Regularization Techniques In Neural Networks

Fethi BOUGARES - Le Mans Université
2017-2018

Neural Net Training

1. Create network architecture
2. Initialize the weigths
3. Repeat :
    1. Sample a Batch *B* from the training data
    2. Update weights using Gradient descent

Gradient descent tells us to modify the weights W in the direction of steepest descent in *E* :

$$W_i \leftarrow W_i - \eta \frac{\partial E}{\partial W_i}$$

Neural Net functions

How to define the network architecture ?

1. The dimensionality of data set determine:

   ✓ The number of input units
   ✓ The number output units

2. The number of hidden units *M* is a free parameter

The larger network you use, the more complex the functions the network can create.

- If you use a small network
  → no enough power to fit the data (**underfit**).
- With a very large network
  → you may **overfit**

What is overfitting

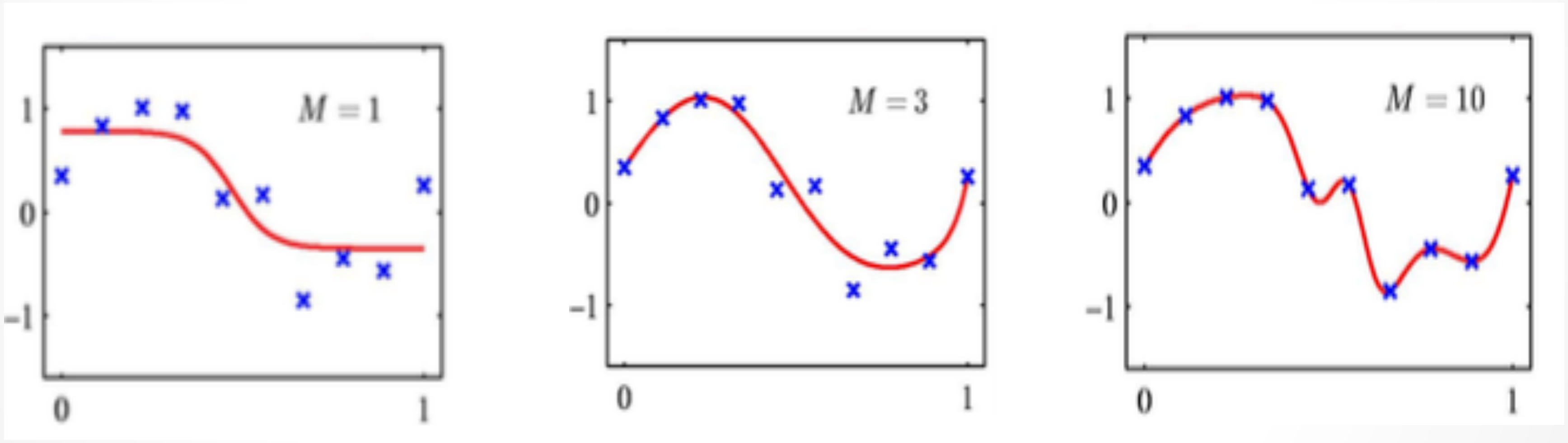
Overfitting is one of the problems that occur during training:

- ✓ The error on the training set is driven to a very small value
- ✓ When new data is presented to the network the error is large

→ The network has **memorized** the training examples
→ The network has not learned to **generalize to new situations**.

What is overfitting

Two layer network trained on *10* data points :



1 hidden units

3 hidden units

10 hidden units

Why Overfitting ?

Training data contains :

- ❖ information about the regularities in the mapping from input to output.
- ❖ but it also contains sampling error.

While learning we cannot distinguish **real regularities** from ones caused by **sampling error**.

→ The model fits both kinds of regularity.
→ If the model is very flexible it can **model the sampling error really well**.

