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

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 9th March, 2021 22:46
Let us consider learning-based systems in general that must set their critical parameters by minimizing the prediction errors they make on the 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 calculating the gradients of the loss that are needed for SGD.
Preamble (contd.)

When I say that Autograd constructs a *dynamic* computational graph, what that means is that a new graph is constructed for each new batch of training data coursing its way through the network. An alternative to a dynamic computational graph is a static computational graph that is used in the TensorFlow platform for deep learning.

Here is a most noteworthy feature of the computational graph that is constructed by the Autograd module during the forward propagation of a batch of training data through a network: Autograd calculates the partial derivatives of the output of each layer vis-a-vis each element of its input. These partial derivatives are stored away in the computational graph.

Subsequently, these pre-computed partial derivatives are used for updating the values of the learnable parameters (that are typically the weights associated with the links in a multi-layer neural network).

In addition to using the forward-computed partial derivatives for updating the learnable parameters, there is one more issue relevant to using SGD: averaging the update calculations over the training samples in a batch.
To make it easier for a student new to DL to understand the ideas mentioned on the last couple of slides, I have created a Python module called ComputationalGraphPrimer that’s available at

https://pypi.org/project/ComputationalGraphPrimer/1.0.5/

The Examples directory of Version 1.0.5 of this Primer contains the following scripts:

one_neuron_classifier.py
multi_neuron_classifier.py

The functions invoked by these scripts illustrate the following: (1) How one can compute the partial derivatives during the forward propagation of a batch of input data; (2) How those partial derivatives can subsequently be used for updating the learnable parameters during backpropagation of loss; and (3) The batch based smoothing as required by SGD of the partial derivatives and the predictions errors before the parameters are updated.
In Version 1.0.5 of the Primer, the partial derivatives of the output of a neural layer with respect to each element of the input to the layer are based on analytical formulas.

In the next version of the module, I am planning to include an example based on using the finite difference method for estimating the partial derivatives.

The Primer also illustrates dataflow in a general DAG (Directed Acyclic Graph) of nodes, both the forward propagation of the input data, the computation of the partial derivatives, and the backpropagation of loss. The goal of this illustration is only to introduce a student to the notion of a DAG. You will see references to DAG in the official PyTorch documentation on the Autograd module.

Also included in the Primer is a demonstration of how to extend the Autograd class for more specialized processing of the data that may require remembering certain designated attributes of the data as it is flowing in the forward direction and using the remembered values to make changes to the loss flowing backwards during backpropagation.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revisiting Gradient Descent</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Problems with a Vanilla Application of GD</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Stochastic Gradient Descent</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>A Computational Graph Primer</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Primer Demo 1: <code>graph_based_dataflow.py</code></td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Primer Demo 2: <code>one_neuron_classifier.py</code></td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Primer Demo 3: <code>multi_neuron_classifier.py</code></td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>Primer Demo 4: <code>verify_with_torchnn.py</code></td>
<td>66</td>
</tr>
<tr>
<td>9</td>
<td>Autograd</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td>Extending Autograd</td>
<td>77</td>
</tr>
<tr>
<td>11</td>
<td>Forward and Backward Calculations in an Ordinary NN</td>
<td>89</td>
</tr>
</tbody>
</table>
# Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revisiting Gradient Descent</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Problems with a Vanilla Application of GD</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Stochastic Gradient Descent</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>A Computational Graph Primer</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Primer Demo 1: graph_based_dataflow.py</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Primer Demo 2: one_neuron_classifier.py</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Primer Demo 3: multi_neuron_classifier.py</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>Primer Demo 4: verify_with_torchnn.py</td>
<td>66</td>
</tr>
<tr>
<td>9</td>
<td>Autograd</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td>Extending Autograd</td>
<td>77</td>
</tr>
<tr>
<td>11</td>
<td>Forward and Backward Calculations in an Ordinary NN</td>
<td>89</td>
</tr>
</tbody>
</table>

Purdue University
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 \((x_i, y_i)\) where the vectors \(x_i, i = 1, \ldots, m\) represent the vector training data samples and \(y_i\) the corresponding ground-truth labels. Given, say, 10 classes, the value of each \(y_i\) will be an integer from the set \(\{1, \ldots, 10\}\). [We could have assumed the ground-truth labels to be one-hot vectors, but that would only complicate the cost-function formulas shown later in this section and detract from the insights this discussion provides.]

- Let the predicted label for training sample \(x_i\) be the integer \(y'_i\).

- We can measure the loss for each sample by, say, the mean square error:

\[
L_i = (y_i - y'_i)^2
\]  
(1)
Loss Surfaces and the Parameter Hyperplane (contd.)

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

\[
L = \sum_{i=1}^{m} L_i = \sum_{i=1}^{m} (y_i - y'_i)^2 \tag{2}
\]

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

\[
y'_i = f_i(x_i, p) \tag{3}
\]

- We can now write for the overall loss:

\[
L = \sum_{i=1}^{m} L_i = \sum_{i=1}^{m} (y_i - f_i(x_i, p))^2 \tag{4}
\]
Since the overall loss is the sum of the squares of the error in each training sample separately, we are allowed to represent the summation over all the training samples by

\[ L = \| Y - F(X, p) \|^2 \]  \hspace{1cm} (5)

where you can think of \( Y \) as us lining up all the ground-truth integer labels for all the \( m \) training samples into one large vector.

By the same token, \( F \) also represents us lining up all the predictions for the individual samples into a large vector whose size is same as that of \( Y \).

The quantity \( X \) in the formula shown above now represents all the training data.
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 $\frac{\delta L}{\delta p}$.

