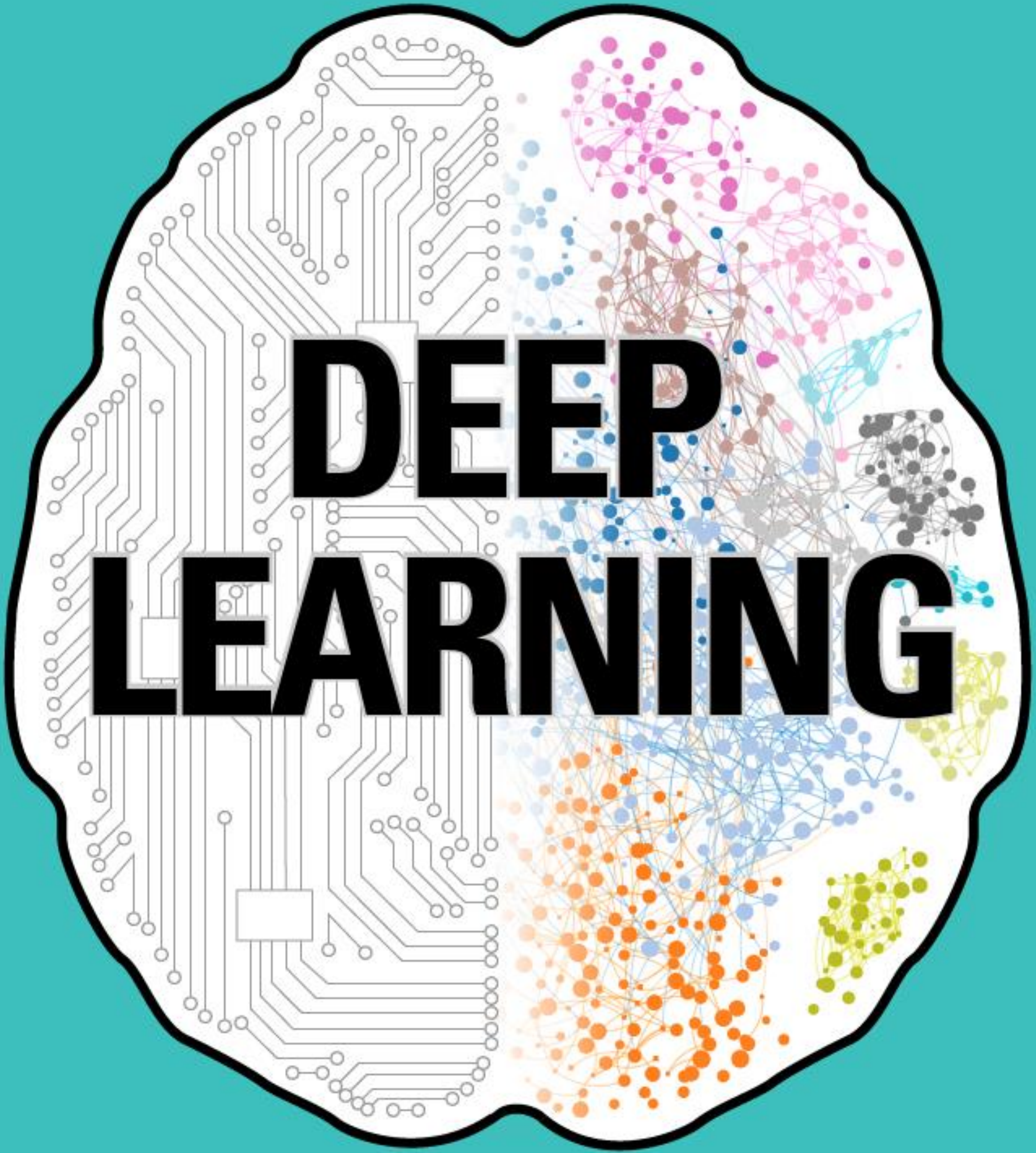


Neural Networks & Deep Learning

Chapter 1. Using neural nets to recognize handwritten digits
Chapter 2. How the backpropagation algorithm works
Chapter 3. Improving the way neural networks learn
Chapter 4. A visual proof that neural nets can compute any function
Chapter 5. Why are deep neural networks hard to train?
Chapter 6. Deep learning

Appendix I. Is there a simple algorithm for intelligence?
Appendix II. Acknowledgements
Appendix III. Frequently Asked Questions

http://neuralnetworksanddeeplearning.com/chap3.html

Keywords in this chapter

Basic techniques has been covered so far. (basic swing)

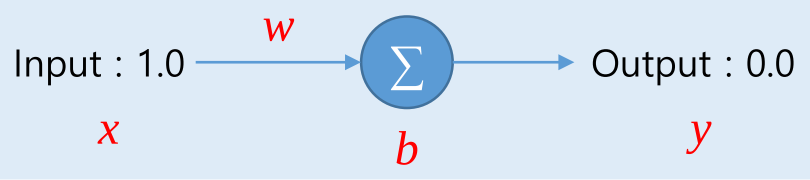
Now, we are dealing with various techniques to improve the 'basic swing'

- **Cross-entropy cost function**
- **Four regularization methods**
  - L_1/L_2 regularization, dropout,
  - artificial expansion of the training data
- **A better method for initializing the weights**
- **A set of heuristics to help choose good hyper-parameters**
- **Several other techniques**

The cross-entropy cost function

We hope and expect that our neural networks will learn fast from their errors.
An example :

An Object



$$C = \frac{(a-y)^2}{2} = \frac{a^2}{2}$$

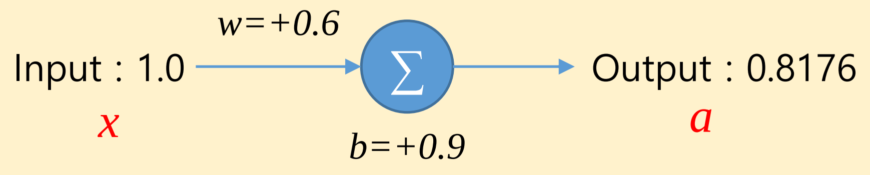
$$a = \sigma(z) = \sigma(wx+b) = \sigma(w+b)$$

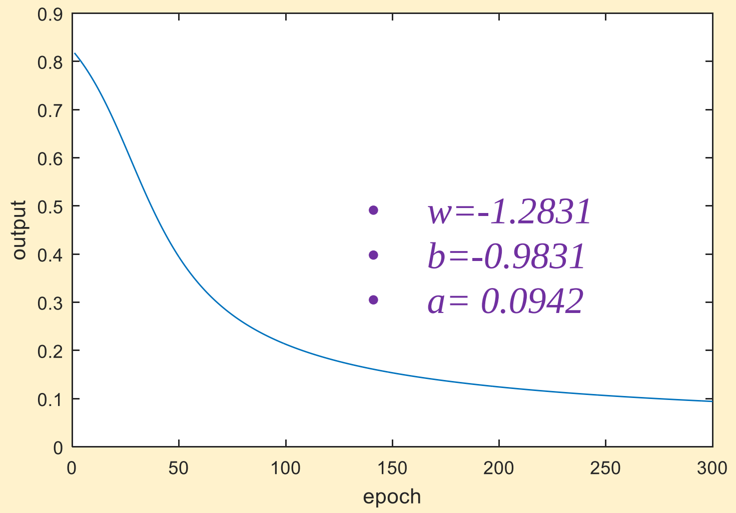
$$\delta = a\sigma'(w+b)$$

$$w = w - \eta\delta$$

$$b = b - \eta\delta$$

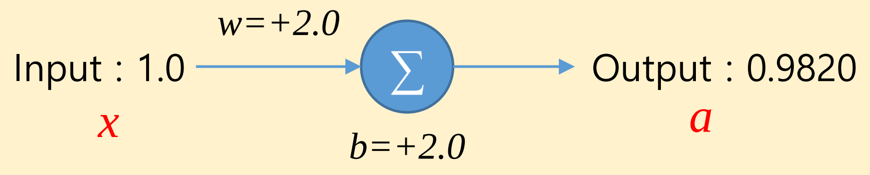
Initialization 1

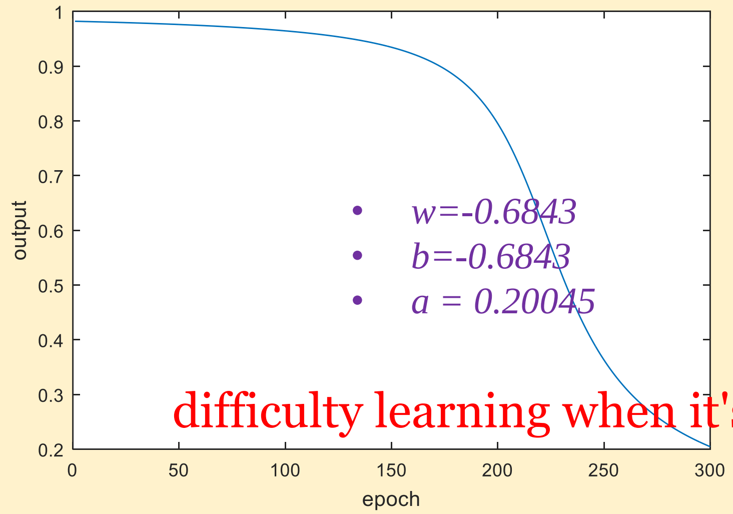




- w=-1.2831
- b=-0.9831
- a= 0.0942

Initialization 2





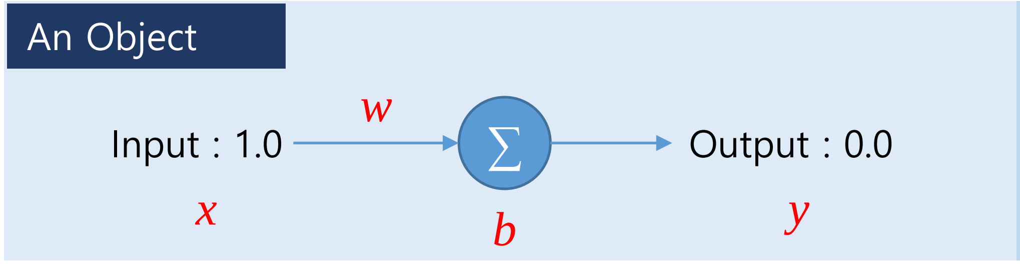
- w=-0.6843
- b=-0.6843
- a = 0.20045

difficulty learning when it's badly wrong

The cross-entropy cost function

We hope and expect that our neural networks will learn fast from their errors.
An example :

An Object



$$C = \frac{(a-y)^2}{2} = \frac{a^2}{2}$$

$$a = \sigma(z) = \sigma(wx+b) = \sigma(w+b)$$

$$\delta = a\sigma'(w+b)$$

$$w = w - \eta\delta$$

$$b = b - \eta\delta$$

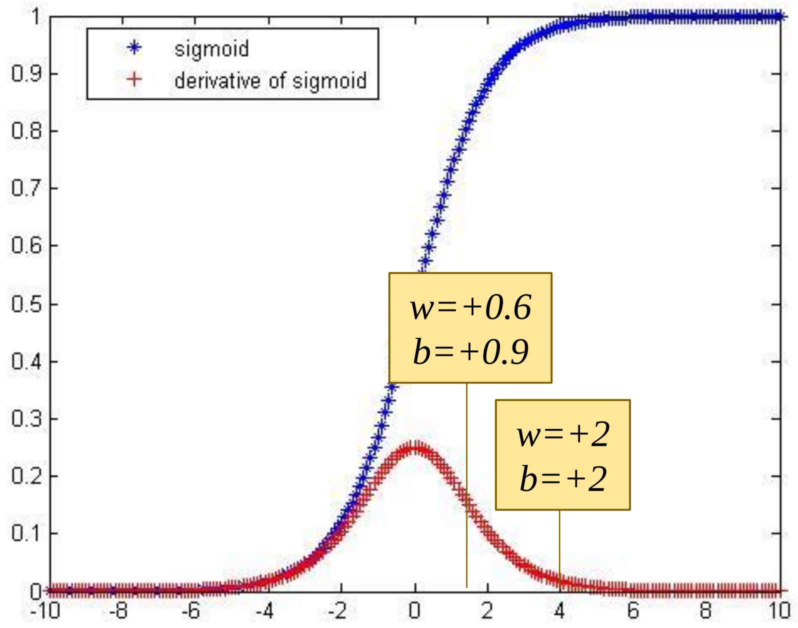
Learning slow means are $\partial C / \partial w$, $\partial C / \partial b$ small

Why they are small???

$$C = \frac{(y-a)^2}{2}$$

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

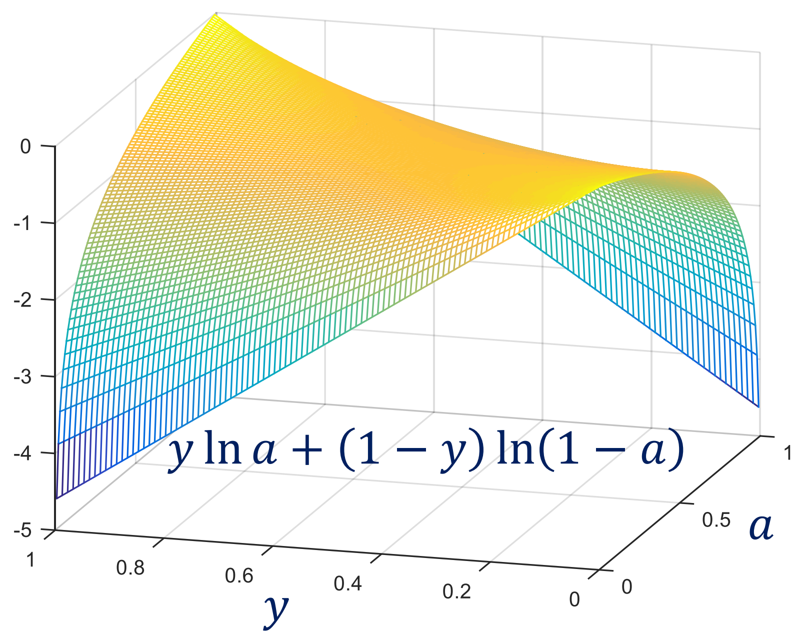
$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z)$$



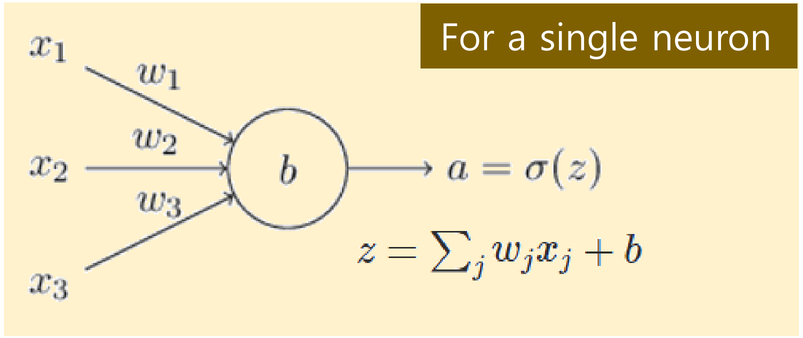
The cross-entropy cost function

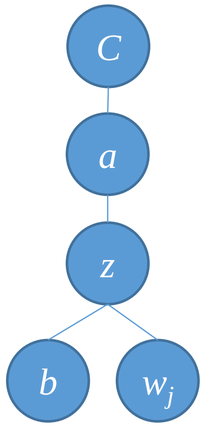
Introducing the cross-entropy cost function

$$C = \frac{-1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)]$$



[Two features]
- Non-negative → cost function
- Zero at y=a





Error gradient

$$\frac{\partial C}{\partial w_j} = \frac{-1}{n}\sum_x \frac{\partial C}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial w_j}$$

$$\frac{\partial C}{\partial a} = \frac{y}{a} + (1-y)\frac{-1}{1-a} = \frac{y-a}{(1-a)a}$$

