UNIVERSITY OF CAMBRIDGE

4F10: Deep Learning

Mark Gales

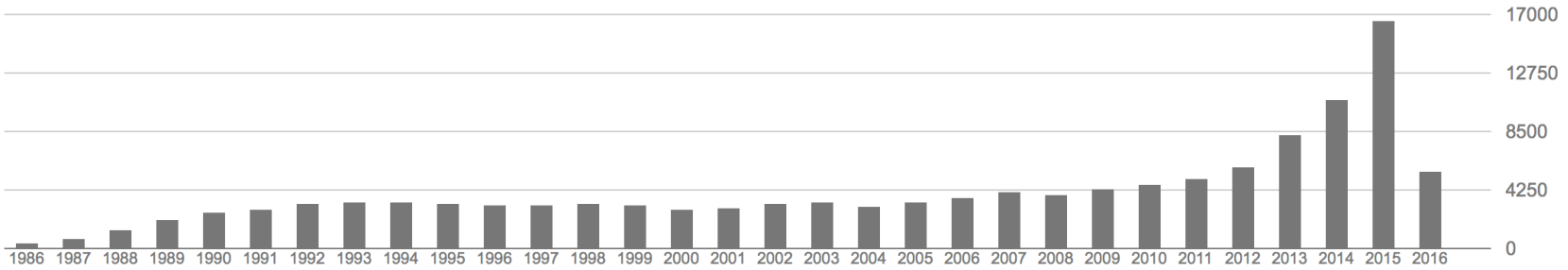
Michaelmas 2016

What is Deep Learning?

From Wikipedia:

> *Deep learning is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers, with complex structures or otherwise, composed of multiple non-linear transformations.*

The Rise of Deep Learning (June 2016)



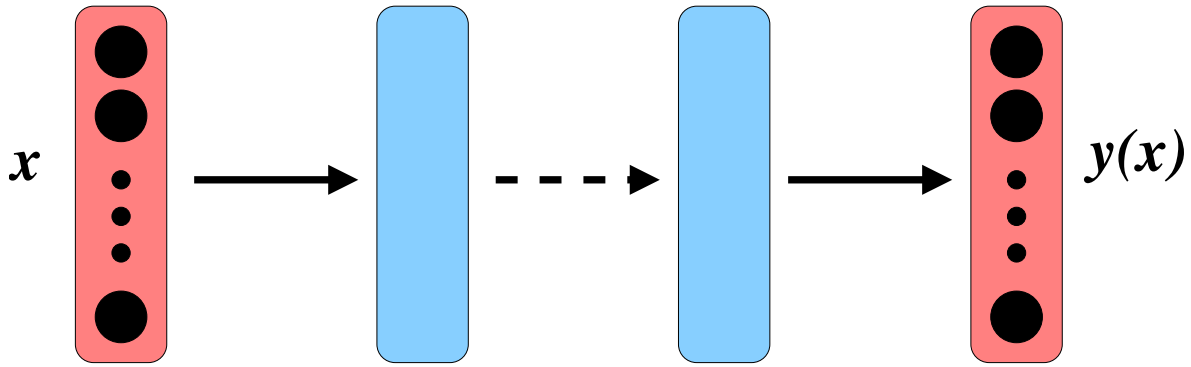
- Plot shows citations to Geoff Hinton papers
  - highly influential researcher in deep learning

Overview

- Basic Building Blocks
  - neural network architectures
  - activation functions
- Error Back Propagation
  - single-layer perceptron (motivation)
  - multiple-layer perceptron
- Optimisation
  - gradient descent refinement
  - second-order approaches, and use of curvature
  - initialisation
- Example
  - encoder-decoder models for sequence-to-sequence models
  - and attention for sequence-to-sequence modelling

Basic Building Blocks

Deep Neural Networks [12]

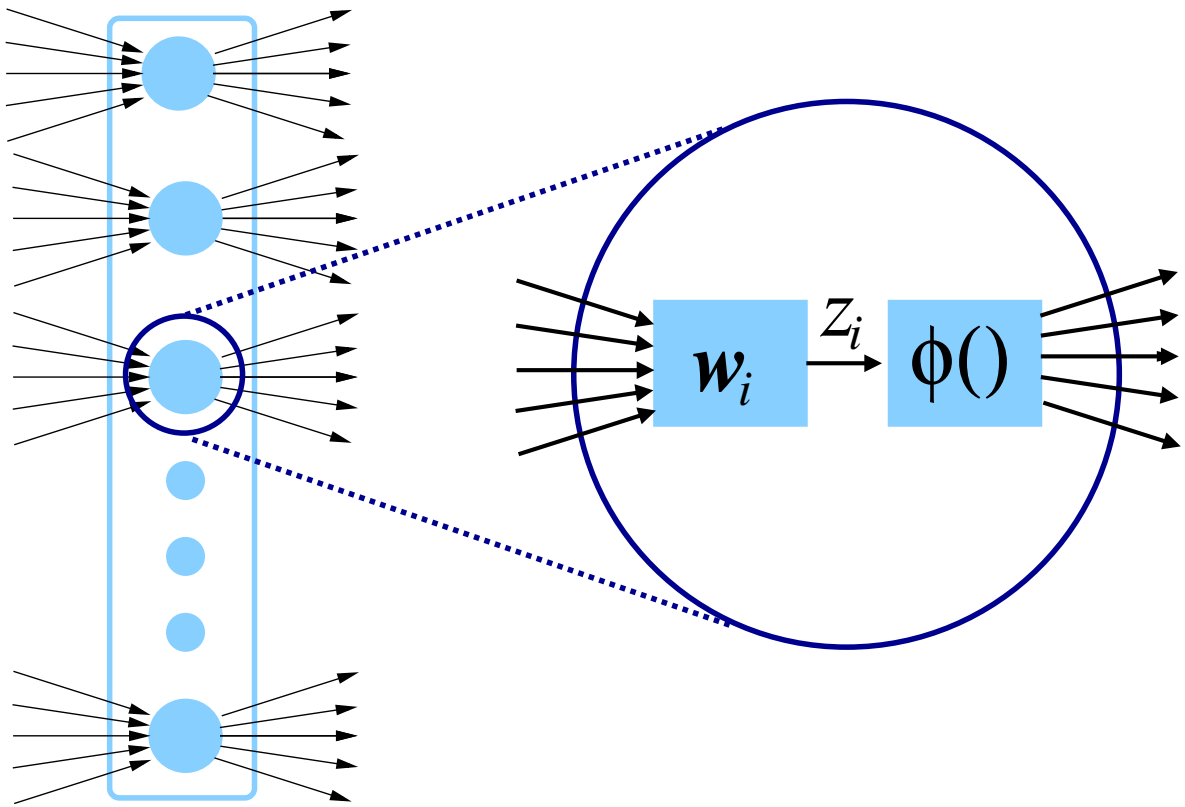


- General mapping process from input $\boldsymbol{x}$ to output $\boldsymbol{y}(\boldsymbol{x})$

$$\boldsymbol{y}(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x})$$

  - deep refers to number of hidden layers
- Output from the previous layer connected to following layer:
  - $\boldsymbol{x}^{(k)}$ is the input to layer k
  - $\boldsymbol{x}^{(k+1)} = \boldsymbol{y}^{(k)}$ the output from layer k

Neural Network Layer/Node



- General form for layer k:

$$y_i^{(k)} = \phi(\boldsymbol{w}_i' \boldsymbol{x}^{(k)} + b_i) = \phi(z_i^{(k)})$$

Initial Neural Network Design Options

- The input and outputs to the network are defined
  - able to select number of hidden layers
  - able to select number of nodes per hidden layer
- Increasing layers/nodes increases model parameters
  - need to consider how well the network generalises
- For fully connected networks, number of parameters (N) is

$$N = d \times N^{(1)} + K \times N^{(L)} + \sum_{k=1}^{L-1} N^{(k)} \times N^{(k+1)}$$

  - L is the number of hidden layers
  - $N^{(k)}$ is the number of nodes for layer k
  - d is the input vector size, K is the output size
- Designing "good" networks is complicated ...

Activation Functions

- Heaviside (or step/threshold) function: output binary

$$\phi(z_i) = \begin{cases} 0, & z_i < 0 \\ 1, & z_i \geq 0 \end{cases}$$

- Sigmoid function: output continuous, $0 \leq y_i(\boldsymbol{x}) \leq 1$.

$$\phi(z_i) = \frac{1}{1 + \exp(-z_i)}$$

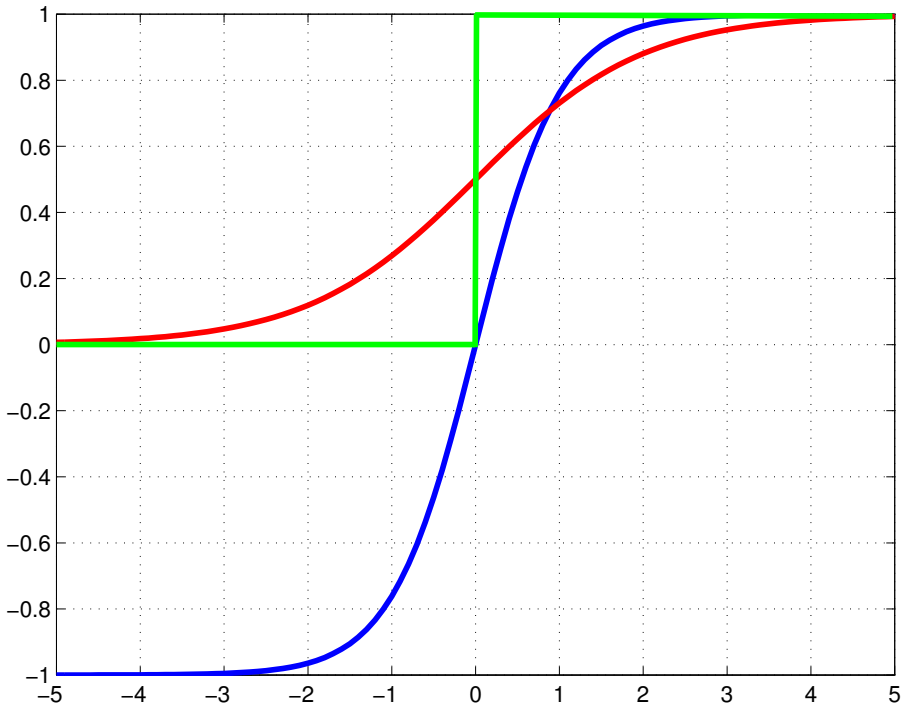
- Softmax function: output $0 \leq y_i(\boldsymbol{x}) \leq 1$, $\sum_{i=1}^{n} y_i(\boldsymbol{x}) = 1$.

$$\phi(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

- Hyperbolic tan function: output continuous, $-1 \leq y_i(\boldsymbol{x}) \leq 1$.

$$\phi(z_i) = \frac{\exp(z_i) - \exp(-z_i)}{\exp(z_i) + \exp(-z_i)}$$

Activation Functions

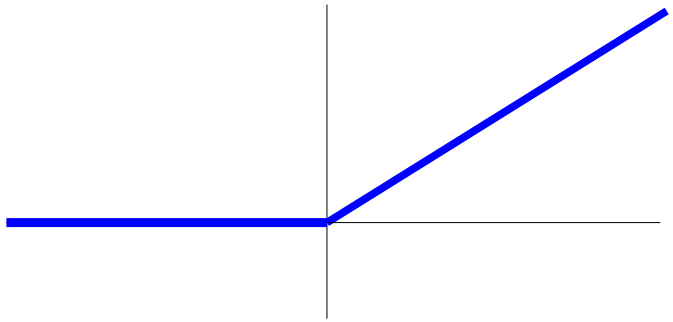


- Activation functions:
  - step function (green)
  - sigmoid function (red)
  - tanh function (blue)

- softmax, usual output layer for classification tasks
- sigmoid/tanh, often used for hidden layers

Activation Functions - ReLUs [18, 29]

- Alternative activation function: Rectified Linear Units



$$\phi(z_i) = \max(0, z_i)$$

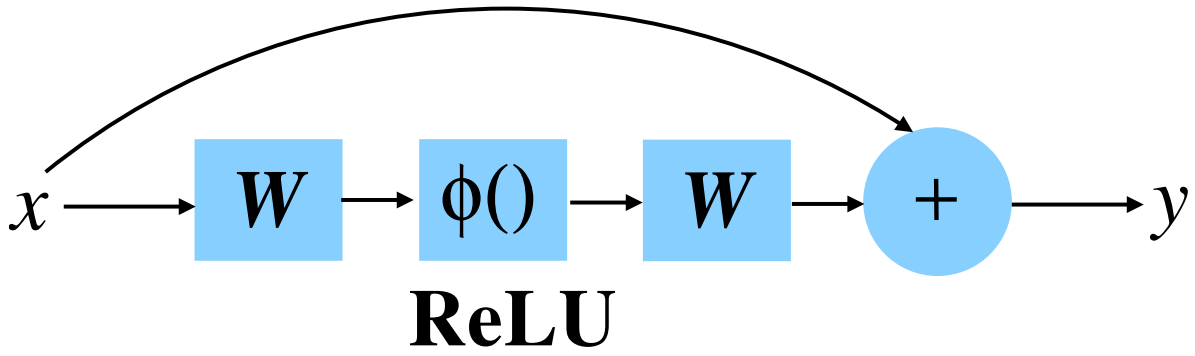
- Related activation function noisy ReLU:

$$\phi(z_i) = \max(0, z_i + \epsilon); \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

