Lecture 12

Non-Linear Function Approximation and Classification

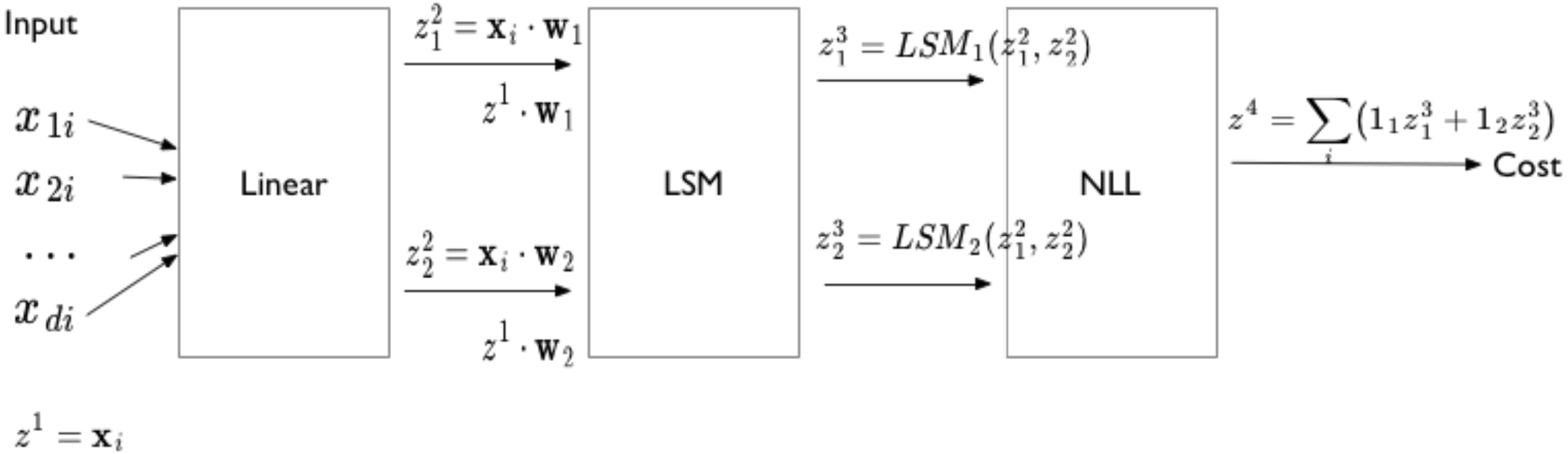
Last Time

- More Logistic Regression: arranging in layers
- Reverse Mode Differentiation
- A general way of solving SGD problems
- Neural Networks
- SGD and linear models: Universal Approximation

Today

- Backprop example
- Universal Approximation
- Learning Representations
- Latent Variables
- Mixture Models

MLE for Logistic Regression



Equations, layer by layer

$$\mathbf{z}^1 = \mathbf{x}_i$$

$$\mathbf{z}^2 = (z_1^2, z_2^2) = (\mathbf{w}_1 \cdot \mathbf{x}_i, \mathbf{w}_2 \cdot \mathbf{x}_i) = (\mathbf{w}_1 \cdot \mathbf{z}_i^1, \mathbf{w}_2 \cdot \mathbf{z}_i^1)$$

$$\mathbf{z}^3 = (z_1^3, z_2^3) = \big(LSM_1(z_1^2, z_2^2), LSM_2(z_1^2, z_2^2)\big)$$

$$z^4 = NLL(\mathbf{z}^3) = NLL(z_1^3, z_2^3) = -\sum_i \big(1_1(y_i) z_1^3(i) + 1_2(y_i) z_1^3(i)\big)$$

Reverse Mode Differentiation

$$Cost = f^{Loss}(\mathbf{f}^3(\mathbf{f}^2(\mathbf{f}^1(\mathbf{x}))))$$

$$\nabla_{\mathbf{x}} Cost = \frac{\partial f^{Loss}}{\partial \mathbf{f}^3} \frac{\partial \mathbf{f}^3}{\partial \mathbf{f}^2} \frac{\partial \mathbf{f}^2}{\partial \mathbf{f}^1} \frac{\partial \mathbf{f}^1}{\partial \mathbf{x}}$$

Write as:

$$\nabla_{\mathbf{x}} Cost = (((\frac{\partial f^{Loss}}{\partial \mathbf{f}^3} \frac{\partial \mathbf{f}^3}{\partial \mathbf{f}^2}) \frac{\partial \mathbf{f}^2}{\partial \mathbf{f}^1}) \frac{\partial \mathbf{f}^1}{\partial \mathbf{x}})$$

From Reverse Mode to Back Propagation

- Recursive Structure
- Always a vector times a Jacobian
- We add a "cost layer" to z^4. The derivative of this layer with respect to z^4 will always be 1.
- We then propagate this derivative back.

Backpropagation

RULE1: FORWARD (`.forward` in pytorch) $\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l)$

RULE2: BACKWARD (`.backward` in pytorch)

$\delta^l = \dfrac{\partial C}{\partial \mathbf{z}^l}$ or $\delta^l_u = \dfrac{\partial C}{\partial z^l_u}$.

$$\delta^l_u = \frac{\partial C}{\partial z^l_u} = \sum_v \frac{\partial C}{\partial z^{l+1}_v} \frac{\partial z^{l+1}_v}{\partial z^l_u} = \sum_v \delta^{l+1}_v \frac{\partial z^{l+1}_v}{\partial z^l_u}$$

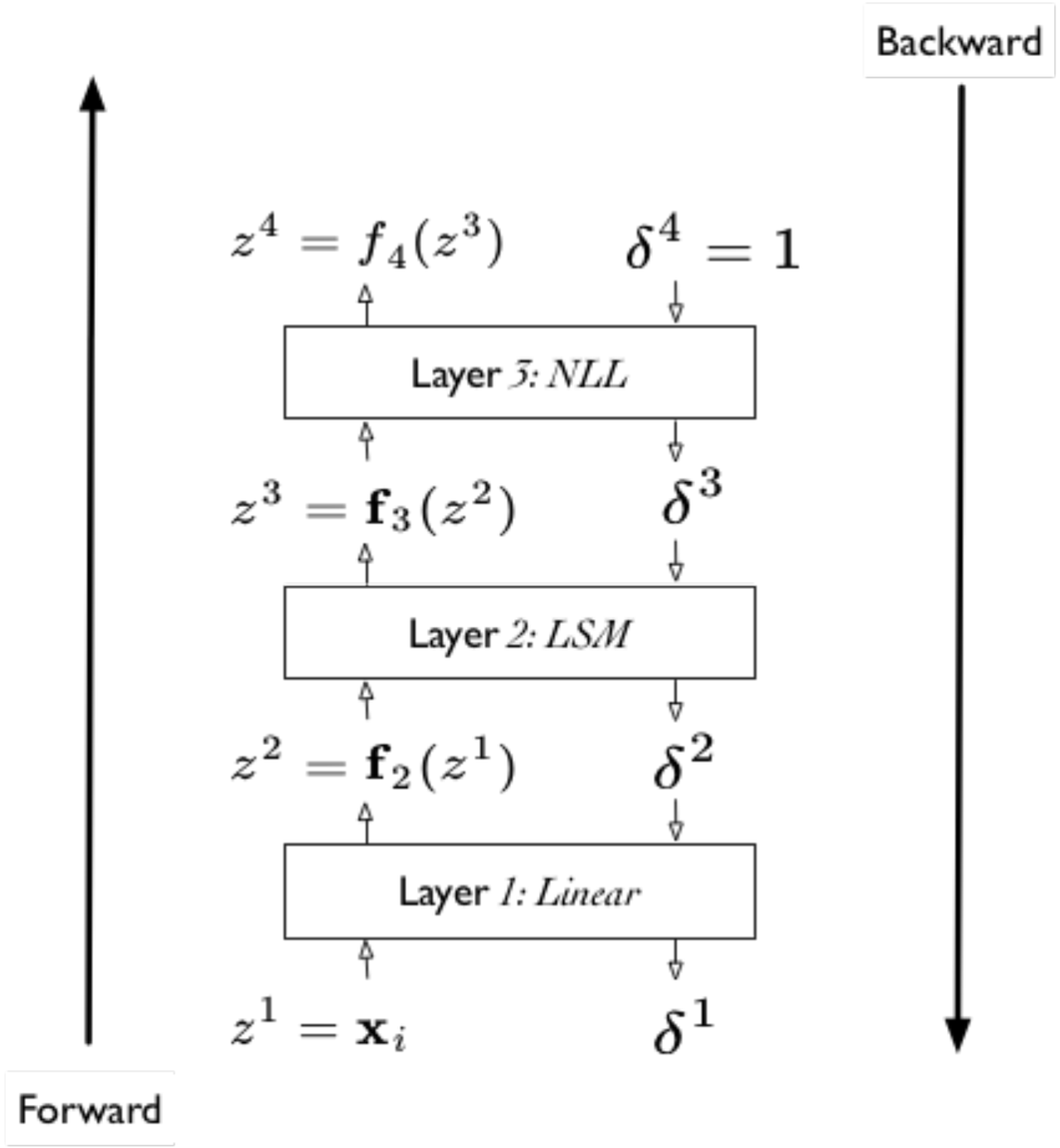
In particular:

$$\delta_u^3 = \frac{\partial z^4}{\partial z_u^3} = \frac{\partial C}{\partial z_u^3}$$

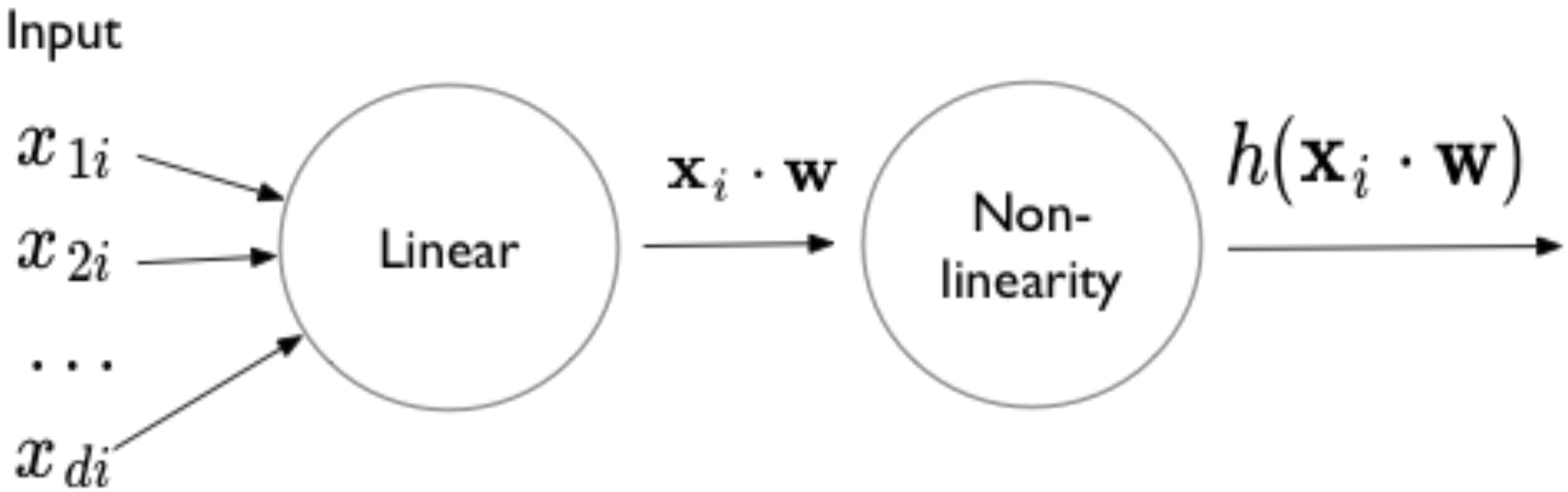
RULE 3: PARAMETERS

$$\frac{\partial C}{\partial \theta^l} = \sum_u \frac{\partial C}{\partial z_u^{l+1}} \frac{\partial z_u^{l+1}}{\partial \theta^l} = \sum_u \delta_u^{l+1} \frac{\partial z_u^{l+1}}{\partial \theta^l}$$

(backward pass is thus also used to fill the `variable.grad` parts of parameters in pytorch)


Backward
$z^4 = f_4(z^3)$
$\delta^4 = 1$
Layer 3: NLL
$z^3 = \mathbf{f}_3(z^2)$
δ^3
Layer 2: LSM
$z^2 = \mathbf{f}_2(z^1)$
δ^2
Layer 1: Linear
$z^1 = \mathbf{x}_i$
δ^1
Forward

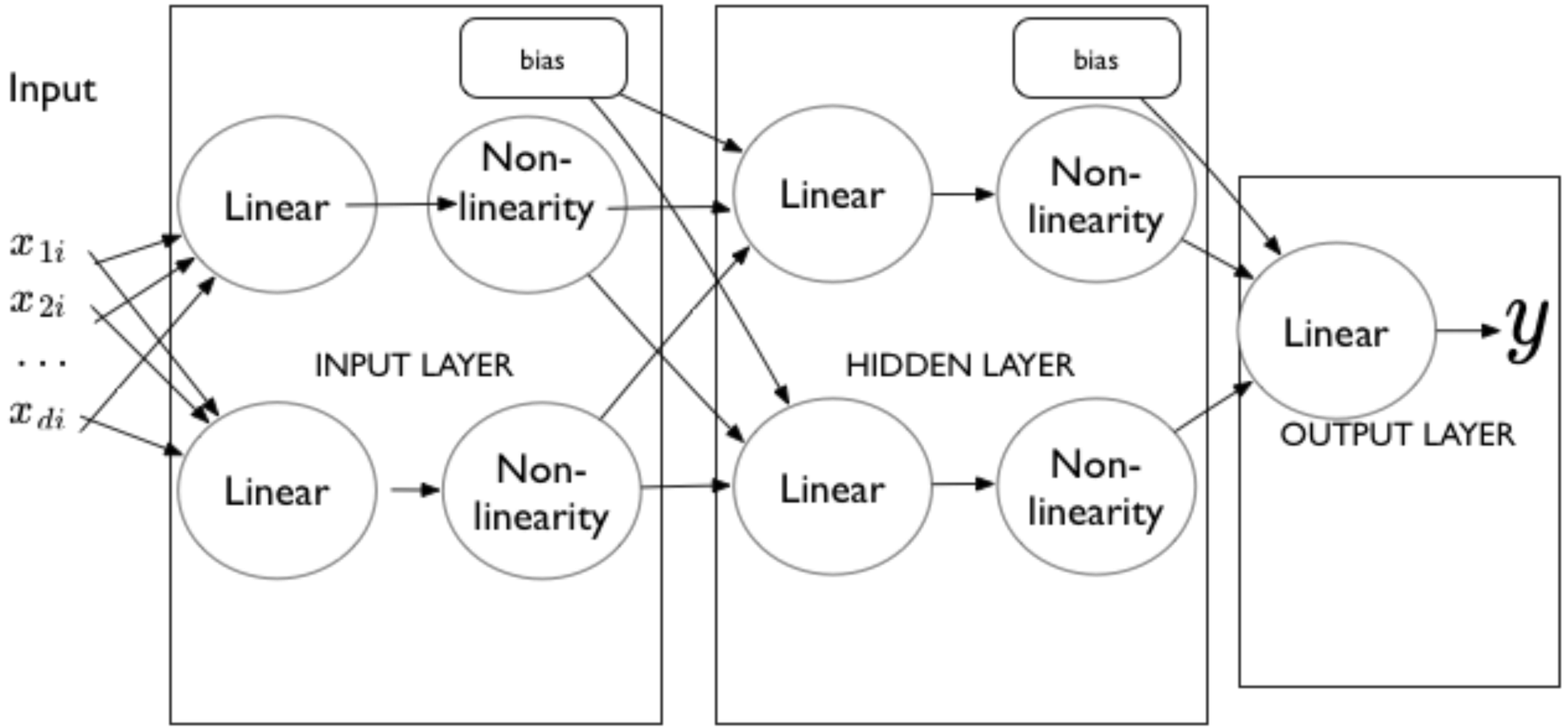
Feed Forward Neural Nets: The perceptron

