SPRING 2023

CS 378: INTRO TO SPEECH AND AUDIO PROCESSING

Neural Network Acoustic Models 1

DAVID HARWATH
Assistant Professor, UTCS

The University of Texas at Austin
Department of Computer Science
College of Natural Sciences

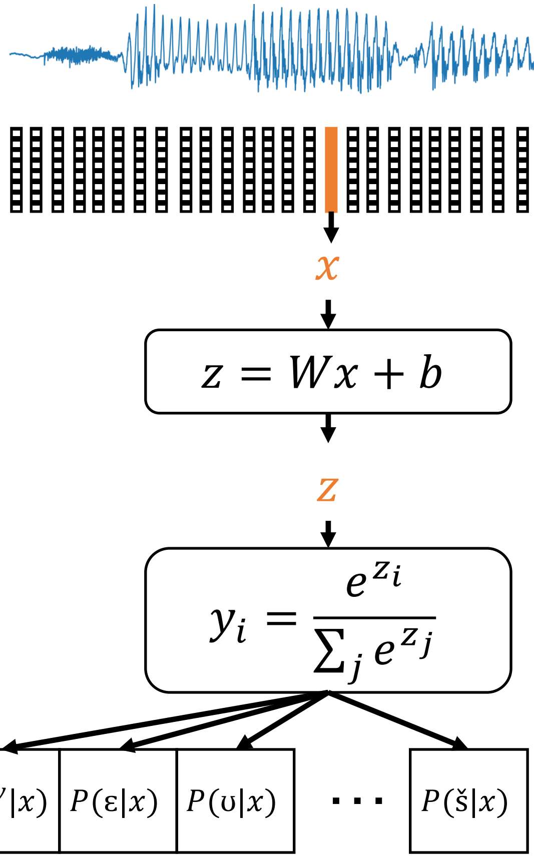
Welcome!

- Today:
  - Intro to neural nets
  - *Non-E2E* neural net *acoustic models*
    - Hybrid/Tandem
    - Architectures

Outline

- Neural Net Overview
  - Definition and Examples
  - Neuron/Layer view
  - Optimization via SGD and Backpropagation
- Architectures used for ASR hybrid models
  - Feedforward
  - Recurrent variants, BPTT
  - CNN

A Logistic Regression Acoustic Model



Speech waveform

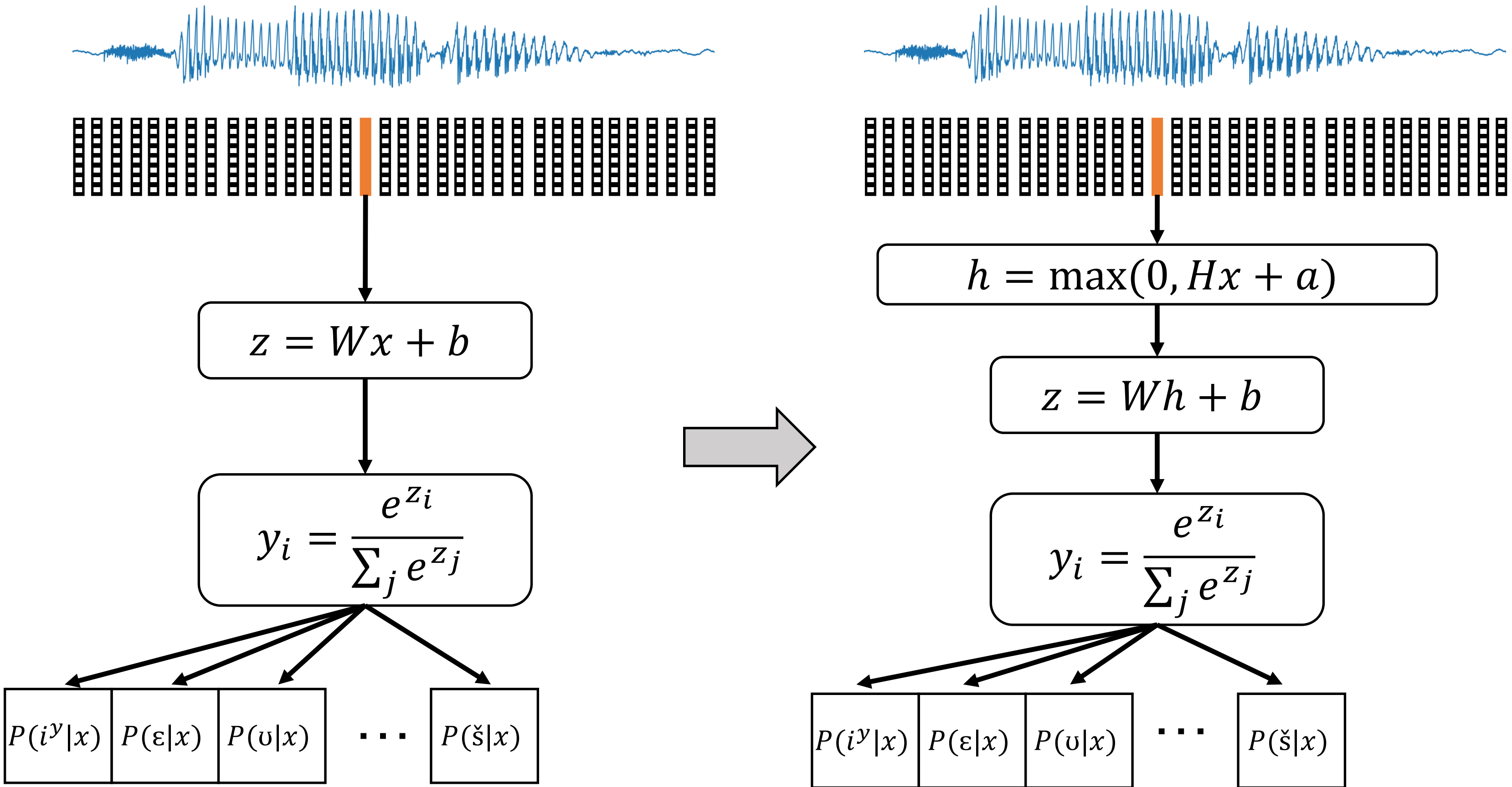
Acoustic features such as MFCCs
Let $x \in \mathbb{R}^D$

