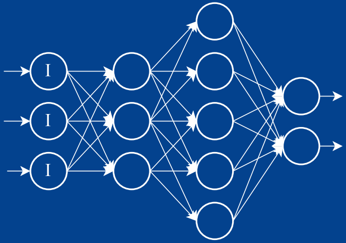
AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

MACHINE LEARNING

LESSON 11: Deep Learning II—Training

CARSTEN EIE FRIGAARD

SPRING 2019



L09: Deep Learning II: Agenda

- J3: Criteria for evaluation on BB.
- J3: Poster-session or Journal Hand-in?

- RESUMÉ
  - The Perceptron and MLP's
  - Neural Network Framework

- Training NN's: Backpropagation
  - Single-layer Perceptron Learning Rule
  - Multi-layer Perceptron Learning Rule
  - Training NN's: Backpropagation
- Training NN's: Practical Guidelines
  - Batch Normalization
  - Regularization
  - Optimizers + Learning rate schedule
- Data Augmentation
- Model Zoos and Transferred Learning

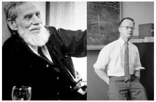
RESUMÉ: History of Neural Networks

Important Discoveries in the History of NN

- 1670: Leibniz, L'Hôpital, chain rule differentiation
- 1805/9: Legendre and Gauss, least-squares
- 1847: Cauchy, numerical gradient descent
- 1901: Pearson, PCA
- 1936: Fisher, linear discriminant analysis of iris data (probability theory: F-distribution)
- 1943: McCulloch and Pitts neuron
- 1949: Hebbs rule for self-organized learning
- 1957: Rosenblatts perceptron and supervised learning
- 1960-70: Multilayer perceptrons
- 1969: Minsky and Papert, XOR crisis, first winter
- 1985-86: BProp, LeCun, Rumelhart et at. second summer
- 95'ish: second winter
- 2006'ish: GPUs, start of third summer
- 2009: Jarrett, reLU activation function
- 2022'ish: third winter?

t


Mr. Fisher


Mr. McCulloch-Pitts


Mr. Rosenblatt


Mr. LeCun


Bprop Summer

RESUMÉ: The Perceptron

Definition

History:

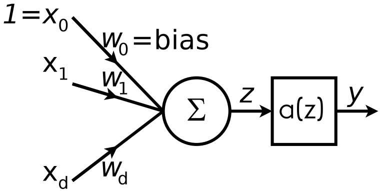
- 1943: McCulloch-Pitts: artificial neuron + network.
- 1957: Rosenblatt: the perceptron. Based on linear regressor + Heaviside activation function.

A linear regressor, with $\mathbf{x} \equiv [1 \; x_1 \; x_2 \; \cdots \; x_d]^\top$

$$\begin{aligned} z &= \mathbf{w}^\top \mathbf{x} \\ &= w_0 \cdot 1 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d \end{aligned}$$

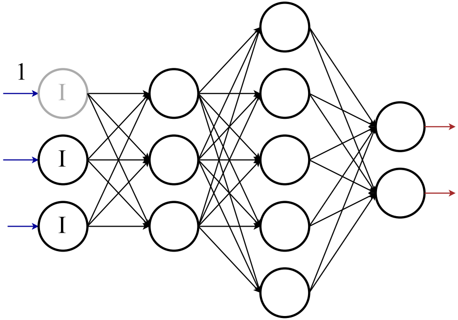
plus activation function = the Perceptron (linear-threshold unit, LTU)

$$y_{neuron}(\mathbf{x}; \mathbf{w}) = a(z) = a(\mathbf{w}^\top \mathbf{x})$$



RESUMÉ: Multi-layer Perceptrons (MLPs)

MLP, three layers, fully connected, simplified nodes...



