CIE 555
Neural Networks and Deep Learning

Deep Feedforward Networks II

1

Overview

- A clarification regarding cost functions (why cross entropy is better than square error)
- Hidden Units (Section 6.3)
  - ReLU and their generalizations
  - Logistic Sigmoid
  - Hyperbolic Tangent
- Architecture Design (Section 6.4)
  - Universal Approximation Properties and Depth
- Backpropagation

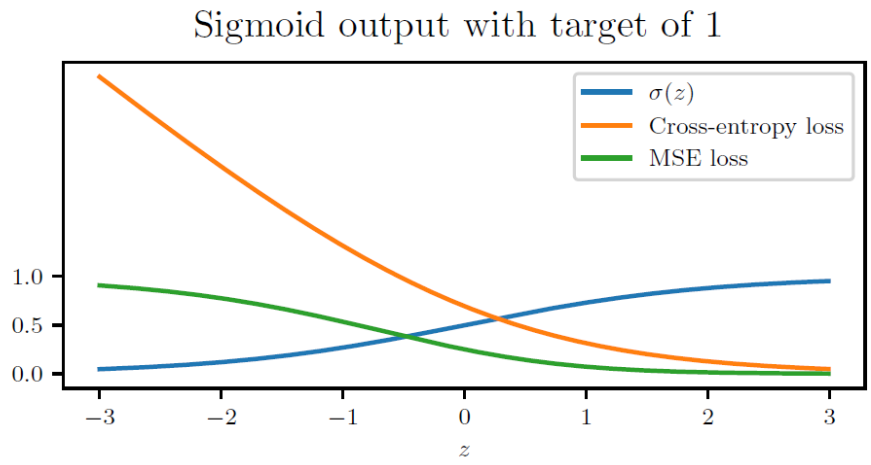
2

Cost functions

- Suppose the desired output of the output unit is 1 and the actual output is 0.000001 what is the square error in this case.
- Now, suppose that the actual output is (0.000001) (10 times of the previous output) what will be the square error in this case.
- There is almost no gradient to fix the error (slope is almost horizontal)
- Now consider the cross entropy
  - log (0.0000001) = -16.1181
  - Log(0.000001)= -13.8155 (considerable change)
- Using cross entropy results in a very steep gradient of the cost function

3

Cost functions



Goodfellow, Bengio, Courville 2016

4

Hidden Units

- The design of hidden units is an active area of research and does not yet have many definitive guiding theoretical principles.
- Most $z = W^{\top}x + b$ hidden units accept a vector of inputs *x*, compute an affine transformation, and then applying an element-wise nonlinear function *g*(*z*). They are distinguished from each other only by the choice of the form of the activation function *g*(*z*).
- It can be difficult to determine when to use which kind
- We will discuss some of the basic intuitions motivating each type of hidden units. These intuitions can help decide when to try out each of these units. However, it is difficult to predict in advance which will work best.
- The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Goodfellow, Bengio, Courville 2016

5

Rectified Linear Units (ReLU)

- Rectified linear units use the activation function $g(z) = \max\{0, z\}$.
- Rectified linear units are easy to optimize because they are so similar to linear units.
- The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain.
- This makes the derivatives through a rectified linear unit remain large whenever the unit is active.
- The gradients are not only large but also consistent.
- When initializing the parameters of the affine transformation $h = g(W^{\top}x + b)$, it can be a good practice **to set all elements of *b* to a small, positive value, such as 0.1**. This makes it very likely that the rectified linear **units will be initially active** for most inputs in the training set and allow the derivatives to pass through.

Goodfellow, Bengio, Courville 2016

6

Rectified Linear Units (ReLU)

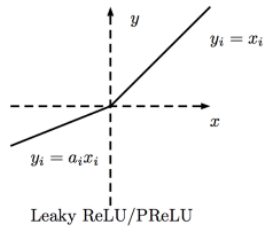
- The rectified linear function $g(z) = \max\{0, z\}$is not differentiable at $z = 0$.
- In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.
- This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly.
- Because we do not expect training to actually reach a point where the gradient is 0 , it is acceptable for the minima of the cost function to correspond to points with undefined gradient.
- Software implementations of neural network training usually return one of the one-sided derivatives (i.e 0 for the left-side gradient or 1 for the right-side gradient)rather than reporting that the derivative is undefined or raising an error.
- This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway.

