Distributed Supervised Machine Learning

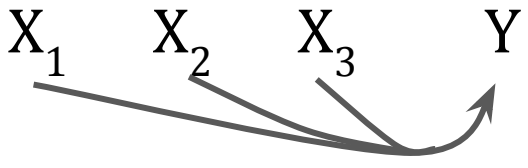
Stony Brook University
CSE545, Spring 2019

Supervised Learning

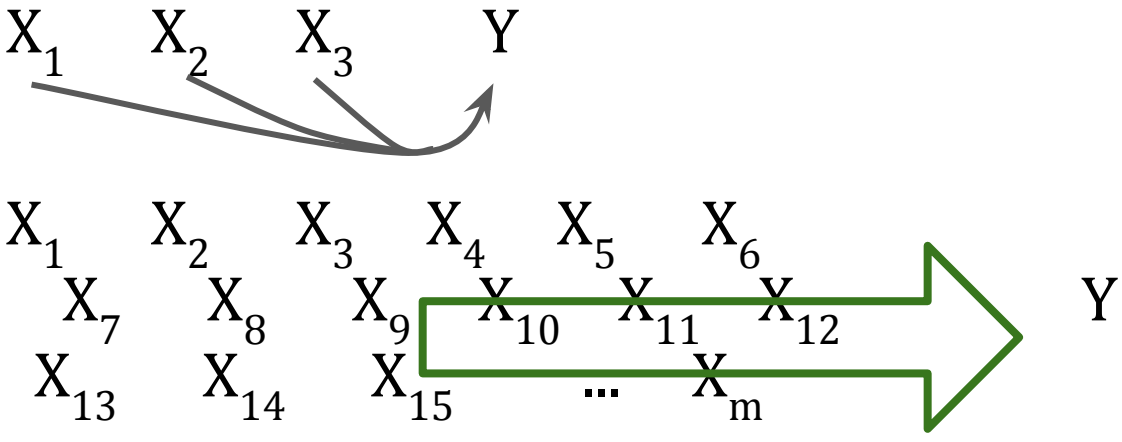
(genes) (health)

X_1 X_2 X_3 Y

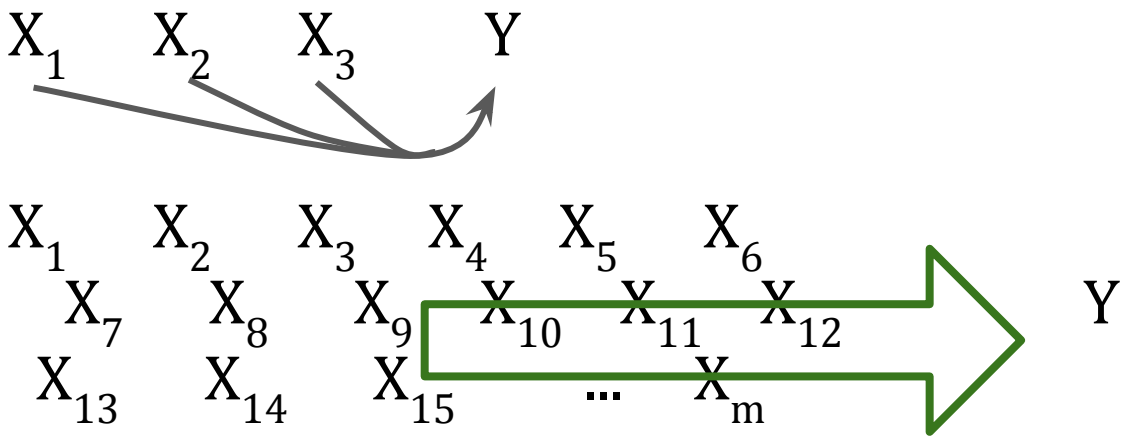
Supervised Learning



Supervised Learning

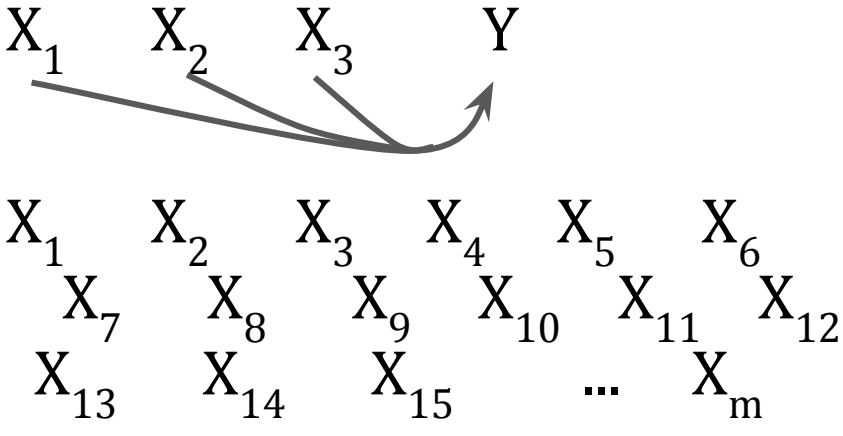


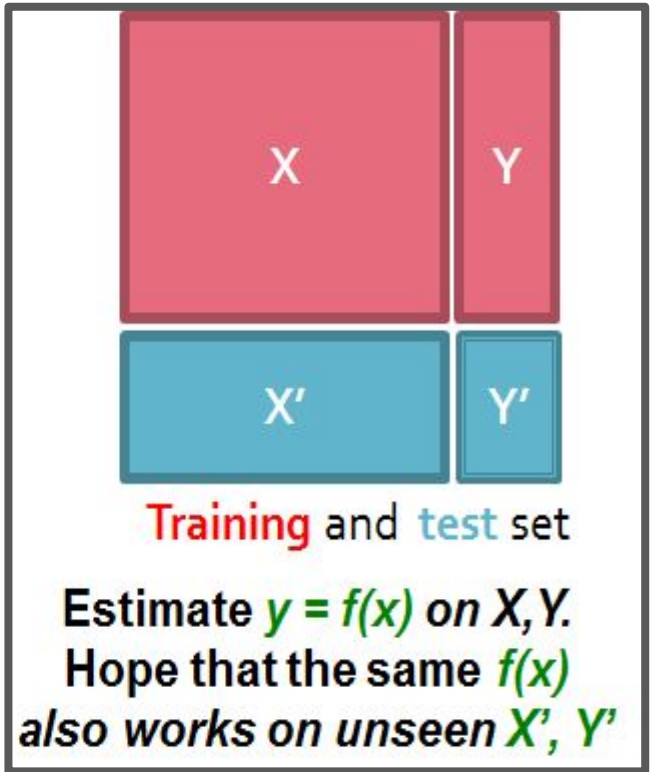
Supervised Learning



Task: Determine a function, f (or parameters to a function) such that $f(X) = Y$

Supervised Learning





J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Task: Determine a function, f (or parameters to a function) such that $f(X) = Y$

Supervised Learning Approaches that we will cover

1. **Regularized linear modeling**
   (linear and logistic regression)

2. Convolutional Neural Networks
   Where X might have spatial relationships

3. Recurrent Neural Networks
   Where X is a sequence of data

Linear Model -- Linear Regression

Finding a linear function based on *X* to best yield *Y.*

X = “covariate” = “feature” = “predictor” = “regressor” = “independent variable”

Y = “response variable” = “outcome” = “dependent variable”

Regression: $r(x) = \mathrm{E}(Y|X = x)$

goal: estimate the function *r*

$$r(x) = \beta_0 + \beta_1 x$$

$$r(x) \approx \mathrm{E}(Y|X = x)$$

Uses of linear and logistic regression

1. Testing the relationship between variables given other variables. β is an "effect size" -- a score for the magnitude of the relationship; can be tested for significance.

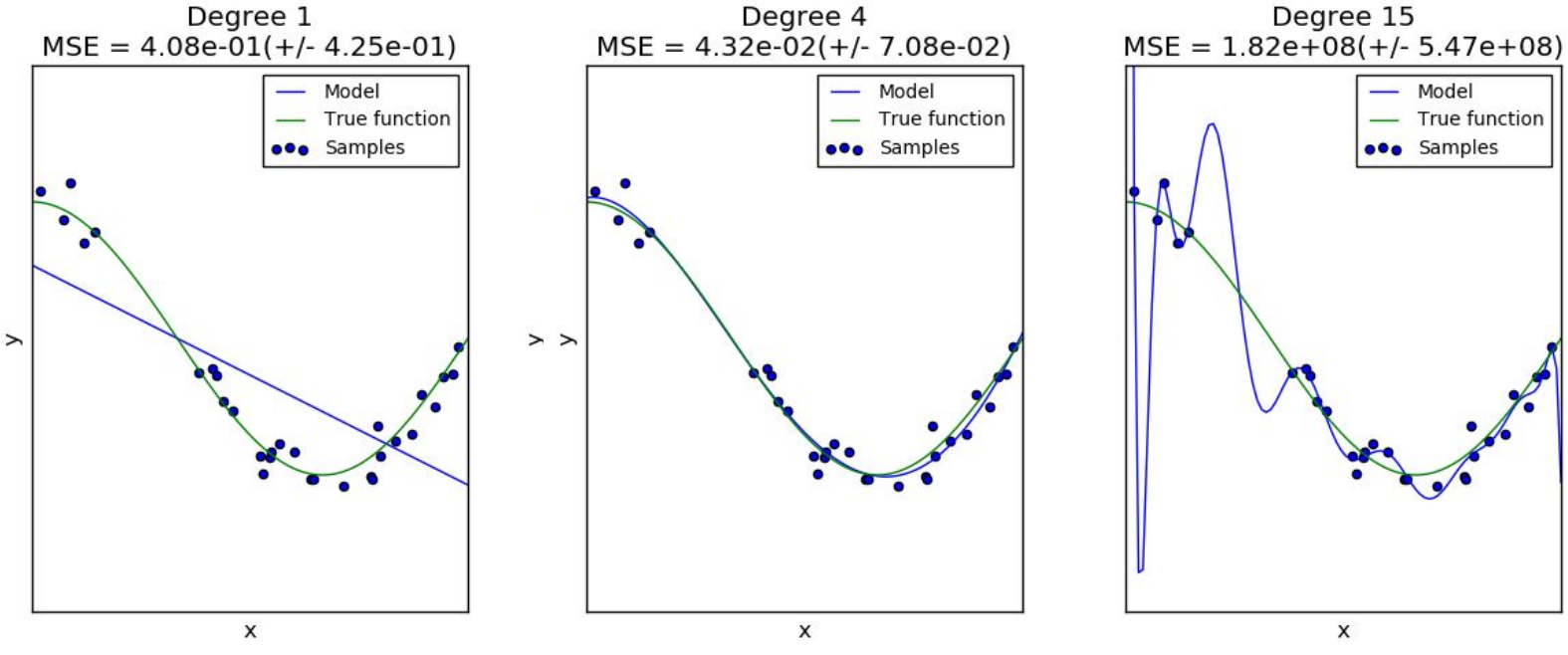
2. Building a predictive model that generalizes to new data. $\hat{Y}$ is an estimate value of Y given X.

Uses of linear and logistic regression

1. Testing the relationship between variables given other variables. β is an "effect size" -- a score for the magnitude of the relationship; can be tested for significance.

2. Building a predictive model that generalizes to new data. $\hat{Y}$ is an estimate value of Y given X.
   However, unless $|X| <<< observatations$ then the model might "overfit".

Overfitting (1-d non-linear example)



Underfit
High Bias

Overfit
High Variance

(image credit: Scikit-learn; in practice data are rarely this clear)

Overfitting (6-d linear example)

Y = X

1	0.5	0	0.6	1	0	0.25
1	0	0.5	0.3	0	0	0
0	0	0	1	1	1	0.5
0	0	0	0	0	1	1
1	0.25	1	1.25	1	0.1	2

Overfitting (5-d linear example)

Y =	X					
1	0.5	0	0.6	1	0	0.25
1	0	0.5	0.3	0	0	0
0	0	0	1	1	1	0.5
0	0	0	0	0	1	1
1	0.25	1	1.25	1	0.1	2

$$logit(Y) = 1.2 + -63*X_1 + 179*X_2 + 71*X_3 + 18*X_4 + -59*X_5 + 19*X_6$$

Overfitting (5-d linear example)

Do we really think we found something generalizable?

Y =	X					
1	0.5	0	0.6	1	0	0.25
1	0	0.5	0.3	0	0	0
0	0	0	1	1	1	0.5
0	0	0	0	0	1	1
1	0.25	1	1.25	1	0.1	2

$$logit(Y) = 1.2 + -63*X_1 + 179*X_2 + 71*X_3 + 18*X_4 + -59*X_5 + 19*X_6$$

Overfitting (2-d linear example)

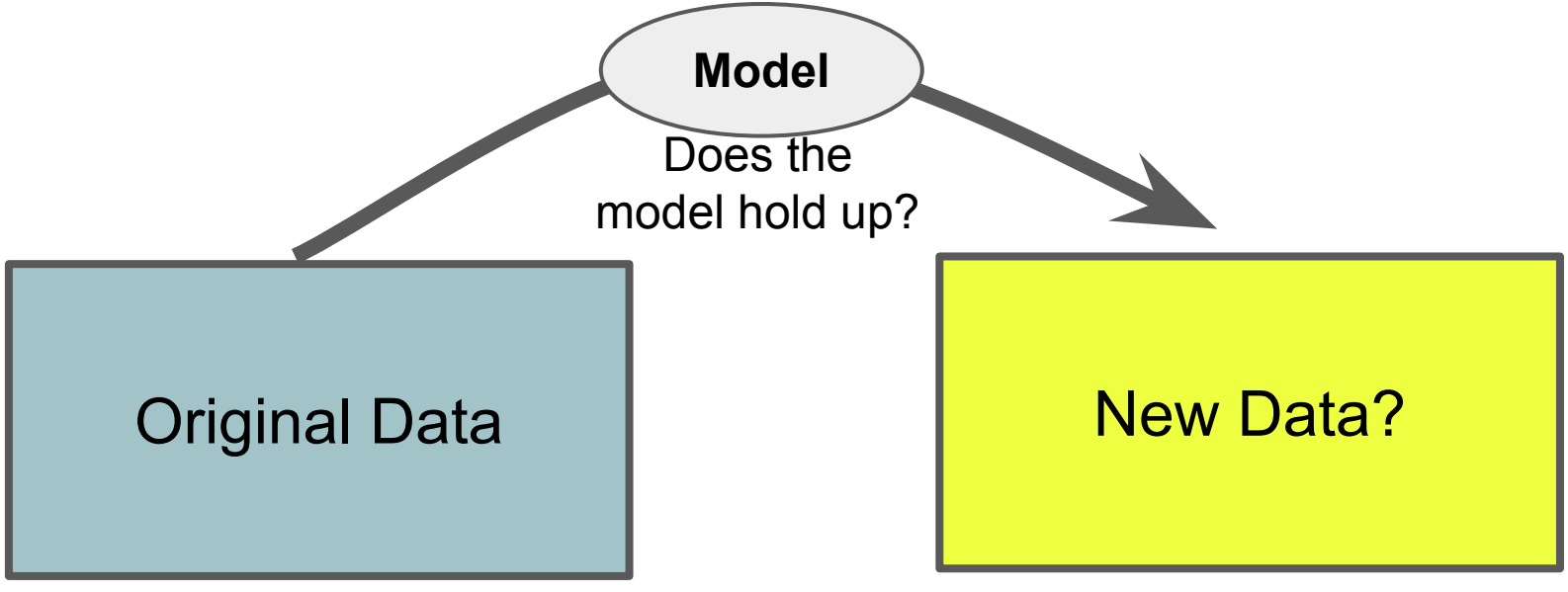
Do we really think we found something generalizable?

