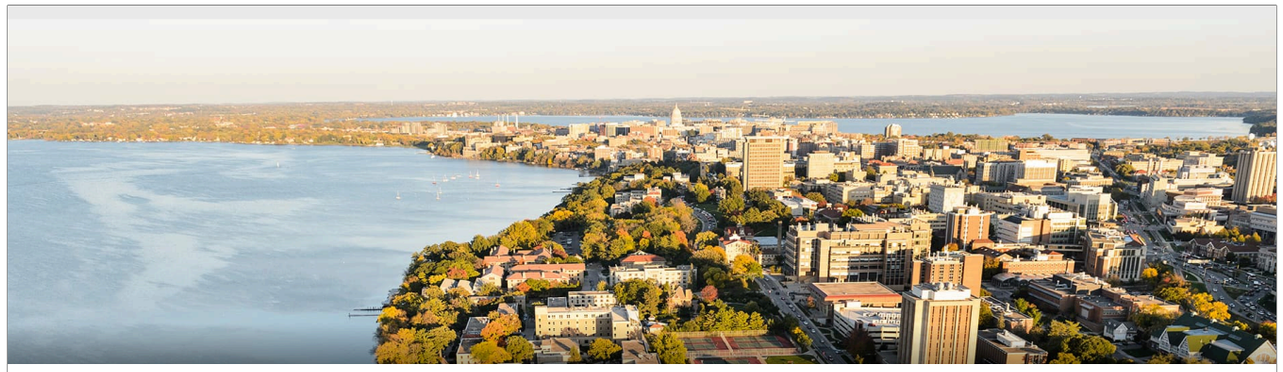


CS540 Introduction to Artificial Intelligence

(Deep) Neural Networks Summary

Yingyu Liang
University of Wisconsin-Madison

Nov 9, 2021

Slides created by Sharon Li [modified by Yingyu Liang]

How to classify

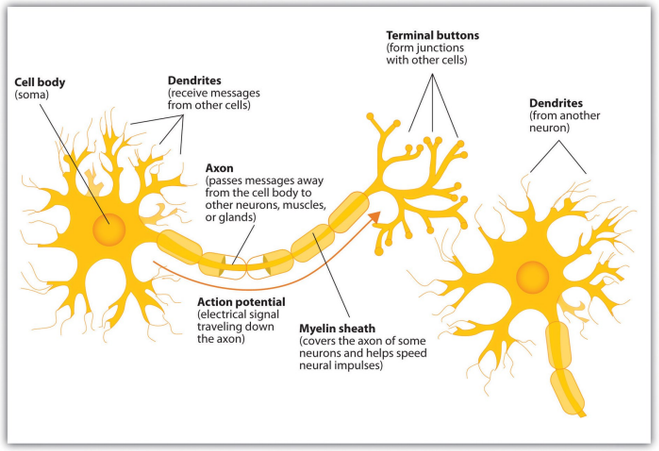
Cats vs. dogs?

- Single-layer Perceptron
- Multi-layer Perceptron
- Training of neural networks
- Convolutional neural networks

Inspiration from neuroscience

- Inspirations from human brains
- Networks of simple and homogenous units (a.k.a **neuron**)

(wikipedia)



Artificial neural networks are inspired by human neural networks. Though we don’t know much about human brains, we do know that they are networks of simple and homogenous units called neurons. Human neurons can be divided into a few types, and neurons of the same type are very similar to each other; each neuron performs simple operations. These facts lead to the design of artificial neural networks.

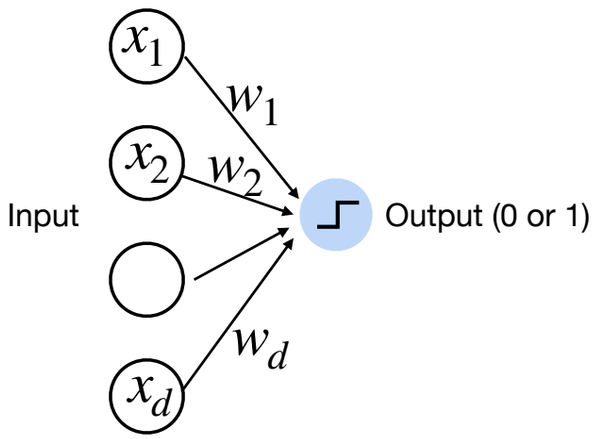
Perceptron

- Given input $\mathbf{x}$, weight $\mathbf{w}$ and bias b, perceptron outputs:

$$o = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right) \qquad \sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Activation function

Cats vs. dogs?



The standard perceptron is simply linear transformation followed by the activation function (the step function). The output 0 or 1 can be viewed as classes.

Training the Perceptron

Perceptron Algorithm

Initialize $\vec{w} = \vec{0}$ // Initialize $\vec{w}$. $\vec{w} = \vec{0}$ misclassifies everything.
while TRUE **do** // Keep looping
  $m = 0$ // Count the number of misclassifications, m
  for $(x_i, y_i) \in D$ **do** // Loop over each (data, label) pair in the dataset, D
    if $o_i \neq y_i$ **then** // If the pair $(\vec{x_i}, y_i)$ is misclassified
      $\vec{w} \leftarrow \vec{w} + x_i$ if $y_i = 1$, $\vec{w} \leftarrow \vec{w} - x_i$ if $y_i = 0$
      $m \leftarrow m + 1$ // Counter the number of misclassification
    end if
  end for
  if $m = 0$ **then** // If the most recent $\vec{w}$ gave 0 misclassifications
    break // Break out of the while-loop
  end if
end while // Otherwise, keep looping!

For simplicity, the weight vector and input vector are extended vectors (including the bias or the constant 1).

This perceptron was proposed in the early days of AI, which has inspired many other methods.

The method follows the intuition of correcting mistakes.
For simplicity of presentation, usually we concatenate the weight vector with the bias to get an extended weight vector \vec{w}. We also concatenate the input point x with a constant 1. In this weight, the linear transformation before the step activation function is simply the inner product between the extended weight vector and the extended input point.

The training algorithm goes over the training dataset; each pass is called an epoch. In an epoch, it goes over the data points one by one. For the current data point, it uses the current perceptron to predict. If no mistake, just do nothing; if there is a mistake then update the weight vector. In the case when the true class label is 1 but we predict class 0, then it means the linear transformation is too small, we should increase that, and we do so by increasing the weight vector by adding the extended input point. In the other case when the true class label is 0 but we predict class 1, then it means the linear transformation is too large, we should decrease that, and we do so by decreasing the weight vector by subtracting the extended input point.

If in an epoch we make no mistakes, it means we cannot update the weight vector anymore. The method just stop.

The method can succeed when there is indeed a ground truth linear hyperplane that can separates the two classes in the training data. If the two classes in the training data are not linearly separable, then the method will loop for ever since there is always some mistake.

Example: Training the Perceptron

- Suppose we begin with:

  $\mathbf{w} = (1,2), b = -1$

- Extended vectors:

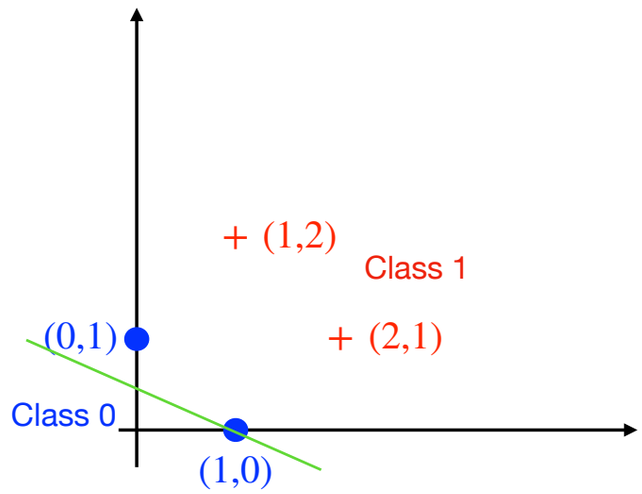
  $\vec{w} = (1,2,-1)$

  $\vec{x_1} = (0,1,1)$

  $\vec{x_2} = (1,0,1)$

  $\vec{x_3} = (2,1,1)$

  $\vec{x_4} = (1,2,1)$



Here we show an example.

For simplicity consider two training data points in 2dim. Also assume we've gotten a current perceptron with weight (1,2) and bias -1. The decision boundary is plotted as the green line in the figure. (Note that the perceptron training method usually begins with weight/bias 0, but here for demonstration we begin with a different one.)

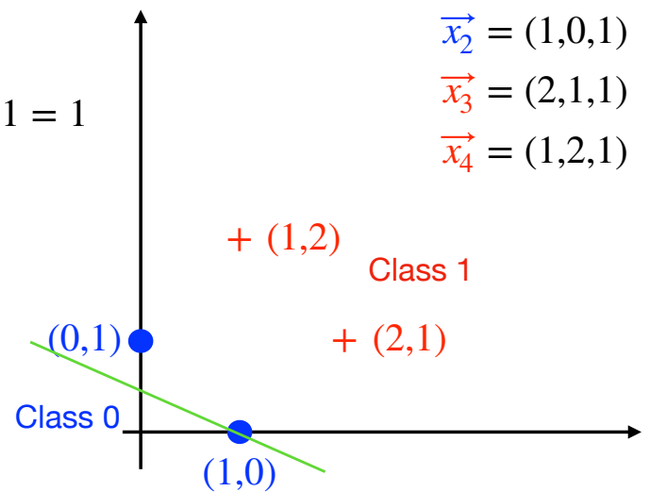
First construct the extended weight and input points: concatenate the weight + bias; concatenate the input point+constant 1.

