Carnegie Mellon University

Introduction to Deep Learning for Engineers

Spring 2025, Introduction to Deep Learning for Engineers
Jan 28, 2025, Fifth Session

Amir Barati Farimani
Associate Professor of Mechanical Engineering and Bio-Engineering
Carnegie Mellon University

Story so far:

MLP
Universal Function Approximation
Empirical Risk Minimization
Neural Networks Ingredients
Optimization and Backpropogation

Problem Setup: Things to define

- Given a training set of input-output pairs

$(X_1, d_1), (X_2, d_2), \ldots, (X_r, d_r),$

- what are input-output pairs?

$$Loss(W) = \frac{1}{T} \sum_i div(f(X_i; W), d_i)$$

The individual neurons

- Individual neurons operate on a set of inputs and produce a single output assume a "layered" network for simplicity

-- Standard setup: A differentiable activation function applied to an affine combination of the input

$$y = f\left(\sum_i w_i x_i + b\right)$$

We will assume this unless otherwise specified

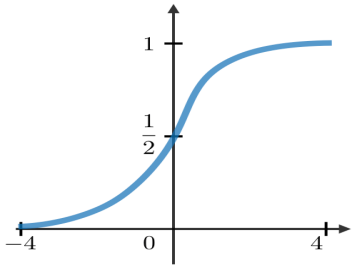
Parameters are weights w_i and bias b

-- More generally: aby differentiable function

$$y = f(x_1, x_2, \ldots, x_n; w)$$

Activation and their derivatives

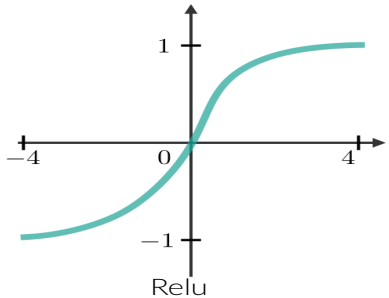
Sigmoid



$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$\acute{f}(z) = f(z)\big(1 - f(z)\big)$$

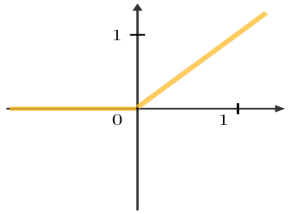
Tanh



$$f(z) = \tanh(z)$$

$$\acute{f}(z) = \big(1 - f^2(x)\big)$$

Relu



$$f(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$

$$\acute{f}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

Some popular activation functions and their derivatives

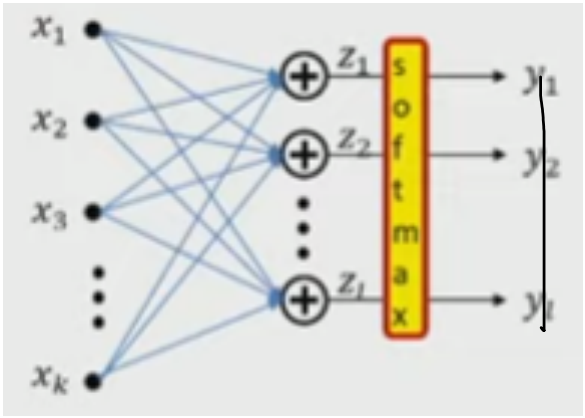
Vector activation example; Softmax

- Example: Softmax vector activation

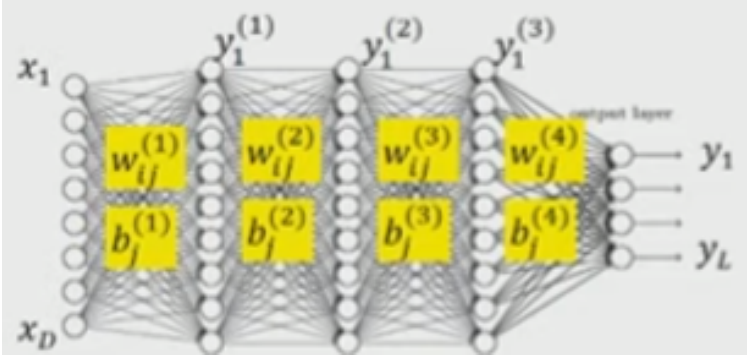
$$z_i = \sum_j w_{ji} x_j + b_i$$

Parameters are weights w_{ji} and bias b_i

$$y = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$



Notation



- The input layer is the layer
- We will represent the output of the i-th perceptron of the k^{th} layer as $y_i^{(k)}$.

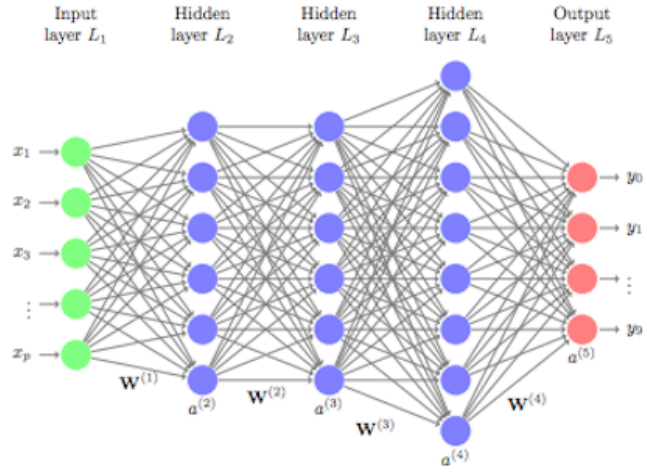
  -- Input to network: $y_i^{(0)} = x_i$

  -- Output of network: $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i-th unit of the k-1th layer and the jth unit of the k-th layer as $w_{ij}^{(k)}$

  -- The bias to the jth unit of the k-th layer is $b_j^{(k)}$

Vector notation

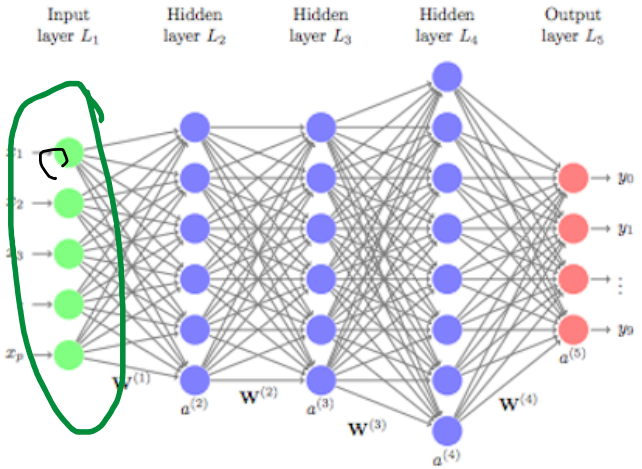
- Given a training set of input-output pairs (X_1, d_1), (X_2, d_2), …, (X_r, d_r)
- $X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]$ is the nth vector of actual outputs of the network
- We will sometimes drop the first subscript when referring to a specific instance



Carnegie Mellon University

Representing the Input

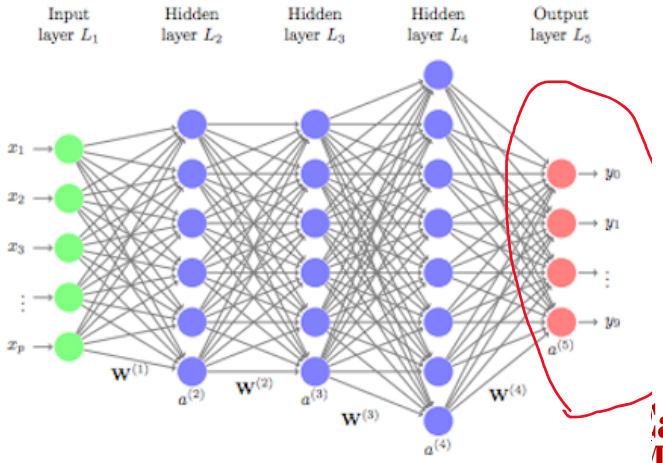
- Vectors of numbers
  -- (or may even be just a scalar, if input layer is of size 1)
  -- e.g. vector of pixel values
  -- e.g. vector of speech features
  -- e.g. real-valued vector representing text
    - We will see hoe this happens later in the course
  -- Other real valued vectors



Carnegie Mellon University

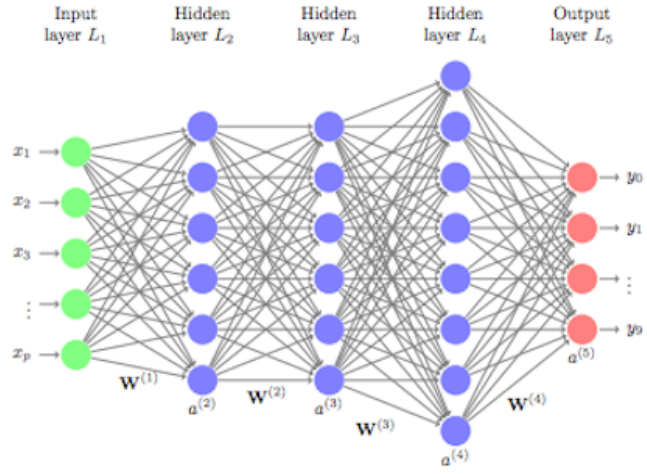
Representing the Output

- If the desired output is real-valued , no special tricks are necessary
  -- Scalar Output : single output neuron
    - d = scalar (real value)
  -- Vector Output : as many output neurons as the dimension of the desire output
    - $d = [d_1, d_2, d_l]$ (vector of real values)



Vector notation

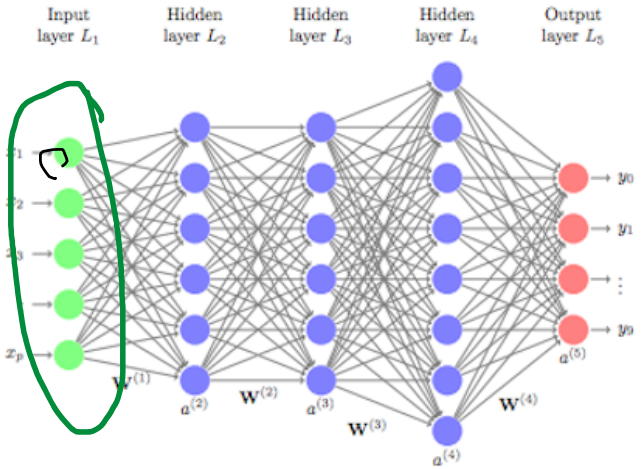
- Given a training set of input-output pairs (X_1, d_1), (X_2, d_2), …, (X_r, d_r)
- $X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]$ is the nth vector of actual outputs of the network
- We will sometimes drop the first subscript when referring to a specific instance



