
Univ.AI

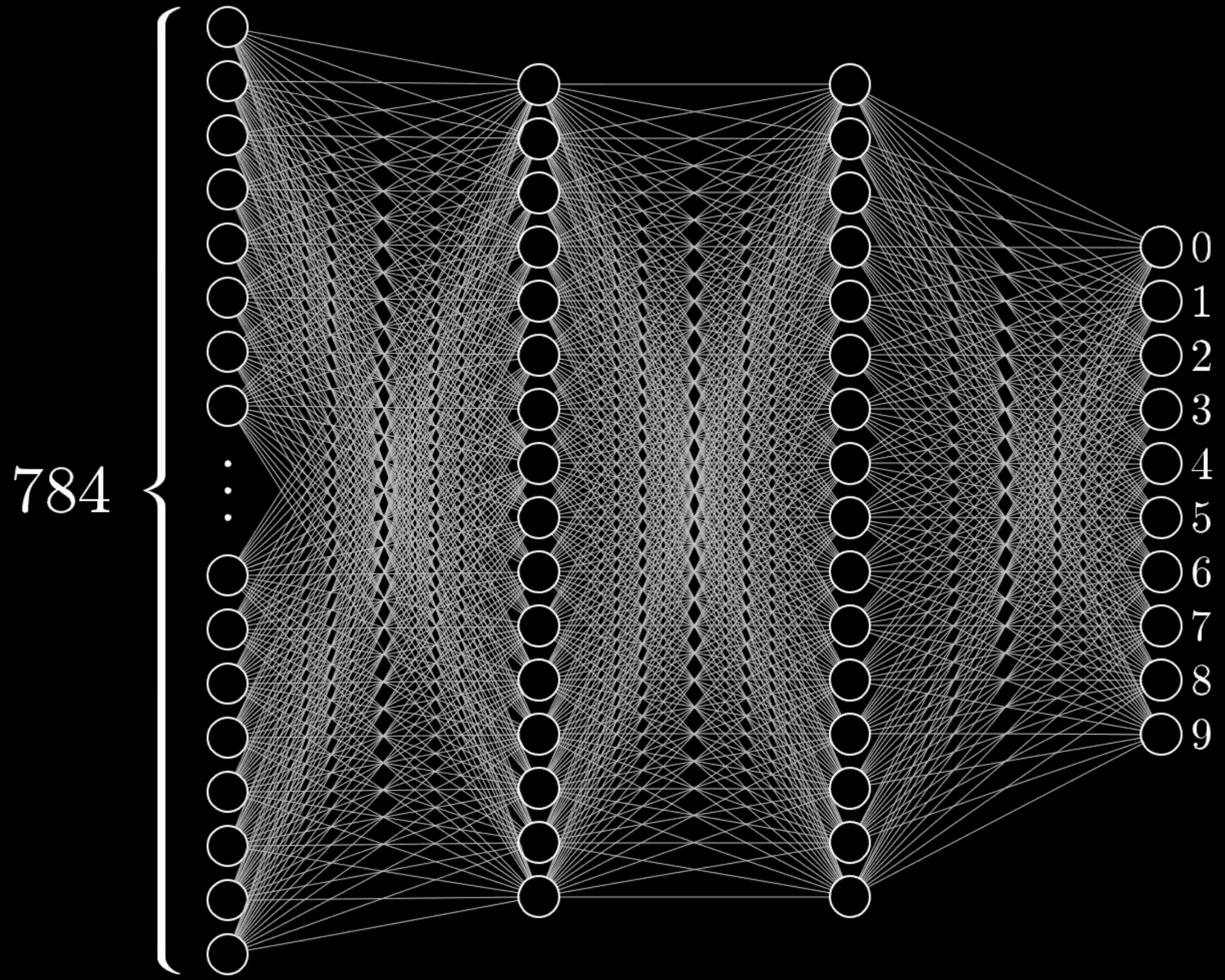
Univ.AI

Losses,

and

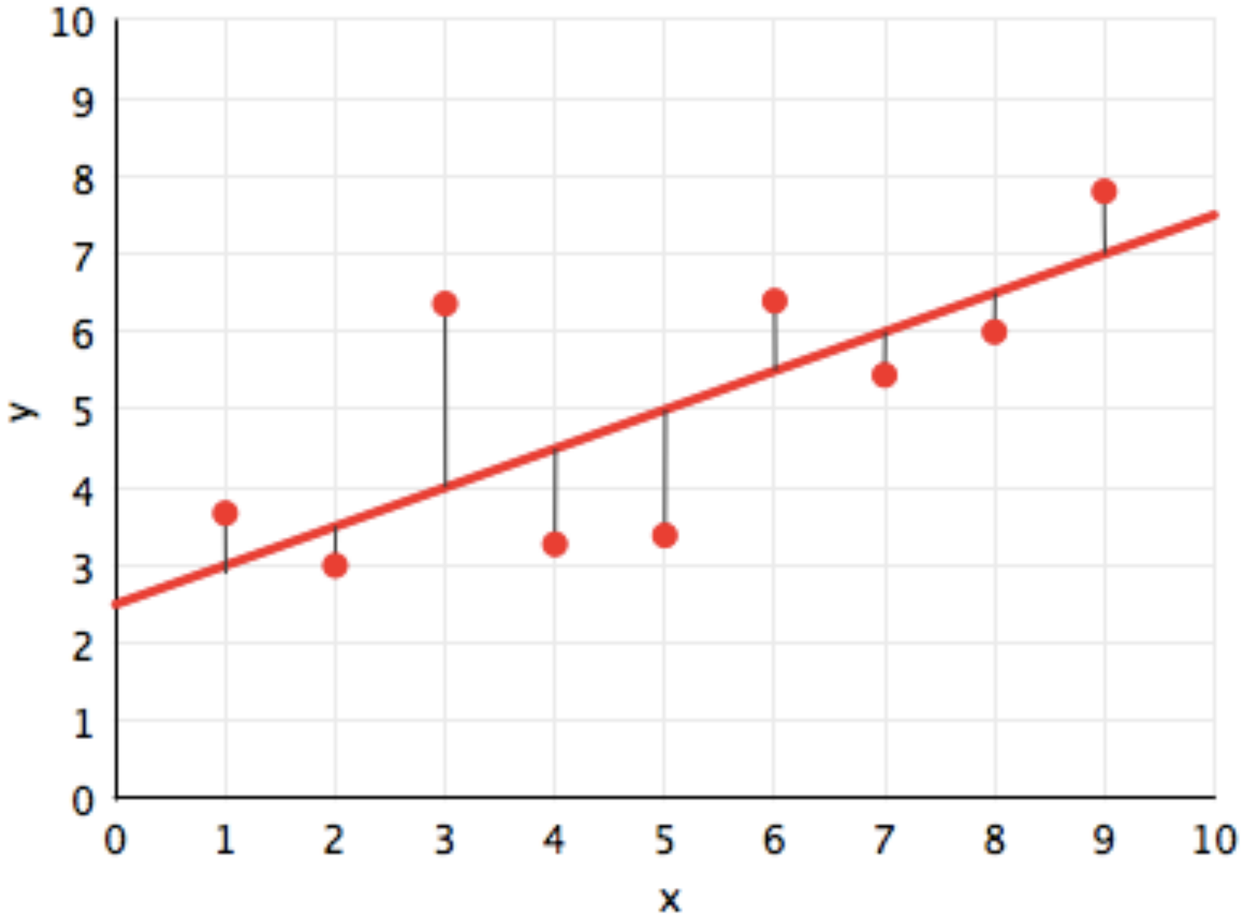
Learning

Minimize Cost


784
0
1
2
3
4
5
6
7
8
9

Regression: continuous prediction

- how many dollars will you spend?
- what is your creditworthiness
- how many people will vote for Bernie t days before election
- use to predict probabilities for classification
- causal modeling in econometrics



Linear Regression

$$\hat{y} = f_\theta(x) = \theta^T x$$

Cost Function:

$$R(\theta) = \frac{1}{2}\sum_{i=1}^{m}(f_\theta(x^{(i)} - y^{(i)})^2$$

MINIMIZE SQUARED ERROR. Its Convex!

Cost function

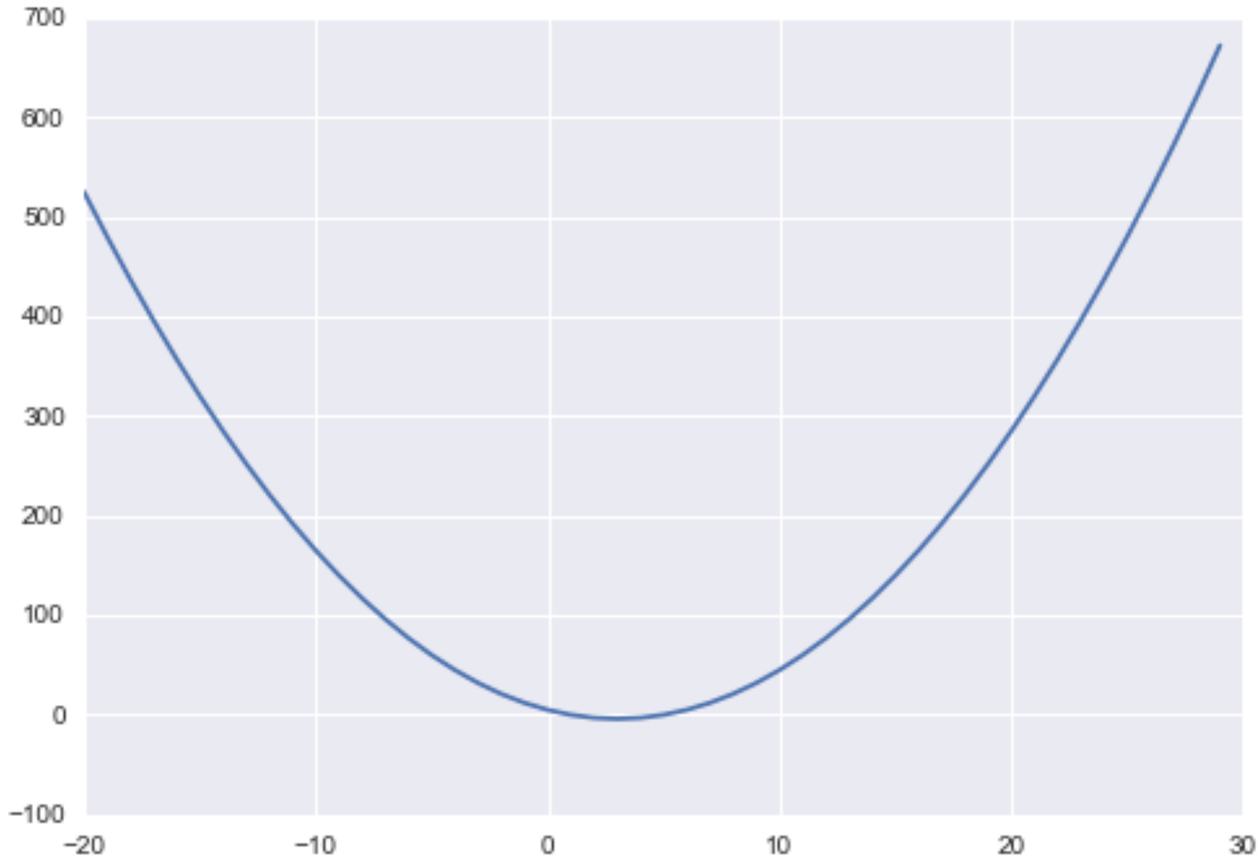
$$C(\underbrace{w_1, w_2, \ldots, w_{13{,}002}}_{\text{Weights and biases}})$$

Gradient ascent (descent)

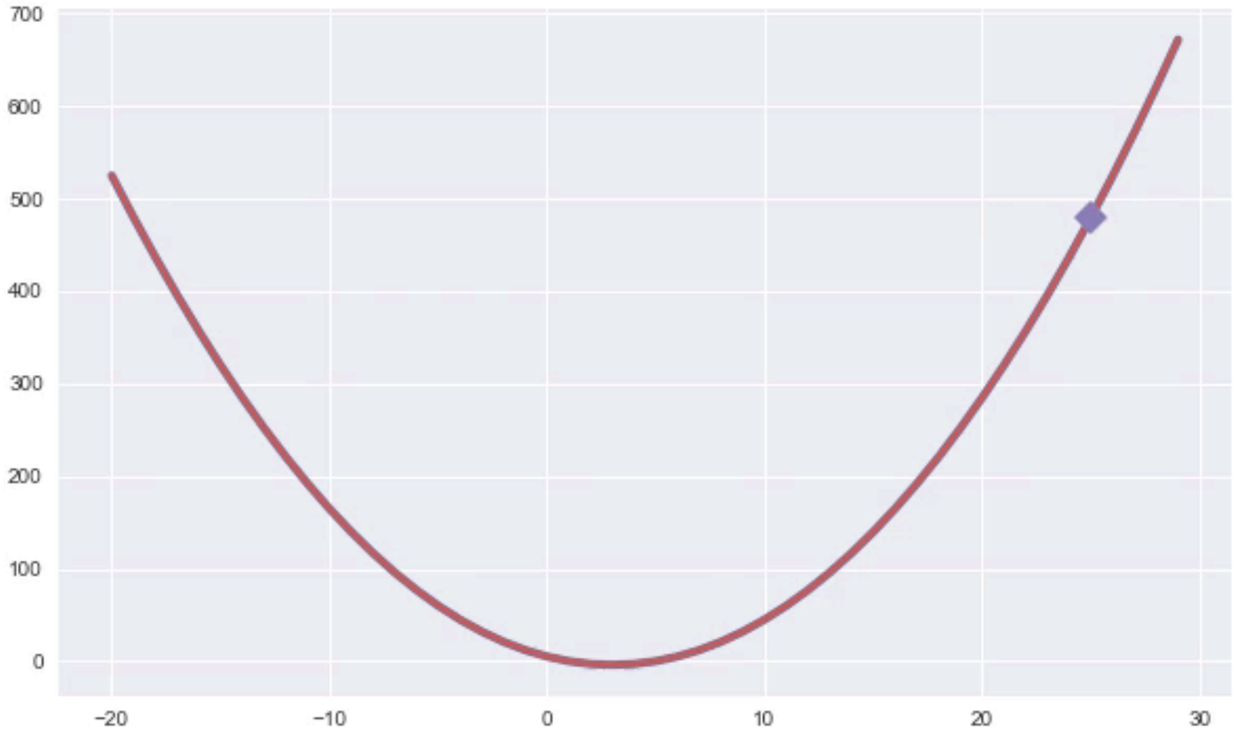
basically go opposite the direction of the derivative.

