Unit 8
Neural Networks

Prof. Phil Schniter

The Ohio State University

ECE 4300: Introduction to Machine Learning, Sp20

Learning objectives

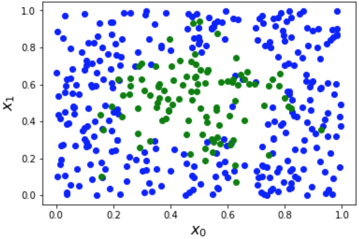
- Understand 2-layer feedforward neural networks:
  - Motivation: learning feature transformations
  - Network architecture: linear and nonlinear layers
  - Choice of activation functions and training loss
- Understand mini-batch training and stochastic gradient descent
- Understand the back-propagation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

Outline

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

Dealing with data that is not linearly separable

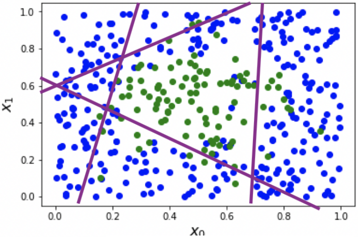
- Consider data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ on right:
  - Features $\boldsymbol{x}_i = [x_{i0}, x_{i1}]^\mathsf{T} \in \mathbb{R}^2$
  - Labels $y_i \in \{0, 1\}$
  - Not linearly separable!



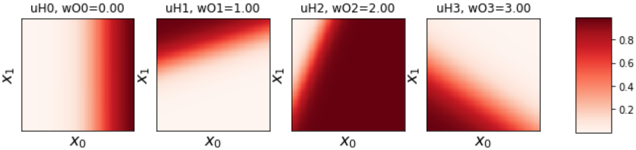
- Could use a feature transformation to enhance linear separability, and then apply linear classification on the transformed features
  - But what if we don't know a good transformation?
  - Can we learn one?
  - Yes!

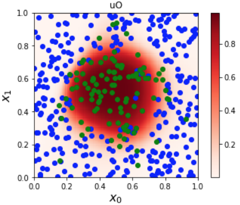
A two-stage approach to classification

- 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



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

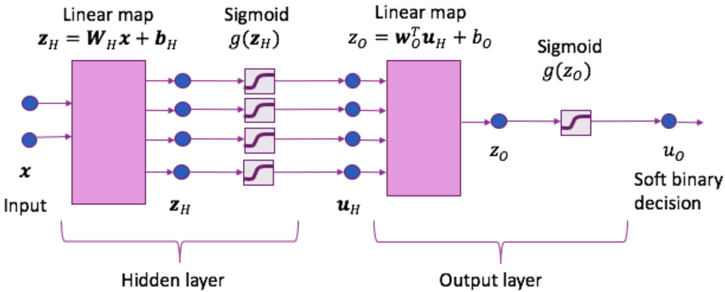




- The overall approach is successful in classifying the data!

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: $\boldsymbol{u}_{\mathsf{H}} = g(\boldsymbol{z}_{\mathsf{H}}) \in [0,1]^{d_{\mathsf{H}}}$
  - $\boldsymbol{u}_{\mathsf{H}}$ are the learned features
- Stage 2: "Output layer"
  - score: $z_{\mathsf{O}} = \boldsymbol{w}_{\mathsf{O}}^{\mathsf{T}}\boldsymbol{u}_{\mathsf{H}} + b_{\mathsf{O}} \in \mathbb{R}$
  - soft output: $u_{\mathsf{O}} = g(z_{\mathsf{O}}) \in [0,1]$
  - u_{O} is the final $\Pr\{y=1 \,|\, \boldsymbol{x}\}$
- If we use logistic regression for both, then $g(z) = \frac{1}{1+e^{-z}}$, a "sigmoid"
- This is a two-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

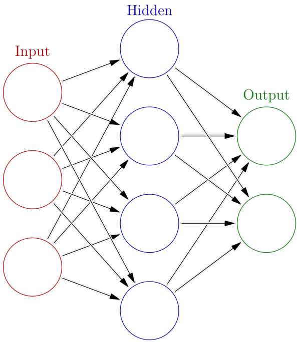
$$\begin{aligned} \widehat{\boldsymbol{\theta}}_{\mathsf{ml}} &= \arg\min_{\boldsymbol{\theta}} \textstyle\sum_{i=1}^n \big[-\ln p(y_i \,|\, \boldsymbol{x}_i; \boldsymbol{\theta}) \big] \\ &= \arg\min_{\boldsymbol{\theta}} \textstyle\sum_{i=1}^n \big(\ln[1 + e^{z_{\mathsf{O},i}}] - y_i z_{\mathsf{O},i}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

