UofSC

Machine Learning: PyTorch

Forest Agostinelli
University of South Carolina

Topics Covered in This Class

- **Part 1: Search**
  - Pathfinding
    - Uninformed search
    - Informed search
  - Adversarial search
  - Optimization
    - Local search
    - Constraint satisfaction
- **Part 2: Knowledge Representation and Reasoning**
  - Propositional logic
  - First-order logic
  - Prolog
- **Part 3: Knowledge Representation and Reasoning Under Uncertainty**
  - Probability
  - Bayesian networks
- **Part 4: Machine Learning**
  - Supervised learning
    - Inductive logic programming
    - Linear models
    - Deep neural networks
    - PyTorch
  - Reinforcement learning
    - Markov decision processes
    - Dynamic programming
    - Model-free RL
  - Unsupervised learning
    - Clustering
    - Autoencoders

Outline

- Automatic differentiation
- PyTorch

Verification

- We need to make sure the output of the neural network matches the expected output and that the gradient is correct
- Verifying the gradient can be done with finite differences
  - $\lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
- For every single parameter
  - Do a forward pass through the network
  - Add a small value to that parameter (i.e. 10^-5)
  - Do a forward pass again
  - Estimate gradient and compare it to the gradient computed by your code

Verification

- Building a separate forward and backwards pass for every iteration of a deep neural network can be very time consuming
- Fortunately, we can divide deep neural networks into functionally independent components (layers)
  - Fully connected layers
  - Convolutional layers
  - Residual layers
  - Activation function layers (logistic, ReLU, tanh, etc.)
- We can then verify each layer independently and then compose these layers to form a deep neural networks

Deep Learning Software

- Building a deep neural network consists of defining a forward pass (obtaining the output) and the backwards pass (backpropagation)
- After backpropagation, the gradient obtained is then used to adjust the parameters
- Furthermore, deep neural networks consist of matrix multiplications which can benefit from parallelization on the simpler, but plentiful, processors of GPUs
- Deep learning software automates some, or all, of this process

Deep Learning Software

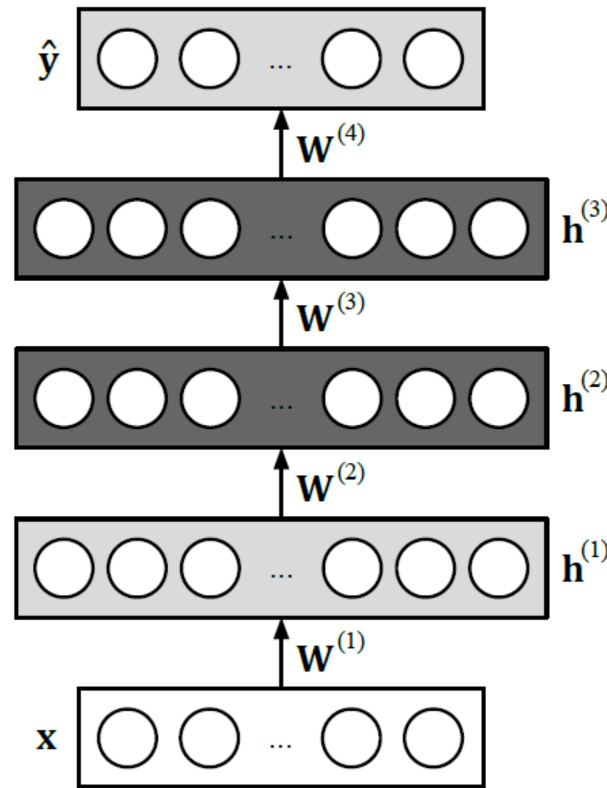
- Modern day deep learning software has abstracted away almost all aspects of the backward pass and many aspects of the forward pass
- However, understanding them can be crucial to your research

