Tutorial on Deep Learning with Theano and Lasagne

Jan Schlüter
Sander Dieleman
EMBL-EBI
2016-04-21

Tutorial Outline

- Foundations
  - What are Artificial Neural Networks?
  - Practical part: Theano and Lasagne
- Learning
  - Loss functions
  - Gradient Descent
  - Practical part: Digit recognition
- Evaluation
  - Overfitting and Underfitting
  - Practical part: Validation and Testing
- ConvNets
  - Motivation
  - Practical part: Better digit recognition
- Tricks of the Trade
- Outlook

image © arkpo, fotolia.de

What are Artificial Neural Networks?

“a simulation of a small brain”

What are Artificial Neural Networks?

“a simulation of a small brain”

not.

What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

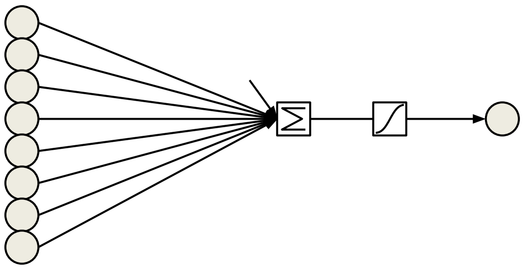
$y = \sigma(b + \mathbf{w}^T\mathbf{x})$ (equivalent to logistic regression)

What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

$y = \sigma(b + \mathbf{w}^T\mathbf{x})$ (equivalent to logistic regression)

expression can be visualized as a graph:

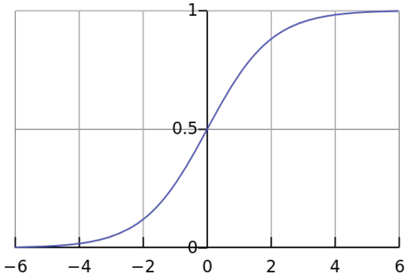


$\mathbf{x}$ $b + \mathbf{w}^T\mathbf{x}$ y

Output value is computed as a **weighted sum of its inputs,**

$$b + \mathbf{w}^T\mathbf{x} = b + \Sigma_i w_i x_i$$

followed by a nonlinear function.

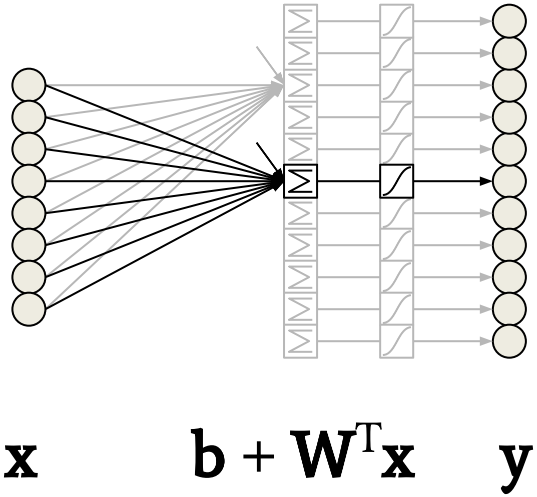


What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

$$\mathbf{y} = \sigma(\mathbf{b} + \mathbf{W}^T\mathbf{x}) \qquad \text{(multiple logistic regressions)}$$

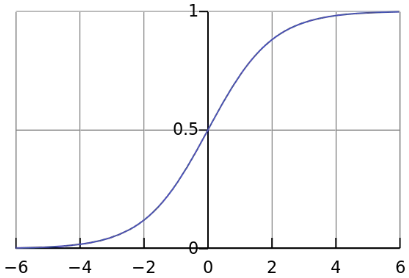
expression can be visualized as a graph:



Output values are computed as **weighted sums of their inputs,**

$$\mathbf{b} + \mathbf{W}^T\mathbf{x} = b_j + \Sigma_i w_{ij} x_i$$

followed by a nonlinear function.

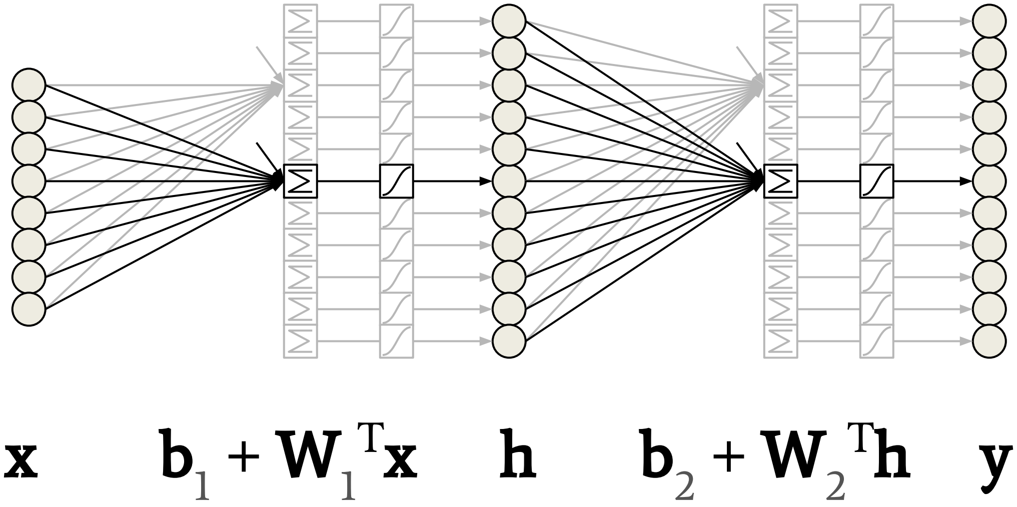


What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^T \sigma(\mathbf{b}_1 + \mathbf{W}_1^T \mathbf{x})) \qquad \text{(stacked logistic regressions)}$$

expression can be visualized as a graph:

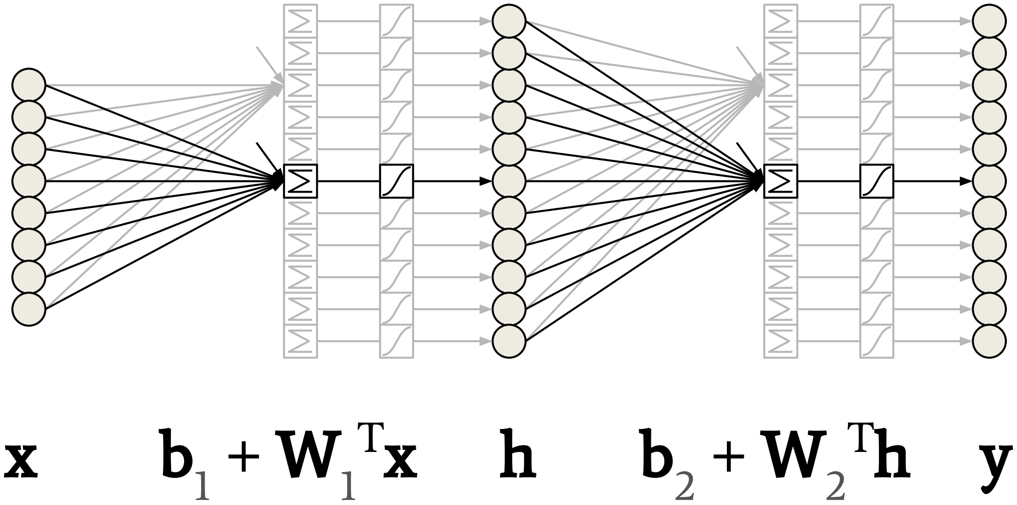

$\mathbf{x}$
$\mathbf{b}_1 + \mathbf{W}_1^T\mathbf{x}$
$\mathbf{h}$
$\mathbf{b}_2 + \mathbf{W}_2^T\mathbf{h}$
$\mathbf{y}$

What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^T \sigma(\mathbf{b}_1 + \mathbf{W}_1^T \mathbf{x})) \qquad \text{(stacked logistic regressions)}$$

expression can be visualized as a graph:


$\mathbf{x}$
$\mathbf{b}_1 + \mathbf{W}_1^T \mathbf{x}$
$\mathbf{h}$
$\mathbf{b}_2 + \mathbf{W}_2^T \mathbf{h}$
$\mathbf{y}$

Universal Approximation Theorem:
This can model any continuous function from $\mathbb{R}^n$ to $\mathbb{R}^m$ arbitrarily well (if **h** is made large enough).

Interlude: Why *Deep* Networks?

Mathematical arguments: (e.g., Hastad et al. 1986, Bengio & Delalleau 2011)

- Some functions are exponentially **more compact** when expressed as a deep network instead of a shallow one
- Evidence indicates that for complex tasks, it is **easier to optimize** a deep network to perform well than a shallow one

Intuitive arguments: (Y. Bengio, Deep Learning, MLSS 2015)

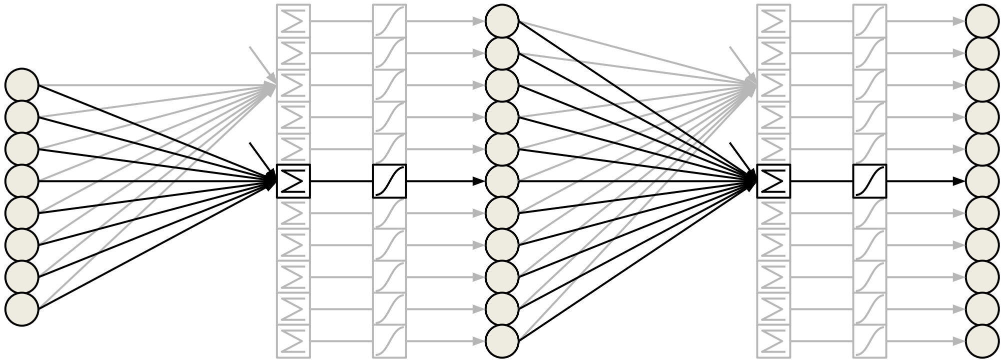
- Humans organize their ideas and concepts **hierarchically**
- Humans **first learn simpler concepts** and then **compose** them to represent more abstract ones
- Engineers **break-up solutions into multiple levels** of abstraction and processing

What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^T \sigma(\mathbf{b}_1 + \mathbf{W}_1^T \mathbf{x})) \qquad \text{(stacked logistic regressions)}$$

expression can be visualized as a graph:



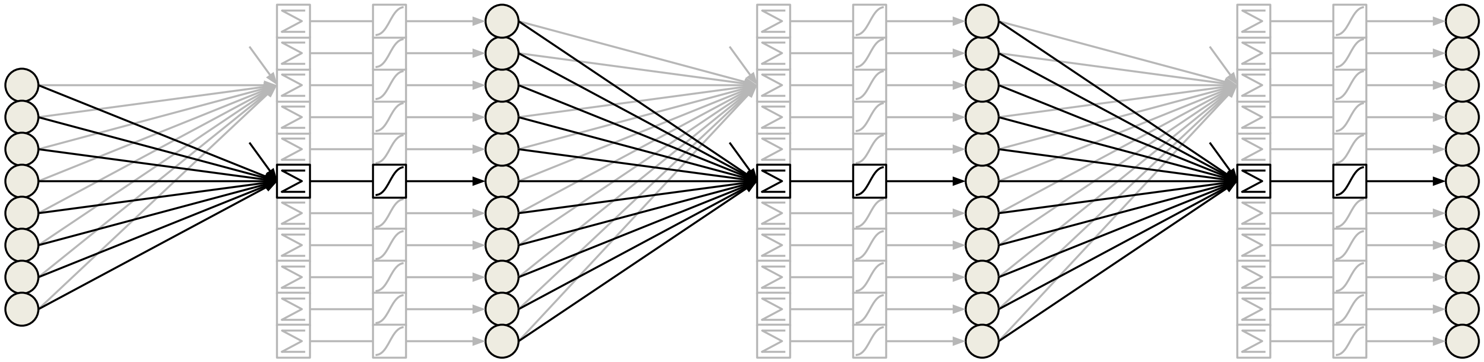
$\mathbf{x}$ $\quad$ $\mathbf{b}_1 + \mathbf{W}_1^T\mathbf{x}$ $\quad$ $\mathbf{h}$ $\quad$ $\mathbf{b}_2 + \mathbf{W}_2^T\mathbf{h}$ $\quad$ $\mathbf{y}$

What are Artificial Neural Networks?

a fancy name for particular mathematical expressions, such as:

$$\mathbf{y} = \sigma(\mathbf{b}_3 + \mathbf{W}_3{}^T\sigma(\mathbf{b}_2 + \mathbf{W}_2{}^T\sigma(\mathbf{b}_1 + \mathbf{W}_1{}^T\mathbf{x})))$$

expression can be visualized as a graph:



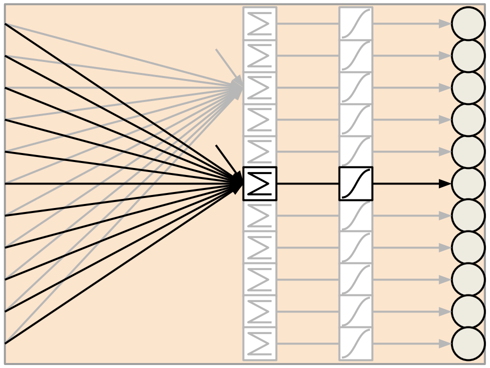
$\mathbf{x}$ $\quad$ $\mathbf{b}_1 + \mathbf{W}_1{}^T\mathbf{x}$ $\quad$ $\mathbf{h}_1$ $\quad$ $\mathbf{b}_2 + \mathbf{W}_2{}^T\mathbf{h}_1$ $\quad$ $\mathbf{h}_2$ $\quad$ $\mathbf{b}_3 + \mathbf{W}_3{}^T\mathbf{h}_2$ $\quad$ $\mathbf{y}$

What are Artificial Neural Networks?

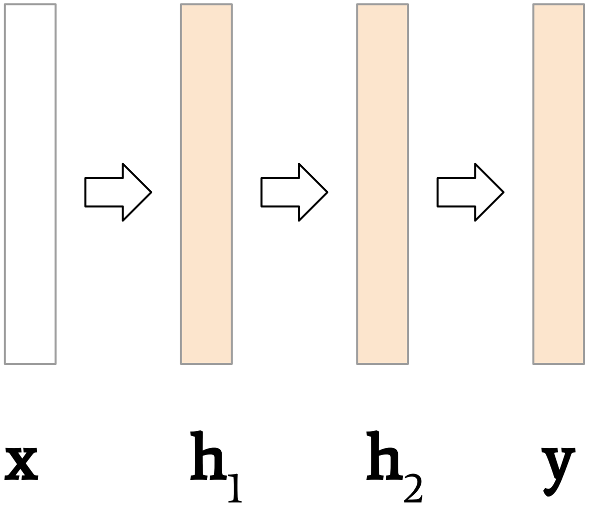
a fancy name for particular mathematical expressions, such as:

$$f_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \sigma(\mathbf{b} + \mathbf{W}^T\mathbf{x}) \qquad \mathbf{y} = (f_{W_3,b_3} \circ f_{W_2,b_2} \circ f_{W_1,b_1})(\mathbf{x})$$

expression can be visualized as a graph:



“dense layer”



composed of simpler functions, commonly termed “layers”

Theano and Lasagne

Theano is a mathematical expression compiler. It allows to define

$$\mathbf{y} = \sigma(\mathbf{b}_3 + \mathbf{W}_3^T\sigma(\mathbf{b}_2 + \mathbf{W}_2^T\sigma(\mathbf{b}_1 + \mathbf{W}_1^T\mathbf{x})))$$

and compile an executable function that computes this expression efficiently, numerically stable, on CPU or GPU, for any input.

Lasagne builds on top of Theano. It provides the building blocks ("layers") to easily define expressions for neural networks.

