Massachusetts
Institute of
Technology

Introduction to Deep Learning

Stefanie Jegelka
MIT

Introduction & Outline

http://people.csail.mit.edu/stefje/

Introduction & Outline

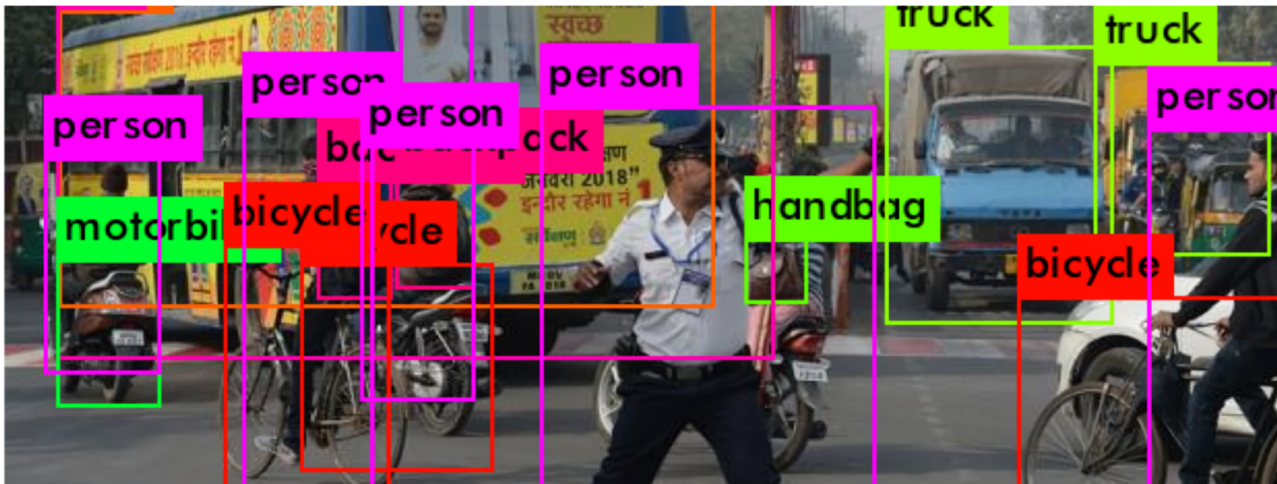
http://people.csail.mit.edu/stefje/

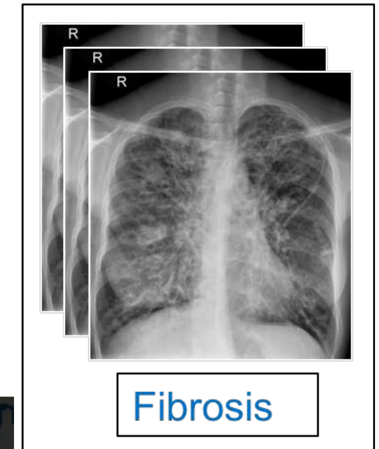
- **Today: Fully connected neural networks**
  - How do neural networks represent data?
  - How can we “train” a neural network?
  - Example: Fashion MNIST

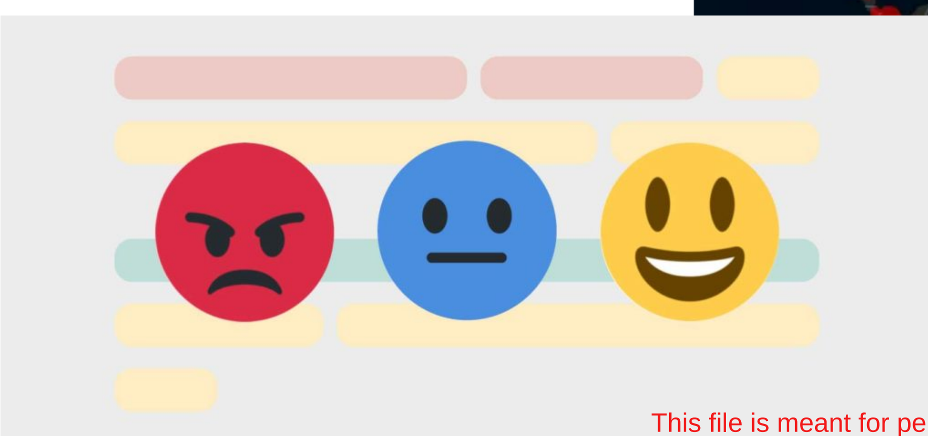
- **Wed:** Neural networks for images: convolutional neural networks

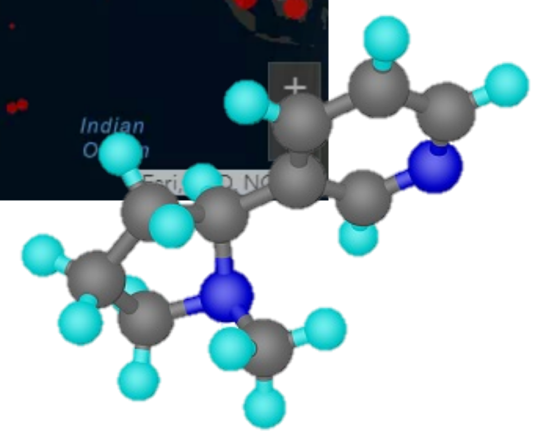
- **Fri:** Transfer learning and Neural networks for graphs

Predictions from "everything"





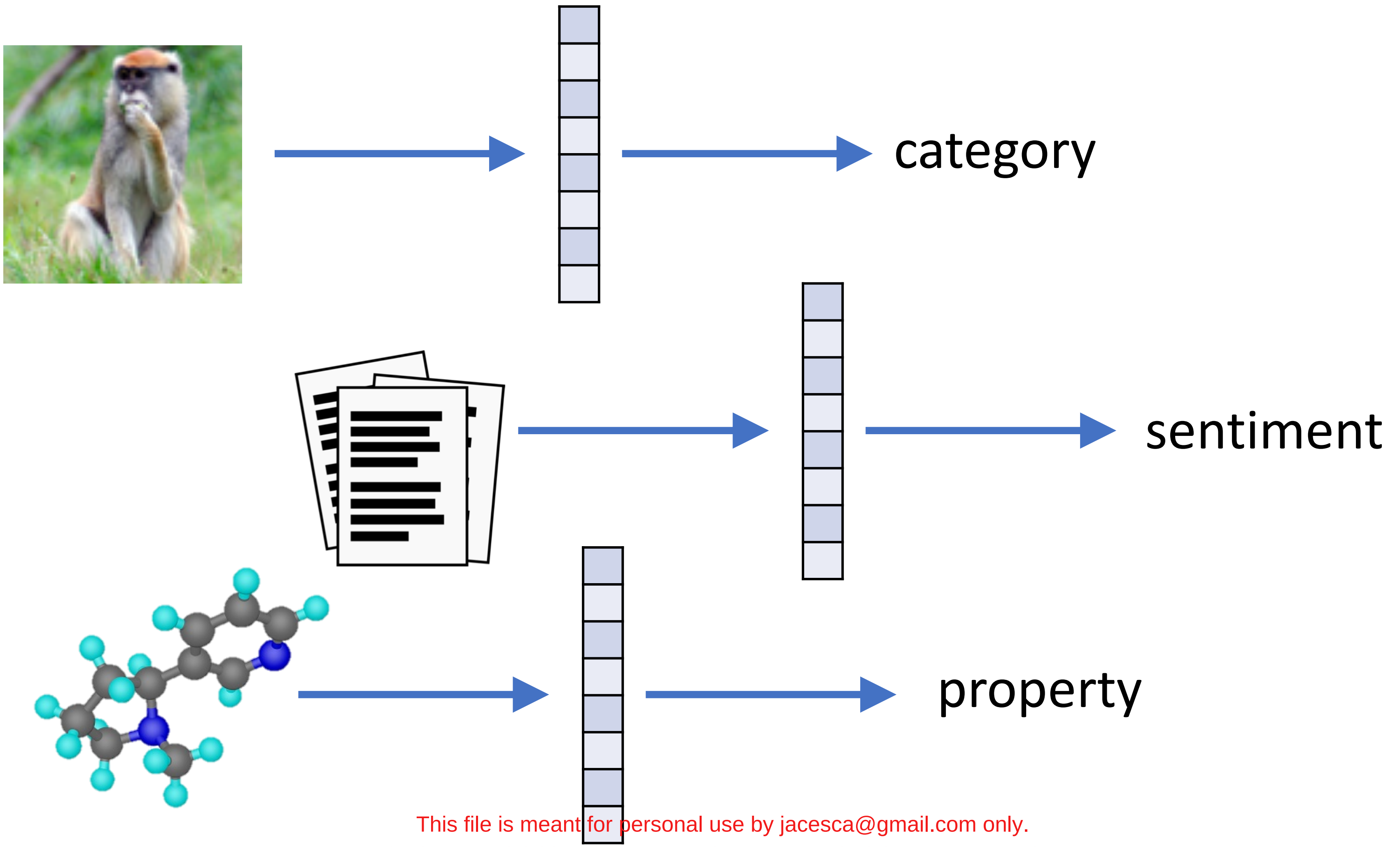




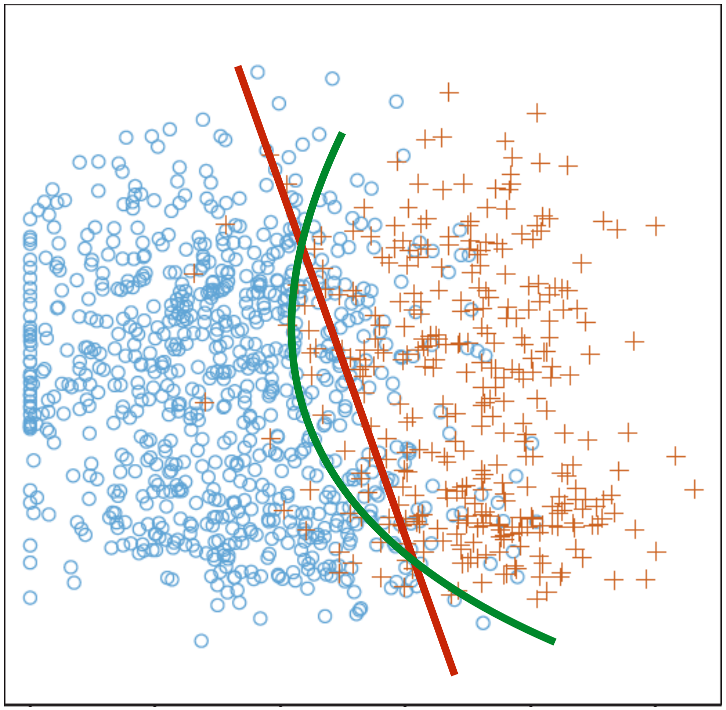
https://hackernoon.com/drafts/523x232ih.png

A general strategy

- Encode data as useful, informative feature vectors



Feature encodings: we've seen them before

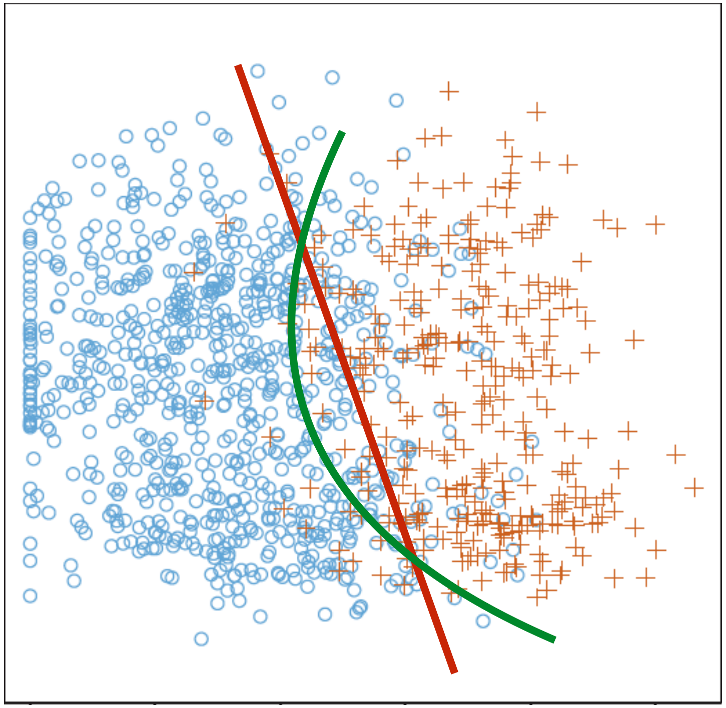


- Linear classifiers:
  compare $\boldsymbol{\theta}^T\mathbf{X}$ to a threshold
  - learn "good" vector $\boldsymbol{\theta}$ and threshold
- Nonlinear classifiers:
  compare $h(\mathbf{X})$ to a threshold
  - learn "good" function h and threshold

- Nonlinear classifier $\theta_1 X_1 + \theta_2 X_2 + \theta_{12} X_1 X_2$
  is actually linear if we redefine $\mathbf{X} = (X_1, X_2, X_1 X_2)$

- Feature-based linear classifier: compare $\boldsymbol{\theta}^T \phi(\mathbf{X})$ to a threshold

Feature encodings: we've seen them before

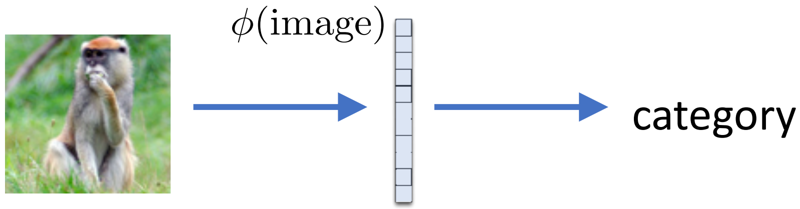


- Linear classifiers:
  compare $\boldsymbol{\theta}^T \mathbf{X}$ to a threshold
  - learn "good" vector $\boldsymbol{\theta}$ and threshold
- Nonlinear classifiers:
  compare $h(\mathbf{X})$ to a threshold
  - learn "good" function h and threshold

- Nonlinear classifier $\theta_1 X_1 + \theta_2 X_2 + \theta_{12} X_1 X_2$ is actually linear if we redefine $\mathbf{X} = (X_1, X_2, X_1 X_2)$
- Feature-based linear classifier: compare $\boldsymbol{\theta}^T \phi(\mathbf{X})$ to a threshold

A general strategy

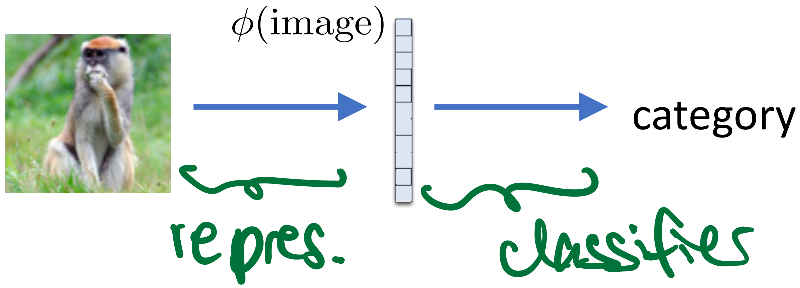
- Encode data as useful, informative feature vectors



- But, what is a good encoding?

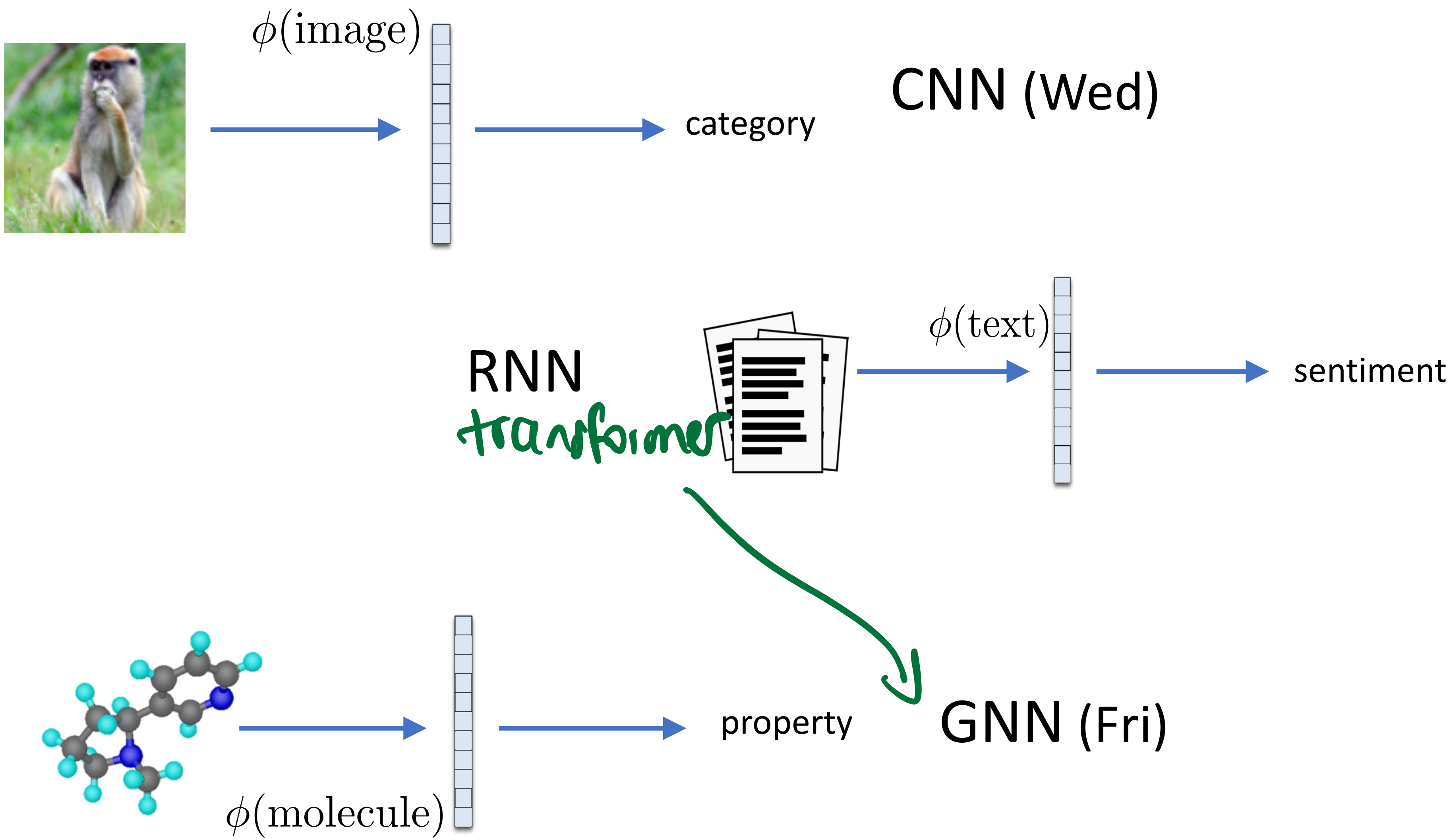
A general strategy

- Encode data as useful, informative feature vectors



- But, what is a good encoding (representation)?
- **Neural Networks: learn it from the data!**
- Challenge: learn encoding and prediction *simultaneously*

Specialized methods to encode



Deep Learning: reasons for success

- **Lots of data**
  - many problems can only be solved at scale

Deep Learning: reasons for success

- **Lots of data**
  - many problems can only be solved at scale
- **Computational resources** (e.g. GPUs)
  - systems that support running deep ML algorithms at scale

Deep Learning: reasons for success

- **Lots of data**
  - many problems can only be solved at scale
- **Computational resources** (e.g. GPUs)
  - systems that support running deep ML algorithms at scale
- **Large models are easier to train**
  - can be successfully estimated with simple gradient based algorithms

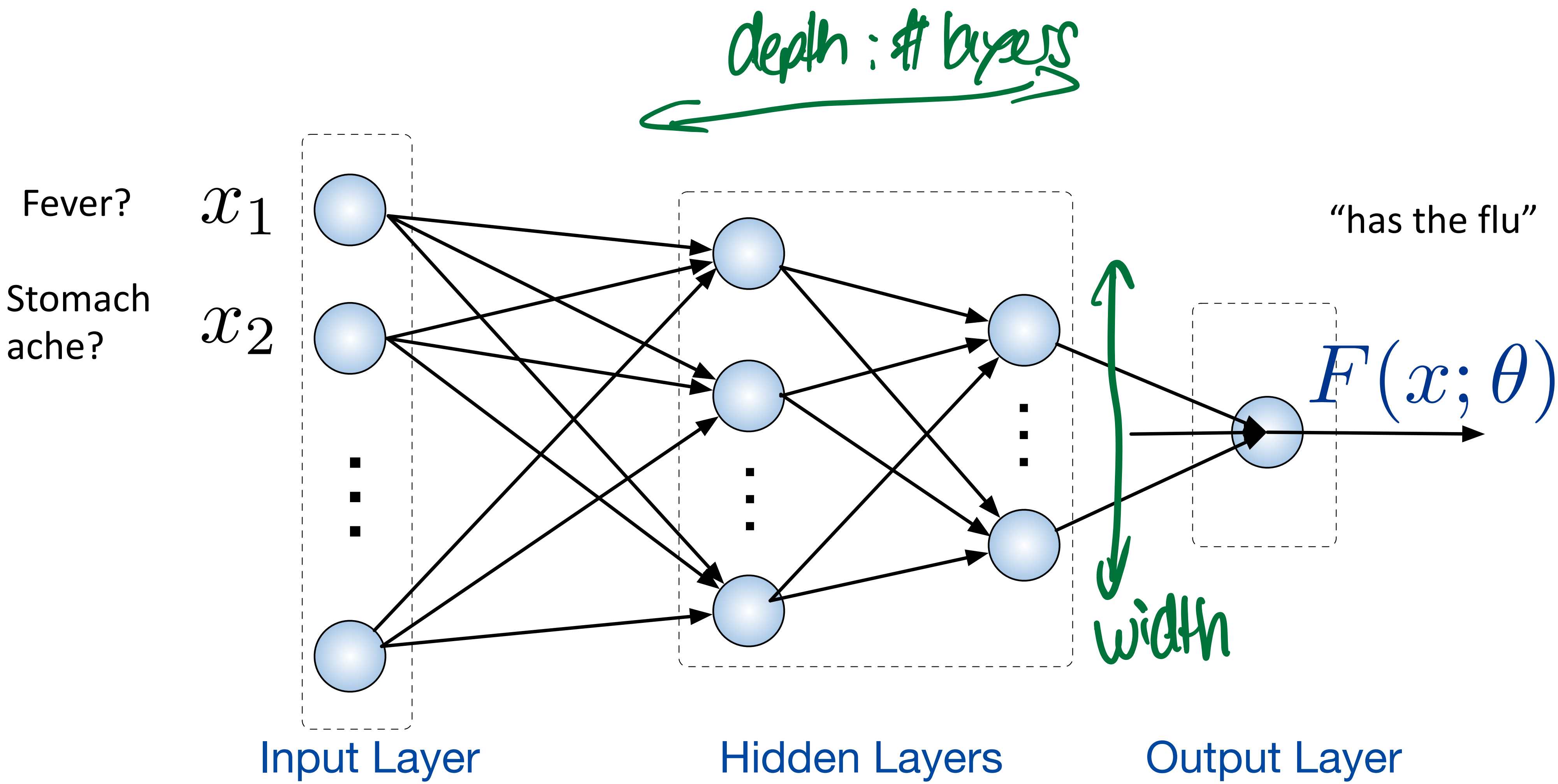
Deep Learning: reasons for success

- **Lots of data**
  - many problems can only be solved at scale
- **Computational resources** (e.g. GPUs)
  - systems that support running deep ML algorithms at scale
- **Large models are easier to train**
  - can be successfully estimated with simple gradient based algorithms
- **Flexible neural "lego pieces"**
  - common representations, diversity of architecture choices

