6.7900 Machine Learning (Fall 2024)

Lecture 11: learning neural networks

(supporting early release slides)

Recall: composing complex models (MLP)

- A linear model $\underset{1x1}{f(x;\theta)} = \underset{1xd}{w^T}\underset{dx1}{x} + \underset{1x1}{b}$ $\quad \theta = \{w, b\}$
- A linear model with features $\underset{1x1}{f(x;\theta)} = \underset{1xm}{w^T}\underset{mx1}{\phi(x)} + \underset{1x1}{b}$ $\quad \theta = \{w, b\}$
- A linear model with learnable linear features… still just a linear model!!

$$\underset{1x1}{f(x;\theta)} = \underset{1xm}{w^T}(\underset{mxd}{W^{(1)}}x + \underset{mx1}{b^{(1)}}) + \underset{1x1}{b} \qquad \theta = \{w, b, W^{(1)}, b^{(1)}\}$$

- One hidden layer model (linear + non-linear + linear)

$$\underset{1x1}{f(x;\theta)} = \underset{1xm}{w^T}\underset{mx1}{\tanh}(\underset{mxd}{W^{(1)}}x + \underset{mx1}{b^{(1)}}) + \underset{1x1}{b} \qquad \theta = \{w, b, W^{(1)}, b^{(1)}\}$$

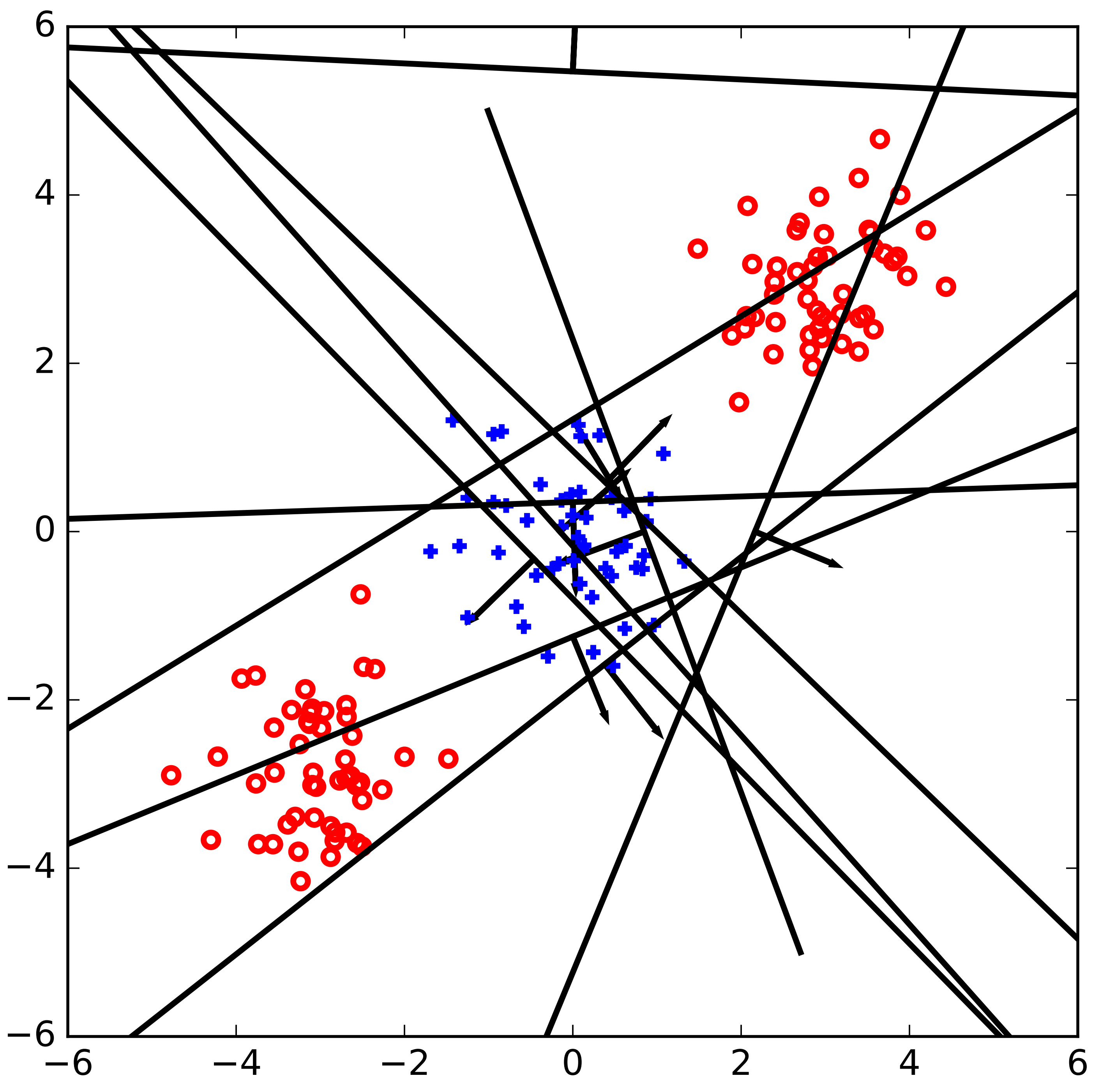
- A multi-layer neural perceptron (MLP, multiple linear +non-linear steps), e.g.,

$$\underset{1x1}{f(x;\theta)} = \underset{1xk}{w^T}\underset{kx1}{\tanh}\left(\underset{kxm}{W^{(2)}}\underset{mx1}{\tanh}(\underset{mxd}{W^{(1)}}x + \underset{mx1}{b^{(1)}}) + \underset{kx1}{b^{(2)}}\right) + b \qquad \theta = \{w, b, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$$

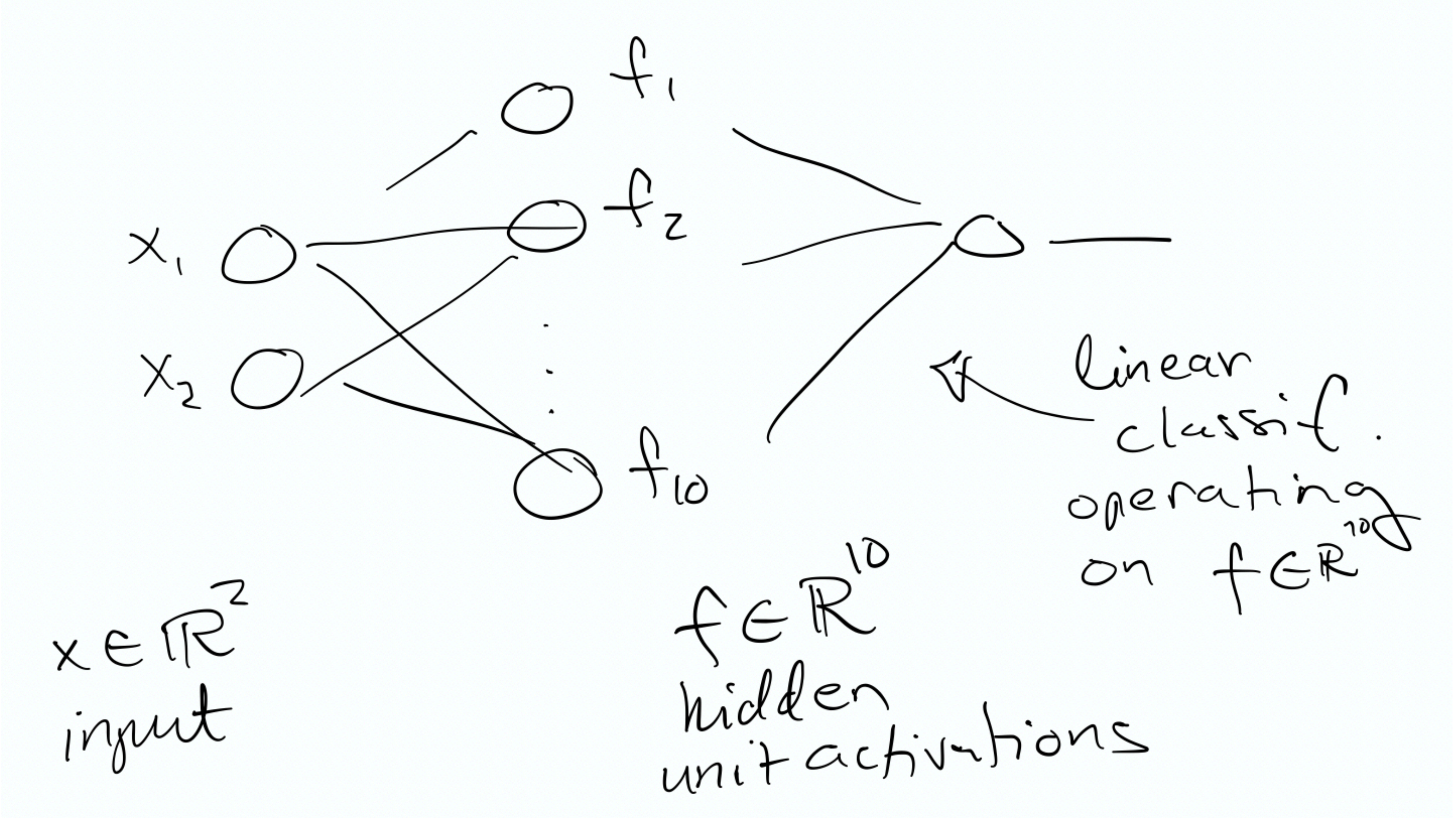
Recall: randomly initialized parameters

- A large number of randomly initialized hidden units gives a meaningful feature expansion

Hidden layer units



(10 randomly initialized units)

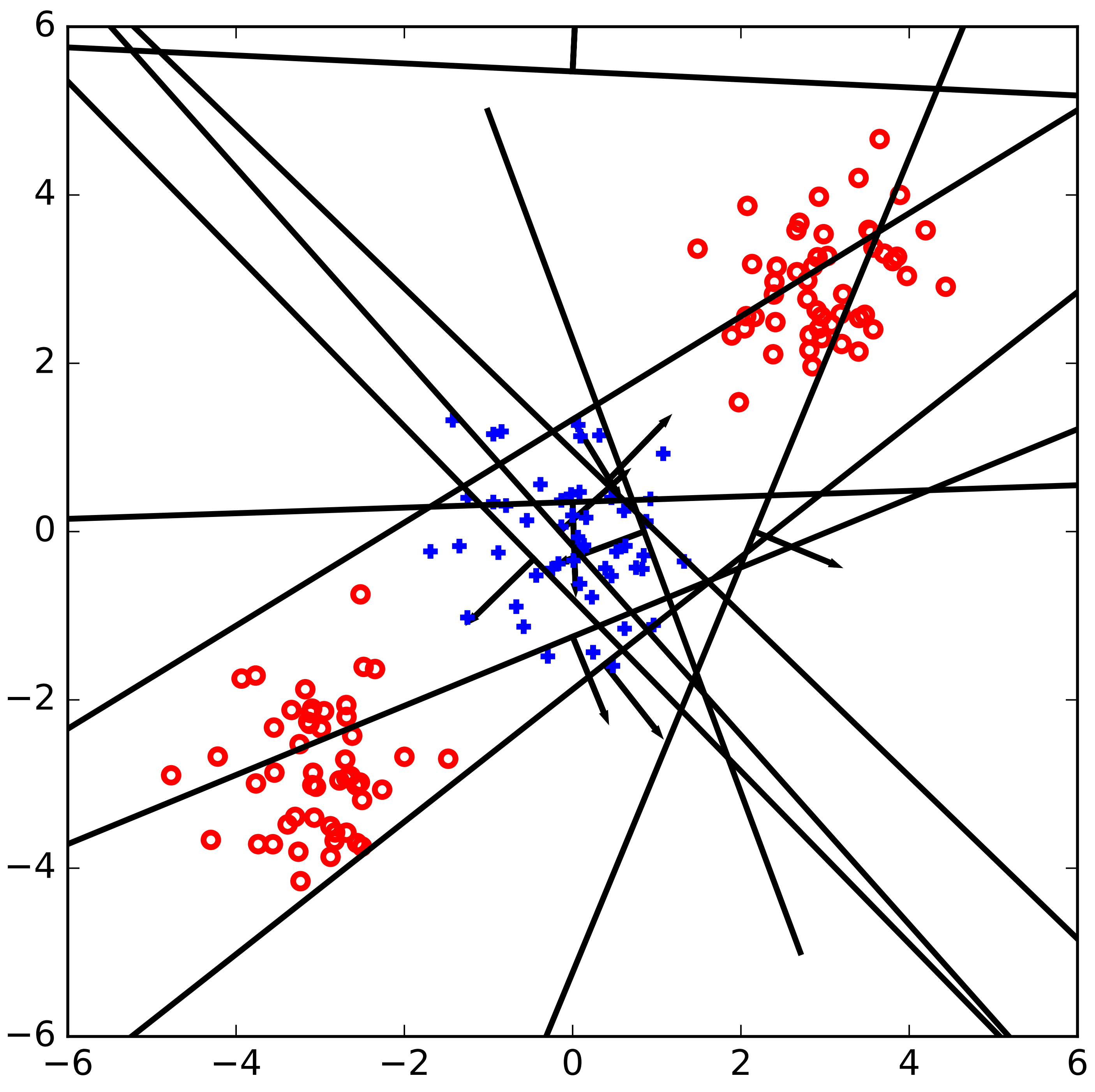


The points are now linearly separable in the resulting 10 dimensional space!

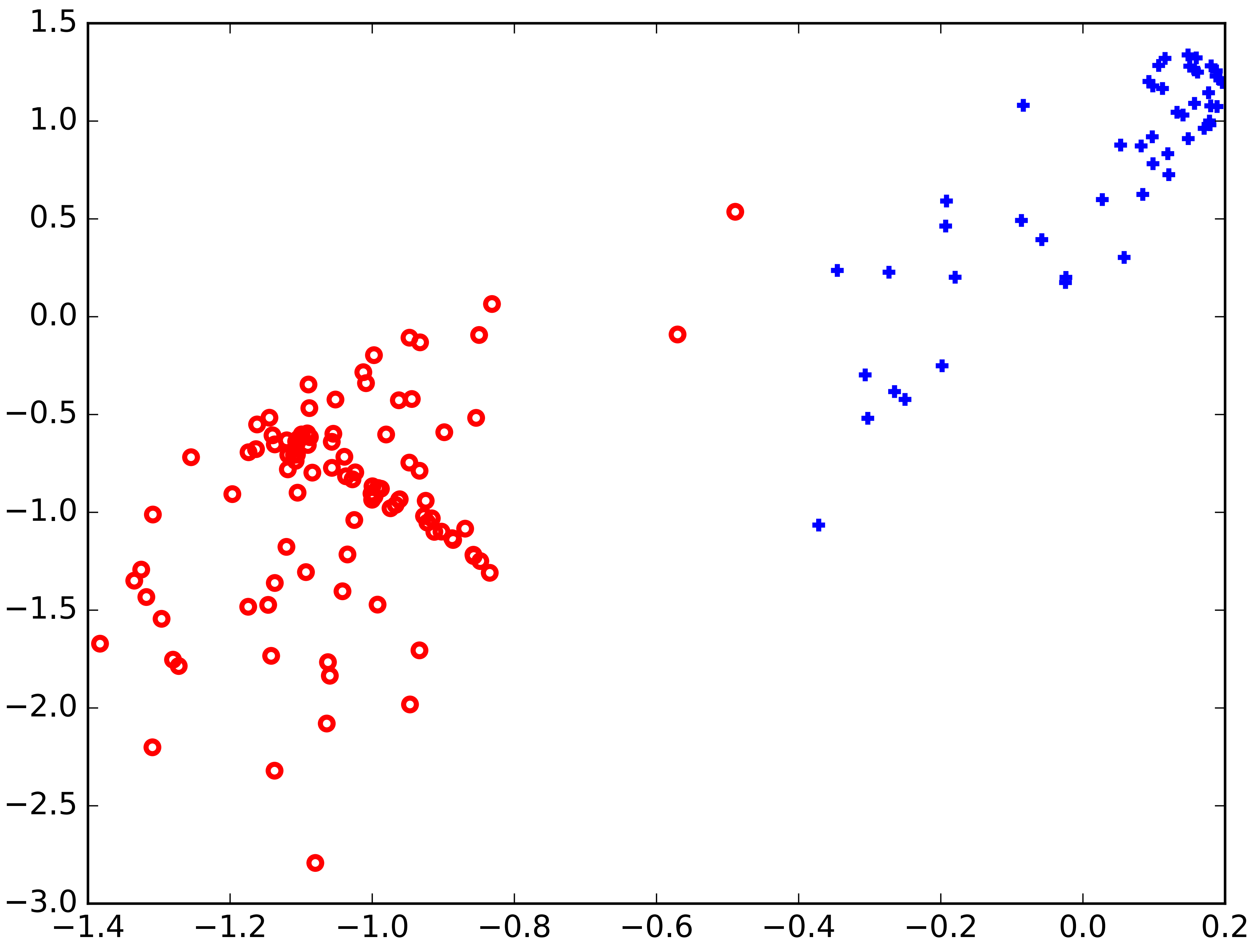
Recall: randomly initialized parameters

- A large number of randomly initialized hidden units gives a meaningful feature expansion

Hidden layer units



(10 randomly initialized units)



This is a 2d linear projection of the 10 dimensional features space (obtained how?)

Learning neural networks

- We can use stochastic gradient descent (SGD) to try to minimize the empirical risk (squared errors for regression, cross-entropy losses for classification, etc)

$$\frac{1}{N}\sum_{i=1}^{N} L(y^i, f(x^i;\theta)) \qquad +\lambda\|\theta\|^2$$

- In response to each randomly chosen data point, we update $\theta \leftarrow \theta - \eta \nabla_\theta L(y^i, f(x^i;\theta))$

Learning neural networks

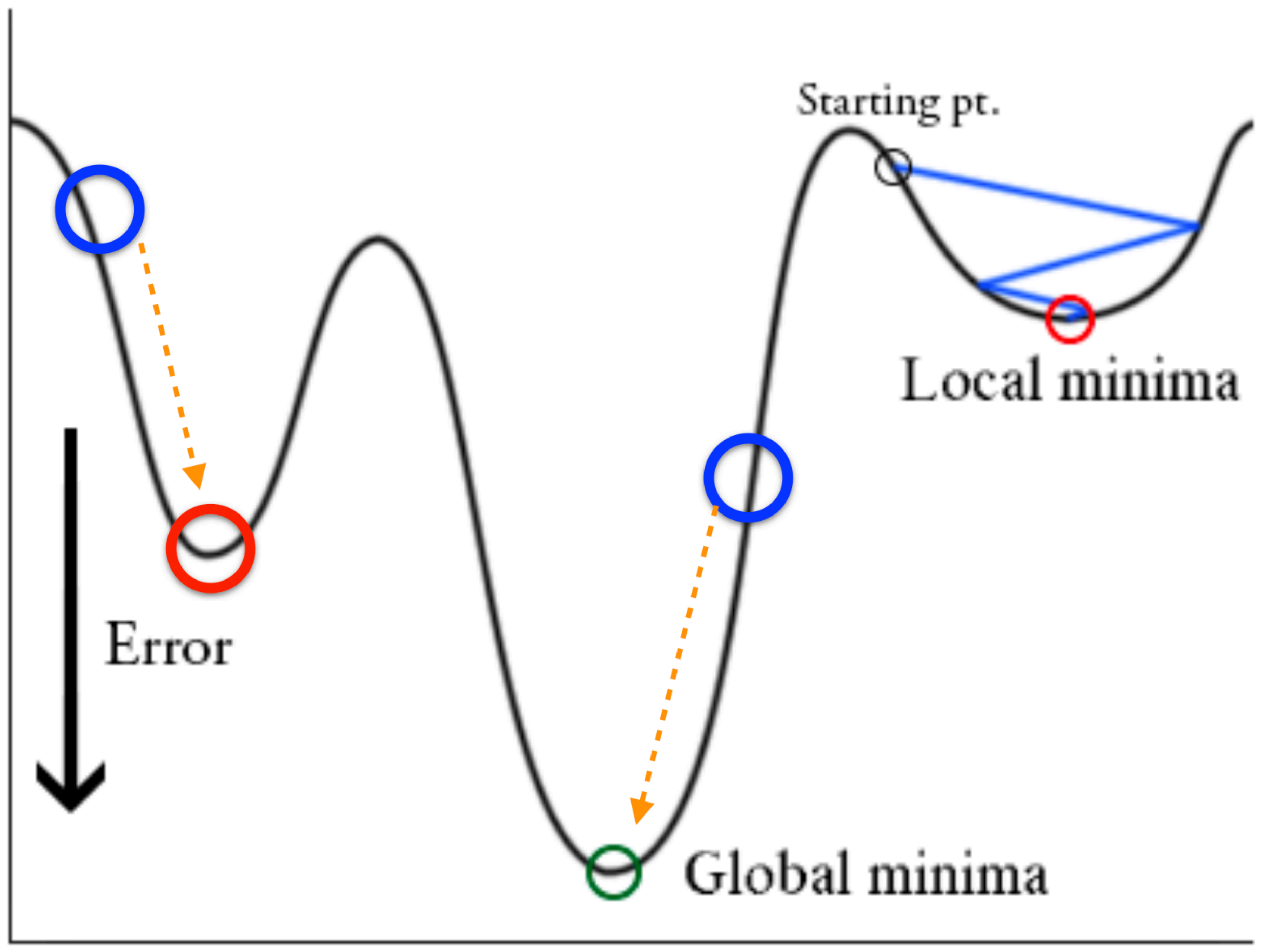
- We can use stochastic gradient descent (SGD) to try to minimize the empirical risk (squared errors for regression, cross-entropy losses for classification, etc)

$$\frac{1}{N}\sum_{i=1}^{N} L(y^i, f(x^i;\theta)) \qquad +\lambda\|\theta\|^2$$

- In response to each randomly chosen data point, we update $\theta \leftarrow \theta - \eta \nabla_\theta L(y^i, f(x^i;\theta))$

- The challenge is that the empirical risk / per example losses are no longer convex…

- Initialization matters! (and zero initialization is terrible!)



