Computational Intelligence Laboratory

Lecture 7
Neural Networks

Thomas Hofmann

ETH Zurich – cil.inf.ethz.ch

03 April 2020

Section 1

Multilayer Perceptrons

Neural Networks

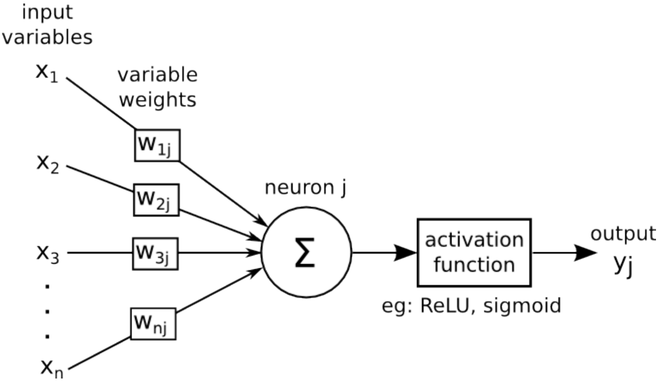
- Neural network: consist of simple, parametrized computational elements = neurons or units

- Basic operation:
  - each unit implements a generalized linear function: $\mathbb{R}^n \to \mathbb{R}$
  - linear + non-linear activation function $\sigma : \mathbb{R} \to \mathbb{R}$
  - parametrized with weights $\mathbf{w} \in \mathbb{R}^{n+1}$

$$f^\sigma(\mathbf{x}; \mathbf{w}) := \sigma\left(w_0 + \sum_{i=1}^{n} w_i x_i\right) \overset{(*)}{=} \sigma(\mathbf{w}^\top \mathbf{x})$$

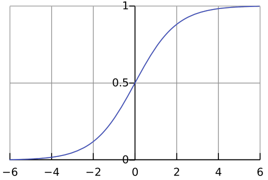
    - (*) will ignore/absorb bias parameter w_0 for clarity

Neuron: Schematic View

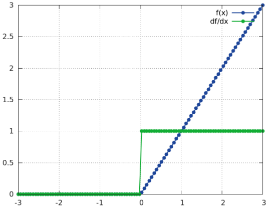


Activation Functions

- Old school: logistic (or $\tanh$) function

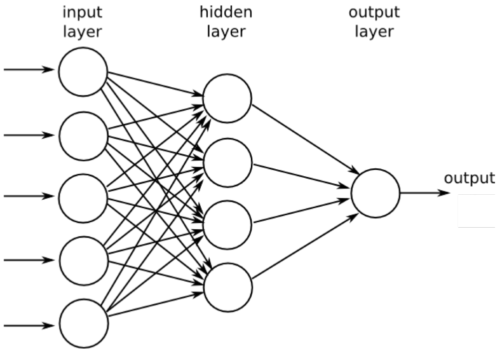


- New school: ReLU (rectified linear unit)
  - linear function over half-space $\mathcal{H} = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} > 0\}$
  - zero on complement $\mathcal{H}^c = \mathbb{R}^n - \mathcal{H}$
  - non-smooth, but simple derivative over $\mathbb{R} - \{\mathbf{0}\}$



Multilayer Perceptron

- Arrange such neurons in a layer (here: hidden layer)
- Input layer = raw input $\mathbf{x}$, no computation
- Output layer = final output, class label, response variables



Units and Layers

- Units are arranged in layers
  - units indexed by j
  - mapping between layers: vector-valued
  - shared choice of σ

$$F^\sigma : \mathbb{R}^n \to \mathbb{R}^m, \quad F^\sigma_j(\mathbf{x}) = \underbrace{\sigma(\mathbf{w}_j^\top \mathbf{x})}_{\substack{\text{transfer fct.} \\ \text{of } j\text{-th unit}}}, \quad j = 1, \ldots, m$$

- Matrix-vector notation (σ applied elementwise)

$$F^\sigma(\mathbf{x}; \mathbf{W}) = \sigma\left(\mathbf{W}\,\mathbf{x}\right), \quad \mathbf{W} = \begin{pmatrix} \mathbf{w}_1^\top \\ \ldots \\ \mathbf{w}_m^\top \end{pmatrix}$$

Units and Layers

- Sometimes we want to index layers by l
- Activation vector of l-th layer: $\mathbf{x}^{(l)}$
  - $\mathbf{x}^{(1)}$ is input; $\mathbf{x}^{(L)}$ is output; $\mathbf{x}^{(l)}$ $(1 < l < L)$ hidden layers
  - indexed notation for layer-to-layer forward propagation

$$\mathbf{x}^{(l)} = \sigma^{(l)} \left(\mathbf{W}^{(l)} \mathbf{x}^{(l-1)} \right)$$