Outline

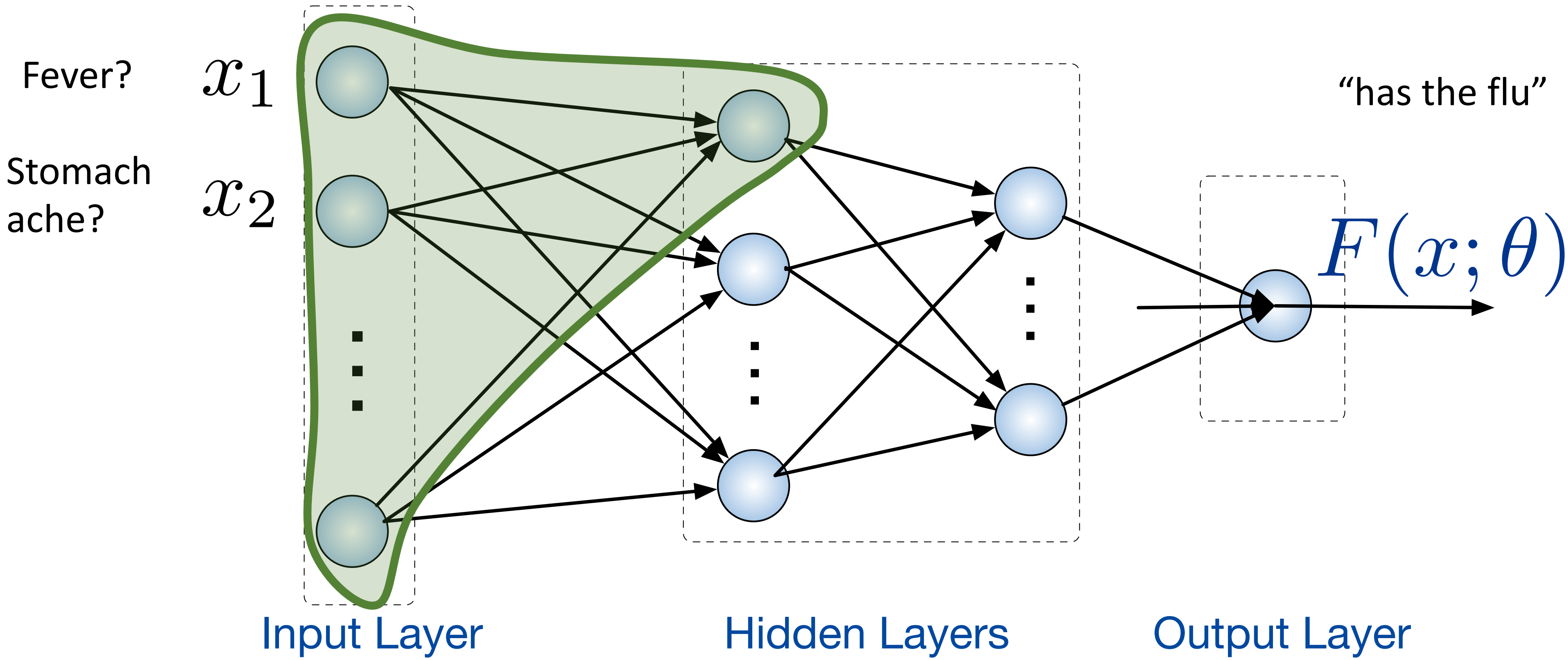
- **How do neural networks represent data?**
- How can we “train” a neural network?
- Example: Fashion MNIST

Feedforward neural networks



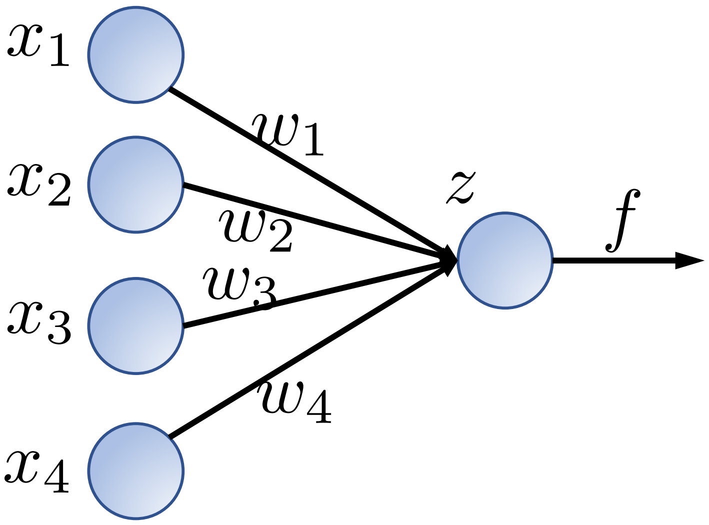
input data point $\mathbf{x}$

Feedforward neural networks



input data point $\mathbf{x}$

A unit in a neural network

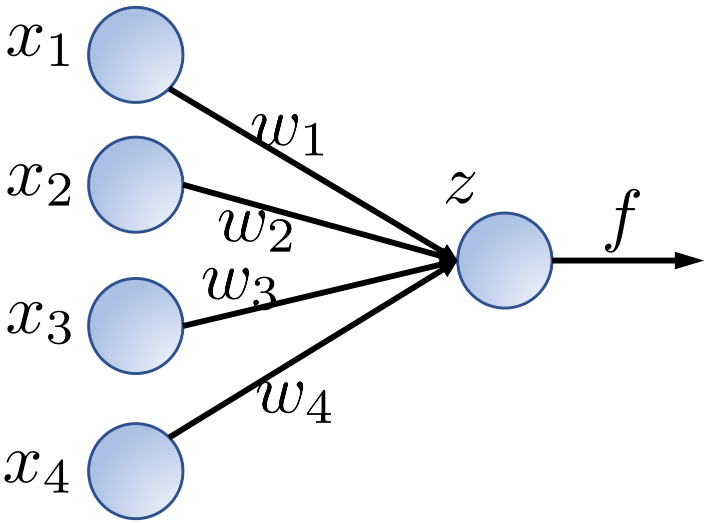


1. Weighted sum of inputs:

$$\sum_{j=1}^{d} w_j x_j = \mathbf{w}^\top \mathbf{x}$$

2. Compare to threshold.

A unit in a neural network



1. Weighted sum of inputs:

$$\sum_{j=1}^{d} w_j x_j = \mathbf{w}^\top \mathbf{x}$$

2. Compare to threshold.

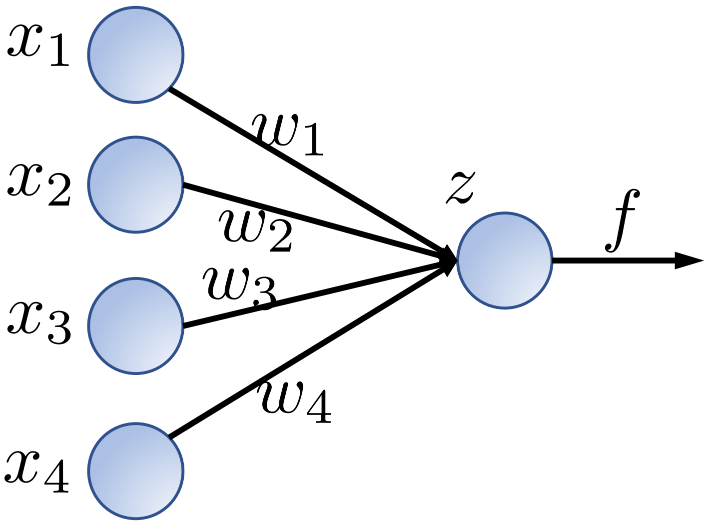
$$z = \mathbf{w}^\top \mathbf{x} + b$$

$$F(\mathbf{x};\theta) = f(z)$$

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

activation fct.

A unit in a neural network



1. Weighted sum of inputs:

$$\sum_{j=1}^{d} w_j x_j = \mathbf{w}^\top \mathbf{x}$$

2. Compare to threshold.

>0 $+$

<0

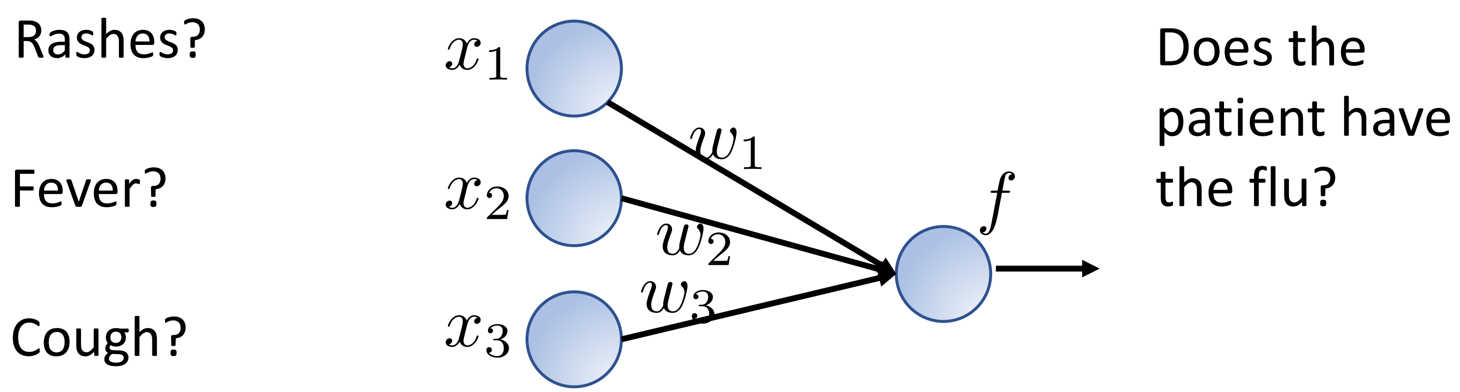
$$z = \mathbf{w}^\top \mathbf{x} + b$$

$$F(\mathbf{x}; \theta) = f(z)$$

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

just a linear classifier with parameters $\theta = (w_1, \ldots, w_d, b)$

More intuition: a simple example

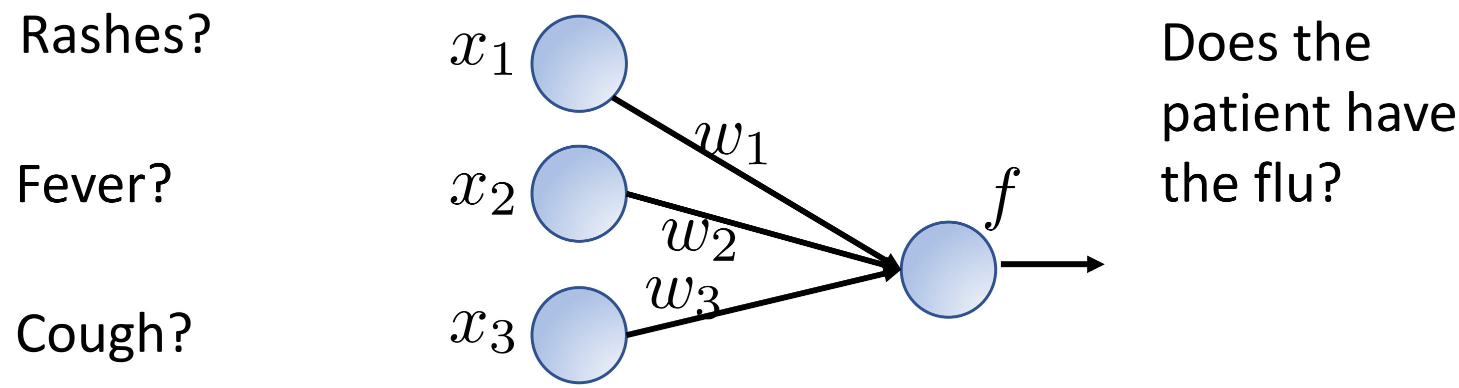


$$w_1 x_1 + w_2 x_2 + w_3 x_3 > 5 \quad ?$$

Intuition:

sum of pieces of evidence, weigh them by trust/importance

More intuition: a simple example



$$w^T x > \text{thresh}$$

$$w_1 x_1 + w_2 x_2 + w_3 x_3 > 5 \quad ?$$

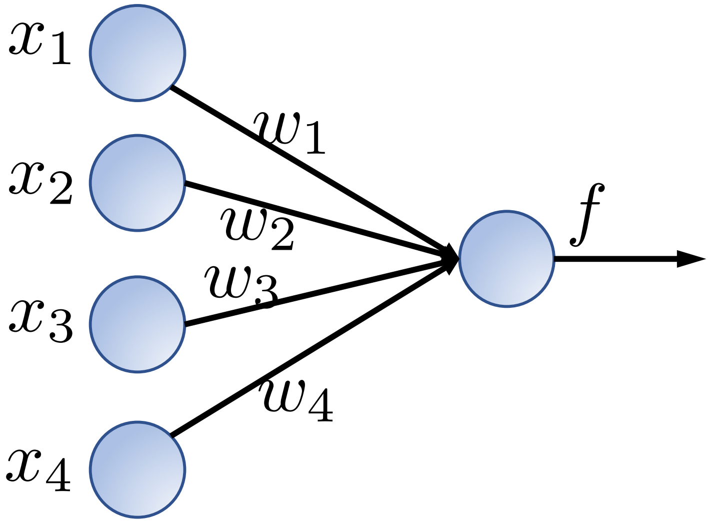
$$w^T x \underbrace{- \text{thresh}}_{+b} > 0$$

$$-1 \cdot x_1 + 3 \cdot x_2 + 3 \cdot x_3 > 5$$

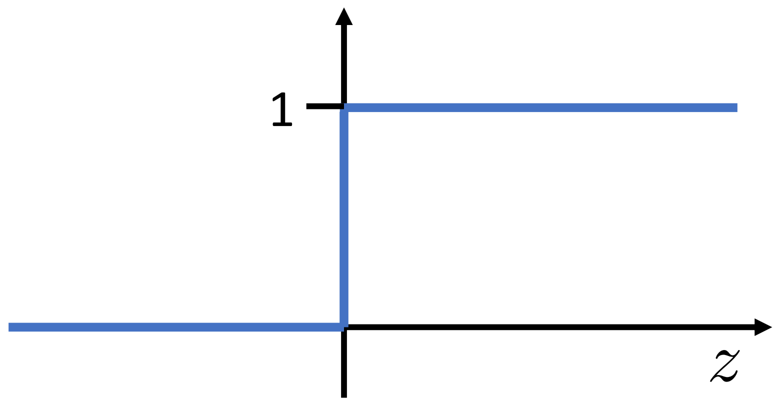
Intuition:

sum of pieces of evidence, weigh them by trust/importance

Activation functions: threshold



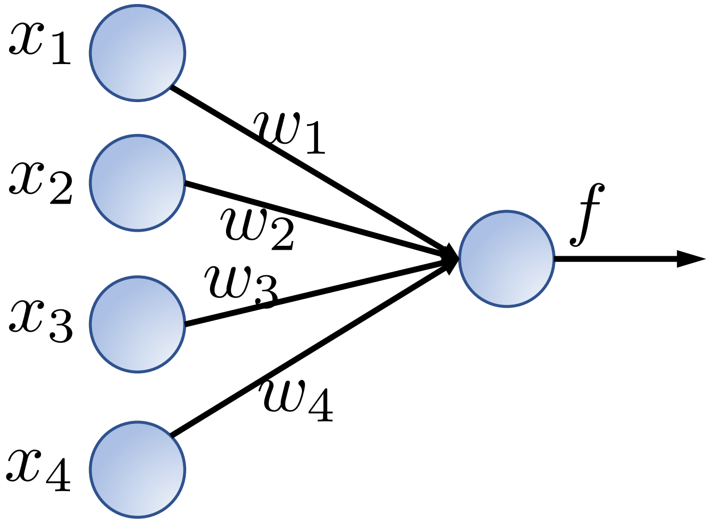
$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$



Small change in weights can cause big change in output

Not good for “learning”!

Common activation functions

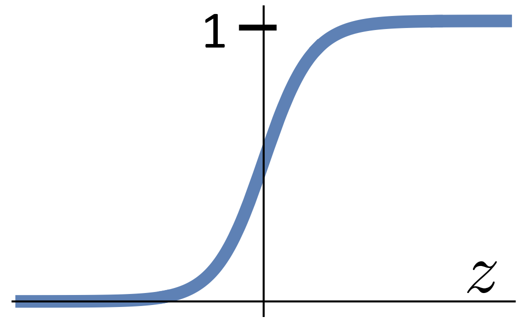


1 unit:

$z = w^T x + b$ pre-activation

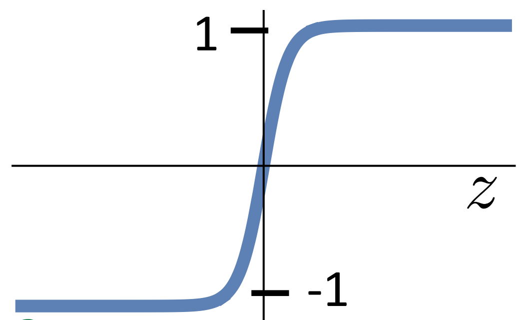
$out = f(z)$

↑ act. fct.



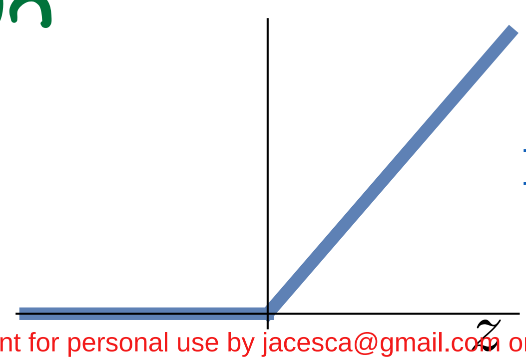
$$\frac{e^z}{1+e^z}$$

sigmoid



$$\tanh(z)$$

tanh



$$\max(0, z)$$

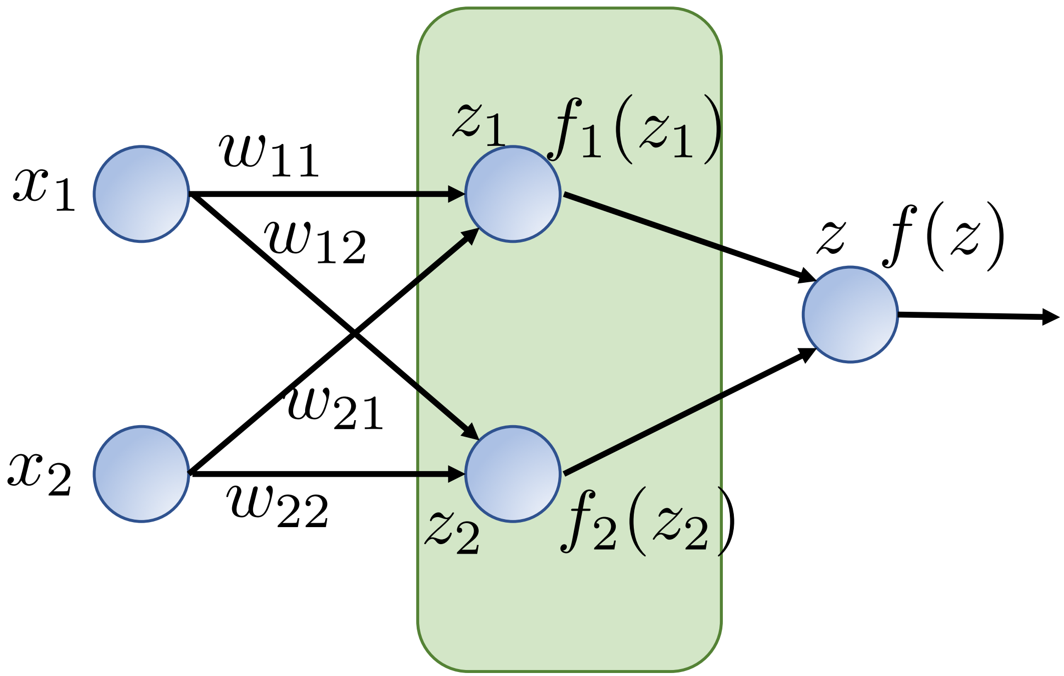
rectifier: Rectified Linear Unit (ReLU)

In summary: one neuron in a neural network…

- is just a linear classifier
- can be a simple pattern detector
- sums up weighted “evidence”

Putting things together: 1 hidden layer

- New encoding of the data: z_1, z_2

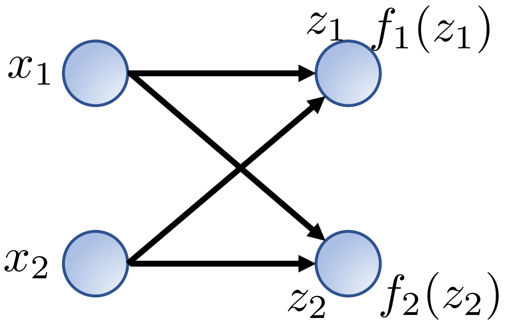


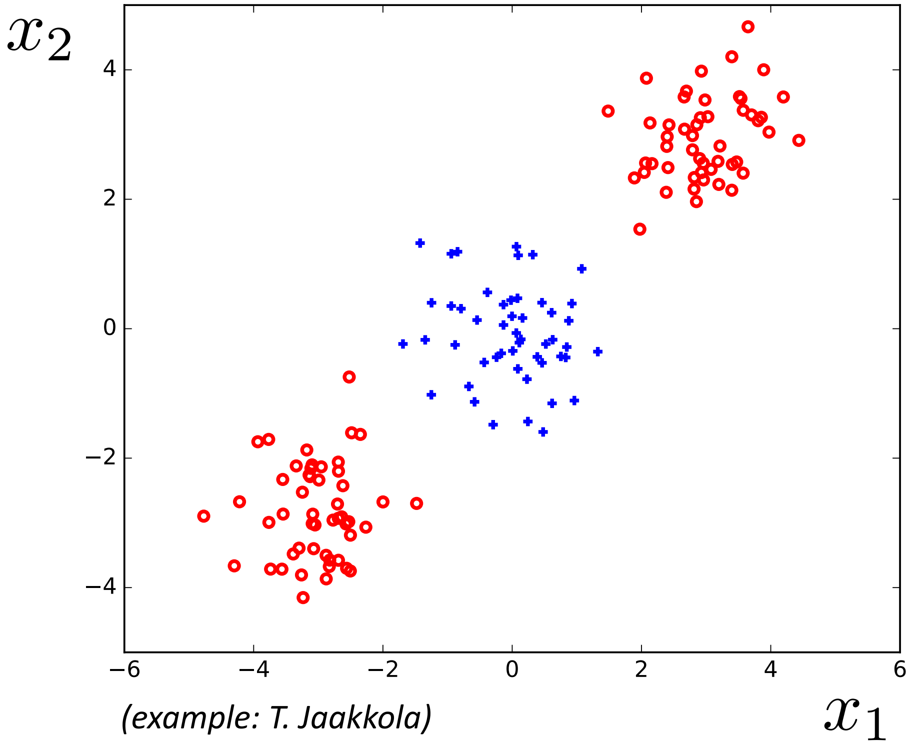
input layer

hidden layer: new encoding (representation)

output layer

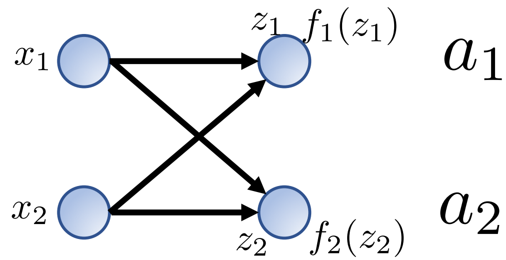
New encoding: example

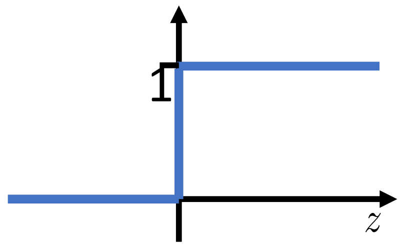


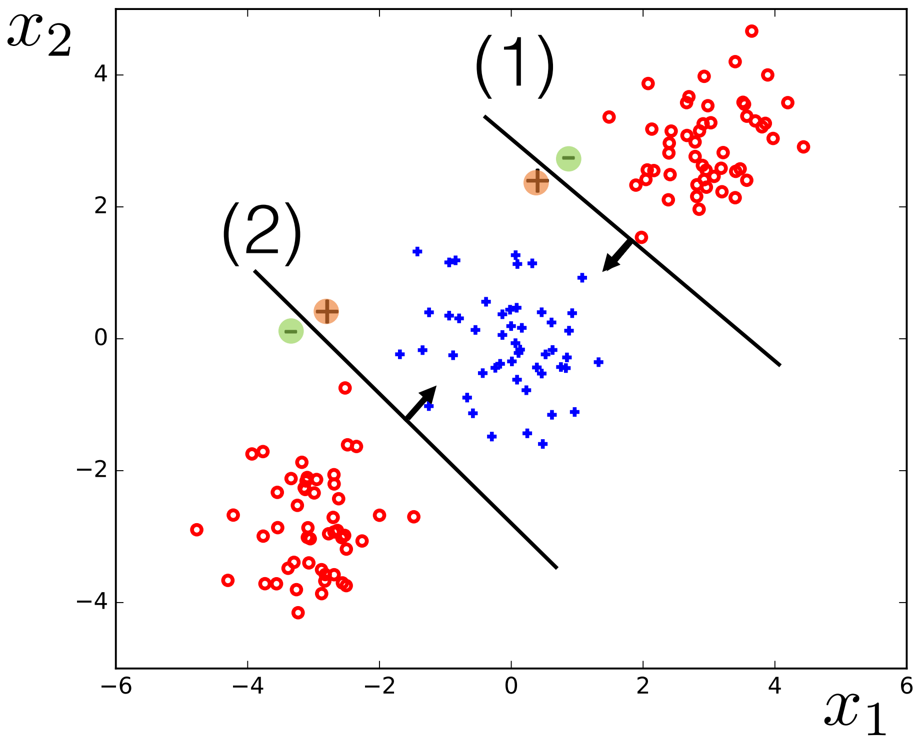


(example: T. Jaakkola)

