EECS 553: Machine Learning (ECE) **University of Michigan**

Multilayer Perceptrons

Winter 2023 **Clayton Scott**

Feedforward neural networks (FNNs) are nonlinear function models whose parameters can be learned in the context of supervised and other learning problems. FNNs are so named because they can be evaluated by feeding the input into the network and propagating values "forward" until the output is produced. In these notes we will focus on FNNs for supervised learning, namely, classification and regression. We will focus specifically on a particular type of FNN, the multilayer perceptrons (MLP). During lecture we will also discuss a different type of FNN called a convolutional neural network (CNN). CNNs offer state-of-the-art performance on a number of image analysis and vision tasks, while MLPs are among the best general-purpose methods for data without spatial or temporal structure.[1]

1 Multilayer Perceptrons

An MLP is a function that is represented by a set of layers, each consisting of some number of nodes. Edges connect all nodes in consecutive layers, with each edge having its own learnable weight. The number of nodes in the input layer is the feature space dimension, while the number of nodes in the output layer is the number of classes (for classification) or the dimension of the response variable (in regression). The layers between the input and output layers are called hidden layers. The number of layers, and the number of nodes in each layer are user-defined. Together, the number and sizes of the hidden layers determine the MLP's *architecture*.

To evaluate the function, information flows from the input layer to the output layer. A feature vector is fed into the input layer. Then, variables at the first hidden layer are computed by taking linear combinations of the values at the previous layer according to the edge weights, followed by the application of a nonlinear activation function. This process is repeated until the output layer values are determined. The output layer does not have an activation function. An MLP with two hidden layers is depicted in Figure 1.

To make this precise, we use the notation in Table 1. The output $f(\boldsymbol{x})$ of an MLP is computed from the input $\boldsymbol{x}$ by calculating

$\boldsymbol{z}_0 = \boldsymbol{x}$
For $\ell = 1, \ldots, L-1$
  $\boldsymbol{a}^{(\ell)} = \boldsymbol{W}^{(\ell)} \boldsymbol{z}^{(\ell-1)}$
  $\boldsymbol{z}^{(\ell)} = \sigma(\boldsymbol{a}^{(\ell)})$, applied element-wise
End
$f(\boldsymbol{x}) = \boldsymbol{a}^{(L)} = \boldsymbol{W}^{(L)} \boldsymbol{z}^{(L-1)}$

Notice that the activation function is not applied at the final layer. The process of calculating $f(\boldsymbol{x})$ from $\boldsymbol{x}$ is referred to as a *forward pass*. If desired, the constant 1 can be prepended to any (or all) $\boldsymbol{z}^{(\ell)}$, $0 \leq \ell \leq L-1$, and a column of weights added to the corresponding $\boldsymbol{W}^{(\ell)}$, to add bias terms.

A classical choice for the activation function is a sigmoid (or "S"-shaped) function, such as the logistic sigmoid (see Figure 2)

$$\sigma(t) = \frac{1}{1+e^{-t}},$$

or the hyperbolic tangent function

$$\sigma(t) = \frac{1-e^{-t}}{1+e^{-t}}.$$

[1] Later we will discuss recurrent neural networks which excel at prediction with temporally structured data.

