AAU
AAiT
SiTE

Course Title: **Deep Learning**

Credit Hour: 3

Instructor: Fantahun B. (PhD) ✉ meetfantaai@gmail.com

Office: NB #

Ch-1A Deep Neural Networks

[Based on your textbook]

Mar-2023, AA

Deep Neural Networks

Agenda

- Gradient Based Learning
- Output Units
- Hidden Units
- Architecture Design
- Backpropagation

Deep Neural Networks

Objectives

After completing this chapter students will be able to:

- Conceptualize Gradient Based Learning
- Define deep learning, appreciate the role of depth in DNNs
- Identify and differentiate Output Units
- Identify Hidden Units of deep neural networks
- Identify architectural concerns in neural network design
- Discuss Architecture Design considerations in deep neural networks
- Explain and implement the Backpropagation algorithm, appreciate its role in gradient based learning

Deep Neural Networks

- Feedforward deep neural networks, Feedforward networks, Multilayer Perceptrons (MLP) are the quintessential deep learning models.
- The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation.
- These models are called feedforward b/c information flows through the function being evaluated from x, through the intermediate computations used to define f , and finally to the output y.
- There are no feedback connections in which outputs of the model are fed back into itself.
- When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks (ch-10).

Deep Neural Networks

- Feedforward neural networks are called networks because they are typically represented by composing together many different functions.
- The model is associated with a directed acyclic graph describing how the functions are composed together.
- For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)} (f^{(2)} (f^{(1)}(x)))$.
- These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on.
- The overall length of the chain gives the *depth* of the model. It is from this depth terminology that the name "deep learning" arises.

Deep Neural Networks

- The final layer of a feedforward network is called the output layer. During neural network training, we drive $f(x)$ to match $f^*(x)$.
- The training data provides us with noisy, approximate examples of $f^*(x)$ evaluated at different training points.
- Each example x is accompanied by a label $y \approx f^*(x)$.
- The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y.
- The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^*.
- Because the training data does not show the desired output for each of these layers, these layers are called hidden layers.

Deep Neural Networks

- Finally, these networks are called neural because they are loosely inspired by neuroscience.
- Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value.
- The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions f(i)(x) used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute.
- However, modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain.
- It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

Deep Neural Networks: Gradient-Based Learning

- The largest difference between the linear models and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex.
- This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.
- Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems).
- Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- As with other machine learning models, to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model.

Deep Neural Networks: Gradient-Based Learning

Cost Functions

- An important aspect of the design of a deep neural network is the choice of the cost function.
- In most cases, our parametric model defines a distribution p(**y** | **x**;**θ**) and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function.
- Sometimes, we take a simpler approach, where rather than predicting a complete probability distribution over **y**, we merely predict some statistic of y conditioned on x. Specialized loss functions allow us to train a predictor of these estimates.
- The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term.

Deep Neural Networks: Gradient-Based Learning

Cost Functions

- An important aspect of the design of a deep neural network is the choice of the cost function.
- Learning Conditional Distributions with Maximum Likelihood
  - Most modern neural networks are trained using maximum likelihood → the cost function is simply the negative log-likelihood, equivalently described as cross-entropy between the training data and the model distribution.

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \qquad (6.12)$$

Deep Neural Networks: Output Units

- The choice of cost function is tightly coupled with the choice of output unit.
- Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.
- Any kind of neural network unit that may be used as an output can also be used as a hidden unit.
- Feedforward network provides a set of hidden features defined by $h = f(x; \theta)$.
- The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

Deep Neural Networks: Output Units

Linear Units for Gaussian Output Distributions

- One simple kind of output unit is an output unit based on an affine transformation with no nonlinearity.
- These are often just called linear units.
- Given features h, a layer of linear output units produces a vector $y_{hat} = W^T h + b$.
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution.
- Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

