Machine Learning	**Assignment 5**
Comp540 Spring 2019	**due 27 March 2019 at 8 pm**

This homework is due March 27 at 8 pm on Canvas. The code base `hw5.zip` *for the assignment is an attachment to Assignment 5 on Canvas. You will add your code at the indicated spots in the files there. Place your answers to Problems 1, 2 and 3 (typeset) in a file called* `writeup.pdf`*. Please follow the usual submission instructions. When you submit, please submit the following items as separate attachments before the due date and time:*

- *Your writeup pdf, named* `writeup.pdf`*.*
- *Your jupyter notebooks, saved in HTML format. If there are multiple notebooks, submit each one separately.*
- *The* `.py` *files that you modified, each one as a separate upload.*
- *The zip file containing your work (notebooks + .py). Be sure that the datasets you use are NOT part of the zip, as this will increase the size considerably.*

You will solve this assignment in a group of two – only one submission per group, please.

We take the Rice Honor Code very seriously in this class. Please read the academic integrity section of the course policies at ***https://canvas.rice.edu/courses/20448/pages/comp-540-statistical-machine-learning-course-policies***, *and enter the statement that you have followed the honor code described there in the submission box for your homework. Homeworks without the honor pledge will not be graded.*

1 Deep neural networks (10 extra credit points) (20 points)

These are a set of short answer questions to help you understand concepts in deep learning. You are free to consult our deep learning text book by Goodfellow et. al. as well as original papers on arXiv to answer these questions. Please cite the resources you used to construct your answers.

- (5 points) Why do deep networks typically outperform shallow networks?
- (2 points) What is leaky ReLU activation and why is it used?
- (3 points) Why are pooling layers included in networks for object recognition?
- (10 points) In a paragraph or so, and using sketches as appropriate, contrast: AlexNet, VGG-Net, GoogleNet and ResNet. What is the one defining characteristic of each network?
- Extra credit: (10 points) Imagine a learning task on images which have cylindrical symmetry. That is, they wrap around in the east-to-west direction. Describe how you can modify the convolution operation to respect this symmetry.

2 Decision trees, entropy and information gain (10 points)

- (2 points) The entropy of a Bernoulli random variable X with $P(X = 1) = q$ is given by
$$H(p) = -q\ log(q) - (1 - q)\ log(1 - q)$$
where logarithms are taken in base 2. Suppose that a set S of examples contains p positive examples and n negative examples. The entropy $H(S)$ of the set S is defined as $H(p/(p + n))$. Show that $H(S) \leq 1$ and that $H(S) = 1$ when $p = n$.

- (5 points) Consider a data set comprising 400 data points from class C_1 and 400 data points from class C_2. Suppose that a decision stump model A splits these into two leaves at the root node; one containing (300,100) and the other containing (100,300) where (n, m) denotes n points are from class C_1 and m points are from class C_2. Similarly a second decision stump model model B splits the examples as (200,400) and (200,0). Calculate the reduction in cost using misclassification rate, entropy, and Gini index for models A and B. Which is the preferred split (model A or model B) according to these cost calculations?

- (3 points) Can the misclassification rate ever increase when splitting on a feature? If so, give an example. If not, give a proof.

3 Bagging (10 points)

Consider a regression problem where we wish to learn a function $f(x)$, where $x \in \Re^d$. Suppose we learn L functions $h_1(x), \ldots, h_L(x)$. The predictions of each of these functions can be expressed as the sum of the true function plus an error term.