Carnegie Mellon University

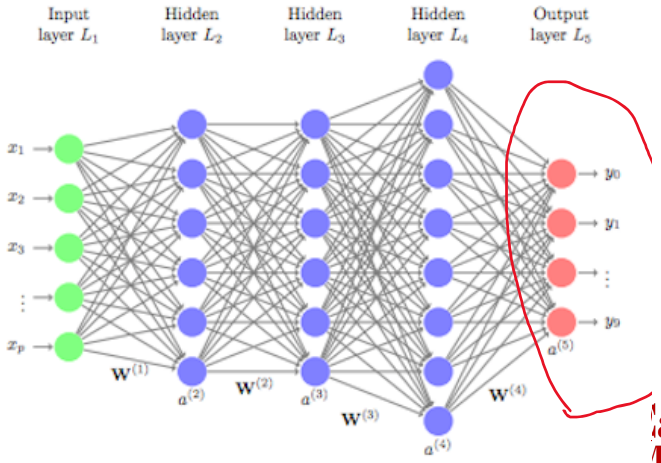
Representing the Input

- Vectors of numbers
  -- (or may even be just a scalar, if input layer is of size 1)
  -- e.g. vector of pixel values
  -- e.g. vector of speech features
  -- e.g. real-valued vector representing text
    - We will see hoe this happens later in the course
  -- Other real valued vectors



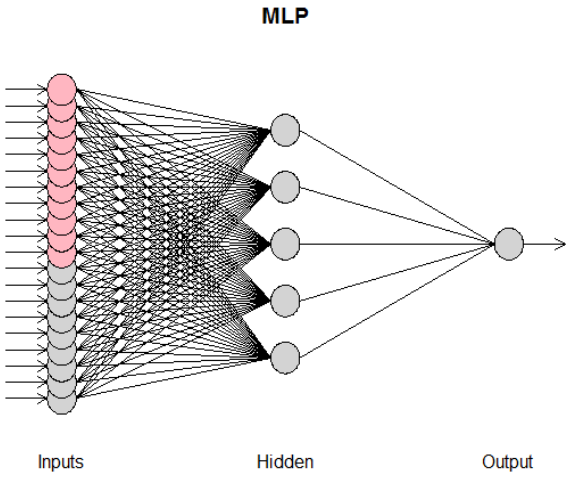
Representing the Output

- If the desired output is real-valued , no special tricks are necessary
  -- Scalar Output : single output neuron
    - d = scalar (real value)
  -- Vector Output : as many output neurons as the dimension of the desire output
    - $d = [d_1, d_2, d_l]$ (vector of real values)



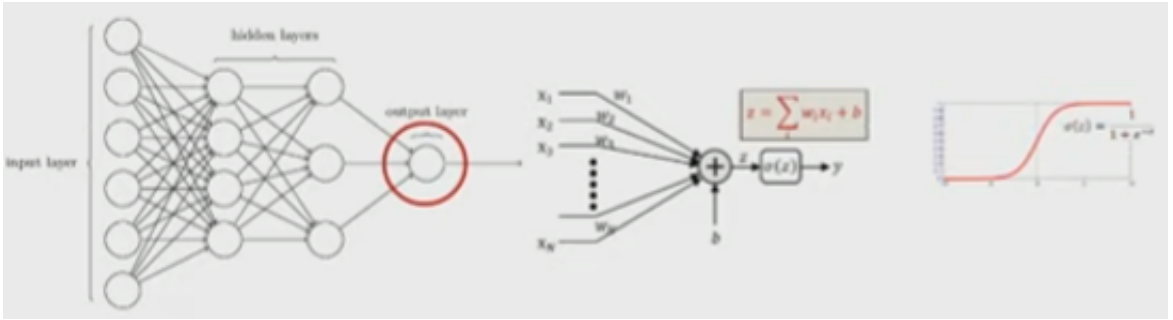
Representing the Output

- If the desired output is binary (is this a cat or not),use a simple 1/0 representation of the desired output
  -- 1 = Yes, it's a cat
  -- 0 = No, it's not a cat



Representing the Output

- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
- Output activation: Typically, a sigmoid
  -- Viewed as the probability P(Y = 1/X) of class value 1
  - Indicating the fact that for actual data, in general a feature value X may occur for both classes, but with different probabilities
  - Is differentiable



Multi-class output: One-hot representations

- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector:

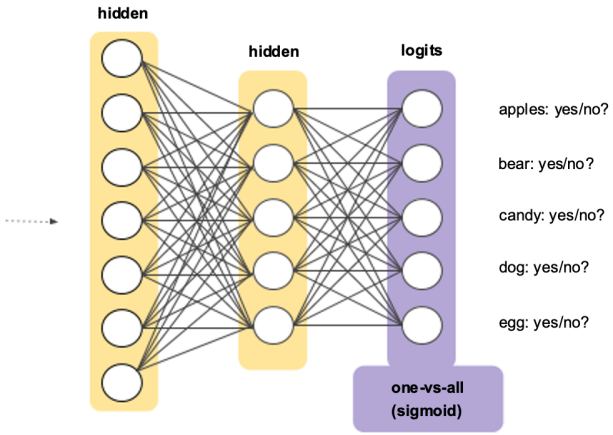
$$[cat\ \ dog\ \ camel\ hat\ \ flower]^T$$

- For inputs of each of the five classes the desired output is:

  cat: $[1\ 0\ 0\ 0\ 0]^T$
  dog: $[0\ 1\ 0\ 0\ 0]^T$
  camel: $[0\ 0\ 1\ 0\ 0]^T$
  hat: $[0\ 0\ 0\ 1\ 0]^T$
  flower: $[0\ 0\ 0\ 0\ 1]^T$

- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class
- This is a one hot vector

Multi-class network



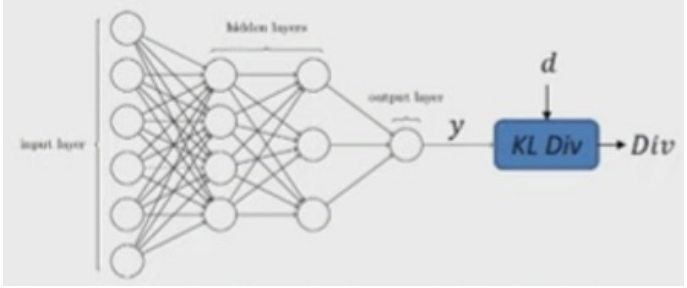
- For a multi-class classifier with N classes, the one-hot representation will have N binary outputs
  -- An N-dimensional binary vector
- The neural network's output too must ideally be binary (N-1 zeros and a single 1 in the right place)
- More realistically, it will be a probability vector
  -- N probability values that sum to 1

For binary classifier

- For binary classifier with scalar output, $Y \in (0,1)$, d is 0/1, the cross entropy between the probability distribution $[Y, 1-Y]$ and the ideal output probability $[d, 1-d]$ is popular

$$Div(Y, d) = -\mathrm{d}\log \mathrm{Y} - (1 - \mathrm{d})\log(1 - \mathrm{Y})$$

-- Minimum when d = Y



- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} \frac{-1}{Y} \; if \; d = 1 \\ \frac{1}{1-Y} \; if \; d = 0 \end{cases}$$

Note: when y=d the derivative is not 0
Even though div()=0 (minimum) when y=d

Cross entropy vs L2

$$Div(Y, d) = -\mathrm{d}\log \mathrm{Y} - (1 - \mathrm{d})\log(1 - \mathrm{Y})$$

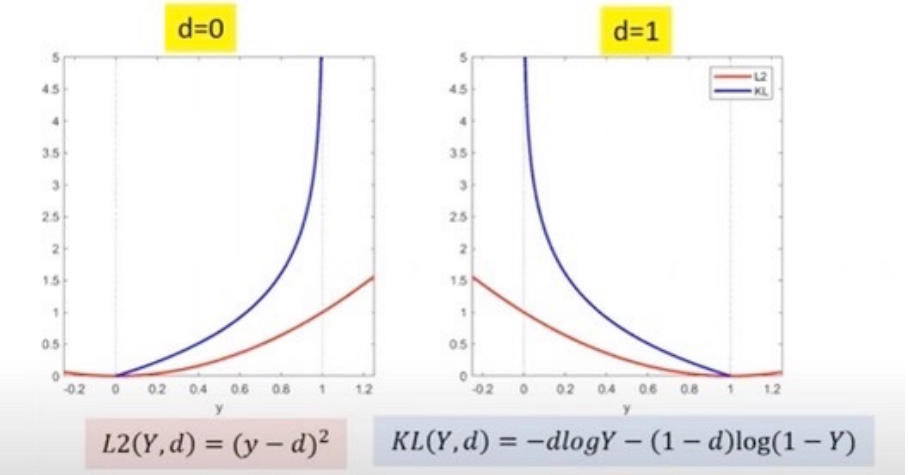
-- Minimum when d = Y

- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} \frac{-1}{Y} \; if \; d = 1 \\ \frac{1}{1-Y} \; if \; d = 0 \end{cases}$$

$$Div(Y, d) = \frac{1}{2}\|Y - d\|^2 = \frac{1}{2}\sum_i (y_i - d_i)^2$$

$$\frac{dDiv(Y, d)}{dy_i} = (y_i - d_i)$$



Note: when y=d the derivative is not 0
Even though div()=0 (minimum) when y=d

For multi-class classification

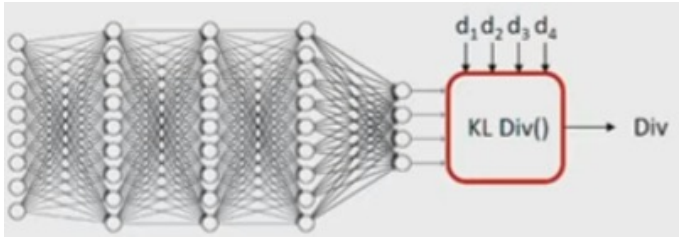
- Desired output d is a one hot vector [0 0 0… 1 …0 0 0] with the 1 in the c-th position {for class c}
- Actual output will be probability distribution $[y_1, y_2, \ldots]$
- The cross-entropy between the desired one-hot output and actual output:

$$Div(Y,d) = -\sum_i d_i \log y_i = -\log y_c$$

- Derivative

$$\frac{dDiv(Y,d)}{dY_i} = \begin{cases} \frac{-1}{y_c} & for\ the\ \mathrm{c}-th\ component \\ 0 & for\ remaining\ component \end{cases}$$

$$\nabla_Y Div(Y,d) = \left[0\ 0\ \ldots\ \frac{-1}{y_c}\ \ldots 0\ 0\right]$$

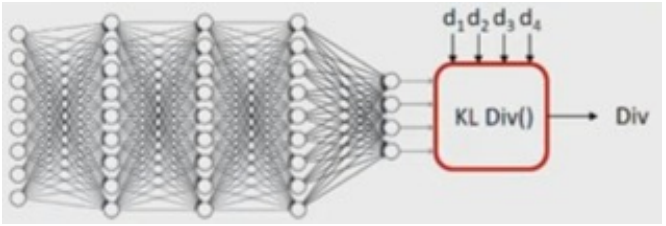


If y_c <1, the slop is negative w.r.t. y_c
Indicates increasing y_c will reduce
Note: when y=d the derivative is not 0
Even though div()=0 (minimum) when y=d

