Autograd: for Automatic Differentiation and for Auto-Construction of Computational Graphs

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Sunday 9th February, 2020 10:42

Preamble

Consider learning-based systems in general that must set their critical parameters by minimizing the errors they make on training data.

If we refer to the prediction error made on a training sample as “loss”, such systems must translate the loss into a change in the values of the parameters in order to continually decrease the loss as the systems ingest more and more training data.

In this lecture, we will first review the gradient descent (GD) based methods for translating loss into incremental changes to the values of the parameters. Although GD is numerically robust, it is not possible to use it directly on account of its slow convergence and its propensity to get stuck in local minima. To address those shortcomings of GD, we will talk about Stochastic Gradient Descent (SGD). That would set us up to talk about PyTorch’s Autograd module that first constructs a dynamic computational graph (CG) during the forward pass of a training sample through the network and then uses the graph during the backpropagation of the loss for the calculation of the gradients of the loss that are needed for SGD.

Outline

Outline

Loss Surfaces and the Parameter Hyperplane

A good place to start this lecture is by reviewing what you have already learned from a previous lecture by Professor Bouman.

- Let's say our training data consist of the pairs $(\mathbf{x}_i, y_i)$ where the vectors $\mathbf{x}_i, i = 1, \ldots, m$ represent the vector training data and y_i the corresponding labels.
- Let the predicted label for training sample $\mathbf{x}_i$ be y'_i.
- If we represent the class labels by, say, one-hot vectors, we can measure the loss for each sample by, say, the mean square error:

$$L_i = ||\mathbf{y}_i - \mathbf{y}'_i||^2$$

  where the bold notation $\mathbf{y}$ is the one-hot vector representation of the label y.

Loss Surfaces and the Parameter Hyperplane (contd.)

- We can now measure the total loss for all m training samples by

$$L = \sum_{1}^{m} L_i = \sum_{1}^{m} ||\mathbf{y}_i - \mathbf{y'}_i||^2$$

- If the vector $\mathbf{p}$ represents all of the learnable parameters of the system, each prediction $\mathbf{y'}_i$ will be a function of $\mathbf{p}$ and also of the input data sample $\mathbf{x}_i$. So we can say:

$$\mathbf{y'}_i = \mathbf{f}_i(\mathbf{x}_i, \mathbf{p})$$

  where we have expressed the prediction function $\mathbf{f}_i$ as a vector also since it will need to make a prediction of each element of the one-hot representation of the output vectors.

- We can now write for the overall loss:

$$L = \sum_{1}^{m} L_i = \sum_{1}^{m} ||\mathbf{y}_i - \mathbf{f}_i(\mathbf{x}_i, \mathbf{p})||^2$$

Loss Surfaces and the Parameter Hyperplane (contd.)

- Since the squared error is the sum of the sum of the squares of the error in each component of the vectors involved, we are allowed to represent the summation over all the training samples by

  $$L = ||\mathbf{Y} - \mathbf{f}(\mathbf{X}, \mathbf{p})||^2$$

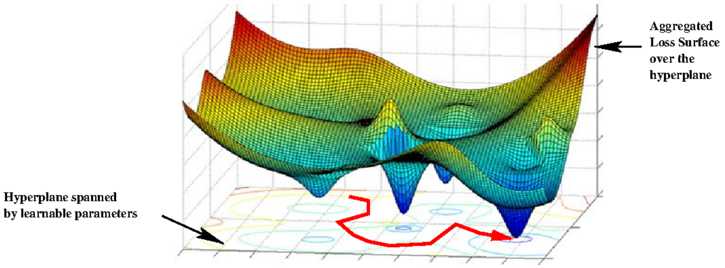
  where you can think of **Y** as us lining up all the one-hot vectors for the true labels for all the *m* training samples into one giant vector.

- By the same token, **f** also represents us lining up all the prediction vectors for each sample into a giant vector whose size is same as that of **Y**.

Loss Surfaces and the Parameter Hyperplane (contd.)

- The only unknowns in the expression for overall loss are the learnable parameters in the vector p. L is obviously a surface of some sort over the hyperplane spanned by the elements of p.
- This surface is best visualized as shown on the next slide.
- Our job is to find that point in the parameter hyperplane where loss surface has the least value.

Visualizing the Loss Surface



The 3D plot shown in the figure is from the following source: `https://www.fromthegenesis.com/gradient-descent-part1/`

Gradient Descent for Finding the Best Point in the Parameter Hyperplane

- The thing to bear in mind about the height of the surface at each point in the parameter space is that L is scalar, no different from any other scalar like, say, the temperature of the air at each point in the classroom.

- To understand the dependence of a scalar on all its defining parameters, you examine the gradient of the scalar, which is a vector that consists of the partial derivatives of the scalar with respect to each of the parameters. We denote the gradient of L at each point in the hyperplane by $\frac{\delta L}{\delta \mathbf{p}}$.

