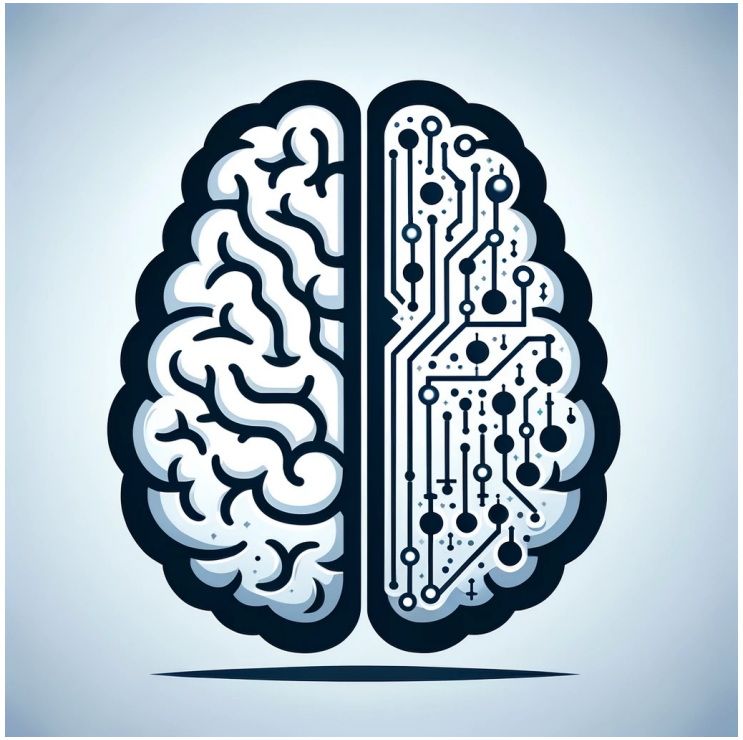
EN 601.473/601.673: Cognitive Artificial Intelligence (CogAI)



Lecture 13: Basics of neural networks, MAP, differentiable probabilistic programming

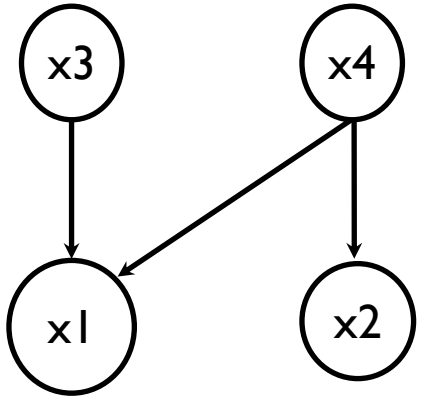
Tianmin Shu

Recap: Probabilistic language of thought

- Bayesian networks
- Probabilistic programs
- Sampling-based inference algorithms

Bayesian networks

- A Bayesian network (or Bayes net) is a way to specify a **joint distribution**: given the (in)dependence structure, then the conditional distributions



- Formal definition: A *Bayesian net*work:
  - A set of *n* variables V = {x_1, …, x_n}
  - A directed acyclic graph (DAG) on V.
  - A local conditional distribution for each variable in V given its parents in the graph (a factor in the joint dist.).

$$P(x_1,\ldots,x_n) = \prod_i P(x_i \mid Parents[x_i])$$

- Factorization simpler than the full joint distribution, which yields efficient learning and inference.

Bayesian modeling: $p(h, x)$

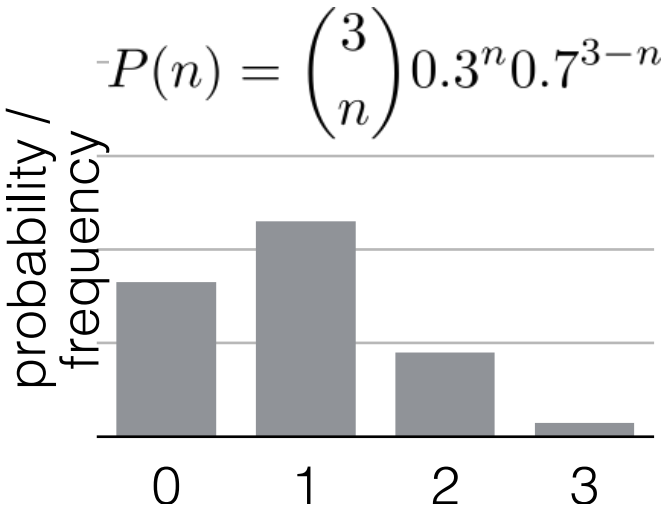
Probabilistic programs

Model **a joint distribution** as a probabilistic program

Random primitives:

```
var a = flip(0.3)
var b = flip(0.3)
var c = flip(0.3)
return a + b + c
```

=> 1 0 0
=> 0 0 0
=> 1 0 1
=> 2 0 1
...

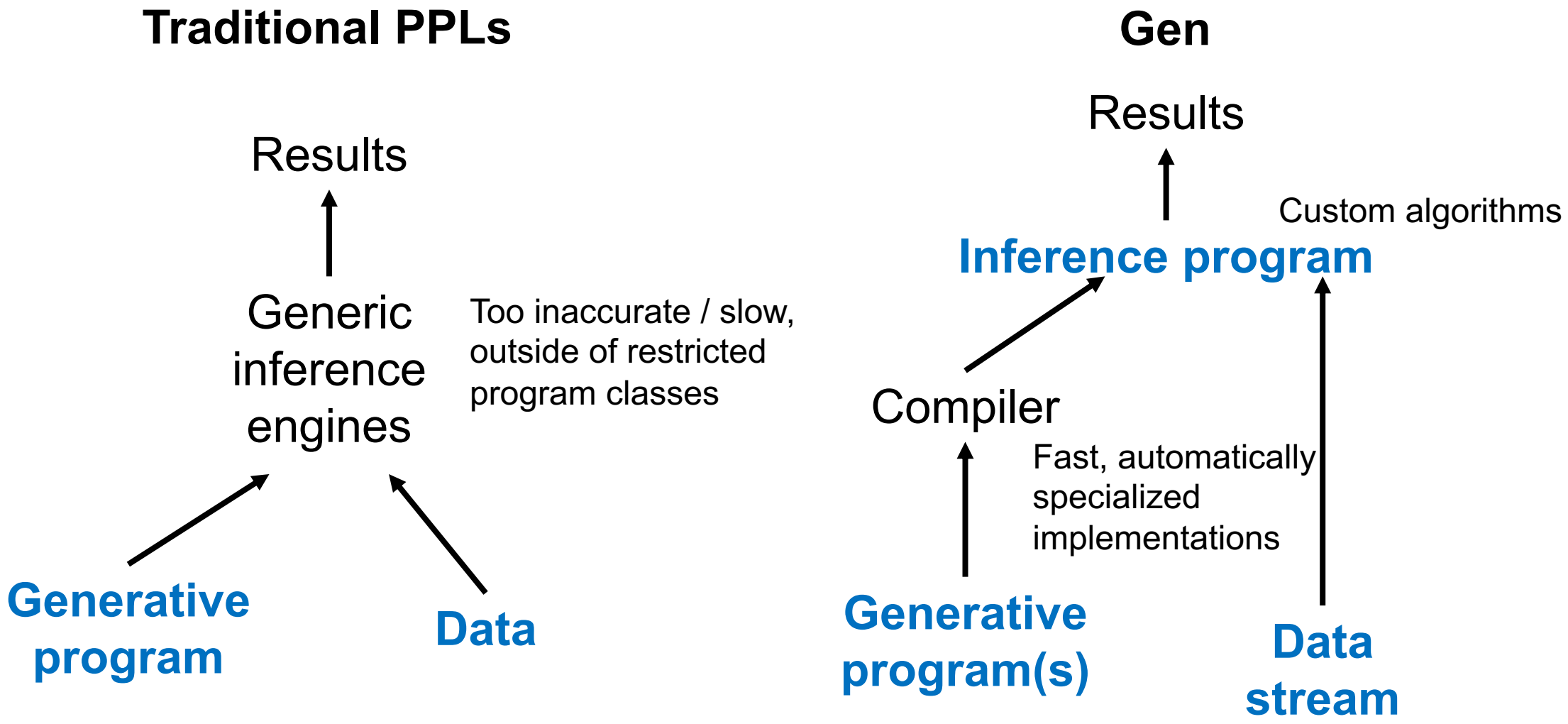


Sampling ≅ Distributions

Theorem: any computable distribution can be represented as the distribution induced by sampling from a probabilistic program

Probabilistic programming using Gen

(**BLUE**: specified by users)



Generative model

`@gen` macro for defining a generative model in Gen

```
using Gen: @gen

@gen function gen_f(p)
    n = {:initial_n} ~ uniform_discrete(1, 10)
    if ({:do_branch} ~ bernoulli(p))
        n *= 2
    end
    return {:result} ~ categorical([i == n ? 0.5 : 0.5/19 for i=1:20])
end;
```

Model **a joint distribution of all random variables (addresses on the trace)**

Bayesian modeling: $p(h, x)$

Sampling-based inference algorithms

- Rejection sampling
  - May waste a lot of samples
- Importance sampling
  - Only sample once and cannot update samples
- Markov chain Monte Carlo (MCMC)
  - Iteratively update samples
  - Better proposal distributions can speed up convergence
  - Proposals can be data–driven and can transition between hypotheses with different parameters (RJMCMC)
- Sequential Monte Carlo (SMC)
  - Inferring sequential latent variables
  - MCMC moves can rejuvenate particles

How to use Gen to implement sampling-based inference algorithms?

Looking ahead: Extending the Bayesian toolbox

- Neural networks
- Inference as optimization
- Neural amortized inference

Extending the Bayesian toolbox

- Neural networks
- Inference as optimization
- Neural amortized inference

Biological implementation

Computational level

- What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?

Algorithmic level

- How can this computational theory be implemented?
- In particular, what is the representation of the input and output, and what is the algorithm for the transformation?

Implementational level

- How can the representation and algorithm be realized physically?
- e.g., neural mechanisms that implement the algorithm.

[Adapted from Lake & Gureckis's slides]

