# 10-714 Homework 4

In this homework, you will leverage all of the components built in the last three homeworks to solve some modern problems with high performing network structures. We will start by adding a few new ops leveraging our new CPU/CUDA backends. Then, you will implement convolution, and a convolutional neural network to train a classifier on the CIFAR-10 image classification dataset. Then, you will implement recurrent and long-short term memory (LSTM) neural networks, and do word-level prediction language modeling on the Penn Treebank dataset.

As always, we will start by copying this notebook and getting the starting code. Reminder: **you must save a copy in drive**.

```
# Code to set up the assignment
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/
!mkdir -p 10714
%cd /content/drive/MyDrive/10714
!git clone https://github.com/dlsys10714/hw4.git
%cd /content/drive/MyDrive/10714/hw4

!pip3 install --upgrade --no-deps git+https://github.com/dlsys10714/mugrade.git
!pip3 install pybind11


!make


%set_env PYTHONPATH ./python
%set_env NEEDLE_BACKEND nd

    env: PYTHONPATH=./python
    env: NEEDLE_BACKEND=nd


import sys
sys.path.append('./python')


# Download the datasets you will be using for this assignment

import urllib.request
import os

!mkdir -p './data/ptb'
# Download Penn Treebank dataset
ptb_data = "https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb."
for f in ['train.txt', 'test.txt', 'valid.txt']:
    if not os.path.exists(os.path.join('./data/ptb', f)):
        urllib.request.urlretrieve(ptb_data + f, os.path.join('./data/ptb', f))

# Download CIFAR-10 dataset
if not os.path.isdir("./data/cifar-10-batches-py"):
    urllib.request.urlretrieve("https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz", "./data/cifar-10-python.tar.gz")
    !tar -xvzf './data/cifar-10-python.tar.gz' -C './data'
```

To finish setting up the assignment, go ahead and fill in all the code in `python/needle/autograd.py` using your solution code from the previous homework. Also copy the solutions in `src/ndarray_backend_cpu.cc` and `src/ndarray_backend_cuda.cu` from homework 3.

## Part 1: ND Backend [10 pts]

Recall that in homework 2, the `array_api` was imported as `numpy`. In this part, the goal is to write the necessary operations with `array_api` imported from the needle backend `NDArray` in `python/needle/backend_ndarray/ndarray.py`. Make sure to copy the solutions for `reshape`, `permute`, `broadcast_to` and `__getitem__` from homework 3.

Fill in the following classes in `python/needle/ops_logarithmic.py` and `python/needle/ops_mathematic.py`:

- `PowerScalar`
- `EWiseDiv`
- `DivScalar`
- `Transpose`
- `Reshape`
- `BroadcastTo`
- `Summation`
- `MatMul`
- `Negate`

- `Log`
- `Exp`
- `ReLU`
- `LogSumExp`
- `Tanh` (new)
- `Stack` (new)
- `Split` (new)

Note that for most of these, you already wrote the solutions in the previous homework and you should not change most part of your previous solution, if issues arise, please check if the `array_api` function used is supported in the needle backend.

`TanhOp`, `Stack`, and `Split` are newly added. `Stack` concatenates same-sized tensors along a new axis, and `Split` undoes this operation. The gradients of the two operations can be written in terms of each other. We do not directly test `Split`, and only test the backward pass of `Stack` (for which we assume you used `Split`).

**Note:** You may want to make your Summation op support sums over multiple axes; you will likely need it for the backward pass of the BroadcastTo op if yours supports broadcasting over multiple axes at a time. However, this is more about ease of use than necessity, and we leave this decision up to you (there are no corresponding tests).

**Note:** Depending on your implementations, you may want to ensure that you call `.compact()` before reshaping arrays. (If this is necessary, you will run into corresponding error messages later in the assignment.)

因为当前实现的 reshape() 中，默认当前数组是已经 compact 的

```
!python3 -m pytest -l -v -k "nd_backend"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "new_nd_backend"
```

## Part 2: CIFAR-10 dataset [10 points]

Next, you will write support for the CIFAR-10 image classification dataset, which consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. There are 50k training images and 10k test images.

