Deep learning

Lecturer: Dr. Yangchen Pan, Department of Engineering Science

Acknowledgement: the slides (8-10, 12) are from the previous instructor Naeemullah Khan

Announcements/common problems

1. In the GLM exercise, the prediction of Poission regression should be the exponential of x^T w

2. Skip the exercise about smoothness in the optimization part (Q5 in ex1-optimization regression)

3. I fixed two typos in ex2-stochasticoptimization, the cross entropy loss function and the gradient in Q2. The loss is a scalar; the gradient is matrix and is a summation of N outer products. You probably won't get confused because of the typos.

4. To prove convex, you need to prove the Hessian matrix of the loss function is positive semidefinite.

5. In the line search method, you need to add a for loop to search for best eta.

Recall the derivation of generalized linear models

Given a single sample (x, y):

1. Define a linear function: $x^\top w$

2. Use a transfer function t() to transfer the linear output: $\hat{y} = t(x^\top w)$

3. Define a loss function between yhat and y: $l(\hat{y}, y) = l(t(x^\top w), y)$

e.g.:
Linear regression: $t(z) = z, l(\hat{y}, y) = (\hat{y}, y)^2$

Logistic regression:

Recall the derivation of generalized linear models

Given a single sample (x, y):

1. Define a linear function: $x^\top w$

2. Use a transfer function t() to transfer the linear output: $\hat{y} = t(x^\top w)$

3. Define a loss function between yhat and y: $l(\hat{y}, y) = l(t(x^\top w), y)$

e.g.:
Linear regression: $t(z) = z, l(\hat{y}, y) = (\hat{y}, y)^2$

Logistic regression: $t(z) = sigmoid(z)\ l(\hat{y}, y) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$

What's special about deep learning?

Given a single sample (x, y):

1. Define a linear function: $x^\top w$

   Use a deep NN: $\phi_\theta(x)^\top w$

2. Use a transfer function t() to transfer the linear output: $\hat{y} = t(x^\top w)$

   Use a deep NN: $t(\phi_\theta(x)^\top w)$

3. Define a loss function between yhat and y : $l(\hat{y}, y) = l(t(x^\top w), y)$

   Use a deep NN: $l(\hat{y}, y) = l(t(\phi_\theta(x)^\top w), y)$

Note: now your training parameters include both w and theta.

To train **w, theta** in a deep NN, the most commonly used algorithm is called backpropagation.

What’s an (artificial) neural network?

A network of neurons – organize neurons together to produce an output

What's an (artificial) neural network?

A network of neurons – organize neurons together to produce an output

What is a neuron then? --- an activation function (i.e., transfer function, usually a nonlinear function) applied to a linear function

What's an (artificial) neural network?

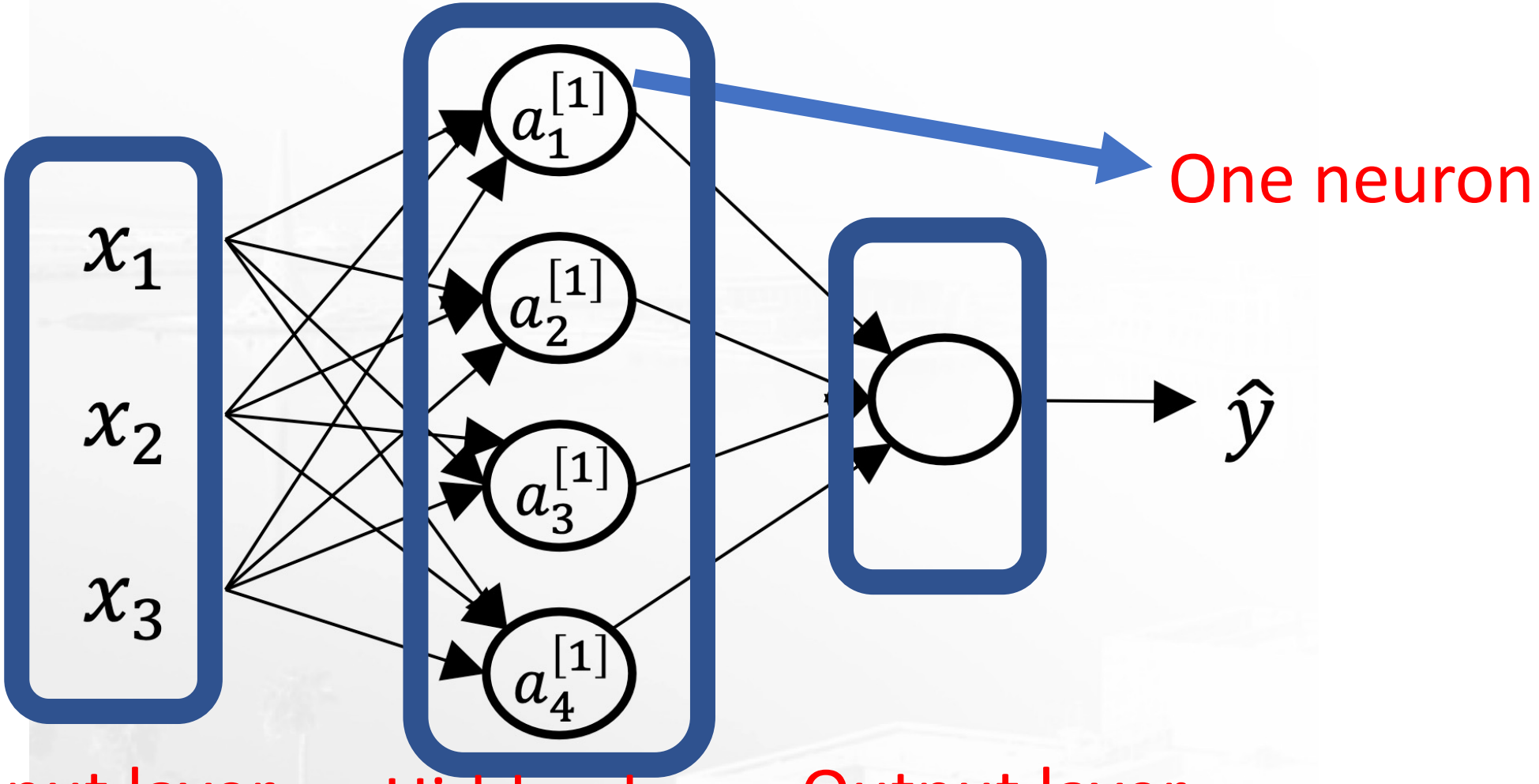
A network of neurons – organize neurons together to produce an output

