NEURAL NETWORKS 2

DATA/MSML 603: Principles of Machine Learning

Neural Network Learning/Training

Optimization for Parameter Learning

- Recall that the parameters to be learned in a neural network with L layers are

$$\boldsymbol{\theta} = \{\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(L-1)}\}$$

  - $\boldsymbol{\theta}^{(l)}, l = 1, \ldots, L-1,$ is an $s_{l+1} \times (s_l + 1)$ matrix, where s_l is the number of nodes in layer l

- In essence, if we knew the distribution $\mathcal{D}$ of samples, we would like to find a model

$$\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} \; \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}} \left[L(\mathbf{x}, y; \boldsymbol{\theta})\right]$$

  - $L(\cdot, \cdot; \boldsymbol{\theta})$ - loss function for fixed $\boldsymbol{\theta}$ (depends on the problem)

- Issue: In general, we do not know the distribution $\mathcal{D}$

Optimization for Parameter Learning

- Solution: Replace the expectation with "sample mean" using data

$$\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}^i, y^i; \boldsymbol{\theta})$$

  - $L(\mathbf{x}^i, y^i; \boldsymbol{\theta})$ - loss for the i-th sample
  - When samples are drawn i.i.d., the sample mean converges to the expected loss with increasing sample size (under some mild technical condition)

Law of Large Numbers

- Suppose that $X_1, X_2, X_3, \ldots$, are i.i.d. random variables with a common distribution F
- (Weak) Law of Large Numbers: Let $\mathbb{E}[X_1] = \mu$. Then,

$$S_n := \frac{\sum_{i=1}^{n} X_i}{n} \to \mu \quad \text{as } n \to \infty$$

in probability, i.e., for all $\epsilon > 0$

$$\lim_{n \to \infty} \mathbb{P}\left(|S_n - \mu| > \epsilon\right) = 0$$

Central Limit Theorem

- In addition, assume $Var(X_1) = \sigma < \infty$. Then,

$$\frac{\sum_{i=1}^{n} X_i - n \cdot \mu}{\sqrt{n}\sigma} \xrightarrow{D} N(0,1)$$

Loss Functions

1. Regression problem: squared error

$$L(\mathbf{x}, y; \boldsymbol{\theta}) = \big(y - h_{\boldsymbol{\theta}}(\mathbf{x})\big)^2$$

2. Classification problem: cross-entropy (aka logarithmic loss or log loss)

$$L(\mathbf{x}, y; \boldsymbol{\theta}) = -\ln\big(g_y(h_{\boldsymbol{\theta}}(\mathbf{x}))\big) = -\ln\left(\frac{\exp(h_{\boldsymbol{\theta},y}(\mathbf{x}))}{\sum_{j=1}^{m}\exp(h_{\boldsymbol{\theta},j}(\mathbf{x}))}\right) = -z_y + \ln\left(\sum_{j=1}^{m}\exp(z_j)\right)$$

- $z_j = h_{\boldsymbol{\theta},j}(\mathbf{x})$ - output of the last layer before softmax (which is assumed here)

Training Neural Networks

- Training performed to find a set of parameters which minimize the aggregate loss over the dataset

$$\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}^i, y^i; \boldsymbol{\theta}) =: J(\boldsymbol{\theta})$$

- Typically solved using a gradient-based iterative algorithm
  - Pick an initialization $\boldsymbol{\theta}_0,\ t = 0$
  - While stopping condition is not met
    - $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$
    - $t{+}{+}$

$$\frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}^i, y^i; \boldsymbol{\theta}) =: J(\boldsymbol{\theta})$$

Two Computational Challenges

1. When the number of examples/samples (n) is large, computing the cost function and its gradient is costly because the cost is given by the sum

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^i, y^i; \boldsymbol{\theta})$$

   - Solution: Compute an approximate gradient by only considering a (random) subset of training data at each iteration (called "mini-batch")
   - Intuition: With so many data points, many of them are probably relatively similar
     - May not need to consider them every time
     - Considering only a subset of them may give sufficient information (called "subsampling")
   - Optimization procedure called "stochastic gradient descent" (= gradient descent + subsampling)

Two Computational Challenges

- Stochastic gradient descent (SGD) algorithm
  - A small subsample of n_b data called a mini-batch selected at each iteration
  - One complete pass through the training data is called an "epoch"
    - Consists of $\lceil \frac{n}{n_b} \rceil$ iterations
  - Oftentimes samples are randomly shuffled and then divided into mini-batches
  - After each epoch, training data is randomly shuffled again

$$\frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}^i, y^i; \boldsymbol{\theta}) =: J(\boldsymbol{\theta})$$

Two Computational Challenges

2. When the number of parameters to learn is large, computing the gradient in a naïve manner is costly
   - Solution: Apply **chain rule** and reuse partial derivatives for efficient computation of the gradient
   - Called the **"backpropagation algorithm"**

Chain rule:

$$h(x) = f \circ g(x) = f(g(x)) \quad \Rightarrow \quad h'(x) = f'(g(x)) \cdot g'(x)$$

