CS5489 - Machine Learning

Lecture 8a - Neural Networks

Prof. Antoni B. Chan

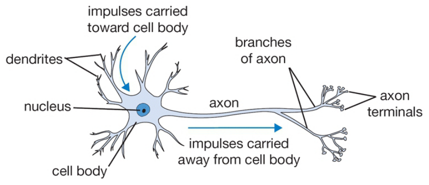
Dept. of Computer Science, City University of Hong Kong

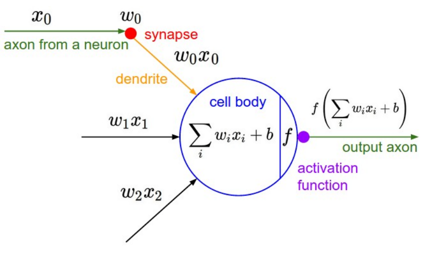
Outline

- History
- Perceptron
- Multi-class logistic regression
- Multi-layer perceptron (MLP)

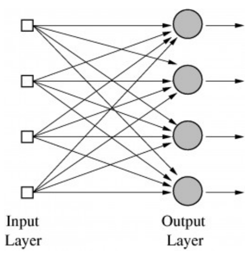
Original idea

- *Perceptron*
  - Warren McCulloch and Walter Pitts (1943), Rosenblatt (1957)
  - Simulate a neuron in the brain
    1. take binary inputs (input from nearby neurons)
    2. multiply by weights (synapses, dendrites)
    3. sum and threshold to get binary output (output axon)
  - Train weights from data.





- Multiple outputs handled by using multiple perceptrons

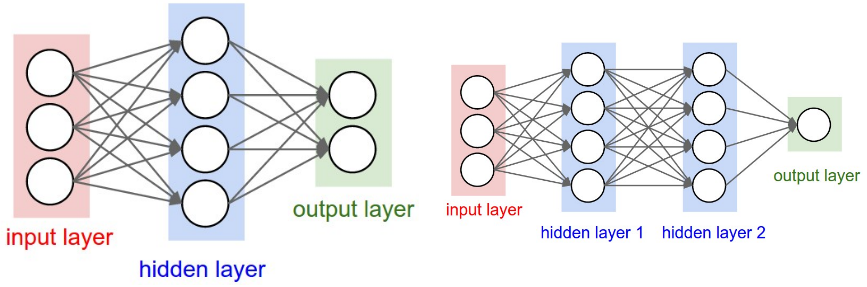


- **Problem:**
  - linear classifier, can't solve harder problems

Multi-layer Perceptron

- Add *hidden* layers between input and output neurons
  - each layer extracts some features from the previous layers

  - can represent complex non-linear functions
  - train weights using *backpropagation* algorithm. (1970-80s)
  - (now called a *neural network*)



- **Problem:**
  - difficult to train.
  - sensitive to initialization.
  - computationally expensive (at the time).

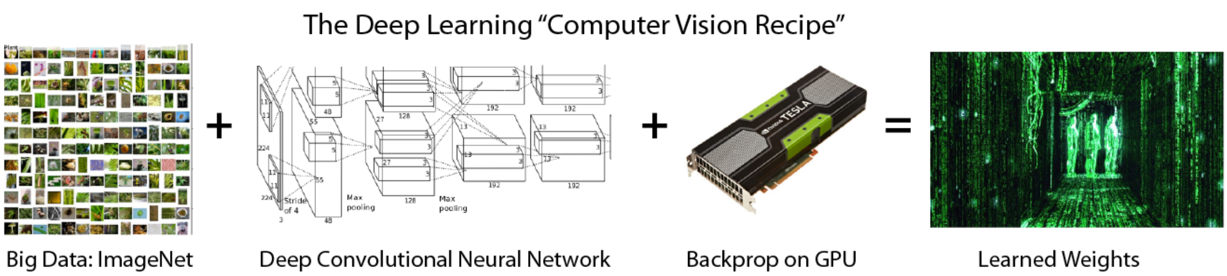
Decline in the 1990s

- Because of those problems, NN became less popular in the 1990s
  - Support vector machines (SVM) had good accuracy
    - easy to use - only one global optimum.
    - learning is not sensitive to initialization.
    - theory about generalization guarantees.
  - Not a lot of data, so kernel methods were still okay.

