Machine Learning, Spring 2019
EN 2105
MWF 9-9:50am

Chao Lan
EN 4084A
clan@uwyo.edu

Artificial Neural Network

An artificial neural network is a network of neurons. A typical neuron is a graphical representation of a nonlinearized linear function; its output has the form

$$\sigma(w^T x), \tag{1}$$

where $\sigma : \mathcal{R} \to \mathcal{R}$ is a nonlinear activation function, each w_j is a weight (to be learned), and $x \in \mathbb{R}^p$ is input of the neuron; each $x_{\cdot j}$ is either a data feature or output of another neuron. There are many choices of activation function[1]; a common one is logistic sigmoid function, i.e.,

$$\sigma(w^T x) = \frac{1}{1 + \exp(-w^T x)}. \tag{2}$$

A network has many connected layers, each made of multiple neurons.

The input layer contains p neurons each corresponding to one of the p data features; the j_{th} neuron directly outputs the j_{th} feature.

The output layer contains K neurons each corresponding to one of the K classes; the k_{th} neuron outputs the probability that input instance belongs to class k (or, outputs 1 if the instance is classified to class k and outputs 0 otherwise). For example, if there are three classes of cat, dog and fish, then there should be three output neurons y_1, y_2, y_3, where $y_1/y_2/y_3$ is the probability that x is a cat/dog/fish. For y_k to be interpreted as probability, the softmax function is often applied on $\{y_k\}$ and the neuron output is refined as

$$y_k \leftarrow \text{softmax}(y_k) = \frac{\exp(y_k)}{\sum_{j=1}^{3} \exp(y_j)}. \tag{3}$$

[*Discussion*] How does the softmax function guarantee a probability interpretation of y_k?

There can be many hidden layers, and each can consist of an arbitrary number of neurons; the inputs of a neuron at layer h are (typically) the weighted outputs of neurons at layer $h - 1$.

The number of hidden layers and the number of neurons per layer are hyper-parameters.

[*Discussion*] How would the above two numbers affect model complexity?

[*Discussion*] What if all activation functions are linear?

[1] `https://en.wikipedia.org/wiki/Activation_function`

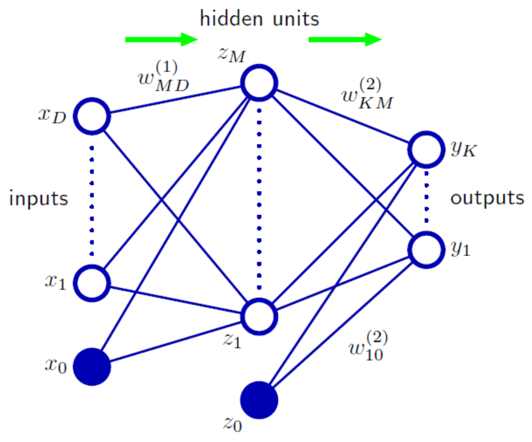


Figure 1: Neural Network Architecture [PRML, Figure 5.1]

There are many network architectures. A common one is feed-forward neural network (Fig 1).

We commonly learn a neural network (its weights) using back-propagation, which is essentially stochastic gradient descent. Let f_k be the function of the neural network at the k_{th} output neuron. Its prediction loss over a sample of n instances is

$$J_n(f_k) = \sum_{i=1}^{n} (f_k(x_i) - y_i)^2, \tag{4}$$

and the total prediction loss over all output neurons is

$$J_n = \sum_{k=1}^{K} \sum_{i=1}^{n} (f_k(x_i) - y_i)^2. \tag{5}$$

Back-propagation find an optimal set of weights that minimizes J_n, using the stochastic gradient descent method. Let $w_j^{(m)}$ be the j_{th} weight at layer m. The gradient descent updates it by

$$w_j^{(m)} = w_j^{(m)} - \eta \cdot \frac{\partial}{\partial w_j^{(m)}} J_n = w_j^{(m)} - \eta \cdot \left(\sum_k \sum_{i=1}^{n} (f_k(x_i) - y_i)^2 \right), \tag{6}$$

where η is the learning rate. The stochastic gradient descent approximates this update rule by

$$w_j^{(m)} = w_j^{(m)} - \eta \cdot \left(\sum_k \sum_{i=1}^{n_t} (f_k(x_i) - y_i)^2 \right), \tag{7}$$

where at iteration t (or, epoch t) only a subset of n_t instances are used to compute the gradient.

Algorithmically, the weights are optimized as follows: at epoch t, input a batch of n_t instances to the network and compute J_{n_t} by (5); then update all weights by (7); repeat until convergence.

A neural network is likely to overfit due to its big model complexity. There are several ways to avoid it. One is weight decay, which adds a regularization term to the objective function

$$J_n = \sum_{k=1}^{K} \sum_{i=1}^{n} (f_k(x_i) - y_i)^2 + \lambda \sum_{j,m} \left(w_j^{(m)} \right)^2. \tag{8}$$

Another approach is early stop, which stops updating the network before it converges. A latest approach is dropout, which only updates a sub-network at every epoch.