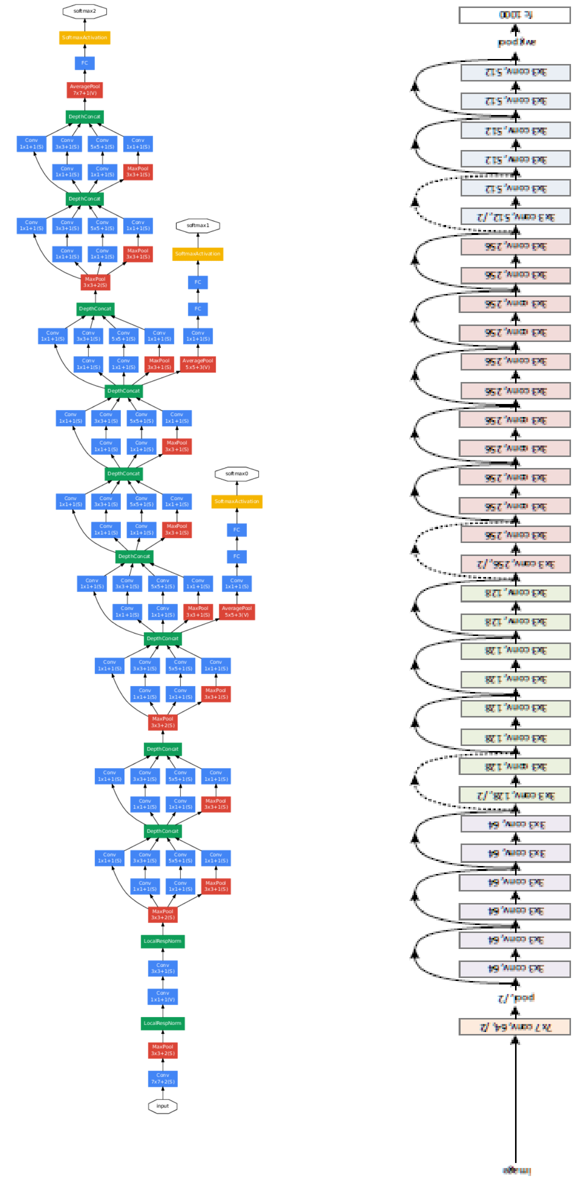
Automatic Differentiation

- Objective: $E(\boldsymbol{w}) = \frac{1}{2n}\sum_n || \boldsymbol{y}_n - f(\boldsymbol{x}_n, \boldsymbol{w})||_2^2$
- Gradient: $\nabla_{\mathbf{w}} E(\mathbf{w}) = \frac{1}{n}\sum_n || \boldsymbol{y}_n - f(\boldsymbol{x}_n, \boldsymbol{w})||_2 \nabla_{\mathbf{w}}\, f(\boldsymbol{x}_n, \boldsymbol{w})$
- For every function we use to calculate $f(\boldsymbol{x}_n, \boldsymbol{w})$, we define:
  - Forward pass
    - Inputs to outputs
  - Backward pass
    - Updates backpropagated gradient to update parameters
    - Differentiate with respect to inputs, multiply result by backpropagated gradient (chain rule)
- Can create many different types of deep neural networks without having to define a backward pass

Automatic Differentiation

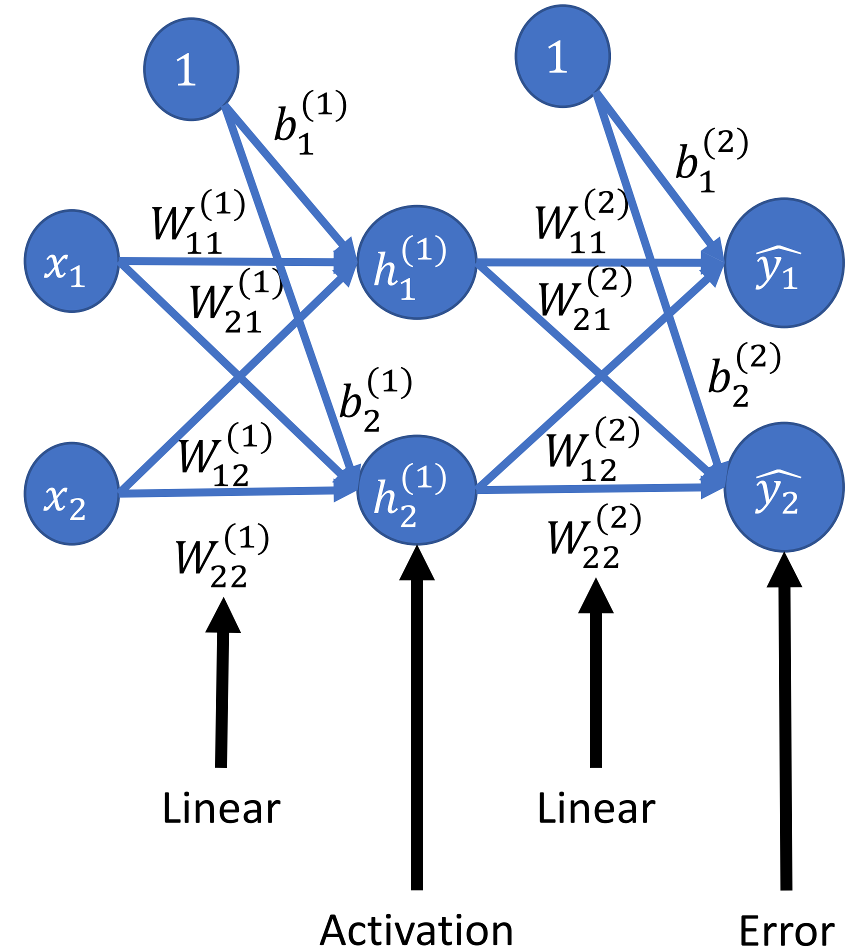
- Can create all these deep neural network architectures using modern deep learning software by only defining a forward pass





