CS4487 - Machine Learning

Lecture 9a - Neural Networks, Deep Learning

Dr. Antoni B. Chan

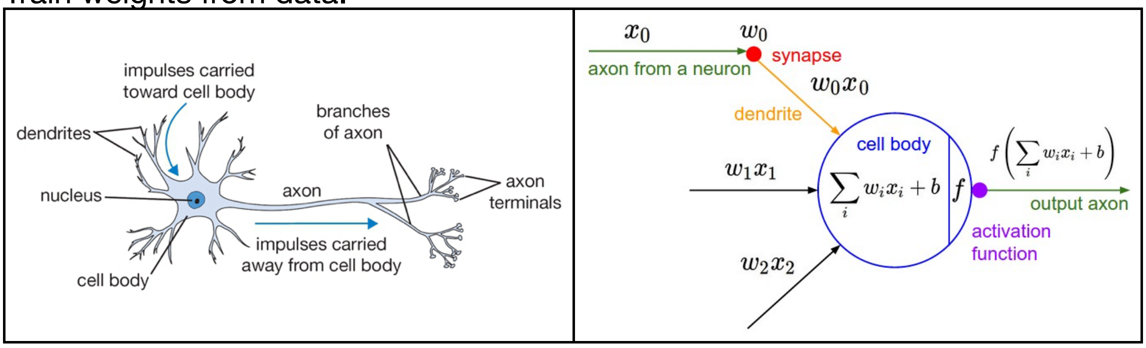
Dept. of Computer Science, City University of Hong Kong

Outline

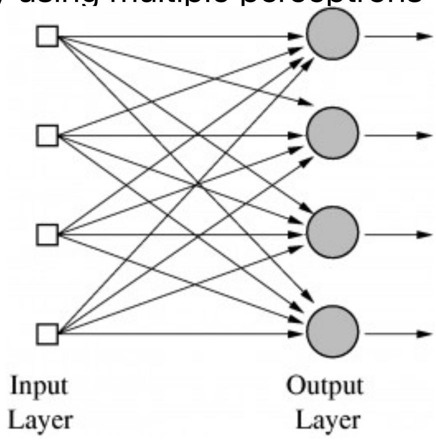
- History
- Perceptron
- Multi-layer perceptron (MLP)
- Convolutional neural network (CNN)
- Autoencoder (AE)

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.



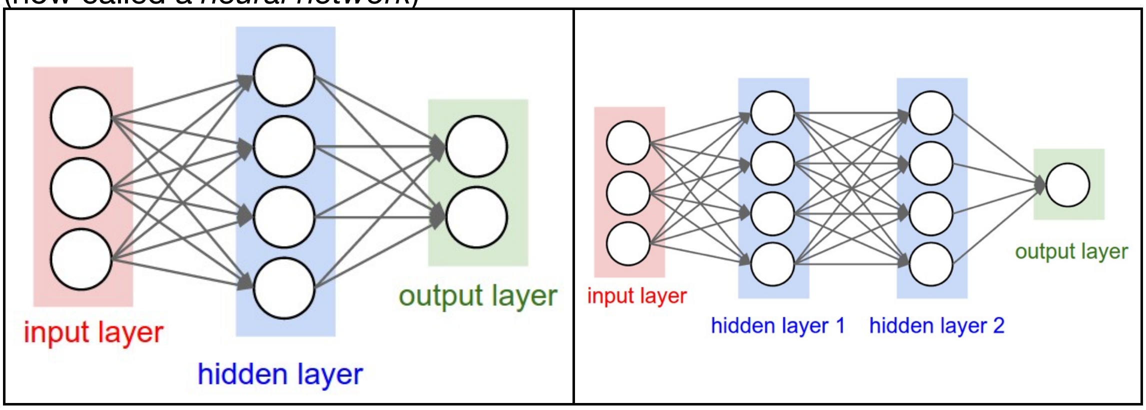
- Multiple outputs handled by using multiple perceptrons



- **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

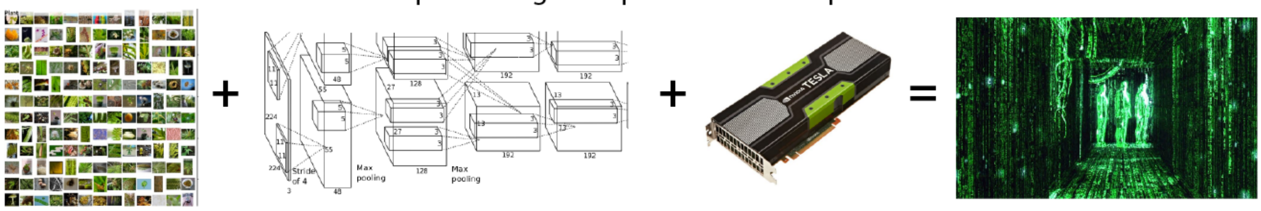
- Because of those problems, NN became less popular in the 1990s
  - Support vector machines (SVM) had good accuracy
    - easy to use - only one global optimum.
    - learning is not sensitive to initialization.
    - theory about performance guarantees.
  - Not a lot of data, so kernel methods were 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
  - more labeled data
    - from Internet
    - crowd-sourcing for labeling data (Amazon Turk)

- We can train NN with more and more layers --> Deep Learning

The Deep Learning "Computer Vision Recipe"



Big Data: ImageNet — Deep Convolutional Neural Network — Backprop on GPU — Learned Weights

---

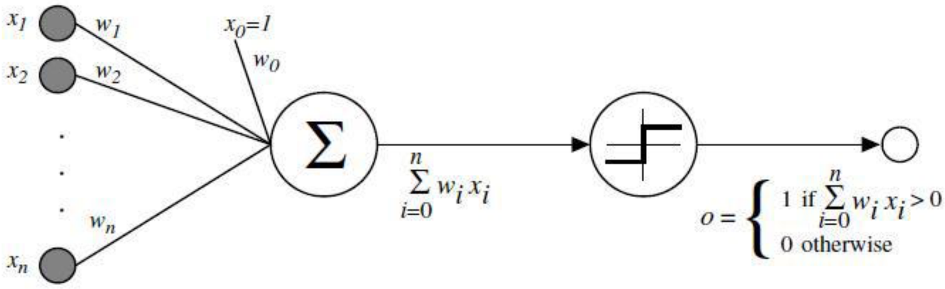
Outline

- History
- **Perceptron**
- Multi-layer perceptron (MLP)
- Convolutional neural network (CNN)
- Autoencoder (AE)