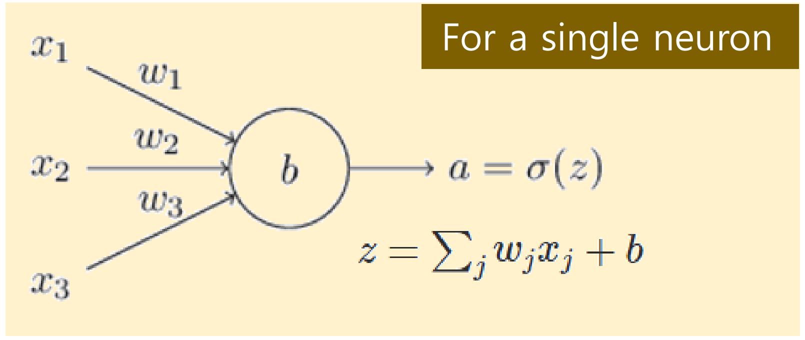
$$\frac{\partial a}{\partial z} = \sigma'(z) = \big(1-\sigma(z)\big)\sigma(z) = (1-a)a \qquad \frac{\partial z}{\partial w_j} = x_j$$

$$\frac{\partial C}{\partial w_j} = \frac{-1}{n}\sum_x (y-a)x_j = \frac{1}{n}\sum_x (\sigma(z)-y)x_j$$

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z)-y)$$

The cross-entropy cost function

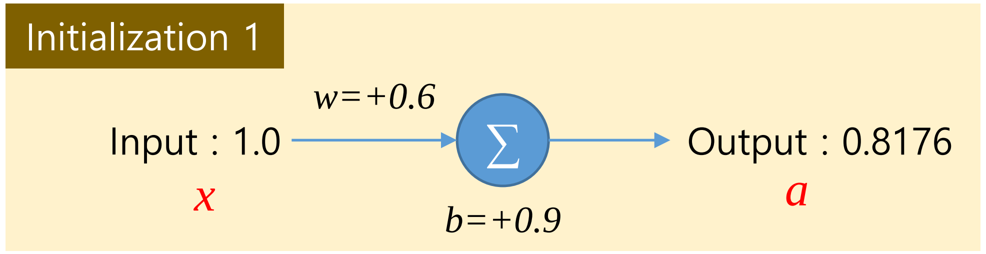
Error gradient comparison

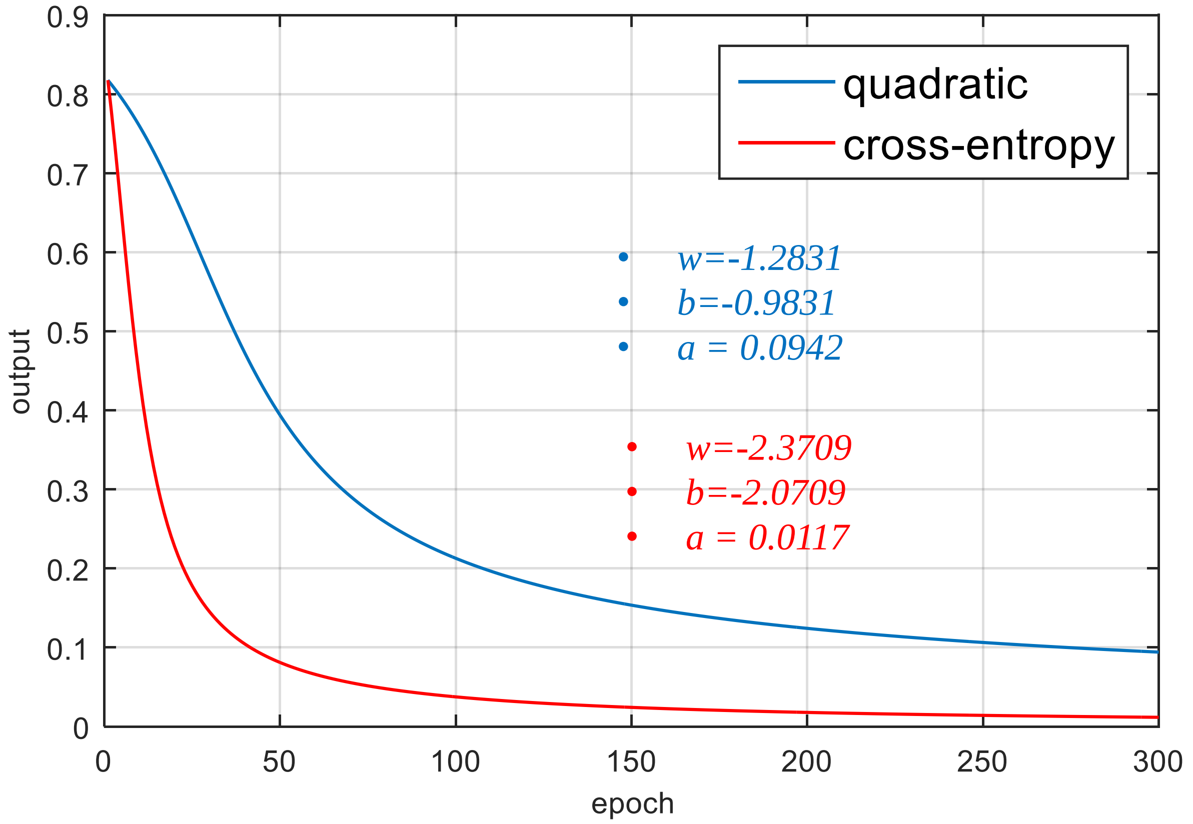


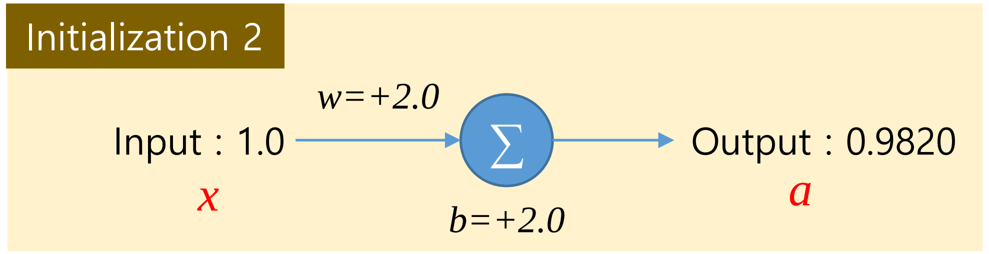
$Cross - entropy\ cost$	$Quadratic\ cost$
$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)\ ]$	$C = \frac{1}{n}\sum_x \frac{1}{2}(a-y)^2$
$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x (\sigma(z)-y)x_j$ $\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z)-y)$	$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x (\sigma(z)-y)\cdot\boxed{\sigma'(z)}\cdot x_j$ Error signal decreaser!!! $\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z)-y)\cdot\boxed{\sigma'(z)}$

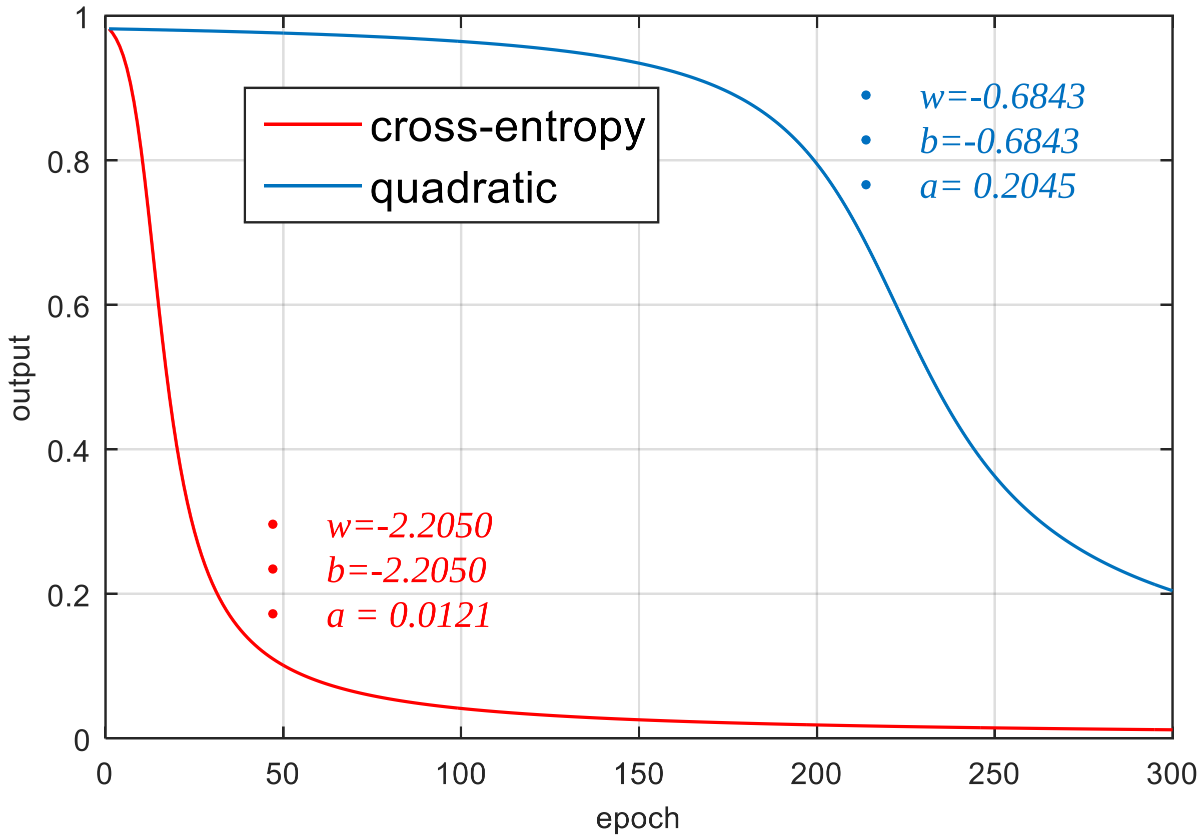
The cross-entropy cost function

A toy example with cross entropy



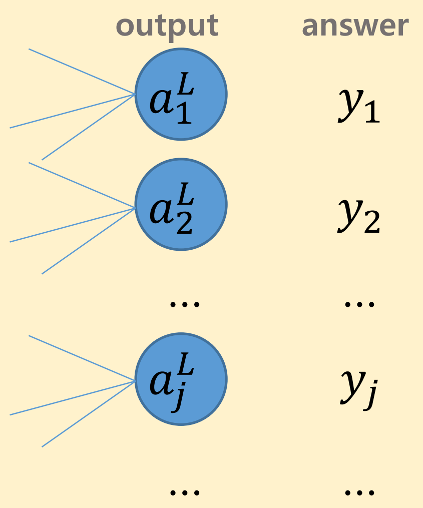






The cross-entropy cost function

For many output neurons,



$$C = -\frac{1}{n}\sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

Remember two assumptions on cost function for back-propagation algorithm

Assumption 1. Total cost of NN over training samples is the sum of cost for each training example OK

Assumption 2. Cost for each training example is a function of a^L OK

The cross-entropy cost function

Four fundamental equations for cross-entropy cost function

EQ 1. Error in the output layer $\delta^L = \nabla_a C \odot \sigma'(z^L)$

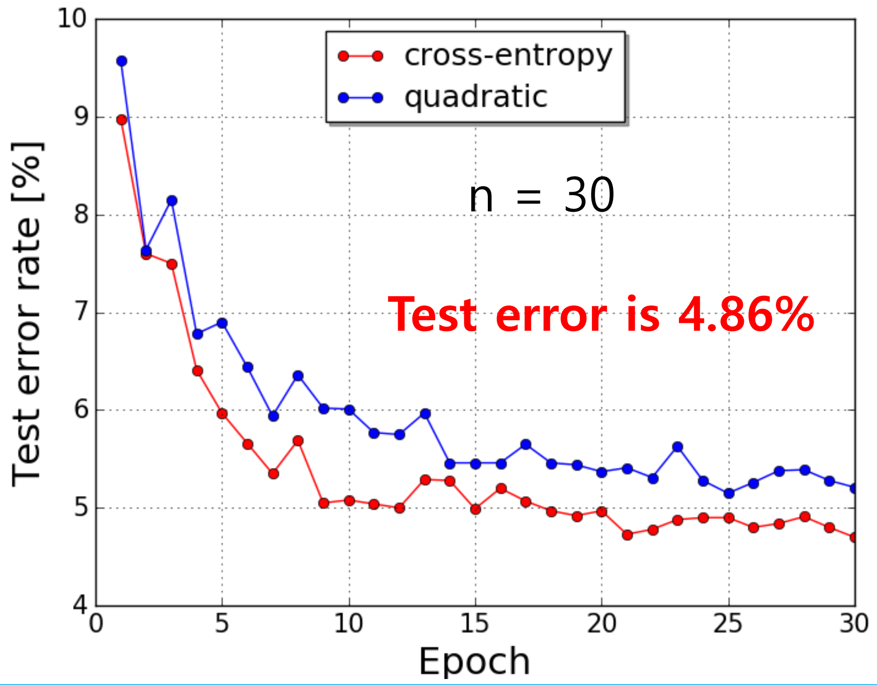
Let's ignore index for training sample because we are focusing cost for a training sample.

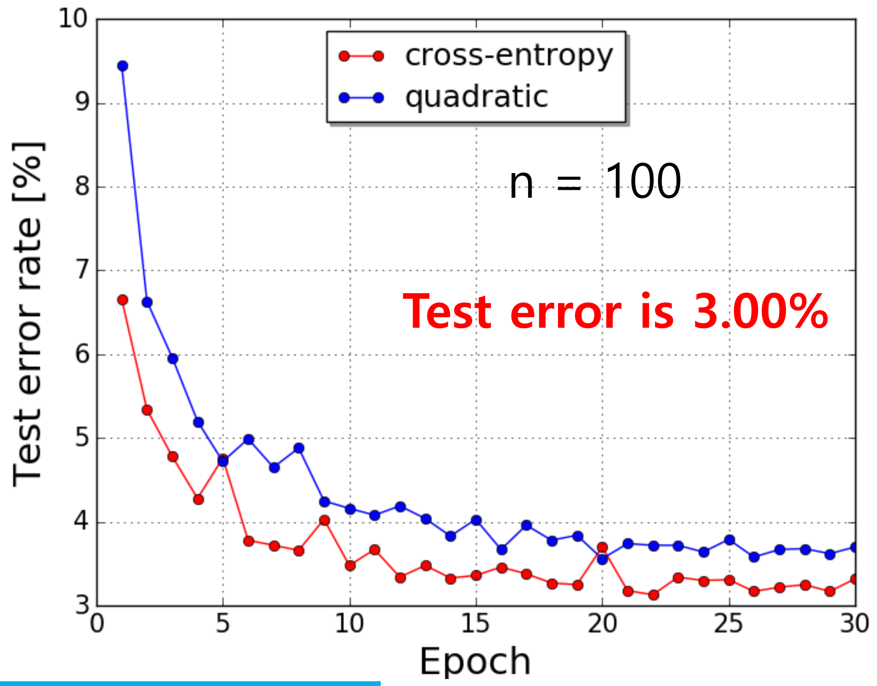
$Cross - entropy\ cost$	$Quadratic\ cost$
$C = -\sum_j \left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L) \right]$	$C = \frac{1}{2}\sum_j \left[y_j - a_j^L \right]^2$
$\begin{aligned} \frac{\partial C}{\partial z_j^L} &= \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \\ &= -\left(\frac{y_j}{a_j^L} + (1-y_j)\frac{-1}{1-a_j^L} \right) \cdot \sigma'(z_j^L) \\ &= (a_j^L - y_j) \end{aligned}$	$\begin{aligned} \frac{\partial C}{\partial z_j^L} &= \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \\ &= (a_j^L - y_j) \cdot \sigma'(z_j^L) \end{aligned}$
$\delta^L = (a^L - y)$	$\delta^L = (a^L - y)\boxed{\odot \sigma'(z^L)}$ Error signal decreaser!!!
$\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{jk}^L} = (a_j^L - y_j) \cdot a_k^{L-1}$	$\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{jk}^L} = (a_j^L - y_j)\boxed{\cdot \sigma'(z_j^L)} \cdot a_k^{L-1}$

The cross-entropy cost function

example2.py

Classify digits





Test Error

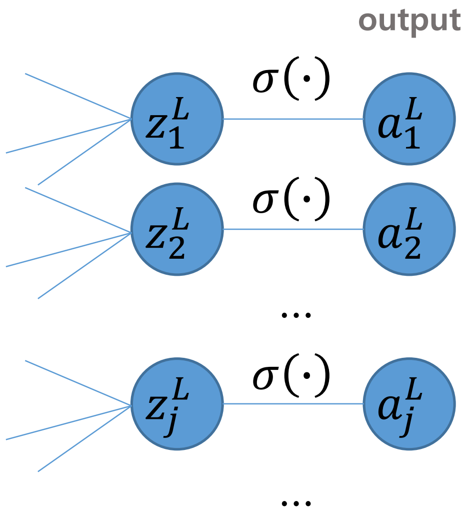
n	Quadratic	Cross-entropy
30	5.53%	4.86%
100	3.74% (some variations)	3.00%

- n=30; //nodes in hidden layer
- b=10; //mini-batch size
- l_q = 3; //learning rate for quadratic
- l_ce = 0.5; //learning rate for cross-entropy

- Usually, cross-entropy cost function shows similar performance or a little better performance as compared with quadratic cost function.
- Remember, hyper-parameters are not optimized.