Automatic Differentiation

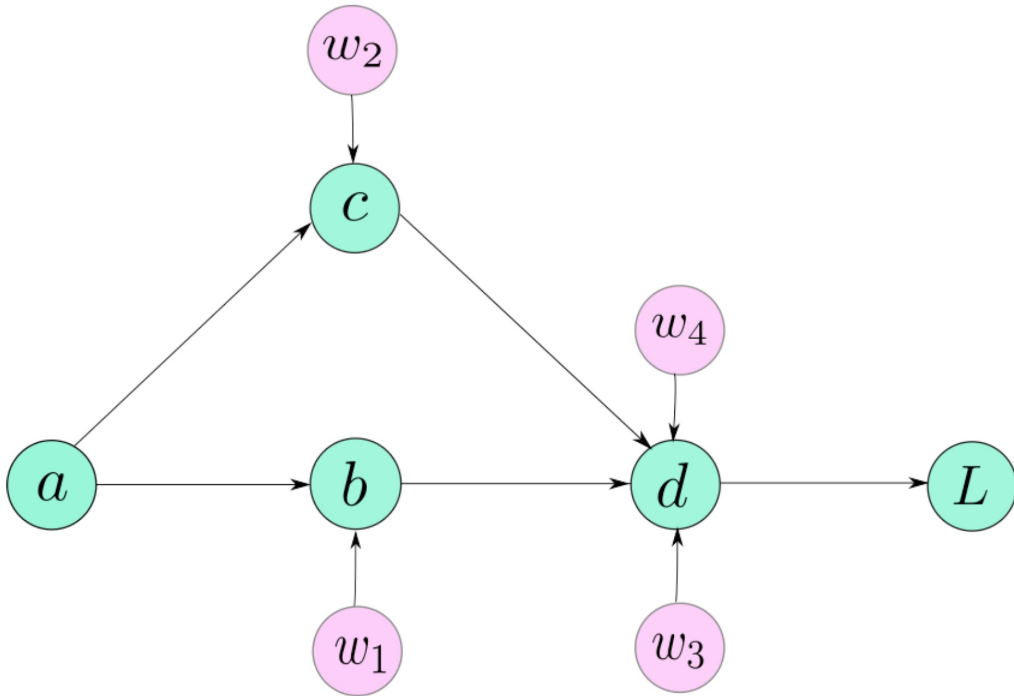
- $\hat{\boldsymbol{y}} = \boldsymbol{W}^{(2)}\sigma(\boldsymbol{W}^{(1)}\boldsymbol{x})$
  - Linear
  - Activation
  - Error
- Can then create networks of this structure with arbitrary depth



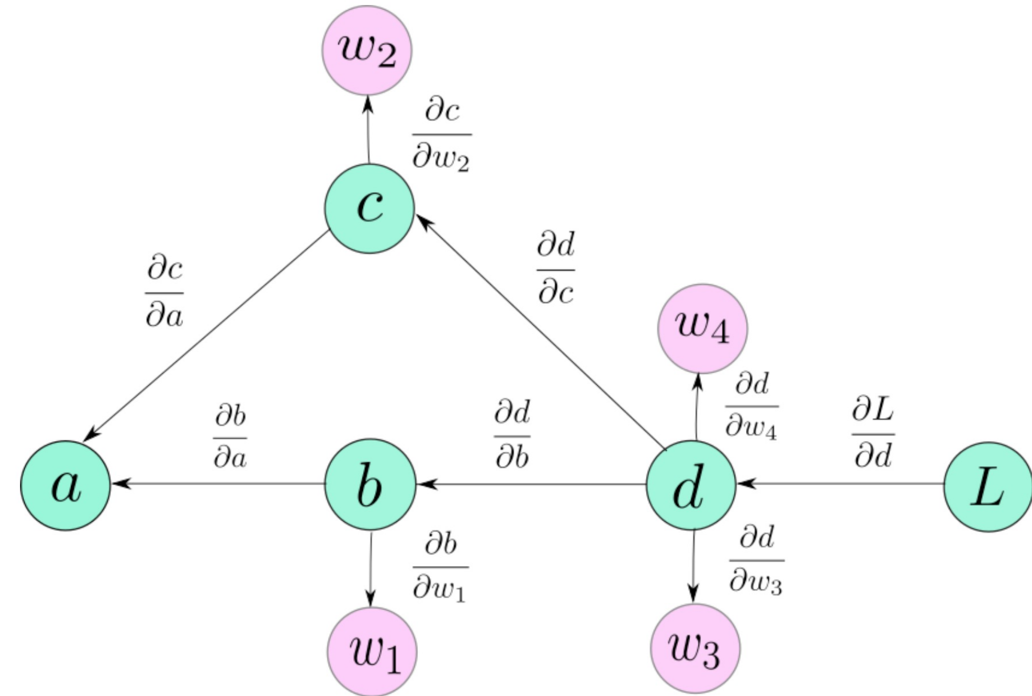
Computation Graph

```
b = w1 * a
c = w2 * a
d = (w3 * b) + (w4 * c)
L = f(d)
```

