CS 189 Introduction to Machine Learning
Spring 2020 Jonathan Shewchuk

HW6

Due: Wednesday, April 22 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/b94c6f749b59461ab12baea4552ca7c1`

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.

1 The History of Neural Networks

Many of the researchers involved in the early development of modern computers were inspired by the computations performed by neurons, including both Alan Turing (inventor of the modern concept of computation) and John von Neumann (inventor of the architecture used in most modern computing devices). **Artificial neural networks** were first conceived in 1943 (!) by Warren McCulloch and Walter Pitts, who aimed to describe the computation of a neuron as a mathematical function. Both Turing and von Neumann described their own versions of artificial neural networks a few years later.

In a *biological neuron*, electrical or chemical signals from other neurons are received at synapses along a neuron's dendrites. Each synapse alters the strength of the signals, which are then integrated in the body of the neuron. When the electrical potential within the neuron crosses some threshold, a chain of events is set off that results in a spike of activity that shoots down the neuron's axon and is sent to other downstream neurons.

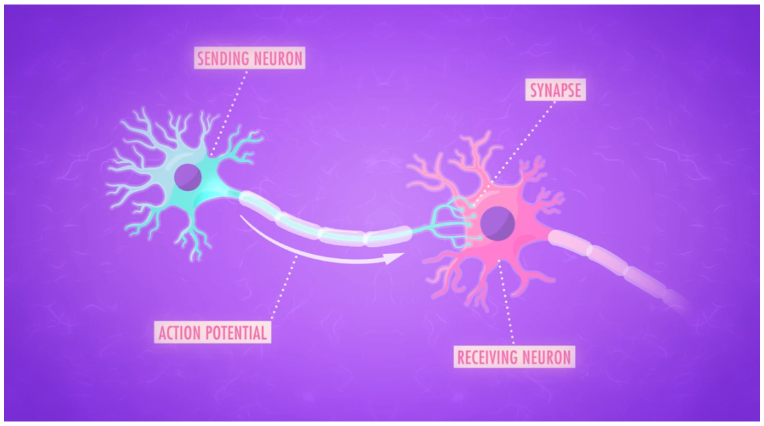

Figure 1: Transmission of signals in biological neurons. Figure from Crash Course

At this very high level of abstraction, a neuron can be described with a simple model. McCulloch & Pitts proposed one of the earliest computational neuron models, which they called a **Linear Threshold Unit**. The McCulloch–Pitts neuron integrates signals from other neurons as a weighted sum. If this weighted sum crosses some threshold, the neuron fires and spits out a 1. Otherwise, it stays silent and spits out a 0. That is, the output of the neuron $f(x)$ is

$$f(x) = \phi(w \cdot x - \tau),$$

where x is a d-dimensional input vector, w is vector of weights over the d features, τ is the threshold, and ϕ is the Heaviside step function. With this framework, *learning* in networks of neurons can then be achieved by altering the strength of the synapses between them. One of the earliest proposals for a biological mechanism for this sort of neural learning was proposed by Donald Hebb.

Modern feed-forward neural networks are direct descendants of this basic model: a nonlinear "activation function" applied to a weighted linear combination of inputs. With modern neural networks, we typically use the Rectified Linear Unit, sigmoid, tanh, or softmax functions as our nonlinear activation functions, which allow for graded rather than binary outputs.

The perceptron, which is a variety of single-layer neural network, was one of the earliest successful machine learning algorithms based on the artificial neuron model. Researchers were initially enormously optimistic about the perceptron learning algorithm. But Marvin Minsky and Seymour Papert delivered a devastating blow to the perceptron when they showed that there were very simple logical functions that a perceptron would be unable to compute, such as the boolean XOR (exclusive-or) function. Minksy and Papert's book led to a major lull in neural networks research. But in the 1980s and '90s, neural network research was revived by two major realizations.

1. A *multi-layer* neural network with a locally bounded, piecewise continuous activation function can approximate any continuous function, including XOR. (See `https://en.wikipedia.org/wiki/Universal_approximation_theorem`).
2. Multi-layer neural networks can be trained efficiently using the backpropagation algorithm, which was introduced by Rumelhart, Hinton, and Williams in 1986.

These two realizations reactivated interest in neural networks. The additional insights that huge datasets (whose existence was largely enabled by the internet) make neural net training much more effective, and that GPUs are dramatically more efficient at performing the computations involved in neural networks, caused an explosion of interest and research in neural networks in the 2010s that has transformed industrial machine learning.

2 Constructing Neural Networks from Scratch

Many of the most exciting recent breakthroughs in machine learning have come from "deep" (read: many-layer) neural networks, such as the deep reinforcement learning algorithm that learned to play Atari from pixels, or the GPT-2 model, which generates text that is nearly indistinguishable from human-generated text.

