UNIVERSITY OF KWAZULU-NATAL ™
INYUVESI YAKWAZULU-NATALI

NITheCS
National Institute for
Theoretical and Computational Sciences

Nearest Neighbors and Decision Trees

A Different Kind of Learning

To this point: *parametric* learning

Given a predetermined family of functions that maps from input features to prediction, learn a set of *parameters* for this function

.. one way: by optimizing against the *loss function*

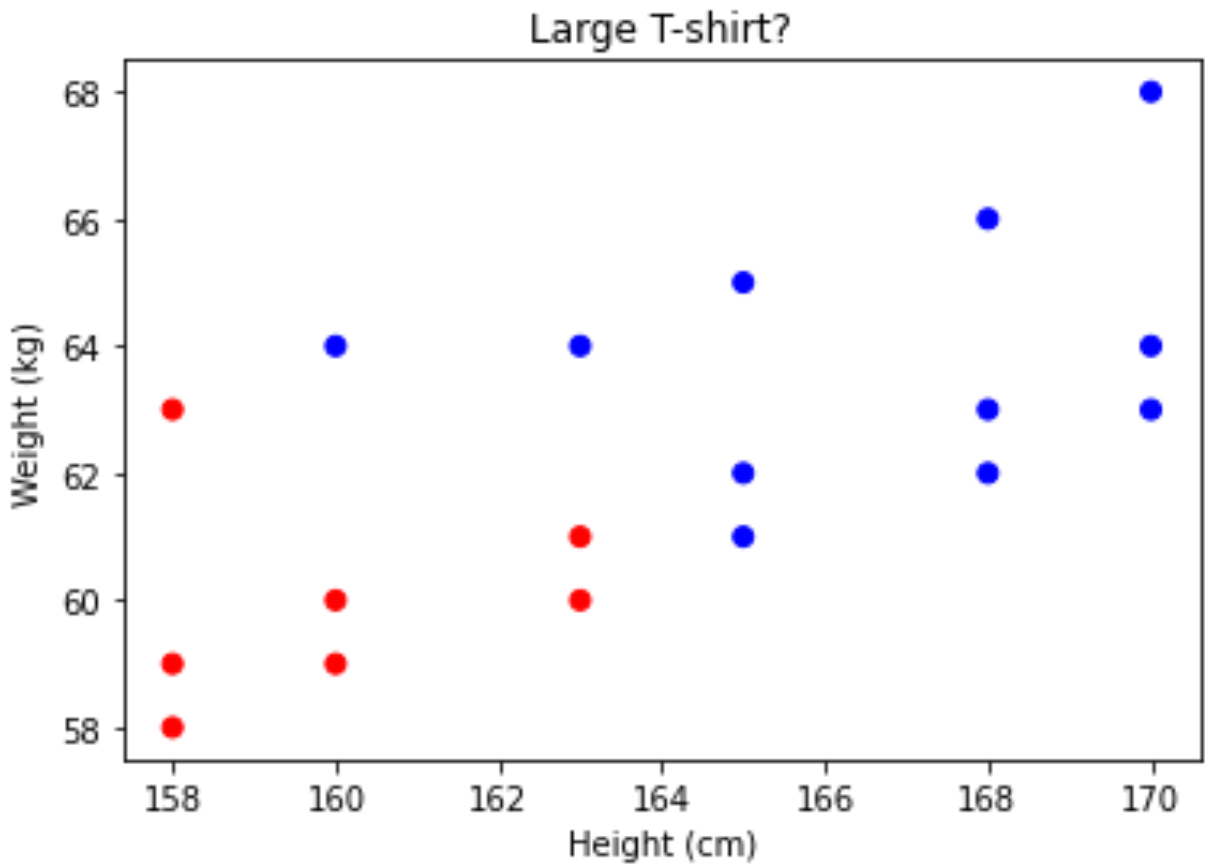
linear regression – continuous-valued output

logistic regression – Boolean-valued output

But this is not the only kind of ML algorithm – now, we'll see two variations on this theme

- k-Nearest Neighbors
- Decision trees

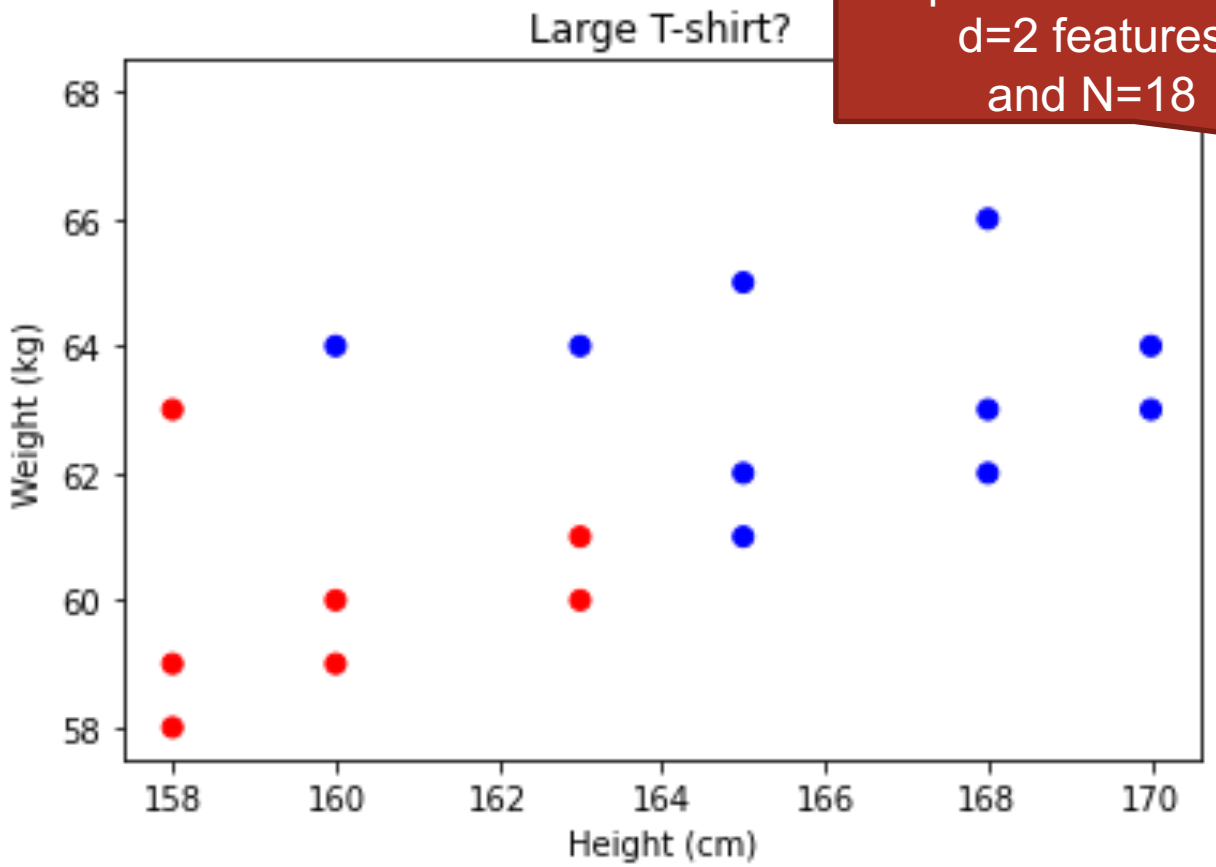
Our Default Setup: Training for Binary Classification



Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	59	F
160	60	F
163	60	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

Our Default Setup: Training for Binary Classification



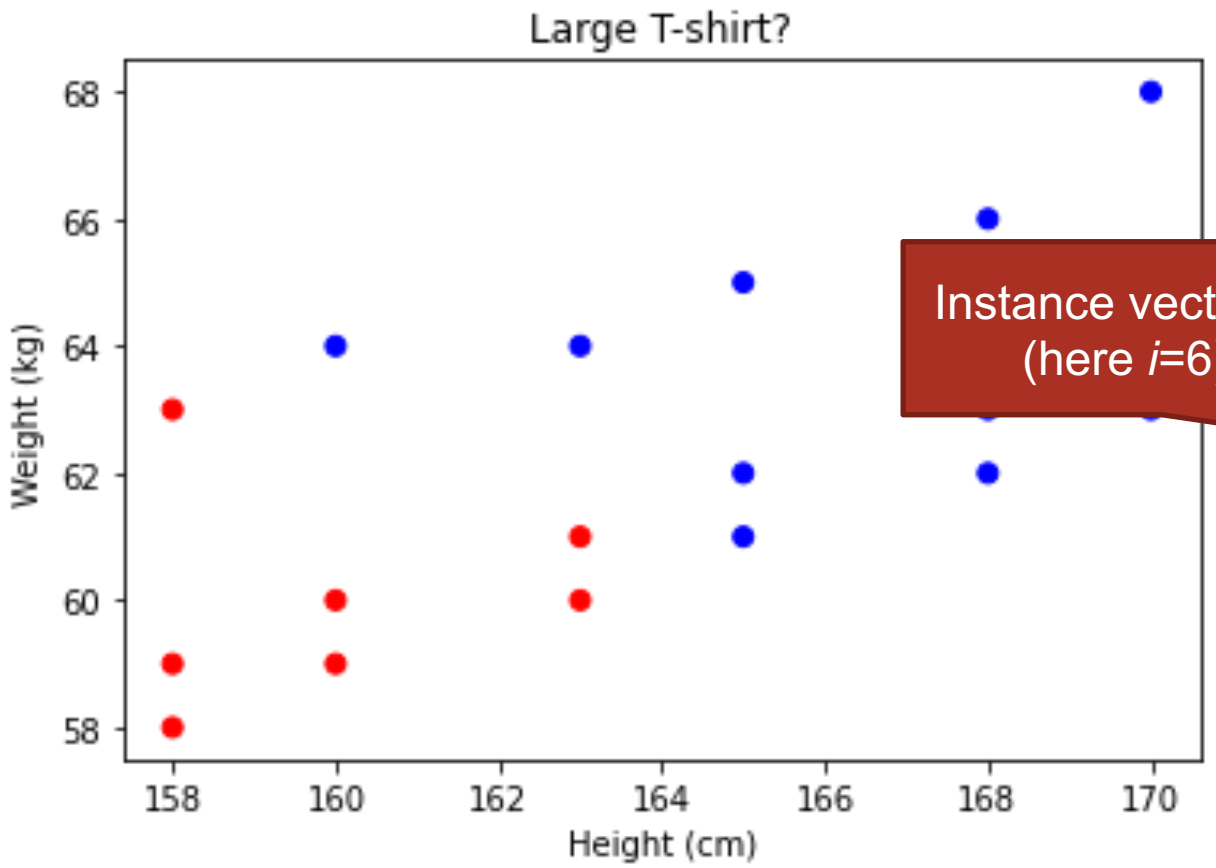
Input matrix ***X*** with d=2 features and N=18

Class vector *y*

Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	59	F
160	60	F
163	60	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

Our Default Setup: Training for Binary Classification



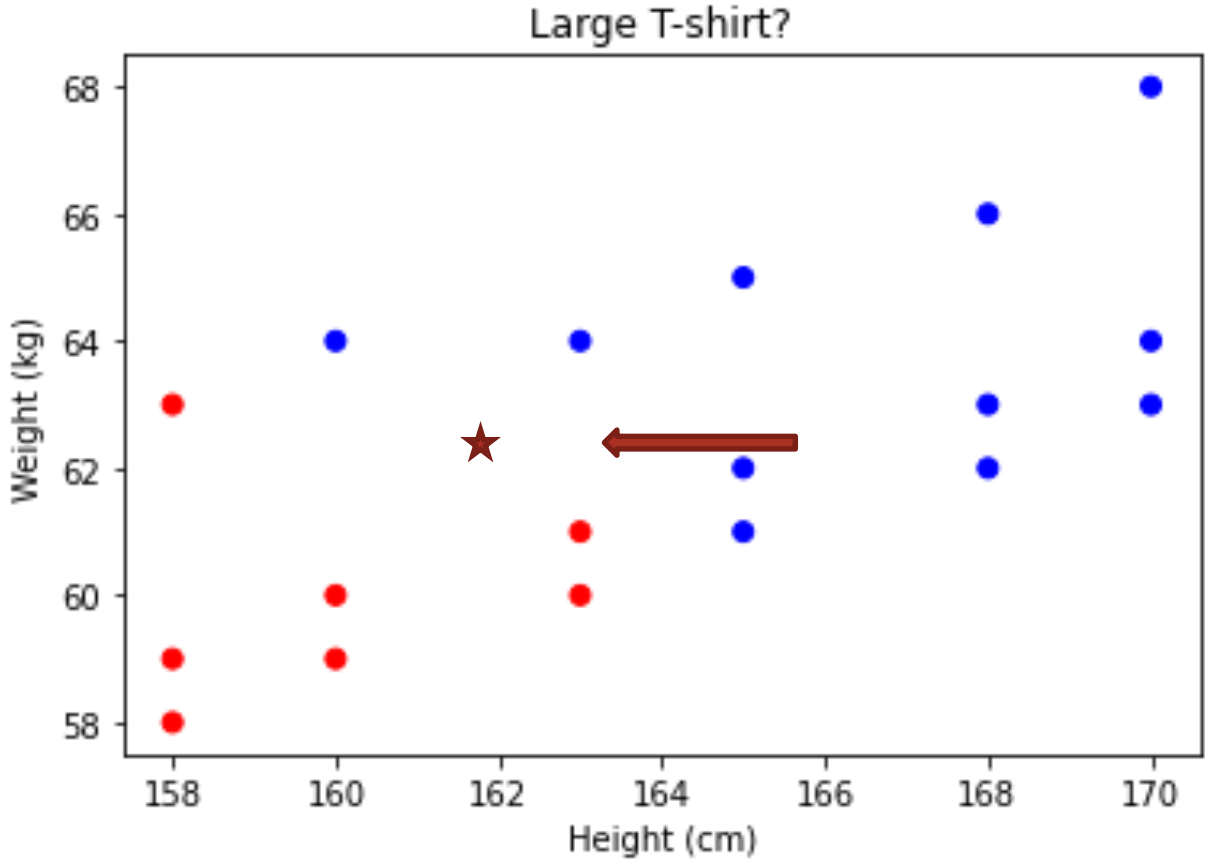
Instance vector x_i, (here i=6)

Class label y_i

Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	[illegible]	F
160	[illegible]	F
163	[illegible]	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

Our Default Setup: Binary Classification for New Data – What Label?



Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	59	F
160	60	F
163	60	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

k-Nearest Neighbors (kNN)

To predict category label y of a new point $\boldsymbol{x}$ (classification):

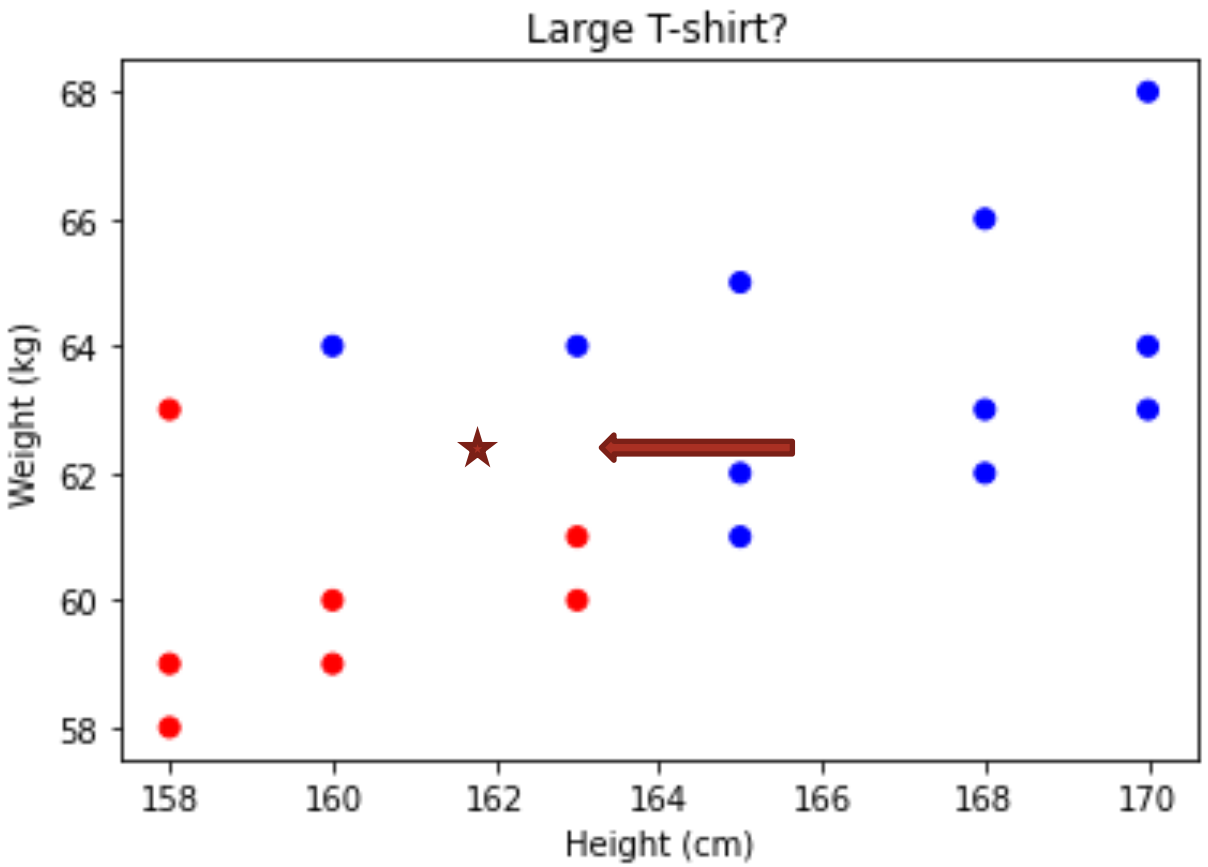
- Find *k* nearest neighbors (according to some distance metric)
- Assign the **majority label** to the new point

To predict numeric value y of a new point $\boldsymbol{x}$ (regression):

- Find k nearest neighbors
- "Average" the values associated with the neighbors

If we change *k* we may get a different prediction

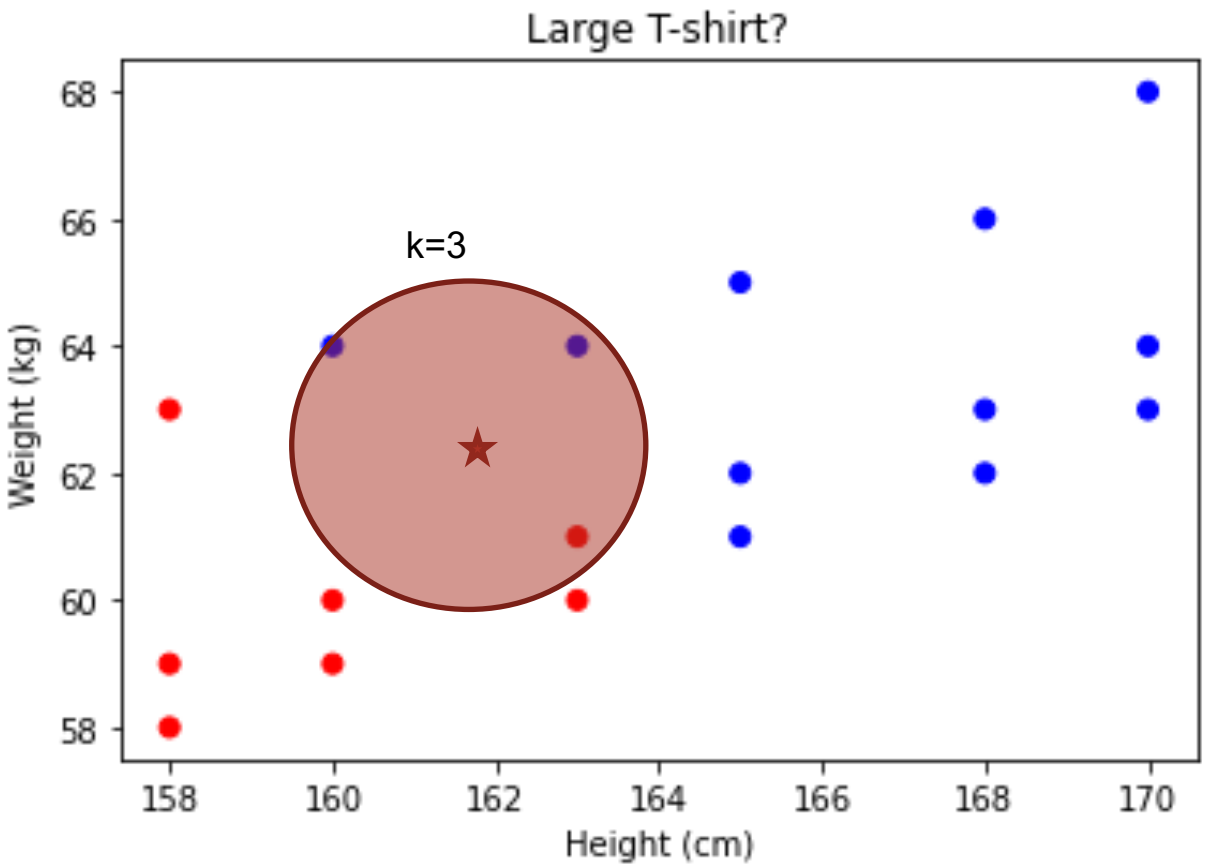
kNN Prediction: What Label?



Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	59	F
160	60	F
163	60	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

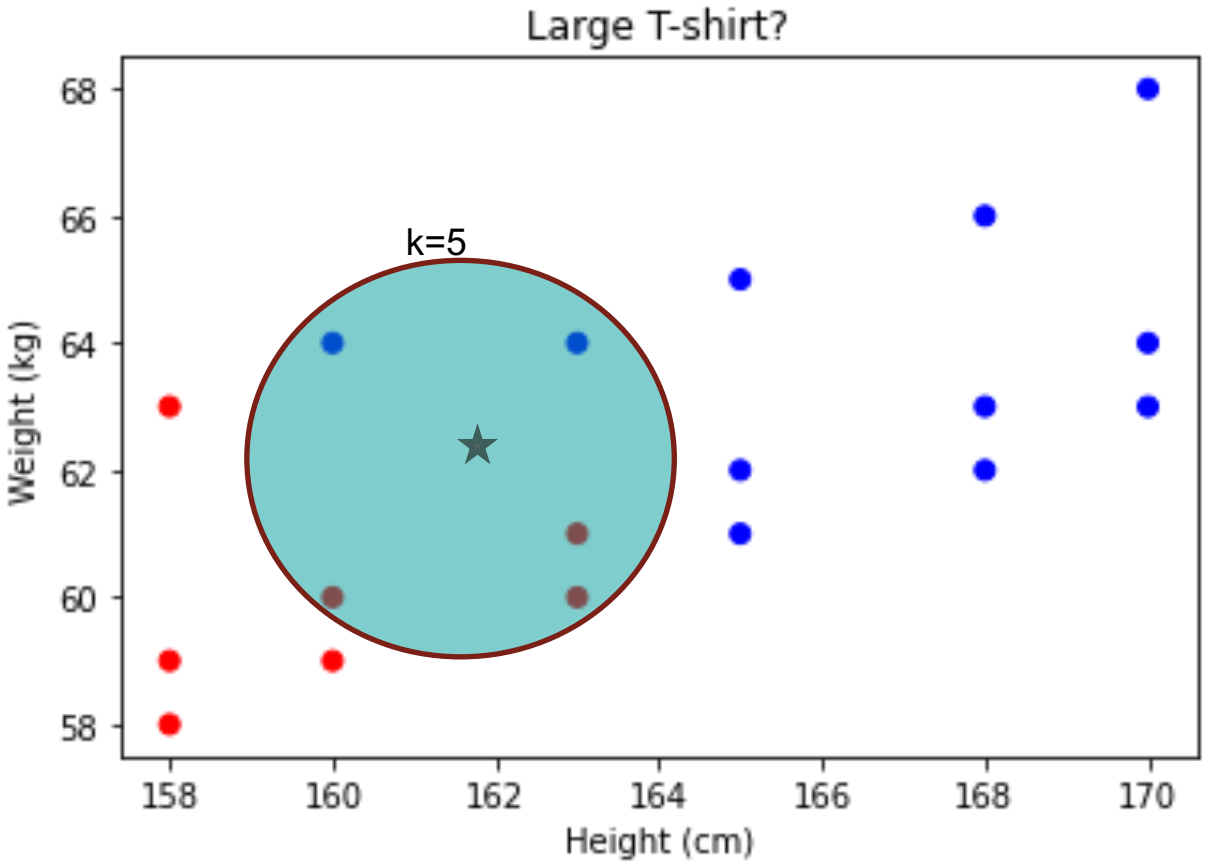
kNN Prediction: What Label?



Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	59	F
160	60	F
163	60	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

kNN Prediction: What Label?



Height (cm)	Weight (kg)	Large (vs Medium) t-shirt?
158	58	F
158	59	F
158	63	F
160	59	F
160	60	F
163	60	F
163	61	F
160	64	T
163	64	T
165	61	T
165	62	T
165	65	T
168	62	T
168	63	T
168	66	T
170	63	T
170	64	T
170	68	T

Based on data from https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html

What Does "Nearest" Mean?

Must define a "distance function" between any two samples $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$

Note: boldface $\boldsymbol{x}$ denotes a vector in widely used notation. In our case, each of these is a 2D vector: $\boldsymbol{x_i} = [x_{i1}, x_{i2}]$

"Nearest neighbor" = sample with least "distance". Some commonly used distances:

$$\left(\sum_d (x_{1j} - x_{2j})^1\right)^{\frac{1}{1}}$$

ℓ_1 distance

$$\sum_d |x_{1j} - x_{2j}|$$

$$\left(\sum_d (x_{1j} - x_{2j})^2\right)^{\frac{1}{2}}$$

ℓ_2 distance

Also, "Euclidean" distance

$$\left(\sum_d (x_{1j} - x_{2j})^{\to\infty}\right)^{\to 0}$$

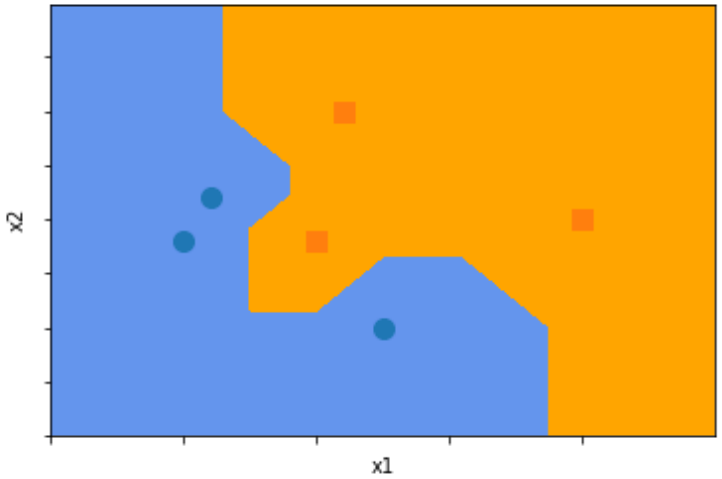
ℓ_∞ distance

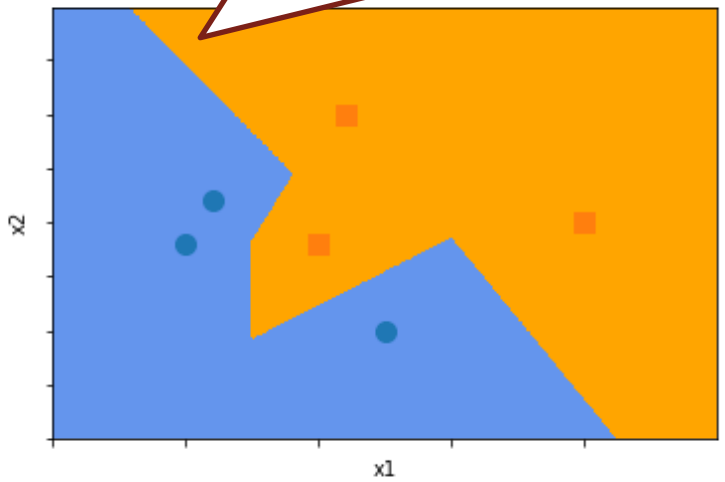
$$\max_d \ (x_{1j} - x_{2j})$$

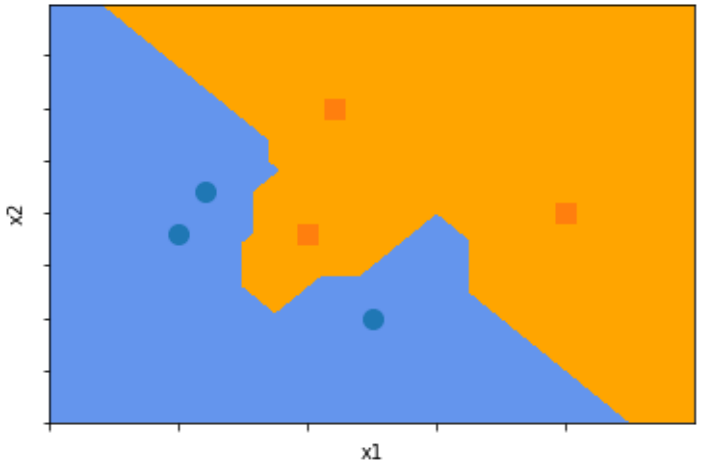
Different Distances Produce Different Outcomes

Fix k =1 neighbors

Classifier "decision boundary" plots show what class would be assigned at *every point* $\boldsymbol{x}$







ℓ_1 distance

$$\sum_d |x_{1j} - x_{2j}|$$

ℓ_2 distance

Also, "Euclidean" distance

ℓ_∞ distance

$$\max_d \ (x_{1j} - x_{2j})$$

What about Distances between Non-numeric Data? Consider Strings…

Hamming distance (number of characters that are different)

ABCDE vs AGDDF → 3

Edit distance (number of character inserts/replacements/deletes to go from one to the other)

ROBOT vs BOT → 2

Jaccard distance between sets $\frac{|A \cap B|}{|A \cup B|}$

between **n-grams** (n-character substrings of the strings, with (n-1) character padding)

$$ROBOT$$ vs $$BOT$$ → |{BOT,OT$,T$$}| / |{$$R,$RO,ROB,OBO,$$B,BO,BOT,OT,T$$}|

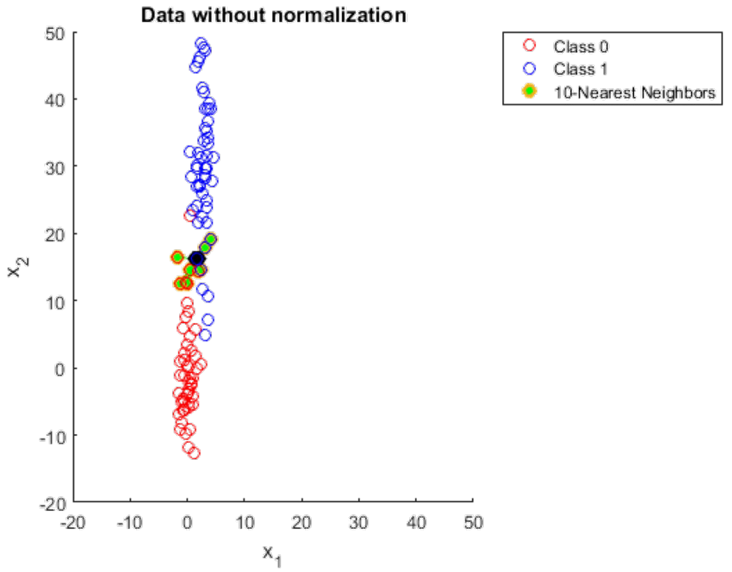
3 9

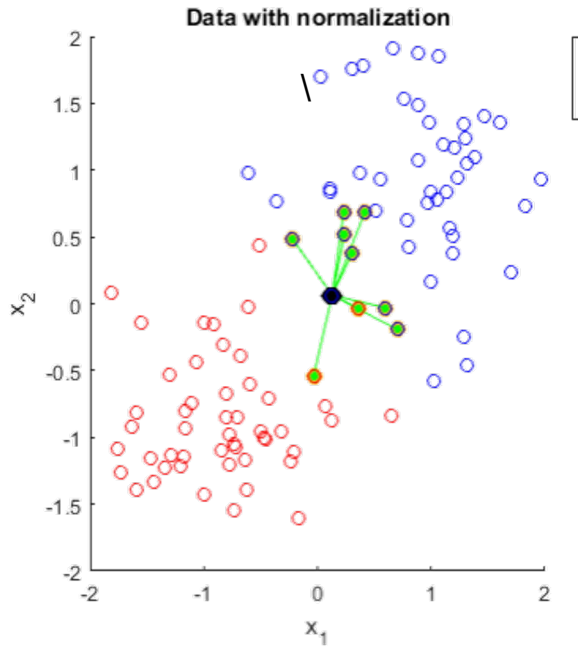
Beware: Feature Scaling affects Nearest Neighbors

Our previous study of linear / logistic regression:

- OLS regression was *scale-invariant*
- Regularization was affected by the scale of different features

Even more of a concern with kNN: note that we are using a distance measure like L2, which is affected dramatically by feature scales!





https://stats.stackexchange.com/questions/287425/why-do-you-need-to-scale-data-in-knn

General Problem: "Curse of Dimensionality"

Adding more dimensions makes lots of things weird and counterintuitive

e.g., the percentage of the volume of a D-dimensional sphere with radius r, that lies beyond $\ell_2$distance $0.99r$ from the center is:

- 3% at $D = 3$
- 63% at $D = 100$
- 99.99% at $D = 1000$

also, with enough dimensions most points are of roughly equal distance!

For k-NN, nearest neighbors become very far apart, and of similar distance – therefore **unreliable predictors**

Stepping back…
where are the *parameters* we learn?

Think broadly of the "parameters" as everything required to produce the output, for a given model class. i.e.

Model class + parameters + new input $\boldsymbol{x} \rightarrow$ predicted y

"kNN classifier" ??

A: The full training dataset!

Funnily, methods like these where the parameters are either *the training data* itself, or *grow in size "automatically"* with the training data, are called "**non-parametric**" machine learning approaches.

Summary of k-Nearest Neighbors

A case of "non-parametric" learning

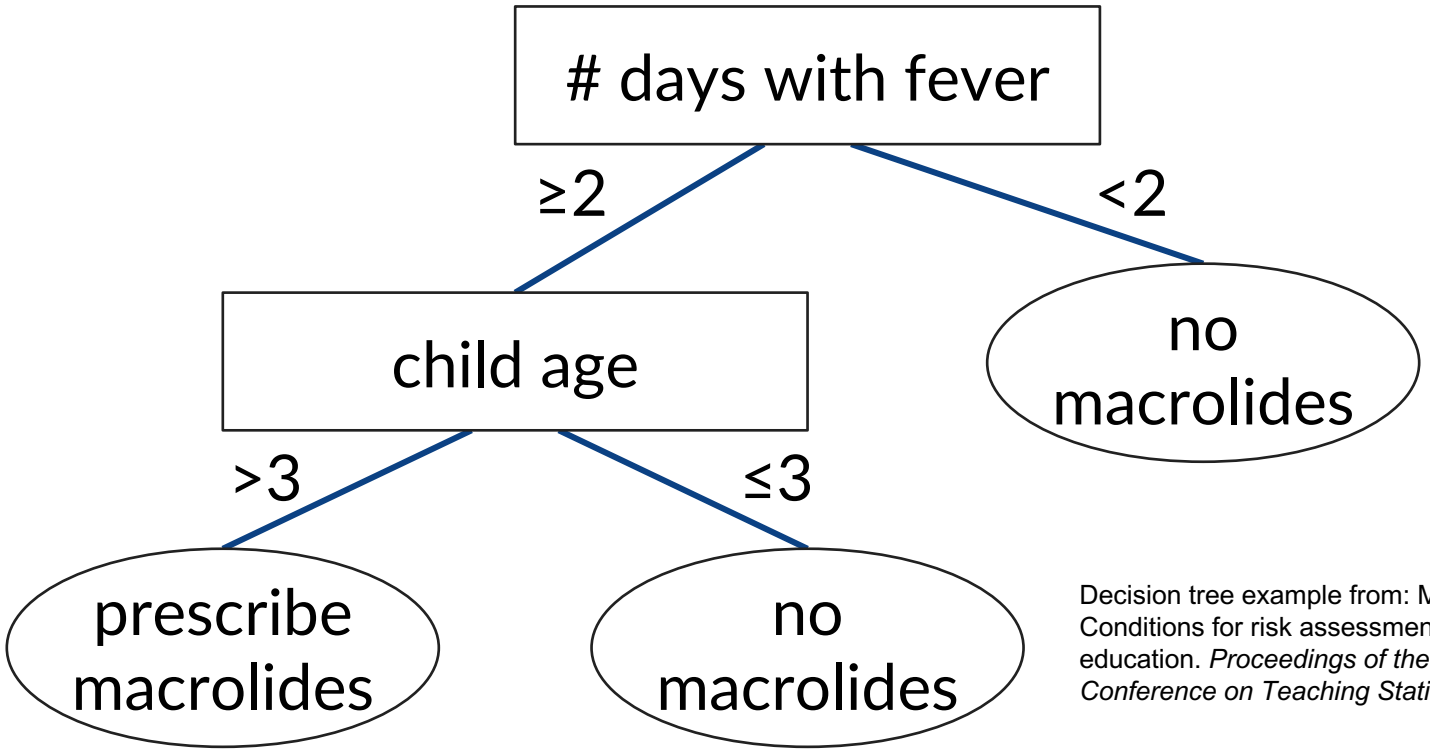
- Uses the full training dataset as parameters
- Requires careful treatment of feature scaling

- Main decisions: the value of k, the distance function

Tends to work well in practice. but beware scalability

Decision Trees for Humans

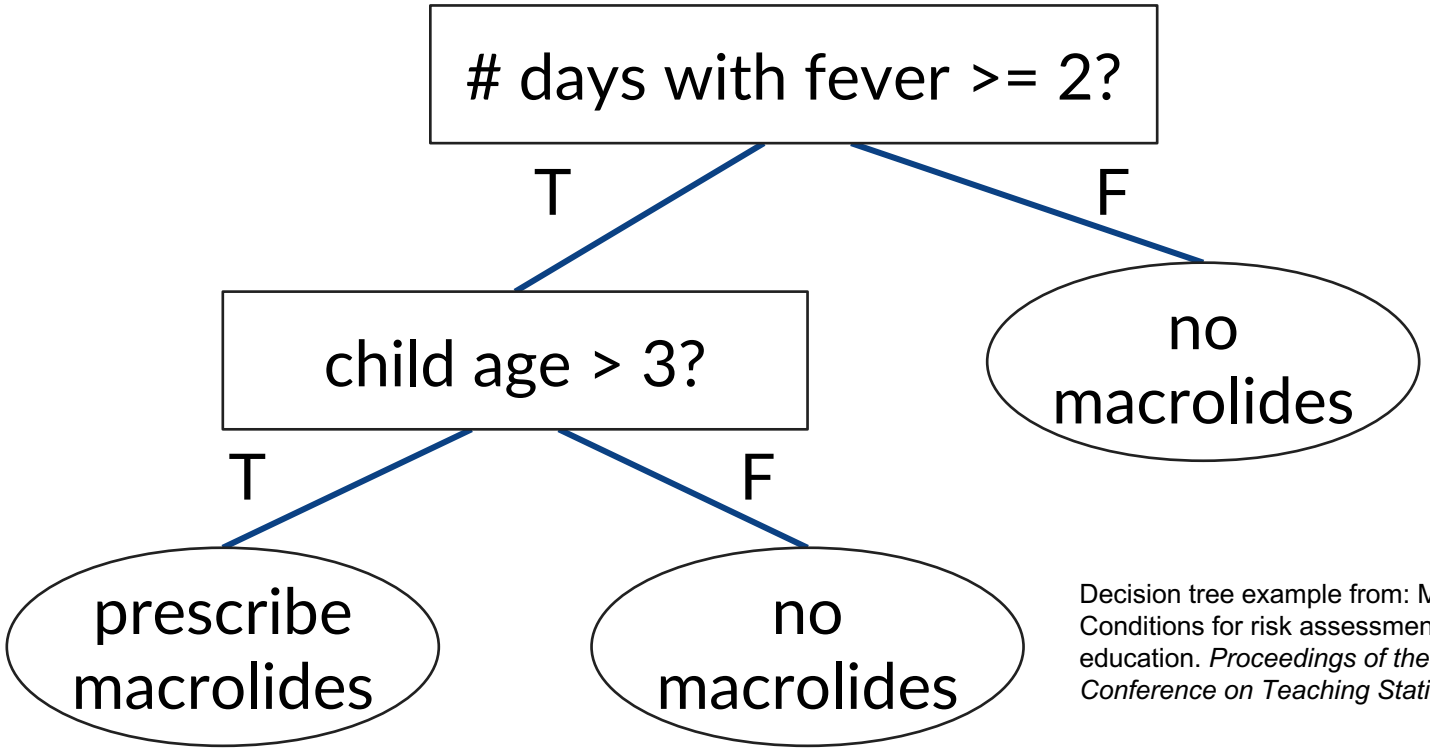
Simple decision tree used in medicine:



Decision tree example from: Martignon and Monti. (2010). Conditions for risk assessment as a topic for probabilistic education. *Proceedings of the Eighth International Conference on Teaching Statistics* (ICOTS8).

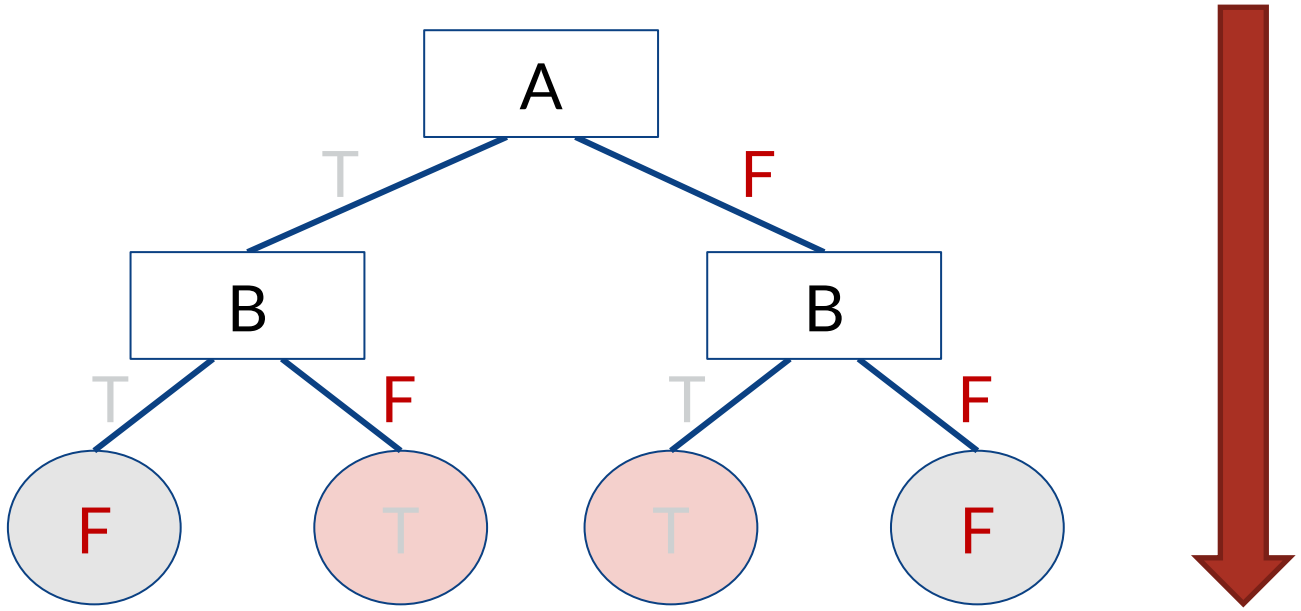
A Decision Tree Based on Boolean Tests

For continuous features, we'll restrict our study to internal nodes that can test the **value of one attribute**. We can generalize to categorical values (binary decision tree).



Decision tree example from: Martignon and Monti. (2010). Conditions for risk assessment as a topic for probabilistic education. *Proceedings of the Eighth International Conference on Teaching Statistics* (ICOTS8).