Forward



Backward

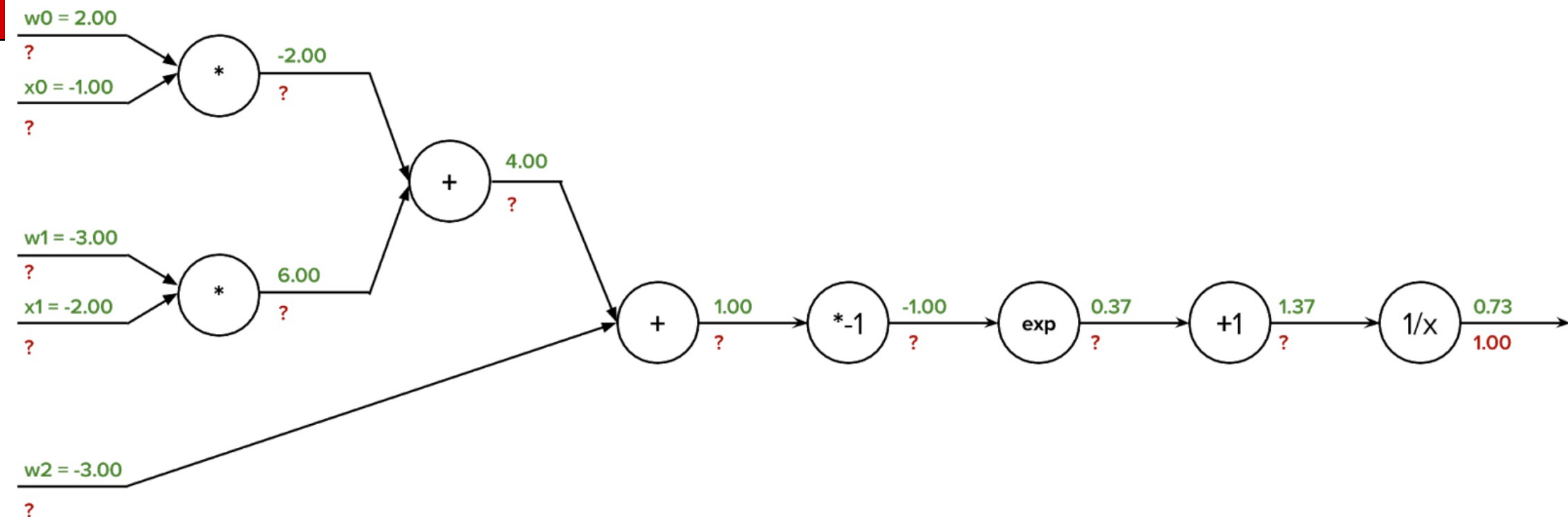


https://towardsdatascience.com/getting-started-with-pytorch-part-1-understanding-how-automatic-differentiation-works-5008282073ec

Backpropagation Exercise

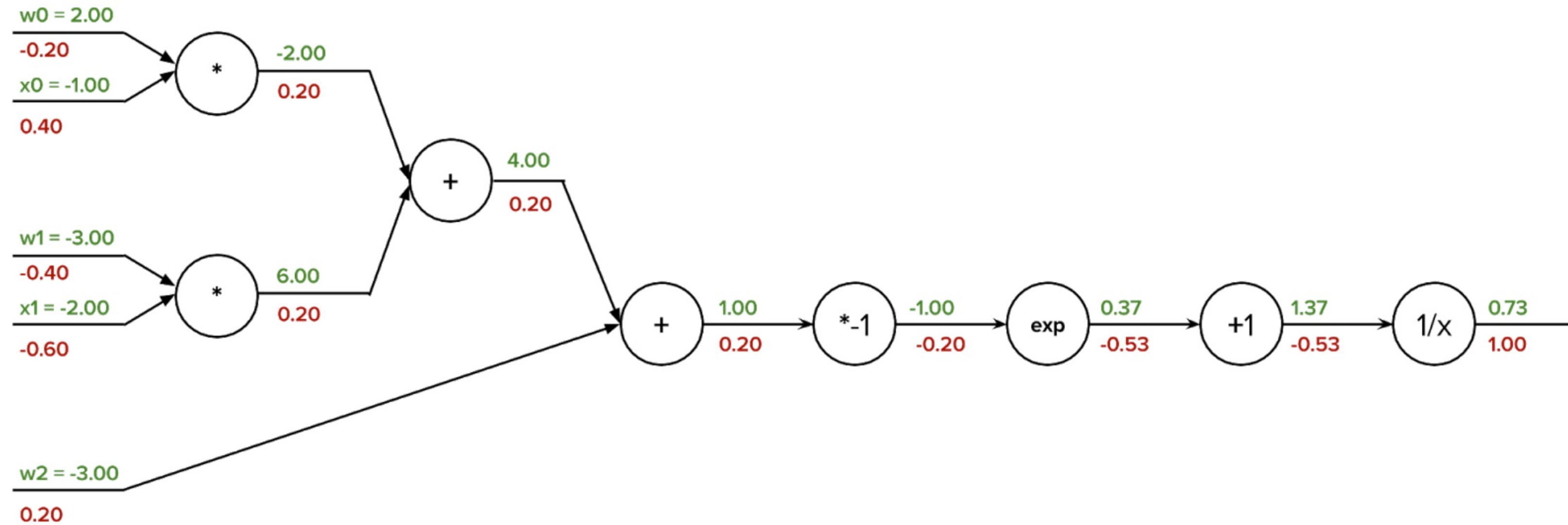
- https://cs230.stanford.edu/winter2020/section3_exercises.pdf
- What are the symbolic gradients?

Now let's perform backpropagation through a single neuron of a neural network with a sigmoid activation. Specifically, we will define the pre-activation $z = w_o x_o + w_1 x_1 + w_2$ and we will define the activation value $\alpha = \sigma(z) = 1 / (1 + e^{-z})$. The computation graph is visualized below:



In the graph we've filled out the **forward activations**, on the top of the lines, as well as the upstream gradient (gradient of the loss with respect to our neuron, $\partial L / \partial \alpha$). Use this information to compute the rest of the gradients (labelled with **question marks**) throughout the graph.

