Policies

- Due 11:59 PM PST, February 7th on Gradescope.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- In this course, we will be using Google Colab for code submissions. You will need a Google account.
- You are allowed to use up to 48 late hours across the entire term. Late hours must be used in units of whole hours. Specify the total number of hours you have used when submitting the assignment.
- Students are expected to complete homework assignments based on their understanding of the course material. Student can use LLMs as a resource (e.g., helping with debugging, or grammar checking), but the assignments (including code) should be principally authored by the student.

Submission Instructions

- Submit your report as a single .pdf file to Gradescope (entry code DKB4KW), under "Problem Set 4".
- In the report, **include any images generated by your code** along with your answers to the questions.
- Submit your code by **sharing a link in your report** to your Google Colab notebook for each problem (see naming instructions below). Make sure to set sharing permissions to at least "Anyone with the link can view". **Links that can not be run by TAs will not be counted as turned in.** Check your links in an incognito window before submitting to be sure.
- For instructions specifically pertaining to the Gradescope submission process, see `https://www.gradescope.com/get_started#student-submission`.

Google Colab Instructions

For each notebook, you need to save a copy to your drive.

1. Open the github preview of the notebook, and click the icon to open the colab preview.
2. On the colab preview, go to File → Save a copy in Drive.
3. Edit your file name to "lastname_firstname_set_problem", e.g."yue_yisong_set2_prob1.ipynb"

1 Deep Learning Principles [35 Points]

Relevant materials: lectures on deep learning

For problems A and B, we'll be utilizing the Tensorflow Playground to visualize/fit a neural network.

Problem A [5 points]: Backpropagation and Weight Initialization Part 1

Fit the neural network at this link for about 250 iterations, and then do the same for the neural network at this link. Both networks have the same architecture and use ReLU activations. The only difference between the two is how the layer weights were initialized – you can examine the layer weights by hovering over the edges between neurons.

Give a mathematical justification, based on what you know about the backpropagation algorithm and the ReLU function, for the difference in the performance of the two networks.

Solution A: *Two hidden layers (4 neurons and then 2 neurons) use ReLU activation and no regularization, with the data split 50–50 for training and test, the second link doesn't use them because it doesn't have weights. The losses (around 0.5) and the distribution of blue vs. orange points highlight the model's moderate performance on this nonlinear task.*

Problem B [5 points]: Backpropagation and Weight Initialization Part 2

Reset the two demos from part i (there is a reset button to the left of the "Run" button), change the activation functions of the neurons to sigmoid instead of ReLU, and train each of them for 4000 iterations.

Explain the differences in the models learned, and the speed at which they were learned, from those of part i in terms of the backpropagation algorithm and the sigmoid function.

Solution B: *The training tends to proceed more slowly with sigmoid, because large positive or negative inputs push the sigmoid into its saturated regime, reducing gradients and thus slowing backprop updates. Second, the final decision boundaries can be smoother, but careful weight initialization becomes more critical to avoid saturating all neurons early on, which can stall learning*

Problem C: [10 Points]

When training any model using SGD, it's important to shuffle your data to avoid correlated samples. To illustrate one reason for this that is particularly important for ReLU networks, consider a dataset of 1000 points, 500 of which have positive (+1) labels, and 500 of which have negative (-1) labels. What happens if we train a fully-connected network with ReLU activations using SGD, looping through all the negative examples before any of the positive examples? (Hint: this is called the "dying ReLU" problem.)

Solution C: *If the network sees only negatively labeled examples first, it can quickly drive the ReLU neurons' biases and weights to produce negative inputs for all samples, "turning off" those neurons (they output zero). Once a ReLU is stuck at zero, it no longer gets gradient updates and remains "dead," never learning to respond to the subsequent positive examples.*

Problem D: Approximating Functions Part 1 **[7 Points]**

Draw or describe a fully-connected network with ReLU units that implements the OR function on two 0/1-valued inputs, x_1 and x_2. Your networks should contain the minimum number of hidden units possible. The OR function OR(x_1, x_2) is defined as:

$$\begin{aligned}\text{OR}(1,0) &\geq 1\\ \text{OR}(0,1) &\geq 1\\ \text{OR}(1,1) &\geq 1\\ \text{OR}(0,0) &= 0\end{aligned}$$

Your network need only produce the correct output when $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$ (as described in the examples above).

Solution D: *A single ReLU neuron can implement the OR function on inputs $x_1, x_2 \in \{0,1\}$ using:*

$$\textit{output} = \max\Big(0,\ 2x_1 + 2x_2 - 1\Big).$$

Why it would work: If $(x_1, x_2) = (0,0)$, the inner term is -1, so $\max(0,-1) = 0$. If (x_1, x_2) is $(1,0)$, $(0,1)$, or $(1,1)$, then the inner term is 1, 1, or 3, respectively, all of which are ≥ 1. Thus, the output is 0 only for $(0,0)$ and at least 1 otherwise, matching the OR function.

