CS 189 Introduction to Machine Learning
Spring 2019 Jonathan Shewchuk

HW6

Due: Wednesday, April 24 at 11:59 pm

Deliverables:

1. Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up (see below). The Kaggle competition for this assignment can be found at
   - `https://www.kaggle.com/t/b500e3c2fb904ed9a5699234d3469894`
2. Submit a PDF of your homework, **with an appendix listing all your code**, to the Gradescope assignment entitled "Homework 6 Write-Up". In addition, please include, as your solutions to each coding problem, the specific subset of code relevant to that part of the problem. You may typeset your homework in LaTeX or Word (submit PDF format, **not** .doc/.docx format) or submit neatly handwritten and scanned solutions. **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
   - In your write-up, please state with whom you worked on the homework.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats.

     "I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."
3. Submit all the code needed to reproduce your results to the Gradescope assignment entitled "Homework 6 Code". Yes, you must submit your code twice: in your PDF write-up following the directions as described above so the readers can easily read it, and once in compilable/interpretable form so the readers can easily run it. Do **NOT** include any data files we provided. Please include a short file named README listing your name, student ID, and instructions on how to reproduce your results. Please take care that your code doesn't take up inordinate amounts of time or memory. If your code cannot be executed, your solution cannot be verified.

In this assignment, you will develop neural network models with MDS189. Many toy datasets in machine learning (and computer vision) serve as excellent tools to help you develop intuitions about methods, but they cannot be directly used in real-world problems. MDS189 could be.

Under the guidance of a strength coach here at UC Berkeley, we modeled the movements in MDS189 after the real-world Functional Movement Screen (FMS). The FMS has 7 different daily movements, and each is scored according to a specific 0-3 rubric. Many fitness and health-care professionals, such as personal trainers and physical therapists, use the FMS as a diagnostic assessment of their clients and athletes. For example, there is a large body of research that suggests that athletes whose cumulative FMS score falls below 14 have a higher risk of injury. In general, the FMS can be used to assess functional limitations and asymmetries. More recent research has begun investigating the relationship between FMS scores and fall risk in the elderly population.

In modeling MDS189 after the real-world Functional Movement Screen, we hope the insight you gain from the experience of collecting data, training models, evaluating performance, etc. will be meaningful.

A large part of this assignment makes use of MDS189. Thank you to those who agreed to let us use your data in MDS189! Collectively, you have enabled everyone to enjoy the hard-earned reward of data collection.

Download MDS189 immediately. At `3GB+` of data, MDS189 is rather large, and it will require a while to download. You can access MDS189 through this Google form. When you gain access to MDS189, you are required to agree that you will not share MDS189 with anyone else. ***Everyone*** **must fill out this form, and sign the agreement.** If you use MDS189 without signing the agreement, you (and whomever shared the data with you) will receive an **automatic zero** on all the problems on this homework relating to MDS189.

The dataset structure for MDS189 is described in `mds189_format.txt`, which you will be able to find in the Google drive folder.

1 Data Visualization

When you begin to work with a new dataset, one of the first things you should do is spend some time visualizing the data. For images, you must look at the pixels to help guide your intuitions while developing models. Pietro Perona, a computer vision professor at Caltech, has said that when you begin working with a new dataset, "you should spend two days just looking at the data." We do not recommend you spend quite that much time looking at MDS189; the point is that the value of quality time spent visualizing a new dataset cannot be overstated.

We provide several visualization tools in `mds189_visualize.ipynb` that will enable you to view montages of: key frames, other videos frames, ground truth keypoints (i.e., what you labeled in LabelBox), automatically detected keypoints from OpenPose, and bounding boxes based on keypoint detections.

Note: Your responses to the questions in this problem should be at most two sentences.

(a) To get a sense of the per-subject labeling quality, follow the `Part 1: Same subject` instructions in the cell titled `Key Frame visualizations`. For your write-up, you do not need to include any images from your visualizations. You do need to include answers to the following questions (these can be general statements, you are not required to reference specific subject ids):

  i. What do you observe about the quality of key frame annotations? Pay attention to whether the key frames reflect the movement labeled.

  ii. What do you observe about the quality of keypoint annotations? Pay attention to things like: keypoint location and keypoint colors, which should give a quick indication of whether a labeled keypoint corresponds to the correct body joint.

Solution: For the plots in this part, the only variables to change in the code are: `subject_idx`, `plot_image`, and `plot_keypoints`.

For part i: Overall, the quality of key frame annotations look great! There are some minor issues that introduce some noise to the dataset: a couple of key frames for `129` are of the wrong movement; the lunge-left and lunge-right key frames are swapped in `029`; the deadbug-left and deadbug-right key frames are swapped in `086`; the pushup key frame for `132` and `184` are not captured at the moment just after the subject's body breaks the floor.

For part ii: Overall, the quality of keypoint annotations is reasonable! Often, the keypoint visibility would have been better with a slightly more frontal camera angle, and if the camera had been closer to the subject. There are also some less-minor issues that introduce noise to the dataset: `129` and `072` mislabeled keypoints with left/right swaps, which is most noticeable with the arms and the feet; `029` and `132` look like they annotated several rotated frames; `048` labeled the wrong side for their deadbug movements in Labelbox, so their keypoint poses are completely swapped; `072` does not label all the visible keypoints.