Goodfellow, Bengio, Courville 2016

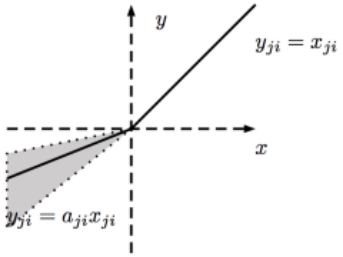
7

Generalizations of ReLU

- Generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0; h_i = g(z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$
  - **Leaky ReLU** fixes α_i to a small value like 0.01.
  - **Absolute value rectification** fixes $\alpha_i = -1$ to obtain $g(z) = |z|$.
  - **Parametric ReLU or PReLU** treats α_i as a learnable parameter.
  - **Randomized ReLU or RReLU:** sample $\alpha_i$from a fixed range (α_i is a random number



Leaky ReLU/PReLU



Randomized Leaky ReLU

Paper: Empirical Evaluation of Rectified Activations in Convolutional Network https://arxiv.org/abs/1505.00853

Goodfellow, Bengio, Courville 2016

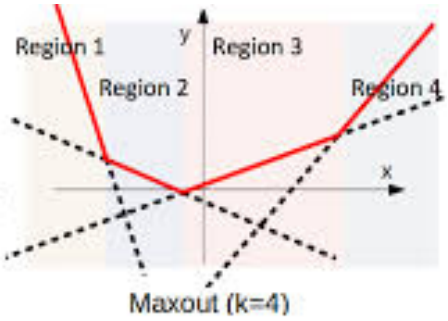
8

Maxout units (Goodfellow *et al.*, 2013a)

- Instead of applying an element-wise function $g(z)$, maxout units divide z into groups of k values.
- Each maxout unit then outputs the maximum element of one of these groups:

$$g(z)_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

- This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.



https://arxiv.org/pdf/1508.00330.pdf

Paper: Maxout Networks - https://arxiv.org/pdf/1302.4389.pdf

Goodfellow, Bengio, Courville 2016

9

Maxout units (cont.)

- A maxout unit can learn a piecewise linear, convex function with up to *k* pieces. Maxout units can thus be seen as *learning the activation function* itself rather than just the relationship between units.
- Each maxout unit is now parametrized by *k* weight vectors instead of just one, so maxout units typically need more regularization than rectified linear units. They can work well without regularization if the training set is large and the number of pieces per unit is kept low (Cai *et al.*, 2013).
- With large enough *k*, a maxout unit can learn to approximate any convex function with arbitrary fidelity.

Goodfellow, Bengio, Courville 2016

10

Logistic Sigmoid and Hyperbolic Tangent

- Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function

$$g(z) = \sigma(z)$$

- or the hyperbolic tangent activation function

$$g(z) = tanh(z)$$

- These activation functions are closely related

$$\tanh(z) = 2\sigma(2z) - 1$$

Goodfellow, Bengio, Courville 2016

11

Logistic Sigmoid and Hyperbolic Tangent: Saturation Challenges

- Unlike piecewise linear units, sigmoidal units saturate across most of their domain—they saturate to a high value when *z* is very positive, saturate to a low value when *z* is very negative, and are only strongly sensitive to their input when *z* is near 0.
- The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged.
- Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.

Goodfellow, Bengio, Courville 2016

12

Using Logistic Sigmoid and Hyperbolic Tangent

- When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$.
- Because tanh is similar to the identity function near 0, training a deep neural network $\hat{y} = w^\top \tanh(U^\top \tanh(V^\top x))$ resembles training a linear model $\hat{y} = w^\top U^\top V^\top x$ so long as the activations of the network can be kept small. This makes training the tanh network easier.
- Sigmoidal activation functions are more common in settings other than feedforward networks. Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation.

