STA 437/2005:
Methods for Multivariate Data

Week 9: Neural Networks and Autoencoders

Piotr Zwiernik

University of Toronto

Table of contents

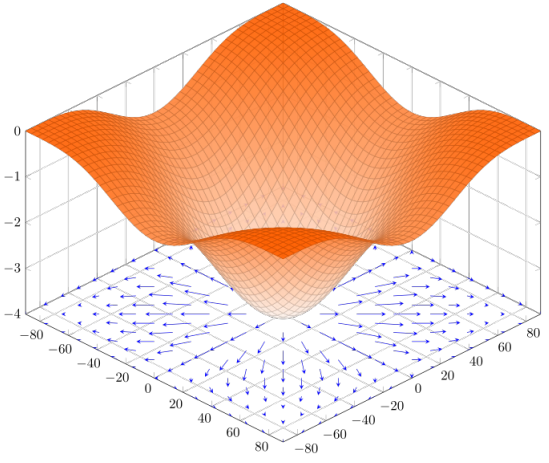
Gradient descent

Gradients

Differentiable function $f : \mathbb{R}^d \to \mathbb{R}$,
$\mathbf{w} = (w_1, \ldots, w_d)$,
gradient of f at w

$$\nabla f(\mathbf{w}) = \begin{bmatrix} \frac{\partial f}{\partial w_1}(\mathbf{w}) \\ \vdots \\ \frac{\partial f}{\partial w_d}(\mathbf{w}) \end{bmatrix}$$

$$f(\mathbf{w} + \eta \mathbf{u}) \approx f(\mathbf{w}) + \eta \nabla f(\mathbf{w})^\top \mathbf{u}$$



Important geometric interpretation of the gradient

The gradient gives the direction of the steepest local increase of f.

Error minimization

- Training statistical models always reduces to solving an optimization problem

$$\text{minimize}_{\mathbf{w}}\ E(\mathbf{w}), \qquad \mathbf{w}^* := \arg\min_{\mathbf{w}}\ E(\mathbf{w}).$$

- Standard approach is gradient descent $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t)$, where $\eta \in (0, 1]$ is the step size (aka learning rate) with w^0 some initial point.

- For the least squares, $\text{minimize}\ E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n^\top \mathbf{w} - t_n)^2$ we have

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} \mathbf{x}_n(\mathbf{x}_n^\top \mathbf{w} - t_n).$$

Gradient descent derivation

- Suppose we are at $\mathbf{w}$ and we want to pick a direction $\mathbf{u}$ such that $E(\mathbf{w} + \eta\mathbf{u})$ is smaller than $E(\mathbf{w})$ for a step size η, $\|\mathbf{u}\| = 1$.
- The first-order Taylor series approximation of $E(\mathbf{w} + \eta\mathbf{u})$ around $\mathbf{w}$ is:

$$E(\mathbf{w} + \eta\mathbf{u}) \;=\; E(\mathbf{w}) + \eta\nabla E(\mathbf{w})^\top \mathbf{u} + o(\eta) \;\approx\; E(\mathbf{w}) + \eta\nabla E(\mathbf{w})^\top \mathbf{u}.$$

- Direction u should have a negative inner product with $\nabla E(\mathbf{w})$, e.g. $-\frac{\nabla E(\mathbf{w})}{\|\nabla E(\mathbf{w})\|}$.
- This approximation gets better as η gets smaller.

How do we choose the step size in GD? $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\nabla E(\mathbf{w}^t)$

- Simple strategy: start with a big η and progressively make it smaller by e.g. halving it until the function decreases.
- The sequence of step sizes is referred to as learning rate schedule.

When did the GD converge?

- The vector $\mathbf{w}$ is a fixed point if $\nabla E(\mathbf{w}) = \mathbf{0}$.
- This is never possible in practice. So we stop iterations if gradient is smaller than a threshold, $\|\nabla E(\mathbf{w})\| < \tau$.
- If the function is convex then we have reached a global minimum.
- If the function is not convex, what did we obtain?
- Probably a local minimum or a saddle point.

Stochastic Gradient Descent

In most cases, we minimize an average over data points:

$$E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(t_n, y(\mathbf{x}_n, \mathbf{w})), \qquad \nabla E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{n} \nabla L(t_n, y(\mathbf{x}_n, \mathbf{w})),$$

which is hard to compute when N is very large.

