UNIVERSITY OF CAMBRIDGE

4F10: Deep Learning

Mark Gales

Michaelmas 2020

What is Deep Learning?

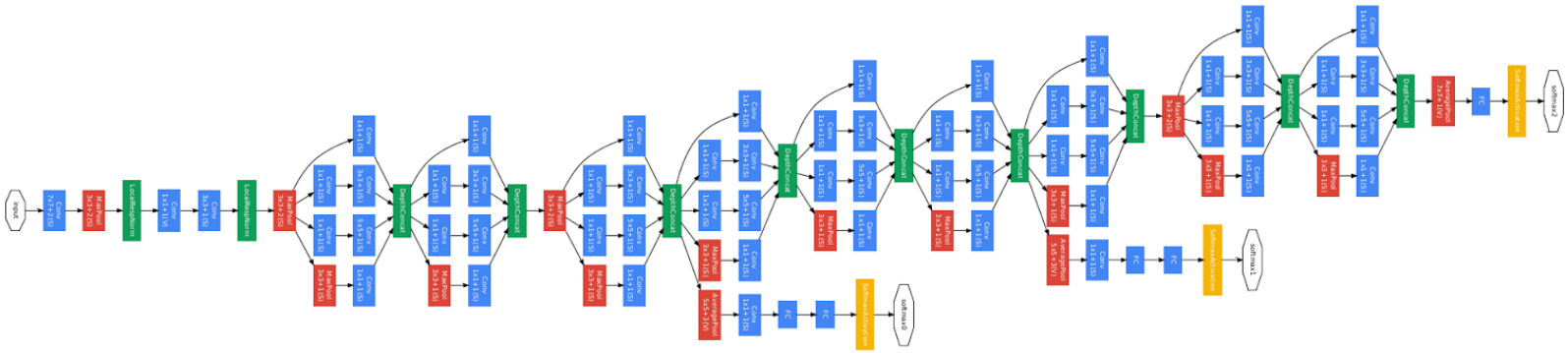
Deep learning is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers, with complex structures or otherwise, composed of multiple non-linear transformations.

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

ImageNet

(a) Siberian husky

(b) Eskimo dog

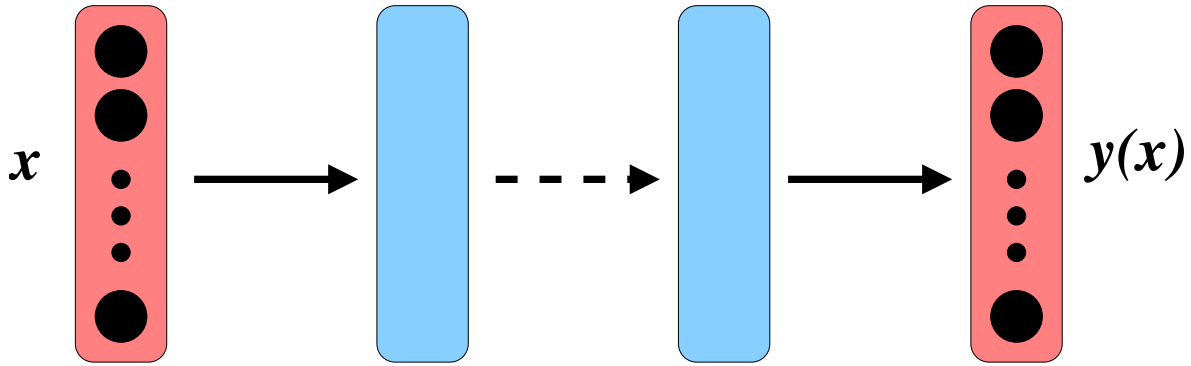


Overview

- 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

Basic Building Blocks

Deep Neural Networks [4]

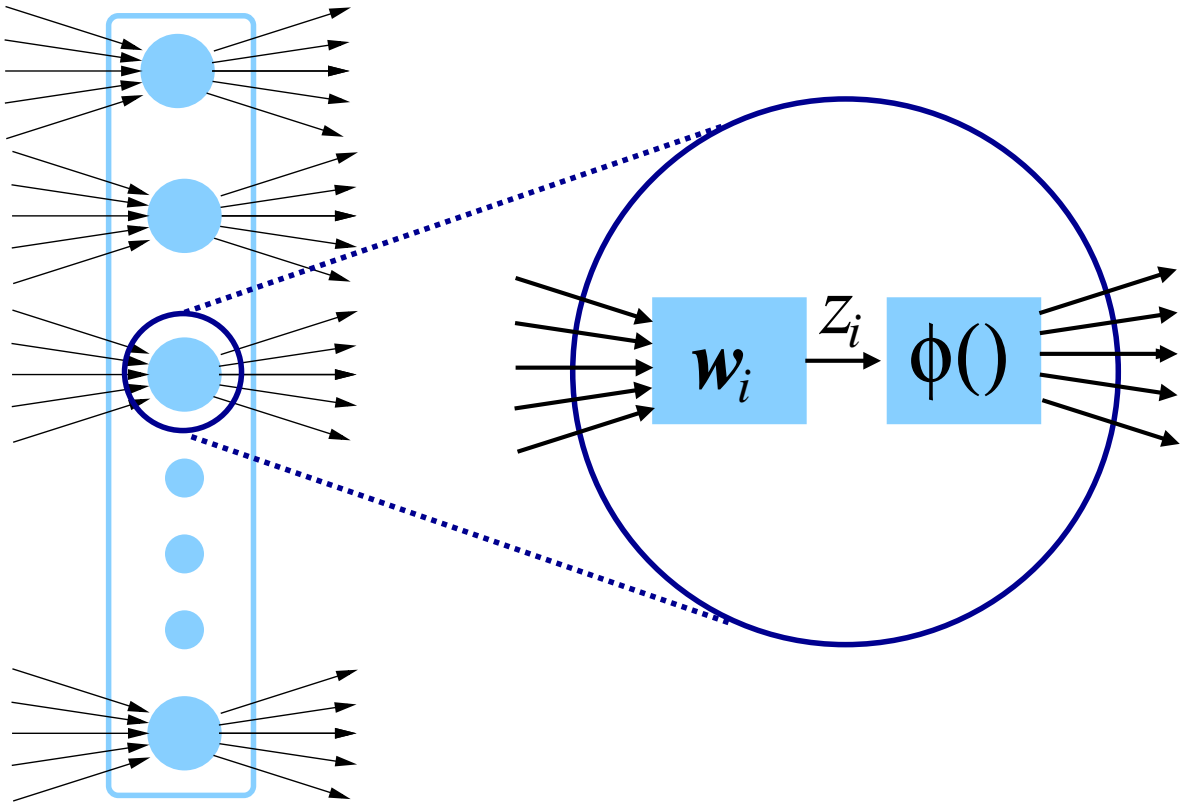


- General mapping process from input $\boldsymbol{x}$ to output $\boldsymbol{y}(\boldsymbol{x})$

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

  - 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:

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

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 (here the bias term is ignored)

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

  - 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, $0 \leq y_i(\boldsymbol{x}) \leq 1$.

