Lecture 8

Deep Learning

Motivation for Deep Learning

- Deep learning (the process of training multi-layered neural networks) provide a scalable framework to train models with very many parameters

- Special network architectures have been developed to deal with non-structured data (we won't discuss these in this course)
    - Transformers – natural language
    - Convolution – computer vision & time series

- Neural networks can perform (almost) as well as boosting trees on tabular data

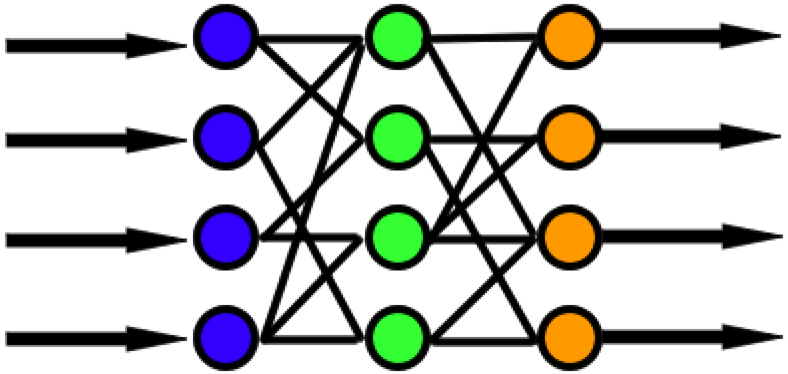
Deep Learning

- Model Architecture
- Training
  - Data loading & why it's necessary
  - Forward pass
  - Loss function calculation
  - Backward pass
  - Optimization
- Inference

Neural Network Architecture

- A neural network is made up of a multiple **layers**, each layer containing numerous **neurons**

- We will consider (for now) only **feedforward neural networks**, in which each layer passes information to only the layer immediately following it



Feedforward Neural Networks

- Each individual neuron has the format

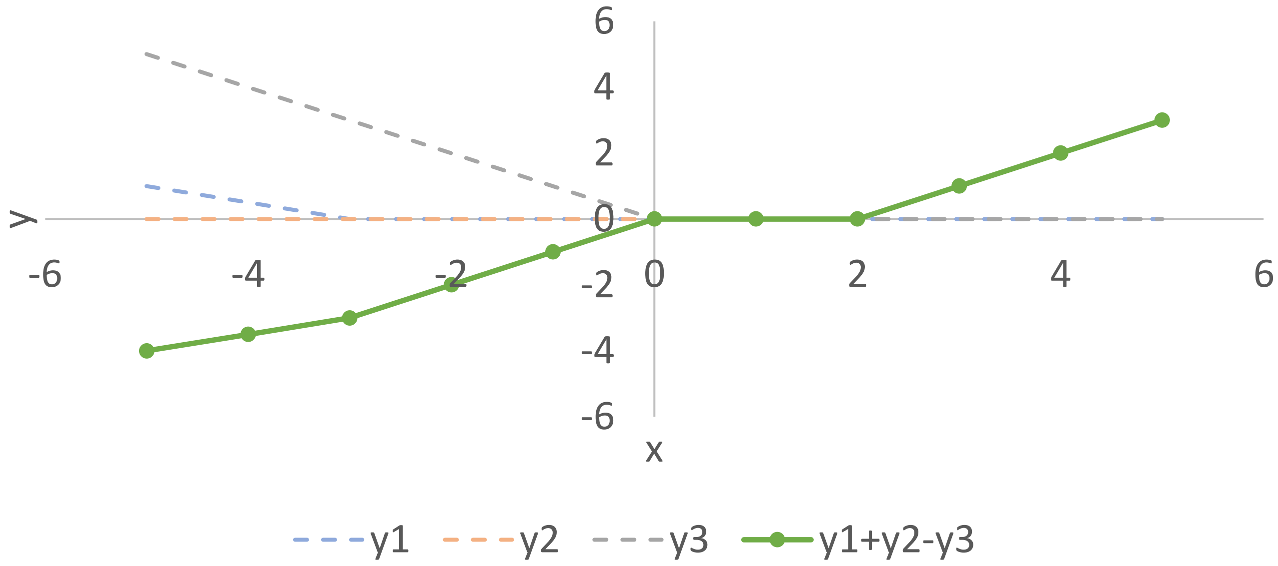
$$y = xA^T + b$$

- Neurons are combined with **non-linear activation functions** to provide non-linear predictive capability

$$ReLU(x) = \begin{cases} x\,; x > 0 \\ 0\,; otherwise \end{cases}$$

Feedforward Neural Networks

- Combining multiple neurons lets the network build complex functions



- With enough of these neurons, we can approximate **any** function

Defining Network Architecture

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

class FFN(nn.module):
    def __init__(self, in_features, out_features, hidden_size):
        self.layer1 = nn.Linear(in_features, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, out_features)