What is a neuron then? --- an activation function (i.e., transfer function, usually a nonlinear function) applied to a linear function

How to organize these neurons to get output?

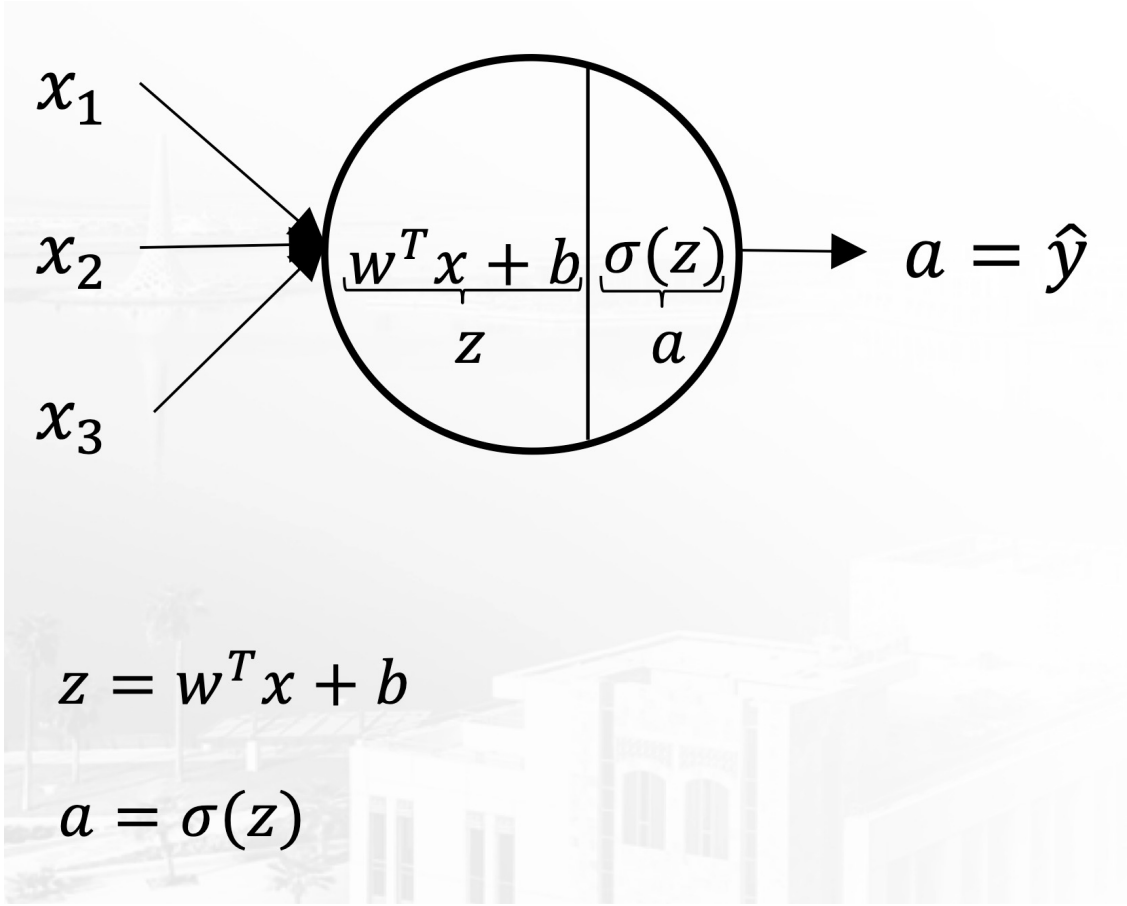
What's an (artificial) neural network?