[*Exercise*] Compute gradients at w_{11} and β_1 in Fig 2. [2]

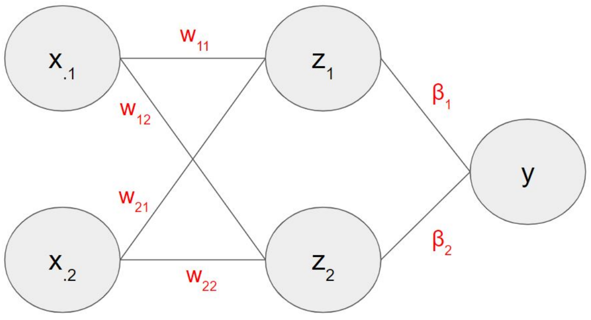


Figure 2: A Toy Feed-Forward Neural Network

Deep neural networks are neural networks with significant amounts of hidden layers (e.g. 100) and more sophisticated architectures. We will briefly introduce two: CNN and RNN.

Convolutional Neural Network (CNN) is good at processing image data. It has multiple meta-layers each consisting of a convolution layer and a maxpooling layer (fig 3). The former runs convolution on an (processed) image to extract its local features (that are good for classification) (fig 4). The latter down samples an image (fig 5). For pixels at the edge of an image, we can apply zero-padding (fig 6). Example outputs of CNN are in fig 7, where first layers learn local features and last layers learn global features. CNN can learn useful representations from unstructured features (e.g. image pixels). Weights can be learned by back-propagation.

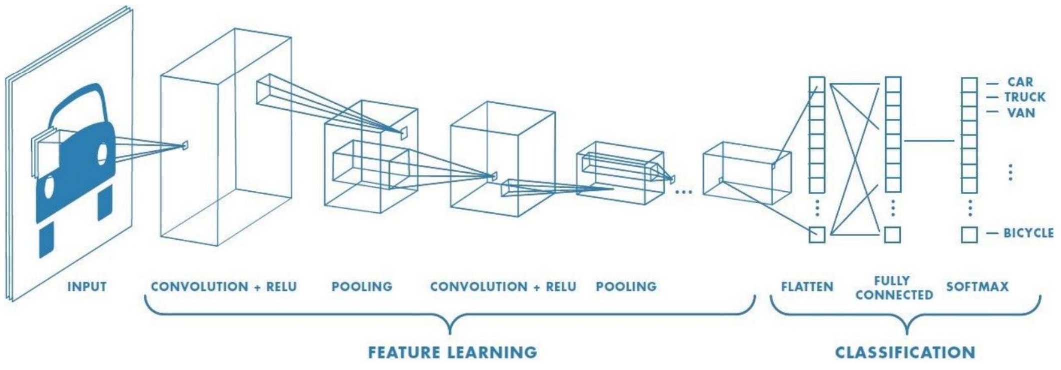
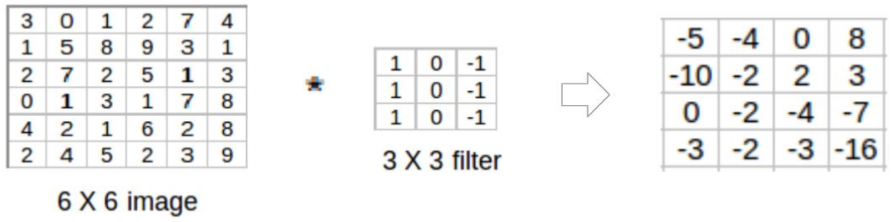


Figure 3: Convolutional Neural Network



$$\begin{array}{|c|c|c|c|c|c|}\hline 3&0&1&2&7&4\\ \hline 1&5&8&9&3&1\\ \hline 2&7&2&5&1&3\\ \hline 0&1&3&1&7&8\\ \hline 4&2&1&6&2&8\\ \hline 2&4&5&2&3&9\\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 1&0&-1\\ \hline 1&0&-1\\ \hline 1&0&-1\\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|}\hline -5&-4&0&8\\ \hline -10&-2&2&3\\ \hline 0&-2&-4&-7\\ \hline -3&-2&-3&-16\\ \hline \end{array}$$

Figure 4: Convolution

[2]For simplicity, assume all non-input neurons share the same activation function.

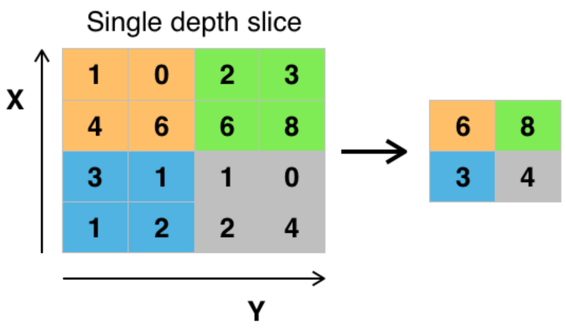
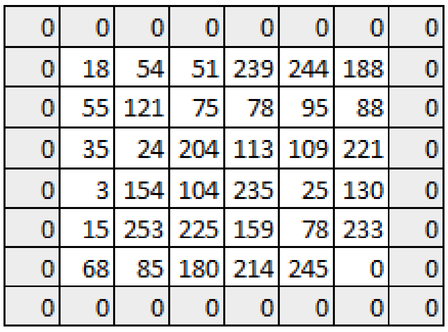