Deep Neural Networks: Output Units

Sigmoid Units for Bernoulli Output Distributions

- Many tasks require predicting the value of a binary variable y . Classification problems with two classes can be cast in this form.
- The maximum-likelihood approach is to define a Bernoulli distribution over y conditioned on x.
- A Bernoulli distribution is defined by just a single number.
- The neural net needs to predict only P (y = 1 | x). For this number to be a valid probability, it must lie in the interval [0, 1].
- Satisfying this constraint requires some careful design effort. Suppose we were to use a linear unit, and threshold its value to obtain a valid probability:

$$P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\}. \tag{6.18}$$

Deep Neural Networks: Output Units

Sigmoid Units for Bernoulli Output Distributions

- This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent.
- Any time that $w^T h + b$ strayed outside the unit interval, the gradient of the output of the model with respect to its parameters would be 0.
- A gradient of 0 is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Deep Neural Networks: Output Units

Sigmoid Units for Bernoulli Output Distributions

- Instead, it is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer.
- This approach is based on using sigmoid output units combined with maximum likelihood.
- A sigmoid output unit is defined by:

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right) \tag{6.19}$$

where σ is the logistic sigmoid function

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

Deep Neural Networks: Output Units

Sigmoid Units for Bernoulli Output Distributions

- We can think of the sigmoid output unit as having two components.
- First, it uses a linear layer to compute $z = w^T h + b$.
- Next, it uses the sigmoid activation function to convert z into a probability.

Deep Neural Networks: Output Units

Softmax Units for Multinoulli Output Distributions

- Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function.
- This can be seen as a generalization of the sigmoid function which was used to represent a probability distribution over a binary variable.
- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

Deep Neural Networks: Output Units

Softmax Units for Multinoulli Output Distributions

- The softmax function is given by:

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \qquad (6.29)$$

- As with the logistic sigmoid, the use of the exp function works very well when training the softmax to output a target value y using maximum log-likelihood.
- In this case, we wish to maximize logP(y = i; z) = log softmax(z)$_i$.
- Defining the softmax in terms of exp is natural because the log in the log-likelihood can undo the exp of the softmax:

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j). \qquad (6.30)$$

Deep Neural Networks: Output Units

Softmax Units for Multinoulli Output Distributions

- The first term of Eq. 6.30 shows that the input z_i always has a direct contribution to the cost function.
- Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of Eq. 6.30 becomes very small.
- When maximizing the log-likelihood, the first term encourages z_i to be pushed up, while the second term encourages all of z to be pushed down.

Deep Neural Networks: Hidden Units

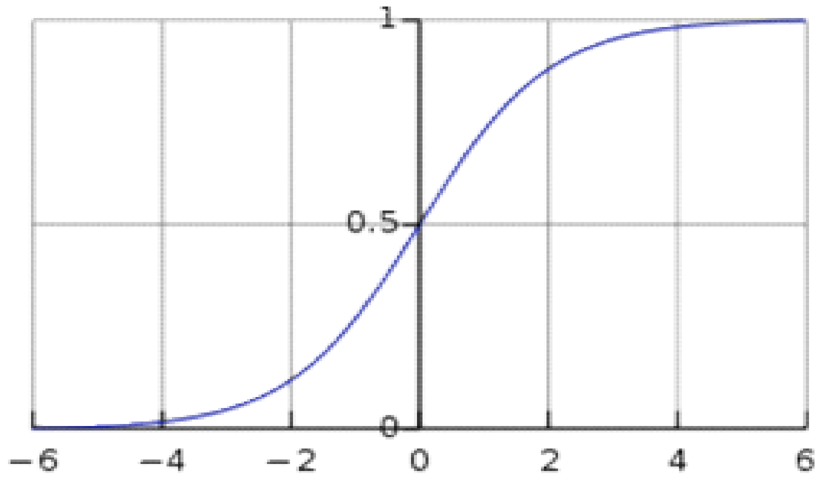
- The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.
- Many types of hidden units are available.
  - Logistic Sigmoid
  - Hyperbolic Tangent
  - Rectified Linear Units
  - etc
- It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice).

