CS286 AI for Science and Engineering

Lecture 6: Introduction to Artificial Neural Networks

Jie Zheng (郑杰)

PhD, Associate Professor

School of Information Science and Technology (SIST), ShanghaiTech University

Fall Semester, 2020

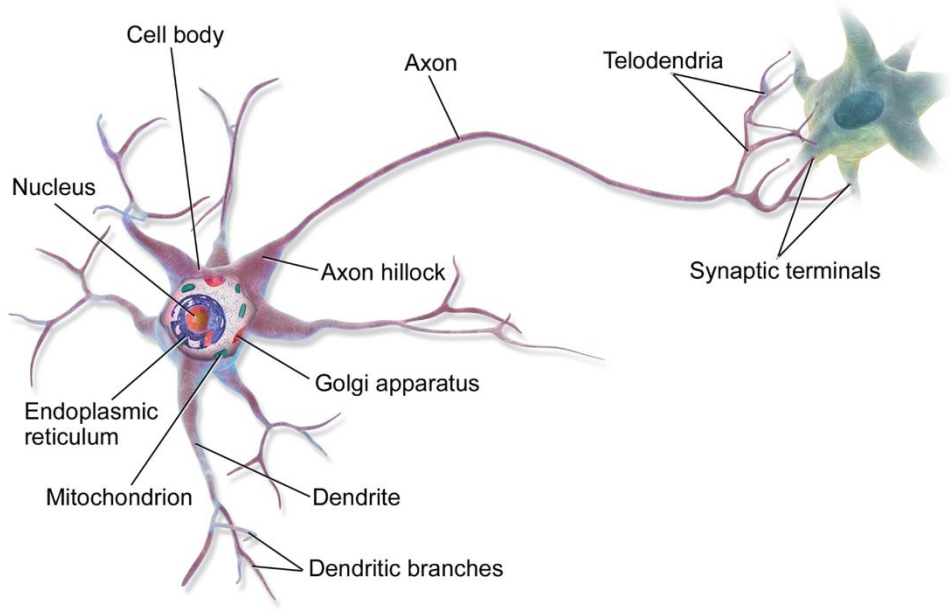
Outline

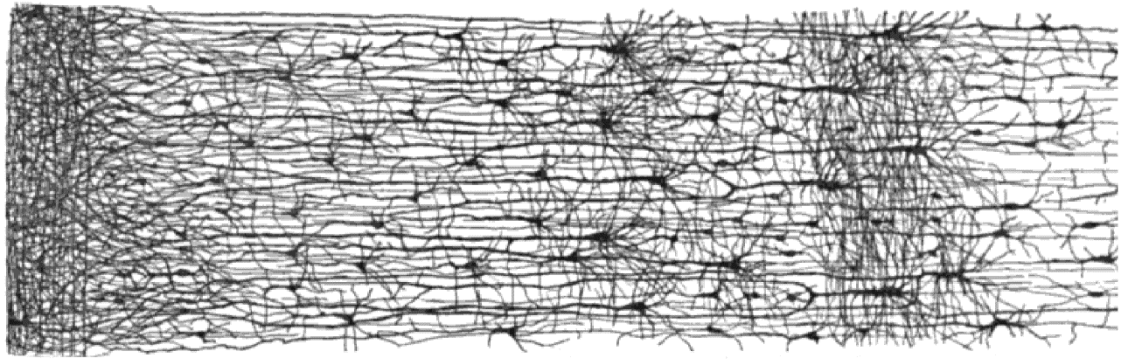
- From biological to artificial neurons
- Training a neural network
- Fine-tuning neural network hyperparameters
- Vanishing / exploding gradients problems
- Reusing pretrained layers
- Faster optimizers
- Avoiding overfitting through regularization

From biological to artificial neurons

Biological neurons

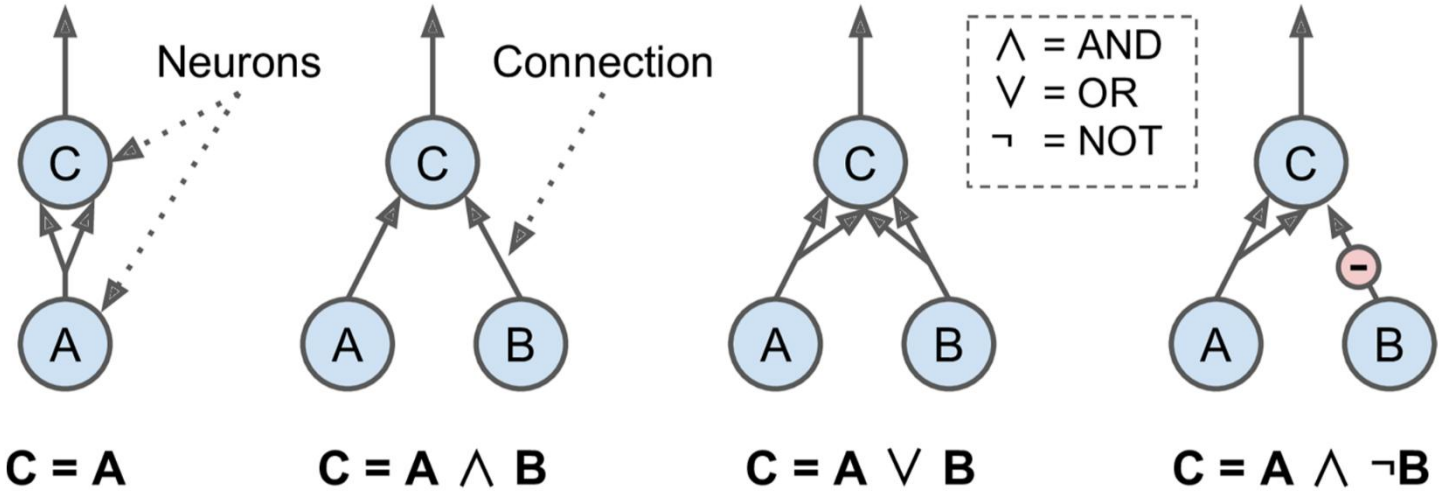
- Components in a biological neuron
  - Cell body
  - Dendrite
  - Axon
  - Synapses
- Communications among neurons
  - Neurons receive signals (i.e. short electrical impulses) from other neurons via synapses
  - When a neuron receives an enough number of signals, it fires its own signals
- Biological neural networks:
  - Neurons are organized in consecutive layers





