

# EECS 445 Discussion 7: Neural Nets

EECS 445 Staff

March 11, 2021

Far away in the heavenly abode of the great god Indra, there is a wonderful net. . . infinite in dimension. . .

---

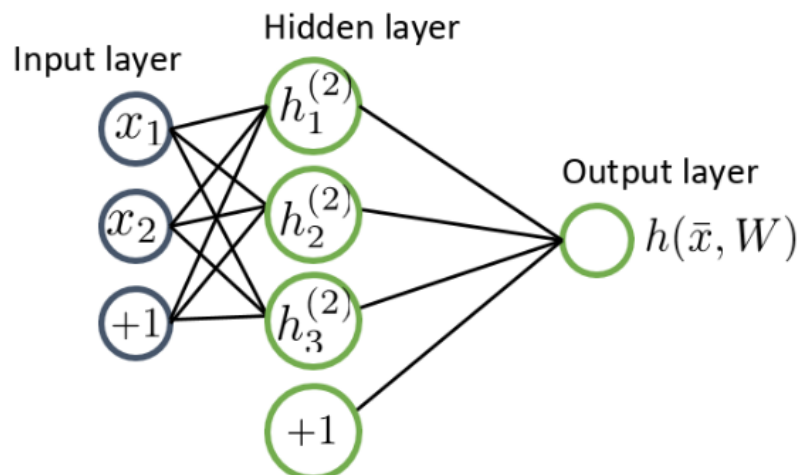
The Avatakamsaka Sutra

## 1 Neural Networks Review

So far in lecture, we've seen how to learn using Perceptrons, SVMs, Decision Trees, AdaBoost, and many variants. A key commonality is that each algorithm returns a function  $f: X \mapsto Y$  (usually parametrized by some  $\theta$  in the same space as  $x$ ) that generalizes a training set  $\{(x^{(i)}, y^{(i)})\}_{i=1}^n$  of input-output pairs to future examples. This learning task is called **supervised learning**.

Neural nets are also a tool for supervised learning. At their core, feed-forward linear neural networks (the kind we've learned about so far) are a series of linear combinations of features and weights, transformed by an **activation function** (Section 2). These activation functions change neural networks from linear regression to a more complicated architecture. We use these non-linear functions to capture non-linear relationships between features.

Let's walk through an example of a neural network with one hidden layer. Consider this example, using **sigmoid activation**:

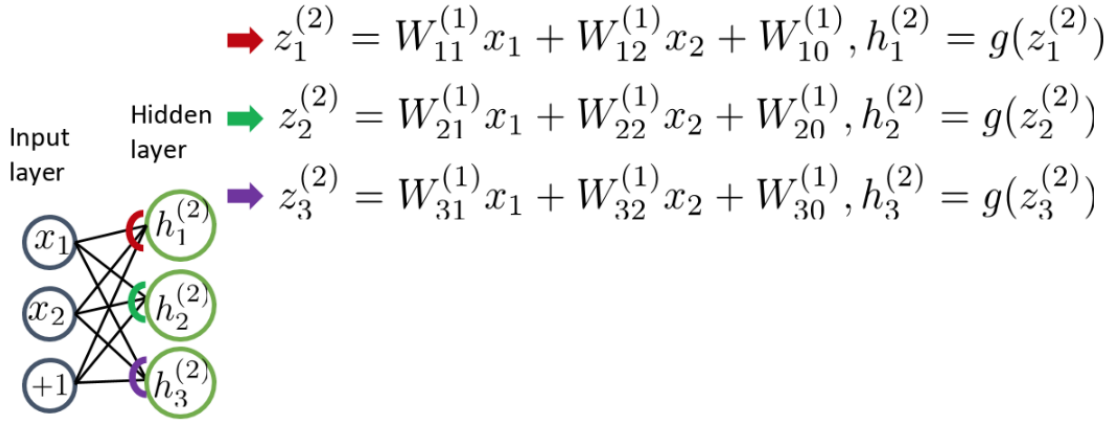


**Discussion Question 1.** What is the formula to calculate  $z_1^{(2)}$ , the input to  $h_1^{(2)}$ ? How about the output  $h_1^{(2)}$ ?