  - efficient, no exponential/division, rapid convergence in training
- Leaky ReLU also possible

$$\phi(z_i) = \begin{cases} z_i; & z_i \geq 0; \\ \alpha z_i & z_i < 0 \end{cases}$$

Activation Functions - Residual Networks [11]



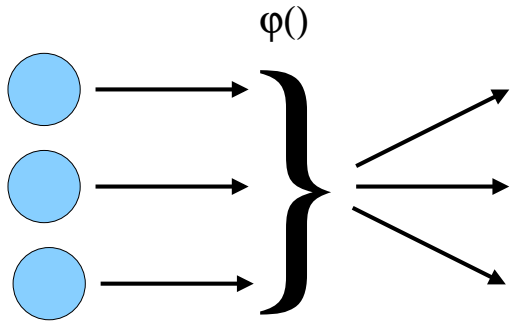
- Modify layer to model the residual

$$\boldsymbol{y}(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$$

  - allows deeper networks to be built
  - deep residual learning
- Links to highway connections

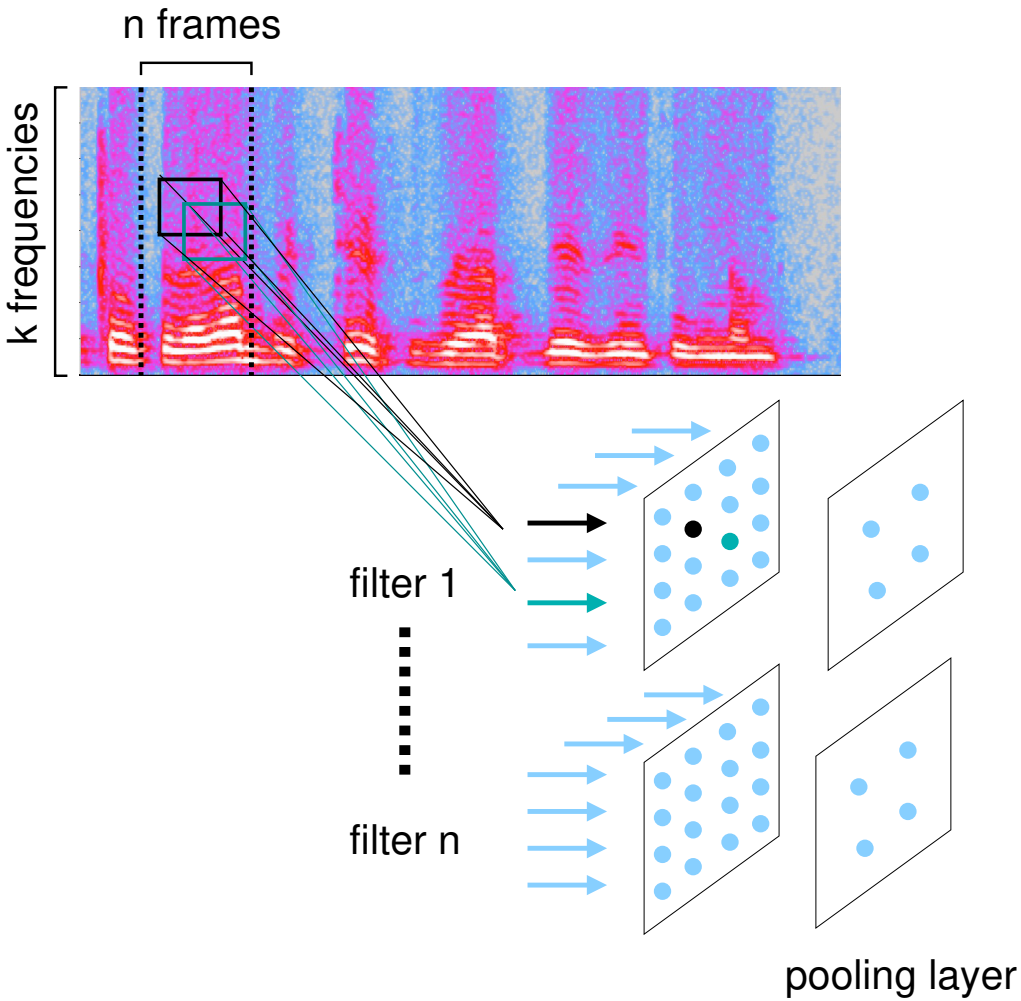
Pooling/Max-Out Functions [15, 26]

- Possible to pool the output of a set of node
  - reduces the number of weights to connect layers together



- A range of functions have been examined
  - maxout $\phi(y_1, y_2, y_3) = \max(y_1, y_2, y_3)$
  - soft-maxout $\phi(y_1, y_2, y_3) = \log(\sum_{i=1}^{3} \exp(y_i))$
  - p-norm $\phi(y_1, y_2, y_3) = (\sum_{i=1}^{3} |y_i|)^{1/p}$
- Has also been applied for unsupervised adaptation

Convolutional Neural Networks [14, 1]



Convolutional Neural Networks

- Various parameters control the form of the CNN:
  - number (depth): how many filters to use
  - receptive field (filter size): height/width/depth ($h \times w \times d$)
  - stride: how far filter moves in the convolution
  - dilation: "gaps" between filter elements
  - zero-padding: do you pad the edges of the "image" with zeroes
- Filter output can be stacked to yield depth for next layer
- To illustrate the impact consider 1-dimensional case - default:
  - zero-padding, stride=1, dilation=0

sequence: $1, 2, 3, 4, 5, 6$ filter: $1, 1, 1$

default: $3, 6, 9, 12, 15, 11$ no padding: $6, 9, 12, 15$

dilation=1: $4, 6, 9, 12, 8, 10$ stride=2: $3, 9, 15$

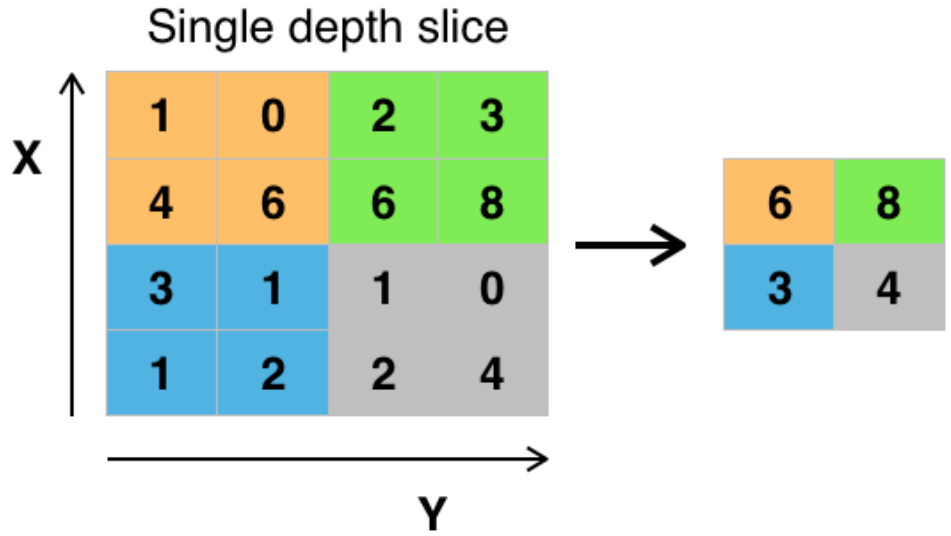
Convolutional Neural Network (cont)

- For a 2-dimensional image the convolution can be written as

$$\phi(z_{ij}) = \phi\left(\sum_{kl} w_{kl} x_{(i-k)(j-l)}\right)$$

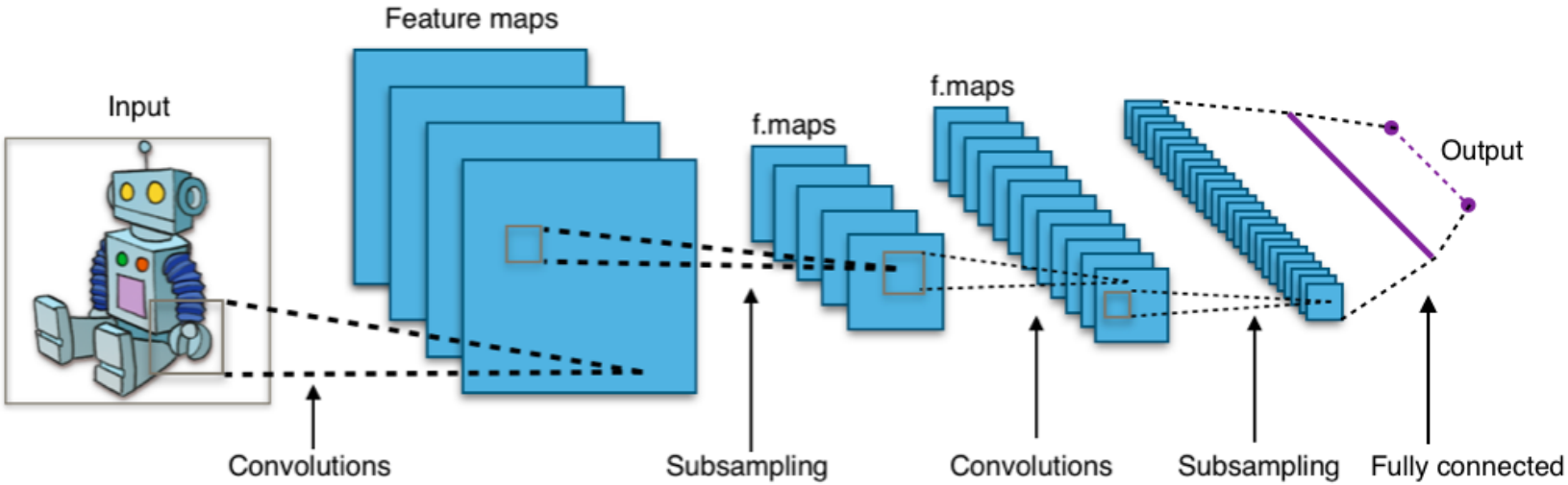
  - x_{ij} is the image value at point i, j
  - w_{kl} is the weight at point k, l
  - ϕ is the non-linear activation function
- For a 5×5 receptive field (no dilation)
  - $k \in \{-2, -1, 0, 1, 2\}$
  - $l \in \{-2, -1, 0, 1, 2\}$
- Stride determines how i and j vary as we move over the image

CNN Max-Pooling (Subsampling)



- Max-Pooling over a 2×2 filter on 4×4 "image"
  - stride of 2 - yields output of 2×2
- Possible to also operate with a stride of 1 overlapping pooling

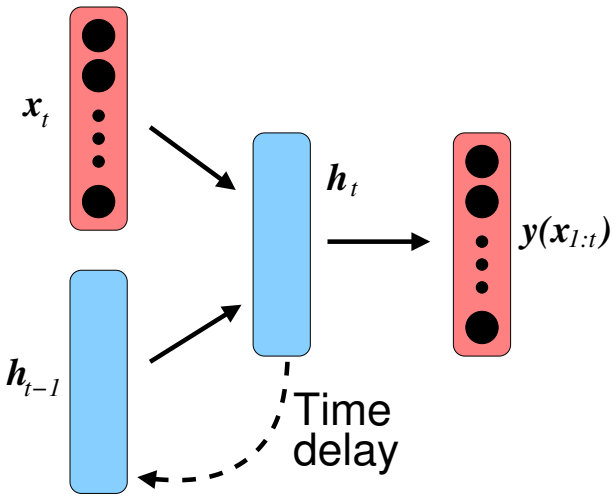
Simple CNN Example



- Simple five layer classifier for images
  - two convolutional layers each followed by
  - pooling layers (two)
  - with a fully connected network and softmax activation function

Recurrent Neural Networks [21, 19]

- Consider a causal sequence of observations $\boldsymbol{x}_{1:t} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t\}$



- Introduce recurrent units

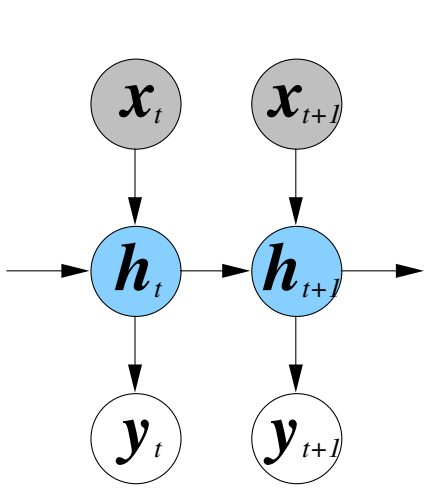
$$\begin{aligned} \boldsymbol{h}_t &= \mathbf{f}^{\mathrm{h}}\left(\mathbf{W}_{\mathrm{h}}^{\mathrm{f}}\boldsymbol{x}_t + \mathbf{W}_{\mathrm{h}}^{\mathrm{r}}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{h}}\right) \\ \boldsymbol{y}(\boldsymbol{x}_{1:t}) &= \mathbf{f}^{\mathrm{f}}\left(\mathbf{W}_{\mathrm{y}}\boldsymbol{h}_t + \boldsymbol{b}_{\mathrm{y}}\right) \end{aligned}$$

