Consider $\tilde{W}^T H = \tilde{V}_j \nabla_j (\nabla_i E)$

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

Define $R\{.\} = \tilde{W}^T \tilde{V}$, note that

$R\{\tilde{W}_k\} = \tilde{V}_j \nabla_j \tilde{W}_k = \tilde{V}_k$, or

$R\{\tilde{W}\} = \tilde{V}$.

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, $R\{a_j\} = \sum_i R\{w_{ji}\} x_i = \sum_i v_{ji} x_i$, $R\{z_j\} = h'(a_j) R\{a_j\}$,

$R\{y_k\} = \sum_j w_{kj} z_j + \sum_j w_{kj} R\{z_j\} = h'(a_j) R\{a_j\}$

Further, $\delta_k = y_k - t_k$,

$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$ as before

backpropagation
Then
\[ \sum_k R\{5_k^3\} = R\{Y_k^3\}, \]
\[ R\{5_j^3\} = h''(a_j) R\{d_j^3\} \sum_k \omega_{kj} 5_k + \]
hidden
\[ + h'(a_j) \sum_k \omega_{kj} 5_k + h'(a_j) \sum_k \omega_{kj} R\{5_k^3\}. \]

Finally,
\[ \frac{\partial E}{\partial w_{kj}} = 5_k 2_j, \]
\[ \frac{\partial E}{\partial w_{ji}} = 5_j x_i. \]

\[ \left\{ \begin{array}{c}
R\left\{ \frac{\partial E}{\partial w_{kj}} \right\} = R\{5_k^3\} 2_j + 5_k R\{2_j^3\}, \\
R\left\{ \frac{\partial E}{\partial w_{ji}} \right\} = x_i R\{5_j^3\}.
\end{array} \right. \quad (\ast) \]

Algorithm: 1. Do a forward pass,
compute \( R\{d_j^3\}, R\{2_j^3\}, R\{Y_k^3\}. \)
2. Compute \( R\{5_k^3\} \) & \( R\{5_j^3\}. \)
3. Compute \( W \) elements of
\( \tilde{V}^T H \) using \( (\ast) \)

This requires \( \Theta(W) \) operations.

Note that this technique can be used
to evaluate \( H \) by choosing \( W \) unit vectors
as \( \tilde{V} : \tilde{V}^T = \underbrace{0 \cdots 1}_{jth \ position} \cdots 0 \) \( \Rightarrow \Theta(W^2) \) operations,
equivalent to the direct calculation described above.
Regularity in NN

M is a parameter to choose.

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

\[ \text{Error} \]

\[ M \]

\[ \text{multiple local min's runs from random initializations} \]

\[ \text{choose globally best realization} \quad (M=8 \text{ here}) \]

More explicitly, one can use

\[ \tilde{E}(\tilde{\omega}) = E(\tilde{\omega}) + \frac{\lambda}{2} \tilde{\omega}^T \tilde{\omega} \]  \hspace{1cm} (*)

However, (*) is NOT invariant certain transformation properties obeyed by NNs.

Indeed, consider a two-layer network:

\[
\begin{align*}
Y_k &= \sum_j w_{kj} Z_j + w_{ko}, & \leftarrow \text{regression} \\
Z_j &= h(\sum_i w_{ji} x_i + w_{jo})
\end{align*}
\]

Consider \( x_i \rightarrow \tilde{x}_i = ax_i + b \), then
\[
\begin{align*}
\tilde{w}_{ji} &= \frac{1}{d} w_{ji}, \\
\tilde{w}_{jo} &= w_{jo} - \frac{b}{d} \sum_i w_{ji}
\end{align*}
\]
give
\[
\tilde{z}_j = h \left( \sum_i \frac{w_{ji}}{d} (ax_i + b) + \tilde{w}_{jo} - \frac{b}{d} \sum_i \tilde{w}_{ji} \right) = \\
= h \left( \sum_i (w_{ji} x_i + \tilde{w}_{jo}) \right) = z_j \implies \\
\implies \tilde{y}_k = y_k.
\]