Backpropagation

- Recall that the parameters to be learned are edge weights (in a matrix form for each layer), including weights for bias units
  - Weights for edges connecting layer l to layer $l+1$

$$\boldsymbol{\theta}^{(l)} = \begin{bmatrix} \theta^{(l)}_{1,0} & \theta^{(l)}_{1,1} & \cdots & \theta^{(l)}_{1,s_l} \\ \vdots & \vdots & \ddots & \vdots \\ \theta^{(l)}_{s_{l+1},0} & \theta^{(l)}_{s_{l+1},1} & \cdots & \theta^{(l)}_{s_{l+1},s_l} \end{bmatrix}$$

$$= \mathbf{b}^{(l)} \qquad = \mathbf{W}^{(l)}$$

- For gradient-based algorithm, need to compute the gradient w.r.t. all parameters

Backpropagation

- Gradients: for $l = 1, \ldots, L-1$

$$d\mathbf{W}^{(l)} := \nabla_{\mathbf{W}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(l)}_{1,1}} & \cdots & \frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(l)}_{1,s_l}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(l)}_{s_{l+1},1}} & \cdots & \frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(l)}_{s_{l+1},s_l}} \end{bmatrix} \quad \text{and} \quad d\mathbf{b}^{(l)} := \nabla_{\mathbf{b}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(l)}_{1,0}} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta^{(l)}_{s_{l+1},0}} \end{bmatrix}$$

$$\mathbf{W}^{(l)}_{t+1} \leftarrow \mathbf{W}^{(l)}_t - \gamma d\mathbf{W}^{(l)}_t$$
$$\mathbf{b}^{(l)}_{t+1} \leftarrow \mathbf{b}^{(l)}_t - \gamma d\mathbf{b}^{(l)}_t$$

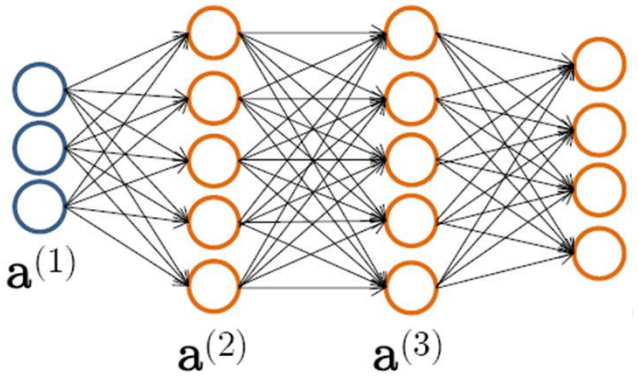
Backpropagation

- Consists of two steps
  - Forward propagation – simply evaluates the cost function using the neural network
  - Backward propagation – requires computation of partial derivatives
- Forward propagation: focus on a single sample

$$\left.\begin{aligned}\mathbf{a}^{(1)} &= \mathbf{x} \\ \mathbf{z}^{(l+1)} &= \mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)} \\ \mathbf{a}^{(l+1)} &= g(\mathbf{z}^{(l)})\end{aligned}\right\} \; l = 1, \ldots, L-2$$

$$z^{(L)} = \mathbf{W}^{(L-1)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L-1)}$$

$$J(\boldsymbol{\theta}) = \begin{cases} (y - z^{(L)})^2 & \text{regression} \\ -z_y^{(L)} + \ln\left(\sum_{j=1}^{m} \exp(z_j^{(L)})\right) & \text{classification} \end{cases}$$



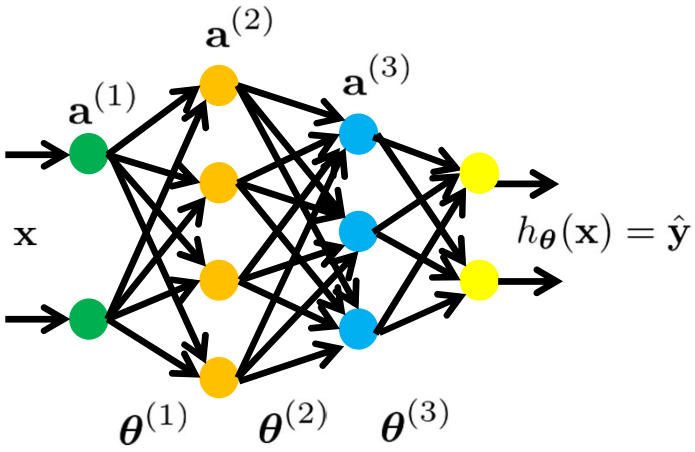
Backpropagation – High-level Picture

- How to compute the gradient of the loss function

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ell\big(\mathbf{y}^i, h_{\boldsymbol{\theta}}(\mathbf{x}^i)\big)$$

where

$$\begin{aligned} h_{\boldsymbol{\theta}}(\mathbf{x}) &= g_3^{\boldsymbol{\theta}^{(3)}}(g_2^{\boldsymbol{\theta}^{(2)}}(g_1^{\boldsymbol{\theta}^{(1)}}(\mathbf{x}))) \\ &= g_3^{\boldsymbol{\theta}^{(3)}} \circ g_2^{\boldsymbol{\theta}^{(2)}} \circ g_1^{\boldsymbol{\theta}^{(1)}}(\mathbf{x}) \end{aligned}$$



