Training Deep Nets: Gradient Descent and Regularization

Lecturer: Raghu Krishnapuram
Distinguished Member of Technical Staff
Robert Bosch Centre for Cyber-Physical Systems
kraghu@iisc.ac.edu

- **Deep Feedforward Networks**

- Also known as multilayer perceptrons (MLP)
- The goal is to approximate a function (e.g., a classifier)
- $\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{\theta})$ approximates a desired function by tuning $\boldsymbol{\theta}$
- There are no feedback loops
  - Feedback is used in recurrent networks

- **Deep Feedfordward Contd...**

- The output is a composition of the form

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}))) \text{ (chain structure)}$$

- The bottom layer is called the input layer, the top layer is the output layer

- **Deep Feedfordward Contd...**

- In function approximation, the (noisy) training data does not specify values at the intermediate nodes
  - Hence "hidden layers"
- Dimensionality of the hidden layers is the "width" of the model
  - (no. of units that operate in parallel)
- What is "neural"?
  - Layers of vector valued functions and the activation functions

- **Relation to Kernel Methods**

- Kernel methods transform $\boldsymbol{x}$ to $\boldsymbol{\phi}(\boldsymbol{x})$ to achieve more complex mappings
- We can
  - (i) Use a higher dim $\boldsymbol{\phi}$ (e.g., RBF kernel)
    or
  - (ii) manually engineer $\boldsymbol{\phi}$ if we have domain knowledge
    or
  - (iii) specify only a broad class of functions and leave parameters
    - e.g., $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$

- These general principles can also be used for learning stochastic mappings, functions with feedback, probability distributions, etc
  - Example: Learning XOR
- Toy example with four training vectors

$$\chi = \{\boldsymbol{x_1} = [0,0]^T, \boldsymbol{x_2} = [0,1]^T, \boldsymbol{x_3} = [1,0]^T, \boldsymbol{x_4} = [1,1]^T\}$$

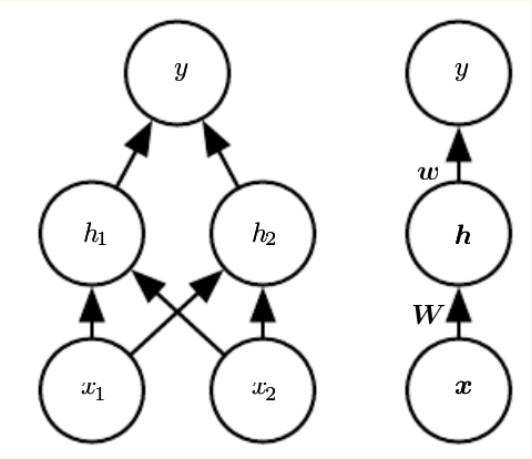
- The output is 1 for $\boldsymbol{x_2}$ and $\boldsymbol{x_3}$

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{x\in\chi}(f^*(\boldsymbol{x}) - f(\boldsymbol{x},\boldsymbol{\theta}))^2$$

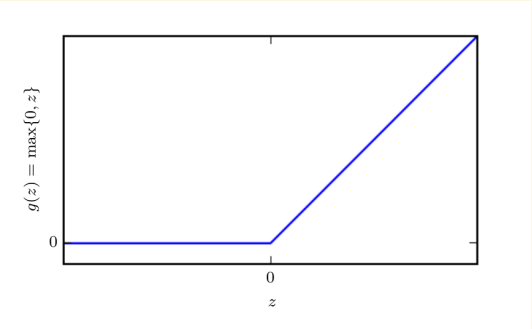
- $\boldsymbol{\theta}$ are the parameters to learn and f^* is the target function

- If we use $f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$, we get the solution $\boldsymbol{w} = 0$ and $b = 1/2$
- Use a feedforward network with a single hidden layer:

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, c) \text{ and } y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$$



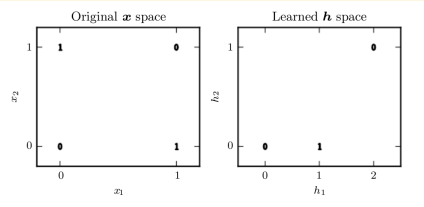
- $f^{(1)}$ cannot be linear. Why? So use:
  - $\boldsymbol{h} = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c})$
- In modern neural networks, we use the "rectified linear unit" (ReLu), ie., $g(z) = \max\{0, z\}$



$$f(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

- We can see that for the follwing values, the solution will work

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}; \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}; \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}; b = 0$$



- **Gradient Based Learning**

- Non-linearity of a neural network means that they are usually trained by iterative, gradient-based approaches
- Solvers such as those used for SVM or logistic regression are not feasible
- Most algorithms are improvements on the stochastic gradient descent algorithm
- Initialization is very important for proper convergence. Typically small weight values are randomly chosen

- **Cost Functions**

- Most commonly, we use the cross-entropy between the training data and the model's predictions as the cost function
- Regularization is very important.
  - Remember "λ" in regression that controls the magnitude of the weights. Larger λ implies higher "weight decay"

- **Learning Conditional Distributions with ML**

- Negative log-likelihood or cross-entropy

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{data}} \log p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x})$$

- If $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x},\boldsymbol{\theta}), \boldsymbol{I})$, then we have a least squares type of formulation

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}}||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{ const}$$

- The log functions also helps in mitigating issues related to "vanishing gradients"

- **Output Units**

- The choice of the cost function is linked with the choice of the output unit
- Assume that a feedforward network provides a set of hidden features defined by

$$\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- These hidden features transformed by the output units.