- The gradient 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 in gradient descent (GD) 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 $p_k$ in the parameter space, our next point in our journey to the global minimum would be at

$$p_{k+1} = p_k - \alpha \cdot \frac{\delta L}{\delta p} \quad (6)$$

where $\alpha$ is a small number that is called the **learning rate**.

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

$$\frac{\delta L}{\delta p} = 2J_\epsilon^T(p)\epsilon(p) \quad (7)$$

where $J_\epsilon(p)$ is the Jacobian and where $\epsilon$ is the prediction error given by $\epsilon = Y - F(X, p)$. 
Using GD Means Having to Calculate the Jacobian (contd.)

- Since the vector $Y$ in $\epsilon = Y - F(X, p)$ consists of the ground-truth class labels, $Y$ is a constant from the standpoint of differentiation. Therefore, we can write $J_\epsilon(p) = -J_F(p)$. Substituting this in our update formula, we get

$$p_{k+1} = p_k + 2 \cdot \alpha \cdot J_F(p_k) \cdot \epsilon_k$$

(8)

- The Jacobian $J_f(p)$ is given by

$$J_F(p) = \begin{bmatrix}
\frac{\delta f_1}{\delta p_1} & \ldots & \frac{\delta f_1}{\delta p_n} \\
\vdots & \ddots & \vdots \\
\frac{\delta f_m}{\delta p_1} & \ldots & \frac{\delta f_m}{\delta p_n}
\end{bmatrix}$$

(9)

[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.]
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:


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.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revisiting Gradient Descent</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Problems with a Vanilla Application of GD</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Stochastic Gradient Descent</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>A Computational Graph Primer</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Primer Demo 1: graph_based_dataflow.py</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Primer Demo 2: one_neuron_classifier.py</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Primer Demo 3: multi_neuron_classifier.py</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>Primer Demo 4: verify_with_torchnn.py</td>
<td>66</td>
</tr>
<tr>
<td>9</td>
<td>Autograd</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td>Extending Autograd</td>
<td>77</td>
</tr>
<tr>
<td>11</td>
<td>Forward and Backward Calculations in an Ordinary NN</td>
<td>89</td>
</tr>
</tbody>
</table>
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 $J_f^T J_f$ which makes it infeasible for DL where you may have millions of learnable parameters.]
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?
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 $\delta p$ 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.
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.
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. (8) and (9) shown earlier:

\[ p_{k+1} = p_k + 2 \cdot \alpha \cdot J_F(p_k) \cdot \epsilon_k \]  
(10)

where the Jacobian \( J_f(p) \) is given by

\[ J_F(p) = \begin{bmatrix} \frac{\delta f_1}{\delta p_1} & \cdots & \frac{\delta f_1}{\delta p_n} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_m}{\delta p_1} & \cdots & \frac{\delta f_m}{\delta p_n} \end{bmatrix} \]  
(11)

- 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.
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

\[ y = f(x, p) = g(W \cdot x) \]  

(12)

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

So we can write:

\[ \nabla_p f = \partial_g \left( \nabla_W (W \cdot x) \right) \]  

(13)

where \( \partial_g \) denotes a point-wise partial derivative of the function \( g() \) with respect to its argument and \( \nabla_W \) a Jacobian-like derivative of the pre-activation output at the hidden layer.
Let’s now bring in one more hidden layer into this network. Let $W_1$ and $W_2$ 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 $h$, we can now write

$$y = f(x, p) = g_2(W_2 h) = g_2(W_2 \cdot g_1(W_1 \cdot x))$$  \hspace{1cm} (14)
Taking the partials of both sides for the case of two hidden layers shown on the previous slide, we get

\[ \nabla_p f = \partial_{g_2}(\nabla_{W_2}(W_2 \cdot (\partial_{g_1}(\nabla_{W_1}(W_1 \cdot x)))))) \] (15)

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.
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.
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

1. Revisiting Gradient Descent  
2. Problems with a Vanilla Application of GD  
3. Stochastic Gradient Descent  
4. A Computational Graph Primer  
5. Primer Demo 1: `graph_based_dataflow.py`  
6. Primer Demo 2: `one_neuron_classifier.py`  
7. Primer Demo 3: `multi_neuron_classifier.py`  
8. Primer Demo 4: `verify_with_torchnn.py`  
9. Autograd  
10. Extending Autograd  
11. Forward and Backward Calculations in an Ordinary NN
The Computational Graph Primer

In order to convey a more precise understanding of what was described in the previous section, I have created a Python module called ComputationalGraphPrimer that you can download from:

https://pypi.org/project/ComputationalGraphPrimer/1.0.5/

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 ← 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.”
Your best entry into the Primer is through the following Python scripts in the Examples directory of the distribution:

1. **graph_based_dataflow.py**
   Demonstrates forward and backward data flows in a DAG while also calculating the partial derivatives needed for parameter update.

2. **one_neuron_classifier.py**
   Demonstrates forward computation of the derivatives and using those derivatives updating the learnable parameters. Uses the Sigmoid activation function.

3. **multi_neuron_classifier.py**
   This is a generalization of the previous script to a multi-layer neural network. Uses the Sigmoid activation function at the nodes.

