Fast multiplication by the Hessian

Lecture **19**

Consider $\vec{v}^T H = v_j \nabla_j (\nabla_i E)$

Goal: calculate $\vec{v}^T H$ faster than $O(W^2)$ required to compute H.

Define $R\{\cdot\} = \vec{v}^T \vec{\nabla}$, note that

$$R\{w_k\} = v_j \underbrace{\nabla_j w_k}_{\delta_{jk}} = v_k \text{, or}$$

$$R\{\vec{w}\} = \vec{v}.$$

Two-layer network:

$$\begin{cases} a_j = \sum_i w_{ji} x_i, \\ z_j = h(a_j), \\ y_k = \sum_j w_{kj} z_j. \end{cases}$$

Now,
$$\begin{cases} R\{a_j\} = \sum_i R\{w_{ji}\} x_i = \sum_i v_{ji} x_i, \\ R\{z_j\} = h'(a_j) R\{a_j\}, \\ R\{y_k\} = \sum_j v_{kj} z_j + \sum_j w_{kj} \underbrace{R\{z_j\}}_{h'(a_j) R\{a_j\}} \end{cases}$$

Further,
$$\begin{cases} \delta_k = y_k - t_k, \\ \delta_j = h'(a_j) \sum_k w_{kj} \delta_k \quad \text{as before} \end{cases}$$

backpropagation

Then $\begin{cases} R\{\overset{\text{output}}{\delta_k}\} = R\{y_k\}, \\ R\{\underset{\text{hidden}}{\delta_j}\} = h''(a_j) R\{a_j\} \sum_k w_{kj} \delta_k + \end{cases}$

$$+ h'(a_j) \sum_k v_{kj} \delta_k + h'(a_j) \sum_k w_{kj} R\{\delta_k\}.$$

Finally, $\begin{cases} \dfrac{\partial E}{\partial w_{kj}} = \delta_k z_j, \\ \dfrac{\partial E}{\partial w_{ji}} = \delta_j x_i. \end{cases}$

$$\begin{cases} R\left\{ \dfrac{\partial E}{\partial w_{kj}} \right\} = R\{\delta_k\} z_j + \delta_k R\{z_j\}, \\ R\left\{ \dfrac{\partial E}{\partial w_{ji}} \right\} = x_i R\{\delta_j\}. \end{cases} \quad (*)$$

<u>Algorithm</u>: 1. Do a forward pass, compute $R\{a_j\}$, $R\{z_j\}$, $R\{y_k\}$.

2. Compute $R\{\delta_k\}$ & $R\{\delta_j\}$.

3. Compute W elements of $\vec{v}^T H$ using $(*)$

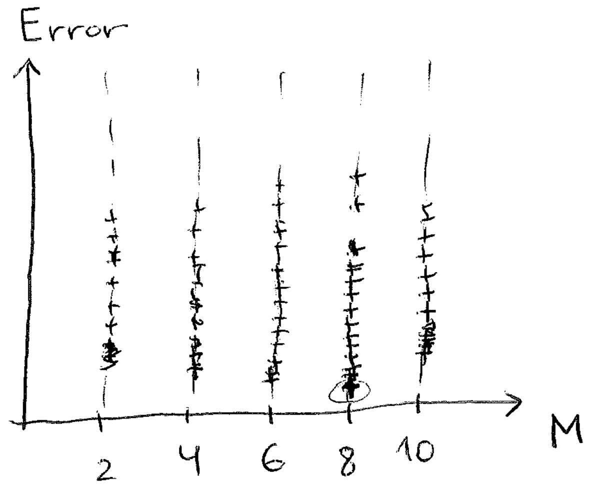
This requires $O(W)$ operations.

Note that this technique can be used to evaluate H by choosing W unit vectors as $\vec{v}$: $\vec{v}^T = \overline{0\ 0 \ldots 1 \ldots 0}$ (jth position) $\Rightarrow O(W^2)$ operations, equivalent to the direct calculation described above

Regularization in NN

M is a parameter to choose.
↑ # hidden units

Train NN on a training set, test on a test set:



← multiple local min'n runs from random initializations

↑ choose globally best realization ($M=8$ here)

More explicitly, one can use

$$\widetilde{E}(\vec{w}) = E(\vec{w}) + \frac{\lambda}{2}\vec{w}^T\vec{w}. \qquad (*)$$

However, $(*)$ is NOT inv wrt certain transformation properties obeyed by NNs.