How to organize these neurons to get output? – a classic example



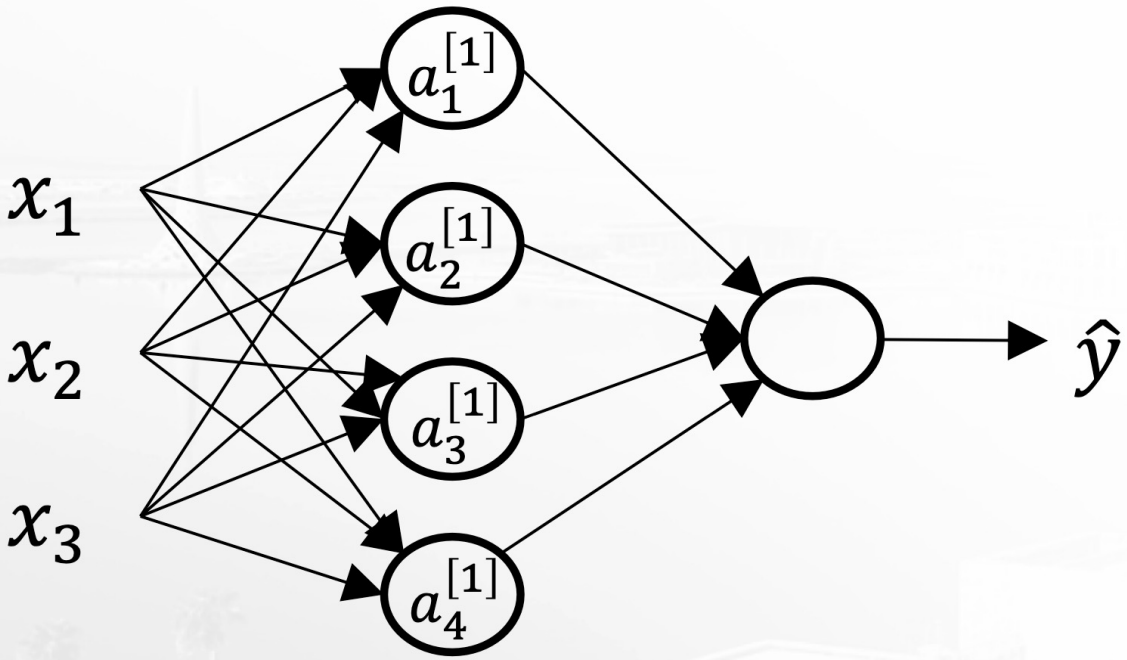
One neuron

How to organize these neurons to get output? – a classic example



Activation function: sigmoid, tanh, relu, etc.

From input to output



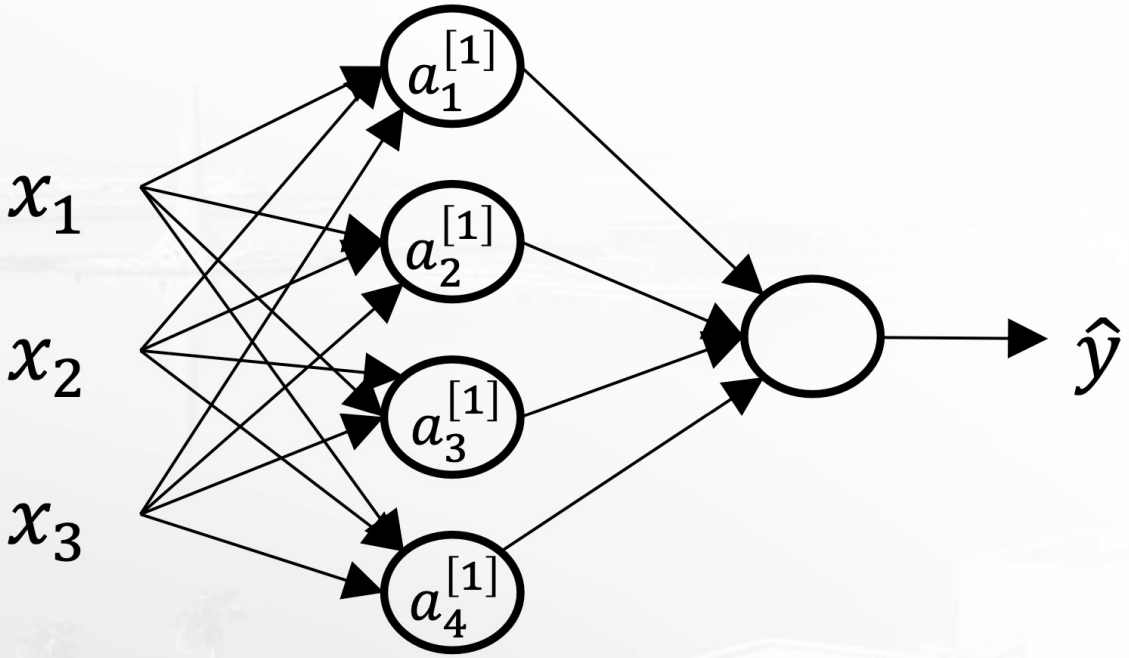
$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

