```
begin
    import Pkg
    # activate the shared project environment
    Pkg.activate(Base.current_project())
    using Omega, Distributions, UnicodePlots, OmegaExamples
end
```

```
  Activating project at `~/Documents/GitHub/Omega.jl/OmegaExamples`
```

```
DiagNormal = OmegaExamples.DiagNormal
```

```
DiagNormal = OmegaExamples.DiagNormal
```

Fitting curves with neural nets

First recall our exercise inferring an unknown curve using polynomials:

```
obs_data =
  [(x = -4, y = 69.7664), (x = -3, y = 36.6359), (x = -2, y = 19.9524), (x = -1, y = 4.81949), (
```

```
obs_data = [(x = -4, y = 69.76636938284166),
    (x = -3, y = 36.63586217969598),
    (x = -2, y = 19.95244368751754),
    (x = -1, y = 4.819485497724985),
    (x = 0, y = 4.027631414787425),
    (x = 1, y = 3.755022418210824),
    (x = 2, y = 6.557548104903805),
    (x = 3, y = 23.922485493795072),
    (x = 4, y = 50.69924692420815)]
```

```
make_poly (generic function with 1 method)
```

```
make_poly(as) = x -> sum(map(i -> as[i]*x^(i-1), 1:length(as)))
```

```
coeffs =  Mv(1:4, Distributions.Normal{Float64}(μ=0.0, σ=2.0))
```

```
coeffs = manynth(Normal(0, 2), 1:4)
```

```
order =
-7470379617070443487@Distributions.Categorical{Float64, Vector{Float64}}(support=Base.OneTo
```

```
order = @~ Categorical([0.25, 0.25, 0.25, 0.25])
```

```
f (generic function with 1 method)
```

```
f(ω) = make_poly(coeffs(ω)[order(ω) + 1])
```

```
obs_fn (generic function with 1 method)
```

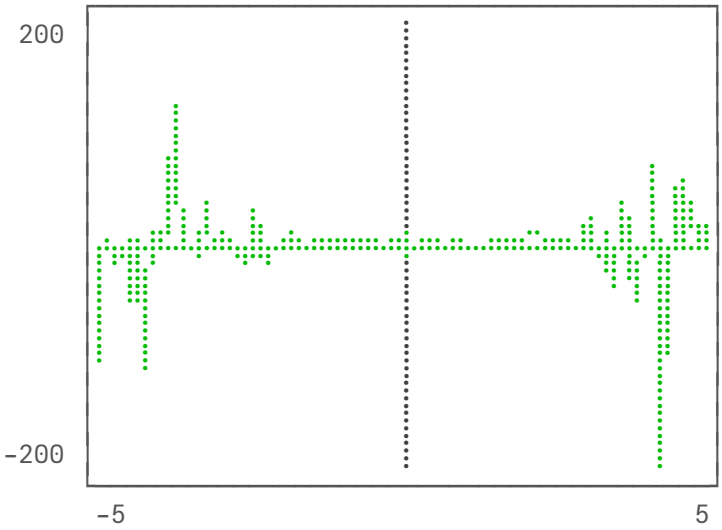
```
obs_fn(ω) =
    all((d ~ Normal(obs_data[d].x, 0.1))(ω) == obs_data[d].y for d in
    1:length(obs_data))
```

```
post =
  Conditional(#7 (generic function with 1 method), obs_fn (generic function with 1 method))
```

```
post = (@joint order coeffs) |ᶜ obs_fn
```

```
post_fn_samples (generic function with 1 method)
```

```
function post_fn_samples(rng)
    ps = Float64[]
    for x in rng
        as = first(randsample(post, 1, alg = MH))
        push!(ps, make_poly(as.coeffs[1:as.order])(x))
    end
    ps
end
```




```
lineplot(-5:0.1:5, post_fn_samples(-5:0.1:5))
```

Another approach to this curve fitting problem is to choose a family of functions that we think is flexible enough to capture any curve we might encounter. One possibility is to simply fix the order of the polynomial to be a high number – try fixing the order to 3 in the above example.

