Lecture 3: Deep Neural Networks

Pranabendu Misra

Chennai Mathematical Institute

Advanced Machine Learning 2022

(based on slides by Madhavan Mukund)

- S is rep of real world distr
- PAC learn thm

Train set S

M ← learnt from S

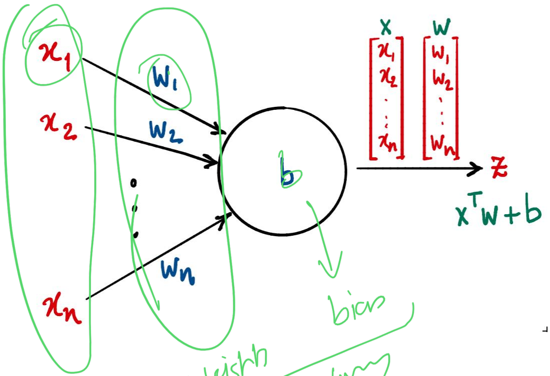
Gen Loss of M is low

- Perceptrons define linear separators
  $x^T w + b$
  - $x^T w + b > 0$, classify Yes $(+1)$
  - $x^T w + b < 0$, classify No (-1)

$x \in \mathbb{R}^n$

$n = h \times w \times 3$

$\boxed{x \cdot w + b}$



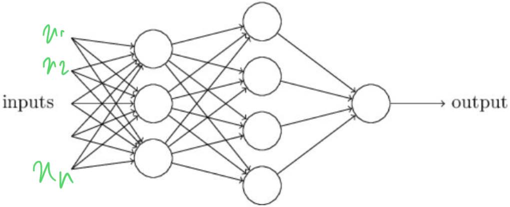
bias

weights learnt during 'Training'

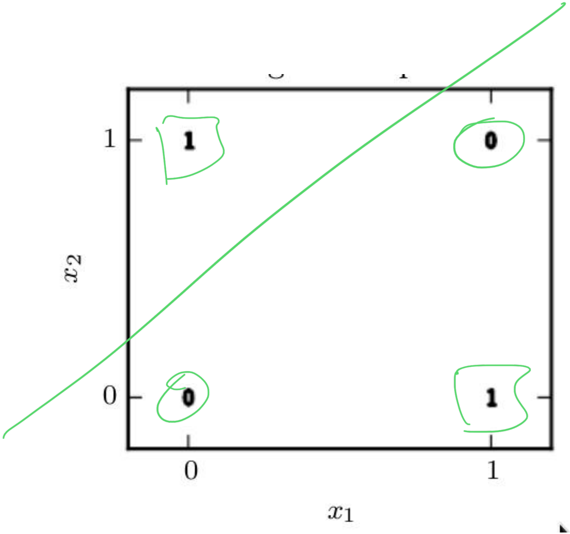
Linear separators and perceptrons

- Perceptrons define linear separators
  $x^T w + b$
  - $x^T w + b > 0$, classify Yes $(+1)$
  - $x^T w + b < 0$, classify No (-1)
- **Unfortunately!** Network of perceptrons still defines only a linear separator

Network of linear perceptrons



- Perceptrons define linear separators
  $x^T w + b$
  - $x^T w + b > 0$, classify Yes $(+1)$
  - $x^T w + b < 0$, classify No (-1)
- **Unfortunately!** Network of perceptrons still defines only a linear separator
- Linear separators cannot describe XOR



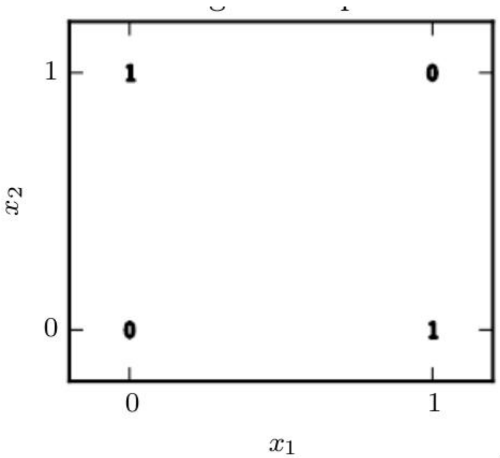
Linear separators and perceptrons

- Perceptrons define linear separators
  $x^T w + b$
  - $x^T w + b > 0$, classify Yes $(+1)$
  - $x^T w + b < 0$, classify No (-1)

- **Unfortunately!** Network of perceptrons still defines only a linear separator

- Linear separators cannot describe XOR

We need non-linearity!