Start by implementing the `__init__` function in the `CIFAR10Dataset` class in `python/needle/data/datasets/cifar10_dataset.py`. You can read in the link above how to properly read the CIFAR-10 dataset files you downloaded at the beginning of the homework. Also fill in `__getitem__` and `__len__`. Note that the return shape of the data from `__getitem__` should be in order (3, 32, 32).

Copy `python/needle/data/data_transforms.py` and `python/needle/data/data_basic.py` from previous homeworks.

```
!python3 -m pytest -l -v -k "test_cifar10"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "cifar10"
```

## Part 3: Convolutional neural network [40 points]

Here's an outline of what you will do in this task.

In `python/needle/backend_ndarray/ndarray.py`, implement:

- `flip`
- `pad`

In `python/needle/ops_mathematic.py`, implement (forward and backward):

- `Flip`
- `Dilate`
- `UnDilate`
- `Conv`

In `python/needle/nn/nn_conv.py`, implement:

- `Conv`

In `apps/models.py`, fill in the `ResNet9` class.

In `apps/simple_ml.py`, fill in:

- `epoch_general_cifar10`,
- `train_cifar10`
- `evaluate_cifar10`

We have provided a `BatchNorm2d` implementation in `python/needle/nn/nn_basic.py` for you as a wrapper around your previous `BatchNorm1d` implementation.

**Note**: Remember to copy the solution of `nn_basic.py` from previous homework, make sure to not overwrite the `BatchNorm2d` module.

## Padding ndarrays

Convolution as typically implemented in deep learning libraries cuts down the size of inputs; e.g., a (1, 32, 32, 3) image convolved with a 3x3 filter would give a (1, 30, 30, c) output. A way around this is to pad the input ndarray before performing convolution, e.g., pad with zeros to get a (1, 34, 34, 3) ndarray so that the result is (1, 32, 32, 3).

Padding is also required for the backward pass of convolution.

You should implement `pad` in `ndarray.py` to closely reflect the behavior of `np.pad`. That is, `pad` should take a tuple of 2-tuples with length equal to the number of dimensions of the array, where each element in the 2-tuple corresponds to "left padding" and "right padding", respectively.

For example, if `A` is a (10, 32, 32, 8) ndarray (think NHWC), then `A.pad( (0, 0), (2, 2), (2, 2), (0, 0) )` would be a (10, 36, 36, 8) ndarray where the "spatial" dimension has been padded by two zeros on all sides.

```
!python3 -m pytest -l -v -k "pad_forward"
```

---

## Flipping ndarrays & FlipOp

```
import numpy as np
import ctypes
```

Some utility code for a demonstration below which you can probably ignore. It might be instructive to check out the `offset` function.

```
# reads off the underlying data array in order (i.e., offset 0, offset 1, ..., offset n)
# i.e., ignoring strides
def raw_data(X):
    X = np.array(X) # copy, thus compact X
    return np.frombuffer(ctypes.string_at(X.ctypes.data, X.nbytes), dtype=X.dtype, count=X.size)

# Xold and Xnew should reference the same underlying data
def offset(Xold, Xnew):
    assert Xold.itemsize == Xnew.itemsize
    # compare addresses to the beginning of the arrays
    return (Xnew.ctypes.data - Xold.ctypes.data)//Xnew.itemsize

def strides(X):
    return ', '.join([str(x//X.itemsize) for x in X.strides])

def format_array(X, shape):
    assert len(shape) == 3, "I only made this formatting work for ndims = 3"
    def chunks(l, n):
        n = max(1, n)
        return (l[i:i+n] for i in range(0, len(l), n))
    a = [str(x) if x >= 10 else ' ' + str(x) for x in X]
    a = ['(' + ' '.join(y) + ')' for y in [x for x in chunks(a, shape[-1])]]
    a = ['|' + ' '.join(y) + '|' for y in [x for x in chunks(a, shape[-2])]]
    return '  '.join(a)

def inspect_array(X, *, is_a_copy_of):
    # compacts X, then reads it off in order
    print('Data: %s' % format_array(raw_data(X), X.shape))
    # compares address of X to copy_of, thus finding X's offset
    print('Offset: %s' % offset(is_a_copy_of, X))
    print('Strides: %s' % strides(X))
```