Consider the objective function:
$J(x) = x^2 - 6x + 5$

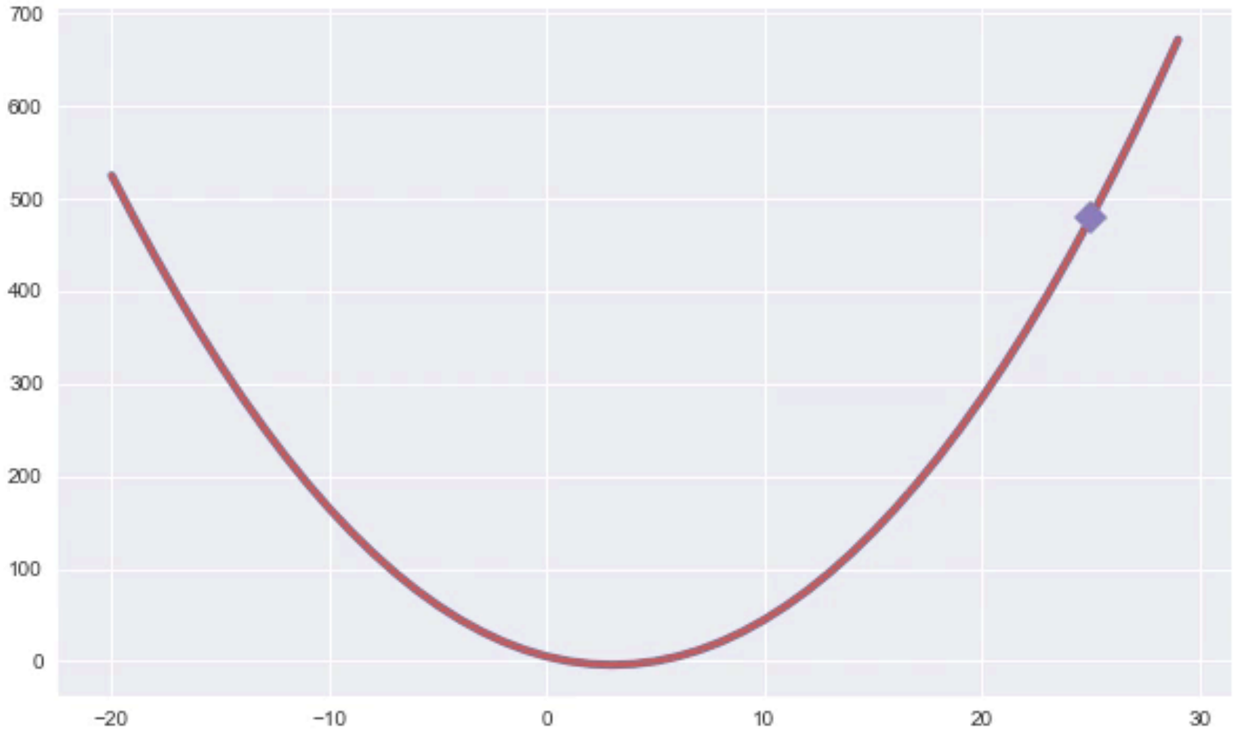
```
gradient = fprime(old_x)
move = gradient * step
current_x = old_x - move
```



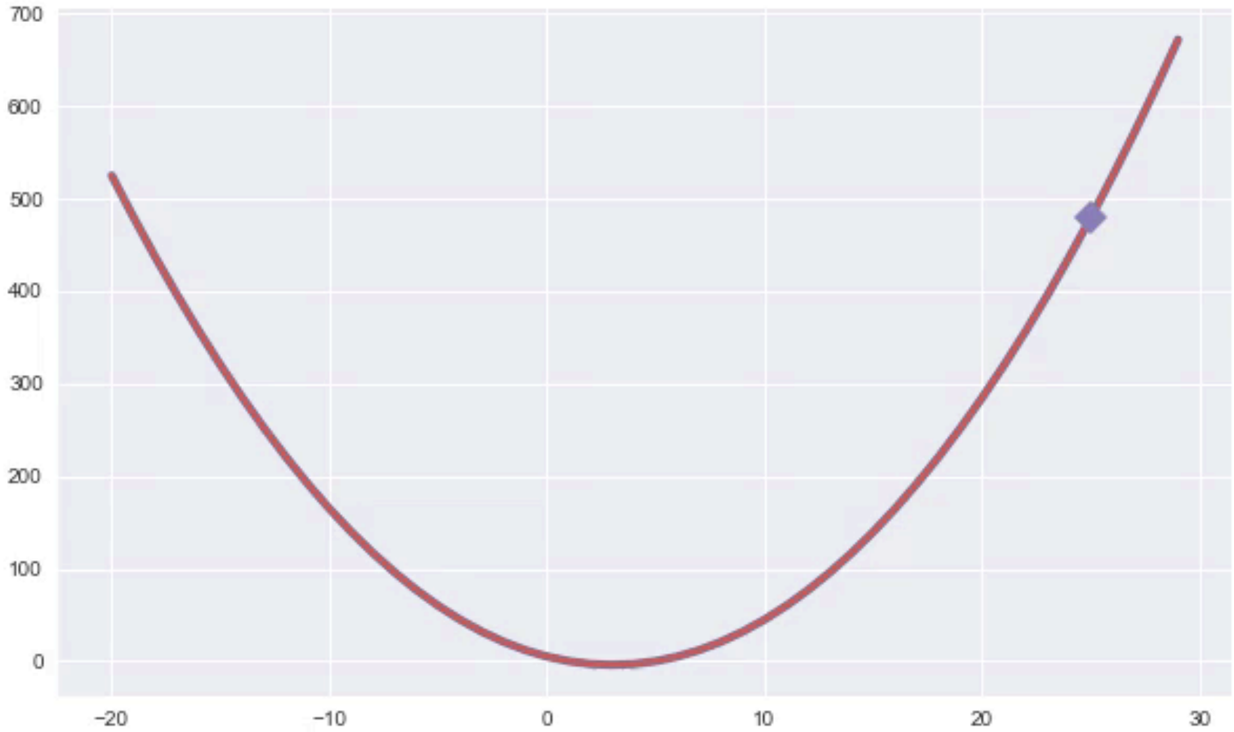
good step size



too big step size



too small step size



Gradient Descent

$$\theta := \theta - \eta \nabla_\theta R(\theta) = \theta - \eta \sum_{i=1}^{m} \nabla R_i(\theta)$$

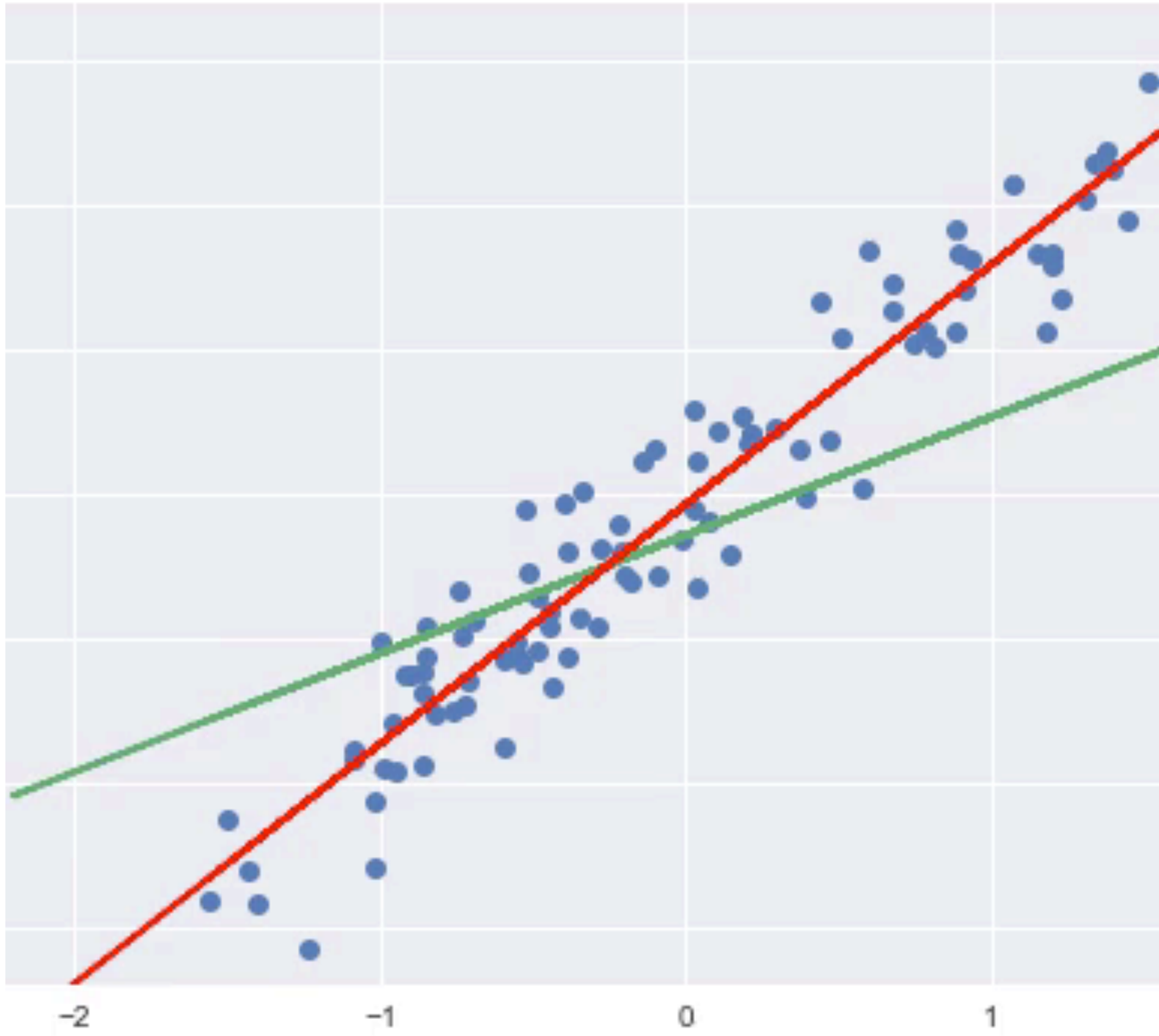
where η is the learning rate.

ENTIRE DATASET NEEDED

```
for i in range(n_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad`
```

Linear Regression: Gradient Descent

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - f_\theta(x^{(i)}))x_j^{(i)}$$



Stochastic Gradient Descent

$$\theta := \theta - \alpha \nabla_\theta R_i(\theta)$$

ONE POINT AT A TIME

For Linear Regression:

$$\theta_j := \theta_j + \alpha(y^{(i)} - f_\theta(x^{(i)}))x_j^{(i)}$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

