CS 189:289A Introduction to Machine Learning
Fall 2021 Jennifer Listgarten and Jitendra Malik

HW6

Due 11/30 at 11:59pm

- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted for the written questions.
- In all of the questions, **show your work**, not just the final answer.
- **Important:** each part of Problem 1 and Problem 2(a) are worth only one point each, and each subsequent part is worth two points each. All parts will still be self-graded on a two point scale, and rescaling of points will happen after.

Deliverables:

1. Submit a PDF of your homework to the Gradescope assignment entitled "HW6 Write-Up". **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. This should be on its own page and should be the first page that you submit.
   - 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."*
   - **Replicate all your code in an appendix**. Begin code for each coding question in a fresh page. Do not put code from multiple questions in the same page. When you upload this PDF on Gradescope, *make sure* that you assign the relevant pages of your code from appendix to correct questions.

1 Fundamentals of Convolutional Neural Networks

(a) **1D Convolution**

Consider a single convolutional layer, where your input is a sequence of length 200. In other words, the input is a 200×1 tensor. Your convolution has:

- Filter size: 4×1
- Filters: 8
- Stride: 2
- No padding

What is the number of parameters (weights) in this layer, including a bias term?

What is the shape of the output tensor?

(b) **2D Convolution**

Consider a single convolutional layer, where your input is a 28×28 pixel, RGB image. In other words, the input is a $28 \times 28 \times 3$ tensor. Your convolution has:

- Filter size: $4 \times 4 \times 3$
- Filters: 8
- Stride: 2
- No padding

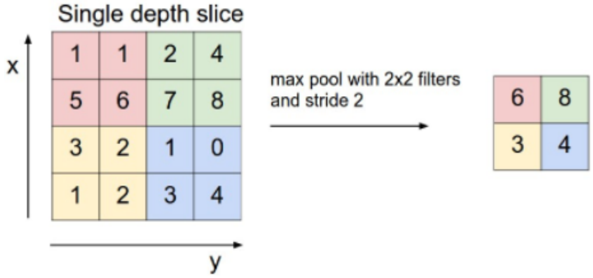
What is the number of parameters (weights) in this layer, including a bias term?

What is the shape of the output tensor?

(c) **Max/Average Pooling**

Pooling is a downsampling technique for reducing the dimensionality of a layer's output. Pooling iterates across patches of an image similarly to a convolution, but pooling and convolutional layers compute their outputs differently: given a pooling layer B with preceding layer A, the output of B is some function (such as the max or average functions) applied to patches of A's output.

Below is an example of max-pooling on a 2-D input space with a 2×2 filter (the max function is applied to 2×2 patches of the input) and a stride of 2 (so that the sampled patches do not overlap):



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

6	8
3	4

Average pooling is similar except that you would take the average of each patch as its output instead of the maximum.

Consider the following two matrices:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

Apply 3×3 average pooling with a stride of 2 to each of the above matrices.

Apply 3×3 max pooling with a stride of 2 to each of the above matrices.

(d) Consider a scenario in which we wish to classify a dataset of images that contain different shapes, taken at various angles/locations and containing small amounts of noise (e.g. some pixels may be missing). Why might pooling be advantageous given these distortions in our dataset? Discuss both max and average pooling.

2 MNIST: Hand-written digit classification

MNIST is a classic dataset in computer vision. It consists of images of handwritten digits (0 - 9) and the correct digit classification. In this problem you will implement deep networks using PyTorch to classify MNIST digits. Specifically, you will compare fully-connected neural networks, convolutional neural networks, and Transformers, and explore how neural network architectures impact the classification accuracy.

For this assignment, we will be using Google Colab (`https://colab.research.google.com/`) for the coding portions. Colab is convenient because the PyTorch package should already be available, and you can access a GPU through Colab. For all parts of this assignment, the neural network training should take from a few minutes up to an hour without a GPU. You can use a GPU by selecting the setting from "Runtime > Change runtime type".

The starter code is available here:
`https://colab.research.google.com/drive/1v9bAvb2Ohga9VuH5Zo-xTHqbUByyBtQd`
This Colab should be viewable through your UC Berkeley account. Click "File > Save a copy in Drive" to make a copy of this Colab and start your work. You can find where the copy is saved via "File > Locate in Drive".