Example: Training the Perceptron

$\vec{w} = (1,2,-1)$
$\vec{x_1} = (0,1,1)$
$\vec{x_2} = (1,0,1)$
$\vec{x_3} = (2,1,1)$
$\vec{x_4} = (1,2,1)$

- First Epoch:

$\vec{x_1} : \langle \vec{w}, \vec{x_1} \rangle = 1 \times 0 + 2 \times 1 + (-1) \times 1 = 1$



Let’s consider the first epoch. We go over the points one by one, beginning with x1.

For x1, the linear transformation is just the inner product of the extended weight vector and the extended input point. It is 1, and thus the current perceptron predicts class 1, which is wrong. So we need to update.

Example: Training the Perceptron

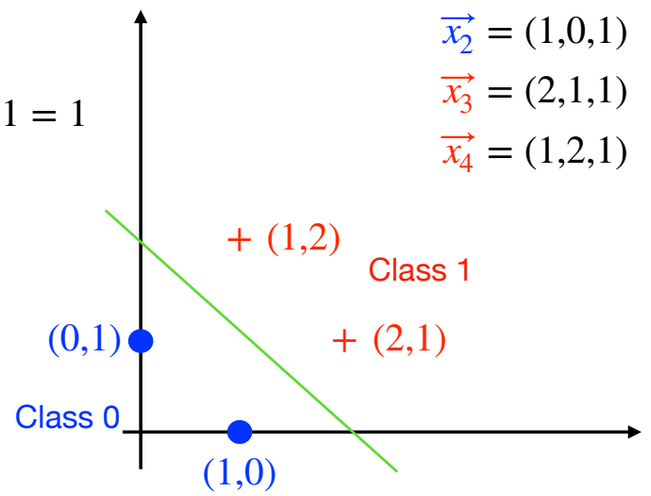
$\vec{w} = (1,1,-2)$
$\vec{x_1} = (0,1,1)$
$\vec{x_2} = (1,0,1)$
$\vec{x_3} = (2,1,1)$
$\vec{x_4} = (1,2,1)$

- First Epoch:

$\vec{x_1}$: $\langle \vec{w}, \vec{x_1} \rangle = 1 \times 0 + 2 \times 1 + (-1) \times 1 = 1$

wrong prediction

update $\vec{w} \leftarrow \vec{w} - \vec{x_1} = (1,1,-2)$



The true class is 0 and the prediction is 1. So the linear transformation is too large, we need to decrease it. We thus decrease the weight vector by subtracting the input point. (Note that the bias also gets updated as it's part of the extended weight vector. The extended notation simplifies the presentation of the algorithm.) The new perceptron's decision boundary moves as shown in the figure.

Example: Training the Perceptron

$\vec{w} = (1,1,-2)$
$\vec{x_1} = (0,1,1)$
$\vec{x_2} = (1,0,1)$
$\vec{x_3} = (2,1,1)$
$\vec{x_4} = (1,2,1)$

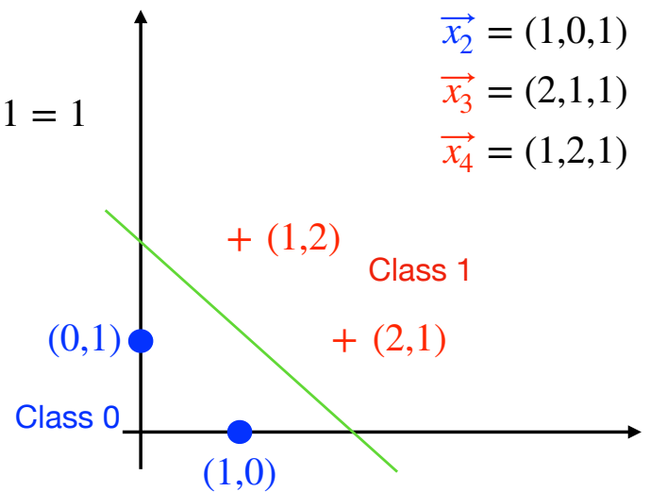
- First Epoch:

$\vec{x_1}: \langle \vec{w}, \vec{x_1} \rangle = 1 \times 0 + 2 \times 1 + (-1) \times 1 = 1$

wrong prediction

update $\vec{w} \leftarrow \vec{w} - \vec{x_1} = (1,1,-2)$

$\vec{x_2}: \langle \vec{w}, \vec{x_2} \rangle = -1$



Now let’s go to the next data point.
For x2: the linear transformation is -1. Note that we need to use the current perceptron (after the update on x1).

Example: Training the Perceptron

$\overrightarrow{w} = (1,1,-2)$
$\overrightarrow{x_1} = (0,1,1)$
$\overrightarrow{x_2} = (1,0,1)$
$\overrightarrow{x_3} = (2,1,1)$
$\overrightarrow{x_4} = (1,2,1)$

- First Epoch:

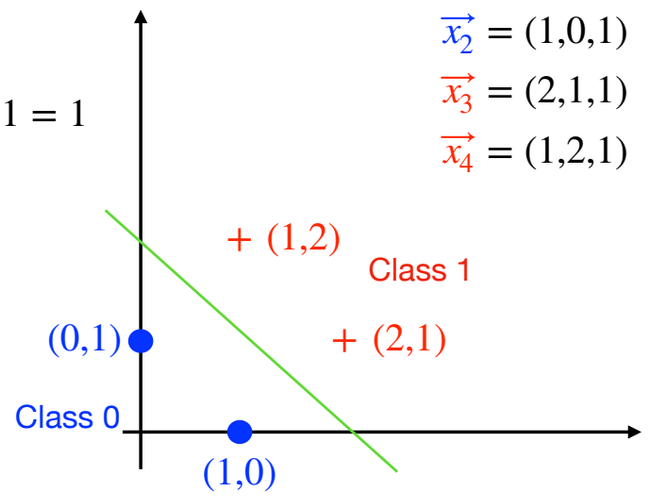
$\overrightarrow{x_1}$: $\langle \overrightarrow{w}, \overrightarrow{x_1} \rangle = 1 \times 0 + 2 \times 1 + (-1) \times 1 = 1$

wrong prediction

update $\overrightarrow{w} \leftarrow \overrightarrow{w} - \overrightarrow{x_1} = (1,1,-2)$

$\overrightarrow{x_2}$: $\langle \overrightarrow{w}, \overrightarrow{x_2} \rangle = -1$

correct prediction, no update



So predict class 0 which is right. Do nothing.

Example: Training the Perceptron

$\vec{w} = (1,1,-2)$
$\vec{x_1} = (0,1,1)$
$\vec{x_2} = (1,0,1)$
$\vec{x_3} = (2,1,1)$
$\vec{x_4} = (1,2,1)$

- First Epoch:

$\vec{x_1}$: $\langle \vec{w}, \vec{x_1} \rangle = 1 \times 0 + 2 \times 1 + (-1) \times 1 = 1$

wrong prediction

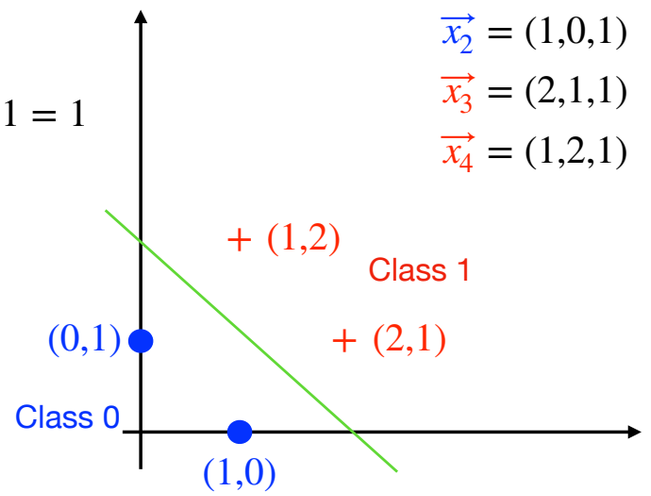
update $\vec{w} \leftarrow \vec{w} - \vec{x_1} = (1,1,-2)$

$\vec{x_2}$: $\langle \vec{w}, \vec{x_2} \rangle = -1$

correct prediction, no update

$\vec{x_3}$: $\langle \vec{w}, \vec{x_3} \rangle = 1$

correct prediction, no update



For the next data point x3: the linear transformation is 1. Correct prediction so do nothing.

