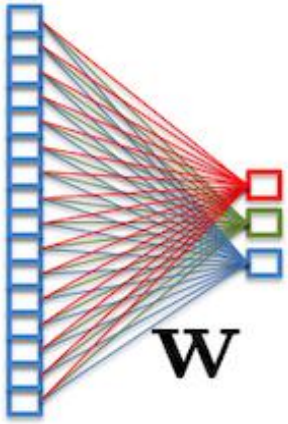
AI and Data Analysis

2.1 Multi layer models



Christian Wolf

The problems of linear models

Lack of capacity, i.e. complexity of the decision to learn.

Example : XOR

Example : Visual Question Answering

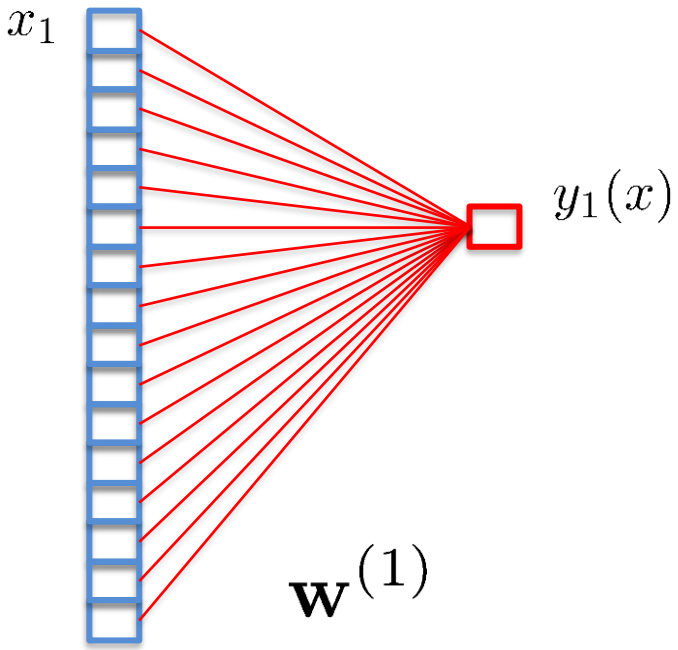
Example: Visual Question Answering

"What is the moustache made of?"

Can the answer be predicted as a linear combination of input pixels and question words ?

Reminder: linear models

$$y(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=0}^{D} \boldsymbol{w}_i \boldsymbol{x}_i$$

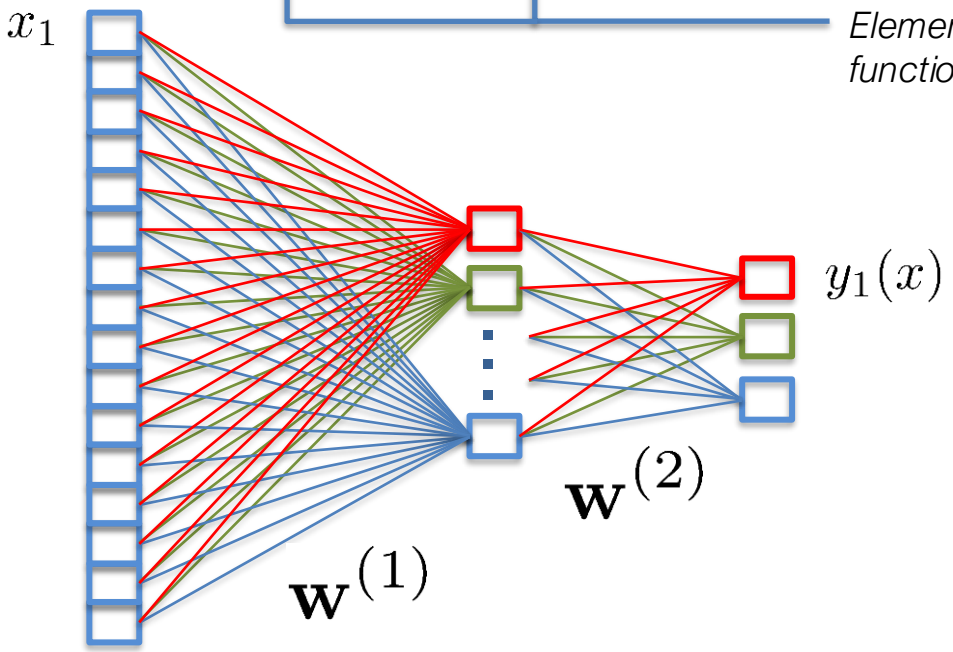


Multi-layer Perceptron (MLP)