    def forward(self, x):
        y = self.layer1(x)
        y = nn.ReLU()(x) # Normal format
        y = self.layer2(x)
        y = F.relu(x) # Functional format
        y = self.layer3(x)
        return y
```

Understanding Linear Layers

- A linear layer is essentially a large matrix multiplication. Each layer has two learnable matrices of parameters

$$A = \mathbb{R}^{d_{in}, d_{out}}$$
$$b = \mathbb{R}^{d_{out}}$$

- It acts on a vector of inputs

$$x = \mathbb{R}^{n_{obs}, d_{in}}$$

- And produces an output

$$y = \mathbb{R}^{n_{obs}, d_{out}}$$

Understanding Linear Layers

```
net = FFN(3,1,2)
print(net.layer1.weight)
>>> tensor([[ 0.5463, -0.0116, -0.3780],
            [-0.3419, -0.0464, -0.2615]], requires_grad=True)

print(net.layer1.bias)
>>> tensor([0.5359, 0.1306], requires_grad=True)

x = torch.ones([10,3])
print(torch.matmul(x, net.layer1.weight.T) + net.layer1.bias)
>>> tensor([[ 0.6926, -0.5192],
            [ 0.6926, -0.5192],
            [ 0.6926, -0.5192],
            [ 0.6926, -0.5192],
            [ 0.6926, -0.5192]], grad_fn=<AddBackward0>)
```

Understanding Linear Layers

```
print(F.relu(torch.matmul(x, net.layer1.weight.T) + net.layer1.bias))
>>> tensor([[0.6926, 0.0000],
            [0.6926, 0.0000],
            [0.6926, 0.0000],
            [0.6926, 0.0000],
            [0.6926, 0.0000]], grad_fn=<ReluBackward0>)
```

Training – Data Loading

- Neural network training often uses customized **data loaders** that provide **batched** data for each iteration

- Data loaders provide a lot of functionality
  - Incrementally loading data when the full dataset doesn't fit into memory
  - Shuffling data records to help escape local optima
  - Transforming data as it's loaded to increase model robustness

- We won't talk much about data loaders in this class

Forward Pass

- The forward pass is how models make predictions

```
net = FFN(in_features = 3, out_features = 1, hidden_size = 2)

X = torch.rand([50,3])
y = net(X)

print(y.shape)
>>> torch.shape([50, 1])

print(y)
>>> tensor([[-0.2216], [-0.2205], [-0.2184], …,
            [-0.2185]], grad_fn=<AddmmBackward0>)
```

Loss Functions

- A loss function calculates how far the predictions of the model are from the desired values.
  - You can be very creative with your loss functions!

MSE Loss

$$l_n = (x_n - y_n)^2$$

L1 Loss

$$l_n = |x_n - y_n|$$

BCE Loss

$$l_n = y_n * \log(x_n) + (1 - y_n) * \log(1 - x_n)$$

Multi Margin Loss

$$l_n = \frac{\sum_i \max(0, margin - x[y] + x[i])^p}{|x|}$$

Loss Functions Example - Continuous

```
torch.manual_seed(0)
x = torch.rand([5,1])
y = torch.rand([5,1])

nn.MSELoss()(x,y)
#>>> tensor(0.1919)

nn.MSELoss(reduction='sum')(x,y)
#>>> tensor(0.9594)

nn.MSELoss(reduction='none')(x,y)
#>>> tensor([[0.0190],[0.0774],[0.6528],[0.1047],[0.1055]])
```

Loss Functions Example - Binary

```
x = torch.rand(5)
print(x)
>>> tensor([0.6147, 0.3810, 0.6371, 0.4745, 0.7136])

y = torch.empty(5).random_(2)
print(y)
>>> tensor([1., 0., 1., 0., 1.])

nn.BCELoss(reduction='none')(x,y)
>>> tensor([0.4866, 0.4797, 0.4508, 0.6433, 0.3374])
```

Loss Functions Example - Multiclass