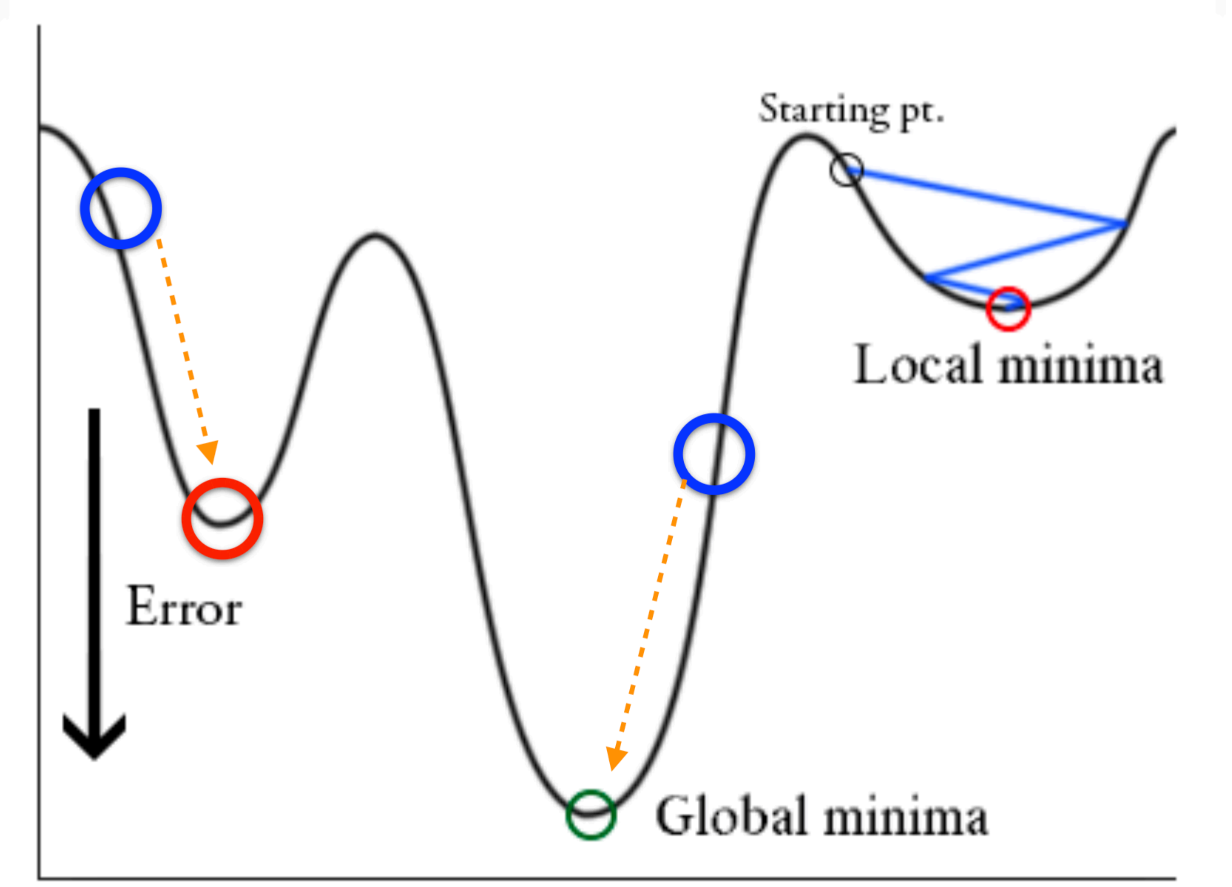
[figure from P. Agrawal]

Initialization, learning rate, loss landscapes

- There are many suggested initialization schemes for weight matrices, e.g., controlling fan-in variance: for a $d \times m$ matrix W where m is the output dimension, we could initialize $W_{ij} \sim N(0, \sigma^2 I),\ \sigma^2 = 1/d$ (e.g.)
- Many choices for learning rate schedules, often adaptive (e.g., Adam optimizer, etc). Optimization parameters are left as "hyper-parameters", to be adjusted based on validation performance

Initialization, learning rate, loss landscapes

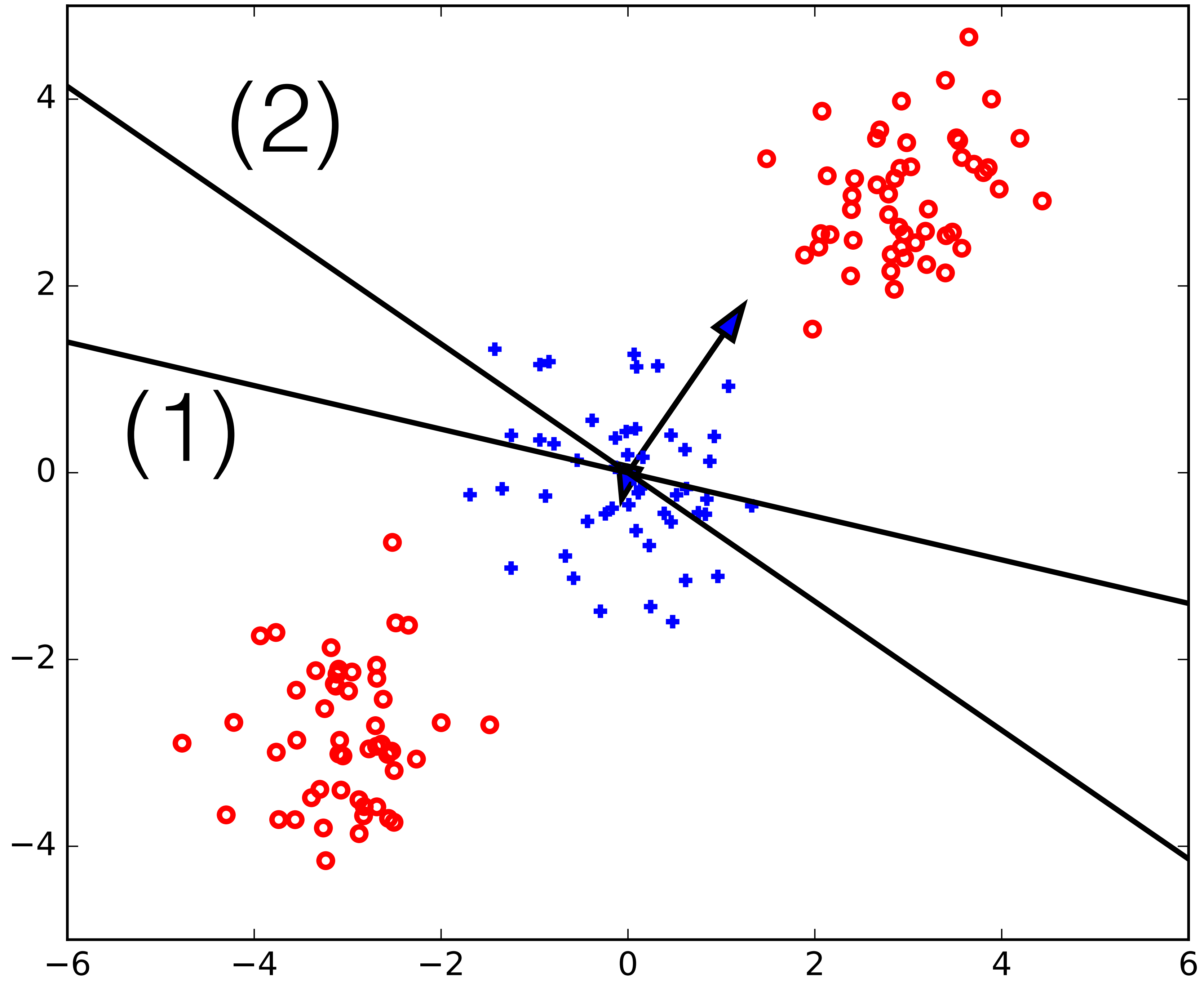
- There are many suggested initialization schemes for weight matrices, e.g., controlling fan-in variance: for a $d \times m$ matrix W where m is the output dimension, we could initialize $W_{ij} \sim N(0,\sigma^2 I),\ \ \sigma^2 = 1/d$ (e.g.)
- Many choices for learning rate schedules, often adaptive (e.g., Adam optimizer, etc). Optimization parameters are left as "hyper-parameters", to be adjusted based on validation performance
- A simple multi-layer perceptron already has a large number of equivalent solutions in terms of weight matrices
  - e.g., we can permute nodes in each hidden layer while keeping the associated weights connected; the matrices would change as a result but the overall mapping would not
- Aspects of the high dimensional loss landscape are not well captured by these low dimensional figures
  - E.g., local minima obtained with different initializations may be "connected" via low loss paths



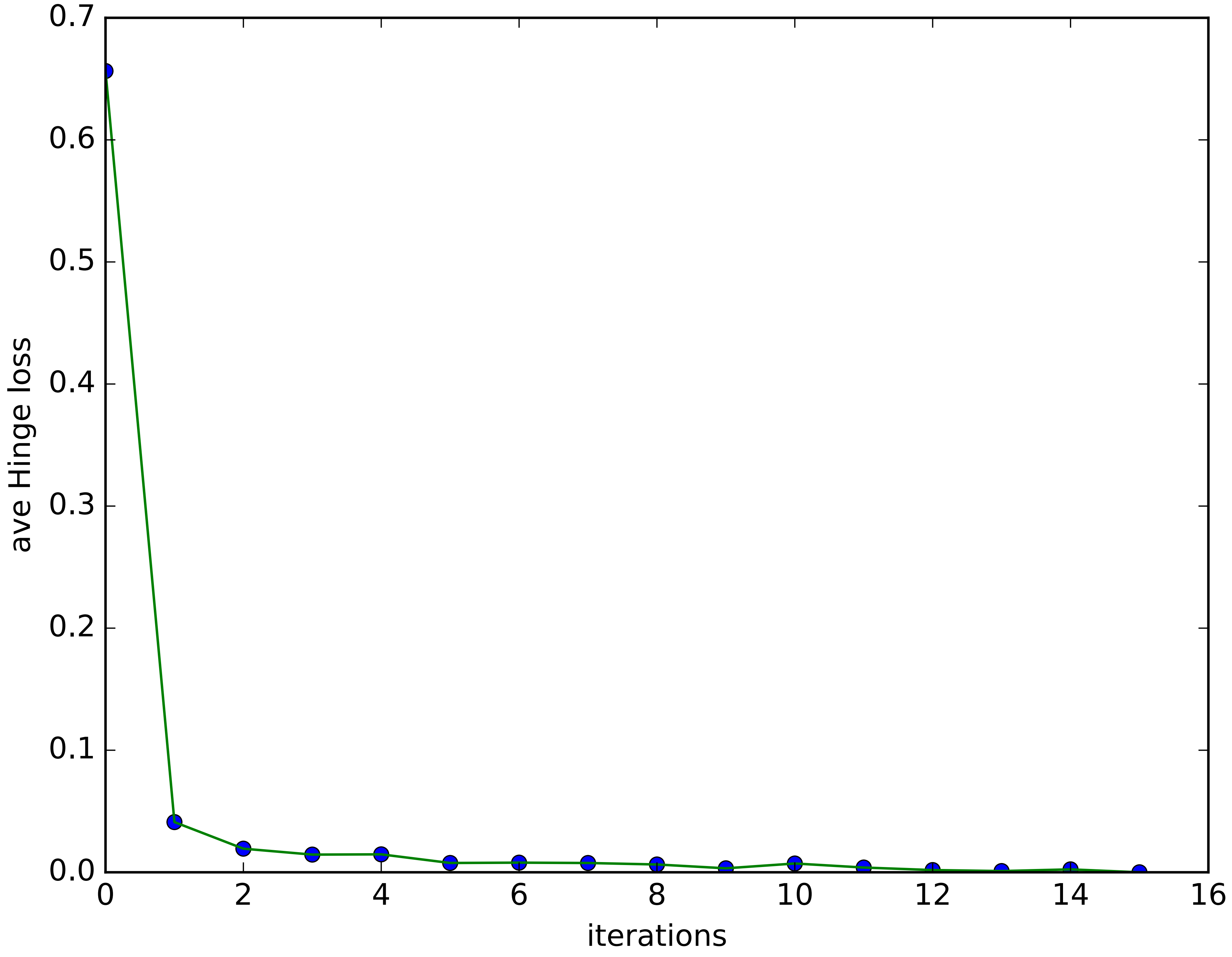
2 hidden units: training

- An epoch = one pass through the examples (random order)

Initial network (hidden units)



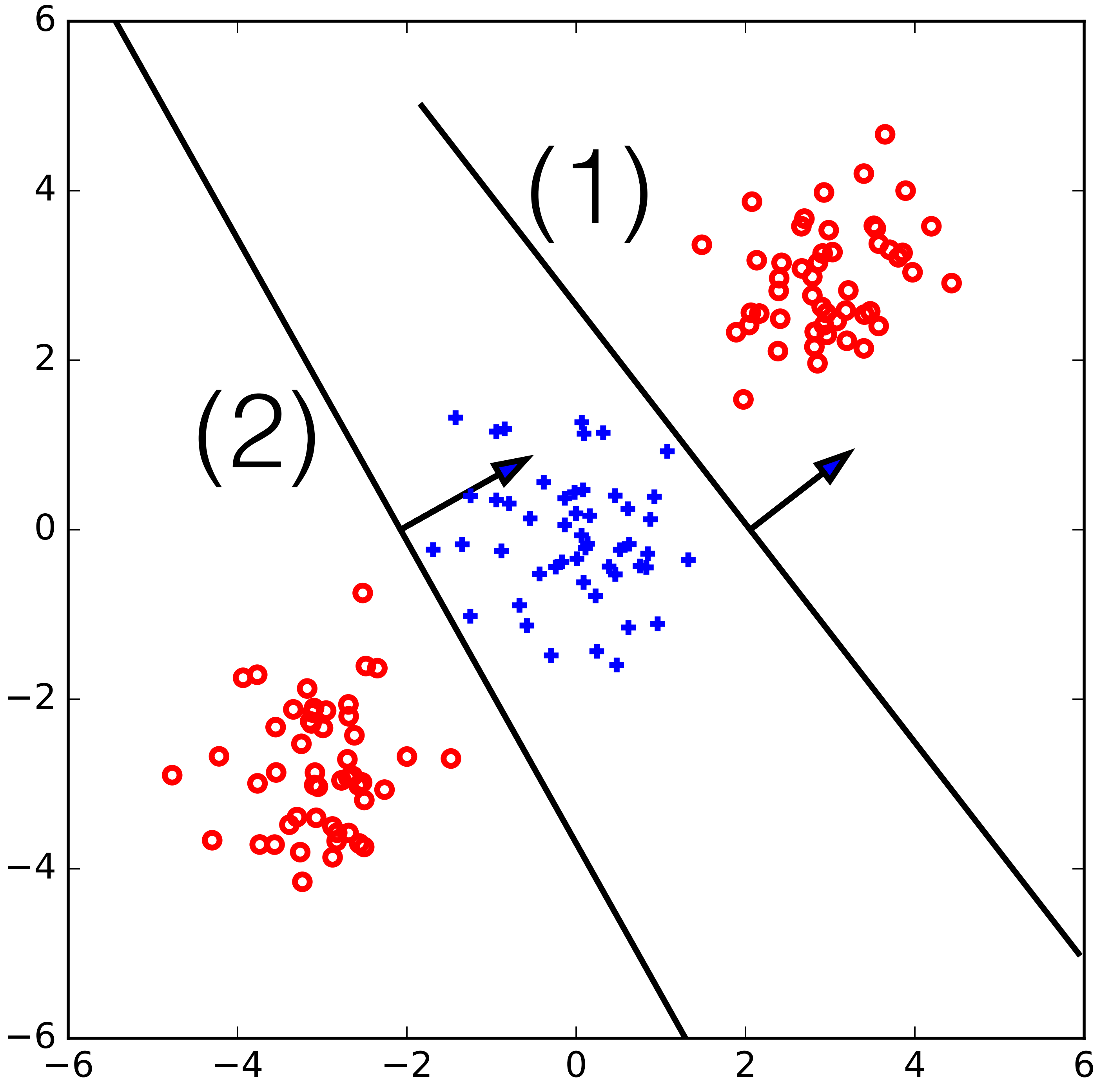
Average loss per epoch

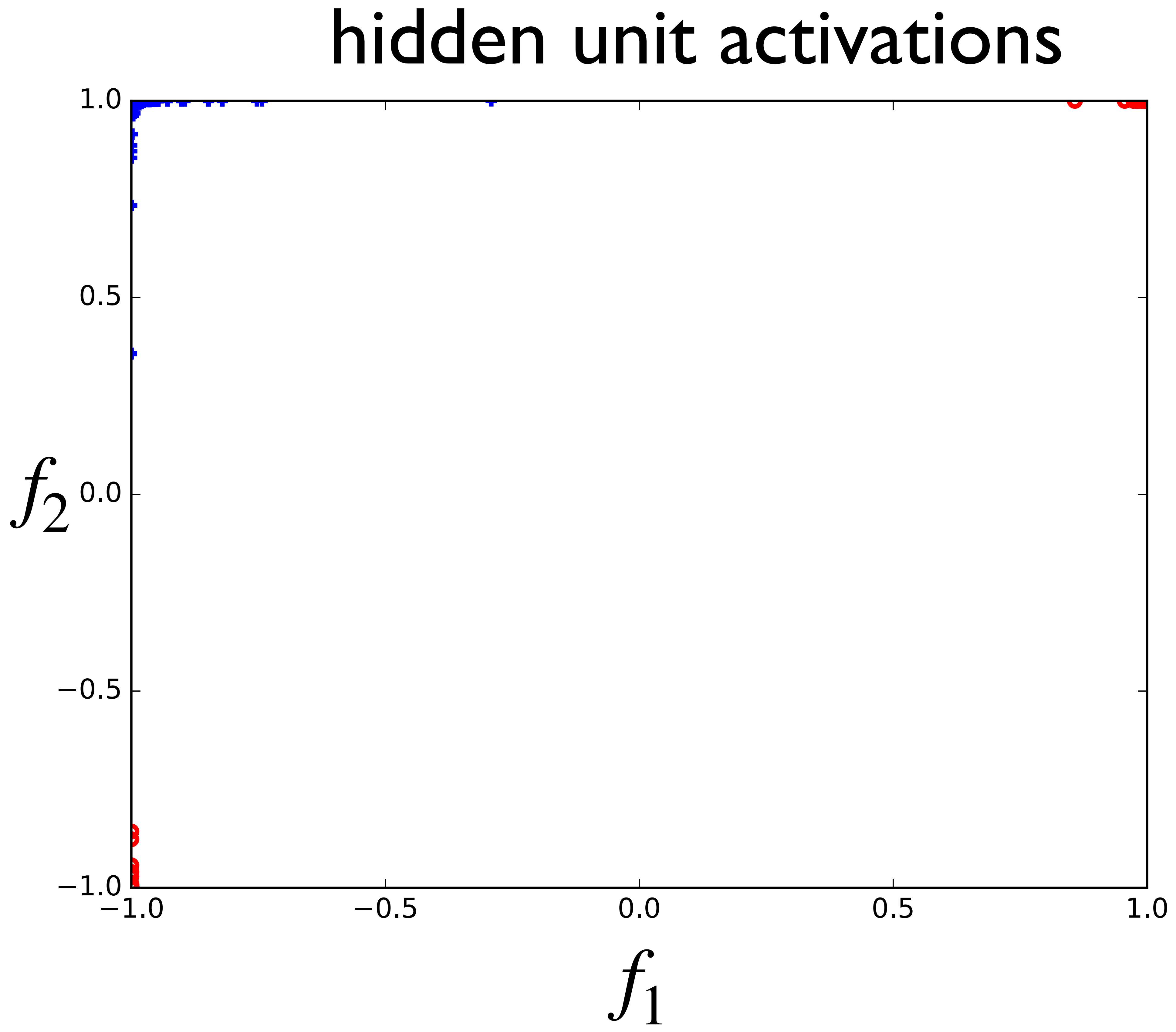


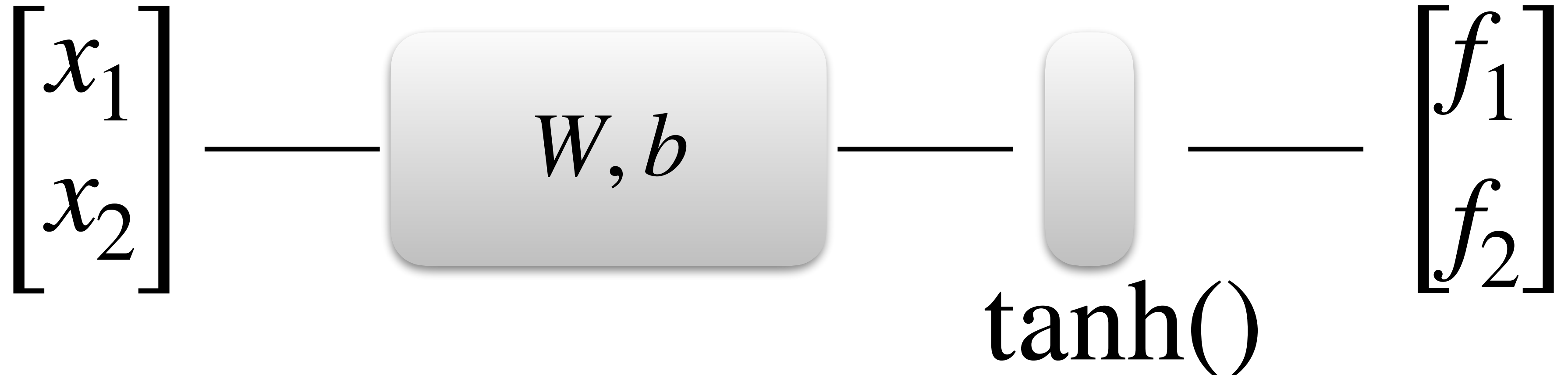
(full disclosure: loss here was hinge loss $\max\{0,\ 1 - yf(x;\theta)\},\ \ y \in \{-1,1\}$)