From input to output – matrix notation



Given input x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

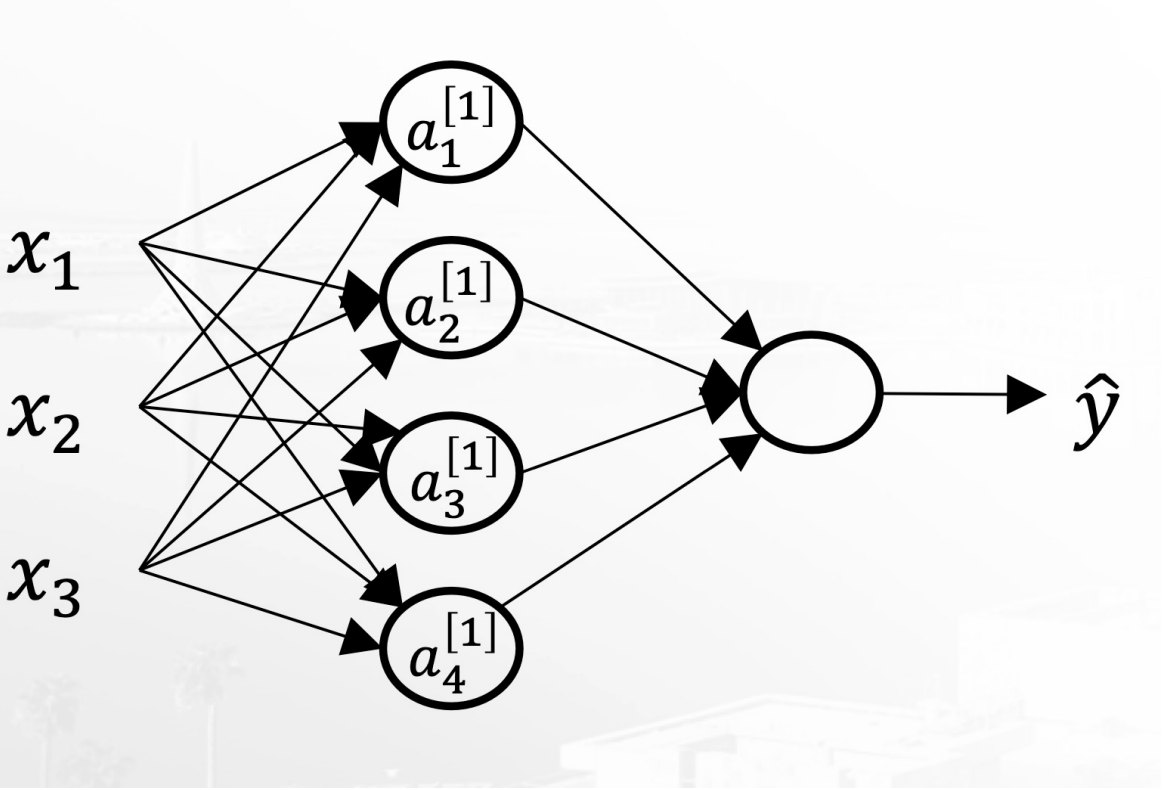
$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$\hat{y} = z^{[2]}$$

Then we define loss function $l(\hat{y}, y)$

How to train a NN?



Given input x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$\hat{y} = z^{[2]}$$

Once we have the loss function $l(\hat{y}, y)$, we can take gradient w.r.t. the training parameters, this algorithm is called backpropagation. It is just taking gradient by chain rule with careful ordering to avoid repeat computations.