Softmax

Original Output Layer



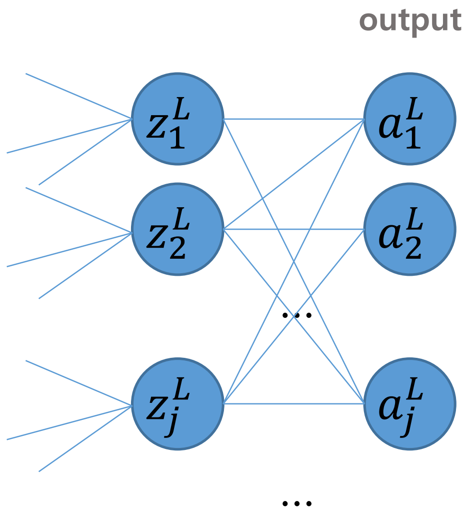
$$a_j^L = \sigma(z_j^L) = \frac{1}{1+e^{-z_j^L}} \in [0,1]$$

$$0 \le \sum_j a_j^L \le N^{(L)}$$

In classification problem, a desired output vector contains zero elements except only one ‘1’ element .

→ This can be view as sum of all elements is 1

Output Layer of Softmax



$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \in [0,1]$$

Non-locality

$$\sum_j a_j^L = 1$$

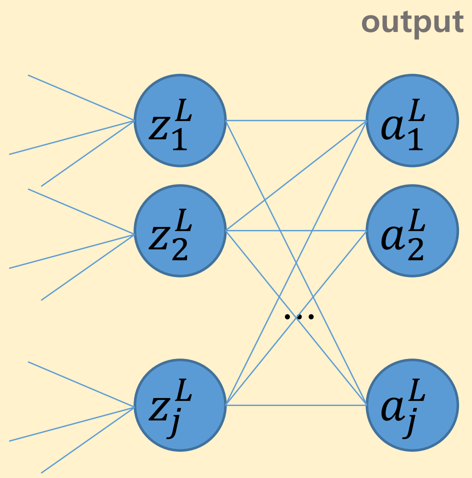
A softmax layer outputs a probability distribution!!

Monotonicity $\frac{\partial a_j^L}{\partial z_k^L}$ = _positive_ if j=k, _negative_ otherwise

As a consequence, increasing z_j^L is guaranteed to increase the corresponding output activation, a_j^L, and will decrease all the other output activations.

Softmax

The learning slowdown problem



$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} = \frac{e^{z_j^L}}{M} = \frac{e^{z_j^L}}{e^{z_j^L} + \sum_{k\neq j} e^{z_k^L}}$$

$$M = \sum_k e^{z_k^L}$$

$$\sum_j a_j^L = 1$$

Cost function for softmax

$$C = -\ln a_y^L$$

a_y^L *is output value of a neuron where must have '1' for desired output 'y'*

Remember that the learning slowdown problem is cause by $\sigma'(z)$ in δ^L

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} = \begin{cases} \frac{e^{z_j^L} - M}{M} = a_j^L - 1 = (a_j^L - y_j), if\ y_j = 1 \\ 0, otherwise \end{cases}$$

$$\frac{\partial C}{\partial a_j^L} = \frac{-1}{a_y^L} = \begin{cases} \frac{-1}{a_j^L} = \frac{-M}{e^{z_j^L}}, if\ y_j = 1 \\ 0, otherwise \end{cases}$$

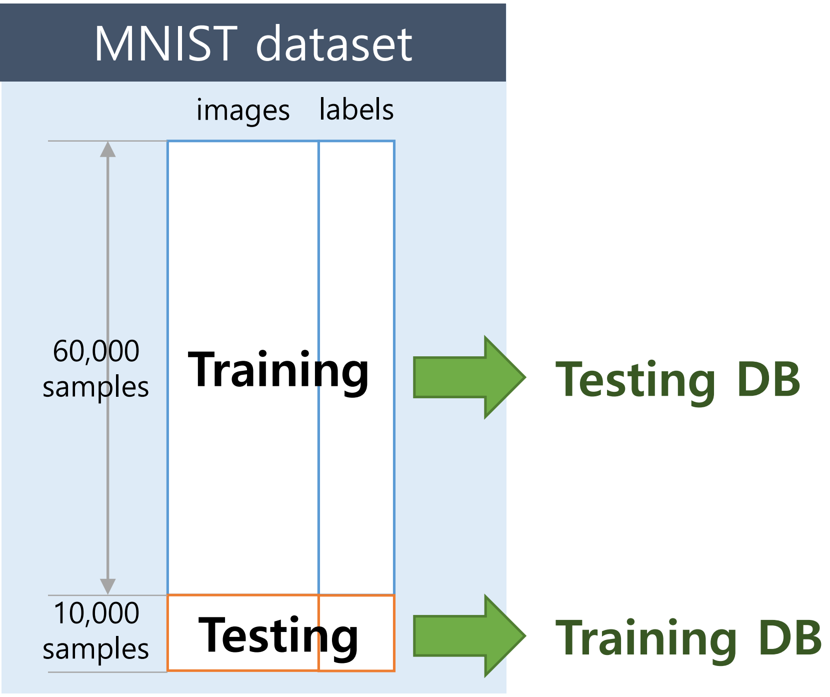
$$\frac{\partial a_j^L}{\partial z_j^L} = \frac{e^{z_j^L} \cdot M - e^{z_j^L} \cdot e^{z_j^L}}{M^2} = \frac{e^{z_j^L}\left(M - e^{z_j^L}\right)}{M^2}$$

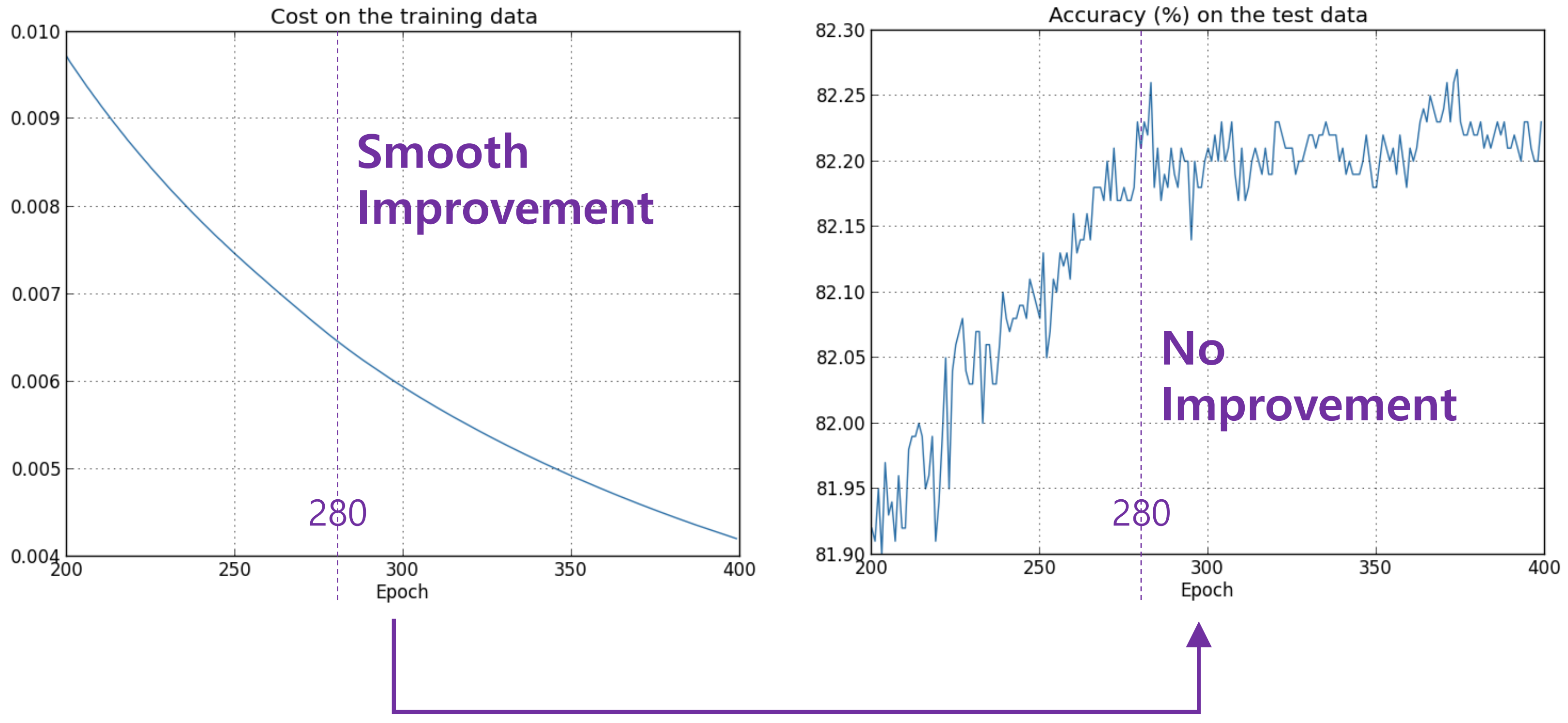
Overfitting and regularization

Overfitting

30 hidden neuron → (30x784+30)+10x30+10) = 23,860 parameters

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)]$$





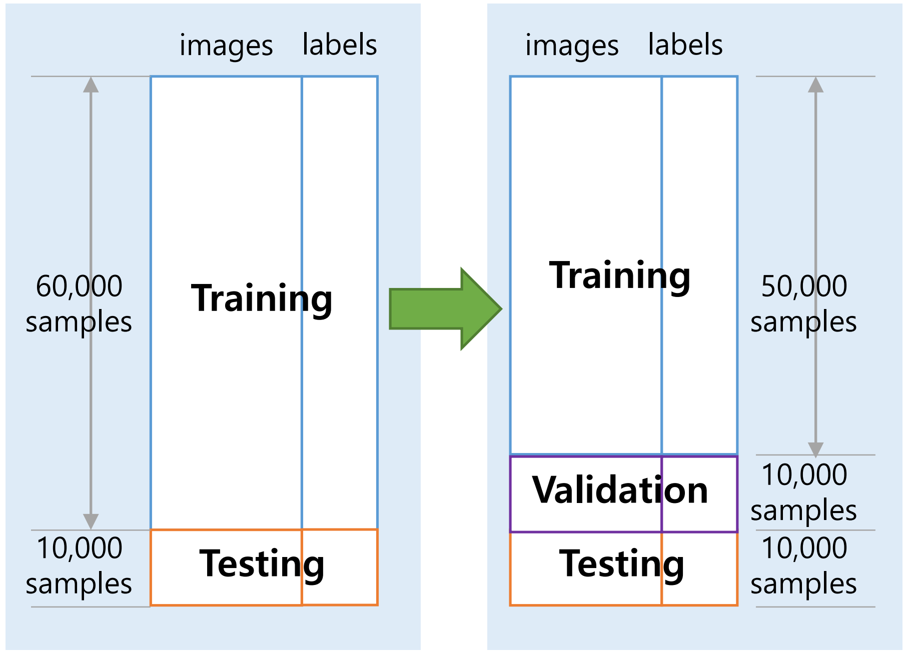
- n=30; //nodes in hidden layer
- b=10; //mini-batch size
- l = 0.5; //learning rate
- Epoch = 400; //number of iterations

A trained model is not generalized well
→ A model is overfitting or overtraining beyond epoch 280!!

Overfitting and regularization

Solution Early stop, hold-out

One way to prevent overfitting is to trace accuracy on the test data.



EARLY STOP

- Trace accuracy on the validation data and stop when there is no relevant improvement

HOLD OUT

- The validation data can be used to find good hypermeters
- Choosing proper hyper-parameters may avoid overfitting

Overfitting and regularization

Solution L2 Regularization (also called weight decay)

Regularization is one way to prevent overfitting given a fixed network and fixed training data.

$$C = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \underline{\frac{\lambda}{2n}\sum_w w^2}$$

Quadratic Cost

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \underline{\frac{\lambda}{2n}\sum_w w^2}$$

Cross-entropy Cost

Additional term (regularization term)
λ is known as a regularization parameter

General Cost

$$C = \underline{C_0} + \frac{\lambda}{2n}\sum_w w^2$$

The original, unregularized cost function

- Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal.
- Large weights will only be allowed if they considerably improve the first part of the cost function.
- Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function.
- The relative importance of the two elements of the compromise depends on the value of λ: when λ is small we prefer to minimize the original cost function, but when λ is large we prefer small weights

Overfitting and regularization

Solution L2 Regularization (also called weight decay)

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

For backpropagation algorithm

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b} \qquad b \rightarrow b - \eta \frac{\partial C_0}{\partial b}$$

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \qquad \begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \underline{\left(1 - \frac{\eta \lambda}{n}\right)} w - \eta \frac{\partial C_0}{\partial w} \end{aligned}$$

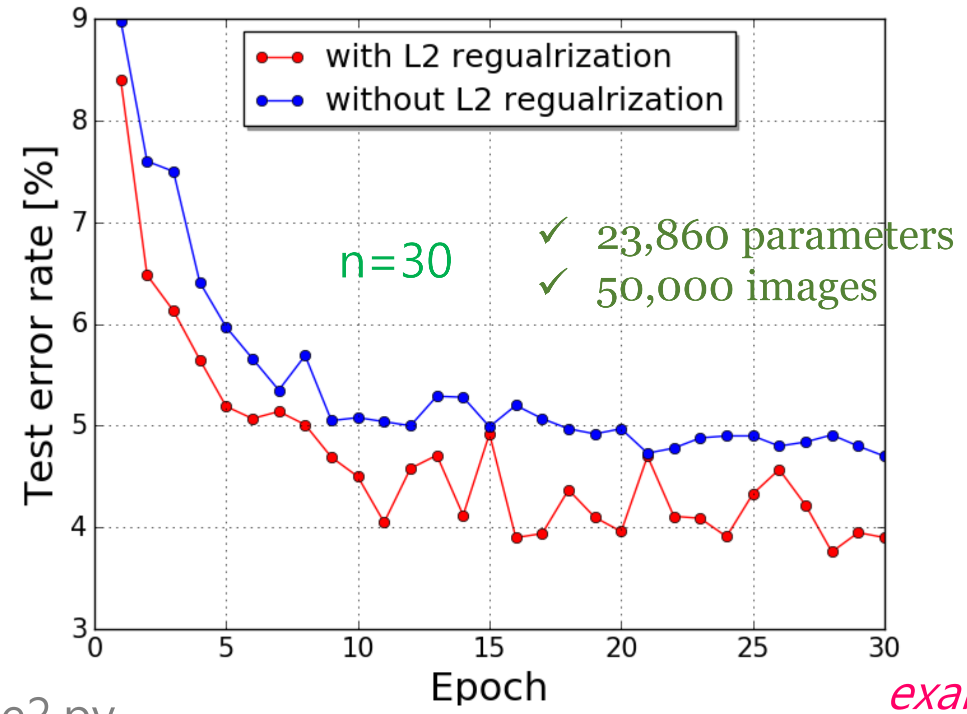
- Originally this is one.
- Usually this is less than 1 and causes ‘weight decay’

