COMPSCI 689
Lecture 11: Multilayer Perceptrons

Benjamin M. Marlin

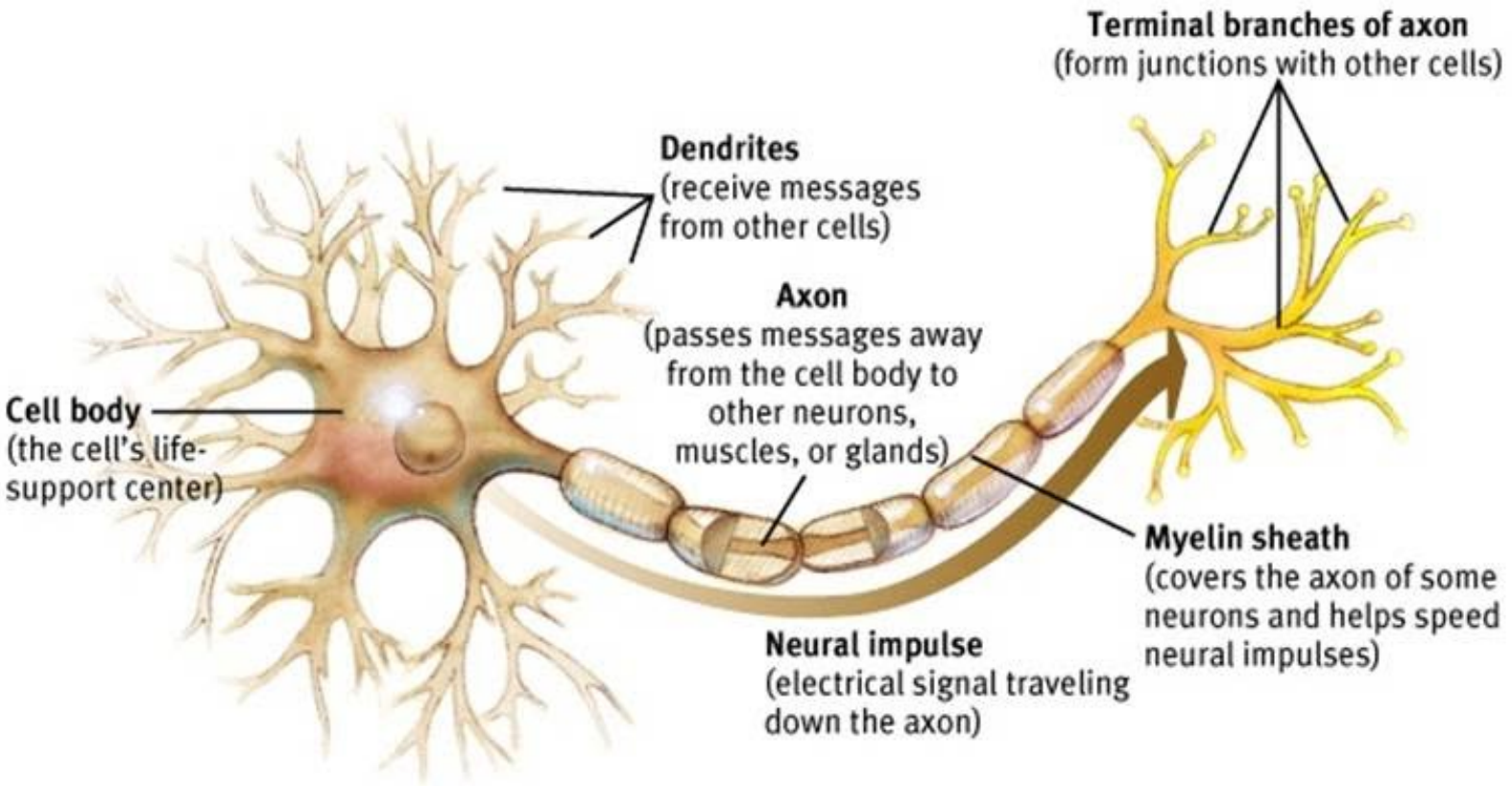
College of Information and Computer Sciences
University of Massachusetts Amherst

Slides by Benjamin M. Marlin (marlin@cs.umass.edu).