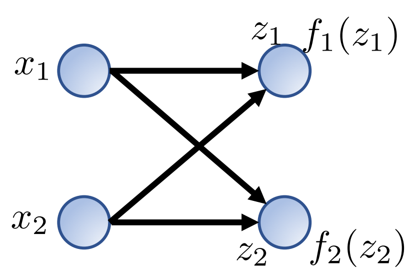
Example: step function activations





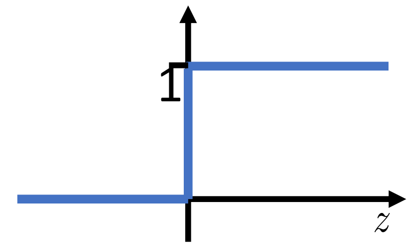


Example: step function activations

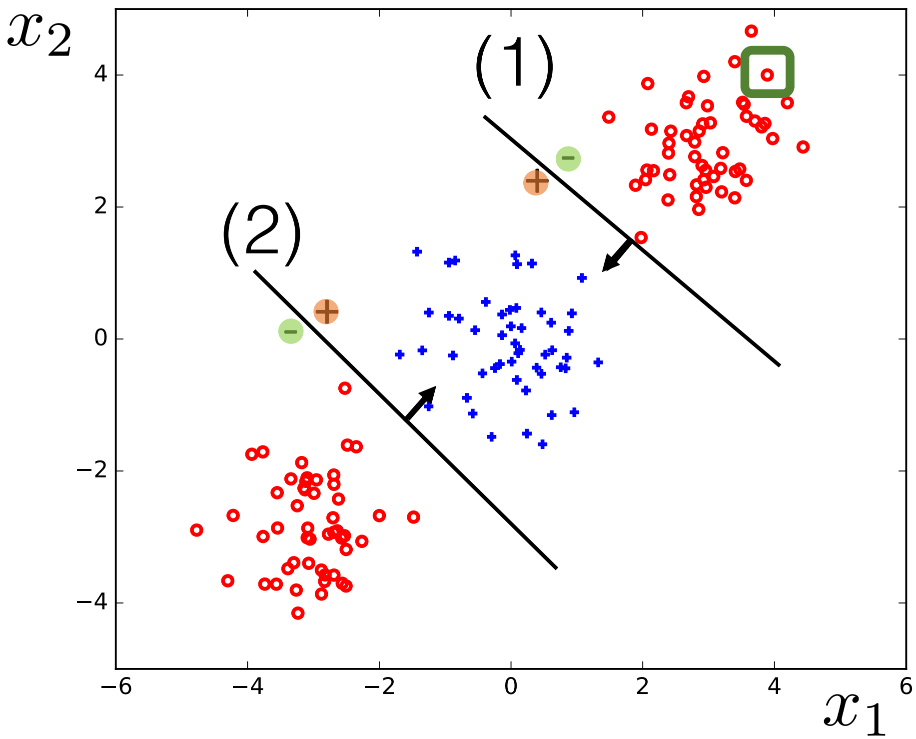


a_1 0

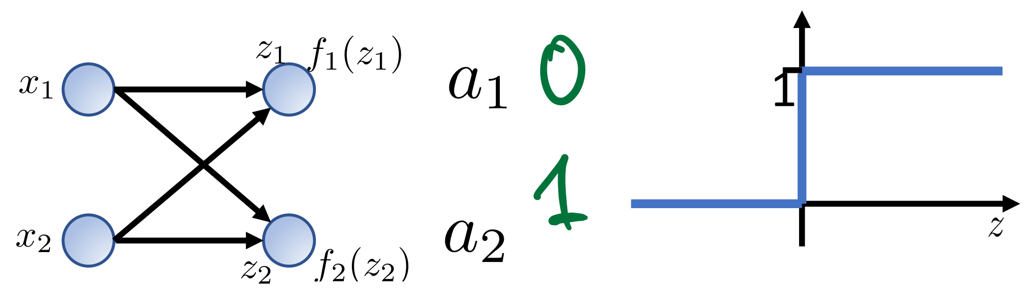
a_2 +1

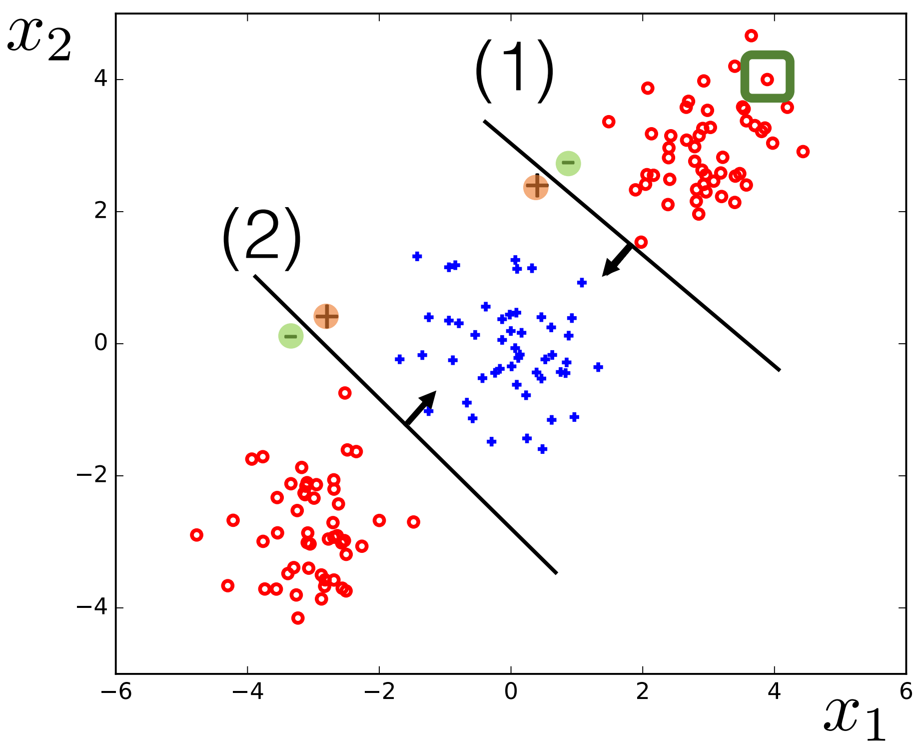


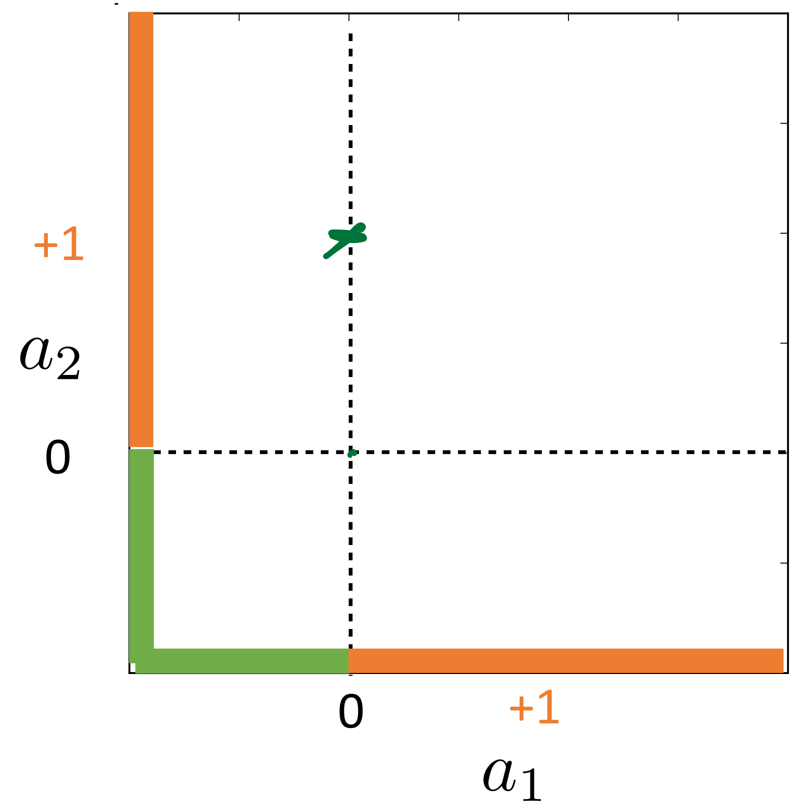
$w_1^T x + b_1 > 0$?