- The gradient descent (GD) of a scalar is a vector that, in the parameter hyperplane, always points in the direction in which the scalar is increasing at the fastest rate. Our goal is to head in the opposite direction since we want to reach the minimum.

Using GD Means Having to Calculate the Jacobian

- Based on the presentation made so far, you can imagine what if we are currently at point $\mathbf{p}_k$ in the parameter space, our next point in our journey to the global minimum would be at

$$\mathbf{p}_{k+1} = \mathbf{p}_k - \alpha \cdot \frac{\delta L}{\delta \mathbf{p}}$$

  where α is a small number that is called the **learning rate**.

- So using GD boils down to estimating gradient vector $\frac{\delta L}{\delta \mathbf{p}}$. If you take a derivative of L with respect to $\mathbf{p}$, one arrives at the following formula for the gradient:

$$\frac{\delta L}{\delta \mathbf{p}} = 2J_{\epsilon}^{T}(\mathbf{p})\epsilon(\mathbf{p})$$

  where $J_{\epsilon}(\mathbf{p})$ is the Jacobian and where ϵ is the prediction error given by $\epsilon = \mathbf{Y} - \mathbf{f}(\mathbf{X}, \mathbf{p})$.

Using GD Means Having to Calculate the Jacobian (contd.)

- Since the vector $\mathbf{Y}$ in $\epsilon = \mathbf{Y} - \mathbf{f}(\mathbf{X}, \mathbf{p})$ consists of the actually predicted class labels, it is constant from the standpoint of differentiation. Therefore, we can write $J_\epsilon(\mathbf{p}) = -J_\mathbf{f}(\mathbf{p})$. Substituting this in our update formula, we get

$$\mathbf{p}_{k+1} = \mathbf{p}_k + 2 \cdot \alpha \cdot J_\mathbf{f}(\mathbf{p}_k) \cdot \epsilon_k \tag{1}$$

- The Jacobian $J_\mathbf{f}(\mathbf{p})$ is given by

$$J_\mathbf{f}(\mathbf{p}) = \begin{bmatrix} \frac{\delta f_1}{\delta p_1} & \dots & \frac{\delta f_1}{\delta p_n} \\ \vdots & \vdots & \vdots \\ \frac{\delta f_M}{\delta p_1} & \dots & \frac{\delta f_M}{\delta p_n} \end{bmatrix} \tag{2}$$

[**Good to commit to memory:** Each row of the Jacobian is a partial derivative of each prediction with respect to all *n* learnable parameters. For simplicity in notation, we are pretending that the predicted class label for each training image is a scalar number. (You could assume that we represent the class labels by distinct integers.) The notation extends easily to the case when you want to consider separately the prediction of each element of the hot-vector representation of each class label.]

Using GD Means Having to Calculate the Jacobian (contd.)

- One of my reasons for explicitly defining the Jacobian in these slides is to help you understand the PyTorch's main documentation page on Autograd at:

  https://pytorch.org/docs/stable/autograd.html

- Now you know the concept of the Jacobian, that it is a matrix, and that the number of columns in this matrix is equal to the number of learnable parameters, and that the number of rows is equal to the number of training samples you want to process at a time.
- As you will soon see, the phrase "*the number of training samples you want to process at a time*" is critical as it leads to the notion of a "batch" in how the training is carried out in deep networks.

Outline

Problems with a Vanilla Application of GD

As Professor Bouman has already pointed out, there are several problems with a straightforward application of GD:

- It can be very slow to converge — especially as the solution point gets closer and closer to the optimum (because the gradients will approach zero).
- Its propensity to get stuck in a local minimum.
- When the loss surface is a long narrow valley in the vicinity of the final solution, the solution path will oscillate around the bottom "spine" of the valley as it slowly approaches the optimum.

[**NOTE:** In order to fix the slow convergence problem, the more traditional practical applications of GD involve combining GD with the estimation of Gauss-Newton (GN) jumps at each iteration. The GN jump can potentially get to the destination in one fell swoop, but it is numerically unstable. For that reason, the GN increment to the solution path is accepted only after it is evaluated for its quality. A commonly used combination GD and GN is known as the Levenberg-Marquardt (LM) algorithm. This is one of the most famous algorithms that is used in computer vision for solving all sorts of cost minimization problems. A numerically fast version of this algorithm is known as Bundle Adjustment. LM involves inverting the matrix product $J_{\mathbf{f}}^T J_{\mathbf{f}}$ which makes it infeasible for DL where you may have millions of learnable parameters.]

Outline

Stochastic Gradient Descent (SGD)

- In DL networks, the learnable parameters are estimated with a powerful variant of GD that is known as the Stochastic Gradient Descent. With SGD, the solution path is less likely to get stuck in a local minimum (which a perennial problem with GD).
- The main idea in SGD is that you only process a small number of training samples at a time — the number may be as few as one, but you are likely to get better results if the number is greater than one.
- A re-examination of the previous slides would show that everything we have done so far holds regardless of the value of m, the number of training samples.
- So let's say you have a large training dataset, is there anything to be gained by only using a small number of images at a time?

