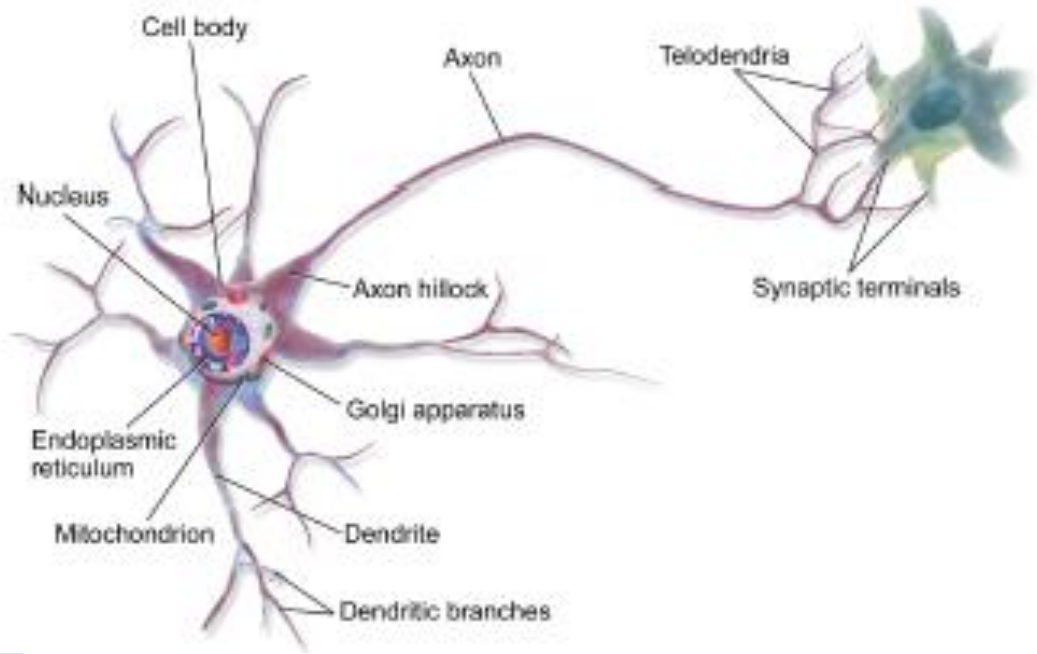
Deep Neural Networks (DNN)

Saeed Sharifian

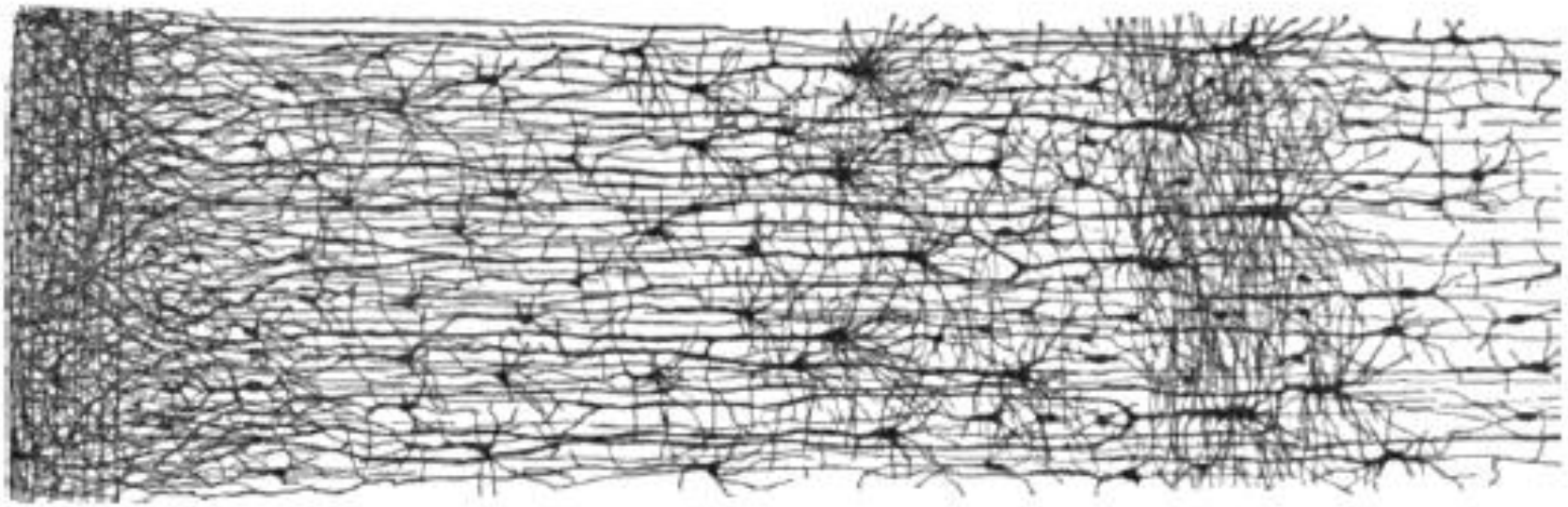
Biological Neurons

- Brain architecture has inspired artificial neural networks.
- A biological neuron is composed of
  - Cell body, many dendrites (branching extensions), one axon (long extension), synapses
- Biological neurons receive signals from other neurons via these synapses.
- When a neuron receives a sufficient number of signals within a few milliseconds, it fires its own signals.



Biological Neurons

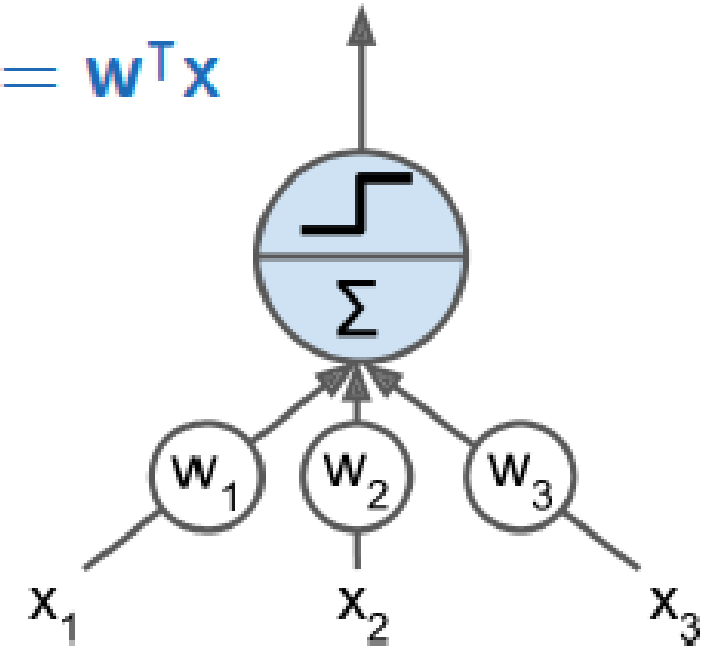
- Biological neurons are organized in a vast network of billions of neurons.
- Each neuron typically is connected to thousands of other neurons.



The Linear Threshold Unit (LTU)

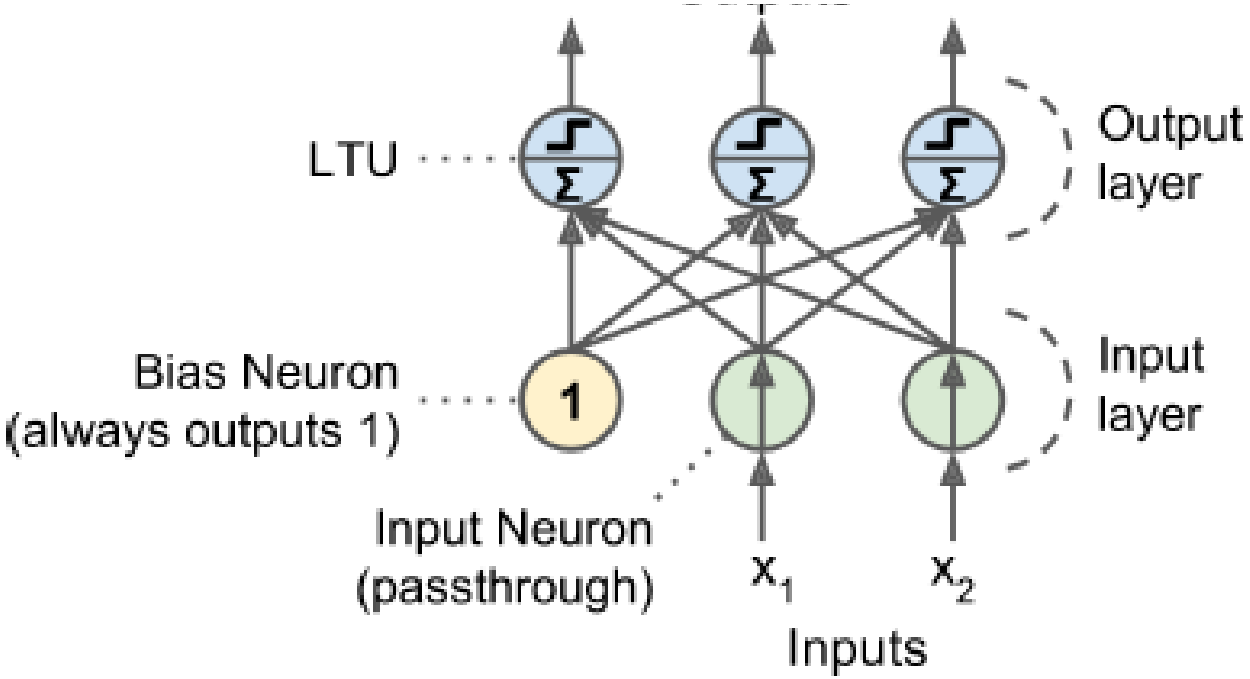
- Inputs of a LTU are numbers
- Each input connection is associated with a weight.
- Computes a weighted sum of its inputs and applies a step function to that sum.

- $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T x$
- $\hat{y} = \text{step}(z) = \text{step}(w^T x)$



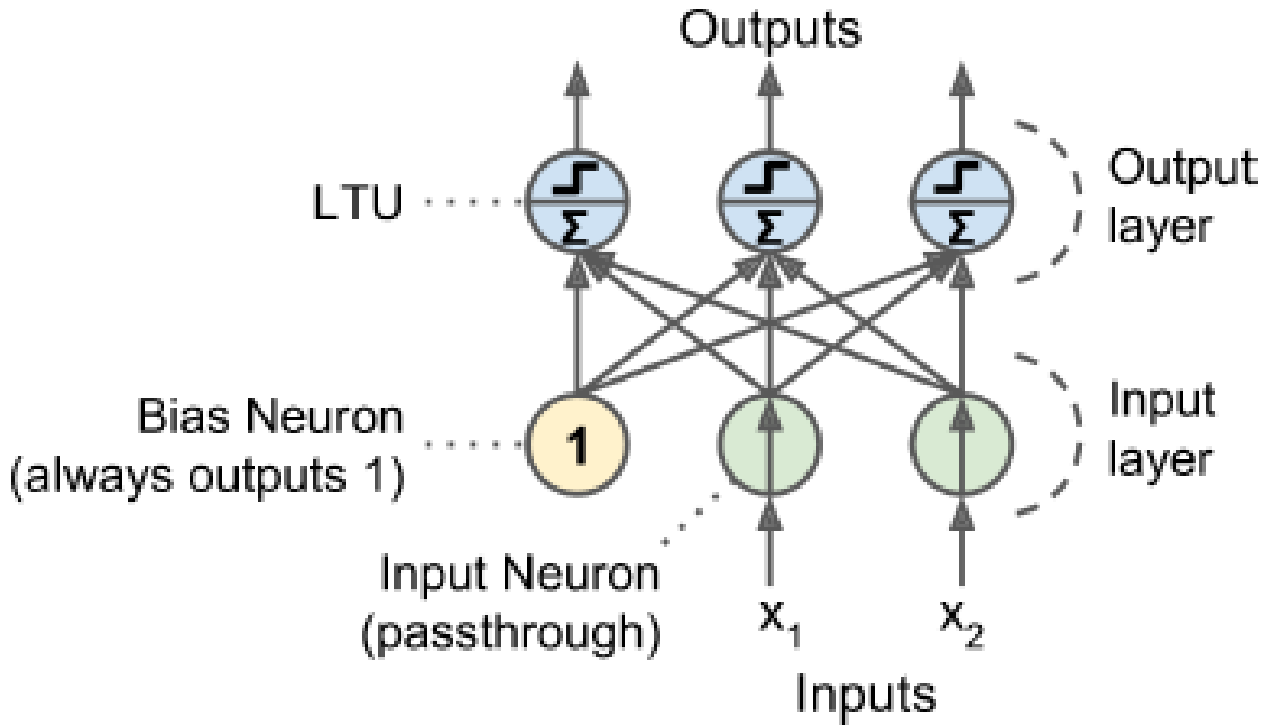
The Perceptron

- The perceptron is a single layer of LTUs.
- The input neurons output whatever input they are fed.
- A bias neuron, which just outputs 1 all the time.
- If we use logistic function (sigmoid) instead of a step function, it computes a continuous output.



