Deep Feedfoward Networks

Deep Feedforward Networks

- Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models.
- A feedforward network defines a mapping $\mathrm{y} = f(\mathrm{x}; \theta)$ and learns the value of the parameters θ that result in the best function approximation.
- These networks are called neural as they are loosely inspired by neuroscience.
- Goal of a deep feedforward network is to approximate some function F, e.g., for a classifier, $\mathrm{y} = F(\mathrm{x})$ maps an input x to a category y.
- These models are called feedforward as information flows through function being evaluated from x, through 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.

Deep Feedforward Networks

- When the feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks.
- They form the basis of many important commercial applications. For example, the convolutional neural networks used for object recognition from photos are a specialized kind of feedforward network.
- They are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.
- They are called networks as they are 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 $f^{(1)}, f^{(2)}, f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}((f^{(2)}(f^{(1)}(x)))$.

Deep Feedforward Networks

- These chain structures are the most commonly used structures of neural networks. Here, $f^{(1)}$ is called the first layer of network, $f^{(2)}$ is called second layer, and so on.
- The overall length of the chain gives the depth of the model. It is from this terminology that the name "**deep learning**" arises.
- The final layer of a feedforward network is called output layer.
- During 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.

Deep Feedforward Networks

- The behavior of 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.
- The learning algorithm must decide how to use these layers to best implement an approximation of F(x).
- Since the training data does not show the desired output for each of these layers, these layers are called **hidden layers**.
- Each hidden layer is typically vector-valued and dimensionality of these hidden layers determines the width of the model.
- Each element of vector may be interpreted as playing a role analogous to a neuron.

Deep Feedforward Networks

- Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many units that act in parallel, each representing a vector-to-scalar function.
- 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 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 their goal is not to perfectly model the brain.

Deep Feedforward Networks

- It is best to think of feedforward networks as function approximation machines/algorithms 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.
- One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations.
- Linear models, such as logistic regression and linear regression, are appealing because they may be fit efficiently and reliably, either in closed form or with convex optimization.
- Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

Deep Feedforward Networks

- To extend linear models to represent nonlinear functions of x, we can apply the linear model not to x itself but to a transformed input ϕ(x), where ϕ is a nonlinear transformation.
- We can think of ϕ as providing a set of features describing x, or as providing a new representation for x.
- The question is then how to choose the mapping ϕ.

1. One option is to use a very generic ϕ, such as infinite-dim ϕ that is implicitly used by kernel machines based on RBF kernel.

   If ϕ(x) is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor.

   Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.

Deep Feedforward Networks

2. Another option is to manually engineer ϕ. Until the advent of deep learning, this was the dominant approach.

   This approach requires decades of human effort for each separate task, with practitioners specializing in different domains such as speech recognition or computer vision, and with little transfer between domains.

3. The strategy of deep learning is to learn ϕ. In this, we have a model $y = f(x; \theta, w) = \phi(x; \theta)^T w$.

   We now have parameters θ that we use to learn ϕ from a broad class of functions, and parameters w that map from $\phi(x)$ to the desired output. This is an example of a deep feedforward network, with ϕ defining a hidden layer.

   This approach is the only one of the three that gives up on the convexity of the training problem, but the benefits outweigh the harms.

   In this, we parametrize the representation as $\phi(x; \theta)$ and use the optimization algorithm to find θ that corresponds to a good representation.

Deep Feedforward Networks

If we wish, this approach can capture the benefit of the first approach by being highly generic—we do so by using a very broad family $\phi(x; \theta)$.

This approach can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(x; \theta)$ that they expect will perform well.

The advantage is that the human designer only needs to find the right general function family rather than finding precisely the right function.

- This general principle of improving models by learning features extends beyond the feedforward networks.
- It is a recurring theme of deep learning that applies to all of the kinds of models.
- Feedforward networks are the application of this principle to learning deterministic mappings from x to y that lack feedback connections.

Deep Feedforward Networks

- We begin with a simple example of a feedforward network: **learning XOR** .
- The XOR function provides the target function $y = F(x)$ that we want to learn.
- Our model provides a function y = f(x; θ) and our learning algorithm will adapt the parameters θ to make f as similar as possible to F.
- We can treat this problem as a regression problem and use a mean squared error loss function.
- The MSE (SSE) is $L(\theta) = \frac{1}{4}\sum_{x \in X}(F(x) - f(x;\theta))^2$
- Now we must choose the form of our model, f(x; θ). Let a linear model, with θ consisting of w and b.
- Our model is defined to be $f(x;\theta) = x^T w + b$
- We can minimize L(θ) in closed form w.r.t. w and b using the normal equations that give w = 0 and b = 0.5.

Deep Feedforward Networks

- The linear model simply outputs 0.5 everywhere.
- We choose this loss function to simplify the math for this example as much as possible.
- (0, 0) -> 0; (1, 1)->0; (0, 1)-> 1; (1,0)->1
- The MSE (SSE) is $L(\theta) = \frac{1}{4}\sum_{x \in X}(F(x) - f(x;\theta))^2$
- Model f(x) = $w^T x+b = w_1x_1+w_2x_2+b$
- It gives $w_1+w_2+2b = 1$, $2w_1+w_2+2b = 1$, $w_1+2w_2+2b = 1$
- Subtracting 1st from 2nd gives $w_1 = 0$; 2nd from 3rd gives $w_1=w_2$. Thus, $w_1 = w_2 = 0$, $b = 0.5$

Deep Feedforward Networks

- Training a feedforward network requires making many of the same design decisions as are necessary for a linear model that include choosing:

