CS 179: LECTURE 15

INTRODUCTION TO CUDNN (CUDA DEEP NEURAL NETS)

LAST TIME

- We derived the minibatch stochastic gradient descent algorithm for neural networks, i.e. $\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \frac{1}{k}\eta\left(\mathbf{X}^{(\ell-1)'}\Delta^{(\ell)^T}\right)$
  - Mostly matrix multiplications to compute $\Delta^{(\ell)}$
  - One other derivative, $\theta'\left(\boldsymbol{Z}_{ij}^{(\ell)}\right)$
- cuBLAS (Basic Linear Algebra Subroutines) already does matrix multiplications for us
- cuDNN will take care of these "other" derivatives for us

TODAY

- Using cuDNN to do deep neural networks

SETTING UP CUDNN

- `cudnnHandle_t`
  - Like cuBLAS, you need to maintain cuDNN library context
  - Call `cudnnCreate(cudnnHandle_t *handle)` to initialize the context
  - Call `cudnnDestroy(cudnnHandle_t handle)` to clean up the context

HANDLING ERRORS

- Almost every function we will talk about today returns a `cudnnStatus_t` (an enum saying whether a cuDNN call was successful or how it failed)
- Like standard CUDA, we will provide you with a `checkCUDNN(cudnnStatus_t status)` wrapper function that parses any error statuses for you
- Make sure you wrap every function call with this function so you know where and how your code breaks!

REMINDERS ABOUT CUDA

- Pattern of allocate → initialize → free (reiterate here for students who may not be as comfortable with C++)

DATA REPRESENTATION

- `cudnnTensor_t`
  - For the purposes of cuDNN (and much of machine learning), a tensor is just a multidimensional array
  - A wrapper around a "flattened" 3-8 dimensional array
  - Used to represent minibatches of data
  - For now, we will be using "flattened" 4D arrays to represent each minibatch `X`

DATA REPRESENTATION

- `cudnnTensor_t`
  - Consider the case where each individual training example x is just a vector (so the last two axes will have size 1 each)
  - Then `X[n,c,0,0]` is the value of component `c` of example `n`
    - If axis `<k>` has size `size<k>`, then `X[n,c,h,w]` (pseudocode) is actually `X[n*size0*size1*size2 + c*size0*size1 + h*size0 + w]`

DATA REPRESENTATION

- `cudnnTensor_t`
  - More generally, a single training example may itself be a matrix or a tensor.
  - For example, in a minibatch of RGB images, we may have `X[n,c,h,w]`, where `n` is the index of an image in the minibatch, `c` is the channel (R = 0, G = 1, B = 2), and `h` and `w` index a pixel (`h`, `w`) in the image (`h` and `w` are height and width)

DATA REPRESENTATION

- `cudnnTensorDescriptor_t`
  - Allocate by calling `cudnnCreateTensorDescriptor(cudnnTensorDescriptor_t *desc)`
  - The ordering of array axes is defined by an enum called a `cudnnTensorFormat_t` (since we are indexing as `X[n,c,h,w]`, we will use `CUDNN_TENSOR_NCHW`)
  - A `cudnnDataType_t` specifies the data type of the tensor (we will use `CUDNN_DATA_FLOAT`)

DATA REPRESENTATION

- `cudnnTensorDescriptor_t`
  - Initialize by calling `cudnnSetTensor4dDescriptor( cudnnTensorDescriptor_t desc, cudnnTensorFormat_t format, cudnnDataType_t dataType, int n, int c, int h, int w)`
  - Free by calling `cudnnDestroyTensorDescriptor( cudnnTensorDescriptor_t desc)`

DATA REPRESENTATION

- `cudnnTensorDescriptor_t`
  - Get the contents by calling `cudnnGetTensor4dDescriptor( cudnnTensorDescriptor_t desc, cudnnDataType_t dataType, int *n, int *c, int *h, int *w, int *nStr, int *cStr, int *hStr, int *wStr)`
  - Standard trick of returning by setting output parameters
  - Don't worry about the strides `nStr, cStr, hStr, wStr`