Goodfellow, Bengio, Courville 2016

13

Architecture Design

- The word **architecture** refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.
- In chain-based architectures, the main architectural considerations are to choose the depth of the network and the width of each layer.
- The **universal approximation theorem** states that a feedforward network **with a linear output layer and at least one hidden layer with any "squashing" activation function** (such s the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.
- Deeper networks often are able to use far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.
- The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.

Goodfellow, Bengio, Courville 2016

14

Universal Approximation Properties and Depth

- The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function.
- However, learning can fail for two different reasons:
  1. The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.
  2. the training algorithm might choose the wrong function due to overfitting.
- Barron (1993) provides some bounds on the size of a single-layer network needed to approximate a broad class of functions.
- Unfortunately, in the worse case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required.
- Consider the binary case: the number of possible binary functions on vectors $v \in \{0, 1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

Goodfellow, Bengio, Courville 2016

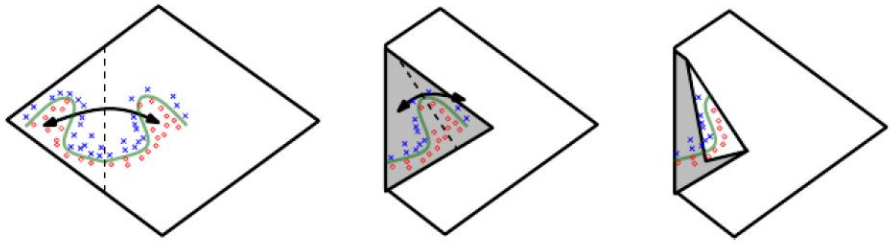
15

The advantage of using Deep networks

- A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.
- In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.
- In summary: why deeper?
  - Shallow net may need (exponentially) more width
  - Shallow net may overfit more

Goodfellow, Bengio, Courville 2016

16



Geometric explanation of the exponential advantage of deeper rectifier networks formally by Montufar *et al.* (2014).

- *(Left)*An absolute value rectification unit has the same output for every pair of mirror points in its input. The mirror axis of symmetry is given by the hyperplane defined by the weights and bias of the unit.
- (Center)The function can be obtained by folding the space around the axis of symmetry.
- (Right)Another repeating pattern can be folded on top of the first (by another downstream unit) to obtain another symmetry (which is now repeated four times, with two hidden layers).

Goodfellow, Bengio, Courville 2016

17

Other Architectural Considerations

- The depth of the network and the width of each layer are not the only considerations for choosing the architecture of a network.
- Many neural network architectures have been developed for specific tasks. For example. Convolutional networks are designed mainly for computer vision tasks and recurrent neural networks are used for for sequence processing.
- Sometimes, the layers need not be connected in a chain, even though this is the most common practice. Many architectures build a main chain but then add extr architectural features to it, such as skip connections going from layer *i* to layer *i* + 2 or higher (make it easier for the gradient to flow from output layers to layers nearer the input.).
- These specialized architectures will be discussed in upcoming chapters.

Goodfellow, Bengio, Courville 2016

18

Backpropagation: Forward Pass

For a neural network of D inputs, J hidden nodes and K outputs.

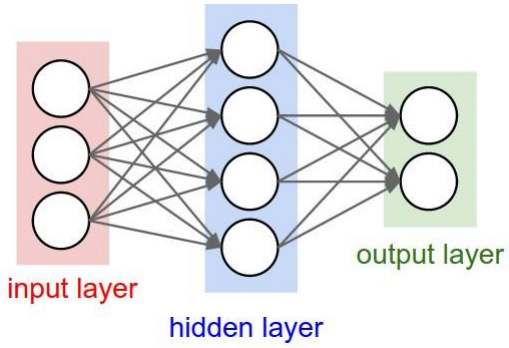
Output of a hidden node j:
$h_j = f(v_{j0} + \sum_{i=1}^{D} x_i \, v_{ji})$

x_i: input of index I
f: activation function of hidden nodes

Output of an output node k:

$$o_k = g\left(w_{k0} + \sum_{j=1}^{J} h_j \, w_{kj}\right)$$

g: activation function of output nodes



19

Backward Pass (Training Neural Networks)

- Goal: find weights to minimize a loss function

$$w^* = \underset{w}{\operatorname{argmin}} \sum_{n=1}^{N} E(o^{(n)}, t^{(n)})$$

- E: loss/error function
- $o^{(n)}$: output of the neural network for training data point n.
- $t^{(n)}$: target output of the training data point n

20

Steps

- Define a loss function. Examples:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$
- k *indexing output units*
- Algorithm:
- Loop until convergence
  - For each training data point n
    1. Initialize the network with a random weights
    2. Propagate forward to produce the output $o^{(n)}$ $(Forward\ pass)$.
    3. Propagate gradients backward $(Backward\ pass)$.
    4. Update each weight via gradient descent.

