BST 261: Data Science II

Lecture 4

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

March 28, 2018

Deep Feedforward Networks

Deep Feedforward Networks

- **Deep feedforward networks**, or **feedforward neural networks**, or **multilayer perceptrons** (MLPs) are the quintessential deep learning models
- The goal of a feedforward network is to approximate some function f^*
- For example, for classifier, a function $y = f(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y; the feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{w}, b)$ and learns the value of the parameters $\boldsymbol{w}, b$ that result in the best function approximation
- These models are called **feedforward** because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$
- There is no feedback connections in feedforward networks, and when those connections are included, the model is called a **recurrent neural network**

Deep Feedforward Networks

- These models are represented by composing together many functions, and this composition can be associated with a directed acyclic graph
- For example, we can have the functions $f^{[1]}$, $f^{[2]}$, and $f^{[3]}$ connected in a chain to form $f(\boldsymbol{x}) = f^{[3]}(f^{[2]}(f^{[1]}(\boldsymbol{x})))$
- Here $f^{[1]}$ is called the **first layer** of the network, $f^{[2]}$ the **second layer**, and $f^{[3]}$ the **third layer**
- Chain structures are the most commonly used structures of neural networks, and the overall length of the chain gives the **depth** of the model
- The final layer of the network is called the **output layer**

Deep Feedforward Networks

- The goal of training is to drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$, the function we would like to approximate with the model
- The training data provides noisy examples of $f^*(\boldsymbol{x})$ and the accompanying label y, and thus specifies what the output layer must do at each point $\boldsymbol{x}$, i.e., to produce y
- Note that the behavior of the other layers is not directly specified by the training data, so the learning algorithm must decide how to use those layers to produce the desired output
- The layer or layers between the input layer and output layer are called **hidden layers**, and the training data does not show the output for these layers

Deep Feedforward Networks

- These models are called **neural** because they are loosely inspired by neuroscience
- Each hidden layer of the network is typically vector valued, and the dimensionality of these layers determines the **width** of the model
- Each element can be seen to play a role analogous to a neuron, where **each unit represents a vector-to-scalar function**

Gradient-Based Learning: Cost Functions and Output Units

- **Sigmoid units** are used as output units when predicting a binary variable y
- The maximum likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$, where the parameter needs to lie in the $[0, 1]$ interval
- While it is possible to use a linear unit and threshold its value to obtain a valid probability, we would not be able to train it very effectively with gradient descent because straying outside the unit interval would cause the gradient to become $\mathbf{0}$
- Instead we use sigmoid output units combined with maximum likelihood, where a sigmoid output unit is defined by

$$\hat{y} = \sigma(z) = \sigma\left(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b\right) \tag{1}$$

Gradient-Based Learning: Cost Functions and Output Units

- Here σ is the logistic sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2}$$

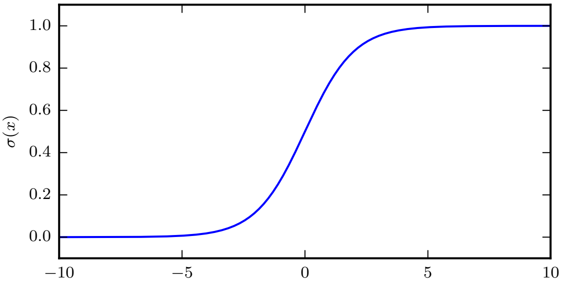
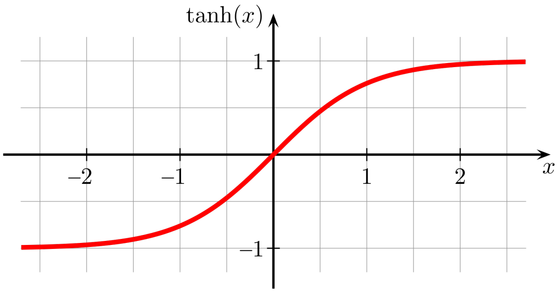


Figure: The logistic sigmoid function is commonly used to produce the parameter of a Bernoulli distribution because of its $(0, 1)$ range. Source: DL.

Hyperbolic Tangent

- Performs better than the logistic sigmoid
- Not as commonly used as the ReLU function

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3}$$

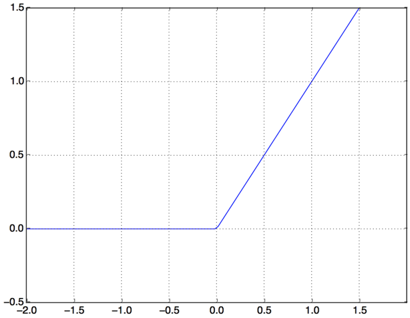


ReLU

- A common activation function for hidden layers is the **Re**ctified **L**inear **U**nit function
- It zeros out negative values

$$g(z) = max(0, z) \tag{4}$$

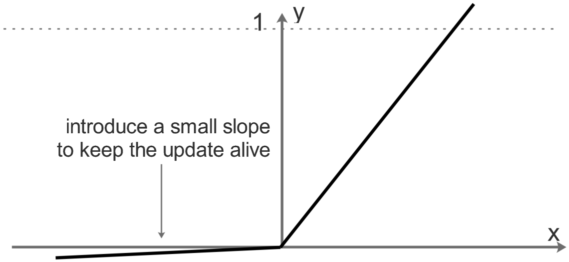
- Derivative is 0 for $x < 0$ and 1 for $x > 0$
- Derivative isn't defined for $x = 0$, but not a problem because of the limited precision of computers