How is a Perceptron Trained?

- The Perceptron training algorithm is inspired by Hebb's rule.
- When a biological neuron often triggers another neuron, the connection between these two neurons grows stronger.

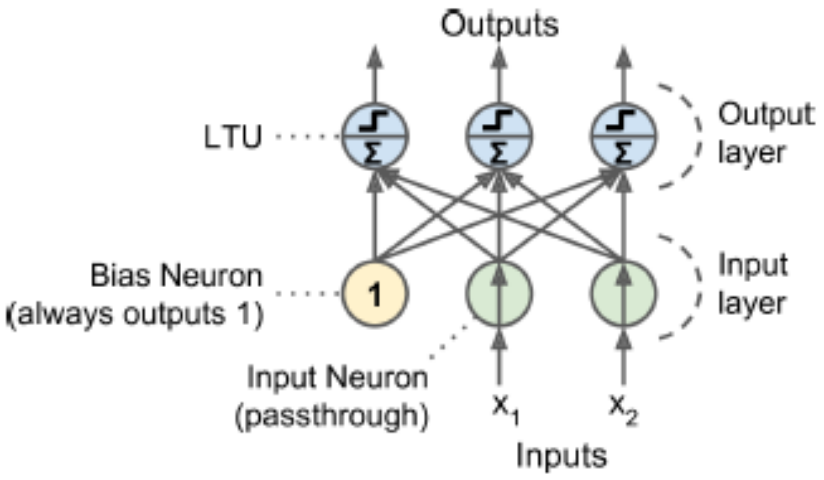


How is a Perceptron Trained?

- Feed one training instance $\mathbf{x}$ to each neuron $\mathtt{j}$ at a time and make its prediction $\hat{\mathtt{y}}$.
- Update the connection weights.

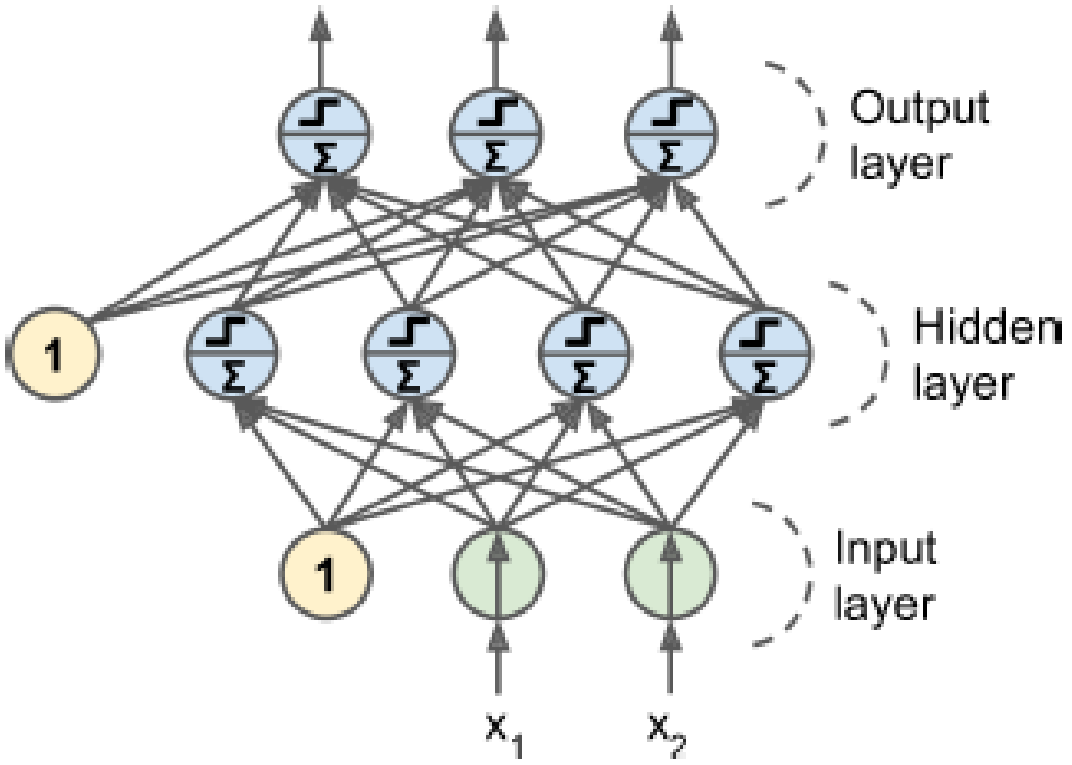
$$\hat{\mathtt{y}}_\mathtt{j} = \sigma(\mathbf{w}_\mathtt{j}^\intercal \mathbf{x} + \mathtt{b})$$
$$\mathtt{J}(\mathbf{w}_\mathtt{j}) = \mathtt{cross\_entropy}(\mathtt{y}_\mathtt{j}, \hat{\mathtt{y}}_\mathtt{j})$$
$$\mathtt{w}_{\mathtt{i},\mathtt{j}}^{(\mathtt{next})} = \mathtt{w}_{\mathtt{i},\mathtt{j}} - \eta \frac{\partial \mathtt{J}(\mathbf{w}_\mathtt{j})}{\mathtt{w}_\mathtt{i}}$$

- $\mathtt{w}_{\mathtt{i},\mathtt{j}}$: the weight between neurons $\mathtt{i}$ and $\mathtt{j}$.
- $\mathtt{x}_\mathtt{i}$: the $\mathtt{i}$th input value.
- $\hat{\mathtt{y}}_\mathtt{j}$: the $\mathtt{j}$th predicted output value.
- $\mathtt{y}_\mathtt{j}$: the $\mathtt{j}$th true output value.
- η: the learning rate.



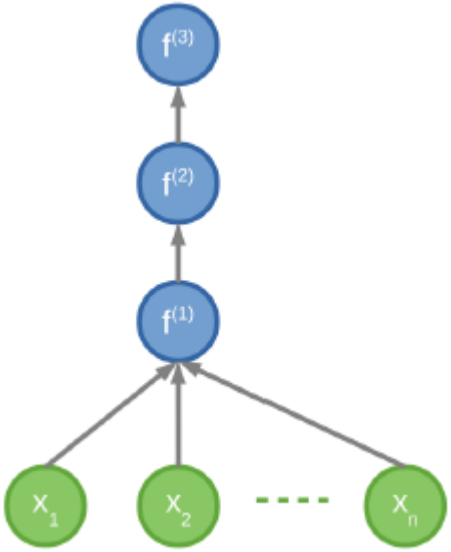
Multi-Layer Perceptron (MLP)

- A feedforward neural network is composed of:
  - One input layer
  - One or more hidden layers
  - One final output layer
- Every layer except the output layer includes a bias neuron and is fully connected to the next layer.



How Does it Work?

- The model is associated with a directed acyclic graph describing how the functions are composed together.
- E.g., assume a network with just a single neuron in each layer.
- Also assume we have three functions $\mathbf{f}^{(1)}$, $\mathbf{f}^{(2)}$, and $\mathbf{f}^{(3)}$ connected in a chain: $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}(\mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\mathbf{x})))$
- $\mathbf{f}^{(1)}$ is called the first layer of the network.
- $\mathbf{f}^{(2)}$ is called the second layer, and so on.
- The length of the chain gives the depth of the model



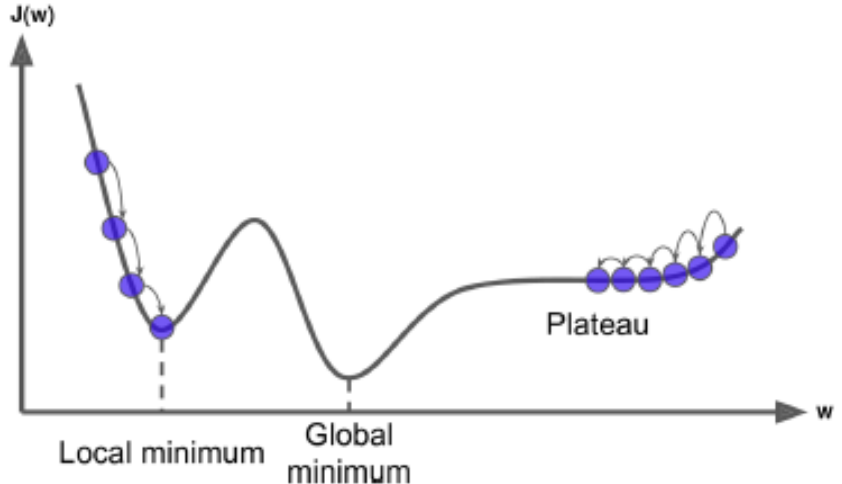
How to Learn Model Parameters W?

- We use the cross-entropy (minimizing the negative log-likelihood) between the training data $\mathbf{y}$ and the model's predictions $\hat{\mathbf{y}}$ as the cost function.

$$\mathtt{cost}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j} \mathbf{y}_j \log(\hat{\mathbf{y}}_j)$$

Gradient-Based Learning

- The most significant difference between the linear models we have seen so far and feedforward neural network?
- The non-linearity of a neural network causes its cost functions to become non-convex.
- Linear models, with convex cost function, guarantee to find global minimum.
  - Convex optimization converges starting from any initial parameters.

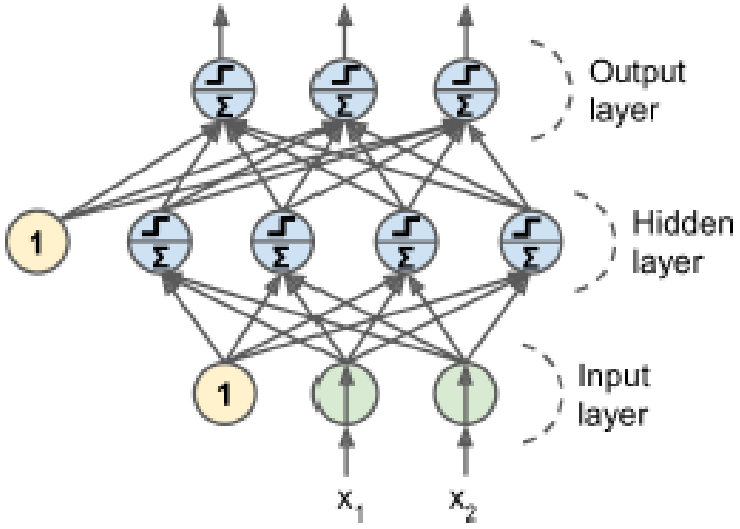


Gradient-Based Learning

- Stochastic gradient descent applied to non-convex cost functions has no such convergence guarantee.
- It is sensitive to the values of the initial parameters.
- For feedforward neural networks, it is important to initialize all weights to small random values.
- The biases may be initialized to zero or to small positive values.

Training Feedforward Neural Networks

- How to train a feedforward neural network?
- For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following steps:
  1. Forward pass: make a prediction (compute $\hat{\mathbf{y}}^{(i)} = \mathtt{f}(\mathbf{x}^{(i)})$).
  2. Measure the error (compute $\mathtt{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).
  3. Backward pass: go through each layer in reverse to measure the error contribution from each connection.
  4. Tweak the connection weights to reduce the error (update $\mathbf{W}$ and $\mathbf{b}$).
- It's called the backpropagation training algorithm



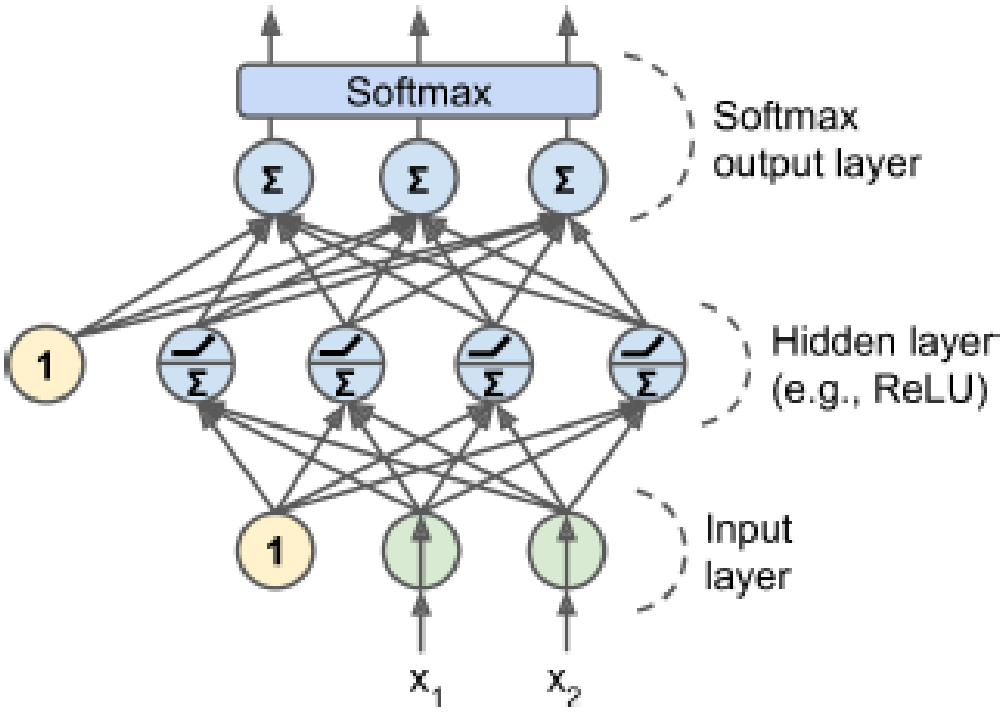
Output Unit

- Linear units in neurons of the output layer.
- Given $\mathbf{h}$ as the output of neurons in the layer before the output layer.
- Each neuron j in the output layer produces $\hat{y}_j = \mathbf{w}_j^T \mathbf{h} + b_j$.
- Minimizing the cross-entropy is then equivalent to minimizing the mean squared error.