Example: Training the Perceptron

$\vec{w} = (1,1,-2)$
$\vec{x_1} = (0,1,1)$
$\vec{x_2} = (1,0,1)$
$\vec{x_3} = (2,1,1)$
$\vec{x_4} = (1,2,1)$

- First Epoch:

$\vec{x_1}$: $\langle \vec{w}, \vec{x_1} \rangle = 1 \times 0 + 2 \times 1 + (-1) \times 1 = 1$

wrong prediction

update $\vec{w} \leftarrow \vec{w} - \vec{x_1} = (1,1,-2)$

$\vec{x_2}$: $\langle \vec{w}, \vec{x_2} \rangle = -1$

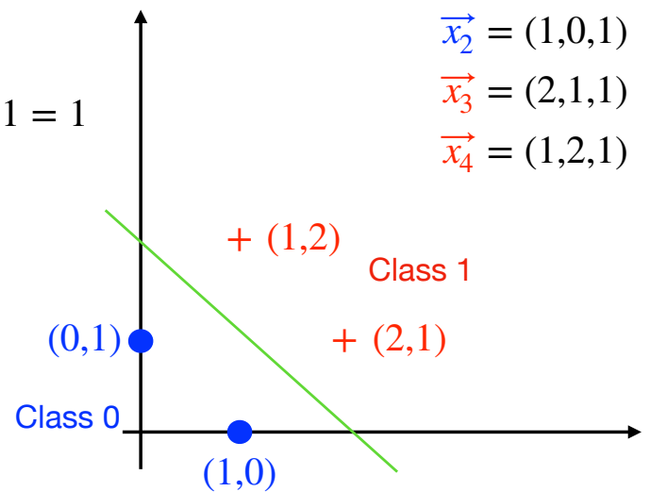
correct prediction, no update

$\vec{x_3}$: $\langle \vec{w}, \vec{x_3} \rangle = 1$

correct prediction, no update

$\vec{x_4}$: $\langle \vec{w}, \vec{x_4} \rangle = 1$

correct prediction, no update



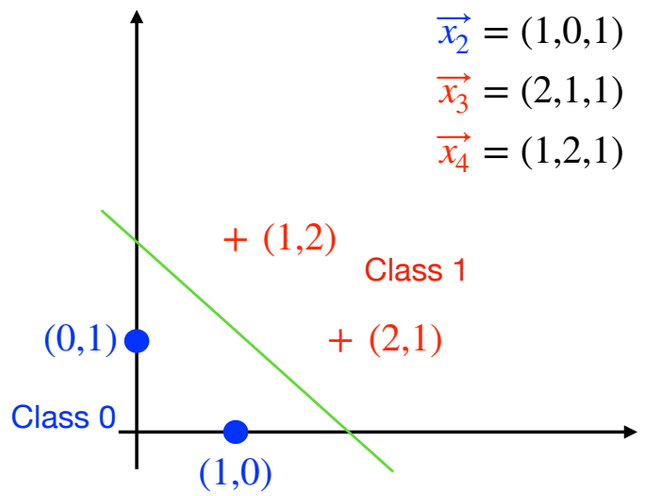
For the next data point x4: the linear transformation is 1. Correct prediction so do nothing.

Example: Training the Perceptron

$\overrightarrow{w} = (1,1,-2)$
$\overrightarrow{x_1} = (0,1,1)$
$\overrightarrow{x_2} = (1,0,1)$
$\overrightarrow{x_3} = (2,1,1)$
$\overrightarrow{x_4} = (1,2,1)$

- Second Epoch:

$\overrightarrow{x_1}: \langle \overrightarrow{w}, \overrightarrow{x_1} \rangle = -1,$ correct
$\overrightarrow{x_2}: \langle \overrightarrow{w}, \overrightarrow{x_2} \rangle = -1,$ correct
$\overrightarrow{x_3}: \langle \overrightarrow{w}, \overrightarrow{x_3} \rangle = 1,$ correct
$\overrightarrow{x_4}: \langle \overrightarrow{w}, \overrightarrow{x_4} \rangle = 1,$ correct



Now go to the second epoch. We again go over data points one by one, and can see that all predictions are correct (no updates).

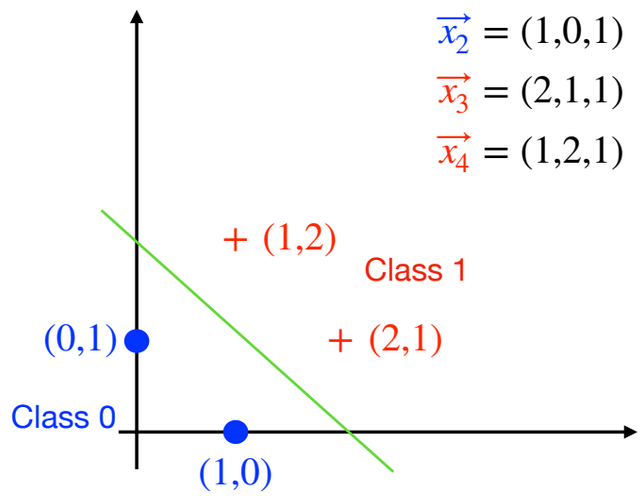
Example: Training the Perceptron

$\vec{w} = (1,1,-2)$
$\vec{x_1} = (0,1,1)$
$\vec{x_2} = (1,0,1)$
$\vec{x_3} = (2,1,1)$
$\vec{x_4} = (1,2,1)$

- Second Epoch:

$\vec{x_1}: \langle \vec{w}, \vec{x_1} \rangle = -1,$ correct
$\vec{x_2}: \langle \vec{w}, \vec{x_2} \rangle = -1,$ correct
$\vec{x_3}: \langle \vec{w}, \vec{x_3} \rangle = 1,$ correct
$\vec{x_4}: \langle \vec{w}, \vec{x_4} \rangle = 1,$ correct

- Success!



Then we can stop can claim success.

Limitation: XOR Problem (Minsky & Papert, 1969)

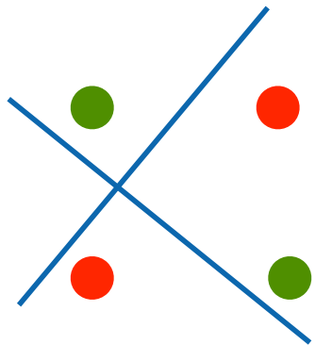
The perceptron cannot learn an XOR function
(neurons can only generate linear separators)

$x_1 = 1, x_2 = 1, y = 0$

$x_1 = 1, x_2 = 0, y = 1$

$x_1 = 0, x_2 = 1, y = 1$

$x_1 = 0, x_2 = 0, y = 0$



On the other hand, perceptron can represent AND OR NOT, and their composition can represent all logic functions

Using one single perceptron has limitation. Its decision boundary is linear. So if the learning task requires a nonlinear decision boundary, then perceptron cannot achieve good performance. There are quite a lot of tasks that require nonlinear decision boundaries. The famous one is the XOR problem.

Minsky & Papert, 1969 pointed out the limitation of a single perceptron using the XOR problem. They used this to motivate using networks of perceptrons instead of a single one. The networks can be powerful: one perceptron can represent any of the basic logic function AND OR NOT, and we know that composition of these basic logic function can represent all logic functions, so composition of perceptron can representation all logic functions. However due to resource constraints (data, computing power, network design etc), the training of multiple layer perceptrons was not successful at that time. This leads to the first AI winter.

Quiz break

Which one of the following is NOT true about perceptron?

A. Perceptron only works if the data is linearly separable.
B. Perceptron can learn AND function
C. Perceptron can learn XOR function
D. Perceptron is a supervised learning algorithm

Quiz break

Which one of the following is NOT true about perceptron?

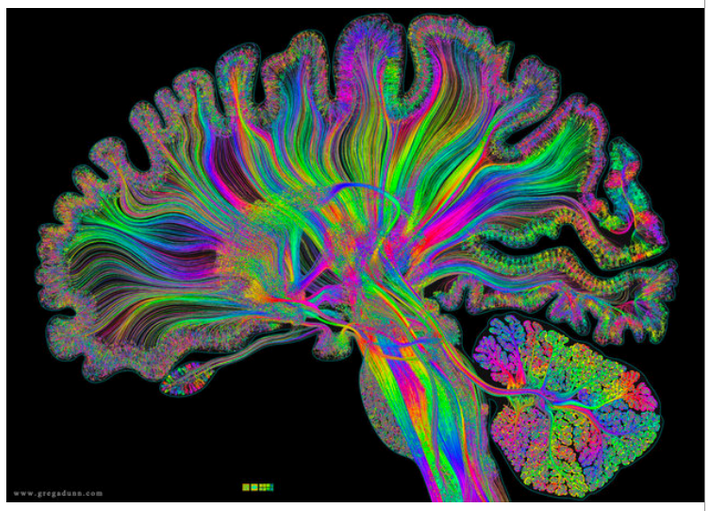
A. Perceptron only works if the data is linearly separable.