$$h_l(x) = f(x) + \epsilon_l(x)$$

where $\epsilon_l(x)$ is drawn from $N(0, \sigma_l^2)$. The expected squared-error of the function $h_l(x)$ is

$$E_X[\{f(x) + \epsilon_l(x) - f(x)\}^2] = E_X[\epsilon_l(x)^2]$$

The averaged error over the entire ensemble is

$$E_{av} = \frac{1}{L}\sum_{l=1}^{L} E_X[\epsilon_l(x)^2]$$

The prediction made by the bagged ensemble is the average over the L individual predictors:

$$h_{bag}(x) = \frac{1}{L}\sum_{l=1}^{L} h_l(x)$$

The error made by the bagged ensemble is:

$$\epsilon_{bag}(x) = h_{bag}(x) - f(x) = \frac{1}{L}\sum_{l=1}^{L} h_l(x) - f(x) = \left(\frac{1}{L}\sum_{l=1}^{L}(f(x) + \epsilon_l(x))\right) - f(x)$$

The expected squared-error of the bagged ensemble is:

$$E_{bag} = E_X[\epsilon_{bag}(x)^2]$$

- (5 points) Assuming that the individual errors $\epsilon_l(x)$ have zero mean and are uncorrelated, that is $E_X[\epsilon_l(x)] = 0$ and $E_X[\epsilon_m(x)\epsilon_l(x)] = 0$ for $m \neq l$, show that

$$E_{bag} = \frac{1}{L}E_{av}$$

- (5 points) In practice, however, the errors may be highly correlated. Nevertheless, using Jensen's inequality for the special case of the convex function $f(x) = x^2$, show that the average expected squared-error E_{av} of the individual functions and the expected error of bagging E_{bag} satisfy $E_{bag} \leq E_{av}$, without any assumptions on $\epsilon_l(x)$. Jensen's equality states that for any convex function $f(x)$, $\lambda_l \geq 0$ and $(\sum_{l=1}^{L} \lambda_l) = 1$,

$$f(\sum_{l=1}^{L} \lambda_l x_l) \leq \sum_{l=1}^{L} \lambda_l f(x_l)$$

4 Fully connected neural networks and convolutional neural networks (20 extra credit points) (130 points)

In this exercise, you will first develop a modular, fully connected neural net model and test it on the CIFAR-10 problem. Then, you will build a three layer convolutional neural net model and test it on the same CIFAR-10 problem. The purpose of this exercise is to give you a working understanding of neural net models and give you experience in tuning their (many) hyper-parameters. Download `hw5.zip` from Canvas and unpack its contents. You will see the files as described in Table 1. A good reference for material on fully connected neural networks is the online text by Goodfellow et. al. available at `http://www.deeplearningbook.org` as well as the textbook by Michael Nielsen available at `http://neuralnetworksanddeeplearning.com/`.

Setting up the CIFAR-10 data

Before you begin the assignment, edit `data_utils.py`, providing it the path to CIFAR-10 data that we downloaded for Assignment 3 (to be specific, the directory `datasets` in that code base). Make sure you have the actual dataset there; else you might have to run the shell script I provided then to re-download it.

Name	Read?	Edit?	Description
FullyConnectedNets.ipynb	Yes	Yes	Notebook to run your fully connected networks.
ConvolutionalNetworks.ipynb	Yes	Yes	Notebook to run your CNN functions.
fc_net.py	Yes	Yes	API for two-layer fully connected networks and networks of arbitrary depth.
cnn.py	Yes	Yes	API for three-layer CNN networks.
layers.py	Yes	Yes	Forward and backward functions for every layer type.
optim.py	Yes	Yes	Implementations of specific gradient descent rules.
layer_utils.py	Yes	No	Sandwich layer implementations.
data_utils.py	Yes	No	data loading utilities.
vis_utils.py	Yes	No	visualization utilities.
gradient_check.py	Yes	No	functions for checking gradients numerically.
fast_layers.py	Yes	No	fast versions of convolutional operations.
solver.py	Yes	No	gradient descent solver.
setup.py	No	No	code to setup fast layers implementation.
im2col.py	No	No	support for fast implementations of convolutional operations.
im2col_cython.c	No	No	support for fast implementations of convolutional operations.