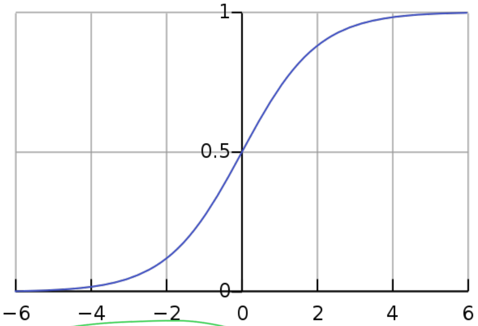
Linear separators and perceptrons

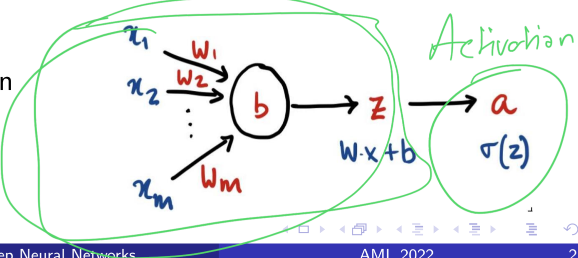
- Perceptrons define linear separators
  $x^T w + b$
  - $x^T w + b > 0$, classify Yes $(+1)$
  - $x^T w + b < 0$, classify No (-1)
- **Unfortunately!** Network of perceptrons still defines only a linear separator
- Linear separators cannot describe XOR

We need non-linearity!

- Introduce a non-linear activation function
  - Traditionally sigmoid,
    $\sigma(z) = 1/(1 + e^{-z})$

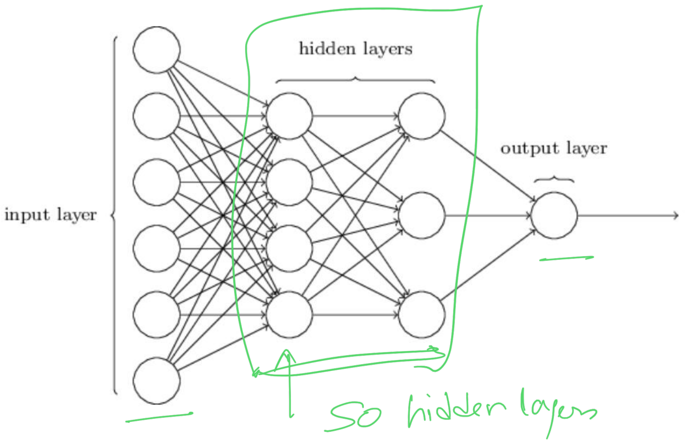
This is a neuron!





(Feed forward) Neural networks

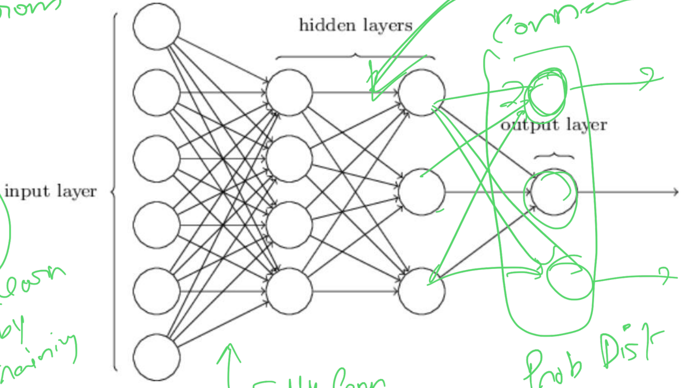
- Acyclic network of perceptrons with non-linear activation functions
  - Universal Approximation Theorem: With just 1 hidden layer, a neural network can approximate any function for any degree of precision.
  - For a function f, it is possible to construct such a neural network.
    For example the XOR function.



(Feed forward) Neural networks

- Acyclic network of perceptrons with non-linear activation functions → neurons
  - The Structure of the network and the value of it's Parameters θ (weights and biases of each neuron).
  - **Objective:** Given a training set S, compute a neural network with *low generalization loss.* learn by training

    of the chosen structure
  - We can estimate the generalization loss using a test set T.



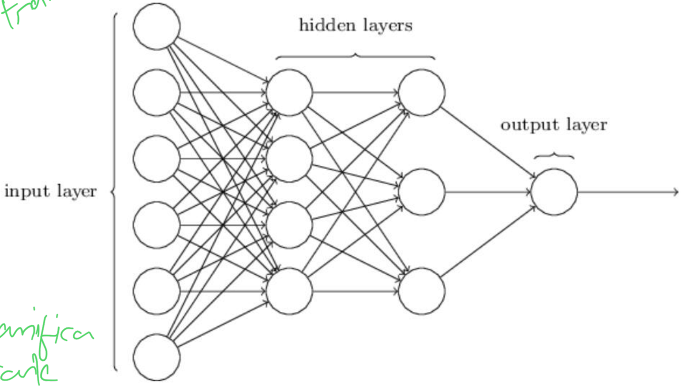
(Feed forward) Neural networks

- How do we compute a neural network?
  - Choose the structure of the neural network, based on the ML task at hand.
  - Initially set the weights and biases of neurons to random numbers.
  - Choose a loss-function $\ell(\theta, S)$ on the output of the neural network, e.g. Cross-Entropy Loss
  - Optimization problem: Given S find the values for θ with least $\ell(\theta, S)$.

also train

Important

classification task



(Feed forward) Neural networks

- How do we compute a neural network?
  - Optimization problem: Given S find the values for θ with least $\ell(\theta, S)$.