Logical computations with neurons

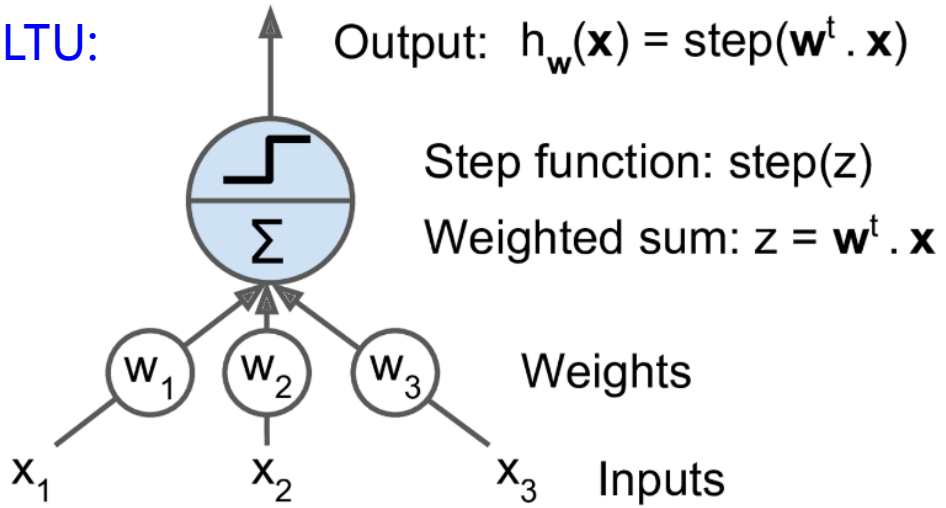
- **Artificial neural network** (McCulloch and Pitts, 1943): A simplified computational model of how biological neurons might work together in animal brains to perform computations using propositional logic
- An **artificial neuron**:
  - has one or more binary (on/off) inputs and one binary output
  - activates its output when more than a certain number (e.g. two in the example below) of its inputs are active



Linear threshold unit (LTU)

- A **linear threshold unit (LTU)** is a type of artificial neuron, where the inputs and output are numbers, instead of binary on/off values
  - Compute a **weighted sum** of inputs
  - Apply a **step function** to the sum to get the output

LTU:



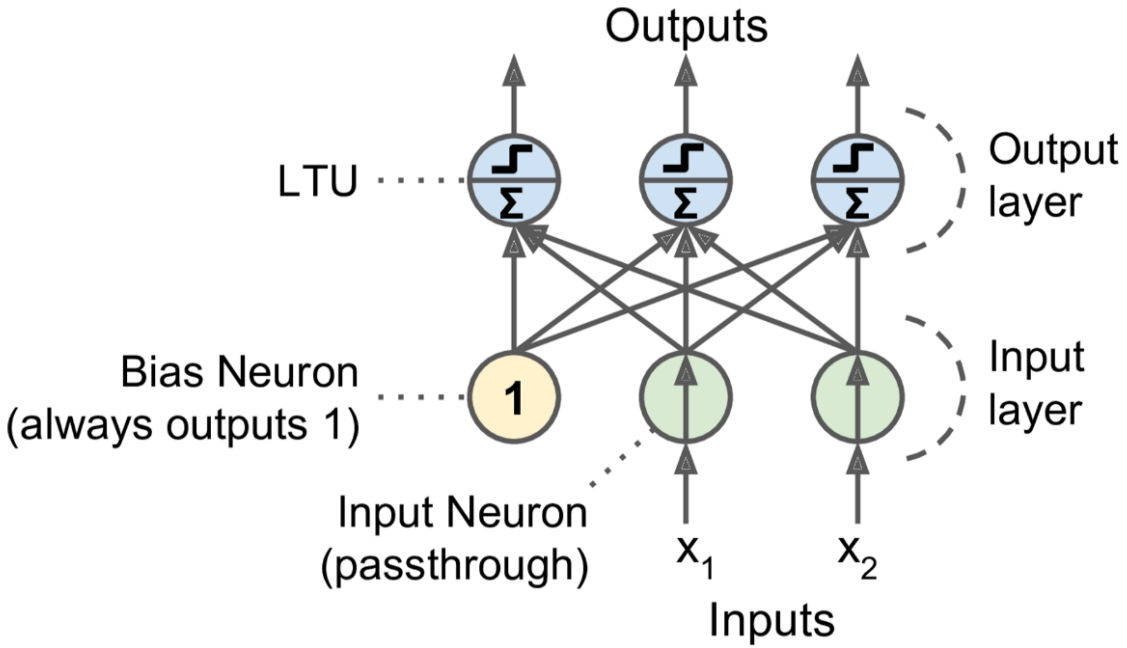
Common step functions:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

Perceptron

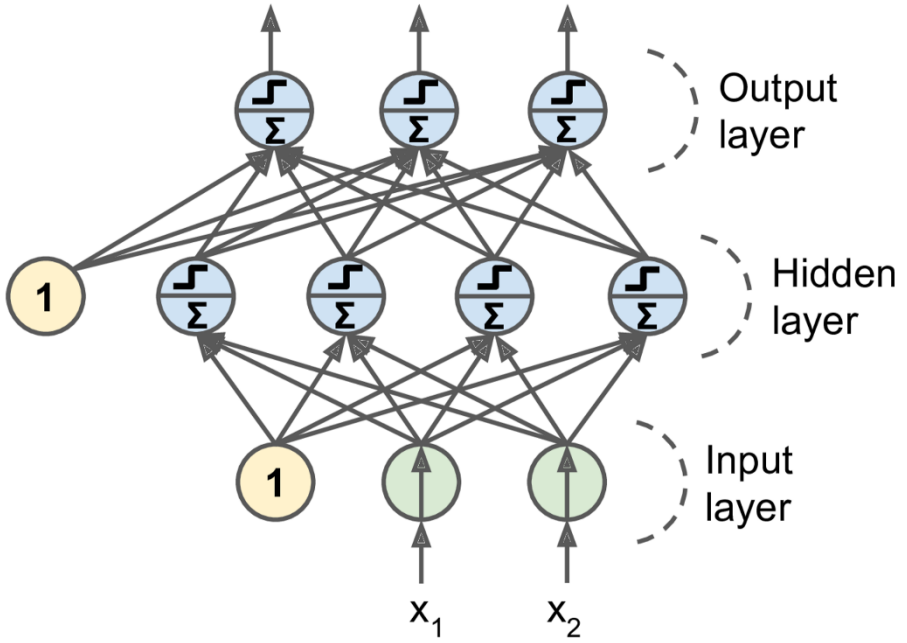
- A **perceptron** (invented by Frank Rosenblatt in 1957) is a simple ANN composed of a single layer of LTUs
  - Each neuron is connected to all the inputs
  - **Input neurons** output whatever input they are fed
  - **Bias neuron**: output 1 all the time
- A Perceptron is trained using **Hebb's rule**:
  - Between two neurons having the same output, increase the connection weight
  - Not to reinforce connections that lead to wrong output

Perceptron diagram:



Multi-Layer Perceptron (MLP)

