Artificial Neural Networks

Feedforward Networks Part II

Outline

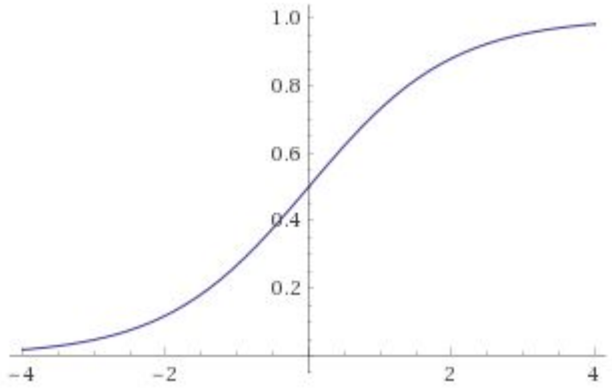
- Activation functions
  - Sigmoid, tanh
  - ReLu and variations
- Output functions & Loss functions
- Regularization
  - L1 and L2 Loss
  - Early Stopping
  - Dropout
  - Dataset augmentation

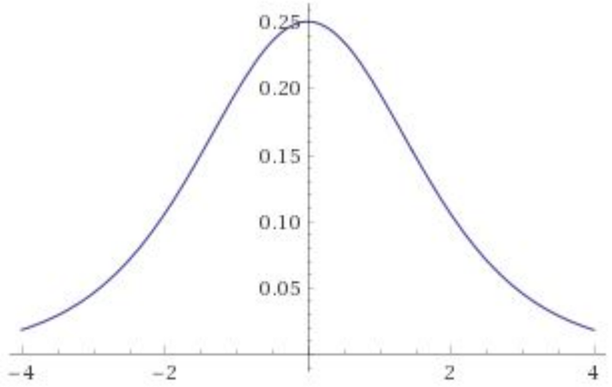
Activation Functions

Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$$



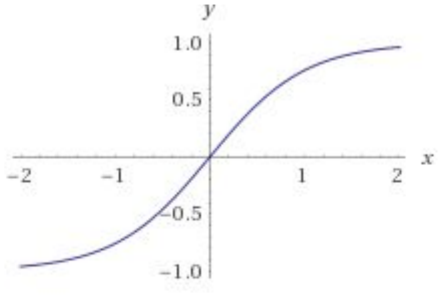


- This function is familiar to us
- Logistic activation, easy to reason about

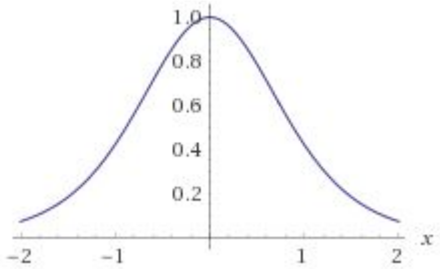
- This function saturates and its gradients are small
- This function isn't 0-centered
- Useful for understanding ANNs, but not used much in practice anymore

Tanh

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

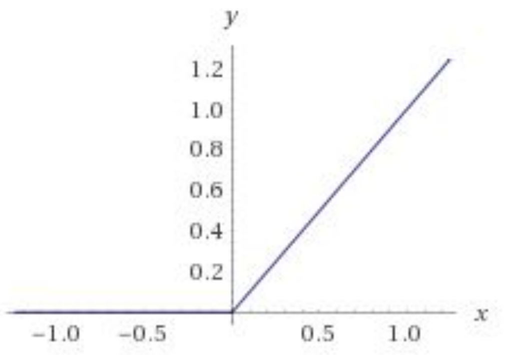


- This function is 0-centered, behaves a little better
- Derivative is little stronger