Neural network libraries such as Tensorflow and PyTorch have made training complicated neural network architectures very easy. You don't even really need to understand how they work! With just a few lines of code, you can take a pre-defined neural network architecture and train it on your dataset. These libraries are wonderful for experienced practitioners who understand neural networks inside and out and want to work with a lot of complex machinery at a high level. They're also wonderful for those who don't care to dive deep into the inner workings of neural networks and want to just use pre-defined functions. But for those who want to dive deep and are just learning the material, they tend to obscure the fundamental simplicity and elegance of the inner workings of neural networks. It is easy to get lost in the complexity of the very many classes and parameters defined in these libraries.

In this assignment, we want to emphasize that neural networks begin with a fundamentally simple model that is just a few steps removed from basic logistic regression. In this assignment, you will build three fundamental types of neural network models, all in plain `numpy`: a **feed-forward fully-connected network**, a **recurrent neural network**, and a **convolutional neural network**. We will start with the essential elements and then build up in complexity.

A neural network model is defined by the following.

- An **architecture** defining the flow of information between layers. This defines the composition of functions that the network performs from input to output.
- A **cost function** (e.g. cross-entropy or mean squared error).
- An **optimization algorithm** (e.g. stochastic gradient descent with backpropagation).
- A set of **hyperparameters** (e.g. learning rate, batch size, etc.).

Each *layer* is defined by the following components.

- A **parameterized function** that defines the layer's map from input to output (e.g. $f(x) = \sigma(Wx + b)$).
- An **activation function** σ (e.g. ReLU, sigmoid, etc.).
- A set of **parameters** (e.g. weights and biases).

Neural networks are commonly used for supervised learning problems, where we have a set of inputs and a set of labels, and we want to learn the function that maps inputs to labels. To learn this function, we need to update the parameters of the network (the weights and biases). We do this using **stochastic gradient descent with backpropagation**.

In the backpropagation algorithm, we first compute what is called a "forward pass" of the network. In the forward pass, we send a mini-batch of input data (e.g. 50 datapoints) through the network. The result is a set of outputs, which we use to compute our loss function. We then take the derivatives of this loss with respect to the parameters of each layer, starting with the output of the network and using the chain rule to propagate backwards through the layers. This is called the "backward pass." The backpropagated errors are then used to update the parameters in the appropriate directions. In essence, backpropagation is a dynamic programming algorithm that avoids recomputing the same intermediate derivatives.

To summarize, training a neural network involves three steps.

1. Forward propagation of inputs.
2. Computing the cost.
3. Backpropagation and parameter updates.

We have provided a modularized codebase for constructing neural networks. The codebase has the following structure.

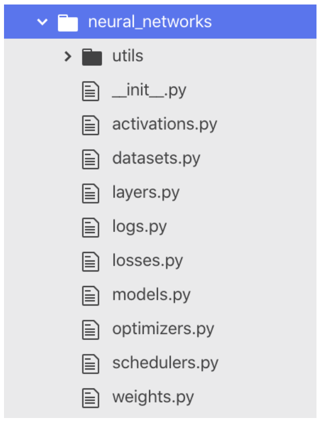
```
neural_networks
    utils
    __init__.py
    activations.py
    datasets.py
    layers.py
    logs.py
    losses.py
    models.py
    optimizers.py
    schedulers.py
    weights.py
```

Figure 2: The structure of the starter codebase.

As you can see, the modules in the codebase reflect the structure outlined above. Different losses, activations, layers, optimizers, hyperparameters, and neural network architectures can be combined to yield different architectures.

Each type of neural network architecture builds in certain assumptions about the structure of the data it receives. We will begin with a feed-forward, fully-connected network, which makes the fewest assumptions and will build up in complexity from there.

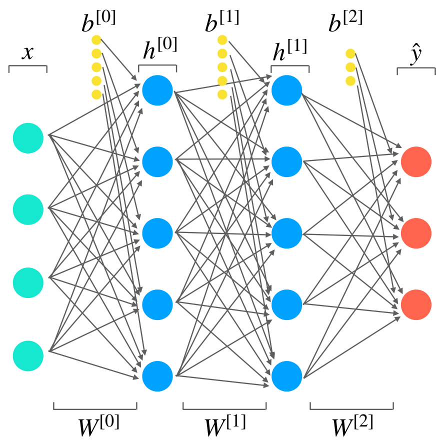

Figure 3: A 3-layer fully-connected neural network.