Indeed, consider a two-layer network:

$$\begin{cases} y_k = \sum_j w_{kj} z_j + w_{k0}, & \leftarrow \text{regression} \\ z_j = h\left(\sum_i w_{ji} x_i + w_{j0}\right). \end{cases}$$

Consider $x_i \to \tilde{x}_i = a x_i + b$, then

$$\begin{cases} w_{ji} \to \tilde{w}_{ji} = \frac{1}{a} w_{ji}, \\ w_{j0} \to \tilde{w}_{j0} = w_{j0} - \frac{b}{a} \sum_i w_{ji} \end{cases}$$ give

$$\tilde{z}_j = h\Big(\sum_i \frac{w_{ji}}{a}(a x_i + b) + w_{j0} - \frac{b}{a} \sum_i w_{ji} \Big) =$$

$$= h\Big(\sum_i w_{ji} x_i + w_{j0} \Big) = z_j \Rightarrow$$

$$\Rightarrow \tilde{y}_k = \underline{\underline{y_k}}.$$

Similarly, $y_k \to \tilde{y}_k = c y_k + d$ can be "undone" by weight / bias rescaling:

$$\begin{cases} w_{kj} \to \tilde{w}_{kj} = c\, w_{kj}, \\ w_{k0} \to \tilde{w}_{k0} = c\, w_{k0} + d. \end{cases} \Rightarrow \tilde{x}_i = x_i .$$

Clearly, the λ-term in (*) breaks this invariance. So, use

$$\frac{\lambda_1}{2} \underbrace{\sum_{w \in W_1} w^2}_{\text{1st layer}} + \frac{\lambda_2}{2} \underbrace{\sum_{w \in W_2} w^2}_{\text{2nd layer}}$$ (biases left unconstrained)

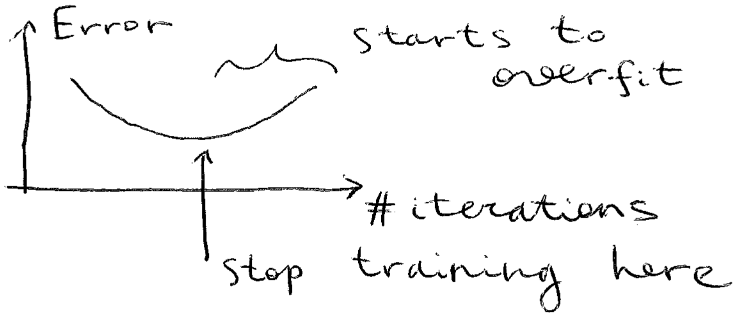
This regularizer will remain inv under $\tilde{w}_{ji} = a^{-1} w_{ji}$, $\tilde{w}_{kj} = c\, w_{kj}$ if

$$\lambda_1 \to a^{1/2} \lambda_1, \quad \lambda_2 \to c^{-1/2} \lambda_2 .$$

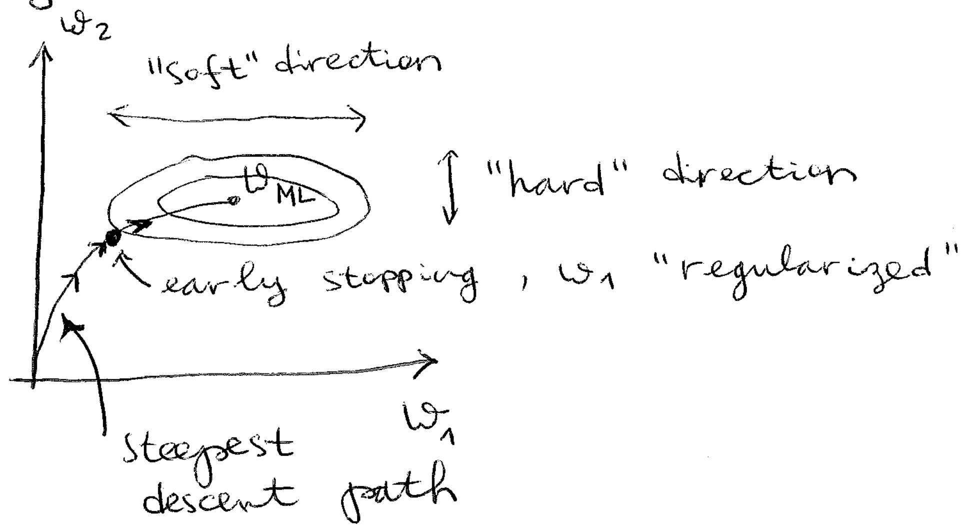
Can divide weights into more groups, not just W_1 & W_2.

Early stopping

Train on a training set, test on a test set:



Early stopping effectively controls model complexity:



Invariances

Transformations of inputs (e.g. translating, rescaling & rotating digits in 2D images) should not affect predictions.

However, raw data (pixel intensities) change in complex ways.

How to deal with this?