Deep learning

- There was a resurgence in NN in the 2000s, due to a number of factors:
  - improvements in network architecture
    - developed nodes that are easier to train
  - better training algorithms
    - better ways to prevent overfitting
    - better initialization methods
  - faster computers
    - massively parallel GPUs (Thanks to gamers!)
  - more labeled data
    - from Internet
    - crowd-sourcing for labeling data (Amazon Turk)

- We can train NN with more and more layers $\Rightarrow$ Deep Learning



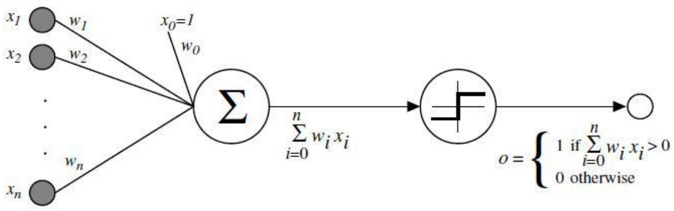
Outline

- History
- **Perceptron**
- Multi-class logistic regression
- Multi-layer perceptron (MLP)

Perceptron

- Model a single neuron
  - input $\mathbf{x} \in \mathbb{R}^d$ is a d-dim vector
  - apply a weight to the inputs
  - sum and threshold to get the output

- Formally,
  - $y = f(\sum_{j=0}^{d} w_j x_j) = f(\mathbf{w}^T \mathbf{x})$
  - $\mathbf{w}$ is the weight vector.
  - $f(a)$ is the activation function
    - $f(a) = \begin{cases} 1, & a > 0 \\ 0, & \text{otherwise} \end{cases}$



Perceptron training criteria

- Train the perceptron on data $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$
- Only look at the points that are misclassified.
  - Loss is based on how badly misclassified

$$E(\mathbf{w}) = \sum_{i=1}^{N} \begin{cases} -y_i \mathbf{w}^T \mathbf{x}_i, & \mathbf{x}_i \text{ is misclassified} \\ 0, & \text{otherwise} \end{cases}$$

- Minimize the loss: $\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} E(\mathbf{w})$

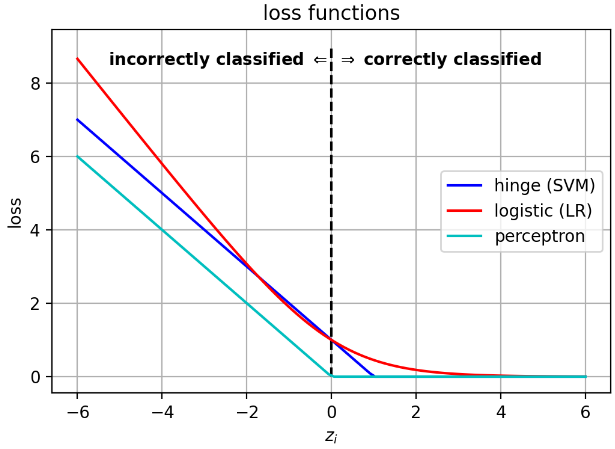
Perceptron Loss Function

- Define $z_i = y_i \mathbf{w}^T \mathbf{x}_i$,
- The loss function is

$$L(z_i) = \max(0, -z_i)$$

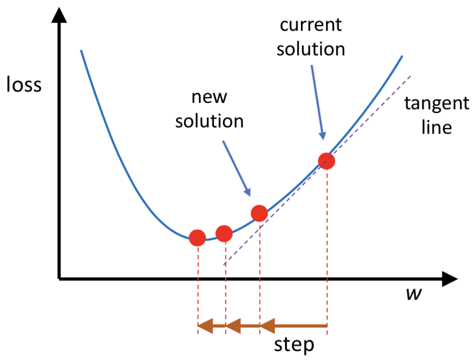
In [5]:
```
lossfig
```

Out[5]:



Training algorithm

- We can learn the model by applying gradient descent.
  - Move $\mathbf{w}$ in the direction to decrease the loss $E(\mathbf{w})$.
  - Gradient descent update: $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{d}{d\mathbf{w}} E(\mathbf{w})$
    - η is the learning rate for gradient descent



- Computers were slow back then...
- Solution: only look at one data point at a time and use gradient descent.
  - loss of a misclassified point $\mathbf{x}_i$: $E_i(\mathbf{w}) = -y_i \mathbf{w}^T \mathbf{x}_i$
  - Gradient: $\frac{d}{d\mathbf{w}} E_i(\mathbf{w}) = -y_i \mathbf{x}_i$

- **Perceptron Algorithm**
  - For each point $\mathbf{x}_i$,
    - If the point $\mathbf{x}_i$ is misclassified,
      - Update weights: $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
  - Repeat until no more points are misclassified

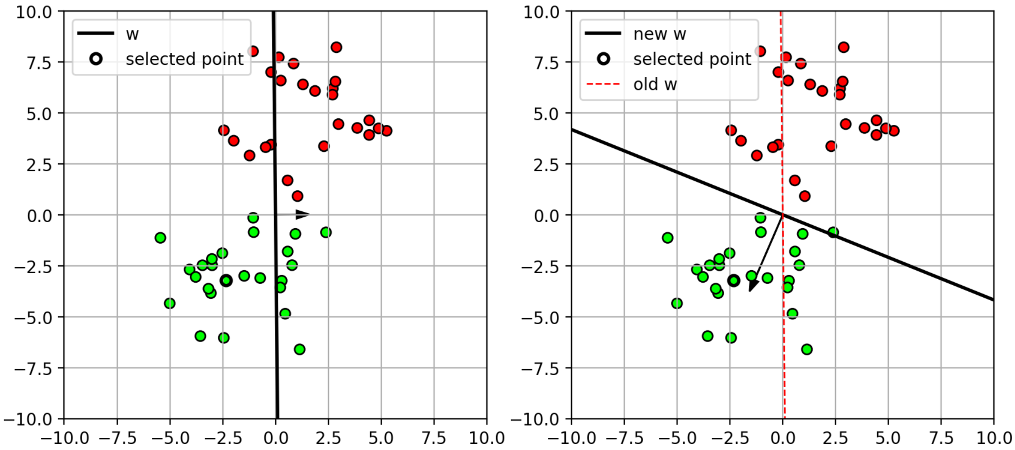
- **Notes:**
  - The effect of the update step is to rotate $\mathbf{w}$ towards the misclassified point $\mathbf{x}_i$.
  - This is called *Stochastic Gradient Descent*.
    - useful because we only need to look at a little bit of data at a time.
    - less computing/memory requirement in each iteration.

Example

- Iteration 1
  - $\mathbf{w}$ rotates towards the misclassified point (bold circle)