Similarly, \( y_k \rightarrow \tilde{y}_k = cy_k + d \) can be "undone" by weight/bias rescaling:
\[
\begin{align*}
\tilde{w}_{kj} &= c w_{kj}, \implies \tilde{x}_i = x_i. \\
\tilde{w}_{ko} &= c w_{ko} + d.
\end{align*}
\]
Clearly, the \( \lambda \)-term in (*) breaks this invariance. So, use
\[
\frac{\lambda_1}{2} \sum_{w \in W_1} w^2 + \frac{\lambda_2}{2} \sum_{w \in W_2} w^2
\]
(\( \lambda \)-\( \lambda \)-terms left unconstrained)
\[
\begin{align*}
\text{1st layer} & \quad \text{2nd layer}
\end{align*}
\]
This regularizer will remain invariant under \( \tilde{w}_{ji} = d^{-1} w_{ji} \), \( \tilde{w}_{kj} = c w_{kj} \) if
\[
\lambda_1 \rightarrow d^{-1/2} \lambda_1, \quad \lambda_2 \rightarrow c^{-1/2} \lambda_2.
\]
Can divide weights into more groups, not just \( W_1 \& W_2 \).
Early stopping

Train on a training set, test on a test set. Error starts to overfit

# iterations
stop training here

Early stopping effectively controls model complexity:

$$w^2$$

"Soft" direction

$$w^1$$

"Hard" direction

early stopping, $$w^1$$ "regularized"

Steepest descent path

Invariances

Transformations of inputs (e.g., resizing, 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 $\tilde{x}$:

$$
\tilde{x}(\xi, 0) = \tilde{x}^0
$$

Then

$$
\frac{\partial \tilde{x}}{\partial \xi} = \frac{\partial \tilde{S}}{\partial \xi} |_{\xi = 0}
$$

one-parameter continuous transform $\xi$
Finally, \[ \frac{\partial Y_{jk}}{\partial \mathbf{z}} \bigg|_{\mathbf{z}=0} = \sum_{i=1}^{D} \frac{\partial Y_{jk}}{\partial x_i} \frac{\partial x_i}{\partial \mathbf{z}} \bigg|_{\mathbf{z}=0} = \sum_{i=1}^{D} J_{ki} T_{0,i}. \]

We can introduce \( \tilde{E} = E + \lambda \mathcal{U} \), where

\[ \mathcal{U} = \frac{1}{2} \sum_{n,k} \left( \frac{\partial Y_{nk}}{\partial \mathbf{z}} \bigg|_{\mathbf{z}=0} \right)^2 = \frac{1}{2} \sum_{n,k} \left( \sum_{i=1}^{D} J_{nk,i} T_{0,i} \right)^2. \]

This will encourage NN weights to exhibit invariance w.r.t the transforms. In practice it can be approximated by finite differences for small \( \lambda \) [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 \( \mathcal{U} \) w.r.t weights).
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. \( p < 1 \). 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:

\[
\hat{w}_{\text{test}} = q \hat{w}_{\text{train}} + p \cdot \bar{w} = q \hat{w}_{\text{train}} + \bar{w}.
\]

average weight over the ensemble of "thinned" NN
"Thinned" NN at some step during optimization.

only hidden layers can be thinned out.


Idea: prevent neuron inputs \( \tilde{a}_{l,j} \) from "saturating" the gradients by being too biased.

The solution is to standardize each input:

\[
\tilde{a}_{l,j} = \frac{a_{l,j} - \text{E}[a_{l,j}]}{\sqrt{\text{Var}[a_{l,j}]}} \tag{1}
\]

where \( \text{E}[\cdot] \) & \( \text{Var}[\cdot] \) are computed over all datapoints in the current mini-batch.

However, this was found to be too restrictive, so another step was imposed:
\[ \hat{a}_j \rightarrow \hat{a}_j = \gamma_j \hat{a}_j + \beta_j \]  

linear rescaling

Eqs. (1) & (2) can be viewed as adding 1 extra layer to the NN, with \( \gamma \)'s & \( \beta \)'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.