Biological implementation

- Neurons

A biological neuron

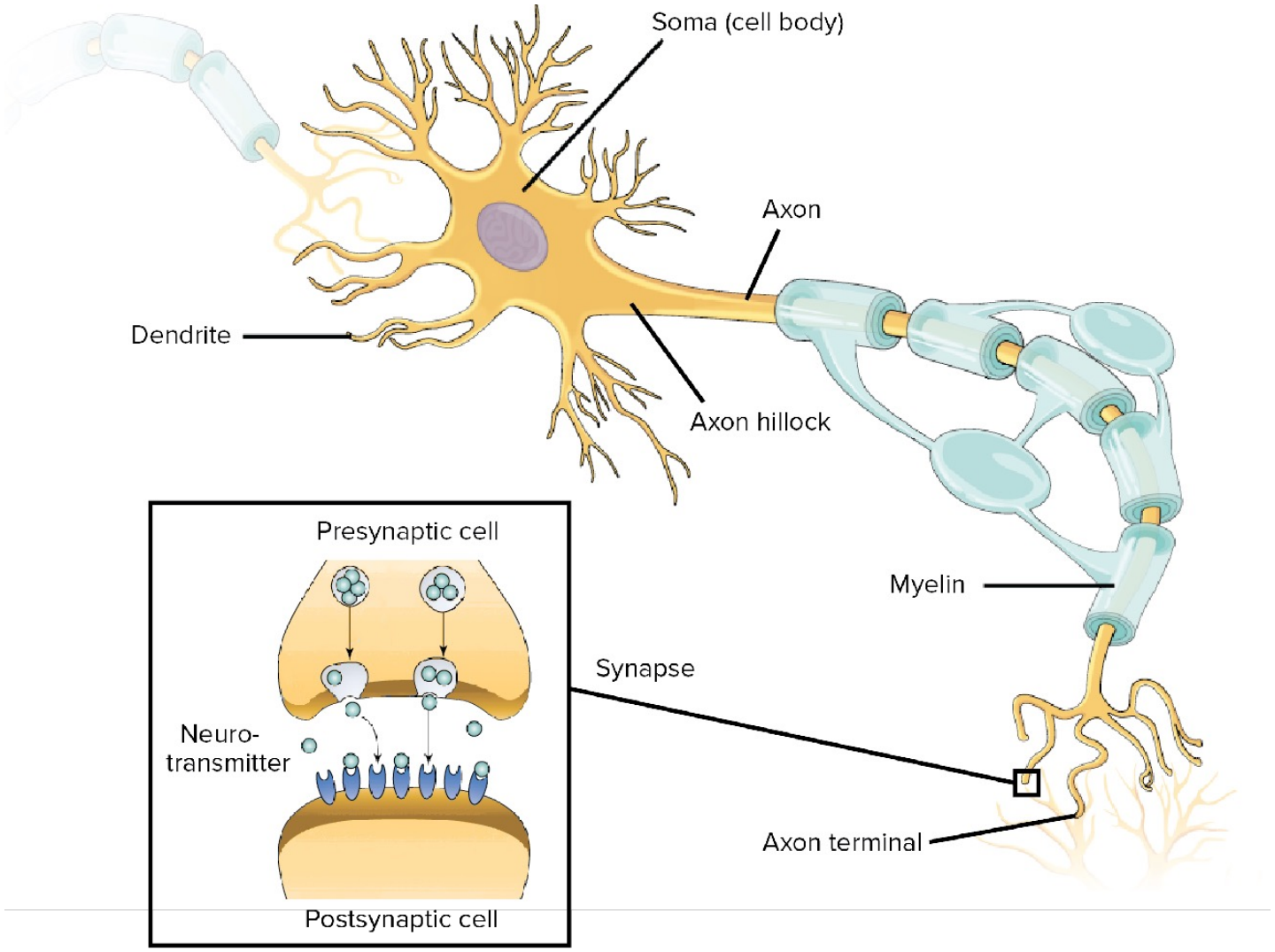
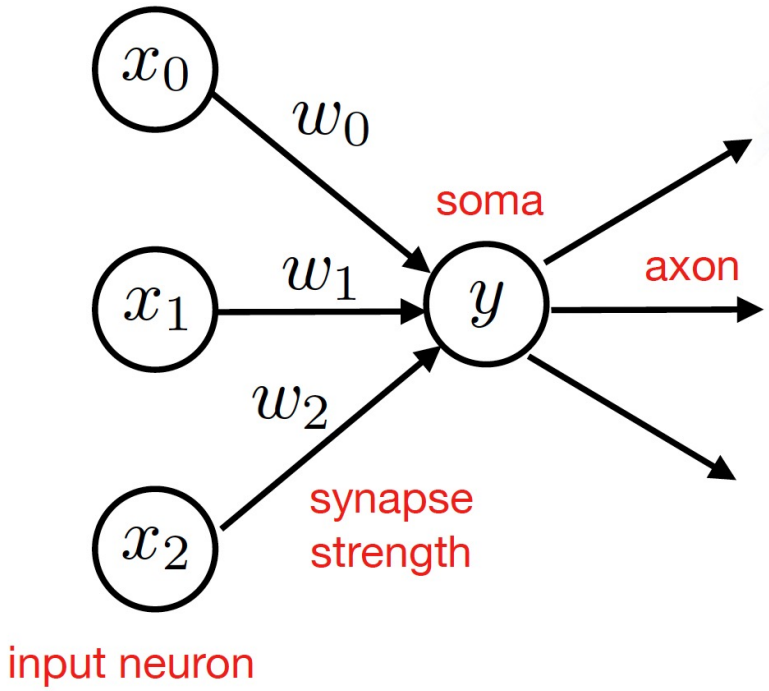
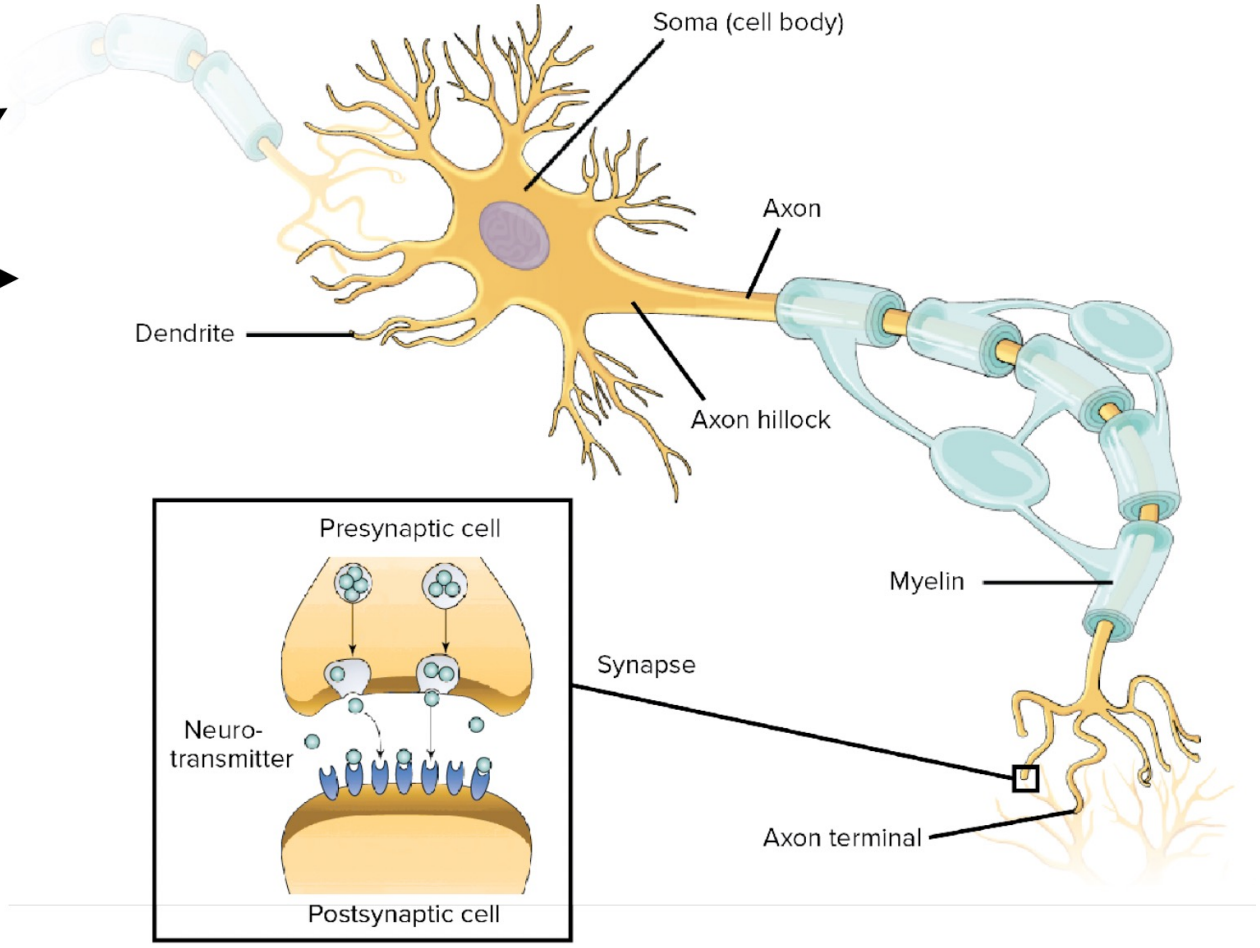


Image from OpenStax Biology

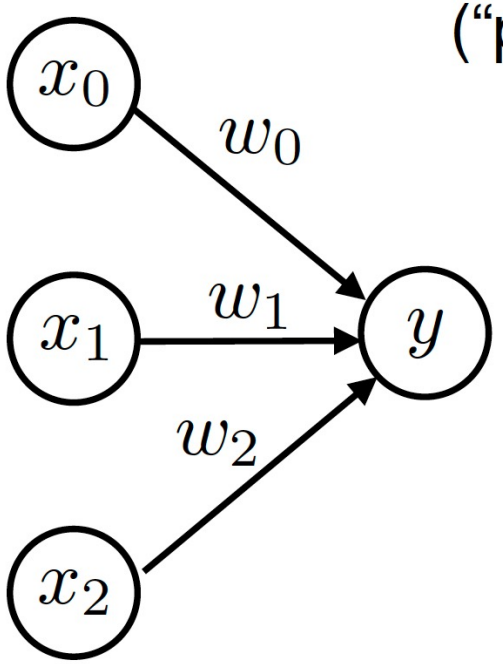
A simple artificial neuron





A simple artificial neuron (or “unit”)

(“perceptron”; Rosenblatt, 1958)



$$y = g(\sum_i x_i w_i + b)$$

x : input vector

w : weight vector

b : bias

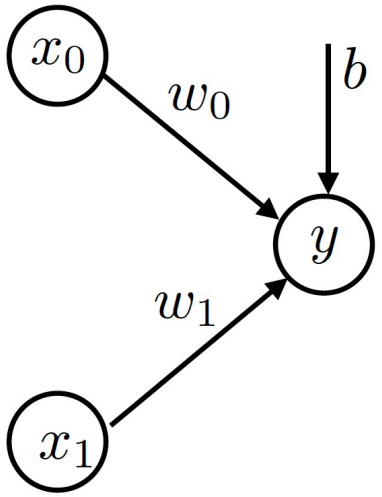
g : activation function

y : output

activation function:

$$g(\mathbf{net}) = \begin{cases} 1 & \mathbf{net} \geq 0 \\ 0 & \mathbf{net} < 0 \end{cases}$$

Computing the logical OR function

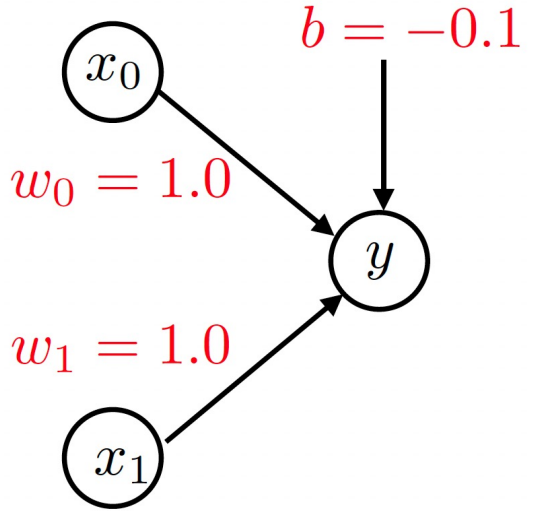


$$y = g(\sum_i x_i w_i + b)$$

$$g(\mathbf{net}) = \begin{cases} 1 & \mathbf{net} \geq 0 \\ 0 & \mathbf{net} < 0 \end{cases}$$

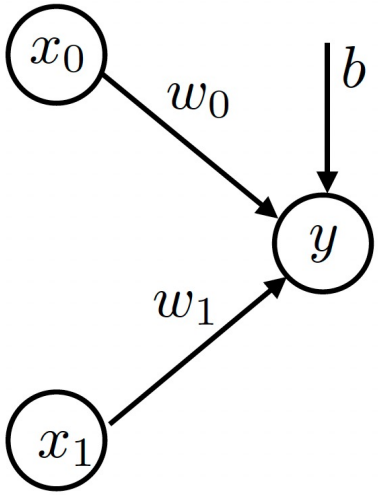
logical OR function

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1



x_0w_0	x_1w_1	b	net	y
0×1	0×1	-0.1	-0.1	0

Computing the logical AND function

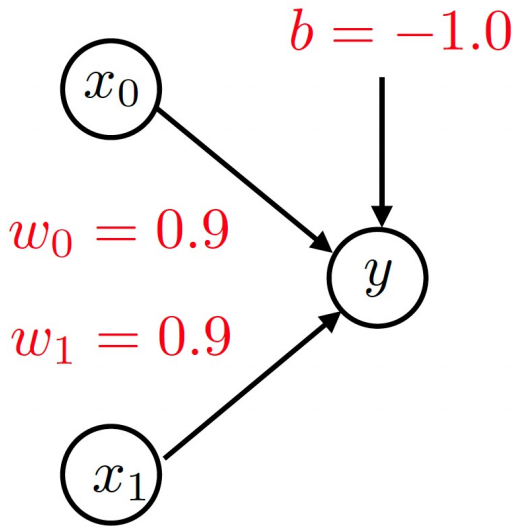


$$y = g(\sum_i x_i w_i + b)$$

$$g(\mathbf{net}) = \begin{cases} 1 & \mathbf{net} \geq 0 \\ 0 & \mathbf{net} < 0 \end{cases}$$

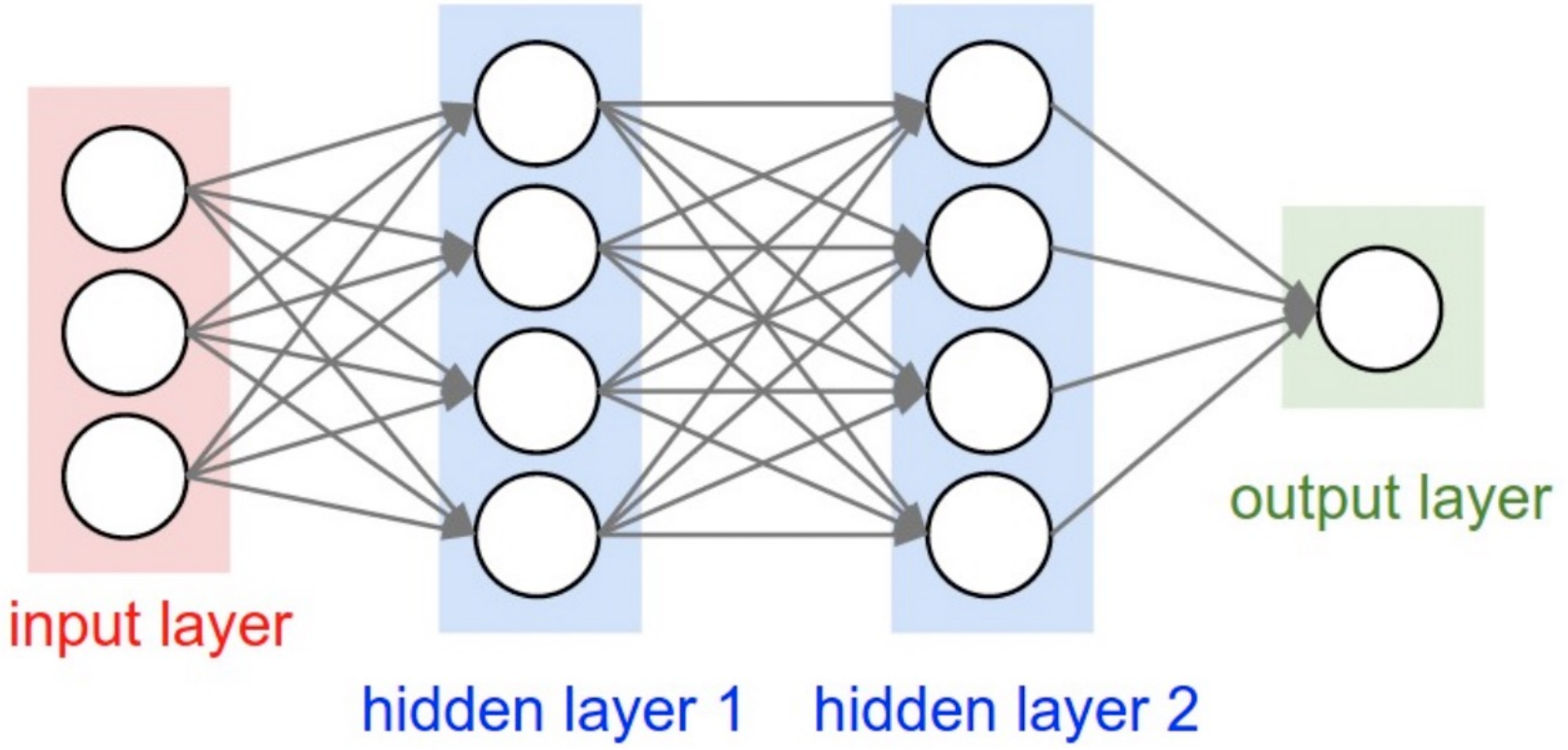
logical AND function

x_0	x_1	y
0	0	0
0	1	0
1	0	0
1	1	1

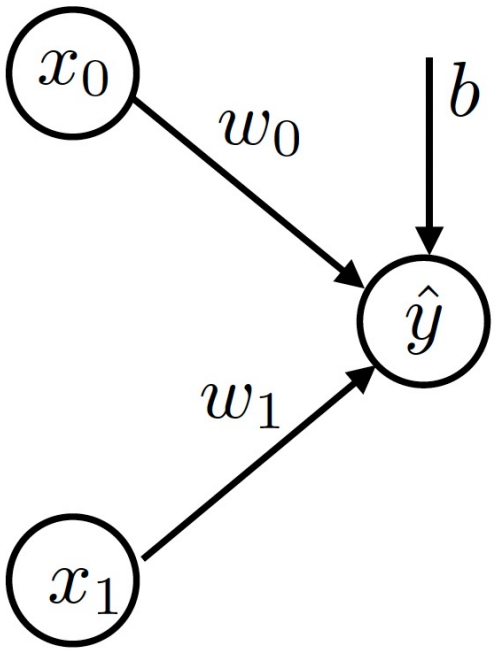


x_0w_0	x_1w_1	b	net	y
0×0.9	0×0.9	-1.0	-1.0	0

Artificial neural networks



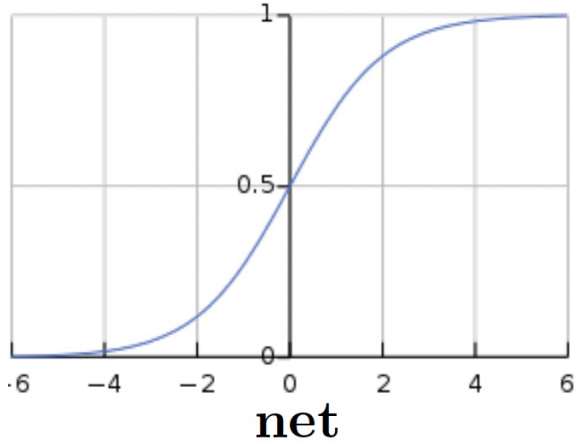
Learning by optimizing an objective (loss) function