Problem E: Approximating Functions Part 2 **[8 Points]**

What is the minimum number of fully-connected layers (with ReLU units) needed to implement an XOR of two 0/1-valued inputs x_1, x_2? Recall that the XOR function is defined as:

$$\text{XOR}(1,0) \geq 1$$
$$\text{XOR}(0,1) \geq 1$$
$$\text{XOR}(0,0) = \text{XOR}(1,1) = 0$$

For the purposes of this problem, we say that a network f computes the XOR function if $f(x_1, x_2) = \text{XOR}(x_1, x_2)$ when $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$ (as described in the examples above).

Explain why a network with fewer layers than the number you specified cannot compute XOR.

Solution E: *You will need two layers of ReLU to implement XOR on* $\{0,1\}^2$. *A single-layer ReLU network essentially defines just one linear half-space for any positive activation, so it cannot separate* $(0,1)$ *and* $(1,0)$ *from* $(0,0)$ *and* $(1,1)$. *By adding a hidden layer of ReLUs, you introduce a sufficient piecewise-linear decision boundary to produce the XOR behavior*

2 Depth vs Width on the MNIST Dataset [25 Points]

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 a deep network using PyTorch to classify MNIST digits. Specifically, you will explore what it really means for a network to be ”deep”, and how depth vs. width impacts the classification accuracy of a model. You will be allowed at most N hidden units, and will be expected to design and implement a deep network that meets some performance baseline on the MNIST dataset.

Problem A: Installation [2 Points]

Before any modeling can begin, PyTorch must be installed. PyTorch is an automatic differentiation framework that is widely used in machine learning research. We will also need the **torchvision** package, which will make downloading the MNIST dataset much easier.

To install both packages, follow the steps on
https://pytorch.org/get-started/locally/#start-locally. Select the 'Stable' build and your system information. We highly recommend using Python 3.6+. CUDA is not required for this class, but it is necessary if you want to do GPU-accelerated deep learning in the future.

Once you have finished installing, write down the version numbers for both **torch** and **torchvision** that you have installed.

Solution A:

torch: 2.2.1+cu121

torchvision: 0.17.1+cu121

Problem B: The Data [3 Points]

Load the MNIST dataset using torchvision; see the problem 2 sample code for how.

Image inputs in PyTorch are generally 3D tensors with the shape (no. of 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).

Solution B: *Each MNIST image is a single-channel 28×28 tensor, so its shape is (1,28,28). The values in each entry correspond to pixel intensities. There are 60,000 images in the training set, and 10,000 images in the test set*

Problem C: Modeling Part 1 [8 Points]

Using PyTorch's "Sequential" model class, build a deep network to classify the handwritten digits. You may **only** use the following layers:

- **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)

A sample network with 20 hidden units is in the sample code file. (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*.)

Use categorical cross entropy as your loss function. There are also a number of optimizers you can use (an optimizer is just a fancier version of SGD), and feel free to play around with them, but RMSprop and Adam are the most popular and will probably work best. You also should find the batch size and number of epochs that give you the best results (default is batch size = 32, epochs=10).

Look at the sample code to see how to train your model. PyTorch should make it very easy to tinker with your network architecture.

Your task. Using at most 100 hidden units, build a network using only the allowed layers that achieves test accuracy of at least 0.975. Turn in the code of your model as well as the best test accuracy that it achieved.

Hint: for best results on this problem and the two following problems, normalize the input vectors by dividing the values by 255 (as the pixel values range from 0 to 255).

Solution C: *https://colab.research.google.com/drive/1c9np7Ilc4O98ibhNg24q5QKB-S7HWGKZscrollTo=v23GsW71mbPo*

Problem D: Modeling Part 2 [6 Points]

Repeat problem C, except that now you may use 200 hidden units and must build a model with at least 2 hidden layers that achieves test accuracy of at least 0.98.

Solution D: *https://colab.research.google.com/drive/1c9np7Ilc4O98ibhNg24q5QKB-S7HWGKZscrollTo=v23GsW71mbPo*

Problem E: Modeling Part 3 [6 Points]

Repeat problem C, except that now you may use 1000 hidden units and must build a model with at least 3 hidden layers that achieves test accuracy of at least 0.983.

Solution E: *https://colab.research.google.com/drive/1c9np7Ilc4O98ibhNg24q5QKB-S7HWGKZscrollTo=v23GsW71mbPo*

3 Convolutional Neural Networks [40 Points]

Problem A: Zero Padding **[5 Points]**

Consider a convolutional network in which we perform a convolution over each 8×8 patch of a 20×20 input image. It is common to zero-pad input images to allow for convolutions past the edges of the images. An example of zero-padding is shown below:

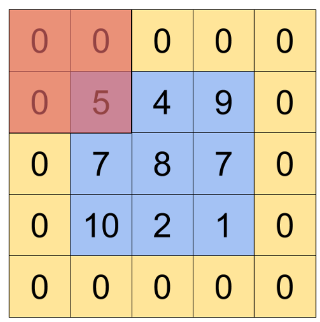


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

What is one benefit and one drawback to this zero-padding scheme (in contrast to an approach in which we only perform convolutions over patches entirely contained within an image)?