In order to implement the backwards pass of 2D convolution, we will (probably) need a function which *flips* axes of ndarrays. We say "probably" because you could probably cleverly implement your convolution forward function to avoid this. However, we think it is easiest to think about this if you have the ability to "flip" the kernel along its vertical and horizontal dimensions.

We will try to build up your intuition for the "flip" operation below in order to help you figure out how to implement it in `ndarray.py`. To do that, we explore numpy's `np.flip` function below. One thing to note is that `flip` is typically implemented by using negative strides and changing the *offset* of the underlying array.

For example, flipping an array on *all* of its axes is equivalent to reversing the array. In this case, you can imagine that we would want all the strides to be negative, and the offset to be the length of the array (to start at the end of the array and "stride" backwards).

Since we did not explicitly support negative strides in our implementation for the last homework, we will merely call `NDArray.make` with them to make our "flipped" array and then immediately call `.compact()`. Other than changing unsigned ints to signed ints in a few places, we suspect your existing `compact` function should not have to change at all to accomodate negative strides. In the .cc and .cu files we distributed, we have already changed the function signatures to reflect this.

Alternatively, you could simply implement `flip` in the CPU backend by copying memory, which you *may* find more intuitive. We suggest following our mini tutorial below to keep your implementation Python-focused, since we believe it is involves approximately the same amount of effort to implement it slightly more naively in C.

Use this array as reference for the other examples:

```
A = np.arange(1, 25).reshape(3, 2, 4)
inspect_array(A, is_a_copy_of=A)
```

We have put brackets around each axis of the array. Notice that for this array, the offset is 0 and the strides are all positive.

---

See what happens when you flip the array along the last axis below. Note that the `inspect_array` function compacts the array after flipping it so you can see the "logical" order of the data, and the offset is calculated by comparing the address of the **non**-compacted flipped array with that of `is_copy_of`, i.e., the array `A` we looked at above.

That is, we are looking at how numpy calculates the strides and offset for flipped arrays in order to copy this behavior in our own implementation.

```
inspect_array(np.flip(A, (2,)), is_a_copy_of=A)
```

So flipping the last axis reverses the order of the elements within each 4-dimensional "cell", as you can see above. The stride corresponding to the axis we flipped has been negated. And the offset is 3 -- this makes sense, e.g., because we want the new "first" element of the array to be 4, which was at index 3 in `A`.

```
inspect_array(np.flip(A, (1,)), is_a_copy_of=A)
```

Again for the middle axis: we negate the middle stride, and the offset is 4, which seems reasonable since we now want the first element to be 5, which was at index 4 in the original array `A`.

```
inspect_array(np.flip(A, (0,)), is_a_copy_of=A)
```

Try to infer the more general algorithm for computing the offset given the axis to flip.

---

Observe what happens when we flip *all* axes.

```
inspect_array(np.flip(A, (0,1,2)), is_a_copy_of=A)
```

As mentioned earlier, the offset is then sufficient to point to the last element of the array, and this is just the "reverse order" version of `A`.

When we flip just axes 1 and 0...

```
inspect_array(np.flip(A, (0,1)), is_a_copy_of=A)
```

The offset is 20. Looking back on our previous offset computations, do you notice something?

---

With this exploration of numpy's ndarray flipping functionality, which uses negative strides and a custom offset, try to implement `flip` in `ndarray.py`. You also must implement "flip" forward and backward functions in `ops.py`; note that these should be extremely short.

**Important:** You should call NDArray.make with the new strides and offset, and then immediately `.compact()` this array. The resulting array is then copied and has positive strides. We want this (less-than-optimal) behavior because we did not account for negative strides in our previous implementation. *Aside:* If you want, consider where/if negative strides break your implementation. `__getitem__` definitely doesn't work due to how we processed slices; is there anything else? (*Note*: this isn't graded.)