Y =	X	
1	0.5	0
1	0	0.5
0	0	0
0	0	0
1	0.25	1

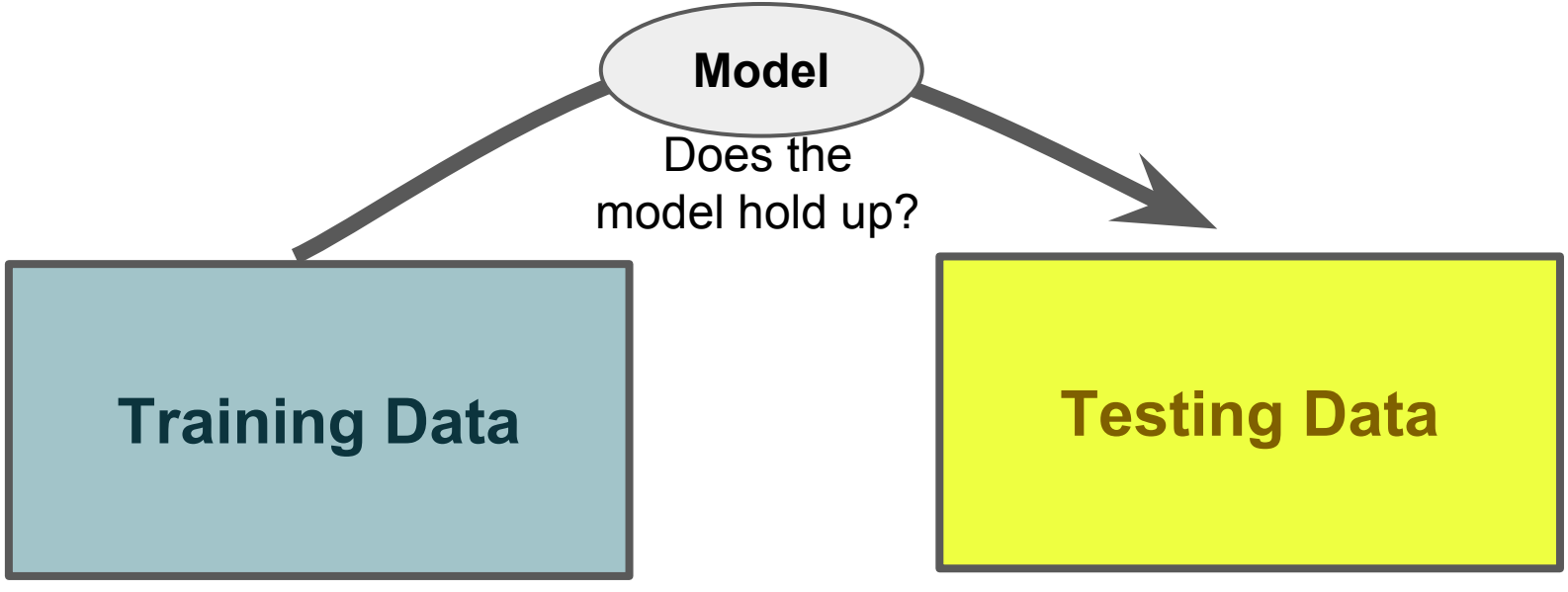
What if only 2 predictors?

$$logit(Y) = 0 + 2*X_1 + 2*X_2$$

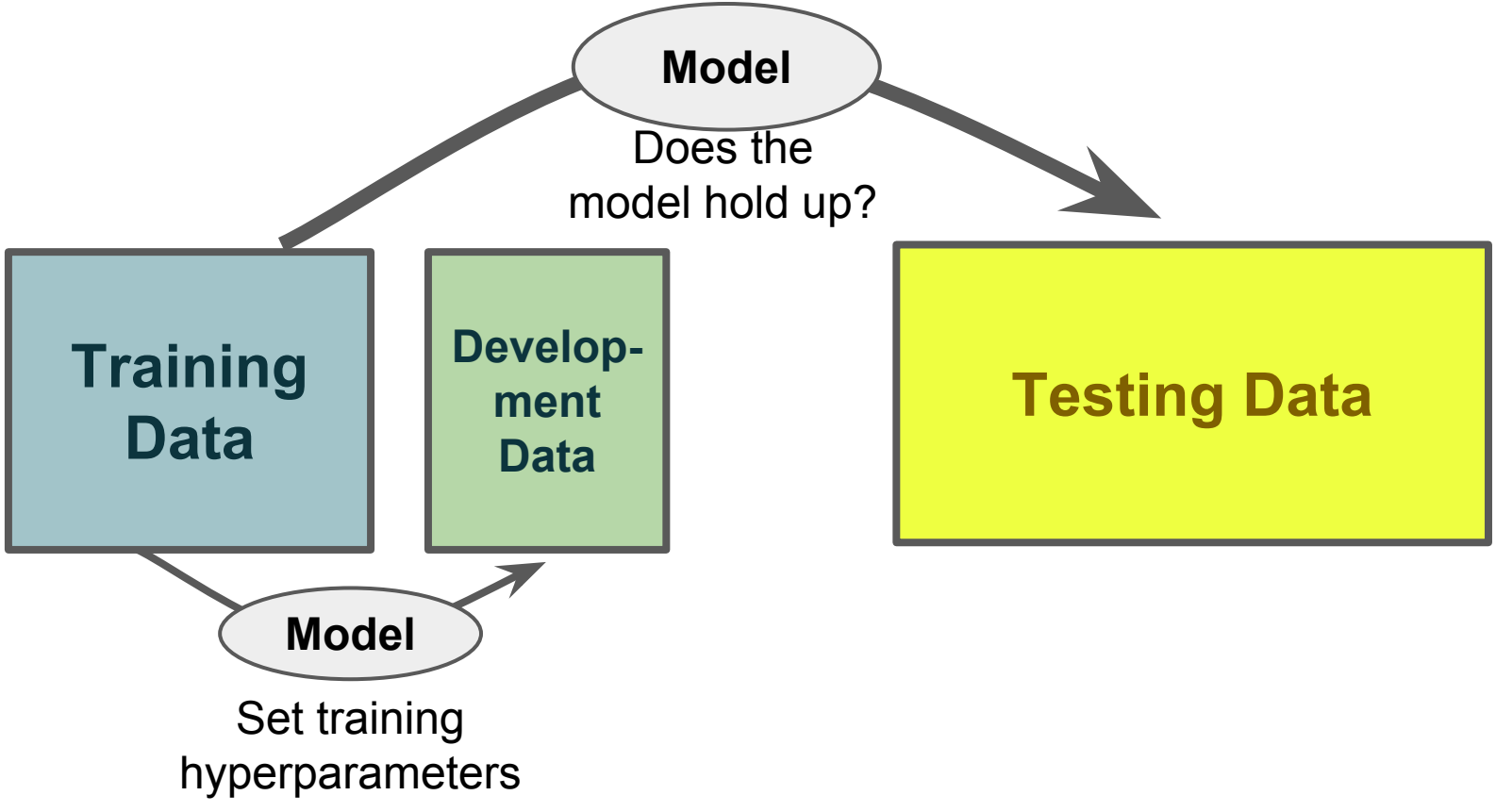
Common Goal: Generalize to new data



Common Goal: Generalize to new data



Common Goal: Generalize to new data



Feature Selection / Subset Selection

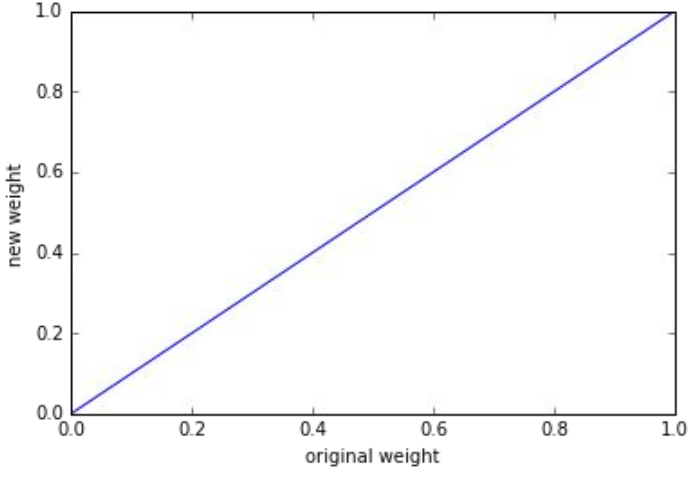
(bad) solution to overfit problem

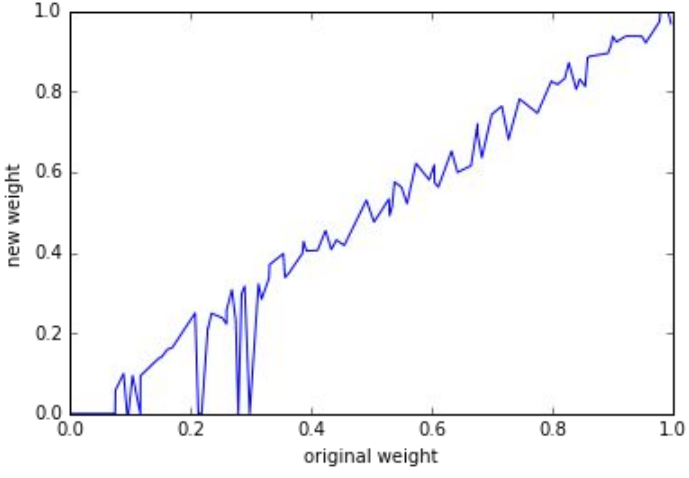
Use less features based on Forward Stepwise Selection:

- start with current_model just has the intercept (mean)
  remaining_predictors = all_predictors

```
for i in range(k):
  #find best p to add to current_model:
  for p in remaining_prepdictors
     refit current_model with p
      #add best p, based on RSS_p to current_model
  #remove p from remaining predictors
```

Regularization (Shrinkage)





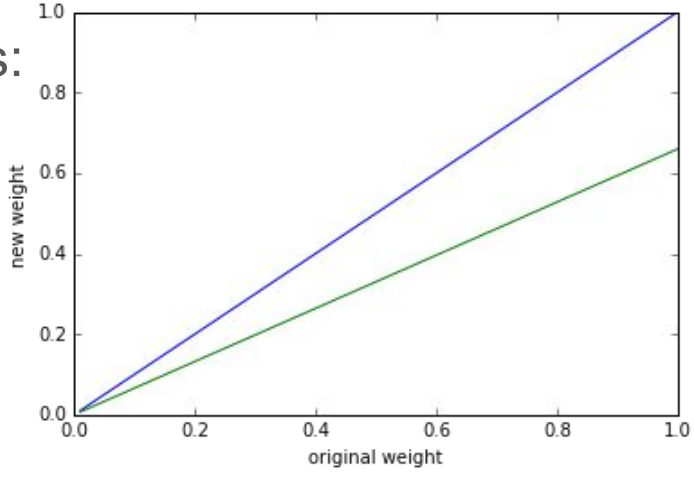
No selection (weight=beta)

forward stepwise

Why just keep or discard features?

Regularization (L2, Ridge Regression)

Idea: Impose a penalty on size of weights:



Ordinary least squares objective:

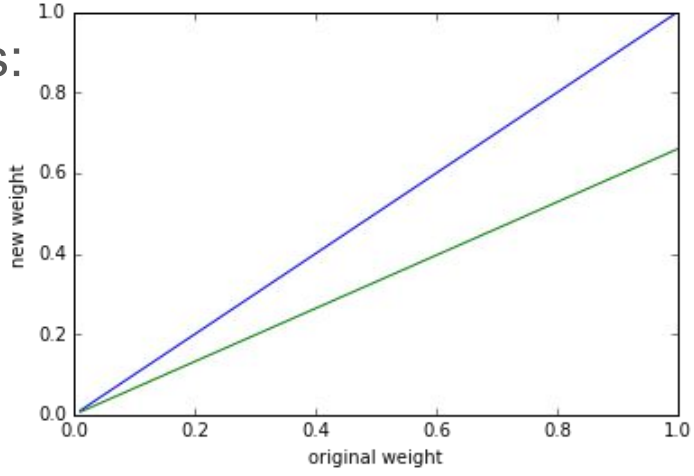
$$\hat{\beta} = argmin_{\beta}\{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2\}$$

Ridge regression:

$$\hat{\beta}^{ridge} = argmin_{\beta}\{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{m} \beta_j^2\}$$

Regularization (L2, Ridge Regression)

Idea: Impose a penalty on size of weights:



Ordinary least squares objective:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2\}$$

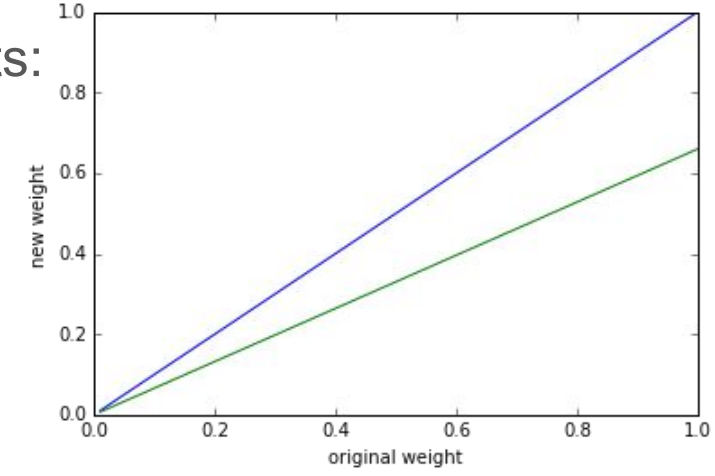
Ridge regression:

$$\hat{\beta}^{ridge} = argmin_{\beta}\{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{m} \beta_j^2\}$$

$$\lambda ||\beta||_2^2$$

Regularization (L2, Ridge Regression)