At each iteration, use a sub-sample of data to estimate the gradient

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|S|} \sum_{n \in S} \nabla L(t_n, y(\mathbf{x}_n, \mathbf{w})).$$

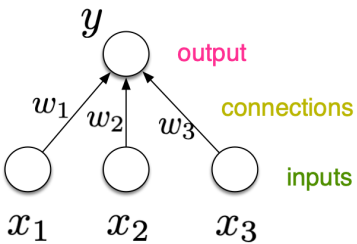
(Here $|S|$ denotes the number of elements in the set S. Standard SGD has $|S| = 1$)

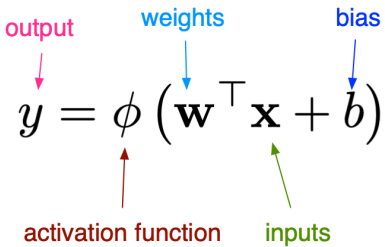
ML terminology: Computing gradients using the full dataset is called batch learning, using subsets of data is called mini-batch learning.

Introducing neural networks

A Simpler Neuron

For neural nets, we use a simple model for neuron, or **unit**:

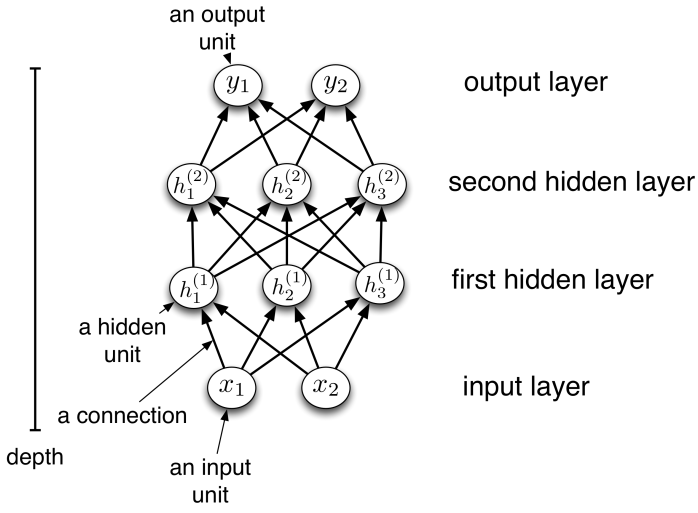




- By throwing together lots of these simple neuron-like processing units, we can do some powerful computations!

A Feed-Forward Neural Network

- A directed acyclic graph
- Units are grouped into layers

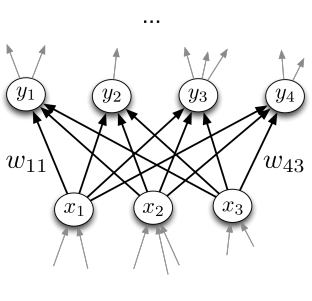


Multilayer Perceptrons

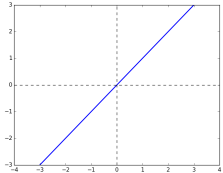
- A multi-layer network consists of fully connected layers.
- In a fully connected layer, all input units are connected to all output units.
- The outputs are a function of the input units:

$$\mathbf{y} = f(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$\phi : \mathbb{R} \to \mathbb{R}$ is applied component-wise.

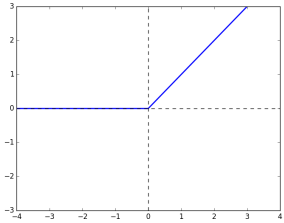


Some Activation Functions



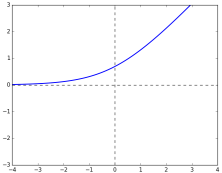
Identity

$\phi(z) = z$



Rectified Linear Unit (ReLU)

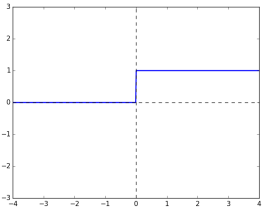
$\phi(z) = \max(0, z)$



Soft ReLU

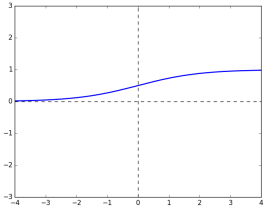
$\phi(z) = \log 1 + e^z$

More Activation Functions



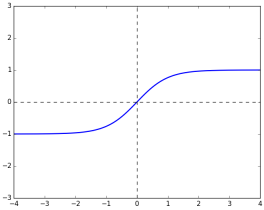
Hard Threshold

$$\phi(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



Logistic

$$\phi(z) = \frac{1}{1 + e^{-z}}$$



Hyperbolic Tangent (tanh)

$$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Computation in Each Layer

Each layer computes a function.

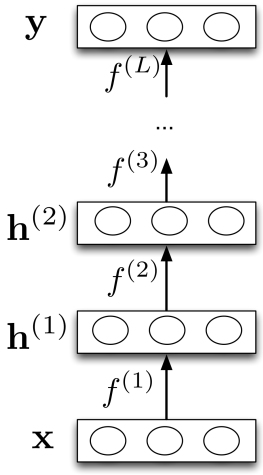
$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x}) = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)}) = \phi(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$
$$\vdots$$
$$\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)})$$

The network computes a composition of functions.

$$\mathbf{y} = f^{(L)} \circ \cdots \circ f^{(1)}(\mathbf{x}).$$

The last layer depends on the task.