Affine transform to compute phonetic state scores
Let $z \in \mathbb{R}^C$ (assuming we have C different phonetic states)

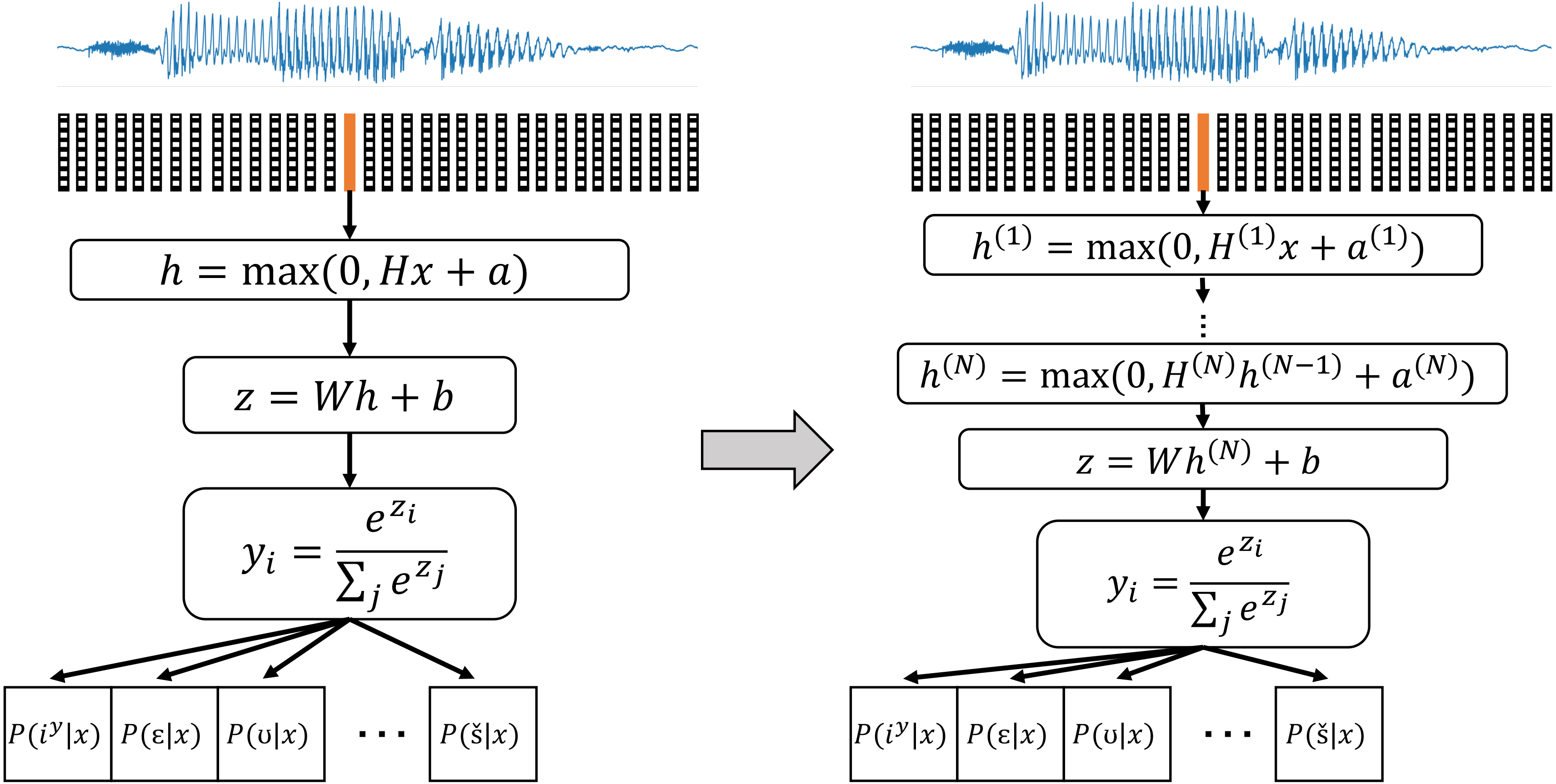
Softmax function to normalize score distribution
(i.e. all scores positive and sum to 1)