Idea: Impose a penalty on size of weights:



Ordinary least squares objective:

$$\hat{\beta} = argmin_{\beta}\{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2\}$$

Ridge regression:

$$\hat{\beta}^{ridge} = argmin_{\beta}\{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{m}\beta_j^2\}$$

In Matrix Form:

$$\mathrm{RSS}(\lambda) = (y - X\beta)^T(y - X\beta) + \lambda\beta^T\beta$$

$$\hat{\beta}^{ridge} = (X^TX + \lambda I)^{-1}X^Ty$$

I: m x m identity matrix

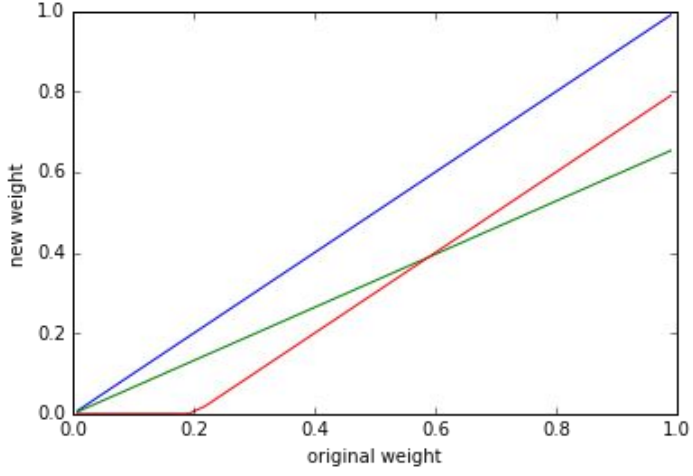
$\lambda||\beta||_2^2$

Regularization (L1, The “Lasso”)

Idea: Impose a penalty and zero-out some weights

The Lasso Objective:

$$\hat{\beta}^{lasso} = argmin_{\beta}\{\frac{1}{2}\sum_{i=1}^{N}(Y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{m} |\beta_j|\}$$



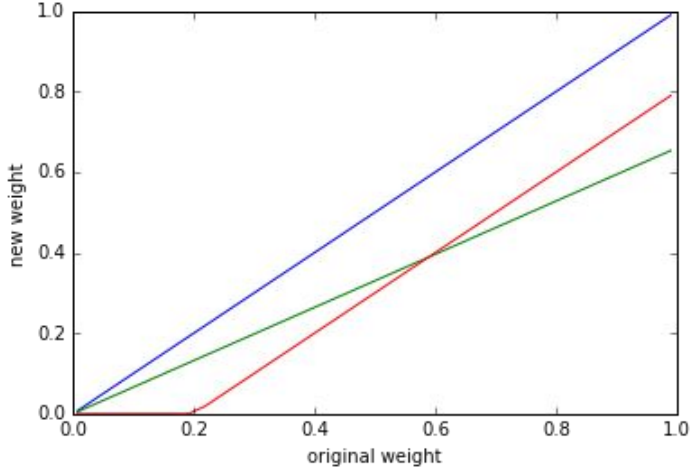
$$\lambda||\beta||_1$$

Regularization (L1, The “Lasso”)

Idea: Impose a penalty and zero-out some weights

The Lasso Objective:

$$\hat{\beta}^{lasso} = argmin_{\beta}\{\frac{1}{2}\sum_{i=1}^{N}(Y_i - \sum_{j=1}^{m} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{m} |\beta_j|\}$$

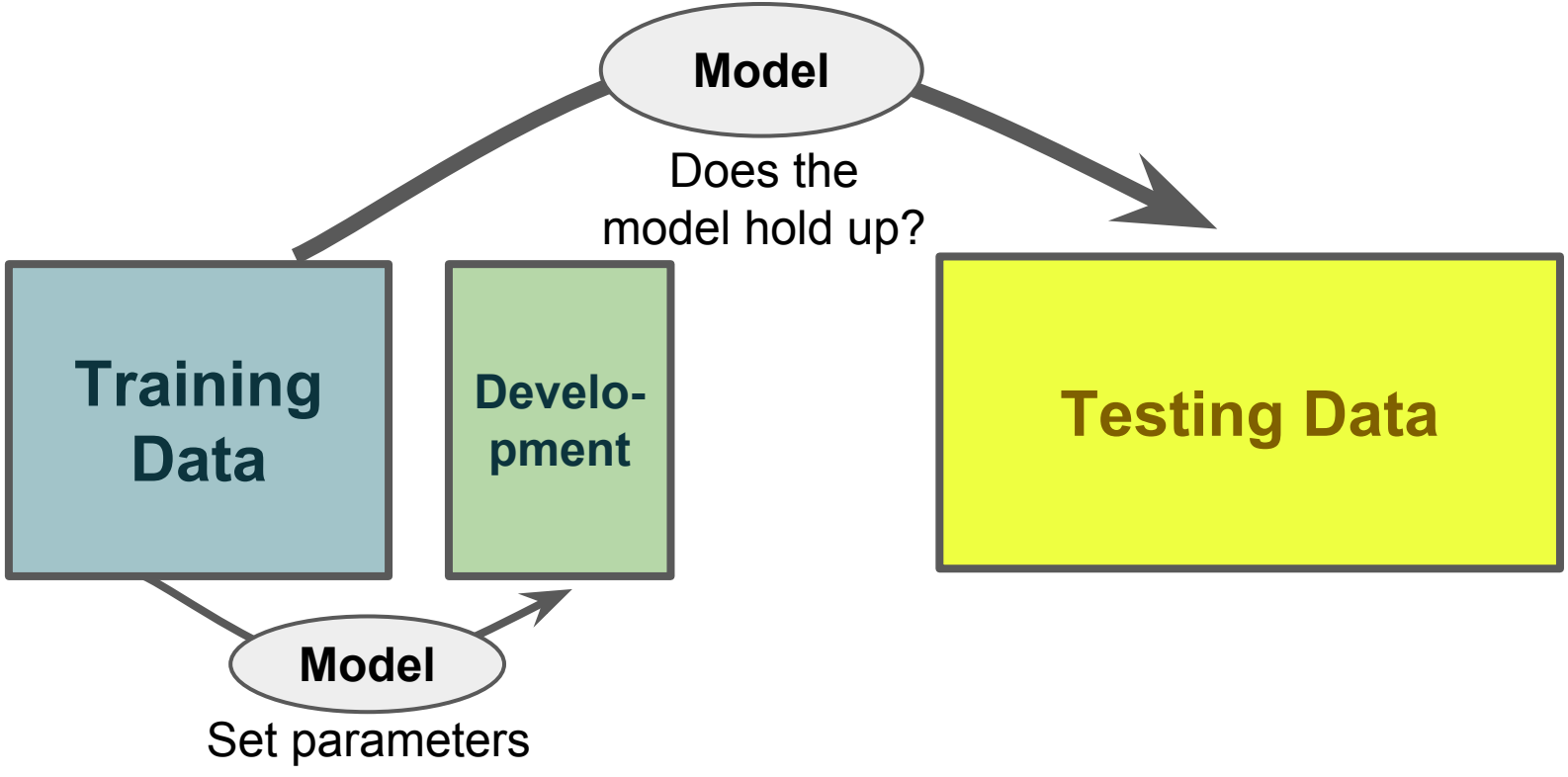


No closed form matrix solution, but often solved with coordinate descent.

$$\lambda||\beta||_1$$

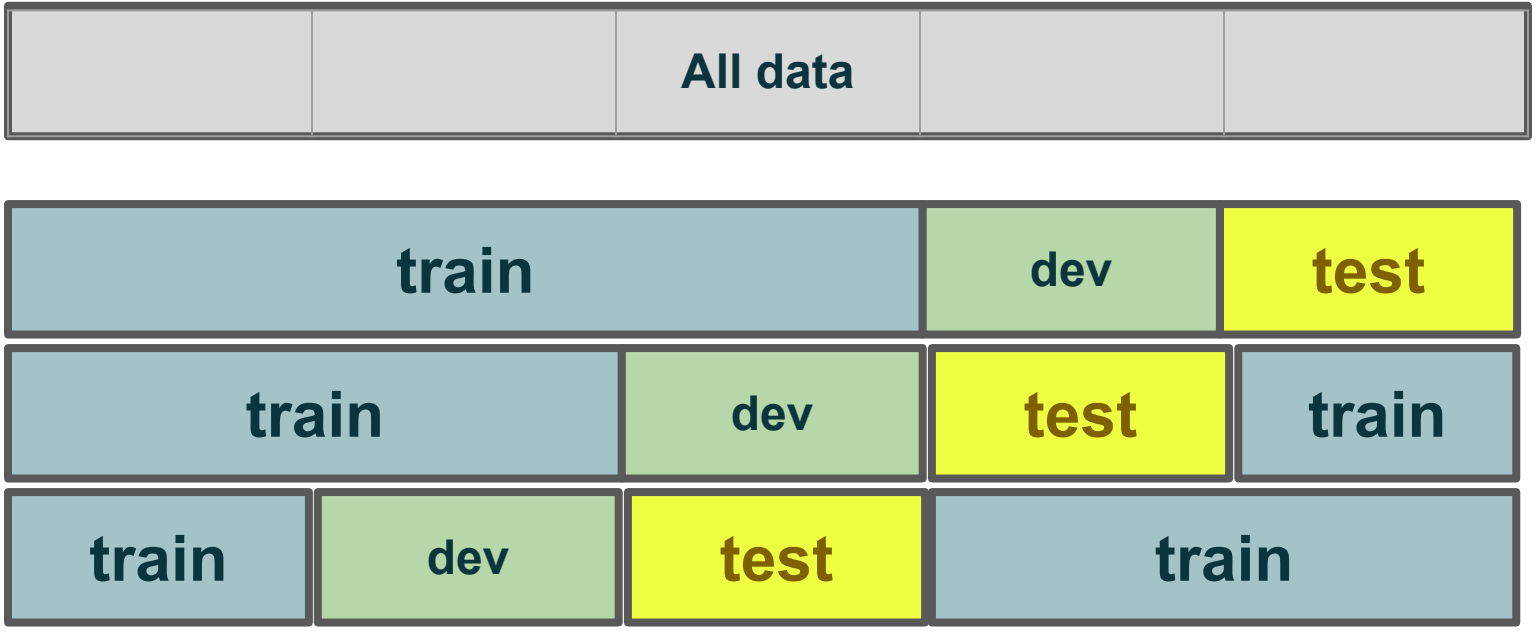
Application: p ≅ n or p >> n (p: features; n: observations)

Common Goal: Generalize to new data



N-Fold Cross-Validation

Goal: Decent estimate of model accuracy



Options for Distributing ML

1. **Distribute copies of entire dataset**
    a. Train over all with different hyperparameters
    b. Train different folds per worker node.

Options for Distributing ML

1. **Distribute copies of entire dataset**
    a. Train over all with different parameters
    b. Train different folds per worker node.

2. **Distribute data**
    a. Each node finds parameters for subset of data
    b. Needs mechanism for updating parameters
        i. Centralized parameter server
        ii. Distributed All-Reduce

Options for Distributing ML

1. **Distribute copies of entire dataset**
    a. Train over all with different parameters
    b. Train different folds per worker node.

2. **Distribute data**
    a. Each node finds parameters for subset of data
    b. Needs mechanism for updating parameters
        i. Centralized parameter server
        ii. Distributed All-Reduce

3. **Distribute model or individual operations** (e.g. matrix multiply)

Options for Distributing ML

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

   Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

   Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

   Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Options for Distributing ML

Done very often in practice. Not talked about much because it's mostly as easy as it sounds.

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Options for Distributing ML

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

Done very often in practice. Not talked about much because it's mostly as easy as it sounds.

Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

Preferred method for big data or very complex models (i.e. models with many internal parameters).

Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

Pro: Parameters can be localized Con: High communication for transferring Intermediar data.

Options for Distributing ML

1. **Distribute copies of entire dataset**
   a. Train over all with different parameters
   b. Train different folds per worker node.

Done very often in practice. Not talked about much because it's mostly as easy as it sounds.

Pro: Easy; Good for compute-bound; Con: Requires data fit in worker memories

2. **Distribute data**
   a. Each node finds parameters for subset of data
   b. Needs mechanism for updating parameters
      i. Centralized parameter server
      ii. Distributed All-Reduce

Data Parellelism

Preferred method for big data or very complex models (i.e. models with many internal parameters).

Pro: Flexible to all situations; Con: Optimizing for subset is suboptimal

3. **Distribute model or individual operations** (e.g. matrix multiply)

Model Parellelism

Pro: Parallelism can be maximized

Con: High communication for transferring Intermediar data.