  - We will discuss how to do this optimization later

Outline

General structure of feed-forward neural network

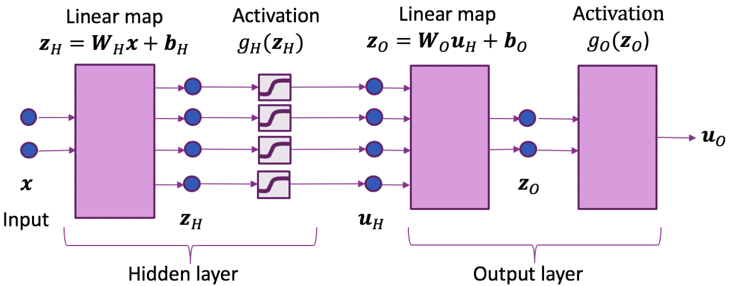
- Input: $\boldsymbol{x} = [x_1, \ldots, x_d]^\mathsf{T}$
  - $d =$ number of features (or inputs)
- Hidden layer:
  - $u_{\mathsf{H},l} = g_\mathsf{H}\Big(b_{\mathsf{H},l} + \sum_{j=1}^{d} w_{\mathsf{H},lj} x_j\Big), \ l = 1 \ldots d_\mathsf{H}$
  - $g_\mathsf{H}(\cdot)$ is a nonlinear "activation"
  - $d_\mathsf{H} =$ number of hidden units
- Output layer:
  - $u_{\mathsf{O},k} = g_\mathsf{O}\Big(b_{\mathsf{O},k} + \sum_{l=1}^{d_\mathsf{H}} w_{\mathsf{O},kl} u_{\mathsf{H},l}\Big), \ k = 1 \ldots d_\mathsf{O}$
  - $g_\mathsf{O}(\cdot)$ determines if regression or classification net
  - $d_\mathsf{O} =$ number of outputs



- Can use more than two layers ("deep network"), but for now we focus on two
  - Other variations: http://www.asimovinstitute.org/neural-network-zoo/

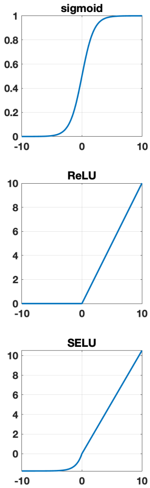
Vector/matrix representation of feed-forward neural network

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



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

- sigmoid: $g_{\mathsf{H}}(z) = \frac{1}{1+e^{-z}} \triangleq \sigma(z)$
  - values are bounded, but not centered
  - often used in shallow networks
- tanh: $g_{\mathsf{H}}(z) = \tanh(z) = 2\sigma(z) - 1$
  - values are bounded and centered
- rectified linear unit (ReLU): $g_{\mathsf{H}}(z) = \max\{0, z\}$
  - most popular choice in deep networks
- scaled exponential linear units (SELU):
  - $g_{\mathsf{H}}(z) = \begin{cases} \lambda z & z \geq 0 \\ \lambda\alpha(e^z - 1) & z < 0 \end{cases}$ with $\begin{array}{l} \lambda = 1.0507 \\ \alpha = 1.67326 \end{array}$
  - preserve mean=0 & variance=1 from input to output
  - network weights must be properly initialized



Common choices for output-layer activation $g_{\mathsf{o}}(\boldsymbol{z}_{\mathsf{o}})$