How to train a neural network?

$$\hat{y} = f^{(2)}(w^{(2)} f^{(1)}(w^{(1)} x)) \text{ where}$$
$$x \in \mathbb{R}^{d \times 1}, w^{(1)} \in \mathbb{R}^{d_1 \times d}, w^{(2)} \in \mathbb{R}^{1 \times d_1}, f^{(1)} \in \mathbb{R}^{d_1 \times 1} \text{ and}$$
$$z^{(1)} = w^{(1)} x, z^{(2)} = w^{(2)} f^{(1)}$$

$$\frac{\partial L(\hat{y}, y)}{\partial w^{(2)}} = \boxed{\frac{\partial L}{\partial f^{(2)}} \cdot \frac{\partial f^{(2)}}{\partial z^{(2)}}} \frac{\partial z^{(2)}}{\partial w^{(2)}}$$
$$= (\hat{y} - y) f^{(1)\top}$$
$$\frac{\partial L(\hat{y}, y)}{\partial w^{(1)}} = \boxed{\frac{\partial L}{\partial f^{(2)}} \cdot \frac{\partial f^{(2)}}{\partial z^{(2)}}} \frac{\partial z^{(2)}}{\partial f^{(1)}} \cdot \frac{\partial f^{(1)}}{\partial z^{(1)}} \cdot \frac{\partial z^{(1)}}{\partial w^{(1)}}$$

Implementation details

$$dz^{[2]} = a^{[2]} - y \qquad dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = dz^{[2]} a^{[1]^T} \qquad dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$$

$$db^{[2]} = dz^{[2]} \qquad db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]}) \qquad dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^T \qquad dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = dz^{[1]} \qquad db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

NNs with special architectures

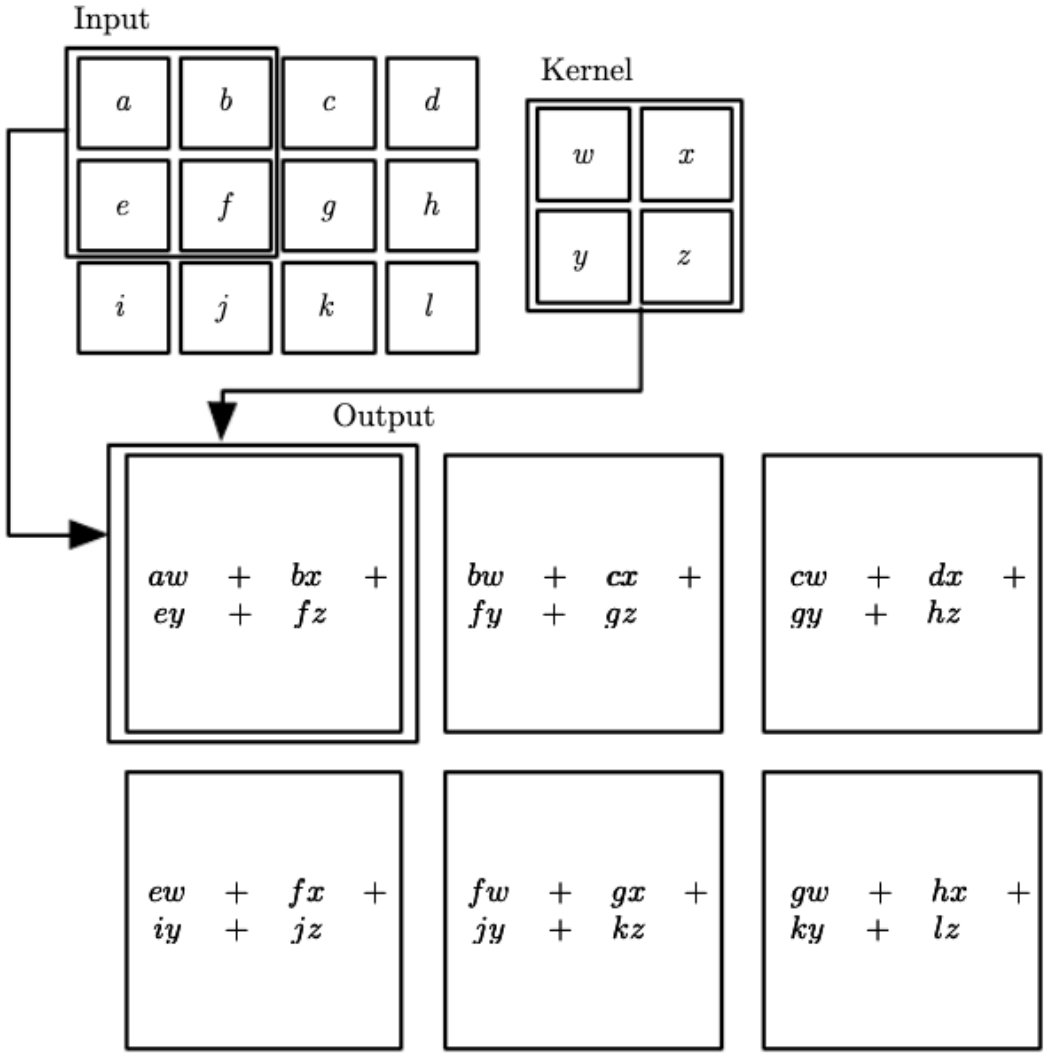
Think about these special architectures as priors: there could be special benefits when dealing with different types of input signal.