  Optimizer,

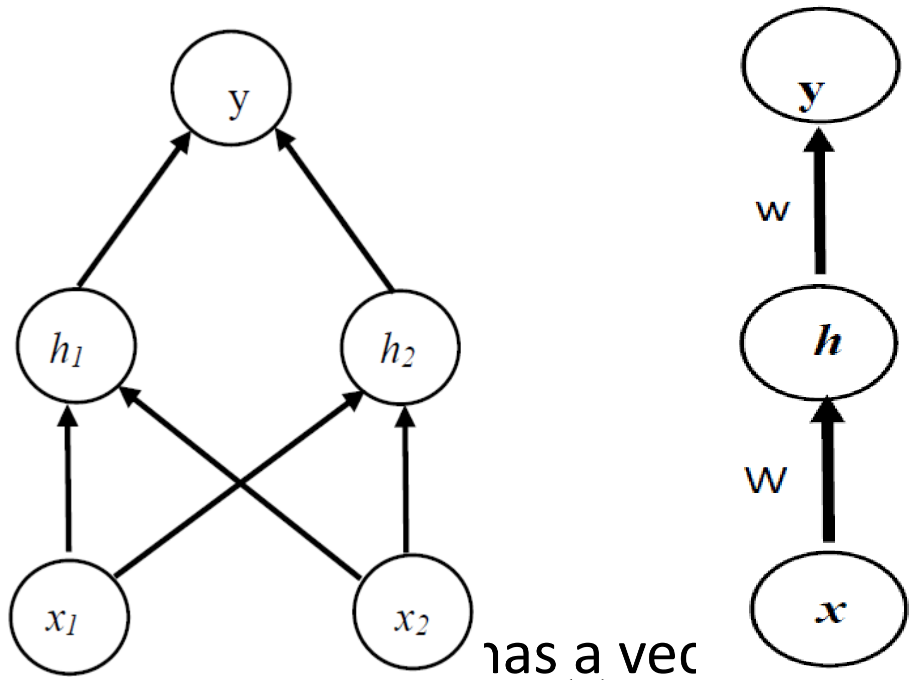
  Cost function,

  Form of output units.

- Feedforward networks have introduced the concept of a hidden layer that requires us to choose the activation functions to compute the hidden layer values.
- We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer.
- Learning in deep neural networks requires computing the gradients of the complicated functions.

Deep Feedforward Networks

- We introduce a very simple feedforward network with one hidden layer containing two hidden units.



- This fee has a vec dden units h that are computed by a function $f^{(1)}(\mathrm{x}; W, \mathrm{c})$.
- The values of these hidden units are then used as input for a second layer, which is the output layer of network.
- The output layer is still just a linear regression model, but now it is applied to h rather than to x.

Deep Feedforward Networks

- The network now contains two functions chained together: h = $f^{(1)}(\mathrm{x}; W, \mathrm{c})$ and y = $f^{(2)}(\mathrm{h}; w, \mathrm{b})$, with the complete model : $f(\mathrm{x}; \mathrm{W}, \mathrm{c}, \mathrm{w}, \mathrm{b}) = f^{(2)}\left(f^{(1)}(\mathrm{x}; W, \mathrm{c}); w, \mathrm{b}\right) = f^{(2)}\left(f^{(1)}(\mathrm{x})\right)$
- What function should $\boldsymbol{f}^{(\mathbf{1})}$compute?
- Linear models have served us well so far, and it may be tempting to make $\boldsymbol{f}^{(\mathbf{1})}$ be linear as well.
- Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input.
- Ignoring the intercept terms for the moment, suppose $f^{(1)}(\mathrm{x}) = W^T\mathrm{x}$ and $f^{(2)}(\mathrm{h}) = h^T\mathrm{w}$. Then $f(\mathrm{x}) = w^T W^T \mathrm{x}$.
- We can represent this function as f(x) = $\mathrm{x}^T\mathrm{w}'$, where w' = Ww.

Deep Feedforward Networks

- Clearly, we must use a **nonlinear function** to describe the features.
- Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function, called as **activation function**.
- We use that strategy here, by defining $h = g(W^T x + c)$, where W refers to weights of a linear transformation and c the biases.
- Previously, to describe a linear regression model we used a vector of weights and a scalar bias parameter to describe an affine transformation from an input vector to an output scalar.
- Now, we describe an affine transformation from a vector x to a vector h, so an entire vector of bias parameters is needed.

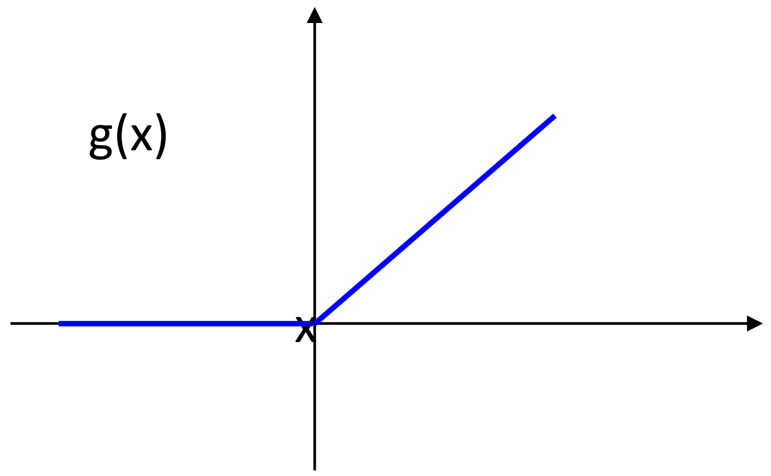
Deep Feedforward Networks

- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(x^T W_{:,i} + c)$
- In modern neural networks, the default recommendation is to use the rectified linear unit or ***ReLU***, defined by the activation function g(z) = max{0, z}.
- We can now specify our complete network as

$$f(X; W, c, w, b) = w^T \max\{0, W^T x + c\} + b$$

- Applying this function to output of a linear transformation yields a nonlinear transformation.