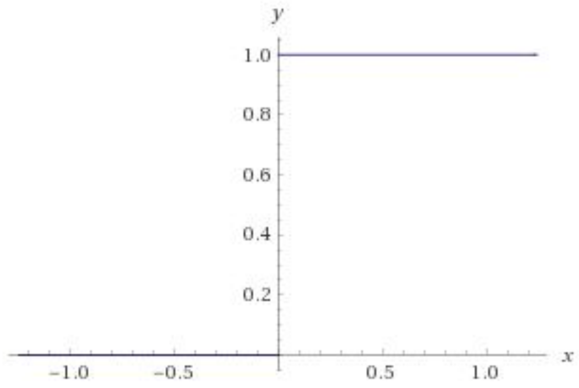
- If you're looking for a smooth S-shaped curve, try this one instead of sigmoid

ReLU

$$\mathrm{ReLu}(x) = \max(0, x)$$



$$\frac{d}{dx}(\max(0, x)) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$$

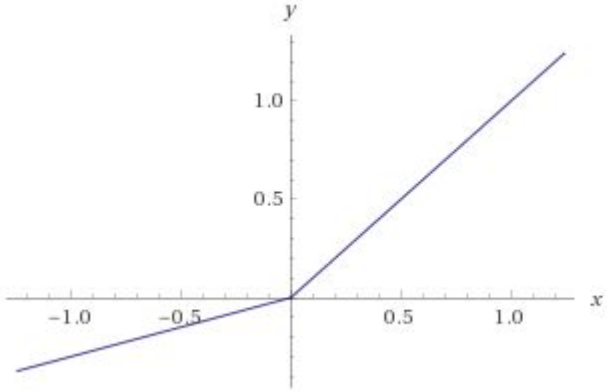


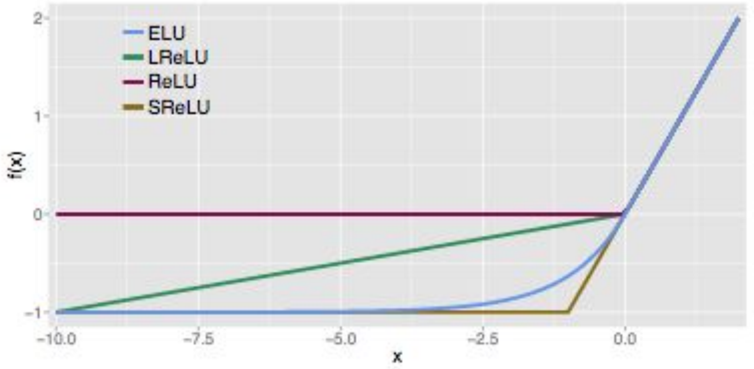
- This function doesn't saturate at large values of x
- The derivative does not saturate at large values of x

- ReLU should be your go-to activation function

ReLU Generalizations

$$\text{LeakyReLu}(x) = \begin{cases} ax & x \leq 0, \text{for } a \in [0,1] \\ x & x > 0 \end{cases}$$



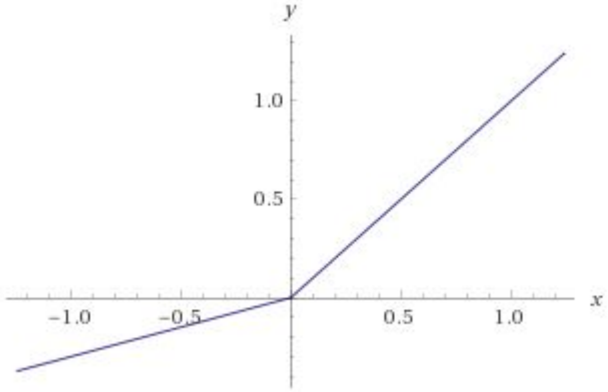


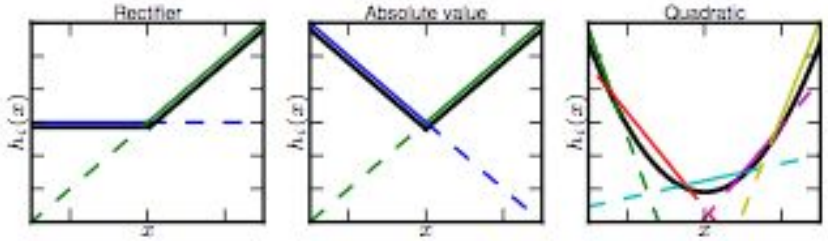
- See also ELU, SReLU
- https://arxiv.org/pdf/1511.07289.pdf