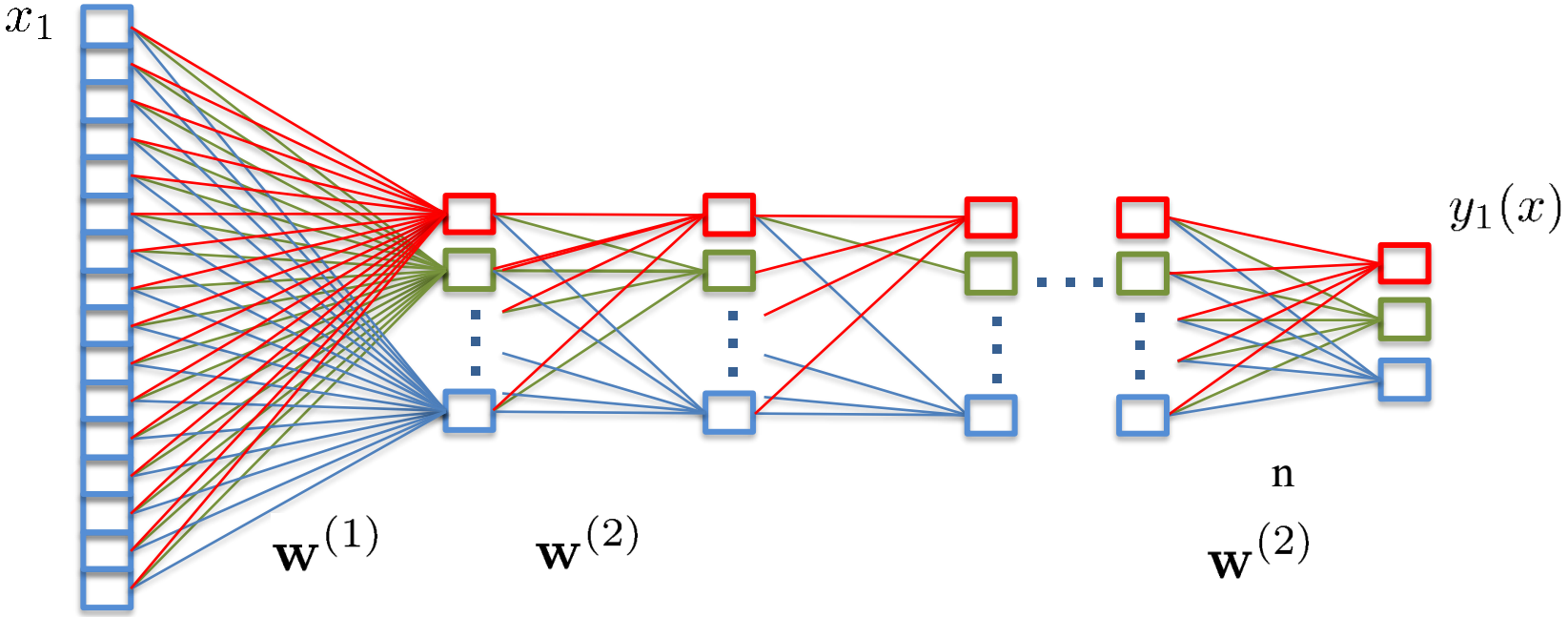
« Fully-connected » layers

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

Elementwise activation functions



Deep neural network



Linear activations

Consider a network with a single hidden layer and two activation functions ζ and σ:

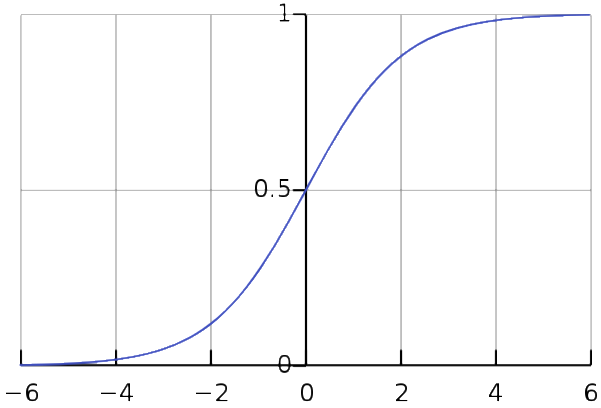
$$y_k(x) = \sigma\left(\sum_j w_{kj}^{(2)} \zeta\left(\sum_i w_{ji}^{(1)} x_i\right)\right)$$

If the activation functions are linear, then we can rewrite the equations in the following vectorial notation:

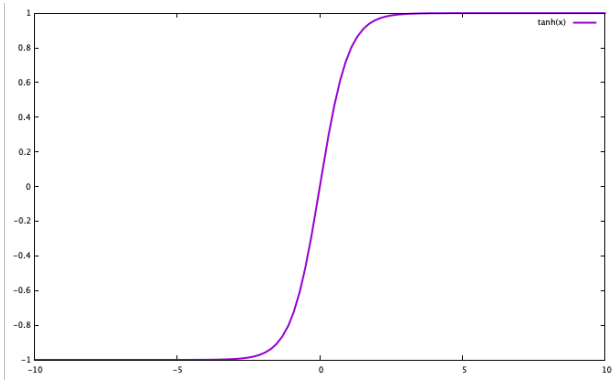
$$\begin{aligned} y &= (\sigma . W^{(2)})(\zeta . W^{(1)} x) \\ &= W^{(2)'}(W^{(1)'} x) \\ &= W'' x \end{aligned}$$

Linear activation functions result in linear models!
$\rightarrow$ The activation functions between layers should be non-linear.

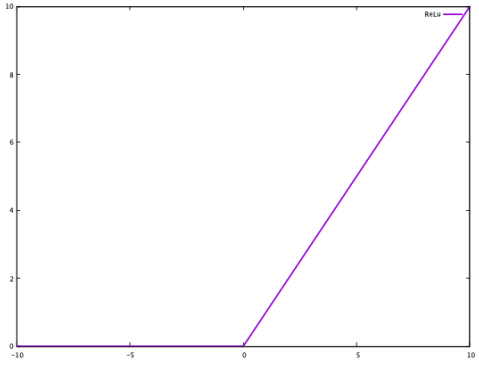
Common activation functions



Logistic function
Sigmoid



Hyperbolic tangent
tanh



Rectified Linear Unit
ReLU

Universal approximation

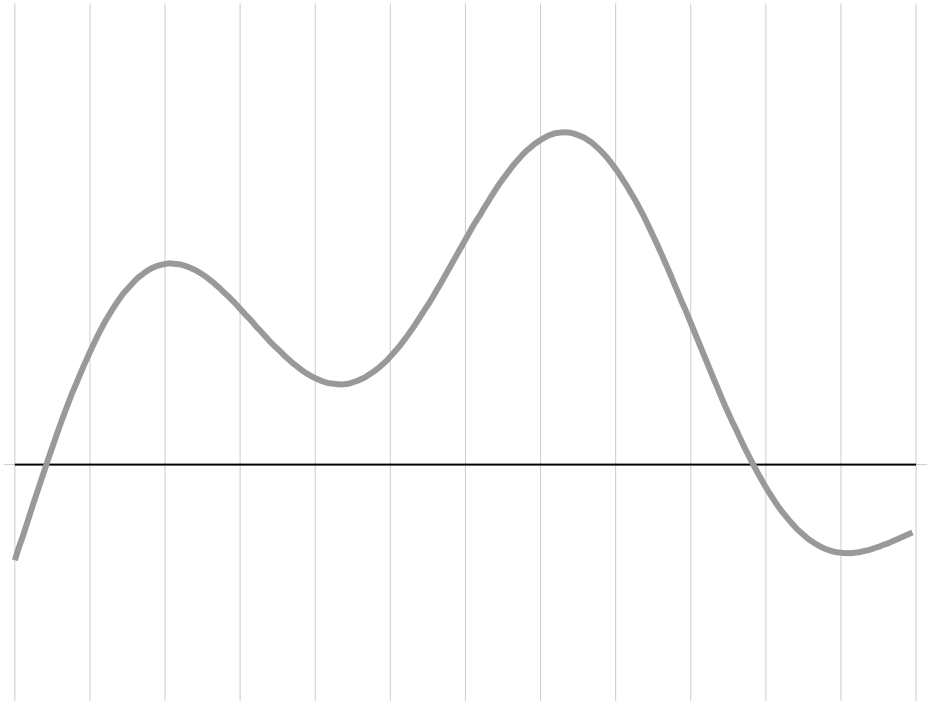
A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation function.