B. Perceptron can learn AND function

C. Perceptron can learn XOR function

D. Perceptron is a supervised learning algorithm

Multilayer Perceptron

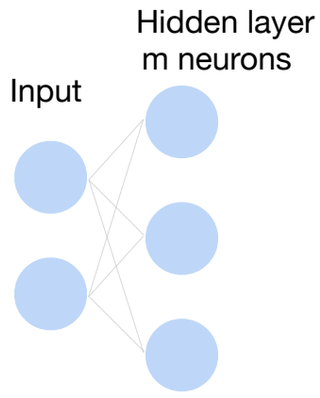


Single Hidden Layer

- Input $\mathbf{x} \in \mathbb{R}^d$
- Hidden $\mathbf{W} \in \mathbb{R}^{m \times d}, \mathbf{b} \in \mathbb{R}^m$
- Intermediate output

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

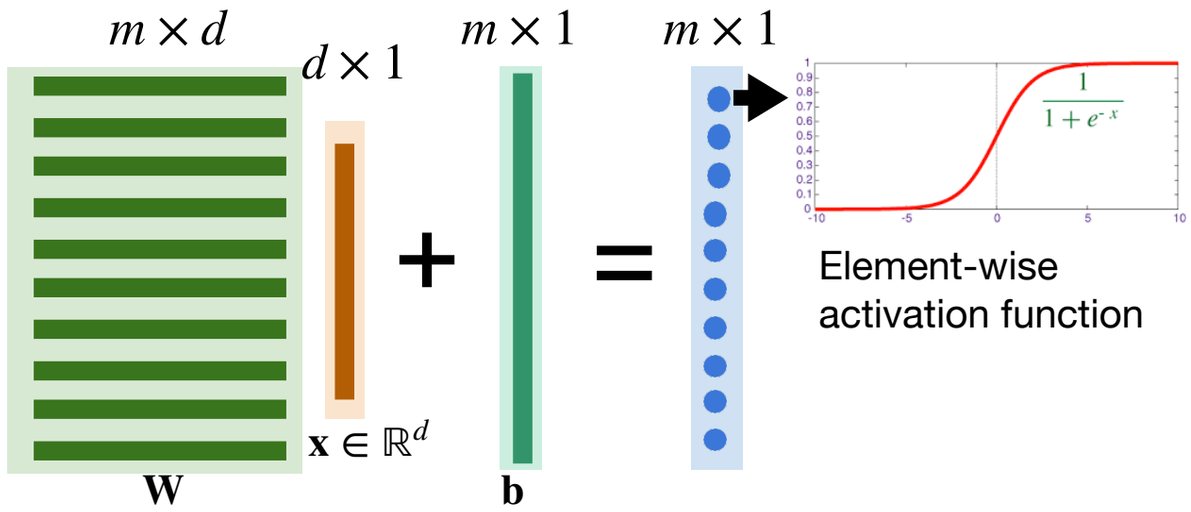
σ is an element-wise activation function



Each hidden neuron is a perceptron (but can be with a different activation function than the step function). Viewing the outputs of the hidden neurons in the layer as a vector, then the layer just first does a linear transformation on the input vector and then applies an activation function on each element of the vector obtained by the linear transformation. Note that here, we apply the activation function \sigma on the vector Wx+b; it simply means applying \sigma on each element of the vector.

Neural networks with one hidden layer

Key elements: linear operations + nonlinear activations

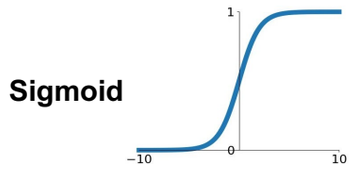


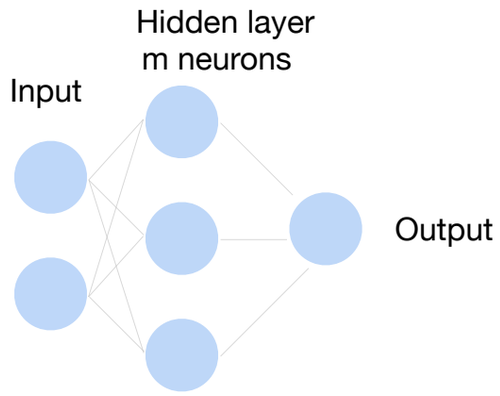
This is an illustration of the operations in a layer. Note the dimensions.

Single Hidden Layer

- Output $f = \mathbf{w}_2^\top \mathbf{h} + b_2$
- Normalize the output into probability using sigmoid

$$p(y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-f}}$$





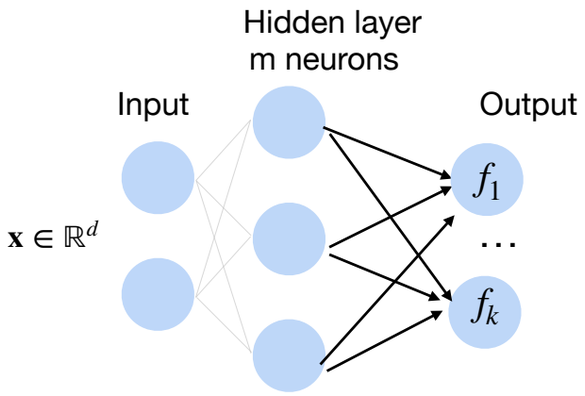
The output layer applies a linear transformation on the result vector of the previous layer, to get a scalar. (That is, view the output vector h of the hidden layer as the input for the output layer.)

Then apply the sigmoid function that squash the result of the linear transformation to a number in (0,1). This is typically used as the conditional probability of y=1 conditioned on the input x, in binary classification on classes {0,1}.

For regression, we can simply use f as the final output of the network.

Multi-class classification

Turns outputs f into k probabilities (sum up to 1 across k classes)



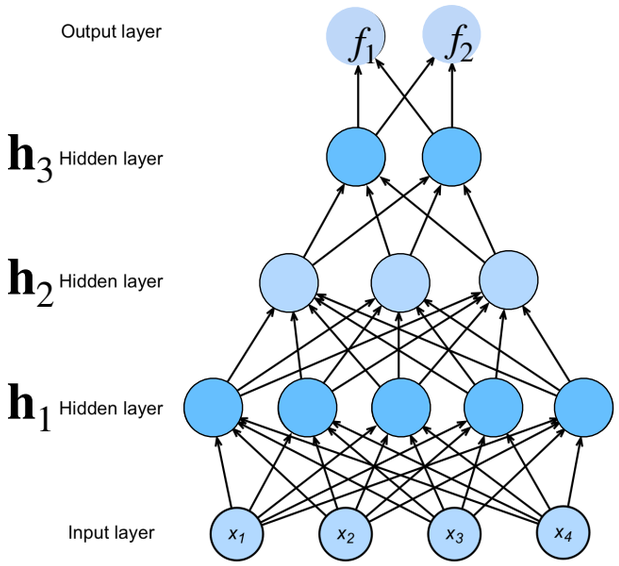
$$p(y \mid \mathbf{x}) = \text{softmax}(\mathbf{f})$$
$$= \frac{\exp f_y(x)}{\sum_i^k \exp f_i(x)}$$

For multiple-class classification, the output layer first applies a linear transformation to get a k dim vector f=(f_1,…,f_k). Then apply the softmax activation to turn f into a probabilistic vector (a distribution) over the k classes. This is typically used to modeled the conditional probabilities of y over the k classes conditioned on the input x.

Note that

1. after softmax, the entries are nonnegative and have sum 1, ie, the vector after softmax is a probabilistic vector.
2. Larger the f_i, larger the corresponding entry after softmax.
3. Softmax can be viewed as an activation function but it’s not element-wise as each output entry depends on all entries in f.

Deep neural networks (DNNs)

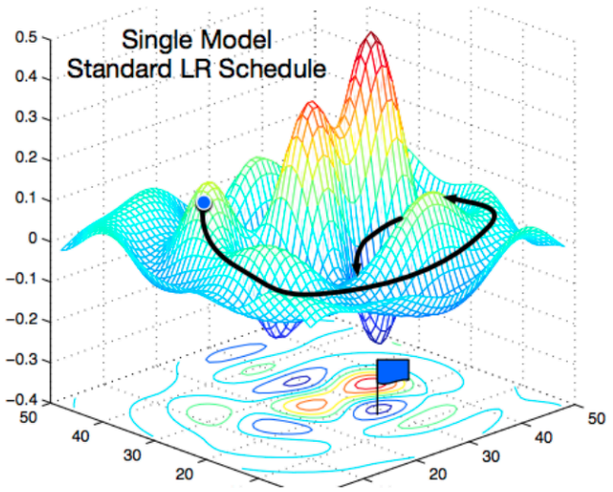


$$\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$
$$\mathbf{f} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4$$
$$\mathbf{y} = \text{softmax}(\mathbf{f})$$

NNs are composition of nonlinear functions

A standard deep network is simply stacking multiple layers of computation; each layer is linear transformation+activation (typically nonlinear). For regression, we can simply use the linear transformation result f in the output layer as the final output of the network. For classification, we apply softmax on f to get the probabilities over k classes (or apply sigmoid for binary classification).