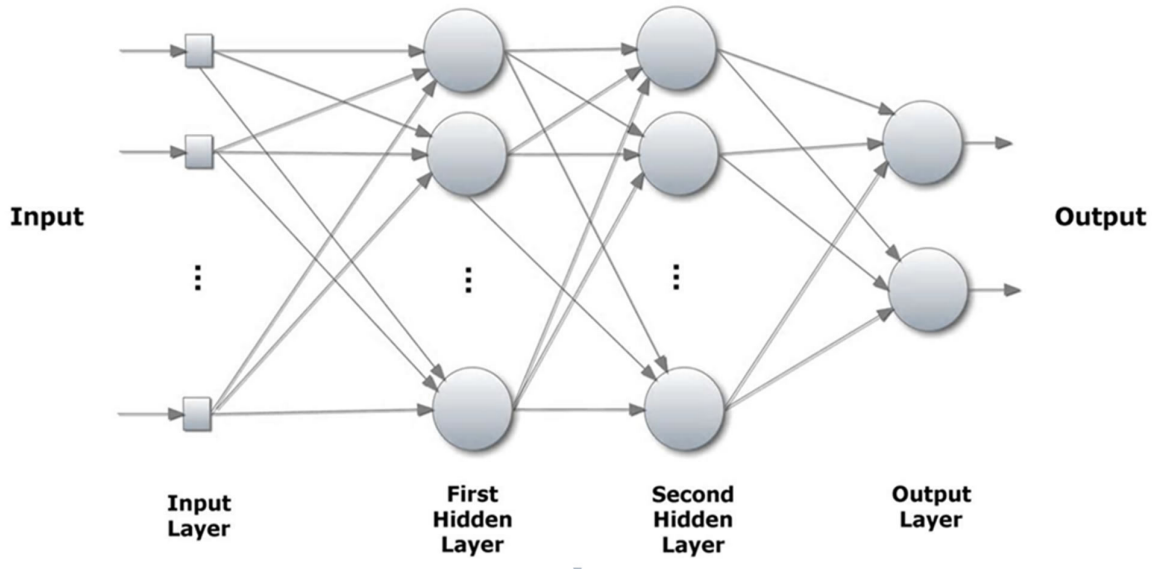

Figure 1: An example of an MLP with two hidden layers.

Table 1: Notation.

Symbol	meaning
$\ell \in \{0, 1, \ldots, L\}$	layer indices, where layer 0 is the input layer, and layer L the output layer
d_ℓ	the number of nodes in layer ℓ
$\boldsymbol{x} \in \mathbb{R}^{d_0}$	feature vector, input to neural network
$\boldsymbol{W}^{(\ell)}$	the $d_\ell \times d_{\ell-1}$ matrix of weights connecting layer $\ell - 1$ and layer ℓ
σ	activation function
$\boldsymbol{a}^{(\ell)} \in \mathbb{R}^{d_\ell}$	layer ℓ values before applying activation function, $1 \leq \ell \leq L$; a function of $\boldsymbol{x}$
$\boldsymbol{z}^{(\ell)} \in \mathbb{R}^{d_\ell}$	layer ℓ values after applying activation function, $0 \leq \ell \leq L - 1$; a function of $\boldsymbol{x}$
$f_k(\boldsymbol{x})$	kth output node value, $= a_k^{(L)}$
$f(\boldsymbol{x})$	output of neural network, $= \boldsymbol{a}^{(L)}$

The rectified linear unit (ReLU), $\sigma(t) = \max(0, t)$, is a common choice for modern deep learning applications such as with CNNs.

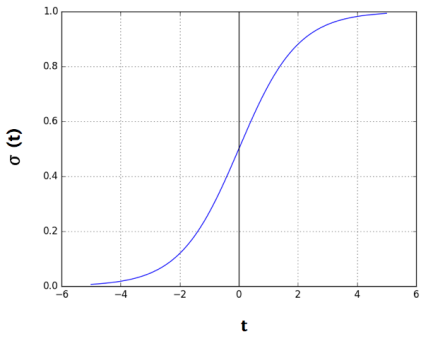

Figure 2: The logistic sigmoid function

For regression, d_L is the dimension of the output $\boldsymbol{y}$. So far in the course we have focused regression with scalar outputs, but neural networks handle multi-dimensional outputs seamlessly.

For classification, $d_L = K$, where K is the number of classes. The kth class in represented as the vector $\boldsymbol{e}_k$ of size K with a 1 in the kth position and zeros elsewhere. The output $f(\boldsymbol{x})$ is converted to a probability vector, representing the conditional probability of each class given $\boldsymbol{x}$, using the *softmax* function. In particular, for $\boldsymbol{v} \in \mathbb{R}^K$, define $\psi(\boldsymbol{v}) \in \mathbb{R}^K$ by $\psi(\boldsymbol{v}) = (\psi_1(\boldsymbol{v}), \ldots, \psi_K(\boldsymbol{v}))$, where

$$\psi_k(\boldsymbol{v}) = \frac{\exp(v_k)}{\sum_j \exp(v_j)}.$$

Then the class probabilities associated to $\boldsymbol{x}$ are given by $\psi(f(\boldsymbol{x}))$. Intuitively, the entries of $f(\boldsymbol{x})$ give the probabilities that $\boldsymbol{x}$ is associated with each class. Ideally $f(\boldsymbol{x})$ would equal $\boldsymbol{e}_k$ when $\boldsymbol{x}$ belongs to class k, but the softmax function assigns some nonzero probability to each class.

