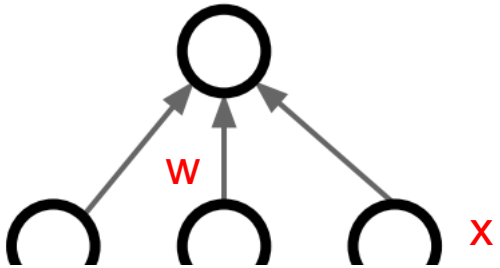
CMSC 510 – L18
Regularization Methods for Machine Learning

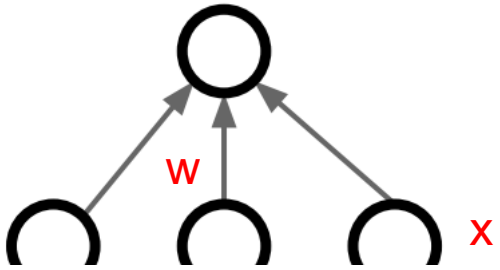
Instructor:
Dr. Tom Arodz

Nonlinear models



So far, we discussed linear models
h(x) = wx+b,
and regularizers that rely on
model parameters being feature weights

Nonlinear models

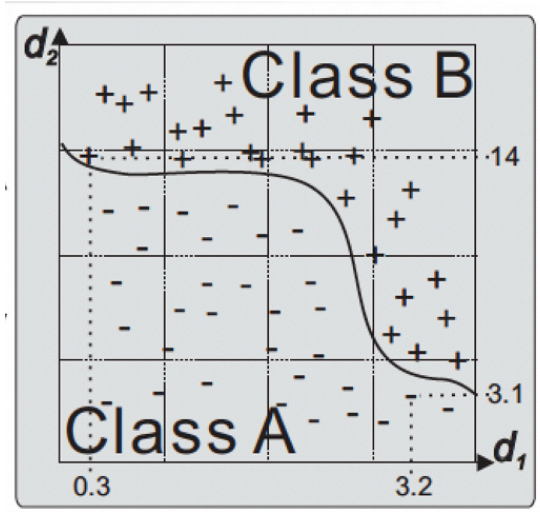


So far, we discussed linear models
h(x) = wx+b,
and regularizers that rely on
model parameters being feature weights

Now, we'll move to nonlinear models

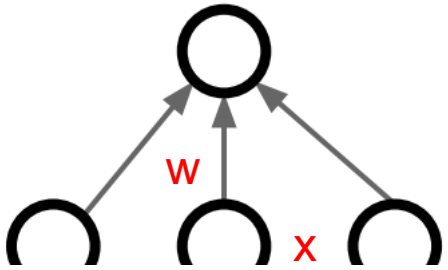
Here, parameters may not have obvious relationship to input features

We'll need different regularization techniques

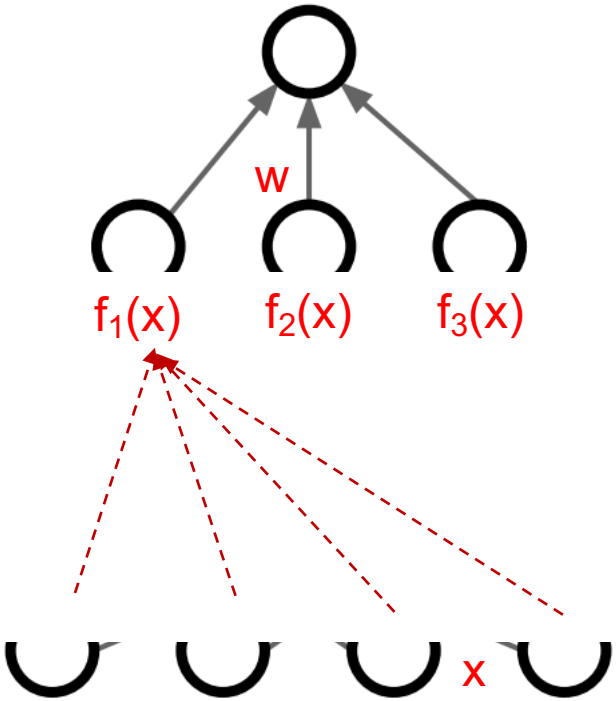


Nonlinear models

- So far, we discussed **linear** models
  h(x) = wx+b,
  and regularizers that rely on
  model parameters being feature weights

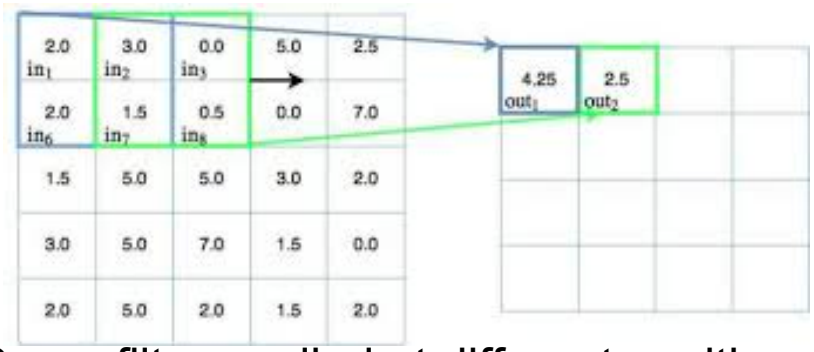


- We will go through two approaches to **nonlinear** models
  - Multilayer neural networks
  - Kernel methods
- In both cases, instead of $h(x)=\sum_j w_j x_j$
- we have $h(x)=\sum_k w_k f_k(x)$
  - We do not use the features x directly,
  - but process them using some function f_k
- WE WILL USE IMAGES AS AN EXAMPLE

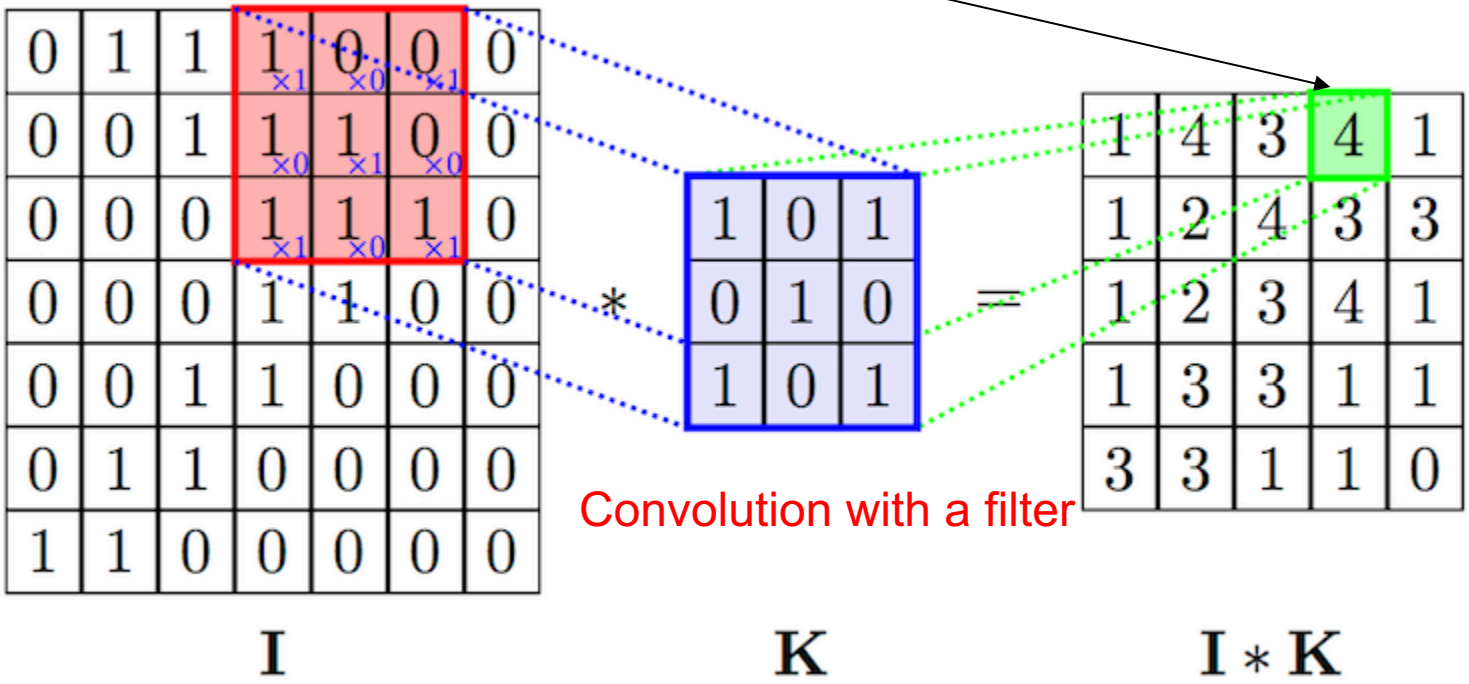


Nonlinear classification

- We can try to pre-construct some "new features" and use linear classifier on those
- $h(x)=\sum_k w_k \text{ new\_features}_k(x)$