Also, if you want to instead add a `flip` operator on the CPU/CUDA backends, that's also okay.

```
!python3 -m pytest -l -v -k "flip"
```

---

## Dilation

The dilation operator puts zeros between elements of an ndarray. We will need it for computing the backward pass of convolution when the stride of the convolution is greater than 1. As an example, dilation should do the following to a 2x2 matrix when dilated by 1 on both axes:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

To get some intuition for why we need dilation for the backward pass of strided convolution, consider a `stride=2, padding="same", input_channels=output_channels=8` convolution applied to an input of size (10, 32, 32, 8). The resulting output will be of size (10, 16, 16, 8) due to the stride, and thus `out_grad` will have shape (10, 16, 16, 8). Yet, the gradient of the input needs to, of course, have shape (10, 32, 32, 8) -- so we must need to increase the size of `out_grad` in some way. Consider also that you could implement strided convolution as `Conv(x)[:, ::2, ::2, :]`, i.e., only keeping every other pixel in the spatial dimension.

Implement `Dilate` in `ops.py`. This function takes two additional parameters (in attrs): the `dilation` amount and the `axes` to dilate. You must also implement the corresponding op `UnDilate`, whose forward pass will be used to implement the gradient of `Dilate`. (This is so we do not have to implement `GetItem` and `SetItem` ops, which can be highly inefficient to backprop through without additional optimizations.)

```
!python3 -m pytest -l -v -k "dilate"
```

---

## Submit new ops (flip/dilation) to mugrade [10 points]

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "new_ops"
```

---

## Convolution forward

Implement the forward pass of 2D multi-channel convolution in `ops.py`. You should probably refer to this notebook from lecture, which implements 2D multi-channel convolution using im2col in numpy.

**Note:** Your convolution op should accept tensors in the NHWC format, as in the example above, and weights in the format (kernel_size, kernel_size, input_channels, output_channels).

However, you will need to add two additional features. Your convolution function should accept arguments for `padding` (default 0) and `stride` (default 1). For `padding`, you should simply apply your padding function to the spatial dimensions (i.e., axes 1 and 2).

Implementing strided convolution should consist of a relatively small set of changes to your plain convolution implementation.

We recommend implementing convolution without stride first, ensuring you pass some of the tests below, and then adding in stride.

```
!python3 -m pytest -l -v -k "op_conv and forward"
```

---

## Convolution backward

Finding the gradients of 2D multi-channel convolution can be technically quite challenging (especially "rigorously"). We will try to provide some useful hints here. Basically, we encourage you to make use of the surprising fact that *whatever makes the dimensions work out is typically right*.

Ultimately, the backward pass of convolution can be done in terms of the convolution operator itself, with some clever manipulations using `flip`, `dilate`, and multiple applications of `transpose` to both the arguments and the results.

In the last section, we essentially implemented convolution as a matrix product: ignoring the various restride and reshape operations, we basically have something like `X @ W`, where `X` is the input and `W` is the weight. We also have `out_grad`, which is the same shape as `X @ W`. Now, you have already implemented the backward pass of matrix multiplication in a previous assignment, and we can use this knowledge to get some insight into the backward pass of convolution. In particular, referencing your matmul backward implementation, you may notice (heuristically speaking here):

```
X.grad = out_grad @ W.transpose
W.grad = X.transpose @ out_grad
```

Surprisingly enough, things work out if we just assume that these are also convolutions (and now assuming that `out_grad`, `W`, and `X` are tensors amenable to 2D multi-channel convolution instead of matrices):

