Deep Learning Tutorial

HAP Workshop – Big Data Science in Astroparticle Physics

David Walz

RWTH Aachen

20.02.2017

Tutorial Overview

- Deep Learning Fundamentals
  - → Neural Networks
  - → Optimization
  - → Generalization
- Image Classification
- TensorFlow & Keras
- Practice Session 1: Image Classification with Neural Networks

Coffee break

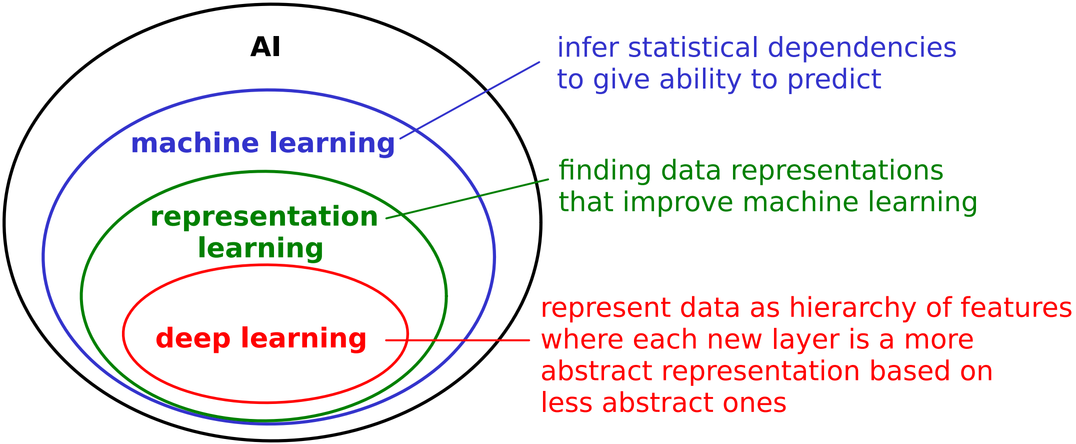
- Convolutional Networks
- Practice Session 2: Image Classification with Convolution Networks
- Image Recognition on Air Shower Footprints
- Practice Session 3: Air Showers Reconstruction

Machine Learning

"**Machine learning** is the subfield of computer science that gives computers the ability to learn without being explicitly programmed." *Arthur Samuel, 1959*

- **Supervised learning**: With data $\{x, y\}$ learn to predict $y(x)$
  - Regression task: $y \in \mathbb{R}$, e.g. predicting house prices
  - Classification task: y discrete, e.g. signal or background
- **Unsupervised learning**: Find structures or patterns in dataset $\{x\}$
  Example: Estimate distribution $p(x)$
- **Reinforcement learning**: Interaction with dynamic environment
  Example: Game playing and control problems

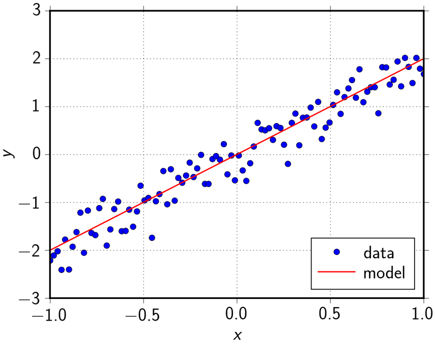
What is Deep Learning



Deep learning is the state-of-the-art approach for everything related to computer vision, speech recognition, natural language processing and many AI tasks in general.

Primer: Linear Regression

Let's define the terminology based on simple linear regression (supervised learning).



- **Data** set $\{x_i, y_i\} \quad i = 1 \dots N$
  and want to predict $y = f(x)$
- Define **model**
  $y_{\mathrm{m}}(x;\theta) = Wx + b$ with parameters $\theta = (W, b)$
- Define **objective** function (also called loss or cost) that quantifies distance between model and data
  $J(\theta|x,y) = \frac{1}{N}\sum_{i=1}^{N}(y_{\mathrm{m}}(x_i) - y_i)^2$
- **Train** the model by optimizing the parameters
  $\hat{\theta} = \arg\min J(\theta)$ e.g. through gradient descent

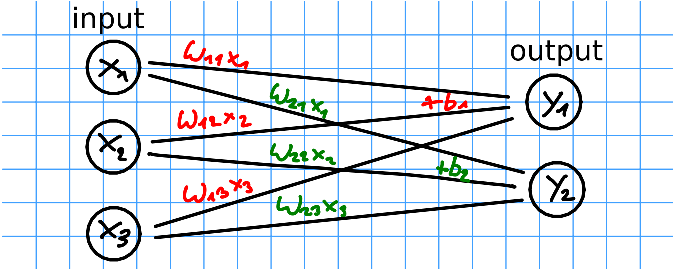
Neural Networks

Multidimensional Linear Model

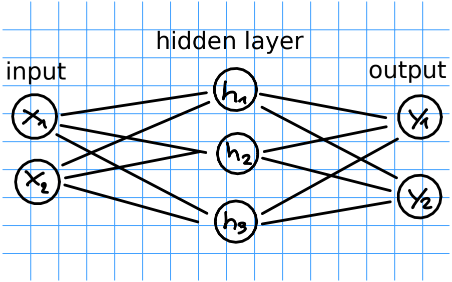
General case: Predict multiple outputs $\mathbf{y} = (y_1 \dots y_n)$ from multiple inputs $\mathbf{x} = (x_1 \dots x_m)$ using linear function $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$

Example: $x \in \mathbb{R}^3$, $y \in \mathbb{R}^2$

$$\begin{pmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$



Nonlinear Models



$\mathbf{Wx} + \mathbf{b}$ only describes linear transformation.
Idea: Create a network by applying several linear transformations

$$h' = W^{(1)}x + b^{(1)}$$
$$y = W^{(2)}h' + b^{(2)}$$

Problem: the model is still linear ...

$$y = W^{(2)}\left(W^{(1)}x + b^{(1)}\right) + b^{(2)}$$
$$= \underbrace{W^{(2)}W^{(1)}}_{W}x + \underbrace{W^{(2)}b^{(1)} + b^{(2)}}_{b}$$

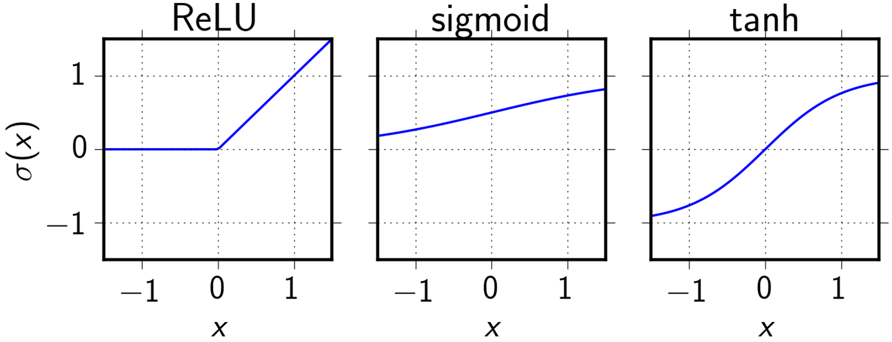
Solution: Apply non-linear **activation** σ to each element of $h' \longrightarrow h = \sigma(h') = \sigma(Wx + b)$

Activation Function

Through the activation function the layer $\sigma(Wx + b)$ becomes a non-linear mapping. Basically any kind of non-linearity can be used.

Some choices are

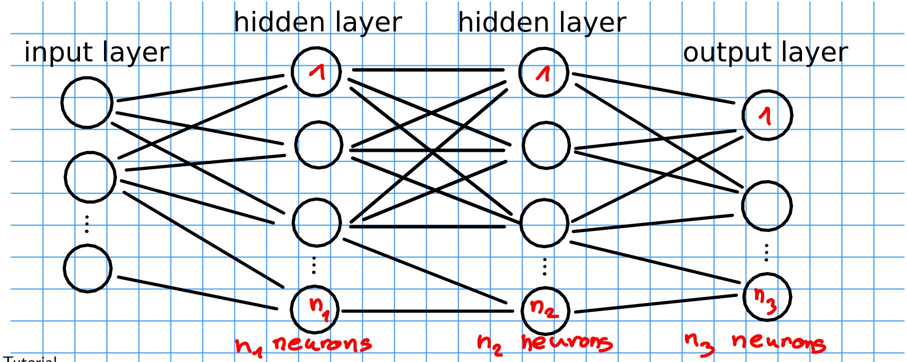
- $\sigma(x) = \max(0, x)$ rectified linear unit (**ReLU**)
- $\sigma(x) = \frac{1}{1+e^{-x}}$ sigmoid function
- $\sigma(x) = \tanh(x)$