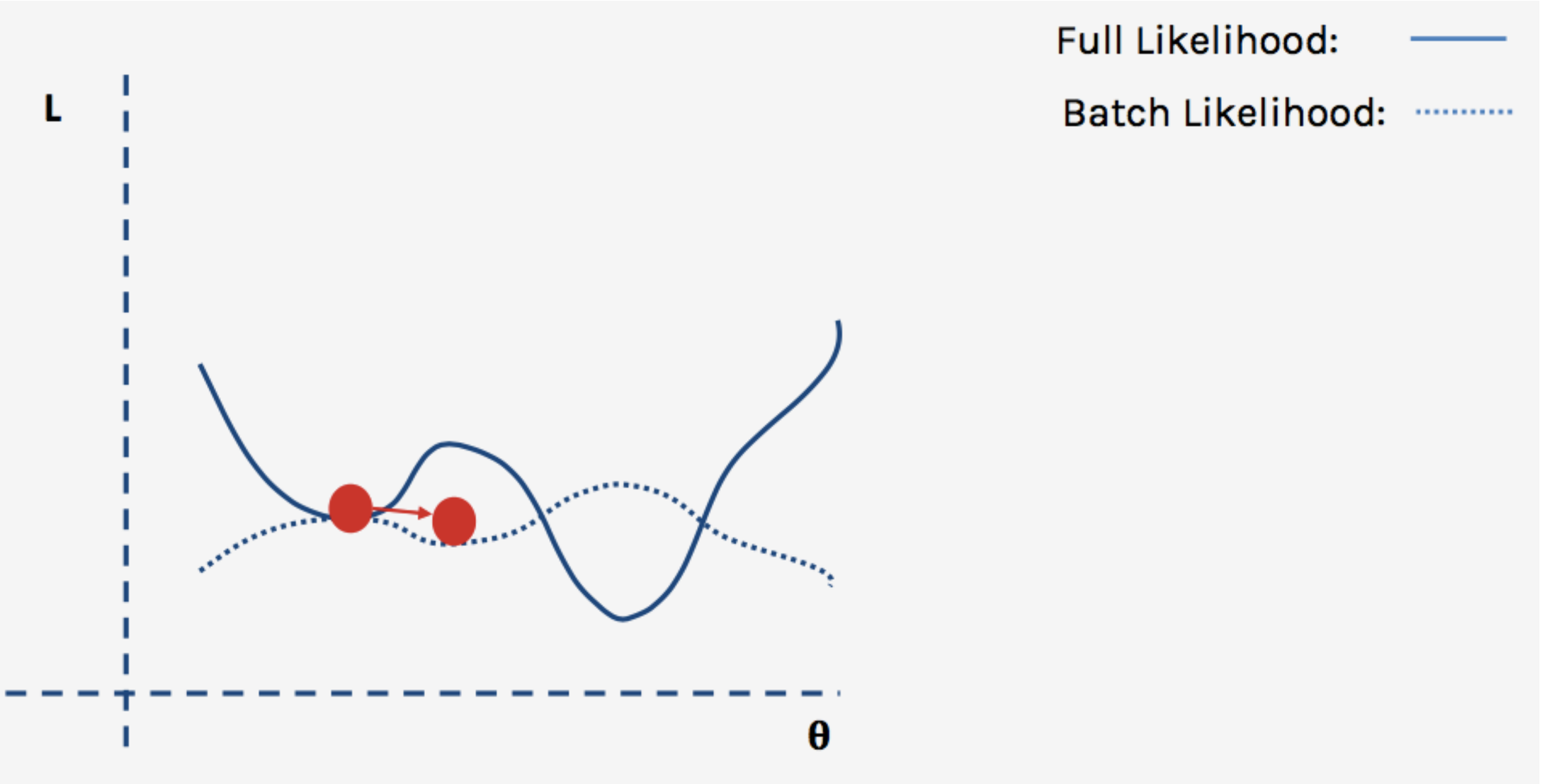
Mini-Batch SGD (the most used)

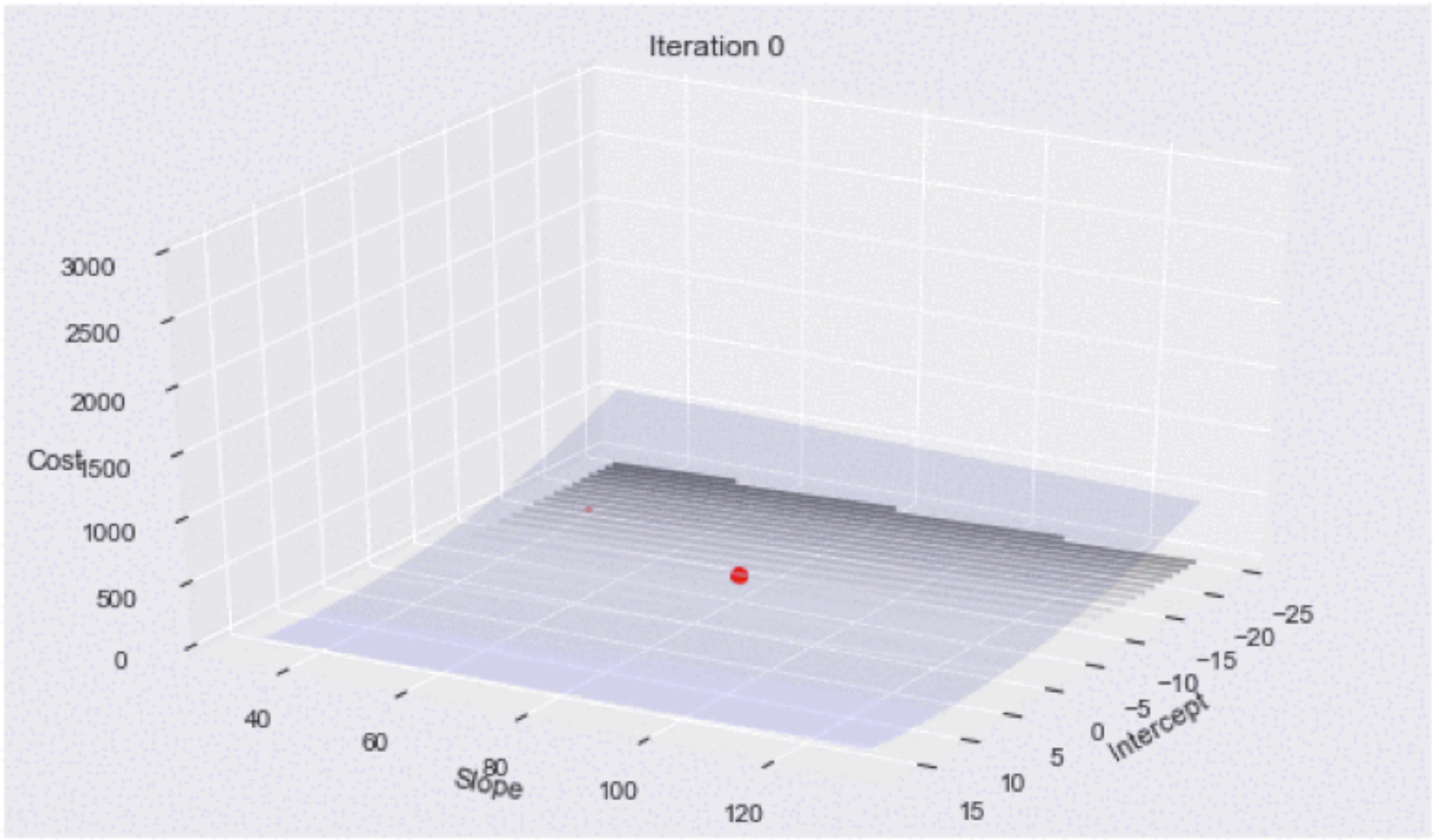
$$\theta := \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(mb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Mini-Batch: do some at a time

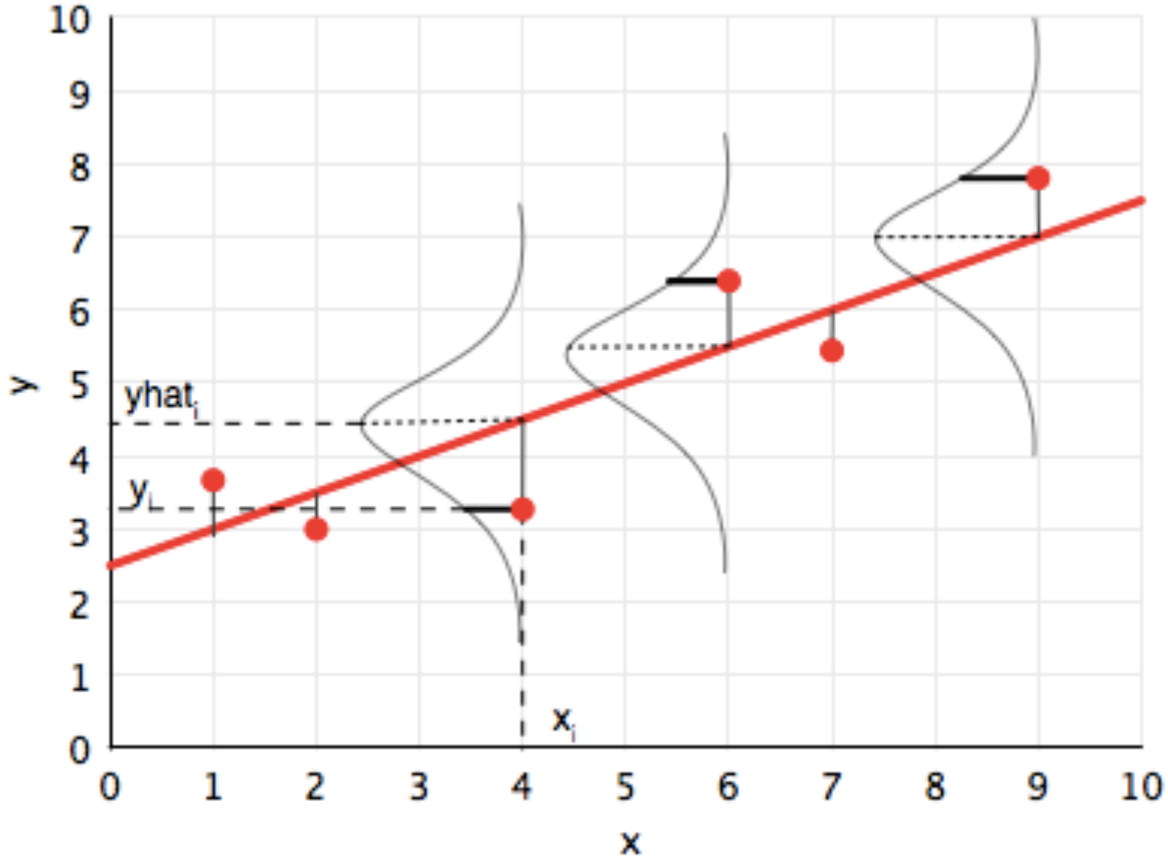
- the risk surface changes at each gradient calculation
- thus things are noisy
- cumulated risk is smoother, can be used to compare to SGD
- epochs are now the number of times you revisit the full dataset
- shuffle in-between to provide even more stochasticity


Full Likelihood:
Batch Likelihood:
L
θ


Iteration 0
Cost
3000
2500
2000
1500
1000
500
0
40
60
80
100
120
Slope
15
10
5
0
−5
−10
−15
−20
−25
Intercept

Ok, so where did this Mean Squared Loss come from?

Gaussian Distribution assumption !



Gaussian Distribution assumption

Each y_i is gaussian distributed with mean $\mathbf{w} \cdot \mathbf{x}_i$ (the y predicted by the regression line) and variance σ^2:

$$y_i \sim N(\mathbf{w} \cdot \mathbf{x}_i, \sigma^2).$$

$$N(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(y-\mu)^2/2\sigma^2},$$

We can then write the likelihood:

$$\mathcal{L} = p(\mathbf{y}|\mathbf{x}, \mathbf{w}, \sigma) = \prod_i p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w}, \sigma)$$

$$\mathcal{L} = (2\pi\sigma^2)^{(-n/2)} e^{\frac{-1}{2\sigma^2}\sum_i (y_i - \mathbf{w}\cdot\mathbf{x}_i)^2}.$$

The log likelihood ℓ then is given by:

$$\ell = \frac{-n}{2} log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_i (y_i - \mathbf{w}\cdot\mathbf{x}_i)^2.$$

Maximize ℓ ?

No. Minimize $-\ell$ using Gradient Descent!

If all you care for are the parameters w, then:

Minimize Loss or Cost:

$$Loss = \frac{1}{N}\sum_i (y_i - \mathbf{w}\cdot\mathbf{x}_i)^2.$$

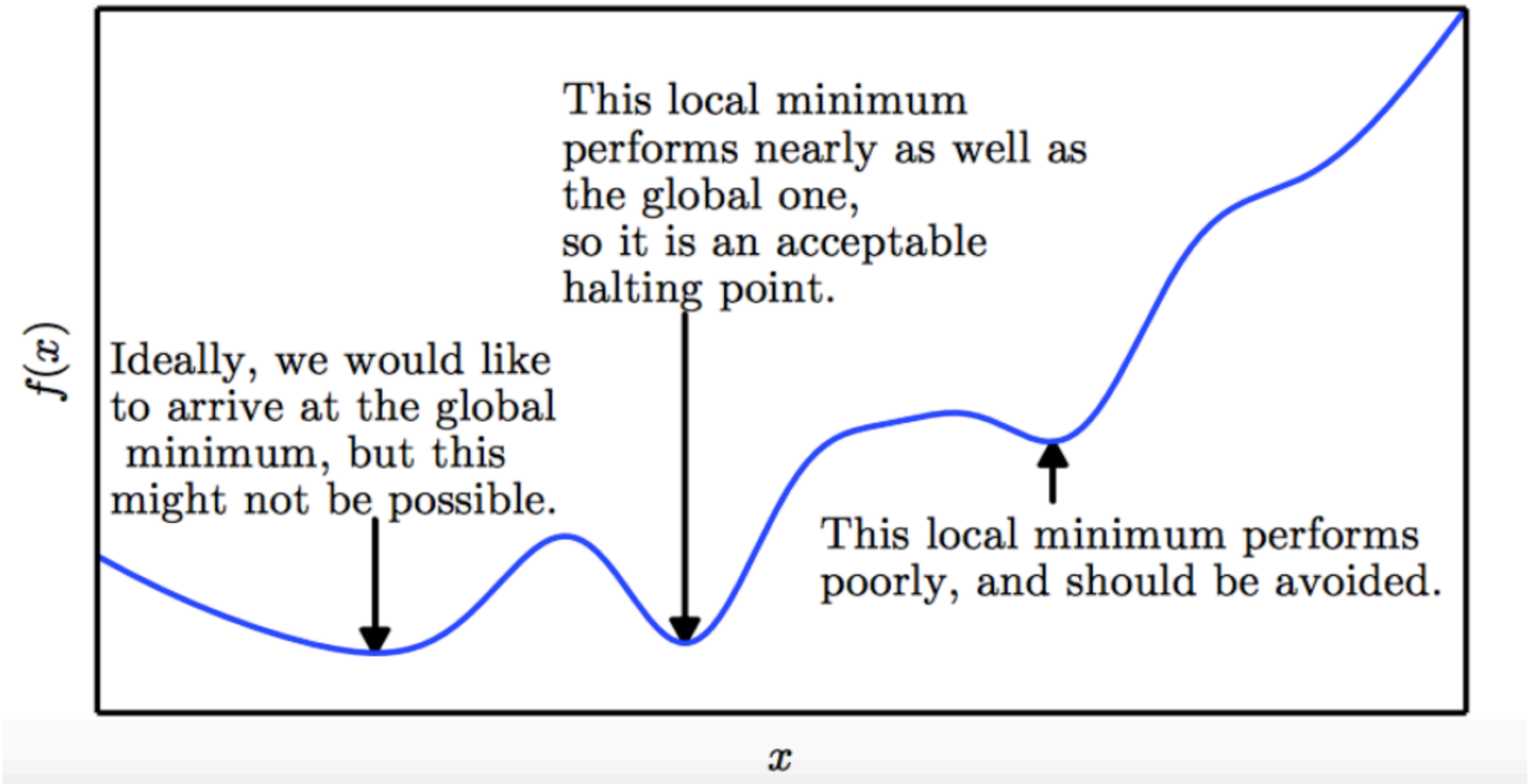
Where's the neural network?