Decision Tree Training – Grow Top-Down

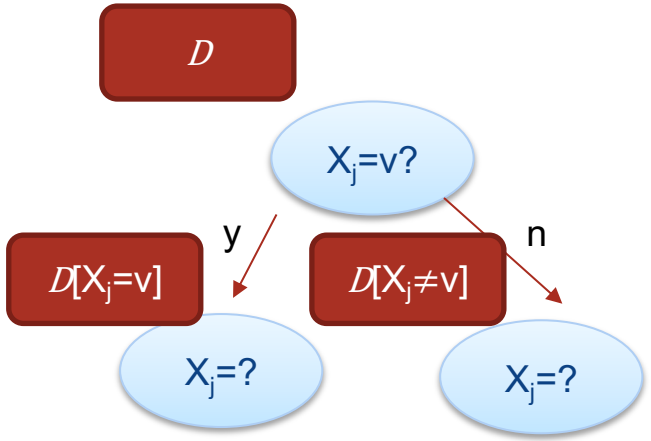


Top-Down Decision Tree Induction

[ID3 (1986), C4.5(1993) by Quinlan]

Let $\mathcal{D}$ be a set of labeled instances; $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N} = [X_{N\times D}, \boldsymbol{y}_{N\times 1}]$

Let $\mathcal{D}[X_j = v]$ be the subset of $\mathcal{D}$ where feature X_j has value v



function `train_tree(`$\mathcal{D}$`)`

1. If data $\mathcal{D}$ **all have the same label** y, return new `leaf_node(`y`)`, else:

2. Pick the "best" feature X_j to partition $\mathcal{D}$
3. Set `node = new decision_node(`X_j`)`
4. For each value v that X_j can take

   Recursively create a new child `train_tree(`$\mathcal{D}[X_j = v]$`)` of `node`
5. Return `node`

Top-Down Decision Tree Induction

[ID3 (1986), C4.5(1993) by Quinlan]

Let $\mathcal{D}$ be a set of labeled instances; $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N} = [X_{N\times D}, y_{N\times 1}]$

Let $\mathcal{D}[X_j = v]$ be the subset of $\mathcal{D}$ where feature X_j has value v

How do we choose which feature is best?

function `train_tree(`$\mathcal{D}$`)`

1. If data $\mathcal{D}$ **all have the same label** y, return new `leaf_node(`y`)`, else:

2. Pick the "best" feature X_j to partition $\mathcal{D}$
3. Set `node = new decision_node(`X_j`)`
4. For each value v that X_j can take

   Recursively create a new child `train_tree(`$\mathcal{D}[X_j = v]$`)` of `node`
5. Return `node`

Learning Bias: Occam's Razor

Principle stated by William of Ockham (1285-1347)

- "non sunt multiplicanda entia praeter necessitatem" --
  "entities are not to be multiplied beyond necessity"

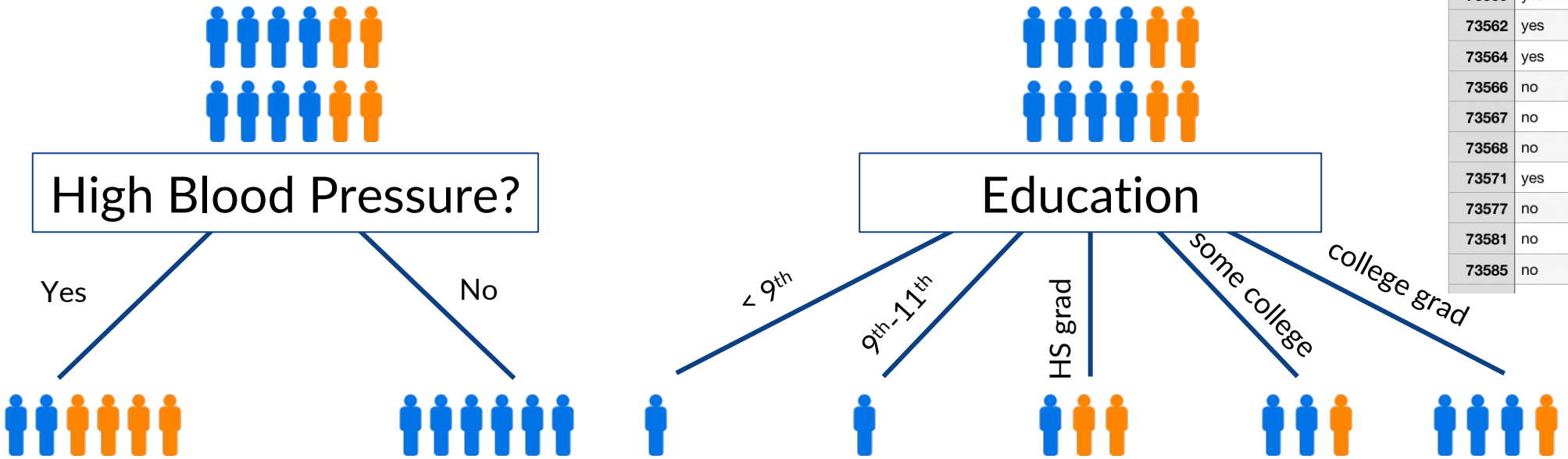
- also called Ockham's Razor, Law of Economy, or Law of Parsimony

Key Idea: The simplest consistent explanation is the best

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

Choosing Features for Short Decision Trees

Key Idea: good features partition the data into subsets that are either "all positive" or "all negative" (ideally)



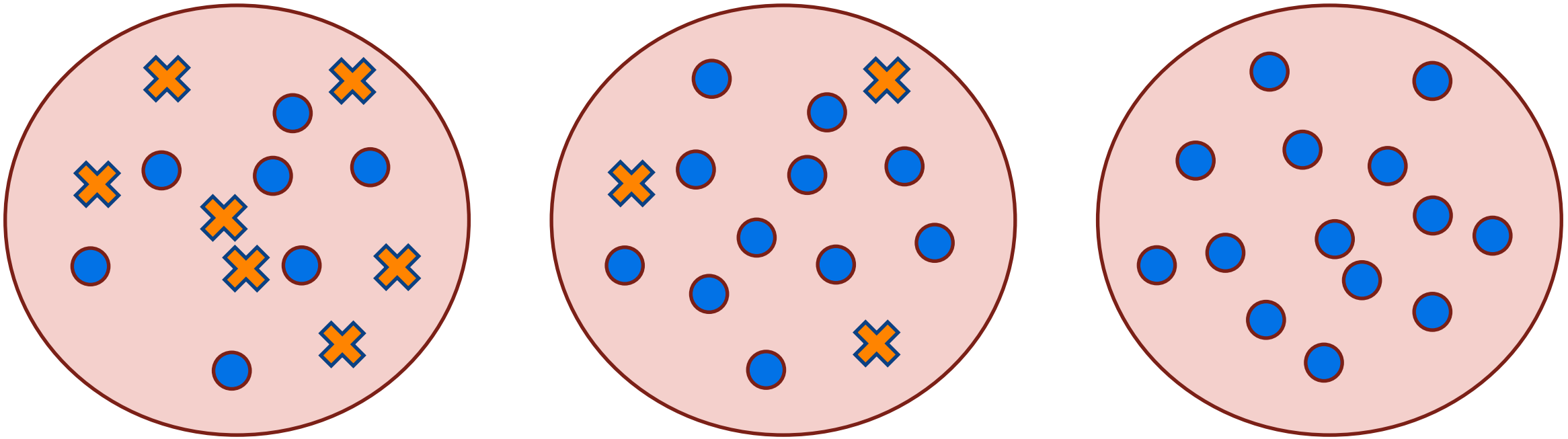
Subset of Data

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no

Which split is more informative?

Formalizing this: Impurity

Could we come up with an "impurity function" of a set of samples?



maximally impure ⟷ minimally impure

Note: All x's is also "pure"

A Candidate For An "Impurity Function": *Entropy*

Let Y be any discrete random variable that can take on n values

The entropy of Y is given by

$$H(Y) = -\sum_{i=1}^{n} P(Y = i) \log_2 P(Y = i)$$

Strictly, the entropy $H(Y)$ maps from a probability distribution (over the class label random variable Y) to an impurity score

$\updownarrow$

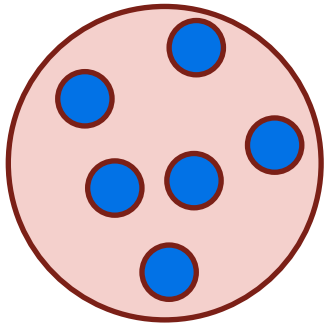
We'll denote $H(\mathcal{D})$ to map from a data subset $\mathcal{D}$ to the impurity score, by setting probability distribution $\approx$ distribution of labels Y in $\mathcal{D}$

Entropy of Binary Classes

Entropy $H(\mathcal{D}) = -\sum_c P(Y = c) \log_2 P(Y = c)$,
where different $c's$ correspond to different class labels

Min Impurity

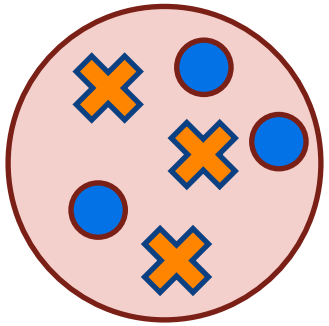
All instances in same class



$H(\mathcal{D}) = -1 \log 1 = 0$

Max Impurity

Instances split evenly among classes



$H(\mathcal{D}) = -0.5 \log 0.5 - 0.5 \log 0.5 = 1$

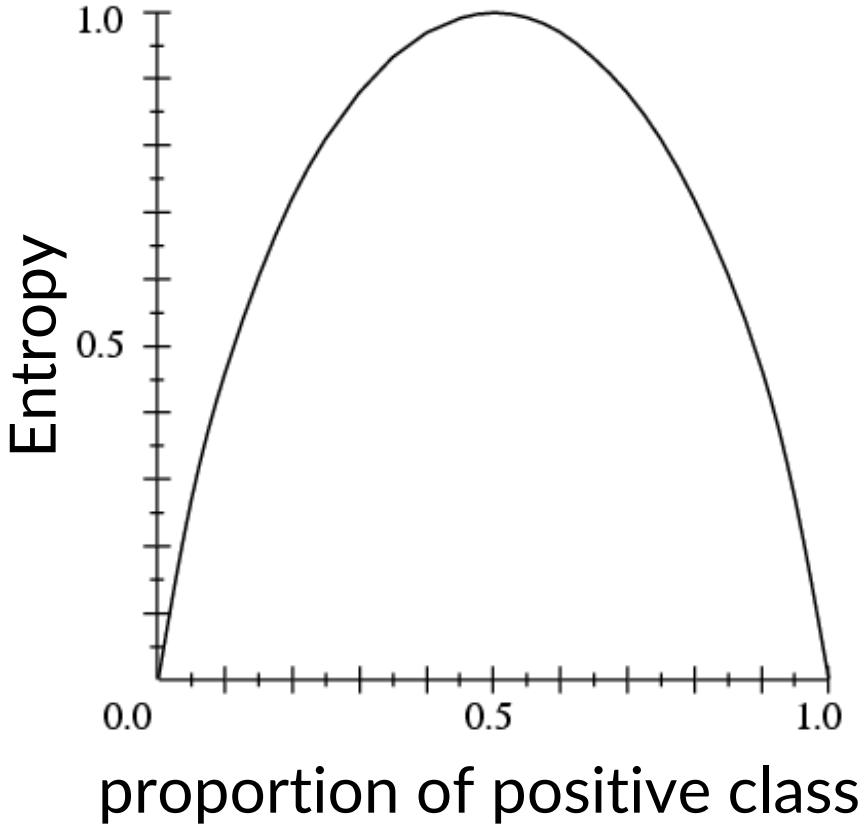
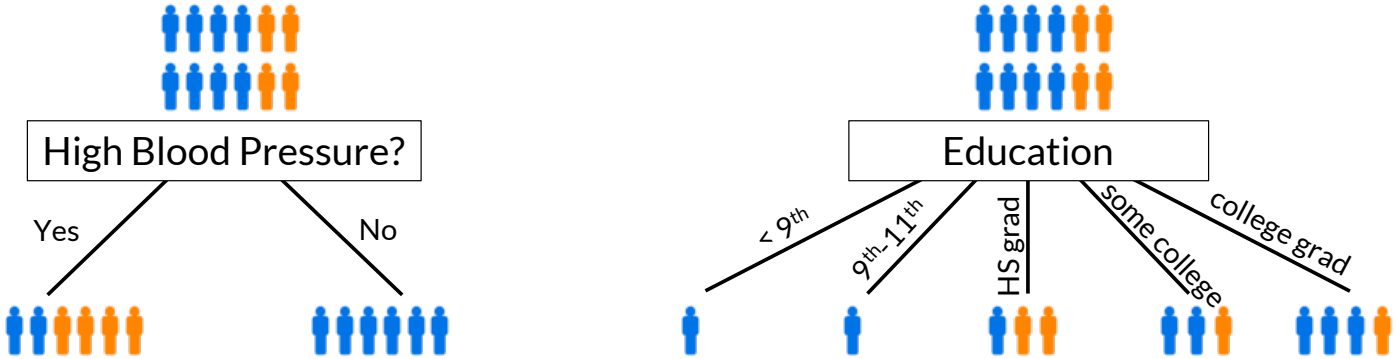


Image: Tom Mitchell

Choosing Features for Short Decision Trees



Recall: Ask questions such that the answers will reduce impurity in child nodes

When considering splitting on attribute / feature X_j,

- Need to estimate the "**expected** drop in impurity" after "getting the answer"/partitioning the data
- "Information Gain" based on our entropy function:

$$\mathrm{IG}(\mathcal{D}, X_j) = H(\mathcal{D}) - \sum_v H(\mathcal{D}[X_j = v])P(X_j = v)$$

Information Gain

Entropy $H(\mathcal{D}) = -\sum_c P(Y = c) \log_2 P(Y = c)$,
where different $c's$ correspond to different class labels

$$\text{IG}(\mathcal{D}, X_j) = H(\mathcal{D}) - \sum_v H(\mathcal{D}[X_j = v]) P(X_j = v)$$

The second term is sometimes called the "conditional entropy":

$$H(\mathcal{D}|X_j) = \sum_v H(\mathcal{D}[X_j = v]) P(X_j = v)$$

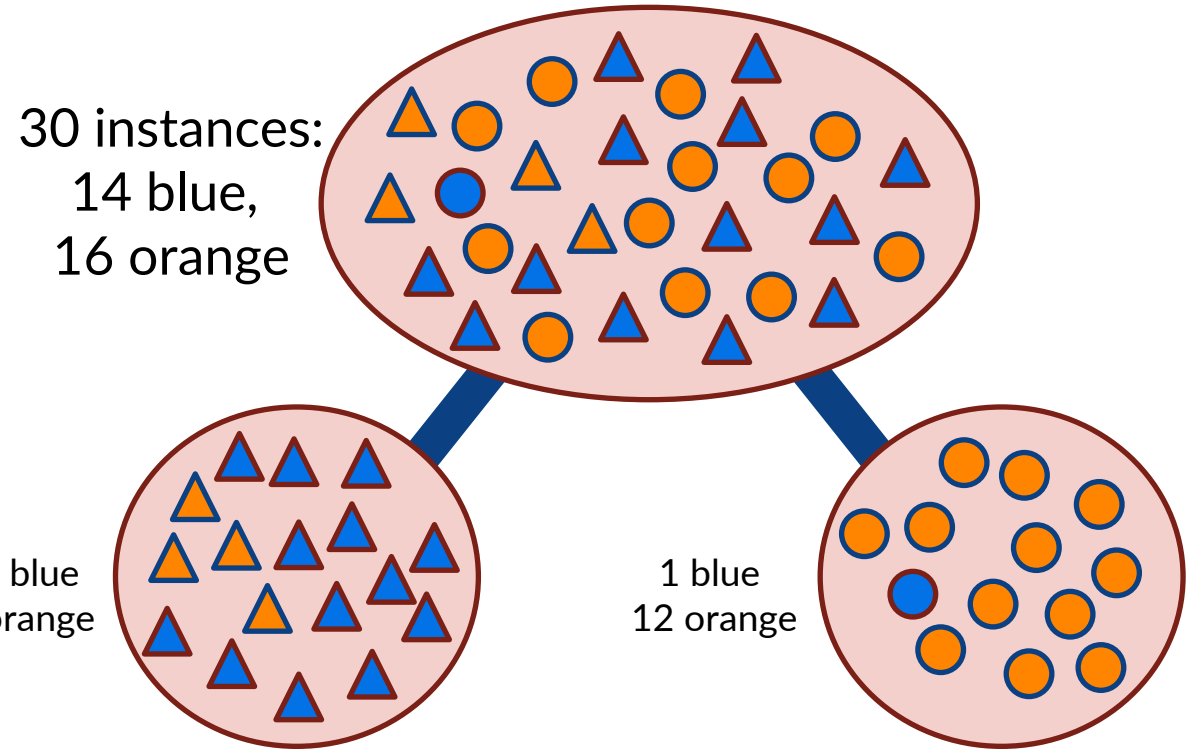
The information gain may then also be written as:

$$IG(\mathcal{D}, X_j) = H(\mathcal{D}) - H(\mathcal{D}|X_j)$$

Example IG Calculation

$$H(\mathcal{D}) = -\sum_c P(Y = c) \log_2 P(Y = c),$$

$$\text{IG}(\mathcal{D}, X_j) = H(\mathcal{D}) - \sum_v H(\mathcal{D}[X_j = v]) P(X_j = v)$$



H(parent) =

$$-\left(\frac{14}{30}\log_2\frac{14}{30}\right) - \left(\frac{16}{30}\log_2\frac{16}{30}\right)$$
$$= 0.996$$

weighted_mean(H(children)) =

$$\frac{17}{30}\cdot 0.787 + \frac{13}{30}\cdot 0.391$$
$$= 0.615$$

H(child)) =

$$-\left(\frac{13}{17}\log_2\frac{13}{17}\right) - \left(\frac{4}{17}\log_2\frac{4}{17}\right)$$
$$= 0.787$$

H(child)) =

$$-\left(\frac{1}{13}\log_2\frac{1}{13}\right) - \left(\frac{12}{13}\log_2\frac{12}{13}\right)$$
$$= 0.391$$

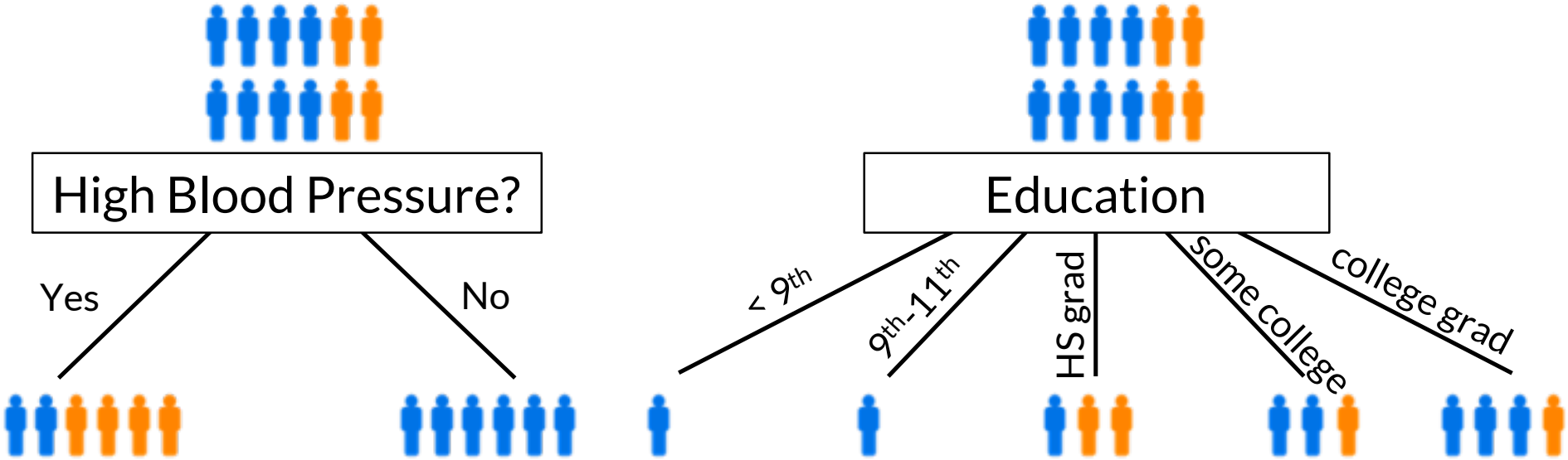
IG = 0.996 – 0.615 = **0.381**

Based on example by Pedro Domingos

Returning to the Diabetes Example Use Case

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no

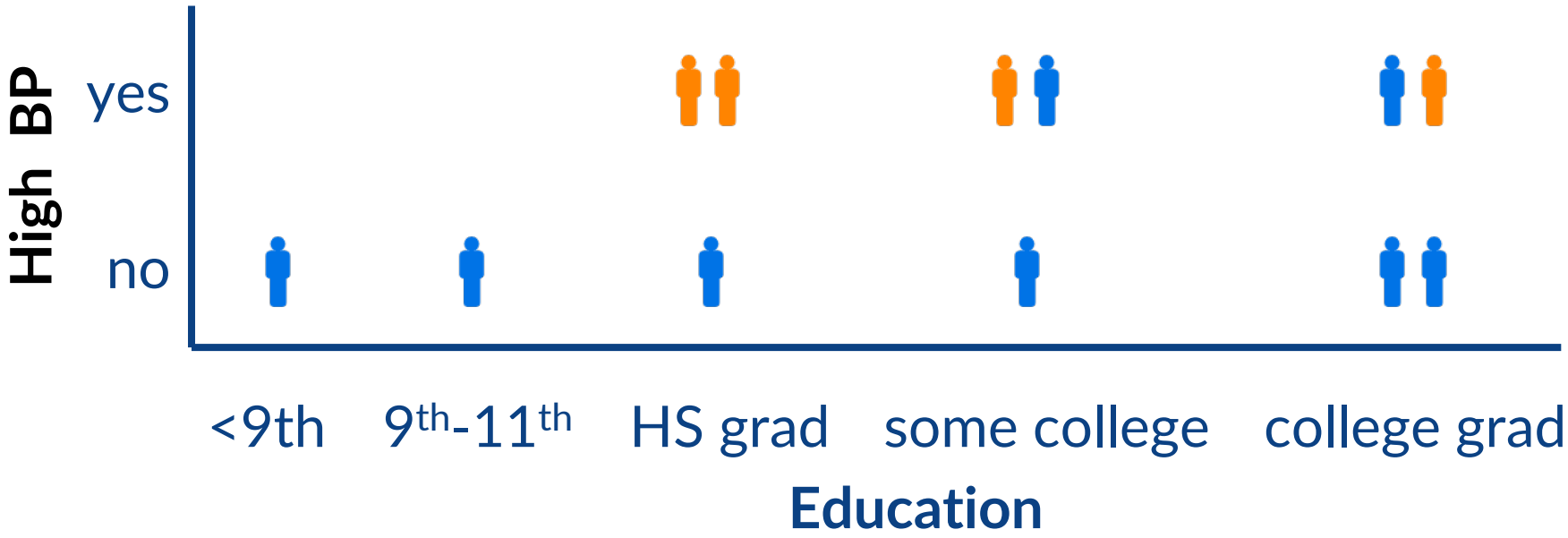
Which split is more informative?



Now we can solve it computationally via information gain

Information Gain Example for Diabetes

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no



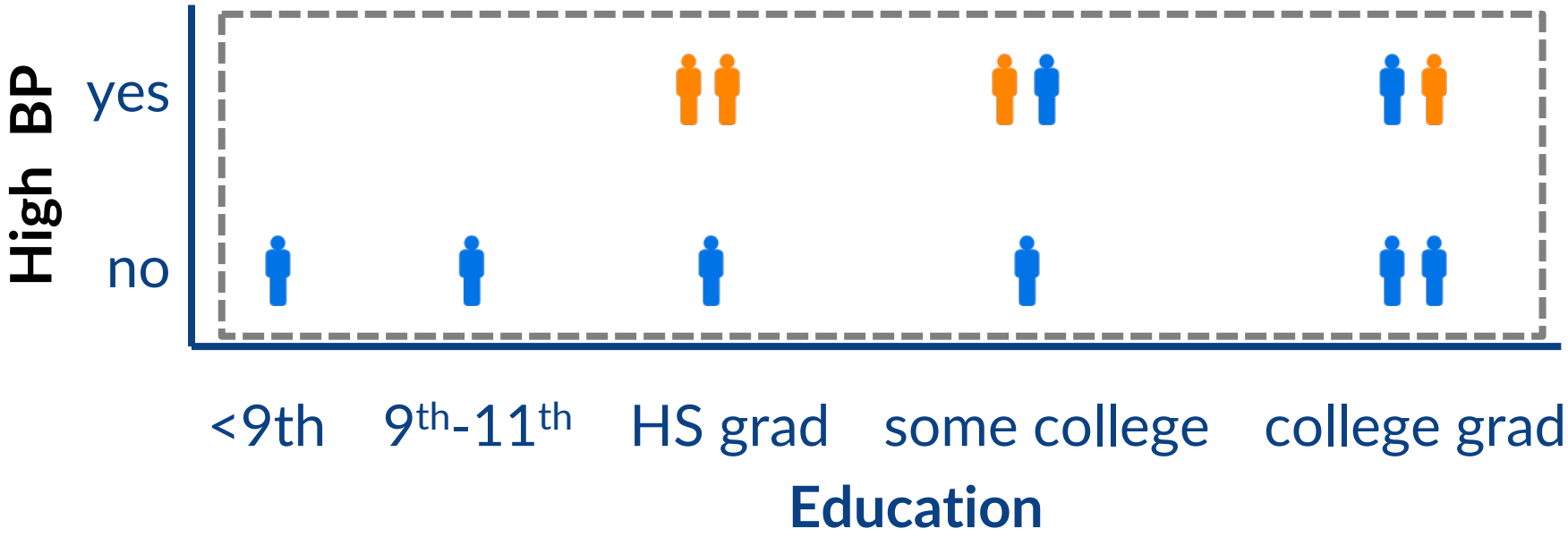
Need to compute:

$$IG(\mathcal{D}, High\ BP) = H(\mathcal{D}) - H(\mathcal{D}\,|High\ BP)$$

$$IG(\mathcal{D}, Education) = H(\mathcal{D}) - H(\mathcal{D}|\ Education)$$

Information Gain Example for Diabetes

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no



Need to compute:

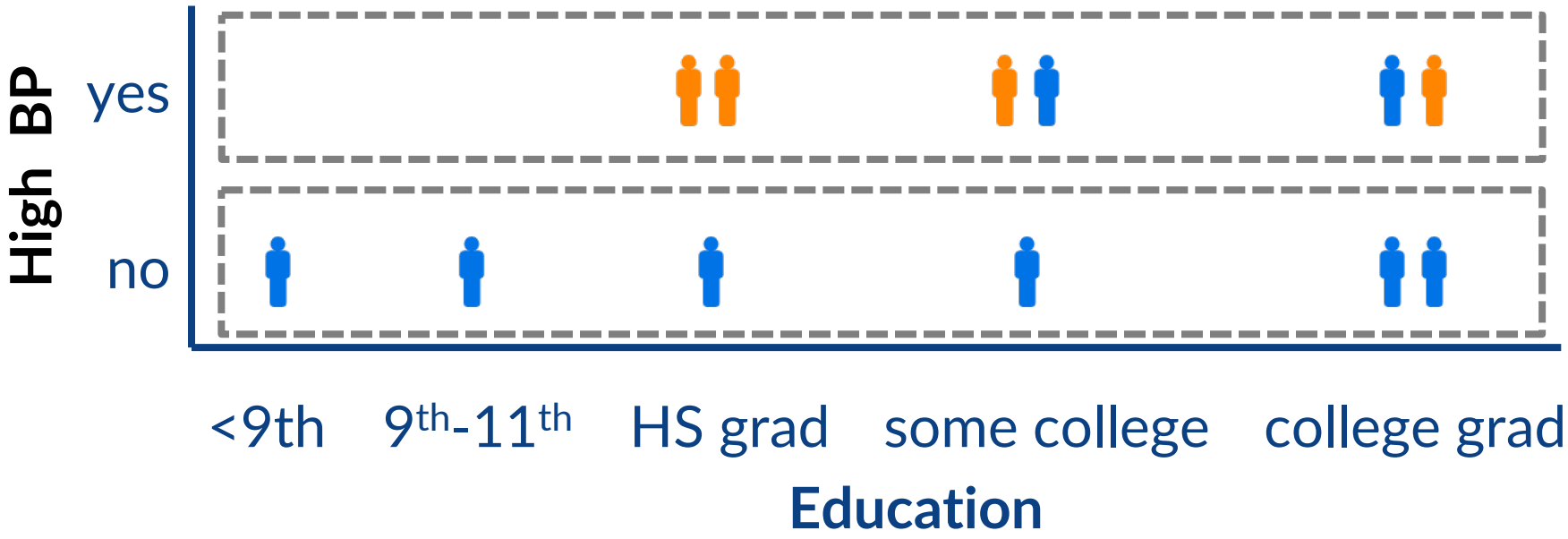
$$IG(\mathcal{D}, High\ BP) = H(\mathcal{D}) - H(\mathcal{D}\,|High\ BP)$$

$$IG(\mathcal{D}, Education) = H(\mathcal{D}) - H(\mathcal{D}|\ Education)$$

$H(\mathcal{D})$ = – 4/12 lg 4/12
– 8/12 lg 8/12
= 0.918

Information Gain Example for Diabetes

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no



Need to compute:

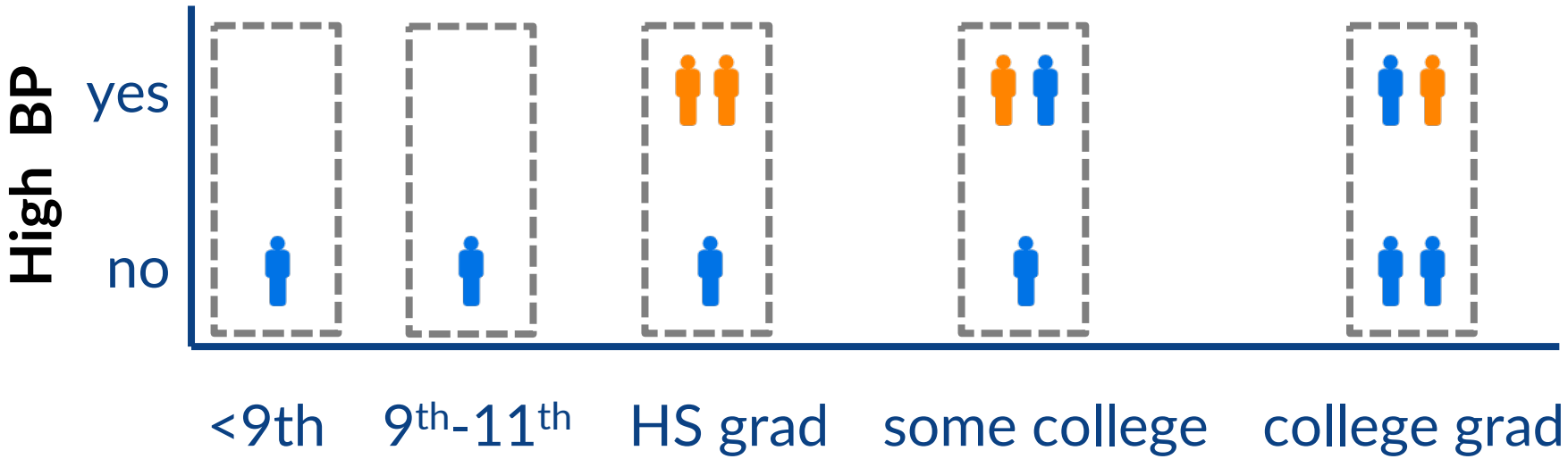
$$IG(\mathcal{D}, High\ BP) = H(\mathcal{D}) - H(\mathcal{D}\,|High\ BP)$$

$$IG(\mathcal{D}, Education) = H(\mathcal{D}) - H(\mathcal{D}|\ Education)$$

= (6/12) * (-2/6 lg 2/6
- 4/6 lg 4/6)
+ (6/12) * (0)
= 0.459

Information Gain Example for Diabetes

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no



Need to compute:

$$IG(\mathcal{D}, High\ BP) = H(\mathcal{D}) - H(\mathcal{D}\,|High\ BP)$$

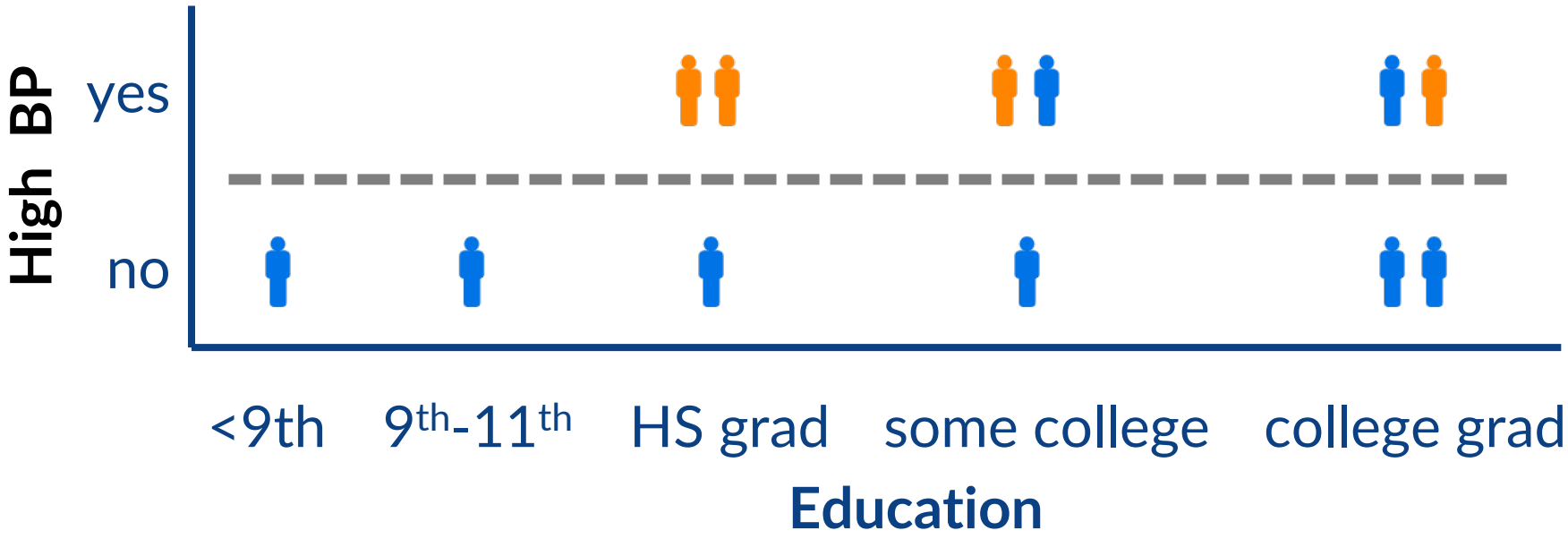
$$IG(\mathcal{D}, Education) = H(\mathcal{D}) - H(\mathcal{D}|\ Education)$$

= (1/12) * 0 + (1/12) * 0
+ (3/12) * (–1/3 lg 1/3
– 2/3 lg 2/3)
+ (3/12) * (–2/3 lg 2/3
– 1/3 lg 1/3)
+ (4/12) * (–3/4 lg 3/4
– 1/4 lg 1/4)

= 0.730

Information Gain Example for Diabetes

ID (SEQN)	HIGH_BP (BPQ020)	EDUCATION (DMDEDUC2)	DIABETIC
73557	yes	high school graduate / GED	yes
73558	yes	high school graduate / GED	yes
73559	yes	some college or AA degree	yes
73562	yes	some college or AA degree	no
73564	yes	college graduate or above	no
73566	no	high school graduate / GED	no
73567	no	9th-11th grade	no
73568	no	college graduate or above	no
73571	yes	college graduate or above	yes
73577	no	Less than 9th grade	no
73581	no	college graduate or above	no
73585	no	some college or AA degree	no



Need to compute:

$$IG(\mathcal{D}, High\ BP) = H(\mathcal{D}) - H(\mathcal{D}\,|High\ BP) = 0.918 - 0.459 = 0.459$$ **0.459**

$$IG(\mathcal{D}, Education) = H(\mathcal{D}) - H(\mathcal{D}|\ Education) = 0.918 - 0.730 = 0.188$$

Avoiding Overfitting

How can we avoid overfitting?

1. Acquire more training data
2. Remove irrelevant attributes (manual process, not always possible)
3. **Stop growing, e.g., when data split is not statistically significant**
4. **Grow full tree, then post-prune**

Try various tree hyperparameters (e.g., tree depth, splitting criterion, termination criterion) and pick the one with the best estimated generalization performance. How to estimate?

- Cross-validation
- Add a complexity penalty to performance measure e.g., training accuracy – average depth of leaf node

Based on Slide by Pedro Domingos

Stopping Growth

- Set a maximum **depth** to the decision tree (max_depth in Scikit-Learn)

- Set a minimum number of samples in a node, for us to split (e.g., 2) (min_samples_split in skl)
- Set a minimum number of samples in a leaf (min_samples_leaf)

(Again: we might use k-fold cross-validation to compare)

But alternatively, we can build "the perfect tree" and then **prune back**, based on validation set

Another Idea to Prevent Overfitting

A single decision tree can be prone to **overfitting** to the training data

What if we use **randomization** to create multiple decision trees, each a bit different:

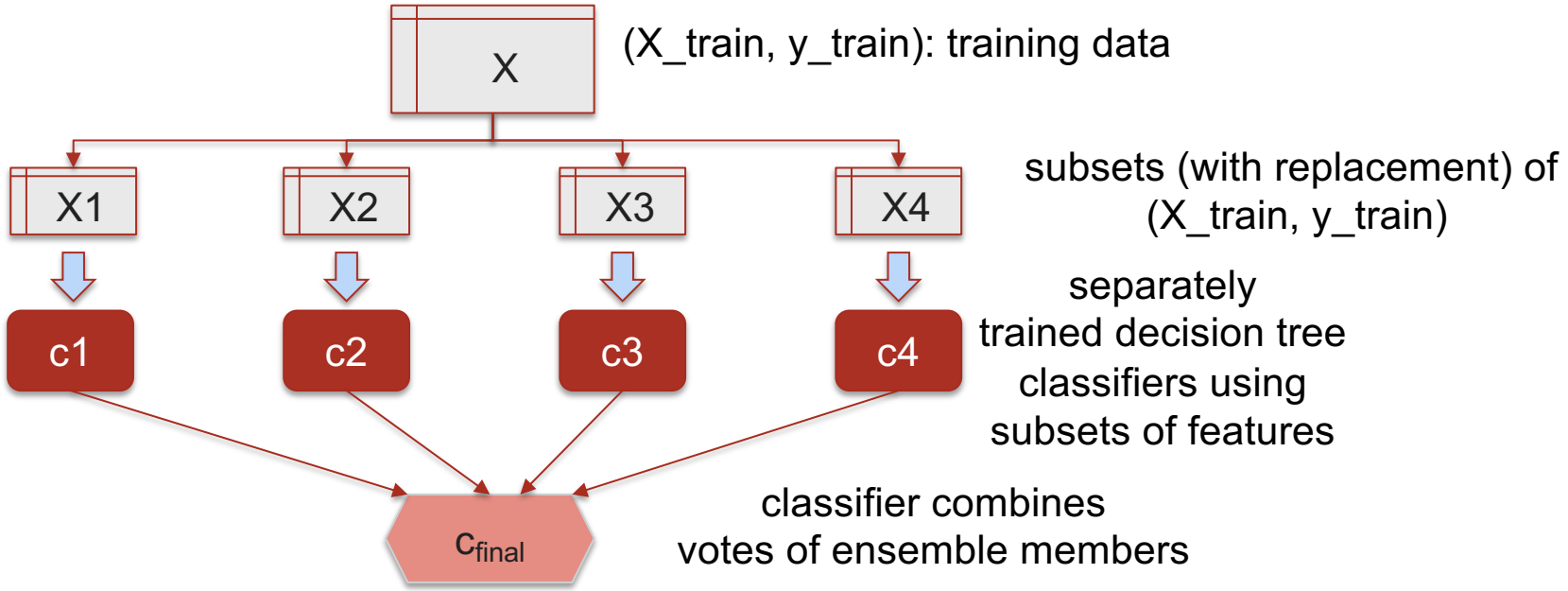
- Each is trained on a **sample** of the training data
- Each splits along a **subset** of the possible features
- Each is a small decision tree ("stump")

Then we rely on **voting** to make this work!

- Intuition: **the most predictive features** will be selected in many decision stumps!

(Note we now give up the "explainability" property)

Random Forests



Training a Decision Tree in a Random Forest

1. Draw a random **bootstrap** data sample of size *n*, with replacement
2. Build ("grow") a small decision tree (often just a "stump")
    - At each split point node, randomly select *d* candidate features (w/o replacement)
    - Split the node using the feature with best split according to objective function (e.g. information gain)
3. Repeat to produce *k* decision trees (a forest!)

4. For prediction, use **majority vote** to predict a class for new data

Benefits of Random Forests

One of the most popular and accurate classifiers for big data (more in a moment)

- Scale-invariant
- Much less susceptible to overfitting than "plain" decision trees
- Can be generalized to continuous data (random forests of CaRT trees)

Also: training is highly parallelizable!

- Take a data set, draw samples of size *n* with replacement
- Train a separate decision tree on this, at each split point selecting from a subset of the features (without replacement **for this tree**)

Let's see a case study…

Strategies for Increasing Model Capacity

- **Approaches so far:**
  - Richer model family
  - Feature engineering

- **Today:** Ensembles
  - Increase capacity of existing, low capacity models (e.g., decision trees)
  - Helps avoid overfitting

Ensemble Learning

- **Step 1:** Learn a set of "base" models $f_1, \ldots, f_k$
- **Step 2:** Construct model $F(x)$ that combines predictions of $f_1, \ldots, f_k$

Example: Netflix Movie Recommendations

- **Goal:** Predict how a user will rate a movie based on:
  - The user's ratings for other movies
  - Other users' ratings for this movie (and others)
  - **No features!**

- **Netflix Prize (2007-2009):** $1 million for the first team to do 10% better than the existing Netflix recommendation system

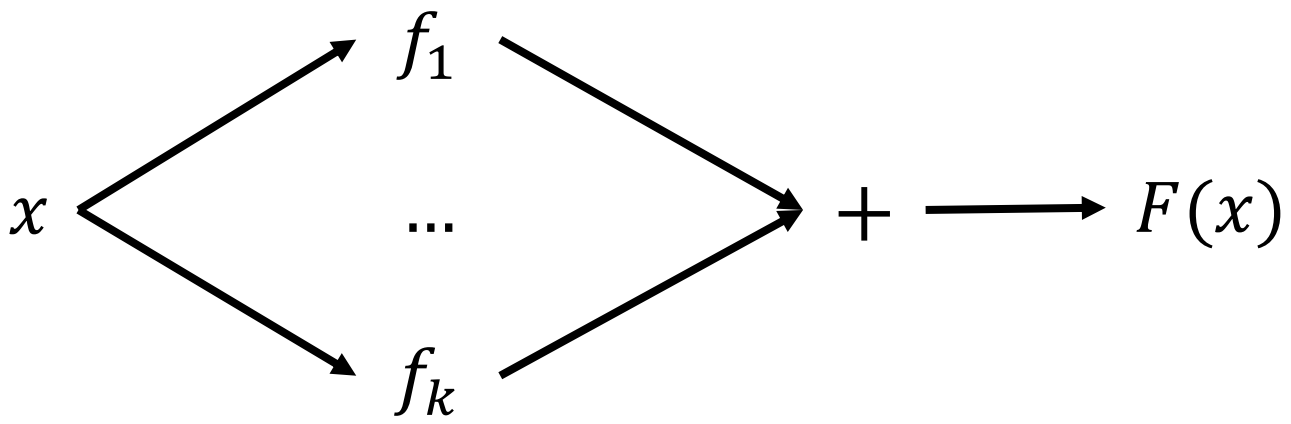
- **Winner:** BellKor's Pragmatic Chaos
  - An ensemble of 800+ rating systems

Ensembles of Decision Trees

- **Strategy 1:** Random forests
- **Strategy 2:** Gradient boosted decision trees
- Among the most powerful and widely-used models for “tabular” data (i.e., not images, text, graphs, or other highly structured data)

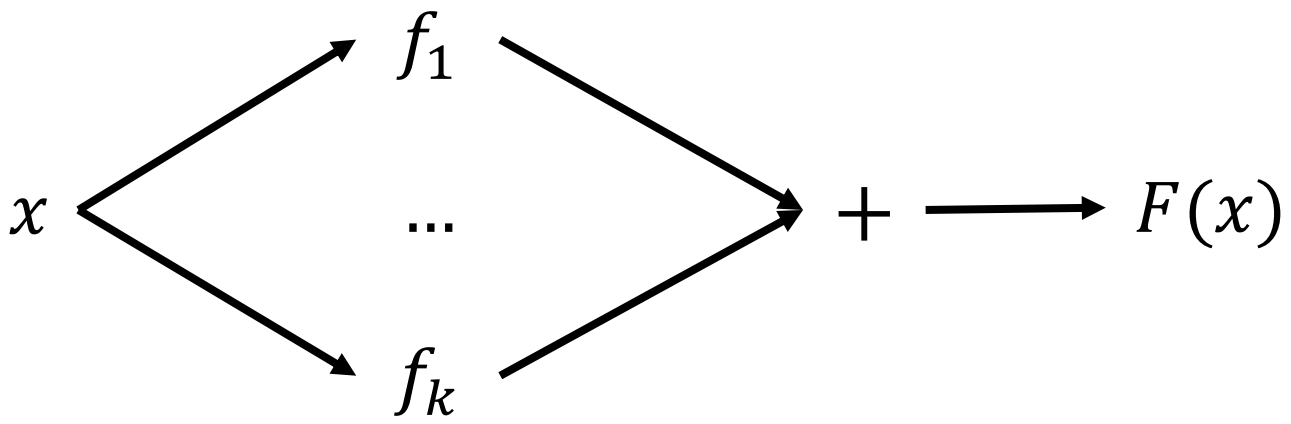
Combining Learned Base Models

- **Regression:** Average predictions $F(x) = \frac{1}{k}\sum_{i=1}^{k} f_i(x)$
  - Works well if the base models have similar performance



Combining Learned Base Models

- **Classification:** Majority vote $F(x) = 1\left(\sum_{i=1}^{k} f_i(x) \geq \frac{k}{2}\right)$ (for binary)
  - Can also average probabilities for classification



Combining Learned Base Models

- Can use weighted average:

$$F(x) = \sum_{i=1}^{k} \beta_i \cdot f_i(x)$$

- Can fit weights using linear regression on second training set

- More generally, can fit a second layer model

$$F(x) = g_\beta\big(f_1(x), \ldots, f_k(x)\big)$$

Combining Learned Base Models

- Second model as “mixture of experts”:

$$F(x) = \sum_{i=1}^{k} g(x)_i \cdot f_i(x)$$

- Second stage model predicts weights over “experts” $f_i(x)$

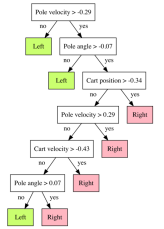
Learning Base Models

- Successful ensembles require **diversity**
  - Different model families
  - Different training data
  - Different features
  - Different hyperparameters

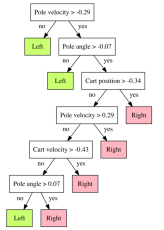
- **Intuition:** Models should make **independent** mistakes

Learning Base Models

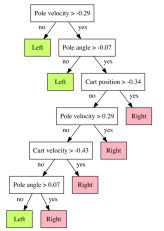
- **Intuition:** Models should make **independent** mistakes



