# Regularization Techniques in Neural Networks

William Fedus and Panqu Wang for Garrison W. Cottrell and slides borrowed from Hinton

Gary's Unbelievable Research Unit (GURU)

Computer Science and Engineering Department

Temporal Dynamics of Learning Center

Institute for Neural Computation

UCSD

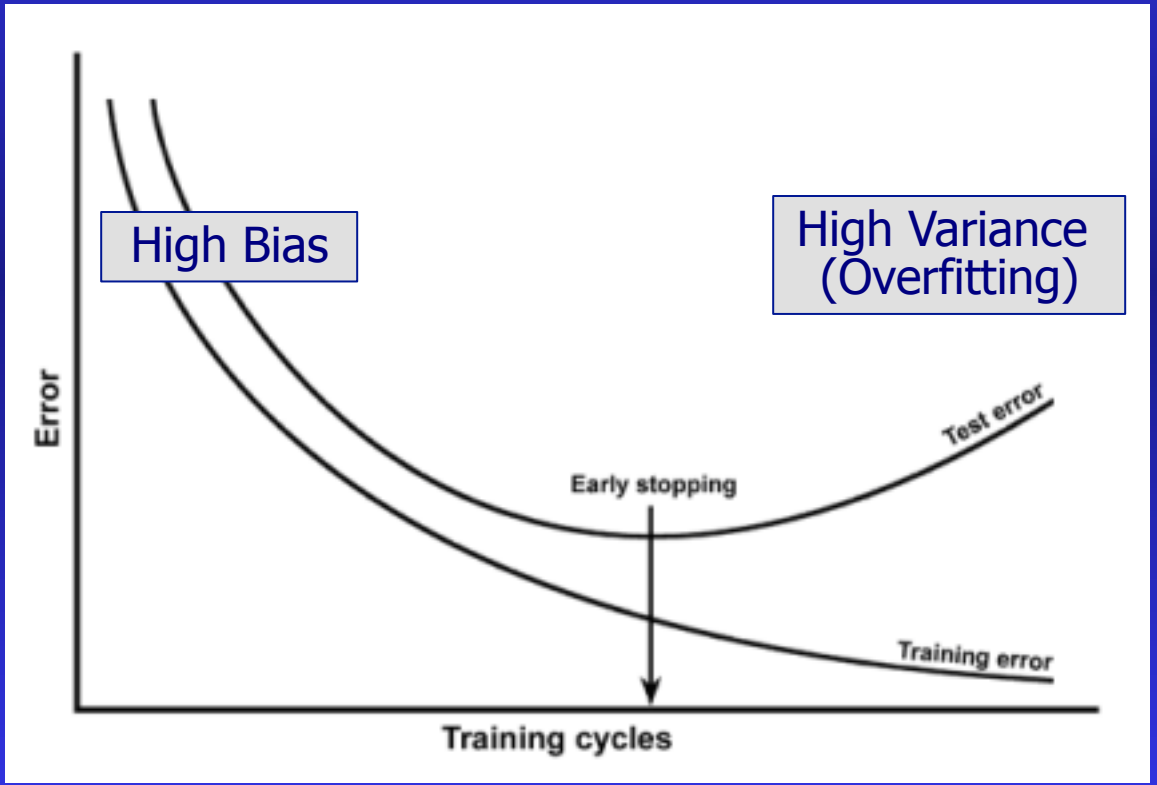


# What is Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also **contains sampling error.**
  - There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity. If the model is very flexible it can model the sampling error really well.
- This means the model will **not generalize well** to unseen data

# Diagnosing Overfitting

- Consider Training vs. Test Error as a function of training iterations



# Preventing Overfitting

- **Approach 1:** Get more data!
  - Almost always the best bet if you have enough compute power to train on more data.
- **Approach 2:** Use a model that has the right capacity:
  - enough to fit the true regularities.
  - not enough to also fit spurious regularities (if they are weaker).
- **Approach 3:** Average many different models.
  - Use models with different forms.
  - Or train the model on different subsets of the training data (this is called “bagging”).

# Limiting 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 squared values (L2 penalty) or absolute values (L1 penalty).
  - **Noise:** Add noise to the weights or the activities.
- Typically, a combination of several of these methods is used.