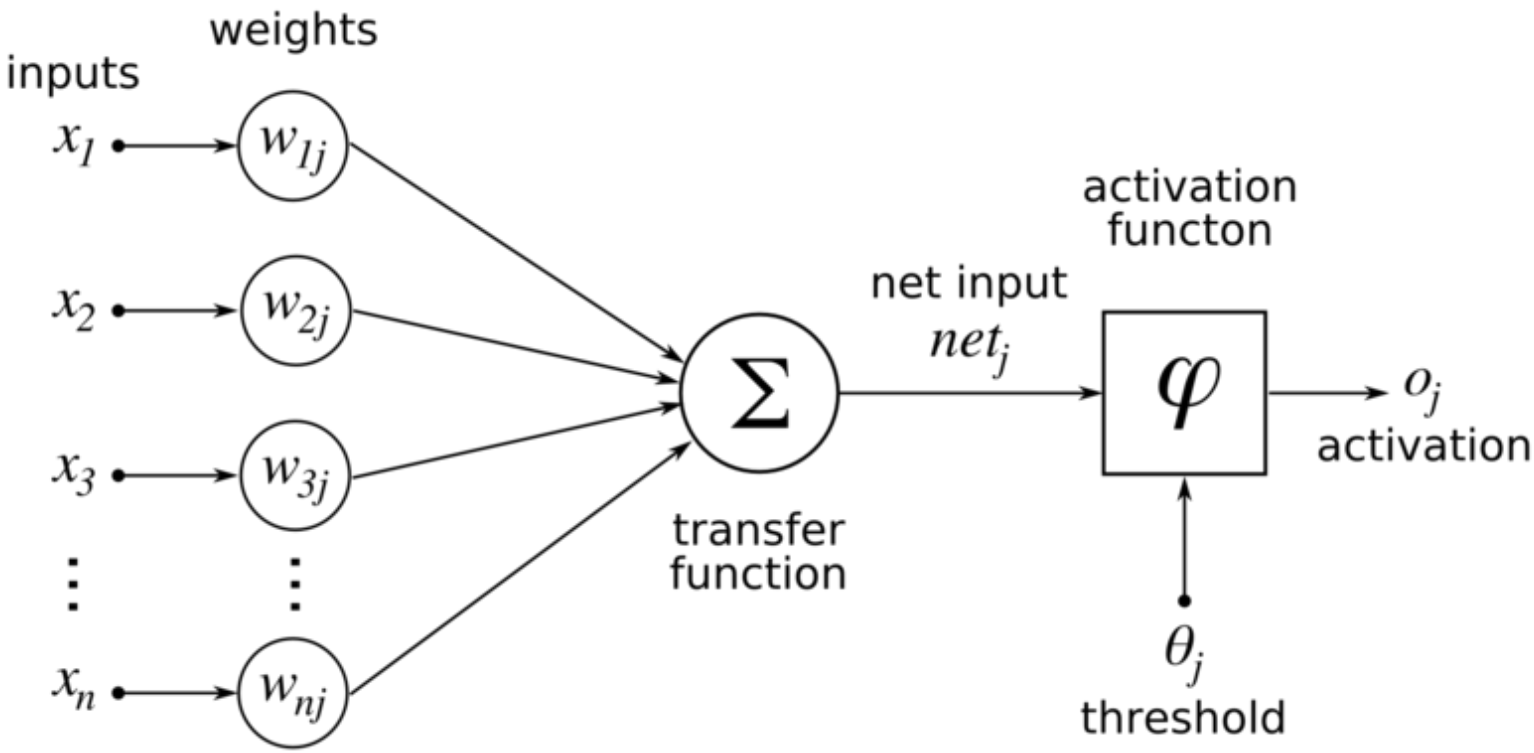
The Trouble with Kernels

- We have previously seen how basis expansions and kernels can be used to increase the capacity of linear models, turning them into models that are non-linear in the features while remaining linear in the parameters.
- **Question:** What is the primary weakness of this approach?
- In this lecture, we'll see how neural networks have the potential to learn appropriate feature representations from data at the same time they learn to solve regression and classification problems.
- We'll start with some historical background...

Biological Neurons



McCulloch and Pitts Neuron (1943)



Assuming $\varphi(x) = \text{sign}(x)$, what model is this?

Assuming $\varphi(x) = x$, what model is this?

The Perceptron (1950)

The Perceptron is a simple online algorithm for adapting the weights in a McCulloch/Pitts neuron with $\text{sign}()$ activation. It was developed in the 1950s by Rosenblatt at Cornell.

