Imperial College London

Pattern Recognition

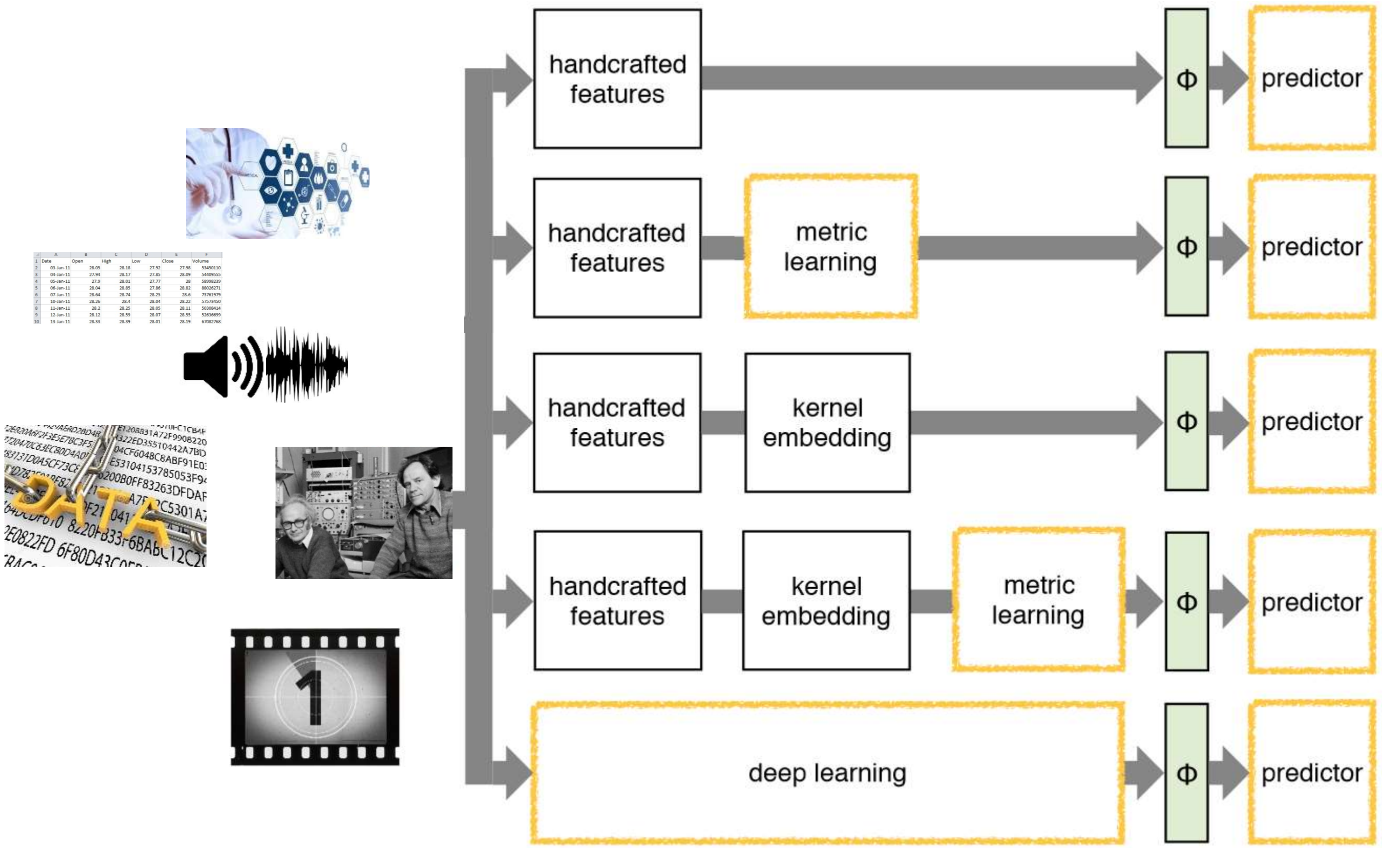
Neural Networks
Deep Learning

Krystian Mikolajczyk

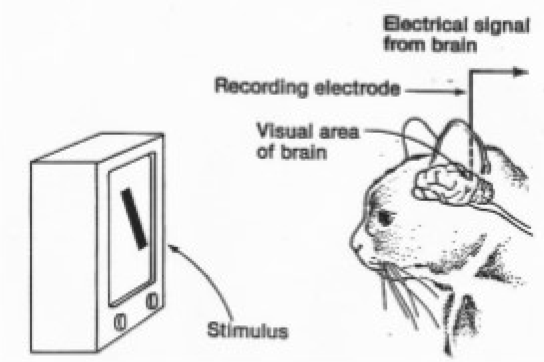
Blackboard

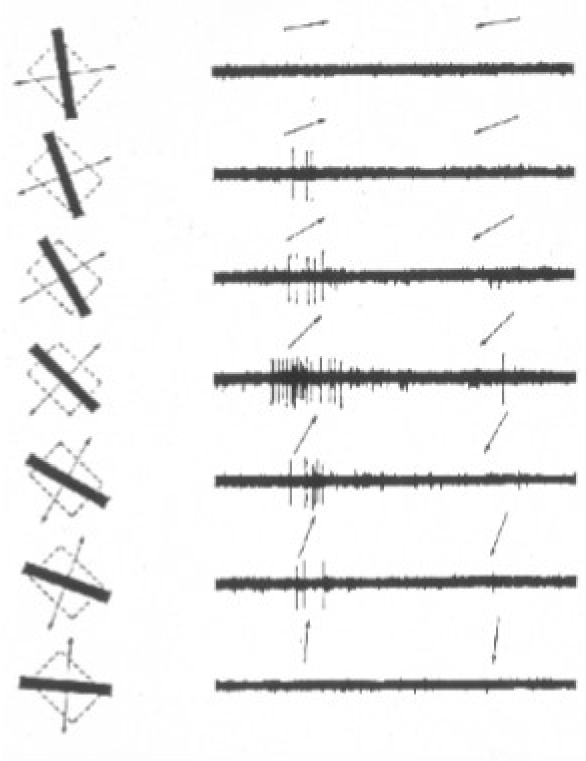
Learning predictors

- Different methods in pattern recognition

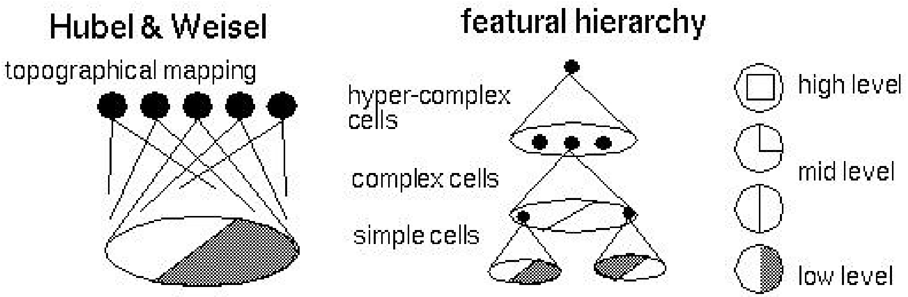


Hubel and Wiesel 1959



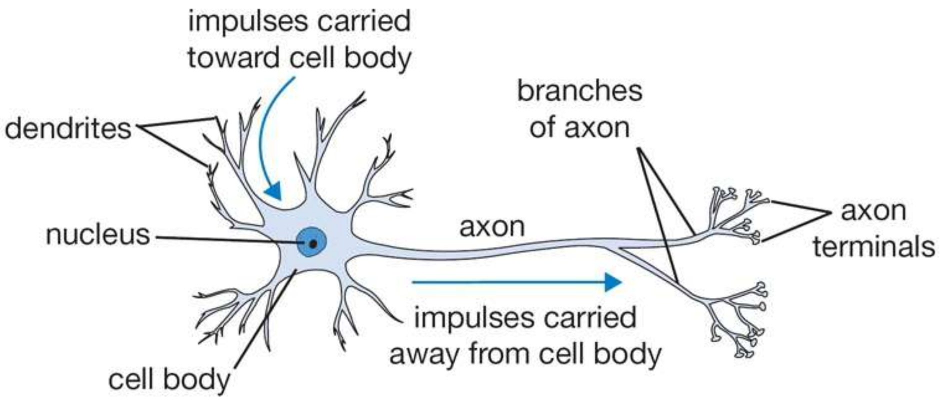


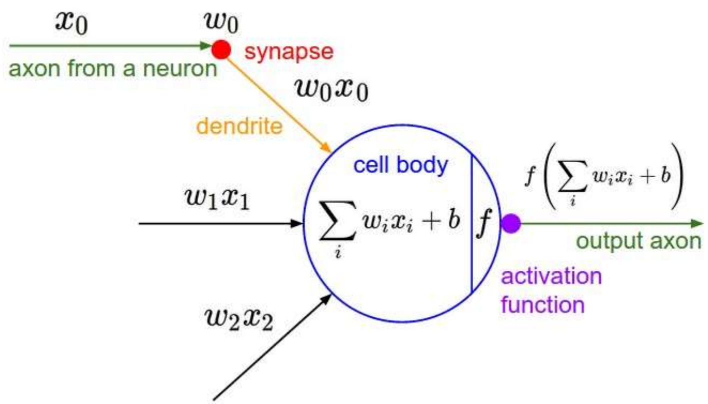
oriented filter



Perceptron model

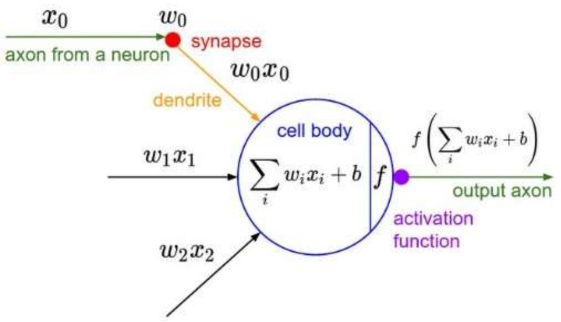
- Biologically inspired





- Perceptron: estimate the posterior probability of the binary label y of a vector **x**
- Convert *[-inf,+inf]* to range *[0,1]*
- Perceptron steps:
  1. Map a vector **x** to a scalar score by an affine projection *(w, b)*
  2. Transform the score monotonically but non-linearly by $\sigma(\boldsymbol{x})$