- **Linear Units for Gaussian Output Distributions**

- $\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$ (affine transformation)
- They are used to produce the mean of a conditional distribution

$$p(\boldsymbol{y}|\hat{\boldsymbol{y}}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$$

- Covariance is generally ignored and the mean-squared error is minimized
- Note that squared error between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ is minimized by Maximum Likelihood.

- **Sigmoid Units for Bernoulli Output Distributions**

- We need a smooth function that can approximate a Bernoully conditional distribution
- Let $\hat{y} = \sigma\underbrace{(\boldsymbol{w}^T\boldsymbol{h} + b)}_{z}$
- Justification:

- **Sigmoid Units Contd...**

- Justification:

$$\begin{aligned}
&\text{Let } \log \tilde{P}(y) = yz \quad (\tilde{P}(y) \text{ is the unnormalized probability}) \\
&\implies \tilde{P}(y) = \exp(yz) \\
&\implies P(y) = \frac{\exp(yz)}{1+\exp(z)} = \frac{1}{1+\exp(-(2y-1)z)} : (\text{ for } y = 0 \text{ or } 1) \\
&\qquad\quad = \sigma((2y-1)z)
\end{aligned}$$

- Where σ is a logistic sigmoid

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

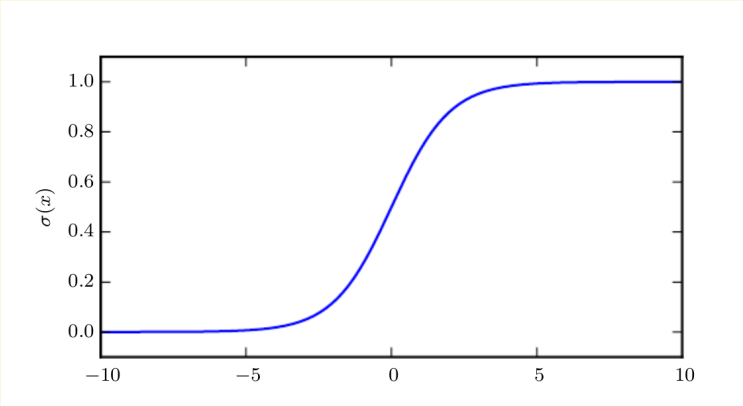
- The z variable defining a distribution over binary variables is called a "logit"

- The log function for maximum likelihood learning becomes

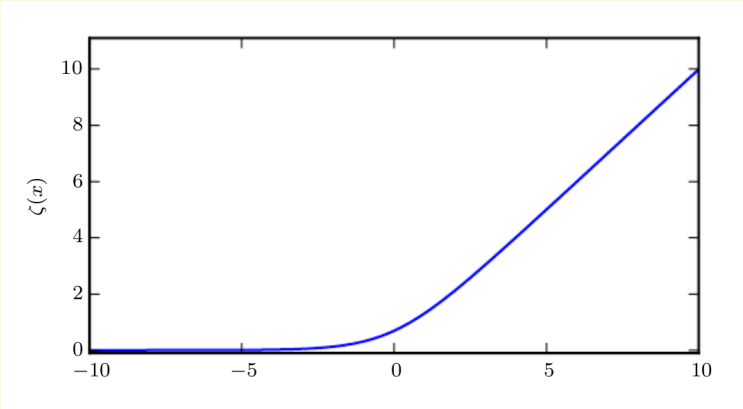
$$\begin{aligned} J(\boldsymbol{\theta}) &= -\log P(y|\boldsymbol{x}) = -\log \sigma((2y-1)z) \\ &= \zeta((1-2y)z) \qquad \text{(softplus function)} \end{aligned}$$

- The log undoes the exp of the sigmoid and prevents saturation of the gradient

☛ **Logistic Sigmoid Function**



☛ **Softplus Function**



- The gradient saturates when $y = 1$ and z is highly positive or $y = 0$ and z is highly negative, i.e., when the model already has the right answer.
- When z has the wrong sign, $(1 - 2y)z) \approx |z|$, and the gradient does not shrink.
- Cross entropy loss can be defined as $-\sum_i (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$
- For other loss functions, loss can saturate when $\sigma(z)$ saturates, and learning may cease. Hence, maximum likelihood should be preferred for sigmoid output function.
- Avoid computing $\log(\sigma())$ to overcome numerical issues.

- **Softmax units for multinoulli output distributions**

- Generalization of the sigmoid function to the multinomial case
- Produce $\hat{\boldsymbol{y}}$ with $\hat{y}_i = P(y = i|\boldsymbol{x}),\ \sum_i \hat{y}_i = 1$
- Use a linear layer $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$, where

$$z_i = \log P(y = i|\boldsymbol{x})_i, \text{ followed by}$$

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_i)}$$

- Behaves well when we use maximum log-likelihood

$$\log \text{ softmax } (\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$$

- First term does not saturate. Summation in the second term is determined by $\exp(\underbrace{\max}_{j} z_j)$, as it is $\approx max_j z_j$
- The correctly classified examples will not contribute, only others will.

- Overall, softmax layer outputs will tend to

$$\text{softmax}\left(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta})\right)_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i, \boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}$$

- Softmax without the log will saturate when an input is maximal or when it is close to zero. Hence not useful for other loss functions such as MSE.
- Softmax is similar to the "winner-take-all" behaviour, which occurs naturally in the cortex.
- Can also fix one of the inputs to 1, because of the constraint $\sum_i \hat{y} = 1$. However, not useful in practice.