MLP nomenclature:

input layer: handles the input **x** data,

output layer: only visible output signal from the network,

hidden layer(s): internal layers in the network,

fully connected: all nodes in layer connected to all neurons in the previous/next layer,

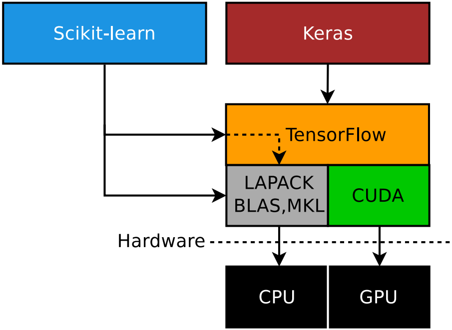
feed-forward: the signals flows only forward in the network (in contrast to feed-backward in BProp),

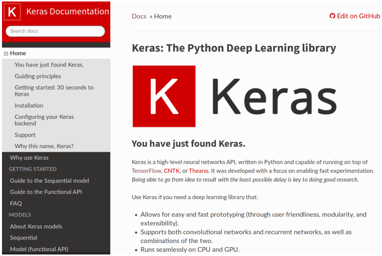
Backpropagation: or BProp, training algo of NN, more in later lesson,

Deep-learning networks: MLP with several hidden layers, say >2 ?

RESUMÉ: Keras and Tensorflow

Using the Keras API instead of Scikit-learn or TensorFlow





NOTE:

- documentation: `https://keras.io/`
- keras provides a `fit-predict`-interface,
- many similiarities to Scikit-learn,
- but also many differences!

Other Neural Network Frameworks

TensorFlow: Python, C++, Google,
Caffe/2: Python, University of California, Berkeley,
Theano: Python, Montreal Institute for Learning Algorithms,
MXNet: Python, C++, Apache
Torch/PyTorch: Facebook,
CNTK: Microsoft Cognitive Toolkit,
Matlab: Proprietary.

Software	Creator	Initial Release	Software license[a]	Open source	Platform	Written in	Interface	OpenMP support	OpenCL support	CUDA support	Automatic differentiation[1]	Has pretrai mode
BigDL	Jason Dai (Intel)	2016	Apache 2.0	Yes	Apache Spark	Scala	Scala, Python			No		Ye
Caffe	Berkeley Vision and Learning Center	2013	BSD	Yes	Linux, macOS, Windows[2]	C++	Python, MATLAB, C++	Yes	Under development[3]	Yes	Yes	Yes
Chainer	Preferred Networks	2015	BSD	Yes	Linux, macOS	Python	Python	No	No	Yes	Yes	Ye

[https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software]

RESUMÉ: Multi-layer Perceptrons (MLPs)

From Perceptron training to MLP Training

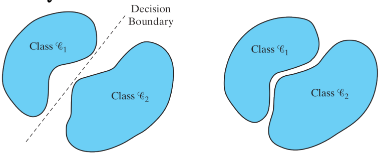
Training perceptrons and Hebb's postulate:

> *"Cells that fire together wire together."*

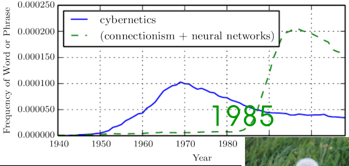
and

> *[..] explain synaptic plasticity, the adaptation of brain neurons during the learning process.*

[https://en.wikipedia.org/wiki/Hebbian_theory]



- for linearly separable problems: perceptron convergence theorem,
- but what about XOR problems: Minsky/Papert XOR crisis,
- and how do you train MLP's?



No method existed until..

1985/86: LeCun, Rumelhart et al.: Backpropagation

RESUMÉ: Num. Solution: Gradient Descent

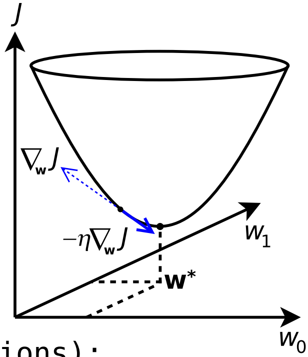
The GD Algorithm

First, find the deriverty of J, via $\nabla_{\mathbf{w}} J$, that after some matrix algebra gives

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{2}{m} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$$