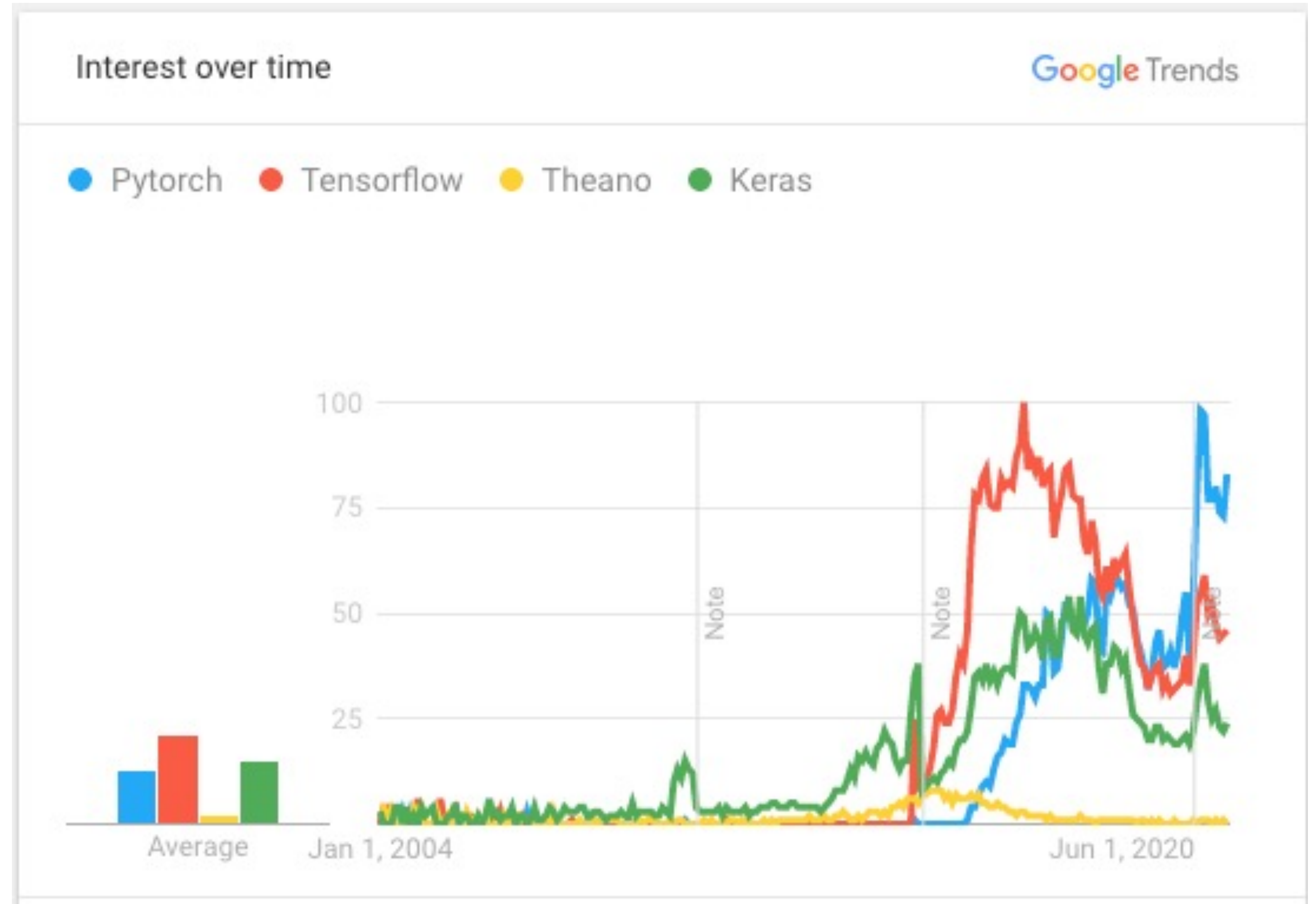
Backpropagation Exercise



1. $\partial\alpha / \partial x_0 = \sigma(z)\,(1 - \sigma(z))\, w_0$
2. $\partial\alpha / \partial w_0 = \sigma(z)\,(1 - \sigma(z))\, x_0$
3. $\partial\alpha / \partial x_1 = \sigma(z)\,(1 - \sigma(z))\, w_1$
4. $\partial\alpha / \partial w_1 = \sigma(z)\,(1 - \sigma(z))\, x_1$
5. $\partial\alpha / \partial w_2 = \sigma(z)\,(1 - \sigma(z))$

Deep Learning Software

- Handmade
  - C++/MATLAB/etc.
- Automatic Differentiation
  - Theano
  - Torch
  - Caffe
  - TensorFlow
  - PyTorch



Outline

- Automatic differentiation
- PyTorch

PyTorch: Tensor

- A multi-dimensional matrix whose elements are of a single data type
- The neural network inputs, intermediate outputs, and outputs are all of type Tensor
- Very similar to a numpy ndarray