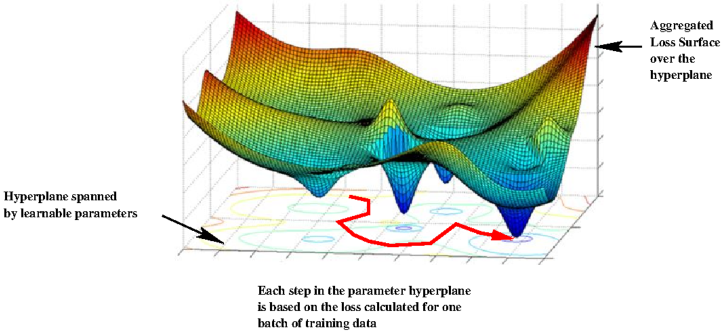
SGD (contd.)

- The question is what happens to our visualization of the solution path if only use small values like 4 or 5 for m at each iteration of training?
- The main point here is that even when m is small, while strictly speaking you are still estimating the path increment $\delta\mathbf{p}$ in the entire parameter hyperplane, *it rather unlikely that every training sample will exhibit the same "sensitivity" to each parameter in the hyperplane.*
- To elaborate, differently sized convolutional operators will be sensitive to image variations at different scales, with different color content, with different textures, etc.
- Therefore, as a training image passes through the network, depending on the scale of the features in the image, the color variations, etc., only some of the learnable parameters would "resonate" to the image.

SGD (contd.)

- Subsequently, when the loss for this image is backpropagated, only those parameters would be modified. In other words, the incremental step taken in the parameter hyperplane is likely to aligned more closely with the parameters most affected by the training image.
- What that implies is that there will be a bit of a "chaos" associated with the path extensions if we only consider a small number of training samples at a time. *As it turns out, there is experimental evidence that this chaos may help the solution path to jump out of local minima.* You might then ask: Why not use just one training sample at a time? Experimental evidence suggests that, in most cases, using more than one sample does a bit of local smoothing in the estimated path increment, which helps with the convergence.

Visualizing the Loss Surface (again)


Aggregated Loss Surface over the hyperplane
Hyperplane spanned by learnable parameters
Each step in the parameter hyperplane is based on the loss calculated for one batch of training data

In SGD, each path increment in the parameter hyperplane is now determined by just the batch-size number of training samples. But note that the loss surface will now be different for each batch — since the surface depends on the training data.

The 3D plot from: `https://www.fromthegenesis.com/gradient-descent-part1/`

Computation of the Derivatives Required by the Jacobian

- We will revisit Eqs. (1) and (2) on Slide 11.

$$\mathbf{p}_{k+1} = \mathbf{p}_k + 2 \cdot \alpha \cdot J_{\mathbf{f}}(\mathbf{p}_k) \cdot \epsilon_k$$

where the Jacobian $J_{\mathbf{f}}(\mathbf{p})$ is given by

$$J_{\mathbf{f}}(\mathbf{p}) \quad = \quad \begin{bmatrix} \frac{\delta f_1}{\delta p_1} & \cdots & \frac{\delta f_1}{\delta p_n} \\ \vdots & \vdots & \vdots \\ \frac{\delta f_M}{\delta p_1} & \cdots & \frac{\delta f_M}{\delta p_n} \end{bmatrix}$$

- While it is true that, overall, a path increment can only be calculated after you have seen the loss at the output, nonetheless, as you will see, it is possible to make the computations efficient by doing some of the work during the forward pass of a training sample through the network and the rest of the work as the loss is being backpropagated.

Computation of the Derivatives Required by the Jacobian (contd.)

- In order to see how one would go about computing the partial derivatives shown on the previous slide during the forward pass, consider first the simple case of a single-layer neural network whose input/output relationship for one training sample is described by

  $$\mathbf{y} = f(\mathbf{x}, \mathbf{p}) = g(\mathbf{W} \cdot \mathbf{x})$$

  where $\mathbf{W}$ is the link matrix between the input and the hidden layer, $\mathbf{W} \cdot \mathbf{x}$ the pre-activation output at the hidden layer, and $g()$ the activation function. [The dot operator in $\mathbf{W} \cdot \mathbf{x}$ is meant to emphasize the role of $\mathbf{W}$ as an operator.]

- So we can write:

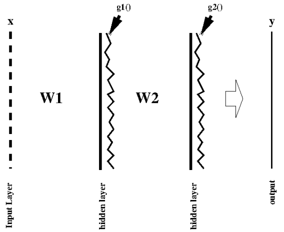
  $$\nabla_{\mathbf{p}} f = \partial_g (\nabla_{\mathbf{W}} (\mathbf{W} \cdot \mathbf{x}))$$

  where ∂_g denotes a point-wise partial derivative of the function $g()$ with respect to its argument and $\nabla_{\mathbf{W}}$ a Jacobian-like derivative of the pre-activation output at the hidden layer.