Vector y representing phonetic state probabilities

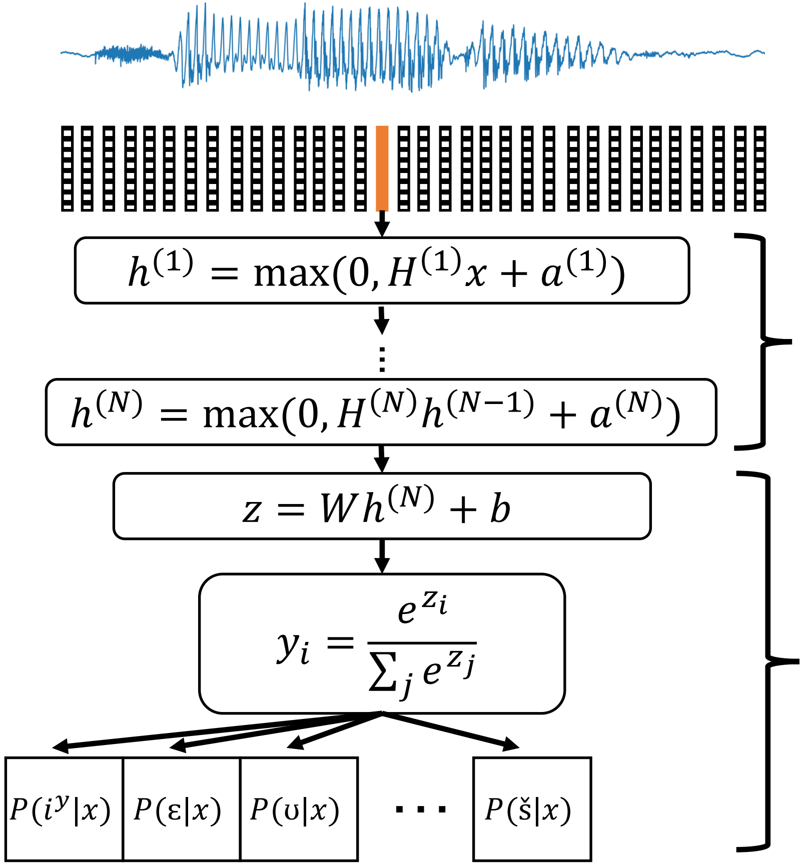
From Logistic Regression to a Neural Net