```
X.grad = ≈conv(≈out_grad, ≈W)
W.grad = ≈conv(≈X, ≈out_grad)
```

In which the "≈" indicates that you need to apply some additional operators to these terms in order to get the dimensions to work out, such as permuting/transposing axes, dilating, changing the `padding=` argument to the convolution function, or permuting/transposing axes of the resulting convolution.

As we saw on the last few slides here in class, the transpose of a convolution can be found by simply flipping the kernel. Since we're working in 2D instead of 1D, this means flipping the kernel both vertically and horizontally (thus why we implemented `flip`).

Summarizing some hints for both `X.grad` and `W.grad`:

`X.grad`

- The convolution of `out_grad` and `W`, with some operations applied to those
- `W` should be flipped over both the kernel dimensions
- If the convolution is strided, increase the size of `out_grad` with a corresponding dilation
- Do an example to analyze dimensions: note the shape you want for `X.grad`, and think about how you must permute/transpose the arguments and add padding to the convolution to achieve this shape
    - This padding depends on both the kernel size and the `padding` argument to the convolution

`W.grad`

- The convolution of `X` and `out_grad`, with some operations applied to those
- The gradients of `W` must be accumulated over the batches; how can you make the conv operator itself do this accumulation?
    - Consider turning batches into channels via transpose/permute
- Analyze dimensions: how can you modify `X` and `out_grad` so that the shape of their convolution matches the shape of `W`? You may need to transpose/permute the result.
    - Remember to account for the `padding` argument passed to convolution

General tips

- Deal with strided convolutions last (you should be able to just drop in `dilate` when you've passed most of the tests)
- Start with the case where `padding=0`, then consider changing `padding` arguments
- You can "permute" axes with multiple calls to `transpose`

It might also be useful to skip ahead to nn.Conv, pass the forward tests, and then use both the tests below and the nn.Conv backward tests to debug your implementation.

```
!python3 -m pytest -l -v -k "op_conv and backward"
```

---

## nn.Conv

### Fixing init._calculate_fans for convolution

Previously, we have implemented Kaiming uniform/normal initializations, where we essentially assigned `fan_in = input_size` and `fan_out = output_size`. For convolution, this becomes somewhat more detailed, in that you should multiply both of these by the "receptive field size", which is in this case just the product of the kernel sizes -- which in our case are always going to be the same, i.e., $k \times k$ kernels.

**You will need to edit your `kaiming_uniform` in `python/needle/init/init_initializers.py`, etc. init functions to support multidimensional arrays.** In particular, it should support a new `shape` argument which is then passed to, e.g., the underlying `rand` function. Specifically, if the argument `shape` is not None, then ignore `fan_in` and `fan_out` but use the value of `shape` for initializations.

You can test this below; though it is not *directly* graded, it must match ours to pass the nn.Conv mugrade tests.

```
!python3 -m pytest -l -v -k "kaiming_uniform"
```

## Implementing nn.Conv

Essentially, nn.Conv is just a wrapper of the convolution operator we previously implemented which adds a bias term, initializes the weight and bias, and ensures that the padding is set so that the input and output dimensions are the same (in the `stride=1` case, anyways).

Importantly, nn.Conv should support NCHW format instead of NHWC format. In particular, we think this makes more sense given our current BatchNorm implementation. You can implement this by applying `transpose` twice to both the input and output.

- Ensure nn.Conv works for (N, C, H, W) tensors even though we implemented the conv op for (N, H, W, C) tensors
- Initialize the (k, k, i, o) weight tensor using Kaiming uniform initialization with default settings
- Initialize the (o,) bias tensor using uniform initialization on the interval $\pm$ `1.0/(in_channels * kernel_size**2)**0.5`
- Calculate the appropriate padding to ensure input and output dimensions are the same
- Calculate the convolution, then add the properly-broadcasted bias term if present

You can now test your nn.Conv against PyTorch's nn.Conv2d with the two PyTest calls below.