Cluster Distribution

Model Parallelism

Multiple devices on multiple machines

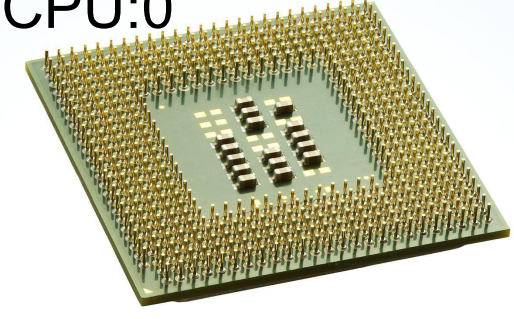
```
with tf.device("/cpu:1")
   beta=tf.Variable(...)
```

```
with tf.device("/gpu:0")
   y_pred=tf.matmul(beta,X)
```

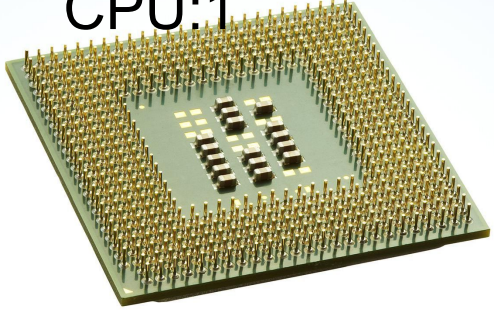
Transfer Tensors

Machine A

CPU:0

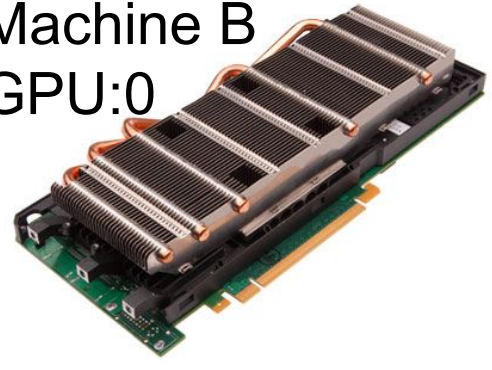


CPU:1



Machine B

GPU:0



Cluster Distribution

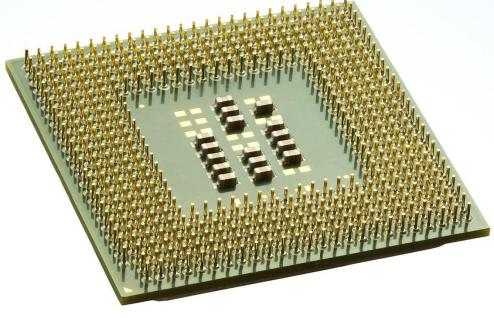
Data Parallelism

```
...
  beta=tf.Variable(...)
  pred=tf.matmul(beta,X)
```

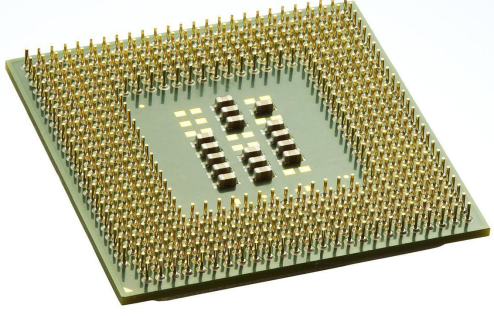
```
...
  beta=tf.Variable(...)
  pred=tf.matmul(beta,X)
```

```
...
  beta=tf.Variable(...)
  pred=tf.matmul(beta,X)
```

CPU:0



CPU:1



GPU:0

Cluster Distribution

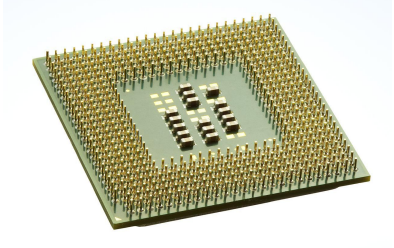
Data Parallelism

```
...
  beta=tf.Variable(...)
  pred=tf.matmul(beta,X)
```

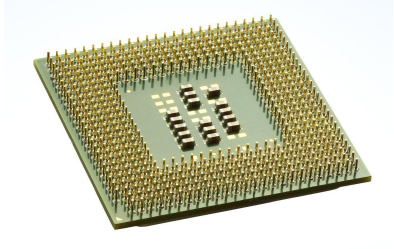
```
...
  beta=tf.Variable(...)
  pred=tf.matmul(beta,X)
```

```
...
  beta=tf.Variable(...)
  pred=tf.matmul(beta,X)
```