- **Logistic sigmoid and hyperbolic tangent**

$$g(z) = \sigma(z)$$
$$g(z) = \tanh(z) = 2\sigma(2z) - 1$$

- Strongly sensitive only when z is close to zero

- Used in output layers when an appropriate cost function can undo the saturation
- The $\tanh$ function is closer to the identity function near zero, hence resembles training linear models

$$\hat{y} = \boldsymbol{w}^T \tanh(\boldsymbol{U}^T \tanh \boldsymbol{V}^{\boldsymbol{T}} \boldsymbol{x}) \approx \boldsymbol{w}^T \boldsymbol{U}^T \boldsymbol{V}^T \boldsymbol{x} \qquad \text{(when activations are small)}$$

- **Linear Units**

- Although consecutive layers that have no non-linear activation functions can be collapsed, if one or more layers have a small number of nodes, there might be savings in the number of parameters. (Think of logic circuits.)
- Also useful to reduce features (PCA Analogy)
- Softmax units are useful as a way to learn to "manipulate memory"

- **Other functions**

- Radial Basis Function (RBF) is useful for "focussed activation"
  - Difficult to train due to vanishing gradients
- Softplus: $g(a) = G(a) = \log(1 + e^a)$ is a smooth version of the rectifier, but generally nor preferred
- Hard $\tanh$: $g(a) = \max(-1, \min(1, a))$ (Collobert 2014)

- **Architecture Design**

- Architecture is defined by the overall structure
  - No. of layers, type and number of units in each layer, connectivity, etc.
- Layered architectures (chain based), require the specification of depth and width.
- In theory, single hidden node is sufficient, but in practice, deeper nets may be able to use fewer nodes.
- Deeper nets generally harder to train.

- **Universal approximation properties**

Universal Approximation Theorem, by Hornick et al, 1989; Cybenko, 1989 -
A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided the network is given enough hiddden units.

- Also true for a wider class of activation functions (e.g. ReLu)
- UAT only guarantees representability and not learnability
- Learning may fail due to the training algorithm
  - Optimiation may fail or overfitting may occur

- UAT does not say what the size of the network should be. However, we may need an exponential number of hidden units (Barron 1993)
- Example: binary case - 2^{2^n} functions with n variables
  - May need 2^n degreed of freedom

- **Other Architectural Considerations**

- Skip connections are used sometimes for specific tasks. This helps faster flow of gradients while training
- Other forms of networks such as recurrent networks and convolutional networks will have other considerations
- Strategies for reducing the number of parameters (e.g., do not fully connect layers) are important.

- **Backpropagation and Other Differentiation Algorithms**

- Forward propagation: Compute output $\hat{\boldsymbol{y}}$ given input $\boldsymbol{x}$ by propagating values through the hidden nodes
- Backpropagation: A method of propagating the error (cost) from the output layer down to the lower layers. This mechanism is used in a learning algorithm.
- Computing the gradient w.r.t. all parameters of the network can be expensive, even though analytically tractable.

- **Chain Rule of Calculus**

- Let $y = g(x)$ and $z = f(y) = f(g(x))$. Then:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- By extension, let $\boldsymbol{x} = \mathbb{R}^m$ and $\boldsymbol{y} = \mathbb{R}^n$ and g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f maps from $\mathbb{R}^n$ to $\mathbb{R}$, i.e., $\boldsymbol{y} = g(\boldsymbol{x}), z = f(\boldsymbol{y})$ then (contd...)

- **Chain Rule Contd...**

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{dy_j}{dx_i}$$

$$= \begin{bmatrix} \frac{\partial y_1}{\partial x_i} \\ \vdots \\ \frac{\partial y_n}{\partial x_i} \end{bmatrix}^T \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{bmatrix}$$

More generally,

$$\nabla_{\boldsymbol{x}} z = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_i} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & & \vdots & & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_i} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}^T \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_n} \end{bmatrix}$$

$$= \underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T}_{\text{Jacobian}} \nabla_{\boldsymbol{y}} z$$

Alternative notation: $\frac{dz}{d\boldsymbol{x}} = \frac{d\boldsymbol{y}}{d\boldsymbol{x}} \frac{dz}{d\boldsymbol{y}}$ (Note change of order)

- In general, we may need the gradient of a variable z w.r.t. a tensor $\mathbf{X}$, i.e., $\nabla_{\mathbf{X}} z$. For example, $\frac{d}{d\boldsymbol{W}} z$, where $\boldsymbol{W}$ is a matrix. We can arrange the elements of the tensor in a vector form and use the above formula.
- Alternatively we can write

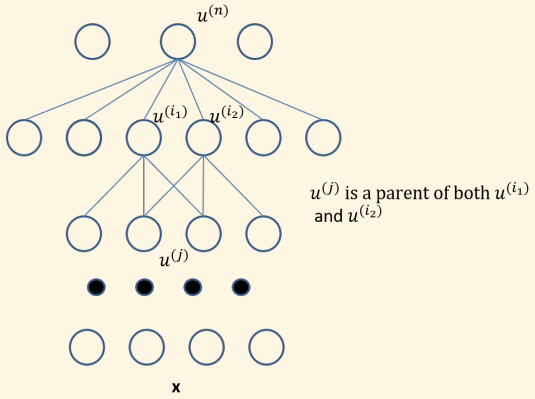
$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} \mathbf{Y}_j) \frac{\partial z}{\partial \mathbf{Y}_j}$$