Units and Layers

- L-layer network: nested function

$$\mathbf{y} = \sigma^{(L)}\left(\mathbf{W}^{(L)}\sigma^{(L-1)}\left(\cdots\left(\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}\right)\cdots\right)\right)\right)$$

- Layer width = "more of the same" features
- Network depth = "more compositionality", feature hierarchy (= deep learning)

Output Layer

- Shortcuts $\mathbf{W} = \mathbf{W}^{(L)}$, $\mathbf{x} = \mathbf{x}^{(L-1)}$
- Linear regression: linear activation

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

- Binary classification (one output): logistic

$$y_1 = P(Y = 1 \mid \mathbf{x}) = \frac{1}{1 + \exp\left[-\mathbf{w}^\top \mathbf{x}\right]}$$

- Multiclass with K classes: soft-max

$$y_k = P(Y = k \mid \mathbf{x}) = \frac{\exp\left[\mathbf{w}_k^\top \mathbf{x}\right]}{\sum_{j=1}^{K} \exp\left[\mathbf{w}_j^\top \mathbf{x}\right]}$$

MLP Classification vs. Logistic Regression

- Logistic regression: computes linear function of inputs

$$P(Y = 1|\mathbf{x}) = \frac{1}{1 + \exp\left[-\langle \mathbf{w}, \mathbf{x} \rangle\right]}$$

- Multilayer Perceptron
  - learn intermediate feature representation
  - perform logistic regression on learned representation $\mathbf{x}^{(L-1)}$

Learning in Massively Parametrized Models :)

- Learning = automatically fiddling with network weights

Loss Function

- How do we adjust, i.e. learn the weights?
- First: define a loss function
  - target output y^*, prediction y
  - loss function $\ell(y^*; y)$
- Squared loss, $y^*, y \in \mathbb{R}$

$$\ell(y^*; y) = \frac{1}{2}(y^* - y)^2$$

- Cross-entropy loss, $0 \le y \le 1$ (Bernoulli), $y^* \in \{0, 1\}$ or $\in [0; 1]$

$$\ell(y^*; y) = -y^* \log y - (1 - y^*) \log(1 - y)$$

Regularized Risk Minimization

- Training set of examples $\mathcal{X} = \{(\mathbf{x}_t, y_t) : t = 1, \ldots, T\}$

- Empirical risk

$$\mathcal{L}(\theta; \mathcal{X}) = \frac{1}{T} \sum_{t=1}^{T} \ell(y_t; \underbrace{y(\mathbf{x}_t; \theta)}_{\text{NN output}}), \quad \theta = (\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)})$$

- L_2 regularization or "weight decay" = favor smaller weights

$$\mathcal{L}_\lambda(\theta; \mathcal{X}) = \mathcal{L}(\theta; \mathcal{X}) + \frac{\lambda}{2} \|\theta\|_2^2$$

- Modern variant: drop out (training with noise)

Section 2

Backpropagation

Stochastic Gradient Descent

- Optimize using gradient descent
  - loss function is typically non-convex: no/little theoretical guarantees
  - practice: just do it; saddle points more of an issue than poor local minima
- SGD (stochastic gradient descent)
  - steepest descent is too expensive for large data sets
  - SGD with step size η, pick data point t at random

$$\theta \leftarrow (1 - \eta\lambda)\theta - \eta \, \nabla_\theta \ell(y_t^*; y(\mathbf{x}_t; \theta))$$

Loss Gradients

- Large (many units) and deep (many layers) networks: many weights = partial derivative for each
  - sensitivity of output/loss with regard to each weight
- Use chain rule to compute derivatives
  - output layer = gradient of loss

$$\nabla_{\mathbf{y}}\, \ell = \ldots \quad \text{(depends on loss)}$$

  - start computation from output!
  - example: squared loss

$$\nabla_y \ell = \frac{\partial \ell}{\partial y} = (y - y^*)$$

Layer-to-Layer Jacobian

- How do units affect each other?
  - $\mathbf{x}$ = previous layer activation
  - $\mathbf{x}^+$ = next layer activation
- Jacobian matrix $\mathbf{J} = (J_{ij})$ of mapping $\mathbf{x} \to \mathbf{x}^+$, $\mathbf{x}_i^+ = \sigma(\mathbf{w}_i^\top \mathbf{x})$

$$\mathbf{J} = \frac{\partial \mathbf{x}^+}{\partial \mathbf{x}}, \quad J_{ij} = \frac{\partial x_i^+}{\partial x_j} = w_{ij} \cdot \sigma'(\mathbf{w}_i^\top \mathbf{x})$$

  - (sometimes transposed definition of $\mathbf{J}$ in the literature)
  - essentially a modified weight matrix!