(b) To quickly get a sense of the overall variety of data, follow the `Part 2: Random subject` instructions in the cell titled `Key Frame visualizations`. Again, for your write-up, you do not need to include any images from your visualizations. Include an answer to the following question:

i. What do you observe about the variety of data? Pay attention to things like differences in key frame pose, appearance, lighting, frame aspect ratio, etc.

Solution: The key frame poses, camera angle, and appearance for the same movement look quite varied. The key frames have a noticeable variety in shape and aspect ratio. Overall the lighting is great!

(c) We ran the per-frame keypoint detector OpenPose on your videos to estimate the pose in your video frames. Based on these keypoints, we also estimated the bounding box coordinates for a rectangle enclosing the detected subject. Follow the `Part 3: same subject` instructions in the cell titled `Video Frame visualizations`. Again, for your write-up, you do not need to include any images from your visualizations. You do need to include answers to the following question:

i. What do you observe about the quality of bounding box and OpenPose keypoint annotations? Pay attention to things like annotation location, keypoint colors, number of people detected, etc.

ii. Based on the third visualization, where you are asked to look at all video frames for on movement, what do you observe about the sampling rate of the video frames? Does it appear to reasonably capture the movement?

Solution: For the plots in this part, the only variables to change in the code are: `subject_idx`, `plot_image`, and `plot_keypoints`.

For part i: Overall, the bounding boxes and OpenPose keypoints look great! OpenPose sometimes finds false detections, doesn't quite capture all of the person, or gets confused by shadows. OpenPose also struggles more with the movements that are on the ground, which makes sense because its an atypical orientation compared to the

For part ii: This can be visualized by setting `frame_idx = range(15)` in the code. The sampling rate looks reasonable to see the full range of motion of the movement. The movements in MDS189 are functional movements, not power exercises. This means you don't have any quick bursts of motion with MDS189 the way you do with something like a jump. So, even with a relatively low sampling rate (we did every 12th frame), it still looks okay. Note that usually, with human motion, people sample something like every 5th frame. We did every 12th so that the dataset wouldn't be too large to share on Google drive.

(d) For the key frames, we can take advantage of the knowledge that the poses should be similar to the labeled poses in `heatherlckwd`'s key frames. Using Procrustes analysis, we aligned each key frame pose with the corresponding key frame pose from `heatherlckwd`. Compare the plot of the raw Neck keypoints with the plot of the (normalized) aligned Neck keypoints. What do you observe?

Solution: The key observation is that that prior to the normalization/alignment, the keypoints are all over the place because of different image size.

Note: We introduce the aligned poses because we offer them as a debugging tool to help you develop neural network code in problem 2. Your reported results cannot use the aligned poses as training data.

2 Modular Fully-Connected Neural Networks

First, we will establish some notation for this problem. We define

$$h_{i+1} = \sigma(z_i) = \sigma(W_i h_i + b_i).$$

In this equation, W_i is an $n_{i+1} \times n_i$ matrix that maps the input h_i of dimension n_i to a vector of dimension n_{i+1}, where n_{i+1} is the size of layer $i + 1$. The vector b_i is the bias vector added after the matrix multiplication, and σ is the nonlinear function applied element-wise to the result of the matrix multiplication and addition. $z_i = W_i h_i + b_i$ is a shorthand for the intermediate result within layer i before applying the activation function σ. Each layer is computed sequentially where the output of one layer is used as the input to the next. To compute the derivatives with respect to the weights W_i and the biases b_i of each layer, we use the chain rule starting with the output of the network and propagate backwards through the layers, which is where the backprop algorithm gets its name.

In this problem, we will implement fully-connected networks with a modular approach. This means different layer types are implemented individually, which can then be combined into models with different architectures. This enables code re-use, quick implementation of new networks and easy modification of existing networks.

2.1 Layer Implementations

Each layer's implementation will have two defining functions:

1. `forward` This function has as input the output h_i from the previous layer, and any relevant parameters, such as the weights W_i and bias b_i. It returns an output h_{i+1} and a `cache` object that stores intermediate values needed to compute gradients in the backward pass.

```
def forward(h, w):
    """ example forward function skeleton code with h: inputs, w: weights"""
    # Do computations...
    z = # Some intermediate output
    # Do more computations...
    out = # the output
    cache = (h, w, z, out) # Values needed for gradient computation
    return out, cache
```

2. `backward` This function has input: upstream derivatives and the `cache` object. It returns the local gradients with respect to the inputs and weights.

```
def backward(dout, cache):
    """ example backward function skeleton code with dout: derivative of loss with respect to outputs and
    ↪  cache from the forward pass """
    # Unpack cache
    h, w, z, out = cache
    # Use values in cache, along with dout to compute derivatives
    dh = # Derivative of loss with respect to a
    dw = # Derivative of loss with respect to w
    return dh, dw
```

Your layer implementations should go into the provided `layers.py` script. The code is clearly marked with TODO statements indicating what to implement and where.

