Neural networks

Handle curse of dimensionality by:

Fixing number of basis functions, and have parameters in these basis functions which can be adapted.

Multiple layers of logistic regression. (Likelihood function will no longer be a convex function)

- Functional form of Neural network
- Determine network parameters (error backpropagation)
- Regularization of training
- Bayesian neural networks

Feed Forward network functions

- Linear models for regression and classification used

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right)$$

- Classification: f is nonlinear activation function
- Regression: f is identity
- Here: Make basis functions $\phi_j(\mathbf{x})$ depend on parameters which are adjustable along with coefficients $\{w_j\}$ during training

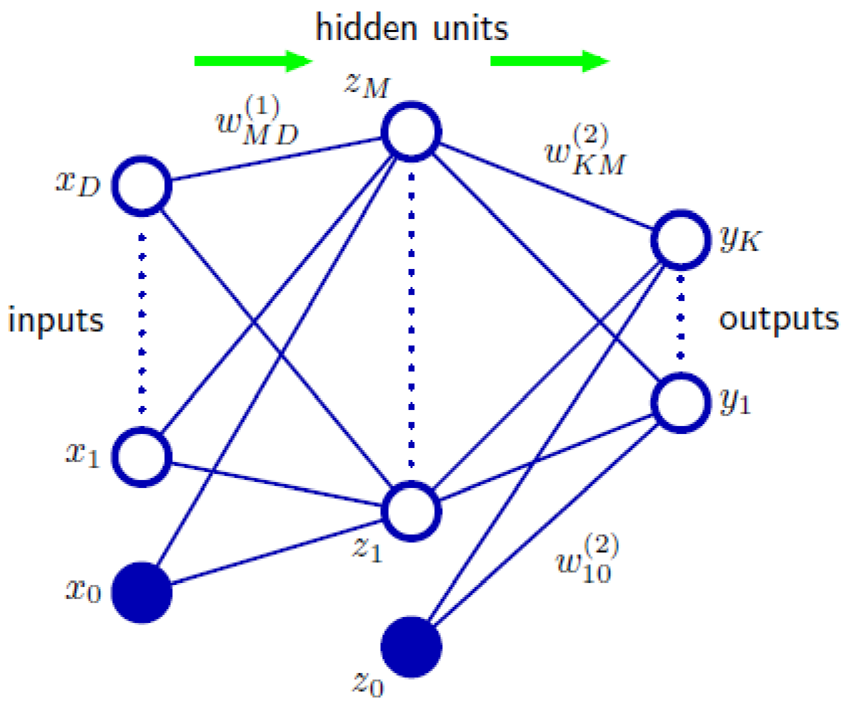
- Basic neural network model: Series of functional transformations
- **M** linear combinations of input
- $$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$ (1) represents the layer number
- D is number of variables
- $W_{ji}^{(1)}$ -> Weights W_{j0} -> Bias a_j -> activations
- Above is transformed by a differentiable non-linear activation function h (.) => $Z_j = h(a_j)$
- Hidden functions are the y(x,w)

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right)$$

- Non-linear functions h are sigmoidal functions like logistic sigmoid or tanh

- Output unit activations $$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

- Above is for second layer



- Nodes-> I/p, o/p, hidden
- Weights -> links between nodes
- x_0 z_0 -> bias

- Standard regression: activation function is identity $y_k = a_k$
- Binary Classification: use logistic sigmoid $y_k = \sigma(a_k)$

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

- Multi class classification: Softmax function
- Overall network function (for sigmoidal o/p unit activation)

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

- Above is a non-linear function from a set of I/p variables $\{x_i\}$ to a set of o/p variables $\{y_k\}$ controlled by a vector w of adjustable parameters

- Evaluating this equation is a "forward propagation" of information
- (Internal nodes are deterministic)

- To absorb bias parameters into weight parameters define x_0 with value =1,

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i.$$

- Absorb other layer biases similarly, then overall network function is

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

- Two stages of processing (one for number of variable, second for number of linear combinations) <= **Multilayer perceptron (MLP)**
- With continuous sigmoidal nonlinearities in hidden units.