PyTorch: Training

- **Important**: If you want to keep the loss around, use loss.item(), not loss
  - Otherwise, computation graph for previous computation is kept
  - Will eventually run out of memory
- **Important**: In most cases, PyTorch will be expecting data of type float32. You can either (for inputs and outputs)
  - Convert to float32 before making tensor
    - nnet_inputs_np = nnet_inputs_np.astype(np.float32)
  - Covert to float32 after making tensor
    - nnet_inputs = nnet_inputs.float()

```
# initialize
train_itr: int = 0
nnet.train()

criterion = nn.MSELoss()  # loss function, mean squared error
optimizer = optim.Adam(nnet.parameters(), lr=lr)

while train_itr < num_itrs:
    # zero the parameter gradients
    optimizer.zero_grad()

    # set learning rate
    lr_itr = lr * (lr_d ** train_itr)  # exponential decay
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr_itr

    # get data in numpy format
    nnet_inputs_np, nnet_targets_np = data

    # send data to device (i.e. CPU or GPU)
    nnet_inputs = torch.tensor(nnet_inputs_np, device=device)
    nnet_targets = torch.tensor(nnet_targets_np, device=device)

    # forward
    nnet_outputs = nnet(nnet_inputs)

    # loss
    loss = criterion(nnet_outputs, nnet_targets)

    # backpropagation
    loss.backward()

    # step
    optimizer.step()
```

PyTorch: Evaluation

```
nnet.eval()
nnet_input = torch.tensor(nnet_input_np, device=device)
output = nnet(nnet_input.float()).cpu().data.numpy()
```

- Put the neural network in evaluation mode to turn off behavior exclusive to training
  - Batch normalization
  - Dropout
- .cpu() ensures that the data is on the CPU and not the GPU

PyTorch: Neural Network Model

- Define parameters
- Define forward pass

```python
class FarmGridStateValueNNet(nn.Module):

    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()

        self.lin1 = nn.Linear(input_dim, hidden_dim)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(hidden_dim, hidden_dim)
        self.act2 = nn.ReLU()

        self.lin3 = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = self.lin1(x)
        x = self.act1(x)

        x = self.lin2(x)
        x = self.act2(x)

        x = self.lin3(x)

        return x
```

PyTorch: Batch Normalization

- Maintains a running average of mean and variance during training
- Uses running average during eval so that evaluation is not stochastic

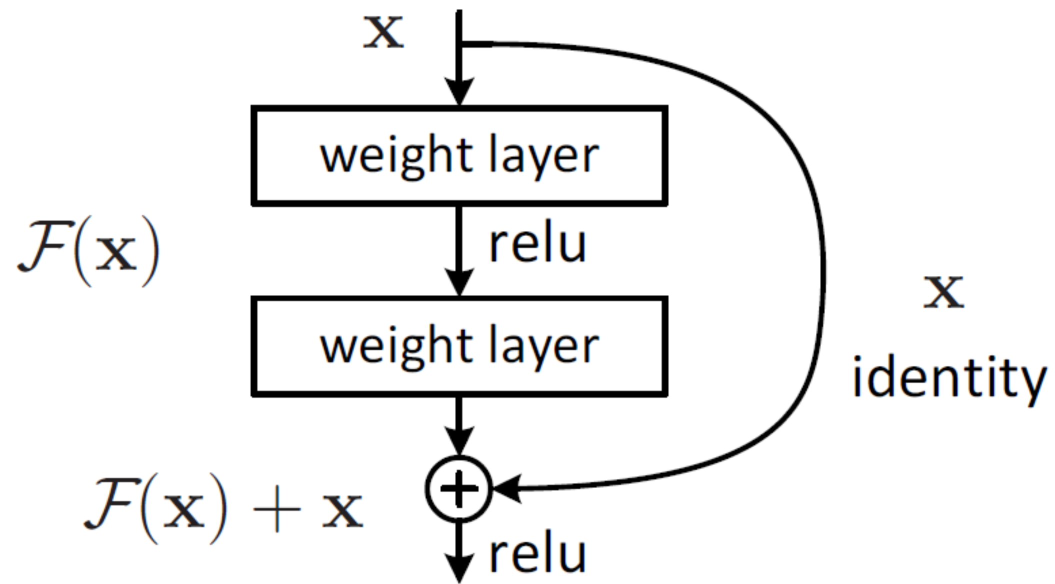
```python
class FarmGridStateValueNNet(nn.Module):

    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()

        self.lin1 = nn.Linear(input_dim, hidden_dim)
        self.bn1 = nn.BatchNorm1d(hidden_dim)
        self.act1 = nn.ReLU()

        self.lin2 = nn.Linear(hidden_dim, hidden_dim)
        self.bn2 = nn.BatchNorm1d(hidden_dim)
        self.act2 = nn.ReLU()

        self.lin3 = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = self.lin1(x)
        x = self.bn1(x)
        x = self.act1(x)

        x = self.lin2(x)
        x = self.bn2(x)
        x = self.act2(x)

        x = self.lin3(x)

        return x
```

