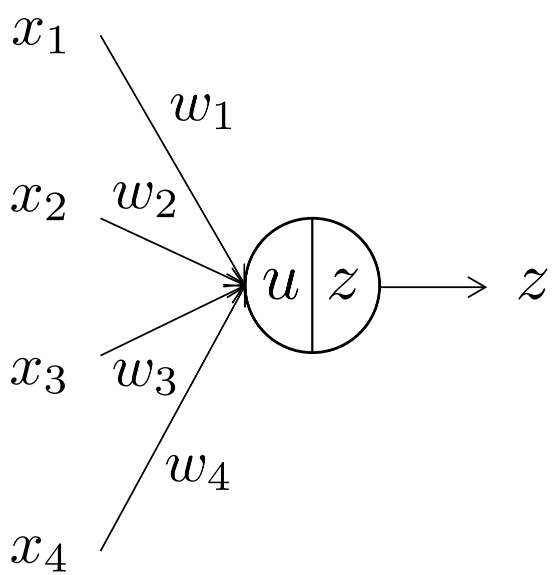
13. Machine learning II

- Neural networks (deep learning)
- Standardization of data
- Training neural networks

Neural networks: Units and activation functions

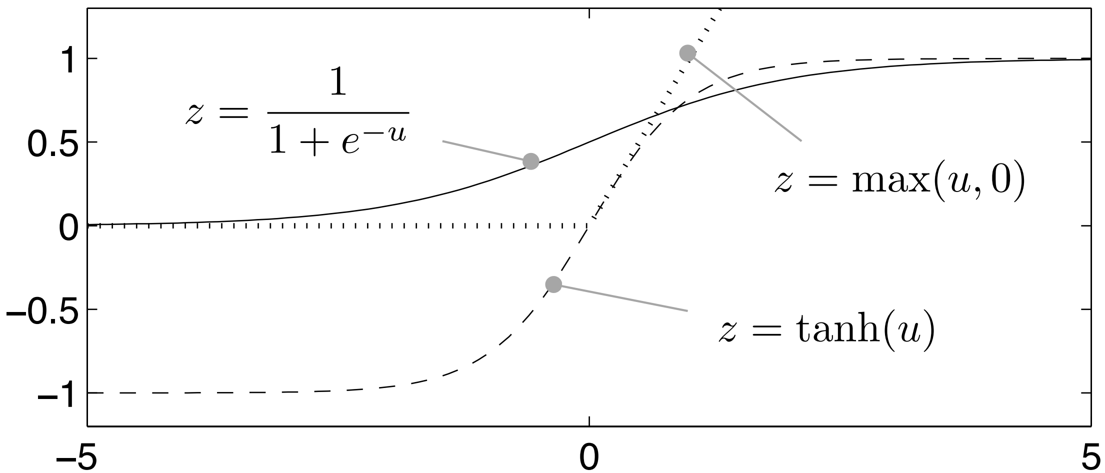
- A *unit* receives multiple input signals as their weighted sum, passes it to a nonlinear function, and outputs a signal
  - Simplified math model of a neuron

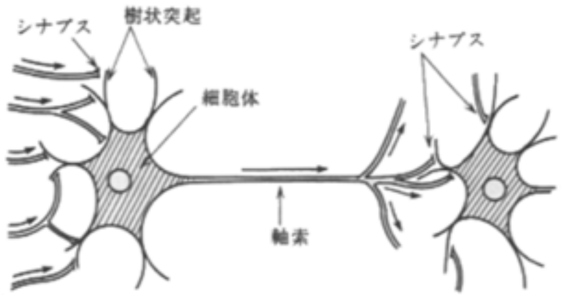


$$u = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b$$

$$z = f(u)$$

  - The func. *f* is called *activation function*
    - Various analytic funcs are used





