Neural network and its training

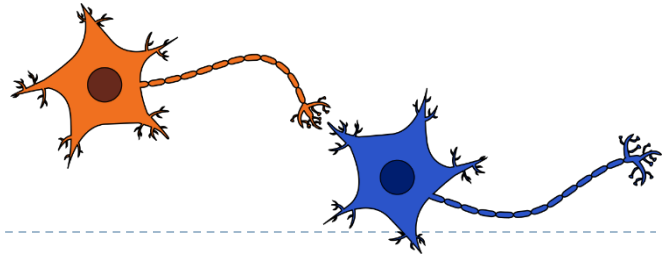
Szu-Chi Chung

Department of Applied Mathematics, National Sun Yat-sen University

The beginning of the story

- Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions
  - This is the logic that sparked artificial neural networks (ANNs)
  - An ANN is a Machine Learning model inspired by the networks of biological neurons found in our brains
- ANNs are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks
  - Such as classifying billions of images (e.g., Google Images)
  - Helping us focus on the applications rather than coding for the functions (e.g. Copilot)
  - Powering speech recognition services (e.g., Apple's Siri)
  - Recommending the best videos to watch to users every day (e.g., YouTube)
  - Learning to beat the world champion at the game of Go (DeepMind's AlphaGo)

The beginning of the story



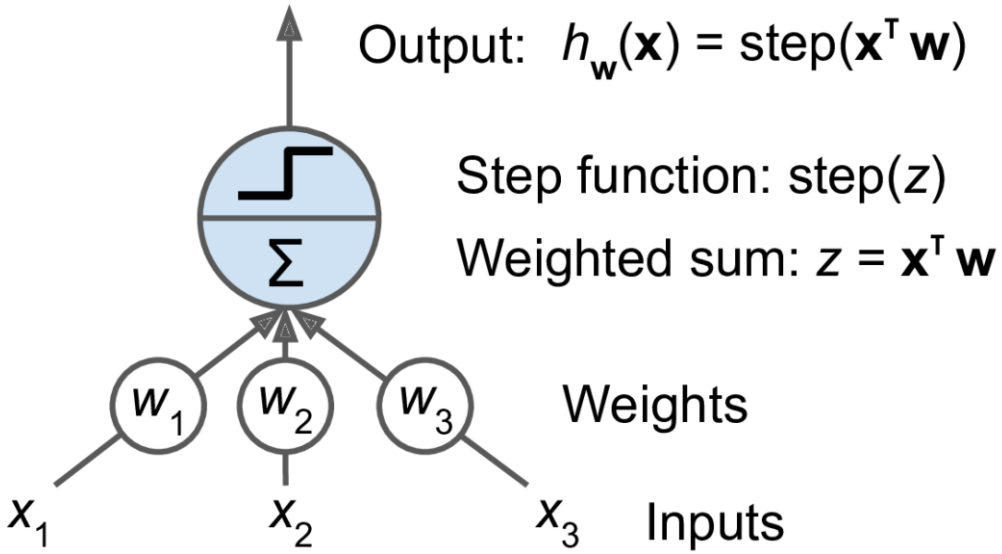
- ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist McCulloch and the mathematician Pitts
  - Neural networks became popular in the 1980s. Then along came SVMs, Random Forests in the 1990s, and Neural Networks took a back seat
- Re-emerged around 2010 as Deep Learning
  - There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems
  - The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time
  - Much of the credit goes to three pioneers and their students: Yann LeCun, Georey Hinton and Yoshua Bengio, who received the 2019 ACM Turing Award for their work in Neural Networks

The Perceptron

- The *threshold logic unit* (TLU) computes a weighted sum of its inputs $\sum_{j=1}^{p} w_{kj} X_j$, then applies a *step (Activation) function* to that sum and output

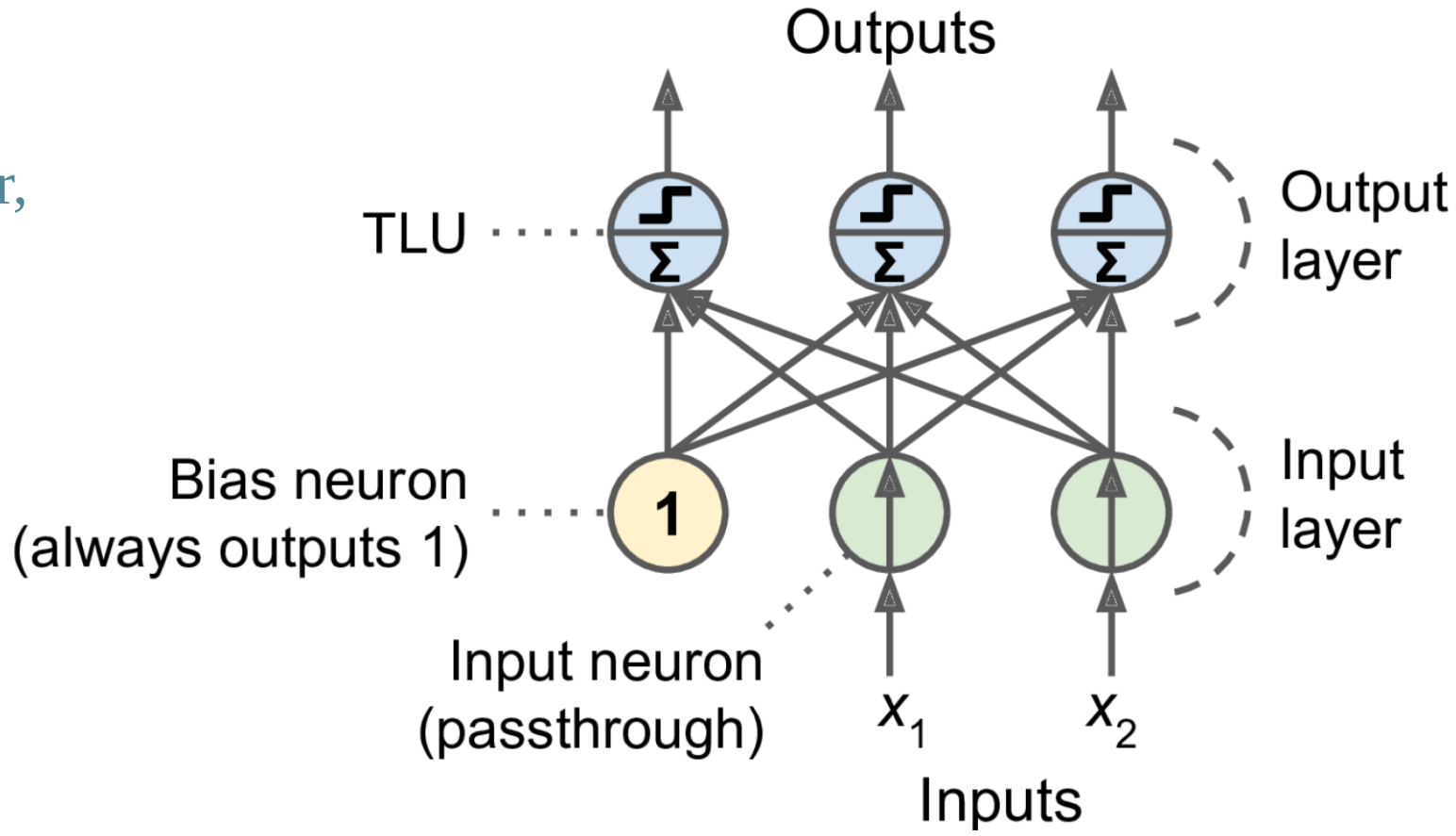
$$heavidide(z) = \begin{cases} 0 \ \ if \ z < 0 \\ 1 \ \ if \ z \geq 0 \end{cases}$$

  - It is an artificial neuron. A single TLU can be used for simple linear binary classification and if the result exceeds a threshold, it outputs positive class



The Perceptron

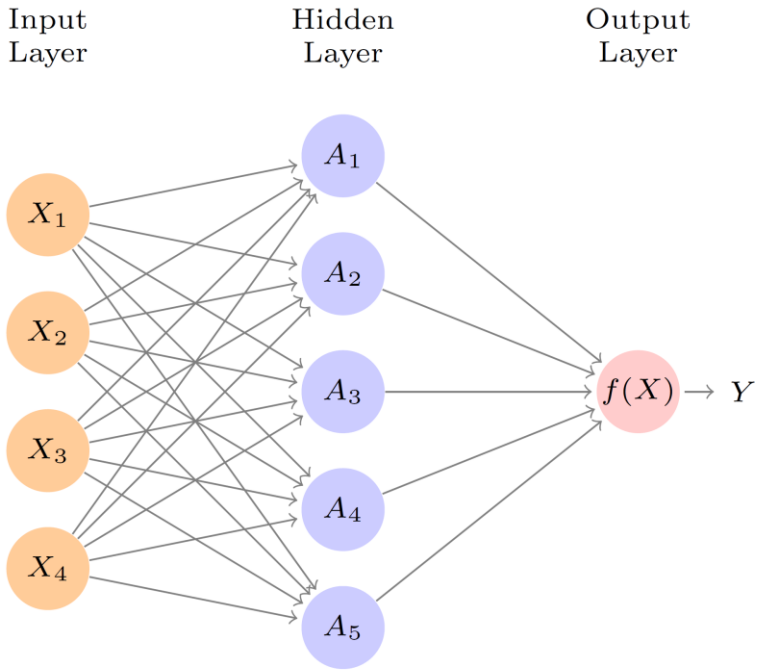
- A Perceptron is composed of a single layer of TLUs, with each TLU connected to all the inputs
  - When all neuron in a layer are connected to every neuron in the previous layer, it is call dense layer
  - The input neuron output whatever input they are fed and form the input layer
  - An extra bias neuron is added which outputs 1 all the time



Neural Network (The Multilayer Perceptron)

- The four features $X_1, \ldots, X_4$ make up the units in the *input layer*
  - Each of the inputs from the input layer feeds into each of the K (5 here) hidden units

