Neural network training optimization problem

$$\min_{\mathbf{w}} J(\mathbf{w})$$

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

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

Deriving backpropagation

- We have to evaluate the derivative $\nabla_{\mathbf{w}} J(\mathbf{w})$.
- Since J is additive over training points, $J(\mathbf{w}) = \sum_n J_n(\mathbf{w})$, it suffices to derive $\nabla_{\mathbf{w}} J_n(\mathbf{w})$.

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 $\nabla_{\mathbf{w}} J(\mathbf{w})$.
- Since J is additive over training points, $J(\mathbf{w}) = \sum_n J_n(\mathbf{w})$, it suffices to derive $\nabla_{\mathbf{w}} J_n(\mathbf{w})$.

Not examinable.

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 $\mathbf{w}^{(k)}$ denote the weights in layer k. The function represented by the network is

$$f_{\mathbf{w}}(\mathbf{x}) \;=\; f_{\mathbf{w}}^{(K)} \circ \cdots \circ f_{\mathbf{w}}^{(1)}(\mathbf{x}) \;=\; f_{\mathbf{w}^{(K)}}^{(K)} \circ \cdots \circ f_{\mathbf{w}^{(1)}}^{(1)}(\mathbf{x})$$

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

$$\frac{d}{d\mathbf{w}} D(f_{\mathbf{w}}(\mathbf{x}_n), y_n) = \frac{dD(\bullet, y_n)}{df_{\mathbf{w}}}\frac{df_{\mathbf{w}}}{d\mathbf{w}}$$

Not examinable.

Decomposing the Derivatives

- 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

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

- The first $k-1$ layers enter only as the function value of $\mathbf{x}$, so we define

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

  and get

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

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

$$\frac{d}{d\mathbf{w}^{(k)}} f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; \frac{df^{(K)}}{df^{(K-1)}} \cdots \frac{df^{(k+1)}}{df^{(k)}} \;\cdot\; \frac{df^{(k)}_{\mathbf{w}^{(k)}}}{d\mathbf{w}^{(k)}}$$

Not examinable.

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

Layer-wise derivative

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

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

- $\Delta^{(k)}$ 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 $\mathbf{w}^{(k+1)}$ and $\mathbf{w}^{(k)}$ of both functions are fixed.

Not examinable.

Let $\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(K)}$ 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 $\mathbf{x}$ and compute

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

for all layers k.

Step 2: Backward pass

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

  $$D\big(f^{(K)}(\mathbf{z}^{(K)}, \mathbf{w}^{(K)}), y\big)$$

  regarded as a function of $\mathbf{w}^{(K)}$ (so $\mathbf{z}^{(K)}$ and y are fixed). Denote the updated weights $\tilde{\mathbf{w}}^{(K)}$.
- Move backwards one layer at a time. At layer k, we have already computed updates $\tilde{\mathbf{w}}^{(K)}, \ldots, \tilde{\mathbf{w}}^{(k+1)}$. Update $\mathbf{w}^{(k)}$ by a gradient step, where the derivative is computed as

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

On reaching level 1, go back to step 1 and recompute the $\mathbf{z}^{(k)}$ using the updated weights.

Not examinable.

SUMMARY: BACKPROPAGATION

- Backpropagation is a gradient descent method for the optimization problem

$$\min_{\mathbf{w}} J(\mathbf{w}) = \sum_{i=1}^{N} D(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$$

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

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

Not examinable.

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.