ReLU Generalizations

$$\text{LeakyReLu}(x) = \begin{cases} ax & x \leq 0, \text{for } a \in [0,1] \\ x & x > 0 \end{cases}$$

$$\text{maxout}(x) = \max_{j \in [1,k]} x^T W_{ij} + b_{ij}$$





https://arxiv.org/pdf/1302.4389.pdf

Exercise

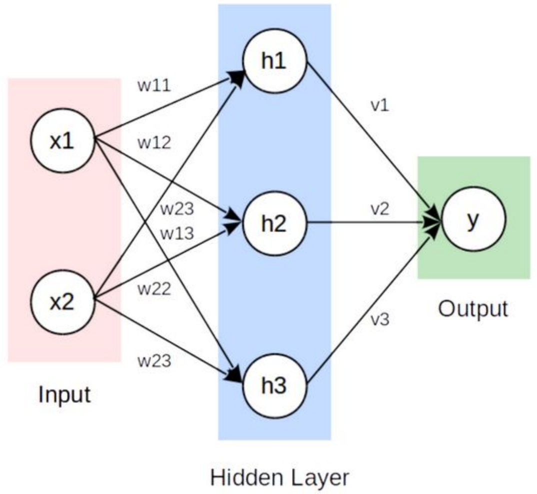
For each one of the activation function we’ve discussed, code up your own version. Make a visualization of all of your activation functions over a small range of x (+/- 1). In your visualization, overlay them on top of one another

Output Functions

Linear

For continuous target variable

$$\hat{y} = \vec{v}^T \vec{h} + c$$



Linear

Accompanying Loss Function

$$\hat{y} = \vec{v}^T \vec{h} + c$$

Squared Error Loss

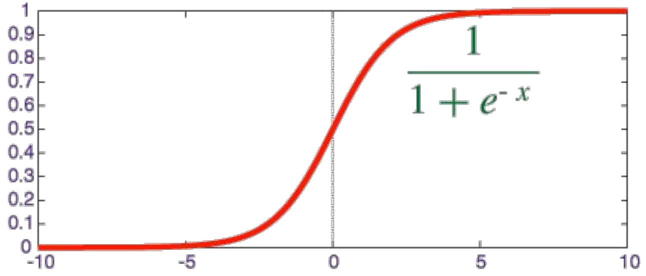
$$L = \frac{1}{N}\frac{1}{2}\sum_i ((y_i - \hat{y_i})^2)$$

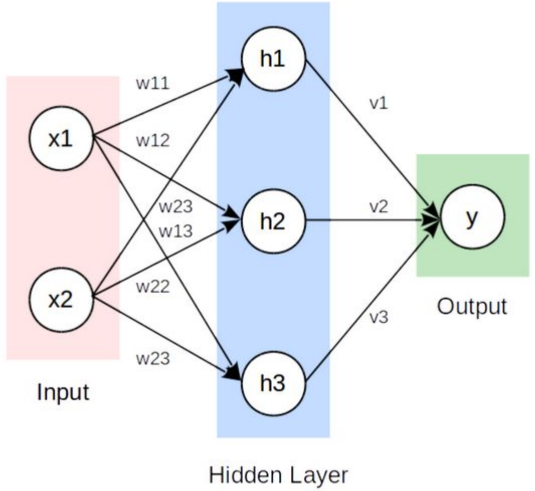
$$\frac{\partial L}{\partial \hat{y}} = y_i - \hat{y_i}$$

Sigmoid

For binary target variables (Bernoulli)

$$\hat{y} = \sigma(\vec{v}^T \vec{h} + c)$$





Intended to represent a probability over the two classes. It “squashes” a real-valued scalar to lie within [0,1].