- Sigmoid units in neurons of the output layer (binomial classification).
- Given $\mathbf{h}$ as the output of neurons in the layer before the output layer.
- Each neuron j in the output layer produces $\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{h} + b_j)$.
- Minimizing the cross-entropy.

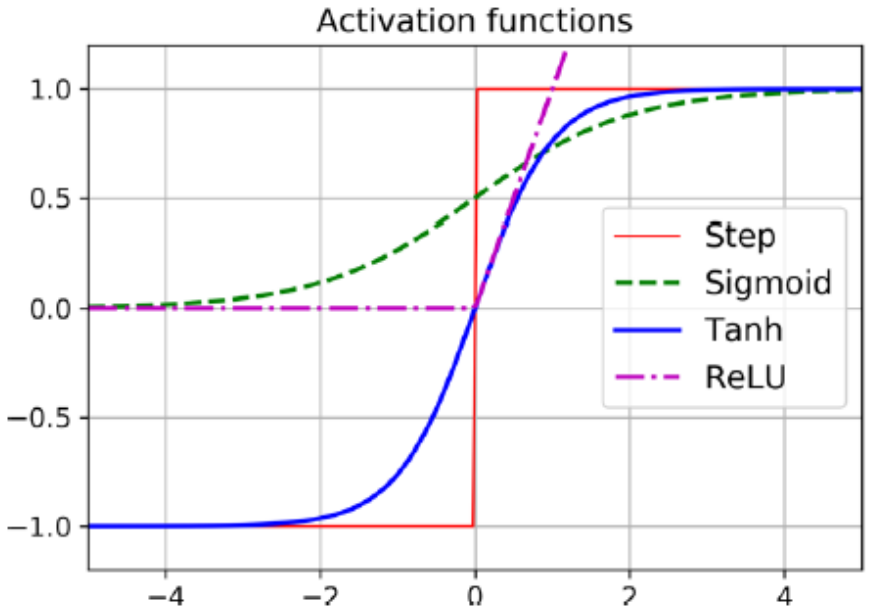
Output Unit

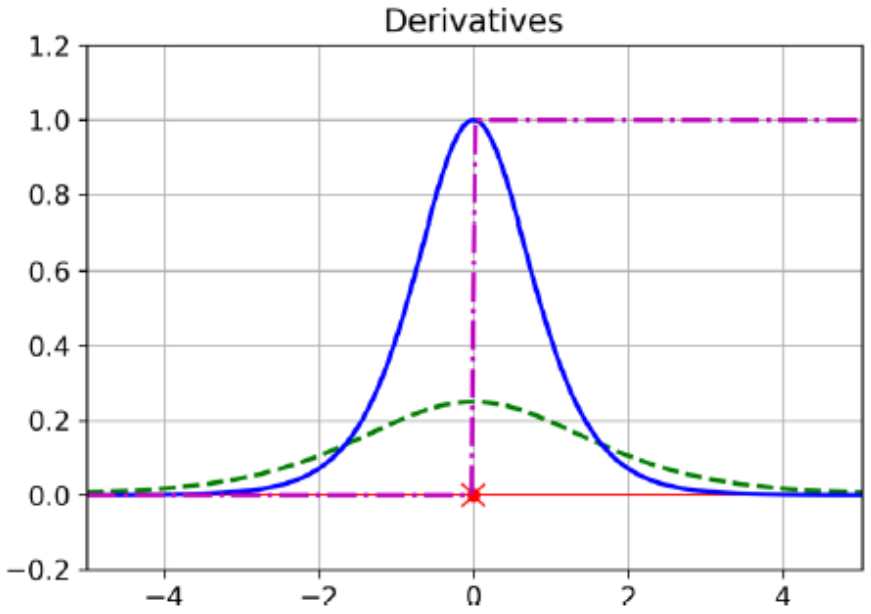
- Softmax units in neurons of the output layer (multinomial classification).
- Given $\mathbf{h}$ as the output of neurons in the layer before the output layer.
- Each neuron $\mathtt{j}$ in the output layer produces $\hat{\mathtt{y}}_\mathtt{j} = \mathtt{softmax}(\mathbf{w}_\mathtt{j}^\mathtt{T}\mathbf{h} + \mathtt{b}_\mathtt{j})$.
- Minimizing the cross-entropy.



Hidden Units

- In order for the backpropagation algorithm to work properly, we need to replace the step function with other activation functions. Why?
- Alternative activation functions:
  1. Logistic function (sigmoid): $\sigma(z) = \frac{1}{1+e^{-z}}$
  2. Hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$
  3. Rectified linear units (ReLUs): $\mathrm{ReLU}(z) = \max(0, z)$



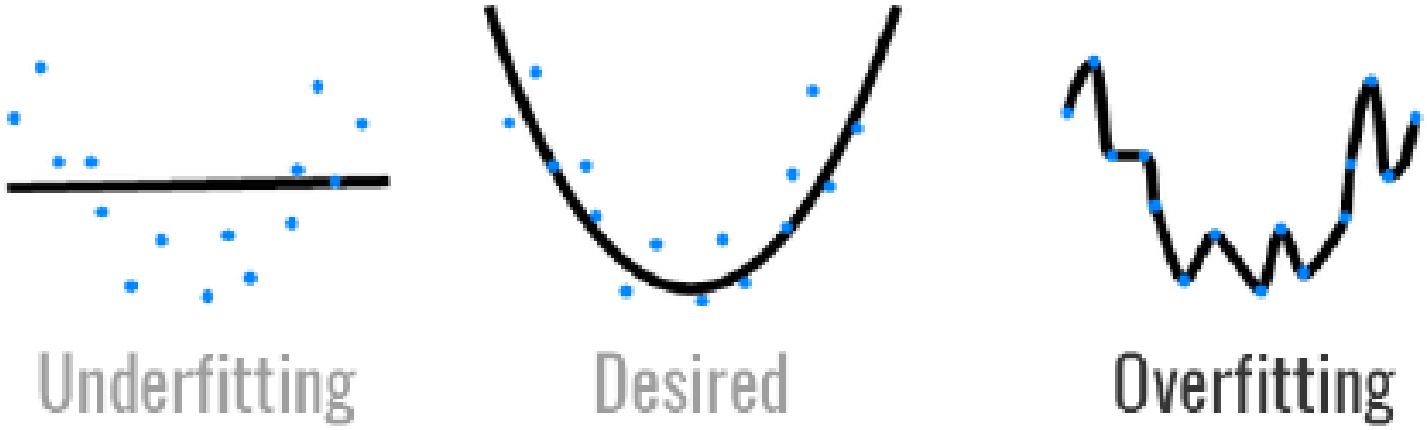


Challenges of Training Deep NN

- Overfitting: risk of overfitting a model with large number of parameters.
- Vanishing/exploding gradients: hard to train lower layers.
- Training speed: slow training with large networks.

High Degree of Freedom and Overfitting

- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.
- This flexibility also means that it is prone to overfitting on training set.
- Regularization: a way to reduce the risk of overfitting.
- It reduces the degree of freedom a model.

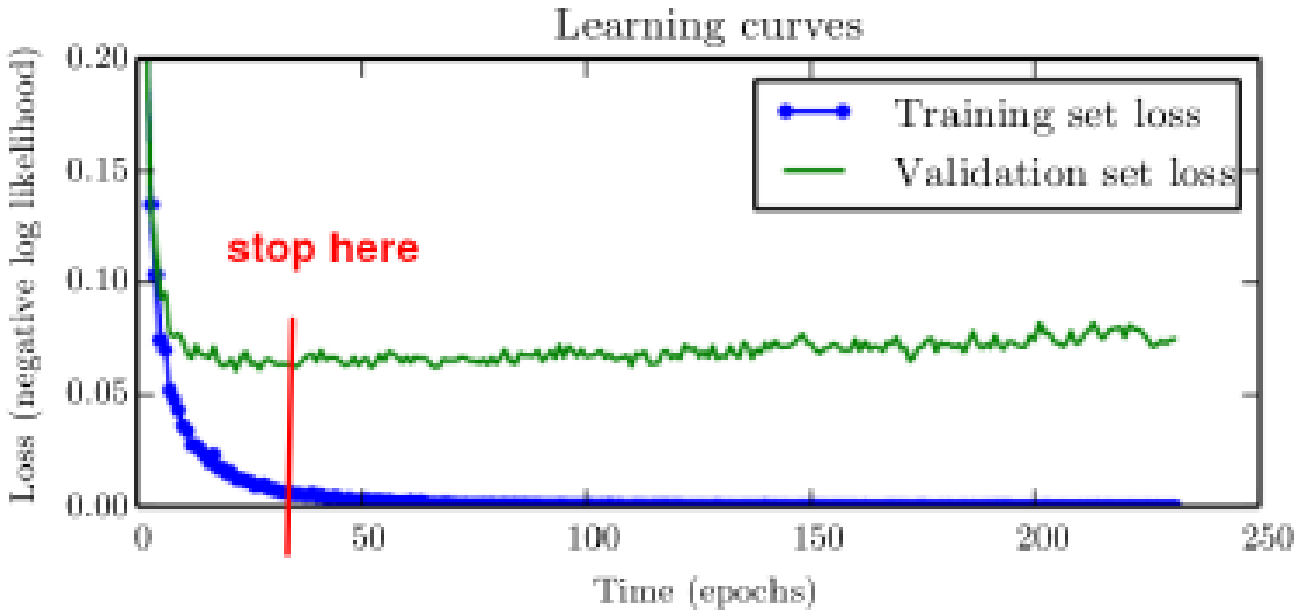


Avoiding Overfitting Through Regularization

- Early stopping
- /1 and /2 regularization
- Max-norm regularization
- Dropout
- Data augmentation

Early Stopping

- As the training steps go by, its prediction error on the training/validation set naturally goes down.
- After a while the validation error stops decreasing and starts to go back up.
  - The model has started to overfit the training data.
- In the early stopping, we stop training when the validation error reaches a minimum.



l1 and l2 Regularization

- Penalize large values of weights w_j. $\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda R(\mathbf{w})$
- Two questions:
  1. How should we define $R(\mathbf{w})$?
  2. How do we determine λ?

- *l1* regression: $R(\mathbf{w}) = \lambda \sum_{i=1}^{n} |w_i|$ is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^{n} |w_i|$$

- *l2* regression: $R(\mathbf{w}) = \lambda \sum_{i=1}^{n} w_i^2$ is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^{n} w_i^2$$

Max-Norm Regularization

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron j.
  - Prevents them from getting too large.