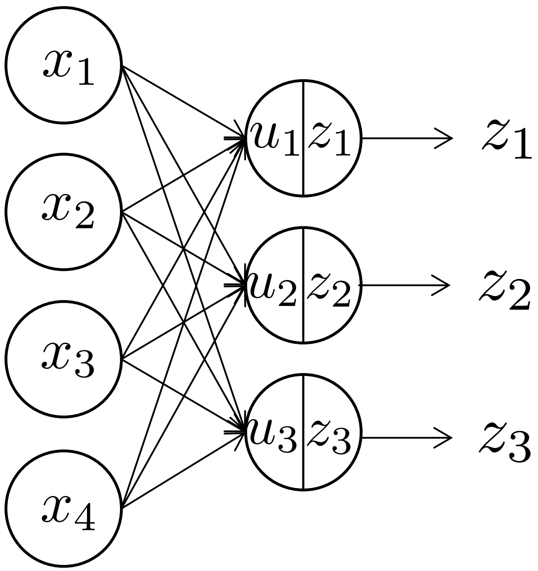
Neural networks: single layer net

- Construct a layer of multiple units
- Denoting inputs to this layer by a vector **x** and outputs by **z**, we can express the computation at this layer as

$$u_j = \sum_{i=1}^{I} w_{ji} x_i + b_j \qquad \qquad \mathbf{u} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

or

$$z_j = f(u_j) \qquad \qquad \mathbf{z} = \mathbf{f}(\mathbf{u})$$



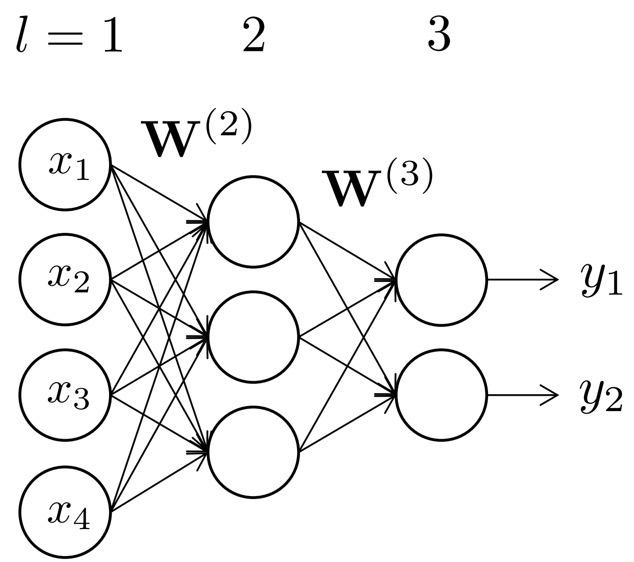
$$\mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_J \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_I \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_J \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_J \end{bmatrix},$$

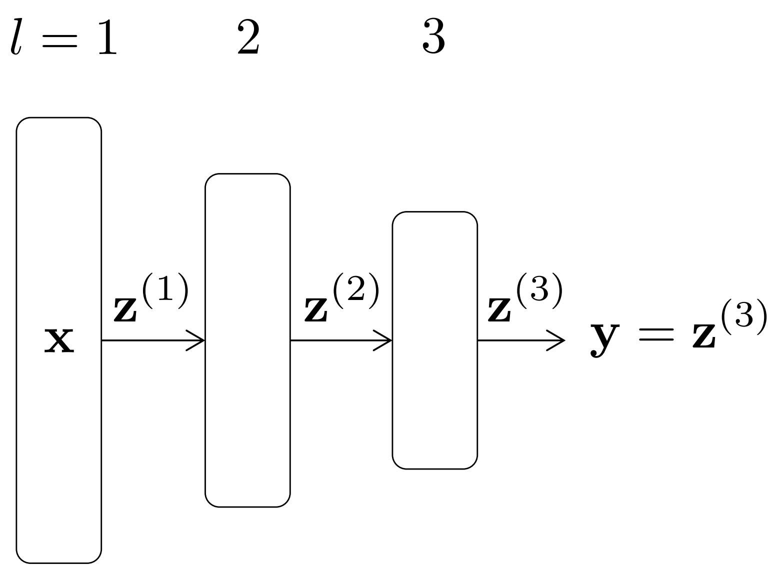
$$\mathbf{W} = \begin{bmatrix} w_{11} & \cdots & w_{1I} \\ \vdots & \ddots & \vdots \\ w_{J1} & \cdots & w_{JI} \end{bmatrix}, \quad \mathbf{f}(\mathbf{u}) = \begin{bmatrix} f(u_1) \\ \vdots \\ f(u_J) \end{bmatrix}$$