$\text{acc} = \frac{3}{4}$

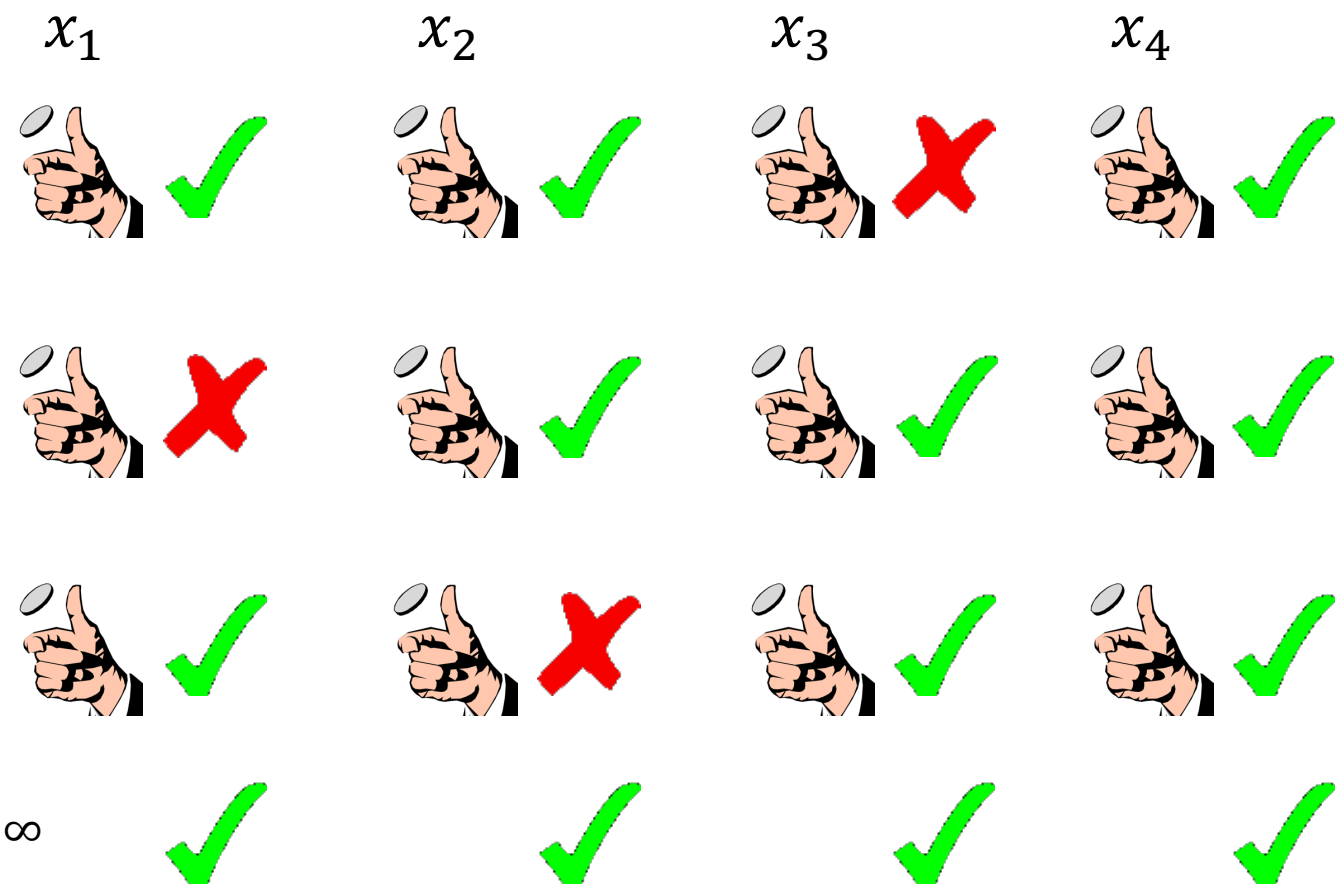


$\text{acc} = \frac{3}{4}$



$\text{acc} = \frac{3}{4}$

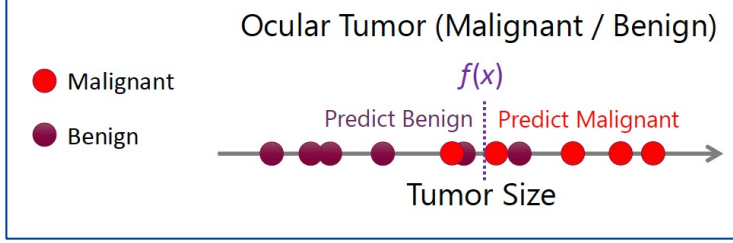
F $\quad \text{acc} \to 1 \text{ as } k \to \infty$

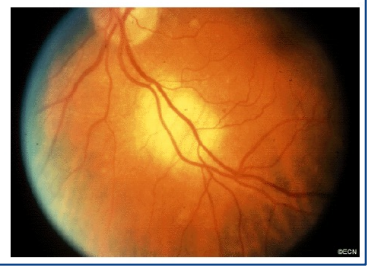


From Supervised to Unsupervised Learning

Supervised Learning: Classification

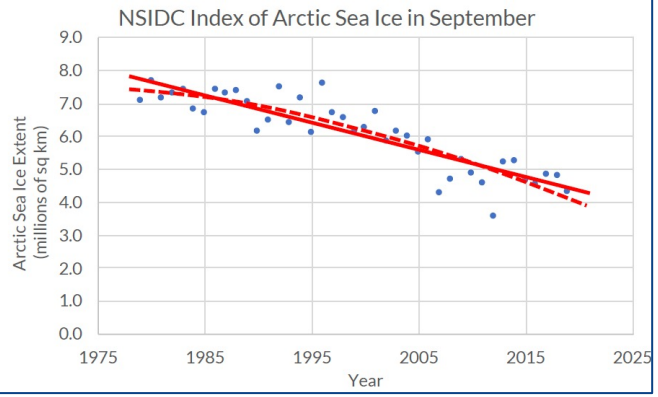
- Given (x_1, y_1), (x_2, y_2), …, (x_n, y_n)
- Learn a function $f(x)$ to predict y given x
  - y is categorical == classification





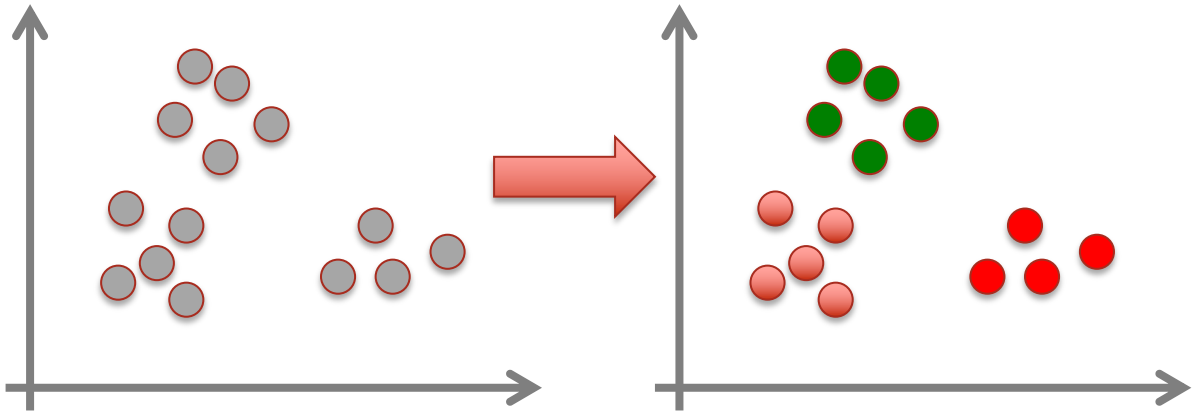
Supervised Learning: Regression

- Given (x_1, y_1), (x_2, y_2), …, (x_n, y_n)
- Learn a function $f(x)$ to predict y given x
  - y is numeric == regression



Given $x_1, x_2, \ldots, x_N$ (without labels)

Learn "hidden structure" in the data



Types and Uses of Unsupervised Learning

Dimensionality Reduction

Map samples $\boldsymbol{x}_i \in \mathbb{R}^D$ to $f(\boldsymbol{x}_i) \in \mathbb{R}^{D' \ll D}$

Special case: clustering, $f(\boldsymbol{x}_i) \in \mathbb{N}$

- **Feature Learning:** For preprocessing inputs to an ML algorithm, since lower-dimensional features permit smaller models and fewer data samples.
- **Compression (for storage):** e.g. JPEG standard for images is now adopting unsupervised ML approaches https://jpeg.org/items/20190327_press.html
- **Visualization:** Exploring a dataset, or an ML model's outputs

Popular Clustering Algorithms

1. One of the simplest, scalable techniques: k-Means clustering

2. Greedy: hierarchical clustering

The Clustering Setting

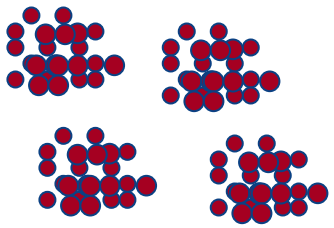
Task:

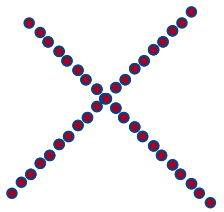
Input: $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^{N}$

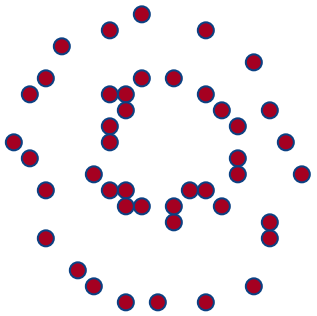
Want to discover a mapping $f(\boldsymbol{x}_i) \in \{1, 2, 3, \ldots, K\}$ that discovers natural groupings in the data.

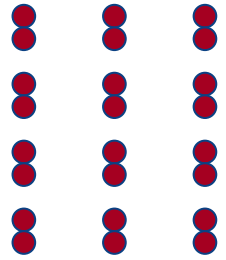
Performance Metric / Objective Function: What is a good mapping $f(.)$?

Somewhat loosely defined, and different clustering algorithms differ in their definition of a good clustering $f(.)$



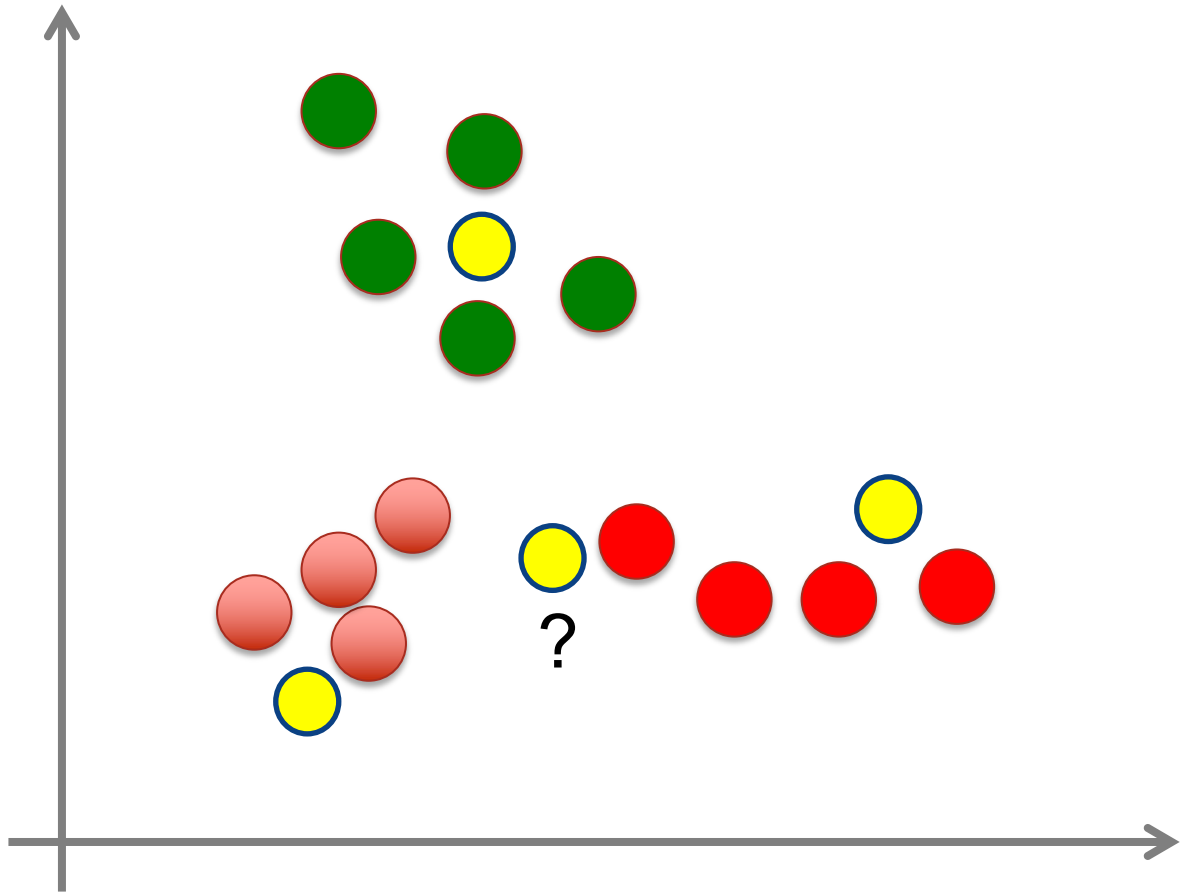






Figures: Dan Roth

Guess The Cluster Assignment

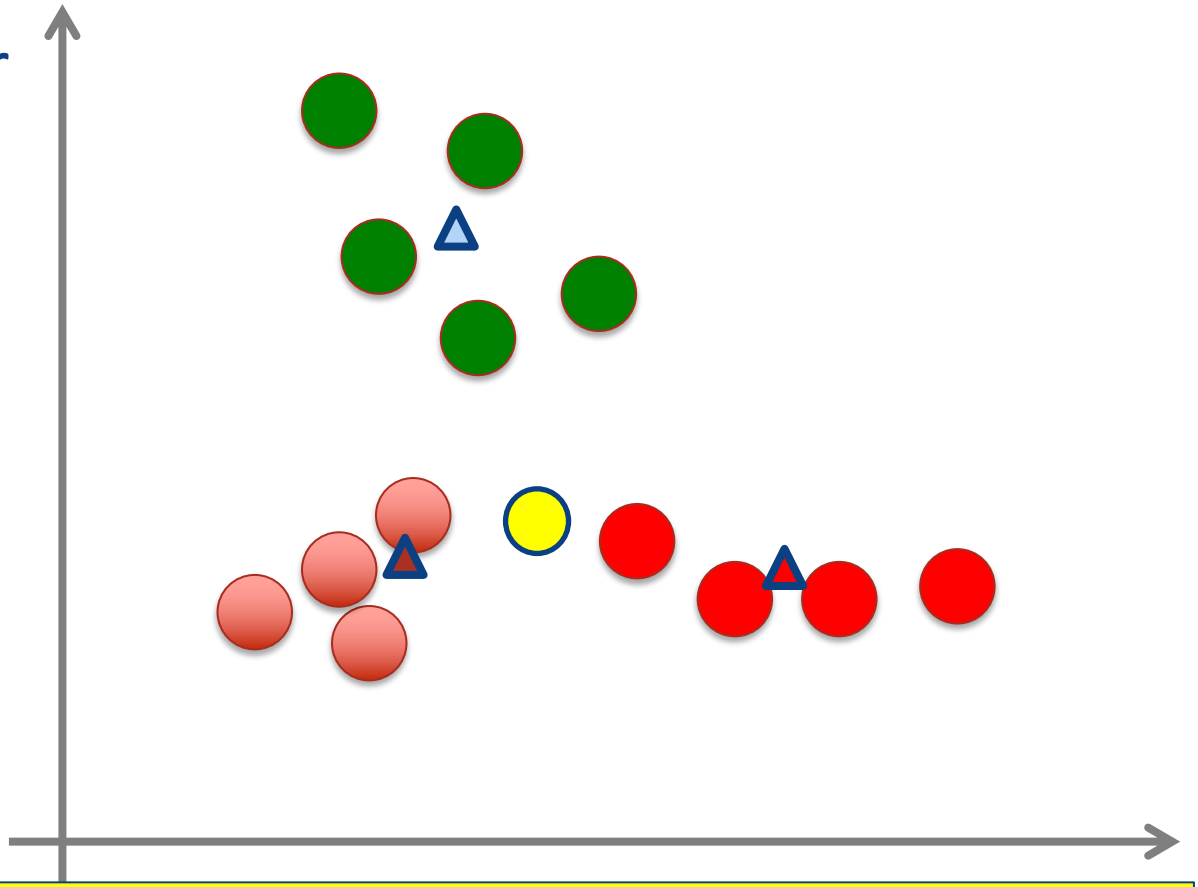


Cluster Assignment in K-Means

- Define a centroid μ_k for each cluster k
- For any new sample, assign the cluster whose centroid/mean is closest!

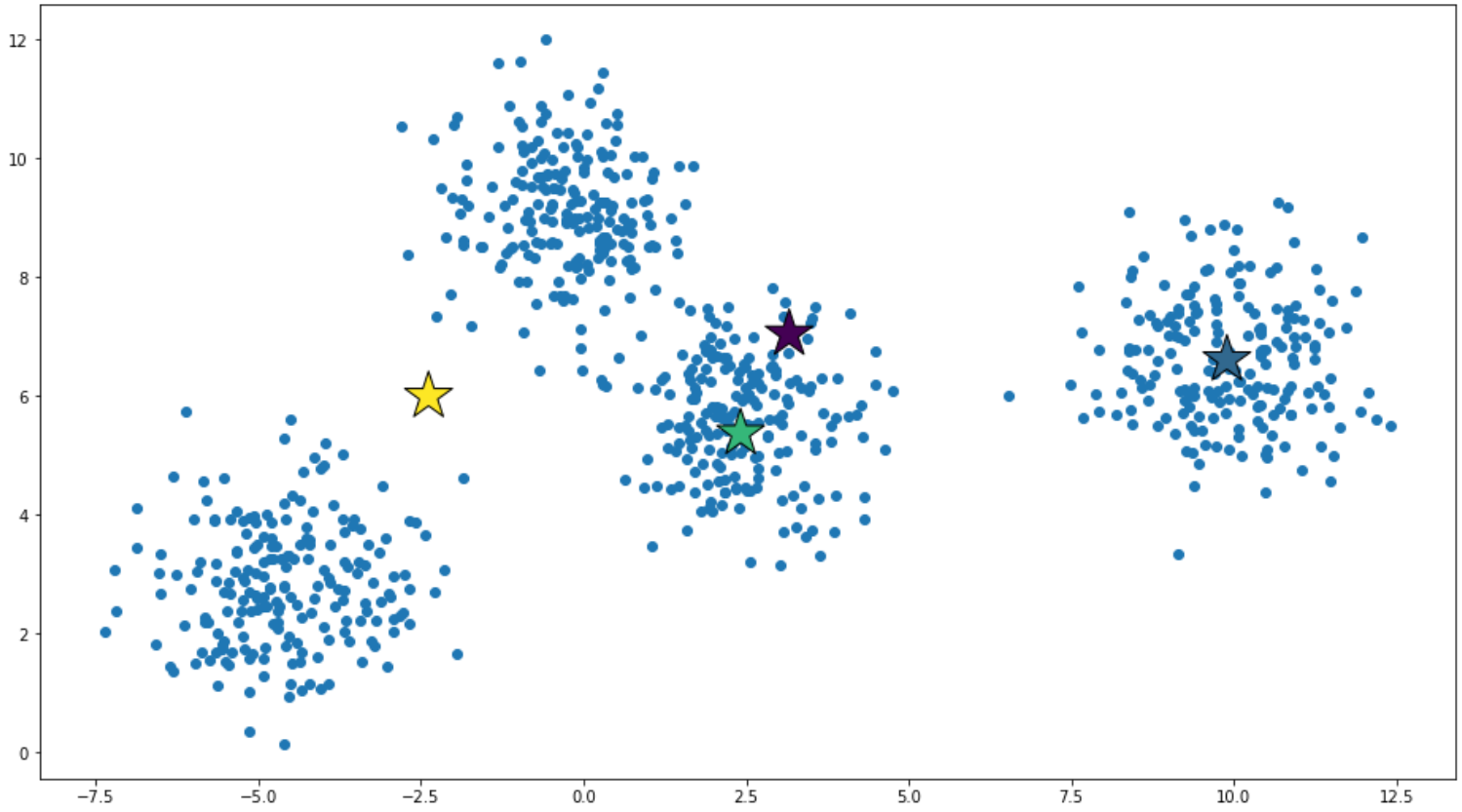
Equivalent to 1-nearest neighbor classification over the cluster means.

Note: Certainly not the only answer we could have come up with!

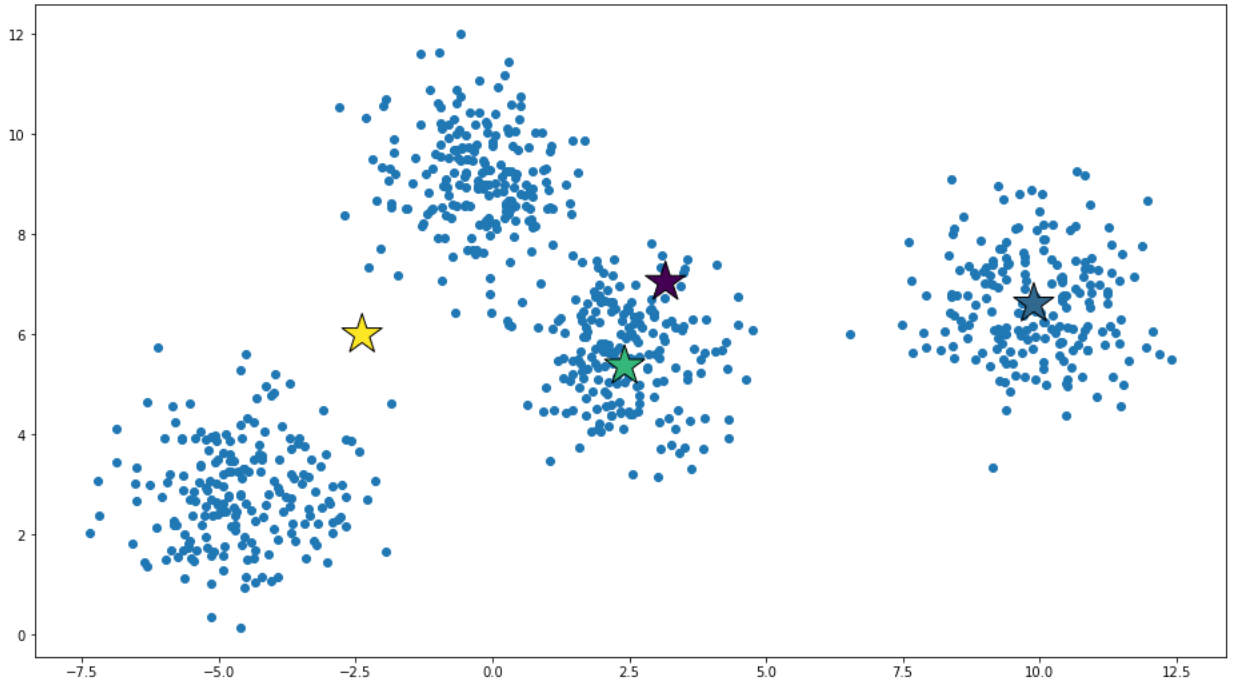


Can we expand this into a full, consistent clustering algorithm?

Clustering Data



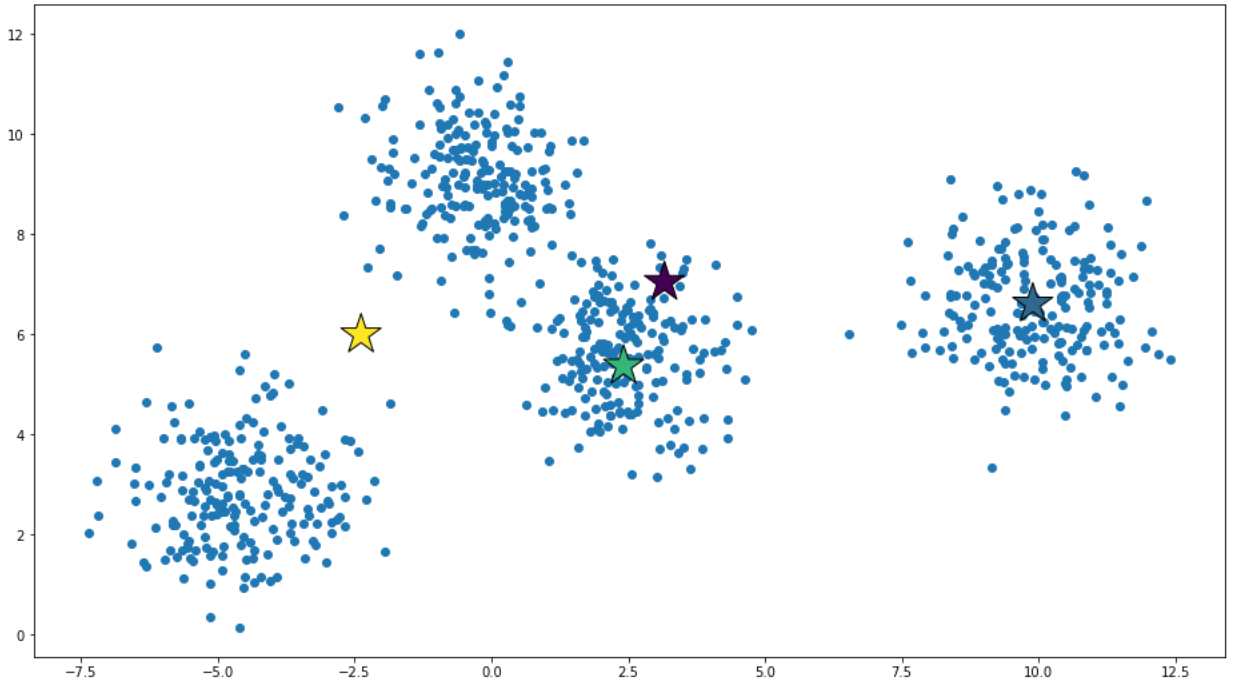
Clustering Data



K-Means Clustering

K-Means (K, X)

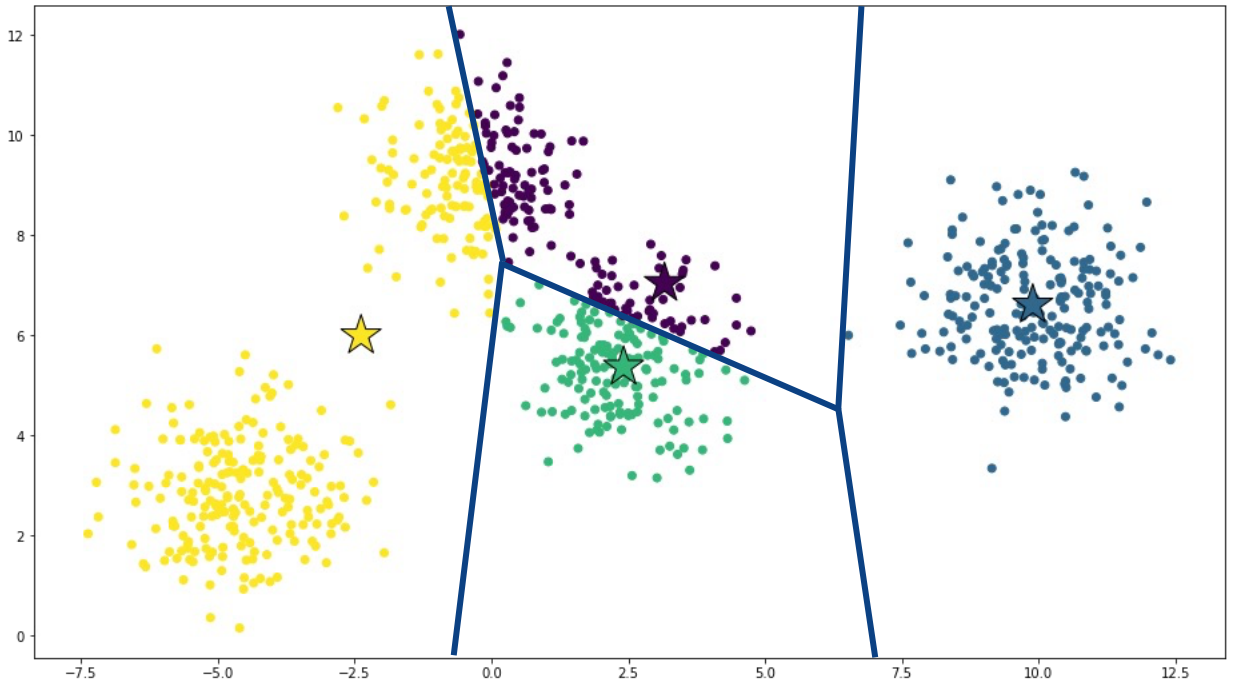
- Randomly choose K cluster means
- Loop until convergence, do:
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster



K-Means Clustering

K-Means (K, X)

- Randomly choose K cluster center locations (centroids)
- Loop until convergence, do:
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster

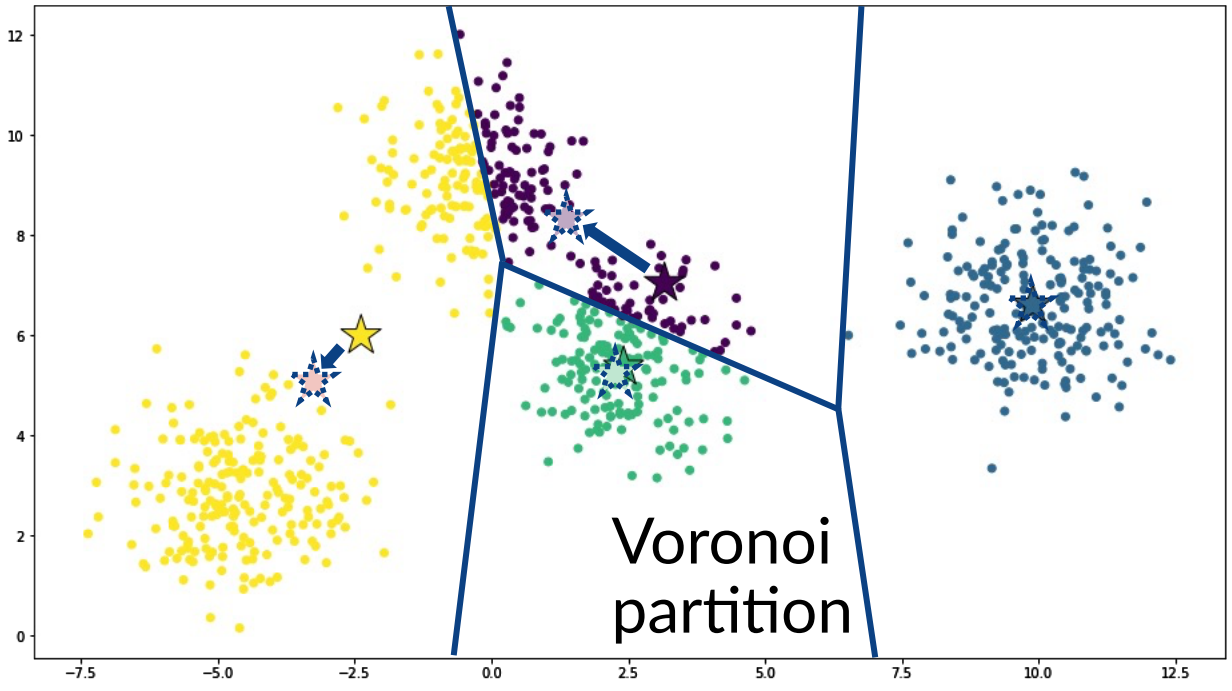


Voronoi partition

K-Means Clustering

K-Means (K, X)

- Randomly choose K cluster center locations (centroids)
- Loop until convergence, do:
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster



K-Means Clustering

K-Means (K, X)

- Randomly choose K cluster center locations (centroids)
- Loop until convergence, do:
  - Assign each point to the cluster of the closest centroid
  - Re-estimate the cluster centroids based on the data assigned to each cluster

Optimizer: "Alternating Minimization"

K-means finds a local optimum of the following objective function:

"Sum of squared distances" loss function

$$\arg\min_{\boldsymbol{S}} \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in S_k} \|\boldsymbol{x} - \boldsymbol{\mu}_K\|_2^2$$

where $\boldsymbol{S} = \{S_1, \ldots, S_K\}$ are sets corresponding to disjoint clusters, and the clusters together include all samples.

K-Means is Too Sensitive to Initialization

Alternative strategies:

1. Do many runs of K-Means, each with different initial centroids, and pick the best
2. Pick initial centroids using a better method than random choice

K-means+ + initialization

- Choose a data point uniformly at random as the first centroid
- Loop for $2{:}K$, do:
  - Let $D(\boldsymbol{x})$ be the distance from each point $\boldsymbol{x}$ to the closest centroid
  - Choose data point $\boldsymbol{x}$ randomly $\propto D(\boldsymbol{x})^2$ as the next centroid. Higher chance to pick points that are far from previous centroids.

UNIVERSITY OF KWAZULU-NATAL ™
INYUVESI YAKWAZULU-NATALI

NITheCS
National Institute for Theoretical and Computational Sciences

Dimensionality Reduction

The Next Key Question: How Do We Get the "Best" Features?

For a variety of reasons, we may want to reduce the number of dimensions:

- Reduce the **complexity** of our **learning problem**
- Remove **multicollinearity** / correlated features
- Remove **less informative features** (results in simpler model)
- **Visualize** the features

Key problem: mapping from D-dimensions down to a D'-dimensional subspace (D' << D)

Dimensionality Reduction Objective

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1D} \\ \vdots & \ddots & \vdots \\ x_{N1} & \cdots & x_{ND} \end{bmatrix}_{N \times D}$$

We can write each row (each data sample) $\boldsymbol{x}_i$ as:

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \end{bmatrix}_D = x_{i1} \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix}_D + x_{i2} \begin{bmatrix} 0 \\ 1 \\ \vdots \end{bmatrix}_D + \cdots + x_{iD} \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix}_D$$

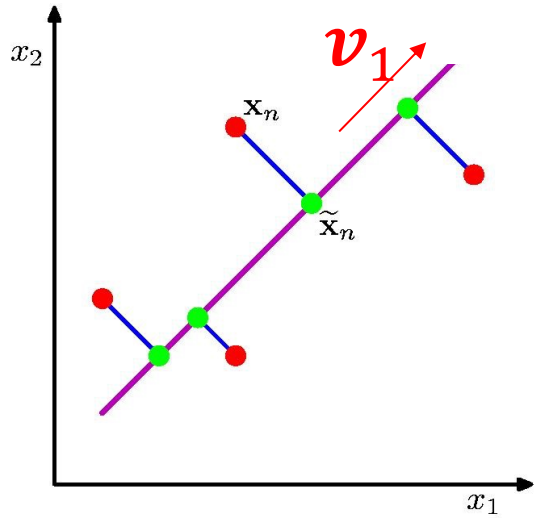
Projections Original axes

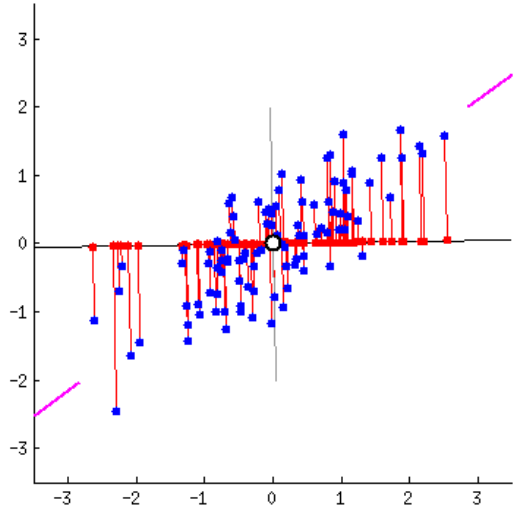
We are looking for a new coordinate system to (approximately) express $\boldsymbol{x}_i$:

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \end{bmatrix} \approx f_1(\boldsymbol{x}_i)\boldsymbol{v}_1 + f_2(\boldsymbol{x}_i)\boldsymbol{v_2} + \cdots + f_{D'}(\boldsymbol{x}_i)\boldsymbol{v}_{D'}$$

where the new axes $\boldsymbol{v}_d$'s are all D-dimensional unit norm, and $D' \ll D$

Principal Component Analysis





(Fig: stats.stackexchange)

Orthogonal projection of data onto lower-dimension linear space :

- maximizes variance of projected data (purple line)
- minimizes mean squared distance between data point and projections (sum of blue lines)

Based on slide by Barnabás Póczos, UAlberta

Reduce to $D' = 1$ dimension?

We are looking for a new coordinate system to (approximately) express $\boldsymbol{x}_i$:

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \end{bmatrix} \approx (x_i . v_1)\boldsymbol{v}_1 + (x_i . v_2)\boldsymbol{v_2} + \cdots + (x_i . v_{D\prime})\boldsymbol{v}_{D'}$$

where the new axes $\boldsymbol{v}_d$'s are all D-dimensional unit norm, and $D' \ll D$

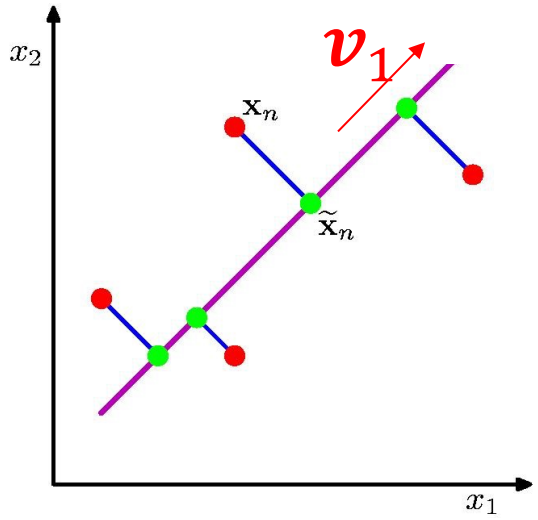
Simplest case: $D' = 1$?

We want to find $\boldsymbol{v}_1$ and $f_1(\boldsymbol{x}_i)$ such that:

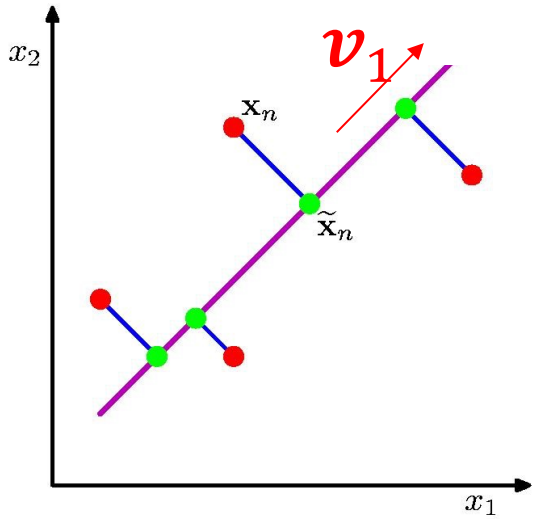
$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \end{bmatrix} \text{ best approximates } f_1(\boldsymbol{x}_i)\boldsymbol{v}_1$$

For a given $\boldsymbol{v}_1$, $f_1(\boldsymbol{x}_i)$ should be the projection $\boldsymbol{x}_i . \boldsymbol{v}_1 / \|\boldsymbol{v}_1\|_2$

So, only need to find $\boldsymbol{v}_1$



Objective Function: Maximizing Variance



Find unit vector $\boldsymbol{v}_1$ (with $\|\boldsymbol{v}_1\|_2 = 1$), to optimize:

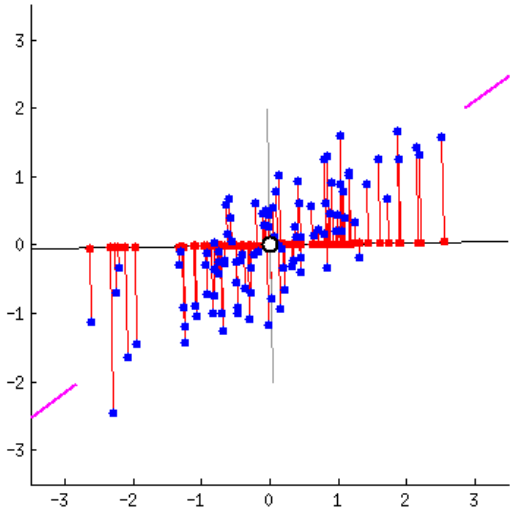
Reconstruction MSE

$$\min_{\|\boldsymbol{v}_1\|_2=1} \frac{1}{N} \sum_i \|(\boldsymbol{x}_i \cdot \boldsymbol{v}_1)\boldsymbol{v}_1 - \boldsymbol{x}_i\|_2^2$$

Projection error

Can show, exactly equal to:

$$\max_{\|\boldsymbol{v}_1\|_2=1} \text{variance}(\boldsymbol{x}_i \cdot \boldsymbol{v}_1)$$



(Fig: stats.stackexchange)

Intuitively, if the variance of the projection on $\boldsymbol{v}_1$ was low, then $\boldsymbol{v}_1$ would not be very informative about samples $\boldsymbol{x}_i$.

Conversely, directions with high variance projections preserve the most information.

So, how to find this direction of maximum variance?

Covariance Matrix

$$\max_{\|\boldsymbol{v}_1\|_2=1} \text{variance}(\boldsymbol{x}_i \cdot \boldsymbol{v}_1)$$

For zero-centered data,

$$\text{Covariance} = C = \mathbb{E}\left[\boldsymbol{x}_i \boldsymbol{x}_i^T\right] = \mathbb{E}\begin{bmatrix} x_{i1}x_{i1} & \cdots & x_{i1}x_{iD} \\ \vdots & x_{ij}x_{ik} & \vdots \\ x_{iD}x_{i1} & \cdots & x_{iD}x_{iD} \end{bmatrix}$$

For any unit vector v_1,

$$\text{variance}(\boldsymbol{x}_i \cdot \boldsymbol{v}_1) = \boldsymbol{v}_1^T C \boldsymbol{v}_1$$

To maximize $\boldsymbol{v}_1^T C \boldsymbol{v}_1$, we can set $\boldsymbol{v}_1 = \boldsymbol{e}_1(C)$, the first unit eigenvector of C

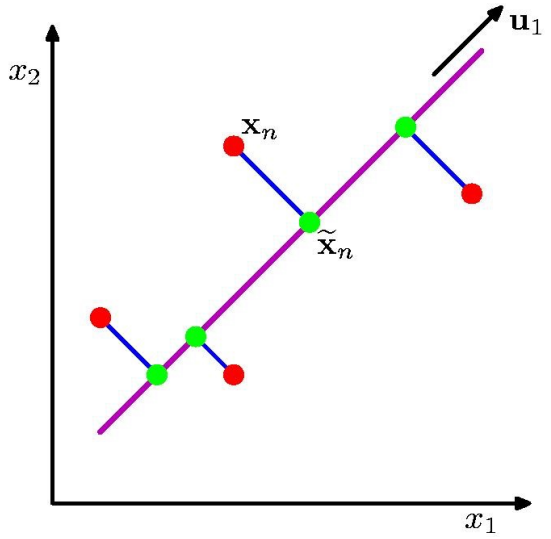
More than 1 dimension?

We are looking to find a new way to express $\boldsymbol{x}_i$: "Reconstruction"

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \end{bmatrix} \approx f_1(\boldsymbol{x}_i)\boldsymbol{v_1} + f_2(\boldsymbol{x}_i)\boldsymbol{v_2} + \cdots + f_{D'}(\boldsymbol{x}_i)\boldsymbol{v}_{D'}$$

Repeat for $d = 1, \ldots, D'$

- Subtract means of all dimensions of X
- Compute $C_d = E[x_i x_i^T]$
- Set $\boldsymbol{v}_d = \boldsymbol{e}_1(C_d)$
- Set $\boldsymbol{x}_i = \boldsymbol{x}_i - (\boldsymbol{x}_i \cdot \boldsymbol{v}_d)\boldsymbol{v}_d$ (i.e., subtract current reconstructions to compute residuals… like gradient boosting!)