Figure 5: Max-Pooling



0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

Figure 6: Zero-Padding

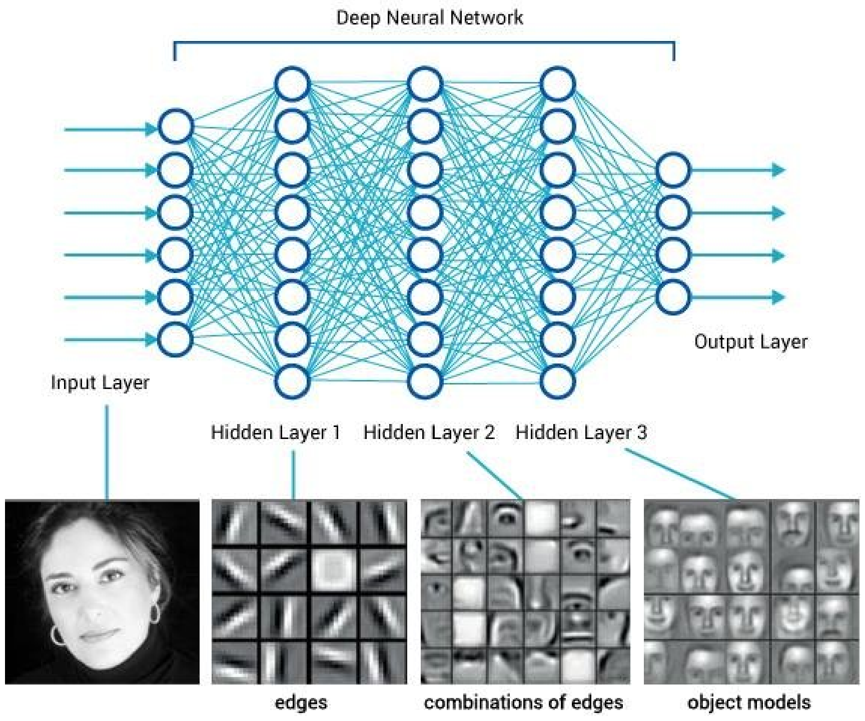


Figure 7: Layer Outputs of CNN

Recurrent Neural Network (RNN) is good at time-series/sequential prediction. It is a (infinite) sequence of cells (fig 8). The t_{th} cell predicts data at time t. It takes two inputs: a feature vector $x_t \in \mathbb{R}^p$ at time t and a hidden state vector $s_{t-1} \in \mathbb{R}^h$ at time $t-1$. It gives two outputs: a predicted label $o_t \in \mathbb{R}$ at time t and a hidden state vector $s_t \in \mathbb{R}^h$ at time t (which will be input to the next cell). The hidden states s_t's encode the historical information of the sequence.

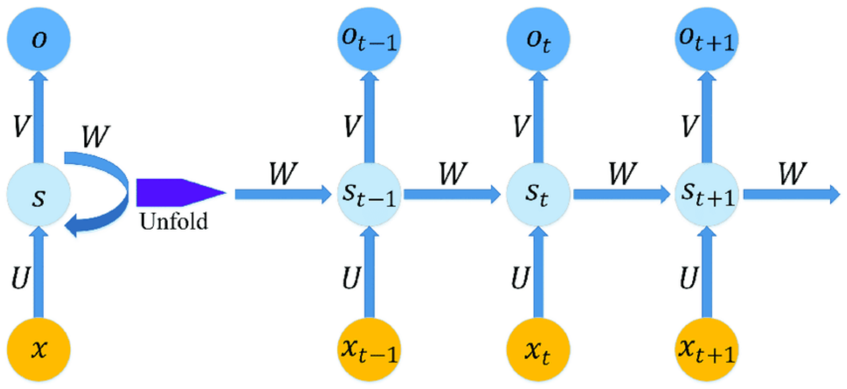


Figure 8: Recurrent Neural Network

[*Discussion*] What are the applications of RNN?

A typical mathematical model of cell t is as follows: the output hidden state is

$$s_t = \sigma(W_s s_{t-1} + W_x x_t + b), \tag{9}$$

where $W_s \in \mathbb{R}^{h\times h}$ is a state transformation matrix, $W_x \in \mathbb{R}^{h\times p}$ is a feature transformation matrix, $b \in \mathbb{R}^h$ is a bias vector and σ is an activation function e.g., the tanh function[3]

$$\sigma(z) = \frac{\exp^z - \exp^{-z}}{\exp^z + \exp^{-z}}. \tag{10}$$

The predicted label is

$$y_t = \text{softmax}(W_y s_t + b_y), \tag{11}$$

where $W_y \in \mathbb{R}^{K\times h}$ is a label transformation matrix (for K classes) and $b_y \in \mathbb{R}^K$ is a bias vector.

The parameters W_s, W_x, b, W_y, b_y are (typically) assumed shared by all cells and can be learned from data by back-propagation.

[3]Tanh function is equivalent to a scaled logistic sigmod function that falls in [-1, +1].