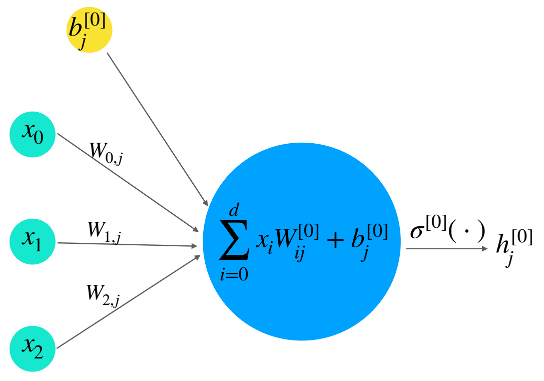

Figure 4: A single fully-connected neuron.

3 Feed-Forward, Fully-Connected Neural Networks

A feed-forward, fully-connected neural network layer performs an affine transformation of an input, followed by a nonlinear activation function. We will use the following notation when defining fully-connected layers, with superscripts surrounded by brackets indexing layers and subscripts indexing the vector/matrix elements.

- x: A single data vector, of shape $1 \times d$, where d is the number of features.
- y: A single label vector, of shape $1 \times k$, where k is the number of classes (for a classification problem), or the number of output features (for a regression problem).
- $n^{[l]}$: The number of neurons in layer l.
- $W^{[l]}$: A matrix of weights connecting layer $l-1$ with layer l, of shape $n^{[l-1]} \times n^{[l]}$. At layer 0, it is shape $d \times n^{[l]}$.
- $b^{[l]}$: The bias vector for layer l, of shape $1 \times n^{[l]}$.
- $h^{[l]}$: The output of layer l. This is a vector of shape $1 \times n^{[i]}$.
- $\sigma^{[l]}(\cdot)$: The nonlinear "activation function" applied at layer l.

A fully-connected layer l is a function

$$\phi^{[l]}(h^{[l-1]}) = \sigma^{[l]}(h^{[l-1]}W^{[l]} + b^{[l]}) = h^{[l]}.$$

At layer 0, $h^{[l-1]}$ is simply the data vector x. We will use the term $z^{[l]} = h^{[l-1]}W^{[l]} + b^{[l]}$ as shorthand for the intermediate result within layer l before applying the activation function σ. Each layer is computed sequentially and the output of one layer is used as the input to the next. A neural network is thus a *composition of functions*. We want to find the parameters of the function that takes us from our input examples x to our labels y.

In this problem you will build a feed-forward neural network for classification. Inputs will be d-dimensional vectors, and the labels will be k-dimensional "one-hot" vectors, where k is the number of classes. A one-hot vector is a binary vector whose elements are computed according to the following function:

$$y_i = \begin{cases} 1 & x \in \text{class } i, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for a classification problem with 3 classes, the label encoding an example from class 2 (zero-indexing) would be: $[0, 0, 1]$.

You 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 reuse, quick implementation of new networks and easy modification of existing networks. You will be feeding the networks "mini-batches" of m datapoints rather than individual examples, so in your implementation, the data X and labels Y will be matrices of dimension $m \times d$ and $m \times k$, respectively.

3.1 Layer Implementations

In the codebase we have provided, each layer is an object with a few relevant attributes.

- `parameters`: An `OrderedDict` containing the weights and biases of the layer.
- `gradients`: An `OrderedDict` containing the derivatives of the loss with respect to the weights and biases of the layer, with the same keys as `parameters`.
- `cache`: An `OrderedDict` containing intermediate quantities calculated in the forward pass that are useful for the backward pass.
- `activation`: An `Activation` instance that is the activation function applied by this layer.
- `n_in`: The number of input units (input channels in CNN).
- `n_out`: The number of output units (output channels in CNN).

You will pass the layer a parameter that selects an activation function from those defined in `activations.py`. This will be stored as an attribute of the layer, which can be called as `layer.activation()`. The forward and backward passes of the layer are defined by the following methods.

- `forward` This method takes as input the output `X` from the previous layer (or input data). This method computes the function $\phi(\cdot)$ from above, combining the input with the weights `W` and bias `b` that are stored as attributes. It returns an output `out` and saves the intermediate value `Z` to the `cache` attribute, as it is needed to compute gradients in the backward pass.

```
def forward(self, X: np.ndarray) -> np.ndarray:
    """Forward pass: multiply by a weight matrix, add a bias, apply activation.
    Also, store all necessary intermediate results in the `cache` dictionary
    to be able to compute the backward pass.
    """
    # initialize layer parameters if they have not been initialized
    if self.n_in is None:
        self._init_parameters(X.shape)

    # unpack model parameters
    W = self.parameters["W"]
    b = self.parameters["b"]

    # perform an affine transformation and activation
    Z = # some intermediate quantity
    out = # the output

    # store information necessary for backprop in `self.cache`
    self.cache[...] = # something useful for backpropagation
    self.cache[...] = ...

    return out
```