g(x) = max{0, x}



Deep Feedforward Networks

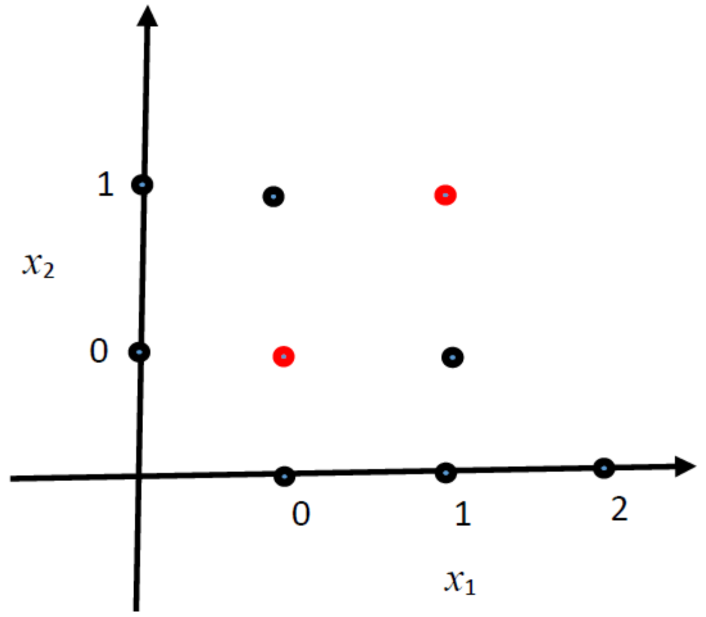
We have $X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$;

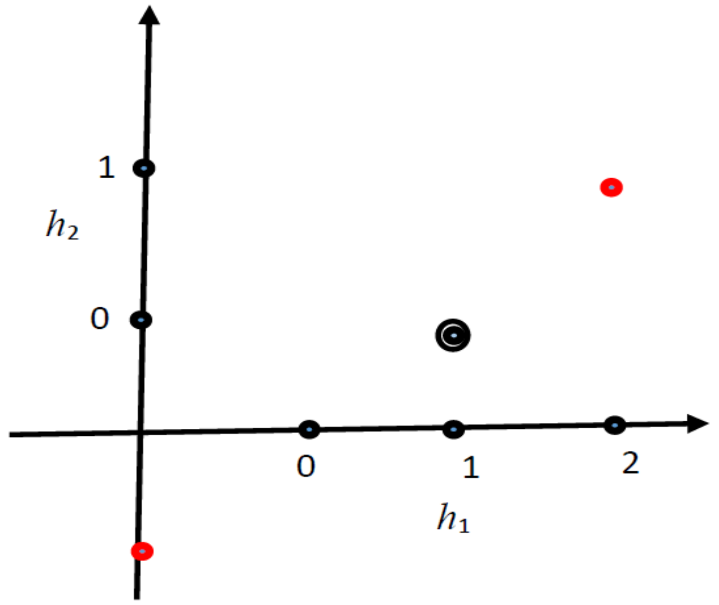
Consider $W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$; $c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$;

$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$; $b = 0$;

We have $XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$;

Adding bias c; $XW+c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$

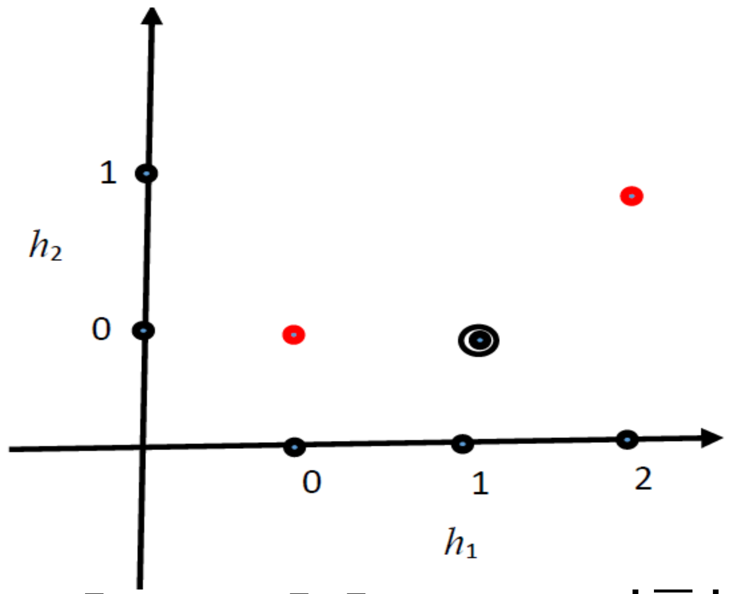




Deep Feedforward Networks

- Applying rectified linear (ReLU) transformation to it:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



- We finish by multiplying

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The neural network has obtained the correct answer for every example in the batch.

Deep Feedforward Networks

- A single sigmoid neuron cannot deal with non-linear data.
- If we connect multiple sigmoid neurons (neuron having sigmoid as activation function) in an effective way, by the combination of neurons we can approximate any complex relationship between the input and output, required to deal with non-linear data.
- The hidden layers are used to handle the complex non-linearly separable relations between the input and output.
- Generally in machine or deep learning, the principle of gradient-based learning is used.

Gradient-based Learning

Gradient Based Learning

- Designing and training a neural network is not much different from training any other machine learning model with gradient descent.
- This means that NNs are 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 convex optimization algorithms with global convergence used to train logistic regression or SVMs.
- The convergence point of gradient descent depends on the initial values of parameters.
- In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.

Deep Feedforward Networks

- Main difference between linear models and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex.
- Convex optimization converges starting from any initial parameters.
- Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and it is sensitive to the values of initial parameters.
- For feedforward neural networks, it is important to initialize all weights to small random values.
- The biases may be initialized to zero or to small positive values.
- To apply gradient-based learning, we must choose a **cost function**, and we must choose how to represent the **output** of the model.

Deep Feedforward Networks

- Of all the many optimization problems involved in deep learning, the most difficult is neural network training.
- It is quite common to invest days to months of time on hundreds of machines in order to solve even a single instance of the neural network training problem.
- Since this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it.
- We reduce a different cost function J(θ) in the hope that doing so will improve P, P is some performance measure
- The cost function can be written as an average over the training set, such as

$$J(\theta) = E_{\mathbf{X},\mathbf{Y}\sim\hat{p}_{data}} L(f(x;\ \theta), y) \qquad \text{(J1)}$$

where L is per-example loss function,
f(x; θ) is predicted output for the input x,
$\hat{p}_{data}$ is empirical distribution; y is target output.