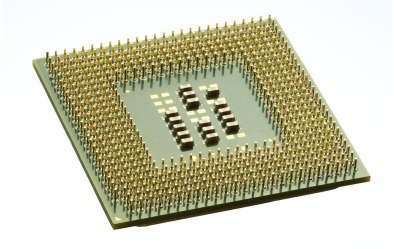
worker:0



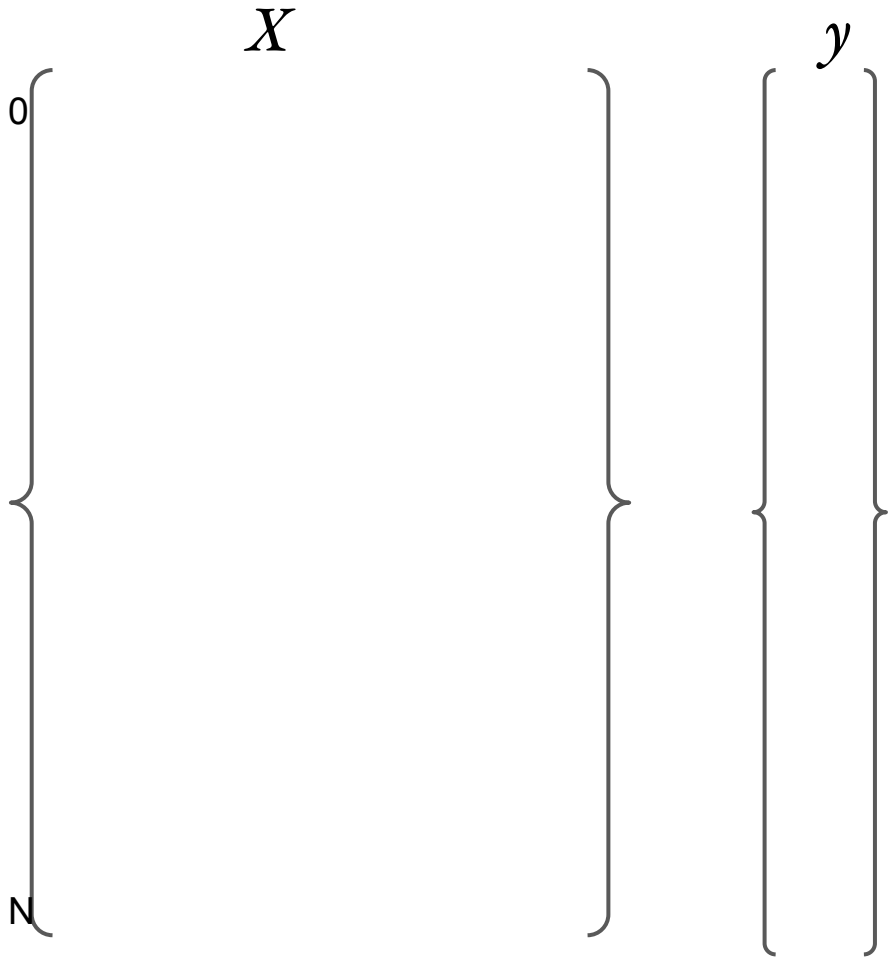
worker:1



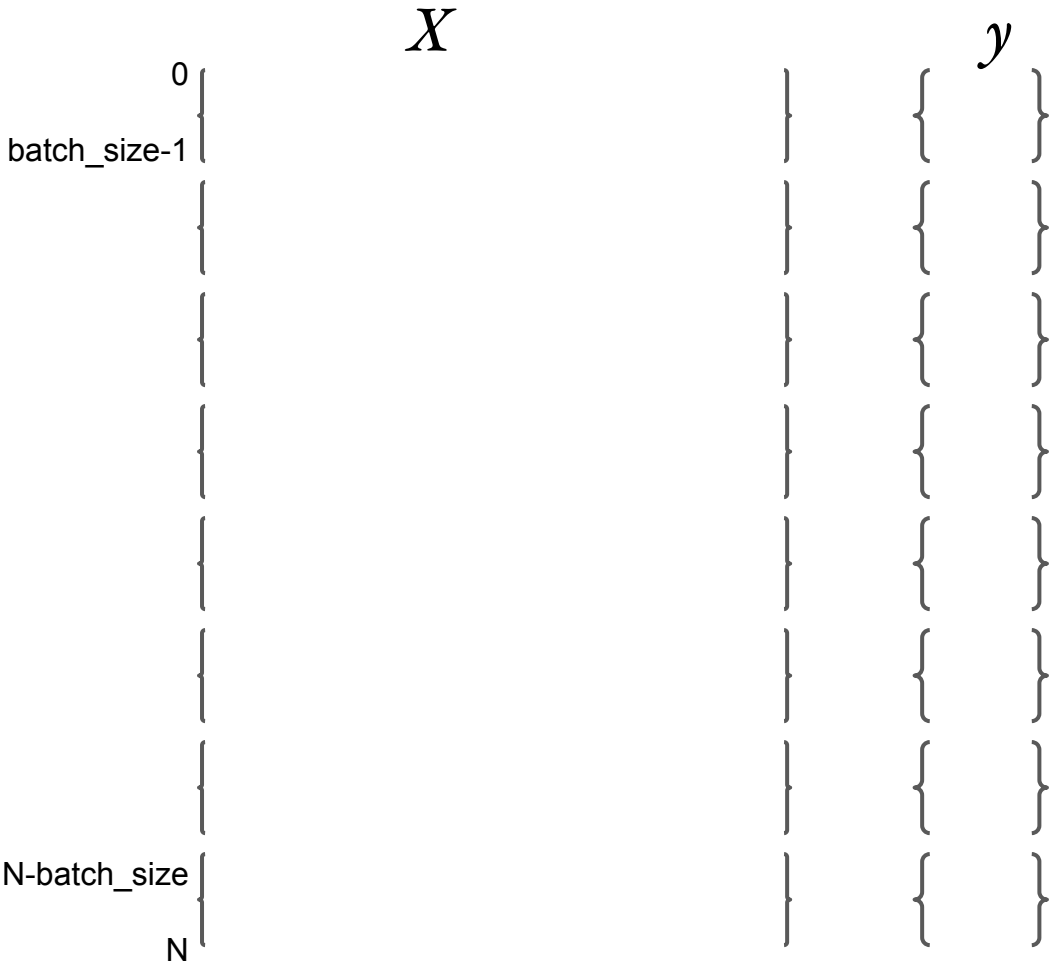
worker:2



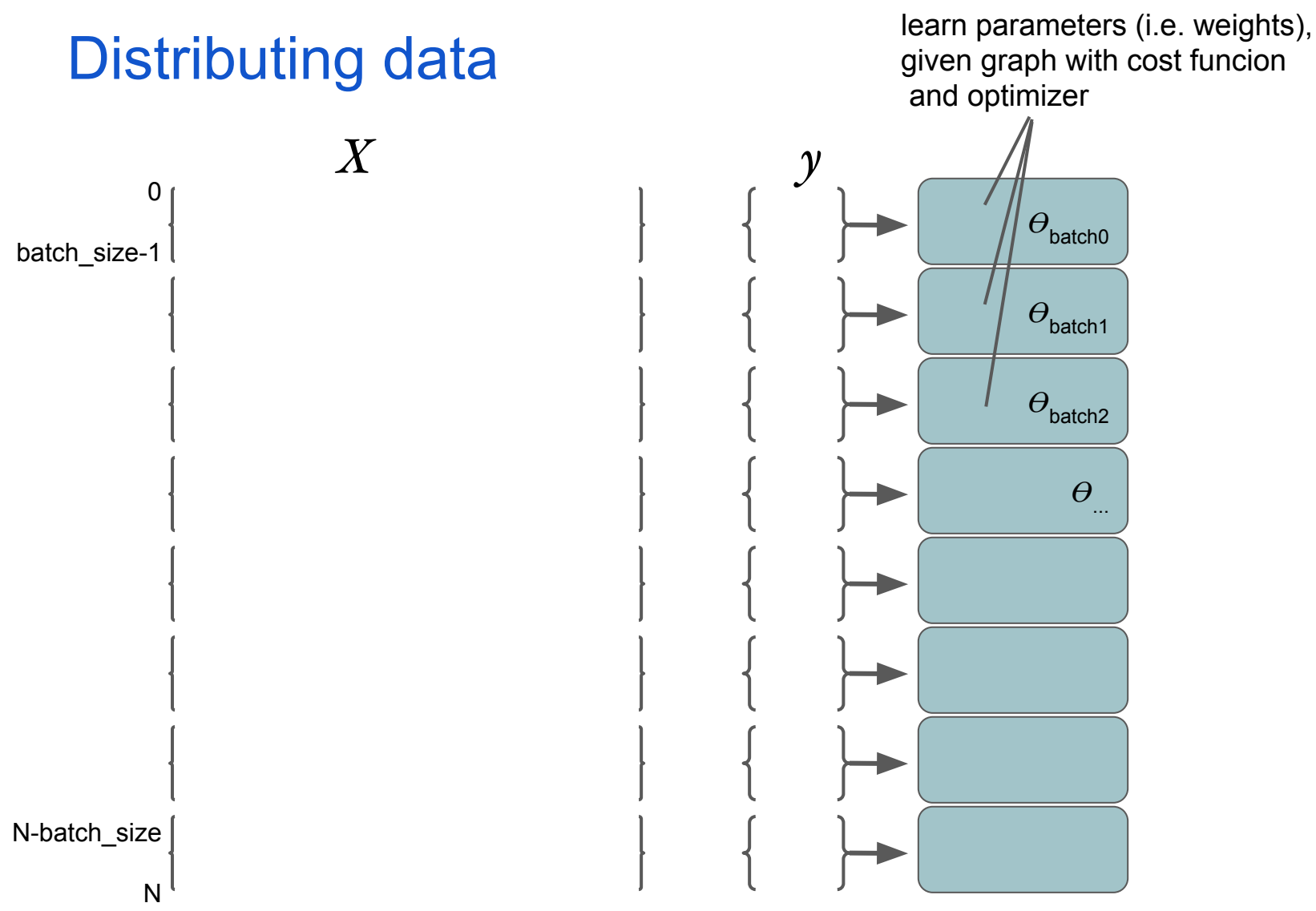
Distributing data



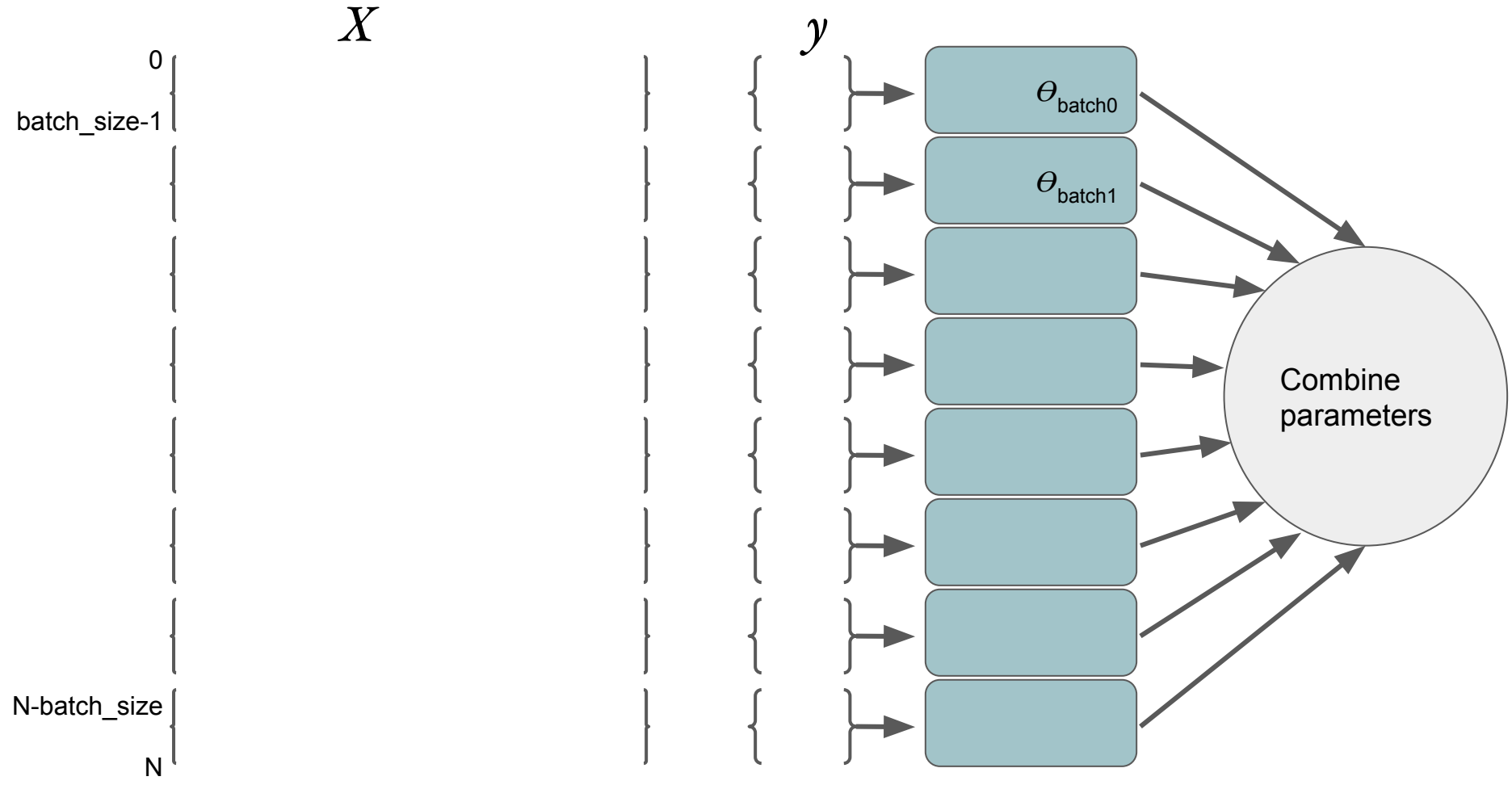
Distributing data

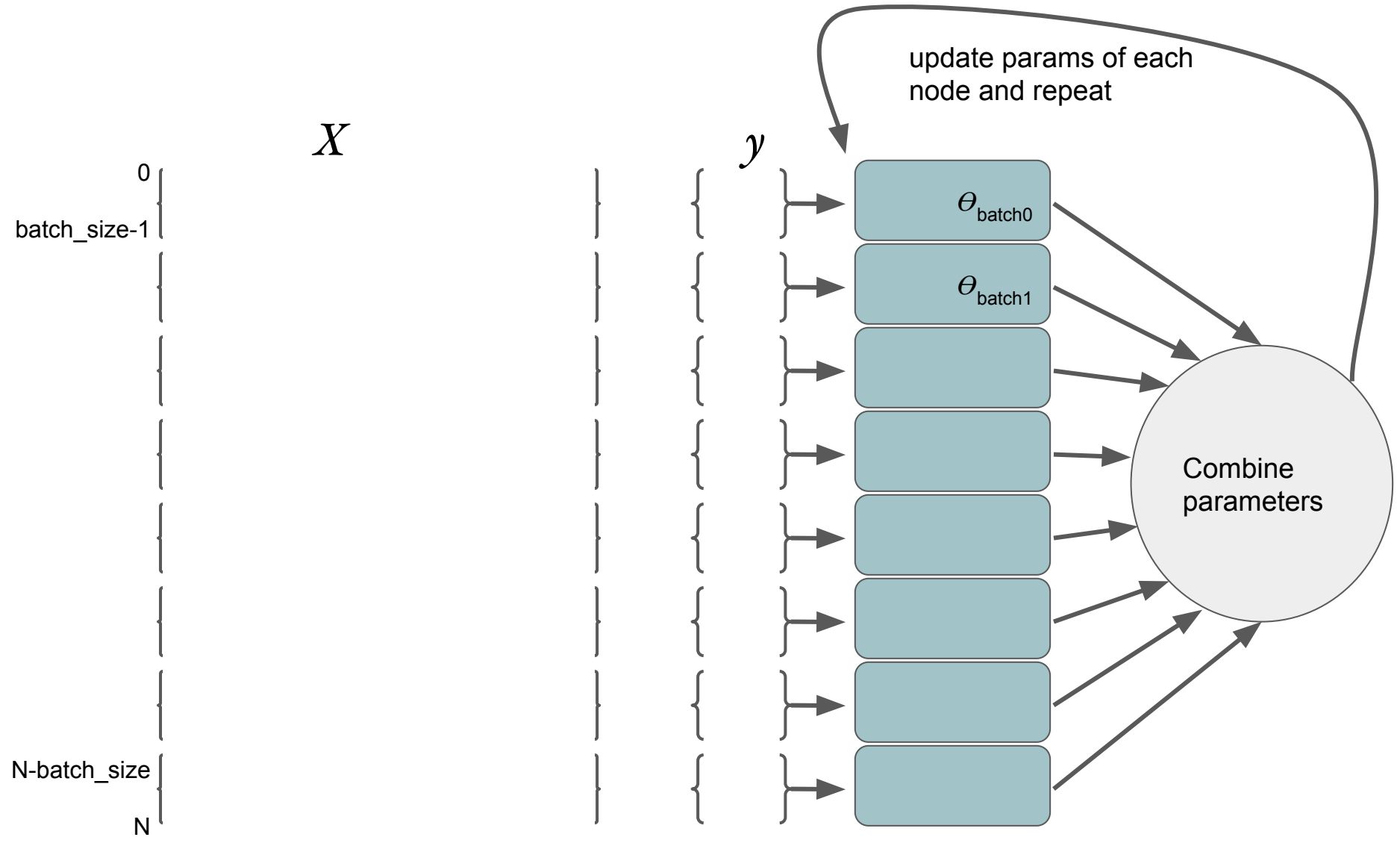


Distributing data

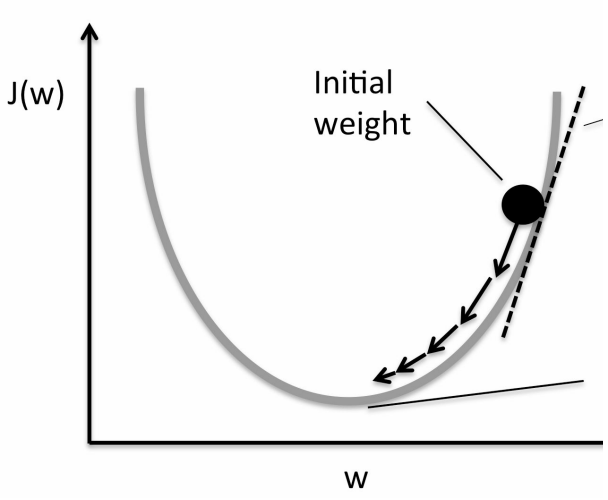


Distributing data




update params of each
node and repeat
X
y
0
batch_size-1
N-batch_size
N
θbatch0
θbatch1
Combine
parameters

Gradient Descent Options for Linear Regression



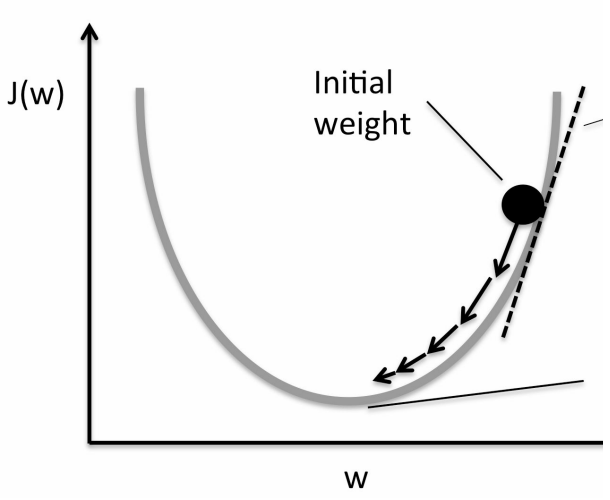
(Geron, 2017)

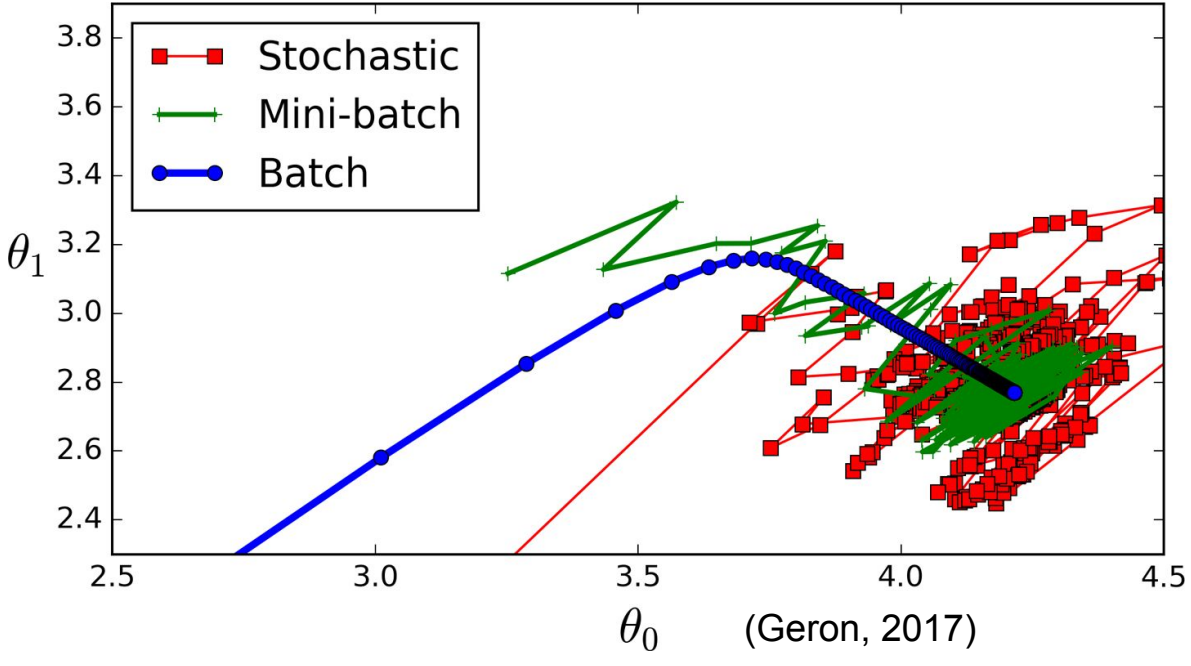
Gradient Descent Options for Linear Regression

Batch Gradient Descent

Stochastic Gradient Descent: One example at a time

Mini-batch Gradient Descent: k examples at a time.





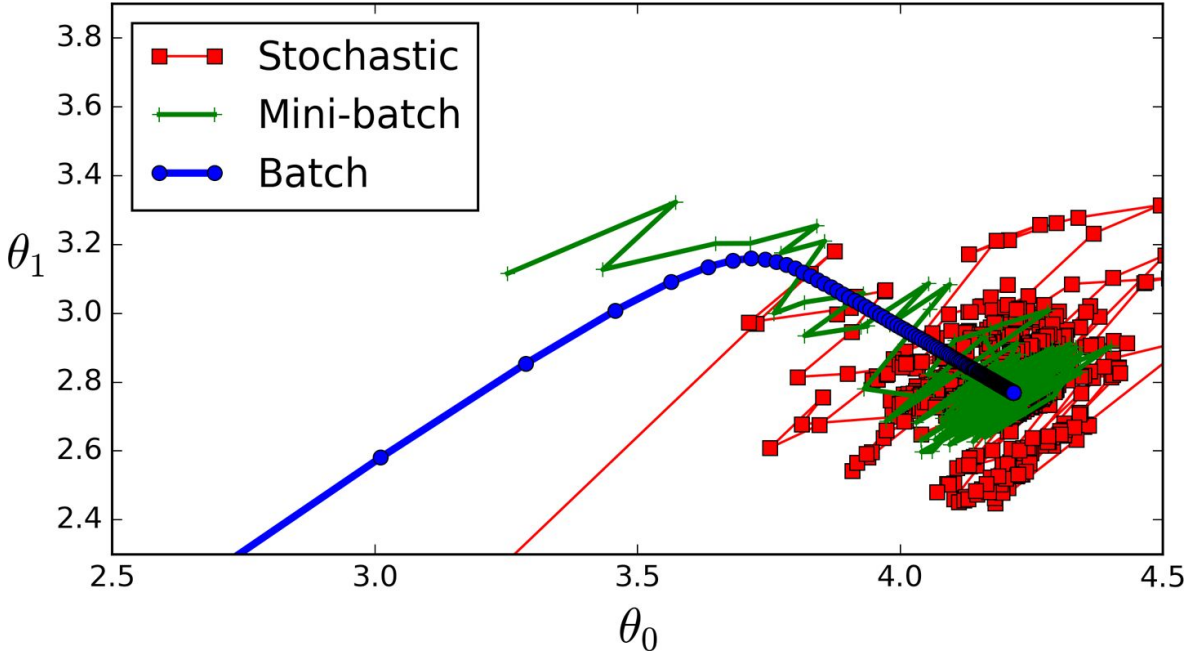
(Geron, 2017)

Gradient Descent Options for Linear Regression

Batch Gradient Descent

Stochastic Gradient Descent: One example at a time

Mini-batch Gradient Descent: k examples at a time.