For multi-class classification

- It is sometimes useful to set the target output to [ϵ ϵ … (1-(K-1) ϵ) … ϵ ϵ ϵ] with the value 1-(K-1) ϵ in the c-th position (for class c) and ϵ elsewhere for some small ϵ
  -- "Label smoothing" aids gradient descent

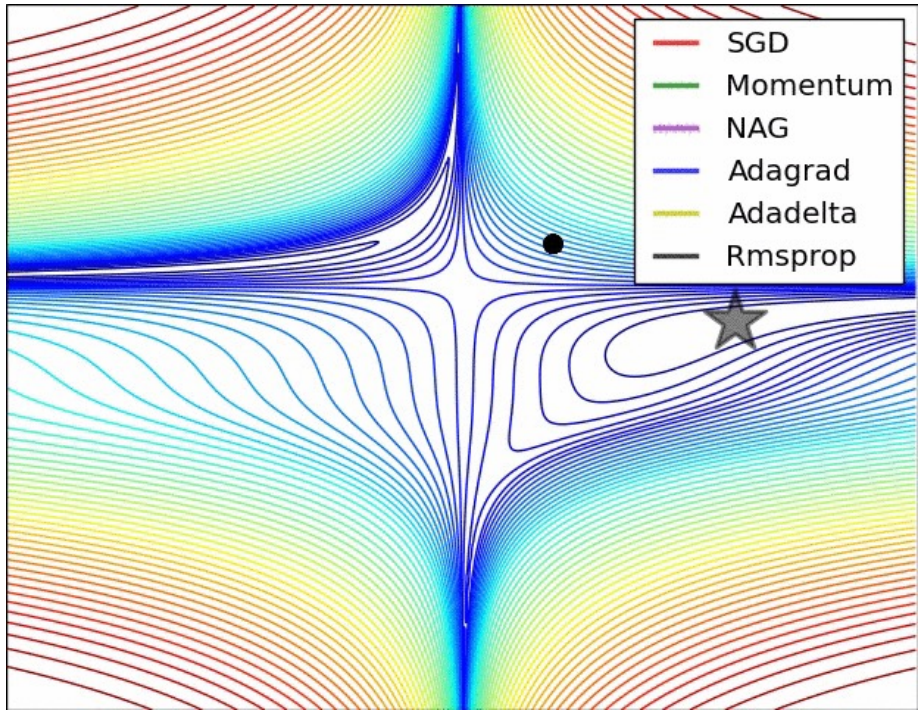
The cross-entropy remains: $Div(Y, d) = -\sum_i d_i \log y_i$

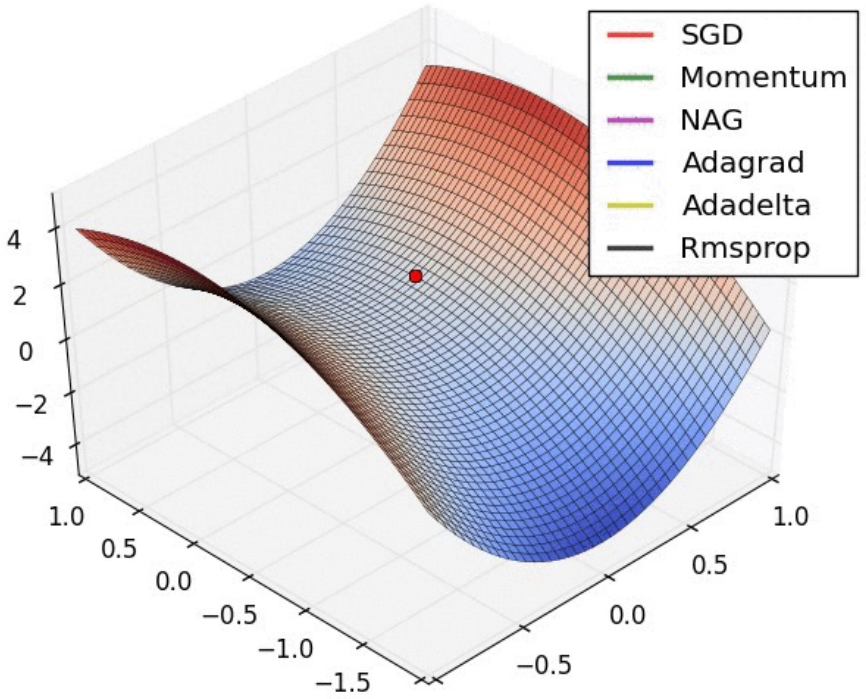


- Derivative

$$\frac{dDiv(Y, d)}{dY_i} = \begin{cases} -\frac{1-(K-1)\epsilon}{y_c} & for\ the\ c-th\ component \\ -\frac{\epsilon}{y_i} & for\ remaining\ component \end{cases}$$

Race of the Optimizers!

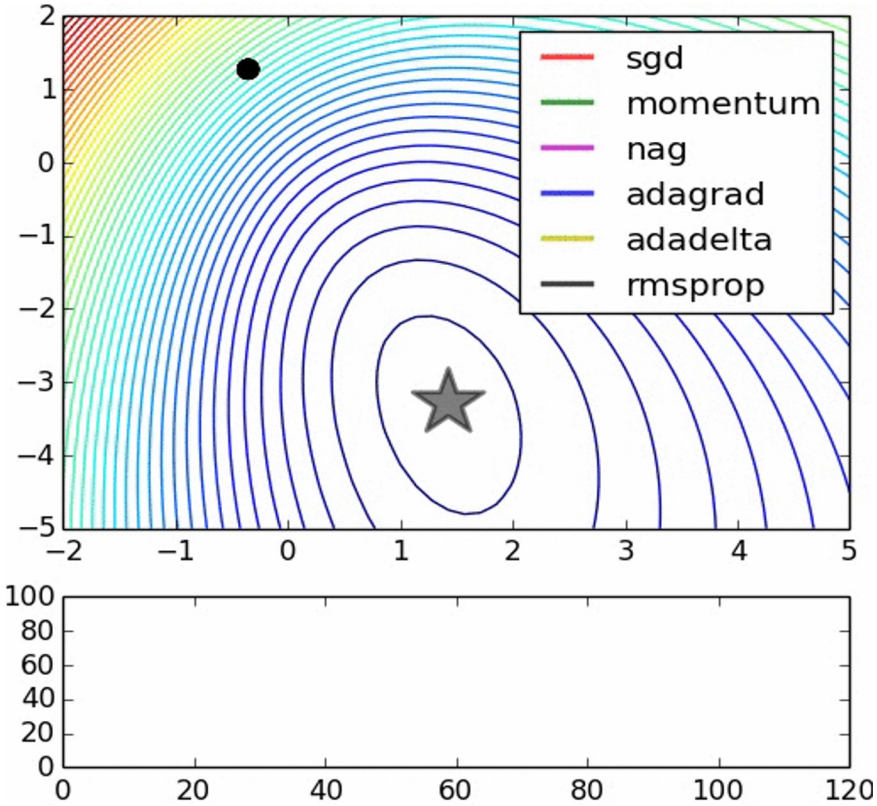




http://cs231n.github.io/neural-networks-3/#hyper

Carnegie Mellon University

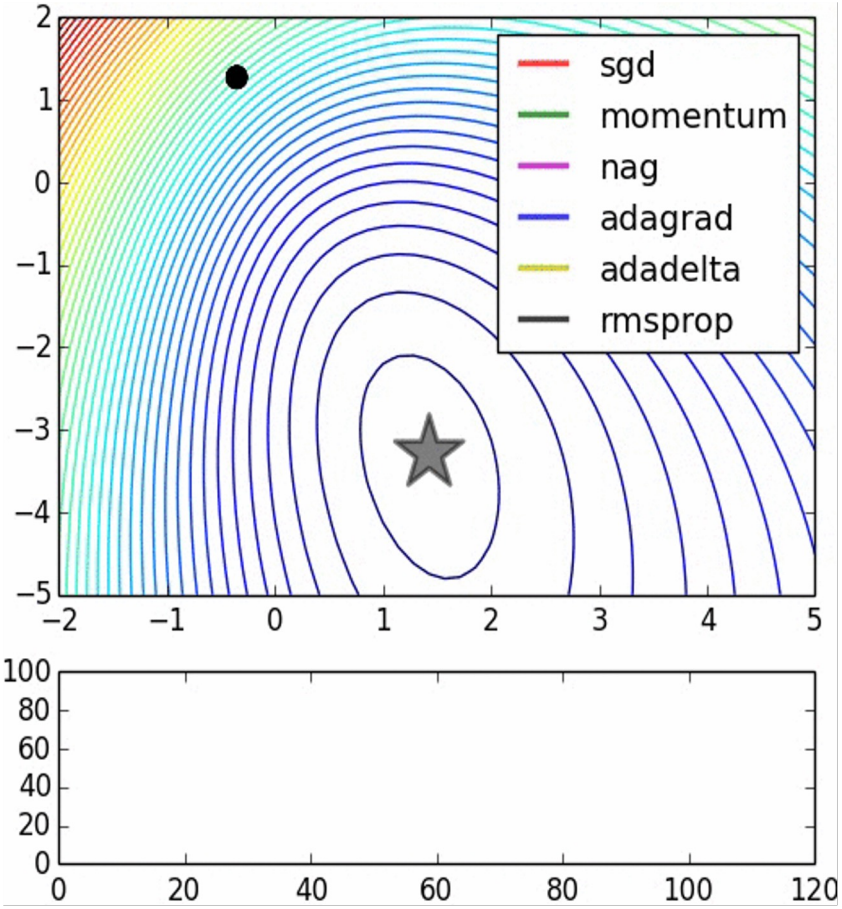
Recap


sgd
momentum
nag
adagrad
adadelta
rmsprop
2
1
0
−1
−2
−3
−4
−5
−2
−1
0
1
2
3
4
5
100
80
60
40
20
0
0
20
40
60
80
100
120

Summary

- **Simple Gradient Methods** like SGD can make adequate progress to an optimum when used on minibatches of data.
- **Second-order** methods make much better progress toward the goal, but are more expensive and unstable.
- **Convergence rates:** quadratic, linear, O(1/n).
- **Momentum:** is another method to produce better effective gradients.
- ADAGRAD, RMSprop diagonally scale the gradient. ADAM scales and applies momentum.