Single hidden layer require an exponential number of hidden units (width).

Since 2017 we know that width bounded networks can approximate any function (mild conditons) if depth can grow.

Lu, Z., Pu, H., Wang, F., Hu, Z., & Wang, L. The Expressive Power of Neural Networks: A View from the Width, NeurIPS, 2017.

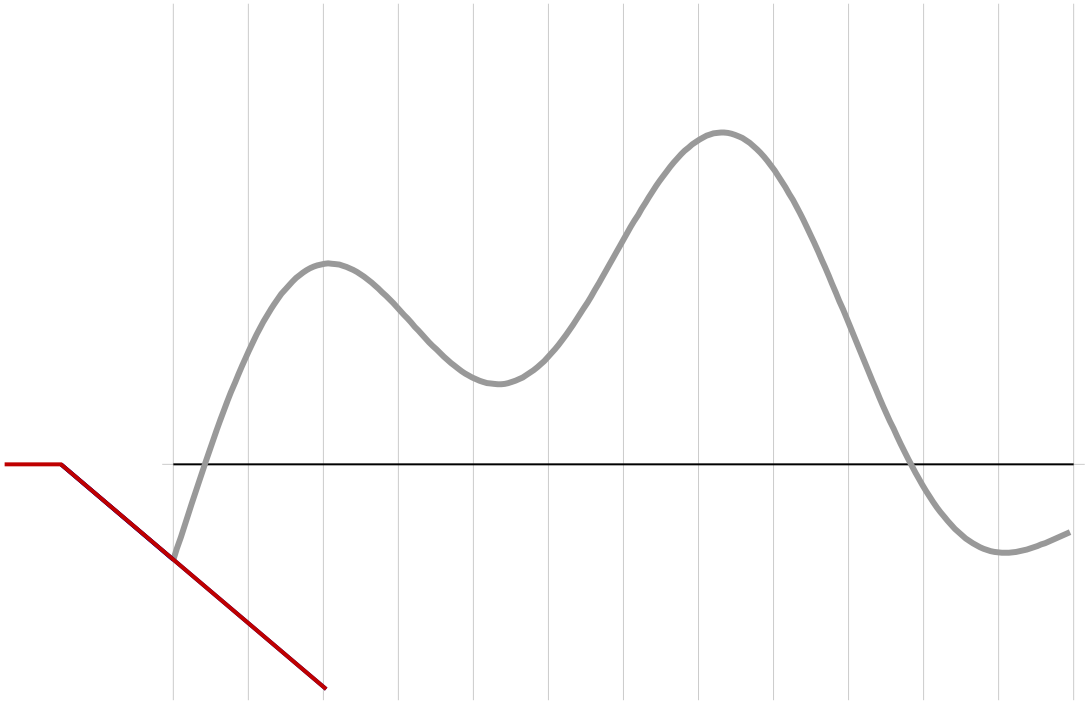
We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.



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.

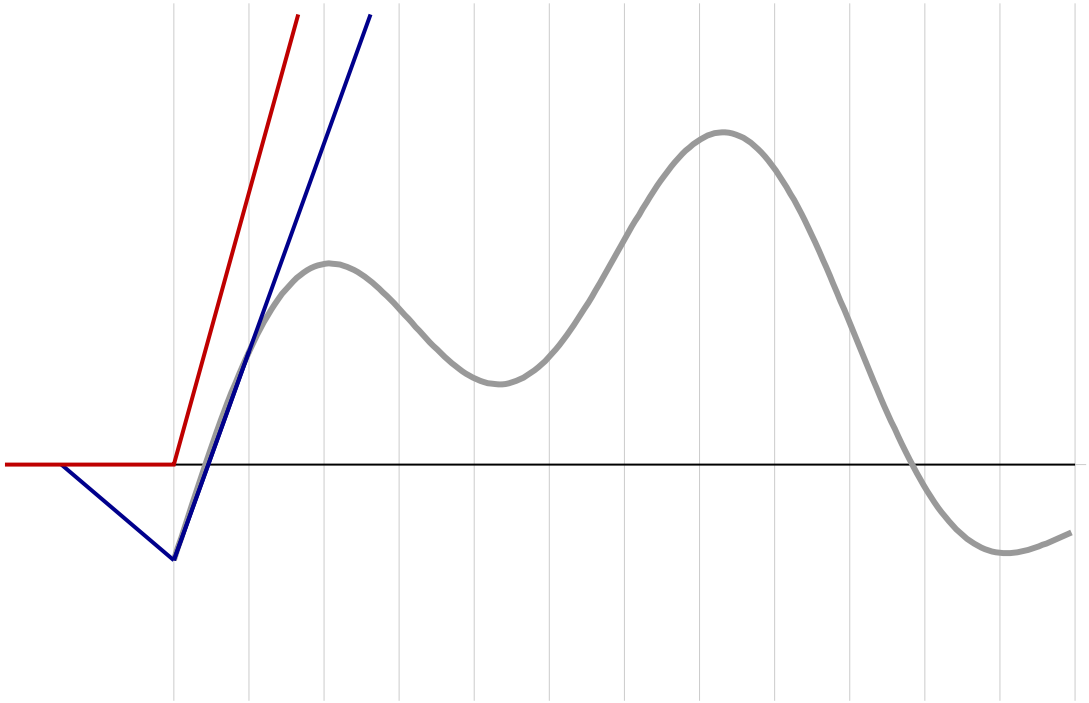
$$f(x) = \sigma(w_1 x + b_1)$$



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.