(a) Load the MNIST dataset using the provided sample code.

Image inputs in PyTorch are generally 3D tensors with the shape (# channels, height, width). Examine the input data. What are the height and width of the images? What do the values in each array index represent? How many images are in the training set? How many are in the testing set? You can use the `imshow` function in matplotlib if you'd like to see the actual pictures (see the sample code).

(b) **Fully-connected neural networks**

We will start with fully-connected neural networks. You will be expected to design and implement a deep network that meets some performance baseline on the MNIST dataset. Write your code under the section titled "Your code: 2 (b)".

Using PyTorch's `nn.Sequential` helper class, build a deep network to classify the handwritten digits. You may **only** use the following layers, all starting with `nn.`:

- `Linear`: A fully-connected layer
- `ReLU` (activation): Sets negative inputs to 0
- `Softmax` (activation): Rescales input so that it can be interpreted as a (discrete) probability distribution.
- `Dropout`: Takes some probability and at every iteration sets weights to zero at random with that probability (effectively regularization)

You may use other PyTorch builtin functions, but not anything else from the `nn` module.

A sample linear model is in the Colab notebook. There is also sample code showing how to train a model. For the hand-written digit classification problem, we use categorical cross entropy as the loss function. There are also a number of optimizers available in PyTorch. Adam is a popular one and will probably work well.

Your task. Using at most 200 hidden units (in total, adding together the number of hidden units in all hidden layers), build a network using only the allowed layers that achieves test accuracy of at least 0.975. You also should find the batch size and number of epochs that give you the best results (default batch size and epochs are 32 and 10, respectively). (Note: activations, `Dropout`, and your last `Linear` layer do not count toward your hidden unit count, because the final layer is "observed" and not *hidden*.)

Deliverables. Turn in the code of your model as well as the best test accuracy that it achieved. Report the best model architecture you found and report all hyperparameters. Plot the loss trajectories over the course of training on both the training set and the validation set.

In addition to the best model, also document what you tried along the way. Describe what worked and what didn't work.

(c) **Convolutional networks**

Using PyTorch's `nn.Sequential` class as you did previously, build a deep *convolutional* network to classify the handwritten digits in MNIST. You are allowed to use the following layers (but **only** the following):

- `Linear`: A fully-connected layer
  - In convolutional networks, Linear (also called dense) layers are typically used to knit together higher-level feature representations.
  - Particularly useful to map the 2D features resulting from the last convolutional layer to categories for classification (like the 1000 categories of ImageNet or the 10 categories of MNIST).
  - Inefficient use of parameters and often overkill: for A input activations and B output activations, number of parameters needed scales as $O(AB)$.
- `Conv2d`: A 2-dimensional convolutional layer
  - The bread and butter of convolutional networks, conv layers impose a translational-invariance prior on a fully-connected network. By sliding filters across the image to form another image, conv layers perform "coarse-graining" of the image.
  - Networking several convolutional layers in succession helps the convolutional network knit together more abstract representations of the input.
  - More efficient use of parameters. For N filters of $K \times K$ size on an input of size $L \times L$, the number of parameters needed scales as $O(NK^2)$. When N, K are small, this can often beat the $O(L^4)$ scaling of a Linear layer applied to the L^2 pixels in the image.
- `MaxPool2d`: A 2-dimensional max-pooling layer
  - Another way of performing "coarse-graining" of images, max-pool layers are another way of ignoring finer-grained details by only considering maximum activations over small patches of the input.

  - Drastically reduces the input size. Useful for reducing the number of parameters in your model.
  - Typically used immediately following a series of convolutional-activation layers.
- `BatchNorm2d`: Performs batch normalization (Ioffe and Szegedy, 2014). Normalizes the activations of previous layer to standard normal (mean 0, standard deviation 1).
  - Accelerates convergence and improves performance of model, especially when saturating nonlinearities (such as sigmoids) are used.
  - Makes model less sensitive to higher learning rates and initialization, and also acts as a form of regularization.
  - Typically used immediately before nonlinearities.
- `Dropout`: At every iteration, sets weights to zero at random with a specified probability
  - An effective form of regularization. During training, randomly selecting activations to shut off forces network to build in redundancies in the feature representation, so it does not rely on any single activation to perform classification.
- `ReLU` (activation): Sets negative inputs to 0
- `Softmax (activation):` Rescales input so that it can be interpreted as a (discrete) probability distribution.
- `Flatten`: Flattens any tensor into a single vector (required in order to pass a 2D tensor output from a convolutional layer as input into Linear layers)

You may use other PyTorch builtin functions, but not anything else from the `nn` module.