Same filter applied at different positions



Convolution with a filter

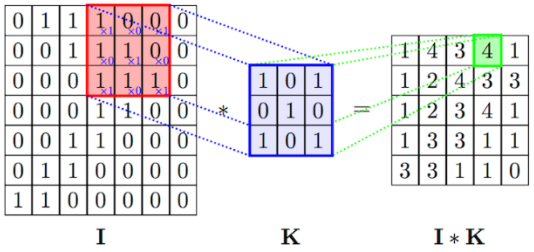
Nonlinear classification

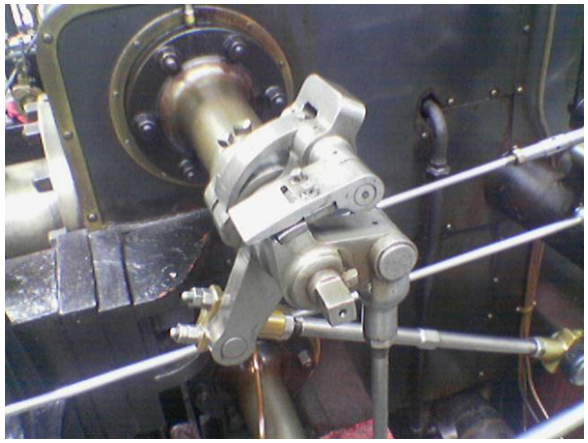
- We can try to pre-construct some "new features" and use linear classifier in those
- $h(x)=\sum_j w_j$ "new features"

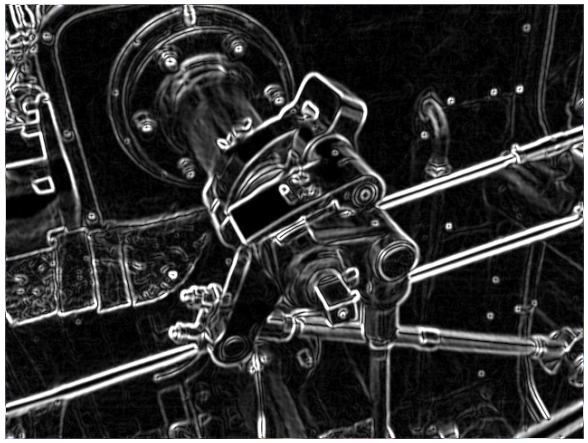
-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Basically, some clever "photoshop" operation





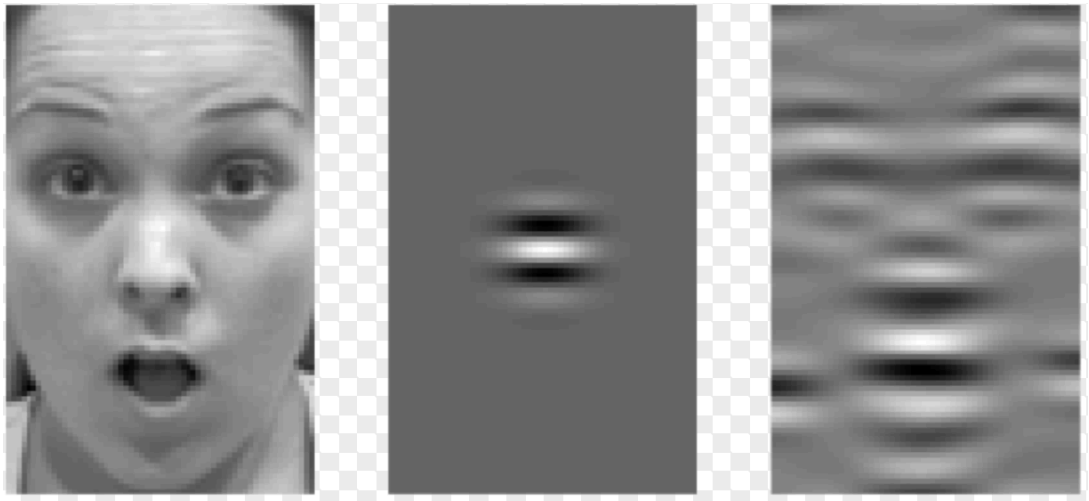


Nonlinear classification

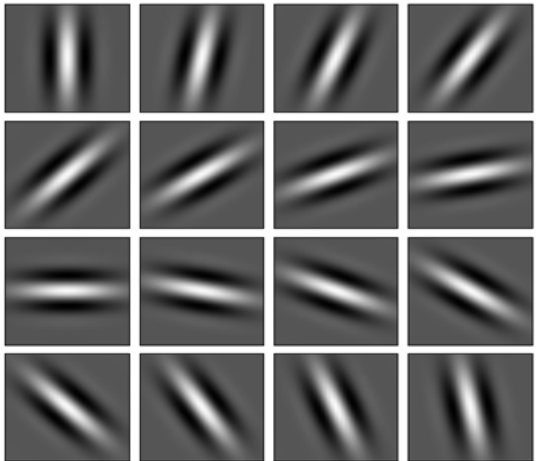
- We can try to pre-construct some "new features" and use linear classifier in those
- $h(x)=\sum_j w_j$ "new features"

Basically, some clever "photoshop" operation

Image Filter Output



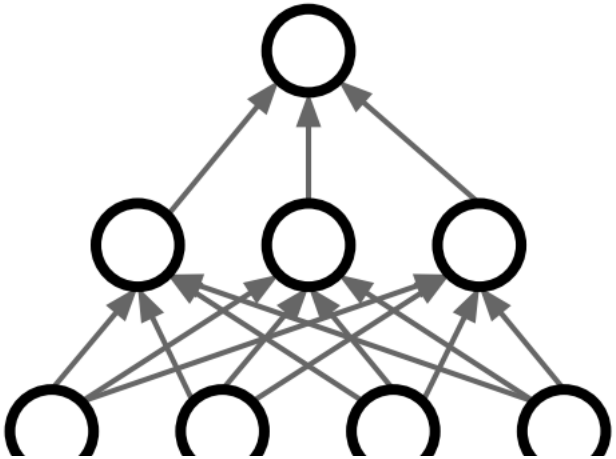
- Drawback: We have to do a good job designing the features (e.g. Gabor filters)

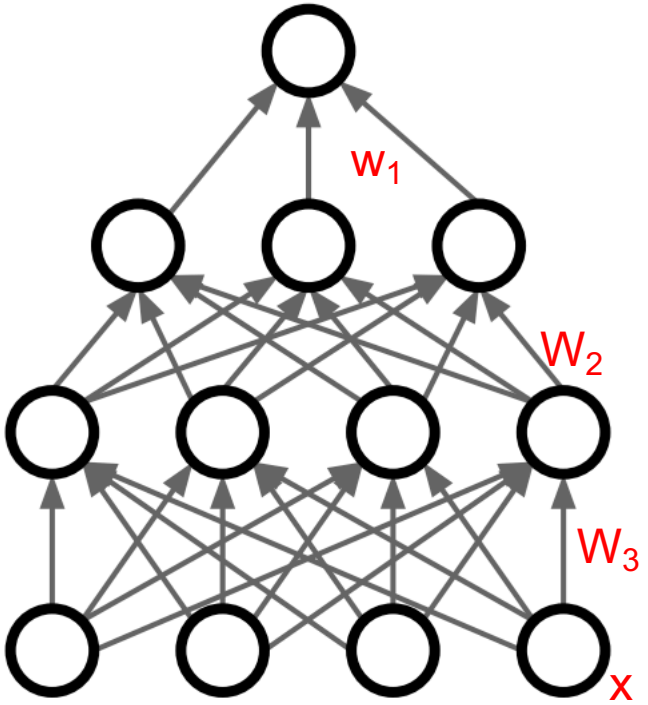


Nonlinear classification

- $h(x)=\sum_j w_j$ "new features"

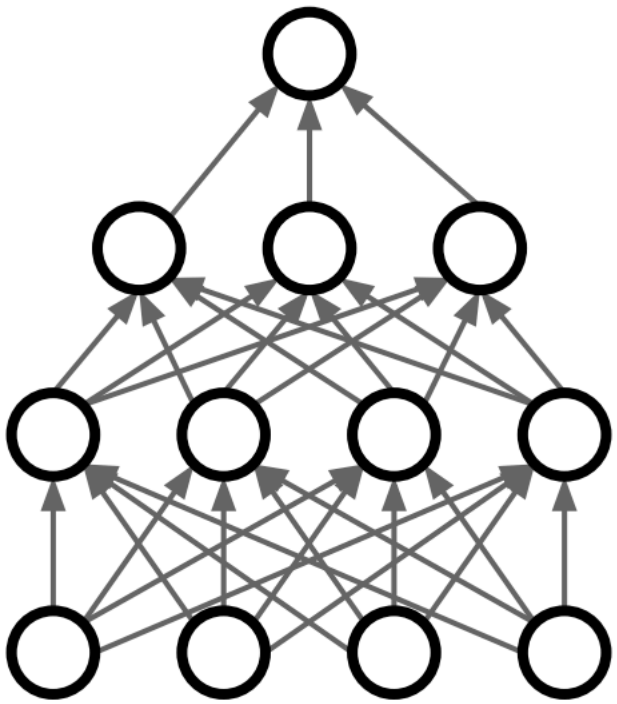
- Instead of figuring out how to design "new features"
- We can try learning them from the dataset
  - Specify some parameters describing the filters, and learn them, just as we learn weights