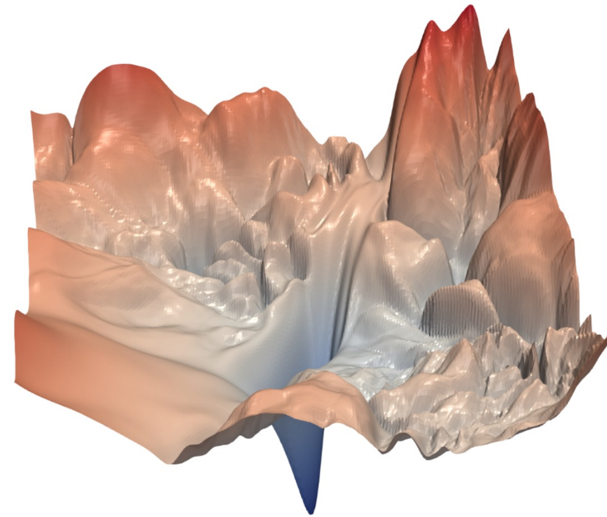
PyTorch: Generalized Fully Connected Model

- For parameters, use nn.ModuleList instead of List

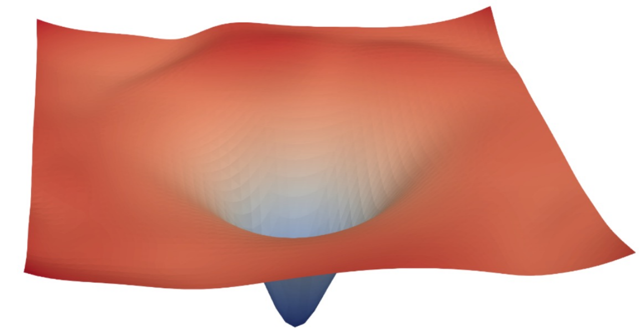
```
class FullyConnectedModel(nn.Module):
    def __init__(self, input_dim: int, layer_dims: List[int], layer_batch_norms: List[bool], layer_acts: List[str])
        super().__init__()
        self.layers: nn.ModuleList[nn.ModuleList] = nn.ModuleList()

        # layers
        for layer_dim, batch_norm, act in zip(layer_dims, layer_batch_norms, layer_acts):
            module_list = nn.ModuleList()

            # linear
            module_list.append(nn.Linear(input_dim, layer_dim))

            # batch norm
            if batch_norm:
                module_list.append(nn.BatchNorm1d(layer_dim))

            # activation
            act = act.upper()
            if act == "RELU":
                module_list.append(nn.ReLU())
            elif act != "LINEAR":
                raise ValueError("Un-defined activation type %s" % act)

            self.layers.append(module_list)

            input_dim = layer_dim

    def forward(self, x):
        x = x.float()

        module_list: nn.ModuleList
        for module_list in self.layers:
            for module in module_list:
                x = module(x)

        return x
```

PyTorch: Residual Networks





(a) without skip connections



(b) with skip connections

Architecture

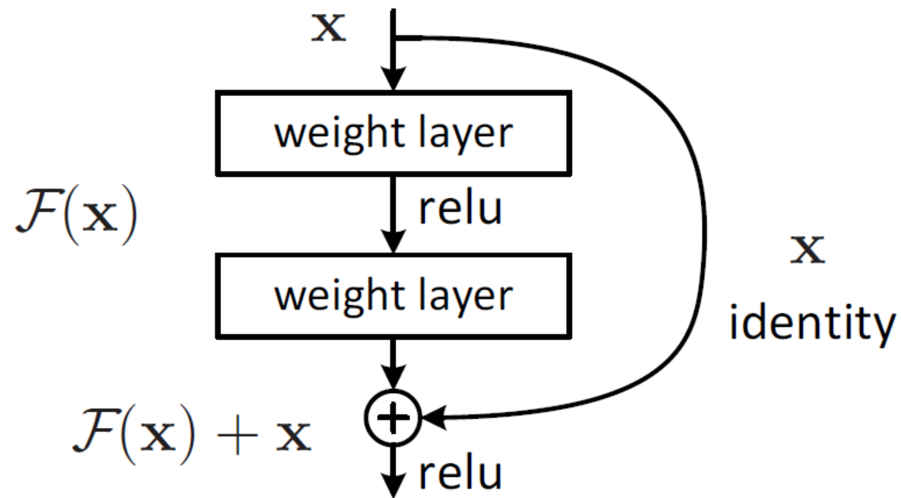
Loss surface

```python
class ResnetModel(nn.Module):
    def __init__(self, resnet_dim: int, num_resnet_blocks: int, out_dim: int, batch_norm: bool):
        super().__init__()
        self.blocks = nn.ModuleList()

        # resnet blocks
        for block_num in range(num_resnet_blocks):
            block_net = FullyConnectedModel(resnet_dim, [resnet_dim] * 2, [batch_norm] * 2, ["RELU", "LINEAR"])
            module_list: nn.ModuleList = nn.ModuleList([block_net])

            self.blocks.append(module_list)

        # output
        self.fc_out = nn.Linear(resnet_dim, out_dim)

    def forward(self, x):
        # resnet blocks
        module_list: nn.ModuleList
        for module_list in self.blocks:
            res_inp = x
            for module in module_list:
                x = module(x)

            x = F.relu(x + res_inp)

        # output
        x = self.fc_out(x)
        return x
```

