Theoretical Foundations of Machine Learning for Economists: Lecture 3

Sonan Memon
Lecturer, Institute of Business Administration, Karachi

14th April 2021

Outline of Lecture 3

i. Introduction to Deep Feed Forward Neural Networks
ii. Gradient Based Learning
iii. Output Units and Hidden Units
iv. Network Architecture Design
v. Backpropagation and Stochastic Gradient Descent
vi. Regularization of DNN's
vii. Applications in Economics
viii. Implementation in R

Readings for Lecture 3

i. Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. MIT Press, chapters 6-8.

ii. Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large scale machine learning. SIAM Review, 60(2):223–311.

iii. Fernández-Villaverde, J., Hurtado, S., & Nuno, G. (2019). Financial frictions and the wealth distribution (No. w26302). National Bureau of Economic Research.

iv. Hartford, J., Lewis, G., Leyton-Brown, K., & Taddy, M. (2016). Counterfactual prediction with deep instrumental variables networks. arXiv preprint arXiv:1612.09596.

v. Slides: https://github.com/sonanmemon/Introduction-to-ML-For-Economists

Motivation

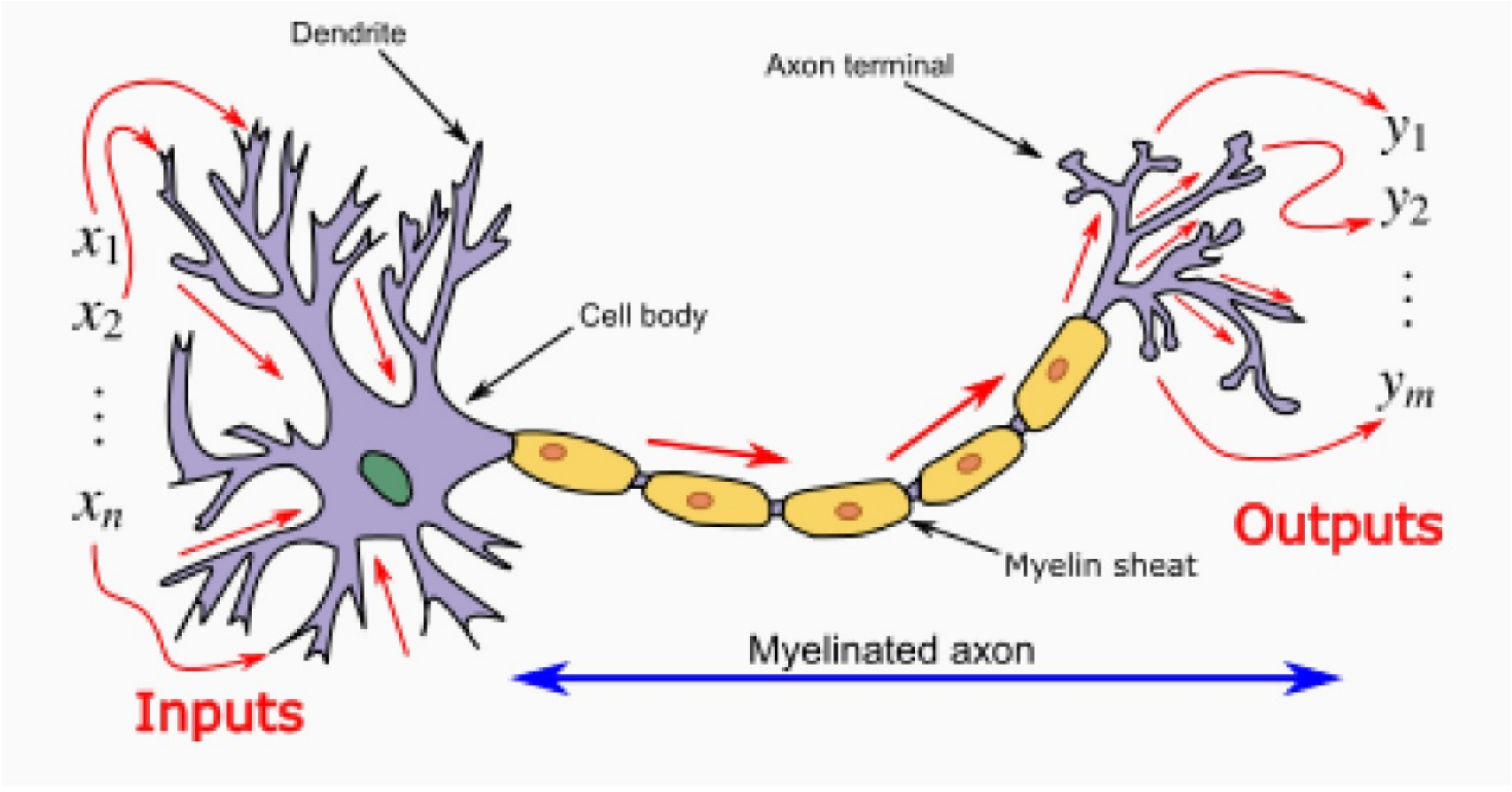
i. Neural Networks are complicated function approximation methods, well suited for high dimensional and nonlinear settings.

ii. Are *Networks* because there is a composition of many functions involved in the function approximation method.

iii. Are *Deep* since the number of layers or the functions to be composed could be quite large.

iv. Are *Neural* because of a loose analogy with neuroscience: hidden units in hidden layers, just like real neurons get inputs from many other units in last hidden layer, which is used to determine their activation.

Biological Motivation



Applications

i. AlphaGo.

ii. Object recognition, Facial Recognition and Computer Vision: Convolutional Networks (form of deep, feed forward network).

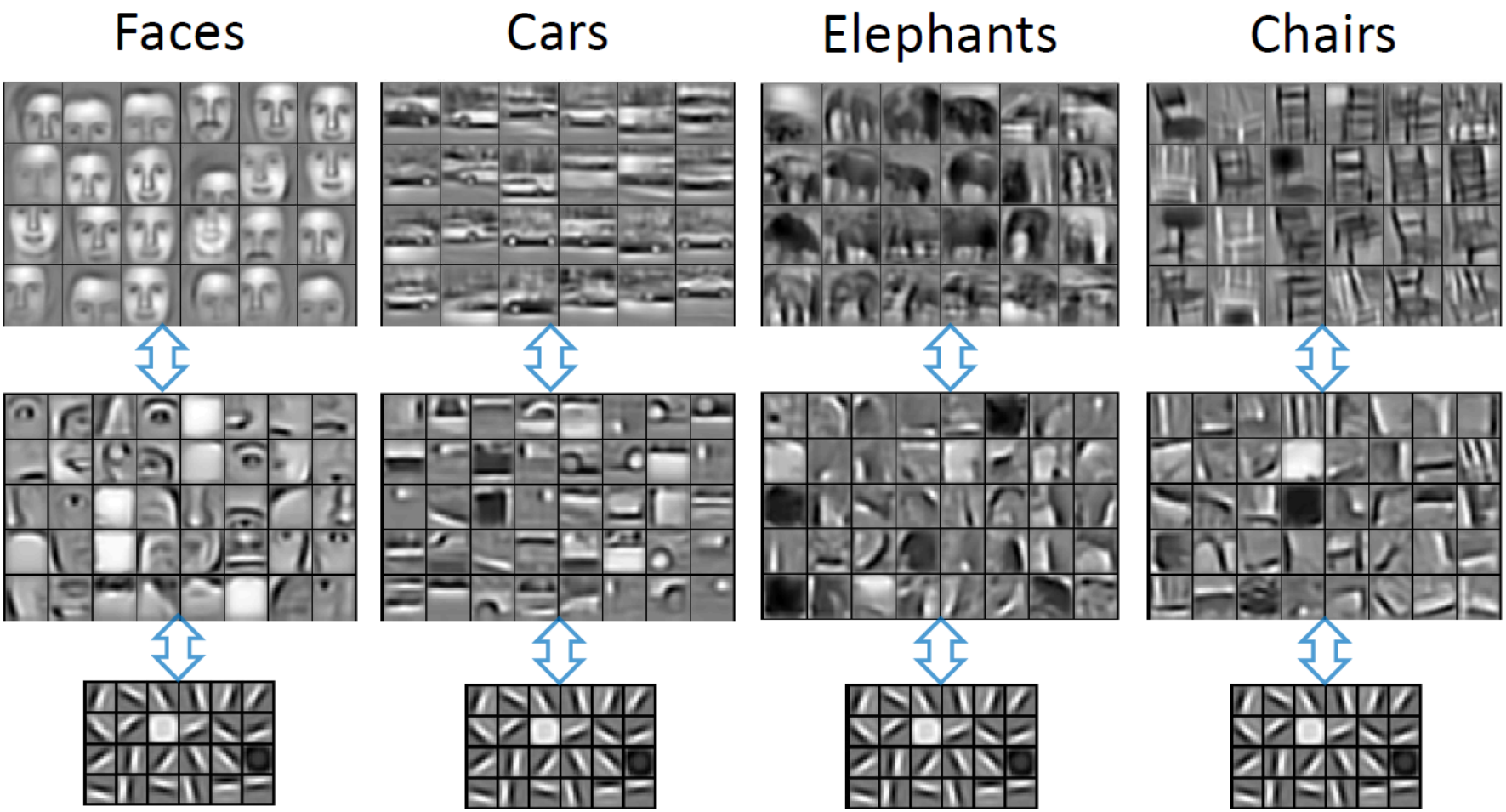
iii. Prediction of stock prices.