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

Training algorithm

- Computers were slow back then...only look at one data point at a time and use gradient descent.
- **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

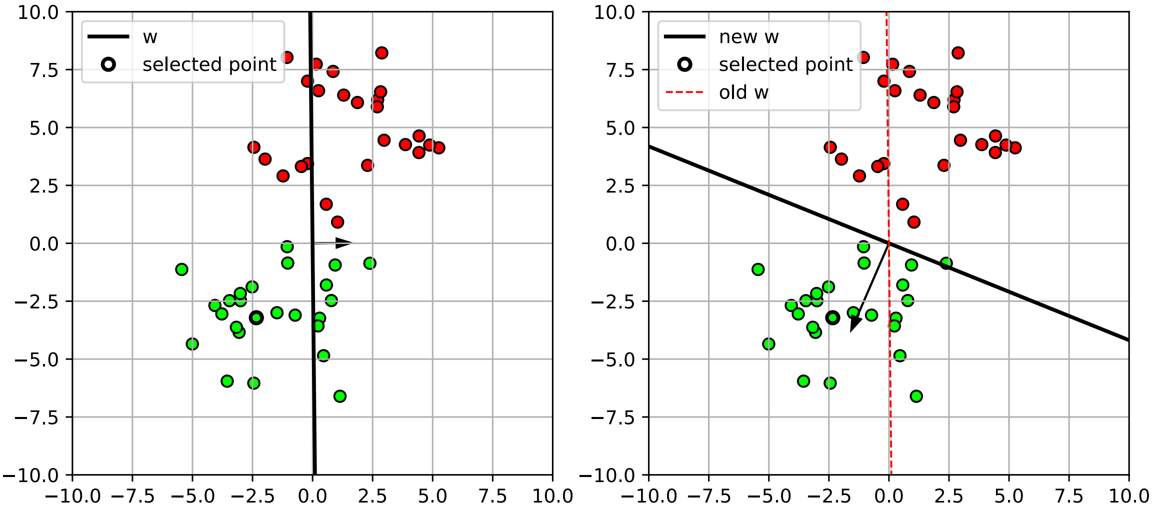
- **Notes:**
  - η is the learning rate for gradient descent
  - The effect of the update step is to rotate $\mathbf{w}$ towards the misclassified point $\mathbf{x}_i$.
  - This is called *Stochastic Gradient Descent*.
    - It is useful because we only need to look at a little bit of data at a time.

Example

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

In [6]:
```
figs[0]
```

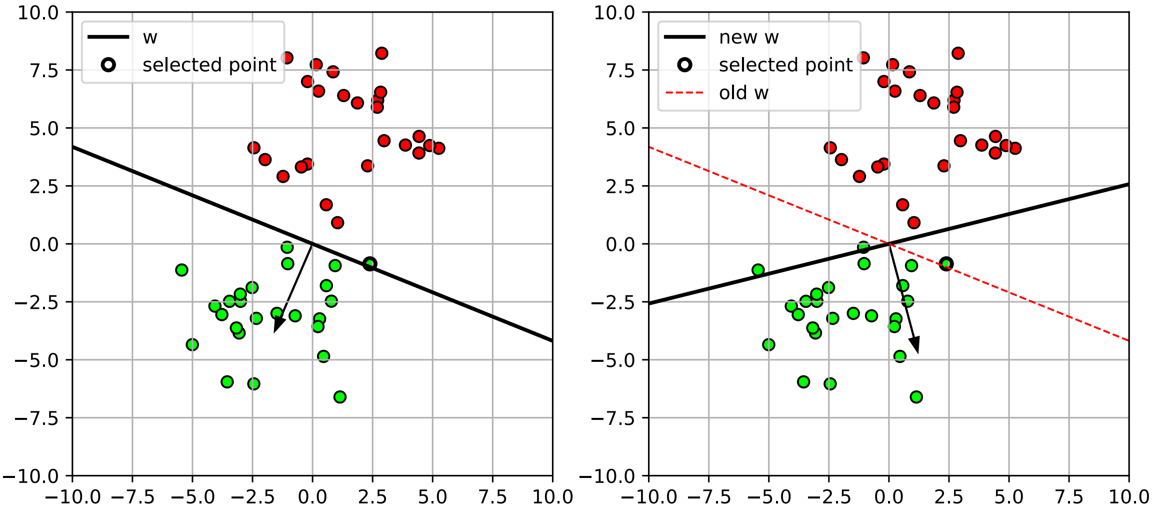
Out[6]:



- Iteration 2

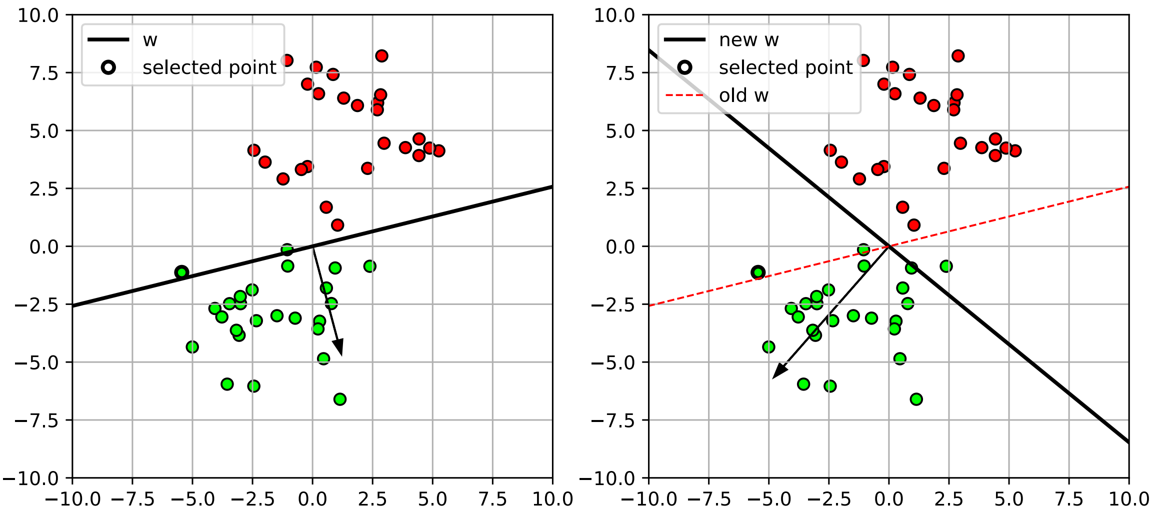
In [7]:
```
figs[1]
```

Out[7]:



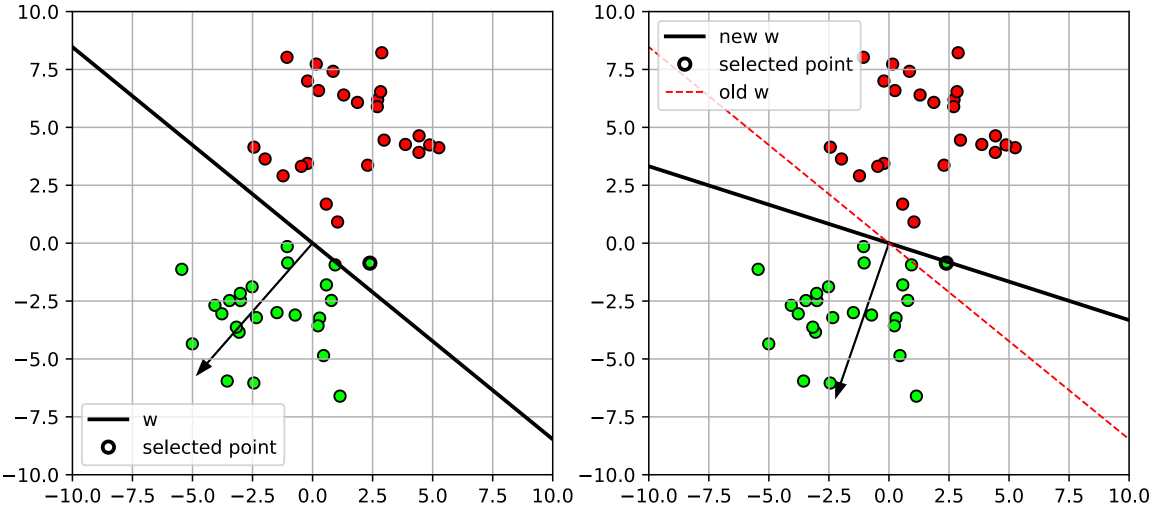
- Iteration 3

In [8]:
```
figs[2]
```
Out[8]:



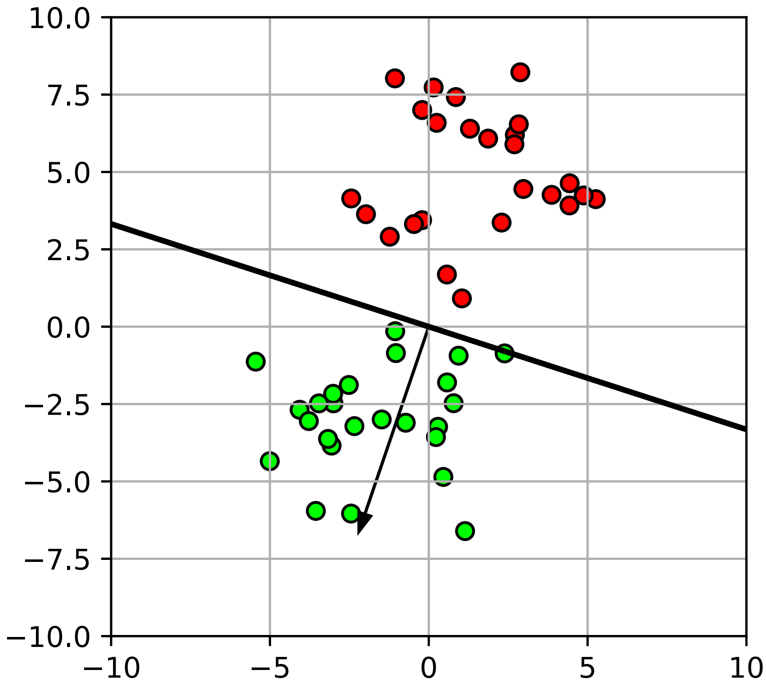
- Iteration 4
  - No more errors

In [9]:
```
figs[3]
```
Out[9]:



- Final classifier

In [10]:
```
plt.figure(figsize=(4,4))
plot_perceptron((w,0),X,Y,axbox)
```

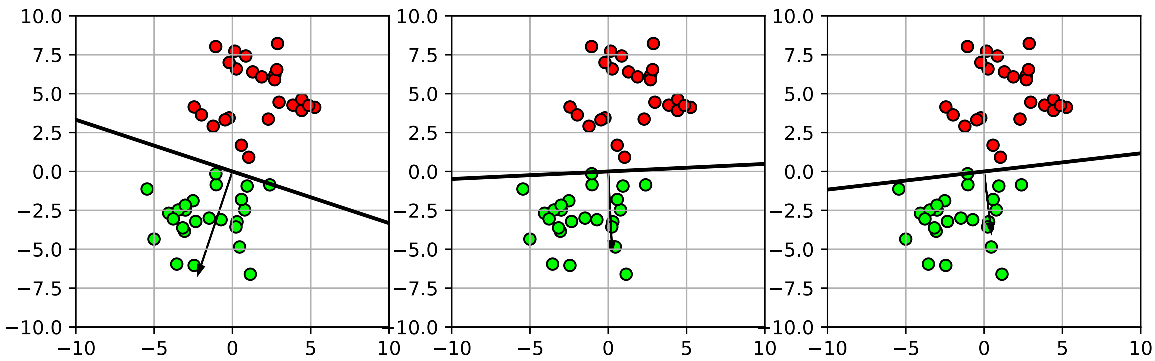


Perceptron Algorithm

- 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.