21

Useful Derivatives

Sigmoid	$\frac{1}{1+e^{-z}}$	$\sigma(z).(1-\sigma(z))$
Tanh	$\frac{e^z - e^{-z}}{e^z + e^{-z}}$	$\frac{1}{cosh^2(z)}$
ReLU	$\max(0, z)$	$\begin{cases} 1, if\ z > 0 \\ 0, if\ z \leq 0 \end{cases}$

22

Example

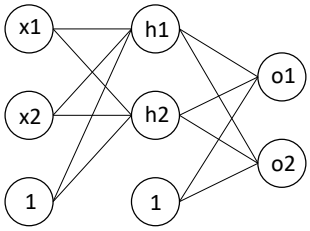
Given the network shown in Figure

For simplicity choose one training point
Training point (0.05,0.1)
Target output (0.99,0.01)

v_1 $(used\ to\ calculate\ h_1)$: [0.15 0.2 0.35]
v_2 $(used\ to\ calculate\ h_2)$: [0.25 0.3 0.35]

w_1 $(used\ to\ calculate\ o_1)$: [0.4 0.45 0.6]
w_2 $(used\ to\ calculate\ o_2)$: [0.5 0.55 0.6]

Use the Sigmoid function to activate outputs in both layers



23

Example

- **Forward Pass**

$h_1 = f(v_{10} + v_{11}x_1 + v_{12}x_2) = 0.5933$
$h_2 = f(v_{20} + v_{21}x_1 + v_{22}x_2) = 0.5969$

$o_1 = g(w_{10} + w_{11}h_1 + w_{12}h_2) = 0.7514$
$o_2 = g(w_{20} + w_{21}h_1 + w_{22}h_2) = 0.7729$

- **Calculate Error**

$E_{o1} = 0.5(o_1 - t_1)^2 = 0.2748$
$E_{o2} = 0.5(o_2 - t_2)^2 = 0.0236$
$E = E_{o1} + E_{o2} = 0.2984$

24

Example

Update weights using gradient descent

$$w^+ = w - \eta \frac{\partial E}{\partial w}$$

$$v^+ = v - \eta \frac{\partial E}{\partial v}$$

Backward pass

$$o_k = g\left(w_{k0} + \sum_{j=1}^{J} h_i \, w_{kj}\right) = g(z_k)$$

Apply the chain rule

$$For\ k = 1, j = 1 \qquad \frac{\partial E}{\partial w_{11}} = \frac{\partial E_{o1}}{\partial o_1} * \frac{\partial o_1}{\partial z_1} * \frac{\partial z_1}{\partial w_{11}}$$

25

Example

$$E_{o1} = 0.5(o_1 - t_1)^2$$

$$\frac{\partial E_{o1}}{\partial o_1} = (o_1 - t_1) = 0.7514 - 0.01 = 0.7414$$

$$o_1 = g(z_1), g\ is\ the\ activation\ function$$

In case of the sigmoid function

$$\frac{\partial o_1}{\partial z_1} = o_1(1 - o_1) = 0.1868$$

26

Example

$$z_k = w_{k0} + \sum_{j=1}^{J} h_i \, w_{kj}$$

$$\frac{\partial z_1}{\partial w_{11}} = h_1 = 0.5933$$

$$\frac{\partial E}{\partial w_{11}} = \frac{\partial E_{o1}}{\partial o_1} * \frac{\partial o_1}{\partial z_1} * \frac{\partial z_1}{\partial w_{11}} = 0.7414 * 0.1868 * 0.5933 = 0.0822$$

$$w_{11}^{+} = w_{11} - \eta \frac{\partial E}{\partial w_{11}} = 0.4 - 0.5 * 0.0822 = 0.3589 \; (\eta = 0.5)$$

27

Example

- Similarly:
  - $w_{12}^{+} = 0.4087$
  - $w_{21}^{+} = 0.5113$
  - $w_{22}^{+} = 0.5614$

- **Note:** we use the **original weights**, not the updated weights, when we continue the backpropagation algorithm.

