Unit 8
Neural Networks

Prof. Phil Schniter

The Ohio State University

ECE 5307: Introduction to Machine Learning, Sp23

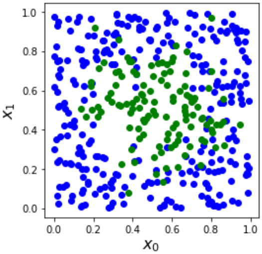
Learning objectives

- Understand 2-layer feedforward neural networks:
  - Motivation: learning feature transformations
  - Network architecture: linear and nonlinear layers
  - Choice of activation functions and training loss
- Understand mini-batch training and stochastic gradient descent
- Understand the back-propagation approach to gradient computation
- Know how to implement a neural network using PyTorch

Outline

Dealing with data that is not linearly separable

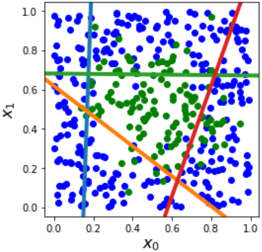
- Consider the data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ on the right:
  - Features $\boldsymbol{x}_i = [x_{i0}, x_{i1}]^\mathsf{T} \in \mathbb{R}^2$
  - Labels $y_i \in \{0, 1\}$
  - Not linearly separable!



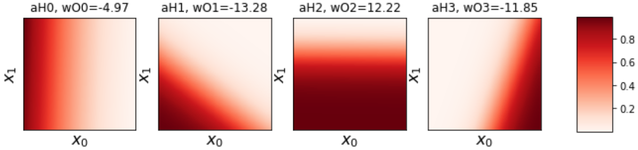
- Could use a feature transformation to enhance linear separability, and then apply linear classification on the transformed features
  - But what if we don't know a good transformation?
  - Can we learn one?
  - Yes!

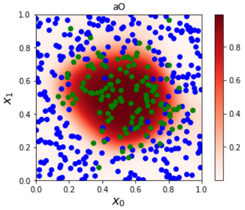
A two-stage approach to classification

- A two-stage approach:
  1) Learn 4 transformed features, each the soft output of a linear classifier
     - "soft" output means in the interval $[0, 1]$
  2) Apply linear classification to the transformed features, giving a final soft output
     - high for one intersection of half-spaces



- Plot of transformed features and final soft output vs. $\boldsymbol{x}$:

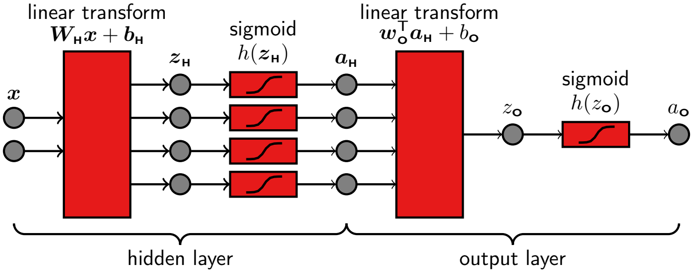




- The overall approach is successful in classifying the data!

Details of two-stage approach

- Stage 1: "Hidden layer"
  - scores: $\boldsymbol{z}_{\mathsf{H}} = \boldsymbol{W}_{\mathsf{H}}\boldsymbol{x} + \boldsymbol{b}_{\mathsf{H}} \in \mathbb{R}^{d_{\mathsf{H}}}$
  - soft outputs: $\boldsymbol{a}_{\mathsf{H}} = h(\boldsymbol{z}_{\mathsf{H}}) \in [0,1]^{d_{\mathsf{H}}}$
  - $\boldsymbol{a}_{\mathsf{H}}$ are the learned features
- Stage 2: "Output layer"
  - score: $z_{\mathsf{O}} = \boldsymbol{w}_{\mathsf{O}}^{\mathsf{T}}\boldsymbol{a}_{\mathsf{H}} + b_{\mathsf{O}} \in \mathbb{R}$
  - soft output: $a_{\mathsf{O}} = h(z_{\mathsf{O}}) \in [0,1]$
  - a_{O} is the final $\Pr\{y=1 \,|\, \boldsymbol{x}\}$
- If we use $h(z) = \frac{1}{1+e^{-z}}$, a sigmoid, the output layer is logistic regression
- This is a multi-layer perceptron, or 2-stage feed-forward neural network!



Training the model

- Let's collect the model parameters into $\boldsymbol{\theta} \triangleq (\boldsymbol{W}_{\mathsf{H}}, \boldsymbol{b}_{\mathsf{H}}, \boldsymbol{w}_{\mathsf{O}}, b_{\mathsf{O}})$
- We fit these parameters using training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$
- Since we used logistic regression, the likelihood function would be

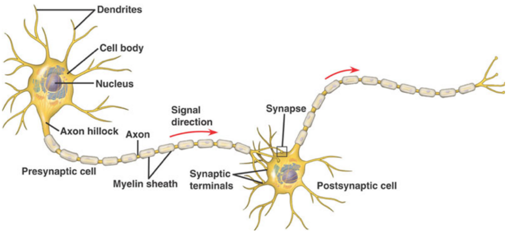
$$\Pr\{y_i = 1 \,|\, \boldsymbol{x}_i; \boldsymbol{\theta}\} = \frac{1}{1 + e^{-z_{\mathsf{O},i}}} \quad \text{for} \quad z_{\mathsf{O},i} = F(\boldsymbol{x}_i; \boldsymbol{\theta})$$

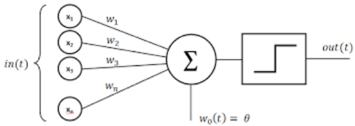
  - $F(\boldsymbol{x}; \boldsymbol{\theta})$ describes the network from input $\boldsymbol{x}$ to output score z_{O}
- Then the maximum-likelihood model parameters are

$$\begin{aligned} \widehat{\boldsymbol{\theta}}_{\mathsf{ml}} &= \arg\min_{\boldsymbol{\theta}} \textstyle\sum_{i=1}^n \big[-\ln p(y_i \,|\, \boldsymbol{x}_i; \boldsymbol{\theta}) \big] \\ &= \arg\min_{\boldsymbol{\theta}} \textstyle\sum_{i=1}^n \big(\ln[1 + e^{z_{\mathsf{O},i}}] - y_i z_{\mathsf{O},i} \big) \quad \text{(binary cross-entropy loss)} \end{aligned}$$

  - We will discuss the details of this optimization later

Why is it called a "neural" network?