- Regression: $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = (\mathbf{w}^{(L)})^\top \mathbf{h}^{(L-1)} + b^{(L)}$
- Classification: $\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)}) = \sigma\big((\mathbf{w}^{(L)})^\top \mathbf{h}^{(L-1)} + b^{(L)}\big)$



Expressive Power of Linear Networks

- Consider a linear layer: the activation function was the identity. The layer just computes an affine transformation of the input.
- Any sequence of linear layers is equivalent to a single linear layer.

$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'}\mathbf{x}$$

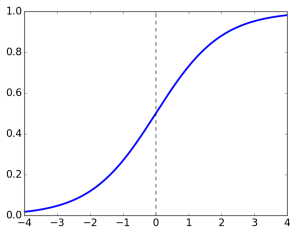
- Deep linear networks can only represent linear functions — no more expressive than linear regression.

Expressive Power of Non-linear Networks

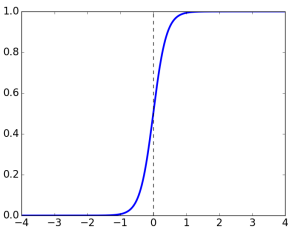
- Multi-layer feed-forward neural networks with non-linear activation functions
- **Universal Function Approximators**: They can approximate any function arbitrarily well.
- True for various activation functions (e.g. thresholds, logistic, ReLU, etc.)

Expressivity of the Logistic Activation Function

- What about the logistic activation function?
- Approximate a hard threshold by scaling up w and b.



$y = \sigma(x)$



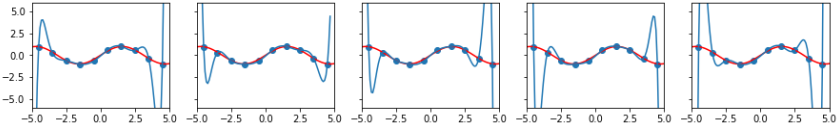
$y = \sigma(5x)$

- Logistic units are differentiable, so we can learn weights with gradient descent.

What is Expressivity Good For?

- May need a very large network to represent a function.
- Non-trivial to learn the weights that represent a function.
- If you can learn any function, over-fitting is potentially a serious concern!

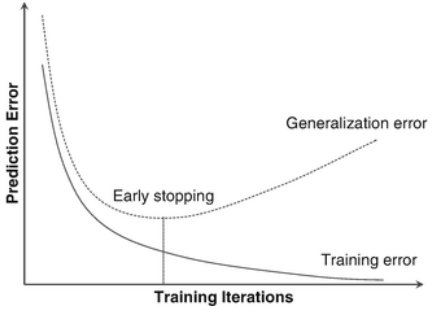
  For the polynomial feature mappings, expressivity increases with the degree M, eventually allowing multiple perfect fits to the training data. This motivated L^2 regularization.



- Do neural networks over-fit and how can we regularize them?

Regularization and Over-fitting for Neural Networks

- The topic of over-fitting (when & how it happens, how to regularize, etc.) for neural networks is not well-understood, even by researchers!
  - In principle, you can always apply L^2 regularization.
- A common approach is early stopping, or stopping training early, because over-fitting typically increases as training progresses.



- Benign overfitting is a heavily studied phenomenon.

Backpropagation

Learning Weights in a Neural Network

- Goal is to learn weights in a multi-layer neural network using gradient descent.
- Weight space for a multi-layer neural net: one set of weights for each unit in every layer of the network
- Define a loss $\mathcal{L}(t, y) = \mathcal{L}(t, y(x, \mathbf{w}))$ and compute the gradient of the cost

$$E(\mathbf{w}) \;=\; \tfrac{1}{N} \sum_{n=1}^{N} \mathcal{L}(t_n, y(x_n, \mathbf{w})),$$

  which is the average loss over all the training examples.
- How we can calculate $\nabla E(\mathbf{w})$ efficiently?

Example: Two-Layer Neural Network

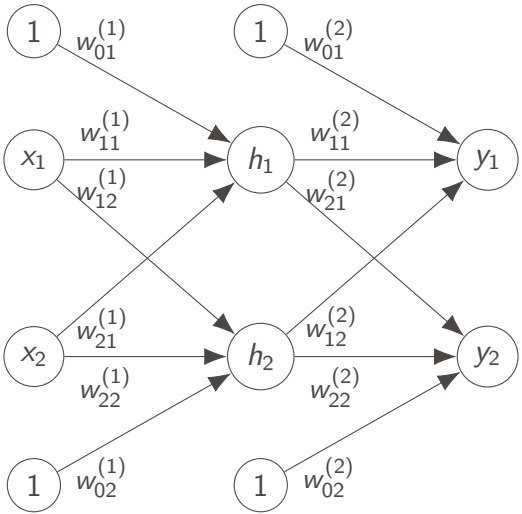
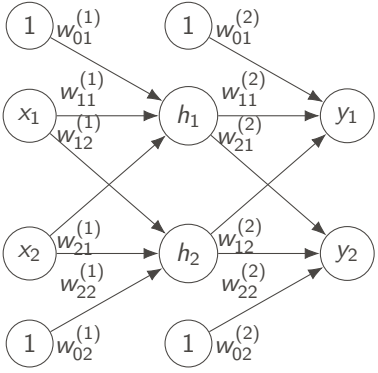