InputLayer → DenseLayer → DenseLayer → DenseLayer

Theano: Baby Steps

Multiplying numbers in Python:

```
>>> a = 6
>>> b = 7
>>> a * b
42
```

Theano: Baby Steps

Multiplying numbers in Python:

```
>>> a = 6
>>> b = 7
>>> a * b
42
```

assigned Python variables:

```
a: 6
b: 7
```

Theano: Baby Steps

Multiplying numbers in Python:

```
>>> a = 6
>>> b = 7
>>> a * b
42
>>> y = a * b
>>> y
42
```

assigned Python variables:

```
a: 6
b: 7


y: 42
```

Theano: Baby Steps

Multiplying numbers in Theano:

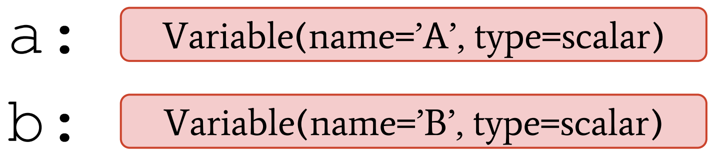
```
>>> import theano
>>> T = theano.tensor
>>> a = T.scalar(‘A’)
>>> b = T.scalar(‘B’)
>>> a * b
Elemwise{mul,no_inplace}.0
```

Theano: Baby Steps

Multiplying numbers in Theano:

```
>>> import theano
>>> T = theano.tensor
>>> a = T.scalar(‘A’)
>>> b = T.scalar(‘B’)
>>> a * b
Elemwise{mul,no_inplace}.0
```

assigned Python variables:



a: Variable(name=’A’, type=scalar)

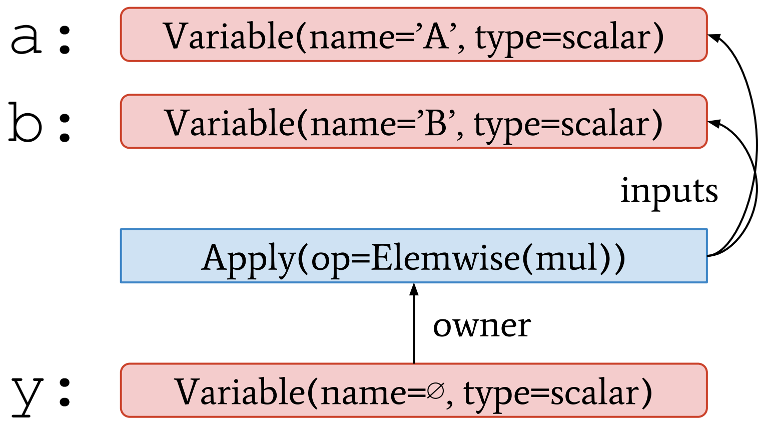
b: Variable(name=’B’, type=scalar)

Theano: Baby Steps

Multiplying numbers in Theano:

```
>>> import theano
>>> T = theano.tensor
>>> a = T.scalar('A')
>>> b = T.scalar('B')
>>> a * b
Elemwise{mul,no_inplace}.0
>>> y = a * b
```

assigned Python variables:



a: Variable(name='A', type=scalar)

b: Variable(name='B', type=scalar)

inputs

Apply(op=Elemwise(mul))

owner

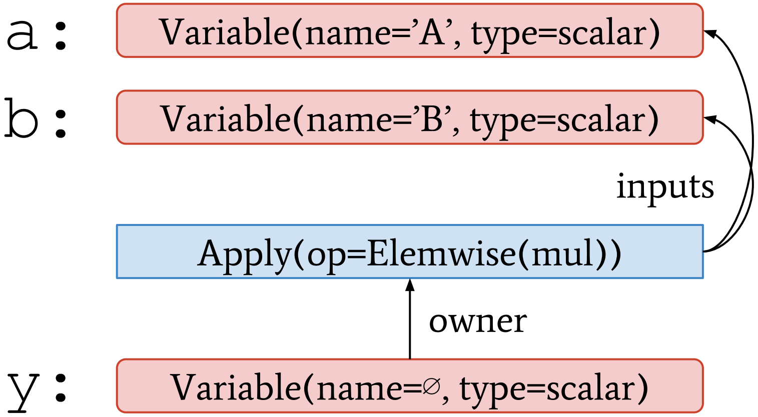
y: Variable(name=∅, type=scalar)

Theano: Baby Steps

Multiplying numbers in Theano:

```
>>> import theano
>>> T = theano.tensor
>>> a = T.scalar('A')
>>> b = T.scalar('B')
>>> a * b
Elemwise{mul,no_inplace}.0
>>> y = a * b
>>> theano.pp(y)
'(A * B)'
```

assigned Python variables:


a:
Variable(name='A', type=scalar)
b:
Variable(name='B', type=scalar)
inputs
Apply(op=Elemwise(mul))
owner
y:
Variable(name=∅, type=scalar)

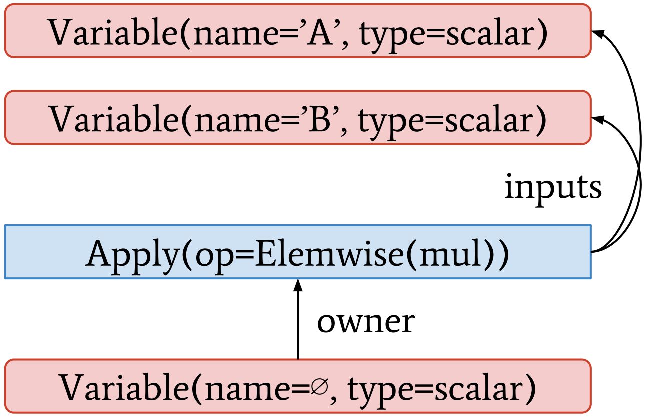
Theano: Baby Steps

Multiplying numbers in Theano:

```
>>> import theano
>>> T = theano.tensor
>>> a = T.scalar(‘A’)
>>> b = T.scalar(‘B’)
>>> a * b
Elemwise{mul,no_inplace}.0
>>> y = a * b
>>> fn = theano.function(
...         [a, b], y)
```

assigned Python variables:

```
a:
b:
y:
fn: Function object
```

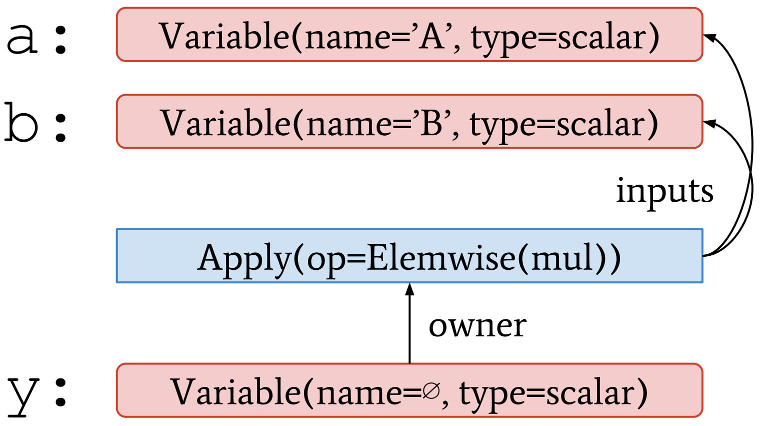

Variable(name=’A’, type=scalar)
Variable(name=’B’, type=scalar)
inputs
Apply(op=Elemwise(mul))
owner
Variable(name=∅, type=scalar)

Theano: Baby Steps

Multiplying numbers in Theano:

```
>>> import theano
>>> T = theano.tensor
>>> a = T.scalar('A')
>>> b = T.scalar('B')
>>> a * b
Elemwise{mul,no_inplace}.0
>>> y = a * b
>>> fn = theano.function(
...         [a, b], y)
>>> r = fn(6, 7)
```

assigned Python variables:



a: Variable(name='A', type=scalar)

b: Variable(name='B', type=scalar)

Apply(op=Elemwise(mul))

y: Variable(name=∅, type=scalar)

```
fn: Function object

r: 42
```

theano.function

```
>>> fn = theano.function([a, b], y)
```

What happens under the hood?

- Theano checks that the output expression indeed depends on the given input variables
- It optimizes the output expression to compute the same, but:
  - more efficiently, e.g., replacing x*y/x by y
  - more numerically stable, e.g., for log(1+exp(x))
  - on a graphics card, if configured to do so
- It emits C++ code computing the outputs given the inputs, compiles the code and imports it to be callable from Python

Theano: Small Neural Network

We will now define a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2{}^{\mathrm{T}}\sigma(\mathbf{b}_1 + \mathbf{W}_1{}^{\mathrm{T}}\mathbf{x}))$$

```
>>> import theano; import theano.tensor as T
>>> x = T.vector(‘x’)
>>> W1 = T.matrix(‘W1’)
>>> b1 = T.vector(‘b1’)
>>> h1 = T.nnet.sigmoid(b1 + T.dot(W1.T, x))
```

Fill in the missing steps!

```
>>> fn = theano.function([x,W1,b1,W2,b2], y)
```

Theano: Small Neural Network

We will now define a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2{}^T\sigma(\mathbf{b}_1 + \mathbf{W}_1{}^T\mathbf{x}))$$

```
>>> import theano; import theano.tensor as T
>>> x = T.vector(‘x’)
>>> W1 = T.matrix(‘W1’)
>>> b1 = T.vector(‘b1’)
>>> h1 = T.nnet.sigmoid(b1 + T.dot(W1.T, x))
>>> W2 = T.matrix(‘W2’)
>>> b2 = T.vector(‘b2’)
>>> y = T.nnet.sigmoid(b2 + T.dot(W2.T, h1))
>>> fn = theano.function([x,W1,b1,W2,b2], y,
...      allow_input_downcast=True)
```

Theano: Small Neural Network

We wanted to define a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^T \sigma(\mathbf{b}_1 + \mathbf{W}_1^T \mathbf{x}))$$

Let's see if our Theano expression matches what we wanted:

```
>>> theano.pp(y)
'sigmoid((b2 + (W2.T \\dot sigmoid((b1 + (W1.
T \\dot x)))))))'
```

Looks good!

Theano: Small Neural Network

We have now defined a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2{}^T\sigma(\mathbf{b}_1 + \mathbf{W}_1{}^T\mathbf{x}))$$

To run the compiled function, we need values for the network parameters. Assuming an input vector of 784 values, a hidden layer of 100 values and a single output value:

```
>>> import numpy as np
>>> weights1 = np.random.randn(784, 100)
>>> bias1 = np.random.randn(100)
>>> weights2 = np.random.randn(100, 1)
>>> bias2 = np.random.randn(1)
```

Theano: Small Neural Network

We have now defined a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^T\sigma(\mathbf{b}_1 + \mathbf{W}_1^T\mathbf{x}))$$

```
>>> import numpy as np
>>> weights1 = np.random.randn(784, 100)
>>> bias1 = np.random.randn(100)
>>> weights2 = np.random.randn(100, 1)
>>> bias2 = np.random.randn(1)
```

To pass some random input through the network:

```
>>> fn(np.random.randn(784), weights1, bias1,
...    weights2, bias2)
array([ 0.99683517], dtype=float32)
```

Theano: Small Neural Network

We have now defined a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^{T}\sigma(\mathbf{b}_1 + \mathbf{W}_1^{T}\mathbf{x}))$$

Note that this definition only needs a minor change to process multiple input vectors in parallel:

$$\mathbf{Y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2^{T}\sigma(\mathbf{b}_1 + \mathbf{W}_1^{T}\mathbf{X}))$$

It’s similarly easy to adapt our network definition:

```
>>> x = T.matrix(‘X’)
```

We just need to re-run three lines:

```
>>> h1 = ...
>>> y = ...
>>> fn = theano.function(...)
```

Theano: Small Neural Network

We have now defined a small neural network in Theano:

$$\mathbf{y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2{}^T\sigma(\mathbf{b}_1 + \mathbf{W}_1{}^T\mathbf{x}))$$

Note that this definition only needs a minor change to process multiple input vectors in parallel:

$$\mathbf{Y} = \sigma(\mathbf{b}_2 + \mathbf{W}_2{}^T\sigma(\mathbf{b}_1 + \mathbf{W}_1{}^T\mathbf{X}))$$

But: In numpy and Theano, data points are usually organized in rows rather than columns (as the underlying memory layout follows C conventions, not Fortran conventions as in Matlab):

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

From now on, we will see this layout only.

Lasagne: Small Neural Network

We will now define the same neural network in Lasagne:

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

```
>>> import lasagne
>>> X = T.matrix(‘X’)
>>> l1 = lasagne.layers.InputLayer(
...         shape=(100, 784), input_var=X)
```

This defines an input layer which expects 100 inputs of 784 elements each (a 100x784 matrix).

Lasagne: Small Neural Network

We will now define the same neural network in Lasagne:

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

```
>>> import lasagne
>>> X = T.matrix(‘X’)
>>> l1 = lasagne.layers.InputLayer(
...     shape=(100, 784), input_var=X)
```

This defines an input layer which expects 100 inputs of 784 elements each (a 100x784 matrix). We can also define it to expect an arbitrary number of inputs of 784 elements each:

```
>>> l1 = lasagne.layers.InputLayer(
...     shape=(None, 784), input_var=X)
```

Lasagne: Small Neural Network

We will now define the same neural network in Lasagne:

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

```
>>> l1 = lasagne.layers.InputLayer(
...       shape=(None, 784), input_var=X)
```

We add two sigmoid dense layers on top:

```
>>> from lasagne.nonlinearities import sigmoid
>>> l2 = lasagne.layers.DenseLayer(
...      l1, num_units=100, nonlinearity=sigmoid)
>>> l3 = lasagne.layers.DenseLayer(
         l2, num_units=1, nonlinearity=sigmoid)
```

Lasagne: Small Neural Network

We will now define the same neural network in Lasagne:

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

```
>>> l2 = lasagne.layers.DenseLayer(
...    l1, num_units=100, nonlinearity=sigmoid)
>>> l3 = lasagne.layers.DenseLayer(
       l2, num_units=1, nonlinearity=sigmoid)
```

Each layer is linked to the layer it operates on, creating a chain (in this case). When creating the layers, Lasagne already creates correctly-sized network parameters for us (that’s why it needs to know the expected shape for the input layer, and the number of units for each dense layer).

Lasagne: Small Neural Network

We wanted to define the same neural network in Lasagne:

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

To obtain the expression, we call `get_output` on the top layer:

```
>>> Y = lasagne.layers.get_output(l3)
```

Comparing against our goal:

```
>>> theano.pp(Y)
'sigmoid(((sigmoid(((X \\dot W) + b)) \\dot W)
+ b))'
```

Looks good! The weights and biases have indistinguishable names because we did not name our layers, but that’s fine.

Lasagne: Small Neural Network

We have defined the same neural network in Lasagne:

$$\mathbf{Y} = \sigma(\sigma(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

Again, we can compile this into a function and pass some random data through it (here, 3 input vectors of 784 values each):

```
>>> fn = theano.function([X], Y,
...      allow_input_downcast=True)
>>> fn(np.random.randn(3, 784))
array([[ 0.88817853],
       [ 0.74262416],
       [ 0.86233407]], dtype=float32)
```

Network parameters are part of the graph and need not be given.

Lasagne: Small Neural Network

Exercise: Modify the network such that `l3` becomes a dense layer with 100 sigmoid units, and add `l4` as a dense layer with 10 units and softmax nonlinearity.

Lasagne: Small Neural Network

Exercise: Replace `l3` by a dense layer with 100 sigmoid units, and add `l4` as a dense layer with 10 units and softmax nonlinearity.