In [12]:
```
pfig
```

Out[12]:

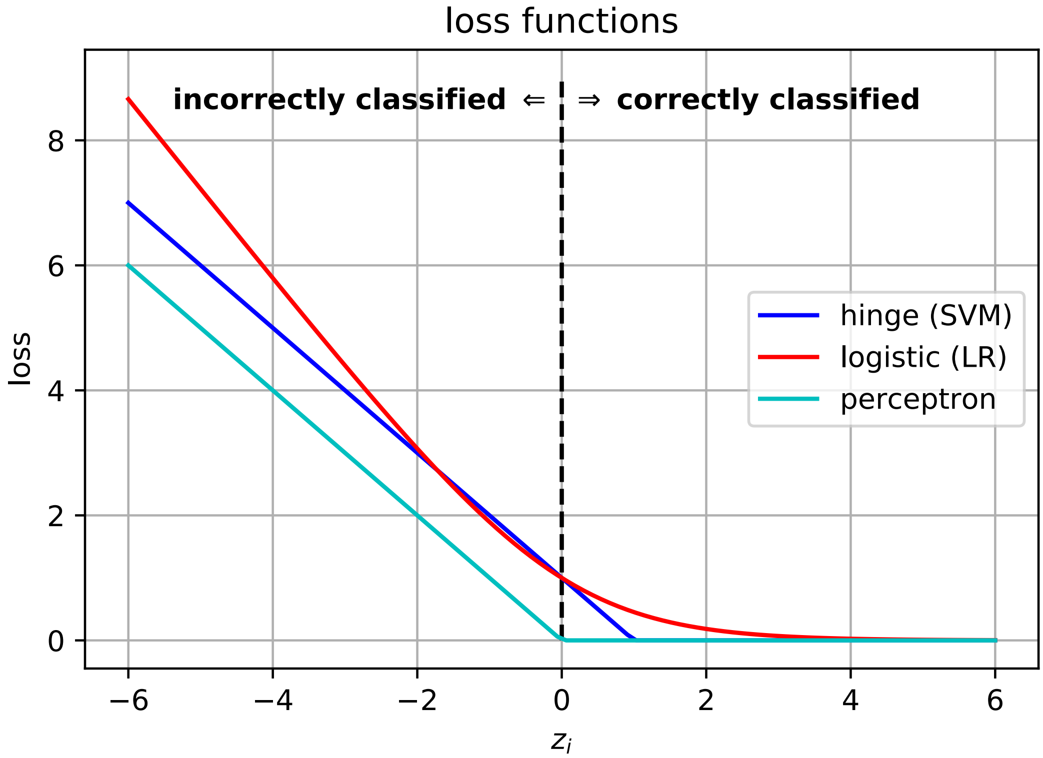


Perceptron Loss Function

- Define $z_i = y_i \mathbf{w}^T \mathbf{x}_i$,
- The loss function is $L(z_i) = \max(0, -z_i)$.

In [14]:
```
lossfig
```

Out[14]:

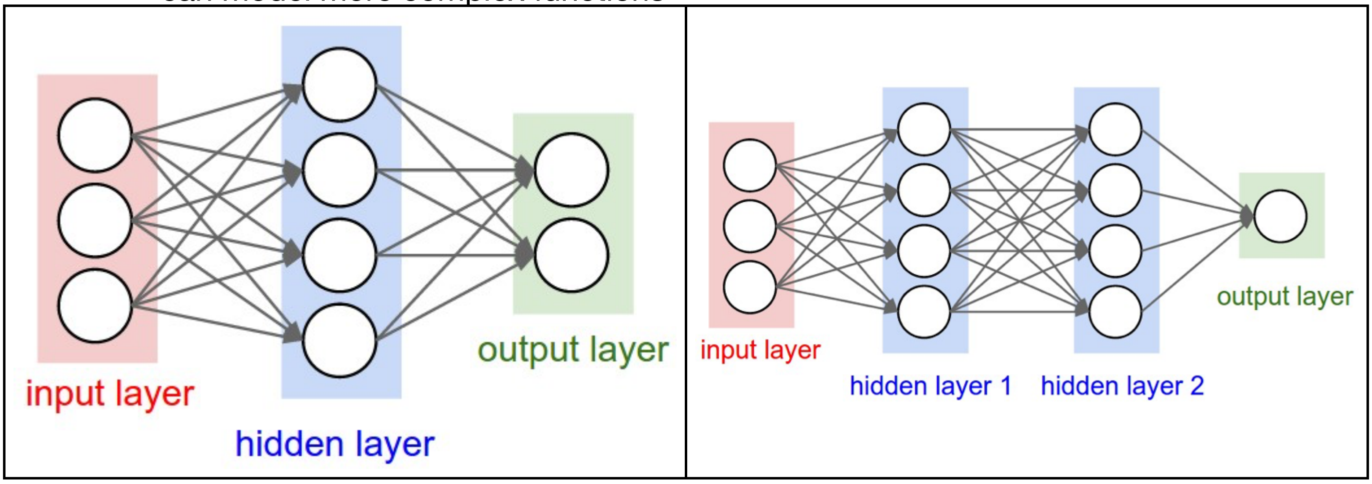


Outline

- History
- Perceptron
- **Multi-layer perceptron (MLP)**
- Convolutional neural network (CNN)
- Autoencoder (AE)

Multi-layer Perceptron

- Add hidden layers between the inputs and outputs
  - each hidden node is a Perceptron (with its own set of weights)
    - its inputs are the outputs from previous layer
    - extracts a feature pattern from the previous layer
  - can model more complex functions



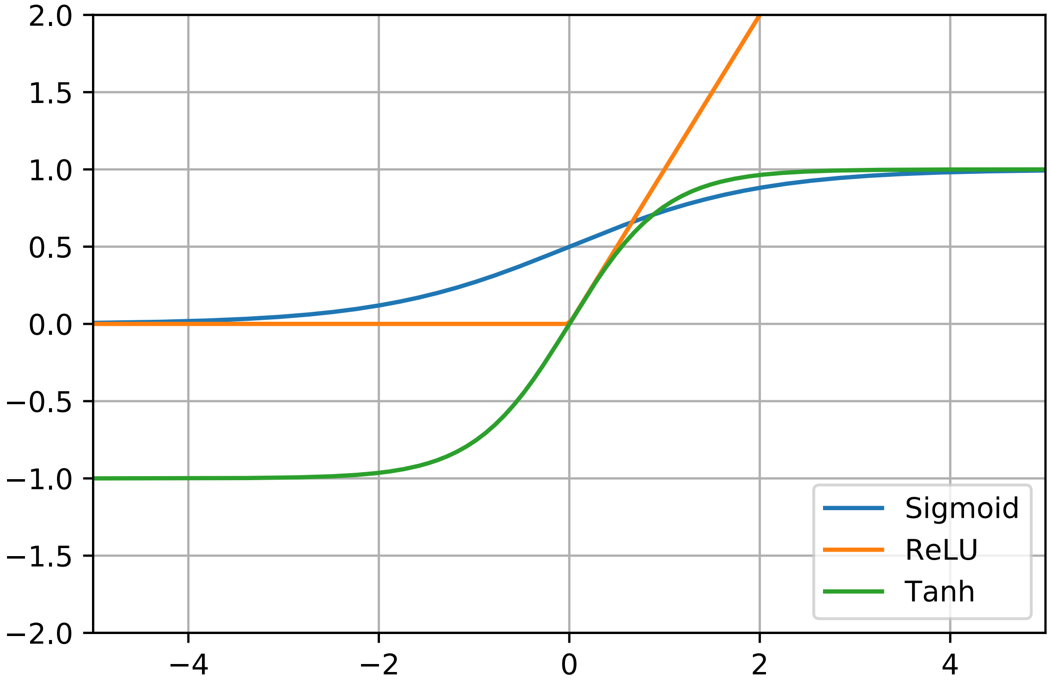
- Formally, for one layer:
  - $\mathbf{h} = f(\mathbf{W}^T\mathbf{x})$
    - Weight matrix $\mathbf{W}$ - one column for each node
    - Input $\mathbf{x}$ - from previous layer
    - Output $\mathbf{h}$ - to next layer
    - $f(a)$ is the activation function - applied to each dimension to get output

