Learning (deep) continuous functions

Fitting curves with neural nets

First recall our exercise inferring an unknown curve using polynomials:

x

```
var observedData = [{"x":-4,"y":69.76636938284166},{"x":-3,"y":36.63586217969598},{"x":-2,"y":19.95244368751754},{"x":-1,"y":4.819485497720
```

```
var inferOptions = {method: 'optimize', samples: 100, steps: 2000, optMethod: {adam: {stepSize: 0.1}}}
```

```
// a0 + a1*x + a2*x^2 + ...
var makePoly = function(as) {
  return function(x) {
    return sum(map(function(i){as[i]*Math.pow(x,i)}, _.range(as.length)))
  }
}
```

```
var post = Infer(inferOptions,
  function() {
    var coeffs = repeat(4, function() {return gaussian(0,2)})
    var order = discrete([0.25,0.25,0.25,0.25])
    var f = makePoly(coeffs.slice(0,order+1))

    var obsFn = function(datum){
      observe(Gaussian({mu: f(datum.x), sigma: 0.1}), datum.y)
    }
    mapData({data: observedData}, obsFn)

    return {order: order,
            coeffs: coeffs}
  }
)
```

```
print("observed data:")

viz.scatter(observedData)



var postFnSample = function(){
  var p = sample(post)
  return makePoly(p.coeffs.slice(0,p.order+1))
}

print("inferred curves:")

var xs = _.range(-5,5,0.1)

viz.line(xs, map(postFnSample(), xs))

viz.line(xs, map(postFnSample(), xs))

viz.line(xs, map(postFnSample(), xs))
```

run ▼

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:

```
var dm = 10 //try changing this!


var makeFn = function(M1,M2,B1){
  return function(x){
    return T.toScalars(
      // M2 * sigm(x * M1 + B1):
      T.dot(M2,T.sigmoid(T.add(T.mul(M1,x),B1)))
    )[0]}
}


var observedData = [{"x":-4,"y":69.76636938284166},{"x":-3,"y":36.63586217969598},{"x":-2,"y":19.95244368751754},{"x":-1,"y":4.8194854977


var inferOptions = {method: 'optimize', samples: 100, steps: 3000, optMethod: {adam: {stepSize: 0.1}}}


var post = Infer(inferOptions,
  function() {
    var M1 = sample(DiagCovGaussian({mu: zeros([dm, 1]), sigma: ones([dm,1])}))
```

```
    var B1 = sample(DiagCovGaussian({mu: zeros([dm, 1]), sigma: ones([dm,1])}))

    var M2 = sample(DiagCovGaussian({mu: zeros([1, dm]), sigma: ones([1,dm])}))


    var f = makeFn(M1,M2,B1)


    var obsFn = function(datum){
      observe(Gaussian({mu: f(datum.x), sigma: 0.1}), datum.y)
    }
    mapData({data: observedData}, obsFn)


    return {M1: M1, M2: M2, B1: B1}
  }
)

print("observed data:")
viz.scatter(observedData)

var postFnSample = function(){
  var p = sample(post)
  return makeFn(p.M1,p.M2,p.B1)
}
print("inferred curves")
var xs = _.range(-5,5,0.1)
viz.line(xs, map(postFnSample(), xs))
viz.line(xs, map(postFnSample(), xs))
viz.line(xs, map(postFnSample(), xs))
```

run ▼

Just as the order of a polynomial effects the complexity of functions that can result, the size and number of the *hidden layers* effects 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.

The posterior samples from the above model probably all look about the same. This is because the observation noise is very low. Try changing it.

