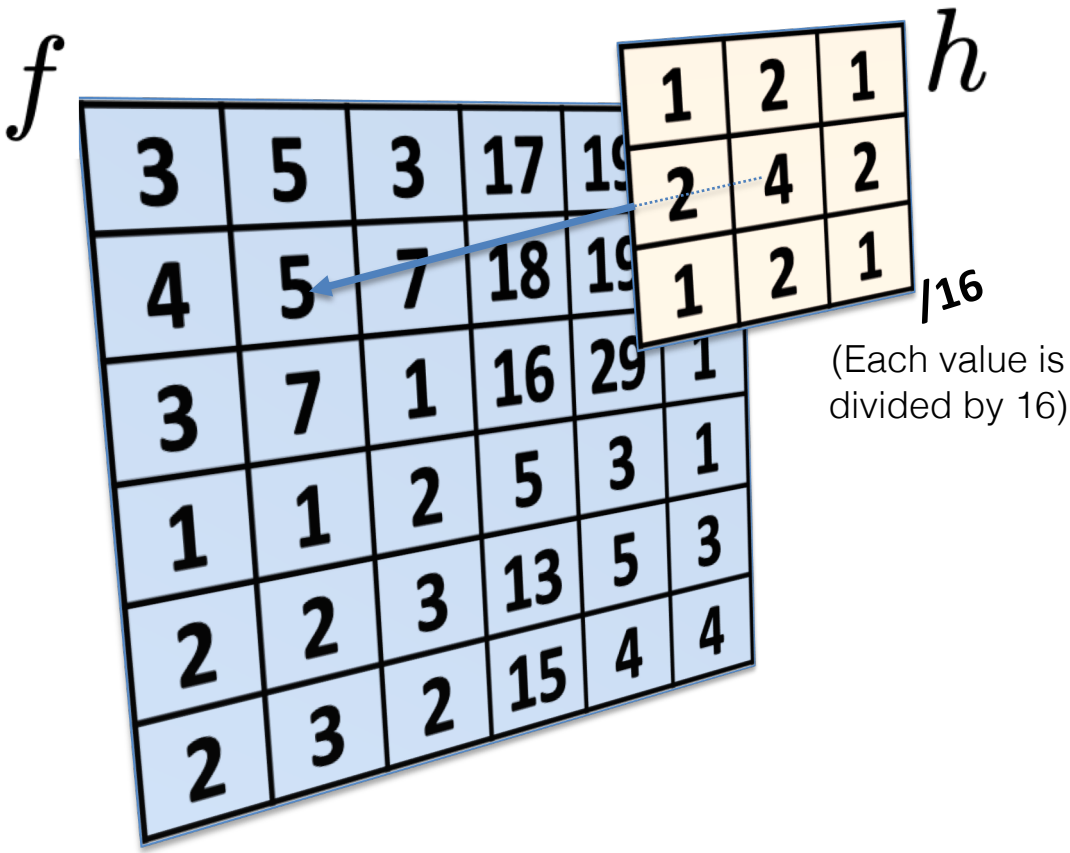
where ${}^{0,0}p$ is a Dirac impulse centered at $0, 0$.

Convolutions

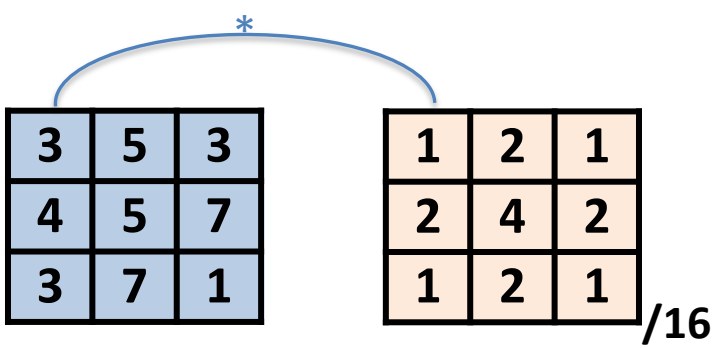


(Each value is divided by 16)

Convolutions



(Each value is divided by 16)



(3+2*5+3+2*4+5*5+2*7 +3+2*7+1)/16 = 5

Convolutions

f

3	5	3	17	19	
4	5	7	18	19	
3	7	1	16	29	1
1	1	2	5	3	1
2	2	3	13	5	3
2	3	2	15	4	4

h

1	2	1
2	4	2
1	2	1

/16

(Each value is divided by 16)