PyTorch: Residual Networks

- Residual networks require that input is the same dimension as the dimension of the hidden layers
- Can use a linear transformation to make them the same dimension

```python
class FarmGridStateValueNNet(nn.Module):

    def __init__(self, input_dim: int, hidden_dim: int, num_blocks: int):
        super().__init__()

        self.lin = nn.Linear(input_dim, hidden_dim)
        self.resnet = ResnetModel(hidden_dim, num_blocks, 1, True)

    def forward(self, x):
        x = self.lin(x)
        x = self.resnet(x)

        return x
```



PyTorch: Initialization

```
# linear
module_list.append(nn.Linear(input_dim, layer_dim))
module_list[-1].weight.data.normal_(0, 0.1)
module_list[-1].bias.data.zero_()
```

- Functions that end with "_" indicate the operation modifies the object "in-place" instead of returning a new object
- PyTorch's default initialization is usually good
  - They do not initialize bias to zero, sometimes may be good to do that

PyTorch: GPU vs CPU

- The CUDA_VISIBLE_DEVICES determines which GPUs will get used
  - If set, device is always "cuda:0"
  - CUDA_VISIBLE_DEVICES will ensure the correct GPUs will get used
- DataParallel
  - DataParallel will distribute computation across multiple GPUs
- On CPU sometimes good to limit the number of threads to 1

```python
on_gpu: bool = False
if ('CUDA_VISIBLE_DEVICES' in os.environ) and torch.cuda.is_available():
    device = torch.device("cuda:0")
    on_gpu = True
else:
    device = torch.device("cpu")
```

```python
nnet.to(device)
if on_gpu:
    nnet = nn.DataParallel(nnet)
```

PyTorch: Saving/Loading a Model

- **Important**: PyTorch only saves parameters, not computation graph
  - When loading, the nnet must correspond to the same nn.Module as the one that was used to train the nnet

```
torch.save(nnet.state_dict(), "%s/model_state_dict.pt" % save_dir)
```

```
def load_nnet(model_file: str, nnet: nn.Module, device: torch.device) -> nn.Module:
    # get state dict
    state_dict = torch.load(model_file, map_location=device)

    # remove module prefix (in case of data parallel)
    new_state_dict = OrderedDict()
    for k, v in state_dict.items():
        k = re.sub('^module\.', '', k)
        new_state_dict[k] = v

    # set state dict
    nnet.load_state_dict(new_state_dict)

    nnet.eval()

    return nnet
```

PyTorch: .detach

- When you want to remove something from the computation graph, use .detach
  - a.detach() returns a new Tensor that is detached
  - a.detach_() detaches the Tensor in place

PyTorch: Debugging

- Debugging is simple
- pdb.set_trace() sets a breakpoint
- Newer versions of Python can just use breakpoint()

```
def forward(self, x):
    # resnet blocks
    module_list: nn.ModuleList
    for module_list in self.blocks:
        res_inp = x
        for module in module_list:
            x = module(x)

        import pdb  # breakpoint
        pdb.set_trace()

        x = F.relu(x + res_inp)

    # output
    x = self.fc_out(x)
```

TensorBoard

- Developed by TensorFlow
- Useable with PyTorch

TensorBoard.dev SCALARS

My latest experiment

Simple comparison of several hyperparameters

Show data download links

Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing

0.6

Horizontal Axis

STEP RELATIVE WALL

Runs

Write a regex to filter runs

lr_1E-03,conv=1,fc=2

lr_1E-03,conv=2,fc=2

lr_1E-04,conv=1,fc=2

lr_1E-04,conv=2,fc=2

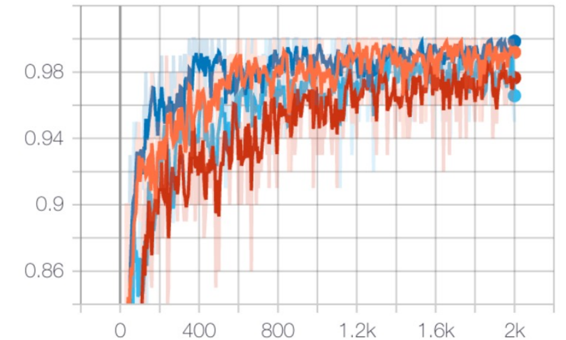
TOGGLE ALL RUNS

experiment AdYd1TgeTlaLWXx6l8JUbA

accuracy

accuracy

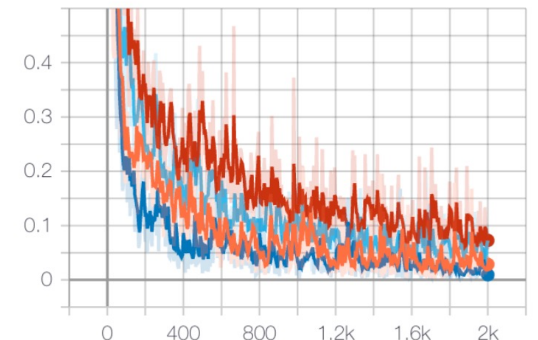
tag: accuracy/accuracy



xent

xent_1

tag: xent/xent_1



PyTorch Tutorial

- PyTorch
  - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- TensorBoard
  - https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html