4. **verify_with_torchnn.py**
   This script is for verifying the performance of the previous two scripts.
Outline

1. Revisiting Gradient Descent ........................................ 7
2. Problems with a Vanilla Application of GD .................. 17
3. Stochastic Gradient Descent ......................................... 19
4. A Computational Graph Primer ..................................... 30
5. Primer Demo 1: graph_based_dataflow.py ....................... 33
6. Primer Demo 2: one_neuron_classifier.py ....................... 45
7. Primer Demo 3: multi_neuron_classifier.py ..................... 54
8. Primer Demo 4: verify_with_torchnn.py ......................... 66
9. Autograd ..................................................................... 73
10. Extending Autograd .................................................... 77
11. Forward and Backward Calculations in an Ordinary NN .... 89
You can create a DAG (Directed Acyclic Graph) with a statement like

```python
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.
In this DAG, \( x_a \) is the only independent node (the same as the input node) and \( x_w \) the only output node. The arcs have learnable parameters associated with them. Is it possible to estimate the learnable parameters? Certainly not without nonlinearities in the network.
In the DAG shown on the previous slide, the variable 'xa' is the only independent variable since it has no incoming arcs, and 'xw' is the only output variable since it has no outgoing arcs.

A DAG of the sort shown above is represented in the Primer module 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:

```python
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.
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 provided by the `leads_to` dictionary. This dictionary for the same graph would be:

```python
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'.”
After you have constructed an instance of ComputationalGraphPrimer by calling its constructor:

```python
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.
After you have constructed the instance cgp as shown on the previous slide, you would first need to call the parser as shown below in order to construct the computational graph:

```python
    cgp.parse_expressions()
```

By analyzing the topology of the graph, the parser figures 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

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

This call gives us a mapping from randomly-generated values at the input nodes to the values produced at the output nodes when the learnable parameters are set as shown above. **Subsequently, we retain the input-output mapping as the ground-truth (GT) and forget the values used for the learnable parameters. The goal of learning would then be to recreate the values used for the learnable parameters.**
As mentioned on the previous slide, you pretend that you don’t know the true values used for the learnable parameters. So you call on the following method:

```python
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.
Each training iteration in the function `train_on_all_data()` called on the previous slide generates the following sort of output at multiples of 1000 iterations in the training loop:

Initial values for all learnable parameters: `cd': 0.8134267156709766, 'ab': 0.6303253744046711, 'ac': 0.6490336075539102, 'bc': 0.0845266346159006

[Forward Propagation] Training with sample indexed: 1000

Predicted value at the output nodes: `xw': 2.1718835784922437
Loss for training sample indexed 1000: 212.7733678538975

Estimated partial derivatives of vars wrt learnable parameters:
- k=xx: `cd': None, 'ab': None, 'ac': None, 'bc': None
- k=xa: `cd': None, 'ab': None, 'ac': None, 'bc': None
- k=xz: `cd': None, 'ab': None, 'ac': None, 'bc': -0.3396212691565975
- k=xw: `cd': None, 'ab': None, 'ac': None, 'bc': -0.3396212691565975

Estimated partial derivatives of vars wrt other vars:
- k=xx: xx: None, 'xa': None, 'xz': None, 'xw': None, 'xy': None
- k=xa: xx: None, 'xa': None, 'xz': None, 'xw': None, 'xy': None
- k=xz: xx: 0.0845287913638082, 'xa': None, 'xz': None, 'xw': None, 'xy': 0.9999999999998899
- k=xw: xx: 0.8134216163468988, 'xa': None, 'xz': 1.1568387019889048, 'xw': None, 'xy': None
- k=xy: xx: 0.6303669260676603, 'xa': None, 'xz': None, 'xw': None, 'xy': None

[sample index: 1000]: input val: `xa': 0.8933787891895923 vals for learnable parameters: `cd': 0.8134339110108809, 'ab': 0.6303649282355167, 'ac': 0.6490734762205318, 'bc': 0.08453193878473678

------------------------------- [Forward Propagation] Training with sample indexed: 2000 -------------------------------

Predicted value at the output nodes: `xw': -0.5157700093872639
Loss for training sample indexed 2000: 38.09662138083069

Estimated partial derivatives of vars wrt learnable parameters:
- k=xx: xx: None, 'xa': 1.1654406692440666, 'xz': None, 'xw': None, 'xy': None
- k=xa: xx: None, 'xa': None, 'xz': None, 'xw': None, 'xy': None
- k=xz: xx: 0.0845287913638082, 'xa': None, 'xz': None, 'xw': None, 'xy': 0.9999999999998899
- k=xw: xx: 0.8134218926023218, 'xa': None, 'xz': 1.1568387019889048, 'xw': None, 'xy': None
- k=xy: xx: 0.6303669260676603, 'xa': None, 'xz': None, 'xw': None, 'xy': None

[sample index: 2000]: input val: `xa': -0.5827703346221946 vals for learnable parameters: `cd': 0.813439110108809, 'ab': 0.6303417051886069, 'ac': 0.6490421190453514, 'bc': 0.08452882457407629

------------------------------- [Forward Propagation] Training with sample indexed: 1000 -------------------------------
What you see on the previous slide is the output every $1000^{th}$ iteration in the training loop. The basic format of the information that is presented is:

- input values for independent variables
- predicted value at the output variables
- loss
- estimated partial derivatives of vars wrt other pars
- estimated partial derivatives of vars wrt learnable parameters

The partial derivatives are stored as two dictionaries at each node of the DAG. One dictionary is for the derivative of the output variable vis-a-vis all other variables in the network. [Obviously, depending on the network topology, some of these partials will be undefined.] The other dictionary is for the partials of the output variable with respect to all the learnable parameters. [Again, some of these partials will also remain undefined depending on the network topology.]
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.

Shown on the next slide is the loss as a function of iterations. Do not expect to see the loss decrease — simply because our DAG is not a learning network. For one, it does not include nonlinear activations at the nodes. My goal in this exercise was only to make you familiar with the notion of a DAG for forward and backward data flow.
The fact that the loss is not decreasing is NOT surprising since there are no activation functions in our DAG. The goal of this exercise was merely to illustrate forward and backward dataflow in a DAG and how in principle you can calculate the partial derivatives during forward propagation for updating the parameters in backpropagation.
<table>
<thead>
<tr>
<th></th>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revisiting Gradient Descent</td>
</tr>
<tr>
<td>2</td>
<td>Problems with a Vanilla Application of GD</td>
</tr>
<tr>
<td>3</td>
<td>Stochastic Gradient Descent</td>
</tr>
<tr>
<td>4</td>
<td>A Computational Graph Primer</td>
</tr>
<tr>
<td>5</td>
<td>Primer Demo 1: graph_based_dataflow.py</td>
</tr>
<tr>
<td>6</td>
<td><strong>Primer Demo 2: one_neuron_classifier.py</strong></td>
</tr>
<tr>
<td>7</td>
<td>Primer Demo 3: multi_neuron_classifier.py</td>
</tr>
<tr>
<td>8</td>
<td>Primer Demo 4: verify_with_torchnn.py</td>
</tr>
<tr>
<td>9</td>
<td>Autograd</td>
</tr>
<tr>
<td>10</td>
<td>Extending Autograd</td>
</tr>
<tr>
<td>11</td>
<td>Forward and Backward Calculations in an Ordinary NN</td>
</tr>
</tbody>
</table>

Purdue University
I’ll now present what’s got to be smallest possible example for illustrating forward propagation of the input data while computing the partial derivatives of the output with respect to the learnable parameters and then updating the parameters during backprop.

A good starting point for understanding the demo based on the one-neuron model is the data that I use for this demo. As shown below, I generate random data for two different Gaussian distributions, with different means and variances. The data generator is happy to create the multivariate data of any arbitrary dimensionality.