- Generalization of network:
    - Skip-layer : directly go from input to output
    - Sparse network: Not all connections are present
- Feed-forward network: Has no closed directed cycles => ensures outputs are deterministic functions of inputs
- Feed forward Neural networks are called "universal approximators" as they can approximate generally all functions

- In a feed forward architecture, each hidden and output unit computes $z_k = h(\Sigma (w_{kj} z_j)$
- Weight Space Symmetry:
  multiple distinct choices for the weight vector can all give rise to the same output for an input
- ex: change the sign of all weights feeding INTO an unit then by correspondingly changing the sign of all the weights coming OUT of the unit the mapping is preserved
- tanh(-a) = -tanha So another negation will give back the original
- for M hidden units there will be M similar "sign flip " symmetries. So any weight vector is one of a set of 2^M equivalent weight vectors
- Similarly one can interchange (swap) weights and bias of a hidden unit with another hidden unit and preserve the mapping of input to output
- any weight vector will belong to a set of M! equivalent weight vectors
- *Above has little practical significance*

Network training:

- Finding network parameters is similar to polynomial curve fitting => minimize sum of squares function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$$

- <u>Probabilistic interpretation of outputs:</u>
- relevant to o/p unit nonlinearity and choice of error function
- For Regression: Assume t is Gaussian distribution, x-dependent mean given by output of neural network

- Then $$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}\right)$$

- Take output unit activation function as identity
- Then likelihood function is

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

- Negative logarithm gives error function:

$$\frac{\beta}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2}\ln\beta + \frac{N}{2}\ln(2\pi)$$

- Learn 'w' and 'β' from this

- Maximization of likelihood function is similar to Minimization of Error

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

- (leaving out the additive and multiplicative *Constants*)
- W value got by minimizing error -> W_{ML}
- Value of β -> minimize negative log likelihood

$$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{N}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}_{\mathrm{ML}}) - t_n\}^2$$

- Case of multiple target variables:
- Assume independent conditional on X and W with shared noise precision β.
- Conditional distribution of target values = $p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \beta^{-1}\mathbf{I}\right)$
- Maximum likelihood weights got by minimizing sum of squares error function
- Noise precision (K target variables) $$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{NK}\sum_{n=1}^{N}\|\mathbf{y}(\mathbf{x}_n, \mathbf{w}_{\mathrm{ML}}) - \mathbf{t}_n\|^2$$

- Error function is linked to output unit activation function, For regression o/p unit activation function is identity => $y_k = a_k$
- Sum of squares error function has property -> $$\frac{\partial E}{\partial a_k} = y_k - t_k$$

- Binar classification: Consider network with single o/p with activation function as logistic sigmoid

$$y = \sigma(a) \equiv \frac{1}{1 + \exp(-a)}$$

- $0 <= y(x,w) <= 1$
- Taking y(x,w) as conditional probability $p(C_1/x)$, conditional distribution of targets becomes Bernoulli distribution (0 or 1)

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \{1 - y(\mathbf{x}, \mathbf{w})\}^{1-t}$$

- Error function becomes a cross entropy error function
- $$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$ $y(x_n, w)$ is y_n

- Cross entropy: Measure of correctness of a classification model (0 is best). Between 2 probability distributions: Average number of bits needed to identify an event of one distribution (p) when represented in the other distribution (q)
- For classification problems : Cross entropy gives better and faster results than sum of squares
- Derivative of error function w.r.t activation for an o/p unit is again

$$\frac{\partial E}{\partial a_k} = y_k - t_k$$

- In Neural network: every layer, especially first layer does feature extraction, as weight parameters are shared between outputs.
- In Linear model: each classification problem is independently solved

- For multi class classification: K mutually exclusive classes
- Target variables have 1 of K coding scheme,
- outputs are $y_k(x,w) = p(t_k = 1/x)$
- Then error function =