- Perceptrons are unable to solve some trivial problems (e.g. the XOR classification problem)
- But some limitations of Perceptrons can be overcome by stacking multiple Perceptrons, resulting in the **Multi-Layer Perceptron (MLP)**
- An MLP is composed of
  - One **input layer**
  - One or more **hidden layers** (of LTUs)
  - One final **output layer** (of LTUs)



A Multi-Layer Perceptron

Deep neural network (DNN)

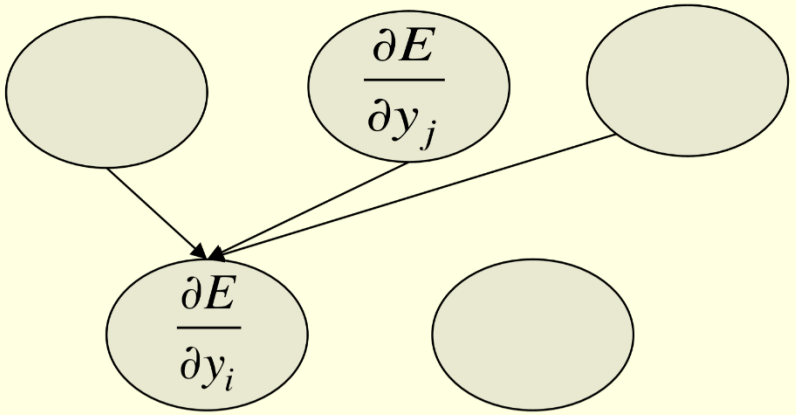
- When an ANN has two or more hidden layers, it is called a **deep neural network (DNN)**
- However, it was difficult to train MLPs for many years, until the introduction of the **backpropagation** algorithm in 1986

Backpropagation

- For each training instance, the backpropagation algorithm goes through 3 steps:
  - **Forward pass**: feed the training instance to the network, and compute the output of every neuron in each consecutive layer, i.e. to make a prediction
  - **Reverse pass**: measure the error (i.e. the difference between the desired output and the actual output), and go through each layer **in reverse** to measure the error contribution from each neuron in the previous hidden layer, until reaching the input layer
  - **Gradient descent**: slightly tweak the connection weights to reduce the error

$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

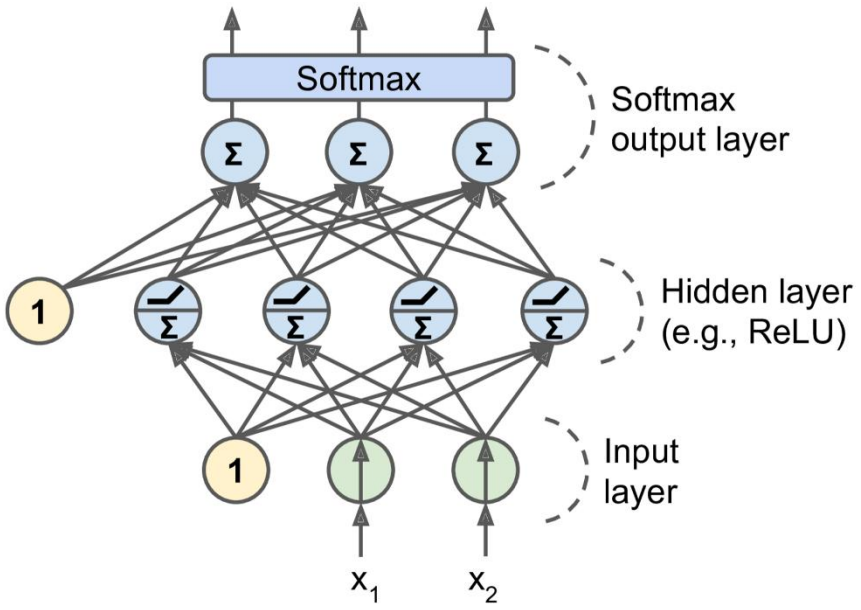
$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$

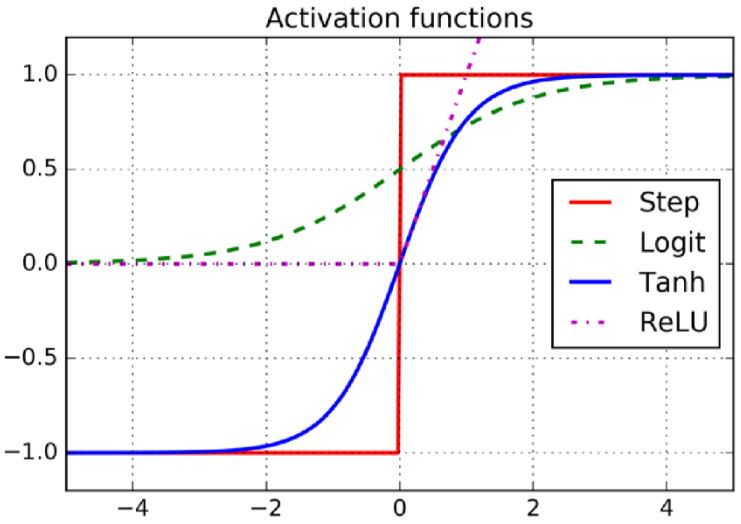


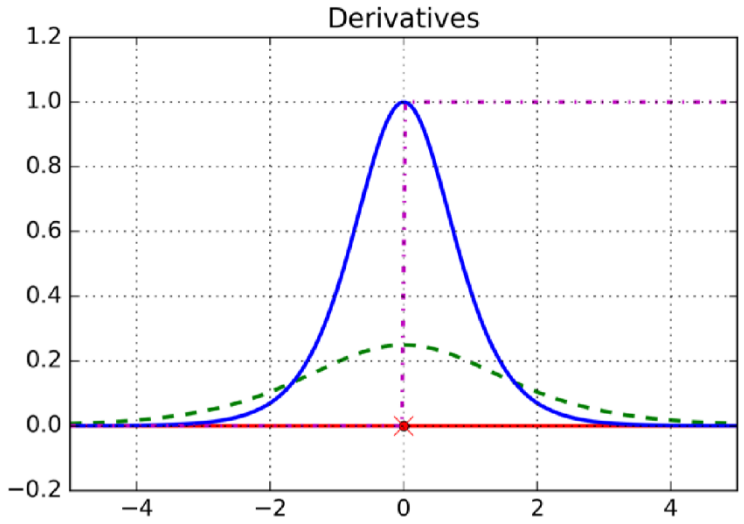
Error derivative in each hidden layer can be computed from the error derivatives in the layer above (i.e. backpropagating dE/dy)

New activation functions

- For the backpropagation algorithm to work properly, the step function is often replaced with other activation functions:
  - Logistic functions: $\sigma(z) = 1/(1 + \exp(-z))$
  - Hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$
  - ReLU function: $ReLU(z) = \max(0, z)$







Training a neural network

Construction phase (1)

