4F10: Deep Learning

Mark Gales

Michaelmas 2016
What is Deep Learning?

From Wikipedia:

*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.*
The Rise of Deep Learning (June 2016)

- Plot shows citations to Geoff Hinton papers
  - highly influential researcher in deep learning
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
- **Example**
  - encoder-decoder models for sequence-to-sequence models
  - and attention for sequence-to-sequence modelling
Basic Building Blocks
Deep Neural Networks [12]

- General mapping process from input $x$ to output $y(x)$

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

- deep refers to number of hidden layers
- Output from the previous layer connected to following layer:
  - $x^{(k)}$ is the input to layer $k$
  - $x^{(k+1)} = y^{(k)}$ the output from layer $k$
- General form for layer $k$:

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

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

- $L$ 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, \(0 \leq y_i(x) \leq 1\).
  \[
  \phi(z_i) = \frac{1}{1 + \exp(-z_i)}
  \]

- **Softmax** function: output \(0 \leq y_i(x) \leq 1\), \(\sum_{i=1}^{n} y_i(x) = 1\).
  \[
  \phi(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}
  \]

- **Hyperbolic tan** function: output continuous, \(-1 \leq y_i(x) \leq 1\).
  \[
  \phi(z_i) = \frac{\exp(z_i) - \exp(-z_i)}{\exp(z_i) + \exp(-z_i)}
  \]
Activation Functions

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

- **softmax**, usual output layer for classification tasks
- **sigmoid/tanh**, often used for hidden layers
• Alternative activation function: **Rectified Linear Units**

\[ \phi(z_i) = \max(0, z_i) \]

• Related activation function **noisy ReLU**:

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

  - efficient, no exponential/division, rapid convergence in training

• **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 [11]

- Modify layer to model the residual

\[ y(x) = F(x) + x \]

- Allows deeper networks to be built
- Deep residual learning
- Links to highway connections
Pooling/Max-Out Functions [15, 26]

- 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** $\phi(y_1, y_2, y_3) = \log\left(\sum_{i=1}^{3} \exp(y_i)\right)$
  - **p-norm** $\phi(y_1, y_2, y_3) = \left(\sum_{i=1}^{3} |y_i|\right)^{1/p}$

- Has also been applied for unsupervised adaptation
Convolutional Neural Networks

- n frames
- k frequencies
- pooling layer
- filter 1
- filter n
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  \hspace{1cm} **filter**: 1, 1, 1

  **default**: 3, 6, 9, 12, 15, 11  \hspace{1cm} **no padding**: 6, 9, 12, 15

  **dilation=1**: 4, 6, 9, 12, 8, 10  \hspace{1cm} **stride=2**: 3, 9, 15
Convolutional Neural Network (cont)

- For a 2-dimensional 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 \( k, l \)
- \( \phi \) is the non-linear activation function

- For a \( 5 \times 5 \) 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 $2 \times 2$ filter on $4 \times 4$ “image”
  - *stride* of 2 - yields output of $2 \times 2$
- Possible to also operate with a stride of 1 *overlapping pooling*
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
Recurrent Neural Networks [21, 19]

- Consider a **causal** sequence of observations \( x_{1:t} = \{x_1, \ldots, x_t\} \)

- Introduce recurrent units

  \[
  h_t = f^h \left( W^f_h x_t + W^r_h h_{t-1} + b_h \right) \]

  \[
  y(x_{1:t}) = f^f \left( W^y h_t + W^x x_t + b_y \right) \]

  - \( h_t \) **history vector** at time \( t \)
  - Two history weight matrices
    - \( W^f_h \) forward, \( W^r_h \) recursion

- Uses approximation to model **history of observations**

  \[
  F(x_{1:t}) = F(x_t, x_{1:t-1}) \approx F(x_t, h_{t-1}) \approx F(h_t) = y(x_{1:t})
  \]

  - network has (causal) memory encoded in **history vector** \((h_t)\)
RNN Variant: Bi-Directional RNN  [23]

- Bi-directional: use complete observation sequence - non-causal

\[
\mathcal{F}_t(x_{1:T}) = \mathcal{F}(x_{1:t}, x_{t:T}) \approx \mathcal{F}(h_t, \tilde{h}_t) = y_t(x_{1:T})
\]
Latent Variable (Variational) RNN (reference) [7]