iv. Recurrent Networks used for natural language processing (backward connections) such as translation, speech or digit recognition and syntactic parsing.

v. Economic Applications: Solution of Nonlinear DSGE Models, High Dimensional Econometrics (Causal AI), High Dimensional Forecasting, Natural Language Processing and Computational Linguistics.

AlphaGo: The Greatest Go Player of All Time?

Object Recognition Examples



Introduction to Deep Neural Networks

i. Choosing the cost function, form of hidden and output units and optimizer.

ii. Network Design: Activation Functions, Network Architecture and Output Layers.

iii. Calculating Gradients for Optimization: Back propagation and Stochastic Gradient Descent.

iv. Regularization: Penalty Terms and Early Stopping.

Deep Feed Forward Networks

i. Deep Feed Forward Networks or MLP's are deep learning models in which information only flows forward and no *feedback*, unlike recurrent networks.

ii. The goal is to learn unknown function $y = f^*(\boldsymbol{x})$.

iii. $y = f(\boldsymbol{x}) = f^n\left(f^{n-1}\left(f^{n-2}...\left(f^1(\boldsymbol{x})\right)\right)\right)$

iv. "Deep" here refers to the number of layers which are n in equation above.

v. Width of network refers to the number of *hidden units* in one layer.

vi. Hidden layers: only observe data on $\boldsymbol{x}$ and y, not on intermediate layers, which have to be determined by the learning algorithm.

Flow Representation

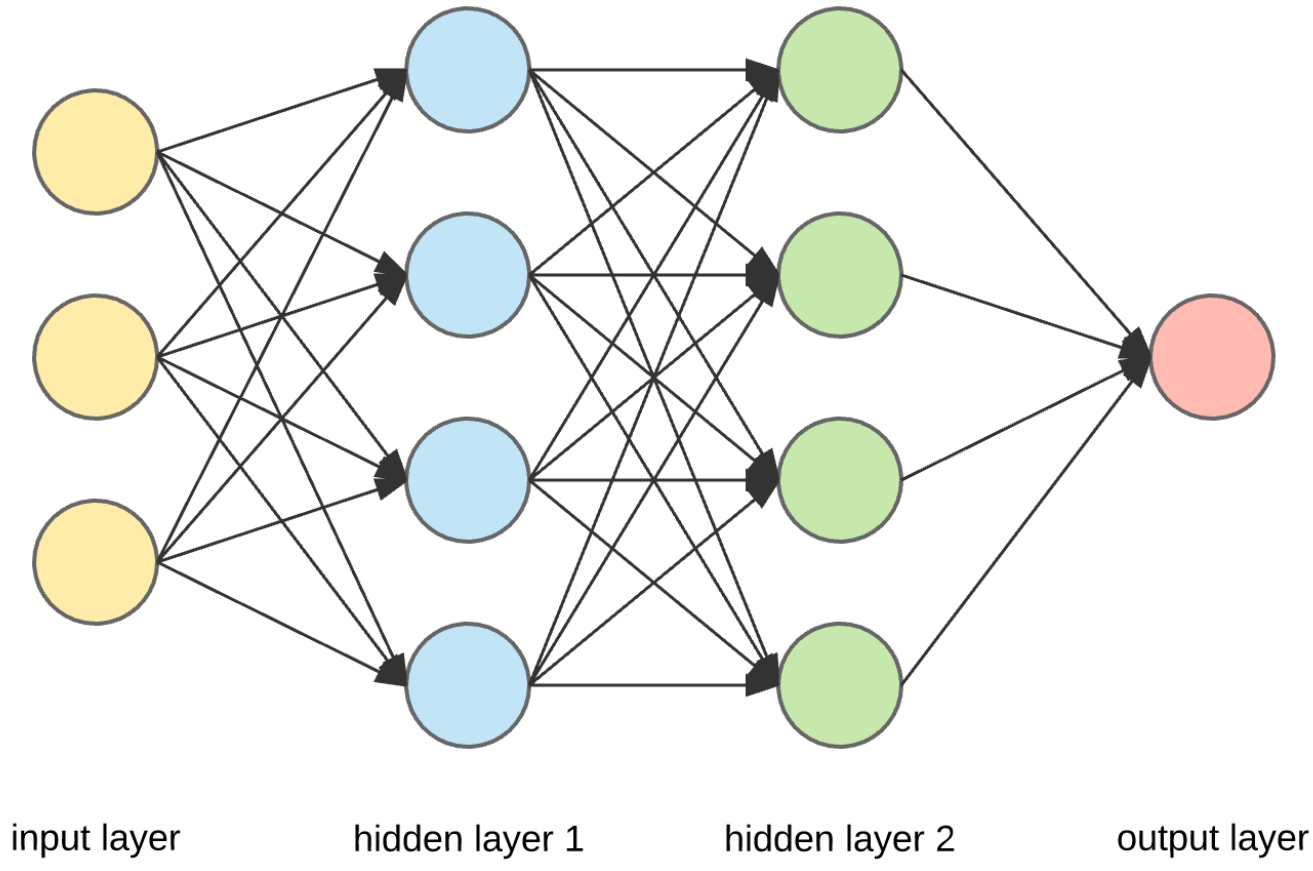


Figure: Flow Representation

Deep Feed Forward Networks

i. Simple linear model of form $f = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + b$ cannot model interactions between inputs and has limited model capacity.

ii. We apply nonlinear transformation $\phi(\boldsymbol{x})$ before applying linear model to the output of ϕ to overcome limitations of linear models: $y = f\left(\boldsymbol{x};\, \boldsymbol{\theta},\, \boldsymbol{w}\right) = \phi\left(\boldsymbol{x};\, \boldsymbol{\theta}\right)^{T} \boldsymbol{w}$.

iii. How to choose ϕ?:
   - Generirc ϕ
   - Manual engineering through specialized human effort.
   - Learn ϕ using deep learning.

iv. Using deep learning, we optimally learn parameters θ to pin down ϕ and this combines the benefits of approaches 1 and 2 above.

Example: Learning XOR

i. The XOR function ("exclusive or") is an operation on two binary values x_1 and x_2.

ii. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0.

iii. The XOR function provides the target function $y = f^*(\boldsymbol{x})$.

iv. Our model provides a function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ and our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*.