From a Neural Net to a Deep Neural Net



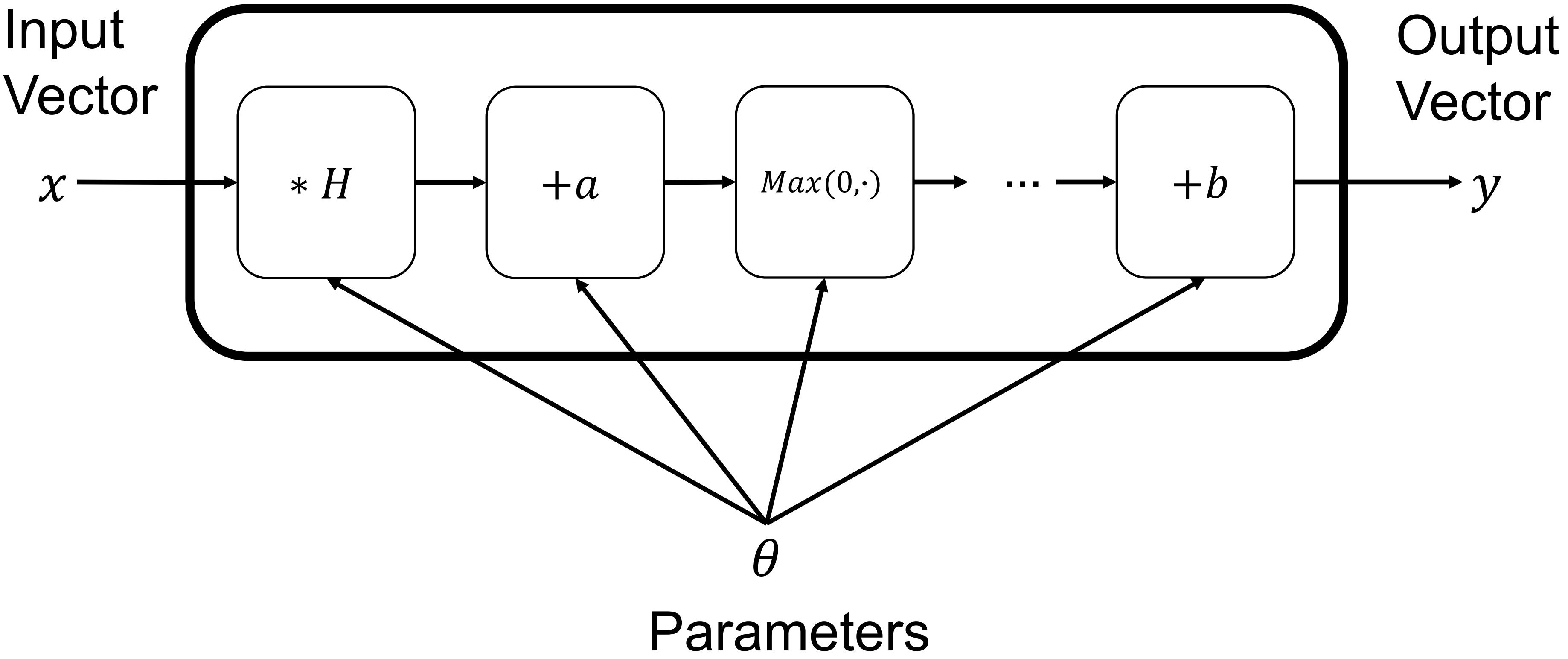
A View of Neural Net Classifiers



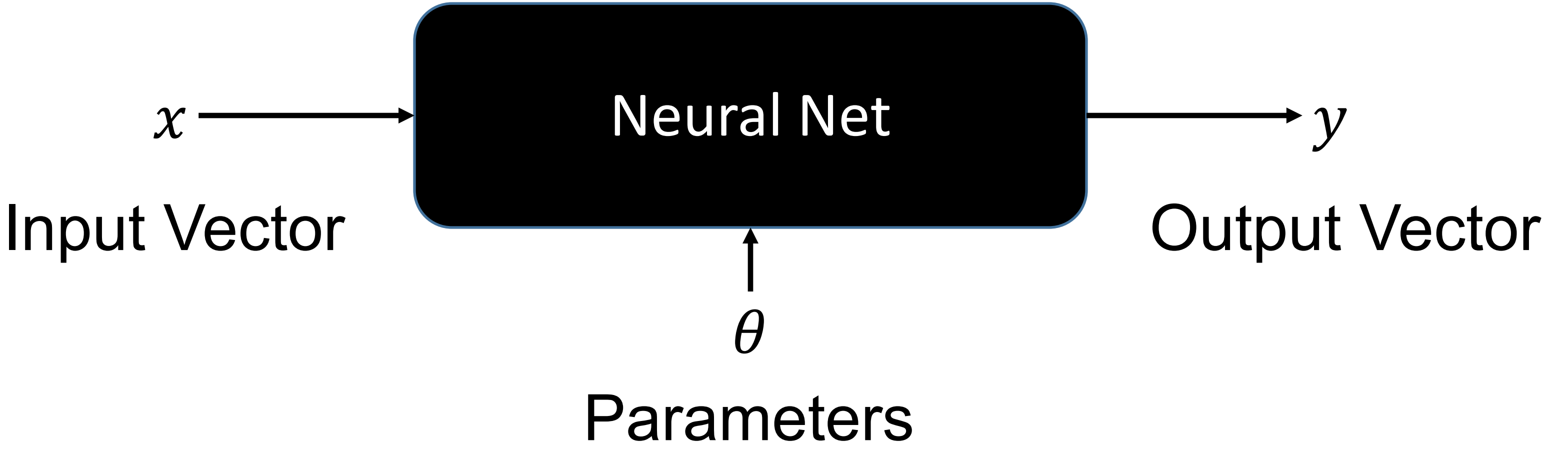
Nonlinear feature extractor with trainable parameters ($H^{(k)}$and $a^{(k)}$)

(Multiclass) Logistic Regression classifier that operates in $h^{(N)}$ space

Neural Nets as Sequences of Transformations

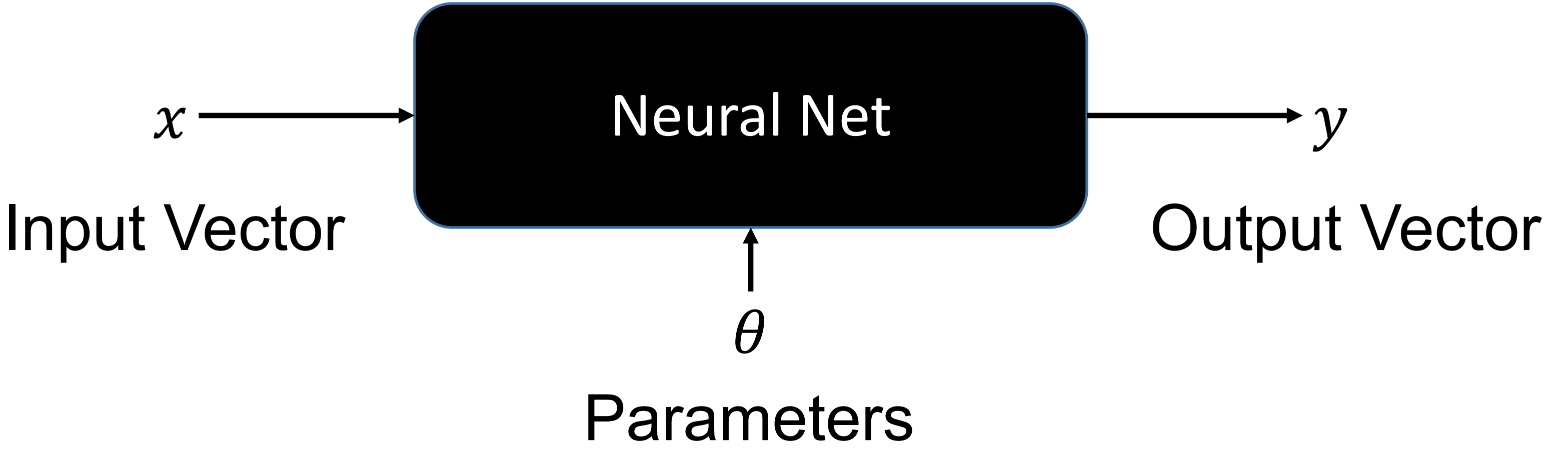


Neural Nets as Function Approximators



We want to learn a function $y' = f(x, \theta)$ and all we have is a finite set of (x, y') pairs

Neural Nets as Function Approximators

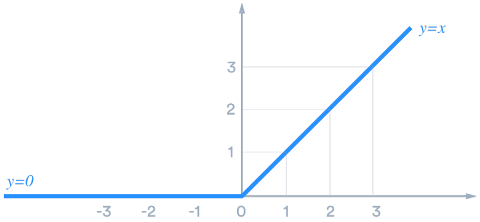


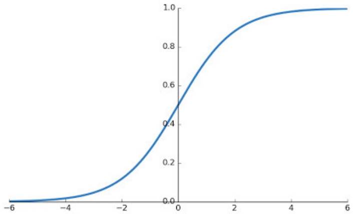
Loss function $L(y, y')$: Measures how *different* (in some well-defined way) the output we got (y) is from the output we want (y')

As long as we can compute (or even estimate) $\nabla_\theta L(y, y')$, we can train the model with gradient descent (the most popular, but not the only way to train).