$$\mathbf{F}\text{: raw data space} \longrightarrow \mathbb{R}^d$$

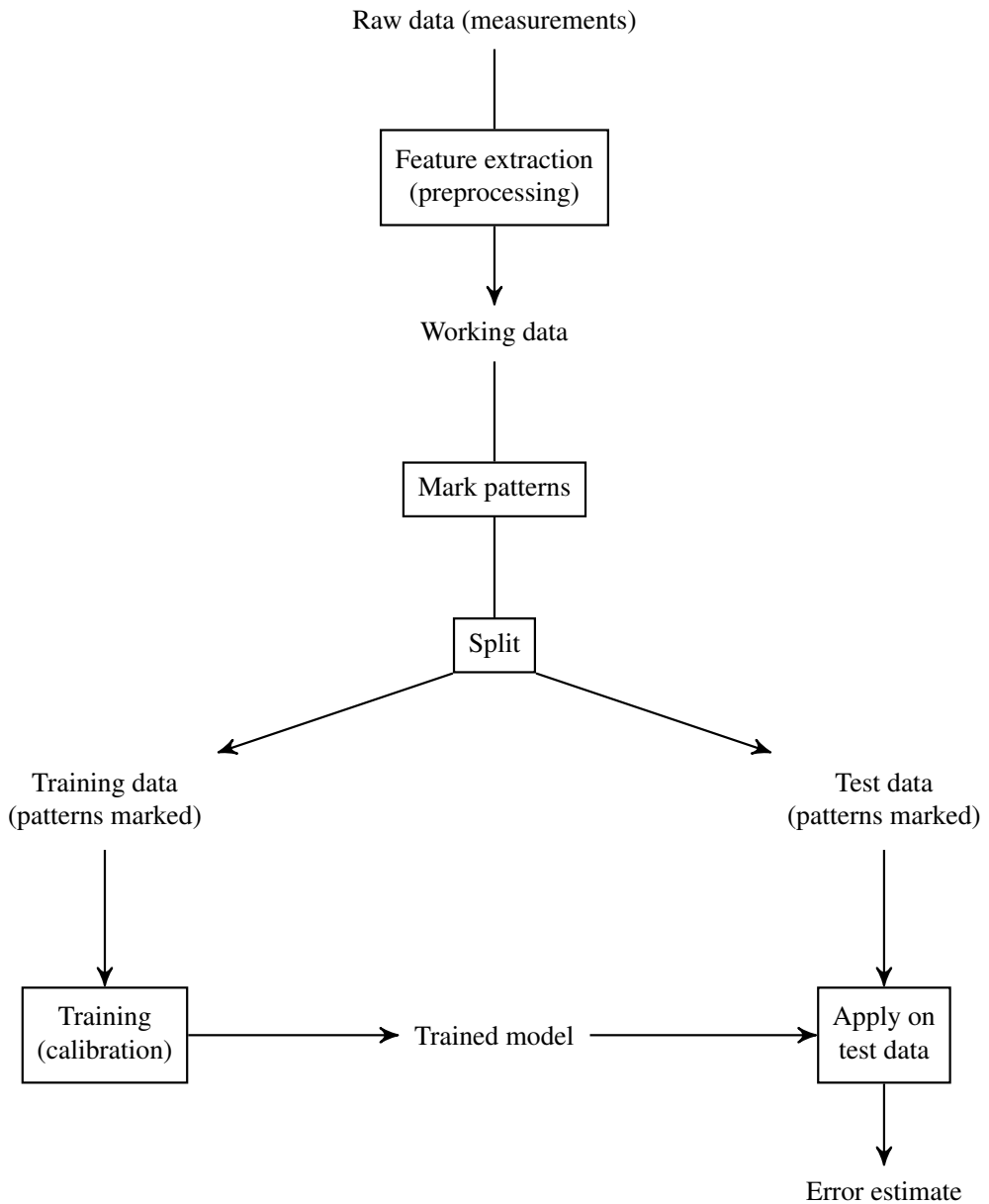
  ($\mathbb{R}^d$ is only an example, but a very common one.)
- If the raw measurements are $\mathbf{m}_1, \ldots, \mathbf{m}_N$, the data points which are fed into the learning algorithm are the images $\mathbf{x}_n := \mathbf{F}(\mathbf{m}_n)$.

Terminology

- $\mathbf{F}$ is called a **feature map**.
- Its dimensions (the dimensions of its range space) are called **features**.
- The preprocessing step (= application of $\mathbf{F}$ to the raw data) is called **feature extraction**.

Example Processing Pipeline

This is what a typical processing pipeline for a supervided learning propblem might look like.