$$\hat{y} = g(\sum_i x_i w_i + b)$$

activation function:
"sigmoid" or "logistic function"

$$g(\mathbf{net}) = \frac{1}{1 + e^{-\mathbf{net}}}$$



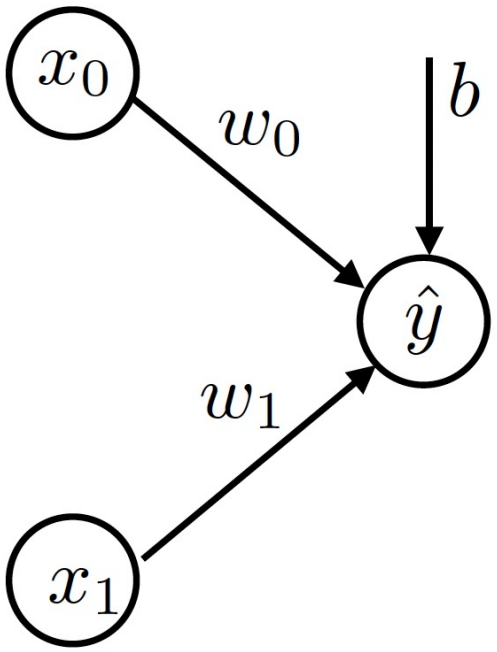
x : input vector
w : weight vector
b : bias
g : activation function
$\hat{y}$: predicted output
y : target output

Error/loss function:
"squared error"

$$E(w, b) = (\hat{y} - y)^2$$

We want to minimize the squared difference between the predicted output and the target output (also could use "sum squared error", or "mean squared error", across multiple different predictions)

Learning by optimizing an objective (loss) function

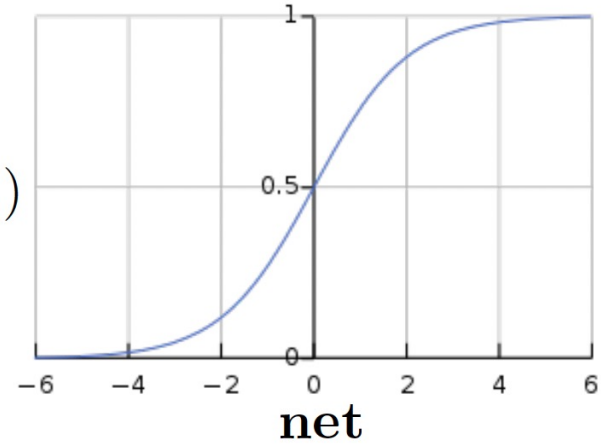


$$\hat{y} = g(\sum_i x_i w_i + b)$$

activation function:

"sigmoid" or "logistic function"

$$g(\mathbf{net}) = \frac{1}{1 + e^{-\mathbf{net}}}$$



x : input vector

w : weight vector

b : bias

g : activation function

$\hat{y}$: predicted output

y : target output

Error/loss functions:

"squared error"

$$E(w, b) = (\hat{y} - y)^2$$

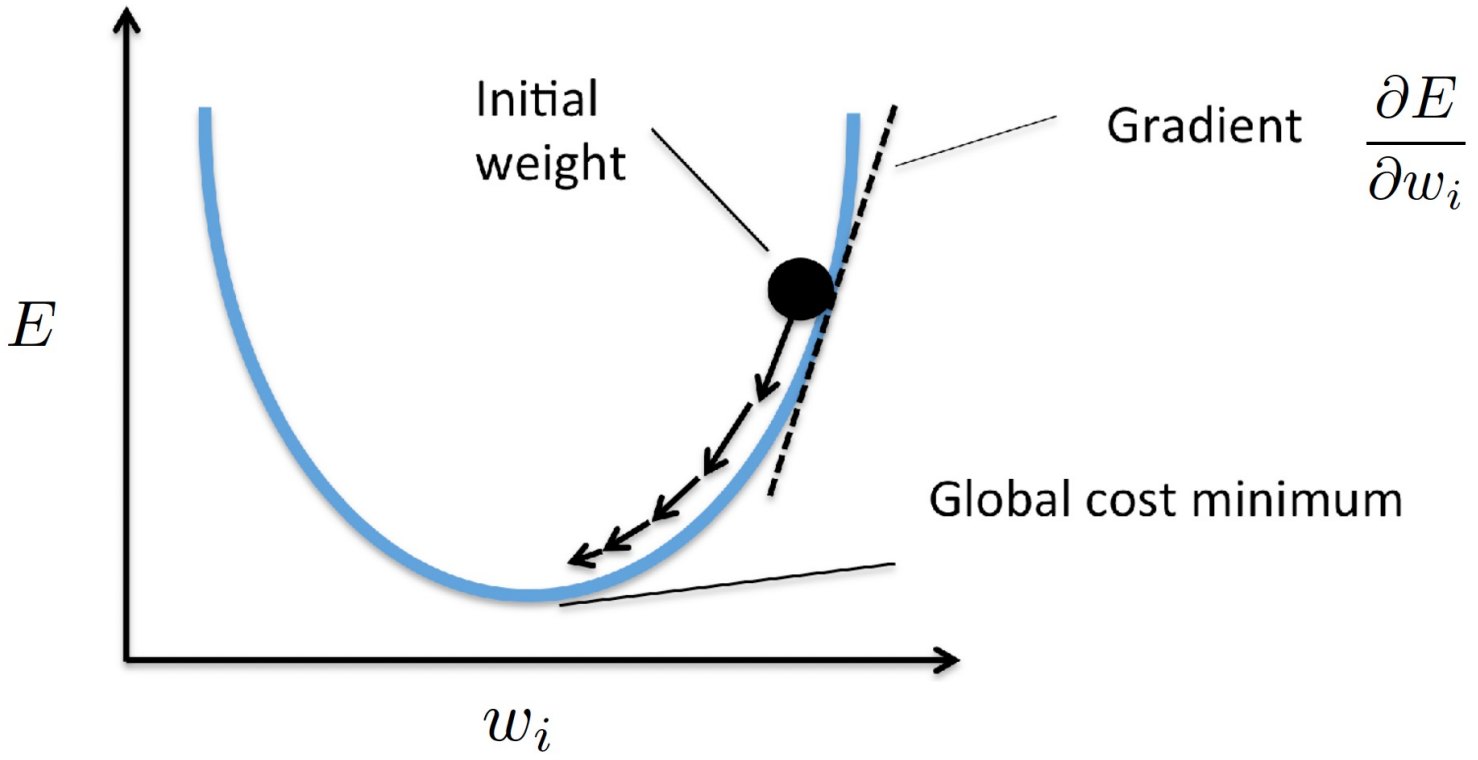
"negative log-likelihood"

$$E(w, b) = -\log[\hat{y}^y (1 - \hat{y})^{1-y}]$$

Some differences:

- Logistic regression only accepts 0 or 1 discrete output, whereas this formulation can accept 0-1 continuous output
- Logistic regression uses a negative log-likelihood loss, so it is fitting a maximum likelihood estimate

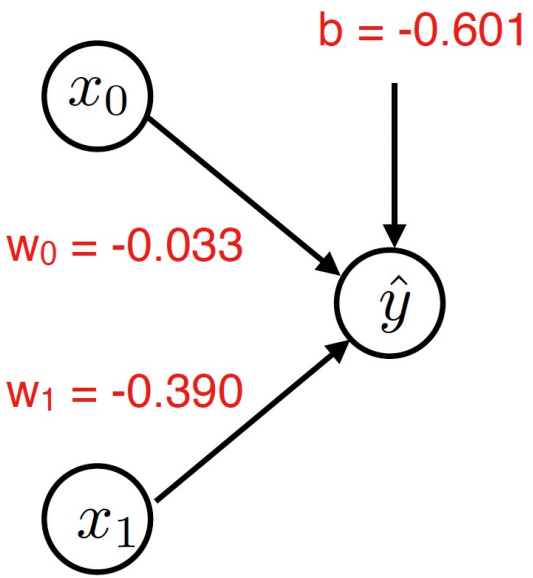
Stochastic gradient descent



Computing the gradient tells us which direction to go for steepest descent:

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i} \qquad \alpha : \text{learning rate}$$

Learning via stochastic gradient descent



x : input vector
w : weight vector
b : bias
g : activation function
$\hat{y}$: predicted output
y : target output

$$\hat{y} = g(\sum_i x_i w_i + b) \quad g(\mathbf{net}) = \frac{1}{1 + e^{-\mathbf{net}}}$$

$$E(w, b) = (\hat{y} - y)^2$$

logical OR

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1

Computing the error:

x = [1, 1]
$\hat{y}$ = 0.26 (predicted output)
E(w,b) = 0.54 (error)

Computing the gradient:

$$\frac{\partial E(w,b)}{\partial w_i} = 2(\hat{y} - y)g(\mathbf{net})(1 - g(\mathbf{net}))x_i$$
$$= 2(0.26 - 1)(.26)(1 - .26)1$$
$$= -0.285$$

(if we increase weight, error goes down)

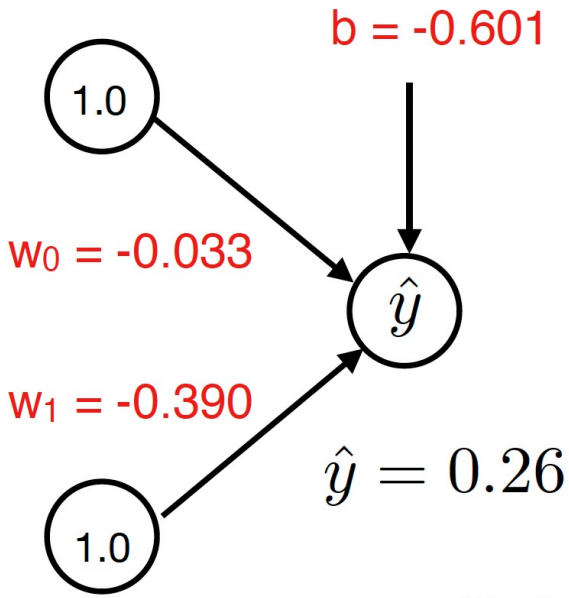
Update with gradient descent

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

α : learning rate

Visualizing an update to the weights

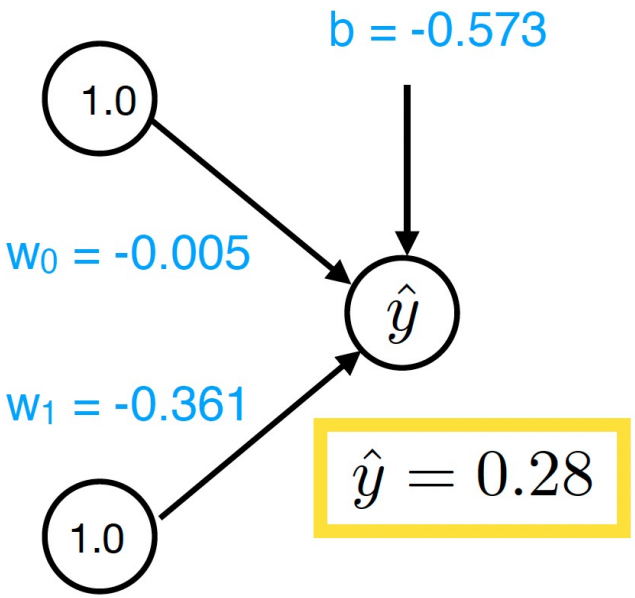
Before update



$\hat{y} = 0.26$

Update

After update
(all parameters increased)



$\hat{y} = 0.28$

Update rules

$$w_0 \leftarrow w_0 - \alpha(-0.285)$$
$$w_1 \leftarrow w_1 - \alpha(-0.285)$$

(similar update for bias)

Learning rate

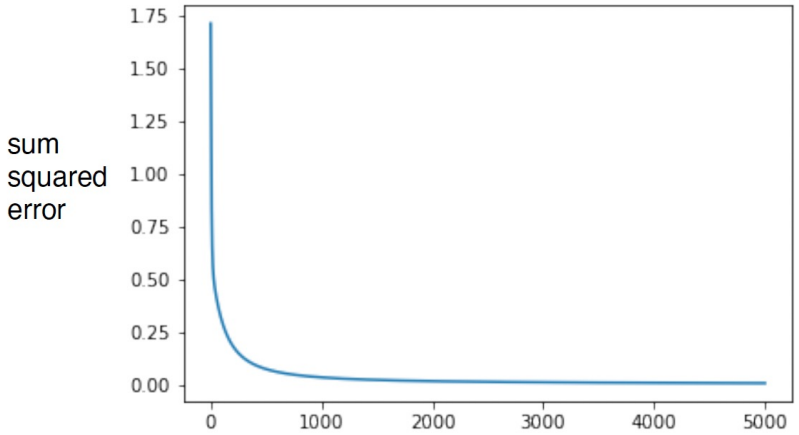
$$\alpha = 0.1$$

Notice that the new prediction is a little bit better! (closer to the target *y = 1.0*)

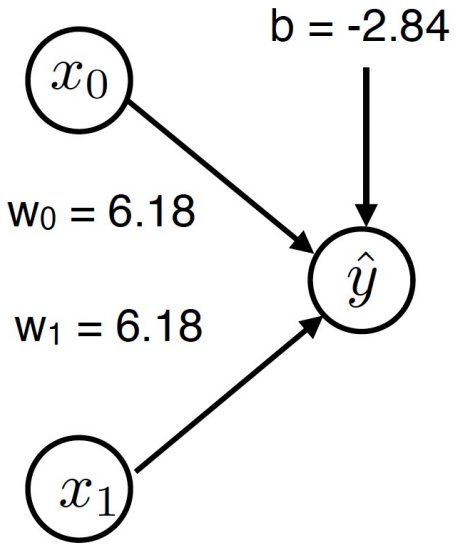
Learning via stochastic gradient descent

We repeatedly cycle through each pattern in the training set, making updates after each pattern as in the previous slide.