Backpropagation

- Across multiple layers (by chain rule), $1 \leq n < l$

$$\frac{\partial {x_i}^{(l)}}{\partial {x_k}^{(l-n)}} = \sum_j \underbrace{\frac{\partial {x_i}^{(l)}}{\partial {x_j}^{(l-1)}}}_{={J_{ij}}^{(l)}} \frac{\partial {x_j}^{(l-1)}}{\partial {x_k}^{(l-n)}},$$

$$\frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{x}^{(l-n)}} = \mathbf{J}^{(l)} \cdot \frac{\partial \mathbf{x}^{(l-1)}}{\partial \mathbf{x}^{(l-n)}} = \mathbf{J}^{(l)} \cdot \mathbf{J}^{(l-1)} \dots \mathbf{J}^{(l-n+1)}$$

  - one simply needs to multiply (layer-to-layer) Jacobians
  - ... and then

$$\nabla^\top_{\mathbf{x}^{(l)}} \ell = \underbrace{\nabla^\top_{\mathbf{y}} \ell \cdot \mathbf{J}^{(L)} \dots \mathbf{J}^{(l+1)}}_{\longrightarrow \text{ back propagation}}$$

From Activities to Weights

- How do weights affect loss?
- Simple local computation

$$\frac{\partial \ell}{\partial {w_{ij}}^{(l)}} = \frac{\partial \ell}{\partial {x_i}^{(l)}} \frac{\partial {x_i}^{(l)}}{\partial {w_{ij}}^{(l)}}, \quad \text{where}$$

$$\frac{\partial {x_i}^{(l)}}{\partial {w_{ij}}^{(l)}} = \underbrace{\sigma' \left(\left[{\mathbf{w}_i}^{(l)} \right]^\top \mathbf{x}^{(l-1)} \right)}_{\substack{\text{sensitivity of} \\ \text{down-stream unit}}} \quad \underbrace{{x_j}^{(l-1)}}_{\substack{\text{activation of} \\ \text{up-stream unit}}}$$

Section 3

Convolutional Neural Networks

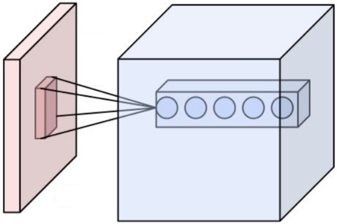
No Free Lunch!

- No learning machine can do well on all problems.
- Need to constrain function class appropriately.

Neural Networks for Images: Receptive Fields

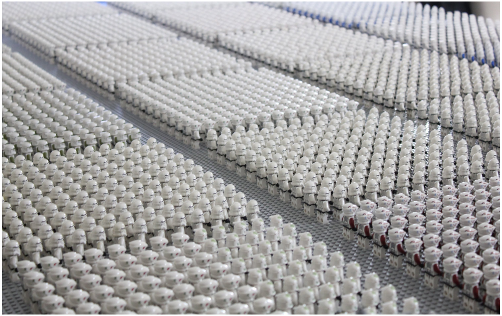
- Topological connectivity
  - encourage network to first extract localized features
  - subsequent layers: less and less localized features

- Receptive field
  - inputs that can affect a neuron (other weights $= 0$)
  - small images patches as receptive fields
  - can have multiple channels (in figure: 5)



Neural Networks for Images: Translation Invariance

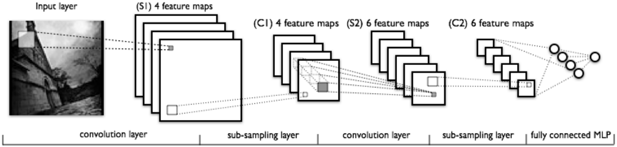
- Translation invariance of images
  - image patches look the same, irrespective of their location
  - idea: extract translation invariant features
  - what does that mean for a neural network?