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 classfication 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 intepreted 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

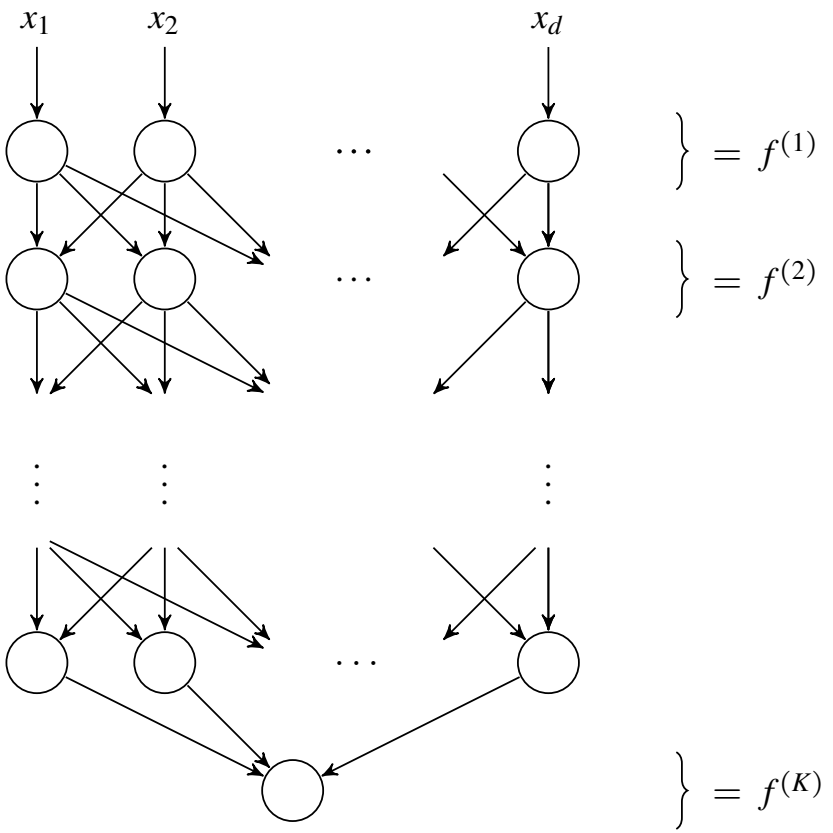
- The network on the right is a classifier $f : \mathbf{R}^d \to \{0, 1\}$.
- Suppose we subdivide the network into the first $K - 1$ layer and the final layer, by defining

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

- The entire network is then

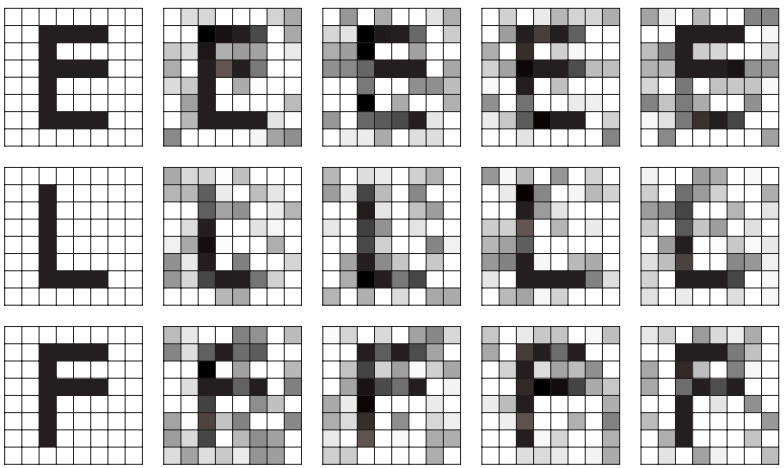
$$f(\mathbf{x}) = f^{(K)} \circ \mathbf{F}(\mathbf{x})$$

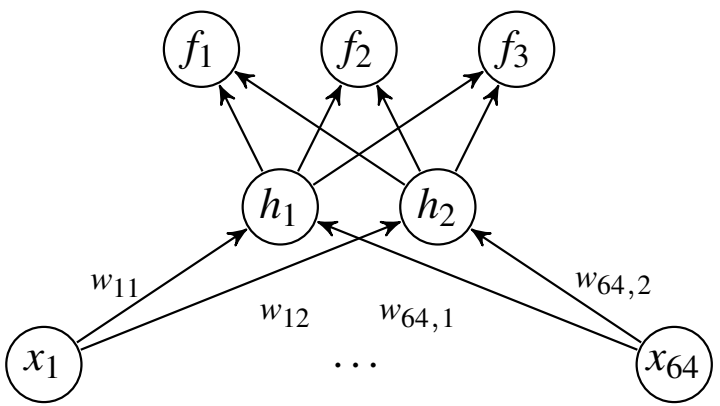
- The function $f^{(K)}$ is a two-class logistic regression classifier.
- We can hence think of f as a feature extraction $\mathbf{F}$ followed by linear classification $f^{(K)}$.