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w})$$

- Output unit activation function =

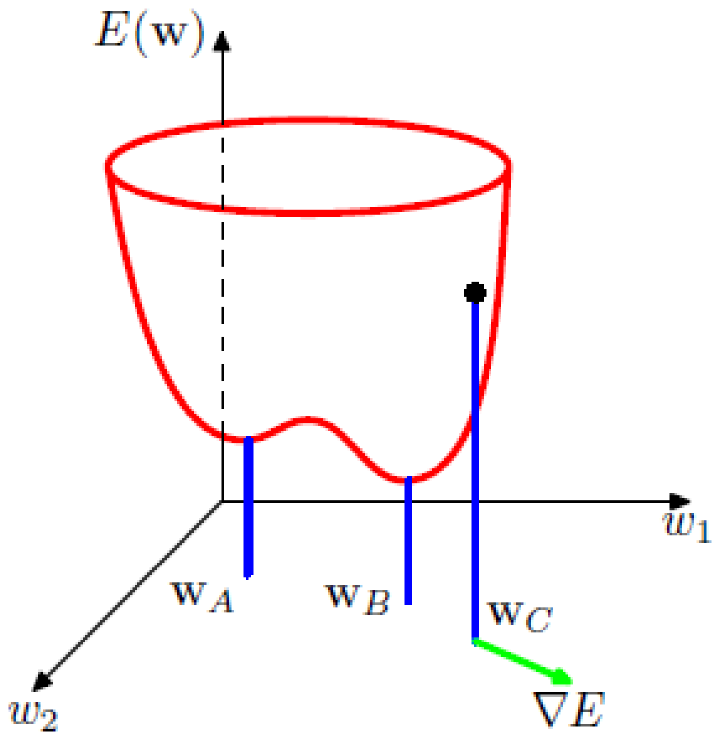
$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}$$

Choice of output unit activation & Error fun

- Regression:
  - Linear outputs and Sum of Squares error
- (independent) Binary Classification:
  - Logistic sigmoid output and Cross Entropy error function
- Two class Classification:
  - Single logistic sigmoid output or network with 2 o/ps having softmax o/p activation
- Multi class Classification:
  - Softmax output and multi-class cross entropy error function

Parameter Optimization

- Find weight vector which minimizes error function E(w)
- W_A – local minimum
- W_B– global minimum
- At any point W_C
- local gradient
- of error surface is vector



$$\nabla E$$

- A small change in weight w becomes $\mathbf{w} + \delta\mathbf{w}$
- Then change in error function is $\delta E \simeq \delta\mathbf{w}^{\mathrm{T}} \nabla E(\mathbf{w})$
- Smallest value of E(w) will be at a point where gradient of error is zero

$$\nabla E(\mathbf{w}) = 0$$

- If not there would still be a way to move in direction of $-\nabla E(\mathbf{w})$

and further reduce error

- Points of gradient vanishing are called stationary points –>
- Minima, maxima, saddle points
- Since error function has a non-linear dependence on weights and bias, Error gradient will vanish at more than one point ->

- **Searching through an Ellipse of values**
- Global minimum
- Local minimum
- To find $\nabla E(\mathbf{w}) = 0$ there are no analytical solutions. So we use iterative solution.
- From an initial value of $w^{(0)}$ move through weight space in

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$$

- The algorithms to update the weight vector utilise gradient information

- Local quadratic approximation to the error function helps us to understand the process
- Taylor expansion of E(w) around point w^ in weight space

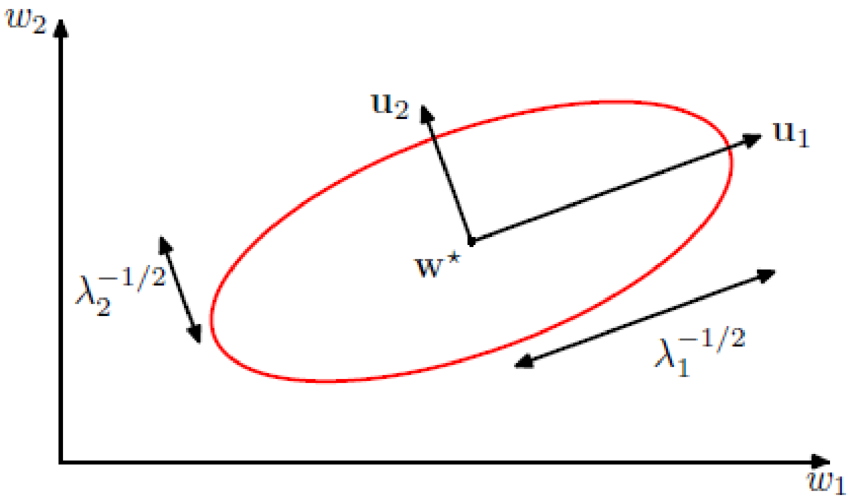
$$E(\mathbf{w}) \simeq E(\widehat{\mathbf{w}}) + (\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{b} + \frac{1}{2}(\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}})$$