- Simple model of neurons:
  - Dendrites: Input currents from other neurons
  - Soma: Cell body, accumulation of charge
  - Axon: Outputs to other neurons
- Operation:
  - Output when the sum of input currents reaches a threshold
  - Similar to (artificial) neural network: $a_{\mathsf{H},j} = h(\boldsymbol{w}_{\mathsf{H},j}^{\mathsf{T}}\boldsymbol{x} + b_{\mathsf{H},j})$

History

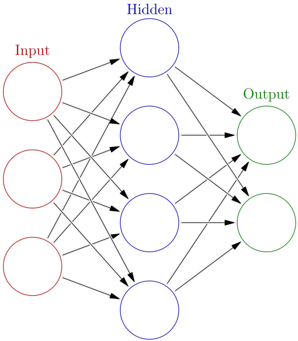
- 1940s: Donald Hebb — Hebbian learning for neural plasticity
  - Hypothesized rule for updating synaptic weights in biological neurons
- 1950s: Frank Rosenblatt — Coined the term "perceptron"
  - Essentially a single-layer network, similar to logistic regression
  - Early computer implementations
  - But linear classifiers are limited, and so was compute power
- 1960s: Back-propagation — Efficient way to train multi-layer networks
  - We'll cover this later in this unit
- 1990s: Resurgence with greater computational power
- 2012-now: The deep-network revolution
  - Many more layers. Massive computational power and data
  - Breakthroughs in speech and image processing first, then many more fields . . .
  - We'll cover deep networks in the next unit

Outline

- Motivating Example: Learning a Feature Transformation
- Feed-Forward Neural Networks
- Training via Stochastic Gradient Descent
- Gradient Computation via Back-Propagation
- Implementing and Training Neural Nets with PyTorch

General structure of a feed-forward neural network

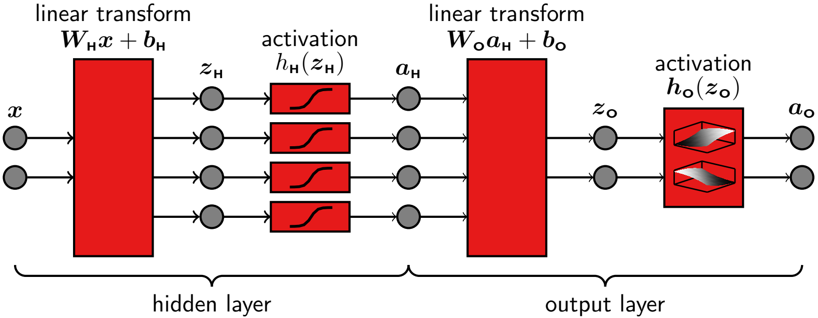
- Input: $\boldsymbol{x} = [x_1, \ldots, x_d]^\mathsf{T}$
  - $d =$ number of features (or inputs)
- Hidden layer:
  - $a_{\mathsf{H},l} = h_{\mathsf{H}}\big(b_{\mathsf{H},l} + \sum_{j=1}^{d} w_{\mathsf{H},lj} x_j\big),\ l = 1 \ldots d_{\mathsf{H}}$
  - $h_{\mathsf{H}}(\cdot)$ is a nonlinear "activation" function
  - $d_{\mathsf{H}} =$ number of hidden units
- Output layer:
  - $a_{\mathsf{O},k} = h_{\mathsf{O}}\big(b_{\mathsf{O},k} + \sum_{l=1}^{d_{\mathsf{H}}} w_{\mathsf{O},kl} a_{\mathsf{H},l}\big),\ k = 1 \ldots d_{\mathsf{O}}$
  - $h_{\mathsf{O}}(\cdot)$ is optional. If present, it must match loss
  - $d_{\mathsf{O}} =$ number of outputs



- Networks of this form also referred to as "multilayer perceptrons"
- Can use more than two layers ("deep network"), but for now we focus on two

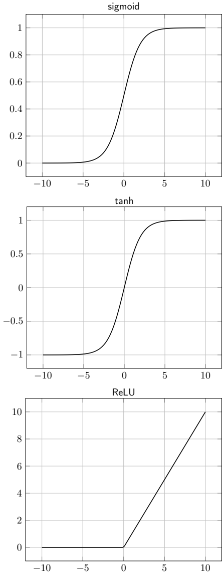
Vector/matrix representation of feed-forward neural network

- Hidden layer:
  - linear transform: $\boldsymbol{z}_\mathsf{H} = \boldsymbol{W}_\mathsf{H}\boldsymbol{x} + \boldsymbol{b}_\mathsf{H}$
  - nonlinear activation: $\boldsymbol{a}_\mathsf{H} = h_\mathsf{H}(\boldsymbol{z}_\mathsf{H}) \in \mathbb{R}^{d_\mathsf{H}}$, where $h_\mathsf{H}(\cdot)$ acts elementwise
- Output layer:
  - linear transform: $\boldsymbol{z}_\mathsf{O} = \boldsymbol{W}_\mathsf{O}\boldsymbol{a}_\mathsf{H} + \boldsymbol{b}_\mathsf{O}$
  - (optional) activation: $\boldsymbol{a}_\mathsf{O} = \boldsymbol{h}_\mathsf{O}(\boldsymbol{z}_\mathsf{O}) \in \mathbb{R}^{d_\mathsf{O}}$, where $\boldsymbol{h}_\mathsf{O}(\cdot)$ may act on full vector



Common choices for hidden-layer activation $h_{\text{H}}(z_{\text{H}})$

- sigmoid: $h_{\text{H}}(z) = \frac{1}{1+e^{-z}} \triangleq \sigma(z)$
  - output is bounded, but not centered
  - sometimes used in shallow networks
- tanh: $h_{\text{H}}(z) = \tanh(z) = 2\sigma(z) - 1$
  - output is bounded and centered
  - often works better than sigmoid, because centered data is consistent with PyTorch weight initializations
- rectified linear unit (ReLU): $h_{\text{H}}(z) = \max\{0, z\}$
  - output is unbounded and not centered
  - most popular choice in deep networks
  - works well in shallow networks too



Common choices for output-layer activation $\boldsymbol{h}_{\circ}(\boldsymbol{z}_{\circ})$

- Binary classification:
  - here $\boldsymbol{z}_{\mathsf{o}} = z_{\mathsf{o}}$ is a scalar
  - option 1: use no output activation fxn (i.e., output z_{o})
  - option 2: $h_{\mathsf{o}}(z_{\mathsf{o}}) = \dfrac{1}{1+e^{-z_{\mathsf{o}}}} = \Pr\{y\!=\!1 \,|\, \boldsymbol{x}\}$ (logistic)
- Multiclass classification with $K > 2$ classes:
  - here $\boldsymbol{z}_{\mathsf{o}} = [z_{\mathsf{o},1}, \dots, z_{\mathsf{o},K}]^{\mathsf{T}} \in \mathbb{R}^K$
  - option 1: use no output activation fxn (i.e., output $\boldsymbol{z}_{\mathsf{o}}$)
  - option 2: $[\boldsymbol{h}_{\mathsf{o}}(\boldsymbol{z}_{\mathsf{o}})]_k = \dfrac{e^{z_{\mathsf{o},k}}}{\sum_{l=1}^K e^{z_{\mathsf{o},l}}} = \Pr\{y\!=\!k \,|\, \boldsymbol{x}\}$ for $k = 1 \dots K$ (softmax)
- Regression with K-dimensional targets (i.e., $\boldsymbol{y}_i \in \mathbb{R}^K$):
  - here $\boldsymbol{z}_{\mathsf{o}} = [z_{\mathsf{o},1}, \dots, z_{\mathsf{o},K}]^{\mathsf{T}} \in \mathbb{R}^K$
  - use no output activation fxn (i.e., output $\boldsymbol{z}_{\mathsf{o}}$)

Loss functions for training

The task determines the loss and the (optional) output activation $\boldsymbol{h}_{\mathsf{o}}(\cdot)$!