- Variational: introduce latent variable sequence $z_{1:T}$

\[
p(y_t|x_{1:t}) \approx \int p(y_t|x_t, z_t, h_{t-1}) p(z_t|h_{t-1}) \, dz_t
\]

\[
\approx \int p(y_t|h_t) p(z_t|h_{t-1}) \, dz_t
\]

- $z_t$ a function of complete history (complicates training)
Network Gating

- A flexible extension to activation function is **gating**
  - standard form is \( \sigma(\cdot) \) sigmoid activation function
    \[
    i = \sigma(W^f x_t + W^r h_{t-1} + b)
    \]
  - vector acts a probabilistic gate on network values

- Gating can be applied at various levels
  - **features**: impact of input/output features on nodes
  - **time**: memory of the network
  - **layer**: influence of a layer’s activation function
Gated Recurrent Unit [6]

Recurrent unit

Gated Recurrent Unit

\[ h_t = f(h_{t-1}, x_t) \]

\[ i_t = \sigma(f(h_{t-1})) \]

\[ o_t = \sigma(f(h_{t-1})) \]

\[ h_t = h_{t-1} + o_t \cdot i_t \]
Gated Recurrent Unit [6]

- Gated Recurrent Unit (GRU) introduces
  - forget gate \( i_f \): gating over time
  - output gate \( i_o \): gating over features (and time)
- Relationships (standard configuration - there are variants)

\[
\begin{align*}
  i_f &= \sigma(W^f_f x_t + W^r_f h_{t-1} + b_f) \\
  i_o &= \sigma(W^f_o x_t + W^r_o h_{t-1} + b_o) \\
  y_t &= f(W^f_y x_t + W^r_y (i_f \circ h_{t-1}) + b_y) \\
  h_t &= i_o \circ h_{t-1} + (1 - i_o) \circ y_t
\end{align*}
\]

- \( \circ \) represents element-wise multiplication between vectors
Long-Short Term Memory Networks (reference) [13, 10]
Long-Short Term Memory Networks (reference)

- The operations can be written as (peephole config):
  - Forget gate ($i_f$), Input gate ($i_i$), Output gate ($i_o$)

  
  \[
  i_f = \sigma(W_f^fx_t + W_f^xh_{t-1} + W_f^mc_{t-1} + b_f) \\
  i_i = \sigma(W_i^fx_t + W_i^xh_{t-1} + W_i^mc_{t-1} + b_i) \\
  i_o = \sigma(W_o^fx_t + W_o^xh_{t-1} + W_o^mc_t + b_o)
  \]

  - Memory Cell, history vector and gates are related by

  \[
  c_t = i_f \odot c_{t-1} + i_i \odot f^m(W_c^fx_t + W_c^xh_{t-1} + b_c) \\
  h_t = i_o \odot f^h(c_t)
  \]

  - more complicated than GRU (three gates, memory cell)
  - memory cell weight matrices ($W^m_f$, $W^m_i$, $W^m_o$) diagonal
  - can allow explicit analysis of individual cell elements
Highway Connections [25]

- Gate the output of the node (example from recurrent unit)
  - combine with output from previous layer ($x_t$)

$$i_h = \sigma(W^f x_t + W^r h_{t-1} + b)$$

$$h_t = i_h \odot f(W^f x_t + W^r h_{t-1} + b) + (1 - i_h) \odot x_t$$
**Example Deep Architecture: ASR (reference)** [22]

- **Example Architecture from Google (2015)**
  - **C**: CNN layer (with pooling)
  - **L**: LSTM layer
  - **D**: fully connected layer
- **Two multiple layer “skips”**
  - (1) connects input to LSTM input
  - (2) connects CNN output to DNN input
- **Additional linear projection layer**
  - reduces dimensionality
  - and number of network parameters!

---

The use of short and long-term features in a neural network has previously been explored (i.e., [13, 14]). The main difference between our work and previous work is that we pass the output of the CNN directly into the LSTM, without extra layers and thus minimal parameter increase.