- b is gradient of E evaluated at w^ $\quad \mathbf{b} \equiv \nabla E|_{\mathbf{w}=\widehat{\mathbf{w}}}$
- Hessian matrix H, is second order differential of error gradient $\nabla\nabla E$
- and has elements $(\mathbf{H})_{ij} \equiv \left.\frac{\partial E}{\partial w_i \partial w_j}\right|_{\mathbf{w}=\widehat{\mathbf{w}}}$
- Now the Taylor expansion of gradient becomes:

$$\nabla E \simeq \mathbf{b} + \mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}})$$

- When w is close to w^ ,a good approximation of error and gradient are got

- <u>For a minimum point:</u> $\nabla E = 0$ at $\mathbf{w}^\star$
- => linear term disappears
- => Original Taylor expansion of E(w) = $E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^\star)$
- (Hessian matrix is evaluated at w*)
-
-
-



Constant error ->

ellipse contour

- Use of "gradient information" will reduce computation time from $O(W^3)$ to $O(W^2)$
- Explanation: In the Taylor expansion error surface is given by b and H. This contains $W(W+3) / 2$ elements (W is dimensionality of w)
- W^2 parameters => $O(w^2)$ to get each of these and each has $O(W)$ steps => $O(W^3)$
- By using gradient information (error back propagation) each evaluation needs only $O(W)$ steps => totally $O(W^2)$ steps

Gradient descent optimization

- Choose weight update to go in the direction of negative gradient

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

- $\eta > 0$ is learnıng rate

- Since error function is defined w.r.t to training set, the entire training set has to be processed for each step to evaluate ∇E
- All data is used <-- Batch process / method
- Gradient descent / steepest descent => decrease of error function
- This approach is not efficient

- Solution: On line gradient descent (stochastic gradient descent / sequential gradient descent)
- One data point at a time.
- Error function $$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$
- Weight update $$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$
- Cycle through data in sequence or pick in random or batches of data points

- Approach handles redundancy in data and avoids local minimum (a little)

Error Backpropagation

- Minimization of error function: Has two stages -->
- Stage 1: Calculate derivatives of error function w.r.t. Weights
- Stage 2: Use derivatives to adjust weight values

- Evaluation of error function derivatives:
- Error functions contain a sum of terms

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- Evaluate $\nabla E_n(\mathbf{w})$ for one term ->

- Case 1: outputs are linear combinations of input variables

$$y_k = \sum_i w_{ki} x_i$$

- Error functions: $$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

- Gradient of error function w.r.t weight w_{ji} $$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}$$
- This is -> product of an error signal of w_{ji} and input
- In a feed forward network every unit is $$a_j = \sum_i w_{ji} z_i$$
- z_i Is activation

- Biases -> add an extra unit with activation fixed as +1
- These sums are transformed by nonlinear activation function h(.) resulting in activation z_j $z_j = h(a_j)$
- Forward propagation: Successively apply weighted sum of inputs and activation using the input vector to all hidden and o/p units
- Evaluate derivative of E_n w.r.t w_{ji}
- E_n depends on weight w_{ji} only
- Differentiating error w.r.t. weight provides the change in weight to e backpropagated
- New weight = old weight + differential of error w.r.t weight

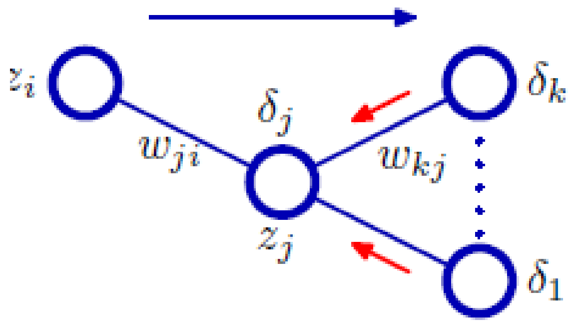
- Define $$\delta_j \equiv \frac{\partial E_n}{\partial a_j}$$
- δ's Are errors
- Using $$a_j = \sum_i w_{ji} z_i$$ gives $$\frac{\partial a_j}{\partial w_{ji}} = z_i$$
- Evaluate E_n w.r.t weight w_{ji} As E_n depends on weight w_{ji} only, apply chain rule for partial derivative
$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$
- Substituting above terms: $$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$$

