

CSC 578

Neural Networks and Deep Learning

3. Improving Neural Networks

(Some figures adapted from [NNDL book](#))

Various Approaches to Improve Neural Networks

1. Activation/Transfer functions
2. Cost/Loss functions
3. Regularization
4. Initial Weights
5. Hyperparameter Optimization
6. Other Cost-minimization Techniques
7. Two Problems

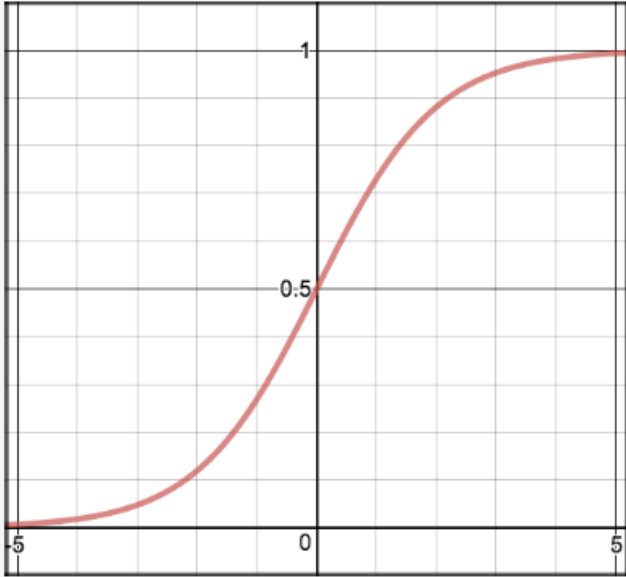
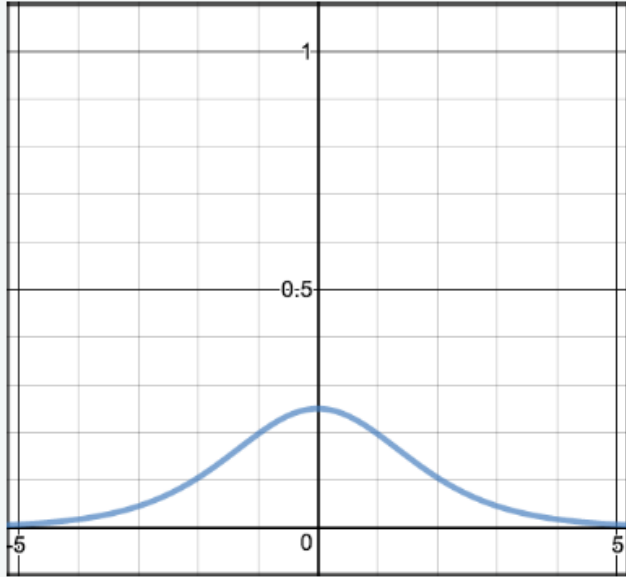
1 Activation/Transfer Functions

- There are several activation functions used in neural networks, including:
 1. Sigmoid
 2. Linear
 3. Tanh
 4. Rectified Linear Unit (ReLU)
 5. Softmax

A good reference on various activation functions is found at https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html?highlight=softmax

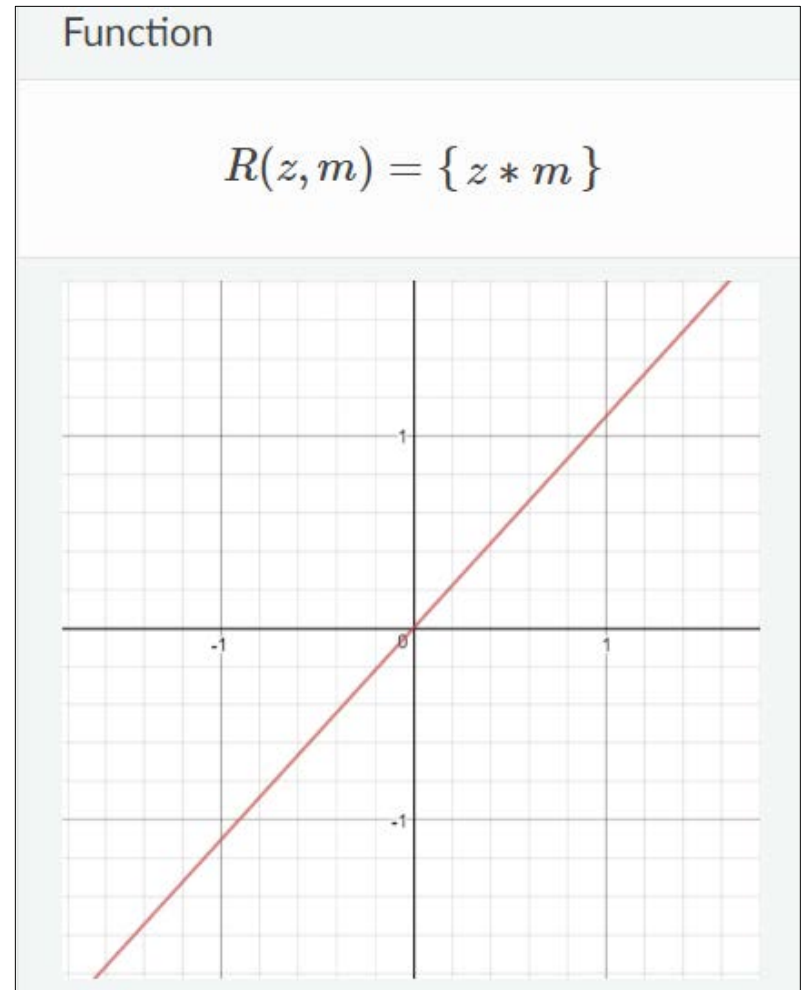
1.1 Sigmoid (or Logistic)

Sigmoid takes a real value as input and outputs another value between 0 and 1. It's easy to work with and has all the nice properties of activation functions: it's non-linear, continuously differentiable, monotonic, and has a fixed output range.

Function	Derivative
$S(z) = \frac{1}{1 + e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$
	

1.2 Linear

- No transformation is applied after weighted sum.
- Derivative is constant (irrespective of x):
Let $y = f(x) = z = \sum w \cdot x + b$
 $y' = f'(x) = w$
- Linear function is usually used for **regression** problems.