  - Where j represents a tuple of indices
- Generally, many sub-expressions repeat when we use the chain rule, so there is a trade-off between storing such values and recomputing them

- **Backpropagation at a high level**

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in P_a(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

$$= \underbrace{\sum_{i:j \in P_a(u^{(i)})} \text{grad-table } [u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}}_{\text{computation } \propto \text{ fanout of } u^{(j)}}$$



- Assume that the grad-table[.] values are stored in a table

- **Observations**

- Assuming that the derivatives such as $\partial u^{(n)}/\partial u^{(i)}$ are all stored, they need to be computed only once.
- The number of computations needed for backpropagation (which eventually computes the derivative of every output node wrt all other modes) scales linearly with the number of edges.
- If storage is an issue, there are alternative algorithms that repeat.

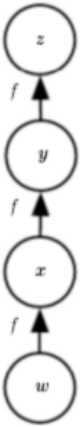


Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient. Let $w \in \mathbb{R}$ be the input to the graph. We use the same function $f : \mathbb{R} \to \mathbb{R}$ as the operation that we apply at every step of a chain: $x = f(w)$, $y = f(x)$, $z = f(y)$.

- Let $x = f(w), y = f(x), z = f(y)$. Then

$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\ &= f'(y)f'(x)f'(w) \leftarrow \text{ stores gradients} \\ &= f'(f(f(w)))f'(f(w))f'(w) \leftarrow \text{ recomputes gradients}\end{aligned}$$

- **Forward Propagation through a typical deep network**

Require: Network depth l

Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model

Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model

Require: $\boldsymbol{x}$, the input to process

Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$

for $k = 1, \ldots l$, do

$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$

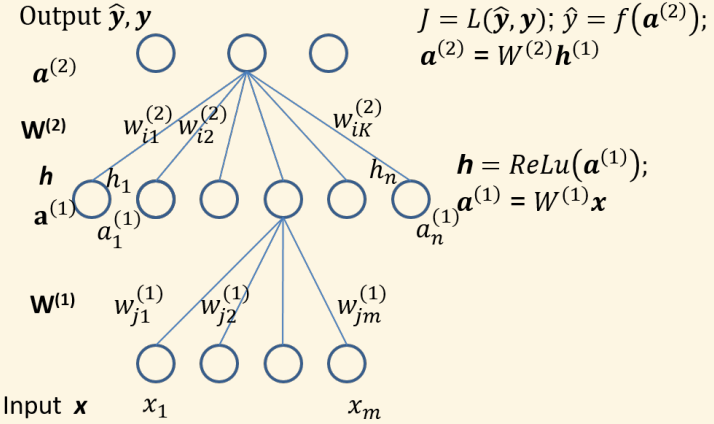
$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$

end for

$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$

$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\boldsymbol{\theta})$

☛ **Illustration of notation for a single hidden layer**



- **Backpropagation through a typical deep network**

Compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert gradient above into gradient pre-nonlinearity activation

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \bigodot f'(\boldsymbol{a}^{(k)})$ (element-wise multiplication)

Compute gradients on weights and biases

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{b^{(k)}} \Omega(\boldsymbol{\theta})$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}(\boldsymbol{h}^{(k-1)})^T + \lambda \nabla_{W^{(k)}} \Omega(\boldsymbol{\theta})$

Propagate the gradients wrt next lower layer's activations

$\boldsymbol{g} \leftarrow \nabla_{h^{(k-1)}} J = (\boldsymbol{W}^{(k)})^T \boldsymbol{g}$

end for

- **Details of computation - Case 1: One hidden layer and single output node.**

Here, $W^{(2)}$ becomes a vector $w^{(2)}$.

$$\frac{dJ}{d\hat{y}} = L'(\hat{y})$$

$$\frac{dJ}{d\boldsymbol{h}} = \frac{da^{(2)}}{d\boldsymbol{h}} \cdot \frac{d\hat{y}}{da^{(2)}} \cdot \frac{dJ}{d\hat{y}} = \boldsymbol{w}^{(2)} f'(a^{(2)}) L'(\hat{y})$$

$$\frac{dJ}{d\boldsymbol{w}^{(2)}} = \frac{da^{(2)}}{d\boldsymbol{w}^{(2)}} \cdot \frac{d\hat{y}}{da^{(2)}} \cdot \frac{dJ}{d\hat{y}} = \boldsymbol{h} f'(a^{(2)}) L'(y)$$

- $\frac{dJ}{d\boldsymbol{W}^{(1)}}$ and $\frac{dJ}{d\boldsymbol{x}}$ are computed using back propagation

$$\frac{d\boldsymbol{h}}{d\boldsymbol{a}^{(1)}} = \begin{bmatrix} \frac{\partial h_1}{\partial a_1^{(1)}} & \cdots & \frac{\partial h_n}{\partial a_1^{(1)}} \\ \vdots & & \vdots \\ \frac{\partial h_1}{\partial a_n^{(1)}} & \cdots & \frac{\partial h_n}{\partial a_n^{(1)}} \end{bmatrix} = \begin{bmatrix} 0/1 & 0 & \dots & 0 \\ 0 & 0/1 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0/1 \end{bmatrix} = \boldsymbol{R}(\boldsymbol{a}^{(1)}), \text{ say}$$

Note: $\frac{\partial h_1}{\partial a_1^{(1)}}$ is 0 or 1 depending on the value of $a_1^{(1)}$, and so on.