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

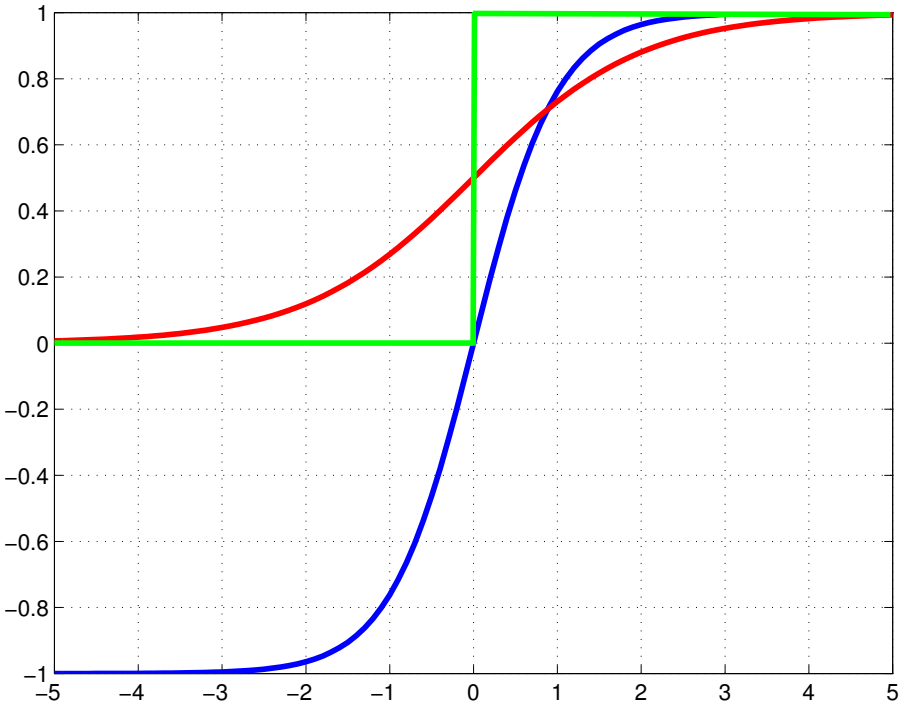
- Softmax function: output $0 \leq y_i(\boldsymbol{x}) \leq 1$, $\sum_{i=1}^{n} y_i(\boldsymbol{x}) = 1$.

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

- 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

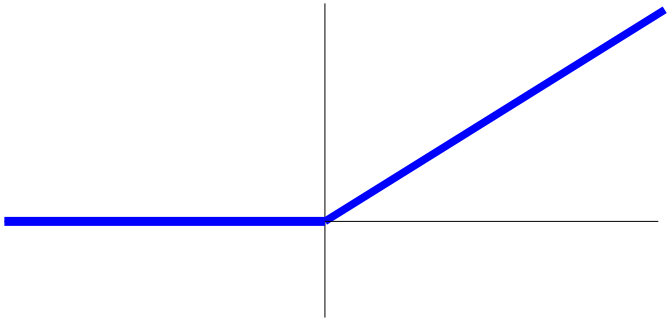


- Activation functions:
  - step function (green)
  - sigmoid function (red)
  - tanh function (blue)

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

Activation Functions - ReLUs [8]

- Alternative activation function: Rectified Linear Units



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

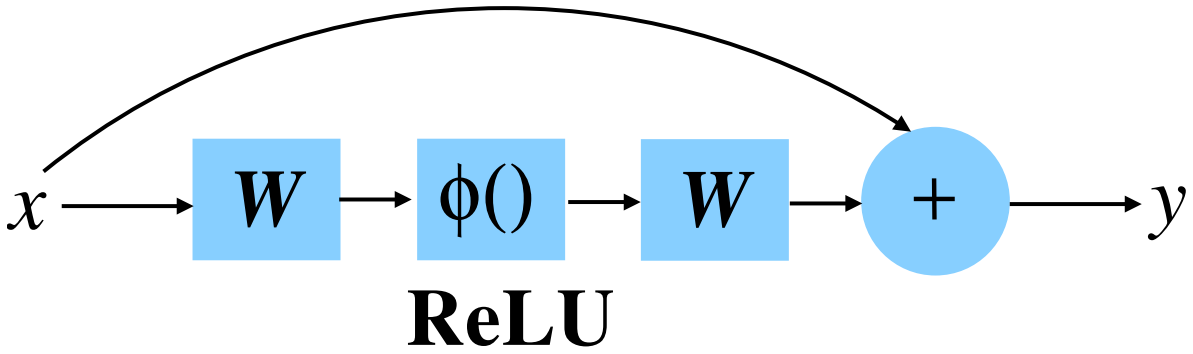
  - efficient, no exponential/division, rapid convergence in training
- Related activation function noisy ReLU:

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

- 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 [3]



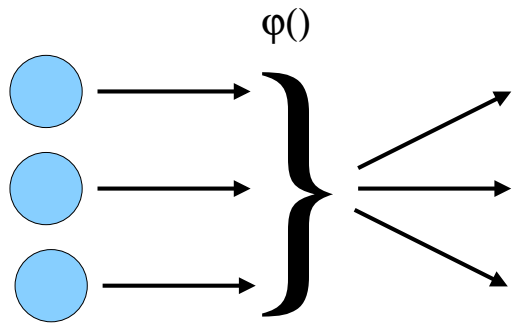
- Modify layer to model the residual

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

  - allows deeper networks to be built - deep residual learning
- Links to highway connections - discussed later in course