Solution:

```
>>> from lasagne.layers import DenseLayer
>>> from lasagne.nonlinearities import softmax
>>> l3 = DenseLayer(l2, 100,
...         nonlinearity=sigmoid)
>>> l4 = DenseLayer(l3, 10,
...         nonlinearity=softmax)
```

Lasagne: Small Neural Network

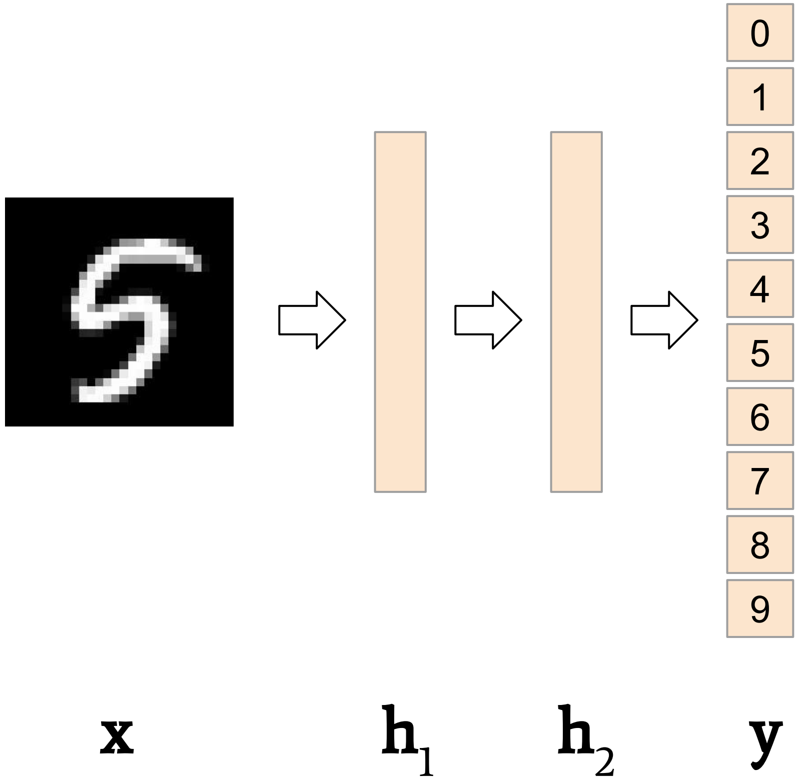
Exercise: Replace `l3` by a dense layer with 100 sigmoid units, and add `l4` as a dense layer with 10 units and softmax nonlinearity.

Solution:

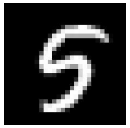
```
>>> from lasagne.layers import DenseLayer
>>> from lasagne.nonlinearities import softmax
>>> l3 = DenseLayer(l2, 100,
...         nonlinearity=sigmoid)
>>> l4 = DenseLayer(l3, 10,
...         nonlinearity=softmax)
```

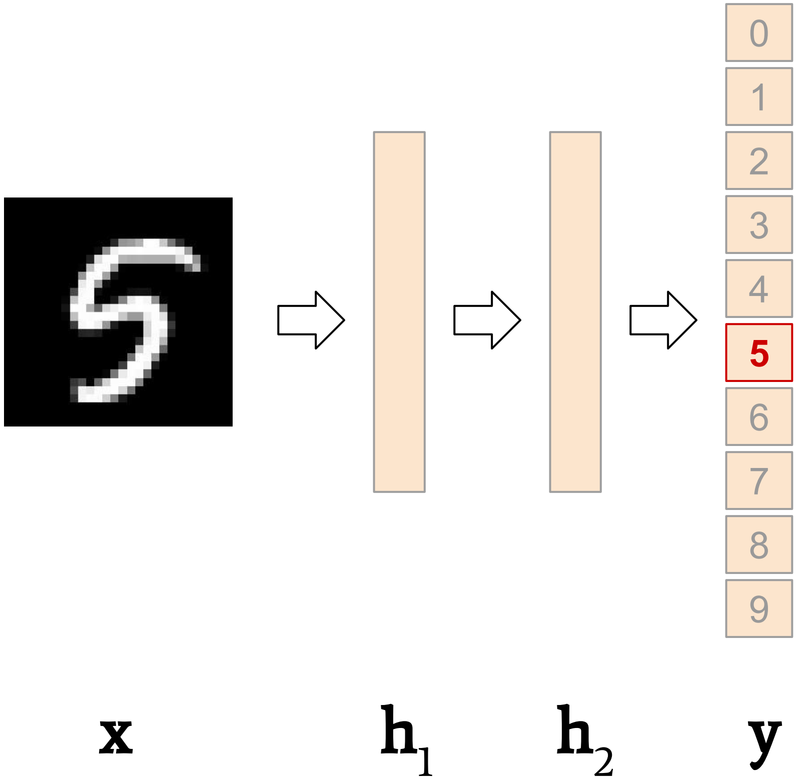
This network can map 28x28=784 pixel images to a probability distribution over 10 classes. So far, its output is fairly random.

Learning

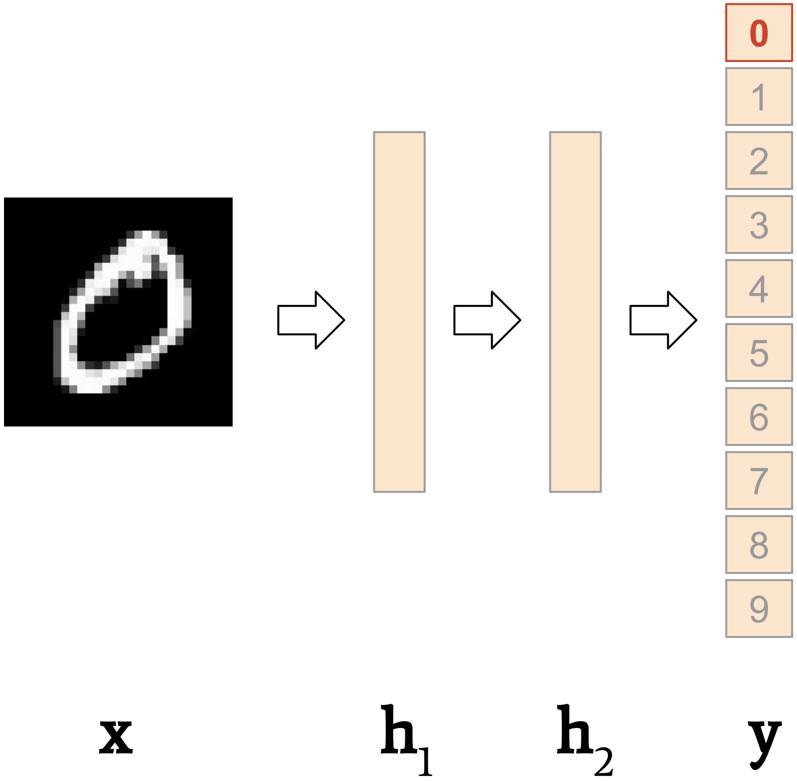


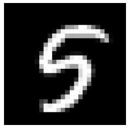
Learning

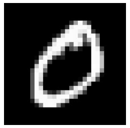
f() = 5



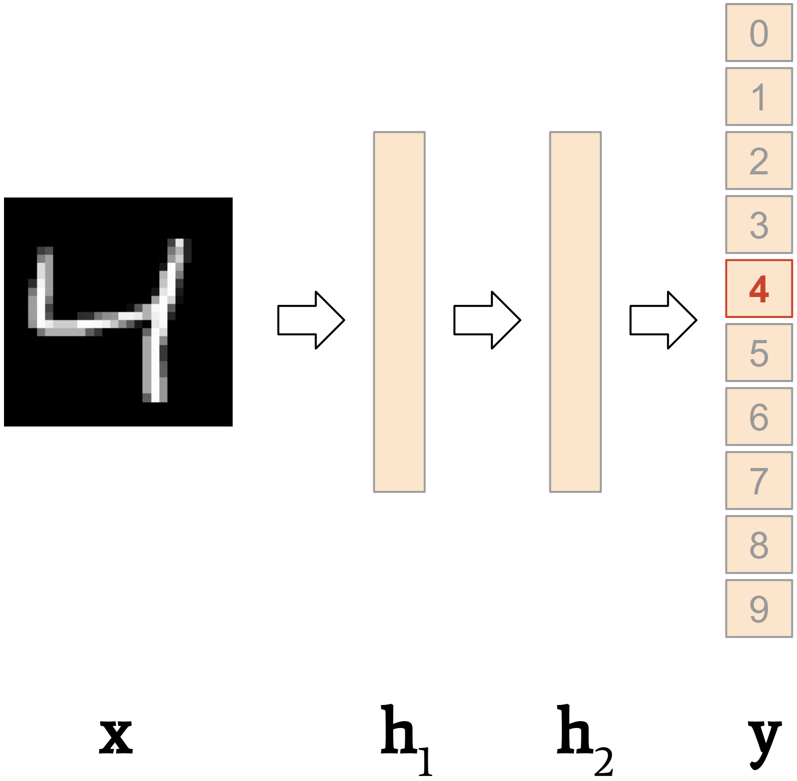
Learning

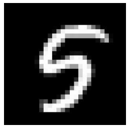


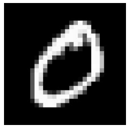
f() = 5

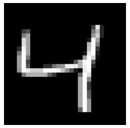
f() = 0

Learning

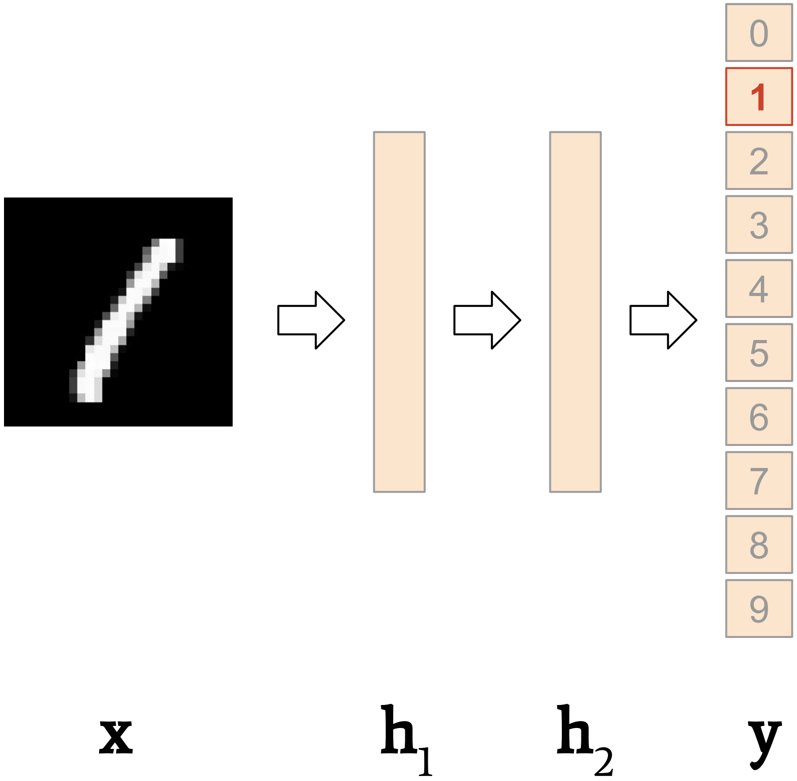


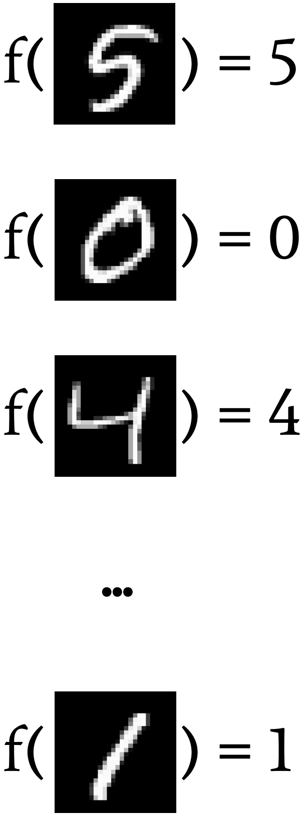
f() = 5

f() = 0

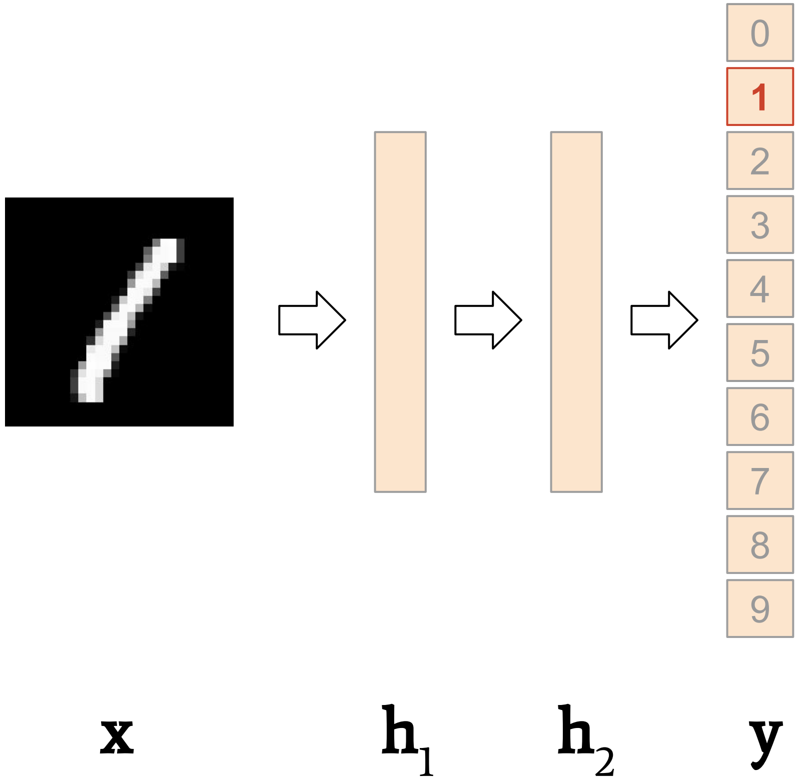
f() = 4

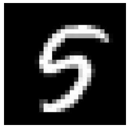
Learning

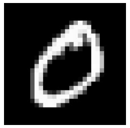


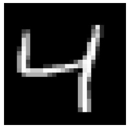


Learning: loss functions

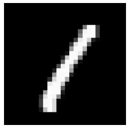


$L(f($  $), 5)$

$+ \ L(f($  $), 0)$

$+ \ L(f($  $), 4)$

...