Perceptron model



- Training data: $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)$ $y \in \{0,1\}$
  - assume i.i.d. and compute the log-likelihood of the labels
- Likelihood $P(y_i = 1|\mathbf{x}_i, \mathbf{w}) = f(\mathbf{x}_i, \mathbf{w})$ $P(y_i = 0|\mathbf{x}_i, \mathbf{w}) = 1 - f(\mathbf{x}_i, \mathbf{w})$

$$P(y_i|\mathbf{x}_i, \mathbf{w}) = f(\mathbf{x}_i, \mathbf{w})^{y_i}(1 - f(\mathbf{x}_i, \mathbf{w}))^{1-y_i}$$

- Negative log likelihood

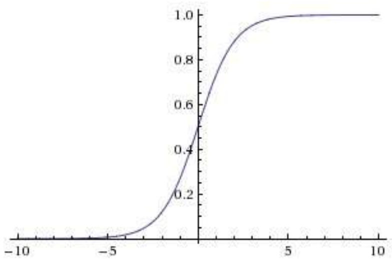
$$-\log P(y_i|\mathbf{x}_i, \mathbf{w}) = -y_i \log f(\mathbf{x}_i, \mathbf{w}) - (1 - y_i)\log(1 - f(\mathbf{x}_i, \mathbf{w}))$$

- Objective function (cross entropy loss) to minimise

$$E(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log f(\mathbf{x}_i, \mathbf{w}) + (1 - y_i)\log(1 - f(\mathbf{x}_i, \mathbf{w}))$$

- Optimizing it leads to sigmoid – binary softmax

$$\sigma(x) = \frac{1}{1 + e^{-w_1 x_1 - \cdots - w_D x_D - b}}$$

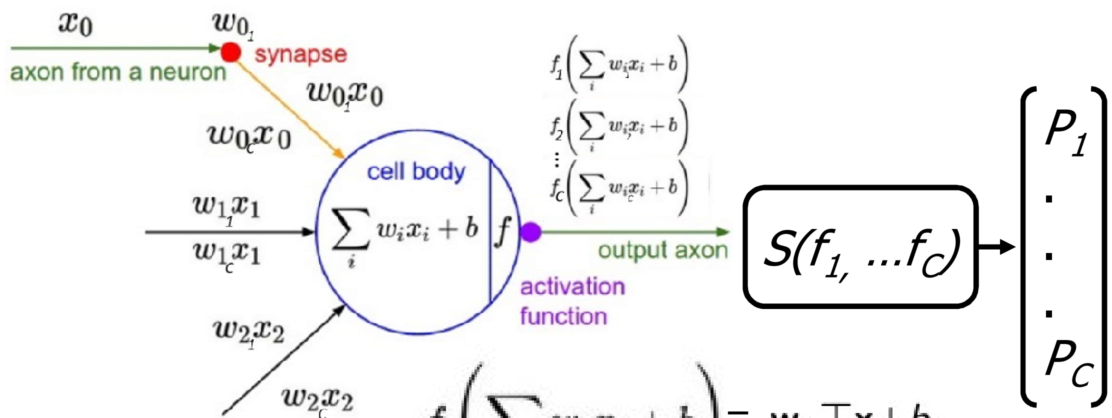


Multi-Class Perceptron model - softmax

- Training data: $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)$ $y \in \{1,2,3,\ldots,C\}$

  assume i.i.d. and compute the log-likelihood of the labels

$$P(y_i = c|\mathbf{x}_i, W) = \frac{e^{\mathbf{w}_c^\top \mathbf{x} + b_c}}{\sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x} + b_q}}$$



$$f\left(\sum_i w_i x_i + b\right) = \mathbf{w}_c^\top \mathbf{x} + b_c$$

- Negative log-likelihood of class $y_i = c$

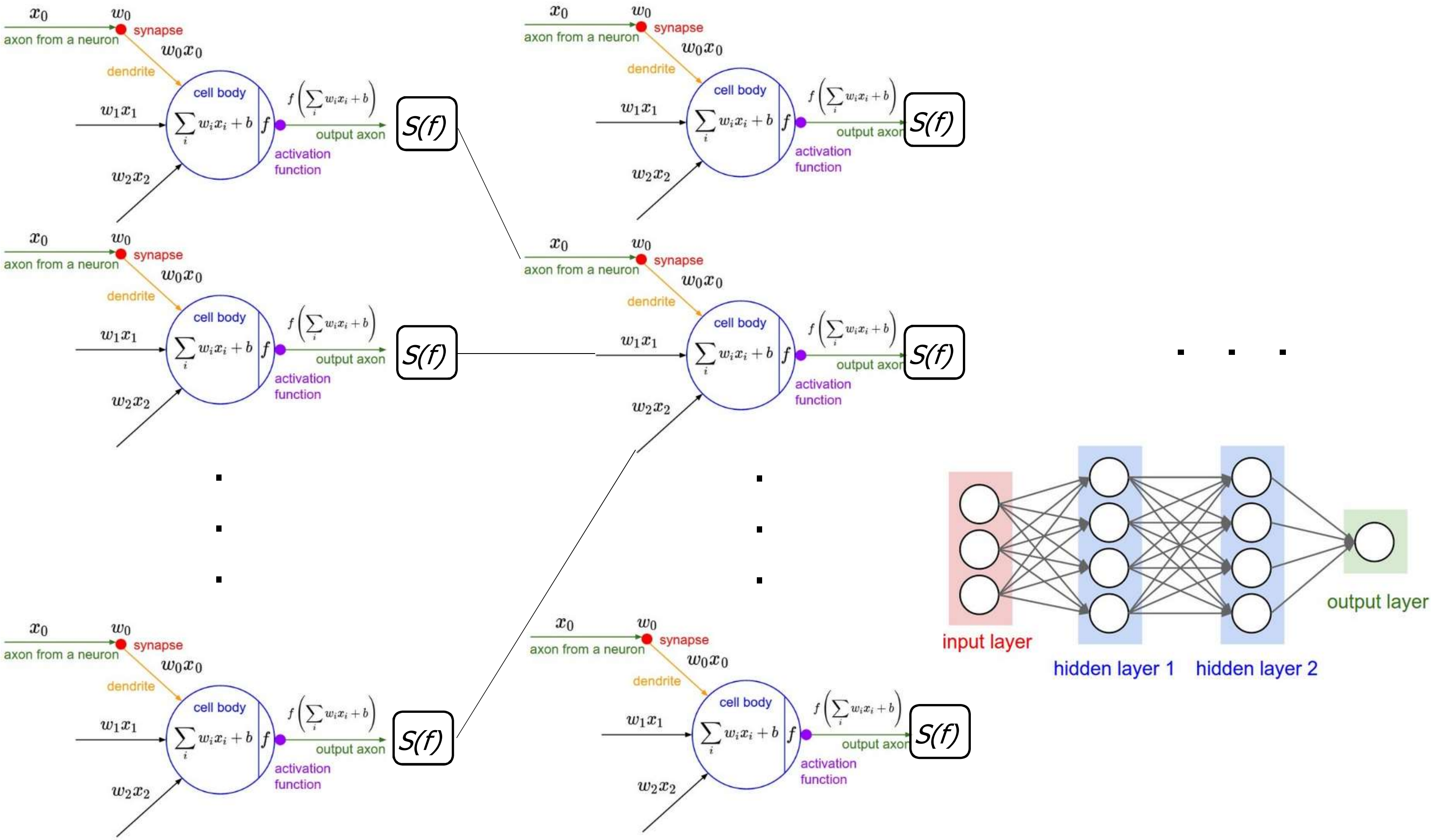
$$-\log P(y_i = c|\mathbf{x}_i, W) = -\log \frac{e^{\mathbf{w}_c^\top \mathbf{x} + b_c}}{\sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x} + b_q}} = -\mathbf{w}_c^\top \mathbf{x} - b_c + \log \sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x} + b_q}$$

- Objective function
  - cross entropy between empirical distribution y_i and the predicted one $P(y_i = c|\mathbf{x}_i, W)$

$$E(W) = \frac{1}{N} \sum_{i=1}^{N} \left(-\mathbf{w}_{y_i}^\top \mathbf{x}_i - b_{y_i} + \log \sum_{q=1}^{C} e^{\mathbf{w}_q^\top \mathbf{x}_i + b_q} \right)$$

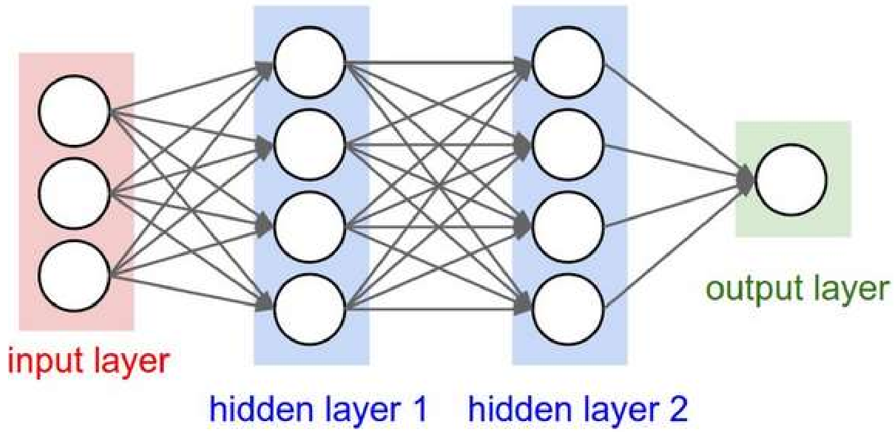
Multi-Layer Perceptron MLP

- Deep network



MLP Layer organization

- Artificial Neural Networks (ANN) or Multi-Layer Perceptrons (MLP) based on neurons-units cells
  - Neurons visualised in graphs
- A N-layer neural network with inputs, hidden layers of K neurons each and one output layer
  - There are connections (synapses) between neurons across layers, but not within a layer.
  - N-layer neural network, excluding input
  - Single layer NN have no hidden layers, input mapper onto output (SVM, logistic regression)
  - Output layer neurons most commonly do not have an activation function - last output layer represents the class scores (real-valued)