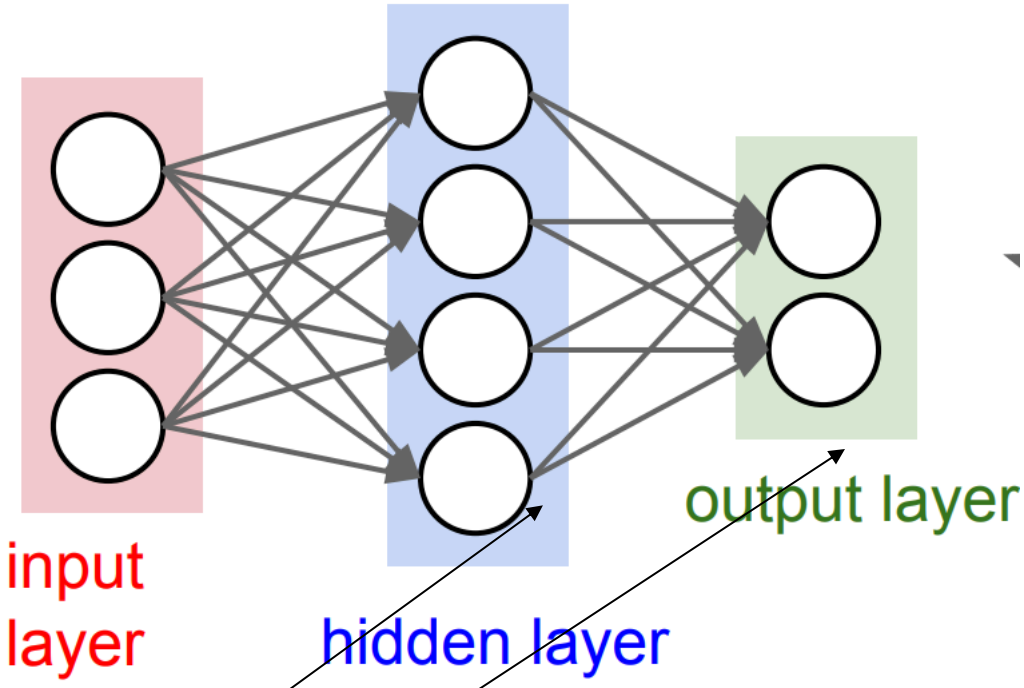




Nonlinear classification

- We get a feed forward neural network





- Typically:
  - Multiple classes / output neurons
  - Multiple nonlinear layers: linear z=Wx followed by nonlinear y=a(z)

Binary classification

- One output neuron – probability of class y=1: P_{model}
- Loss = $H(P_{data}, P_{model}) = KL_{div}(P_{data} || P_{model})$
  - We observe data where class is certain
    - *P(class 1) for given x* is either 1 or 0
  - Model says data came from distrib.
    - *P(class 1) for given x* is $a(w_1^T x)$
  - How surprised we are by data if model was true?
- H = KL only if P_{data} is "all vs nothing" P(y)=0 or 1
  - Entropy of p
    - Measures average "surprise" in seeing events sampled from p
  - Cross-entropy of *p* and *q*:
    - Measures "surprise" in seeing data sampled from p if we have some info/model saying data should follow q
  - Kullback-Leibler divergence of *q* from *p*
    - *Additional surprise: Cross-entropy (p,q)- Entropy(p)*

Multi-class predictions

- One output neuron per class: P_{model}
  - K classes = K outputs: $w_1^T x$, $w_2^T x$,…,$w_k^T x$
  - output j represents the probability $P_{model}(y=j|x)$

- True class: encode as one-hot vector P_{data}
  - $P_{data}(y=\text{wrong class}|x)=0$
  - $P_{data}(y=\text{true class}|x)=1$

- Loss = $H(P_{data}, P_{model}) = KL_{div}(P_{data} \,||\, P_{model})$
  - $P_{model}=[0.1, 0.8, 0.06, 0.04]$
  - $P_{data}=[0, 1, 0, 0]$
  - Loss = - [0*log(0.1) + 1*log(0.8) + 0*log(0.06) +0*log(0.04)]

Multi-class predictions

- One output neuron per class: P_{model}
  - K classes = K outputs: $w_1^T x$, $w_2^T x$,…,$w_k^T x$
- For multi-class problems, output may not be a probability P(y|x)
  - $w_1^T x$, $w_2^T x$,…,$w_k^T x$ may be <0 or >1
- What we used in binary classification:
  - Sigmoid($w_1^T x$)

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T} \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T} \mathbf{w}_k}}$$

Multi-class predictions

- One output neuron per class: P_{model}
  - K classes = K outputs: $w_1^T x$, $w_2^T x$,…,$w_k^T x$
- For multi-class problems, output may not be a probability P(y|x)
  - sigmoid($w_1^T x$), sigmoid($w_2^T x$),…,sigmoid($w_k^T x$) may not add up to 1
- Solution: *softmax layer* as output layer

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^T \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^T \mathbf{w}_k}}$$

Multi-class predictions

- Softmax vs sigmoid

WolframAlpha

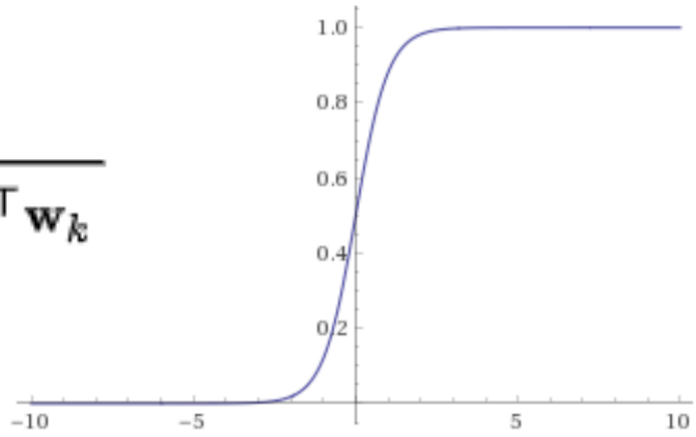
exp(x) / (exp(x)+exp(-x)) plot x from -10 to 10

Extended Keyboard Upload

Input interpretation:

plot	$\frac{\exp(x)}{\exp(x)+\exp(-x)}$	$x = -10$ to 10

Plot:



$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Deep neural networks

- Sigmoid has the problem of "vanishing gradient"
- Softmax is a generalization of sigmoid, it has the same problem

- But:

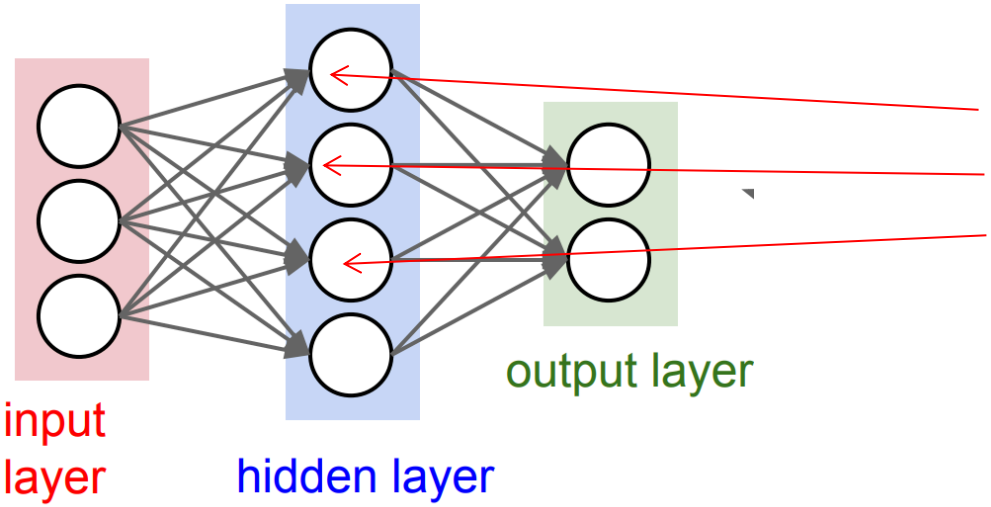
- Cross-Entropy Loss=$H(P_{data}, P_{model})=-\Sigma P_{data}\log(P_{model})$