Deep Neural Networks: Hidden Units

Logistic Sigmoid

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

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} = 1 - S(-x).$$



Deep Neural Networks: Hidden Units

Logistic Sigmoid

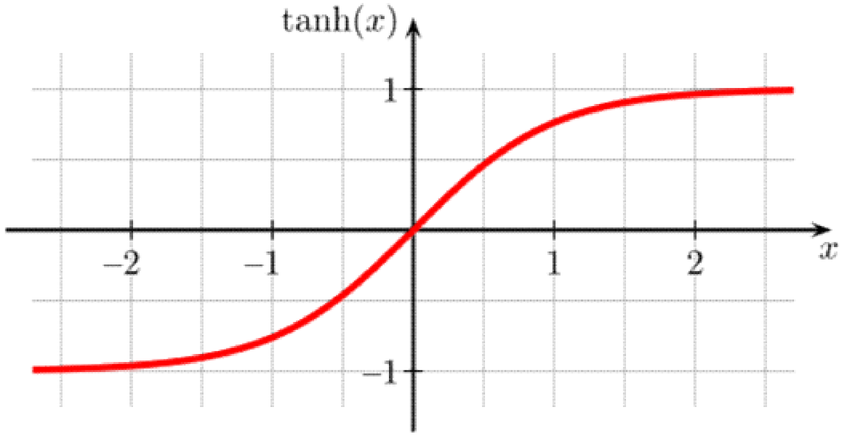
- Sigmoidal units saturate across most of their domain:
  - they saturate to a high value when z is very positive,
  - saturate to a low value when z is very negative,
  - and are only strongly sensitive to their input when z is near 0.
- The widespread saturation of sigmoidal units can make gradient-based learning very difficult.
- For this reason, their use as hidden units in feedforward networks is now discouraged.
- When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid.

Deep Neural Networks: Hidden Units

Hyperbolic Tangent

- tanh function is symmetric about the origin, where the inputs would be normalized

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

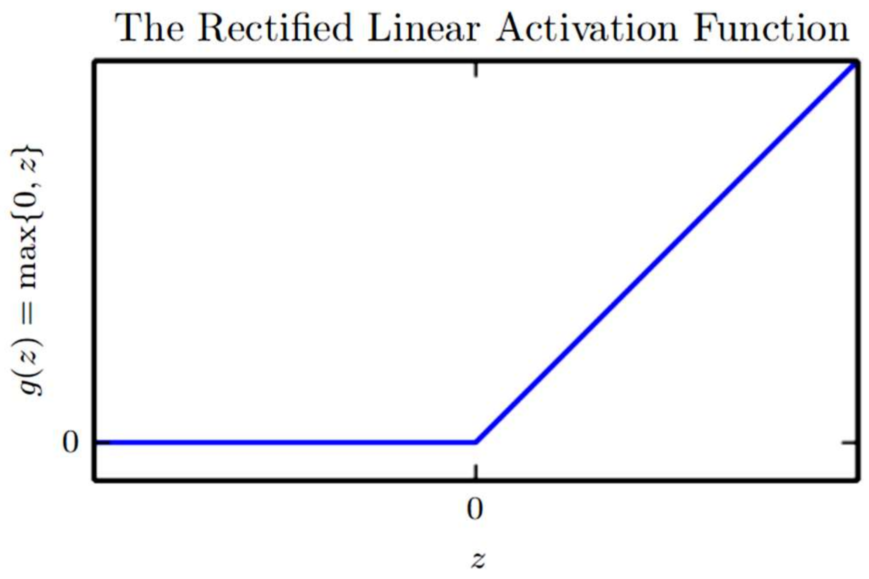


Deep Neural Networks: Hidden Units

Rectified Linear Units (ReLU) and their Generalizations

- Rectified linear units use the activation function:

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



Deep Neural Networks: Hidden Units

ReLU Benefitss

- Computation simplicity
  - Computations are also cheaper: there is no need for computing the exponential function in activations [Deep Sparse Rectifier Neural Networks, 2011]
- Representational Sparsity
  - capable of outputting a true zero value.
  - tanh and sigmoid activation functions learn to approximate a zero output, e.g. a value very close to zero, but not a true zero value.
- Linear behavior
  - Rectified linear units […] are based on the principle that models are easier to optimize if their behavior is closer to linear.
  - Key to this property is that networks trained with this activation function almost completely avoid the problem of vanishing gradients, as the gradients remain proportional to the node activations.
- Train deep Networks

Deep Neural Networks: Hidden Units

ReLU and their Generalizations

- One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero.
- A variety of generalizations of rectified linear units guarantee that they receive gradient everywhere.
- Three generalizations of rectified linear units are based on using a non-zero slope a_i when

$$z_i < 0: \quad h_i = g(z,a)_i = \max(0, z_i) + a_i \min(0, z_i).$$

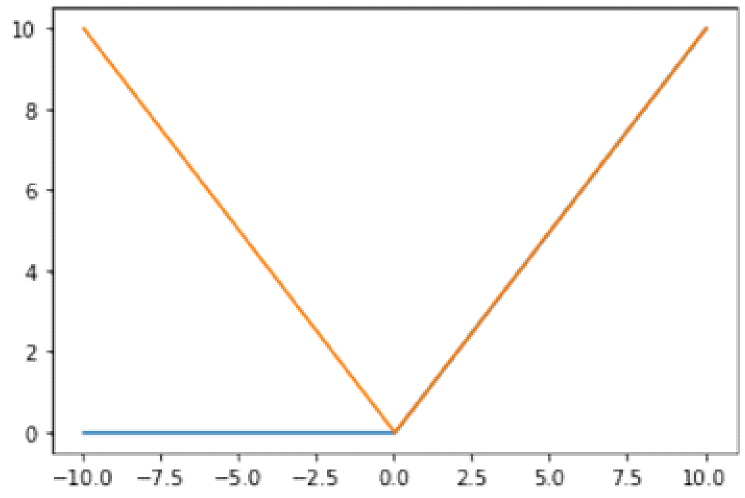
  - Absolute value rectification
  - leaky ReLU
  - Parametric ReLU or PReLU

Deep Neural Networks: Hidden Units

ReLUs and their Generalizations: Absolute vale rectification

- Absolute value rectification fixes $a_i = -1$ to obtain g(z) = |z|.

$$h_i = g(z,a)_i = \max(0, z_i) + a_i \min(0, z_i) = \max(0, z_i) + (-1)^* \min(0, z_i)$$



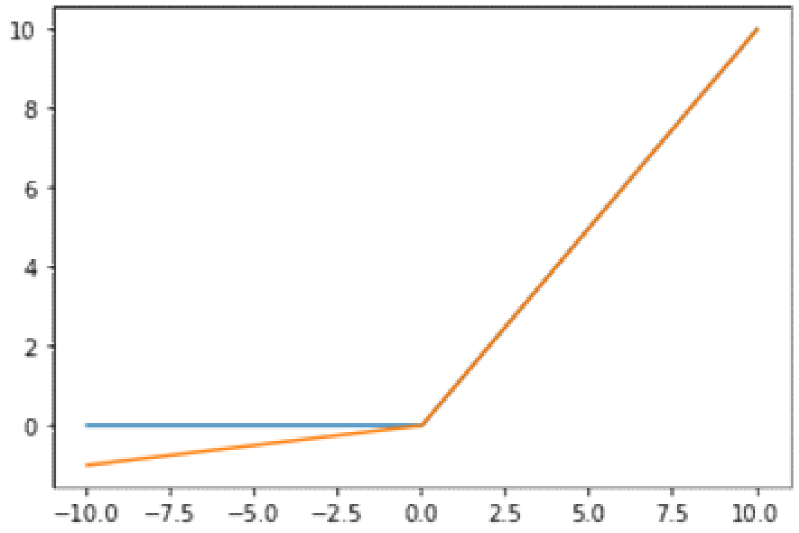
- It is used for object recognition from images (Jarrett et al., 2009), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