- Let us build an MLP using TensorFlow's low-level Python API, and implement Mini-batch Gradient Descent to train it on the MNIST dataset, in two phases:
  - (1) **Construction phase**: build the TensorFlow graph
  - (2) **Execution phase**: run the graph to train the model
- Main steps of the construction phase:
  - (1) Import the tensorflow library
  - (2) Specify the network topology (set the numbers of neurons in the layers)
  - (3) Use placeholder nodes to represent the training data and targets
  - (4) Create the neural network

```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
    stddev = 2 / np.sqrt(n_inputs)
    init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
    W = tf.Variable(init, name="weights")
    b = tf.Variable(tf.zeros([n_neurons]), name="biases")
    z = tf.matmul(X, W) + b
    if activation=="relu":
        return tf.nn.relu(z)
    else:
        return z


with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

Construction phase (2)

- Main steps of the construction phase (continued):
  - (5) Define the **cost function** using **cross entropy** (i.e. to penalize models that estimate low probabilities for the target)
  - (6) Define the **optimizer** (Gradient Descent in the example) that will tweak the model parameters to minimize the cost function
  - (7) Define the **performance measure**
  - (8) Initialize all variables and save the trained model parameters to disk

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")


learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)


with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Execution phase

- After the DNN model is constructed, we train it against the MNIST dataset

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")

n_epochs = 400
batch_size = 50

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                            y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Using the neural network

- After the neural network is trained, we use it to make predictions

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = [...]  # some new images (scaled from 0 to 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

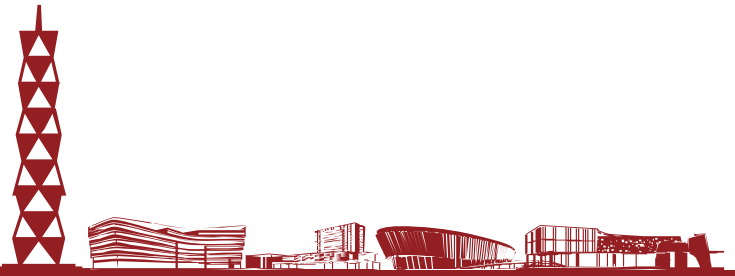
Fine-tuning neural network hyperparameters

Hidden layers

- Real-world data often have **hierarchical structures**, which can be captured by different layers of a deep neural network:
  - Lower hidden layers model low-level structures (e.g. line segments of various shapes and orientation)
  - Intermediate hidden layers combine the low-level structures to model intermediate-level structures (e.g. squares, circles)
  - Highest hidden layers and the output layer combine the intermediate structures to model high-level structures (e.g. faces)
- The hierarchical architecture can improve the ability of networks to **generalize** to new datasets, because the low-level structures can be shared and reused