MLP Feed-forward example

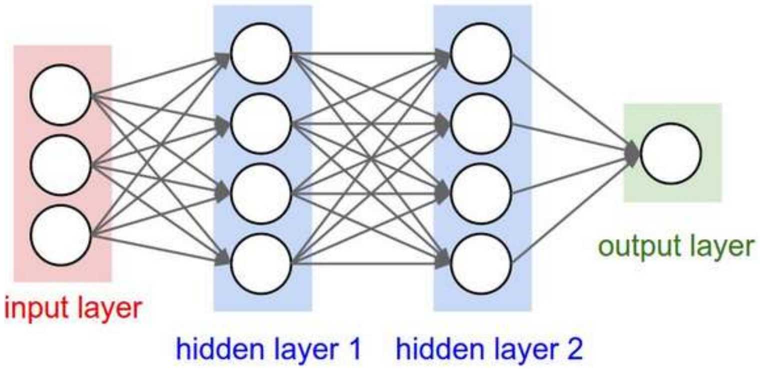
- Repeated matrix-vector multiplications and activation function
  - input is a [3x1] vector
  - weights W_1 of size [4x3], matrix with connections of the hidden layer, and the biases in vector b_1, of size [4x1].
    - single neuron has its weights in a row of W_c
    - matrix-vector multiplication evaluates the activations of all neurons in that layer.
  - W_2 is [4x4] matrix with connections, and W_3 a [1x4] matrix for the last (output) layer.
  - The full forward pass is simply three matrix multiplications and applications of the activation function
  - Size of the network - the number of parameters, number of layers
    - this network has 4 + 4 + 1 = 9 neurons, [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters

Single neuron

$$f\left(\sum_i w_i x_i + b\right) = \mathbf{w}_c{}^\top \mathbf{x} + b_c$$

Layer of neurons (matrix operation)