Key points



Heuristic methods

DROP OUT

WEIGHT REGULARIZATION

WEIGHT INITIALIZATION

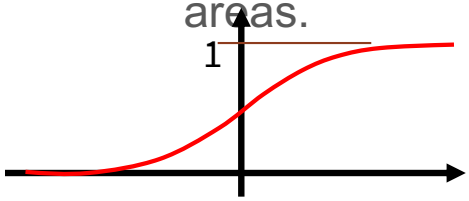
DATA AUGMENTATION

DATA NORMALIZATION

Weight Initialization

USUALLY SMALL RANDOM VALUES.

Try to choose so that typical input to a neuron avoids saturating / non-differentiable areas.



Occasionally inspect units for saturation / blowup.

Larger values may give faster convergence, but worse models!

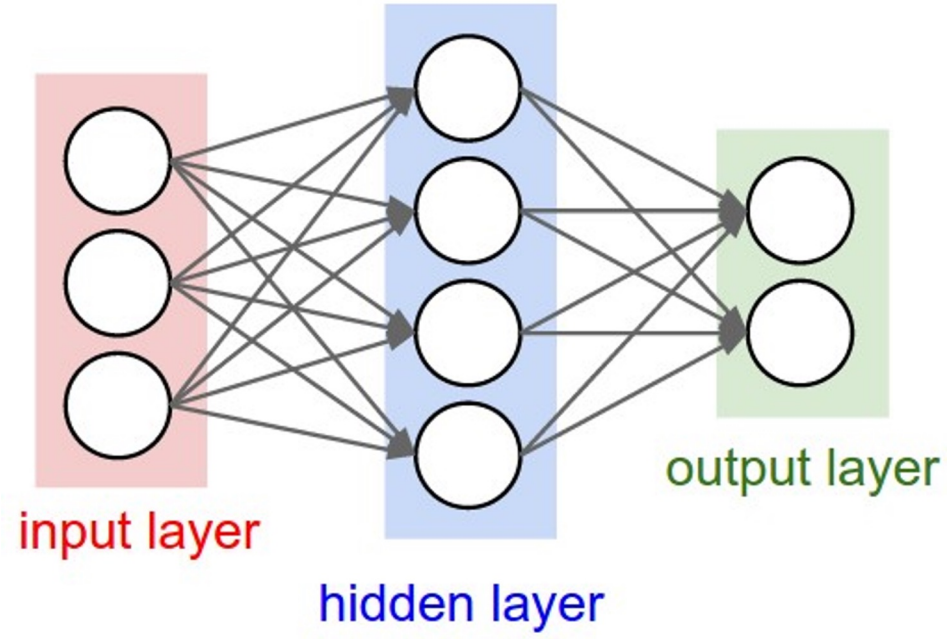
See [Glorot et al., AISTATS 2010]

$$r = \sqrt{6/(\text{fan-in} + \text{fan-out})}$$

Heuristic methods to train faster

Weight Initialization
Weight Regularization
Drop Out
Data Augmentation
Data Normalization

Q: what happens when W=0 init is used?



Q: what happens when W=0 init is used?

What do you notice about the gradients and weights when the initialization method is zero?

Initializing all the weights with zeros leads the neurons to learn the same features during training.

If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.

So they can never learn to be different features.
We break symmetry by initializing the weights to have small random values.

Q: what happens when W=0 init is used?

Case 1: A too-large initialization leads to exploding gradients

The output values increase exponentially in forward propagation. When these activations are used in backward propagation, this leads to the exploding gradient problem. That is, the gradients of the cost with the respect to the parameters are too big. This leads the cost to oscillate around its minimum value.

$w^{15}=1.6^{15}=281$

Carnegie Mellon University

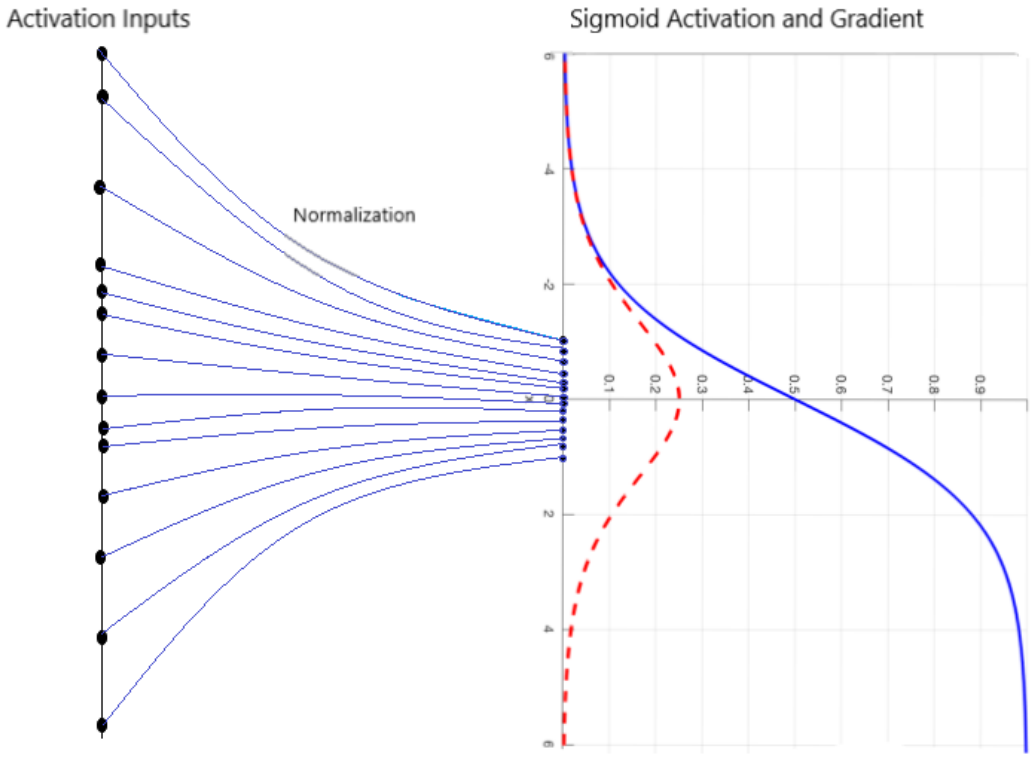
Q: what happens when W=0 init is used?

Case 2: A too-small initialization leads to vanishing gradients

the values of the activation decrease exponentially with number of layers

$w^3=0.6^3=0.21$, $w^{15}=0.6^{15}=0.00047$

Weight Initialization



First idea: **Small random numbers**

```
W = np.random.normal(0, 0.01, (N,D))
```

(gaussian with zero mean and 0.01 standard deviation)

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

How to find appropriate initialization values

To prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:

The *mean* of the activations should be zero.

The *variance* of the activations should stay the same across every layer.

Under these two assumptions, the backpropagated gradient signal should not be multiplied by values too small or too large in any layer. It should travel to the input layer without exploding or vanishing.

Derivation of Xavier Initialization

$$Var(a_i^{[\ell-1]}) = Var(a_i^{[\ell]})$$

$$= Var(z_i^{[\ell]})$$ ← linearity of **tanh** around zero $tanh(z) \approx z$

$$= Var(\sum_{j=1}^{n^{[\ell-1]}} w_{ij}^{[\ell]} a_j^{[\ell-1]})$$

$$= \sum_{j=1}^{n^{[\ell-1]}} Var(w_{ij}^{[\ell]} a_j^{[\ell-1]})$$ ← variance of independent sum $Var(X+Y) = Var(X) + Var(Y)$

$$= \sum_{j=1}^{n^{[\ell-1]}} E[w_{ij}^{[\ell]}]^2 Var(a_j^{[\ell-1]}) + E[a_j^{[\ell-1]}]^2 Var(w_{ij}^{[\ell]}) + Var(w_{ij}^{[\ell]}) Var(a_j^{[\ell-1]})$$ ← variance of independent product $Var(XY) = E[X]^2 Var(Y) + E[Y]^2 Var(X) + Var(X)Var(Y)$

$$= n^{[\ell-1]} Var(w_{ij}^{[\ell]}) Var(a_j^{[\ell-1]}) \implies Var(W) = \frac{1}{n^{[\ell-1]}}$$

Carnegie Mellon University

Second idea: **Xavier Initialization**

Problems with choosing the initial weights:

- If they are too small, the signal strength propagating in the network drops with each level until it becomes too small to be useful
- If they are too big, the signal strength propagating in the network grows with each level until it becomes too big to be useful

- Xavier initialization ensures that the weight have the right magnitude, keeping the signal strength in a reasonable interval.
- The initial weights come from a normal distribution of mean 0 and standard deviation given by the number of perceptrons in previous/next layer:

$$\mathrm{Var}(W) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

[Glorot and Bengio, 2010]