- Denote the output from the l-th layer by $\mathbf{a}^{(l)}$ with $\mathbf{a}^{(1)} = \mathbf{x}$ and $\mathbf{a}^{(4)} = \hat{\mathbf{y}}$

$$\begin{aligned} \mathbf{a}^{(2)} &= g_1^{\boldsymbol{\theta}^{(1)}}(\mathbf{x}) = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \\ \mathbf{a}^{(3)} &= g_2^{\boldsymbol{\theta}^{(2)}}(\mathbf{a}^{(2)}) = g(\mathbf{W}^{(2)}\mathbf{a}^{(2)} + \mathbf{b}^{(2)}) \\ \hat{\mathbf{y}} &= g_3^{\boldsymbol{\theta}^{(3)}}(\mathbf{a}^{(3)}) = \sigma(\mathbf{W}^{(3)}\mathbf{a}^{(3)} + \mathbf{b}^{(3)}) \end{aligned}$$

Question: How do we compute the gradient with respect to weights θ?

Backpropagation – High-level Picture

- For simplicity assume differentiable loss function

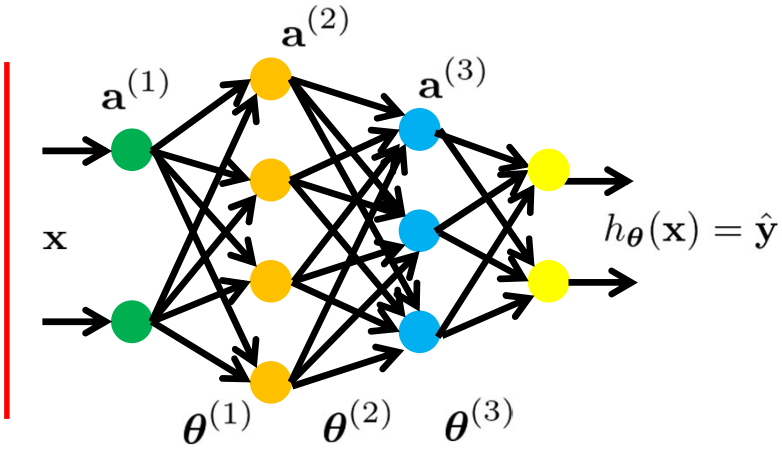
work our way back

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(3)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \boldsymbol{\theta}^{(3)}}$$

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(2)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{a}^{(3)}} \frac{\partial \mathbf{a}^{(3)}}{\partial \boldsymbol{\theta}^{(2)}}$$

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(1)}} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{a}^{(3)}} \frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{a}^{(2)}} \frac{\partial \mathbf{a}^{(2)}}{\partial \boldsymbol{\theta}^{(1)}}$$

new terms



  - Red terms – partial derivatives of the output a layer with respect to the output of previous layer
  - Blue terms – partial derivatives of the output of a layer with respect to its parameters for input
- Computes the gradient of one layer at a time, working our way back starting with the last layer (output layer)
  - Reuses the terms already computed in the previous iteration

Backpropagation – Details

- Basic ingredients of backward propagation

$$d\mathbf{z}^{(l)} := \nabla_{\mathbf{z}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial}{\partial z_1^{(l)}} J(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial z_{s_l}^{(l)}} J(\boldsymbol{\theta}) \end{bmatrix} \quad \text{and} \quad d\mathbf{a}^{(l)} := \nabla_{\mathbf{a}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial}{\partial a_1^{(l)}} J(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial a_{s_l}^{(l)}} J(\boldsymbol{\theta}) \end{bmatrix}$$

- In backward propagation, compute the gradients $d\mathbf{z}^{(l)}$ and $d\mathbf{a}^{(l)}$ for all layers recursively in the opposite direction, starting with the last layer L
  - Regression: $$dz^{(L)} = \frac{\partial}{\partial z^{(L)}} J(\boldsymbol{\theta}) = \frac{\partial}{\partial \mathbf{z}^{(L)}} (y - z^{(L)})^2 = -2(y - z^{(L)})$$
  - Classification:

$$dz_j^{(L)} = \frac{\partial}{\partial z_j^{(L)}} J(\boldsymbol{\theta}) = \frac{\partial}{\partial z_j^{(L)}} \left(-z_y^{(L)} + \ln\left(\sum_{k=1}^{m} \exp(z_k^{(L)})\right)\right) = -\mathbf{1}\{y = j\} + \frac{\exp(z_j^{(L)})}{\sum_{k=1}^{m} \exp(z_k^{(L)})}$$