Equivalent to simply:

Repeat for $d = 1, \ldots, D'$

- Set $v_d = \boldsymbol{e}_d(C_1)$

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{iD} \end{bmatrix} \approx \sum_{d=1}^{D'} (\boldsymbol{x}_i . \boldsymbol{v}_d)\boldsymbol{v}_d$$

So, the new low-dimensional representation is:

$$f(\boldsymbol{x}_i) = [\boldsymbol{x}_i \cdot \boldsymbol{v}_1, \boldsymbol{x}_i \cdot \boldsymbol{v}_2, \ldots, \boldsymbol{x}_i \cdot \boldsymbol{v}_{D'}]$$

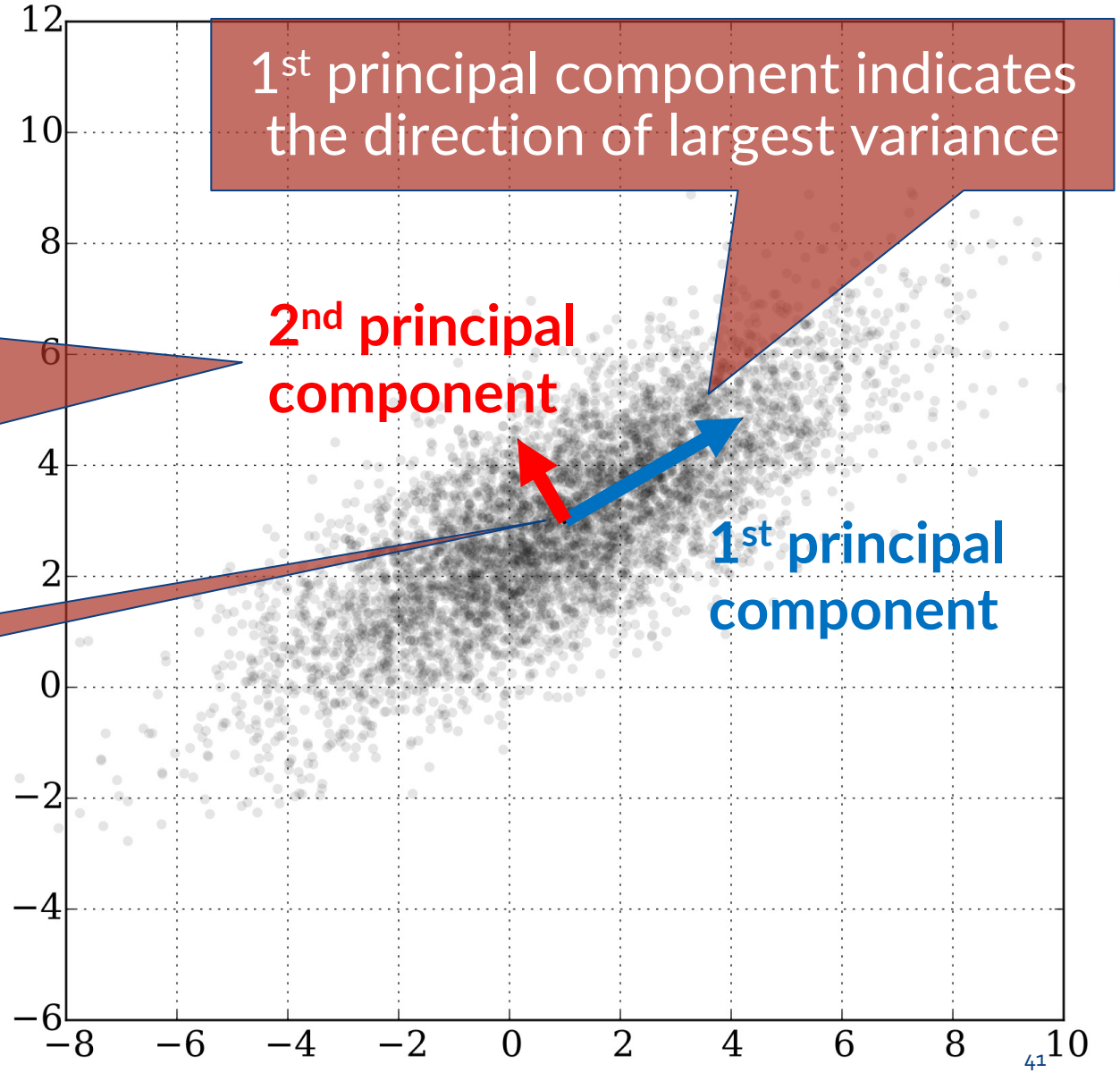
PCA on a 2D Gaussian Dataset

Each subsequent principal component:

- is orthogonal to all previous components
- indicates the direction of largest variance of the residuals

1st principal component indicates the direction of largest variance

Basis vectors originate from the mean



PCA Algorithm

Given data $\{\boldsymbol{x}_1, \ldots \boldsymbol{x}_n\}$, compute covariance matrix C

- $\boldsymbol{X}$ is the $N \times D$ data matrix
- Compute data mean (average over all rows of $\boldsymbol{X}$)
- Subtract mean from each row of $\boldsymbol{X}$ (centering the data)
- Compute covariance matrix $\mathrm{C} = \boldsymbol{X}^T \boldsymbol{X}$ (C is $D \times D$)

PCA basis vectors (new coordinate axes) are given by the eigenvectors of C

- Q, Λ = numpy.linalg.eig(C)
- $\{\boldsymbol{q}_d, \lambda_d\}_{d=1,\ldots,D}$ are the eigenvectors/eigenvalues of C

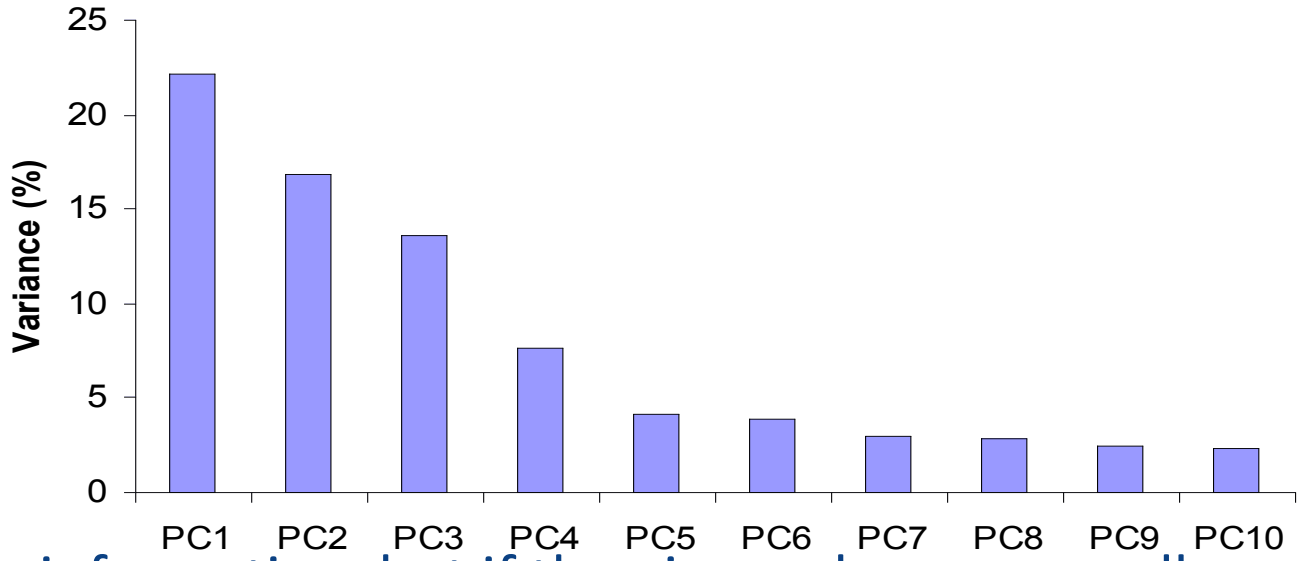
$$(\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_D)$$

But there are D eigenvectors, so where is the dimensionality reduction?

- A: Larger eigenvalue $\Rightarrow$ "more important" eigenvectors

Dimensionality Reduction

- Can *ignore* the components of lesser significance



- You do lose some information, but if the eigenvalues are small, you don't lose much
  - choose only the first D' eigenvectors, based on their eigenvalues
  - final data set has only D' dimensions

Based on slide by Barnabás Póczos, UAlberta

PCA Visualization of Digits

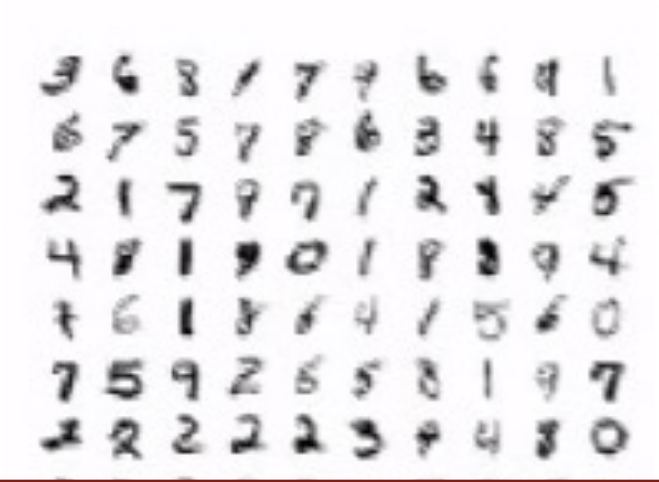
Fig: Laurens van der Maaten

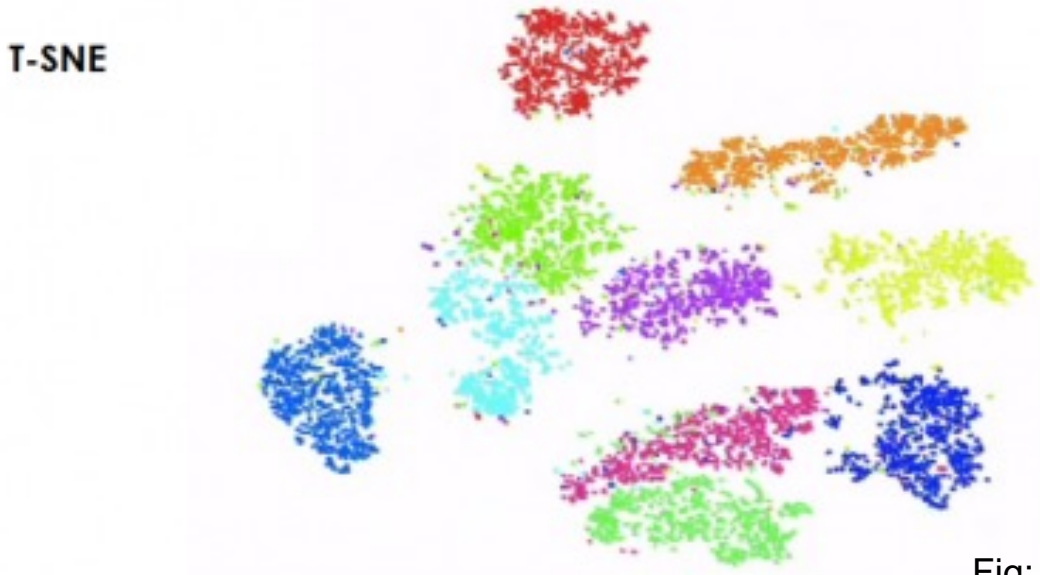
Utility of PCA

- PCA is often used as a preprocessing step for supervised learning
  - reduces dimensionality
  - eliminates multicollinearity

- Can also be used to aid in visualization

Beyond PCA: Non-linear dimensionality reduction





T-SNE and ISOMAP are more powerful methods that use nonlinear mappings, but:

- Require careful hyperparameter tuning
- Harder to optimize
- Not as easy to interpret, no easy projection back to original data

Fig: Laurens van der Maaten

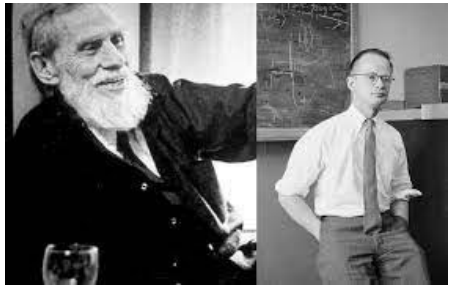
UNIVERSITY OF KWAZULU-NATAL ™
INYUVESI YAKWAZULU-NATALI

NITheCS
National Institute for
Theoretical and Computational Sciences

Neural Nets

Brief History of Neural Networks

- **1943:** Perceptron model (McCulloch & Pitts)
  - Intended as theoretical model of biological neurons
  - Linear classifiers! (Specialized learning algorithm)



- **1958:** Implementation as Mark I Perceptron (Rosenblatt)
  - Demonstrated capabilities of handwritten letter recognition

- **1969:** Perceptrons cannot learn XOR (Minsky & Papert)
  - Highly controversial (may have helped cause “AI winter”)

Brief History of Neural Networks

- **1985:** Representation learning (Rumelhart, Hinton, & Williams)
  - Interpret intermediate computations of neural networks
- **1989:** Convolutional neural networks (Lecun)
  - Convert handcrafted convolutional filters into learnable parameters
- **1995:** Long short-term memory (Hochreiter & Schmidhuber)
  - Refinement of neural networks designed to predict sequences
  - Complex design demonstrates flexibility of neural networks

Brief History of Neural Networks

- **1998:** Convolutional neural networks for MNIST (Lecun)
  - Human-level performance on handwritten digit recognition
- **2012:** ImageNet breakthrough (Krizhevsky, Sutskever, & Hinton)
  - Reduced error on image classification by 50%
- **2017:** Transformer architecture (Vaswani et al.)
- **2018:** Turing award (Bengio, Hinton, & Lecun)
- **To be continued?**

What Changed?

- **More compute:** GPUs
- **More data:** ImageNet, Wikipedia/Web, etc.
  - New applications
- **Better optimization algorithms:** Mini-batch SGD, acceleration, etc.
- **Accumulate "folk knowledge":** Parameter initialization, etc.
  - Encoded in open source software packages
- **Modern perspective:** "Differentiable programming"
- **Lots of investment from tech companies**

Agenda

- **Model family**
  - Custom model family rather than a single model family
- **Optimization**
  - Backpropagation algorithm for computing gradient

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

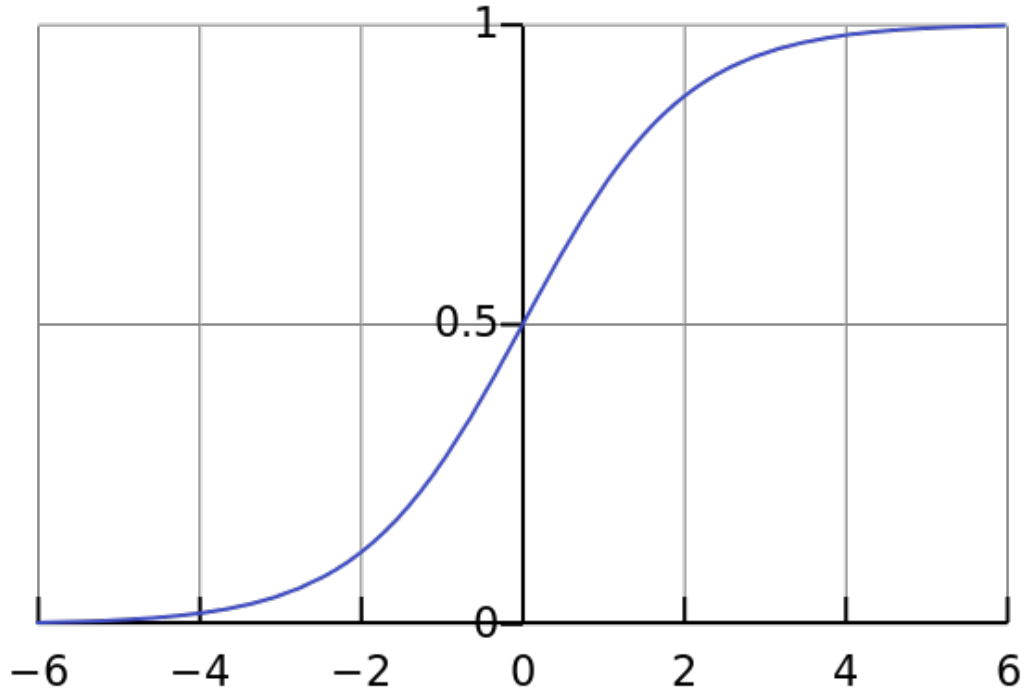
$$f_{W,\beta}(x) = \beta^\top g(Wx)$$

- **Parameters:** Matrix $W \in \mathbb{R}^{d \times k}$ and vector $\beta \in \mathbb{R}^k$
  - k is a hyperparameter called the **number of hidden neurons**

- Here, $g\colon \mathbb{R} \to \mathbb{R}$ is a given **activation function**
  - It is applied componentwise in $f_{W,\beta}$ (i.e., $g\left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\right) = \begin{bmatrix} g(z_1) \\ g(z_2) \end{bmatrix}$)
  - **Example:** $g(z) = \sigma(z)$ (where σ is the sigmoid function)

A Simple Neural Network

- Possible choice of activation function: $g(z) = \sigma(z)$



A Simple Neural Network

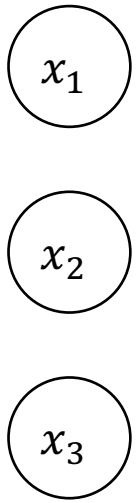
- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) =$$

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

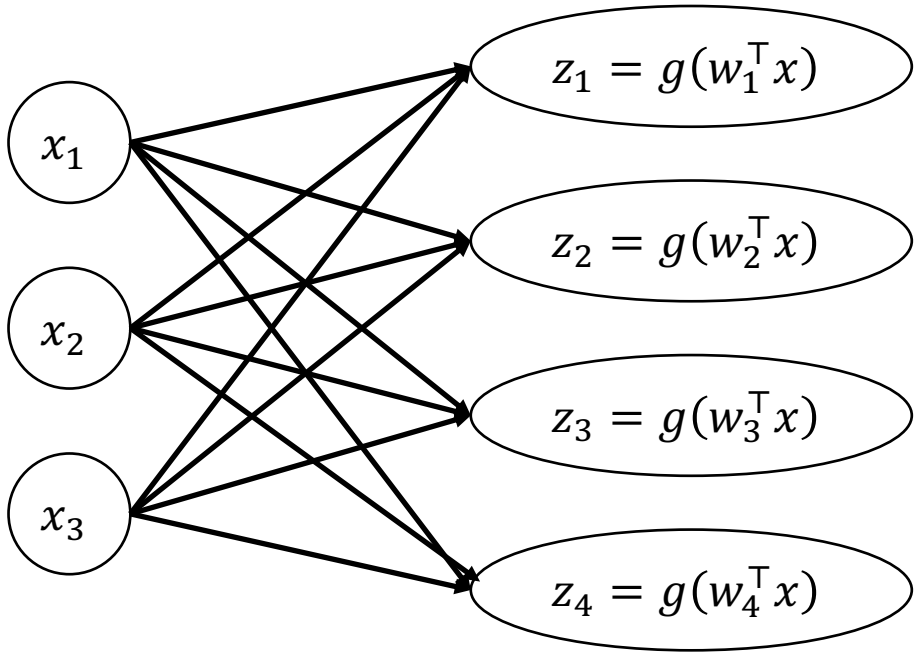
$$f_{W,\beta}(x) = \qquad x$$



A Simple Neural Network

- **Feedforward neural network model family (for regression):**

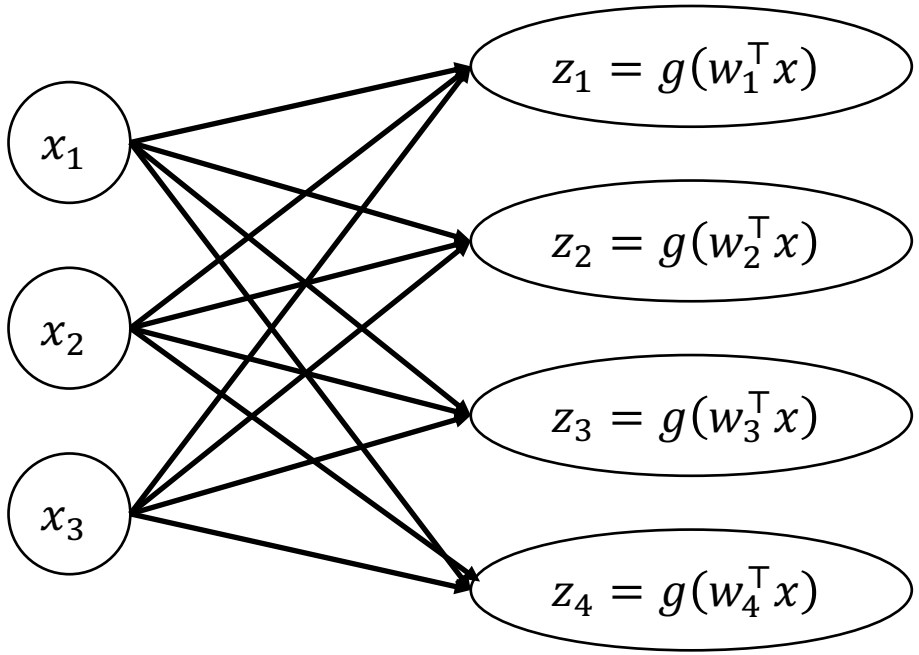
$$f_{W,\beta}(x) = \qquad Wx$$



A Simple Neural Network

- **Feedforward neural network model family (for regression):**

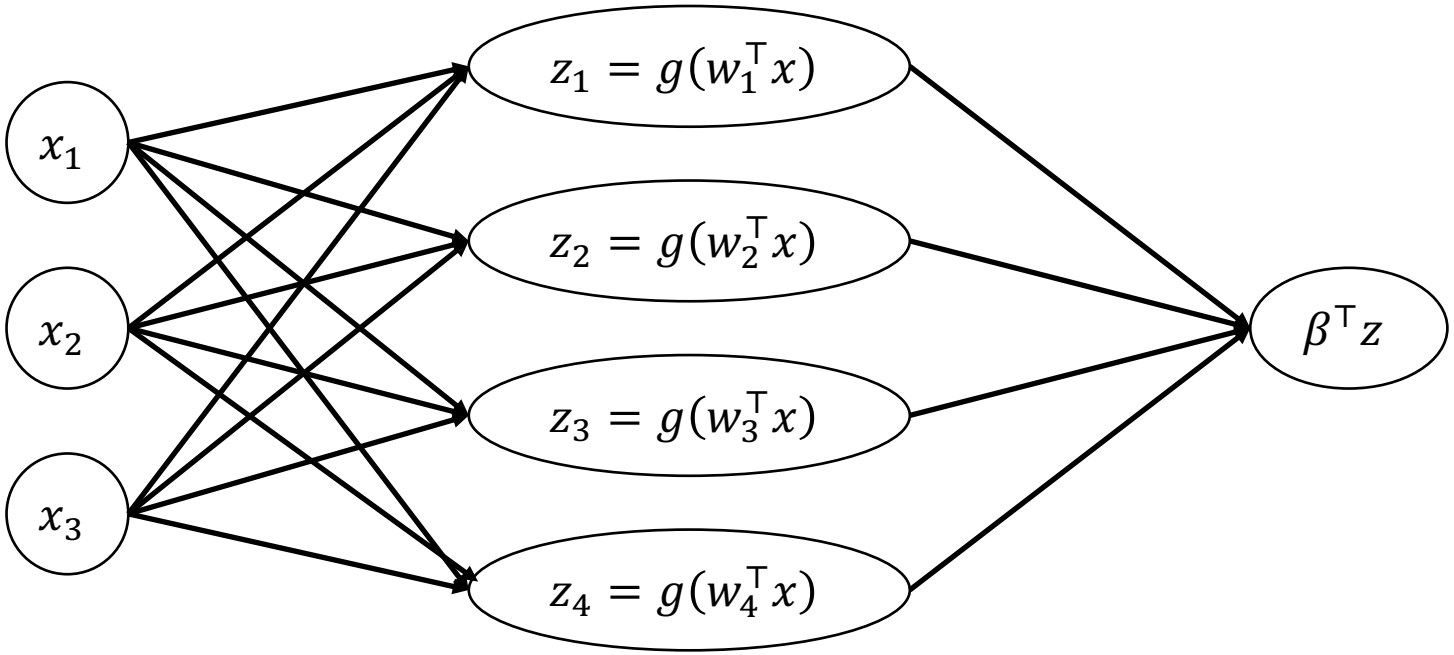
$$f_{W,\beta}(x) = \quad g(Wx)$$



A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^\top g(Wx)$$



A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^{\top} g(Wx)$$

- What happens if g is linear?

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^{\top} g(Wx)$$

- What happens if g is linear? Recovers linear functions!
  - Special case of identity:

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^\top g(Wx)$$

- What happens if g is linear? Recovers linear functions!
  - Special case of identity:

$$f_{W,\beta}(x) = \beta^\top g(Wx)$$

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^{\top} g(Wx)$$

- What happens if g is linear? Recovers linear functions!
  - Special case of identity:

$$f_{W,\beta}(x) = \beta^{\top} g(Wx) = \beta^{\top} W x$$

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^{\top} g(Wx)$$

- What happens if g is linear? Recovers linear functions!
  - Special case of identity:

$$f_{W,\beta}(x) = \beta^{\top} g(Wx) = \beta^{\top} Wx = \tilde{\beta}^{\top} x$$

A Simple Neural Network

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = \beta^\top g(Wx)$$

- What happens if g is linear? Recovers linear functions!
  - Special case of identity:

$$f_{W,\beta}(x) = \beta^\top g(Wx) = \beta^\top Wx = \tilde{\beta}^\top x$$

- Using a nonlinearity is important!
  - **In general:** Linear regression over "features" $g(Wx)$

What About Classification?

- **Recall:** For logistic regression, we choose the likelihood to be

$$p_\beta(Y = 1 \mid x) = \frac{1}{1 + e^{-\beta^\top x}}$$

What About Classification?

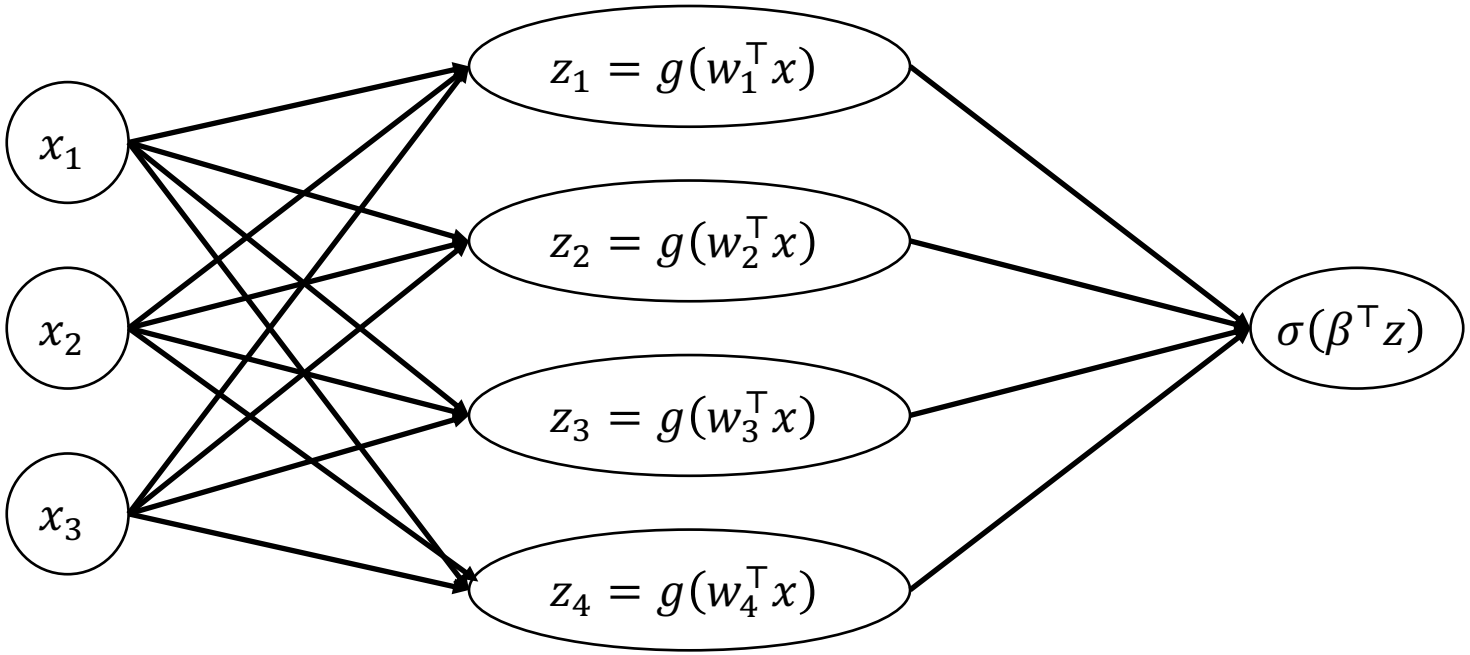
- **Recall:** For logistic regression, we choose the likelihood to be

$$p_\beta(Y = 1 \mid x) = \sigma(\beta^\top x)$$

What About Classification?

- **For binary classification:**

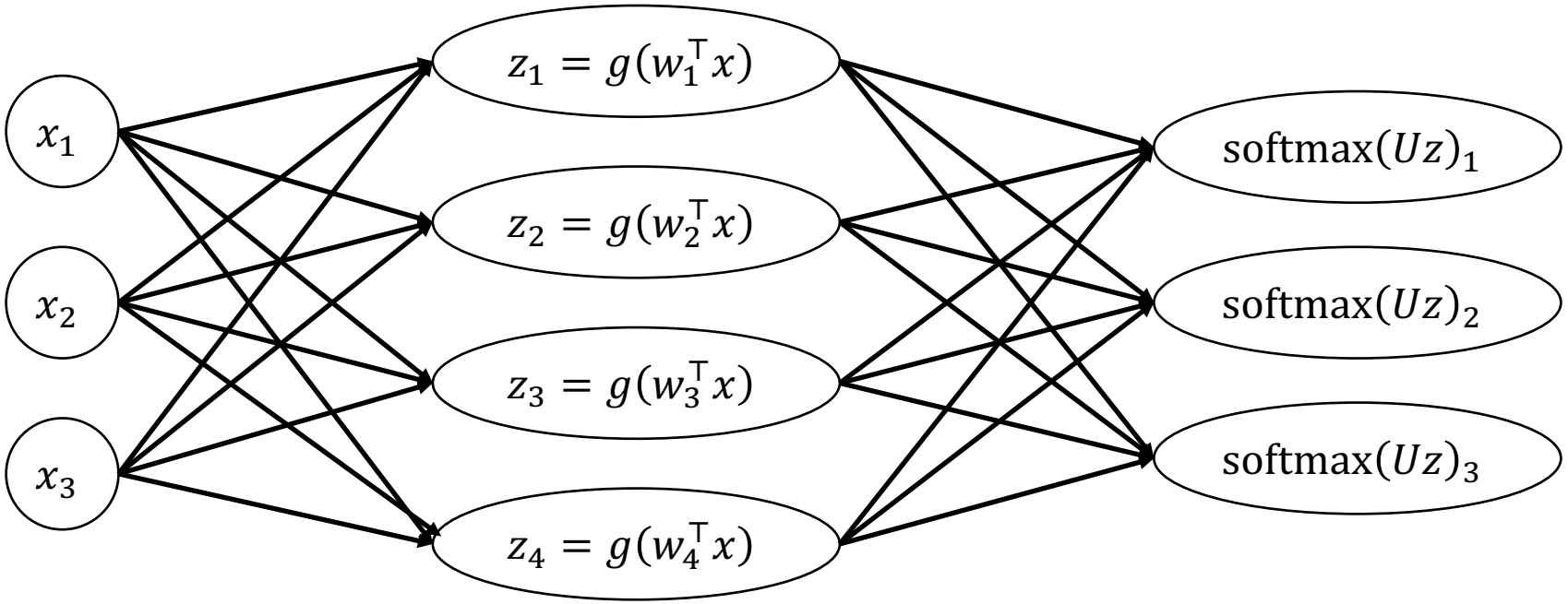
$$p_{W,\beta}(Y = 1 \mid x) = \sigma\left(\beta^\top g(Wx)\right)$$



What About Classification?

- **For multi-class classification:**

$$p_{W,U}(Y = y \mid x) = \text{softmax}\big(Ug(Wx)\big)_y$$



Historical vs. Modern View

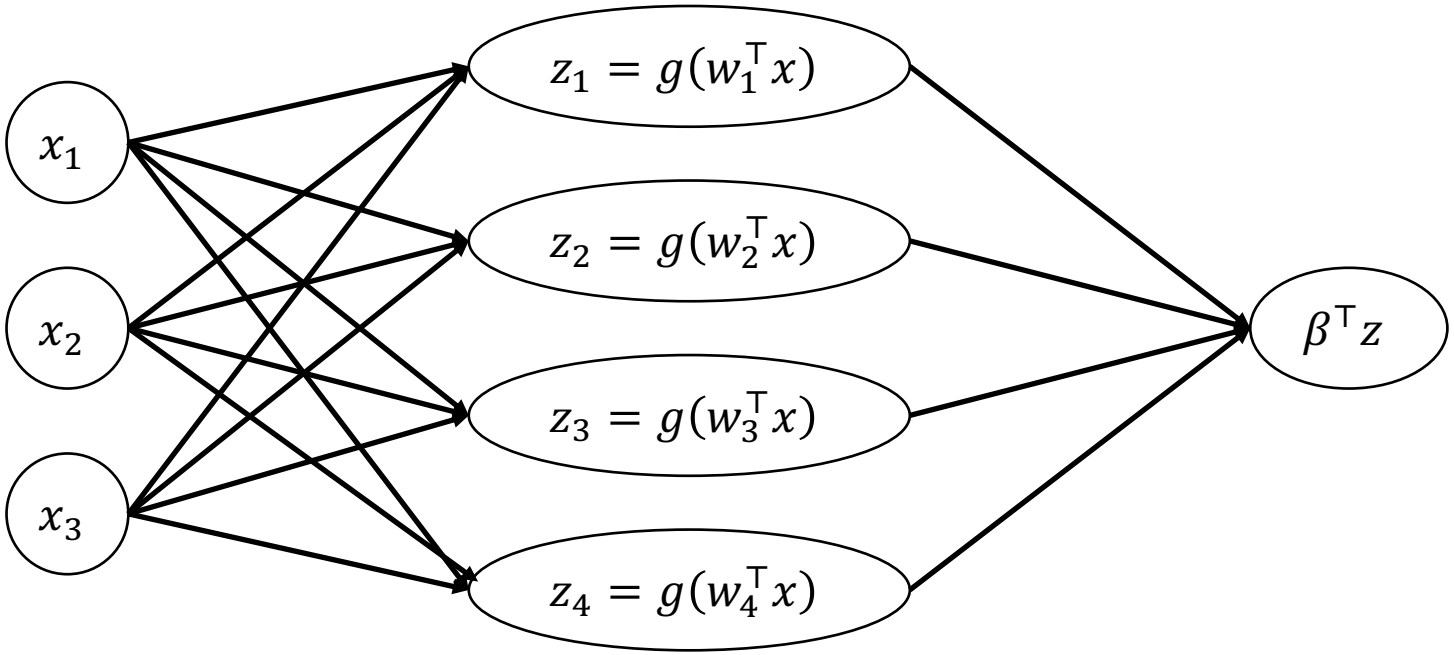
- **Historical view:** Specific model families
  - Feedforward neural networks, convolutional neural networks, etc.
  - Each new model family (“architecture”) requires a custom implementation

- **Modern view:** Design model families by composing building blocks
  - Building blocks are “layers”
  - Layers can be **programmatically** composed together (by stacking, concatenating, etc.) to form different model families

Historical View

- **Feedforward neural network model family (for regression):**

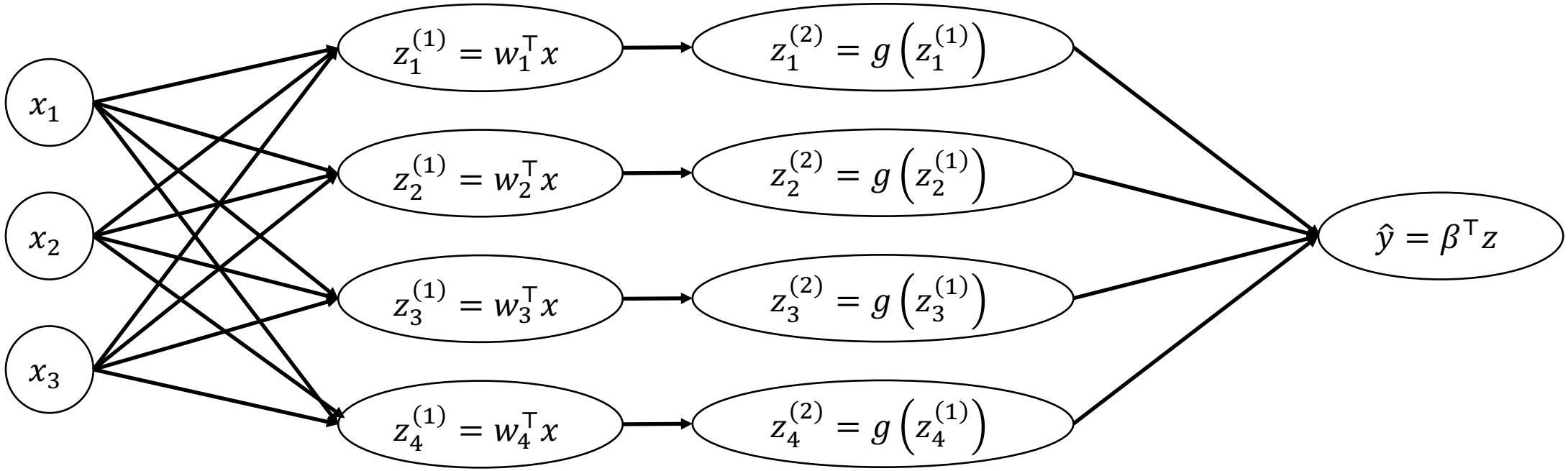
$$f_{W,\beta}(x) = \beta^\top g(Wx)$$



Modern View

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = f_\beta\left(g\left(f_W(x)\right)\right) = f_\beta \circ g \circ f_W(x)$$



Modern View

- Each **layer** is a parametric function $f_{W_j}: \mathbb{R}^k \to \mathbb{R}^h$ for some k, h

- Compose sequentially to form model family:

$$f_W(x) = f_{W_m}\left(\dots\left(f_{W_1}(x)\right)\dots\right)$$

- We will use the following notation:

$$f_W = f_{W_m} \circ \cdots \circ f_{W_1}$$

Modern View

- Each **layer** is a parametric function $f_{W_j}: \mathbb{R}^k \to \mathbb{R}^h$ for some k, h
- Can compose layers in other ways, e.g., concatenation:

$$f_W(x) = f_{W_1}(x) \oplus f_{W_2}(x)$$

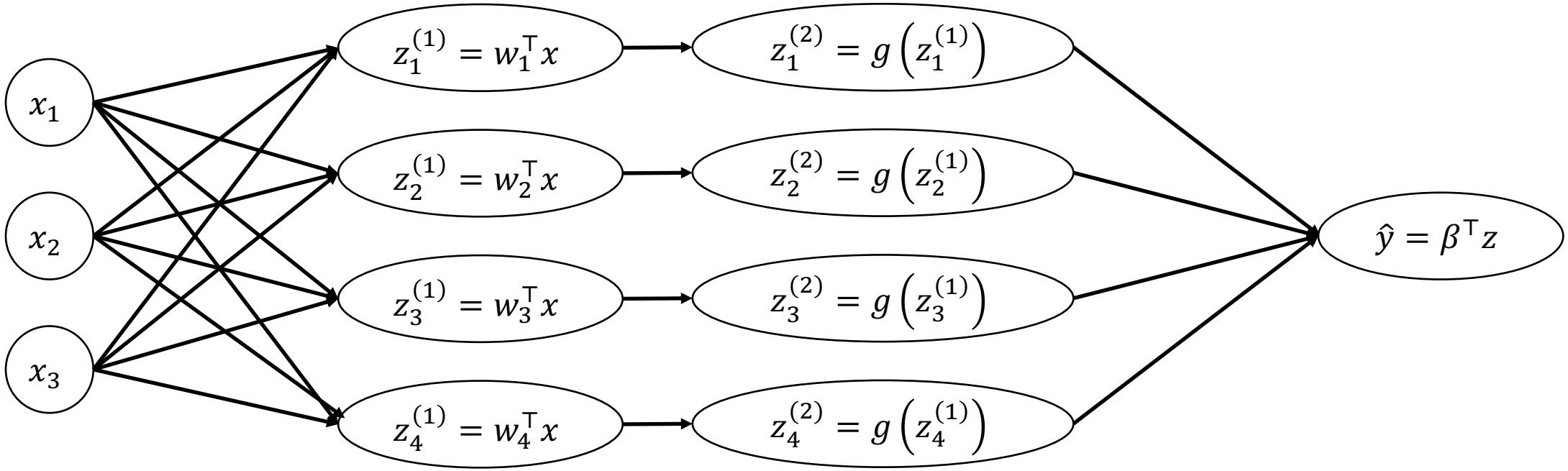
- Here, we have defined

$$\begin{bmatrix} z_1 & \cdots & z_d \end{bmatrix}^\top \oplus \begin{bmatrix} z'_1 & \cdots & z'_{d'} \end{bmatrix}^\top = \begin{bmatrix} z_1 & \cdots & z_d & z'_1 & \cdots & z'_{d'} \end{bmatrix}^\top$$

Modern View

- **Feedforward neural network model family (for regression):**

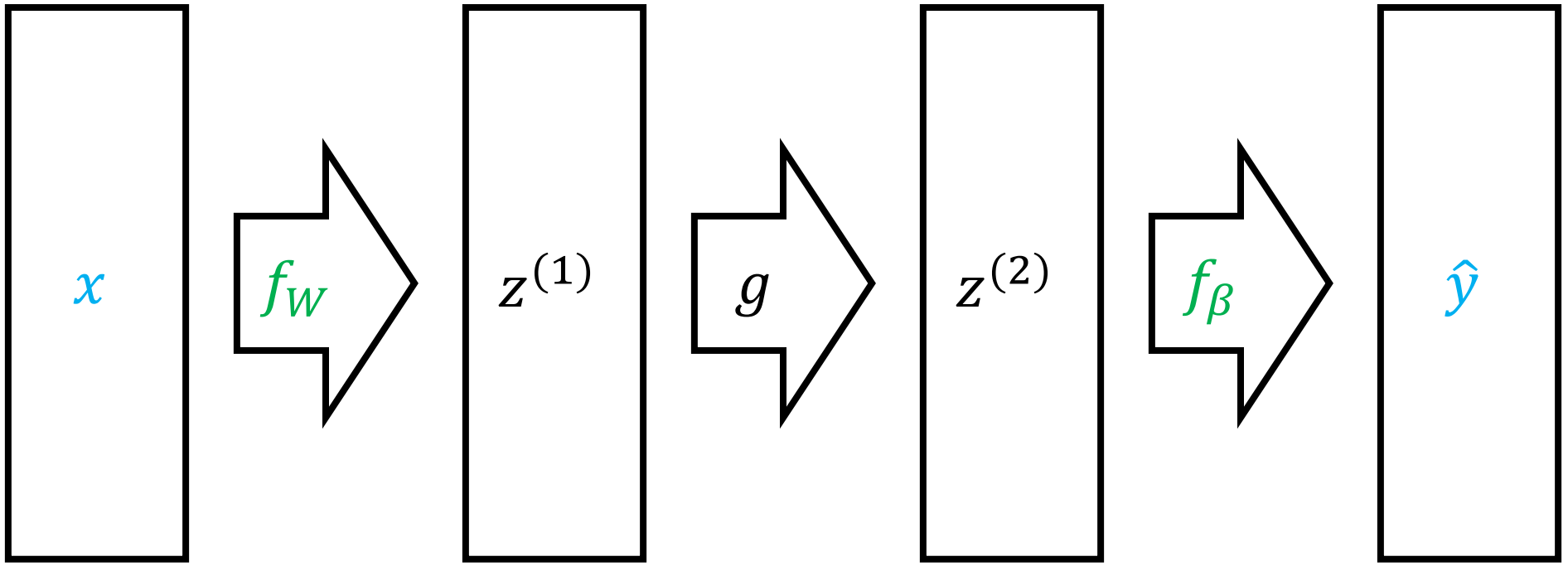
$$f_{W,\beta}(x) = f_\beta \circ g \circ f_W(x)$$



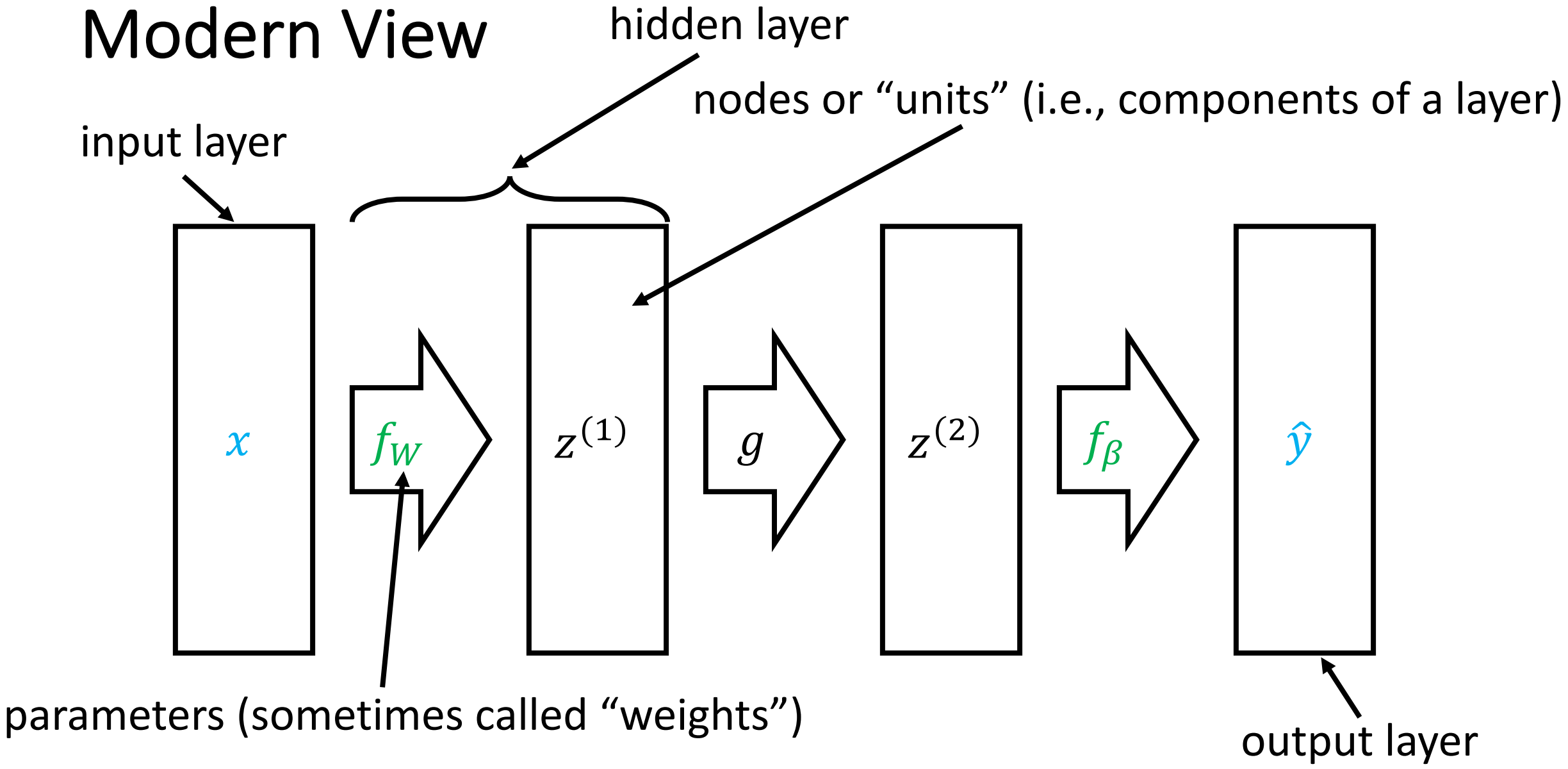
Modern View

- **Feedforward neural network model family (for regression):**

$$f_{W,\beta}(x) = f_\beta \circ g \circ f_W(x)$$



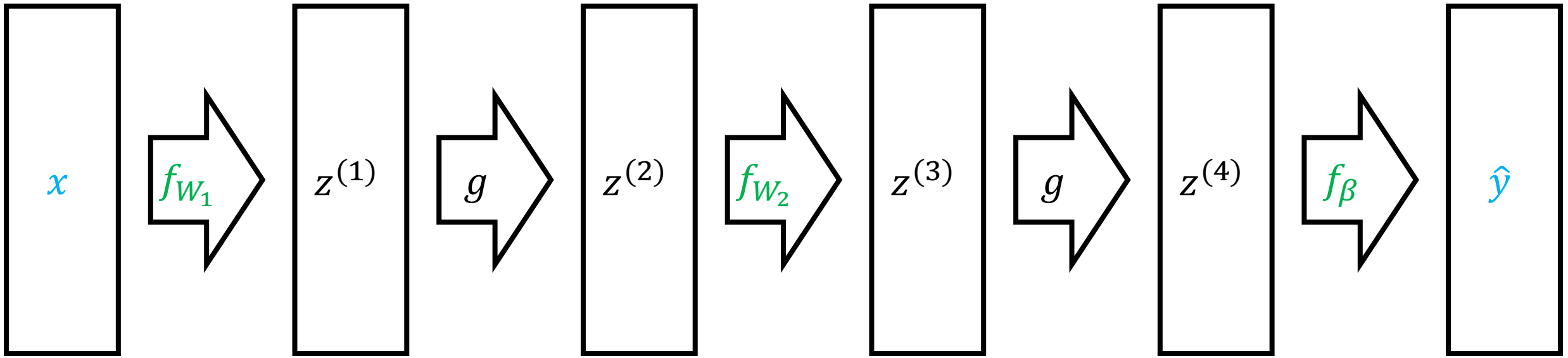
Modern View


hidden layer
nodes or “units” (i.e., components of a layer)
input layer
x
f_W
$z^{(1)}$
g
$z^{(2)}$
f_β
$\hat{y}$
parameters (sometimes called “weights”)
output layer

Modern View

- **Neural network with two hidden linear layers:**

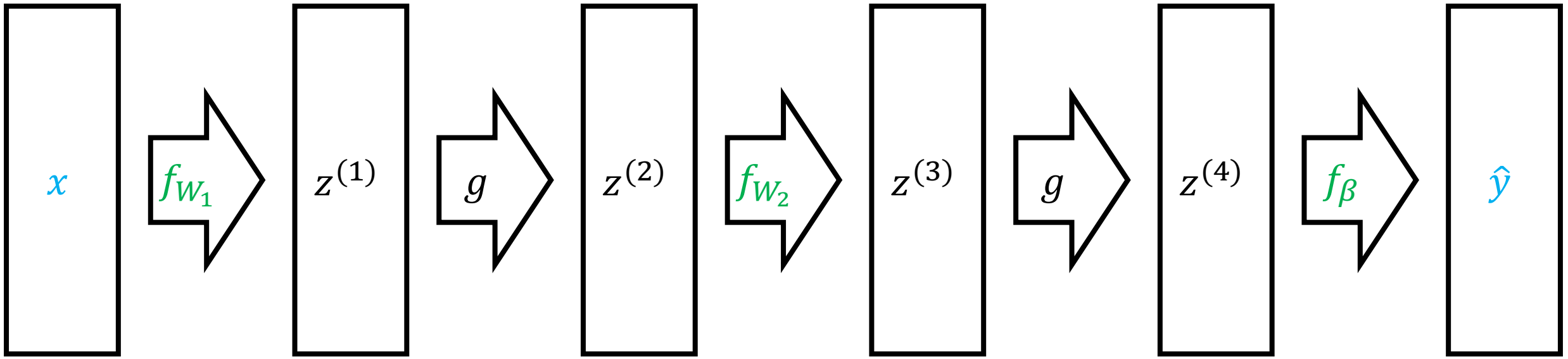
$$f_{W_1,W_2,\beta}(x) = f_\beta \circ g \circ f_{W_2} \circ g \circ f_{W_1}(x)$$



Modern View

- **Neural network with two hidden linear layers:**

$$f_{W_1,W_2,\beta}(x) = f_\beta\left(g\left(f_{W_2}\left(g\left(f_{W_1}(x)\right)\right)\right)\right)$$



Learn successively more “high-level” representations

Neural Networks

- **Pros**
  - **“Meta” strategy:** Enables users to **design** model family
  - Design model families that capture **symmetries/structure** in the data (e.g., read a sentence forwards, translation invariance for images, etc.)
  - “Representation learning” (automatically learn features for certain domains)
  - More parameters!