As shown in [5], these higher layers are appropriate for producing a higher-order feature representation that is more easily separable into different classes we want to discriminate. Each fully connected layer has 1,024 hidden units.

After frequency modeling is performed, we next pass the CNN output to a few fully connected DNN layers. This idea of combining information from CNN and both the LSTM and DNN is indicated by the dashed stream in Figure 1. This diagram shows that the long-term feature from the CNN is connected to the LSTM. This is shown by dashed stream (2) in Figure 1. This connection allows for the combination of short and long-term features, which is motivated by work in other fields.

In our work, each frame is denoted as $x_t$, surrounded by frequency context. Thus, we add a linear layer to reduce feature dimension, before passing this to the LSTM. In fact, the original LSTM work differs in that we pass the output of the CNN directly into the DNN, without extra layers and thus minimal parameter increase.

Finally, after performing frequency and temporal modeling, we pass the output of the LSTM to a few fully connected DNN layers. In our work, each frame input is denoted as $x_t$, and the output targets are denoted as $x_{t-l}, ..., x_t, ..., x_{t+r}$.

The dimension of the last layer of the CNN is large, due to the number of feature-maps $20 \times 20 \times 256$ followed by a 4x3 filter for the second convolutional layer, and these feature-maps are surrounded by temporal context. Thus, we add a linear layer after the CNN layers, which allows for a reduction in parameters with no loss in accuracy. In our experiments, we found that this linear layer after the CNN layers allows for a reduction in parameters.

There are two multiple layer “skips” in the CLDNN architecture shown in Figure 1. This diagram shows that the input is denoted as $x_t$, surrounded by temporal context. Thus, we add a linear layer for dimensionality reduction. Unless otherwise indicated, all linear layers have 1,024 hidden units. The dimension of the last layer of the CNN is large, due to the number of feature-maps $20 \times 20 \times 256$, followed by a 4x3 filter for the second convolutional layer, and these feature-maps are surrounded by temporal context. Thus, we add a linear layer after the CNN layers, which allows for a reduction in parameters with no loss in accuracy. In our experiments, we found that this linear layer after the CNN layers allows for a reduction in parameters.

The LSTM is then unrolled for 20 time steps for training with truncated backpropagation through time (BPTT). In addition, the output state of the LSTM is denoted as $x_t$, surrounded by temporal context. Thus, we add a linear layer after the CNN layers, which allows for a reduction in parameters with no loss in accuracy. In our experiments, we found that this linear layer after the CNN layers allows for a reduction in parameters.
Network Training and Error Back Propagation
Training Data: Classification

- Supervised training data comprises
  - $x_i$: $d$-dimensional training observation
  - $y_i$: class label, $K$ possible (discrete) classes
- Encode class labels as 1-of-$K$ (“one-hot”) coding: $y_i \rightarrow t_i$
  - $t_i$ is the $K$-dimensional target vector for $x_i$
  - zero other than element associated with class-label $y_i$
- Consider a network with parameters $\theta$ and training examples:

\[
\{ \{x_1, t_1\}, \ldots, \{x_n, t_n\}\}
\]

- need “distance” from target $t_i$ to network output $y(x_i)$
Training Criteria

- **Least squares error**: one of the most common training criteria.

\[
E(\theta) = \frac{1}{2} \sum_{p=1}^{n} \|y(x_p) - t_p\|^2 = \frac{1}{2} \sum_{p=1}^{n} \sum_{i=1}^{K} (y_i(x_p) - t_{pi})^2
\]

- **Cross-Entropy**: note non-zero minimum (entropy of targets)

\[
E(\theta) = - \sum_{p=1}^{n} \sum_{i=1}^{K} t_{pi} \log(y_i(x_p))
\]

- **Cross-Entropy for two classes**: single binary target

\[
E(\theta) = - \sum_{p=1}^{n} \left( t_p \log(y(x_p)) + (1 - t_p) \log(1 - y(x_p)) \right)
\]
Supervised training data comprises
- $x_i$: $d$-dimensional training observation
- $y_i$: $K$-dimensional (continuous) output vector