(Geron, 2017)

From linear regression to neural networks

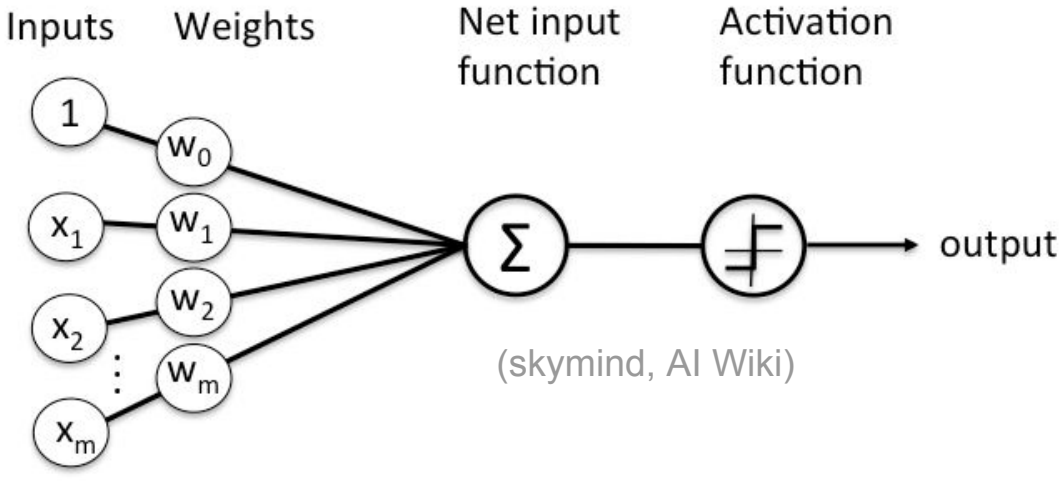
Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$

From linear regression to neural networks

Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$

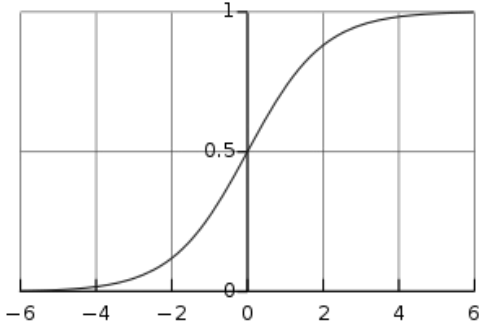


(skymind, AI Wiki)

Common Activation Functions

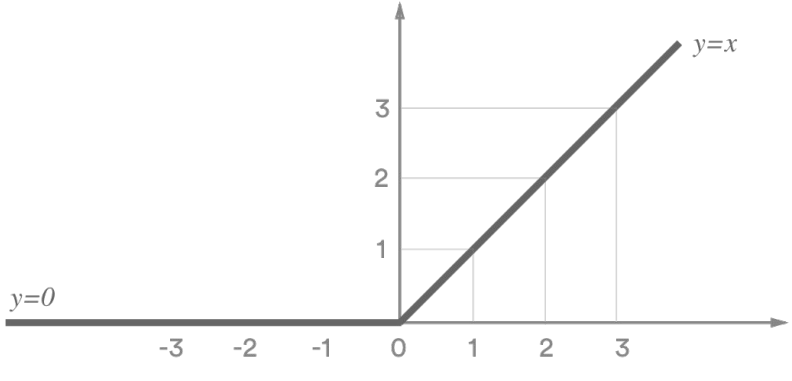
$z = wX$

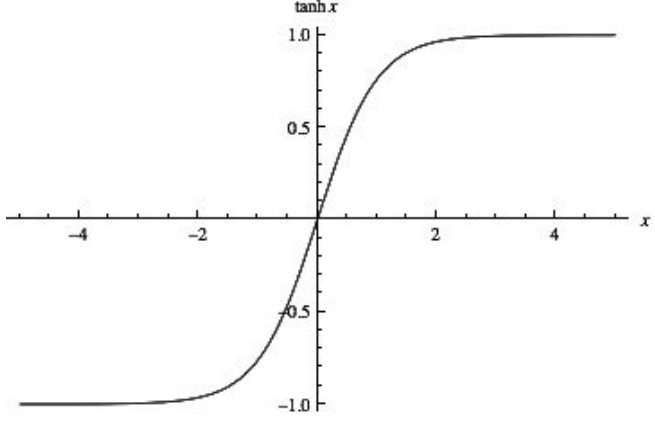
Logistic: $\sigma(z) = 1 / (1 + e^{-z})$



Hyperbolic tangent: $tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1)$

Rectified linear unit (ReLU): $ReLU(z) = \max(0, z)$

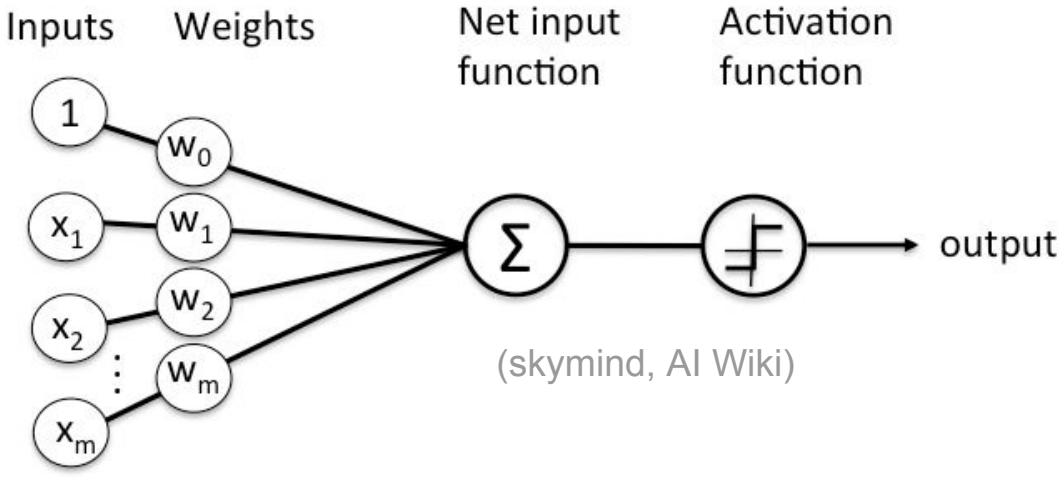




From linear regression to neural networks

Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$

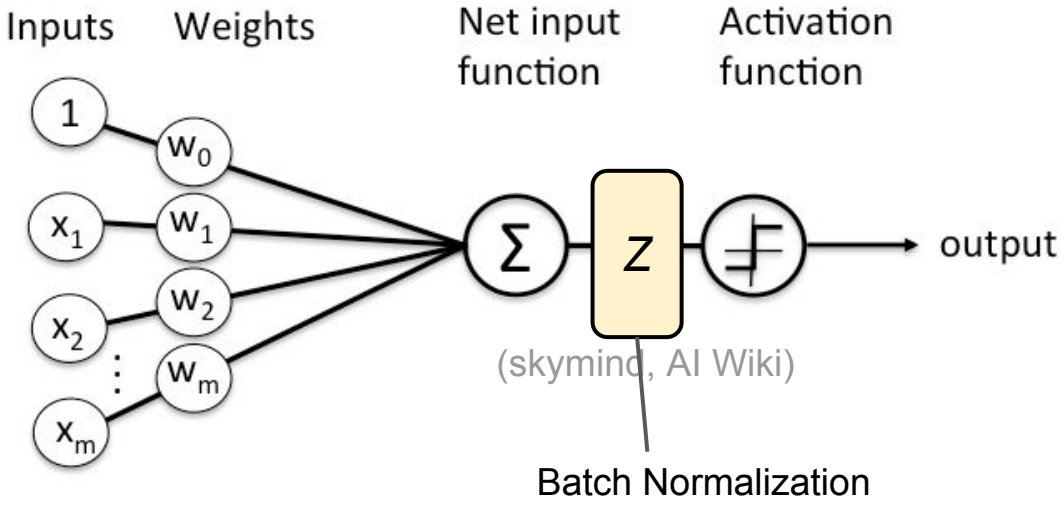


(skymind, AI Wiki)

From linear regression to neural networks

Linear Regression: $y = wX$

Neural Network Nodes: $output = f(wX)$



Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

(Ioffe and Szegedy, 2015)

Batch Normalization

This is just standardizing!
(but within the current batch of observations)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

(Ioffe and Szegedy, 2015)

Batch Normalization

This is just standardizing! (but within the current batch of observations)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

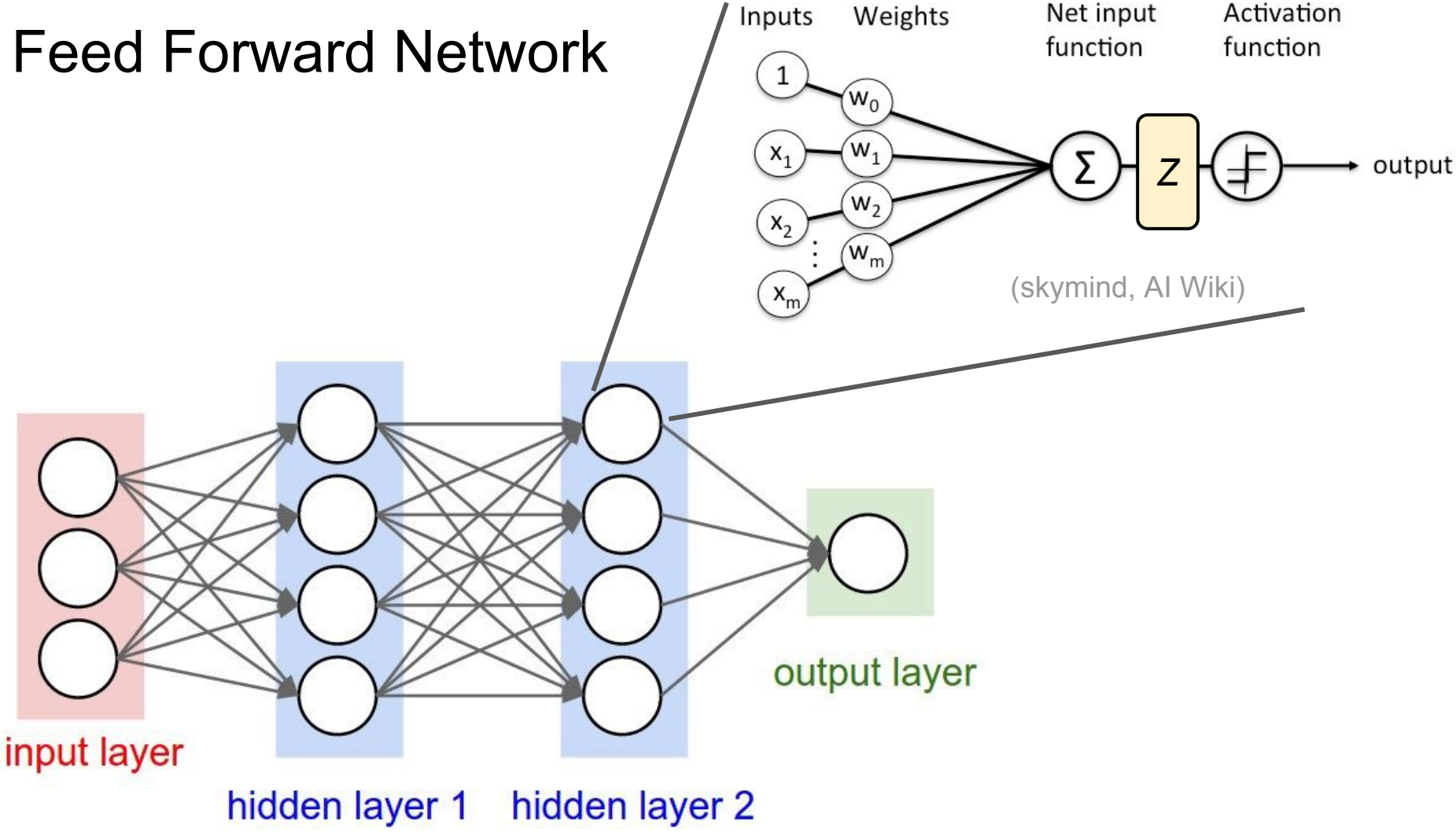
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

(Ioffe and Szegedy, 2015)

Why?

- Empirically, it works!
- Conceptually, generally good for weight optimization to keep data within a reasonable range (dividing by sigma) and such that positive weights move it up and negative down (centering).
- Small effect: When done over mini-batches, adds regularization due to differences between batches.

Feed Forward Network



Recurrent Neural Network

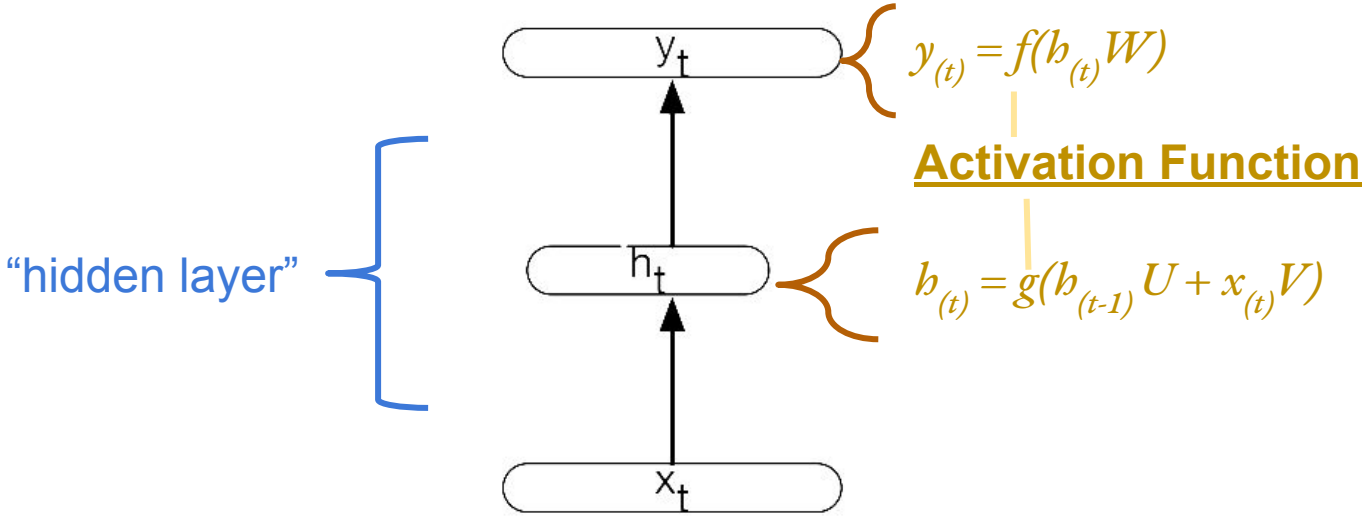
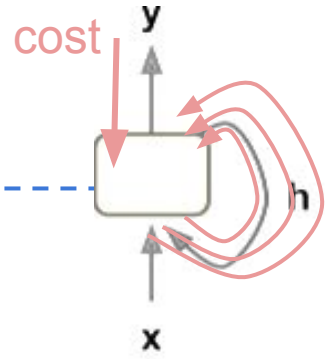


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)