Highly Non-Trivial Problem!

(Feed forward) Neural networks

- How do we compute a neural network?
  - Optimization problem: Given S find the values for θ with least $\ell(\theta, S)$.

Parameters

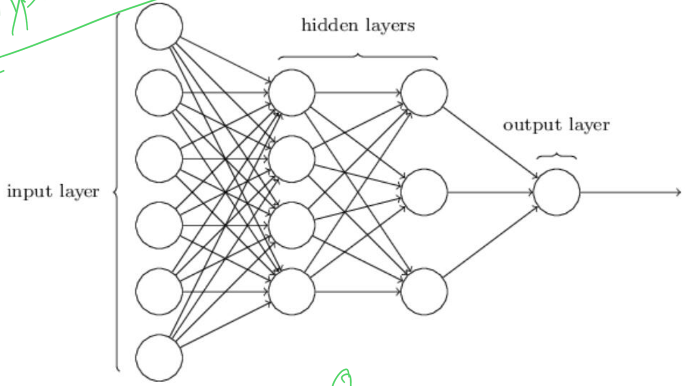
Hyper Parameters

Highly Non-Trivial Problem!

- We use the most basic optimization method:

Gradient Descent

- Update θ so that $\boxed{\ell(\theta, S)}$ decreases $\Longrightarrow$ $\theta \leftarrow \boxed{\theta - \alpha \cdot \frac{d\ell(\theta,S)}{d\theta}}$
- α is the learning rate



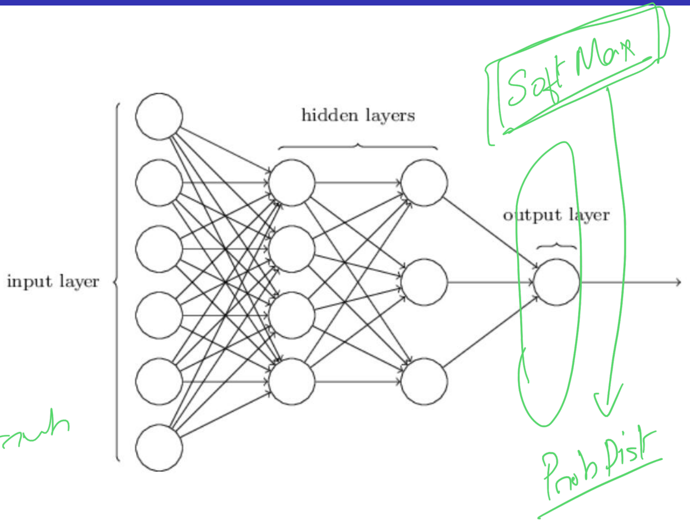
θ

θ

(Feed forward) Neural networks

- Acyclic network of perceptrons with non-linear activation functions
- Ingredients
  - Output layer activation function
  - Loss function for gradient descent on S
  - Hidden layer activation functions
  - Network architecture: Interconnection of layers
  - Initial values of weights and biases

Struct



Training a neural network

- Backpropagation — efficient implementation of gradient descent for neural networks

L

w_i

$\partial L / \partial w_i \;\; \forall i \in \theta$

$$\theta \leftarrow \theta - \alpha \frac{d\,\ell(\theta, S)}{d\theta}$$

Compute z, a

Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$

Update for $\{w_i\}$

$$\boxed{w_i \leftarrow w_i - \alpha \frac{\partial L}{\partial w_i}}$$

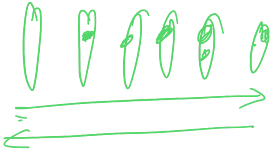
How do we compute this?

$$\frac{d}{dx}\left(f(g(h(x)))\right)$$

$$\frac{d}{dx}$$

Training a neural network

- Backpropagation — efficient implementation of gradient descent for neural networks
- Forward pass, compute outputs, activation values
- Backward pass, use chain rule to compute all gradients in one scan



Compute z, a $\longrightarrow$

$\longleftarrow$ Compute $\frac{\partial C}{\partial z_k^l}, \frac{\partial C}{\partial w_{kj}^l}, \frac{\partial C}{\partial b_k^l}$

$$f(x) = \alpha x + \beta$$
$$g(x) = \gamma x + \delta$$
$$l(x) = g(f(x))$$
$$\partial l/\partial\alpha = \underbrace{\partial g/\partial f}\cdot\underbrace{\partial f/\partial\alpha} = \gamma \cdot x$$

Training a neural network

- Backpropagation — efficient implementation of gradient descent for neural networks
- Forward pass, compute outputs, activation values
- Backward pass, use chain rule to compute all gradients in one scan
- Stochastic gradient descent (SGD)
  - Update parameters in minibatches
  - Epoch: set of minibatches that covers entire training data

S' $|S'| = 100$

small random sample taken from S

Compute z, a $\longrightarrow$

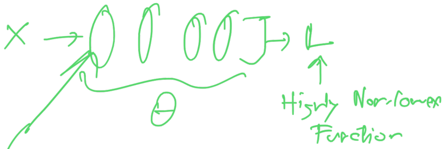
$\longleftarrow$ Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$

$\boxed{\frac{d\, l(\theta, S')}{d\theta}}$ $\boxed{\frac{d\, l(\theta, S)}{d\theta}}$

10000 data points.