Replace $\mathbf{w} \cdot \mathbf{x}_i)^2$ with $NN(\mathbf{w} \cdot \mathbf{x}_i)^2)$.

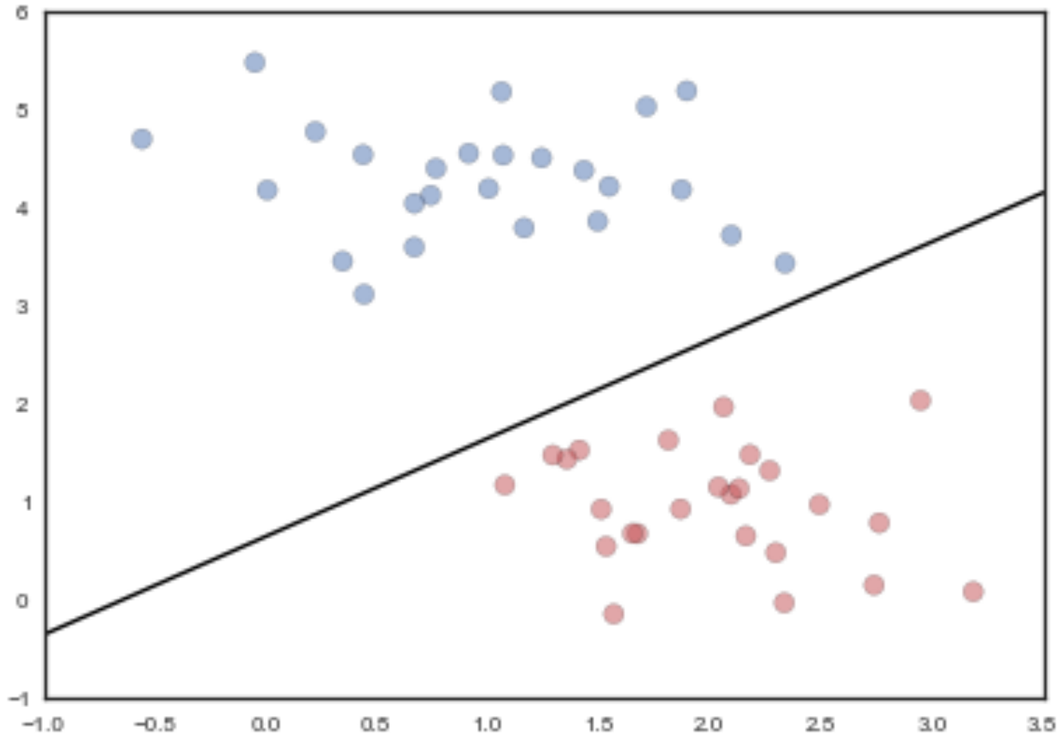
So:

$$Loss = \frac{1}{N}\sum_i (y_i - NN\,(\mathbf{w} \cdot \mathbf{x}_i)^2).$$

The Loss is now NOT CONVEX!



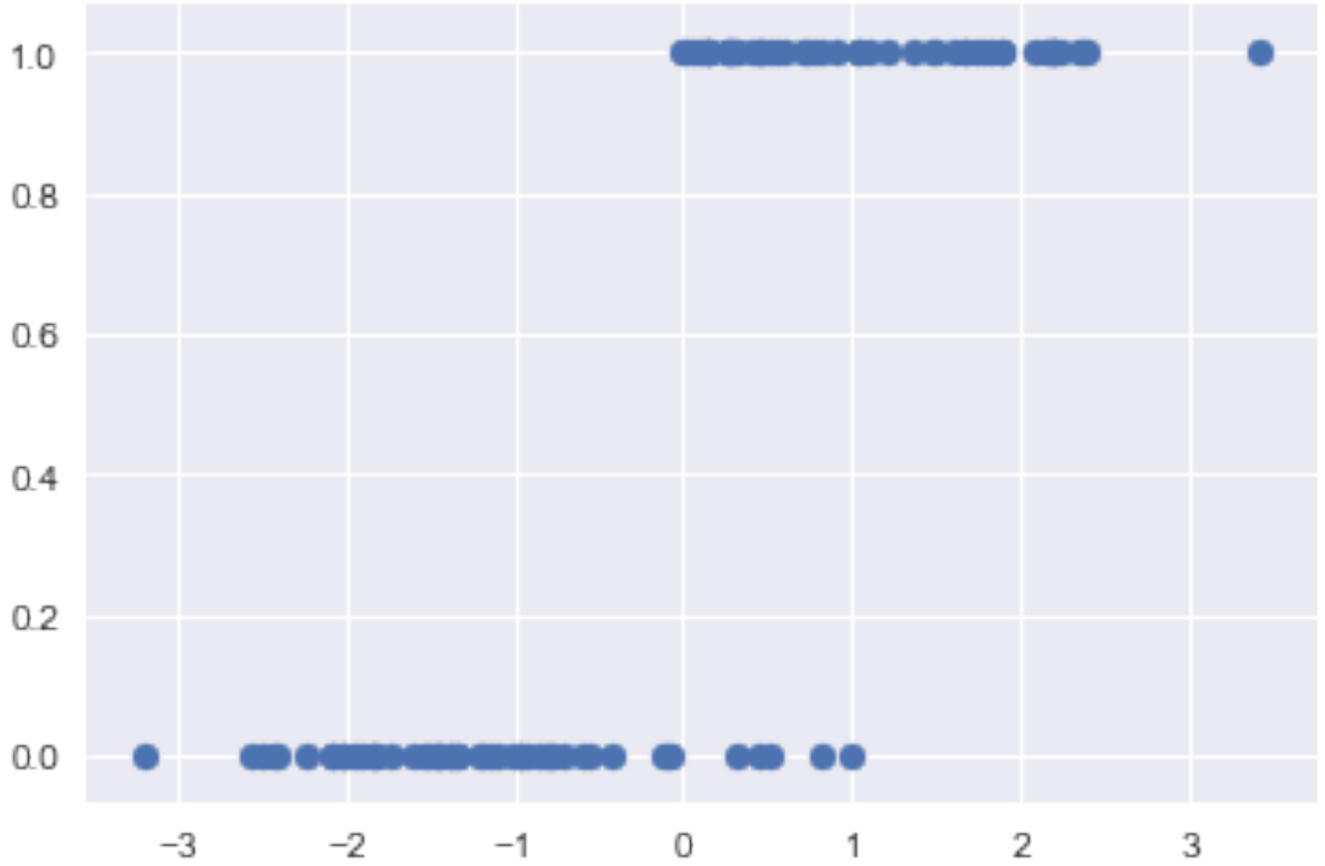
CLASSIFICATION



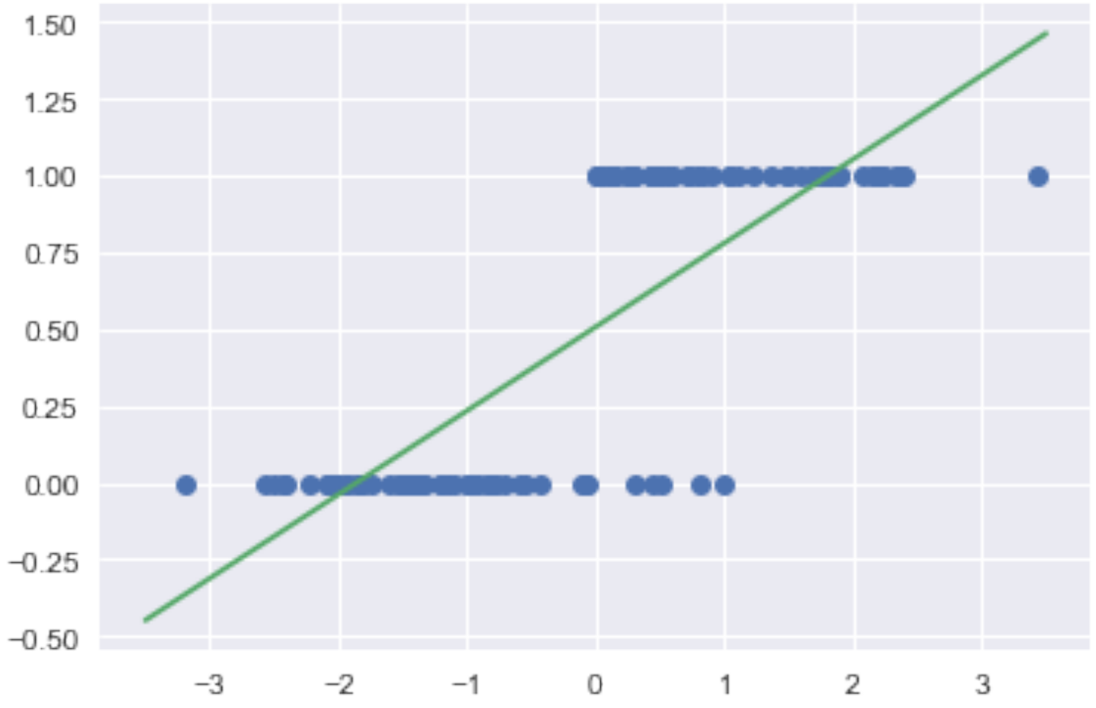
- will a customer churn?
- is this a check? For how much?
- a man or a woman?
- will this customer buy?
- do you have cancer?
- is this spam?
- whose picture is this?
- what is this text about?[j]

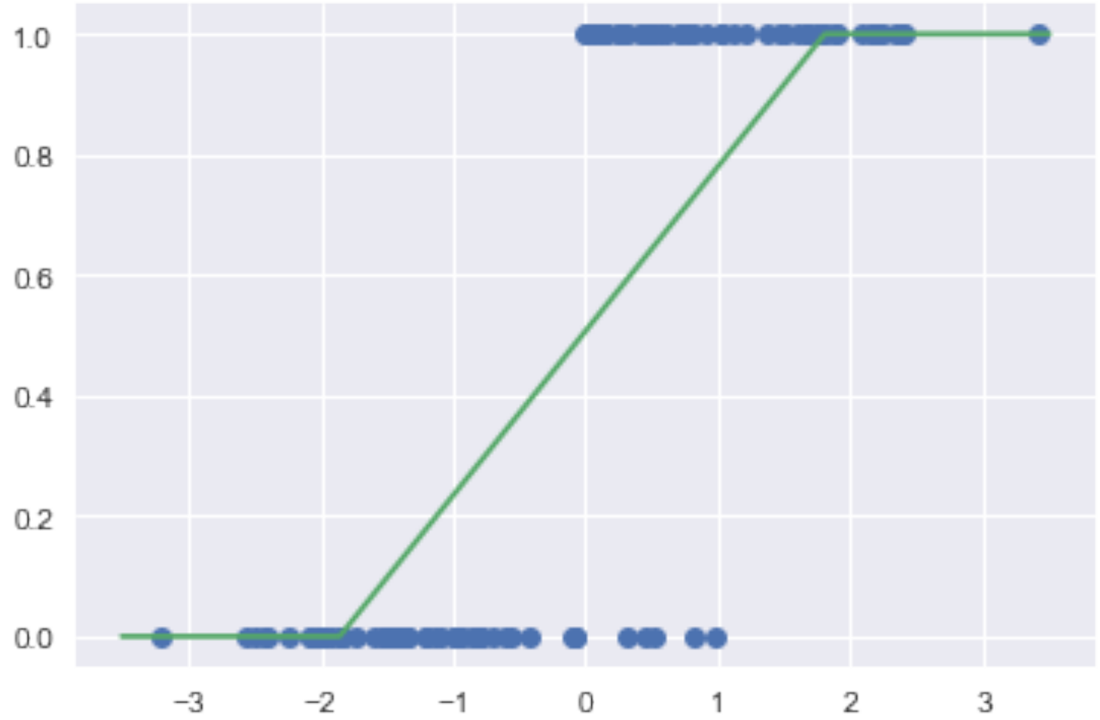
[j] image from code in http://bit.ly/1Azg29G