Special NNs --- Capture Regularity in the input space
Convolutional NN

- CNN:
  - Replace the linear operation in a regular NN by a discretized convolutional operation (in fact, many software implements it as a cross-correlation), which is defined through a filter/kernel
  - For example:

Capture Regularity in the input space Convolutional NN

- CNN: convolutional operation example



Source: Deep learning by Ian Goodfellow et al

Stride: how much the kernel move after each inner product;

Padding: add zeros to the image's perimeter--- we tend to lose pixels on the perimeter of our image

Capture Regularity in the input space Convolutional NN

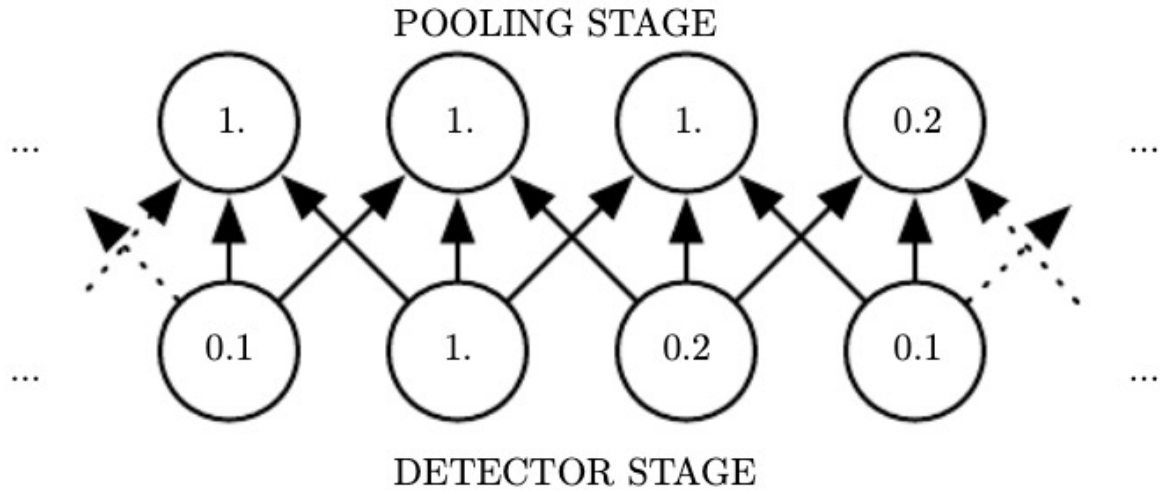
- CNN leverages three important ideas which are helpful to improve learning:
  - Sparse interaction (Also called sparse connectivity/weights)
  - Parameter sharing
    - **Be aware the difference with Parameter Tying**
  - Equivariant representation: a function is equivariant if f(g(x)) = g(f(x)). Convolutional operation is ***translational equivariant***
    - **Be aware the difference with *translational invariant***

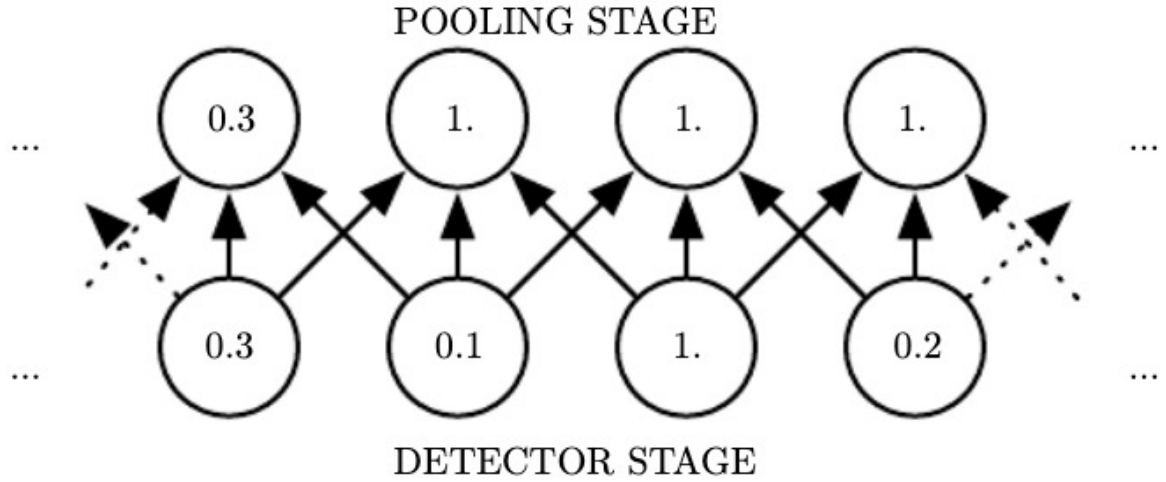
Capture Regularity in the input space Convolutional NN