Deep Neural Networks: Hidden Units

ReLUs and their Generalizations: Leaky ReLU

- A leaky ReLU (Maas et al.,2013) fixes a_i to a small value like 0.01

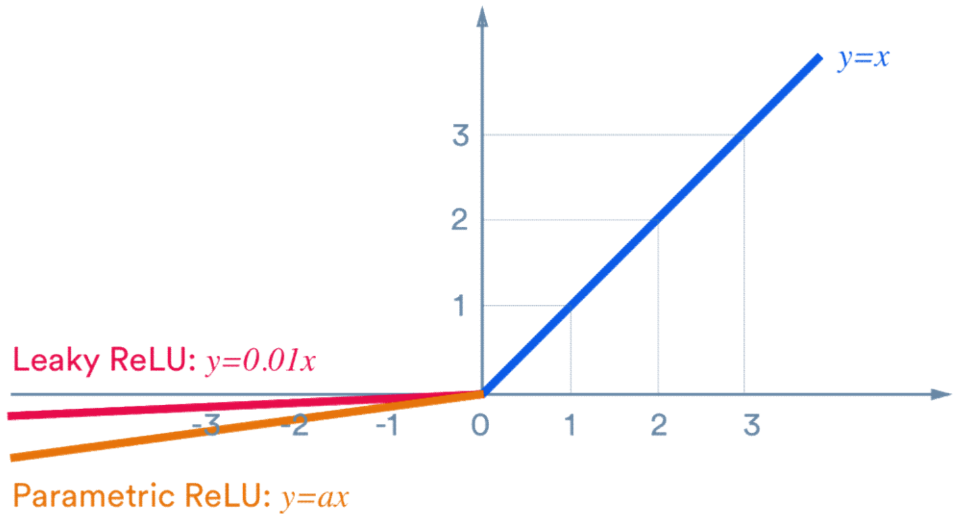
$$h_i = g(z,a)_i = \max(0, z_i) + a_i \min(0, z_i)$$
$$= \max(0, z_i) + 0.01 * \min(0, z_i)$$



Deep Neural Networks: Hidden Units

ReLUs and their Generalizations: Parametric ReLU or PReLU

- Parametric ReLU or PReLU treats a_i as a learnable parameter (He et al., 2015)
- PReLU is a type of leaky ReLU that, instead of having a predetermined slope like 0.01, makes it a parameter for the neural network to figure out itself.



Deep Neural Networks: Hidden Units

ReLUs and Their Generalizations: Maxout Units

- Maxout units (Goodfellow et al., 2013a) generalize rectified linear units further.
- 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:

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j \qquad (6.37)$$

where G(i) is the indices of the inputs for group i, {(i − 1)k + 1, . . . , ik}. This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.

Deep Neural Networks: Hidden Units

ReLUs and Their Generalizations: Maxout Units

- With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity.
- In particular, a maxout layer with two pieces can learn to implement the same function of the input x as a traditional layer using the rectified linear activation function, absolute value rectification function, or the leaky or parametric ReLU, or can learn to implement a totally different function altogether.
- 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).

Deep Neural Networks: Hidden Units

ReLUs and Their Generalizations: Maxout Units

- Because each unit is driven by multiple filters, maxout units have some redundancy that helps them to resist a phenomenon called catastrophic forgetting in which neural networks forget how to perform tasks that they were trained on in the past (Goodfellow et al., 2014a).
- Rectified linear units and all of these generalizations of them are based on the principle that models are easier to optimize if their behavior is closer to linear.
- This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved.