Leaky ReLU

- Usually performs better in practice, but less commonly used

$$g(z) = max(0.01z, z) \tag{5}$$



Gradient-Based Learning: Cost Functions and Output Units

- Here $\zeta(x) = \log(1 + \exp(x))$ is the so-called **softplus** function

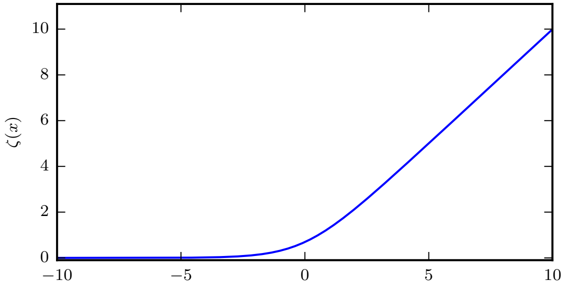


Figure: The softplus function. It's name comes from the fact that it is a smoothed or "softened" version of $x^+ = \max(0, x)$. Source: DL.

Gradient-Based Learning: Cost Functions and Output Units

- **Softmax units** are used as output units when predicting a discrete variable y with k possible values
- In this setting, which can be seen as a generalization of Bernoulli distribution, we need to produce a vector $\hat{\boldsymbol{y}}$ with $\hat{y}_i = P(y = i|\boldsymbol{x})$
- We require not only that each $\hat{y}_i$ lies in the $[0, 1]$ interval, but also that the entire vector sums to 1
- We first use a linear layer to predict unnormalized log probabilities (the same approach as in the Bernoulli case):

$$\boldsymbol{z} = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b} \tag{6}$$

- Here $z_i = \log \tilde{P}(y = i|\boldsymbol{x})$ represents an unnormalized log probability for class i
- The softmax function can then exponentiate and normalize $\boldsymbol{z}$ to obtain $\boldsymbol{y}$
- Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature

Gradient-Based Learning: Cost Functions and Output Units

- Formally, the softmax function is given by

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{7}$$

- In this case we wish to maximize $\log P(y = i; \boldsymbol{z}) = \log \text{softmax}(\boldsymbol{z})_i$

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j) \tag{8}$$

- The first term above shows that the input z_i always has a direct contribution to the cost function and this cannot saturate even if the contribution of z_i to the second term becomes very small
- Because $\log \sum_j \exp(z_j) \approx \max_j z_j$, the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction
- So far the above treatment has been for a single example, but with multiple examples unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome in the training set

Activation Functions

- Rectified linear units are an excellent default choice of hidden units
- Prior to their introduction, most neural networks used one of two activation functions: **logistic sigmoid** $g(z) = \sigma(z)$ or **hyperbolic tangent** $g(z) = \tanh(z)$
- Because sigmoidal units saturate to a high value when z is very positive and to a low value when z is very negative, they are only strongly sensitive to their input when z is near zero
- This saturation can make gradient-based learning difficult, and for this reason their use as *hidden units* in feedforward networks is now discouraged
- Their use as *output units* with gradient-based learning works when an appropriate cost function can undo the saturation of the sigmoid in the output layer
- Sigmoidal activation functions are more common in settings other than feedforward networks (e.g., in recurrent networks)
- Many other types of hidden units are possible, but they are used less frequently
- The ReLU function is by far the most popular and commonly used
- If you're not sure which one to use, try them all and see which works best

Optimization Algorithms

- Recently there have been many proposed variations to GD and SGD:
- RMS prop
- Adam
- Adagrad
- Momentum
- etc.
- For more information visit `http://ruder.io/optimizing-gradient-descent/index.html#rmsprop`

Hyperparameters and Validation Sets

- **Hyperparameters** are parameters that control the learning algorithm's behavior
- They cannot be learned on the training set, because that would always lead to choosing the maximum possible model capacity, leading to overfitting
- To solve this problem, we need a **validation set** of examples that the training algorithm does not observe
- Because no example from the test set can be used in learning, we construct the validation set from the training data by splitting it into two disjoint subsets
- Rule of thumb: use 80% of the training data for training and 20% for validation
- We then fit the model on the training set, and the fitted model is used to predict the responses for observations in the validation set
- The validation set is used to learn the hyperparameters of the model (degree of polynomial, the value of the regularization parameter λ, etc.)
- **Cross-validation** is a commonly used resampling method that divides the data into k folds, disjoint subsets, and uses one of the folds for validation and the remaining $k-1$ folds for training

Deep Feedforward Networks in Python

- Here is generic, 2 layer dense deep feedforward network code

```
# Import all necessary modules
import numpy as np
import keras
from keras import models
from keras import layers

# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=( n, n
    )))
# Layer 2 (Output layer)
network.add(layers.Dense(d, activation='activation function'))

# Create (compile) the network
network.compile(optimizer='optimizing algorithm',
# Include optimizing function, loss
                loss='loss function',
# function and performance measure
                metrics=['performance measure'])

# Train (learn) the network, specify batch size and number of epochs
network.fit(train_data, train_labels, epochs=e, batch_size=b)
# Save loss and performance measure values
test_loss, test_acc = network.evaluate(test_data, test_labels)
```

Deep Feedforward Networks in Python

- `train_data`: training examples (matrix of feature vectors $\boldsymbol{x}$ or $\boldsymbol{x}_{train}$)
- `train_labels`: training labels (y or y_{train})
- `test_data`: test examples used to calculate performance of network ($\boldsymbol{x}_{test}$)
- `test_labels`: test labels (y_{test})
- Optimizing algorithm options: `rmsprop`, `sgd`, `adagrad`, `adam`, etc.
- Loss function options: `mse`, `mae`, `categorical_crossentropy`, etc.
- Performance measure options: `accuracy`, `mae`, etc.