$+ \ L(f($  $), 1)$

$$\sum_i L(f(\mathbf{x}_i), \mathbf{t}_i)$$

Learning: minimising the loss function

$$\xcancel{\frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = 0}$$

Learning: gradient descent

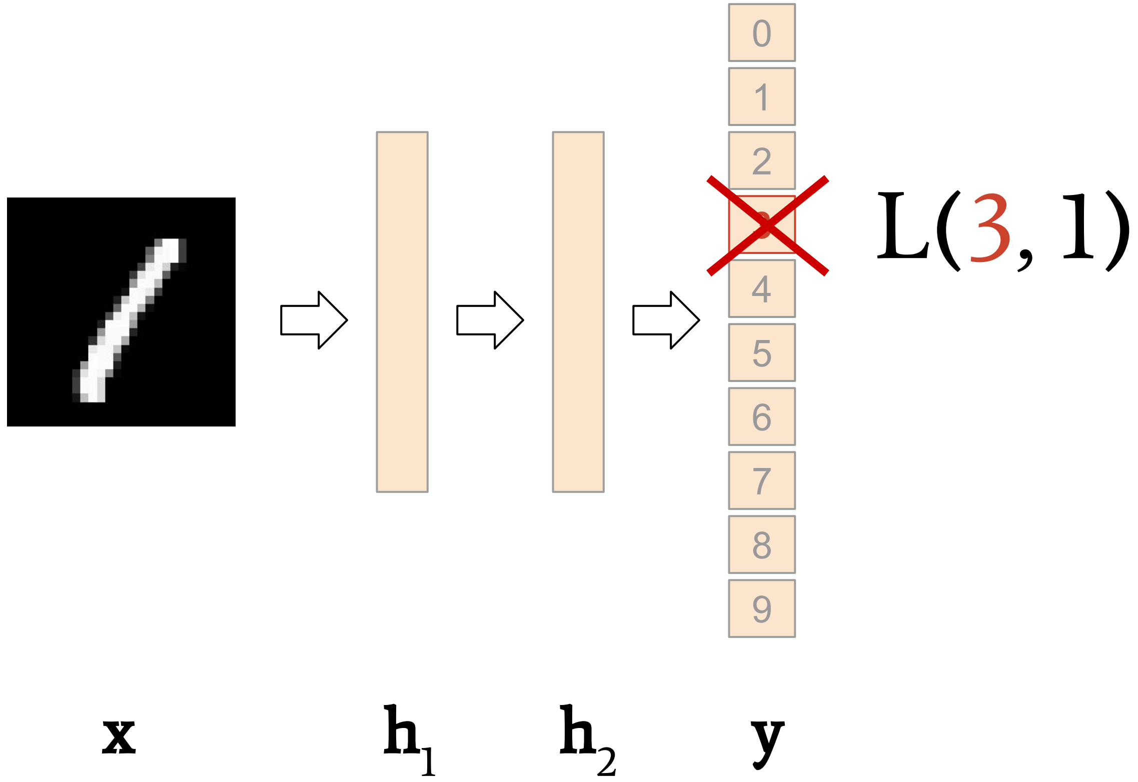
Repeat until convergence:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \underset{\uparrow}{\eta} \frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i)$$

learning rate

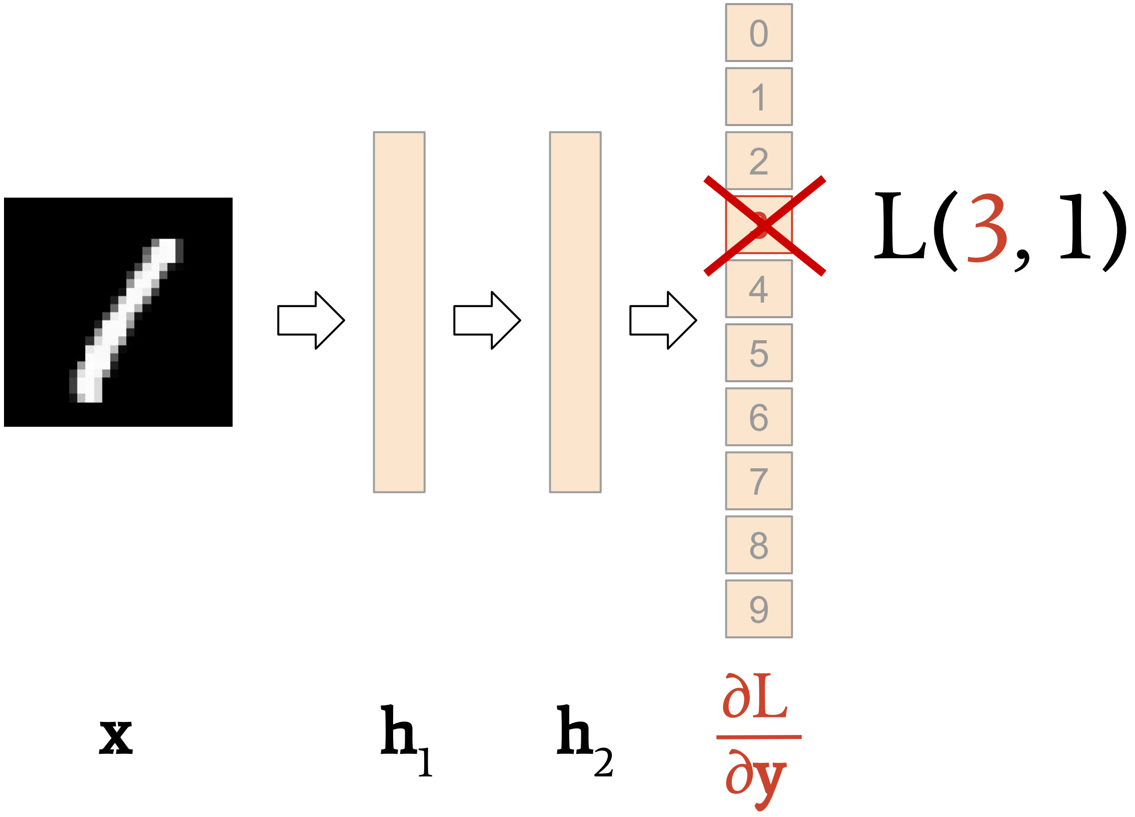
Learning: backpropagation

$$\frac{\partial}{\partial\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = \sum_i \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{\theta}}$$



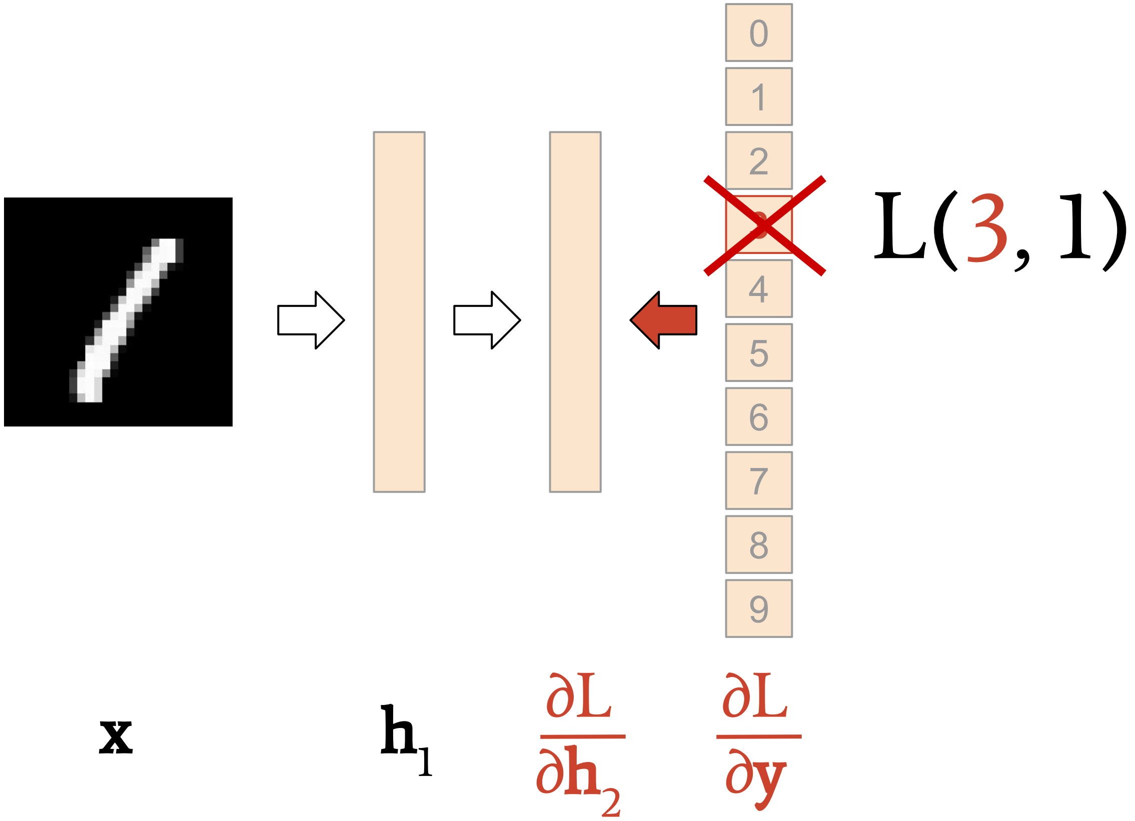
Learning: backpropagation

$$\frac{\partial}{\partial\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = \sum_i \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial\boldsymbol{\theta}}$$



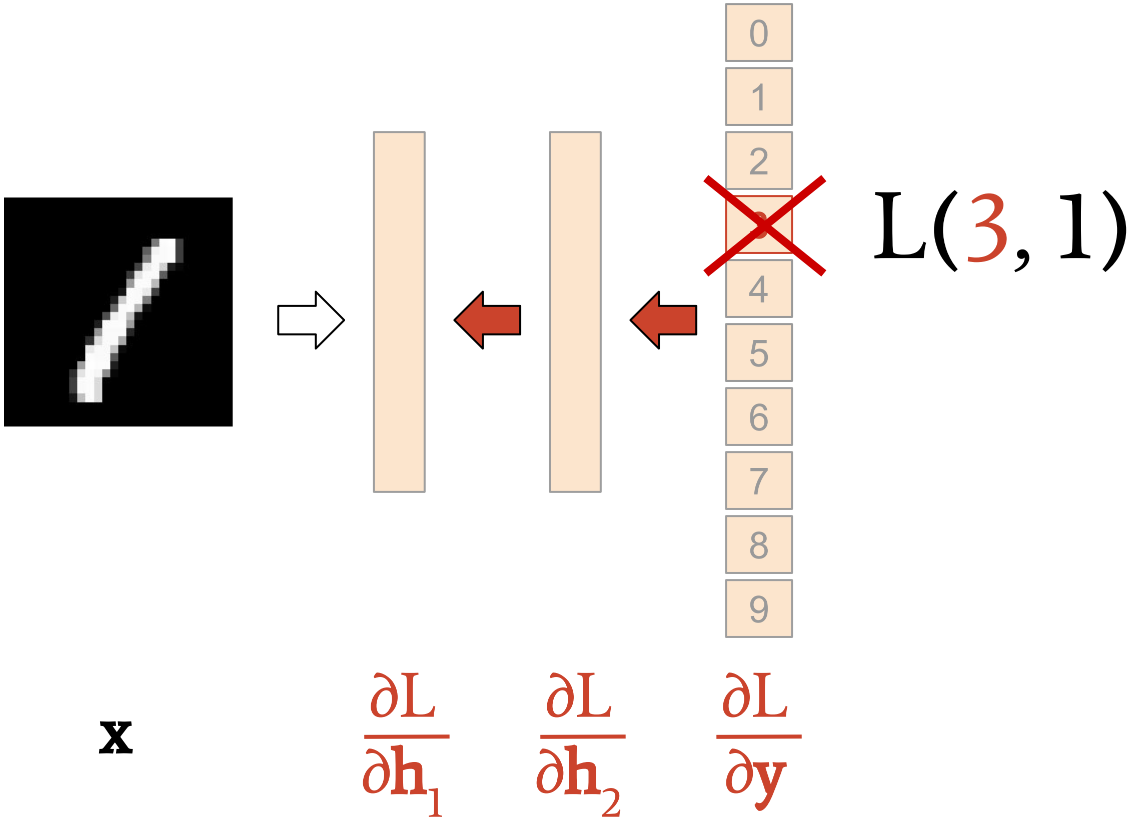
Learning: backpropagation

$$\frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = \sum_i \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{\theta}}$$


0
1
2
3
4
5
6
7
8
9
$L(3, 1)$
$\mathbf{x}$
$\mathbf{h}_1$
$\frac{\partial L}{\partial \mathbf{h}_2}$
$\frac{\partial L}{\partial \mathbf{y}}$

Learning: backpropagation

$$\frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = \sum_i \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{\theta}}$$


0
1
2
3
4
5
6
7
8
9
L(3, 1)
x
$\frac{\partial L}{\partial \mathbf{h}_1}$
$\frac{\partial L}{\partial \mathbf{h}_2}$
$\frac{\partial L}{\partial \mathbf{y}}$

Learning: backpropagation

$$\frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = \sum_i \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{\theta}}$$

$$\frac{\partial L}{\partial \mathbf{W}_1} = \frac{\partial L}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_1}$$

$$\frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial \mathbf{h}_2} \cdot \frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_2}$$

$$\frac{\partial L}{\partial \mathbf{W}_3} = \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}_3}$$

Backpropagation amounts to applying the **chain rule** to compute derivatives w.r.t. model parameters

Learning: backpropagation

$$\frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i) = \sum_i \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial \boldsymbol{\theta}}$$

$$\frac{\partial L}{\partial \mathbf{W}_1} = \frac{\partial L}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_1}$$

$$\frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial \mathbf{h}_2} \cdot \frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_2}$$

$$\frac{\partial L}{\partial \mathbf{W}_3} = \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}_3}$$

Backpropagation amounts to applying the **chain rule** to compute derivatives w.r.t. model parameters

Each layer just needs to know how to compute the partial derivative of its parameters and input given the partial derivative of its output.

Theano does this *per operation*.

Learning: stochastic gradient descent (SGD)

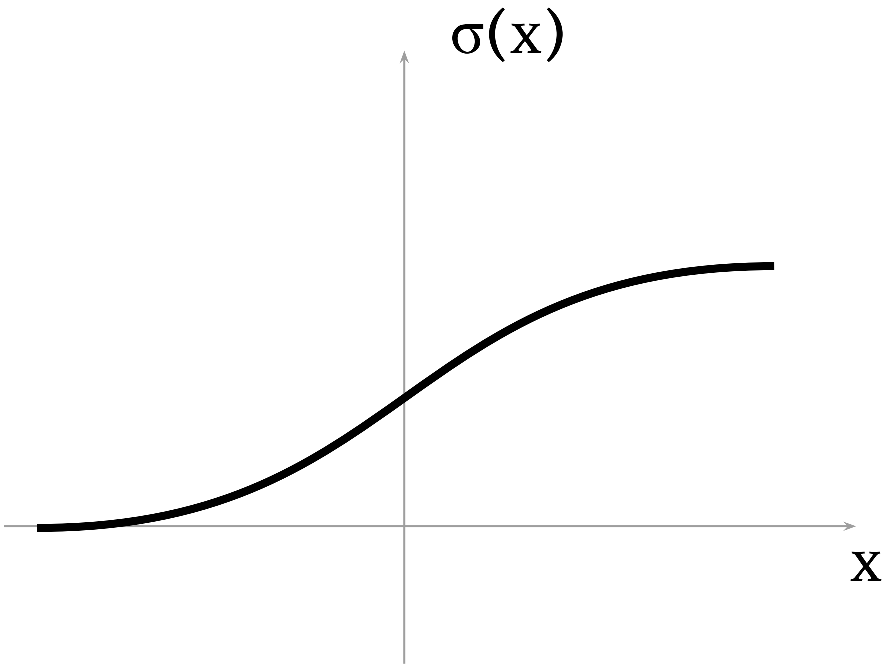
To speed up learning,
approximate the loss using a **small batch** of examples

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial}{\partial \boldsymbol{\theta}} \sum_i L(f(\mathbf{x}_i), \mathbf{t}_i)$$