Preventing overfitting

Approach 1: **Get more training data**

- ✓ Almost the best bet if you train on more data
- ✓ Need more computation power to train with more data

Approach 2: **Use a model that has the right capacity**

- ✓ If spurious regularities are weaker
- ✓ enough to fit the true regularities
- ✓ not enough to also fit spurious regularities

Preventing overfitting

Approach 3: **Average many different models**

Use models with different architectures.
Train the model on different subsets of the training data
→ this is called "bagging"

Approach 4: **With single network architecture**

Average the predictions made by many different weight vectors.
→ using different initializations

Limit the capacity of a neural net

The capacity can be controlled in many ways:

Architecture: Limit the number of hidden layers and the number of units per layer.

Early stopping: Start with small weights and stop the learning before it overfits.

Weight-decay: Penalize large weights using penalties or constraints on their values

Noise: Add noise to the weights or the activities.

Generally a combination of several of these methods is used.

Meta parameters and network capacity

Meta parameters :

1. the number of hidden units
2. the number of hidden units penalty
3. learning rate
4. initialization methods
5. batch size …

The wrong method: try many alternatives and see the impact of the performance on the test set.

Why ? :

→ The best settings on the test set are unlikely the best as well on a new test set

Meta parameters and network capacity

A better way is to divide the total dataset into three subsets:

Training data is used for learning the parameters of the model.

Validation data is not used for learning but is used for deciding what settings of the meta parameters work best.

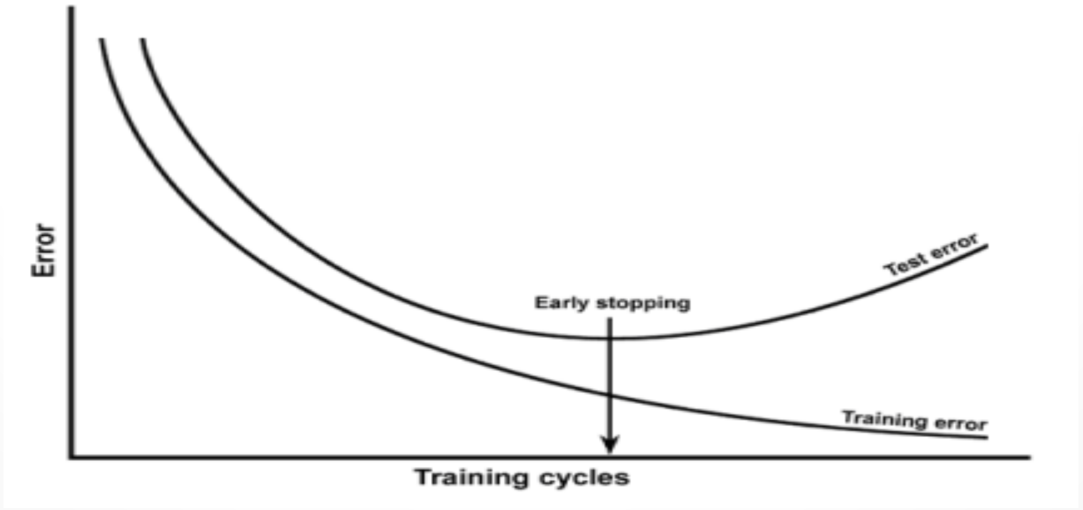
Test data is used to get a final, unbiased estimate of how well the network works (We expect this estimate to be worse than on the validation data.)

→ We can also extend this to N-fold cross-validation.