1-D classification problem



1-D Using Linear regression





MLE for Logistic Regression

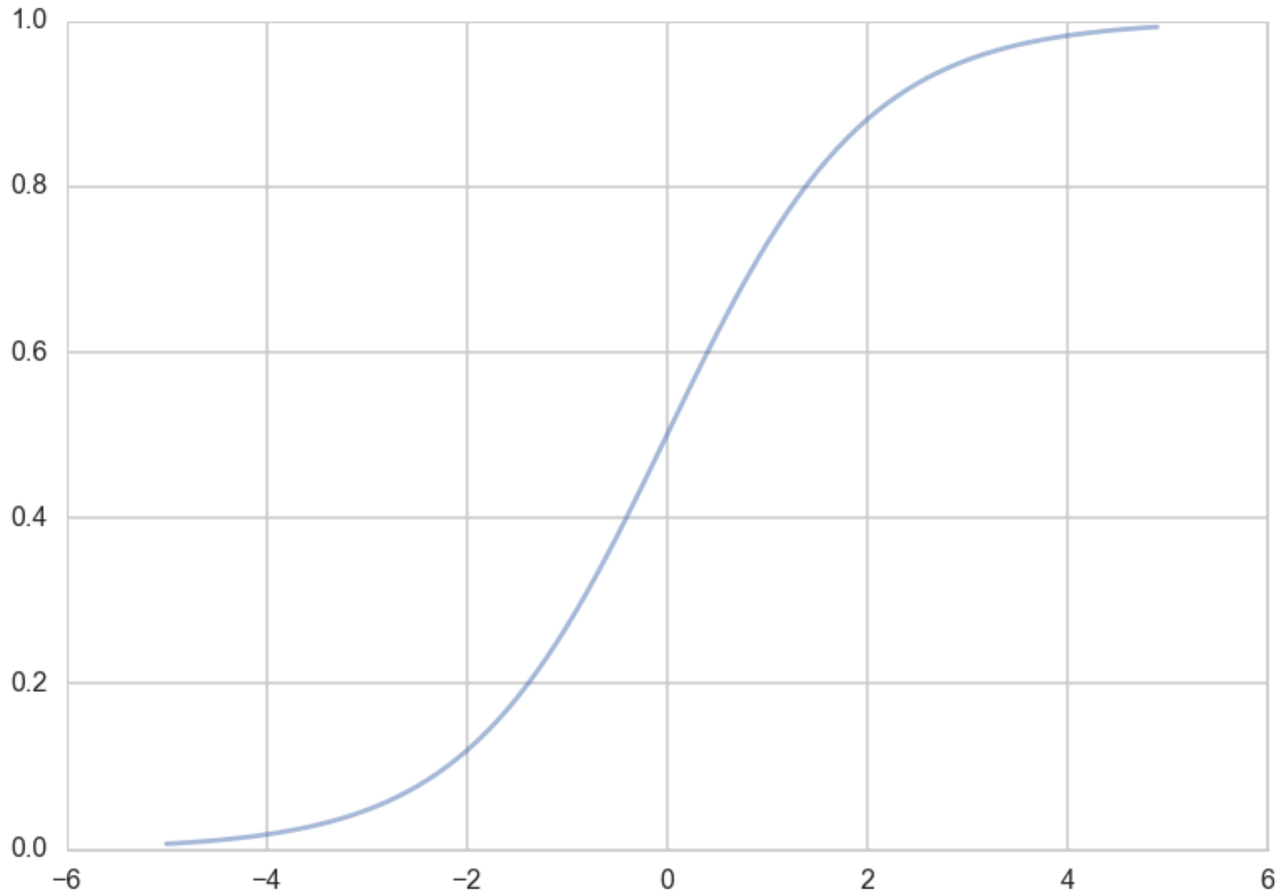
- example of a Generalized Linear Model (GLM)
- "Squeeze" linear regression through a **Sigmoid** function
- this bounds the output to be a probability

Sigmoid function

This function is plotted below:

```
h = lambda z: 1./(1+np.exp(-z))
zs=np.arange(-5,5,0.1)
plt.plot(zs, h(zs), alpha=0.5);
```

Identify: $z = \mathbf{w} \cdot \mathbf{x}$ and $h(\mathbf{w} \cdot \mathbf{x})$ with the probability that the sample is a '1' ($y = 1$).



Then, the conditional probabilities of $y = 1$ or $y = 0$ given a particular sample's features $\mathbf{x}$ are:

$$P(y = 1|\mathbf{x}) = h(\mathbf{w} \cdot \mathbf{x})$$
$$P(y = 0|\mathbf{x}) = 1 - h(\mathbf{w} \cdot \mathbf{x}).$$

These two can be written together as

$$P(y|\mathbf{x}, \mathbf{w}) = h(\mathbf{w} \cdot \mathbf{x})^y (1 - h(\mathbf{w} \cdot \mathbf{x}))^{(1-y)}$$

BERNOULLI!!

Multiplying over the samples we get:

$$P(y|\mathbf{x}, \mathbf{w}) = P(\{y_i\}|\{\mathbf{x}_i\}, \mathbf{w}) = \prod_{y_i \in \mathcal{D}} P(y_i|\mathbf{x}_i, \mathbf{w}) = \prod_{y_i \in \mathcal{D}} h(\mathbf{w} \cdot \mathbf{x}_i)^{y_i} (1 - h(\mathbf{w} \cdot \mathbf{x}_i))^{(1-y_i)}$$

Indeed its important to realize that a particular sample can be thought of as a draw from some "true" probability distribution.

maximum likelihood estimation maximises the **likelihood of the sample y**, or alternately the log-likelihood,

$\mathcal{L} = P(y \mid \mathbf{x}, \mathbf{w})$. OR $\ell = log(P(y \mid \mathbf{x}, \mathbf{w}))$

Thus

$$
\begin{aligned}
\ell &= log\left(\prod_{y_i \in \mathcal{D}} h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i}\left(1-h(\mathbf{w}\cdot\mathbf{x}_i)\right)^{(1-y_i)}\right) \\
&= \sum_{y_i \in \mathcal{D}} log\left(h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i}\left(1-h(\mathbf{w}\cdot\mathbf{x}_i)\right)^{(1-y_i)}\right) \\
&= \sum_{y_i \in \mathcal{D}} log\, h(\mathbf{w}\cdot\mathbf{x}_i)^{y_i} + log\left(1-h(\mathbf{w}\cdot\mathbf{x}_i)\right)^{(1-y_i)} \\
&= \sum_{y_i \in \mathcal{D}} (y_i log(h(\mathbf{w}\cdot\mathbf{x})) + (1-y_i) log(1-h(\mathbf{w}\cdot\mathbf{x})))
\end{aligned}
$$

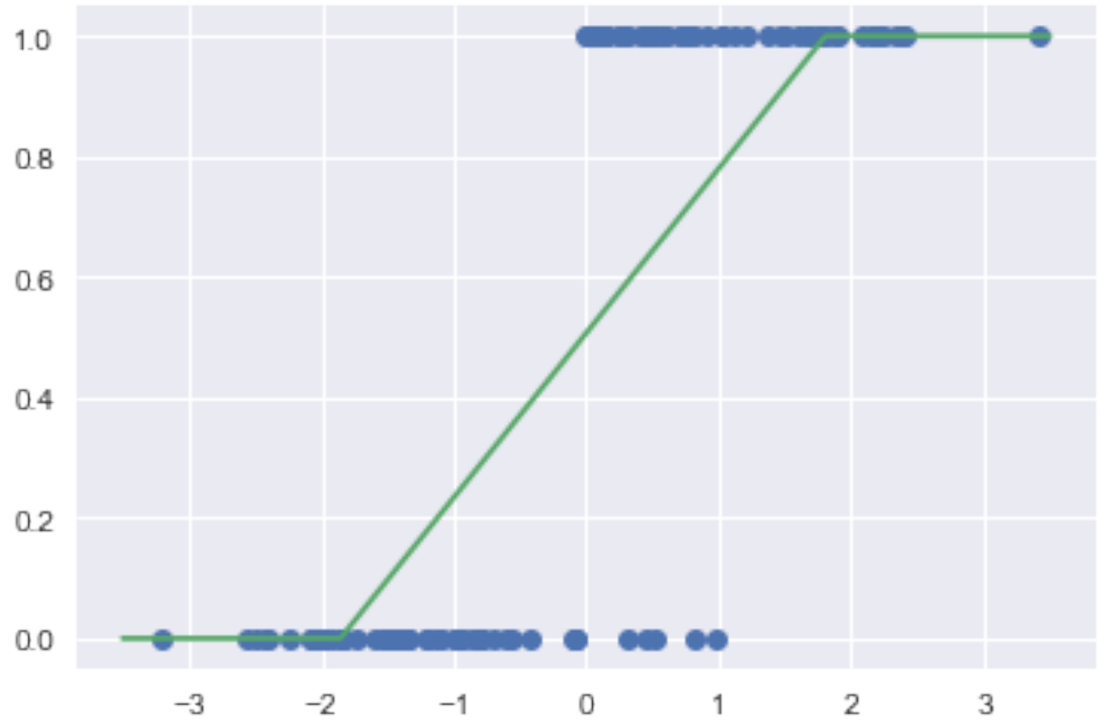
Logistic Regression: NLL

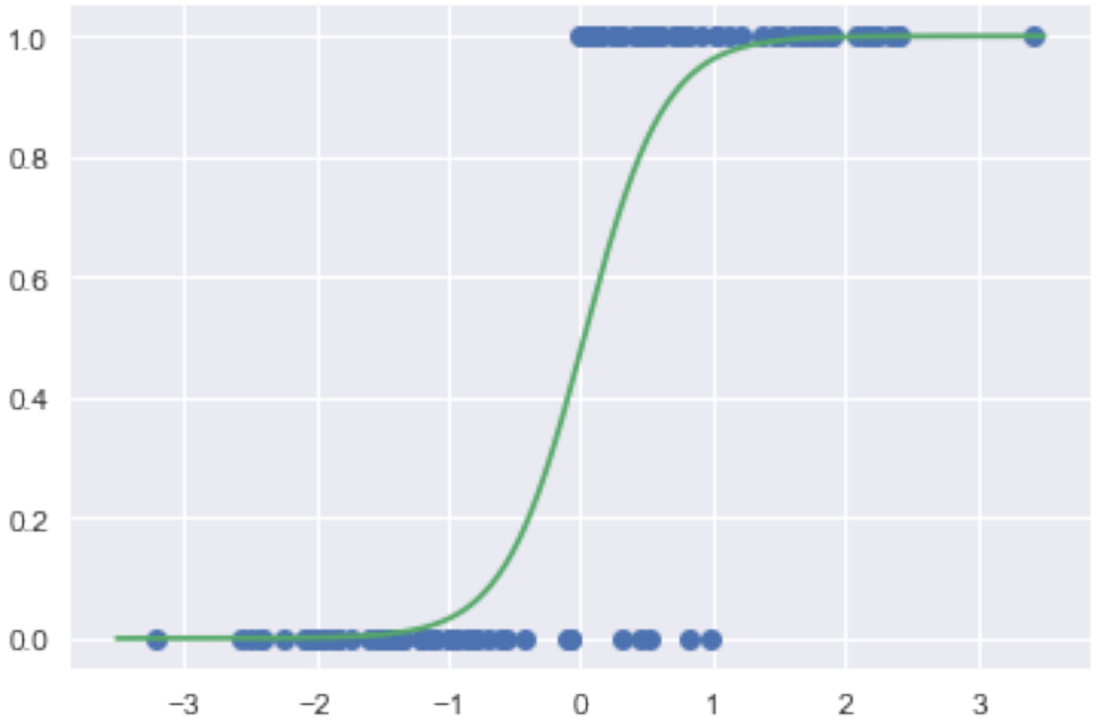
The negative of this log likelihood (NLL), also called *cross-entropy*.

$$NLL = -\sum_{y_i \in \mathcal{D}} (y_i log(h(\mathbf{w} \cdot \mathbf{x})) + (1 - y_i) log(1 - h(\mathbf{w} \cdot \mathbf{x})))$$

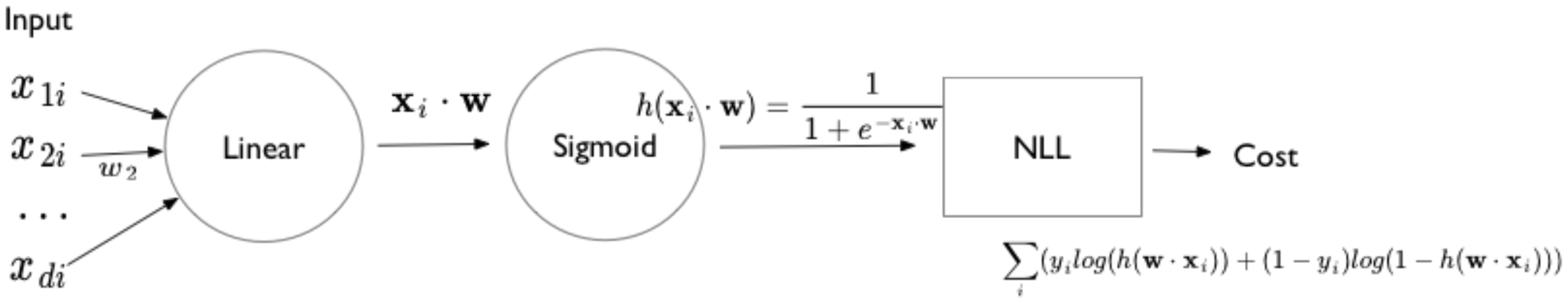
This loss is convex. Many ways can be chosen to minimize it, but we can just use gradient descent :-)

1-D Using Logistic regression





In diagram:


Input
x_{1i}
x_{2i}
w_2
$\cdots$
x_{di}
Linear
$\mathbf{x}_i \cdot \mathbf{w}$
Sigmoid
$h(\mathbf{x}_i \cdot \mathbf{w}) = \frac{1}{1+e^{-\mathbf{x}_i \cdot \mathbf{w}}}$
NLL
Cost
$\sum_i (y_i log(h(\mathbf{w} \cdot \mathbf{x}_i)) + (1-y_i) log(1-h(\mathbf{w} \cdot \mathbf{x}_i)))$

Softmax formulation

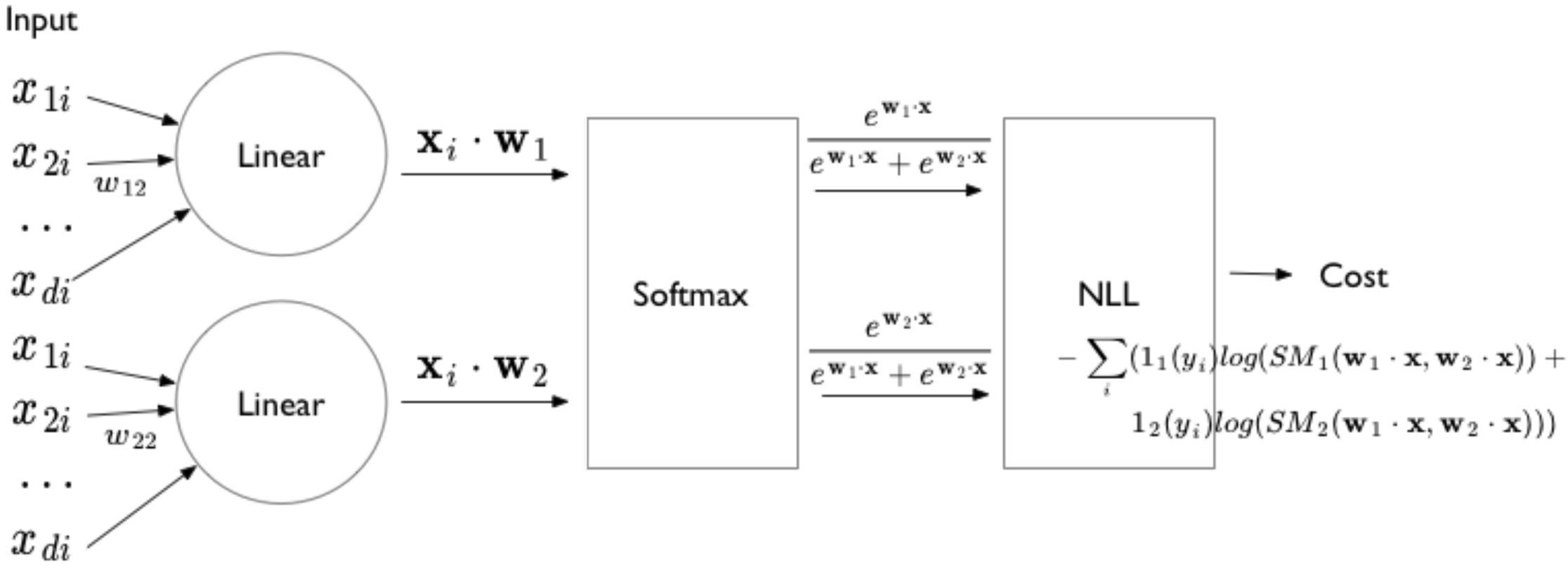
- Identify p_i and $1 - p_i$ as two separate probabilities constrained to add to 1. That is $p_{1i} = p_i ; p_{2i} = 1 - p_i$.
- $p_{1i} = \frac{e^{\mathbf{w}_1 \cdot \mathbf{x}}}{e^{\mathbf{w}_1 \cdot \mathbf{x}} + e^{\mathbf{w}_2 \cdot \mathbf{x}}}$
- $p_{2i} = \frac{e^{\mathbf{w}_2 \cdot \mathbf{x}}}{e^{\mathbf{w}_1 \cdot \mathbf{x}} + e^{\mathbf{w}_2 \cdot \mathbf{x}}}$
- Can translate coefficients by fixed amount ψ without any change

NLL for Softmax

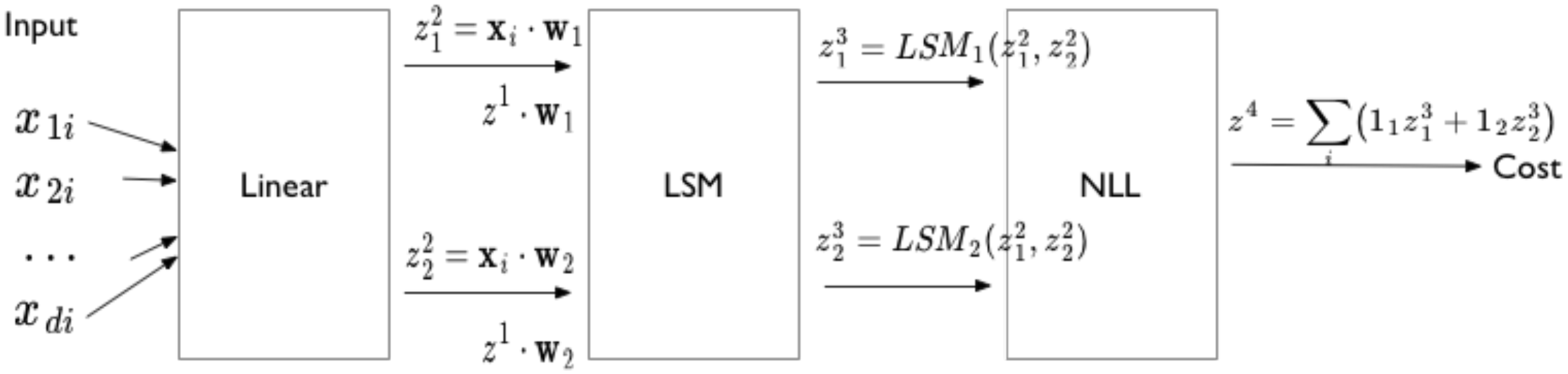
$$\mathcal{L} = \prod_i p_{1i}^{1_1(y_i)} p_{2i}^{1_2(y_i)}$$

$$NLL = -\sum_i (1_1(y_i) log(p_{1i}) + 1_2(y_i) log(p_{2i}))$$

Units diagram for Softmax



Write it in Layers



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

Equations, layer by layer

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

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

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

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

Dude, where is my network?

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

Simply replace the linear regression here by a neural network.

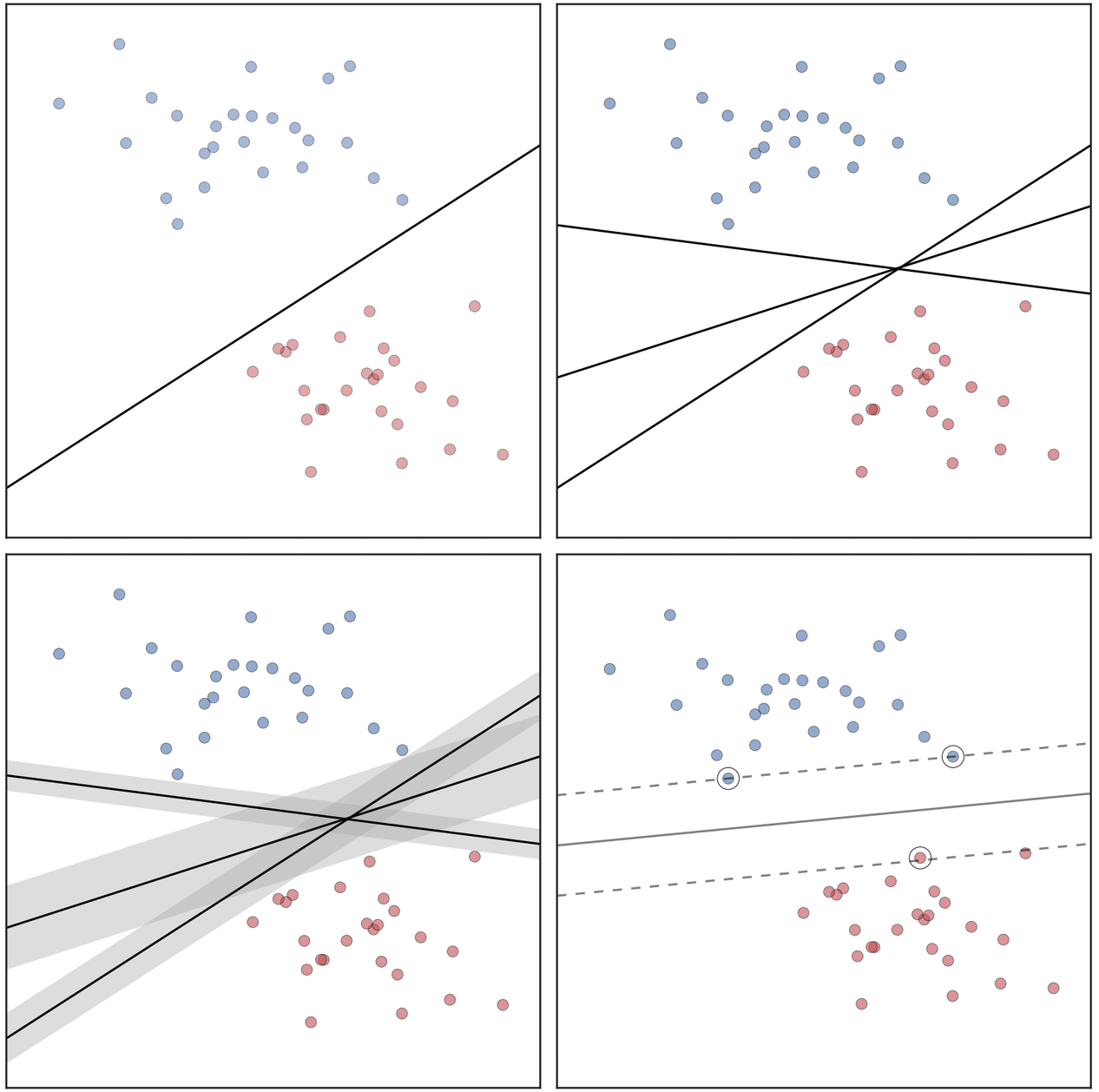
Now find gradients for SGD.

CLASSIFICATION

BY LINEAR SEPARATION

Which line?

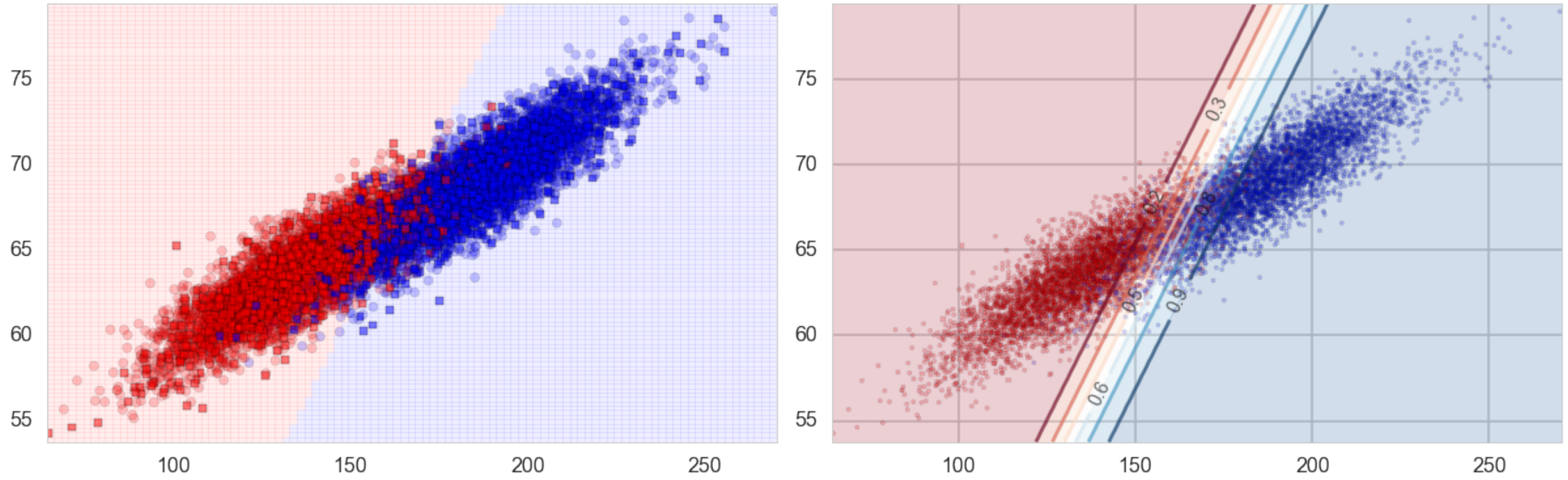
- Different Algorithms, different lines.
- SVM uses max-margin[j]