Training a neural network

- Backpropagation — efficient implementation of gradient descent for neural networks
- Forward pass, compute outputs, activation values
- Backward pass, use chain rule to compute all gradients in one scan
- Stochastic gradient descent (SGD)
  - Update parameters in minibatches
  - Epoch: set of minibatches that covers entire training data
- Difficulties: slow convergence, vanishing and exploding gradients



Compute z, a $\longrightarrow$

$\longleftarrow$ Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$

$\boxed{w_i \leftarrow w_i - \alpha \, \partial L / \partial w_i}$

Learning Rate $\rightarrow$ α is too small $\rightarrow \approx 0$

Unstable gradients

- Vanishing gradients — gradients become smaller towards lower layers (closer to input)
  - Gradient descent updates leave these layers' parameters virtually unchanged

Compute z, a

$\longrightarrow$

$\longleftarrow$

Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$

Unstable gradients

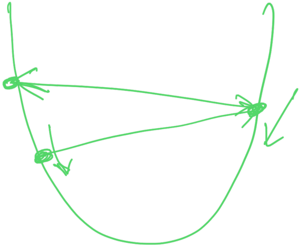
- Vanishing gradients — gradients become smaller towards lower layers (closer to input)
  - Gradient descent updates leave these layers' parameters virtually unchanged
- Also exploding gradients, recurrent neural networks with feedback edges

Compute z, a

$\longrightarrow$

$\longleftarrow$

Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$



$\partial L / \partial w_i$ is too long

$w_i \leftarrow w_i - \underbrace{\alpha}_{} \, \partial L / \partial w_i$

Unstable gradients

- Vanishing gradients — gradients become smaller towards lower layers (closer to input)
  - Gradient descent updates leave these layers' parameters virtually unchanged
- Also exploding gradients, recurrent neural networks with feedback edges
- In general, unstable gradients, different layers learn at different speeds

Compute z, a

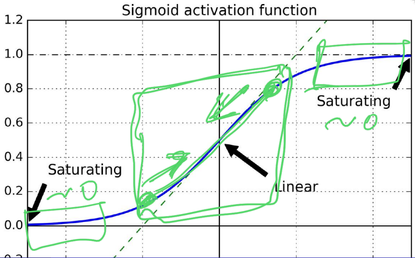
Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$

Unstable gradients

- Vanishing gradients — gradients become smaller towards lower layers (closer to input)
  - Gradient descent updates leave these layers' parameters virtually unchanged
- Also exploding gradients, recurrent neural networks with feedback edges
- In general, unstable gradients, different layers learn at different speeds
- [Xavier Glorot and Joshua Bengio, 2010]
  - Random initialization, traditionally Gaussian distribution $\mathcal{N}(0, 1)$
  - Variance keeps increasing going forward
  - Saturating sigmoid function

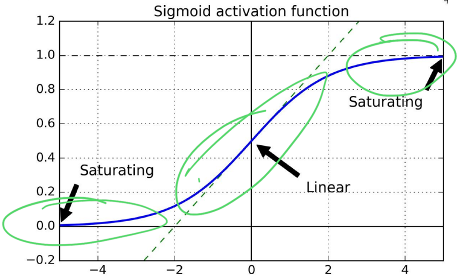
Compute z, a

Compute $\frac{\partial C}{\partial z^l_k}, \frac{\partial C}{\partial w^l_{kj}}, \frac{\partial C}{\partial b^l_k}$



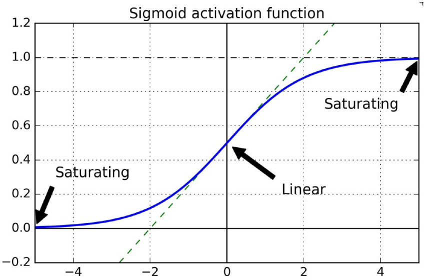
Initializing neural networks

- Want "signal" to flow well in both directions during backpropagation
  - Signal should not die out, explode, saturate



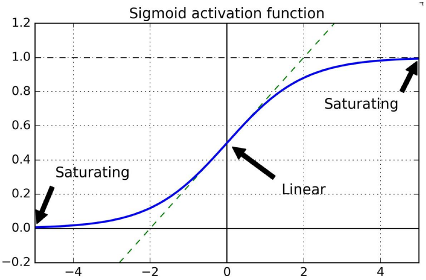
Initializing neural networks

- Want "signal" to flow well in both directions during backpropagation
  - Signal should not die out, explode, saturate
- [Glorot,Bengio] Gradients should have equal variance before and after flowing through a layer in both directions
  - Equal variance requires $fan_{in} = fan_{out}$