Consider a network with parameters $\theta$ and training examples:

$$\{\{x_1, y_1\}, \ldots, \{x_n, y_n\}\}$$

need “distance” from target $y_i$ to network output $y(x_i)$

Least squares commonly used criterion

$$E(\theta) = \frac{1}{2} \sum_{p=1}^{n} (y(x_p) - y_p)'(y(x_p) - y_p)$$

$y(x_i)$ may be viewed as the mean of the prediction
Generalise least squares (LS) to maximum likelihood (ML)

\[ E(\theta) = \sum_{p=1}^{n} \log(p(y_p|x_p, \theta)) \]

- 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
    \[ \exp(y_{i}(x)) > 0 \]
- for means just use a linear activation function
• Predict a \textbf{mixture} of \( M \) Gaussians
  \begin{itemize}
    \item \( \mathcal{F}_m^{(c)}(x_t) \): prior prediction
    \item \( \mathcal{F}_m^{(\mu)}(x_t) \): mean prediction
    \item \( \mathcal{F}_m^{(\sigma)}(x_t) \): variance prediction
  \end{itemize}
• For component \( m \), output
  \[ y_m(x_t) = \begin{bmatrix}
    \mathcal{F}_m^{(c)}(x_t) \\
    \mathcal{F}_m^{(\mu)}(x_t) \\
    \mathcal{F}_m^{(\sigma)}(x_t)
  \end{bmatrix} \]
• Optimise using \textbf{maximum likelihood} where
  \[ p(y_t|x_t) = \sum_{m=1}^{M} \mathcal{F}_m^{(c)}(x_t) \mathcal{N}(y_t; \mathcal{F}_m^{(\mu)}(x_t), \mathcal{F}_m^{(\sigma)}(x_t)) \]
Gradient Descent [20]

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

\[ \theta[\tau + 1] = \theta[\tau] - \Delta \theta[\tau] = \theta[\tau] - \eta \frac{\partial E}{\partial \theta} \bigg|_{\theta[\tau]} \]

- how to get the gradient for all model parameters
- how to avoid local minima
- need to consider how to set \( \eta \)
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
  - merging bias vector for each layer into weight matrix

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

Initially just consider a single layer perceptron
Single Layer Perceptron Training

- Take the example of **least squares error** cost function

\[ E(\theta) = \frac{1}{2} \sum_{p=1}^{n} ||y(x_p) - t_p||^2 = \sum_{p=1}^{n} E^{(p)}(\theta) \]

- Use chain rule to compute derivatives through network

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

  - 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

\[ \frac{\partial E^{(p)}(\theta)}{\partial w_i} = (y(x_p) - t_p) y(x_p)(1 - y(x_p)) x_{pi} \]
Error Back Propagation

- Simple concept can be extended to multiple (hidden) layers
  - output from layer \( k - 1 \) \( (y^{(k-1)}) \) is the input to layer \( k \) \( (x^{(k)}) \)
- \( L + 1 \) layer network - use backward recursion (see notes)
  - model parameters \( \theta = \{ W^{(1)}, \ldots, W^{(L+1)} \} \)
  - final output \( y(x) = y^{(L+1)} \)

\[
\delta^{(k)} = \Lambda^{(k)} W^{(k+1)} \delta^{(k+1)}, \quad \frac{\partial E(\theta)}{\partial W^{(k)}} = \delta^{(k)} x^{(k)'}
\]

- \( \Lambda^{(k)} = \frac{\partial y^{(k)}}{\partial z^{(k)}} \): layer \( k \) activation derivative matrix
- \( W^{(k+1)} \): weight matrix for layer \( k + 1 \)
- \( \delta^{(k+1)} = \frac{\partial E(\theta)}{\partial z^{(k)}} \): error vector for layer \( k + 1 \)
The process to get the derivative involves:

1. For input vector $x_p$ propagate forward: yields $(1 \leq k \leq L)$
   - $y^{(k)}$ the output value for each node of layer all layers
   - $z^{(k)}$ the input value to the non-linearity for layer $k$

2. Compute $\frac{\partial E(\theta)}{\partial y(x)}|_{\theta[\tau]}$ (the gradient at the output layer).

3. From the output gradient propagate backwards: yields
   - $\delta^{(k)}$ the error vector for each layer
   - $\frac{\partial E(\theta)}{\partial W^{(k)}}$: the (desired) derivative for layer $k$ weights
Optimisation
Batch/On-Line Gradient Descent