Your tasks. Build a network with only the allowed layers that achieves **test accuracy of at least 0.985**. Use categorical cross entropy as your loss function and train with the Adam optimizer for 10 epochs with a batch size of 32 (do not tune these hyperparameters). Note: your model must have **fewer than 50,000 parameters**, as measured by the method given in the sample code. It is also possible to achieve the target test accuracy of 0.985 with fewer than 10,000 parameters.

Everything else can change: initial learning rates, dropout probabilities, layerwise regularizer strengths, etc. You are not required to use all of the layers, but *you must have at least one dropout layer and one batch normalization layer in your final model*. Try to figure out the best possible architecture and hyperparameters given these building blocks.

One efficient way to design your model and select hyperparameters is to train your model for 1 epoch (batch size 32) and look at the validation accuracy. This should take no more than 10 minutes, and should give you an immediate sense for how fast your network converges and how good it is.

Deliverables. Turn in the code of your model as well as the best test accuracy that it achieved. Report the best model architecture you found and report all hyperparameters. Plot the loss trajectories over the course of training on both the training set and the validation set. We should have everything needed to reproduce your results.

In addition to the best model, also document what you tried along the way. Describe what worked and what didn't work.

Discuss what you found to be the most effective strategies in designing a convolutional network. Which regularization method was most effective (dropout, layerwise regularization, batch norm)?

Hints:

- You are provided with a sample network that achieves a decent accuracy. Starting with this network, modify some of the regularization parameters (layerwise regularization strength, dropout probabilities) to see if you can maximize the validation accuracy. You can also add layers or modify layers (e.g. changing the convolutional kernel sizes, number of filters, stride, dilation, etc.) so long as the total number of parameters remains under the cap.
- You may want to read up on successful convolutional architectures, and emulate some of their design principles. **Please cite any idea you use that is not your own.**
- To better understand the function of each layer, check the PyTorch documentation.
- If your model is running slowly, try making each layer smaller and stacking more layers so you can leverage deeper representations.
- A relatively standard CNN design blueprint is as follows:
  - CNNs perform well with many stacked convolutional layers, which develop increasingly large-scale representations of the input image.
  - Dropout ensures that the learned representations are robust to some amount of noise.
  - Batch norm is typically done after a convolutional or dense layer and immediately prior to an activation/nonlinearity layer.
  - Max-pooling is typically done after a series of convolutions, in order to gradually reduce the size of the representation.
  - Finally, the learned representation is passed into a dense layer (or two), and then filtered down to the final softmax layer.

3 Vision Transformer

Transformers applied directly to sequences of image patches can also perform very well on image classification tasks. Vision Transformer (ViT) (`https://arxiv.org/abs/2010.11929`) attains results comparable to state-of-the-art convolutional networks while requiring fewer computational resources. Here we explore applying Vision Transformer to the same handwritten digit classification task on the MNIST dataset.

First, we start with the building blocks for Transformer models.

(a) **Scaled dot-product attention**

Scaled dot-product attention (`https://arxiv.org/abs/1706.03762`) is an attention mechanism where the dot products are scaled down by $\sqrt{d_k}$, the squared root of the query dimension. Formally for a sequence of length T, we have a query matrix $Q \in \mathbb{R}^{T\times d_k}$, a key $K \in \mathbb{R}^{T\times d_k}$ and a value $V \in \mathbb{R}^{T\times d_v}$ and calculate the attention matrix as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The resulting attention matrix takes the following form as described in the lecture:

$$\begin{bmatrix} - & a_1^T & - \\ - & a_2^T & - \\ & \cdots & \\ - & a_T^T & - \end{bmatrix},$$

where each vector a_i describes how the i-th position in the sequence attends to all the other positions. For example a_{ij} is the relative attention weight for the i-th position attending to the j-th position.