Figure 1: Two-Layer Neural Network

Computations for Two-Layer Neural Network

A neural network computes a composition of functions.



$$z_1^{(1)} = w_{01}^{(1)} \cdot 1 + w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2$$

$$h_1 = \sigma(z_1)$$

$$z_1^{(2)} = w_{01}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1 + w_{21}^{(2)} \cdot h_2$$

$$y_1 = z_1$$

$$z_2^{(1)} =$$

$$h_2 =$$

$$z_2^{(2)} =$$

$$y_2 =$$

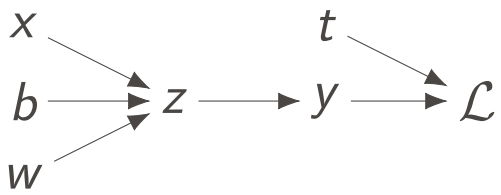
$$\mathcal{L} = \frac{1}{2}\left((y_1 - t_1)^2 + (y_2 - t_2)^2\right)$$

Simplified Example: Logistic Least Squares

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Computation Graph:

- The nodes represent the inputs and computed quantities.
- The edges represent which nodes are computed directly as a function of which other nodes.

Univariate Chain Rule

Let $z = f(y)$ and $y = g(x)$ be uni-variate functions.
Then $z = f(g(x))$.

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\,\frac{\mathrm{d}y}{\mathrm{d}x}$$

Logistic Least Squares: Gradient for w

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the gradient for w:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y}\,\frac{\partial y}{\partial w} \\ &= \frac{\partial \mathcal{L}}{\partial y}\,\frac{\partial y}{\partial z}\,\frac{\partial z}{\partial w} \\ &= (y - t)\;\sigma'(z)\;x \\ &= (\sigma(wx + b) - t)\sigma'(wx + b)x\end{aligned}$$

Logistic Least Squares: Gradient for b

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the gradient for b:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial b} \\
&= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial b} \\
&= (y - t)\ \sigma'(z)\ 1 \\
&= (\sigma(wx + b) - t)\sigma'(wx + b)1
\end{aligned}$$

Comparing Gradient Computations for w and b

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the gradient for w:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w} \\ &= (y - t)\ \sigma'(z)\ x\end{aligned}$$

Computing the gradient for b:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial b} \\ &= (y - t)\ \sigma'(z)\ 1\end{aligned}$$

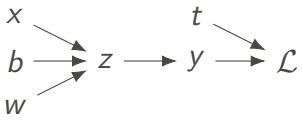
Drawbacks

- For larger networks these computations become cumbersome
- There will be many repeated terms, e.g. $\sigma'(z)$ appears on both sides.

Structured Way of Computing Gradients

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Computing the gradients:

$$\frac{\partial \mathcal{L}}{\partial y} = (y - t)$$
$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial y}\,\sigma'(z)$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,1$$

Error Signal Notation

- Let $\overline{y}$ denote the derivative $\mathrm{d}\mathcal{L}/\mathrm{d}y$, called the error signal.
- Error signals are just values our program is computing (rather than a mathematical operation).

Computing the loss:

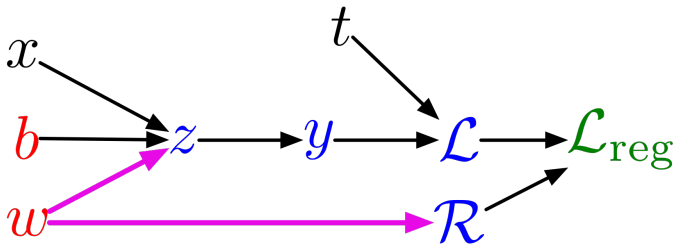
$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\overline{y} = (y - t)$$
$$\overline{z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,x \qquad \overline{b} = \overline{z}$$

(previous slide: $\frac{\partial \mathcal{L}}{\partial y} = (y - t)$, $\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial y}\,\sigma'(z)$, $\frac{\partial \mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x$, $\frac{\partial \mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,1$)