- **Cons**
  - Very hard to train! (Non-convex loss functions)
  - Lots of parameters → need lots of data!
  - Lots of design decisions

Optimization Algorithm

- Based on gradient descent, with a few tweaks
  - **Note:** Loss is nonconvex, but gradient descent works well in practice

- **Key challenge:** How to compute the gradient?
  - **Strategy so far:** Work out gradient for every model family
  - **New strategy:** Algorithm for computing gradient of an arbitrary programmatic composition of layers
  - This algorithm is called **backpropagation**

Gradient Descent

- $W_1 \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2, \ldots\}$ **until** convergence:

$$W_{t+1,j} \leftarrow W_{t,j} - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_{W_j} L\left(f_{W_t}(x_i), y_i\right) \quad \text{(for each } j\text{)}$$

- **return** f_{W_t}

Backpropagation

- **Input**
  - Example-label pair (x, y)
  - Arbitrary model $f_{W_m} \circ \cdots \circ f_{W_1}$
  - Loss $L(\hat{y}, y)$ for predicted label $\hat{y}$ and true label y
  - Derivative $\nabla_{\hat{y}} L(\hat{y}, y)$ (as a function)
  - Derivatives $D_{W_j} f_{W_j}(z)$ and $D_z f_{W_j}(z)$ (e.g., as a function)

- **Output:** $\nabla_{W_j} L(f_W(x), y)$

Recall: Multi-Dimensional Derivatives

- **Given:**
  - Function $f_W(z)$ mapping parameters $W \in \mathbb{R}^d$ and input vector $z \in \mathbb{R}^k$ to a vector $f_W(z) \in \mathbb{R}^h$
  - Current parameters W and z

- The **derivative** of f_W at W and z with respect to z is a matrix

$$D_z f_W(z) \in \mathbb{R}^{h \times k}$$

Recall: Multi-Dimensional Derivatives

- **Given:**
  - Function $f_W(z)$ mapping parameters $W \in \mathbb{R}^d$ and input vector $z \in \mathbb{R}^k$ to a vector $f_W(z) \in \mathbb{R}^h$
  - Current parameters W and z

- The **derivative** of f_W at W and z with respect to W is a matrix

$$D_W f_W(z) \in \mathbb{R}^{h \times d}$$

Recall: Multi-Dimensional Derivatives

- **Given:**
  - Function $f_W(z)$ mapping parameters $W \in \mathbb{R}^d$ and input vector $z \in \mathbb{R}^k$ to a vector $f_W(z) \in \mathbb{R}^h$
  - Current parameters W and z

- **Intuition:** The linear function that best approximates f_W at W and z:

$$f_{W+dW}(z + dz) \approx f_W(z) + D_z f_W(z) dz + D_W f_W(z) dW$$

Backpropagation Example

- **Gradient of MSE loss (for regression):**

$$\nabla_W L(W, \beta; Z) = \nabla_W \frac{1}{n} \sum_{i=1}^{n} \left(f_{W,\beta}(x_i) - y_i \right)^2$$
$$= \frac{2}{n} \sum_{i=1}^{n} \left(f_{W,\beta}(x_i) - y_i \right) D_W f_{W,\beta}(x_i)$$

$$\nabla_\beta L(W, \beta; Z) = \nabla_\beta \frac{1}{n} \sum_{i=1}^{n} \left(f_{W,\beta}(x_i) - y_i \right)^2$$
$$= \frac{2}{n} \sum_{i=1}^{n} \left(f_{W,\beta}(x_i) - y_i \right) D_\beta f_{W,\beta}(x_i)$$

Backpropagation Example

- **Derivative of neural network:**

$$D_\beta f_{W,\beta}(x) = D_\beta\left(f_\beta \circ g \circ f_W\right)(x)$$
$$= D_\beta f_\beta\left(g \circ f_W(x)\right)$$

$$D_W f_{W,\beta}(x) = D_W\left(f_\beta \circ g \circ f_W\right)(x)$$
$$= D_z f_\beta\left(g \circ f_W(x)\right) D_W(g \circ f_W)(x)$$
$$= D_z f_\beta\left(g \circ f_W(x)\right) D_z g\left(f_W(x)\right) D_W f_W(x)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_m} f_W(x)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}\left(z^{(j-1)}\right) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_m} f_W(x) = D_{W_m} f_{W_m}\left(z^{(m-1)}\right)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_m} f_W(x) = D_{W_m} f_{W_m}(z^{(m-1)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-1}} f_W(x)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-1}} f_W(x) = D_z f_{W_m}\left(z^{(m-1)}\right) D_{W_{m-1}} f_{W_{m-1}}\left(z^{(m-2)}\right)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-1}} f_W(x) = D_z f_{W_m}(z^{(m-1)}) D_{W_{m-1}} f_{W_{m-1}}(z^{(m-2)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-1}} f_W(x) = D_z f_{W_m}(z^{(m-1)}) D_{W_{m-1}} f_{W_{m-1}}(z^{(m-2)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-2}} f_W(x)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-2}} f_W(x) = D_z f_{W_m}(z^{(m-1)}) D_z f_{W_{m-1}}(z^{(m-2)}) D_{W_{m-2}} f_{W_{m-2}}(z^{(m-3)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-2}} f_W(x) = D_z f_{W_m}(z^{(m-1)}) D_z f_{W_{m-1}}(z^{(m-2)}) D_{W_{m-2}} f_{W_{m-2}}(z^{(m-3)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-2}} f_W(x) = D_z f_{W_m}(z^{(m-1)}) D_z f_{W_{m-1}}(z^{(m-2)}) D_{W_{m-2}} f_{W_{m-2}}(z^{(m-3)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_{m-2}} f_W(x) = D_z f_{W_m}(z^{(m-1)}) D_z f_{W_{m-1}}(z^{(m-2)}) D_{W_{m-2}} f_{W_{m-2}}(z^{(m-3)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_j} f_W(x)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_j} f_W(x) = D_z f_{W_m}\left(z^{(m-1)}\right) \ldots D_z f_{W_{j+1}}\left(z^{(j)}\right) D_{W_j} f_{W_j}\left(z^{(j-1)}\right)$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_j} f_W(x) = D_z f_{W_m}(z^{(m-1)}) \ldots D_z f_{W_{j+1}}(z^{(j)}) D_{W_j} f_{W_j}(z^{(j-1)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_j} f_W(x) = D_z f_{W_m}(z^{(m-1)}) \dots D_z f_{W_{j+1}}(z^{(j)}) D_{W_j} f_{W_j}(z^{(j-1)})$$

Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x) = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$$

- We have

$$D_{W_j} f_W(x) = D_z f_{W_m}(z^{(m-1)}) \ldots D_z f_{W_{j+1}}(z^{(j)}) D_{W_j} f_{W_j}(z^{(j-1)})$$

Backpropagation

- We have

$$D_{W_j} f_W(x) = \underbrace{D_z f_{W_m}\left(z^{(m-1)}\right) \dots D_z f_{W_{j+1}}\left(z^{(j)}\right)}_{\text{Portions shared across terms}} D_{W_j} f_{W_j}\left(z^{(j-1)}\right)$$

Denote it by $D^{(j)}$

Backpropagation Algorithm

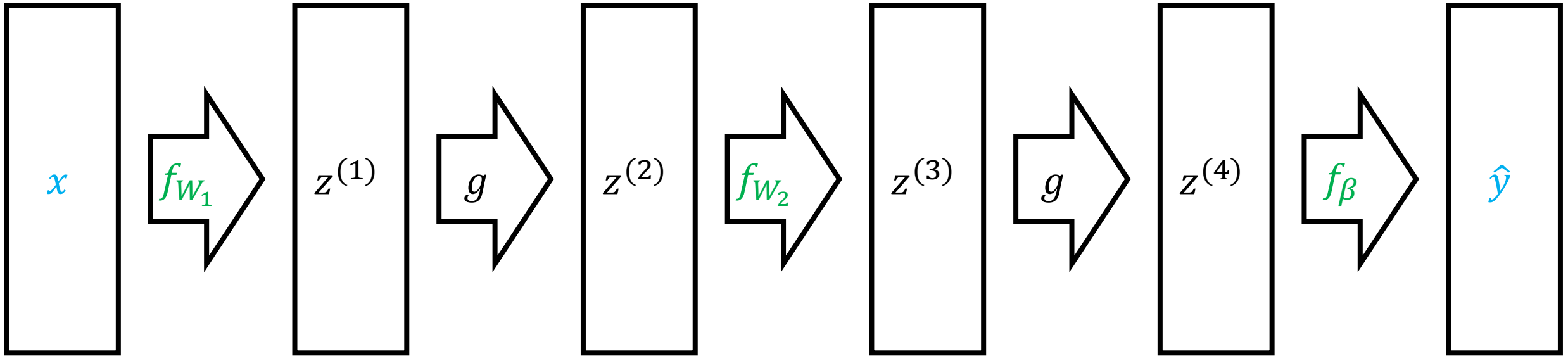
- Compute recursively starting from $j = m$ to $j = 1$:

$$D^{(j)} = D_z f_{W_m}\left(z^{(m-1)}\right) \dots D_z f_{W_{j+1}}\left(z^{(j)}\right)$$

$$= \begin{cases} 1 & \text{if } j = m \\ D^{(j+1)} D_z f_{W_{j+1}}\left(z^{(j)}\right) & \text{if } j < m \end{cases}$$

$$D_{W_j} f_W(x) = D^{(j)} D_{W_j} f_{W_j}\left(z^{(j-1)}\right)$$

Backpropagation

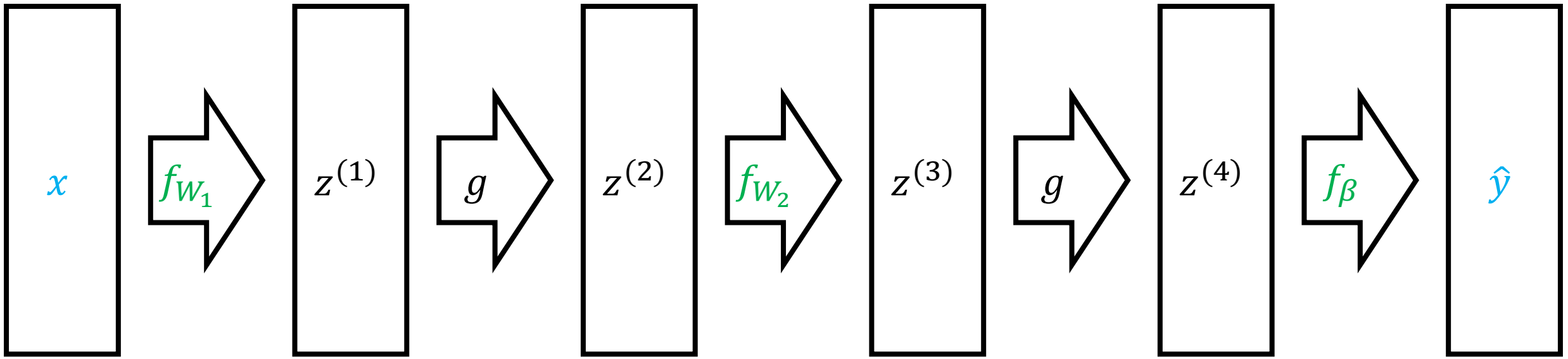


Forward pass: Compute $z^{(j)} = f_{W_j}\left(z^{(j-1)}\right)$

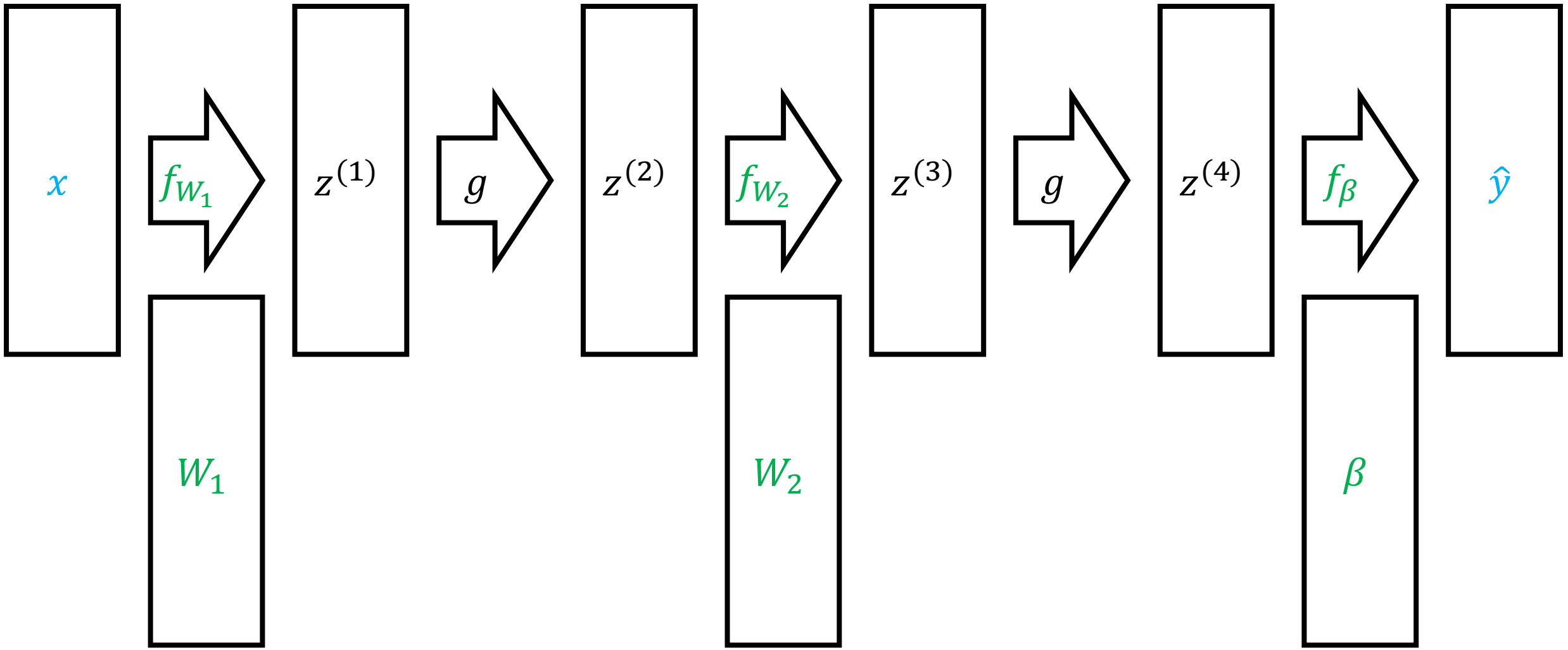
Backward pass: Compute $D^{(j)} = D^{(j+1)} D_z f_{W_{j+1}}\left(z^{(j)}\right)$ and $D_{W_j} f_W(x) = D^{(j)} D_{W_j} f_{W_j}\left(z^{(j-1)}\right)$

