Neural network training optimization problem

\[ \min_{w} J(w) \]

The application of gradient descent to this problem is called backpropagation.

Backpropagation is gradient descent applied to \( J(w) \) in a feed-forward network.

Deriving backpropagation

- We have to evaluate the derivative \( \nabla_{w} J(w) \).
- Since \( J \) is additive over training points, \( J(w) = \sum_{n} J_{n}(w) \), it suffices to derive \( \nabla_{w} J_{n}(w) \).
The next few slides were written for a different class, and you are not expected to know their content. I show them only to illustrate the interesting way in which gradient descent interleaves with the feed-forward architecture.
Deriving backpropagation

- We have to evaluate the derivative $\nabla_w J(w)$. 
- Since $J$ is additive over training points, $J(w) = \sum_n J_n(w)$, it suffices to derive $\nabla_w J_n(w)$. 
Recall from calculus: Chain rule
Consider a composition of functions $f \circ g(x) = f(g(x))$.

\[
\frac{d(f \circ g)}{dx} = \frac{df}{dg} \frac{dg}{dx}
\]

If the derivatives of $f$ and $g$ are $f'$ and $g'$, that means:

$$\frac{d(f \circ g)}{dx}(x) = f'(g(x))g'(x)$$

Application to feed-forward network
Let $w^{(k)}$ denote the weights in layer $k$. The function represented by the network is

$$f_w(x) = f_w^{(K)} \circ \cdots \circ f_w^{(1)}(x) = f_w^{(K)}(w^{(K)} \circ \cdots \circ f_w^{(1)}(x))$$

To solve the optimization problem, we have to compute derivatives of the form

$$\frac{d}{dw} D(f_w(x_n), y_n) = \frac{dD(\cdot, y_n)}{df_w} \frac{df_w}{dw}$$
• The chain rule means we compute the derivates layer by layer.
• Suppose we are only interested in the weights of layer $k$, and keep all other weights fixed. The function $f$ represented by the network is then

$$f_{w(k)}(x) = f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f_{w(k)}^{(k)} \circ f^{(k-1)} \circ \cdots \circ f^{(1)}(x)$$

• The first $k - 1$ layers enter only as the function value of $x$, so we define

$$z^{(k)} := f^{(k-1)} \circ \cdots \circ f^{(1)}(x)$$

and get

$$f_{w(k)}(x) = f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f_{w(k)}^{(k)}(z^{(k)})$$

• If we differentiate with respect to $w^{(k)}$, the chain rule gives

$$\frac{d}{dw^{(k)}} f_{w(k)}(x) = \frac{df^{(K)}}{df^{(K-1)}} \cdots \frac{df^{(k+1)}}{df^{(k)}} \cdot \frac{df_{w(k)}^{(k)}}{dw^{(k)}}$$
Within a Single Layer

- Each $f^{(k)}$ is a vector-valued function $f^{(k)} : \mathbb{R}^{d_k} \rightarrow \mathbb{R}^{d_{k+1}}$.
- It is parametrized by the weights $w^{(k)}$ of the $k$th layer and takes an input vector $z \in \mathbb{R}^{d_k}$.
- We write $f^{(k)}(z, w^{(k)})$.

Layer-wise derivative

Since $f^{(k)}$ and $f^{(k-1)}$ are vector-valued, we get a Jacobian matrix

$$
\frac{df^{(k+1)}}{df^{(k)}} = \left( \begin{array}{ccc}
\frac{\partial f_1^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_1^{(k+1)}}{\partial f_{d_k}^{(k)}} \\
\vdots & \ddots & \vdots \\
\frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_{d_k}^{(k)}}
\end{array} \right) =: \Delta^{(k)}(z, w^{(k+1)})
$$

- $\Delta^{(k)}$ is a matrix of size $d_{k+1} \times d_k$.
- The derivatives in the matrix quantify how $f^{(k+1)}$ reacts to changes in the argument of $f^{(k)}$ if the weights $w^{(k+1)}$ and $w^{(k)}$ of both functions are fixed.
Let $w^{(1)}, \ldots, w^{(K)}$ be the current settings of the layer weights. These have either been computed in the previous iteration, or (in the first iteration) are initialized at random.

**Step 1: Forward pass**
We start with an input vector $x$ and compute

$$z^{(k)} := f^{(k)} \circ \cdots \circ f^{(1)}(x)$$

for all layers $k$.

**Step 2: Backward pass**

- Start with the last layer. Update the weights $w^{(K)}$ by performing a gradient step on

$$D \left( f^{(K)}(z^{(K)}, w^{(K)}), y \right)$$

regarded as a function of $w^{(K)}$ (so $z^{(K)}$ and $y$ are fixed). Denote the updated weights $\tilde{w}^{(K)}$.