- backward This method takes the gradient of the downstream loss as input and uses the cached values to compute gradients with respect to its inputs and weights. It returns the gradient of the loss with respect to the input of the layer.

```
def backward(self, dLdY: np.ndarray) -> np.ndarray:
    """Backward pass for fully connected layer.
    Compute the gradients of the loss with respect to:
        1. the weights of this layer (mutate the `gradients` dictionary)
        2. the bias of this layer (mutate the `gradients` dictionary)
        3. the input of this layer (return this)
    """

    # unpack the cache
    ... = self.cache[...]

    # use values in the cache, along with dLdY to compute derivatives
    dX = # Derivative of loss with respect to X
    dW = # Derivative of loss with respect to W
    dB = # Derivative of loss with respect to b

    # store the gradients in `self.gradients`
    # the gradient for self.parameters["W"] should be stored in
    # self.gradients["W"], etc.

    self.gradients["W"] = dW
    self.gradients["b"] = dB

    return dX
```

Each activation function has a similar (but simpler) structure:

```
class Linear(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for f(z) = z."""
        return Z

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for f(z) = z."""
        return dY
```

3.1.1 Activation Functions

First, you will implement an activation function in `activations.py`. You will implement the forward and backward passes for the ReLU activation function, commonly used in the hidden layers of neural networks,

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

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

Instructions

1. First, derive the gradient of the downstream loss with respect to the input of the ReLU activation function, Z.

2. Next, implement the forward and backward passes of the ReLU activation in the script `activations.py`. Include a screenshot of your code in your writeup.

3.1.2 Fully-Connected Layer

Now you will implement the forward and backward passes for the fully-connected layer in the `layers.py` script. The code is marked with `YOUR CODE HERE` statements indicating what to implement and where. Please read the docstrings and the function signatures too. Write the fully-connected layer for a general input h that contains a mini-batch of m examples with d features.

When implementing a new layer, it is important to manually verify correctness of the forward and backward passes. We have provided a Jupyter notebook `check_gradients.ipynb` for you to use to numerically check the gradients of your layer implementations. Simply run the cell corresponding to each layer. The printed errors should be very small, usually on the order of 10^{-8} or smaller.

Instructions

1. First, derive the partial derivatives of the downstream loss L with respect to W and b in the fully-connected layer, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. You will also need to take the derivative of the loss with respect to the input of the layer $\frac{\partial L}{\partial X}$, which will be passed to lower layers.

2. Implement the forward and backward passes of the fully-connected layer in `layers.py`. First, initialize the weights of the model using `_init_parameters`, which takes the shape of the design matrix as input and initializes the parameters, cache, and gradients of the layer. The `backward` method takes in an argument `dLdY`, the derivative of the loss with respect to the output of the layer, which is computed by higher layers and backpropagated. This should be incorporated into your gradient calculation. In your writeup, include screenshots of the parts of the code you have implemeted.

3. Use the numerical gradient checking notebook to check the validity of your implementations. Provide us with screenshots of the output of this script.

3.1.3 Softmax Activation

Next, we need to define an activation function for the output layer. The ReLU activation function returns continuous values that are (potentially) unbounded to the right. Since we are building a classifier, we want to return *probabilities* over classes. The softmax function has the desirable property that it outputs a probability distribution. That is, the softmax function squashes continuous values into the range $[0, 1]$ and normalizes the outputs so that they add up to 1. For this reason, many classification neural networks use the softmax activation. The softmax activation takes in a vector s of k un-normalized values $s_1, \ldots, s_k$ and outputs a probabiity distribution over the k possible classes. The forward pass of the softmax activation on input s_i is

$$\sigma_i = \frac{e^{s_i}}{\sum_{j=1}^{k} e^{s_j}},$$

where k ranges over all elements in s. Due to issues of numerical stability, the following modified version of this function is commonly used.

$$\sigma_i = \frac{e^{s_i - m}}{\sum_{j=1}^{k} e^{s_j - m}},$$

where $m = \max_{j=0}^{k} s_j$. We recommend implementing this method.

Instructions

1. Derive the Jacobian of the softmax activation function.
2. Implement the forward and backward passes of the softmax activation in `activations.py`. We recommend vectorizing the backward pass for efficiency.