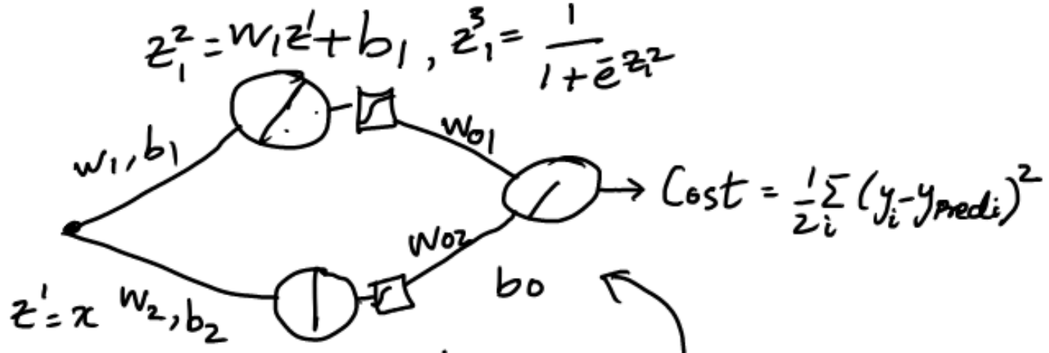


Just combine perceptrons

- both deep and wide
- this buys us complex nonlinearity
- both for regression and classification
- key technical advance: BackPropagation with
- autodiff
- key technical advance: gpu

Combine Perceptrons





$$z^5 = \frac{1}{2}(z^4 - y)^2 \qquad z^6 = \text{Cost} = \sum_i z^5(i)$$

$$\frac{\partial z^6}{\partial z^5} = 1, \quad \frac{\partial z^5}{\partial z^4} = z^4 - y, \quad \frac{\partial z^4}{\partial z_1^3} = w_{01}$$

$$\frac{\partial z^4}{\partial z_2^3} = w_{02}, \quad \frac{\partial z^4}{\partial w_{01}} = z_1^3, \quad \frac{\partial z^4}{\partial w_{02}} = z_2^3$$

$$\frac{\partial z^4}{\partial b_0} = 1, \quad \frac{\partial z_1^3}{\partial z_1^2} = z_1^3(1 - z_1^3),$$

$$\frac{\partial z_2^3}{\partial z_2^2} = z_2^3(1 - z_2^3), \quad \frac{\partial z_1^2}{\partial z^1} = w_1,$$

$$\frac{\partial z_2^2}{\partial z^1} = w_2, \quad \frac{\partial z_1^2}{\partial w_1} = z^1, \quad \frac{\partial z_2^2}{\partial w_2} = z^1,$$

$$\frac{\partial z_1^2}{\partial b_1} = 1, \quad \frac{\partial z_2^2}{\partial b_2} = 1$$

Forward Pass

We want to obtain gradients. For example: $\frac{\partial Cost}{\partial param} = \frac{\partial z^6}{\partial w_1}$

First we do the **Forward Pass**. Say we have 1 sample: (x=0.1, y=5). Initialize $b_1, w_1, b_2, w_2, w_{o1}, w_{o2}, b_o$. Then, plugging in the numbers will give us some Cost (z^5, z^6).

$z^5 = \frac{1}{2}(z^4 - y)^2$ $z^4 = w_{o1} z_1^3 + w_{o2} z_2^3 + b_o$ $z_1^3 = 1/(1+e^{-z_1^2})$, $z_2^3 = 1/(1+e^{-z_2^2})$ $z_1^2 = w_1 z^1 + b_1$, $z_2^2 = w_2 z^1 + b_2$, $z_1 = x = 0.1$

Backward Pass

Now it is time to find the gradients, for eg, $\frac{\partial z^6}{\partial w_1}$

The basic idea is to gather all parts that go to w_1, and so on and so forth. Now we perform GD (SGD) with some learning rate.

The parameters get updated. Now we repeat the forward pass.

Thats it! Wait for convergence.

$$\frac{\partial z^5}{\partial w^1} = \frac{\partial z^5}{\partial z^4}\frac{\partial z^4}{\partial z_1^3}\frac{\partial z_1^3}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_1}$$

$$= (z^4 - y) \times w_{01} \times z_1^3(1 - z_1^3)\, z^1$$

Backpropagation

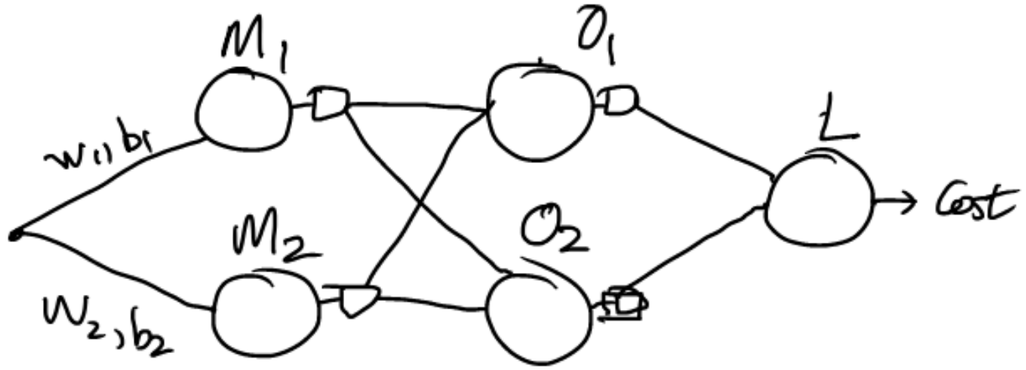
FORWARD: $\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l)$

BACKWARD: $\delta_u^l = \dfrac{\partial C}{\partial z_u^l} = \displaystyle\sum_v \frac{\partial C}{\partial z_v^{l+1}} \frac{\partial z_v^{l+1}}{\partial z_u^l} = \sum_v \delta_v^{l+1} \frac{\partial z_v^{l+1}}{\partial z_u^l}$

$$\frac{\partial C}{\partial \theta^l} = \sum_u \frac{\partial C}{\partial z_u^{l+1}} \frac{\partial z_u^{l+1}}{\partial \theta^l} = \sum_u \delta_u^{l+1} \frac{\partial z_u^{l+1}}{\partial \theta^l}$$

Another model

what if we had ore more layer?



Now $\frac{\partial Cost}{\partial w_1}$ follows 2 paths.

$$\frac{\partial Cost}{\partial w_1} = \frac{\partial Cost}{\partial L} \frac{\partial L}{\partial M_1} \frac{\partial M_1}{\partial w_1}$$

$$\frac{\partial L}{\partial M_1} = \frac{\partial L}{\partial O_1} \frac{\partial O_1}{\partial M_1} + \frac{\partial L}{\partial O_2} \frac{\partial O_2}{\partial M_2}$$

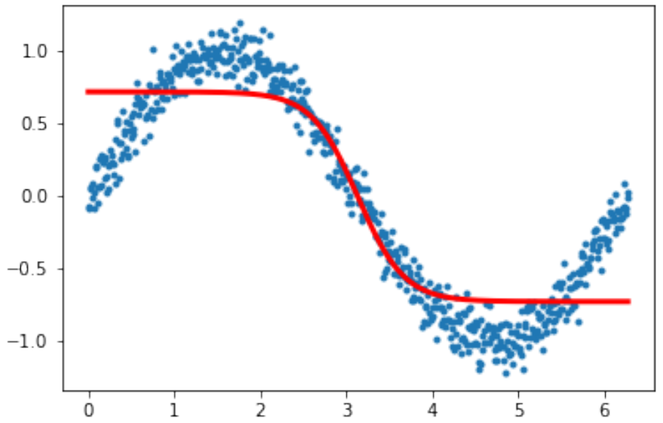
Basic code outline

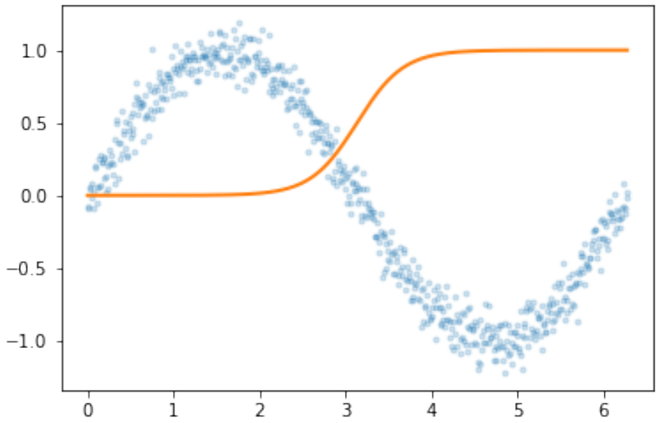
```
dataset = torch.utils.data.TensorDataset(torch.from_numpy(xgrid.reshape(-1,1)), torch.from_numpy(ygrid))
loader = torch.utils.data.DataLoader(dataset, batch_size=64,shuffle=True)
def run_model(model, epochs):
    criterion = nn.MSELoss()
    lr, epochs, batch_size = 1e-1 , epochs , 64
    optimizer = torch.optim.SGD(model.parameters(), lr = lr )
    accum=[]
    for k in range(epochs):
        localaccum = []
        for localx, localy in iter(loader):
            localx = Variable(localx.float())
            localy = Variable(localy.float())
            output, _, _ = model.forward(localx)
            loss = criterion(output, localy)
            model.zero_grad()
            loss.backward()
            optimizer.step()
            localaccum.append(loss.data[0])
        accum.append((np.mean(localaccum), np.std(localaccum)))
    return accum
```

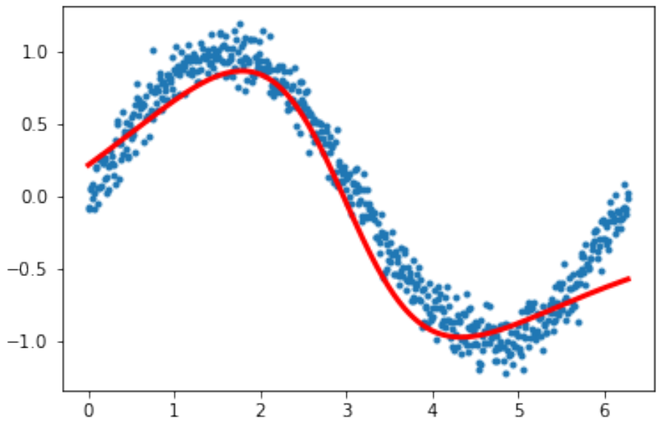
Universal Approximation

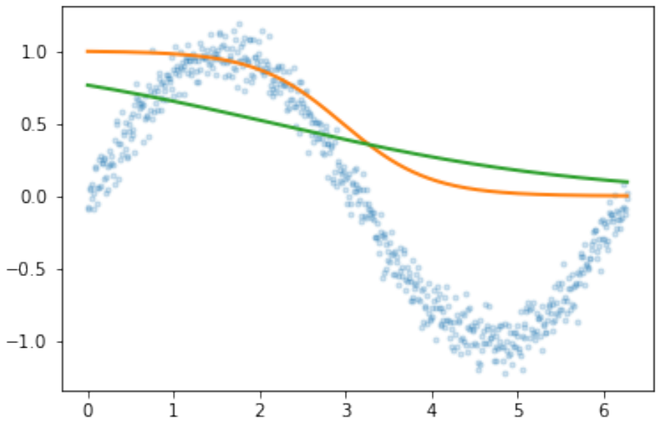
- any one hidden layer net can approximate any continuous function with finite support, with appropriate choice of nonlinearity
- under appropriate conditions, all of sigmoid, tanh, RELU can work
- but may need lots of units
- and will learn the function it thinks the data has, not what you think