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$



Lower layer

Higher (top) layer

Details

- $g(z)$ is called the activation function
  - Activation functions in hidden layers are typically nonlinear, otherwise the model collapses to a linear model
  - A_k are the activations which are different transformation of original features (Like the basis functions in the Generalized additive model)
- Fitting a neural network requires estimating the unknown parameters. For a quantitative response, typically squared-error loss is used

$$\sum_{i=1}^{n}(y_i - f(x_i))^2$$

Details

- The sigmoid activation function was favored in the early age

$$g(z) = \frac{1}{1 + e^{-z}}$$

- The preferred choice in modern neural networks is the ReLU (rectified linear unit) activation function, which takes the form

$$g(z) = (z)_+ = \begin{cases} 0 & if \;\; z < 0 \\ z & otherwise \end{cases}$$

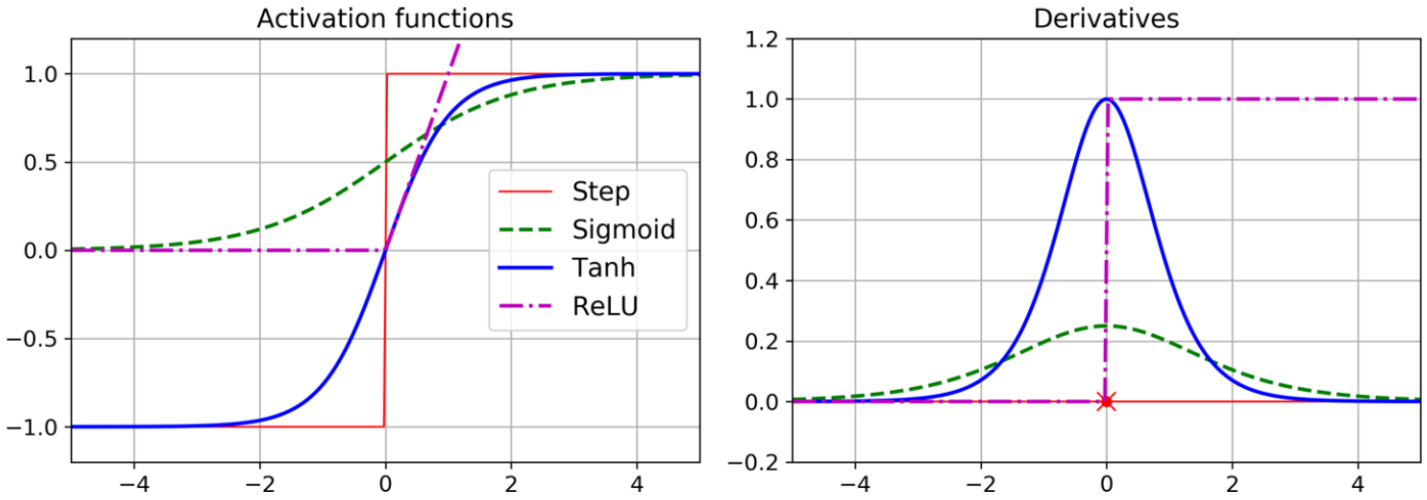


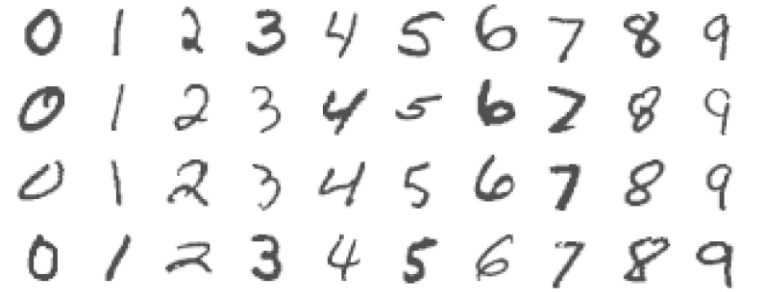
Figure 10-8. Activation functions and their derivatives

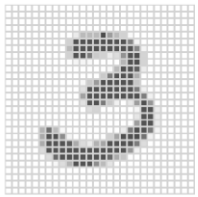
Multilayer Neural Networks

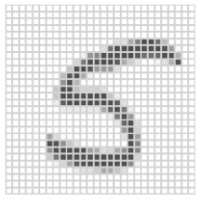
- Modern neural networks typically have more hidden layers, and often many units per layer. In theory, a single hidden layer with a large number of units has the ability to *approximate most functions* (Universal approximation theorem)
- However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size –This is why deep!
- When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN). The field of Deep Learning studies DNNs
  - However, many people talk about Deep Learning whenever neural networks are involved (even shallow ones)

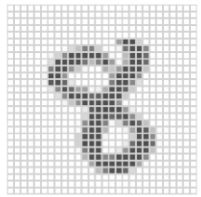
Example: MNIST Digits

- Handwritten digits 28×28 grayscale images with $60K$ train and $10K$ test
  - Features are the 784 pixel grayscale values $\in [0, 255]$
  - Labels are the digit class $0 - 9$ with one-hot encoding
- Goal: build a classifier to predict the image class
  - We build a network with two hidden layer which has 256 units at first layer, 128 units at second layer, and 10 units at output layer. Along with intercepts (called *biases*) there are 235,146 parameters (referred to as *weights*)









Multilayer Neural Networks

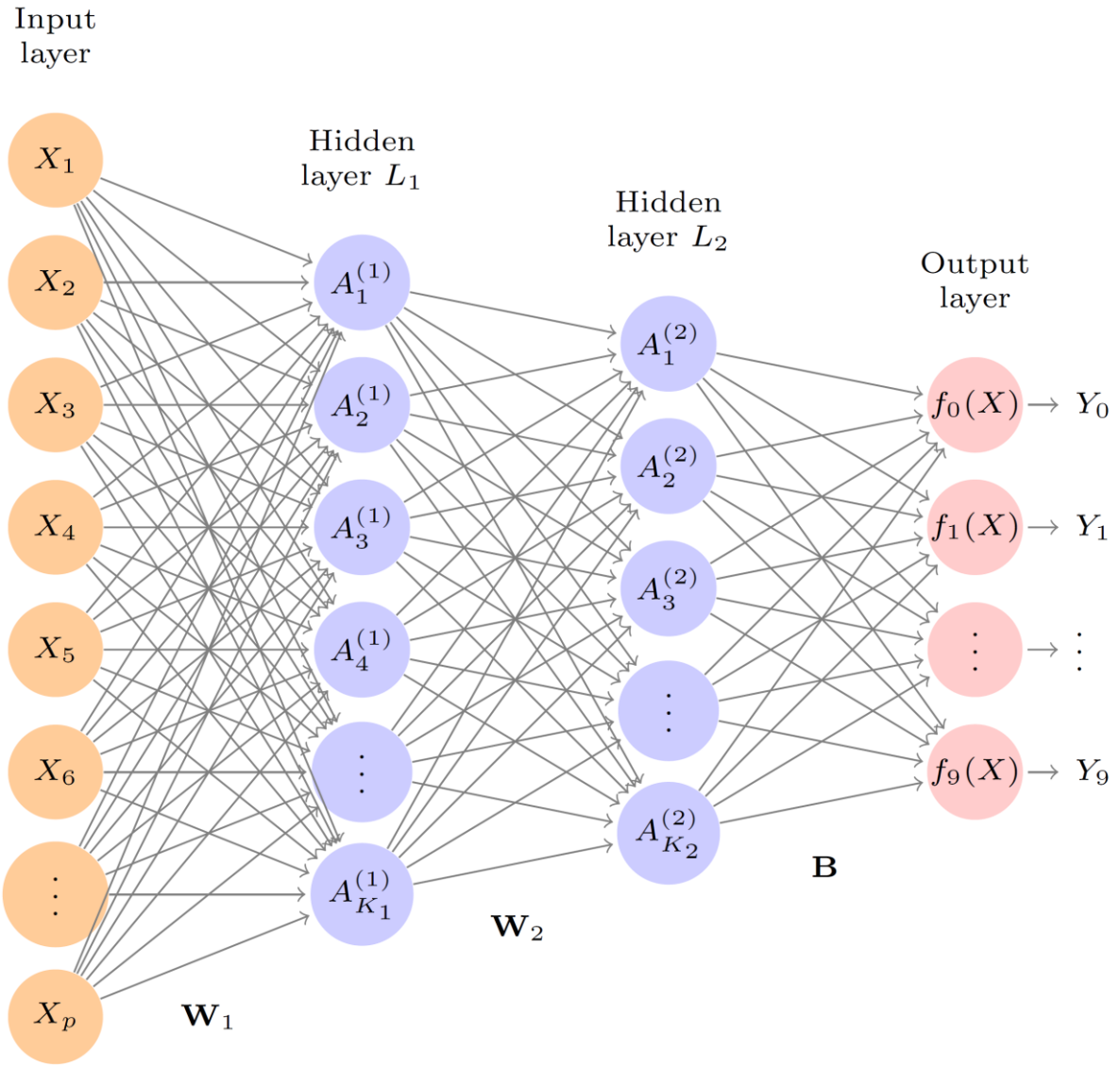
- $A_k^{(1)} = g\left(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j\right)$

for $k = 1, \ldots, K_1$

- $A_l^{(2)} = g\left(w_{l0}^{(2)} + \sum_{k=1}^{K_1} w_{lk}^{(2)} A_k^{(1)}\right)$

for $l = 1, \ldots, K_2$

- The W_1, W_2 and B has $785 \times 256, 257 \times 128$ and 129×10 elements, respectively



Multilayer Neural Networks

- Let $Z_m = \beta_m + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)}, m = 0,1, \ldots, 9$ be 10 linear combinations of activations at second layer
  - Output activation function encodes the softmax function (This ensures that the 10 numbers behave like probabilities (non-negative and sum to one))

$$f_m(X) = \Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}}$$

- We fit the model by minimizing the negative multinomial log-likelihood (or cross-entropy, just like in multinomial logistic regression):

$$-\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log(f_m(x_i))$$

- y_{im} is 1 if true class for observation i is m, else 0 – i.e. one-hot encoded.