Deep Feedforward Networks

- Eqn.(J1) defines an objective function w.r.t. training set.
- We usually prefer to minimize corresponding objective function where the expectation is taken across the data generating distribution p_{data} rather than just over the finite training set:

$$J^*(\theta) = E_{\mathbf{X},\mathbf{Y}\sim p_{data}} L(f(x;\ \theta), y) \qquad \text{(J2)}$$

- When both x and y are discrete. the generalization error (eqn. J2) can be written as a sum

$$J^*(\theta) = \sum_x \sum_y p_{data}(x, y)\, L(f(x;\ \theta), y) \qquad \text{(J2)}$$

with the exact gradient

$$g = \nabla_\theta J^*(\theta) = \sum_x \sum_y p_{data}(x, y)\, \nabla_\theta L(f(x;\ \theta), y)$$

- The goal of a machine learning algorithm is to reduce the expected generalization error given by eqn. (J2), which is known as the **risk**. The expectation is taken over the true underlying distribution p_{data}.

Deep Feedforward Networks

- If we knew true distribution $p_{data}(x, y)$, then the risk minimization would be an optimization task solvable by an optimization algorithm.
- When we do not know $p_{data}(x, y)$ but only have a training set of samples, we have a machine learning problem.
- The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set, i.e., replacing true distribution p(x, y) with empirical distribution $\hat{p}_{data}$(x, y) defined by training set.
- We now minimize empirical risk (for m of training examples)

$$E_{\mathbf{X},\mathbf{Y}\sim\hat{p}_{data}}[L(f(x;\ \theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f\left(x^{(i)};\ \theta\right),y^{(i)}) \quad \text{(J3)}$$

- Using true distribution p(x, y) of data, risk is estimated as

$$E_{\mathbf{X},\mathbf{Y}\sim p_{data}}[L(f(x;\ \theta),y)] = \sum_{i=1}^{m} L(f\left(x^{(i)};\ \theta\right),y^{(i)})\mathrm{p}(x^{(i)}, y^{(i)})$$

Deep Feedforward Networks

- Optimization algorithms for machine learning typically compute each update to the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function.
- For example, maximum likelihood estimation problems, when viewed in log space, decompose into a sum over each example:

$$\theta_{ML} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log p_{model}(x^{(i)}, y^{(i)}; \theta)$$

- Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set:

$$J(\theta) = E_{\boldsymbol{X},\boldsymbol{Y}\sim\hat{p}_{data}}\, log\; p_{model}(f(x;\ \theta), y)$$

Deep Feedforward Networks

- Most of the properties of objective function J used by most of our optimization algorithms are also expectations over the training set. For example, the most commonly used property is the gradient:

$$\nabla_\theta J(\theta) = E_{\mathbf{X},\mathbf{Y}\sim\hat{p}_{data}} \nabla_\theta log\ p_{model}(f(x;\ \theta), y)$$

- Computing this expectation exactly is very costly as it needs evaluating model on each input in entire dataset.
- Optimization algorithms that use the entire training set are called batch or deterministic gradient methods, because they process all of the training examples simultaneously in a large batch.
- Typically the term "batch gradient descent" implies the use of the full training set, while the use of the term "batch" is used to describe a group of examples that does not contain the full training set.

Deep Feedforward Networks

- Optimization algorithms that use only a single example at a time are sometimes called stochastic or sometimes online methods.
- The term online is usually reserved for the case where the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.
- Most algorithms used for deep learning fall somewhere in between, using more than one, but less than all of the training examples.
- These were traditionally called minibatch or minibatch stochastic methods and it is now common to simply call them stochastic methods.

Deep Feedforward Networks

- **Minibatch** sizes are generally driven by the following factors:
  - Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
  - Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
  - If all examples in batch are to be processed in parallel, then the amount of memory scales with batch size. For many hardware setups it is the limiting factor in batch size.
  - Some kinds of hardware achieve better runtime with specific sizes of arrays. It is common for power of 2 batch sizes to offer better runtime in GPU. Typical power of 2 are 32-256.
- Small batches can offer a regularizing effect perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1.

Deep Feedforward Networks

- The most important property of SGD and related minibatch based optimization is that computation time per update does not grow with the number of training examples.
- This allows convergence even when the number of training examples becomes very large.
- For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set.
- While SGD remains a very popular optimization strategy, learning with it can sometimes be slow.
- The method of momentum helps to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.
- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Deep Feedforward Networks

Stochastic Gradient Descent

- Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular.
- It is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data generating distribution.

SDG Algorithm

```
input: initial parameter θ, learning rate $\epsilon_0$
while stopping criteria is not met do
    $\epsilon_k = \left(1 - \frac{k}{\tau}\right) \epsilon_0 + \frac{k}{\tau} \epsilon_\tau$    //at kth iteration after τ iteration , LR is fixed
    Sample a minibatch of m examples {$x^{(1)}, x^{(2)}, \ldots, x^{(m)}$}
      from training set with corresponding targets $y^{(i)}$
    Compute gradient estimate:
        $\hat{g} = \hat{g} + \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x^{(i)};\ \theta), y^{(i)})$
    Update $\theta$ as:  $\theta = \theta - \epsilon_k\ \hat{g}$
End //while
```

Deep Feedforward Networks

Momentum

- The momentum algorithm introduces a variable v that plays the role of velocity—it is the direction and speed at which the parameters move through theparameter space.
- The velocity is set to an exponentially decaying average of the negative gradient.
- In this algorithm, we assume unit mass, so the velocity vector may also be regarded as momentum of particle.
- A hyper parameter α ∈ [0, 1) determines how quickly the contributions of previous gradients exponentially decay.
- The update rule is given by:

$$v = \alpha v - \in \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f\left(x^{(i)};\ \theta\right), y^{(i)})$$

$$\theta = \theta + v$$

Deep Feedforward Networks

SDG Algorithm with momentum

```
input: learning rate ∈, momentum parameter α,
       initial velocity $v$, initial parameter θ
while stopping criteria is not met do
    Sample a minibatch of m examples {$x^{(1)}$, $x^{(2)}$,…, $x^{(m)}$}
      from training set with corresponding targets $y^{(i)}$
    Compute gradient estimate:
        $g = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x^{(i)};\ \theta), y^{(i)})$
    Update velocity $v = \alpha v - \in g$
    Update $\theta$ as: $\theta = \theta + v$
End //while
```

Deep Feedforward Networks

Nesterov Momentum

- Sutskever et al. (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method (Nesterov, 1983, 2004).
- The update rules in this case are given by:

$$v = \alpha v - \in \nabla_\theta \left[\frac{1}{m} \sum_{i=1}^{m} L(f\left(x^{(i)};\ \theta + \alpha v\right), y^{(i)}) \right]$$

$$\theta = \theta + v$$

- Here the parameters α and ∈ play a similar role as in standard momentum method.
- The difference between Nesterov momentum and standard momentum is where the gradient is evaluated.
- With Nesterov momentum the gradient is evaluated after the current velocity is applied.
- Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum.