Sigmoid

Accompanying Loss Function

Binary Cross Entropy

$$L = -y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)$$

$$\frac{dL}{d\hat{y}} = \left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i} \right)$$

Sigmoid

Accompanying Loss Function - Binary Cross Entropy

$$L = -y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)$$

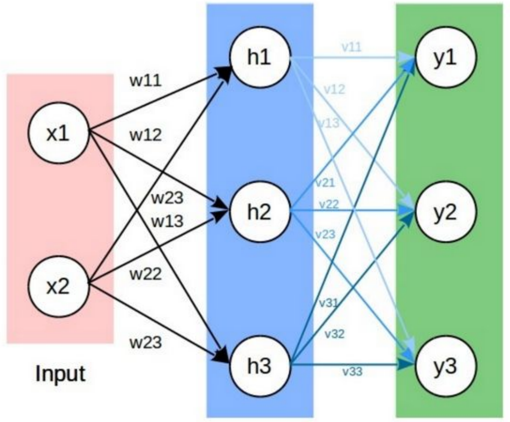
y	yhat	L
1	.51	0.67
1	.49	0.71
1	.99	0.01
1	0.01	4.60
0	.99	4.60

Softmax

For categorical outputs

$$\vec{z} = W^T \vec{h} + \vec{b}$$

$$\hat{y}_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$



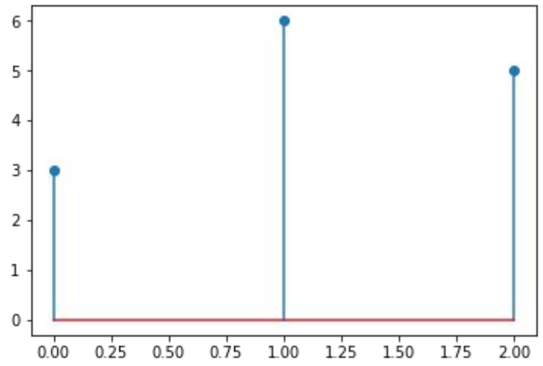
Converts real-valued vector (logits) into a probability vector over K classes. Similar to Sigmoid in how it squashes input into a valid probability vector.

(Boltzmann distribution)

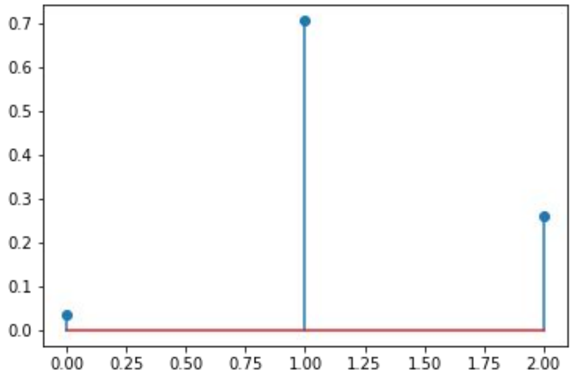
Softmax

$$\hat{y}_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

$$\vec{z} = \{3, 6, 5\}$$



$$\text{softmax}(\vec{z}) = \{0.04, 0.7, 0.26\}$$



A monotonic but non-linear transformation.

Softmax

Accompanying Loss Function

Categorical Cross Entropy

$$L = -\sum_j y_j \log(\hat{y}_j)$$

$$\vec{y} = \{0, 0, 1, 0\}$$

$$\vec{\hat{y}} = \{.25, .25, .4, .1\}$$

$$L = -\vec{y}^T \cdot \log(\vec{\hat{y}}) = -1 * \log(0.4)$$

Softmax

Accompanying Loss Function - Cross Entropy

$$\frac{dL}{d\hat{y}} = \vec{\hat{y}} - \vec{y}$$

$$\begin{aligned} \vec{y_i} &= \{0, 1, 0\} \\ \vec{\hat{y_i}} &= \{0.1, 0.7, 0.2\} \\ \frac{dL}{d\hat{y}} &= \{0.1, -0.3, 0.2\} \end{aligned}$$

https://deepnotes.io/softmax-crossentropy

Regularization Strategies

Parameter Penalization

- Just like with LASSO and Ridge Regression, the premise here is that we can constrain the magnitudes of the learned parameters
- By keeping parameters “small” this should limit overfitting
- No longer interested in parameter trajectories and feature selection, since most parameters are for hidden interactions