Carnegie Mellon University

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks
by Saxe et al, 2013

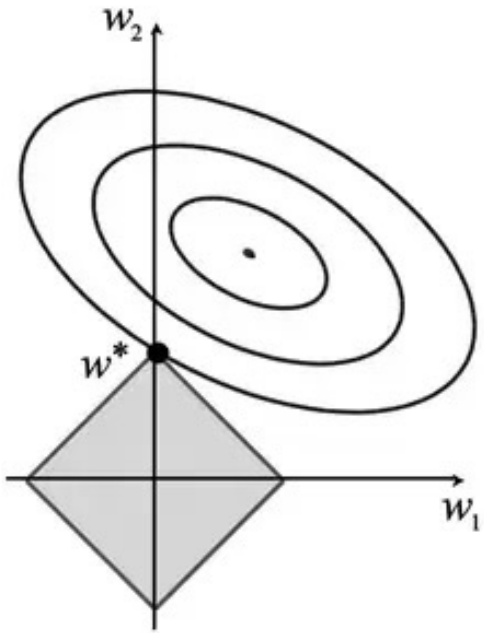
Random walk initialization for training very deep feedforward networks
by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

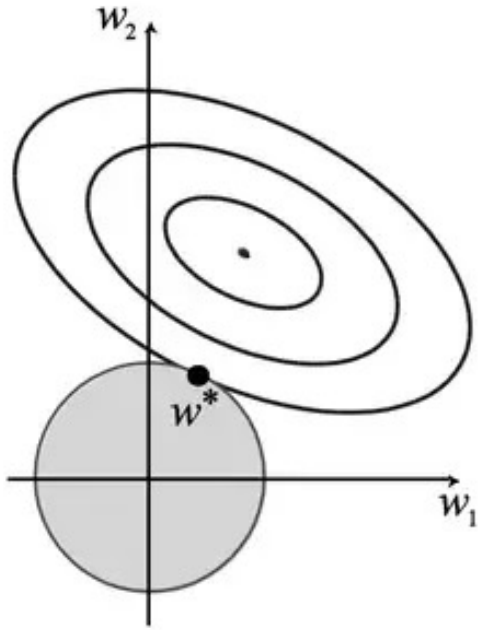
Data-dependent Initializations of Convolutional Neural Networks
by Krähenbühl et al., 2015

All you need is a good init
by Mishkin and Matas, 2015
…

Regularization



L1



L2

Regularization

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_t l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) + \lambda \Omega(\boldsymbol{\theta})$$

- L2 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)}\right)^2 = \sum_k ||\mathbf{W}^{(k)}||_F^2$$

- L1 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

Regularization

We know that L1 regularization encourages sparse weights (many zero values), and that L2 regularization encourages small weight values, but **why does this happen?**

Let's consider some cost function $J(w_1, \ldots, w_l)$, a function of weight matrices $w_1, \ldots, w_l$. Let's define the following two regularized cost functions:

$$J_{L_1}(w_1, \ldots, w_l) = J(w_1, \ldots, w_l) + \lambda \sum_{i=1}^{l} |w_i|$$

$$J_{L_2}(w_1, \ldots, w_l) = J(w_1, \ldots, w_l) + \lambda \sum_{i=1}^{l} ||w_i||^2$$

Let's derive the gradient updates for these cost functions.

The update for w_i when using J_{L_1} is:

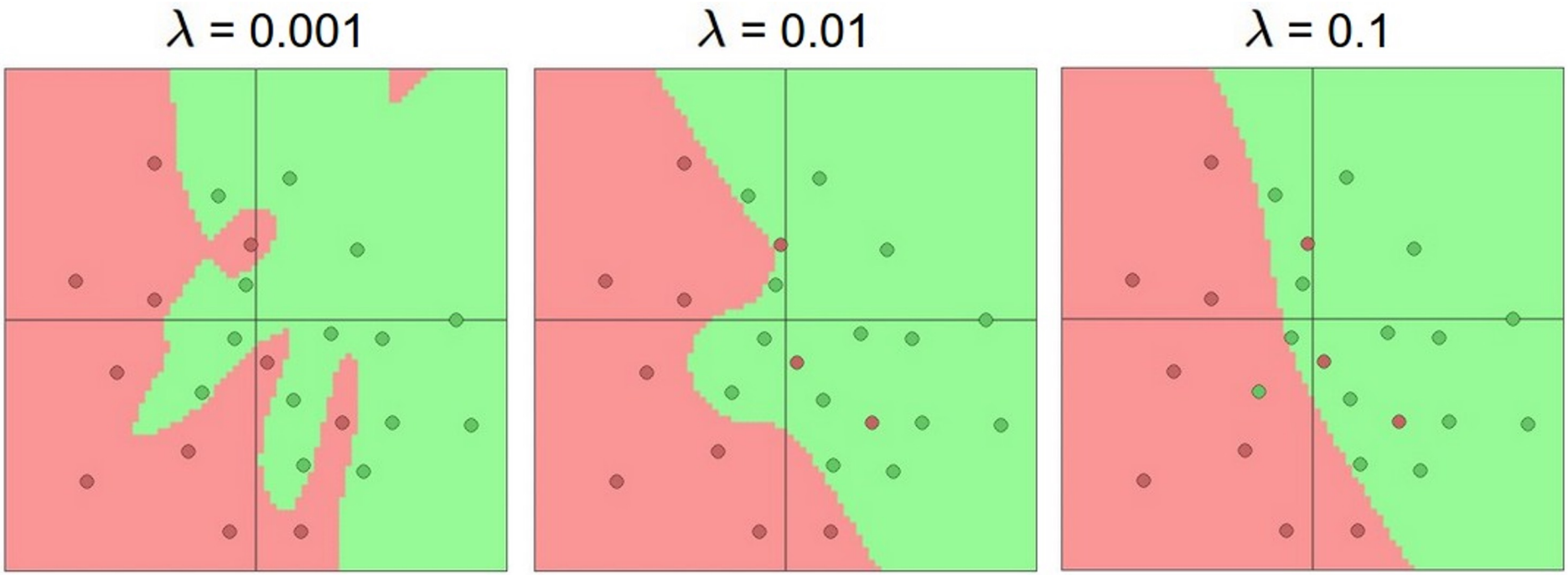
$$w_i^{k+1} = w_i^k - \underbrace{\alpha\lambda sign(w_i)}_{L_1 \text{ penalty}} - \alpha\frac{\partial J}{\partial w_i}$$

The update for w_i when using J_{L_2} is:

$$w_i^{k+1} = w_i^k - \underbrace{2\alpha\lambda w_i}_{L_2 \text{ penalty}} - \alpha\frac{\partial J}{\partial w_i}$$

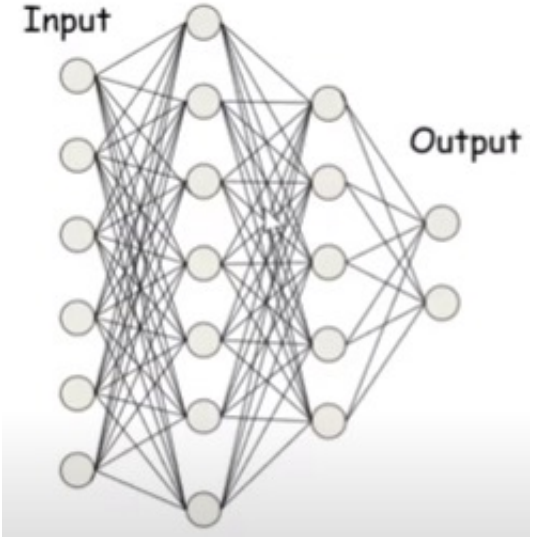
Carnegie Mellon University

Do not use size of neural network as a regularizer. Use stronger regularization instead:



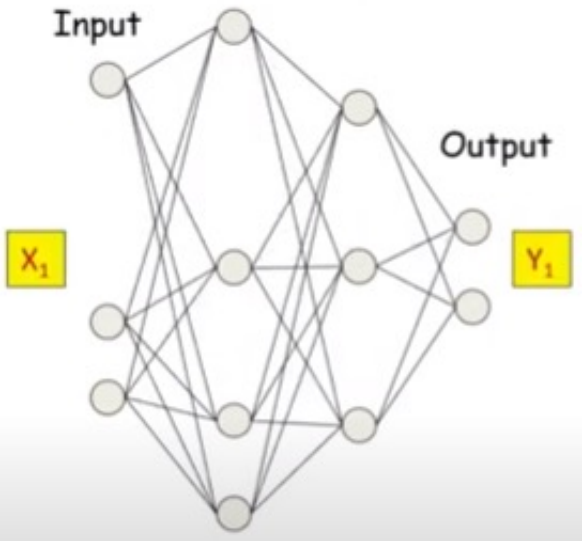
Practical advice: In general, it is better to use stronger regularization than reducing the model's capacity

Dropout



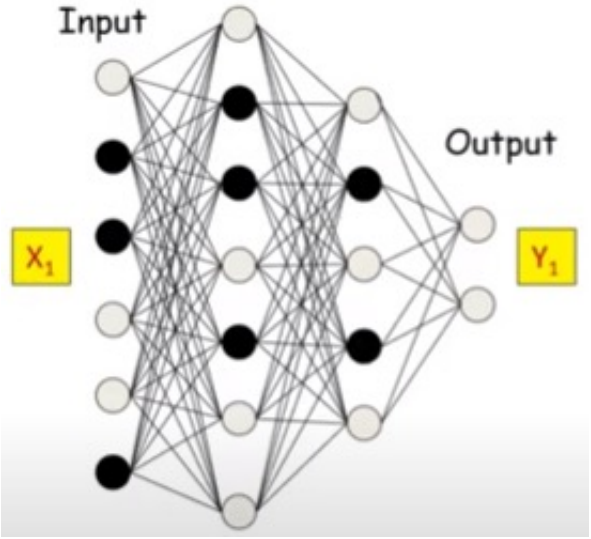
- During training: For each input, at each iteration, “turn off” each neuron with a probability 1-α

Dropout



- During training: For each input, at each iteration, “turn off” each neuron with a probability 1-α
  -- Also turn off inputs similarly