- Binary classification:
  - Likelihood: $\prod_{i=1}^{n} p(y_i|\boldsymbol{x},\boldsymbol{\theta})$ with $\Pr\{y_i\!=\!1\,|\,\boldsymbol{x}\} = \dfrac{1}{1+e^{-z_{\mathsf{o},i}}}$ (logistic)
  - Loss: $\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \big(\ln[1+e^{z_{\mathsf{o},i}}] - y_i z_{\mathsf{o},i}\big)$ (binary cross entropy)
  - Or $\mathcal{J}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \big(y_i \ln a_{\mathsf{o},i} + [1-y_i]\ln[1-a_{\mathsf{o},i}]\big)$ if $a_{\mathsf{o},i} = h_{\mathsf{o}}(z_{\mathsf{o},i}) = \frac{1}{1+e^{-z_{\mathsf{o},i}}}$
- Multiclass classification with $K>2$ classes:
  - Likelihood: $\prod_{i=1}^{n} p(y_i|\boldsymbol{x},\boldsymbol{\theta})$ with $\Pr\{y_i\!=\!k\,|\,\boldsymbol{x}\} = \dfrac{e^{z_{\mathsf{o},ik}}}{\sum_{l=1}^{K} e^{z_{\mathsf{o},il}}}$ (softmax)
  - Loss: $\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \big(\ln \sum_{k=1}^{K} e^{z_{\mathsf{o},ik}} - \sum_{k=1}^{K} y_{ik} z_{\mathsf{o},ik}\big)$ (cross entropy)
  - Or $\mathcal{J}(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} \ln a_{\mathsf{o},ik}$ if $a_{\mathsf{o},ik} = [\boldsymbol{h}_{\mathsf{o}}(\boldsymbol{z}_{\mathsf{o},i})]_k = \frac{e^{z_{\mathsf{o},ik}}}{\sum_{l=1}^{K} e^{z_{\mathsf{o},il}}}$
- Regression with K-dimensional targets:
  - Likelihood: $p(\boldsymbol{y}\,|\,\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y};\boldsymbol{z}_{\mathsf{o}},\sigma_\epsilon^2\boldsymbol{I})$ (additive white Gaussian noise)
  - Loss: $\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik}-z_{\mathsf{o},ik})^2$ (quadratic)

Outline

- Motivating Example: Learning a Feature Transformation
- Feed-Forward Neural Networks
- Training via Stochastic Gradient Descent
- Gradient Computation via Back-Propagation
- Implementing and Training Neural Nets with PyTorch

Gradient descent

- Goal: Find model parameters $\boldsymbol{\theta}$ that minimize the loss function $\mathcal{J}(\boldsymbol{\theta})$:

$$\widehat{\boldsymbol{\theta}}_{\mathsf{ml}} = \arg\min_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \quad \text{for} \quad \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} J(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i),$$

  where $J(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)$ is the contribution from training sample i

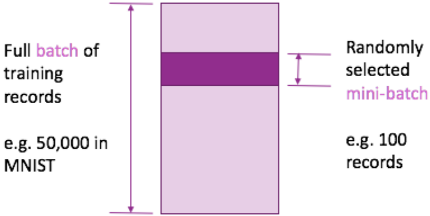
- Can tackle this using the gradient descent (GD) algorithm:

$$\begin{aligned}\boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \alpha_k \nabla \mathcal{J}(\boldsymbol{\theta}^k) \\ &= \boldsymbol{\theta}^k - \frac{\alpha_k}{n}\sum_{i=1}^{n} \nabla J(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_i)\end{aligned}$$

  - Each iteration computes n gradients
  - This is very expensive when n (# training samples) is large!

Stochastic gradient descent using mini-batches

- Main idea: In each step ...
  - select a random mini-batch
  - evaluate gradient on mini-batch
- Details: At step $t = 1 \dots T$:
  - randomly select subset of indices:
    $$I_t \subset \{1, \dots, n\}$$
  - compute approximate gradient:
    $$\boldsymbol{g}^t \triangleq \frac{1}{|I_t|} \sum_{i \in I_t} \nabla J(\boldsymbol{\theta}^t, \boldsymbol{x}_i, \boldsymbol{y}_i)$$
  - update parameters:
    $$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \alpha_t \boldsymbol{g}^t$$



- Called stochastic gradient descent (SGD) because $\boldsymbol{g}^t$ is a random approximation of the true gradient

Justification for stochastic gradient descent

- The gradient approximation $\boldsymbol{g}^t$ is correct on average (i.e., unbiased):

$$\mathbb{E}\left\{\boldsymbol{g}^t \,\middle|\, \boldsymbol{\theta}^t\right\} = \frac{1}{n}\sum_{i=1}^{n} \nabla J(\boldsymbol{\theta}^t, \boldsymbol{x}_i, \boldsymbol{y}_i) = \nabla \mathcal{J}(\boldsymbol{\theta}^t)$$

- Thus the SGD-updated parameters are also correct on average (i.e., unbiased):

$$\begin{aligned}\mathbb{E}\left\{\boldsymbol{\theta}^{t+1} \,\middle|\, \boldsymbol{\theta}^t\right\} &= \mathbb{E}\left\{\boldsymbol{\theta}^t - \alpha_t \boldsymbol{g}^t \,\middle|\, \boldsymbol{\theta}^t\right\} \\ &= \boldsymbol{\theta}^t - \alpha_t \nabla \mathcal{J}(\boldsymbol{\theta}^t)\end{aligned}$$

- We can think of $\boldsymbol{g}^t$ as having "gradient noise" $\boldsymbol{\epsilon}^t$ that is zero mean:

$$\boldsymbol{g}^t = \nabla \mathcal{J}(\boldsymbol{\theta}^t) + \boldsymbol{\epsilon}^t \quad \text{with} \quad \mathbb{E}\{\boldsymbol{\epsilon}^t \,|\, \boldsymbol{\theta}^t\} = \mathbf{0}$$

  - This noise makes the SGD trajectory more noisy than the true GD trajectory
  - Can be mitigated by reducing the step-size α_t

- For more details, see
  https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Implementing mini-batch