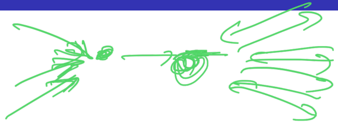


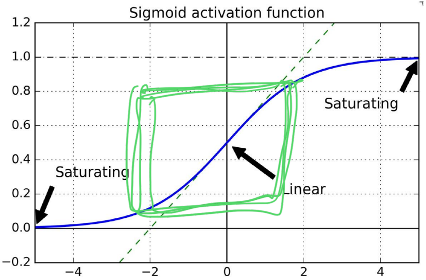
Initializing neural networks

- Want "signal" to flow well in both directions during backpropagation
  - Signal should not die out, explode, saturate
- [Glorot,Bengio] Gradients should have equal variance before and after flowing through a layer in both directions
  - Equal variance requires $fan_{in} = fan_{out}$
- Let $fan_{avg} = (fan_{in} + fan_{out})/2$

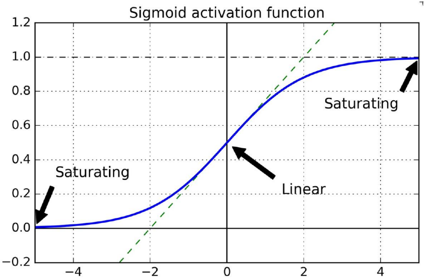


- Want "signal" to flow well in both directions during backpropagation
  - Signal should not die out, explode, saturate
- [Glorot,Bengio] Gradients should have equal variance before and after flowing through a layer in both directions
  - Equal variance requires $fan_{in} = fan_{out}$
- Let $fan_{avg} = (fan_{in} + fan_{out})/2$
- Initialize with
  - Gaussian, $\boxed{\mathcal{N}(0, 1/fan_{avg})}$
  - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\dfrac{3}{fan_{avg}}}$



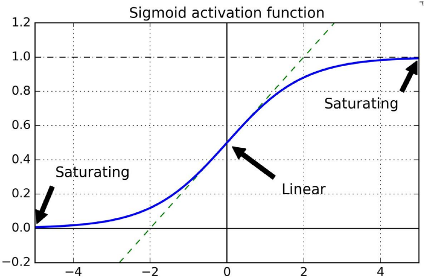


- Let $fan_{avg} = (fan_{in} + fan_{out})/2$
- Initialize with
  - Gaussian, $\mathcal{N}(0, 1/fan_{avg})$
  - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\dfrac{3}{fan_{avg}}}$



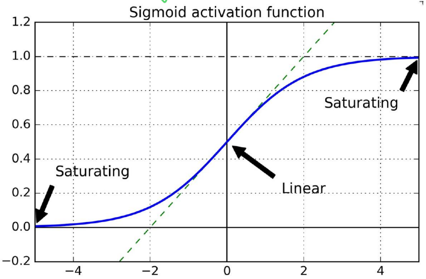
Initializing neural networks

- Let $fan_{avg} = (fan_{in} + fan_{out})/2$
- Initialize with
  - Gaussian, $\boxed{\mathcal{N}(0, 1/fan_{avg})}$
  - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\frac{3}{fan_{avg}}}$
- [Yann LeCun, 1990s] earlier proposed the same with fan_{avg} replaced by fan_{in}
  - Equivalent if $fan_{in} = fan_{out}$



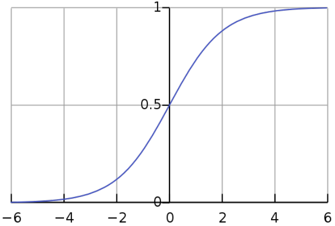
Initializing neural networks

- Let $fan_{avg} = (fan_{in} + fan_{out})/2$
- Initialize with
  - Gaussian, $\mathcal{N}(0, 1/fan_{avg})$
  - Uniform, $\mathcal{U}(-r, r)$, $r = \sqrt{\frac{3}{fan_{avg}}}$
- [Yann LeCun, 1990s] earlier proposed the same with fan_{avg} replaced by fan_{in}
  - Equivalent if $fan_{in} = fan_{out}$
- Other choices for specific activation function
  - ReLU, [He et al, 2015], $\mathcal{N}(0, 2/fan_{in})$



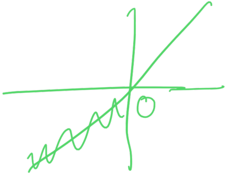
Non-saturating activation functions

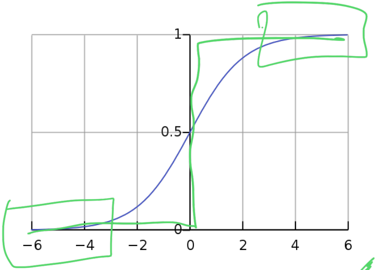
- Sigmoid was initially chosen as a "smooth" step

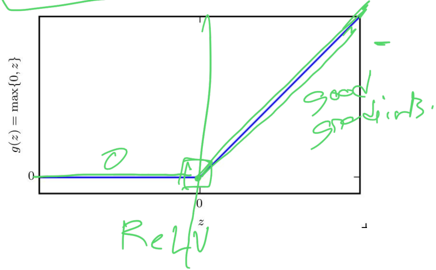


Non-saturating activation functions

- Sigmoid was initially chosen as a "smooth" step
- Rectified linear unit (ReLU): $g(z) = \max(0, z)$
  - Fast to compute
  - Non-differentiable point not a bottleneck