- Required derivate is got by multiplying value of δ for the nit at o/p end by value of 'Z' at input end
- This is similar to linear model

- For o/p units $\delta_k = y_k - t_k$

- To evaluate δ for hidden units
- Unit 'j' sends to unit 'k'
-
-
-

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$



Blue is forward

Red is backpropagation

- <u>Backpropagation formula: </u>(substitute δ use $a_j = \sum_i w_{ji} z_i$
- and $z_j = h(a_j)$

- We get $\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$

- <u>Error backpropagation</u>
- 1) Apply I/p vector to Xn to network and forward propagate using
- $a_j = \sum_i w_{ji} z_i$ And $z_j = h(a_j)$

2)Evaluate δ_k for all outputs using $\delta_k = y_k - t_k$

3) Backpropagate δ using $\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$

4) evaluate using $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$

Regularization in NN

- Dimensionality of data decides number of inputs and outputs in NN
- # of Hidden units M -> determines predictive performance
- M gives number of weights and biases in network

=> in a maximum likelihood setting -> optimum value for M=> best generalization performance

1) Choose M is to plot graph of M with different (initial) weight vectors

2) Start with large value for M and then regularize

Simplest regularizer: Quadratic ->

Regularized error =

<u>(weight decay)</u>

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$$

- Regularization co-efficient : λ

- Weight decay is also identified as negative logarithm of zero-mean Gaussian prior distribution over weight vector 'w'
- Problem with using weight decay:
- Weight decay: issue of inconsistency when the network needs to scale up.
- Linear transformation of input data where x_i becomes ax_i+b will not be handled by weight decay.
- Sol: Consistent Gaussian priors
- In above case weights will need to be similarly transformed, linearly

- w_{ji} becomes (1/a) w_{ji}

<u>Output data</u>: Linear transformation of o/p is, y_k becomes cy_k+d

2nd layer weights w_{kj} become cw_{kj}

Consistency: One network trained with original data and another with transformed data should result is equivalent neural networks (with weights differing as above)

Regularizer should not change due to re-scaling of weights and shifts in biases. =>

$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2$$

W_1 -> set of weights in first layer W_2 -> set of weights in second layer

- This will maintain consistency as long as the regularization parameters are rescaled using $\lambda_1 \to a^{1/2}\lambda_1$ $\lambda_2 \to c^{-1/2}\lambda_2$

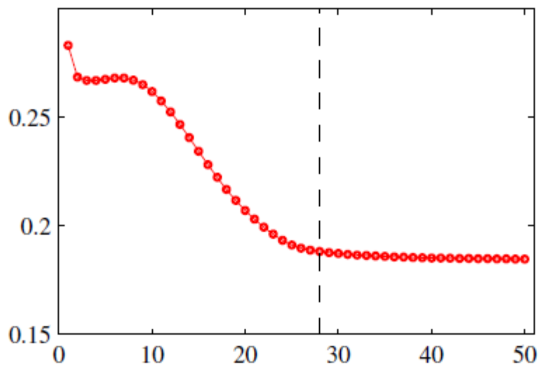
- Regularizer (previous page) has Prior $p(\mathbf{w}|\alpha_1,\alpha_2) \propto \exp\left(-\frac{\alpha_1}{2}\sum_{w\in\mathcal{W}_1} w^2 - \frac{\alpha_2}{2}\sum_{w\in\mathcal{W}_2} w^2\right)$