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

- We will be taking $\log(\exp(x^T w))$
  - Reducing the problem

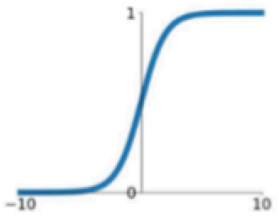
Deep neural networks

- With Cross-Entropy after softmax, the last layer does not lead to flat regions (vanishing gradients)
  - Just like logistic loss for two classes
  - Unlike Mean Square Error loss
- We do not have that option for hidden layers
  - A sigmoid as activation function in hidden layers may get saturated: "vanishing gradients"



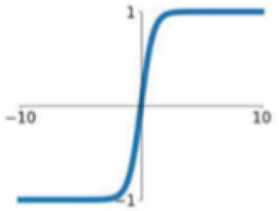
Sigmoid

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



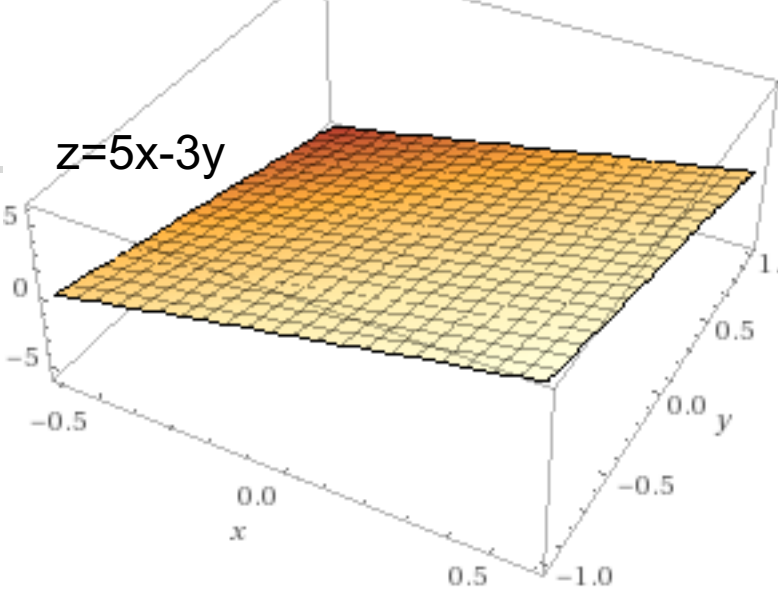
tanh

$$\tanh(x)$$

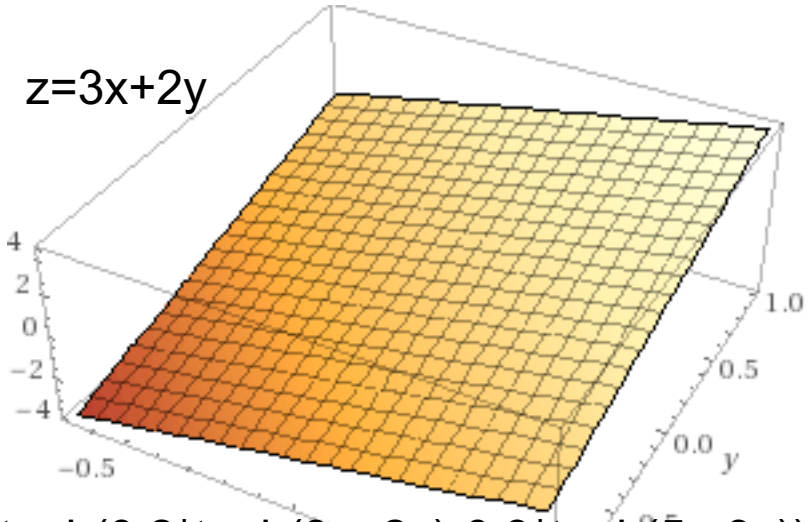


Vanishing gradients

z=5x-3y



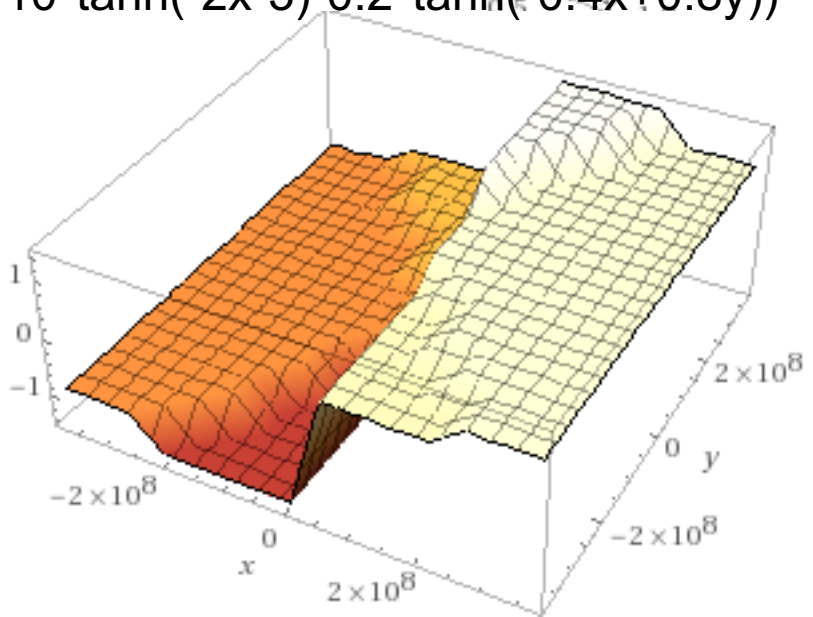
z=3x+2y

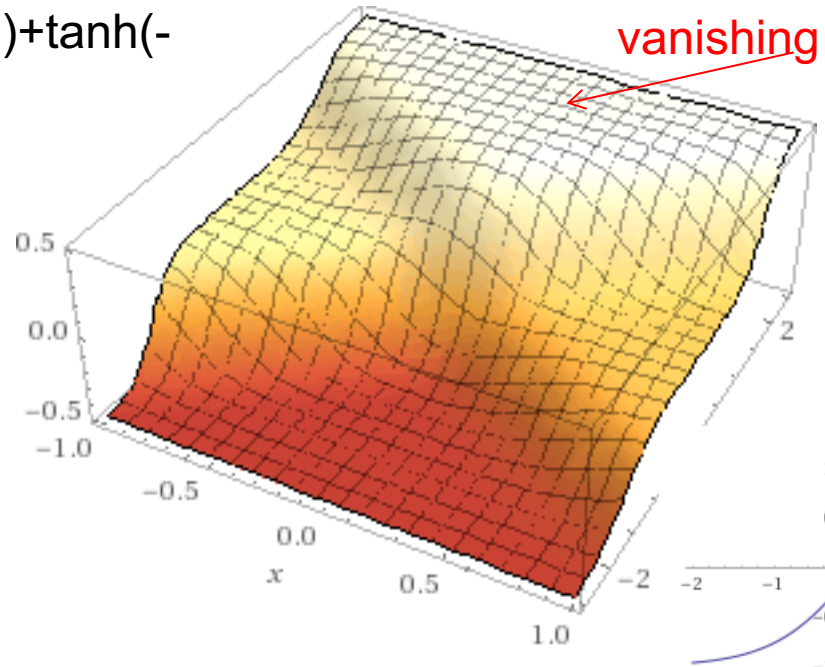


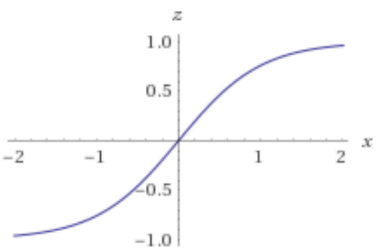
z=0.2*tanh(3x+2y)-0.3*tanh(5x-3y)

vanishing gradients

z=tanh(0.2*tanh(3x+2y)-0.3*tanh(5x-3y))+tanh(-10*tanh(-2x-3)-0.2*tanh(-0.4x+0.8y))







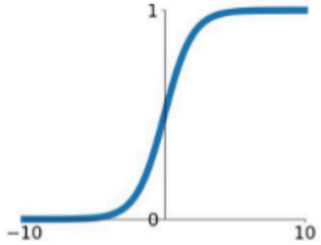
Modern activation functions

- Rectified Linear Unit (and similar):

Activation Functions

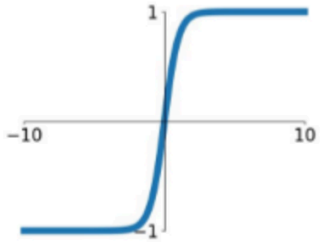
Sigmoid

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



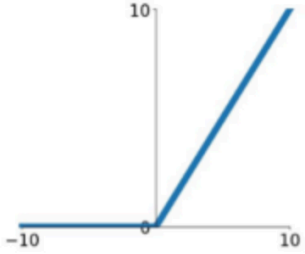
tanh

$$\tanh(x)$$



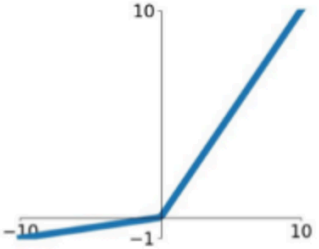
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