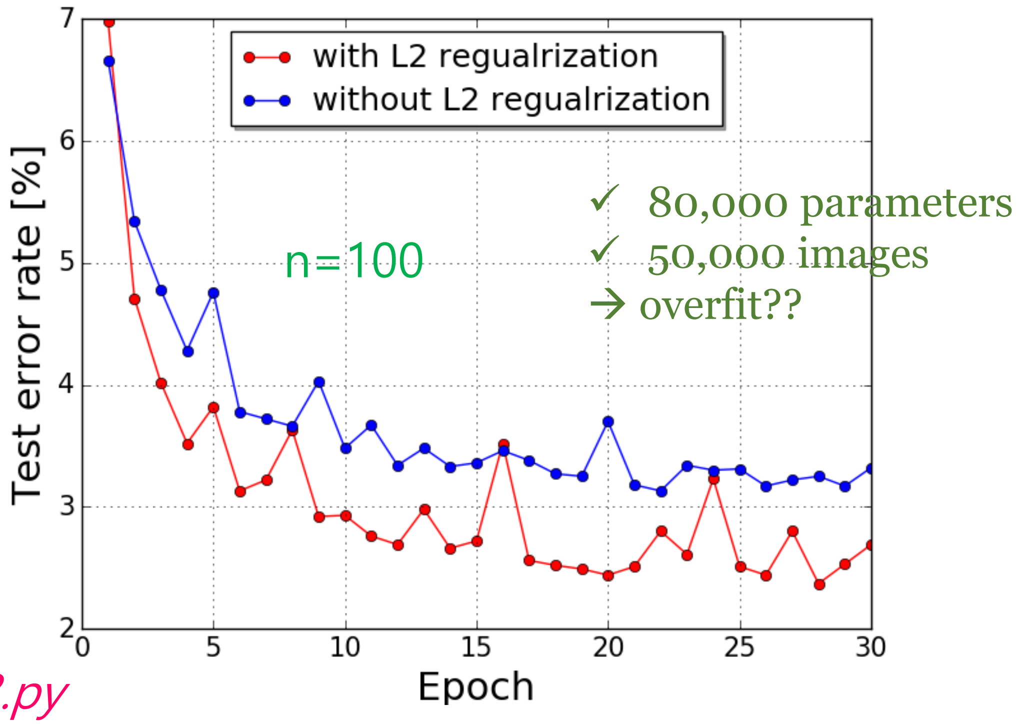
Overfitting and regularization

Solution L2 Regularization (also called weight decay)

n	Quadratic	Cross-entropy	+Regularization
30	5.53%	4.86%	4.03%
100	3.74%	3.00%	2.56%

- n=30; //nodes in hidden layer
- b=10; //mini-batch size
- l_ce = 0.5; //learning rate for cross-entropy
- λ = 5; //regularization parameter





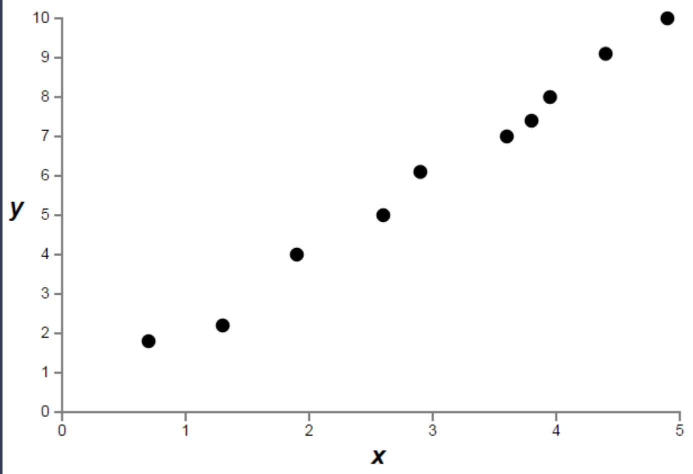
Baseline : example2.py

example3.py

Overfitting and regularization

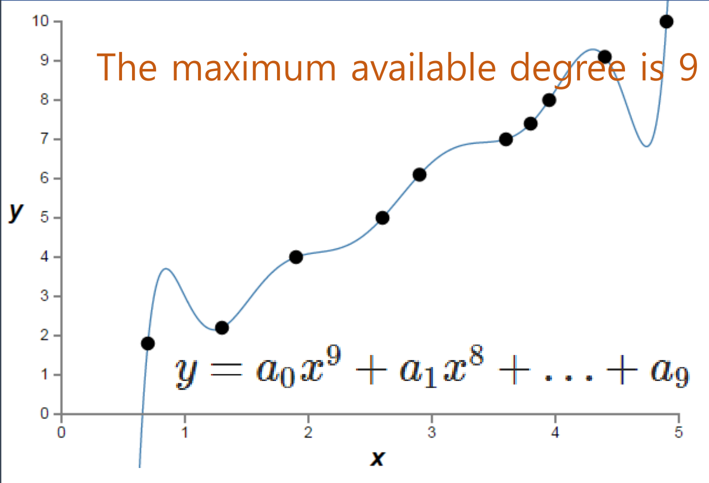
Why does regularization help reduce overfitting?

Given 10 points, find a model

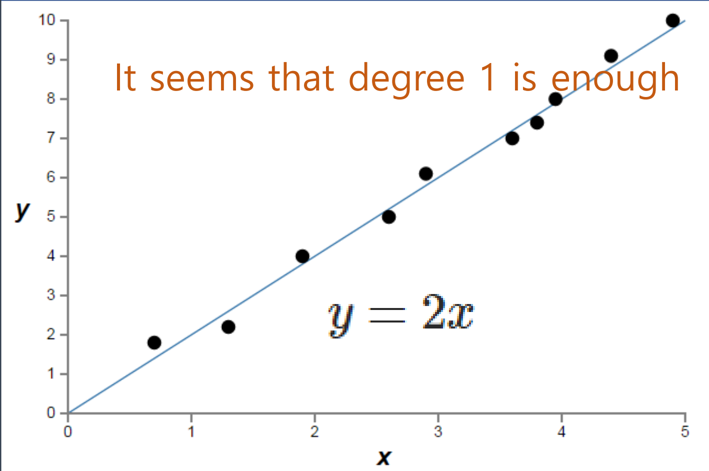


We are given 10 points and try to find relationship via polynomial fitting

Maximum degree : 9



Proper degree : 1



Which is better model????

In science we should go with the simpler explanation, unless compelled not to.
Ockham's Razor

Think about when x is 10000, which will cause more error??

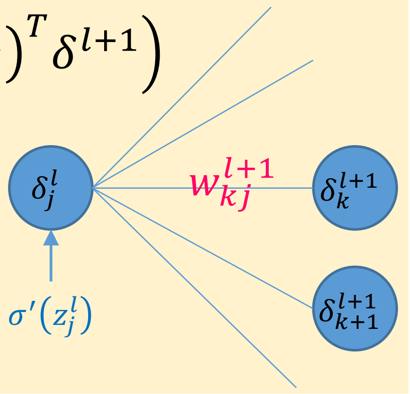
Overfitting and regularization

Why does regularization help reduce overfitting?

Small weights network (Regularized)	Large weights network
• The smallness of the weights means that the behavior of the network won't change too much if we change a few random inputs here and there. • That makes it difficult for a regularized network to learn the effects of local noise in the data. • A regularized network learns to respond to types of evidence which are seen often across the training set. • Regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data.	• A network with large weights may change its behavior quite a bit in response to small changes in the input. • An unregularized network can use large weights to learn a complex model that carries a lot of information about the noise in the training data.

Remeber $\delta^l = \sigma'(z^l)\odot\left(\left(w^{l+1}\right)^T\delta^{l+1}\right)$

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sigma'(z_j^l)\sum_k \delta_k^{l+1} w_{kj}^{l+1}$$



Overfitting and regularization

Bias?

- The fact that L2 regularization doesn't constrain the biases.

- Of course, it would be easy to modify the regularization procedure to regularize the biases.
- Empirically, doing this often doesn't change the results very much, so to some extent it's merely a convention whether to regularize the biases or not.

- However, it's worth noting that having a large bias doesn't make a neuron sensitive to its inputs in the same way as having large weights. And so we don't need to worry about large biases enabling our network to learn the noise in our training data.

- At the same time, allowing large biases gives our networks more flexibility in behavior - in particular, large biases make it easier for neurons to saturate, which is sometimes desirable.

- For these reasons we don't usually include bias terms when regularizing.

Overfitting and regularization

Other techniques for regularization

1. L1 regularization
→ Constraints on weights

2. Dropout
→ Reduce the number of parameters by rejecting some neurons

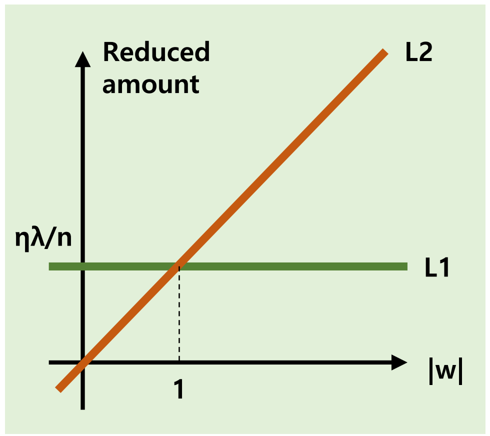
3. Data augmentation (Artificially expanding the training data)
→ Increase the number of training samples

Overfitting and regularization

Other techniques for regularization

1. L1 regularization

L1 regularization	L2 regularization
$C = C_0 + \frac{\lambda}{n}\sum_w \lvert w \rvert$	$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$
$w \rightarrow w' = \underline{w - \frac{\eta\lambda}{n}\text{sgn}(w)} - \eta\frac{\partial C_0}{\partial w}$	$w \rightarrow w - \eta\frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n}w = \left(1 - \frac{\eta\lambda}{n}\right)w - \eta\frac{\partial C_0}{\partial w}$
• w > 0 → w – positive : shrink the weight • w < 0 → w + positive : shrink the weight sgn(0)=0: There is no weight decay if w = 0	
Weight is reduced by absolute amount	**Weight is reduced by relative amount to w**