In [8]:
```
figs[0]
```

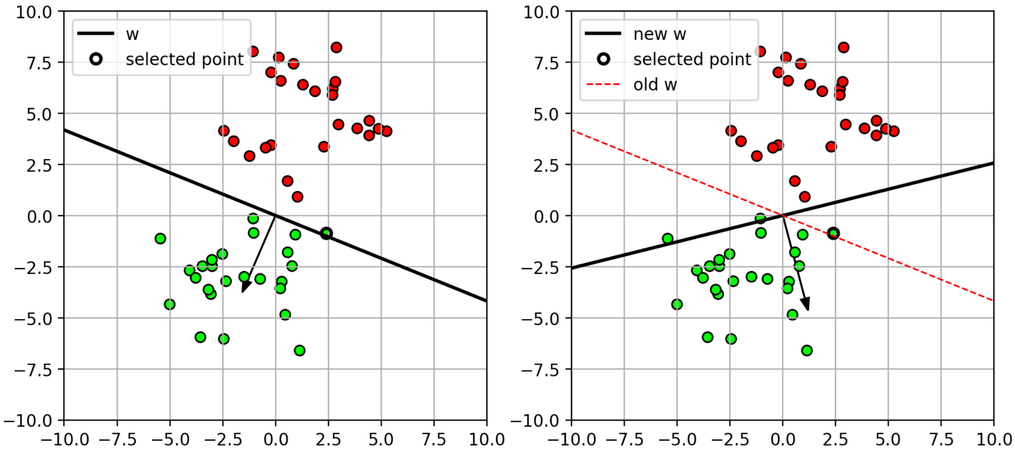
Out[8]:



- Iteration 2

In [9]:
```
figs[1]
```

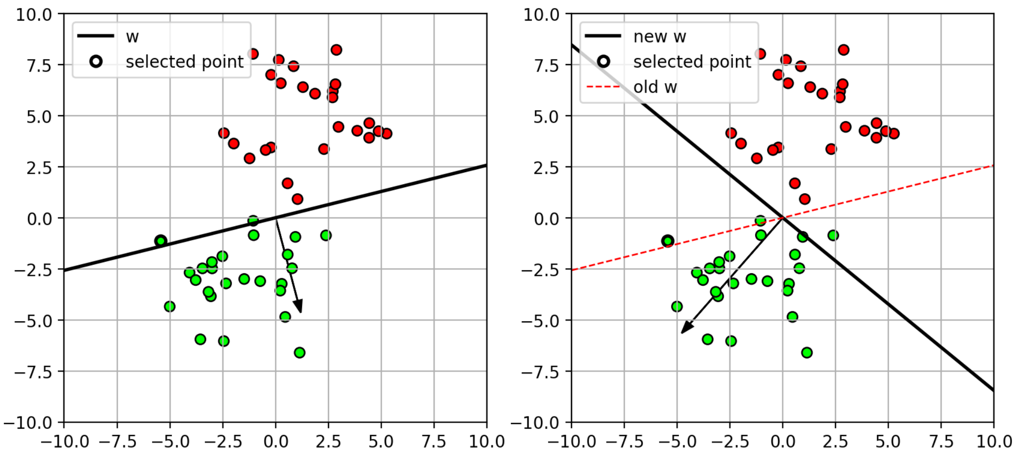
Out[9]:



- Iteration 3

In [10]:
```
figs[2]
```

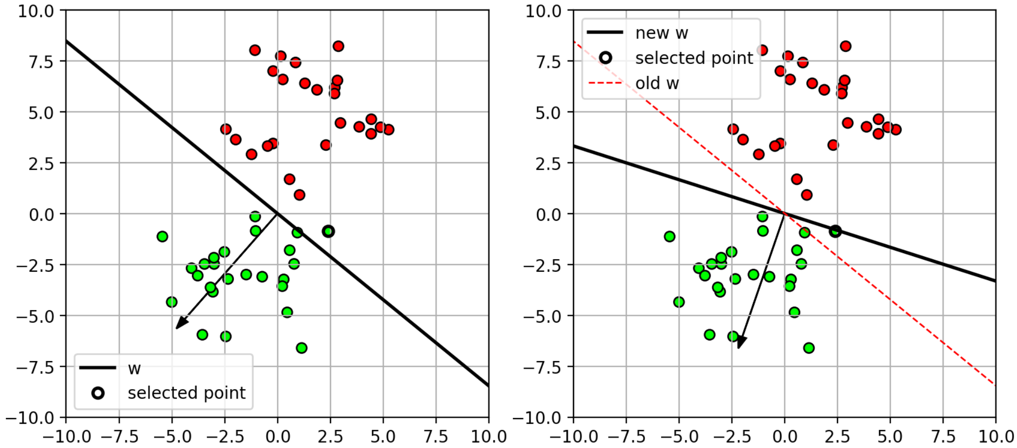
Out[10]:



- Iteration 4
  - No more errors

In [11]:
```
figs[3]
```

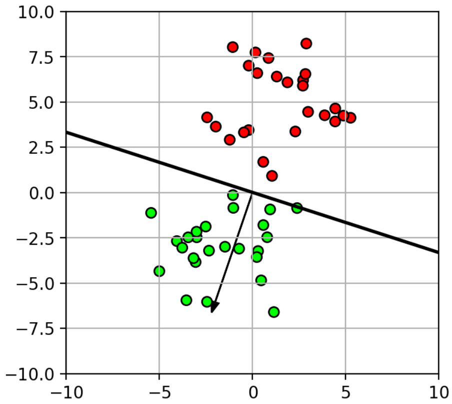
Out[11]:



- Final classifier

In [12]:

```
plt.figure(figsize=(4,4))
plot_perceptron((w,0),X,Y,axbox)
```



Perceptron Algorithm

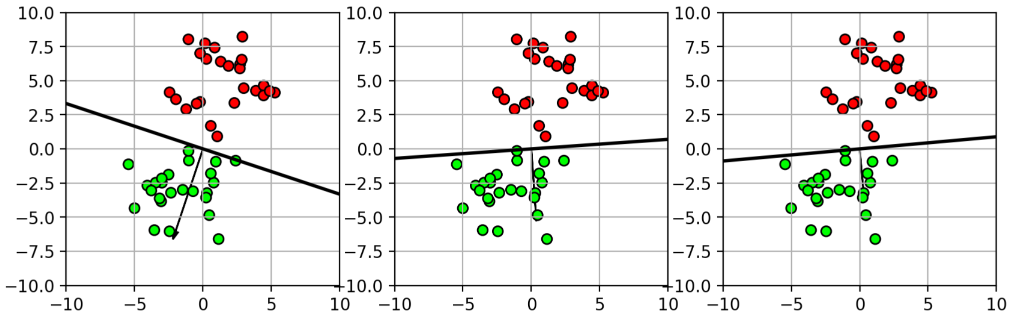
- Fails to converge if data is not linearly separable
- Rosenblatt proved that the algorithm will converge if the data is linearly separable.
  - the number of iterations is inversely proportional to the separation (margin) between classes.
  - *This was one of the first machine learning results!*

- Different initializations can yield different weights
  - There are multiple decision boundaries with 0 loss.

In [14]:

```
pfig
```

Out[14]:



Outline

- History

- Perceptron
- **Multi-class logistic regression**
- Multi-layer perceptron (MLP)

Revisiting Multiclass logistic regression

- Consider a multi-class classification problem with C classes
  - class labels $y \in \{1, \cdots, C\}$
  - equivalently, class vectors:

$$\mathbf{y} \in \{ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \} = \{\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_C\}$$

  - $\mathbf{e}_j$ is the canonical vector.

Linear functions

- Construct C linear functions, one for each class
  - $g_j(\mathbf{x}) = \mathbf{w}_j^T \mathbf{x}$, for $j = \{1, \cdots, C\}$
  - $\mathbf{w}_j$ is the weight vector for the j-th class.
  - (to reduce clutter, we implicitly include the bias term)
- Combine into a vector-valued function ($\mathbb{R}^C$):
  - $\mathbf{g}(\mathbf{x}) =

$$\begin{bmatrix} g_1(\mathbf{x}) \\ \vdots \\ g_C(\mathbf{x}) \end{bmatrix}$$

    = \mathbf{W}^T
  \mathbf{x}$,
  - Weight matrix: $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_C]$

Mapping to probabilities

- output of $\mathbf{g}(\mathbf{x})$ is a real vector in $\mathbb{R}^C$.
- How to map it to set of class probabilities?
  - require $0 \leq p(y = j|\mathbf{x}) \leq 1$
  - and $\sum_{j=1}^C p(y = j|\mathbf{x}) = 1$.