Example: Learning XOR

i. We want our network to perform exactly correctly on the four points $\mathbb{X} = \{[0,0]^T, [0.1]^T, [1,0]^T, [1,1]^T\}$ in training set and no statistical generalization is needed here.

ii. Evaluated on our whole training set, the MSE loss function is: $L(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta})\right)^2$.

iii. The linear model cannot capture the nonlinearity of this function.

Example: Learning XOR

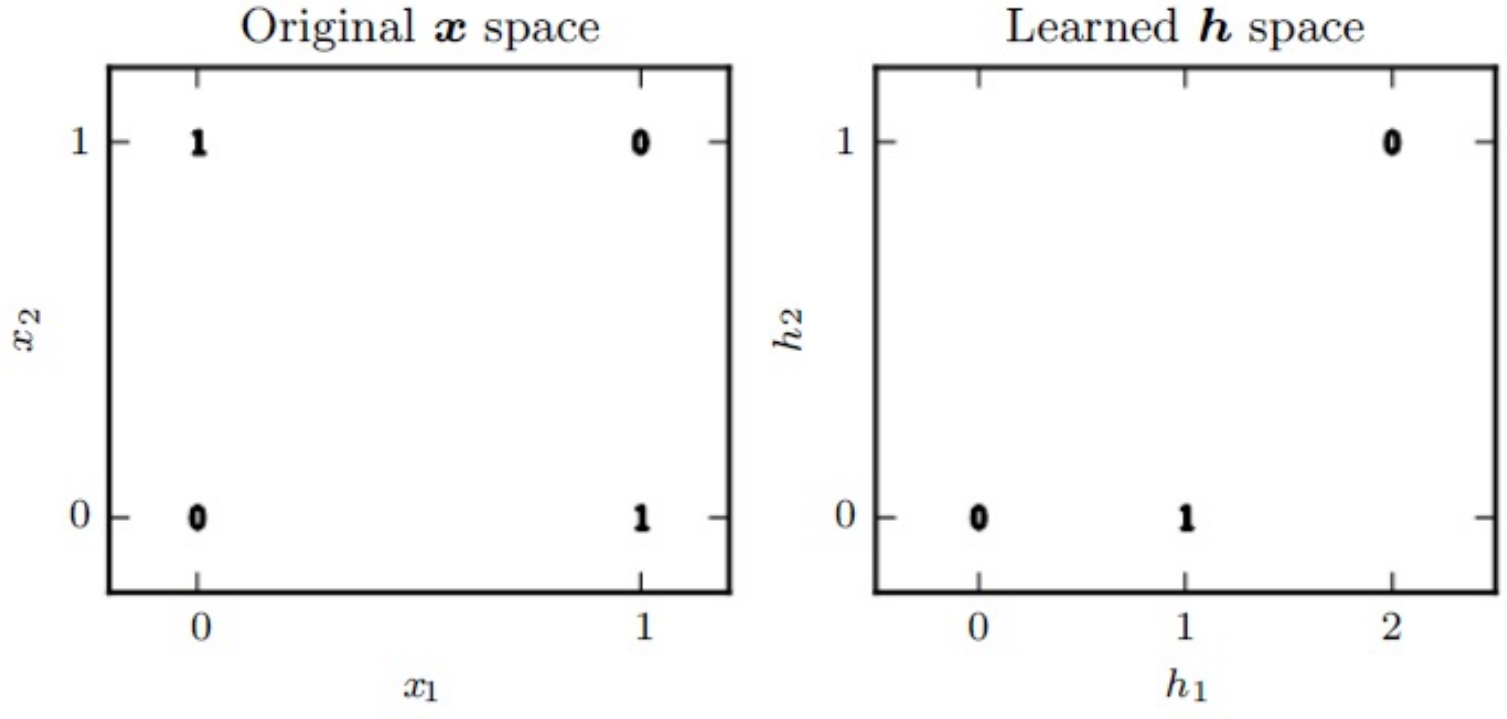
i. We use feed forward network with one hidden layer by defining $f^1(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}$ and $f^2(\boldsymbol{h}) = \boldsymbol{w}^T\boldsymbol{h} + b$, $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$.

ii. If g = ReLU activation function, $f\left(\boldsymbol{x};\, \boldsymbol{W},\, \boldsymbol{c},\, \boldsymbol{w}, b\right) = \boldsymbol{w}^T max\{0, \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}\} + b$.

iii. The hidden layer does feature extraction and a linear model is applied to transformed inputs after applying activation function g.

iv. The initial untransformed space cannot be learned by the linear function but the transformed feature space can be learned by linear function.

Original Space Versus Learned Space



Example: Learning XOR

i. $\boldsymbol{W}$, $\boldsymbol{c}$, b, $\boldsymbol{w}$ can be chosen to exactly find the function f^* in this case.

ii. $\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, $\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, $\boldsymbol{b} = 0$.

iii. With above parameters, the neural network learns every example in the batch perfectly, with zero error.

iv. In a real situation, there might be billions of model parameters and billions of training examples, implying that a gradient-based optimization algorithm can find parameters that produce very little error but not zero error.

Gradient Based Learning

i. Neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than zero value.

ii. Backpropagation and stochastic gradient descent is used for computational efficiency.

iii. Stochastic gradient descent applied to non-convex loss functions has no convergence guarantee, and is sensitive to the values of the initial parameters.

iv. Computing the gradient for neural network models is more complicated but fundamentally similar to gradient based learning of linear models.

Loss Function Choice

i. Parametric model defines conditional distribution $p(\boldsymbol{y}|\boldsymbol{x};\ \boldsymbol{\theta})$ and we use maximum likelihood for loss function, which is simply the negative log-likelihood (cross-entropy between data and model): $L(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \log\ p_{model}\ (\boldsymbol{y}|\boldsymbol{x})$

ii. The loss function used in practice will often add a regularization term.

iii. Functions that saturate make the gradient vanish and inhibit learning. The negative log-likelihood helps to avoid this problem because log in the loss function undoes the exp of some output units such as softmax and sigmoid.

Output Units

i. The choice of cost function is tightly coupled with the choice of output unit since how we represent the output determines the form of the likelihood function.

ii. The role of the output layer is to apply additional transformation on the features in hidden layers: $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$, in line with nature of task.

iii. Linear units, sigmoid units and softmax units are some commonly used output units.

Linear Output Units

i. Simple output units are affine transformation of the hidden units. For instance, given features h, linear output unit will be $\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$.

ii. 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.

iii. Used for regression problems.