```python
def gen_training_data(self):
    num_input_vars = len(self.independent_vars)
    training_data_class_0 = []
    training_data_class_1 = []
    for i in range(self.dataset_size // 2):
        for_class_0 = np.random.standard_normal( num_input_vars )
        for_class_1 = np.random.standard_normal( num_input_vars )
        for_class_0 = for_class_0 + 2.0
        for_class_1 = for_class_1 * 2 + 4.0
        training_data_class_0.append( for_class_0 )
        training_data_class_1.append( for_class_1 )
    self.training_data = {0 : training_data_class_0, 1 : training_data_class_1}
```
Here is the data loader that for the demonstration. Note the implementation of the \_getitem()\ method: it randomly chooses either a sample of Class 0 or a sample of Class 1. Also pay attention to data scaling and batch construction in the getbatch() method. The randomization used in creating a new batch means that there is no need to bring in the idea of epochs for training. [With epochs, you create a fresh randomization of the training data for each epoch.]

class DataLoader:
  def \_init\_\_(self, training_data, batch_size):
    self\.training_data = training_data
    self\.batch_size = batch_size
    self\.class_0_samples = [(item, 0) for item in self\.training_data[0]]
    self\.class_1_samples = [(item, 1) for item in self\.training_data[1]]
  def \_getitem\_(self):
    cointoss = random\.choice([0,1])
    if cointoss == 0:
      return random\.choice(self\.class_0_samples)
    else:
      return random\.choice(self\.class_1_samples)
  def getbatch(self):
    batch_data, batch_labels = [], []
    maxval = 0.0
    for _ in range(self\.batch_size):
      item = self\.\_getitem\_(
      if np\.max(item[0]) > maxval:
        maxval = np\.max(item[0])
      batch_data\.append(item[0])
      batch_labels\.append(item[1])
    batch_data = [item/maxval for item in batch_data]
    batch = [batch_data, batch_labels]
    return batch
Shown below is the call to the Primer constructor for constructing a one-neuron model.

The neuron is constructing by parsing the expression that is supplied through the expressions constructor option. Since the node names start with the letter 'x', we have four input nodes and four corresponding learnable link weights in this example.

```python
import ComputationalGraphPrimer

cgp = ComputationalGraphPrimer(
    one_neuron_model = True,
    expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    output_vars = ['xw'],
    dataset_size = 5000,
    learning_rate = 1e-3,
    training_iterations = 40000,
    batch_size = 8,
    display_loss_how_often = 100,
)

cgp.parse_expressions()
cgp.display_network2()
cgp.gen_training_data()
cgp.run_training_loop_one_neuron_model()
```
Shown below is the network constructed by the constructor call on the previous slide.

The learnable parameters associated with the links are not displayed.

The one-neuron network. The data aggregated at the output node is subject to Sigmoid activation.
Here is the training loop for the one-neuron model. Forward propagation is dispatched to the function `forward_prop_one_neuron_model()`. After that function has returned the prediction and the partial derivatives, we dispatch a call to the function `backprop_and_update_params_one_neuron_mode()`. As required by SGD, the values returned by the forward function are averaged over the batch before they are shipped off for backprop.

```python
def run_training_loop_one_neuron_model(self):
    training_data = self.training_data
    self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
    self.bias = random.uniform(0,1)  ## used in the forward function
    data_loader = DataLoader(self.training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_literations = 0.0
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples, class_labels = data[0], data[1]
        y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(data_tuples)
        loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])
        loss_avg = loss / float(len(class_labels))
        avg_loss_over_literations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_literations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_literations)
            print("[iter=%d] loss = %.4f" % (i+1, avg_loss_over_literations))
            avg_loss_over_literations = 0.0
        y_errors = list(map(operator.sub, class_labels, y_preds))
        y_error_avg = sum(y_errors) / float(len(class_labels))
        deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
        data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
        data_tuple_avg = list(map(operator.truediv, data_tuple_avg, [float(len(class_labels))] * len(class_labels)))
        self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
```
Shown below are the forward and the backprop functions for the one-neuron case. In addition to the prediction, the forward function need return only the partial derivative of the Sigmoid — this is justified on the next slide.

As shown on the previous slide, we send to the backprop function the batch averaged values for the prediction error and the partial derivatives.