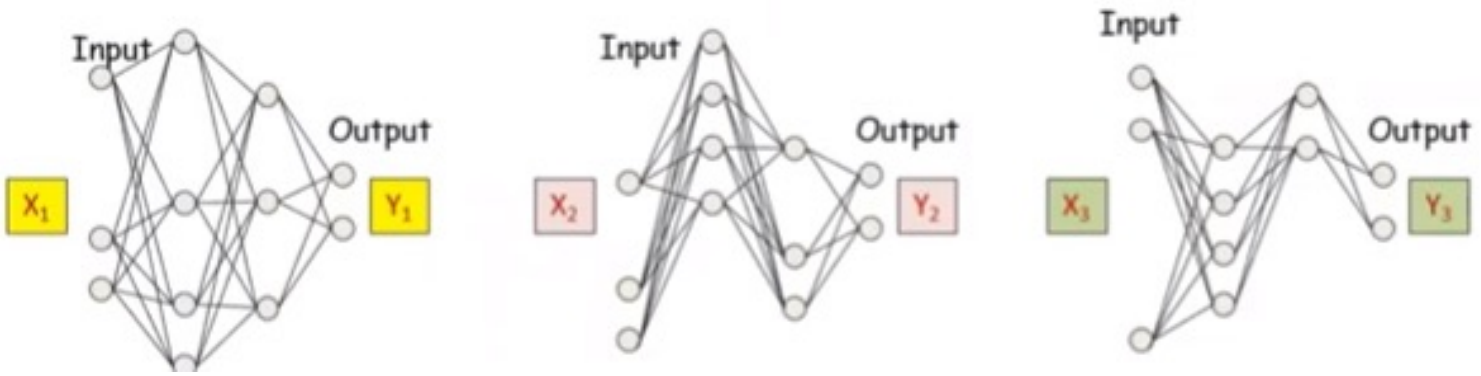
Dropout



- During training: For each input, at each iteration, "turn off" each neuron (including inputs) with a probability 1-α

-- In practice, set them to 0 according to the failure of a Bernoulli random number generator with success probability α

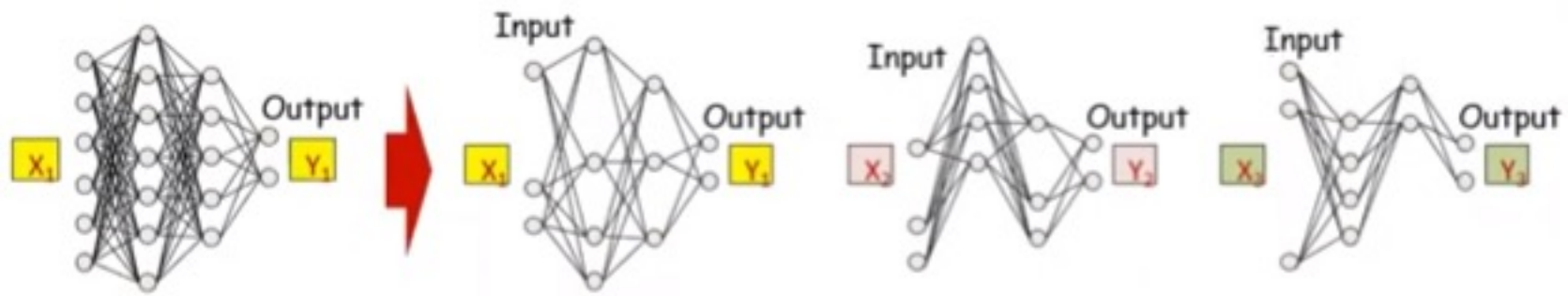
Dropout



The pattern of dropped nodes changes for each input i.e. in every pass through the net

- During training: Backpropagation is effectively performed only over the remaining network
  -- The effective network is different for different inputs
  -- Gradients are obtained only for the weights and biases *from* "On" nodes *to* "On" nodes
    - For the remaining the gradient is just 0

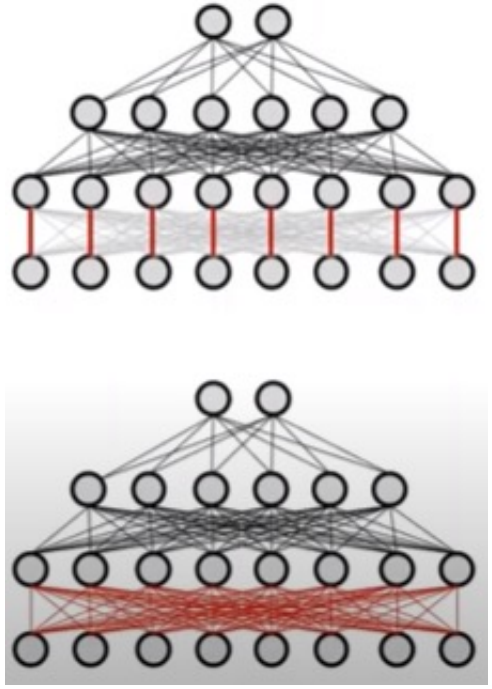
Statistical Interpretation



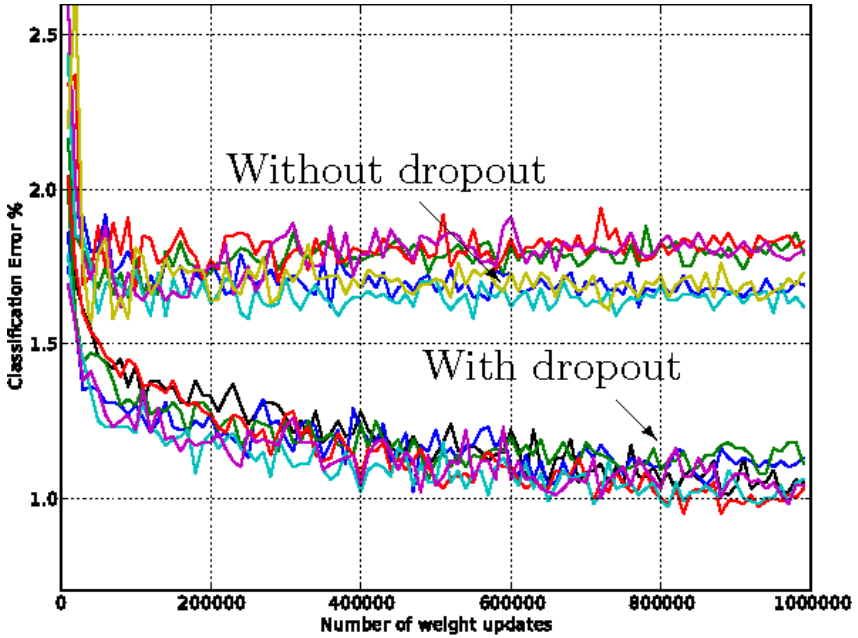
- For a network with a total of N neurons, there are 2^N possible sub-networks
  - Obtained by choosing different subsets of nodes
  - Dropout samples over all 2^N possible networks
  - Effectively learns a network that averages over all possible networks
    - Bagging

Dropout as a mechanism to increase pattern density

- Dropout forces the neurons to learn “rich” and redundant patterns
- E.g. without dropout, a non-compressive layer may just “clone” its input to its output
  -- Transferring the task of learning to the rest of the network upstream
- Dropout forces the neurons to learn denser patterns
  - With redundancy



Dropout effect



- From Srivastava et al., 2013. Test error for different architectures on MNIST with and without dropout
  - 2-4 hidden layers with 1024-208 units

Variations on dropout

- Zoneout: For RNNs
  -- Randomly chosen units remain unchanged across a time transition

- Dropconnect
  -- Drop individual connections, instead of nodes

- Shakeout
  -- Scale up the weights of randomly selected weights
    - $|w| \rightarrow \alpha|w| + (1-\alpha)c$

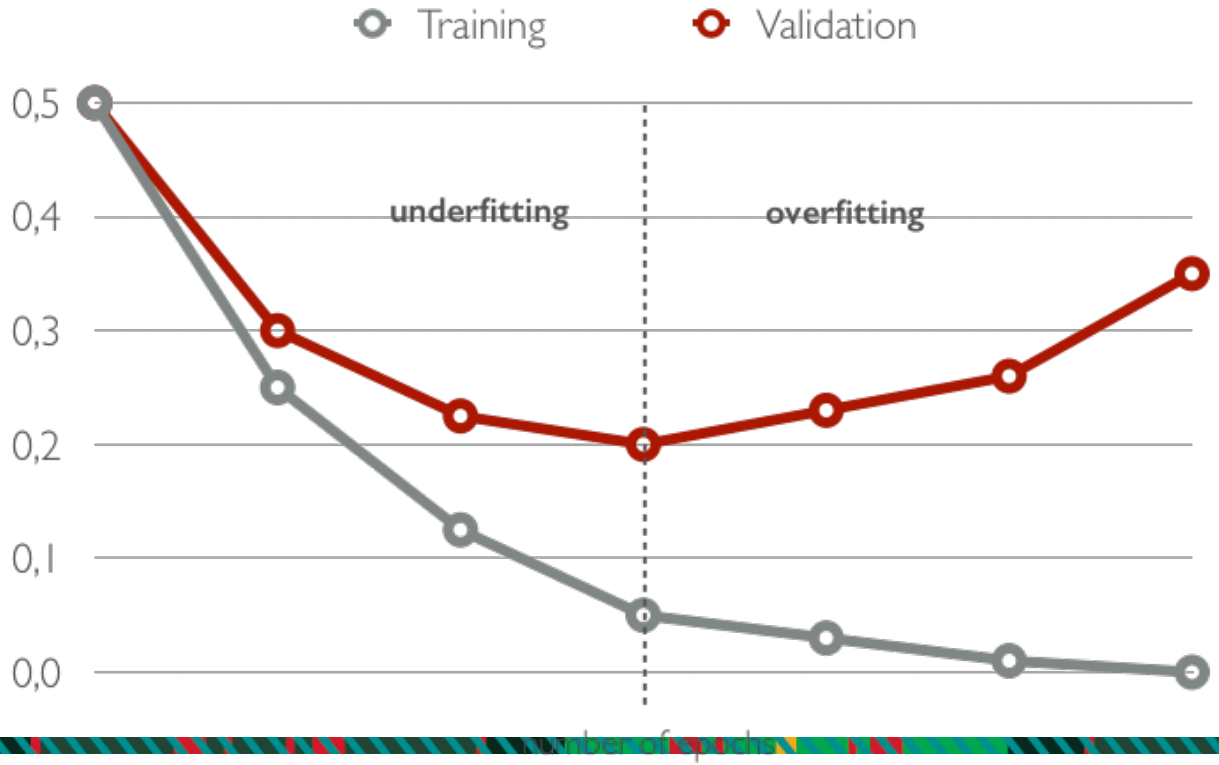
  -- Fix remaining weights to a negative constant
    - $w \rightarrow c$

- Whiteout
  -- Add or multiply weight-dependent Gaussian noise to the signal on each connection

Early Stopping

To select the number of epochs, stop training when validation set error increases

`Large Model can Overfit!