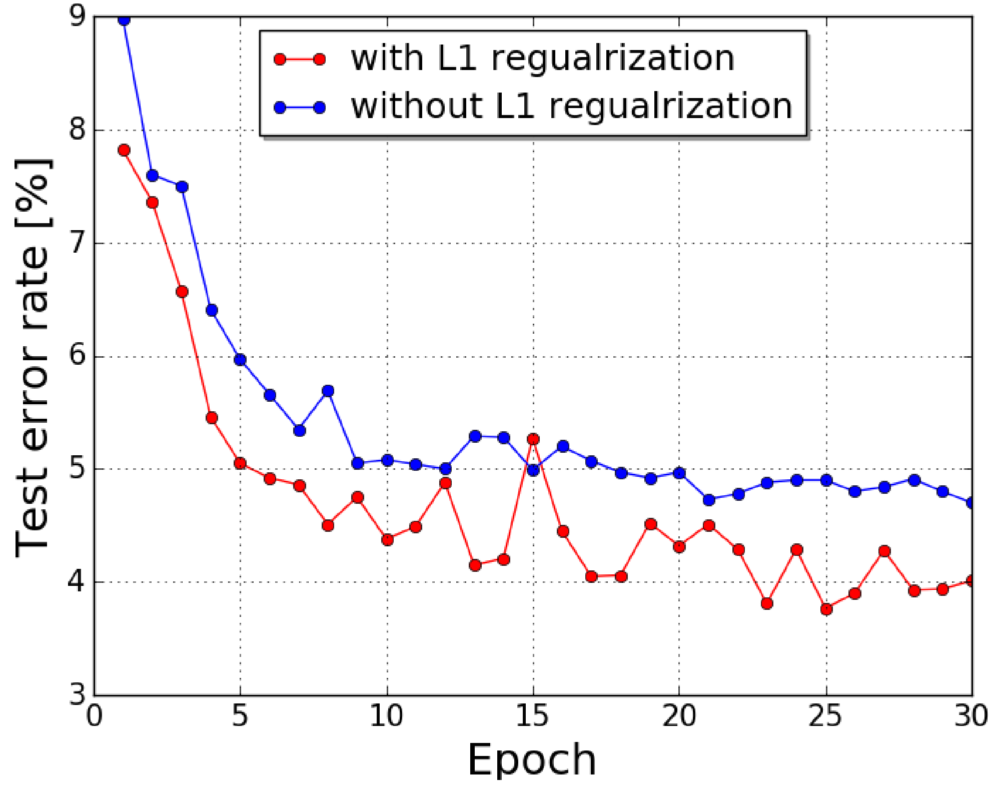


Overfitting and regularization

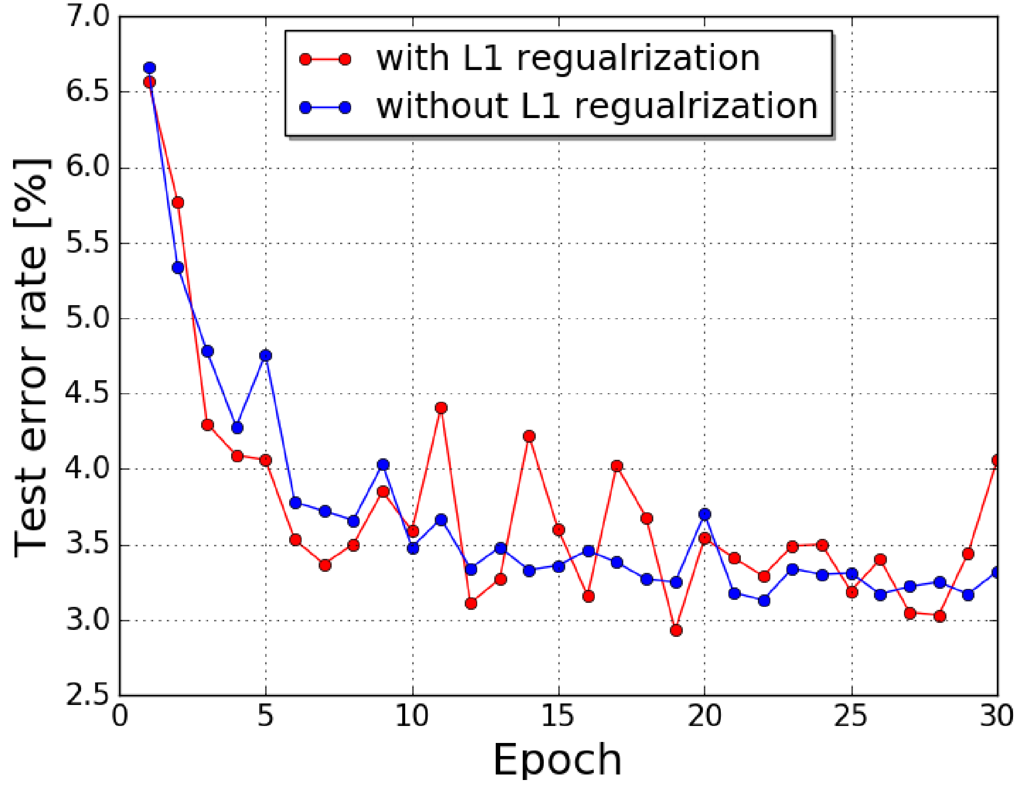
Other techniques for regularization

1. L1 regularization

n=30



n=100



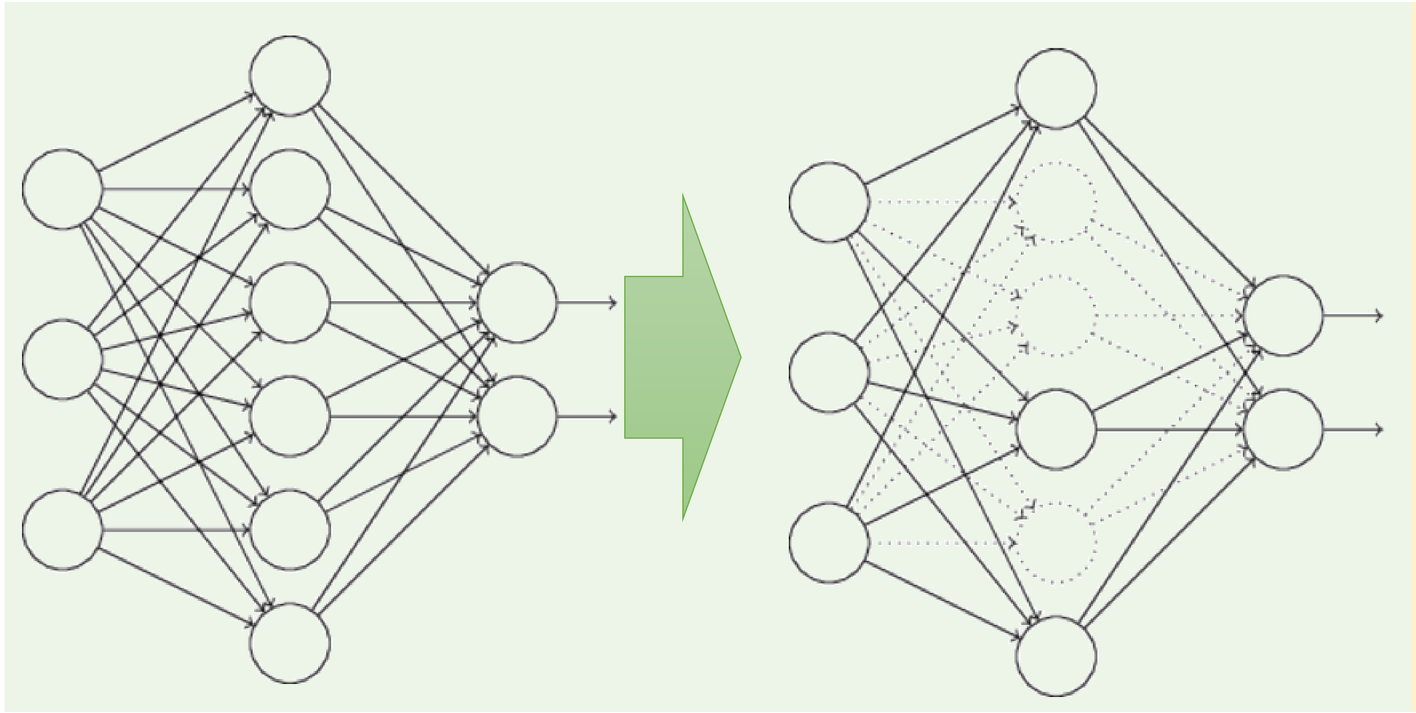
example4.py

Baseline : example2.py

Overfitting and regularization

Other techniques for regularization

2. Dropout : It makes the model robust to the loss of any individual piece of evidence



Training
For each hidden neuron, for each training samples, for each iteration, ignore (zero out) a different random fraction p of input activations

Testing
Use all activations, but reduce them by a factor p to simulate the missing activations during training

[Hidden story]
For instance, if we have trained five networks, and three of them are classifying a digit as a "3", then it probably really is a "3". The other two networks are probably just making a mistake. This kind of averaging scheme is often found to be a powerful (though expensive) way of reducing overfitting.
If we think of our network as a model which is making predictions, then we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence.

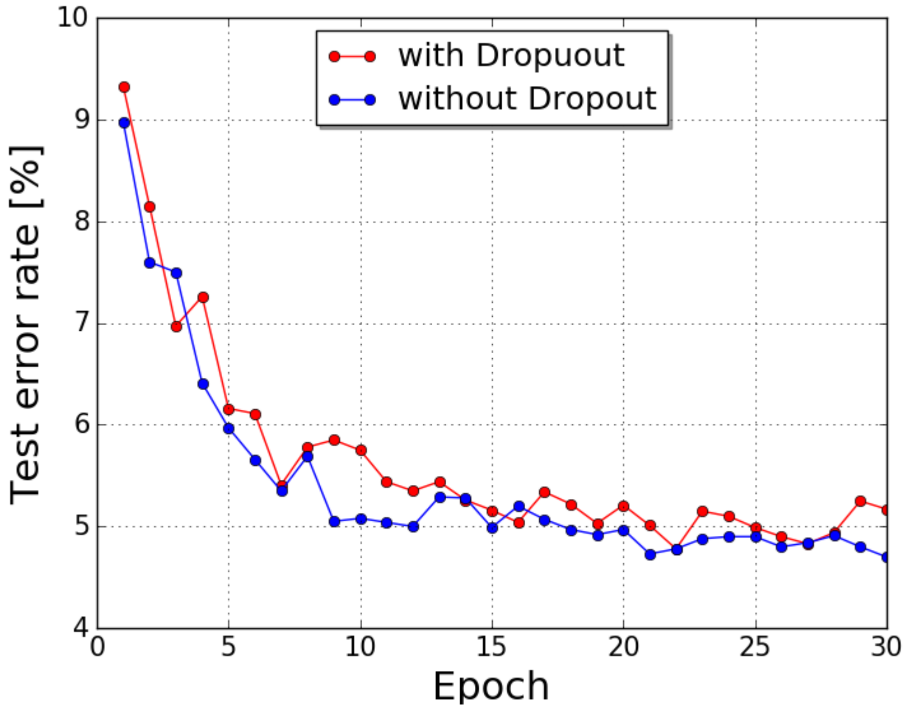
Overfitting and regularization

Other techniques for regularization

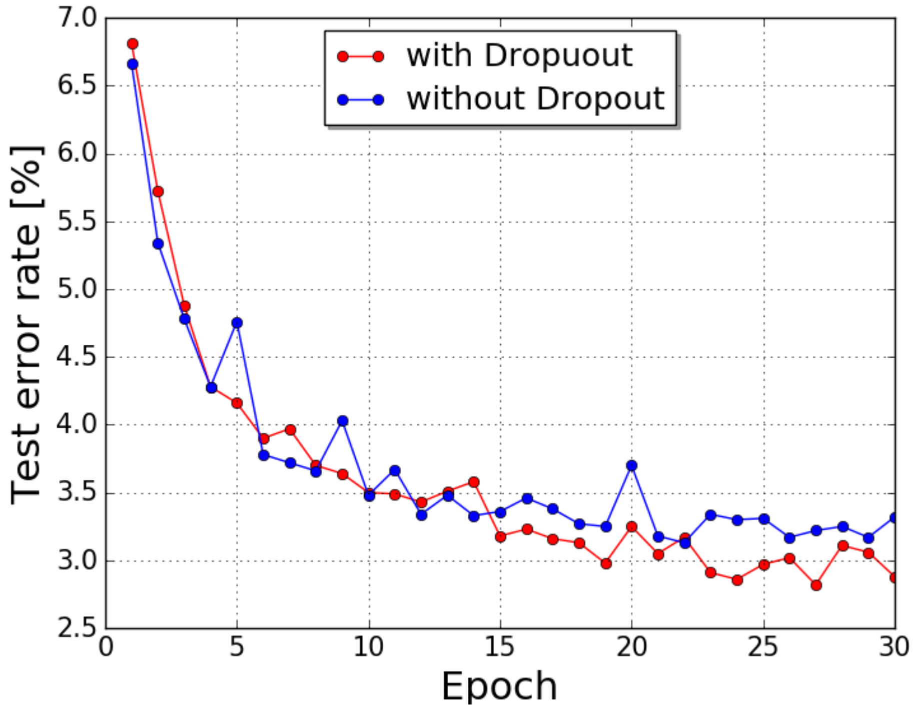
2. Dropout : It makes the model robust to the loss of any individual piece of evidence

https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/dropout_layer.html

n=30



n=100



Baseline : example2.py

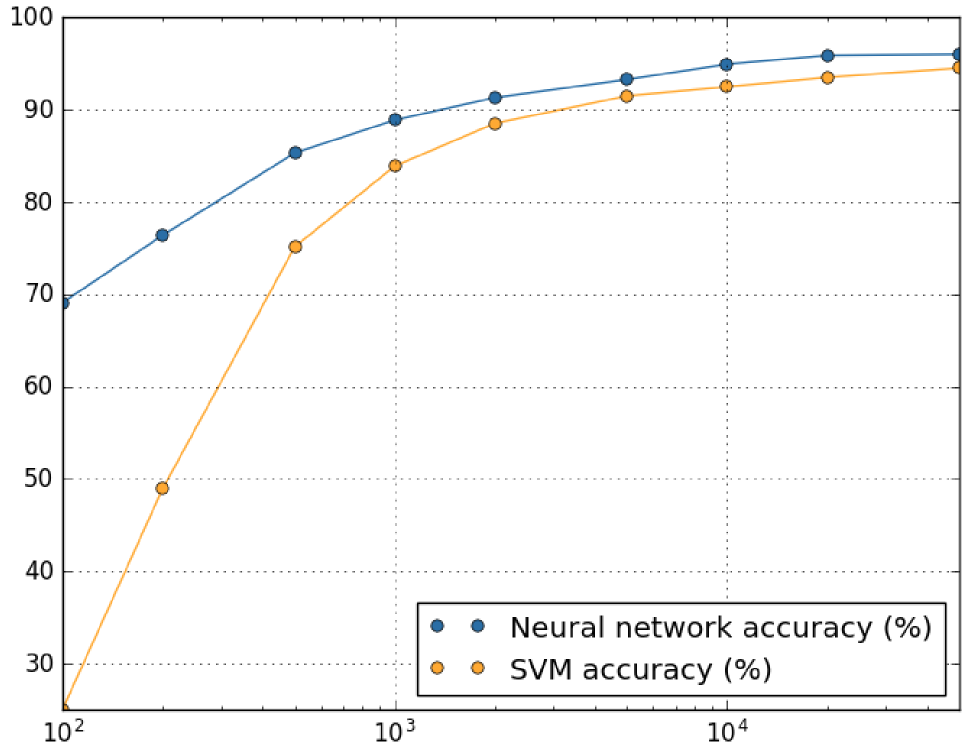
example5.py

Overfitting and regularization

Other techniques for regularization

3. Data augmentation (Artificially expanding the training data)

- n=30; //nodes in hidden layer



Training set size

Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis, by Patrice Simard, Dave Steinkraus, and John Platt (2003)

- n=800; //nodes in hidden layer

98.4

Add random rotations, translations, skews

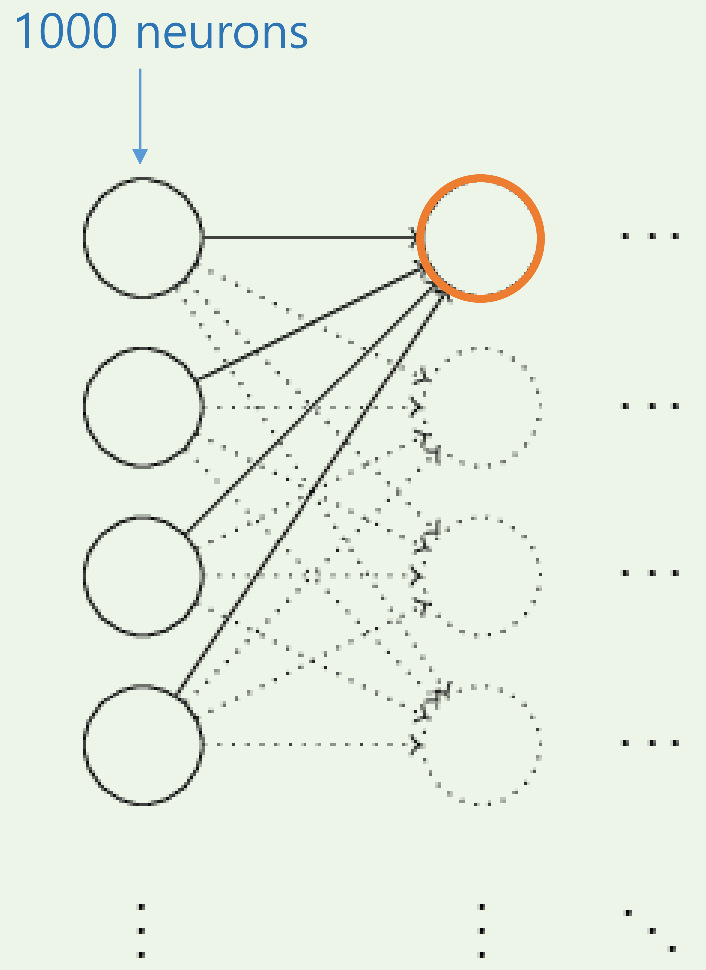
98.9

Add 'elastic distortions' intended to emulate the random oscillations found in hand muscles

99.3

Weight initialization

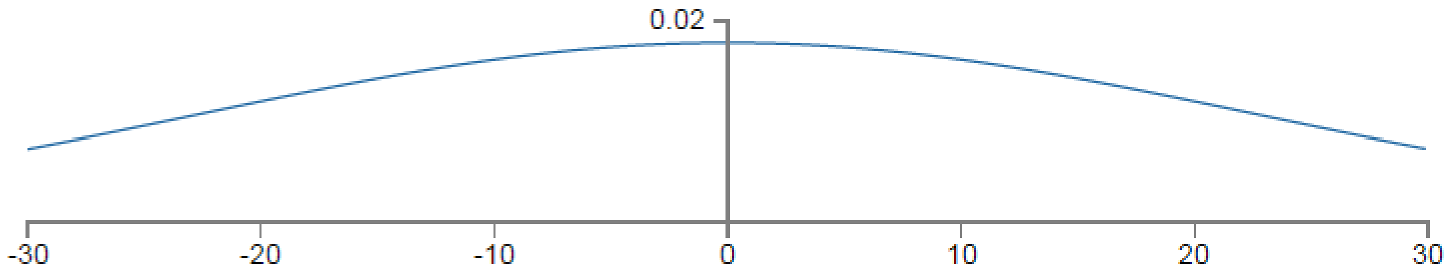
Initializing with normalized Gaussians



Let's say half of input neurons are deactivated
(i.e. selected inputs are set to zero, other inputs are set to one)

$$z = \sum_j w_j x_j + b \Rightarrow z = \sum_{x_j=1} w_j + b \ \ where\ w_j, b \sim N(0,1)$$

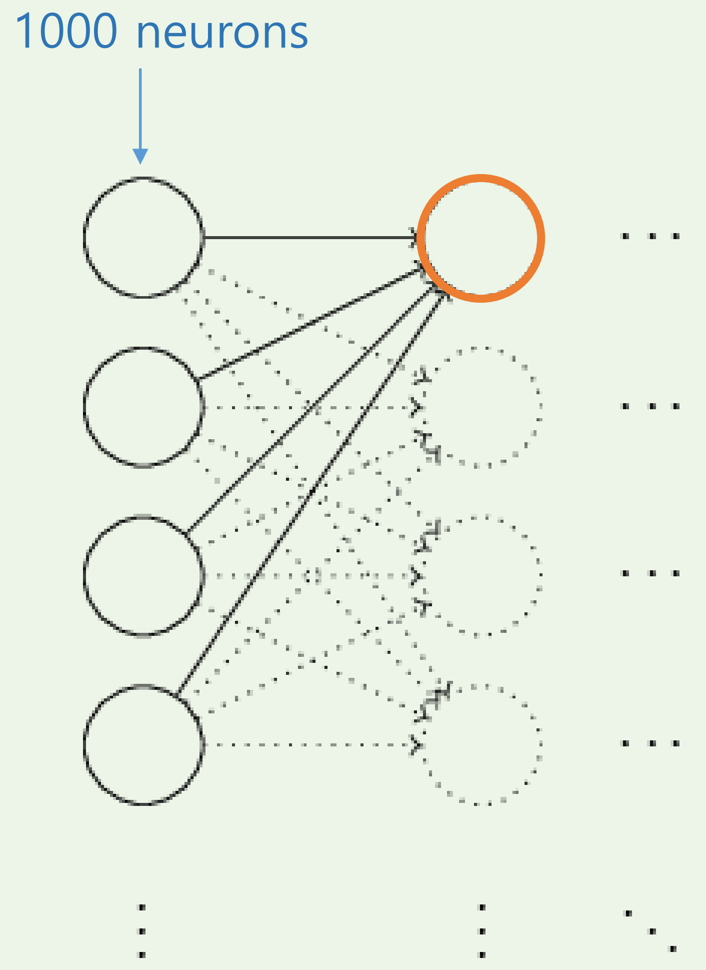
So z is a sum over a total of 501 normalized Gaussian random variables, accounting for the 500 weight terms and 1 extra bias term.
→ z is itself distributed as a Gaussian with zero mean, standard deviation sqrt(501) nearly 22.4