Deep Feedforward Networks

SDG Algorithm with Nesterov momentum

```
input: learning rate $\in$, momentum parameter $\alpha$,
        initial velocity $v$, initial parameter $\theta$
while stopping criteria is not met do
    Sample a minibatch of m examples $\{x^{(1)}, x^{(2)},\dots, x^{(m)}\}$
    from training set with corresponding targets $y^{(i)}$
    Apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$
    Compute gradient (at interim point):
        $g = \frac{1}{m} \nabla_{\theta} \sum_{i=1}^{m} L(f(x^{(i)};\ \theta), y^{(i)})$
    Update velocity $v = \alpha v - \in g$
    Update $\theta$ as: $\theta = \theta + v$
End //while
```

- One heuristics for choosing the initial scale of the weights of a fully connected layer with m inputs and n outputs is by sampling each weight from U $(-1/\sqrt{m}, 1/\sqrt{m})$.

Deep Feedforward Networks

There are following main components in a machine/ deep learning model:

- **COST FUNCTIONS**
- **OUTPUT UNITS**
- **HIDDEN UNITS**
- **ARCHITECTURE DESIGN**

Deep Feedforward Networks

A. COST FUNCTIONS

- Fortunately, the cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- In most cases, a parametric model defines a distribution $p(y|x; \theta)$ and we simply use the principle of maximum likelihood.
- It means that we use the **cross-entropy** between the training data and model's predictions as **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 $p(y|x)$.
- The loss functions allow us to train a predictor of these estimates.

Deep Feedforward Networks

- The total cost function to train a neural network often combines one of the primary cost functions with a regularization term.
- We need to define a loss function to objectively measure how much the network's predicted output is different than the expected output (the corresponding label).
- Most modern neural networks are trained using maximum likelihood, i.e., **cost function** is simply the negative log-likelihood, equivalently described as the **cross-entropy** between training data and model distribution
- We use the cross entropy loss function for classification problems, and quadratic loss function for regression problems.
- Cost function is $L(\theta) = -E_{\boldsymbol{X},\boldsymbol{Y}\sim\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x})$ (A1)

  which is expectation of $\log p_{model}(\boldsymbol{y}|\boldsymbol{x})$ w.r.t. $\hat{p}_{data}$(x, y).

Deep Feedforward Networks

- The specific form of cost function changes from model to model, depending on the specific form of log p_{model}.
- Eqn.(A1) typically yields some terms that do not depend on model parameters & may be discarded. For example,

$$p_{model}(y|x) = \mathbb{N}(y; f(x;\theta), I) = \frac{1}{2\pi I}\, e^{-\frac{(y-f(x;\theta))^2}{2I}}$$

- The mean squared error cost,

$$J(\theta) = -E_{\mathbf{X},\mathbf{Y}\sim\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x})$$

- Cost function: $J(\theta) = -\frac{1}{2}E_{\mathbf{X},\mathbf{Y}\sim\hat{p}_{data}}\|y - f(x;\theta)\|^2 + \text{constnt}$

  up to a scaling factor of 1/2 and a term independent of θ
- There is equivalence between maximum likelihood estimation with an output distribution and minimization of mean squared error for a linear model.
- This equivalence holds regardless of f(x; θ) used to predict the mean of the Gaussian.

Deep Feedforward Networks

- One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for learning algorithm.
- Functions that saturate (become very flat) undermine this objective because they make the gradient to become very small.
- In many cases this happens because the activation functions used to produce the output of hidden units or the output units saturate.
- The negative log-likelihood helps to avoid this problem for many models.
- Many output units involve an exp function that can saturate when its argument is very negative.
- The log function in negative log-likelihood cost function undoes the exp of some output units.

Deep Feedforward Networks

- One unusual property of the cross-entropy cost used to perform maximum likelihood estimation is that it usually does not have a minimum value when applied to the models commonly used in practice.
- For discrete output variables, most models are parametrized in such a way that they cannot represent a probability of zero or one, but can come arbitrarily close to doing so. Example is Logistic regression .
- For real-valued output variables, if the model can control the density of output distribution (for example, by learning the variance parameter of a Gaussian output distribution), then it becomes possible to assign extremely high density to the correct training set outputs, resulting in cross-entropy approaching negative infinity.
- The choice of cost function is tightly coupled with the choice of output unit.