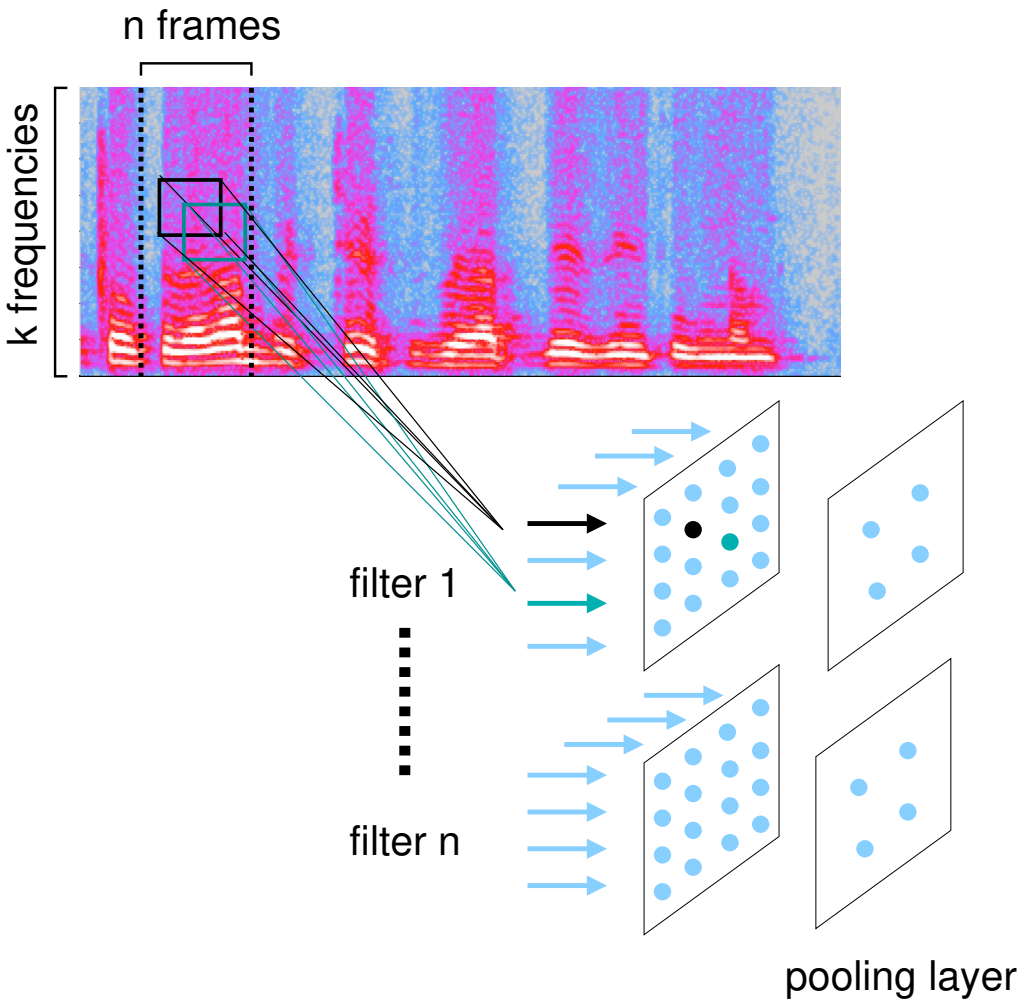
Pooling/Max-Out Functions [7]

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



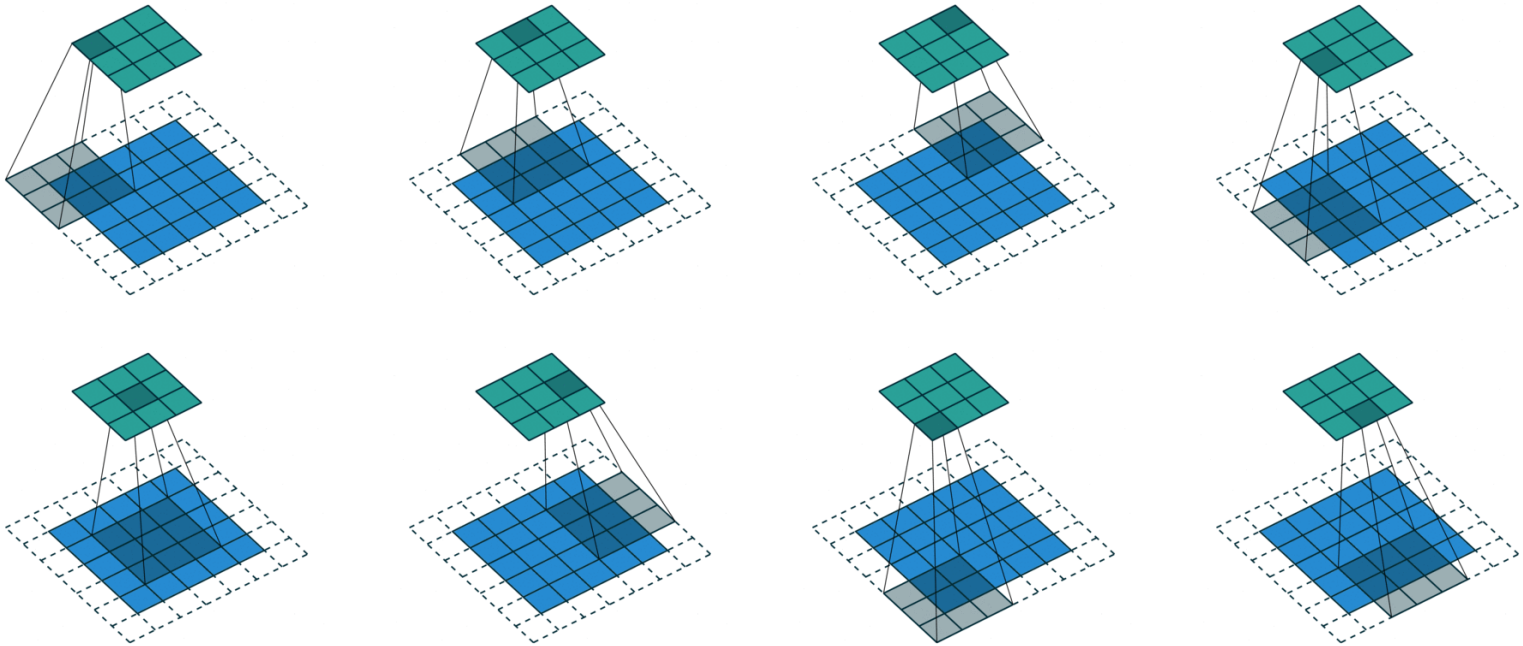
- A range of functions have been examined
  - maxout $\phi(y_1, y_2, y_3) = \max(y_1, y_2, y_3)$
  - soft-maxout $\phi(y_1, y_2, y_3) = \log(\sum_{i=1}^{3} \exp(y_i))$
  - p-norm $\phi(y_1, y_2, y_3) = (\sum_{i=1}^{3} |y_i|^p)^{1/p}$