- Equivariant to translation: if we move an object from one location to another, the representation will move the same amount in the output
  - NOTE: a convolutional operation itself is NOT naturally equivariant to rotations/scaling operation

- Translation Invariant: pooling operation (usually applied after the filtering and activation function) has this property ***to some degree***.
  - Max/Average pooling operation: down sampling the input to a scalar by taking the maximum/average value among the input

Capture Regularity in the input space Convolutional NN





The bottom figure shifts the input from the top figure to the right by one unit. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are sensitive only to the maximum value in the neighborhood, not its exact location.

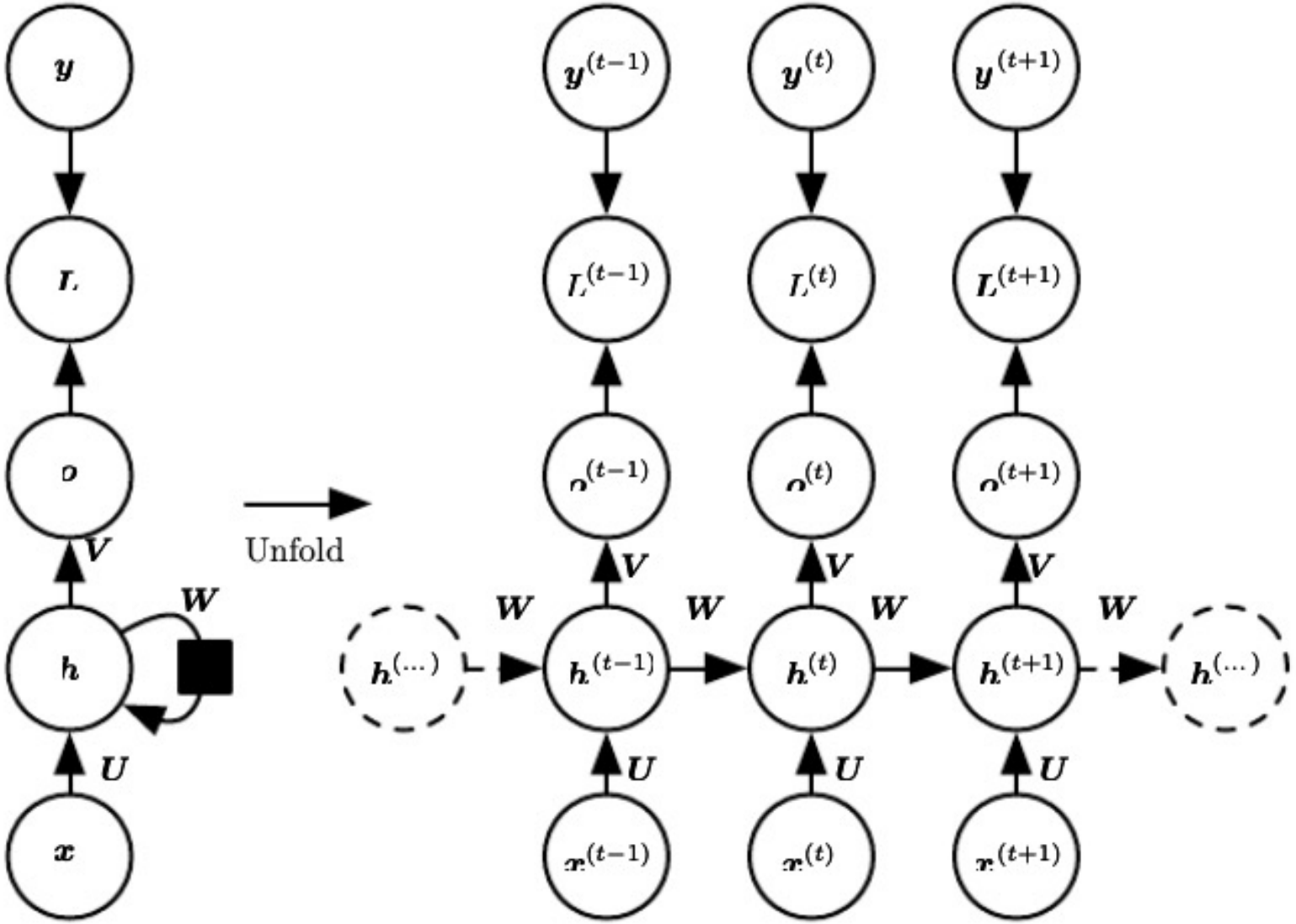
Source: Deep learning by Ian Goodfellow et al