- The 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, \( \tilde{D} \),

\[
E(\theta) = - \sum_{p \in \tilde{D}} \sum_{i=1}^{K} t_{pi} \log(y_i(x_p))
\]

- How to select the subset, \( \tilde{D} \)?
  - small subset “poor” estimate of true gradient
  - large subset each parameter update is expensive
- One extreme is to update after each sample
  - \( \tilde{D} \) 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
     - \( \tilde{D} \) is a (random) subset of the training data
     - better estimate of the gradient at each update
     - but reduces number of iterations

- Mini-batch updates are (almost) always 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:

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

- smooths parameter changes over iterations
- introduces an additional tunable parameter
- For simplicity introduce compact notation

\[
\left. \frac{\partial E(\theta)}{\partial \theta} \right|_{\theta[\tau]} = \nabla(E(\theta[\tau]))
\]
Adaptive Learning Rates

- Speed of convergence depends on $\eta$
  - 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
    $$\theta[\tau + 1] = \theta[\tau] - \Delta\theta[\tau] = \theta[\tau] - \eta[\tau] \nabla(E(\theta[\tau]))$$
  - how to set $\eta[\tau]$ (or generally parameter update $\Delta\theta[\tau]$)?
- One very simple approach
  $$\eta[\tau + 1] = \begin{cases} 
   1.1\eta[\tau]; & \text{if } E(\theta[\tau]) < E(\theta[\tau - 1]) \\
   0.5\eta[\tau]; & \text{if } E(\theta[\tau]) > E(\theta[\tau - 1]) 
  \end{cases}$$
  - increase learning rate when going in “correct direction”
Gradient Descent Refinements (reference)

- **Nesterov**: concept of gradient at next iteration

\[
\Delta \theta[\tau] = \eta \nabla (E(\theta[\tau] - \alpha \Delta \theta[\tau - 1])) + \alpha \Delta \theta[\tau - 1]
\]

- **AdaGrad**: dimension specific learning rates (\(\epsilon\) floor parameter)

\[
\Delta \theta[\tau] = \eta \beta_t \odot \frac{\partial E(\theta)}{\partial \theta} \bigg|_{\theta[\tau]} ; \quad \beta_{ti} = \frac{1}{\sqrt{\epsilon + \sum_{t=1}^{\tau} \nabla_i(E(\theta[t]))^2}}
\]

- \(\epsilon\) is a smoothing term to avoid division by zero

- **Adam**: Adaptive Moment Estimation: use dimension moments

\[
\Delta \theta_i[\tau] = \frac{\eta}{\sqrt{\sigma_{\tau i}^2 + \epsilon}} \mu_{\tau i} ; \quad \mu_{\tau i} = \alpha_1 \mu_{(\tau - 1)i} + (1 - \alpha_1) \nabla_i (E(\theta[\tau])) \\
\sigma_{\tau i}^2 = \alpha_2 \sigma_{(\tau - 1)i}^2 + (1 - \alpha_2) \nabla_i (E(\theta[\tau]))^2
\]

- additional normalisation applied to \(\mu_{\tau i}\) and \(\sigma_{\tau i}^2\) to offset initialisation bias
Second-Order Approximations

- Gradient descent makes use of first-order derivatives of
  - what about higher order derivatives? Consider

\[
E(\theta) = E(\theta[\tau]) + (\theta - \theta[\tau])'g + \frac{1}{2}(\theta - \theta[\tau])'H(\theta - \theta[\tau]) + O(\theta^3)
\]

where

\[
g = \nabla E(\theta[\tau]); \quad (H)_{ij} = h_{ij} = \left. \frac{\partial^2 E(\theta)}{\partial \theta_i \partial \theta_j} \right|_{\theta[\tau]}
\]

- Ignoring higher order terms and equating to zero

\[
\nabla E(\theta) = g + H(\theta - \theta[\tau])
\]

Equating to zero (check minimum!) - \(H^{-1}g\) Newton direction

\[
\theta[\tau + 1] = \theta[\tau] - H^{-1}g; \quad \Delta \theta[\tau] = H^{-1}g
\]
Issues with Second-Order Approaches

1. The evaluation of the Hessian may be computationally expensive as $O(N^2)$ parameters must be accumulated for each of the $n$ training samples.