Results

- Early success for neural networks in the 1990s
- There are almost four times as many coefficients in the neural network model!
  - With so many parameters, regularization is essential
- Some details of regularization and fitting will come later
  - Best reported rates are $< 0.5\%$!
  - Human error rate is reported to be around 0.2%, or 20 of the 10K test images

Method	Test Error
Neural Network + Ridge Regularization	2.3%
Neural Network + Dropout Regularization	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

Fitting a Neural Network

- This problem is difficult because the objective is nonconvex, for a neural network with one hidden layer we have

$$\min_{\{w_k\}_1^K,\beta} \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2$$

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\right)$$

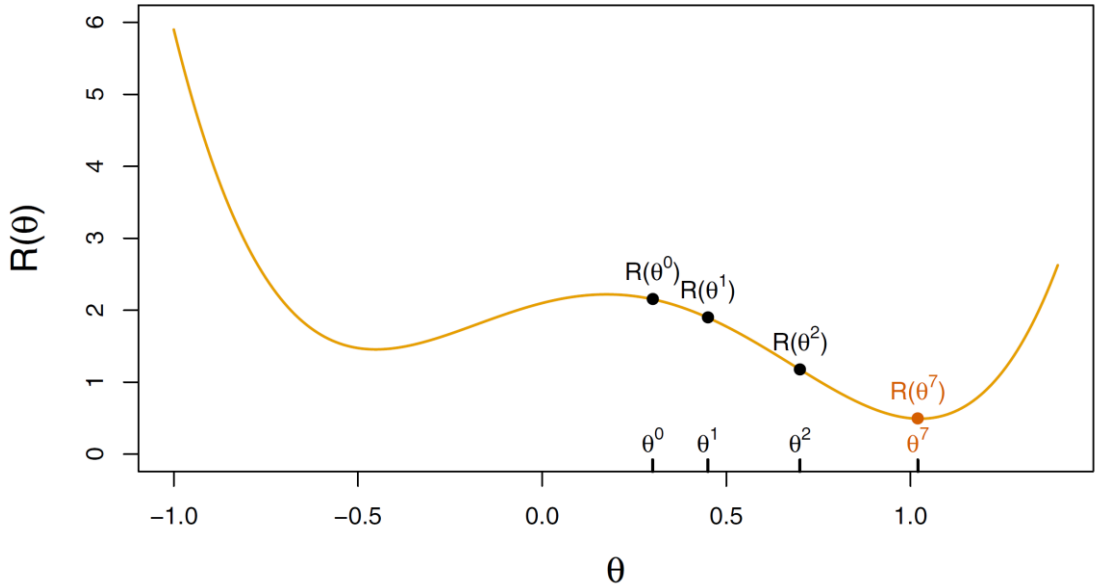
- Suppose we represent all the parameters in one long vector θ, $\mathrm{R}(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$
- Slow learning and regularization are the keys for the successful of training

Fitting a Neural Network

1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$
2. Iterate until the objective fails to decrease:

   (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e. such that $R(\theta^{t+1}) < R(\theta^t)$

   (b) Set $t \leftarrow t + 1$



1. Slow learning - Gradient descent

- How to find a direction δ that points downhill? We compute the gradient vector

$$\nabla R(\theta^t) = \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta^t}$$

- The gradient points uphill, so our update is $\delta = -\eta \nabla R(\theta^t)$

$$\theta^{t+1} \leftarrow \theta^t - \eta \nabla R(\theta^t)$$

Where η is the *learning rate* which is typically small

- $R(\theta) = \sum_{i=1}^{n} R_i(\theta)$ is a sum, so gradient is sum of gradients
- For a small enough value of the learning rate η, this step will decrease the objective
- If the gradient vector is zero, then we may have arrived at a minimum of the objective

Gradients and Backpropagation

- $R_i(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2 = \frac{1}{2}\left(y_i - \beta_0 + \sum_{k=1}^{K}\beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\right)\right)^2$
- Let $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}$
- Backpropagation uses the chain rule for differentiation:

$$\frac{\partial R_i(\theta)}{\partial \beta_k} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} = -(y_i - f(x_i)) \cdot g(z_{ik})$$

$$\frac{\partial R_i(\theta)}{\partial w_{kj}} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} = -(y_i - f(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}$$

  - Notice that both these expressions contain the residual. So the act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule
  - In just two passes through the network (one forward, one backward), it can find out how each connection weight and each bias term should be tweaked in order to reduce the error

Gradients and Backpropagation

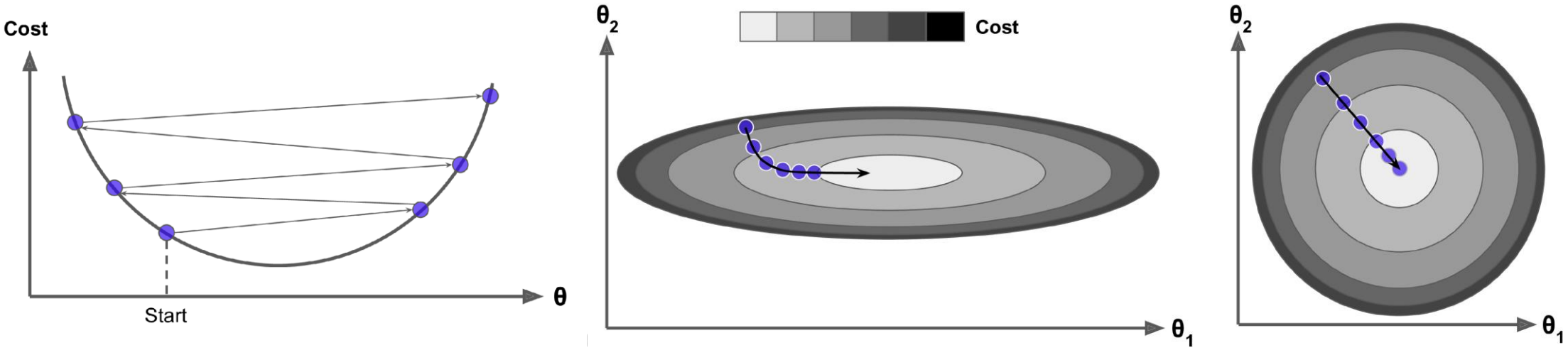
- The *forward pass* computes the output of all the neurons in current layer and send it to next layer. It is exactly like making predictions, except all intermediate results are *preserved* since they are needed for the backward pass

- The *backward pass* measures how much of output error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network

- The algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed

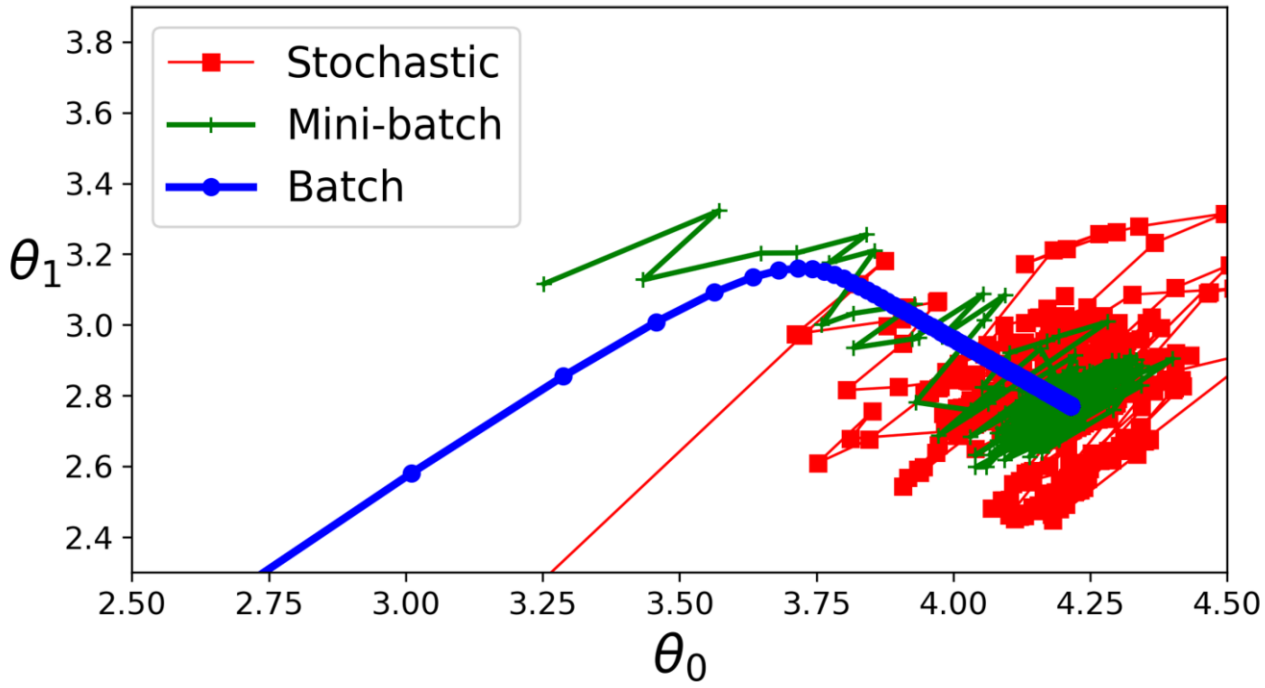
Gradient descent

- Some notice
  - Determine the step η which is call the *learning rate* is important
  - It is important to ensure that features have similar scale