Backpropagation – Details

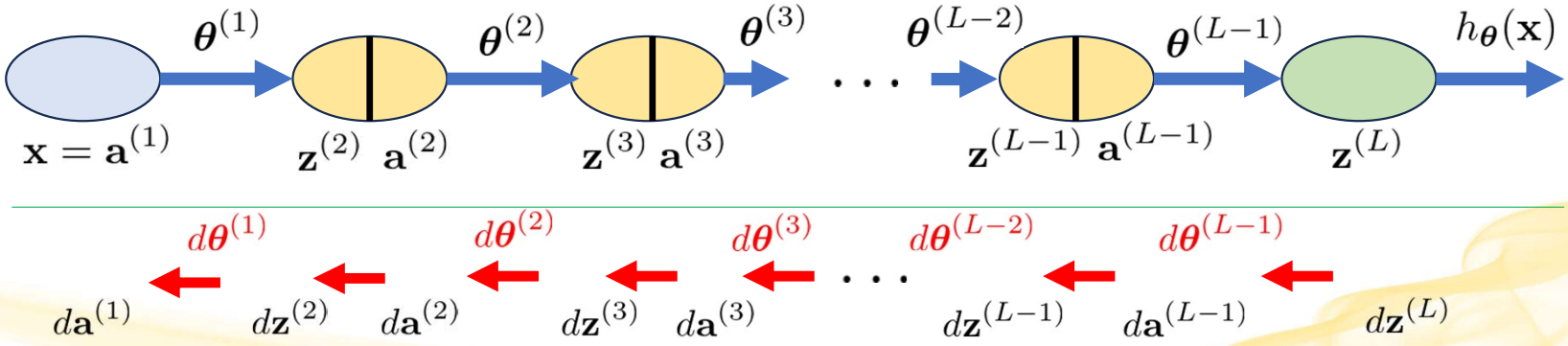
- Recursion:

$$d\mathbf{a}^{(l-1)} = \mathbf{W}^{(l-1)^T} d\mathbf{z}^{(l)},\ l = 3, \ldots, L$$
$$d\mathbf{z}^{(l-1)} = d\mathbf{a}^{(l-1)} \odot g'(\mathbf{z}^{(l-1)}),\ l = 3, \ldots, L$$

$\odot$ – elementwise product

- Recall $\mathbf{a}^{(l)} = g(\mathbf{z}^{(l)}) = g(\mathbf{W}^{(l-1)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l-1)})$ and $\mathbf{z}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l-1)}$

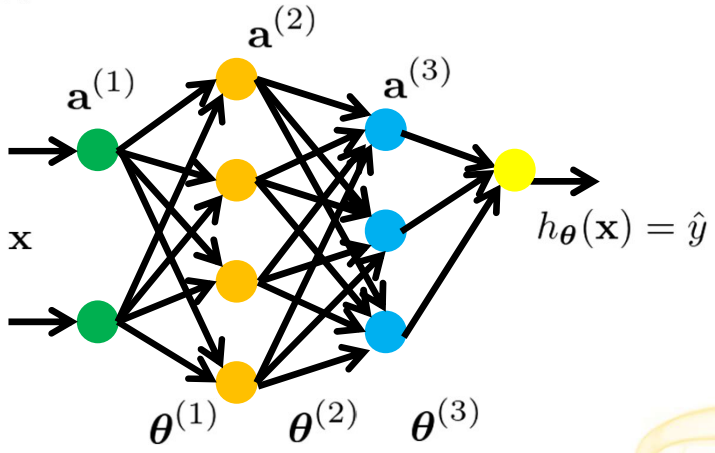
$$d\mathbf{W}^{(l)} = \nabla_{\mathbf{W}^{(l)}} J(\boldsymbol{\theta}) = d\mathbf{z}^{(l+1)}\mathbf{a}^{(l)^T}$$
$$d\mathbf{b}^{(l)} = d\mathbf{z}^{(l+1)}$$