3.1.4 Cross-Entropy Loss

For this classification network, we will be using the multi-class cross-entropy loss function

$$L = -y \ln(\hat{y}),$$

where y is the binary one-hot vector encoding the ground truth labels and $\hat{y}$ is the network's output, a vector of probabilities over classes. The cross-entropy cost calculated for a mini-batch of m samples is

$$J = -\frac{1}{m}\left(\sum_{i=1}^{m} Y_i \ln(\hat{Y}_i)\right).$$

Instructions

1. Derive the gradient of the cross-entropy cost with respect to the network's predictions, $\hat{Y}$.
2. Implement the forward and backward passes of the cross-entropy cost. Note that in the codebase we have provided, we use the words "loss" and "cost" interchangeably. This is consistent with most large neural network libraries, though technically "loss" denotes the function computed for a single datapoint whereas "cost" is computed for a batch. You will be computing over batches.

3.2 Two-Layer Networks

Now, you will use the methods you've written to train a two-layer network (also referred to as a one *hidden* layer network). You will use the **Iris Dataset**, which contains 4 features for 3 different classes of irises.

Instructions

1. Fill in the `forward` and `backward` methods for the `NeuralNetwork` class in `models.py`. Define the parameters of your network in `train_ffnn.py`. We have provided you with several other classes that are critical for the training process.

    - The data loader (in `datasets.py`), which is responsible for loading batches of data that will be fed to your model during training. You may wish to alter the data loader to handle data pre-processing. Note that all datasets you are given have not been normalized or standardized.
    - The stochastic gradient descent optimizer (in `optimizers.py`), which performs the gradient updates and optionally incorporates a momentum term.
    - The learning rate scheduler (in `schedulers.py`), which handles the optional learning rate decay. You may choose to use either a constant or exponentially decaying learning rate.
    - Weight initializers (in `weights.py`). We provide you with many options to explore, but we recommend using `xavier_uniform` as a default.
    - A logger (in `logs.py`), which saves hyperparameters and learned parameters and plots the loss as your model trains.

    Outputs will be saved to the folder `experiments/`. You can change the name of the folder a given run saves to by changing the parameter called `model_name`. Be careful about overwriting folders; if you forget to change the name and perform a run with identical hyperparameters, your previous run will be overwritten!

2. Train a 2-layer neural network on the Iris Dataset while varying the following hyperparameters.

    - Learning rate
    - Hidden layer size

    You must try at least 4 different *combinations* of these hyperparameters. Report the results of your exploration, including the values of the parameters you explored and which set of parameters gave the best test error. Provide plots showing the loss versus iterations and report your final test error.

3.3 Kaggle Competition: Learning to Detect the Higgs Boson with Deep Fully-Connected Neural Networks

Now you will implement a "deep" fully-connected neural network with an arbitrary number of hidden layers. You will be training on the Higgs dataset, an open-source dataset generated from simulations of particle collisions. Each data point is a collision event, and is represented as a vector of 28 state variables. In some of these events, Higgs bosons were created. Your objective is to distinguish these events from all others, which we classify as "background noise." The first 21 features are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features derived by physicists to help discriminate between the two classes. See the dataset webpage and the original paper for more details. You will use the version of the dataset provided in the homework folder, which consists of a subset of 500,000 training examples and 50,000 validation examples. The dataset contains raw features that have not been normalized or standardized.

Instructions

1. Train a multi-layer fully-connected network on the Higgs dataset, adjusting the number of layers, hidden units, and hyperparameters to improve your accuracy. In your write-up, describe all architectures and hyperparameters you have tried and report which combination works best.

NOTE: You may wish to implement and use the sigmoid function as the output activation for this network, which is an option because there are only two classes. The backward pass of the sigmoid is considerably faster than the softmax, so this will improve your training efficiency.

You are also welcome to implement additional methods that might improve your test accuracy, such as a dropout layer, batch normalization, or any other technique that you think might improve your score. This is not a requirement, but might give you a boost!

2. Run your best model on the test data using the `model.test_kaggle()` method. This will generate a file called `kaggle_predictions.csv`, which you can upload to Kaggle. Include your Kaggle display name and your public scores in your writeup.

4 Recurrent Neural Networks

A feed-forward neural network assumes that every datapoint is independent from each other. Shuffling and reordering the data makes no difference to a feed-forward neural network. What if we have data that has temporal structure, such as music or language? We would want the word that occurs at time t to influence our prediction of the word that comes at time $t + 1$.

Recurrent neural networks (RNNs) were invented for exactly this reason. The earliest recurrent neural networks were invented in different papers in the '80s and '90s by John Hopfield, Jeffrey Elman, Sepp Hochreiter & Jurgen Schmidhuber, and Berkeley's own Prof. Michael Jordan. The simple RNN we will be implementing in this homework is the one proposed by Elman, who was a cognitive scientist at UC San Diego.