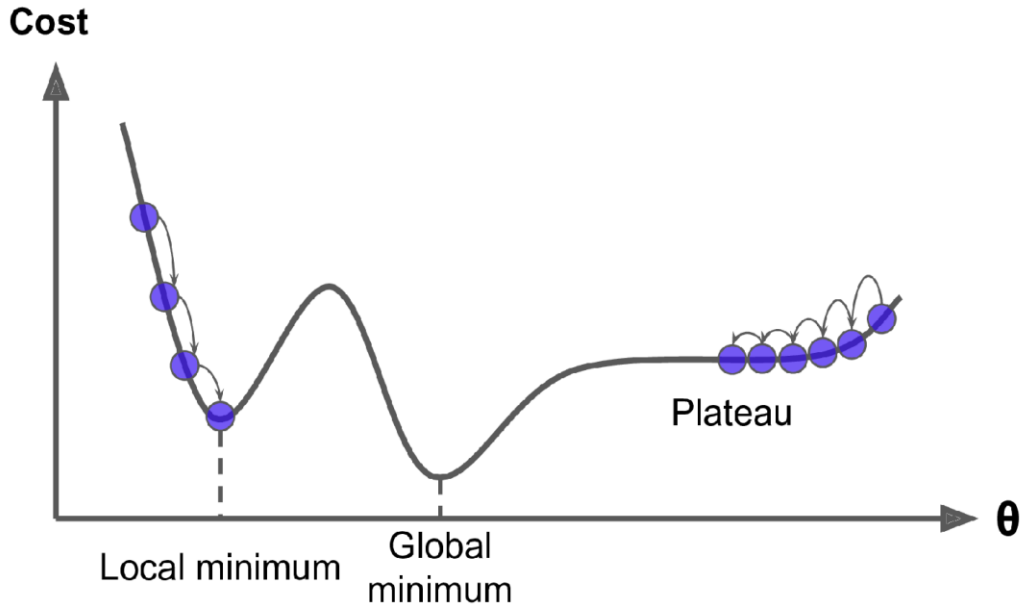
Gradient descent

- The Gradient Descent uses the whole training set to compute the gradients at every step $\sum_{i=1}^{n} R_i(\theta)$, which makes it very slow
  - *Stochastic Gradient Descent (SGD)* picks a random instance in the training set at every step and computes the gradients based only on it. Due to its stochastic nature, it is much less regular than Gradient Descent
  - *Mini-batch Gradient Descent or again SGD* instead computes the gradients on small random sets of instances. The main advantage of Mini-batch GD is that you can get a performance boost from hardware optimization of matrix operations using GPUs



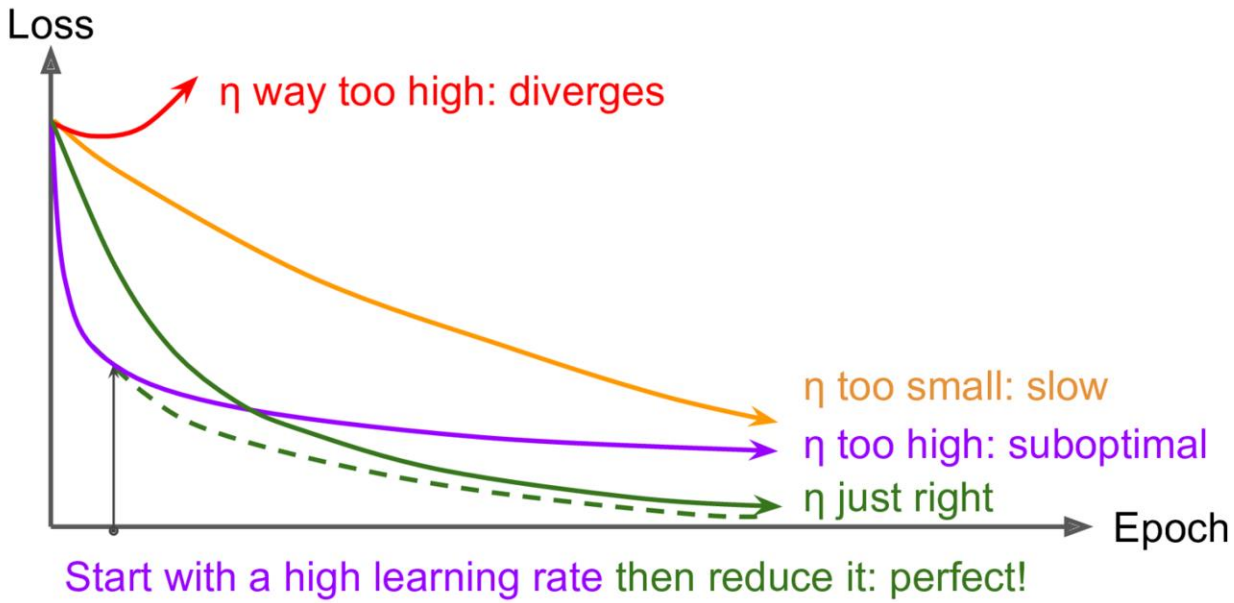
Gradient descent

- Randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum
  - One solution to this dilemma is to gradually reduce the learning rate
  - The steps start out large, then get smaller and smaller, allowing the algorithm to settle at the global minimum. The function that determines the learning rate at each iteration is called the *learning schedule*.



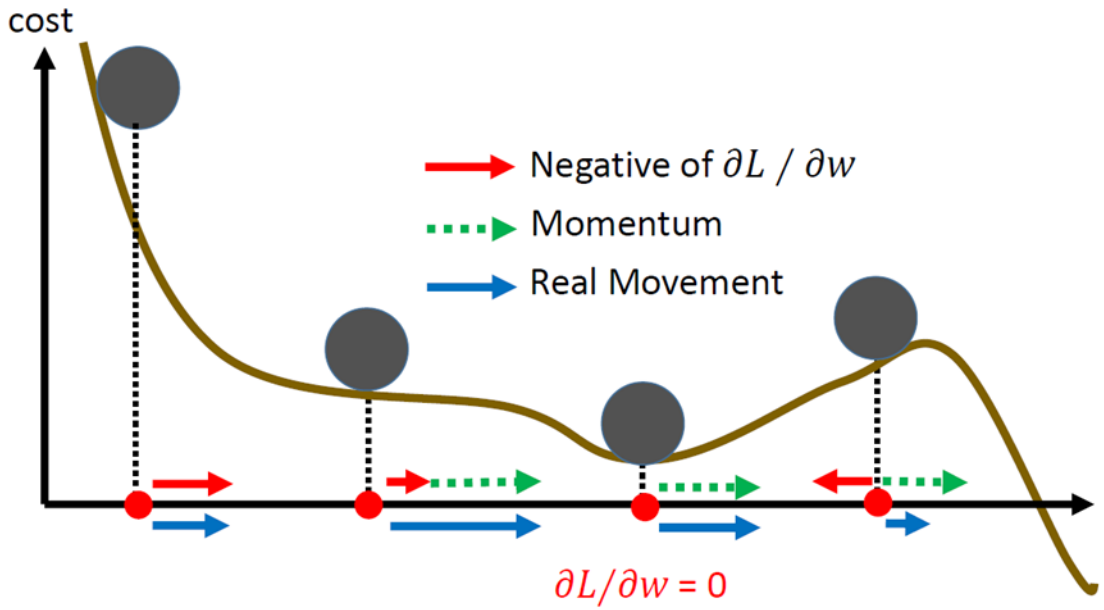
Learning rate scheduling

- Rather than constant, finding a good learning rate is very important
  - Power/exponential scheduling: Set the learning rate to a function of the iteration number t, for instance $\eta(t) = \eta_0 \cdot 0.1^{t/s}$
  - Piecewise constant scheduling: Use a constant learning rate for a number of iterations, then a smaller learning rate for another number of iterations…
  - Performance scheduling: Measure the validation error every N steps, and reduce the learning rate by a factor of C when the error stops dropping
  - 1cycle scheduling



Gradient descent

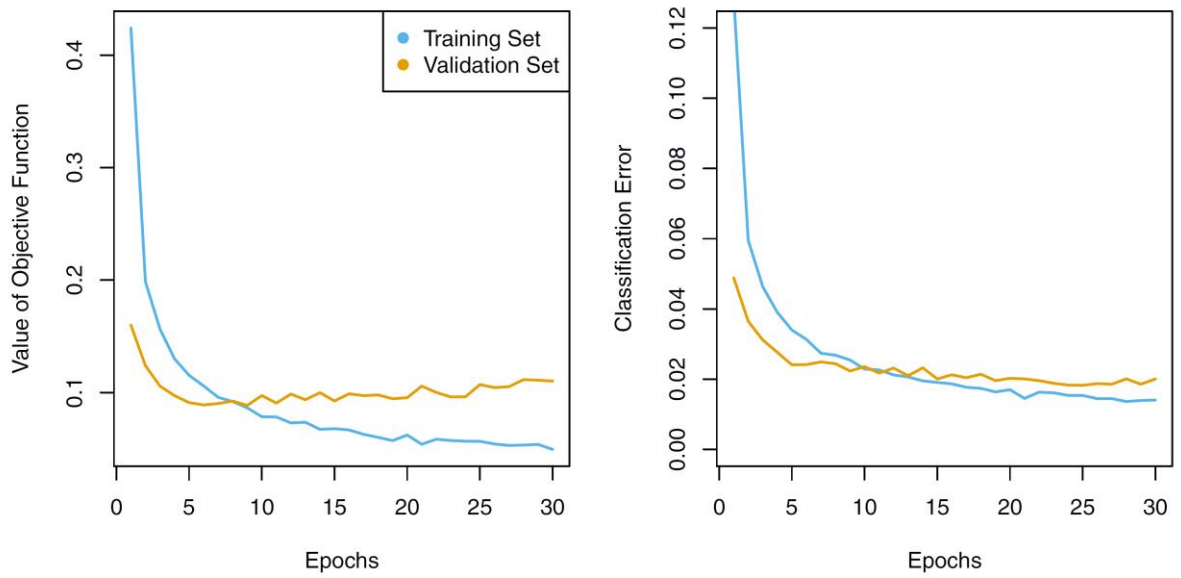
- Training a very large deep neural network can be painfully slow, many faster optimizers are proposed based on the idea of *Momentum*
  - $\theta_i^{t+1} = \theta_i^t + v_i^t$, where $v_i^t = \beta v_i^{t-1} - \eta \nabla R(\theta_i^t)$, $v_i^0 = 0$ and β (set to 0.9) is momentum
  - More optimizer see here



https://speech.ee.ntu.edu.tw/~tlkagk/courses/ML2020/Optimization.pdf

Tricks of the Trade

- The MNIST problem
  - Slow Learning. the model is fit using gradient descent. The fitting process is then stopped when overfitting is detected (With early stopping)
  - *Stochastic gradient descent*. Use a small minibatch drawn at random at each step. E.g. for MNIST data, we use minibatches of 128 observations
  - 20% of the 60,000 training observations were used as a validation set
  - An *epoch* is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e. $48K/128 = 375$ *steps* per epoch for MNIST