When implementing a new layer, it is important to manually verify correctness of the forward and backward passes. Typically, the gradients in the backward pass are checked against numerical gradients. We provide a test script `starter_code.ipynb` for you to use to check each of layer implementations, which handles the gradient checking. Please see the comments of the code for how to appropriately use this script.

In your write-up, provide the following for each layer you've implemented.

1. Listings of (the relevant parts of) your code.
2. Written justification/derivation for the derivatives in your backward pass for *all* the layers that you implement.
3. The output of running numerical gradient checking.
4. Answers to any inline questions.

2.1.1 Fully-Connected (fc) Layer

In `layers.py`, you are to implement the forward and backward functions for the fully-connected layer. The fully-connected layer performs an affine transformation of the input: $\text{fc}(h) = Wa + b$. Write your fc layer for a general input h that contains a mini-batch of B examples, each of which is of shape $(d_1, \cdots, d_k)$.

Solution: First, let's derive the partial derivatives. Assume $y = xw + b$, where $x \in N \times D$, $w \in D \times M$, $b \in N \times M$ and $y \in N \times M$. Also assume the gradient from the upper stream is $dout \in N \times M$. Here we copied the bias N times. After getting the derivative of the duplicated bias, we will sum the derivative over the N dimension to get back to the non-copied derivative of b. Then

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial y}\frac{\partial y}{\partial x} = dout \times w^T$$

$$\frac{\partial loss}{\partial w} = \frac{\partial loss}{\partial y}\frac{\partial y}{\partial w} = x^T \times dout$$

$$\frac{\partial loss}{\partial b} = \frac{\partial loss}{\partial y}\frac{\partial y}{\partial b} = dout$$

The derivative of the non-copied b is $numpy.sum(dout, axis = 0)$

Here is the code implementation of the forward and backward pass.

```python
def affine_forward(x, w, b):
    """
    Computes the forward pass for an affine (fully-connected) layer.

    The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
    examples, where each example x[i] has shape (d_1, ..., d_k). We will
    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
    then transform it to an output vector of dimension M.

    Inputs:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """
    out = None
    ###########################################################################
    # TODO: Implement the affine forward pass. Store the result in out. You   #
    # will need to reshape the input into rows.                               #
    ###########################################################################
    pass
    tmp = np.reshape(x, (x.shape[0], -1))
    out = np.matmul(tmp, w) + b
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b)
    return out, cache
```

```python
def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
      - x: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)
      - b: Biases, of shape (M,)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    ###########################################################################
    # TODO: Implement the affine backward pass.                               #
    ###########################################################################
    pass
    dx = np.matmul(dout, w.T)
    dx = np.reshape(dx, x.shape)
    x_flat = np.reshape(x, (x.shape[0], -1))
    dw = np.matmul(x_flat.T, dout)
    db = np.matmul(dout.T, np.ones((x.shape[0], 1)))
    db = np.reshape(db, (db.shape[0], ))
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dw, db
```

Here is the numerical validation of the gradient implementation.

```
# gradient checking: compare the analytical gradient with the numerical gradient
# taking the affine layer as an example
from gradient_check import eval_numerical_gradient_array
import numpy as np
from layers import *
N = 2
D = 3
M = 4
x = np.random.normal(size=(N, D))
w = np.random.normal(size=(D, M))
b = np.random.normal(size=(M, ))
dout = np.random.normal(size=(N, M))

# do a forward pass first
out, cache = affine_forward(x, w, b)
# check grad f/grad w, the [0] below gets the output out of the (output, cache) original output
f=lambda x: affine_forward(x, w, b)[0]
# compute the analytical gradient you wrote, [1] get the dw out of the (dx, dw, db) original output
grad = affine_backward(dout, cache)[0]
# compute the numerical gradient using the provided utility function
ngrad = eval_numerical_gradient_array(f, x, dout)
print(grad)
print(ngrad)
# they should be similar enough within some small error tolerance
```

```
[[-0.18182506 -1.03269485 -0.33482687]
 [ 1.73661752 -0.35510202  0.4531802 ]]
[[-0.18182506 -1.03269485 -0.33482687]
 [ 1.73661752 -0.35510202  0.4531802 ]]
```

```
# gradient checking: compare the analytical gradient with the numerical gradient
# taking the affine layer as an example
from gradient_check import eval_numerical_gradient_array
import numpy as np
from layers import *
N = 2
D = 3
M = 4
x = np.random.normal(size=(N, D))
w = np.random.normal(size=(D, M))
b = np.random.normal(size=(M, ))
dout = np.random.normal(size=(N, M))

# do a forward pass first
out, cache = affine_forward(x, w, b)
# check grad f/grad w, the [0] below gets the output out of the (output, cache) original output
f=lambda w: affine_forward(x, w, b)[0]
# compute the analytical gradient you wrote, [1] get the dw out of the (dx, dw, db) original output
grad = affine_backward(dout, cache)[1]
# compute the numerical gradient using the provided utility function
ngrad = eval_numerical_gradient_array(f, w, dout)
print(grad)
print(ngrad)
# they should be similar enough within some small error tolerance
```

```
[[ 0.0783162  -2.09749933  3.78325199  3.71384037]
 [ 0.32709012 -2.22313992  1.65602808  1.74813519]
 [ 0.09982858  0.4828864  -2.00756334 -1.91158435]]
[[ 0.0783162  -2.09749933  3.78325199  3.71384037]
 [ 0.32709012 -2.22313992  1.65602808  1.74813519]
 [ 0.09982858  0.4828864  -2.00756334 -1.91158435]]
```

```python
# gradient checking: compare the analytical gradient with the numerical gradient
# taking the affine layer as an example
from gradient_check import eval_numerical_gradient_array
import numpy as np
from layers import *
N = 2
D = 3
M = 4
x = np.random.normal(size=(N, D))
w = np.random.normal(size=(D, M))
b = np.random.normal(size=(M, ))
dout = np.random.normal(size=(N, M))

# do a forward pass first
out, cache = affine_forward(x, w, b)
# check grad f/grad w, the [0] below gets the output out of the (output, cache) original output
f=lambda b: affine_forward(x, w, b)[0]
# compute the analytical gradient you wrote, [1] get the dw out of the (dx, dw, db) original output
grad = affine_backward(dout, cache)[2]
# compute the numerical gradient using the provided utility function
ngrad = eval_numerical_gradient_array(f, b, dout)
print(grad)
print(ngrad)
# they should be similar enough within some small error tolerance
```

```
[ 0.85878534 -0.40494985  4.12716819  1.96736251]
[ 0.85878534 -0.40494985  4.12716819  1.96736251]
```