Deep Feedforward Networks

B. OUTPUT UNITS

- Most of the time, we simply use the cross-entropy between the data distribution and 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. Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well.
- We suppose that the feedforward network provides a set of hidden features, defined by $h = f(x; \theta)$.
- The role of output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

Deep Feedforward Networks

i. Linear Units for Gaussian Output Distributions

- One simple kind of output unit is an unit based on an affine transformation with no nonlinearity, which are often just called linear units.
- Given features h, a layer of linear output units produces a vector $\hat{y} = w^T h + b$.
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:
$$p(y|x) = \mathbb{N}(y; \hat{y}, I)$$
- Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.
- The 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 Feedforward Networks

ii. Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- 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 network needs to predict only P(y = 1 | x). For this number to be a valid probability, it must lie in the interval [0, 1].
- Suppose we were to use a linear unit, and threshold its value to obtain a valid probability:

$$p(y = 1|x) = \max\{0, \min\{1, w^T h + b\}\}$$

Deep Feedforward Networks

- This indeed defines 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 output w.r.t. its parameters would be 0.
- A gradient of 0 is problematic because learning algorithm no longer has a guide for how to improve corresponding parameters.
- It is better to use a different approach, which ensures that there is always a strong gradient whenever the model has the wrong answer.
- This approach is based on using the **sigmoid output** units combined with maximum likelihood, which is defined as

$$\hat{y} = \rho(w^T h + b)$$

Deep Feedforward Networks

- We can think of sigmoid unit as having two components:
  i. *it uses a linear layer to compute* ***z = w^{T}h + b****.*
  ii. *it uses* ***Sigmoid activation*** *function to convert z into a probability.*
- The sigmoid can be motivated by constructing an unnormalized probability distribution $\hat{P}(y)$, which does not sum to 1 and then divide by an appropriate constant to obtain a valid probability distribution.
- If we assume that unnormalized log probabilities are linear in y & z, we exponentiate to obtain unnormalized probabilities, i.e., $\log \hat{P}(y) = yz$ or $\hat{P}(y) = \exp(yz)$
- We then normalize to see that this yields a Bernoulli distribution controlled by a Sigmoidal transformation of z:

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)} \quad \text{or} \quad P(y) = \sigma((y-1)z),$$

where $\sigma(x) = (1 + \exp(-x))^{-1}$

Deep Feedforward Networks

iii. Softmax Units

- When we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function.
- It can be seen as a generalization of Sigmoid function that is used to represent a probability distribution over a binary variable.
- In case of binary variables, we produce a single number: $\hat{y} = P(y = 1|x)$ between 0 and 1, and because we want the logarithm of the number to be well-behaved for gradient-based optimization of log-likelihood, we choose to instead predict a number $\mathrm{z} = \log\hat{P}(y = 1|x)$.
- Exponentiating and normalizing gives us a Bernoulli distribution controlled by the Sigmoid function.
- To generalize for a discrete variable with n values, we need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\boldsymbol{x})$.

Deep Feedforward Networks

iii. Softmax Units

- We require not only that each element $\widehat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution.
- The same approach is followed to generalize the multinoulli distribution.
- First, a linear layer predicts the unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{h} + \boldsymbol{b}$$

where, $z_i = \log\hat{P}(y = i|\boldsymbol{x})$, bold letters are vectors

- The softmax function can then exponentiate and normalize z to obtain the desired $\widehat{\boldsymbol{y}}$. $(\widehat{y} = \rho(w^T h + b))$

$$softmax(z)_i = \frac{\exp(zi)}{\sum_j \exp(z_j)}$$

Deep Feedforward Networks

C. HIDDEN UNITS

- So far we have focused our discussion on design choices for neural networks that are common to most parametric machine learning models trained with gradient based optimization.
- Now we turn to an issue that is unique to **feedforward neural networks**: ***how to choose the type of hidden unit to use in hidden layers of a model***.
- The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.
- **Rectified linear (ReLU)** units are an excellent default choice of a hidden unit.
- Many other types of hidden units are available. It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice).

Deep Feedforward Networks

- It is usually impossible 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.
- Some of hidden units are not actually differentiable at all input points, e.g., ReLU function g(z) = max{0, z} at z = 0.
- This may seem that it invalidates gradient g for use with a gradient based learning algorithm.
- In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.
- This is because of:
  - Neural network training algorithms do not arrive at a local minimum of the cost function, but instead merely reduce its value significantly.

Deep Feedforward Networks

- we do not expect training to actually reach a point where the gradient is 0, it is acceptable for the minima of cost function to correspond to the points with undefined gradient.

- Hidden units that are not differentiable are usually non-differentiable at only a small number of points.
- The functions used in the context of neural networks usually have defined left derivatives and defined right derivative.
- In case of $g(z) = \max\{0, z\}$, the left derivative at $z = 0$ is defined as 0 and the right derivative is 1.
- Most hidden units can be considered as accepting a vector of inputs x, computing an affine transformation $z = w^T x+b$, & applying an element-wise nonlinear function g(z) and *they are distinguished from each other only by the choice of the form of the activation function g(z).*

Deep Feedforward Networks

- **Rectified linear units (ReLU)** 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 to remain large whenever the unit is active.
- The gradients are not only large but also consistent and 2nd derivative of rectifying operation is 0 almost every where, and its 1st derivative is 1 every where the unit is active.
- The gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.

Deep Feedforward Networks

- ReLU are used on top of an affine transformation:

$$\boldsymbol{h} = \boldsymbol{g}(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{h} + \boldsymbol{b})$$

- When initializing the parameters of affine transformation, it can be a good practice to set all elements of $\boldsymbol{b}$ to a small, positive value, such as 0.1.
- This makes it very likely that ReLU will be initially active for most inputs in training set and allow the derivatives to pass through.
- One drawback to ReLU is that they cannot learn via gradient based methods on examples for which their activation is zero.
- A variety of generalizations of ReLU guarantee that they receive gradient everywhere.
- Most of these generalizations perform comparably to rectified linear units and occasionally perform better.

Deep Feedforward Networks

- Three generalizations of rectified linear units are based on using a non-zero slope α_i for $z_i < 0$:

$$h_i = g(z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0,\ z_i)$$

- **Absolute value rectification** fixes $\alpha_i = -1$ to obtain g(z) = |z|. It is used for object recognition from images,

  where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

- A **leaky ReLU** fixes α_i to a small value like 0.01.

- A **parametric ReLU** or PReLU treats α_i as a learnable parameter.

Deep Feedforward Networks

Maxout units generalize the rectified linear units further.

They, instead of applying an element-wise function g(z) on inputs, divide z into groups of k values.

- Each maxout unit then outputs the maximum element of one of these groups:

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

where $G^{(i)}$ is set of indices into 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.
- A maxout unit can learn a piecewise linear, convex function with up to k pieces and thus can be seen as learning the activation function itself rather than just the relationship between units.