The query Q, the key K, and the value V are each projected from the hidden state through different projection matrices (no bias in the projections). In general, d_k and d_v could be different dimensions, but in the scope of this problem they will be the same (`inner_dim` in the Colab notebook).

Deliverables. Implement (single-head) scaled dot-product attention as a module in PyTorch based on the template in the code sample, and include your code in your writeup. You may **only** use the following PyTorch functions that start with `nn.`:

- `Linear`: A fully-connected layer
- `Softmax` (activation): Rescales input so that it can be interpreted as a (discrete) probability distribution.
- `Dropout`: Takes some probability and at every iteration sets weights to zero at random with that probability (effectively regularization)

You may use other PyTorch builtin functions, such as `torch.einsum`, but not anything else from the `nn` module. For matrix operations, you may also find `torch.matmul` and `torch.transpose` useful. Note that, in the call to `forward`, `x` will have an additional mini-batch dimension in the front that your implementation needs to handle.

(b) **Transformer layers**

The implementation for multihead attention is provided to you and uses your implementation from part (a). In PyTorch, layer normalization is available as `torch.nn.layernorm`. Putting together these building blocks, implement the Transformer encoder layer as a PyTorch module, based on the template provided in the Colab notebook.

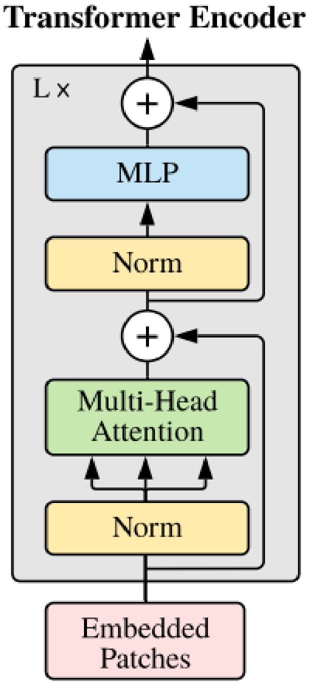


Figure 1: Each layer in the Transformer encoder consists of layer normalization, multi-head attention, another layer normalization, and a fully-connected MLP.

Residual connections. Within each layer, we are applying residual connections twice (as illustrated by "+" in Figure 1), once for multihead attention and once for the MLP.

The basic idea behind residual connections is to make each layer close to the identity map. Traditional building blocks of neural networks typically look like an affine transformations A with a nonlinearity in the middle:

$$f(x) = \text{ReLU}(A(x))$$

Residual networks modify these building blocks by adding the input x back to the output:

$$h(x) = x + B(\text{ReLU}(A(x)))$$

In this example, say x is of input dimension d_i, and A is an affine transformation that maps the input to a hidden vector of dimension d_h, *i.e.*, $A : \mathbb{R}^{d_i} \to \mathbb{R}^{d_h}$. Then, B is an affine transformation that maps back to the input dimension, *i.e.*, $B : \mathbb{R}^{d_h} \to \mathbb{R}^{d_i}$, so that $B(\mathrm{ReLU}(A(x)))$ has the same dimension as the input x.

Thinking about the computation graph, we create a connection from the input to the output of the building block that skips the transformation. Such connections are therefore called *skip connections*. This seemingly innocuous change was hugely successful and allowed for model depths not seen previously.

Deliverables. Turn in the code you implemented for the encoder layer module. Remember to include layer normalization, dropout, and residual connections at appropriate places. See Figure 1 for the illustration of each layer in the Transformer encoder. Note that dropout is used after the attention, inside the MLP, and after the MLP.

(c) **Vision Transformer on MNIST**

Using the transformer layer module implemented above, train the Vision Transformer model on MNIST for 10 epochs using batch size 32 (see provided Vision Transformer code in the Colab notebook). For the Vision Transformer, use 6 Transformer layers each with 8 heads, patch size 7, query dimension 64, and MLP hidden dimension 128.

Deliverables. What test accuracy does this verison of the Vision Transformer model achieve? Plot the loss trajectories on both the training set and the validation set.

It should take 30-60 minutes to train the Vision Transformer model and the accuracy should be roughly comparable with the convolutional neural network (± 1%).

For the first example in the validation set, plot the (post-softmax) attention weights as a heat map for each of the heads in the last layer. There should be 8 heat maps for the 8 heads, and each heat map should be a square.