Computation Graph has a Fan-Out > 1



$$z = wx + b$$

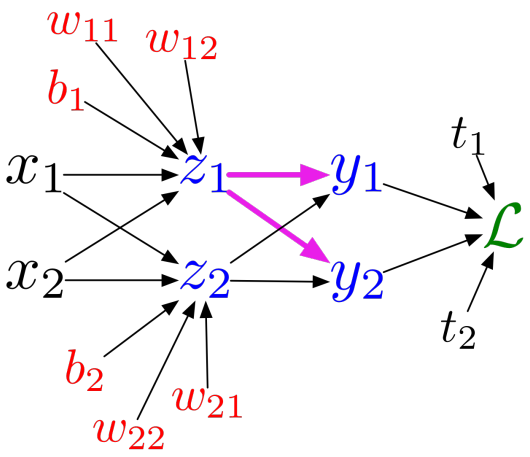
$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Computation Graph has a Fan-Out > 1



$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$

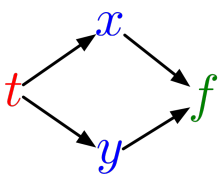
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$

$$\mathcal{L} = -\sum_k t_k \log y_k$$

Multi-variate Chain Rule

Suppose we have functions $f(x, y)$, $x(t)$, and $y(t)$.

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$



Example:

$$f(x, y) = y + e^{xy}$$
$$x(t) = \cos t$$
$$y(t) = t^2$$

$$\begin{aligned}\frac{\mathrm{d}f}{\mathrm{d}t} &= \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} \\ &= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t\end{aligned}$$

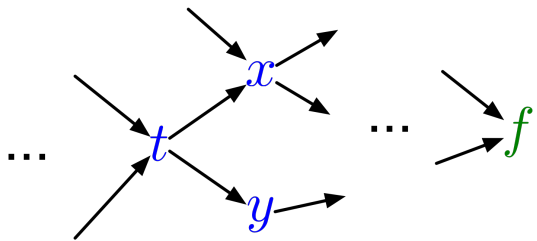
Multi-variate Chain Rule

In the context of back-propagation:

Mathematical expressions to be evaluated

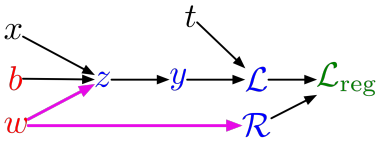
$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed by our program



In our new notation: $\overline{t} = \overline{x}\frac{\mathrm{d}x}{\mathrm{d}t} + \overline{y}\frac{\mathrm{d}y}{\mathrm{d}t}$

Backpropagation for Regularized Logistic Least Squares



Forward pass:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Backward pass:

$$\overline{\mathcal{L}_{\text{reg}}} = 1$$

$$\overline{\mathcal{R}} = \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} = \lambda$$

$$\overline{\mathcal{L}} = \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} = 1$$

$$\overline{y} = \overline{\mathcal{L}}\,\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = \overline{\mathcal{L}}\,(y - t)$$

$$\overline{z} = \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}z} = \overline{y}\,\sigma'(z)$$

$$\overline{w} = \overline{z}\,\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} = \overline{z}\,x + \overline{\mathcal{R}}\,w$$

$$\overline{b} = \overline{z}\,\frac{\partial z}{\partial b} = \overline{z}$$

Full Backpropagation Algorithm:

Let $v_1, \ldots, v_N$ be an ordering of the computation graph where parents come before children (aka topological ordering).

v_N denotes the variable for which we try to compute gradients ($\mathcal{L}$, $\mathcal{L}_{\text{reg}}$ etc).

- forward pass:

$$\text{For } i = 1, \ldots, N,$$
$$\text{Compute } v_i \text{ as a function of Parents}(v_i).$$

- backward pass:

$$\bar{v}_N = 1$$
$$\text{For } i = N-1, \ldots, 1,$$
$$\bar{v}_i = \sum_{j \in \text{Children}(v_i)} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

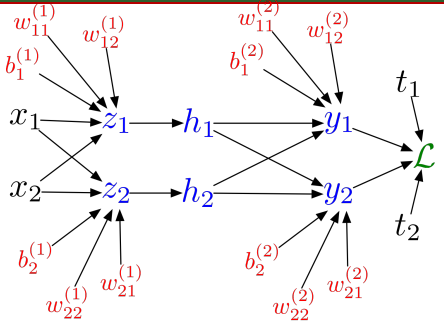
Backpropagation

- The algorithm for efficiently computing gradients in neural nets.
- Gradient descent with gradients computed via backprop is used to train the overwhelming majority of neural nets today.
- Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.

Auto-Differentiation

- Autodifferentiation performs backprop in a completely mechanical and automatic way.
- Many autodiff libraries: PyTorch, Tensorflow, Jax, etc.
- Although autodiff automates the backward pass for you, it's still important to know how things work under the hood.
- In the tutorial, we will use an autodiff framework to build complex neural networks.