Convolutional Neural Networks [6]



2-D Convolutional Neural Network

- Example CNN for a 2-D image



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

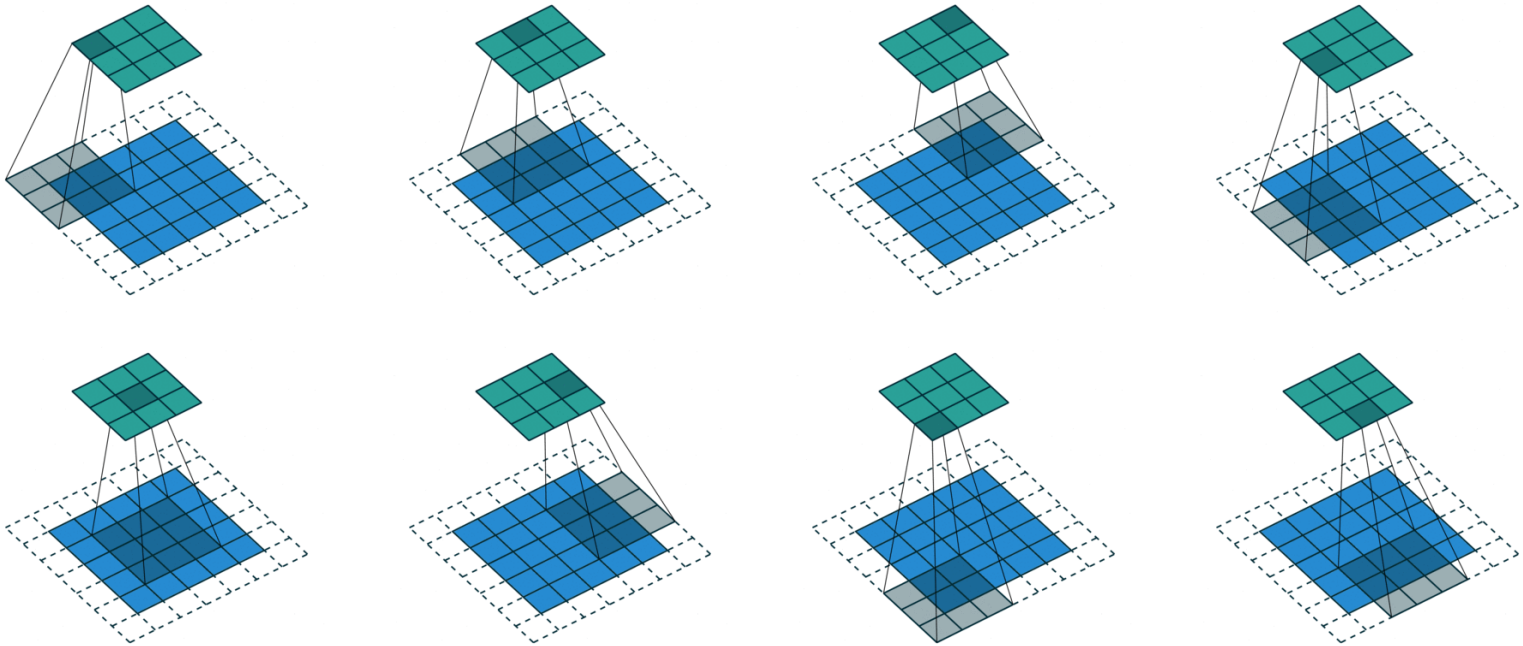
  sequence: $1, 2, 3, 4, 5, 6$ filter: $1, 1, 1$

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

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

2-D Convolutional Neural Network

- 2-D example: receptive field 3×3, stride=2, zerp-padding



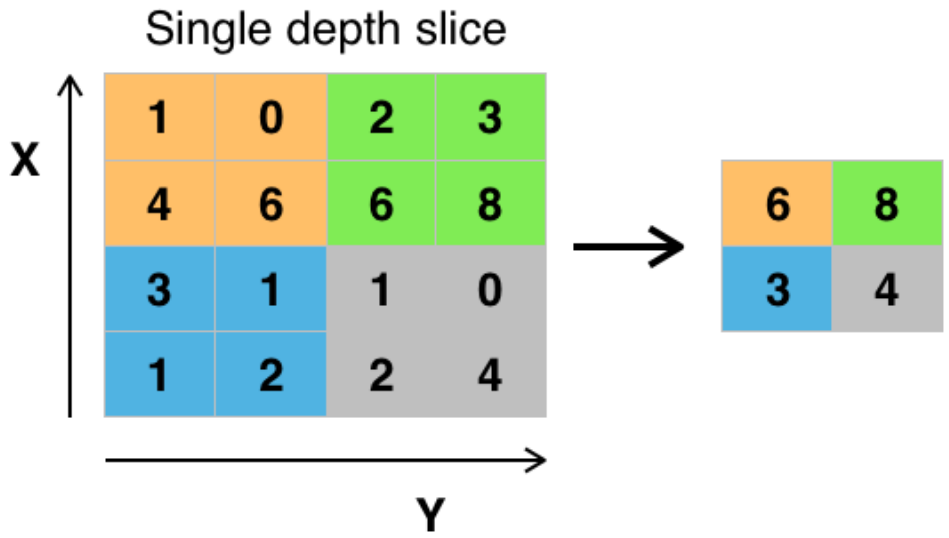
2-D Convolutional Neural Network (cont)

- For a 2-D 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

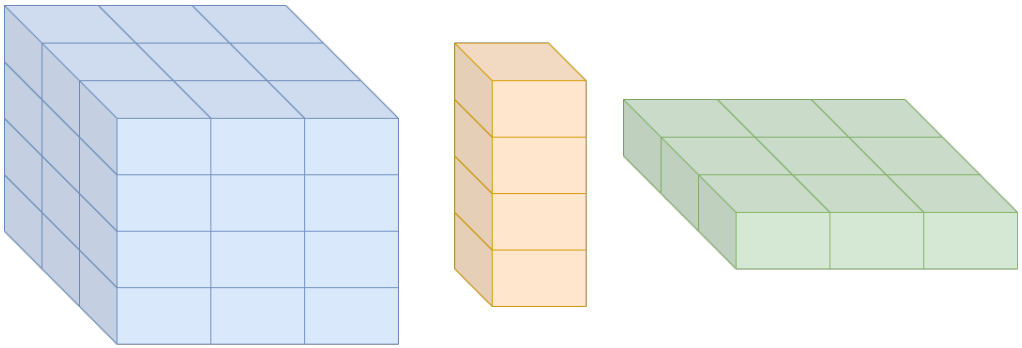
CNN Max-Pooling (Subsampling)