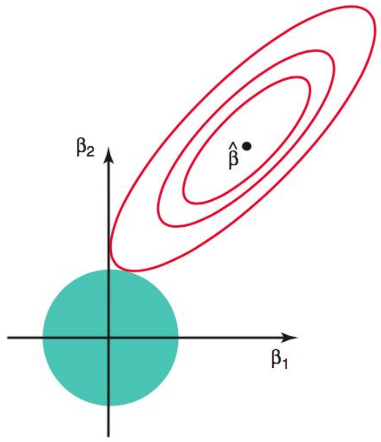
$$L = L(\theta; X, y) + \lambda g(\theta)$$

Parameter Penalization

$$L = L(\theta; X, y) + \lambda g(\theta)$$

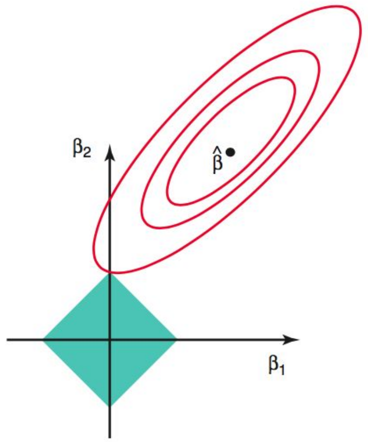
L2 Regularization

$$g(\theta) = ||\theta||_2 = \sum_k \theta_k^2$$



L1 Regularization

$$g(\theta) = ||\theta||_1 = \sum_k |\theta_k|$$



Data Augmentation

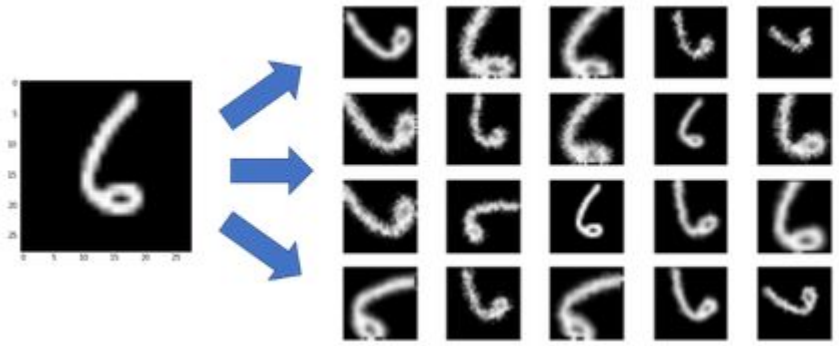
Overfitting is one of the greatest challenges for deep networks. With large models and (relatively) small datasets, it becomes hard to avoid memorizing the training data.

Data Augmentation is a set of techniques that allows us to create more training data in a way that preserves important aspects, and improve generalizability.

This has been mostly applied to images, and might be more challenging in other domains.