Backpropagation for Two-Layer Neural Network



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}, \quad h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}, \quad \mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}} \, (y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k} \, h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

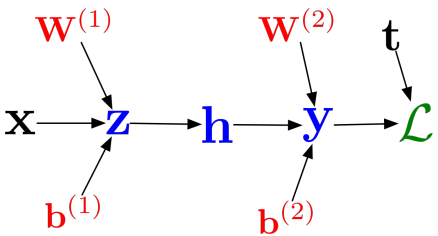
$$\overline{z_i} = \overline{h_i} \, \sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i} \, x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

Backpropagation for Two-Layer Neural Network

In vectorized form:



Forward pass:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \quad \mathbf{h} = \sigma(\mathbf{z})$$

$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}, \quad \mathcal{L} = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{\mathbf{y}} = \overline{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$

$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^\top$$

$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$

$$\overline{\mathbf{h}} = \mathbf{W}^{(2)\top}\overline{\mathbf{y}}$$

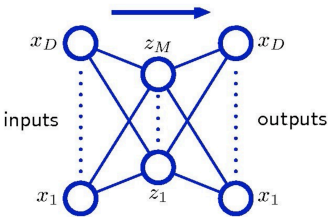
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^\top$$

$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

Autoencoders

Non-linear Dimension Reduction



- Neural networks can be used for nonlinear dimensionality reduction.
- This is achieved by having the same number of outputs as inputs. These models are called autoencoders.
- Consider a feed-forward neural network that has D inputs, D outputs, and M hidden units, with $M < D$.
- We can squeeze the information through a bottleneck.
- If we use a linear network (linear activation) this is very similar to Principal Components Analysis.

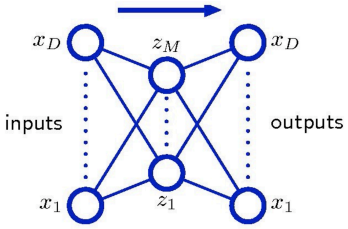
Autoencoders and PCA

- Given an input $\boldsymbol{x}$, its corresponding reconstruction is given by:

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \sum_{j=1}^{M} w_{kj}^{(2)} \sigma\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i \Big), \quad k = 1, ..., D.$$

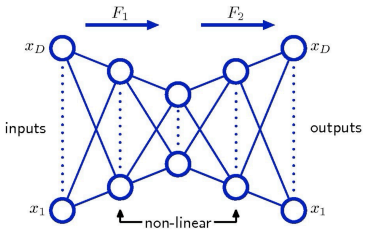
- We learn the parameters $\boldsymbol{w}$ by minimizing the reconstruction error:

$$E(\boldsymbol{w}) = \frac{1}{2N} \sum_{n=1}^{N} \| y(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{x}_n \|^2$$



- In the case when layers are linear:
  - it will learn hidden units that are linear functions of the data and minimize squared error.
  - M hidden units will span the same space as the first M principal components (PCA).

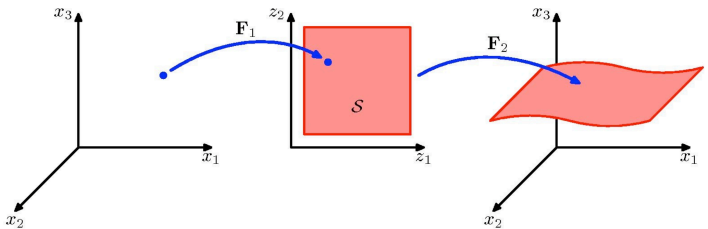
Deep Autoencoders



- We can put extra nonlinear hidden layers between the input and the bottleneck and between the bottleneck and the output.
- This gives nonlinear generalization of PCA, providing non-linear dimensionality reduction.
- The network can be trained by the minimization of the reconstruction error function.
- Much harder to train.

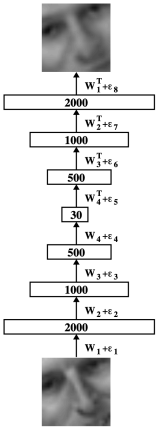
Geometrical Interpretation

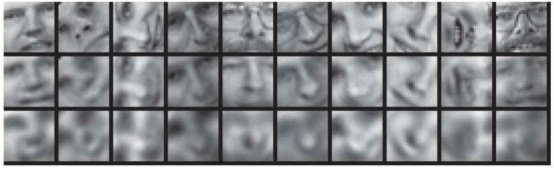
- Geometrical interpretation of the mappings performed by the network with 2 hidden layers for the case of $D = 3$ and $M = 2$ units in the middle layer.



- The mapping F_1 defines a nonlinear projection of points in the original D-space into the M-dimensional subspace.
- The mapping F_2 maps from an M-dimensional space into D-dimensional space.