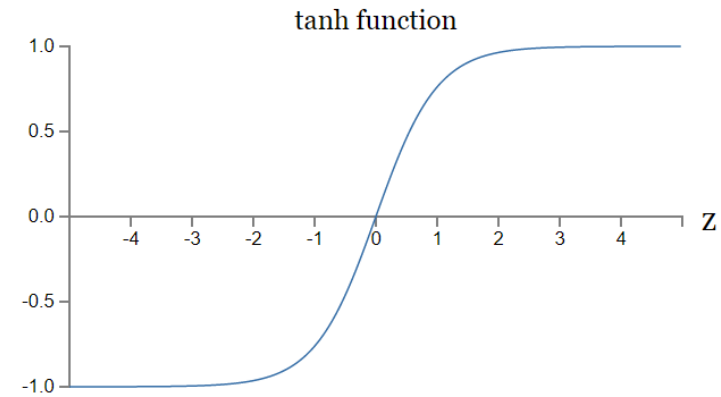


1.3 Tanh (Hyperbolic Tangent)

- Definition: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- The range of tanh is between -1 and 1 (rather than 0 and 1).
- Tanh is also a transformation

of sigmoid: $\sigma(z) = \frac{1 + \tanh\left(\frac{z}{2}\right)}{2}$.

And the output can be linearly rescaled to [0,1] by $0.5 * (\tanh(z) + 1)$.



- Derivative: Let $y = f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
 $y' = f'(z) = 1 - f(z)^2$, with $f(0) = 0$

Note: Tanh's derivative is steeper than sigmoid, so oftentimes learning is faster with tanh. But the same 'vanishing gradient' problem still exists.

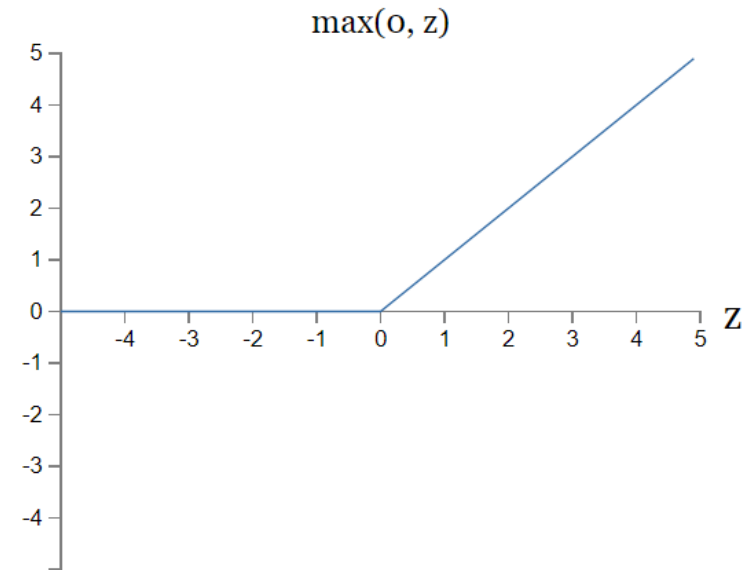
1.4 ReLU (Rectified Linear Unit)

- Definition: $f(z) = z^+ = \max(0, z)$

- This function is also called a Ramp function.
- Advantage -- Neurons **do not saturate** when $z > 0$.
- Disadvantage -- Gradient **vanishes** when $z < 0$.

- Derivative: $f'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$

But the function is technically NOT differentiable at 0.
=> 0 is arbitrarily substituted above.



1.5 Softmax

- Softmax function is usually applied to the **output layer only**.
- Also **when there is more than one node** on the output layer.
- Definition:

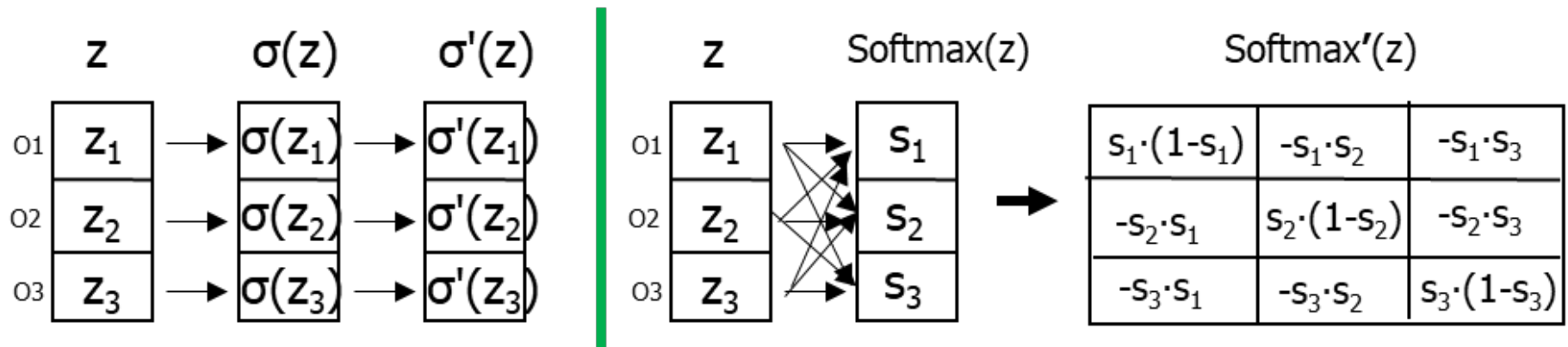
$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \quad \text{where } z_j^L = \sum_m w_{jm}^L \cdot a_m^{L-1} + b_j^L$$

- First the function ensures all values are positive by taking the exponent. Then it normalizes the resulting values and makes them a **probability distribution** (which sum up to 1).

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$$

Derivative of Softmax

- The derivative of Softmax (for a layer of node activations $a_1 \dots a_n$) is a 2D matrix, NOT a vector because the activation of a_j depends on all other activations ($\sum_k a_k$).



- Derivative: Let $s_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$

$$\frac{\partial s_i}{\partial a_j} = \begin{cases} s_i \cdot (1 - s_j) & \text{when } i = j \\ -s_i \cdot s_j & \text{when } i \neq j \end{cases}$$

References:

- [1] [Eli Bendersky](#)
- [2] [Aerin Kim](#)

2 Cost/Loss Functions

- For classification:

Quadratic	$C(w, b) \equiv \frac{1}{2n} \sum_x \ y(x) - a\ ^2$
Cross Entropy	$C = \frac{1}{n} \sum_x [-y(x) \cdot \ln(a) - (1 - y(x)) \cdot \ln(1 - a)]$
Likelihood	$C = \frac{1}{n} \sum_x -\ln(a_y^L) \dots$ usually used with Softmax activation function (for multiple output nodes)

- For regression:

Mean Squared Error (MSE)	$C = \frac{1}{n} \sum_x (y(x) - a)^2$ Or Root Mean Squared Error (RMSE)
--------------------------	--

Differentials of Cost Functions

General form (in gradient):

For a cost function **C** and an activation function **a**
(and **z** is the weighted sum, $z = \sum w_i \cdot x_i$),

$$\frac{\partial \mathbf{C}}{\partial w_i} = \frac{\partial \mathbf{C}}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial w_i}$$

- Quadratic:

$$\begin{aligned}\frac{\partial \mathbf{C}}{\partial w_i} &= \frac{\partial \mathbf{C}}{\partial a} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\ &= \frac{\partial}{\partial a} \left\{ \frac{1}{2n} \sum_x (y(x) - a)^2 \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\ &= \frac{1}{2n} \cdot \sum_x \left\{ \frac{\partial}{\partial a} (y(x) - a)^2 \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\ &= \frac{1}{n} \cdot \sum_x \{ -(y(x) - a) \} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right]\end{aligned}$$

Where x is an instance in the dataset, which contains n instances.