2. Regularization

- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy

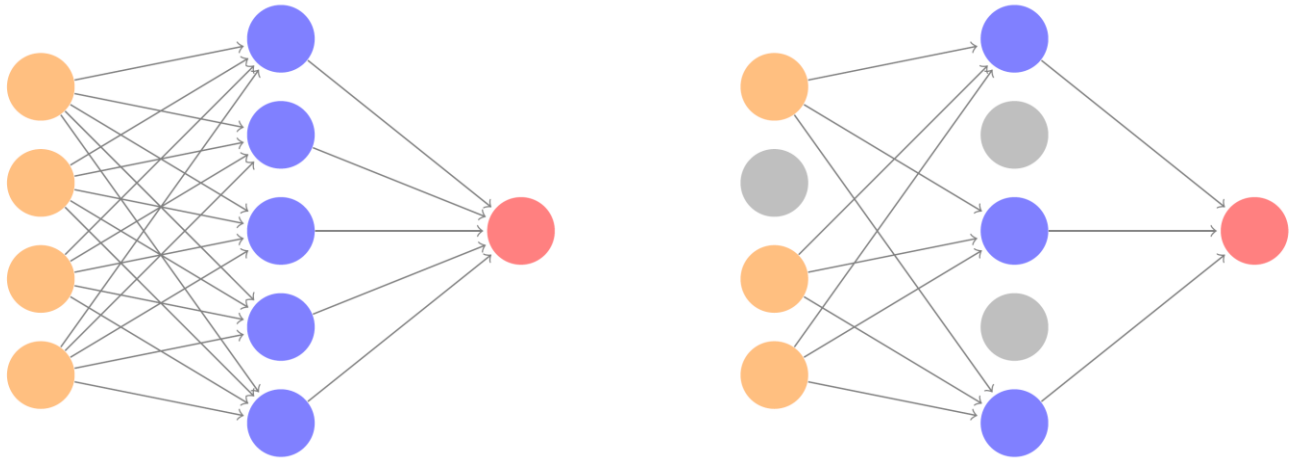
- Regularization

$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i)) + \lambda\sum_{j}\theta_j^2$$

  - If you want a sparse model (with many weights equal to 0), you can use l_1

Dropout Learning

- Similar to randomly omitting variables when growing trees in random forests
  - At each SGD step, randomly remove units with probability Φ, the surviving units stand in for those missing, and their output weights are scaled up by a factor $1/(1-\Phi)$ to compensate during training or we can multiply each input connection weight by the $(1-\Phi)$ after training
  - We do not perform dropout at the output layer or after the training is done, neurons don't get dropped anymore. In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer)

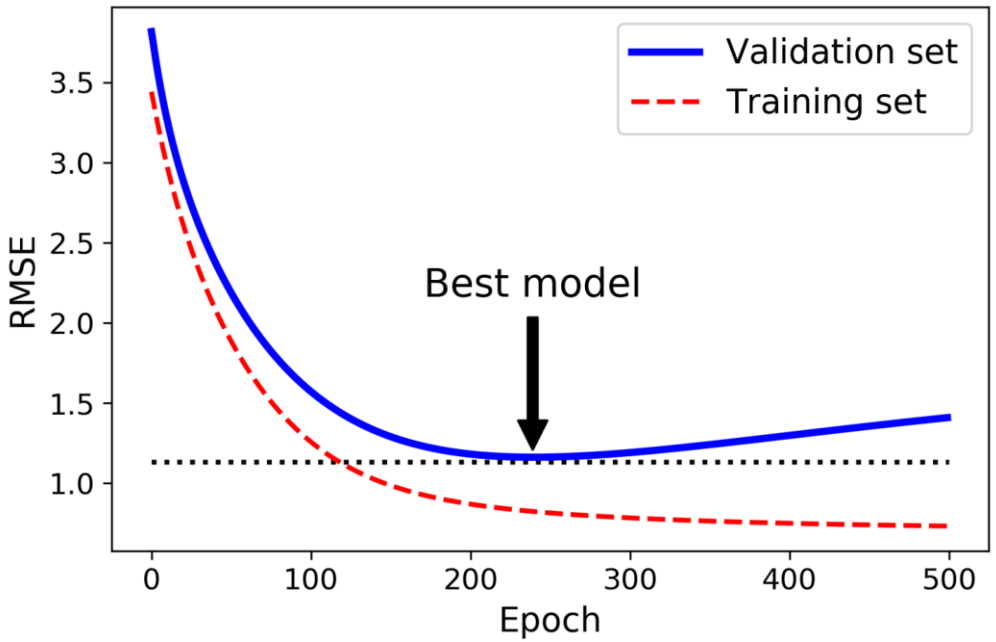


Dropout Learning

- Neurons trained with dropout have to be as useful as possible on their own. They also cannot rely on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs so you get a more robust network that generalizes better
  - If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set
  - Since each neuron can be either present or absent, there are a total of 2^n possible networks after n steps, The resulting neural network can be seen as an averaging ensemble of part of these smaller neural networks

Early stopping

- Another way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called early stopping
  - With early stopping you just stop training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch.”

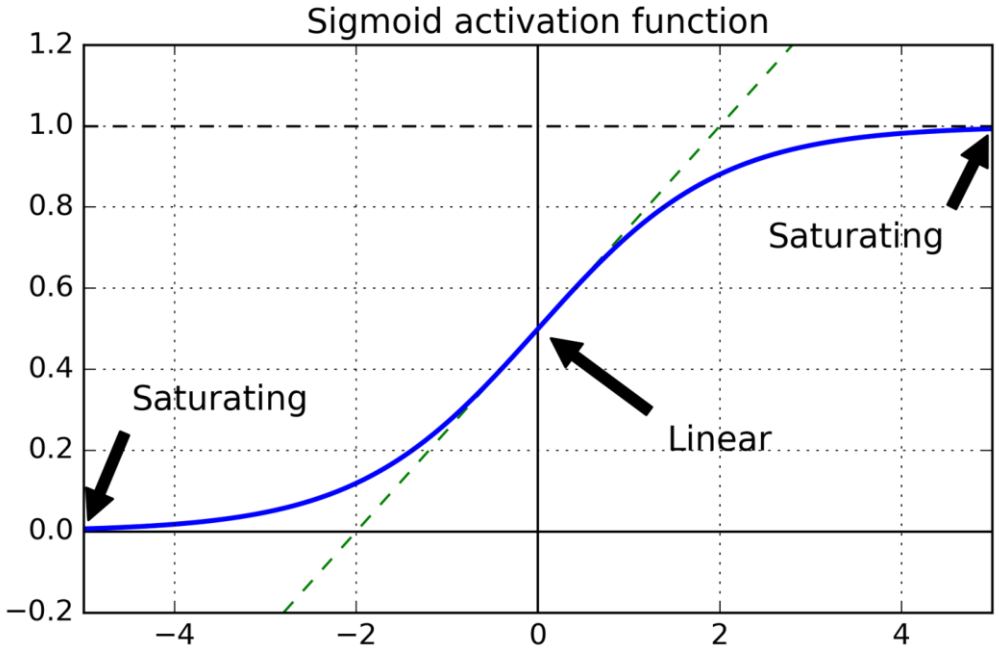


3. The Vanishing/Exploding Gradients Problems

- You may be faced with vanishing/exploding gradients. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training
  - The cumulative errors that occur in sequential transmission over a noisy channel
  - $y = f4\left(f3\left(f2(f1(x))\right)\right)$
  - To adjust the parameters of each function in the chain based on the error recorded on the output of $f4$ (the loss of the model). To adjust $f1$, you'll need to percolate error information through $f2$, $f3$, and $f4$. However, each successive function in the chain introduces some amount of noise. If your function chain is too deep, this noise starts overwhelming gradient information, and backpropagation stops working!

The Vanishing/Exploding Gradients Problems

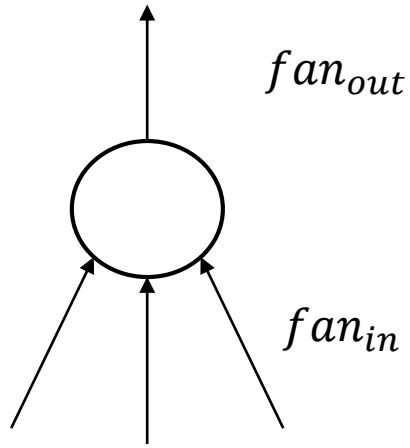
- The combination of logistic sigmoid activation function and the weight initialization (i.e., a normal distribution with a mean of 0 and a standard deviation of 1) cause problem
  - With this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates
  - The function saturates at 0 or 1, with a derivative extremely close to 0!