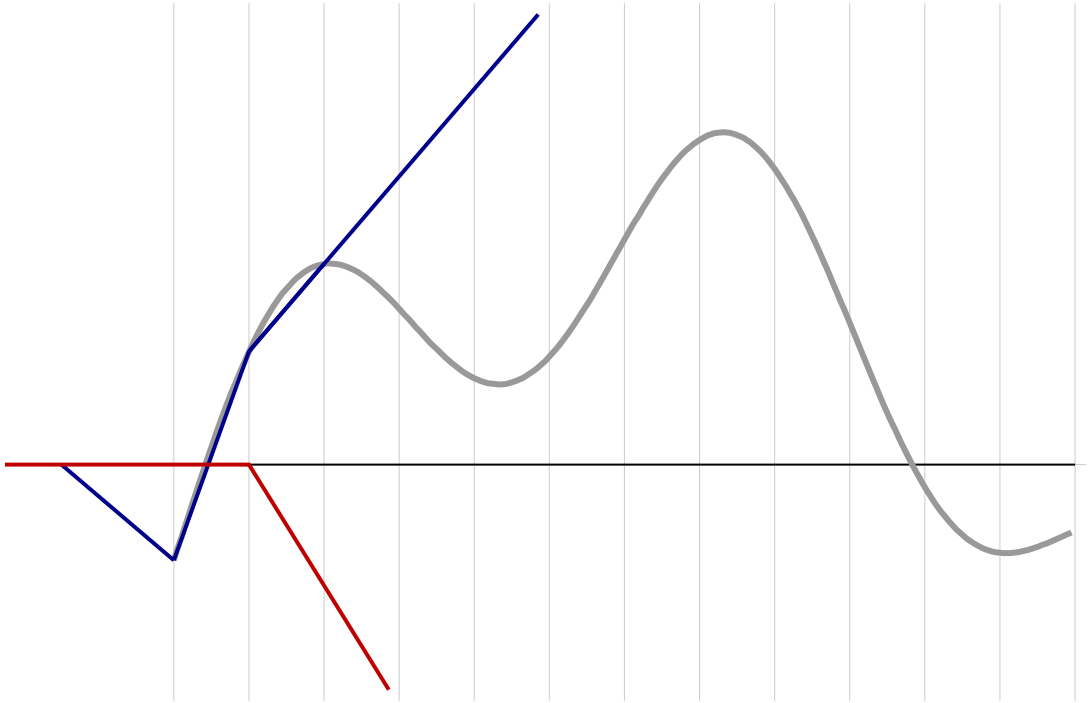
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$$



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.

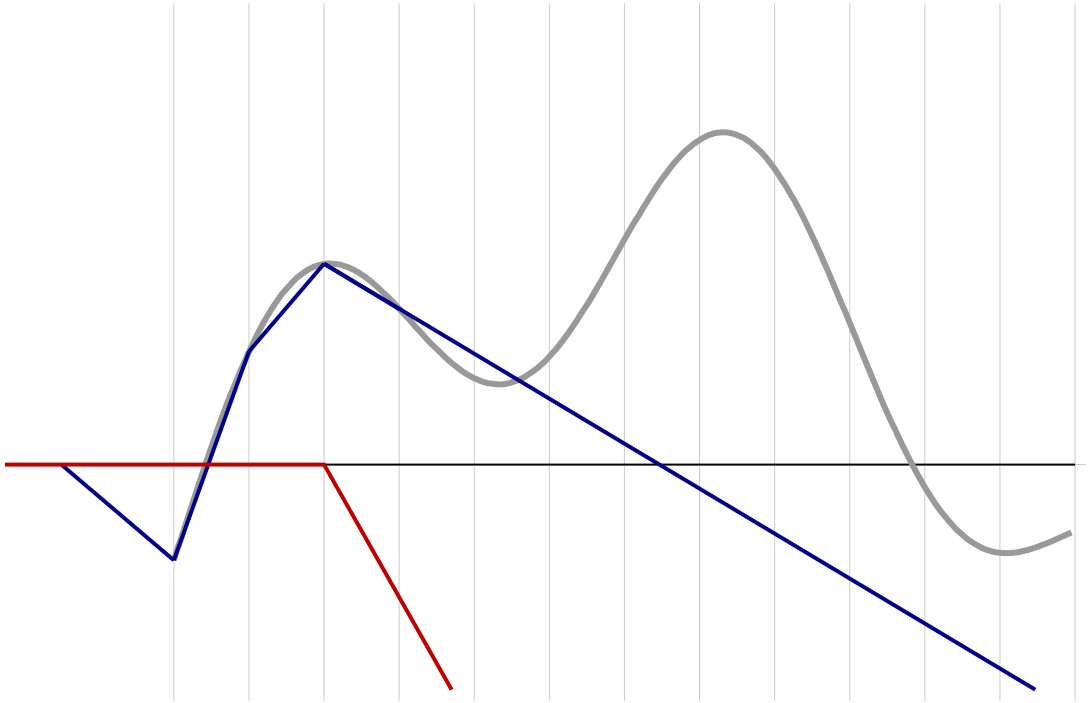
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3)$$



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.