One hidden, 1 vs 2 neurons

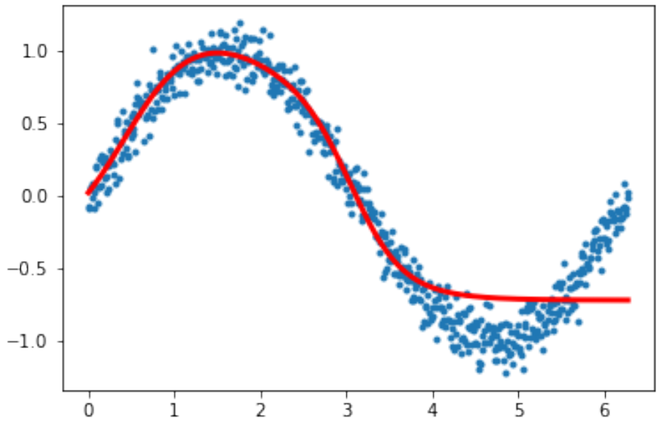


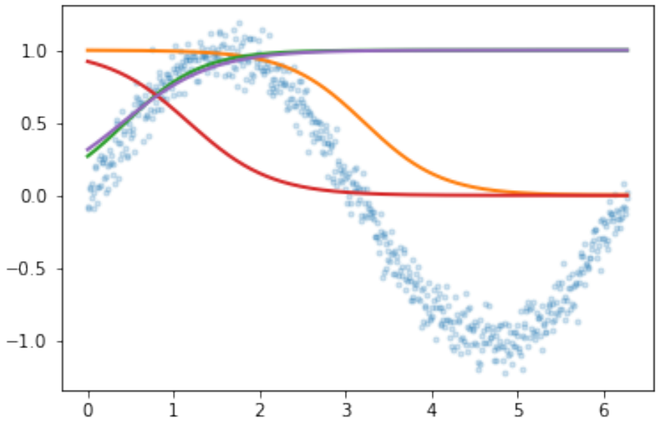


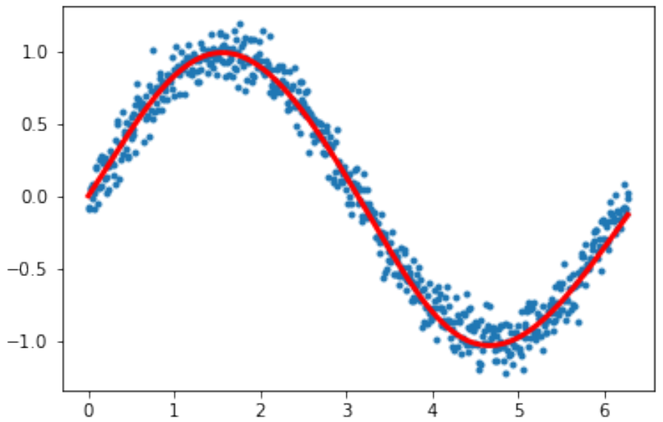


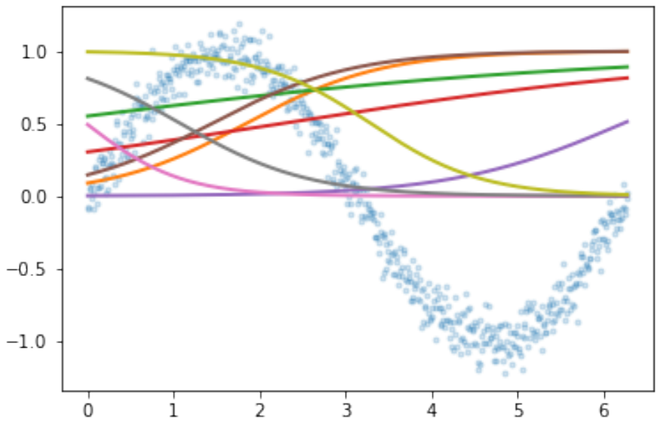


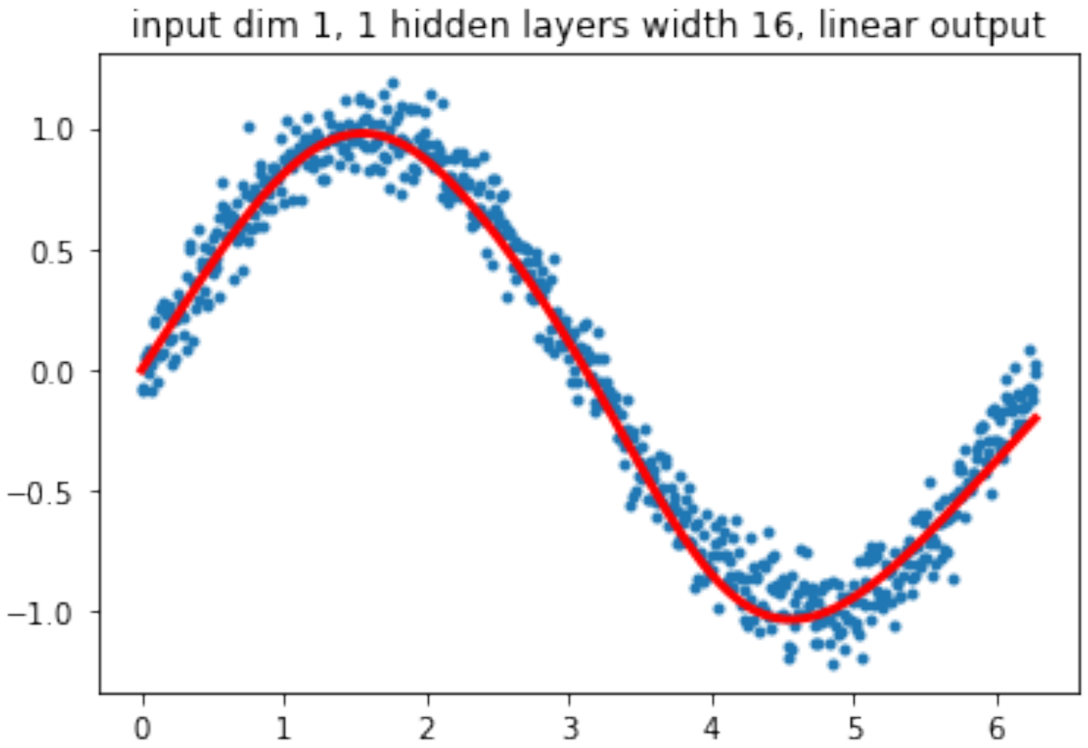
Two hidden, 4 vs 8 neurons

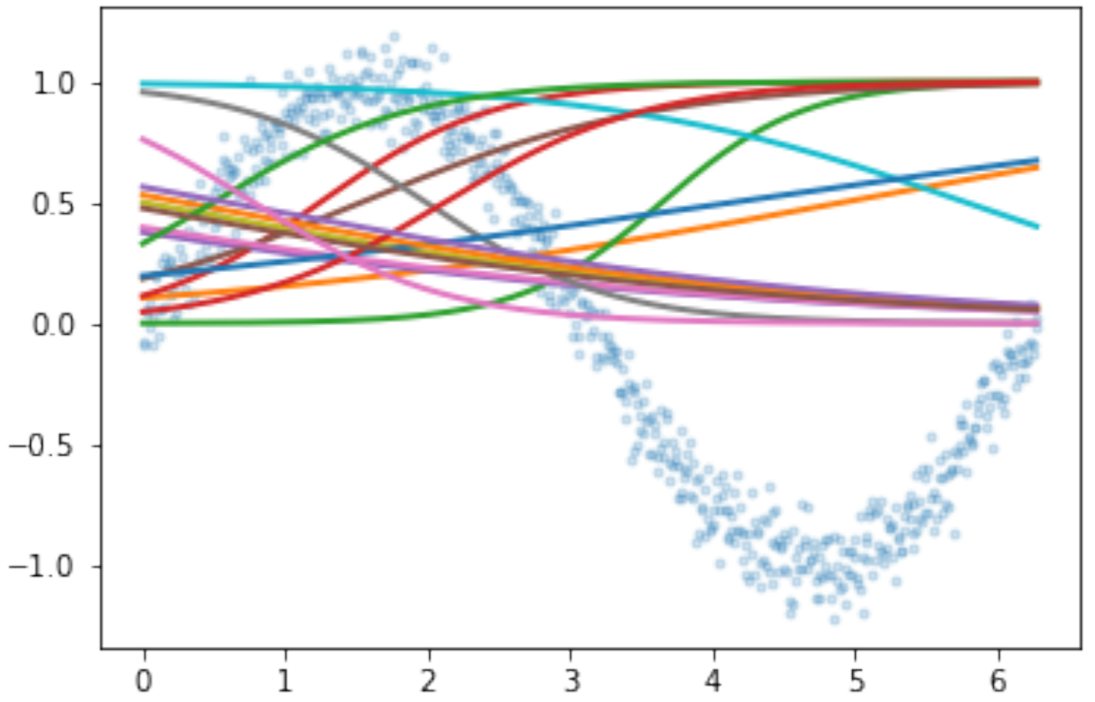




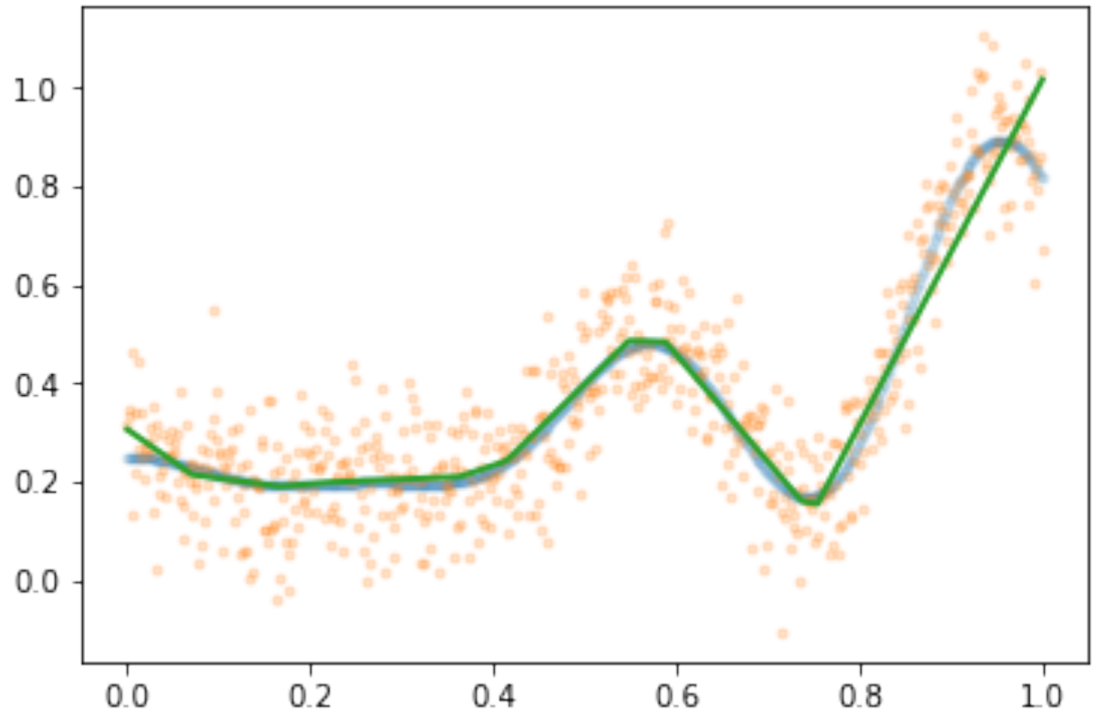


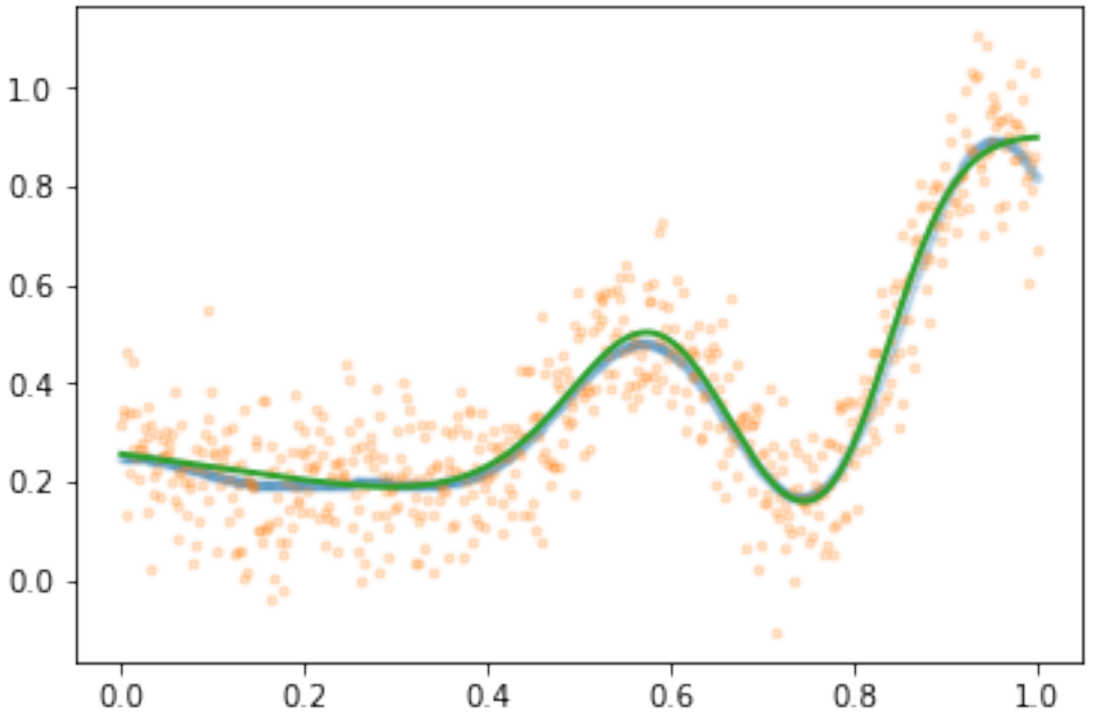



input dim 1, 1 hidden layers width 16, linear output
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6


1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Relu (80, 1 layer) and tanh(40, 2 layer)

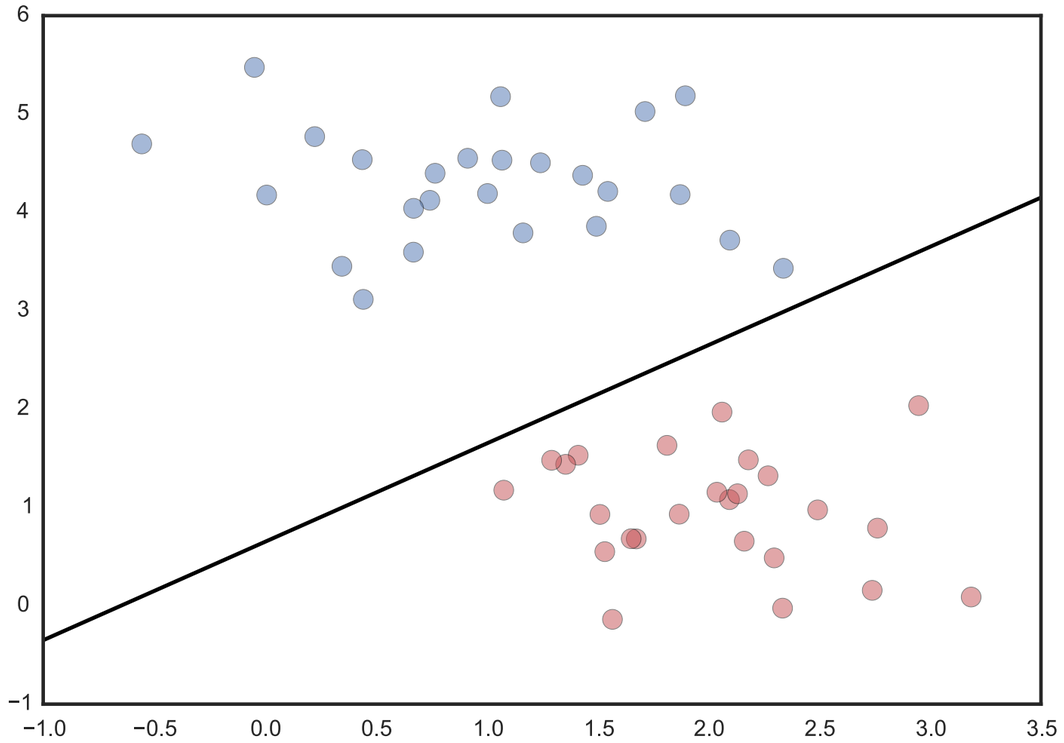




Some rules of thumb

- relu and tanh are better non-linearities in hidden layers
- normalize your data by squashing to unit interval or standardizing so that no feature gets more important than the other
- outputs from non-linearity at any intermediate layer may need normalizing