- 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

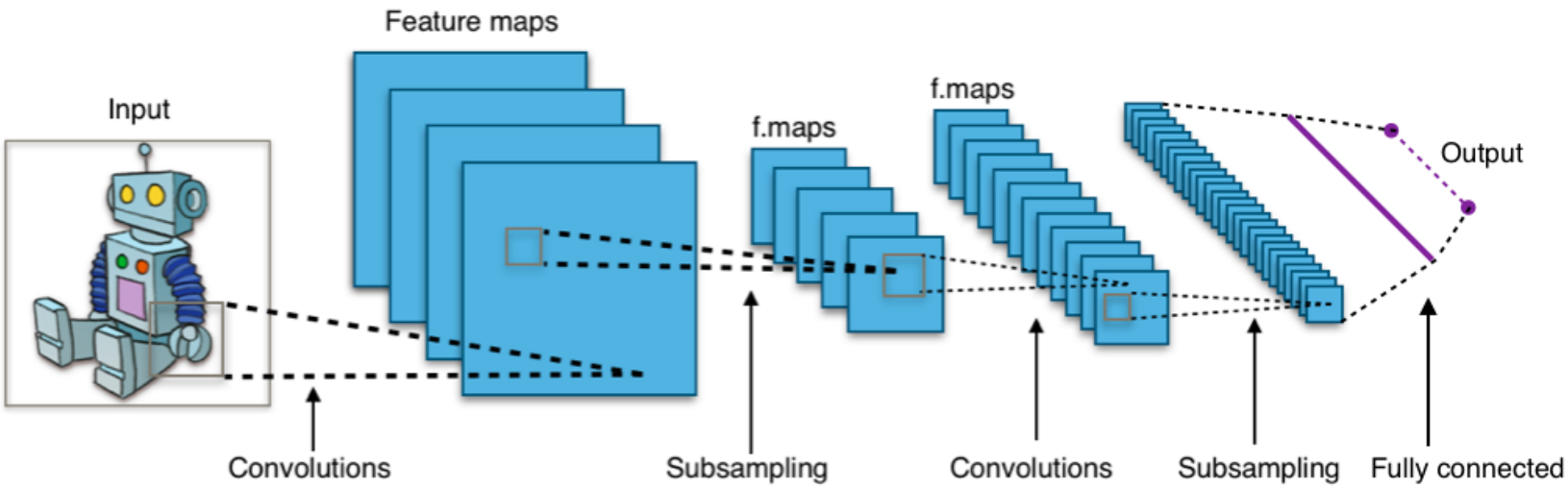
1-D Convolutional Neural Network

- One extreme version of CNN is the 1-D CNN
  - receptive field is 1×1 acting on four 3×3 image features



- Enables the number of features to be modified
  - additional 1xd CNN filters used to increase the output size
  - simple approach to controlling number of model parameters

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

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 $\{\omega_1, \ldots, \omega_K\}$
- 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

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

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

Maximum Likelihood - Regression Criterion

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

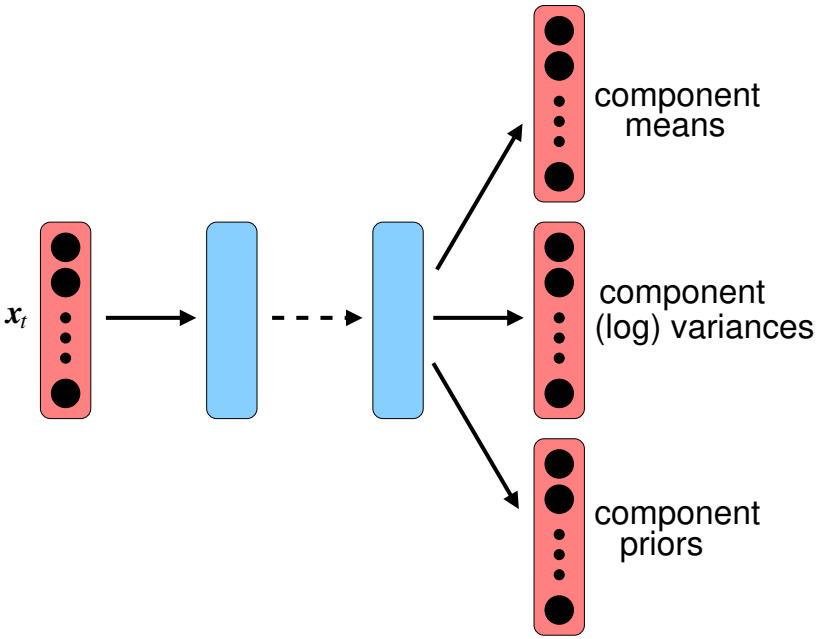
$$E(\boldsymbol{\theta}) = \sum_{p=1}^{N} \log(p(\boldsymbol{y}_p|\boldsymbol{x}_p;\boldsymbol{\theta}))$$

  - 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 [1, 11]