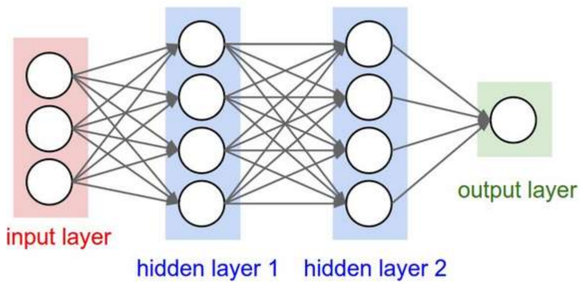
$$f(W_c\mathbf{x} + b_c)$$

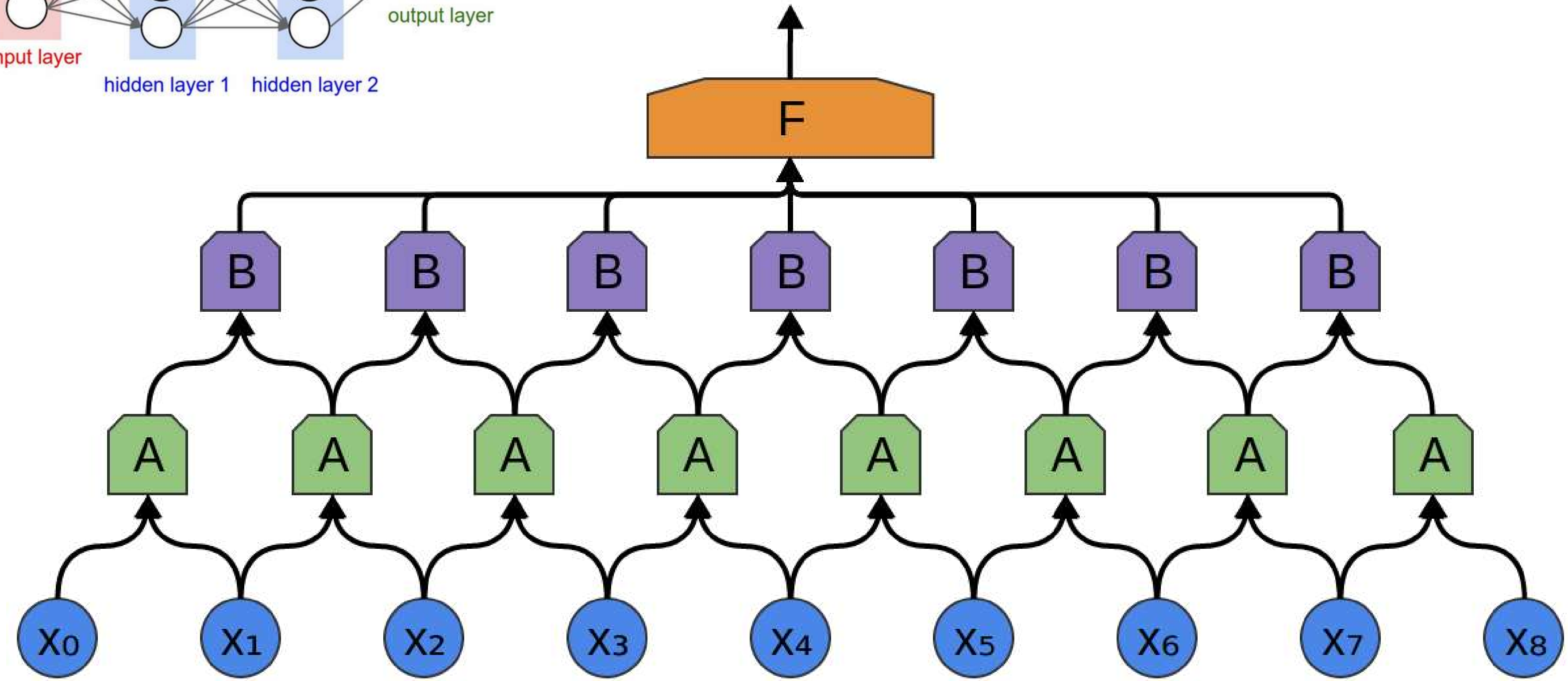


$f(W_1\mathbf{x} + b_1)$ $f(W_2\mathbf{x} + b_2)$ $f(W_2\mathbf{x} + b_2)$

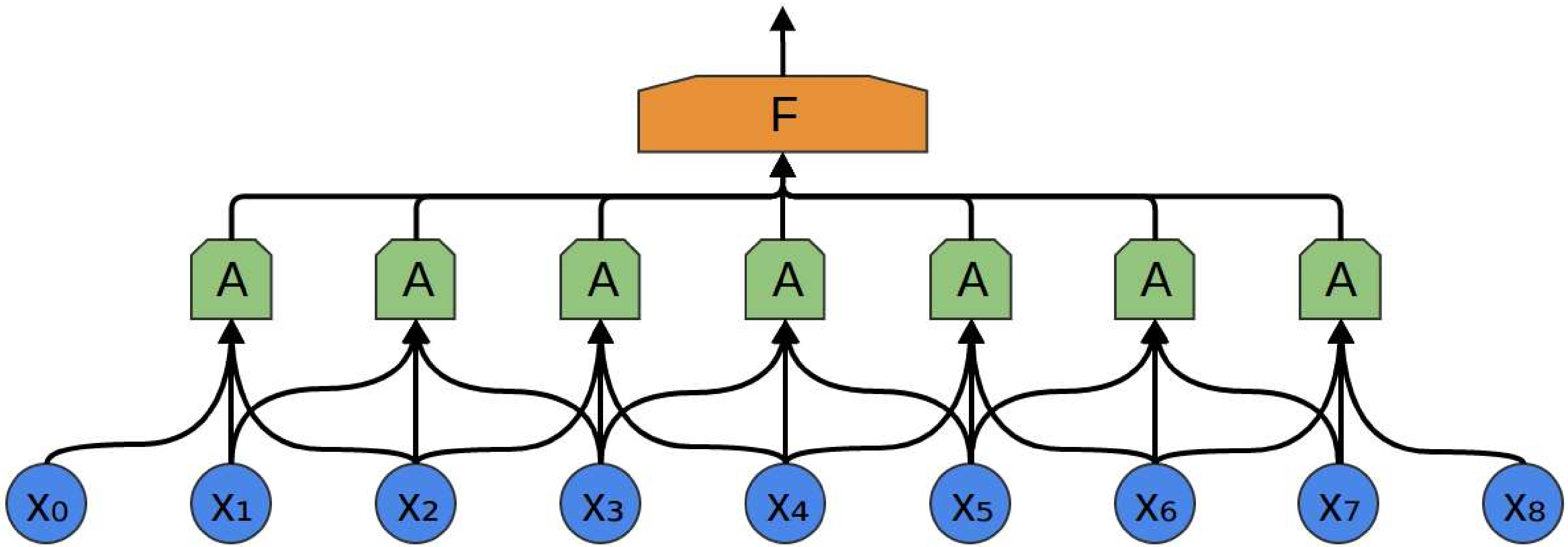
Convolutional Networks

Convolutional Networks

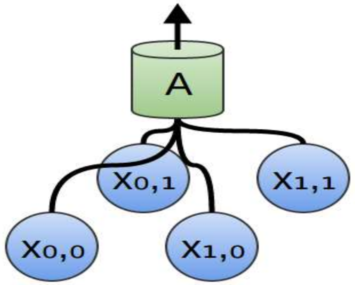




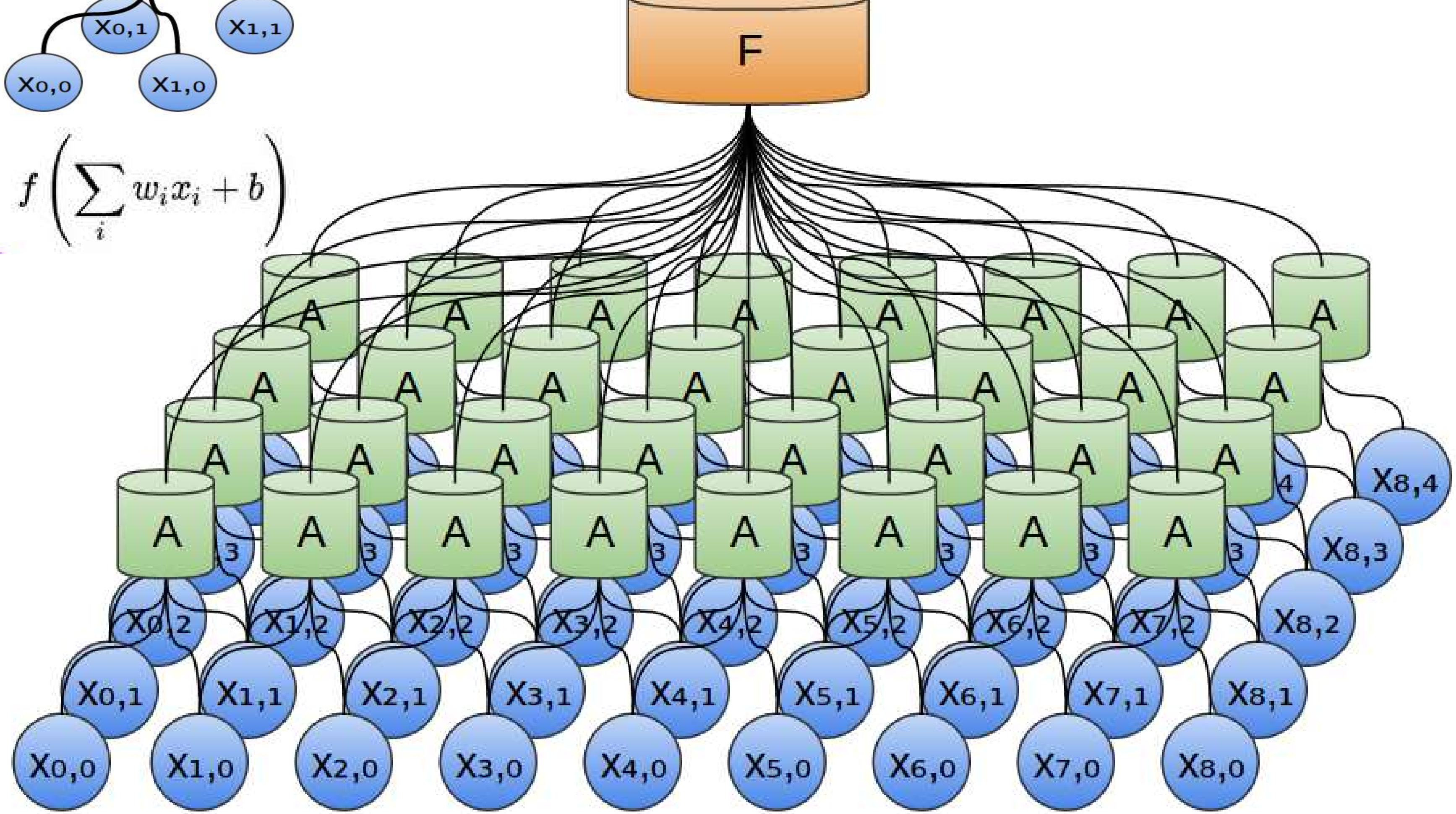
Convolutional Networks



Convolutional Networks

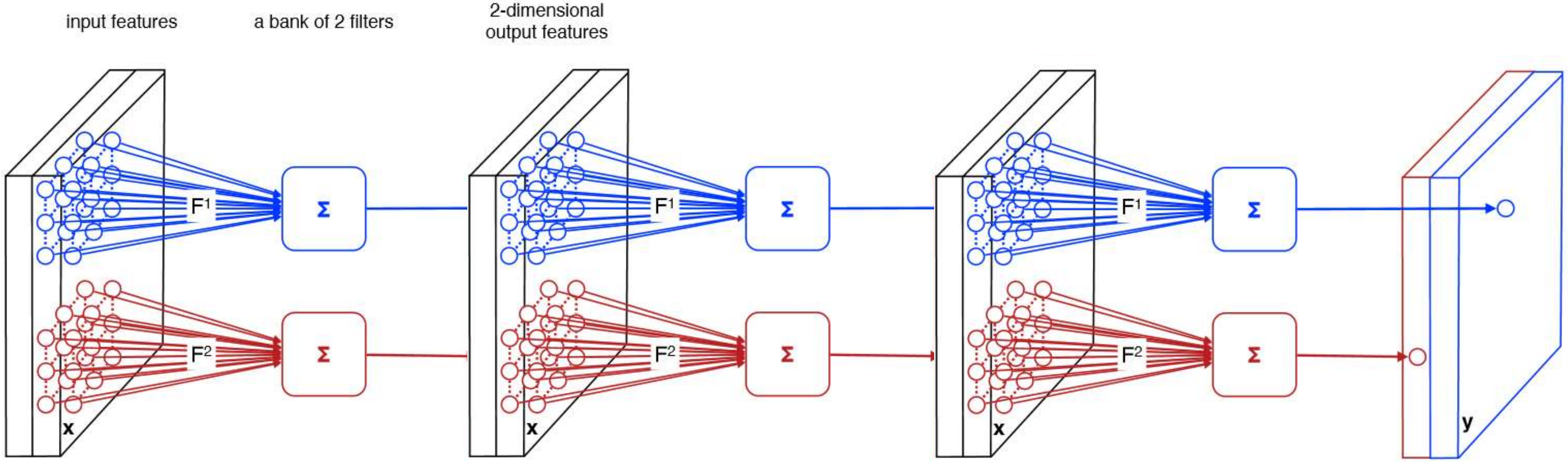


$$f\left(\sum_i w_i x_i + b\right)$$



Convolutional Networks

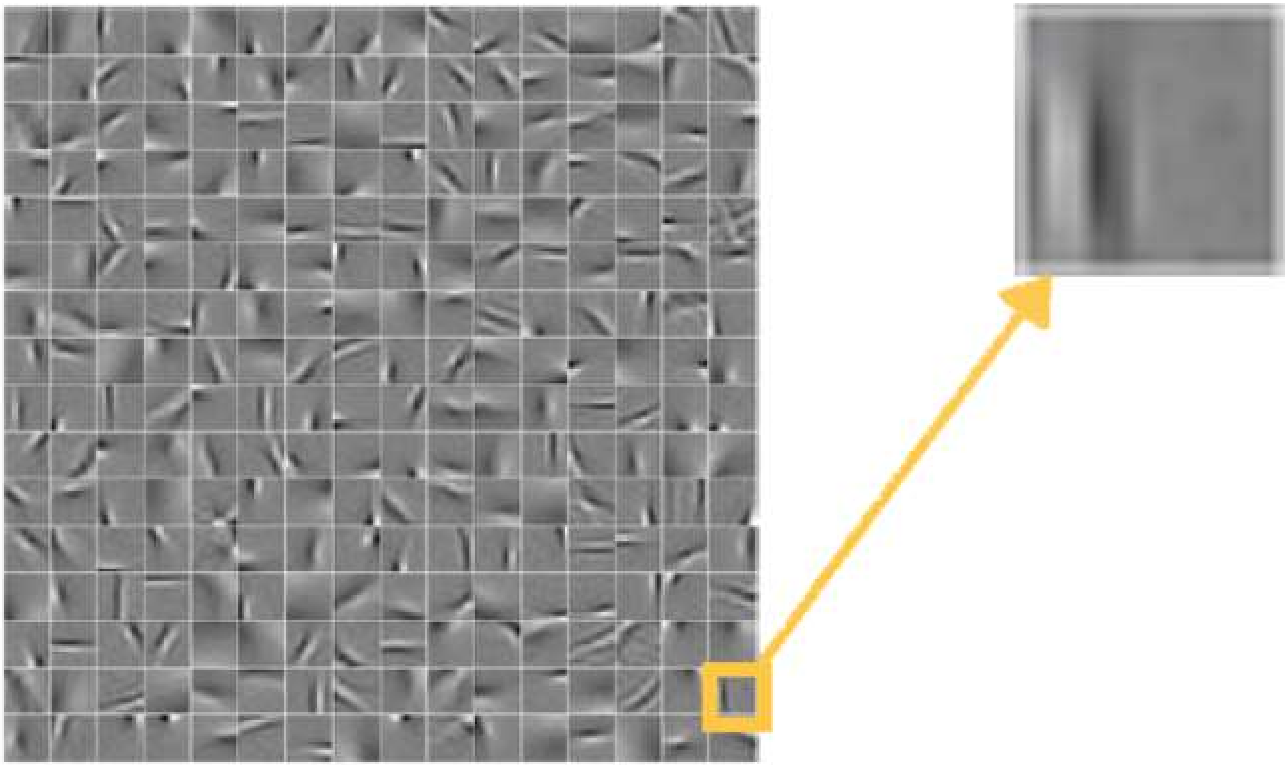
- Linear, translation invariant, local:
- Parameters for array input
  - Input x, array of size $H \times W \times K$
  - Filter bank, array of size $F = H' \times W' \times K \times Q$
  - Output $y = (H - H' + 1) \times (W - W' + 1) \times Q$ array
    - K input channels
    - Q filters of size H'xW'xK



Questions: goo.gl/Ad4cxw

Convolutional Networks

- Examples of filters learnt from image data
  - 256 of 16x16 of w_{ij}

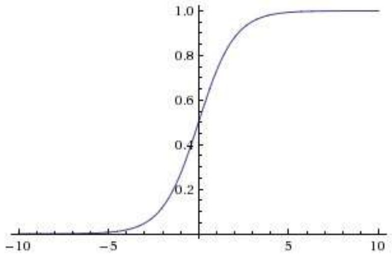


Network layers

Activation layers - gating

- **Sigmoid**

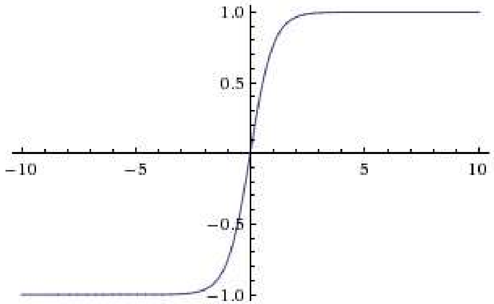
$$\sigma(x) = \frac{1}{1 + e^{-w_1 x_1 - \cdots - w_D x_D - b}}$$



  – output data that is not zero-centered (all positive or all negative), then the gradient on the weights w during backpropagation becomes all positive, or all negative (depending on the gradient of the whole expression *f* leading to undesirable zig-zagging dynamics in the gradient updates for the weights.