# Choosing Hyperparameters

- The wrong method is to try lots of alternatives and see which gives the best performance on the test set.
  - This is easy to do, but it gives a false impression of how well the method works.
  - The settings that work best on the test set are unlikely to work as well on a new test set drawn from the same distribution.
- **An extreme example:** Suppose the test set has **random** answers that do not depend on the input.
  - The best architecture will do better than chance on the test set.
  - But it cannot be expected to do better than chance on a new test set.

# Cross-Validation for Hyperparameters

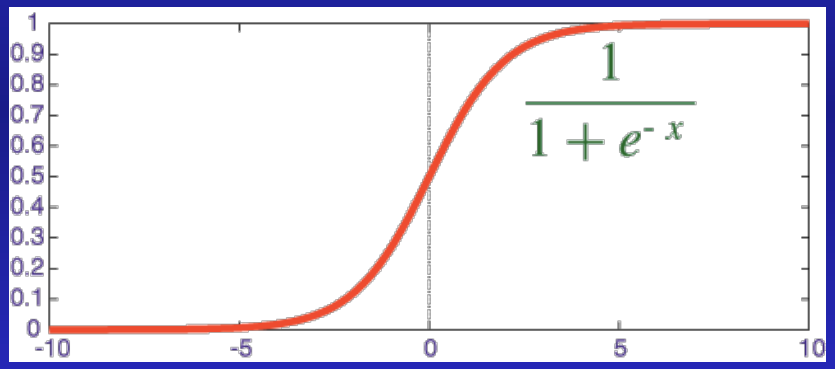
- Divide the total dataset into three subsets:
  - **Training data:** is used for learning the parameters of the model.
  - **Validation data:** is not used for learning but is used for deciding what settings of the meta parameters work best.
  - **Test data:** is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could divide the total dataset into one final test set and N other subsets and train on all but one of those subsets to get N different estimates of the validation error rate.
  - This is called N-fold cross-validation.
  - The N estimates are not independent.

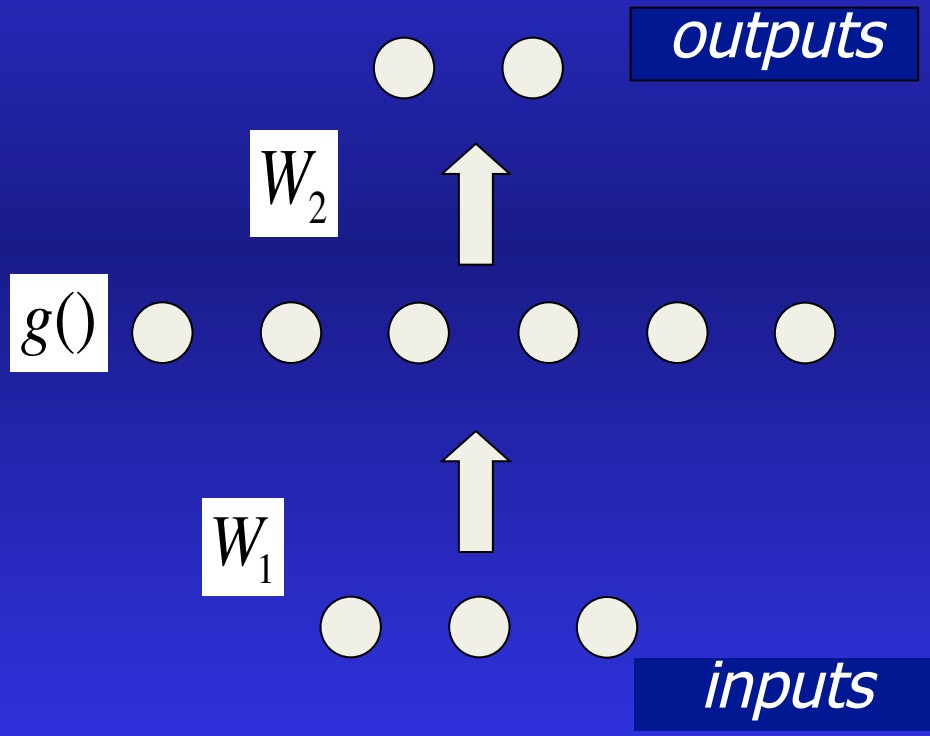
# Prevent Overfitting with Early Stopping