CLASSIFICATION



- will a customer churn?
- is this a check? For how much?
- a man or a woman?
- will this customer buy?
- do you have cancer?
- is this spam?
- whose picture is this?
- what is this text about?[j]

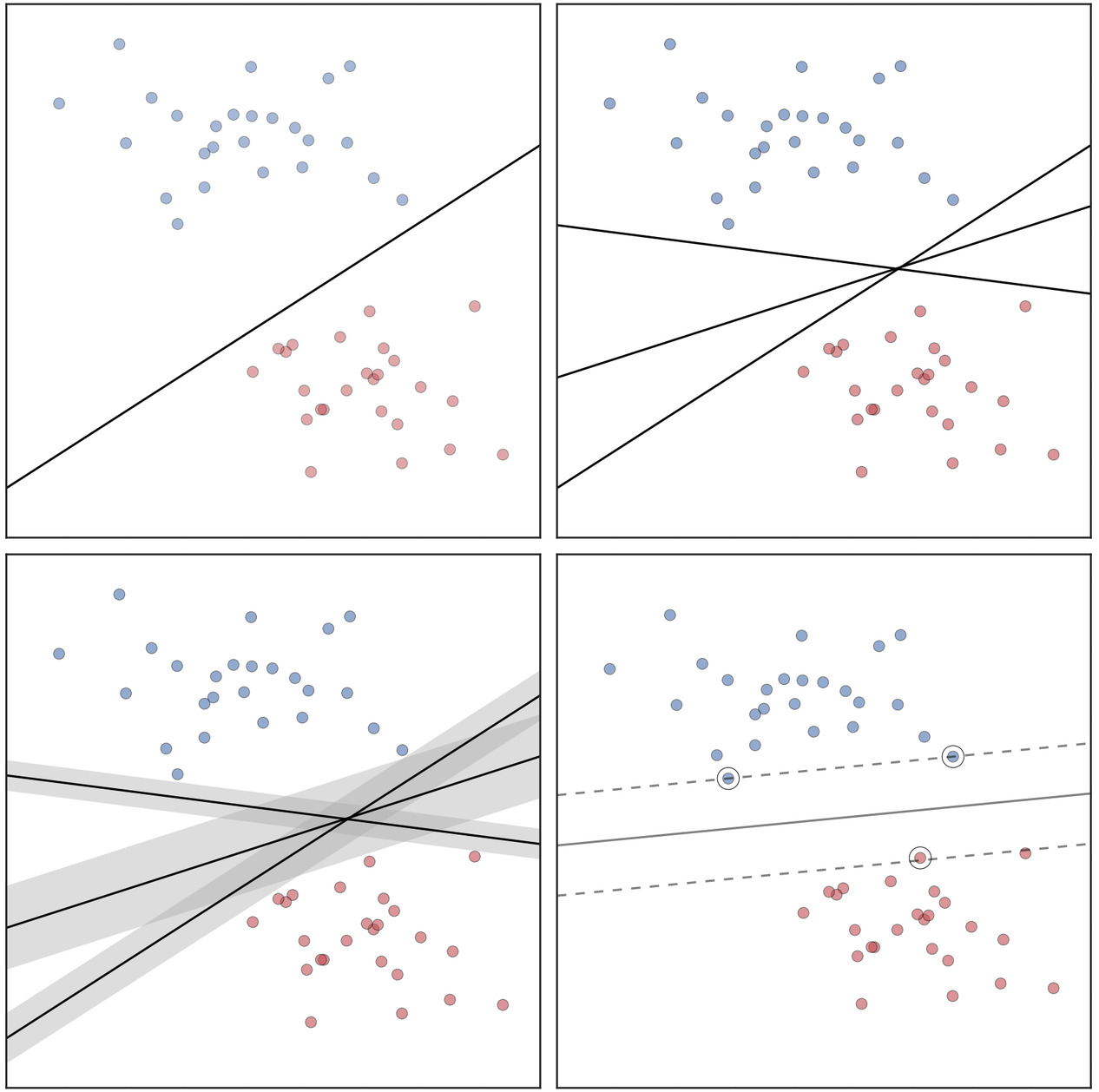
[j] image from code in http://bit.ly/1Azg29G

CLASSIFICATION

BY LINEAR SEPARATION

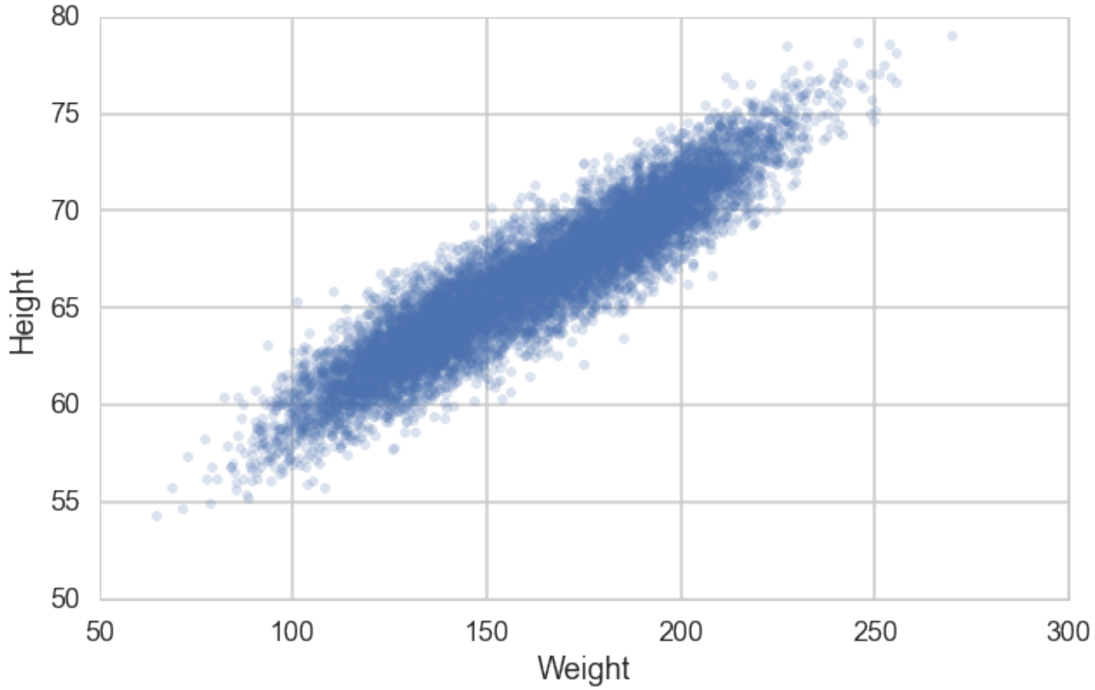
Which line?

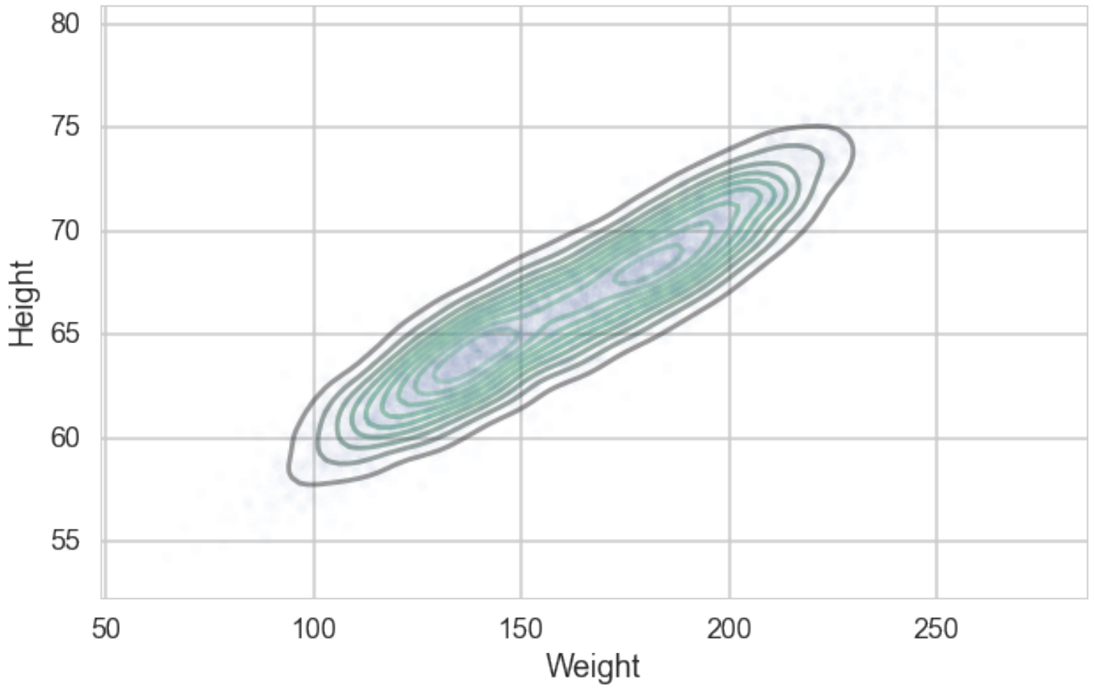
- Different Algorithms, different lines.
- SVM uses max-margin[j]



[j] image from code in http://bit.ly/1Azg29G

PROBABILISTIC CLASSIFICATION



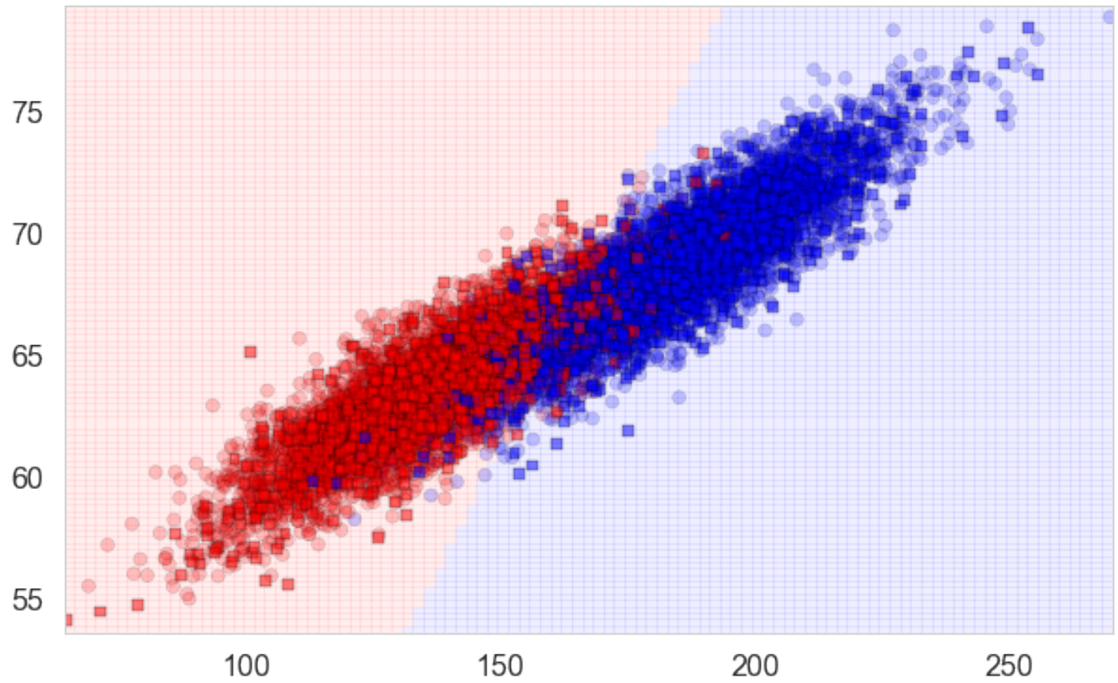


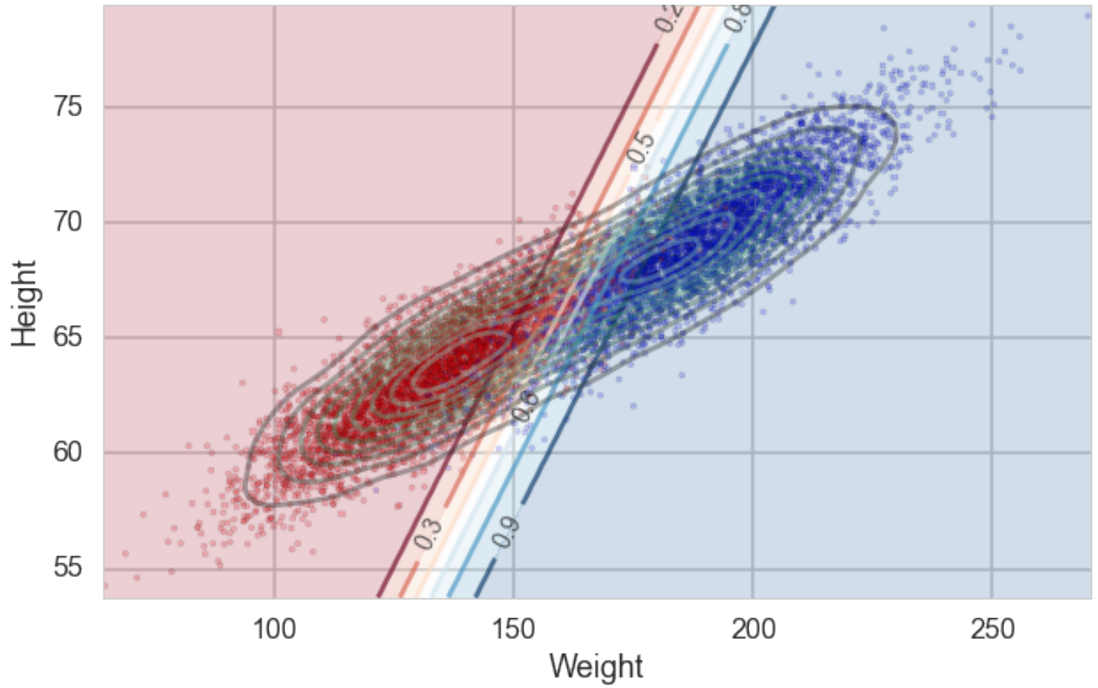
Model $P(y|x)$ or $P(x|y)$.