If we actually don't care very much about the uncertainty over functions we learn, or are not optimistic that we can capture the true posterior, we can do *maximum likelihood* inference. Here we choose the guide family to be Delta (technically this is regularized maximum likelihood, because we still have a Gaussian prior over the matrices):

```
var dm = 10


var makeFn = function(M1,M2,B1){
```

```
  return function(x){return T.toScalars(T.dot(M2,T.sigmoid(T.add(T.mul(M1,x),B1))))[0]}
}

var observedData = [{"x":-4,"y":69.76636938284166},{"x":-3,"y":36.63586217969598},{"x":-2,"y":19.95244368751754},{"x":-1,"y":4.819485497720

var inferOptions = {method: 'optimize', samples: 100, steps: 3000, optMethod: {adam: {stepSize: 0.1}}}

var post = Infer(inferOptions,
  function() {
    var M1 = sample(DiagCovGaussian({mu: zeros([dm, 1]), sigma: ones([dm,1])}), {
      guide: function() {return Delta({v: param({dims: [dm, 1]})})}})
    var B1 = sample(DiagCovGaussian({mu: zeros([dm, 1]), sigma: ones([dm,1])}), {
      guide: function() {return Delta({v: param({dims: [dm, 1]})})}})
    var M2 = sample(DiagCovGaussian({mu: zeros([1, dm]), sigma: ones([1,dm])}), {
      guide: function() {return Delta({v: param({dims: [1, dm]})})}})

    var f = makeFn(M1,M2,B1)

    var obsFn = function(datum){
      observe(Gaussian({mu: f(datum.x), sigma: 0.1}), datum.y)
    }
    mapData({data: observedData}, obsFn)

    return {M1: M1, M2: M2, B1: B1}
  }
)

print("observed data:")
viz.scatter(observedData)

var postFnSample = function(){
  var p = sample(post)
  return makeFn(p.M1,p.M2,p.B1)
}
```

```
print("infered curve")
```

```
var xs = _.range(-5,5,0.1)
```

```
viz.line(xs, map(postFnSample(), xs))
```

run ▼

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 (https://en.wikipedia.org/wiki/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 (https://en.wikipedia.org/wiki/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.

```
var hd = 10
```

```
var ld = 2
```

```
var outSig = Vector([0.1, 0.1])
```

```

```

```
var makeFn = function(M1,M2,B1){
```

```
  return function(x){return T.dot(M2,T.sigmoid(T.add(T.dot(M1,x),B1)))}
```

```
}
```

```

```

```
var observedData = [{"x":-4,"y":69.76636938284166},{"x":-3,"y":36.63586217969598},{"x":-2,"y":19.95244368751754},{"x":-1,"y":4.8194854977
```

```

```

```
var inferOptions = {method: 'optimize', samples: 100, steps: 3000, optMethod: {adam: {stepSize: 0.1}}, verbose: true}
```

```

```

```
var post = Infer(inferOptions,
```

```
  function() {
```

```
    var M1 = sample(DiagCovGaussian({mu: zeros([hd,ld]), sigma: ones([hd,ld])}), {
```

```
      guide: function() {return Delta({v: param({dims: [hd, ld]})})}})
```

```
    var B1 = sample(DiagCovGaussian({mu: zeros([hd, 1]), sigma: ones([hd,1])}), {
```

```
      guide: function() {return Delta({v: param({dims: [hd, 1]})})}})
```

```
    var M2 = sample(DiagCovGaussian({mu: zeros([2,hd]), sigma: ones([2,hd])}), {
```

```
      guide: function() {return Delta({v: param({dims: [2,hd]})})}})
```

```
    var f = makeFn(M1,M2,B1)
    var sampleXY = function(){return f(sample(DiagCovGaussian({mu: zeros([ld, 1]), sigma: ones([ld,1])})))}

    var means = repeat(observedData.length, sampleXY)
    var obsFn = function(datum,i){
      observe(DiagCovGaussian({mu: means[i], sigma: outSig}), Vector([datum.x, datum.y]))
    }
    mapData({data: observedData}, obsFn)

    return {means: means,
            pp: repeat(100, sampleXY)}
  }
)

print("observed data:")
viz.scatter(observedData)

print("reconstructed data:")
viz.scatter(map(function(v){return {x: T.toScalars(v)[0], y: T.toScalars(v)[1]}}, sample(post).means))

print("posterior predictive:")
viz.scatter(map(function(v){return {x: T.toScalars(v)[0], y: T.toScalars(v)[1]}}, sample(post).pp))
```