- If we have **lots of data and a big model**, its very expensive to keep re-training it with different sized penalties on the weights.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse.
  - But it can be hard to decide when performance is getting worse.
- The capacity of the model is limited because the weights have not had time to grow big

# Why Early Stopping Works (1)

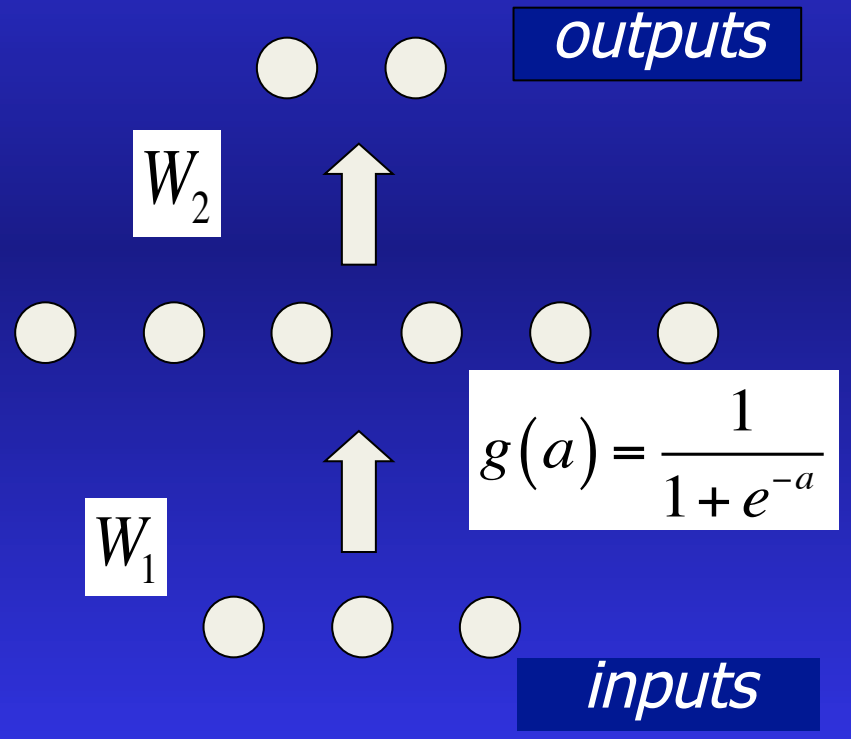
- So if we consider small weights on Sigmoidal hidden units



# Why Early Stopping Works (2)

- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



# Regularization: L2 Weight Penalty

- The standard L2 weight penalty involves adding an extra term to the cost function that **penalizes the squared weights.**
  - This keeps the weights small unless they have big error derivatives.

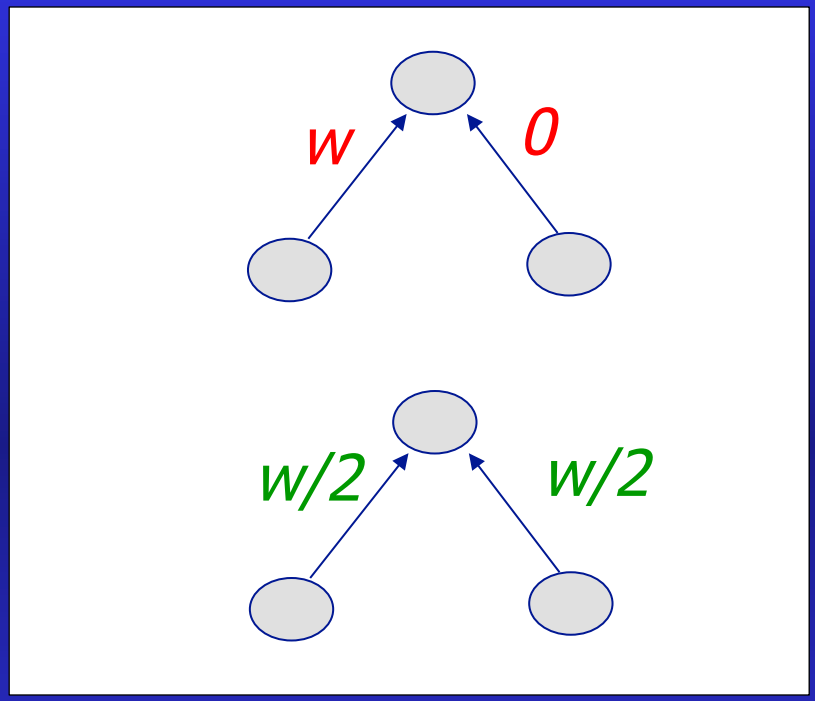
$$C = E + \frac{\lambda}{2}\sum_i w_i^2$$