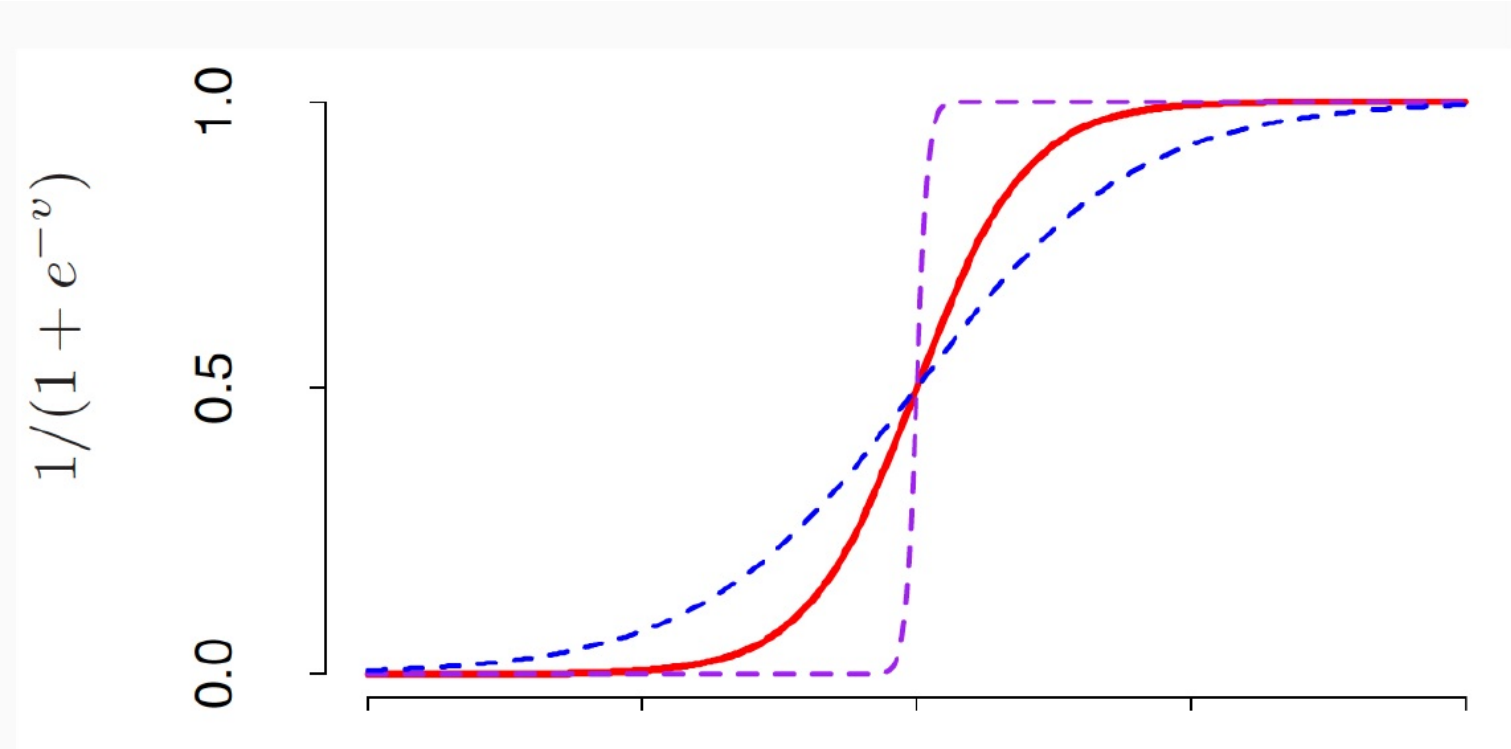
Sigmoid Output Units

i. Many tasks require predicting the value of a binary variable y such as classification problems with two classes.

ii. A Bernoulli distribution is defined by the conditional probability of success, which is what the neural net predicts: $\mathbb{P}(y = 1|\boldsymbol{x})$.

iii. A common solution in this case is to use sigmoid output units, applied on affine transformation of $\boldsymbol{h}$: $y = \sigma\left(\boldsymbol{w}^T\boldsymbol{h} + b\right)$, where σ is logistic sigmoid function.

Sigmoidal Function



Softmax Output Units

i. When we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function.

ii. We need to produce $\hat{\boldsymbol{y}} \in \mathbb{R}^n$ such that $\hat{y}_i = \mathbb{P}\left(y = i|\boldsymbol{x}\right)$, where $\sum_{i=1}^{n} \hat{y}_i = 1$.

iii. $z_i = \log \widetilde{P}(y = i|\boldsymbol{x})$ is the log of un-normalized probability.

iv. First apply affine transformation to get $\boldsymbol{z} = \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{h} + \boldsymbol{b}$, then apply softmax function element-wise to get: $\text{softmax}(z_i) = \frac{\exp\{z_i\}}{\sum_{j=1}^{n} \exp\{z_j\}}$ (exponentiation plus normalization).

Hidden Units

i. Hidden units refer to the activation functions used in hidden layers.

ii. In the 1990's, logistic/sigmoid functions were default choices and hyperbolic tangents took over in late 1990's to 2000's.

iii. Rectified Linear Units (ReLU) are a default choice of hidden units in the modern ML literature since it solves the vanishing gradient problem etc.

iv. $\boldsymbol{h} = g\left(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b}\right)$, where $g(z) = max\{0, z\}$ is the ReLU activation function.

v. $g(z) = \log(1 + e^{z})$ is the softplus activation function. It has been used in recent econ literature.