It is very likely for the hidden neuron to saturate → slow learning
→ Sigmoid outputs zero or one for inputs over +/-5

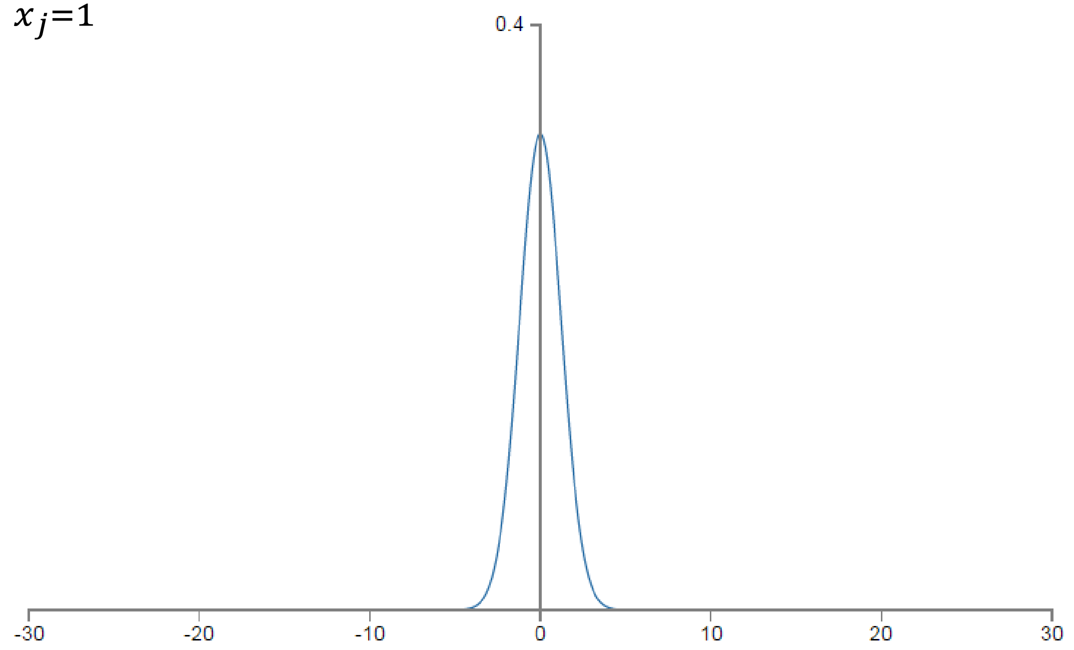
Weight initialization

Initializing with normalized Gaussians



If weight are normalized to distribute as a Gaussian with mean zero and standard deviation of 1/sqrt(number of input neurons),
z is itself distributed as a Gaussian with mean zero standard deviation sqrt(1+1/2) nearly 1.22 (1/2 comes from half-activation)

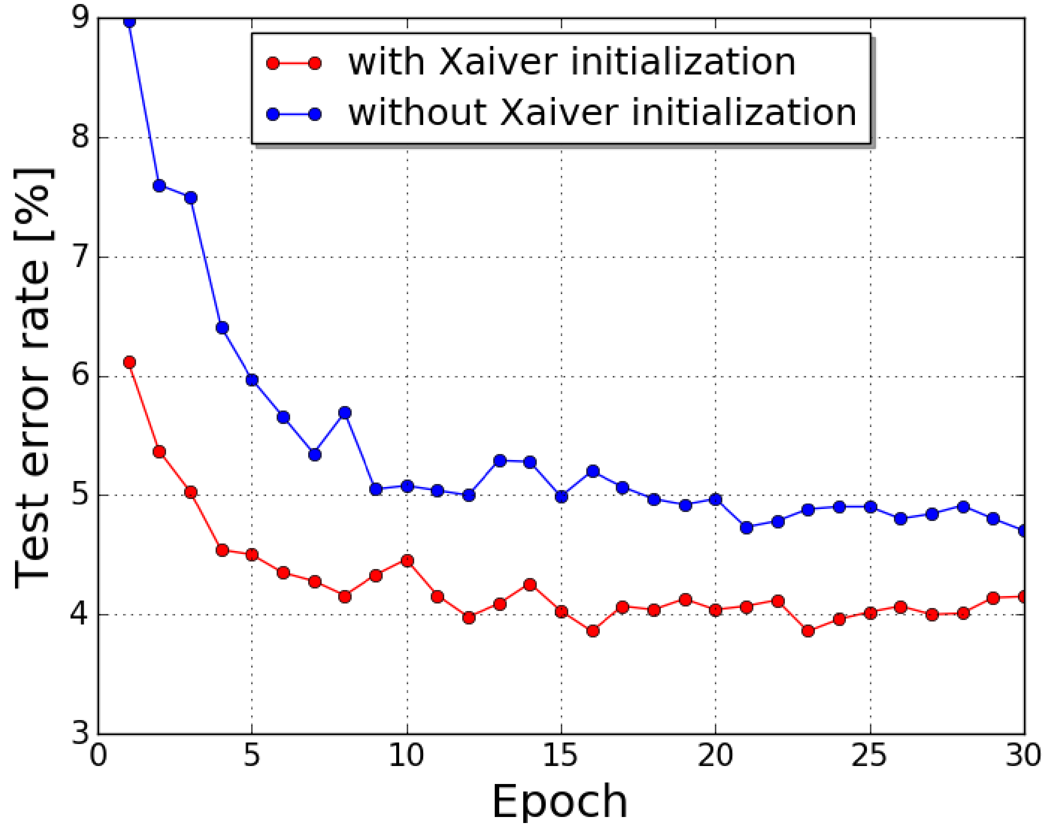
$$z = \sum_{x_j=1} w_j + b \ \ where \ w_j \sim N(0,1/1000), b \sim N(0,1)$$



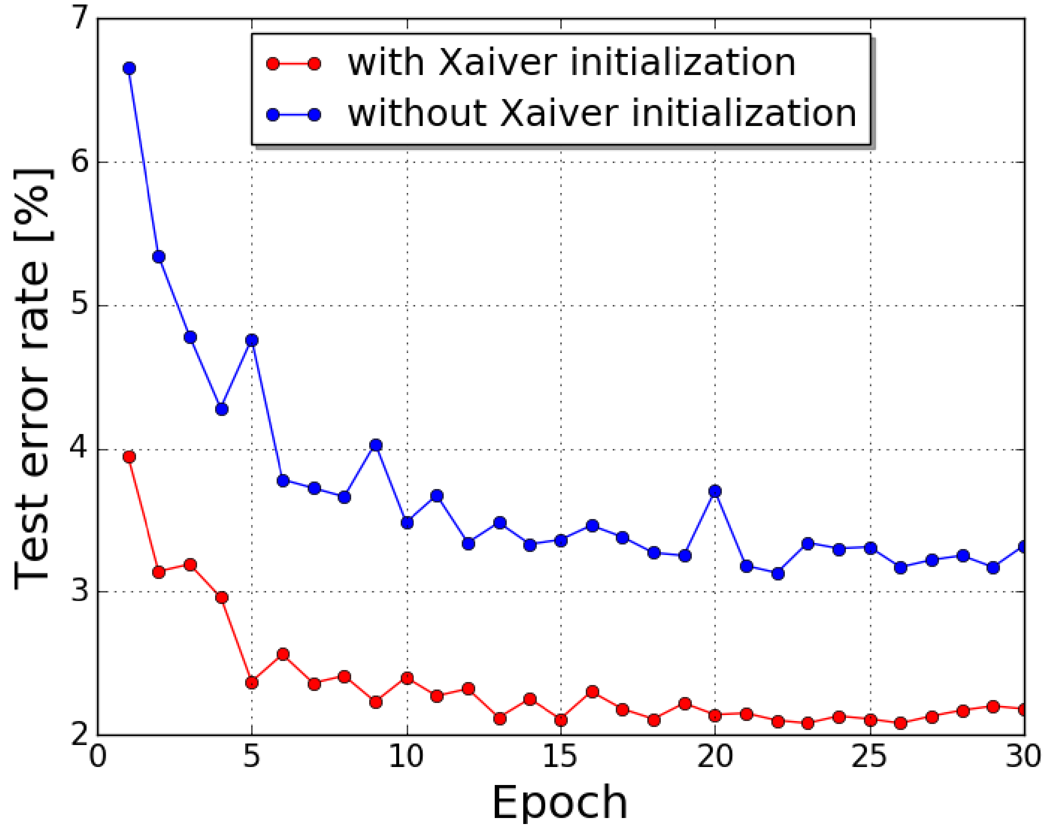
Weight initialization

Initializing with normalized Gaussians

n = 30



n = 100



Baseline : example2.py

example6.py

How to choose a neural network's hyper-parameters?

Parameters are optimized by ML, and other parameters are called hyper-parameters

Hyper-parameters specifying a function of family F

- Number of hidden layers
- Number of hidden neurons at each layer
- Type of neuron
- …

Hyper-parameters on training

- Type of cost-function
- Learning rate
- Regularization parameter
- Mini-batch size
- Number of epochs
- …

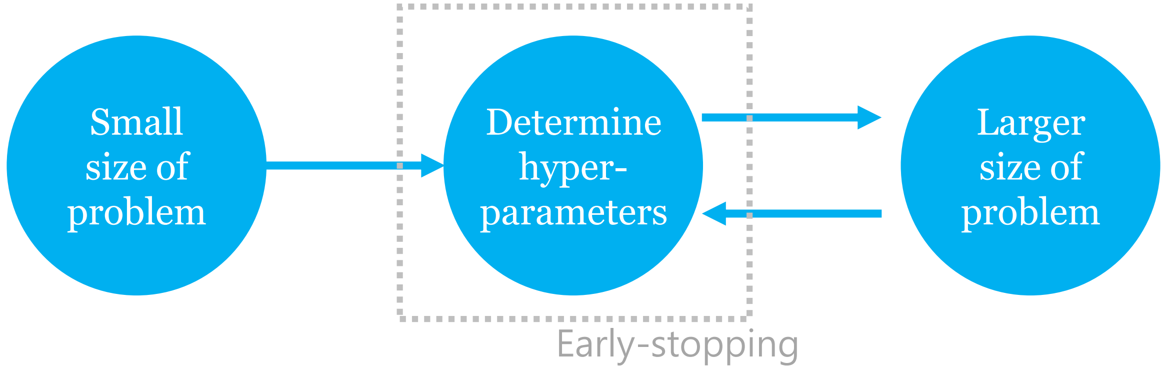
How to choose a neural network's hyper-parameters?

Broad strategy

START : get *any* non-trivial learning, i.e., for the network to achieve results better than chance

TIP. Reduce the size of problem : Get quick feedback from experiments

→ Two output neurons for '0' and '1'
→ Eliminate the hidden layer
→ Using a part of training images
(regularization parameter must be changed to keep the weight decay the same) $\eta\lambda/n = \eta\lambda'/n'$
→ Using a part of validation images



How to choose a neural network's hyper-parameters?

Use early stopping to determine the number of training epochs

Early stopping means that at the end of each epoch we should compute the classification accuracy on the validation data. When that stops improving, terminate.

Pros. 1 : we don't need to worry about explicitly figuring out how the number of epochs depends on the other hyper-parameters.

Pros. 2 : early stopping also automatically prevents us from overfitting.

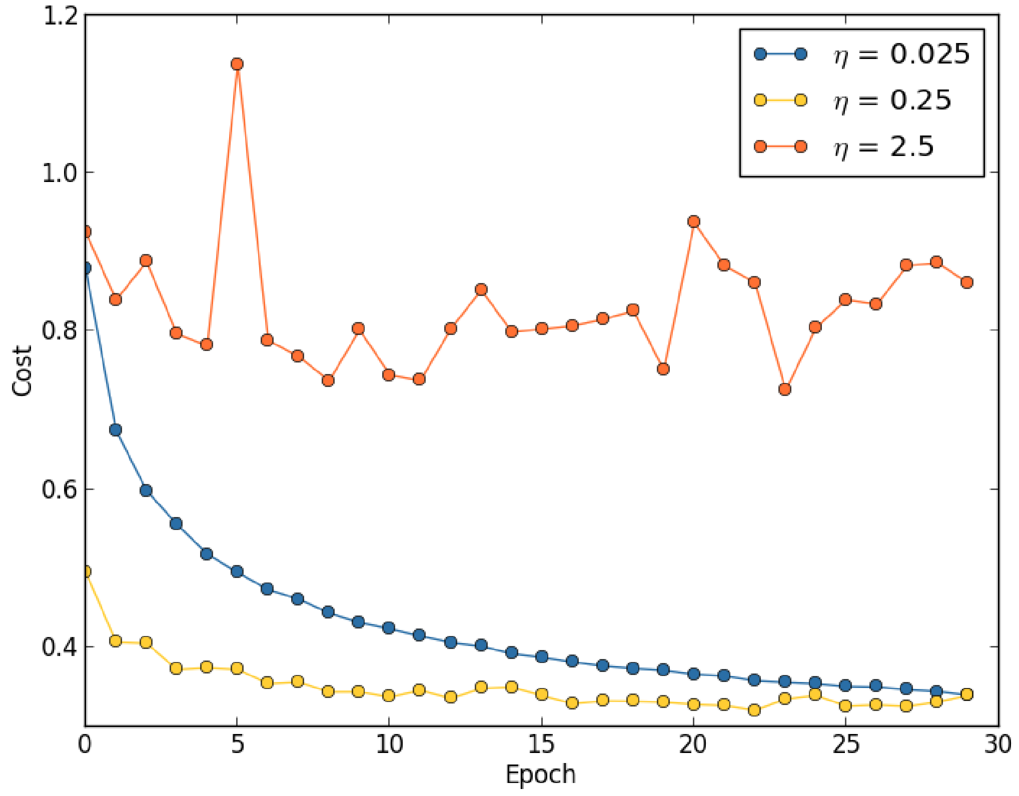
Early stopping is usually done by two rules.

Rule 1. A rule is to terminate if the best classification accuracy doesn't improve for quite some time. (like no-improvement-in-ten rule)

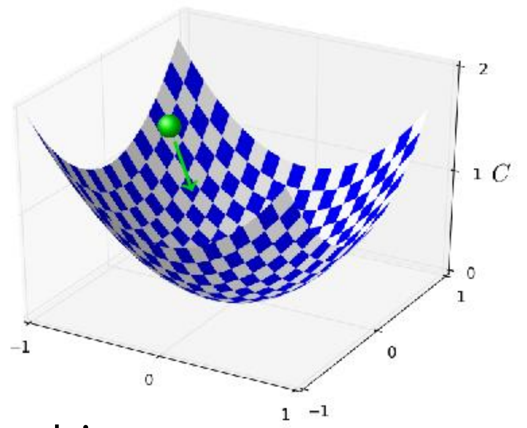
Rule 2. increase the number of limit no-improvement epochs as time goes by (like no-improvement-in-ten → in-twenty → in-fifty and so on)

How to choose a neural network's hyper-parameters?