- Cross Entropy:

$$\begin{aligned}
\frac{\partial \mathcal{C}}{\partial w_i} &= \frac{\partial \mathcal{C}}{\partial a} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
&= \frac{\partial}{\partial a} \left\{ \frac{1}{n} \sum_x [-y(x) \cdot \ln a - (1 - y(x)) \cdot \ln(1 - a)] \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
&= \frac{1}{n} \cdot \sum_x - \left\{ \frac{\partial}{\partial a} [y(x) \cdot \ln a] + \frac{\partial}{\partial a} [(1 - y(x)) \cdot \ln(1 - a)] \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
&= \frac{1}{n} \cdot \sum_x - \left\{ \frac{y(x)}{a} + \left[(1 - y(x)) \cdot \frac{\partial}{\partial a} \ln(1 - a) \right] \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
&= \frac{1}{n} \cdot \sum_x - \left\{ \frac{y(x)}{a} + \frac{(1 - y(x))}{(1 - a)} \cdot (-1) \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
&= \frac{1}{n} \cdot \sum_x \left\{ \frac{a - y(x)}{a \cdot (1 - a)} \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right]
\end{aligned}$$

- Log Likelihood:

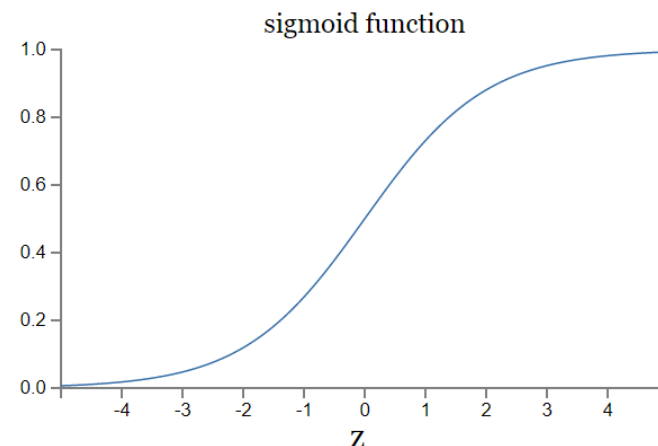
$$\begin{aligned}
 \frac{\partial \mathcal{C}}{\partial w_i} &= \frac{\partial \mathcal{C}}{\partial a} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
 &= \frac{\partial}{\partial a} \left\{ \frac{1}{n} \sum_x [-\ln(a_y^L)] \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
 &= \frac{1}{n} \cdot \sum_x \frac{\partial}{\partial a} \{-\ln(a_y^L)\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right] \\
 &= \frac{1}{n} \cdot \sum_x \left\{ -\frac{1}{a_y^L} \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right]
 \end{aligned}$$

where a_y^L is the y^{th} activation value **where y is the index of the target category**. The derivative values for **all other components are 0**.

A good reference on cross-entropy cost function is at
<https://www.ics.uci.edu/~pjsadows/notes.pdf> .

2.1 Cross Entropy

- Since sigmoid curve is flat on both ends, **cost functions** whose gradient include sigmoid is slow to converge (to minimize the cost), especially when the output is very far from the target.



- Using sigmoid neurons but with a different cost function, in particular **Cross Entropy**, helps speed up convergence.

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

- With Cross Entropy, C is always > 0 .

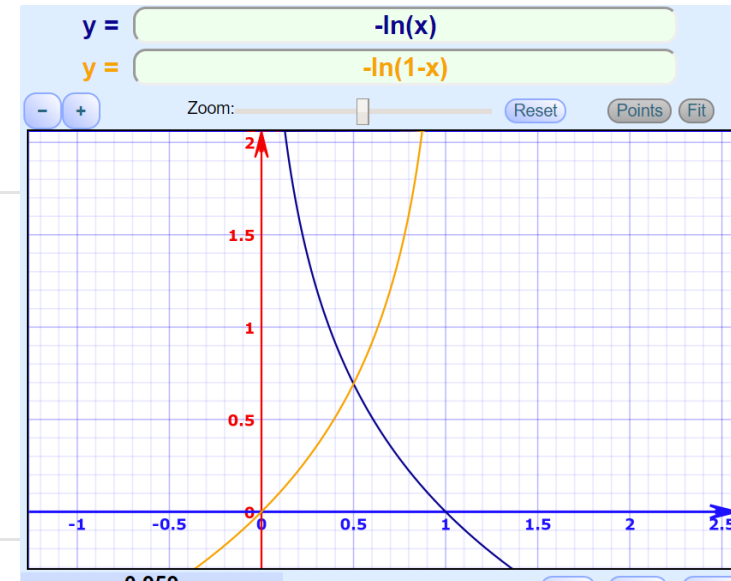
where y is the target output, and a is the network output, which may have multiple nodes.

- Cross-entropy cost (or ‘loss’) can be divided into two separate cost functions: one for $y=1$ and one for $y=0$.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_{\theta}(x), y) = -\log(h_{\theta}(x)) \quad \text{if } y = 1$$

$$\text{Cost}(h_{\theta}(x), y) = -\log(1 - h_{\theta}(x)) \quad \text{if } y = 0$$



- The two cases are combined into one expression (as a **vector notation**).

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

If $y=0$, the first side cancels out. If $y=1$, the second side cancels out. In both cases we only perform the operation we need to perform, **for each node in the output vector**.