Music, language, and other temporal sequences are difficult to model because they contain *long-term dependencies*. That is, what happens at time t might effect what happens downstream at time $t + 4$ or $t + 20$. To capture these temporal dependencies, researchers realized that we could incorporate variables in our neural networks that are "carried over" between time steps. This can be achieved with *recurrent* (feedback) connections. In an ordinary feed-forward network, output is fed into downstream layers only. In a recurrent network, output is additionally fed back into the layer itself. Data from a temporal stream are fed into the network one at a time. The output at time t contributes to a "state variable" in the layer that persists through time. Thus, the output at time t potentially influences the layer's predictions at time $t + 4$ or $t + 20$.

We will use the following notation when defining recurrent layers, with superscripts surrounded by brackets indexing layers and subscripts indexing *timesteps*.

- x_t: A single data vector at timestep t, of shape $1 \times d$, where d is the number of features.
- y: A single label vector, of shape $1 \times k$, where k is the number of classes (for a classification problem), or the number of output features (for a regression problem).
- t: The current timestep
- $n^{[l]}$: The number of neurons in layer l.
- $s_t^{[l]}$: The output ("state") of layer l at timestep t. This is a vector of dimension $n^{[l]}$.
- $W^{[i]}$: A matrix of weights connecting layer $l - 1$ with layer l, of shape $n^{[l-1]} \times n^{[l]}$
- $U^{[i]}$: A matrix of weights connecting the state at the previous timepoint $s_{t-1}^{[l]}$ to the state at the current timestep $s_t^{[l]}$, of shape $n^{[l]} \times n^{[l]}$

- $b^{[l]}$: The bias vector for layer l, of shape $n^{[l]}$
- $\sigma^{[l]}(\cdot)$: The nonlinear "activation function" applied at layer l.

An Elman recurrent layer l is defined by the function $\rho^{[l]}(\cdot,\cdot)$, computed at each timestep t,

$$\rho^{[l]}(x_t, s_{t-1}^{[l]}) = \sigma^{[l]}(x_t W^{[l]} + s_{t-1}^{[l]} U^{[l]} + b^{[l]}) = s_t^{[l]}. \tag{1}$$

The state variable s is initialized with zeros at the first step. In this problem, you will only be dealing with recurrent neural networks with a single recurrent layer followed by a single fully-connected layer, as in the figure below.

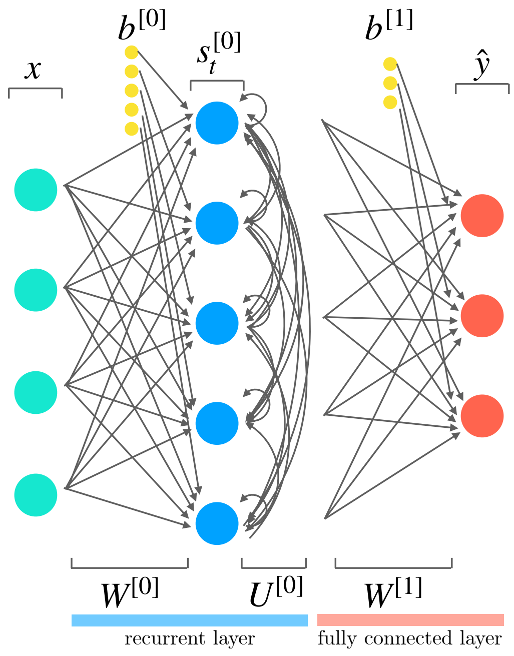

Figure 5: A two layer recurrent neural network.

The recurrent layer iterates through all timesteps in the data before passing the output to the fully-connected layer. In the backward pass of a recurrent network, we use a variant of the backpropagation algorithm called **backpropagation through time**. Just like in the ordinary backpropagation algorithm, errors are backpropagated through activation functions and layers. But when the error gets to the recurrent layer, it must also backpropagate through each timestep.

In this problem, you will implement the forward and backward passes for the Elman layer and train a two-layer Elman Recurrent Neural Network to learn to predict the next timestep of a sinusoidal function. Each data sample is a randomly shifted sinewave that extends for t timesteps. The labels are scalar values—the next value in each sinewave. Since we have added a time dimension to our data, we will feed the network a data *3-tensor* X, where the first dimension is the number of samples m in the minibatch, the second dimension is the number of features d, and the third dimension is the number of timesteps t in the sample. The labels are stored in a vector Y of shape $m \times 1$.