```python
def forward_prop_one_neuron_model(self, data_tuples_in_batch):
    output_vals, deriv_sigmoids = [], []
    for vals_for_input_vars in data_tuples_in_batch:
        input_vars = self.independent_vars
        vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
        exp_obj = self.exp_objects[0]
        output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params)
        output_val = output_val + self.bias
        output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
        deriv_sigmoid = output_val * (1.0 - output_val)
        output_vals.append(output_val)
        deriv_sigmoids.append(deriv_sigmoid)
    return output_vals, deriv_sigmoids

def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid):
    input_vars = self.independent_vars
    vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
    vals_for_learnable_params = self.vals_for_learnable_params
    for i,param in enumerate(self.vals_for_learnable_params):
        step = self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
        self.vals_for_learnable_params[param] += step
    self.bias += self.learning_rate * y_error * deriv_sigmoid
```
The output of the neuron is given by the following formula in which $g()$ is the Sigmoid activation and $N$ is the dimensionality of the data:

$$y_{pred} = g \left( \sum_{i=n}^{N} x_i a_i + b \right)$$

(16)

Therefore, we can write for the loss for each training sample where $y_{gt}$ is the true label for the sample:

$$L = \left[ y_{gt} - g \left( \sum_{i=1}^{N} x_i a_i + b \right) \right]^2$$

(17)

We only need to take the derivatives of the loss with respect to the learnable parameters (since there is no point to backpropagating the loss to the input layer). Using $y_{err} = y_{gt} - y_{pred}$ for the prediction error and $\theta$ for the argument to the activation function:

$$\frac{\partial L}{\partial a_i} = -2y_{err} \frac{\partial}{\partial a_i} g \left( \sum_{i=1}^{N} x_i a_i + b \right)$$

$$= -2y_{err} \frac{\partial g}{\partial \theta} \frac{\partial}{\partial a_i} \left( \sum_{i=1}^{N} x_i a_i + b \right)$$

$$= -2y_{err} \frac{\partial g}{\partial \theta} x_i$$
Shown below is how the loss decreases with the training iterations for the on-neuron model with Sigmoid activation.

(a) $lr = 1e^{-3}$

(b) $lr = 5e^{-2}$

Loss vs. training-iterations for two different learning rates
In Demo 3, I will generalize what was demonstrated in Demo 2 to a multi-layer neural network.

I will continue to use the same data source and the same data loader I presented for the previous demo.

Shown below is the constructor call that creates a 3-layer (including the input layer) network. That is, we now have an input layer, a hidden layer, and an output layer.

```python
cgp = ComputationalGraphPrimer(
    num_layers = 3,
    layers_config = [4,2,1],
    expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                   'xz=bp*xp+bq*xq+br*xr+bs*xs',
                   'xo=cp*xw+cq*xz'],
    output_vars = ['xo'],
    dataset_size = 5000,
    learning_rate = 1e-3,
    training_iterations = 40000,
    batch_size = 8,
    display_loss_how_often = 100,
    debug = True,
)
```
As you will see in the script `multi_neuron_classifier.py` in the Examples directory, the first call you invoke on the instance of the Primer class constructed above is:

```python
cgp.parse_multi_layer_expressions()
```

If you peer inside the code for the method `parse_multi_layer_expressions()`, you will notice that each expression that you supply through the constructor option `expressions` is stored as instance of the `Exp` class with calls like

```python
exp_obj = Exp(exp, right, left, right_vars, right_params)
```

The class `Exp` used above is defined as shown below:

```python
class Exp:
    def __init__(self, exp, body, dependent_var, right_vars, right_params):
        self.exp = exp
        self.body = body
        self.dependent_var = dependent_var
        self.right_vars = right_vars
        self.right_params = right_params
```

The values for the variables `exp`, `right`, `left`, `right_vars`, `right_params` are extracted from the individual expressions as you would expect.
Shown below is the network constructed by the constructor call shown on Slide 52. The learnable parameters associated with the links are not displayed. Sigmoid activation is used for the data aggregated at each node.

Blame my graphics package for producing a network diagram that does not look like what you are accustomed to seeing for a neural network. The input nodes — xp, xq, xr, and xs — are all at the lower-left, the hidden layer consists of the two nodes xw and xs. Finally, the output node is xo.
Here is the training loop for the multi-neuron model. Forward propagation is dispatched to the function `forward_prop_multi_neuron_model()`. Note that the predictions and the derivatives calculated by `forward` are communicated directly to the backprop function `backprop_and_update_params_multi_neuron_model()` as you will see.

def run_training_loop_multi_neuron_model(self):
    training_data = self.training_data
    self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
    self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]
    data_loader = DataLoader(self.training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_literations = 0.0
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = data[0]
        class_labels = data[1]
        self.forward_prop_multi_neuron_model(data_tuples)
        predicted_labels_for_batch = self.forward_prop_vals_at_layers[self.num_layers-1]
        y_preds = [item for sublist in predicted_labels_for_batch for item in sublist]
        loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])
        loss_avg = loss / float(len(class_labels))
        avg_loss_over_literations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_literations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_literations)
            print("[iter=%d] loss = %.4f" % (i+1, avg_loss_over_literations))
        y_errors = list(map(operator.sub, class_labels, y_preds))
        y_error_avg = sum(y_errors) / float(len(class_labels))
        self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)
    plt.figure()
    plt.plot(loss_running_record)
    plt.show()
Shown on the next slide is the forward function for the multi-neuron case. In addition to the predictions for the samples in a batch, the forward function must also calculate the partial derivatives needed during the backprop step.

In the code shown on the next slide, the expressions to evaluate for computing the pre-activation values at a node are stored at the layer in which the nodes reside. That is, the dictionary look-up `self.layer_exp_objects[layer_index]` returns the Expression objects for which the left-side dependent variable is in the layer pointed to `layer_index`. So the example shown above, `self.layer_exp_objects[1]` will return two Expression objects, one for each of the two nodes in the second layer of the network (that is, layer indexed 1).

The pre-activation values obtained by evaluating the expressions at each node are then subject to Sigmoid activation, followed by the calculation of the partial derivative of the output of the Sigmoid function with respect to its input.
In the forward, the values calculated for the nodes in each layer are stored in the dictionary `self.forw_prop_vals_at_layers[ layer_index ]` and the gradients values calculated at the same nodes in the dictionary: `self.gradient_vals_for_layers[ layer_index ]`.