  - $\boldsymbol{h}_t$ history vector at time t
- Two history weight matrices
  - $\mathbf{W}_{\mathrm{h}}^{\mathrm{f}}$ forward, $\mathbf{W}_{\mathrm{h}}^{\mathrm{r}}$ recursion
- Uses approximation to model history of observations

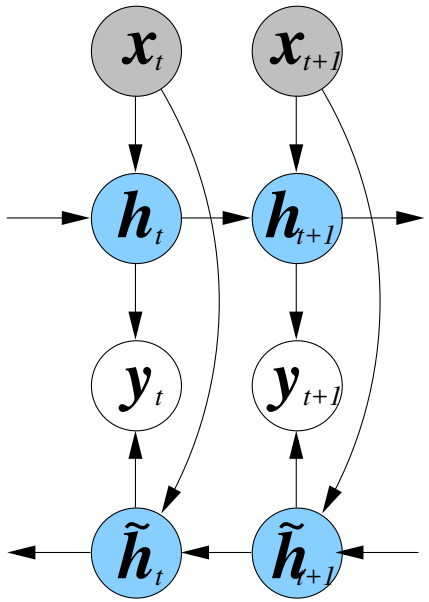
$$\mathcal{F}(\boldsymbol{x}_{1:t}) = \mathcal{F}(\boldsymbol{x}_t, \boldsymbol{x}_{1:t-1}) \approx \mathcal{F}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \approx \mathcal{F}(\boldsymbol{h}_t) = \boldsymbol{y}(\boldsymbol{x}_{1:t})$$

  - network has (causal) memory encoded in history vector ($\boldsymbol{h}_t$)

RNN Variant: Bi-Directional RNN [23]



RNN

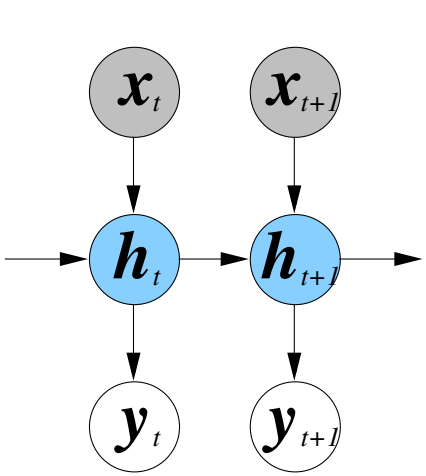


Bi-Directional RNN

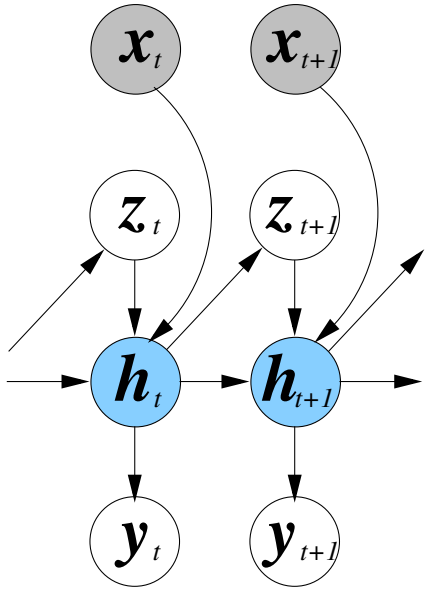
- Bi-directional: use complete observation sequence - non-causal

$$\mathcal{F}_t(\boldsymbol{x}_{1:T}) = \mathcal{F}(\boldsymbol{x}_{1:t}, \boldsymbol{x}_{t:T}) \approx \mathcal{F}(\boldsymbol{h}_t, \tilde{\boldsymbol{h}}_t) = \boldsymbol{y}_t(\boldsymbol{x}_{1:T})$$

Latent Variable (Variational) RNN (reference) [7]



RNN



Variational RNN

- Variational: introduce latent variable sequence $\boldsymbol{z}_{1:T}$

$$p(\boldsymbol{y}_t|\boldsymbol{x}_{1:t}) \approx \int p(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{z}_t, \boldsymbol{h}_{t-1}) p(\boldsymbol{z}_t|\boldsymbol{h}_{t-1}) d\boldsymbol{z}_t$$

$$\approx \int p(\boldsymbol{y}_t|\boldsymbol{h}_t) p(\boldsymbol{z}_t|\boldsymbol{h}_{t-1}) d\boldsymbol{z}_t$$

  - $\boldsymbol{z}_t$ a function of complete history (complicates training)

Network Gating

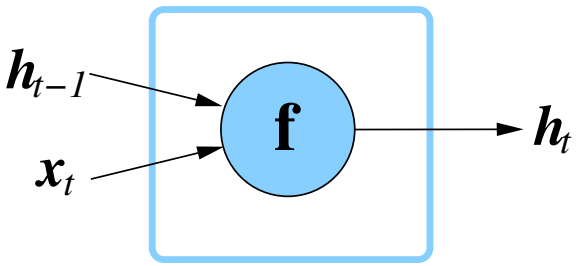
- A flexible extension to activation function is gating
  - standard form is ($\sigma()$ sigmoid activation function)

$$\boldsymbol{i} = \sigma(\mathbf{W}^{\mathrm{f}} \boldsymbol{x}_t + \mathbf{W}^{\mathrm{r}} \boldsymbol{h}_{t-1} + \boldsymbol{b})$$

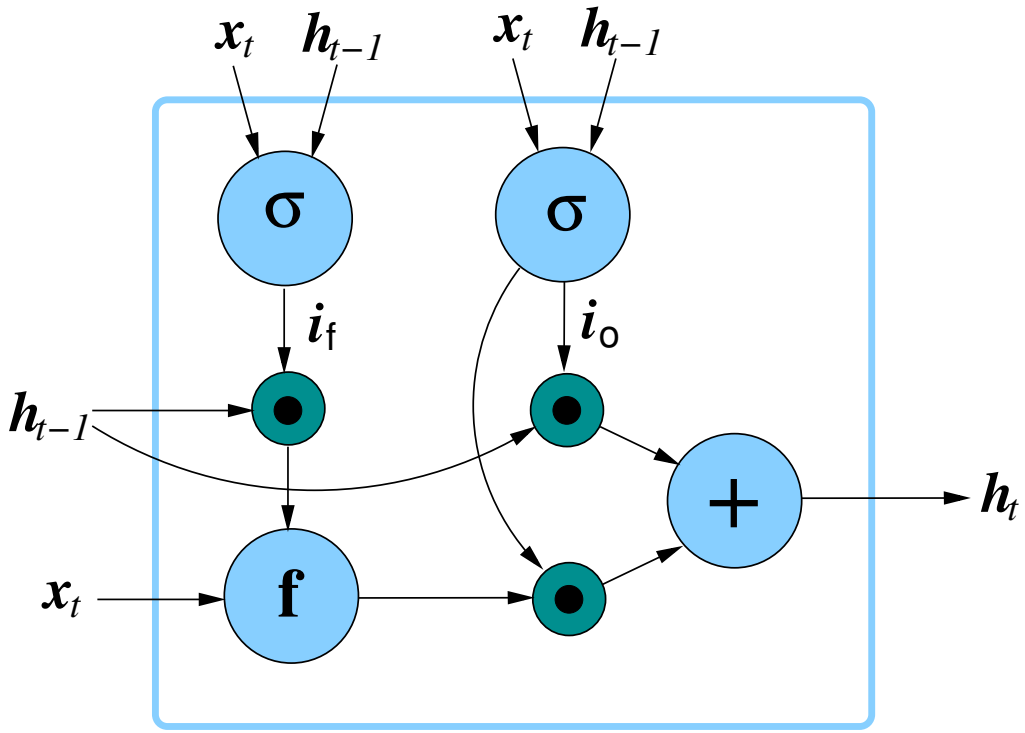
  - vector acts a probabilistic gate on network values

- Gating can be applied at various levels
  - features: impact of input/output features on nodes
  - time: memory of the network
  - layer: influence of a layer's activation function

Gated Recurrent Unit [6]



Recurrent unit



Gated Recurrent Unit

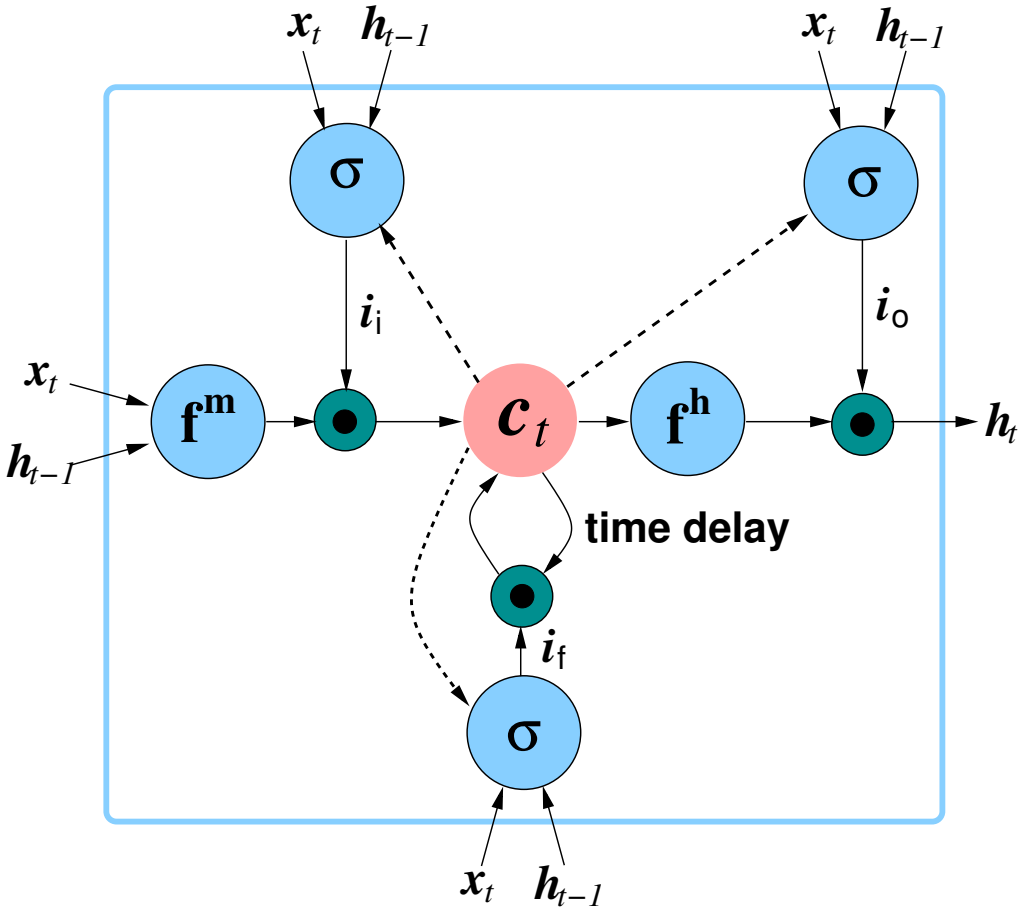
Gated Recurrent Unit [6]

- Gated Recurrent Unit (GRU) introduces
  - forget gate ($\boldsymbol{i}_{\mathrm{f}}$): gating over time
  - output gate ($\boldsymbol{i}_{\mathrm{o}}$): gating over features (and time)
- Relationships (standard configuration - there are variants)

$$
\begin{aligned}
\boldsymbol{i}_{\mathrm{f}} &= \sigma(\mathbf{W}_{\mathrm{f}}^{\mathrm{f}}\boldsymbol{x}_t + \mathbf{W}_{\mathrm{f}}^{\mathrm{r}}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{f}}) \\
\boldsymbol{i}_{\mathrm{o}} &= \sigma(\mathbf{W}_{\mathrm{o}}^{\mathrm{f}}\boldsymbol{x}_t + \mathbf{W}_{\mathrm{o}}^{\mathrm{r}}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{o}}) \\
\boldsymbol{y}_t &= \mathrm{f}(\mathbf{W}_{\mathrm{y}}^{\mathrm{f}}\boldsymbol{x}_t + \mathbf{W}_{\mathrm{y}}^{\mathrm{r}}(\boldsymbol{i}_{\mathrm{f}} \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_{\mathrm{y}}) \\
\boldsymbol{h}_t &= \boldsymbol{i}_{\mathrm{o}} \odot \boldsymbol{h}_{t-1} + (\mathbf{1} - \boldsymbol{i}_{\mathrm{o}}) \odot \boldsymbol{y}_t
\end{aligned}
$$

  - $\odot$ represents element-wise multiplication between vectors

Long-Short Term Memory Networks (reference) [13, 10]