2 hidden units: training

- After ~10 passes through the data

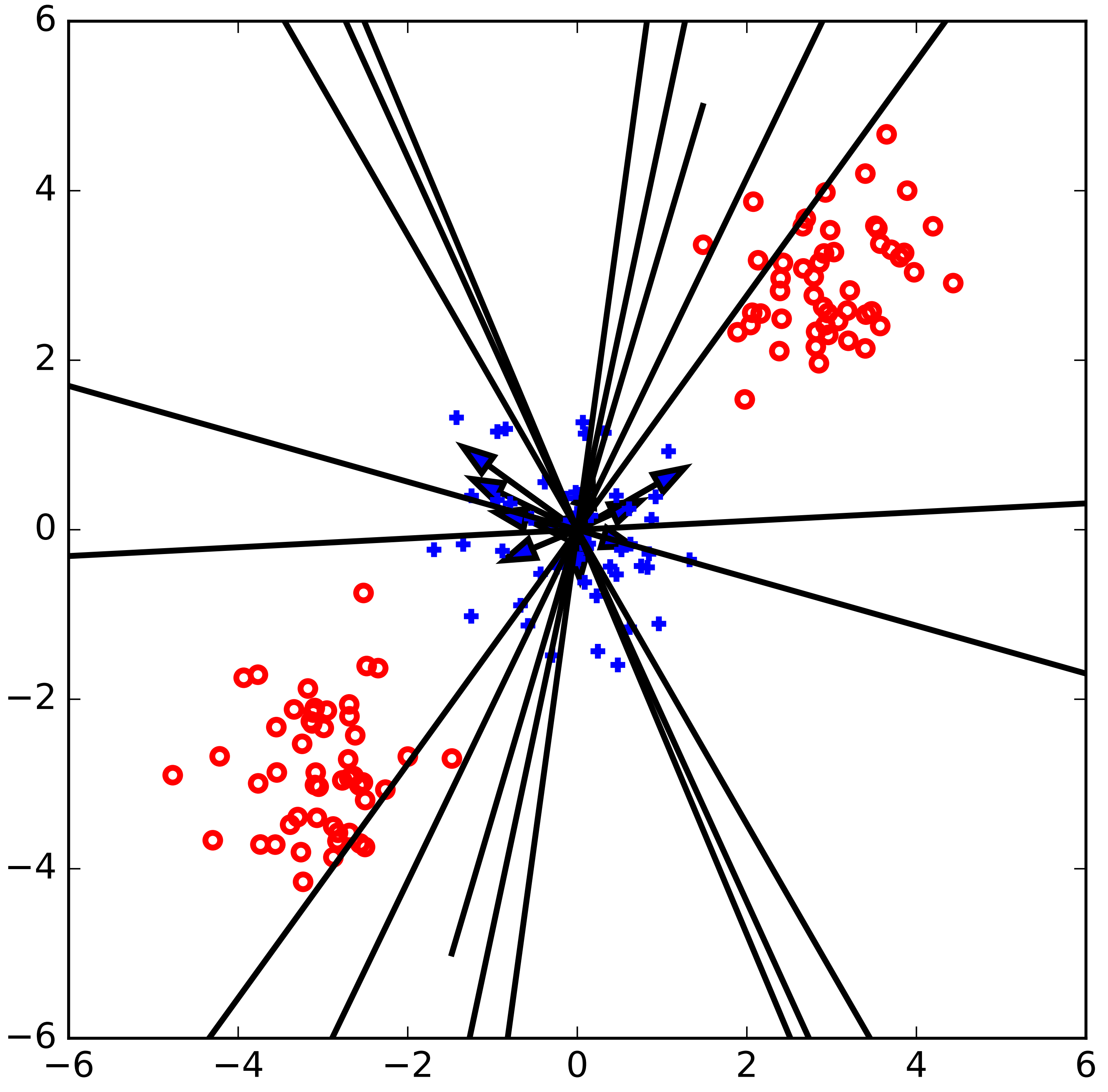






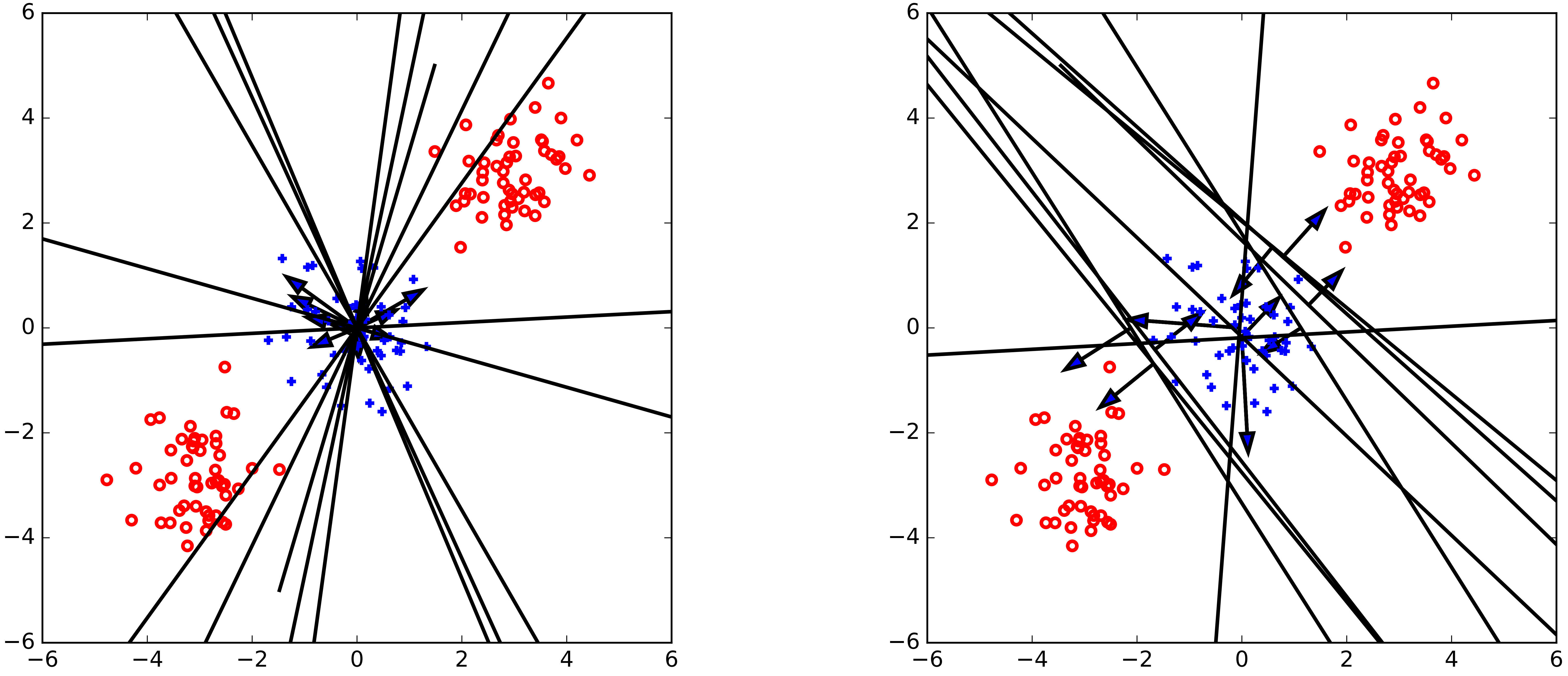
10 hidden units

- Randomly initialized weights (zero offset) for the hidden units



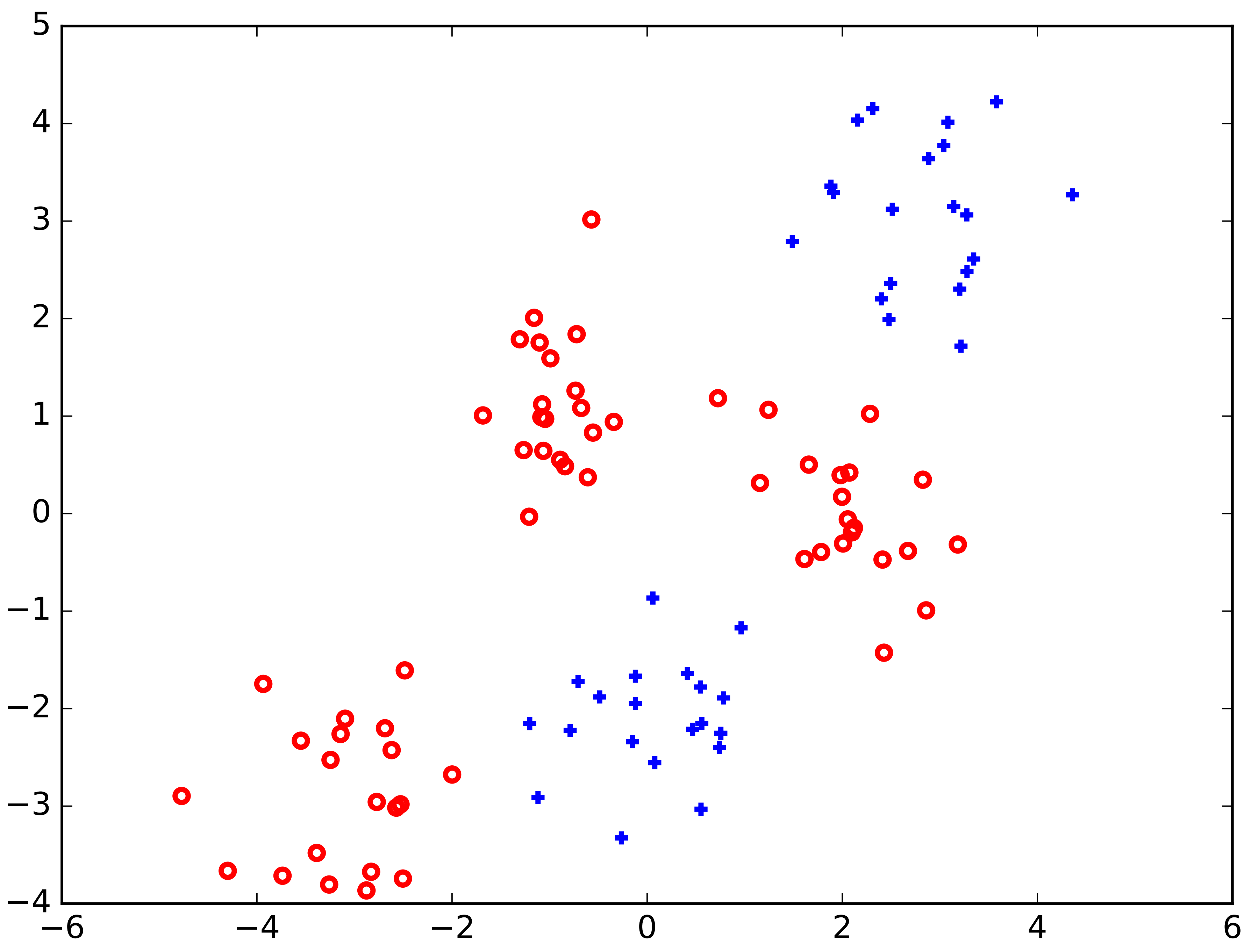
10 hidden units

- After ~ 10 epochs the hidden units are arranged in a manner sufficient for the task (but not otherwise perfect)



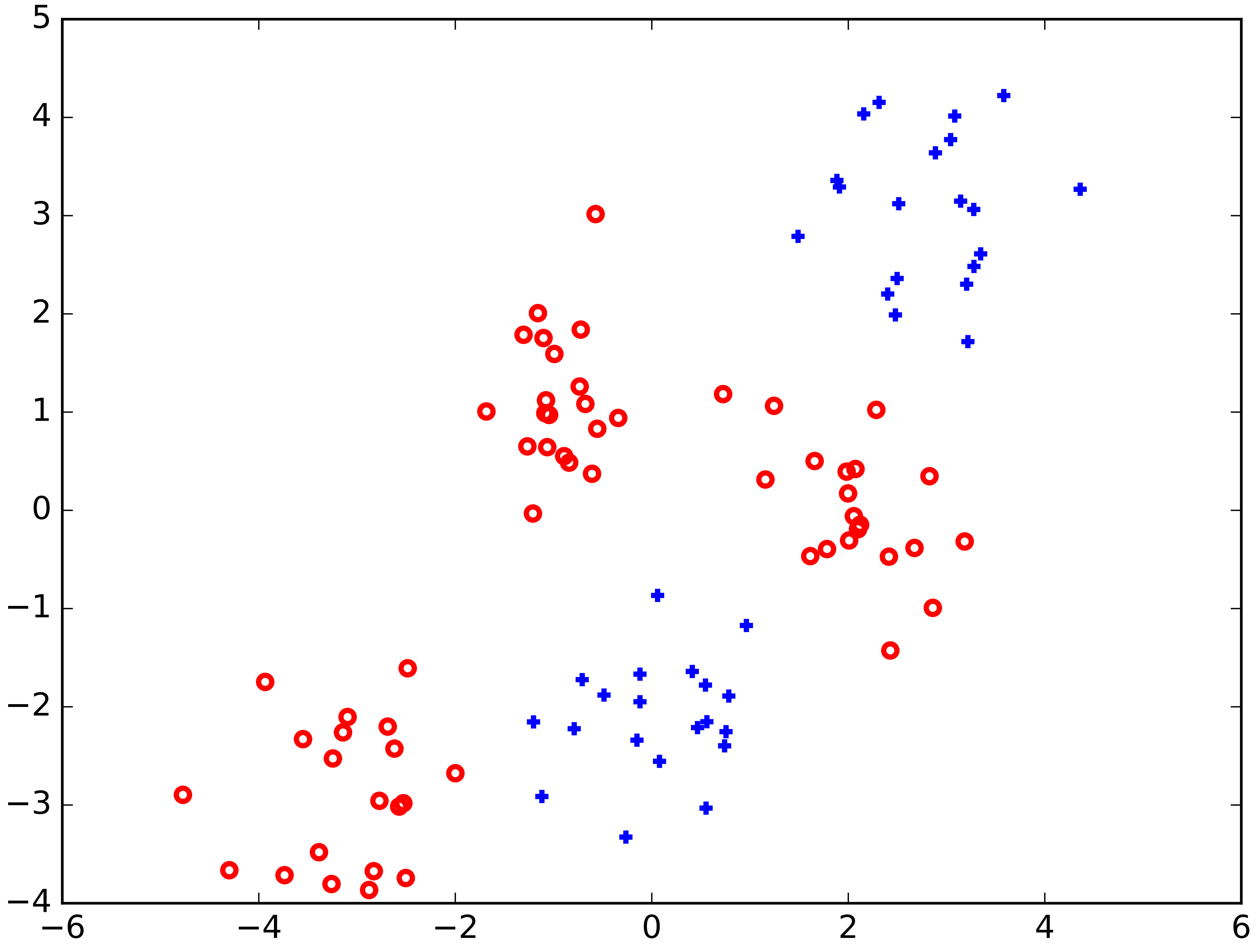
Decisions (and a harder task)

- 2 hidden units can no longer solve this task

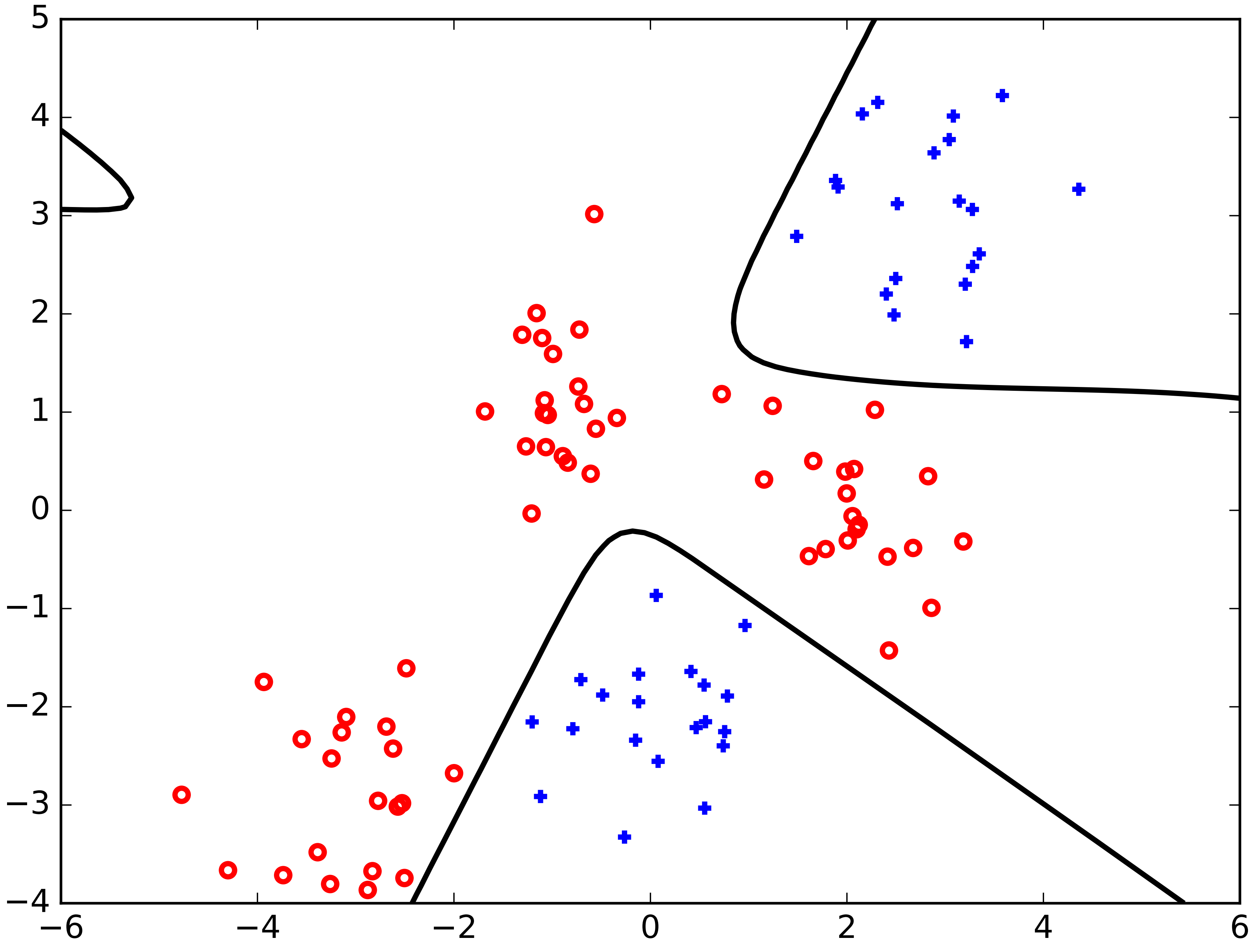


Decisions (and a harder task)

- 2 hidden units can no longer solve this task

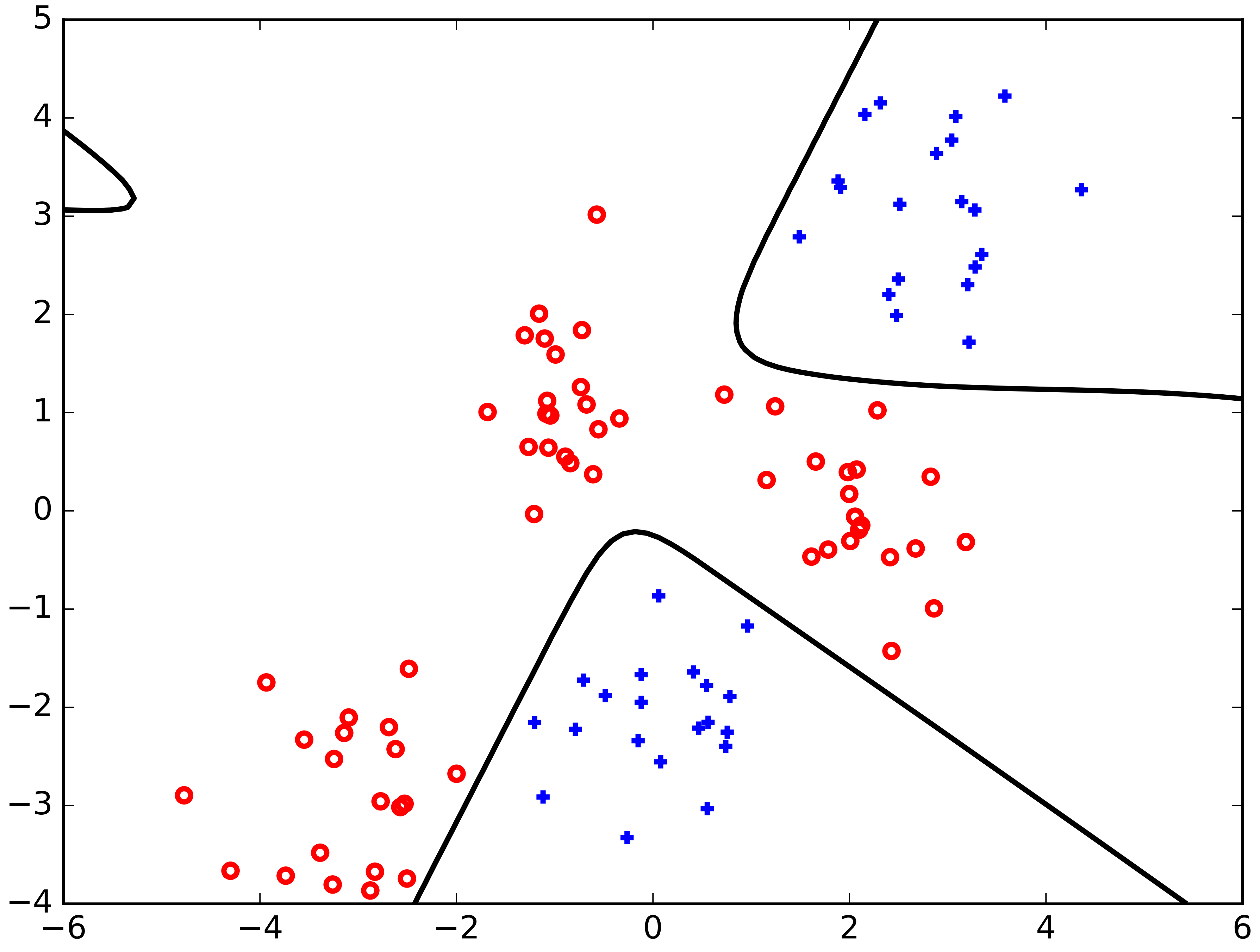


10 hidden units

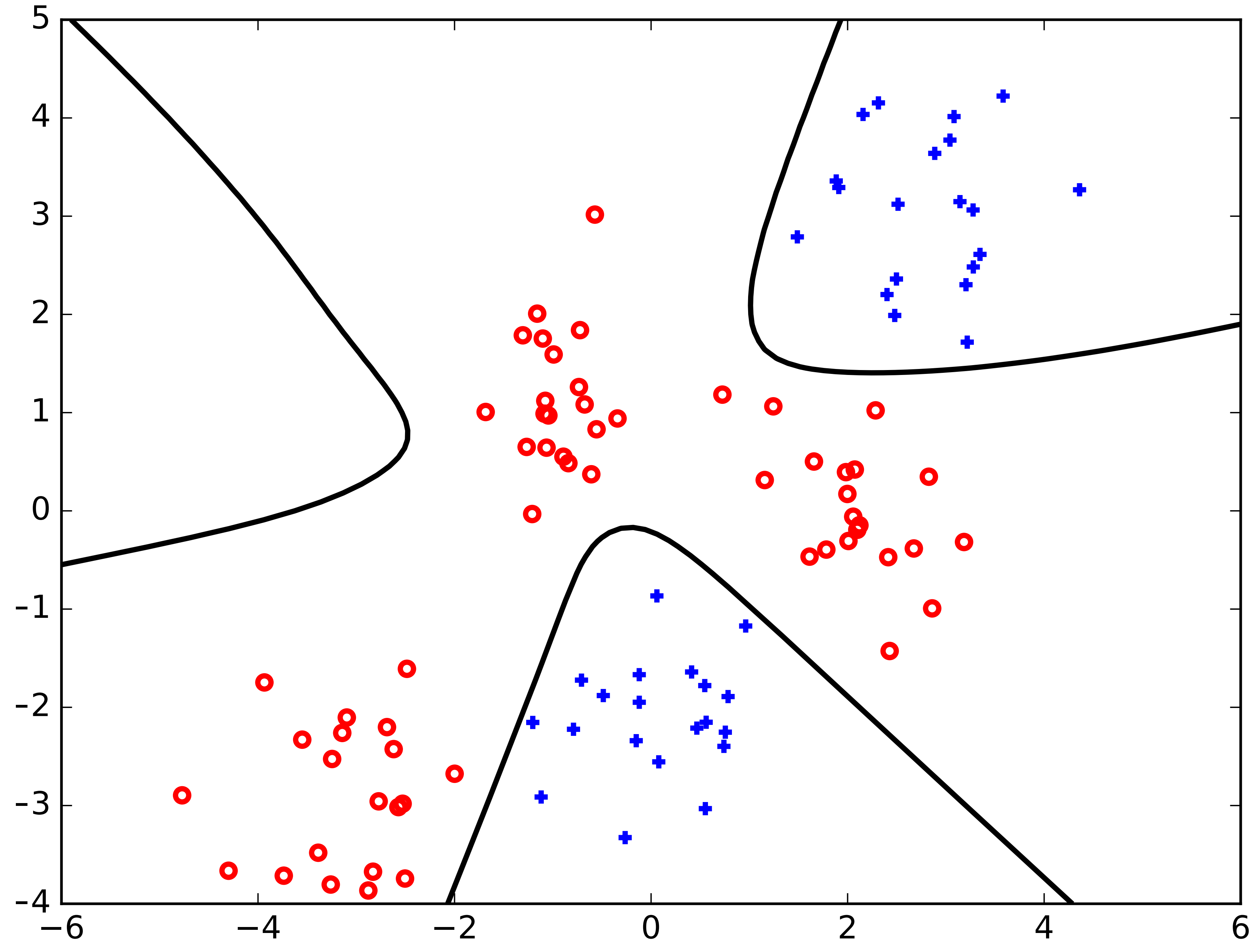


Decisions (and a harder task)