Stochastic approximation of the loss

- + much faster gradient steps
- - more steps to convergence

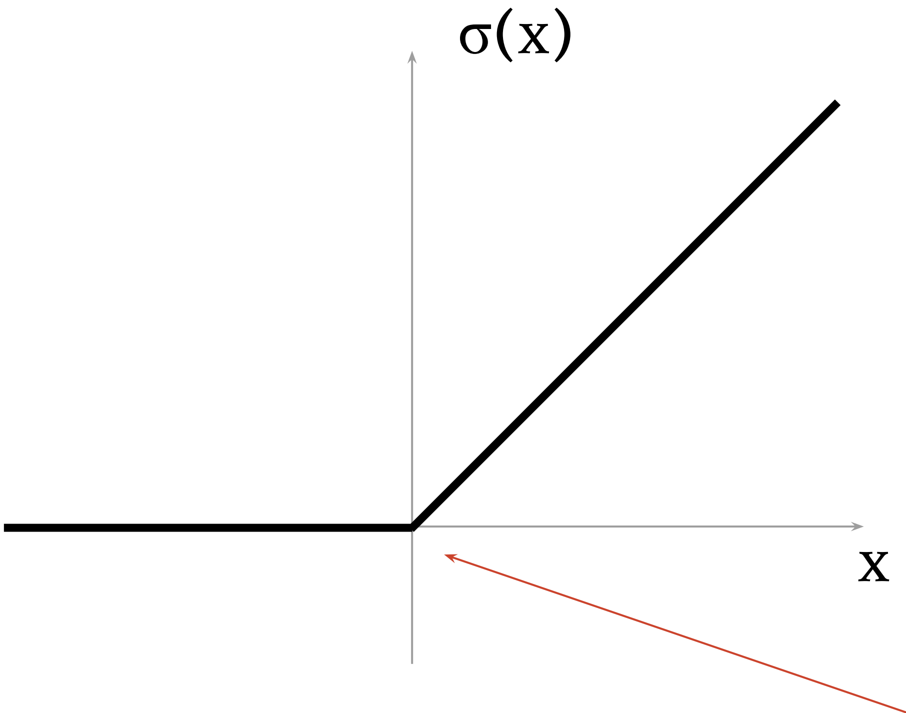
Loss and model must be differentiable



Sigmoid: $\sigma(x) = \dfrac{1}{1 + \exp(-x)}$

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

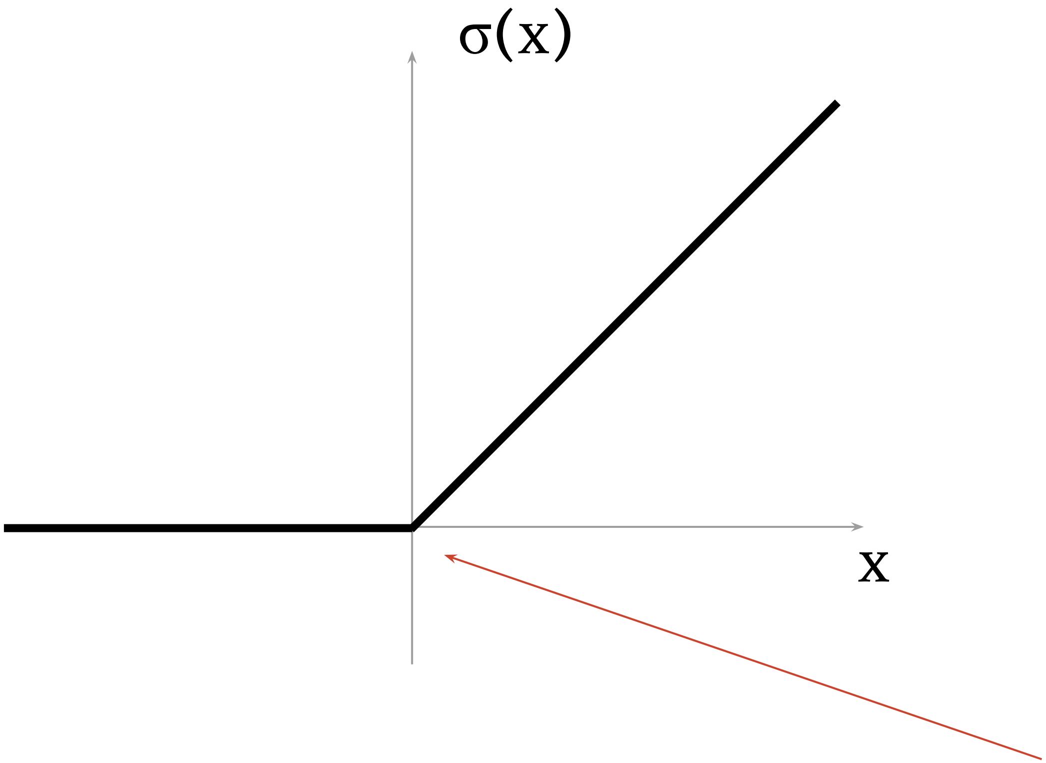
Loss and model must be differentiable



ReLU: $\sigma(x) = \max(x, 0)$

$$\frac{\partial\sigma}{\partial x}(0) = ??$$

Loss and model must be *sort of* differentiable



$$\frac{\partial\sigma}{\partial x}(0) = ??$$

ReLU: $\sigma(x) = \max(x, 0)$

Initialisation

$\frac{\partial L}{\partial \theta}$ is sensitive to the **initial values** of θ

If we're not careful, gradients will either **vanish** or **blow up**

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

Initialisation

$\frac{\partial L}{\partial \theta}$ is sensitive to the **initial values** of θ

If we're not careful, gradients will either **vanish** or **blow up**

$$\theta \leftarrow \theta - \eta \cdot 0$$

Small gradients: θ does not change, no learning happens

Initialisation

$\frac{\partial L}{\partial \theta}$ is sensitive to the **initial values** of θ

If we're not careful, gradients will either **vanish** or **blow up**

$$\theta \leftarrow \theta - \eta \cdot \infty$$

Huge gradients: θ blows up, learning **diverges**

Initialisation

$\frac{\partial L}{\partial \theta}$ is sensitive to the **initial values** of θ

If we're not careful, gradients will either **vanish** or **blow up**

The deeper the network, the more sensitive it is.

Luckily, researchers have found simple recipes for initialization.
Lasagne includes these as well.

Training in Theano

First ingredient: **Symbolic differentiation**.

```
>>> x = T.scalar(‘x’)
>>> y = 5 * x**2
>>> g = theano.grad(y, wrt=x)
>>> theano.pp(g)
```

Training in Theano

First ingredient: **Symbolic differentiation**.

```
>>> x = T.scalar(‘x’)
>>> y = 5 * x**2
>>> g = theano.grad(y, wrt=x)
>>> theano.pp(g)
'(((fill((TensorConstant{5} * (x **
TensorConstant{2})), TensorConstant{1.0}) *
TensorConstant{5}) * TensorConstant{2}) * (x
** (TensorConstant{2} - TensorConstant{1})))'
```

Wading through this, it says:

$$\text{fill}(5x^2, 1.0) \cdot 5 \cdot 2 \cdot x^{(2-1)} = 10\,x$$

Training in Theano

Second ingredient: **Shared variables**,

symbolic variables with an associated value:

```
>>> x = theano.shared(np.array([1.0, 2.0]))
>>> x.get_value()
array([1., 2.])
```

Can participate in expressions like other variables:

```
>>> fn = theano.function([], x**2)
>>> fn()
array([1., 4.])
```

Training in Theano

Second ingredient: **Shared variables**,

symbolic variables with an associated value:

```
>>> x = theano.shared(np.array([1.0, 2.0]))
>>> x.get_value()
array([1., 2.])
```

Can participate in expressions like other variables:

```
>>> fn = theano.function([], x**2)
>>> fn()
array([1., 4.])
>>> x.set_value(np.array([3., 5.])); fn()
array([9., 25.])
```

Training in Theano

Third ingredient: **Updates**,

expressions for new values of shared variables:

```
>>> upd = {x: x + 0.5}
```

Can be applied as part of a function call:

```
>>> fn = theano.function([], [], updates=upd)
```

Training in Theano

Third ingredient: **Updates**,

expressions for new values of shared variables:

```
>>> upd = {x: x + 0.5}
```

Can be applied as part of a function call:

```
>>> fn = theano.function([], [], updates=upd)
>>> fn()
>>> x.get_value()
array([3.5, 5.5])
>>> fn()
>>> x.get_value()
array([4., 6.])
```

Training in Theano

Putting these ingredients together, we can:

- Define a neural network using **shared variables** for its parameters
- Use **symbolic differentiation** to obtain the gradient of a loss function with respect to these parameters
- Define an **update dictionary** that applies a learning rule to the parameters, such as gradient descent

Training in Lasagne

- Define a neural network using **shared variables** for its parameters

Lasagne already does this. To obtain the parameters for a network, call `get_all_params` on its output layer (or layers):

```
>>> params = lasagne.layers.get_all_params(
...          l4, trainable=True)
```

The second argument restricts the list to trainable parameters; some layers may have parameters that are not to be trained.

Training in Lasagne

- Use **symbolic differentiation** to obtain the gradient of a loss function with respect to these parameters

We will use a dummy loss function: The output for the first class.

```
>>> loss = lasagne.layers.get_output(l4)[:,0]
>>> loss = loss.mean()   # average over batch
```

We can obtain the gradients of all parameters at once now:

```
>>> grads = theano.grad(loss, wrt=params)
```

Training in Lasagne

- Define an **update dictionary** that applies a learning rule to the parameters, such as gradient descent

For gradient descent, the update rule would be:

```
>>> eta = 0.01   # learning rate
>>> upd = {p: p - eta*g
...        for p, g in zip(params, grads)}
```

Training in Lasagne

- Define an **update dictionary** that applies a learning rule to the parameters, such as gradient descent

For gradient descent, the update rule would be:

```
>>> eta = 0.01   # learning rate
>>> upd = {p: p - eta*g
...        for p, g in zip(params, grads)}
```

Lasagne provides a shortcut for this (and many other rules):

```
>>> upd = lasagne.updates.sgd(
...       grads, params, eta)
```

The shortcut can also be called with `loss` instead of `grads`.

Training in Lasagne

We can now compile a function that updates the network to minimize our dummy loss (the output for the first class):

```
>>> fn = theano.function(
...        [X], loss, updates=upd)
```

Let’s call it a few times:

```
>>> x = np.random.randn(100, 784)
>>> fn(x)
array(0.1963508427143097, dtype=float32)
>>> fn(x)
array(0.18758298456668854, dtype=float32)
```

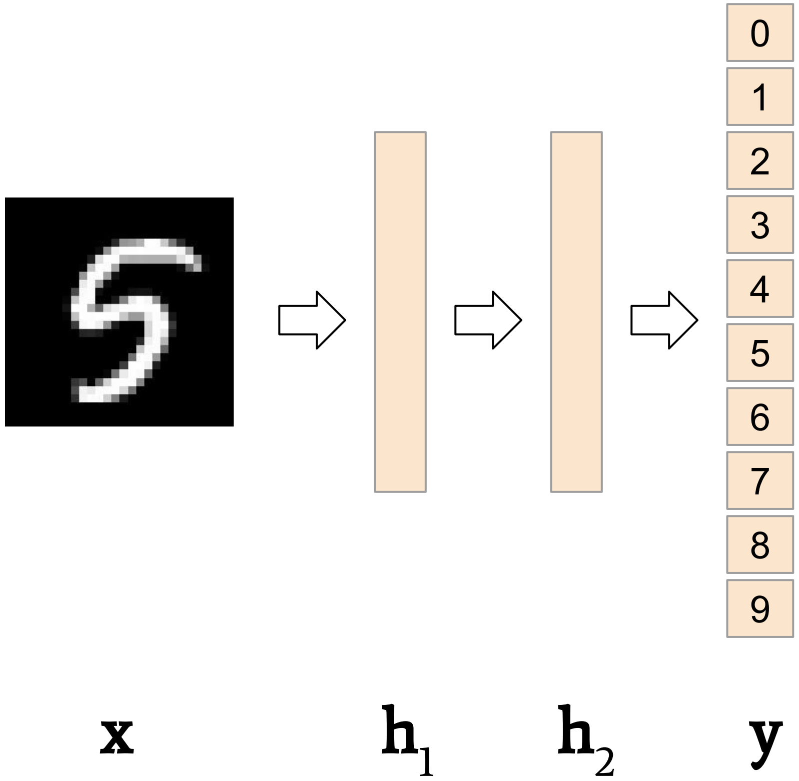
It goes down!

Loss functions

- The **loss function** measures “how wrong” the network is.
- Compares predictions $\mathbf{p} = f(\mathbf{X}; \boldsymbol{\theta})$ to targets $\mathbf{t}$ for inputs $\mathbf{X}$.
- Most common loss functions, computing a scalar “wrongness”:
  - for regression: **Mean-Squared Error** (MSE), usually with real-valued targets and linear output units
    $L(\mathbf{p}, \mathbf{t}) = \sum_i (p_i - t_i)^2$
  - for binary classification: **Binary Cross-Entropy**, usually with binary targets and sigmoid output units
    $L(\mathbf{p}, \mathbf{t}) = \sum_i -\log(p_i)\, t_i - \log(1-p_i)\, (1-t_i)$
  - for general classification: **Categorical Cross-Entropy**, usually with integer targets and softmax output units
    $L(\mathbf{p}, \mathbf{t}) = \sum_{i,c} -\log(p_{i,c})\, [t_i=c]$

Digit Recognition

Finally we've got everything in place to train a network *for real!*



The "Hello World" of Neural Nets: Digit Recognition.

Dataset: "MNIST": 60,000 hand-written digits

Network: We'll start with the one we defined before.

Loss function: Multinomial cross-entropy

Digit Recognition

Exercise:

Download `mlp.py` from `http://f0k.de/lasagne-embl/`

It defines a skeleton of:

- dataset download
- network definition
- training loop

Missing pieces:

- define loss expression
- obtain updates dictionary
- compile training function

Open the file in a text editor and fill in the missing pieces.

Open a terminal, `cd` to the file's directory and run:

```
python mlp.py
```

Validation / Testing

Example output:

```
Epoch 1 took 1.508 sec.
  training loss: 1.752
...
Epoch 5 took 1.451 sec.
  training loss: 0.354
```

We see that the “wrongness” decreases. But how well does the network do? Let’s also compute the classification accuracy.

Validation / Testing

We see that the “wrongness” decreases. But how well does the network do? Let’s also compute the classification accuracy.

```
acc = lasagne.objectives.categorical_accuracy(
      predictions, targets).mean()
train_fn = theano.function([X, targets],
                           [loss, acc],
                           updates=updates)

err = np.array([0., 0.])

print("  training loss: %.3f" % (err[0] / ...))
print("  training acc.: %.3f" % (err[1] / ...))
```

Validation / Testing

We see that the “wrongness” decreases. But how well does the network do? Let’s also compute the classification accuracy. Let’s then run it for 100 epochs at learning rate 0.2:

Validation / Testing

We see that the “wrongness” decreases. But how well does the network do? Let’s also compute the classification accuracy. Let’s then run it for 100 epochs at learning rate 0.2:

```
Epoch 1 took 0.599 sec.
  training loss: 1.248
  training acc.: 0.640
Epoch 50 took 0.555 sec.
  training loss: 0.030
  training acc.: 0.993
Epoch 100 took 0.563 sec.
  training loss: 0.006
  training acc.: 1.000
```

100% correct!

Validation / Testing

We see that the “wrongness” decreases. But how well does the network do? Let’s also compute the classification accuracy. Let’s then run it for 100 epochs at learning rate 0.2:

```
Epoch 1 took 0.599 sec.
  training loss: 1.248
  training acc.: 0.640
Epoch 50 took 0.555 sec.
  training loss: 0.030
  training acc.: 0.993
Epoch 100 took 0.563 sec.
  training loss: 0.006
  training acc.: 1.000
```

100% correct!
... on the
training set.

Validation / Testing

100% correct!
... on the
training set.

Does that mean it recognizes digits?

Validation / Testing

100% correct!
... on the
training set.

Does that mean it recognizes digits?

Not necessarily: Could have learned all examples by heart.

Validation / Testing

100% correct!
... on the
training set.

Does that mean it recognizes digits?

Not necessarily: Could have learned all examples by heart.

We hope that in order to get all training examples correct, the model had to discover something general about digits.

Important goal in Machine Learning: **Generalization**.
How well does the model perform on previously *unseen* examples?

Validation / Testing

How well does the model perform on previously *unseen* examples?

Best way to find out: Try it. Spare some training examples (the “validation set”), see how the model fares without training on them.

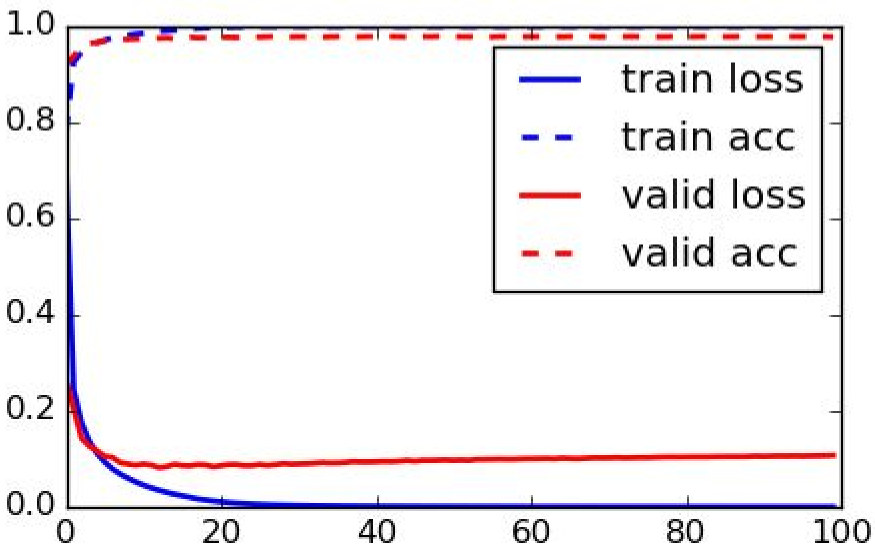
```
train_fn = theano.function([X, targets],
                           [loss, acc],
                           updates=updates)

val_fn = theano.function([X, targets],
                         [loss, acc])
```

Running this one in a loop over the validation set, after training, we get an accuracy of 97.7%.