- We need to compute $\frac{d\boldsymbol{a}^{(1)}}{d\boldsymbol{W}^{(1)}}$ and $\frac{d\boldsymbol{a}^{(1)}}{d\boldsymbol{x}}$
- Both are tensors

$$\frac{d\boldsymbol{a}^{(1)}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{\partial a_1^{(1)}}{\partial x_1} & \cdots & \frac{\partial a_n^{(1)}}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial a_1^{(1)}}{\partial x_m} & \cdots & \frac{\partial a_n^{(1)}}{\partial x_m} \end{bmatrix} = \begin{bmatrix} w_{11}^{(1)} & \cdots & w_{n1}^{(1)} \\ \vdots & & \vdots \\ w_{1m}^{(1)} & \cdots & w_{nm}^{(1)} \end{bmatrix} = (\boldsymbol{W^{(1)}})^T$$

$$\nabla_{\boldsymbol{x}} \boldsymbol{h} = \frac{d\boldsymbol{h}}{d\boldsymbol{x}} = \frac{d\boldsymbol{a}^{(1)}}{d\boldsymbol{x}} \frac{d\boldsymbol{h}}{d\boldsymbol{a}^{(1)}} = (\boldsymbol{W}^{(1)})^T \boldsymbol{R}(\boldsymbol{a}^{(1)})$$

- Column j of $\boldsymbol{W}^{(1)} \boldsymbol{R}(\boldsymbol{a}^{(1)})$ will be zero or non-zero depending on the value of $a_j^{(1)}$.
- $\frac{d\boldsymbol{a}^{(1)}}{d\boldsymbol{W}^{(1)}}$ is a 3-D tensor; with n matrices stacked on top of each other

- The i-th matrix is

$$\nabla_{\boldsymbol{W}^{(1)}} \boldsymbol{a}_{(i)}^{(1)} = \frac{d a_i^{(1)}}{d \boldsymbol{W}^{(1)}} = \begin{bmatrix} \frac{\partial a_i^{(1)}}{\partial \omega_{11}} & \cdots & \frac{\partial a_i^{(1)}}{\partial \omega_{1m}} \\ \vdots & \ddots & \\ \frac{\partial a_i^{(1)}}{\partial \omega_{n1}} & \cdots & \frac{\partial a_i^{(1)}}{\partial \omega_{nm}} \end{bmatrix} = \begin{bmatrix} 0 & \dots & 0 & \dots & 0 \\ & \dots & & \dots & \\ x_1 & \dots & x_j & \dots & x_m \\ & \dots & & \dots & \\ 0 & \dots & 0 & \dots & 0 \end{bmatrix}$$

- We can also arrange all of these matrices in the form of a long column vector of size $n^2 \times m$ and do the manipulations. However, a simpler expression is possible, because of the structure of the network. (See derivation for multiple output case.)

- **Case 2: There are K output nodes**

- Let $(\boldsymbol{w}_i^{(2)})^T$ be the i-th row of $\boldsymbol{W}^{(2)}$. Note that $\boldsymbol{w}_i$ is the i-th row of $\boldsymbol{W}^{(2)}$ represented as a column vector.
- Then $a_i^{(2)} = (\boldsymbol{w}_i^{(2)})^T \boldsymbol{h}$

$$\frac{da_i^{(2)}}{d\boldsymbol{w}_i^{(2)}} = \boldsymbol{h}; \ \frac{d\hat{y}_i}{da_i^{(2)}} = f'(a_i^{(2)})$$

$$\implies \frac{d\hat{y}_i}{d\boldsymbol{w}_i^{(2)}} = f'(a_i^{(2)})\boldsymbol{h}$$

$$\frac{dJ}{d\boldsymbol{W}^{(2)}} = \begin{bmatrix} \left(\frac{dJ}{d\boldsymbol{w}_1^{(2)}}\right)^T \\ \left(\frac{dJ}{d\boldsymbol{w}_2^{(2)}}\right)^T \\ \cdots \\ \left(\frac{dJ}{d\boldsymbol{w}_K^{(2)}}\right)^T \end{bmatrix} = \begin{bmatrix} \left(\frac{d\hat{y}_1}{d\boldsymbol{w}_1^{(2)}}\frac{dJ}{d\hat{y}_1}\right)^T \\ \left(\frac{d\hat{y}_2}{d\boldsymbol{w}_2^{(2)}}\frac{dJ}{d\hat{y}_2}\right)^T \\ \cdots \\ \left(\frac{d\hat{y}_K}{d\boldsymbol{w}_K^{(2)}}\frac{dJ}{d\hat{y}_K}\right)^T \end{bmatrix} = \begin{bmatrix} \boldsymbol{h}^T f'(a_1^{(2)}) L'(\hat{y}_1) \\ \boldsymbol{h}^T f'(a_2^{(2)}) L'(\hat{y}_2) \\ \cdots \\ \boldsymbol{h}^T f'(a_K^{(2)}) L'(\hat{y}_K) \end{bmatrix} \quad (1)$$