Training Neural Networks



[Gao and Li et al., 2018]

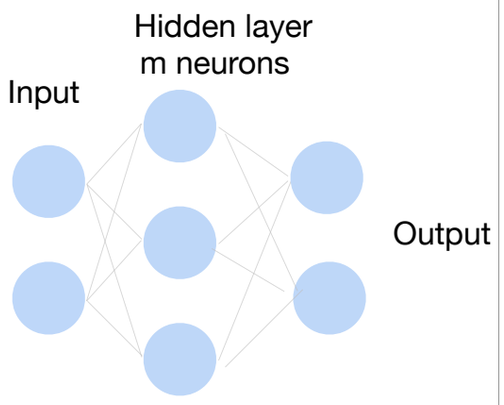
Cross-Entropy Loss

Loss: $$\frac{1}{|D|}\sum_{(\mathbf{x},y)\in D}\ell(\mathbf{x},y)$$

Per sample loss (cross-entropy or softmax loss):

$$\ell(\mathbf{x},y)=\sum_{j=1}^{K}-Y_j\log p_j=-\log p_y$$

where Y is one-hot encoding of y



Three elements to specify a machine learning method:
1. Model class (here the networks)
2. Loss function (cross-entropy for classification, usually squared loss for regression)
3. Optimization method (gradient descent)

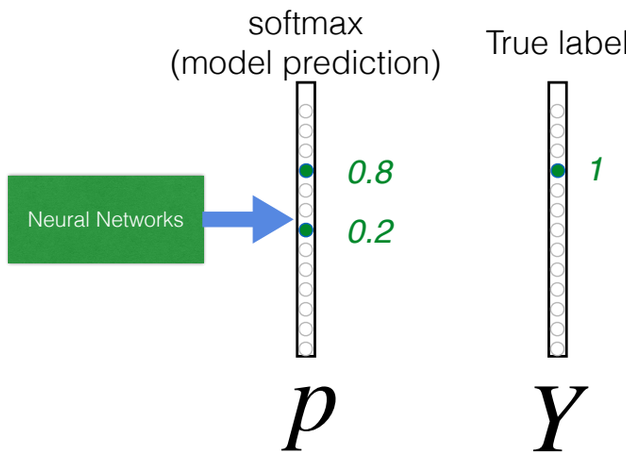
We will consider classification.

The loss on the training set is an average over the training data points. The loss for each data point (per sample loss) is defined to be the cross-entropy between the output probabilistic vector (a distribution over the k classes) and the true probabilistic vector (another distribution over the k classes). cross-entropy is a notion from information theory, measuring the “difference” between two distributions. Here it’s applied on the output probabilities and the true probabilities.

The output probabilities: the output after softmax in the network.
The true probabilities: turn the label y (taking value in {1,2,…,k}) into a one-hot encoding vector Y. Here, Y is a vector of dimension k (corresponding to the k classes), and has value 1 on the dimension corresponding to the true label class y, and has value 0 for the other dimensions. It can be viewed as a probabilistic vector, putting probability mass 1 on the class y, and mass 0 on the other classes.

Since Y is a one-hot vector, the cross-entropy reduces to the negation of the log on the output probability over the true class y. Minimizing the loss is maximizing the output probability over the true class y (i.e., pushing the vector p to be the same as the vector Y).

Cross-Entropy Loss



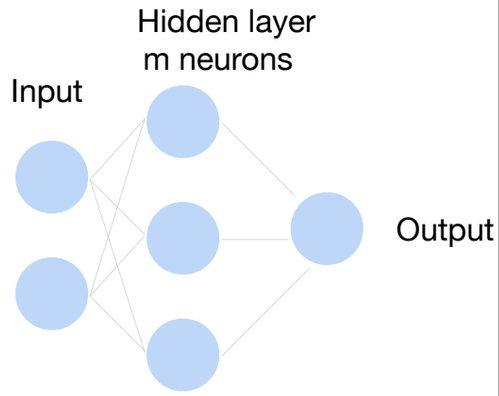
$$L_{CE} = \sum_j - Y_j \log(p_j)$$

$$= -\log(0.8)$$

Goal: push **p** and **Y** to be identical

Binary Cross-Entropy Loss

Loss: $\frac{1}{|D|}\sum_{(\mathbf{x},y)\in D}\ell(\mathbf{x},y)$



Binary cross-entropy loss can be viewed as a special case of cross-entropy loss:

$$\ell(\mathbf{x},y) = -y\log p - (1-y)\log(1-p)$$

Think of the output as a probability vector $(p_0 = 1-p, p_1 = p)$ over the classes {0, 1}:

$$\ell(\mathbf{x},y) = -\log p_y$$

The cross-entropy loss on the previous slide is defined for multiple-class classification. For binary classification, we use the binary cross-entropy loss. Here we show that it can be viewed as a special case of the general cross-entropy loss.

Recall that in binary classification, we have classes y from {0,1}, and we use the output of the network p to model the conditional probability of y=1 conditioned on the input x. The binary cross-entropy loss using these notation is then defined as in this slide.

To see it can be viewed as a special case, we view the output of the network as a probabilistic vector (p_0, p_1), ie, a distribution over the classes {0,1}. Here, p_0 = 1-p, p_1 = p. Then using these notations:

1. If the true class y=0, then the loss simplifies to -log (1-p) = -log p_0
2. If the true class y=1, then the loss simplifies to -log p = -log p_1

Therefore, the binary cross-entropy loss is just -log p_y.

Now we can also apply the general cross-entropy loss definition. Turn the true class y into a one-hot vector Y (a distribution over the classes {0,1}), and then compute the cross-entropy between Y and the output probabilities (p_0, p_1), which is \sum_{j \in {0,1} } -Y_j \log p_j.

1. If the true class y=0, then Y = [1 0], and the cross-entropy loss is -log p_0
2. If the true class y=1, then Y = [0 1], and the cross-entropy loss is -log p_1

We can see that the general cross-entropy loss is also -log p_y.

In summary, if we view the output of the network as a distribution (1-p, p) over the two classes, and also view the true class label as a distribution (one-hot encoding) over the two classes, then the cross-entropy between the two distributions gives the binary cross-entropy loss.

Quiz break

The output probabilities of a network on classes 1 to 5 are shown in the figure. If the label is class 3, which is the cross-entropy loss for this data point?

A. $-\log(3)$
B. $-\log(0.01)$
C. $-\log(0.05)$
D. $-\log(0.90)$

Output probabilities

$$\begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

Quiz break

The output probabilities of a network on classes 1 to 5 are shown in the figure. If the label is class 3, which is the cross-entropy loss for this data point?

A. $-\log(3)$
B. $-\log(0.01)$
C. $-\log(0.05)$
D. $-\log(0.90)$

Output probabilities

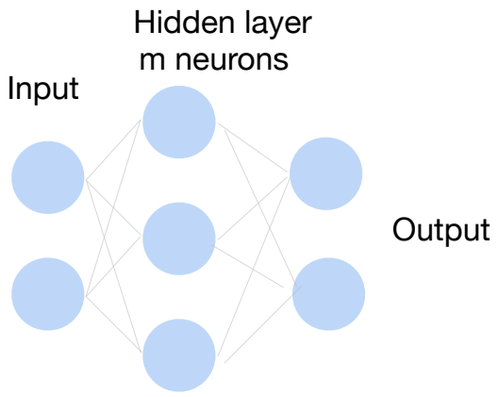
$$\begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

How to train a neural network?

Update the weights W to minimize the loss function

$$L = \frac{1}{|D|} \sum_{(\mathbf{x},y)\in D} \ell(\mathbf{x}, y)$$

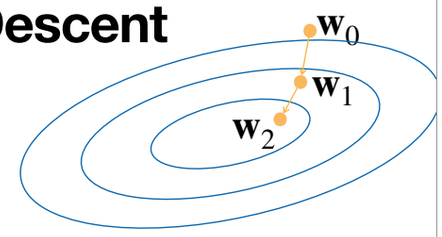
Use gradient descent!



Now we've specified the models and the loss, we are ready to specify the third element in the learning method: the optimization method for minimizing the loss.

For neural networks, typically we use gradient descent. More precisely, we use some variants of gradient descent.

Minibatch Stochastic Gradient Descent



- Choose a learning rate $\alpha > 0$
- Initialize the model parameters w_0
- For t =1,2,…
  - **Randomly sample a subset (mini-batch)** $B \subset D$
    Update parameters:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \frac{1}{|B|} \sum_{\mathbf{x} \in B} \frac{\partial \ell(\mathbf{x}_i, y_i)}{\partial \mathbf{w}_{t-1}}$$

The gradient w.r.t. all parameters is obtained by concatenating the partial derivatives w.r.t. each parameter

- Repeat

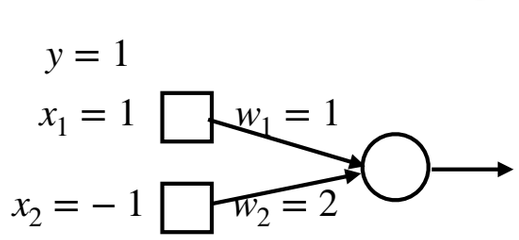
We typically use stochastic gradient descent.

We initialize the model parameter (usually using Gaussian random numbers). We also pick some learning rate \alpha.
Then we run in iterations. In each iteration, randomly sample a batch of training data points; average the gradient of the loss on the sampled training data points w.r.t. the parameters; move along the negation of the average gradient direction with step size \alpha, to get a new set of parameter values. Repeat this until we run out of time budget or are satisfied with the loss.

What's the gradient of the loss on a sampled training data point w.r.t. the parameters?
1. It's a concatenation of the partial derivative w.r.t. each parameter.
2. The partial derivative w.r.t. one particular parameter: we can view the loss as a single-variable function on that particular parameter, and take the gradient of that function. That is, view all the other variables as constant, and compute the gradient. This is the partial derivative w.r.t. that particular parameter.

Note: Here we describe the algorithm as iterations and each iteration samples a batch. In practice, usually it's slightly different: we run in epochs; in each epoch we randomly partition the training data into batches, and then use the batches one by one (ie, run in iterations and each iteration uses one batch).