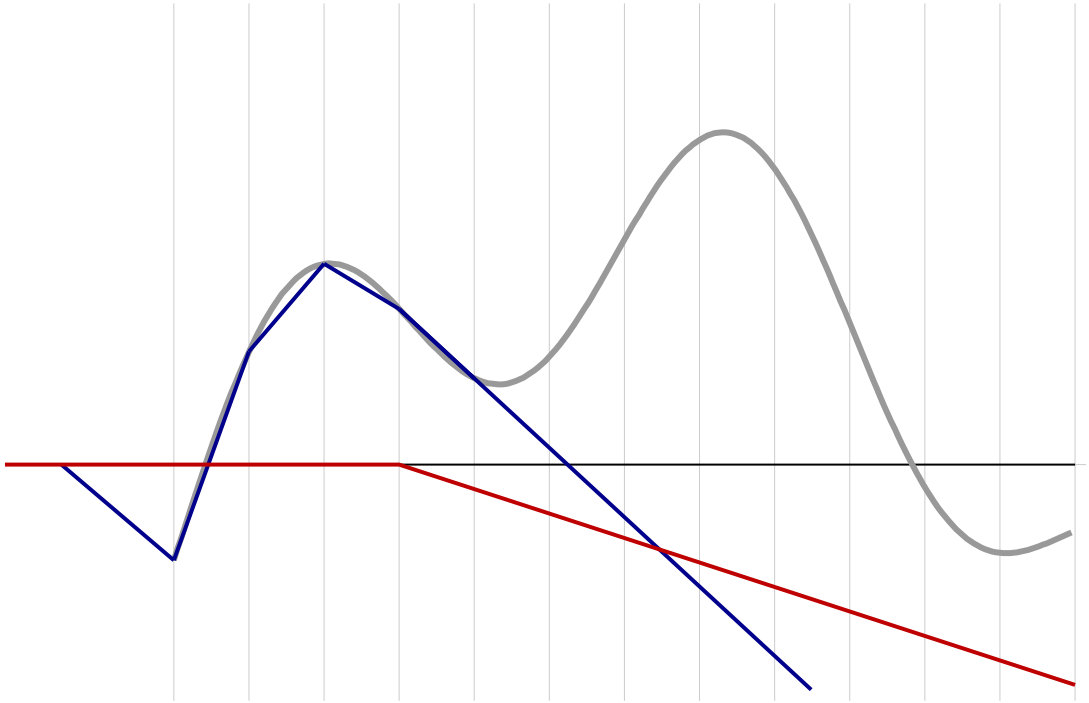
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

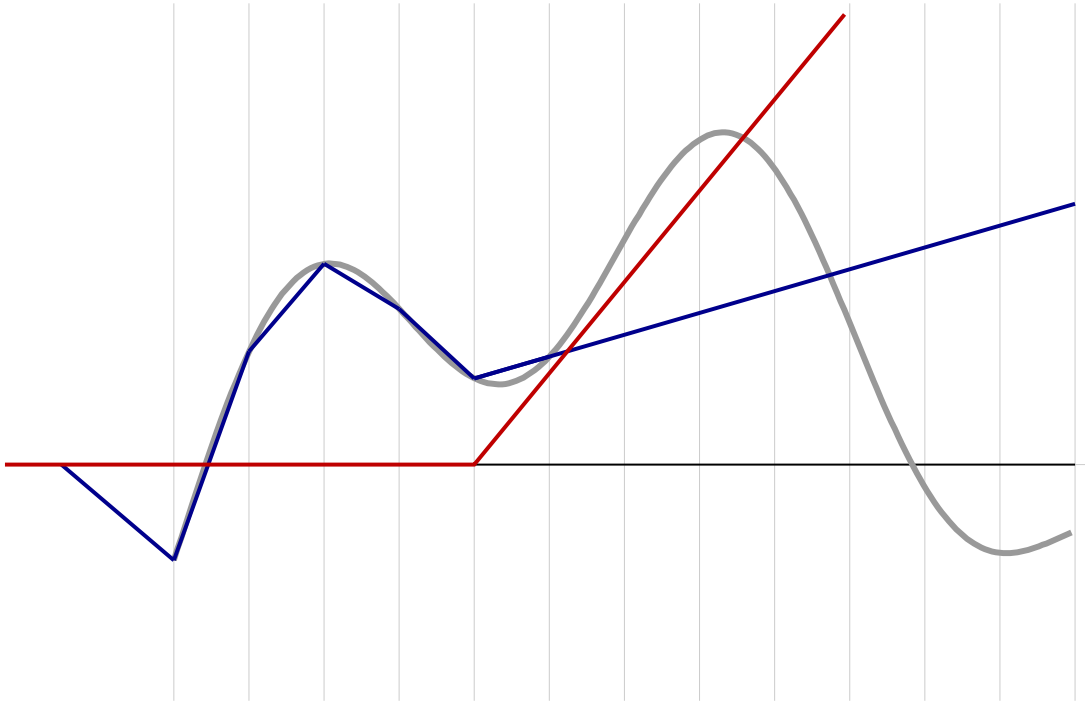
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

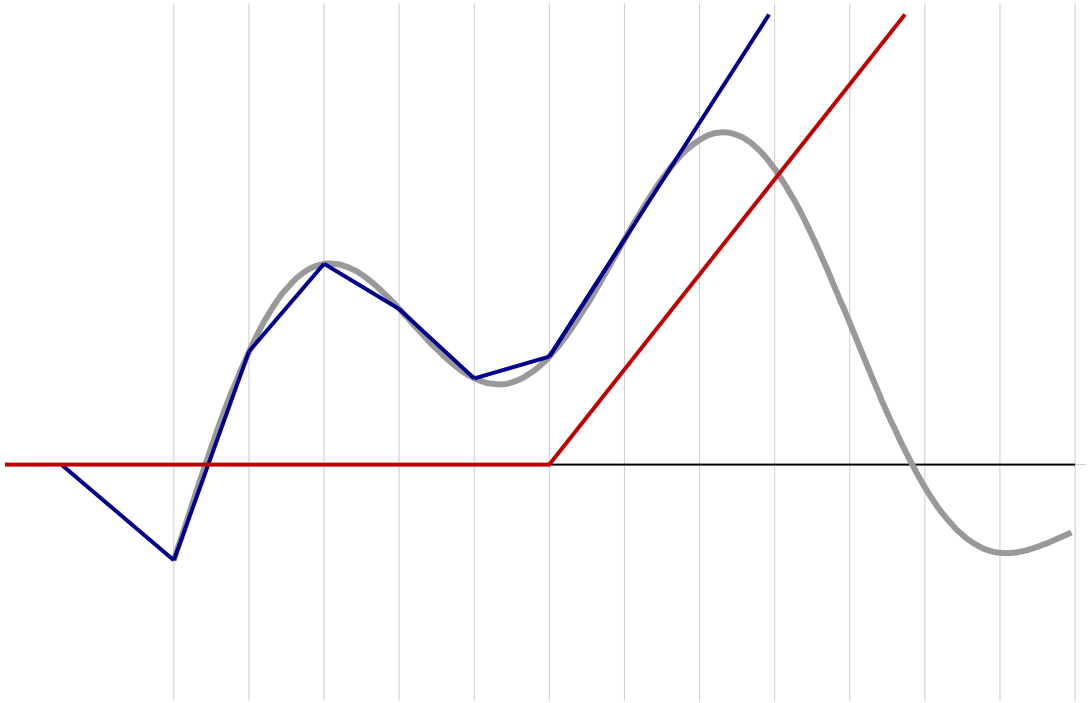
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