10 hidden units



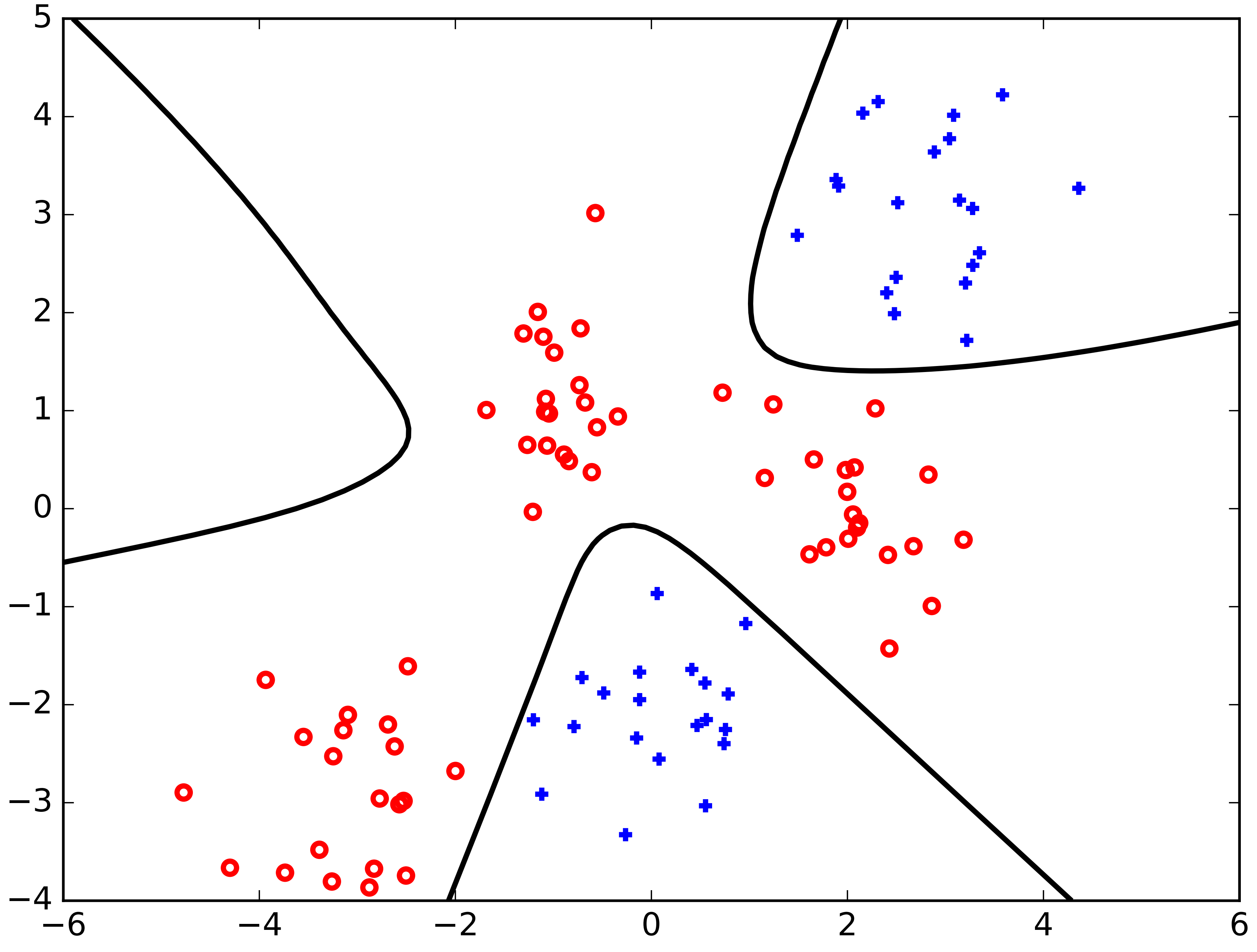
100 hidden units



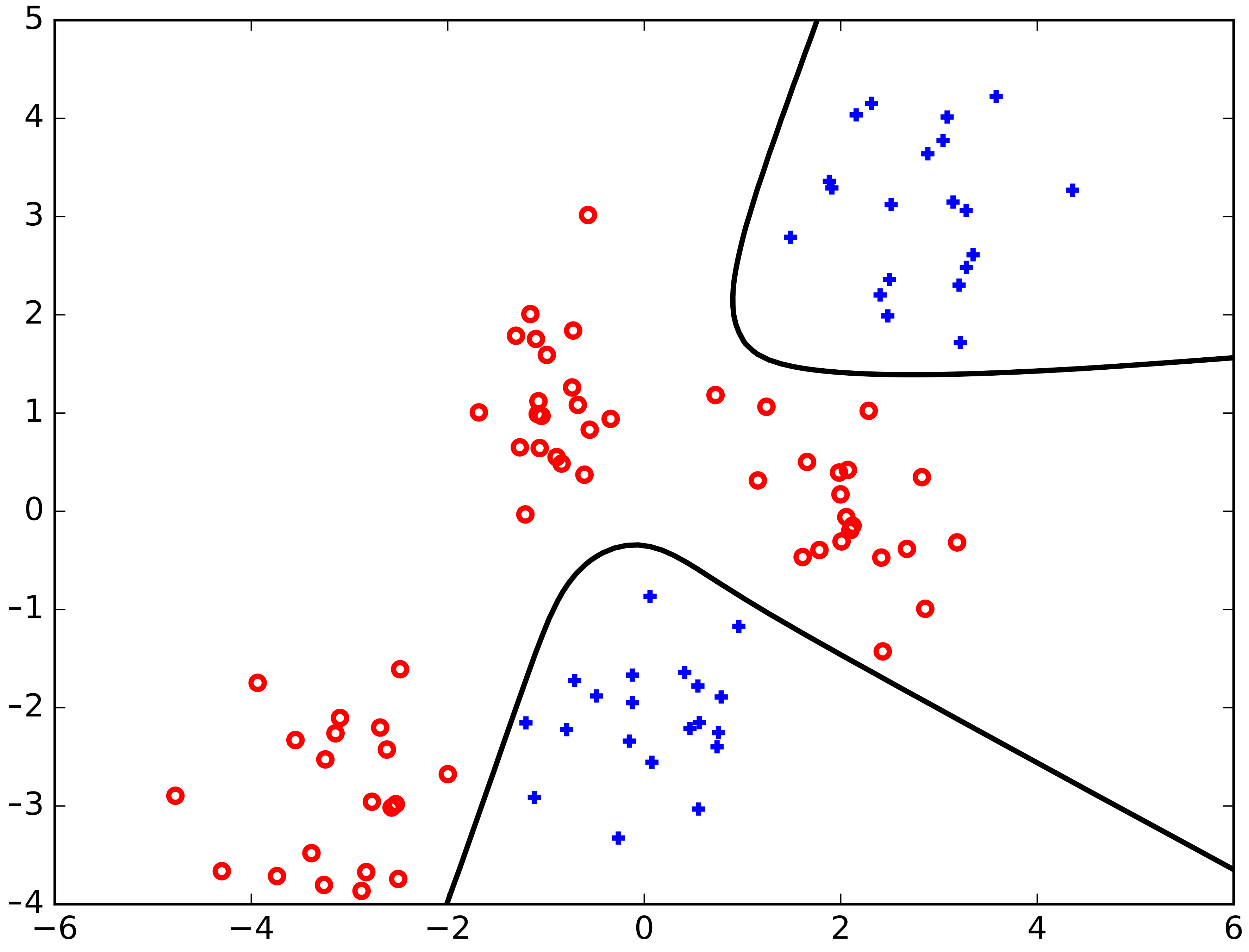
Decision boundaries

- Effects of initialization can persist… good initialization is important

100 hidden units
(with zero offset initialization)



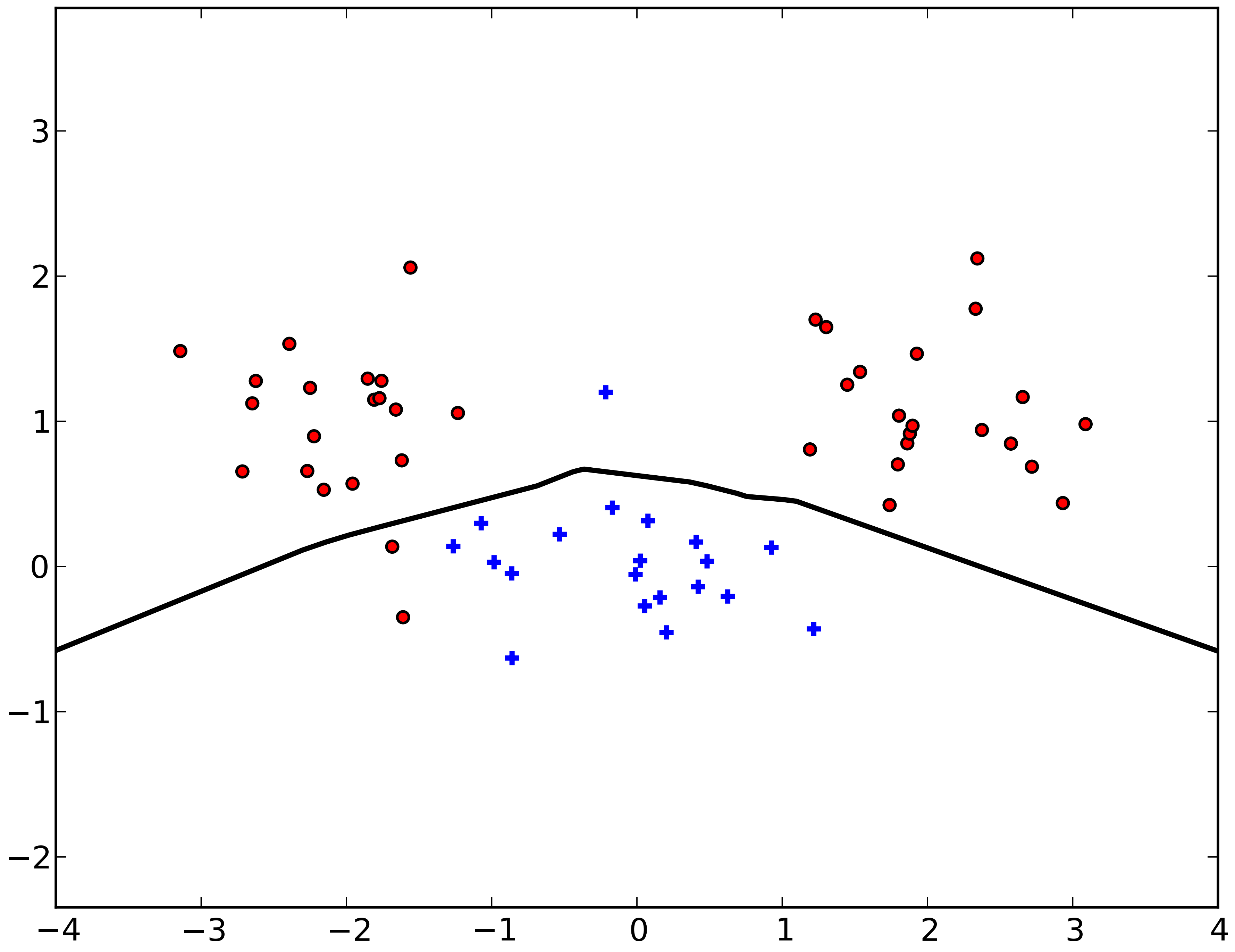
100 hidden units
(with random offset initialization)



Size, optimization

- Many recent architectures use ReLU units (cheap to evaluate, sparsity)
- Easier to learn as large models…

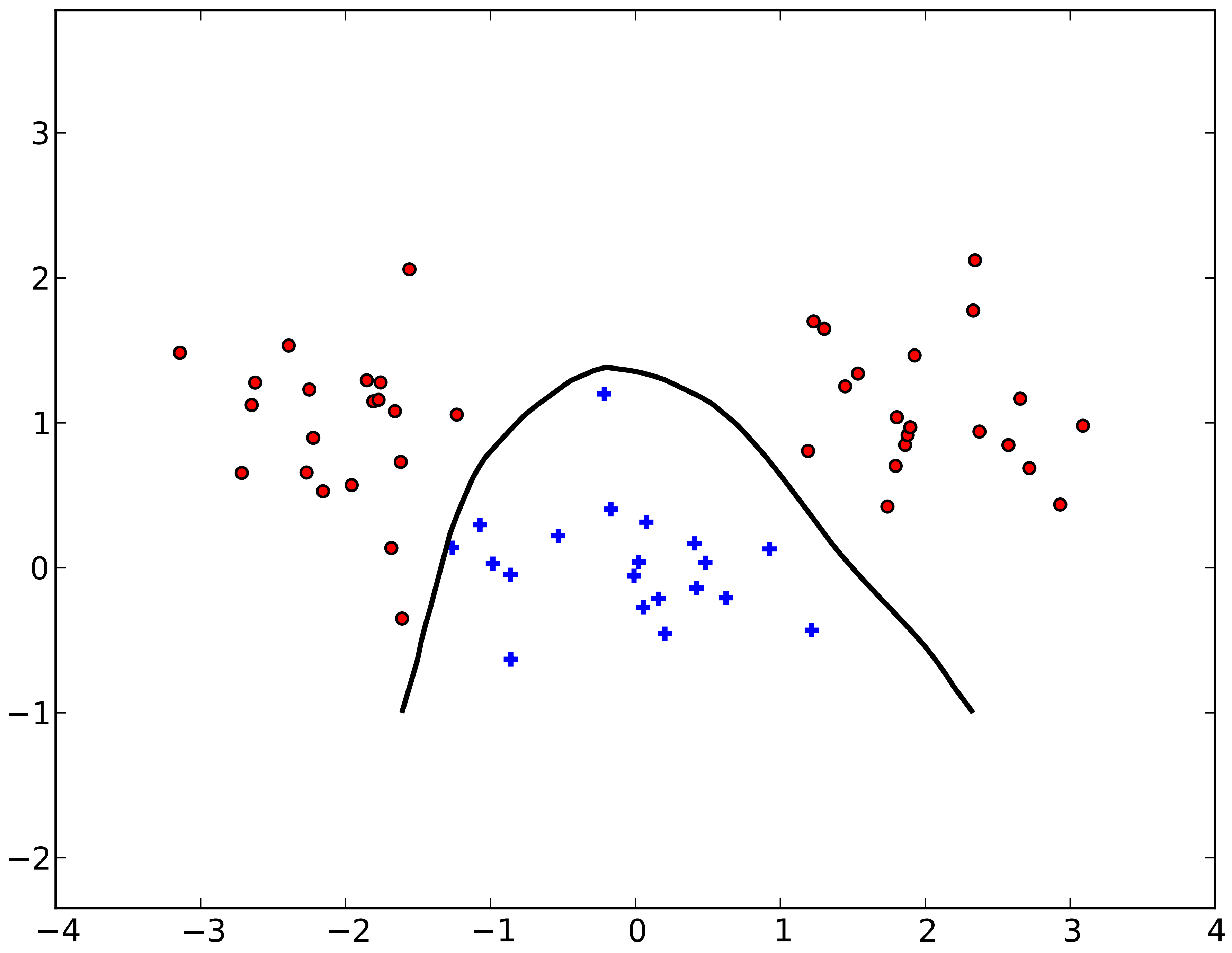
10 hidden units (should be sufficient but hard to find a good solution)



Size, optimization

- Many recent architectures use ReLU units (cheap to evaluate, sparsity)
- Easier to learn as large models…

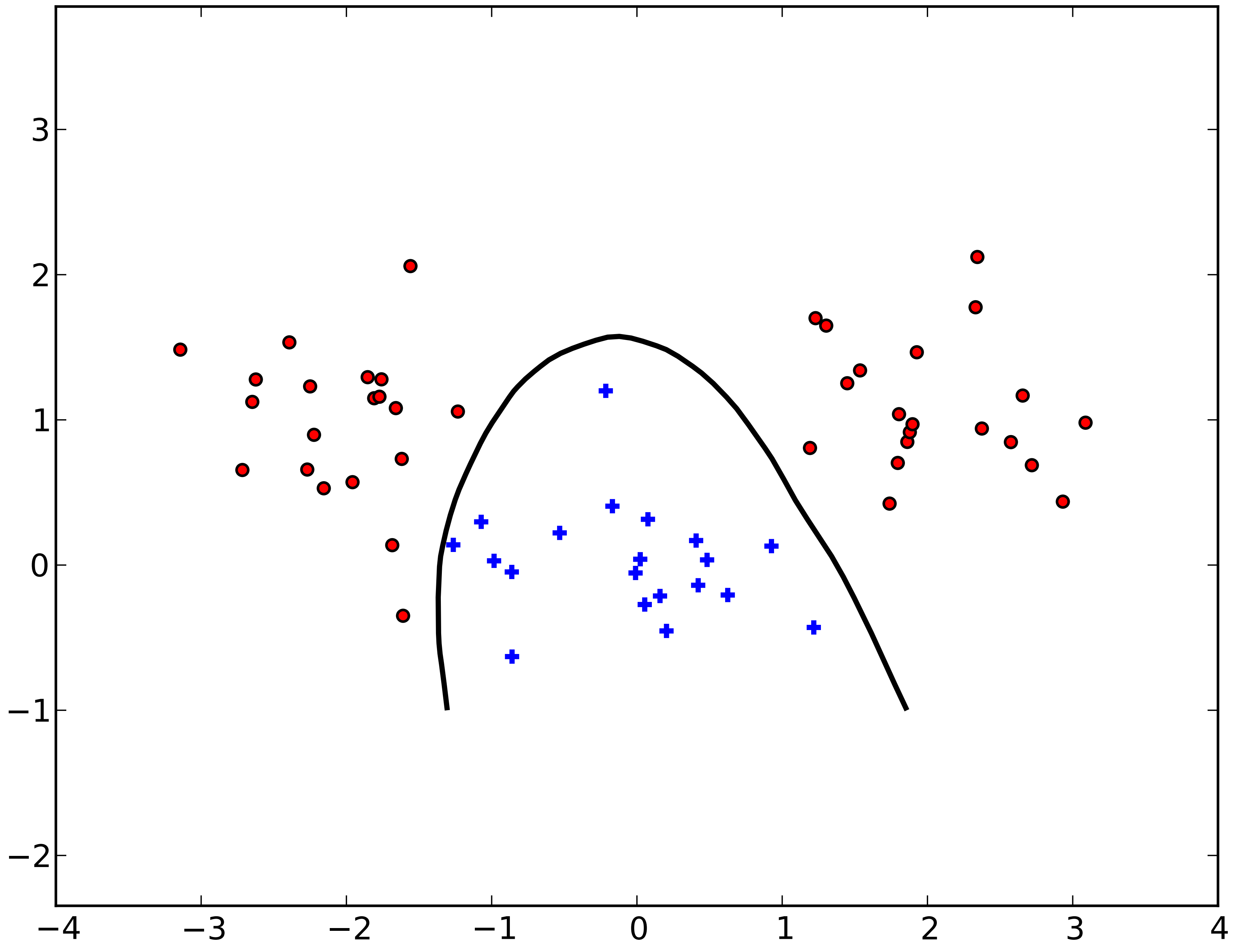
100 hidden units (substantial overcapacity)



Size, optimization

- Many recent architectures use ReLU units (cheap to evaluate, sparsity)
- Easier to learn as large models…

500 hidden units (substantial overcapacity)



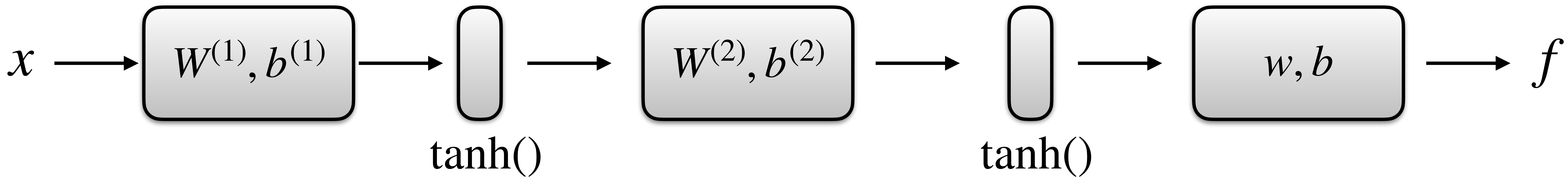
Computation graph, backpropagation

- The remaining question is how we actually evaluate the gradient with respect to all the parameters for a complicated model

$$\theta \leftarrow \theta - \eta \nabla_\theta L(y^i, f(x^i;\theta))$$

- where, e.g.,

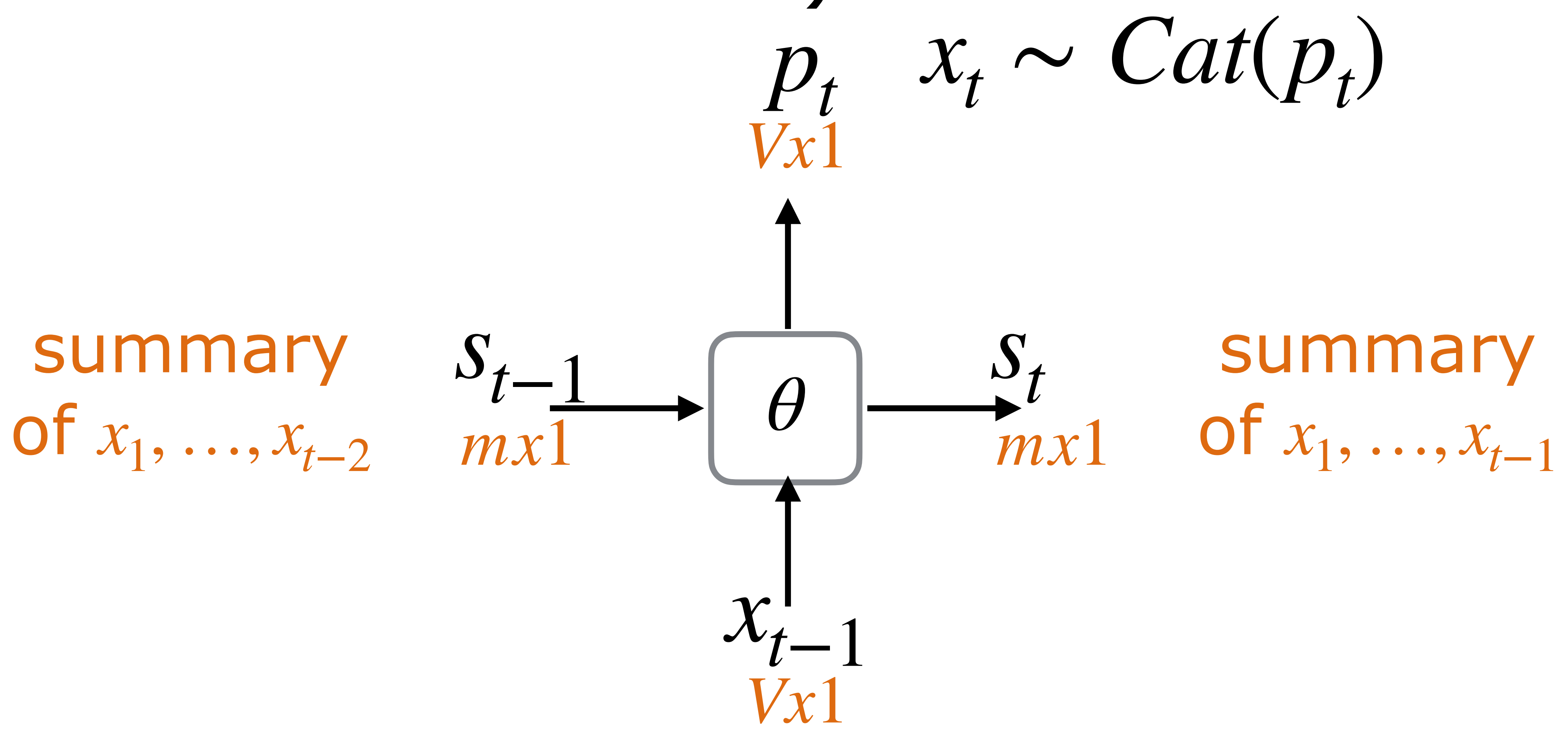
$$\underset{1x1}{f(x;\theta)} = \underset{1xk}{w^T} \underset{kx1}{\tanh}\Big(\underset{kxm}{W^{(2)}} \underset{mx1}{\tanh}(\underset{mxd}{W^{(1)}} x + \underset{mx1}{b^{(1)}}) + \underset{kx1}{b^{(2)}}\Big) + b \qquad \theta = \{w, b, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$$