Long-Short Term Memory Networks (reference)

- The operations can be written as (peephole config):
  - Forget gate ($\boldsymbol{i}_{\mathtt{f}}$), Input gate ($\boldsymbol{i}_{\mathtt{i}}$), Output gate ($\boldsymbol{i}_{\mathtt{o}}$)

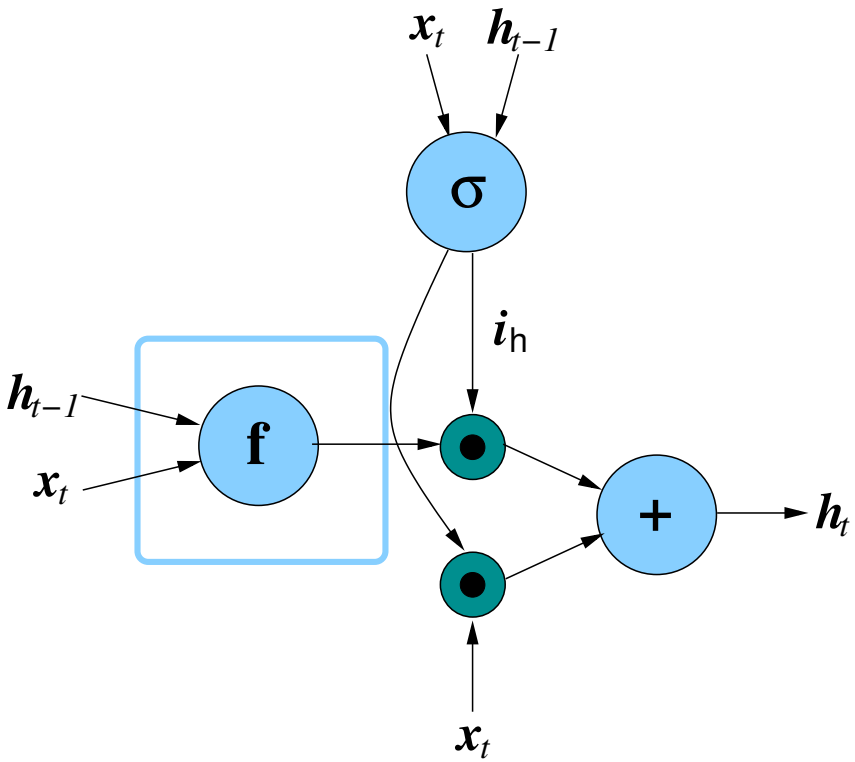
$$
\begin{aligned}
\boldsymbol{i}_{\mathtt{f}} &= \sigma(\mathbf{W}^{\mathtt{f}}_{\mathtt{f}}\boldsymbol{x}_t + \mathbf{W}^{\mathtt{r}}_{\mathtt{f}}\boldsymbol{h}_{t-1} + \mathbf{W}^{\mathtt{m}}_{\mathtt{f}}\boldsymbol{c}_{t-1} + \boldsymbol{b}_{\mathtt{f}}) \\
\boldsymbol{i}_{\mathtt{i}} &= \sigma(\mathbf{W}^{\mathtt{f}}_{\mathtt{i}}\boldsymbol{x}_t + \mathbf{W}^{\mathtt{r}}_{\mathtt{i}}\boldsymbol{h}_{t-1} + \mathbf{W}^{\mathtt{m}}_{\mathtt{i}}\boldsymbol{c}_{t-1} + \boldsymbol{b}_{\mathtt{i}}) \\
\boldsymbol{i}_{\mathtt{o}} &= \sigma(\mathbf{W}^{\mathtt{f}}_{\mathtt{o}}\boldsymbol{x}_t + \mathbf{W}^{\mathtt{r}}_{\mathtt{o}}\boldsymbol{h}_{t-1} + \mathbf{W}^{\mathtt{m}}_{\mathtt{o}}\boldsymbol{c}_{t} + \boldsymbol{b}_{\mathtt{o}})
\end{aligned}
$$

  - Memory Cell, history vector and gates are related by

$$
\begin{aligned}
\boldsymbol{c}_t &= \boldsymbol{i}_{\mathtt{f}} \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_{\mathtt{i}} \odot \mathbf{f}^{\mathtt{m}}(\mathbf{W}^{\mathtt{f}}_{\mathtt{c}}\boldsymbol{x}_t + \mathbf{W}^{\mathtt{r}}_{\mathtt{c}}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathtt{c}}) \\
\boldsymbol{h}_t &= \boldsymbol{i}_{\mathtt{o}} \odot \mathbf{f}^{\mathtt{h}}(\boldsymbol{c}_t)
\end{aligned}
$$

  - more complicated than GRU (three gates, memory cell)
  - memory cell weight matrices ($\mathbf{W}^{\mathtt{m}}_{\mathtt{f}}, \mathbf{W}^{\mathtt{m}}_{\mathtt{i}}, \mathbf{W}^{\mathtt{m}}_{\mathtt{o}}$) diagonal
  - can allow explicit analysis of individual cell elements

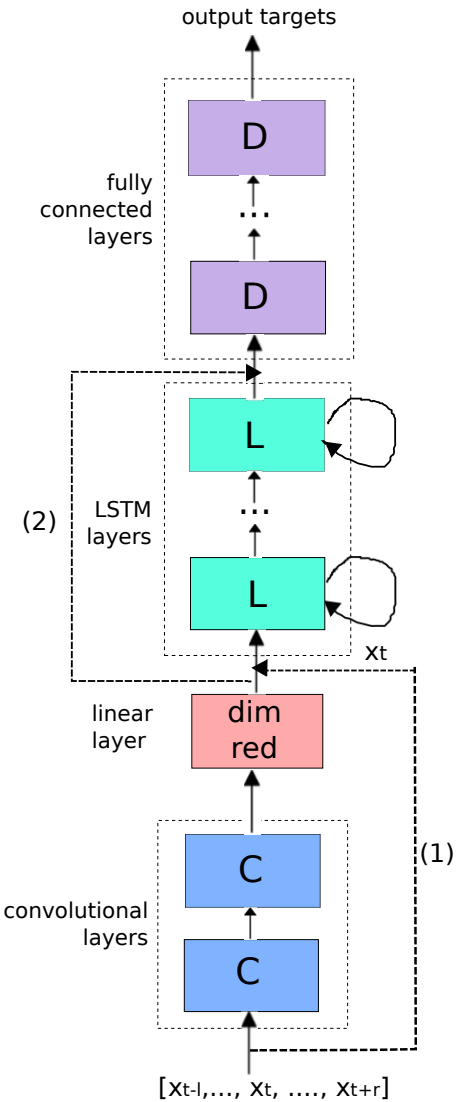
Highway Connections [25]



- Gate the output of the node (example from recurrent unit)
  - combine with output from previous layer ($\boldsymbol{x}_t$)

$$\begin{aligned} \boldsymbol{i}_{\mathtt{h}} &= \boldsymbol{\sigma}(\mathbf{W}^{\mathtt{f}}_{\mathtt{l}}\boldsymbol{x}_t + \mathbf{W}^{\mathtt{r}}_{\mathtt{l}}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathtt{l}}) \\ \boldsymbol{h}_t &= \boldsymbol{i}_{\mathtt{h}} \odot \mathbf{f}(\mathbf{W}^{\mathtt{f}}_{\mathtt{h}}\boldsymbol{x}_t + \mathbf{W}^{\mathtt{r}}_{\mathtt{h}}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathtt{h}}) + (\mathbf{1} - \boldsymbol{i}_{\mathtt{h}}) \odot \boldsymbol{x}_t \end{aligned}$$

Example Deep Architecture: ASR (reference) [22]



- Example Architecture from Google (2015)
  - C: CNN layer (with pooling)
  - L: LSTM layer
  - D: fully connected layer
- Two multiple layer "skips"
  - (1) connects input to LSTM input
  - (2) connects CNN output to DNN input
- Additional linear projection layer
  - reduces dimensionality
  - and number of network parameters!

Network Training and Error Back Propagation

Training Data: Classification

- Supervised training data comprises
  - $\boldsymbol{x}_i$: d-dimensional training observation
  - y_i: class label, K possible (discrete) classes
- Encode class labels as 1-of-K ("one-hot") coding: $y_i \rightarrow \boldsymbol{t}_i$
  - $\boldsymbol{t}_i$ is the K-dimensional target vector for $\boldsymbol{x}_i$
  - zero other than element associated with class-label y_i
- Consider a network with parameters $\boldsymbol{\theta}$ and training examples:

$$\{\{\boldsymbol{x}_1, \boldsymbol{t}_1\} \ldots, \{\boldsymbol{x}_n, \boldsymbol{t}_n\}\}$$

  - need "distance" from target $\boldsymbol{t}_i$ to network output $\boldsymbol{y}(\boldsymbol{x}_i)$

Training Criteria

- Least squares error: one of the most common training criteria.

$$E(\boldsymbol{\theta}) = \frac{1}{2}\sum_{p=1}^{n} \|\boldsymbol{y}(\boldsymbol{x}_p) - \boldsymbol{t}_p)\|^2 = \frac{1}{2}\sum_{p=1}^{n}\sum_{i=1}^{K} (y_i(\boldsymbol{x}_p) - t_{pi})^2$$

- Cross-Entropy: note non-zero minimum (entropy of targets)

$$E(\boldsymbol{\theta}) = -\sum_{p=1}^{n}\sum_{i=1}^{K} t_{pi} \log(y_i(\boldsymbol{x}_p))$$

- Cross-Entropy for two classes: single binary target

$$E(\boldsymbol{\theta}) = -\sum_{p=1}^{n} (t_p \log(y(\boldsymbol{x}_p)) + (1 - t_p) \log(1 - y(\boldsymbol{x}_p)))$$

Training Data: Regression

- Supervised training data comprises
  - $\boldsymbol{x}_i$: d-dimensional training observation
  - $\boldsymbol{y}_i$: K-dimensional (continuous) output vector
- Consider a network with parameters $\boldsymbol{\theta}$ and training examples:

$$\{\{\boldsymbol{x}_1, \boldsymbol{y}_1\} \ldots, \{\boldsymbol{x}_n, \boldsymbol{y}_n\}\}$$

  - need "distance" from target $\boldsymbol{y}_i$ to network output $\boldsymbol{y}(\boldsymbol{x}_i)$
- Least squares commonly used criterion

$$E(\boldsymbol{\theta}) = \frac{1}{2} \sum_{p=1}^{n} (\boldsymbol{y}(\boldsymbol{x}_p) - \boldsymbol{y}_p)'(\boldsymbol{y}(\boldsymbol{x}_p) - \boldsymbol{y}_p)$$

  - $\boldsymbol{y}(\boldsymbol{x}_i)$ may be viewed as the mean of the prediction

Maximum Likelihood - Regression Criterion

- Generalise least squares (LS) to maximum likelihood (ML)

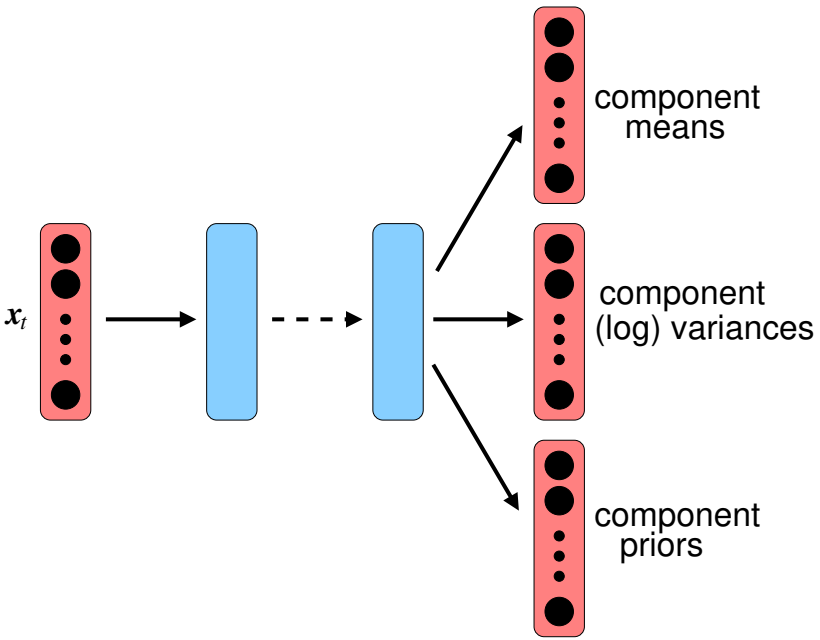
$$E(\boldsymbol{\theta}) = \sum_{p=1}^{n} \log(p(\boldsymbol{y}_p|\boldsymbol{x}_p, \boldsymbol{\theta}))$$

  - LS is ML with a single Gaussian, identity covariance matrix
- Criterion appropriate to deep learning for generative models
- Output-layer activation function to ensure valid distribution
  - consider the case of the variance $\sigma > 0$
  - apply an exponential activation function for variances

$$\exp(y_i(\boldsymbol{x})) > 0$$

  - for means just use a linear activation function

Mixture Density Neural Networks [3, 28]