DISCRIMINATIVE CLASSIFIER

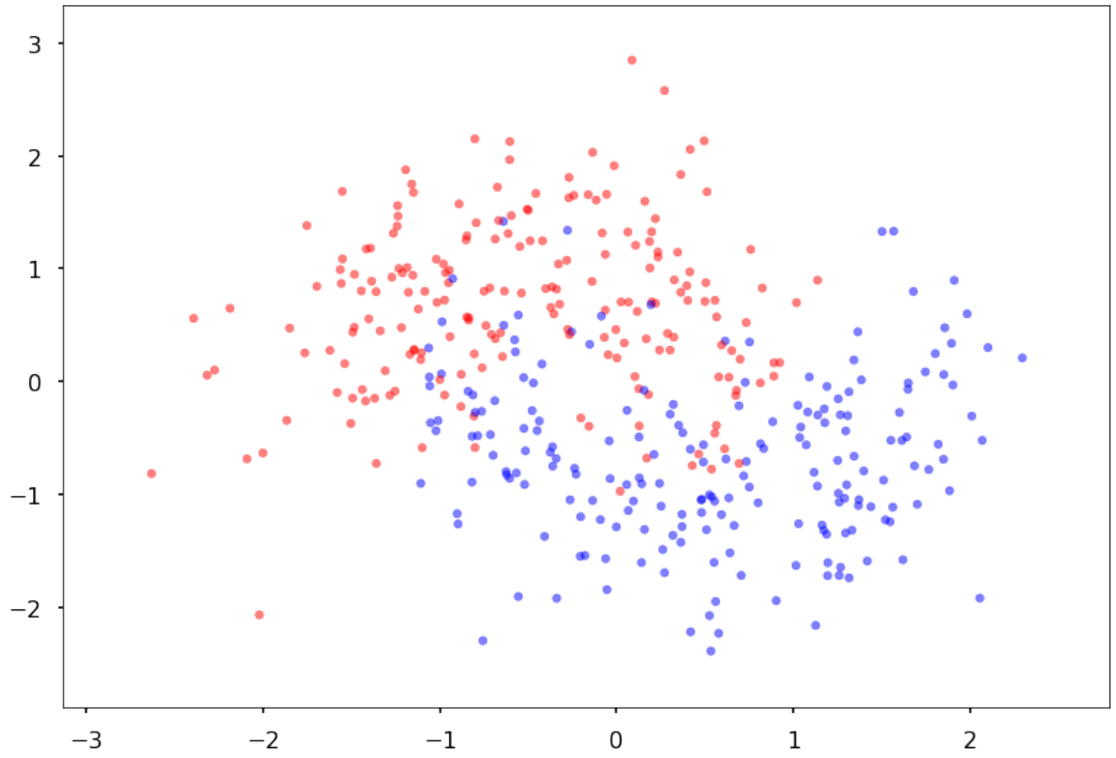
$$P(y|x) : P(male|height, weight)$$

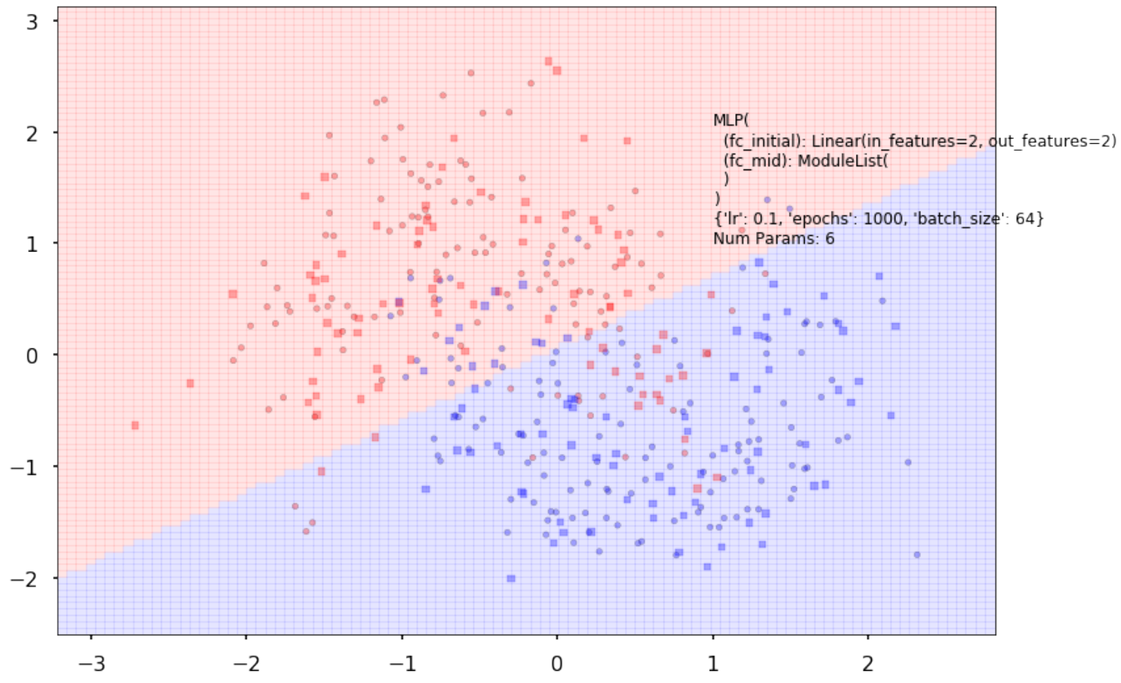
Learn the 2 discriminative boundaries



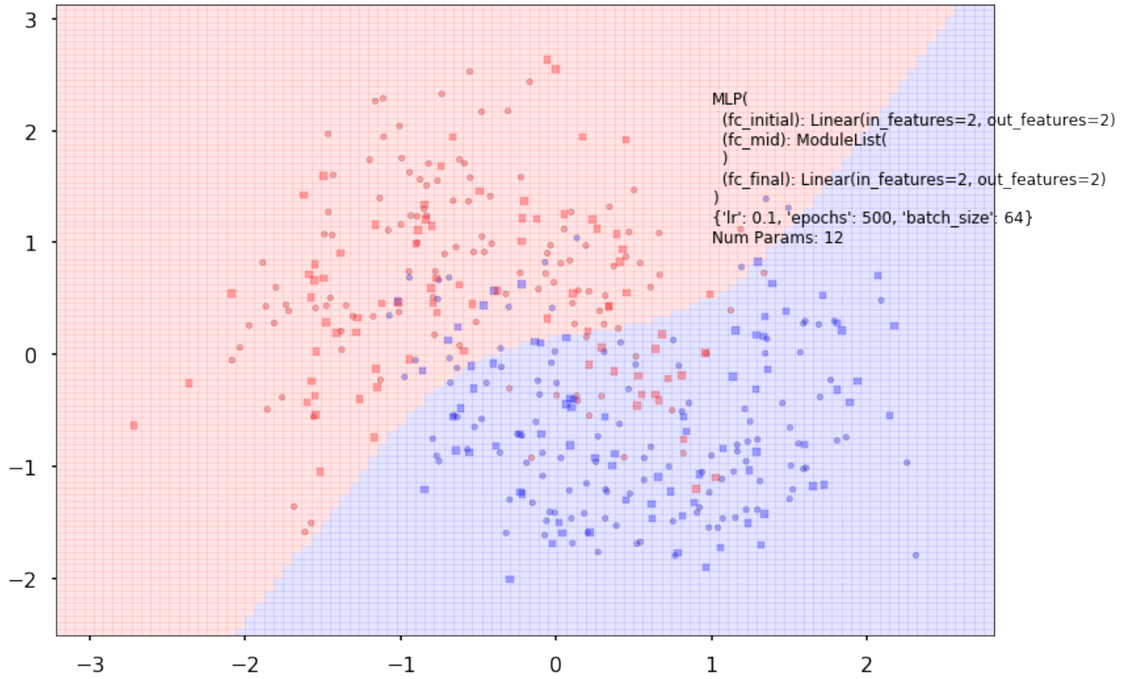


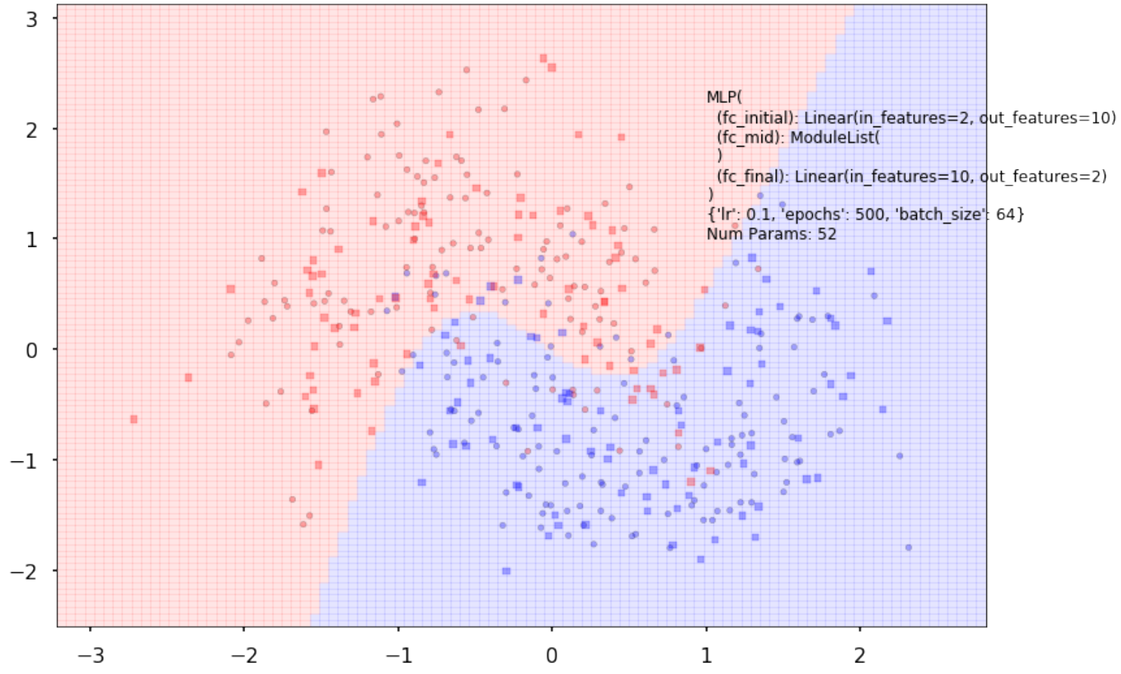
Half moon dataset (artificially GENERATED)