Input:  $z_1^{(2)} = W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{10}^{(1)}$

Output:  $h_1^{(2)} = \sigma(z_1^{(2)})$  where  $\sigma(x) = \frac{1}{1+e^{-x}}$  is the sigmoid activation function.

The final output from the hidden layers will be as follows:



**Discussion Question 2.** What is the formula to calculate the input to the output layer? How about the output  $h(\bar{x}, W)$  (using sigmoid activation as well)?

Input:  $z^{(3)} = W_{11}^{(2)}z_1^{(2)} + W_{12}^{(2)}z_2^{(2)} + W_{13}^{(2)}z_3^{(2)} + W_{10}^{(2)}$

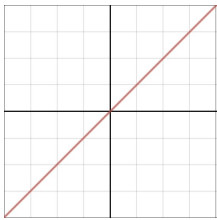
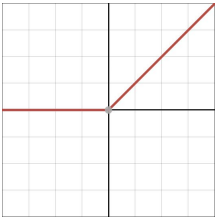
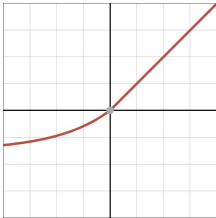
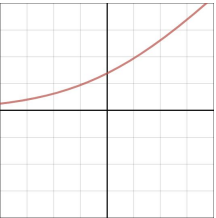
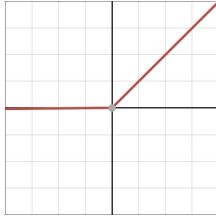
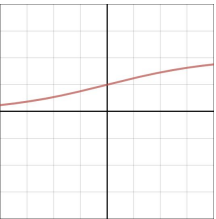
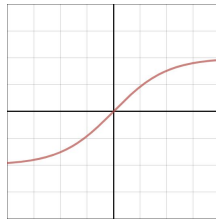
Output:  $h(\bar{x}, W) = \sigma(z^{(3)})$

**Discussion Question 3.** Given that  $h_1^{(2)}$  is a non-linear combination of the features of  $\bar{x}$ , what does  $h_1^{(2)}$  correspond to in a linear regression problem?

$h_1^{(2)}$  essentially transforms the input so that we learn a new feature transformation! We are essentially learning a linear classifier in a new feature space. The new feature space can be defined as  $[h_1^{(2)}, h_2^{(2)}, h_3^{(2)}, 1]$