Neural networks: multi-layer net

- Stack of multiple single-layer nets = a multi-layer net also known as a feed-forward network

1st (input) layer	$\mathbf{x} \equiv \mathbf{z}^{(1)}$
Propagation from lth to $(l+1)$th layer	$\mathbf{u}^{(l+1)} = \mathbf{W}^{(l+1)}\mathbf{z}^{(l)} + \mathbf{b}^{(l+1)}$ $\mathbf{z}^{(l+1)} = \mathbf{f}(\mathbf{u}^{(l+1)})$
Lth (output) layer	$\mathbf{y} \equiv \mathbf{z}^{(L)}$





Neural networks: Output layer and loss

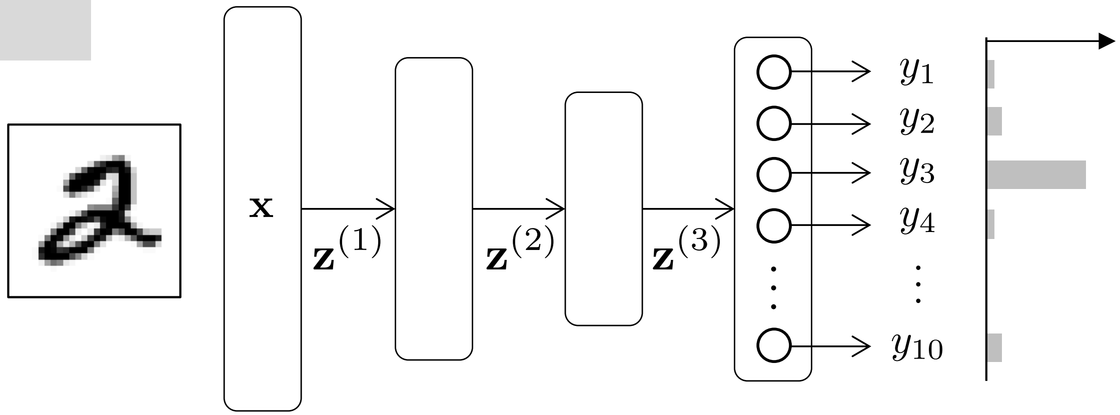
- We give the output layer the same number of units as classes and regard their output as probability (or likelihood) of the classes; k^{th} output = probability of k^{th} class
    - Sigmoid func. or softmax func. are employed for activation func. of the output layer
- Classes are encoded by a vector **d** of length K; if the class is k, then k^{th} element is 1 and all other elements are 0 (called *one-hot*/*one-of-K*)
    - You can generate one-hot vectors for 10-class MNIST data by the following procedure:

$$\mathbf{d} = [d_1, d_2, \ldots, d_K]$$

```
>> A=eye(10,10);
>> train_d=A(train_lbl+1,:);
>> test_d=A(test_lbl+1,:);
```

- We assume here that `train_lbl & test_lbl` store the label data of MNIST
- Type these commands after loading the data onto these variables; see p.70 for details

$$p(\mathcal{C}_k|\mathbf{x}) = y_k = z_k^{(L)}$$



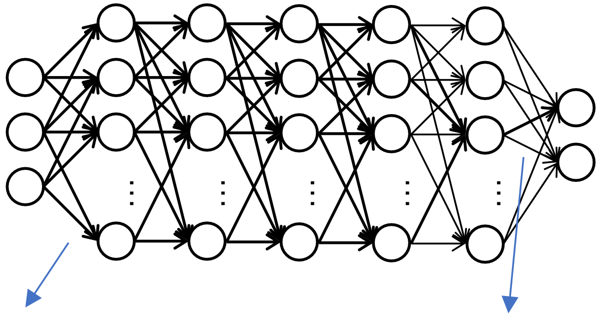
Training a feed-forward network

- We are given a set of samples; each sample is a pair of an input **x** and its target **d** (one-hot vector of the true class of the input)

$$\mathcal{S} = \{(\mathbf{x}_1, \mathbf{d}_1), \ldots, (\mathbf{x}_N, \mathbf{d}_N)\}$$

- Using this sample set, we want to *train* the neural net, where the goal is to make the output **y** for **x** as close to **d** as possible