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

$$when \quad \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda}\frac{\partial E}{\partial w_i}$$

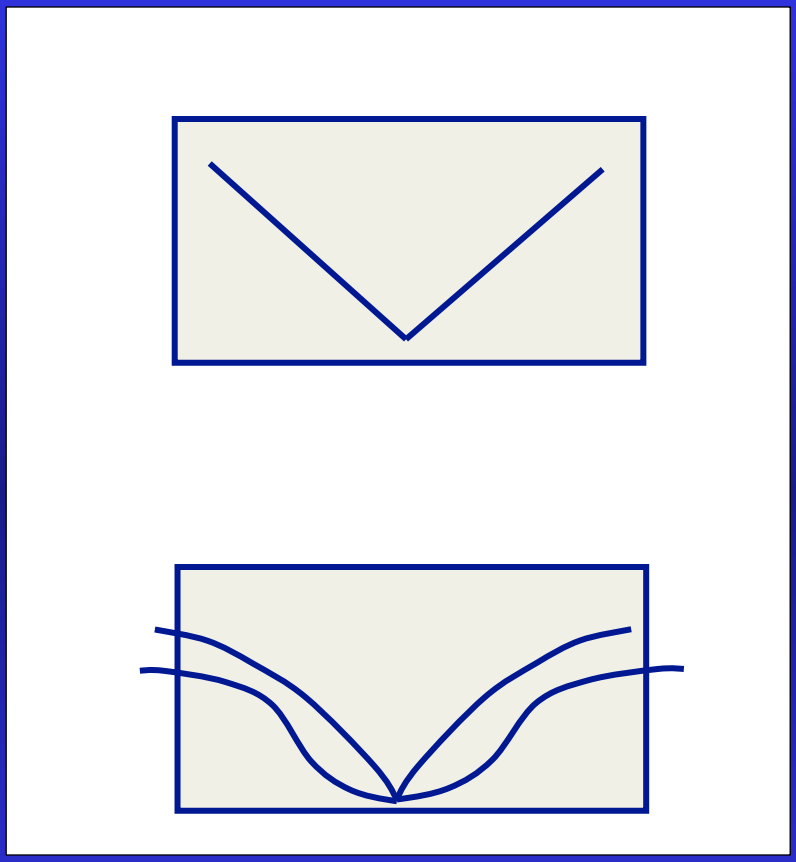
# Regularization: Effect of L2 Penalty

- It prevents the network from using weights that it does not need.
  - This can often 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.
- If the network has two very similar inputs it prefers to put **half the weight on each rather than all the weight on one.**



# Regularization: Other Penalties

- Sometimes it works better to penalize the absolute values of the weights.
  - This can make many weights exactly equal to zero which helps interpretation a lot.
- Sometimes it works better to use a weight penalty that has negligible effect on large weights.
  - This allows a few large weights.