run ▼

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):

```
var hd = 10
var ld = 2
var outSig = Vector([0.1, 0.1])

var makeFn = function(M1,M2,B1){
  return function(x){return T.dot(M2,T.sigmoid(T.add(T.dot(M1,x),B1)))}
```

```
}

var observedData = map(function(x){return {x:x,y:x*x}}, _.range(-4,4,0.1))

var inferOptions = {method: 'optimize', samples: 100, steps: 3000, optMethod: {adam: {stepSize: 0.1}}, verbose: tr

var post = Infer(inferOptions,
  function() {
    var M1 = sample(DiagCovGaussian({mu: zeros([hd,ld]), sigma: ones([hd,ld])}), {
      guide: function() {return Delta({v: param({dims: [hd, ld]})})}})
    var B1 = sample(DiagCovGaussian({mu: zeros([hd, 1]), sigma: ones([hd,1])}), {
      guide: function() {return Delta({v: param({dims: [hd, 1]})})}})
    var M2 = sample(DiagCovGaussian({mu: zeros([2,hd]), sigma: ones([2,hd])}), {
      guide: function() {return Delta({v: param({dims: [2,hd]})})}})

    var f = makeFn(M1,M2,B1)
    var makeData = function(){return f(sample(DiagCovGaussian({mu: zeros([ld, 1]), sigma: ones([ld,1])})))}
    var means = repeat(observedData.length, makeData)
    var obsFn = function(datum,i){
      observe(DiagCovGaussian({mu: means[i], sigma: outSig}), Vector([datum.x, datum.y]))
    }
    mapData({data: observedData}, obsFn)

    return {means: means,
            pp: repeat(100, makeData)}
  }
)

print("observed data:")
viz.scatter(observedData)

print("reconstructed data:")
viz.scatter(map(function(v){return {x: T.toScalars(v)[0], y: T.toScalars(v)[1]}}, sample(post).means))
```

```
print("posterior predictive:")

viz.scatter(map(function(v){return {x: T.toScalars(v)[0], y: T.toScalars(v)[1]}}, sample(post).pp))
```

run ▼

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....

Minibatches and amortized inference

In order to do approximate variational inference with much larger data, we'll use minibatches: the idea is that randomly sub-sampling the data on each step can give us a good enough approximation to the whole data set. In WebPPL this requires only a small hint to `mapData`:

```
var hd = 10

var ld = 2

var outSig = Vector([0.1, 0.1])



var makeFn = function(M1,M2,B1){

  return function(x){return T.dot(M2,T.sigmoid(T.add(T.dot(M1,x),B1)))}

}



var observedData = map(function(x){return {x:x,y:x*x}}, _.range(-4,4,0.1))



var inferOptions = {method: 'optimize', samples: 100, steps: 3000, optMethod: {adam: {stepSize: 0.1}}, verbose: tr



var post = Infer(inferOptions,

  function() {

    var M1 = sample(DiagCovGaussian({mu: zeros([hd,ld]), sigma: ones([hd,ld])}), {

      guide: function() {return Delta({v: param({dims: [hd, ld]})})}})

    var B1 = sample(DiagCovGaussian({mu: zeros([hd, 1]), sigma: ones([hd,1])}), {

      guide: function() {return Delta({v: param({dims: [hd, 1]})})}})

    var M2 = sample(DiagCovGaussian({mu: zeros([2,hd]), sigma: ones([2,hd])}), {

      guide: function() {return Delta({v: param({dims: [2,hd]})})}})



    var f = makeFn(M1,M2,B1)

    var makeData = function(){return f(sample(DiagCovGaussian({mu: zeros([ld, 1]), sigma: ones([ld,1])})))}



    var obsFn = function(datum,i){

      observe(DiagCovGaussian({mu: makeData(), sigma: outSig}), Vector([datum.x, datum.y]))

    }
```

```
    mapData({data: observedData, batchSize: 10}, obsFn)

    return {pp: repeat(observedData.length, makeData)}
  }
)

print("observed data:")
viz.scatter(observedData)

print("posterior predictive:")
viz.scatter(map(function(v){return {x: T.toScalars(v)[0], y: T.toScalars(v)[1]}}, sample(post).pp))
```