2.1.2 Activation Functions

In `layers.py`, implement the forward and backward passes for the ReLU activation function

$$\sigma_{\text{ReLU}}(\gamma) = \begin{cases} 0 & \gamma < 0 \\ \gamma & \text{otherwise} \end{cases}$$

Note that the activation function is applied element-wise to a vector input.

There are many other activation functions besides ReLU, and each activation function has its advantages and disadvantages. One issue commonly seen with activation functions is vanishing gradients, i.e., getting zero (or close to zero) gradient flow during backpropagation. Which of activation functions (among: linear, ReLU, tanh, sigmoid) experience this problem? Why? What types of one-dimensional inputs would lead to this behavior?

Solution: First, let's derive the backward pass of the ReLU activation. Assume that $y = max(x, 0)$ and $x \in N \times M$. Also assume $\frac{\partial loss}{\partial y} = dout \in N \times M$. Then

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial y}\frac{\partial y}{\partial x} = dout \odot \mathbb{1}_{x>0}$$

Here $\odot$ denotes elementwise multiplication and $\mathbb{1}_{x>0}$ denotes a $0/1$ matrix with the same size as x where 1 is filled when the corresponding entry in x is bigger than 0 and 0 is filled otherwise.

Here is the code implementation.

```python
def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    pass
    out = np.maximum(x, 0.0)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache
```

```python
def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLUs).

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    dx, x = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    pass
    is_non_0 = x > 0.0
    dx = dout * is_non_0
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx
```

Here is the numerical gradient checking.

gradient checking for ReLU layer

```
N = 2
M = 4
x = np.random.normal(size=(N, M))
dout = np.random.normal(size=(N, M))

out, cache = relu_forward(x)
f=lambda x: relu_forward(x)[0]
grad = relu_backward(dout, cache)
ngrad = eval_numerical_gradient_array(f, x, dout)
print(grad)
print(ngrad)
# they should be similar enough within some small error tolerance
```

```
[[-0.08830989 -0.          -0.          -1.29792256]
 [-1.31975742  0.           0.          -0.        ]]
[[-0.08830989  0.           0.          -1.29792256]
 [-1.31975742  0.           0.           0.        ]]
```

Among the linear, ReLU, tanh and sigmoid activation functions, tanh and sigmoid activation would have the vanishing gradient problem. For tanh function,

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

and

$$\frac{\partial y}{x} = \frac{4e^{2x}}{(e^{2x}+1)^2}$$

when x becomes sufficiently large or negatively small, the gradient will goes to zero.

For the sigmoid function,

$$y = \frac{1}{1+e^{-x}}$$

and

$$\frac{\partial y}{x} = \frac{e^{-x}}{(e^{-x}+1)^2}$$

when x becomes sufficiently large or negatively small, the gradient will goes to zero.

2.1.3 Softmax Loss

In subsequent parts of this problem, we will train a network to classify the movements in MDS189. Therefore, we will need the softmax loss, which is comprised of the softmax activation followed by the cross-entropy loss. It is a minor technicality, but worth noting that the softmax is just the squashing function that enables us to apply the cross-entropy loss. Nevertheless, it is a commonly used shorthand to refer to this as the softmax loss.

The softmax function has the desirable property that it outputs a probability distribution. For this reason, many classification neural networks use the softmax. Technically, the softmax activation takes in C input numbers and outputs C scores which represents the probabilities for the sample being in each of the possible C classes. Formally, suppose $s_1 \cdots s_C$ are the C input scores; the outputs of the softmax activations are

$$t_i = \frac{e^{s_i}}{\sum_{k=1}^{C} e^{s_k}}$$

for $i \in [1, C]$. The cross-entropy loss is

$$E = -\log t_c,$$

where c is the correct label for the current example.