- We'll explain this in the context of a recurrent neural network (RNN) and its associated "computation graph"

Recurrent neural network (RNN)

- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

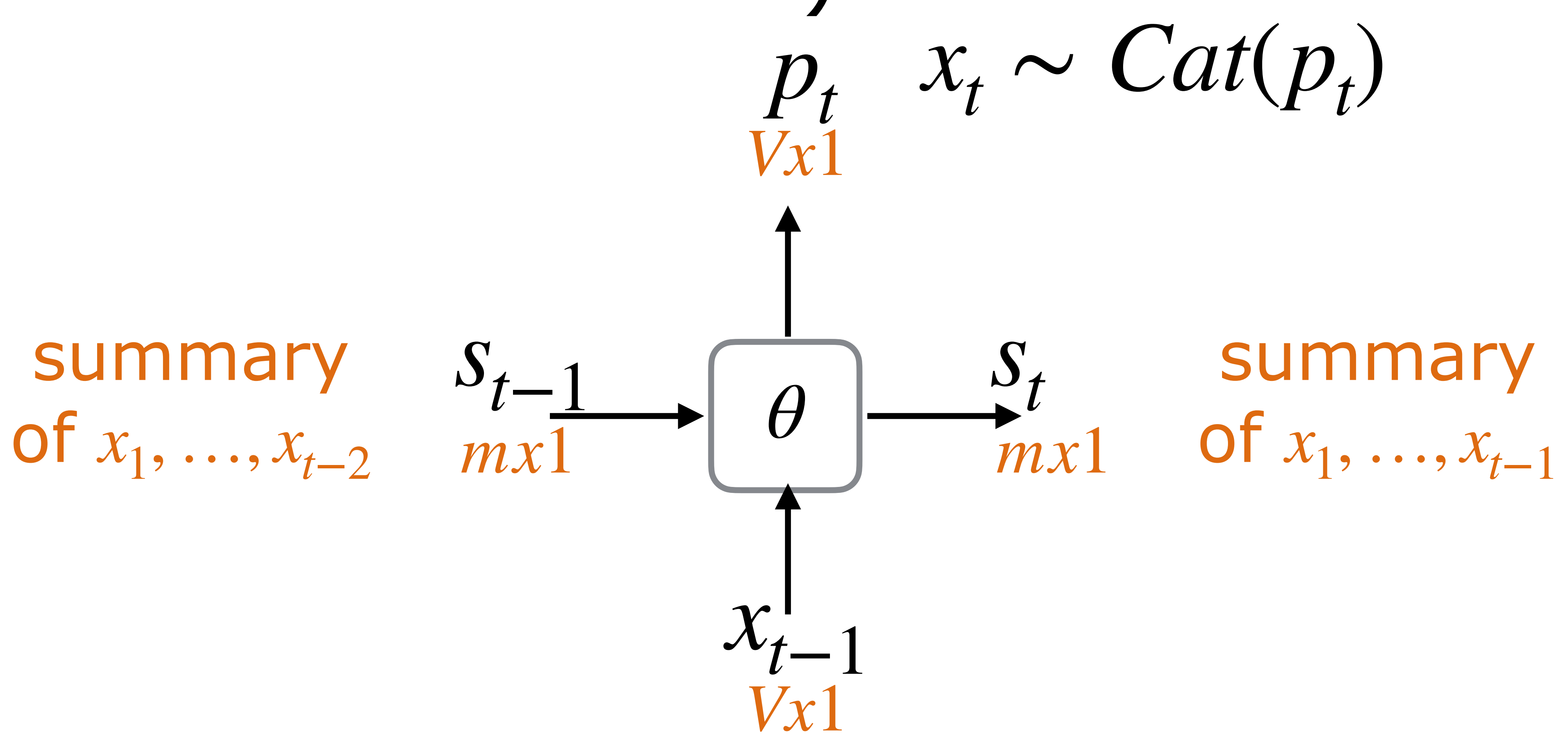


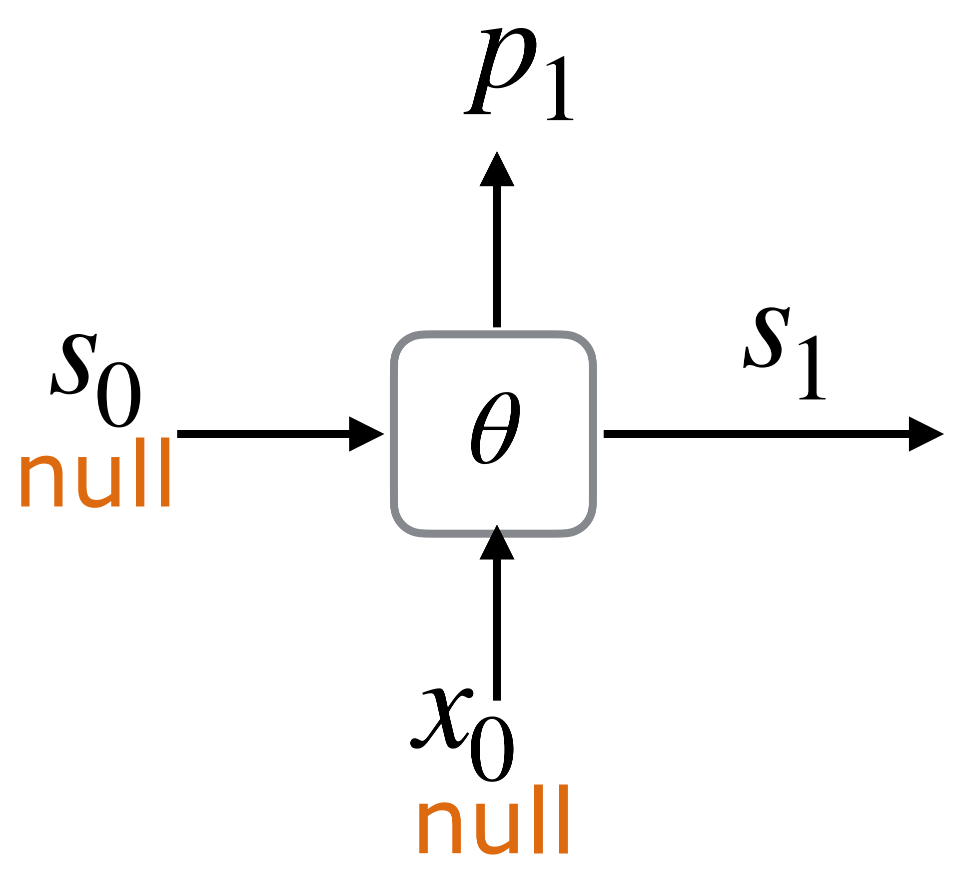
$$P(x_t \mid x_{t-1}, \ldots, x_1) = P(x_t \mid x_{t-1}, s_{t-1}) = P(x_t \mid s_t)$$

- V = vocabulary size
- m = state (summary) dimension

Recurrent neural network (RNN)

- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

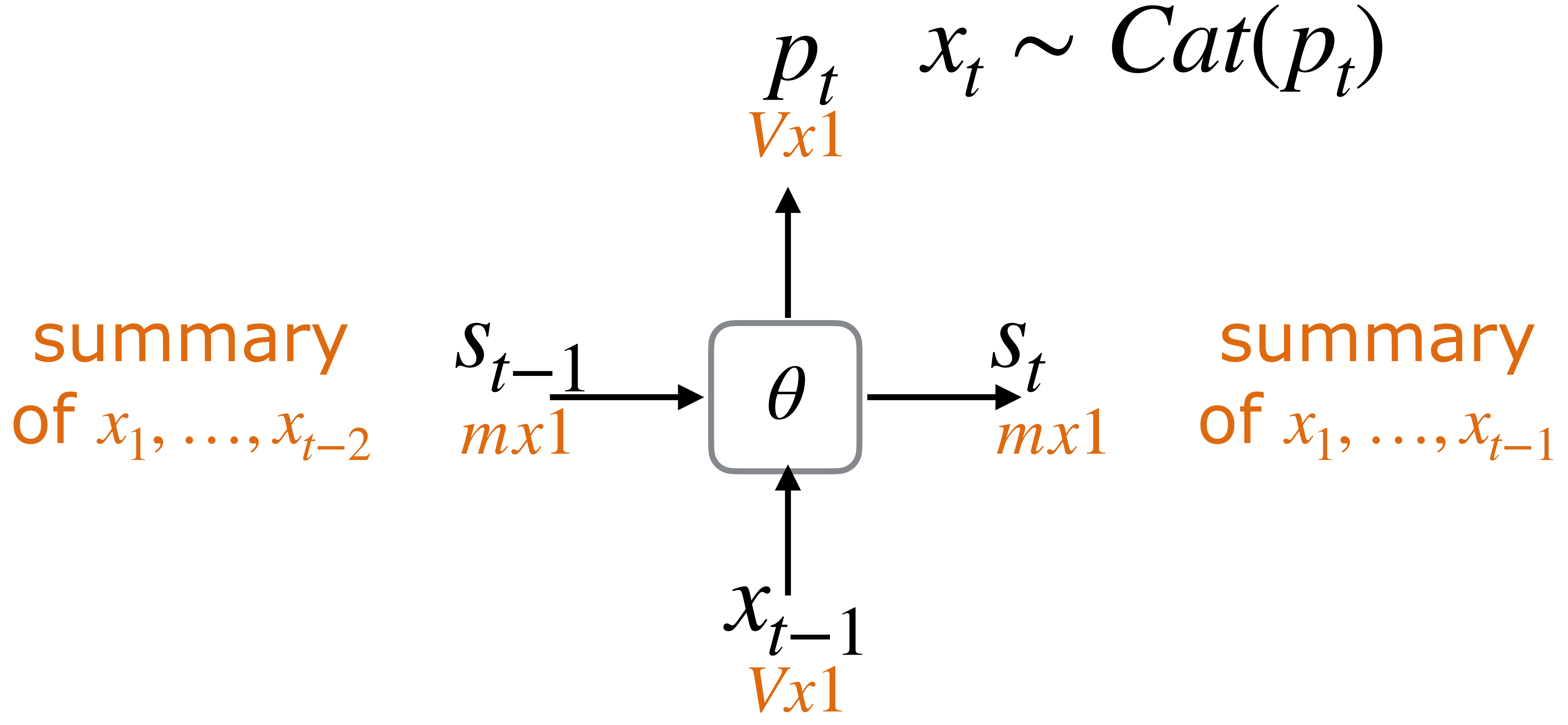


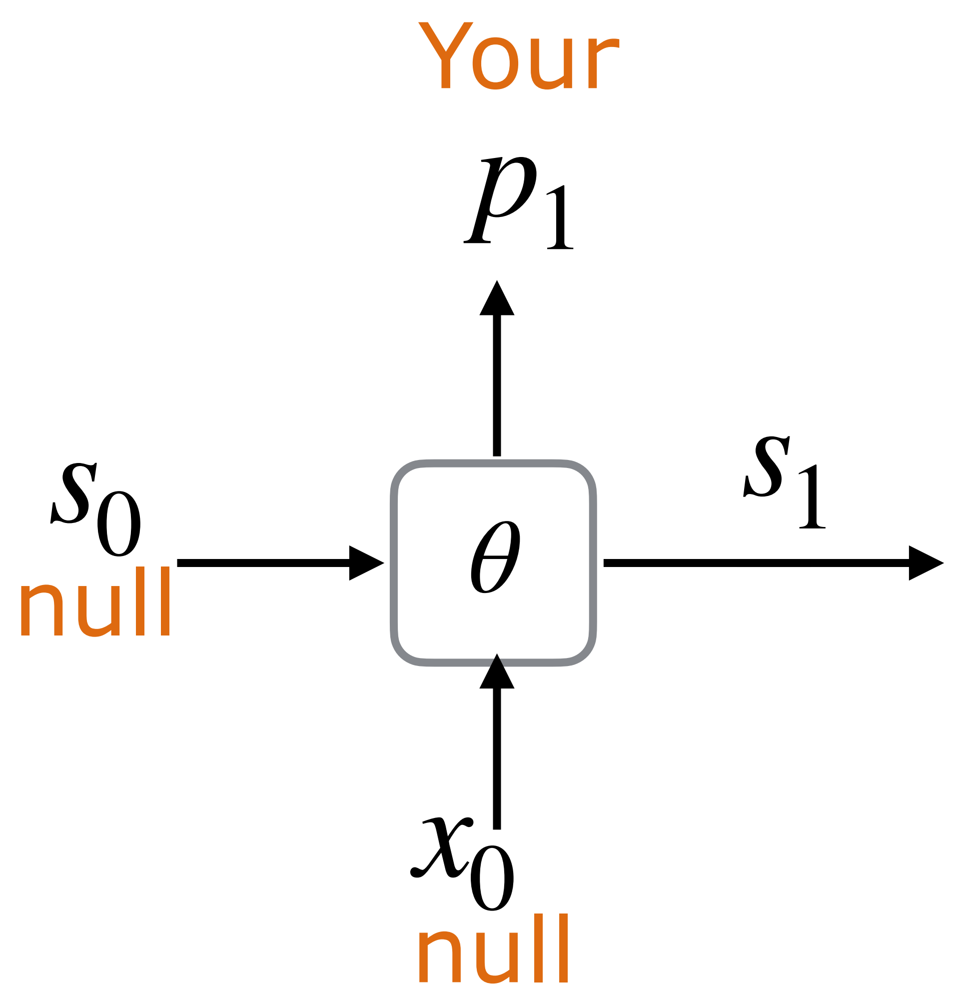


Recurrent neural network (RNN)

- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

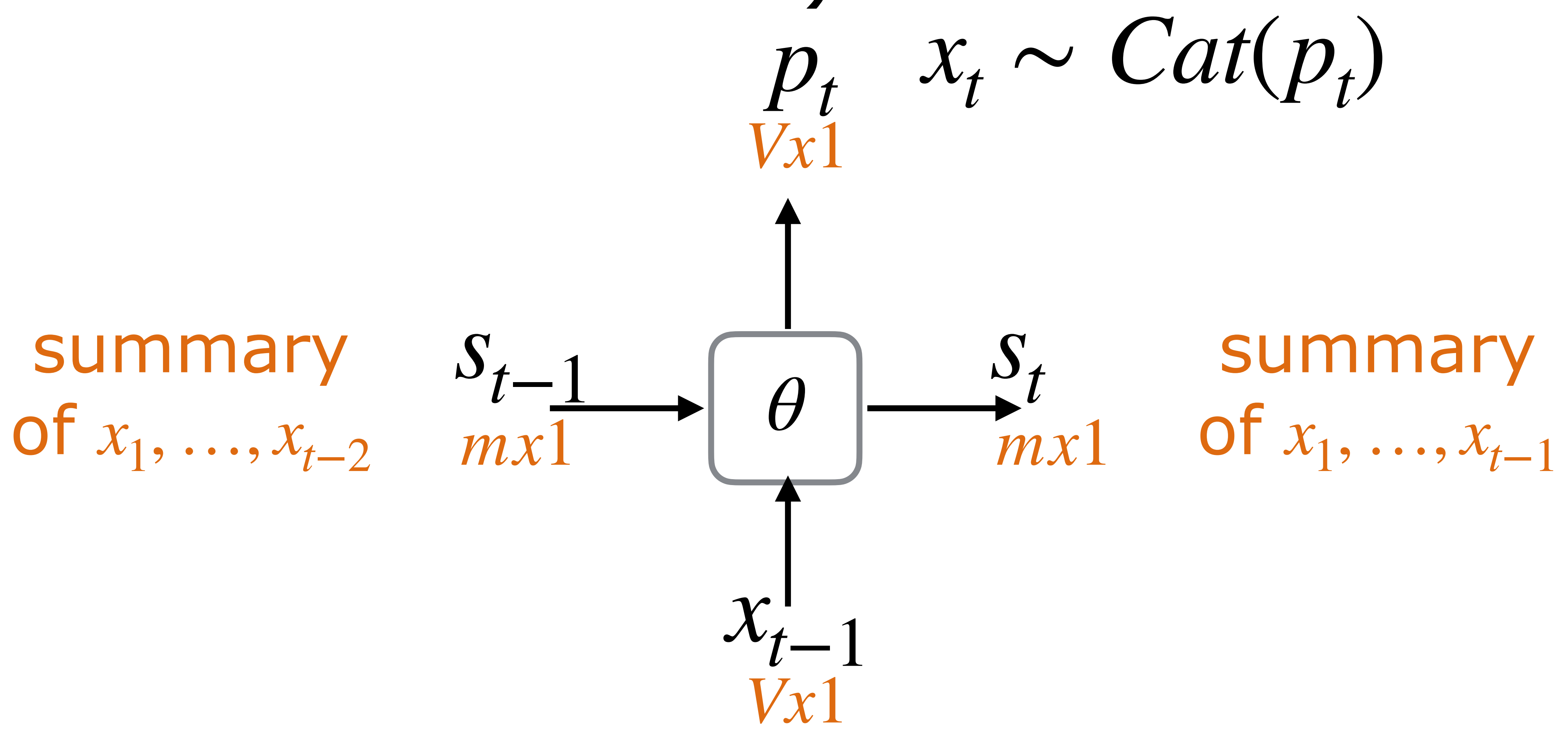
$x_t \sim Cat(p_t)$

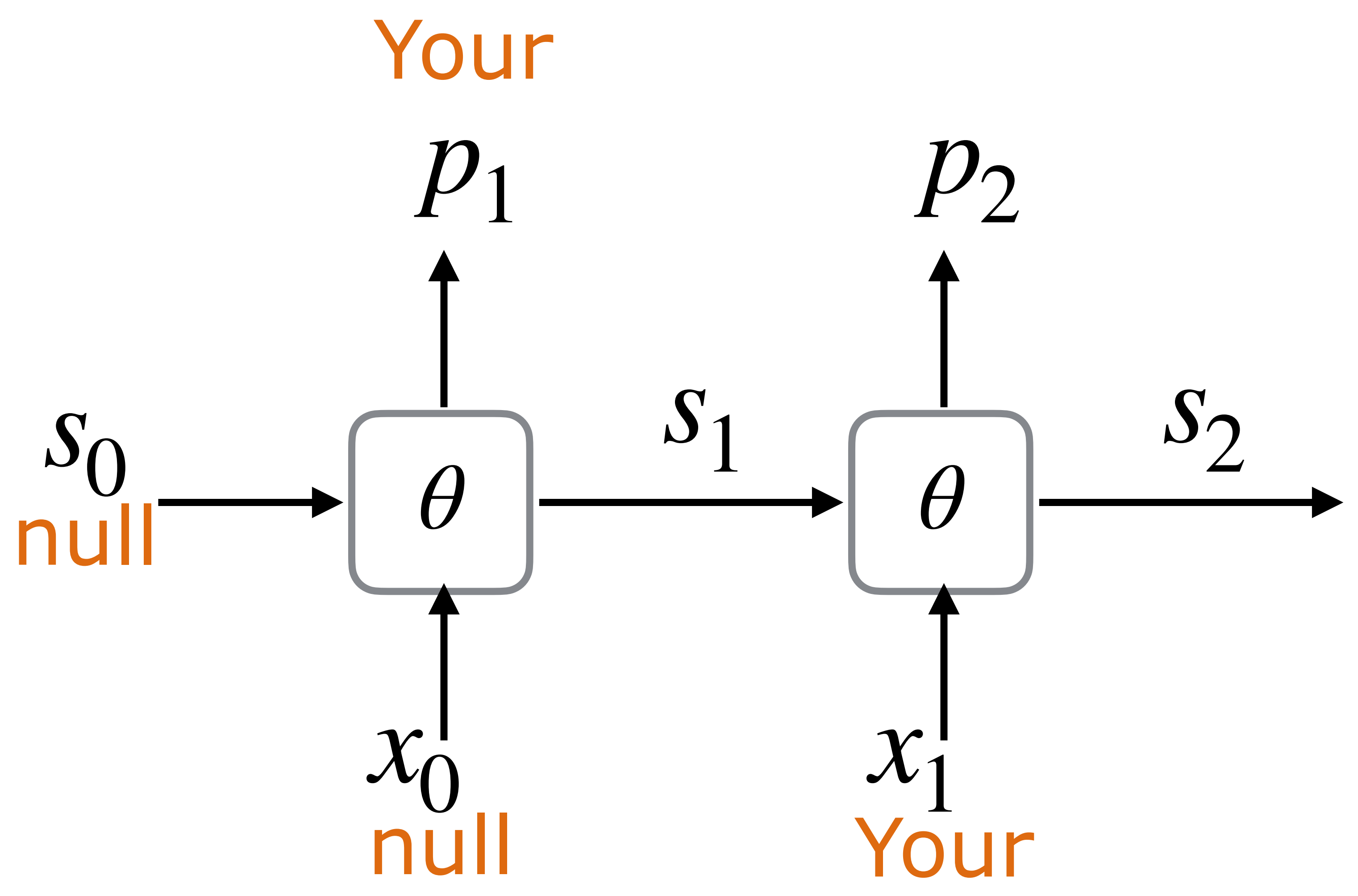




Recurrent neural network (RNN)