- **Tanh**

$$\tanh(x) = 2\sigma(2x) - 1 = \frac{2}{1 + e^{-2x}} - 1$$



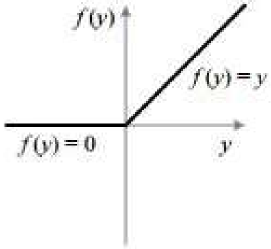
  – Transforms a real-valued number to the range [-1, 1], simply a scaled sigmoid

  – tanh non-linearity is always preferred to the sigmoid nonlinearity

  – Like the sigmoid, its activations saturate, but its output is zero-centered.

Activation layers - gating

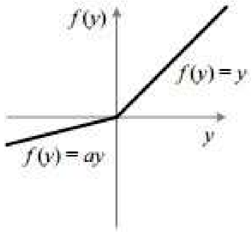
- **ReLU. Rectified Linear Unit** $f(x) = \max(0, x)$
  - The most frequently used, adds non-linearity to model non-linear functions
  - Has been argued to be more biologically plausible than sigmoid or tanh.
  - Simple, efficient, known as a ramp function - similar to half-wave rectification in EE.
  - A smooth approximation to the rectifier is softplus $f(x) = \ln(1 + e^x)$



- **Leaky ReLU. Parametric ReLUs**

$$f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x) \qquad f(x) = \max(x, ax)$$

  - attempt to fix the "dying ReLU" problem
  - allow a small, non-zero gradient when the unit is not active (of 0.01) or α is a small constant, can be a parameter for each neuron that can be learnt



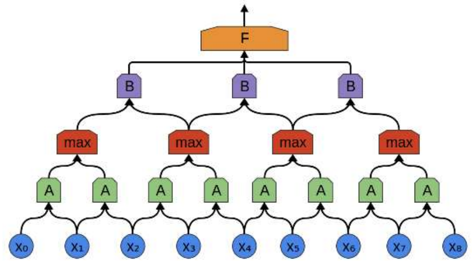
- **Maxout** $\max(w_1^T x + b_1, w_2^T x + b_2)$
  - both ReLU and Leaky ReLU are a special case - all the benefits of a ReLU unit (no saturation) and no its drawbacks (dying ReLU).
  - it doubles the number of parameters for every single neuron

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

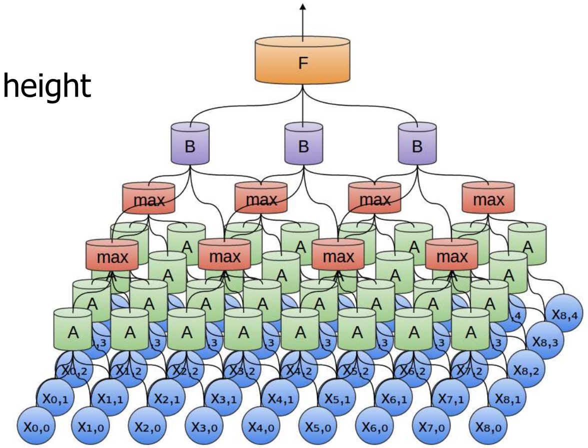
Pooling layers

Once feature has been found – (strong response of a receptive field), its sufficient to know its rough location relative to other features (robustness).

- partitions the input image into a set of non-overlapping/overlapping rectangles and outputs the combination of inputs for each such sub-region.
- it is a form of non-linear down-sampling
- provides a form of translation invariance.
- reduces the amount of parameters and computation,
- helps control overfitting in small datasets



- **Max pooling**
  - Most common in 2D, filters of size 2x2 applied with a stride of 2, takes a max over 4 numbers, discards 75% of the activations.
  - has been shown to work better in practice in neural nets.
  - downsamples at every depth slice in the input by 2 along both width and height
  - The depth dimension remains unchanged.

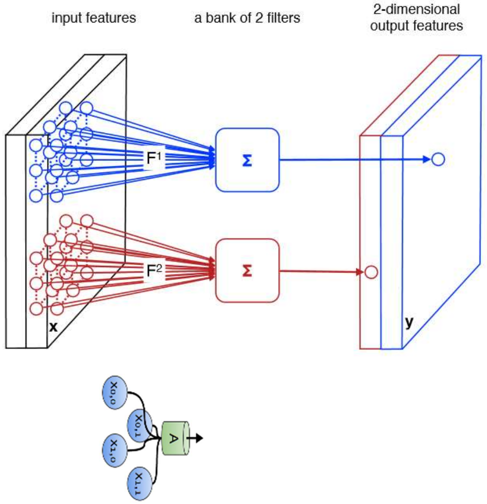


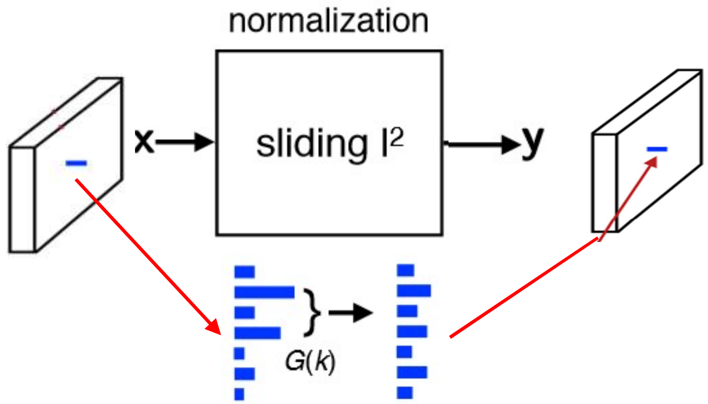
- **Average pooling**
  - Averaging neighbourhood
- **L2-norm pooling**
  - often used with BoW

Current trend is towards using smaller filters with a stride or discarding the pooling layer altogether (too aggressive reduction of the representation)

Normalization layers

- Normalize groups G(k) across channels (dimensions)