Early stopping

With a lots of data and a big model :
→ very expensive to keep re-training the model

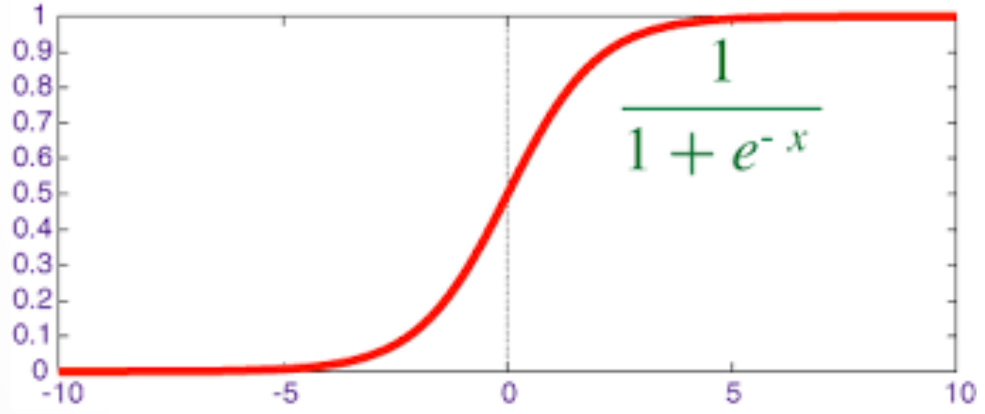
It is much cheaper to:
1. start with very small weights
2. let them grow until the performance on the validation starts getting worse.



Why Early stopping works ?

When the weights are very small :

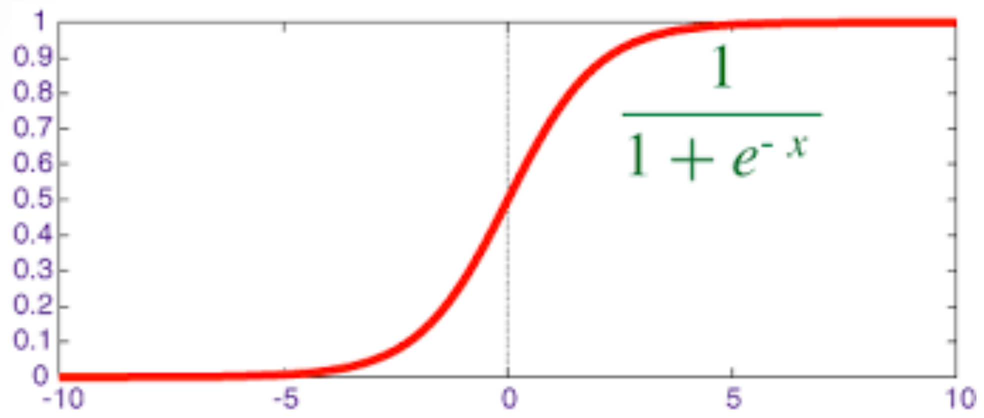
→ every hidden unit is in its linear range.
→ The net with a large layer of hidden units is linear.
→ It has no more capacity than a linear net



As the weights grow:

→ the hidden units start using their non-linear ranges
→ the capacity of the network grows
→ the network can learn more complex functions

Regularization



NN overfitting is often characterized by weight values that are very large in magnitude.

The network can learn complex function very close to the training data

The main idea of regularization is to reduce the magnitude of weights to reduce overfitting.

L2 Weight Penalty

The standard L2 weight penalty (also called weight decay):

→ adding an extra term to the **cost function**

→ penalizes the squared weights.

$$C = E + \boxed{\frac{\lambda}{2} \sum_i w_i^2}$$

This keeps the weights small unless they have big error derivatives.

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

When $\frac{\partial C}{\partial w_i} = 0$ we have $W_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}$

With L2 regularization, the update equation for the bias values doesn't change

L2 Weight Penalty

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$W_i \leftarrow W_i - \eta \frac{\partial C}{\partial W_i}$$

$$W_i \leftarrow W_i - \eta \frac{\partial E}{\partial W_i} - \eta \lambda W_i$$

$$W_i \leftarrow W_i (1 - \eta \lambda) - \eta \frac{\partial E}{\partial W_i}$$

λ is the weight cost → It determine how strong the penalty is.