- Differentials of the cost functions when $a = \sigma(z)$:

Quadratic	$\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial a} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_j} \right] = \frac{1}{n} \cdot \sum_x (\sigma(z) - y) \cdot [\sigma'(z) \cdot x_j]$
Cross-Entropy	$\begin{aligned} \frac{\partial C}{\partial w_j} &= \frac{\partial C}{\partial a} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_j} \right] = \frac{1}{n} \cdot \sum_x \frac{\sigma(z) - y}{\sigma(z)(1 - \sigma(z))} \cdot [\sigma'(z) \cdot x_j] \\ &= \frac{1}{n} \cdot \sum_x (\sigma(z) - y) \cdot [x_j] \end{aligned}$

- Since the gradient of Cross-Entropy's formula removes the term $\sigma'(z)$, the weight convergence could become faster.
- This also affects the BP algorithm, since the errors become different:
 - Step 3: Output error: $\delta^L = a^L - y$
 - Step 4: Backpropagate the error: $\delta^l = (w^{l+1})^T \cdot \delta^{l+1}$
 - Step 5: Weight update is the same: $\Delta w_{ji} = \eta \cdot \delta_j \cdot x_i$


```

42 class CrossEntropyCost(object):
43
44     @staticmethod
45     def fn(a, y):
46         """Return the cost associated with an output ``a`` and desired output
47         ``y``. Note that np.nan_to_num is used to ensure numerical
48         stability. In particular, if both ``a`` and ``y`` have a 1.0
49         in the same slot, then the expression (1-y)*np.log(1-a)
50         returns nan. The np.nan_to_num ensures that that is converted
51         to the correct value (0.0).
52
53         """
54         return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))
55
56     @staticmethod
57     def delta(z, a, y):
58         """Return the error delta from the output layer. Note that the
59         parameter ``z`` is not used by the method. It is included in
60         the method's parameters in order to make the interface
61         consistent with the delta method for other cost classes.
62
63         """
64         return (a-y)
65

```

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
... monitor_evaluation_accuracy=True)
```

You can specify the cost function in the constructor call (by “cost=...”).

2.2 Log Likelihood

- Log likelihood (or **Maximum Likelihood Estimate**) is essentially equivalent to cross entropy, but for multiple outcomes.

$$C = \frac{1}{n} \sum_x -\ln(a_k^L)$$

where a_k^L is the k^{th} activation value **where k is the index of the target category** (assuming a 'one-hot-vector' representation).

- In this case, the likelihood function is the **identity function** ($id(x) = x$, when x is a probability distribution $[0,1]$).
- Maximizing the likelihood (to estimate the parameter) is equivalent to minimizing the **negative log-likelihood** ($-\ln(id(a_k^L))$), which in this case is applied to minimize the cost generated by the network.

- Also conveniently, when the target value y is binary (0 or 1) and in the *one-hot-vector* format (having exactly one 1 for the target category), the formula fits well to a vector notation:

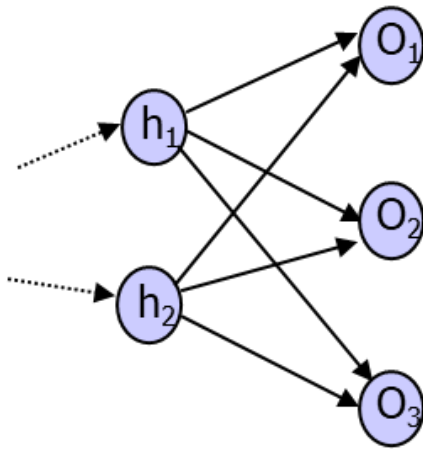
$$C = \frac{1}{n} \sum_x -y \cdot \ln(a^L)$$

- Softmax** activation + LogLikelihood cost function ($-\ln(a_k^L)$):

Softmax'(z)	•	C'	=	delta																
<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="padding: 5px;">$a_1 \cdot (1 - a_1)$</td> <td style="padding: 5px;">$-a_1 \cdot a_2$</td> <td style="padding: 5px;">$-a_1 \cdot a_3$</td> </tr> <tr> <td style="padding: 5px;">$-a_2 \cdot a_1$</td> <td style="padding: 5px;">$a_2 \cdot (1 - a_2)$</td> <td style="padding: 5px;">$-a_2 \cdot a_3$</td> </tr> <tr> <td style="padding: 5px;">$-a_3 \cdot a_1$</td> <td style="padding: 5px;">$-a_3 \cdot a_2$</td> <td style="padding: 5px;">$a_3 \cdot (1 - a_3)$</td> </tr> </table>	$a_1 \cdot (1 - a_1)$	$-a_1 \cdot a_2$	$-a_1 \cdot a_3$	$-a_2 \cdot a_1$	$a_2 \cdot (1 - a_2)$	$-a_2 \cdot a_3$	$-a_3 \cdot a_1$	$-a_3 \cdot a_2$	$a_3 \cdot (1 - a_3)$		<table style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 5px;">0</td></tr> <tr><td style="padding: 5px;">$-\frac{1}{a_2}$</td></tr> <tr><td style="padding: 5px;">0</td></tr> </table>	0	$-\frac{1}{a_2}$	0		<table style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 5px;">a_1</td></tr> <tr><td style="padding: 5px;">$a_2 - 1$</td></tr> <tr><td style="padding: 5px;">a_3</td></tr> </table>	a_1	$a_2 - 1$	a_3	When O_2 was the target output
$a_1 \cdot (1 - a_1)$	$-a_1 \cdot a_2$	$-a_1 \cdot a_3$																		
$-a_2 \cdot a_1$	$a_2 \cdot (1 - a_2)$	$-a_2 \cdot a_3$																		
$-a_3 \cdot a_1$	$-a_3 \cdot a_2$	$a_3 \cdot (1 - a_3)$																		
0																				
$-\frac{1}{a_2}$																				
0																				
a_1																				
$a_2 - 1$																				
a_3																				

and $\frac{\partial C}{\partial w_i} = \frac{1}{n} \cdot \sum_x \left\{ -\frac{1}{a_y^L} \right\} \cdot \left[\frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i} \right]$

$= \frac{1}{n} \cdot \sum_x \left\{ -\frac{1}{a_y^L} \right\} \cdot \{ a_y^L \cdot (1 - a_y^L) \} \cdot \left[\frac{\partial z}{\partial w_i} \right] = \frac{1}{n} \cdot \sum_x (a_y^L - 1) \cdot \left[\frac{\partial z}{\partial w_i} \right]$



Activation Functions

	z	Sigma	Tanh	ReLU	Softmax	e
O1	1.5	0.8176	0.9051	1.5	0.5775	4.4817
O2	0.9	0.7109	0.7163	0.9	0.317	2.4596
O3	-0.2	0.4502	-0.197	0	0.1055	0.8187
					sum	7.76

Sigmoid + Cost Functions

			yes	yes	no				NO	yes	YES
	y	a	Quadratic	CrossEnt	Likelihood		y	a	Quadratic	CrossEnt	Likelihood
	target	Sigma	(y-a)^2		-ln(a)		target	Softmax	(y-a)^2		-ln(a)
O1	0	0.8176	0.668428	1.701413	0	O1	0	0.57754	0.333547	0.86165	0
O2	1	0.7109	0.08355	0.341154	0.341154	O2	1	0.31696	0.466546	1.148985	1.148985
O3	0	0.4502	0.202649	0.598139	0	O3	0	0.10551	0.011132	0.111497	0
		sum	0.954628	2.640706	0.341154			sum	0.811225	2.122133	1.148985
		0.5*sum	0.477314								

3 Regularization Methods

- **Overfitting** is a big problem in Machine Learning.
- Regularization is applied in Neural Networks to avoid overfitting.
- There are several regularization methods, such as:
 - Weight decay / L2, L1 normalization
 - Dropout
 - Artificially enhancing the training data

3.1 L2 Regularization

- Weight decay.
- Add an extra term, a quadratic values of the weights, in the cost function:

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

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

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

- The extra term causes the gradient descent search to seek weight vectors with small magnitudes, thereby reducing the risk of overfitting (by biasing against complex model).

- Differentials of the cost function with L2 regularization.

- An extra term for weights: $\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w$
- No change for bias: $\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$.

- Weight update rule in the BP algorithm:

- A negative value $-\frac{\eta\lambda}{n} \cdot w$ in weights
=> **weight decay**:

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

- No change for bias:

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}$$