Table 1: Code base for Assignment 5: fully connected neural networks

Fully connected feedforward neural networks: a modular approach

We will implement fully-connected networks using a modular approach. By this we mean that we will develop a backward and a forward function for each type of layer (affine, ReLU, fully-connected). The forward function will receive inputs, weights, and other parameters and will return both an output and a cache object storing data needed for the backward pass,

```
def layer_forward(x, theta):
  """ Receive inputs x and weights theta """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, theta, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the cache object computed during the forward pass, and will return gradients with respect to the inputs and weights,

```
def layer_backward(dout, cache):
  """
  Receive derivative of loss with respect to outputs and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, theta, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dtheta = # Derivative of loss with respect to theta

  return dx, dtheta
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures. In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization especially suited for deep networks.

Problem 4.1.1: Affine layer: forward (5 points)

An affine layer computes a linear function of inputs in a fully connected network. An affine layer has h outputs, d inputs, and connection weights θ of size $d \times h$ connecting the outputs and inputs, as well as a bias weight vector θ_0 of size h. The forward function computes output a_j for every $j \in \{1, \ldots, h\}$ for an input $x \in \Re^d$ by computing

$$a_j = \theta_j^T x + \theta_{0j}$$

where θ_j is the j^{th} column of the θ matrix and corresponds to the connections between the inputs and output unit j, and θ_{0j} is the j^{th} component of the bias vector θ_0. To accommodate inputs shaped as a volume for convolutional networks, the function accepts $x \in \Re^{d_1 \times \ldots d_n}$ by first reshaping it into a vector of size d equal to the product of the d_i's.

In the file `layers.py` implement the `affine_forward` function. You have to vectorize the computation described in the equation above to handle an `X` matrix of size $m \times d$. Once you are done, you should test your implementation of `affine_forward` by running the appropriate cell in `FullyConnectedNets.ipynb`.

Problem 4.1.2: Affine layer: backward (5 points)

The `affine_backward` function propagates derivatives from the outputs back to the inputs of an affine layer. Let the error derivative vector at the output of the layer be $\frac{\partial J}{\partial a_j}$. We now need to calculate $\frac{\partial J}{\partial x}$, $\frac{\partial J}{\partial \theta_j}$ and $\frac{\partial J}{\theta_{0j}}$. We know that

$$a_j = \theta_j^T x + \theta_{0j}$$

Use the chain rule of derivatives together with the linear relationship between a_j and x, θ_j, θ_{0j} to derive the formulas for these three partial derivatives. Hint: $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial x}$.

In the file `layers.py` implement the `affine_backward` function. You have to vectorize the computation for an `X` matrix of size $m \times d$ and a derivative `dout` of size $m \times h$. Once you are done, you should test your implementation of `affine_backward` by running the appropriate cell in `FullyConnectedNets.ipynb`.

Problem 4.1.3: ReLU layer: forward (2 points)

A ReLU layer constitutes a non-linear layer in a fully-connected network. It takes input $x \in \Re^d$ and produces output $a \in \Re^d$ such that

$$a_j = \begin{cases} x_j & \text{if } x_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

In the file `layers.py` implement the vectorized form of the `relu_forward` function for `X` of size $m \times d$. Once you are done, you should test your implementation of `relu_forward` by running the appropriate cell in `FullyConnectedNets.ipynb`.

Problem 4.1.4: ReLU layer: backward (3 points)

The `relu_backward` function propagates derivatives from the outputs back to the inputs of an ReLU layer. Let the error derivative vector at the output of the layer be $\frac{\partial J}{\partial a_j}$. We now need to calculate $\frac{\partial J}{\partial x}$ for the ReLU layer. As before, use the chain rule to compute it.

In the file `layers.py` implement the `relu_backward` function. Once you are done, you should test your implementation of `relu_backward` by running the appropriate cell in `FullyConnectedNets.ipynb`.

Sandwich layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `layer_utils.py`. For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions. The notebook `FullyConnectedNets.ipynb` will numerically gradient check the backward pass for the affine-relu layer.