```python
def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
    self.forw_prop_vals_at_layers = {i : [] for i in range(self.num_layers)}
    self.gradient_vals_for_layers = {i : [] for i in range(1, self.num_layers)}
    for vals_for_input_vars in data_tuples_in_batch:
        self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
        for layer_index in range(1, self.num_layers):
            input_vars = self.layer_vars[layer_index-1]
            if layer_index == 1:
                vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
            output_vals_arr = []
            gradients_val_arr = []
            for exp_obj in self.layer_exp_objects[layer_index]:
                output_val = self.eval_expression(exp_obj.body, vals_for_input_vars_dict,
                                                   self.vals_for_learnable_params, input_vars)
                output_val = output_val + self.bias[layer_index-1]
                ## apply sigmoid activation:
                output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                output_vals_arr.append(output_val)
                ## calculate partial of the activation function as a function of its input
                deriv_sigmoid = output_val * (1.0 - output_val)
                gradients_val_arr.append(deriv_sigmoid)
                vals_for_input_vars_dict[ exp_obj.dependent_var ] = output_val
            self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
            self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)
```
Shown on Slide 64 is the backprop function. The following statements toward the end of the training loop code on Slide 58:

```python
y_errors = list(map(operator.sub, class_labels, y_preds))
y_error_avg = sum(y_errors) / float(len(class_labels))
```

mean that we send to the backprop function the batch averaged values for the prediction errors. The batch averaged values for the partial derivatives of the Sigmoid activation are calculated in the backprop function.

To explain the backprop logic, let’s denote the prediction error in layer $l$ by $y_{err,l}$ and the parameters matrix for the link weights between the layers indexed $l−1$ and $l$ by $w_l$. I’ll use $x_l$ to denote the forward propagated values at layer $l$.

About the shape of the link matrix $w_l$, that should be clear from the relationship $x_l = w_l \cdot x_{l−1}$. Therefore, if layer $l−1$ has 4 nodes and the layer $l$ has 2 nodes, the link matrix $w_l$ will be of shape $2 \times 4$. 
The backprop step is defined by the following two operations:

1. Backprop the prediction error from layer \( l \) to layer indexed \( l - 1 \):

   \[
   y_{err,l-1} = w_l^T \cdot y_{err,l}
   \]  
   (19)

   For backprop, for the \( 2 \times 4 \) matrix \( w_l \) mentioned on the previous slide, the transpose matrix shown above would be \( 4 \times 2 \).

2. Calculate the gradient of the loss with respect to the parameters in the matrix \( w_l \) by

   \[
   \nabla_{w_l} L = y_{err,l} \otimes x_{l-1}^T
   \]  
   (20)

   Note that the right hand side here involves an outer product that would result in a \( 2 \times 4 \) matrix for the example of 4 nodes in layer \( l - 1 \) and 2 in layer \( l \).

Gradient of the loss with respect to the learnable parameters of a layer above must be “mediated” by the partial derivatives of the activation function for the layer.
About the backprop code on the next slide, note that loop index variable `back_layer_index` starts with the index of the last layer. For our 3-layer example shown in the constructor call earlier, the value of `back_layer_index` starts with a value of 2, its next value is 1, and that’s it.

As previously mentioned, the value of the `y_error` parameter is the batch averaged values for the prediction errors for the training samples in a batch.

Note the use of the syntax `input_vals_avg = [sum(x) for x in zip(*input_vals)]` for averaging the input samples in a batch. If the dimensionality of input data is 8, the average will be an 8-dimensional vector for the batch.

Again as required by SGD, note the batch-based averaging of the Sigmoid derivatives. The syntax `[sum(x) for x in zip(*deriv_sigmoid)]` that you see is a bit of an overkill for what it accomplishes. I say so because the value of `deriv_sigmoid` is a list of single element lists, which each value being the partial of the Sigmoid activation for one input sample.
Shown below is the backprop function for the multi-neuron case.