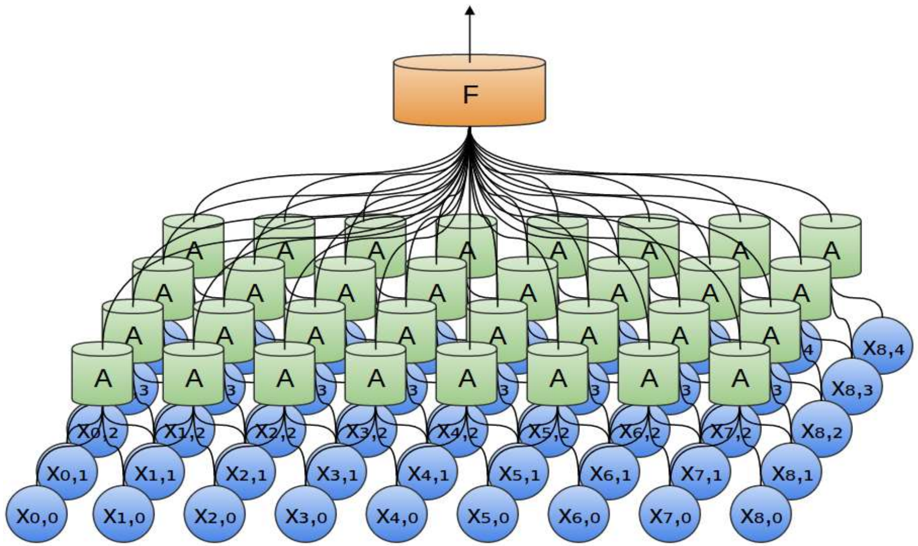


Operates at each spatial location independently

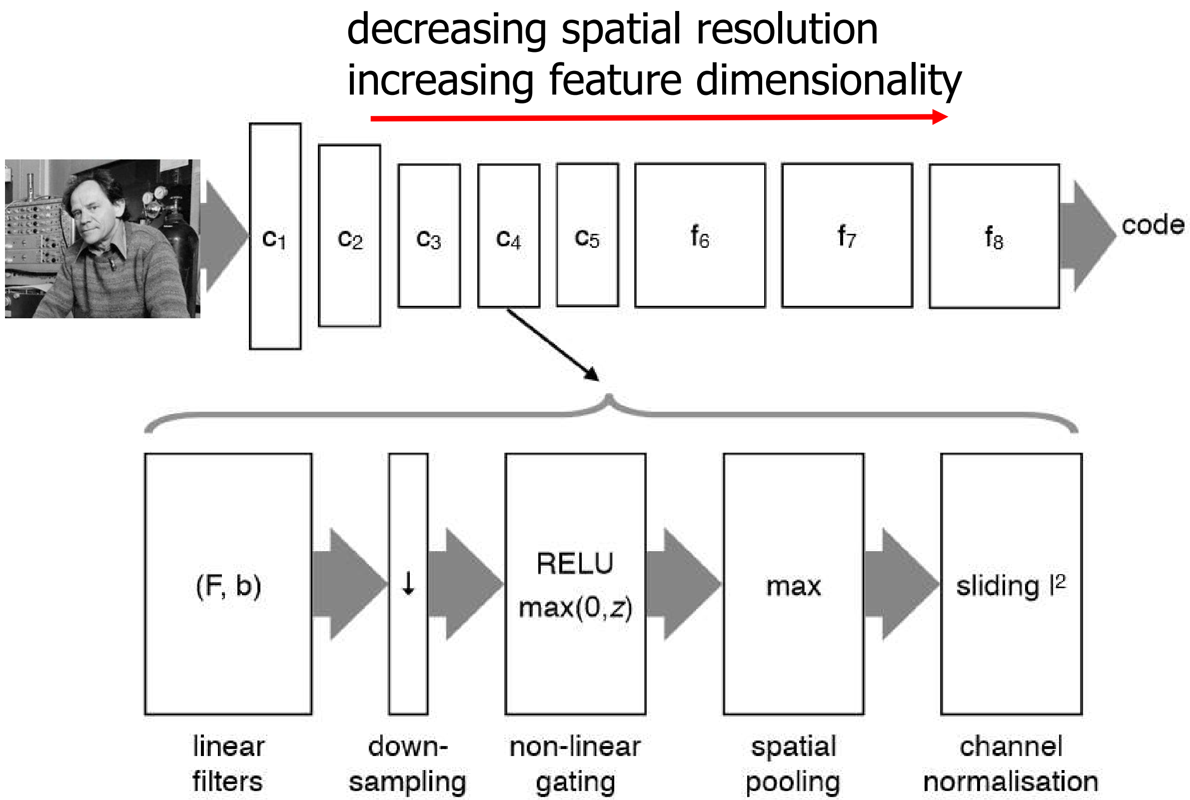
$$y_{ijk} = x_{ijk}\left(\kappa + \alpha \sum_{q\in G(k)} x_{ijq}^2\right)^{-\beta}$$

Fully connected layers

- The high-level reasoning in the neural network is done via fully connected layers.
- Neurons have full connections to all activations in the previous layer
- Their activations can be computed with a matrix multiplication followed by a bias offset (no convolution)
- Fully connected layer occupies most of the parameters, it is prone to overfitting
- Can be interpreted as a convolutional layer with filters of size equal to the input volume



CNN Summary

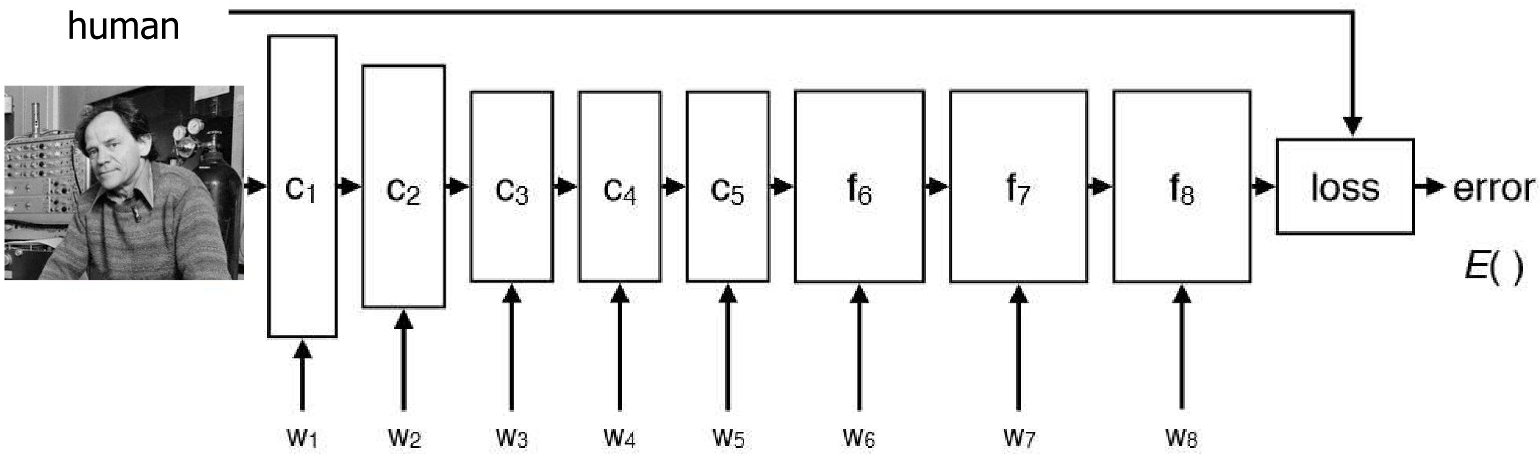


- Linear filters
- Downsampling
- Activation functions
- Pooling
- Normalization

- A linear classifier
  - INPUT -> FC
- Typical sequence of layers in CNN
  - INPUT -> CONV -> RELU -> FC
- Two sequences of single CONV layers followed by ReLU and POOL
  - INPUT -> [CONV -> Norm-> RELU -> POOL]*2 -> FC -> RELU -> FC
- Two CONV layers stacked before every POOL layer.
  - INPUT -> [CONV ->RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC

Traning Neural Network

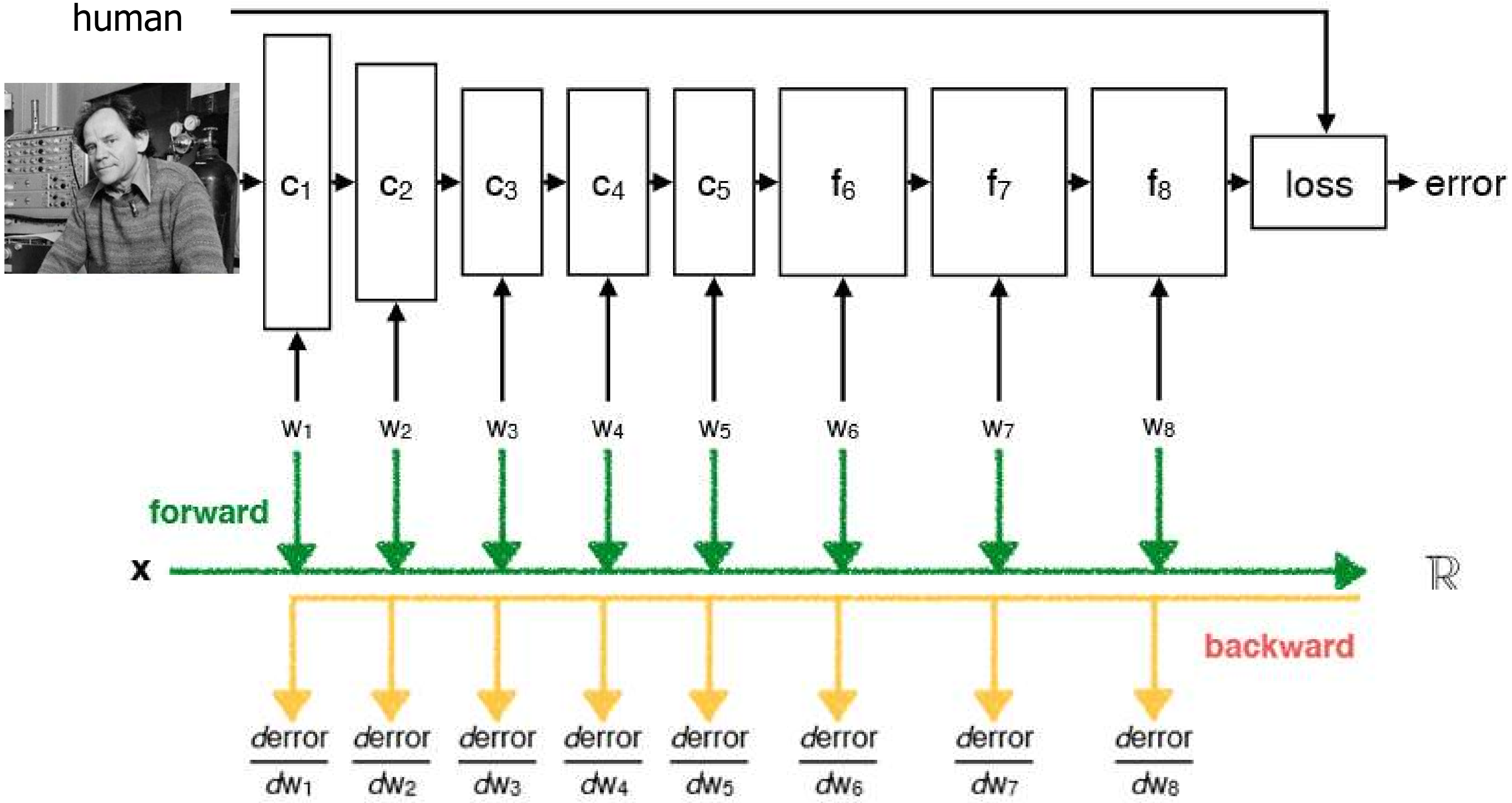
Learning NN



- Problems
  - many parameters,
  - prone to overfitting
- Key components
  - large annotated data
  - regularisation (dropout)
  - stochastic gradient descent
  - GPU(s) machines
  - days—weeks of training

$$\boxed{\operatorname{argmin} E(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_8)}$$