Then move along in the *opposite* direction of this gradient, taking a step of size η

$$\mathbf{w}^{(step\ N+1)} = \mathbf{w}^{(step\ N)} - \eta \nabla_{\mathbf{w}} J(\mathbf{w})$$



The SG-algo in python code

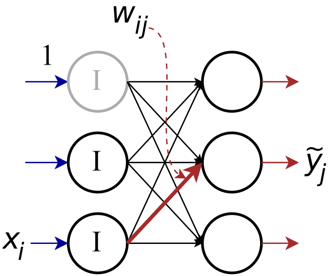
```
for iteration in range(n_iterations):
    gradients=2/m*X_b.T.dot(X_b.dot(theta)-y)
    theta=theta - eta * gradients
```

NOTE: The `X_b` is a `X` with a all-1 column prepended, and using the constant factor $2/m$ instead of just $1/2$ and with $\mathbf{w} = \boldsymbol{\theta}$

Perceptron Learning Rule

Weight Update for Rosenblatt Perceptron Single Layer Network

$$w_{ij} = w_{ij} + \eta\,(y_j - \hat{y}_j)\,x_i$$



perceptron: a single-layer network, not a single neuron,

$a(z)$: Rosenblatt acti-fun, Heaviside,

w_{ij}: weight from input neuron *i* to output neuron *j*,

x_i: *i'th* input data,

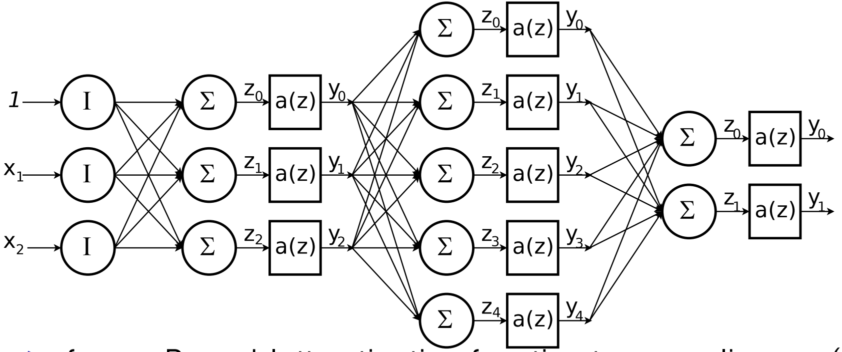
$\hat{y}_i$: *j'th* output neuron value ($y_{\text{j,pred}}$),

y_j: *j'th* true output ($y_{\text{j,true}}$),

η: learning rate

Multi-layer Perceptron Learning Rule

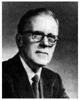
Weight Update an MLP, using Non-Linear Activation Function?



- from a Rosenblatt activation function to a non-linear $a(z)$
  $(y - \hat{y})x \rightarrow \nabla J$,
- init of weights: random values,
- output layer: trained via Perceptron Learning Rule,
- hidden layers: trained via ??

Training NN's: Backpropagation

Feed-forward and Feed-back phases


Mr. Hebb

- Training a single-layer Perceptron network is easy: by the least- mean-square rule, via $\nabla_{\mathbf{w}} J$,
- Hebbs rule and Perceptron Learning Rule: insufficient for a Multi-layer Perceptron network,


Mr. Rumelhart

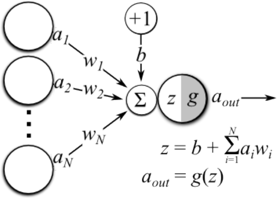
- Backpropagation: (re)-invented 1985-86, LeCun and Rumelhart et al.