Algorithm PERCEPTRONTRAIN(**D**, *MaxIter*)

```
w_d ← 0, for all d = 1 ... D                          // initialize weights
b ← 0                                                  // initialize bias
for iter = 1 ... MaxIter do
    for all (x,y) ∈ D do
        a ← Σ_{d=1}^{D} w_d x_d + b                    // compute activation for this example
        if ya ≤ 0 then
            w_d ← w_d + y x_d, for all d = 1 ... D     // update weights
            b ← b + y                                  // update bias
        end if
    end for
end for
return w_0, w_1, ..., w_D, b
```

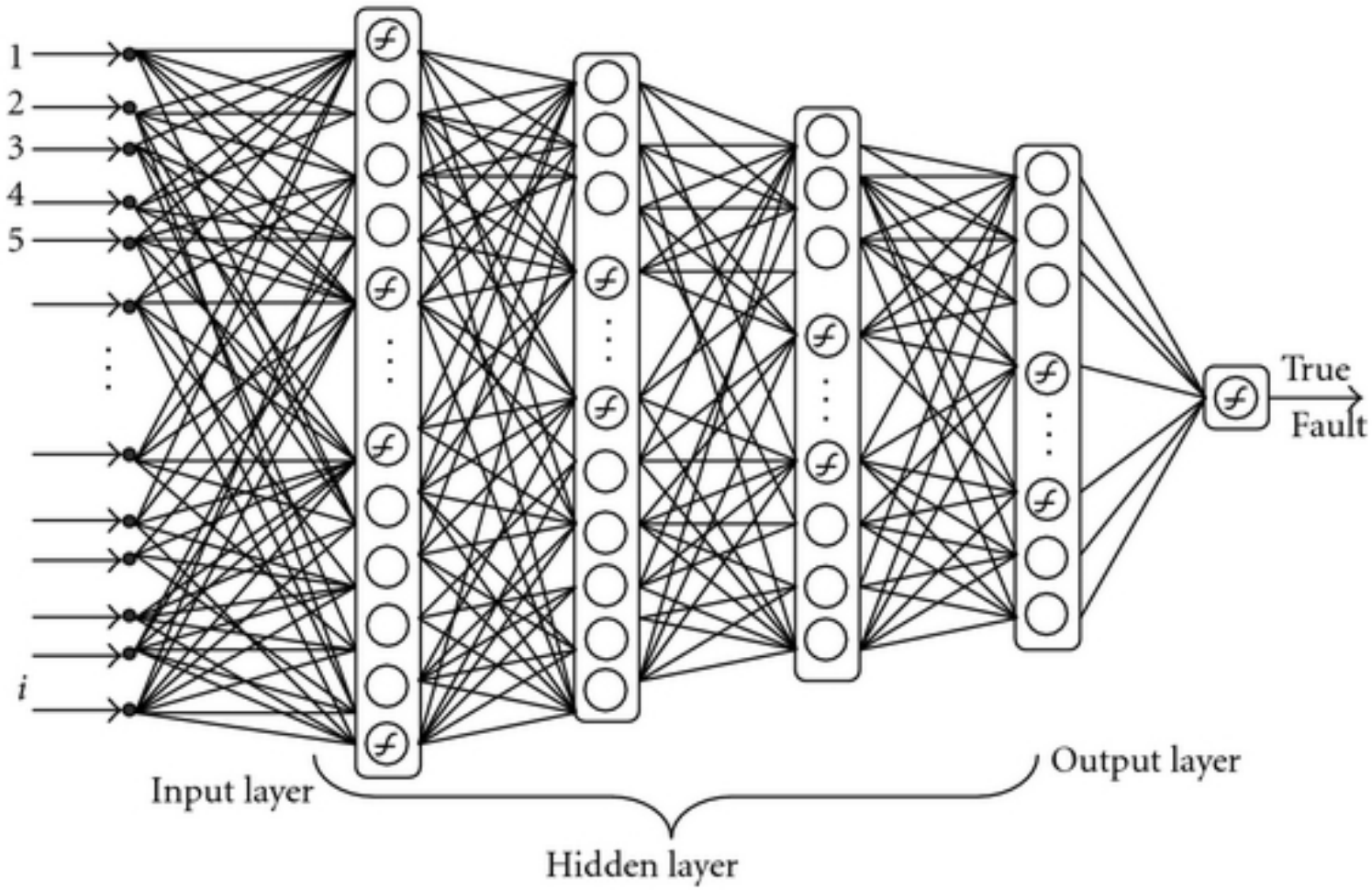
Perceptron Limitations

- This particular algorithm converges to a random separating hyperplane when the data are linearly separable.
- However, the algorithm is not derived from optimizing an objective function and does not converge when data are not linearly separable.
- ERM/RRM for linear models typically gives classification results with improved generalization performance compared to this approach in the separable case, and can also handle the non-separable case correctly.