$\phi(f)$

	5				

*

3	5	3
4	5	7
3	7	1

1	2	1
2	4	2
1	2	1

/16

(3+2*5+3+2*4+5*5+2*7 +3+2*7+1)/16 = 5

Convolutions

f

3	5	3	17	19	15
4	5	7	18	19	14
3	7	1	16	29	1
1	1	2	5	3	1
2	2	3	13	5	3
2	3	2	15	4	4

h

1	2	1
2	4	2
1	2	1

/16

(Each value
divided by 1

$\phi(f)$

	5				

$*$

5	3	17
5	7	18
7	1	16

1	2	1
2	4	2
1	2	1

/16

Convolutions

f

3	5	3	17	19	15
4	5	7	18	19	14
3	7	1	16	29	1
1	1	2	5	3	1
2	2	3	13	5	3
2	3	2	15	4	4

h

1	2	1
2	4	2
1	2	1

/16

$\phi(f)$

	5	7			

*

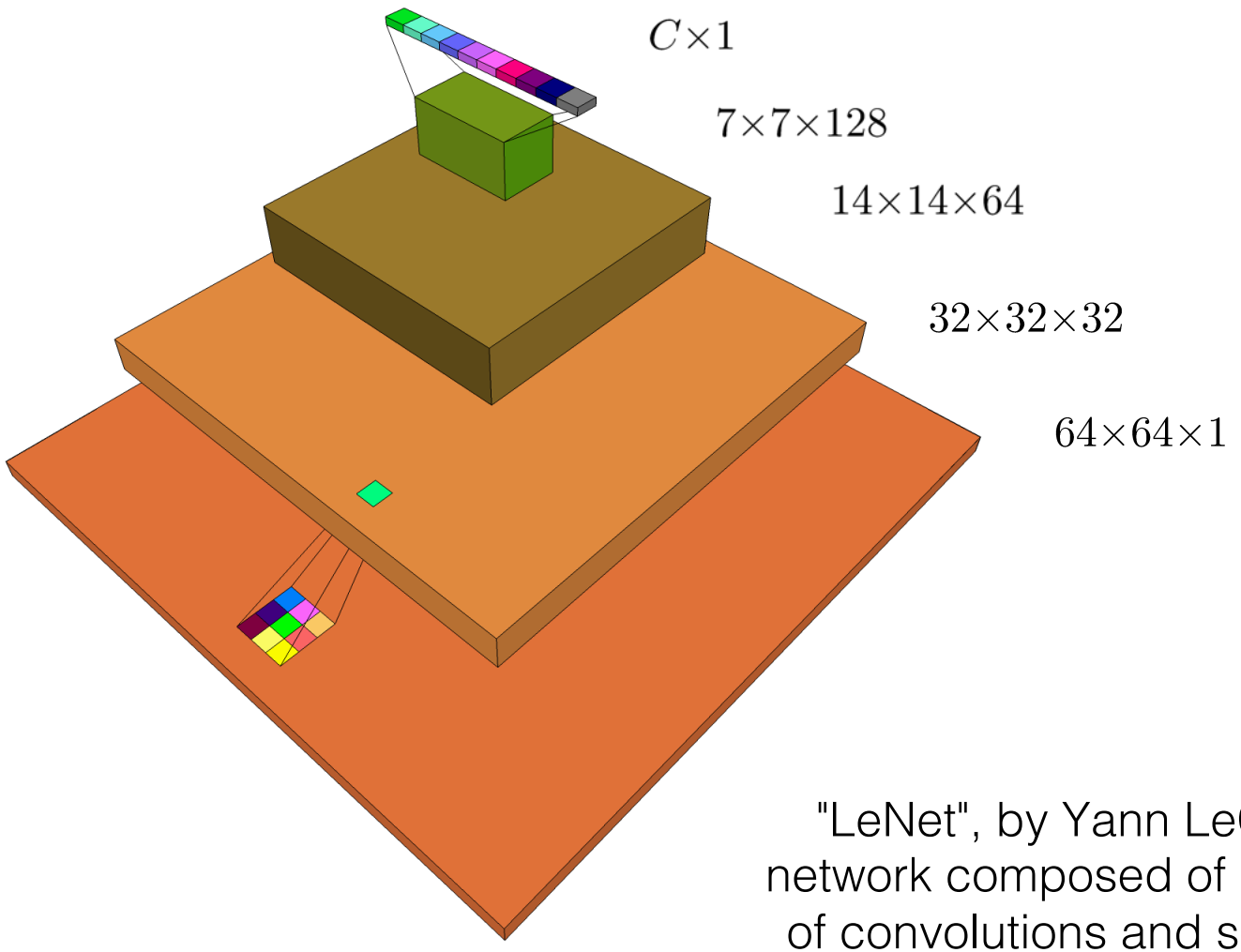
5	3	17
5	7	18
7	1	16

1	2	1
2	4	2
1	2	1

/16

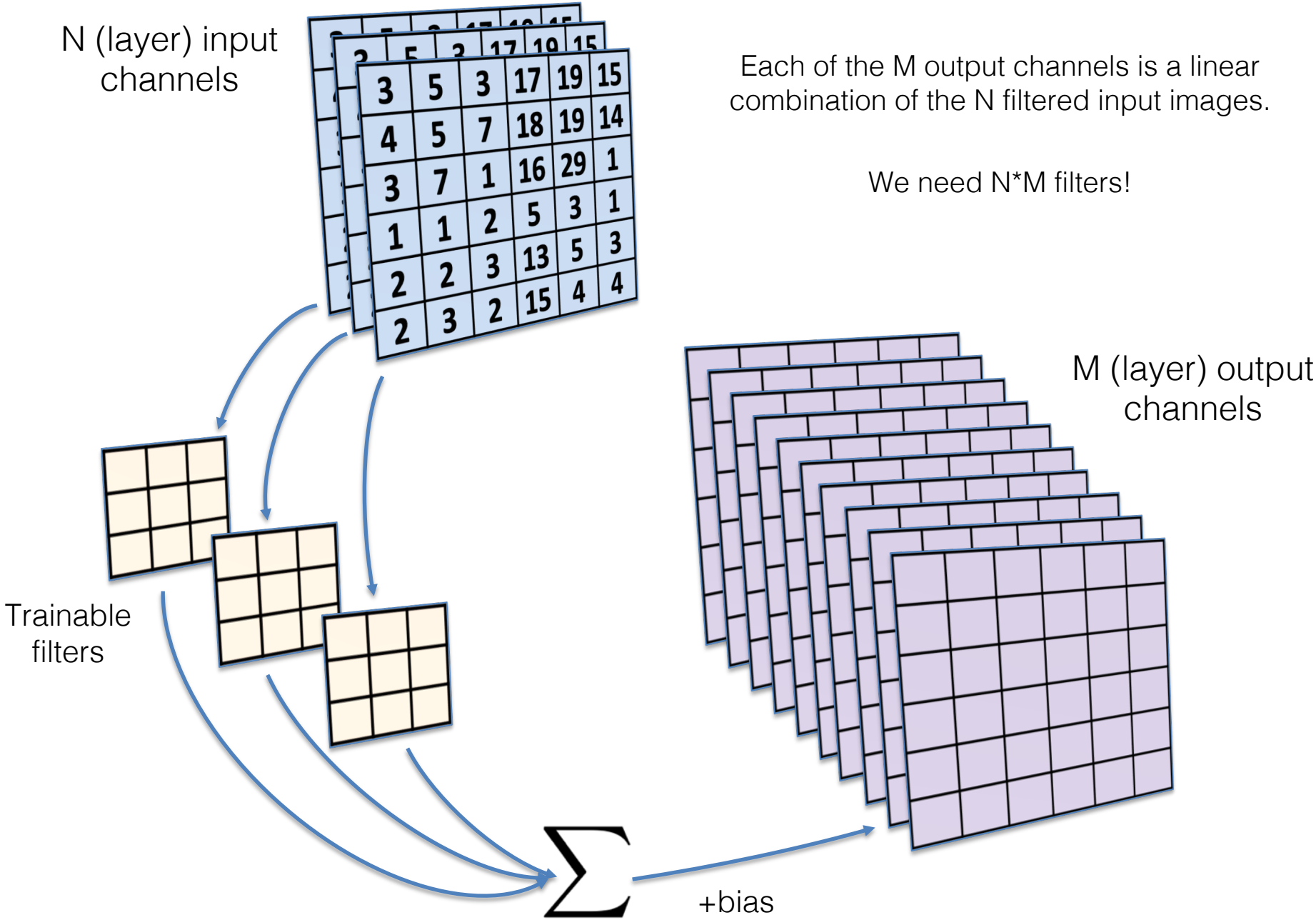
(5+2*3+17+2*5+4*7+2*18+
7+2*1+16)/16 = 7

« LeNet »



"LeNet", by Yann LeCun, is a network composed of sequences of convolutions and spatial size reductions (« pooling »).

[LeCun et al., *1998*]