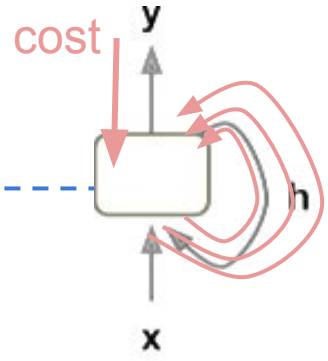
Optimization:



Backward Propagation through Time

```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,h(i-1))+ tf.matmul(W,x(i))) #update hidden
state
    y(i) = tf.softmax(tf.matmul(V, h(i))) #update output
...
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
```

Optimization:



Backward Propagation through Time

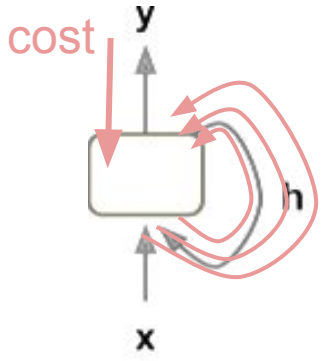
```
...
#define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U,
state
    y(i) = tf.softmax(tf.matmul
...
cost = tf.reduce_mean(-tf.redu
```

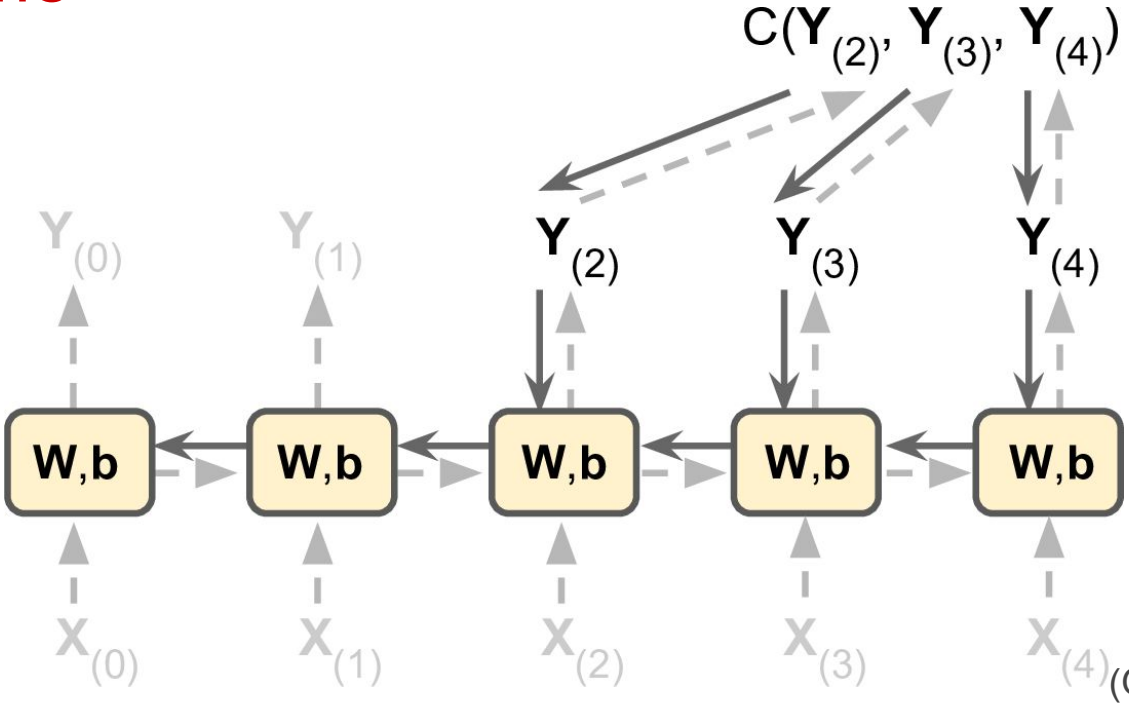
To find the gradient for the overall graph, we use **back propogation,** which *essentially* chains together the gradients for each node (function) in the graph.

With many recursions, the gradients can vanish or explode (become too large or small for floating point operations).

Optimization:

Backward Propagation through Time

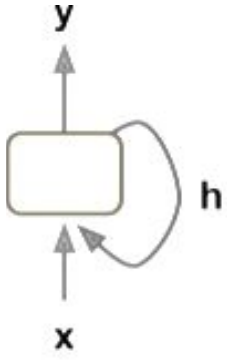




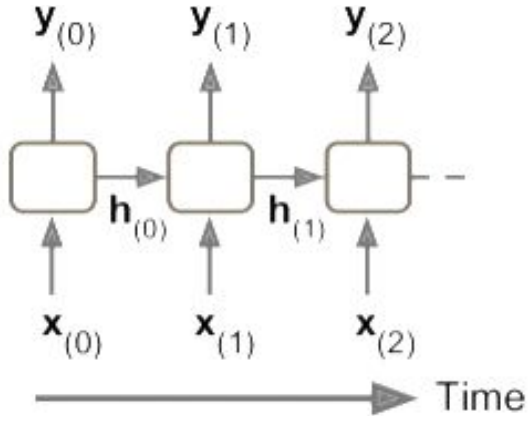
(Geron, 2017)

How to address exploding and vanishing gradients?

Dominant approach: Use Long Short Term Memory Networks (LSTM)



RNN model

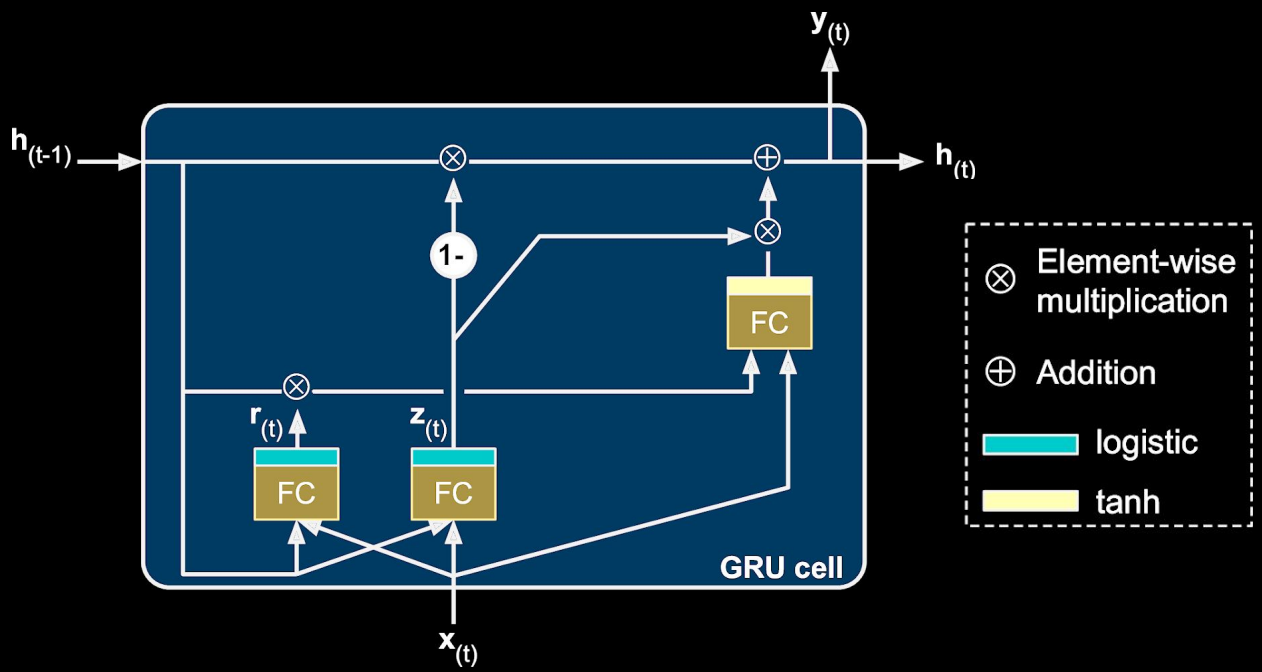


"unrolled" depiction

(Geron, 2017)

The GRU Cell

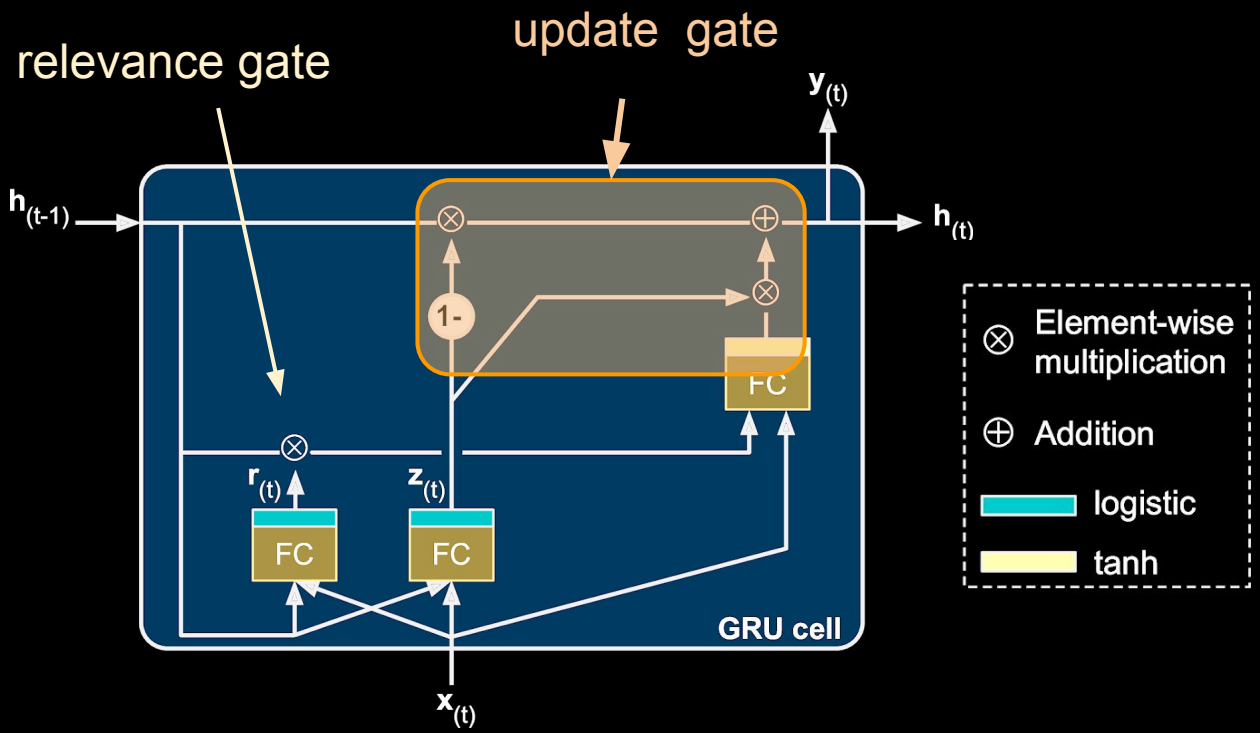
Gated Recurrent Unit



(Geron, 2017)

The GRU

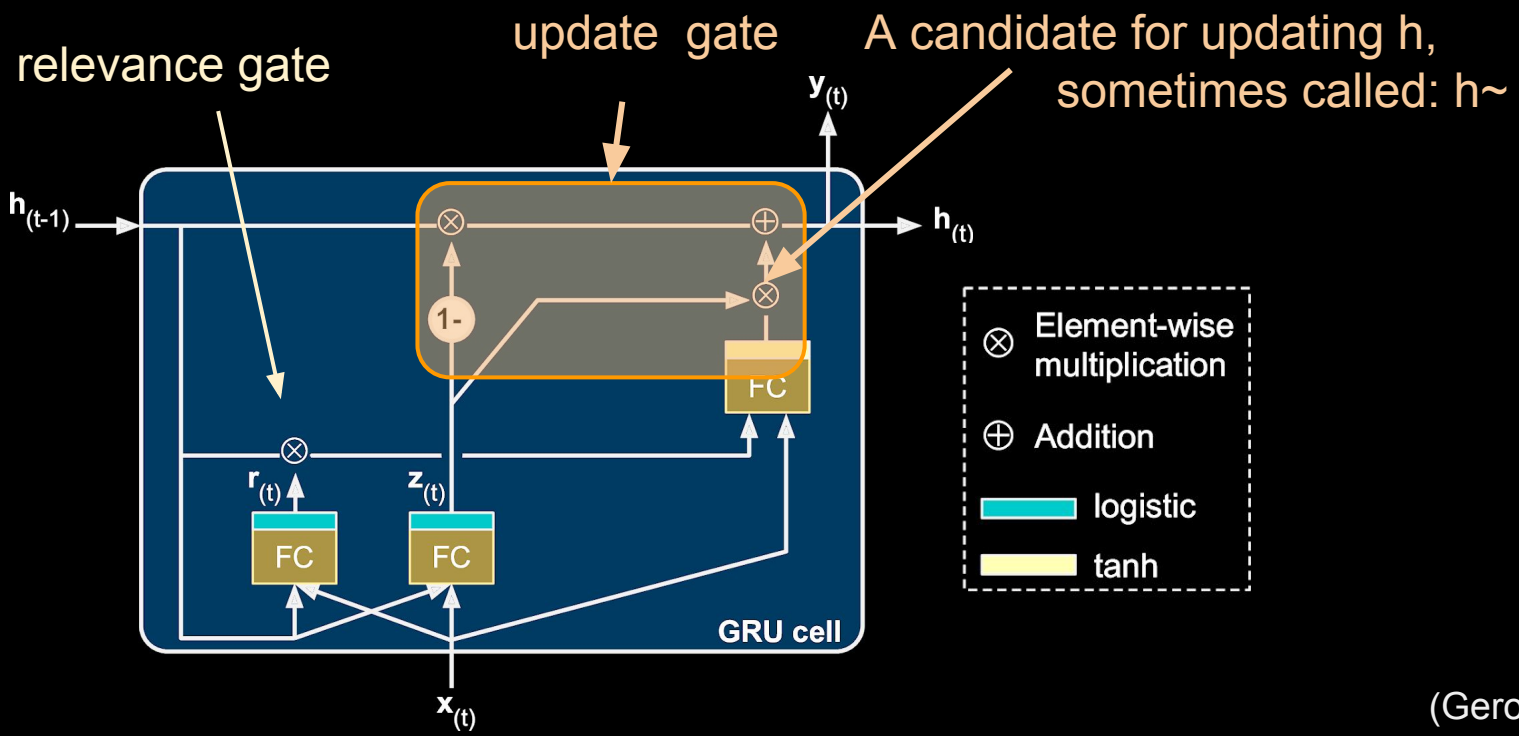
Gated Recurrent Unit



(Geron, 2017)