Loss layers: softmax and SVM

You implemented these loss functions in the last assignment, so we will give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `layers.py`. The notebook `FullyConnectedNets.ipynb` will make sure that the implementations are correct.

Problem 4.1.5: Two layer network (5 points)

Now that you have built modular versions of the necessary layers, you will implement a two layer fully connected network using these modular implementations. In particular, you will implement a two-layer fully-connected neural network with ReLU nonlinearity and softmax loss. The architectural layers are `affine - relu - affine - softmax`. Open the file `fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You will have to fill in code for `init` and `loss` for the `TwloLayerNet` class. A cell in `FullyConnectedNets.ipynb` will test your implementation, including your network initialization, forward and backward computations.

Problem 4.1.6: Overfitting a two layer network (5 points)

Open the file `solver.py` and read through it to familiarize yourself with the API for the `Solver` class. The documentation includes sample calls to create a `Solver` instance with default parameters that are a good starting point for this problem. Then, use a `Solver` instance at the indicated spot in the `FullyConnectedNets.ipynb` notebook to train a `TwoLayerNet` on the CIFAR-10 data. Select the size of the hidden layer and the regularization parameter to achieve at least 50% accuracy on the validation set.

Problem 4.1.7: Multilayer network (10 points)

Next you will implement a fully-connected network with an arbitrary number of hidden layers. Read through the `FullyConnectedNet` class in `fc_net.py`. Implement the initialization, the forward pass, and the backward pass. As a sanity check, we have code in `FullyConnectedNets.ipynb` to check the initial loss and gradients of the network both with and without regularization. Do the initial losses seem reasonable? For gradient checking, you should expect to see errors around 1e-6 or less.

Problem 4.1.8: Overfitting a three layer network (2 points)

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. You will need to tweak the learning rate and initialization scale in `FullyConnectedNets.ipynb` at the indicated point, and you should be able to achieve 100% training accuracy within 20 epochs.

Problem 4.1.9: Overfitting a five layer network (3 points)

Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, and you should be able to achieve 100% training accuracy within 20 epochs. Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net?

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

Problem 4.1.10: SGD+Momentum (5 points)

Open the file `optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the appropriate cell in `FullyConnectedNets.ipynb` to check your implementation. You should see errors less than 1e-8.

The SGD+momentum rule is parameterized by momentum μ and it maintains a velocity variable v updated as follows:

$$v \leftarrow \mu v - \alpha \frac{\partial J}{\partial \theta}$$

where α is the learning rate. Then, θ is updated as

$$\theta \leftarrow \theta + v$$

Then we train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

Problem 4.1.11: RMSProp (5 points)

RMSProp is an update rule that sets per-parameter learning rates by using a moving average of squared gradients. In the file `optim.py`, implement the RMSProp update rule in the `rmsprop` function check your implementations using the tests in `fully_connected_nets.py`. The RMSProp rule is parameterized by a decay rate γ, and an ϵ. It maintains a cache c of squared gradients which is updated as follows:

$$c \leftarrow \gamma c + (1-\gamma)\frac{\partial J}{\partial \theta} * \frac{\partial J}{\partial \theta}$$

Then, θ is updated as

$$\theta \leftarrow \theta - \alpha \frac{\partial J}{\partial \theta} \frac{1}{(\sqrt{c} + \epsilon)}$$

where the ϵ is added to the denominator to keep it from being zero.

Problem 4.1.12: Adam (5 points)

Adam is a smoothed version of RMSProp with momentum. The update rule for Adam looks just like the one for RMSProp, except the smooth version of the gradient `m` is used, rather than the raw gradient $\frac{\partial J}{\partial \theta}$. In addition, it exponentially decays the smoothed gradient and the square of the gradient using the hyperparameter `t`.

$$\begin{aligned} m &\leftarrow \beta_1 m + (1-\beta_1)\frac{\partial J}{\partial \theta} \\ v &\leftarrow \beta_2 v + (1-\beta_2)\frac{\partial J}{\partial \theta} * \frac{\partial J}{\partial \theta} \\ t &\leftarrow t+1 \\ mb &\leftarrow m/(1-{\beta_1}^t) \\ vb &\leftarrow v/(1-{\beta_2}^t) \\ \theta &\leftarrow \theta - \alpha\, mb/(\sqrt{vb} + \epsilon) \end{aligned}$$

The hyperparameter β_1 is typically chosen to be 0.9, β_2 to be 0.999 and $\epsilon = 10^{-8}$.