- After each training step, clip $\mathbf{w}_j$ as below:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{r}{||\mathbf{w}_j||_2}$$

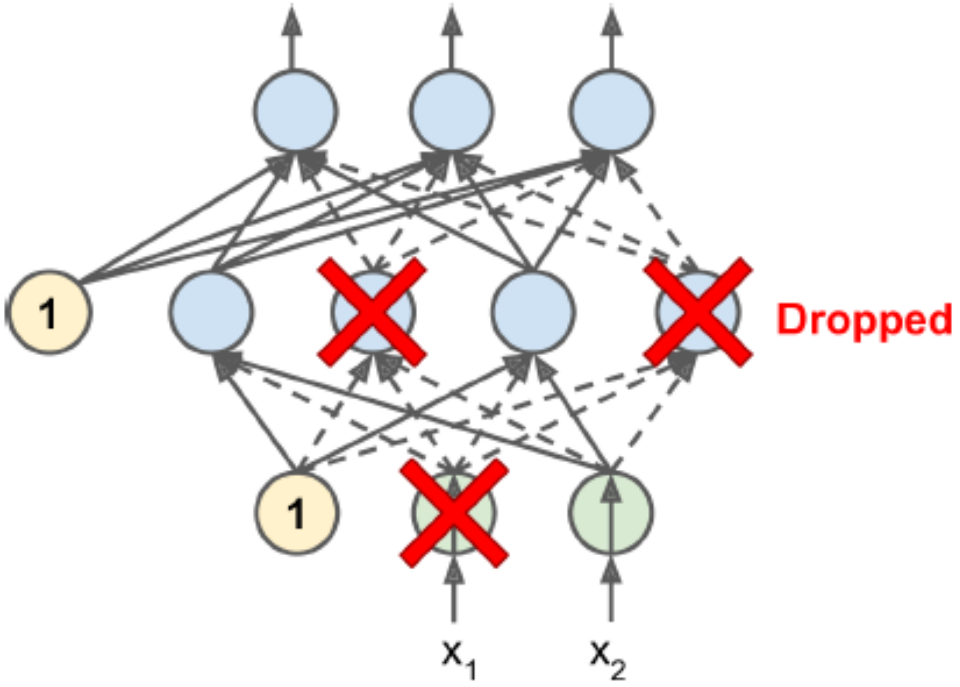
- We have $||\mathbf{w}_j||_2 \leq r$.
  - r is the max-norm hyperparameter
  - $||\mathbf{w}_j||_2 = (\sum_i w_{i,j}^2)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \cdots + w_{n,j}^2}$

Dropout

- Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work?

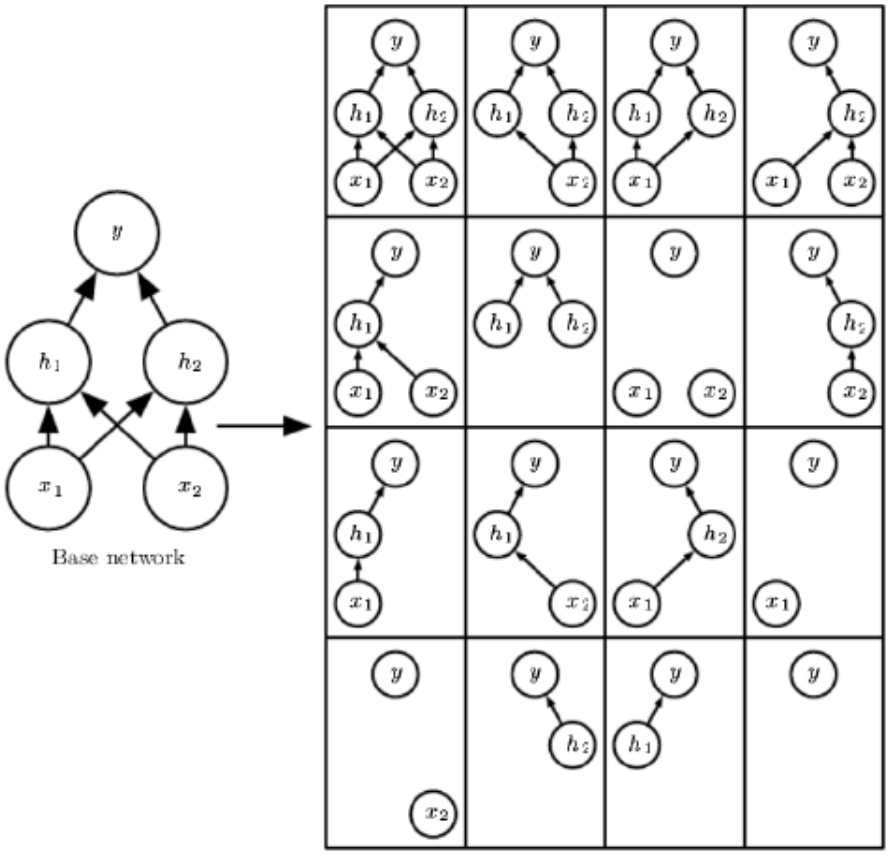
Dropout

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.
  - A neuron will be entirely ignored during this training step.
  - It may be active during the next step.
  - Exclude the output neurons.
- After training, neurons don't get dropped anymore.



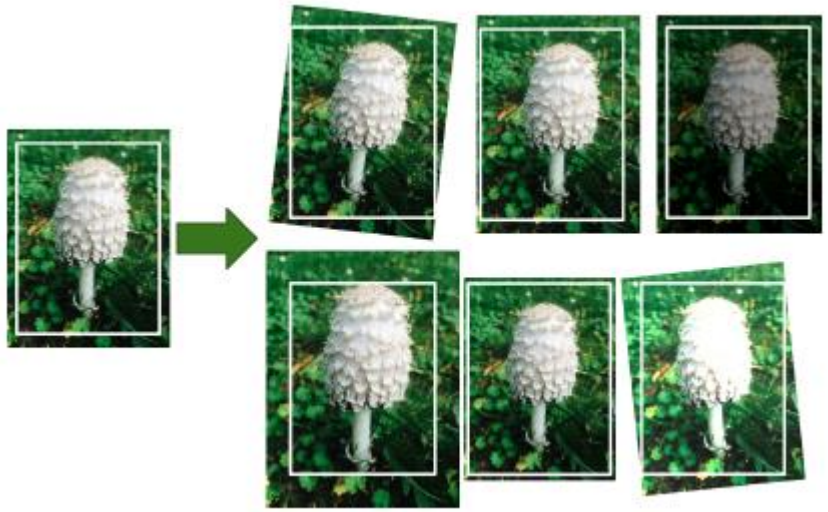
Dropout

- Each neuron can be either present or absent.
- 2^N possible networks, where N is the total number of droppable neurons.
  - $N = 4$ in this figure.



Data Augmentation

- One way to make a model generalize better is to train it on more data.
- This will reduce overfitting.
- Create fake data and add it to the training set.
  - E.g., in an image classification we can slightly shift, rotate and resize an image.
  - Add the resulting pictures to the training set.



Vanishing/Exploding Gradients Problem

- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

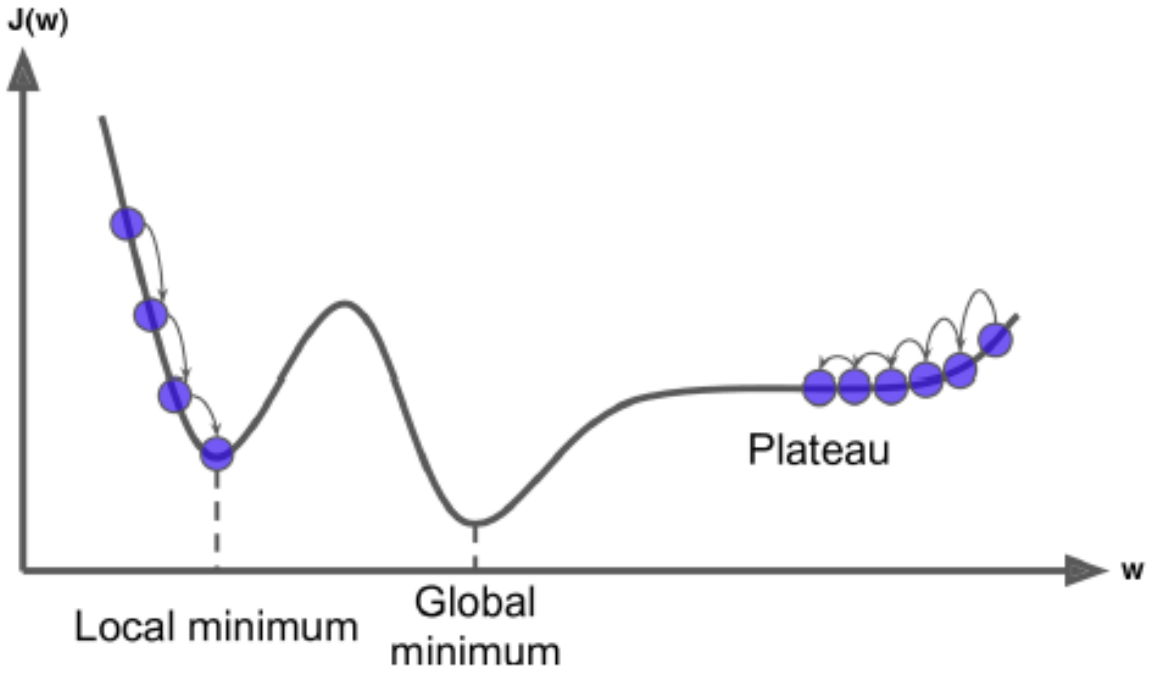
- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.
- This is called the vanishing gradients problem.

Overcoming the Vanishing Gradient

- Parameter initialization strategies
- Nonsaturating activation function
- Batch normalization
- Gradient clipping

Parameter Initialization Strategies

- The non-linearity of a neural network causes the cost functions to become non-convex.
- The stochastic gradient descent on non-convex cost functions performs is sensitive to the values of the initial parameters.
- Designing initialization strategies is a difficult task.



Parameter Initialization Strategies

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.
  - The goal of having each unit compute a different function.
- It motivates random initialization of the parameters.
  - Typically, we set the biases to constants, and initialize only the weights randomly.
- We need the signals to flow properly in both directions.
- The Xavier initialization proposed that:
  - The variance of the outputs of each layer to be equal to the variance of its inputs.
  - The gradients to have equal variance before and after flowing through a layer in the reverse direction.

Parameter Initialization Strategies

- Based on the Xavier initialization, the weights are initialized using normal distribution with mean 0 and the following standard deviation.
  - For the sigmoid activation function:

$$\sigma = \sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$$

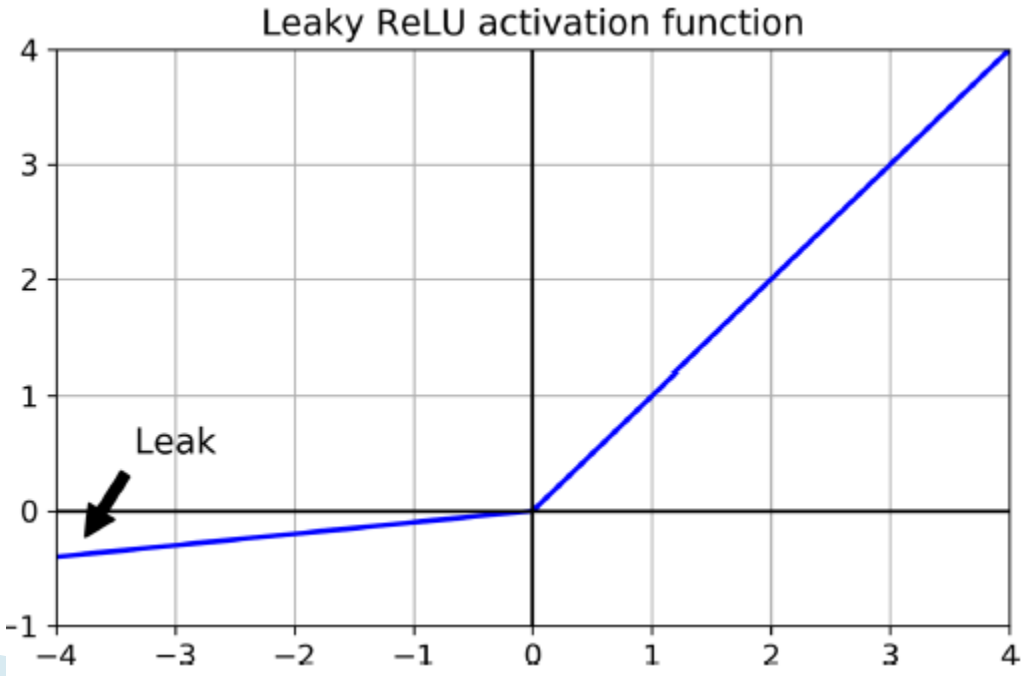
  - For the ReLU activation function:

$$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$$

  - n_{inputs} and $n_{outputs}$ are the number of input and output connections for the layer whose weights are being initialized.