($\lambda = 0$; no regularization)

when adjusting a weight value, first reduce the weight by a factor of $(1 - \eta \lambda)$ → **The weight values tend to decrease**

Effect of L2 Penalty

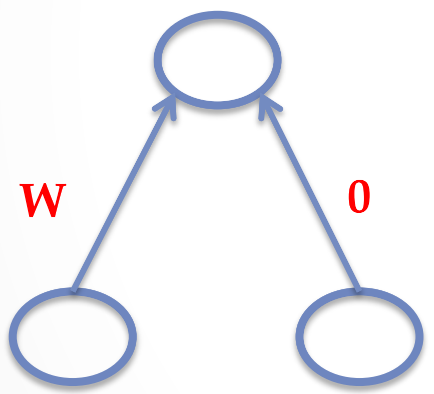
It prevents the network from using weights that it does not need.

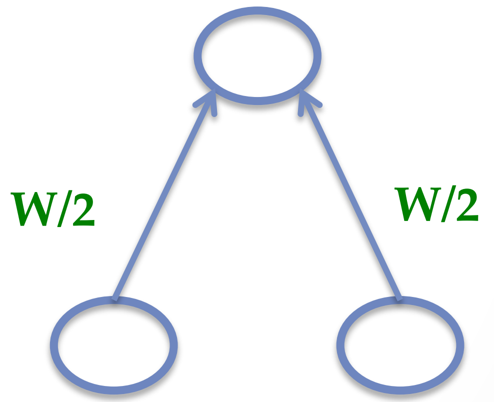
→ This can improve generalization a lot because it helps to stop the network from fitting the sampling error.

→ It makes a smoother model in which the output changes more slowly as the input changes.

Effect of L2 Penalty

If the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.





Reg : Other penalties

Sometimes it works better to penalize the absolute values of the weights.

L1 regularization :

$$\mathbf{C = E +} \boxed{\lambda \ \sum_i |\mathbf{w}_i|}$$

→ This can make many weights exactly equal to zero which helps interpretation a lot.

Reg : L1 vs L2 penalties

How the L1 and L2 contribute to a change in weights ?

Consider a model with (W_1 … W_m) weights :

L1 regularization : penalize the loss by $\sum_i |w_i|$

L2 regularization : penalize the loss by $1/2 \sum_i w_i^2$

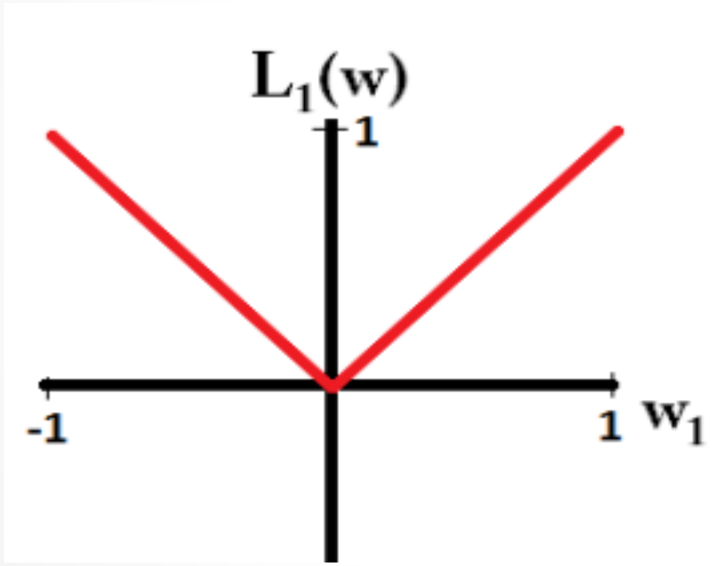
With gradient descent, we will iteratively make the weights change in the opposite direction of the gradient with a step size η :