Since the loss is the last layer within a neural network, and the backward pass of the layer is immediately calculated after the foward pass, `layers.py` merges the two steps with a single function called `softmax_loss`.

You have to be careful when you implement this loss, otherwise you will run into issues with numerical stability. Let $m = \max_{i=1}^{C} s_i$ be the max of the s_i. Then

$$E = -\log t_c = \log \frac{e^{s_c}}{\sum_{k=1}^{C} e^{s_k}} = \log \frac{e^{s_c - m}}{\sum_{k=1}^{C} e^{s_k - m}} = -(s_c - m) + \log \sum_{k=1}^{C} e^{s_k - m}.$$

We recommend using the rightmost expression to avoid numerical problems.

Finish the softmax loss in `layers.py`.

Solution: First, let's derive the backward pass for the softmax loss layer.

$$\frac{\partial E}{\partial t_c} = -\frac{1}{t_c}$$

$$\frac{\partial t_c}{\partial s_c} = \frac{e^{s_c}(\sum_{k=1}^{C} e^{s_k} - e^{s_c})}{(\sum_{k=1}^{C} e^{s_k})^2}$$

$$\frac{\partial t_c}{\partial s_i} = \frac{-e^{s_c} e^{s_i}}{(\sum_{k=1}^{C} e^{s_k})^2}$$

for $i \neq c$ Thus the final gradient is

$$\frac{\partial E}{\partial s_c} = -1 + \frac{e^{s_c}}{\sum_{k=1}^{C} e^{s_k}}$$

$$\frac{\partial E}{\partial s_i} = \frac{e^{s_i}}{\sum_{k=1}^{C} e^{s_k}}$$

for $i \neq c$

The implementation of this part is

```python
def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """

    loss = 0.0
    dx = None
    ###########################################################################
    # TODO: Implement the softmax loss                                        #
    ###########################################################################
    pass
    shifted_logits = x - np.max(x, axis=1, keepdims=True)
    Z = np.sum(np.exp(shifted_logits), axis=1, keepdims=True)
    log_probs = shifted_logits - np.log(Z)
    probs = np.exp(log_probs)
    N = x.shape[0]
    loss = -np.sum(log_probs[np.arange(N), y]) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return loss, dx
```

The numerical gradient checking is

gradient checking for softmax

```
N = 2
C = 4
x = np.random.normal(size=(N, C))
y = [0, 2]

_, dx = softmax_loss(x, y)
f=lambda x: softmax_loss(x, y)[0]
grad = dx
ngrad = eval_numerical_gradient_array(f, x, 1.0)
print(grad)
print(ngrad)
# they should be similar enough within some small error tolerance
```

```
[[-0.34691788  0.19790347  0.03199411  0.1170203 ]
 [ 0.02315597  0.02366487 -0.1409693   0.09414846]]
[[-0.34691788  0.19790347  0.03199411  0.1170203 ]
 [ 0.02315597  0.02366487 -0.1409693   0.09414846]]
```

2.2 Two-layer Network

Now, you will use the layers you have written to implement a two-layer network (also referred to as a one *hidden* layer network) that classifies movement type based on keypoint annotations. The input features are pre-processed keypoint annotations of an image, and the output are one of 8 possible movement types: deadbug, hamstrings, inline, lunge, stretch, pushup, reach, or squat. You should implement the following network architecture: input - fc layer - ReLU activation - fc layer - softmax loss. Implement the class `FullyConnectedNet` in `fc_net.py`. Note that this class supports multi-layer networks, not just two-layer networks. You will need this functionality in the next part. In order to train your model, you need two other components, listed below.

1. The data loader, which is responsible for loading batches of data that will be fed to your model during training. Data pre-processing should be handled by the data loader.

2. The solver, which encapsulates all the logic necessary for training models.

You don't need to worry about those, since they are already implemented for you. See `starter_code.ipynb` for an example.

For your part, you will need to instantiate a model of your two-layer network, load your training and validation data, and use a `Solver` instance to train your model. Explore different hyperparameters including the learning rate, learning rate decay, batch size, the hidden layer size, and the weight_scale initialization for the parameters. Report the results of your exploration, including what parameters you explored and which set of parameters gave the best validation accuracy.

Debugging note: The default data loader returns raw poses, i.e., the ones that you labeled in LabelBox. As a debugging tool only, you can replace this with the `heatherlckwd`-aligned, normalized poses. It's easier and faster to get better performance with the aligned poses. Use this for debugging only! You can use this feature by setting `debug = True` in the starter code. All of your reported results **must** use the un-aligned, raw poses for training data.

Solution: See the ipython notebook for the hyperparameter tuning. The best hyper-parameter we get is hidden dimension 80, weight scale 0.03, learning rate 0.3, learning rate decay 0.98, number of epochs 300 and batch size of 30. With five training trails, we get a validation accuracy of 92.87% with std of 0.26%.

2.3 Multi-layer Network

Now you will implement a fully-connected network with an arbitrary number of hidden layers. Use the same code as before and try different number of layers (1 hidden layer to 4 hidden layers) as well as different number of hidden units. Include in your write-up what kinds of models you have tried, their hyperparameters, and their training and validation accuracies. Report which architecture works best.