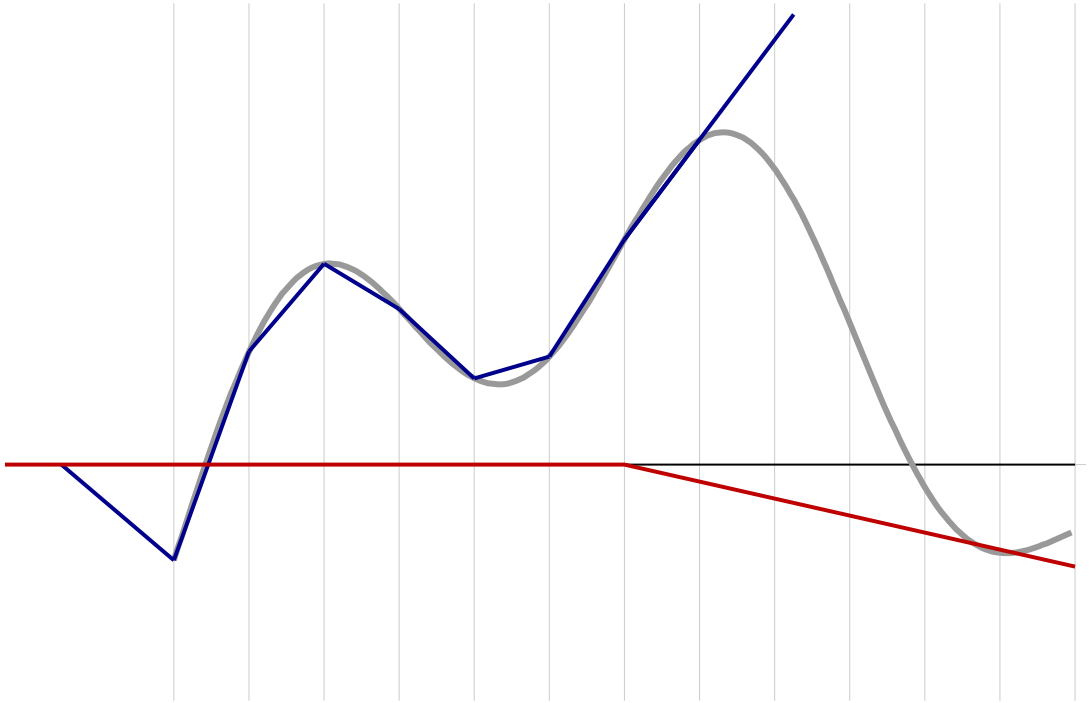
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

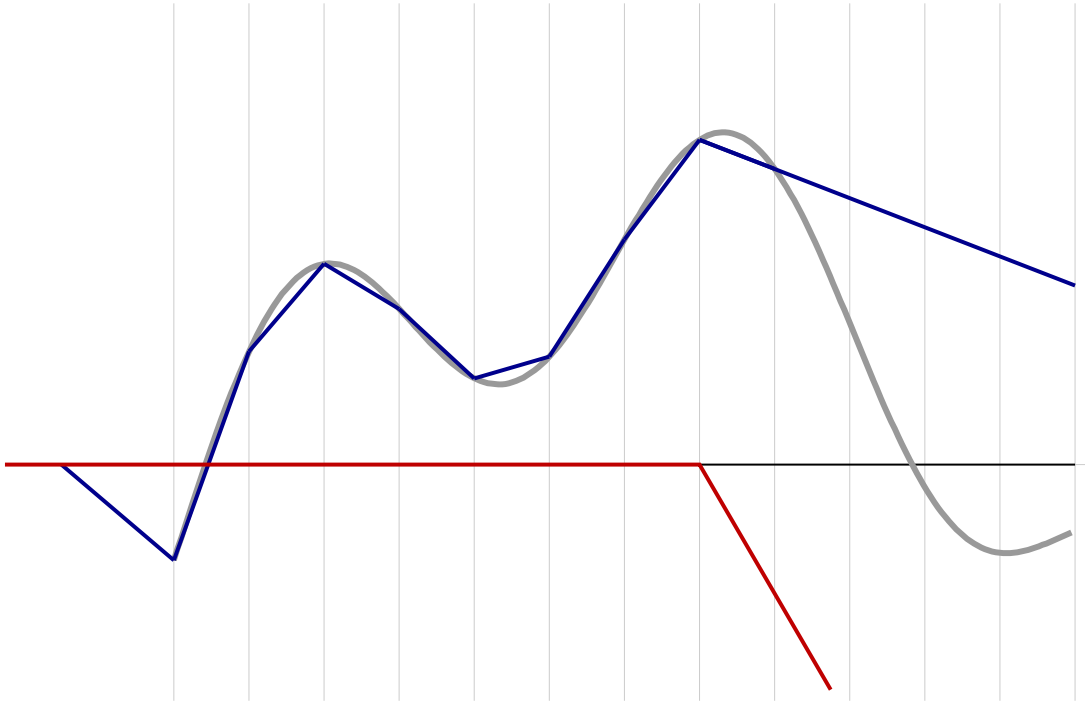
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