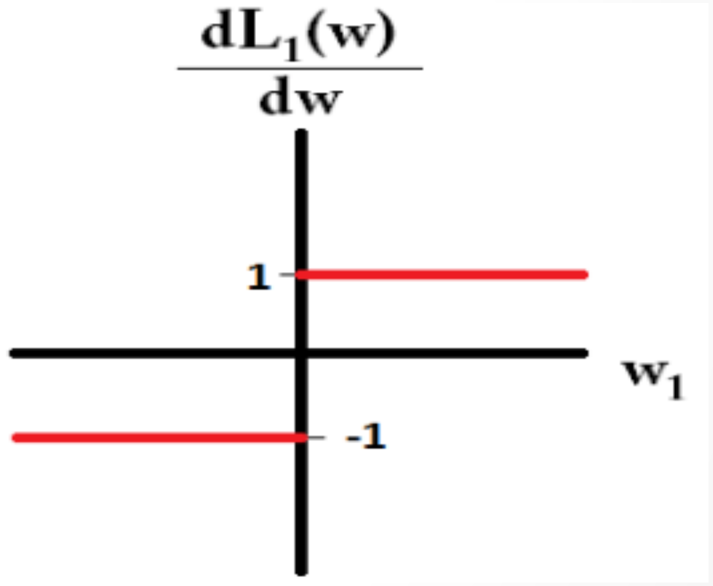
$$\frac{\partial L1}{\partial W} = \text{sign}(w) \ \text{avec} \ \text{sign}(w) = (\ w_1/|w_1|, \ldots \ w_m/|w_m| \ )$$

$$\frac{\partial L2}{\partial W} = w$$

Reg : L1 vs L2 penalties

If we plot the L1 loss function and it's derivatives of just a single parameter W :