Overfitting

Let's monitor performance on the validation set during training, after each epoch: http://f0k.de/lasagne-embl/mlp4.py

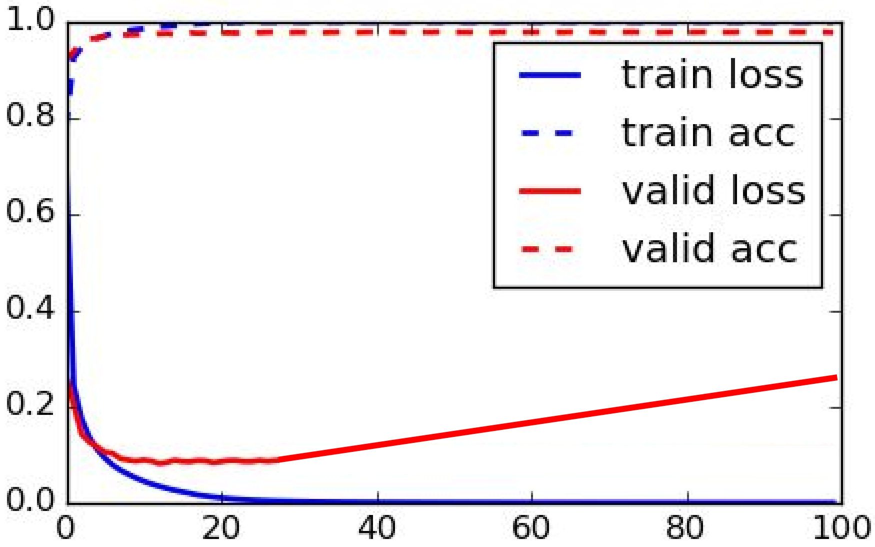
After about 10 epochs, performance improves on training set only: The model *overfits* to the training data.



Overfitting

Extreme case: After prolongued training, performance can even *decrease* on validation data while improving on training data.

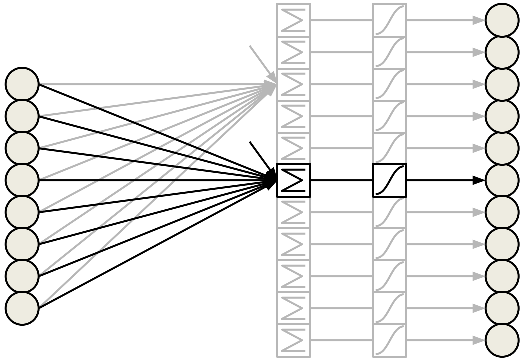
Counter-measures: Early stopping (if validation error increases),
Regularization (e.g., weight decay, dropout),
Different network architecture



Convolutions

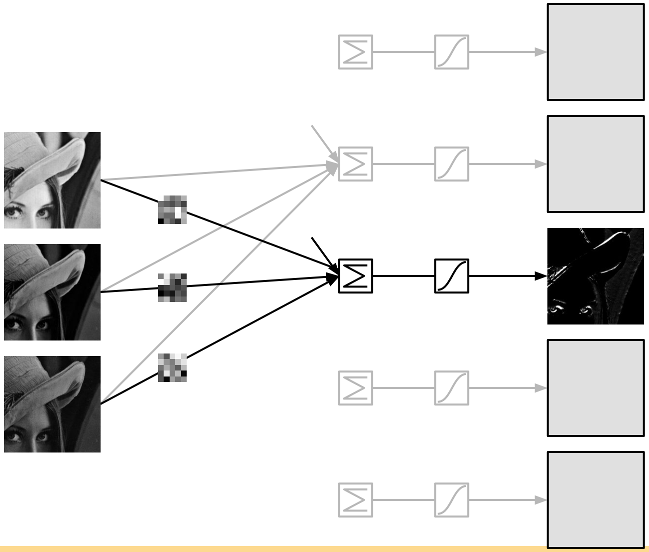
Fully-connected layer:

Each **input** is a **scalar** value, each **weight** is a **scalar** value, each output is the sum of inputs **multiplied** by weights.



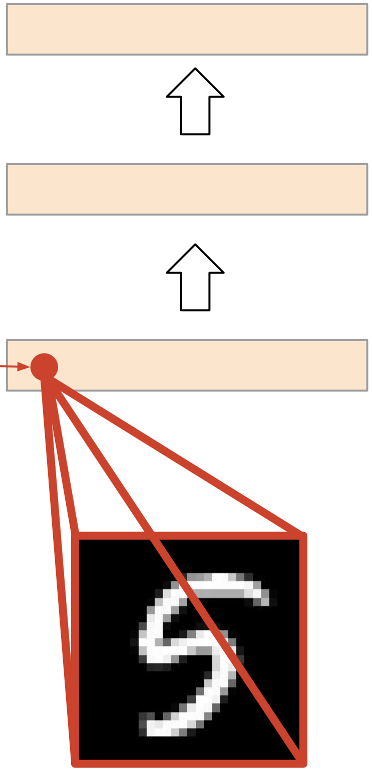
Convolutional layer:

Each **input** is a **tensor** (e.g., 2D), each **weight** is a **tensor**, each output is the sum of inputs **convolved** by weights.



Convolutions

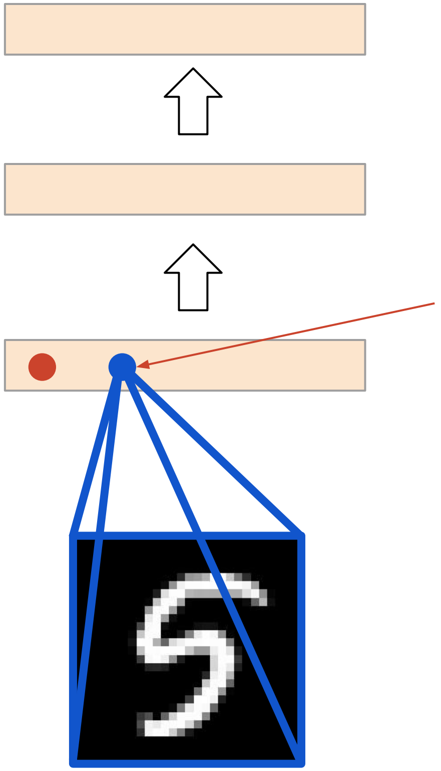
Fully-connected layer:



this unit is connected to all pixels in the image

Convolutions

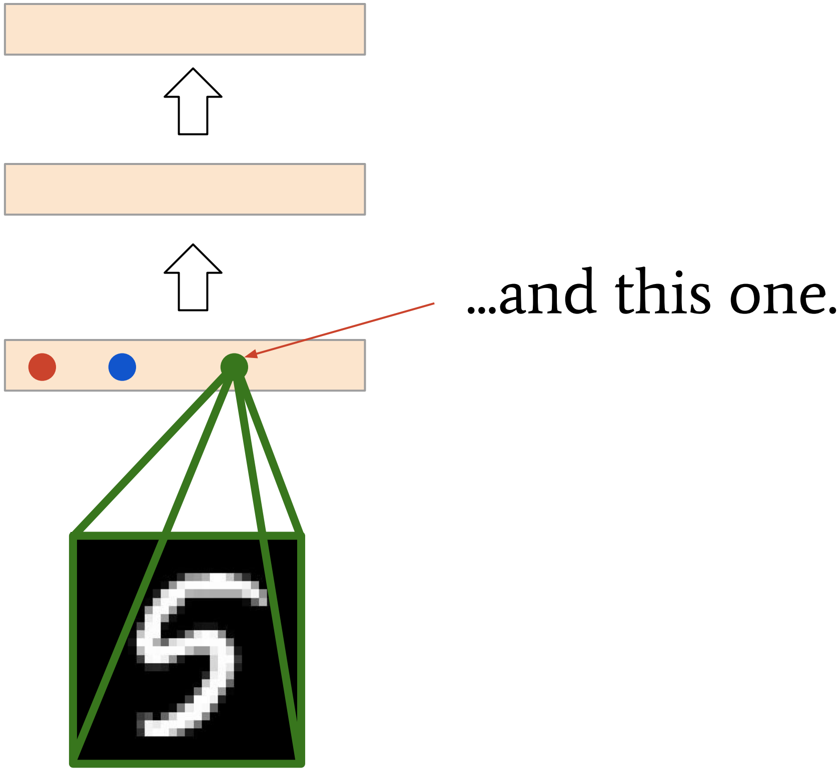
Fully-connected layer:



so is this one!

Convolutions

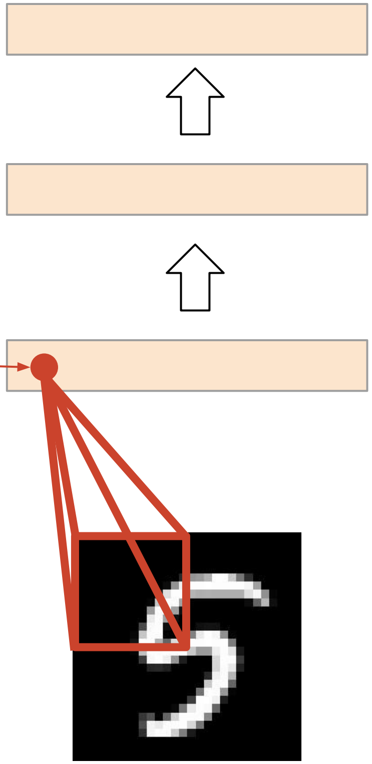
Fully-connected layer:



…and this one.

Convolutions

Locally-connected layer:

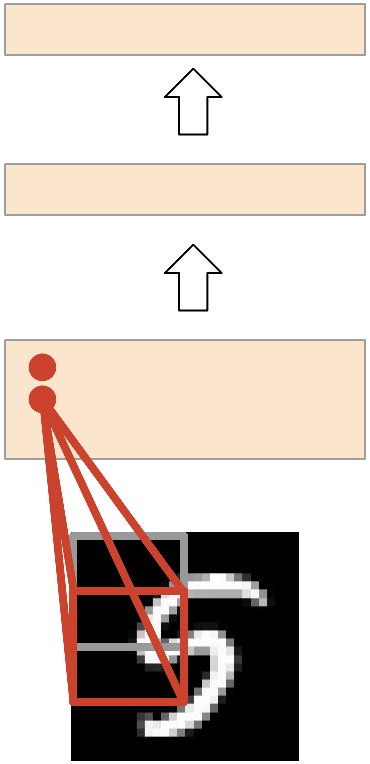


consider a unit
that only sees part
of the image

This unit can learn to
detect a **local pattern**
in the image, e.g.
a pen stroke.

Convolutions

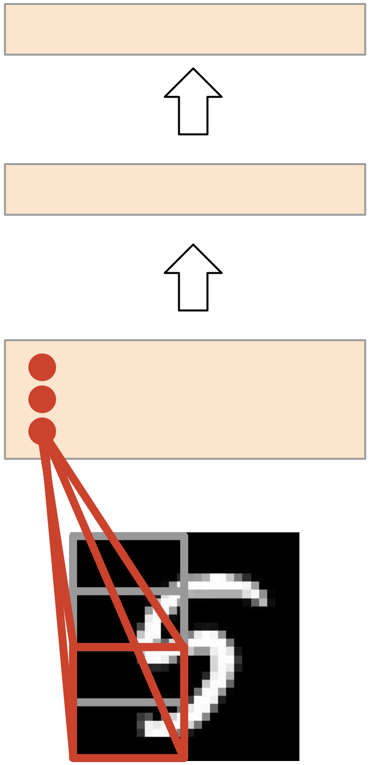
Convolutional layer:



This pattern may also occur elsewhere, so we can replicate the unit across the image.

Convolutions

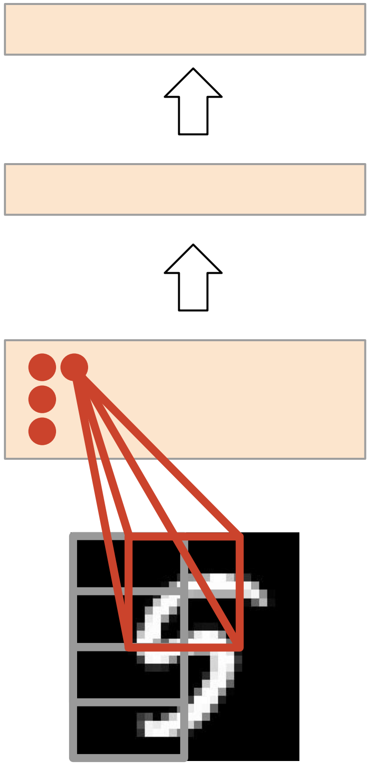
Convolutional layer:



This pattern may also occur elsewhere, so we can replicate the unit across the image.

Convolutions

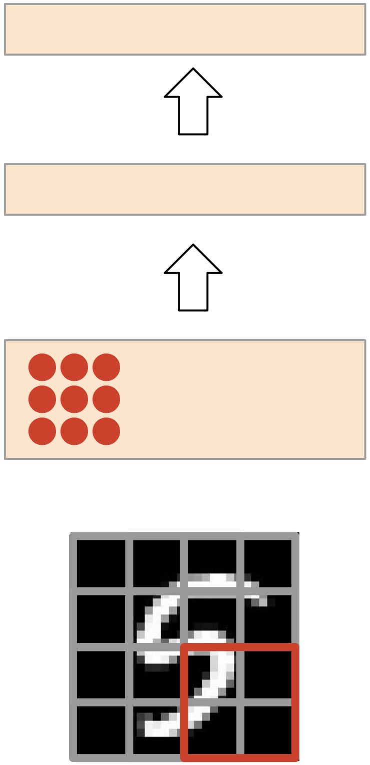
Convolutional layer:



This pattern may also occur elsewhere, so we can replicate the unit across the image.

Convolutions

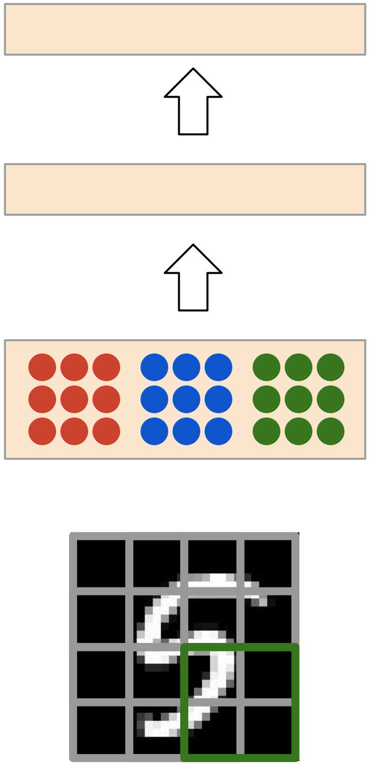
Convolutional layer:



...until we've covered all of it.

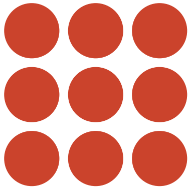
Convolutions

Convolutional layer:

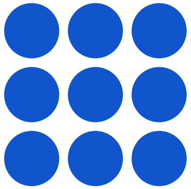


We can do the same for the other units.