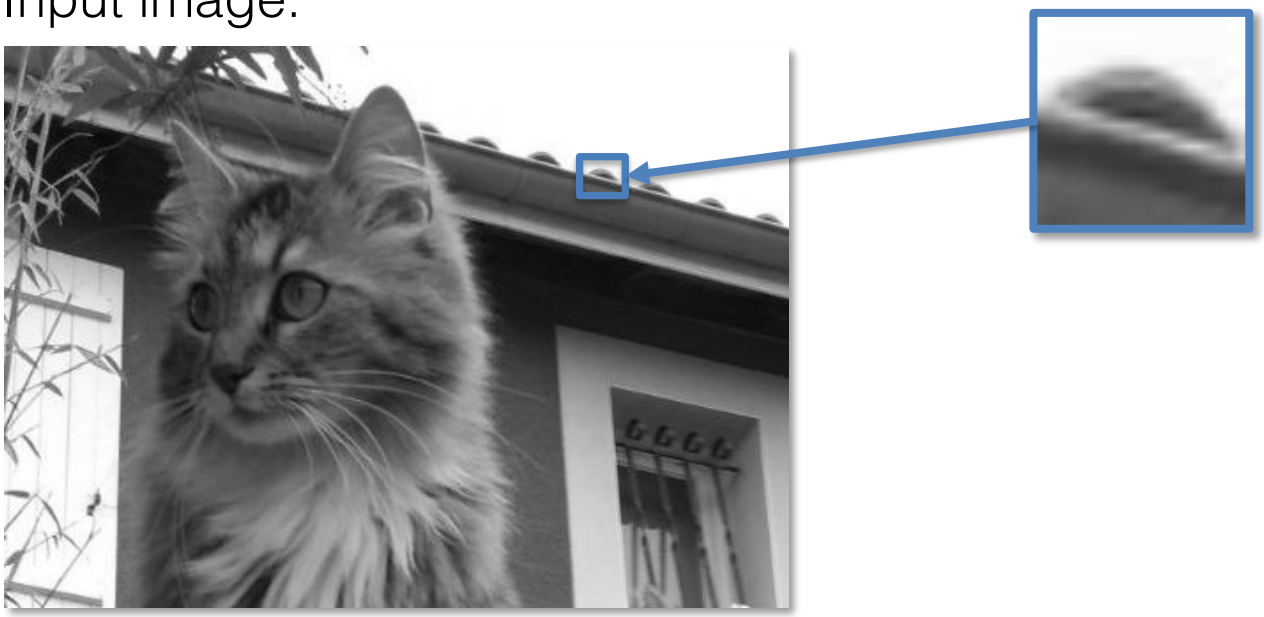


Each of the M output channels is a linear combination of the N filtered input images.

We need N*M filters!

Template matching

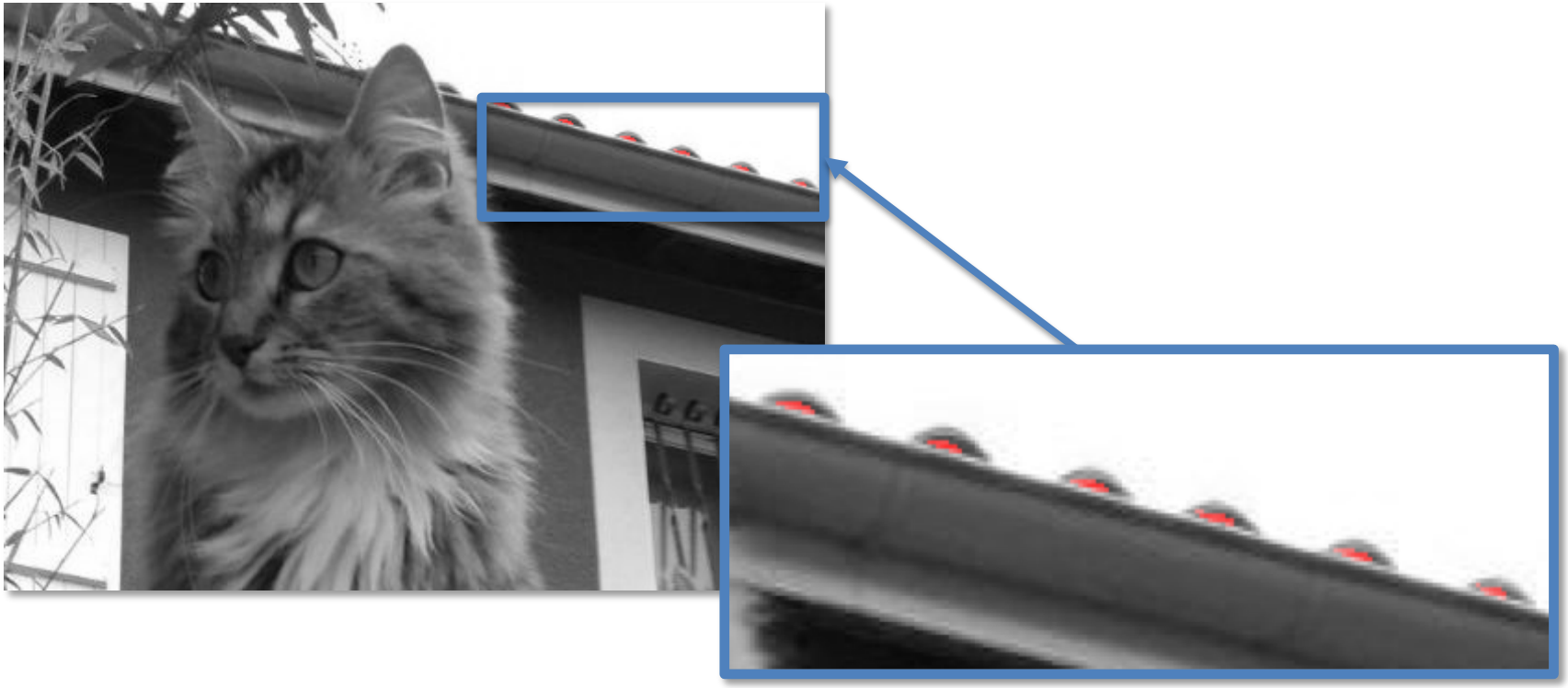
Input image:



Filter kernel

- We normalize input and kernel (subtract mean, divide by standard deviation).
- We convolve the input with the kernel
- We threshold response and superimpose it on the red channel.