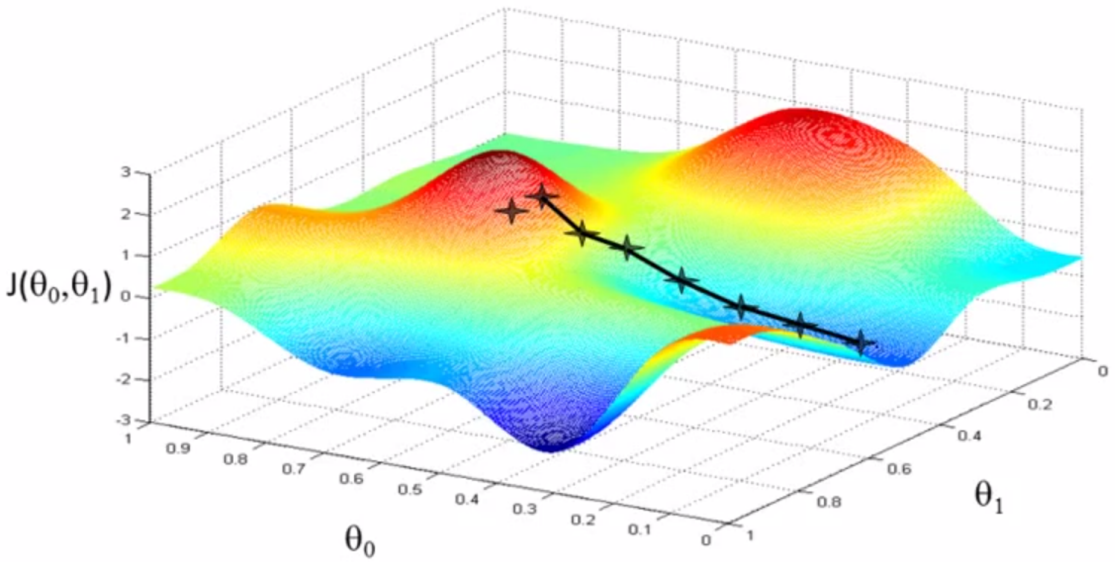
- Predict a mixture of M Gaussians
  - $\mathcal{F}_m^{(c)}(\boldsymbol{x}_t)$: prior prediction
  - $\mathcal{F}_m^{(\mu)}(\boldsymbol{x}_t)$: mean prediction
  - $\mathcal{F}_m^{(\sigma)}(\boldsymbol{x}_t)$: variance prediction
- For component m, output

$$\boldsymbol{y}_m(\boldsymbol{x}_t) = \begin{bmatrix} \mathcal{F}_m^{(c)}(\boldsymbol{x}_t) \\ \mathcal{F}_m^{(\mu)}(\boldsymbol{x}_t) \\ \mathcal{F}_m^{(\sigma)}(\boldsymbol{x}_t) \end{bmatrix}$$

- Optimise using maximum likelihood where

$$p(\boldsymbol{y}_t|\boldsymbol{x}_t) = \sum_{m=1}^{M} \mathcal{F}_m^{(c)}(\boldsymbol{x}_t)\mathcal{N}(\boldsymbol{y}_t; \mathcal{F}_m^{(\mu)}(\boldsymbol{x}_t), \mathcal{F}_m^{(\sigma)}(\boldsymbol{x}_t))$$

Gradient Descent [20]



- If there is no closed-form solution - use gradient descent

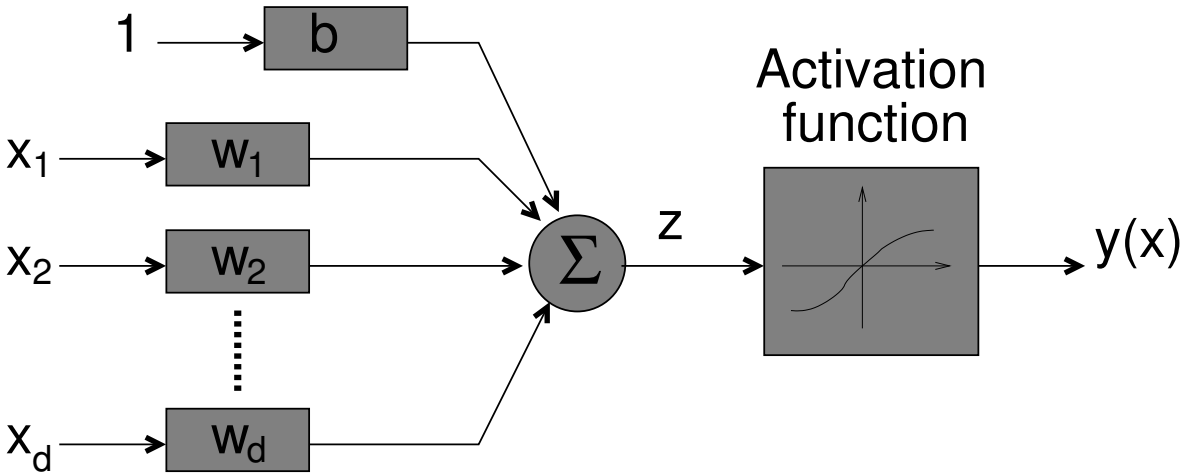
$$\boldsymbol{\theta}[\tau+1] = \boldsymbol{\theta}[\tau] - \boldsymbol{\Delta\theta}[\tau] = \boldsymbol{\theta}[\tau] - \eta \left.\frac{\partial E}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}[\tau]}$$

  - how to get the gradient for all model parameters
  - how to avoid local minima
  - need to consider how to set η

Network Training

- Networks usually have a large number of hidden layers (L)
  - enables network to model highly non-linear, complex, mappings
  - complicates the training process of the network parameters
- Network parameters are (usually) weights for each layer
  - merging bias vector for each layer into weight matrix

$$\boldsymbol{\theta} = \left\{ \boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L+1)} \right\}$$



- Initially just consider a single layer perceptron

Single Layer Perceptron Training

- Take the example of least squares error cost function

$$E(\boldsymbol{\theta}) = \frac{1}{2}\sum_{p=1}^{n} \|y(\boldsymbol{x}_p) - t_p)\|^2 = \sum_{p=1}^{n} E^{(p)}(\boldsymbol{\theta})$$

- Use chain rule to compute derivatives through network

$$\frac{\partial E(\boldsymbol{\theta})}{\partial w_i} = \left(\frac{\partial E(\boldsymbol{\theta})}{\partial y(\boldsymbol{x})}\right)\left(\frac{\partial y(\boldsymbol{x})}{\partial z}\right)\left(\frac{\partial z}{\partial w_i}\right)$$

  - change of error function with network output (cost function)
  - change of network output with z (activation function)
  - change of z with network weight (parameter to estimate)

- For a sigmoid activation function this yields

$$\frac{\partial E^{(p)}(\boldsymbol{\theta})}{\partial w_i} = (y(\boldsymbol{x}_p) - t_p)y(\boldsymbol{x}_p)(1 - y(\boldsymbol{x}_p))x_{pi}$$

Error Back Propagation

- Simple concept can be extended to multiple (hidden) layers
  - output from layer $k-1$ ($\boldsymbol{y}^{(k-1)}$) is the input to layer k ($\boldsymbol{x}^{(k)}$)
- $L+1$ layer network - use backward recursion (see notes)
  - model parameters $\boldsymbol{\theta} = \left\{ \boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L+1)} \right\}$
  - final output $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{y}^{(L+1)}$

$$\boldsymbol{\delta}^{(k)} = \boldsymbol{\Lambda}^{(k)} \boldsymbol{W}^{(k+1)\prime} \boldsymbol{\delta}^{(k+1)}; \quad \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{W}^{(k)}} = \boldsymbol{\delta}^{(k)} \boldsymbol{x}^{(k)\prime}$$

- $\boldsymbol{\Lambda}^{(k)} = \frac{\partial \boldsymbol{y}^{(k}}{\partial \boldsymbol{z}^{(k)}}$: layer k activation derivative matrix
- $\boldsymbol{W}^{(k+1)}$: weight matrix for layer $k+1$
- $\boldsymbol{\delta}^{(k+1)} = \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{z}^{(k)}}$: error vector for layer $k+1$

Error Back Propagation (cont)

- The process to get the derivative involves:

1. For input vector $\boldsymbol{x}_p$ propagate forward: yields $(1 \leq k \leq L)$
   - $\boldsymbol{y}^{(k)}$ the output value for each node of layer all layers
   - $\boldsymbol{z}^{(k)}$ the input value to the non-linearity for layer k
2. Compute $\left.\frac{\partial E(\boldsymbol{\theta}}{\partial \boldsymbol{y}(\boldsymbol{x})}\right|_{\boldsymbol{\theta}_{[\tau]}}$ (the gradient at the output layer).
3. From the output gradient propagate backwards: yields
   - $\boldsymbol{\delta}^{(k)}$ the error vector for each layer
   - $\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{W}^{(k)}}$: the (desired) derivative for layer k weights

Optimisation

Batch/On-Line Gradient Descent

- The default gradient is computed over all samples
  - for large data sets very slow - each update
- Modify to batch update - just use a subset of data, $\tilde{\mathcal{D}}$,

$$E(\boldsymbol{\theta}) = -\sum_{p \in \tilde{\mathcal{D}}} \sum_{i=1}^{K} t_{pi} \log(y_i(\boldsymbol{x}_p))$$

- How to select the subset, $\tilde{\mathcal{D}}$?
  - small subset "poor" estimate of true gradient
  - large subset each parameter update is expensive
- One extreme is to update after each sample
  - $\tilde{\mathcal{D}}$ comprises a single sample in order
  - "noisy" gradient estimate for updates

Stochastic Gradient Descent (SGD)

- Two modifications to the baseline approaches
  1. Randomise the order of the data presented for training
     - important for structured data
  2. Introduce mini-batch updates
     - $\tilde{\mathcal{D}}$ is a (random) subset of the training data
     - better estimate of the gradient at each update
     - but reduces number of iterations

- Mini-batch updates are (almost) always used
  - make use of parallel processing (GPUs) for efficiency
- Research of parallel versions of SGD on-going

Momentum

- A number of issues for gradient descent including:
  - stops at local maxima
  - handling "ravines"
- Momentum aims to address this - parameter change becomes:

$$\boldsymbol{\Delta\theta}[\tau] = \eta \left.\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}[\tau]} + \alpha \boldsymbol{\Delta\theta}[\tau - 1]$$

  - smooths parameter changes over iterations
  - introduces an additional tunable parameter
- For simplicity introduce compact notation

$$\left.\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}[\tau]} = \nabla(E(\boldsymbol{\theta}[\tau]))$$

Adaptive Learning Rates

- Speed of convergence depends on η
  - too large: updates may diverge rather than converge
  - too small: very slow convergence (impractical)
- The standard expression has a fixed learning rate
  - can we have learning rate change with iteration

$$\boldsymbol{\theta}[\tau+1] = \boldsymbol{\theta}[\tau] - \boldsymbol{\Delta\theta}[\tau] = \boldsymbol{\theta}[\tau] - \eta[\tau]\nabla(E(\boldsymbol{\theta}[\tau]))$$

  - how to set $\eta[\tau]$ (or generally parameter update $\boldsymbol{\Delta\theta}[\tau]$)?
- One very simple approach

$$\eta[\tau+1] = \begin{cases} 1.1\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) < E(\boldsymbol{\theta}[\tau-1]) \\ 0.5\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) > E(\boldsymbol{\theta}[\tau-1]) \end{cases}$$

  - increase learning rate when going in "correct direction"

Gradient Descent Refinements (reference)

- Nesterov: concept of gradient at next iteration

$$\boldsymbol{\Delta\theta}[\tau] = \eta\nabla(E(\boldsymbol{\theta}[\tau] - \alpha\boldsymbol{\Delta\theta}[\tau-1])) + \alpha\boldsymbol{\Delta\theta}[\tau-1]$$

- AdaGrad: dimension specific learning rates (ϵ floor parameter)

$$\boldsymbol{\Delta\theta}[\tau] = \eta\beta_t \odot \left.\frac{\partial E(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right|_{\boldsymbol{\theta}_{[\tau]}}; \quad \beta_{ti} = \frac{1}{\sqrt{\epsilon + \sum_{t=1}^{\tau}\nabla_i(E(\boldsymbol{\theta}[t]))^2}}$$

  - ϵ is a smoothing term to avoid division by zero

- Adam: Adaptive Moment Estimation: use dimension moments

$$\Delta\theta_i[\tau] = \frac{\eta}{\sqrt{\sigma^2_{\tau i}+\epsilon}}\mu_{\tau i}; \qquad \begin{aligned}\mu_{\tau i} &= \alpha_1\mu_{(\tau-1)i} + (1-\alpha_1)\nabla_i(E(\boldsymbol{\theta}[\tau])) \\ \sigma^2_{\tau i} &= \alpha_2\sigma^2_{(\tau-1)i} + (1-\alpha_2)\nabla_i(E(\boldsymbol{\theta}[\tau]))^2\end{aligned}$$

  - additional normalisation applied to $\mu_{\tau i}$ and $\sigma^2_{\tau i}$ to offset initialisation bias