Error over time



Final network

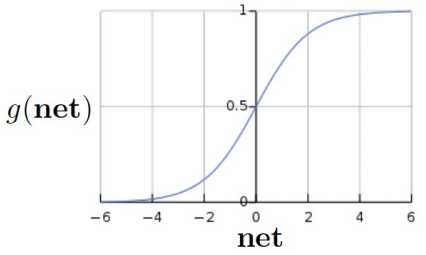


Learned function (logical OR)

x_0	x_1	$\hat{y}$	y
0	0	0.05	0
0	1	0.97	1
1	0	0.97	1
1	1	0.99	1

Pretty good fit!

$$g(\mathbf{net}) = \frac{1}{1 + e^{-\mathbf{net}}}$$



Chain rule for computing the gradient

Definitions

$$E(w,b) = (\hat{y} - y)^2$$
$$= (g(\text{net}) - y)^2$$

$$\hat{y} = g(\mathbf{net}) = g(\sum_i x_i w_i + b)$$

For logistic function, we have this useful property:

$$\frac{\partial g(\mathbf{net})}{\partial \mathbf{net}} = g(\mathbf{net})(1 - g(\mathbf{net}))$$

Chain rule

$$\frac{\partial E(u)}{\partial w} = \frac{\partial E(u)}{\partial u}\frac{\partial u}{\partial w}$$

where $u = f(w)$

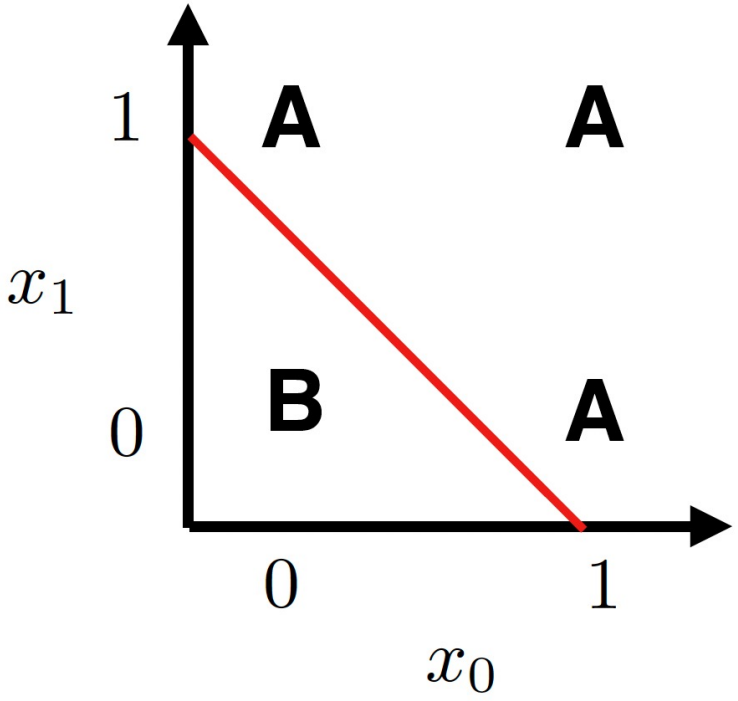
Computing the gradient of the error with respect to the weight:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial g(\mathbf{net})}\frac{\partial g(\mathbf{net})}{\partial \mathbf{net}}\frac{\partial \mathbf{net}}{\partial w_i}$$
$$= 2(\hat{y} - y)\frac{\partial g(\mathbf{net})}{\partial \mathbf{net}}\frac{\partial \mathbf{net}}{\partial w_i}$$
$$= 2(\hat{y} - y)g(\mathbf{net})(1 - g(\mathbf{net}))\frac{\partial \mathbf{net}}{\partial w_i}$$
$$= 2(\hat{y} - y)g(\mathbf{net})(1 - g(\mathbf{net}))x_i$$

Limits of linear classifiers

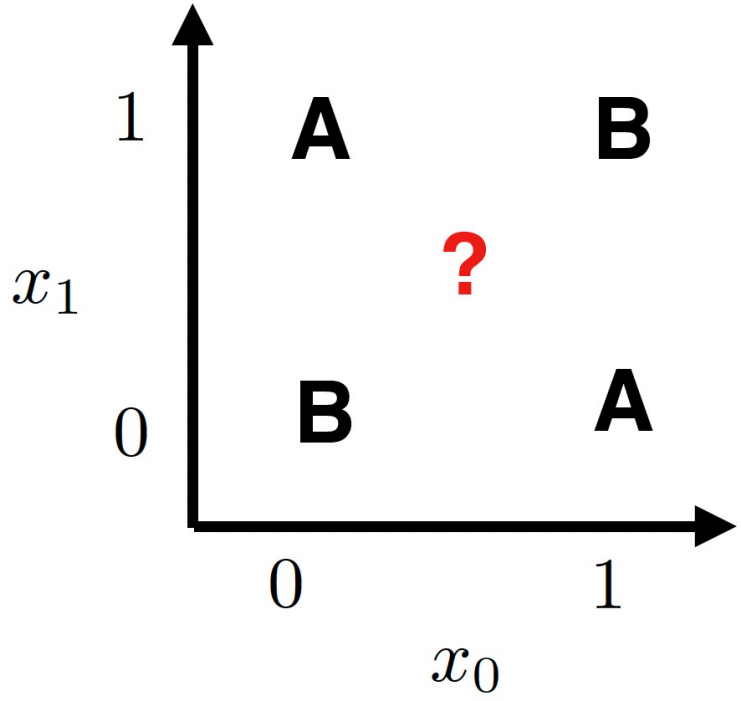
Linearly separable

Class A vs. B (logical OR)



Non-linearly separable

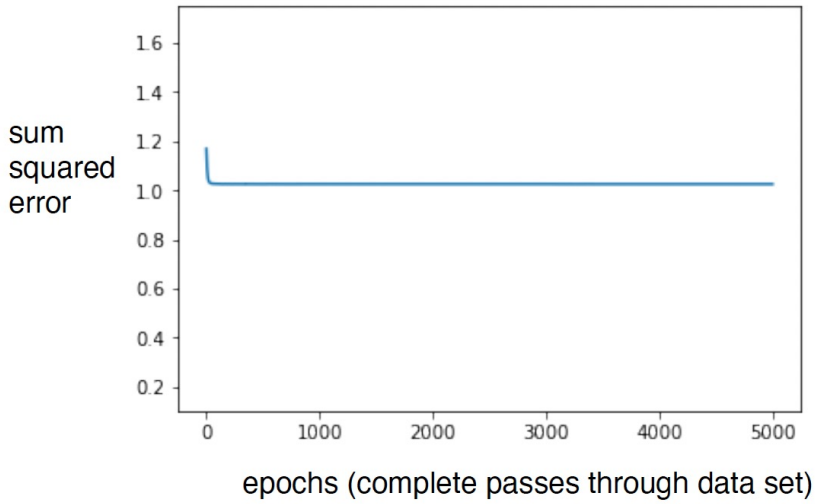
Class A vs. B (logical XOR)



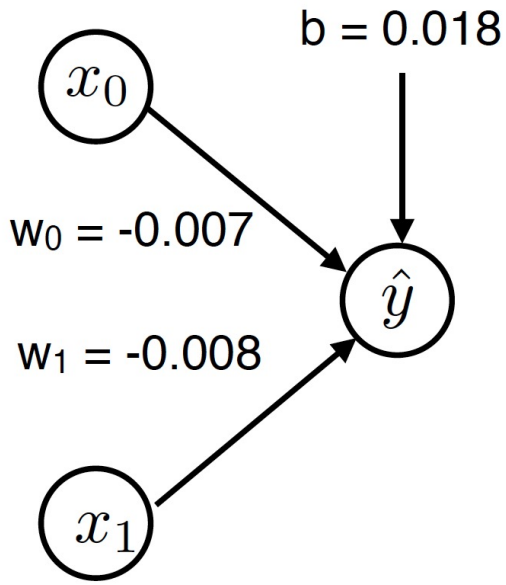
Failing to learn XOR with a linear classifier

("one or the other, but not both")

Error over time



Final network



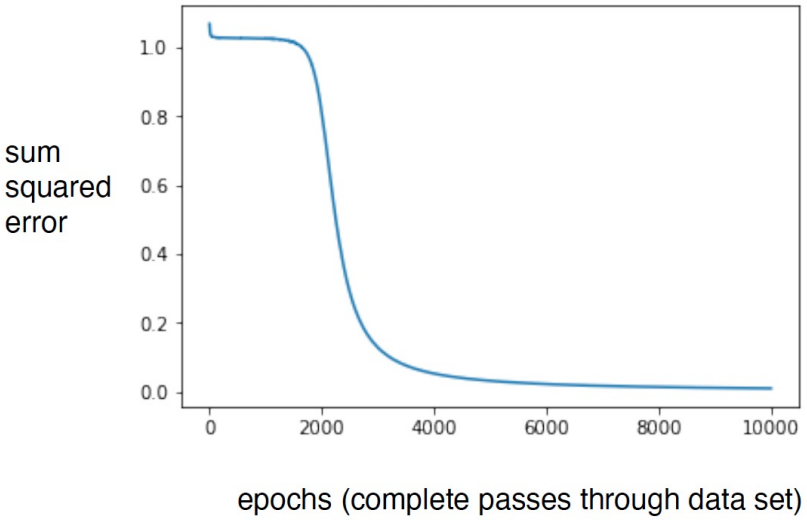
Learned function (logical XOR)

x_0	x_1	$\hat{y}$	y
0	0	0.50	0
0	1	0.50	1
1	0	0.50	1
1	1	0.50	0

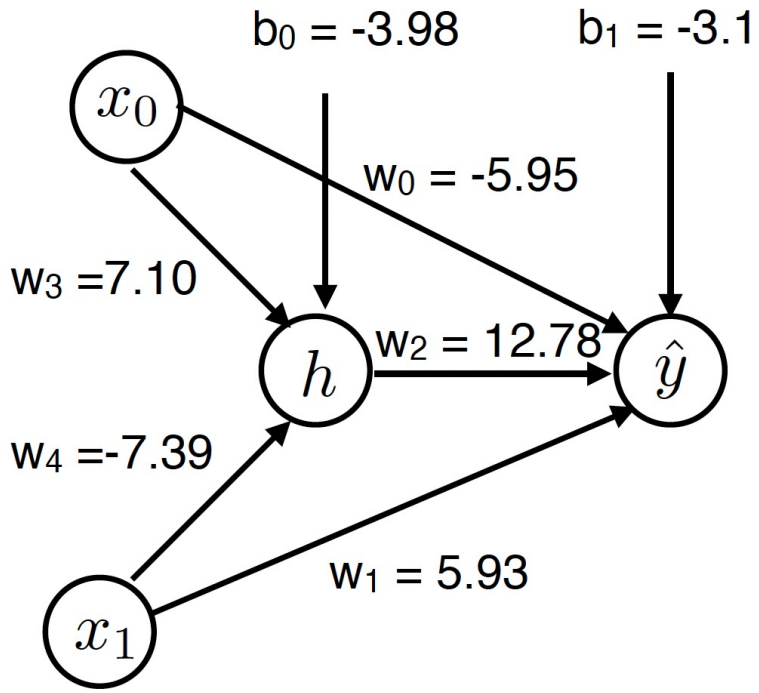
Terrible fit!

Learning XOR with a multi-layer classifier

Error over time



Final network



Learned function (logical XOR)

x_0	x_1	$\hat{y}$	y
0	0	0.05	0
0	1	0.94	1
1	0	0.96	1
1	1	0.05	0

Pretty good fit!

Backpropagation algorithm for computing gradient

Updates for these weights the same as before:

$$\frac{\partial E}{\partial w_0} \quad \frac{\partial E}{\partial w_1} \quad \frac{\partial E}{\partial w_2}$$

What about the other weights?

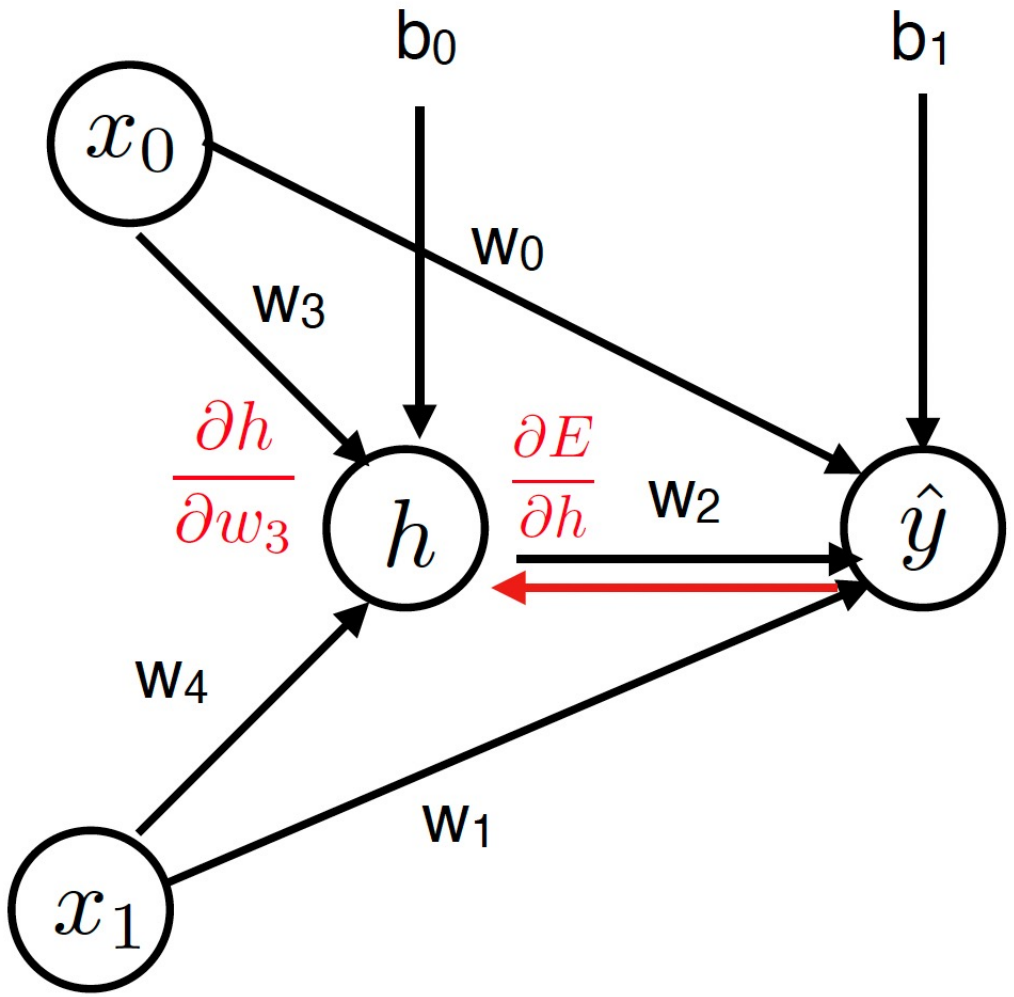
$$\frac{\partial E}{\partial w_3} \quad \frac{\partial E}{\partial w_4}$$