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

DISCRIMINATIVE CLASSIFIER

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

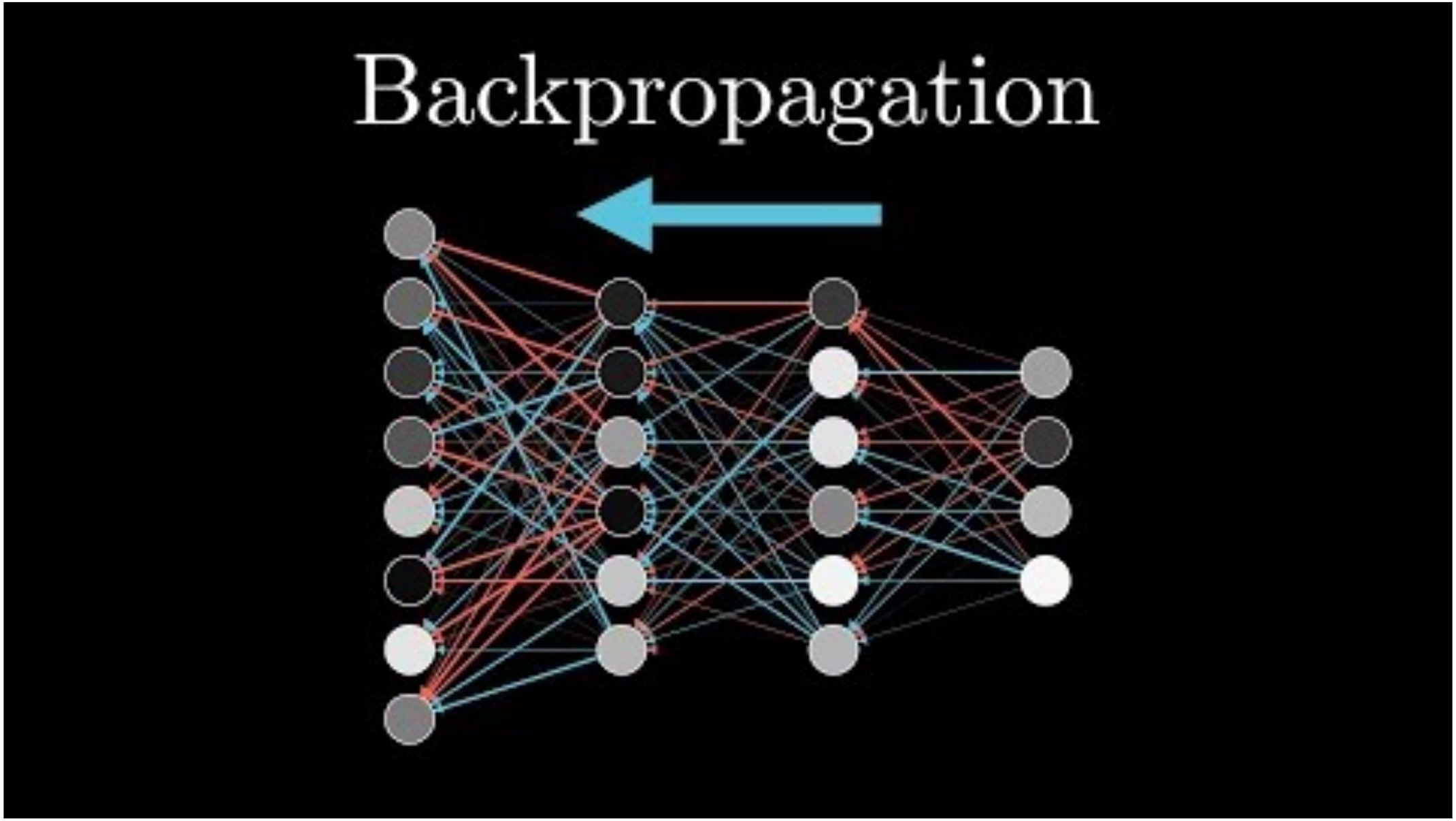


Discriminative Learning

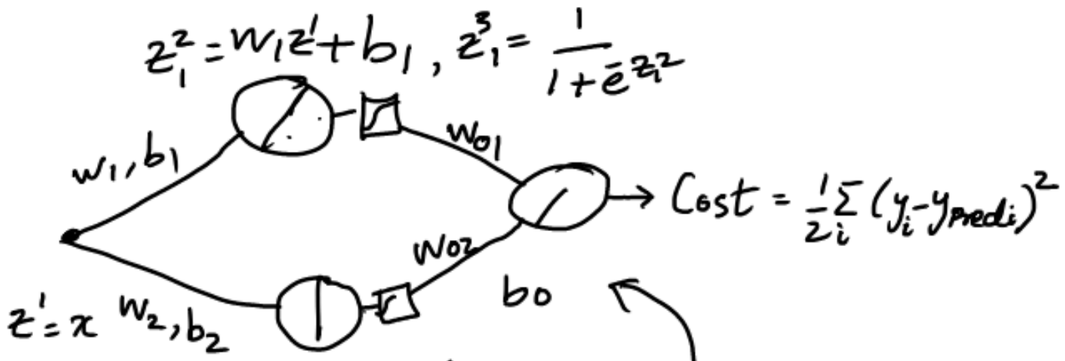
- are these classifiers any good?
- they are discriminative and draw boundaries, but thats it
- could it be better to have a classifier that captured the generative process of the data?
- Thats Ai 2

Calculating Gradients for learning: Backprop (Intuition)

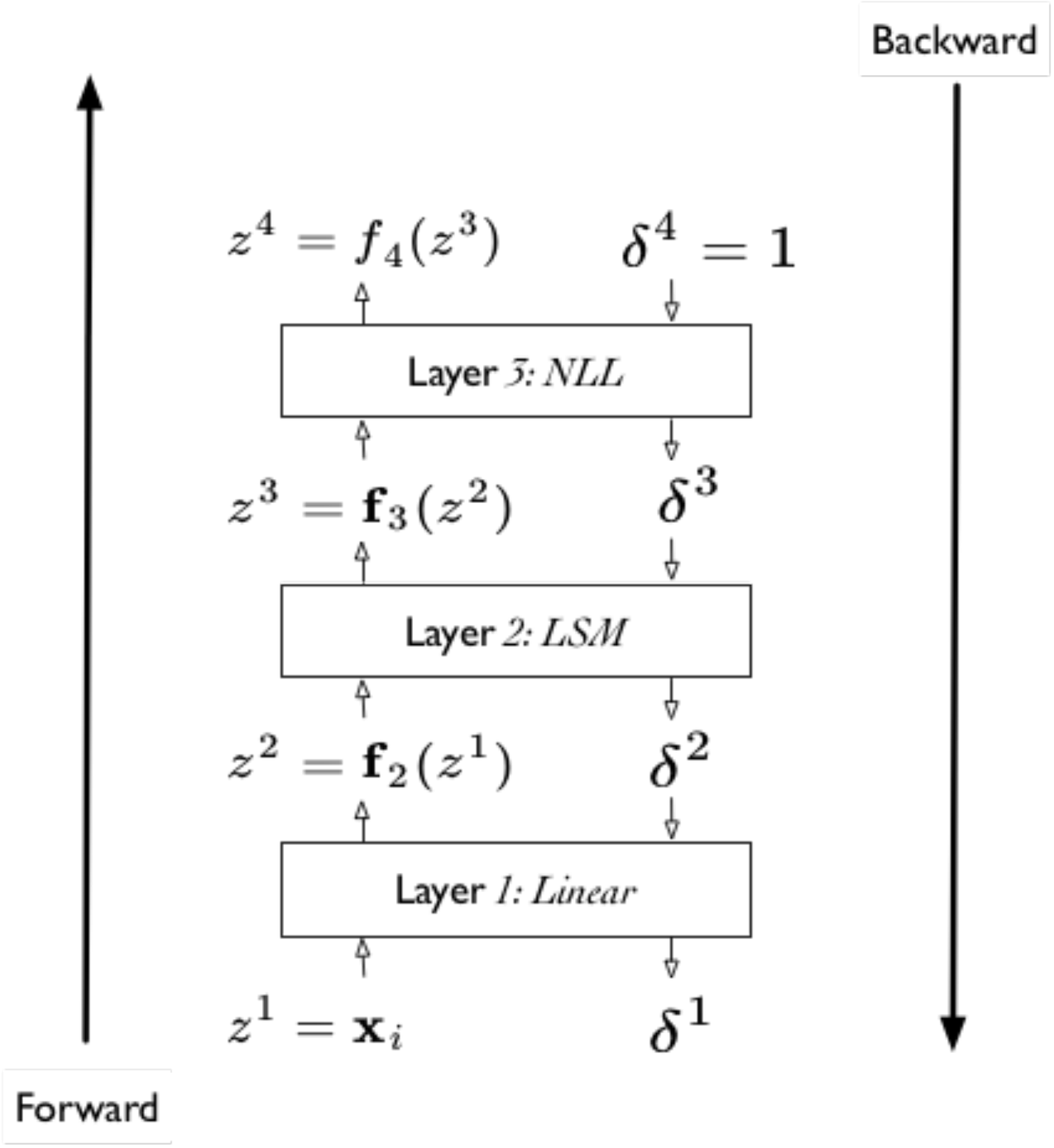
Backpropagation



Simple MLP for regression



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


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

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 wi
ll give us some Cost (z^5, z^6).

$z^5 = \frac{1}{2}(z^4 - y)^2$ $z^4 = w_{o1} z^3_1 + w_{o2} z^3_2 + b_o$ $z^3_1 = 1/(1+e^{-z^2_1})$, $z^3_2 = 1/(1+e^{-z^2_2})$ $z^2_1 = w_1 z^1 + b_1$, $z^2_2 = w_2 z^1 + b^2$, $z_1 = x = 0.1$

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.

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

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

Losses in Neural Nets

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	GANS

How to fit?

Our recipe has been:

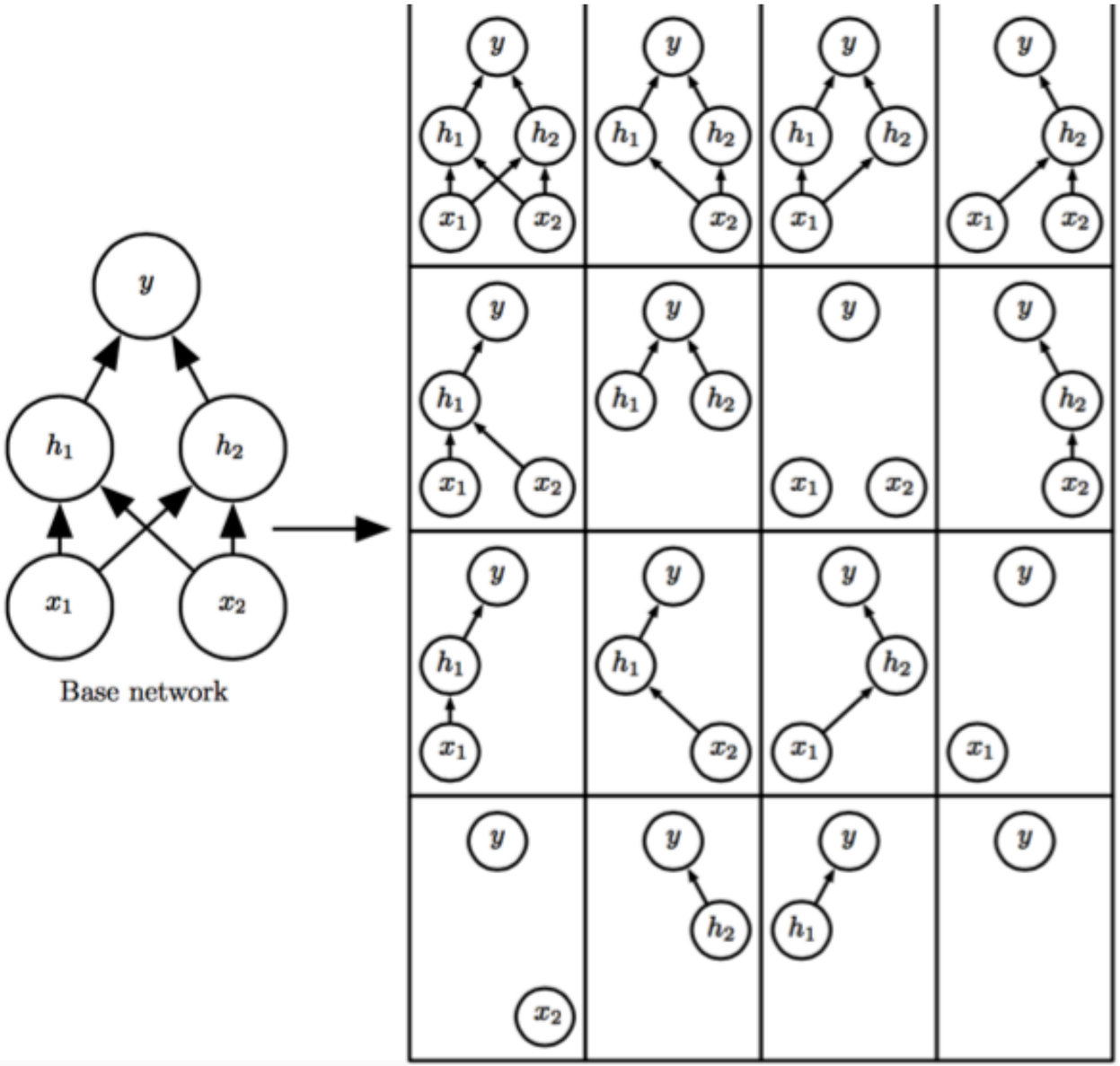
- Train a neural network until it overfits.
- Then add "regularization"

But what is **Regularization**.

Its a way to make parameters behave, to constrain them.