Sigmoid and Hyberbolic Tangent

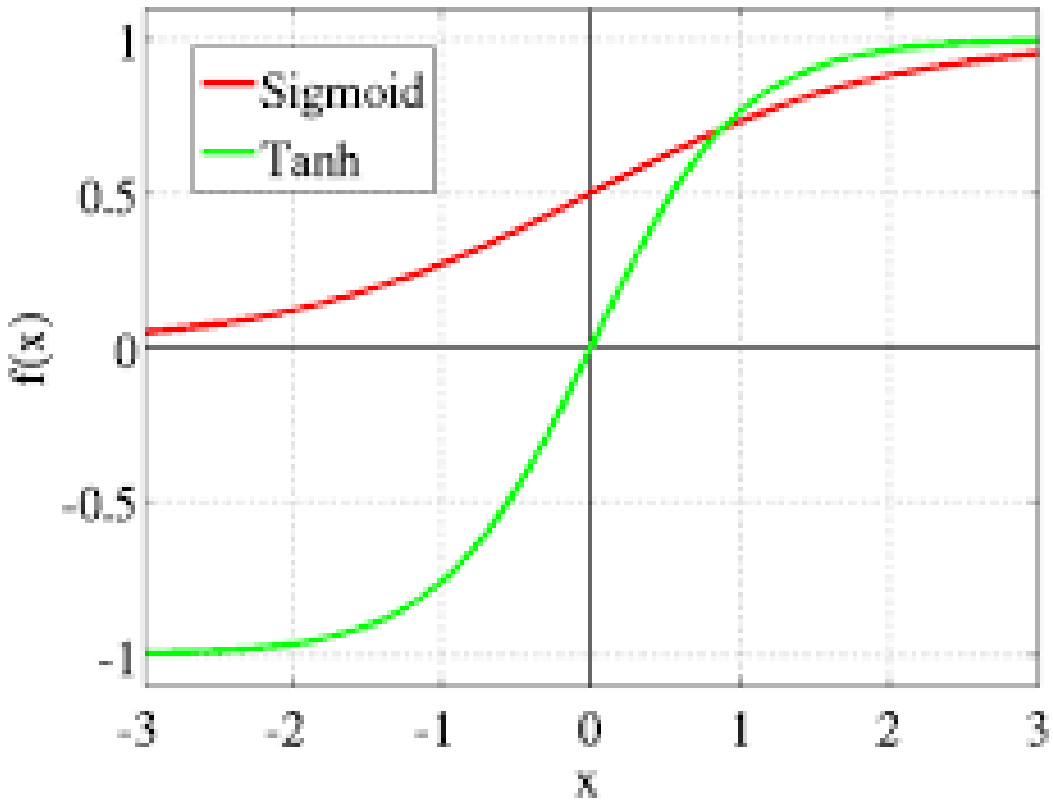
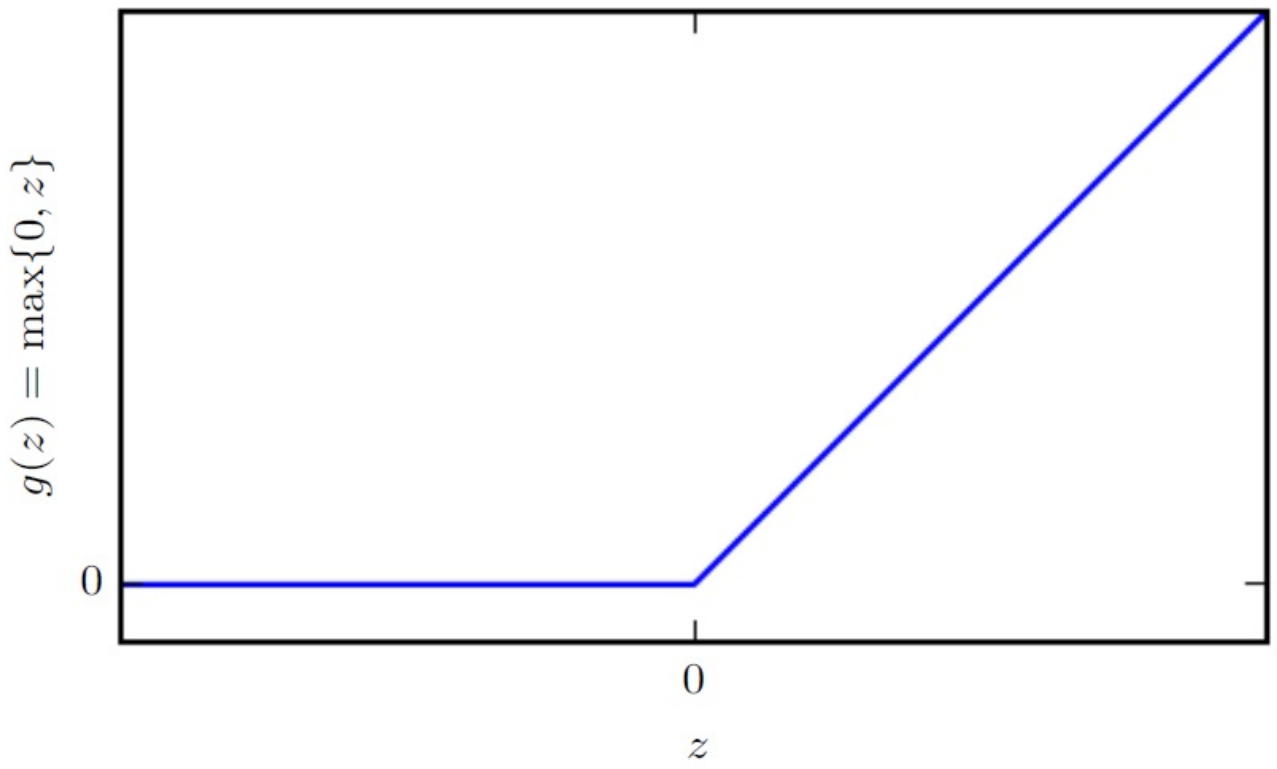
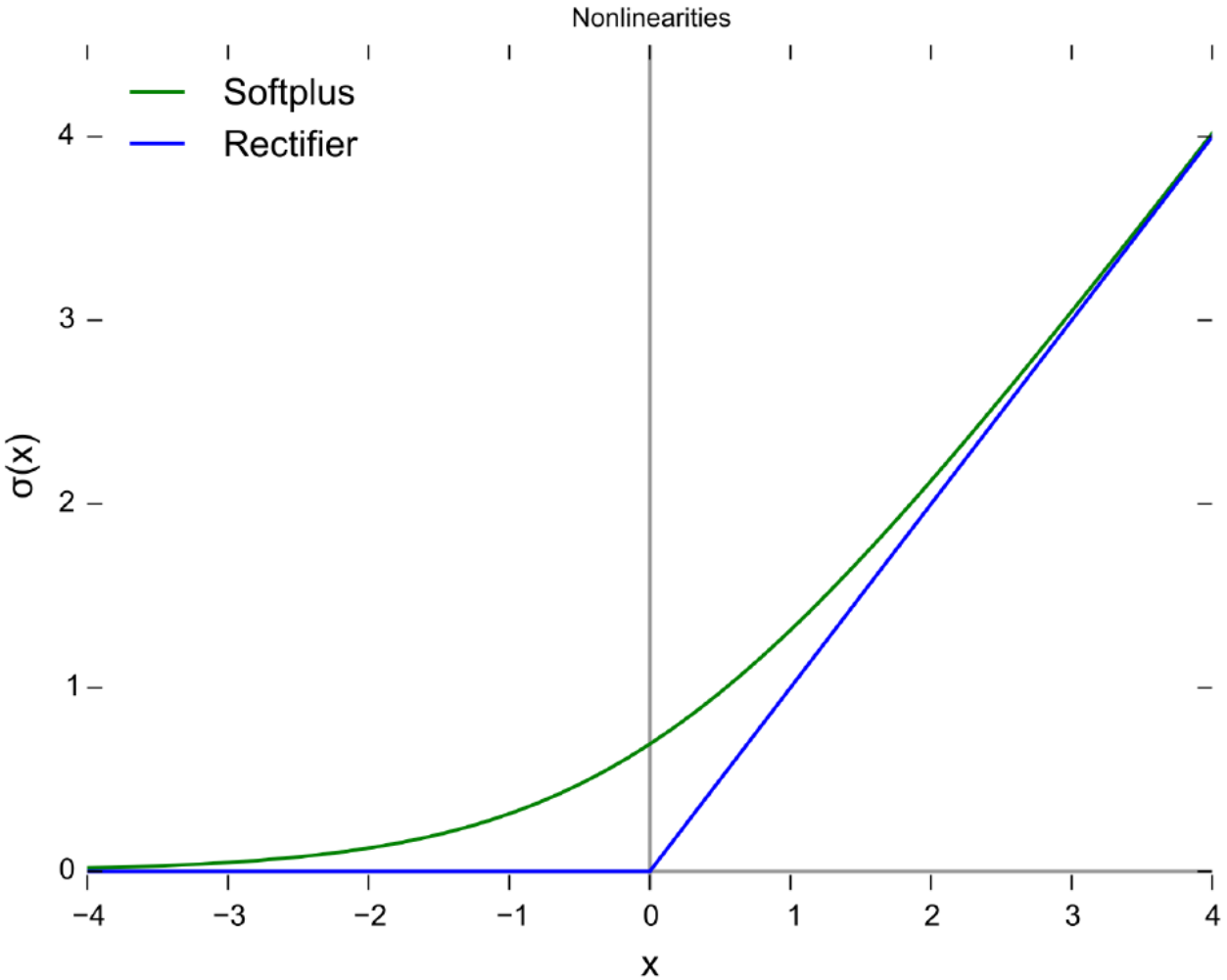


Figure: Sigmoid and Tanh

ReLU Activation Function



Activation Functions: ReLU and Softplus



Network Architecture Design

i. The *depth*, *width* and structure of connections between hidden units across layers are key design features of the architecture of a neural network.

ii. Depth refers to number of layers and width refers to number of hidden units per layer.

iii. For instance, $\boldsymbol{h}^1 = g^1\left((\boldsymbol{W}^1)^T\boldsymbol{x} + \boldsymbol{b}^1\right)$ may be first layer and $\boldsymbol{h}^2 = g^2\left((\boldsymbol{W}^2)^T\boldsymbol{x} + \boldsymbol{b}^2\right)$ may be second layer etc.

iv. Deeper networks can use far fewer units per layer and far fewer parameters to fit data and generalize to the test set better, but are also often harder to optimize.

v. Even one layer with sufficient parameters and units can fit the data in theory.

Universal Approximation Theorem

Theorem (Universal Approx Theorem Hornik et al 1989)

A feed forward network with a linear output layer and at least one hidden layer with any "squashing" activation function such as logistic 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.

Universal Approximation Theorem

i. A feedforward network with a single layer or few layers is sufficient to represent any function in theory but in worst cases, the number of units required in the one or few layers may be prohibitively large.

ii. There is no universally superior optimization algorithm to be applied on a particular training data set, which means that we may fail to learn parameters corresponding to the true function.

iii. Failures may also occur due to over fitting or failure to generalize correctly.

Why Deeper Models?

i. 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.

ii. From a Bayesian perspective, deeper models may reflect our priors that there are many but simpler underlying factors of variation which when composed together constitute the complexity of what has to be learned.