2. The Hessian must be inverted to find the direction, $O(N^3)$. 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.

   $$v'Hv > 0$$

   for all $v$. The Hessian may be made positive definite using

   $$\tilde{H} = H + \lambda I$$

   If $\lambda$ is large enough then $\tilde{H}$ 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

\[ E(\theta) \approx E(\theta[\tau]) + b(\theta - \theta[\tau]) + a(\theta - \theta[\tau])^2 \]
\[ \frac{\partial E(\theta)}{\partial \theta} \approx b + 2a(\theta - \theta[\tau]) \]

- 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 \( \tau \) is \( g[\tau] \)
  - after new update \( \Delta \theta[\tau] \) the gradient should be zero
- The following equalities are obtained

\[ g[\tau - 1] = b - 2a\Delta \theta[\tau - 1], \quad 0 = b + 2a\Delta \theta[\tau], \quad g[\tau] = b \]

\[ \rightarrow \Delta \theta[\tau] = \frac{g[\tau]}{g[\tau - 1] - g[\tau]} \Delta \theta[\tau - 1] \]
• 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
• Parameter at minimum of quadratic approximation:
  \[ \theta[\tau + 1] = 1 \]
A major issue 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 ($N$ is the total number of weights):

\[
\tilde{E}(\theta) = E(\theta) + \nu \Omega(\theta); \quad \Omega(\theta) = \frac{1}{2} \sum_{i=1}^{N} w_i^2
\]

- a zero "prior" is used for the model parameters
- Simple to include in gradient-descent optimisation

\[
\nabla \tilde{E}(\theta[\tau]) = \nabla E(\theta[\tau]) + \nu w[\tau]
\]
Dropout is a simple way of improving generalisation to new data. It involves:

1. Randomly de-activate (say) 50% of the nodes in the network
2. Update the model parameters

This prevents a single node from specialising to a task.
Network Initialisation: Data Pre-Processing

- As with standard classifiers, two-stage classification is 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).
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

- Pass 1-dimensional data through a **sigmoid**
- 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 $y = W^T x$, assuming all weights/observations independent
  - $x$ $n$ dimensional, zero mean, identity variance

$$\text{Var}(y_i) = \text{Var}(w_i^T x) = n\text{Var}(w_{ij})\text{Var}(x_i)$$

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

$$\text{Var}(w_{ij}) = \frac{1}{n} \quad (\text{for: } \text{Var}(y_i) = \text{Var}(x_i))$$
Alternative Initialisation schemes

- **Restricted Boltzmann machine: generative pre-training**
  - initialise the network parameters using a generative RBM
  - train the RBM layer-by-layer
  - if only the output layer trained this is a deep belief network

- **Layer-by-layer training: discriminative pre-training**
  - iterative process:
    1. remove o/p layer, add a random initialised layer, add o/p layer
    2. train (limited iterations e.g. 1) network and then goto (1)
Example Systems
(reference)
Autoencoders (Non-Linear Feature Extraction)

- An autoencoder is a particular form of feed-forward network
  - a (often low-dimensional) code layer \( h_p \)
  - trained to reproduce the input at the output

- Training criterion

\[
E(\theta) = \sum_{p=1}^{n} f(x_p, \hat{x}_p)
\]

- Can be used to denoise data
  - “noise-corrupted” data in,
  - distance to “clean” data for training
Automatic Speech Recognition [22]

- Example Architecture from Google (2015)
  - C: CNN layer (with pooling)
  - L: LSTM layer
  - D: fully connected layer

- Two multiple layer “skips”
  - (1) connects input to LSTM input
  - (2) connects CNN output to DNN input

- Additional linear projection layer
  - reduces dimensionality
  - and number of network parameters!

- Understood the CLDNN architecture are presented in Section 4. Results on the larger data sets are then discussed in Section 5.

- We feel there is complementary information in also passing the short-timesteps, and thus consumes a larger context of \( r \) in that we pass the output of the CNN directly into the DNN, without extra layers and thus minimal parameter increase.