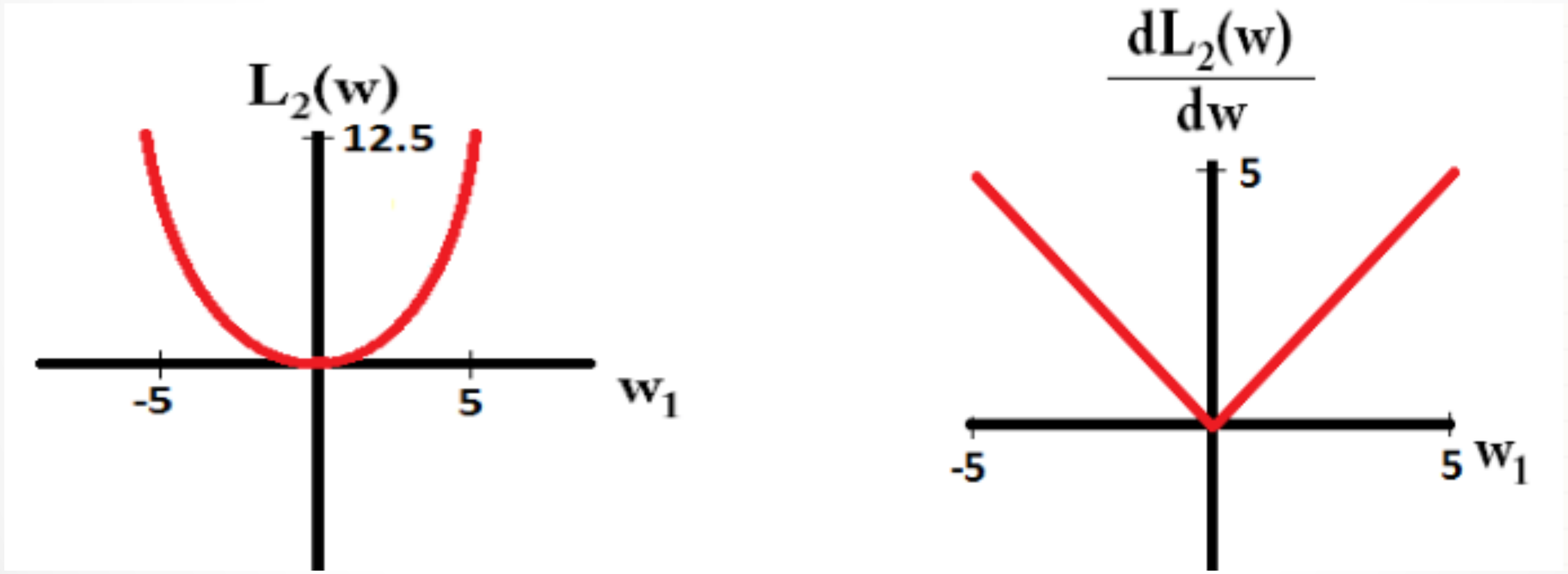


For *L*1, the gradient is either 1 or -1, except for when w_1=0

→ L1-regularization will move any weight towards 0 with the same step size

Reg : L1 vs L2 penalties

If we plot the L2 loss function and it's derivatives of just a single parameter W :



L2-regularization will also move any weight towards 0, but it will take smaller and smaller steps as a weight approaches 0.

Weight Penalties vs Weight Constraints

We usually penalize the squared value of each weight separately.

→ Instead, we can put a constraint on the maximum squared length of the incoming weight vector of each hidden or output unit.

→ If an update violates this constraint, we scale down the vector of incoming weights to the allowed length. (we divide all weights by the same value)

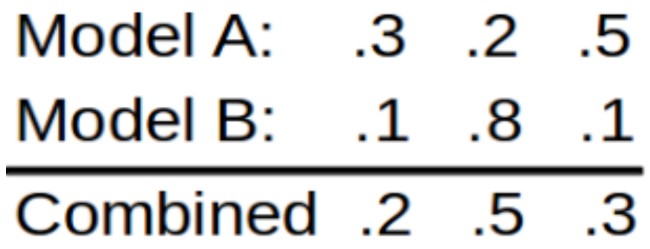
Regularization: Dropout

In order to improve the performance :

1. Train many different models
2. Use the average the decision as final output

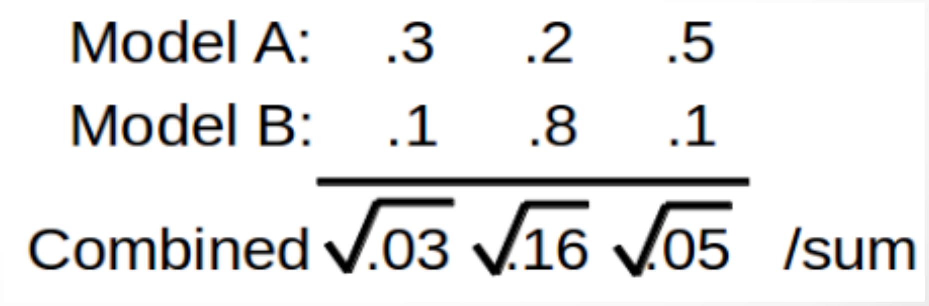
Two classical ways to average models:

1. MIXTURE: We combine models by averaging their output probs



Model A:	.3	.2	.5
Model B:	.1	.8	.1
Combined	.2	.5	.3

2. PRODUCT: We combine models by taking the geometric means of their output probabilities.



Model A:	.3	.2	.5	
Model B:	.1	.8	.1	
Combined	$\sqrt{.03}$	$\sqrt{.16}$	$\sqrt{.05}$	/sum

Regularization: Dropout

Dropout is an efficient way to combine neural nets models, **without training many different models**!

Consider a neural net with one hidden layer.