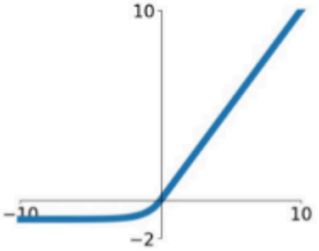


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

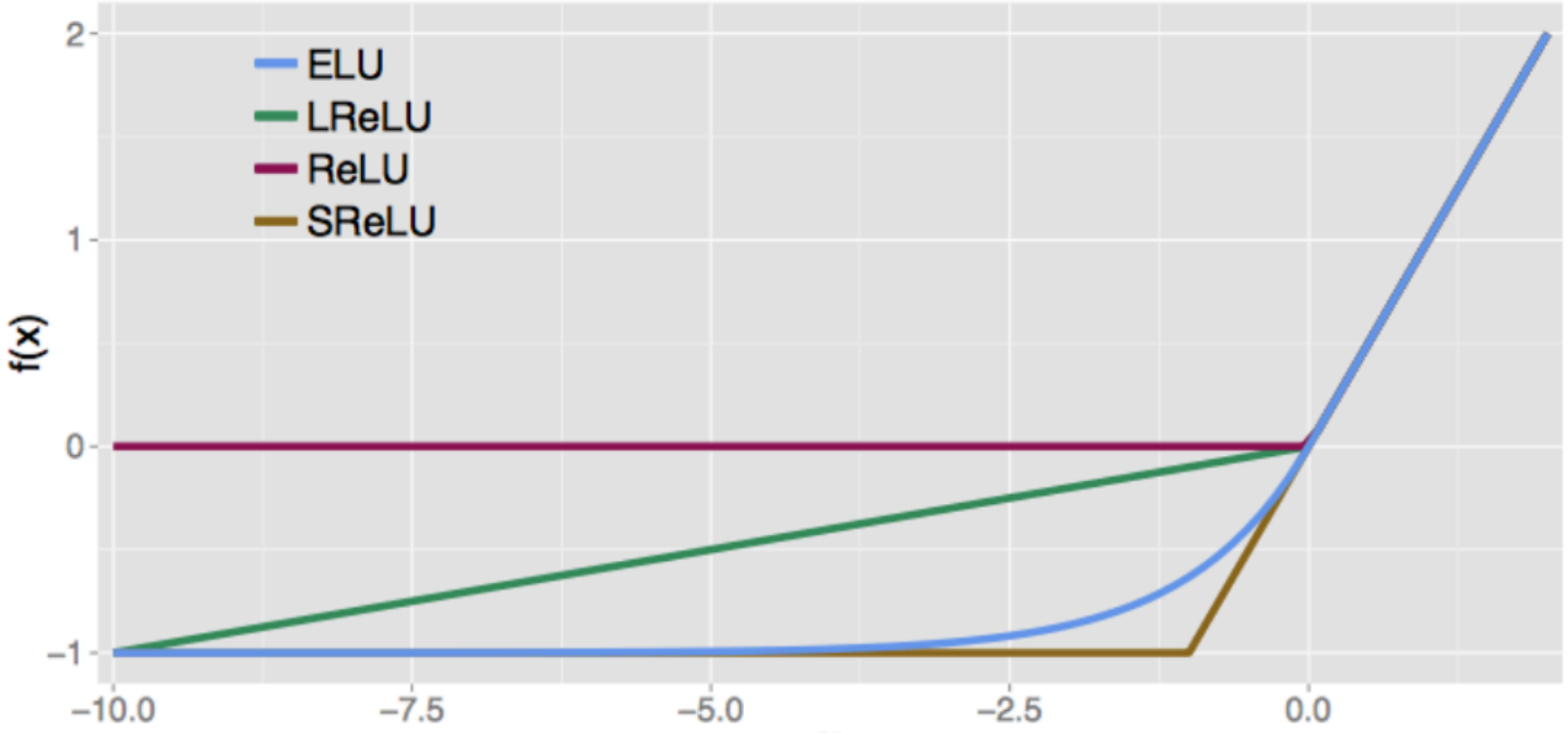
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



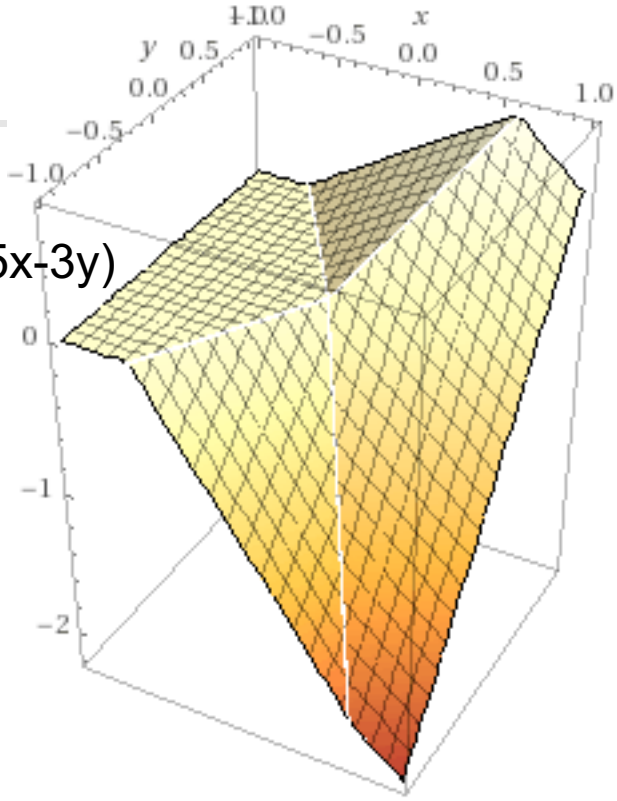
Deep neural networks

- Rectified Linear Unit (and similar):

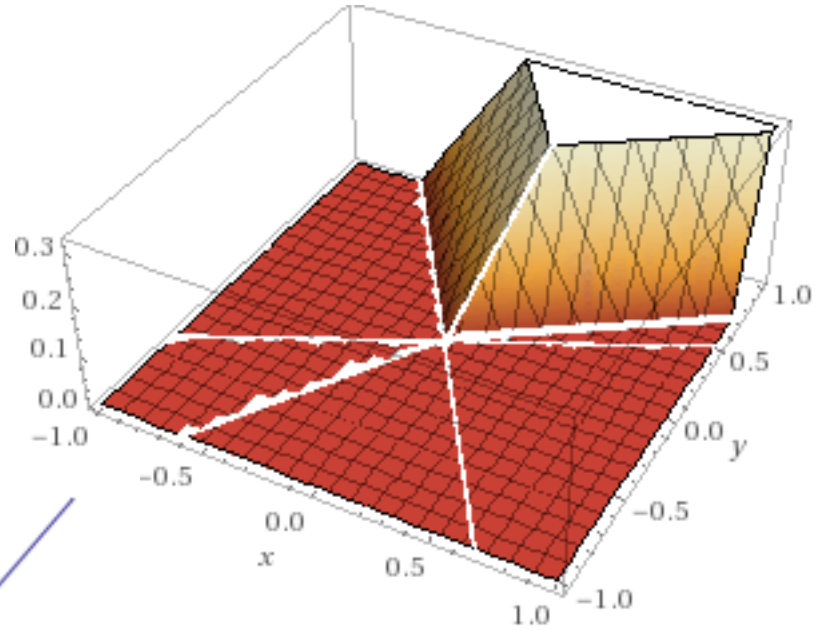


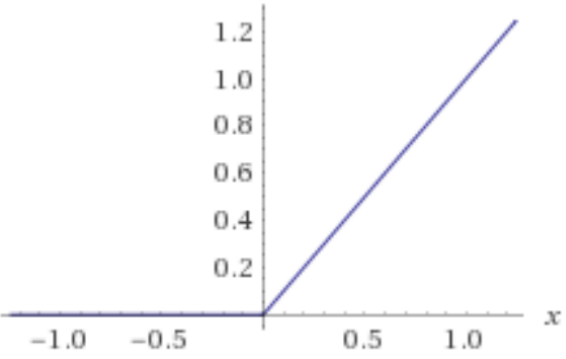
Nonlinearity

- ReLU is cabable of nonlinear classification



0.2*ReLU(3x+2y)-0.3*ReLU(5x-3y)