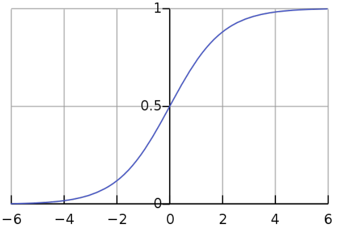


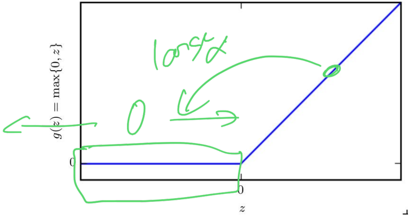




Non-saturating activation functions

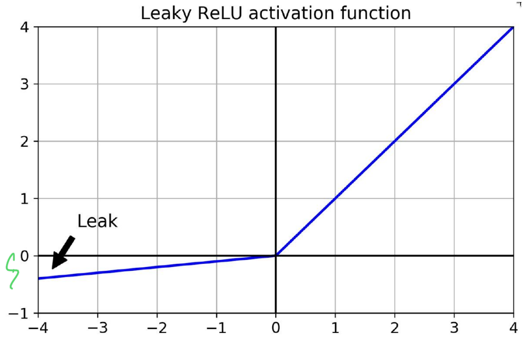
- Sigmoid was initially chosen as a "smooth" step
- Rectified linear unit (ReLU): $g(z) = \max(0, z)$
  - Fast to compute
  - Non-differentiable point not a bottleneck
- "Dying ReLU"
  - Neuron dies — weighted sum of outputs is negative for all training samples
  - With a large learning rate, half the network may die!





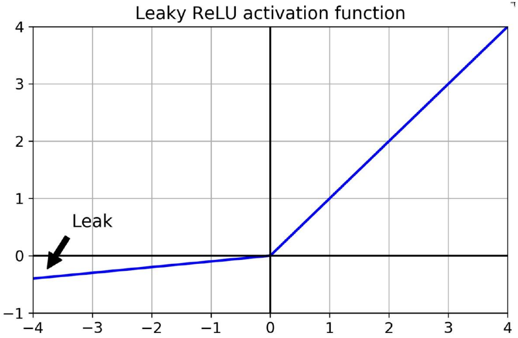
Non-saturating activation functions

- Leaky ReLU, $\max(\alpha z, z)$
  - "Leak" α is a hyperparameter



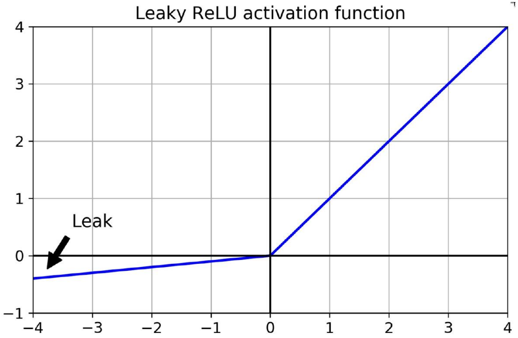
Non-saturating activation functions

- Leaky ReLU, $\max(\alpha z, z)$
  - "Leak" α is a hyperparameter
- RReLU — random leak
  - Pick α from a random range during training
  - Fix to an average value when testing
  - Seems to work well, act as a regularizer



Non-saturating activation functions

- Leaky ReLU, $\max(\alpha z, z)$
  - "Leak" α is a hyperparameter
- RReLU — random leak
  - Pick α from a random range during training
  - Fix to an average value when testing
  - Seems to work well, act as a regularizer
- PReLU — parametric ReLU [He et al, 2015]
  - α is learned during training
  - Often outperforms ReLU, but could lead to overfitting

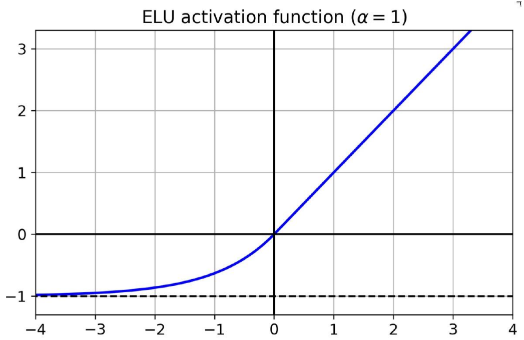


Non-saturating activation functions

- ELU — Exponential Linear Unit
  [Clevert et al, 2015]

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

  - Training converges faster
  - Computing exponential is slower
  - In practice, slower than ReLU