Dropout

Dropout is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and then use it to modify your fully-connected network to optionally use dropout. Dropout is described in detail in Geoffrey E. Hinton et al, ”Improving neural networks by preventing co-adaptation of feature detectors”, arXiv 2012. For this section you will use `Dropout.ipynb` to test the functions you implement.

Problem 4.2.1: Dropout forward pass (5 points)

In the file `layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes. While training, dropout is implemented by setting the activation of each unit in a hidden layer to zero with probability p (a hyperparameter). This means that a unit in a hidden layer is active with some probability $1 - p$. A naive implementation of this approach will require us to scale hidden unit outputs during testing by the factor of $1 - p$. Instead, we recommend the approach shown in pseudo-code below, called inverted drop out, in which we do the scaling during training time, so the predict function during testing remains the same.

```
#### forward pass for example 3-layer neural network
def train_step(X,p):
  drop = 1 - p
  H1 = np.maximum(0, np.dot(theta1, X) + theta1_0)
  U1 = (np.random.rand(*H1.shape) < drop) / drop # first dropout mask.
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(theta2, H1) + theta2_0)
  U2 = (np.random.rand(*H2.shape) < drop) / drop # second dropout mask.
  H2 *= U2 # now implement the drop
  out = np.dot(theta3, H2) + theta3_0
```

Once you have completed the function in `layers.py` which only needs you to consider only one layer, run the cell in `Dropout.ipynb` to test your implementation.

Problem 4.2.2: Dropout backward pass (5 points)

In the file `layers.py`, implement the backward pass for dropout. In training mode, use the mask cached from the forward pass to zero out the derivatives of the units that were inactive during the forward pass (hint: multiply the given derivative by the mask). In test mode, pass the derivatives through the layer unaltered. After doing so, run the appropriate cell in `Dropout.ipynb` to numerically gradient-check your implementation.

Problem 4.2.3: Fully connected nets with dropout (5 points)

In the file `fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a nonzero value for the dropout parameter, then the net should add a dropout layer immediately after every ReLU nonlinearity. After completing your implementation, run the appropriate cell in `Dropout.ipynb` to numerically gradient-check your implementation. You should see gradient errors of the order of 10^{-8} or lower.

Problem 4.2.4: Experimenting with fully connected nets with dropout (5 points)

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a dropout probability of 0.75. We will then visualize the training and validation accuracies of the two networks over time. You should see the network with dropout converge faster. Comment on the shape of the training and validation accuracy plots for networks trained with and without dropout,

Problem 4.3: Training a fully connected network for the CIFAR-10 dataset with dropout (10 points)

Use the `FullyConnectedNet` class that you have developed and the new gradient descent rules and dropout to train a model that achieves greater than 50% accuracy on the validation set of the full CIFAR-10 dataset. You will need to find the number of hidden layers, the number of units in each layer, the weight scale, the learning rule, the learning rate, batch size, number of training epochs, dropout probability, and batch size to achieve this training goal. Use the `show_net_weights` function to visualize the first level weights. Complete the cell in `FullyConnected.ipynb` for this exercise.