Solution: See the ipython notebook for detailed ablation. The best 2 hidden layer network we found has 160 and 40 hidden units and the validation performance is 93.5% with std of 0.32%. Also when further adding more hidden layers does not improve the performance.

3 Convolution and Backprop Revisited

In this problem, we will explore how image masking can help us create useful high-level features that we can use instead of raw pixel values. We will walk through how discrete 2D convolution works and how we can use the backprop algorithm to compute derivatives through this operation.

(a) To start, let's consider convolution in one dimension. Convolution can be viewed as a function that takes a signal $I[]$ and a mask $G[]$, and the discrete convolution at point t of the signal with the mask is

$$(I * G)[t] = \sum_{k=-\infty}^{\infty} I[k]G[t-k]$$

If the mask $G[]$ is nonzero in only a finite range, then the summation can be reduced to just the range in which the mask is nonzero, which makes computing a convolution on a computer possible.

As an example, we can use convolution to compute a derivative approximation with finite differences. The derivative approximation of the signal is $I'[t] \approx (I[t+1] - I[t-1])/2$. Design a mask $G[]$ such that $(I * G)[t] = I'[t]$.

Solution:

If we set $G[1] = -1/2$, $G[-1] = 1/2$, and $G[t] = 0$ everywhere else, we can use the equation for the discrete convolution to verify that $I * G$ gives us the derivative approximation.

(b) Convolution in two dimensions is similar to the one-dimensional case except that we have an additional dimension to sum over. If we have some image $I[x, y]$ and some mask $G[x, y]$, then the convolution at the point (x, y) is

$$(I * G)[x, y] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} I[m, n]G[x - m, y - n]$$

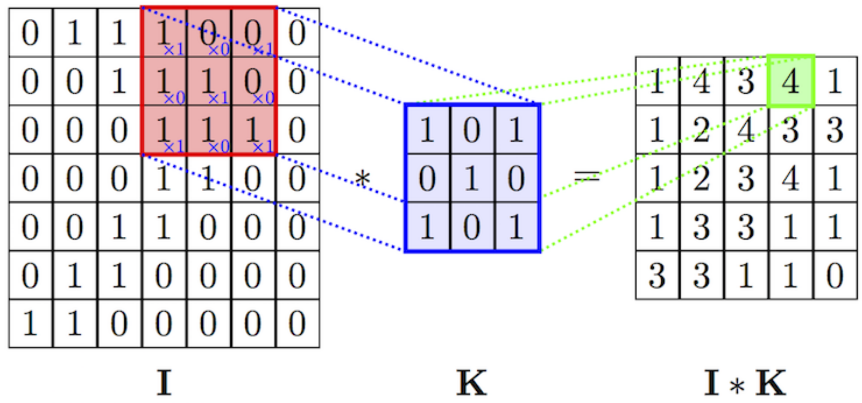

Figure 1: Figure showing an example of one convolution.

or equivalently,

$$(I * G)[x, y] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} G[m, n] I[x - m, y - n],$$

because convolution is commutative.

In an implementation, we'll have an image I that has three color channels I_r, I_g, I_b each of size $W \times H$ where W is the image width and H is the height. Each color channel represents the intensity of red, green, and blue for each pixel in the image. We also have a mask G with finite support. The mask also has three color channels, G_r, G_g, G_b, and we represent these as a $w \times h$ matrix where w and h are the width and height of the mask. (Note that usually $w \ll W$ and $h \ll H$.) The output $(I * G)[x, y]$ at point (x, y) is

$$(I * G)[x, y] = \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} \sum_{c \in \{r,g,b\}} I_c[x + a, y + b] \cdot G_c[a, b]$$

In this case, the size of the output will be $(1 + W - w) \times (1 + H - h)$, and we evaluate the convolution only within the image I. (For this problem we will not concern ourselves with how to compute the convolution along the boundary of the image.) To reduce the dimension of the output, we can do a strided convolution in which we shift the convolutional mask by s positions instead of a single position, along the image. The resulting output will have size $\lfloor 1 + (W - w)/s \rfloor \times \lfloor 1 + (H - h)/s \rfloor$.

Write pseudocode to compute the convolution of an image I with a set of masks G and a stride of s. Hint: to save yourself from writing low-level loops, you may use the operator $*$ for element-wise multiplication of two matrices (which is not the same as matrix multiplication) and invent other notation when convenient for simple operations like summing all the elements in the matrix.

Solution:

Note that the weights in a mask are shared across all the pixels in the input. For a convolutional network, we always use weight sharing because the same masks are applied across multiple positions of an input and because it reduces model complexity, which allows for fewer parameters than using a fully connected network.

```
for x in {0,s,2s,...,W-w}
```

```
for y in {0,s,2s,...,H-h}
 total = 0
 for c in {r,g,b}
  window = I_c[x:x+w,y:y+h]
  conv = window * G_c // * is element-wise multiplication
  total = total + summation(conv)
 out_c[x/s,y/s] = total
```

The operator $*$ is element-wise multiplication of the two matrices, and summation() is the sum of all elements in the matrix.