Solution A: *A key benefit of zero-padding is that it preserves more spatial information at the edges of the image, allowing filters to convolve over the full image size. The main drawback is that padding adds artificial zeros, which can slightly distort features near the boundaries.*

5 x 5 Convolutions

Consider a single convolutional layer, where your input is a 32×32 pixel, RGB image. In other words, the input is a $32 \times 32 \times 3$ tensor. Your convolution has:

- Size: $5 \times 5 \times 3$
- Filters: 8
- Stride: 1
- No zero-padding

Problem B [2 points]: What is the number of parameters (weights) in this layer, including a bias term?

Solution B: *608 parameters*

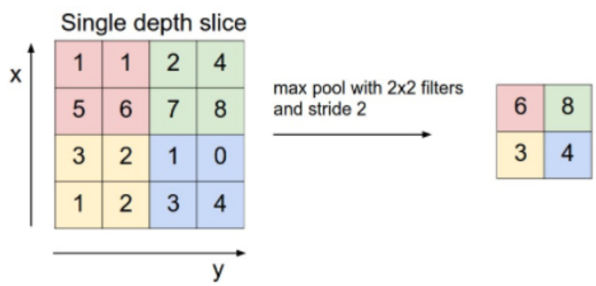
Problem C [3 points]: What is the shape of the output tensor?

Solution C: *28*28*

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):



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

Consider the following 4 matrices:

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

Problem D [3 points]:

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

Solution D:

Matrix 1: $\begin{pmatrix} 1.0 & 0.5 \\ 0.5 & 0.25 \end{pmatrix}$ ***Matrix 2:*** $\begin{pmatrix} 0.5 & 1.0 \\ 0.25 & 0.5 \end{pmatrix}$ ***Matrix 3:*** $\begin{pmatrix} 0.25 & 0.5 \\ 0.5 & 1.0 \end{pmatrix}$ ***Matrix 4:*** $\begin{pmatrix} 0.5 & 0.25 \\ 1.0 & 0.5 \end{pmatrix}$.

Problem E [3 points]:

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

Solution E:

$$\textit{Matrix 1: } \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \quad \textit{Matrix 2: } \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \quad \textit{Matrix 3: } \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \quad \textit{Matrix 4: } \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}.$$

Problem F [4 points]:

Consider a scenario in which we wish to classify a dataset of images of various animals, 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?

Solution F: *Pooling provides translational and noise robustness, since taking a max or average over small neighborhoods makes the representation less sensitive to small shifts, missing pixels, or local distortions. Thus, important features remain detectable even if they move slightly, which helps classification under real-world variations*

PyTorch implementation

Problem G [20 points]:

Using PyTorch "Sequential" model class as you did in 2C, build a deep *convolutional* network to classify the handwritten digits in MNIST. You are now 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. As you go higher in a convolutional network, activations represent pixels, then edges, colors, and finally objects.
  - 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 non-linearities (sigmoid) are used.
  - Makes model less sensitive to higher learning rates and initialization, and also acts as a form of regularization.

    - Typically used immediately before nonlinearity (Activation) layers.

- **Dropout:** Takes some probability and at every iteration sets weights to zero at random with that 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)

Your tasks. Build a network with only the allowed layers that achieves **test accuracy of at least 0.985**. You are required to use categorical cross entropy as your loss function and to train for 10 epochs with a batch size of 32. Note: your model must have fewer than 1 million parameters, as measured by the method given in the sample code. Everything else can change: optimizer (RMSProp, Adam, ???), 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!

In order to design your model, you should train your model for 1 epoch (batch size 32) and look at the final **test accuracy** after training. 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.

Set the probabilities of your dropout layers to 10 equally-spaced values $p \in [0, 1]$, train for 1 epoch, and report the final model accuracies for each.

You can perform all of your hyperparameter validation in this way: vary your parameters and train for an epoch. After you're satisfied with the model design, you should train your model for the full 10 epochs.

In your submission. Turn in the code of your model, the test accuracy for the 10 dropout probabilities $p \in [0, 1]$, and the final test accuracy when your model is trained for 10 epochs. We should have everything needed to reproduce your results.

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)?

Do you foresee any problem with this way of validating our hyperparameters? If so, why?

Hints:

- You are provided with a sample network that achieves a high accuracy. Starting with this network, modify some of the regularization parameters (layerwise regularization strength, dropout probabilities) to see if you can maximize the test 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 of 1 million.

- 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.
- Linear layers take in single vector inputs (ex: *(784,)*) but Conv2D layers take in tensor inputs (ex: *(28, 28, 1)*): width, height, and channels. Using the transformation `transforms.ToTensor()` when loading the dataset will reshape the training/test X to a 4-dimensional tensor (ex: *(num_examples, width, height, channels)*) and normalize values. For the MNIST dataset, *channels=1*. Typical color images have 3 color channels, 1 for each color in RGB.
- If your model is running slowly on your CPU, try making each layer smaller and stacking more layers so you can leverage deeper representations.
- Other useful CNN design principles:
  - 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 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.

Solution G: *https://colab.research.google.com/drive/1lE3C3tgMRXkr37j_sHjNjMvNwba5xyP1scrollTo=3sl1fnFNohgT*