run ▼

An issue with this approach is that the latent random choice associated with each data point (inside `makeData`) is chosen fresh on each mini-batch and may not get to be very good before we move on to a new mini-batch. A solution explore in recent work is to *amortize* the inference, that is to learn an approximation mapping from an observation to a guess about the latent choice.

```
var hd = 10
var ld = 2
var outSig = Vector([0.1, 0.1])

var makeFn = function(M1,M2,B1){
  return function(x){return T.dot(M2,T.sigmoid(T.add(T.dot(M1,x),B1)))}
}

var observedData = map(function(x){return {x:x,y:x*x}}, _.range(-4,4,0.1))

var inferOptions = {method: 'optimize', samples: 100, steps: 3000, optMethod: {adam: {stepSize: 0.1}}, verbose: tr
```

```
var post = Infer(inferOptions,
  function() {
    var M1 = sample(DiagCovGaussian({mu: zeros([hd,ld]), sigma: ones([hd,ld])}), {
      guide: function() {return Delta({v: param({dims: [hd, ld]})})}})
    var B1 = sample(DiagCovGaussian({mu: zeros([hd, 1]), sigma: ones([hd,1])}), {
      guide: function() {return Delta({v: param({dims: [hd, 1]})})}})
    var M2 = sample(DiagCovGaussian({mu: zeros([2,hd]), sigma: ones([2,hd])}), {
```

```
      guide: function() {return Delta({v: param({dims: [2,hd]})})}})

    var f = makeFn(M1,M2,B1) //the forward generative model

    var makeData = function(xy){
      var fguide = function(){
        //the inverse (inference) model:
        var q = makeFn(param({dims: [hd, 2]}),
                       param({dims: [ld, hd]}),
                       param({dims: [hd, 1]}))
        return Delta({v: q(xy)})}
      return f(sample(DiagCovGaussian({mu: zeros([ld, 1]), sigma: ones([ld,1])}),
                      {guide: fguide}))
    }

    var obsFn = function(datum,i){
      var xy = Vector([datum.x, datum.y])
      observe(DiagCovGaussian({mu: makeData(xy), sigma: outSig}), xy)
    }
    mapData({data: observedData, batchSize: 10}, obsFn)

    return {pp: repeat(observedData.length,
                       function(){f(sample(DiagCovGaussian({mu: zeros([ld, 1]), sigma: ones([ld,1])})))})}
  }
)

print("observed data:")
viz.scatter(observedData)

print("posterior predictive:")
viz.scatter(map(function(v){return {x: T.toScalars(v)[0], y: T.toScalars(v)[1]}}, sample(post).pp))
```

run ▼

The move to amortized inference is, in one sense, a minor technical change to make inference easier. However, seen from another point of view it makes a radical cognitive claim: we are not only learning a generative model of our observations, we are learning a model of the right posterior inferences to draw from observations. That is, we are learning how to do inference. Some authors have claimed that people do the same – e.g. Gershman and Goodman, 2014 (https://web.stanford.edu/~ngoodman/papers/amortized_inference.pdf).

In recent years there have been many other answers to the technical difficulty of learning deep generative models: generative adversarial networks, normalizing flows, etc.

Reading & Discussion: Readings (/readings/function-learning.html)

Test your knowledge: Exercises (/exercises/function-learning.html)

Next chapter: 14. Mixture models (/chapters/mixture-models.html)