- Setup:
  - n samples in the training dataset
  - B samples in each (non-overlapping) mini-batch
  - $T = n/B$ mini-batches total
  - 1 SGD update step per mini-batch
  - T updates in each "training epoch"
    - Each of the n training samples is used once per training epoch
- Implementation: *In each epoch...*
  - Randomly shuffle all n training indices $\{i\}$
    - This way, the mini-batches will change over the epochs!
  - Partition the shuffled indices $\{i\}$ into T contiguous subsets $\{I_t\}_{t=1}^{T}$
  - Use subset I_t to compute the gradient and cost in SGD update step t

Outline

- Motivating Example: Learning a Feature Transformation
- Feed-Forward Neural Networks
- Training via Stochastic Gradient Descent
- Gradient Computation via Back-Propagation
- Implementing and Training Neural Nets with PyTorch

Computing the gradient

- The network parameters $\boldsymbol{\theta} = (\boldsymbol{W}_{\mathsf{H}}, \boldsymbol{b}_{\mathsf{H}}, \boldsymbol{W}_{\mathsf{O}}, \boldsymbol{b}_{\mathsf{O}})$ must be trained
- To do this, we described the mini-batch SGD update

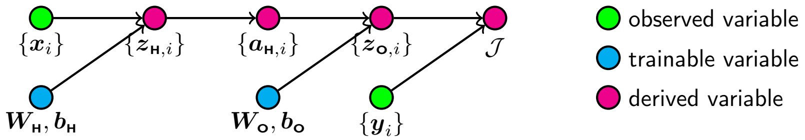
$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^{t} - \frac{\alpha_t}{|I_t|} \nabla \underbrace{\sum_{i \in I_t} J(\boldsymbol{\theta}^t; \boldsymbol{x}_i, \boldsymbol{y}_i)}_{\triangleq\, \mathcal{J}^t(\boldsymbol{\theta}^t)}$$

- But how do we compute the gradient?
  - Recall: the gradient is the partial derivative w.r.t. each parameter in $\boldsymbol{\theta}$
  - So we need to compute

$$\frac{\partial \mathcal{J}^t(\boldsymbol{\theta}^t)}{\partial w_{\mathsf{H},lj}}\ \forall l,j, \qquad \frac{\partial \mathcal{J}^t(\boldsymbol{\theta}^t)}{\partial b_{\mathsf{H},l}}\ \forall l, \qquad \frac{\partial \mathcal{J}^t(\boldsymbol{\theta}^t)}{\partial w_{\mathsf{O},kl}}\ \forall k,l, \qquad \frac{\partial \mathcal{J}^t(\boldsymbol{\theta}^t)}{\partial b_{\mathsf{O},k}}\ \forall k$$

  - Going forward, we simplify our notation by dropping the batch index "t"

The computation graph



- The computation graph will help us to organize our computations
- The loss $\mathcal{J}$ can be computed using a forward pass through the graph:

$$\begin{aligned}
\boldsymbol{z}_{\mathsf{H},i} &= \boldsymbol{W}_{\mathsf{H}}\boldsymbol{x}_i + \boldsymbol{b}_{\mathsf{H}}, \quad \forall i = 1 \dots B \\
\boldsymbol{a}_{\mathsf{H},i} &= h_{\mathsf{H}}(\boldsymbol{z}_{\mathsf{H},i}), \quad \forall i = 1 \dots B \\
\boldsymbol{z}_{\mathsf{O},i} &= \boldsymbol{W}_{\mathsf{O}}\boldsymbol{a}_{\mathsf{H},i} + \boldsymbol{b}_{\mathsf{O}}, \quad \forall i = 1 \dots B \\
\mathcal{J} &= \textstyle\sum_{i=1}^{B} J(\boldsymbol{z}_{\mathsf{O},i}; \boldsymbol{y}_i)
\end{aligned}$$

- The gradients can then be computed using a backward pass, as described next
- Note: we write the loss $J(\cdot, \boldsymbol{y}_i)$ in terms of $\boldsymbol{z}_{\mathsf{O},i}$, not $\boldsymbol{a}_{\mathsf{O},i}$
  - PyTorch allows either, as discussed on page 44

Review of the chain rule

- Suppose that x, y, z are scalars. Then the chain rule says

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

$$x \longrightarrow y \longrightarrow z$$

- Now suppose that another variable u is inserted. Previous expression still holds!

$$\frac{\partial z}{\partial x} = \underbrace{\frac{\partial z}{\partial u}\frac{\partial u}{\partial y}}_{\frac{\partial z}{\partial y}}\frac{\partial y}{\partial x}$$

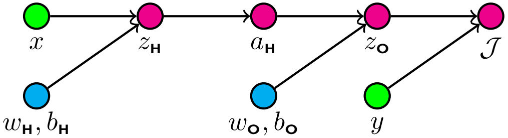
$$x \longrightarrow y \longrightarrow u \longrightarrow z$$

- Now suppose that there are several intermediate variables $\{y_i\}_{i=1}^{B}$. The multivariable chain rule says

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{B} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

$$x \longrightarrow \{y_i\} \longrightarrow u \longrightarrow z$$

Simple case 1: Scalar variables and batch size $B = 1$



$$\begin{aligned} z_H &= w_H x + b_H \\ a_H &= h_H(z_H) \\ z_O &= w_O a_H + b_O \\ \mathcal{J} &= J(z_O; y) \end{aligned}$$

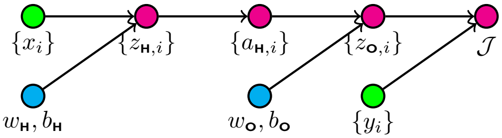
Gradients follow from the chain rule. Previous computations can be reused!

- grad w.r.t. variables: $\frac{\partial \mathcal{J}}{\partial z_O} = J'(z_O)$, $\frac{\partial \mathcal{J}}{\partial a_H} = \frac{\partial \mathcal{J}}{\partial z_O} \underbrace{\frac{\partial z_O}{\partial a_H}}_{=w_O}$, $\frac{\partial \mathcal{J}}{\partial z_H} = \frac{\partial \mathcal{J}}{\partial a_H} \underbrace{\frac{\partial a_H}{\partial z_H}}_{=h'_H(z_H)}$

- grad w.r.t. parameters: $\frac{\partial \mathcal{J}}{\partial b_O} = \frac{\partial \mathcal{J}}{\partial z_O} \underbrace{\frac{\partial z_O}{\partial b_O}}_{=1}$, $\frac{\partial \mathcal{J}}{\partial w_O} = \frac{\partial \mathcal{J}}{\partial z_O} \underbrace{\frac{\partial z_O}{\partial w_O}}_{=a_H}$ can reuse $\partial \mathcal{J}/\partial z_O$!

  $\frac{\partial \mathcal{J}}{\partial b_H} = \frac{\partial \mathcal{J}}{\partial z_H} \underbrace{\frac{\partial z_H}{\partial b_H}}_{=1}$, $\frac{\partial \mathcal{J}}{\partial w_H} = \frac{\partial \mathcal{J}}{\partial z_H} \underbrace{\frac{\partial z_H}{\partial w_H}}_{=x}$ can reuse $\partial \mathcal{J}/\partial z_H$!