How to Connect Two Layers?

i. When we apply affine transformations of form $\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$ in networks, the matrix $\boldsymbol{W}$ will in general connect every input unit to every output unit.

ii. 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.

iii. This sparsity can reduce the number of parameters and the amount of computation required to evaluate the network but specific pattern of sparsity that is efficient is highly problem-dependent.

Backpropagation (bakcprop)

i. Numerically evaluating the gradient for optimization can be computationally expensive and backprop algorithm does so using a simple and inexpensive procedure.

ii. In a feed forward network information flows forward from inputs $\boldsymbol{x}$, through hidden layers, ultimately producing loss $L(\theta)$.

iii. backprop allows the information from the loss to then flow backwards through the network, in order to compute the gradient.

iv. backprop is not an learning algorithm but an efficient method for computing derivative, which is an intermediate step in the full learning algorithm such as stochastic gradient descent.

Backpropagation: Chain Rule

i. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, where $\boldsymbol{y} \in \mathbb{R}^n$, $\boldsymbol{x} \in \mathbb{R}^m$, we know that $\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$ or in gradient notation,
$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z.$

ii. A straightforward computation of chain rule introduces many subexpressions which have to be recomputed several times and these wasteful computations may even grow exponentially.

iii. backprop performs the chain rule operation for each "edge" of the graph and visits each edge from node u^i to u^j *exactly once*.

iv. backprop stores intermediate derivatives rather than recomputing them, which when combined with its recursive structure improves efficiency.

Backpropagation: Computational Graph

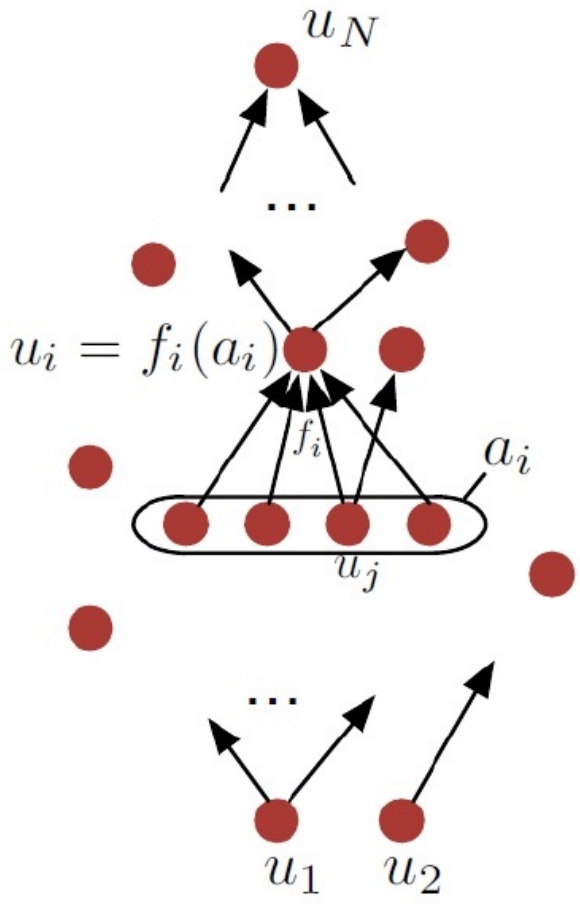
i. Consider computational graph in which we want to compute $\frac{\partial u^n}{\partial u^i}$: gradient of u^n (e.g loss associated with mini-batch) with respect to M input nodes u^i where $i \in \{1, 2, ..., M\}$.

ii. We will assume that the nodes of the graph have been ordered in such a way that we can compute their output one after the other and $u^i = f(A^i)$, where A^i are set of parent nodes of node u^i.

iii. Chain rule implies that $\frac{\partial u^n}{\partial u^j} = \sum_{i:j \in Pa(u^i)} \frac{\partial u^n}{\partial u^i} \frac{\partial u^i}{\partial u^j}$.

iv. The forward computation step shown next has already stored $\frac{\partial u^n}{\partial u^i}$ and the computation for each edge corresponds to computing a partial derivative $\frac{\partial u^i}{\partial u^j}$, performing one multiplication and one addition only.

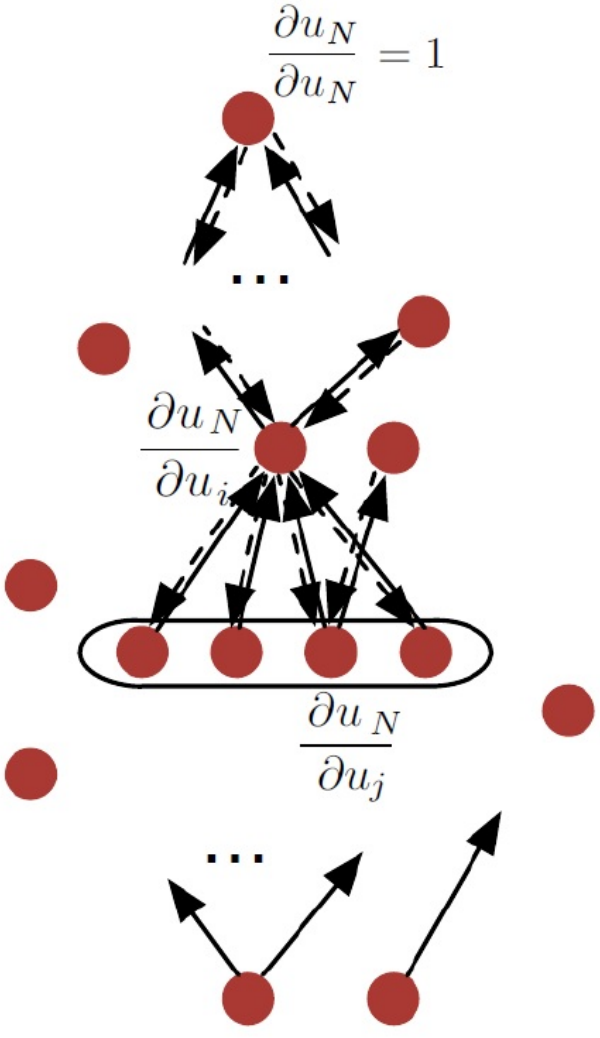
Recursive Forward Computation



Recursive Forward Computation

for $i = 1, ..., M$ **do**
$u_i = x_i$
end for
for $i = M + 1, ..., N$ **do**
$a_i = (u_j)_{j \in parents(i)}$
$u_i = f_i(a_i)$
end for
return u_N

Recursive Backward Computation



Recursive Backward Computation

$\frac{\partial u_N}{\partial u_N} = 1$
for $j = N-1, \ldots, 1$ **do**
$\frac{\partial u_N}{\partial u_j} = \sum_{i:j \in parents(i)} \frac{\partial u_N}{\partial u_i} \frac{\partial u_i}{\partial u_j}$
end for
return $\left(\frac{\partial u_N}{\partial u_i} \right)_{i=1}^{M}$

Regularization for Deep Neural Networks

i. As in other machine learning algorithms, improving predictive performance requires regularization, which can be done ab introducing L^1 (LASSO) or L^2 (Ridge Regression) penalty.

ii. However, tuning using cross-validation is computationally demanding with deep nets.

iii. Alternative regularization method, commonly used is **early stopping:**

   i. Split the data into training and validation samples.
   ii. Run gradient based optimization on training sample and check prediction loss in validation sample at each iteration.
   iii. Stop when prediction loss starts increasing.