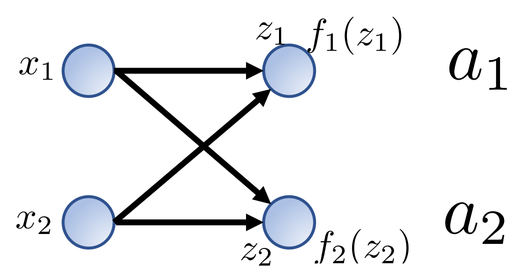
Example: step function activations

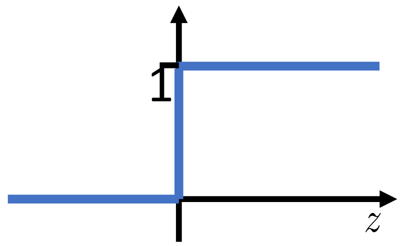


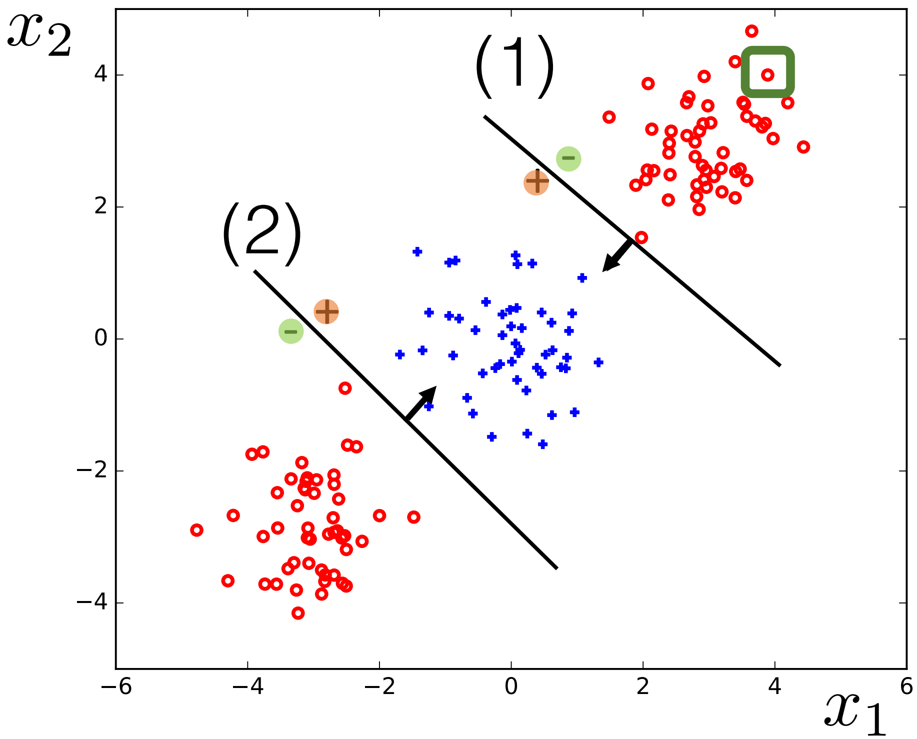


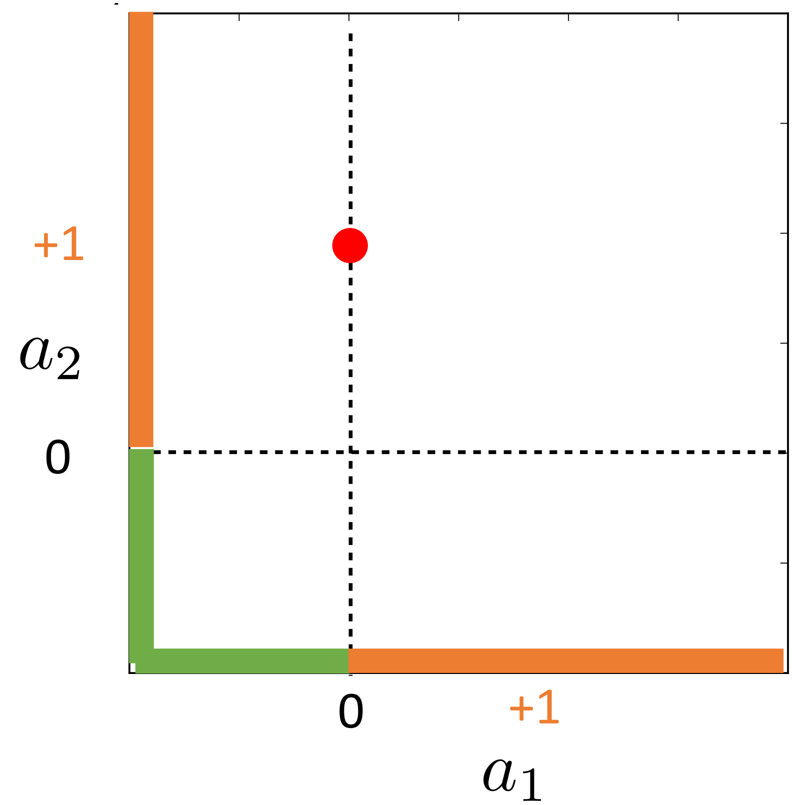


Example: step function activations

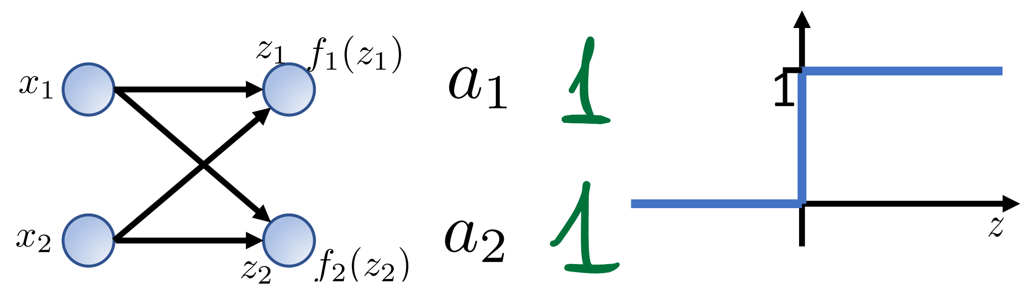


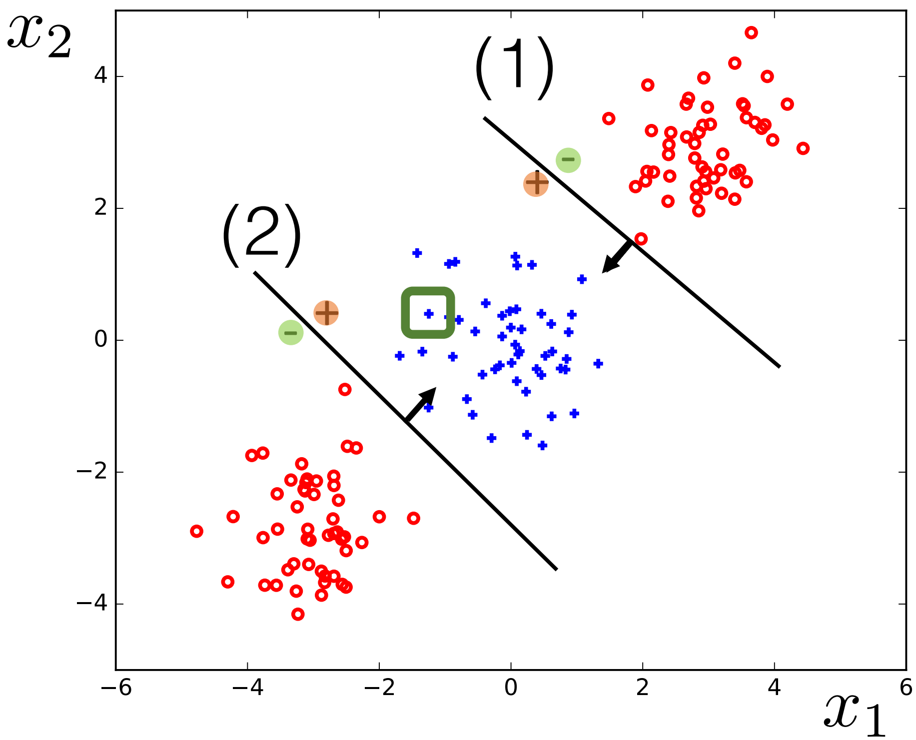


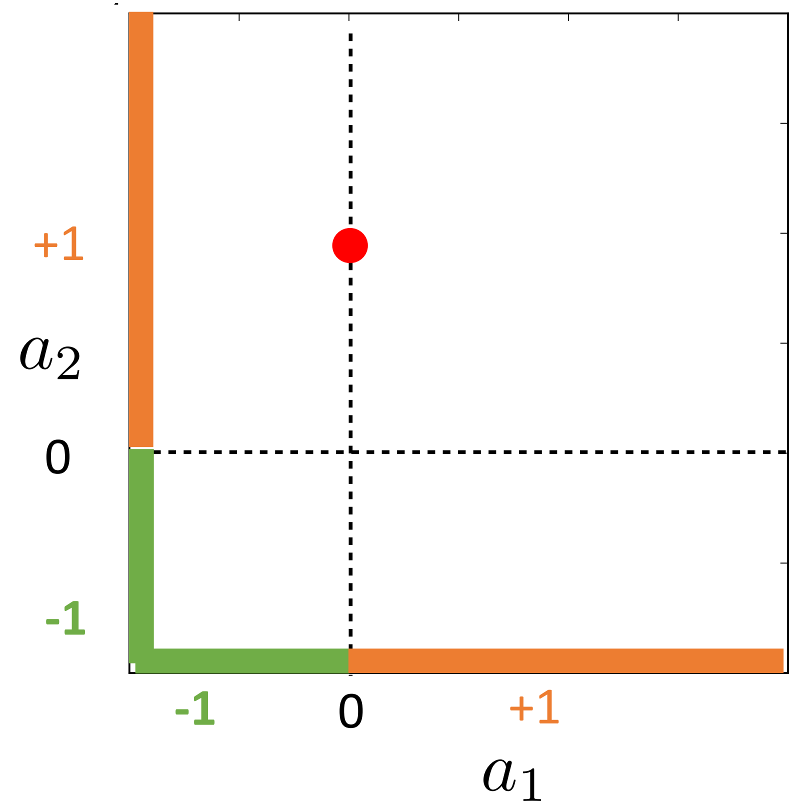




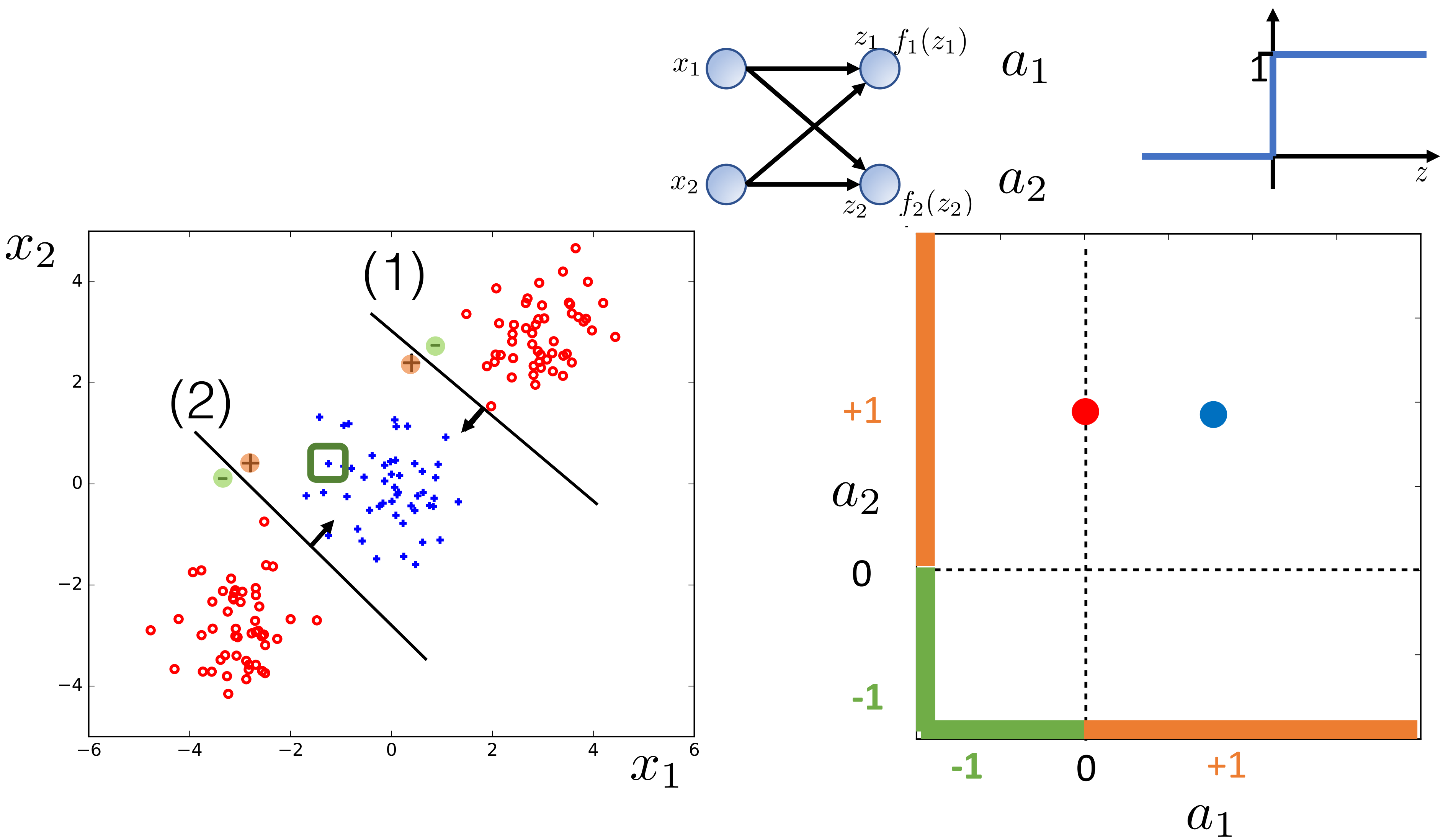
Example: step function activations



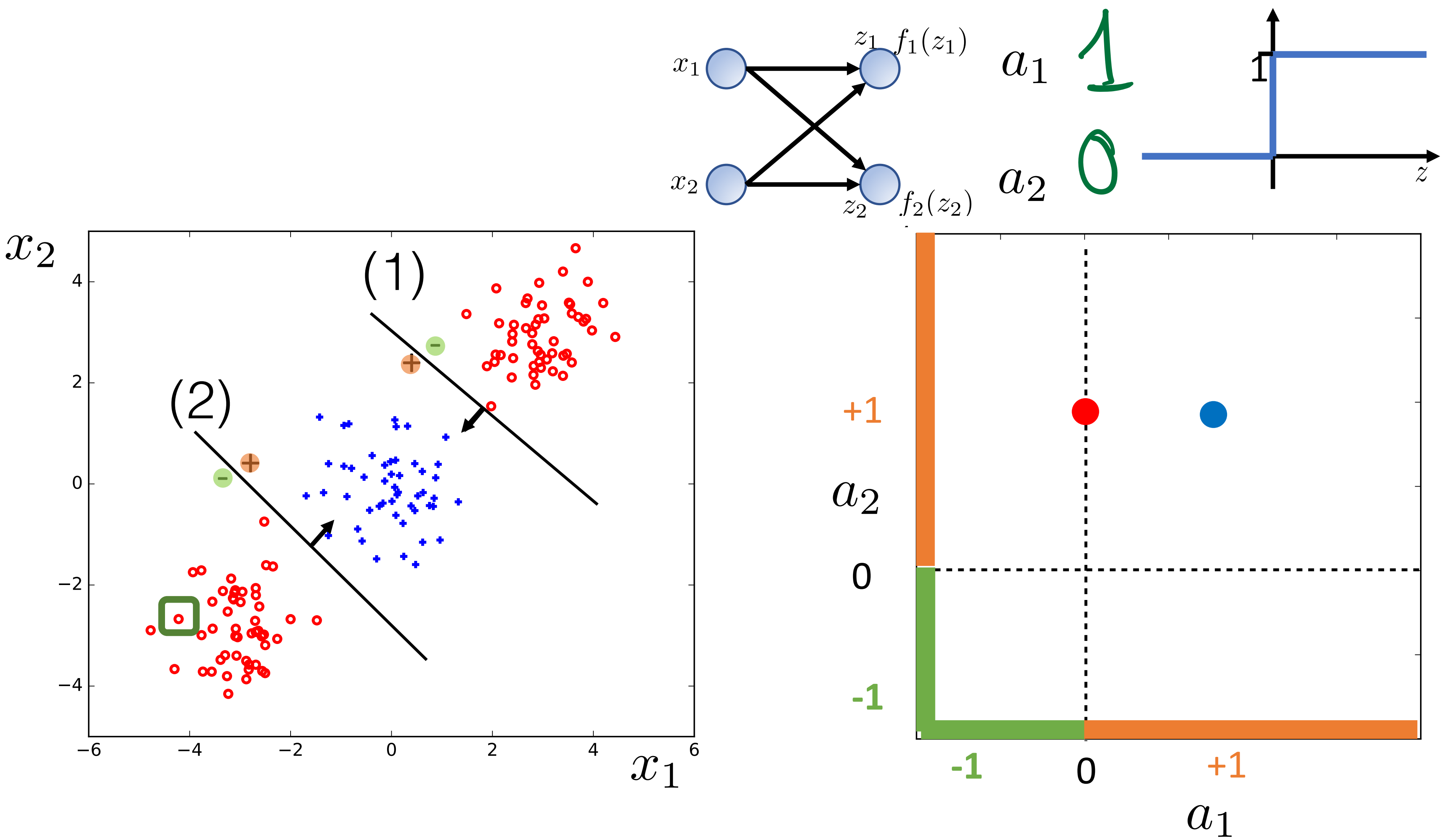




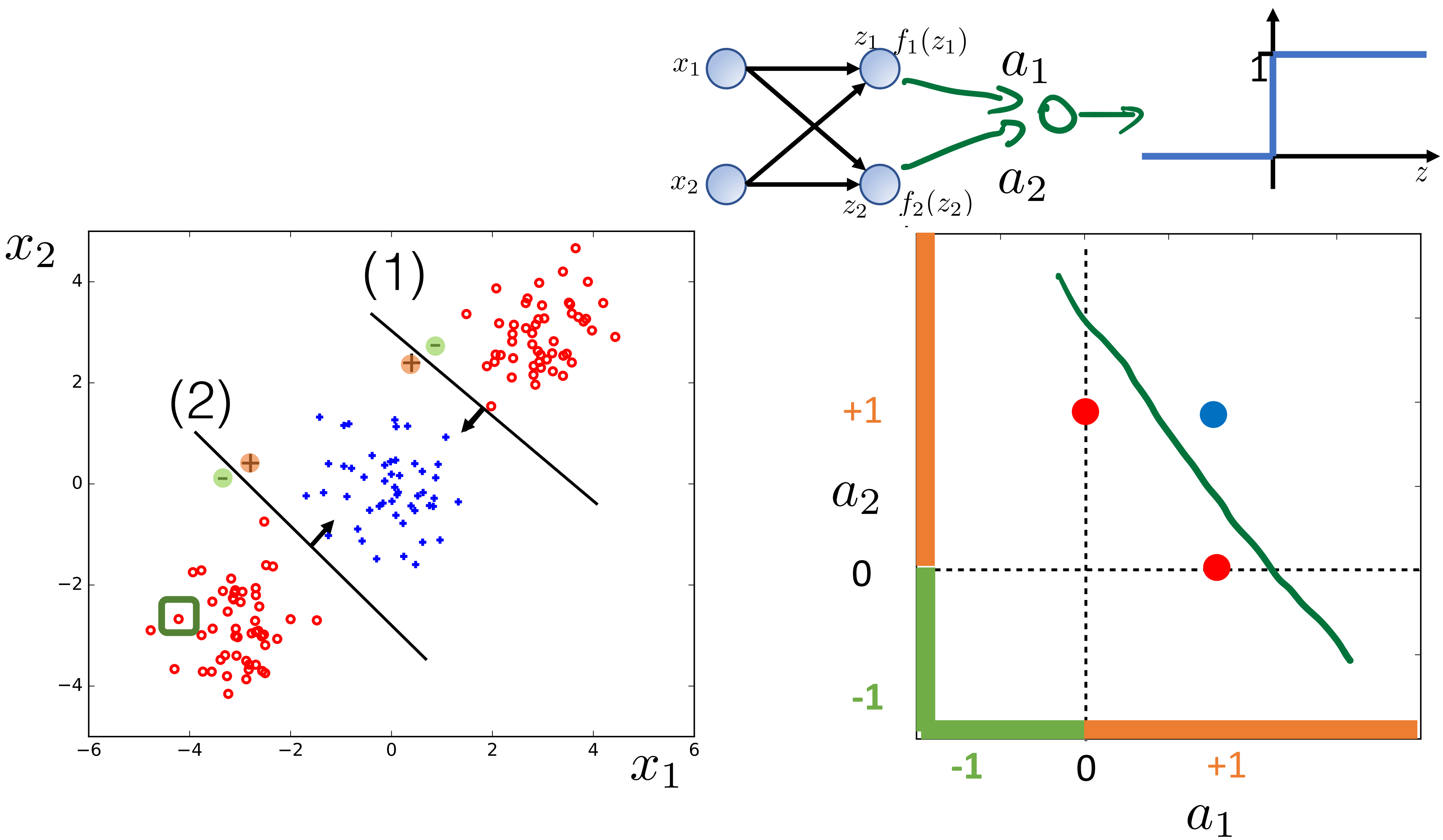
Example: step function activations



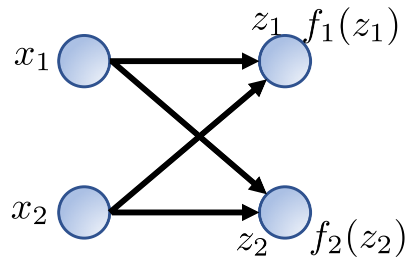
Example: step function activations

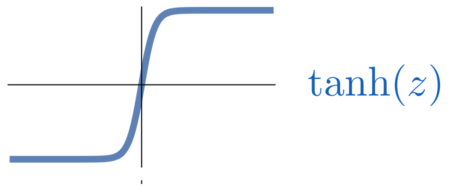


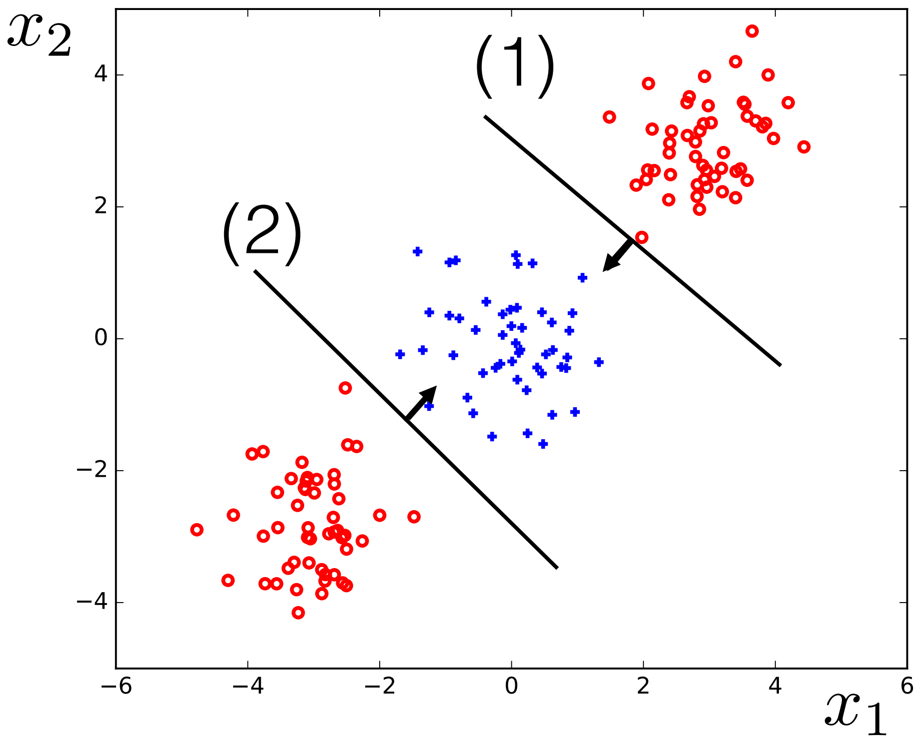
Example: step function activations

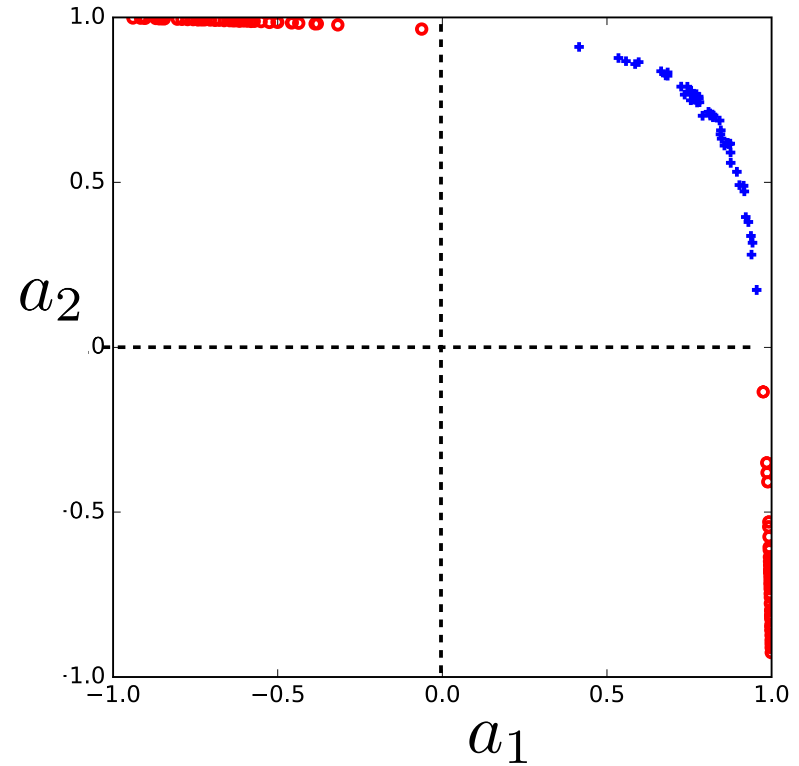


Example: tanh activations

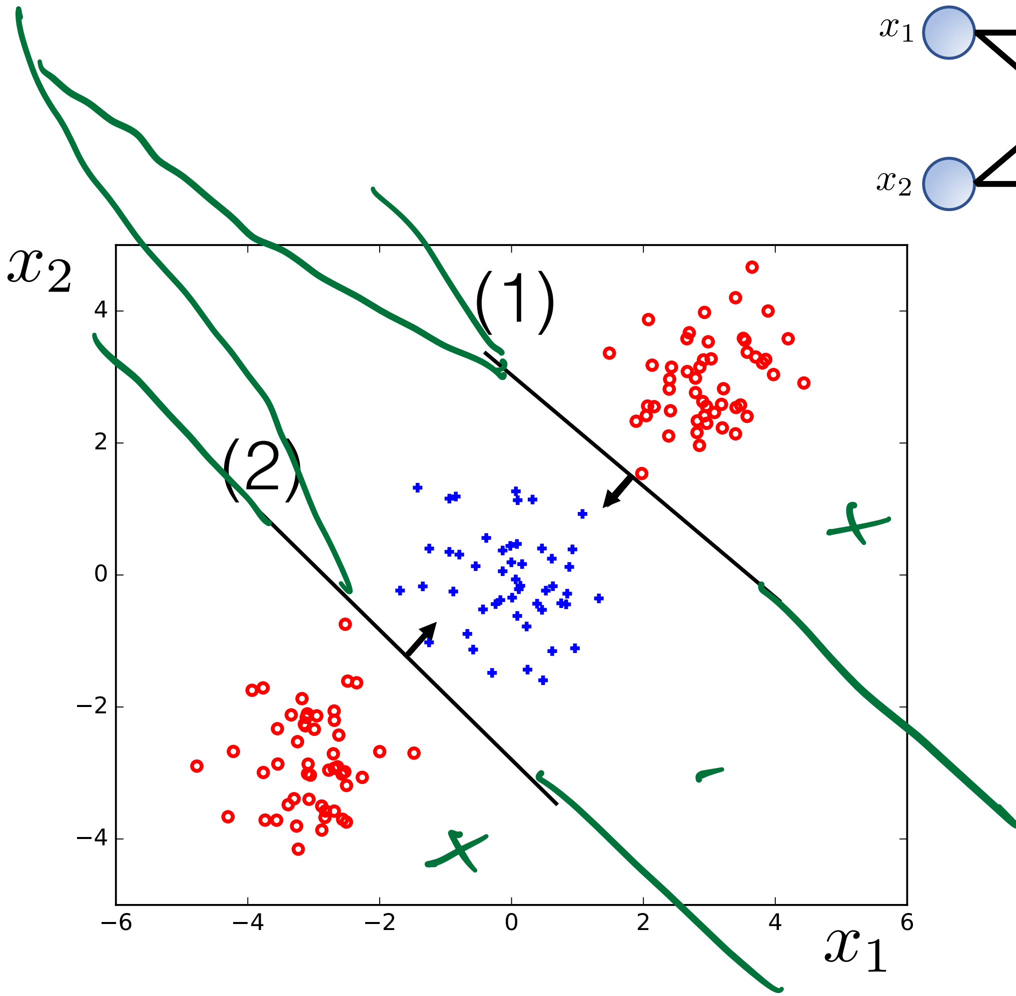


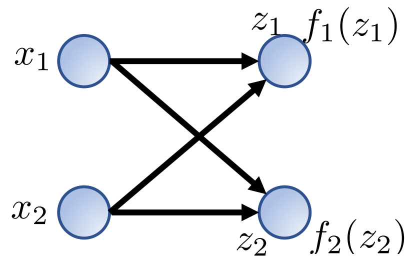


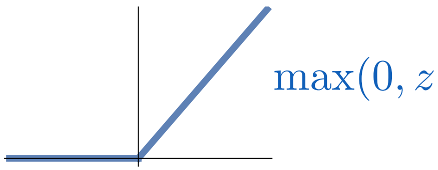


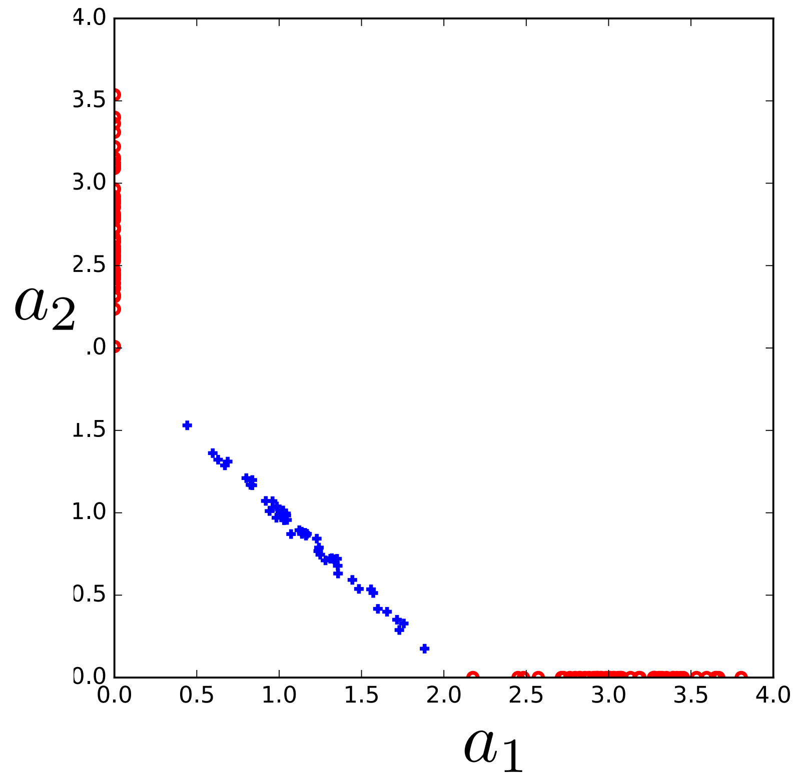


Example: ReLU activations

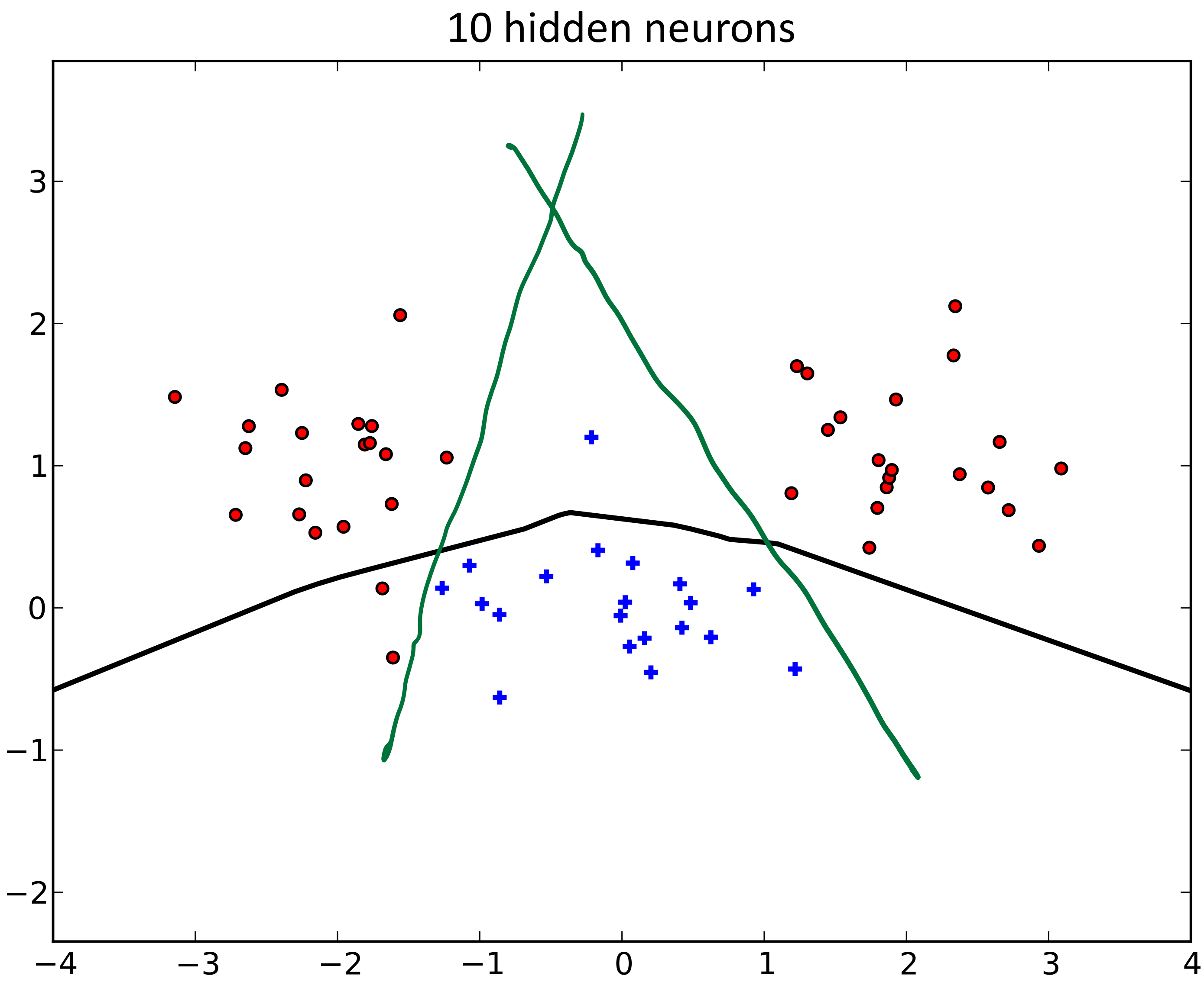






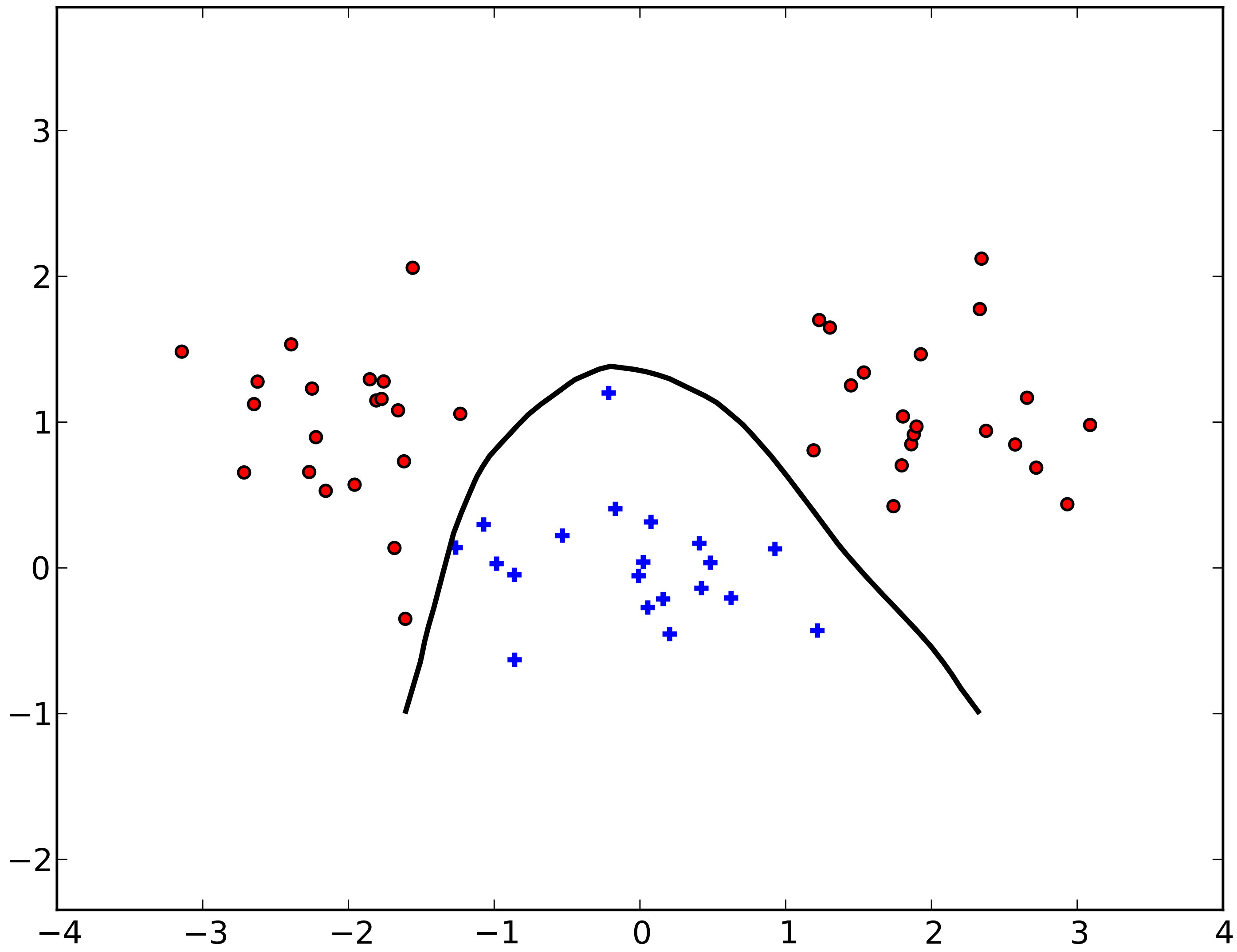


What happens with more hidden neurons?