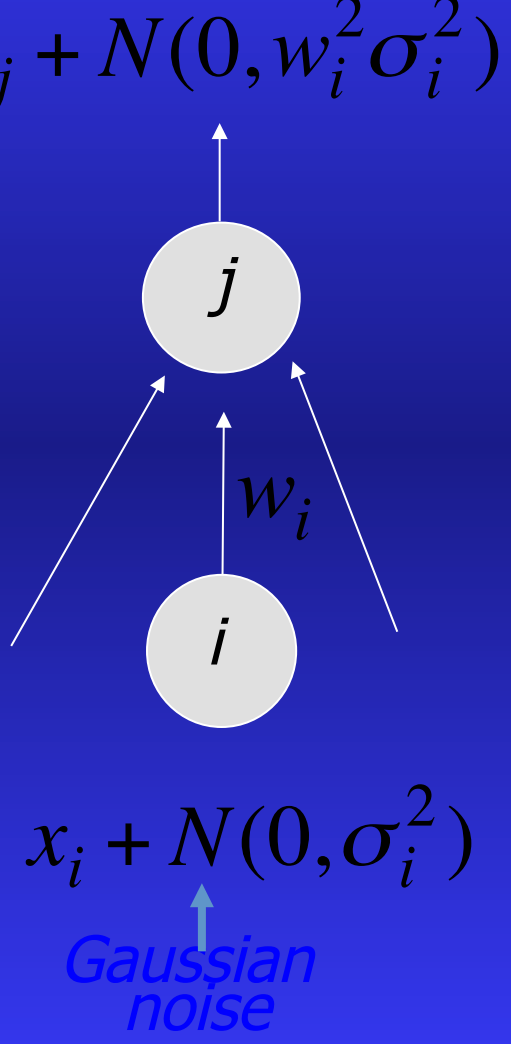


# Weight Penalties vs Weight Constraints

- 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 unit.
  - If an update violates this constraint, we scale down the vector of incoming weights to the allowed length.
- Weight constraints have **several advantages** over weight penalties.
  - Its easier to set a sensible value.
  - They prevent hidden units getting stuck near zero.
  - They prevent weights exploding.
- When a unit hits it's limit, the effective weight penalty on all of it's weights is determined by the big gradients.
  - This is more effective than a fixed penalty at pushing irrelevant weights towards zero.

# Regularization: L2 weight-decay via noisy inputs

- Suppose we add Gaussian noise to the inputs.
  - The variance of the noise is amplified by the squared weight before going into the next layer.
- In a simple net with a linear output unit directly connected to the inputs, the amplified noise gets added to the output.
- This makes an additive contribution to the squared error.
  - So minimizing the squared error tends to minimize the squared weights when the inputs are noisy.



# Derivation

$$y^{noisy} = \sum_i w_i x_i + \sum_i w_i \varepsilon_i \quad \text{where } \varepsilon_i \text{ is sampled from } N(0,\sigma_i^2)$$

$$E\left[(y^{noisy} - t)^2\right] = E\left[\left(y + \sum_i w_i \varepsilon_i - t\right)^2\right] = E\left[\left((y-t) + \sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y-t)^2 + E\left[2(y-t)\sum_i w_i \varepsilon_i\right] + E\left[\left(\sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y-t)^2 + E\left[\sum_i w_i^2 \varepsilon_i^2\right]$$

*because* $\varepsilon_i$ *is independent of* $\varepsilon_j$ *and* $\varepsilon_i$ *is independent of* $(y-t)$

$$= (y-t)^2 + \sum_i w_i^2 \sigma_i^2$$

# Regularization: Dropout

- An efficient way to **combine neural nets models**, without training many different models!
- Two classical ways to average models:

  MIXTURE: We can combine models by averaging their output probabilities.