The sinewave dataset poses a *regression* rather than classification problem, so the activation function for the output layer should be linear. We will also need to use the *squared error* (ℓ_2) loss:

$$L = (y - \hat{y})^2$$

where y is a vector of ground-truth labels and $\hat{y}$ is the network's predictions. For a mini-batch of m samples, we use the mean squared error cost:

$$J = \frac{\sum_{i=0}^{m}(Y_i - \hat{Y}_i)^2}{m}$$

Instructions

1. Implement the forward and backward passes of the mean squared error cost in `losses.py`
2. Derive the gradient of the downstream loss with respect to the parameters of the Elman layer W, U, and b, the state at time t, and the input, x.
3. Appropriately initialize the layer parameters, gradients, and cache, and implement the forward and backward passes of the Elman layer in `layers.py`. The forward pass is broken up into two methods: `forward_step` and `forward`. The method `forward_step` implements one step of forward propagation, i.e., performs the computation described in Equation (1). The method `forward` takes in the entire design matrix and performs forward propagation for t time steps using the helper method `forward_step`. Include your code here. Remember to iterate backwards through timesteps in the backward pass.
4. Attach a screenshot of the output of gradient checking for the recurrent layer.
5. Train a two layer recurrent neural network using the script `train_rnn.py`. Your network should consist of a single Elman layer followed by a fully connected layer. Note: do not attempt to stack recurrent layers. Use just one. We recommend using the hyperparameters provided in the script, but you are welcome to explore. Include the plot of training and validation loss as well as your score on the test data in your writeup.

5 Convolutional Neural Networks (CNNs)

With fully-connected networks, we represent every datapoint as a 1-dimensional vector. It is generally assumed that dimension d is independent from dimension $d + 1$; there's no inherent relationship between different dimensions of the vector. But what if we want to classify *images*? Some of these assumptions break down. Images are inherently 2-dimensional. And there are dependencies between neighboring pixels; if you see part of an image containing a line oriented at 45 degrees, you can probably fill in the rest of the image, extending that line through the 2D plane. To capture these properties, we will need to switch from representing datapoints as 1D vectors to a format that includes 2 spatial dimensions. More generally, we will represent images as *3-tensors* with a third dimension that captures the number of "channels" in the image. For color images, we typically use 3 channels—red, green, and blue (RGB).

We'll also want the "features" (weights/filters) learned by our network to have 2 spatial dimensions, so that we can detect things like circles and eyes and faces. If our input is an $d_1 \times d_2$ image X, we could imagine building a network where our weights in a given layer are stored in a tensor of shape $d_1 \times d_2 \times c \times n$, where n is the number of neurons in that layer and c is the number of channels. But we will quickly face a combinatorial explosion in the number of weights needed to learn useful features from natural images. Imagine that one of the weights learns to represent a cat in the top left corner of the image. What if our dataset includes images with cats in the bottom right corner as well? The top-left-cat feature will be useless for those images and the network would have to dedicate a different weight to representing bottom-right-cat. But it's worse than that. If cats could be expected to appear in any arbitrary location in the image, the network would need to learn a separate weight for every possible position. It would also have to do this for every possible feature that might be needed for the task at hand, such as human faces or hands or buildings. Rapidly, we face a combinatorial explosion.

We can avoid this problem by allowing the weights of the network to be *translation invariant*, conforming the structure of the neural network architecture to the translation invariant structure of natural images. **Convolutional neural networks** do exactly this. Convolutional neural networks were inspired by models of the visual cortex. In the classical model of the visual cortex, each neuron responds to a particular feature in a particular region of the visual field, called the neuron's "receptive field". Information processing in the visual cortex is hierarchical. Neurons in regions of the brain involved in early visual processing extract simple features such as dots and oriented straight lines. As we move up towards later stages of processing, we find neurons representing more complex features, such as curves and crosses, and in the highest areas of visual processing, we find neurons that are selective for faces and other objects. These more complex features are computed as compositions of the simpler features represented in earlier stages. As we move through the visual hierarchy, the size of each neuron's receptive field increases as well, with highly localized features represented in early stages and larger features that dominate most of the visual scene represented in later stages.

Convolutional neural networks achieve these properties by incorporating the following.

- **Convolutional filters**: The weights of a convolutional network are typically referred to as *filters* or *kernels*. In a convolutional network, filters with 2 spatial dimensions (generally smaller than the original image) are *convolved* with the image. This is often referred to as "weight sharing," as the same weights are applied across many different locations in the image.
- **Pooling layers**: Convolutional neural networks typically incorporate layers that downsample the image so that later layers represent the image at a coarser level of resolution, mimicking the increase in receptive field size observed in biological brains.
- **Deep, hierarchical processing**: The convolutional networks used in state-of-the-art image processing are typically very deep (on the order of 15–25 layers). This allows the network to buld up complex features as compositions of simpler features.

Remarkably, visualizations of features learned by a convolutional neural network bear resemblance to many of the features observed in biological neurons in visual cortex, such as oriented lines and curves. If you're interested in understanding the representations learned by convolutional neural networks trained on images, I highly recommend checking out this work: `https://distill.pub/2017/feature-visualization/`

We will use the following notation when defining a convolutional neural network layer.