Neural Networks for Images: Weight Sharing

- Weight Sharing
  - neurons share the same weights = compute same function
  - differ in location of their receptive field = different input
  - mirrors what has been done in image processing (manually)
- Shift-invariant Filters
  - layers learn shift-invariant filters
  - weights define a filter mask (e.g. 3x3 or 5x5)
  - typically as many neurons as inputs (border padding etc.)
    - e.g. 64x64 pixel per image $\Rightarrow$ 64x64 neurons per channel
  - color images: 3 color channels, 3-dimensional filter mask

CNN: Buildings blocks



- **Three building blocks:**
  - Convolutional layer
  - Pooling layer
  - Fully-connected layer

Convolutional Layers

- **Convolution:**
  - Mathematical operation on two functions (f and g)
  - It produces a third function that is typically viewed as a modified version of one of the original function
  - This operation can be used to detect edges in an image

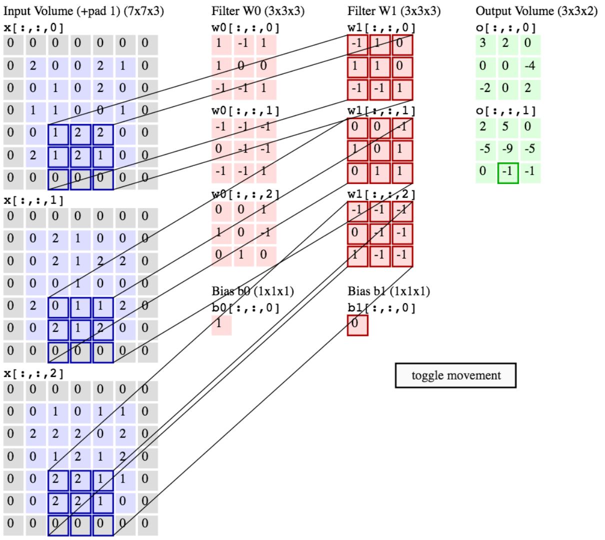
-1	0	1
-2	0	2
-1	0	1

Horizontal

-1	-2	-1
0	0	0
-1	-2	-1

Vertical

Convolutional Layers: Animation



cs231n.github.io/assets/conv-demo/index.html

Convolutional Layers: Mathematics

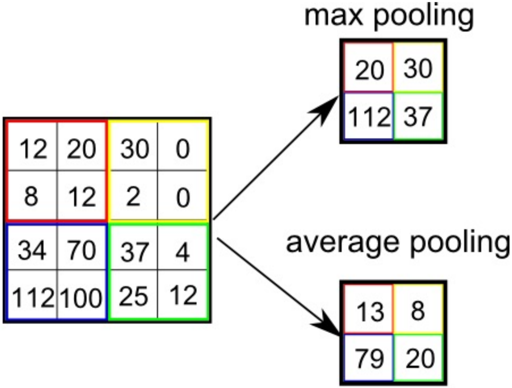
- Convolution in 2D (5x5)

$$F_{n,m}(\mathbf{x};\mathbf{w}) = \sigma\left(b + \sum_{k=-2}^{2}\sum_{l=-2}^{2} w_{k,l}\cdot x_{n+k,m+l}\right)$$

  - (n,m): center of receptive field
  - $\mathbf{x}$: image (2D pixel field)
  - $\mathbf{w}$: weights = arranged as a 2D mask
  - related to convolution in mathematics

Pooling

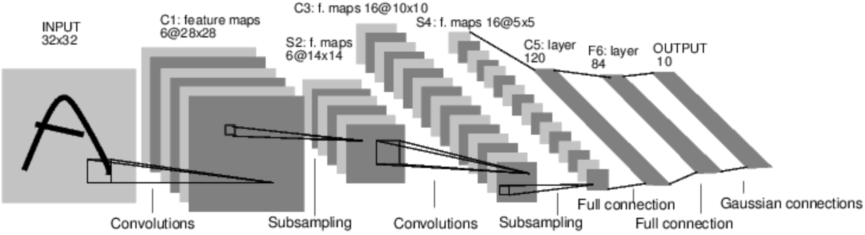
- Reduce size of convolutional layers by down-sampling
- Take average over window (e.g. 2x2)
- Common practice: max pooling = take maximum in window



Fully-connected layer

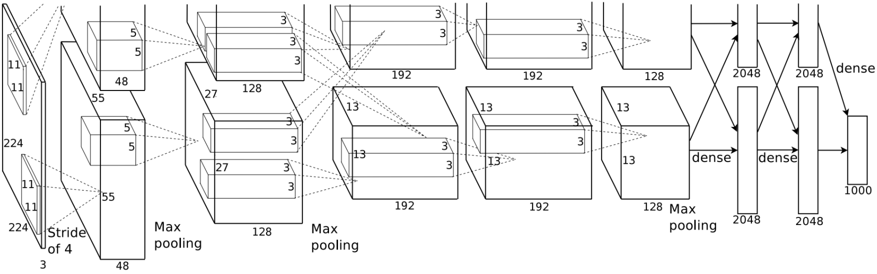
- High-level reasoning
- Connects all neurons in the previous layer to every single neuron it has
- Can be computed with a matrix multiplication