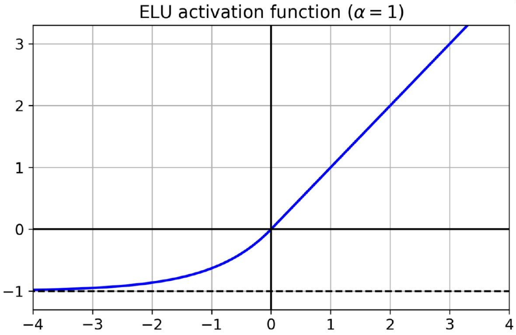


Non-saturating activation functions

- SELU — Scaled ELU
  [Klambauer et al, 2017]

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

  - Self-normalizing — output of each layer preserves mean 0 and standard deviation 1 during training
  - Use LeCun initialization, $\mathcal{N}(0, 1/fan_{in})$



Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input

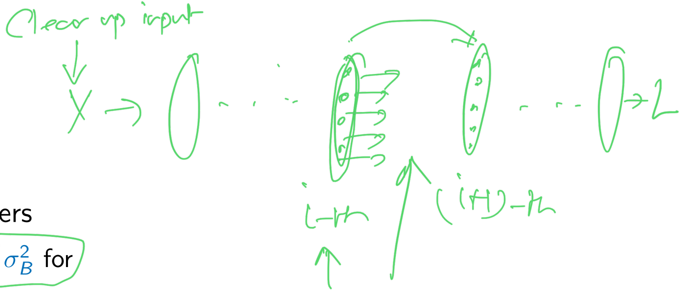
    $$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input

$$\boxed{\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}}$$ + Random Noise

  - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$



Maybe Clear Up here too

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input

    $$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
  - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
  - Learn optimal scaling and shifting parameters for each layer

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input
    $$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
  - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
  - Learn optimal scaling and shifting parameters for each layer
- At input, BN layer avoids need for standardizing

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input

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

  - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
  - Learn optimal scaling and shifting parameters for each layer

- At input, BN layer avoids need for standardizing
- Difficulties

→ not much variation

  - Mean and variance differ across minibatches
  - How to estimate parameters for entire dataset?
  - Practical solution: maintain a moving average of means and standard deviations for each layer

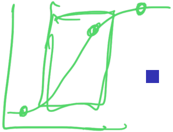
Good estimate for current minibatch.

Batch normalization [Joffe, Szegedy 2015]

- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input

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

  - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
  - Learn optimal scaling and shifting parameters for each layer



- At input, BN layer avoids need for standardizing
- Difficulties
  - Mean and variance differ across minibatches
  - How to estimate parameters for entire dataset?
  - Practical solution: maintain a moving average of means and standard deviations for each layer
- Batch normalization greatly speeds up learning rate

Batch normalization [Joffe, Szegedy 2015]

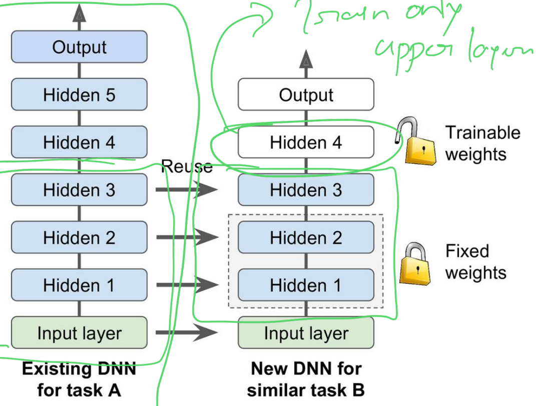
- Good activation function and initialization mitigates vanishing/exploding gradients
- May still recur during training
- Add batch normalization (BN) layers
  - Estimate mean μ_B and variance σ_B^2 for inputs across minibatch
  - Zero-centre and normalize each input
    $$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
  - Scale and shift $z_i = \lambda \cdot \hat{x}_i + \beta$
  - Learn optimal scaling and shifting parameters for each layer
- At input, BN layer avoids need for standardizing
- Difficulties
  - Mean and variance differ across minibatches
  - How to estimate parameters for entire dataset?
  - Practical solution: maintain a moving average of means and standard deviations for each layer
- Batch normalization greatly speeds up learning rate
- Even works as a regularizer!

Transfer learning

- Reuse trained layers across deep neural networks (DNNs)

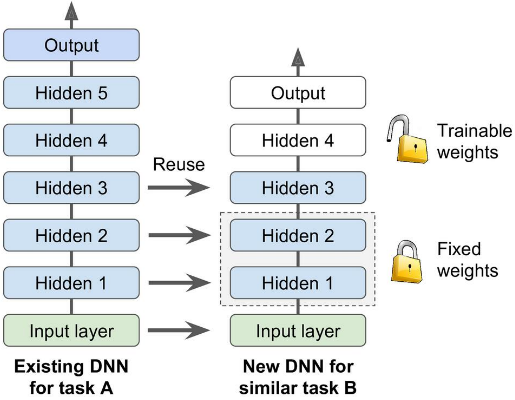
Recognize first features

Train only upper layers



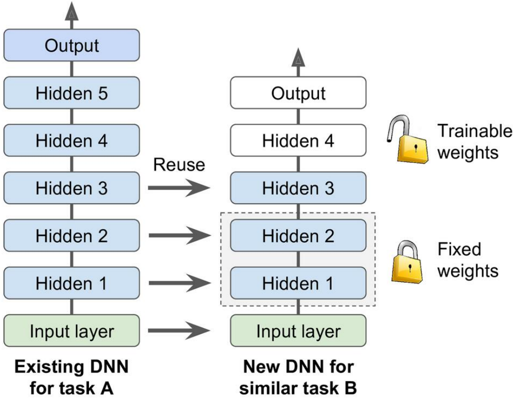
Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, . . .)