Computation of the Derivatives Required by the Jacobian (contd.)

- Let's now bring in one more hidden layer into this network. Let $\mathbf{W}_1$ and $\mathbf{W}_2$ be the link matrices between the input and the first hidden layer and between the first hidden layer and the next hidden layer. If we represent the output of the first hidden layer by $\mathbf{h}$, we can now write

$$\mathbf{y} = f(\mathbf{x}, \mathbf{p}) = g_2(\mathbf{W}_2\mathbf{h}) = g_2(\mathbf{W}_2 \cdot g_1(\mathbf{W}_1 \cdot \mathbf{x}))$$



Computation of the Derivatives Required by the Jacobian (contd.)

- Taking the partials of both sides for the case of two hidden layers shown on the previous slide, we get

$$\nabla_{\mathbf{p}} f = \partial_{g_2}(\nabla_{\mathbf{W}_2}(\mathbf{W}_2 \cdot (\partial_{g_1}(\nabla_{\mathbf{W}_1}(\mathbf{W}_1 \cdot \mathbf{x})))))$$

- Although the calculation of the Jacobian shown on Slide 21 now involves a chain of derivatives as shown above, one thing is clear:

  If you unpack the chain shown above from right to left, all these partials can be calculated during the forward pass of a training sample through the network.

Computation of the Derivatives Required by the Jacobian (contd.)

- The derivation shown on last few slides was for the very simple case involving what's referred to a "single-channel" input — meaning that we could think of the input **x** as a vector — and no biases, let alone the absence of any convolutional processing of the data.

- The more general case would involve multi-channel inputs and multi-channel outputs for each layer. This requires generalizing our derivatives to those for tensors.

- Consult Professor Bouman's slides for those generalizations.

Estimating the Partials During the Forward Pass (contd.)

- The overall implication of the chaining of the partial derivatives is fundamental to how the loss is backpropagated and the gradients of the loss computed by Autograd.

- During forward propagation of the training data through the network, Autograd computes all the partial derivatives needed for the Jacobian.

- Subsequently, after the loss has been estimated at the output, it is backpropagated through the network using the previously computed partial derivatives during the forward pass. Note that, in general, complex neural networks involve multiple chains of dependencies that are best represented by a graph data structure. These graphs are called computational graphs — a subject we take up next.

Outline

The Computational Graph Primer

- In order to convey a more precise understanding what was described on the previous slide, I have created a Python module called `ComputationalGraphPrimer` that you can download from:

  `https://pypi.org/project/ComputationalGraphPrimer/1.0.2/`

- What prompted me to create `ComputationalGraphPrimer` was the text shown below from the official documentation page for Autograd. Considering that we are now at a point where practically every science and engineering student wants to learn how deep learning works, I can't imagine that unless you are a CS or a CompE major, you would have any idea as to what this official text from PyTorch is saying. Here it is:

From the official Autograd doc page: "Every operation performed on Tensors creates a new function object, that performs the computation, and records that it happened. The history is retained in the form of a DAG of functions, with edges denoting data dependencies (input $\leftarrow$ output). Then, when backward is called, the graph is processed in the topological ordering, by calling backward() methods of each Function object, and passing returned gradients on to next Functions." **and** "Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in inputs that are of floating point type and with requires_grad=True."

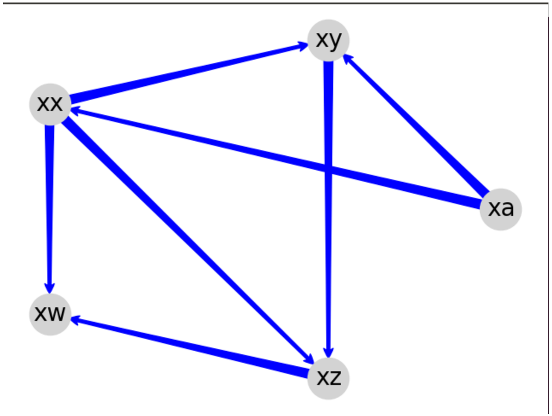
The Computational Graph Primer (contd.)

The `ComputationalGraphPrimer` module allows you to create a DAG (Directed Acyclic Graph) of variables with a statement like

```
expressions = ['xx=xa^2',
               'xy=ab*xx+ac*xa',
               'xz=bc*xx+xy',
               'xw=cd*xx+xz^3']
```

where we assume that a symbolic name that starts with the letter 'x' is a variable and that all other symbolic names are learnable parameters, and where we use ^ for exponentiation. The four expressions shown above contain five variables — 'xx', 'xa', 'xy', 'xz', and 'xw' — and four learnable parameters: 'ab', 'ac', 'bc', and 'cd'. The DAG that is generated by these expressions looks like what is shown on the next slide.