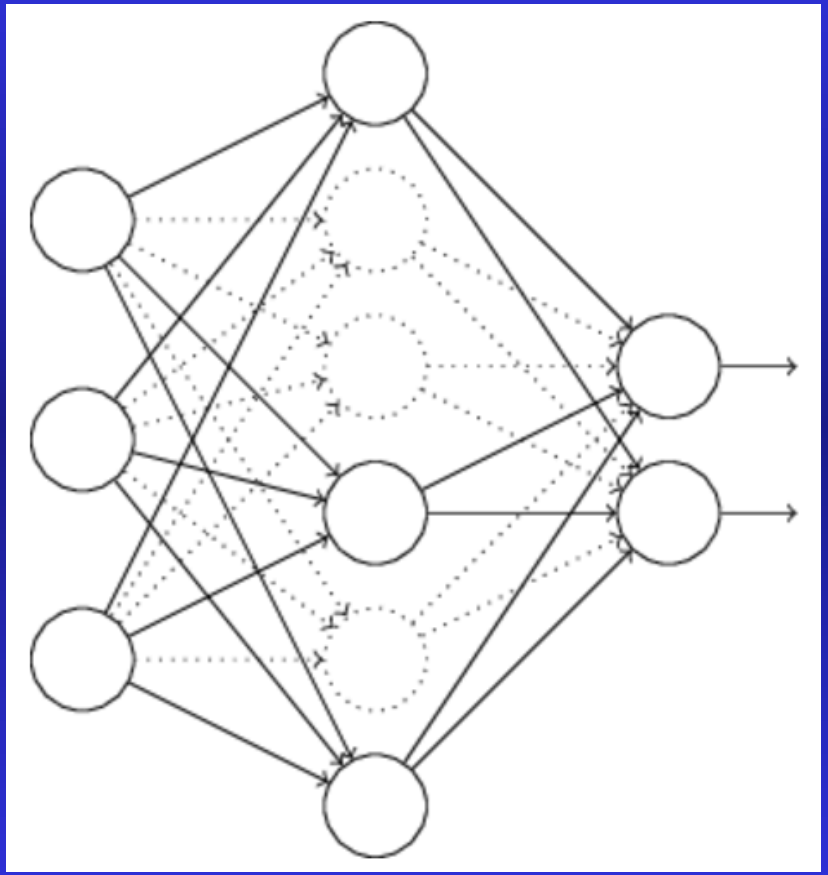
| | | | |
|---|---|---|---|
| Model A: | .3 | .2 | .5 |
| Model B: | .1 | .8 | .1 |
| Combined | .2 | .5 | .3 |

  PRODUCT: We can 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

- An efficient way to average many large neural nets.
- Consider a neural net with one hidden layer.
- **Procedure:**
    1. Each time we present a training example, 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.