Let us denote the components of the neural network output by $f(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_K(\boldsymbol{x}))$. Then the classifier defined by the neural network is

$$\boldsymbol{x} \longmapsto \arg\max_{\boldsymbol{x}} f_k(\boldsymbol{x}).$$

2 Training Neural Networks via Backpropagation

The weights of a neural networks are typically trained by empirical risk minimization, using the least-squares loss for regression and cross-entropy loss for classification. The empirical risk is typically minimized using gradient descent, or some variation such as stochastic gradient descent. In this section we will derive backpropagation, which is an algorithm for efficiently implementing gradient descent (and its variants).

Consider training data $(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, (\boldsymbol{x}_N, \boldsymbol{y}_N)$. Let $\boldsymbol{\theta}$ denote the complete set of parameters to be estimated,

$$\boldsymbol{\theta} = (\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)}).$$

The objective function depends on the setting (ignoring the leading factor of $1/N$):

- Regression, scalar output (switching to non-bolded notation y_i)

$$R(\boldsymbol{\theta}) = \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n))^2$$

- Regression, vector output (generalizing the previous case)

$$R(\boldsymbol{\theta}) = \sum_{n=1}^{N} \|\boldsymbol{y}_n - f(\boldsymbol{x}_n)\|^2$$

- K-class Classification: Denote $\boldsymbol{y}_n = [y_{n1} \ \cdots \ y_{nK}]^T$, vector with 0 everywhere except a 1 in the position associated to the label of $\boldsymbol{x}_n$. The empirical risk based on the cross-entropy loss is

$$\begin{aligned} R(\boldsymbol{\theta}) &= -\sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \log \psi_k(f(\boldsymbol{x}_n)) \\ &= -\sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \log \left(\frac{\exp f_k(\boldsymbol{x})}{\sum_j \exp(f_j(\boldsymbol{x}))} \right). \end{aligned}$$

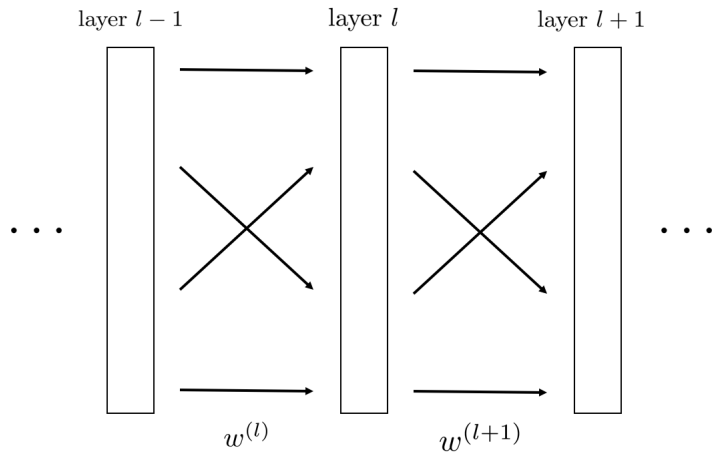

Figure 3: A general layer. The weight matrices should be capitalized.

A general layer of the network is shown in Figure 3.
Regardless of the loss, the objective function can be written

$$R(\boldsymbol{\theta}) = \sum_{n=1}^{N} R_n(\boldsymbol{\theta})$$

where $R_n(\boldsymbol{\theta})$ can easily be deduced from the above cases depending on the loss. Thus we may focus our attention on calculating $\nabla_{\boldsymbol{\theta}} R_n(\boldsymbol{\theta})$.