Convolutional neural networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead. First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

Convolutional layer

The convolutional layer is the core building block of a convolutional network, and its output volume can be interpreted as holding neurons arranged in a 3D volume. The convolutional layer's parameters consist of a set of learnable filters. Every filter is small spatially (along

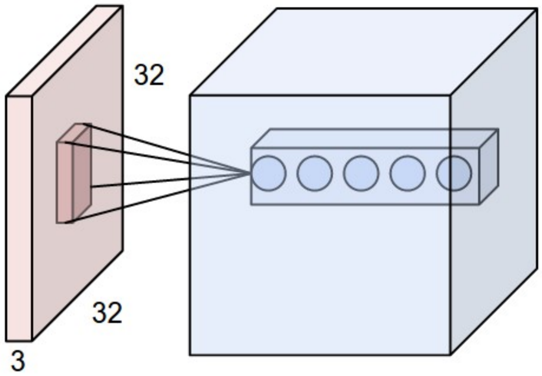


Figure 1: An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input. The neurons compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

width and height), but extends through the full depth of the input volume as shown in Figure 1. During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume, producing a 2-dimensional activation map of that filter. As we slide the filter, across the input, we are computing the dot product between the entries of the filter and the input. Intuitively, the network will learn filters that activate when they see some specific type of feature at some spatial position in the input. Stacking these activation maps for all filters along the depth dimension forms the full output volume. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with neurons in the same activation map (since these numbers all result from applying the same filter).

When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (also called filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to note this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.

We have explained the connectivity of each neuron in the convolutional layer to the input volume, but we haven't yet discussed how many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume: the depth, stride and zero-padding.

- First, the depth of the output volume is a hyperparameter that we can pick; It controls the number of neurons in the convolutional layer that connect to the same region of the input volume. This is analogous to a regular neural network, where we had multiple neurons in a hidden layer all looking at the exact same input. As we will see, all of these neurons will learn to activate for different features in the input. For example, if the first convolutional layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edged, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a depth column.

- Second, we must specify the stride with which we allocate depth columns around the spatial dimensions (width and height). When the stride is 1, then we will allocate a new depth column of neurons to spatial positions only 1 spatial unit apart. This will lead to heavily overlapping receptive fields between the columns, and also to large output volumes. Conversely, if we use higher strides then the receptive fields will overlap less and the resulting output volume will have smaller dimensions spatially.

- Sometimes it will be convenient to pad the input with zeros spatially on the border of the input volume. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes. In particular, we will sometimes want to exactly preserve the spatial size of the input volume.

We can compute the spatial size of the output volume as a function of the input volume size (W), the receptive field size of the convolutional layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The formula for calculating how many neurons ”fit” is given by $1+(W-F+2P)/S$. If this number is not an integer, then the strides are set incorrectly and the neurons cannot be tiled so that they ”fit” across the input volume neatly, in a symmetric way. In general, setting zero padding to be $P = (F-1)/2$ when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially. It is very common to use zero-padding in this way in designing convolutional network architectures.

Note that the spatial arrangement hyperparameters have mutual constraints. For example, when the input has size $W = 10$, no zero-padding is used $P = 0$, and the filter size is $F = 3$, then it would be impossible to use stride $S = 2$, since $1 + (W - F + 2P)/S = 1+(10-3+0)/2 = 4.5$, i.e. not an integer, indicating that the neurons don’t ”fit” neatly and symmetrically across the input. Therefore, this setting of the hyperparameters is considered to be invalid, and a convolutional network library would likely throw an exception.

Here is a real-world example. The Krizhevsky et al. architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size $F = 11$, stride $S = 4$ and no zero padding $P = 0$. Since $1 + (227 - 11)/4 = 55$, and since the convolutional layer had a depth of $K = 96$, the convolutional layer output volume had size [55x55x96]. Each of the 55*55*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but of course with different weights.

Parameter sharing scheme is used in convolutional layers to control the number of parameters. Using the real-world example above, we see that there are 55*55*96 = 290,400 neurons in the first convolutional layer, and each has 11*11*3 = 363 weights and 1 bias. Together, this adds up to 290400 * 364 = 105,705,600 parameters on the first layer of the convolutional network alone. Clearly, this number is very high.