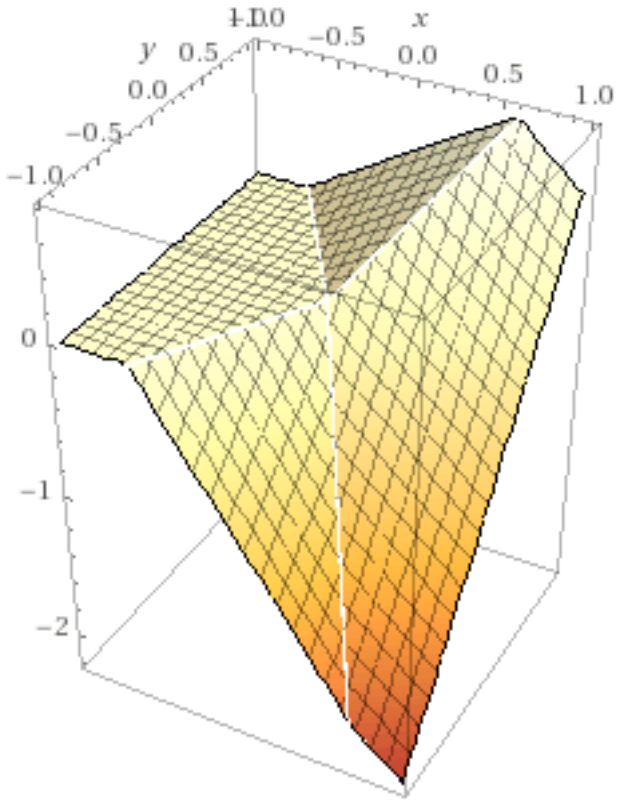




ReLU(0.2*ReLU(3x+2y)-0.3*ReLU(5x-3y))+
ReLU(-1*ReLU(-2x-3)-0.2*ReLU(-0.4x+0.8y))

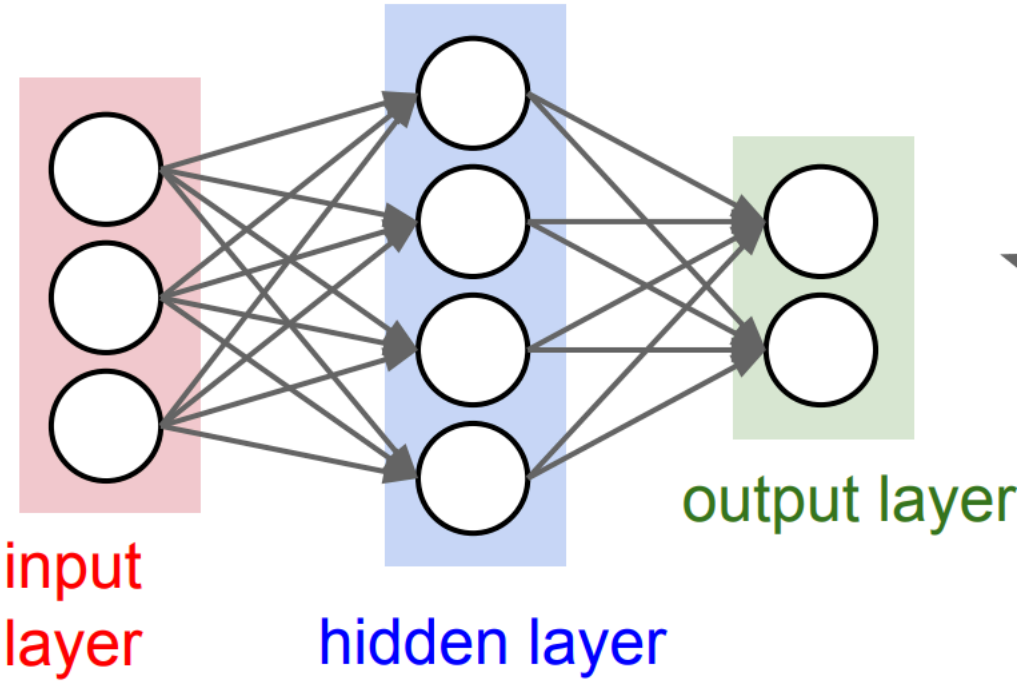
Deep neural networks

- We change the activation function:
  - ReLU
  - ELU
- These still have flat regions ("dying ReLU problem"), but only on one side, it easier to manage



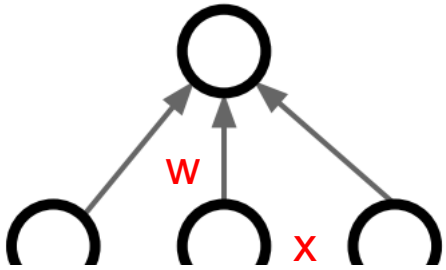
Nonlinear classification

- Each neuron in the hidden layer has its own set of weights
  - Many, many parameters to adjust using gradient descent!
  - Overfitting!

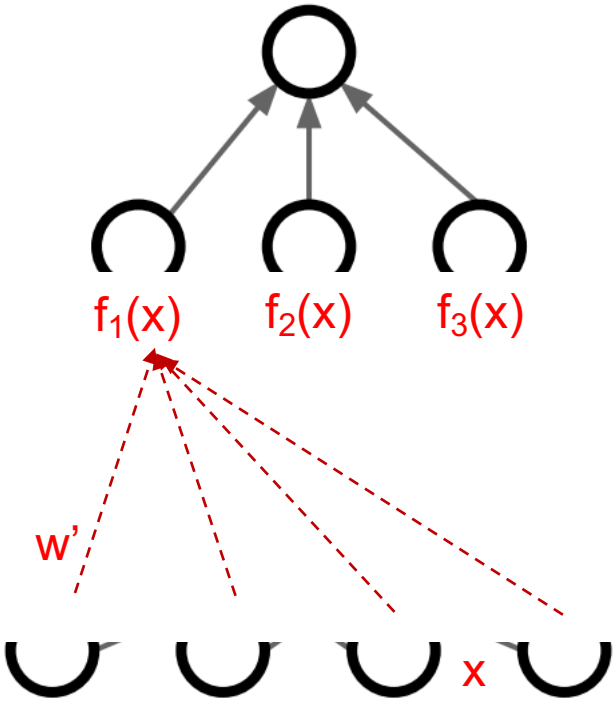


Nonlinear models

- So far, we discussed **linear** models
  h(x) = wx+b,
  and regularizers that rely on
  model parameters being feature weights

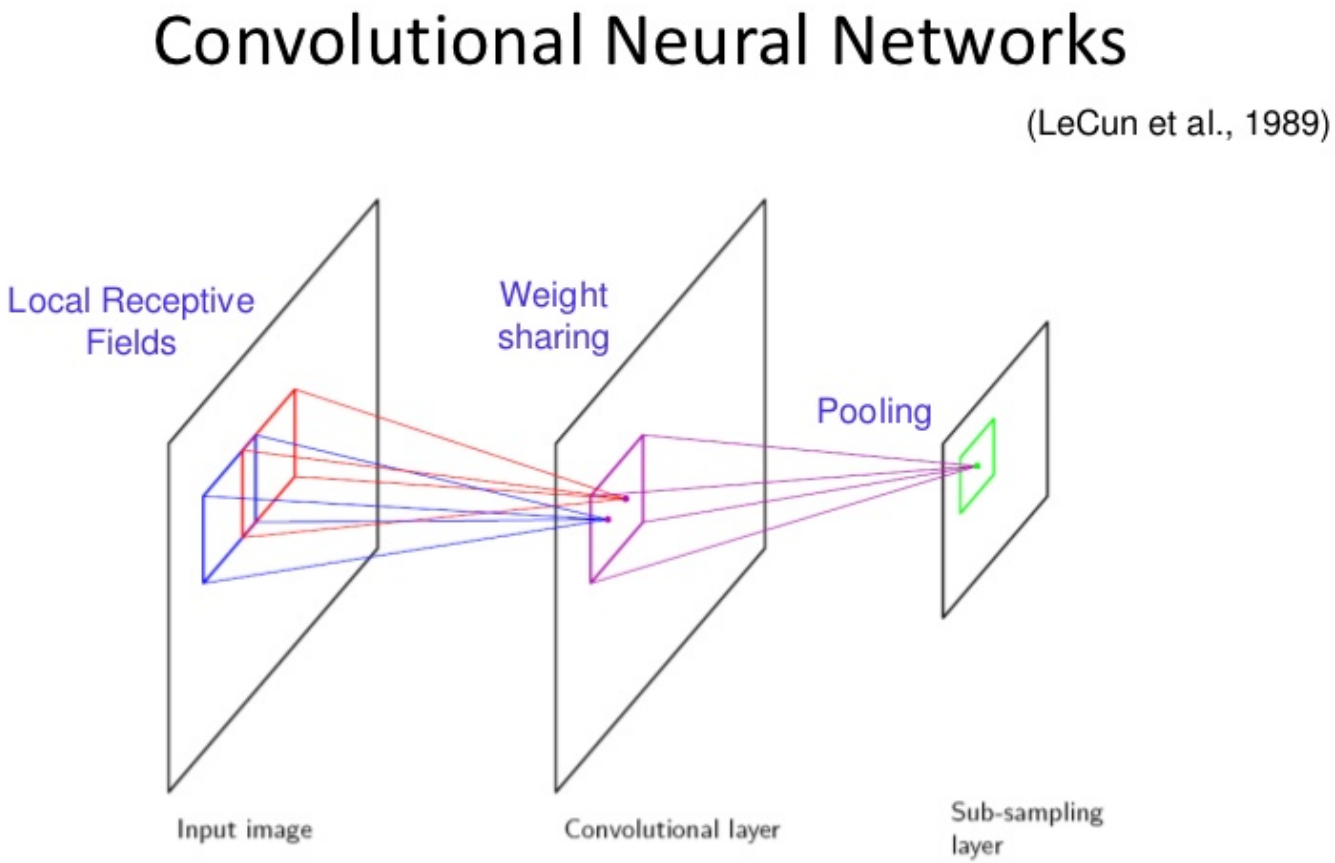


- Still, at the level close to input, parameters can have some relationship to features
  - A relationship that can be exploited for regularization