Backpropagation - Example

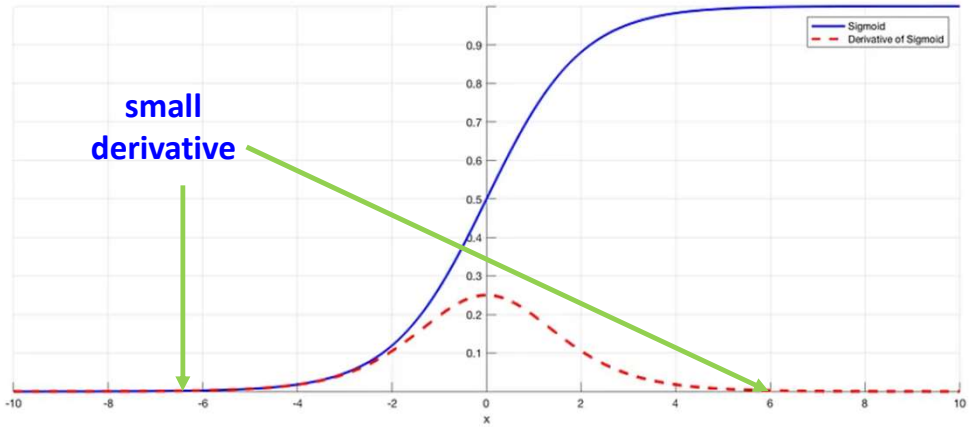
- Regression problem

$$
\begin{aligned}
dz^{(4)} &= \frac{\partial}{\partial z^{(4)}} J(\boldsymbol{\theta}) = -2(y - z^{(4)}) = -2(y - \hat{y}) \\
d\mathbf{W}^{(3)} &= \nabla_{\mathbf{W}^{(3)}} J(\boldsymbol{\theta}) = dz^{(4)} \mathbf{a}^{(3)^T} \\
db^{(3)} &= dz^{(4)} \\
d\mathbf{a}^{(3)} &= \mathbf{W}^{(3)^T} dz^{(4)} \\
d\mathbf{z}^{(3)} &= d\mathbf{a}^{(3)} \odot g'(\mathbf{z}^{(3)}) \\
d\mathbf{W}^{(2)} &= d\mathbf{z}^{(3)} \mathbf{a}^{(2)^T} \\
d\mathbf{b}^{(2)} &= d\mathbf{z}^{(3)} \\
d\mathbf{a}^{(2)} &= \mathbf{W}^{(2)^T} dz^{(3)} \\
d\mathbf{z}^{(2)} &= d\mathbf{a}^{(2)} \odot g'(\mathbf{z}^{(2)}) \\
d\mathbf{W}^{(1)} &= d\mathbf{z}^{(2)} \mathbf{a}^{(1)^T} \\
d\mathbf{b}^{(1)} &= d\mathbf{z}^{(2)}
\end{aligned}
$$



Vanishing Gradient in Backpropagation

- Well-known issue: vanishing gradient



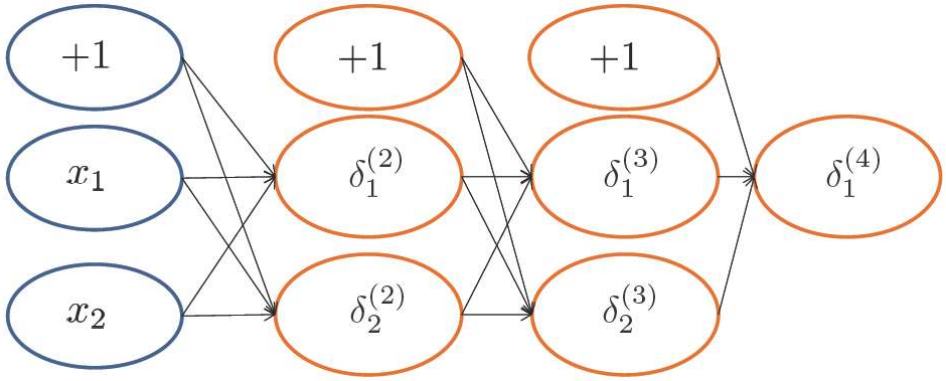
  - As more (hidden) layers are added in neural networks, with certain activation function (e.g., sigmoid), gradients of loss function become smaller
    - Makes training harder with small gradients
  - Solutions: ReLU activation function, batch normalization layer, residual networks

Other Issues with Backpropagation

- “Backpropagation is the cockroach of machine learning. It’s ugly, and annoying, but you just can’t get rid of it.” –Geoff Hinton
- Other problems:
  - Black box
  - Local minima

Random Initialization

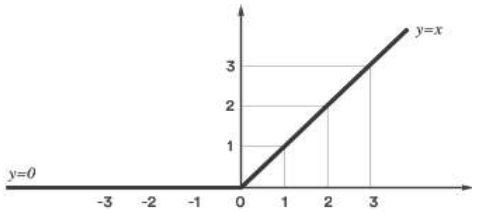
- Important to **randomize** initial weight matrices
- Cannot have uniform initial weights
  - Otherwise, all updates will be identical & **the neural network will NOT learn**



Activation Functions

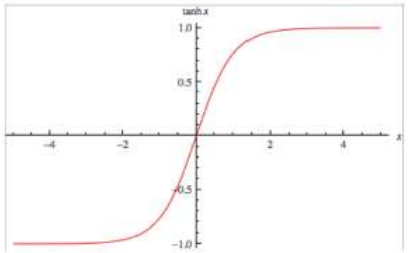
Activation Functions

ReLU $\sigma(z) = \max(0, z)$



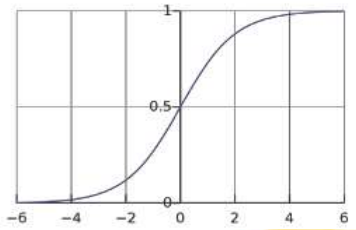
- Very fast computation.
- Most common activation function.
- Piecewise linear (Not actually differentiable everywhere, but this is okay.)

tanh $\sigma(z) = \dfrac{e^{2z} - 1}{e^{2z} + 1}$



- "hyperbolic tangent", often pronounced like "tanch".
- Smoothes out discontinuities.
- Used to be more popular, replaced by ReLU.

logistic sigmoid $\sigma(z) = \dfrac{1}{1 + e^{-z}}$



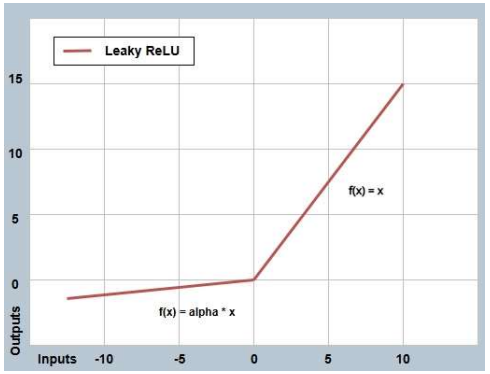
- Similar to tanh. Used rarely nowadays.

Activation Functions

- Leaky ReLU

$$\sigma(z) = \max(\alpha \times z, z),\ 0 < \alpha \ll 1$$

  - Similar to ReLU
  - Piecewise linear
  - Copes with dying ReLU problem
    - Can be caused by high learning rates or large negative bias



"Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue."