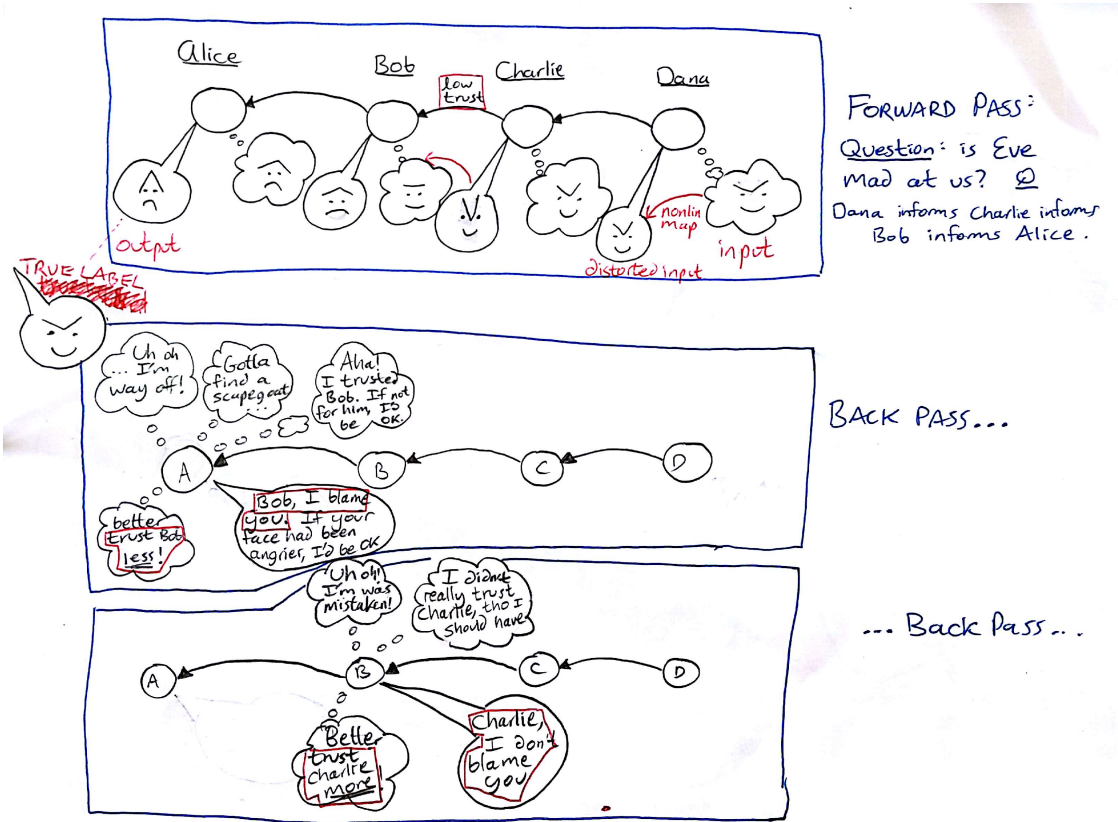
## 2 Architecture

### 2.1 Activation Functions

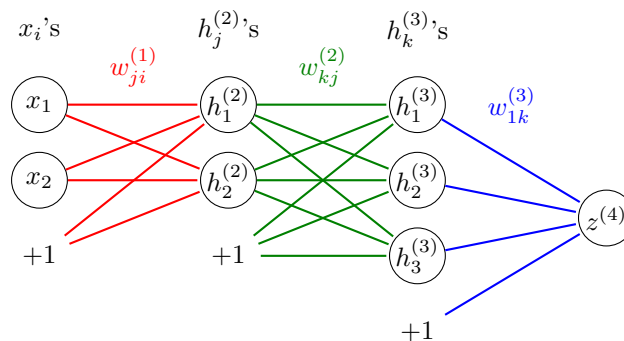
<p><b>Identity</b>  <math>h(z) = z</math></p> 	
<p><b>ReLU</b>  <math>h(z) = \max(0, z)</math></p> 	<p><b>ELU</b>  <math>h(z) = \begin{cases} \alpha(e^z - 1) &amp; \text{for } z &lt; 0 \\ z &amp; \text{for } z \geq 0 \end{cases}</math></p> 
<p><b>SoftPlus</b>  <math>h(z) = \ln(1 + e^z)</math></p> 	<p><b>LReLU</b>  <math>h(z) = \begin{cases} 0.01z &amp; \text{for } z &lt; 0 \\ z &amp; \text{for } z \geq 0 \end{cases}</math></p> 
<p><b>Sigmoid</b>  <math>h(z) = \frac{1}{1 + e^{-z}}</math></p> 	<p><b>Tanh</b>  <math>h(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}</math></p> 
<p><b>Softmax</b>  <math>h_i(\bar{z}) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}</math></p>	<p><b>OneHot</b>  <math>h_i(\bar{z}) = \begin{cases} 1, &amp; \text{if } i = \arg \max_j z_j \\ 0, &amp; \text{otherwise} \end{cases}</math></p>

Here is a very nice [interactive visualization](#) of Activation Functions.