Deep Feedforward Networks

- 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 leaky or parametric ReLU, or can learn to implement a totally different function altogether.
- The maxout layer will be parametrized differently from any of these other layer types, so the learning dynamics will be different even in the cases where maxout learns to implement the same function of x as one of the other layer types.
- Each maxout unit is parametrized by k weight vectors instead of just one, so maxout units typically need more regularization than the rectified linear units.

Deep Feedforward Networks

- They can work well without regularization if the training set is large and the number of pieces per unit is kept low.
- In some cases, one gains some statistical & computational advantages by requiring fewer parameters.
- Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.
- Since each unit is driven by multiple filters, maxout units have some redundancy that helps them to resist a phenomenon called catastrophic forgetting in which the neural networks forget how to perform tasks that they were trained on in the past.
- ReLU and its variants are based on the principle that models are easier to optimize if their behavior is closer to linear.

Deep Feedforward Networks

- Prior to the introduction of rectified linear units, most neural networks used **logistic sigmoid** activation function $g(z) = \sigma(z)$ or **hyperbolic tangent** activation function $g(z) = \tanh(z)$.
- These functions are closely related as $\tanh(z) = 2\sigma(2z) - 1$.
- We have already discussed sigmoid units as output units, used to predict the probability that a binary variable is 1.
- 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.

Deep Feedforward Networks

- Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo saturation of sigmoid in output layer.
- 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 σ(0) = ½.
- Because tanh is similar to the identity function near 0, training a deep neural network:

$$\hat{y} = \boldsymbol{w}^T \; tanh(\boldsymbol{U}^T tanh(\boldsymbol{V}^T \boldsymbol{x}))$$

  resembles training a linear model $\hat{y} = \boldsymbol{w}^T \; \boldsymbol{U}^T \boldsymbol{V}^T \boldsymbol{x}$ so long

  as the activations of the network can be kept small.
- Sigmoidal activation functions are more common in settings other than feedforward networks.

Deep Feedforward Networks

Some other reasonably common hidden unit types include:

- Radial basis function (RBF) unit: $h_i = \exp(-\frac{1}{\sigma_i^2}\left|\left|\boldsymbol{W}_{:,\boldsymbol{i}} - \boldsymbol{x}\right|\right|^2)$

  This function becomes more active as x approaches a template $\boldsymbol{W}_{:,\boldsymbol{i}}$. As it saturates to 0 for most x, it can be difficult to optimize.

- Softplus: $g(a) = \log(1 + e^a)$.

  This is a smooth version of rectifier, for function approximation and for the conditional distributions of the undirected probabilistic models.

- Hard tanh: this is shaped similarly to the tanh and the rectifier, but it is bounded, g(a) = max(−1, min(1 , a)).

- Hidden unit design remains an active area of research and many useful hidden unit types remain to be discovered.

Deep Feedforward Networks

D. 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.
- 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

$$h^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)T}\ \boldsymbol{x} + \boldsymbol{b}^{(1)}$$

and the second layer is given by

$$h^{(2)} = g^{(2)}(\boldsymbol{W}^{(2)T}\ \boldsymbol{x} + \boldsymbol{b}^{(2)}$$ and so on.

Deep Feedforward Networks

- A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only the linear functions.
- It has the advantage of being easy to train as many loss functions result in convex optimization problems when applied to linear models. Unfortunately, we often want to learn nonlinear functions.
- One might think to presume that learning a nonlinear function requires designing a specialized model family for the kind of nonlinearity we want to learn.
- Fortunately, the feedforward networks with hidden layers provide a universal approximation framework.
- The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.

Deep Feedforward Networks

- Specifically, 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 as logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another space with any desired non-zero amount of error, provided that the network is given enough hidden units.
- The concept of Borel measurability, for our purposes, suffices to say that any continuous function on a closed and bounded subset of R^n 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.

Measurable Functions

A measurable space is a set A along with its non-empty collection of subsets of A, **S**, such that (A, **S**) satisfies two conditions:

- If A and B are among the collection of subsets in **S**, then A – B will be also a part of **S**.
- For any collection of subsets A_1, A_2, A_3, …, the union of all the A_i's is equal to **S**.
- The elements of the collection **S** are called measurable sets.
- let (A, **X**) and (B, **Y**) be measurable spaces and if f be a function from **X** into **Y**, that is, f: A→B is said to be measurable if $f^{-1}(B) \in$ **X** for every B in **Y**.

Deep Feedforward Networks

- Universal approximation theorem means that regardless of what function we are trying to learn, a large MLP will be able to represent this function.
- But, it is not guaranteed that the training algorithm will be able to learn that function.
- Even if the MLP is able to represent the function, learning can fail for two different reasons.

   i. The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.

   ii. The training algorithm might choose the wrong function due to overfitting.

- Feedforward networks provide a universal system for representing the functions, in the sense that, for a given function, there exists a feedforward network that approximates the function.

Deep Feedforward Networks

- There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to the points not in the training set.
- Universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but it doesn’t 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.
- In worse case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs be distinguished) may be required.
- This is easy in binary case: no. of possible binary functions on vectors v∈$\{0, 1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits that will in general require O(2^n) degrees of freedom.

Deep Feedforward Networks

- 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 can reduce the amount of generalization error.
- There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value *d*, but which require a much larger model if depth is restricted to be less than or equal to d.
- In many cases, the number of hidden units required by the shallow model is exponential in n.