Neural Networks

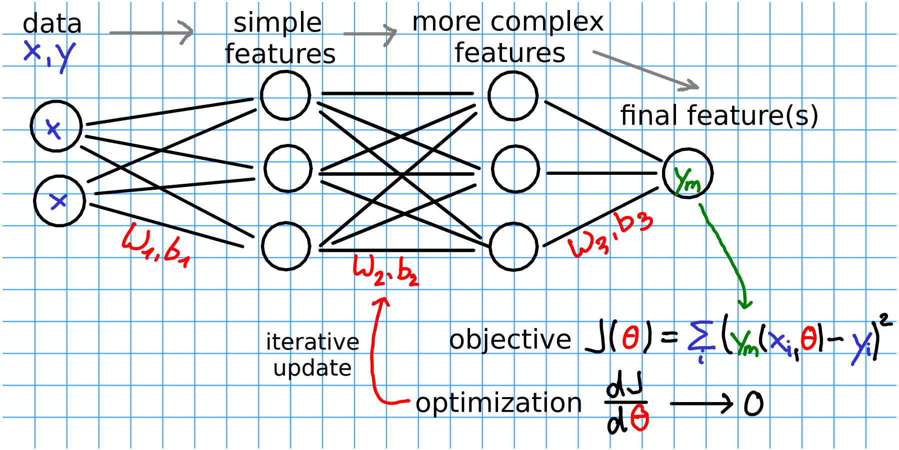
Basic unit $\sigma(Wx + b)$ called **neuron** in analogy with the brain

- Non-linear activation according to a number of inputs (= outputs of previous layer)
- Strength of connection between neurons specified by **W**
- **Depth**: number of weighted layers (don't count input here)
- **Width**: number of neurons per layer



Feature Learning

Each new layer extracts more complex features about the data. Learning these features is an **automatic** process when training the model.



$\longrightarrow$ No need for hand-crafted features, network learns meaningful features from data itself.

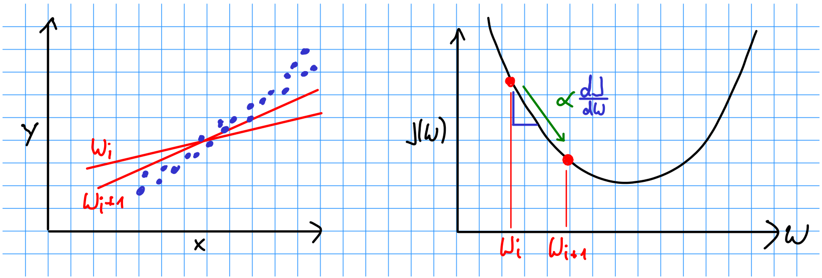
Optimization

Gradient Descent

Minimize objective function $J(\theta)$ by iteratively updating θ in negative direction of gradient. Update proportional to gradient $\frac{dJ}{d\theta}$ and learning rate α:

$$\theta \longrightarrow \theta - \alpha \frac{dJ}{d\theta}$$

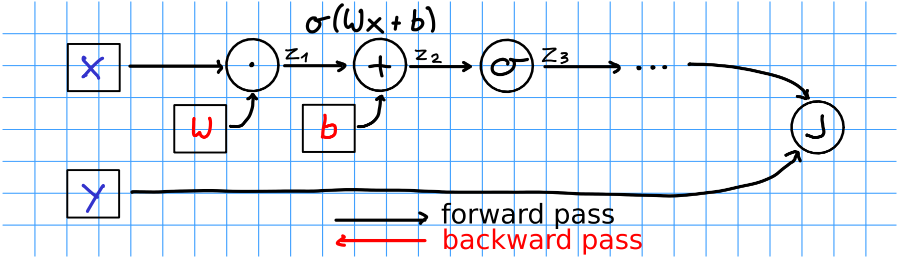
Example: Linear regression with mean squared error $J \sim \sum_{i=1}^{N} (y_{\mathrm{m}}(x_i) - y_i)^2$



Gradient can be evaluated numerically (expensive) or analytically (cheap).

Computation Graph

- A network is a series of simple operations $\rightarrow$ computation graph
- Each operation knows how to calculate
  - its local output (forward pass)
  - its local derivative (backward pass)
- Use chain rule to evaluate derivative $\frac{dJ(\theta|x,y)}{d\theta}$ for each parameter through multiplying the local derivatives
- Allows to evaluate the analytic gradient in automated way (**backprop**)



Stochastic Gradient Descent

For large datasets evaluating $J(\theta|x, y)$ and $\frac{dJ}{d\theta}$ is prohibitively expensive

Idea: Only use small subset of data in each iteration (**mini-batch**)

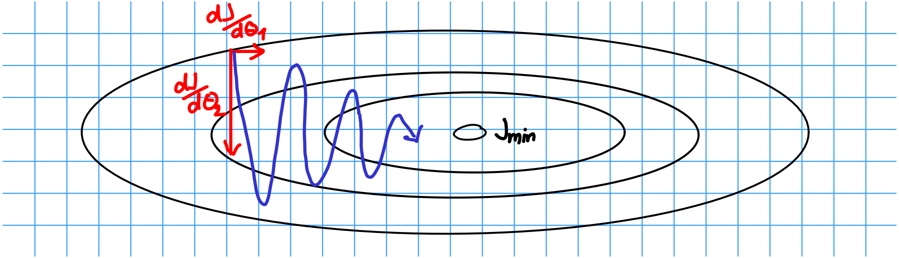
- Smaller batches $\rightarrow$ more parameter updates per compute
- Stochasticity in objective function helps escaping local minima
- **Batch size** is typically small, say 16 - 256 samples (hyperparameter!)
- **Epoch** $\equiv$ full pass through the training data

Stochastic gradient descent (**SGD**): Loop

- Sample random mini-batch of data
- Calculate objective J and gradient $\frac{dJ}{d\theta}$
- Update parameters $\theta \rightarrow \theta - \alpha\frac{dJ}{d\theta}$

Improved Optimizers

Typical situation: $\frac{dJ}{d\theta_1} \ll \frac{dJ}{d\theta_2}$ leading to slow learning in θ_1

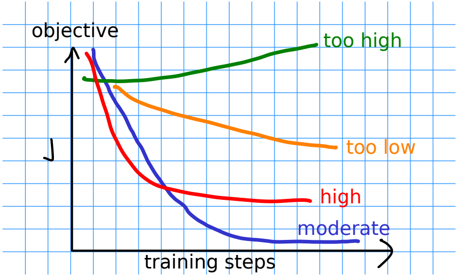


Improve learning with modified parameter update

- **Momentum**: build up momentum
- **Adagrad**: scale down by $1/\sqrt{\text{sum of past gradients}}$
- **RMSProp**: similar to Adagrad but with decay of scale factor
- **Adam**: combination of RMSProp and momentum

Learning Rate

In gradient-based optimizers ($\theta \rightarrow \theta - \alpha \frac{dJ}{d\theta}$) learning rate α determines speed of training



Best learning rate decreases over time (implicitly done in Adagrad, RMSProp and Adam)

- Typically start with $\alpha = 10^{-3}$ (for standard objective functions)
- Reduce learning rate $\alpha/10$ when objective stops decreasing

Parameter Initialization

Need to provide initial weights W and biases b at start of training.
Set weights to small random values to break symmetry $W \sim \mathcal{N}(\mu = 0, \sigma)$.
Scale of initial weights σ is critical for deeper nets.

- Weights too large $\rightarrow$ exploding signals and gradients
- Weights too small $\rightarrow$ vanishing signals and gradients

To keep the expected signal constant, scale initial weights with number of ingoing n_{in} and outgoing connections n_{out} of neuron.

- **Xavier**: $\sigma^2(W) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$ (X. Glorot & Y. Bengio)
- **He**: for ReLU use $\sigma^2(W) = \frac{2}{n_{\text{in}}}$ (He et. al)

Batch Normalization: feature normalization operation inside the network

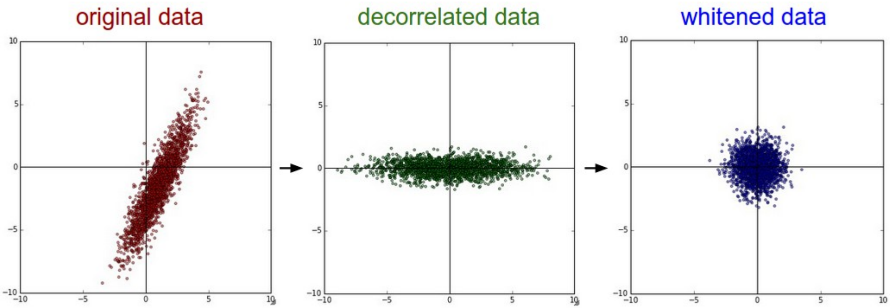
- Reduces problem of vanishing / exploding signals and gradients
- Reduces sensitivity to bad initialization

Data Preprocessing

Highly beneficial if input features $x_1, x_2 \ldots x_n$ of dataset are on same scale.

Preprocessing options:

- Limit input features to range $x_i \in [-1, 1]$
- Standard normalize: $\text{mean}(x_i) = 0$, $\text{std}(x_i) = 1$
- Whitening: Decorrelate and standard normalize



taken from http://cs231n.github.io/neural-networks-2/

Generalization

Universal Approximation Theorem

Many network design considerations: How many layers, how many nodes per layer, which kind of connections, ... ?