RELATION TO ASSIGNMENT 5

- Forward pass (Algorithm)
  - For each minibatch $(\mathbf{X}^{(0)}, \mathbf{Y})$ of k training examples
    - (Each example and its label are a column in matrices $\mathbf{X}^{(0)}$ and $\mathbf{Y}$ respectively)
    - For each ℓ counting up from 1 to L
      - Compute matrix $\mathbf{Z}^{(\ell)} = {\mathbf{W}^{(\ell)}}^T {\mathbf{X}^{(\ell-1)}}'$
      - Compute matrix $\mathbf{X}^{(\ell)} = \theta^{(\ell)}\left(\mathbf{Z}^{(\ell)}\right)$
    - Our model's prediction is $\mathbf{X}^{(L)}$

RELATION TO ASSIGNMENT 5

- Forward pass (Implementation)
  - Calculate the expected sizes of the inputs $\mathbf{X}^{(\ell-1)}$ and outputs $\mathbf{Z}^{(\ell)}$ of each layer and allocate arrays of the appropriate size
    - Input $\mathbf{X}^{(\ell-1)}$ has shape $d_{\ell-1} \times k$
    - Weight matrix $\mathbf{W}^{(\ell)}$ has shape $d_{\ell-1} \times d_\ell$
    - Outputs $\mathbf{Z}^{(\ell)} = {\mathbf{W}^{(\ell)}}^T \mathbf{X}^{(\ell-1)'}$ and $\mathbf{X}^{(\ell)} = \theta\left(\mathbf{Z}^{(\ell)}\right)$ have shape $d_\ell \times k$
  - Initialize tensor descriptors for each $\mathbf{X}^{(\ell)}$ and $\mathbf{Z}^{(\ell)}$

RELATION TO ASSIGNMENT 5

- Forward pass (Implementation)
  - Note that cuBLAS puts matrices in column-major order, so $\mathbf{X}^{(\ell)}$ and $\mathbf{Z}^{(\ell)}$ will be tensors of shape $(k, d_\ell, 1, 1)$
  - In this assignment, the skeleton code we provide will handle the bias terms for you (this is the extra $x_0 = 1$ term that we've been carrying in $x' = (1, x)$ this whole time)
  - Just remember that when we write $\mathbf{X}^{(\ell)'}$, we are implicitly including this bias term!

RELATION TO ASSIGNMENT 5

- Backward pass (Algorithm)
  - Initialize gradient matrix $\Delta^{(L)} = \mathbf{X}^{(L)} - \mathbf{Y}$
  - For each ℓ counting down from L to 1
    - Calculate $\mathbf{A}^{(\ell)} = \nabla_{\mathbf{X}^{(\ell-1)}{}'}[J] = \mathbf{W}^{(\ell)}\Delta^{(\ell)}$
    - Calculate $\Delta_{ij}^{(\ell-1)} = \frac{\partial J}{\partial \mathbf{Z}_{ij}^{(\ell-1)}} = \mathbf{A}_{ij}^{(\ell)}\theta^{(\ell-1)'}\left(\mathbf{Z}_{ij}^{(\ell-1)}\right)$ for each $i = 1, \dots, k$ and $j = 1, \dots, d_{\ell-1}$
    - Update $\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \frac{1}{k}\eta\left(\mathbf{X}^{(\ell-1)'}{\Delta^{(\ell)}}^{T}\right)$

RELATION TO ASSIGNMENT 5

- Backward pass (Implementation)
  - Each $\Delta^{(\ell)}$ matrix has the same shape as the input $\mathbf{X}^{(\ell)}$ to its corresponding layer, i.e. $k \times d_\ell$
    - Have each $\Delta^{(\ell)}$ share a tensor descriptor with its corresponding $\mathbf{X}^{(\ell)}$
    - Update each $\mathbf{W}^{(\ell)}$ using cuBLAS's GEMM
  - cuDNN needs the associated tensor descriptor when applying the derivative of the activation/nonlinearity θ