Activation functions

- There are different types of activation functions:
  - *Sigmoid* - output [0,1]
  - *Tanh* - output [-1,1]
  - *Rectifier Linear Unit (ReLU)* - output [0,∞]

In [16]:
```
actfig
```

Out[16]:



- Activation functions specifically for output nodes:
  - *Linear* - output for regression
  - *Softmax* - output for classification (same as multi-class logistic regression)
- Each layer can use a different activation function.

Which activation function is best?

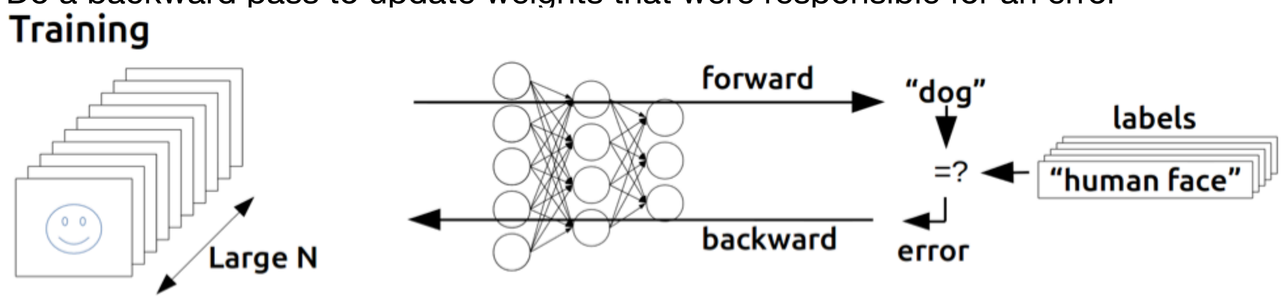
- In the early days, only the Sigmoid and Tanh activation functions were used.
  - these were notoriously hard to train with.
    - "vanishing gradient" problem
- Recently, ReLU has become very popular.
  - easier to train with - no "vanishing gradient"
  - faster - don't need to calculate exponential
  - sparse representation - most nodes will output zero.

Training an MLP

- For classification, we use the cross-entropy loss
  - $E = -\sum_{j=1}^{K} y_j \log \hat{y_j}$
    - y_j is 1 for the true class, and 0 otherwise
    - $\hat{y_j}$ is the softmax output for the j-th class
- Use gradient descent as before:
  - $w_{ij} \leftarrow w_{ij} - \eta \frac{dE}{dw_{ij}}$
    - layer i, node j
  - η is the learning rate
    - controls convergence rate
      - too small --> converges very slowly
      - too large --> possibly doesn't converge

Backpropagation (backward propagation)

- Do a forward pass to calculate the prediction
- Do a backward pass to update weights that were responsible for an error



Gradient descent with the chain-rule

- Suppose we have a 2-layer network
  - E is the cost function
  - g_1, g_2 are the output functions of the two layers
    - $g_j(\mathbf{x}) = f(\mathbf{W}_j^T \mathbf{x})$
    - $\mathbf{W}_1, \mathbf{W}_2$ are the weight matrices
- Prediction for input $\mathbf{x}$: $y = g_2(g_1(\mathbf{x}))$
- Cost for input $\mathbf{x}$: $E(\mathbf{x}) = E(g_2(g_1(\mathbf{x})))$

- Apply the chain rule to get the gradients of weights in layer
  - $\frac{dE(\mathbf{x})}{d\mathbf{W}_2} = \frac{dE}{dg_2}\frac{dg_2}{d\mathbf{W}_2}$
  - $\frac{dE(\mathbf{x})}{d\mathbf{W}_1} = \frac{dE}{dg_2}\frac{dg_2}{dg_1}\frac{dg_1}{d\mathbf{W}_1}$
- Defines a set of recursive relationships
  - 1) calculate the output of each node from first to last layer
  - 2) calculate the gradient of each node from last to first layer
- NOTE: the gradients multiply in each layer!
  - if two gradients are small (<1), their product will be even smaller. This is the "vanishing gradient" problem.

Stochastic Gradient Descent (SGD)

- The datasets needed to train NN are typically very large
- Use SGD so that only a small portion of the dataset is needed at a time
  - Each small portion is called a *mini-batch*
  - Use a *momentum* term, which averages the current gradient with those from previous mini-batches.
  - One complete pass through the data is called an *epoch*.

Other Tricks

- Normalize the inputs to [-1,1] or [0,1]
  - improves numerical stability.
- Separate the training set into training and validation
  - use the training set to run backpropagation
  - test the NN on the validation set for diagnostics
    - check for convergence - adjust learning rate if necessary
    - check for diverging loss - adjust learning rate
    - stopping criteria - stop when no change in the validation error.
    - decay learning rate after each epoch.