Perceptron Representational Limitations

- In 1969, Minsky and Papert at MIT popularized a set of arguments showing that the single-layer perceptron could not learn certain classes of functions (including XOR).
- However, Minsky and Papert also showed that more complex functions could be represented using a *multi-layer perceptron* or MLP.
- Unfortunately, at the time, no algorithms were known that could learn such networks from data.

Multi-Layer Perceptron



Sigmoid Neural Networks (1980s)

The eventual solution to MLP learning for classification was to:

1. Make the hidden layer non-linearities smooth (sigmoid/logistic) functions.
2. Make the output layer non-linearity a differentiable function.
3. Use a differentiable loss function.
4. Learn using the *backpropagation algorithm*, which was popularized by Rumelhart, Hinton and Williams in the 1980s (backprop is just vanilla gradient descent on the ERM objective defined by the loss).

Example: 1-Hidden Layer Sigmoid NN Classifier

- The simplest binary classification architecture of this type is a 1-hidden layer sigmoid neural network:

$$h_k = \frac{1}{1 + \exp(-(\mathbf{x} \cdot \mathbf{w}_k^1 + b_k^1)}$$

$$\mathbf{h} = [h_1, ..., h_K]$$

$$g_\theta(\mathbf{x}) = \mathbf{h} \cdot \mathbf{w}^o + b^o$$

- We assume K hidden units and D input features. h_k is the value of the k^{th} hidden unit. $g_\theta(\mathbf{x})$ is the discriminant value output by the network. The predicted class is $\hat{y} = \text{sign}(g_\theta(\mathbf{x}))$.
- The parameters of the model are a weight vector $\mathbf{w}_k^1 \in \mathbb{R}^D$ and bias $b_k \in \mathbb{R}$ for each hidden unit k, and a weight vector $\mathbf{w}^o \in \mathbb{R}^K$ and a bias b^o for the output.

Binary Sigmoid NN Learning Problem

- Binary Sigmoid NN Classifiers can be learned in the ERM framework using the logistic loss (assuming $y \in \{-1, 1\}$): $L_{log}(y, d) = \log(1 + \exp(-yd))$.
- For the 1-hidden layer sigmoid neural network model with parameters $\theta = [\mathbf{w}^1_{1:K}, b^1_{1:K}, \mathbf{w}^o, b^o]$, the learning problem is to minimize $R(\theta, \mathcal{D})$ with respect to θ:

$$R(\theta, \mathcal{D}) = \sum_{n=1}^{N} \log(1 + \exp(-y_n g_\theta(\mathbf{x}_n)))$$
$$g_\theta(\mathbf{x}_n) = \mathbf{h}_n \mathbf{w}^o + b^o$$
$$h_{kn} = \sigma(\mathbf{x}_n \mathbf{w}^1_k + b^1_k)$$

- Where $\sigma(x) = 1/(1 + \exp(-x))$.

Binary Sigmoid NN Classifier

- This model can be viewed as simultaneously learning a basis expansion and a logistic regression classifier in this new basis.
- The hidden unit values are the basis elements.
- The output layer is simply a logistic regression model applied to the basis expansion given by the hidden unit values.
- However, the basis expansion is itself a parametric function that can be learned from data.

Binary Sigmoid NN Learning Problem

The objective function for the learning problem is as shown below:

$$R(\theta, \mathcal{D}) = \sum_{n=1}^{N} \log(1 + \exp(-y_n d_n))$$
$$d_n = \mathbf{h}_n \mathbf{w}^o + b^o$$
$$h_{kn} = \sigma(\mathbf{x}_n \mathbf{w}_k^1 + b_k^1)$$

Where $\sigma(x) = 1/(1 + \exp(-x))$.