ACTIVATION FUNCTIONS

- `cudnnActivationDescriptor_t`
  - Allocate with `cudnnCreateActivationDescriptor(cudnnActivationDescriptor_t *desc)`
  - Destroy with `cudnnDestroyActivationDescriptor(cudnnActivationDescriptor_t desc)`

ACTIVATION FUNCTIONS

- `cudnnActivationMode_t`
  - An enum that specifies the type of activation we should apply after any given layer
  - Specify as `CUDNN_ACTIVATION_<type>`
  - `<type>` can be `SIGMOID`, `RELU`, `TANH`, `CLIPPED_RELU`, or `ELU` (the last 2 are fancier activations that address some of the issues with ReLU); use `RELU` for this assignment

ACTIVATION FUNCTIONS

- Graphs of activations as a reminder

ACTIVATION FUNCTIONS

- `cudnnNanPropagation_t`
  - An enum that specifies whether to propagate NAN's
  - Use `CUDNN_PROPAGATE_NAN` for this assignment

ACTIVATION FUNCTIONS

- `cudnnActivationDescriptor_t`
  - **Set with** `cudnnSetActivationDescriptor( cudnnActivationDescriptor_t desc, cudnnActivationMode_t mode, cudnnNanPropagation_t reluNanOpt, double coef)`
  - `coef` **is relevant only for clipped ReLU and ELU activations, so just use** `0.0` **for this assignment**

ACTIVATION FUNCTIONS

- `cudnnActivationDescriptor_t`
  - Get contents with `cudnnGetActivationDescriptor( cudnnActivationDescriptor_t desc, cudnnActivationMode_t *mode, cudnnNanPropagation_t *reluNanOpt, double *coef)`
  - `coef` is relevant only for clipped ReLU and ELU activations, so just give it a reference to a `double` for throwaway values

ACTIVATION FUNCTIONS

- Forward pass for an activation $\theta^{(\ell)}$
  - Computes tensor x = alpha[0] $*$ $\theta^{(\ell)}$(z) + beta[0] $*$ x
  - Note: numeric $*$ means element-wise multiplication
  - ```
    cudnnActivationForward(
    cudnnHandle_t handle,
    cudnnActivationDescriptor_t activationDesc,
    void *alpha,
    cudnnTensorDescriptor_t zDesc, void *z,
    void *beta,
    cudnnTensorDescriptor_t xDesc, void *x)
    ```
    ```

ACTIVATION FUNCTIONS

- Backward pass for an activation $\theta^{(\ell-1)}$
  - Computes dz = alpha[0] $* \nabla_z \theta^{(\ell-1)}$(z) $*$ dx + beta[0] $*$ dz
  - ```
    cudnnActivationBackward(
    cudnnHandle_t handle,
    cudnnActivationDescriptor_t activationDesc,
    void *alpha,
    cudnnTensorDescriptor_t xDesc,  void *x,
    cudnnTensorDescriptor_t dxDesc, void *dx,
    void *beta,
    cudnnTensorDescriptor_t zDesc,  void *z,
    cudnnTensorDescriptor_t dzDesc, void *dz)
    ```
    ```

ACTIVATION FUNCTIONS

- Backward pass for an activation $\theta^{(\ell-1)}$
  - Computes dz = alpha[0] * $\nabla_{\mathrm{z}}\theta^{(\ell-1)}(\mathrm{z})$ * dx + beta[0] * dz
  - These are element-wise products, not matrix products!
  - `x`: output of the activation, $\mathbf{X}^{(\ell-1)} = \theta^{(\ell-1)}\left(\mathbf{Z}^{(\ell-1)}\right)$
  - `dx`: derivative wrt `x`, $\mathbf{A}^{(\ell)} = \nabla_{\mathbf{X}^{(\ell-1)}}[J] = \mathbf{W}^{(\ell)}{\Delta^{(\ell)}}^{T}$
  - `z`: input to the activation, $\mathbf{Z}^{(\ell-1)}$
  - `dz`: tensor to accumulate $\Delta^{(\ell-1)} = \nabla_{\mathbf{Z}^{(\ell-1)}}[J]$ as output