Final output: $\nabla_{\hat{y}} L\left(z^{(m)}, y\right)^\top D_{W_j} f_W(x)$

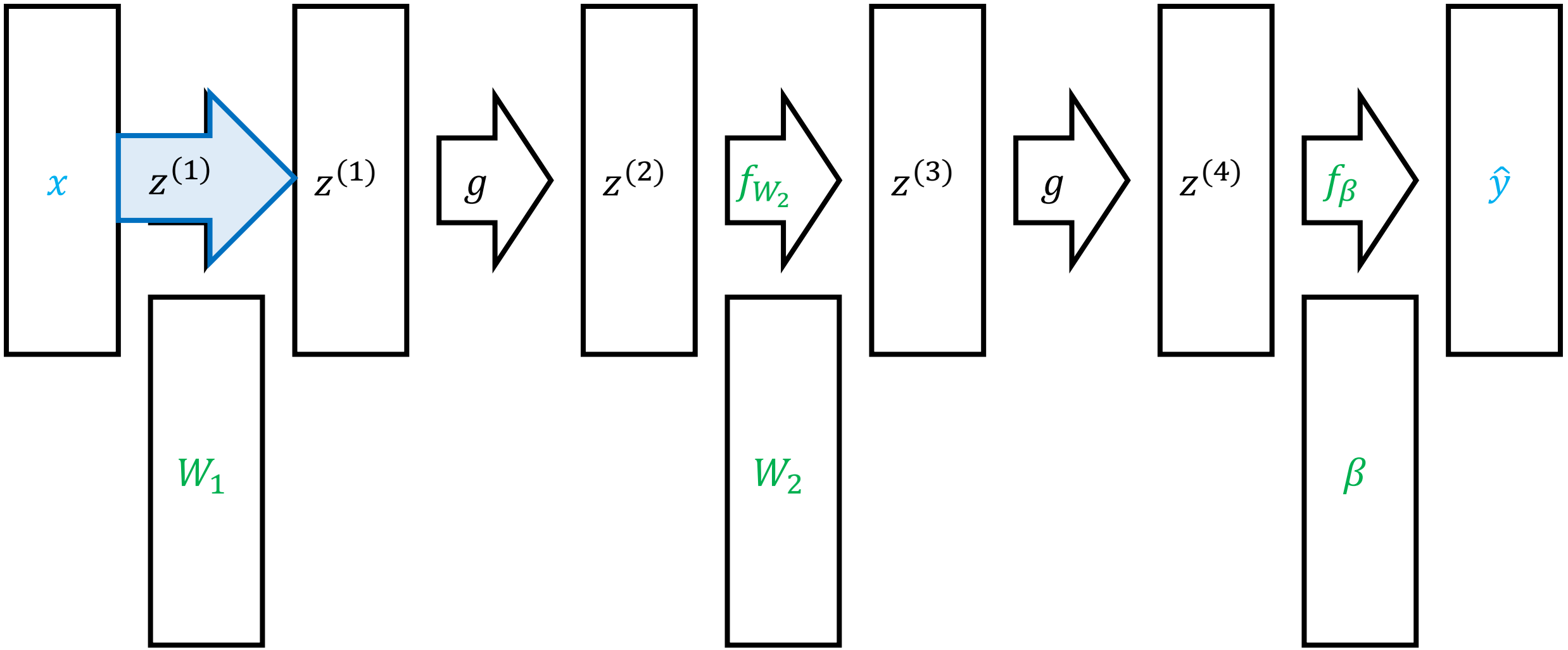
Backpropagation



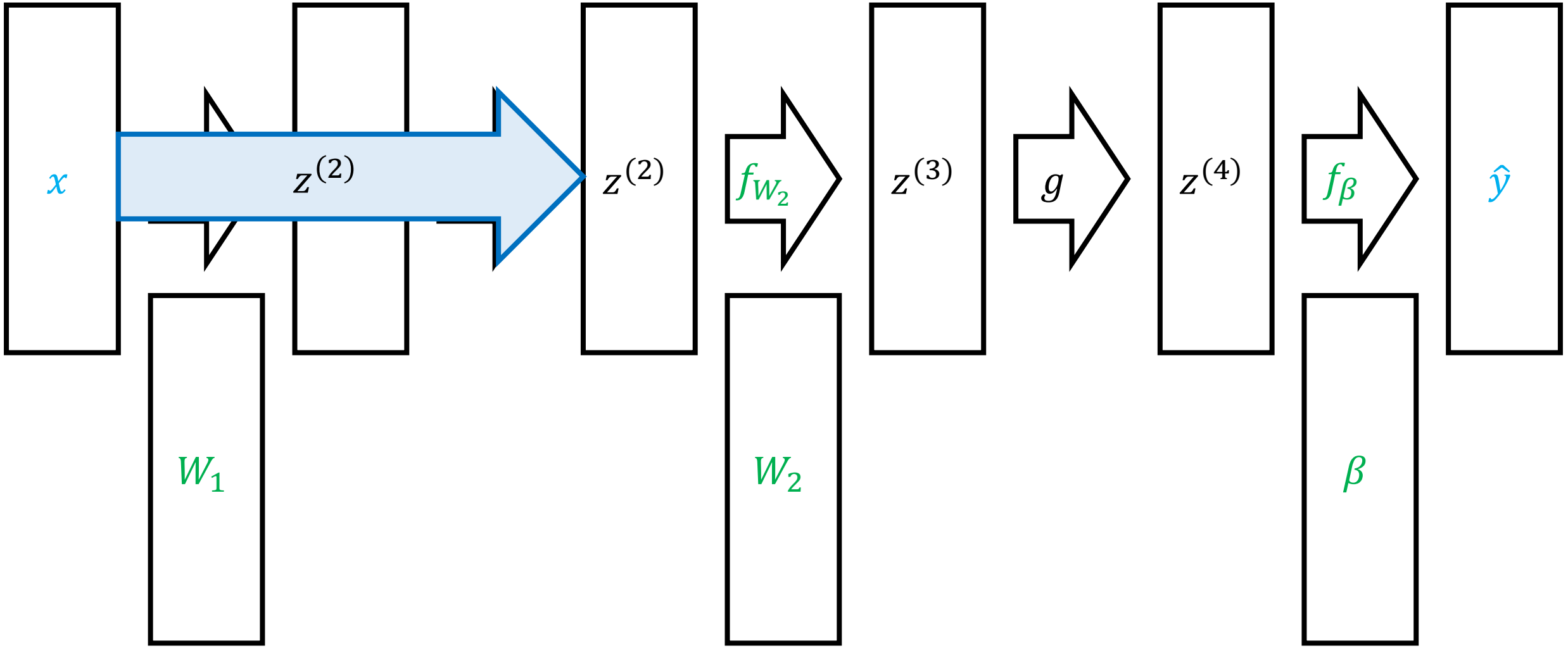
Backpropagation



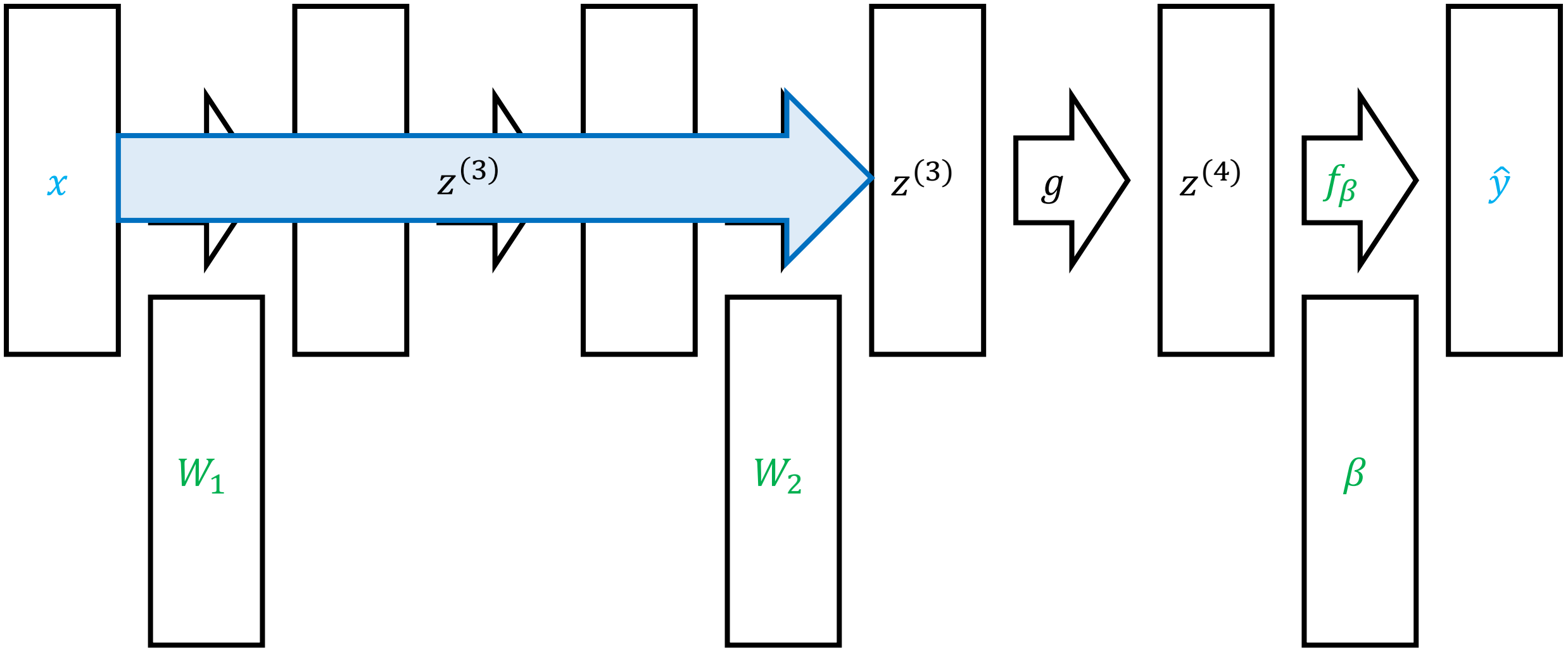
Backpropagation



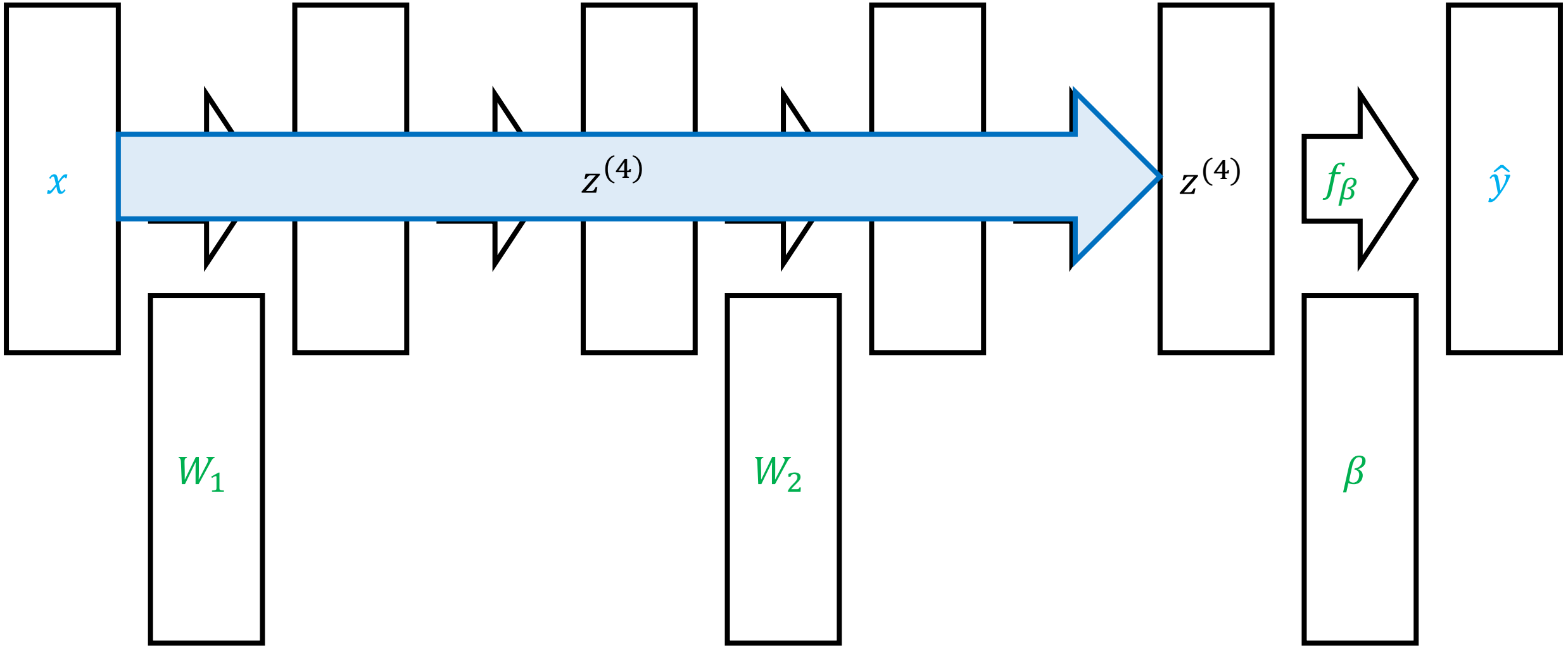
Backpropagation



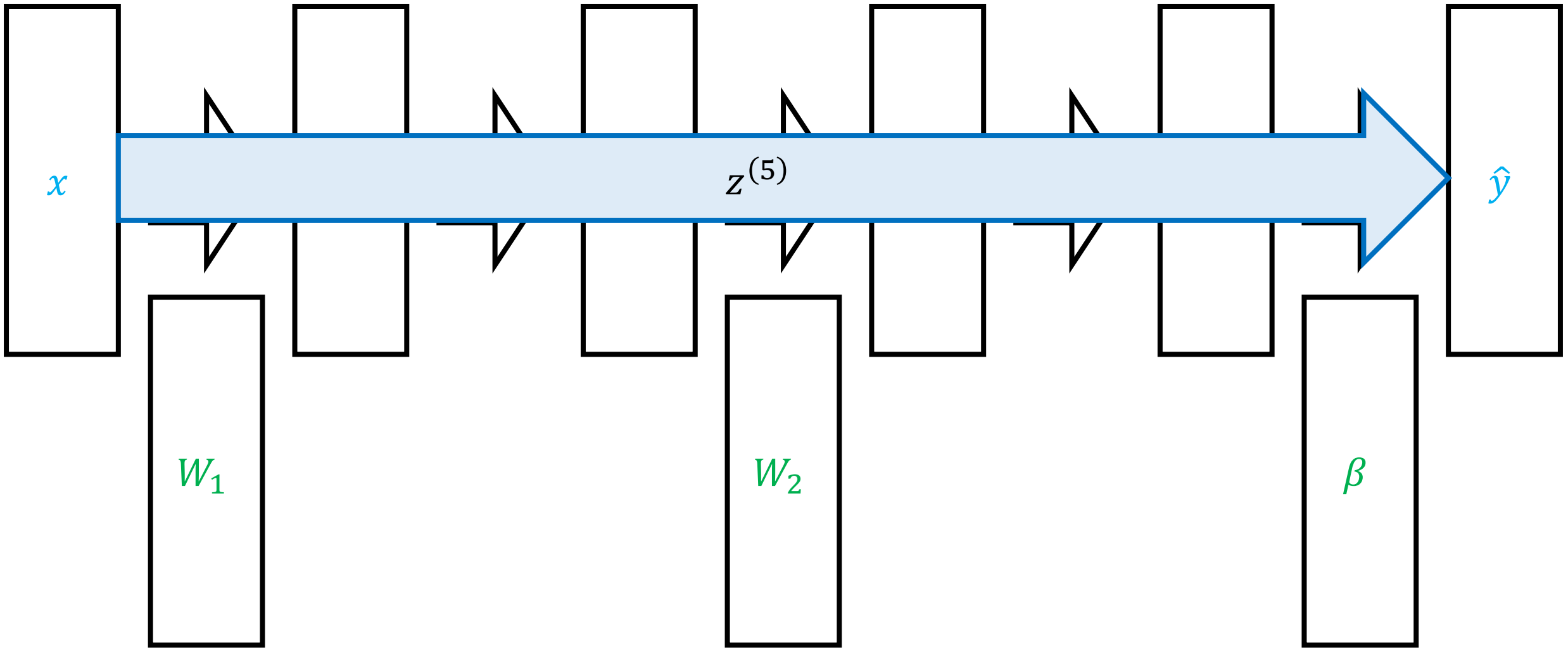
Backpropagation



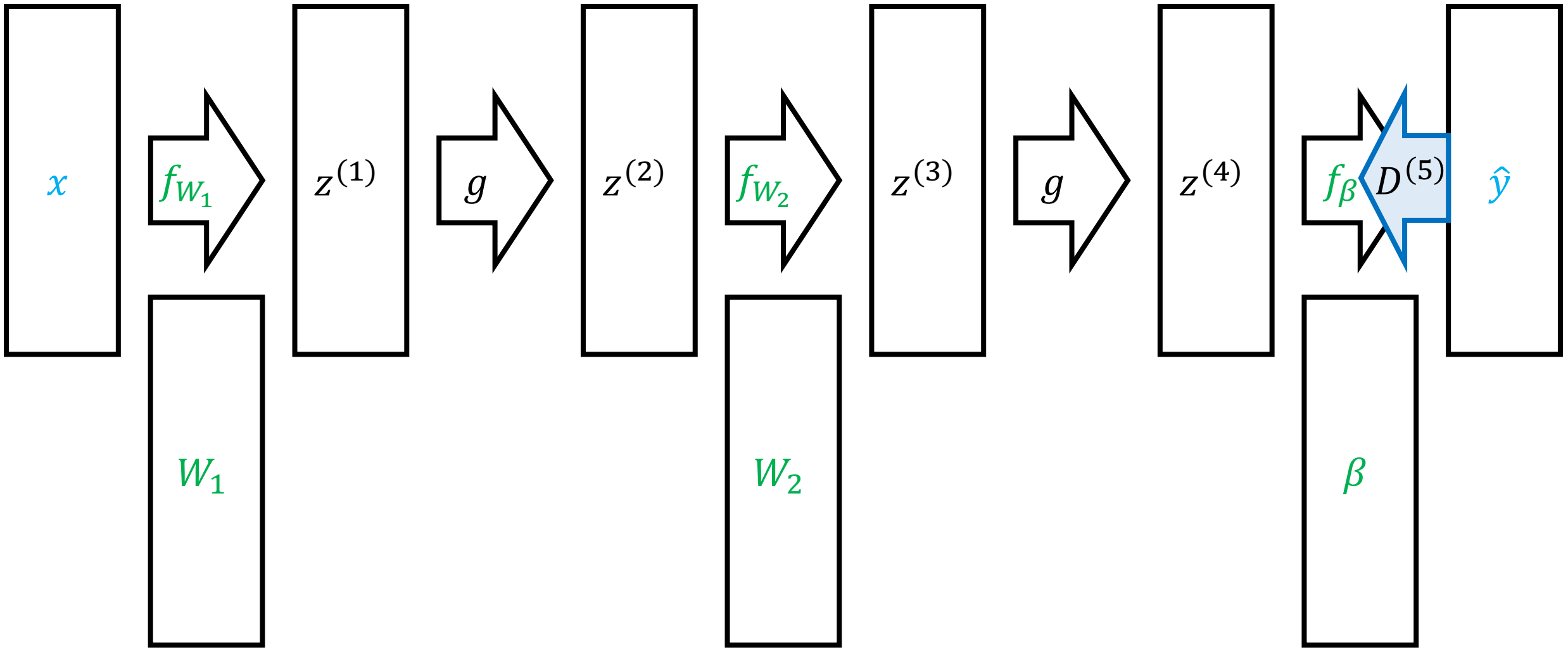
Backpropagation



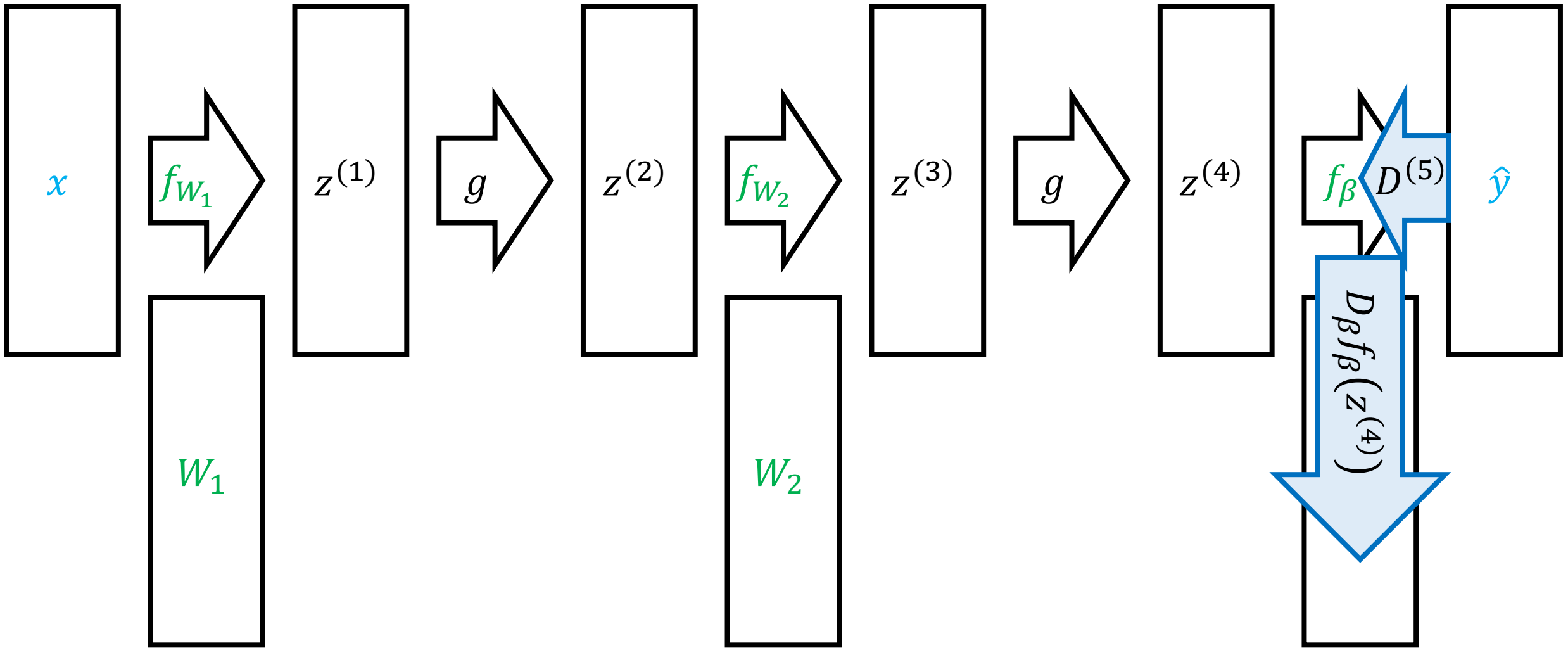
Backpropagation



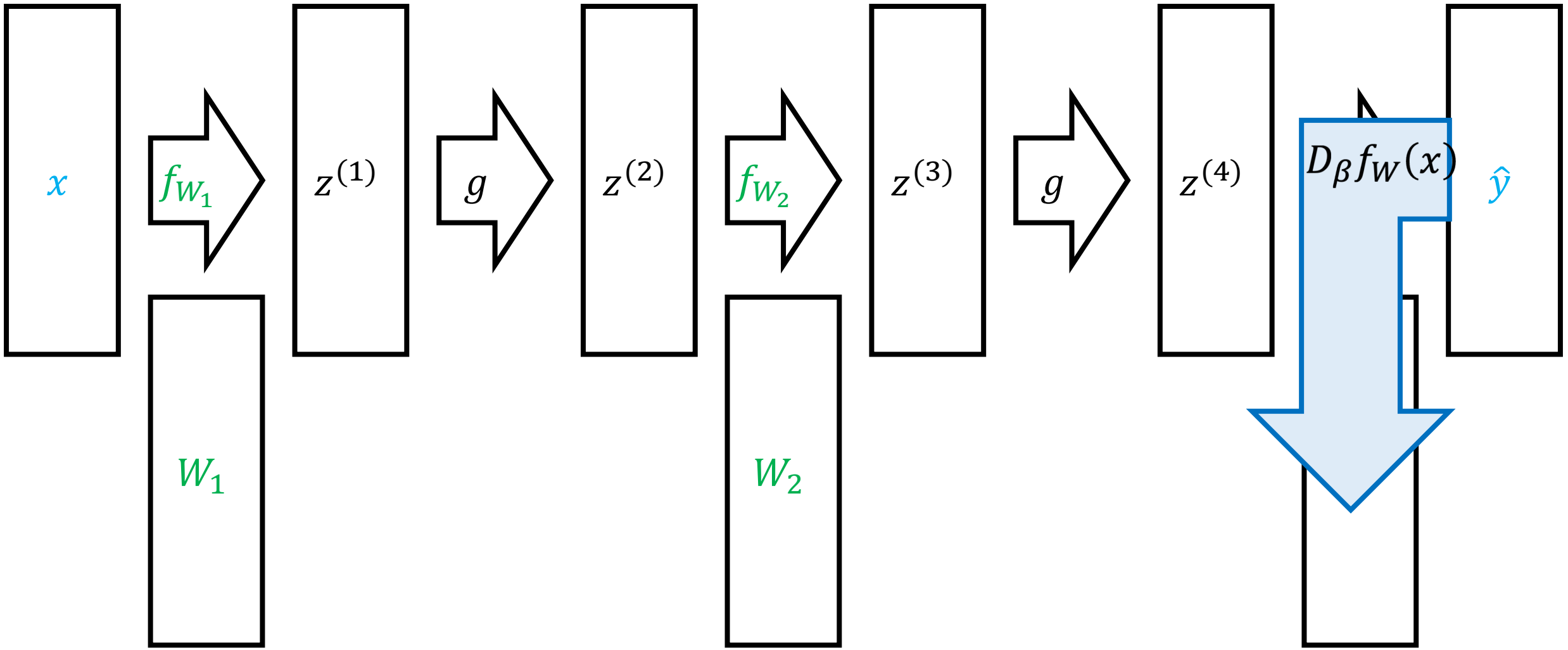
Backpropagation



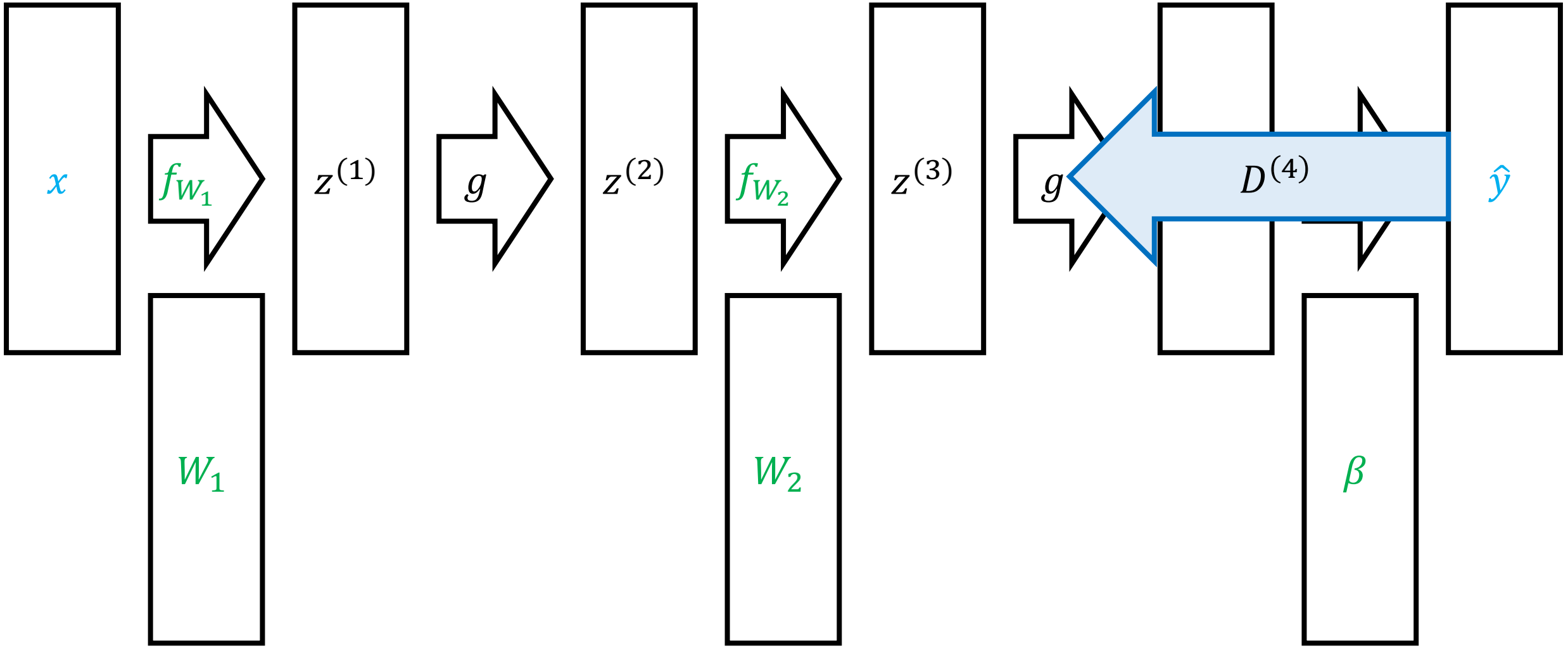
Backpropagation



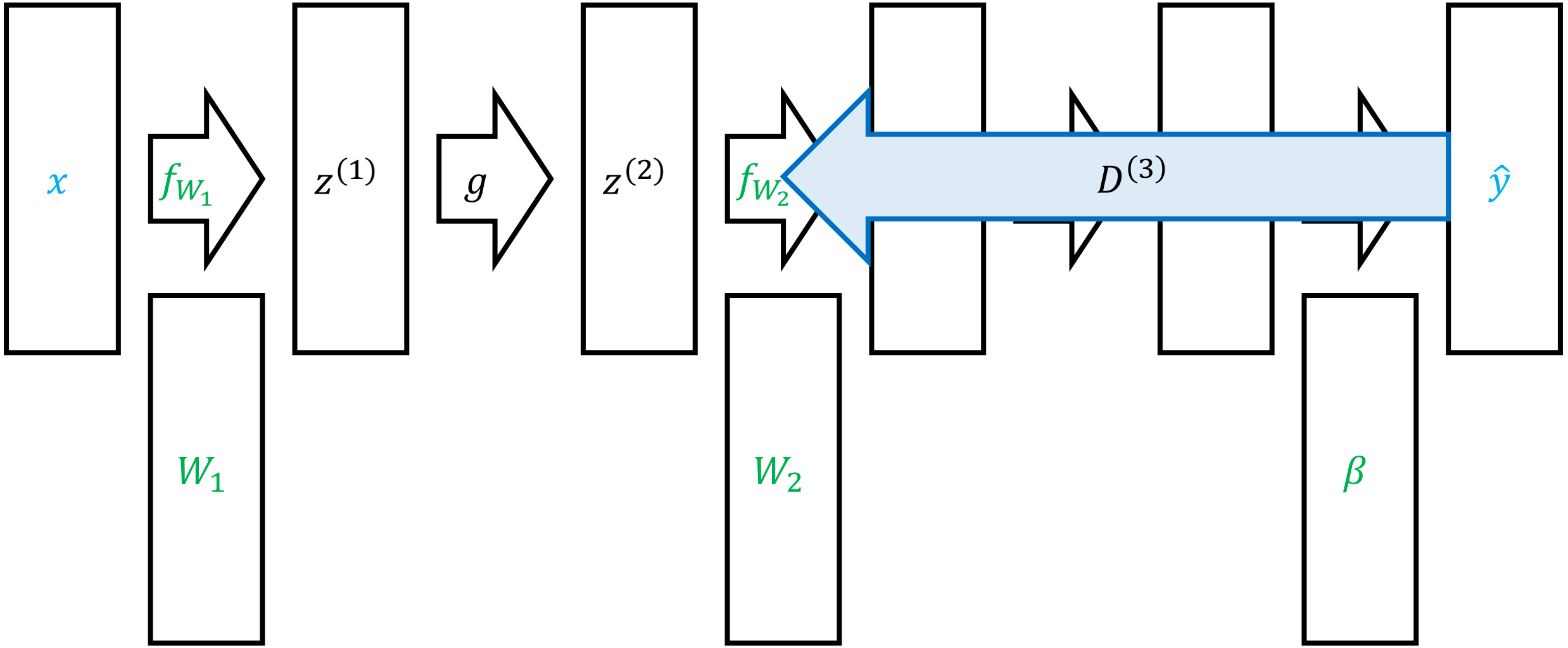
Backpropagation



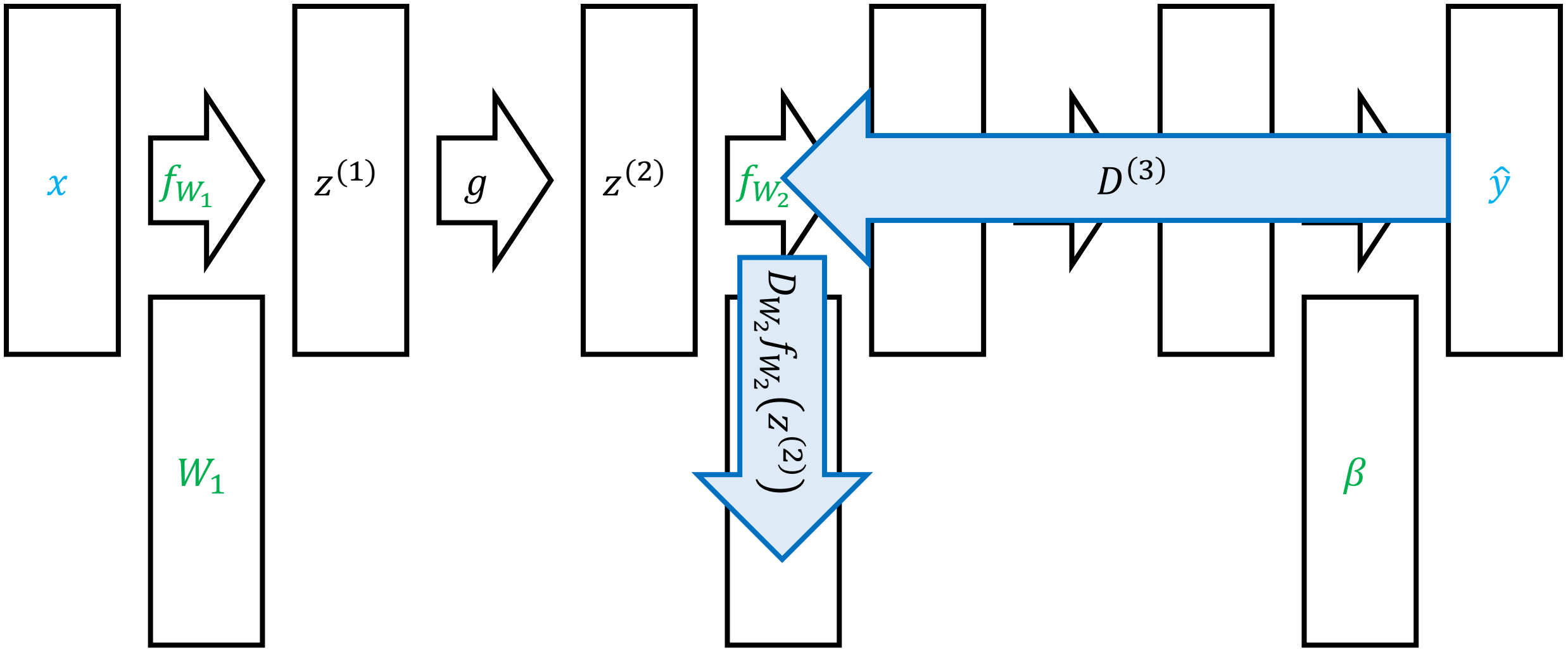
Backpropagation



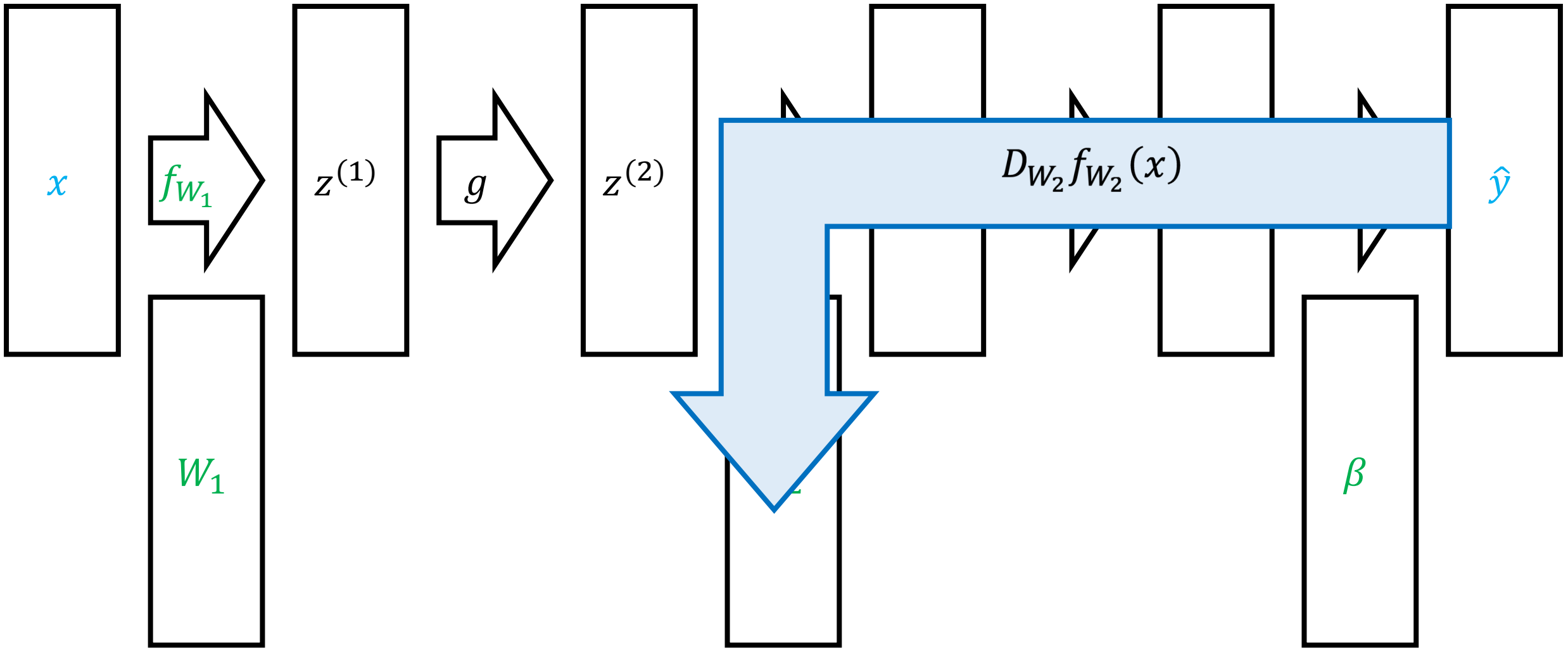
Backpropagation



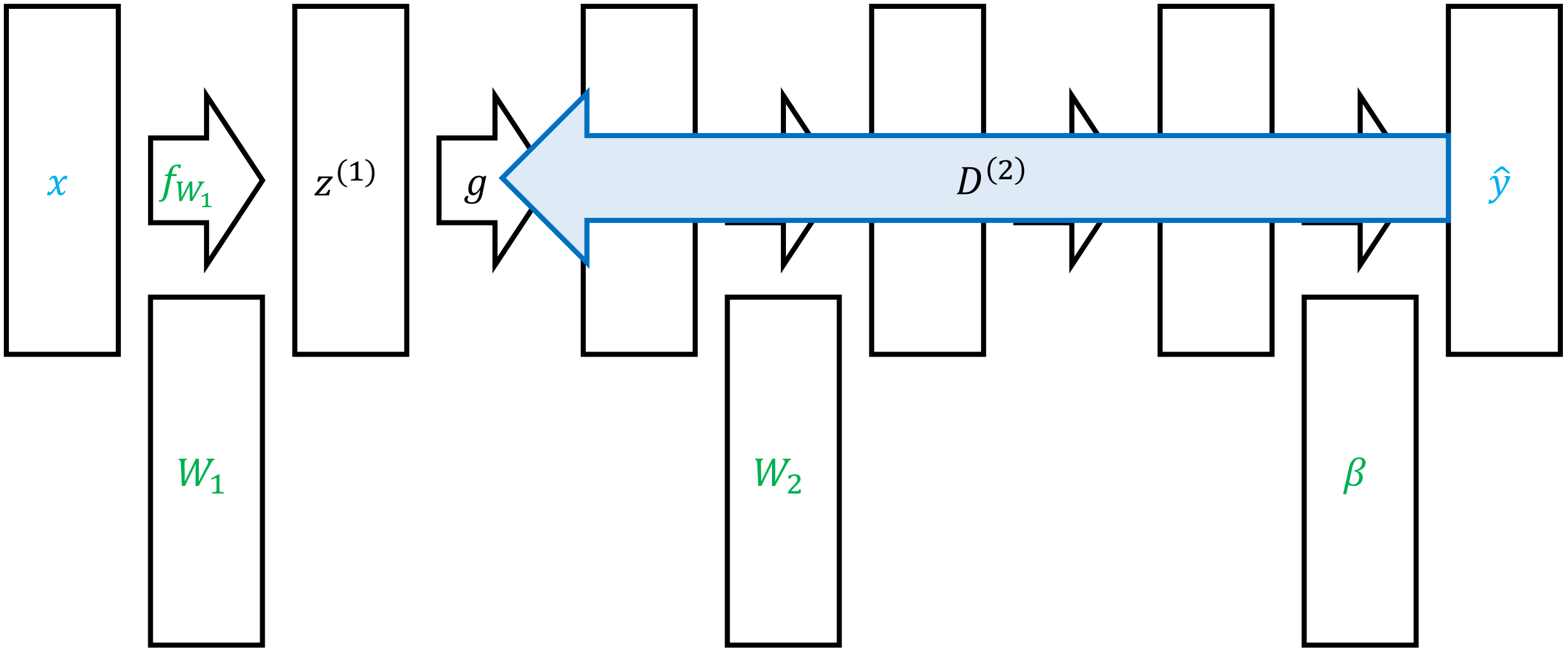
Backpropagation



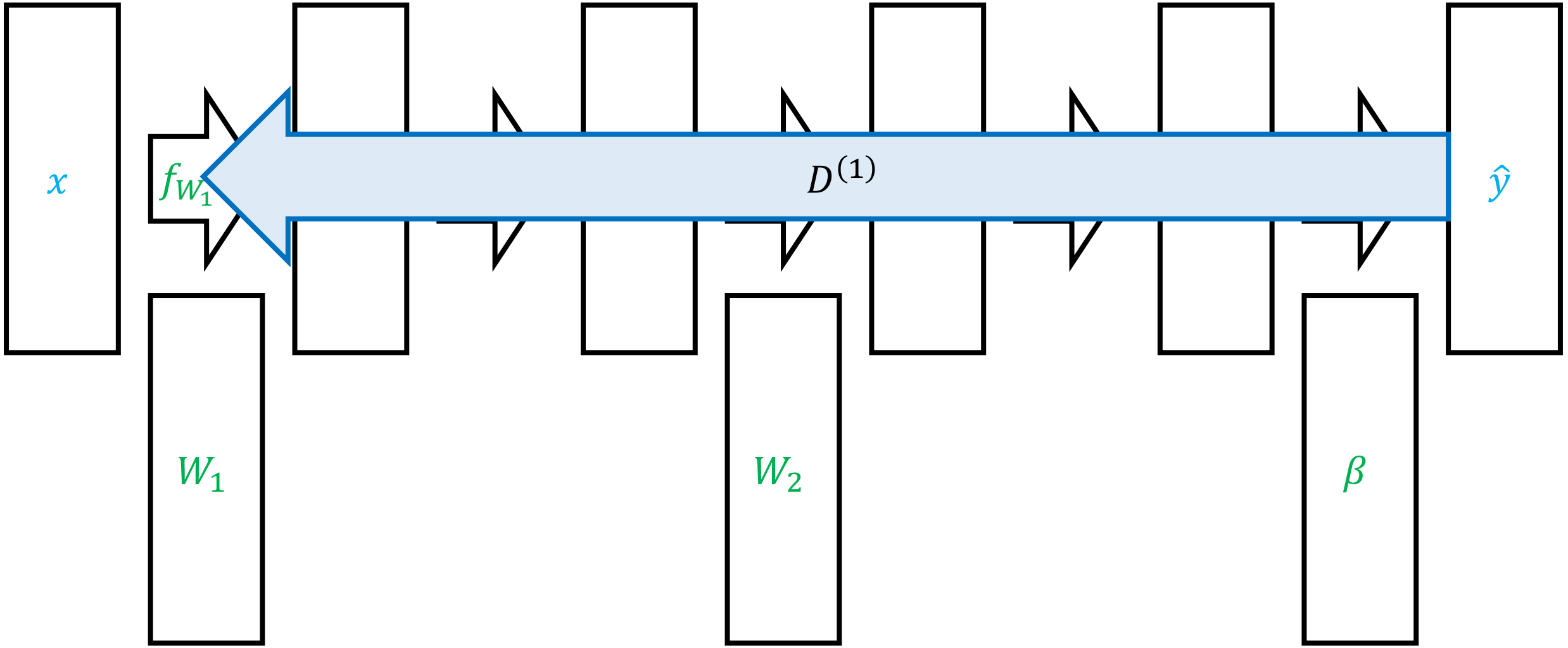
Backpropagation



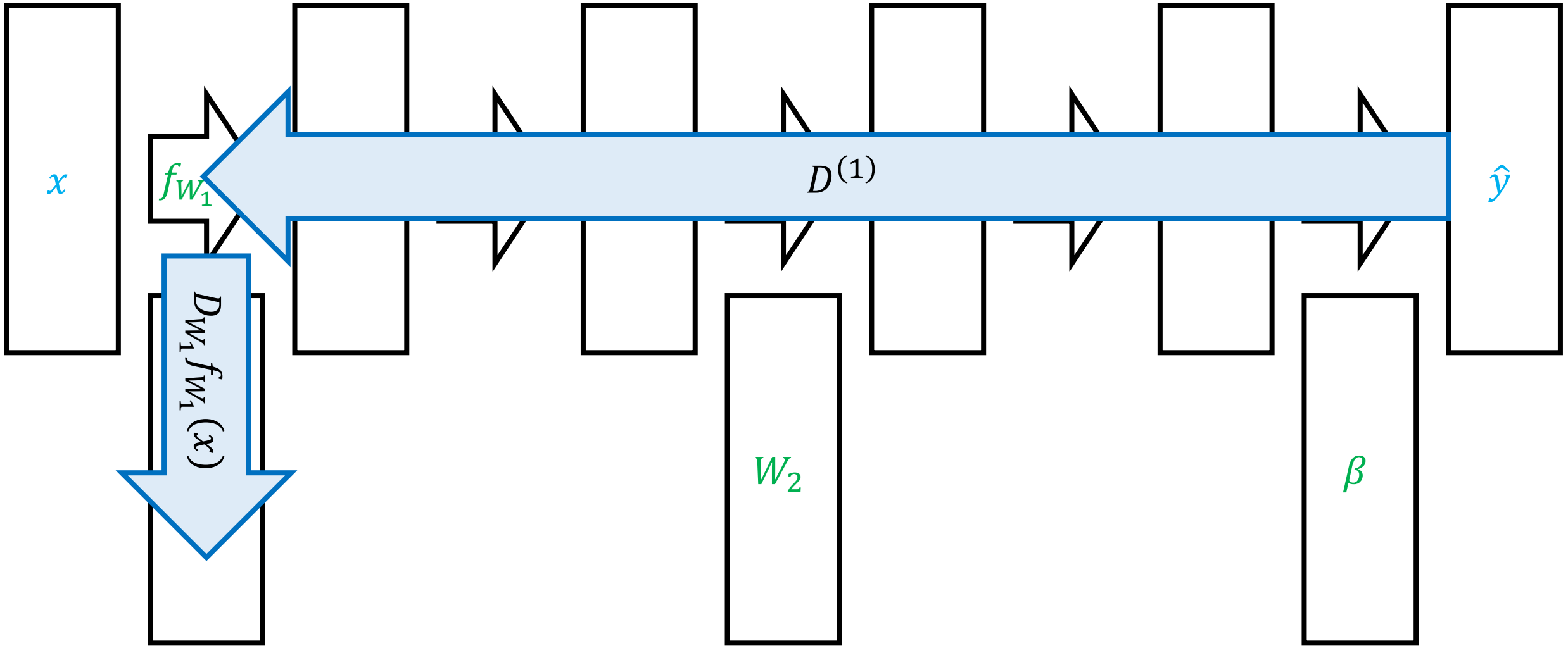
Backpropagation



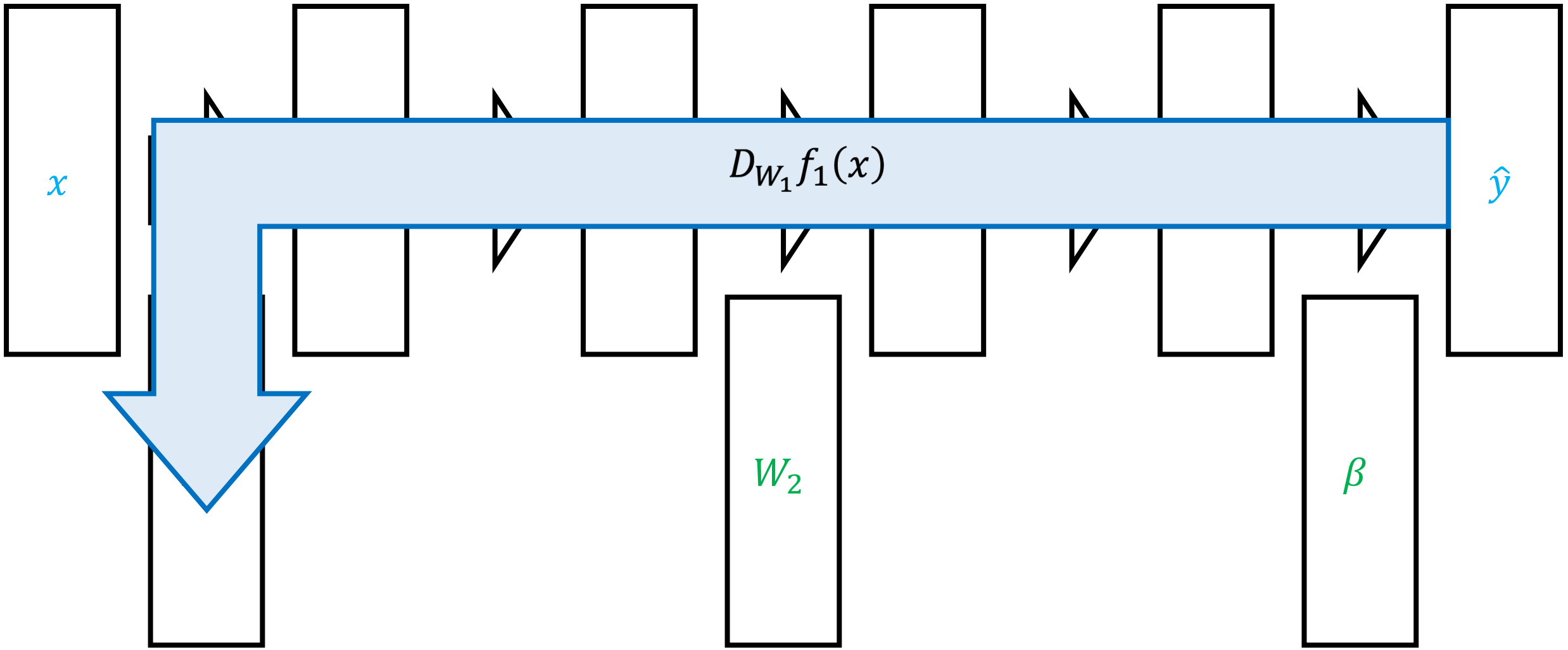
Backpropagation



Backpropagation



Backpropagation



Backpropagation Algorithm

- **Forward pass:** Compute forwards from $j = 0$ to $j = m$
  - $z^{(j)} = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}(z^{(j-1)}) & \text{if } j > 0 \end{cases}$

- **Backward pass:** Compute backwards from $j = m$ to $j = 1$
  - $D^{(j)} = \begin{cases} 1 & \text{if } j = m \\ D^{(j+1)} D_z f_{W_{j+1}}(z^{(j)}) & \text{if } j < m \end{cases}$
  - $D_{W_j} f_W(x) = D^{(j)} D_{W_j} f_{W_j}(z^{(j-1)})$

- **Final output:** $\nabla_{W_j} L(f_W(x), y)^\top = \nabla_{\hat{y}} L(z^{(m)}, y)^\top D_{W_j} f_W(x)$ for each j

Gradient Descent

- $W_1 \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2, \ldots\}$ **until** convergence:

$$W_{t+1,j} \leftarrow W_{t,j} - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_{W_j} L\left(f_{W_t}(x_i), y_i\right) \quad \text{(for each } j\text{)}$$

- **return** f_{W_t}

Gradient Descent

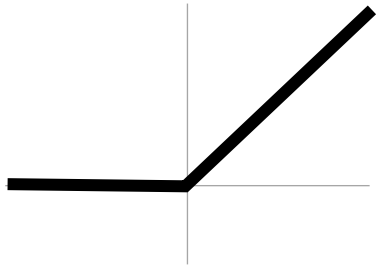
- $W_1 \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2, \ldots\}$ **until** convergence:
  - Compute gradients $\nabla_{W_j} L\big(f_{W_t}(x_i), y_i\big)$ using backpropagation
  - Update parameters:

$$W_{t+1,j} \leftarrow W_{t,j} - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_{W_j} L\big(f_{W_t}(x_i), y_i\big) \quad (\text{for each } j)$$

- **return** f_{W_t}

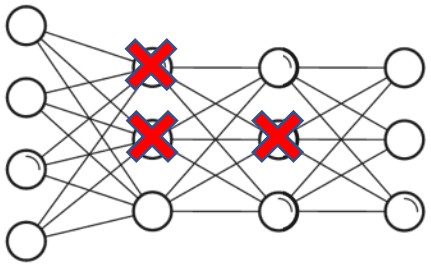
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

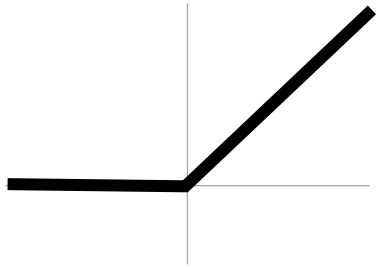


Dropout

Managing Training

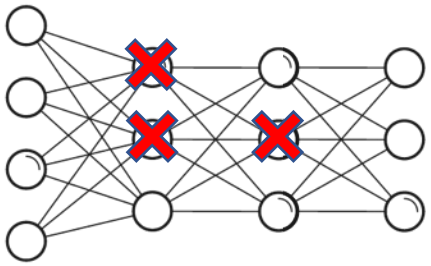
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



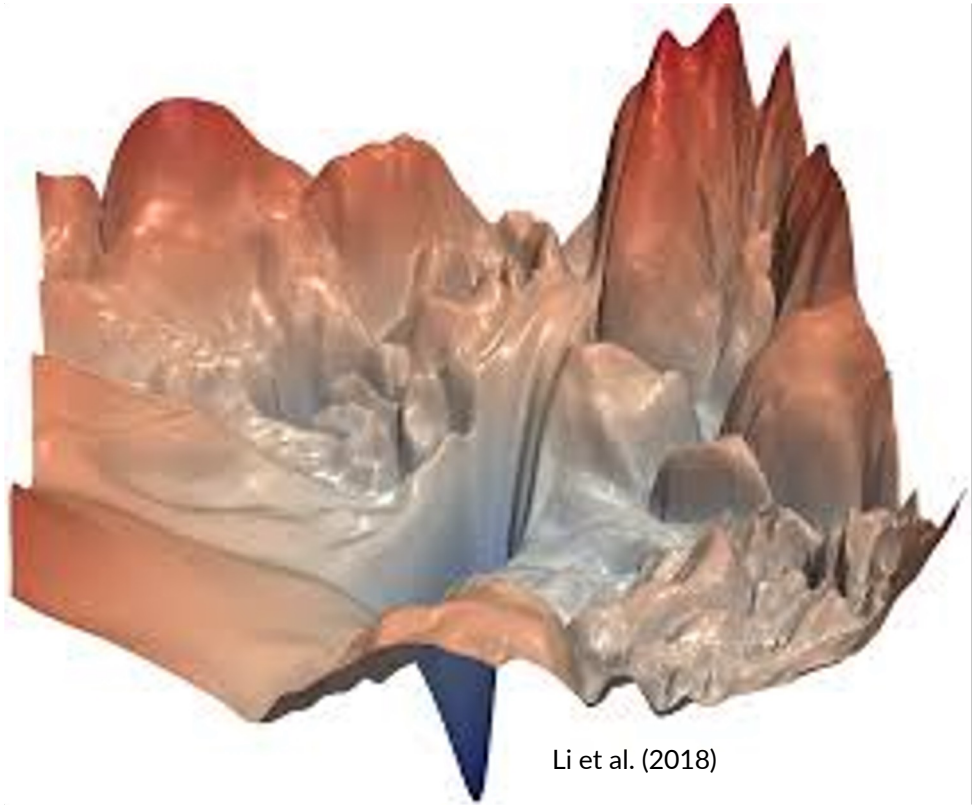
Dropout

Managing Training

Optimization Challenges

- **Challenges**
  - Local minima, saddle points due to non-convex loss
  - Exploding/vanishing gradients
  - Ill-conditioning

- Have heuristics that work in common cases (but not always)



Li et al. (2018)

Gradient Descent

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\ldots,T\}$:

$$\beta \leftarrow \beta - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_\beta L\big(f_\beta(x_i), y_i\big)$$

- **return** f_β

Gradient Descent

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\ldots,T\}$:

$$\beta \leftarrow \beta - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_\beta L\big(f_\beta(x_i), y_i\big)$$

- **return** f_β

Stochastic Gradient Descent

usually $T \in \{1, \dots, 10\}$

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2, \dots, T\}$:
  - **for** $i \in \{1,2, \dots, n\}$:

$$\beta \leftarrow \beta - \alpha \cdot \nabla_\beta L\big(f_\beta(x_i), y_i\big)$$

- **return** f_β

Minibatch Stochastic Gradient Descent

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\ldots,T\}$:
  - **for** $i' \in \left\{1,2,\ldots,\frac{n}{k}\right\}$:

$$\beta \leftarrow \beta - \frac{\alpha}{k} \cdot \sum_{i=i'k}^{i'(k+1)-1} \nabla_\beta L\big(f_\beta(x_i), y_i\big) \quad \text{(for each } j\text{)}$$

- **return** f_β

Accelerated Gradient Descent

- Vanilla gradient descent:

$$\beta \leftarrow \beta - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$$

- Accelerated gradient descent:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$$
$$\beta \leftarrow \beta + \rho$$

Accelerated Gradient Descent

- Vanilla gradient descent:

$$\beta \leftarrow \beta - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$$

- Accelerated gradient descent:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$$
$$\beta \leftarrow \beta + \rho$$

Accelerated Gradient Descent

- Vanilla gradient descent:

$$\beta \leftarrow \beta - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$$

- Accelerated gradient descent:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$$
$$\beta \leftarrow \beta + \rho$$

Accelerated Gradient Descent

- **Intuition:** ρ holds the previous update $\alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big)$, except it "remembers" where it was heading via momentum

- New hyperparameter μ (typically $\mu = 0.9$ or $\mu = 0.99$)

Nesterov Momentum

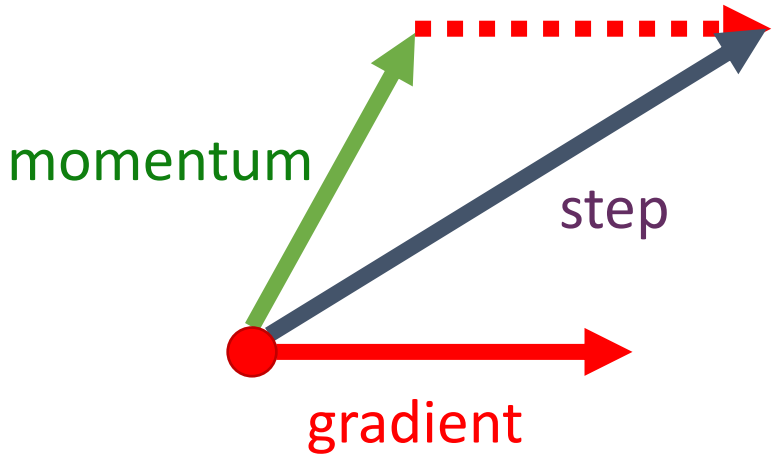
- Accelerated gradient descent:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_{\beta} L\left(f_{\beta}(x), y\right)$$
$$\beta \leftarrow \beta + \rho$$

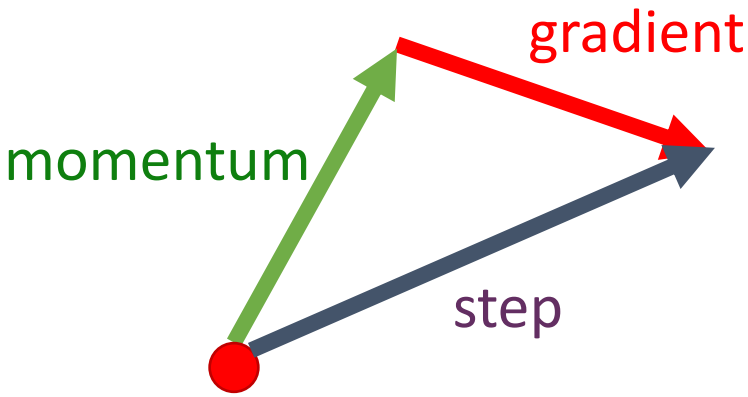
- Nesterov momentum:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_{\beta} L\left(f_{\beta+\mu\cdot\rho}(x), y\right)$$
$$\beta \leftarrow \beta + \rho$$

Nesterov Momentum



vanilla momentum



Nesterov momentum

"Lookahead" helps avoid overshooting when close to the optimum

Adaptive Learning Rates

- **AdaGrad:** Letting $g = \nabla_\beta L\big(f_\beta(x), y\big)$, we have

$$G \leftarrow G + g^2 \quad \text{and} \quad \beta \leftarrow \beta - \frac{\alpha}{\sqrt{G}} \cdot g$$

- **RMSProp:** Use exponential moving average instead:

$$G \leftarrow \lambda \cdot G + (1-\lambda) g^2 \quad \text{and} \quad \beta \leftarrow \beta - \frac{\alpha}{\sqrt{G}} \cdot g$$

Adaptive Learning Rates

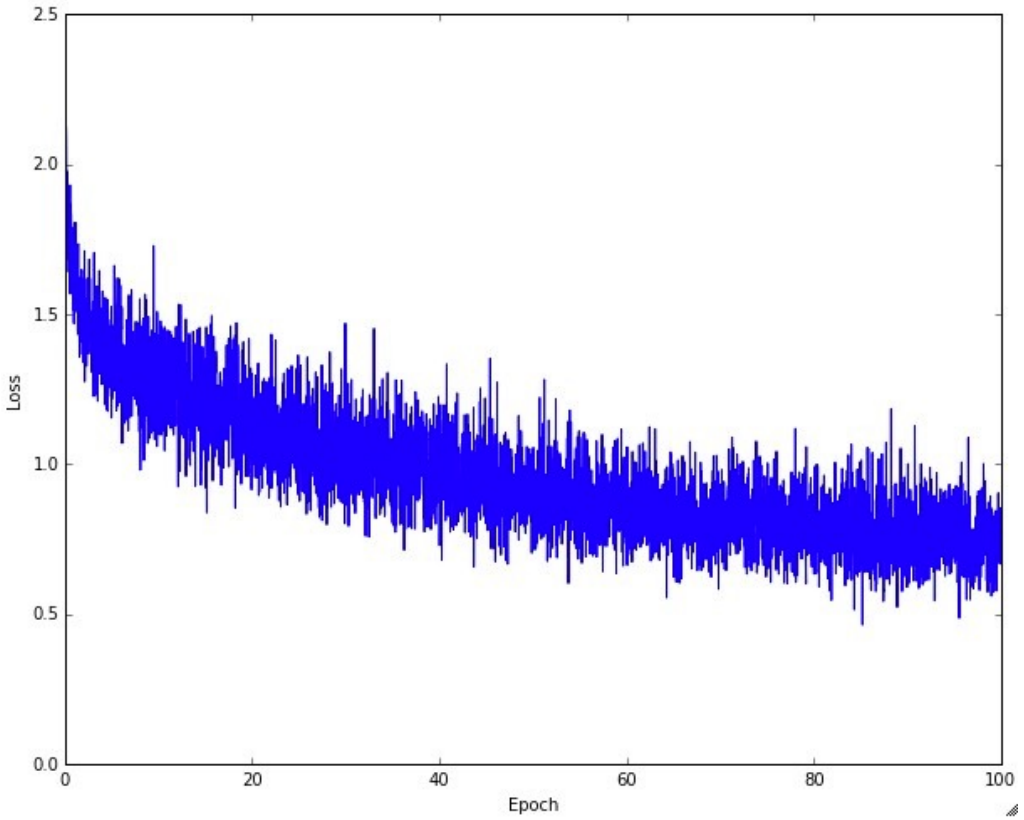
- **Adam:** Similar to RMSprop, but with both the first and second moments of the gradients

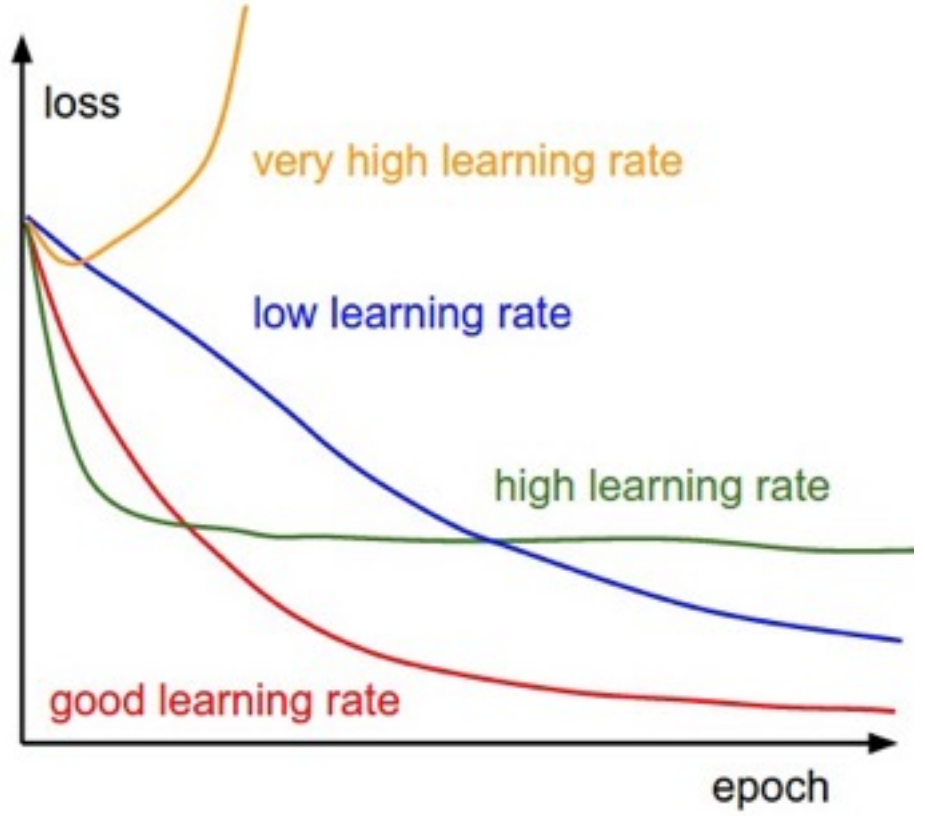
$$G \leftarrow \lambda \cdot G + (1 - \lambda) \cdot g^2$$
$$g' \leftarrow \lambda' \cdot g' + (1 - \lambda') \cdot g$$
$$\beta \leftarrow \beta - \frac{g'}{\sqrt{G}}$$

- **Intuition:** RMSProp with momentum
- Most commonly used optimizer

Learning Rate

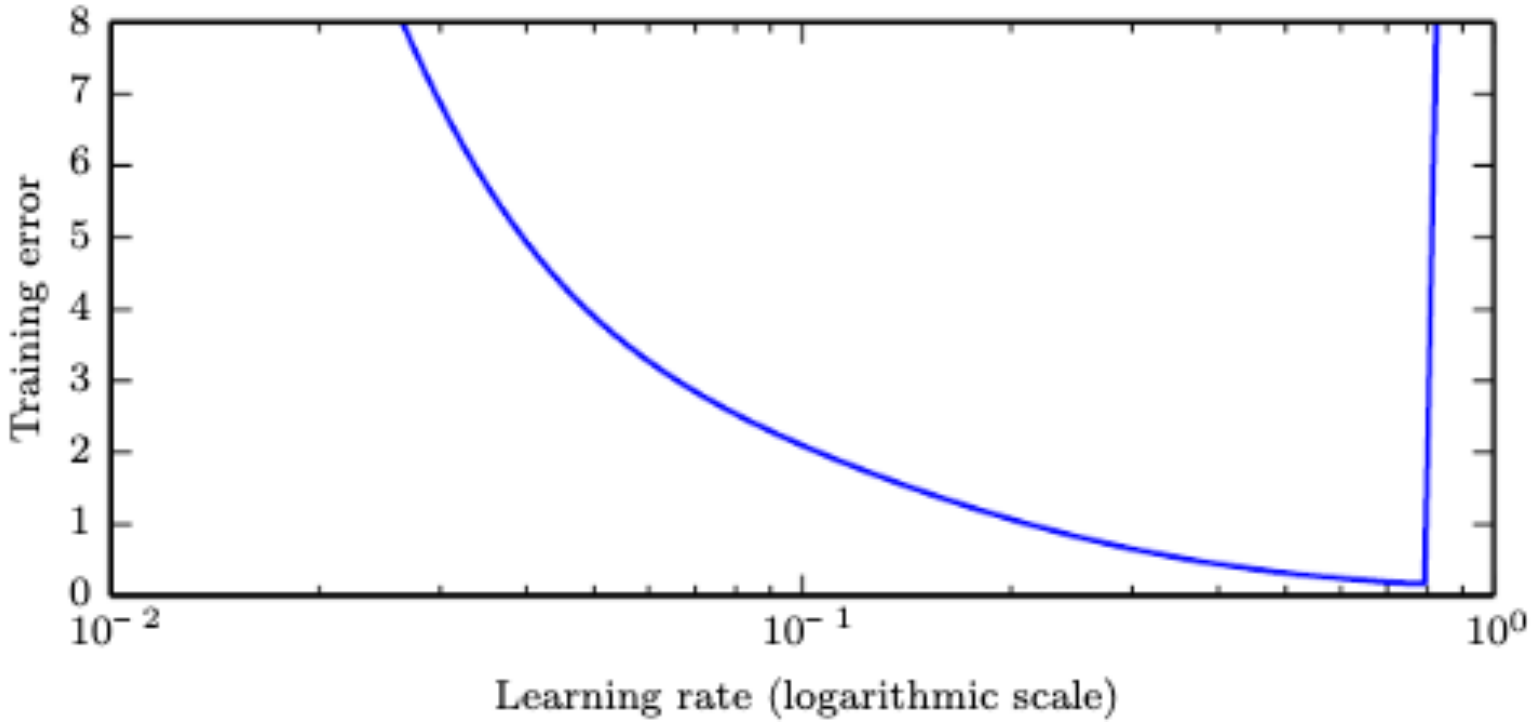
- Most important hyperparameter; tune by looking at training loss





Learning Rate

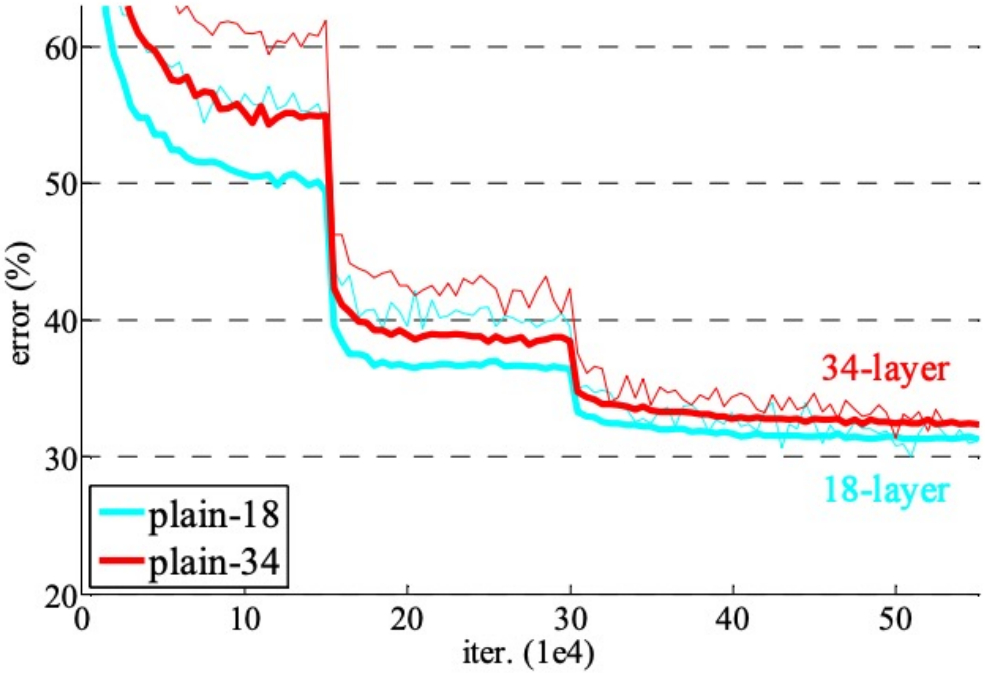
- Learning rate vs. training error:



Goodfellow et al, Deep Learning Book, 2019

Learning Rate

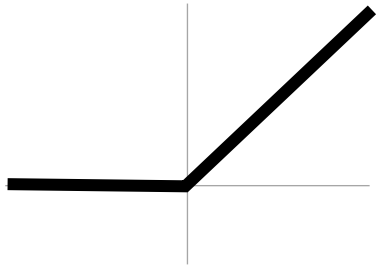
- **Schedules:** Reducing the learning rate every time the validation loss stagnates can be very effective for training