Activation functions

- **Softmax**
  - Applied to an entire vector, not component-wise
  - Normalizes a vector. Entries of the output sum to 1
  - The most common way to predict a probability distribution
  - Typically used in the final layer of networks for classification tasks

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Loss Functions

Loss Functions

1. Mean squared error

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{k=1}^{n} \|\mathbf{y} - h_{\boldsymbol{\theta}}(\mathbf{x})\|^2$$

- Often used for regression problem, i.e., a network is used to predict the value of output for a new instance

Loss Functions

2. Cross-entropy

- Most common loss function for classification tasks (with m classes/labels)
- Measure of distance between two probability distributions
- Apply only to probabilities (e.g., after softmax as we assumed)

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{k=1}^{n} L(\mathbf{x}^i, y^i; \boldsymbol{\theta}) = -\frac{1}{n}\sum_{k=1}^{n} \ln\left(\frac{\exp(h_{\boldsymbol{\theta},y^i}(\mathbf{x}^i)))}{\sum_{j=1}^{m}\exp(h_{\boldsymbol{\theta},j}(\mathbf{x}^i))}\right)$$
$$= \frac{1}{n}\sum_{k=1}^{n}\left(-z^k_{y^i} + \ln\left(\sum_{j=1}^{m}\exp(z^k_j)\right)\right)$$

Loss Functions

3. Cosine similarity (between two vectors)

- Often used in natural language processing (NLP) contexts
- For example, two vectors might represent the frequencies of different words

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{k=1}^{n}\frac{\langle \mathbf{y}^i, h_{\boldsymbol{\theta}}(\mathbf{x}^i)\rangle}{\|\mathbf{y}^i\| \cdot \|h_{\boldsymbol{\theta}}(\mathbf{x}^i)\|}$$

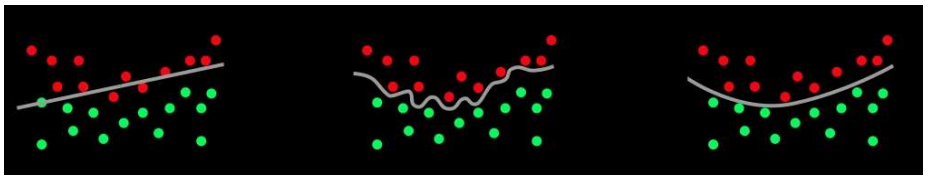
Underfitting & Overfitting

Key Terminology

- Bias – measures the difference between the target value and the model prediction
  - Oversimplified models typically produce predictions that are far from the target values, resulting in a higher bias
- Variance – measure of inconsistency or variability of predictions over different datasets
  - If predictions over different datasets are close, there is less variance
- Model generalization – ability of a model to extract important features in training data and generate accurate predictions on new/unseen data

Overfitting vs. Underfitting

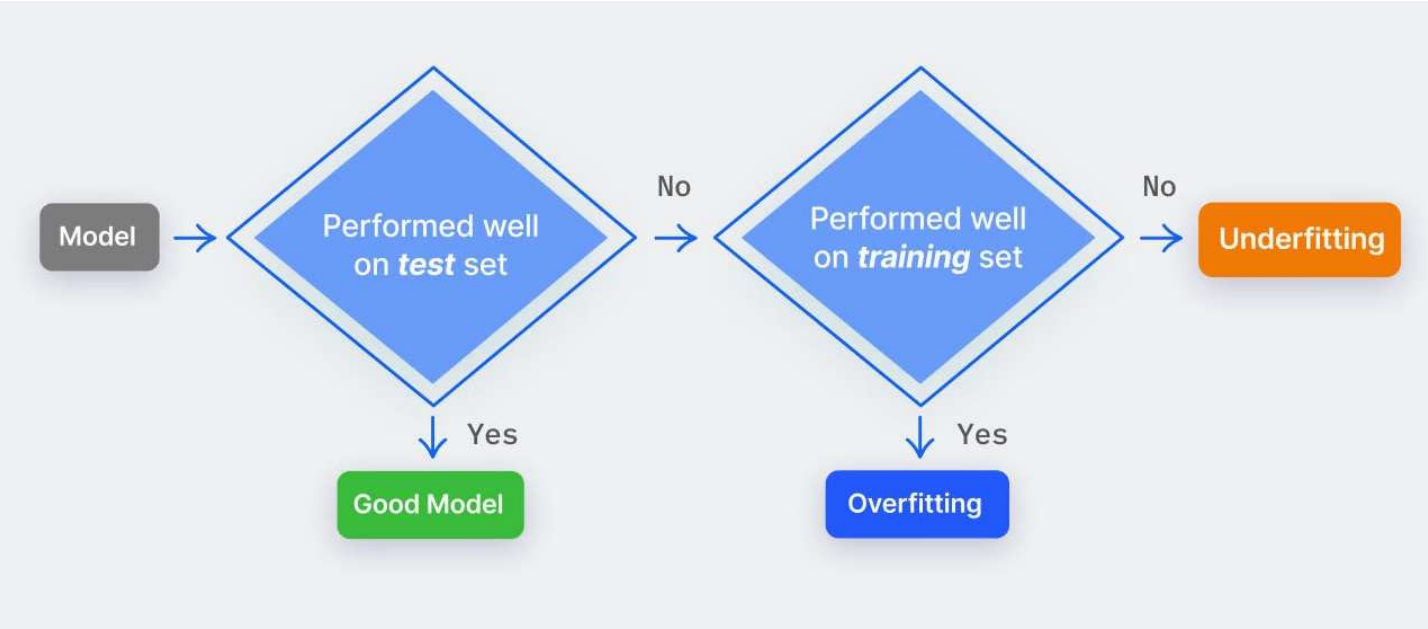
- Underfitting occurs when the model is oversimplified and cannot generate accurate predictions
  - Model has a high bias
- Overfitting happens when the model has a high variance
  - Model has good performance on training data, but fails to generalize and performs poorly on new/unseen data
  - Result of “memorizing” the data patterns and learning from noise in training dataset