```
!python3 -m pytest -l -v -k "nn_conv_forward"
```

```
!python3 -m pytest -l -v -k "nn_conv_backward"
```

---

## Submit nn.Conv to mugrade [20 points]

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "conv_forward"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "conv_backward"
```

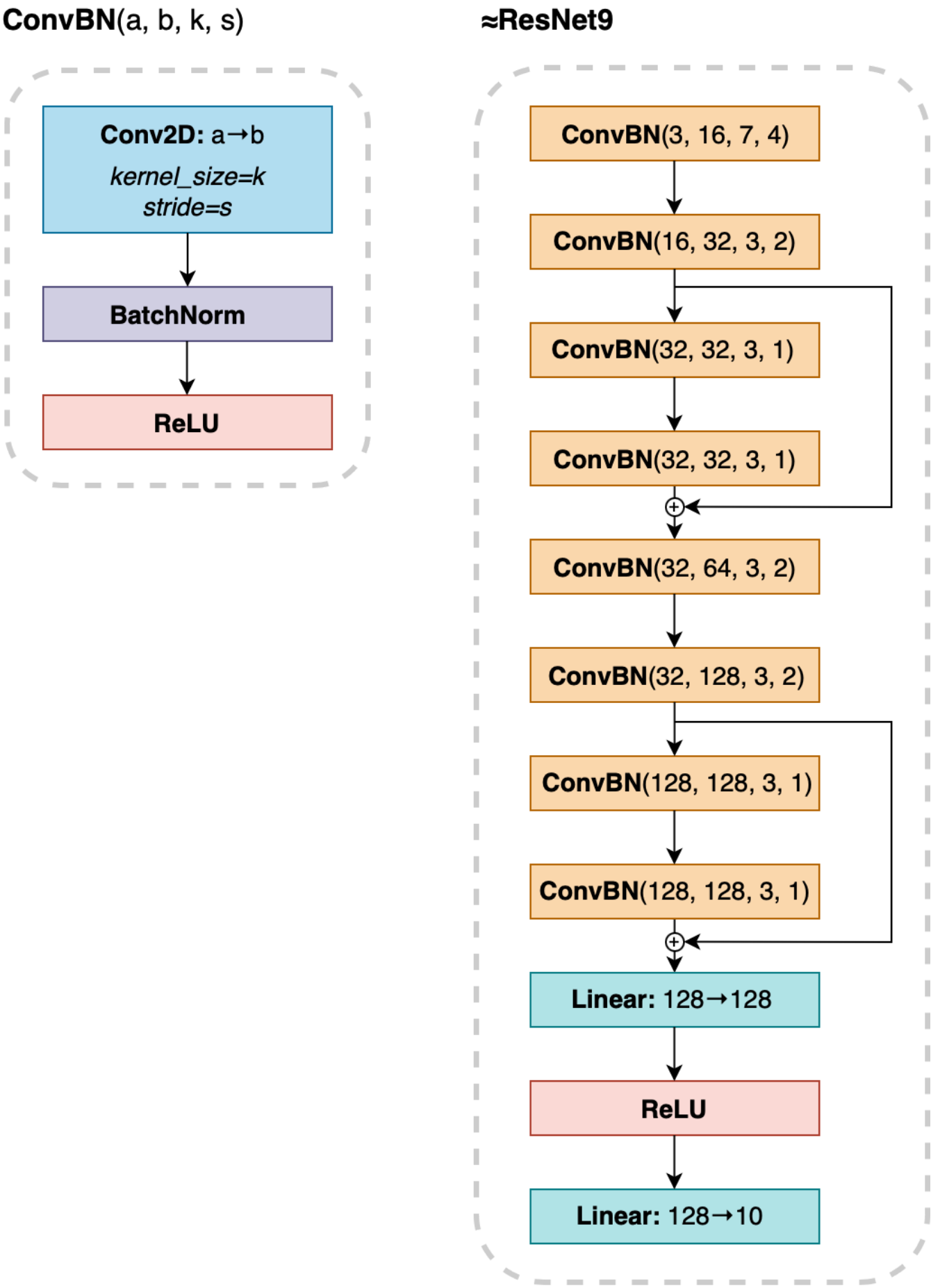
---

## Implementing "ResNet9"

You will now use your convolutional layer to implement a model similar to *ResNet9*, which is known to be a reasonable model for getting good accuracy on CIFAR-10 quickly (see here). Our main change is that we used striding instead of pooling and divided all of the channels by 4 for the sake of performance (as our framework is not as well-optimized as industry-grade frameworks).