- Move backwards one layer at a time. At layer $k$, we have already computed updates $\tilde{w}^{(K)}, \ldots, \tilde{w}^{(k+1)}$. Update $w^{(k)}$ by a gradient step, where the derivative is computed as

$$\Delta^{(K-1)}(z^{(K-1)}, \tilde{w}^{(K)}) \cdot \cdots \cdot \Delta^{(k)}(z^{(k)}, \tilde{w}^{(k+1)}) \frac{df^{(k)}}{dw^{(k)}}(z, w^{(k)})$$

On reaching level 1, go back to step 1 and recompute the $z^{(k)}$ using the updated weights.
Backpropagation is a gradient descent method for the optimization problem

$$\min_w J(w) = \sum_{i=1}^{N} D(f_w(x_i), y_i)$$

$D$ must be chosen such that it is additive over data points.

- It alternates between forward passes that update the layer-wise function values $z^{(k)}$ given the current weights, and backward passes that update the weights using the current $z^{(k)}$.
- The layered architecture means we can (1) compute each $z^{(k)}$ from $z^{(k-1)}$ and (2) we can use the weight updates computed in layers $K, \ldots, k+1$ to update weights in layer $k$. 

Not examinable.

Peter Orbanz · Applied Data Mining
Features

- Raw measurement data is typically not used directly as input for a learning algorithm. Some form of preprocessing is applied first.
- We can think of this preprocessing as a function, e.g.

\[ F : \text{raw data space} \rightarrow \mathbb{R}^d \]

(\( \mathbb{R}^d \) is only an example, but a very common one.)
- If the raw measurements are \( \mathbf{m}_1, \ldots, \mathbf{m}_N \), the data points which are fed into the learning algorithm are the images \( \mathbf{x}_n := F(\mathbf{m}_n) \).

Terminology

- \( F \) is called a feature map.
- Its dimensions (the dimensions of its range space) are called features.
- The preprocessing step (= application of \( F \) to the raw data) is called feature extraction.
This is what a typical processing pipeline for a supervised learning problem might look like.

1. **Raw data (measurements)**
2. **Feature extraction (preprocessing)**
3. **Working data**
4. **Mark patterns**
5. **Split**
   - **Training data** (patterns marked)
   - **Test data** (patterns marked)
6. **Training (calibration)**
7. **Trained model**
8. **Apply on test data**
9. **Error estimate**
Where does learning start?

- It is often a matter of definition where feature extraction stops and learning starts.
- If we have a perfect feature extractor, learning is trivial.
- For example:
  - Consider a classification problem with two classes.
  - Suppose the feature extractor maps the raw data measurements of class 1 to a single point, and all data points in class to to a single distinct point.
  - Then classification is trivial.
  - That is of course what the classifier is supposed to do in the end (e.g. map to the points 0 and 1).

Multi-layer networks and feature extraction

- An interesting aspect of multi-layer neural networks is that their early layers can be interpreted as feature extraction.
- For certain types of problems (e.g. computer vision), features were long “hand-tuned” by humans.
- Features extracted by neural networks give much better results.
- Several important problems, such as object recognition and face recognition, have basically been solved in this way.
Deep Networks as Feature Extractors

• The network on the right is a classifier $f : \mathbb{R}^d \rightarrow \{0, 1\}$.

• Suppose we subdivide the network into the first $K - 1$ layer and the final layer, by defining

$$F(x) := f^{(K-1)} \circ \ldots \circ f^{(1)}(x)$$

• The entire network is then

$$f(x) = f^{(K)} \circ F(x)$$

• The function $f^{(K)}$ is a two-class logistic regression classifier.

• We can hence think of $f$ as a feature extraction $F$ followed by linear classification $f^{(K)}$.

$$f^{(K)}(\cdot) = \sigma(\langle w^{(K)}, \cdot \rangle)$$
A SIMPLE EXAMPLE

- Problem: Classify characters into three classes (E, F and L).
- Each digit given as a $8 \times 8 = 64$ pixel image
- Neural network: 64 input units (=pixels)
- 2 hidden units
- 3 binary output units, where $f_i(x) = 1$ means image is in class $i$.
- Each hidden unit has 64 input weights, one per pixel. The weight values can be plotted as $8 \times 8$ images.
A Simple Example

- Dark regions = large weight values.
- Note the weights emphasize regions that distinguish characters.
- We can think of weight (= each pixel) as a feature.
- The features with large weights for \( h_1 \) distinguish \{E,F\} from \( L \).
- The features for \( h_2 \) distinguish \{E,L\} from \( F \).

An example for the effect of layer are autoencoders.

- An autoencoder is a neural network that is trained on its own input: If the network has weights $W$ and represents a function $f_W$, training solves the optimization problem

$$\min_W \|x - f_W(x)\|^2$$

or something similar for a different norm.

- That seems pointless at first glance: The network tries to approximate the identity function using its (possibly nonlinear) component functions.

- However: If the layers in the middle have much fewer nodes that those at the top and bottom, the network learns to *compress the input*. 
**Autoencoders**

\[ f^{(x)} \approx x \]

- Layers have same width: No effect

\[ f^{(x)} \approx x \]

- Narrow middle layers: Compression effect

- Train network on many images.
- Once trained: Input an image \( x \).
- Store \( x' := f^{(2)}(x) \). Note \( x' \) has fewer dimensions than \( x \) → compression.
- To decompress \( x' \): Input it into \( f^{(3)} \) and apply the remaining layers of the network → reconstruction \( f(x) \approx x \) of \( x \).