```

276
277 def update_mini_batch(self, mini_batch, eta, lambda, n):
278     """Update the network's weights and biases by applying gradient
279     descent using backpropagation to a single mini batch. The
280     ``mini_batch`` is a list of tuples ``(x, y)``, ``eta`` is the
281     learning rate, ``lambda`` is the regularization parameter, and
282     ``n`` is the total size of the training data set.
283
284     """
285     nabla_b = [np.zeros(b.shape) for b in self.biases]
286     nabla_w = [np.zeros(w.shape) for w in self.weights]
287     for x, y in mini_batch:
288         delta_nabla_b, delta_nabla_w = self.backprop(x, y)
289         nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
290         nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
291     self.weights = [(1-eta*(lambda/n))*w-(eta/len(mini_batch))*nw
292                     for w, nw in zip(self.weights, nabla_w)]
293     self.biases = [b-(eta/len(mini_batch))*nb
294                   for b, nb in zip(self.biases, nabla_b)]
295

```

```

274 def total_cost(self, data, lambda, convert=False):
275     """Return the total cost for the data set ``data``. The flag
276     ``convert`` should be set to False if the data set is the
277     training data (the usual case), and to True if the data set is
278     the validation or test data. See comments on the similar (but
279     reversed) convention for the ``accuracy`` method, above.
280     """
281     cost = 0.0
282     for x, y in data:
283         a = self.feedforward(x)
284         if convert: y = vectorized_result(y)
285         cost += self.cost.fn(a, y)/len(data)
286     cost += 0.5*(lambda/len(data))*sum(
287         np.linalg.norm(w)**2 for w in self.weights)
288     return cost
289

```

3.2 L1 Regularlization

- Similar to L1, but adds an extra term that is the absolute values of the weights in the cost function:

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

- Differentials of the cost function with L1 regularlization.
 - An extra term for weights:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \text{sgn}(w)$$

$$\text{where } \text{sgn}(w) = \begin{cases} +1 & \text{if } w > 0 \\ 0 & \text{if } w == 0 \\ -1 & \text{if } w < 0 \end{cases}$$

- Weight update rule in the BP algorithm:

- A negative value

$-\frac{\eta\lambda}{n} \cdot \text{sgn}(w)$ in weights

=> weight decay

- No change for bias

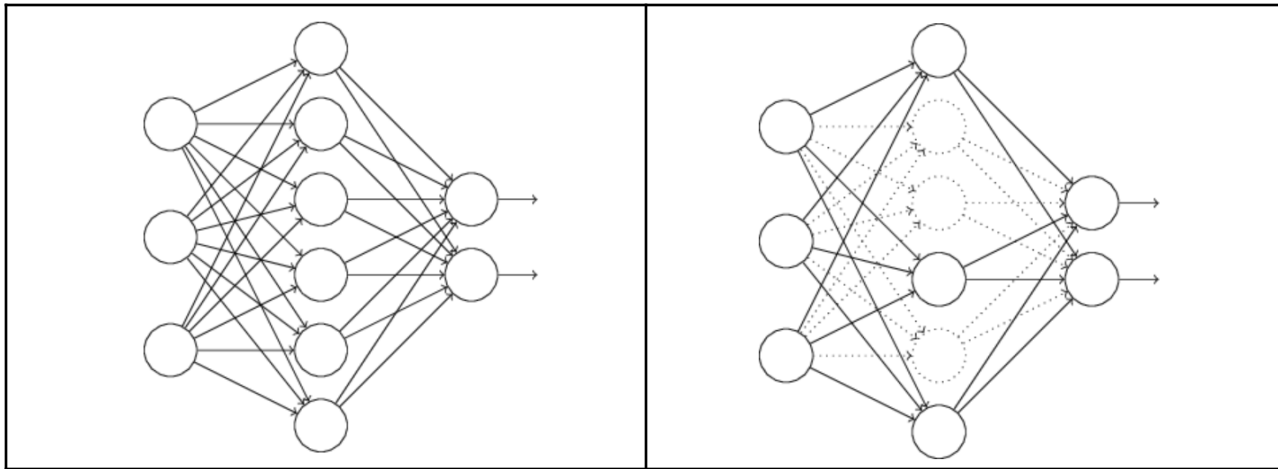
$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \text{sgn}(w) - \eta \frac{\partial C_0}{\partial w}$$

L2 vs. L1

- “In L1 regularization, the weights shrink by a constant amount toward 0.
- In L2 regularization, the weights shrink by an amount which is proportional to w .
- So when a particular weight has a large magnitude, $|w|$, L1 regularization shrinks the weight much less than L2 regularization does.