The Vanishing/Exploding Gradients Problems

- We need the variance of the outputs of each layer to be equal to the variance of its inputs, and we need the gradients to have equal variance before and after flowing through a layer in the reverse direction
  - Let $fan_{avg} = \frac{fan_{in}+fan_{out}}{2}, r = \sqrt{3\sigma^2}$
  - Initialize weight with normal distribution (mean 0, variance σ^2) or uniform distribution between $-r$ and r

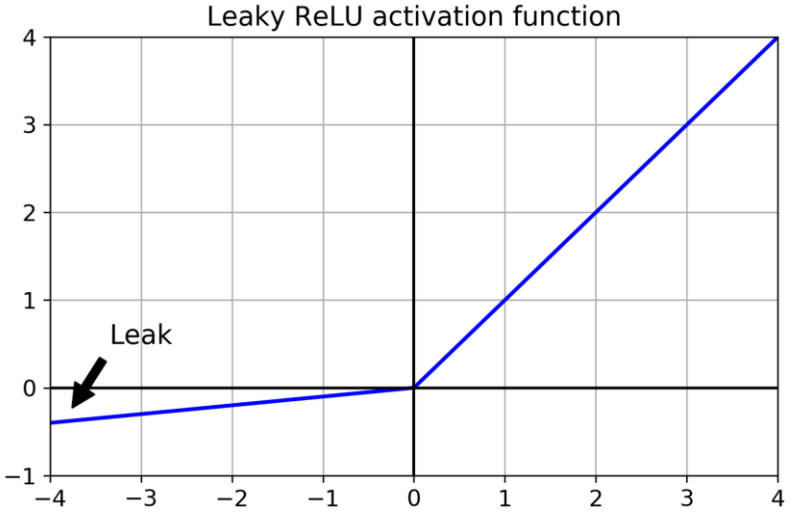
Initialization	Activation functions	σ^2
Glorot (Xavier)	None, tanh, sigmoid, softmax	$1/fan_{avg}$
He	ReLu and its variants	$2/fan_{in}$
Lecun	SELU	$1/fan_{in}$



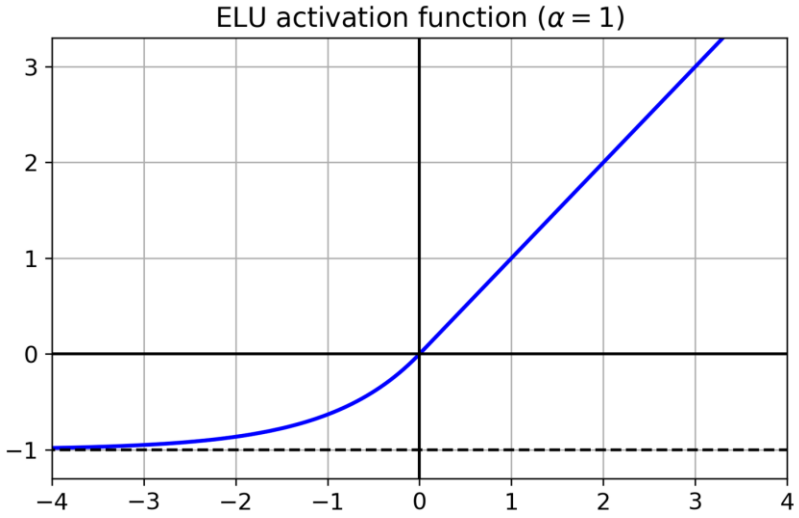
Nonsaturating Activation Functions

- ReLU activation function does not saturate for positive values but
  - It suffers from a problem known as the dying ReLUs: during training, some neurons stop outputting anything other than 0 and Gradient Descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative
  - You may want to use a variant of the ReLU function, such as the *leaky ReLU or exponential linear unit (ELU)*

$$\max(\alpha z, z)$$



$$\begin{cases} \alpha(e^z - 1) & if\ z < 0 \\ z & if\ z \geq 0 \end{cases}$$



Batch normalization

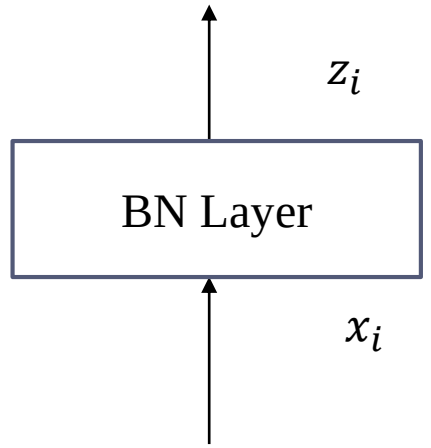
- Adding an operation in the model before or after the activation function of each hidden layer
  - This operation zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting
  - The operation lets the model learn the optimal scale and mean of each of the layer's inputs using current mini-batch (γ and β are learned through regular backpropagation)

$$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x_i$$

$$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}(x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$z_i = \gamma \cdot \hat{x}_i + \beta$$



Batch normalization

- Batch Normalization estimate μ and σ for testing (to replace μ_B and σ_B) during training by using a moving average of the layer's input means and standard deviations (Totally four parameters per layer if we consider γ and β)
  - $\hat{\mu} = \hat{\mu} \times \text{momentum} + \mu(1-\text{momentum})$
- Batch Normalization significantly speed up the training and avoid the use of regularization
  - The main effect of batch normalization appears to be that it helps with gradient propagation by feature normalization and thus allows for deeper networks
  - However, it adds some complexity and there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it's often possible to fuse the BN layer with the previous layer, after training

Why deep?

- If you want to make a complex system simpler, there's a recipe you can apply: just structure it into modules, organize the modules into a hierarchy, and start reusing the same modules in multiple places as appropriate
  - If you're a software engineer, you're already keenly familiar with these principles: an effective codebase is one that is modular, hierarchical, and where you don't re-implement the same thing twice, but instead rely on reusable classes and functions
  - Deep learning itself is simply the application of this recipe to continuous optimization via gradient descent: you take a classic optimization technique (gradient descent over a continuous function space), and you structure the search space into modules (layers), organized into a deep hierarchy, where you reuse whatever you can (for instance, convolutions are all about reusing the same information in different spatial locations)
  - Deeper hierarchies are intrinsically good because they encourage feature reuse

Why deep?

- For complex problems, deep networks have a much higher *parameter efficiency* than shallow ones
  - Model complex functions using exponentially fewer neurons than shallow nets

- Real-world data is often structured in such a hierarchical way, and deep neural networks take advantage of this fact
  - Lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces)
  - It can also generalize better by keeping the weight of lower layer and perform transfer learning

How to choose the network architectures?

- Number of Hidden Layers
  - You can ramp up the number of hidden layers until you start overfitting the training set
- Number of Neurons per Hidden Layer
  - The number of neurons in the input and output layers is determined by the type of input and output your task requires
  - For the hidden layers, using the same number of neurons in all hidden layers so that there is only one hyperparameter to tune
  - Another common practice to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low level features can coalesce into far fewer high-level features
- It's simpler and more efficient to pick a model with more layers and neurons than you need, then use regularization techniques to prevent it from overfitting

How to choose the hyperparameters?

- Learning rate
  - Find optimal learning rate empirically or use learning rate scheduling
- Number of iterations
  - Just use enough iterations and use early stopping
- Batch size
  - The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently
- Activation function
  - In general, the variants of ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task

Architecture

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

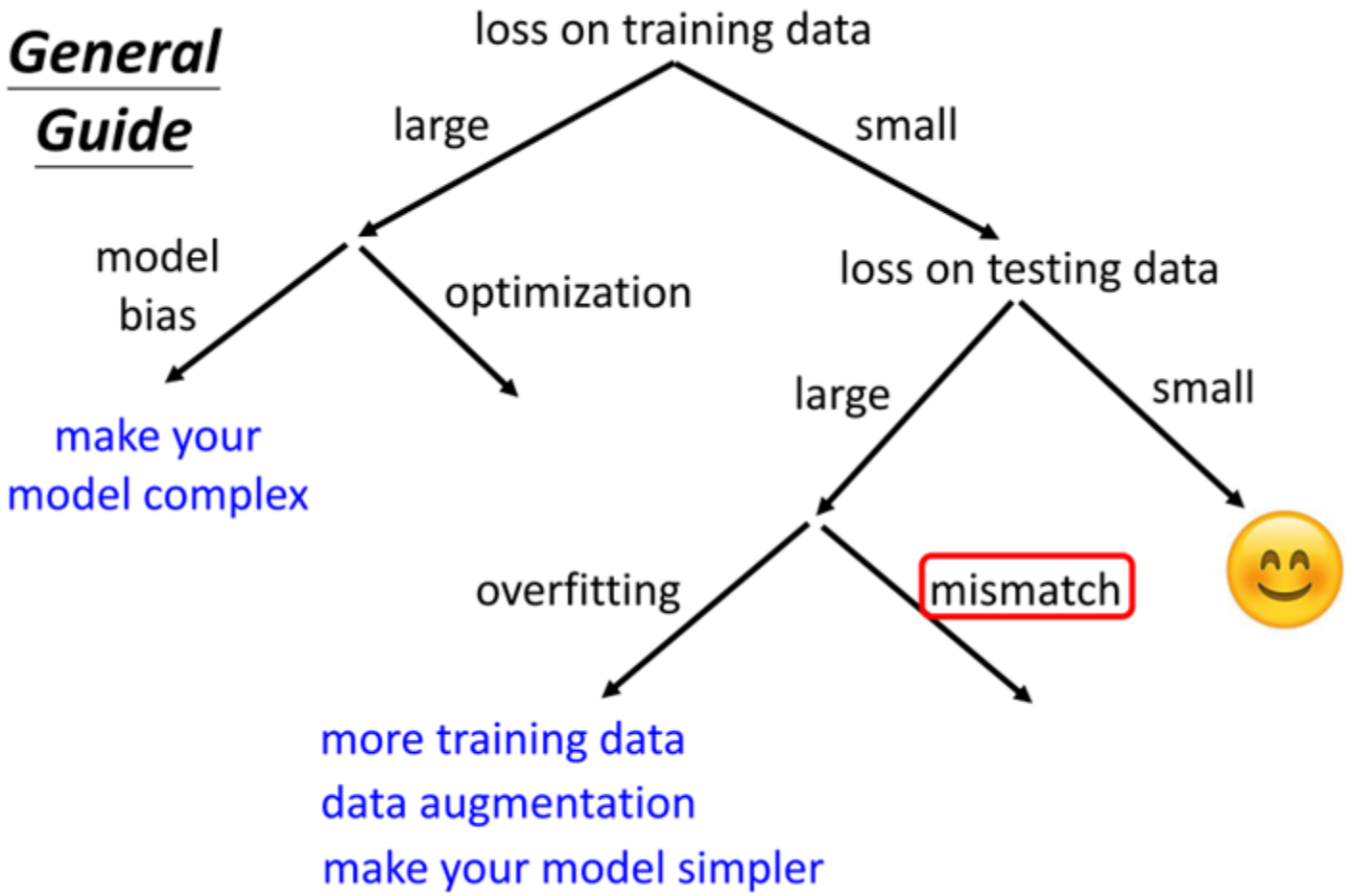
Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Hyperparameters

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle



https://speech.ee.ntu.edu.tw/~hylee/ml/ml2021-course-data/overfit-v6.pdf

When to Use Deep Learning?