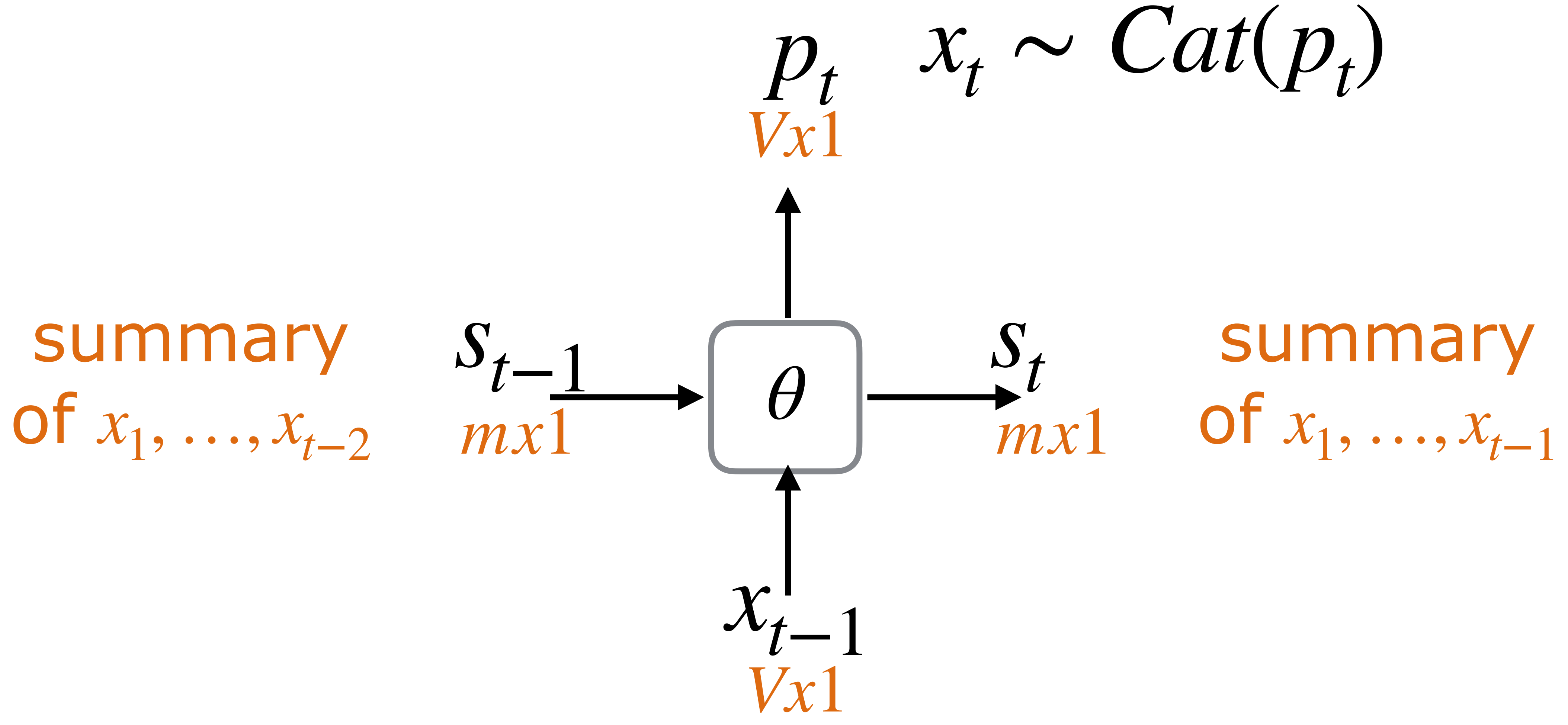
- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

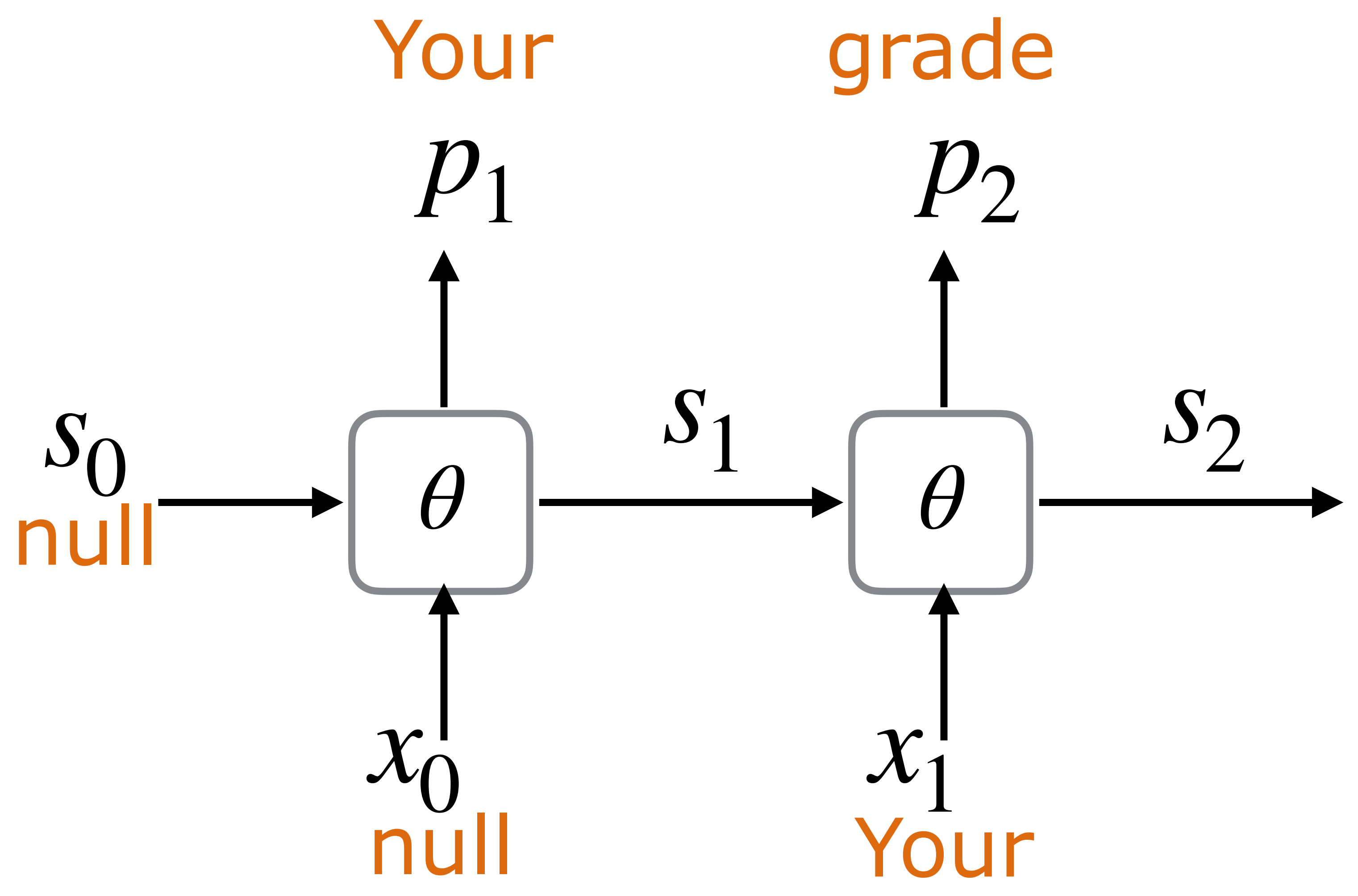




Recurrent neural network (RNN)

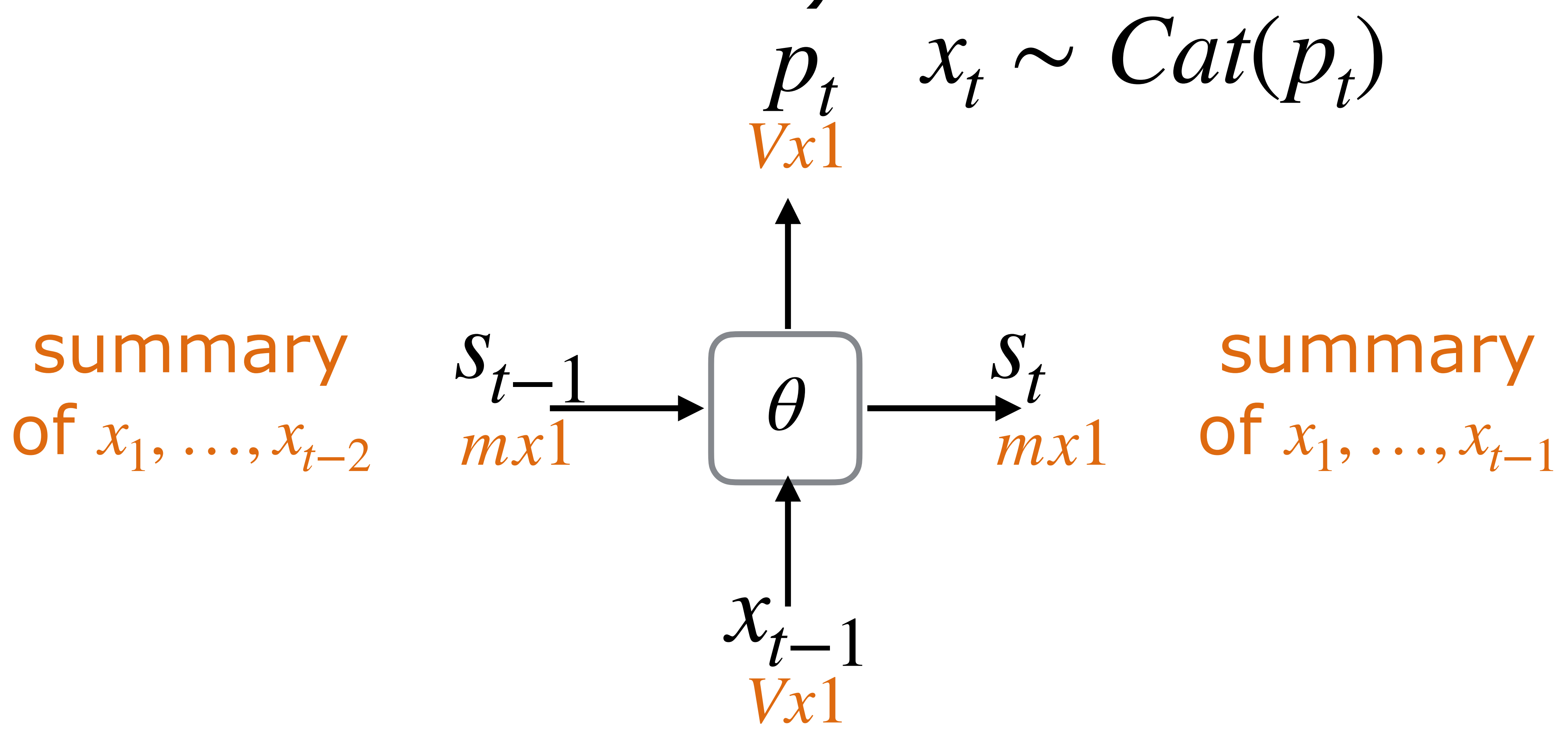
- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

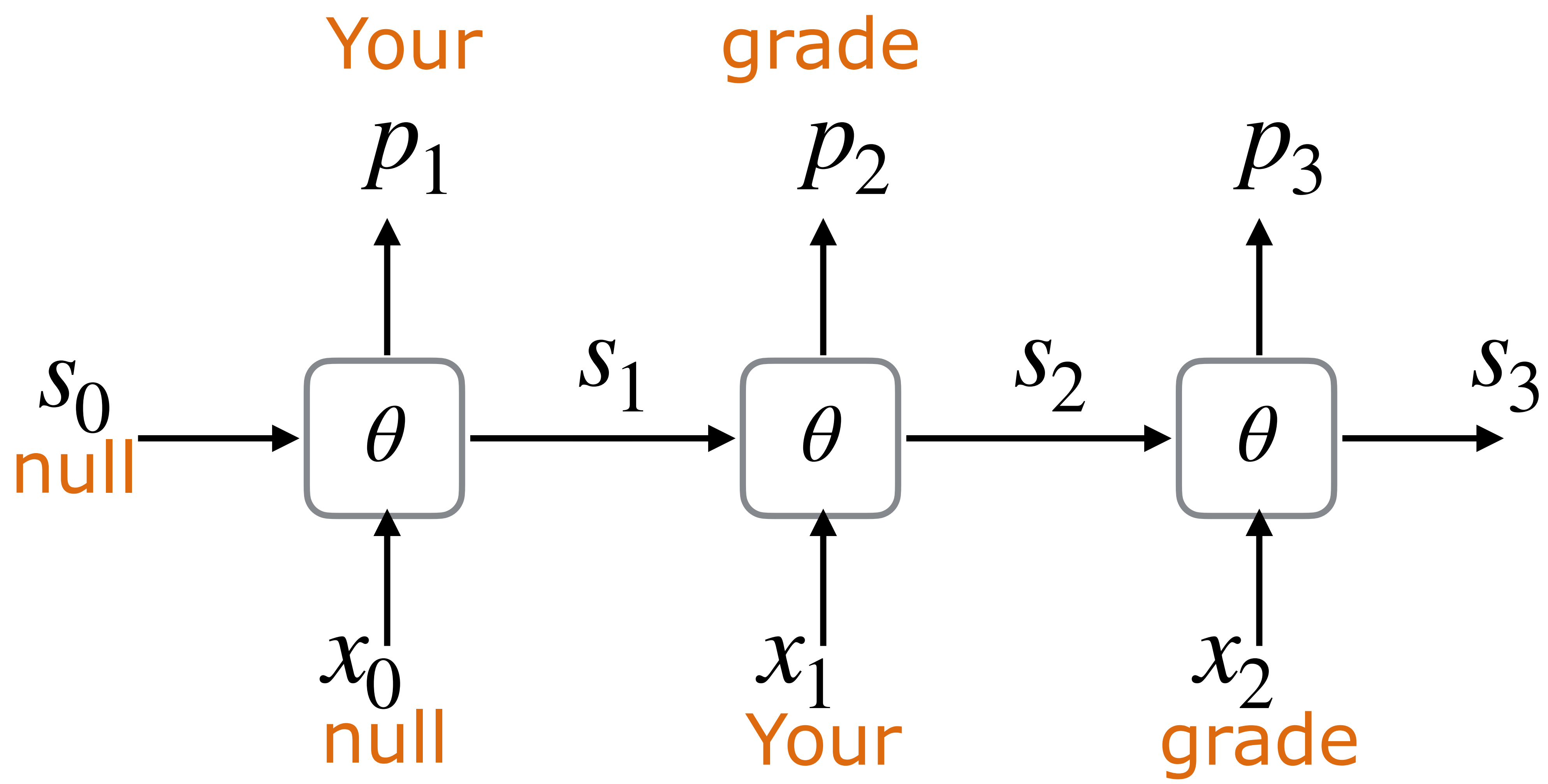




Recurrent neural network (RNN)

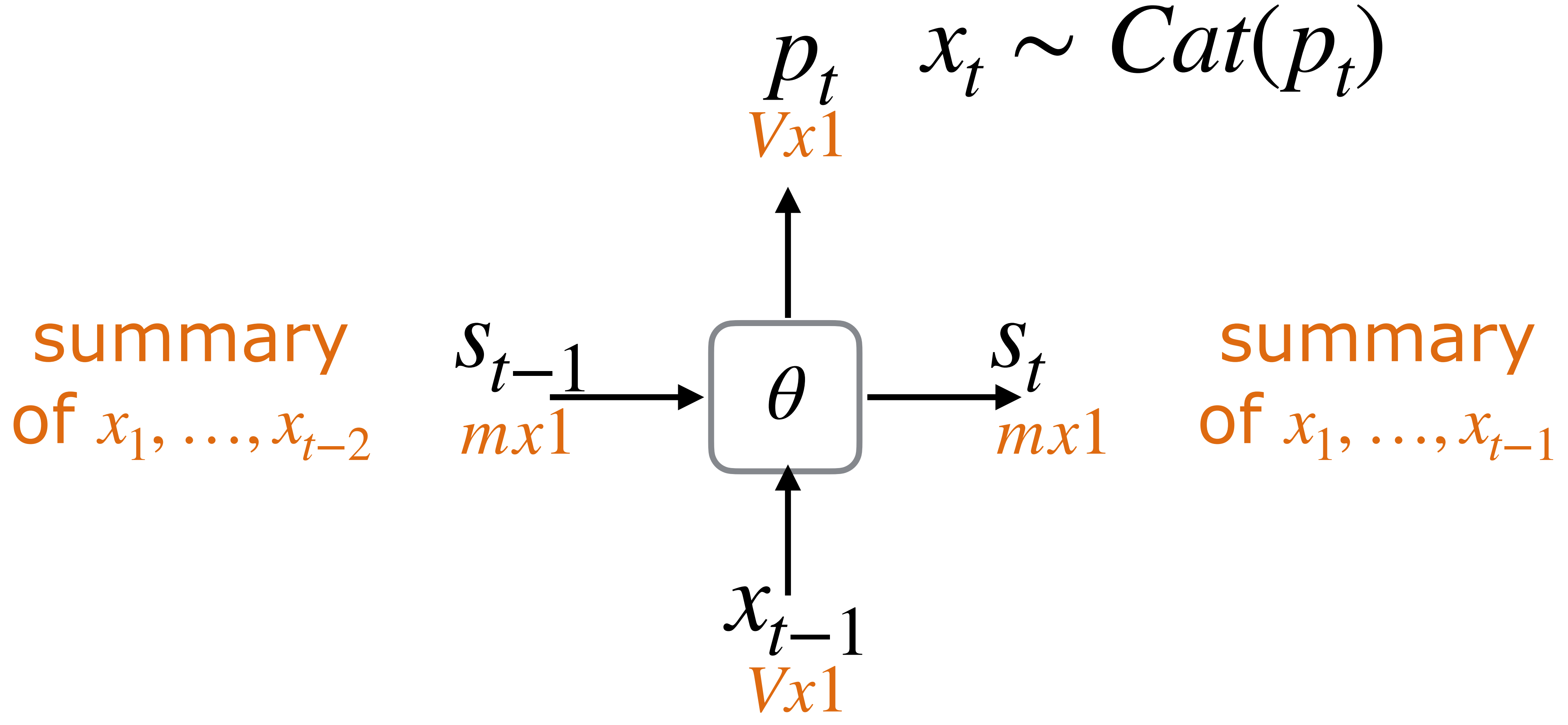
- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

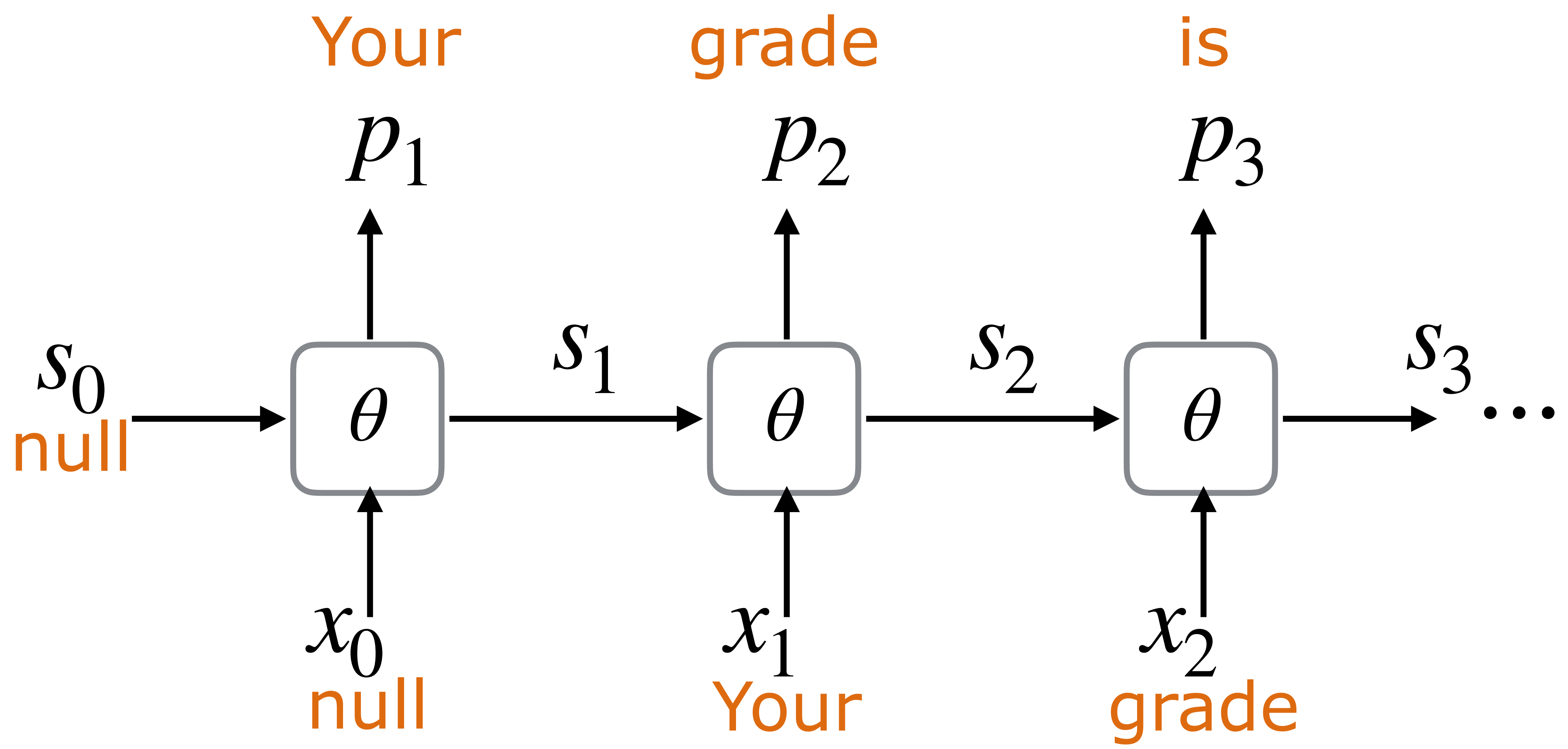




Recurrent neural network (RNN)

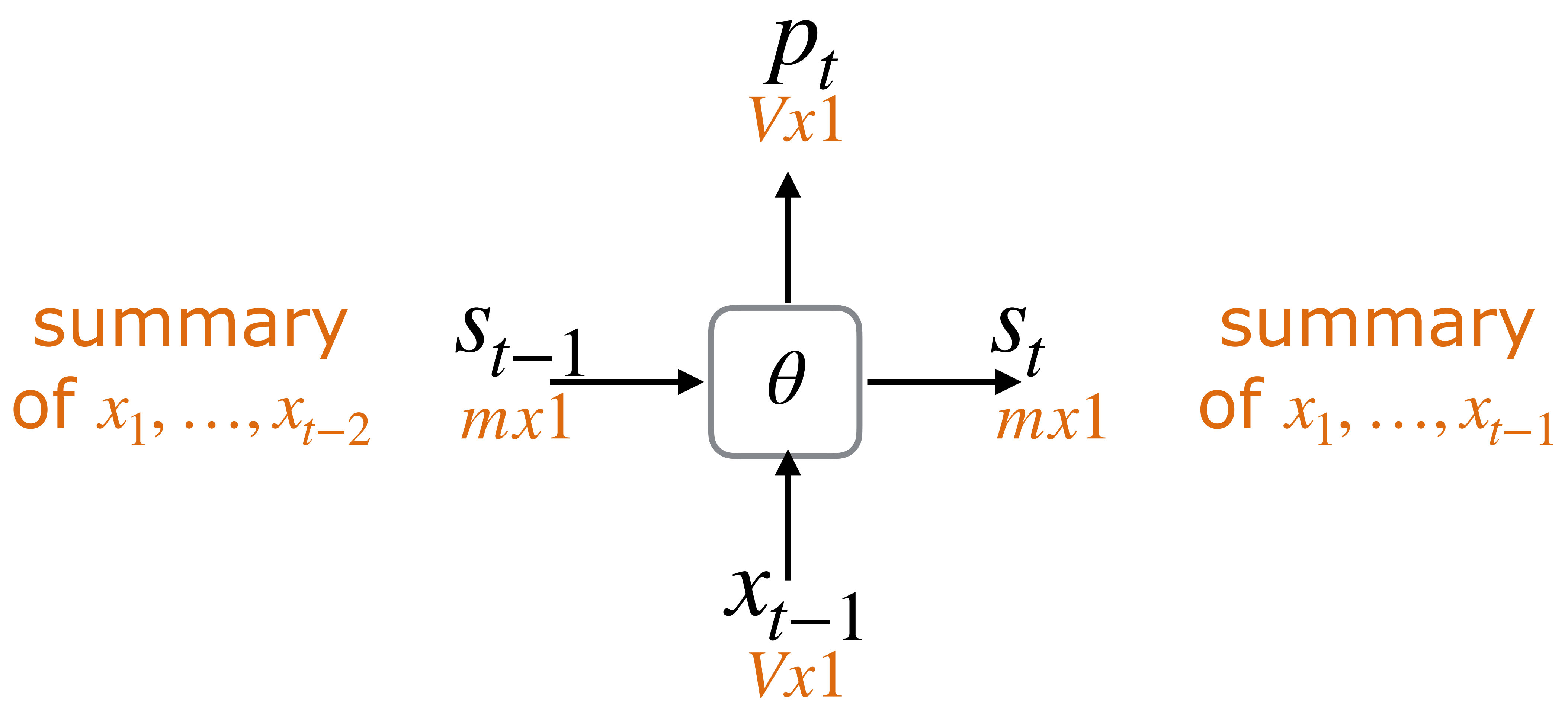
- Consider a simple state space model that we can use to generate natural language sentences (one word at a time)