Mr. LeCun

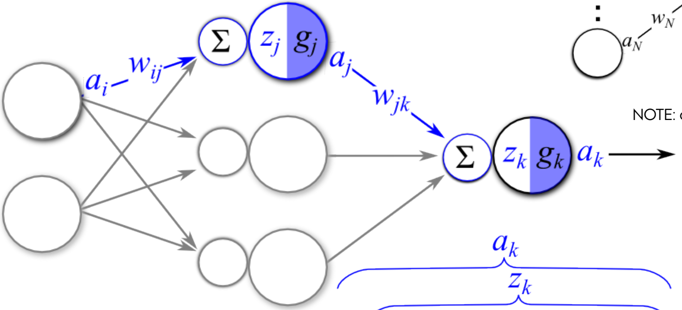
- Basic BProp:
  - forward-pass: signal only flows forward during the prediction phase,
  - backward-pass: when adjusting the weights in the training phase,
  - **weight updates in the backward-pass: uses the gradient of the error and partial derivatives/chain rule to adjust each weight in a 'neuron' with respect to is contribution to the total error...**

Training NN's: Backpropagation 1-2-3-4

1: Forward-Propagate Input Signals



NOTE: different notation

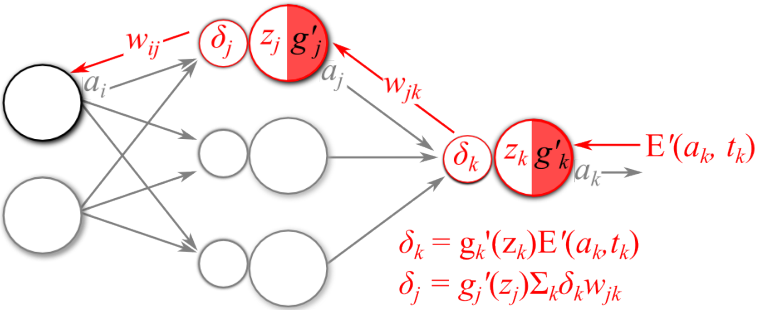


$$a_k = \overbrace{g_k(\overbrace{b_k + \Sigma_j \underbrace{g_j(\underbrace{b_j + \Sigma_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k})}^{a_k}$$

[https://theclevermachine.wordpress.com/category/algorithms/gradient-descent/]

Training NN's: Backpropagation 1-2-3-4

2: Back-propagate Error Signals

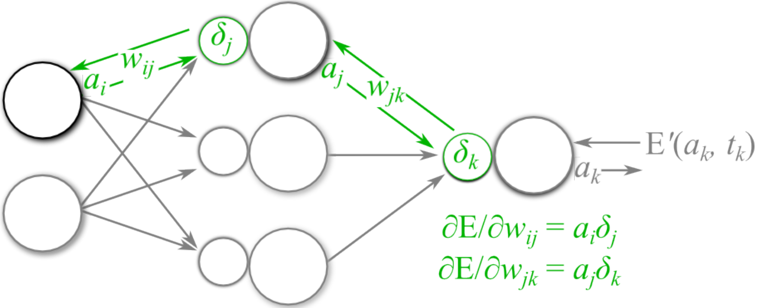


$$\delta_k = g_k'(z_k)E'(a_k, t_k)$$
$$\delta_j = g_j'(z_j)\Sigma_k \delta_k w_{jk}$$

[https://theclevermachine.wordpress.com/category/algorithms/gradient-descent/]

Training NN's: Backpropagation 1-2-3-4

3: Calculate Weight Gradients



$$\partial E/\partial w_{ij} = a_i\delta_j$$
$$\partial E/\partial w_{jk} = a_j\delta_k$$

4: Update Weights

$$w_{ij} = w_{ij} - \eta(\partial E/\partial w_{ij})$$
$$w_{jk} = w_{jk} - \eta(\partial E/\partial w_{jk})$$

for learning rate η

[https://theclevermachine.wordpress.com/category/algorithms/gradient-descent/]