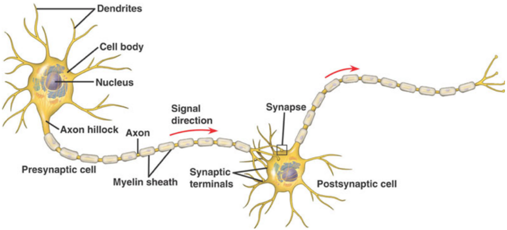
- Binary classification:
  - $\boldsymbol{z}_{\mathsf{o}} = z_{\mathsf{o}}$ is a scalar
  - Hard decision: $\widehat{y} = \operatorname{sgn}(z_{\mathsf{o}})$
  - Soft decision: $\Pr\{y=1 \,|\, \boldsymbol{x}\} = \dfrac{1}{1+e^{-z_{\mathsf{o}}}}$ (logistic)
- Multiclass classification with K classes:
  - $\boldsymbol{z}_{\mathsf{o}} = [z_{\mathsf{o},1}, \dots, z_{\mathsf{o},K}]^{\mathsf{T}} \in \mathbb{R}^K$
  - Hard decision: $\widehat{y} = \arg\max_{k=1\dots K} z_{\mathsf{o},k}$
  - Soft decision: $\Pr\{y=k \,|\, \boldsymbol{x}\} = \dfrac{e^{z_{\mathsf{o},k}}}{\sum_{l=1}^K e^{z_{\mathsf{o},l}}}$ (softmax)
- Regression with K-dimensional targets (i.e., $\boldsymbol{y}_i \in \mathbb{R}^K$):
  - $\boldsymbol{z}_{\mathsf{o}} = [z_{\mathsf{o},1}, \dots, z_{\mathsf{o},K}]^{\mathsf{T}} \in \mathbb{R}^K$
  - $\widehat{\boldsymbol{y}} = \boldsymbol{z}_{\mathsf{o}}$ (linear)

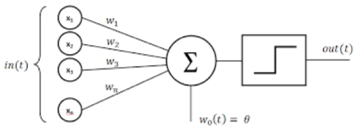
Loss functions for training

The application determines the output activation $g_{\mathsf{o}}(\cdot)$ and the loss function!

- Binary classification:
  - Likelihood: $\Pr\{y=1 \,|\, \boldsymbol{x}\} = \dfrac{1}{1+e^{-z_{\mathsf{o}}}}$ (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)
- Multiclass classification with K classes:
  - Likelihood: $\Pr\{y=k \,|\, \boldsymbol{x}\} = \dfrac{e^{z_{\mathsf{o},k}}}{\sum_{l=1}^{K} e^{-z_{\mathsf{o},l}}}$ (softmax)
  - Loss: $\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \Big(\ln\Big[\sum_{k=1}^{K} e^{z_{\mathsf{o},ik}}\Big] - \sum_{k=1}^{K} y_{ik} z_{\mathsf{o},ik} \Big)$ (cross entropy)
- Regression with K-dimensional targets:
  - Likelihood: $\Pr\{\boldsymbol{y} \,|\, \boldsymbol{x}\} = \mathcal{N}(\boldsymbol{y}; \boldsymbol{z}_{\mathsf{o}}, \sigma_\epsilon^2 \boldsymbol{I})$ (i.i.d. Gaussian)
  - Loss: $\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - z_{\mathsf{o},ik})^2$ (quadratic)

Inspiration from biology





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

History

- Interest in understanding the brain for thousands of years
- 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
- Implementing and Training Neural Nets with PyTorch
- Gradient Computation via Back-Propagation

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

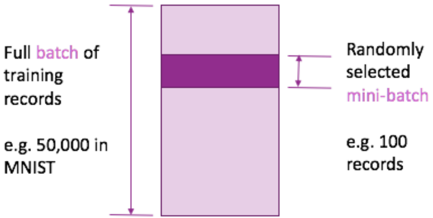
- Can tackle this using the gradient descent (GD) algorithm:

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

  - Each iteration computes n gradients
  - This is too 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 $t = 1 \dots T$:
  - select mini-batch 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

Rationale for stochastic gradient descent

- The gradient approximation $\boldsymbol{g}^t$ is correct on average:

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

- Thus the updated parameters are also correct on average:

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

- 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`

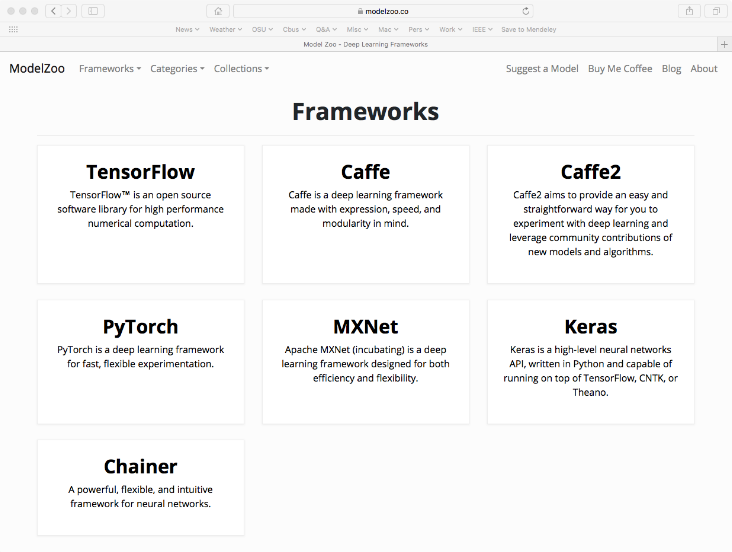
Implementing mini-batch

- Setup:
  - n samples in the training set
  - 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 training epoch goes through the entire training data once
- Implementation:
  - Begin each epoch by randomly shuffling all n training indices i
  - Then partition the shuffled indices into T subsets $\{I_t\}_{t=1}^{T}$
  - Subset I_t defines the mini-batch $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i \in I_t}$ used at GD update step t
    - This way, the mini-batches change over the epochs!

Outline

Frameworks for implementing neural nets

The frameworks most popular today:



Top 3 frameworks

- TensorFlow
  - Advantages: widespread (many examples!), very flexible, runs pretty fast
  - Disadvantages: difficult to code, not well integrated with Python
- Keras (runs on top of TensorFlow or Theano or CNTK)
  - Advantages: very easy to code
  - Disadvantages: difficult to debug and customize (much is hidden), runs slower
- PyTorch
  - Advantages: easy to code & debug, runs fast, excellent Python integration
  - Disadvantages: not as widespread as TensorFlow

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

- PyTorch vs. TensorFlow: The Gradient (10/19), Towards Data Science (2/20)
- PyTorch vs. Keras: deepsense.ai (6/18), fast.ai (10/18)

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

1) Construct the dataset and dataloader objects

```
import torch
```

```
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:

```
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 $\{\boldsymbol{x}_i\}$:

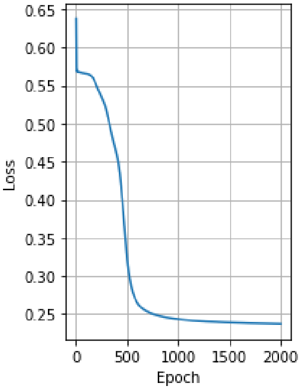
- compute outputs $\{u_{\mathsf{o},i}\}$
- compute loss $\mathcal{J}(\boldsymbol{\theta}^t)$
- compute gradient $\nabla\mathcal{J}(\boldsymbol{\theta}^t)$
- update parameters $\boldsymbol{\theta}^{t+1}$
- record loss & accuracy

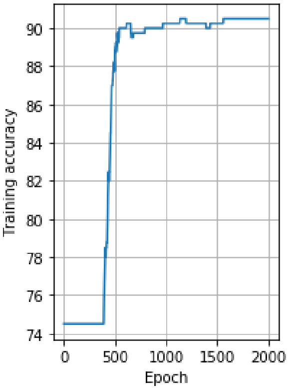
```
num_epoch = 2000

print_intvl = 100
a_loss = np.zeros([num_epoch])
a_accuracy = 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()

        # Compute number of decision errors
        guess = out.round() # hard binary outputs
        error += torch.sum(torch.abs(guess - y_batch))
        total += len(y_batch)

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

Performance vs. epoch

Over the epochs . . .

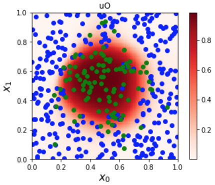
- Loss gradually decreases
- Classification accuracy gradually increases





5) Applying the network: Visualizing the decision region

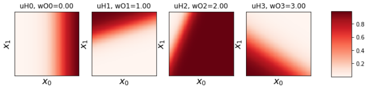
- Xplot created with rows densely sampling $\boldsymbol{x}^{\mathsf{T}}$ over $[0,1]^2$
- Xplot recast as a torch.Tensor
- model() applies network
  - It calls model.__call__(), which eventually calls the forward method and does other book-keeping
- .detach().numpy() extracts the data and converts it 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
u0plot = model(Xplot_tensor).detach().numpy()
u0plot_mat = u0plot[:,0].reshape((nplot, nplot))

# Plot the recovered region
plt.imshow(np.flipud(u0plot_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('u0')
```