Example: Backpropagation



sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

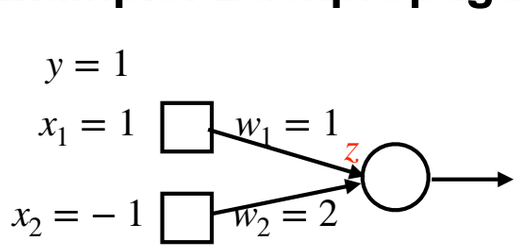
$$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$

$$\hat{y} \longrightarrow \ell(\mathbf{x}, y)$$

Now the question is how to compute the derivative of the loss on one data point w.r.t. a particular parameter. The method for this is called back propagation.

Here we give an example to demonstrate the method. For simplicity, we assume no bias.

Example: Backpropagation



sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$

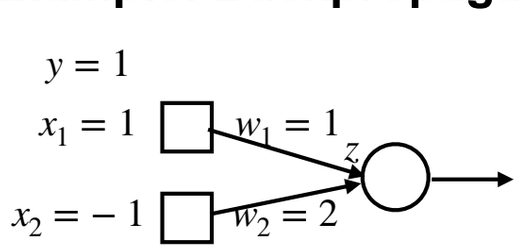
$\hat{y} \longrightarrow \ell(\mathbf{x}, y)$

- Forward:

  $z = w_1 x_1 + w_2 x_2 = 1 \times 1 + (-1) \times 2 = -1$

We first perform the forward computation.
We introduce an intermediate variable z to denote the result of the linear transformation. The value is -1.

Example: Backpropagation



sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$

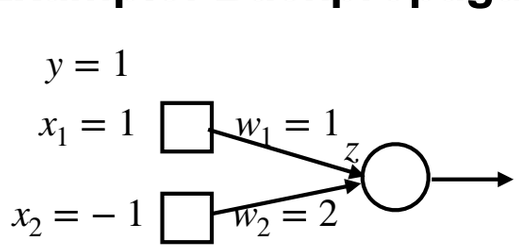
- Forward:

$$z = w_1 x_1 + w_2 x_2 = 1 \times 1 + (-1) \times 2 = -1$$

$$\hat{y} = \sigma(z) = \sigma(-1)$$

Then we move one step forward: compute the output \hat{y} which is sigmoid on z.

Example: Backpropagation



sigmoid activation, binary cross-entropy loss

$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$

$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y})$

$\hat{y} \longrightarrow \ell(\mathbf{x}, y)$

- Forward:

$z = w_1 x_1 + w_2 x_2 = 1 \times 1 + (-1) \times 2 = -1$

$\hat{y} = \sigma(z) = \sigma(-1)$

$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y}) = -\log \hat{y} = -\log \sigma(-1)$

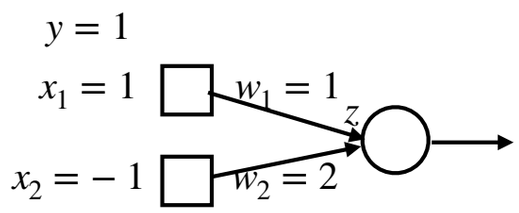
We again move one step forward: compute the loss which depends on the output \hat{y} and the true label y.
(To compute the derivative actually we don’t need to compute the loss. Here we show it for illustration.)

Example: Backpropagation

sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1-y)\log(1-\hat{y})$$



- Forward:
  $z = -1, \hat{y} = \sigma(-1), \ell(\mathbf{x}, y) = -\log \sigma(-1)$
- Backward: $\frac{\partial \ell}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}(-y \log \hat{y} - (1-y)\log(1-\hat{y}))$

Partial derivative: view all the other variables as constants, and compute the gradient

$$= \frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} = \frac{\sigma(-1)-1}{\sigma(-1)(1-\sigma(-1))}$$

Now we are ready to compute the backward direction.

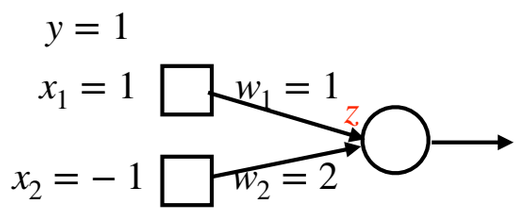
First go back one step: compute the derivative (of the loss) w.r.t. the output \hat{y}. Note that we can view the loss as a function on the single variable \hat{y} and view y as a constant, then compute the gradient.

Example: Backpropagation

sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -\, y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$



- Forward:

  $z = -1, \hat{y} = \sigma(-1), \ell(\mathbf{x}, y) = -\log \sigma(-1)$

- Backward: $\dfrac{\partial \ell}{\partial \hat{y}} = \dfrac{\sigma(-1) - 1}{\sigma(-1)(1 - \sigma(-1))}$

$$\frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z}$$

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial}{\partial z}\sigma(z) = \sigma(z)(1 - \sigma(z)) = \hat{y}(1 - \hat{y}) = \sigma(-1)(1 - \sigma(-1))$$

Now we go backward one step: compute the derivative w.r.t. z.

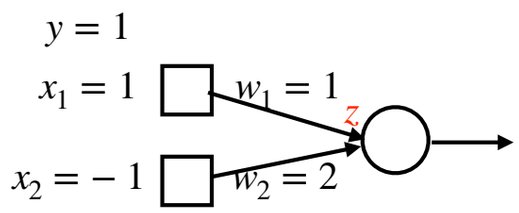
Applying chain rule, the derivative of the loss w.r.t. z is the derivative of the loss w.r.t. \hat{y}, times the derivative of \hat{y} w.r.t. z. The first term has already been computed (that’s why we go backward!), so we only need to compute the second term. Because we only go backward one step, the second term is simple: \hat{y} = sigmoid(z) so the derivative is just the derivative of the sigmoid function. (This is why each time we go backward only for *one step*.)

Example: Backpropagation

sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -\, y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$

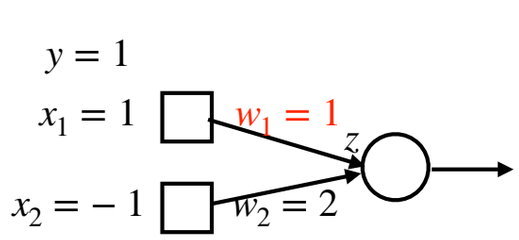


- Forward:

  $z = -1, \hat{y} = \sigma(-1), \ell(\mathbf{x}, y) = -\log \sigma(-1)$

- Backward: $\dfrac{\partial \ell}{\partial z} = \dfrac{\partial \ell}{\partial \hat{y}} \dfrac{\partial \hat{y}}{\partial z} = \sigma(-1) - 1$

Example: Backpropagation



sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$

- Forward:

  $z = -1, \hat{y} = \sigma(-1), \ell(\mathbf{x}, y) = -\log \sigma(-1)$

- Backward: $\dfrac{\partial \ell}{\partial z} = \dfrac{\partial \ell}{\partial \hat{y}} \dfrac{\partial \hat{y}}{\partial z} = \sigma(-1) - 1$

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial \ell}{\partial z} \frac{\partial z}{\partial w_1}$$

$$\frac{\partial z}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 x_1 + w_2 x_2) = x_1 = 1$$

Finally, we go backward one step and reach the goal: compute the derivative of the loss w.r.t. the parameter w_1.

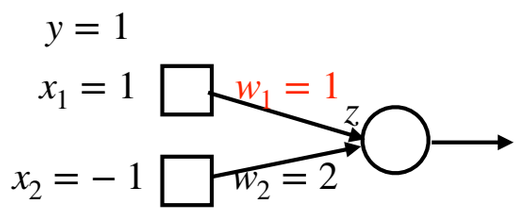
Applying chain rule, the derivative of the loss w.r.t. w_1 is the derivative of the loss w.r.t. z, times the derivative of z w.r.t. w_1. The first term has already been computed in the previous step, so we only need to compute the second term. The second term is simple: z is a linear function of w_1 so the derivative is just the coefficient x_1.

Example: Backpropagation

sigmoid activation, binary cross-entropy loss

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = -\, y \log \hat{y} - (1 - y)\log(1 - \hat{y})$$



- Forward:

  $z = -1, \hat{y} = \sigma(-1), \ell(\mathbf{x}, y) = -\log \sigma(-1)$

- Backward:

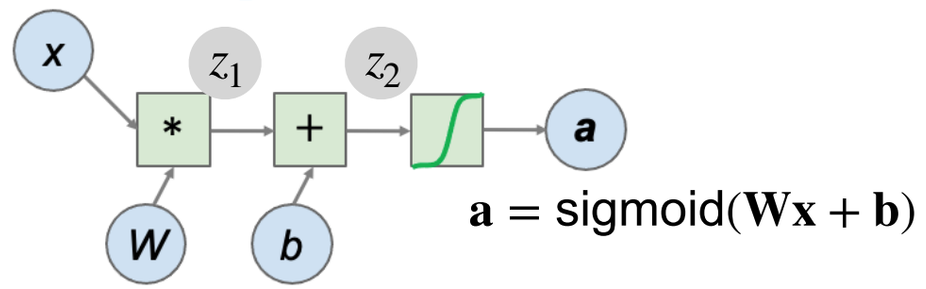
$$\frac{\partial \ell}{\partial w_1} = \frac{\partial \ell}{\partial z}\frac{\partial z}{\partial w_1} = (\sigma(-1) - 1) \times 1 = \sigma(-1) - 1$$

Calculate Gradient: Backpropagation with Chain Rule

- Define a loss function L
- Gradient to a variable =

  gradient on the top x gradient from the current operation

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{z_1}} \frac{\partial \boldsymbol{z_1}}{\partial \boldsymbol{W}}$$



$$\mathbf{a} = \text{sigmoid}(\mathbf{Wx} + \mathbf{b})$$

In general, we can draw a computational graph (introducing intermediate variables after each computational operation).