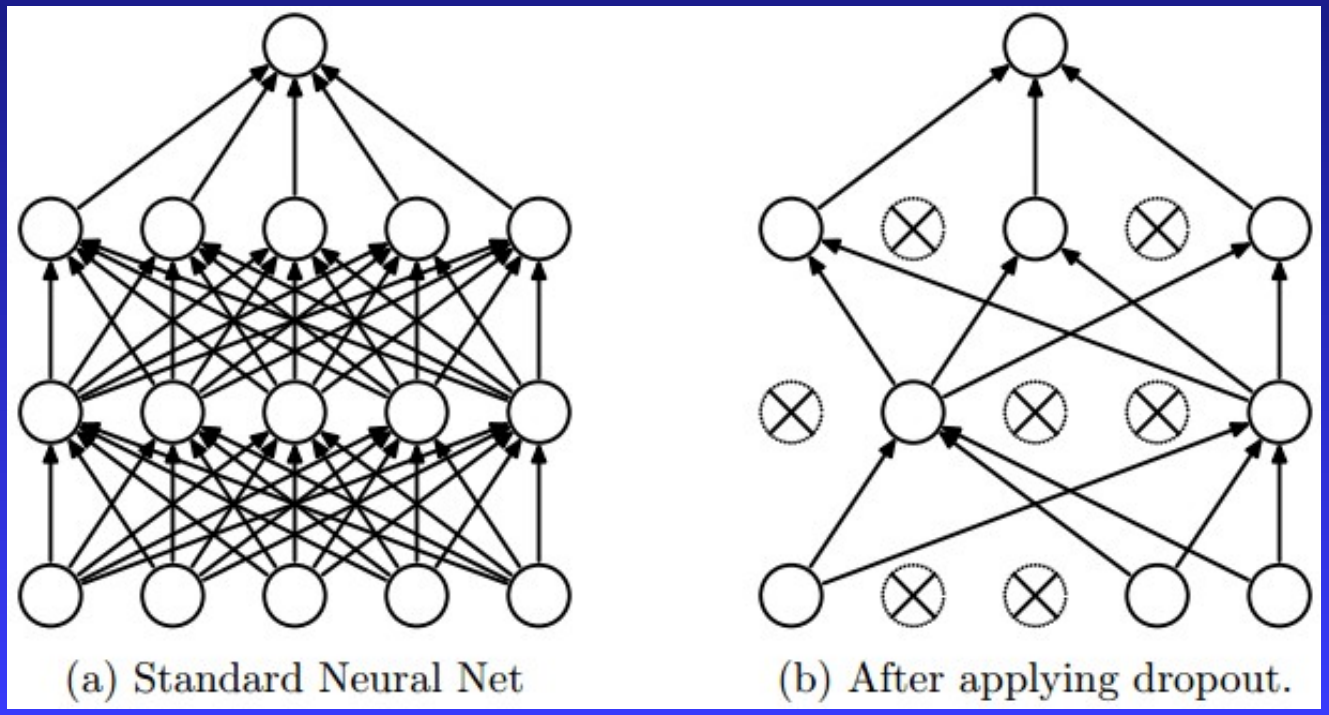


# Dropout: Form of Model Averaging

- We sample from 2^H models. So 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.
- 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.
    - 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.
- At test time, use the "mean net" that has all the outgoing weights halved.
  - This is not exactly the same as averaging all the separate dropped out models, but it's a pretty good approximation, and its fast.
- Alternatively, run the stochastic model several times on the same input.
  - This gives us an idea of the uncertainty in the answer.



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

# Dropout: how and why does it work?

- The record breaking object recognition net developed by Alex Krizhevsky uses dropout and it helps a lot.

- Almost every deep network today uses dropout.

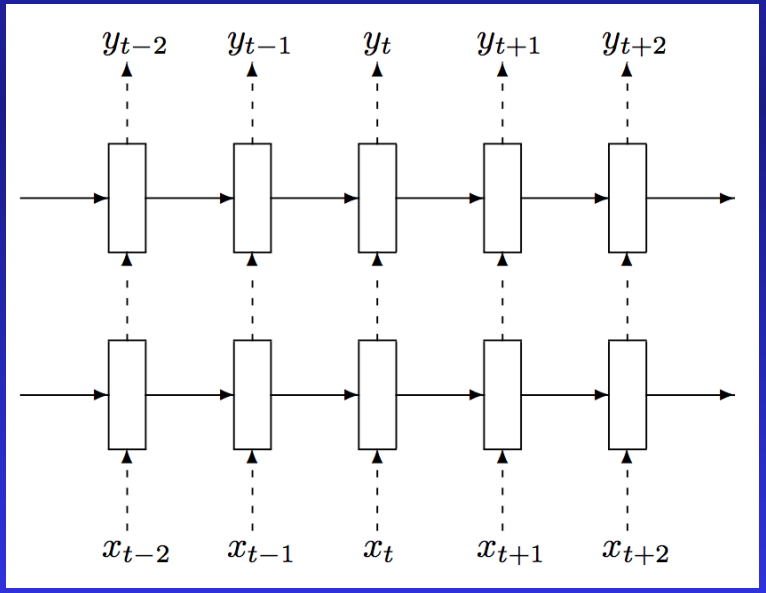
- If your deep neural net is significantly overfitting, dropout will usually reduce the number of errors by a lot.
    - Any net that uses "early stopping" can do better by using dropout (at the cost of taking quite a lot longer to train).

- If your deep neural net is not overfitting you should be using a bigger one!
- Hidden units may co-adapt when it knows what other hidden units are present
- Dropout forces the units to do something individually **useful** and **different** than what the other hidden units are doing

# Dropout: Recurrent Networks

- Deep recurrent neural networks also have high excess capacity and may overfit.
- Until recently, dropout was believed to corrupt the information carried by recurrent networks and this makes it difficult for LSTMs to learn to store information for extended periods

- Wojciech et al. 2014 determined that by applying Dropout only to non-recurrent connections, performance could be enhanced in deep recurrent neural networks
- Architecture, as shown to the right, reduced overfitting on a variety of tasks



# Regularization: Dropconnect

- Generalization of Hinton's Droput procedure, Dropconnect instead drops connections (weights), not entire activations (nodes)
- Wan et al, ICML 2014 showed that Dropconnect could lead to faster convergence than use of Dropout and that it often outperforms Dropout