(c) Masks can be used to identify different types of features in an image such as edges or corners. Design a mask G that outputs a large value for vertically oriented edges in image I. By "edge," we mean a vertical line where a black rectangle borders a white rectangle. (We are not talking about a black line with white on both sides.)

Solution: An example vertical edge detector could look like

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \\ \vdots & \vdots \\ -1 & 1 \end{bmatrix} \text{ or, better yet, } \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ & \vdots & \\ -1 & 0 & 1 \end{bmatrix}$$

This detector would return a large positive value for edges that go from low color intensity to high color intensity and will return a large negative value for edges that go from high color intensity to low color intensity. The height of the detector determines the length of the edge that it can detect.

(d) Although handcrafted masks can produce edge detectors and other useful features, we can also learn masks (sometimes better ones) as part of the backpropagation algorithm. These masks are often highly specific to the problem that we are solving. Learning these masks is a lot like learning weights in standard backpropagation, but because the same mask (with the same weights) is used in many different places, the chain rule is applied a little differently and we need to adjust the backpropagation algorithm accordingly. In short, during backpropagation each weight w in the mask has a partial derivative $\frac{\partial L}{\partial w}$ that receives contributions from every patch of image where w is applied.

Let L be the loss function or cost function our neural network is trying to minimize. Given the input image I, the convolution mask G, the convolution output $R = I * G$, and the partial derivative of the error with respect to each scalar in the output, $\frac{\partial L}{\partial R[i,j]}$, write an expression for the partial derivative of the loss with respect to a mask weight, $\frac{\partial L}{\partial G_c[x,y]}$, where $c \in \{r, g, b\}$. Also write an expression for the derivative of $\frac{\partial L}{\partial I_c[x,y]}$.

Solution:

By the chain rule, we have

$$\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} \frac{\partial R[i,j]}{\partial G_c[x,y]}$$

and

$$\frac{\partial L}{\partial I_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} \frac{\partial R[i,j]}{\partial I_c[x,y]}.$$

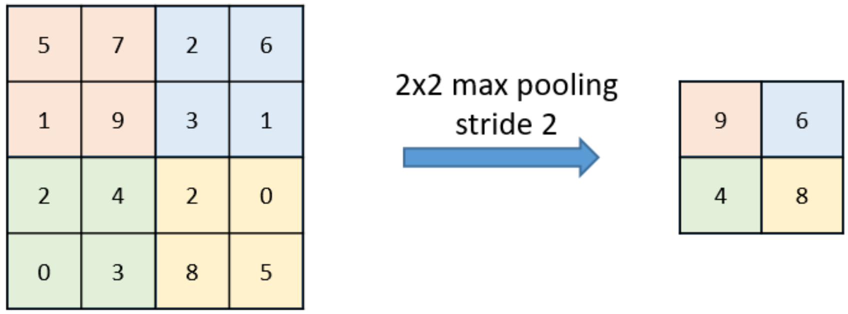

Figure 2: Figure showing an example of one maxpooling.

From the equation for discrete convolution, the derivative for each entry $R[i, j]$ is

$$\frac{\partial R[i,j]}{\partial G_c[x,y]} = \frac{\partial}{\partial G_c[x,y]} \sum_{c\in\{r,g,b\}} \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} I_c[i+a, j+b] \cdot G_c[a,b]$$
$$= I_c[i+x, j+y].$$

For the input image, the derivative is

$$\frac{\partial R[i,j]}{\partial I_c[x,y]} = \frac{\partial}{\partial I_c[x,y]} \sum_{c\in\{r,g,b\}} \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} I_c[i+a, j+b] \cdot G_c[a,b]$$
$$= G_c[x-i, y-j].$$

When $x-i$ or $y-j$ go outside the boundary of the mask, we can treat the derivative as zero.

It follows that

$$\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} I_c[i+x, j+y], \tag{1}$$

and for the image,

$$\frac{\partial L}{\partial I_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} G_c[x-i, y-j].$$

(e) Sometimes, the output of a convolution can be large, and we might want to reduce the dimensions of the result. A common method to reduce the dimension of an image is called max pooling. This method works similar to convolution in that we have a mask that moves around the image, but instead of multiplying the mask with a subsection of the image, we take the maximum value in the subimage. Max pooling can also be thought of as downsampling the image but keeping the largest activations for each channel from the original input. To reduce the dimension of the output, we can do a strided max pooling in which we shift the max pooling mask by s positions instead of a single position, along the input. Given a mask size of $w \times h$, and a stride s, the output will be $\lfloor 1 + (W-w)/s \rfloor \times \lfloor 1 + (H-h)/s \rfloor$ for an input image of size $W \times H$.

Let the output of a max pooling operation be an array R. Write a simple expression for element $R[i, j]$ of the output.

Solution:

$$R[i, j] = \max_{a=\{0,\ldots,w-1\}} \max_{b=\{0,\ldots,h-1\}} I_c[i+a, j+b].$$

(f) Explain how we can use the backprop algorithm to compute derivates through the max pooling operation. (A plain English answer will suffice; equations are optional.)

Solution: Similar to how we computed the derivatives through a convolution layer, we'll be given the derivative with respect to the output of the maxpool layer.