- Let $\boldsymbol{g} = \begin{bmatrix} f'(a_1^{(2)})L'(\hat{y}_1) \\ f'(a_2^{(2)})L'(\hat{y}_2) \\ \cdots \\ f'(a_K^{(2)})L'(\hat{y}_K) \end{bmatrix} = \begin{bmatrix} f'(a_1^{(2)}) \\ f'(a_2^{(2)}) \\ \cdots \\ f'(a_K^{(2)}) \end{bmatrix} \bigodot \begin{bmatrix} L'(\hat{y}_1) \\ L'(\hat{y}_2) \\ \cdots \\ L'(\hat{y}_K) \end{bmatrix} = f'(\boldsymbol{a}^{(2)}) \bigodot L'(\hat{\boldsymbol{y}}).$
- Then, from (1) we have:
- $\frac{dJ}{d\boldsymbol{W}^{(2)}} = \boldsymbol{g}\boldsymbol{h}^T$

☛ **Aside ...**

☛ Let $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}$

$$\begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} \begin{bmatrix} h_1 \\ \vdots \\ h_m \end{bmatrix}$$

$$\frac{d\boldsymbol{a}}{d\boldsymbol{h}} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} = \boldsymbol{W}^T$$

$\nabla_{\boldsymbol{h}} J = \frac{d\boldsymbol{a}}{d\boldsymbol{h}} \frac{dJ}{d\boldsymbol{a}} = \boldsymbol{W}^T \boldsymbol{g}$, where $\boldsymbol{g}$ is the gradient vector being backpropagated from the top.

- **Multiple Input-Output Pairs**

- Each row of $\boldsymbol{X}$ is a training vector, and there are t training vectors. Other matrices are defined similarly.
- $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2, \ldots, (\boldsymbol{x}_t, \boldsymbol{y}_t)\}$

$$X = \begin{bmatrix} \leftarrow \boldsymbol{x}_1^T \rightarrow \\ \leftarrow \boldsymbol{x}_2^T \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{x}_t^T \rightarrow \end{bmatrix} \quad \boldsymbol{Y} = \begin{bmatrix} \leftarrow \boldsymbol{y}_1^T \rightarrow \\ \leftarrow \boldsymbol{y}_2^T \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{y}_t^T \rightarrow \end{bmatrix} \quad \hat{\boldsymbol{Y}} = \begin{bmatrix} \leftarrow \hat{\boldsymbol{y}}_1^T \rightarrow \\ \leftarrow \hat{\boldsymbol{y}}_2^T \rightarrow \\ \vdots \\ \leftarrow \hat{\boldsymbol{y}}_t^T \rightarrow \end{bmatrix} = f(A^{(2)})$$

$$\boldsymbol{A}^{(2)} = \begin{bmatrix} \leftarrow (\boldsymbol{a}_1^{(2)})^T \rightarrow \\ \leftarrow (\boldsymbol{a}_2^{(2)})^T \rightarrow \\ \vdots \\ \leftarrow (\boldsymbol{a}_t^{(2)})^T \rightarrow \end{bmatrix} \quad \boldsymbol{H} = \begin{bmatrix} \leftarrow \boldsymbol{h}_1^T \rightarrow \\ \leftarrow \boldsymbol{h}_2^T \rightarrow \\ \vdots \\ \leftarrow \boldsymbol{h}_t^T \rightarrow \end{bmatrix}$$

$$\boldsymbol{A}^{(2)} = \boldsymbol{H}\boldsymbol{W}^{(2)}$$

$$\boldsymbol{H} = \mathsf{ReLu}(\boldsymbol{A}^{(1)})$$

$$\boldsymbol{A}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)}$$

IMPORTANT NOTE: In the above equations, the weight matrices are transposed versions of the ones we discussed earlier.

- If there are multiple units in the output layer, and for a training data set with t input-output pairs, then the expression for $dJ/d\boldsymbol{W}^{(2)}$ can be generalized to

$$\frac{dJ}{d\boldsymbol{W}^{(2)}} = \underset{n\times t}{\boldsymbol{H}^T}\,\underset{t\times n}{\boldsymbol{G}'}$$

- Each column of $\boldsymbol{H}^T$ is the output of the hidden layer for a training vector
- Each row of $\boldsymbol{G}'$ is a gradient vector evaluated at a training vector and arranged as a row.

- **Complexity of Backpropagation**

- Computing a gradient in a graph with n nodes will be $O(n^2)$ (worst case) in terms of number of operations in the graph - why?
- The computational graph is a directed acyclic graph, hence has at the most $O(n^2)$ edges
- Forward pass will at worst execute all n nodes
- Backpropagation adds one Jacobian vector product per edge

- The naive approach can lead to an exponential number of computations since

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path } (u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_k - 1)}}$$

- The number of paths from $u^{(n)}$ tp $u^{(j)}$ can be exponential in terms of the difference of depth between them.

- Practical implementations are not so straightforward, because they need to consider:
  - returning multiple values, providing access to tunable parameters,
  - managing memory consumption
  - manipulating many datatypes
  - handling functions whose gradients are not defined at certain points
  - how to simplify derivatives in automatic differentiation, and
  - how to do matrix-vector products efficiently, including order

- **Regularization for Deep Learning**

- "Regularization is any modification to a learning algorithm (typically the objective function) that is intended to reduce its generalization error, but not its training error"
- Typically this takes the form of hard or soft constraints on the parameter values
- Ensemble methods are also a form of regularization
- Variance-bias dilemma
- Regularization is primarily to reduce variance without increasing the bias

- **Parameter Norm Penalties**

- Regularized objective function:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