```
torch.manual_seed(2)
x = torch.rand((4,3))
print(x)
>>> tensor([[0.6147, 0.3810, 0.6371],
            [0.4745, 0.7136, 0.6190],
            [0.4425, 0.0958, 0.6142],
            [0.0573, 0.5657, 0.5332]])

y = torch.empty(4, dtype=torch.long).random_(3)
print(y)
>>> tensor([2, 1, 0, 2])

nn.MultiMarginLoss(reduction='none')(x,y)
>>> tensor([0.5738, 0.5554, 0.6083, 0.5189])
```

Loss Functions for Distributions

- Pytorch has some loss functions implemented that work nicely with parameterized distributions
  - Poisson Negative Log Likelihood Loss
  - Gaussian Negative Log Likelihood Loss

- For numerical reasons, they are calculated differently than the usual method. For optimization it works the same

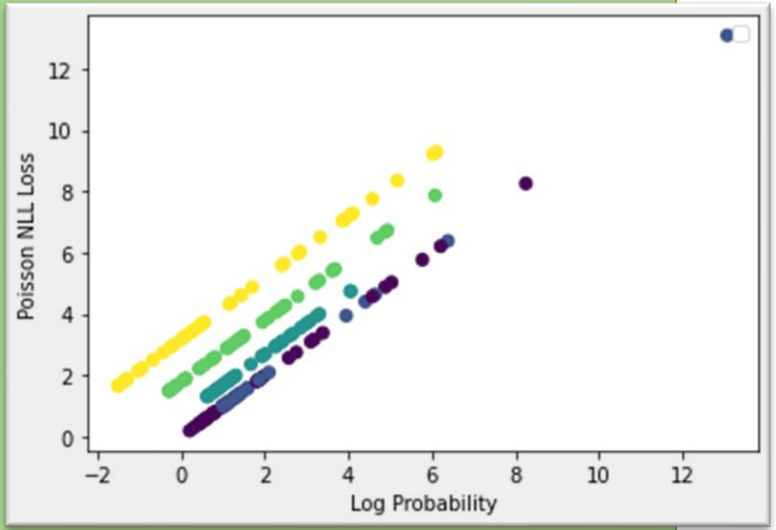
Poisson Distribution Example

```
N = 5
loss = nn.PoissonNLLLoss(reduction='none’)

# log_input would be the output of your neural network
# It predicts the log of the rate parameter for the distribution
log_input = torch.randn(N)
print(log_input)
>>> tensor([ 0.3923, -0.2236, -0.3195, -1.2050, 1.0445])

# Target is from your data – actual counts
target = torch.empty(N).random_(5)
print(target)
>>> tensor([4., 0., 0., 3., 4.])

output = loss(log_input, target)
print(output)
>>> tensor([-0.0888, 0.7997, 0.7265, 3.9148, -1.3360])
```



Gaussian Distribution Example

```
loss = nn.GaussianNLLLoss(reduction='none')
mean = torch.randn(N)
print(mean)
>>> tensor([ 0.3923, -0.2236, -0.3195, -1.2050, 1.0445])

var = torch.randn(N) + 3
print(var)
>>> tensor([2.3668, 3.5731, 3.5409, 2.6081, 1.9573])

target = torch.randn(N)
print(target)
>>> tensor([ 1.3186, 0.7476, -1.3265, -1.2413, -0.1028])

output = loss(mean, target, var)
print(output)
>>> tensor([0.6120, 0.7687, 0.7754, 0.4796, 0.6720])
```

NLL Loss for Other Distributions

- You can calculate you own NLL for other distributions using **torch.distributions**

```
# Exponential distribution example
# Network output gives the rate parameter λ
# We reshape it from (Nobs, 1) to (Nobs) using tensor.squeeze()
# This is because the distribution expects a one-dimensional input
p = net(X).squeeze(1)

# Convert the rate parameter into a distribution
dist = torch.distributions.exponential.Exponential(p)

# Find the log probability of each observation
loss = -dist.log_prob(y)
sum_loss = loss.sum()
```

NLL Loss for Two-Parameter Distribution

```
# Weibull distribution example
# Network gives the scale and concentration in the shape (Nobs,2)
p = net(X)

# Separate the scale and concentration into two tensors of shape (Nobs, 1)
# These parameters cannot be negative, so we set a minimum value
scale_p = torch.clamp(p[:,0].unsqueeze(1), min=1e-6)
concentration_p = torch.clamp(p[:,1].unsqueeze(1), min=1e-6)