Nonlinear Layers

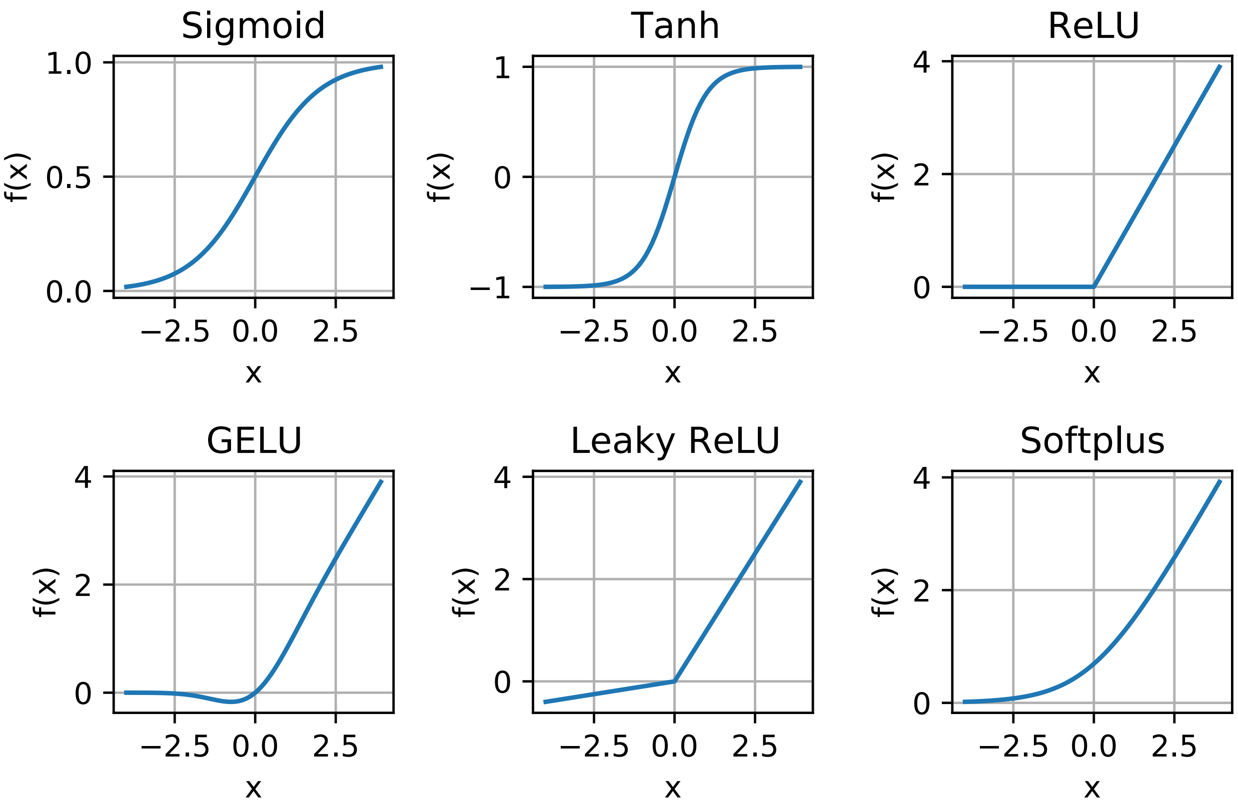
- Much of the power of neural nets comes from their ability to learn complex, nonlinear input-output relations
- The canonical way to introduce nonlinearity into the model is with an “activation function” like ReLU or sigmoid that we insert between linear transformations (basically just some form of clipping)





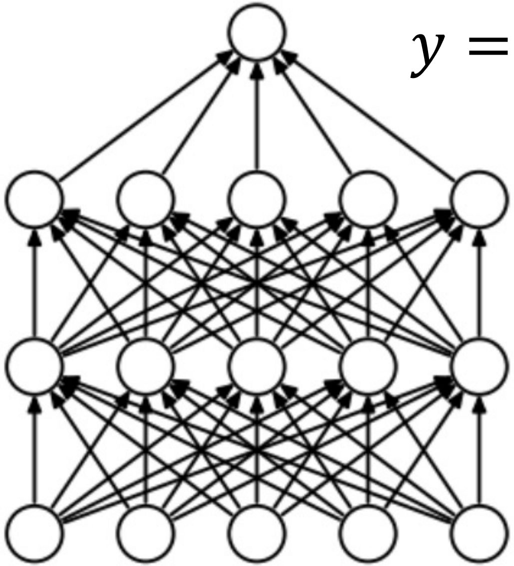
$$x_0 \rightarrow \boxed{A_1x_0 + b_1} \xrightarrow{x_1} \boxed{\max(0, x_1)} \xrightarrow{x_2} \boxed{A_2x_2 + b_2} \xrightarrow{x_3} \boxed{\frac{1}{1 + e^{-x_3}}} \rightarrow y$$

Common Activation Functions



Why “Neural” Networks?