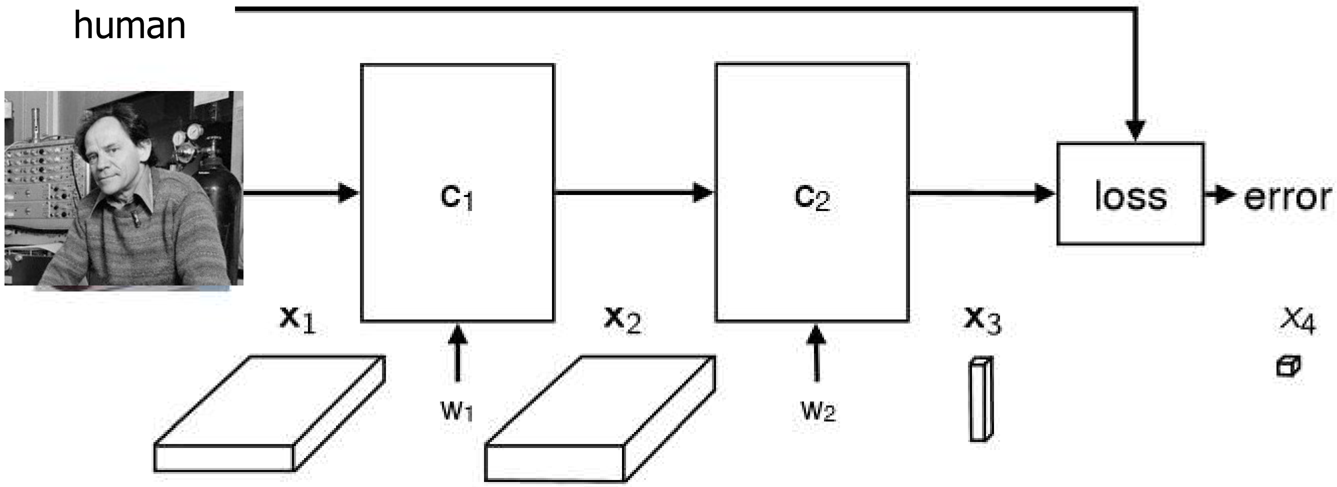
Learning NN



$$\boxed{\operatorname{argmin} E(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_8)}$$

Backpropagation

Naive application

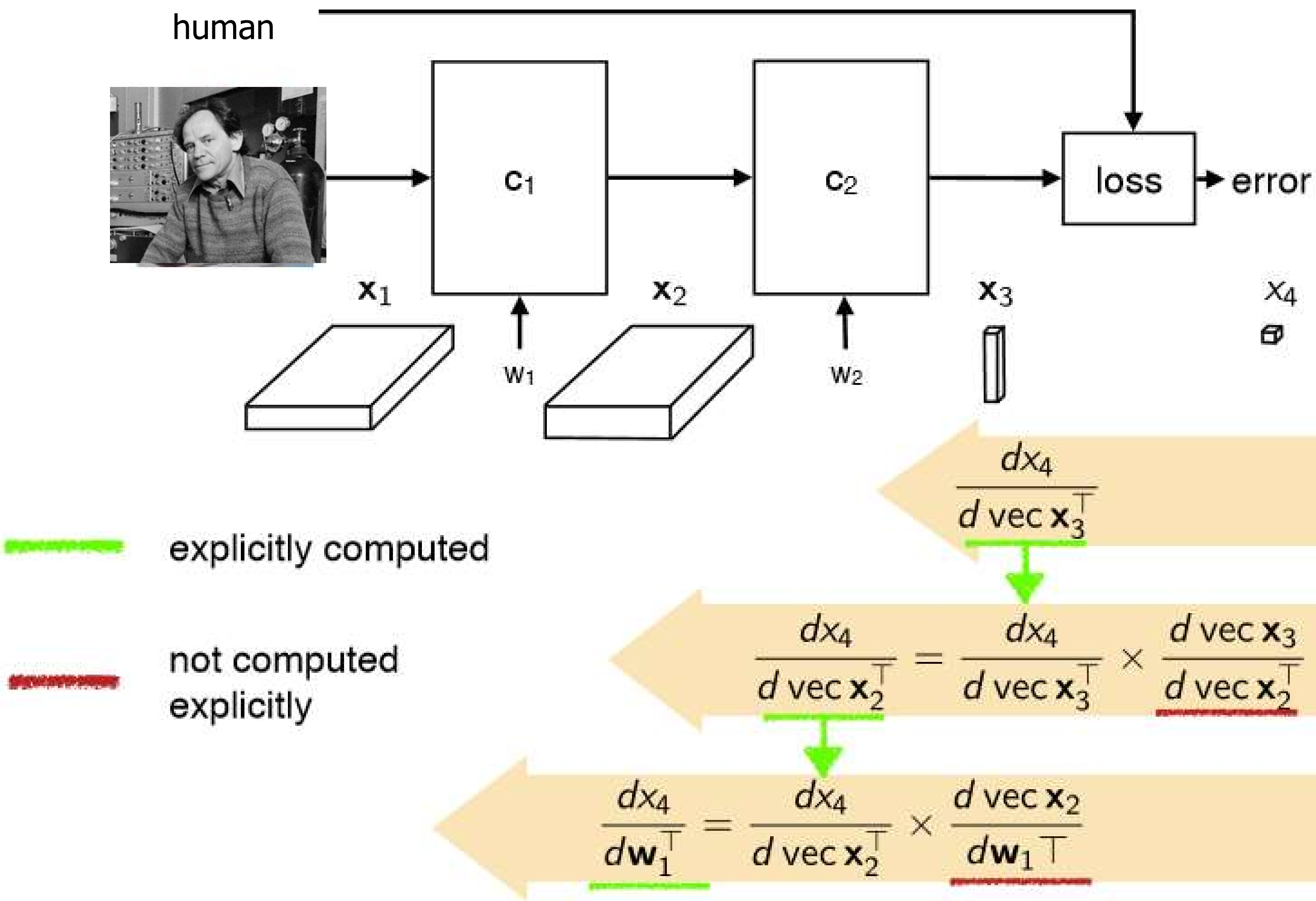


$$\frac{dx_4}{d\mathbf{w}_1^\top} = \frac{dx_4}{d\,\mathrm{vec}\,\mathbf{x}_3^\top} \times \frac{d\,\mathrm{vec}\,\mathbf{x}_3}{d\,\mathrm{vec}\,\mathbf{x}_2^\top} \times \frac{d\,\mathrm{vec}\,\mathbf{x}_2}{d\mathbf{w}_1^\top}$$

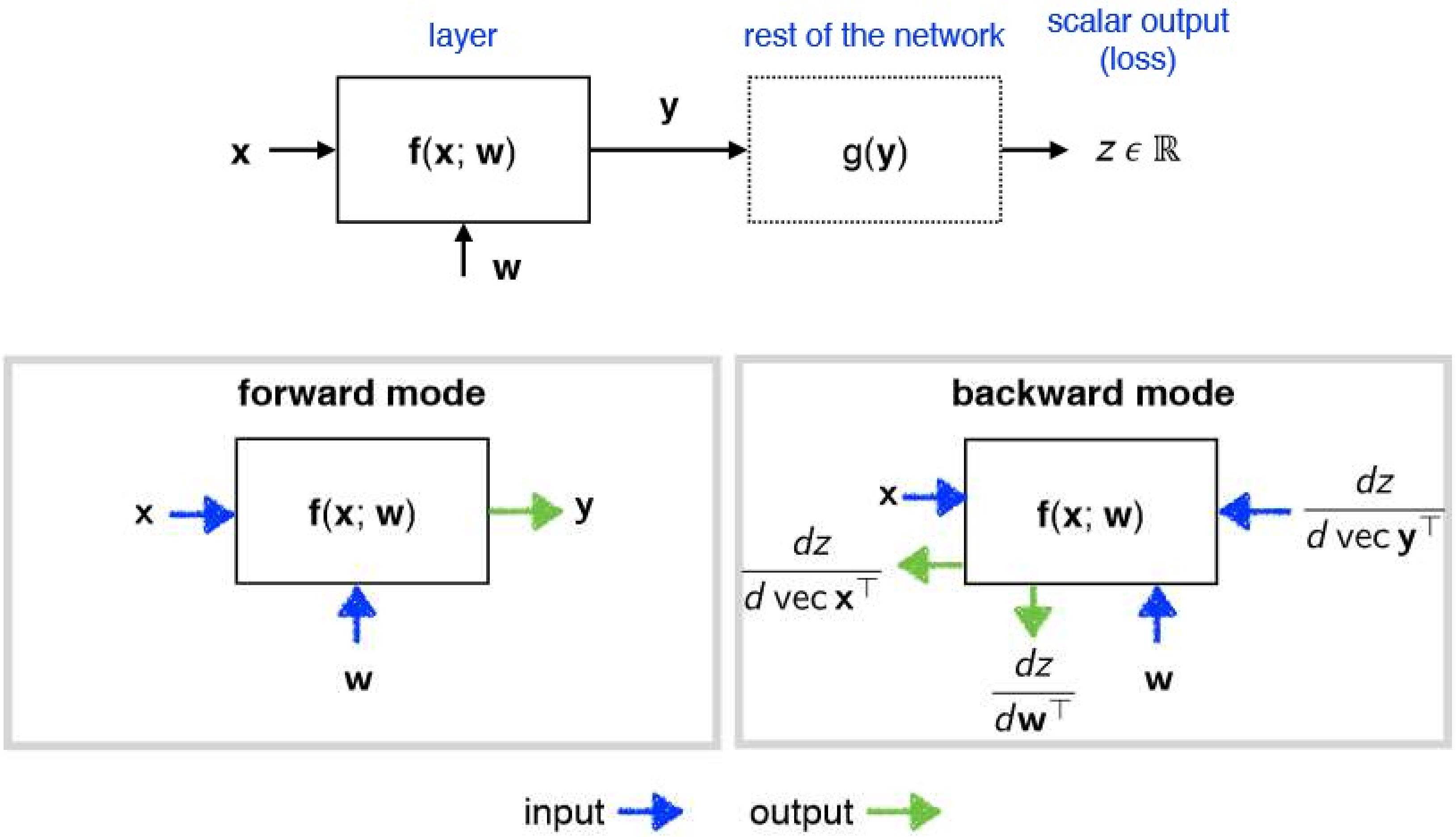
derivative matrix dimension	$1 \times \dim(\mathbf{w}_1)$	$1 \times H_3W_3K_3$	$H_3W_3K_3 \times H_2W_2K_2$	$H_2W_2K_2 \times \dim(\mathbf{w}_1)$
	✔	✔	✘	✘

E.g. $H_3=H_2=W_3=W_2=64$ and $K_3=K_2=256$ $\Longrightarrow$ 8GB for a single derivative!