Universal approximation theorem

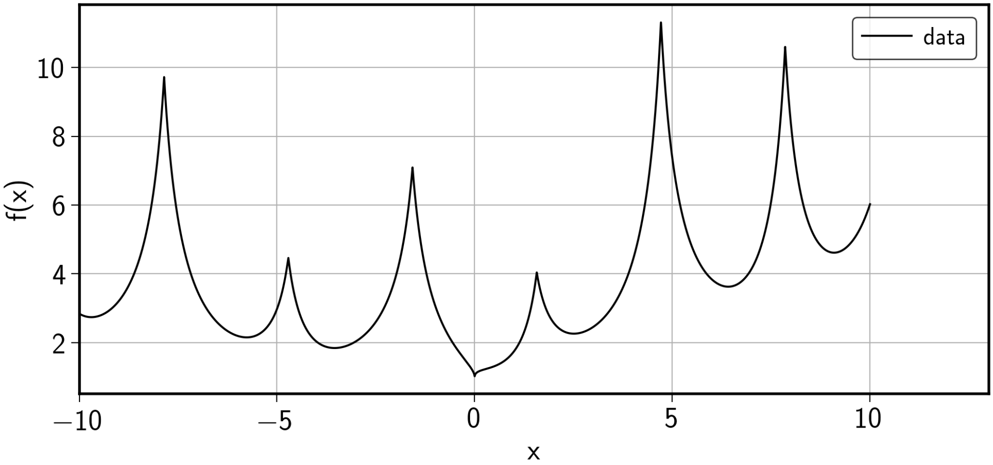
A feed-forward network with a linear output and at least one hidden layer with a finite number of nodes can approximate any function to arbitrary precision.

Caveats

- It doesn't specify how many nodes are needed
- There's no statement on how to train the network

1D Function Fitting

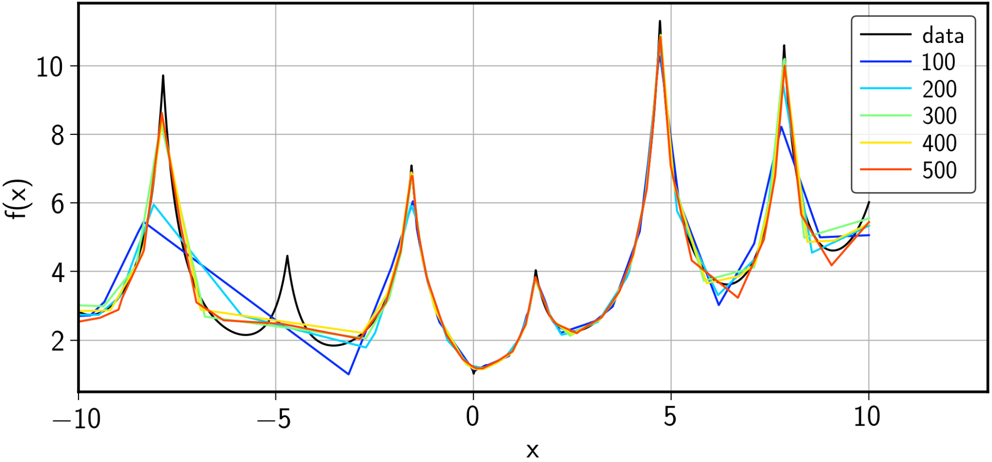
Try to fit some complicated function with neural network



1D Function Fitting 1

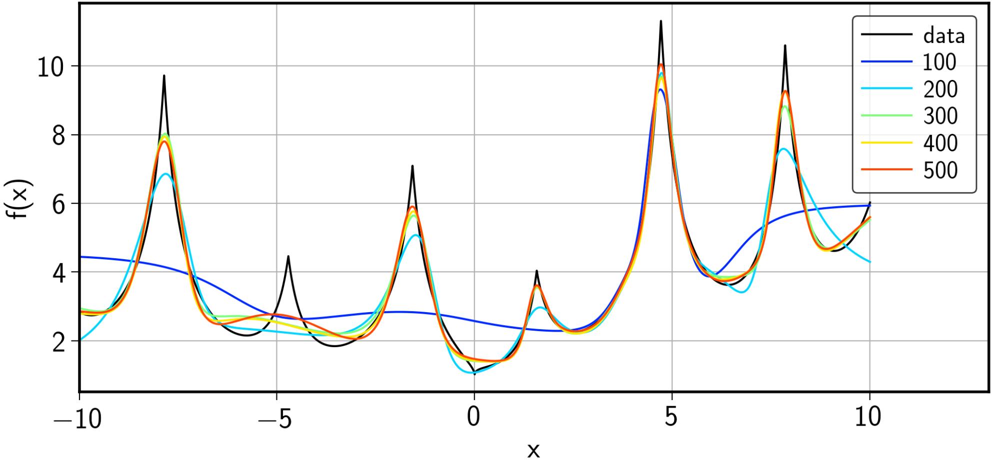
Train network with 3 hidden layers of 20 neurons and **ReLU** as activation

Fit after different number of training iterations:



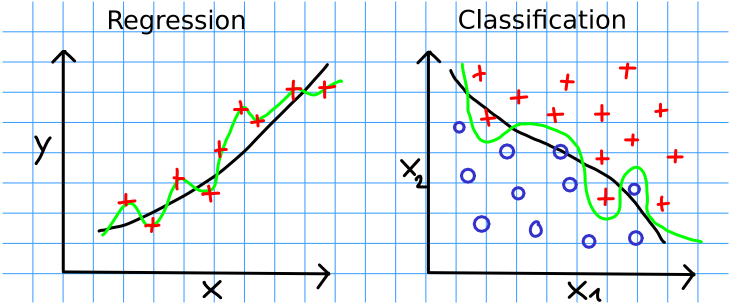
1D Function Fitting 2

Train network with 3 hidden layers of 20 neurons and **sigmoid** as activation
Fit after different number of training iterations:



Overfitting

If a complex enough network can learn any function, how do we know that it's not overfitting, i.e. fine-tuning to statistical fluctuations?



Validation

Unknown true distribution $p(x, y)$ from which data $\{x, y\}$ is drawn.
Trained model provides prediction $y_{\mathrm{m}}(x)$ based on this limited dataset.

Generalization

How good is our estimator when faced with new data?

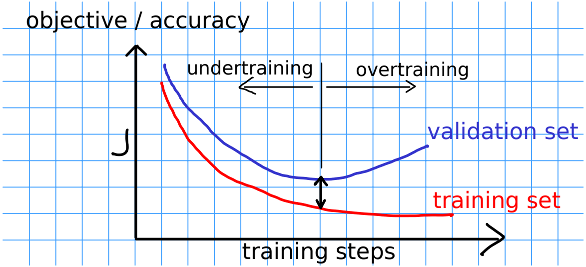
Validation

Since $p(x, y)$ unknown, estimate generalization error on fraction of data not used during training. $\longrightarrow$ Typically split data into **training**, **validation** and **test** set.

- Validation set: Evaluate performance for given set of hyper-parameters.
- Test set: Estimate final performance. Use only once, or risk of "look elsewhere effect"

Over-/Undertraining

During training, monitor the objective or another accuracy metric separately for the training and validation sets. Typical observation:



- Training loss decreases continuously ($\rightarrow$ 0 if model complex enough)
- Validation loss has minimum $\rightarrow$ network starts to **overtrain**
- Validation loss higher than training loss $\rightarrow$ **generalization gap**

$\longrightarrow$ **Early stopping**: Stop training when validation error stops to decrease

Regularization and Capacity

Regularization

Methods to reduce the validation error, possibly at the expense of higher training error.

- Early stopping
- Parameter norm penalties: add term to objective to penalize large weights
  - L^2 norm: $J + \lambda \sum W^2$
  - L^1 norm: $J + \lambda \sum |W|$
  - Need careful tuning of λ
- Data augmentation: generate artificial data samples
- Ensemble methods: combination of multiple networks (special case: Dropout)

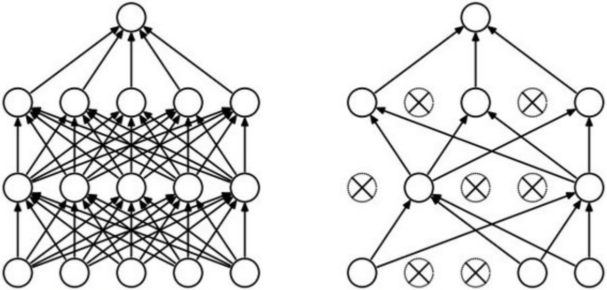
Model capacity

Complexity of functions that the network can represent. Effective capacity determined by

- Network architecture
- Training & regularization

Dropout

Randomly turn off a fraction (say $p = 0.5$) of neurons in each training step

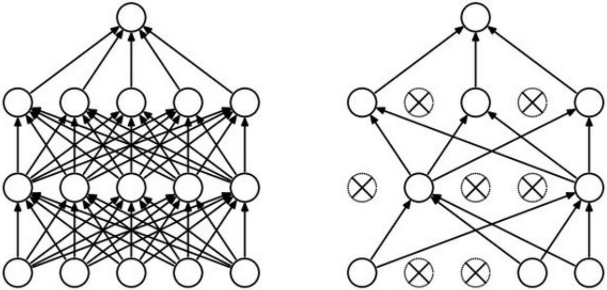


(a) Standard Neural Net

(b) After applying dropout.