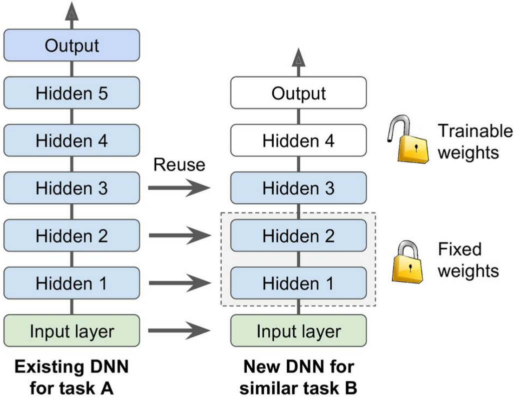
Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, . . .)
- New DNN to classify types of vehicles



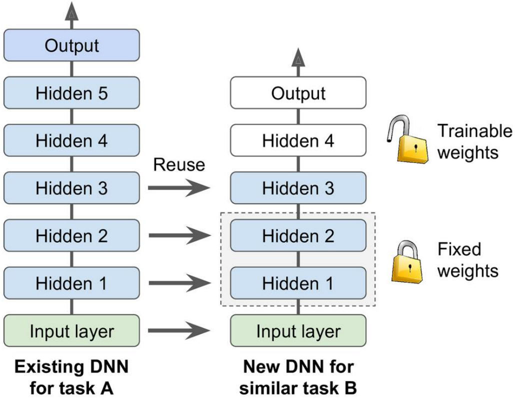
Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, ...)
- New DNN to classify types of vehicles
- Tasks similar, even overlapping



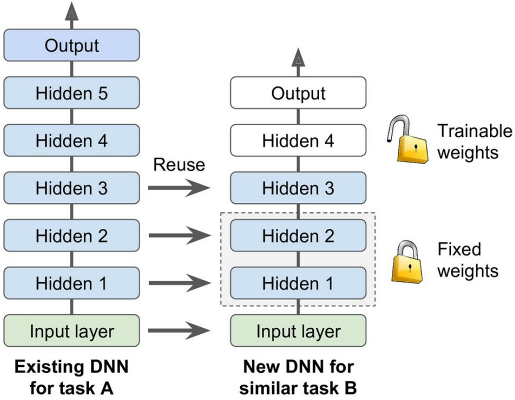
Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, . . .)
- New DNN to classify types of vehicles
- Tasks similar, even overlapping
- Lower layers identify basic features, upper layers combine them to classify



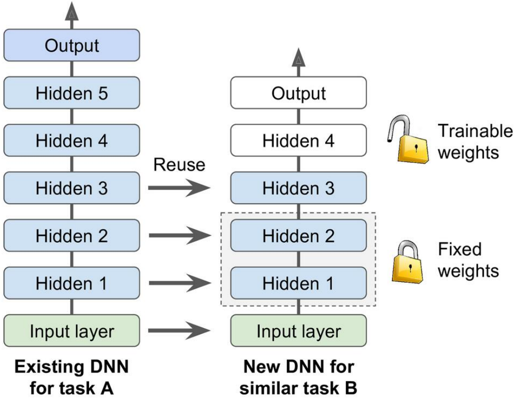
Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, ...)
- New DNN to classify types of vehicles
- Tasks similar, even overlapping
- Lower layers identify basic features, upper layers combine them to classify
- Freeze weights of lower layers, re-learn upper layers



Transfer learning

- Reuse trained layers across deep neural networks (DNNs)
- Old DNN trained on images of daily objects (animals, plants, vehicles, . . .)
- New DNN to classify types of vehicles
- Tasks similar, even overlapping
- Lower layers identify basic features, upper layers combine them to classify
- Freeze weights of lower layers, re-learn upper layers
- Unfreeze in stages to determine how much to reuse



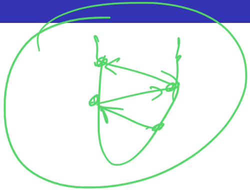
Still to come

- Optimizing rate of updates in backpropagation

$$w_i \leftarrow w_i - \alpha \boxed{\partial L / \partial w_i}$$

Very important value

- α too small $\longrightarrow$ long time to train
- α too large $\longrightarrow$ unable to properly minimize loss



- Want a large α at start of training
- Small α towards end of training.

Still to come

- Optimizing rate of updates in backpropagation
- How problematic are local minima?

Still to come

- Optimizing rate of updates in backpropagation
- How problematic are local minima?
- Identifying and dealing with unstable gradients