Simple case 2: Scalar variables and batch size $B > 1$



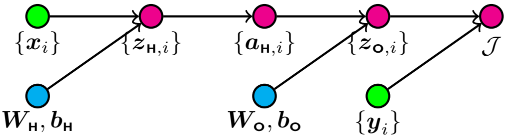
$$\begin{aligned} z_{\mathsf{H},i} &= w_{\mathsf{H}} x_i + b_{\mathsf{H}}, \ \forall i \\ a_{\mathsf{H},i} &= h_{\mathsf{H}}(z_{\mathsf{H},i}), \ \forall i \\ z_{\mathsf{O},i} &= w_{\mathsf{O}} a_{\mathsf{H},i} + b_{\mathsf{O}}, \ \forall i \\ \mathcal{J} &= \textstyle\sum_i J(z_{\mathsf{O},i}; y_i) \end{aligned}$$

Now we need the multivariable chain rule for the parameter gradients:

- grad w.r.t. variables: $\dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{O},i}} = J'(z_{\mathsf{O},i}), \quad \dfrac{\partial \mathcal{J}}{\partial a_{\mathsf{H},i}} = \dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{O},i}} \underbrace{\dfrac{\partial z_{\mathsf{O},i}}{\partial a_{\mathsf{H},i}}}_{=w_{\mathsf{O}}}, \quad \dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{H},i}} = \dfrac{\partial \mathcal{J}}{\partial a_{\mathsf{H},i}} \underbrace{\dfrac{\partial a_{\mathsf{H},i}}{\partial z_{\mathsf{H},i}}}_{=h'_{\mathsf{H}}(z_{\mathsf{H},i})}$
- grad w.r.t. parameters: $\dfrac{\partial \mathcal{J}}{\partial b_{\mathsf{O}}} = \sum_i \dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{O},i}} \underbrace{\dfrac{\partial z_{\mathsf{O},i}}{\partial b_{\mathsf{O}}}}_{=1}, \quad \dfrac{\partial \mathcal{J}}{\partial w_{\mathsf{O}}} = \sum_i \dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{O},i}} \underbrace{\dfrac{\partial z_{\mathsf{O},i}}{\partial w_{\mathsf{O}}}}_{=a_{\mathsf{H},i}}$

$$\frac{\partial \mathcal{J}}{\partial b_{\mathsf{H}}} = \sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H},i}} \underbrace{\frac{\partial z_{\mathsf{H},i}}{\partial b_{\mathsf{H}}}}_{=1}, \qquad \frac{\partial \mathcal{J}}{\partial w_{\mathsf{H}}} = \sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H},i}} \underbrace{\frac{\partial z_{\mathsf{H},i}}{\partial w_{\mathsf{H}}}}_{=x_i}$$

Practical case: Vector variables and batch size $B > 1$



$$\begin{aligned}
\boldsymbol{z}_{\mathsf{H},i} &= \boldsymbol{W}_{\mathsf{H}}\boldsymbol{x}_i + \boldsymbol{b}_{\mathsf{H}}, \ \forall i \\
\boldsymbol{a}_{\mathsf{H},i} &= h_{\mathsf{H}}(\boldsymbol{z}_{\mathsf{H},i}), \ \forall i \\
\boldsymbol{z}_{\mathsf{O},i} &= \boldsymbol{W}_{\mathsf{O}}\boldsymbol{a}_{\mathsf{H},i} + \boldsymbol{b}_{\mathsf{O}}, \ \forall i \\
\mathcal{J} &= \textstyle\sum_i J(\boldsymbol{z}_{\mathsf{O},i}; \boldsymbol{y}_i)
\end{aligned}$$

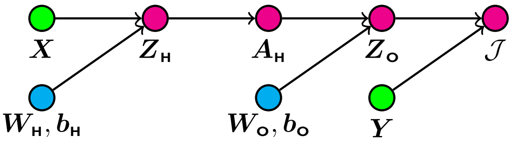
Now we also use the multivariable chain rule for gradient w.r.t. $a_{\mathsf{H},il}$:

- grad wrt variables : $\dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{O},ik}} = \dfrac{\partial J(\boldsymbol{z}_{\mathsf{O},i})}{\partial z_{\mathsf{O},ik}}$, $\quad \dfrac{\partial \mathcal{J}}{\partial a_{\mathsf{H},il}} = \displaystyle\sum_k \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O},ik}} \underbrace{\frac{\partial z_{\mathsf{O},ik}}{\partial a_{\mathsf{H},il}}}_{=w_{\mathsf{O},kl}}$, $\quad \dfrac{\partial \mathcal{J}}{\partial z_{\mathsf{H},il}} = \dfrac{\partial \mathcal{J}}{\partial a_{\mathsf{H},il}} \underbrace{\dfrac{\partial a_{\mathsf{H},il}}{\partial z_{\mathsf{H},il}}}_{=h'_{\mathsf{H}}(z_{\mathsf{H},il})}$
- grad wrt params :

$$\frac{\partial \mathcal{J}}{\partial b_{\mathsf{O},k}} = \sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O},ik}} \underbrace{\frac{\partial z_{\mathsf{O},ik}}{\partial b_{\mathsf{O},k}}}_{=1}, \qquad \frac{\partial \mathcal{J}}{\partial w_{\mathsf{O},kl}} = \sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O},ik}} \underbrace{\frac{\partial z_{\mathsf{O},ik}}{\partial w_{\mathsf{O},kl}}}_{=a_{\mathsf{H},il}}$$

$$\frac{\partial \mathcal{J}}{\partial b_{\mathsf{H},l}} = \sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H},il}} \underbrace{\frac{\partial z_{\mathsf{H},il}}{\partial b_{\mathsf{H},l}}}_{=1}, \qquad \frac{\partial \mathcal{J}}{\partial w_{\mathsf{H},lj}} = \sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H},il}} \underbrace{\frac{\partial z_{\mathsf{H},il}}{\partial w_{\mathsf{H},lj}}}_{=x_{ij}}$$

Practical case: Matrix/vector formulation



$$
\begin{aligned}
\boldsymbol{Z}_{\mathsf{H}}^{\mathsf{T}} &= \boldsymbol{W}_{\mathsf{H}}\boldsymbol{X}^{\mathsf{T}} + \boldsymbol{b}_{\mathsf{H}}\mathbf{1}_B^{\mathsf{T}} \\
\boldsymbol{A}_{\mathsf{H}} &= h_{\mathsf{H}}(\boldsymbol{Z}_{\mathsf{H}}) \\
\boldsymbol{Z}_{\mathsf{O}}^{\mathsf{T}} &= \boldsymbol{W}_{\mathsf{O}}\boldsymbol{A}_{\mathsf{H}}^{\mathsf{T}} + \boldsymbol{b}_{\mathsf{O}}\mathbf{1}_B^{\mathsf{T}} \\
\mathcal{J} &= \textstyle\sum_{i=1}^{B} J(\boldsymbol{z}_{\mathsf{O},i}; \boldsymbol{y}_i)
\end{aligned}
$$