Template matching



Convolutions can match patterns:

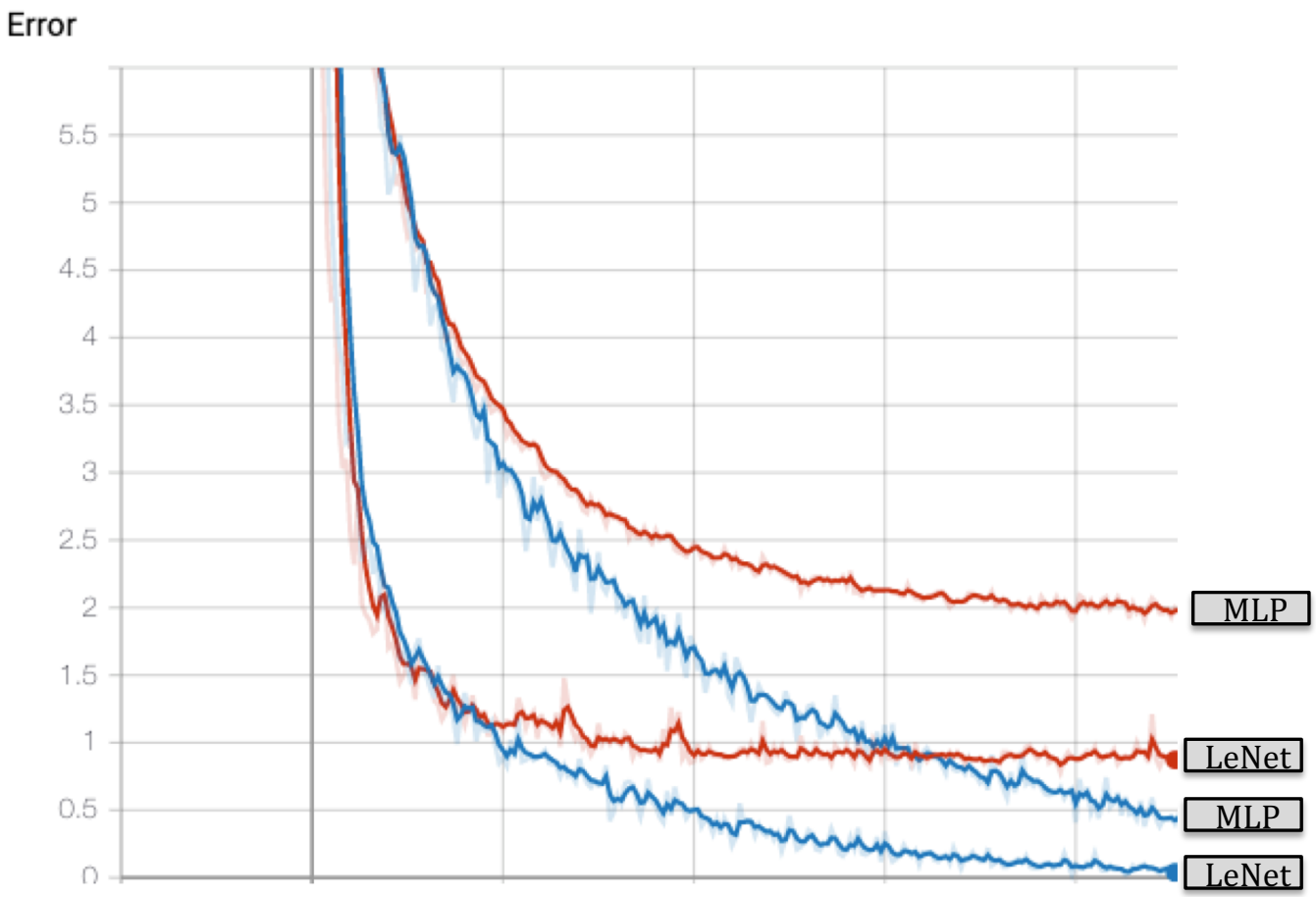
- either in input space
- or in intermediate network layers (which still have a spatial organization).