No saturating Activation Functions

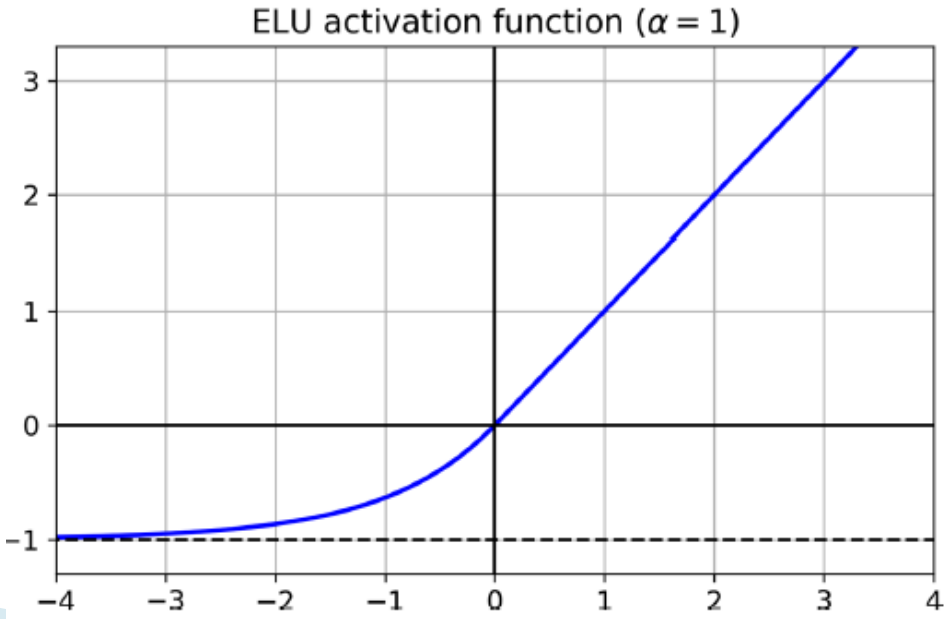
- $\mathrm{ReLU}(z) = \max(0, z)$
- The dying ReLUs problem.
  - During training, some neurons stop outputting anything other than 0.
  - E.g., when the weighted sum of the neuron's inputs is negative, it starts outputting 0.
- Use leaky ReLU instead: $\mathrm{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$.
  - α is the slope of the function for $z < 0$.



No saturating Activation Functions

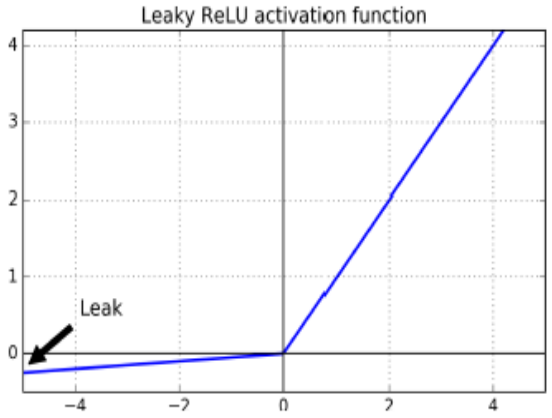
- Randomized Leaky ReLU (RReLU)
  - α is picked randomly during training, and it is fixed during testing.
- Parametric Leaky ReLU (PReLU)
  - Learn α during training (instead of being a hyperparameter).
- Exponential Linear Unit (ELU)

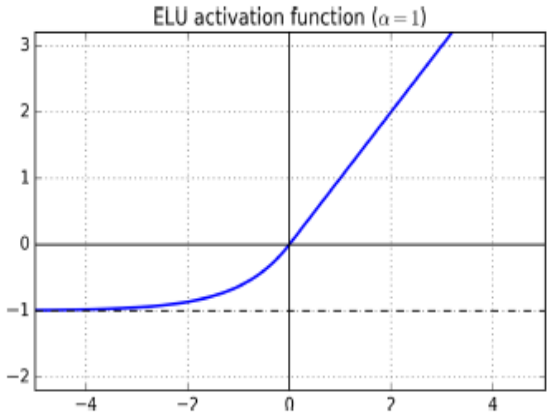
$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

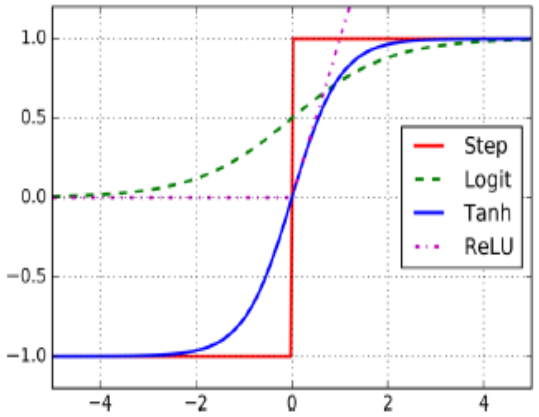


No saturating Activation Functions

- Which activation function should we use?
- In general logistic < tanh < ReLU < leaky ReLU (and its variants) < ELU
- If you care about runtime performance, then leaky ReLUs works better than ELUs.







Batch Normalization

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.
  - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.
- The technique consists of adding an operation in the model just before the activation function of each layer.

Batch Normalization

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

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

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

- μ_B: the empirical mean, evaluated over the whole mini-batch B.
- σ_B: the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B: the number of instances in the mini-batch.

Batch Normalization

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

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

- $\hat{\mathbf{x}}^{(i)}$: the zero-centered and normalized input.
- γ: the scaling parameter for the layer.
- β: the shifting parameter (offset) for the layer.
- ϵ: a tiny number to avoid division by zero.
- $\mathbf{z}^{(i)}$: the output of the BN operation, which is a scaled and shifted version of the inputs.

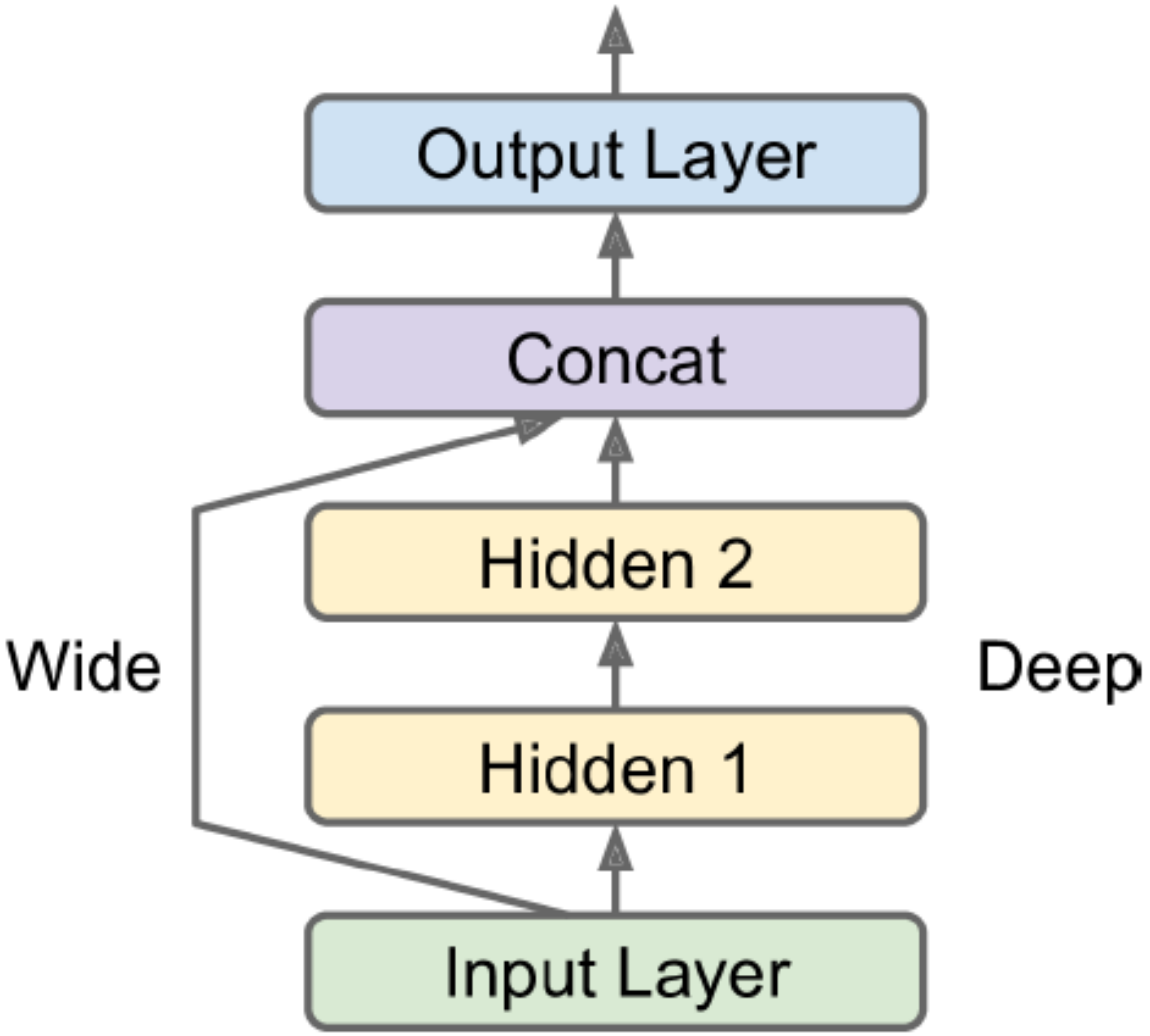
Gradient Clipping

- Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.

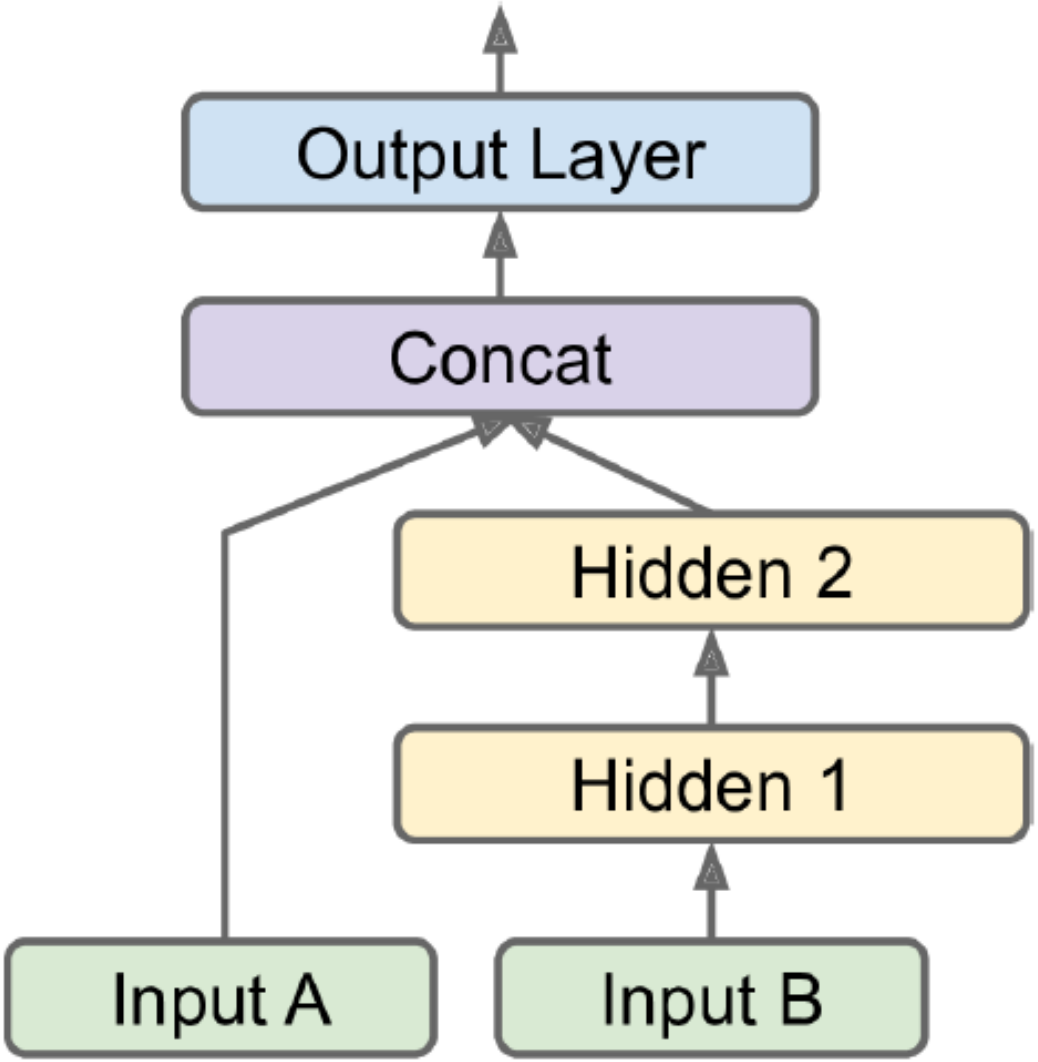
- In TensorFlow, the optimizer's `minimize()` function takes care of:
  1. Compute the gradients with `compute_gradients()`
  2. Apply the processed gradients with `apply_gradients()`

- To enable the gradient clipping, you must instead of calling `minimize()`, call:
  1. Compute the gradients with `compute_gradients()`
  2. Process the gradients as you wish.
  3. Apply the processed gradients with `apply_gradients()`