- Define $\boldsymbol{X} \triangleq [\boldsymbol{x}_1, \dots, \boldsymbol{x}_B]^{\mathsf{T}}$, $\boldsymbol{Z}_{\mathsf{H}} \triangleq [\boldsymbol{z}_{\mathsf{H},1}, \dots, \boldsymbol{z}_{\mathsf{H},B}]^{\mathsf{T}}$, $\boldsymbol{A}_{\mathsf{H}} \triangleq [\boldsymbol{a}_{\mathsf{H},1}, \dots, \boldsymbol{a}_{\mathsf{H},B}]^{\mathsf{T}}$, $\boldsymbol{Y} \triangleq [\boldsymbol{y}_1, \dots, \boldsymbol{y}_B]^{\mathsf{T}}$, $\boldsymbol{Z}_{\mathsf{O}} \triangleq [\boldsymbol{z}_{\mathsf{O},1}, \dots, \boldsymbol{z}_{\mathsf{O},B}]^{\mathsf{T}}$
- grad wrt variables : $\left[\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right]_{ik} = \frac{\partial J(\boldsymbol{z}_{\mathsf{O},i})}{\partial z_{\mathsf{O},ik}}$, $\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{A}_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\boldsymbol{W}_{\mathsf{O}}$, $\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{A}_{\mathsf{H}}} \odot h'_{\mathsf{H}}(\boldsymbol{Z}_{\mathsf{H}})$
- grad wrt params :

$$
\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}_{\mathsf{O}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right)^{\mathsf{T}}\mathbf{1}_B \in \mathbb{R}^{d_{\mathsf{O}}}, \qquad \frac{\partial \mathcal{J}}{\partial \boldsymbol{W}_{\mathsf{O}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right)^{\mathsf{T}}\boldsymbol{A}_{\mathsf{H}} \in \mathbb{R}^{d_{\mathsf{O}} \times d_{\mathsf{H}}}
$$

$$
\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}_{\mathsf{H}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}}\right)^{\mathsf{T}}\mathbf{1}_B \in \mathbb{R}^{d_{\mathsf{H}}}, \qquad \frac{\partial \mathcal{J}}{\partial \boldsymbol{W}_{\mathsf{H}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}}\right)^{\mathsf{T}}\boldsymbol{X} \in \mathbb{R}^{d_{\mathsf{H}} \times d}
$$

- Above, $\odot$ denotes the elementwise or "Hadamard" product

Summary of forward and backward passes

So, to compute the gradients w.r.t. the parameters $\boldsymbol{\theta} = (\boldsymbol{W}_{\mathsf{H}}, \boldsymbol{b}_{\mathsf{H}}, \boldsymbol{W}_{\mathsf{O}}, \boldsymbol{b}_{\mathsf{O}})$, we ...

- first perform the forward pass:

$$\begin{aligned}
\boldsymbol{Z}_{\mathsf{H}}^{\mathsf{T}} &= \boldsymbol{W}_{\mathsf{H}}\boldsymbol{X}^{\mathsf{T}} + \boldsymbol{b}_{\mathsf{H}}\boldsymbol{1}_B^{\mathsf{T}}\\
\boldsymbol{A}_{\mathsf{H}} &= h_{\mathsf{H}}(\boldsymbol{Z}_{\mathsf{H}})\\
\boldsymbol{Z}_{\mathsf{O}}^{\mathsf{T}} &= \boldsymbol{W}_{\mathsf{O}}\boldsymbol{A}_{\mathsf{H}}^{\mathsf{T}} + \boldsymbol{b}_{\mathsf{O}}\boldsymbol{1}_B^{\mathsf{T}}\\
\mathcal{J} &= \textstyle\sum_i J(\boldsymbol{z}_{\mathsf{O},i};\boldsymbol{y}_i)
\end{aligned}$$

- then the backward pass:

$$\begin{aligned}
\left[\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right]_{ik} &= \frac{\partial J(\boldsymbol{z}_{\mathsf{O},i};\boldsymbol{y}_i)}{\partial z_{\mathsf{O},ik}} \;\forall i = 1\dots B,\; k = 1\dots d_{\mathsf{O}}\\
\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}_{\mathsf{O}}} &= \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right)^{\mathsf{T}}\boldsymbol{1}_B \quad\text{and}\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{W}_{\mathsf{O}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right)^{\mathsf{T}}\boldsymbol{A}_{\mathsf{H}}\\
\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}} &= \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\boldsymbol{W}_{\mathsf{O}}\right) \odot h'_{\mathsf{H}}(\boldsymbol{Z}_{\mathsf{H}})\\
\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}_{\mathsf{H}}} &= \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}}\right)^{\mathsf{T}}\boldsymbol{I}_B \quad\text{and}\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{W}_{\mathsf{H}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}}\right)^{\mathsf{T}}\boldsymbol{X}
\end{aligned}$$

Called "back-propagation," since gradient computations work *backwards* from end!

Examples of loss derivative

- With binary cross-entropy loss, we have $d_{\mathsf{o}} = 1$ and

$$\mathcal{J} = \textstyle\sum_{i=1}^{B} J(z_{\mathsf{o},i}, y_i) \quad \text{with} \quad y_i \in \{0,1\}$$

$$\text{where} \quad J(z_{\mathsf{o},i}, y_i) = \ln\left(1 + e^{z_{\mathsf{o},i}}\right) - y_i z_{\mathsf{o},i}$$

$$\Rightarrow \quad \frac{\partial J(z_{\mathsf{o},i}, y_i)}{\partial z_{\mathsf{o},i}} = \frac{e^{z_{\mathsf{o},i}}}{1 + e^{z_{\mathsf{o},i}}} - y_i$$

- With K-ary cross-entropy loss, we have $d_{\mathsf{o}} = K$ and

$$\mathcal{J} = \sum_{i=1}^{B} J(\boldsymbol{z}_{\mathsf{o},i}, \boldsymbol{y}_i) \quad \text{with} \quad y_{ik} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{if } y_i \neq k \end{cases}$$