In the figure below, before the linear layer, you should "flatten" the tensor. You can use the module `Flatten` in `nn_basic.py`, or you can simply use `.reshape` in the `forward()` method of your ResNet9.

Make sure that you pass the device to all modules in your model; otherwise, you will get errors about mismatched devices when trying to run with CUDA.




We have tried to make it easier to pass the tests here than for previous assignments where you have implemented models. In particular, we are just going to make sure it has the right number of parameters and similar accuracy and loss after 1 or 2 batches of CIFAR-10.

```
!python3 -m pytest -l -v -k "resnet9"
```

Now we can train a ResNet on CIFAR10: (remember to copy the solutions in `python/needle/optim.py` from previous homeworks)

```
!python3 -m pytest -l -v -k "train_cifar10"
```

## Submit ResNet9 to mugrade [10 points]

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "resnet9"
```

---

Now, you can train your model on CIFAR-10 using the following code. Note that this is likely going to be quite slow, and also not all that accurate due to the lack of data augmentation. You should expect it to take around 500s per epoch.

```
import sys
sys.path.append('./python')
sys.path.append('./apps')
import needle as ndl
from models import ResNet9
from simple_ml import train_cifar10, evaluate_cifar10

device = ndl.cpu()
dataset = ndl.data.CIFAR10Dataset("data/cifar-10-batches-py", train=True)
dataloader = ndl.data.DataLoader(\
         dataset=dataset,
         batch_size=128,
         shuffle=True,)
model = ResNet9(device=device, dtype="float32")
train_cifar10(model, dataloader, n_epochs=10, optimizer=ndl.optim.Adam,
      lr=0.001, weight_decay=0.001)
evaluate_cifar10(model, dataloader)
```

## Part 4: Recurrent neural network [10 points]

**Note:** In the following sections, you may find yourself wanting to index into tensors, i.e., to use getitem or setitem. However, we have not implemented these for tensors in our library; instead, you should use `stack` and `split` operations.

In `python/needle/nn_sequence.py`, implement `RNNCell`.

$h' = \tanh(xW_{ih} + b_{ih} + hW_{hh} + b_{hh})$. If nonlinearity is 'relu', then ReLU is used in place of tanh.

All weights and biases should be initialized from $\square(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$.

In `python/needle/nn_sequence.py`, implement `RNN`.

For each element in the input sequence, each layer computes the following function:

$h_t = \tanh(x_t W_{ih} + b_{ih} + h_{(t-1)} W_{hh} + b_{hh})$

where $h_t$ is the hidden state at time $t$, $x_t$ is the input at time $t$, and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time $0$. If nonlinearity is 'relu', then ReLU is used in place of tanh.

In a multi-layer RNN, the input $x_t^{(l)}$ of the $l$-th layer ($l \geq 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer.

```
!python3 -m pytest -l -v -k "test_rnn"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "rnn"
```

## Part 5: Long short-term memory network [10 points]

In `python/needle/nn/nn_sequence.py`, implement `Sigmoid`.

$\sigma(x) = \frac{1}{1+\exp(-x)}$

In `python/needle/nn/nn_sequence.py`, implement `LSTMCell`.

$$
\begin{aligned}
i &= \sigma(xW_{ii} + b_{ii} + hW_{hi} + b_{hi}) \\
f &= \sigma(xW_{if} + b_{if} + hW_{hf} + b_{hf}) \\
g &= \tanh(xW_{ig} + b_{ig} + hW_{hg} + b_{hg}) \\
o &= \sigma(xW_{io} + b_{io} + hW_{ho} + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * \tanh(c')
\end{aligned}
$$

where $\sigma$ is the sigmoid function, and $i, f, g, o$ are the input, forget, cell, and output gates, respectively.

All weights and biases should be initialized from $\square(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$.

Now implement `LSTM` in `python/needle/nn/nn_sequence.py`, which applies a multi-layer LSTM RNN to an input sequence. For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(x_t W_{ii} + b_{ii} + h_{(t-1)} W_{hi} + b_{hi}) \\
f_t &= \sigma(x_t W_{if} + b_{if} + h_{(t-1)} W_{hf} + b_{hf}) \\
g_t &= \tanh(x_t W_{ig} + b_{ig} + h_{(t-1)} W_{hg} + b_{hg}) \\
o_t &= \sigma(x_t W_{io} + b_{io} + h_{(t-1)} W_{ho} + b_{ho}) \\
c_t &= f * c_{(t-1)} + i * g \\
h_t &= o * \tanh(c_t)
\end{aligned}
$$

, where $h_t$ is the hidden state at time $t$, $c_t$ is the cell state at time $t$, $x_t$ is the input at time $t$, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time $0$, and $i_t, f_t, g_t, o_t$ are the input, forget, cell, and output gates at time $t$ respectively.

In a multi-layer LSTM, the input $x_t^{(l)}$ of the $l$-th layer ($l \geq 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer.