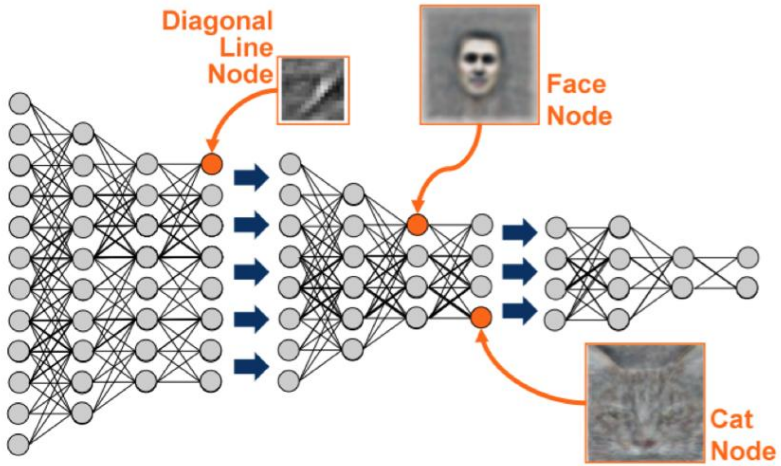
Number of hidden layers

- For many simple problems, a single hidden layer will give reasonable results
- For more complex tasks, however, more hidden layers will be better, because:
  - Deeper networks have higher parameter efficiency, i.e. they can model complex functions using exponentially fewer neurons than shallow nets, and speed up the training
- However, deeper networks **need large amounts of data**, and are prone to **overfitting** due to more parameters and higher model complexity



Number of neurons per hidden layer

- The numbers of neurons in the input and output layers are determined by a specific task
- For hidden layers, a common practice is to form a **funnel**:
  - With fewer and fewer neurons at each layer, because many low-level features can coalesce into fewer high-level features
  - Sometimes, all the hidden layers are set to the same size to reduce the number of parameters



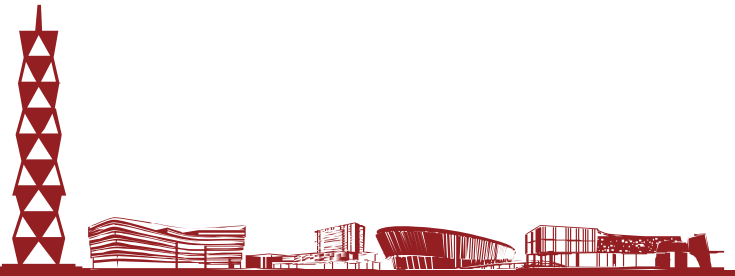
Strategies to find the number of neurons per layer

- Try increasing the number of neurons gradually until the network starts overfitting
- The **"stretch pants"** approach: Pick a model with more layers and neurons than you need, and then use regularization techniques (e.g. early stopping, dropout) to prevent it from overfitting

Activation functions

- In hidden layers, the ReLU function (or one of its variants) is often used, because:
  - It is faster to compute than other activation functions
  - It does not saturate for large input values （ in contrast to logistic or tanh which saturate at 1)
  - With ReLU, Gradient Descent does not stuck as much on plateaus

- For output layer:
  - Classification : the softmax function is generally a good choice
  - Regression: can use no activation function at all



Vanishing/exploding gradients problems

Challenges of training deep neural networks

- The **vanishing gradients** problem (or the related **exploding gradients** problem) that makes lower layers very hard to train
- There are **not enough training data**, or it is too costly to label the data
- Training may be extremely **slow**
- The risk of **overfitting** the training set, when
  - the model has many parameters,
  - there are not enough training instances, or
  - the data are too noisy

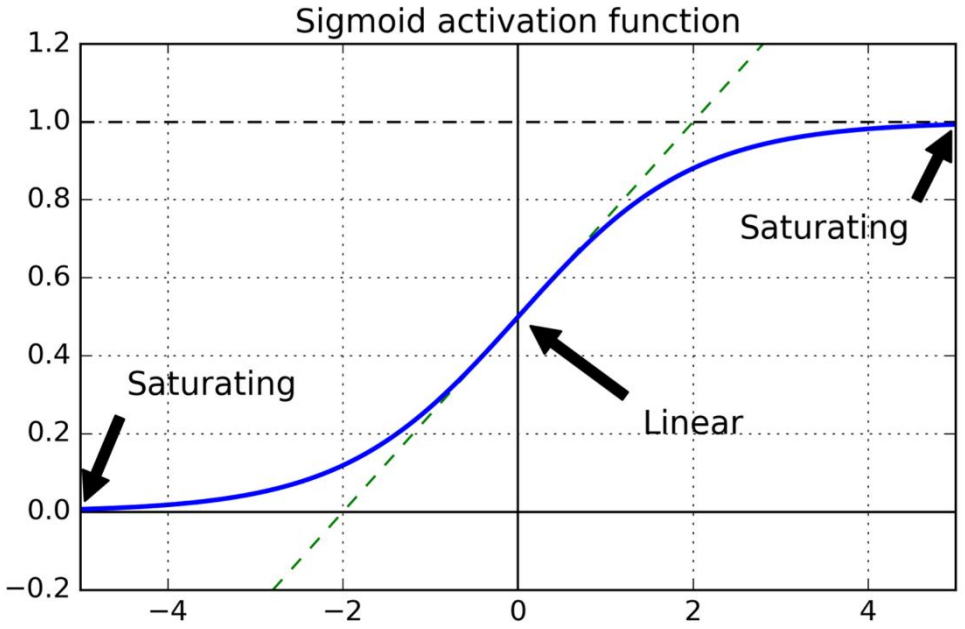
Vanishing/exploding gradients problems

- **Vanishing gradients** problem: In the backpropagation to lower layers, gradients often get smaller and smaller
  - Gradient Descent update leaves the lower-layer weights virtually unchanged, and training cannot converge to a good solution

- **Exploding gradients** problem: The gradients grow bigger and bigger
  - Many layers get large weight updates
  - The algorithm diverges
  - Mostly happens in recurrent neural networks

Vanishing/exploding gradients problems

- Reasons found by Xavier Glorot and Yoshua Bengio, around 2010:
  - With the combination of **random initialization** with a normal distribution and **logistic sigmoid activation function**, the variance of outputs of each layer is much bigger than the variance of its inputs
  - The activation function tends to saturate at the top layers



Saturation of logistic activation function

Vanishing/exploding gradients problems

- To alleviate the vanishing/exploding gradients problems, we need the signals to flow properly in both directions of backpropagation:
  - The variance of the outputs of each layer should be equal to the variance of its inputs
  - The gradients should have equal variance before and after flowing through a layer in the reverse direction

Glorot and He initialization

- **Glorot initialization** (i.e. **Xavier initialization**)
  - For each layer, the numbers of input and neurons are called **fan_{in}** and **fan_{out}** of the layer
  - The connection weights of each layer must be initialized randomly as described in the equations on the right side, where

$$fan_{avg} = (fan_{in} + fan_{out})/2$$

Glorot initialization with logistic activation function

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{avg}}}$

- **LeCun initialization**: Replace fan_{avg} with fan_{in} in the equations of Glorot initialization
- **He initialization**: For RELU and its variants

Initialization for different activation functions

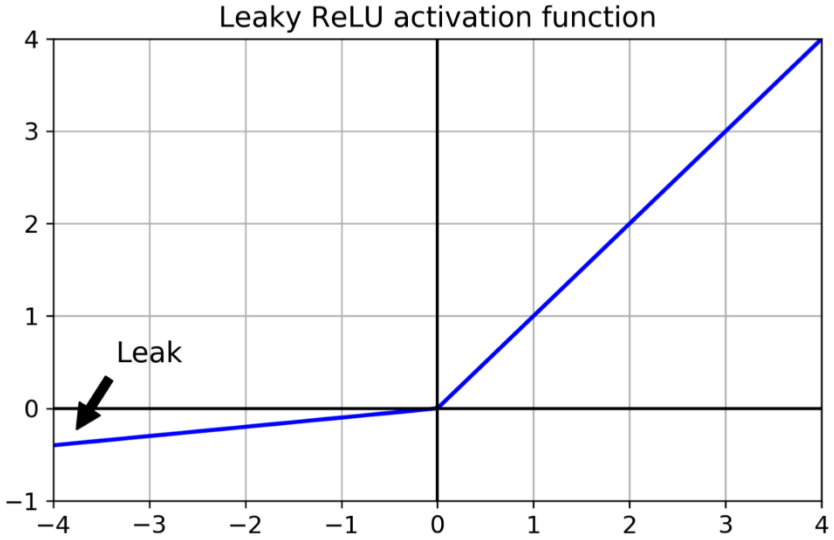
Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

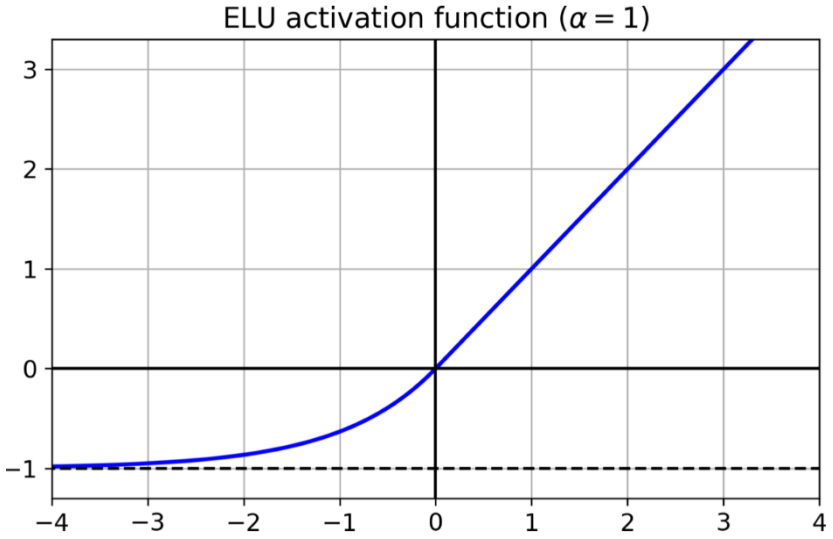
Nonsaturating activation function

- The **ReLU (Rectified linear unit)** activation function behaves better than sigmoid functions, because:
  - It does not saturate for positive values
  - It is fast to compute
- But ReLU suffers the problem of **dying ReLUs**, i.e. during training some neurons die, outputting only 0
- Variants of ReLU:
  - **Leaky ReLU**: $LeakyReLU_\alpha(z) = \max(\alpha z, z)$, where α is a hyperparameter that defines how much it "leaks", and α is usually set to 0.01
  - **RReLU (randomized leaky ReLU)**: α is picked randomly in a given range during training, and fixed during testing
  - **PReLU (parametric leaky ReLU)**: α is to be learned during training as a parameter (modified by backpropagation)
- **ELU (exponential linear unit)** activation function:

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

- **SELU (scaled ELU)**: make the network self-normalize





Batch Normalization (BN)

- How to reduce vanishing/exploding gradients problems during training?
- **Batch Normalization (BN)** is a technique (proposed in 2015):
  - Add an operation in the model just before or after the activation function of each hidden layer
  - Zero-center and normalize each input
  - Scale and shift the result using two parameter vectors per layer: one for scaling, the other for shifting
  - Evaluate the mean and standard deviation of each input over the current mini-batch