He et al, Residual Networks, 2015

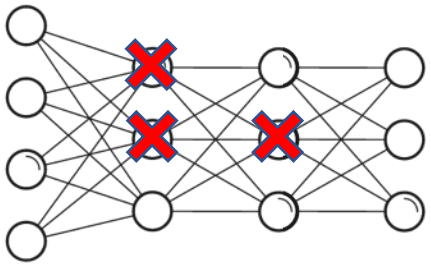
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

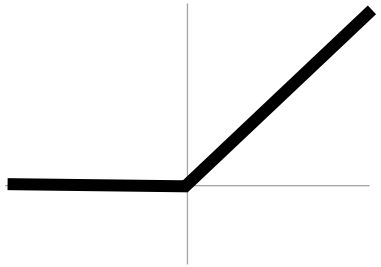


Dropout

Managing Training

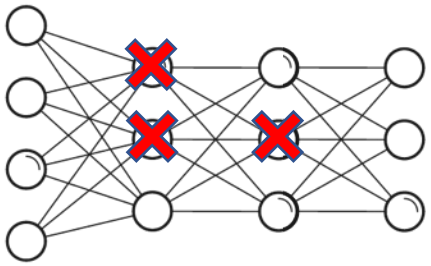
Neural Network Tips & Tricks

Optimization



Activation Functions

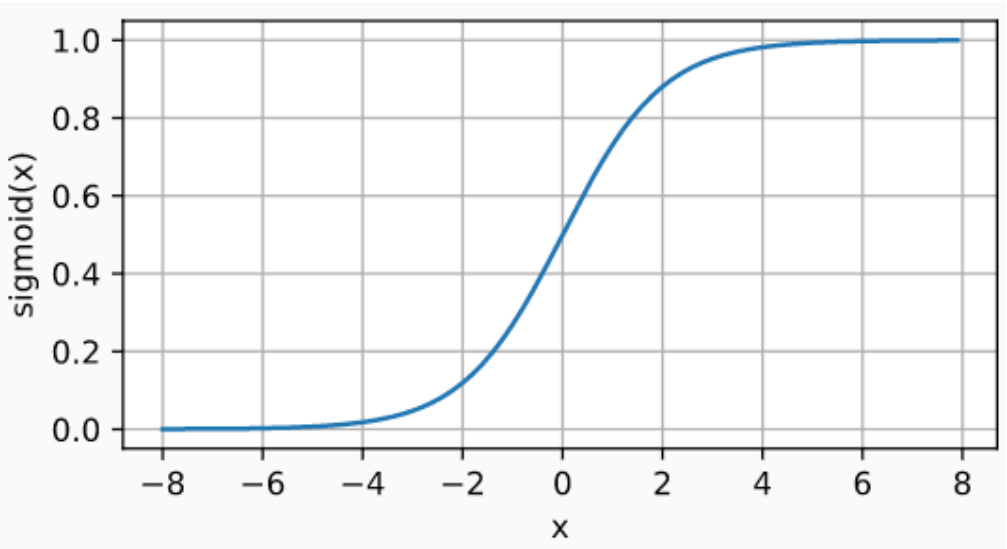
Managing Weights



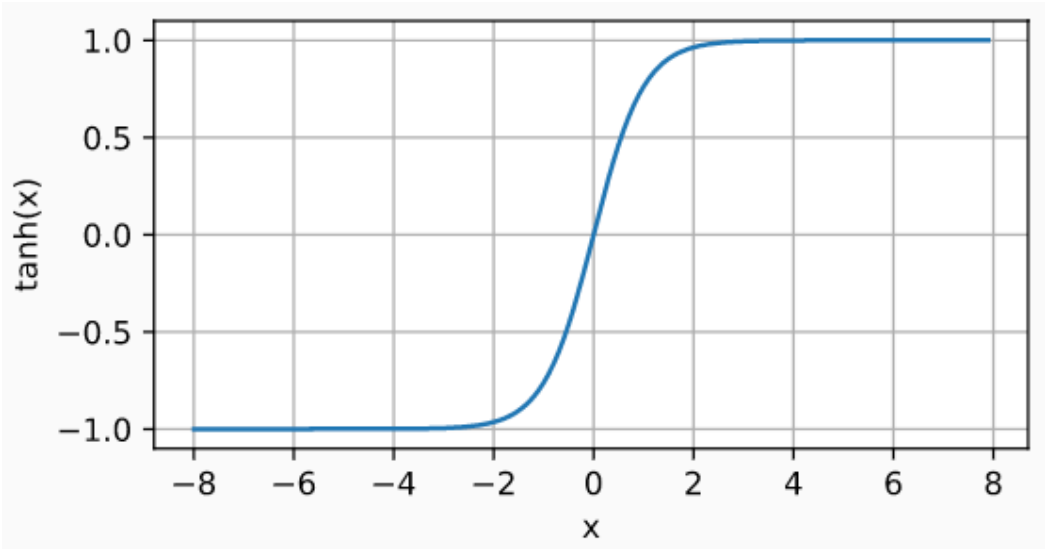
Dropout

Managing Training

Historical Activation Functions



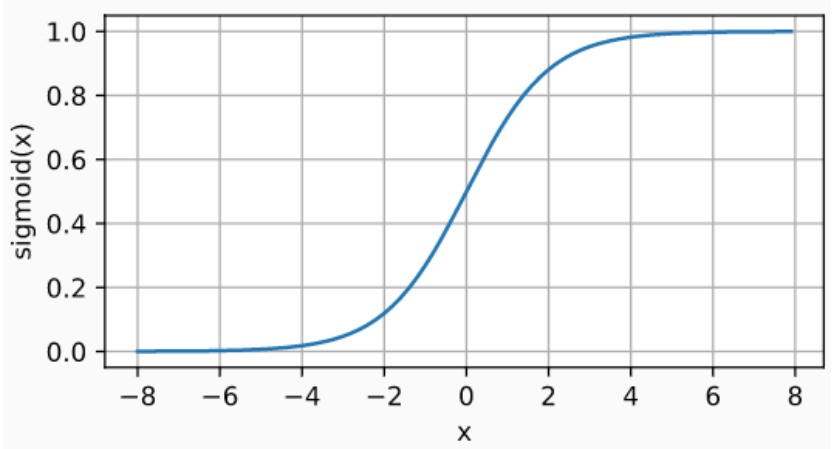
sigmoid



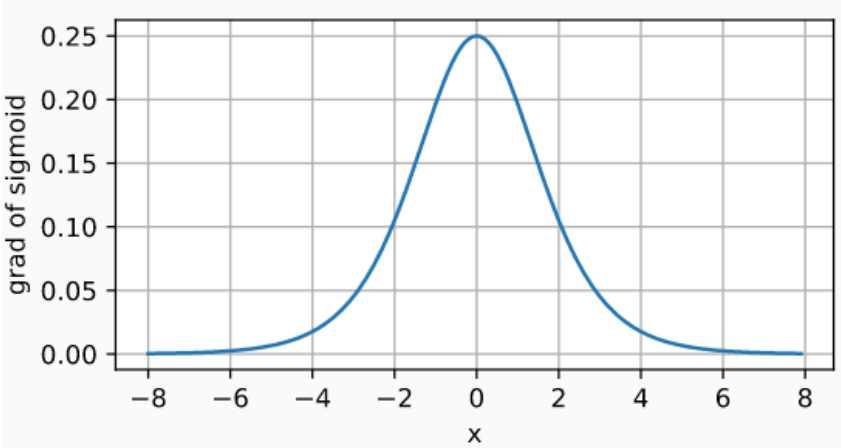
tanh

Vanishing Gradient Problem

- The gradient of the sigmoid function is often nearly zero
- **Recall:** In backpropagation, gradients are products of $\partial_z g\left(z^{(j)}\right)$
- **Quickly multiply to zero!**
  - Early layers update very slowly



sigmoid



sigmoid gradient

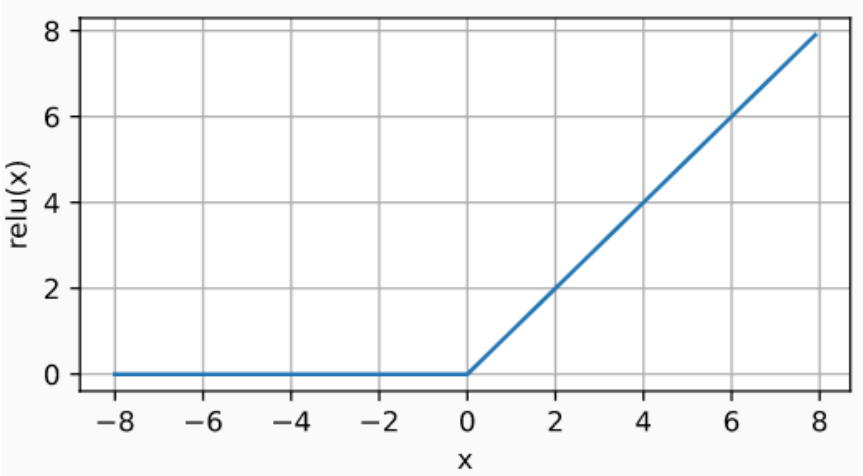
ReLU Activation

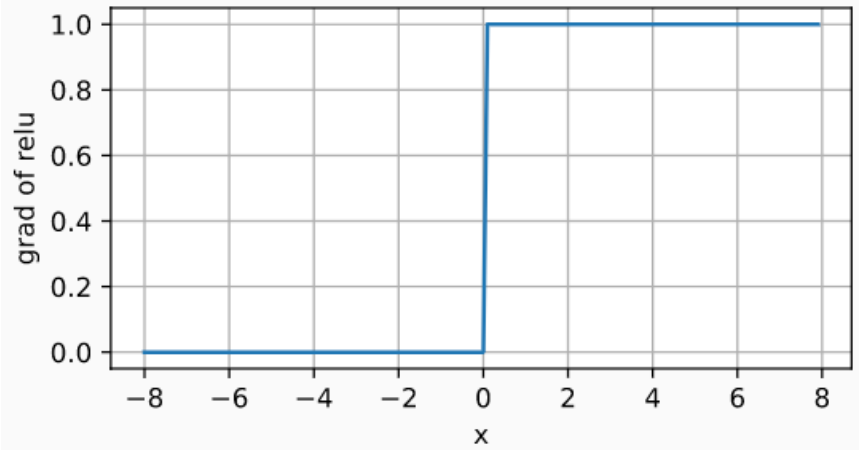
- Activation function

$$g(z) = \max\{0, z\}$$

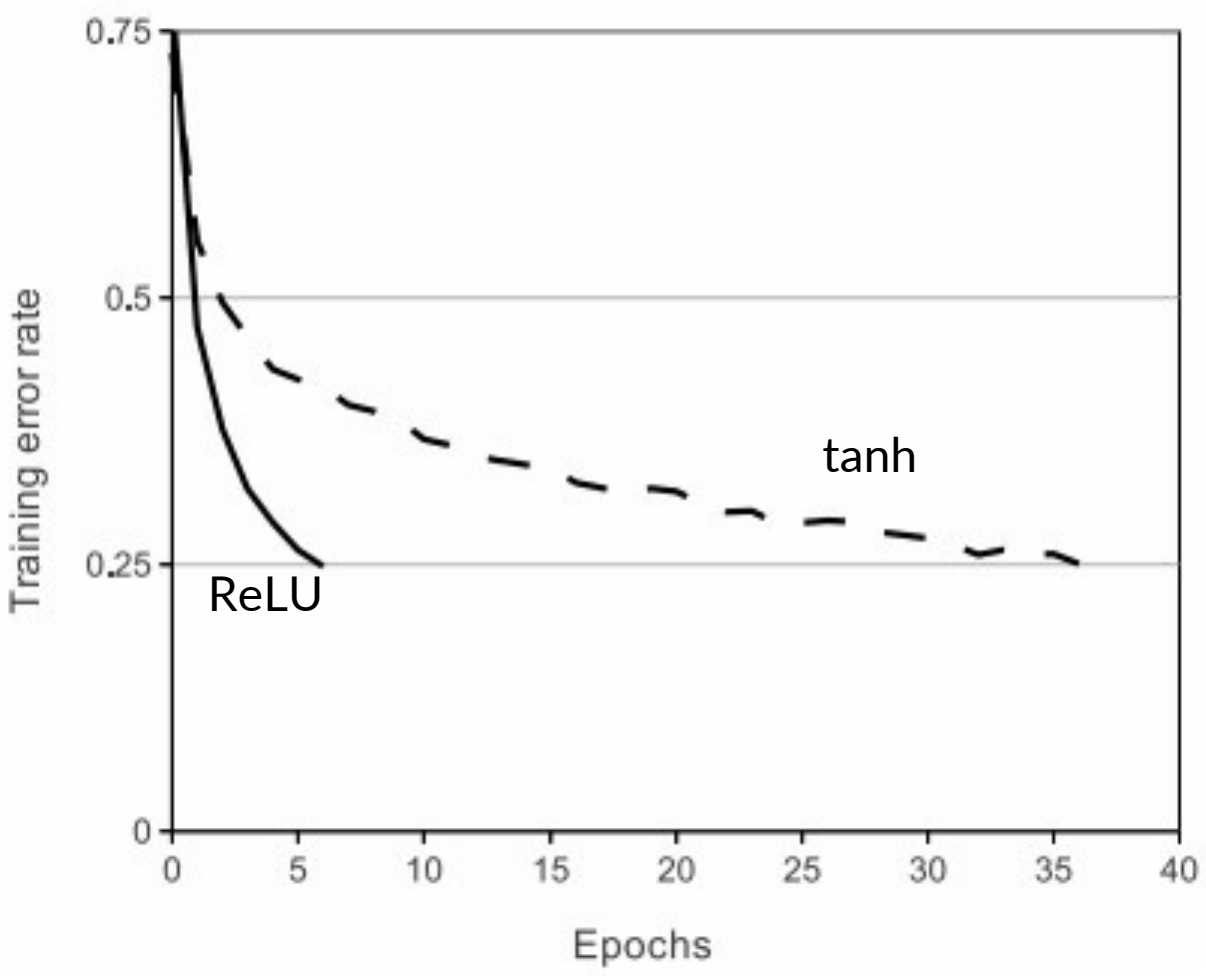
- Gradient now positive on the entire region $z \geq 0$

- Significant performance gains for deep neural networks

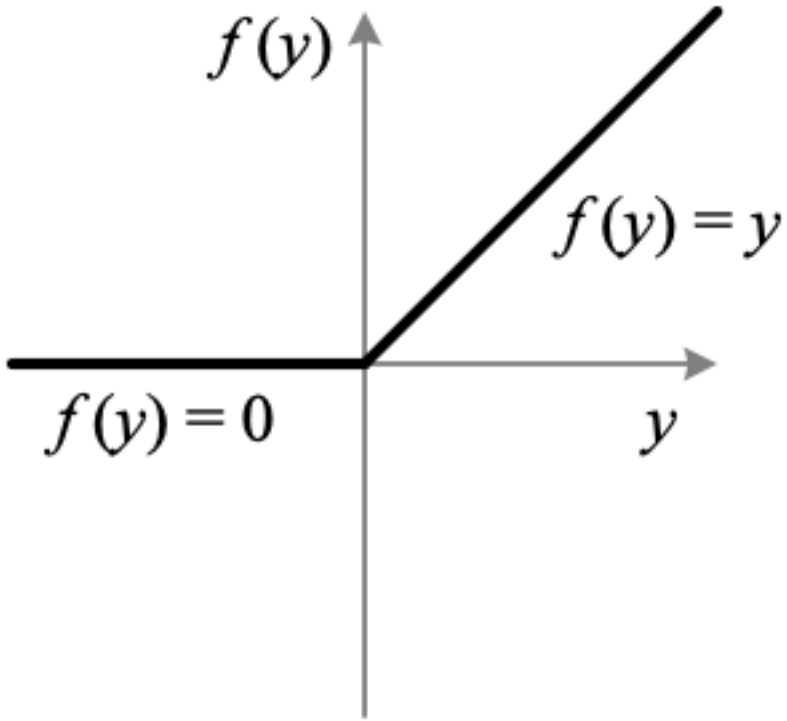


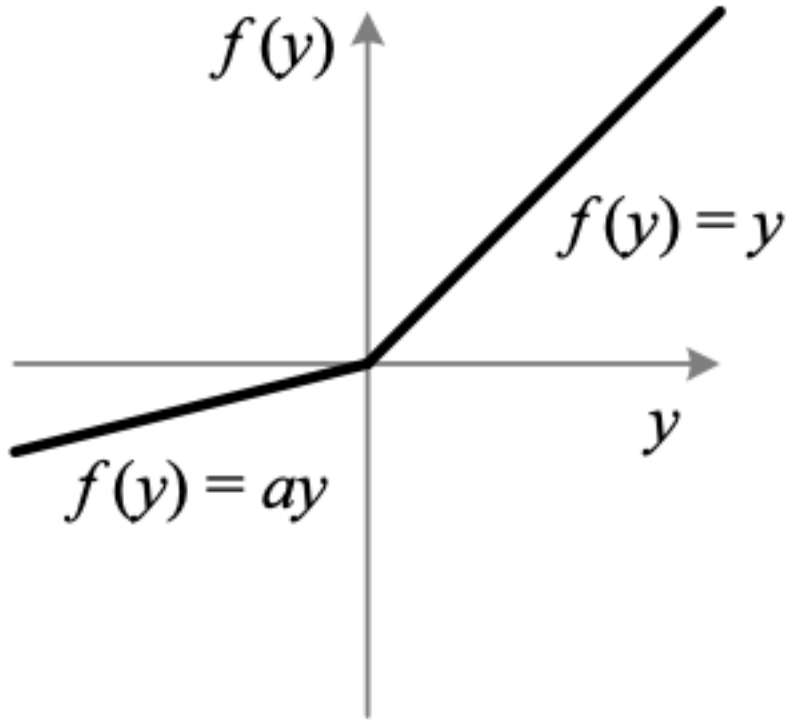


ReLU Activation



PRReLU Activation



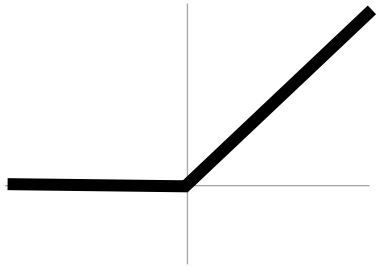


Activation Functions

- ReLU is a good standard choice
- Tradeoffs exist, and new activation functions are still being proposed

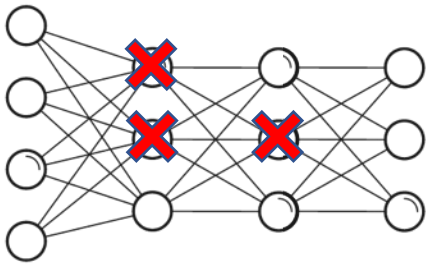
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

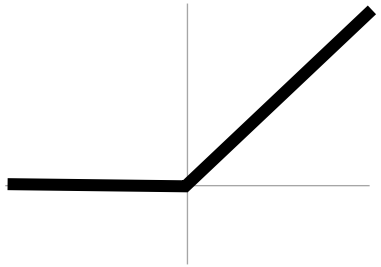


Dropout

Managing Training

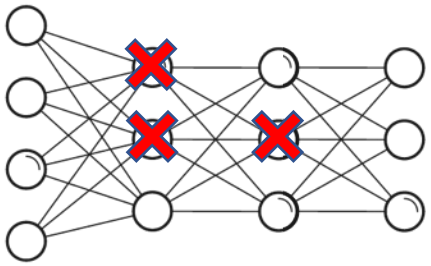
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

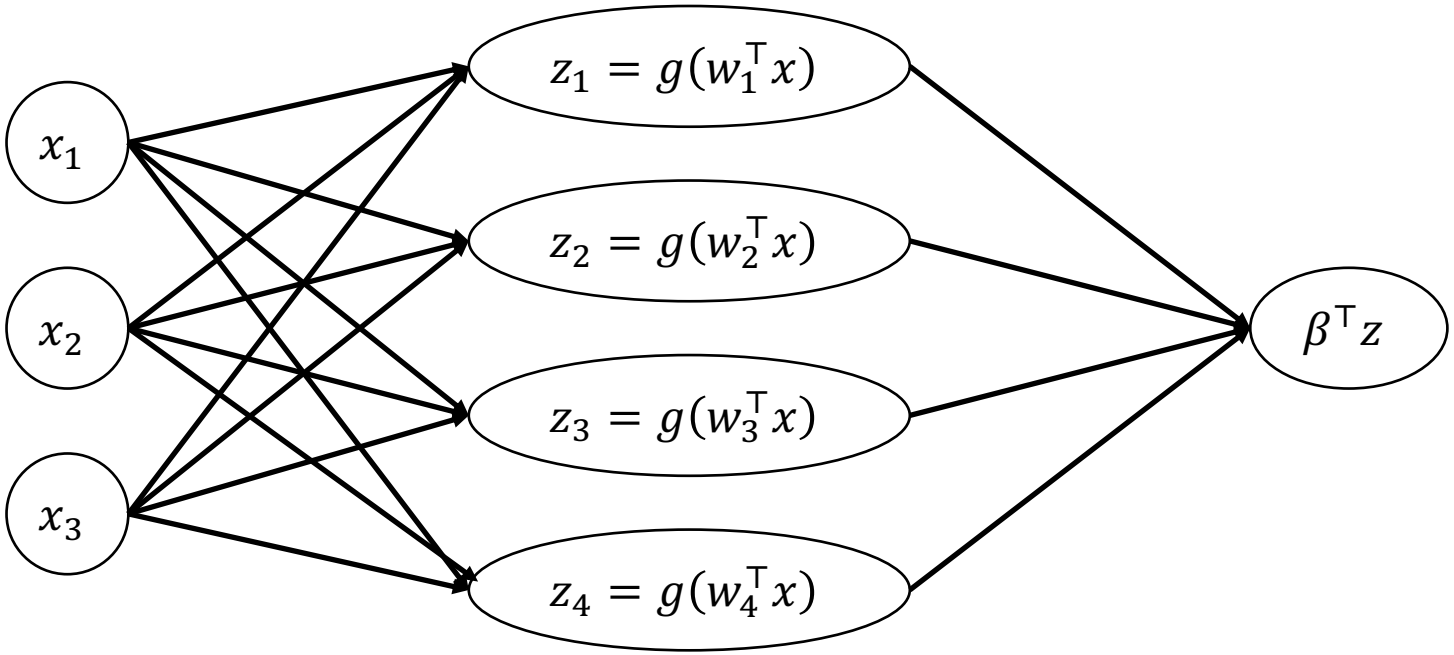


Dropout

Managing Training

Weight Initialization

- **Zero initialization:** Very bad choice!
  - All neurons $z_i = g\left(w_i^\top x\right)$ in a given layer remain identical
  - **Intuition:** They start out equal, so their gradients are equal!



Weight Initialization

- Long history of initialization tricks for W_j based on "fan in" d_{in}
  - Here, d_{in} is the dimension of the input of layer W_j
  - **Intuition:** Keep initial layer inputs $z^{(j)}$ in the "linear" part of sigmoid
  - **Note:** Initialize intercept term to 0

- **Kaiming initialization (also called "He initialization")**
  - For ReLU activations, use $W_j \sim N\left(0, \frac{2}{d_{\text{in}}}\right)$

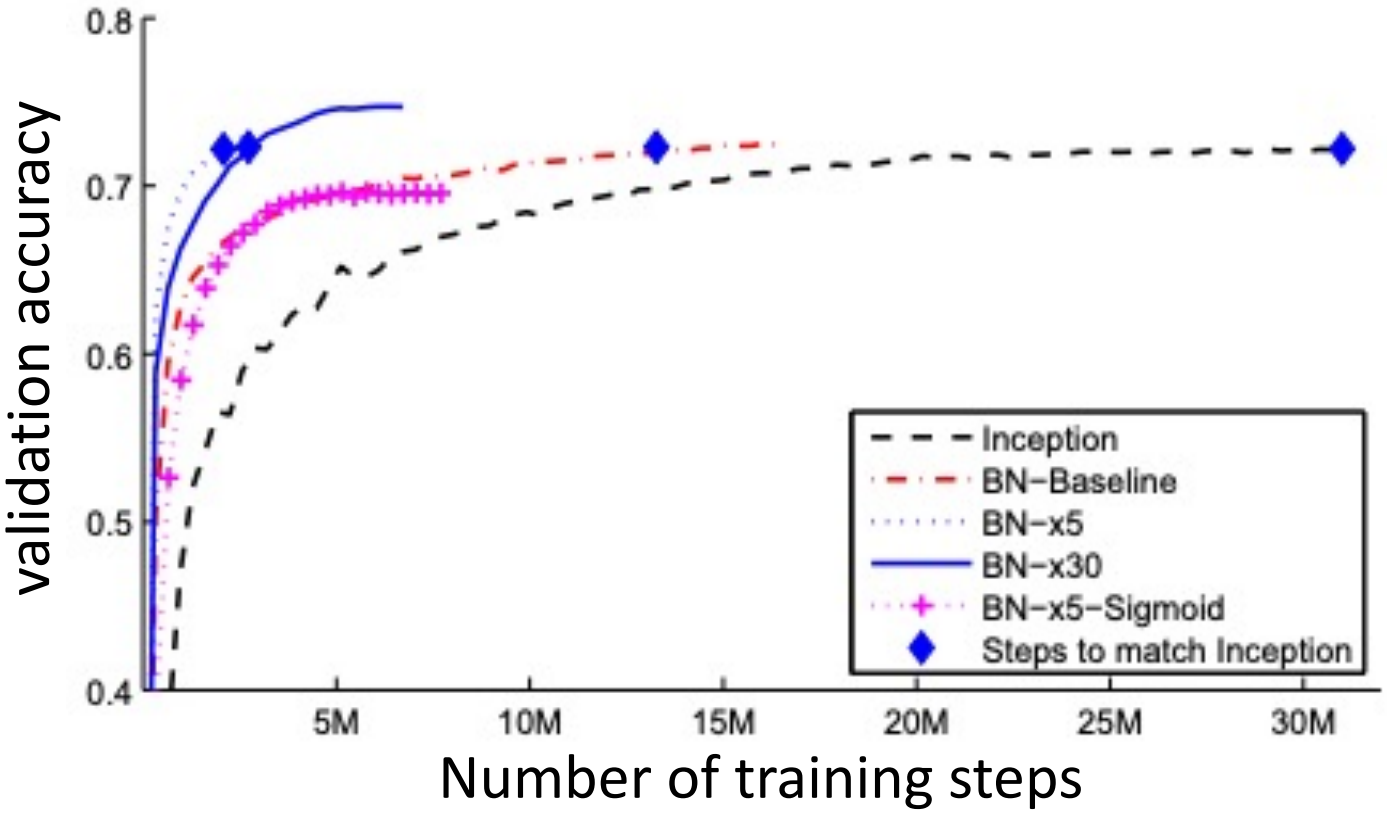
- **Xavier initialization**
  - For tanh activations, use $W_j \sim N\left(0, \frac{1}{d_{\text{in}}+d_{\text{out}}}\right)$ (d_{out} is output dimension)

Batch Normalization

- **Problem**
  - During learning, the distribution of inputs to each layer are shifting (since the layers below are also updating)
  - This “covariate shift’’ slows down learning

- **Solution**
  - As with feature standardization, standardize inputs to each layer to $N(0, I)$
  - **Batch norm:** Compute mean and standard deviation of current minibatch and use it to normalize the current layer $z^{(j)}$ (this is differentiable!)
  - **Note:** Needs nontrivial mini-batches or will divide by zero
  - Apply after every layer (before or after activation; after can work better)

Batch Normalization



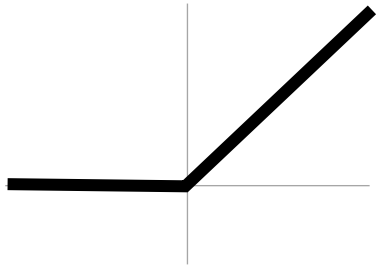
Regularization

- Can use L_1 and L_2 regularization as before
  - As before, do not regularize any of the intercept terms!
  - L_2 regularization more common

- Applied to "unrolled" weight matrices
  - Equivalently, Frobenius norm $\left\|W_j\right\|_F = \sum_{i=1}^{k} \sum_{i'=1}^{h} W_{i,i'}^2$

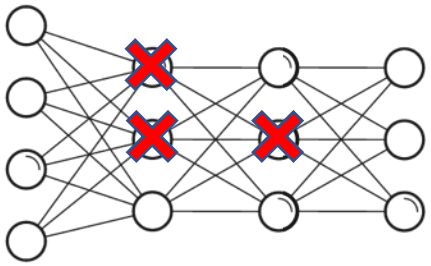
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

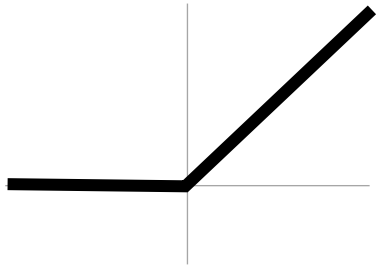


Dropout

Managing Training

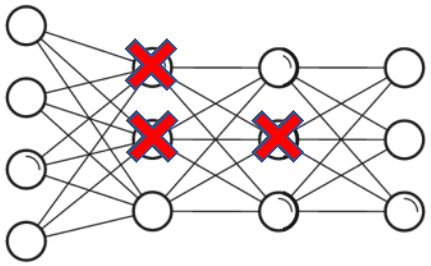
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



Dropout

Managing Training

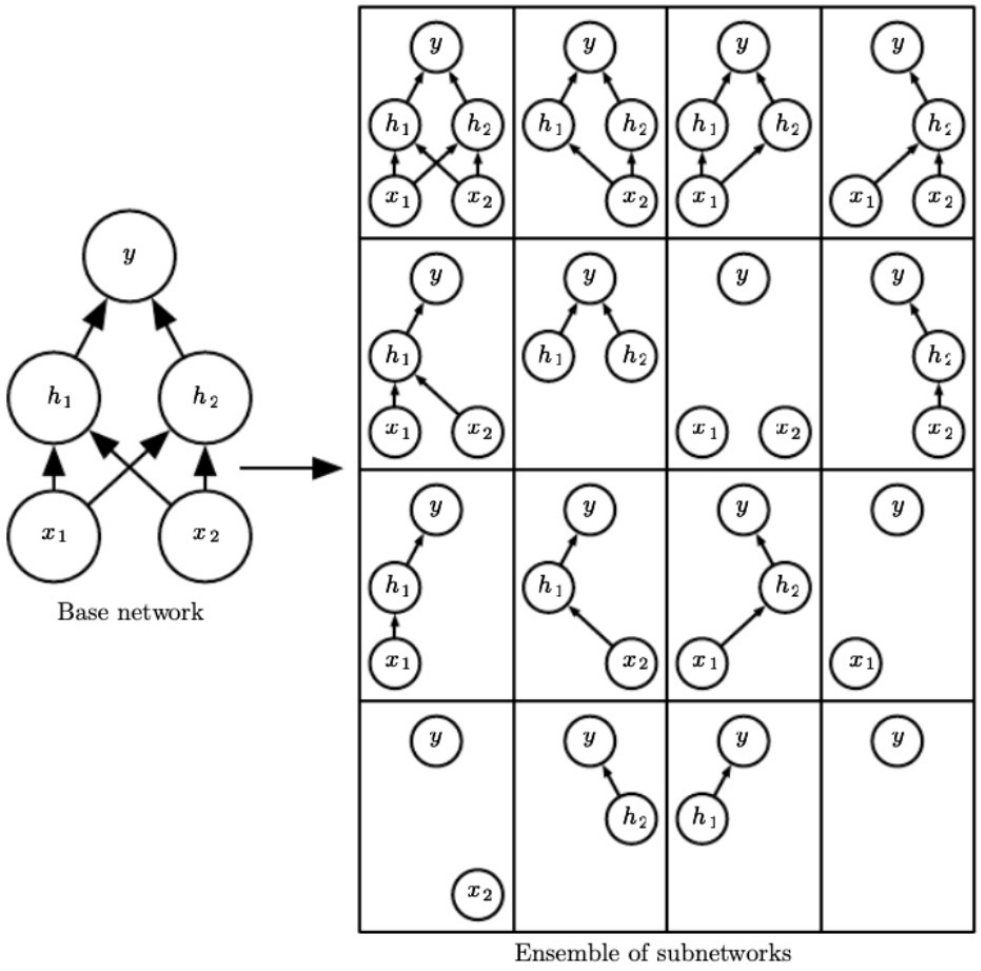
Dropout

- **Idea:** During training, randomly "drop" (i.e., zero out) a fraction p of the neurons $z_i^{(j)}$ (usually take $p = \frac{1}{2}$)

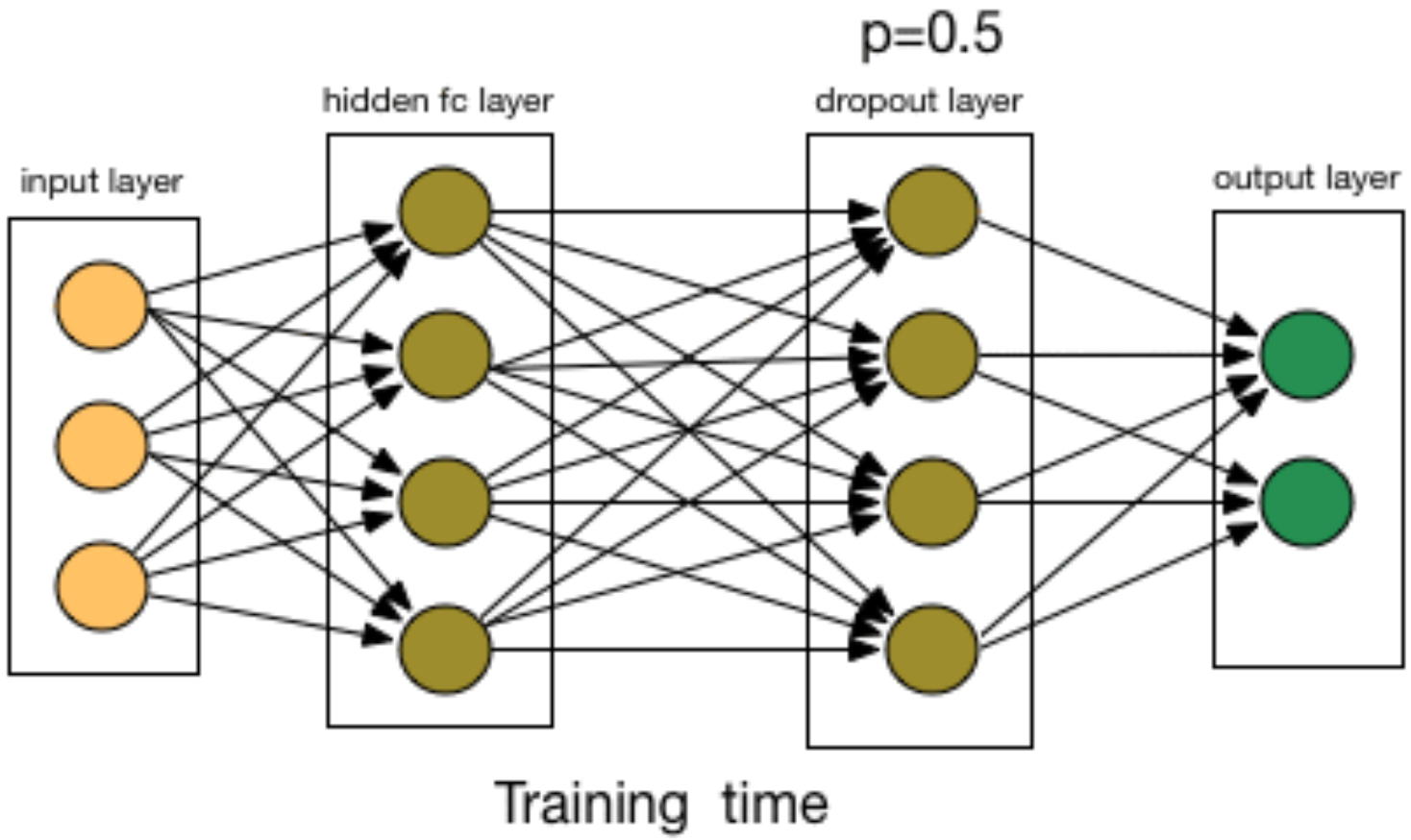
- Implemented as its own layer

$$\text{Dropout}(z) = \begin{cases} z & \text{with prob. } p \\ 0 & \text{otherwise} \end{cases}$$

- Usually include it at a few layers just before the output layer



Dropout


p=0.5
hidden fc layer
dropout layer
input layer
output layer
Training time

Dropout

- **Intuition:** A form of regularization
  - Encourages robustness to missing information from the previous layer
  - Each neuron works with many different kinds of inputs
  - Makes them more likely to be individually competent

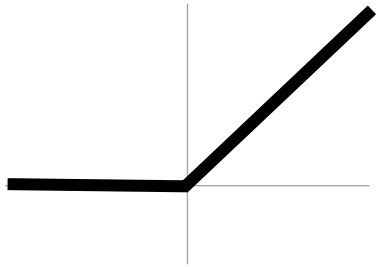
- **Connection to ensembles**
  - Each training iteration is training a slightly different network, selected at random out of $2^{\#\text{neurons}}$ networks!
  - Since the networks share weights, training one network updates others

Dropout at Test Time

- **Naïve strategy:** Stop dropping neurons
  - **Problem:** Not the distribution the layer was trained on (covariate shift)!
- **Naïve strategy:** Average across all possible predictions
  - **Problem:** There are $2^{\#\text{neurons}}$ possible realizations of the randomness
- **Solution:** Turn off dropout but divide the outgoing weights by 2
  - Good approximation of the geometric mean of all $2^{\#\text{neurons}}$ predictions
- **Note:** Can also leave dropout on, sample multiple realizations of the randomness, and report distribution to help quantify uncertainty

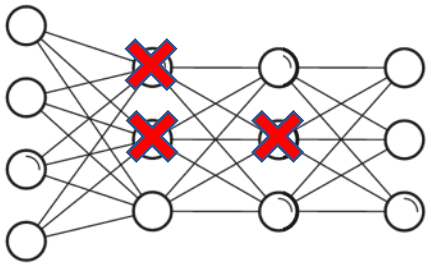
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

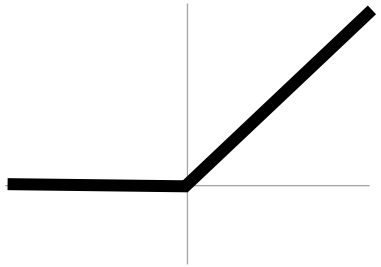


Dropout

Managing Training

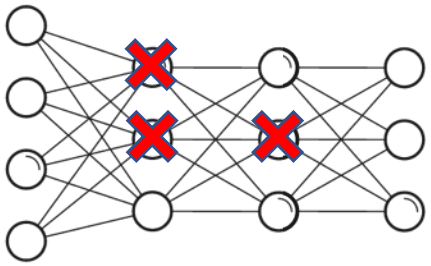
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

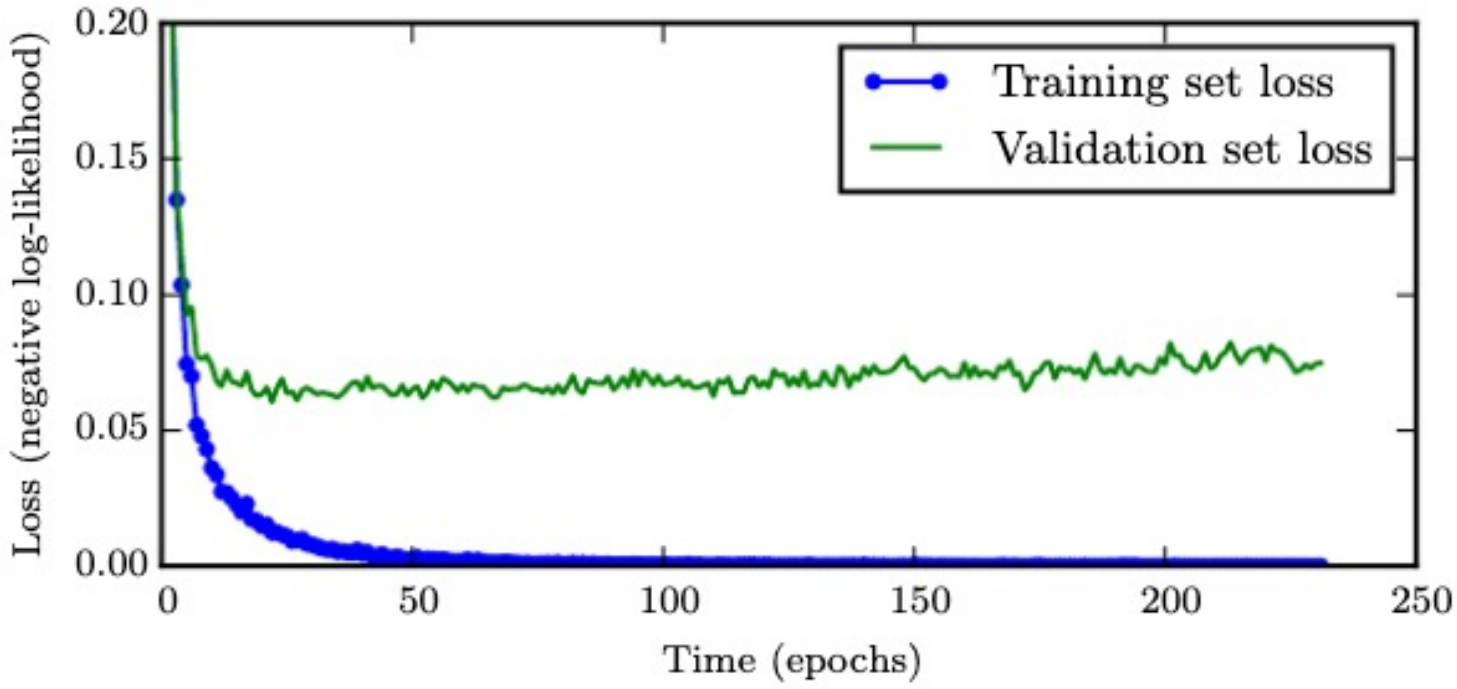


Dropout

Managing Training

Early Stopping

- Stop when your validation loss starts increasing (alternatively, finish training and choose best model on validation set)
  - Simple way to introduce regularization



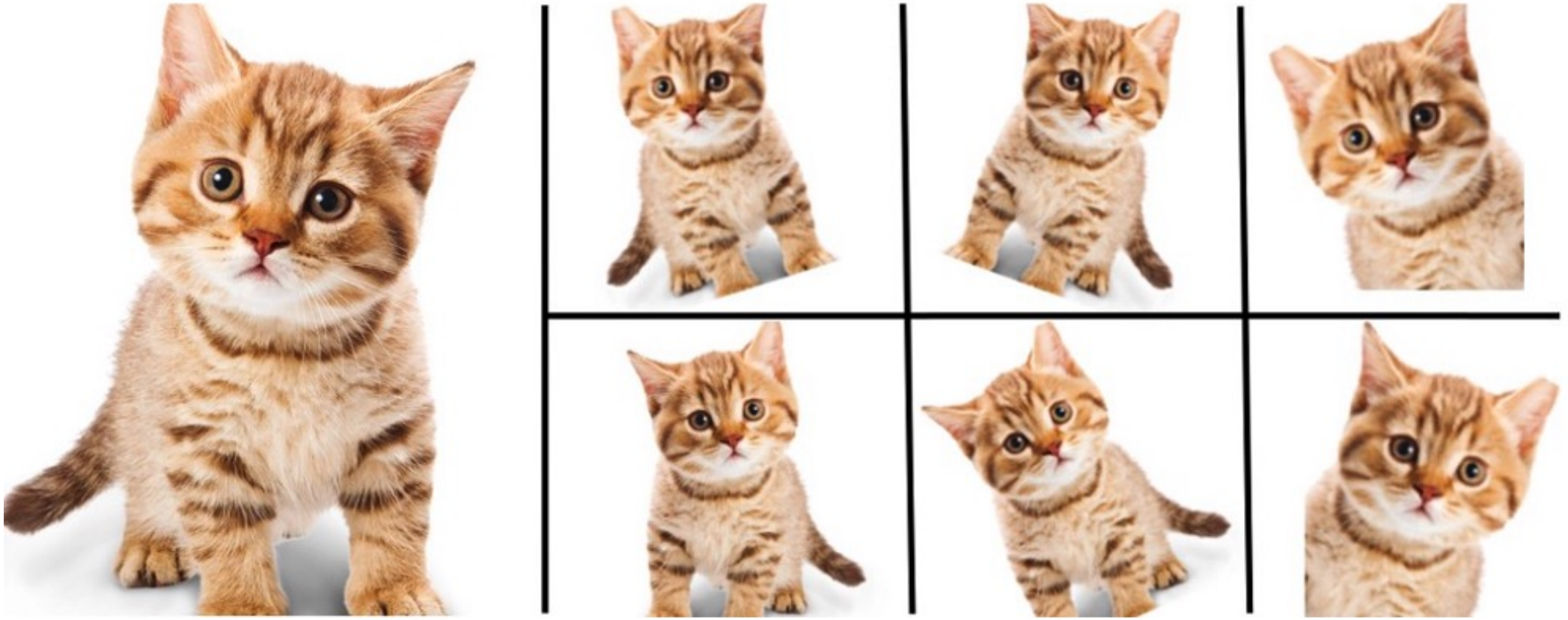
Data Augmentation

- **Data augmentation:** Generate more data by modifying training inputs

- Often used when you know that your output is robust to some transformations of your data
  - **Image domain:** Color shifts, add noise, rotations, translations, flips, crops
  - **NLP domain:** Substitute synonyms, generate examples (doesn't work as well but ongoing research direction)
  - Can combine primitive shifts

- **Note:** Labels are simply the label of original image

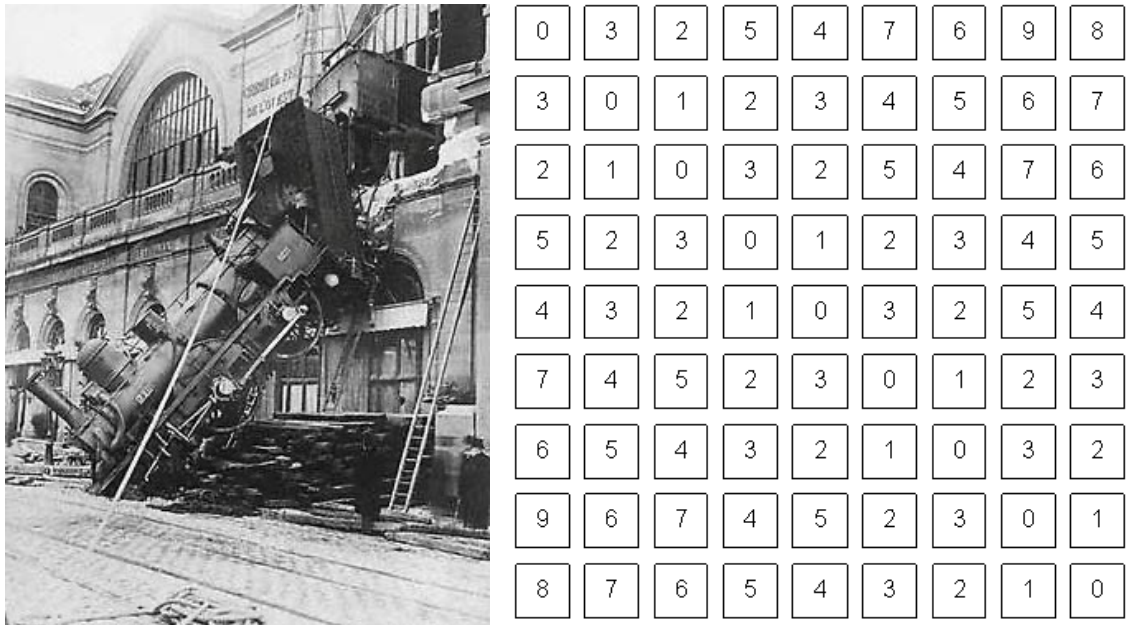
Data Augmentation



Images as 2D Arrays

- Grayscale image is a 2D array of pixel values

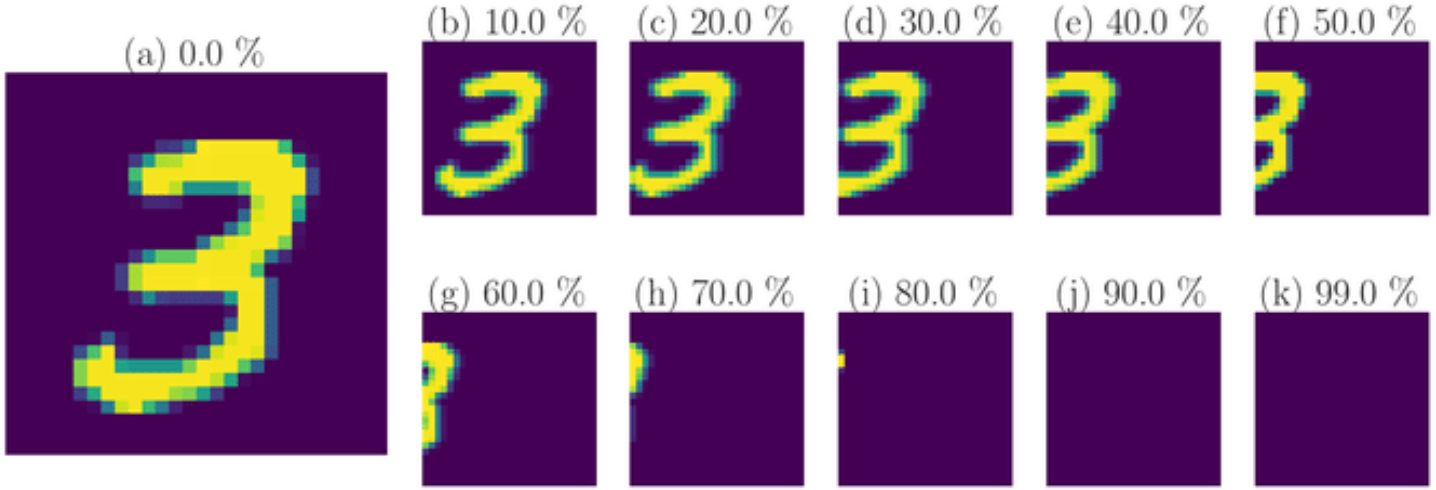
- Color images are 3D array
  - 3rd dimension is color (e.g., RGB)
  - Called “channels”



Source: S. Narasimhan, S. Lazebnik

Structure in Images

- **Translation invariance**
  - Consider image classification (e.g., labels are cat, dog, etc.)
  - **Invariance:** If we translate an image, it does not change the category label



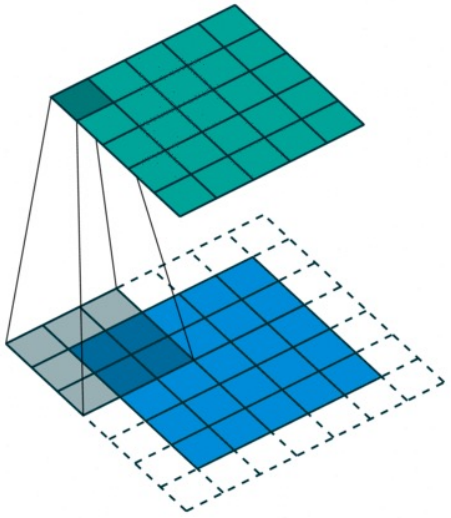
Source: Ott et al., Learning in the machine: To share or not to share?