The GRU

Gated Recurrent Unit

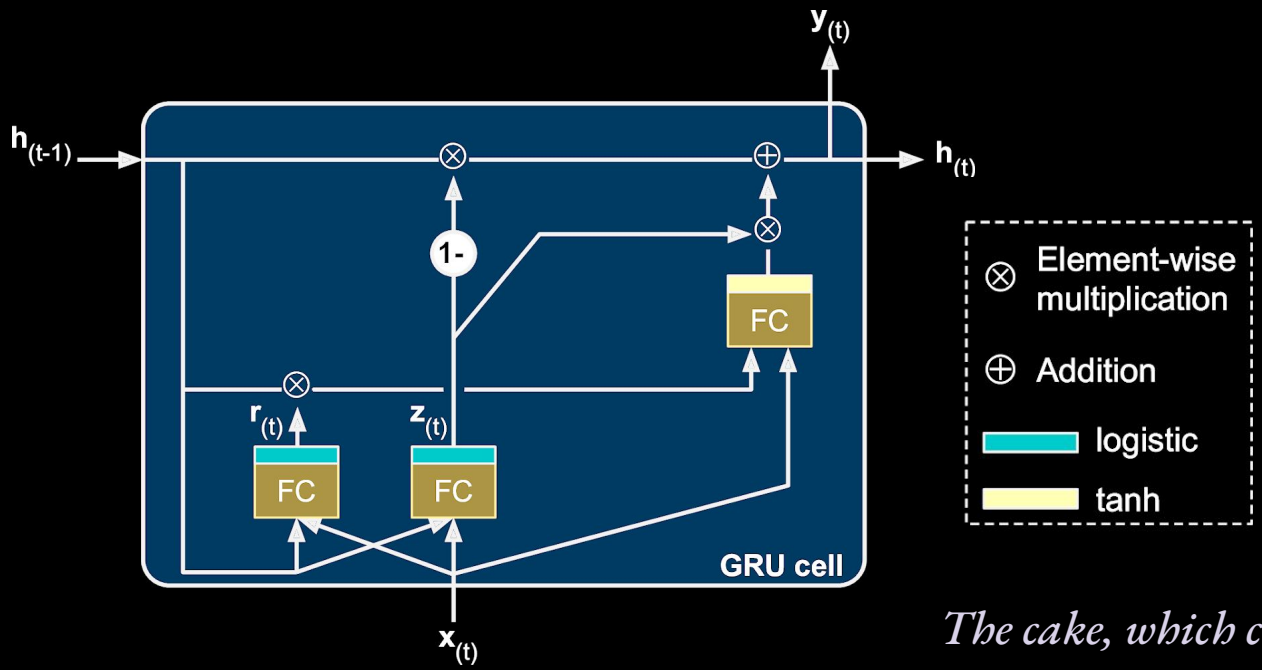


(Geron, 2017)

The GRU

Gated Recurrent Unit

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z) \\
\mathbf{r}_{(t)} &= \sigma(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g\right) \\
\mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}
\end{aligned}
$$



The cake, which contained candles, was eaten.

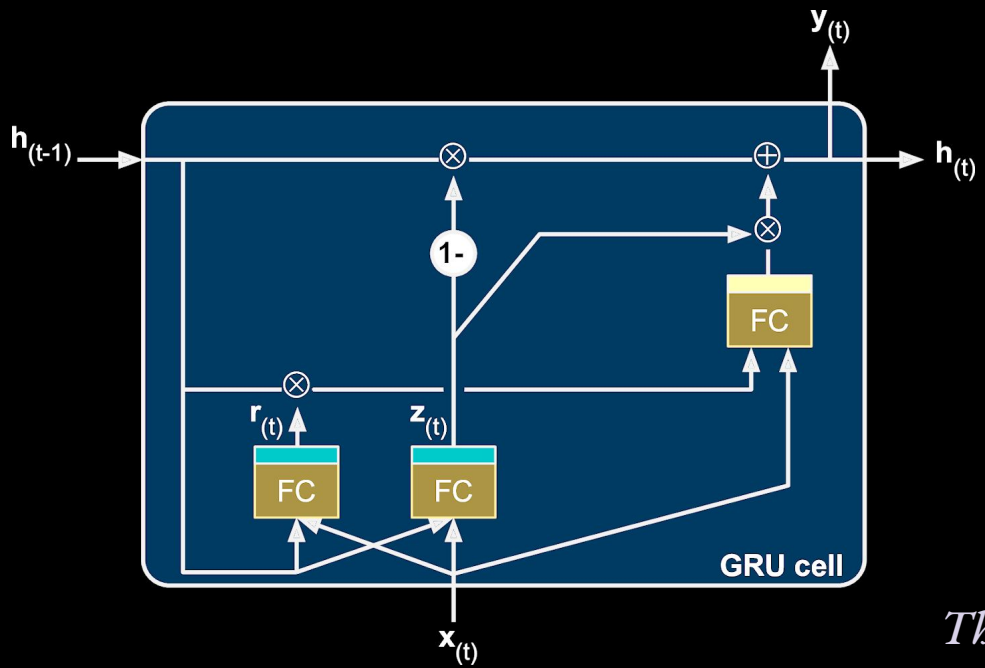
What about the gradient?

$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z)$$

$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$

$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g)$$

$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}$$



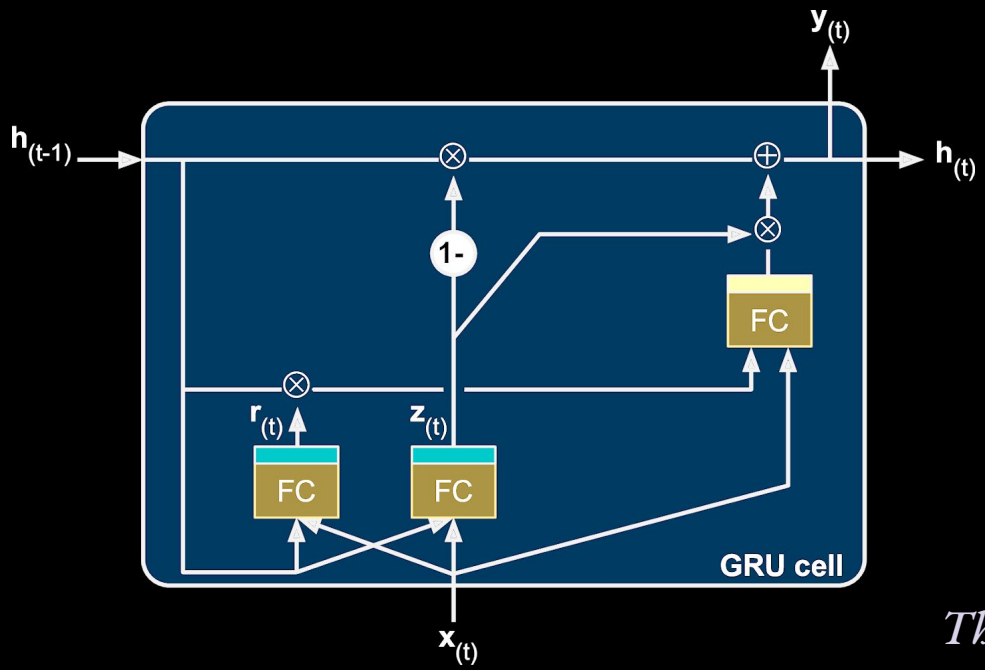
The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of h,

$$h_{(t)} \approx h_{(t-1)}$$

The cake, which contained candles, was eaten.

What about the gradient?

$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z)$$
$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$
$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g)$$
$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}$$



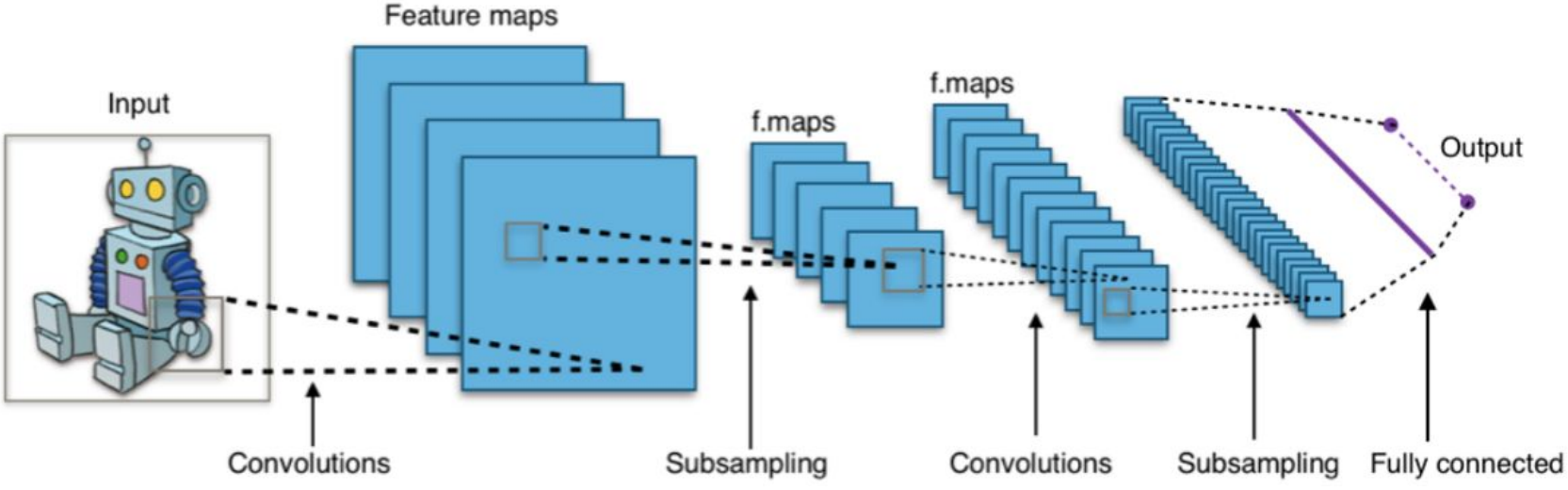
The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of h,

$$h_{(t)} \approx h_{(t-1)}$$

This tends to keep the gradient from vanishing since the same values will be present through multiple times in backpropagation through time. (The same idea applies to LSTMs but is easier to see here).

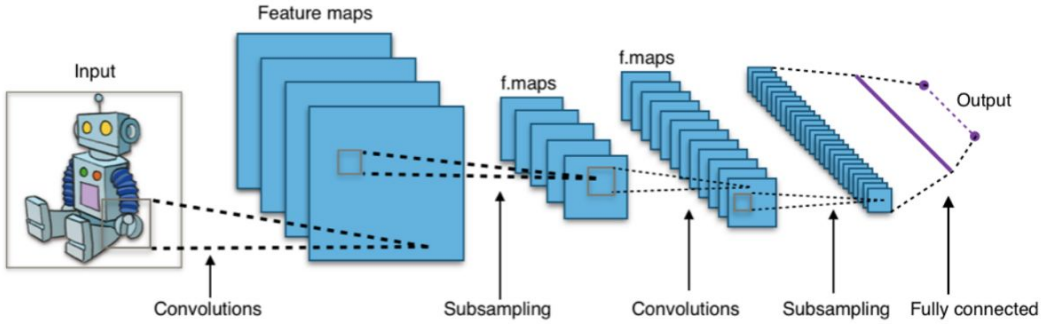
The cake, which contained candles, was eaten.

Convolutional Layer



(wikipedia)

Convolutional Layer



3	1	1	2	8	4
1	0	7	3	2	6
2	3	5	1	1	3
1	4	1	2	6	5
3	2	1	3	7	2
9	2	6	2	5	1

Original image 6x6

"Convolution"

✖

1	0	-1
1	0	-1
1	0	-1

Filter 3x3

=

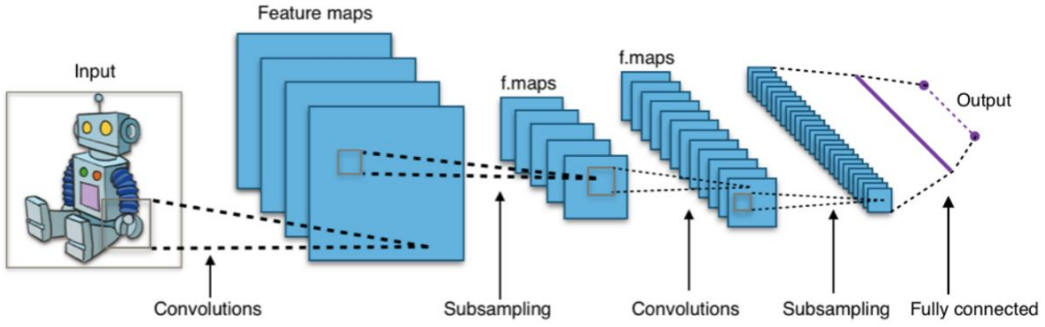
-7	...		
...	...		

Output 4x4

Result of the element-wise product and sum of the filter matrix and the orginal image

(Barter, 2018)

Convolutional Layer



"Convolution"

3	1	1	2	8	4
1	0	7	3	2	6
2	3	5	1	1	3
1	4	1	2	6	5
3	2	1	3	7	2
9	2	6	2	5	1

Original image 6x6

✖

1	0	-1
1	0	-1
1	0	-1

Filter 3x3

=

-7	...		
...	...		

Output 4x4

Result of the element-wise product and sum of the filter matrix and the orginal image

Breakthrough in image classification: Let the model automatically learn the filter weights!

How to train deep models for classification?

Short Answer: Same as logistic regression.

```
RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))
            #where did this come from?
```

Logistic Regression Likelihood: $$L(\beta_0, \beta_1, ..., \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$$

Log Likelihood: $$\ell(\beta) = \sum_{i=1}^{N} y_i log\ p(x_i) + (1-y_i) log\ (1-p(x_i))$$

Log Loss: $$J(\beta) = -\frac{1}{N}\sum_{i=1}^{N} y_i log\ p(x_i) + (1 - y_i) log\ (1 - p)(x_i))$$

Cross-Entropy Cost: $$J = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_i log\ p(x_{i,j})$$ (a “multiclass” log loss)

Final Cost Function: $$J^{(t)} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|V|} y_{i,j}^{(t)} log\ \hat{y}_{i,j}^{(t)}$$ -- ”cross entropy error”

Summary

- Goal is accurate prediction of y (outcome) given features (x)
- Use L1 or L2 penalization (as a regularization) to avoid overfit
- Reason for Train, Dev, Test split
- Components of a neural network
- Batch Normalization
- Distribution options: why is data parallelism preferred?
- Recurrent Neural Network
- Convolution Operation with Filters