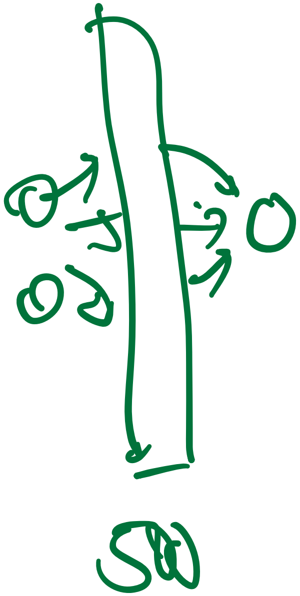


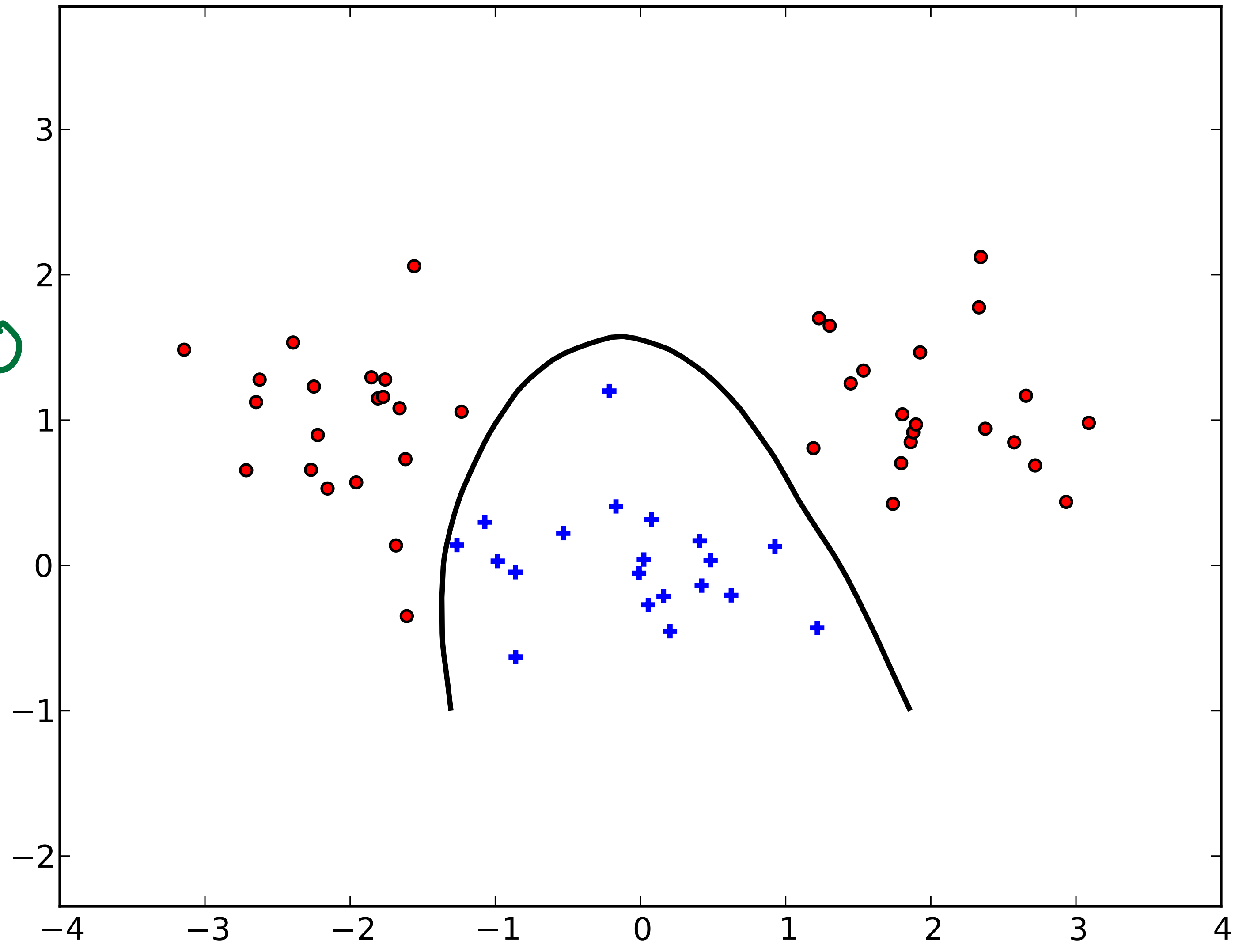
How many neurons would we need at least to separate the 2 classes?

100 hidden neurons

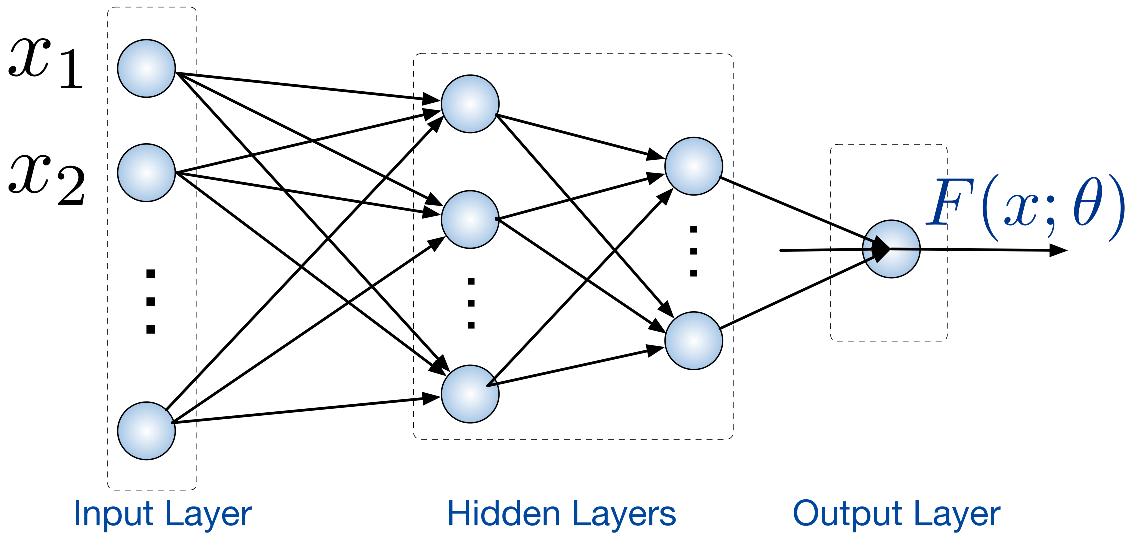


500 hidden neurons





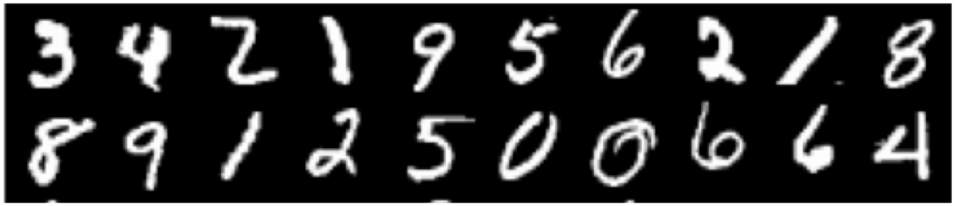
Hierarchical representations: multiple layers



Complicated decisions by breaking them down into simpler ones, e.g.:

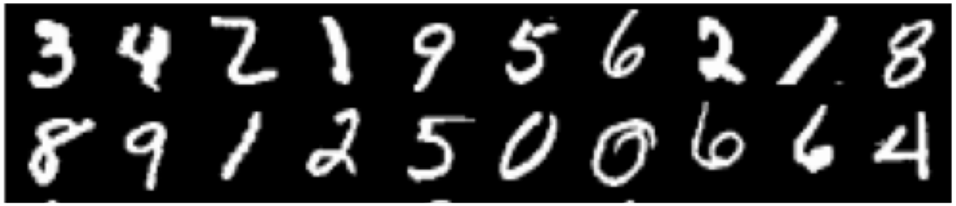
Input → edges → simple parts → parts → objects → scenes

Multi-class predictions

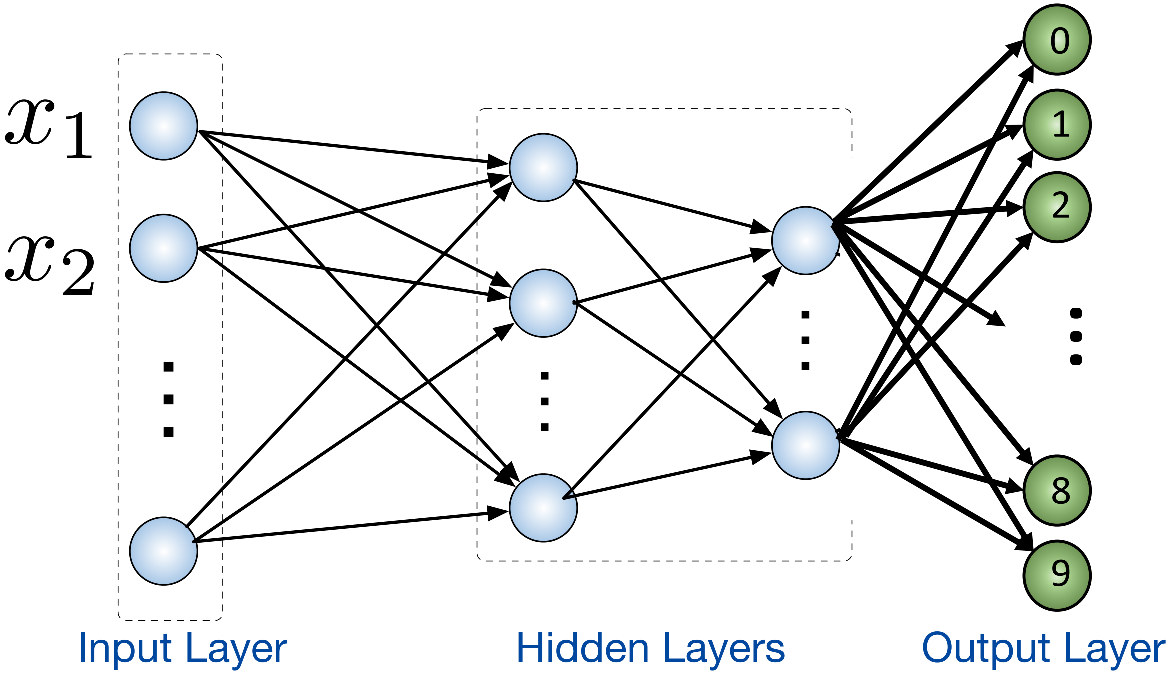


- What will the output of the network be? How encode 0—9?

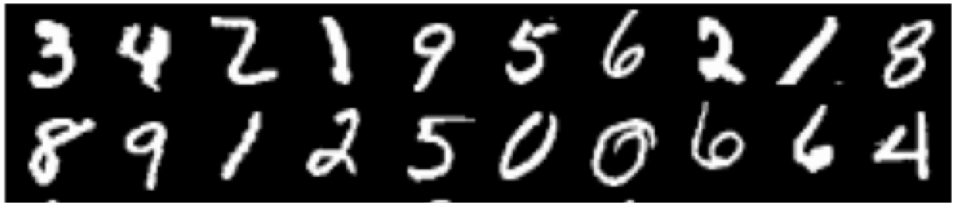
Multi-class predictions



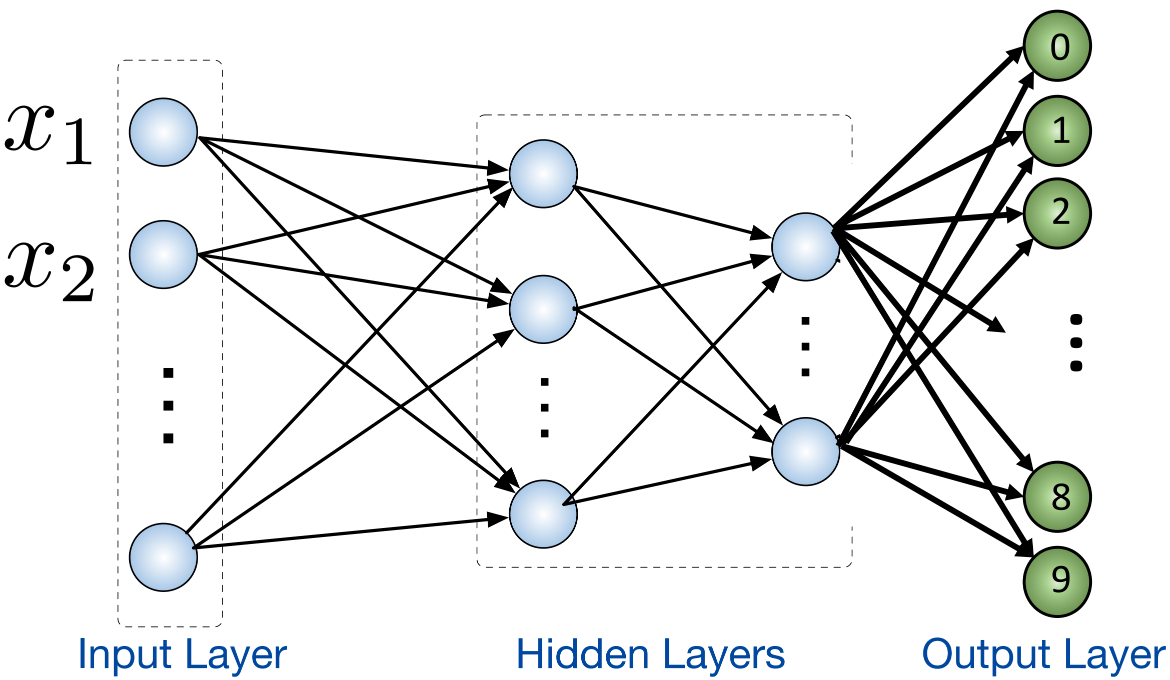
- What will the output of the network be? How encode 0—9?
- 10 output units (= 10 classifiers):
  e.g. “ideal”: 2 = [0,0,1,0,0,0,0,0,0,0]



Multi-class predictions



- What will the output of the network be? How encode 0—9?
- 10 output units (= 10 classifiers):
  e.g. "ideal": 2 = [0,0,1,0,0,0,0,0,0,0]



for probabilities:
use *softmax*
(cf logistic regression)

$$a_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)}$$

$z_j = w_j^T x + b_j$

Summary: neural representations

- Multiple simple units arranged in layers
- Each unit is a linear “classifier”
- Complicated decision by breaking it up into simpler questions
- More units often learn more easily
- Can have multiple outputs

Outline

- How do neural networks represent data?
- **How can we "train" a neural network?**
- Example: Fashion MNIST

Learning Neural Networks

- Small adjustment to weights to minimize a loss function

Common loss functions:

- Regression: squared loss

parameters

$$\mathcal{L}(\text{data}, \theta) = \frac{1}{n}\sum_{i=1}^{n}(y^i - F(\mathbf{x}^i; \theta))^2$$

label I want

network output

Learning Neural Networks

- Small adjustment to weights to minimize a loss function

Common loss functions:

- Regression: squared loss

$$\mathcal{L}(\mathrm{data}, \theta) = \frac{1}{n} \sum_{i=1}^{n} (y^i - F(\mathbf{x}^i; \theta))^2$$

- Classification: negative log-likelihood / cross-entropy *(like logistic regression)* for 2 classes:

$$\mathcal{L}(\mathrm{data}, \theta) = -\frac{1}{n} \left[\overbrace{\sum_{i:y^i=1} \underbrace{\log F(\mathbf{x}^i; \theta)}_{\mathbb{P}(y=1 \mid \mathbf{x}^j)} + \sum_{j:y^j=0} \underbrace{\log(1 - F(\mathbf{x}^j; \theta))}_{\mathbb{P}(y=0 \mid \mathbf{x}^i)}}^{\text{maximize this}} \right]$$

Gradient Descent: main idea

- **Main idea:**
  small adjustments to weights w_{ij} to decrease the loss.
- Decrease or increase w_{ij} ?
- How does a small change Δw_{ij} of weight w_{ij} change the loss?

Gradient Descent: main idea

- **Main idea:**
  small adjustments to weights w_{ij} to decrease the loss.
- Decrease or increase w_{ij} ?
- How does a small change Δw_{ij} of weight w_{ij} change the loss?

Change in loss $\approx \boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}}} \Delta w_{ij} < 0$ **Partial derivative**

Gradient Descent: main idea

- **Main idea:**
  small adjustments to weights w_{ij} to decrease the loss.
- Decrease or increase w_{ij} ?
- How does a small change Δw_{ij} of weight w_{ij} change the loss?

$$\approx \boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}}} \Delta w_{ij}$$

Partial derivative

- Hence: if $\frac{\partial \mathcal{L}}{\partial w_{ij}}$ is >0, then decrease weight
  otherwise increase weight

Gradient Descent: main idea

$i \xrightarrow{w_{ij}} j$

- **Main idea:**
  scalar
  small adjustments to weights w_{ij} to decrease the loss.
- Decrease or increase w_{ij} ?
- How does a small change Δw_{ij} of weight w_{ij} change the loss?

$$\approx \boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}}} \Delta w_{ij}$$

Partial derivative

Starting with some w_{ij}^0, iterate:

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

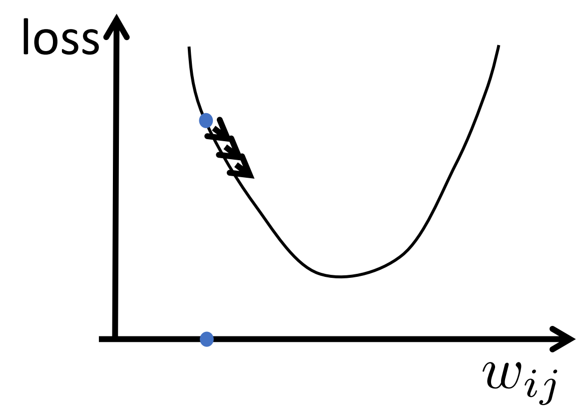
new weight

current weight

step size

learning rate

"direction"



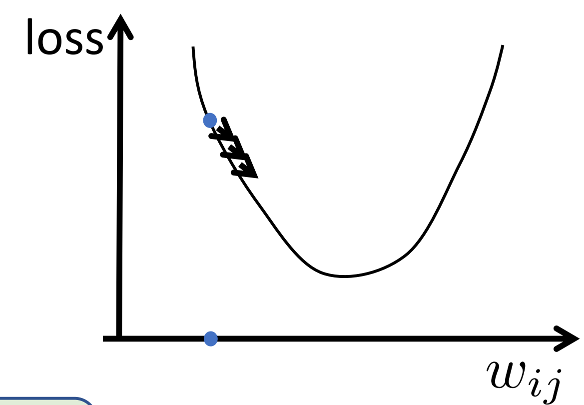
Gradient Descent: main idea

- **Main idea:**
  small adjustments to weights w_{ij} to decrease the loss.
- Decrease or increase w_{ij} ?
- How does a small change Δw_{ij} of weight w_{ij} change the loss?

$$\approx \boxed{\frac{\partial \mathcal{L}}{\partial w_{ij}}} \Delta w_{ij}$$

Partial derivative

Starting with some $\boxed{w_{ij}^0}$, iterate:



$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

new weight

current weight

step size

"direction"

Stochastic gradient descent (SGD)

- Initialize all weights w_{jk}^{0}

- For iteration $t = 1, \dots.:$

  Pick a data point $\mathbf{x}^i$ randomly
  and make the prediction for that point a bit better:

  Update weights $w_{jk}^{t+1} \leftarrow w_{jk}^{t} - \eta \frac{\partial}{\partial w_{jk}} \mathcal{L}(\mathbf{x}^i, y^i, \theta)$

Stochastic gradient descent (SGD)

- Initialize all weights w_{jk}^0

- For iteration $t = 1, \ldots.:$

  Pick a data point $\mathbf{x}^i$ randomly
  and make the prediction for that point a bit better:

  Update weights $w_{jk}^{t+1} \leftarrow w_{jk}^{t} - \eta \frac{\partial}{\partial w_{jk}} \mathcal{L}(\mathbf{x}^i, y^i, \theta)$

Variation: **Batch SGD:** use *B* instead of 1 data point and average the derivatives

Challenges

Starting with $\boxed{w_{ij}^{0}}$, iterate:

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

new weight

current weight

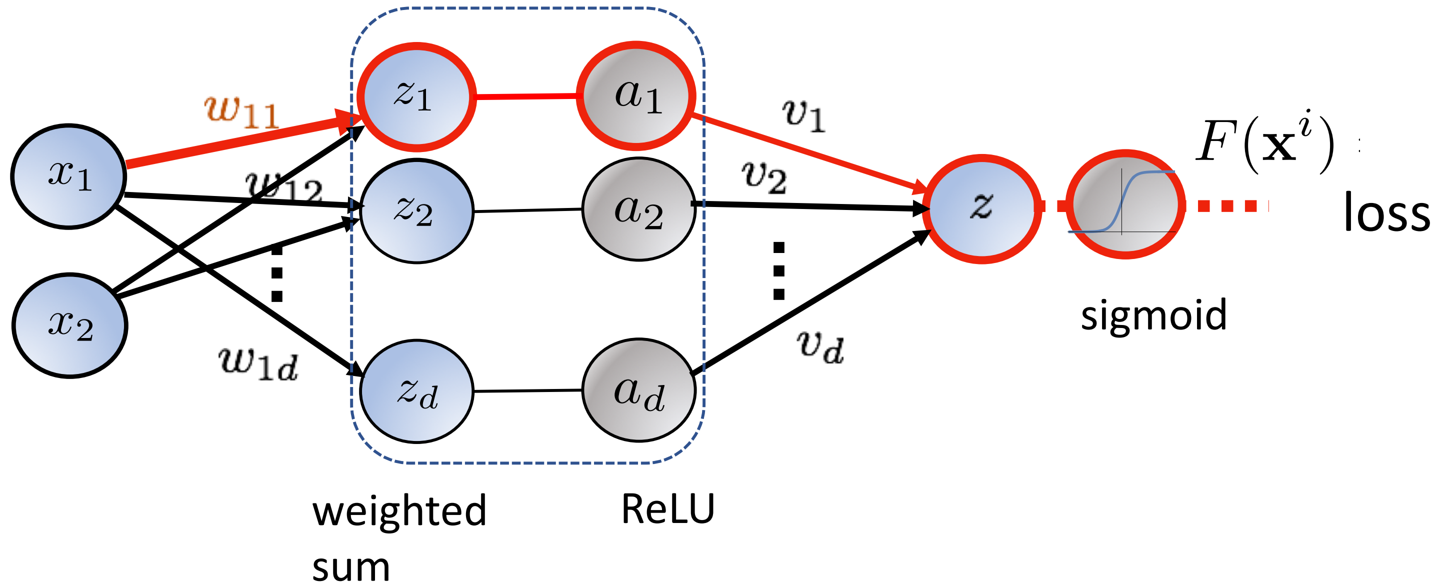
step size

“direction”

- How compute the derivatives (gradient)?
- What is a good initialization?
- What is a good step size?