Many styles of regularization

- Dropout
- L2/L1 Regularization: kernel regularization in Keras (also called weight decay)
- Early Stopping
- Simply add more data
- Data Augmentation

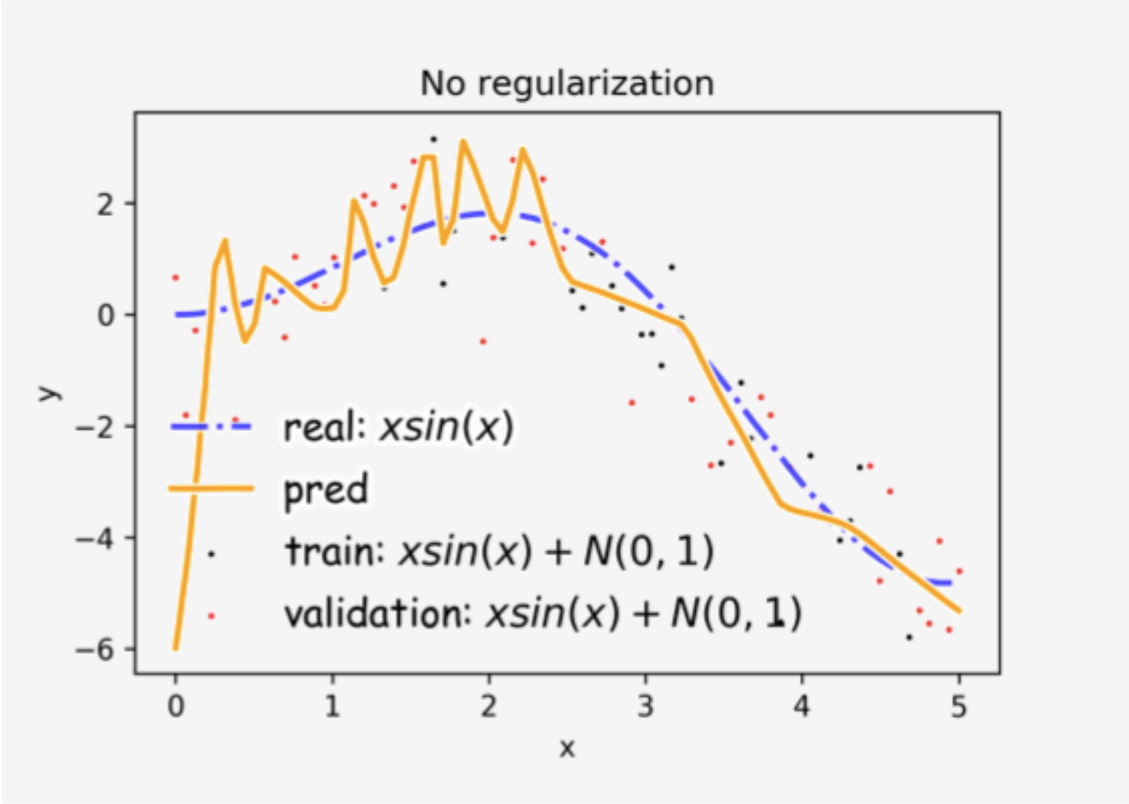


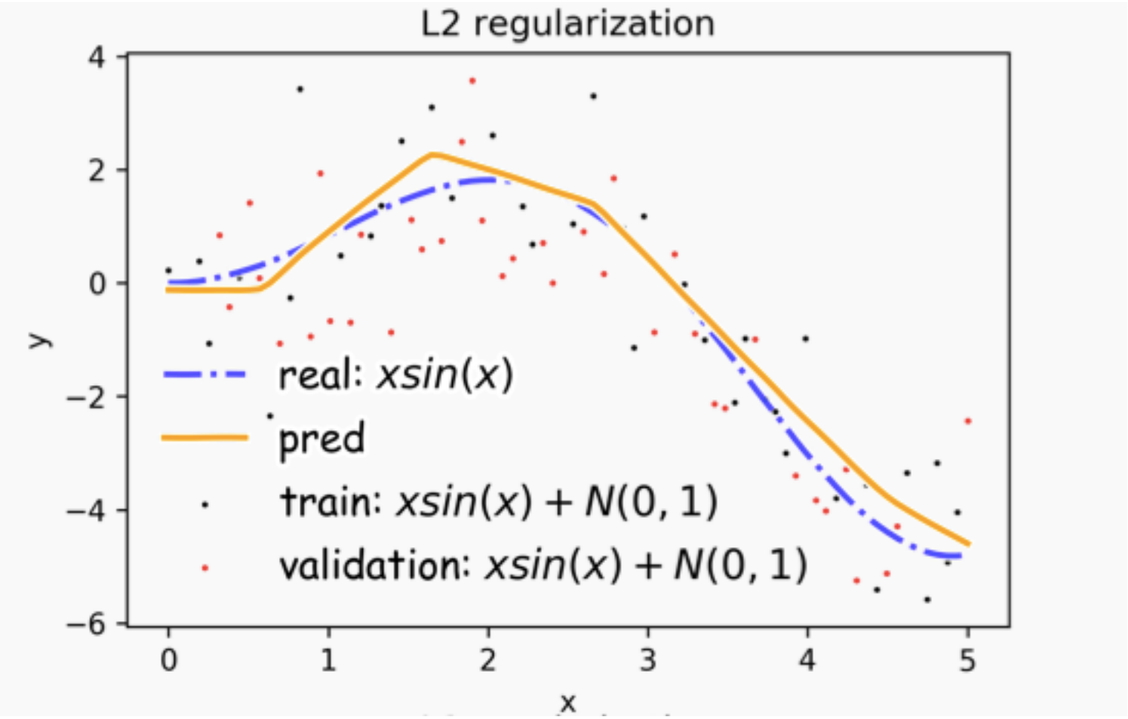
Dropout

Dropout is a kind of pseudo layer.

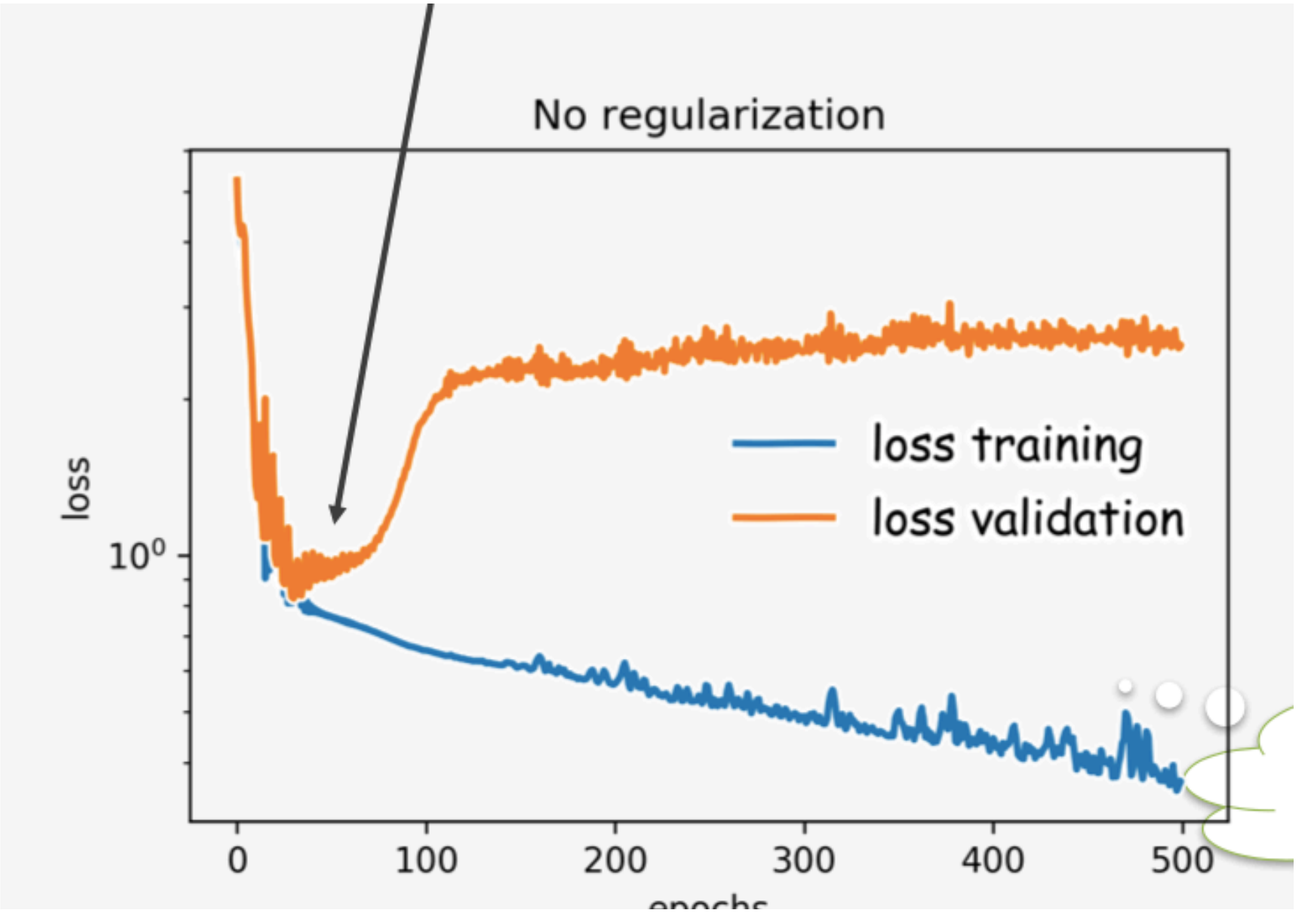
- choose a probability p
- randomly sever p of the connections from the previous layer in the training phase
- this forces other connections to take up the slack and prevents them from over-specializing
- in the testing phase multiply the learned weights with this probability but dont sever connections

L2 Norm


No regularization
real: $xsin(x)$
pred
train: $xsin(x) + N(0, 1)$
validation: $xsin(x) + N(0, 1)$
x
y


L2 regularization
real: $xsin(x)$
pred
train: $xsin(x) + N(0, 1)$
validation: $xsin(x) + N(0, 1)$
x
y

Early Stopping



Optimization

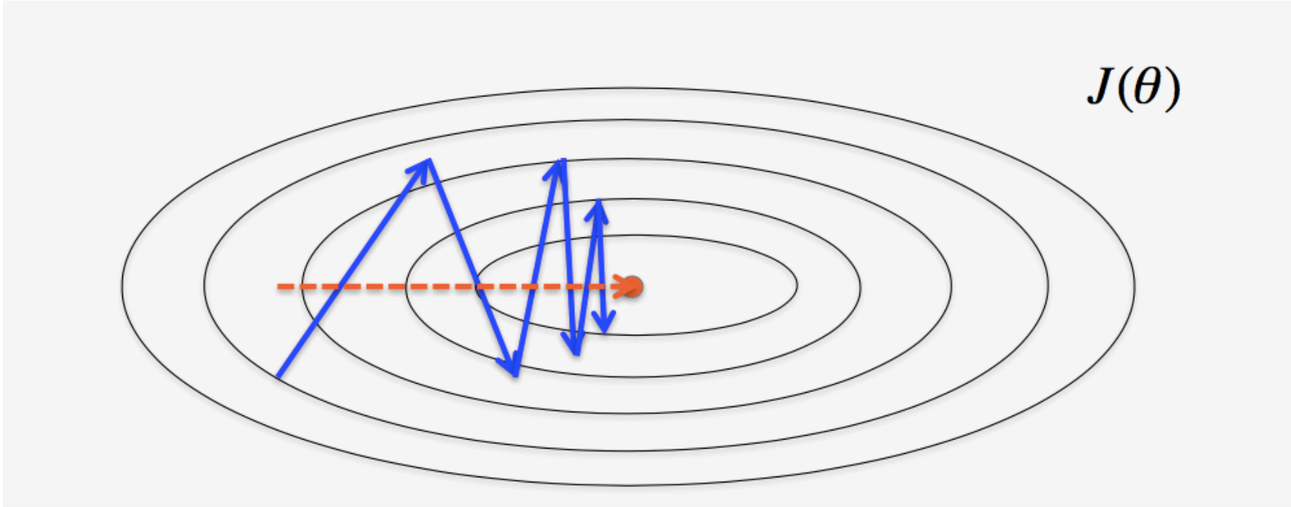
The workhorse: SGD

- has one parameter, the learning rate
- the learning rate is how much of the gradient to use in your parameter updates
- the more you use, the faster you can train, and thus you dont get stuck in local minima
- but if you keep it large you bounce around stochastically

Messing with SGD

- it would be better to have a learning rate schedule, or adaptive learning rates, perhaps different for each parameter
- SGD oscillates in high curvature areas
- averaging the gradient: $< g >$ over a few steps helps as it eliminates oscillations
- momentum helps as well to keep you going straight down:

$$v_{t+1} = \alpha v_t - \eta < g >_t, w_{t+1} = w_t + v$$



Other Variants

- **Adagrad**: $r_{t+1,j} = r_{t,j} + < g >_{tj}^2, w_{t+1} = w_t - \eta \frac{< g >_{tj}}{(\delta + \sqrt{r_{tj}})}$ has greater progress along gently sloped directions
- **RMSprop** uses exponentially weighted average for gradient accumulation. Change to Adagrad: $r_{t+1,j} = \rho r_{t,j} + (1 - \rho) < g >_{tj}^2$
- **Adam** adds momentum to RMSprop works well in practice:

$$r_{t+1,j} = \rho_r r_{t,j} + (1 - \rho_r) < g >_{tj}^2, v_{t+1,j} = \rho_v v_{t,j} + (1 - \rho_v) < g >_{tj}^2,$$
$$w_{t+1} = w_t - \eta \frac{v_{tj}}{(\delta + \sqrt{r_{tj}})}$$

Other considerations

- standardize your inputs(features) so no one input dominates
- we'll come back to Adam (later)
- parameter inittialization (Xavier vs He) (later)
- Batch normalization (later)