### 3 Understanding Backpropagation



Consider the following two hidden-layer neural network, with sigmoid activations at each hidden neuron and loss function  $L$ , which solves a classification task with  $\bar{x} \in \mathbb{R}^2$  and  $y \in \{-1, +1\}$ :



We can write down the forward and backward propagation of values in following table:

Forward	Backward	
$z_j^{(2)} = \sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)}$	$\frac{\partial z_j^{(2)}}{\partial w_{ji}^{(1)}} = x_i$	$\frac{\partial z_j^{(2)}}{\partial x_i} = w_{ji}^{(1)}$
$h_j^{(2)} = \sigma(z_j^{(2)})$	$\frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = h_j^{(2)}(1 - h_j^{(2)})$	
$z_k^{(3)} = \sum_j w_{kj}^{(2)} h_j^{(2)} + w_{k0}^{(2)}$	$\frac{\partial z_k^{(3)}}{\partial w_{kj}^{(2)}} = h_j^{(2)}$	$\frac{\partial z_k^{(3)}}{\partial h_j^{(2)}} = w_{kj}^{(2)}$
$h_k^{(3)} = \sigma(z_k^{(3)})$	$\frac{\partial h_k^{(3)}}{\partial z_k^{(3)}} = h_k^{(3)}(1 - h_k^{(3)})$	
$z^{(4)} = \sum_k w_{1k}^{(3)} h_k^{(3)} + w_{10}^{(3)}$	$\frac{\partial z^{(4)}}{\partial w_{1k}^{(3)}} = h_k^{(3)}$	$\frac{\partial z^{(4)}}{\partial h_k^{(3)}} = w_{1k}^{(3)}$
$L(y, z^{(4)})$	$\frac{\partial L}{\partial z^{(4)}}$	

With these local partial derivative worked out, we can easily write down the partial derivative of loss with respect to each weight parameter:

- $\frac{\partial L}{\partial w_{11}^{(3)}} = \frac{\partial L}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial w_{11}^{(3)}}$
- $\frac{\partial L}{\partial w_{11}^{(2)}} = \frac{\partial L}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial w_{11}^{(2)}}, \quad \frac{\partial L}{\partial w_{21}^{(2)}} = \frac{\partial L}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial w_{21}^{(2)}}.$

These two partial derivatives differ only in the last term in the product.

- $\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{\partial L}{\partial z^{(4)}} \sum_k \frac{\partial z^{(4)}}{\partial h_k^{(3)}} \frac{\partial h_k^{(3)}}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial h_1^{(2)}} \frac{\partial h_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(1)}}, \quad \frac{\partial L}{\partial w_{21}^{(1)}} = \frac{\partial L}{\partial z^{(4)}} \sum_k \frac{\partial z^{(4)}}{\partial h_k^{(3)}} \frac{\partial h_k^{(3)}}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial h_1^{(2)}} \frac{\partial h_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{21}^{(1)}}$

These two partial derivatives also differ only in the last term in the product.

Backpropagation is just a fancy name for this organizational system for computing the  $\frac{\partial L}{\partial w^{(\cdot)}}$ 's neatly and efficiently, by observing that local partial derivatives are shared. If you remember **dynamic programming** from EECS 281, the trick is exactly that.

## 4 Architecture Specifications

In Project 2, we present a few different architecture specifications that describe deep neural network models. One architecture describes a convolutional neural network, and the other describes an autoencoder. Here, we present a few examples that may help you learn not only how to read these specifications when implementing the Project 2 networks, but also how to describe your own architecture design for the challenge.

### 4.1 CNN Architecture

We now introduce a diagram that describes a convolutional neural network architecture. The convolution layers in a CNN are often clarified by such a diagram, especially for two- or three-dimensional input data.