Load NN software

- We will use *keras*
  - compatible with scikit-learn
  - keras is an easy-to-use front-end for other (more complicated) NN backends.
    - using Tensorflow backend (could also use Theano)

In [18]:
```
# use TensorFlow backend
%env KERAS_BACKEND=tensorflow
import keras
import tensorflow
from keras.models import Sequential
from keras.layers import Dense, Activation
```

```
env: KERAS_BACKEND=tensorflow

Using TensorFlow backend.
/anaconda3/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: numpy.
dtype size changed, may indicate binary incompatibility. Expected 96, got 88
  return f(*args, **kwds)
```

In [19]:
```
keras.__version__
```

Out[19]:
```
'2.2.2'
```

In [20]:
```
tensorflow.__version__
```

Out[20]:
```
'1.9.0'
```

- train 1 NN with just one output layer
  - this is the same as logistic regression

In [39]:
```
# initialize random seed
random.seed(4487)
tensorflow.set_random_seed(4487)  # remove if using Theano

# convert class labels to binary indicators (required for Keras)
Yb = keras.utils.np_utils.to_categorical(Y)

# build the network
nn = Sequential()
nn.add(Dense(units=2,              # number of output nodes (classes)
             input_dim=2,          # input dimension
             activation='softmax' # softmax is for classification
            )
      )
```

In [40]:

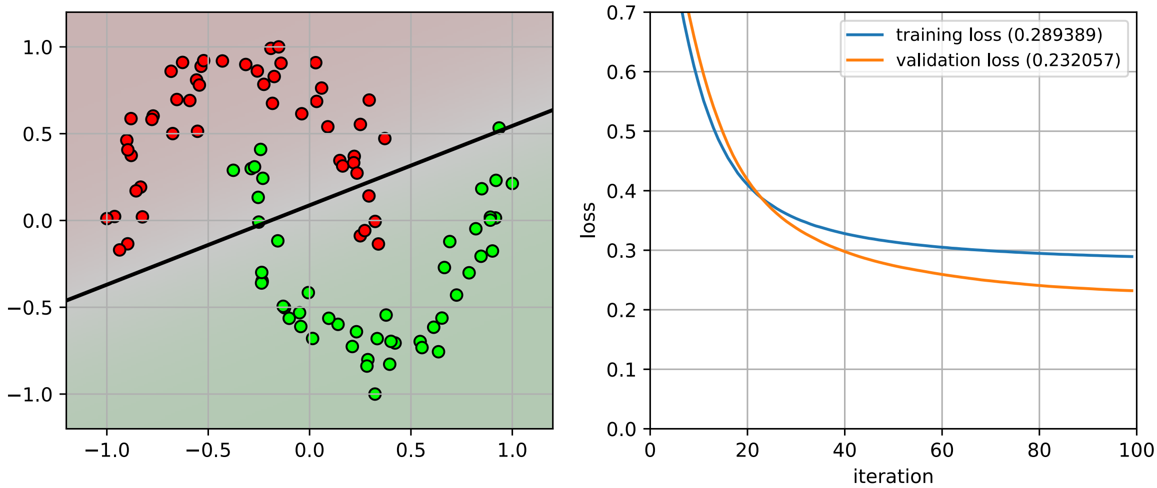
```
# compile the network
nn.compile(loss=keras.losses.categorical_crossentropy,   # classification loss
     optimizer=keras.optimizers.SGD(  # use SGD for optimization
                   lr=0.01,        # learning rate
                   momentum=0.9,   # momentum for averaging over batches
                   nesterov=True   # use Nestorov momentum
             ))

# fit the network
history = nn.fit(X, Yb,                     # the input/output data
                  epochs=100,               # number of iterations
                  batch_size=32,            # batch size
                  validation_split=0.1,  # ratio of data for validation
                  verbose=False             # set to True to see each iteration
                 )
```

- training and validation loss have converged

In [42]:

```
nnfig
```

Out[42]:

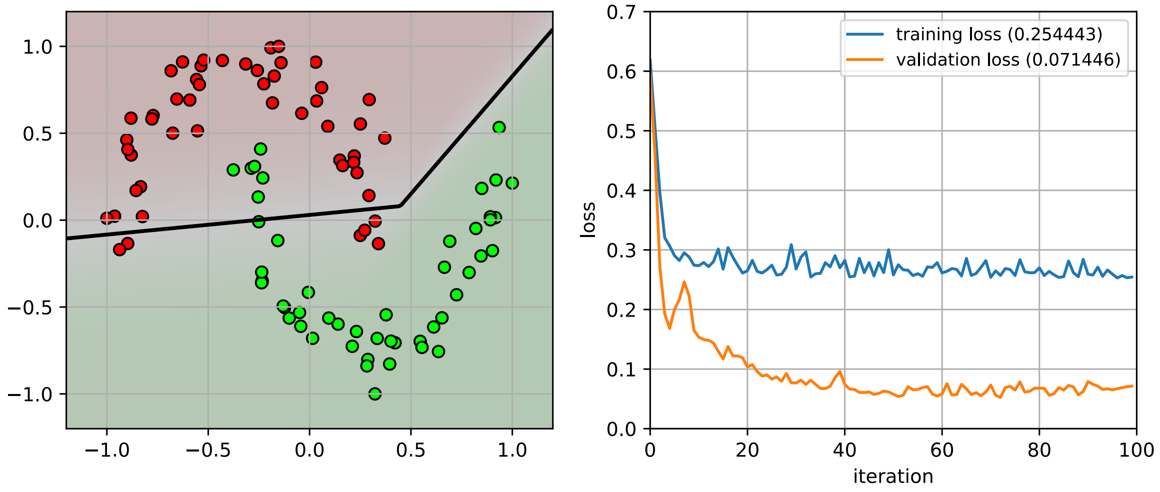


- Add one 1 hidden layer with 2 ReLU nodes
  - can carve out part of the red class.

In [43]:
```
# initialize random seed
random.seed(4487); tensorflow.set_random_seed(4487)

# build the network
nn = Sequential()
nn.add(Dense(units=2,                # 2 nodes in the hidden layer
             input_dim=2,
             activation='relu'))
nn.add(Dense(units=2,                # 2 output nodes (one for each class)
             activation='softmax'))

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(lr=0.3, momentum=0.9, nesterov=True))
history = nn.fit(X, Yb, epochs=100, batch_size=32, validation_split=0.1, verbose
=False)
```

In [45]:
```
nnfig
```

Out[45]:



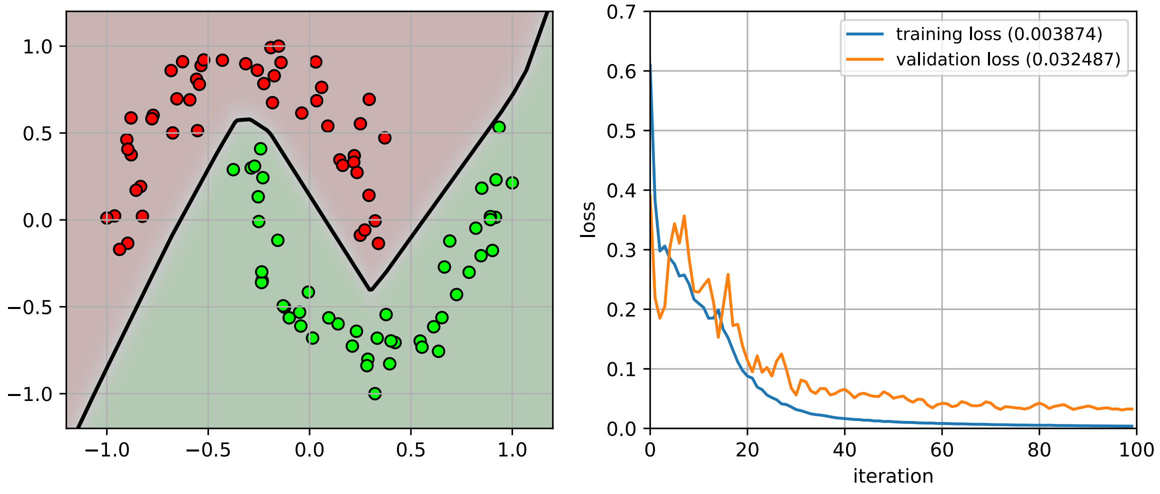
- Let's try more nodes
  - 1 hidden layer with 20 hidden nodes
  - with enough nodes, we can get a perfect classifier.

In [46]:
```
# initialize random seed
random.seed(4487); tensorflow.set_random_seed(4487)

# build the network
nn = Sequential()
nn.add(Dense(units=20, input_dim=2, activation='relu'))
nn.add(Dense(units=2, activation='softmax'))

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
    optimizer=keras.optimizers.SGD(lr=0.3, momentum=0.9, nesterov=True))
history = nn.fit(X, Yb, epochs=100, batch_size=32, validation_split=0.1, verbose
=False)
```

In [48]:

```
nnfig
```

Out[48]:



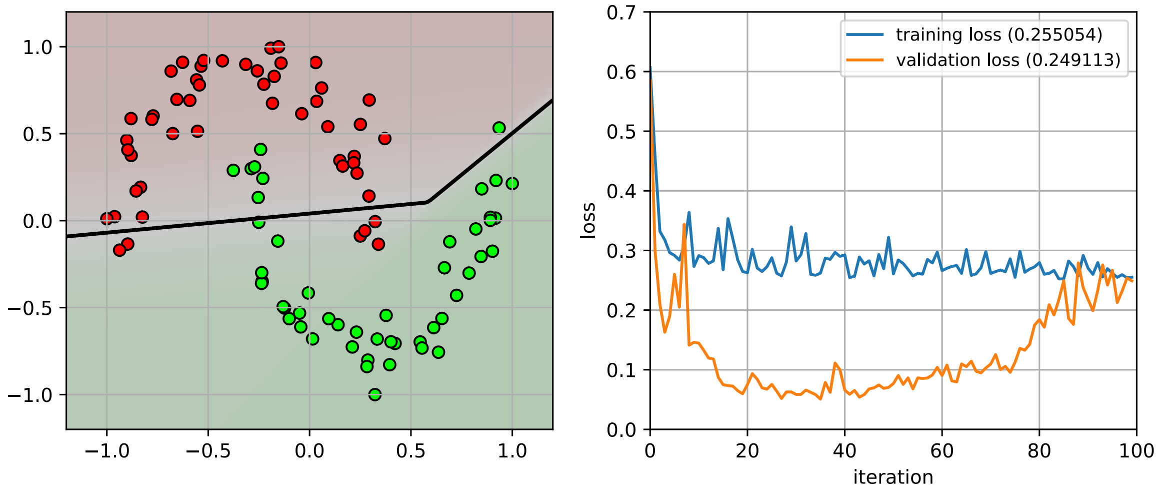
Overfitting

- Continuous training will sometimes lead to overfitting
  - the training loss decreases, but the validation loss increases

In [51]:

```
nnfig
```

Out[51]:



Early stopping

- Training can stopped when the validation loss is stable for a number of iterations
  - stable means change below a threshold
  - this is to prevent overfitting the training data.
  - we can limit the number of iterations.

In [52]:

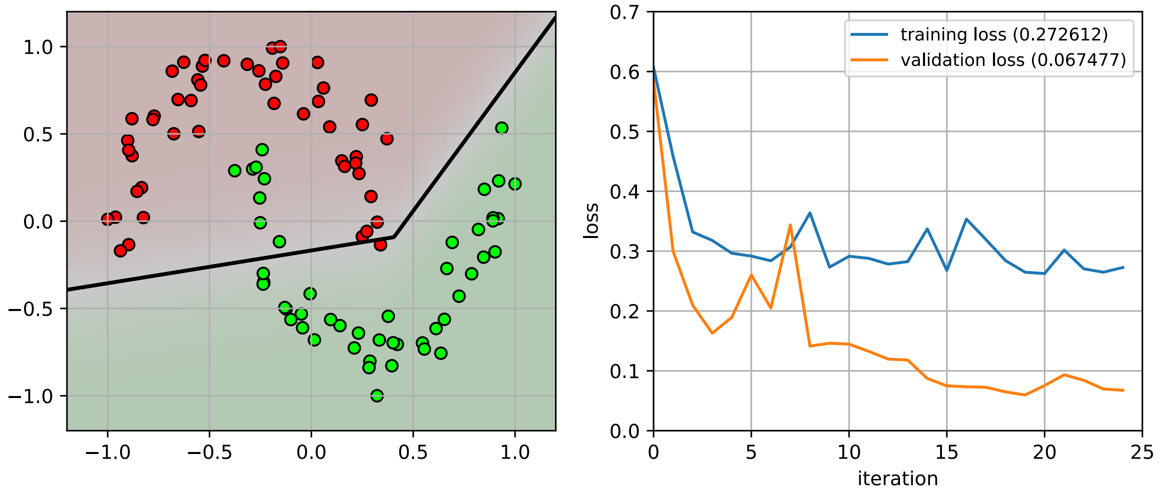
```
# initialize random seed
random.seed(4487); tensorflow.set_random_seed(4487)

# build the network
nn = Sequential()
nn.add(Dense(units=2, input_dim=2, activation='relu'))
nn.add(Dense(units=2, activation='softmax'))

# setup early stopping callback function
earlystop = keras.callbacks.EarlyStopping(
    monitor='val_loss',       # look at the validation loss
    min_delta=0.0001,         # threshold to consider as no change
    patience=5,               # stop if 5 epochs with no change
    verbose=1, mode='auto'
)
callbacks_list = [earlystop]

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(lr=0.5, momentum=0.9, nesterov=True))
history = nn.fit(X, Yb, epochs=100, batch_size=32, validation_split=0.1,
                 verbose=False,
                 callbacks=callbacks_list)  # setup the callback list
```

```
Epoch 00025: early stopping
```

In [54]:

```
nnfig
```

Out[54]:



Universal Approximation Theorem

- Cybenko (1989), Hornik (1991)
  - *A multi-layer perceptron with a single hidden layer and a finite number of nodes can approximate any continuous function.*
    - The number of nodes needed might be very large.
    - Doesn't say anything about how difficult it is to train it.
- Deep learning corrolary
  - A deep network can learn the same function using less nodes.
  - Given the same number of nodes, a deep network can learn more complex functions.
    - Doesn't say anything about how difficult it is to train it.