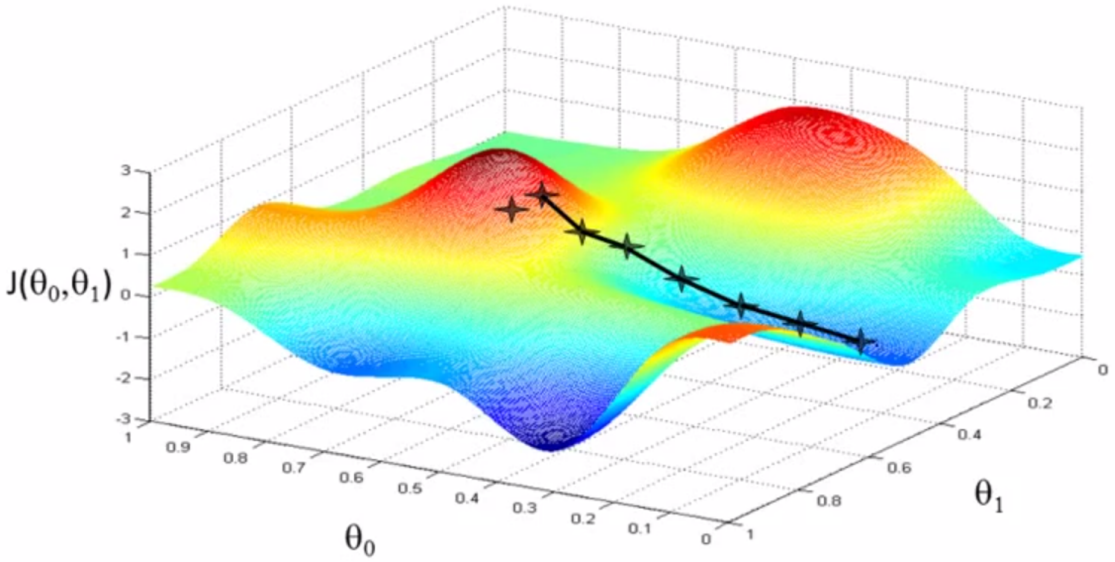
- Predict a mixture of M Gaussians
  - $\mathcal{F}_m^{(c)}(\boldsymbol{x}_p)$: prior prediction
  - $\mathcal{F}_m^{(\mu)}(\boldsymbol{x}_p)$: mean prediction
  - $\mathcal{F}_m^{(\sigma)}(\boldsymbol{x}_p)$: variance prediction
- For component m, output

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

- Optimise using maximum likelihood where

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

Gradient Descent [9]



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

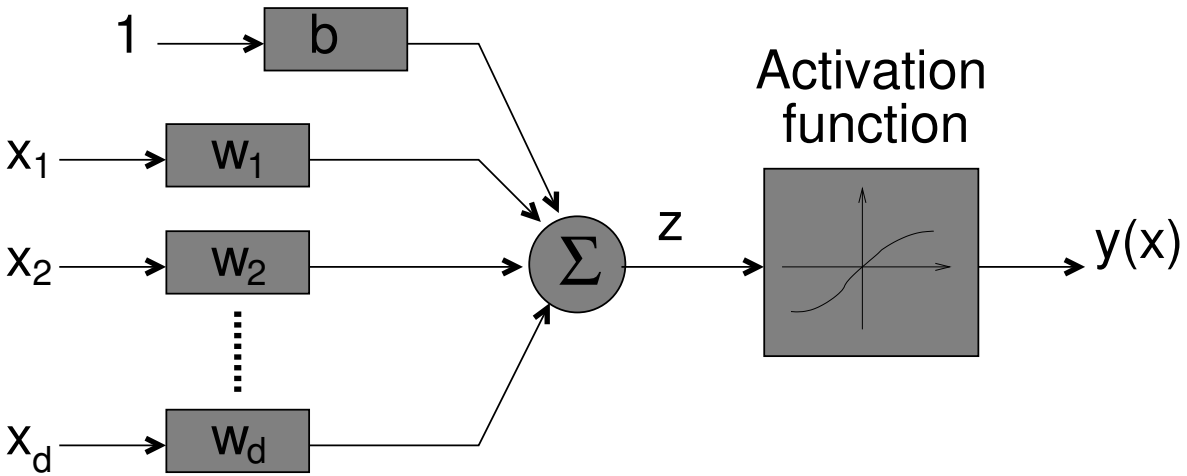
$$\boldsymbol{\theta}[\tau+1] = \boldsymbol{\theta}[\tau] - \boldsymbol{\Delta\theta}[\tau] = \boldsymbol{\theta}[\tau] - \eta \left.\frac{\partial E}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}[\tau]}$$

  - 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
  - ignore bias vector - see detailed notes if interested

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



- 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

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

  - 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

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

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 $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{y}^{(L+1)}$

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

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

- Complete 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, $\tilde{\mathcal{D}}$,

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

- 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

- Multiple mini-batch updates often 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)

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

  - 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

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

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.

$$\mathbf{v}^{\mathsf{T}}\mathbf{H}\mathbf{v} > 0$$

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

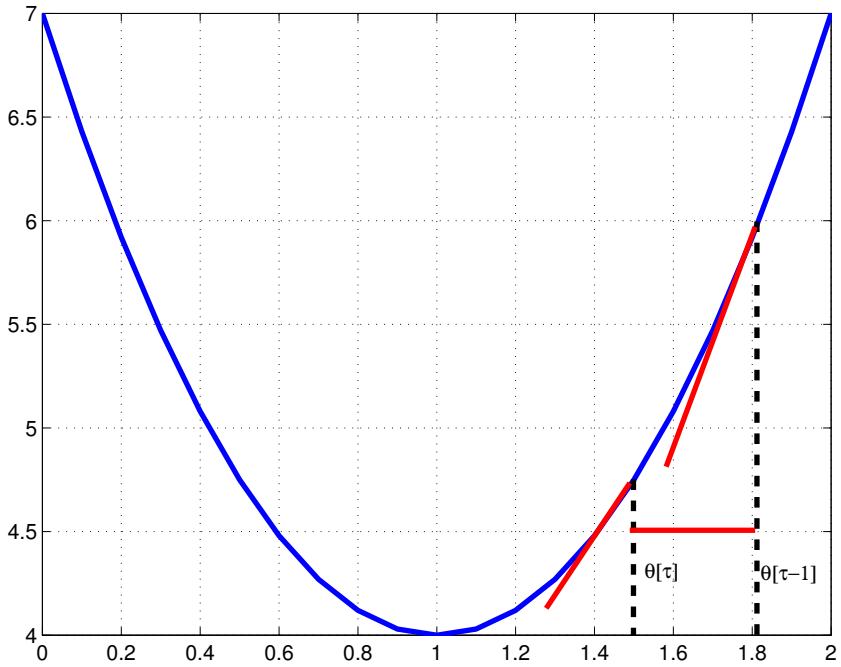
$$\begin{aligned} E(\theta) &\approx E(\theta[\tau]) + b(\theta - \theta[\tau]) + a(\theta - \theta[\tau])^2 \\ \frac{\partial E(\theta)}{\partial \theta} &\approx b + 2a(\theta - \theta[\tau]) \end{aligned}$$

- 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
- At minimum of quadratic approximation: $\theta[\tau + 1] = 1$