Dropout

Randomly turn off a fraction (say $p = 0.5$) of neurons in each training step

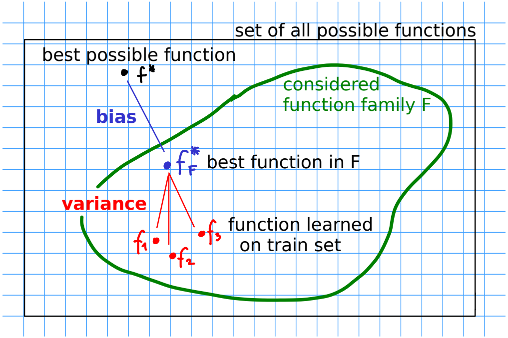


(a) Standard Neural Net (b) After applying dropout.

- Forces network to learn redundant representations, more robust predictions
- At test time use all neurons: large ensemble of models that share parameters
- Very effective regularization method. Easy to tune (hyperparameter p)

Bias and Variance Dilemma

- **Model Bias**: Error when approximating f^* with given model
- **Model Variance**: Error when training model using limited dataset



Effective capacity trades off bias against variance

- Choose more complex F: bias↓ variance↑
- Choose simpler F: bias↑ variance↓
- Optimal choice depends on size of dataset

Ensemble methods

- **Ensemble averaging**: Train multiple networks and combine their predictions
  $\Rightarrow$ variance↓, free boost in test performance

Summary

- Neural networks: layers of connected neurons $\sigma(Wx + b)$
  - Linear transformation $Wx + b$
  - Non-linear activation σ
  - Universal approximation theorem: can fit any function
- Optimization
  - Stochastic gradient descent on mini-batches of data
  - Optimizers: Adam, learn rate, data preprocessing, parameter initialization
- Generalization
  - Split data into training, validation and test set
    - Validation set for monitoring training and tuning hyperparameters
    - Estimate generalization error with test set
  - Regularization to control overfitting
    - early stopping, dropout, data augmentation
  - Bias / variance, ensemble averaging

Image Classification

Image Classification

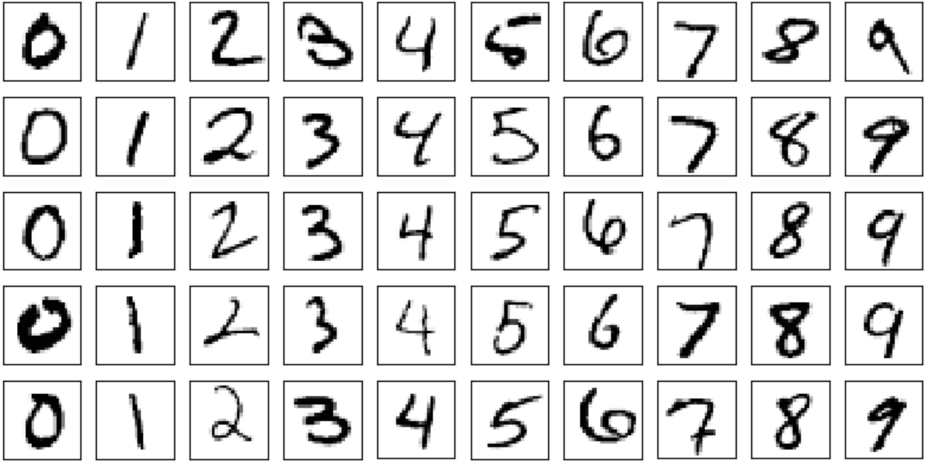
What do these images show?

Very easy for humans, very hard for classical machine learning methods. Challenges are

- High dimensional input $\mathbf{x} \in \mathbb{R}^{10^3 - 10^7}$
- Many possible classes depending on task
- Need to deal with multiple variations
  - → viewing angle, occlusion, deformation, light conditions, object variations ...

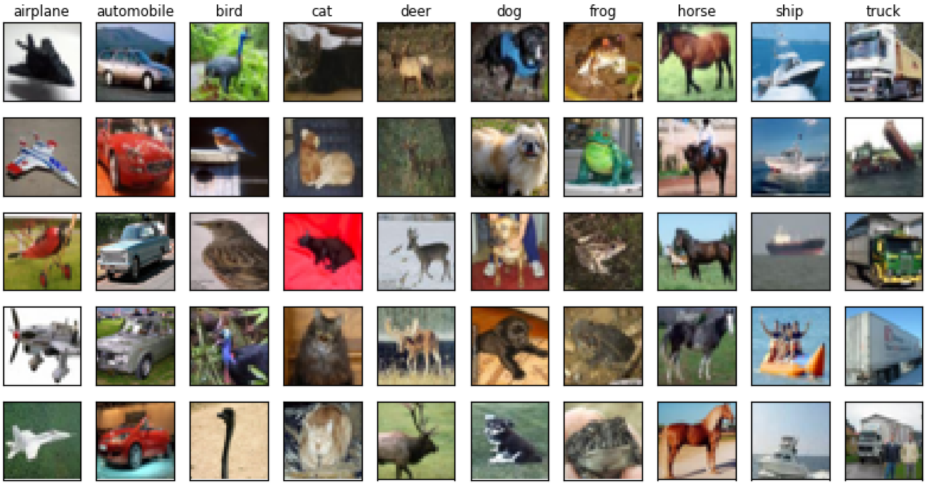
MNIST

Hand-written digit dataset, 28 x 28 pixels (grayscale)



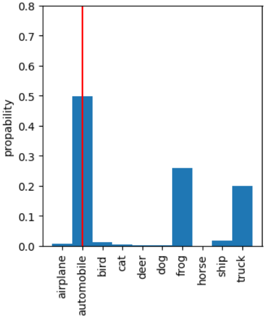
CIFAR-10

Tiny natural image dataset, 32 x 32 pixels (8-bit RGB), 10 categories



CIFAR-10 Classification Task

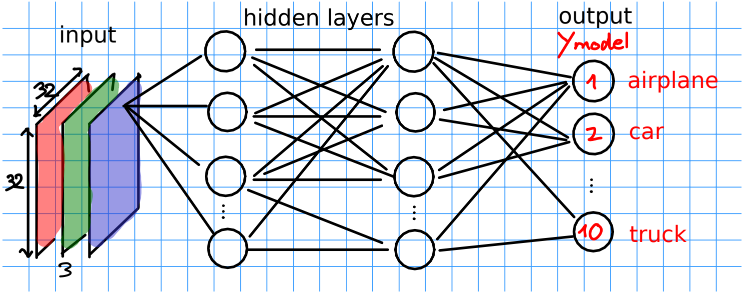
- Input $\mathbf{x} = (x_1, \ldots x_{3072})$ for $32 \times 32 \times 3 = 3072$ features
- Output $\mathbf{y} = (y_1, \ldots y_{10})$, one variable for each category (**one-hot encoding**)
  Categories: airplane, car, bird, cat, deer, dog, frog, horse, ship, truck
  Image $\mathbf{x}^i$ showing a car has $\mathbf{y}^i = (0, 1, 0 \ldots 0)$.



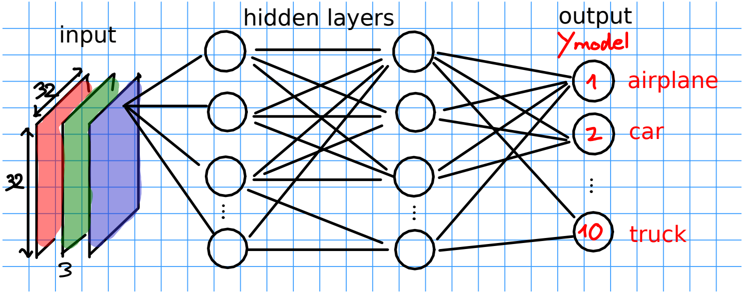
Model predicts probability for each class
$\rightarrow \mathbf{y}_{\text{model}}(\mathbf{x}^i|\theta) = (p_{\text{airplane}}, p_{\text{car}} \ldots)$

- Take highest p_j as predicted category
- Value of $\max(p_j)$ gives measure of certainty

Classification Layer



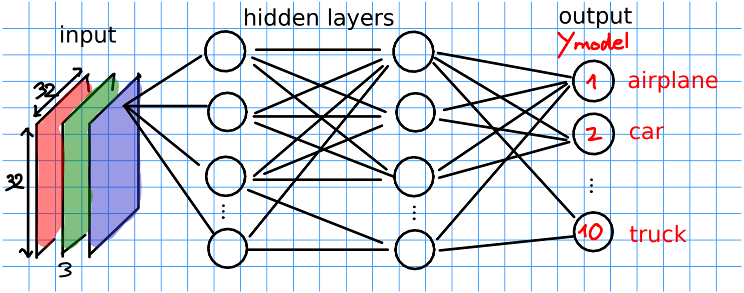
Classification Layer



Softmax as activation, $y_{\text{model},j} = \sigma(z_j) = e^{z_j} / \sum e^{z_j}$, thus