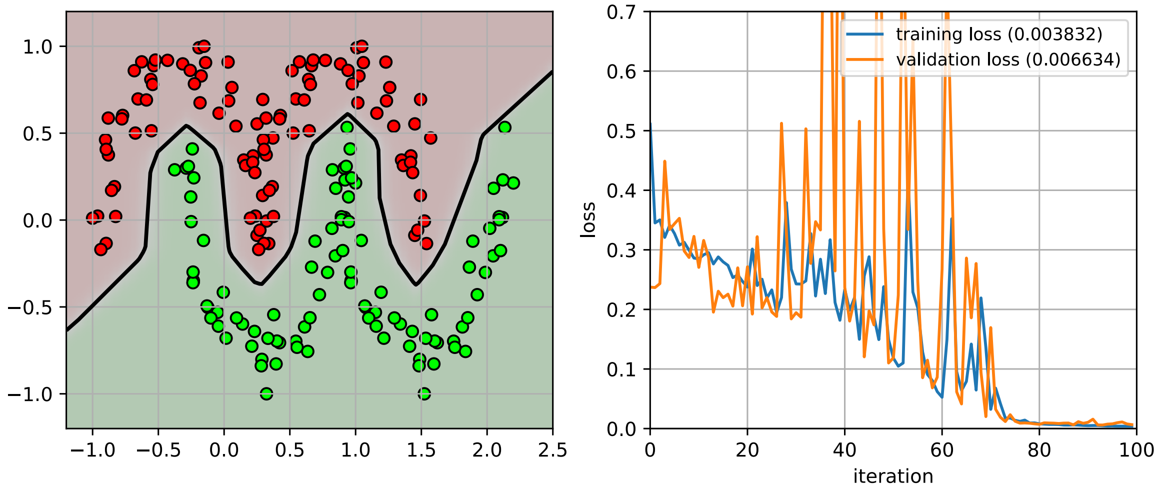
Example

- Network with 1 hidden layer
  - input (2D) -> 40 hidden nodes -> output (2D)

In [56]:
```
# initialize random seed
random.seed(4487); tensorflow.set_random_seed(4487)

# build the network
nn = Sequential()
nn.add(Dense(units=40, input_dim=2, activation='relu'))
nn.add(Dense(units=2, activation='softmax'))

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(lr=0.4, momentum=0.9, nesterov=True))
history = nn.fit(X2, Y2b, epochs=100, batch_size=32, validation_split=0.1, verbo
se=False)
```

In [58]:
```
nnfig
```

Out[58]:



In [59]:
```
nn.summary()
```

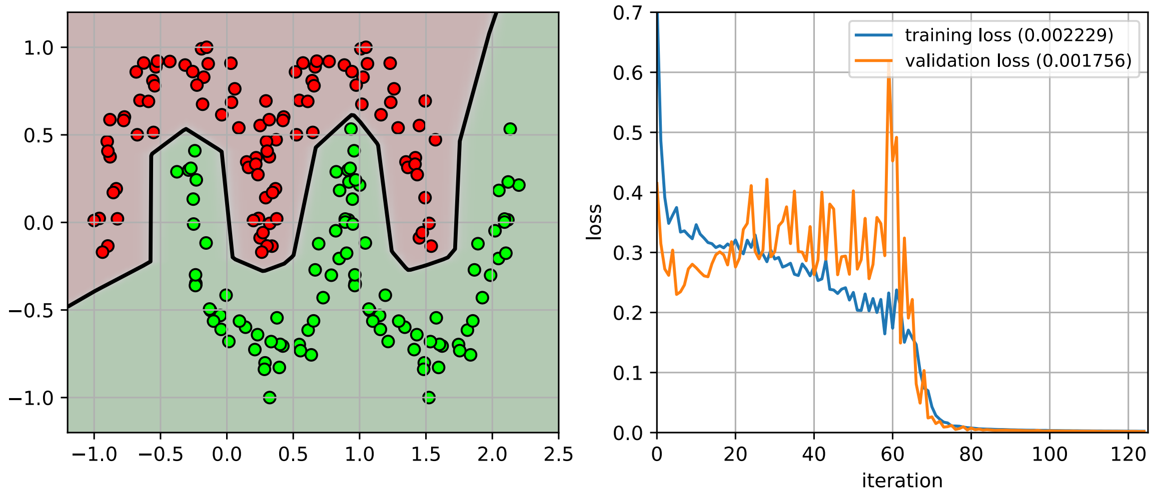
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_13 (Dense)             (None, 40)                120       
_________________________________________________________________
dense_14 (Dense)             (None, 2)                 82        
=================================================================
Total params: 202
Trainable params: 202
Non-trainable params: 0
_________________________________________________________________
```

- 3 hidden layers:
  - input (2D) -> 8 nodes -> 5 nodes -> 3 nodes -> output (2D)

In [66]:
```
# initialize random seed
random.seed(449); tensorflow.set_random_seed(4488)

# build the network
nn = Sequential()
nn.add(Dense(units=8, input_dim=2, activation='relu'))
nn.add(Dense(units=5, activation='relu'))
nn.add(Dense(units=3, activation='relu'))
nn.add(Dense(units=2, activation='softmax'))

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(lr=0.07, momentum=0.9, nesterov=True))
history = nn.fit(X2, Y2b, epochs=125, batch_size=32, validation_split=0.1, verbo
se=False)
```

In [68]:
```
nnfig
```

Out[68]:



In [69]:
```
nn.summary()
# less parameters, similar classifier.
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 8)                 24
_________________________________________________________________
dense_24 (Dense)             (None, 5)                 45
_________________________________________________________________
dense_25 (Dense)             (None, 3)                 18
_________________________________________________________________
dense_26 (Dense)             (None, 2)                 8
=================================================================
Total params: 95
Trainable params: 95
Non-trainable params: 0
_________________________________________________________________
```