Deep Neural Networks

Reading Assignment

- Section 6.3.3 Other Hidden Units

Deep Neural Networks: Architecture Design

- Another key design consideration for neural networks is determining the **architecture**.
- 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.

Deep Neural Networks: Architecture Design

- Most neural networks are organized into groups of units called layers.
- Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by:

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right), \tag{6.40}$$

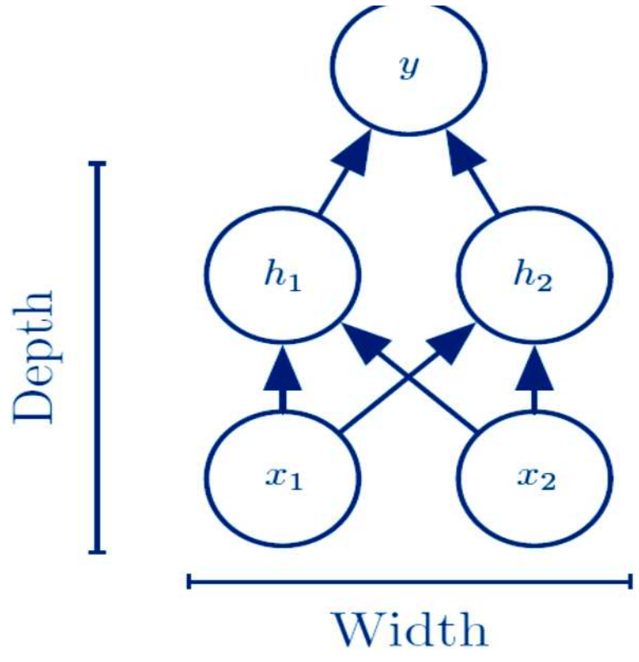
the second layer is given by

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right), \tag{6.41}$$

and so on.

Deep Neural Networks: Architecture Design

- In these chain-based architectures, the main architectural considerations are to choose the
  - depth of the network and
  - width of each layer.
- A network with even one hidden layer is sufficient to fit the training set.
- Deeper networks often
  - use far fewer units per layer
  - far fewer parameters
  - generalize to the test set,
  - but are also harder to optimize.
- The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.



Deep Neural Networks: Architecture Design

Universal Approximation Properties and Depth

- A linear model, can by definition represent only linear functions. It is easy to train because many loss functions result in convex optimization problems when applied to linear models.
- However, we often want to learn nonlinear functions. Does it require designing a specialized model family for the kind of nonlinearity? Fortunately, feedforward networks with hidden layers provide a universal approximation framework.
  - The universal approximation theorem (Hornik et al., 1989; Cybenko, 1989) states that *a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as sigmoid) 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*.
  - The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik et al., 1990).

Deep Neural Networks: Architecture Design
Universal Approximation Properties and Depth

- The concept of Borel measurability is beyond the scope of this course; For our purposes it suffices to say that:
  - any continuous function on a closed and bounded subset of Rn is Borel measurable and therefore may be approximated by a neural network.
- A neural network may also approximate any function mapping from any finite dimensional discrete space to another.
- While the original theorems were first stated in terms of units with activation functions that saturate both for very negative and for very positive arguments, universal approximation theorems have also been proven for a wider class of activation functions, which includes the now commonly used rectified linear unit (Leshno et al., 1993).

Deep Neural Networks: Architecture Design

Universal Approximater Theorem

- One hidden layer is enough to represent (not learn) an approximation of any function to an arbitrary degree of accuracy.
- We are not guaranteed that the training algorithm will be able to learn that function. Learning can fail for two different reasons:
    - The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.
    - The training algorithm might choose the wrong function due to overfitting.
- So why deeper?
    - Shallow net may need (exponentially) more width
    - Shallow net may overfit more

Deep Neural Networks: Architecture Design

Universal Approximater Theorem

- (Sec. 5.2.1) the “no free lunch” theorem shows that there is no universally superior machine learning algorithm.
- Feedforward networks provide a universal system for representing functions, in the sense that, given a function, there exists a feedforward network that approximates the function.
- There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.
- Experimentation should reveal this.