- Pre-activation outputs are unnormalized log-probabilities $z_j \sim \log(p_j)$
- Softmax takes out the log and normalizes $\sum_j p_j = 1$

Classification Layer



Softmax as activation, $y_{\text{model},j} = \sigma(z_j) = e^{z_j} / \sum e^{z_j}$, thus

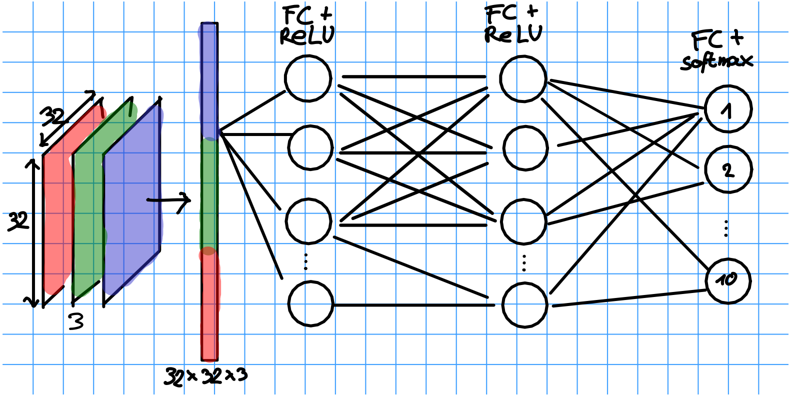
- Pre-activation outputs are unnormalized log-probabilities $z_j \sim \log(p_j)$
- Softmax takes out the log and normalizes $\sum_j p_j = 1$

Categorial cross-entropy as objective function $J(\theta) = -\frac{1}{n} \sum_i \sum_j \mathbf{y}^i \log\left(\mathbf{y}_{\text{model}}(\mathbf{x}^i | \theta)\right)$

- Since $y_j = 0$ for all but the true class, only the predicted probability for the correct classification contributes
- Corresponds to maximum likelihood

First Ansatz: Fully Connected Network

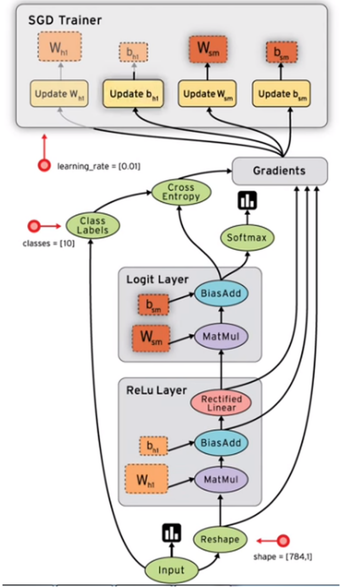
- Input layer: Flatten image to $32 \times 32 \times 3 = 3072$ vector
- Some hidden layers: Fully connected with ReLU, dropout for regularization
- Classification layer: Fully connected with softmax



TensorFlow & Keras

What is TensorFlow

- Open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while edges represent tensors communicated between them.
- Deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.
- Originally developed at Google Brain, but general enough for application in variety of domains.
- Most popular deep learning framework
- Stable API since last week (version 1.0)



TensorFlow Example

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 3072])
y = tf.placeholder(tf.float32, [None, 10])

# hidden layer: h1 = ReLU(W1*x + b1)
W1 = tf.Variable(tf.random_normal([256, 3072]))
b1 = tf.Variable(tf.random_normal([256]))
h1 = tf.nn.relu(tf.matmul(x, W1) + b1)

# output layer: y = softmax(W2*h1 + b2)
W2 = tf.Variable(tf.random_normal([256, 10]))
b2 = tf.Variable(tf.random_normal([10]))
ym = tf.nn.softmax(tf.matmul(h1, W2) + b2)
```

TensorFlow is **low-level** framework: very verbose, focus on maximum flexibility

Keras Example

Keras as high-level library on top of TensorFlow.

- More readable
- Default settings follow best practices
- Quicker prototyping

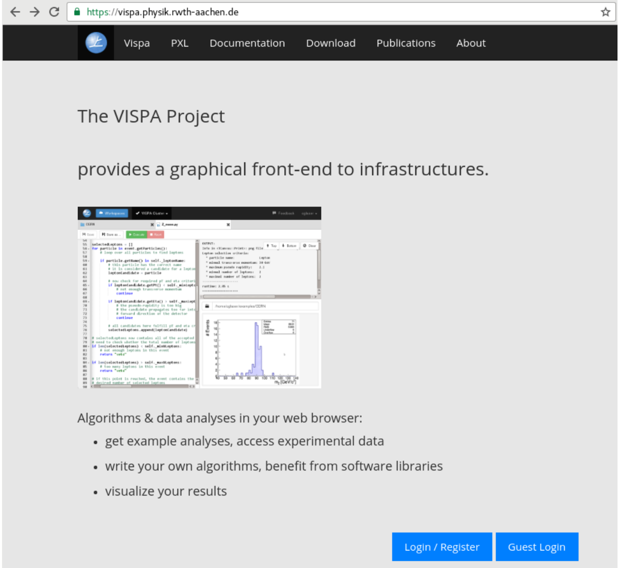
```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential([
  Dense(256, activation='relu', input_shape=(3072)),
  Dense(10, activation='softmax')
  ])
```

“Keras is gaining official Google support . . . If you want a high-level object-oriented TF API to use for the long term, Keras is the way to go.” *F. Chollet, a few days ago*

Practice Session 1: Image Classification with Neural Networks

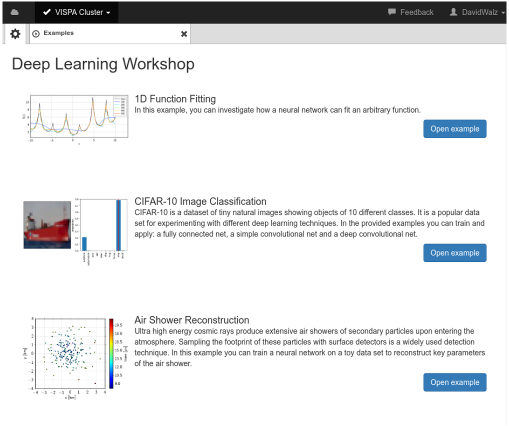
VISPA

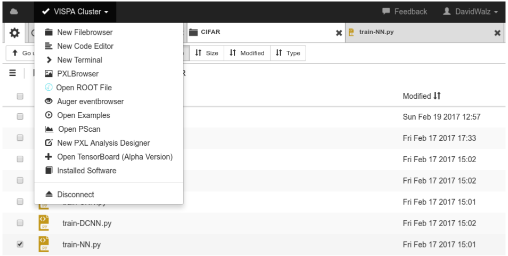
- Open `vispa.rwth-aachen.de`
- Register a new or login with your existing account
- *Note:* Guest accounts work as well, but data is lost when logging out



Deep Learning Examples

Open the CIFAR-10 example





Note: File browser, code editor and terminal tabs can be opened under "VISPA Cluster"

Code Example: `train-NN.py`

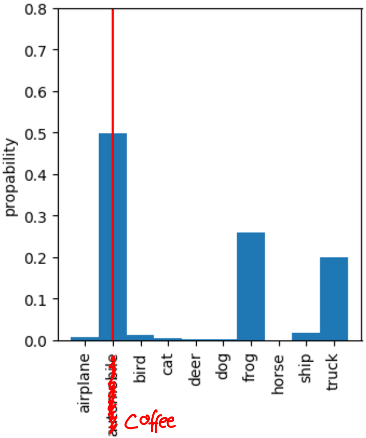
```
# define 3 layer network
model = Sequential([
    Flatten(input_shape=(32, 32, 3)),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(256, activatio n='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax') ])

# set objective and optimizer, set up computation graph
model.compile(loss='categorical_crossentropy',
    optimizer=keras.optimizers.Adam(lr=1E-3),
    metrics=['accuracy'])