- in Convolutional layer 2:

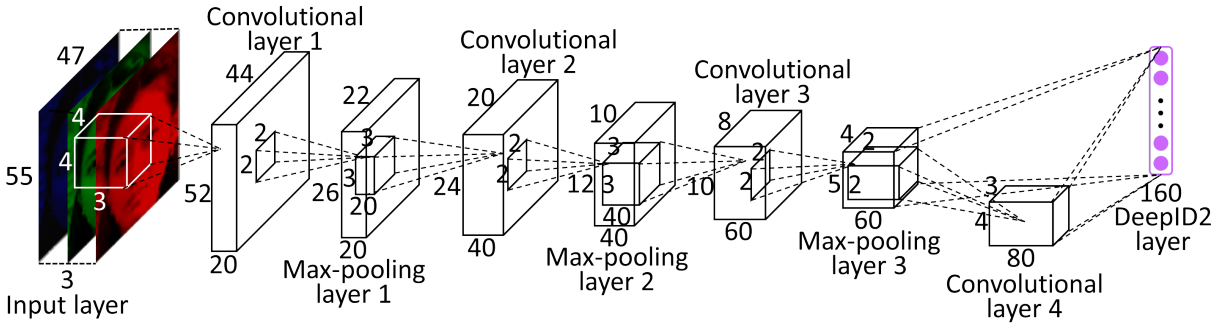


Figure 1: CNN architecture, from the DeepID2 network:

<https://arxiv.org/abs/1406.4773>

- What is size of each filter (aka kernel)?  $(20 \times) 3 \times 3$ .
- How many filters are there? 40
- How many model parameters are associated with the filters? How many bias parameters are in this layer?  $40 \times 20 \times 3 \times 3$  weights, 40 biases
- What are the input and output dimensions? input:  $20 \times 26 \times 22$ , output:  $40 \times 24 \times 20$ .  
Note: We follow the PyTorch's tensor shape notation of CHW: n\_channels  $\times$  height  $\times$  width.
- What is the padding, and what is the stride size? No padding, unit stride.
- What would the output dimension be if we used SAME padding and stride=1?  $40 \times 26 \times 22$
- What would the output dimension be if we used SAME padding and stride=2?  $40 \times 13 \times 11$
- in Max-pooling layer 2:
  - What is the filter size?  $2 \times 2$ . Each pooling filter is applied to every input channel independently.
  - What are the input and output dimensions? input:  $40 \times 24 \times 20$ , output:  $40 \times 12 \times 10$ .
  - What is the stride size? 2

## 5 Project Architectures

Here, we go over some justifications for the decisions that were made in designing the architectures you will be implementing in Project 2.

- Preprocessing is used to normalize the images into a more standard format that can better be represented by a single set of weights in the neural network architecture. If this was not done, images would have different norms (e.g. bright vs. dark images).
- We provided you with a class-balanced dataset so that the loss function equally represents the loss across each of the classes. Otherwise, in an imbalanced class scenario, loss can be naively minimized by choosing the most likely class label. In that case, it is common to assign training instances different importances (weights).
- The optimizer that we recommend starting with is the `torch.optim.Adam`, first described in the paper by Kingma and Ba: <https://arxiv.org/abs/1412.6980>.

This optimizer uses a stochastic approach with a changing learning rate to efficiently optimize the model parameters to minimize loss. However, for simpler models other approaches such as stochastic gradient descent `torch.optim.SGD`, which we have seen in various instances in lecture, can also be used.

- We typically initialize weights to a Gaussian distribution with mean 0.0 and variance  $\frac{1}{n}$  where  $n =$  [number of input nodes].

This normalizes the variance of the layer's output to a standard Gaussian distribution rather than a Gaussian distribution with variance that grows with the number of inputs.

- We use the ReLU activation function, which is popular as it mitigates the vanishing gradient problem and is fast to compute.

Why is the ReLU function fast to compute?

## 5.1 Convolutional Neural Network

- The convolutional layers are used to add position-invariance through parameter sharing, as input dog images may be in different perspectives and of different sizes. The series of convolutional layers yield smaller and smaller outputs with a greater number of channels, which condenses the image into a series of small images that potentially represent different features.
- The fully-connected and output layers map the convolutional layer output into label predictions by allowing for interactions between the convolutional filters.