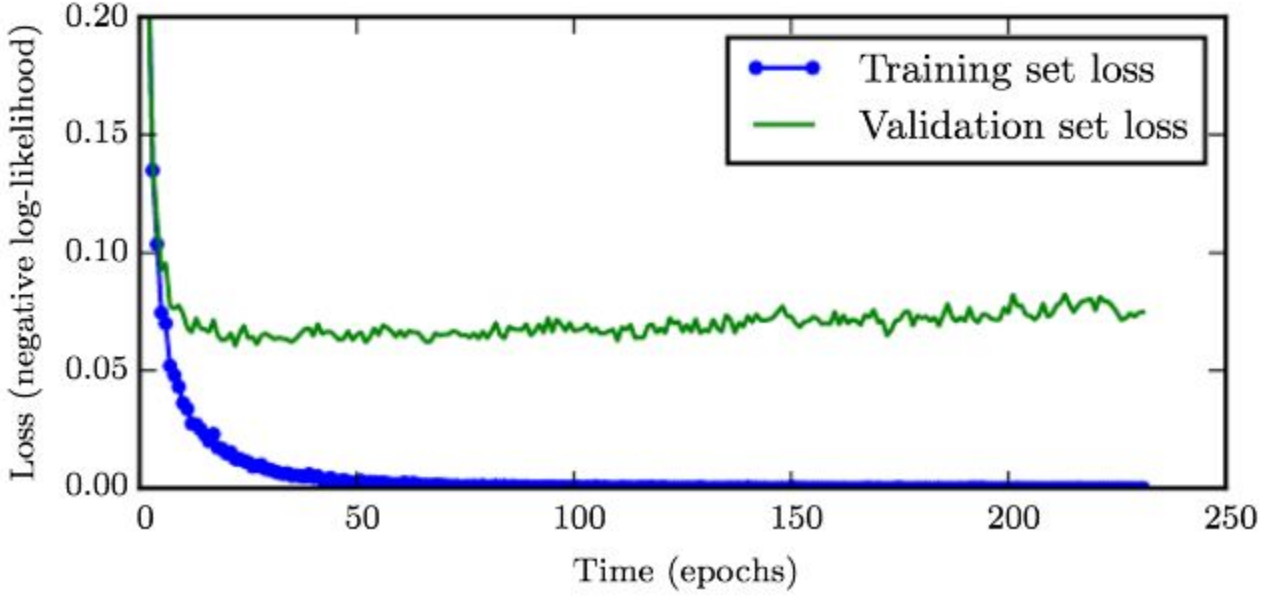
Data Augmentation

- Rotation
- Translation
- Scaling
- Patch sampling



- For any attribute that you think your model should be **invariant to**, creating synthetic examples is a form of regularizing the model
- Nice description of lots of methods and importance in image classification tasks - http://benanne.github.io/2014/04/05/galaxy-zoo.html

Early Stopping

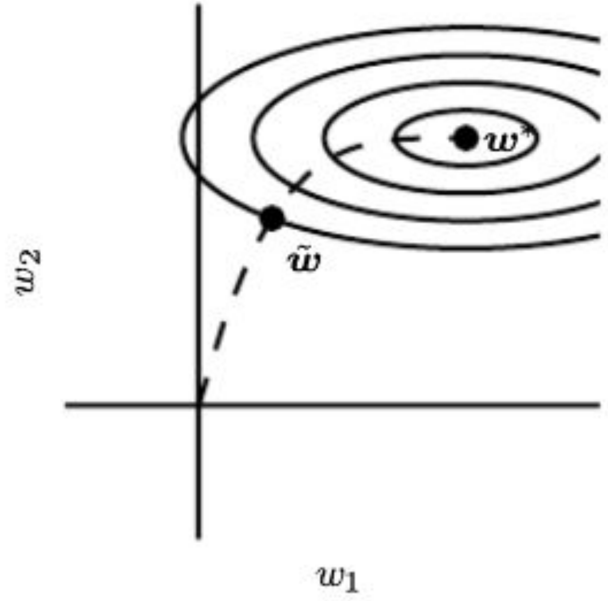


As training iteration increases, we are more likely to be optimizing the weights toward areas that over-fit the training data. Simple solution - just don't train for very long, or only use the learned weights from early on in the training process.

Early Stopping

We limit the dynamics of backprop - from our initial starting point, we can only travel within a certain volume of parameter space in a given number of iterations.