Deep Autoencoders





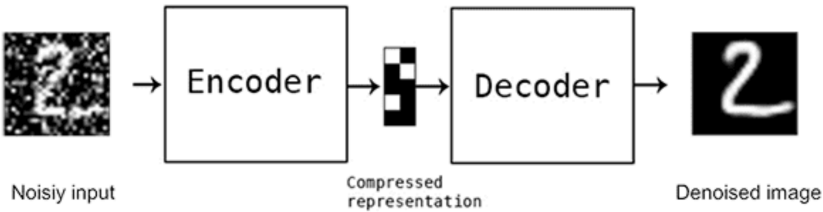
- We can consider very deep autoencoders.
- By row: Real data, Deep autoencoder with a bottleneck of 30 units, and 30-d PCA.

Deep Autoencoders

- Similar model for MNIST handwritten digits:

- Deep autoencoders produce much better reconstructions.

Application: Image Denoising



- We can train a **denoising autoencoder**.
- We feed noisy image as an input to the encoder
- Minimize the reconstruction error between the decoder output and original image.
- This method requires training and knowledge of the noise structure (fully supervised).
- In contrast, loopy BP works for a single noisy image and doesn't require the knowledge of noise structure (unsupervised).

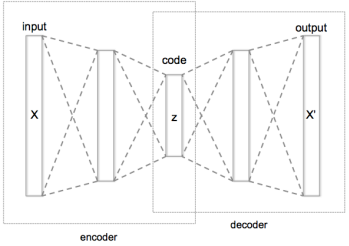
Autoencoders: Summary

Autoencoders reconstruct their input via an encoder and a decoder.

- **Encoder**: $e(x) = z \in F, \quad x \in X$
- **Decoder**: $d(z) = \tilde{x} \in X$
- where X is the data space, and F is the feature (latent) space.
- z is the code, compressed representation of the input, x. It is important that this code is a bottleneck, i.e. that

$$\dim F \ll \dim X$$

- Goal: $\tilde{x} = d(e(x)) \approx x$.



Issues with (deterministic) Autoencoders

- **Issue 1**: Proximity in data space does not mean proximity in feature space
  - The codes learned by the model are deterministic, i.e.

$$g(x_1) = z_1 \implies f(z_1) = \tilde{x}_1$$
$$g(x_2) = z_2 \implies f(z_2) = \tilde{x}_2$$

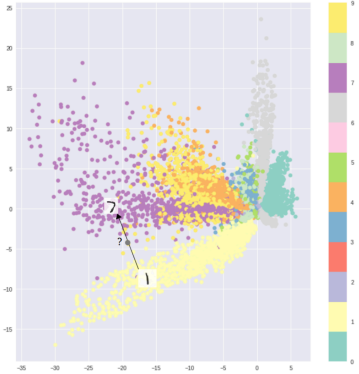
  - but proximity in feature space is not "directly" enforced for inputs in close proximity in data space, i.e.

$$x_1 \approx x_2 \not\implies z_1 \approx z_2$$

  - The latent space may not be continuous, or allow easy interpolation.

Issues with (deterministic) Autoencoders

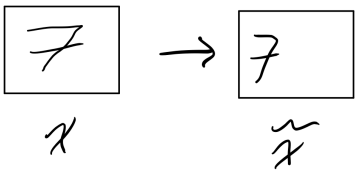
- **Issue 1**: Proximity in data space does not mean proximity in feature space
  - If the space has discontinuities (eg. gaps between clusters) and you sample/generate a variation from there, the decoder will simply generate an unrealistic output.



Two dimensional latent space for the MNIST digit data.(Image credit: I. Shafkat)

Issues with (deterministic) Autoencoders

- **Issue 2**: How to measure the goodness of a reconstruction?



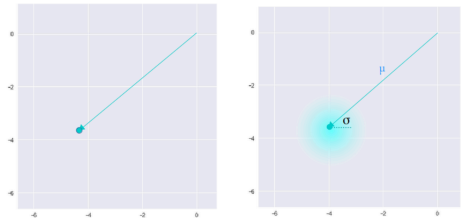
  - The reconstruction looks quite good. However, if we chose a simple distance metric between inputs and reconstructions, we would heavily penalize the left-shift in the reconstruction $\tilde{x}$.
  - Choosing an appropriate metric for evaluating model performance can be difficult, and that a miss-aligned objective can be disastrous.

Variational Autoencoders

- Variational autoencoders (VAEs) encode inputs with uncertainty.
- Unlike standard autoencoders, the encoder of a VAE outputs a probability distribution, $q_\phi(z)$ to approximate $p(z|x)$.
- We assume that q_ϕ is a product of univariate normal distributions.
- Instead of the encoder learning an encoding vector, it learns two vectors: vector of means, μ, and another vector of standard deviations, σ.

Variational Autoencoders

- The mean μ controls where encoding of input is centered while the standard deviation controls how much can the encoding vary.