Multi-step strategy:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial h}\frac{\partial h}{\partial w_3}$$

Step 1) Compute how error changes as a function of hidden unit activation

Step 2) Compute how hidden unit activation changes as a function of weight



Backpropagation algorithm for computing gradient

Multi-step strategy:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial h}\frac{\partial h}{\partial w_3} = \frac{\partial E}{\partial g(\mathbf{net}_h)}\frac{\partial g(\mathbf{net}_h)}{\partial w_3}$$

Step 1) Compute how error changes as a function of hidden unit activation *(we worked most of this step out already for single layer net)*

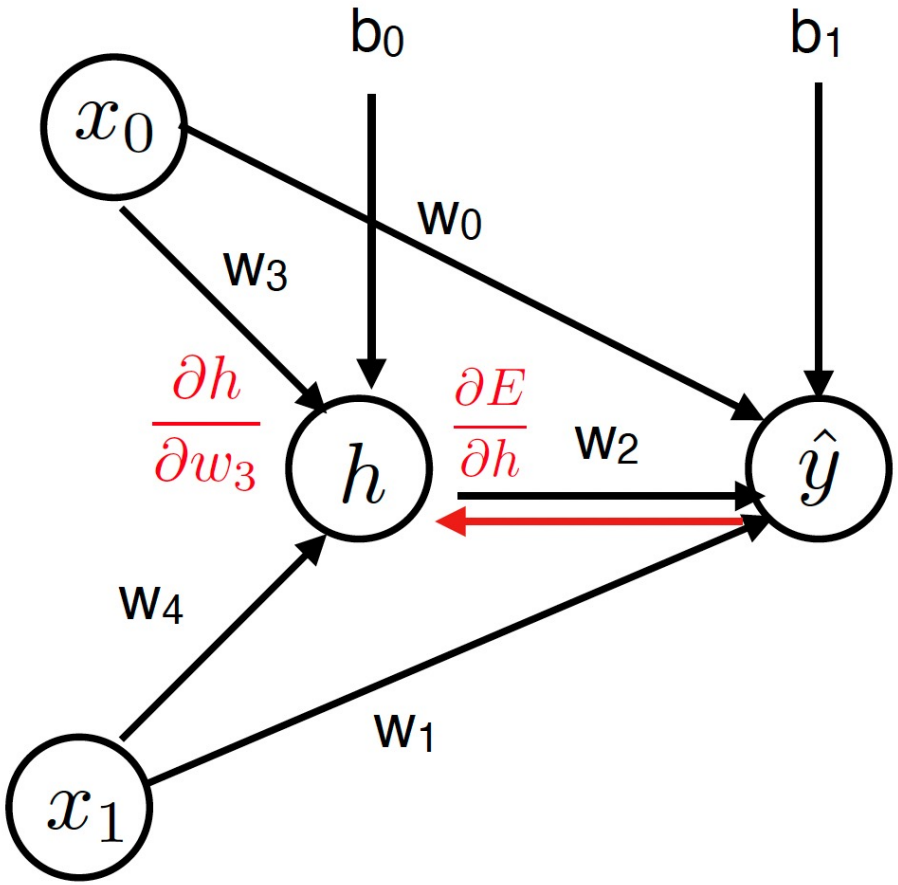
$$\frac{\partial E}{\partial g(\mathbf{net}_h)} = \frac{\partial E}{\partial g(\mathbf{net}_y)}\frac{\partial g(\mathbf{net}_y)}{\partial \mathbf{net}_y}\frac{\partial \mathbf{net}_y}{\partial g(\mathbf{net}_h)}$$

$$= 2(\hat{y} - y)g(\mathbf{net}_y)(1 - g(\mathbf{net}_y))w_2$$

Step 2) Compute how hidden unit activation changes as a function of weight

$$\frac{\partial g(\mathbf{net}_h)}{\partial w_3} = \frac{\partial g(\mathbf{net}_h)}{\partial \mathbf{net}_h}\frac{\partial \mathbf{net}_h}{\partial w_3}$$

$$= g(\mathbf{net}_h)(1 - g(\mathbf{net}_h))x_0$$

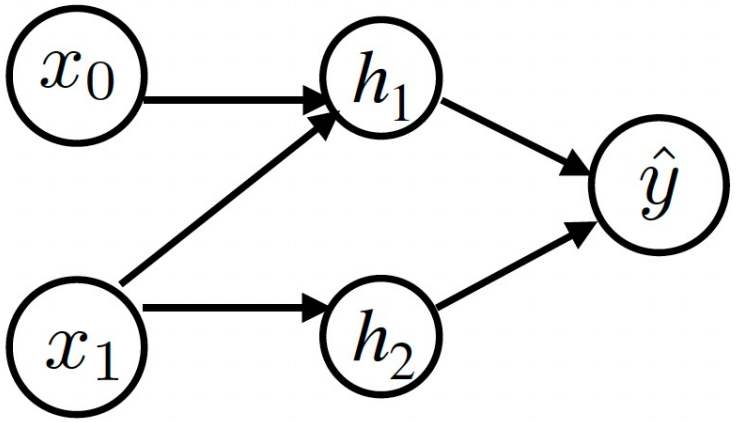


As before, update with gradient descent:

$$w_i \leftarrow w_i - \alpha\frac{\partial E}{\partial w_i} \quad \alpha : \text{learning rate}$$

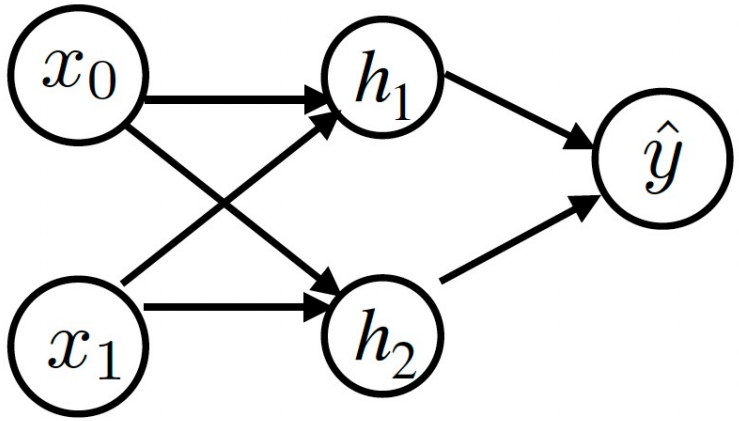
Chain rule for multivariable functions

1 variable case



$$\frac{\partial E}{\partial x_0} = \frac{\partial E}{\partial h_1}\frac{\partial h_1}{\partial x_0}$$

multivariable case



$$\frac{\partial E}{\partial x_0} = \sum_i \frac{\partial E}{\partial h_i}\frac{\partial h_i}{\partial x_0}$$

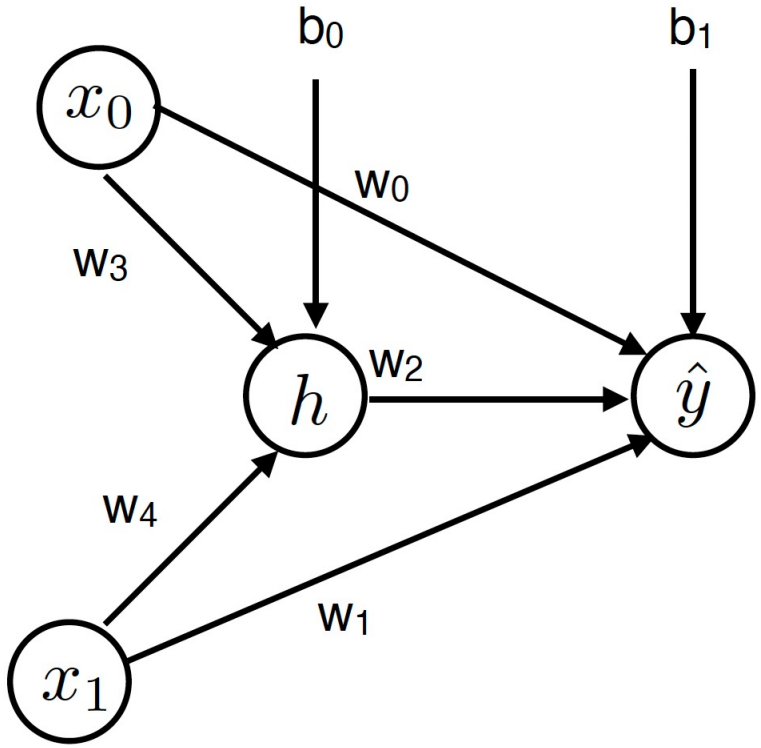
Modern DL frameworks compute gradients for you!

- Autodiff: automatic differentiation

PyTorch code for learning the XOR model

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.i2h = nn.Linear(2, 1)
        self.all2o = nn.Linear(3, 1)

    def forward(self, x):
        netinput_h = self.i2h(x)
        h = F.sigmoid(netinput_h)
        x2 = torch.cat((x,h))
        netinput_y = self.all2o(x2)
        out = F.sigmoid(netinput_y)
        return out