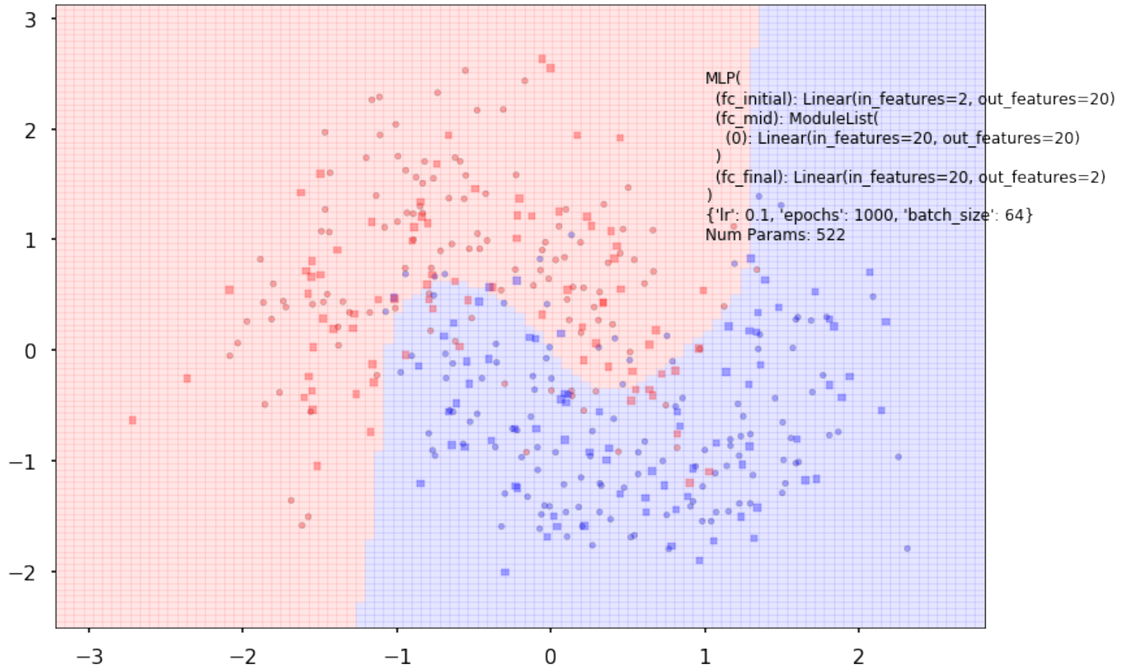


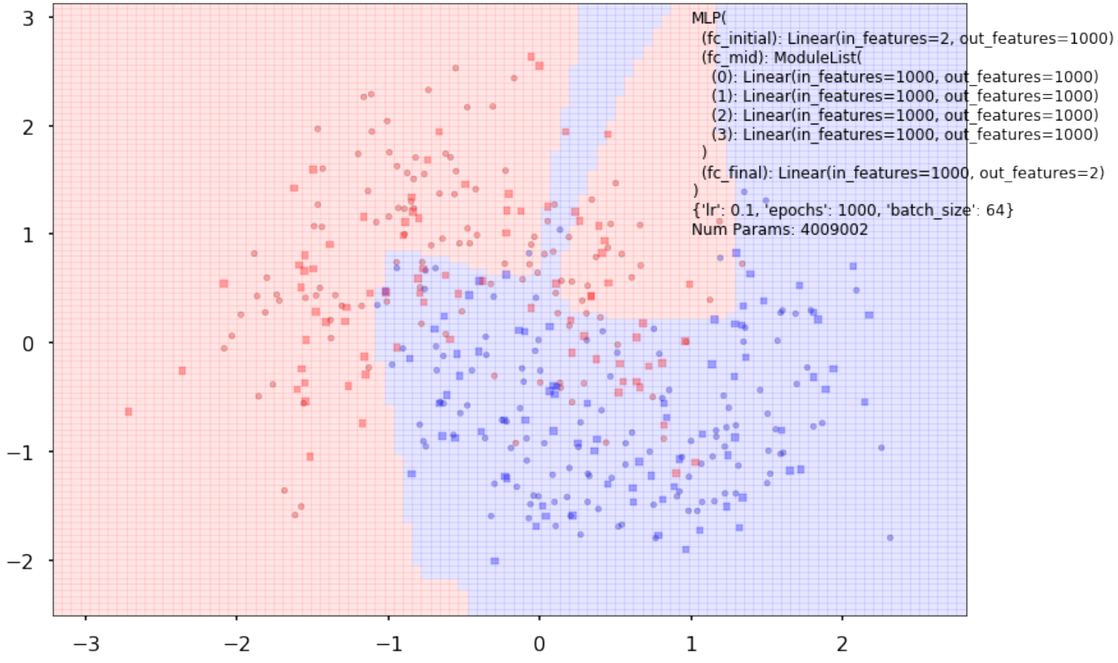
1 layer, 2 vs 10 neurons





2 layers, 20 neurons vs 5 layers, 1000 neurons



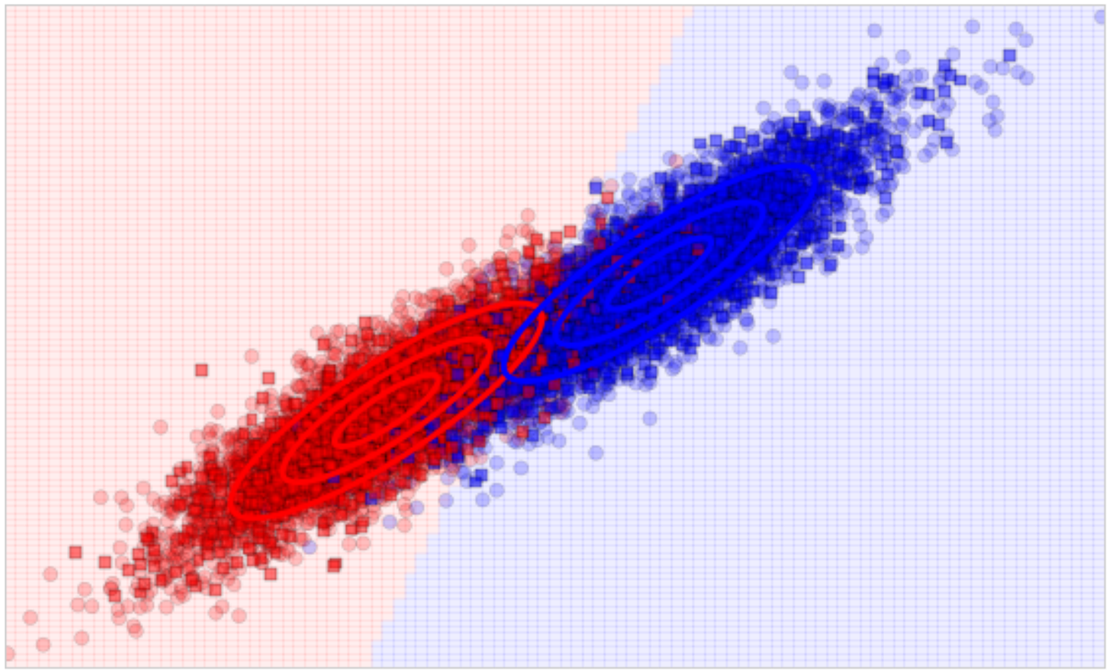


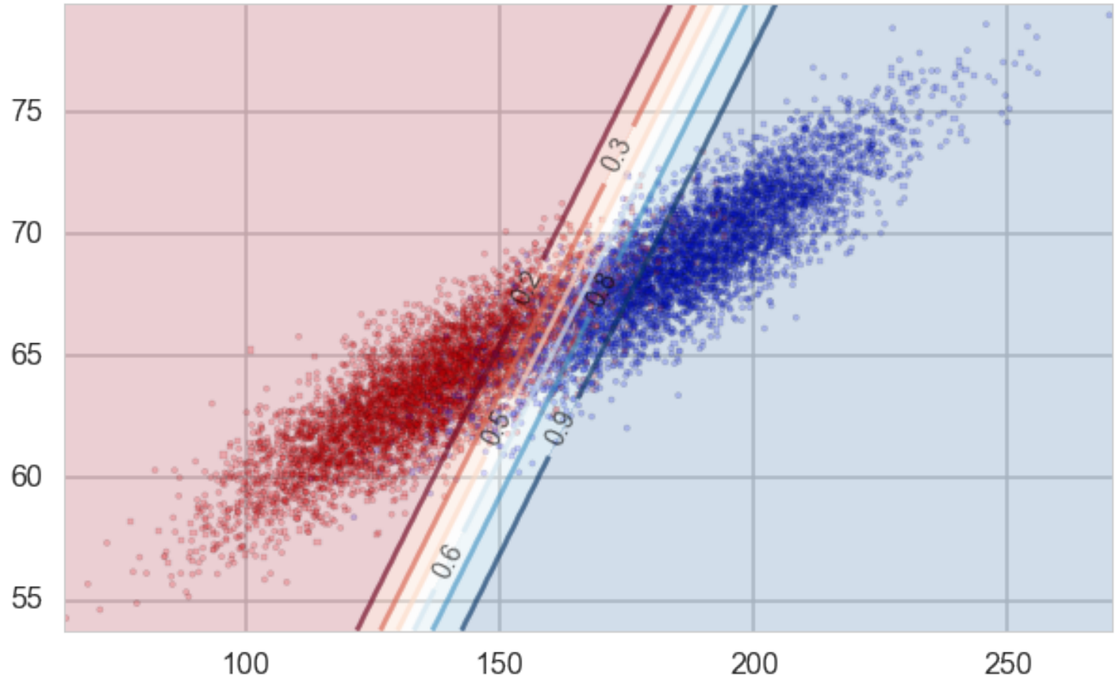
Discriminative Learning

- are these classifiers any good?
- they are discriminative and draw boundaries, but thats it
- they are cheaper to calculate but shed no insight
- would it not be better to have a classifier that captured the generative process

GENERATIVE CLASSIFIER

$$P(y|x) \propto P(x|y)P(x) : P(height, weight|male) \times P(male)$$





Representation Learning

- the idea of generative learning is to capture an underlying representation (compressed) of the data
- in the previous slide it was 2 normal distributions
- generally more complex, but the idea if to fit a "generative" model whose parameters represent the process
- wait, we've been doing this in our bayesian or conditional-on-data-marginalize-over-all-else paradigm
- besides gpus and autodiff on backprop, this is the third pillar of the AI rennaissance: the choice of better representations: e.g. convolutions

Ok, so how do we model (simple) representations. We've been doing it already.

Latent Variables

we marginalize over...

- instead of bayesian vs frequentist, think hidden vs not hidden
- key concept: full data likelihood $p(\mathbf{x}, \mathbf{z})$ vs partial data likelihood $p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$
- For regression/classification "$\mathbf{z} = y$ or c", full is supervised with partial being unsupervised
- observed variables $\mathbf{x}$ correspond to data, and latent variables $\mathbf{z}$ to classes/parameters

From `edwardlib` docs: $p(\mathbf{x} \mid \mathbf{z})$

describes how any data $\mathbf{x}$ depend on the latent variables $\mathbf{z}$.

- **The likelihood posits a data generating process**, where the data $\mathbf{x}$ are assumed drawn from the likelihood conditioned on a particular hidden pattern described by $\mathbf{z}$.
- The *prior* $p(\mathbf{z})$ is a probability distribution that describes the latent variables present in the data. **The prior posits a generating process of the hidden structure**.

Mixture Models motivation

- $\mathbf{z}$ as "classes" in a classification problem leads to a generative classifier
- but in general, that identification is very strong, indeed $\mathbf{z}$ may just be a representation

Mixture Models

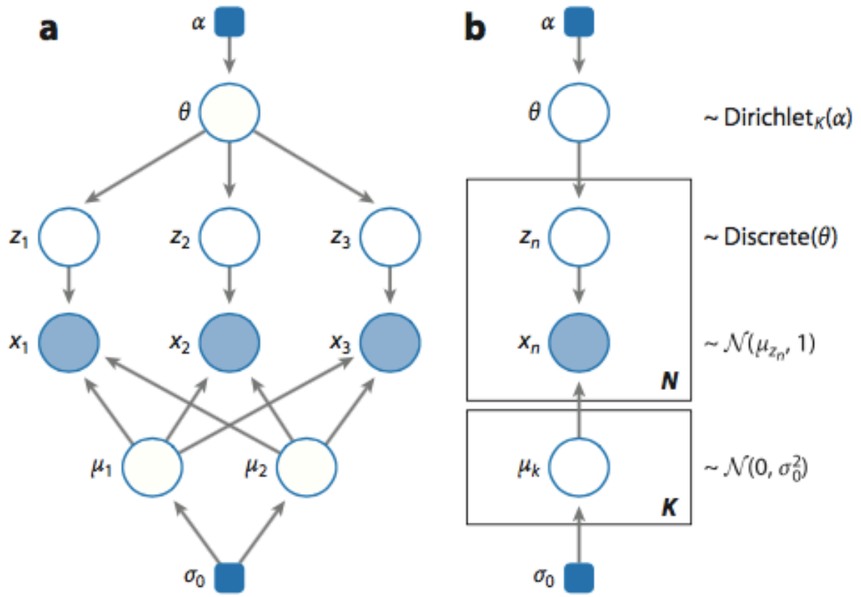


Figure 3

(*a*) A graphical model for a mixture of two Gaussians. There are three data points. The shaded nodes are observed variables, the unshaded nodes are hidden variables, and the blue square boxes are fixed hyperparameters (such as the Dirichlet parameters). (*b*) A graphical model for a mixture of K Gaussians with N data points.

A distribution $p(x|\{\theta_k\})$ is a mixture of K component distributions $p_1, p_2, \ldots p_K$ if:

$$p(x|\{\theta_k\}) = \sum_k \lambda_k p_k(x|\theta_k)$$

with the λ_k being mixing weights, $\lambda_k > 0$, $\sum_k \lambda_k = 1$.

Generative Model: How to simulate from it?

$$Z \sim Categorical(\lambda_1, \lambda_2, \ldots, \lambda_K)$$

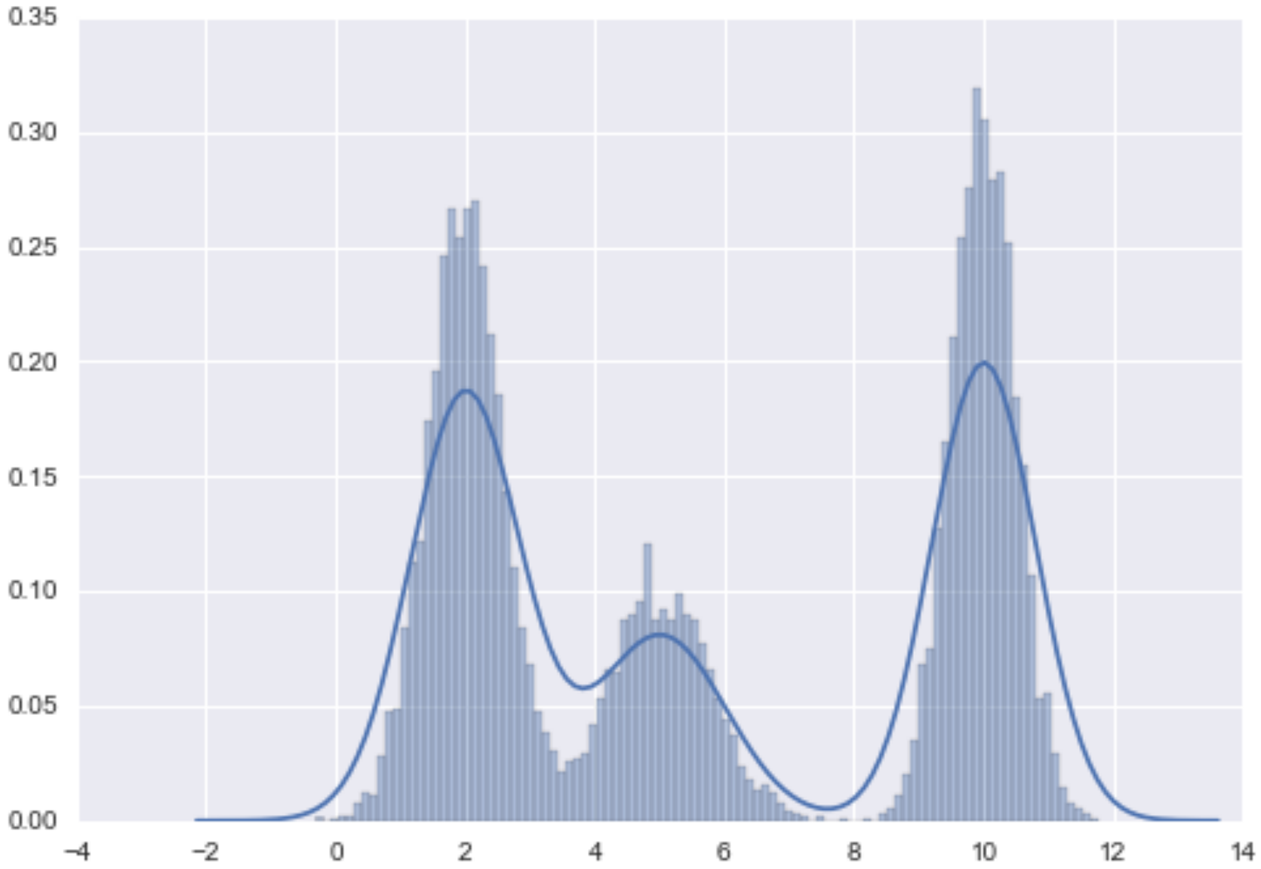
where Z says which component X is drawn from.

Thus λ_j is the probability that the hidden class variable $z = j$.