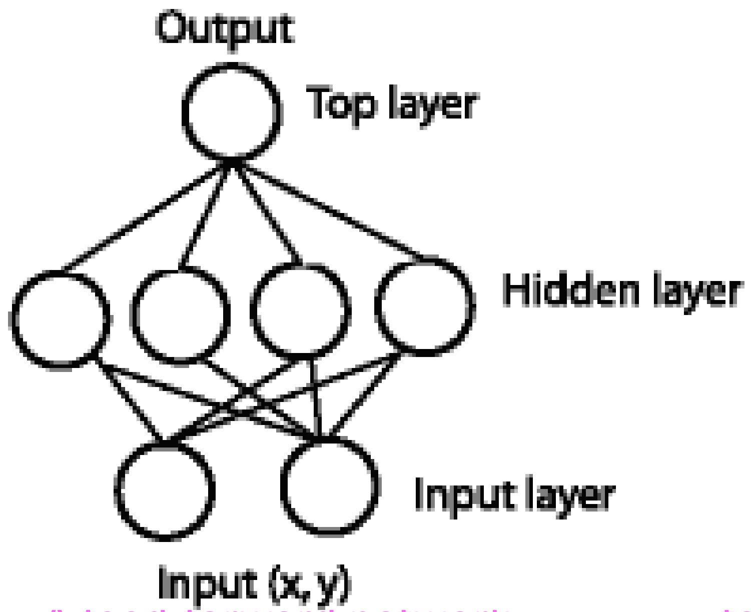
Summary: Feedforward Networks

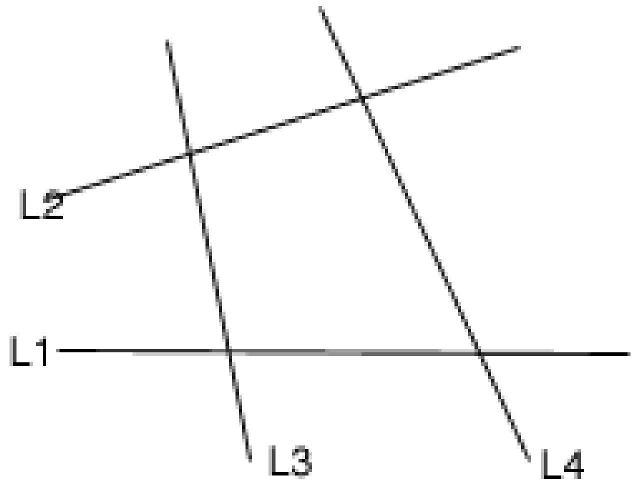
A feedforward network has the following characteristics:

- A computation unit or perceptron or neuron can classify points into two regions that are linearly separable.
- We can extend it to separate out points into two regions that are not linearly separable.
- Perceptrons are arranged in layers, with first layer taking in inputs and last layer producing outputs.
- Middle layers have no connection with the external world, and hence are called hidden layers.
- A perceptron in one layer is connected to some perceptron on next layer.
- Hence information is constantly "fed forward" from one layer to next, and that is why these networks are called feed-forward networks.
- There is no connection among perceptrons in same layer.

Summary: Feedforward Networks

- Consider the following network:





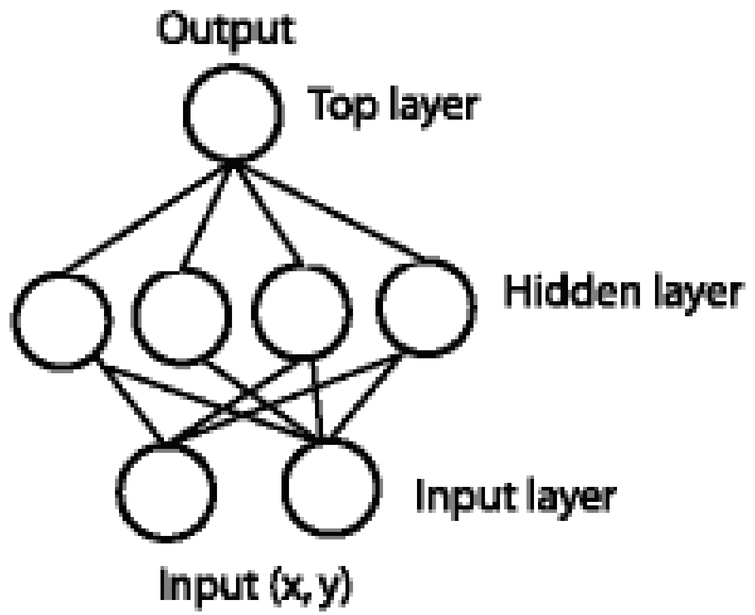
A feed-forward network one hidden layer

four lines each dividing plane into with two linearly separable regions

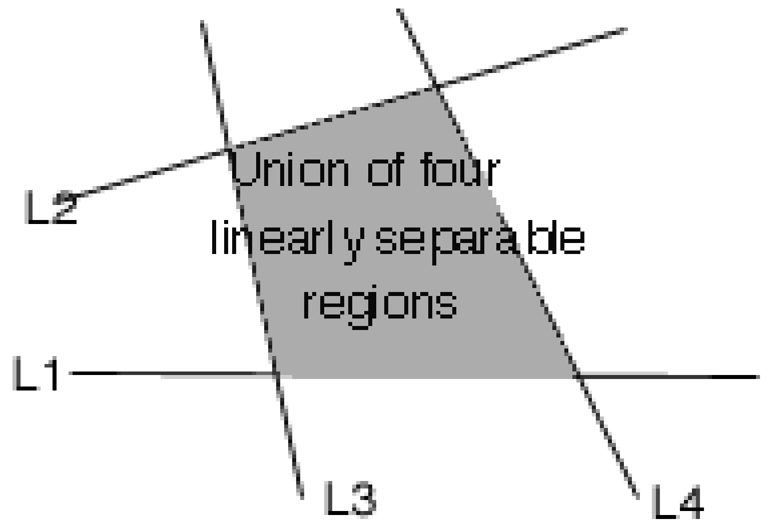
- Training data (x, y) is fed into network through units in input layer. With four units that are independent of each other in hidden layer, point (x, y) is classified into 4 pairs of linearly separable regions, each of which has a unique line separating the region.
- Top unit performs logical operations on outputs of hidden layers so that the whole network classifies input points in 2 regions that might not be linearly separable.

Summary: Feedforward Networks

- Using AND operator on these four outputs, one gets the intersection of four regions that forms the center region.



A feed-forward network with one hidden layer



Intersection of 4 linearly separable regions forms center region

- By varying number of nodes in hidden layer, number of layers, and number of input and output nodes, one can classify the points in arbitrary dimension into an arbitrary number of groups.
- Hence the feed-forward networks are commonly used for classification.