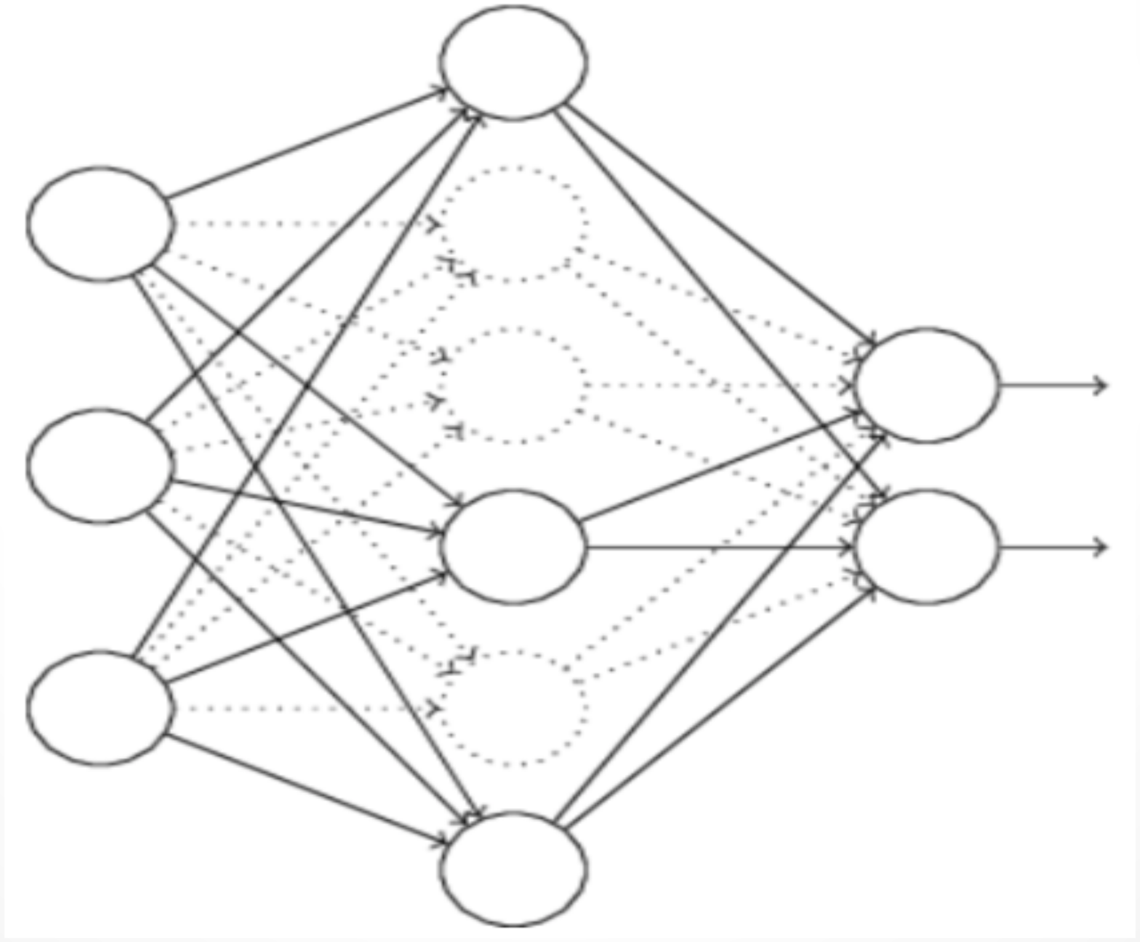
Procedure:

1. Each time
    1. we present a training example
    2. we randomly omit each hidden unit with probability 0.5.

2. So we are randomly sampling from 2^H different architectures.
    - All architectures share weights.

Regularization: Dropout

Dropout is an efficient way to combine neural nets models, **without training many different models**!



Regularization: Dropout

We sample from **2^H** models.
→ only a few of the models ever get trained, and they only get one training example.

→ This is as extreme as bagging can get.

The sharing of the weights means that every model is very strongly regularized.

→ It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.