A Toy Example of a Computational Graph



Computational Graph for the expressions shown on the previous slide. For a symbolic name to be a variable, its first letter must be 'x'. All other symbolic names are to be treated like learnable parameters.

The Computational Graph Primer (contd.)

- In the DAG shown on the previous slide, the variable 'xa' is an independent variable since it has no incoming arcs, and 'xw' is an output variable since it has no outgoing arcs.
- A DAG of the sort shown above is represented in `ComputationalGraphPrimer` by two dictionaries: `depends_on` and `leads_to`. Here is what the `depends_on` dictionary would look like for the DAG shown on the previous slide:

```
depends_on['xx']  =  ['xa']
depends_on['xy']  =  ['xa', 'xx']
depends_on['xz']  =  ['xx', 'xy']
depends_on['xw']  =  ['xx', 'xz']
```

- Something like `depends_on['xx'] = ['xa']` is best read as "the vertex 'xx' depends on the vertex 'xa'." Similarly, the line `depends_on['xz'] = ['xx','xy']` is best read aloud as "the vertex 'xz' depends on the vertices 'xx' and 'xy'." And so on.

The Computational Graph Primer (contd.)

- Whereas the `depends_on` dictionary is a complete description of a DAG, for programming convenience, `ComputationalGraphPrimer` also maintains another representation for the same graph, as provide by the `leads_to` dictionary. This dictionary for the same graph would be:

```
leads_to['xa']    =  ['xx', 'xy']
leads_to['xx']    =  ['xy', 'xz', 'xw']
leads_to['xy']    =  ['xz']
leads_to['xz']    =  ['xw']
```

- The `leads_to[xa] = [xx]` is best read as "the outgoing edge at the node 'xa' leads to the node 'xx'." Along the same lines, the `leads_to['xx'] = ['xy','xz','xw']` is best read as "the outgoing edges at the vertex 'xx' lead to the vertices 'xy', 'xz', and 'xw'."

How Does ComputationalGraphPrimer Work?

- After you have constructed an instance of `ComputationalGraphPrimer` by calling its constructor:

```
cgp = ComputationalGraphPrimer(
               expressions = ['xx=xa^2',
                              'xy=ab*xx+ac*xa',
                              'xz=bc*xx+xy',
                              'xw=cd*xx+xz^3'],
               output_vars = ['xw'],
               dataset_size = 10000,
               learning_rate = 1e-6,
               display_vals_how_often = 1000,
               grad_delta = 1e-4,
      )
```

  you need to make the calls shown on the next slide for a demonstration of how the data flows forward in the graph while the node-to-node partial derivatives are calculated at the same time.

- In the call shown above, we have designated the output var as '`xw`'. If you leave this option out, the module can figure out on its own as to which variables to designate as the output vars on the basis of there being no outgoing arcs at those vertices.

How Does ComputationalGraphPrimer Work? (contd.)

- On the instance constructed as shown on the previous slide, you would first need to call the parser to construct a graph from the expressions supplied through the constructor:

```
cgp.parse_expressions()
```

  By analyzing the topology of the graph, the parser would also figure out all the input vars and the output vars. If it sees that you have specified the output variables, it would only use those for the output.

- Next, you must generate the **ground-truth data** by a call like

```
cgp.gen_gt_dataset(vals_for_learnable_params = {'ab':1.0, 'bc':2.0, 'cd':3.0, 'ac':4.0})
```

  For the values supplied for the learnable parameters, the ground-truth (GT) data consists of `(input,output)` pairs in which the input values are randomly generated and the corresponding output values calculated by pushing the input values through the graph.

How Does ComputationalGraphPrimer Work? (contd.)

Next you pretend that you don't know the true values for the learnable parameters. So you call on the following method:

```
cgp.train_on_all_data()
```

This method starts with randomly guessed values for the learnable parameters and goes through the following steps for each training sample in the GT dataset:

1. pushes each input value in the GT dataset through the network;
2. as the data propagates forward though the network, it simultaneously calculates the vertex-to-vertex partial derivatives using the finite difference method;
3. when the input training sample reaches the output nodes, it estimates the loss with respect to the GT output for the input; and, finally,
4. it backpropagates the loss and, using the chain rule, estimates its gradients with respect to the variables at the vertices.

How Does ComputationalGraphPrimer Work? (contd.)

- Here are a couple of additional methods defined for the `ComputationalGraphPrimer` module that you will find useful:

  ```
  cgp.display_network2()

  cgp.plot_loss()
  ```

  The first is for displaying the network graph and the second for plotting the loss as a function of the training iterations.

- Do not expect to see the loss decrease with training iterations — simply because a network of the sort you would supply to `ComputationalGraphPrimer` is not a learning network. It is definitely not a neural network. It does no node based aggregations of the incoming data, does not apply activations to the output values at the nodes, etc. The goal of the primer is just to play with the calculation of the partial derivatives in a network of dependencies.