Gradients for 1 hidden layer

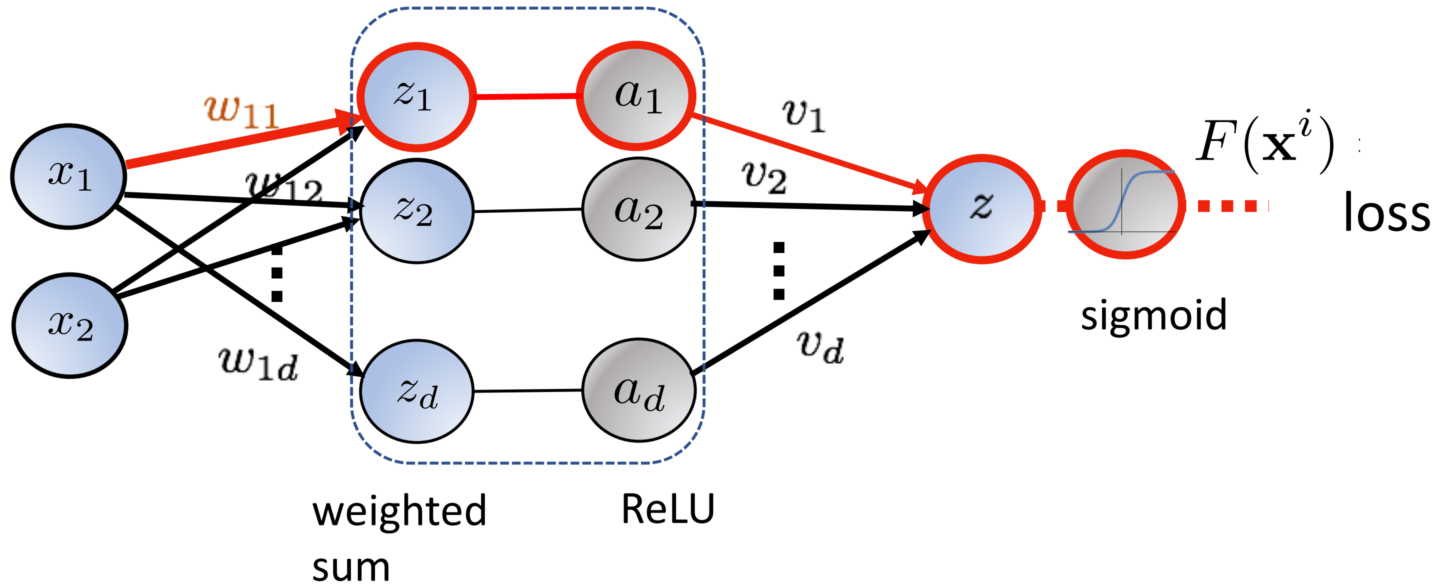
Want to update w_{11}



$$\frac{\partial}{\partial w_{11}} \mathcal{L}(\mathbf{x}^i, y^i, \theta)$$

Gradients for 1 hidden layer

Want to update w_{11}

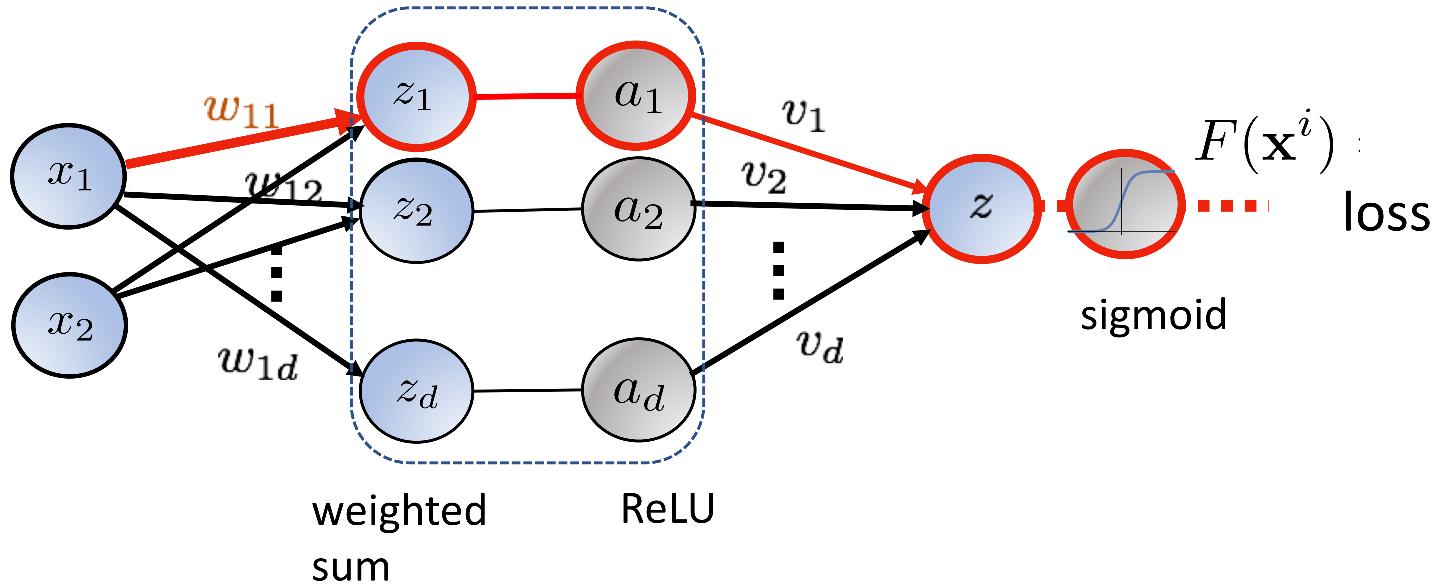


$$\frac{\partial}{\partial w_{11}}\mathcal{L}(\mathbf{x}^i, y^i, \theta) = \frac{\partial z_1}{\partial w_{11}} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z}{\partial a_1} \cdot \frac{\partial}{\partial z}\mathcal{L}$$

Chain rule of calculus

Gradients for 1 hidden layer

Want to update w_{11}



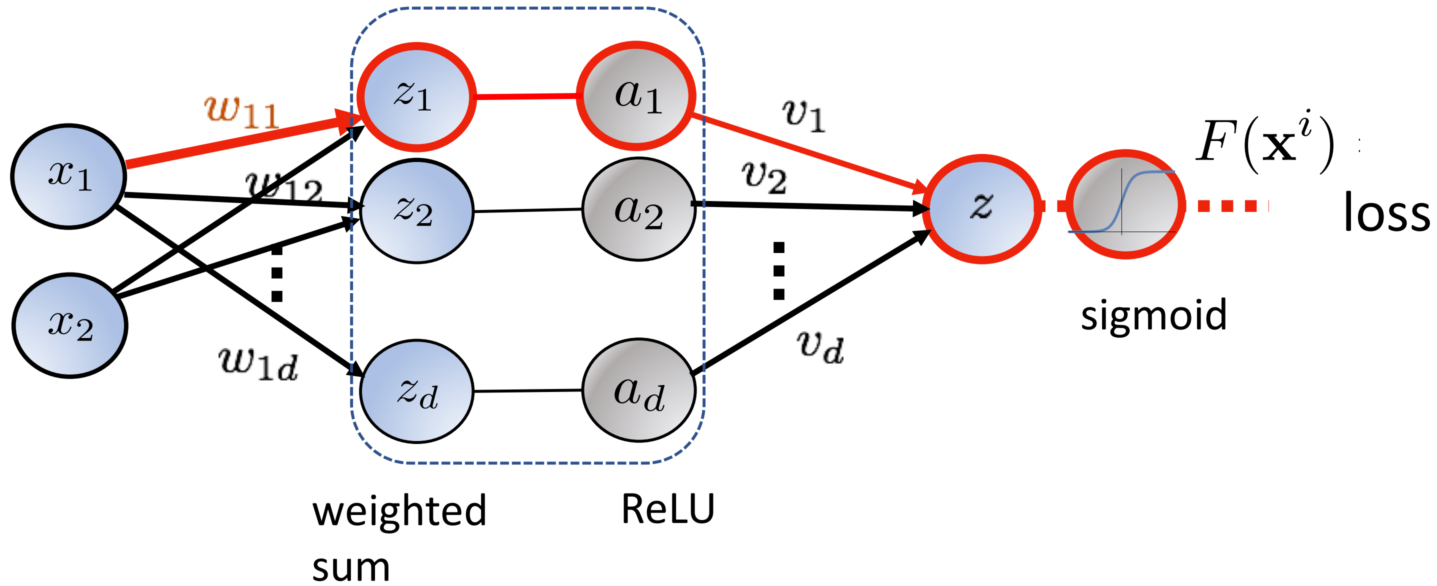
$$\frac{\partial}{\partial w_{11}}\mathcal{L}(\mathbf{x}^i, y^i, \theta) \quad = \frac{\partial z_1}{\partial w_{11}} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z}{\partial a_1} \cdot \frac{\partial}{\partial z}\mathcal{L}$$

Chain rule of calculus

$$= x_1^i \cdot \mathbf{1}[z_1 > 0] \cdot v_1 \cdot \left[F(\mathbf{x}^i) - y^i\right]$$

Gradients for 1 hidden layer

Want to update w_{11}



$$\frac{\partial}{\partial w_{11}}\mathcal{L}(\mathbf{x}^i, y^i, \theta) = \frac{\partial z_1}{\partial w_{11}} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z}{\partial a_1} \cdot \frac{\partial}{\partial z}\mathcal{L}$$

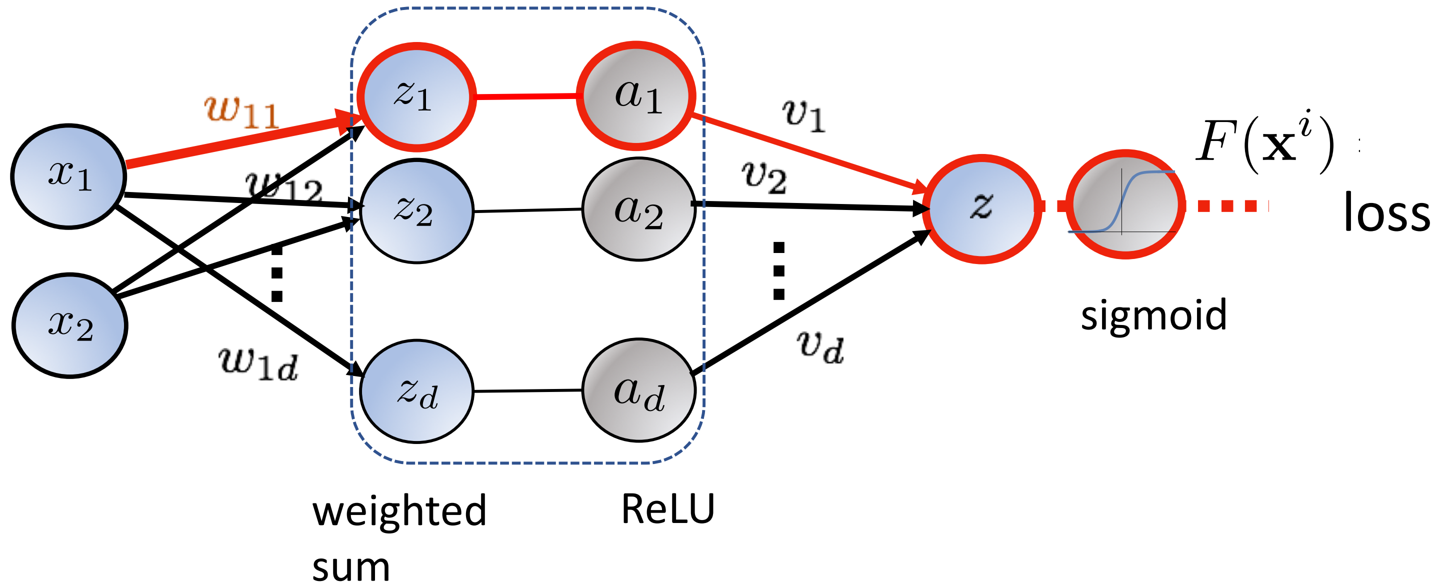
Chain rule of calculus

$$= x_1^i \cdot \mathbf{1}[z_1 > 0] \cdot v_1 \cdot \left[F(\mathbf{x}^i) - y^i\right]$$

prediction desired

Gradients for 1 hidden layer

Want to update w_{11}



$$\frac{\partial}{\partial w_{11}}\mathcal{L}(\mathbf{x}^i, y^i, \theta) = \frac{\partial z_1}{\partial w_{11}} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z}{\partial a_1} \cdot \frac{\partial}{\partial z}\mathcal{L}$$

Chain rule of calculus

$$= x_1^i \cdot \mathbf{1}[z_1 > 0] \cdot v_1 \cdot \left[F(\mathbf{x}^i) - y^i\right]$$

subsequent weight

prediction

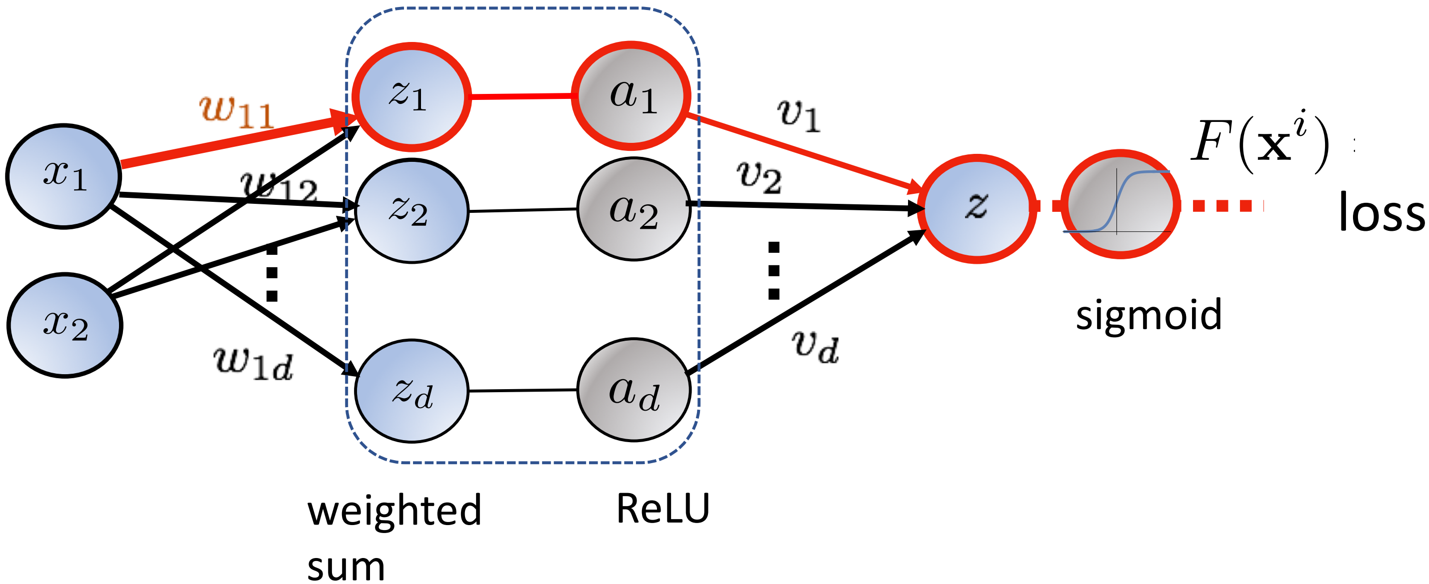
desired

$$\frac{\partial}{\partial w} f(g(w)) = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial w} = f'(g(w)) \cdot g'(w)$$

chain rule

Gradients for 1 hidden layer

Want to update w_{11}



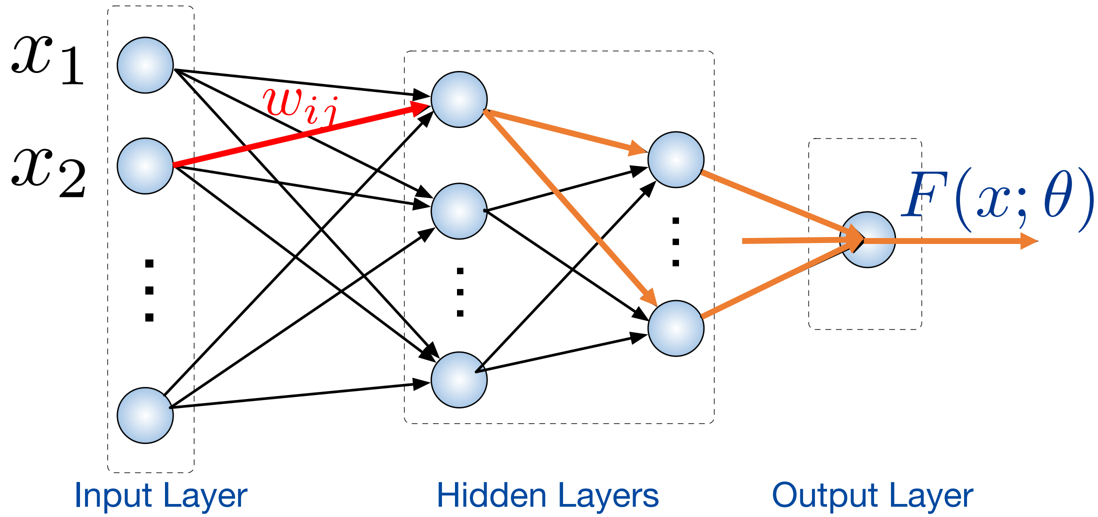
$$\frac{\partial}{\partial w_{11}}\mathcal{L}(\mathbf{x}^i, y^i, \theta) \quad = \frac{\partial z_1}{\partial w_{11}} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z}{\partial a_1} \cdot \frac{\partial}{\partial z}\mathcal{L}$$

Chain rule of calculus

$$= x_1^i \cdot \mathbf{1}[z_1 > 0] \cdot v_1 \cdot \left[F(\mathbf{x}^i) - y^i\right]$$

activation subsequent weight prediction desired

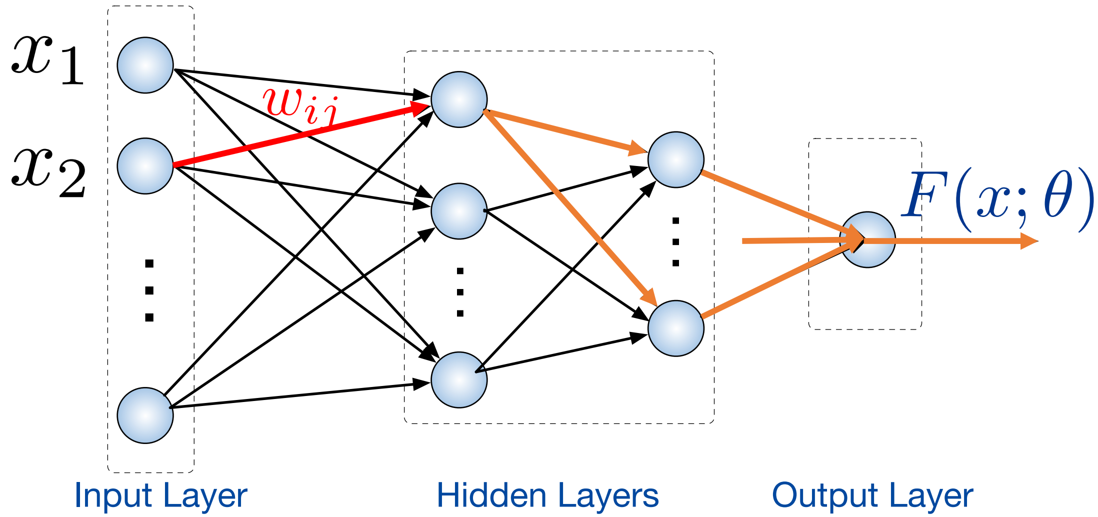
Chain rule for multiple layers



Updates scaled by:
error,
subsequent weights,
activations

- w_{ij} affects many activations downstream!