def update(pat, target, net):
        net.train()
        optimizer.zero_grad()
        output = net(pat)
        loss = F.mse_loss(output, target, size_average=False)
        loss.backward()
        optimizer.step()
```

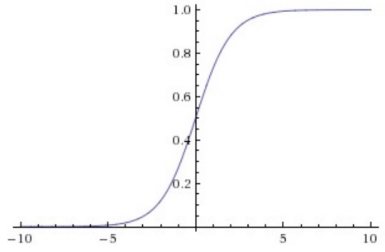
(this line computes all of the gradients for you.)



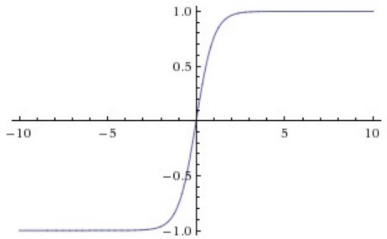
Important tricks for training neural networks

- **The learning rate is extremely important.** Your model may not learn if you set the rate too high or too low. Often, you want to start the rate high and decrease it over the course of learning. There are variants of gradient descent such as “Adam” (Kingma & Ba, 2017) that are faster and automatically adjust the learning rate.
- **Mini-batch learning.** Usually we don’t update the weights after every input pattern. Instead, we present a set of patterns in a “batch,” add their gradients together, and compute a single update to the weights for the whole mini-batch. This is more stable and usually much faster (especially if training your network on a GPU)
- **The are much better activation functions than the “sigmoid”.**

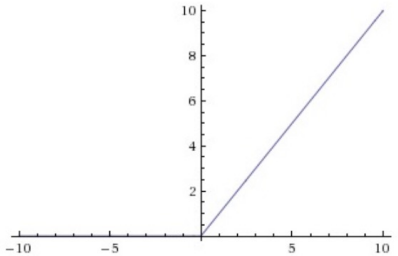
sigmoid (bad)



tanh (better)



ReLU (best; train much faster)



- **Classification requires a different loss and activation function.**

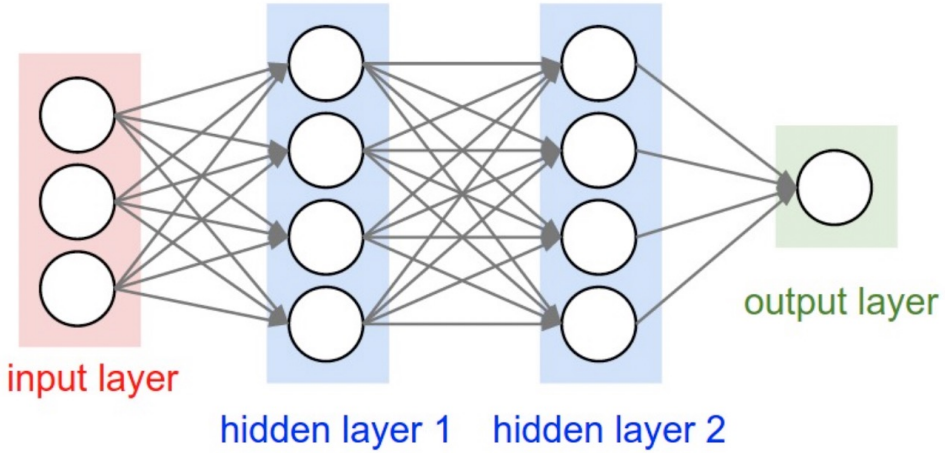
softmax output layer (for c possible classes)

$$g(\mathbf{net}_i) = \frac{e^{\mathbf{net}_i}}{\sum_c e^{\mathbf{net}_c}}$$

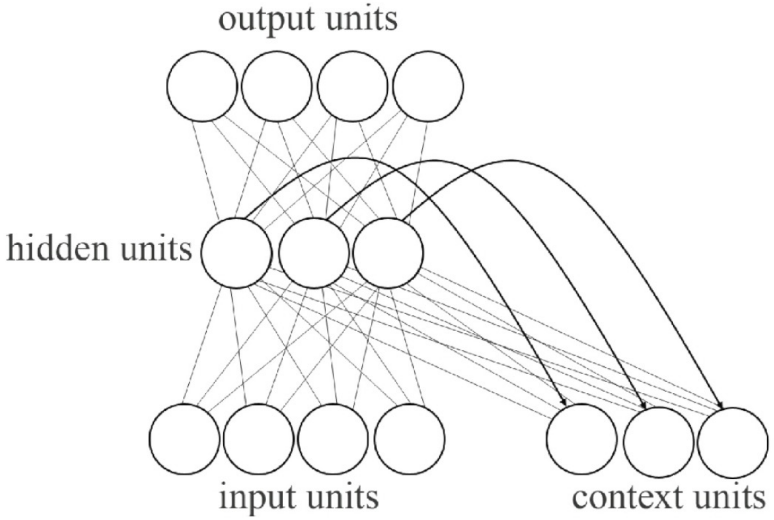
negative log-likelihood loss

A wide range of architectures

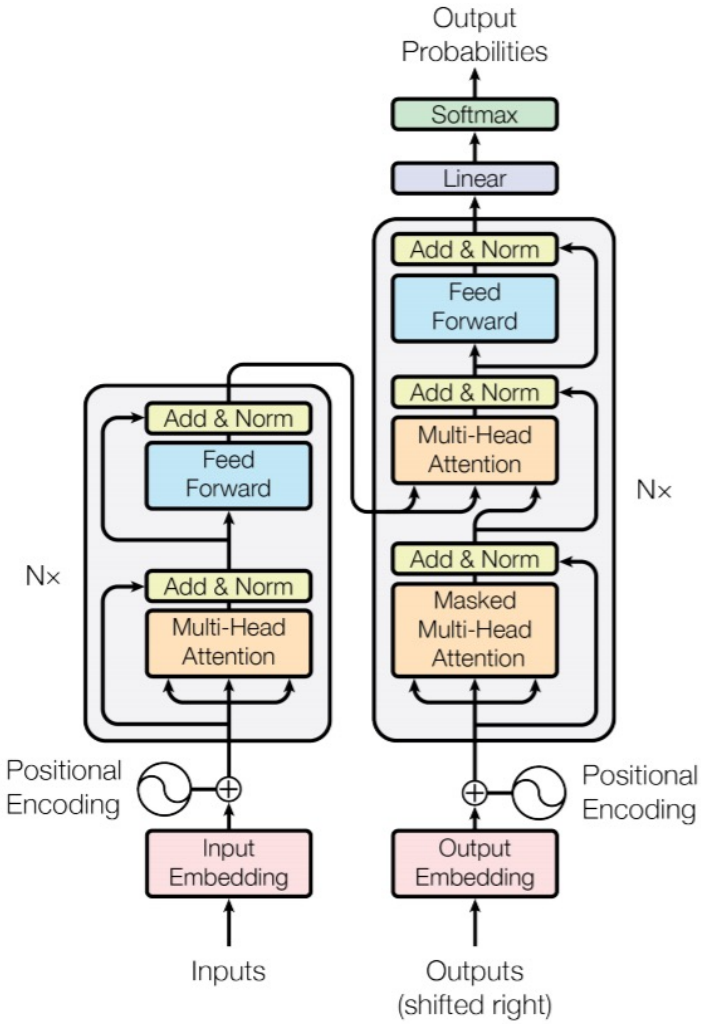
Deep fully-connected neural network



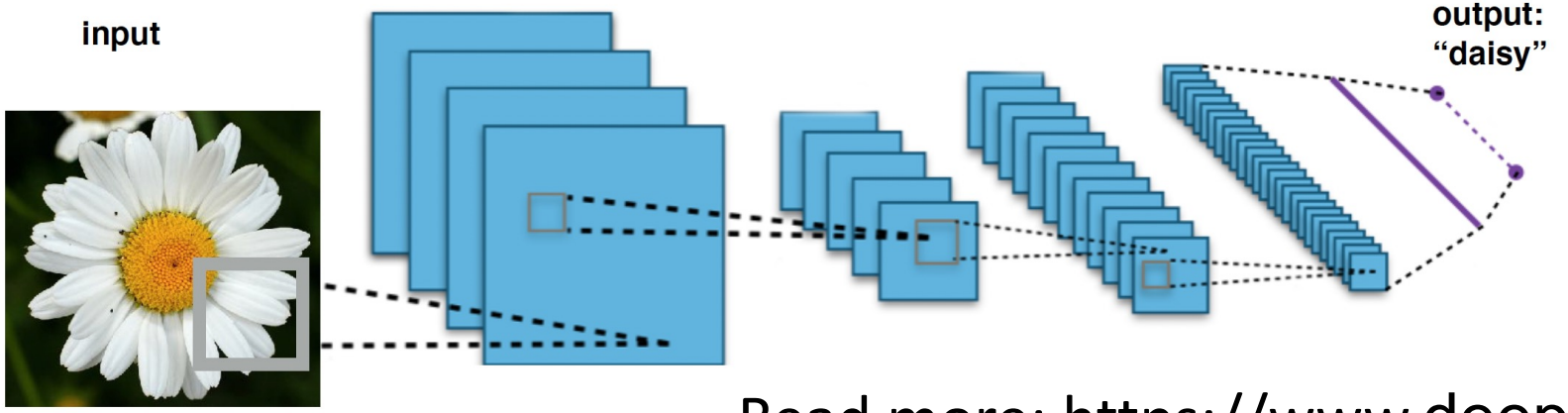
Recurrent neural network



Transformer



Deep convolutional neural network



Read more: https://www.deeplearningbook.org/

Extending the Bayesian toolbox

- Neural networks
- Inference as optimization
- Neural amortized inference

Inference as optimization

- Maximum a posteriori (MAP) optimization

- Differentiable probabilistic programming

Inference as optimization

- Maximum a posteriori (MAP) optimization

- Differentiable probabilistic programming

MAP optimization

- Recall sampling-based inference: sample from posterior distribution

- But sometimes we want to get the hypothesis with the highest posterior probability, i.e., maximum a posteriori (MAP)

$$h^* = \text{argmax}_{h \in H}\, p(h|D)$$
$$= \text{argmax}_{h \in H}\, p(D|h)p(h)/Z$$
$$= \text{argmax}_{h \in H}\, p(D|h)p(h)$$

Optimized via gradient ascent

MAP optimization in Gen

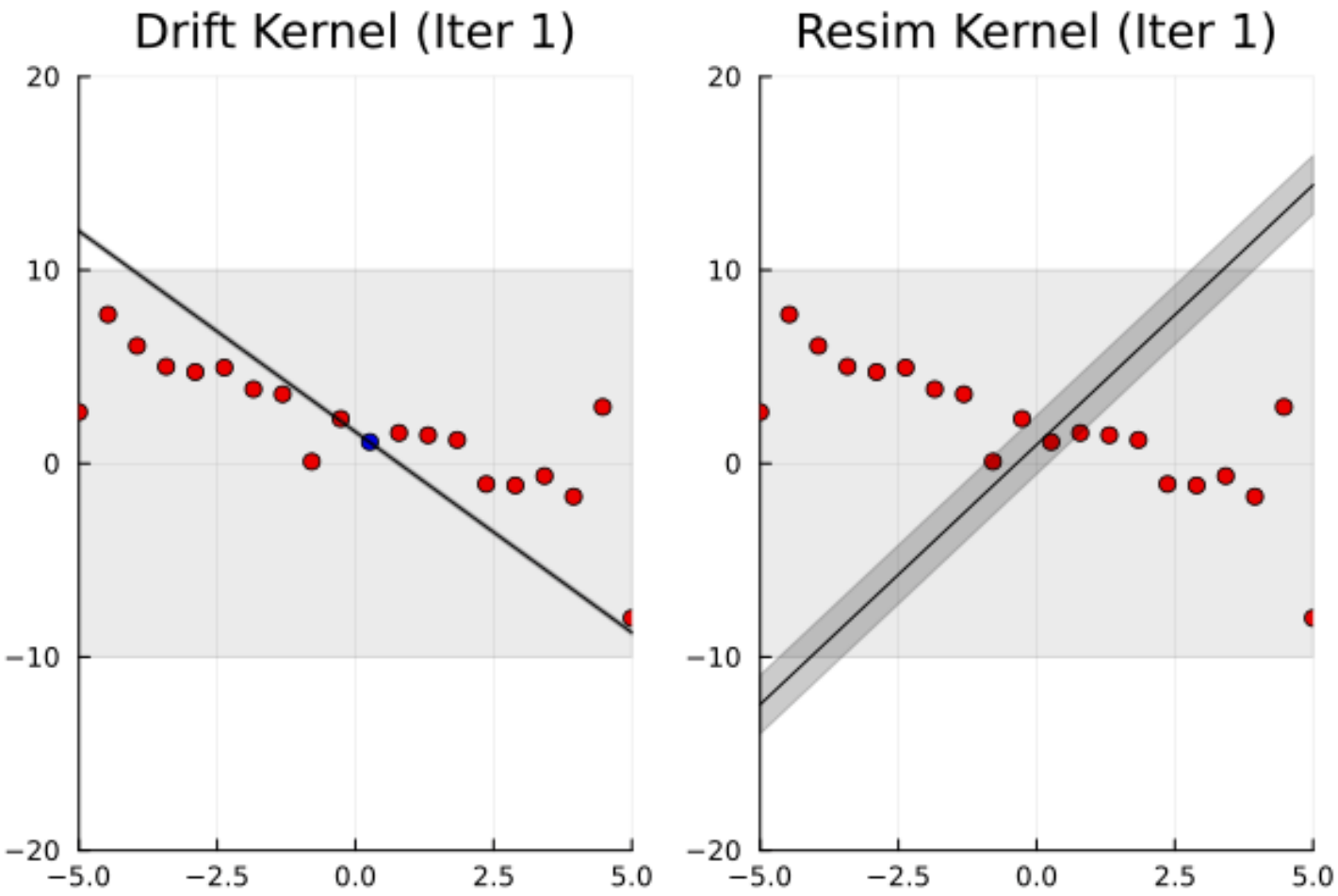
- map_optimize()

```
new_trace = map_optimize(trace, selection::Selection,
max_step_size=0.1, min_step_size=1e-16, …)
```

- Perform backtracking gradient ascent to optimize the log probability of the trace over selected continuous choices (chain rule on trace!)
- argmax(get_score(trace))
- Compute gradient at addresses B (only continuous variables) given input x

$$\nabla_B \log p(t; x)$$

Recall: MCMC for Bayesian curve fitting



MAP for Bayesian curve fitting

Update continuous variables using map_optimize()

```
tr = map_optimize(tr, select(:slope, :intercept), max_step_size=1., min_step_size=1e-5)
tr = map_optimize(tr, select(:noise), max_step_size=1e-2, min_step_size=1e-5)
```

Update discrete variables using update()

```
function is_outlier_map_update(tr)
    (xs,) = get_args(tr)
    for i=1:length(xs)
        constraints = choicemap(:prob_outlier => 0.1)
        constraints[:data => i => :is_outlier] = false
        (trace1,) = update(tr, (xs,), (NoChange(),), constraints)
        constraints[:data => i => :is_outlier] = true
        (trace2,) = update(tr, (xs,), (NoChange(),), constraints)
        tr = (get_score(trace1) > get_score(trace2)) ? trace1 : trace2
    end
    tr
end;
```

MAP for Bayesian curve fitting

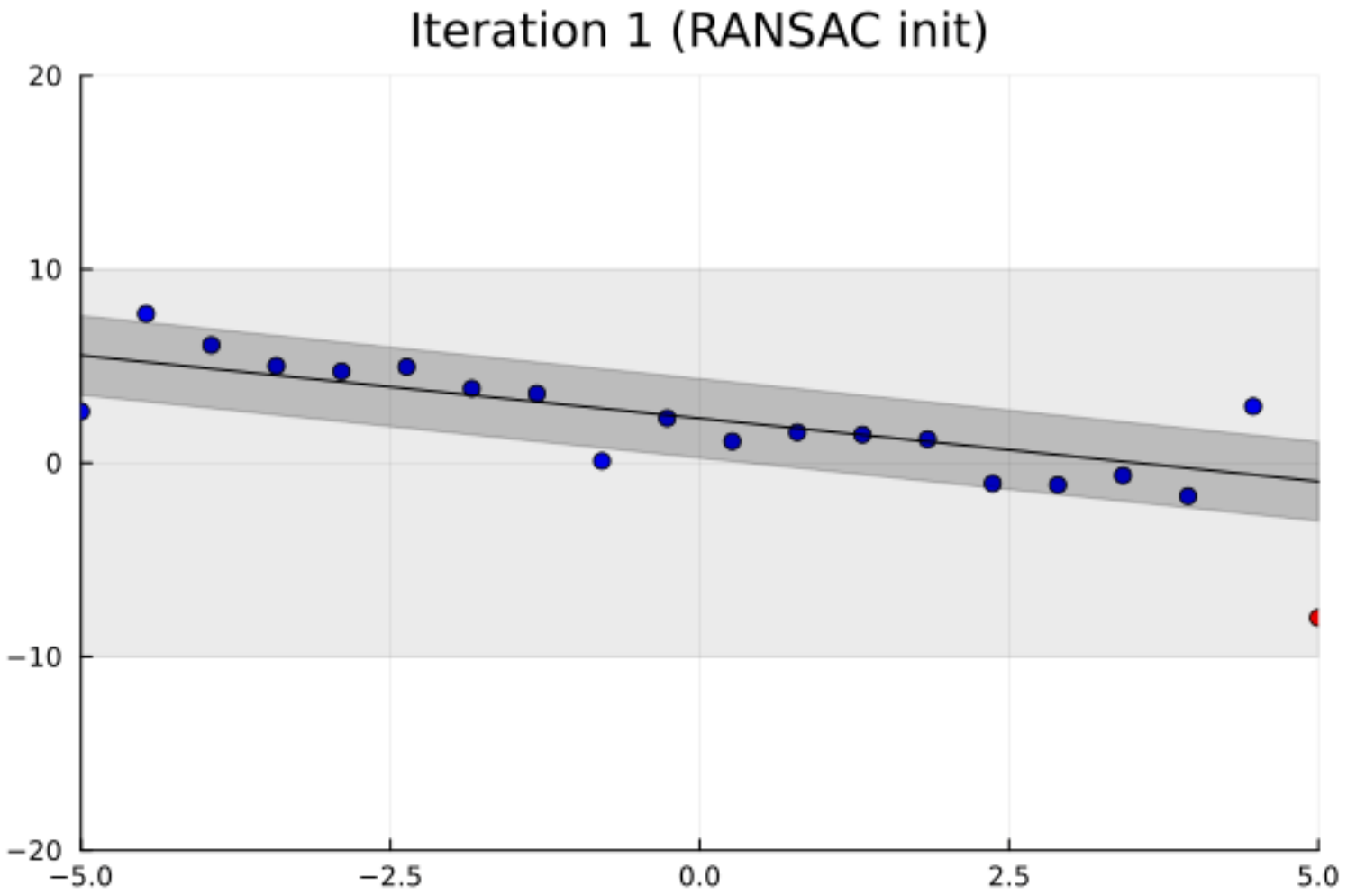
MCMC (RANSAC MH)