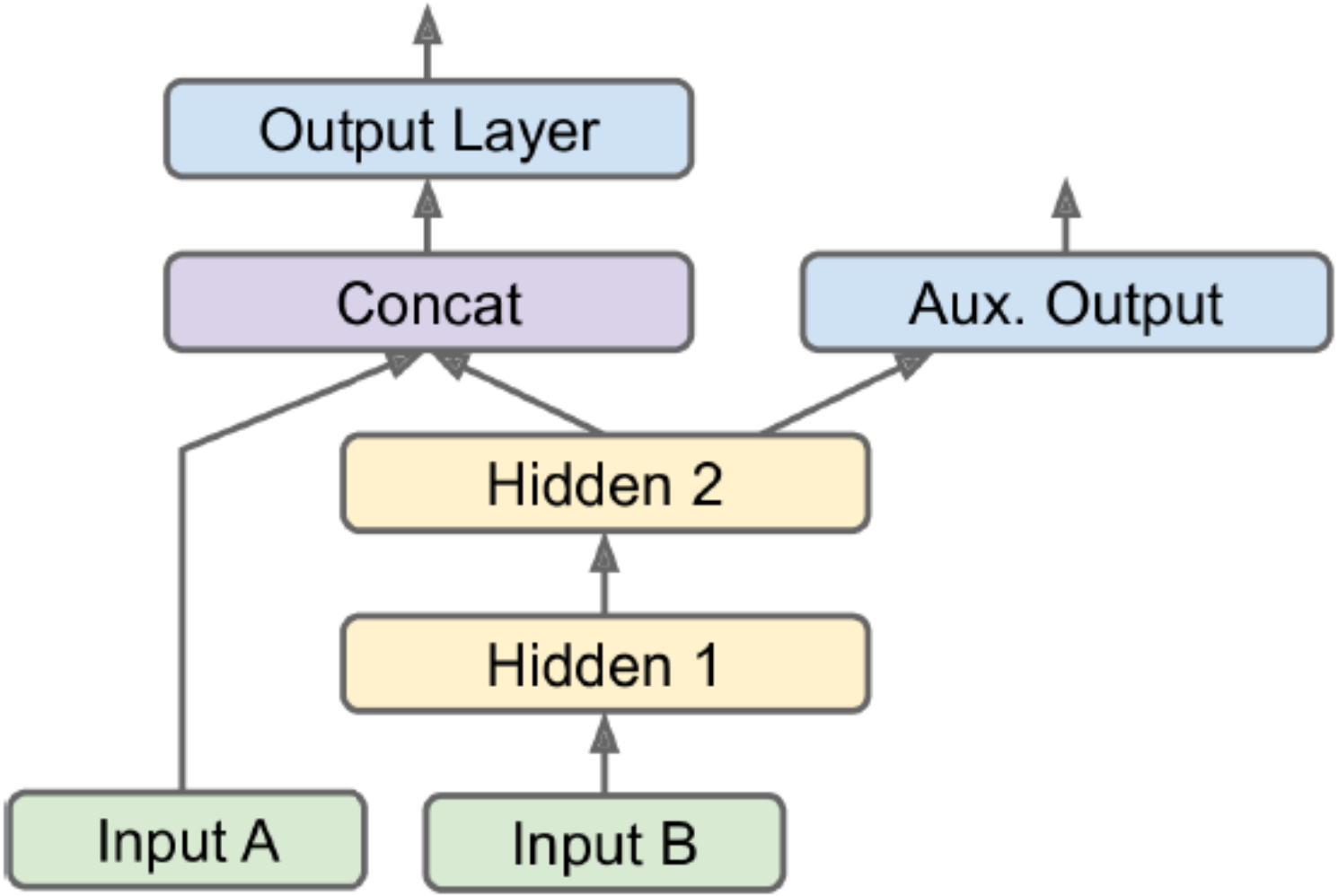
Building Complex Models



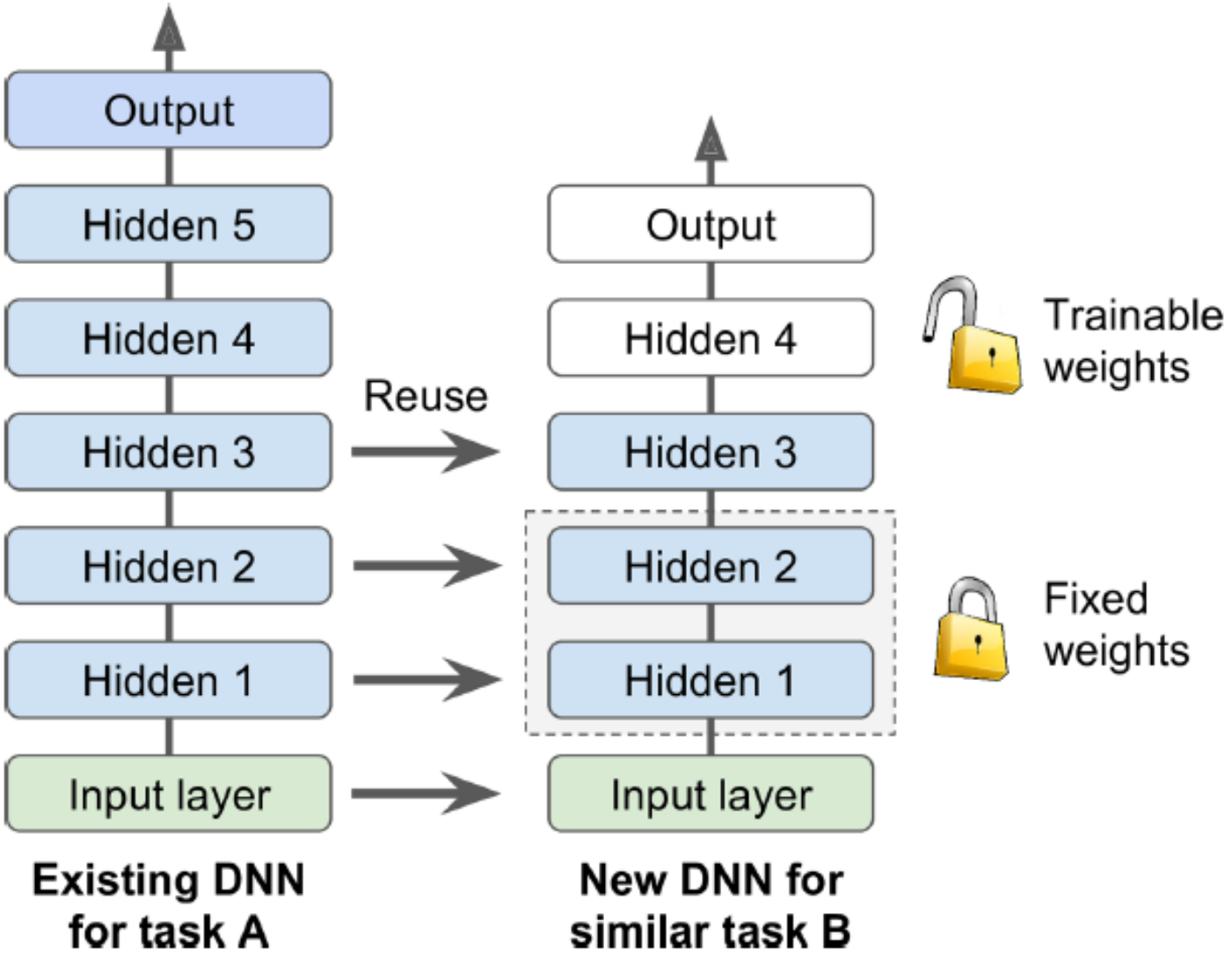
Handling multiple inputs



Handling multiple inputs/outputs



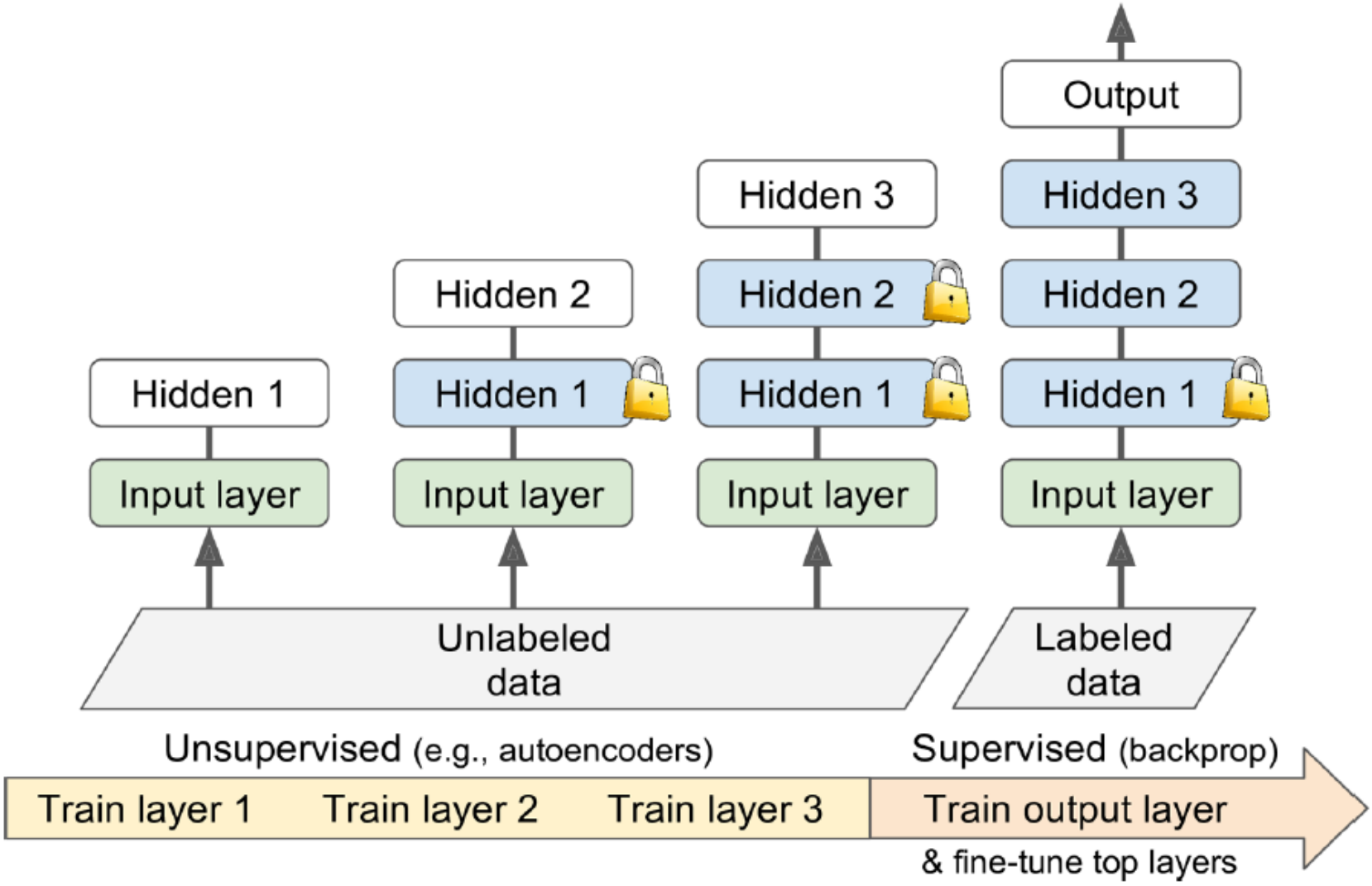
Reusing Pretrained Layers



Transfer Learning

	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

unsupervised training

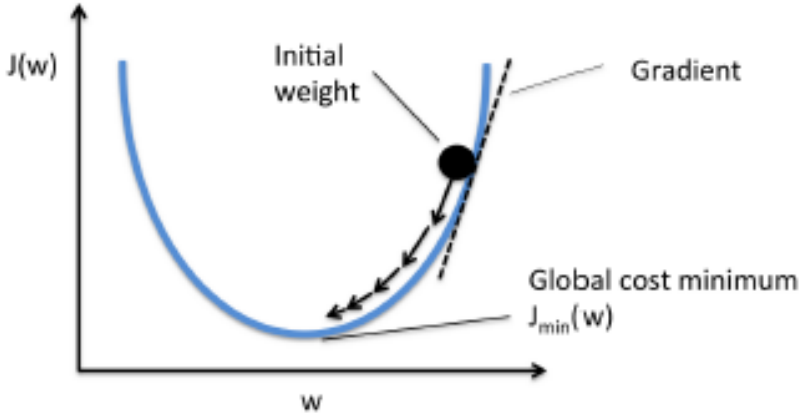


Optimization Algorithms

- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam Optimization

Momentum

- Momentum is a concept from physics: an object in motion will have a tendency to keep moving.
- It measures the resistance to change in motion.
  - The higher momentum an object has, the harder it is to stop it.
- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters $\mathbf{w}$ as motion: $w_i^{(next)} = w_i - \eta \frac{\partial J(\mathbf{w})}{\partial w_i}$
- We can thus use the concept of momentum to give the update process a tendency to keep moving in the same direction.
- It can help to escape from bad local minima pits.