- $\alpha \in [0, \inf]$ is a hyperparameter controlling the influence of the soft constraint $\Omega()$
- Choice of $\Omega()$ leads to different solutions
- In neural nets, only weights are typically regularized, not the biases. (Biases do not induce too much variance)

- **L^2 Parameter Regularization**

- Also known as "weight decay", "ridge regression", and "Tichonov regularization"
- Adds $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2^2$ to the effective function

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- Update equation is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \boldsymbol{w} - \epsilon(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$
$$\boldsymbol{w} \leftarrow \underbrace{(1 - \epsilon\alpha)}_{\text{shrinkage factor}} \boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- Assume we can use a quadratic expansion of $J(\boldsymbol{w})$ around the optimum value, $\boldsymbol{w}^*$, of unregularized $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$

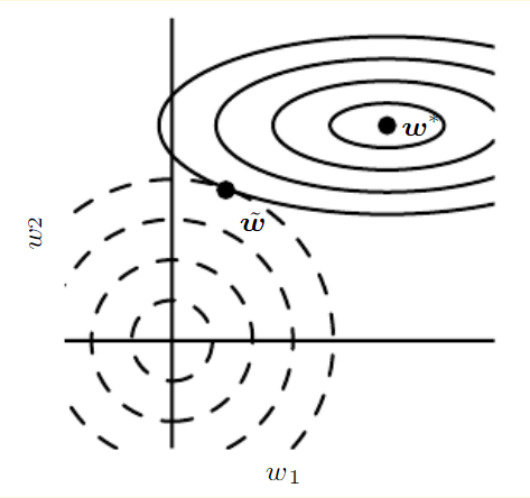
$$\hat{J}(\boldsymbol{w}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)\boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \text{ where } \boldsymbol{H} \text{ is the Hessian}$$

- Set the gradient to zero to find the optimum $\boldsymbol{w}$ denoted by $\tilde{\boldsymbol{w}}$.

$$\begin{aligned} & \nabla_{\boldsymbol{w}} \left(\frac{\alpha}{2} \boldsymbol{w}^T \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \right) = 0 \\ \implies & \alpha \tilde{\boldsymbol{w}} + H(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0 \\ \implies & \tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H} \boldsymbol{w}^* \end{aligned}$$

- Equivalent to adding extra values to the diagonal elements of $\boldsymbol{H}$. Provides more stability.
- Assuming $\boldsymbol{H}$ is diagonal with diagonal elements λ_i (we can use PCA to diagonalize it), we see that w_i for which $\lambda_i \gg \alpha$ are not affected, but w_i for which $\lambda_i \ll \alpha$ will be scaled down.

$$\tilde{\boldsymbol{w}} = \begin{bmatrix} \lambda_1/(\lambda_1+\alpha) & & \\ & \ddots & \\ & & \lambda_n/(\lambda_n+\alpha) \end{bmatrix} \boldsymbol{w}^* = \begin{bmatrix} \dfrac{\lambda_1 w_1^*}{\lambda_1+\alpha} \\ \vdots \\ \dfrac{\lambda_n w_n^*}{\lambda_n+\alpha} \end{bmatrix}$$



Geometric interpretation of L^2 regularization (in 2D) when the Hessian is diagonal

- For the case where J is the sum of all squared errors, we have the objective function

$$J(\boldsymbol{w}) = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha\boldsymbol{w}^T\boldsymbol{w}$$

- The solution is

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

- Adds more "variance" to features that have low variance, and shrinks the corresponding weights

- **L^1 Parameter Regularization**

- Also known as LASSO (Least Absolute Shrinkage and Selection Operator)
- L^1 uses: $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|$

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$
$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, y) = \alpha \text{ sign } (\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}, \boldsymbol{X}, y)$$

- sign $(\boldsymbol{w})$ is constant wrt w_i

- To gain insight, consider the case of the cost function being quadratic (or a truncated Taylor's series expansion) of $J(\boldsymbol{w})$ given by $\hat{J}(\boldsymbol{w})$
- $\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$
- Assuming that the Hessian, $\boldsymbol{H}$, is diagonal, we have

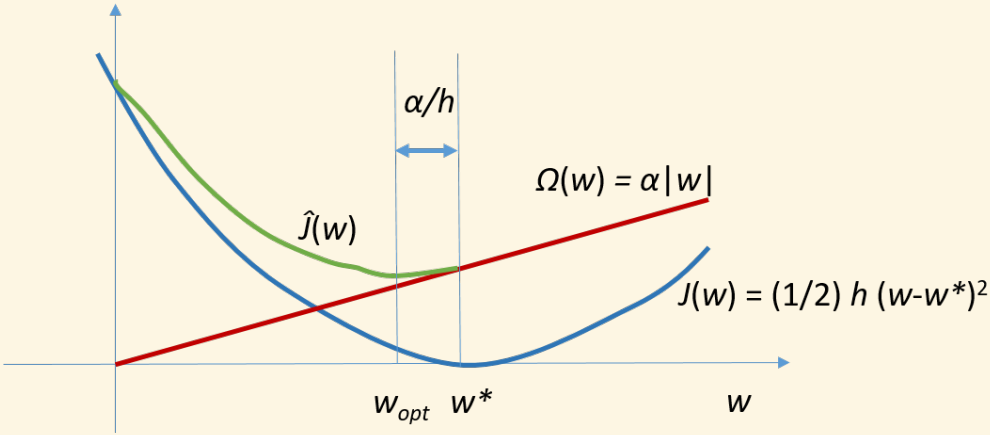
$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} h_{ii} (w_i - w_i^*)^2 + \alpha |w_i| \right]$$