underfitting overfitting balanced

[source:towardsdatascience.com]

Overfitting vs. Underfitting



[source:v7labs.com]

Underfitting

- Causes for underfitting
  - Inability of model to capture and learn the relationship between the input examples (e.g., feature values) and the target values (e.g., label)
  - Oversimplified model, e.g., a linear model for a highly nonlinear relationship
  - Noisy data with outliers

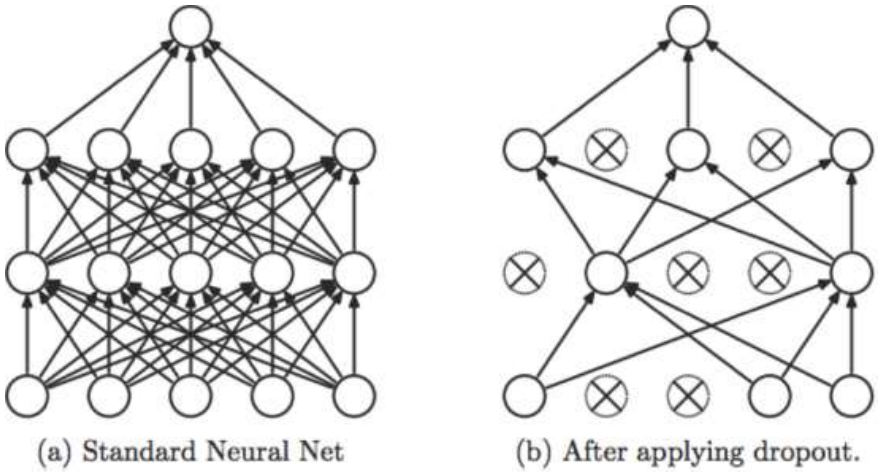
Overfitting

- ML model offers accurate predictions for training data, but fails to generalize
  - Occurs for various reasons
    - Model is too flexible in relation to the complexity/size of data
    - Training data contains a lot of irrelevant information or garbage values
    - Model trained too long on the same training data (starts to learn from the noise in the training data)

Overfitting

- Techniques for mitigating/preventing overfitting
  - Cross-validation
  - Data augmentation
  - Dropout
  - Early stopping
  - Ensembling
  - Feature selection/pruning
  - Hold-out
  - Regularization

Dropout

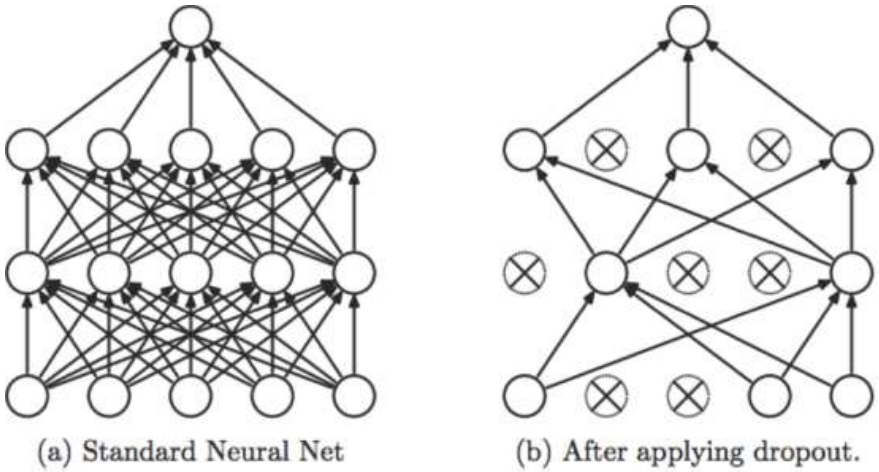


(a) Standard Neural Net (b) After applying dropout.