A 123-line Neural Network in Python

A three-layer MLP with Backpropagation [L11/nn_demo.ipynb]

```python
# A Neural Network in Python, three layer MLP with activationfunction and BProp

# NOTE: transfer() and transfer_derivative() defined here, on outer level, to allow for later modification
# Transfer neuron activation,
def transfer(z):
    # a(z) = 1/(1+exp(-z))
    return 1.0 / (1.0 + exp(-z))

# Calculate the derivative of an neuron output
def transfer_derivative(output):
    # for a(z) = 1/(1+exp(-z))
    #  a'(z) = d(a(z)) / dz = exp(-z)/ ((1+exp(-z)+1)^2)
    #        = a(z)*(1-a(z))
    # NOTE: no need to recalc anything, just use a(z)/output to
    #       fast find the deriverty!
    # [https://en.wikipedia.org/wiki/Backpropagation]
    #
    return output * (1.0 - output)

# Forward propagate input to a network output
def forward_propagate(network, row):

    # Calculate neuron activation for an input
    def activate(weights, inputs):
        activation = weights[-1]
        for i in range(len(weights)-1):
            activation += weights[i] * inputs[i]
        return activation

    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):

    # Backpropagate error and store in neurons
    def backward_propagate_error(network, expected):
        for i in reversed(range(len(network))):
            layer = network[i]
            errors = list()
            if i != len(network)-1:
                for j in range(len(layer)):
                    error = 0.0
                    for neuron in network[i + 1]:
                        error += (neuron['weights'][j] * neuron['delta'])
                    errors.append(error)
            else:
                for j in range(len(layer)):
                    neuron = layer[j]
                    errors.append(expected[j] - neuron['output'])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

```python
    #print("  shape of train=",[len(train),len(train[0])])
    assert l_rate>0
    assert n_epoch>0
    assert len(train)>0
    assert len(train[0])>0
    assert n_outputs>0

    for epoch in range(n_epoch):
        for row in train:
            outputs = forward_propagate(network, row)
            # NOTE: the following is in effect a to_categorical() fun
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            #print("expected=",expected)

            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)

# Make a prediction with a network for a single row
def predict_row(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))

# Make a prediction with a network for a full dataset
def predict_network(network, test):
    predictions = list()
    for row in test:
        prediction = predict_row(network, row)
        predictions.append(prediction)
    return(predictions)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    #print("    init: n_inputs=",n_inputs,", n_hidden=",n_hidden,", n_outputs=",n_outputs)
    network = list()
    hidden_layer = [{'weights':[random.random() for i in range(n_inputs + 1)]} for i in range(n_hidden)
    network.append(hidden_layer)
    output_layer = [{'weights':[random.random() for i in range(n_hidden + 1)]} for i in range(n_outputs
    network.append(output_layer)
    return network

# Backpropagation Algorithm With Stochastic Gradient Descent
def train_predict_backprop(train, test, l_rate, n_epoch, n_hidden):
    n_inputs = len(train[0]) - 1
    n_outputs = len(set([row[-1] for row in train]))
    print("    back_propagation: n_inputs=",n_inputs,", n_hidden=",n_hidden," n_outputs=",n_outputs,",
    network = initialize_network(n_inputs, n_hidden, n_outputs)

    train_network(network, train, l_rate, n_epoch, n_outputs)
    return predict_network(network, test)
```

Training NN's: Backpropagation

Backpropagation in Python

```
# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
  for i in reversed(range(len(network))): # i: layer2, 1, 0
    layer = network[i]
    errors = list()
    if i != len(network)-1:
      for j in range(len(layer)): # j: neuron0,1,..in layer i
        error = 0.0
        for n in network[i + 1]:  # n: neuron0,1,..in layer i+1
          error += (n['weights'][j] * n['delta'])
        errors.append(error)
    else:
      for j in range(len(layer)): # j: neuron0,1,..in layer i
        n = layer[j]              # get the neuron
        errors.append(expected[j] - n['output'])

    for j in range(len(layer)):   # j: neuron0,1,..in layer i
      n = layer[j]                # get the neuron
      n['delta'] = errors[j] * d_transfer(n['output'])
```

NOTE: not 100% same notation as in previous slides...just here to demo the 'small'ness of BProb in code.

Training NN's: Backpropagation

Update Weights in Python

Using the 'new' Perceptron notation

$$\begin{aligned} w_{ij} &= w_{ij} - \eta \, \frac{\partial E}{\partial w_{ij}} \\ &= w_{ij} - \eta \, a_i \delta_j \end{aligned}$$

That in Python can be implemented as simple as