Backpropagtion is going from the output backward to the input. Each time we go backward one step and compute the derivative of the loss w.r.t. some variable; apply chain rule so that we can reuse the derivative computed in previous steps.

Note: Here the notations like \partial L /\partial W are vector/matrix notations. You can think of \partial L /\partial W as concatenating the derivative of the loss w.r.t. each parameter. Another example is \partial z_1 /\partial W: you can think of it as a matrix, where the rows correspond to dimension in z_1 and columns corresponds to parameters in W, and each entry is the derivative of the corresponding dimension in z_1 w.r.t. the corresponding parameter in W. So the chain rule now is a product of multiple matrices (it can contain vectors which can also be viewed as a special kind of matrices).

In this course we don't require to handle chain rule in this matrix form. We present it here only for

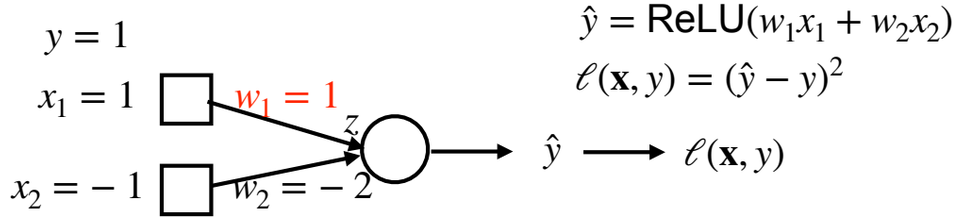
1. demonstrating how backpropagation is done in general (the matrix form can compactly represent this).
2. showing the reason for vanishing gradients: when each matrix in the chain is small (w.r.t. certain "size" measurement of the matrix) then the product result (the gradient) will be very small.

Quiz break

ReLU activation, square loss

$$\hat{y} = \text{ReLU}(w_1 x_1 + w_2 x_2)$$

$$\ell(\mathbf{x}, y) = (\hat{y} - y)^2$$



What is the value of $\frac{\partial \ell}{\partial w_1}$?

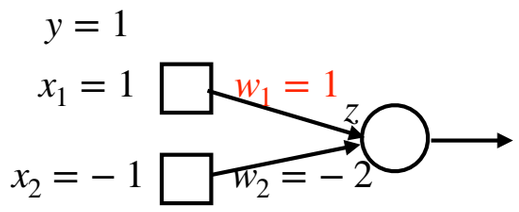
A. 1

B. 2

C. 3

D. 4

Quiz break



What is the value of $\frac{\partial \ell}{\partial w_1}$?

A. 1

B. 2

C. 3

D. 4

ReLU activation, square loss

$\hat{y} = \text{ReLU}(w_1 x_1 + w_2 x_2)$

$\ell(\mathbf{x}, y) = (\hat{y} - y)^2$

$\hat{y} \longrightarrow \ell(\mathbf{x}, y)$

- Forward: $z = 3, \hat{y} = 3, \ell(\mathbf{x}, y) = 4$
- Backward: $\frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y) = 4,$

$\frac{\partial \ell}{\partial z} = 4 \times 1 = 4,$

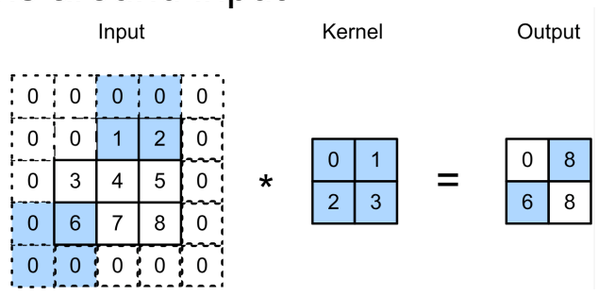
$\frac{\partial \ell}{\partial w_1} = 4 \times 1 = 4$

Note that the gradient of ReLU(z) is 1 when z>0, and is 0 otherwise.

Convolutional Neural Networks

2-D Convolution Layer with Stride and Padding

- Stride is the #rows/#columns per slide
- Padding adds rows/columns around input
- Output shape



Kernel/filter size

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

Input size Pad Stride

This page compactly summarizes the knowledge about 2D convolution. Note that we can derive the output shape expression quite easily.

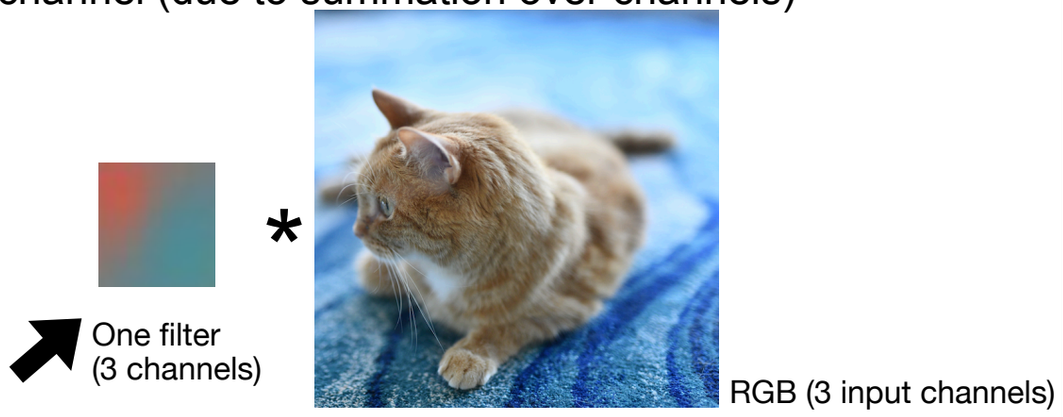
In general, suppose we have input matrix X with height n_h and width n_w, and we have kernel matrix W with height k_h and k_w, and we pad p_h rows and p_w columns, and we use stride s_h for height and s_w for width. Then the output is a matrix of a particular size in the expression.

To see this, consider only the width (same reasoning for the height). The padded input has n_w+p_w columns, indexed as column 1 to column (n_w+p_w).
In the leftmost location, the sliding window's left end aligns with the column 1 of the input.
If we slide it to the right once, the sliding window's left end aligns with the column 1+s_w of the input.
If we slide it to the right 2 times, the sliding window's left end aligns with the column 1+2*s_w of the input.
…
Suppose the rightmost location is by sliding the original window to the right t times, then the sliding window's left end aligns with the column 1+t*s_w of the input.
Then the sliding window's right end aligns with the column (1+t*s_w + k_w -1) of the input.

We know that (1+t*s_w + k_w -1) <= n_w+p_w. This means
t <= (n_w - k_w + p_w)/s_w.
Note that the total number of possible locations for the sliding window is (t+1):
t+1 <= (n_w - k_w + p_w + s_w)/s_w.
It needs to be an integer. So
t+1 = floor((n_w - k_w + p_w + s_w)/s_w).
This is just the output width.

Multiple Input Channels

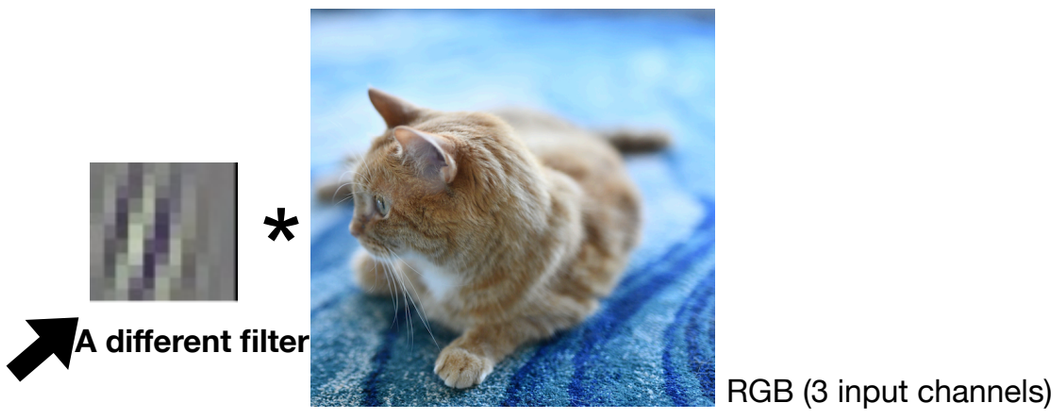
- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Also call each 3D kernel a "**filter**", which produce only **one** output channel (due to summation over channels)



The input can have multiple channels (RGB in colored images). Then we can have a kernel with the same number of channels (each channel is a kernel matrix). We can do the convolution on each channels, then sum up the results into one output matrix (so only one output channel).