## 6 PyTorch

A variety of frameworks have come out in recent years for learning deep neural nets, including: Caffe, CNTK, Theano, and TensorFlow. There are also high-level frameworks that rely on the backend of the above frameworks to facilitate implementation of common networks. The most notable example at present is Keras, which provides sane defaults for quick experimentation.

We will introduce the fundamentals of **PyTorch**, a library open-sourced by Facebook which has since been adopted by many researchers in the field of machine learning. This framework is used for **Project 2**.

This tutorial is adapted from tutorials in the PyTorch library (linked in section 6.4).

### 6.1 PyTorch Tensors

PyTorch is built upon the concept of Tensors - which are essentially vectors, but they're optimized for performance. Everything in PyTorch is a Tensor, including your data and your neural network layers. To implement a convolutional neural network, we'll be using tensor functionality described below.

### 6.2 Defining a Neural Net

Using PyTorch, let's start by creating a feed forward neural network that corresponds to linear regression. This model will perform linear regression on a dataset with values  $x^{(i)} \in \mathbb{R}^3$  and labels  $y^{(i)} \in \mathbb{R}$  (scalar inputs and labels). The code for the **training process** has been outlined below.

1. Dataset: `x_train, y_train, x_test, y_test`
2. Define model parameters:
  - Learning rate: 0.01
  - Number of training epochs: 1000 (loops over training data)
3. Start with creating a neural network class to house your neural network.

---

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
```

```

def __init__(self):
    super(Net, self).__init__()
    # Put Neural Net Structure Here

def forward(self, x):
    # Define your forward pass here

```

---

4. You'll want to define your network architecture with a single linear layer.

```

# Creates a fully connected layer. Useful for the end
# of your net
# input_size will be the size of your feature vector
# output_size will be the size of your output
fc = nn.Linear(input_size, output_size)

```

---

5. After you've defined your network architecture, you'll want to define your forward propagation. This will require walking through each step in your network, and applying an activation function to the linear layers

```

def forward(self, x):
    # Pass your input through your linear layer,
    # making sure to apply sigmoid activation
    x1 = F.sigmoid(self.fc(x))

    # Return your output
    return x1

```

---

6. Define your loss and your optimizer. Here we're using cross entropy loss and SGD with our learning rate of .01.

```

import torch.optim as optim

net = Net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

```

---

7. Finally, you'll want to train your network. you'll use PyTorch's super convenient gradient calculator to calculate the gradient of your network for you.

```

for epoch in range(1000): # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

```



```

# forward + backward + optimize
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

# print statistics
running_loss += loss.item()
if i % 2000 == 1999:    # print every 2000 mini-batches
    print('[%d, %5d] loss: %.3f' %
          (epoch + 1, i + 1, running_loss / 2000))
    running_loss = 0.0

print('Finished Training')

```

---

You can find the complete tutorial here, which includes helpful tips for the project: [https://pytorch.org/tutorials/beginner/blitz/cifar10\\_tutorial.html](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)

### 6.3 Fixed epochs vs Early Stopping

Given the size of our training data, we're unlikely to have learned a good model after a single epoch. Thus, we will have to loop over the data multiple times i.e., train for multiple epochs. But how will we know when to stop? We will use 'early stopping' and leverage the performance on the validation data. More specifically, in project 2, we will stop training after the validation loss does not decrease for some number of epochs which we define with the variable patience.

What's

- What's the benefit of early stopping?  
More efficient; avoid overfitting.

### 6.4 Other Helpful Tutorials

PyTorch Neural Networks:  
[https://pytorch.org/tutorials/beginner/blitz/neural\\_networks\\_tutorial.html](https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html)

Training a Classifier (used above):  
[https://pytorch.org/tutorials/beginner/blitz/cifar10\\_tutorial.html](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)

Transfer Learning (for the challenge):  
[https://pytorch.org/tutorials/beginner/transfer\\_learning\\_tutorial.html](https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html)