Deep Neural Networks: Architecture Design
Universal Approximation Properties and Depth

- The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be.
- 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.
- This is easiest to see in 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.

Deep Neural Networks: Architecture Design

Universal Approximation Properties and Depth

- In summary,
  - 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
    - reduce the amount of generalization error.

Deep Neural Networks: Architecture Design
Universal Approximation Properties and Depth

- Empirically, greater depth does seem to result in better generalization for a wide variety of tasks.

  (Bengio et al., 2007; Erhan et al., 2009; Bengio, 2009; Mesnil et al., 2011; Ciresan et al., 2012; Krizhevsky et al., 2012; Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013; Kahou et al., 2013; Goodfellow et al., 2014d; Szegedy et al., 2014a).
- See Fig. 6.6 and Fig. 6.7 for examples of some of these empirical results.
- This suggests that using deep architectures does indeed express a useful prior over the space of functions the model learns.

Deep Neural Networks: Architecture Design

Universal Approximation Properties and Depth

- Better Generalization with Greater Depth

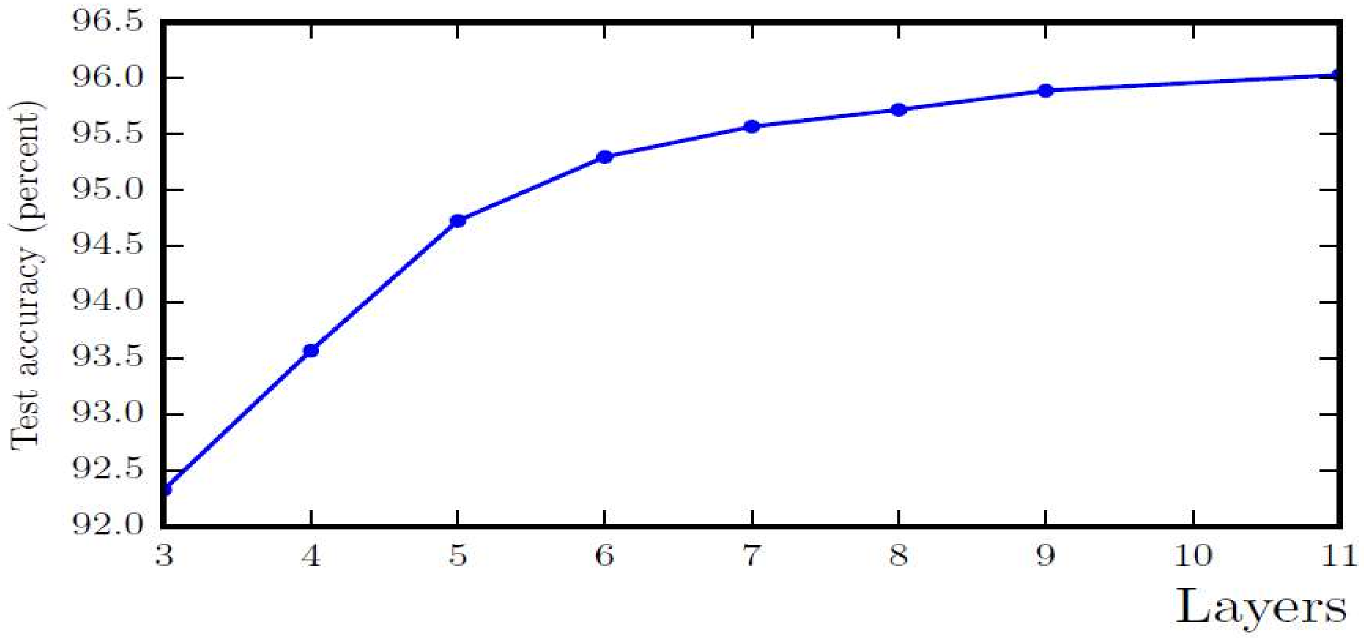


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow et al. (2014d). The test set accuracy consistently increases with increasing depth.