- By contrast, when $|w|$ is small, L1 regularization shrinks the weight much more than L2 regularization.
- The net result is that L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero.”

3.3 Dropout

- Dropout ‘modifies’ the network by periodically disconnecting some nodes in the network.



- “We start by **randomly** (and temporarily) **deleting half the hidden neurons in the network**, while leaving the input and output neurons untouched. We forward-propagate the input x through the modified network, and then backpropagate the result, also through the modified network. “
- “We then **repeat the process**, first restoring the dropout neurons, then **choosing a new random subset** of hidden neurons to delete,..”

- Robustness (and generalization power) of dropout:
 - “Heuristically, when we dropout different sets of neurons, it's rather like we're training different neural networks. And so the dropout procedure is like **averaging the effects of a very large number of different networks**. The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.”
 - “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**.”
- Dropout has been shown to be effective especially “in **training large, deep networks**, where the problem of overfitting is often acute.”

3.4 Artificial Expansion of Data

- In general, neural networks learn better with **larger** training data.
- We can expand the data by inserting artificially generated data instances obtained by **distorting** the original data.
- “The general principle is to expand the training data by applying operations that reflect **real-world variation**.”

4 Initial Weights

- Setting 'good values' for the initial weights could speed up the learning (convergence).
- Common values:
 - Uniform random (in a fixed range)
 - Gaussian ($\mu = 0$, $\sigma = 1$)
 - Gaussian ($\mu = 0$, $\sigma = 1$ for bias, $\frac{1}{\sqrt{n_{in}}}$ for weights on the input layer, where n_{in} is the number of nodes)

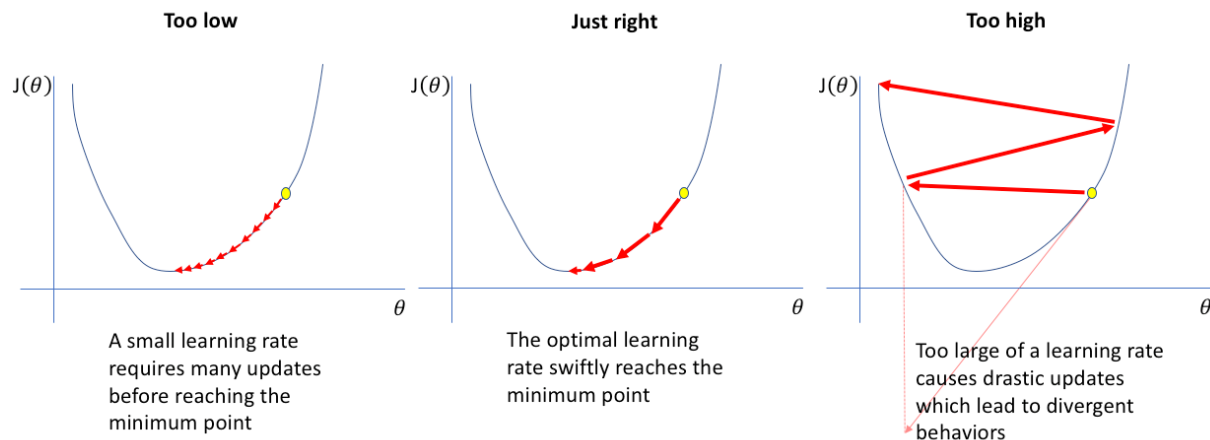
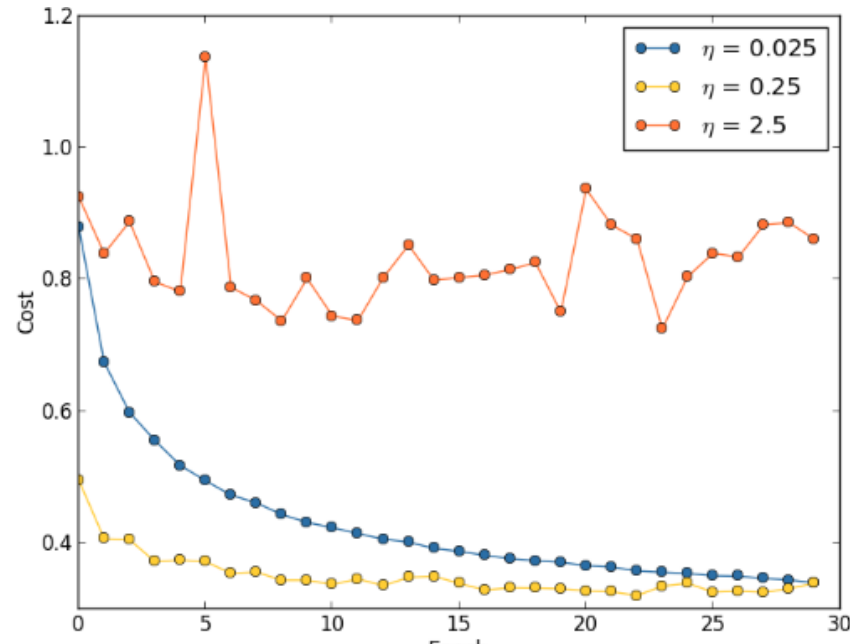
```
def default_weight_initializer(self):  
    self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]  
    self.weights = [np.random.randn(y, x)/np.sqrt(x)  
                     for x, y in zip(self.sizes[:-1], self.sizes[1:])]
```

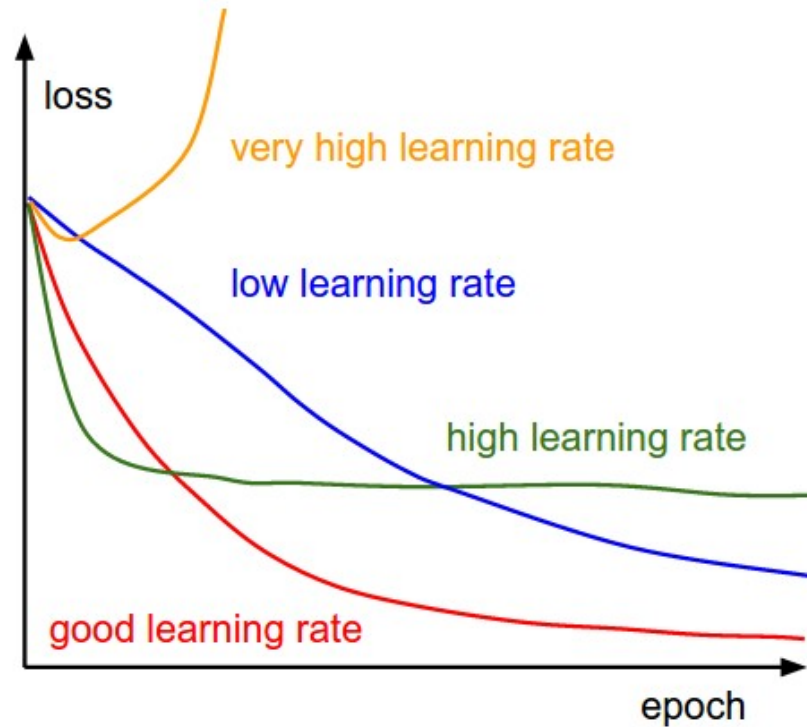
5 Hyperparameter Optimization

- How do we set ‘good values’ for hyper parameters such as η (learning rate), λ (regularization parameter) to speed up the experimentation?
- A broad strategy is to try to obtain a **signal** during the early stages of experiment that gives a promising result. From there, you tweak hyperparameters using heuristics.
 1. Strip the problem down
 - Reduce the data (i.e., simplify the problem)
 - Start with a simple network (i.e., simplify the architecture)
 2. Speed up testing by monitoring performance frequently
 3. Try rough variations of parameters