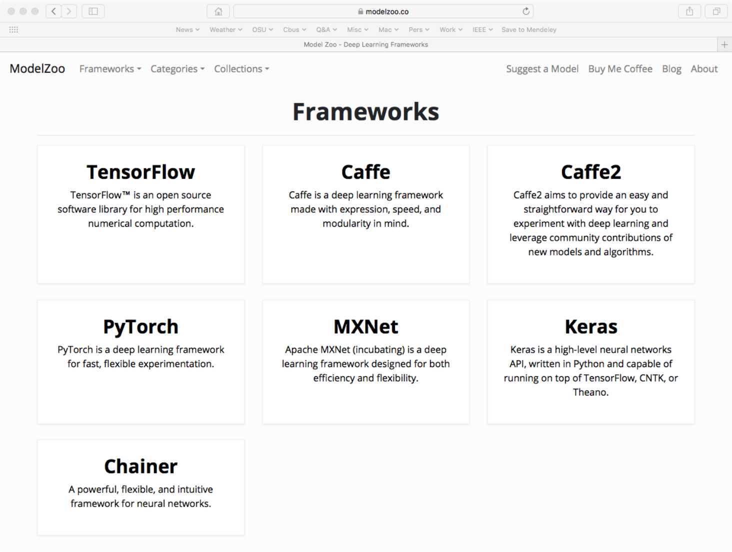
$$\text{where} \quad J(\boldsymbol{z}_{\mathsf{o},i}, \boldsymbol{y}_i) = \ln\left(\sum_{l=1}^{K} e^{z_{\mathsf{o},il}}\right) - \sum_{l=1}^{K} y_{il} z_{\mathsf{o},il}$$

$$\Rightarrow \quad \frac{\partial J(\boldsymbol{z}_{\mathsf{o},i}, \boldsymbol{y}_i)}{\partial z_{\mathsf{o},ik}} = \frac{e^{z_{\mathsf{o},ik}}}{\sum_{l=1}^{K} e^{z_{\mathsf{o},il}}} - y_{ik}$$

Outline

Frameworks for implementing neural nets

The frameworks most popular today:



Top 3 current frameworks

- TensorFlow (2015)
  - Widespread and mature for deployment
  - Original version was difficult to use. Newest version is better, but not great
- Keras (runs on top of TensorFlow or Theano or CNTK)
  - Very convenient interface for simple/standard tasks
  - Difficult to customize and debug for complex/nonstandard tasks
- PyTorch (2017)
  - Easier than TensorFlow to use and just as powerful
  - Most popular in research/academic community

We will use PyTorch! See the following for more about pros & cons:

- PyTorch vs. TensorFlow: Towards Data Science (6/20)
- PyTorch vs. TensorFlow vs. Keras: Simplilearn (9/20), The Startup (7/20)

PyTorch recipe

1) Construct the dataset and dataloader objects
2) Construct the network
   - # hidden units, # output units, activations, etc ...
3) Select the optimizer and loss criterion
4) Fit the network parameters
5) Apply the network

PyTorch

Great tutorials at `https://pytorch.org/tutorials`

1) Construct the dataset and dataloader objects

```python
import torch
```

```python
import torch.utils.data

# Convert the numpy arrays to Tensor type
X_torch = torch.Tensor(X)
y_torch = torch.Tensor(y)

# Create a Dataset from the Tensors
dataset = torch.utils.data.TensorDataset(X_torch, y_torch)
# Create a DataLoader from the Dataset
loader = torch.utils.data.DataLoader(dataset, batch_size=100)
```

- `torch.Tensor`: the datatype used by PyTorch for multi-dimensional matrices
- `dataset`: the object used to hold the training data
- `DataLoader`: adds random sampling and multi-processor support to the Dataset
- Later we will draw mini-batches via:

```python
for batch, data in enumerate(loader):
    x_batch,y_batch = data
```

2) Construct the network

- The neural net is described by a `torch.nn.Module`
  - `__init__()` defines the network components
  - `forward()` defines one forward pass through the network
- We instantiate the neural net as "`model`"

```
import torch.nn as nn

nh = 4

# nin: dimension of input data
# nh: number of hidden units
# nout: number of outputs = 1 since this is bi

class Net(nn.Module):
    def __init__(self,nin,nh,nout):
        super(Net,self).__init__()
        self.activation = nn.Sigmoid()
        self.Dense1 = nn.Linear(nin,nh)
        self.Dense2 = nn.Linear(nh,nout)

    def forward(self,x):
        x = self.activation(self.Dense1(x))
        out = self.activation(self.Dense2(x))
        return out

model = Net(nin=nx,nh=nh,nout=1)
```

3) Select the optimizer and loss criterion

```
import torch.optim as optim

opt = optim.Adam(model.parameters(), lr=0.01)
criterion = nn.BCELoss()
```

- Learning algorithms are stored as "optimizer objects" in `torch.optim`
  - Many options, e.g., SGD, Rprop, RMSprop, AdaDelta, AdaGrad, Adam, etc.
  - Some descriptions can be found here, here, and here
- We instantiated the Adam optimizer as "`opt`" using ...
  - the network parameters $\boldsymbol{\theta}$ that we want to optimize
  - Adam's algorithmic parameters (i.e., learning rate)
- The `torch.nn` library includes many loss criteria
  - Examples: BCEloss, CrossEntropyLoss, MSELoss (i.e., RSS), L1Loss, etc.
  - These can be scaled and combined (e.g., "MSELoss $+ \lambda$ L1Loss" for Lasso)

4) Fit the network parameters

For each mini-batch $\{\boldsymbol{x}_i\}$:

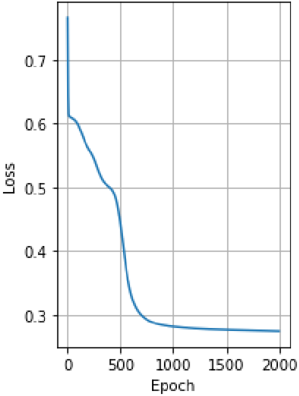
- compute outputs $\{a_{\mathsf{o},i}\}$
- compute loss $\mathcal{J}(\boldsymbol{\theta}^t)$
- compute gradient $\nabla\mathcal{J}(\boldsymbol{\theta}^t)$
- update parameters $\boldsymbol{\theta}^{t+1}$
- record loss & accuracy

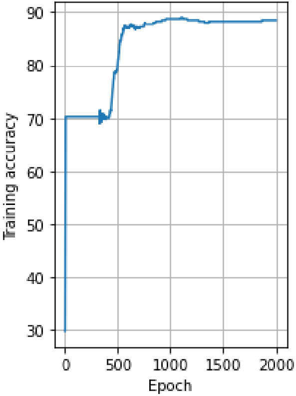
```
num_epoch = 2000

print_intvl = 100
avg_loss = np.zeros([num_epoch])
avg_acc = np.zeros([num_epoch])

# Outer loop over epochs
for epoch in range(num_epoch):
    error = 0 # initialize error counter
    total = 0 # initialize total counter
    batch_loss = []
    # Inner loop over mini-batches
    for batch, data in enumerate(loader):
        x_batch,y_batch = data
        y_batch = y_batch.view(-1,1) # resizes y_batch to (batch_size,1)
        out = model(x_batch)
        # Compute loss
        loss = criterion(out,y_batch)
        batch_loss.append(loss.item())
        # Compute gradients using back-propagation
        opt.zero_grad()
        loss.backward()
        # Take an optimization 'step' (i.e., update parameters)
        opt.step()
        # Do hard decision
        guess = out.round()
        # Compute number of decision errors
        error += torch.sum(torch.abs(guess - y_batch))
        total += len(y_batch)

    acc = 100*(1-error/total) # Compute accuracy over epoch
    avg_loss[epoch] = np.mean(batch_loss) # Compute average loss over epoch
    avg_acc[epoch] = acc
```

Performance vs. epoch

Over the epochs . . .