Inspired by biological neurons – taking a weighted sum of inputs followed by saturating nonlinearity resembles “integrate and fire” response of biological neurons



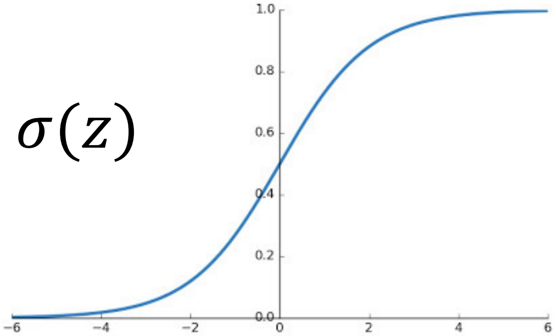
$$y = v^t\sigma\big(C\sigma(Ax + b)\big) + d$$

$$\sigma\big(C\sigma(Ax + b)\big) + d$$

$$\sigma(Ax + b)$$

$$x$$

$\sigma(z)$



z

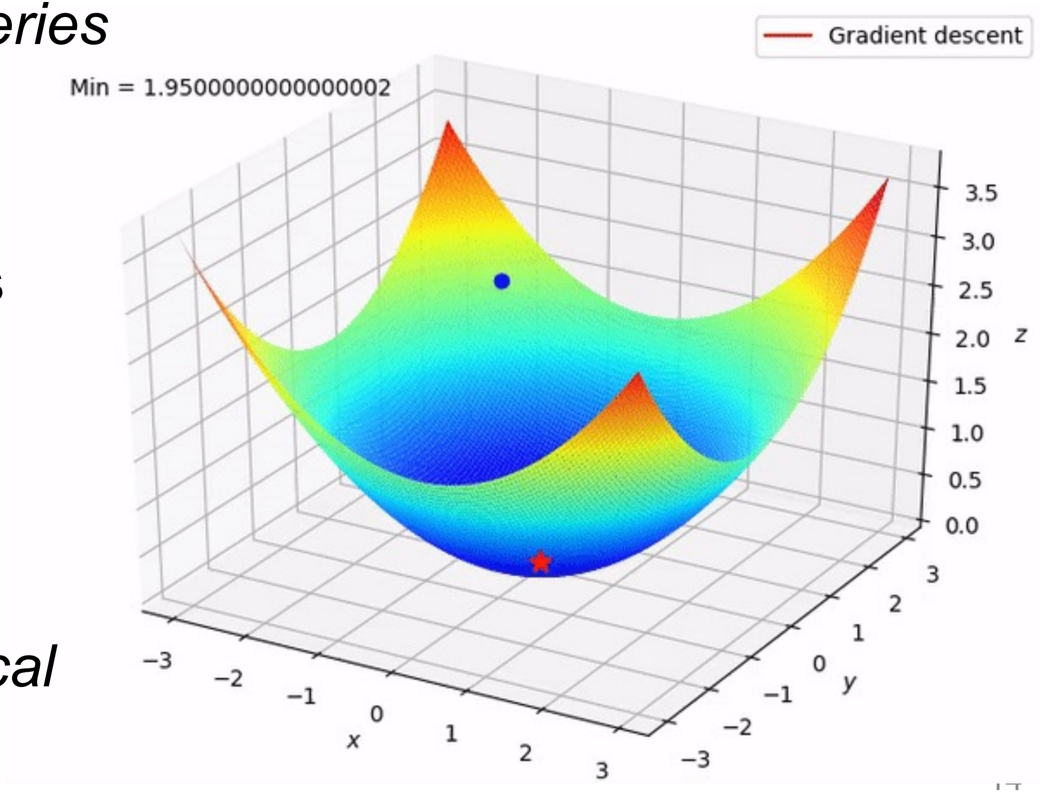
Training via Gradient Descent

Gradient descent computes a *series* of parameter vectors:

$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta L(y, y')$$

We stop when some criteria has been met (convergence, pre-defined number of steps, model performance on held-out validation set, etc.)

Guaranteed to converge to a *local* minimum of the loss function.



Gradient Descent Flavors

Working with a *set* of training examples (assumed i.i.d.):

Using all examples at once:

$$\theta_{t+1} = \theta_t - \gamma \frac{1}{N} \sum_{i=0}^{N-1} \nabla_\theta L(y_i, y_i')$$

Using a *random subset (minibatch)* $\boldsymbol{M_t}$ of examples at each step:

$$\theta_{t+1} = \theta_t - \gamma \frac{1}{B} \sum_{(x,y,y') \in \boldsymbol{M}_t} \nabla_\theta L(y, y')$$