$$f^{(K)}(\bullet) = \sigma(\langle \mathbf{w}^{(K)}, \bullet \rangle)$$

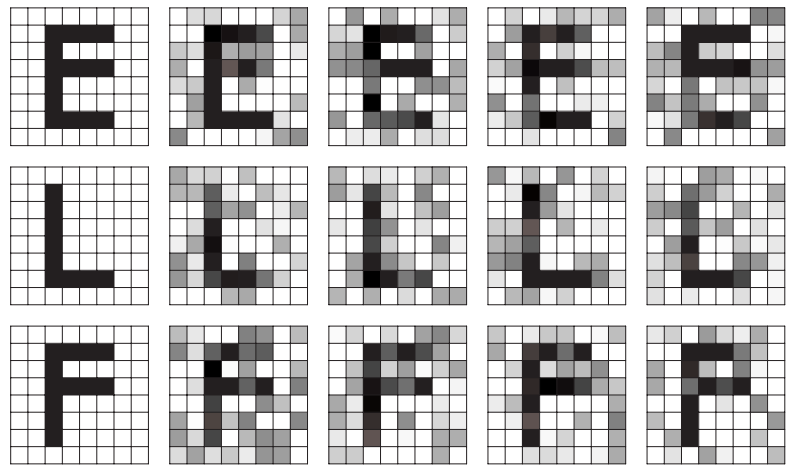
A Simple Example



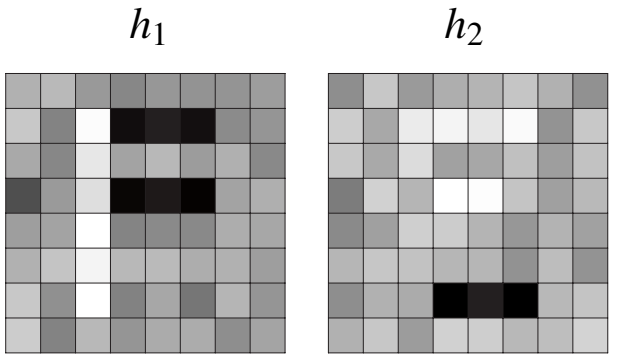


- Problem: Classify characters into three classes (E, F and L).
- Each digit given as a $8 \times 8 = 64$ pixel image
- Neural network: 64 input units (=pixels)
- 2 hidden units
- 3 binary output units, where $f_i(\mathbf{x}) = 1$ means image is in class i.
- Each hidden unit has 64 input weights, one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

A Simple Example



training data (with random noise)



weight values of h_1 and h_2 plotted as images

- 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.

Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

EXAMPLE: AUTOENCODERS

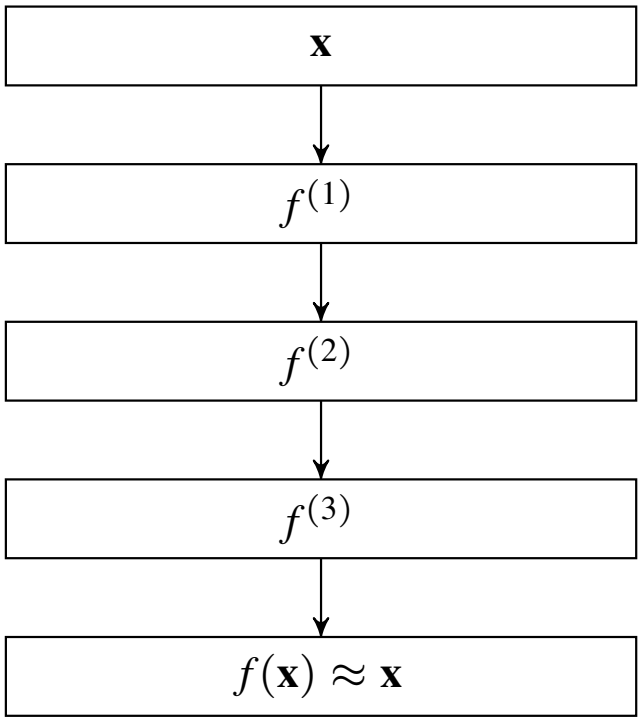
An example for the effect of layer are autoencoders.