- On a dataset with 263 baseball player and 19 features where we would like to predict their salary base on the performance of previous year
  - A linear model was used to fit the training data, and make predictions on the test data
  - The same linear model was fit with lasso regularization. The tuning parameter was selected by 10-fold cross-validation on the training data. It selected a model with 12 variables having nonzero coefficients
  - A neural network with one hidden layer was fit to the data. This model has 1,409 parameters

Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1409	257.4	0.54

When to Use Deep Learning?

- When faced with new data modeling and prediction problems, its tempting to always go for the trendy new methods. Especially when the datasets are very large and can support the fitting of high-dimensional nonlinear models
  - If we can produce models with the simpler tools that perform as well, they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches. Wherever possible, it makes sense to try the simpler models as well, and then make a choice based on the performance/complexity tradeoff
  - Typically we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority

References

[1] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition Chapter10~11

[2] An Introduction to Statistical Learning with Applications in R. Second Edition Chapter 10

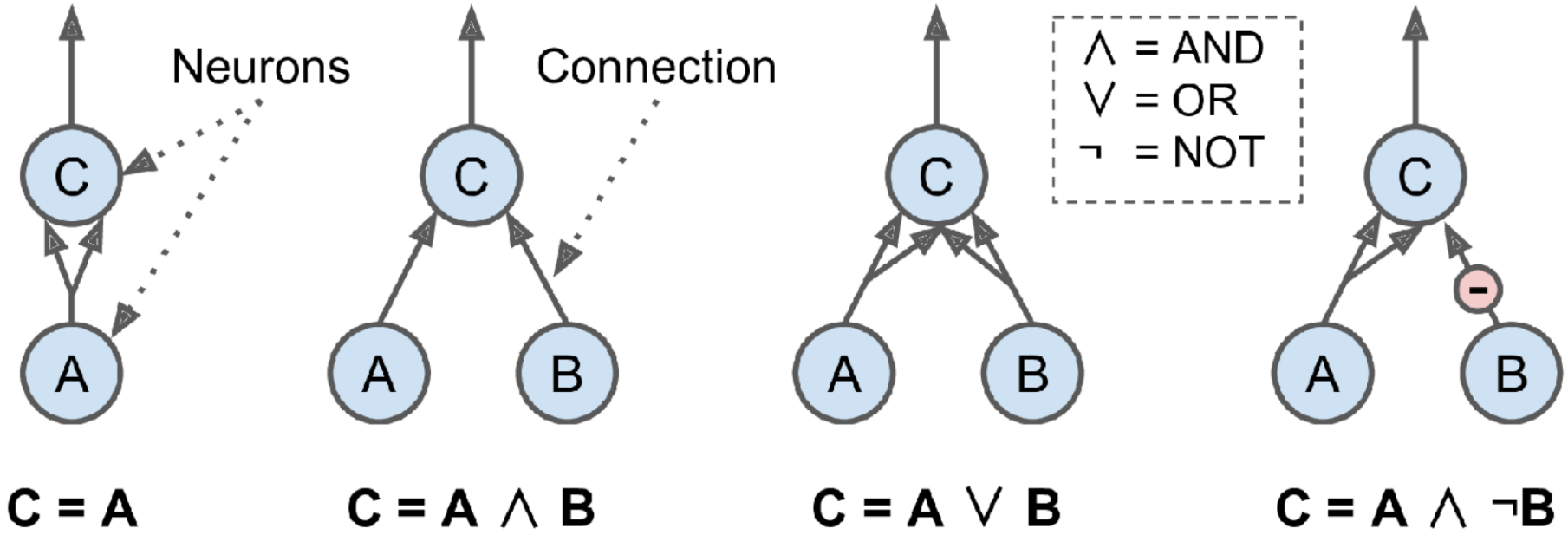
Appendix

Resources

- Deep learning libraries
  - Tensorflow
  - Pytorch
  - Keras
  - https://github.com/krzjoa/awesome-python-data-science#deep-learning
- Playground
  - http://playground.tensorflow.org/
  - http://tfmeter.icsi.berkeley.edu/
- Logger for deep learning
  - https://github.com/microsoft/tensorwatch
  - https://github.com/wandb/client

Logical Computations with Neurons

- Let's build a few ANNs that perform various logical computations, assuming that a neuron is activated when at least two of its inputs are active
  - You can imagine how these networks can be combined to compute complex logical expressions



Faster Optimizers

- All the optimization techniques discussed so far only rely on the first-order partial derivatives (Jacobians). The optimization literature also contains algorithms based on the second-order partial derivatives
  - Since DNNs typically have tens of thousands of parameters, the second order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

Other regularization techniques

- Max-Norm Regularization
  - Max-norm regularization does not add a regularization loss term to the overall loss function. Instead, it is typically implemented by computing $|w|$ after each training step and rescaling w if needed $w = \frac{w \times r}{|w|}$

Other regularization techniques - Monte Carlo (MC) Dropout

- Established a profound connection between dropout networks and approximate Bayesian inference, giving dropout a solid mathematical justification
  - It may boost the performance of any trained dropout model, without having to retrain it or even modify it at all. It just take the average of dropout preictions!
- MC Dropout is a fantastic technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer

- The number of Monte Carlo samples you is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates will be. Try to find the right trade-off between latency and accuracy, depending on your application

Nonsaturating Activation Functions

- Scaled ELU (SELU) activation function ensures certain network will be self-normalized: the output of each layer tend to preserve a mean of 0 and standard deviation of 1 during training
  - The input features must be standardized (mean 0 and standard deviation 1)
  - Every hidden layer's weights must be initialized with LeCun normal initialization
  - The network's architecture must be sequential and contains only dense layer
- In general, the performance of SELU and ELU and will performs better than traditional activation function
- If the speed is the first priority, try using leaky ReLU and its variants
  - If you want to regularize a self-normalizing network based on the SELU activation function (as discussed later), you should use alpha dropout: this is a variant of dropout that preserves the mean and standard deviation of its inputs

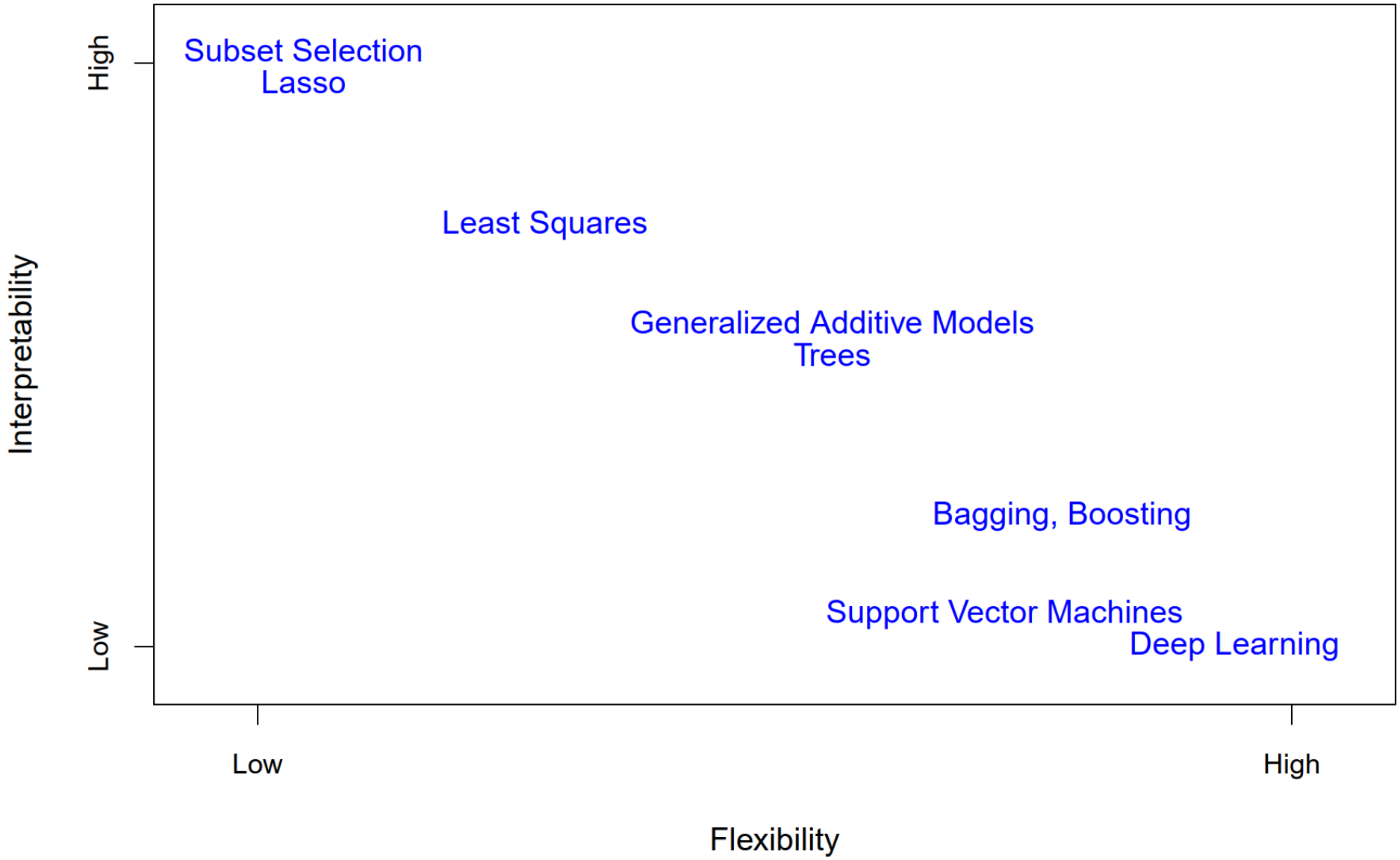
Hyperparameters for self-normalizing net