An alternative is to construct a class of functions by composing matrix multiplication with simple non-linearities. Functions constructed in this way are called *artificial neural nets*. Let's explore learning with this class of functions:

```
dm = 10
```

```
dm = 10
```

```
σ (generic function with 1 method)
```

```
σ(z) = one(z) / (one(z) + exp(-z))
```

```
make_fn (generic function with 1 method)
```

```
make_fn(m1, m2, b1) = x ->  m2 * σ.(m1 * x .+ b1)
```

```
m1 (generic function with 1 method)
```

```
m1(i, ω::Ω) = ((@uid, i) ~ DiagNormal(zeros(dm), ones(dm)))(ω)
```

```
b1 (generic function with 1 method)
```

```
b1(i, ω::Ω) = ((@uid, i) ~ DiagNormal(zeros(dm), ones(dm)))(ω)
```

```
m2 (generic function with 1 method)
```

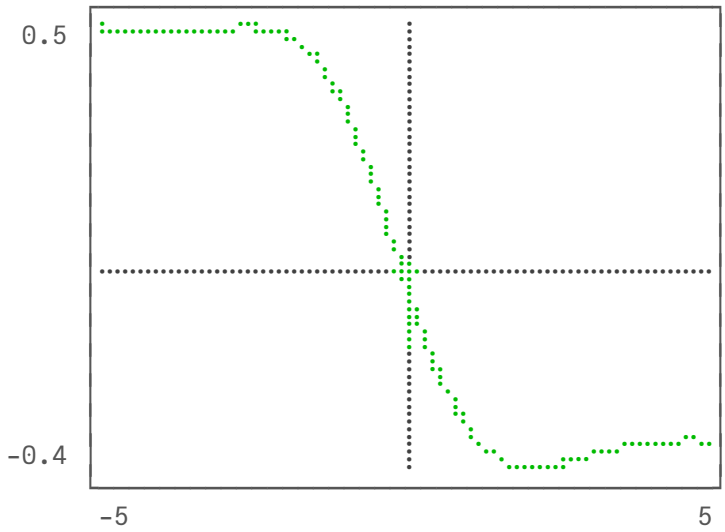
```
m2(i, ω::Ω) = transpose(((@uid, i) ~ DiagNormal(zeros(dm), ones(dm)))(ω))
```

```
posterior (generic function with 1 method)
```

```
function posterior(ω, data)
    for (i, d) in enumerate(data)
        cond!(ω, (i ~ Normal(d.x, 0.1))(ω) == d.y)
    end
    return (m1 = (@~ m1)(ω), m2 = (@~ m2)(ω), b1 = (@~ b1)(ω))
end
```

```
post_func_samples (generic function with 1 method)
```

```
post_func_samples(xs) =
    map(make_fn(first(randsample(ω -> posterior(ω, obs_data), 1, alg = MH))...), xs)
```




```
lineplot(-5:0.1:5, post_func_samples(-5:0.1:5))
```

Just as the order of a polynomial affects the complexity of functions that can result, the size and number of the *hidden layers* affect the complexity of functions for neural nets. Try changing `dm` (the size of the single hidden layer) in the above example – pay particular attention to how the model generalizes out of the [-4,4] training interval.

Neural nets are a very useful class of functions because they are very flexible, but can still (usually) be learned by maximum likelihood inference.

Gaussian processes

Given the importance of the hidden dimension `hd`, you might be curious what happens if we let it get really big. In this case `y` is the sum of a very large number of terms. Due to the central limit theorem, and assuming uncertainty over the weight matrices, this sum converges on a Gaussian as the width `hd` goes to infinity. That is, infinitely "wide" neural nets yield a model where `f(x)` is Gaussian distributed for each `x`, and further (it turns out) the covariance among different `x`s is also Gaussian. This kind of model is called a *Gaussian Process*.