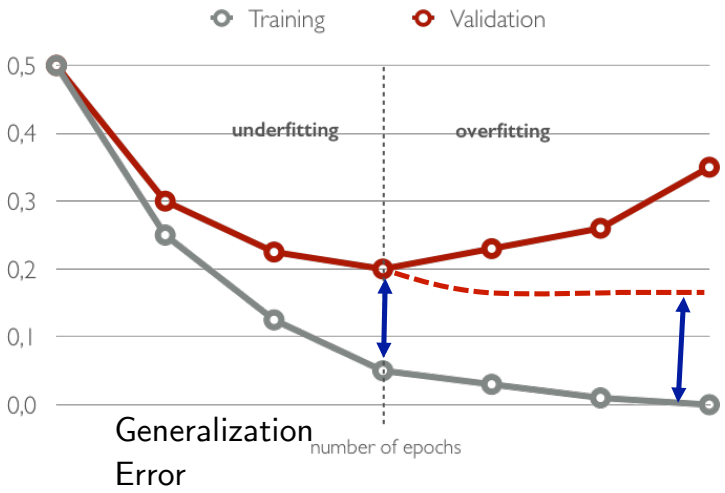


Early Stopping

To select the number of epochs, stop training when validation set error increases

`Large Model can Overfit!

- To select the number of epochs, stop training when validation set error increases → Large Model can Overfit

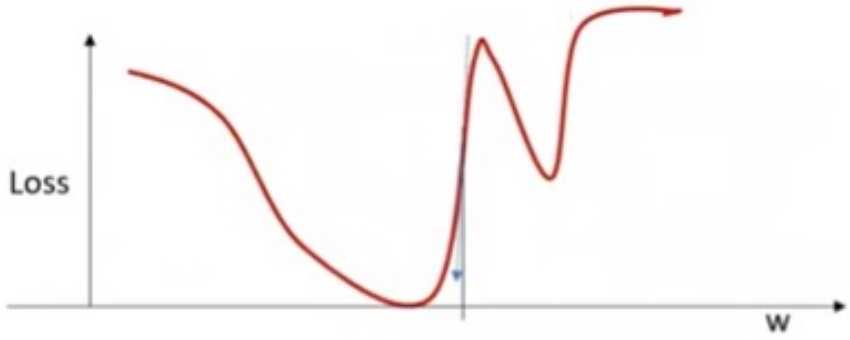


Implicit Regularization

- Optimization plays a crucial role in generalization
- Generalization ability is not controlled by network size but rather by some other implicit control

Behnam Neyshabur, PhD thesis 2017
Neyshabur et al., Survey Paper, 2017

Additional heuristics: Gradient clipping



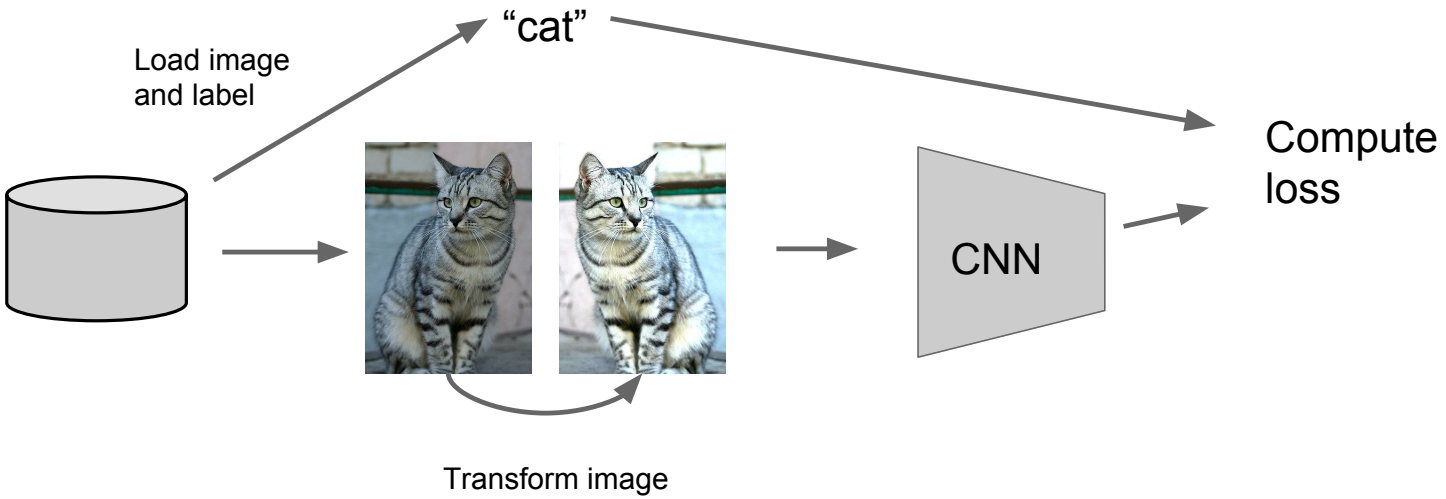
- Often the derivative will be too high
  -- When the divergence has a steep slope
  -- This can result in instability
- Gradient clipping: set a ceiling on derivative value

If $\partial_w D > \theta$ then $\partial_w D = \theta$

-- Typical θ value is 5

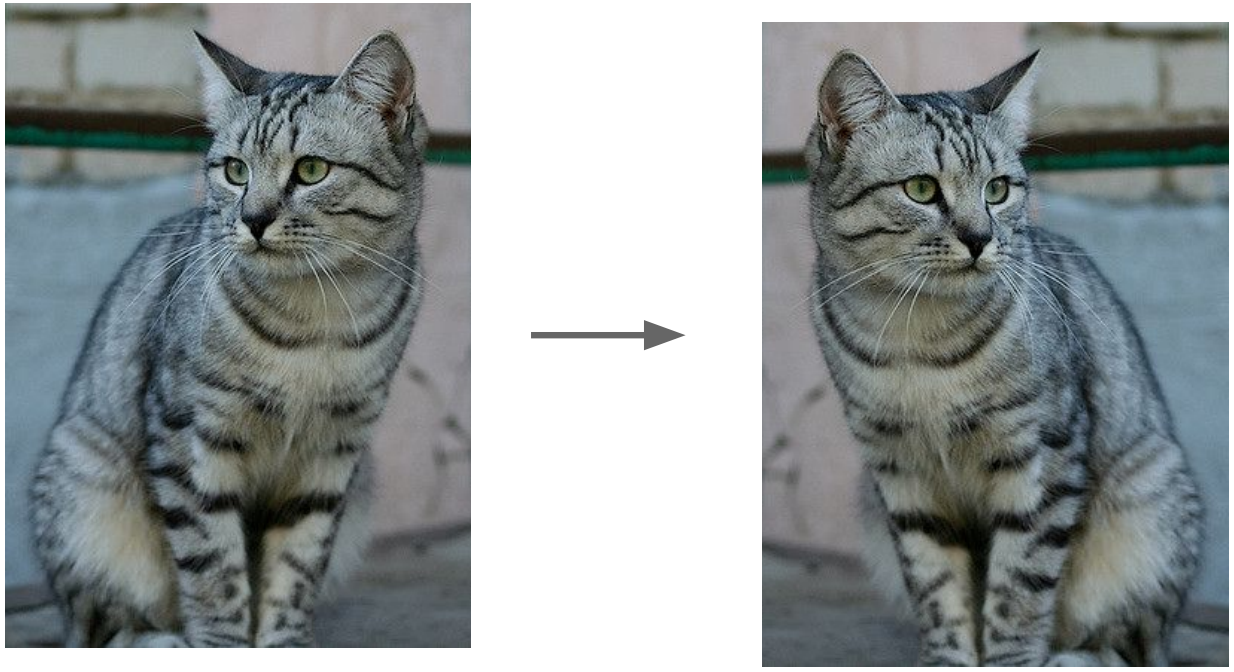
Data Augmentation

Regularization: Data Augmentation



Data Augmentation

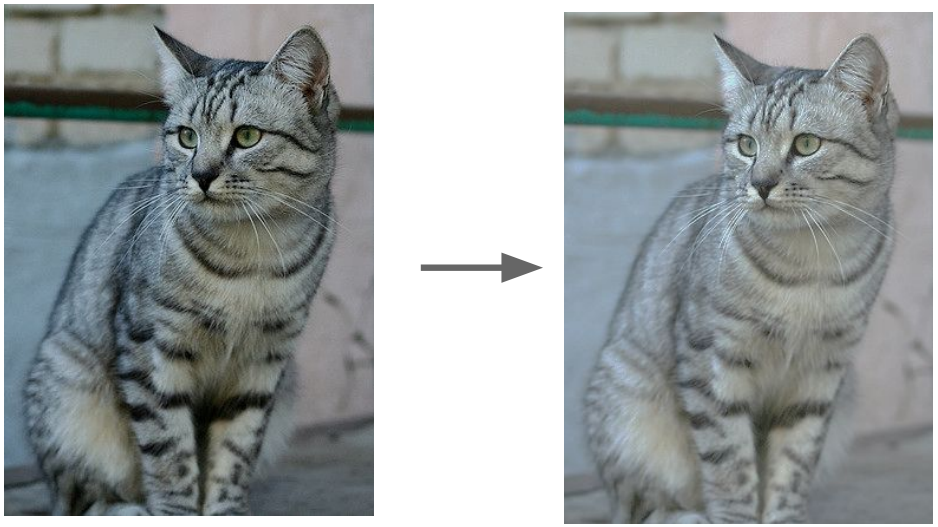
Horizontal Flips



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Learned linear transformation to adapt to non-linear activation function (γ and β are trained)!!