Srivastava *et. al.,* "Dropout: A simple way to prevent neural networks from overfitting," JMLR (2014)

- At each iteration, randomly "drop" some of hidden nodes (and inputs)
  - Can be done with binary mask
  - When a node is removed, also remove all of its incoming and outgoing edges
  - Results in a "subnetwork"
  - Probability of keeping a node is a hyperparameter (possibly different for different layers)

Dropout



Srivastava *et. al.,* "Dropout: A simple way to prevent neural networks from overfitting," JMLR (2014)

- Only the gradient for the subnetwork is used for update
  - Parameters associated with dropped nodes left untouched
- In the next iteration, repeat the same process with a newly generated subnetwork
- Typical dropout probability: 0.2 for input and 0.5 for hidden units

Dropout

- Intuition: Reduce the variance by imitating the effects of ensembling (e.g., bagging)
  - More than one model is used to generate predictions
  - Key difference is that we do not explicitly train multiple models and use them to compute the “average” for prediction
  - Found to be effective in practice

Data Argumentation

- Construct new data points/examples by duplicating existing data (often with invariant transformation)
  - Commonly used for images – cropping, rotation, vertical flipping, noise addition, color shift, and contrast change
  - Adding noise to the input: a special kind of augmentation
- Be careful about the transformation applied (some are not invariant)
  - Example: classifying ‘b’ and ‘d’
  - Example: classifying ‘6’ and ‘9’

Data Argumentation

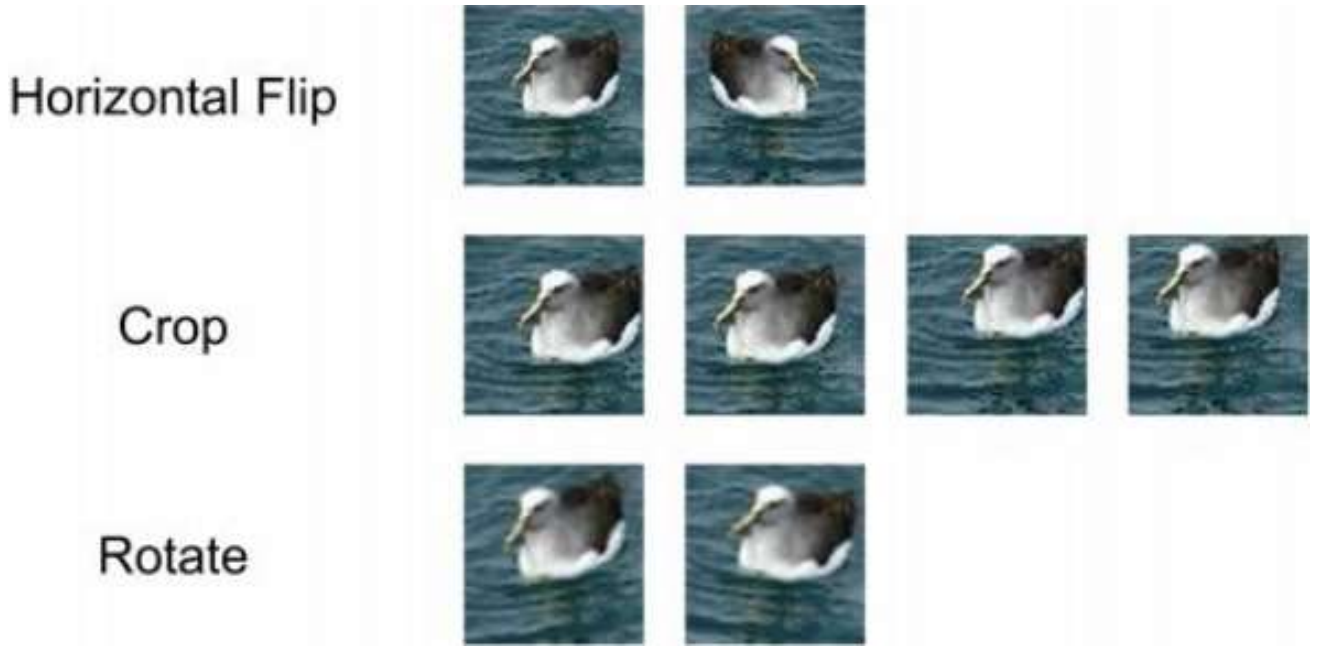


Image source: *Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7,* by Keven Wang

Hold-Out

- After training a model, we want to know, for example, error

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}}\left[\|\mathbf{y} - h_{\boldsymbol{\theta}}(\mathbf{x})\|^2\right] \quad \text{or} \quad J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\left[\mathbf{1}\{y \neq h_{\boldsymbol{\theta}}(\mathbf{x})\}\right]$$

  - Cannot be computed in practice
- Using the sample mean is not reliable or advised
- When a large dataset is available, a portion of the dataset D_V can be set aside for validation
- When partitioning the dataset, there is a trade-off between the size of training dataset and validation dataset

K-Fold Cross-Validation

- Question: What if the dataset is not very large and we cannot afford to set aside a chunk of examples for validation?
- k-fold cross-validation: Repeat the hold-out validation multiple times with a different hold-out dataset each time
  - Training dataset is partitioned into k batches of similar size (called “folds”)
  - For l = 1,…,k
    - Train using all batches, except for batch l
    - Validate using batch l
  - Compute the average of k validation errors as the validation error

Regularization

- Key idea is to keep the parameters small unless the data says otherwise
  - If a model with small values of parameters fits the data almost as well as a model with larger parameter values, the model with small parameter values should preferred

$$\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta})$$

- Two popular regularization methods
  - L2 – regularization: forces the parameter towards small values

$$\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2$$

  - L1 – regularization: tends to favor sparse solutions (only a few of parameters are nonzero)
    - Can be used for input or feature selection method

$$\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_1$$