Standard Autoencoder
(direct encoding coordinates)

Variational Autoencoder
(μ and σ initialize a probability distribution)

Image credit: I. Shafkat

- Encodings are generated at random from the "ball", the decoder learns that all nearby points refer to the same input.

To minimize the reconstruction error, the algorithm may want to concentrate around the direct encodings. We will try to discourage this.

VAE: Specifics

- Our model is generated by the joint distribution over the latent codes and the input data $p_\theta(x, z)$:
  - The prior: $z \sim p_\theta(z)$ often Gaussian.
  - The likelihood: $x|z \sim \mathrm{Expfam}(x|d_\theta(z))$ with the decoder $d_\theta(z)$ given by a DNN,
    - e.g. for binary observations: $p_\theta(x|z) = \prod_{d=1}^{D} \mathrm{Ber}(x_d|\sigma(d_\theta(z)))$
    - e.g. for continuous observations: $p_\theta(x|z) = \prod_{d=1}^{D} N(x_d|d_\theta(z))$
- We could use the posterior $p_\theta(z|x) = p_\theta(x, z)/p_\theta(x)$ for encoding.

Issue: Learning $p(x) = \int p(x|z)p(z)dz$ is intractable.

Introduce an approximation q_ϕ with its own set of parameters ϕ, such that

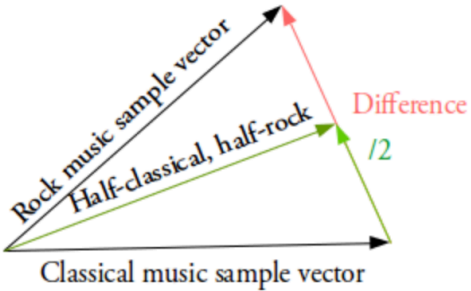
$$q_\phi(z|x) \approx p_\theta(z|x).$$

After VAE is trained

- Once a VAE is trained, we can sample new inputs (generative model)

$$z \sim p(z) \qquad \tilde{x} \sim p_\theta(x|z)$$

- We can also interpolate between inputs, using simple vector arithmetic.



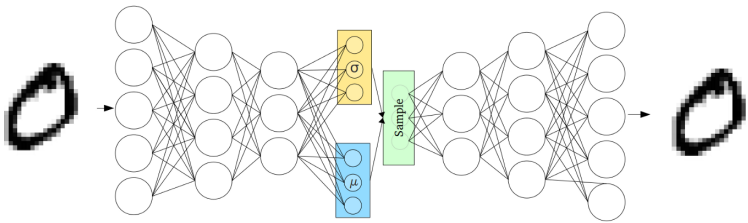
Interpolating between samples

Example: MNIST

- We choose the prior on z to be the standard Gaussian: $p(z) \sim \mathcal{N}(0, I)$.
- The likelihood function: $p_\theta(x|z) = \prod_{i=1}^{D} \text{Bernoulli}(\theta_i)$.
- Approximate posterior: $q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \Sigma_\phi(x))$, where Σ_ϕ diagonal.
- Finally, we use neural networks as our encoder and decoder
  - **Encoder**: $e_\phi(x) = [\mu_\phi(x), \log \Sigma_\phi(x)]$
  - **Decoder**: $d_\theta(z) = (\theta_1(z), \ldots, \theta_d(z))$
  - where θ_i are parameters of a Bernoulli rv for each input pixel.
- To get our reconstructed input, we simply take

$$\tilde{x} \sim p_\theta(x|z)$$

Example: MNIST



- We use neural networks for both the encoder and the decoder.
- We compute the loss function $\mathcal{L}(\theta, \phi; x)$ and propagate its derivative with respect to θ and ϕ, $\nabla_\theta \mathcal{L}$, $\nabla_\phi \mathcal{L}$, through the network during training.
- We use reparametrization trick as described before.

MNIST: Autoencoder vs VAE

Codes generated by L: AE R: VAE

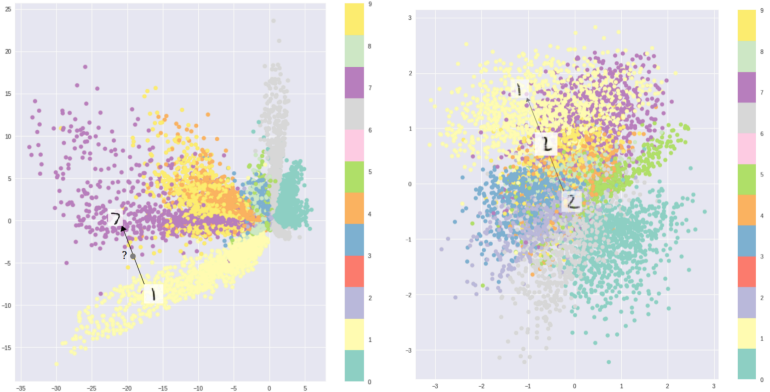


Image credit: I. Shafkat