Optimization: Gradient Descent

i. The algorithm involves updating the model parameters $\boldsymbol{\theta}$ which would include the weights and biases associated with each layer with a small step in the direction of the gradient of the loss function.

ii. With training data $(\boldsymbol{x^t}, \boldsymbol{y^t})$,
$\boldsymbol{\theta} = \boldsymbol{\theta} + \epsilon \boldsymbol{\nabla}_{\boldsymbol{\theta}} \sum_t L(f(\boldsymbol{x^t}; \boldsymbol{\theta}), \boldsymbol{y^t}; \boldsymbol{\theta})$.

iii. Choice of learning rate is crucial since a too low learning rate slows down learning and too high value can cause divergence or oscillation around the minima.

Gradient Descent

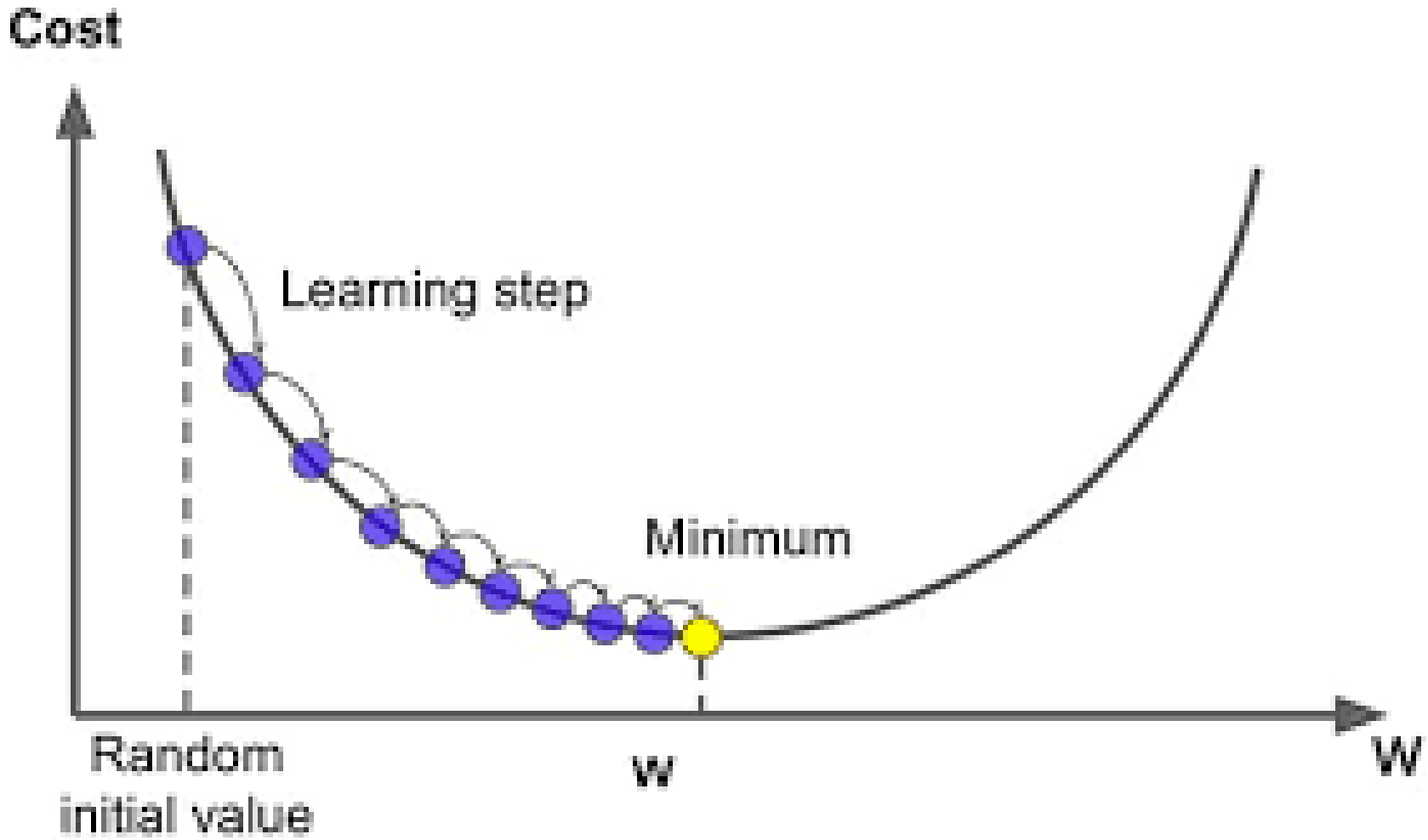


Figure: Learning via GD

Why Stochastic Gradient Descent (SGD)?

i. SGD uses just a random subsample of the data for gradient updating.

ii. The key observation that motivates SGD is that in an (i.i.d.) sampling context, further observations are increasingly redundant.

iii. Formally, the standard error of a gradient estimate based on b observations is of order $\frac{1}{\sqrt{b}}$ but computation time is of order b.

iv. For a large dataset, it would take forever to just calculate one gradient, and do one updating step. During the same time, SGD might have made many steps and come considerably closer to the truth (see Bottou et al. (2018))

Stochastic Gradient Descent Algorithm

Note: η is learning rate
while stopping criterian not met **do**
sample **minibatch** of m examples from the training set $\{\boldsymbol{x}^1, \boldsymbol{x}^2, .., \boldsymbol{x}^m\}$.
set $g = 0$
for $t = 1 : m$ **do**
$\boldsymbol{g} = \boldsymbol{g} + \nabla_\theta L(f(\boldsymbol{x}^t; \boldsymbol{\theta}), \boldsymbol{y}^t; \boldsymbol{\theta})$
end for
Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{g}$
end while

SGD with Momentum

i. Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations of SGD.

ii. The momentum term is usually set to 0.9 or a similar value.

iii. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Stochastic Gradient Descent with Momentum

Note: η is learning rate, momentum parameter is α
Initial parameter θ and initial velocity $\boldsymbol{v}$ must be specified.
while stopping criterian not met **do**
sample **minibatch** of m examples from the training set $\{\boldsymbol{x}^1, \boldsymbol{x}^2, .., \boldsymbol{x}^m\}$.
set $g = 0$
for $t = 1 : m$ **do**
$\boldsymbol{g} = \boldsymbol{g} + \nabla_\theta L(f(\boldsymbol{x}^t; \boldsymbol{\theta}), \boldsymbol{y}^t; \boldsymbol{\theta})$
end for
velocity update: $\boldsymbol{v} = \alpha \boldsymbol{v} - \eta \boldsymbol{g}$
apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{v}$
end while

Economic Application 1: Computation of Nonlinear DSGE Models

Financial Sector and the Wealth Distribution Jesús et al (2020).

i. Empirical evidence in Jorda et al. (2016) shows long term evidence for 17 advanced economies that output growth, volatility, skewness, and tail events depend on leverage.

ii. High leverage also leads recessions Adrian et al. (2019a)

iii. Their model is a nonlinear DSGE model in continuous time with heterogeneous households, representative firm and financial expert along with financial frictions.

Economic Application 1: Computation of Nonlinear DSGE Models

i. Households are subject to uninsurable idiosyncratic risk and can only save in the risk-free bond to insure against idiosyncratic shocks. Meanwhile, financial experts absorb all the capital return risk.

ii. The interaction between the supply of bonds by the financial sector and the precautionary demand for bonds by households produces significant endogenous aggregate risk and an endogenous regime-switching for real variables including leverage across the two stochastic steady states of model (SSS).

Economic Application 1: Computation of Nonlinear DSGE Models

i. Endogenous regime switching generates bimodal and skewed ergodic distributions of aggregate variables, time-variation in volatility and skewness, and supercycles of borrowing and deleveraging.

ii. In the high (low) leverage SSS, endogenous aggregate risk is high (low) and after a negative aggregate shock, the economy suffers deep and protracted (mild) recessions with persistently low wages (quick recovery) since financial sector loses a lot of its wealth.