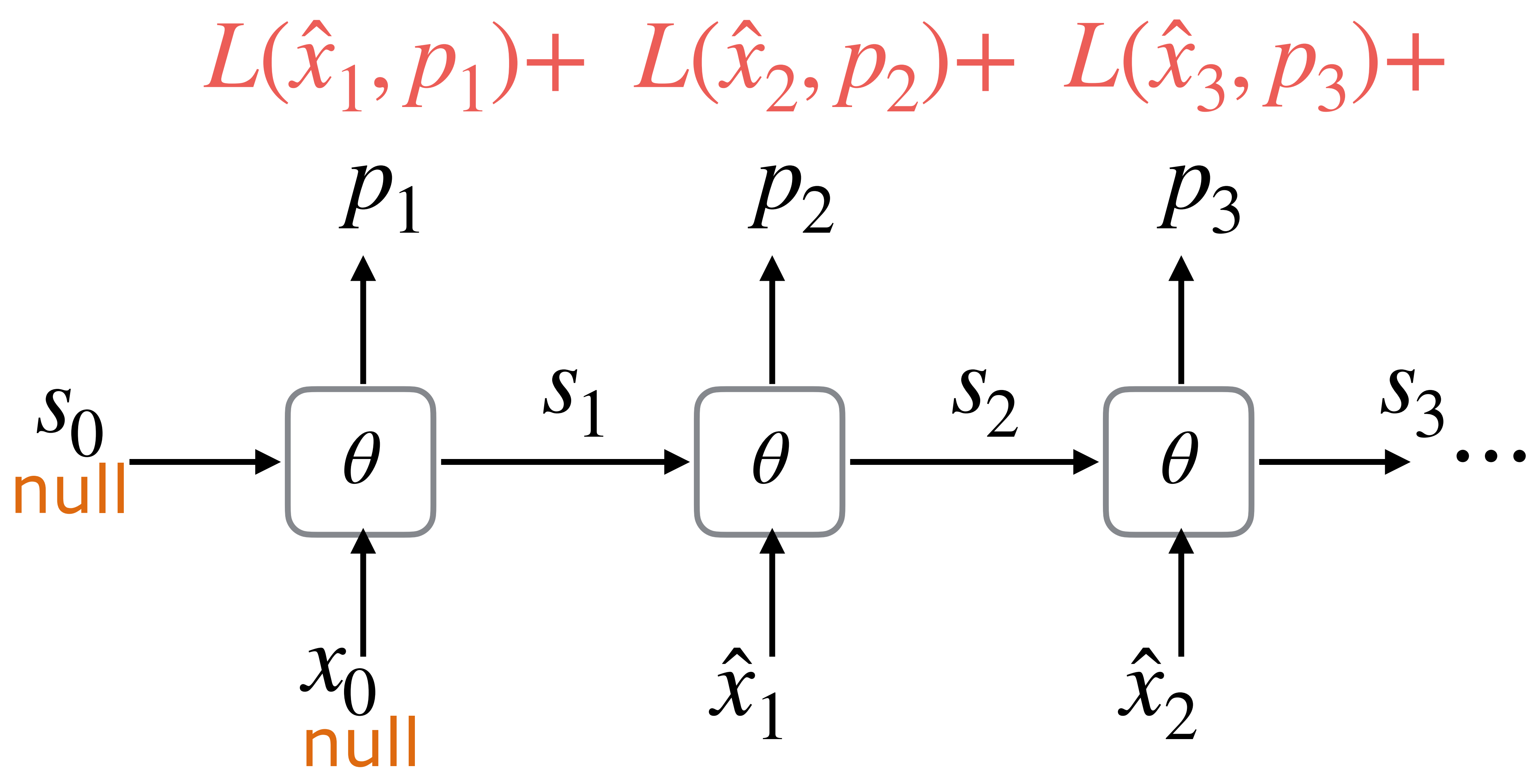




Learning recurrent neural networks (RNNs)

- When learning the model from data, e.g., observed sequence $\hat{x}_1, \hat{x}_2, \hat{x}_3, \ldots$, we introduce losses at the outputs (log-likelihood) and "teacher" force its inputs



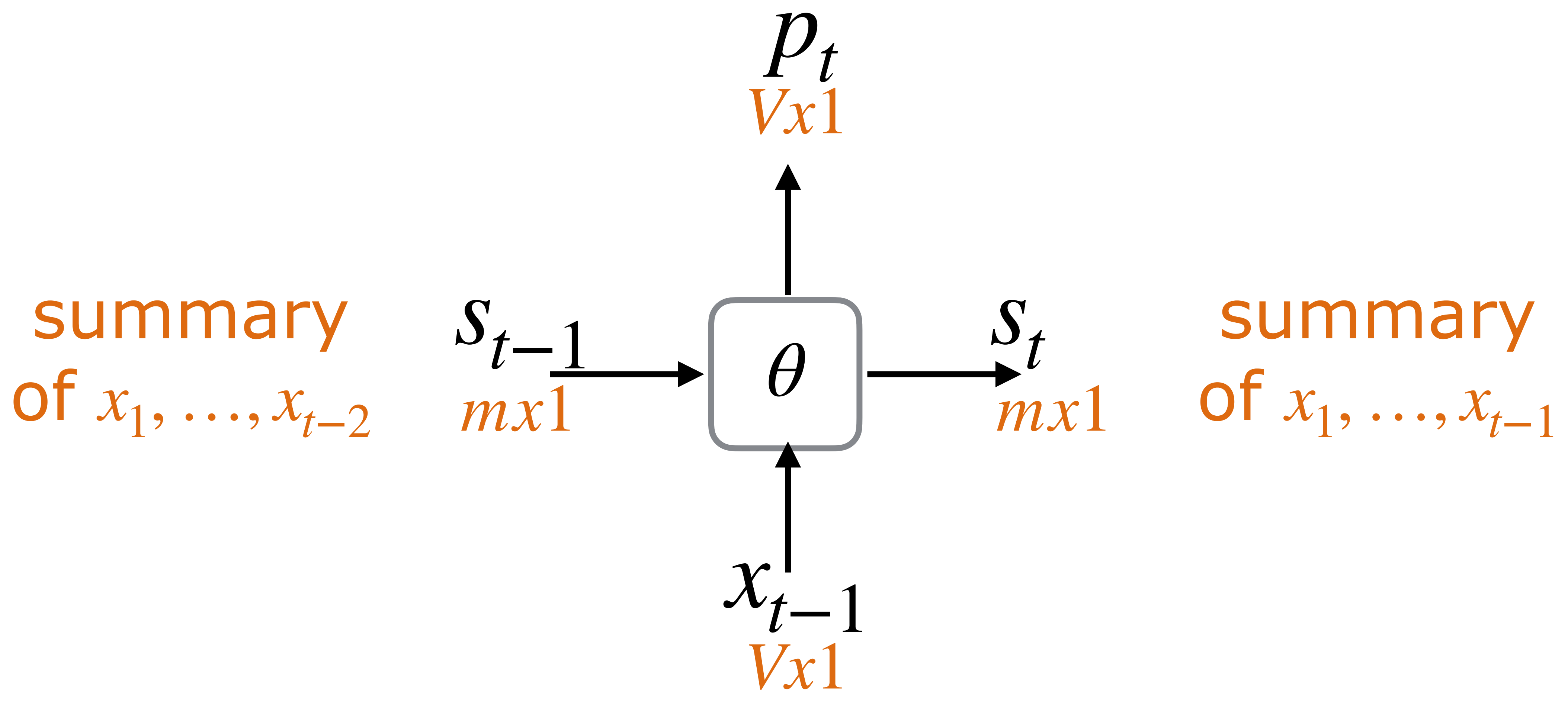


$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t)$$

$$= -\log P(\hat{x}_t \,|\, \hat{x}_{t-1}, \ldots, \hat{x}_1; \theta)$$

Recurrent neural network (RNN)

- Consider a simple RNN parameterization (so as to discuss how we learn its associated parameters)



$$\underset{mx1}{s_t} = \tanh\left(\underset{mxm}{W^s}\,\underset{mx1}{s_{t-1}} + \underset{mxV}{W^x}\,\underset{Vx1}{x_{t-1}}\right) \qquad \theta = \{W^s, W^x, W^o\}$$

$$\underset{Vx1}{p_t} = \mathrm{softmax}\left(\underset{Vxm}{W^o}\,\underset{mx1}{s_t}\right)$$

$$P(x_t \mid x_{t-1}, \ldots, x_1) = P(x_t \mid x_{t-1}, s_{t-1}) = P(x_t \mid s_t)$$

- (Offsets omitted for clarity)

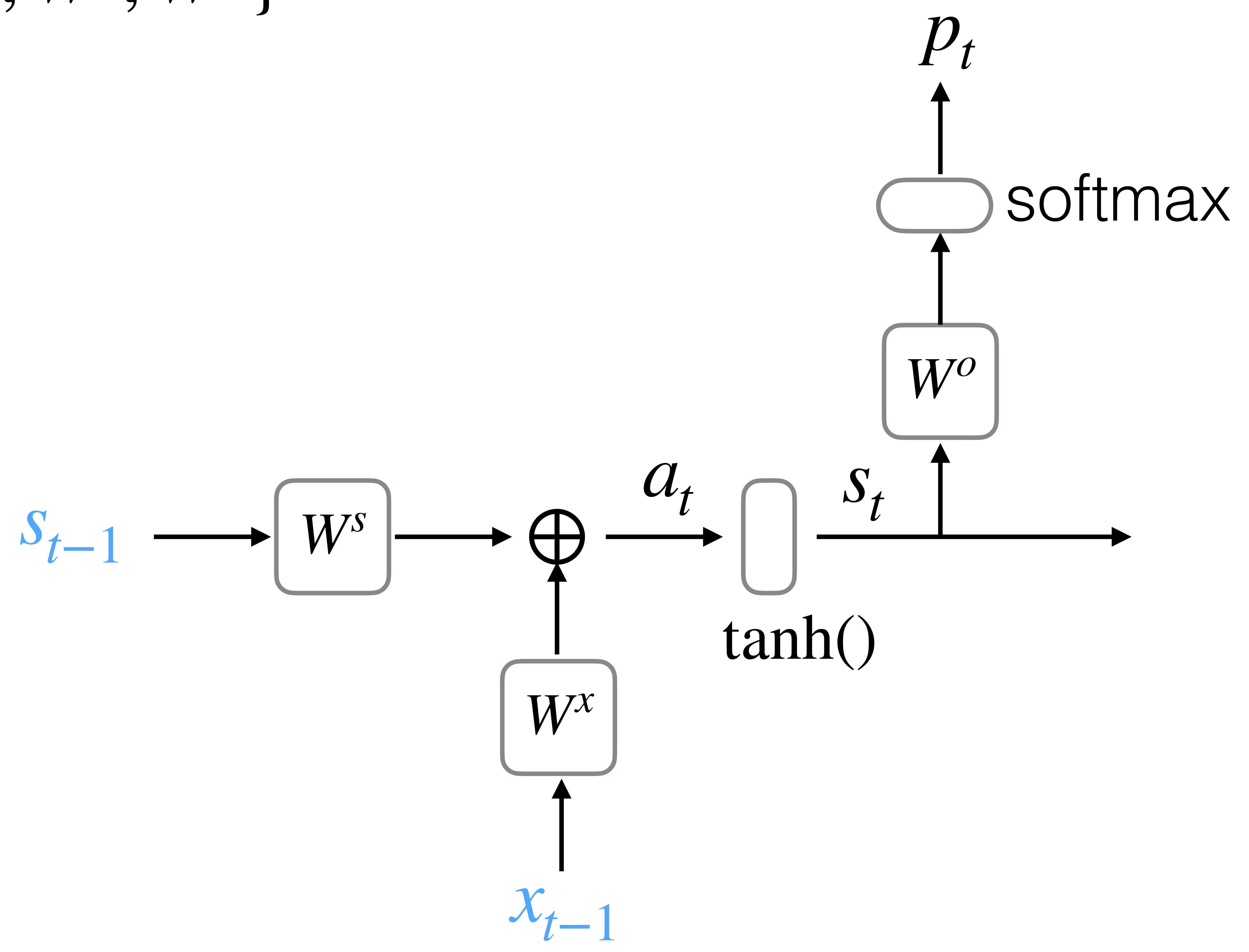
Elementary computation graph

- We can decompose the model into simple transformations, either linear (with parameters) or non-linear (no parameters)

$$\underset{mx1}{s_t} = \tanh\left(\underset{mxm}{W^s}\underset{mx1}{s_{t-1}} + \underset{mxV}{W^x}\underset{Vx1}{x_{t-1}}\right) \qquad \theta = \{W^s, W^x, W^o\}$$

$$\underset{Vx1}{p_t} = \text{softmax}\left(\underset{Vxm}{W^o}\underset{mx1}{s_t}\right)$$

- If we know s_{t-1}, x_{t-1}, we can evaluate the vector activations in the forward direction

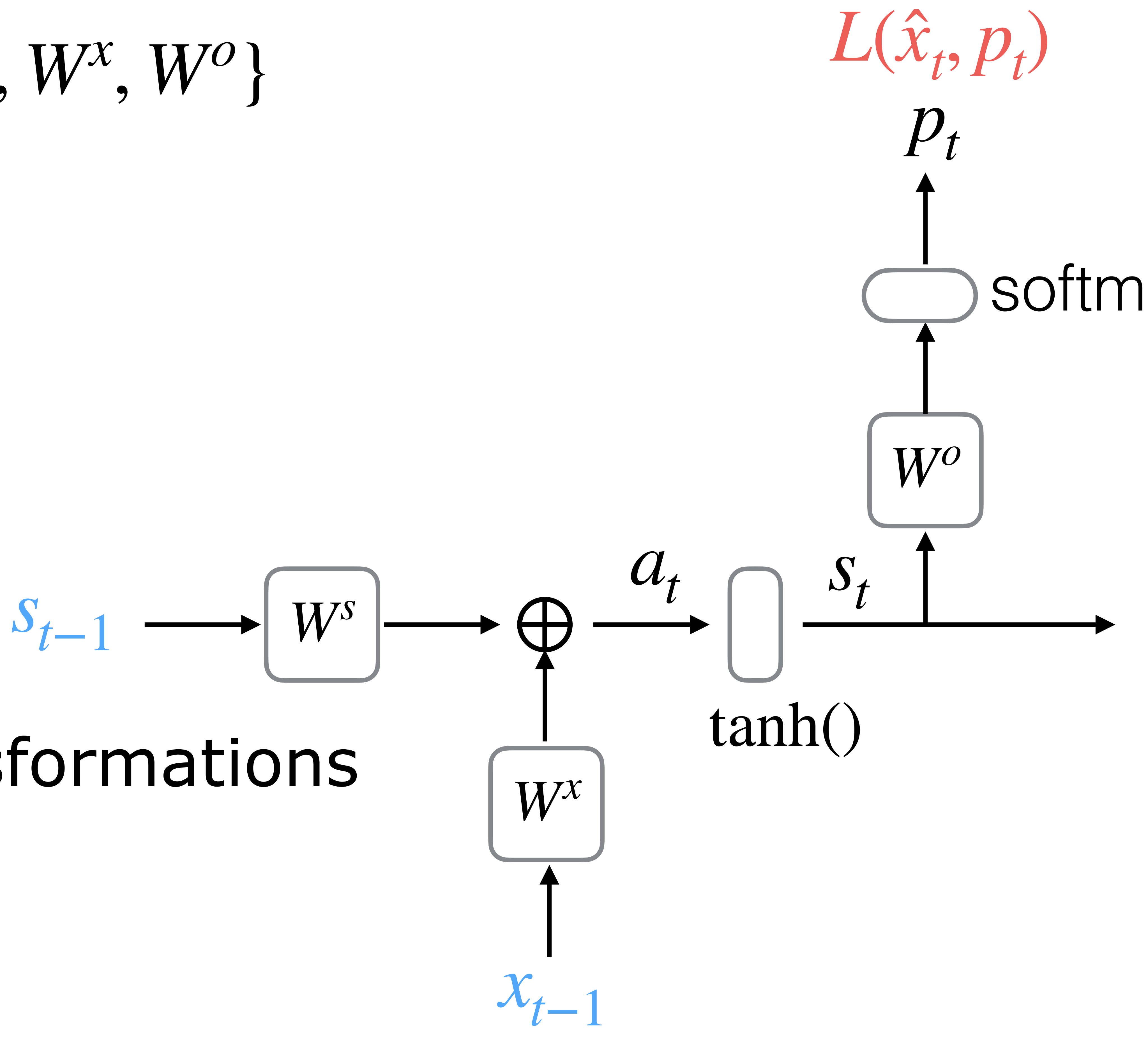


Gradient steps within the computation graph

- We can decompose the model into simple transformations, either linear (with parameters) or non-linear (no parameters)

$$\underset{mx1}{s_t} = \tanh\big(\underset{mxm}{W^s}\,\underset{mx1}{s_{t-1}} + \underset{mxV}{W^x}\,\underset{Vx1}{x_{t-1}}\big) \qquad \theta = \{W^s, W^x, W^o\}$$

$$\underset{Vx1}{p_t} = \text{softmax}\big(\underset{Vxm}{W^o}\,\underset{mx1}{s_t}\big)$$



- If we know s_{t-1}, x_{t-1}, we can evaluate the vector activations in the forward direction
- And then try to adjust the linear transformations in response to the desired output $\hat{x}_t$

(1) Updating a generic linear transformation

- Let x,z be generic inputs and outputs of any linear transformation in the model

$$\underset{dx1}{x} \longrightarrow \boxed{\underset{mxd}{W}} \longrightarrow \underset{mx1}{z} \qquad\qquad z = Wx$$

- We have x and z (forward computation); we can update the weights if we also have access to

$$\underbrace{\frac{\partial L}{\partial z}}_{mx1}$$

gradient of the loss wrt the output of the linear transformation

(1) Updating a generic linear transformation

- Let x,z be generic inputs and outputs of any linear transformation in the model

$$\underset{dx1}{x} \longrightarrow \boxed{\underset{mxd}{W}} \longrightarrow \underset{mx1}{z} \qquad z = Wx$$

- We have x and z (forward computation); we can update the weights if we also have access to

$$\underset{mx1}{\frac{\partial L}{\partial z}}$$

gradient of the loss
wrt the output of
the linear transformation

- By chain rule

$$\frac{\partial L}{\partial W_{ij}} = \sum_{k=1}^{m} \frac{\partial z_k}{\partial W_{ij}} \frac{\partial L}{\partial z_k} = x_j \frac{\partial L}{\partial z_i}$$

- or in terms of matrices

$$\underset{mxd}{\frac{\partial L}{\partial W}} = \underset{mx1}{\frac{\partial L}{\partial z}} \underset{1xd}{x^T}$$

- which is the gradient we need to update W

(2) One step backpropagation

- Let x,z be generic inputs and outputs of any linear transformation in the model

$$\underset{dx1}{x} \longrightarrow \boxed{\underset{mxd}{W}} \longrightarrow \underset{mx1}{z} \qquad z = Wx$$

- We have x and z (forward computation); we can push the gradient

$$\underset{mx1}{\frac{\partial L}{\partial z}}$$

gradient of the loss
wrt the output of
the linear transformation

- one step further back (to be wrt inputs) by again evoking the chain rule

$$\underset{dx1}{\frac{\partial L}{\partial x}} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z} = J^T \frac{\partial L}{\partial z} = \underset{dxm}{W^T} \underset{mx1}{\frac{\partial L}{\partial z}}$$

gradient of the loss
wrt the input of
the linear transformation

- where $J_{ij} = \partial z_i / \partial x_j = W_{ij}$ is the Jacobian matrix of the linear transformation

(2) One step backpropagation

- Let x,z be generic inputs and outputs of any linear transformation in the model

$$\underset{dx1}{x} \longrightarrow \boxed{\underset{mxd}{W}} \longrightarrow \underset{mx1}{z} \qquad z = Wx$$

- We have x and z (forward computation); we can push the gradient

$$\underset{mx1}{\frac{\partial L}{\partial z}}$$

gradient of the loss
wrt the output of
the linear transformation

- one step further back (to be wrt inputs) by again evoking the chain rule

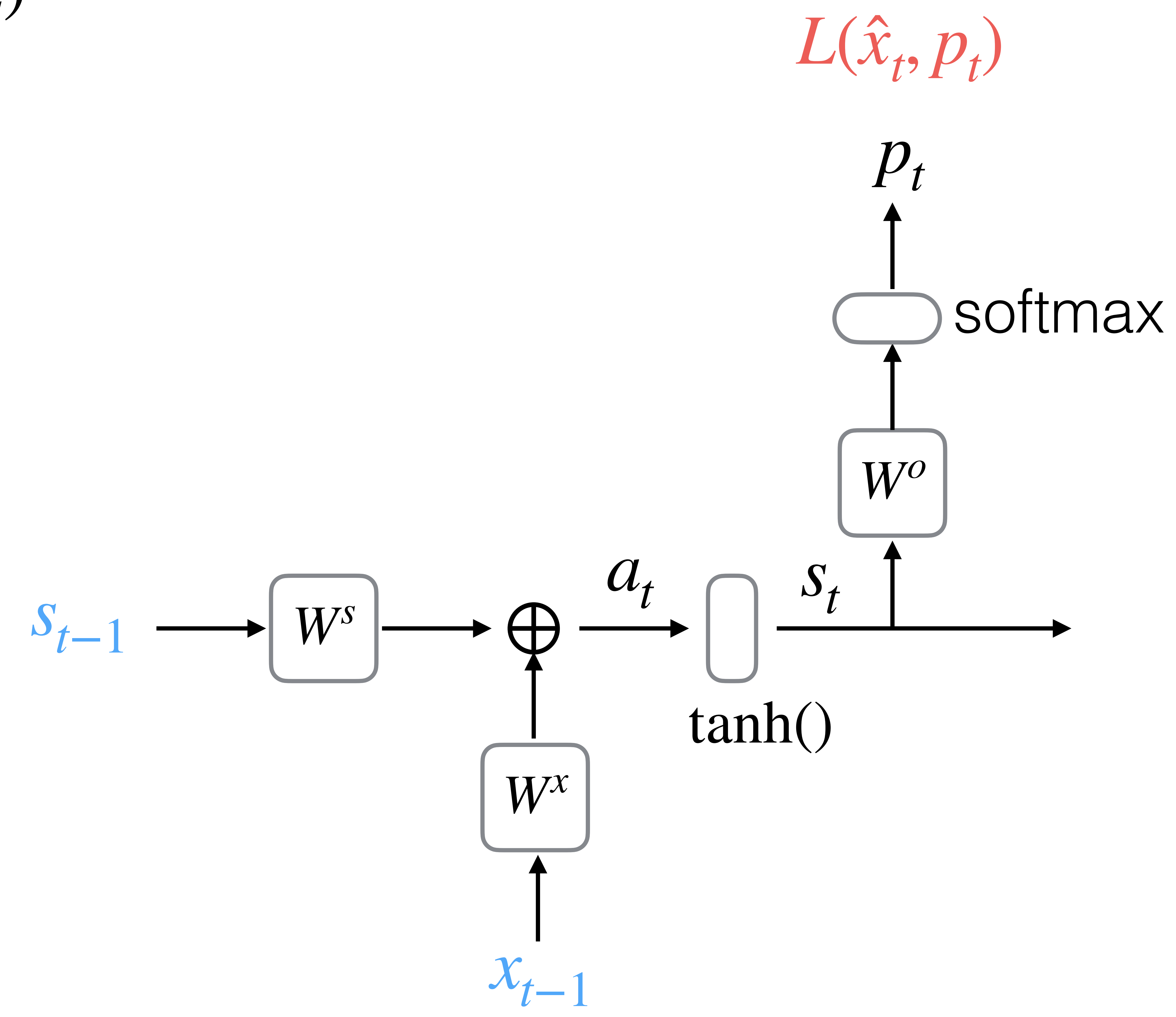
$$\underset{dx1}{\frac{\partial L}{\partial x}} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z} = J^T \frac{\partial L}{\partial z} = \underset{dxm}{W^T} \underset{mx1}{\frac{\partial L}{\partial z}}$$

gradient of the loss
wrt the input of
the linear transformation

- where $J_{ij} = \partial z_i / \partial x_j = W_{ij}$ is the Jacobian matrix of the linear transformation. Any non-linear transformation acts the same, just has a different Jacobian