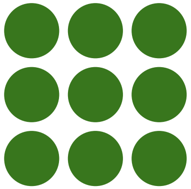
Convolutions



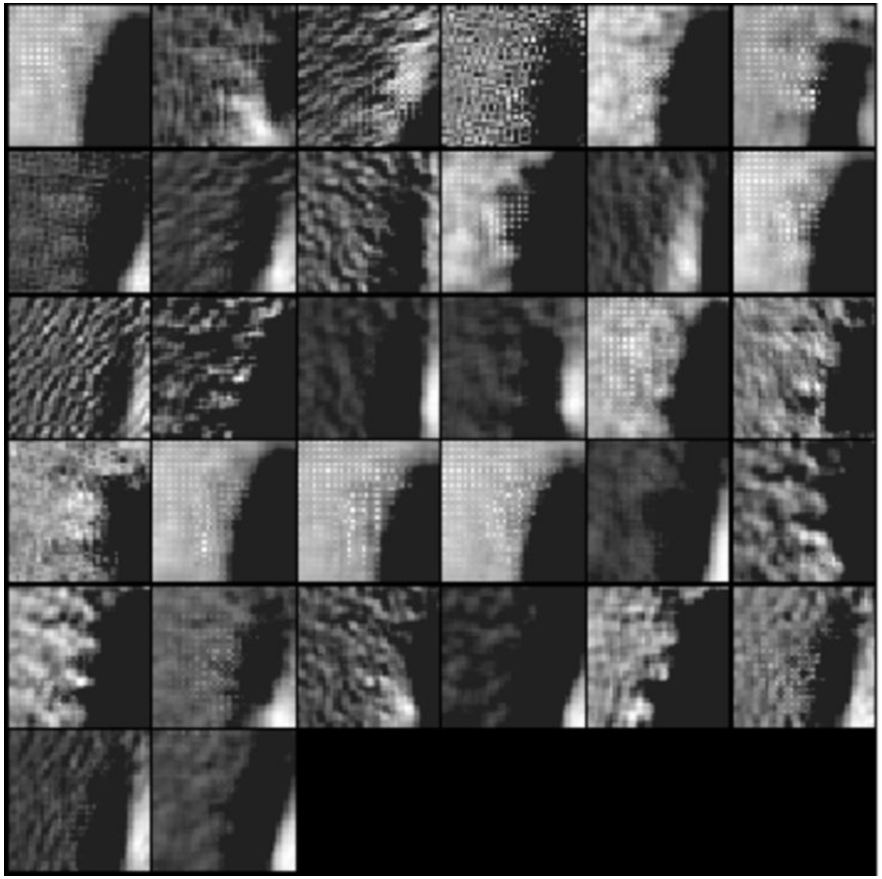
These units, detecting the same pattern in different spatial positions across the image, form a **feature plane** or **feature map** together.



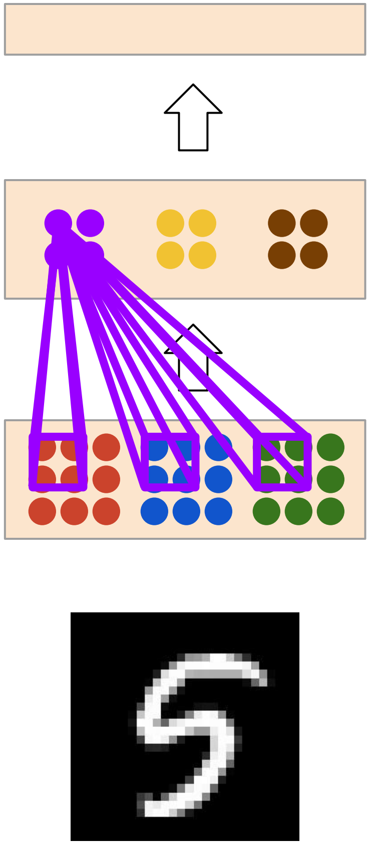
There is a separate feature map for each **type** of unit, for each pattern.



Visualising feature maps



Convolutions



We can also apply this in other layers of the network.

Now we combine local information from different feature maps.

Convolutions

$$\mathbf{Y} = \sigma(\mathbf{b} + \mathbf{X}\mathbf{W})$$

$\mathbf{X}$: ? x 784

$\mathbf{W}$: 784 x 100

$\mathbf{b}$: 100

$\mathbf{Y}$: ? x 100

⇨

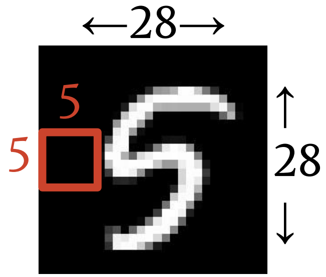
$$\mathbf{Y} = \sigma(\mathbf{b} + \mathbf{X} * \mathbf{W})$$

$\mathbf{X}$: ? x 1 x 28 x 28

$\mathbf{W}$: 100 x 1 x 5 x 5

$\mathbf{b}$: 100

$\mathbf{Y}$: ? x 100 x 24 x 24



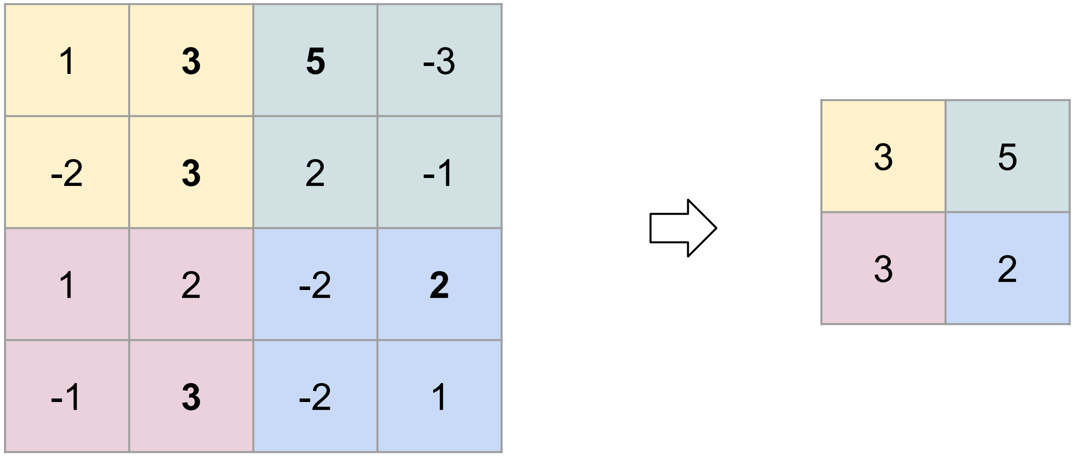
'*' represents the **convolution** operation

Pooling

A **pooling layer** reduces the size of feature maps.

It’s just a fancy name for downsampling.

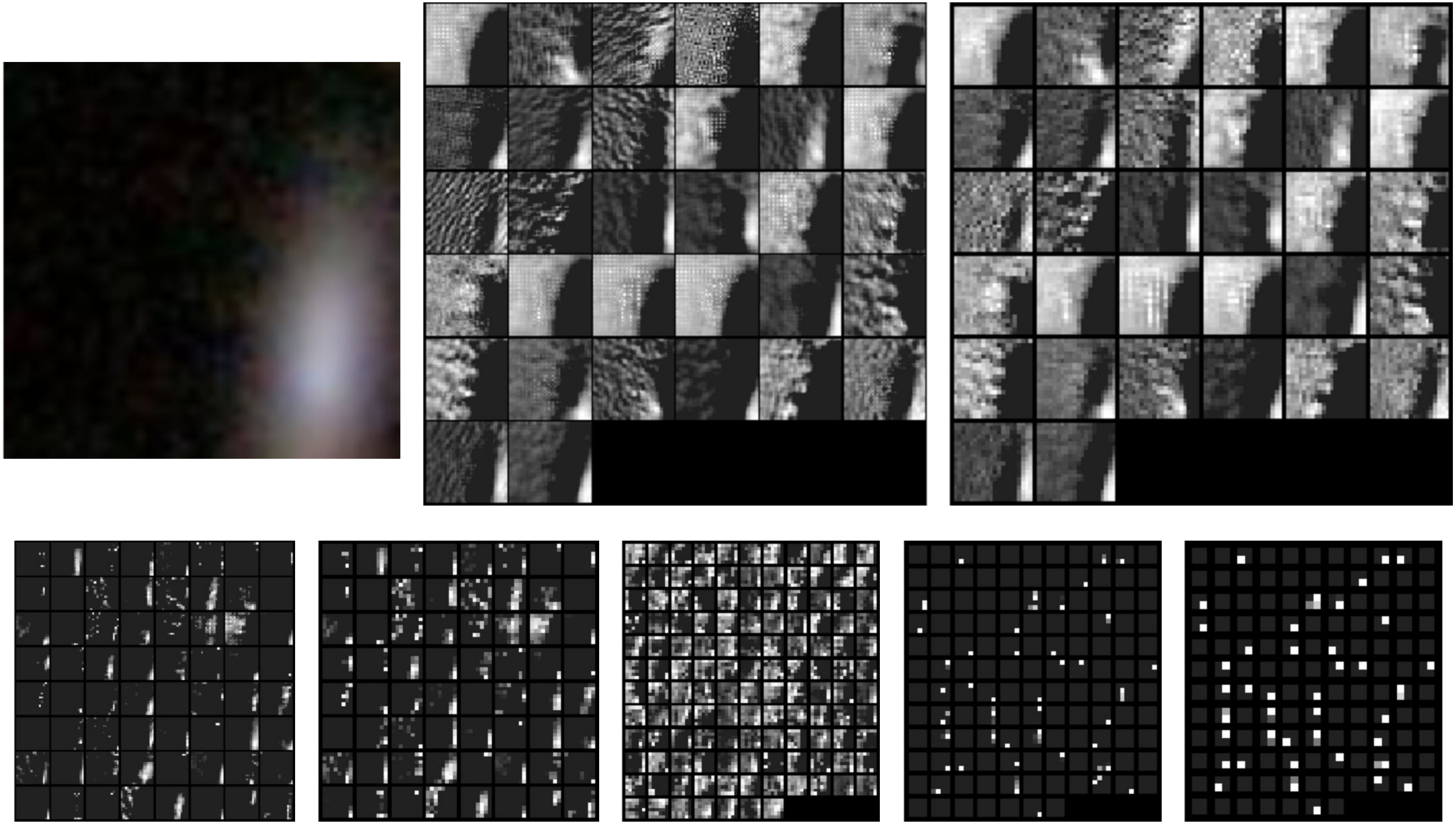
Max-pooling: take the max. activation across small regions



1	**3**	**5**	-3
-2	**3**	2	-1
1	2	-2	**2**
-1	**3**	-2	1

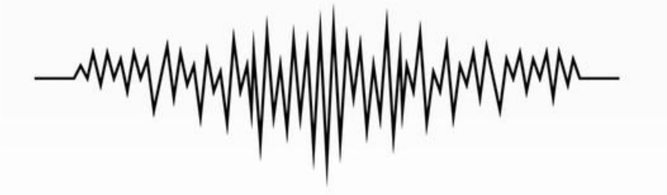
3	5
3	2

Visualising feature maps

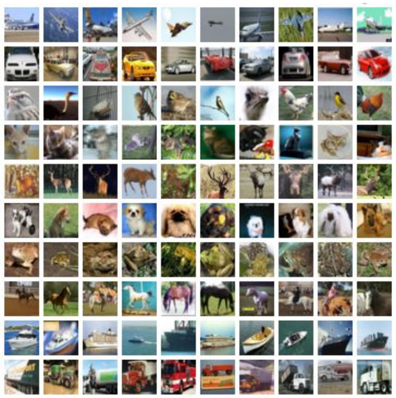


Convolutions in 1D, 2D, 3D

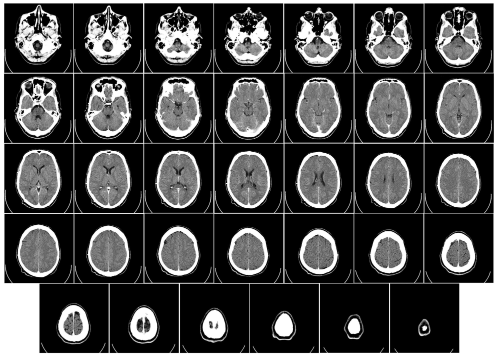
1D: time series, sound



2D: images



3D: volumetric data, video



Digit Recognition with Convolutions

Exercise:

Open your existing script or http://f0k.de/lasagne-embl/mlp4.py.

Instead of Input(784) → Dense(100) → Dense(100) → Dense(1),
use Input(1,28,28) → Conv(16, 3) → Conv(16, 3) → Pool(2)
→ Conv(32, 3) → Conv(32, 3) → Pool(2)
→ Dense(100) → Dense(1).

`Conv2DLayer(incoming, num_filters, filter_size)` and `MaxPool2DLayer(incoming, pool_size)` are the layer classes to use. Change `X` to `T.tensor4`. Set `flatten=False` for loading the dataset. When ready, run the modified script.

Digit Recognition with Convolutions

The solution is at http://f0k.de/lasagne-embl/cnn.py It also changes the network to use the rectifier nonlinearity instead of sigmoids.

Example output:

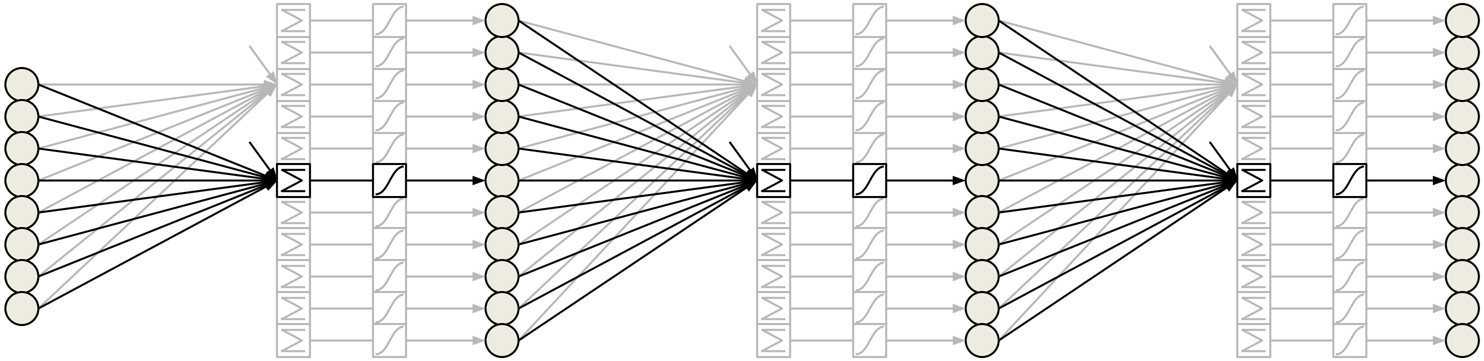
```
Epoch 36 took 1.837 sec.
  training loss: 0.000
  training acc.: 1.000
  valid.   loss: 0.071
  valid.   acc.: 0.990
```

Convolutions make assumptions that tend to work well for images (process neighboring pixels together, learn translation-invariant features), and MNIST is no exception. But we might still improve!

Trick of the Trade: Dropout

Randomly delete half of the units for each update step.

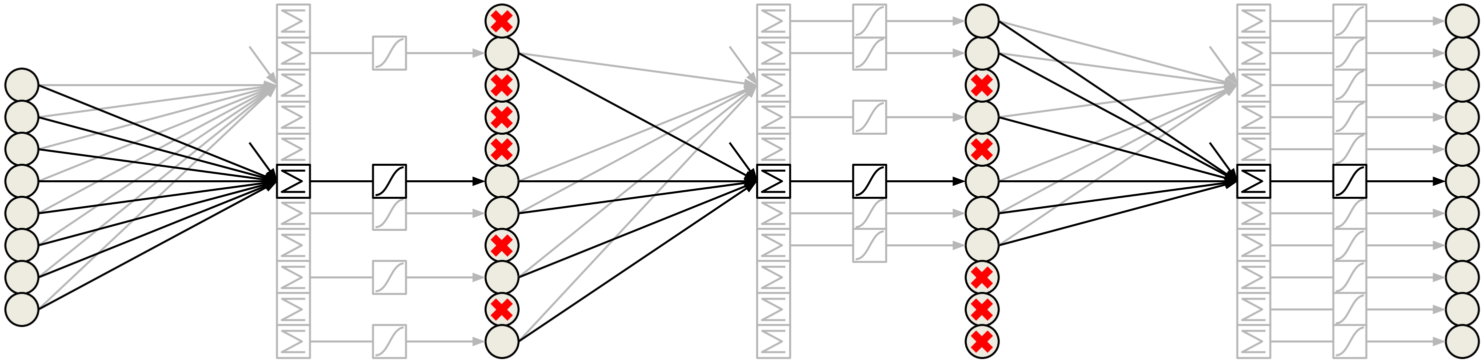
This makes the units more robust (it prevents *co-adaptation*).