```
function map_inference(xs, ys, observations)
    (slope, intercept) = ransac(xs, ys, RANSACParams(10, 3, 1.))
    slope_intercept_init = choicemap()
    slope_intercept_init[:slope] = slope
    slope_intercept_init[:intercept] = intercept
    (tr, _) = generate(regression_with_outliers, (xs,), merge(observations, slope_intercept_init))
    for iter=1:5
        tr = ransac_update(tr)
    end
```

MAP optimization

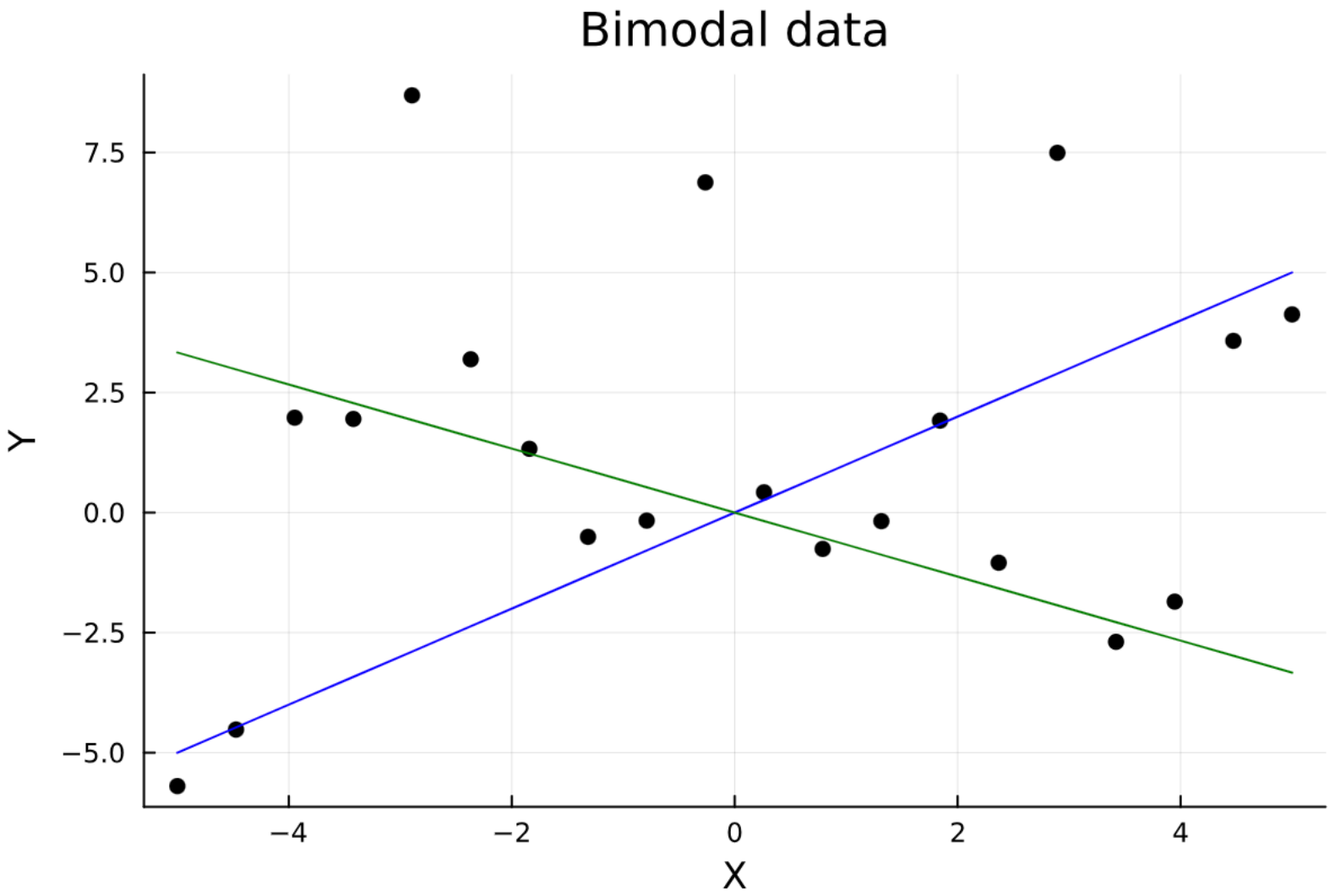
```
    for iter = 1:20
        # Take a single gradient step on the line parameters.
        tr = map_optimize(tr, select(:slope, :intercept), max_step_size=1., min_step_size=1e-5)
        tr = map_optimize(tr, select(:noise), max_step_size=1e-2, min_step_size=1e-5)

        # Choose the most likely classification of outliers.
        tr = is_outlier_map_update(tr)

        # Update the prob outlier
        choices = get_choices(tr)
        optimal_prob_outlier = count(i -> choices[:data => i => :is_outlier], 1:length(xs)) / length(xs)
        optimal_prob_outlier = min(0.5, max(0.05, optimal_prob_outlier))
        (tr, _) = update(tr, (xs,), (NoChange(),), choicemap(:prob_outlier => optimal_prob_outlier))
    end
    tr
end
```

MAP for Bayesian curve fitting

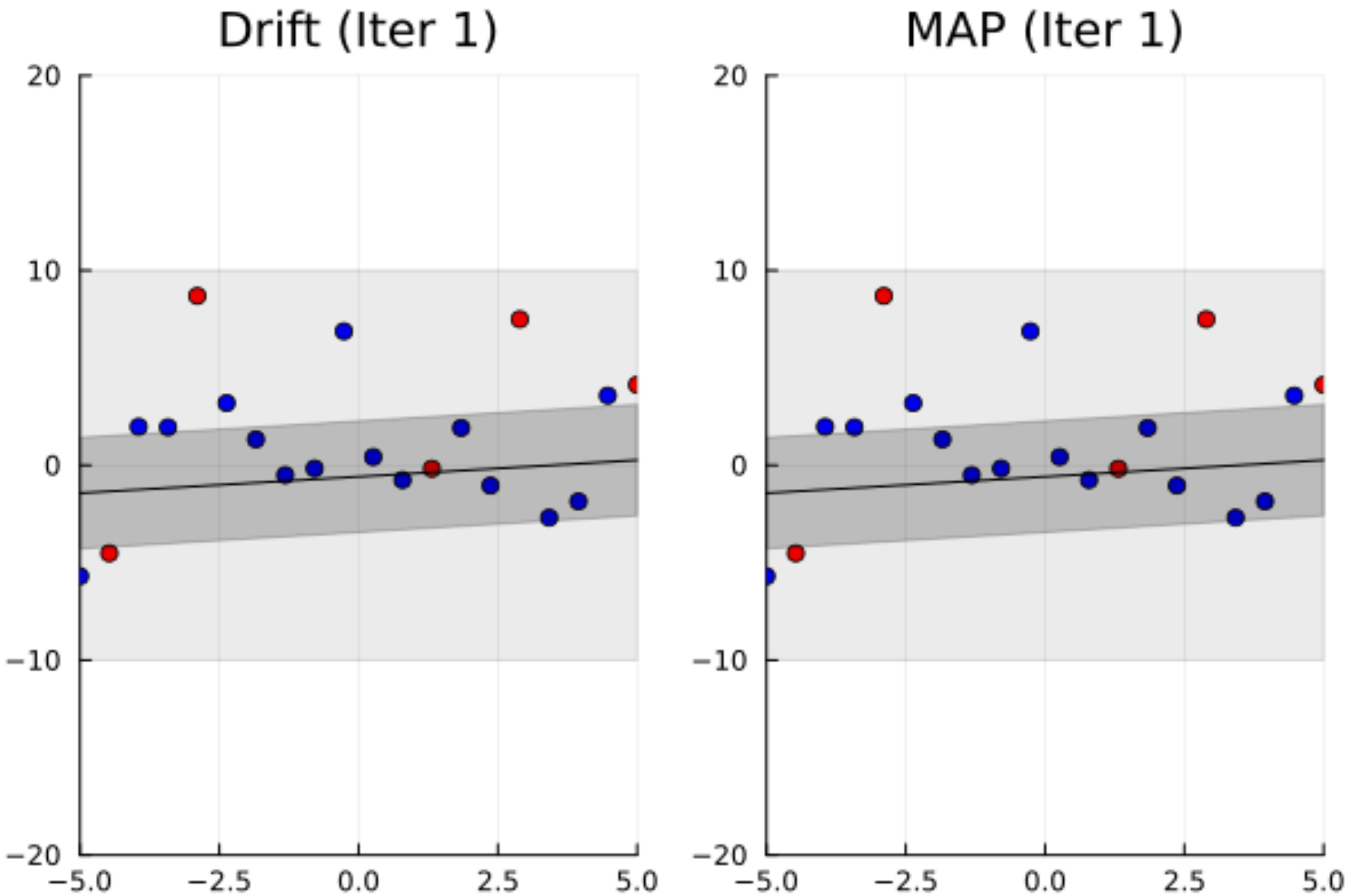


Score after ransac: -44.740896879572254. Final score: -41.01265555802364.

Sampling vs MAP optimization



Sampling vs MAP optimization



How often does a drift / MAP kernel visit both modes?

25 runs with different initial line parameters

```
WITH INITIAL SLOPE 1 AND INTERCEPT 0
TOTAL RUNS EACH: 25

       times neg. slope    times pos. slope

drift: 25                  20

MAP:   14                    15


WITH INITIAL SLOPE -1 AND INTERCEPT 0
TOTAL RUNS EACH: 25

       times neg. slope    times pos. slope

drift: 25                  13

MAP:   17                    9


WITH INITIAL SLOPE -0.6257688854452014 AND INTERCEPT -0.32620717564321533
TOTAL RUNS EACH: 25

       times neg. slope    times pos. slope

drift: 25                  12

MAP:   19                    7
```

Inference as optimization

- Maximum a posteriori (MAP) estimation

- Differentiable probabilistic programming

Differentiable generative models

- A generative function with trainable parameters

$$y \sim G(x; \theta)$$

- Differentiable probabilistic programming:
  - Define a generative model with trainable parameters in Gen
  - Train the models with autodiff in Gen via stochastic gradient ascent (to max the log probability of trace)

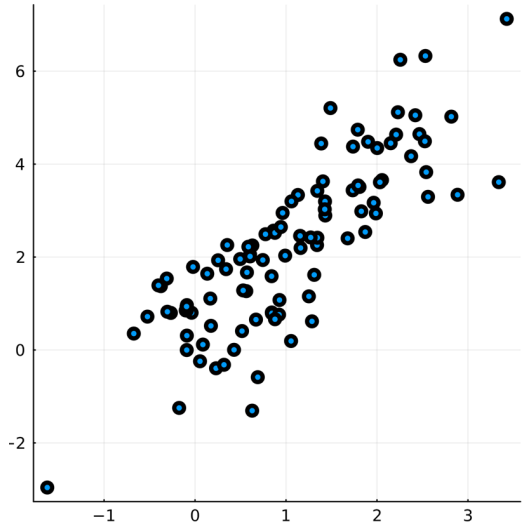
```
arg_grads = accumulate_param_gradients!(trace, retgrad=nothing, scale_factor=1.)
```

$$\nabla_\Theta \log P(t;x)$$

Example

- A generative model with fixed parameters

```
@gen function model_fixed_params()
    x_mu::Float64 = 1.0
    a::Float64 = 1.5
    b::Float64 = 0.7
    x ~ normal(x_mu, 1.)
    y ~ normal(a * x + b, 1.)
end
```

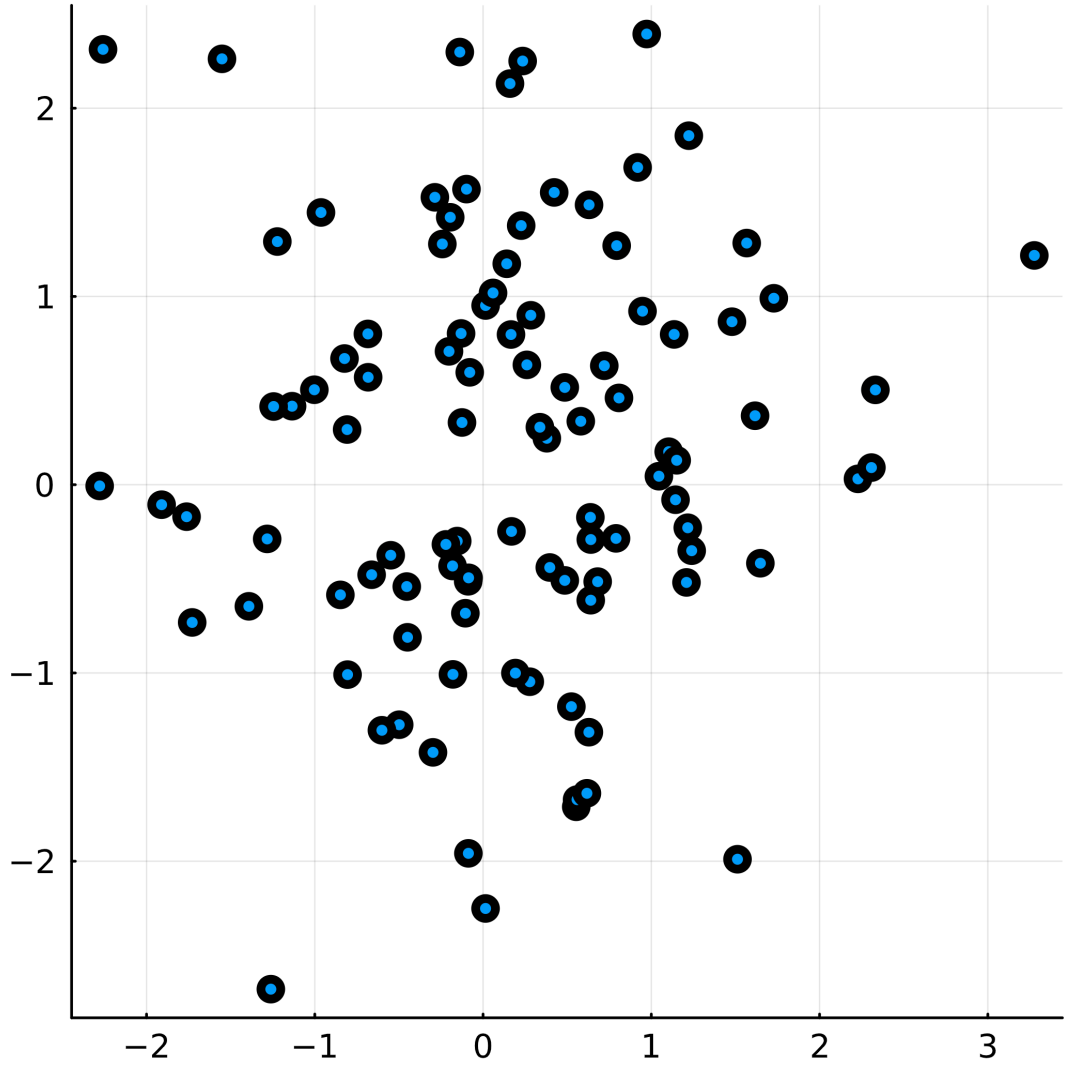


- Previously, we assume priors for the parameters and conduct MAP inference
- Here, let's make these trainable parameters, and train them in a way just like how we can train standard NNs

Differentiable generative function with trainable params