Economic Application 1: Computation of Nonlinear DSGE Models

i. The wealth distribution is an infinite-dimensional object so standard DP techniques cannot be employed.

ii. The consumption decision rule of households is sharply nonlinear with respect to the aggregate state variables (equity and debt).

iii. The global, nonlinear solution method used is ML: DNN to obtain a flexible, nonlinear approximation of the perceived law of motion of the cross-sectional distribution of assets with finite moments.

iv. Neural networks can capture a nonlinear PLM without having to specify, ex-ante, any concrete structure for it.

Economic Application 1: Computation of Nonlinear DSGE Models

i. By solving the model (read paper for derivation), they derive that $dB_t = (w_t + r_t B_t - C_t)d_t$ is the actual law of motion of debt in the model.

ii. Given their model, it is crucial to find the households' aggregate consumption C_t to compute the equilibrium of the economy given the structural parameters.

iii. Follow Krusell and Smith (1998) and assume that households only use a finite set of n moments of the cross-sectional distribution of assets instead of complete distribution for expectation formation.

iv. The PLM with $n = 1$ is $dB_t = h(B_t, N_t)d_t$ where $h(B_t, N_t) = \frac{\mathbb{E}[dB_t|B_t,N_t]}{dt}$.

Economic Application 1: Computation of Nonlinear DSGE Models

i. The functional form $h(B_t, N_t)$ is approximated by applying NN's to simulated data from the aggregate ergodic distribution.

ii. NN's help approximate the PLM arbitrarily well, allowing for complex nonlinearities in the law of motion of B_t and N_t in the model.

iii. NN's also allow for good extrapolation outside their training areas of the state space unlike other methods using Chebyshev polynomials.

Economic Application 1: Computation of Nonlinear DSGE Models

i. A neural network with a single, hidden layer is used: $h(\boldsymbol{s};\boldsymbol{\theta}) = \theta_0^2 + \sum_{q=1}^{Q} \theta_q^2 \phi\left(\theta_0^1 + \sum_{i=1}^{N} \theta_i^1 s^i\right)$.

ii. ReLU is common choice in ML literature but they use $\phi(x) = log(1 + e^x)$ (softplus) to avoid the kink in ReLU.

iii. Given a choice of Q (number of hidden units), input vector of states $\boldsymbol{s}$ and choice of number of layers and loss function, the θ parameters can be estimated, which help approximate the PLM.

iv. They use batch gradient (not SGD) algorithm in which loss gradient is computed using backprop to update θ parameters.

Economic Application 2: Counterfactual Prediction with Deep IV Networks

i. Decision makers look to the data to model counterfactual questions and ML will do a poor job of predicting the many potential futures associated with each policy option.

ii. For example, optimal pricing requires predicting sales under *changes* to prices.

iii. In order to accurately answer such counterfactual questions it is necessary to model the structural relationship between policy and outcome variables.

iv. This paper uses the concept of IV's to construct systems of ML tasks that can be applied in causal inference on large and unstructured data.

Economic Application 2: Counterfactual Prediction with Deep IV Networks

i. Set up IV system of machine learning tasks that can be targeted with deep learning and allow us to make counterfactual claims and perform causal inference.

ii. Two ML stages: a first stage that models the conditional distribution for treatment given the instruments and covariates, and a second stage which targets a loss function involving integration over the conditional treatment distribution from the first stage.

iii. Both stages use deep neural nets trained via stochastic gradient descent.

iv. Out of sample causal validation as regularization also performed.

Economic Application 2: Counterfactual Prediction with Deep IV Networks

i. $y = g(p, x) + e$, where $\mathbb{E}[pe|x] \neq 0$.

ii. $h(p, x) = g(p, x) + \mathbb{E}[e|x]$

iii. To evaluate policy options, say p_0 and p_1, we can look at the difference in mean outcomes:
$h(p_1; x) - h(p_0; x) = g(p_1; x) - g(p_0; x)$.

iv. In standard, unstructured ML, the prediction model is trained to fit $\mathbb{E}[y|p, x] = g(p, x) + \mathbb{E}[e|p, x] \neq h(p, x)$ since $\mathbb{E}[e|p, x] \neq \mathbb{E}[e|x]$ so that ML will lead to biased counterfactual analysis.

v. IV's allow us to get unbiased estimate $\widehat{h(p, x)}$.

Economic Application 2: Counterfactual Prediction with Deep IV Networks

i. This paper avoids explicit linearization used by standard IV procedures such as 2SLS by instead using DNN's, optimized via SGD to learn F and h.

ii. In first stage, we learn $F(p|x,z)$ by choosing appropriate distribution, parametrized by $\widehat{F} = F_\theta(p|x,z)$.

iii. Estimation proceeds by maximum likelihood via stochastic gradient descent on the implied negative log likelihood.

iv. In the second stage, conditional on estimate of F, the counterfactual function h is approximated by a DNN, paramterized as h_θ, based on minimization of some loss function.

Economic Application 2: Counterfactual Prediction with Deep IV Networks

i. Each stage is evaluated in turn, with second stage validation using the best-possible network as selected in the first stage.

ii. In the first stage, cross-validation avoid overfitting and guards against weak instruments.

iii. In the second stage, validation will lead to best-possible performance on the objective $\mathbb{E}[y|x,z]$ under the constraint imposed by F_θ from first stage.

iv. Dubbed as Causal AI or Artificial Economic Intelligence.

Implementation in R

i. playground.tensorflow.org is a website where you can tweak and visualize neural networks.

ii. Many packages exist in R for training deep, neural networks such as `mxnet`, `H20`, `keras` etc.

iii. *mx*. functions from `mxnet` package.

iv. See https://www.datacamp.com/community/tutorials/keras-r-deep-learning for an interesting application using image data.

Pseudo Code in R

i. library(mxnet)

ii. Network Structure:

- data <- mx.symbol.Variable("data")
- 1st hidden layer with 128 neurons:
  f_1 <- mx.symbol.FullyConnected(data, name = "f_1", $num_{hidden} = 128$)
- Set the activation which takes output of f_1:
  act_1 <- mx.symbol.Activation(f_1, name = "relu1", act_{type} = "relu")
- second layer takes output from act_1 and then define second activation function and so on:
  f_2 <- mx.symbol.FullyConnected(act_1, name = f_2, num_{hidden} = 64)
- ...
- finally output layer with 10 neurons:
  f_3 <- mx.symbol.FullyConnected(act_2, name = "f_3, $num_{hidden} = 10$)

Pseudo Code in R

i. softmax output unit for multinomial output case:
softmax <- mx.symbol.SoftmaxOutput(f_3, name = "sm")

ii. assign cpu to mxnet
devices <- mx.cpu()

iii. set seed to control the random process in mxnet
mx.set.seed(0)

Pseudo Code in R

Train the Neural Network:

i. model <- mx.model.FeedForward.create(softmax, $X = train_x$, $y = train_y$, ctx = devices, num.round = 10, array.batch.size = 100, learning.rate = 0.07, momentum = 0.9, eval.metric = mx.metric.accuracy, initializer = mx.init.uniform(0.07), epoch.end.callback = mx.callback.log.train.metric(100))

ii. num.round is number of iterations for training, batch size is batch size for SGD. Learning and momentum parameters are also for SGD. "initializer" is initialization scheme for parameters.

iii. eval.metric is the evaluation function applied on the results to monitor performance on the validation set. "accuracy" calculates the mean accuracy per sample for softmax in all dimensions.

iv. Make a prediction: preds <- predict(model, $test_x$)

Pseudo Code in R

i. num.round is number of iterations for training, batch size is batch size for SGD. Learning and momentum parameters are also for SGD.

ii. "initializer" is initialization scheme for parameters.

iii. eval.metric is the evaluation function applied on the results to monitor performance on the validation set. "accuracy" calculates the mean accuracy per sample for softmax in all dimensions.

iv. callback function saves a checkpoint to files during each period iteration. "log.train.metric" logs a training metric each period.

Thank you