Additional notation is introduce in Table 2. It is important to keep in mind that the values $a_{ni}^{(\ell)}$ and

Table 2: More notation.

Symbol	meaning
$w_{ij}^{(\ell)}$	(i,j) entry of $\boldsymbol{W}^{\ell}$
$a_{ni}^{(\ell)}$	ith entry of $\boldsymbol{a}^{(\ell)}$ when input is $\boldsymbol{x}_n$
$z_{ni}^{(\ell)}$	ith entry of $\boldsymbol{z}^{(\ell)}$ when input is $\boldsymbol{x}_n$

$z_{ni}^{(\ell)}$ are functions of $\boldsymbol{\theta}$ even though this is not reflected in the notation. Using this notation, we have from definitions

$$a_{ni}^{(\ell)} = \sum_j w_{ij}^{(\ell)} z_{nj}^{(\ell-1)} \tag{1}$$

$$z_{ni}^{(\ell)} = \sigma(a_{ni}^{(\ell)}). \tag{2}$$

The gradient is comprised of partial derivatives of the form

$$\frac{\partial R_n(\boldsymbol{\theta})}{\partial w_{ij}^{(\ell)}}.$$

To find these partial derivatives, consider an arbitrary layer ℓ, $1 \leq \ell < L$. The output layer will be treated as a special case later on. Since $a_{ni}^{(\ell)}$ is a function of $w_{ij}^{(\ell)}$ for each i and j, we can apply the chain rule to

obtain

$$\begin{aligned}\frac{\partial R_n(\boldsymbol{\theta})}{\partial w_{ij}^{(\ell)}} &= \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(\ell)}} \frac{\partial a_{ni}^{(\ell)}}{\partial w_{ij}^{(\ell)}} \\ &= \delta_{ni}^{(\ell)} z_{nj}^{(\ell-1)}\end{aligned}$$

where we have used (1) and introduced the notation

$$\delta_{ni}^{(\ell)} := \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(\ell)}}.$$

The main idea behind the backpropagation algorithm is to compute $\delta_{ni}^{(L)}$ starting at the output layer, and to work back toward the input layer. In particular, we will develop a formula for $\delta_{ni}^{(\ell)}$, $1 \leq \ell < L$, in terms of $\delta_{nj}^{(\ell+1)}$.

By the chain rule for partial derivatives,

$$\begin{aligned}\delta_{ni}^{(\ell)} &= \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{ni}^{(\ell)}} \\ &= \sum_k \frac{\partial R_n(\boldsymbol{\theta})}{\partial a_{nk}^{(\ell+1)}} \frac{\partial a_{nk}^{(\ell+1)}}{\partial a_{ni}^{(\ell)}} \\ &= \sum_k \delta_{nk}^{(\ell+1)} \frac{\partial a_{nk}^{(\ell+1)}}{\partial a_{ni}^{(\ell)}},\end{aligned}$$

where k indexes the nodes in layer $\ell + 1$. Since

$$a_{nk}^{(\ell+1)} = \sum_j w_{kj}^{(\ell+1)} z_{nj}^{(\ell)} = \sum_j w_{kj}^{(\ell+1)} \sigma(a_{nj}^{(\ell)}),$$

we have

$$\frac{\partial a_{nk}^{(\ell+1)}}{\partial a_{ni}^{(\ell)}} = w_{ki}^{(\ell+1)} \sigma'(a_{ni}^{(\ell)}).$$

In summary,

$$\delta_{ni}^{(\ell)} = \sum_k \delta_{nk}^{(\ell+1)} w_{ki}^{(\ell+1)} \sigma'(a_{ni}^{(\ell)}).$$

It remains to determine $\delta_{ni}^{(L)}$.