```
!python3 -m pytest -l -v -k "test_lstm"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "lstm"
```

## Part 6: Penn Treebank dataset [10 points]

In word-level language modeling tasks, the model predicts the probability of the next word in the sequence, based on the words already observed in the sequence. You will write support for the Penn Treebank dataset, which consists of stories from the Wall Street Journal, to train and evaluate a language model on word-level prediction.

In `python/needle/data/datasets/ptb_dataset.py`, start by implementing the `Dictionary` class, which creates a dictionary from a list of words, mapping each word to a unique integer.

Next, we will use this `Dictionary` class to create a corpus from the train and test txt files in the Penn Treebank dataset that you downloaded at the beginning of the notebook. Implement the `tokenize` function in the `Corpus` class to do this.

In order to prepare the data for training and evaluation, you will next implement the `batchify` function. Starting from sequential data, batchify arranges the dataset into columns. For instance, with the alphabet as the sequence and batch size 4, we'd get

```
┌ a g m s ┐
│ b h n t │
│ c i o u │
│ d j p v │
│ e k q w │
└ f l r x ┘
```

These columns are treated as independent by the model, which means that the dependence of e. g. 'g' on 'f' cannot be learned, but allows more efficient batch processing.

Next, implement the `get_batch` function. `get_batch` subdivides the source data into chunks of length `bptt`. If source is equal to the example output of the batchify function, with a bptt-limit of 2, we'd get the following two Variables for i = 0:

```
┌ a g m s ┐ ┌ b h n t ┐
└ b h n t ┘ └ c i o u ┘
```

Note that despite the name of the function, the subdivison of data is not done along the batch dimension (i.e. dimension 1), since that was handled by the batchify function. The chunks are along dimension 0, corresponding to the seq_len dimension in the LSTM or RNN.

```
!python3 -m pytest -l -v -k "ptb"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "ptb"
```

## Part 7: Training a word-level language model [10 points]

Finally, you will use the `RNN` and `LSTM` components you have written to construct a language model that we will train on the Penn Treebank dataset.

First, in `python/needle/nn/nn_sequence.py` implement `Embedding`. Consider we have a dictionary with 1000 words. Then for a word which indexes into this dictionary, we can represent this word as a one-hot vector of size 1000, and then use a linear layer to project this to a vector of

some embedding size.

In `apps/models.py`, you can now implement `LanguageModel`. Your language model should consist of

- An embedding layer (which maps word IDs to embeddings)
- A sequence model (either RNN or LSTM)
- A linear layer (which outputs probabilities of the next word)

In `apps/simple_ml.py` implement `epoch_general_ptb`, `train_ptb`, and `evaluate_ptb`.

```
!python3 -m pytest -l -v -k "language_model_implementation"
```

```
!python3 -m pytest -l -v -k "language_model_training"
```

```
!python3 -m mugrade submit "YOUR KEY HERE" -k "language_model"
```

Now, you can train your language model on the Penn Treebank dataset:

```
import needle as ndl
```