Gradients for Binary Sigmoid NN Classifiers

- First, $\sigma'(x) = \sigma(x)(1-\sigma(x))$
- $\frac{\partial R(\theta,\mathcal{D})}{\partial d_n} = \frac{1}{1+\exp(-y_n d_n)}\exp(-y_n d_n)(-y_n) = -(1-\sigma(y_n d_n))y_n$
- $\frac{\partial d_n}{\partial w_k^o} = \frac{\partial}{\partial w_k^o}(\mathbf{h}_n \mathbf{w}^o + b^o) = h_{nk}$
- $\frac{\partial d_n}{\partial h_{nk}} = \frac{\partial}{\partial h_{nk}}(\mathbf{h}_n \mathbf{w}^o + b^o) = w_k^o$
- $\frac{\partial h_{nk}}{\partial w_{dk}^1} = \frac{\partial}{\partial w_{dk}^1}\sigma(\mathbf{x}_n \mathbf{w}_k^1 + b_k^1) = \sigma'(\mathbf{x}_n \mathbf{w}_k^1 + b_k^1)x_{nd} = h_{nk}(1-h_{nk})x_{nd}$
- Thus $\frac{\partial R(\theta,\mathcal{D})}{\partial w_k^o} = \sum_{n=1}^N \frac{\partial R(\theta,\mathcal{D})}{\partial d_n}\frac{\partial d_n}{\partial w_k^o} = \sum_{n=1}^N y_n(\sigma(y_n d_n)-1)h_{nk}$
- and $\frac{\partial R(\theta,\mathcal{D})}{\partial w_{dk}^1} = \sum_{n=1}^N \frac{\partial R(\theta,\mathcal{D})}{\partial d_n}\frac{\partial d_n}{\partial h_{nk}}\frac{\partial h_{nk}}{\partial w_{dk}^1} =$
  $\sum_{n=1}^N y_n(\sigma(y_n d_n)-1)(w_k^o)h_{nk}(1-h_{nk})x_{nd}$
- Nearly identical results hold with respect to the biases.

Learning for Sigmoid NN Regression

- Learning sigmoid neural networks for regression follows the same basic pattern.
- All we need to do is switch to a regression loss. The model defined so far already has a linear output layer.
- The derivation of the gradient is nearly identical regardless of the loss used.

Regularization

- Like with linear regression, logistic regression, and SVMs, regularizing neural networks typically improves generalization performance.
- Standard regularizers can be added to the objective function to accomplish this. (i.e. $+\lambda\|\mathbf{w}\|_2^2$).
- In the neural networks literature, the addition of two-norm regularization is sometimes referred to as *weight decay* because of the contribution to the learning step (i.e., $-\lambda\mathbf{w}$).
- The number of layers and the width of the layers are also both important architectural hyper-parameters that impact model capacity.

Optimization Details

- **Optimizers:** Second-order optimizers are rarely used to learn neural network models. Instead, first-order methods are typically used (i.e., gradient descent).
- **Stochastic Approximation:** It is also typical to compute gradients using sub-sets of the data. This typically speeds convergence. In this literature, this is often referred to as *mini-batch* learning or *stochastic gradient descent*.
- **Step Sizes:** Fixed step sizes or fixed step size decay schedules were often used in early work. In this literature, the step size is often referred to as the *learning rate*.
- **Acceleration:** There are a wide array of approaches that either modify the gradient direction or the step sizes to attempt to achieve faster convergence on certain classes of problems (i.e, use of momentum, Adadelta, Adagrad, Adam, RMSprop, etc.).

Sigmoid Neural Network Limitations

- Unlike SVMs, logistic regression and OLS linear regression, multi-layer sigmoid neural network models have many local optima.
- In practice, it can be hard to find a combination of layer sizes and regularization that yields good generalization performance when learning from small or moderate amounts of data.
- In addition, the use of sigmoid functions for the hidden units results in a *vanishing gradient* problem that can further slow down learning for deep networks.
- As a result, there were few examples of deep neural network models that resulted in performance that exceeded hand-crafted features, basis expansions and kernels until the last decade.

Towards Modern Deep Learning

1. Address the vanishing gradient problem using rectified linear units: $relu(x) = max(0, x)$.
2. Have access to lots of labeled data (ie: millions of examples).
3. Do the computing on GPUs to achieve significant speedups (i.e.: model training takes days instead of weeks or months).
4. The breakthrough paper in this area is Krizhevsky, Sutskever, and Hinton. *Imagenet classification with deep convolutional neural networks.* Advances in neural information processing systems 25, 2012.

Deep Learning Modeling Tools

- As we have seen, deep learning models just require (sub-) gradients of the loss with respect to the parameters to enable learning.
- However, correctly deriving gradients from complex model architectures by hand can be tedious and is error prone.
- As a result, many deep learning tools exist (i.e., PyTorch, TensorFlow, Caffe, Keras) that allow models to be specified only in terms of the *forward pass* computation (from inputs to loss).
- This makes it much easier to quickly change the model architecture because only the specification of the forward pass needs to be updated.

Autodiff

- An idea called *automatic differentiation* is used to convert the forward pass specification into the gradient computations needed for learning.
- A computation graph is extracted from the specification of the forward pass.
- The chain rule is then applied to each node in the computation graph to transform it into a computation graph for computing the gradient of the objective.
- Importantly, this procedure yields analytic gradients and learning is identical to classical backprop.
- As a byproduct of the representations used, it's possible to compile both computation graphs against arbitrary numerical libraries for either CPUs or GPUs.