Table 11-4. DNN configuration for a self-normalizing net

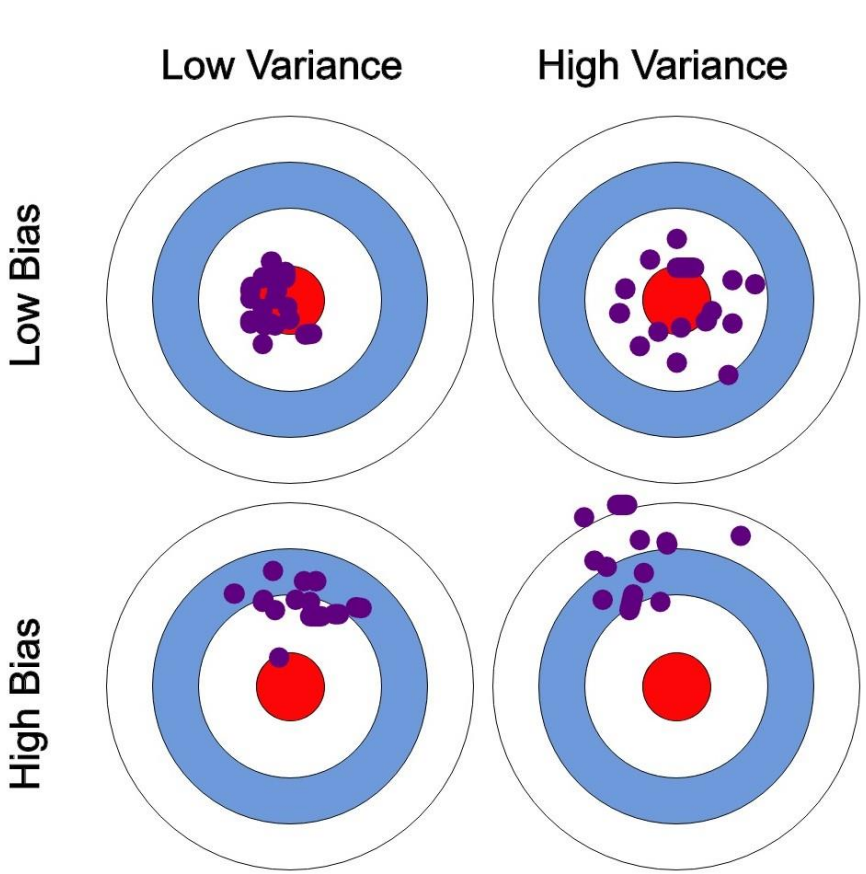
Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

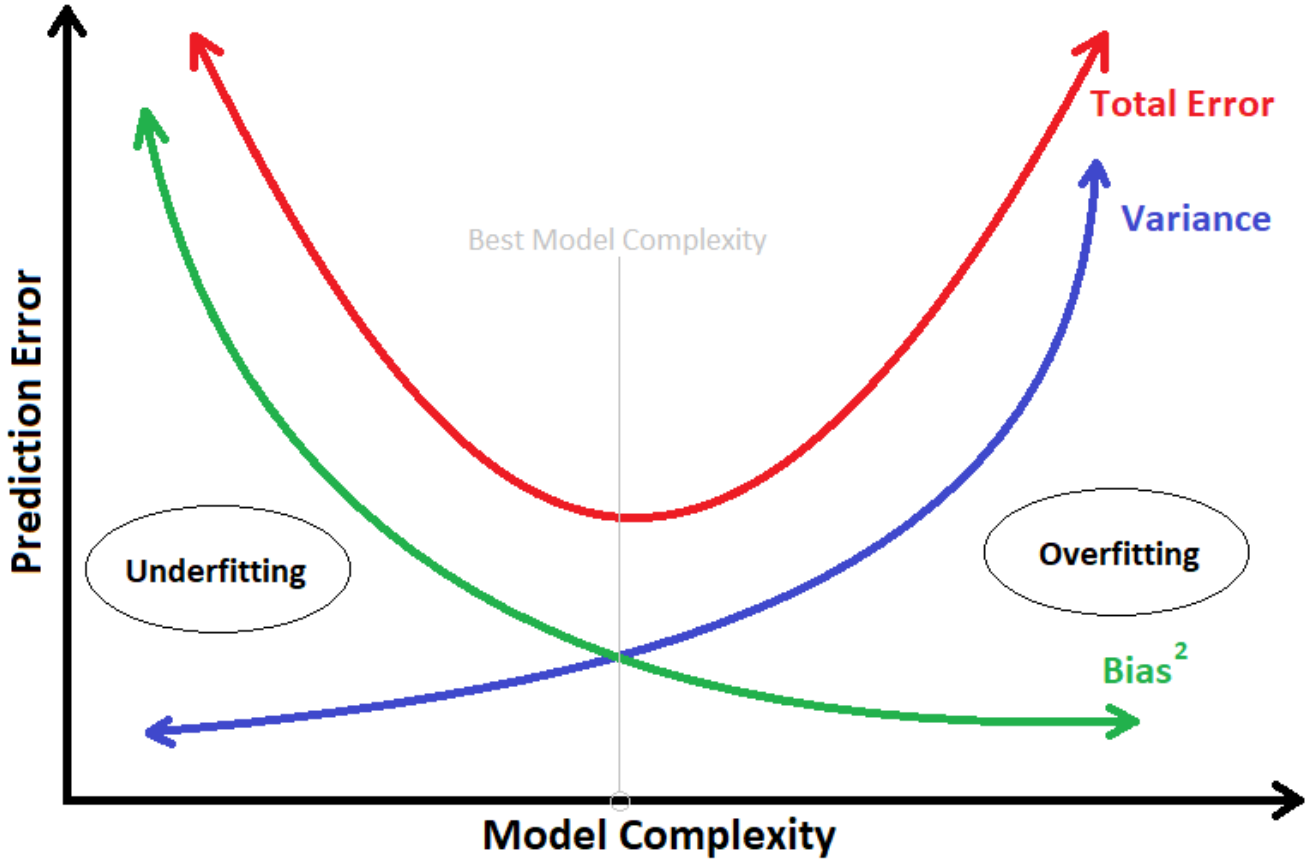
Gradient clipping

- Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold
  - If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try both clipping by value and clipping by norm, with different thresholds, and see which option performs best on the validation set
  - The difference between clipping by value and by norm
    - [0.9, 100.0] will clip to [0.9, 1.0] or [0.00899964, 0.9999595]


Subset Selection
Lasso
Least Squares
Generalized Additive Models
Trees
Bagging, Boosting
Support Vector Machines
Deep Learning
High
Low
Interpretability
Low
High
Flexibility

Bias-variance trade-off

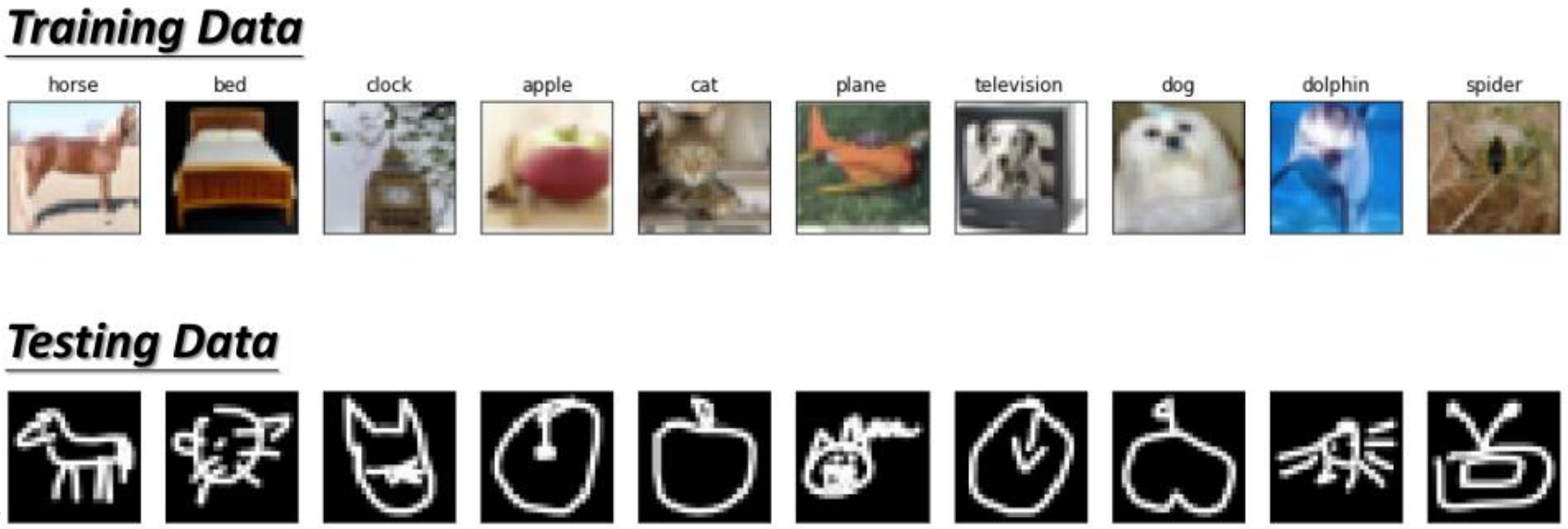




https://jason-chen-1992.weebly.com/home/-bias-variance-tradeoff

Data Mismatch

- Sometimes the data probably won't be perfectly representative of the data that will be used in production
  - The most important rule to remember is that the validation set and the test set must be as representative as possible of the data you expect to use in production, so they should be composed exclusively of representative pictures: you can shuffle them and put half in the validation set and half in the test set



https://speech.ee.ntu.edu.tw/~hylee/ml/ml2021-course-data/overfit-v6.pdf

When to Use Machine/Deep Learning

- To summarize, Machine/Deep Learning is great for:
  1. Problems for which existing solutions require a lot of fine-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better than the traditional approach
  2. Complex problems for which using a traditional approach yields no good solution: the best Machine Learning techniques can perhaps find a solution
  3. Fluctuating environments: a Machine Learning system can adapt to new data
  4. Getting insights about complex problems and large amounts of data