LeNet in PyTorch

```
class LeNet(torch.nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 1 input channel, 20 output channels,
        # 5x5 filters, stride=1, no padding
        self.conv1 = torch.nn.Conv2d(1, 20, 5, 1, 0)
        self.conv2 = torch.nn.Conv2d(20, 50, 5, 1, 0)
        self.fc1 = torch.nn.Linear(4*4*50, 500)
        self.fc2 = torch.nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        # Max pooling with a filter size of 2x2
        # and a stride of 2
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

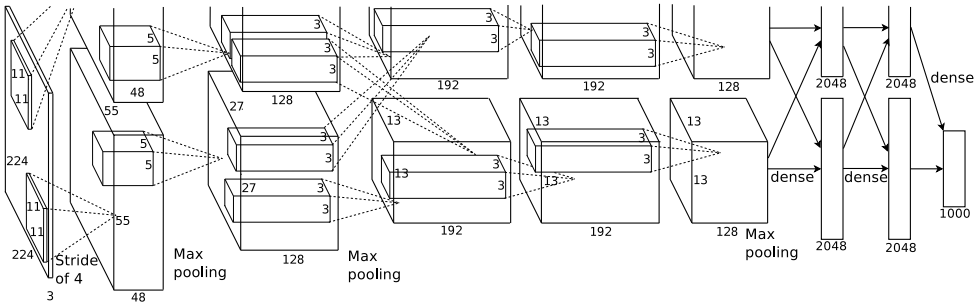
MLP vs LeNet on MNIST



Blue: training, Red: validation

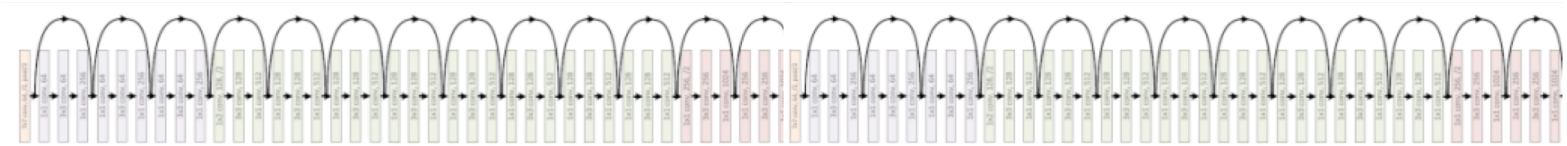
Going deeper

2012 : AlexNet, 8 layers. New techniques: dropout, ReLU

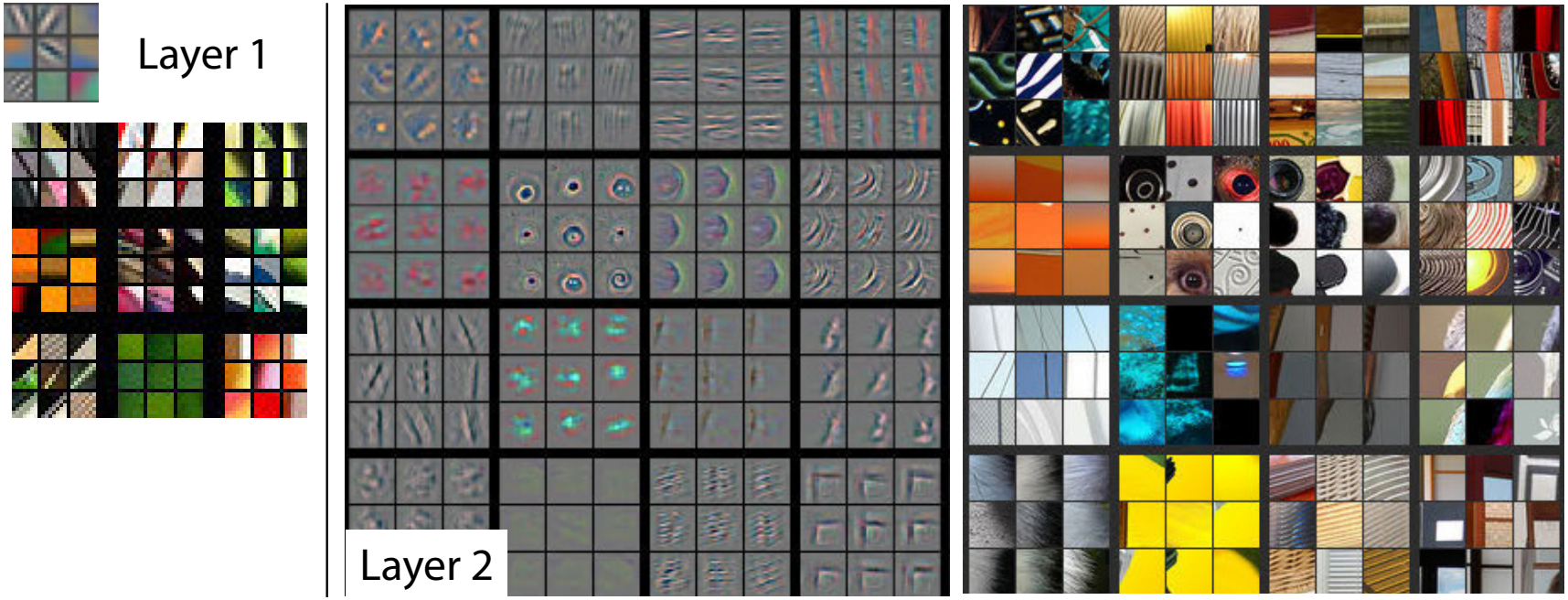


2014 : GoogLEnet, 20 layers. New technique: intermediate supervision

2015 : Microsoft research, 150 layers. New technique: residual learning

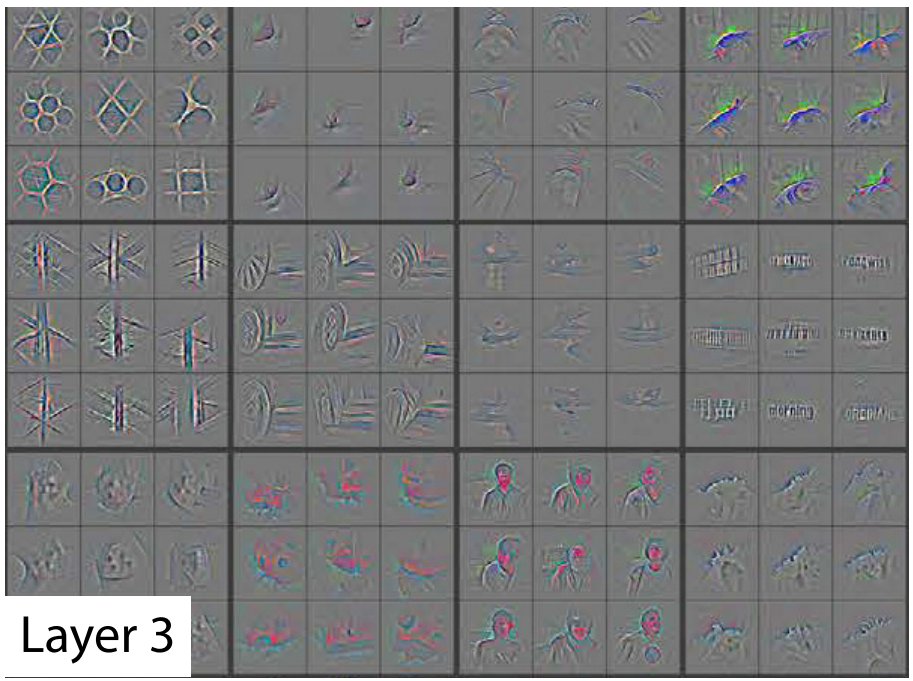


Visualizing feature map activations



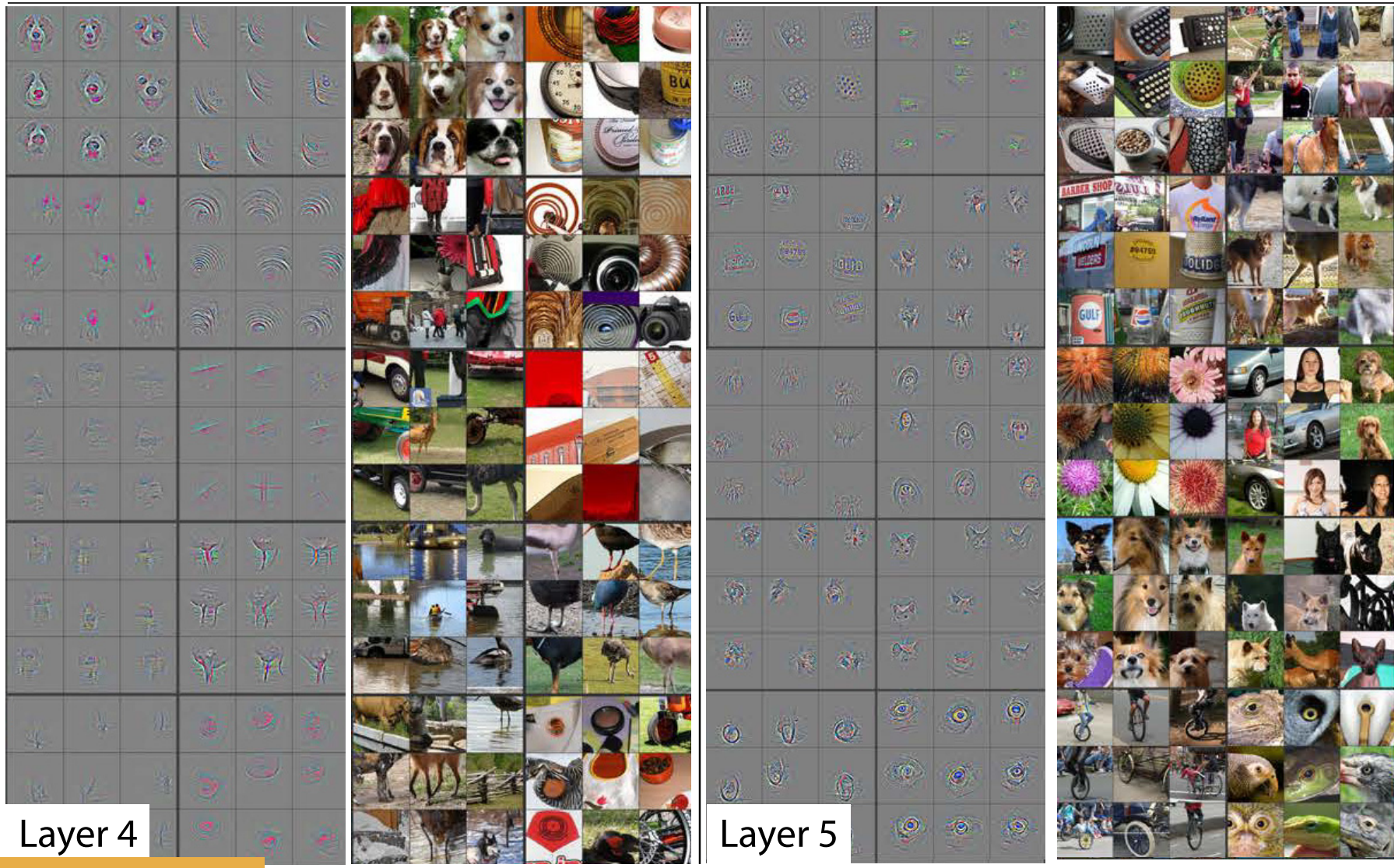