Learning rate



Remember that learning rate is used for gradient descent algorithm.



If it is used in a proper way, cost must be gradually decreased.
(η=2.5 is not, which means too big!)

1. Find η at which the cost on the training data immediately begins decreasing, instead of oscillating or increasing (It determines the order of magnitude)
   ➔ This gives some range of η like 10^-1 ~ 10^2
2. Refinement η for the larger epochs

How to choose a neural network's hyper-parameters?

Learning rate schedule

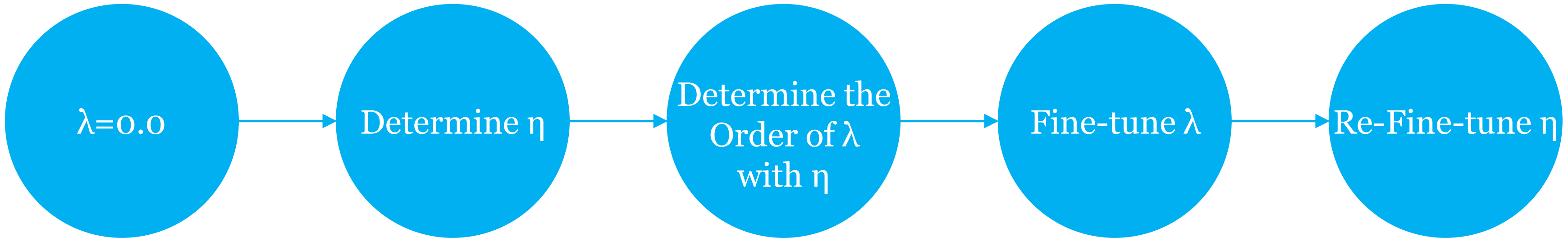
- Learning rate is not a constant. It changes during the learning process.
- One natural approach is to use the same basic idea as early stopping.
- The idea is to hold the learning rate constant until the validation accuracy starts to get worse. Then decrease the learning rate by some amount, say a factor of two or ten.
- We repeat this many times, until, say, the learning rate is a factor of 1,024 (or 1,000) times lower than the initial value. Then we terminate.

How to choose a neural network's hyper-parameters?

The regularization parameter

- I suggest starting initially with no regularization ($\lambda=0.0$), and determining a value for η, as above. Using that choice of η, we can then use the validation data to select a good value for λ. Start by trialling $\lambda=1.0$, and then increase or decrease by factors of 10, as needed to improve performance on the validation data.

- Once you've found a good order of magnitude, you can fine tune your value of λ. That done, you should return and re-optimize η again.



How to choose a neural network's hyper-parameters?

Mini-batch size

- Mini-batch is designed to reduce computational loads of Gradient-Descent Algorithm. (also thanks to matrix-computation)
- All we need is an estimate accurate enough that our cost function tends to keep decreasing.
- Too small, and you don't get to take full advantage of the benefits of good matrix libraries optimized for fast hardware.
- Too large and you're simply not updating your weights often enough.
- Fortunately, the choice of mini-batch size at which the speed is maximized is relatively independent of the other hyper-parameters. Thus, other parameters are first optimized, then mini-batch size is on concern.
- The plot with real elapsed time vs validation accuracy would be helpful.

How to choose a neural network's hyper-parameters?

Automated techniques

There are some automated techniques of determining hyper-parameters.

1. Grid search : it systematically searches through a grid in hyper-parameter space

   Random search for hyper-parameter optimization, by James Bergstra and Yoshua Bengio (2012).

2. Bayesian approach

   Practical Bayesian optimization of machine learning algorithms, by Jasper Snoek, Hugo Larochelle, and Ryan Adams.

   https://github.com/jaberg/hyperopt

Variations on stochastic gradient descent

Hessian technique

- Cost function : C(w)
- Taylor's approximation : $C(w + \Delta w) \approx C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w$
- The optimal solution : $\Delta w = -H^{-1} \nabla C$.
- The learning process : $w_{k+1} = w_k - \eta H^{-1} \nabla C$

- ∇C: gradient of C
- H : Hessian of C

- η: learning rate

While there are some back-propagation algorithms for Hessian technique, it has one very undesirable property : the sheer size of the Hessian matrix

If you have a neural network with 10^7 weights and biases, the corresponding Hessian matrix will 10^7 x 10^7 = 10^{14} , and $H^{-1}\nabla C$ must be computed…

Advantage Hessian optimization has is that it incorporates not just information about the gradient, but also information about how the gradient is changing.

Variations on stochastic gradient descent

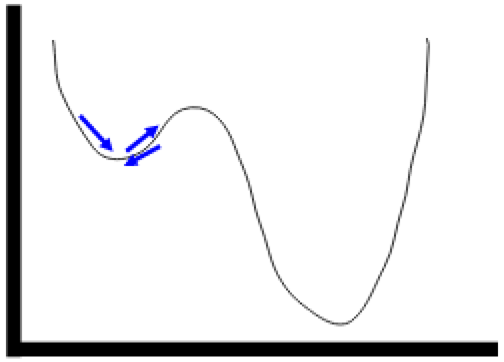
Momentum-based gradient descent

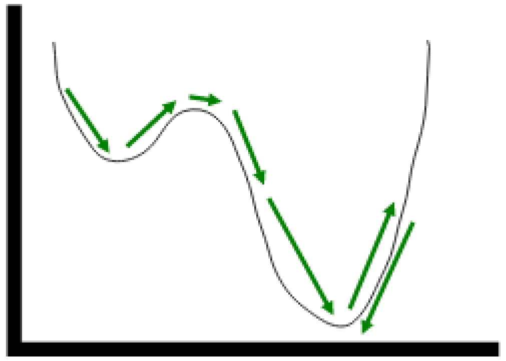
The momentum technique modifies gradient descent in two ways that make it more similar to the physical picture.

1. Velocity : gradient does not change position but velocity
2. Friction term : to gradually reduce the velocity

Gradient Descent	Momentum-based
$w \rightarrow w' = w - \eta \nabla C$	$v \rightarrow v' = \mu v - \eta \nabla C$ $w \rightarrow w' = w + v'.$

- μ : friction term or momentum coefficient





Adds a percentage of the last movement to the current movement

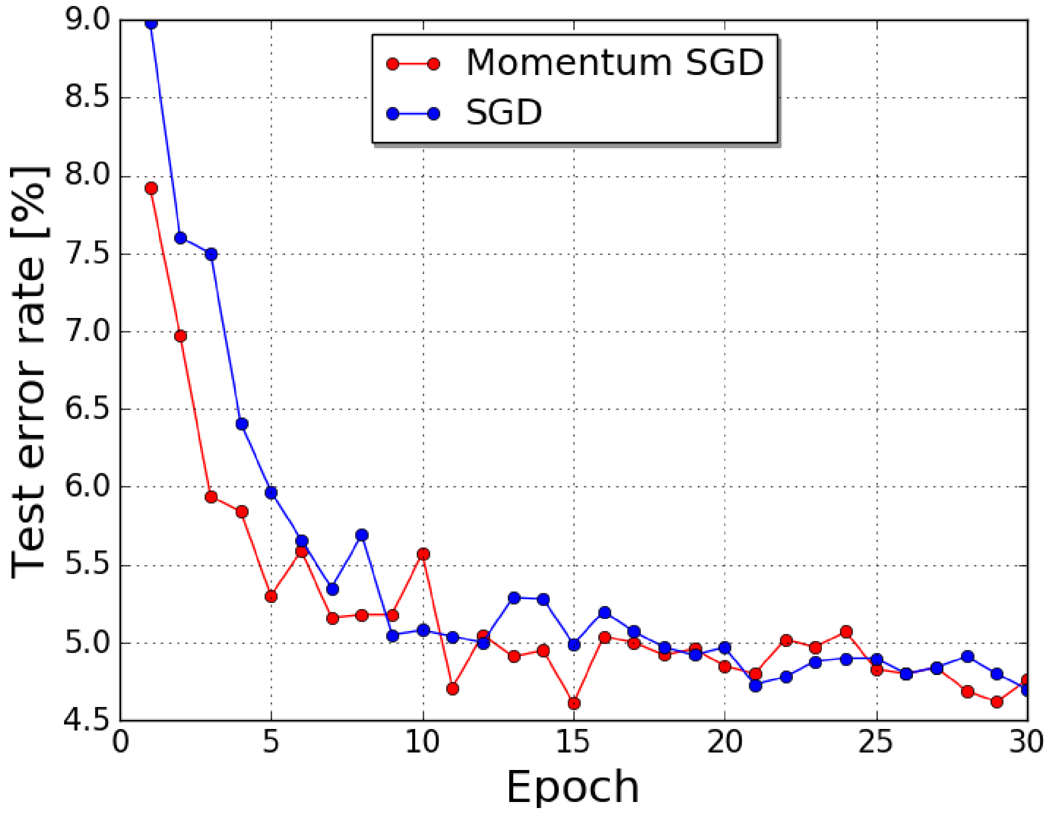
We can use the same back-propagation scheme as the GD

example7.py

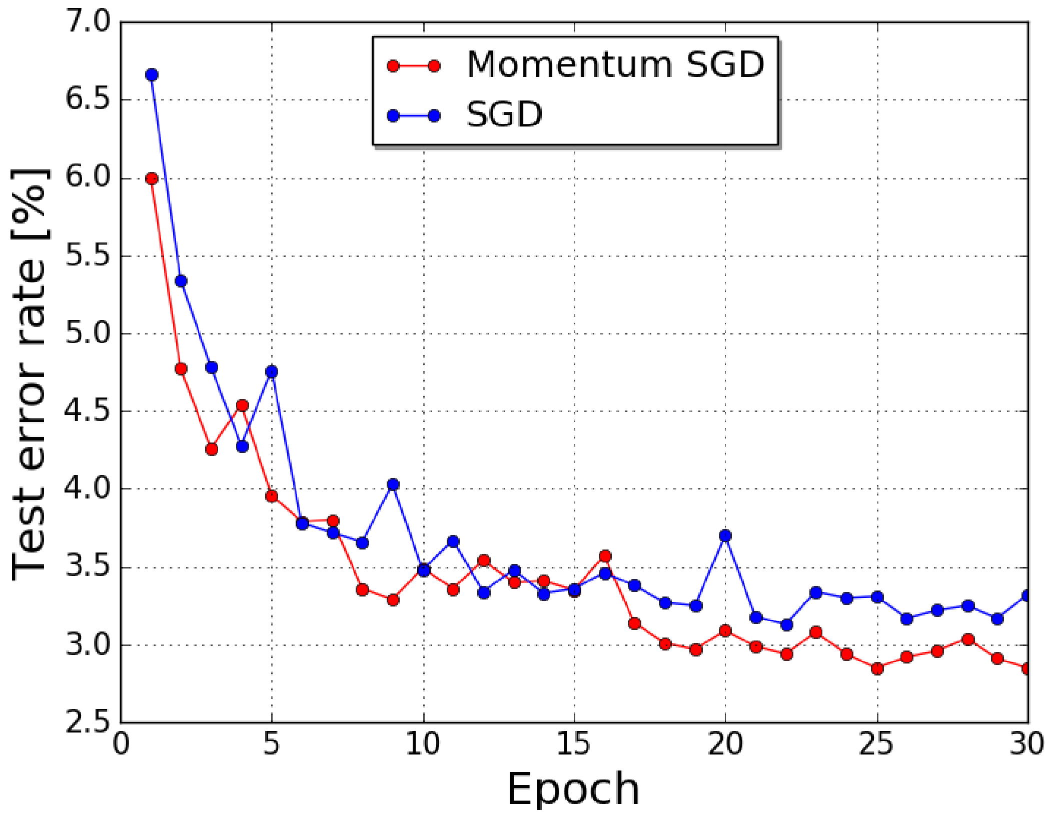
Variations on stochastic gradient descent

Momentum-based gradient descent

n = 30



n = 100



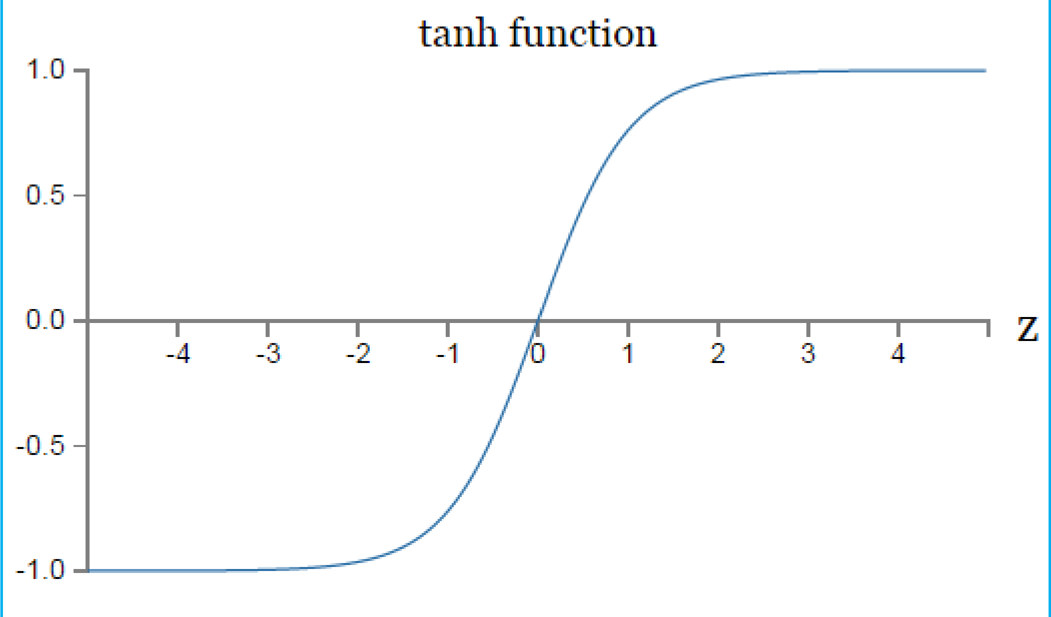
Baseline : example2.py

example7.py

Other models of artificial neuron

Tanh neuron

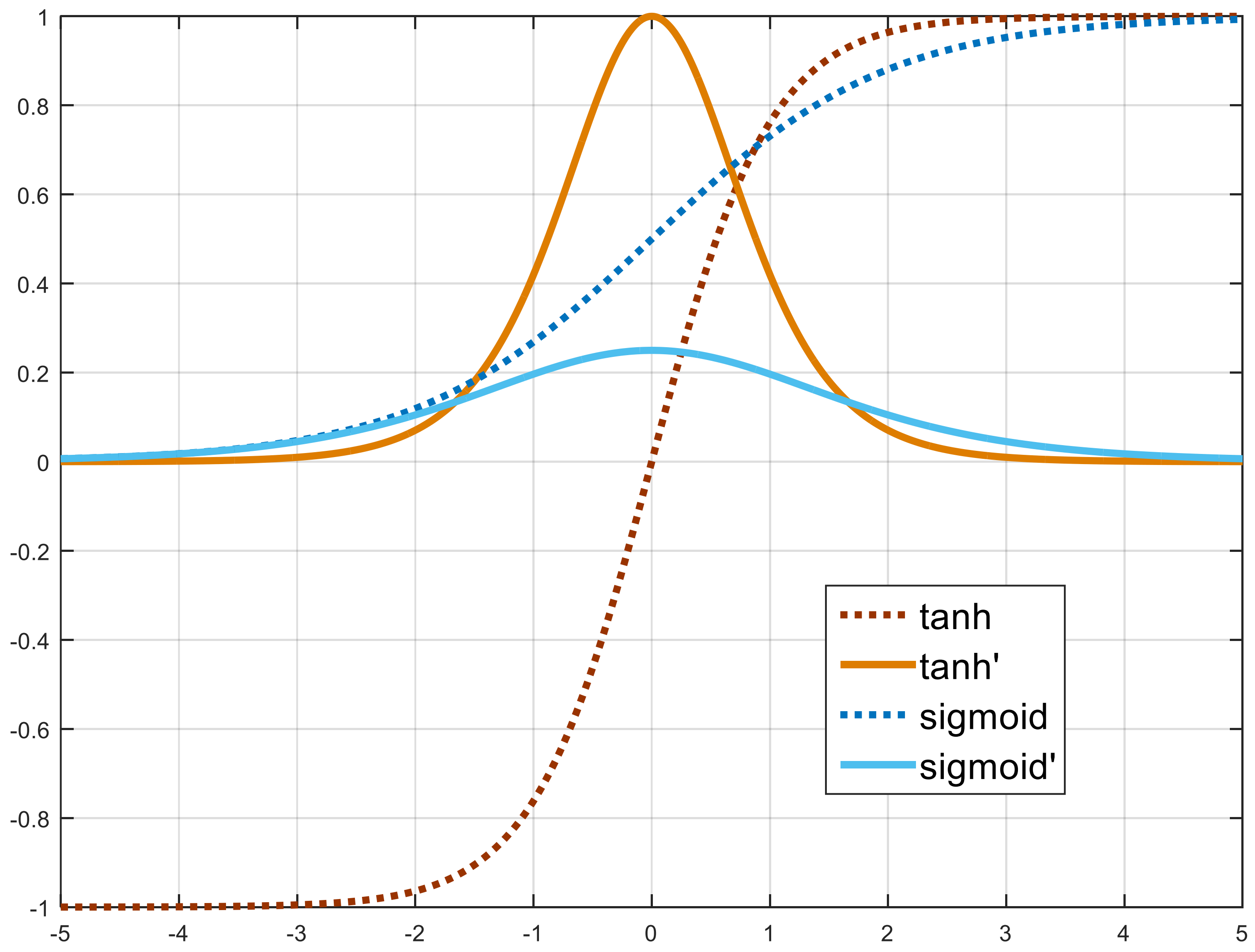
$$tanh(z) = 2 \cdot \sigma(2z) - 1$$



- One difference between tanh neurons and sigmoid neurons is that the output from tanh neurons ranges from -1 to 1, not 0 to 1.
- This means that if you're going to build a network based on tanh neurons you may need to normalize your outputs(and, depending on the details of the application, possibly your inputs) a little differently than in sigmoid networks.