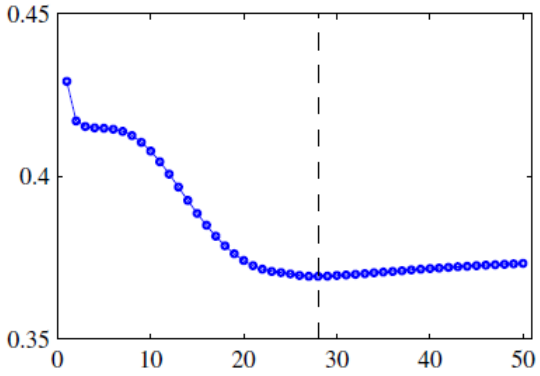
- This Prior cannot be normalized (improper) -> Bias parameters are not constrained
- Sol: Separate priors for biases with own hyperparameters
- Generalizing: Consider Priors where weights are divided into W_k groups
- Then with: $\|\mathbf{w}\|_k^2 = \sum_{j\in\mathcal{W}_k} w_j^2.$ prior becomes $p(\mathbf{w}) \propto \exp\left(-\frac{1}{2}\sum_k \alpha_k \|\mathbf{w}\|_k^2\right)$

Early Stopping

- In certain cases of data and weight values the error decreases for a time, on the training data. But on validation data the error may actually increase.
- Better to stop training at a point where the error for training and validation data will be together lowest. => Generalize better
- Basically model is becoming too complex -> overfitting

- Training set





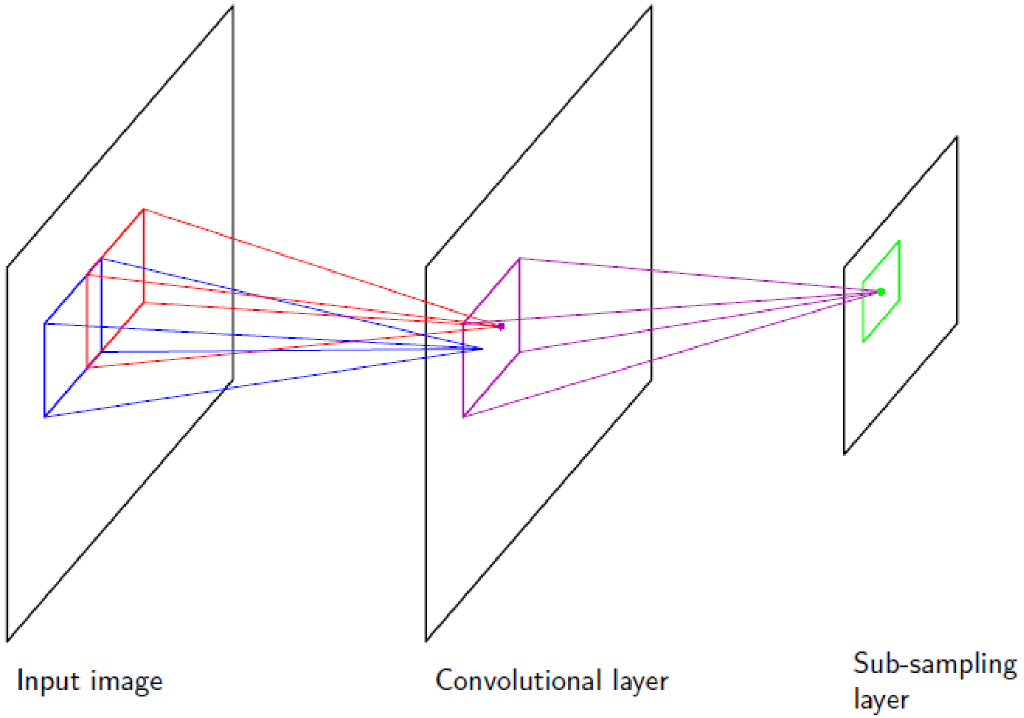
Validation set

Invariances

- Prediction should be independent (invariant) of certain changes of data -> translational invariance and size invariance
- Solution: Large training data with all position and size options <- not practical
- Alternatives: 1) Augment training set with transformed data
- 2) Add regularization that discourage change in o/p even when there are transformational changes In I/p -> one technique is tangent propagation
- 3) Extract the features that are invariant under the transformations. (difficult to find features)
- 4) Structure of neural network includes invariance properties (CNN - local receptive fields and shared weights)

- Approach 4: Build invariance properties into neural network: Convolutional neural networks: Recognize images
- Handwritten digits: I/p image is pixel intensity values, O/p is posterior probability over ten digit classes (0-9)
- Digit identity will be invariant over scaling and translations.
- Also handle small rotations and elastic deformations
- Property to be used: Nearby pixels are strongly correlated
- => Extract local features (local regions)
- Merge information from local features to detect higher order features in next stage of processing and finally identify image.