Second-Order Approximations

- Gradient descent makes use of first-order derivatives of
  - what about higher order derivatives? Consider

$$E(\boldsymbol{\theta}) = E(\boldsymbol{\theta}[\tau]) + (\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])'\mathbf{g} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])'\mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau]) + \mathcal{O}(\boldsymbol{\theta}^3)$$

where

$$\mathbf{g} = \nabla E(\boldsymbol{\theta}[\tau]); \quad (\mathbf{H})_{ij} = h_{ij} = \left.\frac{\partial^2 E(\boldsymbol{\theta})}{\partial\theta_i \partial\theta_j}\right|_{\boldsymbol{\theta}[\tau]}$$

- Ignoring higher order terms and equating to zero

$$\nabla E(\boldsymbol{\theta}) = \mathbf{g} + \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}[\tau])$$

Equating to zero (check minimum!) - $\mathbf{H}^{-1}\mathbf{g}$ Newton direction

$$\boldsymbol{\theta}[\tau + 1] = \boldsymbol{\theta}[\tau] - \mathbf{H}^{-1}\mathbf{g}; \quad \boldsymbol{\Delta\theta}[\tau] = \mathbf{H}^{-1}\mathbf{g}$$

Issues with Second-Order Approaches

1. The evaluation of the Hessian may be computationally expensive as $\mathcal{O}(N^2)$ parameters must be accumulated for each of the n training samples.
2. The Hessian must be inverted to find the direction, $\mathcal{O}(N^3)$. This gets very expensive as N gets large.
3. The direction given need not head towards a minimum - it could head towards a maximum or saddle point. This occurs if the Hessian is not positive-definite i.e.

$$\mathbf{v}'\mathbf{H}\mathbf{v} > 0$$

for all $\mathbf{v}$. The Hessian may be made positive definite using

$$\tilde{\mathbf{H}} = \mathbf{H} + \lambda\mathbf{I}$$

If λ is large enough then $\tilde{\mathbf{H}}$ is positive definite.

4. If the surface is highly non-quadratic the step sizes may be too large and the optimisation becomes unstable.

QuickProp

- Interesting making use of the error curvature, assumptions:
  - error surface is quadratic in nature
  - weight gradients treated independently (diagonal Hessian)
- Using these assumptions

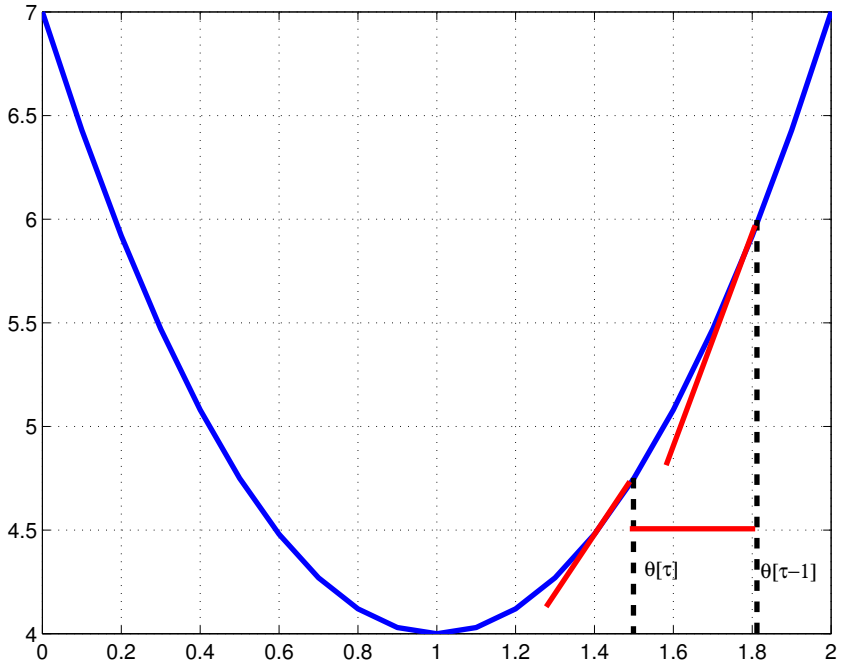
$$\begin{aligned} E(\theta) &\approx E(\theta[\tau]) + b(\theta - \theta[\tau]) + a(\theta - \theta[\tau])^2 \\ \frac{\partial E(\theta)}{\partial \theta} &\approx b + 2a(\theta - \theta[\tau]) \end{aligned}$$

- To find a and b make use of:
  - update step, $\Delta\theta[\tau-1]$, and gradient, $g[\tau-1]$, iteration $\tau-1$
  - the gradient at iteration τ is $g[\tau]$
  - after new update $\Delta\theta[\tau]$ the gradient should be zero
- The following equalities are obtained

$$g[\tau-1] = b - 2a\Delta\theta[\tau-1], \quad 0 = b + 2a\Delta\theta[\tau], \quad g[\tau] = b$$

$$\rightarrow \Delta\theta[\tau] = \frac{g[\tau]}{g[\tau-1] - g[\tau]}\Delta\theta[\tau-1]$$

QuickProp (cont)



- The operation of quick-prop is illustrated above:
  - the assumed quadratic error surface is shown in blue
  - the statistics for quickprop are shown in red
- Parameter at minimum of quadratic approximation:
  $\theta[\tau + 1] = 1$

Regularisation

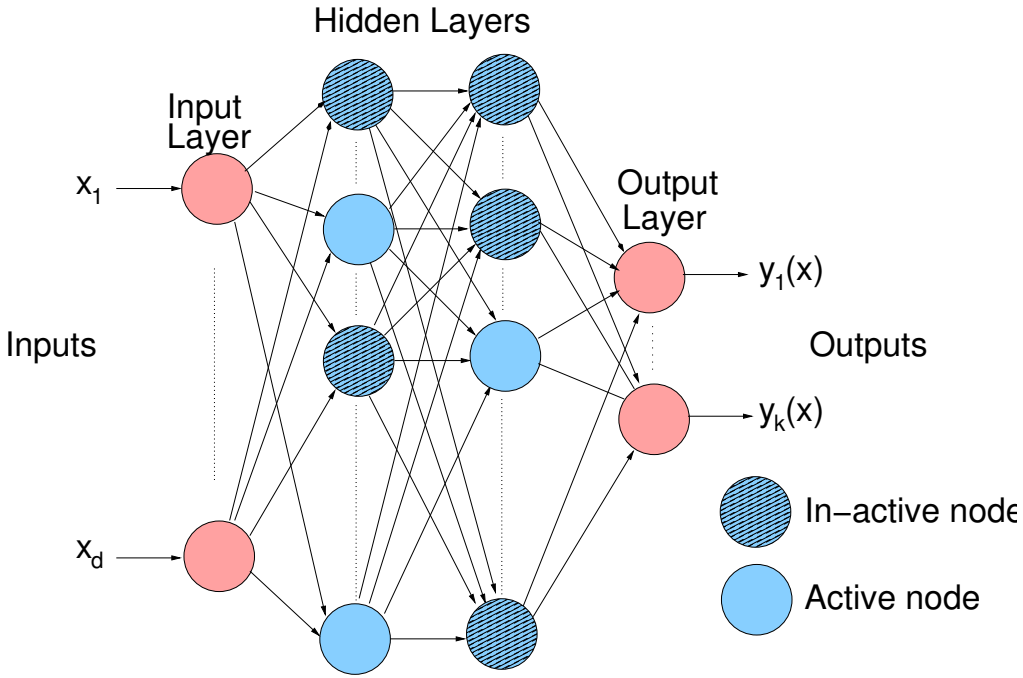
- A major issues with training networks is generalisation
- Simplest approach is early stopping
  - don't wait for convergence - just stop ...
- To address this forms of regularisation are used
  - one standard form is (N is the total number of weights):

$$\tilde{E}(\boldsymbol{\theta}) = E(\boldsymbol{\theta}) + \nu\Omega(\boldsymbol{\theta}); \quad \Omega(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{N} w_i^2$$

  - a zero "prior" is used for the model parameters
- Simple to include in gradient-descent optimisation

$$\nabla\tilde{E}(\boldsymbol{\theta}[\tau]) = \nabla E(\boldsymbol{\theta}[\tau]) + \nu\boldsymbol{w}[\tau]$$

Dropout [24]



- Dropout is simple way of improving generalisation
  1. randomly de-activate (say) 50% of the nodes in the network
  2. update the model parameters
- Prevents a single node specialising to a task

Network Initialisation: Data Pre-Processing

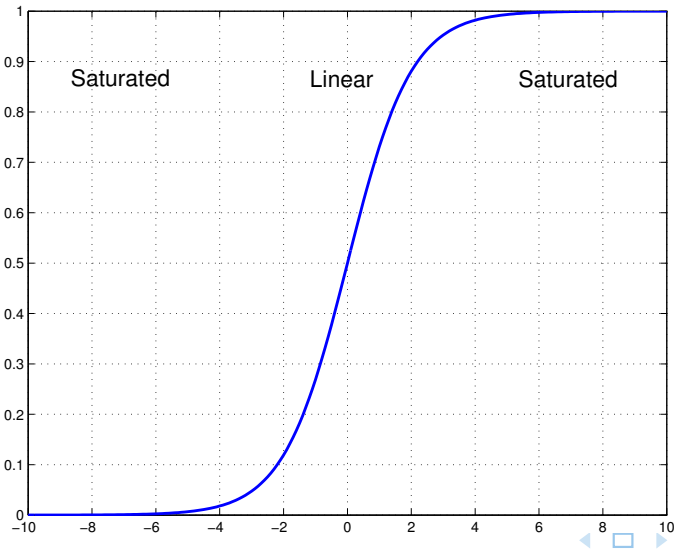
- As with standard classifiers two stage classification often used
  - features are designed by expert
  - current trend to remove two-stage process end-to-end
- Features may have different dynamic ranges
  - consider dimension 1:$-1000 \rightarrow 1000$ vs dimension 2:$-1 \rightarrow 1$
  - can influence "importance" of features at start of training
- Data whitening often employed

$$\tilde{x}_{pi} = \frac{x_{pi} - \mu_i}{\sigma_i}; \quad \mu_i = \frac{1}{n}\sum_{p=1}^{n} x_{pi} \quad \sigma_i^2 = \frac{1}{n}\sum_{p=1}^{n}(x_{pi} - \mu_i)^2$$

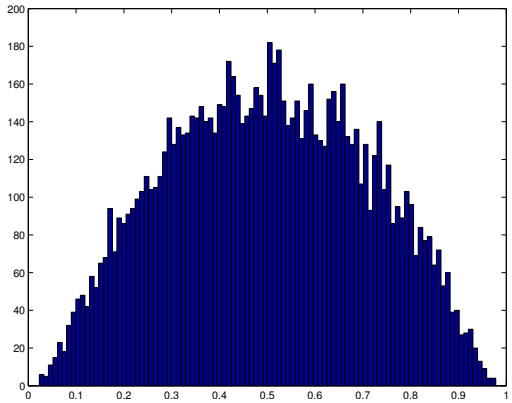
  - only influences initialisation (linear transform and bias)

Network Initialisation: Weight Parameters

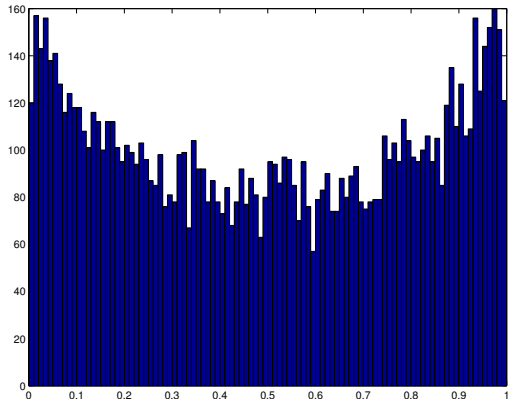
- A starting point (initialisation) for gradient descent is useful
  - one of the "old" concerns with deep networks was initialisation
  - recurrent neural networks are very deep!
- It is not possible to guarantee a good starting point, but
  - would like a parsimonious initialisation
- What about Gaussian random initialisation
  - consider zero mean distribution, scale the variance
  - sigmoid non-linearity



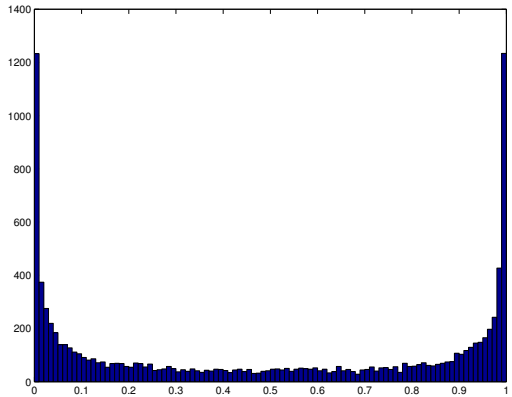
Gaussian Initialisation



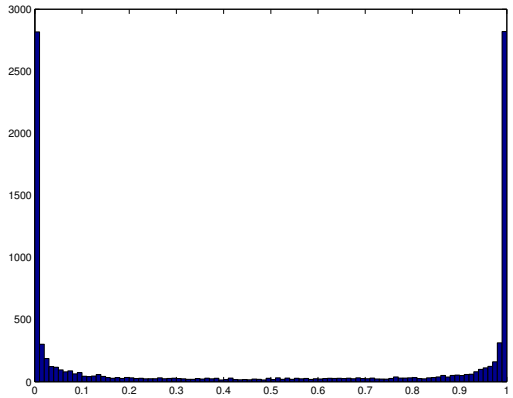
$\mathcal{N}(0, 1)$



$\mathcal{N}(0, 2)$



$\mathcal{N}(0, 4)$



$\mathcal{N}(0, 8)$

- Pass 1-dimensional data through a sigmoid

Exploding and Vanishing Gradients [8]