 4. Find a good threshold for learning rate
 5. Stop training when performance is good (or shows no improvement)
 6. Adaptive learning rate schedule

5.1 Learning Rate

- Try values of different magnitude, e.g. 0.025, 0.25, 2.5
- Monitor for oscillation and speed of convergence.

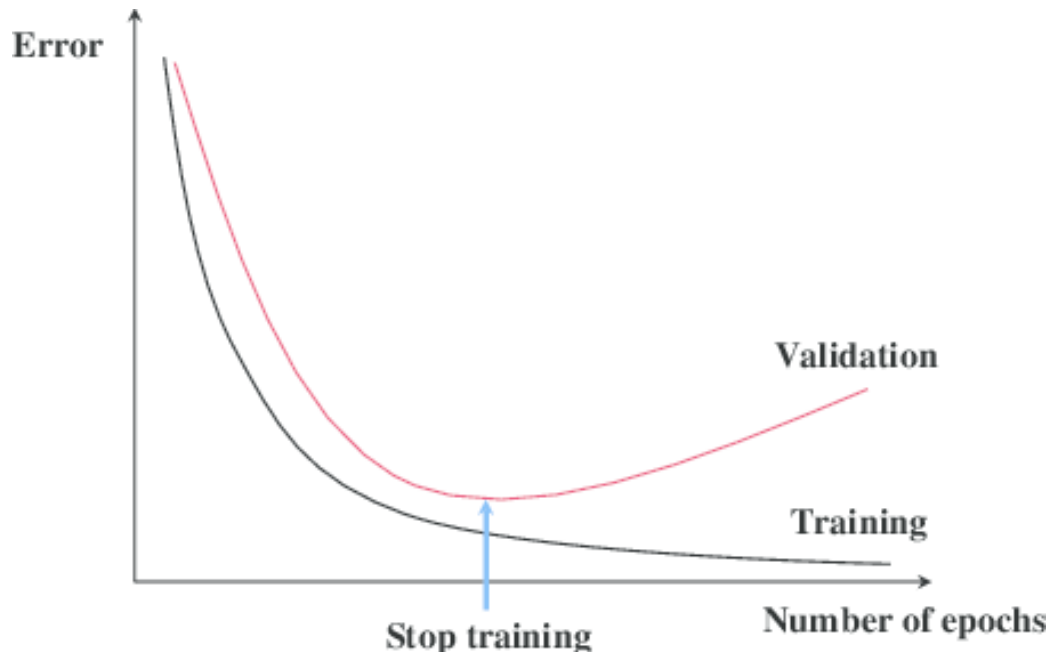




- You can also adjust the learning rate (by decreasing) later in the learning to narrowing in on the optimal solution (hopefully the global minimum).

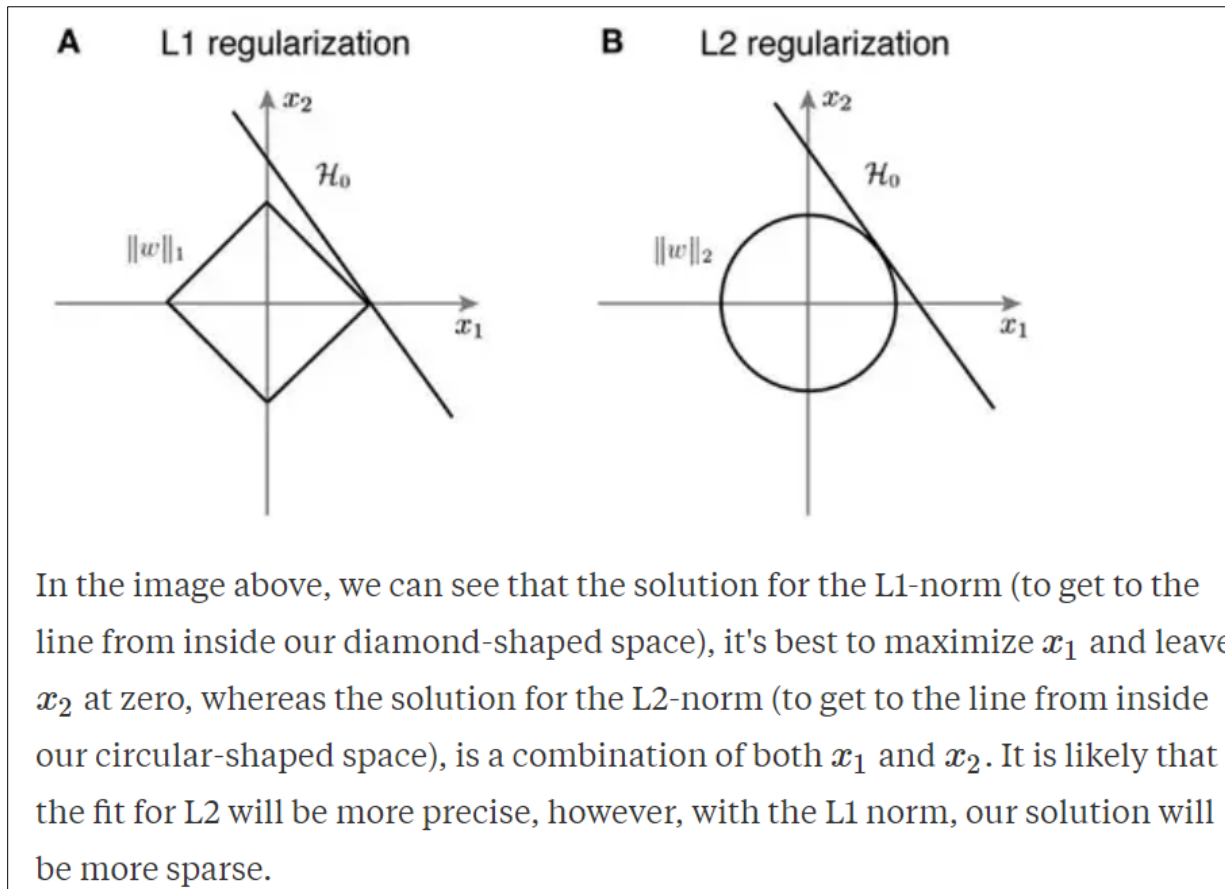
5.2 Early Stopping

- Stop training when **no improvement** is observed for some epochs (for the training set) – control the number of epochs.
- A common heuristics is ‘no-improvement-in-ten’.
- Another way is to use a **validation set** and monitor the learning for the set – to avoid overfitting.

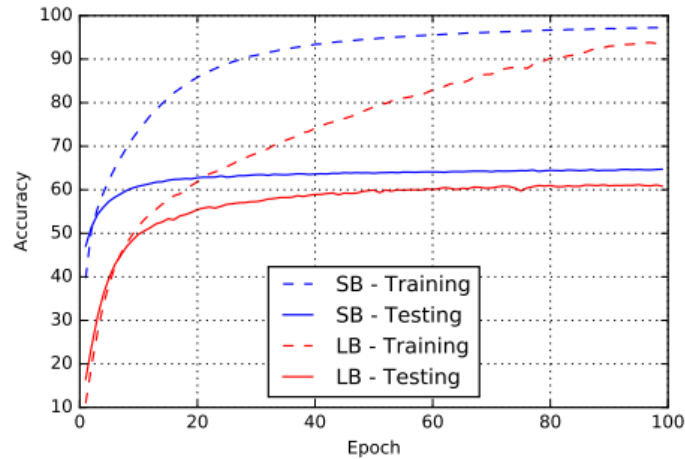


5.3 Regularization

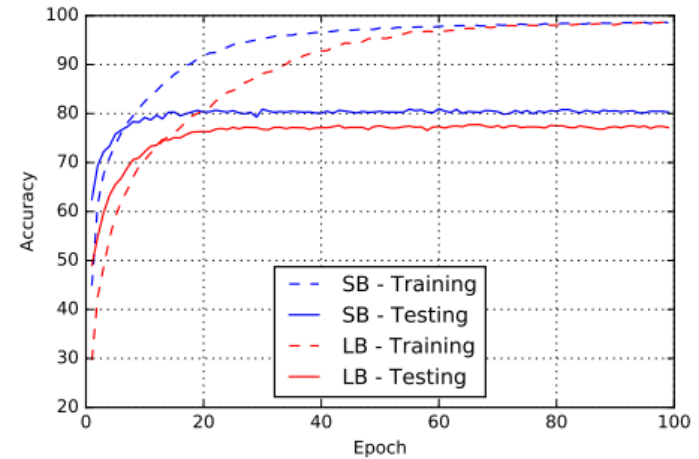
- Start with 0 and fix η . Then try 1.0, then * or / by 10.
- Note about [L1 vs. L2 regularizations](#):



5.4 Mini-Batch Size



(a) F_2



(b) C_1

Figure 2: Convergence trajectories of training and testing accuracy for SB and LB methods

Table 2: Performance of small-batch (SB) and large-batch (LB) variants of ADAM on the 6 networks listed in Table 1

Network Name	Training Accuracy		Testing Accuracy	
	SB	LB	SB	LB
F_1	99.66% \pm 0.05%	99.92% \pm 0.01%	98.03% \pm 0.07%	97.81% \pm 0.07%
F_2	99.99% \pm 0.03%	98.35% \pm 2.08%	64.02% \pm 0.2%	59.45% \pm 1.05%
C_1	99.89% \pm 0.02%	99.66% \pm 0.2%	80.04% \pm 0.12%	77.26% \pm 0.42%
C_2	99.99% \pm 0.04%	99.99 \pm 0.01%	89.24% \pm 0.12%	87.26% \pm 0.07%
C_3	99.56% \pm 0.44%	99.88% \pm 0.30%	49.58% \pm 0.39%	46.45% \pm 0.43%
C_4	99.10% \pm 1.23%	99.57% \pm 1.84%	63.08% \pm 0.5%	57.81% \pm 0.17%

5.5 Automated Techniques

- **Grid Search:** A grid is created based on parameter values. The do an exhaustive searching of all combinations in the grid.
- **Random Search:** Instead of trying all possible combinations as in Grid Search, only randomly selected subset of the parameters is tried.
- **Bayesian Optimization:** Gaussian Process uses a set of previously evaluated parameters and resulting accuracy to make an assumption about unobserved parameters.
- **Gradient-based Optimization:** Use the gradient with respect to hyperparameters and then optimize the hyperparameters using gradient descent.
- **Evolutionary Optimization:** Use evolutionary algorithms to search the space of hyperparameters for a given algorithm.

6 Other Cost-minimization Techniques

- In addition to *Stochastic* Gradient Descent, there are other techniques to minimize cost function:
 1. Hessian technique
 2. Momentum
 3. Other learning techniques

6.1 Hessian Technique

- The Hessian matrix or Hessian is a square matrix of **second-order** partial derivatives of a scalar-valued function (of many variables), $f: \mathbb{R}^n \rightarrow \mathbb{R}$.