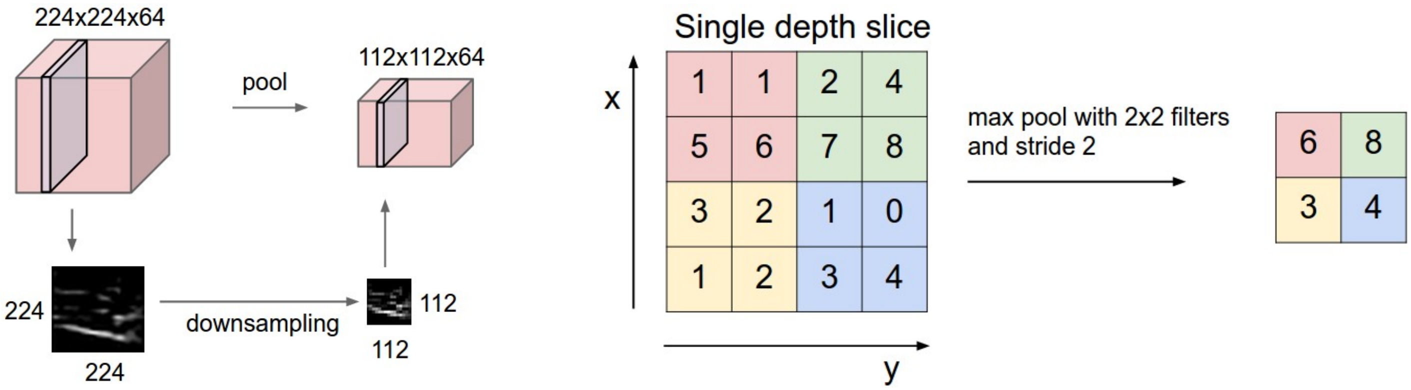
It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one patch feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position $(x2, y2)$. In other words, denoting a single 2-dimensional slice of depth as a depth slice (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), we are going to constrain the neurons in each depth slice to use the same weights and bias. With this parameter sharing scheme, the first convolutional layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases). Alternatively, all 55*55 neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the convolutional layer can in each depth slice be computed as a convolution of the neuron's weights with the input volume (Hence the name: convolutional layer). Therefore, it is common to refer to the sets of weights as a filter (or a kernel), which is convolved with the input. The result of this convolution is an activation map (e.g. of size [55x55]), and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume (e.g. [55x55x96]).

The backward pass for a convolution operation (for both the data and the weights) is also a convolution. Use the chain rule to propagate the upstream derivatives across each filter.

Pooling layer

It is common to periodically insert a pooling layer in-between successive convolutional layers in a convolutional network architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the `max` operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 to downsample every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every `max` operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged. More generally, the pooling



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

Figure 2: Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

layer, accepts a volume of size $W1 \times H1 \times D1$ and produces a volume of size $W2 \times H2 \times D1$ where $W2 = 1 + (W1 - F)/S$ and $H2 = 1 + (H1 - F)/S$. The parameters F and S are the filter size and stride respectively.

To compute the backward pass for a max(x, y) operation we route the gradient to the input that had the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation so that gradient computation is efficient during backpropagation.

Problem 4.4.1: Convolution: naive forward pass (10 points)

The core of a convolutional network is the convolution operation, explained above. In the file `layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`. You don't have to worry too much about efficiency at this point; just write the function in whatever way you find most clear. You can test your implementation by running the `ConvolutionalNetworks.ipynb` notebook.

Problem 4.4.2: Convolution: naive backward pass (10 points)

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `layers.py`. Again, you don't need to worry too much about computational efficiency. When you are done, run the appropriate cell in the `ConvolutionalNetworks.ipynb` notebook to check your backward pass with a numeric gradient check.

Problem 4.4.3: Max pooling: naive forward pass (5 points)

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `layers.py`. Again, don't worry too much about computational efficiency. Check your implementation by running the appropriate cell in the `ConvolutionalNetworks.ipynb` notebook.