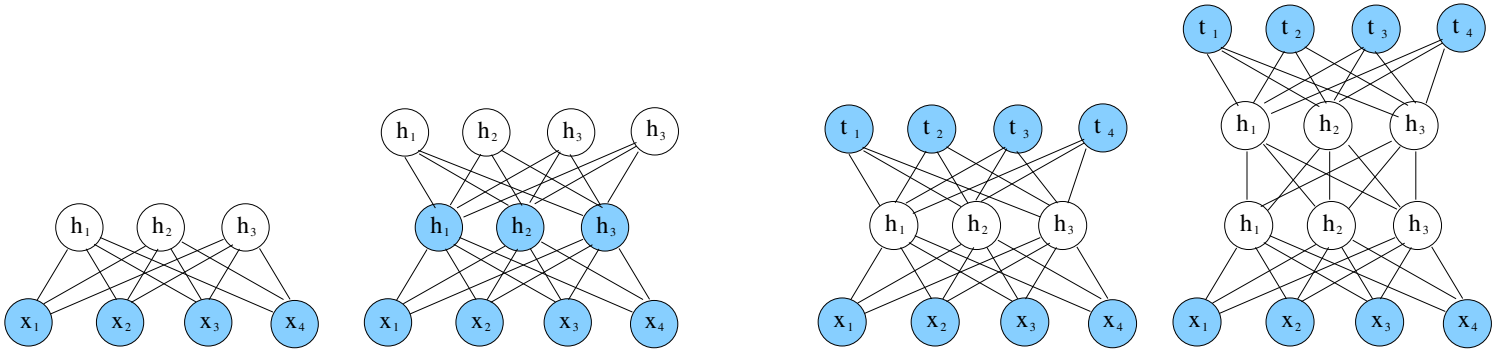
- Need to worry about the following gradient issues
  - Vanishing: derivatives go to zero - parameters not updated
  - Exploding: derivatives get very large - cause saturation
- Xavier Initialisation: simple scheme for initialising weights
  - linear activation functions $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$,
  - assuming all weights/observations independent
  - $\boldsymbol{x}$ n dimensional, zero mean, identity variance

$$\texttt{Var}(y_i) = \texttt{Var}(\boldsymbol{w}_i'\boldsymbol{x}) = n\texttt{Var}(w_{ij})\texttt{Var}(x_i)$$

- Would like variance on output to be the same as the input

$$\texttt{Var}(w_{ij}) = \frac{1}{n} \quad (\texttt{for}: \texttt{Var}(y_i) = \texttt{Var}(x_i)))$$

Alternative Initialisation schemes



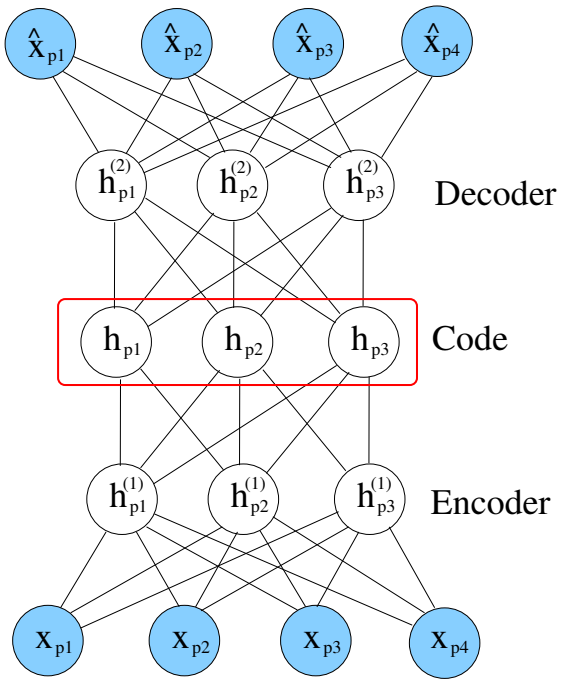
Generative Pre-Training Discriminative Pre-Training

- Restricted Boltzmann machine: generative pre-training
  - initialise the network parameters using a generative RBM
  - train the RBM layer-by-layer
  - if only the output layer trained this is a deep belief network
- Layer-by-layer training: discriminative pre-training
  - iterative process:
    1. remove o/p layer, add a random initialised layer, add o/p layer
    2. train (limited iterations e.g. 1) network and then goto (1)

Example Systems (reference)

Autoencoders (Non-Linear Feature Extraction)

- An autoencoder is a particular form of feed-forward network
  - a (often low-dimensional) code layer ($\boldsymbol{h}_p$)
  - trained to reproduce the input at the output

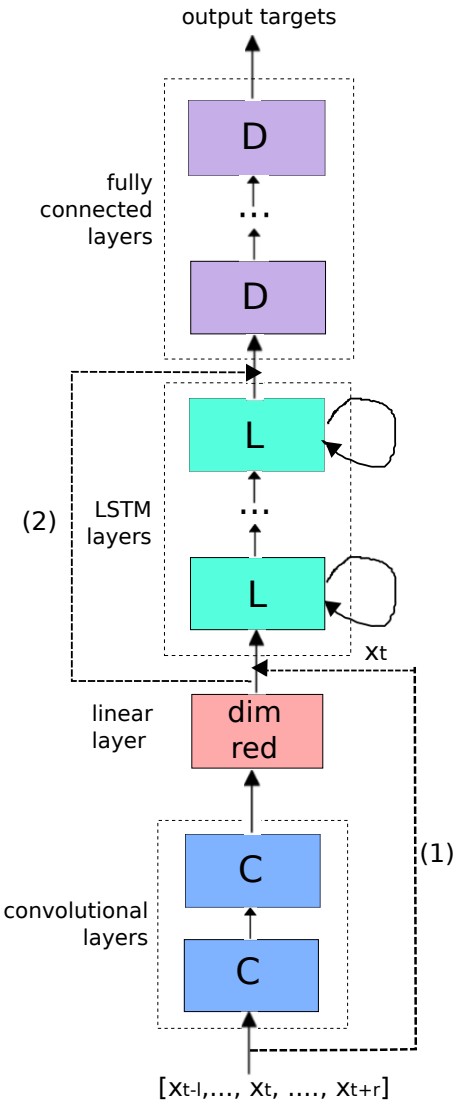


- Training criterion

$$E(\boldsymbol{\theta}) = \sum_{p=1}^{n} f(\boldsymbol{x}_p, \hat{\boldsymbol{x}}_p)$$

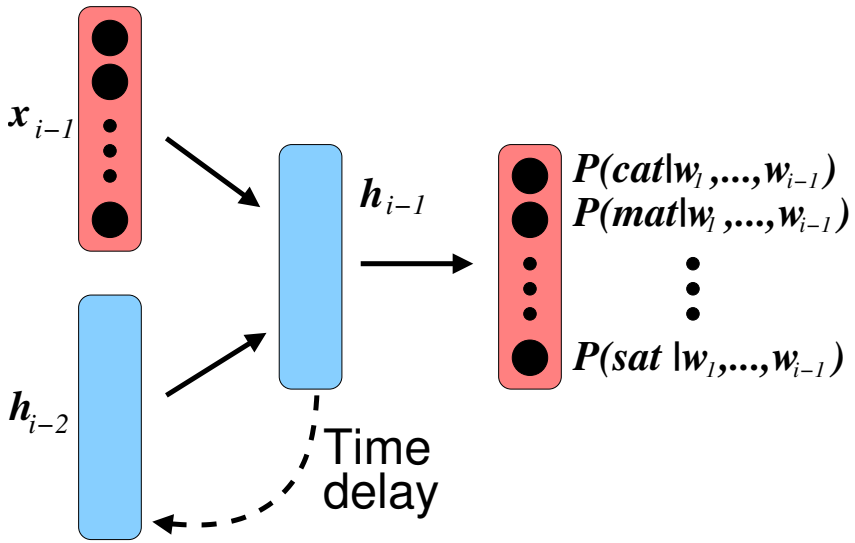
- Can be used to denoise data
  - "noise-corrupted" data in,
  - distance to "clean" data for training

Automatic Speech Recognition [22]



- Example Architecture from Google (2015)
  - C: CNN layer (with pooling)
  - L: LSTM layer
  - D: fully connected layer
- Two multiple layer "skips"
  - (1) connects input to LSTM input
  - (2) connects CNN output to DNN input
- Additional linear projection layer
  - reduces dimensionality
  - and number of network parameters!

Neural Network Language Models [2, 17]



- Neural networks extensively used for language modelling
  - compute $P(\omega_{1:L})$: used in translation/ASR/topic spotting etc
- 1-of-K ("one-hot") coding for ith word, ω_i, $\boldsymbol{x}_i$
  - additional out-of-shortlist symbol may be added
  - softmax activation function on output layer

Neural Network Language Models [2, 17]

- Neural networks extensively used for language modelling
  - recurrent neural networks - complete word history

$$P(\boldsymbol{\omega}_{1:L}) = \prod_{i=1}^{L} P(\omega_i|\boldsymbol{\omega}_{1:i-1}) \approx \prod_{i=1}^{L} P(\omega_i|\omega_{i-1}, \tilde{\boldsymbol{h}}_{i-2}) \approx \prod_{i=1}^{L} P(\omega_i|\tilde{\boldsymbol{h}}_{i-1})$$

- Input and output layer sizes can be very large
  - size of vocabulary ($> 10K$), not an issue for input
  - output-layer (softmax) expensive (normalisation term)
- Issues that need to be addressed
  1. training: how to efficiently train on billions of words?
  2. decoding for ASR: how to handle dependence on complete history?

Machine Translation: Encoder-Decoder Sequence Models

- Train a discriminative model from
  - $\boldsymbol{x}_{1:L} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_L\}$: L-length input sequence (source language)
  - $\boldsymbol{y}_{1:K} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{x}_K\}$: K-length output (target language)

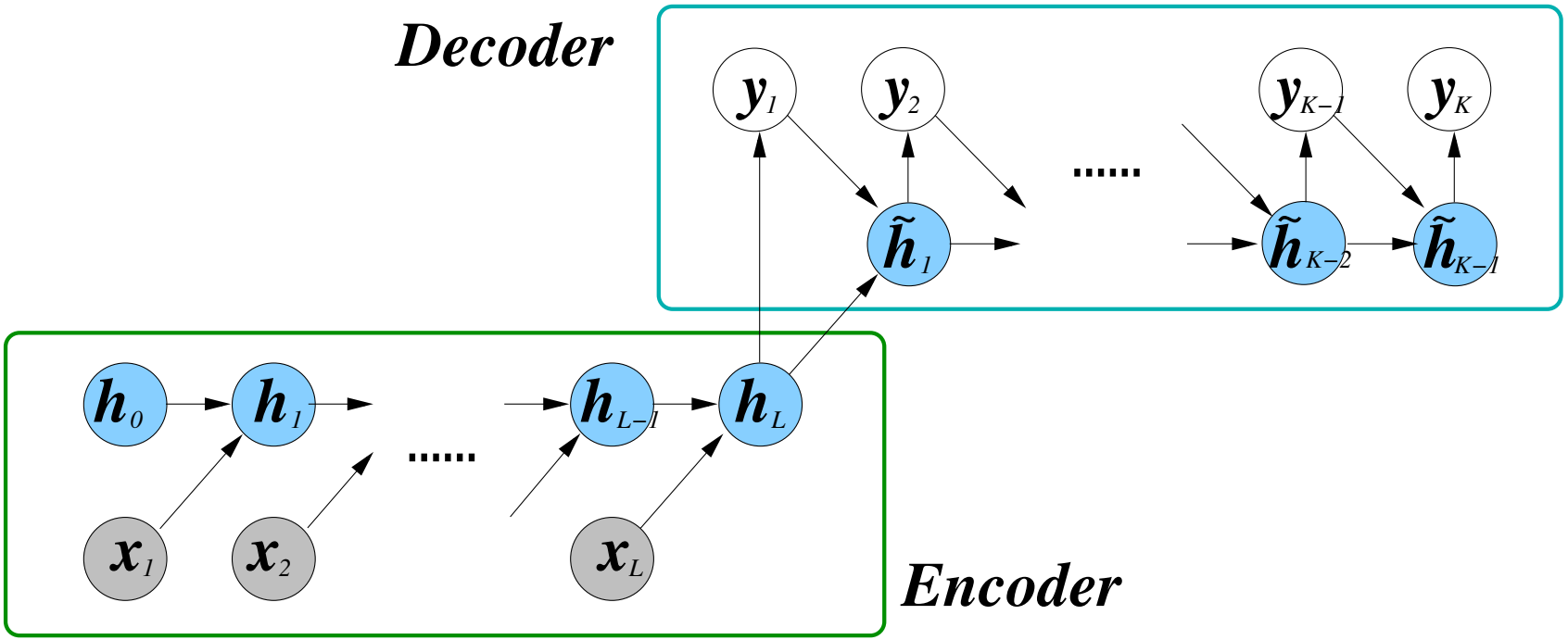
$$
\begin{aligned}
p(\boldsymbol{y}_{1:K}|\boldsymbol{x}_{1:L}) &= \prod_{i=1}^{K} p(\boldsymbol{y}_i|\boldsymbol{y}_{1:i-1}, \boldsymbol{x}_{1:L}) \\
&\approx \prod_{i=1}^{L} p(\boldsymbol{y}_i|\boldsymbol{y}_{i-1}, \tilde{\boldsymbol{h}}_{i-2}, \boldsymbol{c})
\end{aligned}
$$