Trick of the Trade: Dropout

Randomly delete half of the units for each update step.

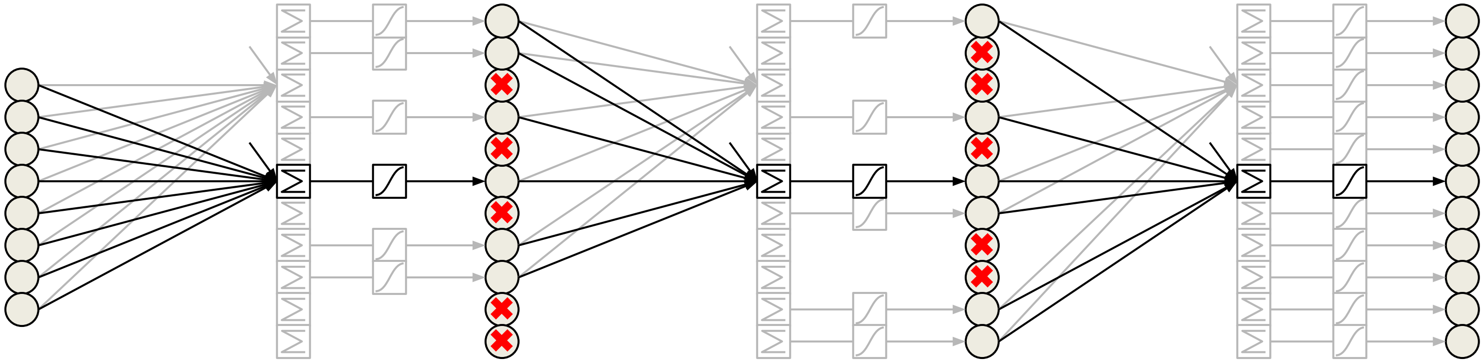
This makes the units more robust (it prevents *co-adaptation*).



Trick of the Trade: Dropout

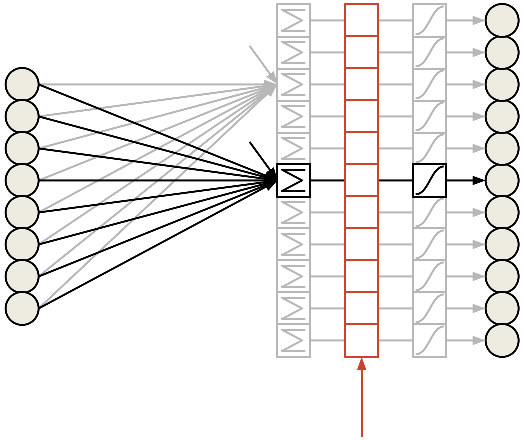
Randomly delete half of the units for each update step.

This makes the units more robust (it prevents *co-adaptation*).

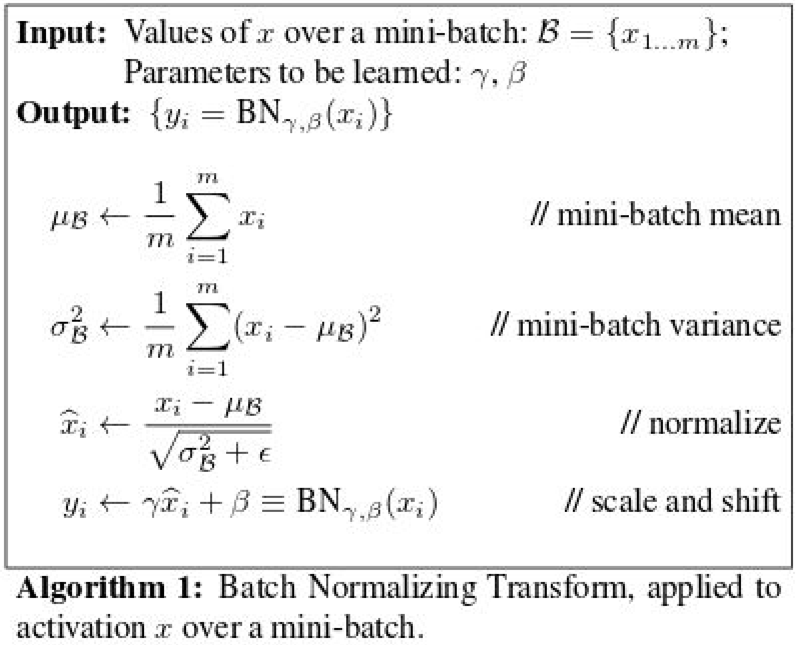


Trick of the Trade: Batch normalisation

Weight gradients depend on the unit activations.
Very small or very large activations lead to small/large gradients.
They make learning harder. For input units, Z-scoring the training data helps. Batch normalisation does the same for hidden units (countering changes).



batch normalisation



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Tricks of the trade in Lasagne

Exercise:

Open your existing script or http://f0k.de/lasagne-embl/mlp4.py or http://f0k.de/lasagne-embl/cnn.py.

1. Add dropout before each dense layer:
   `DenseLayer(., .)` → `DenseLayer(dropout(.), .)`
2. Add batch normalization after each hidden non-pooling layer:
   `SomeLayer(...)` → `batch_norm(SomeLayer(...))`
   i.e., not after the input layer and not after the output layer!

Both `dropout` and `batch_norm` are in `lasagne.layers`.

Tricks of the trade in Lasagne

Dropout / batch normalisation make predictions **nondeterministic**:
Predictions with dropout are random, predictions with batch normalisation depend on other examples in mini-batch.

For validation (or testing), we need a **deterministic** alternative:

Dropout: Scale activations by non-dropout probability.

Batch normalisation: Collect running averages during training.

Only change required in Lasagne:

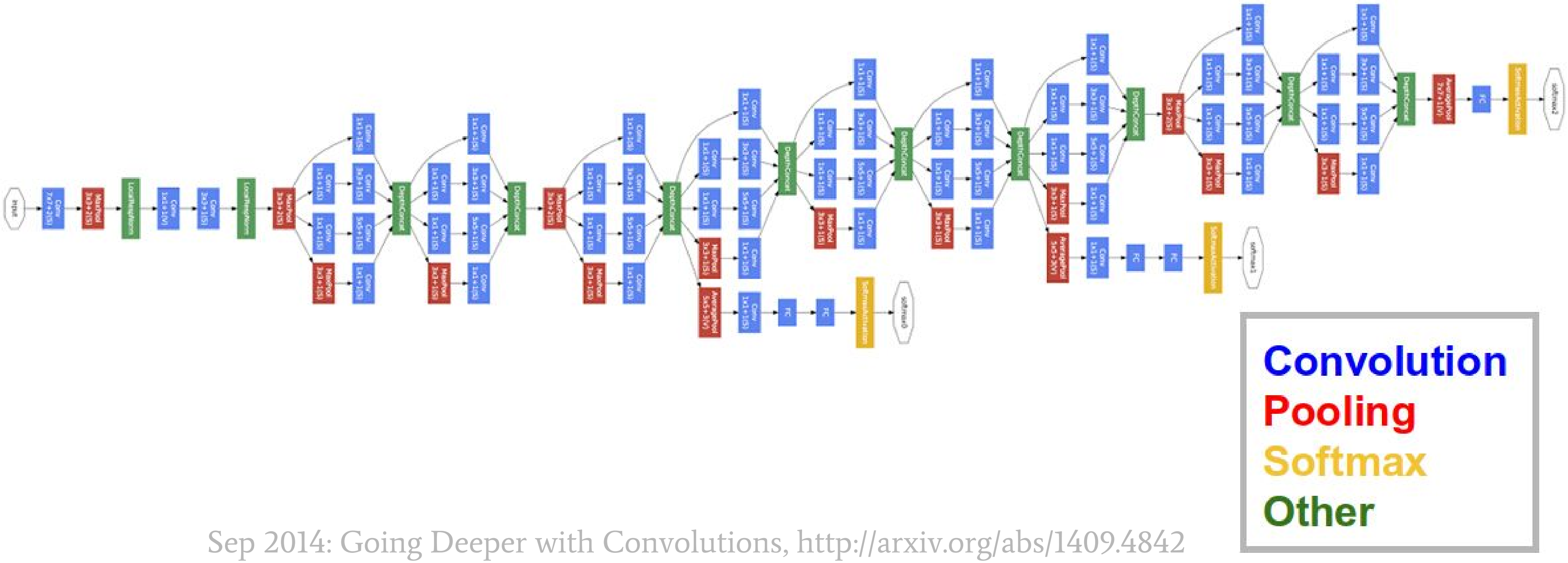
```
test_predictions = get_output(...,
               deterministic=True)
```

Use these predictions for validation loss, not for training loss!

Outlook: Going Deeper

IMAGENET Large Scale Visual Recognition Challenge:
1.2 million training images of 1000 classes

GoogLeNet: 22 layers, with **auxiliary classifiers**

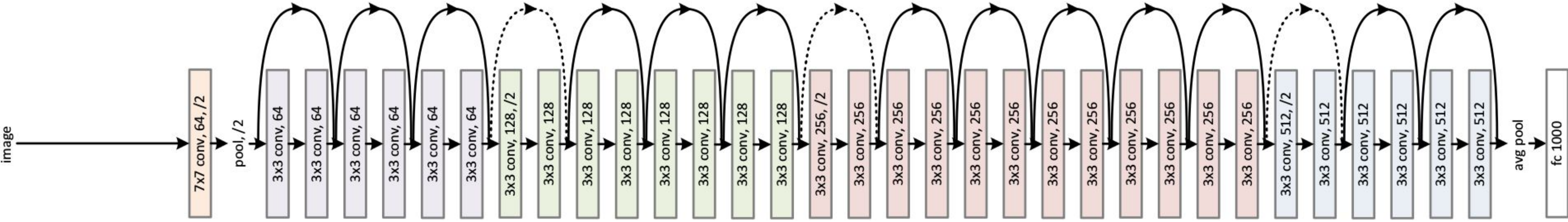


Sep 2014: Going Deeper with Convolutions, http://arxiv.org/abs/1409.4842

Outlook: Going Deeper

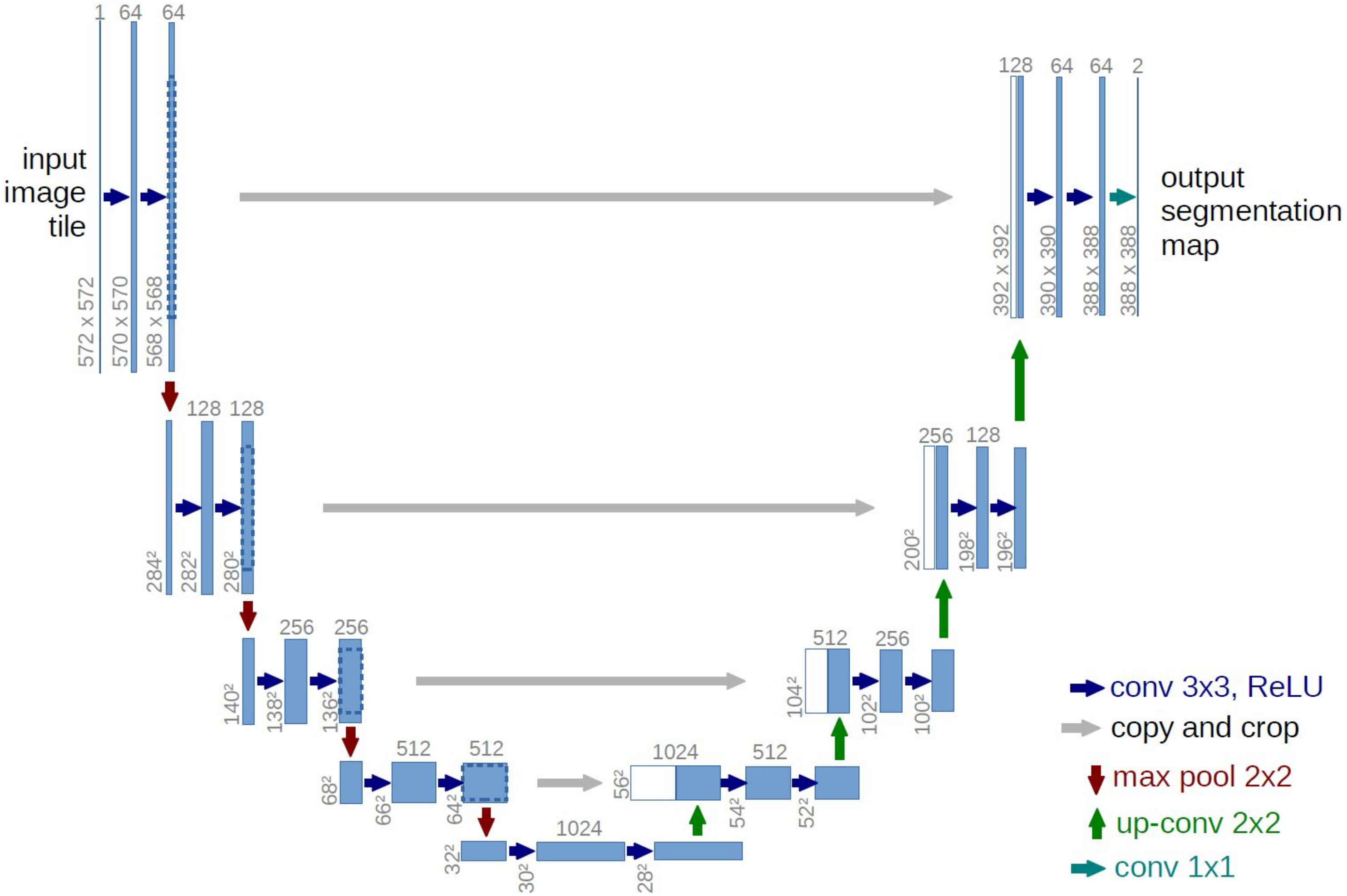
IMAGENET Large Scale Visual Recognition Challenge:
1.2 million training images of 1000 classes

ResNet: 152 layers, with **shortcut connections** (here: 38 layers)



Dec 2015: Deep Residual Learning for Image Recognition, http://arxiv.org/abs/1512.03385
Mar 2016: Identity Mappings in Deep Residual Networks, http://arxiv.org/abs/1603.05027

Outlook: Going Rounder



May 2015: U-Net: Convolutional Networks for Biomedical Image Segmentation, http://arxiv.org/abs/1505.04597

Wrap-up

Lasagne is just one of many deep learning frameworks. Popular alternatives: Caffe, Torch, Chainer, MXNet, TensorFlow, ...

+ Lasagne provides all layers and optimization methods needed to reimplement most research papers
+ Due to Theano’s symbolic differentiation, adding custom layers or loss functions is easier than in other frameworks

- Lasagne does not provide a ready-to-use training loop (but some addons do, e.g. https://github.com/dnouri/nolearn)
- Due to Theano, any new model will need initial compilation, which can take some time especially for recurrent networks

Wrap-up

This tutorial only covered the basics. Where to go from here?

- http://lasagne.readthedocs.org/en/latest/user/tutorial.html gives a walk-through for a more fully-fledged MNIST script
- https://github.com/Lasagne/Recipes is a growing collection of reimplementations of recent research papers in Lasagne
- The web page for this tutorial links to further material: http://f0k.de/lasagne-embl/
- Our mailing list is the perfect place for questions, except of course for the ones you'd like to ask right now! https://groups.google.com/forum/#!forum/lasagne-users