# training
results = model.fit(X_train, Y_train,
  batch_size=32, nb_epoch=25,
  validation_data=(X_valid, Y_valid))
```

Practice Time: `train-NN.py`

- `pygpu %file` executes your script on the VISPA GPU cluster
  - ➔ 20 $\times$ GeForce GTX 1080
  - ➔ 2 jobs per GPU $\rightarrow$ 40 jobs in parallel
  - ➔ Type `condor_q` in terminal to query job status
- Familiarize yourself with data preprocessing etc.
- Run and inspect the training results
  - ➔ `condor/` $\rightarrow$ log, error and standard output files
  - ➔ `train-NN-XYZ/` $\rightarrow$ plots, training history, trained model
- Experiment with your model
  - ➔ Modify the network layout: Add more layers, neurons
  - ➔ Tune hyperparameters: dropout, learning rate, batch size

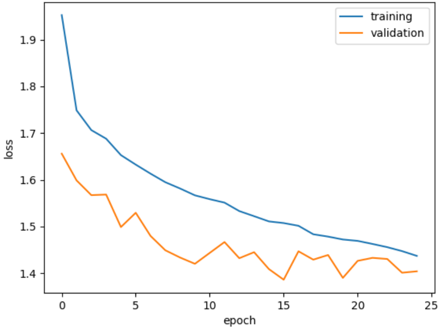
Note: We'll get to `train-CNN.py` and `train-DCNN.py` after the coffee break!

Coffee Break

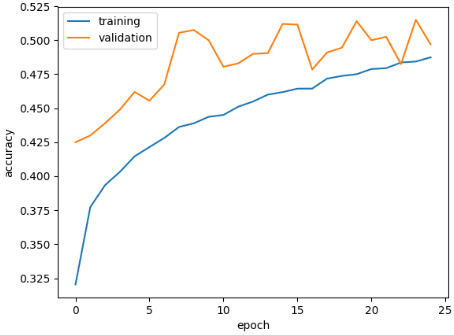


Results: CIFAR-10 with Dense Neural Network

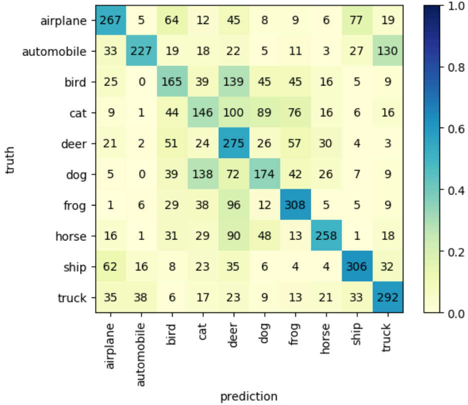
Objective

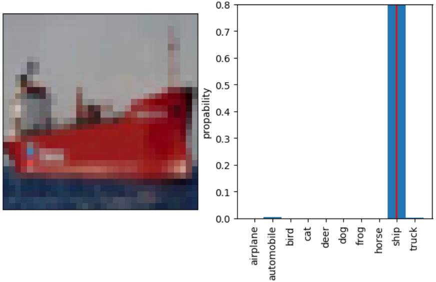


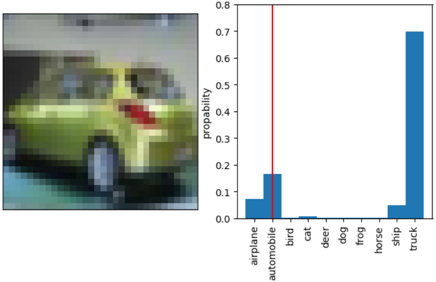
Accuracy



Results: CIFAR-10 with Dense Neural Network







Results: CIFAR-10 with Dense Neural Network

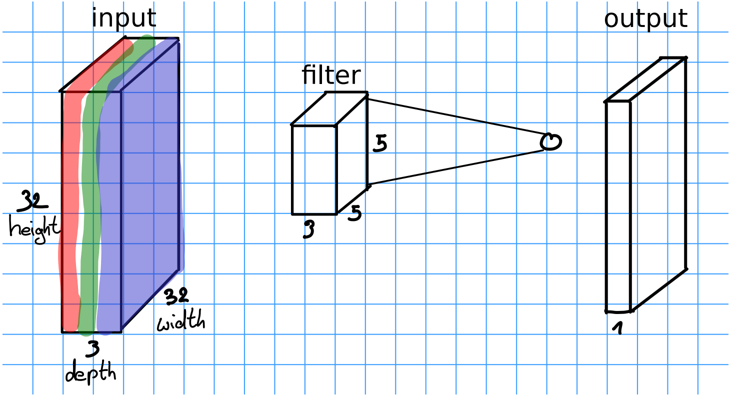
```
Layer                          Output Shape          Param
==========================================================
Flatten                        (None, 3072)          0
Dense                          (None, 256)           786688
Activation                     (None, 256)           0
Dropout                        (None, 256)           0
Dense                          (None, 256)           65792
Activation                     (None, 256)           0
Dropout                        (None, 256)           0
Dense                          (None, 10)            2570
Activation                     (None, 10)            0
==========================================================
Trainable params: 855,050
```

High dimensional input $\longrightarrow$ many parameters in fully connected layers

Convolutional Networks

2D Convolution Operation

- Consider input **volume**, e.g. color image
- Use convolution filter with much smaller width and height, but same depth as input
- Slide **filter** w spatially over input **volume** and calculate $w^T x + b$ to get one output value at each position



2D Convolution: Filters

Each filter scans the input for the presence of one feature

edge

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

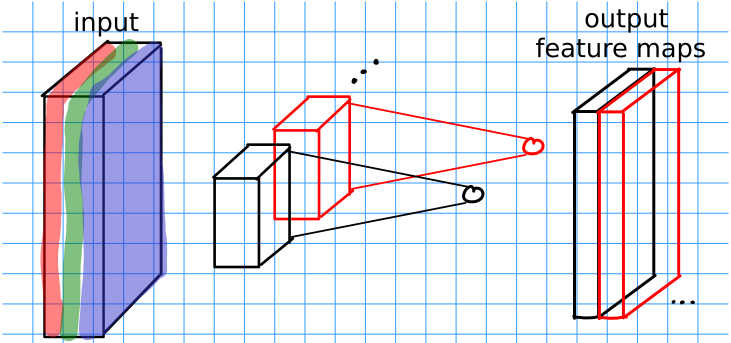
horizontal edge

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}$$

diagonal edge

$$\begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

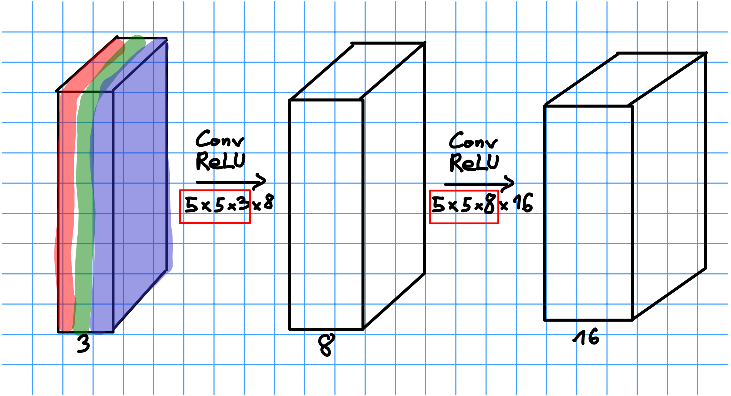
Use multiple filters and stack the output feature maps depth-wise



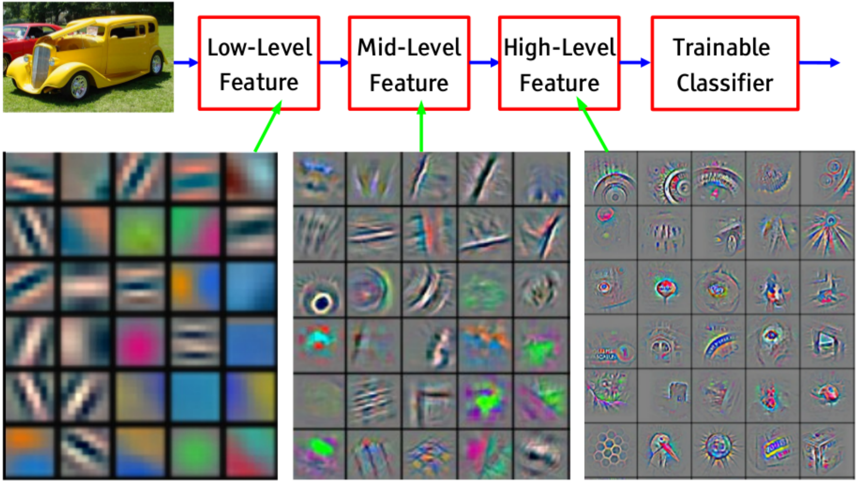
Convolution Stack

Stack multiple layers of convolution + activation

- Each convolution acts on the feature map of the previous layer
- Receptive field increases
- Complexity of features increases



Hierarchical Feature Extraction



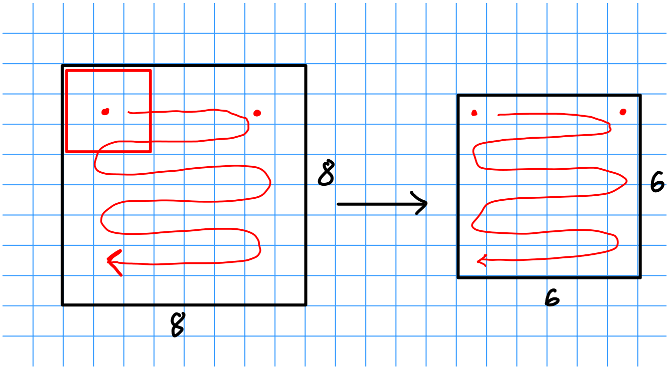
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

slide from Yann Le Cun

Spatial Output Size

Standard convolution reduces the output size.
$\longrightarrow$ Sets an upper bound to the number of convolution layers.

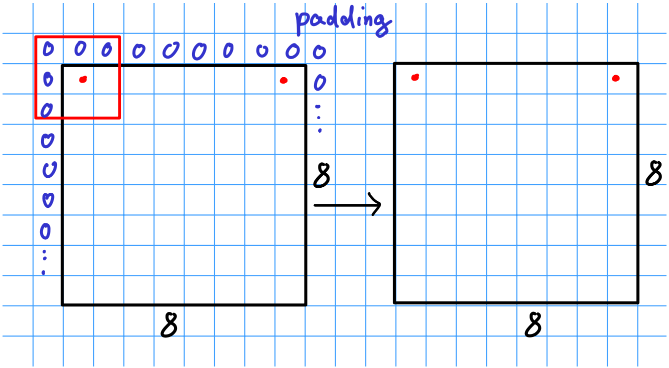
Example: Convolution with 3x3 filter



Padding

Padding the input with zeros prevents the spatial extent from shrinking too fast
$\longrightarrow$ Necessary for deep stacks of many convolution layers

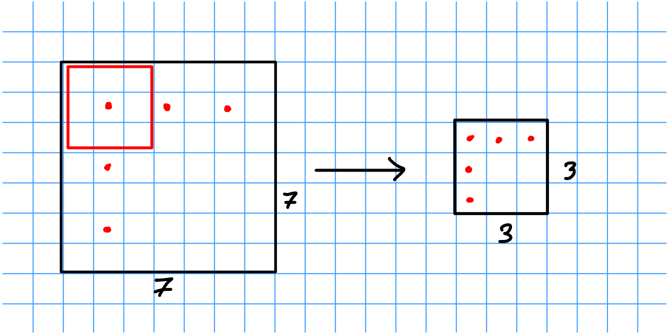
Example: Convolution with 3x3 filter and padding of one



Striding

Using a larger **stride** when sliding over the input reduces the output size

Example: Convolution with 3×3 filter and stride of 2

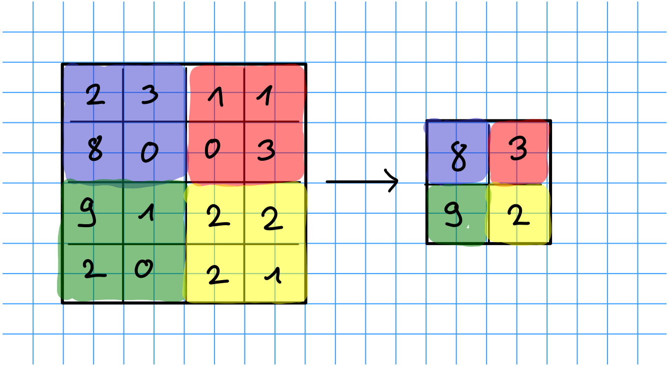


Pooling

Sub-sample the input to reduce the output size

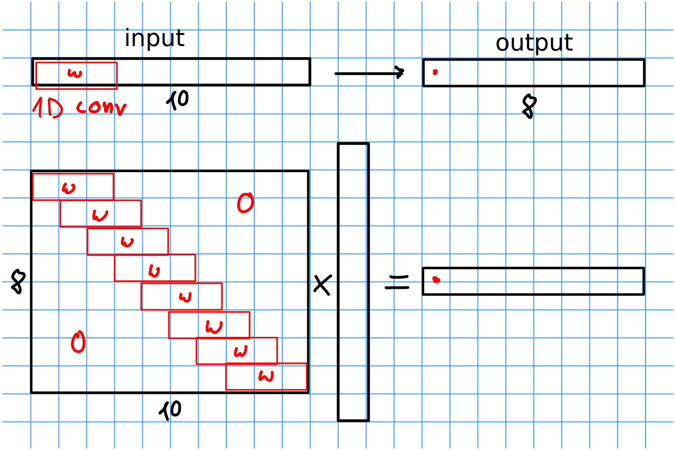
- **AveragePooling**: Take the mean of each patch
- **MaxPooling**: Take the maximum of each patch (better)
- Pooling is more precise than striding but also a little more expensive

Example: MaxPooling on 2×2 patches



Convolution vs Dense Layers

Convolutions are a special case of dense (fully connected) weight layers.

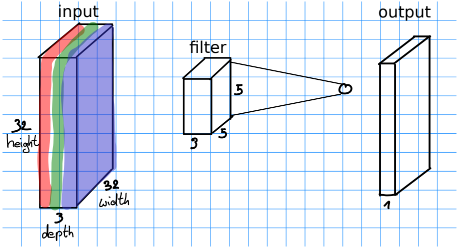


Number of parameters greatly reduced due to **sparsity** and **weight sharing**.

Convolution: Strong prior for **local correlation** and **translational invariance**.

Summary: 2D Convolution

- Acts on a 3D input volume: $W \times H \times D$ (width, height, depth)
- Slides small filter over input volume and compute dot product and bias add
- Hyperparameters:
  - → Size of filters F, typically 3 or 5
  - → Number of filters K
  - → Zero padding to maintain spatial extent
  - → Stride or MaxPooling to reduce spatial extent
- Small number of parameters: $F^2 \cdot D \cdot K$ (weights) and K (biases)



Convolutions in Keras