  - need to map $\boldsymbol{x}_{1:L}$ to a fixed-length vector

$$\boldsymbol{c} = \phi(\boldsymbol{x}_{1:L})$$

  - $\boldsymbol{c}$ is a fixed length vector - like a sequence kernel

RNN Encoder-Decoder Model [9, 16]

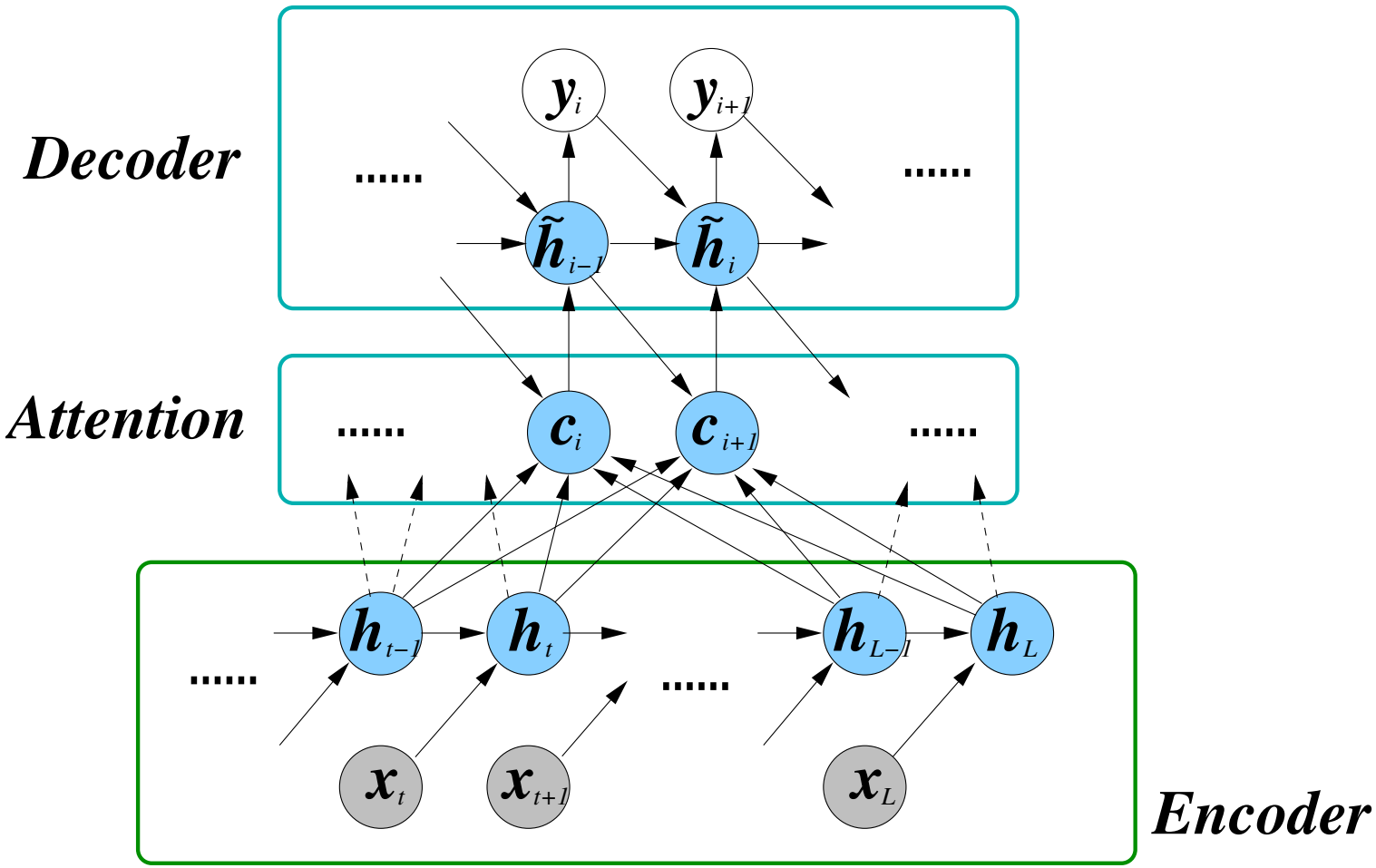


- One form is to use hidden unit from acoustic RNN/LSTM

$$\boldsymbol{c} = \phi(\boldsymbol{x}_{1:L}) = \boldsymbol{h}_L$$

  - dependence on context is global via $\boldsymbol{c}$ - possibly limiting

Attention-Based Models [5, 4, 16]



Attention-Based Models

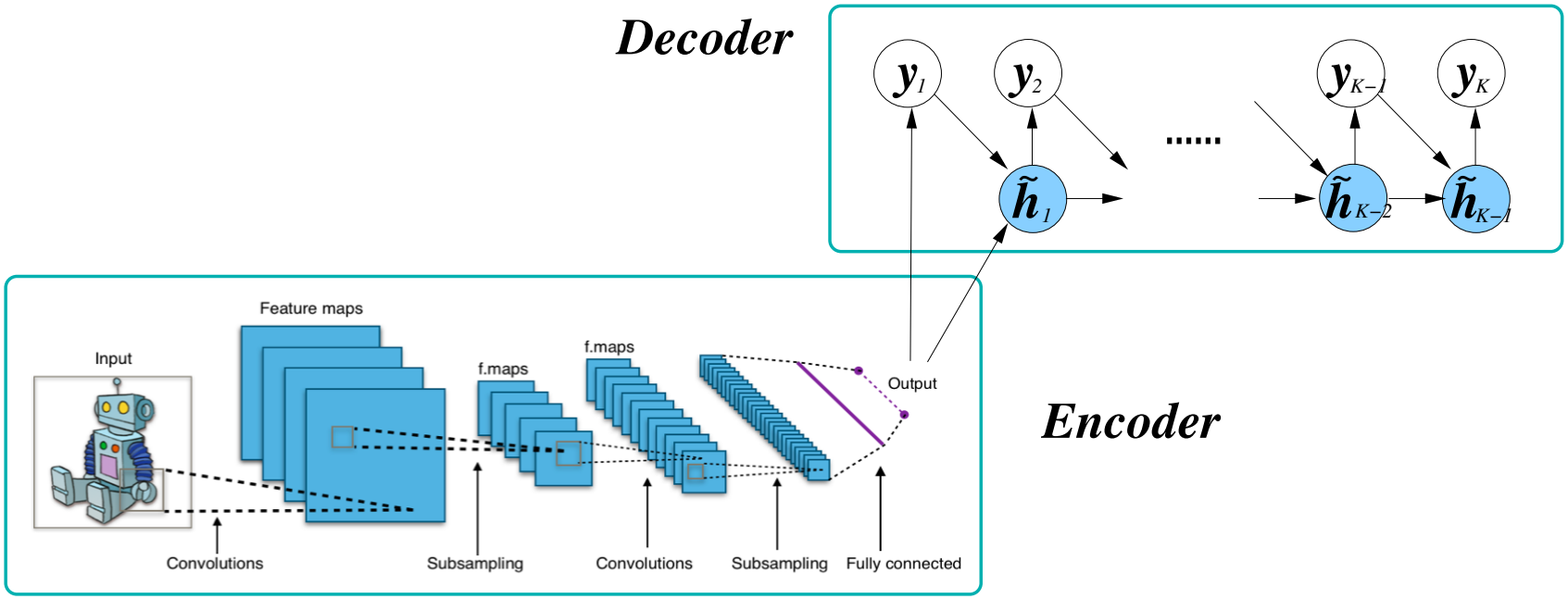
- Introduce attention layer to system
  - introduce dependence on locality i

$$p(\boldsymbol{y}_{1:K}|\boldsymbol{x}_{1:L}) \approx \prod_{i=1}^{K} p(\boldsymbol{y}_i|\boldsymbol{y}_{i-1}, \tilde{\boldsymbol{h}}_{i-1}, \boldsymbol{c}_i) \approx \prod_{i=1}^{K} p(\boldsymbol{y}_i|\tilde{\boldsymbol{h}}_{i-1})$$

$$\boldsymbol{c}_i = \sum_{\tau=1}^{L} \alpha_{i\tau} \boldsymbol{h}_\tau; \quad \alpha_{i\tau} = \frac{\exp(e_{i\tau})}{\sum_{j=1}^{L} \exp(e_{ij})}, \quad e_{i\tau} = f^{\mathrm{e}}\left(\tilde{\boldsymbol{h}}_{i-2}, \boldsymbol{h}_\tau\right)$$

  - $e_{i\tau}$ how well position $i-1$ in input matches position τ in output
  - $\boldsymbol{h}_\tau$ is representation (RNN) for the input at position τ

Image Captioning



- Encode-image as vector use a deep convolutional network
  - generate caption using recurrent network (RNN/LSTM)
  - all parameters optimised (using example image captions)

Conclusions

Is Deep Learning the Solution?

- Deep Learning: state-of-the-art performance in range of tasks
  - machine translation, image recognition/captioning,
  - speech recognition/synthesis ...
- Traditionally use two-stage approach to build classifier:
  1. feature extraction: convert waveform to parametric form
  2. modelling: given parameters train model
- Limitations in feature extraction cannot be overcome ...
  - integrate feature extraction into process
  - attempt to directly model/synthesise waveform (WaveNet)
- BUT
  - require large quantities of data (research direction)
  - networks are difficult to optimise - tuning required
  - hard to interpret networks to get insights
  - sometimes difficult to learn from previous tasks ...

[1] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition," in *2012 IEEE international conference on Acoustics, speech and signal processing (ICASSP)*. IEEE, 2012, pp. 4277–4280.

[2] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[3] C. Bishop, "Mixture density networks," in *Tech. Rep. NCRG/94/004, Neural Computing Research Group, Aston University*, 1994.

[4] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals, "Listen, attend and spell," *CoRR*, vol. abs/1508.01211, 2015. [Online]. Available: http://arxiv.org/abs/1508.01211

[5] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," *CoRR*, vol. abs/1506.07503, 2015. [Online]. Available: http://arxiv.org/abs/1506.07503

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[7] J. Chung, K. Kastner, L. Dinh, K. Goel, A. C. Courville, and Y. Bengio, "A recurrent latent variable model for sequential data," *CoRR*, vol. abs/1506.02216, 2015. [Online]. Available: http://arxiv.org/abs/1506.02216

[8] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks." in *Aistats*, vol. 9, 2010, pp. 249–256.

[9] A. Graves, "Sequence transduction with recurrent neural networks," *CoRR*, vol. abs/1211.3711, 2012. [Online]. Available: http://arxiv.org/abs/1211.3711

[10] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[12] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[14] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[15] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[16] L. Lu, X. Zhang, K. Cho, and S. Renals, "A study of the recurrent neural network encoder-decoder for large vocabulary speech recognition," in *Proc. INTERSPEECH*, 2015.

[17] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[18] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[19] T. Robinson and F. Fallside, "A recurrent error propagation network speech recognition system," *Computer Speech & Language*, vol. 5, no. 3, pp. 259–274, 1991.

[20] S. Ruder, "An overview of gradient descent optimization algorithms," http://sebastianruder.com/optimizing-gradient-descent/index.html, accessed: 2016-10-14.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362.

[22] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 4580–4584.

[23] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[24] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[25] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *CoRR*, vol. abs/1505.00387, 2015. [Online]. Available: http://arxiv.org/abs/1505.00387

[26] P. Swietojanski and S. Renals, "Differentiable pooling for unsupervised acoustic model adaptation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. PP, no. 99, pp. 1–1, 2016.

[27] C. Wu, P. Karanasou, M. Gales, and K. C. Sim, "Stimulated deep neural network for speech recognition," in *Proceedings interspeech*, 2016.

[28] H. Zen and A. Senior, "Deep mixture density networks for acoustic modeling in statistical parametric speech synthesis," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2014, pp. 3844–3848.

[29] C. Zhang and P. Woodland, "DNN speaker adaptation using parameterised sigmoid and ReLU hidden activation functions," in *Proc. ICASSP'16*, 2016.