Structure in Images

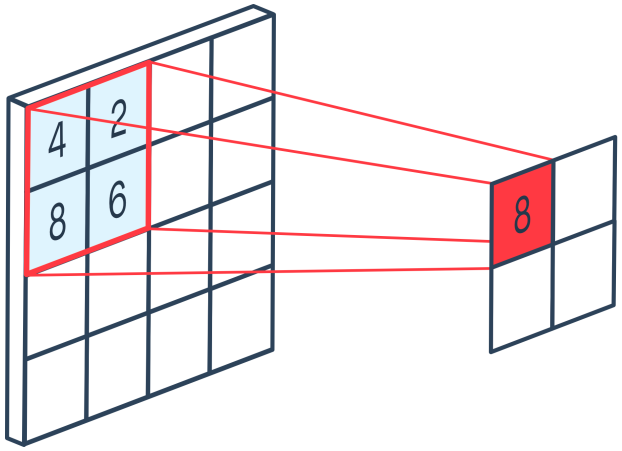
- **Translation equivariance**
  - Consider object detection (e.g., find the position of the cat in an image)
  - **Equivariance:** If we translate an image, the the object is translated similarly

Structure in Images

- Use layers that capture structure



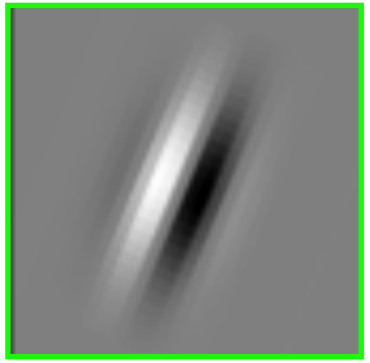
Convolution layers
(Capture equivariance)

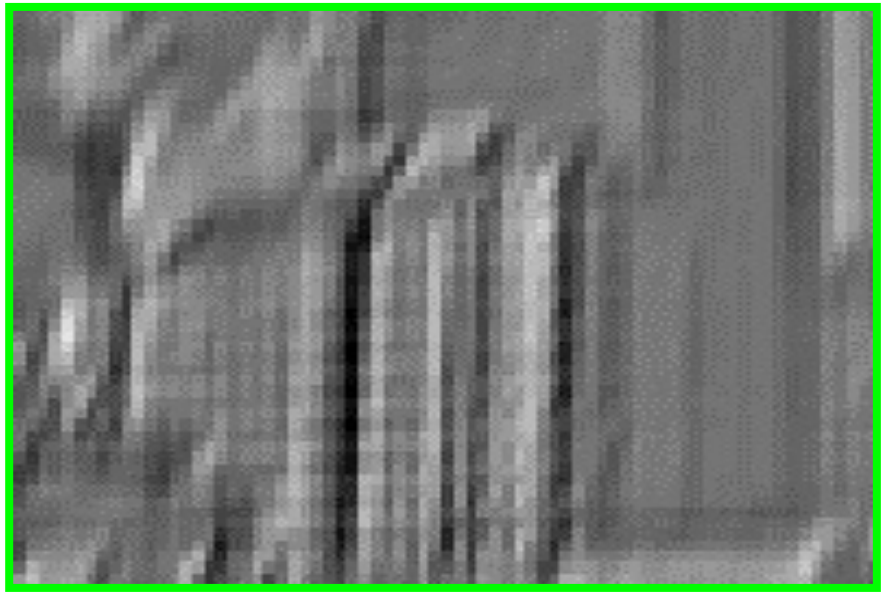


Pooling layers
(Capture invariance)

https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d
https://peltarion.com/static/2d_max_pooling_pa1.png

Convolution Filters





$$\text{output}[i,j] = \sum_{\tau=0}^{k-1}\sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

2D Convolution Filters

- **Given:**
  - A 2D input x
  - A 2D $h \times w$ kernel k

- The 2D convolution is:

$$y[s,t] = \sum_{\tau=0}^{h-1} \sum_{\gamma=0}^{w-1} k[\tau,\gamma] \cdot x[s+\tau, t+\gamma]$$

2D Convolution Filters

-1	-1	-1
2	2	2
-1	-1	-1

Horizontal lines

-1	2	-1
-1	2	-1
-1	2	-1

Vertical lines

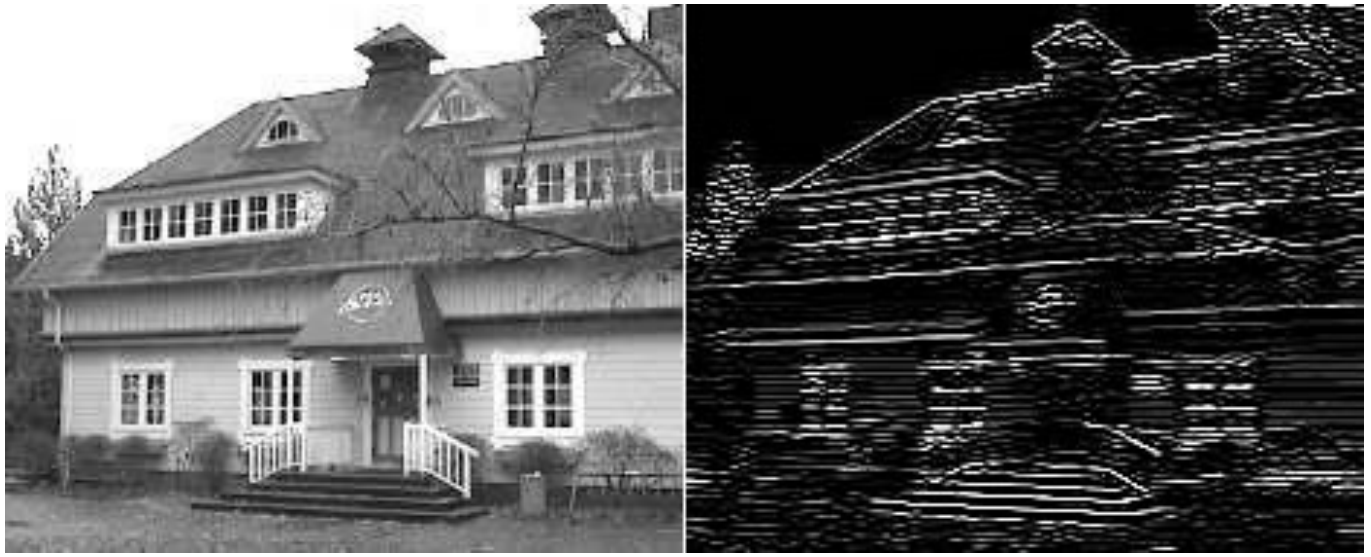
-1	-1	2
-1	2	-1
2	-1	-1

45 degree lines

2	-1	-1
-1	2	-1
-1	-1	2

135 degree lines

Example Edge Detection Kernels



Result of Convolution with Horizontal Kernel

https://aishack.in/tutorials/image-convolution-examples/

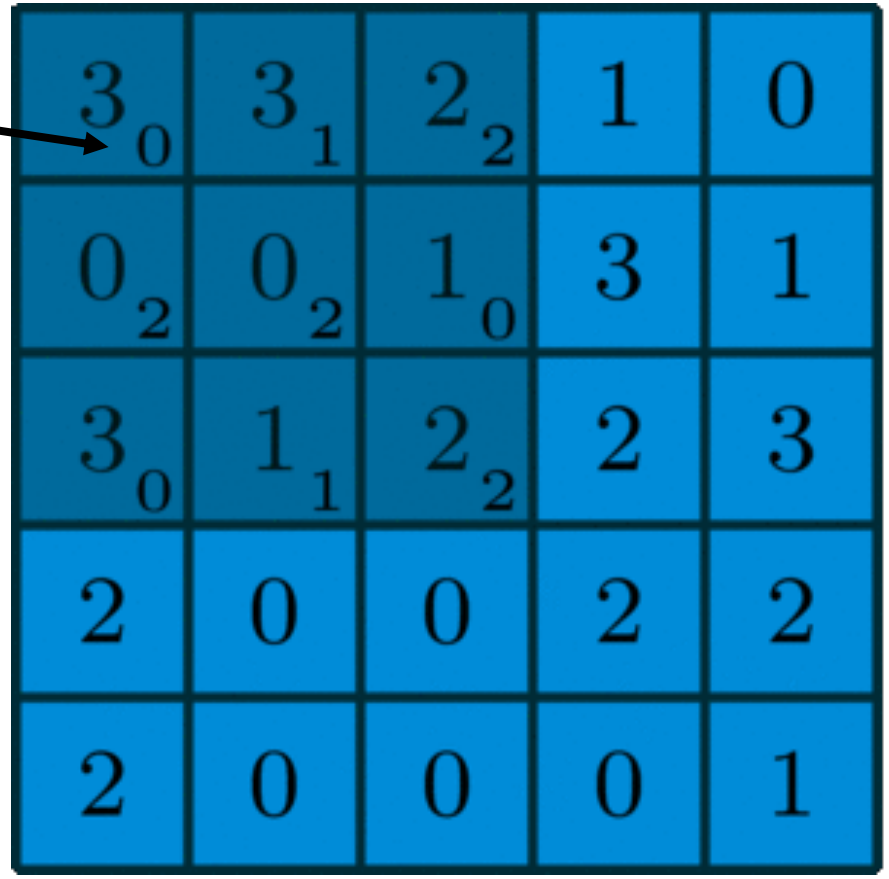
2D Convolution Filters

- Historically (until late 1980s), kernel parameters were handcrafted
  - E.g., “edge detectors”

- In convolutional neural networks, they are learned
  - Essentially a linear layer with fewer “connections”
  - Backpropagate as usual!

Convolution Layers

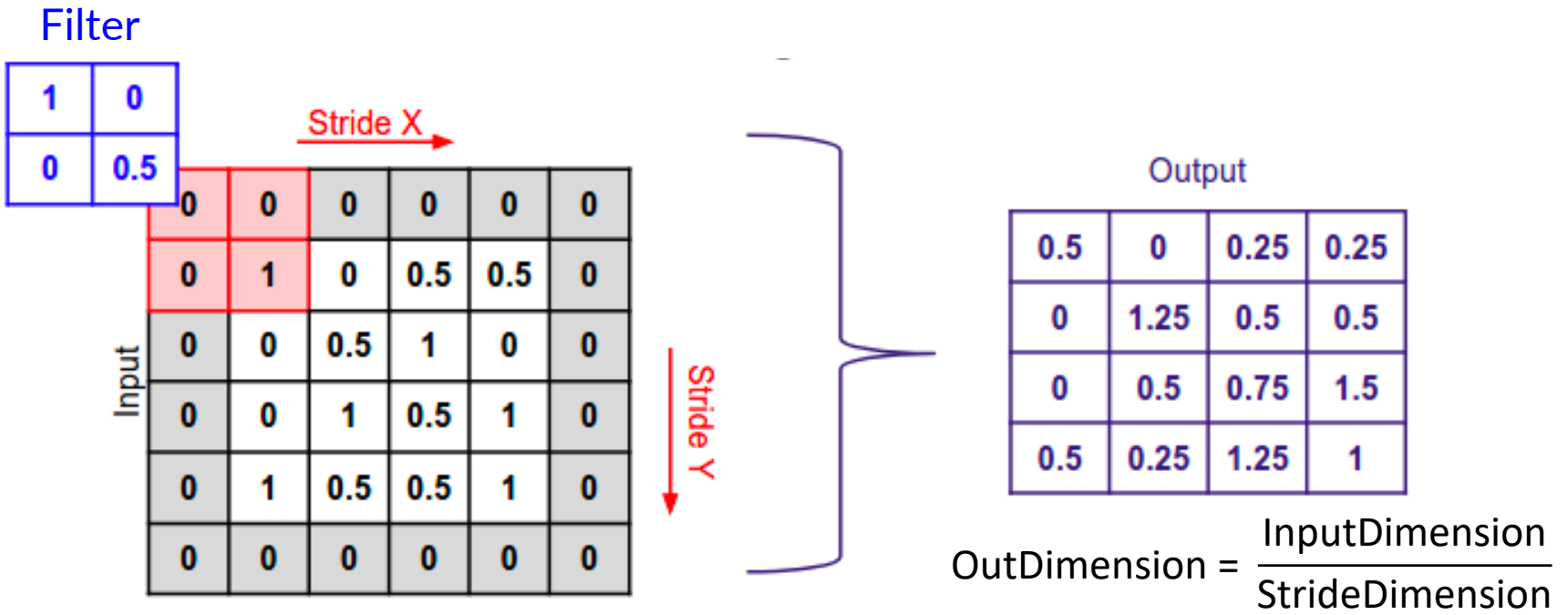
Learnable parameters



12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Convolution Layer Parameters

- **Stride:** How many pixels to skip (if any)
  - **Default:** Stride of 1 (no skipping)

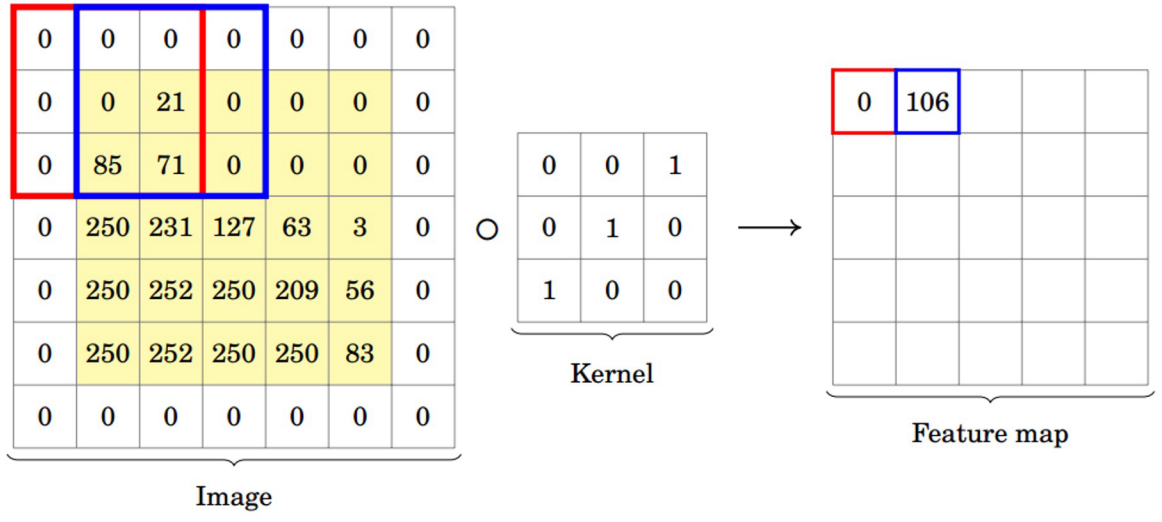


$$\text{OutDimension} = \frac{\text{InputDimension}}{\text{StrideDimension}}$$

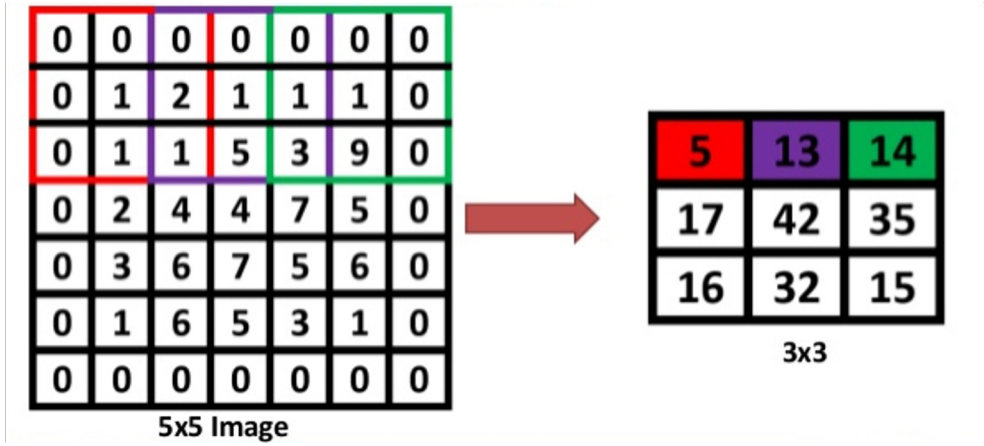
https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolution Layer Parameters

- **Padding:** Add zeros to edges of image to capture ends
  - **Default:** No padding



stride = 1, zero-padding = 1



stride = 2, zero-padding = 1

https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolution Layer Parameters

- **Summary:** Hyperparameters
  - Kernel size
  - Stride
  - Amount of zero-padding
  - Output channels

- Together, these determine the relationship between the input tensor shape and the output tensor shape

- Typically, also use a single bias term for each convolution filter

Convolution Layers

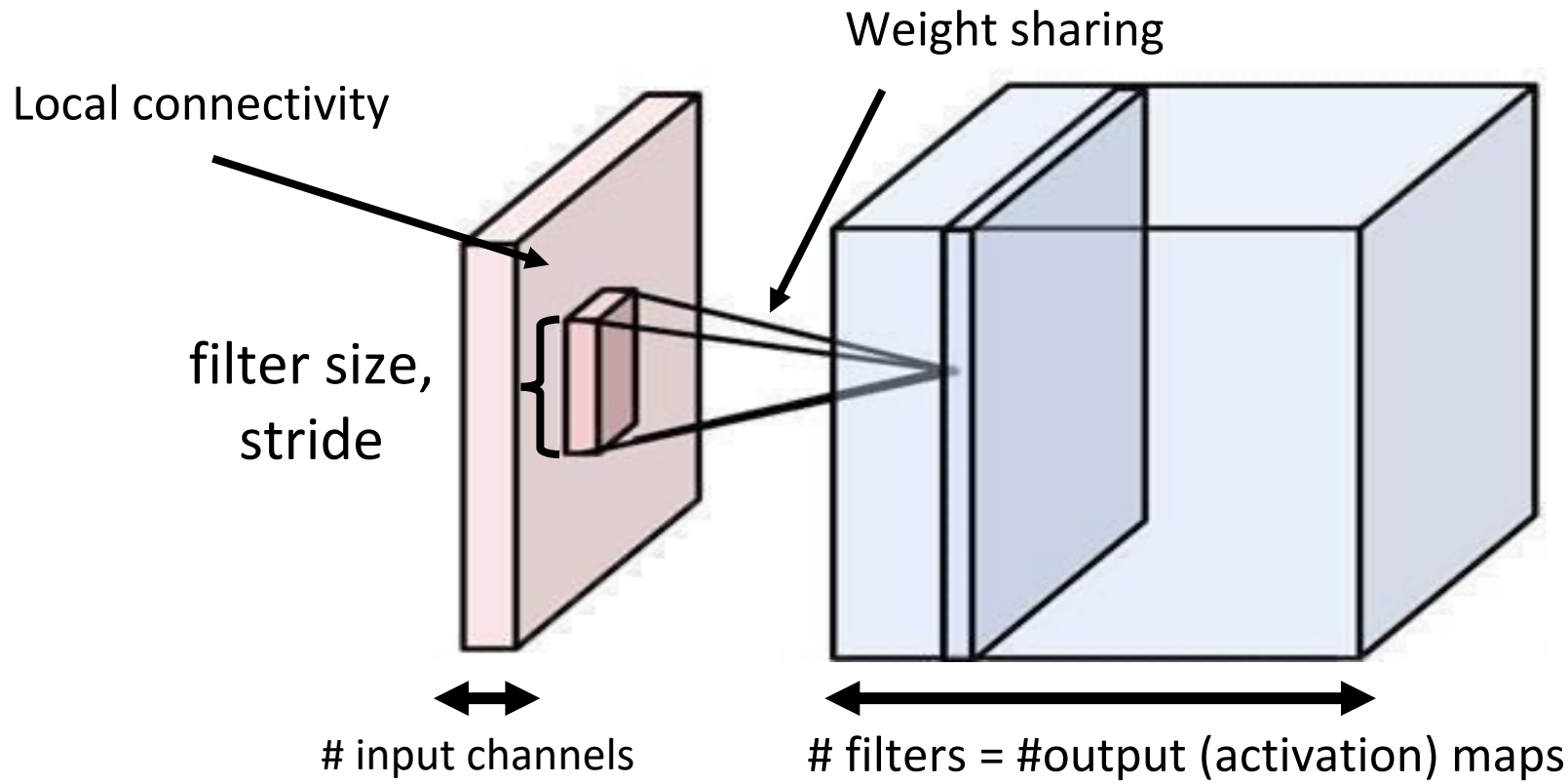
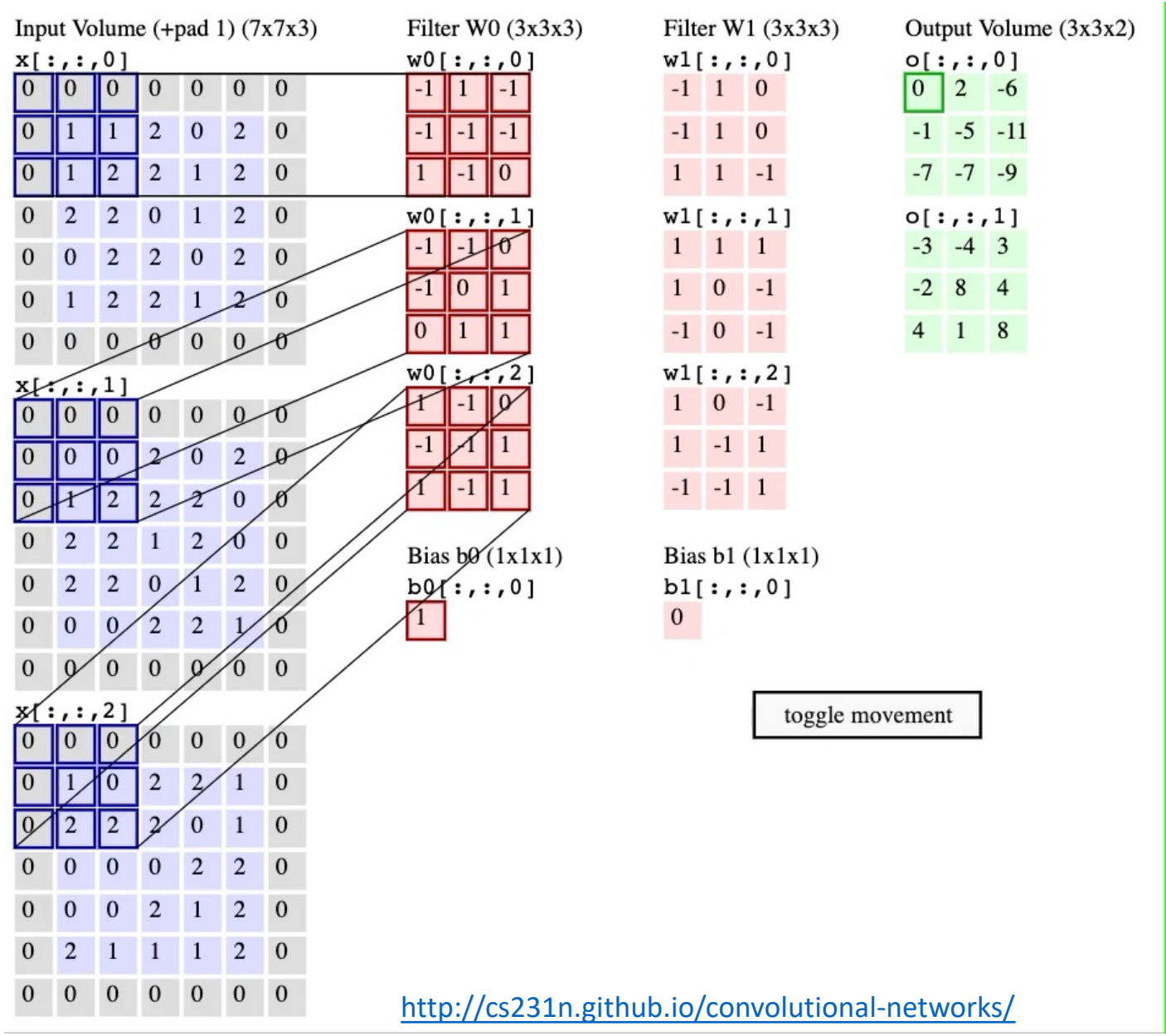


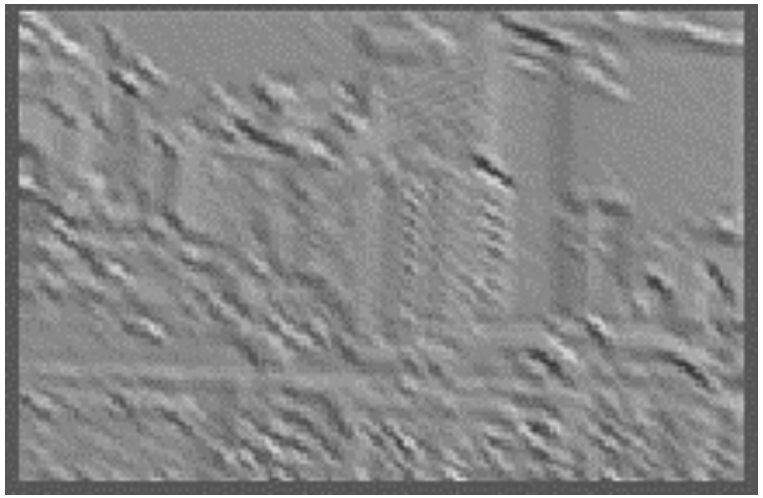
Image credit: A. Karpathy Slide credit: Jia-Bin Huang

Example

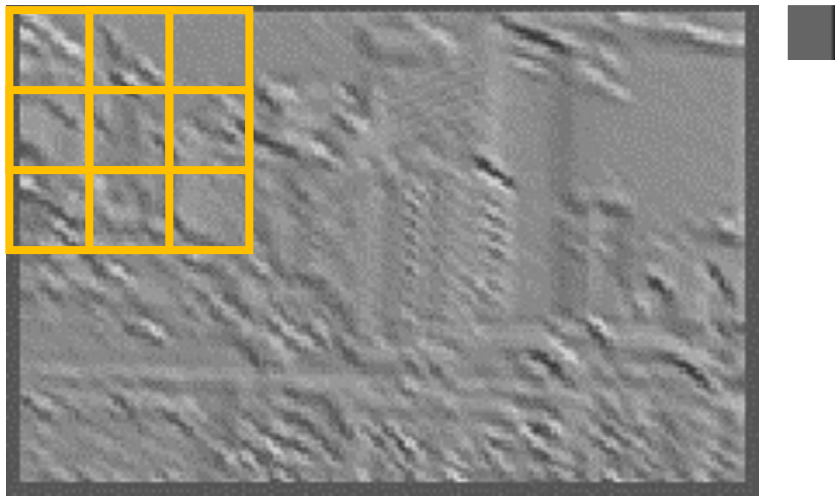
- Kernel size 3, stride 2, padding 1
- 3 input channels
  - Hence kernel size 3×3×3
- 2 output channels
  - Hence 2 kernels
- Total # of parameters:
  - $(3\times3\times3+1)\times2 = 56$



Pooling Layers

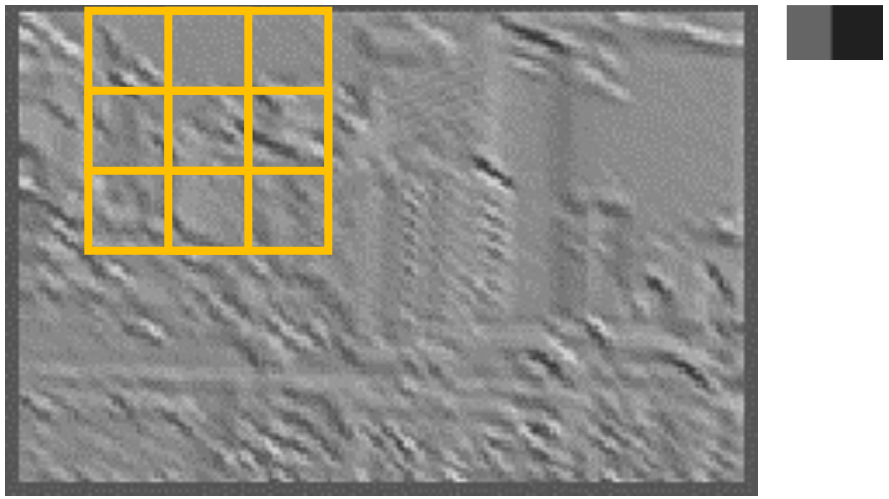


Pooling Layers



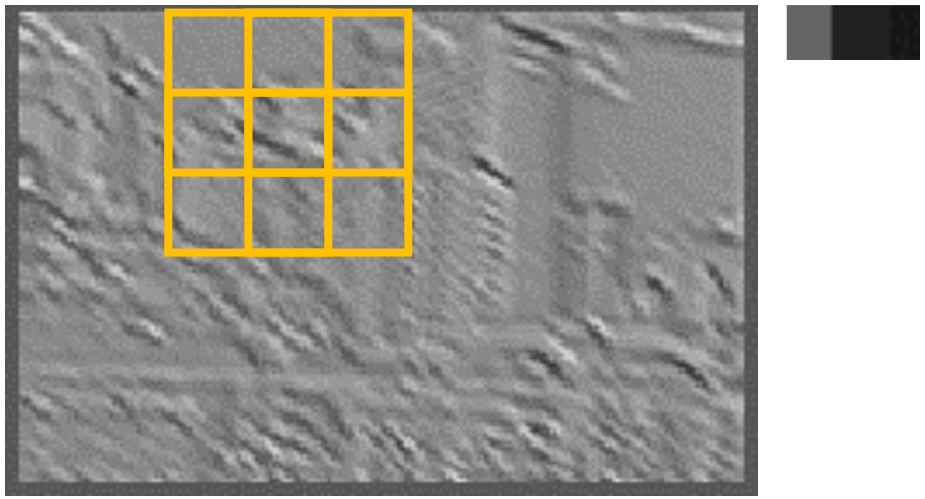
$$\text{output}[0,0] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[0 + \tau, 0 + \gamma]$$

Pooling Layers



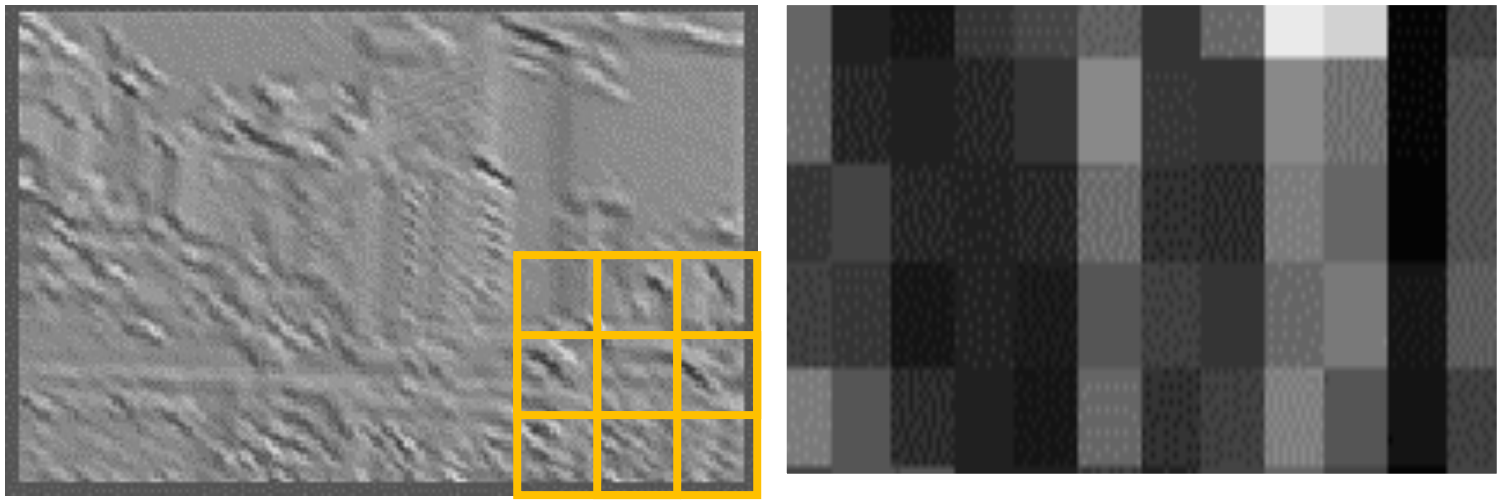
$$\text{output}[0,1] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[0 + \tau, 1 + \gamma]$$

Pooling Layers



$$\text{output}[0,2] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[0 + \tau, 2 + \gamma]$$

Pooling Layers



$$\text{output}[i,j] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[i+\tau, j+\gamma]$$

Pooling Layers

- **Summary:** Hyperparameters
  - Kernel size
  - Stride (usually >1)
  - Amount of zero-padding
  - Pooling function (almost always “max”)

- Together, these determine the relationship between the input tensor shape and the output tensor shape

- **Note:** Unlike convolution, pooling operates on channels separately
  - Thus, n input channels → n output channels

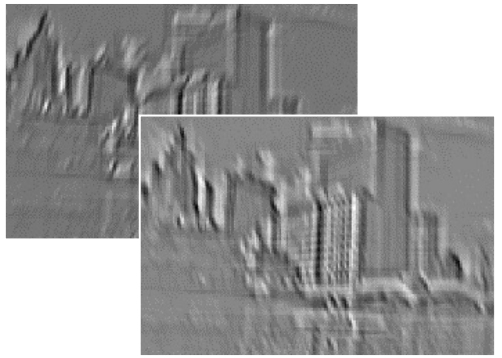
Summary: Convolution vs. Pooling

- **Convolution layers:** Translation equivariant
  - If object is translated, convolution output is translated by same amount
  - Produce “image-shaped” features that retain associations with input pixels

- **Pooling layers:** Translation invariant
  - Binning to make outputs insensitive to translation
  - Also reduces dimensionality

- Combined in modern architectures
  - Convolution to construct equivariant features
  - Pooling to enable invariance

Example Architecture: AlexNet



Fully connected
(i.e., linear) layers

output

fc, 1000

fc, 4096

fc, 4096

Input

3x3 conv, 256, pool/2

3x3 conv, 384

3x3 conv, 384

5x5 conv, 256, pool/2

11x11 conv, 96, /4, pool/2

input

Local Response Normalization

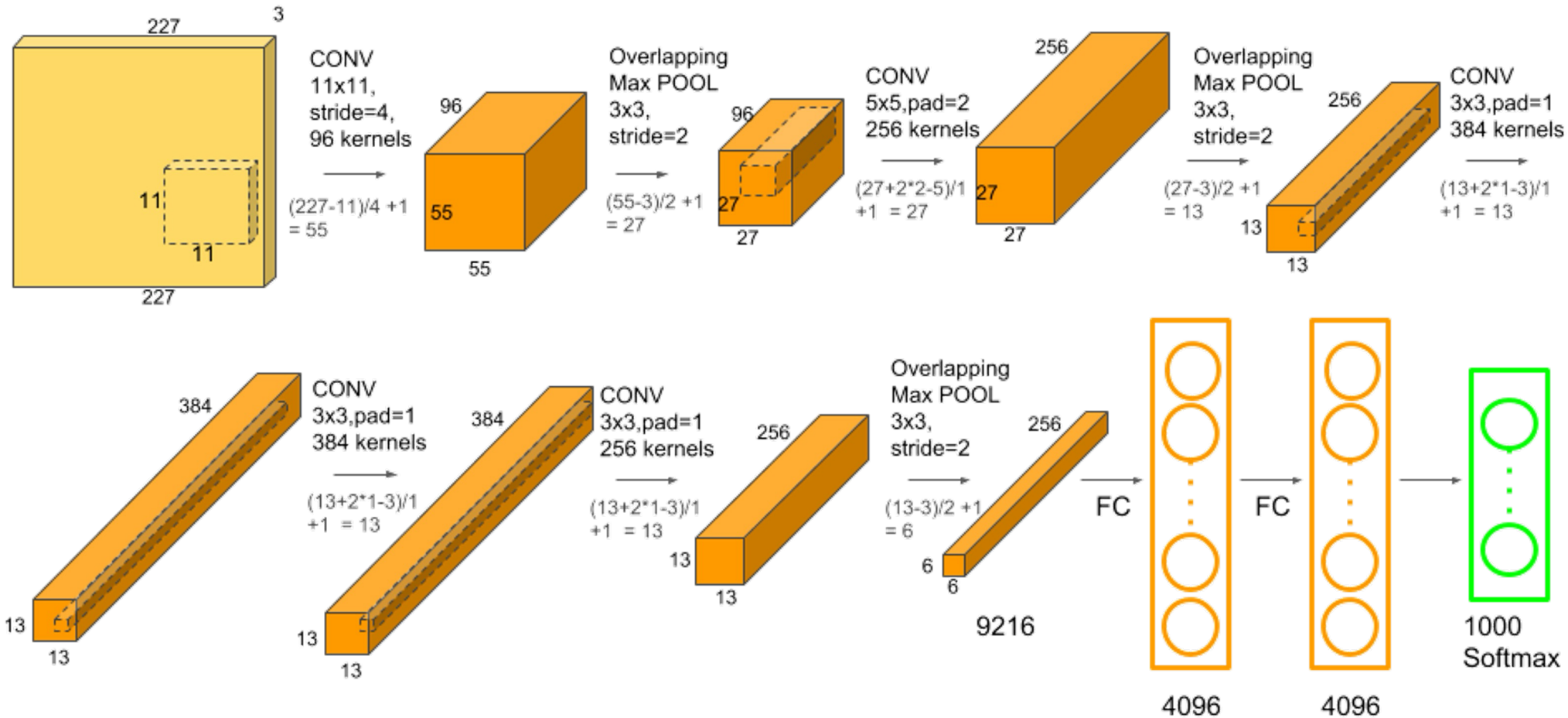
Pooling (kernel size 3, stride 2,
no padding)

ReLU Activation

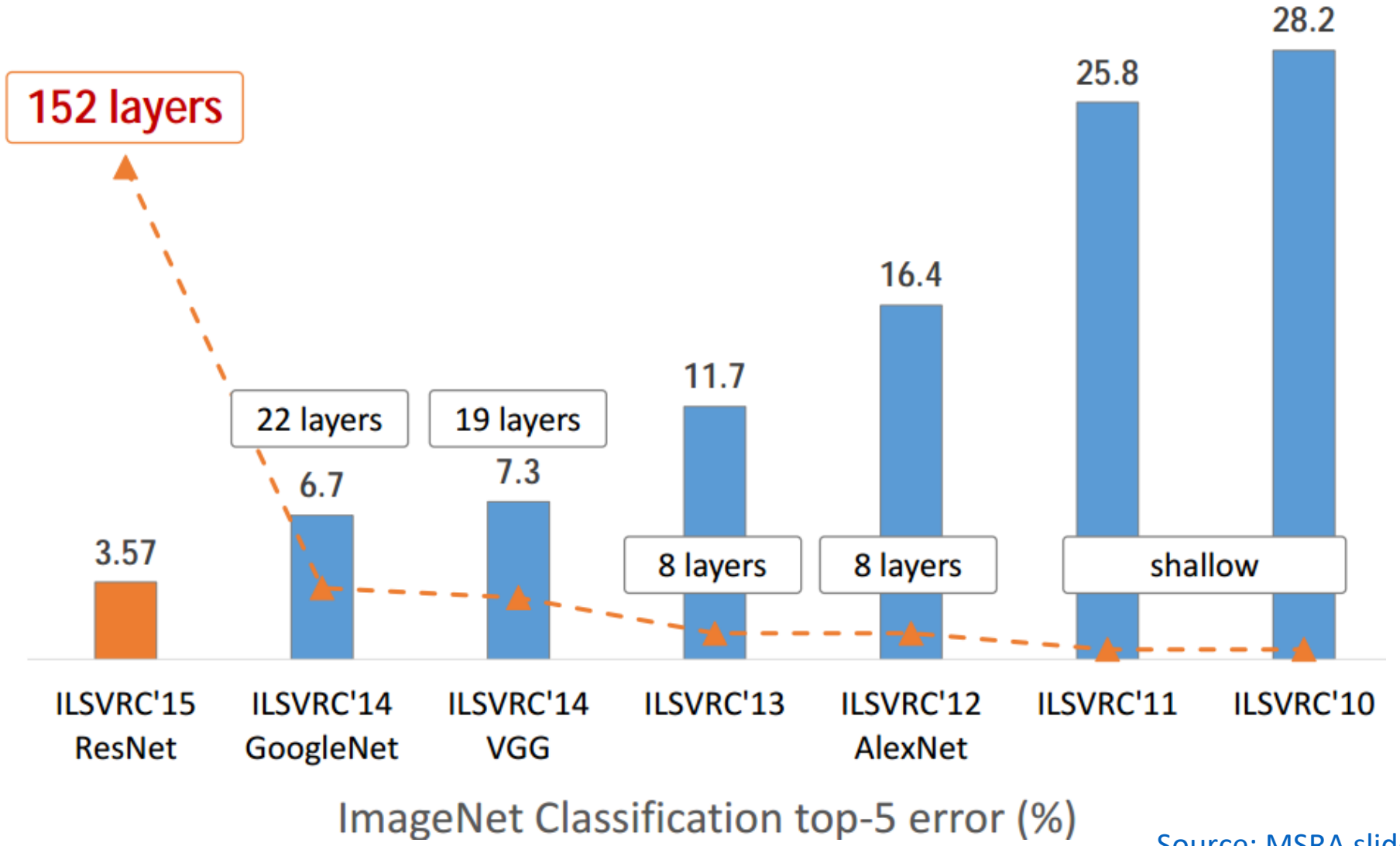
Convolution (kernel size 11, stride 4,
96 output channels, no padding)

slide credit: S. Lazebnik

Example Architecture: AlexNet



Evolution of Neural Networks



Source: MSRA slides at ILSVRC15