2.1 Output layer gradient: regression

We consider the case of scalar outputs y_n. The vector case is similar and is left as an exercise. Then

$$R_n(\boldsymbol{\theta}) = (y_n - f(\boldsymbol{x}_n))^2 = (y_n - a_{n1}^{(L)})^2,$$

and so

$$\begin{aligned}\delta_{n1}^{(L)} &= \frac{\partial R(\boldsymbol{\theta})}{\partial a_{n1}^{(L)}} \\ &= -2(y_n - a_{n1}^{(L)}).\end{aligned}$$

2.2 Output layer gradient: classification

In this case

$$\begin{aligned} R_n(\boldsymbol{\theta}) &= -\sum_k y_{nk} \log(\psi_k(f(\boldsymbol{x}_n))) \\ &= -\sum_k y_{nk} \log\left(\frac{\exp(\boldsymbol{a}_{nk}^{(L)})}{\sum_j \exp(\boldsymbol{a}_{nj}^{(L)})}\right). \end{aligned}$$

Note that only one term in the sum is nonzero. Then $\delta_{ni}^{(L)}$ can be computed using the identity

$$\frac{\partial \psi_k(\boldsymbol{a})}{\partial a_i} = \begin{cases} \psi_k(\boldsymbol{a})(1-\psi_i(\boldsymbol{a})) & i = k \\ -\psi_k(\boldsymbol{a})\psi_i(\boldsymbol{a}) & i \neq k \end{cases},$$

which in turn may be established using the quotient rule. The details are left as an exercise.

2.3 Backpropagation

Putting everything together, we may now state the backpropagation algorithm for efficiently computing the gradient of $\boldsymbol{\theta}$ with respect to $R_n(\boldsymbol{\theta})$ for a fixed n.

Forward pass:
  Using current weights $\boldsymbol{\theta}$ compute $f(\boldsymbol{x}_n)$
  and store intermediate values $a_{ni}^{(\ell)}, z_{nj}^{(\ell)}$
Initialize backward pass:
  For $i = 1$ to d_L
    Compute $\delta_{ni}^{(L)}$
  End
Backward pass:
  For $\ell = L-1$ downto 1
    For $i = 1$ to d_ℓ
      $\delta_{ni}^{(\ell)} = \sum_k \delta_{nk}^{(\ell+1)} w_{ki}^{(\ell+1)} \sigma'(a_{ni}^{(\ell)})$
      For $j = 1$ to $d_{\ell-1}$
        $\frac{\partial R_n(\boldsymbol{\theta})}{\partial w_{ij}^{(\ell)}} \longleftarrow \delta_{ni}^{(\ell)} z_{nj}^{(\ell-1)}$
      End
    End
  End

3 Implementation Details

There are many practical considerations when training neural networks. Here are some things to keep in mind.

1. The above derivation assumes σ is differentiable. If σ is not differentiable, it's derivative is replaced by a subgradient of σ. Note that this does not necessarily lead to a subgradient of the overall objective; subgradients are not guaranteed to exist for nonconvex functions.

2. If using the sigmoid activation function, to avoid saturating the sigmoids, data should be preprocessed to have zero mean and unit variance

3. In general, $R(\boldsymbol{\theta})$ will have several local minima. It has been observed empirically that random initialization of the weights leads to better performance than deterministic initialization, such as all zeros.

4. Over-fitting is a concern when training neural networks. Therefore some form of regularization is needed. One approach is early stopping, i.e., stopping backpropagation before reaching a local min. In the case of the sigmoid activiation function, this is based on the following intuition: In the initial iterations, when the weights are near zero, the model is approximately linear because $\sigma(t)$ is approximately linear near $t = 0$. As the weights increase, the model becomes more nonlinear. Thus, early stopping can be thought of as regularization toward a linear model.

5. Another approach to regularization is *dropout*, in which a set fraction of weights are randomly selected to be updated at each iteration of gradient descent, with the other weights held fixed.

6. Practical implementation issue: Calculating the exponentials in Softmax is numerically unstable, since the values could be extremely large. We can mitigate such problems with a simple trick, namely

$$\begin{aligned}\psi_i(\boldsymbol{a}) &= \frac{e^{a_i}}{\sum_j e^{a_j}} \\ &= \frac{Ce^{a_i}}{C\sum_j e^{a_j}} \\ &= \frac{e^{a_i+\log C}}{\sum_j e^{a_j+\log C}}.\end{aligned}$$

   A common choice for the constant is $\log C = -\max_j a_j$.

We will discuss other aspects of neural networks in future lectures.