Regularisation

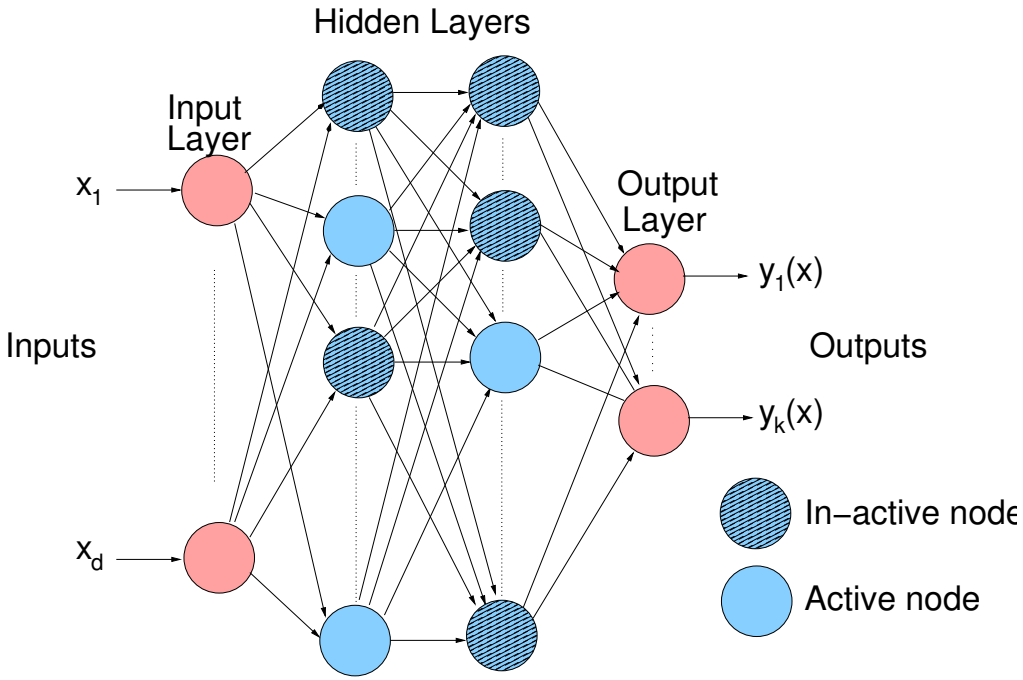
- A major issues with training networks is generalisation
- Simplest approach is early stopping
  - don't wait for convergence - just stop ...
- To address this forms of regularisation are used
  - one standard form is

$$\tilde{E}(\boldsymbol{\theta}) = E(\boldsymbol{\theta}) + \nu\Omega(\boldsymbol{\theta}); \quad \Omega(\boldsymbol{\theta}) = \frac{1}{2}\sum_{l=1}^{L+1}\sum_{i,j} w_{ij}^{(l)2}$$

  - a zero "prior" is used for the model parameters
- Simple to include in gradient-descent optimisation
  - considering only parameters for layer l ($\boldsymbol{W}^{(l)}$)

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

Dropout [10]



- 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

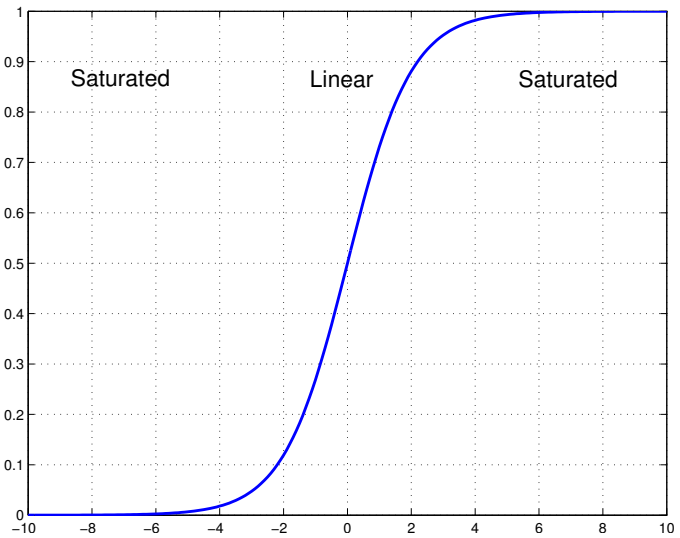
- As with standard classifiers two stage classification often used
  - features are designed by expert
  - current trend to remove two-stage process end-to-end
- Features may have different dynamic ranges
  - consider dimension 1:$-1000 \rightarrow 1000$ vs dimension 2:$-1 \rightarrow 1$
  - can influence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

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

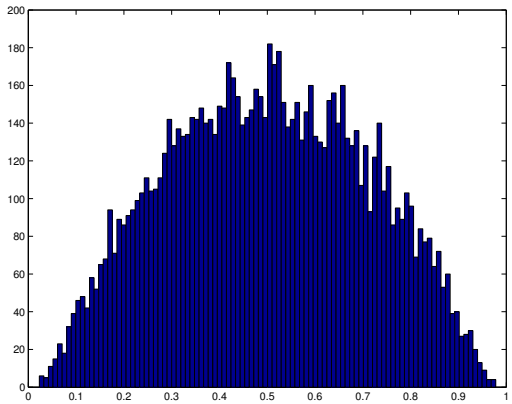
  - only influences initialisation (linear transform and bias)

Network Initialisation: Weight Parameters

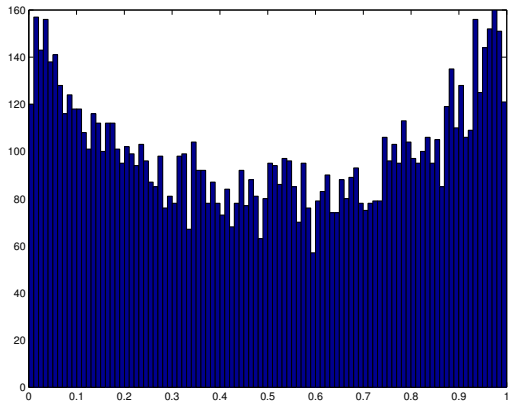
- A starting point (initialisation) for gradient descent is useful
  - one of the "old" concerns with deep networks was initialisation
  - recurrent neural networks are very deep!
- It is not possible to guarantee a good starting point, but
  - would like a parsimonious initialisation
- What about Gaussian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