Then: $X \sim p_z(x|\theta_z)$ and general structure is:

$$p(x|\{\theta_z\}) = \sum_z p(x,z) = \sum_z p(z)p(x|z,\theta_z)\ .$$

Gaussian Mixture Model



$$p(x|\{\theta_k\}) = \sum_k \lambda_k N(x|\mu_k, \Sigma_k)$$

Generative:

```
mu_true = np.array([2, 5, 10])
sigma_true = np.array([0.6, 0.8, 0.5])
lambda_true = np.array([.4, .2, .4])
n = 10000

# Simulate from each distribution according to mixing proportion psi
z = multinomial.rvs(1, lambda_true, size=n) #categorical
x=np.array([np.random.normal(mu_true[i.astype('bool')][0],\
    sigma_true[i.astype('bool')][0]) for i in z])
```

```
multinomial.rvs(1,[0.6,0.1, 0.3], size=10)
array([[1, 0, 0],[0, 0, 1],...[1, 0, 0],[1, 0, 0]])
```

Generative Classifier

For a feature vector x, we use Bayes rule to express the posterior of the class-conditional as:

$$p(z = c|x, \theta) = \frac{p(z = c|\theta)p(x|z = c, \theta)}{\sum_{c'} p(z = c'|\theta)p(x|z = c', \theta)}$$

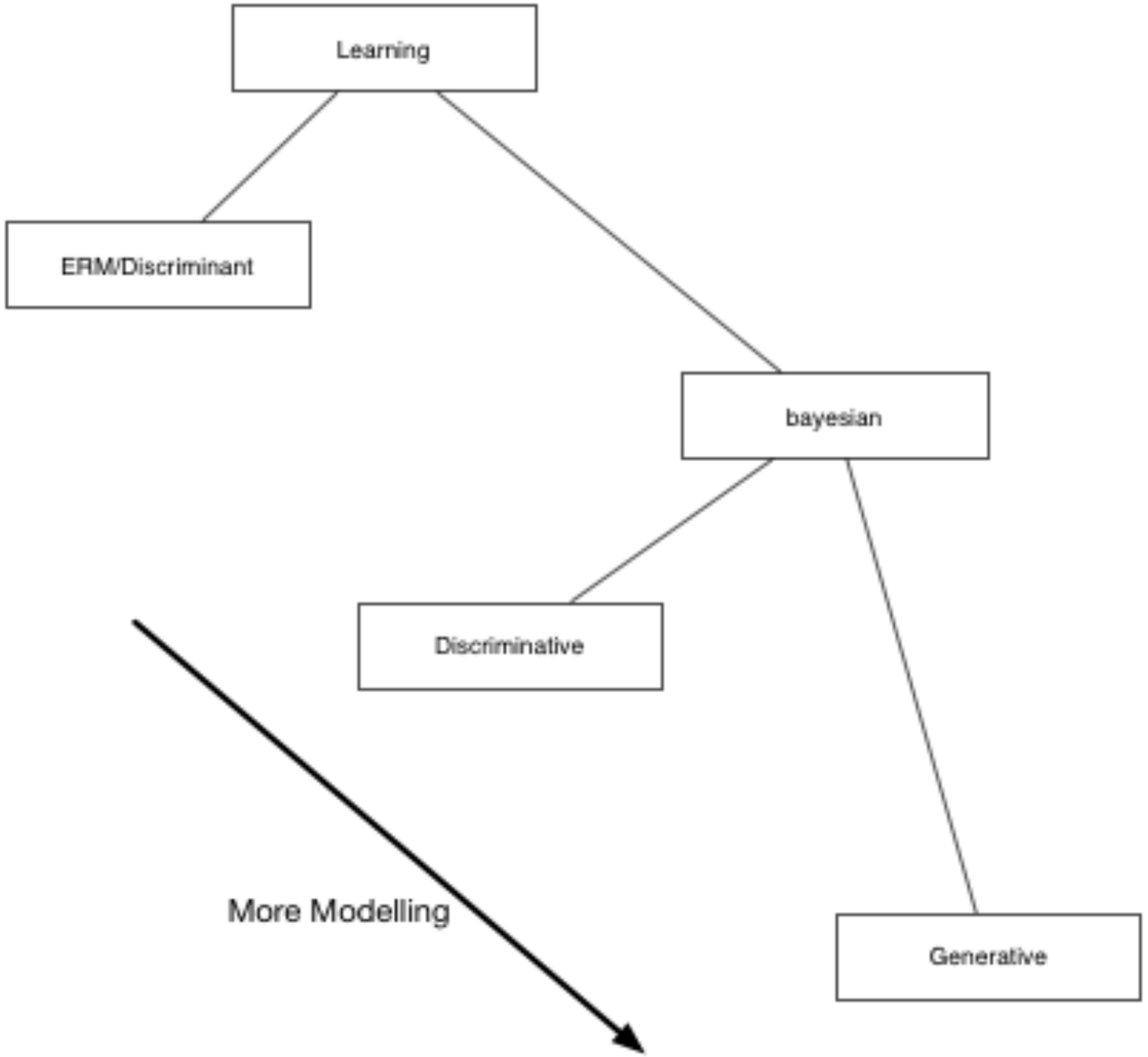
This is a **generative classifier**, since it specifies how to generate the data using the class-conditional density $p(x|z = c, \theta)$ and the class prior $p(z = c|\theta)$.

Generative vs Discriminative classifiers

- LDA vs logistic respectively.
- Both have "generative" bayesian models: $p(c|x, \theta)$ or $p(y|x, \theta)$. Here think of $\mathbf{z} = \theta$
- LDA is generative as it models $p(x|c)$ while logistic models $p(c|x)$ directly. Here think of $\mathbf{z} = c$
- we do know c on the training set, so think of the unsupervised learning counterparts of these models where you dont know c

Generative vs Discriminative classifiers (contd)

- generative handles data asymmetry better
- sometimes generative models like LDA and Naive Bayes are easy to fit. Discriminative models require convex optimization via Gradient descent
- can add new classes to a generative classifier without retraining so better for online customer selection problems
- generative classifiers can handle missing data easily
- generative classifiers are better at handling unlabelled training data (semi-supervized learning)
- preprocessing data is easier with discriminative classifiers
- discriminative classifiers give generally better callibrated probabilities
- discriminative usually less expensive


Learning
ERM/Discriminant
bayesian
Discriminative
Generative
More Modelling

The two meanings of generative

Thus we **abuse** the world **generative** in two senses:

1. A way to generate data drom a data story. Here think of $\mathbf{z} = \theta$

2. A Model in which we try to figure $p(\mathbf{x}, \mathbf{z})$ or $p(\mathbf{x}|\mathbf{z})$. Here think of $\mathbf{z} = c$ or a class label.

Now lets focus on the latter. Suppose we believe their exists a "class" or representation $\mathbf{z}$. Then a dichotomy arises depending on whether $\mathbf{z}$ is observed or not.

Supervised vs Unsupervised Learning

In **Supervised Learning**, Latent Variables $\mathbf{z}$ are observed.

In other words, we can write the full-data likelihood $p(\mathbf{x}, \mathbf{z})$

In **Unsupervised Learning**, Latent Variables $\mathbf{z}$ are hidden.

We can only write the observed data likelihood:

$$p(\mathbf{x}) = \sum_z p(\mathbf{x}, \mathbf{z}) = \sum_z p(\mathbf{z}) p(\mathbf{x}|\mathbf{z})$$

GMM supervised formulation

$$Z \sim \text{Bernoulli}(\lambda)$$
$$X|z=0 \sim \mathcal{N}(\mu_0, \Sigma_0),\ X|z=1 \sim \mathcal{N}(\mu_1, \Sigma_1)$$

Full-data loglike: $l(x, z|\lambda, \mu_0, \mu_1, \Sigma) = -\sum_{i=1}^{m} \log((2\pi)^{n/2}|\Sigma|^{1/2})$

$$-\frac{1}{2}\sum_{i=1}^{m}(x-\mu_{z_i})^T \Sigma^{-1}(x-\mu_{z_i}) + \sum_{i=1}^{m}[z_i \log\lambda + (1-z_i)\log(1-\lambda)]$$

Solution to MLE

$$\lambda = \frac{1}{m}\sum_{i=1}^{m} \delta_{z_i,1}$$

$$\mu_0 = \frac{\sum_{i=1}^{m} \delta_{z_i,0}\, x_i}{\sum_{i=1}^{m} \delta_{z_i,0}}$$

$$\mu_1 = \frac{\sum_{i=1}^{m} \delta_{z_i,1}\, x_i}{\sum_{i=1}^{m} \delta_{z_i,1}}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{z_i})(x_i - \mu_{z_i})^T$$

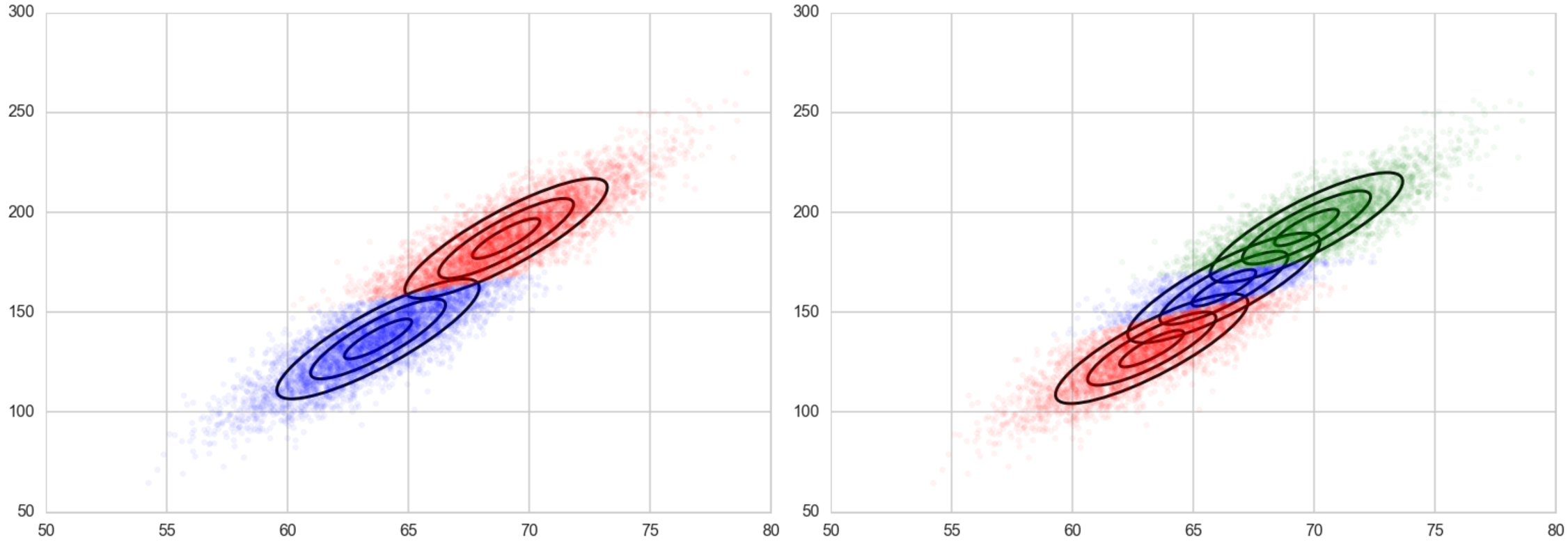
Classification

We can use the log likelihood at a given x as a classifier: assign class depending upon which probability $p(x_j|\lambda, z, \Sigma)$ is larger. (JUST x likelihood, as we want to compare probabilities at fixed zs).

$$log\, p(x_j|\lambda, z, \Sigma) = -\sum_{i=1}^{m} \log((2\pi)^{n/2}|\Sigma|^{1/2}) - \frac{1}{2}\sum_{i=1}^{m}(x-\mu_{z_i})^T\,\Sigma^{-1}(x-\mu_{z_i})$$

The first term of the likelihood does not matter since it is independent of z.

Unsupervised: How many clusters $\mathbf{z}$?



Concrete Formulation of unsupervised learning

Estimate Parameters by $\mathbf{x}$-MLE:

$$
\begin{aligned}
l(x|\lambda, \mu, \Sigma) &= \sum_{i=1}^{m} \log p(x_i|\lambda, \mu, \Sigma) \\
&= \sum_{i=1}^{m} \log \sum_{z} p(x_i|z_i, \mu, \Sigma)\, p(z_i|\lambda)
\end{aligned}
$$

Not Solvable analytically! EM and Variational. Or do MCMC.

Semi-supervised learning

We have some labels, but typically very few labels: not enough to form a good training set. Likelihood a combination.

$$l(\{x_i\}, \{x_j\}, \{z_i\}|\theta, \lambda) = \sum_i log\, p(x_i, z_i|\lambda, \theta) + \sum_j log\, p(x_j|\lambda, \theta)$$

$$= \sum_i log\, p(z_i|\lambda) p(x_i|z,\theta) + \sum_j log \sum_z p(z_j|\lambda) p(x_j|z_j, \theta)$$

Here i ranges over the data points where we have labels, and j over the data points where we dont.

Semi-supervised learning

Basic Idea: there is structure in $p(x)$ which might help us divine the conditionals, thus combine full-data and $\mathbf{x}$-likelihood.

Include x on the validation set in the likelihood, and x and z on the training set in the likelihood.

Has been very useful for Naive Bayes.

Decision Theory

Predictions (or actions based on predictions) are described by a utility or loss function, whose values can be computed given the observed data.