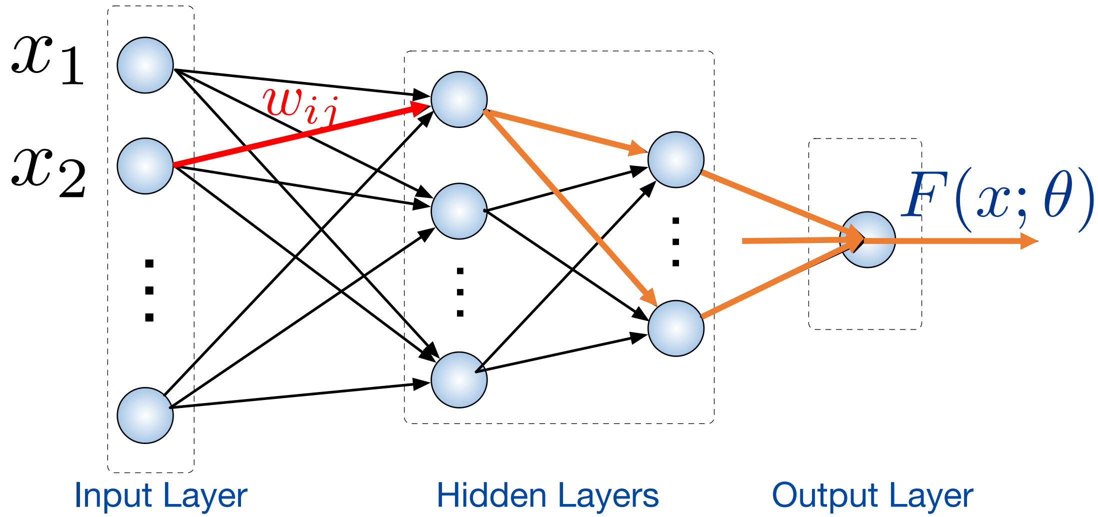
Chain rule for multiple layers



Updates scaled by:
error,
subsequent
weights,
activations

- w_{ij} affects many activations downstream!
- Still possible to compute the gradient efficiently: memorization
- **Backpropagation** *(60s,70s; popularized by Rumelhart, Hinton, Williams 1986)*
  forward pass: compute activations
  backward pass: compute gradients, store & reuse

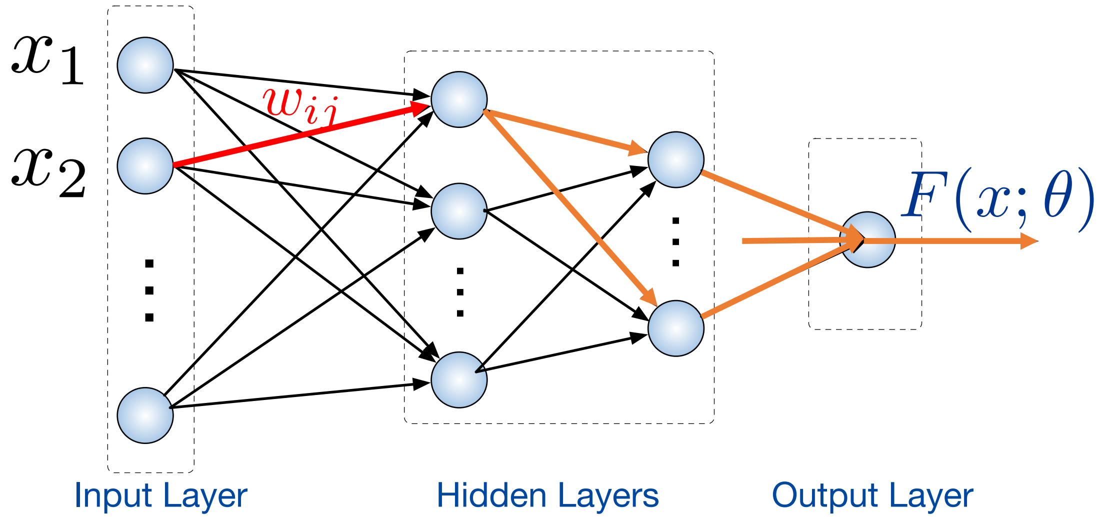
Chain rule for multiple layers



Updates scaled by:
error,
subsequent weights,
activations

- w_{ij} affects many activations downstream!
- Still possible to compute the gradient efficiently: memorization
- **Backpropagation** *(60s,70s; popularized by Rumelhart, Hinton, Williams 1986)*
  forward pass: compute activations
  backward pass: compute gradients, store & reuse
- Problem: **vanishing / exploding** gradients

Chain rule for multiple layers



Updates scaled by:
error,
subsequent
weights,
activations

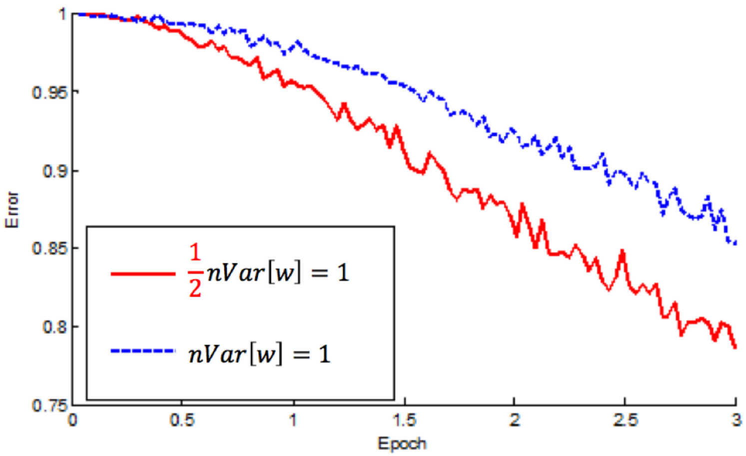
- w_{ij} affects many activations downstream!
- Still possible to compute the gradient efficiently: memorization
- **Backpropagation** *(60s,70s; popularized by Rumelhart, Hinton, Williams 1986)*
  forward pass: compute activations
  backward pass: compute gradients, store & reuse
- Problem: **vanishing / exploding** gradients

→ **Initialization** matters

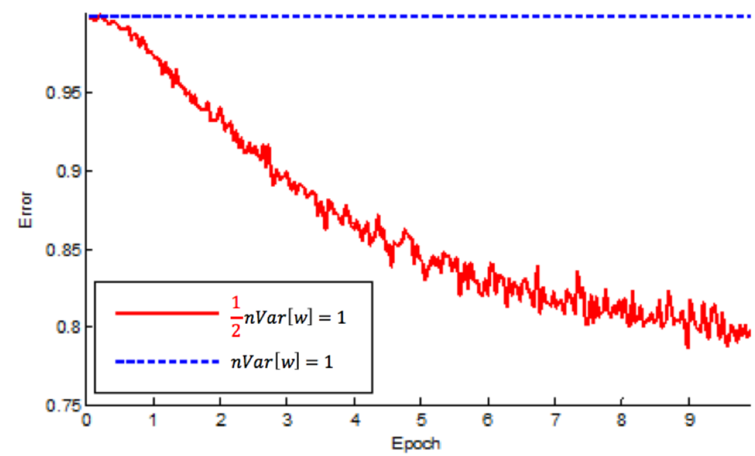
Initialization

- Typically: random, with Gaussian distribution, zero mean.
  Variance depends on #units in a layer.
  E.g. variance = 2/#units *(He initialization for ReLU)*
- Initialization can have a big impact!

22-layer ReLU net:
good init converges faster



30-layer ReLU net:
good init is able to converge

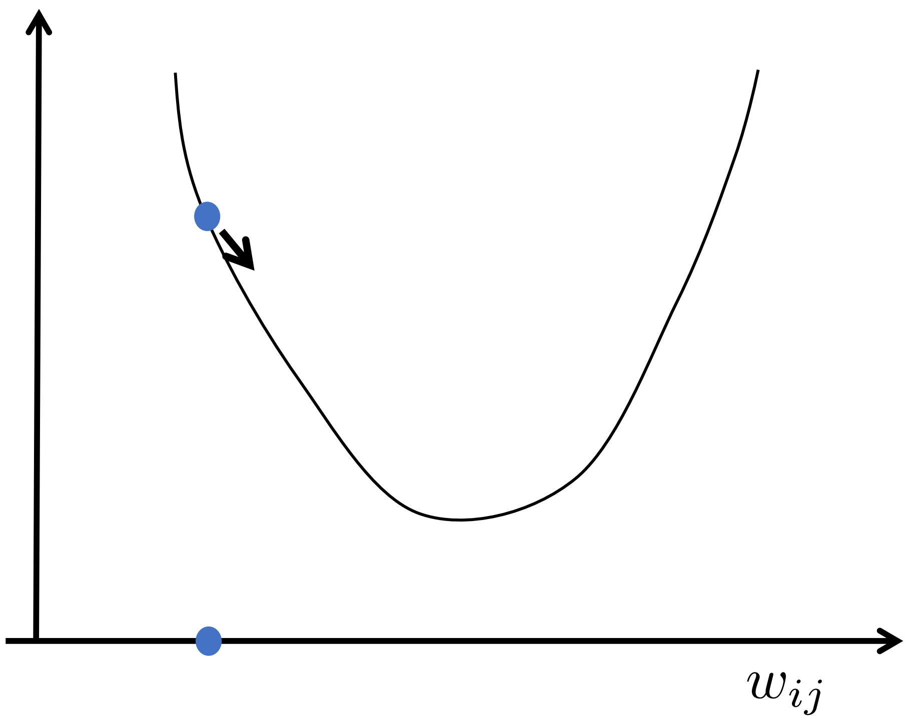


*Figures show the beginning of training

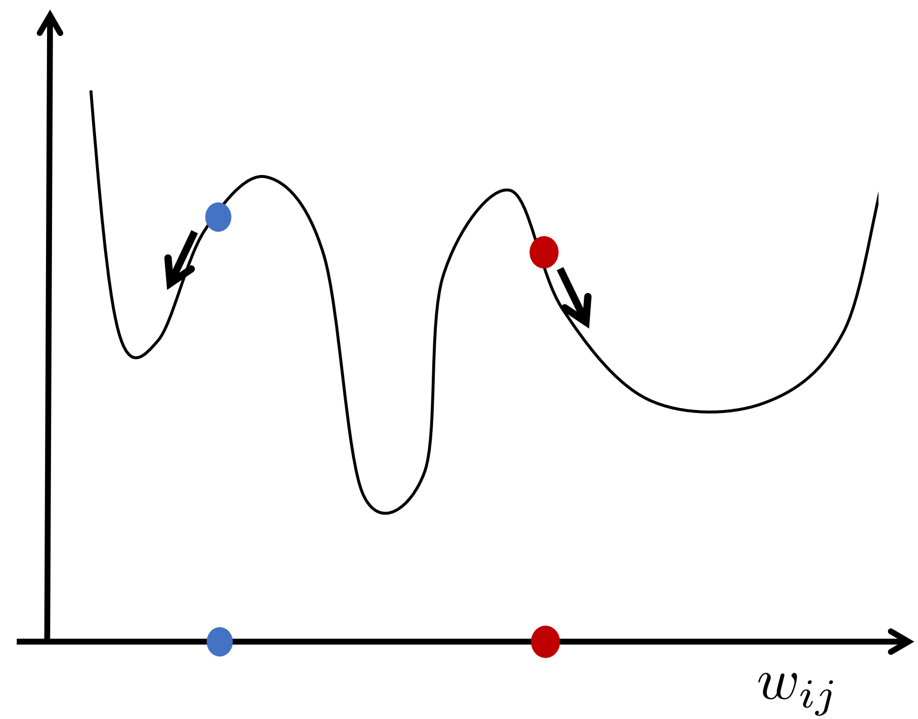
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". ICCV 2015.

Convexity and Initialization

Convex function



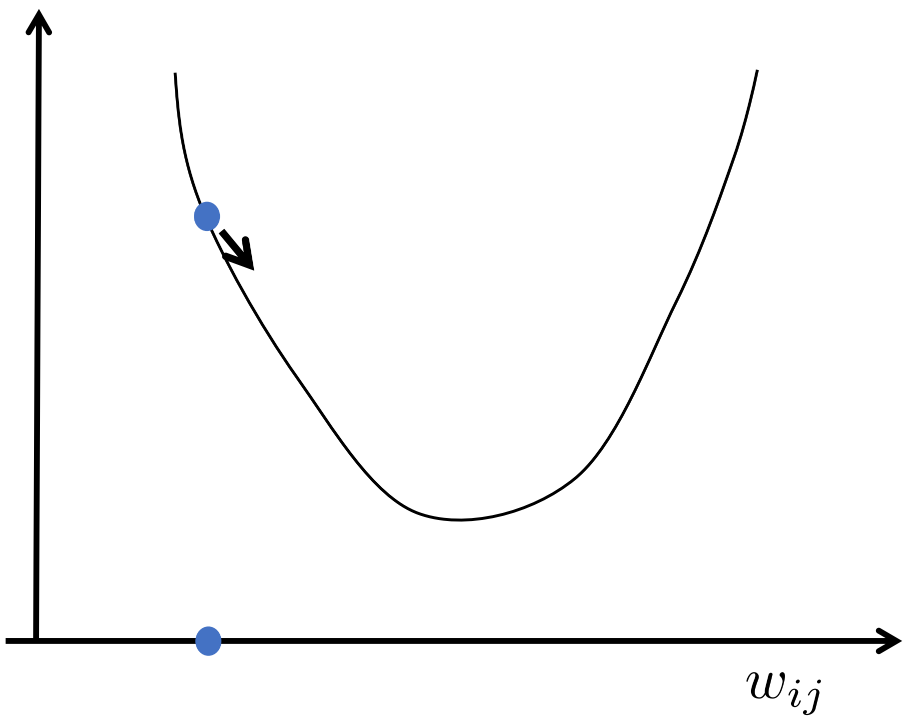
Nonconvex function (neural networks)



- “one valley”: global minimum
- with the right step size, will always reach minimum

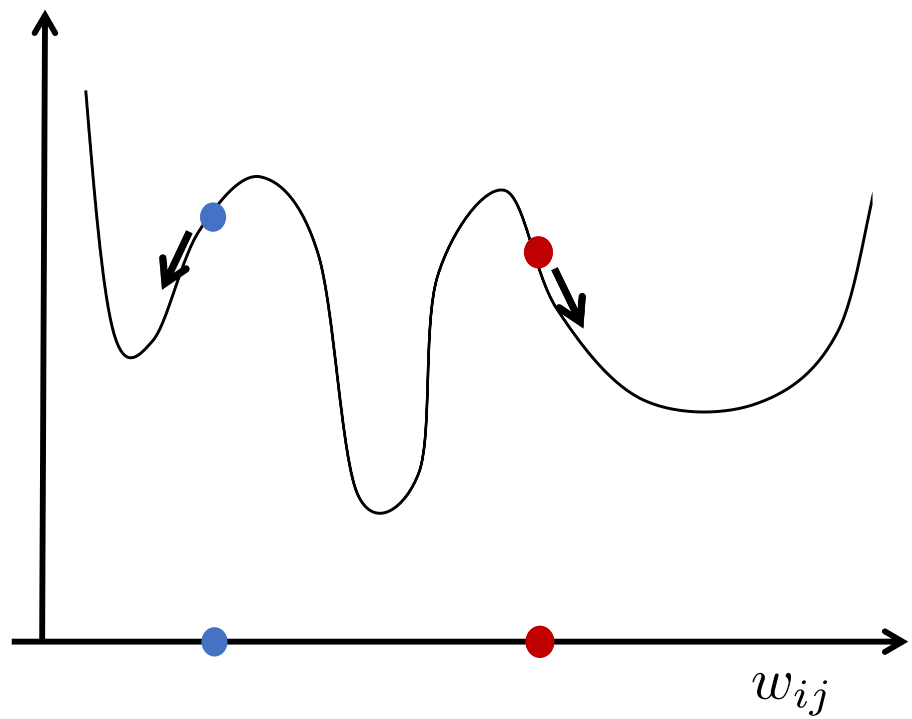
Convexity and Initialization

Convex function



- “one valley”: global minimum
- with the right step size, will always reach minimum

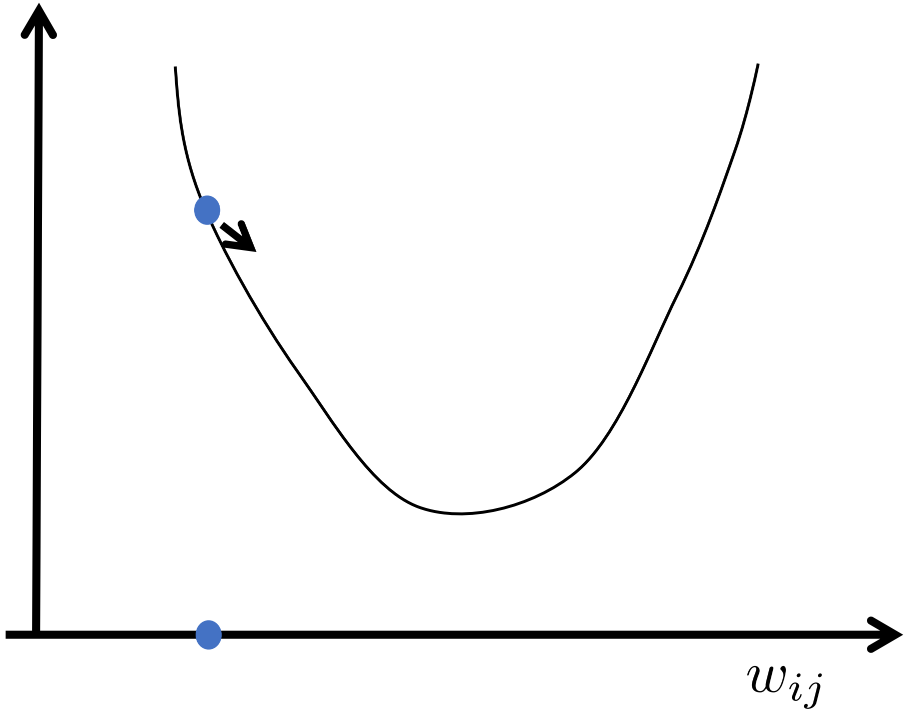
Nonconvex function (neural networks)



- multiple minima
- Different initializations can lead to different outcomes!
- may not even reach a local minimum

Step sizes

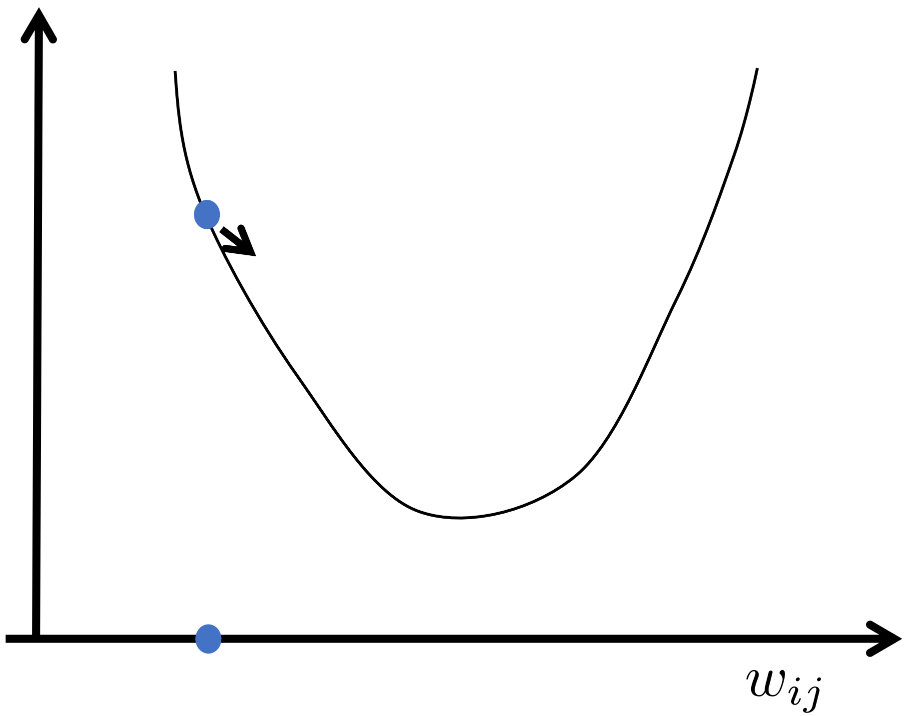
$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$



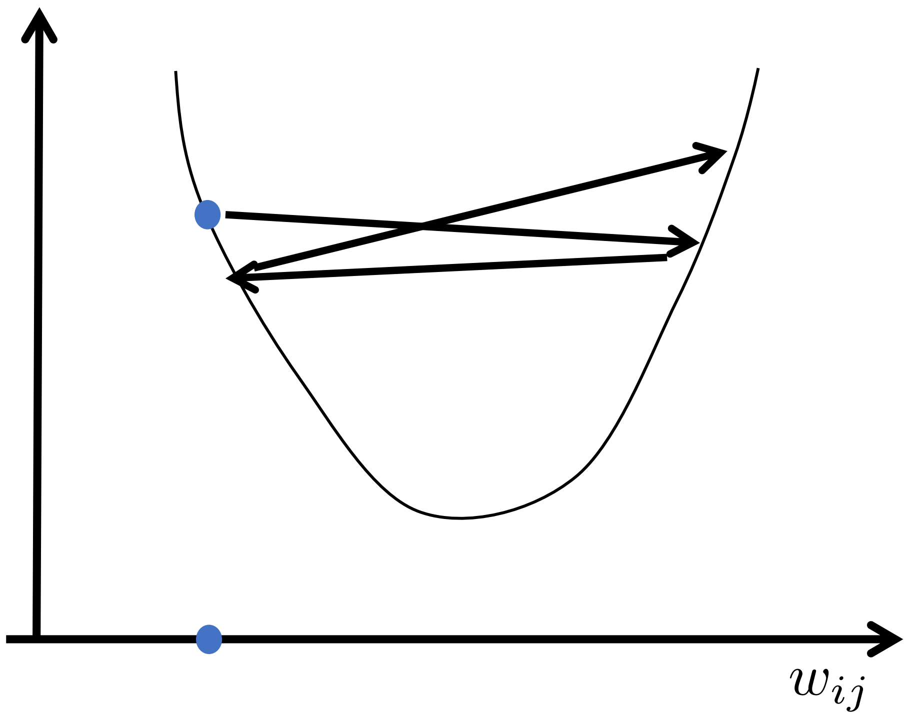
Too small: very slow progress

Step sizes

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$



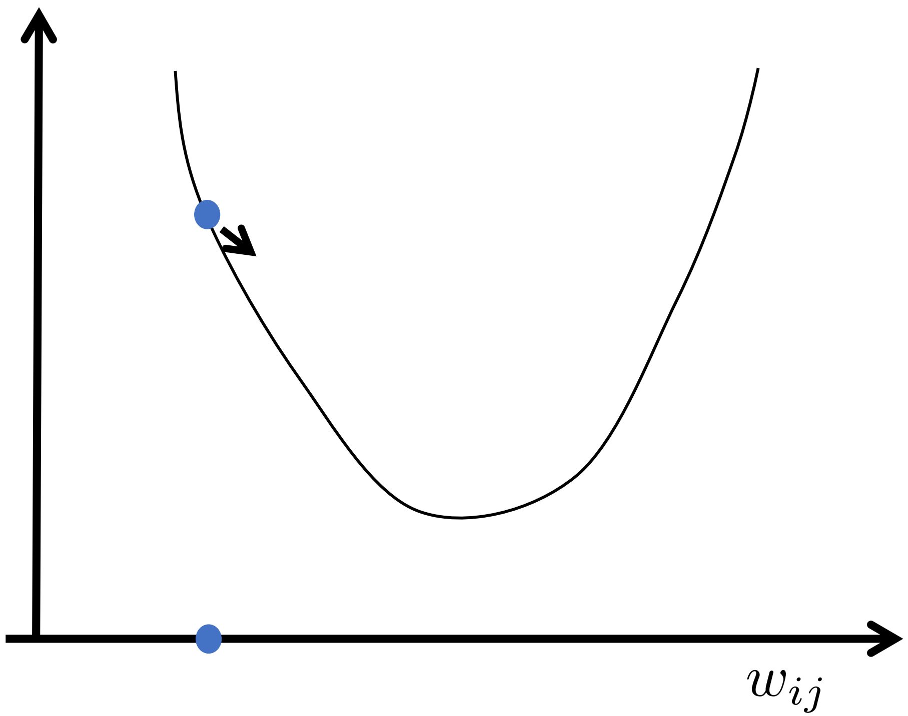
Too small: very slow progress



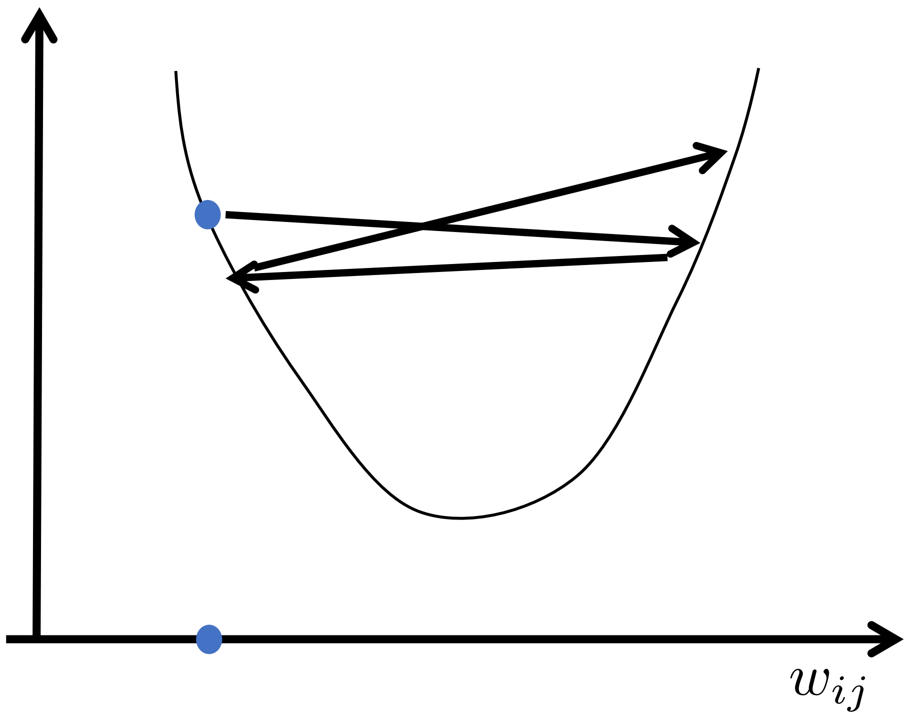
Too large: “overshoot”, loss can increase again, will not converge