Momentum

- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector **m**.

$$m_i = \beta m_i + \eta \frac{\partial J(\mathbf{w})}{\partial w_i}$$

- β is called momentum, ans it is between 0 and 1.
- Updates the weights by subtracting this momentum vector.

$$w_i^{(next)} = w_i - m_i$$

Nesterov Momentum

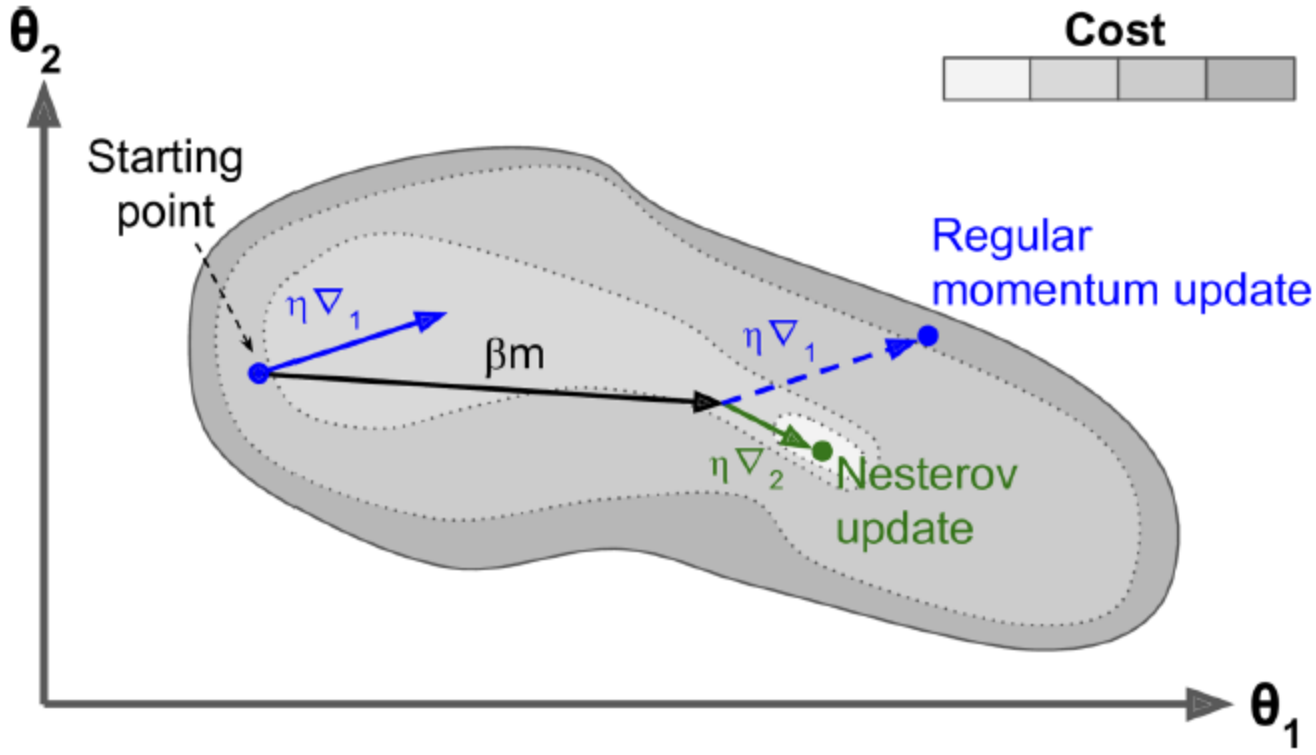
- Nesterov Momentum is a small variant to Momentum optimization.
- Faster than vanilla Momentum optimization.
- Measure the gradient of the cost function slightly ahead in the direction of the momentum (not at the local position).

$$\mathtt{m_i} = \beta \mathtt{m_i} + \eta \frac{\partial \mathtt{J}(\mathbf{w} + \beta \mathbf{m})}{\partial \mathtt{w_i}}$$

$$\mathtt{w_i^{(next)}} = \mathtt{w_i} - \mathtt{m_i}$$

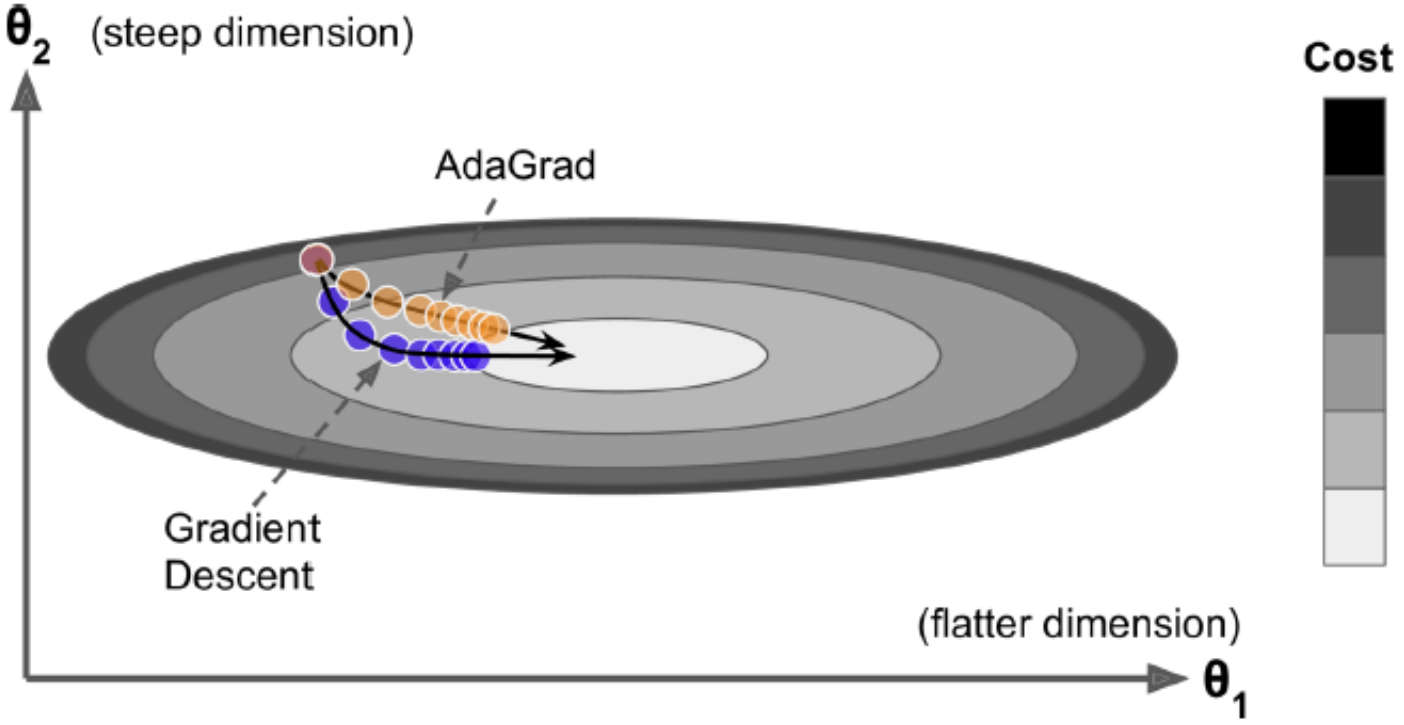
Nesterov Momentum

- $\nabla 1$ represents the gradient of the cost function measured at the starting point $\mathbf{w}$, and $\nabla 2$ represents the gradient at the point located at $\mathbf{w} + \beta \mathbf{m}$.



Ada Grad

- AdaGrad keeps track of a learning rate for each parameter.
- Adapts the learning rate over time (adaptive learning rate).



Ada Grad

- For each feature w_i, we do the following steps:

$$s_i = s_i + \left(\frac{\partial J(\mathbf{w})}{\partial w_i}\right)^2$$

$$w_i^{(next)} = w_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \frac{\partial J(\mathbf{w})}{\partial w_i}$$

- Parameters with large partial derivative of the cost have a rapid decrease in their learning rate.
- Parameters with small partial derivatives have a small decrease in their learning rate.

RMS Prop

- AdaGrad often stops too early when training neural networks.
- The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.
- The RMSProp fixed the AdaGrad problem.
- It is like the AdaGrad problem, but accumulates only the gradients from the most recent iterations (not from the beginning of training).
- For each feature w_i, we do the following steps:

$$s_i = \beta s_i + (1 - \beta)\left(\frac{\partial J(w)}{\partial w_i}\right)^2$$

$$w_i^{(next)} = w_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \frac{\partial J(w)}{\partial w_i}$$

Adam and Nadam Optimization

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.
- Like RMSProp, it keeps track of an exponentially decaying average of past squared gradients.

Adam and Nadam Optimization

$$1.\ \mathbf{m}^{(\text{next})} = \beta_1 \mathbf{m} + (1 - \beta_1)\nabla_{\mathbf{w}} J(\mathbf{w})$$

$$2.\ \mathbf{s}^{(\text{next})} = \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_{\mathbf{w}} J(\mathbf{w}) \otimes \nabla_{\mathbf{w}} J(\mathbf{w})$$

$$3.\ \mathbf{m}^{(\text{next})} = \frac{\mathbf{m}}{1 - \beta_1^T}$$

$$4.\ \mathbf{s}^{(\text{next})} = \frac{\mathbf{s}}{1 - \beta_2^T}$$

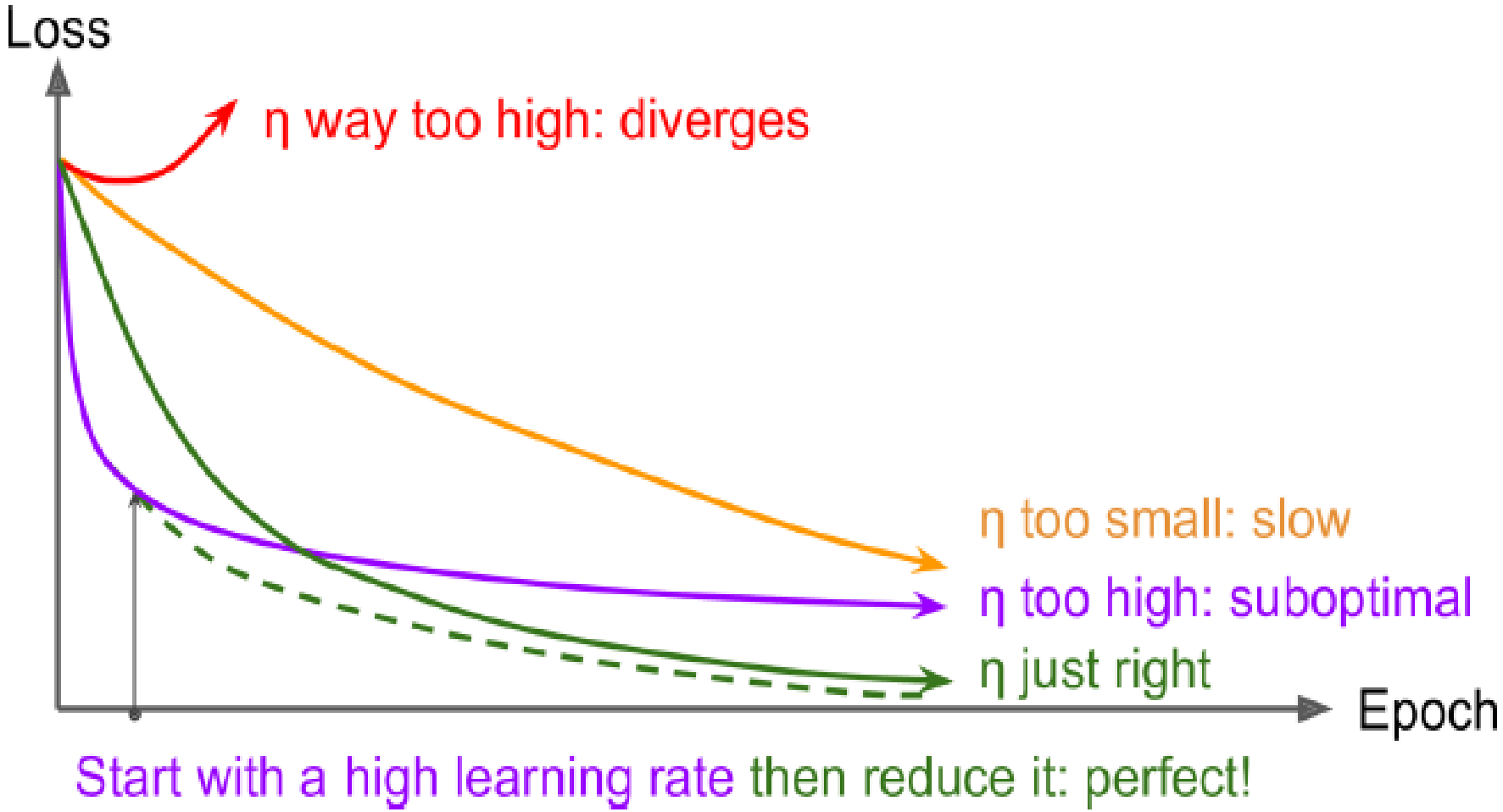
$$5.\ \mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$$

- $\otimes$ and $\oslash$ represents the represents the element-wise multiplication and division.
- Steps 1, 2, and 5: similar to both Momentum optimization and RMSProp.
- Steps 3 and 4: since $\mathtt{m}$ and $\mathtt{s}$ are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost $\mathtt{m}$ and $\mathtt{s}$ at the beginning of training.