5) Applying the network: Visualizing the hidden layer

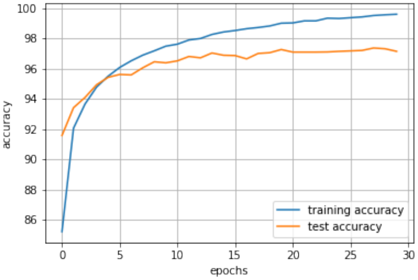
- model.Dense1() applies the first linear stage
- model.activation() applies the activation
- .detach().numpy() extracts the data and converts it to NumPy
- model.state_dict() exports the model parameters




```
# Get the outputs of the hidden layer
uhid = model.activation(model.Dense1(Xplot_tensor))
uhid_plot = uhid.detach().numpy()
uhid_plot = uhid_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)
    uhid_ploti = np.flipud(uhid_plot[:,:,i])
    im = plt.imshow(uhid_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('uH{0:d}, wO{0:d}={1:4.2f}'.format(i,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
  - logistic activation fxn $g_{\mathsf{H}}(\cdot)$
  - 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!
  - And further fine-tuning should 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 $g_{\mathsf{H}}(\cdot)$
- Optimizer details
  - which optimizer (e.g., `Adam`)
  - learning rate
  - # epochs
- Regularization
  - L2 regularization can be implemented via optimizer's `weight_decay` option
  - By default, L2 regularization applied to both weights and biases
  - Can restrict to weights via per-parameter options. (See example here)

Ideally these are all selected via cross-validation!

PyTorch loss options

- Throughout these lecture notes, the loss $\mathcal{J}(\boldsymbol{\theta})$ has been defined on 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 $\boldsymbol{u}_{\mathsf{O}}$:
  - Binary cross-entropy: $J(\boldsymbol{\theta}) = -\sum_i \big(y_i \ln u_{\mathsf{O},i} + [1 - y_i] \ln[1 - u_{\mathsf{O},i}] \big)$ with sigmoid activation $u_{\mathsf{O},i} = \frac{1}{1+e^{-z_{\mathsf{O},i}}}$
- PyTorch allows the loss to be defined in either way, so be careful!
  - `nn.BCEWithLogitsLoss`: takes logits as input
  - `nn.BCELoss`: takes sigmoid activations as input
  - `nn.CrossEntropyLoss`: takes logits as input
  - `nn.NLLLoss`: takes log-softmax activations as input
- For more discussion, see this blog post

Outline

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

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 proposed the mini-batch SGD update

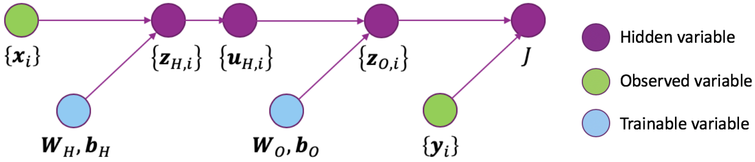
$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \frac{\alpha_t}{|I_t|} \nabla \underbrace{\sum_{i \in I_t} J(\boldsymbol{\theta}^t; \boldsymbol{x}_i, \boldsymbol{y}_i)}_{\triangleq \mathcal{J}^t(\boldsymbol{\theta}^t)}$$

- 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 the 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:

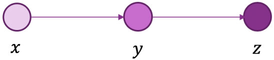
$$\begin{aligned}
\boldsymbol{z}_{\mathsf{H},i} &= \boldsymbol{W}_{\mathsf{H}}\boldsymbol{x}_i + \boldsymbol{b}_{\mathsf{H}}, \quad \forall i = 1 \dots B \\
\boldsymbol{u}_{\mathsf{H},i} &= g_{\mathsf{H}}(\boldsymbol{z}_{\mathsf{H},i}), \quad \forall i = 1 \dots B \\
\boldsymbol{z}_{\mathsf{O},i} &=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

- 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 $\boldsymbol{u}_{\mathsf{O},i}$, as in past lectures
  - PyTorch allows either, but note that the example on page 28 used $\boldsymbol{u}_{\mathsf{O},i}$