Capture Regularity in the input space
Recurrent NN

- Recurrent NN:
  - Weight sharing across time: **W, U, b, c, V** are parameters shared across a sequence of input samples: $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$
$$h^{(t)} = tanh(a^{(t)})$$
$$o^{(t)} = c + Vh^{(t)}$$

Capture Regularity in the input space



Source: Deep learning by Ian Goodfellow et al

Issues with big NNs

- Recurrent NN has difficulty in training when the time length is long, to see this assume the recurrent weight W admits an eigenvalue decomposition:

$$W = Q\Lambda Q^\top, h^{(t)} = Q^\top \Lambda^t Q h^{(0)}$$

- Elements in h will become explode or vanished eventually
- Similar thing happens during backpropagation process

Capture Regularity in the input space

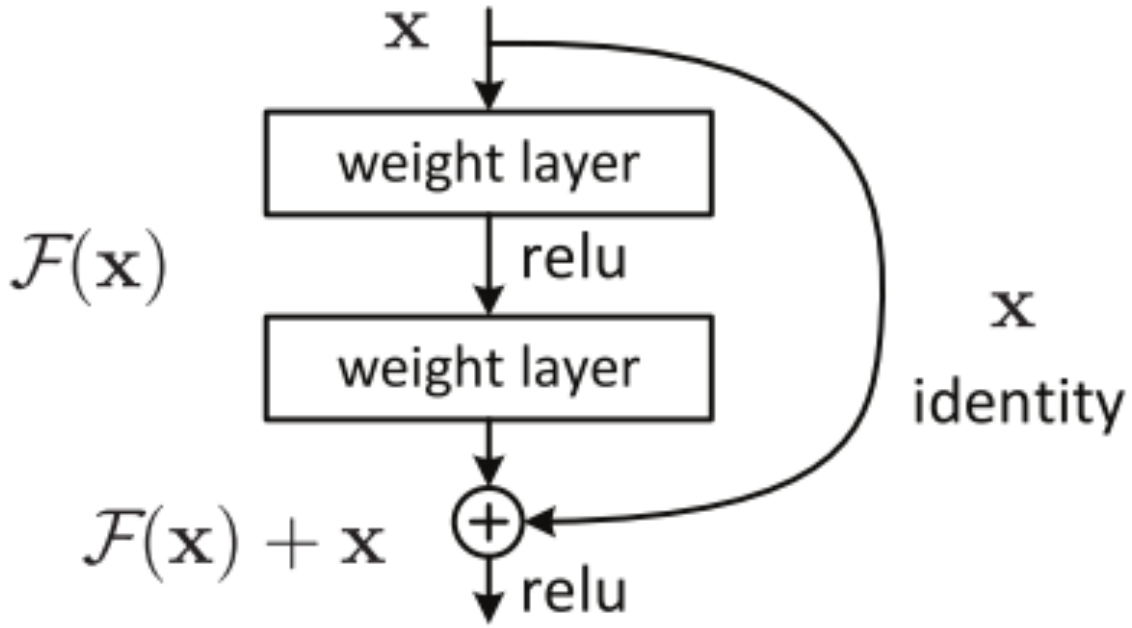
- Some ideas to help:
  - Echo State NN: fix the recurrent weights and only train the output weights, fix the recurrent weight such that the spectral radius is around some reasonably small value
  - Some explicit regularization technique to force the gradient magnitude across time steps to be similar
  - Leaky units/skip connections/remove connections
  - LSTM, Gated RNN (maybe practically most useful)

Capture Regularity in the input space

- Leaky units: $h_t = \alpha h_{t-1} + (1 - \alpha) h_t$
- Skip connection: add connection between distant nodes
- Remove connection: actively replace one-step time connection by multiple time step connection
- LSTM/GRU(Gated Recurrent Unit)
  - **Self-loop** within a cell: similar to leaky unit but with a forgetting gate to control the self-loop weight
  - **Gating** mechanism for input, forgetting, output

Gradient vanish issue in ***really deep*** neural networks

- The well-known ResNet has similar idea: it uses shortcut connection to skip one or more layers



A residual block. Source: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035

Thank you!