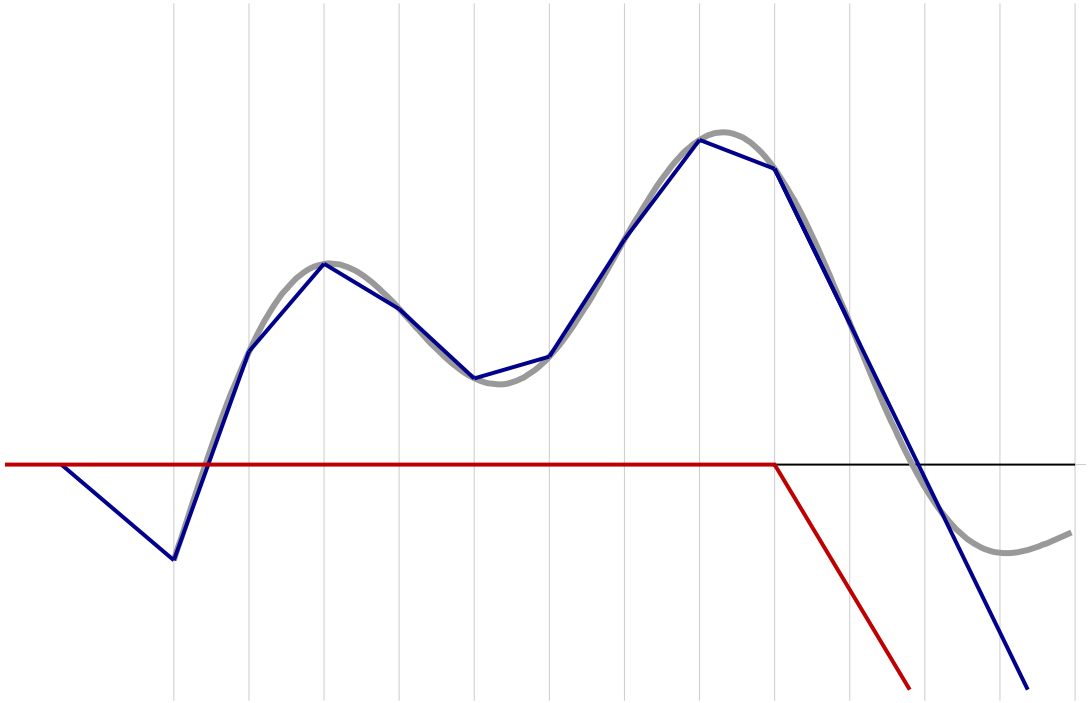
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

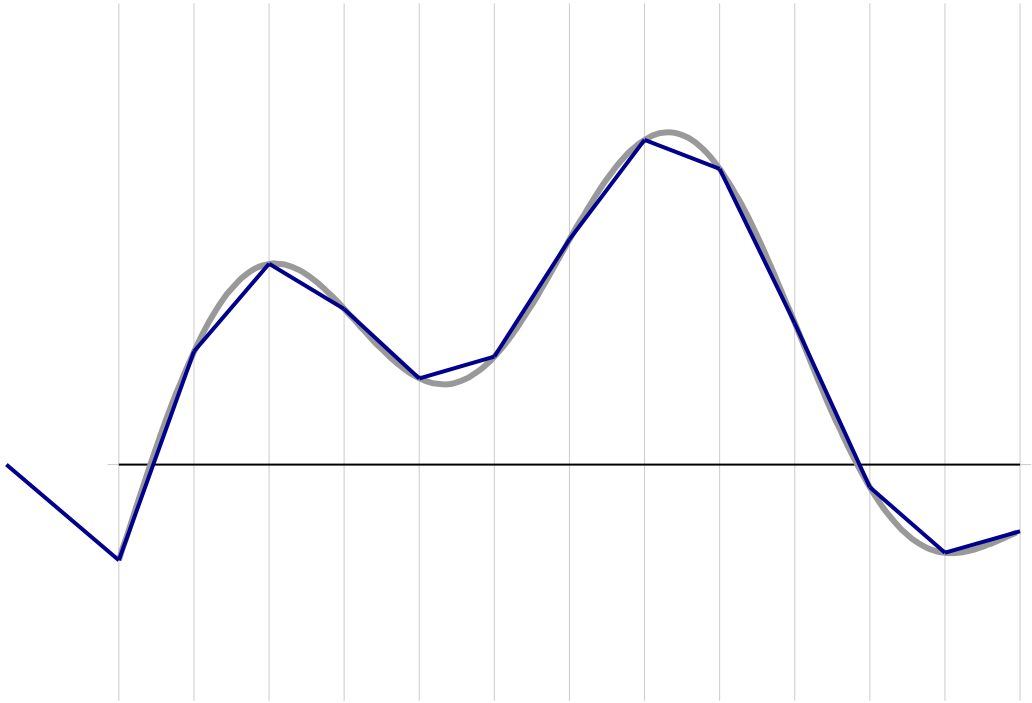
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



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.

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



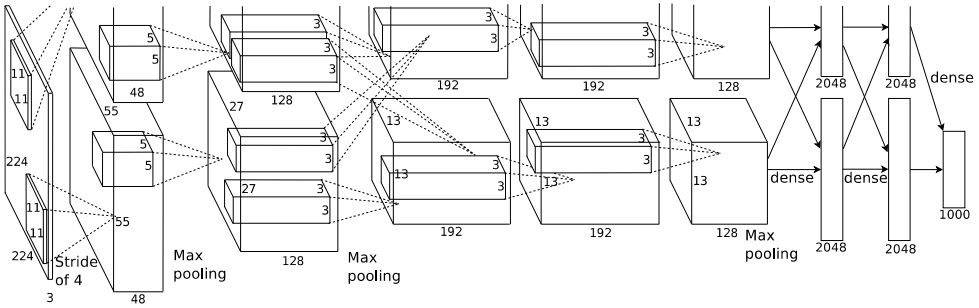
This is true for other activation functions u

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

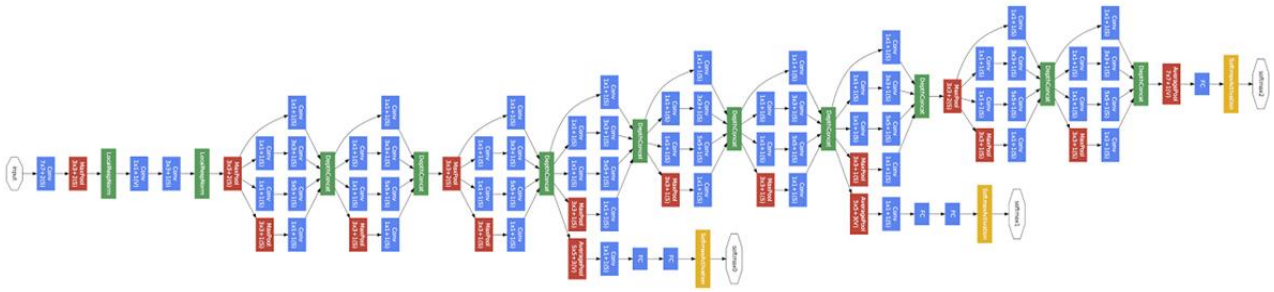
Is adding layers enough?