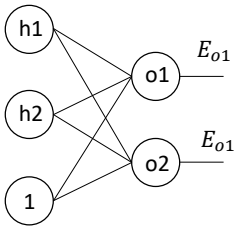
28

Example

$$h_j = f\left(v_{j0} + \sum_{i=1}^{D} x_i\, v_{ji}\right) = f(u_i)$$

$For\ j = 1, i = 1$ $\quad \frac{\partial E}{\partial v_{11}} = \frac{\partial E}{\partial h_1} * \frac{\partial h_1}{\partial u_1} * \frac{\partial u_1}{\partial v_{11}}$

$$\frac{\partial E}{\partial h_1} = \frac{\partial E_{o1}}{\partial h_1} + \frac{\partial E_{o2}}{\partial h_1}$$



29

Example

$$\frac{\partial E_{o1}}{\partial h_1} = \frac{\partial E_{o1}}{\partial o_1} * \frac{\partial o_1}{\partial z_1} * \frac{\partial z_1}{\partial h_1}$$

$$\frac{\partial z_1}{\partial h_1} = w_{11} = 0.4$$

$$\frac{\partial E_{o1}}{\partial h_1} = 0.7414 * 0.1868 * 0.4 = 0.0554$$

Similarly,

$$\frac{\partial E_{o2}}{\partial h_1} = \frac{\partial E_{o2}}{\partial o_2} * \frac{\partial o_2}{\partial z_2} * \frac{\partial z_2}{\partial h_1} = -0.0190$$

$$\frac{\partial E}{\partial h_1} = \frac{\partial E_{o1}}{\partial h_1} + \frac{\partial E_{o2}}{\partial h_1} = 0.0364$$

30

Example

$$h_1 = f(u_1), f \text{ is the activation function}$$

In case of the sigmoid function

$$\frac{\partial h_1}{\partial u_1} = h_1(1 - h_1) = 0.2413$$

$$\frac{\partial u_1}{\partial v_{11}} = x_1 = 0.05$$

$$\frac{\partial E}{\partial v_{11}} = \frac{\partial E}{\partial h_1} * \frac{\partial h_1}{\partial u_1} * \frac{\partial u_1}{\partial v_{11}} = 0.0004$$

$$v_{11}^{+} = v_{11} - \eta \frac{\partial E}{\partial v_{11}} = 0.1498$$

31

Example

- Similarly
  - $v_{12}^{+} = 0.1996$
  - $v_{21}^{+} = 0.2498$
  - $v_{22}^{+} = 0.2995$

- Key idea behind the backpropagation
  - We don't have target for hidden inputs.
  - Instead of using desired output for hidden layer, use error derivatives w.r.t hidden layer outputs (chain rule)

32

How often we update the weights

- Batch Gradient Descent: Use all training data to update your weights

$$w_{kj}^{+} \leftarrow w_{kj} - \eta \frac{\partial E}{\partial w_{ki}} = w_{kj} - \eta \sum_{n=1}^{N} \frac{\partial E(o^{(n)}, t^{(n)}; w)}{\partial w_{ki}}$$

- Stochastic Gradient Descent: update after each training case
- Mini-batch: a trade-off between stochastic gradient descent and batch gradient.

$$w_{kj}^{+} \leftarrow w_{kj} - \eta \frac{\partial E}{\partial w_{ki}} = w_{kj} - \eta \sum_{n=1}^{m} \frac{\partial E(o^{(n)}, t^{(n)}; w)}{\partial w_{ki}}$$

Where m is the batch size

One **epoch** is when an entire dataset passes the forward and backward only once.

Learning rates:

- Use a fixed learning rate
- Adapt a learning rate
- Add momentum

33