```
@gen function model_trainable()
    @param center::Float64
    @param a::Float64
    @param b::Float64
    x ~ normal(center, 1.)
    y ~ normal(a * x + b, 1.)
end
```

Data generated by the function with initial parameters



Data loader

```
function data_generator()
    dgp_trace = Gen.simulate(model_fixed_params, ())
    constraints = Gen.choicemap()
    constraints[:x] = dgp_trace[:x]
    constraints[:y] = dgp_trace[:y]
    return ((), constraints)
end
```

Returns the inputs and outputs (constraints) of a data point

This can also be reading a training example from the training set, which is more common in real applications.

Differentiable generative function with trainable params

First initialize the parameters

Define optimizer

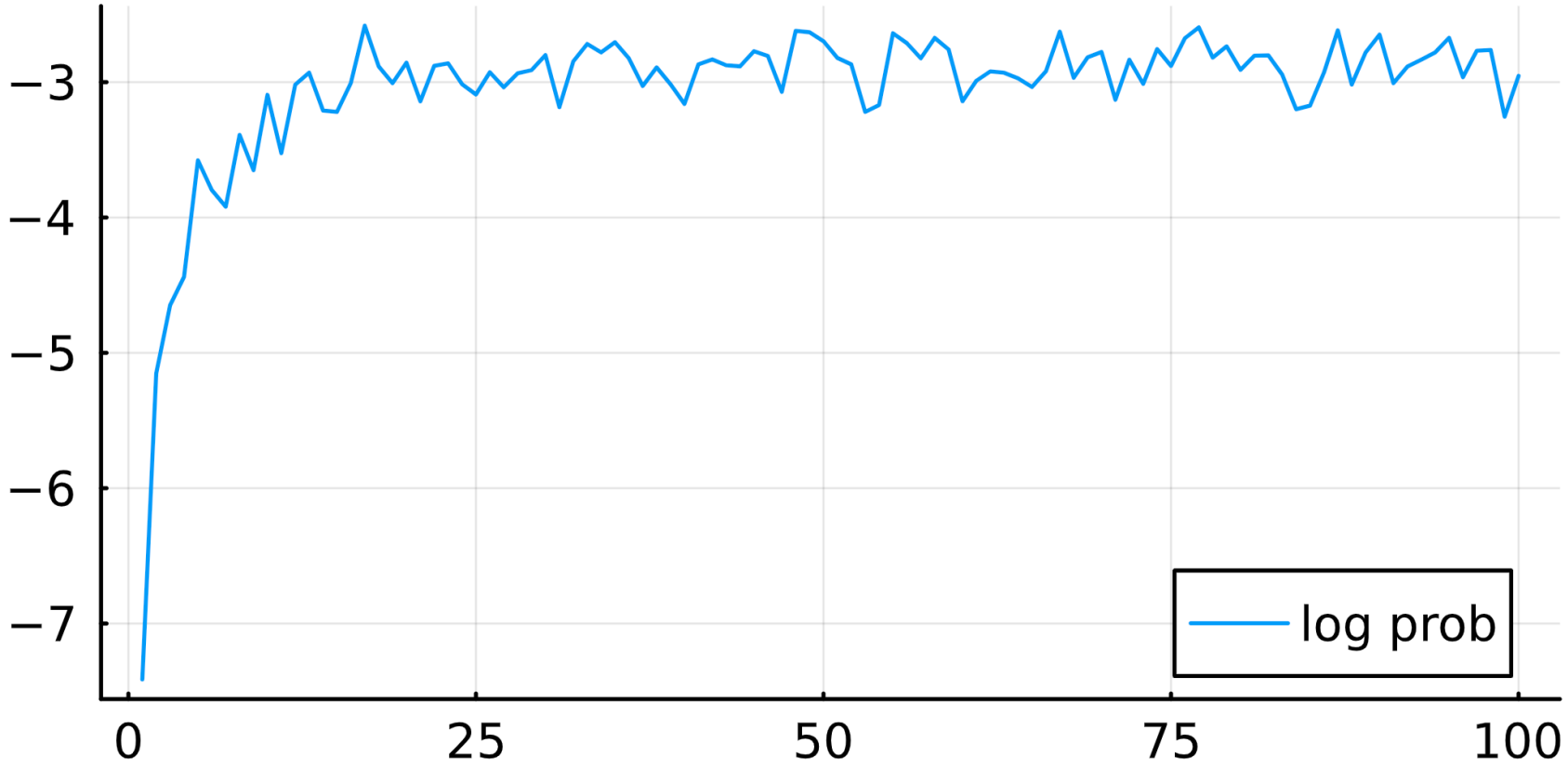
Compute log prob & gradient for data points in a batch

Update once

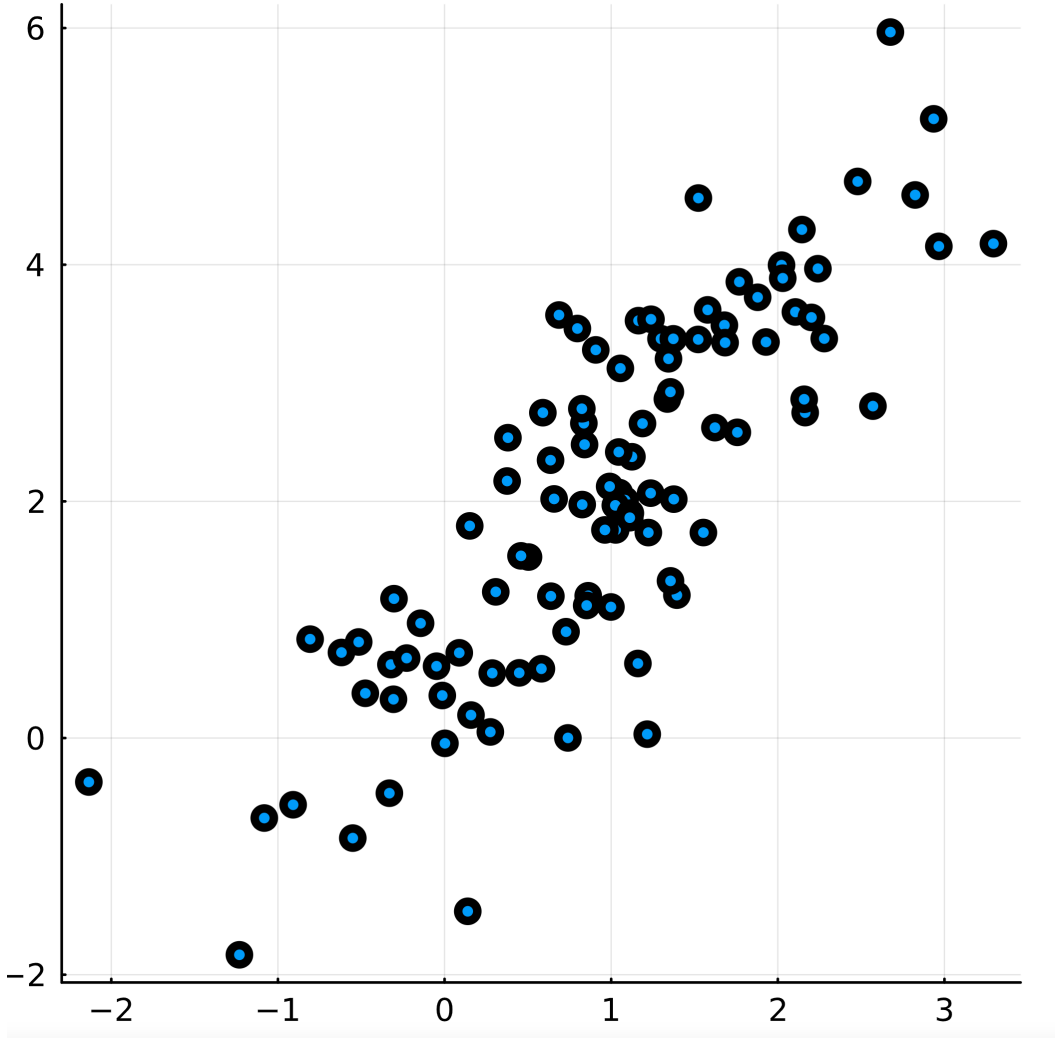
```
function train_model()
    init_param!(model_trainable, :a, 0.1)
    init_param!(model_trainable, :b, 0.1)
    init_param!(model_trainable, :center, 0.1)
    max_iter = 100
    batch_size = 50
    loss = []
    update = ParamUpdate(FixedStepGradientDescent(0.001), model_trainable)
    for iter=1:max_iter
        objective = 0
        for k in 1:batch_size
            inputs, contraints = data_generator()
            trace, weight = Gen.generate(model_trainable, inputs, contraints)
            objective += weight
            accumulate_param_gradients!(trace)
        end
        push!(loss, objective/batch_size)
        apply!(update)
    end
    return loss
end
```

Training curve

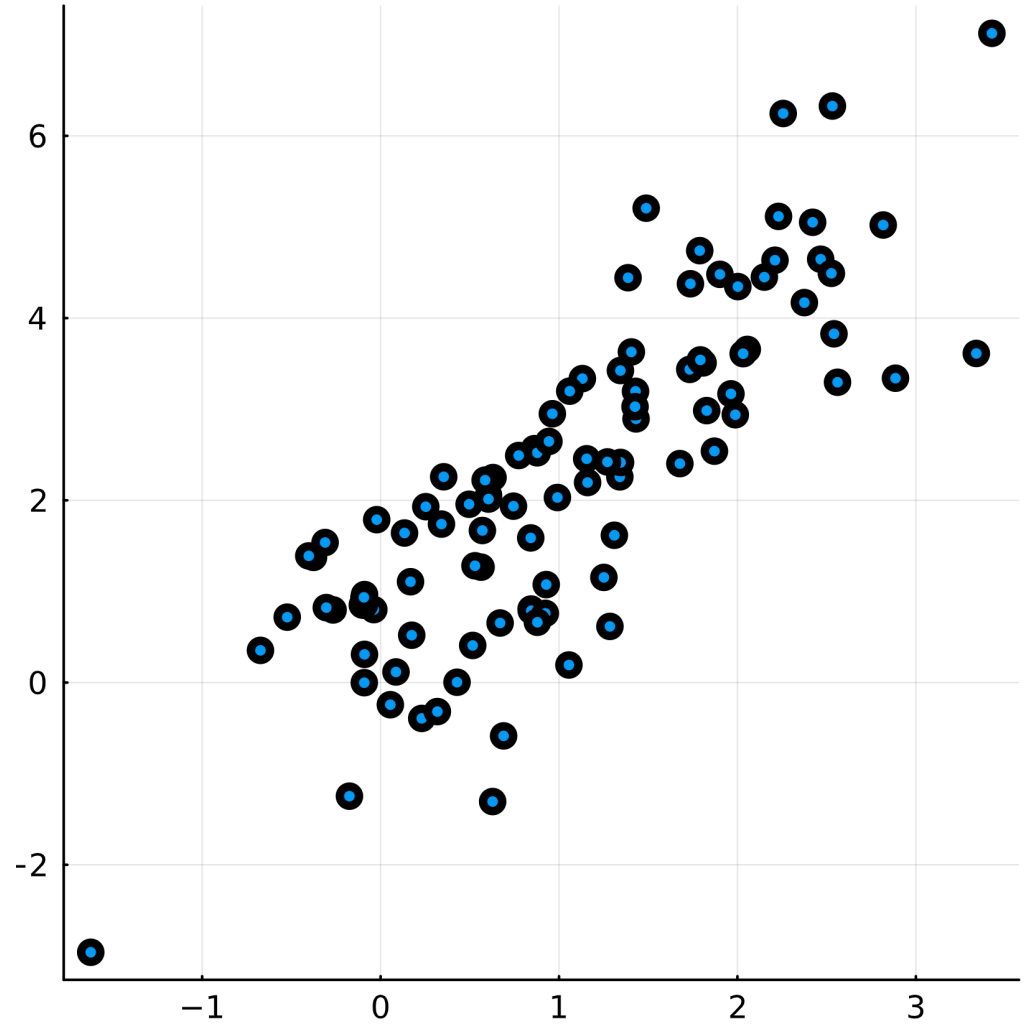
```
loss = train_model()
plot(loss, labels="loss", thickness_scaling=3.5, size=(1600, 800))
```



Data generated by the function with trained parameters



Data generated by the GT function



A simple implementation using Gen.train!()

```
init_param!(model_trainable, :a, 0.1)
init_param!(model_trainable, :b, 0.1)
init_param!(model_trainable, :center, 0.1)

update = ParamUpdate(FixedStepGradientDescent(0.001), model_trainable)

scores = Gen.train!(model_trainable, data_generator, update,
    num_epoch=100, epoch_size=100, num_minibatch=2, minibatch_size=50, evaluation_size=100, verbose=true);
```

- An epoch:
  - Go through all data (epoch_size defines #datapoints)
- At each epoch, multiple iterations (num_minibatch):
  - At each iteration:
  - Get a minibatch (with #minibatch_size datapoints)
  - Compute gradient of all datapoints in this minibatch and update parameters using the specified optimizer
- After each epoch, evaluate log prob with #evaluation_size datapoints

A simple implementation using Gen.train!()

```
init_param!(model_trainable, :a, 0.1)
init_param!(model_trainable, :b, 0.1)
init_param!(model_trainable, :center, 0.1)

update = ParamUpdate(FixedStepGradientDescent(0.001), model_trainable)

scores = Gen.train!(model_trainable, data_generator, update,
    num_epoch=100, epoch_size=100, num_minibatch=2, minibatch_size=50, evaluation_size=100, verbose=true);
```

```
epoch 1: generating 100 training examples...
epoch 1: training using 2 minibatches of size 50...
epoch 1: evaluating on 100 examples...
epoch 1: est. objective value: 0.0
epoch 2: generating 100 training examples...
epoch 2: training using 2 minibatches of size 50...
epoch 2: evaluating on 100 examples...
epoch 2: est. objective value: 0.0
epoch 3: generating 100 training examples...
epoch 3: training using 2 minibatches of size 50...
epoch 3: evaluating on 100 examples...
epoch 3: est. objective value: 0.0
epoch 4: generating 100 training examples...
epoch 4: training using 2 minibatches of size 50...
epoch 4: evaluating on 100 examples...
epoch 4: est. objective value: 0.0
```