Outline

Autograd for Automatic Calculation of Gradients

- I have already mentioned several things about Autograd in the material we have covered so far. In this section, I am now going to summarize how this module functions.

- With Autograd, during the forward pass of a training sample, a computational graph is constructed and the partial derivatives calculated in the same manner as in `ComputationalGraphPrimer` — although Autograd uses a more sophisticated computational graph in which the operators on the tensors define the nodes of the graph as opposed to the tensors themselves.

- It would not be too difficult to extend the implementation of the `ComputationalGraphPrimer` module so that the operators inside the expressions serve as the nodes of the graph. However, the basic demonstration of the principle of automatic differentiation during the forward pass of the training data would remain unchanged.

Autograd for Automatic Calculation of Gradients (contd.)

- After the loss is calculated at the output, the partial derivatives calculated during the forward pass are used to propagate the loss backwards and the gradient of the loss computed with respect to the parameters encountered. This is initiated by the following statement in your code:

```
    loss.backward()
```

- For this behavior of Autograd, all you have to do is to set the `requires_grad` attribute of the tensors that define the learnable parameters to `True`.
- The value of the gradient itself is stored in the attribute `grad` of tensors involved.

Static vs. Dynamic Computational Graphs

- PyTorch's computational graphs are dynamic, in the sense that a new graph is created in each forward pass.
- On the other hand, the computational graphs constructed by Tensorflow are static. That is, the same graph is used over and over in all iterations during training.
- In general, static graphs are more efficient because you need to optimize them only once. Optimization generally consists of distributing the computations over the graph nodes across multiple GPUs if more than one GPU is available or just fusing some nodes of the graph if the resulting logic won't be impacted by such fusion.
- Static graphs do not lend themselves well to recurrent neural computations because the graph itself can change from iteration to iteration.

Outline

Extending Autograd

- Being highly object-oriented, you would think that PyTorch would give you all kinds of freedom in extending the platform. But that's not the case.
- On account of how PyTorch sits on top of the C++ code base that knows how to work the GPUs, the normal code extension facilities that one attributes to OO platforms do not apply to PyTorch.
- You are allowed to extend only two things in PyTorch: Autograd and the torch.nn module, and that too in only certain specific ways. In this section, I'll show what it is you have to do to extend Autograd.
- I'll illustrate how to extend Autograd with the help in the inner class `AutogradCustomization` of my `DLStudio` module.

Extending Autograd (contd.)

- The Examples subdirectory of the `DLStudio` distribution contains a script named `extending_autograd.py` with the following code:

```
from DLStudio import *
dls = DLStudio(                                                                    ## (A)
                  learning_rate = 1e-3,
                  epochs = 5,
                  use_gpu = True,
              )
ext_auto = DLStudio.AutogradCustomization(                                         ## (B)
                                           dl_studio = dls,
                                           num_samples_per_class = 1000,
                                         )
ext_auto.gen_training_data()                                                       ## (C)
ext_auto.train_with_straight_autograd()                                            ## (D)
ext_auto.train_with_extended_autograd()                                            ## (E)
```

where in line (A), we construct an instance of the `DLStudio` class. Subsequently, in line (B), we construct an instance of `DLStudio`'s inner class `AutogradCustomization` that has the code for demonstrating how to extend PyTorch's Autograd module.

Extending Autograd (contd.)

- In line (C) in the previous slide, we generate the labeled training data from a 2D multivariate Gaussian. The definition of the function `ext_auto.gen_training_data()` invoked in line (C) has the following code:

```
mean1,mean2   = [3.0,3.0], [5.0,5.0]
covar1,covar2 = [[1.0,0.0], [0.0,1.0]], [[1.0,0.0], [0.0,1.0]]
data1 = [(list(x),1) for x in np.random.multivariate_normal(mean1, covar1,self.num_samples_per_class)]
data2 = [(list(x),2) for x in np.random.multivariate_normal(mean2, covar2,self.num_samples_per_class)]
training_data = data1 + data2
random.shuffle( training_data )
```

- As you see, we are drawing training samples from two bivariate Gaussian distributions, with different means but identical covariances.
- The next slide shows the definition of the function `train_with_straight_autograd()` that has been invoked in line (D) of the previous slide. By "straight autograd", I mean Autograd without any modifications.

Extending Autograd (contd.)