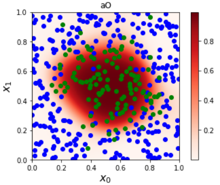
- Loss gradually decreases
- Classification accuracy gradually increases





5) Applying the network: Visualizing the decision region

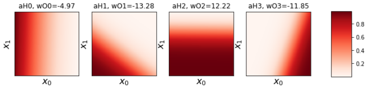
- Xplot created with rows densely sampling $\boldsymbol{x}^\mathsf{T}$ over $[0,1]^2$
- Xplot recast as a `torch.Tensor`
- `model()` applies network
  - It calls `model.__call__()`, which eventually calls the `forward` method and does other book-keeping
- `.detach().numpy()` extracts the data and converts it to NumPy




```
# Limits to plot the response.
xmin = [0,0]
xmax = [1,1]

# Use meshgrid to create the 2D input
nplot = 100
x0plot = np.linspace(xmin[0],xmax[1],nplot)
x1plot = np.linspace(xmin[0],xmax[1],nplot)
x0mat, x1mat = np.meshgrid(x0plot,x1plot)
Xplot = np.column_stack([x0mat.ravel(), x1mat.ravel()])
Xplot_tensor = torch.Tensor(Xplot)

# Compute the output and export to numpy
a0plot = model(Xplot_tensor).detach().numpy()
a0plot_mat = a0plot[:,0].reshape((nplot, nplot))

# Plot the recovered region
plt.imshow(np.flipud(a0plot_mat), extent=[xmin[0],xmax[0]
plt.colorbar()

# Overlay the samples
I0 = np.where(y==0)[0]
I1 = np.where(y==1)[0]
plt.plot(X[I0,0], X[I0,1], 'bo')
plt.plot(X[I1,0], X[I1,1], 'go')
plt.xlabel('$x_0$', fontsize=16)
plt.ylabel('$x_1$', fontsize=16)
plt.title('a0');
```

5) Applying the network: Visualizing the hidden layer

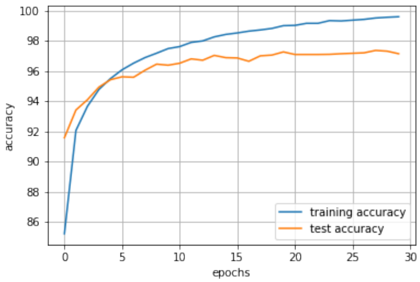
- `model.Dense1()` applies the first linear stage
- `model.activation()` applies the activation
- `.detach().numpy()` extracts the data and converts it to NumPy
- `model.state_dict()` exports the model parameters




```
# Get the outputs of the hidden layer
ahid = model.activation(model.Dense1(Xplot_tensor))
ahid_plot = ahid.detach().numpy()
ahid_plot = ahid_plot.reshape((nplot,nplot,nh))

# Get the weights of the output layer
state_dict = model.state_dict()
Wo, bo = state_dict['Dense2.weight'], state_dict['Dense2.bias

fig = plt.figure(figsize=(10, 4))

for i in range(nh):
    plt.subplot(1,nh,i+1)
    ahid_ploti = np.flipud(ahid_plot[:,:,i])
    im = plt.imshow(ahid_ploti, extent=[xmin[0],xmax[0],xmin[
    plt.xticks([])
    plt.yticks([])
    plt.xlabel('$x_0$', fontsize=16)
    plt.ylabel('$x_1$', fontsize=16)
    plt.title('aH{0:d}, wO{0:d}={1:4.2f}'.format(i,Wo[0,i]))

fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.9, 0.30, 0.05, 0.4])
fig.colorbar(im, cax=cbar_ax);
```

Application to MNIST

- In a second demo, we use a 2-layer neural network for MNIST digit classification
- For this, we used
  - $d_{\mathsf{H}} = 100$ hidden nodes
  - $d_{\mathsf{O}} = 10$ output nodes (10 classes)
  - cross-entropy loss
  - sigmoidal activation fxn $h_{\mathsf{H}}(\cdot)$
  - the `Adam` optimizer with $\alpha = 10^{-3}$
  - 50,000 training samples
  - 10,000 test samples



- We achieved a test accuracy of $\approx 97\%$. Not bad!
  - Accuracy is similar to SVM. But neural net is much faster to apply!
  - And further fine-tuning could improve neural-net performance

Fine-tuning the implementation

We have several decisions to make when implementing a 2-layer neural network:

- Network details
  - Number of hidden units d_{H}
  - Type of hidden-layer activation functions $h_{\mathsf{H}}(\cdot)$
- Optimizer details
  - which optimizer (e.g., `Adam`)
  - learning rate (and how to schedule it over the epochs)
  - batch size
  - # epochs
- Regularization
  - L2 regularization can be implemented via optimizer's `weight_decay` option

Packages like Optuna can be used to tune all these hyperparameters

PyTorch loss options: Be careful!

- Throughout these lecture notes, the loss $\mathcal{J}(\boldsymbol{\theta})$ has been defined using the output-layer linear scores $\boldsymbol{z}_{\mathsf{o}}$, also called logits. For example,
  - Binary cross-entropy: $J(\boldsymbol{\theta}) = \sum_i \big(\ln[1 + e^{z_{\mathsf{o},i}}] - y_i z_{\mathsf{o},i} \big)$
- But the loss can also be written in terms of the output-layer activations $\boldsymbol{a}_{\mathsf{o}}$:
  - Binary cross-entropy: $J(\boldsymbol{\theta}) = -\sum_i \big(y_i \ln a_{\mathsf{o},i} + [1 - y_i] \ln[1 - a_{\mathsf{o},i}] \big)$
    with sigmoid activation $a_{\mathsf{o},i} = \frac{1}{1+e^{-z_{\mathsf{o},i}}}$
- PyTorch allows the loss to be defined in either way, so be careful!
  - `nn.BCEWithLogitsLoss`: takes logits $\{z_{\mathsf{o},i}\}$ as input
  - `nn.BCELoss`: takes sigmoid activations $\{a_{\mathsf{o},i}\}$ as input
  - `nn.CrossEntropyLoss`: takes logits $\{\boldsymbol{z}_{\mathsf{o},i}\}$ as input
  - `nn.NLLLoss`: takes log-softmax activations $\{\boldsymbol{a}_{\mathsf{o},i}\}$ as input
- For more discussion, see this blog post

Learning objectives

- Understand 2-layer feedforward neural networks:
  - Motivation: learning feature transformations
  - Network architecture: linear and nonlinear layers
  - Choice of activation functions and training loss
- Understand mini-batch training and stochastic gradient descent
- Understand the back-propagation approach to gradient computation
- Know how to implement a neural network using PyTorch