Review of the chain rule

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

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



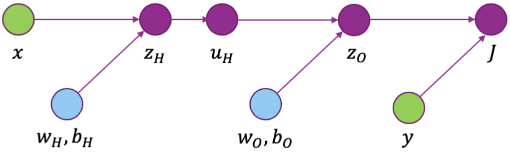
- Now suppose that another variable u is inserted. Previous expression still holds!

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

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

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

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



$$\begin{aligned} z_{\mathsf{H}} &= w_{\mathsf{H}} x + b_{\mathsf{H}} \\ u_{\mathsf{H}} &= g_{\mathsf{H}}(z_{\mathsf{H}}) \\ z_{\mathsf{O}} &= w_{\mathsf{O}} u_{\mathsf{H}} + b_{\mathsf{O}} \\ \mathcal{J} &= J(z_{\mathsf{O}}; y) \end{aligned}$$

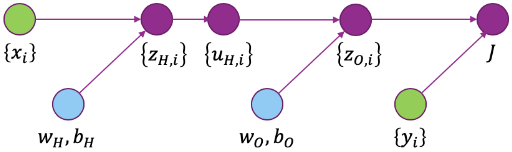
Gradients follow from the chain rule. Previous computations can be reused!

- grad w.r.t. variables:
$$\frac{\partial \mathcal{J}}{\partial z_{\mathsf{O}}} = J'(z_{\mathsf{O}}), \qquad \frac{\partial \mathcal{J}}{\partial u_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O}}} \underbrace{\frac{\partial z_{\mathsf{O}}}{\partial u_{\mathsf{H}}}}_{=w_{\mathsf{O}}}, \qquad \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial u_{\mathsf{H}}} \underbrace{\frac{\partial u_{\mathsf{H}}}{\partial z_{\mathsf{H}}}}_{=g'_{\mathsf{H}}(z_{\mathsf{H}})}$$

- grad w.r.t. parameters:
$$\frac{\partial \mathcal{J}}{\partial b_{\mathsf{O}}} = \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O}}} \underbrace{\frac{\partial z_{\mathsf{O}}}{\partial b_{\mathsf{O}}}}_{=1}, \qquad \frac{\partial \mathcal{J}}{\partial w_{\mathsf{O}}} = \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O}}} \underbrace{\frac{\partial z_{\mathsf{O}}}{\partial w_{\mathsf{O}}}}_{=u_{\mathsf{H}}}$$
can reuse $\partial \mathcal{J} / \partial z_{\mathsf{O}}$!

$$\frac{\partial \mathcal{J}}{\partial b_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H}}} \underbrace{\frac{\partial z_{\mathsf{H}}}{\partial b_{\mathsf{H}}}}_{=1}, \qquad \frac{\partial \mathcal{J}}{\partial w_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial z_{\mathsf{H}}} \underbrace{\frac{\partial z_{\mathsf{H}}}{\partial w_{\mathsf{H}}}}_{=x}$$
can reuse $\partial \mathcal{J} / \partial z_{\mathsf{H}}$!

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



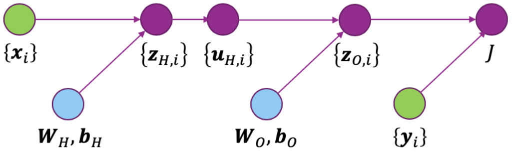
$$z_{\mathsf{H},i} = w_{\mathsf{H}} x_i + b_{\mathsf{H}}, \ \forall i$$
$$u_{\mathsf{H},i} = g_{\mathsf{H}}(z_{\mathsf{H},i}), \ \forall i$$
$$z_{\mathsf{O},i} = w_{\mathsf{O}} u_{\mathsf{H},i} + b_{\mathsf{O}}, \ \forall i$$
$$\mathcal{J} = \textstyle\sum_i J(z_{\mathsf{O},i}; y_i)$$

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

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

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

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



$$\boldsymbol{z}_{\mathsf{H},i} = \boldsymbol{W}_{\mathsf{H}} \boldsymbol{x}_i + \boldsymbol{b}_{\mathsf{H}}, \ \forall i$$
$$\boldsymbol{u}_{\mathsf{H},i} = g_{\mathsf{H}}(\boldsymbol{z}_{\mathsf{H},i}), \ \forall i$$
$$\boldsymbol{z}_{\mathsf{O},i} = \boldsymbol{W}_{\mathsf{O}} \boldsymbol{u}_{\mathsf{H},i} + \boldsymbol{b}_{\mathsf{O}}, \ \forall i$$