- Before explaining how you can extend Autograd, in order to compare the before and after results, let's first look at the implementation of the function `train_with_straight_autograd()` that's based on Autograd as it comes:

```
def train_with_straight_autograd(self):
    dtype = torch.float                                                           # (1)
    D_in,H,D_out = 2,10,2                                                         # (2)
    w1 = torch.randn(D_in, H, device="cpu", dtype=dtype)                          # (3)
    w2 = torch.randn(H, D_out, device="cpu", dtype=dtype)                         # (4)
    w1 = w1.to(self.dl_studio.device)                                             # (5)
    w2 = w2.to(self.dl_studio.device)                                             # (6)
    w1.requires_grad_()                                                           # (7)
    w2.requires_grad_()                                                           # (8)
    Loss = []                                                                     # (9)
    for epoch in range(self.dl_studio.epochs):                                    # (10)
        for i,data in enumerate(self.training_data):                              # (11)
            input, label = data                                                   # (12)
            x,y = torch.as_tensor(np.array(input)), torch.as_tensor(np.array(label))  # (13)
            x,y = x.float(), y.float()                                            # (14)
            if self.dl_studio.use_gpu is True:                                    # (15)
                x,y = x.to(self.dl_studio.device), y.to(self.dl_studio.device)    # (16)
            y_pred = x.view(1,-1).mm(w1).clamp(min=0).mm(w2)                      # (17)
```

(Continued on the next slide .....)

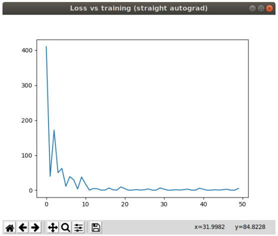
(...... continued from the previous slide)

```
loss = (y_pred - y).pow(2).sum()                                        # (18)
loss.backward()                                                         # (19)
with torch.no_grad():                                                   # (20)
    w1 -= self.dl_studio.learning_rate * w1.grad                        # (21)
    w2 -= self.dl_studio.learning_rate * w2.grad                        # (22)
    w1.grad.zero_()                                                     # (23)
    w2.grad.zero_()                                                     # (24)
```

Explanation of the code:

The statement in line (2) supplies the number of nodes to be used for a one-hidden-layer neural network. Line (3) defines the link matrix for the input to the hidden layer and line (4) for the hidden layer to the output. We intentionally create the learnable parameters for w1 and w2 in the cpu and not in the gpu in order for the results to be reproducibly the same regardless of which devices is being used for computing. Of course, for reproducability, you will also have to set the different seeds to zero.

The rest of the code is self explanatory. In line (11), we get hold of the input/output data pairs from the training dataset created by the data generator on Slide 44. Subsequently, in line (17), we push the training sample through the network and calculate the predicted label for the input. A plot of the loss against the training iterations is shown below:



Extending Autograd (contd.)

- Now we are all set for me to explain how you can extend Autograd. The first thing you need to do is to define your verb class that can: (1) trap a training sample in its forward pass through the network; (2) modify the training sample if necessary; (3) use a context variable to remember whatever is deemed important about the change that was made to the training sample in its forward journey; and then (4) on the backward pass related to the same training sample, recall what was stored away in the context variable, and do whatever is needed.

- This class that your code must define must include implementations for two static methods `forward()` and `backward()`.

- To satisfy these requirements, the code for the inner class `AutogradCustomization` of the `DLStudio` module contains the class definition shown on the next slide. The name of this class is `DoSillyWithTensor`.

Extending Autograd (contd.)

- Shown below is the class `DoSillyWithTensor` mentioned on the previous slide:

```
class DoSillyWithTensor(torch.autograd.Function):                            # (1)
    @staticmethod                                                             # (2)
    def forward(ctx, input):                                                  # (3)
        input_orig = input.clone().double()                                   # (4)
        input = input.to(torch.uint8).double()                                # (5)
        diff = input_orig.sub(input)                                          # (6)
        ctx.save_for_backward(diff)                                           # (7)
        return input                                                          # (8)

    @staticmethod                                                             # (9)
    def backward(ctx, grad_output):                                           # (10)
        diff, = ctx.saved_tensors                                             # (11)
        grad_input = grad_output.clone()                                      # (12)
        grad_input = grad_input + diff                                        # (13)
        return grad_input                                                     # (14)
```

Explanation of the code:

The parameter `input` in line (3) is set to the training sample that is being processed by an instance of `DoSillyWithTensor` in the `forward()` of the network. In line (4), we first make a deep copy of this tensor (which should be a 32-bit float) and then we subject the copy to a conversion to a one-byte integer in line (5). This should cause a significant loss of information in the training sample. In line (6), we calculate the difference between the original 32-bit float and the 8-bit version and store it away in the context variable `ctx`. Subsequently, we retrieve this quantization error during the backward pass in line (11) and add it to the value in the `grad` attribute of the tensor.

Extending Autograd (contd.)