The gradient from the next layer is passed back only to the neuron which achieved the max. All other neurons get zero gradient.

Because maxpooling doesn't have any trainable parameters, we won't need to worry about calculating any derivatives for weights.

Once we have the derivative with respect to the input, the backprop algorithm can continue on to the layer before the maxpool by using this derivative.

4 Convolutional Neural Networks (CNNs)

In this problem we will revisit the problem of classifying movements based on the key frames. The fully-connected networks we have worked with in the previous problem have served as a good testbed for experimentation because they are very computationally efficient. However, in practice state-of-the-art methods on image data use convolutional networks.

It is beyond the scope of this class to implement an efficient forward and backward pass for convolutional layers. Therefore, it is at this point that we will leave behind your beautiful code base from problem 1 in favor of developing code for this problem in the popular deep learning framework PyTorch.

PyTorch executes dynamic computational graphs over Tensor objects that behave similarly to numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual backpropagation. You should install PyTorch and take a look at the basic tutorial here: `https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`. The installation instructions can be found at `https://pytorch.org/` under 'Quick Start Locally'. You will be able to specify your operating system and package manager (e.g., `pip` or `conda`).

Debugging notes

1. One of the most important debugging tools when training a new network architecture is to train the network first on a small set of data, and verify that you can overfit to that data. This could be as small as a single image, and should not be more than a batch size of data.

2. You should see your training loss decrease steadily. If your training loss starts to increase rapidly (or even steadily), you likely need to decrease your learning rate. If your training loss hasn't started noticeably decreasing within one epoch, your model isn't learning anything. In which case, it may be time to either: a) change your model, or b) increase your learning rate.

3. It can be helpful to save a log file for each model that contains the training loss for each N steps, and the validation loss for each $M >> N$ steps. This way, you can plot the loss curve vs number of iterations, and compare the loss curves between models. It can help speed up the comparison between model performances.

4. Do not delete a model architecture you have tried from the code. Often, you want the flexibility to run any model that you have experimented with at any time without a re-coding effort.

5. Keep track of the model architectures you run, save each model's weights, and record the evaluation scores for each model. For example, you could record this information in a spreadsheet with structure: model architecture info (could be as simple as the name of the model used in the code), accuracy for each of the 8 classes, average accuracy across all 8 classes, and location of the model weights.

These networks take time to train. Please start early!

Cloud credits. Training on a CPU is much slower than training on a GPU. We don't want you to be limited by this. You have a few options for training on a GPU:

1. Google has generously provided \$50 in cloud credits for each student in our class. This is exclusively for students in CS 189/289A. Please do not share this link outside of this class. We were only given enough cloud credits for each student in the class to get one \$50 credit. Please be reasonable.

2. Google Cloud gives first-time users \$300 in free credits, which anyone can access at `https://cloud.google.com/`

3. (least user-friendly) Amazon Web Services gives first-time users \$100 in free credits, which anyone can access at `https://aws.amazon.com/education/awseducate/`

4. (most user-friendly) Google Colab, which interfaces with Google drive, operates similarly to Jupyter notebook, and offers free GPU use for anyone at `https://colab.research.google.com/` Google Colab also offers some nice tools for visualizing training progress (see debugging note 3 above).

(a) Implement a CNN that classifies movements based on a single key frame as input. We provide skeleton code in `problem4`, which contains the fully implemented data loader (`mds189.py`) and the solver (in `train.py`). For your part, you are to write the model, the loss, and modify the evaluation. There are many `TODO` and `NOTE` statements in `problem4/train.py` to help guide you. Experiment with a few different model architectures, and report your findings.

(b) For your best CNN model, plot the training and validation loss curves as a function of number of steps.

(c) Draw the architecture for your best CNN model. How do the number of parameters compare between your best CNN and a comparable architecture in which you replace all convolutional layers with fully-connected layers?

(d) Train a movement classification CNN with your best model architecture from part (a) that now takes as input a random video frame, instead of a key frame. Note: there are many more random frames than there are key frames, so you are unlikely to need as many epochs as before.

(e) Compare your (best) key frame and (comparable architecture) random frame CNN performances by showing their per-movement accuracy in a two-row table. Include their overall accuracies in the table.

(f) When evaluating models, it is important to understand your misclassifications and error modes. For your random image and key frame CNNs, plot the confusion matrices. What do you observe? For either CNN, visualize your model's errors, i.e., look at the images and/or videos where the network misclassifies the input. What do you observe about your model's errors? Be sure to clearly state which model you chose to explore.

(g) For the Kaggle competition, you will evaluate your best CNN trained for the task of movement classification based on a random video frame as input. In part (d), we did not ask you to tune your CNN in any way for the video frame classifier. For your Kaggle submission, you are welcome to make any improvements to your CNN. The test set of images is located in the `test_kaggle_frames` directory in the dataset Google drive folder. For you to see the format of the Kaggle submission, we provide the sample file `kaggle_submission_format.csv`, where the `predicted_labels` should be replaced with your model's prediction for the movement, e.g., reach, squat, inline, lunge, hamstrings, stretch, deadbug, or pushup.