LeNet5



- Architecture LeNet5
  - layers C1/S2: 6 channels, cutting at border, 2x subsampling (4704 neurons)
  - layers C3/S4: 16 channels, cutting at border, 2x subsampling (1600 neurons)
  - layers F5/F6: fully-connected
  - output: Gaussian noise model (squared loss)

AlexNet



- AlexNet
  - Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, 2012 ImageNet Classification with Deep Convolutional NN
  - 60 million parameters and 500,000 neurons
  - 5 convolutional layers, some followed by max-pooling
  - 2 globally connected layers with a final 1000-way softmax

Learning the Filters

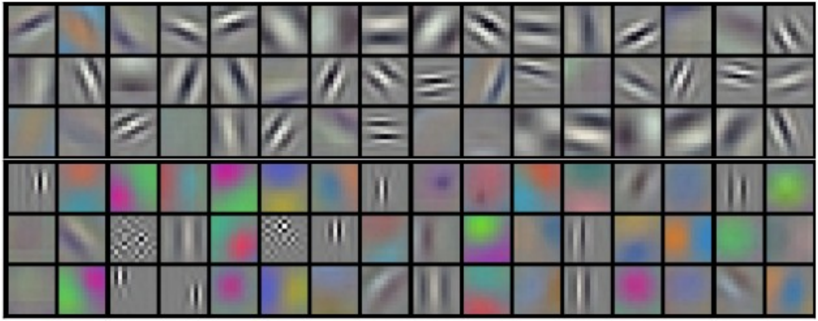
- Recall from last week: Optimize using stochastic gradient descent

$$\theta \leftarrow (1 - \eta\lambda)\theta - \eta \, \nabla_\theta L(y_t^*; y(\mathbf{x}_t; \theta))$$

- What do the filters look like then?

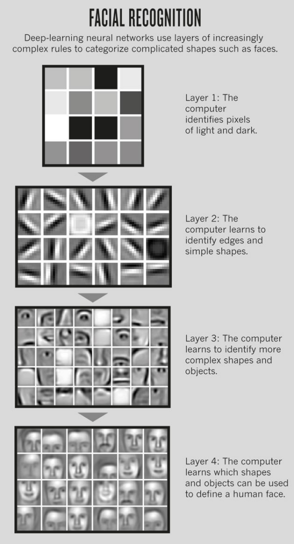
Learning Local Image Features

- Example: filters learned at first layer
- cf. Krizhevsky et al.: 96 filters of size 11x11x3



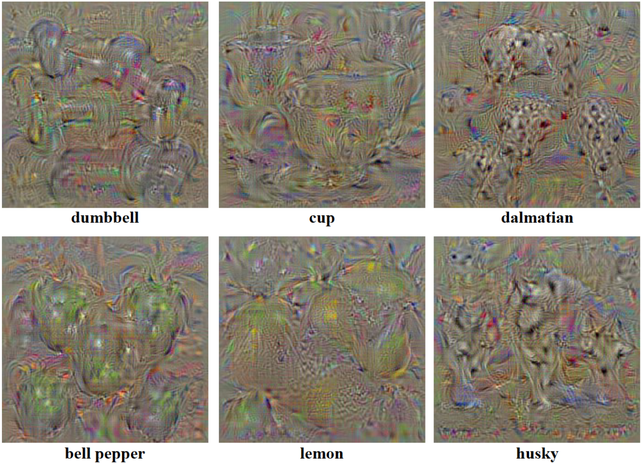
Learning Higher Level Features

- (c) Andrew Ng,
  trained on face images



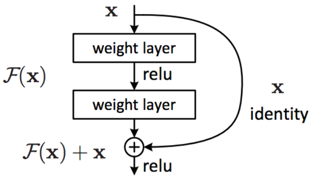
Saliency Maps

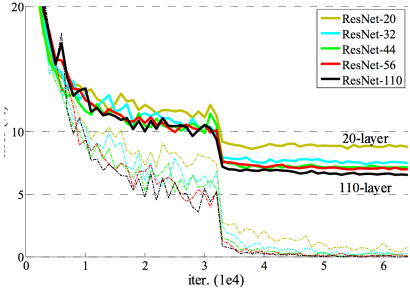
Per-class saliency maps for a CNN trained for visual classification (cf. Simonyan et al, 2015)



Deeper Nets

- ImageNet 2015 (Dec):
  - winner: residual networks
  - more than 100 layers deep





Semantic Segmentation

- CNNs can also be used for semantic segmentation.
- Typical architecture of a de-convolutional network (from Noh et al. 2015)