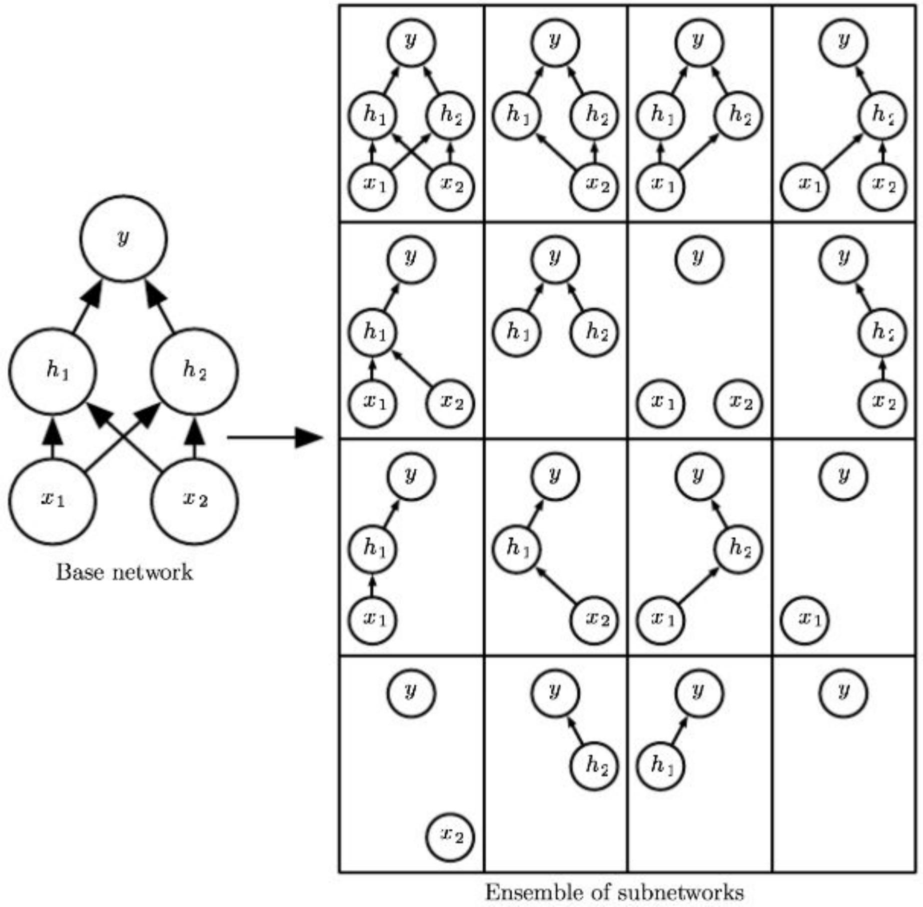
This forces us to “settle” for slightly worse training error, but ends up improving test error.



Dropout

Ensemble methods (like boosting or bagging) are probably helpful with ANNs, but too expensive to be feasible.

How about a method to approximate an ensemble of ANNs?