Typically called *Stochastic Gradient Descent (SGD)*

Typical value of γ: usually 0.01 to 0.0001, **but highly problem specific!**

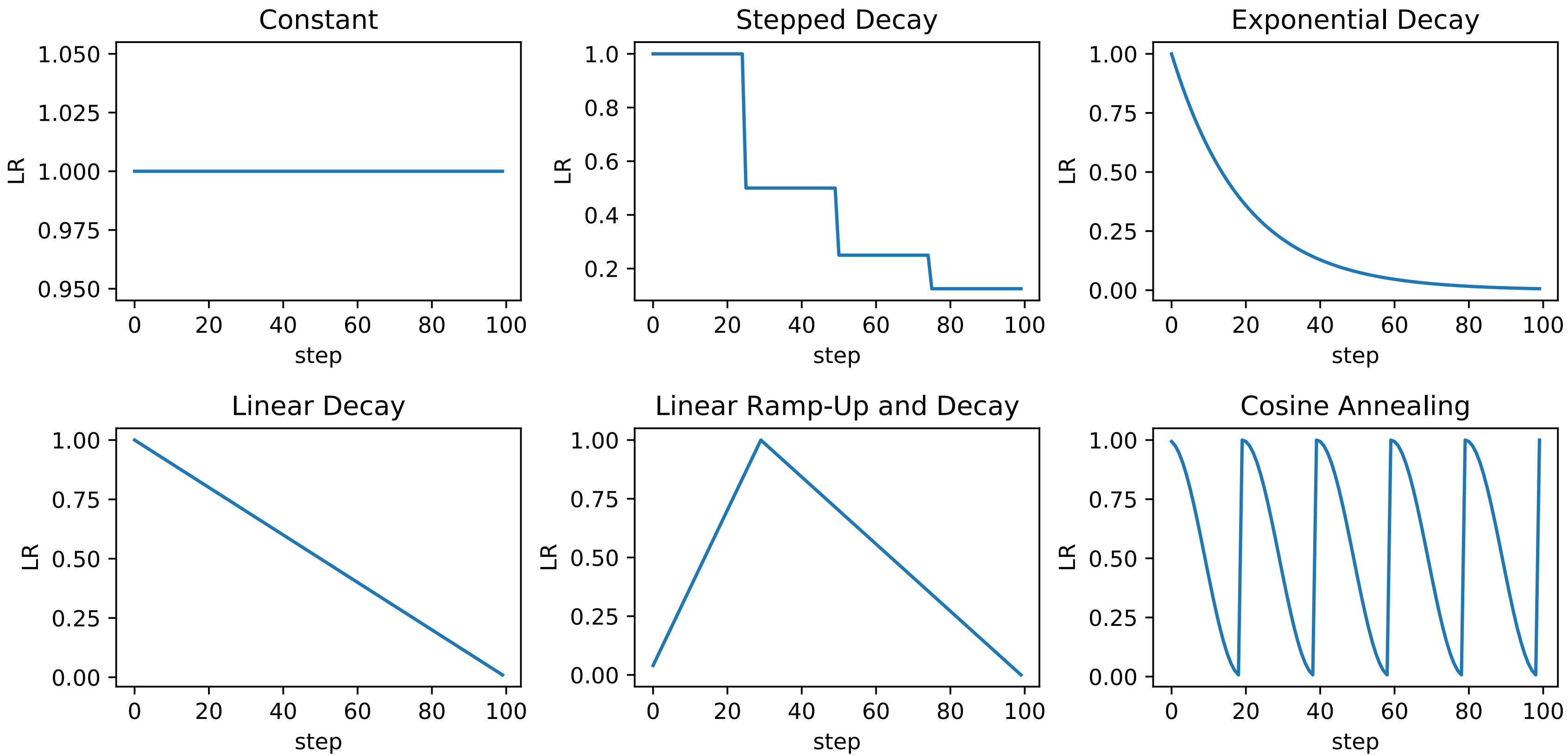
SGD with Momentum

- Problem with SGD: high variance of the gradient
- One solution: keep a memory or “momentum” α (typical value: 0.9) of past updates and blend it with the current gradient

$$v_t = \alpha v_{t-1} + \gamma \frac{1}{B} \sum_{(x,y,y') \in \boldsymbol{M_t}} \nabla_\theta L(y, y')$$

$$\theta_{t+1} = \theta_t - v_t$$

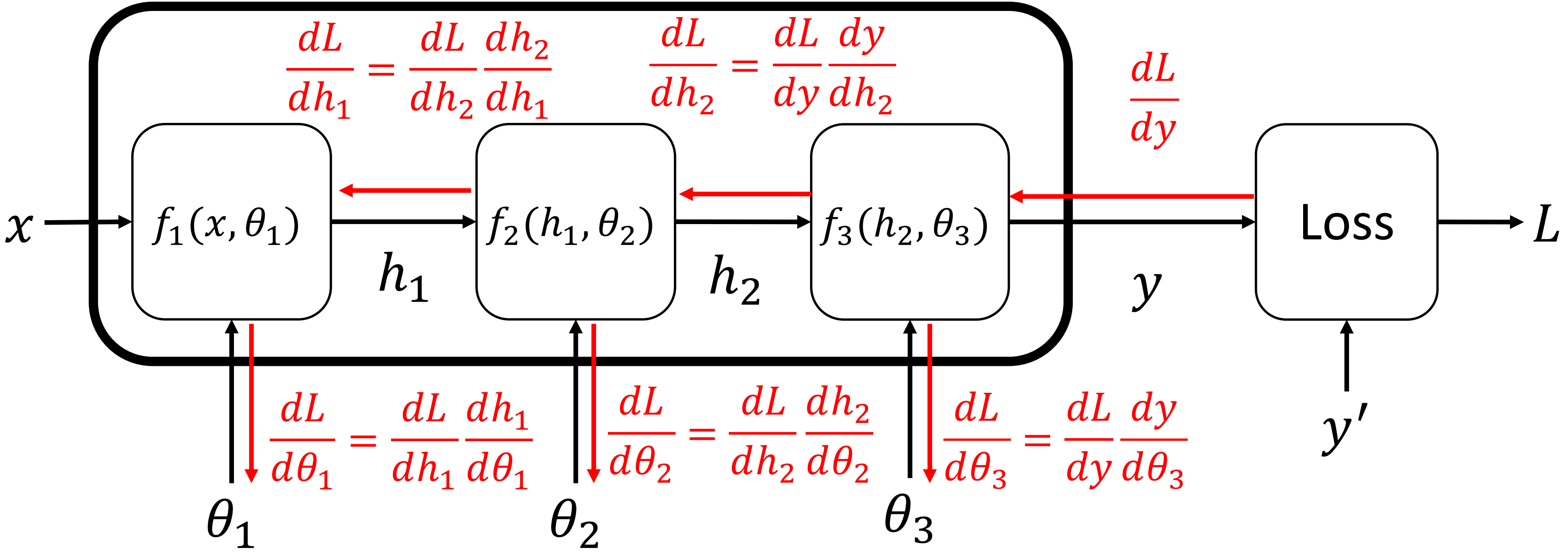
Some Learning Rate Schedules



Adaptive SGD variants

- Try to use individual learning rates for each parameter
- Many variants: Adagrad, Adadelta, RMSprop, Adam…
- Adam arguably the most popular optimizer right now