```
# Input example: CIFAR-10
X = ...  # X.shape = (60000,32,32,3), feature axis last


# Convolution layer example
keras.layers.Convolution2D(
  16, 3, 3,               # applies 16 (3x3) filters
  subsample=(1, 1),       # no striding, use (2,2) for stride 2
  border_mode='valid',    # no padding, use 'same' for padding
  activation=None)


# MaxPooling layer example
keras.layers.MaxPooling2D(
  pool_size=(2, 2),       # (2x2) pooling
  strides=None,           # no striding
  border_mode='valid')    # no padding, use 'same' for padding
```

Practice Session 2: Image Classification with ConvNets

CIFAR-10 with Convolutional Networks

Inspect, run and evaluate

- `train-CNN.py` simple convolutional net
- `train-DCNN.py` deep convolutional net, $\sim$ 10-20 min runtime

Extras

- Visualize trained convolutional filters and activation in first layer
- `test-ensemble.py` form ensemble average over multiple models

Results: CIFAR-10 with Simple Convolutional Network

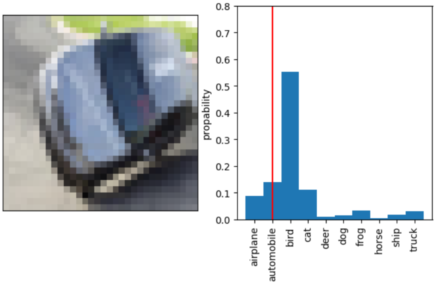
```
Layer                                   Output Shape         Param
==================================================================
Convolution2D                          (None, 14, 14, 32)   2432
Activation                             (None, 14, 14, 32)   0
Dropout                                (None, 14, 14, 32)   0
Convolution2D                          (None, 5, 5, 64)     51264
Activation                             (None, 5, 5, 64)     0
Dropout                                (None, 5, 5, 64)     0
Convolution2D                          (None, 1, 1, 128)    204928
Activation                             (None, 1, 1, 128)    0
Dropout                                (None, 1, 1, 128)    0
Flatten                                (None, 128)          0
Dense                                  (None, 128)          16512
Activation                             (None, 128)          0
Dropout                                (None, 128)          0
Dense                                  (None, 10)           1290
Activation                             (None, 10)           0
==================================================================
Trainable params: 276,426
Test accuracy: ~ 75%
```

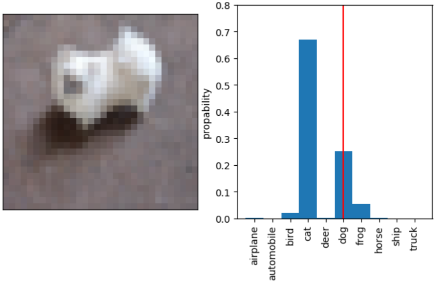
Results: CIFAR-10 with Deep Convolutional Network