- CNN structure:



- Three modules (mechanisms) : Local receptive fields, Weight sharing and Subsampling

- In convolutional layer units are feature maps (different planes)
- Units in one feature map take inputs from a subregion. All units in one feature map share the same weight values.
- All units in a feature map detect same pattern, but at different locations of I/p image
- Evaluating the activation of these units is: A convolution of image pixel intensities with a kernel of weight parameters (made possible due to the weight sharing)
- Approach handles "shifting" of I/p, by correspondingly changing activations of feature map. => invariance to translations and distortions of I/p image

- O/p from convolutional layer given to subsampling layer.
- Each Feature map -> plane of units in subsampling layer
- Subsampling : Pooling / average of I/p * adaptive weight + adaptive bias -> sigmoidal nonlinear activation function
- Subsampling will have lesser rows and columns compared to convolutional layer (even half)
- => Units in subsampling layer less responsive to small shifts of image

- In a CNN -> Many feature maps, => gradual reduction in spatial resolution is compensated by increasing number of features
- Final (o/p) layer : fully connected, fully adaptive with softmax output

Training CNN

- Error minimization using back propagation:
- Small change in algorithm to use shared weights -> Lesser weights to be learnt and

Soft weight sharing

- Sharing weights (weights are forced to be equal within group) can only be used when the constraints can be specified at the beginning
- Alternative: Soft weight sharing: Not necessarily equal but 'Similar' values for every group of weights
- Learning process includes: groups of weights, mean weight value for each group, spread of values within group
- Weight decay regularizer $\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$ is a negative log of Gaussian prior distribution
- Instead consider a mixture of Gaussians: Here the weight values can form different (more than one) groups

- Takin Mixing coefficients as π_j
- gives $$p(w_i) = \sum_{j=1}^{M} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2)$$
- Probability density will be $$p(\mathbf{w}) = \prod_i p(w_i)$$
- Negative logarithm of this gives regularization function

$$\Omega(\mathbf{w}) = -\sum_i \ln \left(\sum_{j=1}^{M} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \right)$$

- Total error function is = $\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\Omega(\mathbf{w})$

- Error is minimized w.r.t weights w_i and w.r.t. parameters $\{\pi_j, \mu_j, \sigma_j\}$
- Weights are learnt continuously
- To avoid numerical instability, joint optimization is done over weights and 'mixture model' parameters
- <u>Evaluate derivative of error w.r.t. Parameters:</u>
- Take $\{\pi_j\}$ as prior probability. Add corresponding posterior probabilities (using Bayes theorem)
- Gives:

$$\gamma_j(w) = \frac{\pi_j \mathcal{N}(w|\mu_j, \sigma_j^2)}{\sum_k \pi_k \mathcal{N}(w|\mu_k, \sigma_k^2)}$$

- Derivatives of error w.r.t weights is $\frac{\partial \widetilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda \sum_j \gamma_j(w_i) \frac{(w_i - \mu_j)}{\sigma_j^2}$

- Derivatives of error w.r.t. centers of Gaussian
- This moves μ_j towards average of weight values, weighted by posterior probabilities of each weight being generated by component 'j'
- =>

$$\frac{\partial \widetilde{E}}{\partial \mu_j} = \lambda \sum_i \gamma_j(w_i) \frac{(\mu_i - w_j)}{\sigma_j^2}$$

- Derivatives of error w.r.t Variances: Moves σ_j towards weighted average of square of derivation of weights around corresponding center μ_j . Weighting coefficients are again, the posterior probability of each weight being generated by component 'j'

- Derivatives of error w.r.t mixing coefficients π_j
- Taking π_j as prior probabilities results in the constraint $\sum_j \pi_j = 1$
- We have to use softmax function and variables $\{\eta_j\}$
- π_j Becomes $\frac{\exp(\eta_j)}{\sum_{k=1}^{M}\exp(\eta_k)}$
- Derivatives of error function w.r.t $\{\eta_j\}$ becomes
- $$\frac{\partial \widetilde{E}}{\partial \eta_j} = \sum_i \{\pi_j - \gamma_j(w_i)\}$$
- π_j moves to average posterior probability for the component 'j'