Deep generative models

So far in this chapter, we have considered *supervised* learning, where we are trying to learn the dependence of `y` on `x`. This is a special case because we only care about predicting `y`. Neural nets are particularly good at this kind of problem. However, many interesting problems are *unsupervised*: we get a bunch of examples and want to understand them by capturing their distribution.

Having shown that we can put an unknown function in our supervised model, nothing prevents us from putting one anywhere in a generative model! Here we learn an unsupervised model of x,y pairs, which are generated from a latent random choice passed through a (learned) function.

```
hd = 10
```

```
hd = 10
```

```
ld = 2
```

```
ld = 2
```

```
out_sig =  [1.0, 1.0]
```

```
out_sig = ones(2)
```

```
m1_ (generic function with 1 method)
```

```
m1_(i, ω) =
    reduce(hcat, map(l -> ((@uid, i, l) ~ DiagNormal(zeros(hd), ones(hd)))(ω), 1:ld))
```

```
b1_ (generic function with 1 method)
```

```
b1_(i, ω) = ((@uid, i) ~ DiagNormal(zeros(hd), ones(hd)))(ω)
```

```
m2_ (generic function with 1 method)
```

```
m2_(i, ω) =
    reduce(hcat, map(l -> ((@uid, i, l) ~ DiagNormal(zeros(hd), ones(hd)))(ω), 1:ld))'
```

```
f (generic function with 2 methods)
```

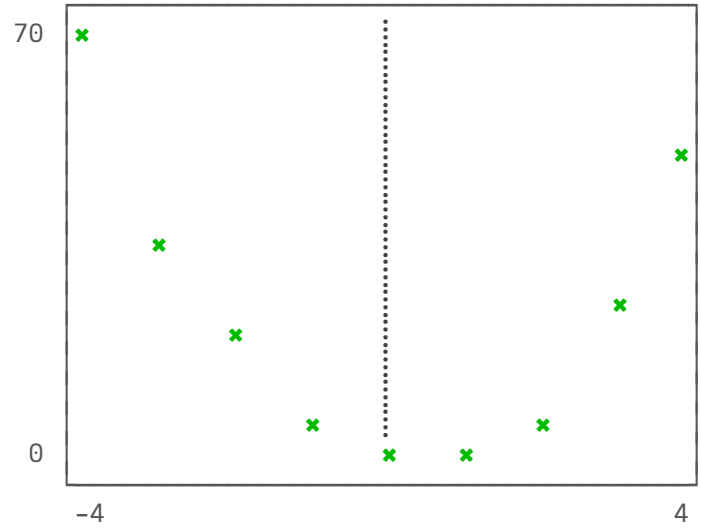
```
f(i, ω) = make_fn(m1_(i, ω), m2_(i, ω), b1_(i, ω))
```

```
sample_XY (generic function with 1 method)
```

```
sample_XY(i, ω) = randsample(i~f)((i~DiagNormal(zeros(ld), ones(ld)))(ω))
```

```
post_unsupervised (generic function with 1 method)
```