$\mathbf{x}_n \Rightarrow$



$\Rightarrow \mathbf{y}_n \leftrightarrow \mathbf{d}_n$

Weights of layers $\mathbf{w} = \{(\mathbf{W}^{(2)}, \mathbf{b}^{(2)}), \cdots, (\mathbf{W}^{(L)}, \mathbf{b}^{(L)})\}$

Loss (difference between **y** and **d**) $E(\mathbf{w}, \mathcal{S})$

- Thus, the problem becomes a minimization of the loss:

$$\min_{\mathbf{w}} E(\mathbf{w}, \mathcal{S})$$

Software library

- In this course, we use the following library for MATLAB/Octave
  - https://github.com/rasmusbergpalm/DeepLearnToolbox
  - The author declares the software is outdated and no longer maintained; although better software such as tensorflow and torch is available for deeplearning, they are not compact for the purpose of this course;
- Download and extract a zip file from the course page, and then do as follows:

```
>> addpath('DeepLearnToolbox/NN')
>> addpath('DeepLearnToolbox/util')
```

Problem: MNIST handwritten digit recognition

- To train and test SVM, we used only a portion of 10,000 samples belonging to t10k-* files
- Here we use 60,000 samples for training NNs and 10,000 for testing them
    - To load all the data, type as follows:

```
>> fid=fopen('t10k-images-idx3-ubyte','r','b');
>> fread(fid,4,'int32')
>> test_img=fread(fid,[28*28,10000],'uint8');
>> test_img=test_img';
>> fclose(fid);

>> fid=fopen('t10k-labels-idx1-ubyte','r','b');
>> fread(fid,2,'int32')
>> test_lbl=fread(fid,10000,'uint8');
>> fclose(fid);

>> fid=fopen('train-images-idx3-ubyte','r','b');
>> fread(fid,4,'int32')
>> train_img=fread(fid,[28*28,60000],'uint8');
>> train_img=train_img';
>> fclose(fid);

>> fid=fopen('train-labels-idx1-ubyte','r','b');
>> fread(fid,2,'int32')
>> train_lbl=fread(fid,60000,'uint8');
>> fclose(fid);
```

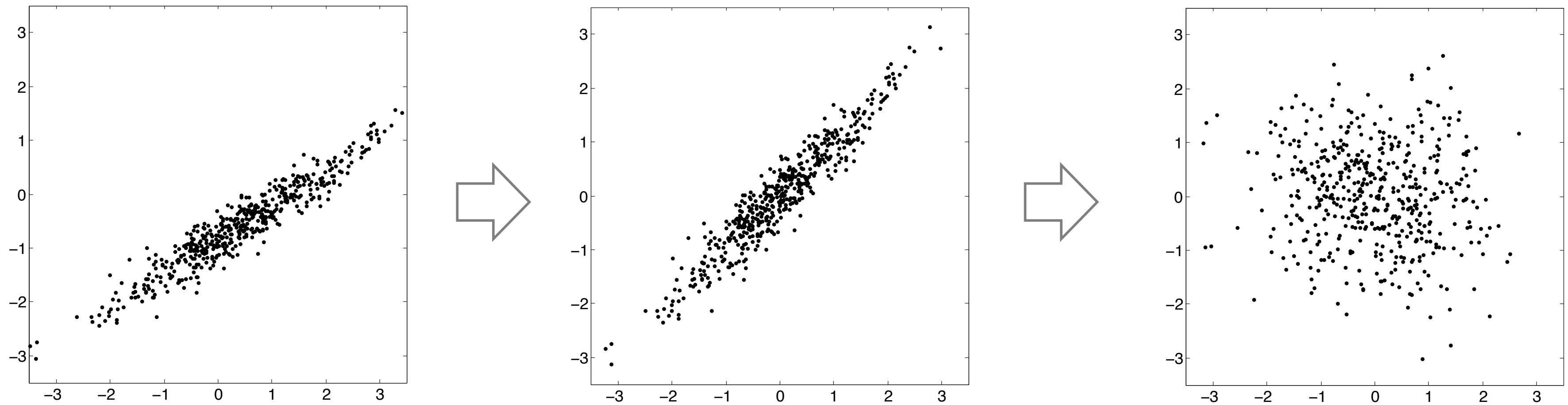
Standardization of data (1/2)