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Hessian is used to approximate the minimization of cost function.
- First approximate $C(w)$ by $C(w) = C(w + \Delta w)$ at a point w (based on Taylor Series).

That gives

$$C(w + \Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w + \dots,$$

By dropping higher order terms above quadratic, we have

$$C(w + \Delta w) \approx C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w$$

Finally we set $\Delta w = -H^{-1} \nabla C$ to minimize $C(w + \Delta w)$.

- Then in the algorithm, the weight update becomes
$$w + \Delta w = w - H^{-1} \nabla C$$
- The Hessian technique incorporates information about **second-order changes in the cost function**, and has been shown to converge to a minimum faster.
- However, computing H at every step, for every weight, is computationally costly (with regard to memory as well computation time).

6.2 Momentum

- The **momentum** method is similar to Hessian in that it incorporates the second-order changes in the cost function as an analogical notion of *velocity*.
- “**Stochastic gradient descent with momentum** remembers the update Δw at each iteration, and determines the next update as a linear combination of the gradient ($\eta \cdot \nabla C$) and the previous update ($\alpha \cdot \Delta w$), that is,

$$\Delta w = \alpha \cdot \Delta w - \eta \cdot \nabla C$$

and the base weight update rule:

$$w = w + \Delta w$$

gives the final weight update formula

$$w + \Delta w = w + (\alpha \cdot \Delta w - \eta \cdot \nabla C)$$

[Wikipedia]

- Momentum adds extra term for the same direction, thereby preventing oscillation.

6.3 Other Learning Techniques

- **Resilient Propagation:**
 - Use only the magnitude of the gradient and allow each neuron to learn at its own rate. No need for learning rate/momentum; however, only works in full batch mode.
- **Nesterov accelerated gradient:**
 - Helps mitigate the risk of choosing a bad mini-batch.
- **Adagrad:**
 - Allows an automatically decaying per-weight learning rate and momentum concept.
- **Adadelta:**
 - Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- **Non-Gradient Methods:**
 - Non-gradient methods can sometimes be useful, though rarely outperform gradient-based backpropagation methods. These include: [simulated annealing](#), [genetic algorithms](#), [particle swarm optimization](#), [Nelder Mead](#), and [many more](#).

7 Two Problems

1. Vanishing Gradient problem

With gradient descent, sometimes the gradient becomes vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

2. Neuron Saturation

When a neuron's activation value becomes close to the maximum or minimum of the range (e.g. 0/1 for sigmoid, -1/1 for tanh), the neuron is said to be saturated. In those cases, the gradient becomes small (vanishing gradient), and the network stops learning.