Softmax function

- Given a real vector $\mathbf{a} \in \mathbb{R}^C$
- Let $s_j(\mathbf{a}) = \frac{\exp(a_j)}{\sum_{k=1}^C \exp(a_k)}$
  - if $a_j \gg a_i$, then the exponent will cause $s_j(\mathbf{a}) \to 1$.
  - denominator ensures $\sum_{j=1}^C s_j(\mathbf{a}) = 1$.
- Let $\mathbf{s}(\mathbf{a}) = [s_1(\mathbf{a}) \cdots s_C(\mathbf{a})]^T$
  - the output vector is ~1 in the dimension of $\mathbf{a}$ with largest value, and 0 elsewhere.
  - called the **softmax** function ("soft" because the values can be between 0 and 1)

Mapping to probabilities

- Define the probability of the j-th class $p(y=j|\mathbf{x})$ as:
  - $p(y=j|\mathbf{x}) = f_j(\mathbf{x}) = s_j(\mathbf{g}(\mathbf{x})) = \frac{\exp(g_j(\mathbf{x}))}{\sum_{k=1}^{C}\exp(g_k(\mathbf{x}))}$
    - if $g_j(\mathbf{x}) \gg g_i(\mathbf{x})$, then the exponent will cause numerator to be very large, and thus $p(y=j|\mathbf{x}) \rightarrow 1$.
    - the class with largest response $g_j(\mathbf{x})$ will have highest probability.
    - denominator ensures probabilities sum to 1 over classes.
- Finally, define the posterior probability vector:

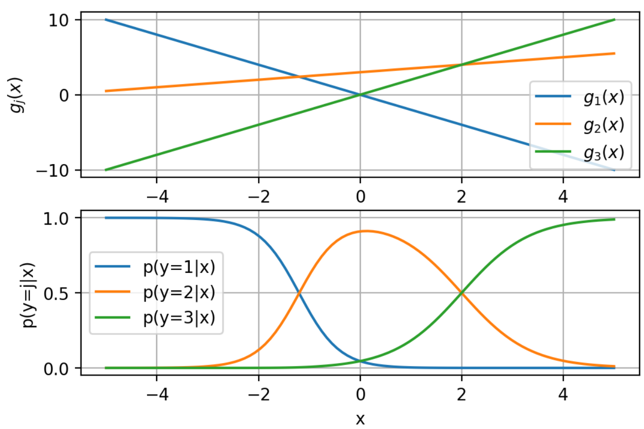
$$\begin{bmatrix} p(y=1|\mathbf{x}) \\ \vdots \\ p(y=C|\mathbf{x}) \end{bmatrix} = \mathbf{f}(\mathbf{x}) = \mathbf{s}(\mathbf{g}(\mathbf{x})))$$

Example

- linear functions and mapped probabilities

In [16]:
```
sfig
```

Out[16]:



Learning with MLE

- let $\mathbf{y}$ be the *class vector* representation of the class
  - i.e. $y_j = 1$ indicates class $y = j$, and 0 otherwise.
- **Log-likelihood function**
  - categorical distribution

$$\log p(\mathbf{y}|\mathbf{x}) = \log \prod_{j=1}^{C} f_j(\mathbf{x})^{y_j}$$
$$= \sum_{j=1}^{C} y_j \log f_j(\mathbf{x}) = \mathbf{y}^T \log \mathbf{f}(\mathbf{x})$$

```
- Note: the $\log$ of a vector is element-wise log.
```