Multiple Filters Lead to Multiple Output Channels

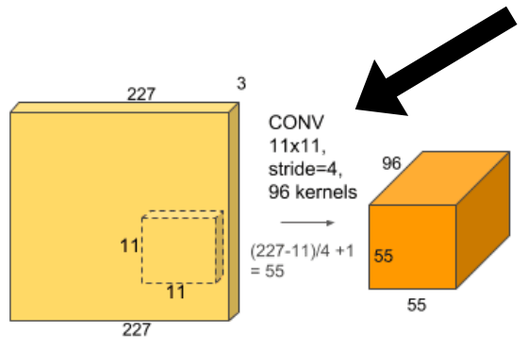
- Apply multiple filters on the input
- Each filter may learn different features about the input
- Each filter (3D kernel) produces one output channel

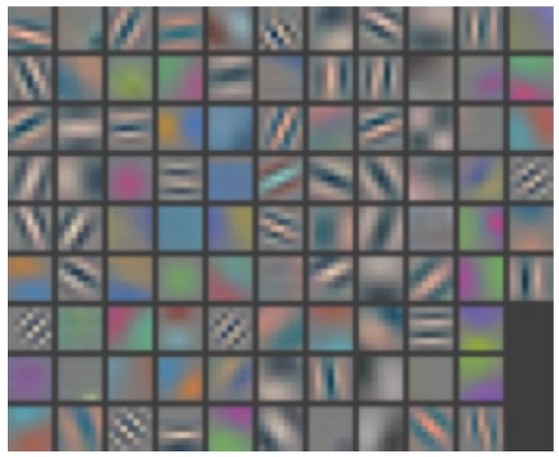


If we want multiple output channels, then we can use multiple 3D kernels (each 3D kernel has multiple channels and each channel in the kernel is a matrix). Each 3D kernel will give an output channel.

Conv1 Filters in AlexNet

- 96 filters (each of size 11x11x3)
- Gabor filters





Figures from Visualizing and Understanding Convolutional Networks by *M. Zeiler and R. Fergus*

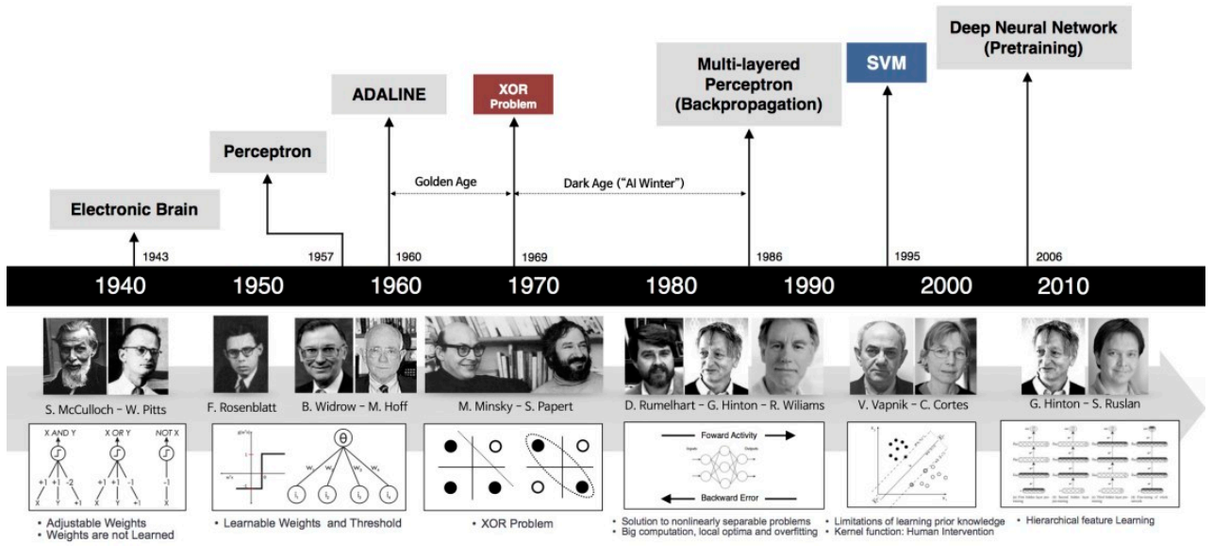
Kernels can learn interesting patterns in the images.

For example, here shows the learned kernels in the first convolutional layer in AlexNet. They are essentially the so-called Gabor filters, which are known to be important for images and for human vision systems. Note that the networks and the training methods are not designed with Gabor filters in mind, but they automatically learn these important and interesting filters! It's an active research direction to understand why they can automatically do so.

Recall that the output entry is computed by multiplying the corresponding entries in the kernel and the sliding window and sum up the results. If we flatten the kernel and window as vectors, then the output entry is just their inner product. We know that inner product measures the similarity. So if the window contains the pattern that is represented in the kernel, then we get a large output entry, ie, we detect the pattern in the window.

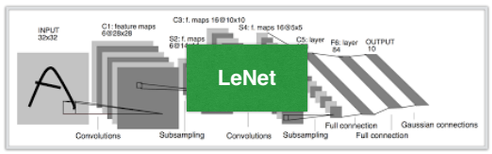
Then the convolution layers learn interesting patterns and store them in the kernels. After training, when applied on an image, they are trying to detect these patterns in the sliding windows on the input and based on the detection information the network can perform different tasks like image classification.

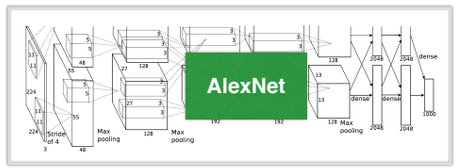
Brief history of neural networks

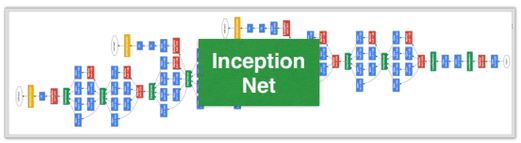


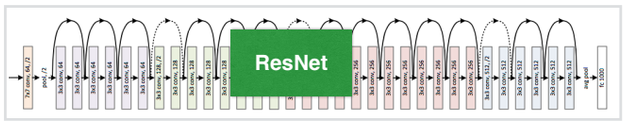
A very brief summary of the history. Note that LeNet (1998) and Pretraining techniques (2006) have been proposed quite early, but only after AlexNet, the community are convinced about the superior performance of deep networks and turn to them.

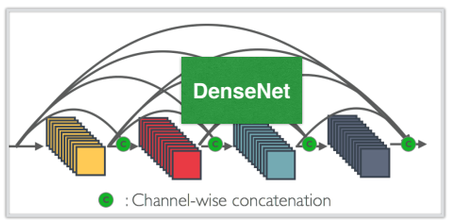
Evolution of modern deep neural net architectures









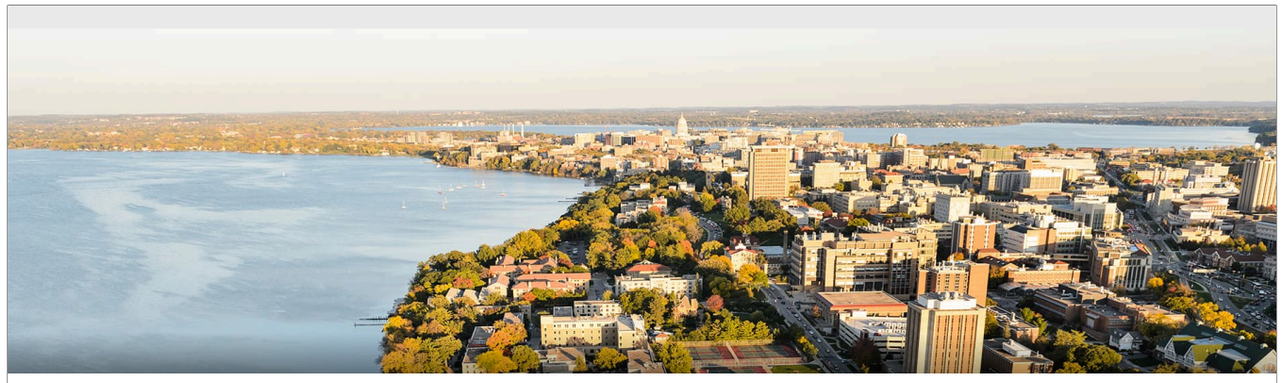


Looking at the history of deep learning, the evolution of neural network architectures is an important part of the picture.

In the last several years, the research community has made tremendous amount of progress towards this goal. We've seen better and better model architecture design which improves the accuracy over time. This is what powers a lot of applications you are using on your phone.

What we've learned today…

- Modeling a single neuron
  - Perceptron
  - Limited power of a single neuron
- Multi-layer perceptron
- Training of neural networks
  - Loss function (cross entropy)
  - Backpropagation and SGD
- Convolutional neural networks
  - Convolution, pooling, stride, padding
  - Basic architectures (LeNet etc.)
  - More advanced architectures (AlexNet, ResNet etc)



Thank you!

Some of the slides in these lectures have been adapted from materials developed by Alex Smola and Mu Li:
https://courses.d2l.ai/berkeley-stat-157/index.html