- Data ‘in the wild’ often distribute in the data space in an unfavorable manner; applying a linear transform to make them distribute uniformly usually helps training NNs and SVMs
  - A transformation making the mean 0 and the variance 1 will work well

n^{th} sample: $\mathbf{x}_n = \left[x_{n1}, x_{n2}, \ldots, x_{nI}\right]^\top$

$$x_{ni} \leftarrow \frac{x_{ni} - \bar{x}_i}{\sigma_i} \qquad \bar{x}_i \equiv \sum_{n=1}^{N} x_{ni}/N \qquad \sigma_i = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(x_{ni} - \bar{x}_i)^2}$$

mean variance



standardization
(normalization)

whitening
(we don’t consider here)

Standardization of data (2/2)

- First, compute the mean **μ** and standard deviation **σ** of training samples **x**'s

```
>> mu = mean(train_img);
>> sigma = max(std(train_img), eps);
```

- Second, subtract **μ** from each training sample and divide it by **σ**
  - Note that **μ** and **σ** are vectors of the same length as **x**'s

```
>> train_img = (train_img – mu)./sigma;
```

element-wise division

- Third, apply the same transformation with the same **μ** and **σ** to
  - Not allowed to use the mean and std. dev. of test samples; we may use only information from training samples; explain why?

```
>> test_img = (test_img – mu)./sigma;
```

Experiments

- Design a two-layer NN with 784(=28x28) elements in the input, 100 units in the intermediate layer, and 10 units in the output layer

```
>> nn = nnsetup([784 100 10]);
```

- Train the net using the training samples

```
>> opts.numepochs = 1;
>> opts.batchsize = 100;
>> [nn, L] = nntrain(nn, train_img, train_d, opts);
```

Number of times the net sees each sample during training

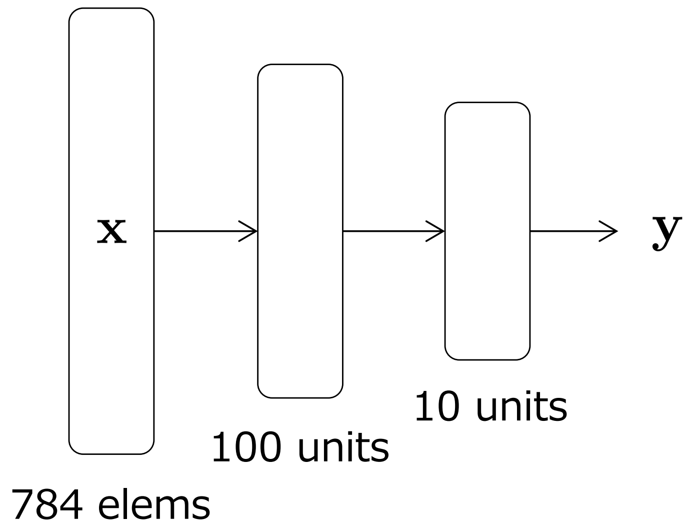
The weights are updated once for this number of samples

- Evaluate performance of the trained net using test samples

```
>> pred = nnpredict(nn, test_img);
>> pred(1:10)'
ans =
    8    3    2    1    5    2    5   10    5   10
>> test_lbl(1:10)'
ans =
    7    2    1    0    4    1    4    9    5    9
>> sum(pred-1==test_lbl)/10000*100
ans =   92.900
```

labels range from 1-10 in nnpredict

labels range from 0-9 in orig. data



Exercises 13.1

- You can run `nntrain` repeatedly; it will update the net incrementally using the same training samples
  - To perform this, just type:

```
>> [nn, L] = nntrain(nn, train_img, train_d, opts);
```

  - If you want to reset the training, initialize the net as follows

```
>> nn = nnsetup([784 100 10]);
```

1. Repeat training for, say, 10 steps, from initialization and evaluate performance of the net at each step; plot ‘training counts’-vs-’accuracy’
2. Design a three-layer NN, *for instance*, having two intermediate layers with 30 units each, and train it; and evaluate the difference in performance from the earlier two-layer net