Deep Neural Networks: Architecture Design

Universal Approximation Properties and Depth

- Large, Shallow Models Overfit More

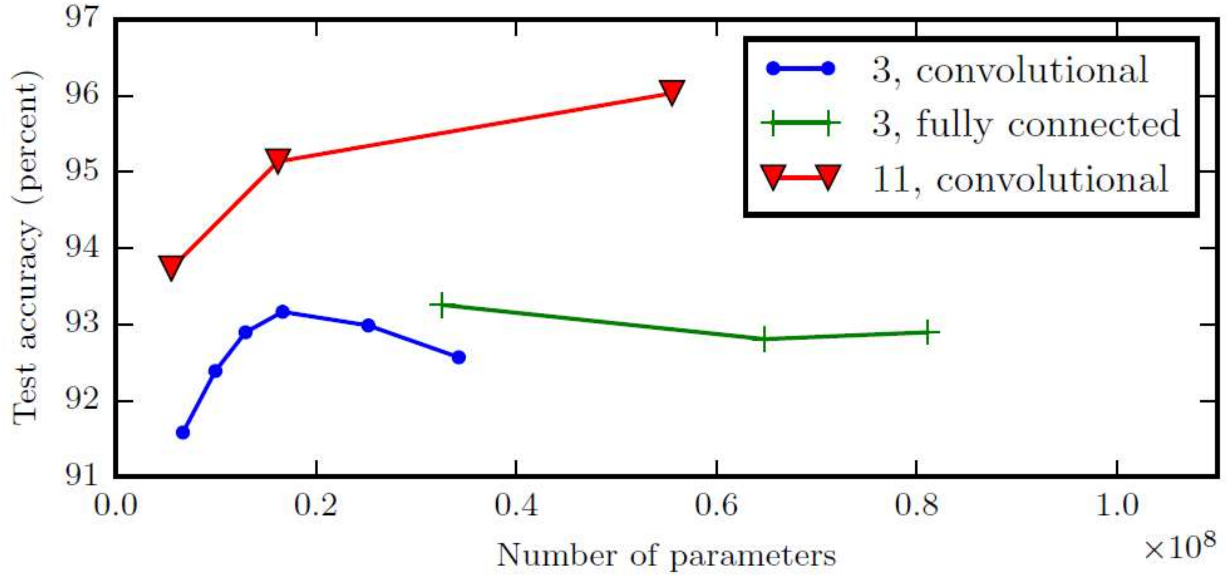


Figure 6.7: Deeper models tend to perform better. This experiment from Goodfellow et al. (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million.

Deep Neural Networks: Architecture Design
Other Architectural Considerations

- So far we have described neural networks as being simple chains of layers, with the main considerations being the depth of the network and the width of each layer. In practice, neural networks show considerably more diversity.
- In general, 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 extra architectural features to it, such as skip connections going from layer i to layer i + 2 or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.
- Many neural network architectures have been developed for specific tasks. Specialized architectures for computer vision, for example, called convolutional networks (Chapter 9).
- Feedforward networks may also be generalized to the recurrent neural networks for sequence processing (Chapter 10), which have their own architectural considerations.

Deep Neural Networks: Architecture Design
Other Architectural Considerations

- Another key consideration of architecture design is exactly how to connect a pair of layers to each other. (eg. dense, sparse)
- In the default neural network layer described by a linear transformation via a matrix **W** , every input unit is connected to every output unit.
- Many specialized networks have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer.
- These strategies for reducing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network, but are often highly problem-dependent.
- For example, CNNs, use specialized patterns of sparse connections that are very effective for computer vision problems.

Deep Neural Networks

Continue with
Part-II (Ch-1B) slide.