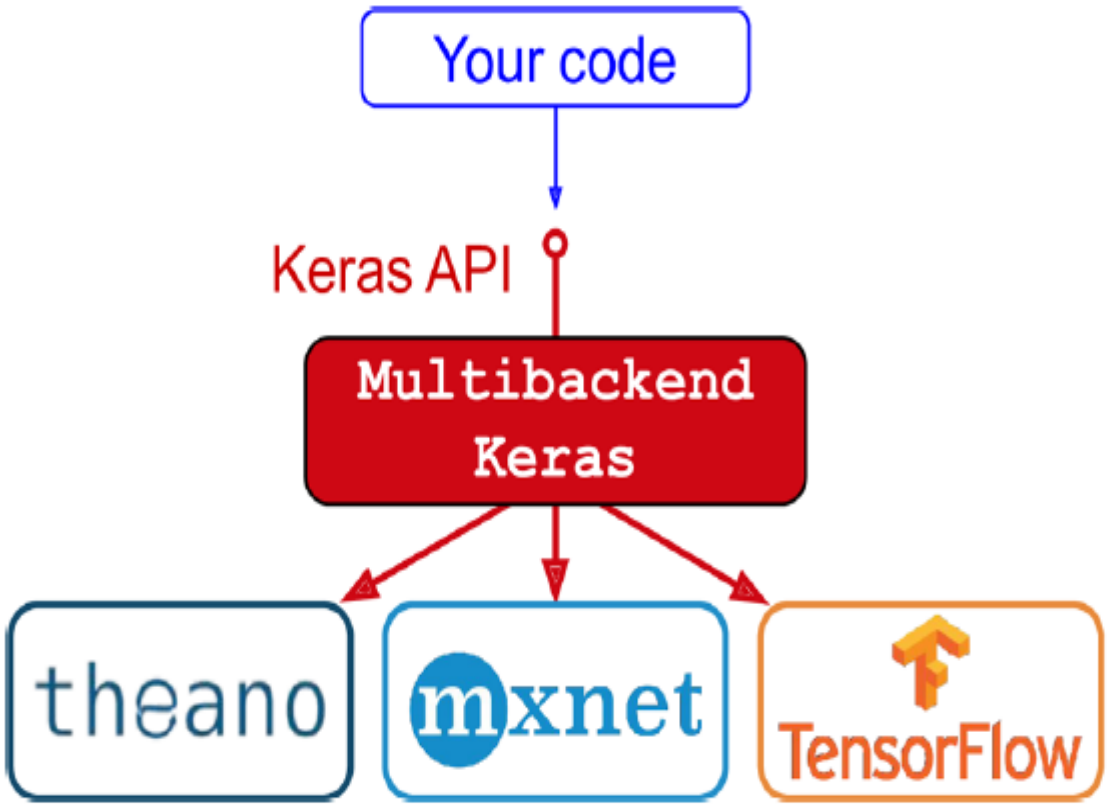
Optimizer comparison

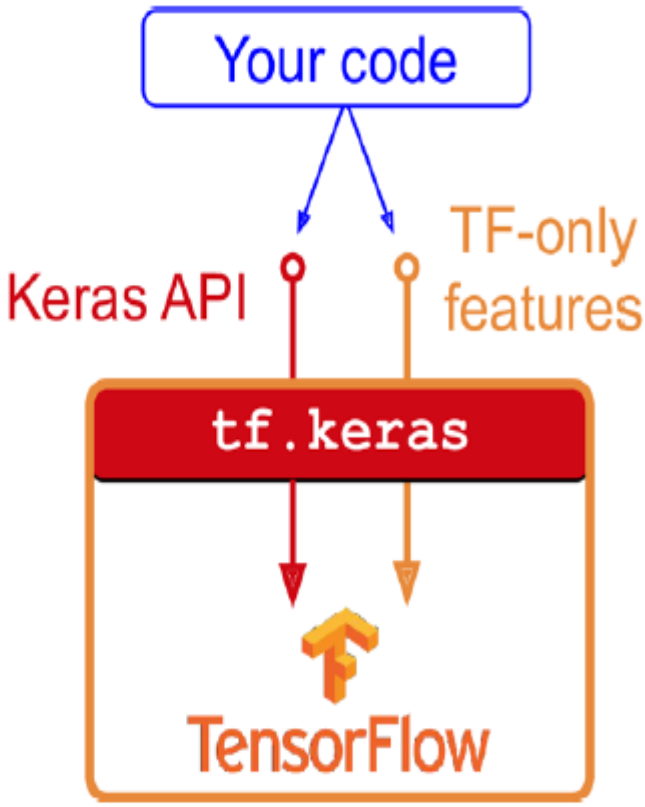
Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

Learning Rate Scheduling

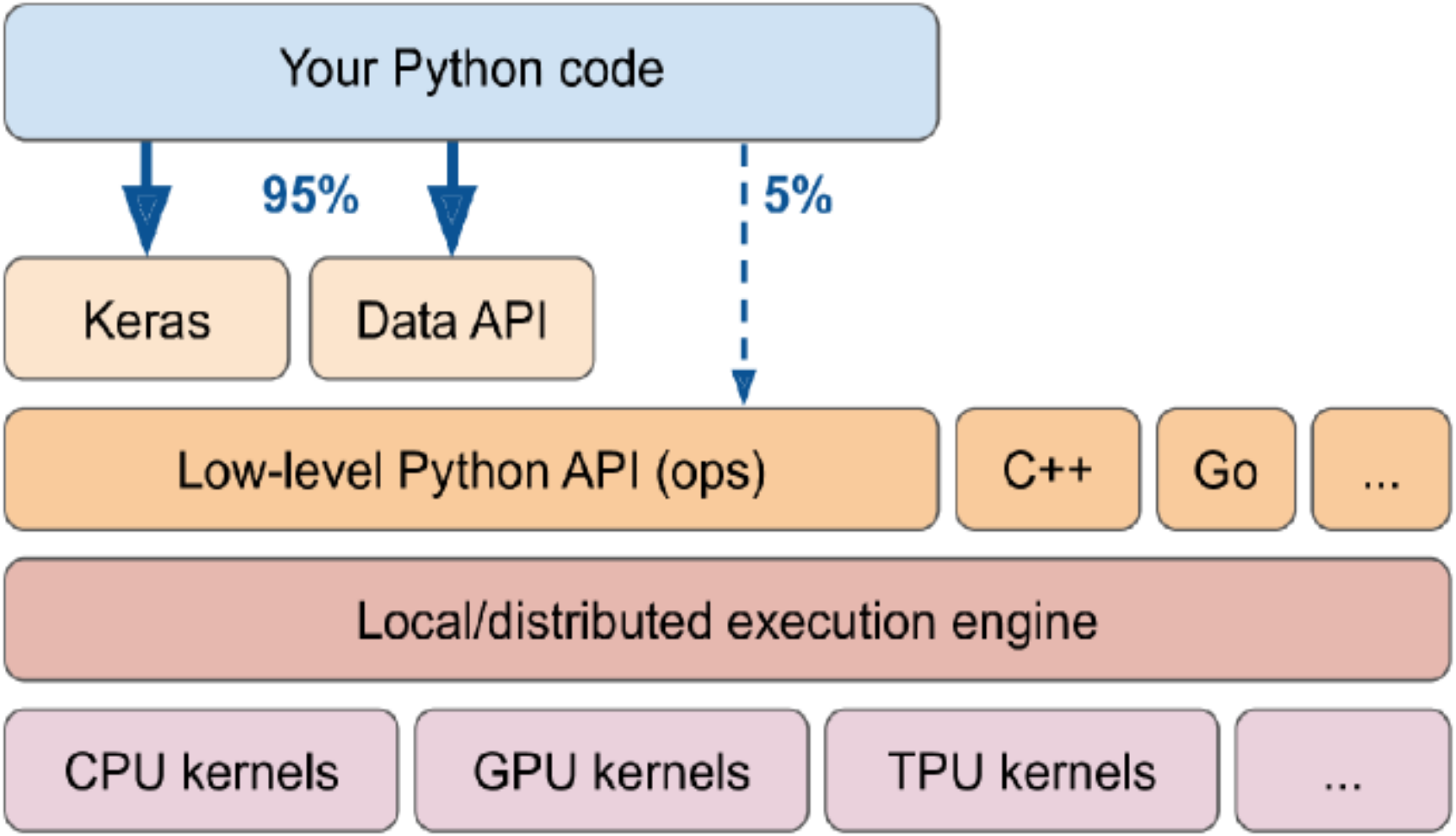


TensorFlow's tf.keras

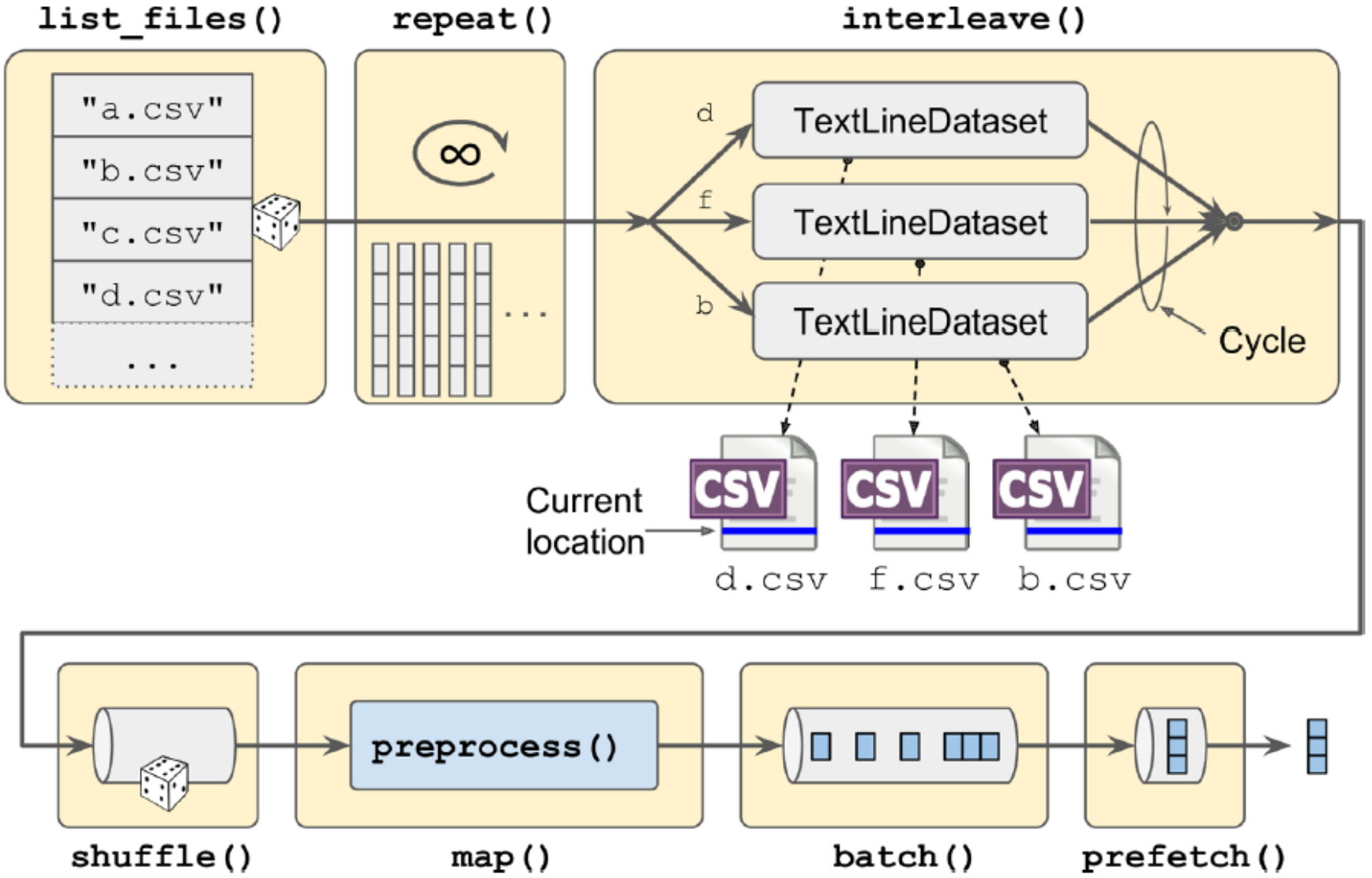




TensorFlow's architecture



Loading and preprocessing data



Prefetching