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

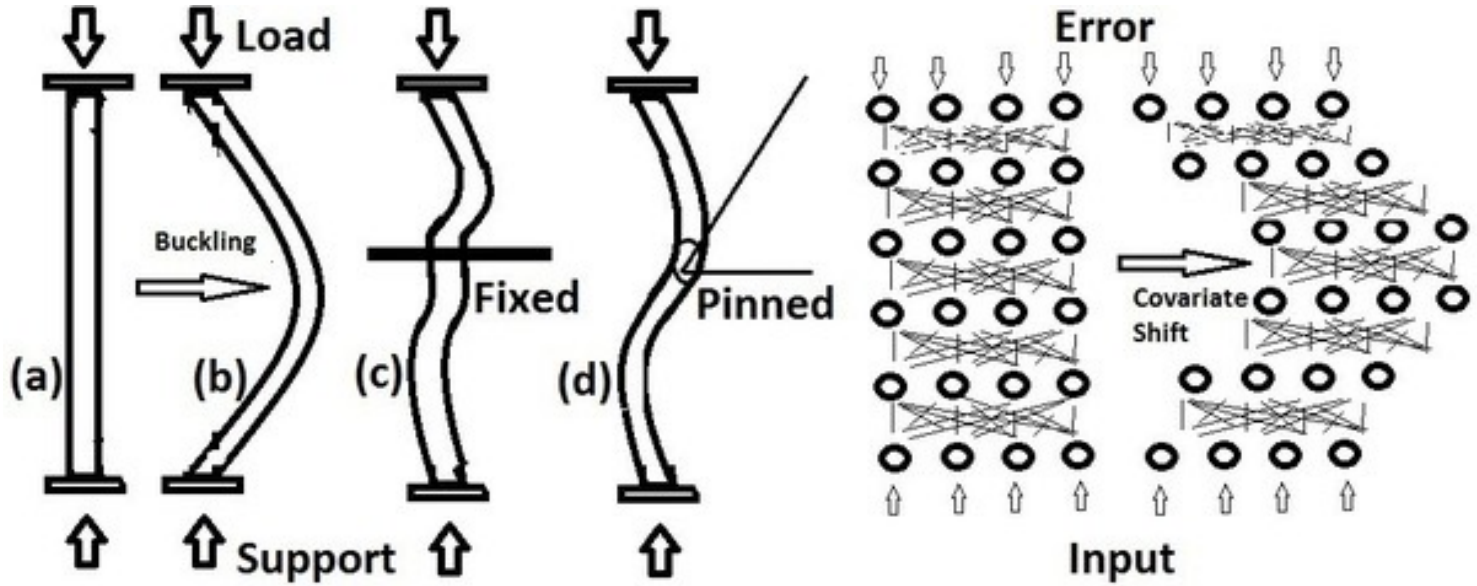
"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization



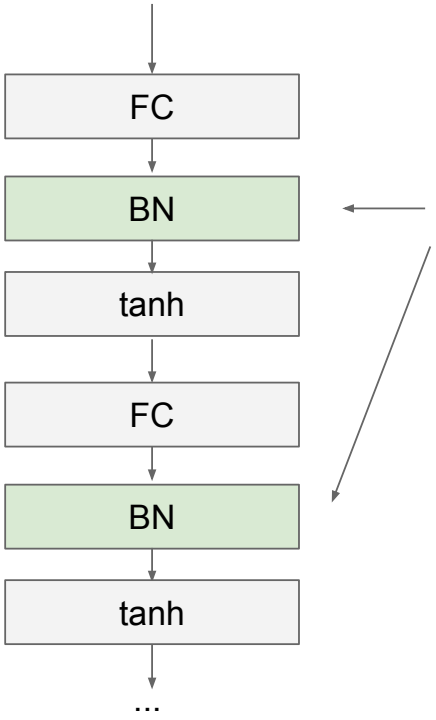
For both, Buckling or Co-Variate Shift a small perturbation leads to a large change in the later.

Debiprasad Ghosh, PhD, Uses AI in Mechanics

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 20

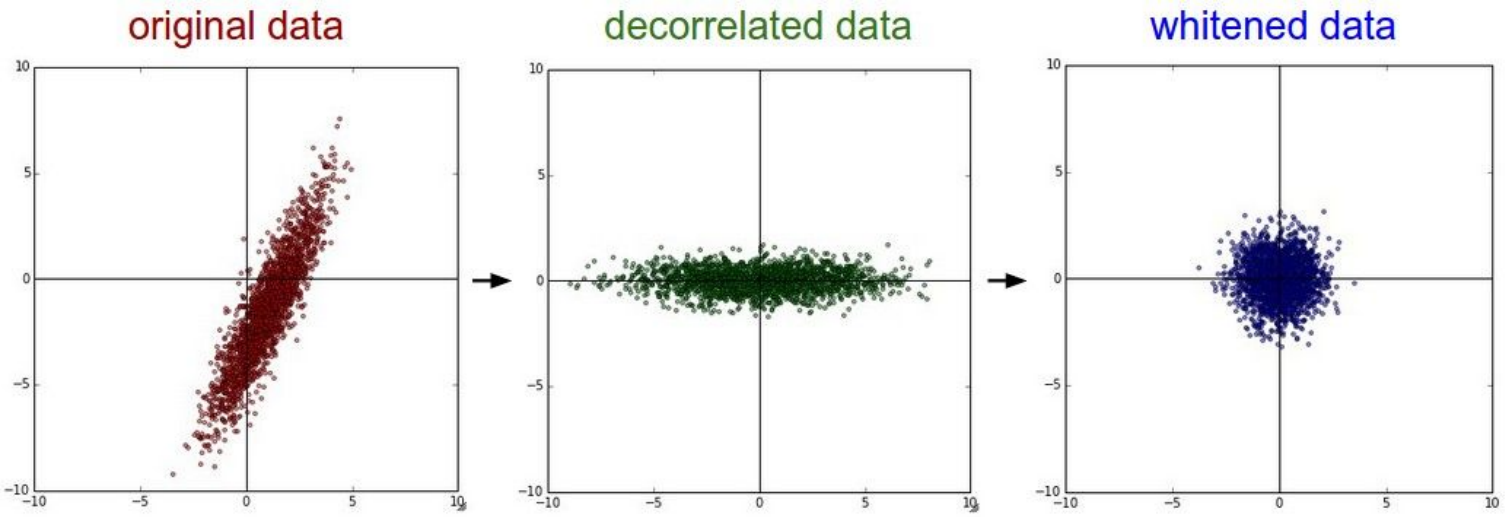


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Preprocessing

In practice, you may also see **PCA** and **Whitening** of the data

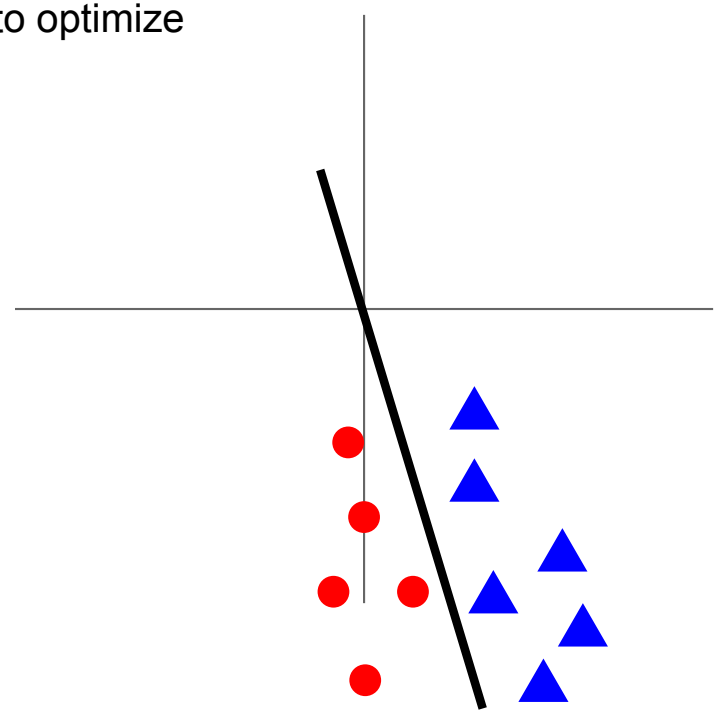


(data has diagonal covariance matrix)

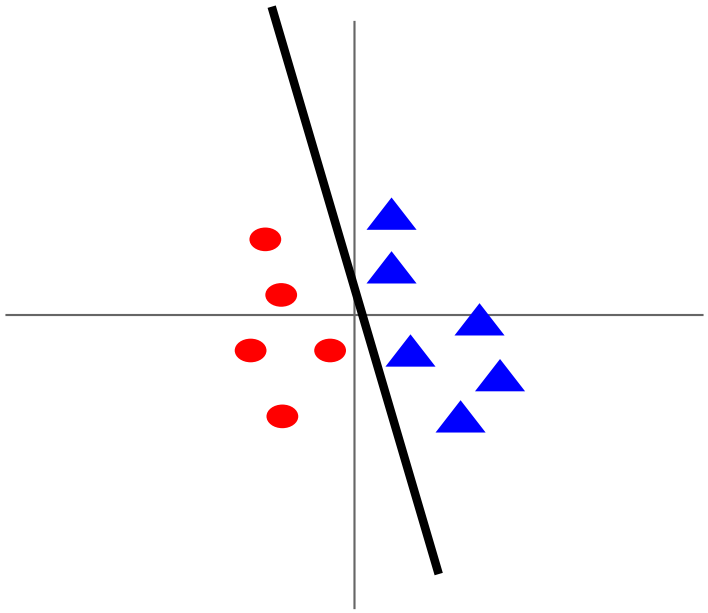
(covariance matrix is the identity matrix)

Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Other tricks

- Normalize the input:
  -- Normalize entire training data to make it 0 mean, until variance
  -- Equivalent of batch norm on input

- A variety of other tricks are applied
  -- Initialization techniques
  - Xavier, Kaiming, SVD, etc.
  - Key point: neurons with identical connections that are identically initialized will never diverge

Setting up a problem

- Obtain training data
  -- Use appropriate representation for inputs and outputs
- Choose network architecture
  -- More neurons need more data
  -- Deep is better, but harder to train
- Choose the appropriate divergence function
  -- Choose regularization
- Choose heuristics (batch norm, dropout, etc.)
- Choose optimization algorithm
  -- e.g. ADAM
- Perform a grid search for hyper parameters (learning rate, regularization parameter, …) on held-out data
- Train
  -- Evaluate periodically on validation data, for early stopping if required

Summary

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
  (random sampling, in log space when appropriate)