Going deeper

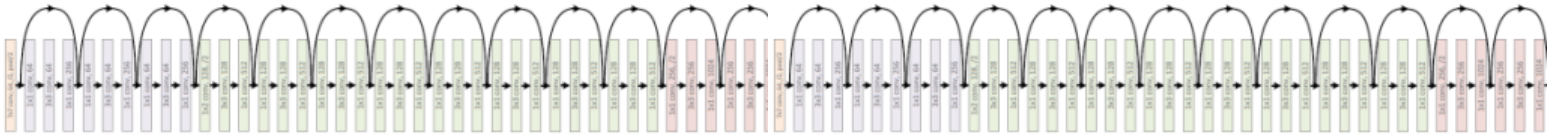
2012 : AlexNet, 8 layers. New techniques: dropout, ReLU



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



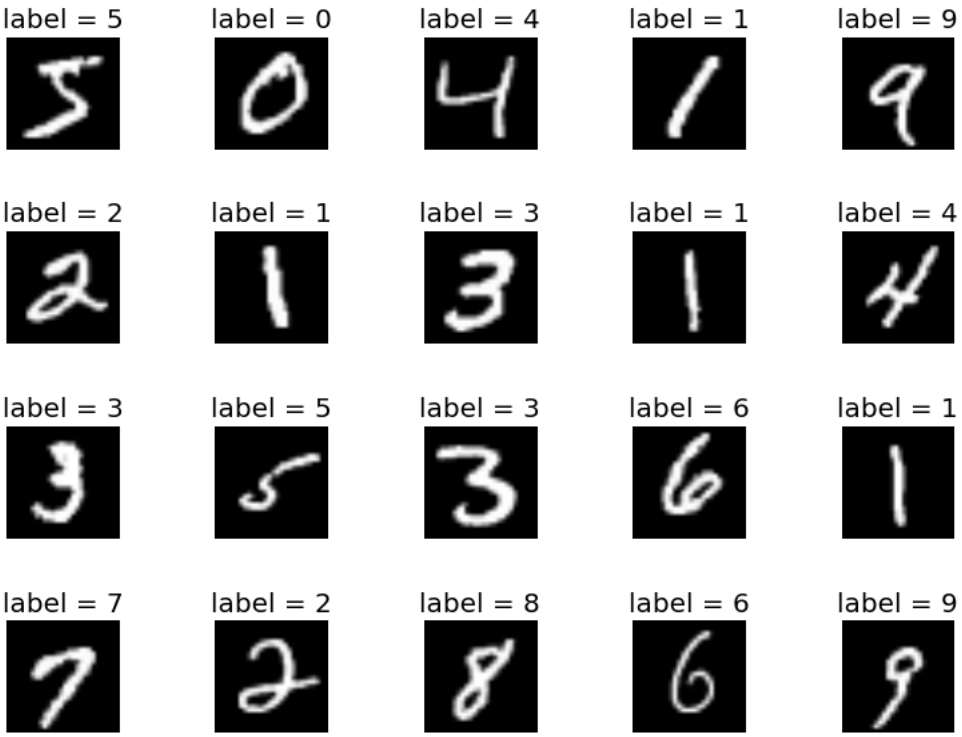
2015 : Microsoft research, 150 layers. New technique: residual learning



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/

MNIST : MLP performance

linear classifier (1-layer NN)	none	12.0	LeCun et al. 1998
linear classifier (1-layer NN)	deskewing	8.4	LeCun et al. 1998
pairwise linear classifier	deskewing	7.6	LeCun et al. 1998

2-layer NN, 300 hidden units, mean square error	none	4.7	LeCun et al. 1998
2-layer NN, 300 HU, MSE, [distortions]	none	3.6	LeCun et al. 1998
2-layer NN, 300 HU	deskewing	1.6	LeCun et al. 1998
2-layer NN, 1000 hidden units	none	4.5	LeCun et al. 1998
2-layer NN, 1000 HU, [distortions]	none	3.8	LeCun et al. 1998
3-layer NN, 300+100 hidden units	none	3.05	LeCun et al. 1998
3-layer NN, 300+100 HU [distortions]	none	2.5	LeCun et al. 1998
3-layer NN, 500+150 hidden units	none	2.95	LeCun et al. 1998
3-layer NN, 500+150 HU [distortions]	none	2.45	LeCun et al. 1998

(Validation performance)

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`.

```
1  import torch
2  from torch.utils.data import Dataset, DataLoader
3  from torchvision import transforms
4
5  class MNISTDataset(Dataset):
6      def __init__(self, dir, transform=None):
7          # Read in the files from the directory and
8          # store them in self.images and self.labels
9          # See MOODLE for the full code of this method
10         (...)
11
12     # The access is _NOT_ shuffled. The PyTorch
13     # Dataloader will need to do this.
14     def __getitem__(self, index):
15         return self.images[index], self.labels[index]
16     # Return the dataset size
17     def __len__(self):
18         return self.no_images
```

Writing Data Access

The MNIST dataset is actually already supported by PyTorch, which includes are ready to use Dataset class, which can even download the data from Yann Lecun's website.

```
1 dataset = datasets.MNIST(dir, train, download=True,
2     transform=transforms.ToTensor())
```

We use our own dataloader in this lecture to learn how to write them.

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:

```
1 dataset = MNISTDataset ("MNIST-png/testing",
2     transforms.Compose([
3     transforms.ToTensor(),
4     transforms.Normalize((0.1307,), (0.3081,))]))
5
6 loader = torch.utils.data.DataLoader(dataset,
7     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!