- The solution is given by

$$\frac{d\hat{J}}{d\boldsymbol{w}} = 0 \text{ or } \sum_{i=1}^{n} [h_{ii}(w_i - w_i^*) + \alpha \text{sign}(w_i)] = 0$$

- This gives us n independent equations that can be solved

$$w_i = \text{ sign } (w_i^*) \cdot \max\left(|w_i| - \frac{\alpha}{h_{ii}}, 0\right)$$

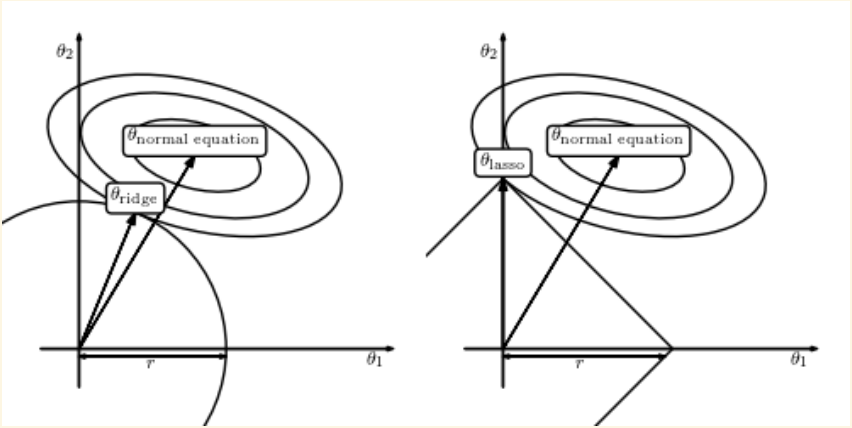


Geometric interpretation of L^1 regularization when the Hessian is diagonal

- **Explanation:**

- When $w_i^* > 0$ for all i
  - (i) $w_i^* \leq \dfrac{\alpha}{h_{ii}} \implies w_i = 0$. Here, the contribution of regularization is too high and the minimum occurs at zero. (see figure)
  - (ii) $w_i^* > \dfrac{\alpha}{h_{ii}} \implies w_i$ is moved towards zero by an amount α/h_{ii}
- Other cases are similar.
- When $\boldsymbol{H}$ is not diagonal, will need quadratic programming.

- In general, L^1 metric gives rise to sparse solutions relative to L^1, where many w_i are zero $\implies$ can be used to eliminate redundant features.

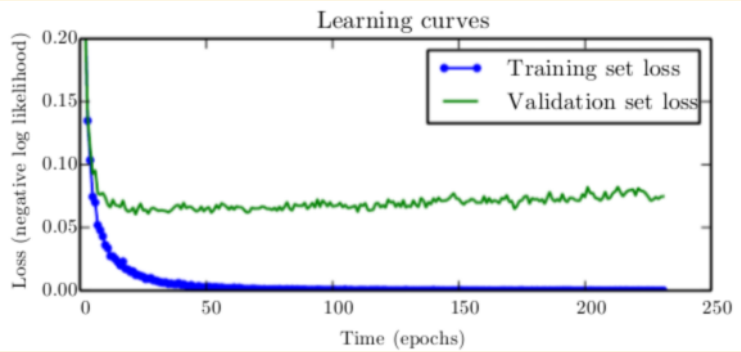


- **Dataset Augmentation**

- In general, we may not have sufficient data for problems that use networks with a very large number of parameters, especially to cover a large number of invariances.
- In some problems such as image classification, it is possible to generate variation, to cover the desired invariances.
- When translations, rotations, scaling, and other distortions used, need to be careful not to cause overlaps between data of different classes (e.g., "b" and "d" or "6" and "9" in character recognition).

- **Early Stopping**

- "Overtraining" generally results in "overfitting".
- Best to keep a validation set and observe the error on it rather than relying only on training error



- Generally better to revert back to parameters $\boldsymbol{\theta}$ that give a lower validation error $\implies$ early stopping.
- Train for a few iterations ("patience") and check on validation. If error is lower, retain new θ. Repeat this process till error is higher.
- Validation can be done separately.
- # of training iterations, i.e., "training time" i^*, can be treated as a hyperparameter. The ideal training time i^* can be determined using the process above.

- Early stopping is an unobtrusive way of regularization. Can be used with or without an additional regularization term.
- The validation set is not used in training, so it is in a sense wasted.
- Two strategies to use validation data, both are heuristic:
  - Use the early stopping method to determine i^*. Initialize all parameters randomly and retrain the network for i^* iterations.
  - Use the early stopping method to determine i^* and the trainign error ϵ. Add the validation data to training pool and train until the error falls below ϵ.

- It can be shown that early stopping with learning rate ϵ and number of iterations τ is equivalent to L^2 regularization where

$$\alpha \approx \frac{1}{\tau\epsilon}$$

- Advantage of early stopping over regularization is that τ can be easily determined using validation data in just one run.

- **Parameter Tying and Parameter Sharing**

- Based on domain knowledge, it is possible to force certain weights to be similar, using an additional regularization term, e.g.,

$$\Omega(\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}) = ||\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}||_2^2$$

- Where $\boldsymbol{w}^{(A)}$ and $\boldsymbol{w}^{(B)}$ are two sets of parameters that need to be similar.
- Convolutional networks use weight sharing by design.