```
Layer                          Output Shape          Param
==================================================================
Convolution2D                  (None, 32, 32, 32)    896
BatchNormalization             (None, 32, 32, 32)    128
... 8 more convolution blocks ...
Convolution2D                  (None, 4, 4, 256)     590080
BatchNormalization             (None, 4, 4, 256)     1024
Activation                     (None, 4, 4, 256)     0
MaxPooling2D                   (None, 2, 2, 256)     0
Flatten                        (None, 1024)          0
Dropout                        (None, 1024)          0
Dense                          (None, 256)           262400
BatchNormalization             (None, 256)           1024
Activation                     (None, 256)           0
Dropout                        (None, 256)           0
Dense                          (None, 10)            2570
Activation                     (None, 10)            0
==================================================================
Trainable params: 2,178,090
Test accuracy: ~ 88%
```

Results: CIFAR-10 with Deep Convolutional Network

truth \ prediction	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	448	3	17	4	6	0	9	3	15	7
automobile	5	454	4	2	0	1	7	1	10	11
bird	20	0	386	5	19	9	42	6	1	0
cat	8	0	31	343	23	34	57	3	4	0
deer	4	0	27	14	419	3	19	7	0	0
dog	2	0	18	58	17	391	12	14	0	0
frog	1	0	7	6	3	3	488	0	1	0
horse	1	0	10	13	22	1	4	454	0	0
ship	13	6	6	5	1	1	3	0	459	2
truck	7	24	5	5	1	2	13	4	9	417



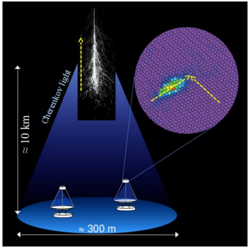


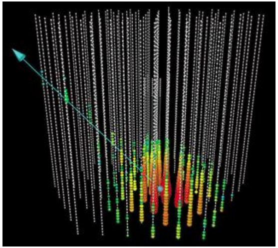
Human level accuracy $\sim 95\%$, need data augmentation to reach this level

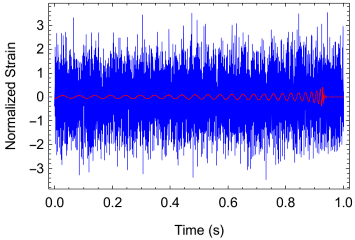
Image Recognition on Air Shower Footprints

Deep Learning Applications in Astroparticle Physics

Applications in astroparticle experiments: Cherenkov telescopes, gamma & cosmic ray satellites, neutrino & cosmic ray observatories, dark matter & gravitational wave detectors

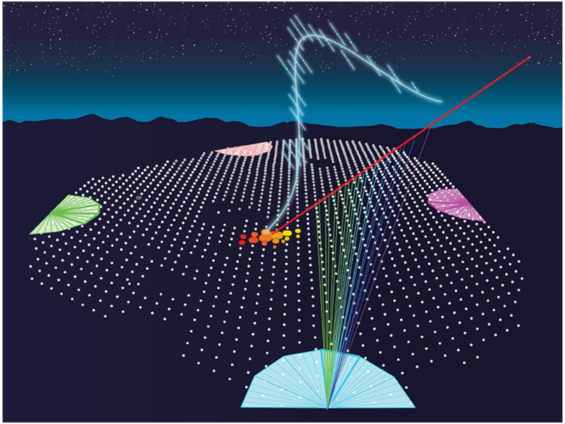






- Event reconstruction
- Signal classification & background rejection
- Pattern recognition on signal distributions, e.g. gamma or cosmic ray skies

Air Shower Detection at the Pierre Auger Observatory



https://physics.aps.org/articles/v9/125

- **Fluorescence detector**
  - → Fluorescence light traces longitudinal shower development
  - → 2D image
- **Surface detector**
  - → Water Cherenkov tanks detect passage of charged particles
  - → 2D image sequence
- **Radio detector**
  - → Radio footprint with time information
  - → Pulse measured in 2 or 3 polarizations
  - → 2D image sequence / time traces

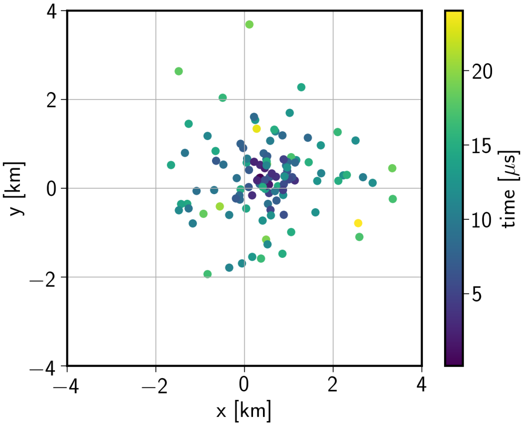
Air Shower Footprint – Toy Data Generator

Simulation of spatial and time distribution of air shower muons on ground

Procedure

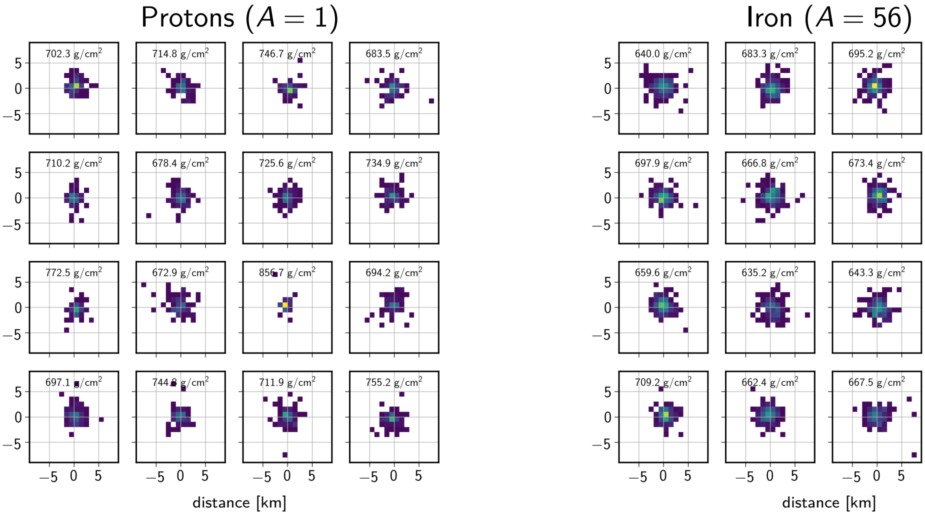
- Random number muon $\propto \frac{E}{E_0} A^{0.15}$
- Random production depth $\propto \mathcal{N}(\mu, \sigma(X_{\max}, A))$
- Random direction $\propto$ Normal distribution with opening angle
- Calculate arrival on ground (x, y, t)
- Add random offset to (x, y, t) per event

Shower footprint



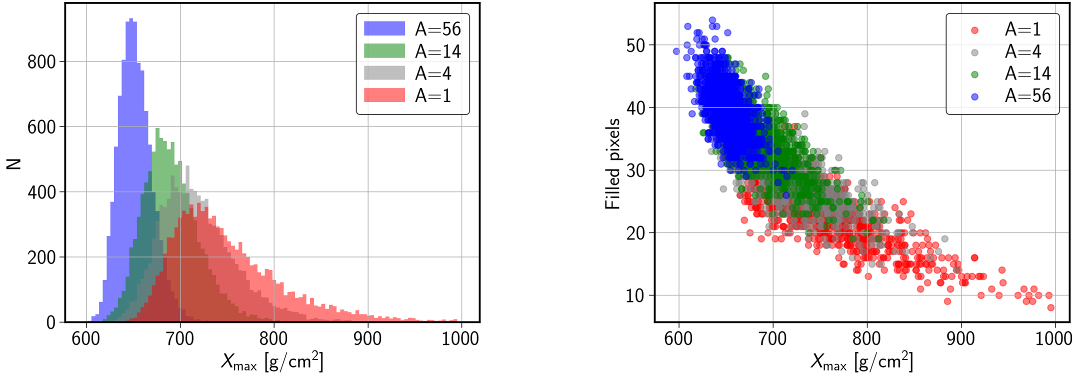
Toy Data

40.000 vertical showers, same energy, 25% protons, helium, nitrogen, iron



Toy Data

40.000 vertical showers, same energy, 25% protons, helium, nitrogen, iron



$\longrightarrow$ No clear separation possible on either X_{max} or number of filled pixels

Practice Session 3: Air Showers Reconstruction

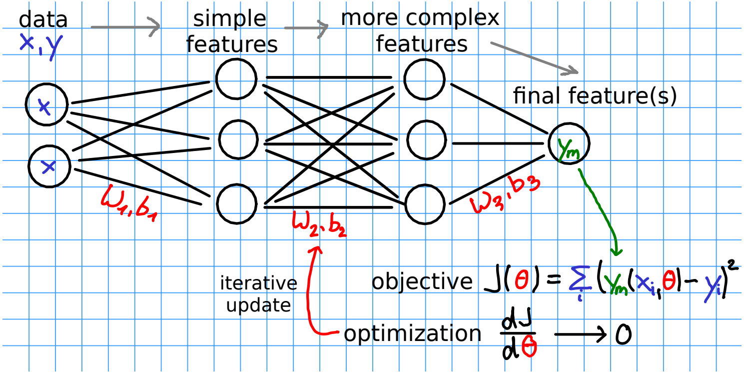
Toy Air Shower Reconstruction

- Toy data generator: `generate_data.py`
- Regression Task: `train-Xmax.py` – reconstruct X_{max}
- Classification Task: `train-A.py` – separate the four different species $A = 1, 4, 14, 56$

Links & Resources

- TensorFlow Tutorial
  `https://www.tensorflow.org/get_started/get_started`
- Deep Learning (Goodfellow, Bengio and Courville)
  MIT Press, ISBN: 0262035618
  `http://www.deeplearningbook.org/`
- Neural Networks and Deep Learning (Nielson)
  `http://neuralnetworksanddeeplearning.com/`
- CS231n - Convolutional Neural Networks for Visual Recognition (Kaparthy)
  `http://cs231n.stanford.edu/syllabus.html`
- Deep Learning by Google (Vanhoucke), Udacity
  `https://www.udacity.com/course/deep-learning--ud730`

The Physicist's Toolbox



- With deep learning we have a powerful new tool at our disposal!
- Availability of deep learning frameworks and GPUs make this previously very challenging method accessible to anyone with scripting abilities and a gaming PC