Other models of artificial neuron

Tanh neuron



Feature 1

Fast learning

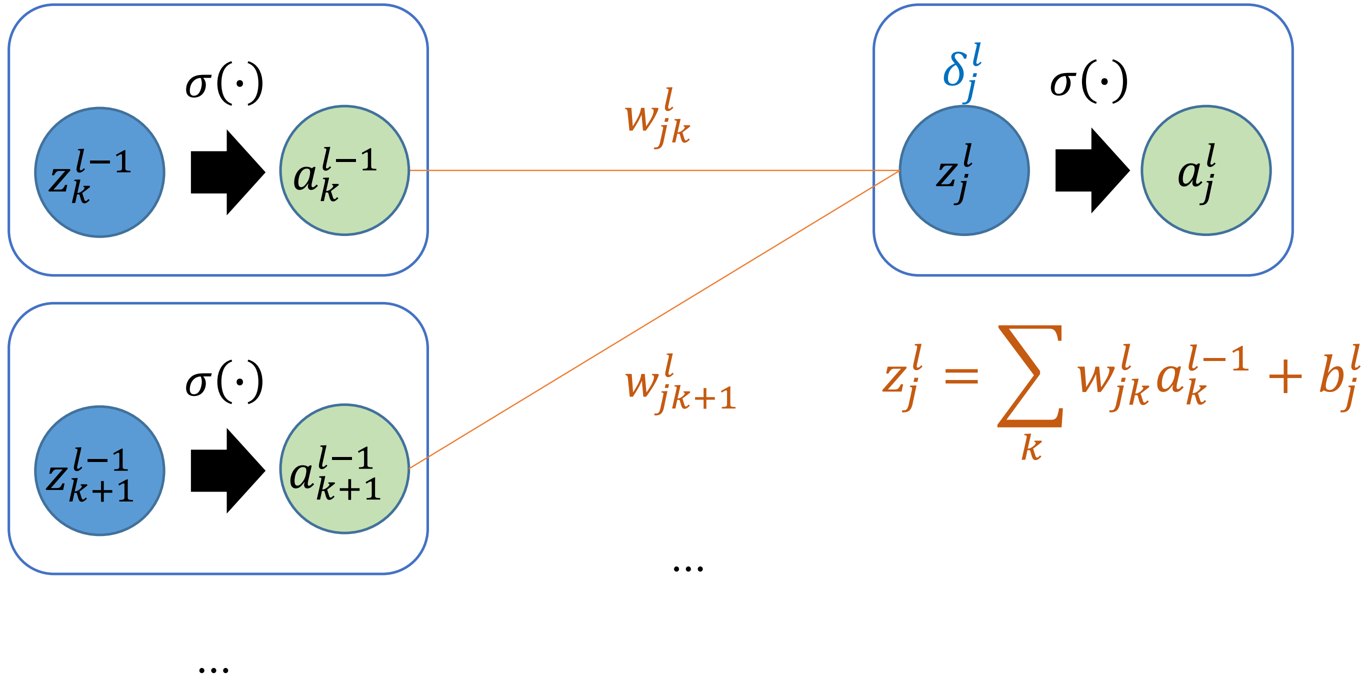
Tanh neuron has usually larger amount of derivative as compared with sigmoid neuron, which makes fast learning

Other models of artificial neuron

Tanh neuron

$$\frac{\partial C}{\partial w_{jk}^{l}} = \delta_j^l a_k^{l-1}$$

- $a_k^{l-1} > 0$ for sigmoid neuron
- Sign of gradient is the same as sign of δ_j^l



$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

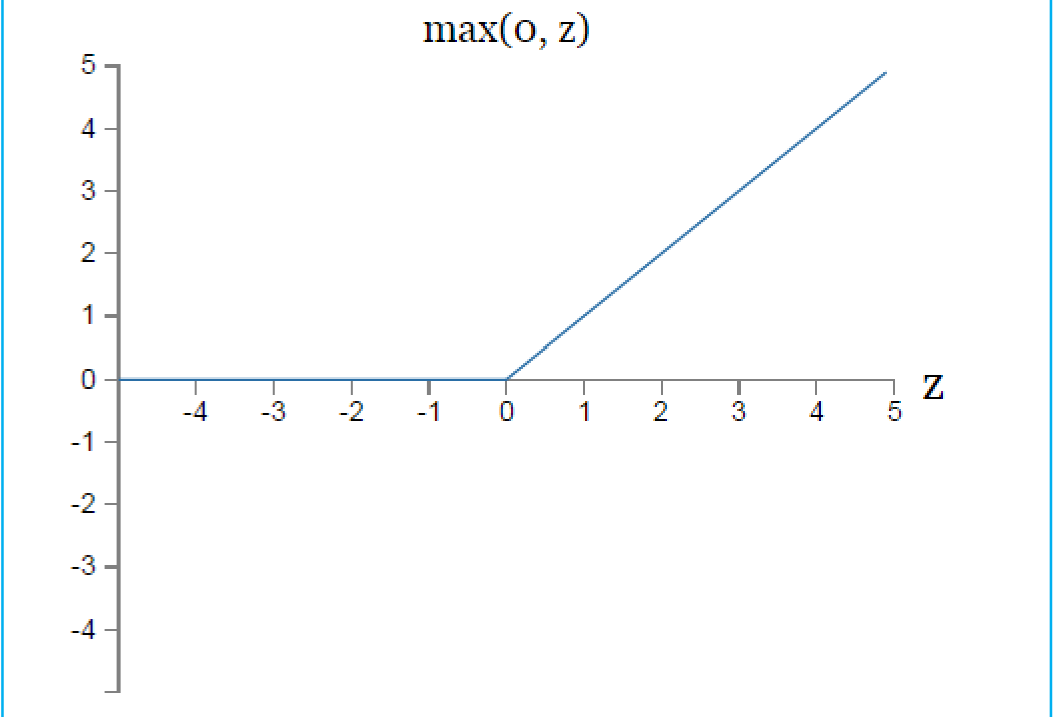
Feature 2 Independent behavior between weights

- If $\delta_j^l > 0$ then *all* the weights w_{jk}^l will decrease during gradient descent,
- If $\delta_j^l < 0$ then *all* the weights w_{jk}^l will increase during gradient descent.
- In other words, all weights to the same neuron must either increase together or decrease together.
- The case that some of the weights increase while others decrease can only happen if some of the input activations have different signs. (Tanh neuron does)

Other models of artificial neuron

Rectified linear neuron (Rectified Linear Unit, ReLU)

$$ReLU(z) = max(0, z)$$

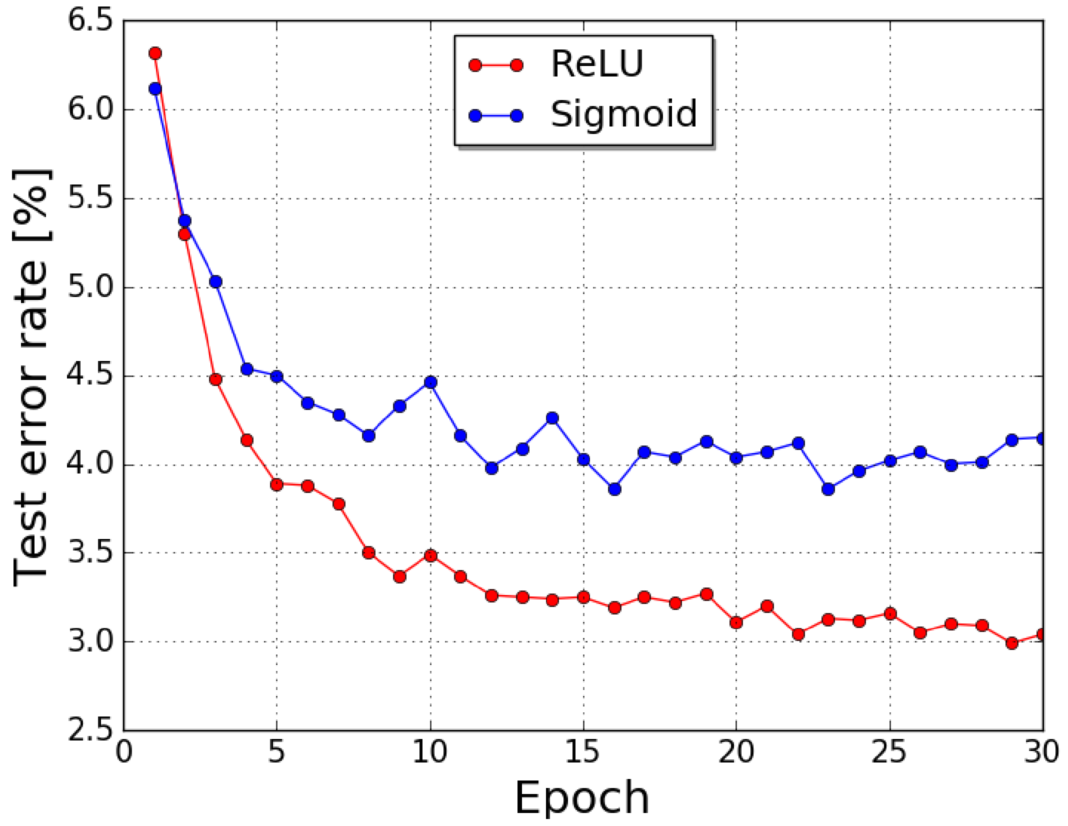


- Some recent work on image recognition has found considerable benefit in using rectified linear units through much of the network.
- However, as with tanh neurons, we do not yet have a really deep understanding of when, exactly, rectified linear units are preferable, nor why.
- Sigmoid and tanh neurons stop learning when they saturate, and it causes slow learning
- ReLU never saturates when the weighted input is positive. On the other hand when the weighted input in negative, the gradient vanishes, and so the neuron stops learning entirely.

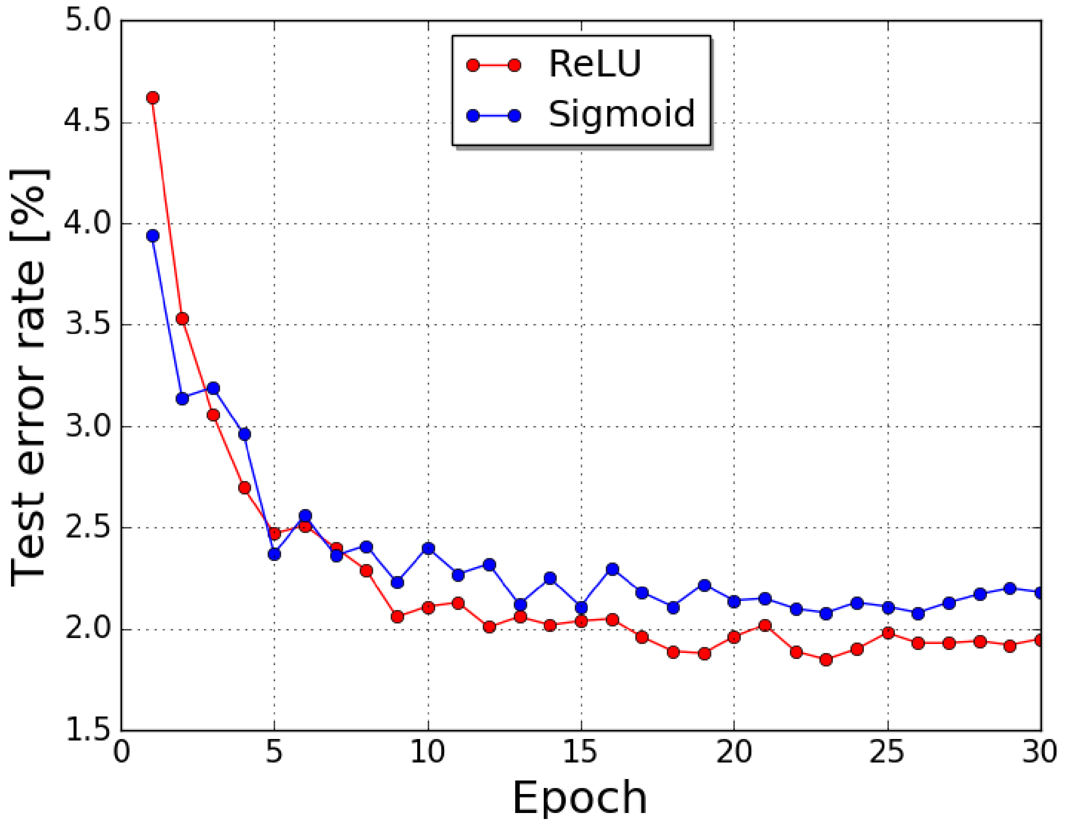
Other models of artificial neuron

Rectified linear neuron (Rectified Linear Unit, ReLU)

n = 30



n = 100



Baseline : example6.py

example8.py

On stories in neural networks

One question for LeCun

Question: *How do you approach utilizing and researching machine learning techniques that are supported almost entirely empirically, as opposed to mathematically? Also in what situations have you noticed some of these techniques fail?*

Answer: You have to realize that our theoretical tools are very weak. Sometimes, we have good mathematical intuitions for why a particular technique should work. Sometimes our intuition ends up being wrong [...] The questions become: how well does my method work on this particular problem, and how large is the set of problems on which it works well.

- *Question and answer with neural networks researcher Yann LeCun*

In many parts of science - especially those parts that deal with simple phenomena - it's possible to obtain very solid, very reliable evidence for quite general hypotheses. But in neural networks there are large numbers of parameters and hyper-parameters, and extremely complex interactions between them. In such extraordinarily complex systems it's exceedingly difficult to establish reliable general statements. Understanding neural networks in their full generality is a problem that, like quantum foundations, tests the limits of the human mind.

Numpy Packages

Package 1 : https://github.com/mnielsen/neural-networks-and-deep-learning

Package 2 : https://github.com/cthorey/CS231/tree/master/assignment2

Providing adam optimizer, batch normalization, convolutional layer, etc