1. Add transformed patterns to the dataset, let the NN "sort it out" (i.e. learn the invariances) [data intensive]

2. Add a term to the error function to penalize changes in NN output when input is transformed (tangent propagation)

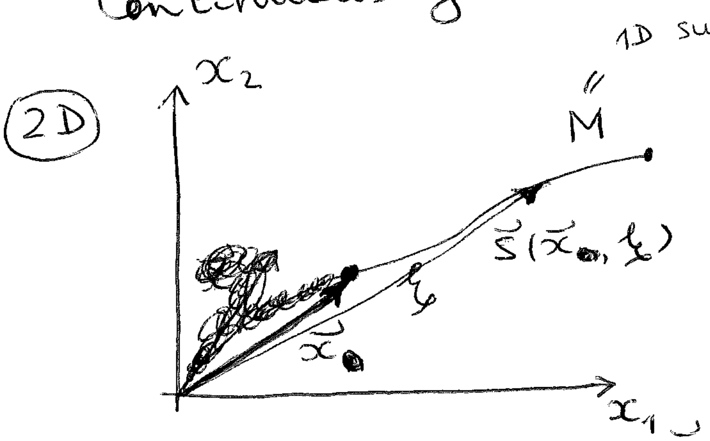
3. Extract transform-inv features from raw data first [domain knowledge]

4. Build the invariance into structure and weight distributions of the NN

Tangent propagation

Use regularization to encourage models to be transform-inv.

Continuously transform $\vec{x}_0$:



one-parameter continuous transform: ξ

Note that $\vec{s}(\vec{x}_0, 0) = \vec{x}_0$

Then $\vec{\tau} = \dfrac{\partial \vec{s}}{\partial \xi} \Rightarrow \vec{\tau}_0 = \left.\dfrac{\partial \vec{s}}{\partial \xi}\right|_{\xi=0}$

tangent vector

Finally,

$$\left.\frac{\partial y_k}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D \,\swarrow \text{ \# input nodes}} \left.\frac{\partial y_k}{\partial x_i}\frac{\partial x_i}{\partial \xi}\right|_{\xi=0} = \sum_i \underbrace{J_{ki}}_{\text{Jacobian}} \tau_{o,i} .$$

We can introduce $\tilde{E} = E + \lambda \Omega$, where

$$\Omega = \frac{1}{2}\sum_{n,k}\left(\left.\frac{\partial y_{nk}}{\partial \xi}\right|_{\xi=0}\right)^2 =$$

$$= \frac{1}{2}\sum_{n,k}\left(\sum_{i=1}^{D} J_{n,ki}\,\tau_{on,i}\right)^2 .$$

This will encourage NN weights to exhibit invariance wrt the transforms. In practice $\vec{\tau}$ can be approximated by finite differences for small ξ [e.g. rotated digits, see. Fig. 5.16 in the book]

Also, need to extend the backpropagation technique to $\tilde{E}$ (that is, backpropagate derivatives of Ω wrt weights).

additional regularization techniques

1. Dropout (~2014)

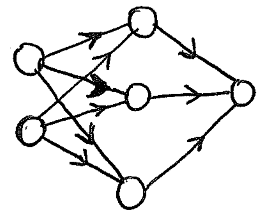
Idea: prevent overfitting / reduce spurious correlations between neurons by randomly dropping out (removing) neurons and their connections from the neural network (NN).

Typically, for each mini-batch (i.e., each gradient descent step), each neuron is dropped from NN with probab. $p \ll 1$. The grad. descent is then performed on the "thinned" network.

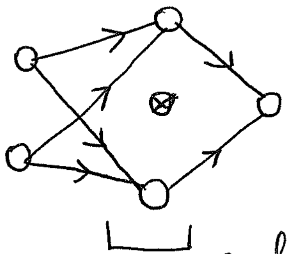
Since on average each weight is present only a fraction $q = 1-p$ of the time, the corresponding full-network weights (to be used e.g. on a hold-out test set) are given by

$$\underbrace{\vec{w}_{test}}_{\substack{\text{average weight over}\\ \text{the ensemble of "thinned" NN}}} = q\,\vec{w}_{train} + p \cdot \vec{0} = \underline{\underline{q\,\vec{w}_{train}}}.$$

Full NN



"Thinned" NN
at some step
during optimization



only hidden layers can be thinned out

(2.) Batch normalization (~2015)

<u>Idea</u>: prevent neuron inputs a_j^ℓ ← layer index; j ← neuron index, layer ℓ
from "saturating" the gradients
by being too biased.

The solution is to standardize each input :

$$a_j^\ell \to \hat{a}_j^\ell = \frac{a_j^\ell - E[a_j^\ell]}{\sqrt{Var[a_j^\ell]}}, \qquad (1)$$

where $E[\cdot]$ & $Var[\cdot]$ are computed over all datapoints in the current mini-batch.

However, this was found to be too restrictive, so another step was imposed:

$$\hat{a}^\ell_j \to \hat{\hat{a}}^\ell_j = \underbrace{\gamma^\ell_j \hat{a}^\ell_j + \beta^\ell_j}_{\text{linear rescaling}} \qquad (2)$$

Eqs. (1) & (2) can be viewed as adding 1 extra layer to the NN, with γ's & β's treated as fitting prms which can be treated using backpropagation.

In practice, batch normalization improves convergence but also acts as a regularizer, for reasons that are not fully understood at the moment.