Point Predictions: squared loss

Sometimes we want to make point predictions. In this case a is a single numb er.

squared error loss/utility: $l(a, y^*) = (a - y^*)^2$

The optimal point prediction that minimizes the expected loss (negative expected utility):

$$\bar{l}\,(a) = \int dy^*\,(a - y^*)^2\,p(y^*|D, M),$$

is the posterior predictive mean:

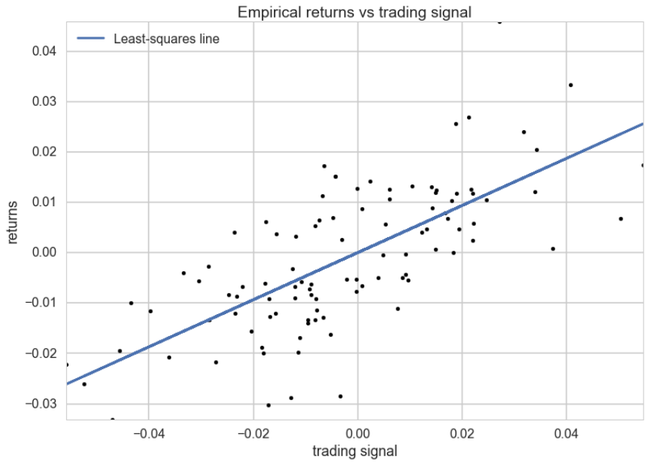
$$\hat{a} = E_p[y^*].$$

The expected loss then becomes:

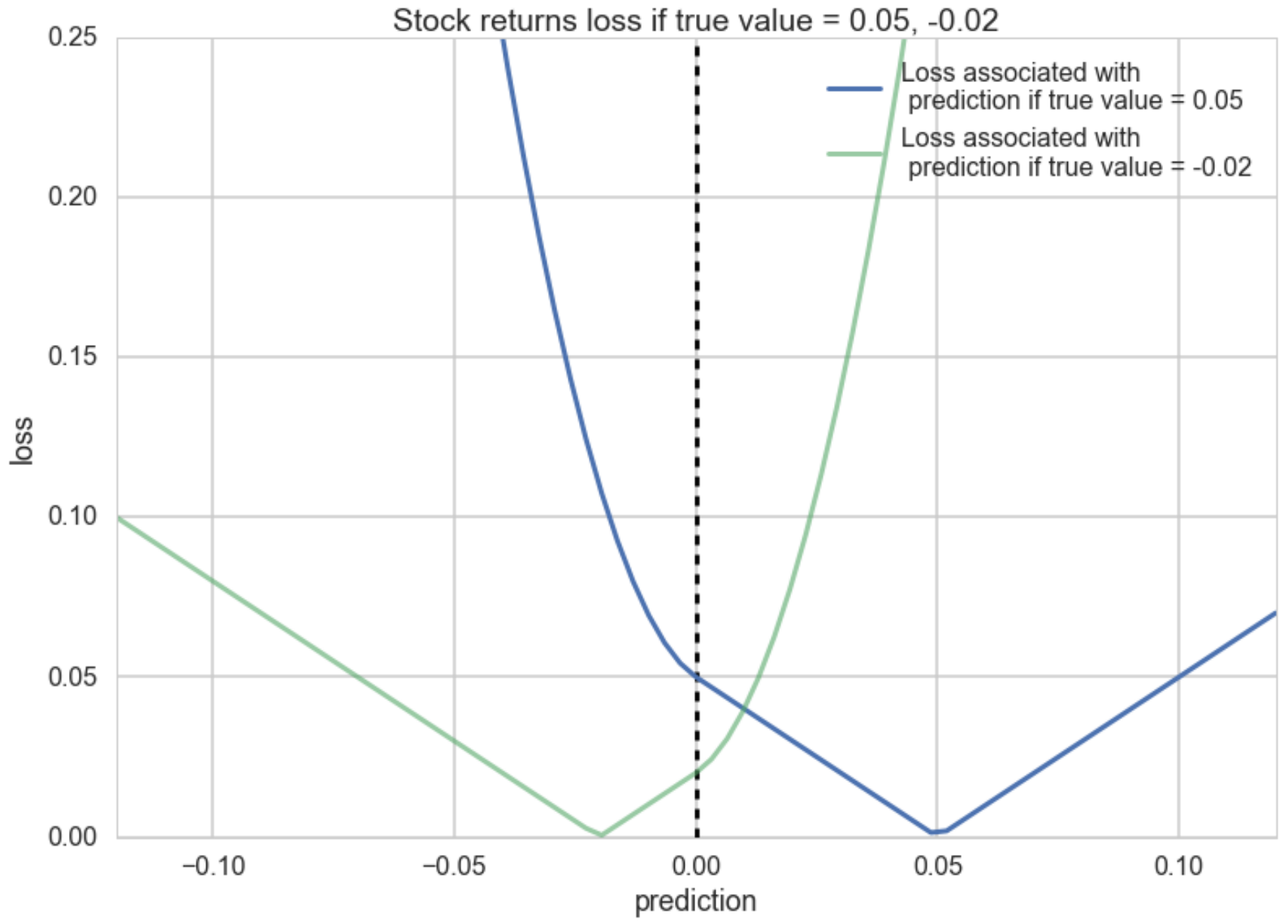
$$\bar{l}(\hat{a}) = \int dy^* \, (\hat{a} - y^*)^2 \, p(y^*|D, M) = \int dy^* \, (E_p[y^*] - y^*)^2 \, p(y^*|D, M) = Var_p[y^*]$$

Squared loss $\implies$ we dont care about skewness or kurtosis

Custom Loss: Stock Market Returns




```
def stock_loss(stock_return, pred, alpha = 100.):
    if stock_return * pred < 0:
        #opposite signs, not good
        return alpha*pred**2 - np.sign(stock_return)*pred \
                        + abs(stock_return)
    else:
        return abs(stock_return - pred)
```

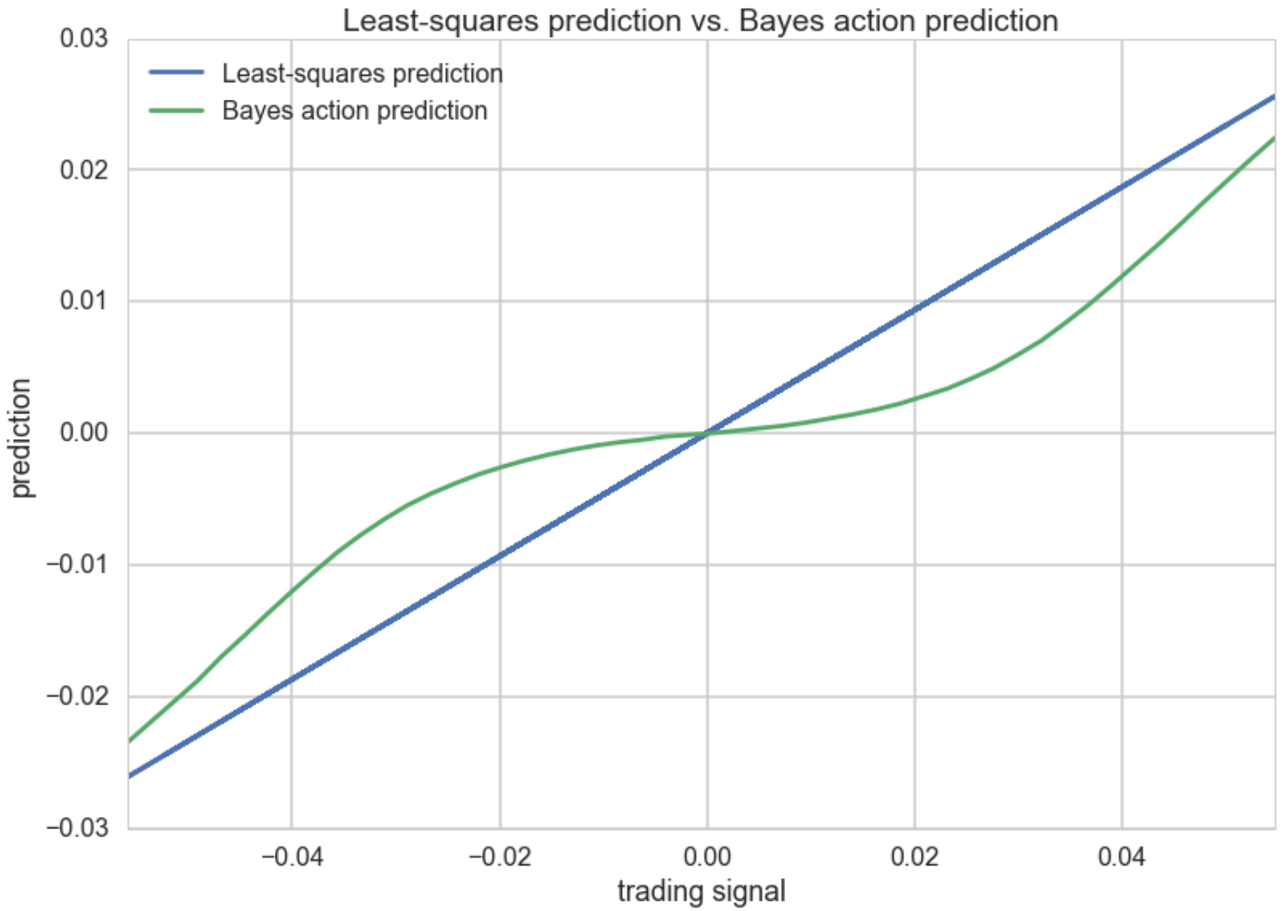


Loss at every x

```
noise = std_samples*np.random.randn(N)

#posterior predictive samples at every x
possible_outcomes = lambda signal: alpha_samples + beta_samples*signal + noise


opt_predictions = np.zeros(50)
trading_signals =  np.linspace(X.min(), X.max(), 50)
for i, _signal in enumerate(trading_signals):
        _possible_outcomes = possible_outcomes(_signal)
        #expected loss over posterior predictive
        tomin = lambda pred: stock_loss(_possible_outcomes, pred).mean()
        #bayes action minimizes expected loss
        opt_predictions[i] = fmin(tomin, 0, disp = False)
```



The two risks

There are *two risks in learning* that we must consider, one to *estimate probabilities*, which we call **estimation risk**, and one to *make decisions*, which we call **decision risk**.

The **decision loss** $l(y, a)$ or **utility** $u(l, a)$ (profit, or benefit) in making a decision a when the predicted variable has value y. For example, we must provide all of the losses l(no-cancer, biopsy), l(cancer, biopsy), l(no-cancer, no-biopsy), and l(cancer, no-biopsy). One set of choices for these losses may be 20, 0, 0, 200 respectively.

Classification Risk

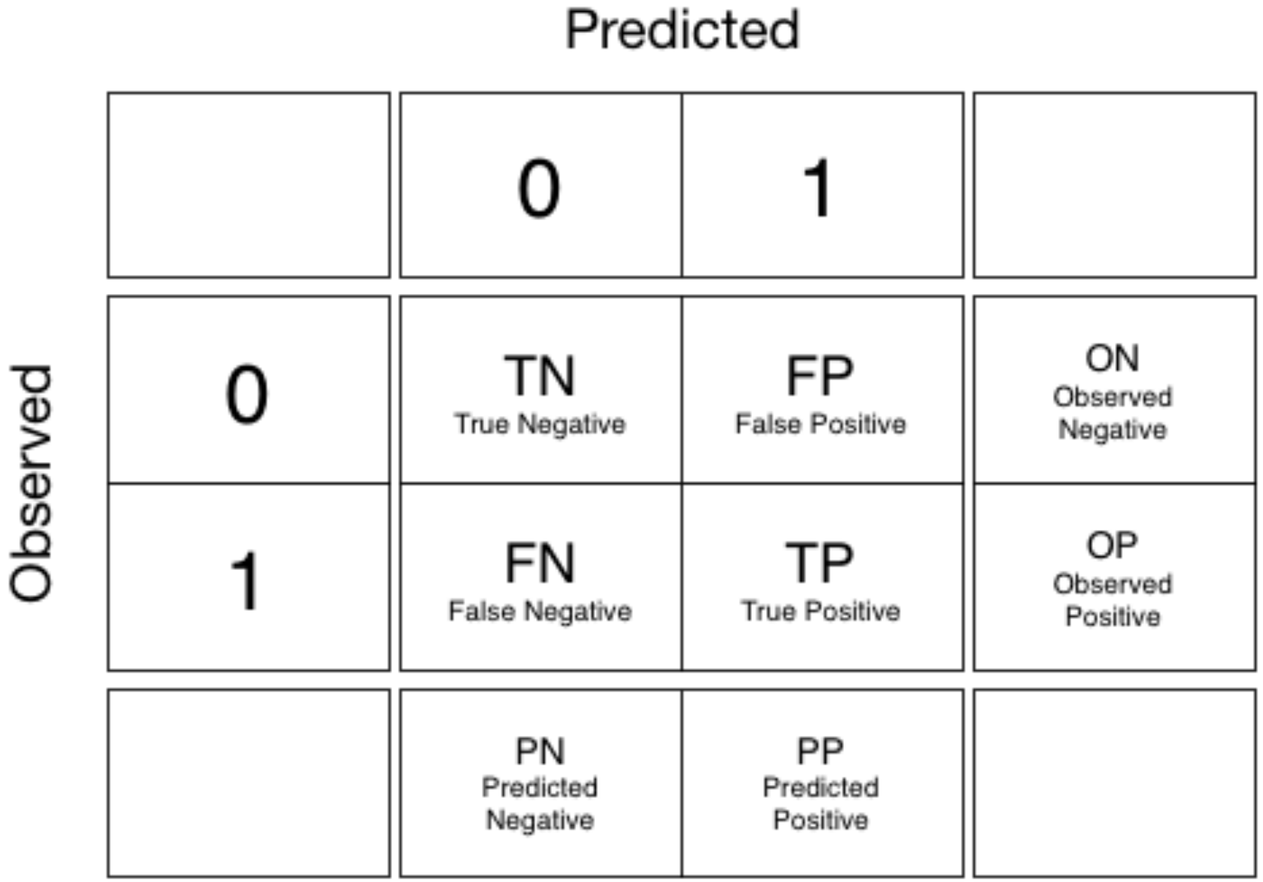
$$R_a(x) = \sum_y l(y, a(x))p(y|x)$$

That is, we calculate the **predictive averaged risk** over all choices y, of making choice a for a given data point.

Overall risk, given all the data points in our set:

$$R(a) = \int dx p(x) R_a(x)$$

Two class Classification



$$R_a(x) = l(1,g)p(1|x) + l(0,g)p(0|x).$$

Then for the "decision" $a = 1$ we have:

$$R_1(x) = l(1,1)p(1|x) + l(0,1)p(0|x),$$

and for the "decision" $a = 0$ we have:

$$R_0(x) = l(1,0)p(1|x) + l(0,0)p(0|x).$$

Now, we'd choose 1 for the data point at x if:

$$R_1(x) < R_0(x).$$

$$P(1|x)(l(1,1) - l(1,0)) < p(0|x)(l(0,0) - l(0,1))$$

So, to choose '1', the Bayes risk can be obtained by setting:

$$p(1|x) > rP(0|x) \implies r = \frac{l(0,1) - l(0,0)}{l(1,0) - l(1,1)}$$

$$P(1|x) > t = \frac{r}{1+r}.$$

One can use the prediction cost matrix corresponding to the consufion matrix

$$r == \frac{c_{FP} - c_{TN}}{c_{FN} - c_{TP}}$$

If you assume that True positives and True negatives have no cost, and the cost of a false positive is equal to that of a false positive, then $r = 1$ and the threshold is the usual intuitive $t = 0.5$.

		Predicted	
		0	1
Observed	0	TNC True Negative Cost	FPC False Positive Cost
	1	FNC False Negative Cost	TPC True Positive Cost