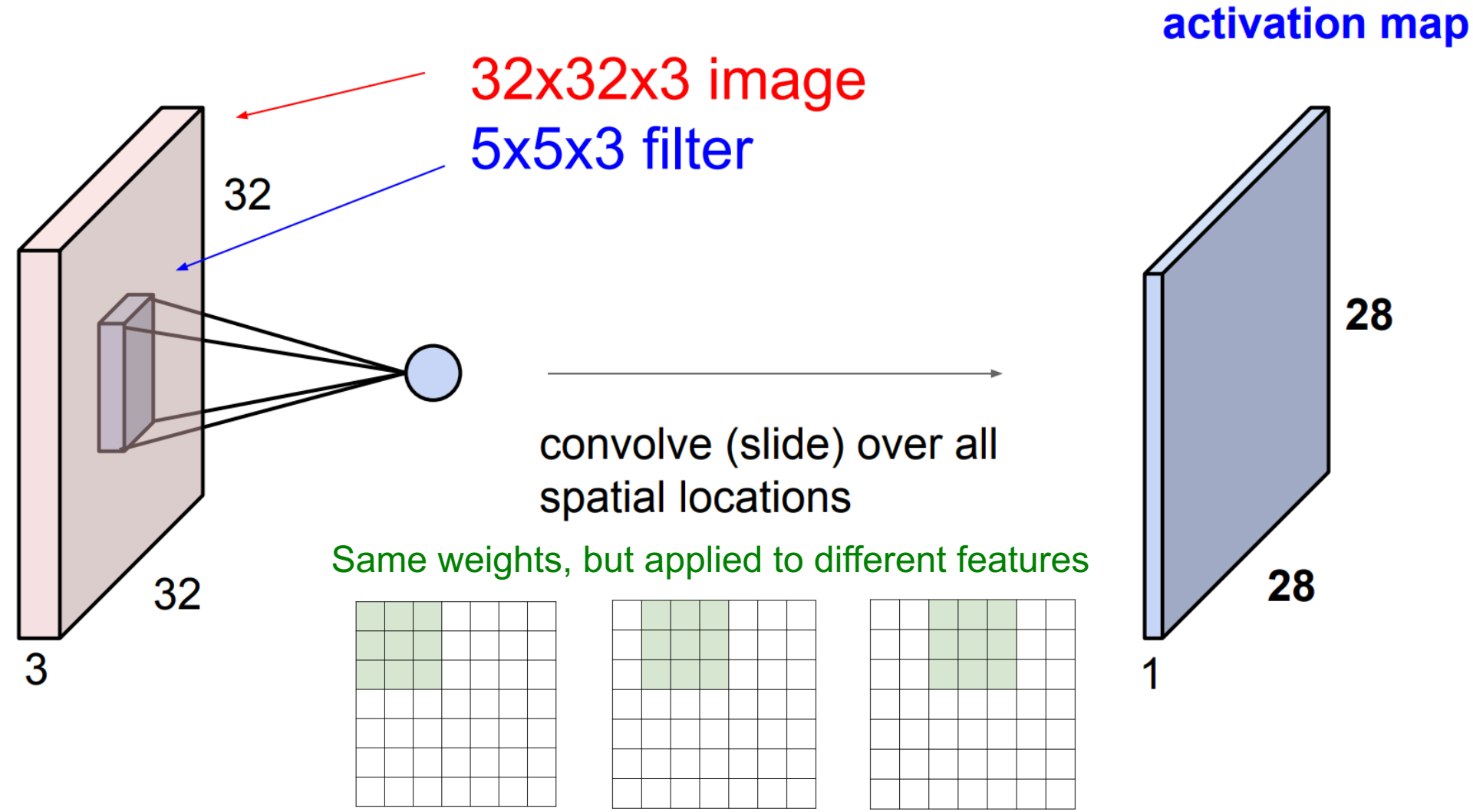
Regularization in Neural Nets

- Weight sharing
  - Red and blue weights are the same
  - We reduce the number of parameters significantly