Regularization: Dropout

In test time:

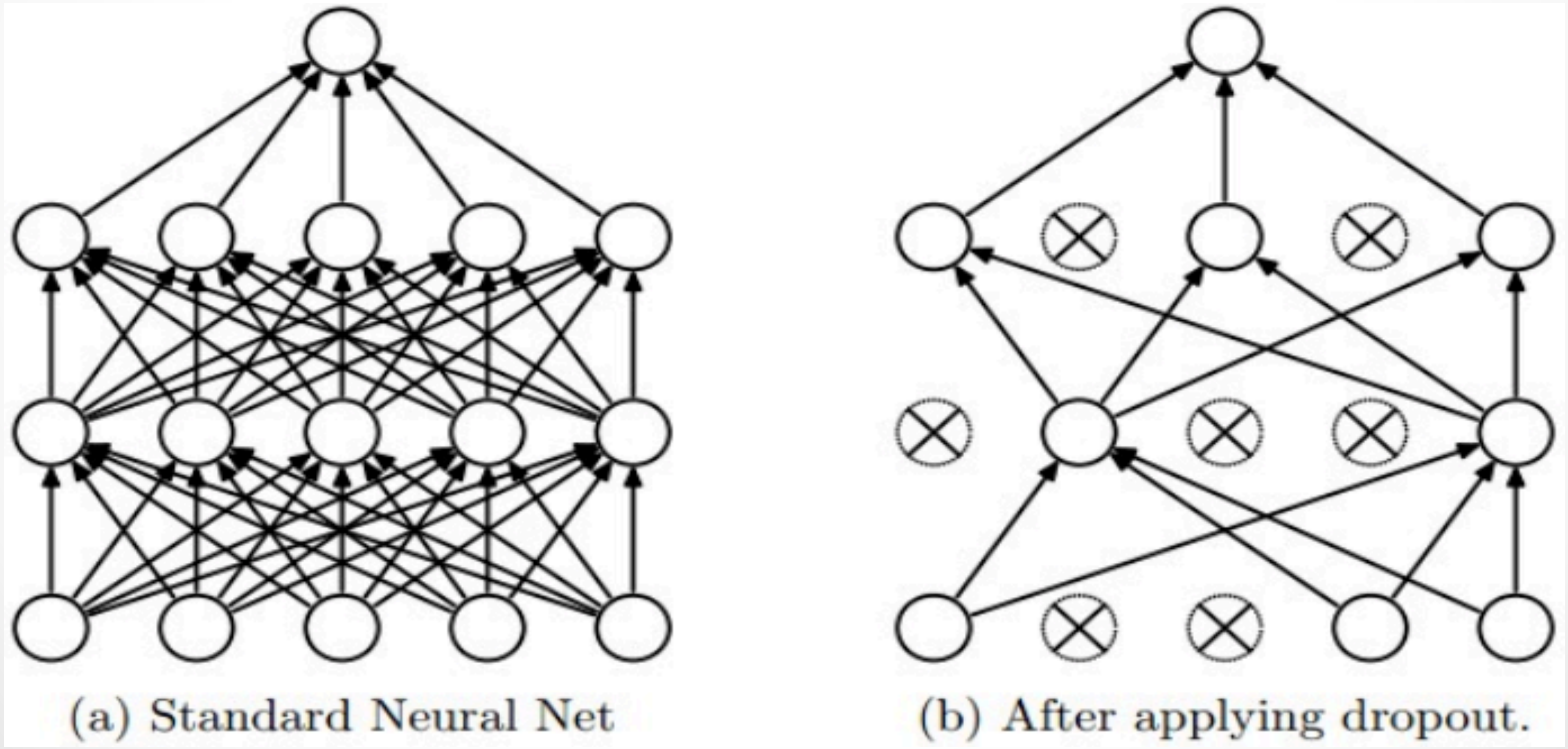
- We could sample many different architectures and take the geometric mean of their output distributions.

- It better to use all of the hidden units, but to **halve** their outgoing weights.1

  → This exactly computes the geometric mean of the predictions of all 2^H models.

Dropout for more hidden layers

Use dropout of 0.5 in every layer.


(a) Standard Neural Net
(b) After applying dropout.