SOFTMAX/CROSS-ENTROPY LOSS

- Consider a single training example $x^{(0)}$ transformed as $x^{(0)} \to z^{(1)} \to x^{(1)} \to \cdots \to z^{(L)} \to x^{(L)}$
- The softmax function is $x_i^{(L)} = p_i\left(z^{(L)}\right) = \frac{\exp\left(z_i^{(L)}\right)}{\sum_{j=1}^{d_L} \exp\left(z_j^{(L)}\right)}$
- The cross-entropy loss is $J\left(z^{(L)}\right) = -\sum_{i=1}^{d_L} y_i \ln\left(p_i(z^{(L)}\right)$
- Gives us a notion of how good our classifier is

SOFTMAX/CROSS-ENTROPY LOSS

- Forward pass
  - Computes tensor x = alpha[0] * softmax(z) + beta[0] * x
  - ```
    cudnnSoftmaxForward(cudnnHandle_t handle,
    cudnnSoftmaxAlgorithm_t alg,
    cudnnSoftmaxMode_t mode,
    void *alpha,
    cudnnTensorDescriptor_t zDesc, void *z,
    void *beta,
    cudnnTensorDescriptor_t xDesc, void *x)
    ```
    ```

SOFTMAX/CROSS-ENTROPY LOSS

- `cudnnSoftmaxAlgorithm_t`
  - Enum that specifies how to do compute the softmax
  - Use `CUDNN_SOFTMAX_ACCURATE` for this class (scales everything by $\max_i \left(z_i^{(L)} \right)$ to avoid overflow)
  - The other options are `CUDNN_SOFTMAX_FAST` (less numerically stable) and `CUDNN_SOFTMAX_LOG` (computes the natural log of the softmax function)

SOFTMAX/CROSS-ENTROPY LOSS

- `cudnnSoftmaxMode_t`
  - Enum that specifies over which data to compute the softmax
  - `CUDNN_SOFTMAX_MODE_INSTANCE` does it over the entire input (sum over all `c`, `h`, `w` for a single `n` in `X[n,c,h,w]`)
  - `CUDNN_SOFTMAX_MODE_CHANNEL` does it over each channel (sum over all `c` for each `n`, `h`, `w` triple in `X[n,c,h,w]`)
- Since `h` and `w` are both size 1 here, either is fine to use

SOFTMAX/CROSS-ENTROPY LOSS

- Backward pass
  - cuDNN has a built-in function to compute the gradient of the softmax activation on its own
  - However, when coupled with the cross-entropy loss, we get the following gradient wrt $\mathbf{Z}^{(L)}$: $\Delta^{(L)} = \nabla_{\mathbf{Z}^{(L)}}[J] = \mathbf{X}^{(L)} - \mathbf{Y}$
  - This is easier and faster to compute manually!
- Therefore, you will implement the kernel for this yourself

SOFTMAX WITH OTHER LOSSES

- Backward pass
  - For different losses, use the following function:
  - ```
    cudnnSoftmaxBackward(cudnnHandle_t handle,
    cudnnSoftmaxAlgorithm_t alg,
    cudnnSoftmaxMode_t mode,
    void *alpha,
    cudnnTensorDescriptor_t xDesc,  void *x,
    cudnnTensorDescriptor_t dxDesc, void *dx,
    void *beta,
    cudnnTensorDescriptor_t dzDesc, void *dz)
    ```
    ```

SOFTMAX WITH OTHER LOSSES

- Backward pass
  - As with other backwards functions in cuDNN, this function computes the tensor dz = alpha[0] * $\nabla_z J(z)$ + beta[0] * dz
  - x is the output of the softmax function and dx is the derivative of our loss function J wrt x (cuDNN uses them internally)
  - Note that unlike backwards activations, we don't need a z input parameter (where z is the input to the softmax function)

SUMMARY

- Defining data in terms of tensors
- Using those tensors as arguments to cuDNN's built-in functions for both the forwards and backwards passes through a neural network
- You can find more details about everything we discussed in NVIDIA's official cuDNN developer guide
- Next week: convolutional neural nets