```python
# Update network weights with error
def update_weights(network, row, l_rate):
  for i in range(len(network)):
    inputs = row[:-1]
    if i != 0:
      inputs = [n['output'] for n in network[i - 1]]
    for n in network[i]:
      for j in range(len(inputs)):
        n['weights'][j] += l_rate * n['delta'] * inputs[j]
      n['weights'][-1] += l_rate * n['delta'] # input[-1]==1
```

NOTE: remember code and math with different notation. Code just an example.

Training NN's: Practical Guidelines

Default DNN configuration, [HOML,p312]

Initialization	He initialization
Activation function	ELU
Normalization	**batch normalization***
Regularization	**dropout**
Optimizer	**Nesterov accelerated gradient**†
Learning rate schedule	**none**‡

Notes:

*: As known from lesson L07 and "Preprocessing of Data".

†: Is this really the best optimizer? Why not Adam?

‡: No adaptive learning rate? Then learning can be slow!

Training NN's: Practical Guidelines

Normalization: Example of Bad Data Scaling, MNIST as Raw (float) Image Data

```
X, y = dataloaders.MNIST_GetDataSet(fetchmode=False)

X = X.reshape(70000, 784)

# NOTE 0: ups, remembered convert to float but forgot scale
X_norm = X*np.float32(1)
# NOTE 1: remembered convert to float and scale
#X_norm = X/np.float32(255)

print(f"X_norm.shape={X_norm.shape}")
print(f"  type(X_norm[0][0])={type(X_norm[0][0])}")
print(f"  X_norm.dtype={X_norm.dtype}")
print(f"  np.max(X_norm)={np.max(X_norm)}")
print(f"  np.min(X_norm)={np.min(X_norm)}")
```

prints:

```
X_norm.shape=(70000, 784)
    type(X_norm[0][0])=<class 'numpy.float32'>
    X_norm.dtype=float32
    np.max(X_norm)=255.0
    np.min(X_norm)=0.0
```

Training NN's: Practical Guidelines

Normalization: Solution I, Preprocess Scaling

```python
X, y = dataloaders.MNIST_GetDataSet(fetchmode=False)

X = X.reshape(70000, 784)

# NOTE 0: ups, remembered convert to float but forgot scale
#X_norm = X*np.float32(1)
# NOTE 1: remembered convert to float and scale
X_norm = X/np.float32(255)

print(f"X_norm.shape={X_norm.shape}")
print(f"  type(X_norm[0][0])={type(X_norm[0][0])}")
print(f"  X_norm.dtype={X_norm.dtype}")
print(f"  np.max(X_norm)={np.max(X_norm)}")
print(f"  np.min(X_norm)={np.min(X_norm)}")
```

prints:

```
X_norm.shape=(70000, 784)
    type(X_norm[0][0])=<class 'numpy.float32'>
    X_norm.dtype=float32
    np.max(X_norm)=1.0
    np.min(X_norm)=0.0
```

Training NN's: Practical Guidelines

Normalization: Solution II, Batch Normalization

Remember lesson "Preprocessing of Data:

Standardization of a feature vector $\mathbf{x}$, *giving* $\mathbf{x}'$ *mean zero, and standard deviation one*

$$\mathbf{x}' = \frac{\mathbf{x} - \mu_{\mathbf{x}}}{\sigma_{\mathbf{x}}} \simeq \frac{\mathbf{x} - \mu_{\mathbf{x}}}{\sqrt{\sigma_{\mathbf{x}}^2 + \epsilon}}$$

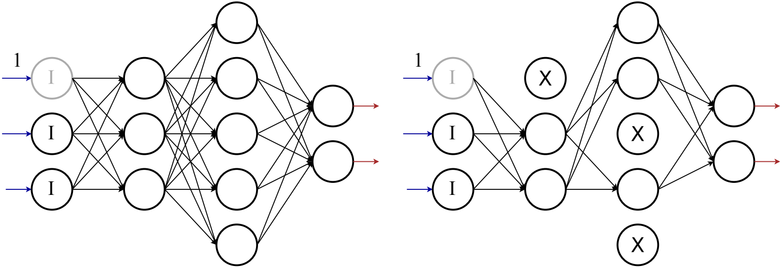
Put such a 'layer' into action, overcoming the MNIST image data scaling problem:

```
model = Sequential()
model.add(BatchNormalization())
model.add(Dense(input_dim=(784),
    units=20,
    activation="elu",
    kernel_initializer="he_normal",
    bias_initializer  ="he_normal")
)
model.add(Dense(units=..,
..
```

without scaling, without `BatchNormalization`:	
Test loss:	14.6
Test accuracy:	**0.095**
without scaling, with `BatchNormalization` (solution II):	
Test loss:	0.18
Test accuracy:	**0.96**
with scaling, without `BatchNormalization` (solution I):	
Test loss:	0.14
Test accuracy:	**0.96**

Training NN's: Practical Guidelines

Regularization: Dropout and $\mathcal{L}_1$ $\mathcal{L}_2$ Penalties




```
model.add(Dense(units=3, ..
model.add(Dropout(rate=0.3))
model.add(Dense(units=5, ..
```

Keras regulizers, with no/sparse documentation

- `kernel_regularizer`
- `bias_regularizer`
- `activity_regularizer`

```
model.add(Dense(64,
                kernel_regularizer  =regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))}
```

Training NN's: Practical Guidelines

Optimizer: Nesterov accelerated gradient (not Adam anymore)

Learning rate schedule: none

This book initially recommended using Adam optimization, because it was generally considered faster and better than other methods. However, a 2017 paper (*https://goo.gl/NAkWIa*)[17] by Ashia C. Wilson et al. showed that adaptive optimization methods (i.e., AdaGrad, RMSProp and Adam optimization) can lead to solutions that generalize poorly on some datasets. So you may want to stick to Momentum optimization or Nesterov Accelerated Gradient for now, until researchers have a better understanding of this issue.

[HOML,p302]

Remember: Moon-data needed large learing rate, say $\eta = 0.1$, for both `Adam` and `SGD`. (vanishing-gradient-problem?)

NOT so for MNIST data and an MLP (exploding-gradient-problem?)

New kid-on-the-block: Nadam (Nesterov adam)

CONCLUSION: no optimizer+η fits all data!

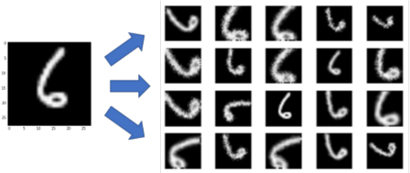
MLP-a-la-ITMALGRP10, MNIST data, epochs=3

`Adam (lr=0.1)`	
Test loss:	14.6
Test accuracy:	**0.092**
`Adam (lr=0.01)`	
Test loss:	1.8
Test accuracy:	**0.26**
`Nadam(lr=0.002)`	
Test loss:	0.17
Test accuracy:	**0.95**
`SGD (lr=0.01, ..., nesterov=True)`	
Test loss:	0.18
Test accuracy:	**0.95**

Data Augmentation

The new problem in ML: labelling data is time-consuming.

If not enough data or time: add synthetic data!



With image data, say by

- image translation,
- image rotation/mirroring,
- image scaling,
- image cropping
- image intensity scaling,
- image colour jittering,
- adding noise.

WARN: augmentation can introduce statistical side-effects!

Model Zoos and Transferred Learning

We examined the data zoo site, `www.kaggle.com`.
→ Similar **model zoo** sites exist.

Find a network 'similar' to your problem and use **transferred leaning**

- `github.com/tensorflow/models`
- `github.com/BVLC/caffe/wiki/Model-Zoo`

Model `<your-name-here>Net` names like

- `LeNet`,
- `AlexNet`,
- `GoogLeNet`,
- `VGGNet`,
- `ResNet`,
- `Inception, Xception`

...or cross-convert models from, say Caffe2/Theano/Torch?