- The CNN takes a long-term feature, seeing a context of \( 1,024 \) to the LSTM. In fact, the original LSTM work in computer vision [10], we explore passing the output of the CNN into a DNN, without extra layers and thus minimal parameter increase.

- As shown in [5], these higher layers are appropriate for producing a linear layer, was appropriate. In our experiments, we found that adding this linear layer after the CNN layers allows for a reduction in parameters with no loss in accuracy.

- In addition, we explore if there is complementarity between the different classes we want to discriminate. Each fully connected layer has 1,024 hidden units.

- The dimension of the last layer of the CNN is large, due to the higher-order feature representation that is more easily separable into dimensionality reduction. Unless otherwise indicated, layers, where each LSTM layer has 832 cells, and a 512 unit projection layer for dimensionality reduction. Unless otherwise indicated, we set \( \tau \) = 0.

- Finally, after performing frequency and temporal modeling, we use a 9x9 frequency-time filter for the first convolutional layer, followed by a 4x3 filter for the second convolutional layer, and these features results in a negligible increase in the number of network parameters.

- For CLDNNs, \( \bullet \) •
  - Additional linear projection layer
  - reduces dimensionality
  - and number of network parameters!
Neural networks extensively used for language modelling

- compute $P(\omega_1:L)$: used in translation/ASR/topic spotting etc

- 1-of-K ("one-hot") coding for $i^{th}$ word, $\omega_i$, $x_i$
  - additional out-of-shortlist symbol may be added
  - softmax activation function on output layer
Neural networks extensively used for language modelling
  - recurrent neural networks - complete word history

\[
P(\omega_{1:L}) = \prod_{i=1}^{L} P(\omega_i|\omega_{1:i-1}) \approx \prod_{i=1}^{L} P(\omega_i|\omega_{i-1}, \tilde{h}_{i-2}) \approx \prod_{i=1}^{L} P(\omega_i|\tilde{h}_{i-1})
\]

- Input and output layer sizes can be very large
  - size of vocabulary (> 10K), not an issue for input
  - output-layer (softmax) expensive (normalisation term)

- Issues that need to be addressed
  1. **training**: how to efficiently train on billions of words?
  2. **decoding for ASR**: how to handle dependence on complete history?
Train a discriminative model from

- \( x_{1:L} = \{x_1, \ldots, x_L\} \): \( L \)-length input sequence (source language)
- \( y_{1:K} = \{y_1, \ldots, x_K\} \): \( K \)-length output (target language)

\[
p(y_{1:K}|x_{1:L}) = \prod_{i=1}^{K} p(y_i|y_{1:i-1}, x_{1:L})
\]

\[
\approx \prod_{i=1}^{L} p(y_i|y_{i-1}, \tilde{h}_{i-2}, c)
\]

need to map \( x_{1:L} \) to a fixed-length vector

\[
c = \phi(x_{1:L})
\]

\( c \) is a fixed length vector - like a sequence kernel
One form is to use hidden unit from acoustic RNN/LSTM

\[ c = \phi(x_{1:L}) = h_L \]

- dependence on context is global via \( c \) - possibly limiting
Attention-Based Models [5, 4, 16]
• Introduce **attention** layer to system
  - introduce dependence on locality \( i \)

\[
p(y_{1:K} | x_{1:L}) \approx \prod_{i=1}^{K} p(y_i | y_{i-1}, \tilde{h}_{i-1}, c_i) \approx \prod_{i=1}^{K} p(y_i | \tilde{h}_{i-1})
\]

\[
c_i = \sum_{\tau=1}^{L} \alpha_{i\tau} h_{\tau}; \quad \alpha_{i\tau} = \frac{\exp(e_{i\tau})}{\sum_{j=1}^{L} \exp(e_{ij})}, \quad e_{i\tau} = f^e(\tilde{h}_{i-2}, h_{\tau})
\]

• \( e_{i\tau} \) how well position \( i-1 \) in input matches position \( \tau \) in output
• \( h_{\tau} \) is representation (RNN) for the input at position \( \tau \)
• Encode-image as vector use a deep convolutional network
  ▪ generate caption using recurrent network (RNN/LSTM)
  ▪ all parameters optimised (using example image captions)
Conclusions
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...