$$\mathcal{J} = \textstyle\sum_i J(\boldsymbol{z}_{\mathsf{O},i}; \boldsymbol{y}_i)$$

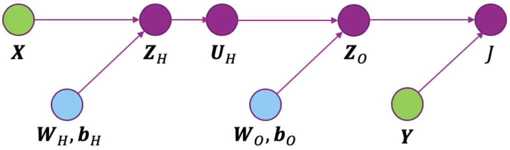
Now we also use the multivariable chain rule for gradient w.r.t. $u_{\mathsf{H},il}$:

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

- grad wrt params : $\dfrac{\partial \mathcal{J}}{\partial b_{\mathsf{O},k}} = \displaystyle\sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O},ik}} \underbrace{\frac{\partial z_{\mathsf{O},ik}}{\partial b_{\mathsf{O},k}}}_{=1}$, $\quad \dfrac{\partial \mathcal{J}}{\partial w_{\mathsf{O},kl}} = \displaystyle\sum_i \frac{\partial \mathcal{J}}{\partial z_{\mathsf{O},ik}} \underbrace{\frac{\partial z_{\mathsf{O},ik}}{\partial w_{\mathsf{O},kl}}}_{=u_{\mathsf{H},il}}$

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

Practical case: Matrix/vector formulation



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

- Define $\boldsymbol{X} \triangleq [\boldsymbol{x}_1, \dots, \boldsymbol{x}_B]^{\mathsf{T}}$, $\boldsymbol{Z}_{\mathsf{H}} \triangleq [\boldsymbol{z}_{\mathsf{H},1}, \dots, \boldsymbol{z}_{\mathsf{H},B}]^{\mathsf{T}}$, $\boldsymbol{U}_{\mathsf{H}} \triangleq [\boldsymbol{u}_{\mathsf{H},1}, \dots, \boldsymbol{u}_{\mathsf{H},B}]^{\mathsf{T}}$,
  $\boldsymbol{Y} \triangleq [\boldsymbol{y}_1, \dots, \boldsymbol{y}_B]^{\mathsf{T}}$, $\boldsymbol{Z}_{\mathsf{O}} \triangleq [\boldsymbol{z}_{\mathsf{O},1}, \dots, \boldsymbol{z}_{\mathsf{O},B}]^{\mathsf{T}}$
- grad wrt variables : $\left[\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right]_{ik} = \frac{\partial J(\boldsymbol{z}_{\mathsf{O},i})}{\partial z_{\mathsf{O},ik}}$, $\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{U}_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\boldsymbol{W}_{\mathsf{O}}$, $\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{U}_{\mathsf{H}}} \odot g'_{\mathsf{H}}(\boldsymbol{Z}_{\mathsf{H}})$
- grad wrt params : $\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}_{\mathsf{O}}} = \mathbf{1}_B^{\mathsf{T}}\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}} \in \mathbb{R}^{d_{\mathsf{O}}}$, $\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{W}_{\mathsf{O}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{O}}}\right)^{\mathsf{T}}\boldsymbol{U}_{\mathsf{H}} \in \mathbb{R}^{d_{\mathsf{O}} \times d_{\mathsf{H}}}$
  $\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}_{\mathsf{H}}} = \mathbf{1}_B^{\mathsf{T}}\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}} \in \mathbb{R}^{d_{\mathsf{H}}}$, $\quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{W}_{\mathsf{H}}} = \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{Z}_{\mathsf{H}}}\right)^{\mathsf{T}}\boldsymbol{X} \in \mathbb{R}^{d_{\mathsf{H}} \times d}$
- 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:

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

- then the backward pass:

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

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

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

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

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

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