Step sizes

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$



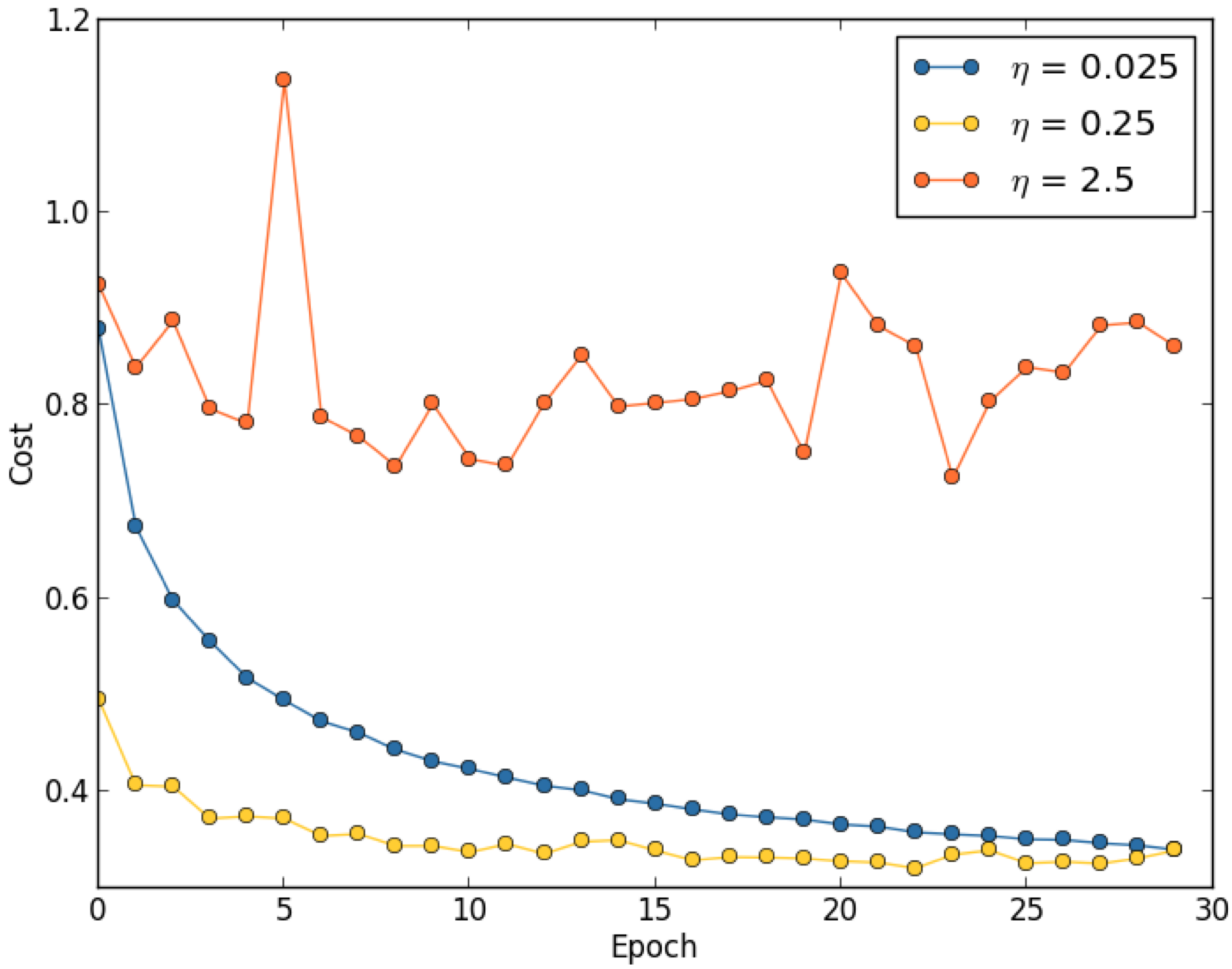
Too small: very slow progress

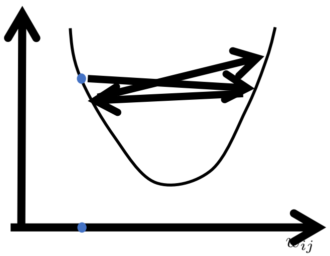


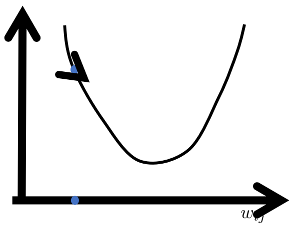
Too large: “overshoot”, loss can increase again, will not converge

- Typically small, decreasing. Tune…

Step sizes = learning rate

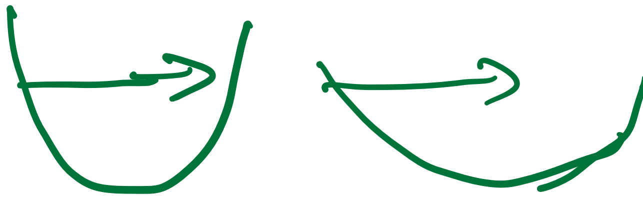






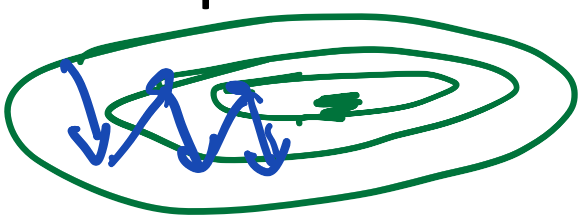
http://neuralnetworksanddeeplearning.com/chap3.html

Widely used improvements of SGD



- Adaptive Methods (e.g. AdaGrad)
  adaptively set step size for each network weight

- Momentum
  use weighted sum of current gradient with past gradients to be less erratic



- Adam
  combines Momentum & AdaGrad

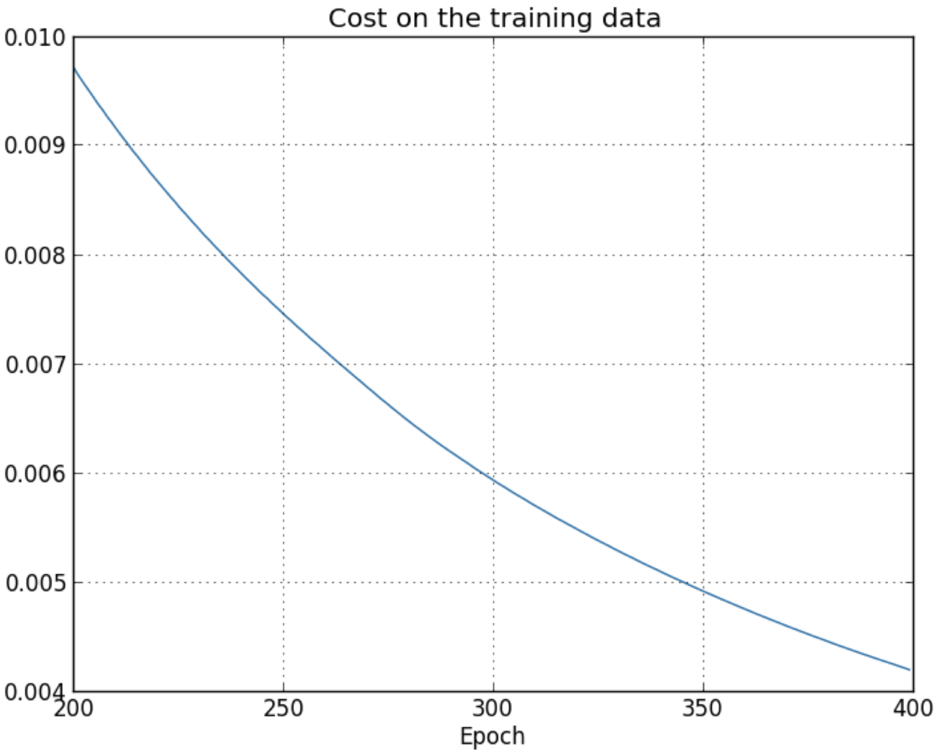
Summary: Neural Network training

- "Training": minimize a loss function over training data
- Iterative, small adjustments of network weights: stochastic gradient descent
- Gradients can be computed with backpropagation (a special case of auto-differentiation)
- Step sizes and initialization matter!

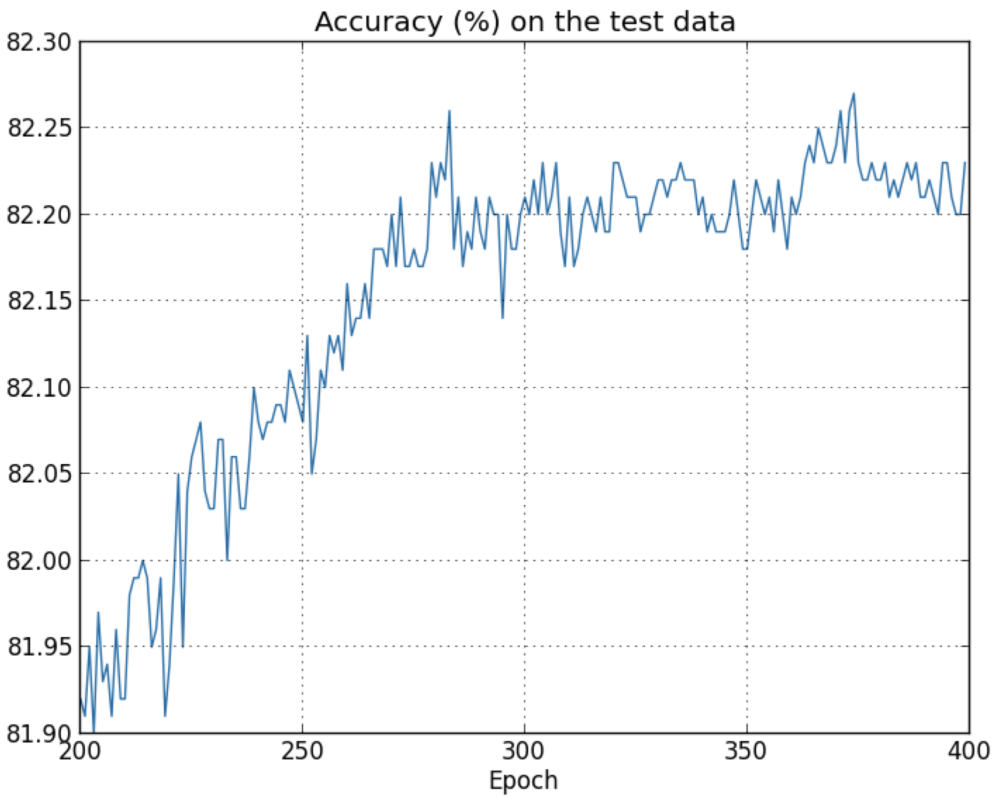
- *More practical hints for training: e.g.*
  Y. Bengio. Practical Recommendations for Gradient-Based Training of Deep Architectures.
  https://arxiv.org/pdf/1206.5533v2.pdf

Regularization

Loss on the training data



Accuracy on the test data



http://neuralnetworksanddeeplearning.com/chap3.html

Regularization

- **Squared norm / weight decay**
  add $+\frac{\lambda}{2}\|\theta\|^2$ to training loss

Regularization

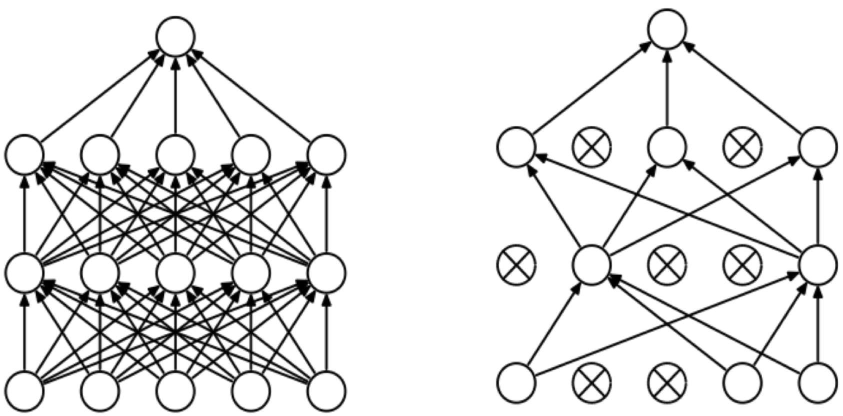
- **Squared norm / weight decay**
  add $+\frac{\lambda}{2}\|\theta\|^2$ to training loss
- **Early stopping**
  Check error on validation set after each epoch

Regularization

- **Squared norm / weight decay**
  add $+\frac{\lambda}{2}\|\theta\|^2$ to training loss
- **Early stopping**
  Check error on validation set after each epoch
- **Data augmentation**
  Perturbations (rotation, noise,…), averaging

Regularization

- **Squared norm / weight decay**
  add $+\frac{\lambda}{2}\|\theta\|^2$ to training loss
- **Early stopping**
  Check error on validation set after each epoch
- **Data augmentation**
  Perturbations (rotation, noise,…), averaging
- **Dropout**
  for each training data point, randomly switch off ½ of the units, don't update them either;
  at test time, scale weights by ½

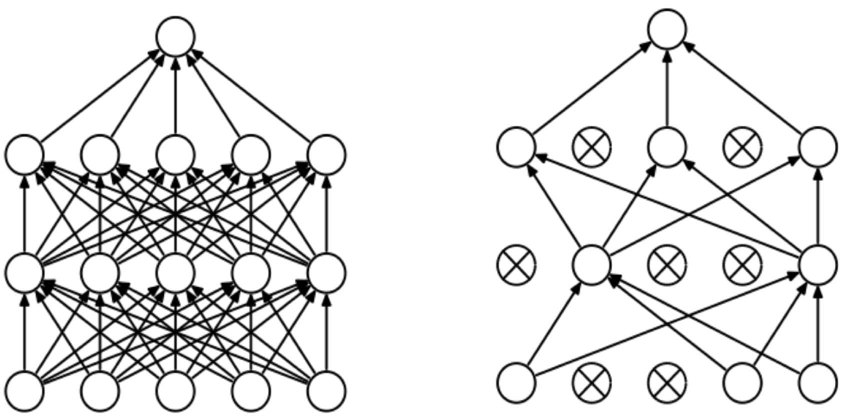


(a) Standard Neural Net (b) After applying dropout.

Figure: (Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. JMLR, 2014)

Regularization

- **Squared norm / weight decay**
  add $+\frac{\lambda}{2}\|\theta\|^2$ to training loss
- **Early stopping**
  Check error on validation set after each epoch
- **Data augmentation**
  Perturbations (rotation, noise,…), averaging
- **Dropout**
  for each training data point, randomly switch off ½ of the units, don't update them either;
  at test time, scale weights by ½
- **Batch normalization**
  normalize inputs of each layer



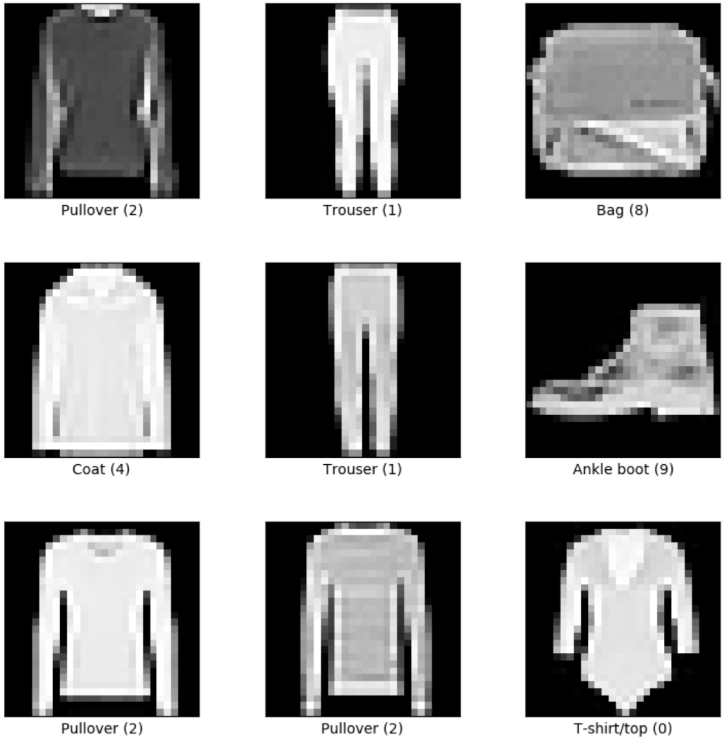
(a) Standard Neural Net

(b) After applying dropout.

Figure: (Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. JMLR, 2014)

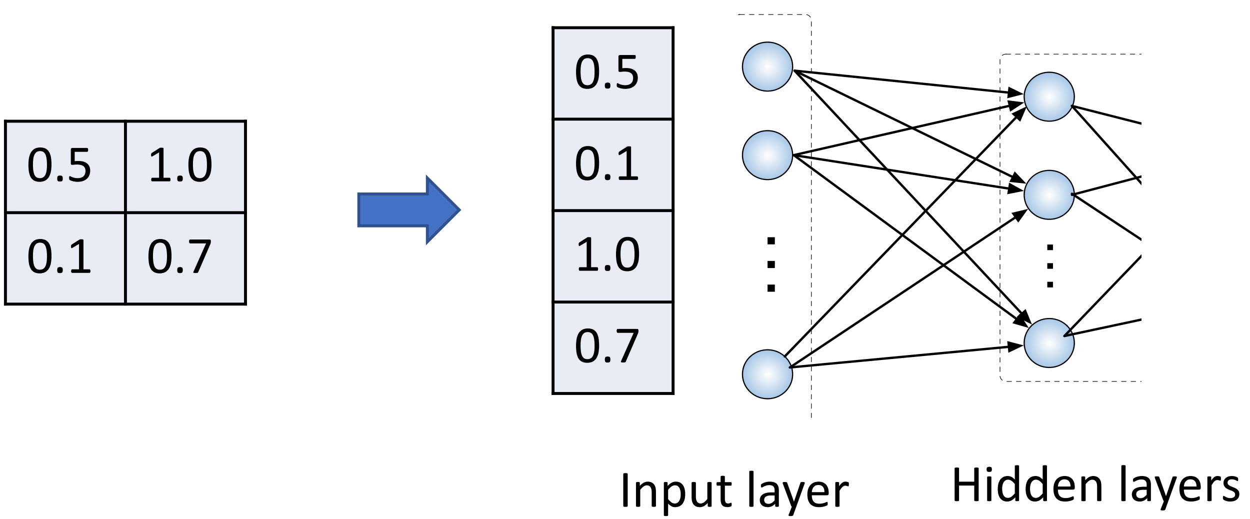
Example: Fashion MNIST

- 60,000 (train) + 10,000 (test) article images from Zalando
- 28x28 images
- 10 classes:
  T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle Boot



A neural network for Fashion-MNIST

- What are the inputs?
- Grayscale images = matrices

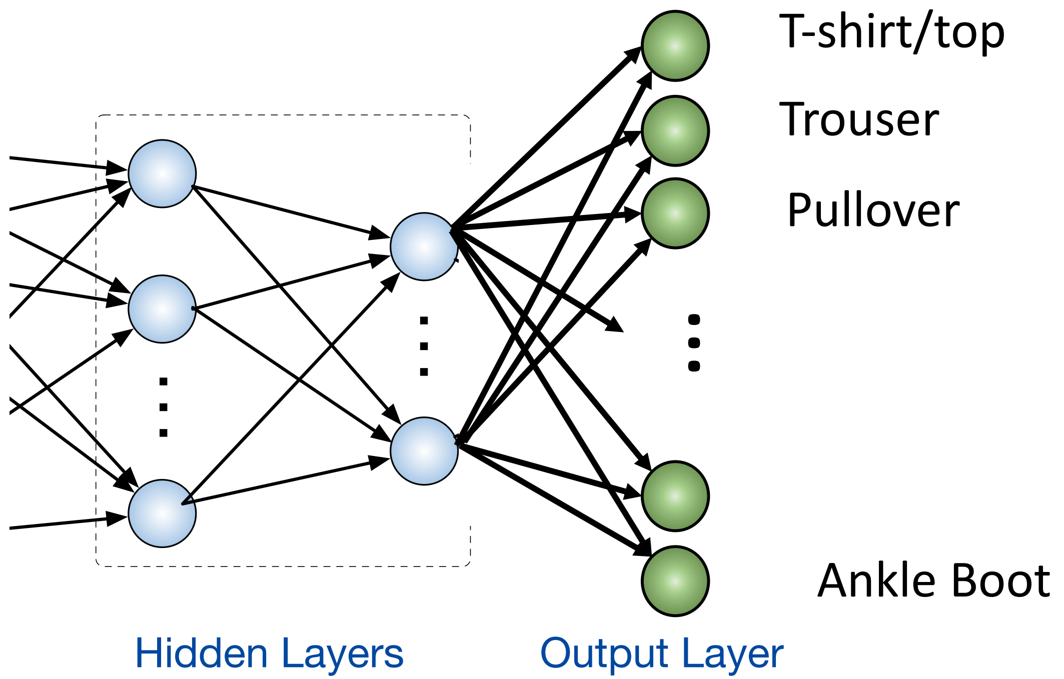


A neural network for Fashion-MNIST

- What are the outputs? Loss function?
- 10 classes

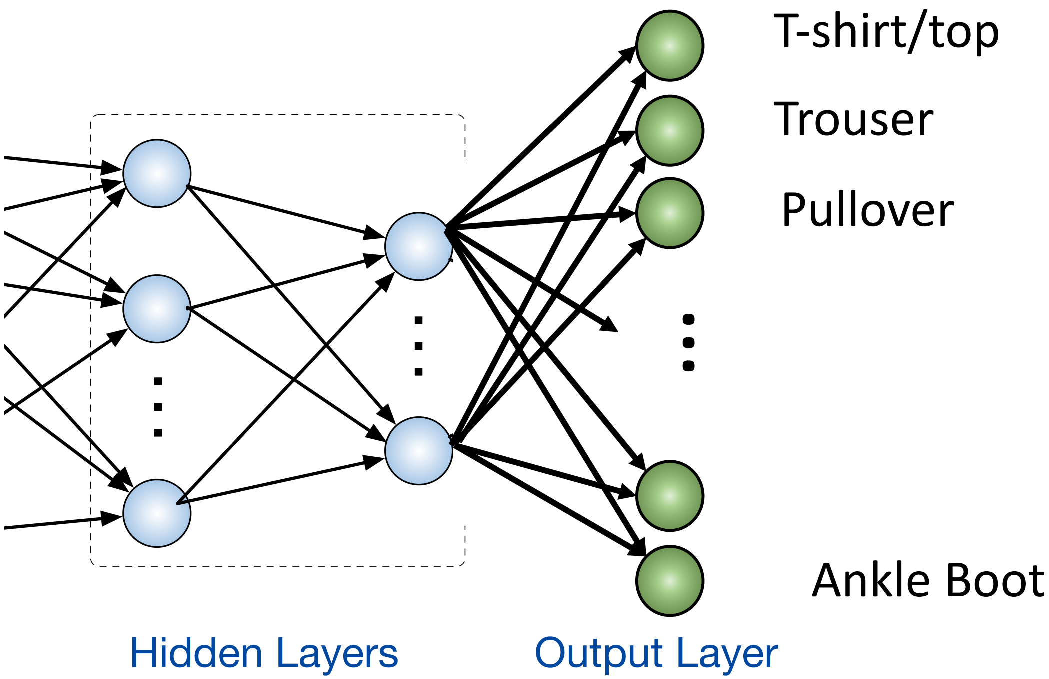
A neural network for Fashion-MNIST

- What are the outputs? Loss function?
- 10 classes



A neural network for Fashion-MNIST

- What are the outputs? Loss function?
- 10 classes



- Classification → Cross-entropy

Summary: Neural Networks

- Multiple simple units arranged in layers
- Complicated decision by breaking it up into simpler questions
- Modular structure: “lego pieces”
- Trainable by Stochastic gradient descent
- Compute gradients via Backpropagation
- Several hyperparameters to tune: step size, regularization/stopping, width, depth