- Maximum Likelihood Estimation (MLE)
  - Let $\mathcal{D} = \{(\mathbf{y}_i, \mathbf{x}_i)\}$ be the training set.
  - MLE goal:

$$
\begin{aligned}
\mathbf{W}^* &= \underset{\mathbf{W}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(\mathbf{y}_i|\mathbf{x}_i) \\
&= \underset{\mathbf{W}}{\operatorname{argmax}} \sum_{i=1}^{N} \mathbf{y}_i^T \log \mathbf{f}(\mathbf{x}_i)
\end{aligned}
$$

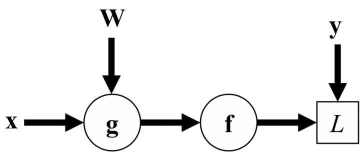
- Equivalently, turn maximization problem into minimization

$$
\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{N} \left\{ -\sum_{j=1}^{C} y_{ij} \log f_j(\mathbf{x}_i) \right\} = \sum_{i=1}^{N} L(\mathbf{y}_i, \mathbf{f})
$$

- Called the **cross-entropy loss** between ground-truth $\mathbf{y}_i$ and prediction $\mathbf{f}(\mathbf{x}_i)$
  - $L(\mathbf{y}, \mathbf{f}) = -\sum_{j=1}^{C} y_j \log f_j(\mathbf{x})$

How to optimize?

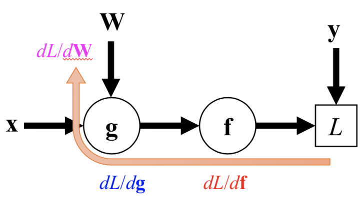
- Use gradient descent:
  - $\mathbf{W}^{(t)} = \mathbf{W}^{(t-1)} - \eta \frac{dL}{d\mathbf{W}}\Big|_{\mathbf{W}^{(t-1)}}$
    - gradient evaluted at current parameters $\mathbf{W}^{(t-1)}$.
- How do we compute the gradient?
  - We have a composition of functions:
    - $\mathbf{g}(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$
    - $\mathbf{f}(\mathbf{x}) = \mathbf{s}(\mathbf{g}(\mathbf{x}))$
    - $L(\mathbf{y}, \mathbf{f}) = -\mathbf{y}^T \log \mathbf{f}(\mathbf{x})$



- Use the chain rule!
  - in one-dimension:
    - suppose $f(x) = s(g(x))$
    - by the chain rule: $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$
  - our case is more complicated because of vector-valued functions.

Applying the Chain rule

- Work backwards to compute gradient



- Gradient of loss wrt $\mathbf{f}$:

$$
\frac{dL}{d\mathbf{f}} = \begin{bmatrix} \frac{dL}{df_1} \\ \vdots \\ \frac{dL}{df_C} \end{bmatrix}
$$

- Gradient of loss wrt $\mathbf{g}$:
  - First, look at individual g_j
    - changes in g_j affect all f_k, so sum over derivatives of f_k.

$$\frac{dL}{dg_j} = \sum_{k=1}^{C} \frac{dL}{df_k}\frac{df_k}{dg_j} = \frac{d\mathbf{f}^T}{dg_j}\frac{dL}{d\mathbf{f}}$$

`- where $\frac{d\mathbf{f}^T}{dg_j} = [\frac{df_1}{dg_j} \cdots \frac{df_C}{dg_j}]$.`

- Gradient of loss wrt $\mathbf{g}$:
  - For all linear functions $\mathbf{g}$

$$\frac{dL}{d\mathbf{g}} = \begin{bmatrix} \frac{dL}{dg_1} \\ \vdots \\ \frac{dL}{dg_C} \end{bmatrix} = \begin{bmatrix} \frac{d\mathbf{f}^T}{dg_1}\frac{dL}{d\mathbf{f}} \\ \vdots \\ \frac{d\mathbf{f}^T}{dg_C}\frac{dL}{d\mathbf{f}} \end{bmatrix} = \frac{d\mathbf{f}^T}{d\mathbf{g}}\frac{dL}{d\mathbf{f}}$$