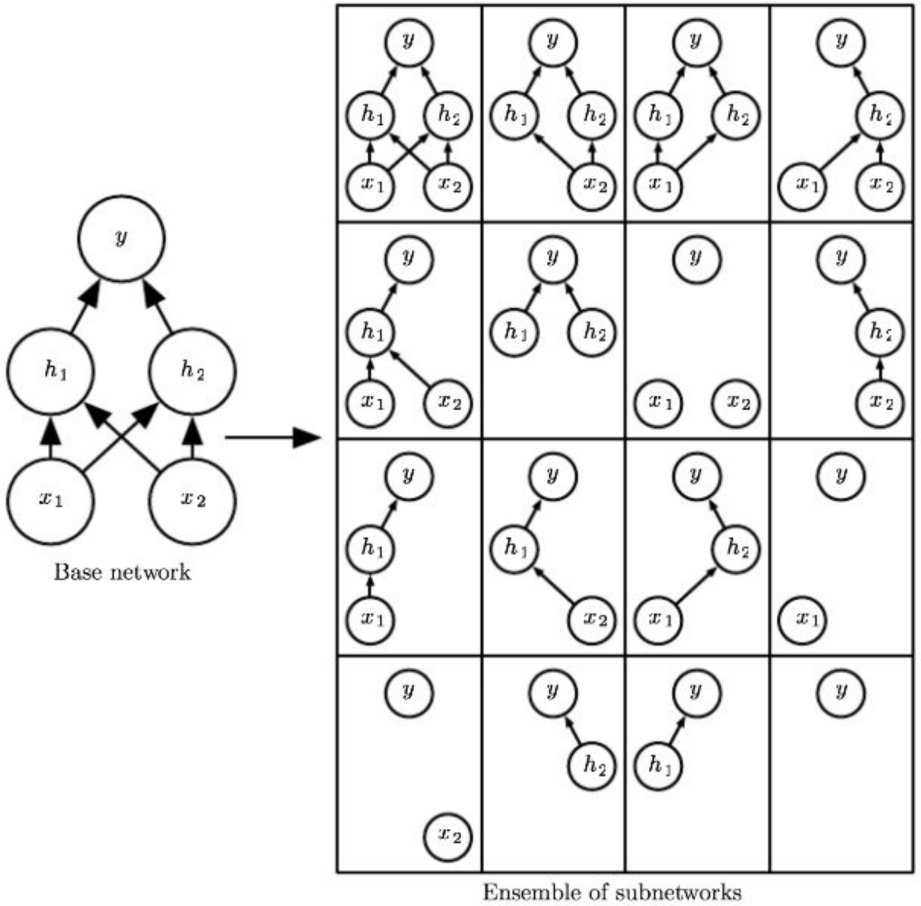


Dropout

If we were to randomly remove some connections in our model, consider the set of all possible sub-networks.

The **dropout** method simultaneously trains the ensemble of all the sub-networks.

Each sub-net is an impaired weak learner, the overall ensemble is highly regularized relative to a single full network.



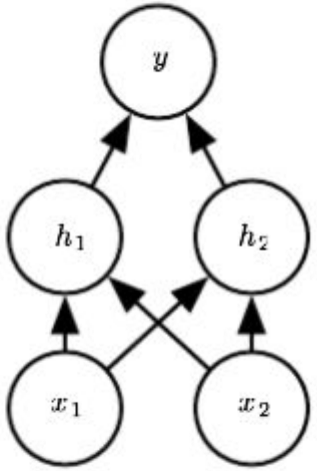
Dropout

Procedure

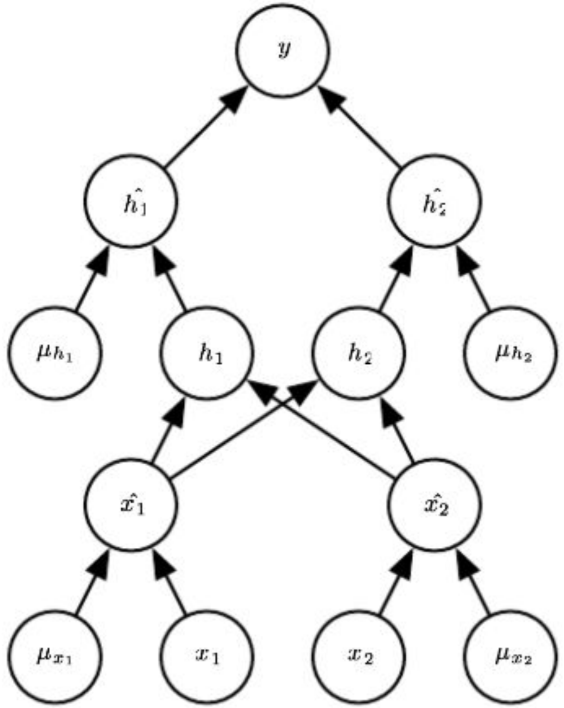
1. For each batch of model training, we randomly remove some connections from our model. This can achieved by creating a *binary mask* matrix and multiplying it with our weight matrices during the forward pass.
2. We compute the forward pass and backward pass and updated the weights as usual.
3. On the next batch, we sample a new binary mask, and repeat.

Dropout

Original network



Network with mask



Dropout

Prediction
Once we have trained a network, our optimal prediction should be a weighted average over all sub-networks

$$p(y|x) = \sum_{\mu} p(\mu) p(y|x, \mu)$$

But this is intractable because there are exponentially-many binary masks, and surely we've only sampled a few of them.

Turns out, a useful estimator of the true ensemble prediction is to just a single network - the one with all the weights.