Backpropagation



Backpropagation



Initialization

- The **weights** should be different than zero and vary
  - if every neuron in the network has the same output, then it'll have the same gradients during backpropagation and undergo the exact same parameter updates – no proper learning
- Approx. half of the weights are positive and half of them are negative – symmetry
- Every neuron's weight vector is initialized as a random vector sampled from a multi-dimensional Gaussian w_0= randn(n)
- Distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.
  - normalize the variance of each neuron's output to give approximately the same initial output distribution and empirically improve the rate of convergence.
  - For ReLU neurons the variance of neurons in the network should be 2.0/n

w_0= randn(n) * sqrt(2.0/n)

- Biases ***b*** should be zero, as the asymmetry breaking is provided in the weights.

Training with Gradient Descent

- Loss function $\ell(\hat{y}, y)$ for a family $\mathcal{F}$ of functions $f_w(x)$ parametrized by a weight vector *w,* seek the function $f \in \mathcal{F}$ that minimizes the loss $Q(z, w) = \ell(f_w(x), y)$ averaged on samples $z_1 \dots z_n$

$$E(f) = \int \ell(f(x), y)\, dP(z) \qquad E_n(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i)$$

Expected risk — Empirical risk

statistical learning theory justifies minimizing the empirical risk instead of the expected risk

Each iteration updates the weights *w* on the basis of the gradient of E_n

$$w_{t+1} = w_t - \gamma \frac{1}{n}\sum_{i=1}^{n} \nabla_w Q(z_i, w_t)$$

Under sufficient regularity assumptions, when the initial estimate w_0 is close enough to the optimum, and when the learning rate is sufficiently small, GD achieves linear convergence.

Gradient descent

- Batch gradient descent = use all examples in each iteration
- Stochastic gradient descent: use 1 example in each iteration
- Mini-batch gradient descent : use 2-100 examples in each iteration

Stochastic gradient descent

The stochastic gradient descent (SGD) – simplification of GD

- Each iteration estimates the gradient with a single random example z_t, instead of computing the gradient of $E_n(f,w)$ exactly,

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t) \quad \Rightarrow \quad w_{t+1} = w_t - \gamma_t \nabla_w Q(z_t, w_t)$$

With regularization

$$w_{t+1} = (1 - \gamma_t \lambda) w_t - \gamma_t y_t x_t \ell'(y_t w_t x_t)$$

- – does not need to remember which examples were visited during the previous iterations
- – it can process examples on the fly in a deployed system
- – SGD directly optimizes the expected risk, since the examples are randomly drawn from the ground truth distribution

Mini-Batch Stochastic Gradient Descent

Mini batch gradient descent e.g. 10 examples

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t)$$

- Faster than full batch GD
- Also better for monitoring progress
- Faster than SGD if the examples are vectorised to parallelise computations.
- One extra parameter to optimize – batch size

LOSS FUNCTIONS

Loss functions

Loss layer specifies how the network training penalizes the deviation between the predicted and true labels

Loss = Data Loss + Regularization Loss

- data loss measures the compatibility between a prediction (e.g. the class scores in classification) and the ground truth label.
  - it is an average over the data losses for every individual example *i* where N is the number of training examples $L = \frac{1}{N} \sum_i L_i$
- the regularization loss part of the objective can be seen as penalizing some measure of complexity of the model
- normally the last layer in the network
- various loss functions appropriate for different tasks
-

Data Loss

Assume a dataset of examples each with a single correct label *(j,y_i)*.

- **SVM** : $L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$
  - squared hinge loss $\max(0, f_j - f_{y_i} + 1)^2)$
- **Sigmoid** - a binary **logistic regression** classifier
- **Softmax** classifier that uses the **cross-entropy loss**
  - predicting a single class of K mutually exclusive classes $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$
- **L2 loss** $L_i = \|f - y_i\|_2^2$
  - **regression** in continuous space, predicting real-valued *f* and comparing to y_i,
- **L1 loss** $L_i = \|f - y_i\|_1$
  - **regression,** similar to L2, only in specific problems otherwise L2 is better
- Other loss functions e.g. structured prediction
  - refers to a case where the labels can be arbitrary structures such as graphs, trees, or other complex objects.

Data Loss

- **L2, L1 loss** is much harder to optimize than a more stable loss such as Softmax.
  - Requires the network to output exactly one correct value for each input in contrast to Softmax, where the precise value of each score is less important, only their magnitudes matter.
  - L2 loss is less robust because outliers can introduce huge gradients.
  - Gives just a single output with no indication of its confidence
  - Instead of regression consider quantizing the output into bins and do classification which can additionally give a distribution over the regression outputs.

Regularization loss

Loss = Data Loss + Regularization Loss

- **L2 regularization** - common $$E_n(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\ell(y_t w x_t)$$
  - where λ is the regularization strength. $1/2$ makes the gradient of this term simply λw.
  - heavily penalizing peaky weight vectors and preferring diffuse, small weight vectors.
  - due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather that some of its inputs a lot.
- **L1 regularization** - relatively common $$E_n(w) = \lambda\,|w| + \frac{1}{n}\sum_{i=1}^{n}\ell(y_t w x_t)$$
  - leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). Neurons with L1 regularization end up using only a sparse subset of their most important inputs and become robust to the "noisy" inputs
  - In practice L1 regularization can be expected to give inferior performance over L2.
- **Elastic net regularization**

$$E_n(w) = \lambda_1|w| + \frac{\lambda_2}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\ell(y_t w x_t)$$

Hyperparameters

- initial learning rate
- learning rate decay
- regularization strength (L2 penalty, dropout strength)
- momentum: smooth gradient using a moving average
- + many more relatively less sensitive hyperparameters

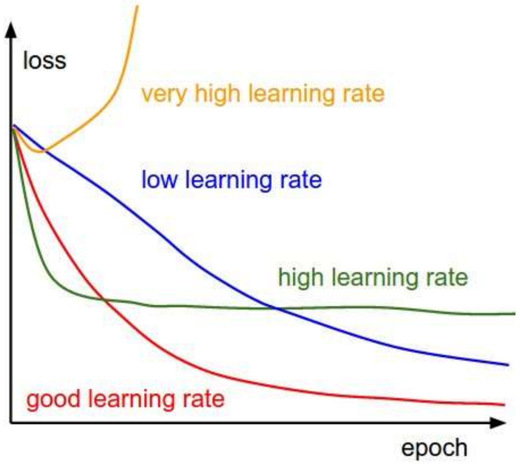
- Optimization

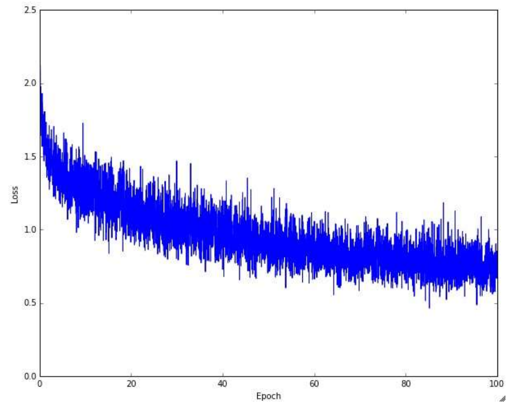
- use validation set of good size - no need for cross-validation with multiple folds.
- Search for hyper-parameters on log scale (eg. learning rates),
- careful about defining limits
- More efficient to optimize hyper-parameter with randomly chosen trials rather than on grid
- Do coarse to fine search

Random Search for Hyper-Parameter Optimization http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

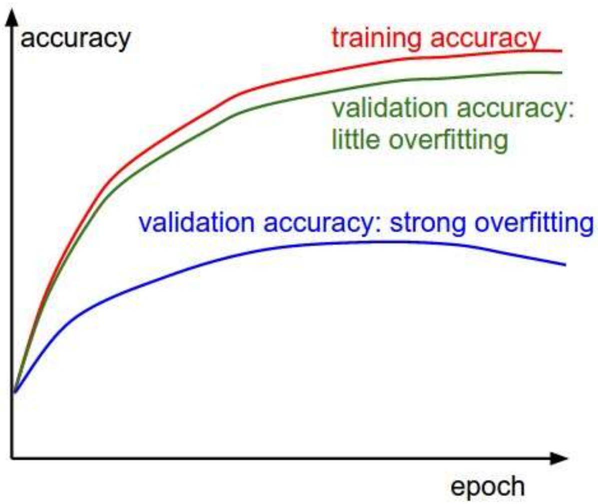
Monitoring Learning

- Loss function





- Train/Val accuracy



- Ratio of weights: update magnitudes
  - increase decrease learning rates
- Activation / Gradient distributions per layer

Model testing

- Use same model but with different initializations. Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization.
- Use cross-validation to determine the best hyperparameters, then pick the top few (e.g. 10) models to form the ensemble.
- If training is very expensive, take different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble.
- Maintain a second copy of the network's weights in memory that maintains an exponentially decaying sum of previous weights during training. This averages the state of the network over last several iterations.

Software Libraries

- CUDA-Convnet 1 & 2 https://code.google.com/p/cudaconvnet/
- Overfeat / Torch [Lua] http://cilvr.nyu.edu/doku.php? id=code:start
- Berkeley Caffe [Python] http://caffe.berkeleyvision.org
- Theano [Python] http://deeplearning.net/software/ theano/
- TensorFlow https://www.tensorflow.org/
- LibCCV http://libccv.org