`- where the Jacobian (transpose) is:`

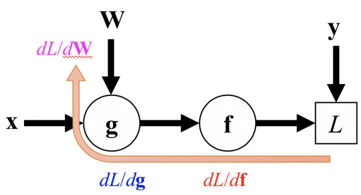
$$\frac{d\mathbf{f}^T}{d\mathbf{g}} = \begin{bmatrix} \frac{df_1}{dg_1} & \cdots & \frac{df_C}{dg_1} \\ \vdots & \ddots & \vdots \\ \frac{df_1}{dg_C} & \cdots & \frac{df_C}{dg_C} \end{bmatrix}$$

- (how each input dimension of affects each output dimension of $\mathbf{f}(\mathbf{g})$)

- Gradient of loss wrt $\mathbf{w}_j$:
  - $\frac{dL}{d\mathbf{w}_j} = \sum_{k=1}^{C} \frac{dg_k}{d\mathbf{w}_j}\frac{dL}{dg_k} = \frac{d\mathbf{g}^T}{d\mathbf{w}_j}\frac{dL}{d\mathbf{g}}$
  - since weight vector $\mathbf{w}_j$ only appears in g_j
    - $\frac{dL}{d\mathbf{w}_j} = \frac{dg_j}{d\mathbf{w}_j}\frac{dL}{dg_j}$

Summary:

- Chain rule:
  - compute the gradient by working backwards from $\mathbf{f}$ to $\mathbf{w}_j$
    1. Gradient of loss wrt $\mathbf{f}$: $\frac{dL}{d\mathbf{f}}$
    2. Gradient of loss wrt $\mathbf{g}$: $\frac{dL}{d\mathbf{g}} = \frac{d\mathbf{f}^T}{d\mathbf{g}}\frac{dL}{d\mathbf{f}}$
    3. Gradient of loss wrt $\mathbf{w}_j$: $\frac{dL}{d\mathbf{w}_j} = \frac{d\mathbf{g}^T}{d\mathbf{w}_j}\frac{dL}{d\mathbf{g}}$



- Final result after some derivation of gradients:
  - $\frac{dL}{d\mathbf{w}_j} = \mathbf{x}(f_j(\mathbf{x}) - y_j)$

Example

In [18]:
```
# learn logistic regression classifier
mlogreg = linear_model.LogisticRegression(C=10,
```

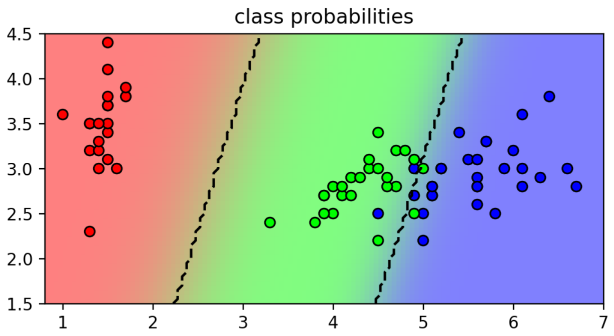
```
                multi_class='multinomial', solver='lbfgs')
                # use multi-class and corresponding solver
mlogreg.fit(trainX, trainY)

# now contains 3 hyperplanes and 3 bias terms (one for each class)
print("w=", mlogreg.coef_)
print("b=", mlogreg.intercept_)

# predict from the model
predY = mlogreg.predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
print("test accuracy=", acc)
```

```
w= [[-4.13092437  1.30718735]
 [-0.71717021  0.23609022]
 [ 4.84809458 -1.54327757]]
b= [ 11.46078594   5.40723484 -16.86802078]
test accuracy= 0.9733333333333334
```

- class probabilites from softmax function.

In [21]:
```
lr3classm
```

Out[21]:



Summary

- Two related linear classification models: Perceptron, Multi-class logistic regression
  - compute a linear function of input
  - non-linear output function (threshold or soft-max)
- We have assumed the inputs are feature vectors already.
  - What if we want to extract the features vectors too?