- X: A single image tensor, of shape $1 \times d_1 \times d_2 \times c$, where d_1 and d_2 are the spatial dimensions, and c is the number of channels.
- y: A single label vector, of shape $1 \times k$, where k is the number of classes.
- $n^{[l]}$: The number of neurons in layer l.
- $(k_1, k_2)^{[l]}$: The size of the spatial dimensions of the filters in layer l. Also referred to as the kernel size.

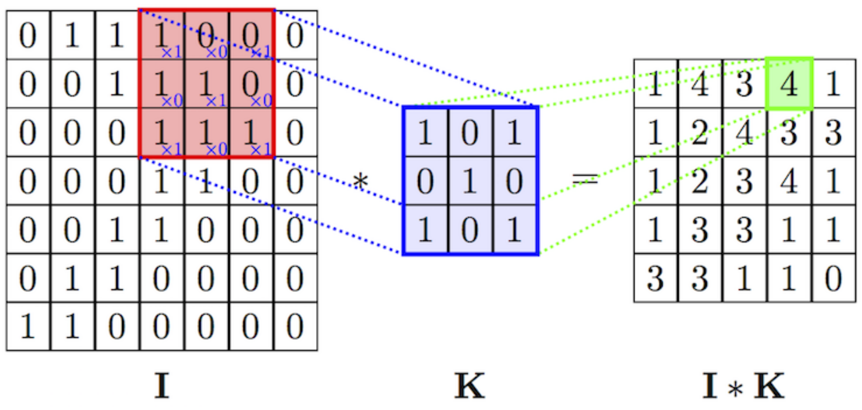

Figure 6: Figure showing an example of one convolution.

- $W^{[l]}$: The tensor of filters convolved at layer l. This tensor has shape $k_1 \times k_2 \times n^{[l-1]} \times n^{[l]}$.
- $b^{[l]}$: The bias vector for layer l, of shape $1 \times n^{[l]}$.
- $H^{[l]}$: The output of layer l. This is a tensor of shape $1 \times r_1 \times r_2 \times n^{[l]}$, where (r_1, r_2) is the shape of output of the convolution operation. Below we will discuss how to calculate this.
- $\sigma^{[l]}(\cdot)$: The nonlinear "activation function" applied at layer l.

In a convolutional layer, each filter is convolved with the input image, across every image channel. This operation is, essentially, a sliding sum of element-wise products. Figure 6 gives a visual example. To compute a single element in the intermediate output Z, for a single neuron n and a single channel c in the input X, we compute

$$Z[d_1, d_2, n] = (X * W)[d_1, d_2, n] = \sum_i \sum_j \sum_c W[i, j, c, n] X[d_1 + i, d_2 + j, c] + b[n].$$

Please note that the formula above is the **cross-correlation** formula from signal processing and NOT the convolution formula. Nevertheless this is what ML people call convolution and so will we. It actually makes sense to use cross-correlation instead of using convolution because the former can be interpreted as producing an output which is higher at locations where the image has the pattern in the filter and low elsewhere. Furthermore, convolution is the same as cross-correlation with a flipped filter, and we learn the filters, so it should not make any difference operationally whether you implement convolution or cross-correlation. However, in order to pass the tests, you must implement **cross-correlation** and call that convolution because that's how we do it in ML-land.

In this equation, we drop the layer superscripts for clarity, and index elements of the matrices in brackets. The output of this operation is what we call a "feature map," which essentially captures the strength of each filter at every region in the image. In the equation above, we slide the filter over the image in increments of one pixel. We can choose to take a larger steps instead. The size of the step taken in the convolution operation is referred to as the *stride*.

The output of the convolutional layer is

$$H^{[l]} = \sigma^{[l]}(Z^{[l]}).$$

In this problem, you will write the forward and backward passes of a convolutional neural network layer. Convolutional neural networks take considerably longer to train than the feedforward and recurrent networks we have built, and the `numpy` implementation you will write here will be impractically slow. So, you will not be asked to train your network. Instead, we will simply run tests on the forward and backward passes of your network.

Instructions

1. Fill in the forward and backward passes of the `Conv2D` layer in `layers.py`. In your writeup, provide a screenshot of or otherwise include your code. **IMPORTANT: DO NOT change `forward_faster` or `backward_faster`.**

2. Verify the correctness of your convolutional layer implementation by running the notebook `check_conv.ipynb`. Please attach a screenshot of the output of that notebook here.

3. Check your layer's gradients in the notebook `check_gradients.ipynb`. Provide a screenshot of your results.