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$

2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$

3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$

4. $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

Batch Normalization algorithm

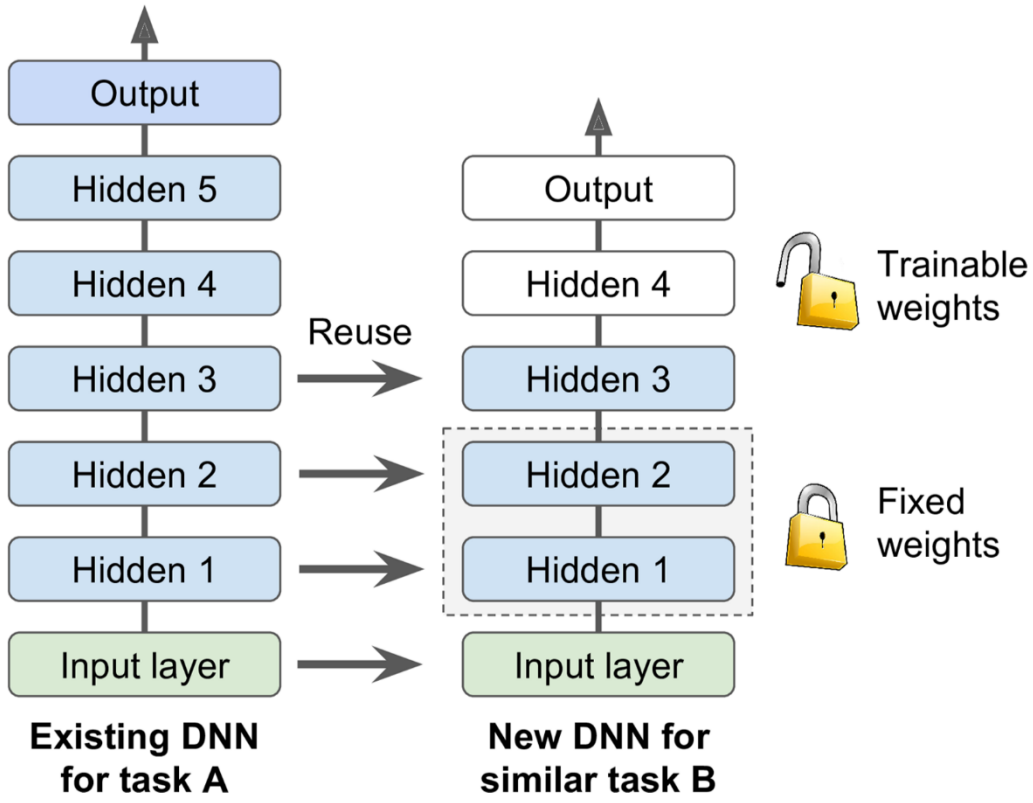
Benefits of using BN

- Can use saturating activation functions
- Less sensitive to weight initialization
- Can use larger learning rates
- Can act like a regularizer, to reduce overfitting

Reusing pretrained layers

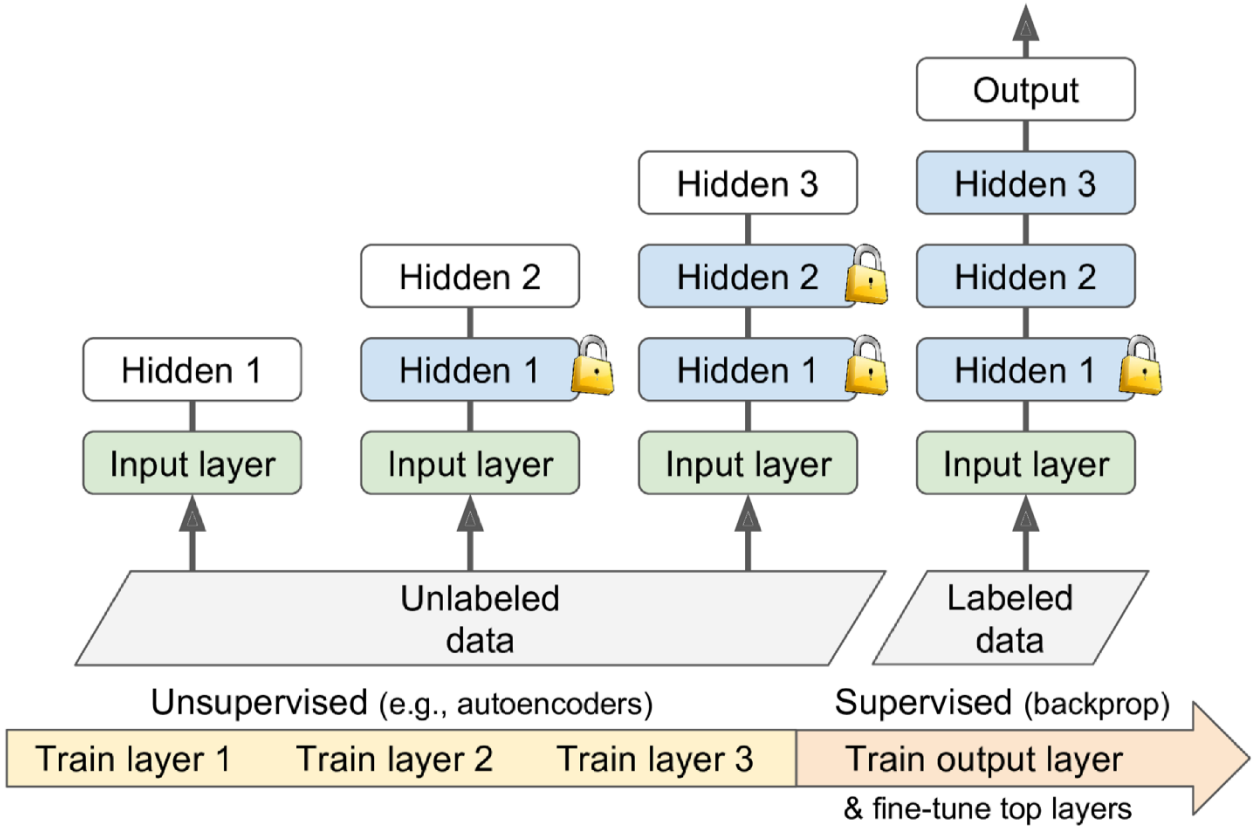
Transfer learning

- **Main idea**: Instead of training a large DNN from scratch,
  - try to find an existing neural network that accomplishes a similar task,
  - reuse the lower layers of the network
- **Challenge**: How to find the right number of layers to reuse?
- **Tips**:
  - Transfer learning works best when the inputs have similar low-level features
  - The more similar the tasks are, the more layers you want to reuse (starting with the lower layers)
  - Take into account the amount of training data



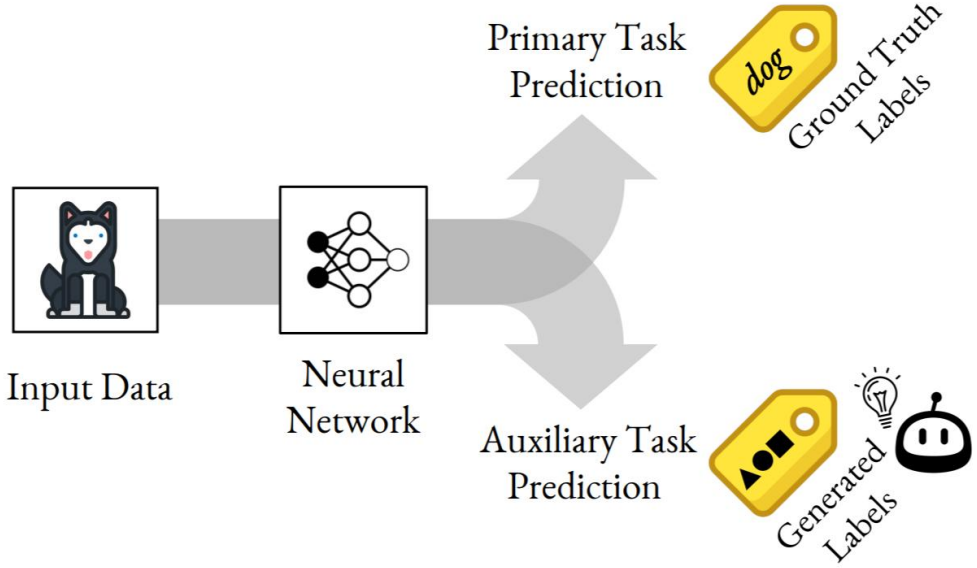
Unsupervised pretraining

- **Motivation**: It is often cheap to gather unlabeled training data, but expensive to label them
- **Idea**:
  - Try to train the layers, starting from the lowest layer and going up, using an **unsupervised** learning method (e.g. autoencoders)
  - Each layer is trained on the output of the previously trained layers
  - Once all layers have been trained this way, add the output layer for the task, and fine-tune the whole network using **supervised** learning (i.e. with the labeled data)
- This technique helped G. Hinton's team in 2006, leading to the revival of neural networks



Pretraining on an auxiliary task

- **Main idea**:
  - Train a first neural network on an auxiliary task, for which you can easily obtain or generate labeled training data.
  - Then, reuse the lower layers of the trained network for your actual task
- The idea involves **self-supervised learning**, which is a special form of **unsupervised learning**

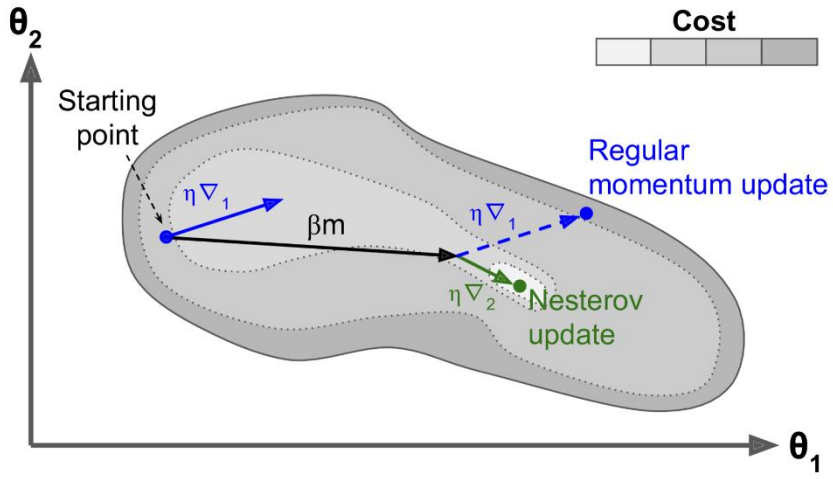


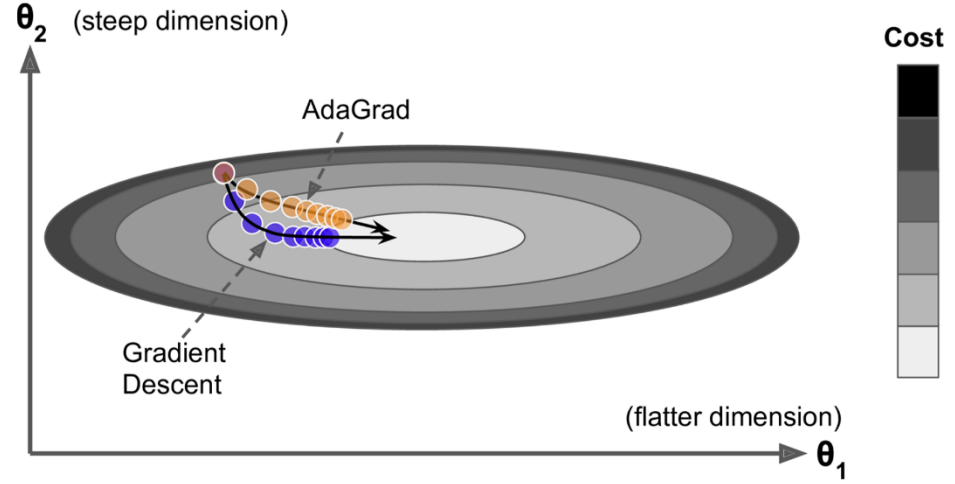
Source: S. Liu et al. arXiv, 2019

Faster optimizers

Optimizers faster than Gradient Descent

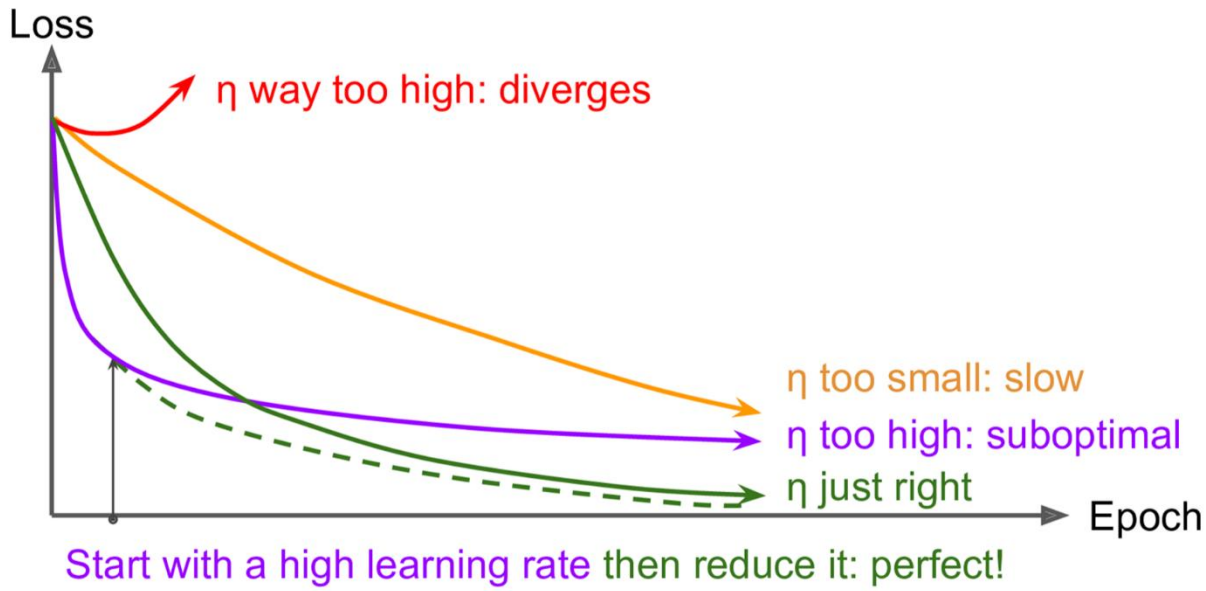
- **Momentum optimization**: rolls down the valley faster and faster until it reaches the bottom (optimum)
- **Nesterov Accelerated Gradient (NAG)**: measures the gradient of the cost function, not at the local position, but ahead in the direction of the momentum
- **AdaGrad**: scales down the gradient vector long the steepest dimensions, to point more directly toward the global optimum
- **RMSProp**: accumulates only the gradients from the most recent iterations, rather than all the gradients since the beginning
- **Adam**: combines the ideas of Momentum optimization and RMSProp