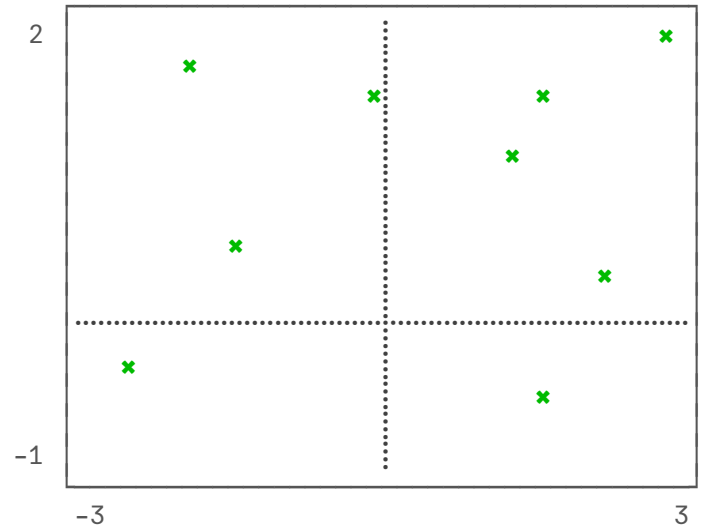
```
function post_unsupervised(ω, data)
    means = manynth(sample_XY, 1:length(data))(ω)
    for (i, d) in enumerate(data)
        cond!(ω, ((@uid, i)~ DiagNormal(means[i], out_sig))(ω) == [d.x, d.y])
    end
    return means
end
```




```
scatterplot(map(o -> o.x, obs_data), map(o -> o.y, obs_data), marker = :xcross)
```

```
samples_unsupervised =
 [[2.21797, 0.321657], [-1.44137, 0.48746], [1.315, 1.13978], [-1.90246, 1.79007], [-2.51726,
```

```
samples_unsupervised = first(randsample(ω -> post_unsupervised(ω, obs_data), 1, alg = MH))
```




```
scatterplot(first.(samples_unsupervised), last.(samples_unsupervised), marker = :xcross)
```

Models of this sort are often called *deep generative models* because the (here not very) deep neural net is doing a large amount of work to generate complex observations.

Notice that while this model reconstructs the data well, the posterior predictive looks like noise. That is, this model *over-fits* the data. To ameliorate over-fitting, we might try to limit the expressive capacity of the model. For instance by reducing the latent dimension for z (i.e. `ld`) to 1, since we know that the data actually lie near a one-dimensional subspace. (Try it!) However that model usually simply over-fits in a more constrained way.

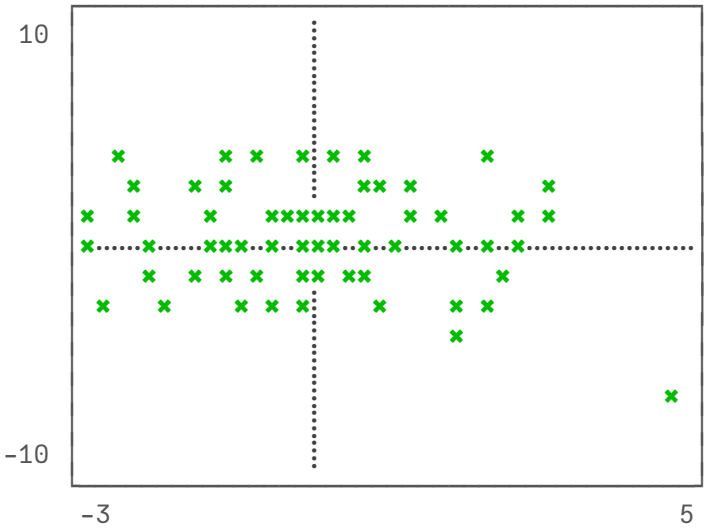
Here, we instead increase the data (by a lot) with the flexible model (warning, this takes much longer to run):

```
obs_data_new =
  [(x = -4.0, y = 16.0), (x = -3.9, y = 15.21), (x = -3.8, y = 14.44), (x = -3.7, y = 13.69), (x
```

```
1 obs_data_new = map(x -> (x = x, y = x*x), -4:0.1:4)
```

```
samples_unsupervised_new =
  [[1.36477, 1.94039], [0.677148, 3.6464], [1.92868, 0.142995], [-2.93733, 1.51155], [-2.34039
```

```
1 samples_unsupervised_new =
2     first(randsample(ω -> post_unsupervised(ω, obs_data_new), 1, alg = MH))
```




```
1 scatterplot(first.(samples_unsupervised_new), last.(samples_unsupervised_new), marker
  = :xcross)
```

Notice that we still fit the data reasonably well, but now we generalize a bit more usefully. With even more data, perhaps we'd capture the distribution even better? But the slowdown in inference time would be intolerable....

```
1 # Minibatches
```