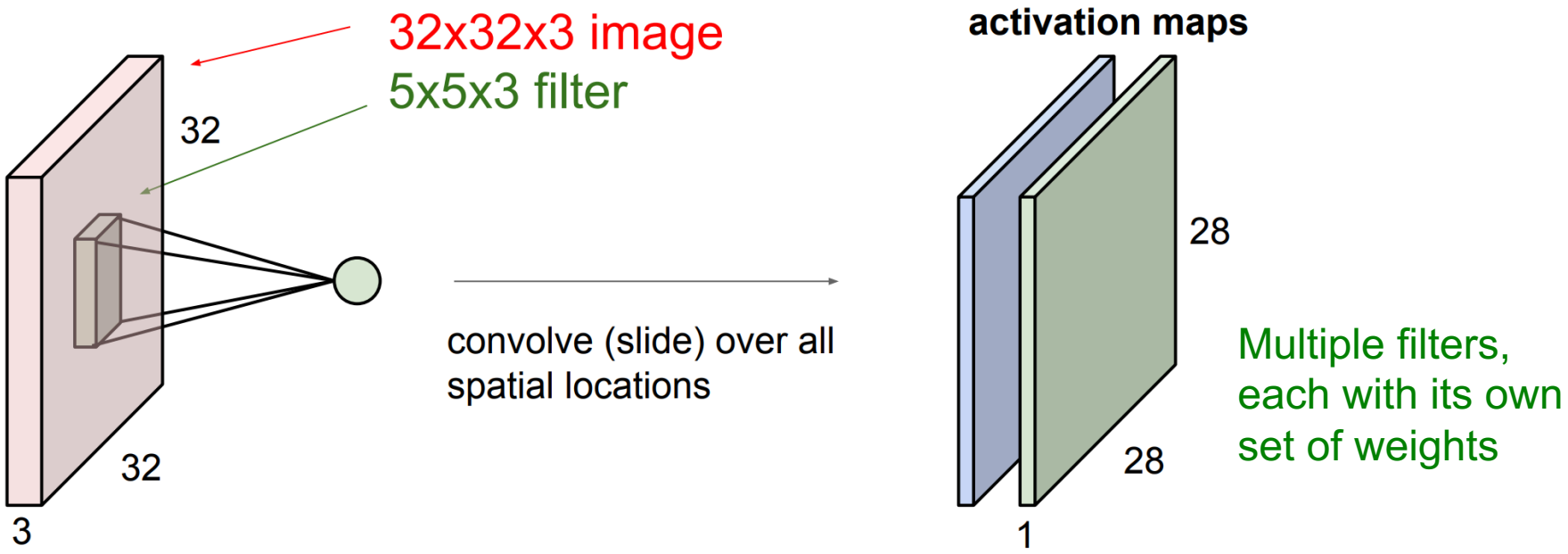
Regularization in Neural Nets

- Weight sharing



Regularization in Neural Nets

- Reducing the number of "features" for next layer



Single depth slice

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

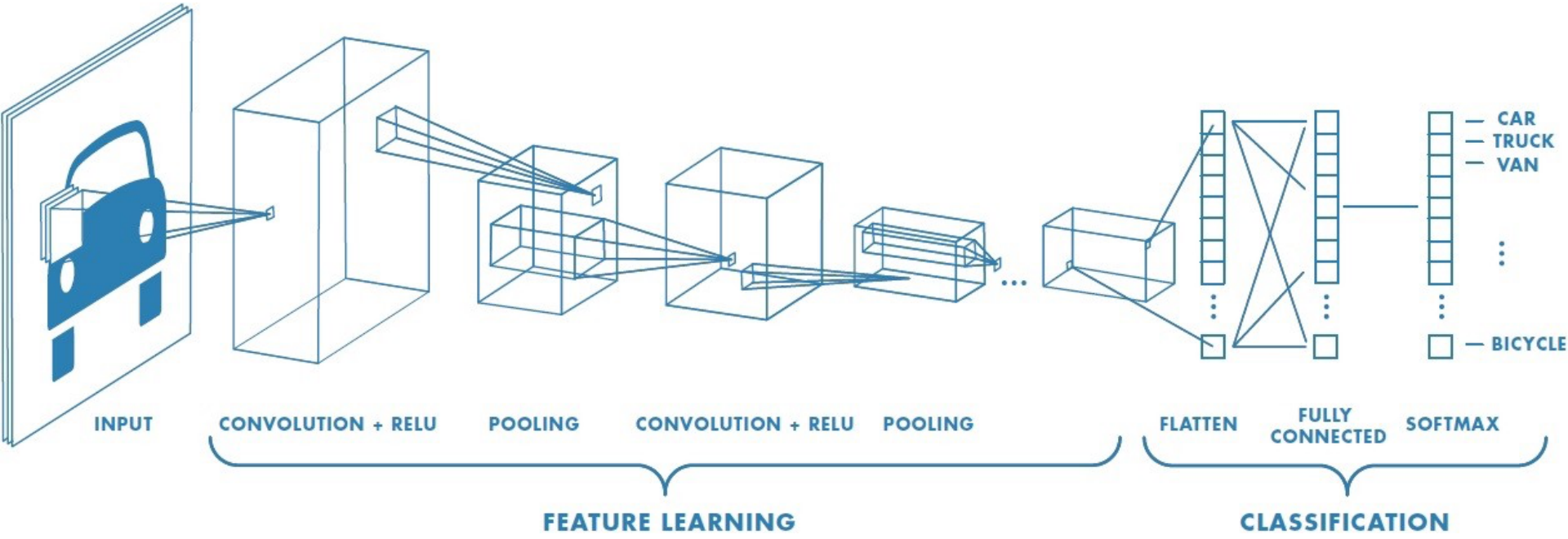
Max-pooling

max pool with 2x2 filters and stride 2

6	8
3	4

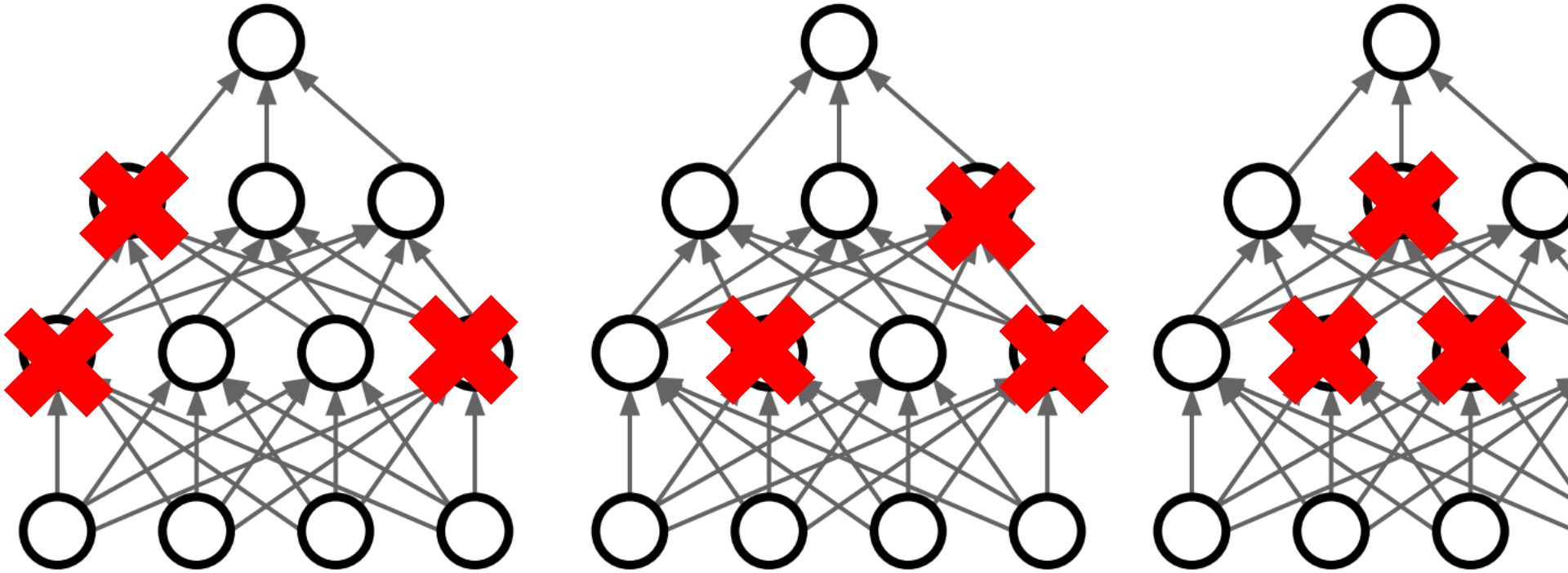
Typical architecture of CNNs

- Multiple layers of convolution + pooling, followed by dense/fully-connected layer



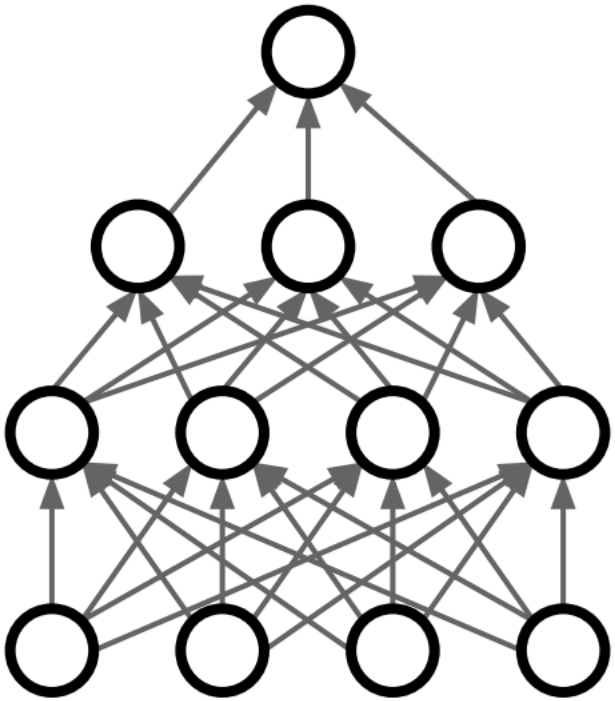
Regularization in Neural Nets

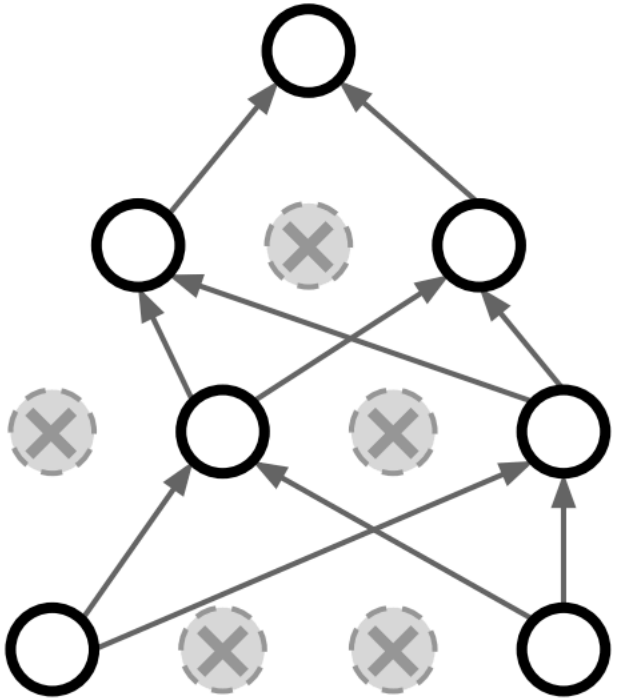
- Weight sharing in an ensemble
  - Dropout – train multiple networks, each with some neurons missing
  - The edges/weights are kept the same across networks



Regularization in Neural Nets

- Weight sharing in an ensemble
  - Dropout – in practice:
    - Train a single network
    - In each iteration, drop some random neurons
      - Multiply their output by zero





Batch normalization

- A neural network with ReLU activation is essentially:
  - $Y=\text{ReLU}(\ W_3\,\text{ReLU}(\ W_2\,\text{ReLU}(\ W_1X\ )\ )\ )$
- We see terms like $w_{3ij}*w_{2kl}*w_{1mn}*x$
- The derivative over w_{1mn} is $w_{3ij}*w_{2kl}*x$
- The derivative can quickly get large even if individual w's are not that much larger than 1
- Or can get very small if individual w's are close to zero

Batch normalization

- A neural network with ReLU activation is essentially:
  - $Y=\text{ReLU}(\ W_3\,\text{ReLU}(\ W_2\,\text{ReLU}(\ W_1X\ )\ )\ )$
- We see terms like $w_{3ij}*w_{2kl}*w_{1mn}*x$
- The derivative over w_{2kl} is $w_{3ij}*\ w_{1mn}\ *x$
- The derivative can quickly get large even if individual w's are not that much larger than 1
- Or can get very small if individual w's are close to zero
- Vanishing/exploding gradient!

Batch normalization

- The derivative over w_{2kl} is w_{3ij}* w_{1mn} *x
  ~ w_{3ij}*activation(prev.layer)
- Solution: make activations "just right" – not to large, not too small
  - E.g. make them roughly follow a Gaussian with 0-mean, unit norm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.