Computing the Gradient via Backpropagation



Backprop takeaway

If:

1. You can express your neural network model as a sequence of atomic operations (layers)
2. Each atomic operation you use is locally differentiable (output w.r.t. input/parameters)

Then: You can use backpropagation to efficiently compute the gradient for every network parameter, *without having to analytically derive anything besides the local gradient of each layer.*

Neural Net Toolkits

- Any modern neural network software toolkit (e.g. PyTorch, Tensorflow) ships with a large library of layer classes that already implement forward and local backward passes (as well as optimization algorithms)

- If you want to create a new layer type, you just need to define the class and implement the forward and backward (local gradient) computations.

PyTorch code example of NN training

```python
import torch  # Basic torch library, includes stuff like Tensors
import torch.nn  # Torch's neural network library
import torch.optim as optim  # Torch's optimization library

data = np.load('dataset.npz') # Load the data
# Cast numpy arrays to torch tensors
# Assume we have 10 classes and our feature dimension is 16
train_feats = torch.tensor(data['train_feats']) # tensor will be size (N_examples, N_features)
train_labels = torch.tensor(data['train_labels']) # tensor will be size (N_examples, N_classes)
# Wrap the features and labels together into a dataset object
train_dataset = torch.utils.data.TensorDataset(train_feats, train_labels)
# Create a dataloader (iterator that samples minibatches from the full dataset)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=8, shuffle=True)

class MyNetwork(nn.Module):
    def __init__(self):
        super(MyNetwork, self).__init__()
        self.linear1 = nn.Linear(16, 64)  # Hidden layer, size (N_features, hidden_dimension)
        self.relu = nn.ReLU() # Rectified Linear Unit nonlinearity
        self.linear2 = nn.Linear(64, 10)  # Output layer, size (hidden_dimension, N_classes)

    def forward(self, x):
        x = self.relu(self.linear1(x))  # Compute the hidden unit activations
        x = self.linear2(x)  # Compute the logits over the classes
        return x

model = MyNetwork()  # Instantiate the model
criterion = nn.CrossEntropyLoss()  # The loss function: will apply both softmax and cross-entropy
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)  # The optimizer

for epoch in range(10):  # loop over the dataset 10 times ("epochs")
    for i, (inputs, labels) in enumerate(train_loader, 0):
        optimizer.zero_grad()  # Reset the optimizer's gradient accumulators
        outputs = model(inputs)  # Compute the network outputs (forward pass)
        loss = criterion(outputs, labels)  # Compute the loss
        loss.backward()  # Backpropagate the loss into the network
        optimizer.step()  # Take a single step of gradient descent
```

Some Common Layers and Their Gradients

Assume: $X \in \mathbb{R}^{N \times D}$ (batch size x feature dimension), and $\frac{dL}{dY}$ is the "upstream" gradient of the loss w.r.t. output Y (will have the same dimension as Y)

Layer Name	Forward Pass	Input Gradient	Parameter Gradient
Linear	$Y = WX$	$\frac{dL}{dX} = W^T \frac{dL}{dY}$	$\frac{dL}{dW} = \frac{dL}{dY} X^T$
Sigmoid	$y_{nd} = \frac{1}{1 + e^{-x_{nd}}}$	$\frac{dL}{dx_{nd}} = \frac{dL}{dy_{nd}} y_{nd}(1 - y_{nd})$	N/A
Rectified Linear Unit (ReLU)	$Y = \max(0, X)$	$\frac{dL}{dx_{nd}} = \begin{cases} \frac{dL}{dy_{nd}} & \text{if } x_{nd} > 0 \\ 0 & \text{else} \end{cases}$	N/A
Softmax	$y_{nd} = \frac{e^{x_{nd}}}{\sum_{d'} e^{x_{nd'}}}$	$\frac{dL_{nj}}{dx_{nd}} = \begin{cases} \frac{dL}{dy_{nd}} y_{nd}(1 - y_{nj}) & \text{if } j = d \\ -\frac{dL}{dy_{nd}} y_{nd} y_{nj} & \text{else} \end{cases}$	N/A

Some Common Loss Functions

- Regression: L2 loss (also called Mean Squared Error or MSE):

$$L(y, y') = \|y - y'\|_2^2$$

$\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\end{bmatrix}$

- Classification: Cross-entropy
  - Assume y' represents the *true* probability distribution over class labels $p(c)$ (usually just a 1-hot vector)
  - Assume y represents our *estimated* distribution over the labels $q(c)$

$$L(q(c) = y, p(c) = y') = -\sum_{c=1}^{N_c} p(c) \log q(c)$$

Cross Entropy in Practice

- Usually the final parameterized layer in a classifier network is a linear layer with N_c neurons
  - Output of this layer z sometimes called “class scores” or “logits”
- We normalize these scores with a softmax layer:

$$y = \frac{e^z}{\sum_{z'} e^{z'}}$$

- We can then write the cross entropy loss as:

$$L(y, y') = -\log(y)^T y'$$