- Finally, we are ready to talk about the call to the function `train_with_extended_autograd()` in line (E) on Slide 43. Here is how that function is implemented in the inner class `AutogradCustomization` of the `DLStudio` module:

```
def train_with_extended_autograd(self):
    dtype = torch.float
    D_in,H,D_out = 2,10,2
    w1 = torch.randn(D_in, H, device="cpu", dtype=dtype)
    w2 = torch.randn(H, D_out, device="cpu", dtype=dtype)
    w1 = w1.to(self.dl_studio.device)
    w2 = w2.to(self.dl_studio.device)
    w1.requires_grad_()
    w2.requires_grad_()
    Loss = []
    for epoch in range(self.dl_studio.epochs):
        for i,data in enumerate(self.training_data):
            do_silly = DLStudio.AutogradCustomization.DoSillyWithTensor.apply               # (1)
            input, label = data
            x,y = torch.as_tensor(np.array(input)), torch.as_tensor(np.array(label))
            x = do_silly(x)                                                                 # (2)
            x,y = x.float(), y.float()
            x,y = x.to(self.dl_studio.device), y.to(self.dl_studio.device)
```

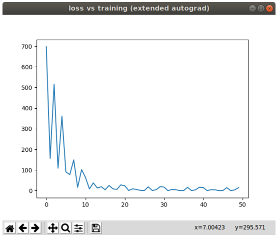
(Continued on the next slide .....)

(...... continued from the previous slide)

```
y_pred = x.view(1,-1).mm(w1).clamp(min=0).mm(w2)
loss = (y_pred - y).pow(2).sum()
loss.backward()
with torch.no_grad():
    w1 -= self.dl_studio.learning_rate * w1.grad
    w2 -= self.dl_studio.learning_rate * w2.grad
    w1.grad.zero_()
    w2.grad.zero_()
```

Explanation of the code:

Except for the lines labeled (1) and (2), this code is exactly the same as what you saw earlier on slides (45) and (46). The call in line (1) constructs an instance of DoSillyWithTensor. Note that this instance is callable. Subsequently, in line (2), the training sample is being processed by the `do_silly` instance in line (1). The loss of precision caused by the extension to Autograd is evident in the following plot of loss versus training iterations. Compare this result with the one shown earlier in Slide 46.



Outline

Calculating the Gradients for Neural Networks

- The focus of this section is exclusively the calculation of the gradient in simple feed-forward neural networks. You are going to need this material for your homework assignment. As it turns out, the gradient calculation in such networks is a pretty simple matter.
- Say we have a neural network with one input layer, one hidden layer, and one output later. Let w_1 be the matrix of link weights between the input and the hidden layer and w_2 the link weights between the hidden layer and the output.
- Also let h denote the output of the hidden layer before activation and h_{relu} the output after the ReLU activation.

Calculating the Gradients for Neural Networks (contd.)

- Additionally, let y_{pred} denote the predicted output at the final layer, and y the ground-truth for a given input x. We can write for the loss

$$\begin{aligned} L &= (y - y_{pred})^T (y - y_{pred}) \\ &= [y - (w_2^T * h_{rel})]^t [y - (w_2^t * h_{rel})] \end{aligned} \quad (3)$$

  where h_{rel} is the output of the hidden layer after it is subject to activation.

- For the partial of L with respect to w_2, we can write:

$$\begin{aligned} \frac{\delta L}{\delta w_2} &= -2[y - (w_2^T * h_{rel})] h_{rel} \\ &= -2[y - y_{pred}] * h_{rel} \\ &= 2[y_{pred} - y] h_{rel} \end{aligned}$$

Calculating the Gradients for Neural Networks (contd.)

- Next we must address the gradients of L with respect to the link weights w_1. For that we need to invoke the chain rule because:

$$
\begin{aligned}
\frac{\delta L}{\delta w_1} &= \frac{\delta L}{\delta y_{pred}} \cdot \frac{\delta y_{pred}}{\delta h_{rel}} \cdot \frac{\delta h_{red}}{\delta w_1} \\
&= \frac{\delta L}{\delta y_{pred}} \cdot w_2^T \cdot \frac{\delta h_{red}}{\delta w_1} \\
&= y_{error} \cdot w_2^T \cdot \frac{\delta h_{red}}{\delta w_1}
\end{aligned}
$$

  where the first term in the last equation follows from the fact that $L = [y - y_{pred}]^T [y - y_{pred}]$.

- As for the third term in the last equation, we note that

$$h_{rel} = RelU(w_1 * x)$$

  which is the same as saying that

$$h_{rel} = \begin{cases} w_1 * x & \text{when } w_x * x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Calculating the Gradients for Neural Networks (contd.)

Therefore

$$\frac{\delta L}{\delta w_1} = ? \qquad \text{(you need to derive it for yourself)}$$

In summary, the backpropagation steps for NN are:

1. At layer l of the network, if w_l is the link matrix between layer $l-1$ and layer l,

$$gradient\ loss\ wrt\ w_l = output_{l-1}.mm(error_l)$$

2. Backpropagate the error to the post-activation point in the previous layer:

$$error_{l-1} = error_l.mm({w_l}^T)$$

3. Further backpropagate the error through the activation: For this you just zero out those elements where the forward propagated values to the same layer are negative. That gives you the backpropagated error on the preactivation side of the layer $l-1$.