# Convert the parameters into a distribution
dist = torch.distributions.weibull.Weibull(scale_p, concentration_p)

# Find the log probability of each observation
loss = -dist.log_prob(y)
sum_loss = loss.sum()
```

Backward Pass

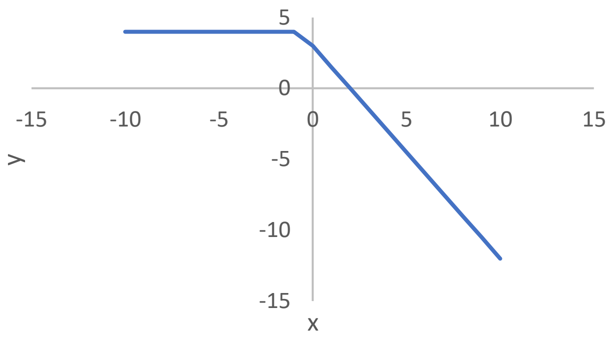
- Now that we have made the prediction and calculated the loss, we can calculate the **gradient**
  - A gradient is a derivative in multiple dimensions

- Neural networks are trained by taking incremental steps in the direction of the gradient

- The process of calculating the gradient at each point along the network is called **backpropagation**

Backpropagation

- Consider the following (very basic) network



$$y_1 = 3 * x + 2$$
$$y_2 = ReLU(y_1)$$
$$y_3 = -0.5y_2 + 4$$

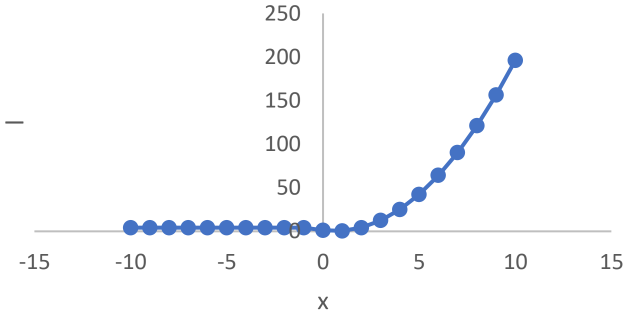
- Suppose our loss was mean squared error from the target value of 2

$$l = (y_3 - 2)^2$$

- We evaluate this loss for our given input, in this case let's consider x=5. Following forward through the network, this gives us y = -4.5

$$l = (-4.5 - 2)^2 = 42.25$$

Backpropagation



- We can take the gradient (derivative) of the loss function at this point

$$l = (y - 2)^2$$

$$\nabla l = 2y - 4$$

- Tracing this gradient back through the network gives us the gradient at each step

$$\frac{dl}{dy_3} = 2y_3 - 4$$
$= -11.5$

$$\frac{dy_3}{dy_2} = -0.5$$
$= -0.5$

$$\frac{dy_2}{dy_1} = 1$$
$= 1$

$$\frac{dy_1}{dx} = 3$$
$= 3$

$$\boldsymbol{\frac{dl}{dx} = \frac{dl}{dy_3} * \frac{dy_3}{dy_2}} \ldots$$
= 17.25

Optimization Step

- Now that we know the gradient we can adjust the weights to reduce the loss

- Using **gradient descent**, we adjust each weight by the negative of it's gradient multiplied by a learning rate

$$w_n' = lr \; * - \frac{dl}{dy_n} + w_n$$

Optimization Step

$w_3 = -0.5$

$$-\frac{dl}{dw_3} = -\frac{dl}{dy_3} * \frac{dy_3}{dw_3}$$

= 11.5

$$w_3' = 11.5 * 1e^{-3} + w_3$$

$$w_3' = -0.4885$$

$b_3 = 4$

$$-\frac{dl}{dw_3} = -\frac{dl}{dy_3} * \frac{dy_3}{db_3}$$

= 11.5

$$b_3' = 11.5 * 1e^{-3} + b_3$$

$$b_3' = 4.0115$$

$w_1 = 3$

$$-\frac{dl}{dw_1} = -\frac{dl}{dy_1} * \frac{dy_1}{dw_1}$$

= -17.25

$$w_1' = -17.25 * 1e^{-3} + w_1$$

$$w_3' = -2.98275$$

$b_1 = 2$

$$-\frac{dl}{dw_1} = -\frac{dl}{dy_1} * \frac{dy_1}{db_1}$$

= -17.25

$$b_1' = -17.25 * 1e^{-3} + b_1$$

$$b_3' = 1.98275$$

End-to-End Example

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

class FFN(nn.Module):
    def __init__(self, in_features, out_features, hidden_size):
        super().__init__()
        self.layer1 = nn.Linear(in_features, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, out_features)

    def forward(self, x):
        y = self.layer1(x)
        y = nn.ReLU()(y)
        y = self.layer2(y)
        y = F.relu(y)
        y = self.layer3(y)
        return y
```

End-to-End Example

```
X = torch.randn([100,4])
y = torch.sum(X, axis=1)

net = FFN(4, 1, 4)
optimizer = torch.optim.SGD(net.parameters(), lr=1e-2)
n_epochs = 400
losses = np.zeros(n_epochs)

for e in range(n_epochs):
    optimizer.zero_grad()
    y_pred = net(X).squeeze(1)
    loss = nn.MSELoss()(y_pred, y)
    loss.backward()
    losses[e] = loss
    optimizer.step()
```

End-to-End Example