[Zeiler and Fergus, ECCV 2014]

Visualizing feature map activations



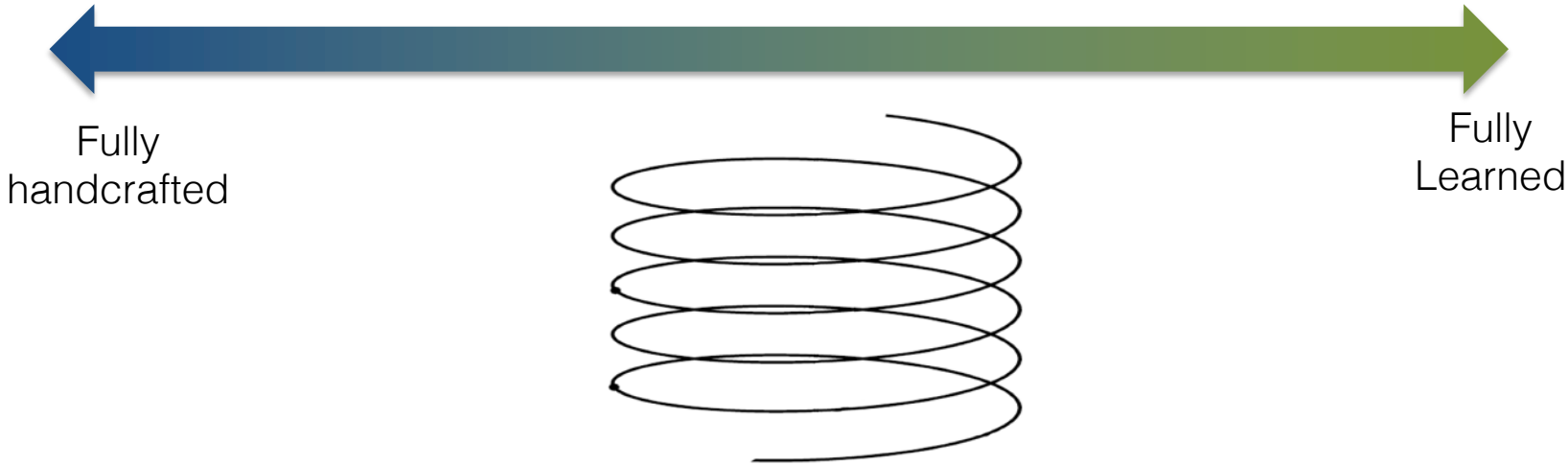
[Zeiler and Fergus, ECCV 2014]

Visualizing feature map activations



[Zeiler and Fergus, ECCV 2014]

Conclusion



Problems to solve:

- Expressivity: model symmetries and invariances in your data
- Trainability
- Generalization

1960-2012: let's handcraft features
2012-2016: let's handcraft architectures (layers, units ...)
2016- ?: let's handcraft data-flow (geometry, causality, attention)