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

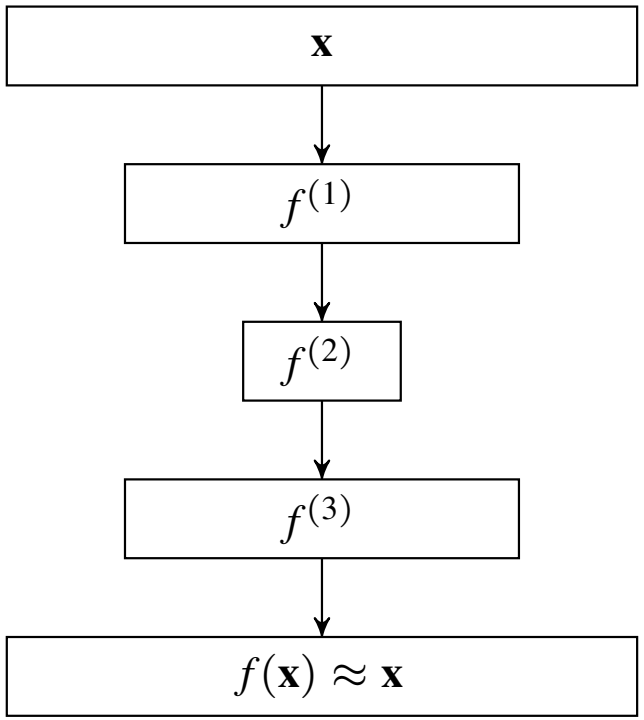
$$\min_{\mathbf{W}} \|\mathbf{x} - f_{\mathbf{W}}(\mathbf{x})\|^2$$

  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



Layers have same width: No effect



Narrow middle layers: Compression effect

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

AUTOENCODERS

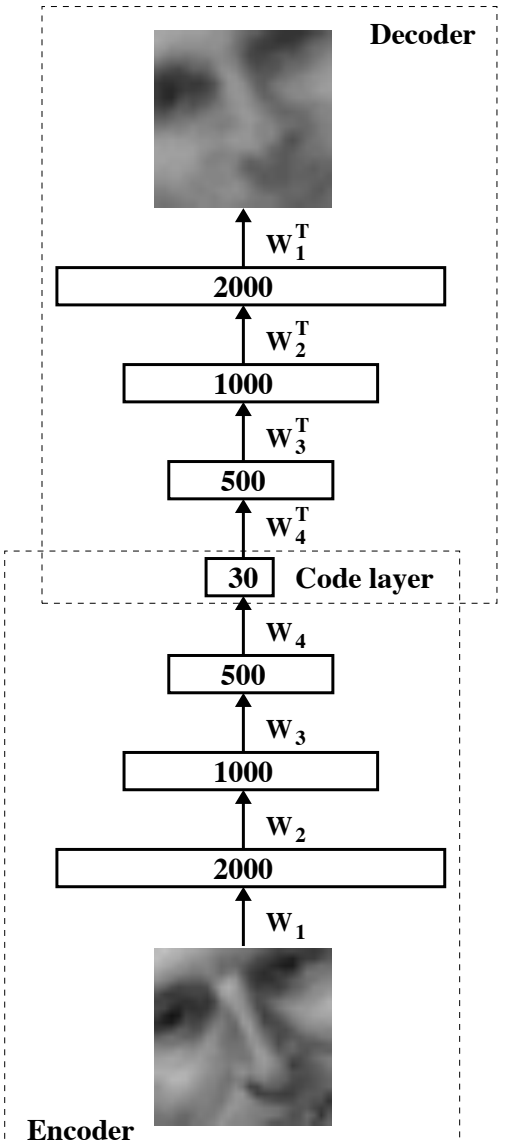


Illustration: K. Murphy, *Machine Learning: A Bayesian perspective*, MIT Press 2012