- **Nadam optimization**: combines Adam optimization and the Nesterov trick





Learning rate scheduling

- It is important but tricky to find a good learning rate
- General idea of **learning schedules**: Start with a high learning rate, and then reduce it once it stops making fast progress
  - Power scheduling: Set the learning rate as a function of the iteration number t, $\eta(t) = \eta_0 / (1 + t/s)^c$, so it iteratively drops to $(1/2)^c$, $(1/3)^c$, $(1/4)^c$, … of the initial rate η_0, and the power c is typically set to 1
  - Exponential scheduling: $\eta(t) = \eta_0 (1/10)^{t/s}$, so the learning rate drops by 10 times every s steps
  - Piecewise constant scheduling: Use a constant learning rate for a number of epochs, then a smaller learning rate for another number of epochs, and so on
  - Performance scheduling: Measure the validation error every N steps and reduce the learning rate by a factor when the error stops dropping

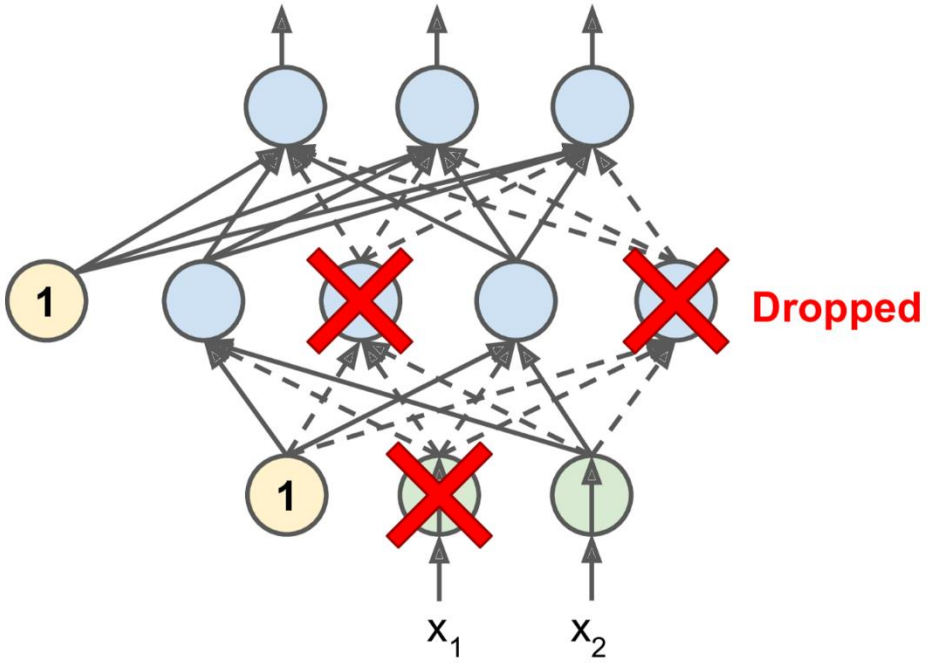


Learning curves for various learning rates

Avoiding overfitting through regularization

Dropout

- The **dropout algorithm**:
  - At every training step, every neuron (except for output neurons) has a probability p of being temporarily "dropped out"
  - The hyperparameter p is called the **dropout rate**, and is typically set to 50%
  - After training, neurons don't get dropped any more
- A variant: Monte-Carlo (MC) dropout:
  - Average over multiple predictions with dropout
  - Link dropout networks with approximate Bayesian inference
  - Provides a better measure of the model's uncertainty



Why dropout works?

- Neurons trained with dropout cannot co-adapt with their neighboring neurons
- Neurons are forced to pay attention to **all** input neurons, and thus be less sensitive to small changes in the inputs

Other methods of regularization

- L_1 and L_2 regularization: Add L_1 and L_2 regularization loss to the final loss, to constrain the connection weights of a neural network
- Early stopping: stop training as soon as the validation error reaches a minimum
- Max-Norm regularization: Constrains the weights $\mathbf{w}$ of the incoming connections such that $\|\mathbf{w}\|_2 \leq r$, where r is the max-norm hyperparameter
- Batch Normalization

Summary

- In this lecture we have learned
  - Analogy between biological neurons and artificial neurons
  - How to train a neural network
  - How to fine-tune neural network hyperparameters
  - Challenges of training very deep neural networks
  - The vanishing / exploding gradients problems and techniques to address them
  - Techniques of reusing pretrained layers to address the challenge of insufficient labeled data
  - Faster optimizers
  - How to avoid overfitting by regularization techniques (e.g. dropout)

- For details, read Chapter 10 and Chapter 11 of Aurélien Géron’ s book “Hands-On Machine Learning with SciKit-Learn & TensorFlow”