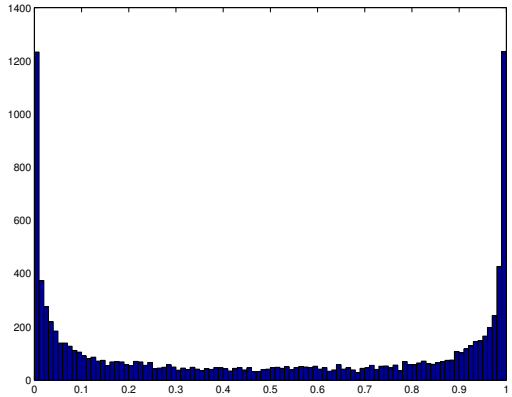
Gaussian Initialisation



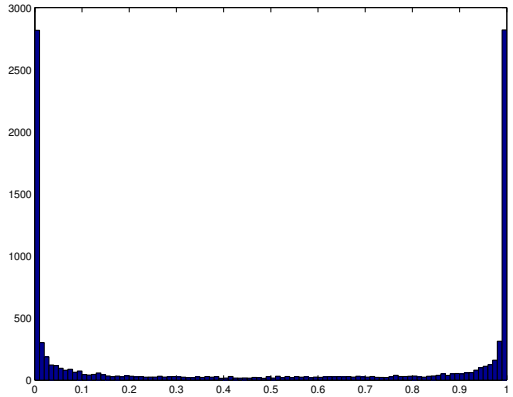
$\mathcal{N}(0, 1)$



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



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



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

- Pass 1-dimensional data through a sigmoid

Exploding and Vanishing Gradients [2]

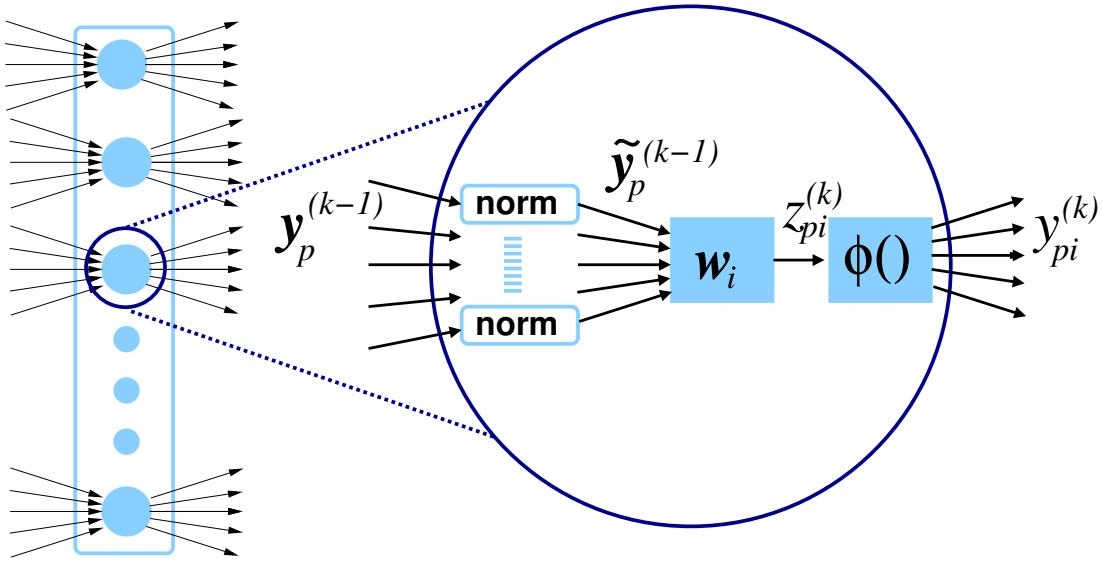
- Need to worry about the following gradient issues
  - Vanishing: derivatives go to zero - parameters not updated
  - Exploding: derivatives get very large - cause saturation
- Xavier Initialisation: simple scheme for initialising weights
  - linear activation functions $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$,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

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

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

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

Optimisation Refinement: Batch Normalisation [5]



- What about applying data-preprocessing for output of a layer
  - output of layer $k-1$ to input $\boldsymbol{x}_p$, $\boldsymbol{y}_p^{(k-1)}$, is the input to layer k
  - applying normalisation to $\boldsymbol{y}_p^{(k-1)}$ yields $\tilde{\boldsymbol{y}}_p^{(k-1)}$
  - normalisation term will vary as system trains ...
- Attempts to "keep" values in linear region ...
  - should speed up training

Optimisation Refinement: Batch Normalisation (cont)

- Expensive to compute normalisation for all training data, $\mathcal{D}$
- Consider applying normalisation over a batch, $\tilde{\mathcal{D}}$ in SGD
  - each normalisation term a noisy estimate of the real term
  - apply normalisation to a particular node j

$$\tilde{y}_{pj}^{(k-1)} = \frac{y_{pj}^{(k-1)} - \tilde{\mu}_j^{(k)}}{\tilde{\sigma}_j^{(k)}}; \quad \tilde{\mu}_j^{(k)} = \frac{1}{|\tilde{\mathcal{D}}|} \sum_{p \in \tilde{\mathcal{D}}} y_{pj}^{(k-1)}; \quad \tilde{\sigma}_j^{(k)2} = \frac{1}{|\tilde{\mathcal{D}}|} \sum_{p \in \tilde{\mathcal{D}}} (y_{pj}^{(k-1)} - \tilde{\mu}_j^{(k)})^2$$

- The gradients can now be expressed as

$$\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{y}^{(k-1)}} = \frac{\partial \tilde{\boldsymbol{y}}^{(k-1)}}{\partial \boldsymbol{y}^{(k-1)}} \frac{\partial E(\boldsymbol{\theta})}{\partial \tilde{\boldsymbol{y}}^{(k-1)}}$$

  - both $\tilde{\mu}_j^{(k)}$ and $\tilde{\sigma}_j^{(k)2}$ depend on $y_{pj}^{(k-1)}$
  - normalisation is computed for each mini-batch (hence name)

Optimisation Refinement: Batch Normalisation (cont)

- At training time a batch is available to compute normalisation

  How to apply this at test time with only one sample $\boldsymbol{x}^\star$

  - not an issue for whitening as normalisation constant ...
- Compute the expected normalisation (after training network)

$$\overline{\mu}_j^{(k)} = \mathbb{E}\left\{\tilde{\mu}_j^{(k)}\right\}; \quad \overline{\sigma}_j^{(k)2} = \frac{m}{(m-1)}\mathbb{E}\left\{\tilde{\sigma}_j^{(k)2}\right\}$$

  - size of each batch is m samples
  - draw large number of batches $\tilde{\mathcal{D}}$ for expectation
- Network is not more "powerful"
  - normalisation can be subsumed by weights (and bias)
  - network should train faster ... empirically it does ...

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

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

[3] 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

[4] 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.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.