Backpropagation

$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t), \;\; p_t(y) = \frac{\exp(z_y)}{\sum_j \exp(z_j)}$$

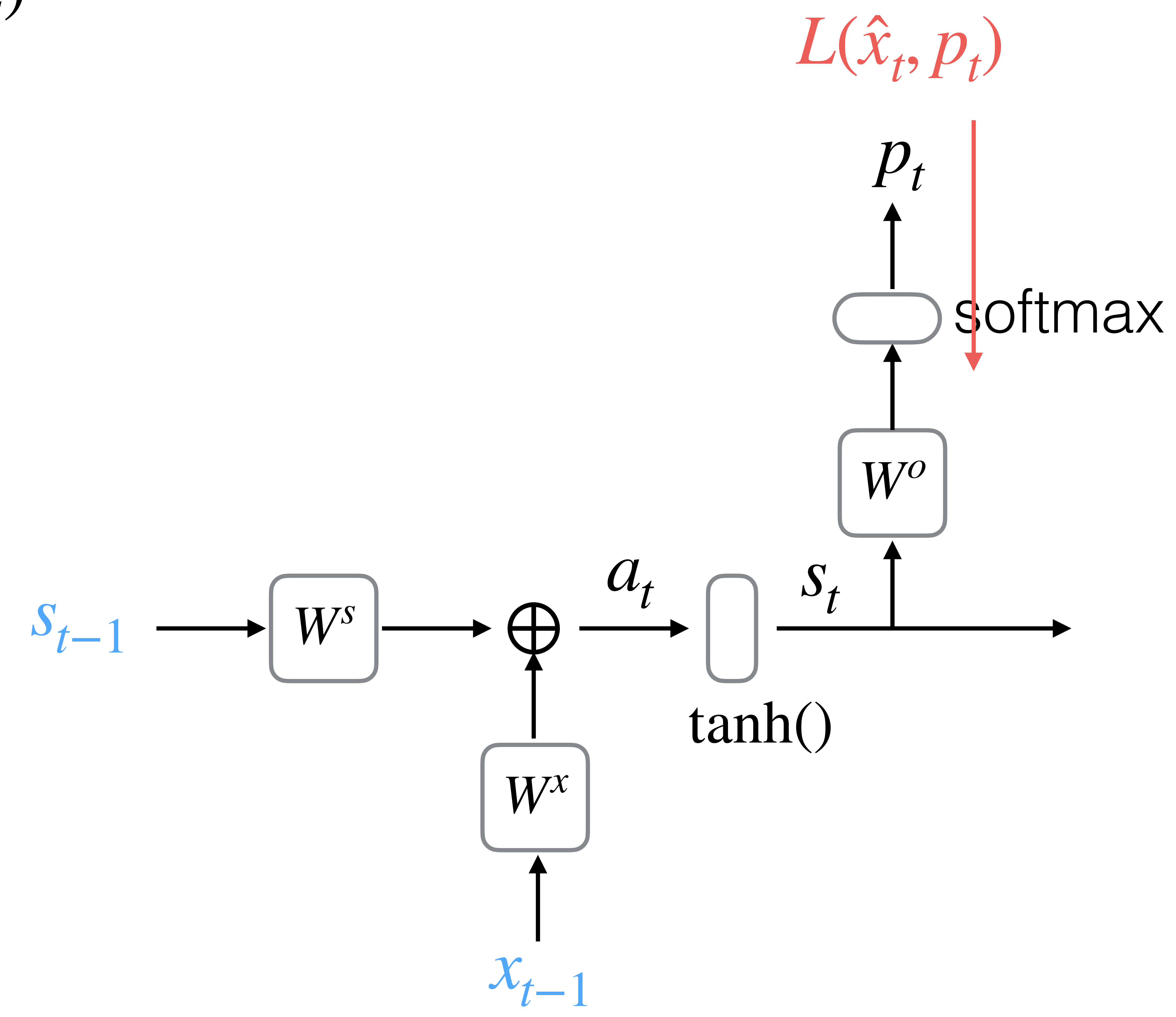


Backpropagation

$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t), \;\; p_t(y) = \frac{\exp(z_y)}{\sum_j \exp(z_j)}$$

$$\underbrace{\frac{\partial L}{\partial z}}_{Vx1} = -\,(\underbrace{I(\hat{x}_t)}_{Vx1} - \underbrace{p_t}_{Vx1})$$

one-hot vector



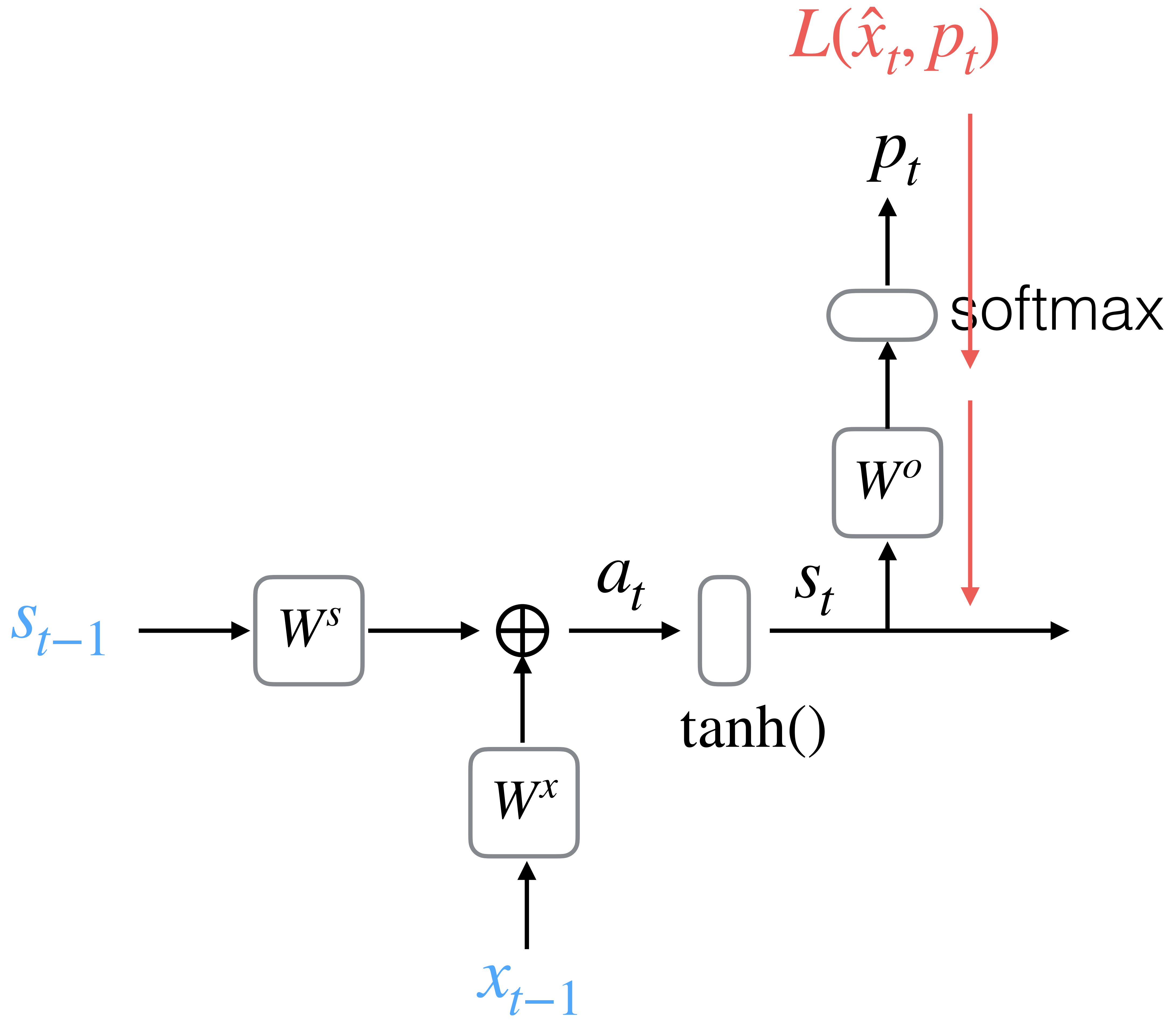
Backpropagation

$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t), \quad p_t(y) = \frac{\exp(z_y)}{\sum_j \exp(z_j)}$$

$$\underbrace{\frac{\partial L}{\partial z}}_{Vx1} = -(\underbrace{I(\hat{x}_t)}_{Vx1} - \underbrace{p_t}_{Vx1})$$

one-hot vector

$$\underbrace{\frac{\partial L}{\partial s_t}}_{mx1} = \underbrace{(W^o)^T}_{mxV} \underbrace{\frac{\partial L}{\partial z}}_{Vx1}$$



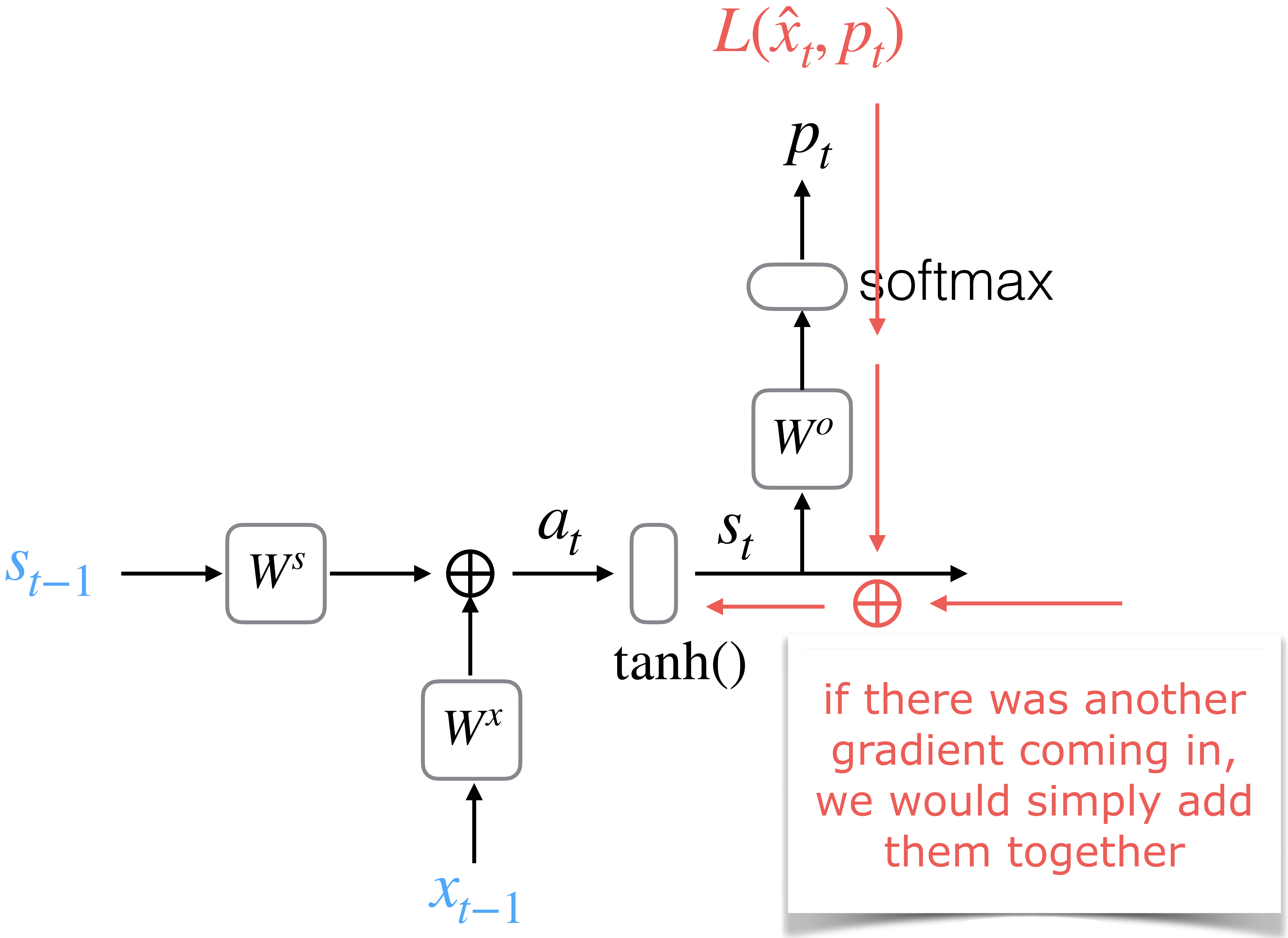
Backpropagation

$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t), \;\; p_t(y) = \frac{\exp(z_y)}{\sum_j \exp(z_j)}$$

$$\underset{Vx1}{\frac{\partial L}{\partial z}} = -\,(\underset{Vx1}{I(\hat{x}_t)} - \underset{Vx1}{p_t})$$

one-hot
vector

$$\underset{mx1}{\frac{\partial L}{\partial s_t}} = \underset{mxV}{(W^o)^T} \underset{Vx1}{\frac{\partial L}{\partial z}}$$



Backpropagation

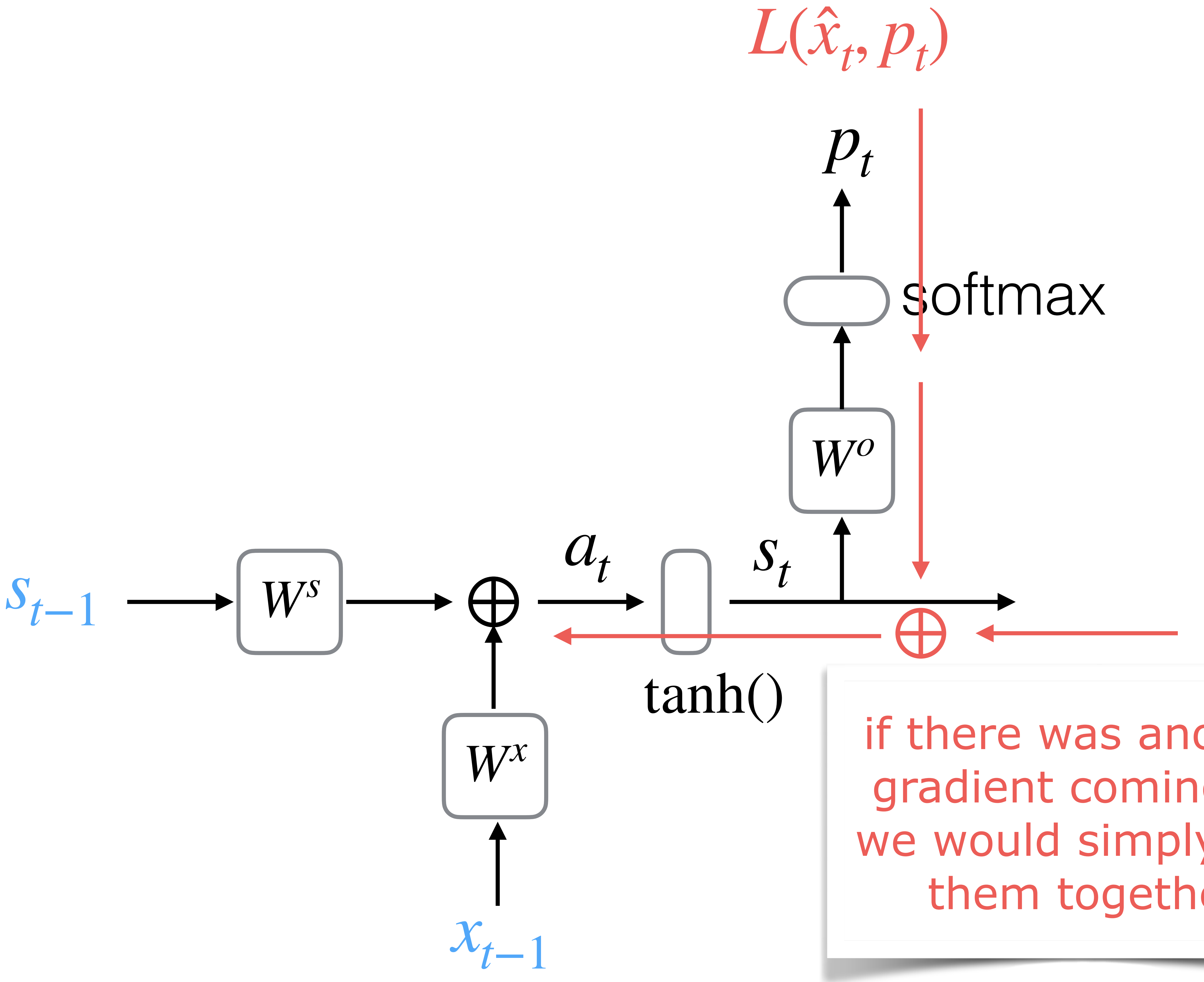
$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t), \;\; p_t(y) = \frac{\exp(z_y)}{\sum_j \exp(z_j)}$$

$$\underset{Vx1}{\frac{\partial L}{\partial z}} = -(\underset{Vx1}{I(\hat{x}_t)} - \underset{Vx1}{p_t})$$

one-hot vector

$$\underset{mx1}{\frac{\partial L}{\partial s_t}} = \underset{mxV}{(W^o)^T} \underset{Vx1}{\frac{\partial L}{\partial z}}$$

$$\underset{mx1}{\frac{\partial L}{\partial a_t}} = \underset{mxm}{\mathrm{diag}(1 - \tanh^2(a_t))} \underset{mx1}{\frac{\partial L}{\partial s_t}}$$



if there was another gradient coming in, we would simply add them together

Backpropagation

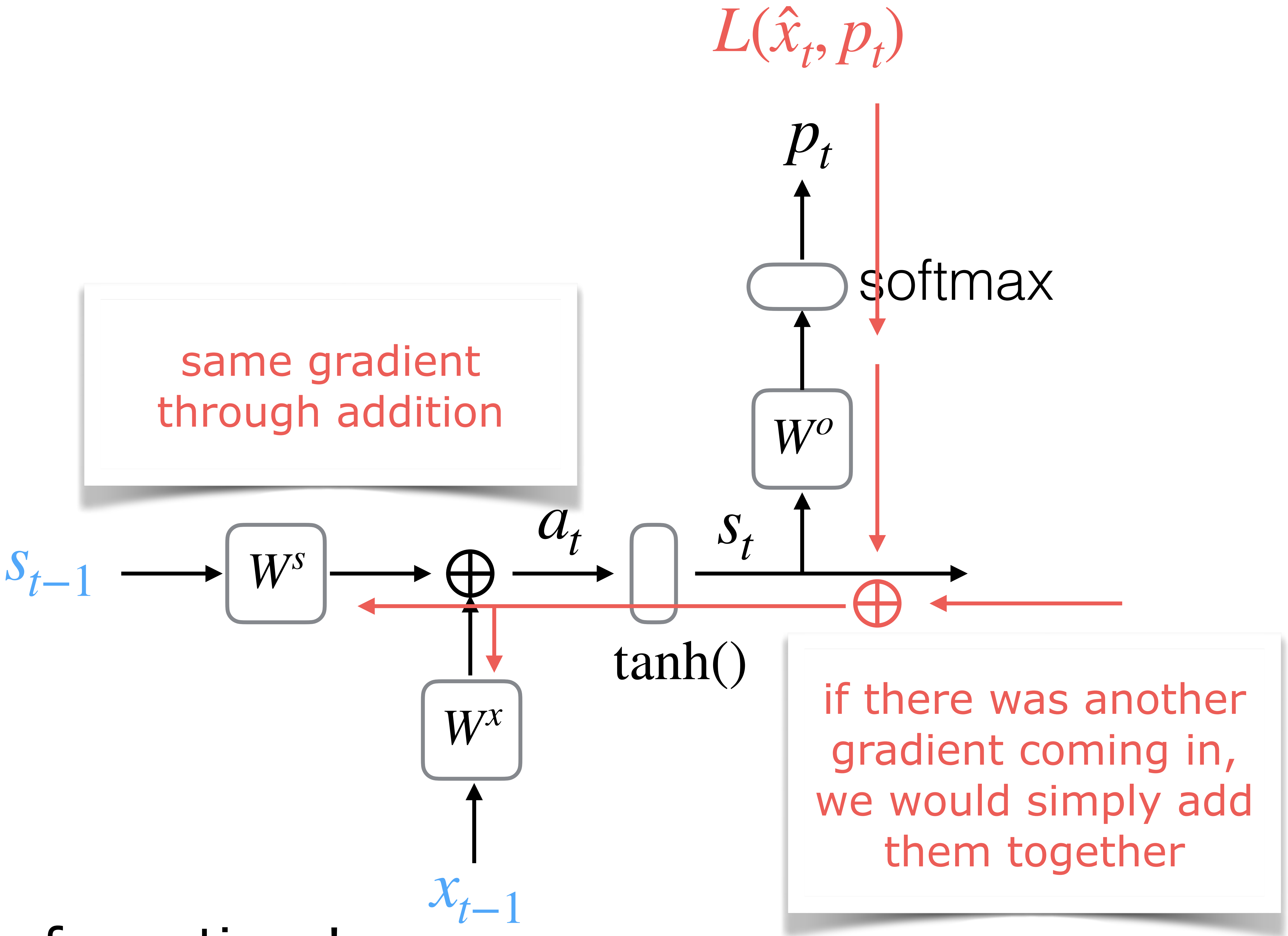
$$L(\hat{x}_t, p_t) = -\log p_t(\hat{x}_t), \;\; p_t(y) = \frac{\exp(z_y)}{\sum_j \exp(z_j)}$$

$$\underset{Vx1}{\frac{\partial L}{\partial z}} = -(\underset{Vx1}{I(\hat{x}_t)} - \underset{Vx1}{p_t})$$

one-hot vector

$$\underset{mx1}{\frac{\partial L}{\partial s_t}} = \underset{mxV}{(W^o)^T} \underset{Vx1}{\frac{\partial L}{\partial z}}$$

$$\underset{mx1}{\frac{\partial L}{\partial a_t}} = \underset{mxm}{\mathrm{diag}(1 - \tanh^2(a_t))} \underset{mx1}{\frac{\partial L}{\partial s_t}}$$



we can now update all the linear transformations!