Problem 4.4.4: Max pooling: naive backward pass (5 points)

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `layers.py`. You don't need to worry about computational efficiency. Check your implementation with numeric gradient checking by running the appropriate cell in the `ConvolutionalNetworks.ipynb` notebook.

Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `fast_layers.py`. The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `pa5` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass receives upstream derivatives and the cache object and produces gradients with respect to the data and weights. The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation. You can compare the performance of the naive and fast versions of these layers by running the appropriate cell in the `ConvolutionalNetworks.ipynb` notebook.

Convolutional sandwich layers

Previously we introduced the concept of ”sandwich” layers that combine multiple operations into commonly used patterns. In the file `layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. The appropriate cell in the `ConvolutionalNetworks.ipynb` notebook will test these implementations next.

Problem 4.4.5: Three layer convolutional neural network (10 points)

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network. Open the file `cnn.py` and complete the implementation of the `ThreeLayerConvNet` class.

Testing the CNN: loss computation

After you build a new network, one of the first things you should do is check the loss computation. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about log(C) for C classes. When we add regularization this should go up. The appropriate cell in the `ConvolutionalNetworks.ipynb` notebook runs this check for you.

Testing the CNN: gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artificial data and a small number of neurons at each layer. The appropriate cell in the `ConvolutionalNetworks.ipynb` notebookruns this check for you.

Testing the CNN: overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy. Plotting the loss, training accuracy, and validation accuracy should show clear overfitting: The appropriate cell in the `ConvolutionalNetworks.ipynb` notebook runs this check for you.

Problem 4.4.6: Train the CNN on the CIFAR-10 data (5 points)

By training a three-layer convolutional network with hidden dimension of 500 for one epoch, you should achieve greater than 40% accuracy on the CIFAR-10 training set. The appropriate cell in the `ConvolutionalNetworks.ipynb` notebook sets up this CNN and runs it for you. It also visualizes the first-layer convolutional filters from the trained network. Play with the hyper parameters to achieve $> 50\%$ accuracy on the validation set.

Extra credit: Problem 4.4.7: Train the best model for CIFAR-10 data (20 points)

Experiment and try to get the best performance that you can on CIFAR-10 using a ConvNet. Here are some ideas to get you started:

Things you should try

- **Filter size**: Above we used 7x7; this makes pretty pictures but smaller filters may be more efficient
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deeper network? You can implement alternative architectures in the file convnet.py. Some good architectures to try include:
  - (conv-relu-pool)xN - conv - relu - (affine)xM - (softmax or SVM)
  - (conv-relu-pool)XN - (affine)XM - (softmax or SVM)
  - (conv-relu-conv-relu-pool)xN - (affine)xM - (softmax or SVM)

Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple important things to keep in mind: If the parameters are working well, you should see improvement within a few hundred iterations Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all. Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.

Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are not required to implement any of these; however they would be good things to try for extra credit.

- **Alternative update steps**: For the assignment we implemented SGD+momentum, RMSprop, and Adam; you could try alternatives like AdaGrad or AdaDelta.
- **Alternative activation functions** such as leaky ReLU
- **Model ensembles**
- **Data augmentation**

If you do decide to implement something extra, clearly describe it in the your `writeup.pdf`.

What we expect

At the very least, you should be able to train a ConvNet that gets at least 65% accuracy on the validation set. This is just a lower bound - if you are careful it should be possible to get accuracies much higher than that! Extra credit points will be awarded for particularly high-scoring models or unique approaches. Insert your code for building, training, visualizing and testing your model at the bottom of `ConvolutionalNetworks.ipynb`. The final cell in this notebook should contain the training, validation, and test set accuracies for your final trained network. In this notebook you should also write an explanation of what you did, any additional features that you implemented, and any visualizations or graphs that you make in the process of training and evaluating your network. Have fun and happy training!

Acknowledgement

This exercise would not be possible without the brilliant work of Andrej Karpathy and his team at Stanford University.