```python
def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
    # backproped prediction error:
    pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
    pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
    for back_layer_index in reversed(range(1,self.num_layers-1)):
        input_vals = self.forw_prop_vals_at_layers[back_layer_index -1] # list of tuples in batch
        input_vals_avg = [sum(x) for x in zip(*input_vals)] # summing the data in tuples element wise
        input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
        deriv_sigmoid = self.gradient_vals_for_layers[back_layer_index] # list of gradients for batch
        deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)] # batch num of one-element lists
        deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg, [float(len(class_labels))] * len(class_labels)))
        vars_in_layer = self.layer_vars[back_layer_index] ## a list like ['xo']
        vars_in_next_layer_back = self.layer_vars[back_layer_index - 1] ## a list like ['xw', 'xz']
        layer_params = self.layer_params[back_layer_index]
        ## note that layer_params are stored in a dict like
        ## 1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]
        transposed_layer_params = list(zip(*layer_params)) ## creating a transpose of the link matrix
        backproped_error = [None] * len(vars_in_next_layer_back)
        for k,varr in enumerate(vars_in_next_layer_back):
            for j,var2 in enumerate(vars_in_layer):
                backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                          pred_err_backproped_at_layers[back_layer_index][i]
                                          for i in range(len(vars_in_layer))])
        pred_err_backproped_at_layers[back_layer_index - 1] = backproped_error
        input_vars_to_layer = self.layer_vars[back_layer_index-1]
        for j,var in enumerate(vars_in_layer):
            for i,param in enumerate(layer_params):
                gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]
                step = self.learning_rate * gradient_of_loss_for_param * deriv_sigmoid_avg[j]
                self.vals_for_learnable_params[param] += step
                self.bias[back_layer_index-1] += self.learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) * \
                sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
```

Purdue University
Shown below is how the loss decreases with the training iterations for the multi-neuron model with Sigmoid activations:

(a) $lr = 1e^{-3}$

(b) $lr = 5e^{-2}$

Loss vs. training-iterations for two different learning rates
Outline

1 Revisiting Gradient Descent 7
2 Problems with a Vanilla Application of GD 17
3 Stochastic Gradient Descent 19
4 A Computational Graph Primer 30
5 Primer Demo 1: graph_based_dataflow.py 33
6 Primer Demo 2: one_neuron_classifier.py 45
7 Primer Demo 3: multi_neuron_classifier.py 54
8 Primer Demo 4: verify_with_torchnn.py 66
9 Autograd 73
10 Extending Autograd 77
11 Forward and Backward Calculations in an Ordinary NN 89
Demo 4 Script: verify_with_torchnn.py

- In Demo 4, my goal is to compare the performance of the scripts
  
  one_neuron_classifier.py
  multi_neuron_classifier.py

  with similar networks constructed using components from PyTorch’s torch.nn module.

- There are two reasons for this comparison:

  1. If the torch.nn based code says that it is possible to learn from a one-neuron network implementation, we would want the same from the script one_neuron_classifier.py. And the same would go for the three-layer “4-2-1” neural network in multi_neuron_classifier.py.

  2. An even more important reason for the comparison is for you to be dazzled by the performance improvement you can get through step optimization in parameter update formulas when using torch.nn.
Shown on the next slide is the constructor call to the Primer class in the script `verify_with_torchnn.py`

The lines following the constructor call that are labeled (A) and (B) determine whether your `torch.nn` based network will use a single neuron as in the `one_neuron_classifier.py` demo, or whether it will use multiple neurons as in the script `multi_neuron_classifier.py`.

When the option value is `one_neuron`, we use the class `OneNeuronNet` for the learning network and when the option is `multi_neuron` we use the class `MultiNeuronNet`. These two classes based on `torch.nn` will be shown later.

**IMPORTANT:** the expression `option` used in the constructor call is used ONLY for telling the system what dimensional multivariate data to generate for training the networks. The expression supplied serves absolutely no other purpose.
Note also there I have shown three different values for the parameter `learning_rate`. When using the `multi_neuron` option in line (B), you will get the best results with the learning rate set to $10^{-6}$.

On the other hand, if you choose the call in line labeled (A), your best choices for the learning rate are either $10^{-3}$ or $5 \times 10^{-2}$.

cgp = ComputationalGraphPrimer(
    expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],  # Only used to determine the data dimensionality
    dataset_size = 5000,
    learning_rate = 1e-6,  # For the multi-neuron option below
    learning_rate = 1e-3,  # For the one-neuron option below
    learning_rate = 5 * 1e-2,  # Also for the one-neuron option below
    training_iterations = 40000,
    batch_size = 8,
    display_loss_how_often = 100,
)

## This call is needed for generating the training data:
cgp.parse_expressions()
cgp.gen_training_data()  
# cgp.run_training_with_torchnn('one_neuron')  ## (A)
cgp.run_training_with_torchnn('multi_neuron')  ## (B)
verify_with_torchnn.py (contd.)

- Shown below are the two `torch.nn` based classes that are used for the “verification” of the performance of the scripts `one_neuron_classifier.py` and `multi_neuron_classifier.py` scripts presented earlier.

- Both these classes are a part of the implementation of the training loop shown on the next slide.

```python
class OneNeuronNet(torch.nn.Module):
    
    This class is used when the parameter 'option' is set to 'one_neuron' in the call to this training function.
    
    def __init__(self, D_in, D_out):
        torch.nn.Module.__init__(self)
        self.linear = torch.nn.Linear(D_in, D_out)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        h_out = self.linear(x)
        y_pred = self.sigmoid(h_out)
        return y_pred

class MultiNeuronNet(torch.nn.Module):
    
    This class is used when the parameter 'option' is set to 'multi_neuron' in the call to this training function.
    
    def __init__(self, D_in, H, D_out):
        torch.nn.Module.__init__(self)
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h.relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h.relu)
        return y_pred
```
Here is the training loop for using the torch.nn based classes for the one-neuron and multi-neuron models.

```python
def run_training_with_torchnn(self, option):
    training_data = self.training_data
    # Code for the two class definition shown on the previous slide goes here
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 0.0
    if option == 'one_neuron':
        N,D_in,D_out = self.batch_size,self.input_size,self.output_size
        model = OneNeuronNet(D_in,D_out)
    elif option == 'multi_neuron':
        N,D_in,H,D_out = self.batch_size,self.input_size,2,self.output_size
        model = MultiNeuronNet(D_in,H,D_out)
    else:
        sys.exit("\n\nThe value of the parameter 'option' not recognized\n\n")
    criterion = torch.nn.MSELoss(reduction='sum')
    optimizer = torch.optim.SGD(model.parameters(), self.learning_rate)
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = torch.FloatTensor( data[0] )
        class_labels = torch.FloatTensor( data[1] )
        # We need to convert the shape torch.Size([8]) into the shape torch.Size([8, 1]):
        class_labels = torch.unsqueeze(class_labels, 1)
        y_preds = model(data_tuples)
        loss = criterion(y_preds, class_labels)
        optimizer.zero_grad()
        loss.backward() optimizer.step()
        avg_loss_over_iterations += loss
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
            print("[iter=%d] loss = %.4f " % (i+1, avg_loss_over_iterations))
            avg_loss_over_iterations = 0.0
    plt.figure()
    plt.plot(loss_running_record)
    plt.show()
```

Shown below is how the loss decreases with the training iterations for the multi-neuron and the one-neuron cases when using \texttt{torch.nn} based networks:

(a) multi-neuron network  
(b) one-neuron network
Outline

1. Revisiting Gradient Descent  
2. Problems with a Vanilla Application of GD  
3. Stochastic Gradient Descent  
4. A Computational Graph Primer  
5. Primer Demo 1: graph_based_dataflow.py  
6. Primer Demo 2: one_neuron_classifier.py  
7. Primer Demo 3: multi_neuron_classifier.py  
8. Primer Demo 4: verify_with_torchnn.py  
9. Autograd  
10. Extending Autograd  
11. Forward and Backward Calculations in an Ordinary NN
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.
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:

```python
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. When you are using the automation provided by `torch.nn`, that module takes care of setting the `requires_grad` property true for all the learnable parameters.

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

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revisiting Gradient Descent</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Problems with a Vanilla Application of GD</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Stochastic Gradient Descent</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>A Computational Graph Primer</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Primer Demo 1: graph_based_dataflow.py</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Primer Demo 2: one_neuron_classifier.py</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Primer Demo 3: multi_neuron_classifier.py</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>Primer Demo 4: verify_with_torchnn.py</td>
<td>66</td>
</tr>
<tr>
<td>9</td>
<td>Autograd</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td><strong>Extending Autograd</strong></td>
<td>77</td>
</tr>
<tr>
<td>11</td>
<td>Forward and Backward Calculations in an Ordinary NN</td>
<td>89</td>
</tr>
</tbody>
</table>

Purdue University
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 of the inner class AutogradCustomization in my ComputationalGraphPrimer module.
The Examples subdirectory of the Primer distribution contains a script named `extending_autograd.py` with the following code:

```python
from ComputationalGraphPrimer import *
cgp = ComputationalGraphPrimer( ## (A)
    learning_rate = 1e-3,
    epochs = 5,
)

ext_auto = ComputationalGraphPrimer.AutogradCustomization( ## (B)
    cgp = cgp,
    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 `ComputationalGraphPrimer` class. Subsequently, in line (B), we construct an instance of the inner class `AutogradCustomization` that has the code for demonstrating how to extend PyTorch's Autograd module.
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:

```python
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.
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:

```python
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.cgp.device) # (5)
    w2 = w2.to(self.cgp.device) # (6)
    w1.requires_grad_() # (7)
    w2.requires_grad_() # (8)
    Loss = [] # (9)
    for epoch in range(self.cgp.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.cgp.device: # (15)
                x,y = x.to(self.cgp.device), y.to(self.cgp.device) # (16)
            y_pred = x.view(1,-1).mm(w1).clamp(min=0).mm(w2) # (17)
```

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

Purdue University
Extending Autograd

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

```python
loss = (y_pred - y).pow(2).sum()  # (18)
loss.backward()  # (19)
with torch.no_grad():  # (20)
    w1 -= self.cgp.learning_rate * w1.grad  # (21)
    w2 -= self.cgp.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 80. 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:
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 ComputationalGraphPrimer module contains the class definition shown on the next slide. The name of this class is DoSillyWithTensor.
Shown below is the class DoSillyWithTensor mentioned on the previous slide:

class DoSillyWithTensor(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        input_orig = input.clone().double()
        input = input.to(torch.uint8).double()
        diff = input_orig.sub(input)
        ctx.save_for_backward(diff)
        return input

    @staticmethod
    def backward(ctx, grad_output):
        diff, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input = grad_input + diff
        return grad_input

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.
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 `ComputationalGraphPrimer` module:

```python
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.cgp.device)
    w2 = w2.to(self.cgp.device)
    w1.requires_grad_()
    w2.requires_grad_()
    Loss = []
    for epoch in range(self.cgp.epochs):
        for i, data in enumerate(self.training_data):
            do_silly = ComputationalGraphPrimer.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.cgp.device), y.to(self.cgp.device)
```

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

Extending Autograd

Excluding Autograd

\[
y_{\text{pred}} = x.{\text{view}}(1,-1).{\text{mm}}(w1).{\text{clamp}}(\text{min}=0).{\text{mm}}(w2)
\]

\[
\text{loss} = (y_{\text{pred}} - y).{\text{pow}}(2).{\text{sum}}()
\]

\[
\text{loss}.{\text{backward}}()
\]

\[
\text{with torch.no_grad():}
\]

\[
w1 -= \text{self}.cgp.\text{learning}.\text{rate} * w1.\text{grad}
\]

\[
w2 -= \text{self}.cgp.\text{learning}.\text{rate} * w2.\text{grad}
\]

\[
w1.\text{grad}.\text{zero}_{\text{()}}
\]

\[
w2.\text{grad}.\text{zero}_{\text{()}}
\]

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.
About the Code Examples Shown in This Section

- The code shown on Slides 81 through 86 represented examples of manually constructed neural networks. I say “manually”, because we had to define the matrices $w_1$ and $w_2$ for the link weights, with $w_1$ representing the learnable weights between the input layer and the hidden layer and $w_2$ doing the same for the weights between the hidden layer and the output layer.

- When you specify a network manually, you must also identify for PyTorch as to which the data entities constitute learnable weights. You can only do that if your learnable entities are tensors and if you have set the `requires_grad` attributes of those tensors to True. That’s what was done in lines (8) and (9) of the code shown on Slide 81.

- When you specify a network manually, you must also specify what it takes to calculate the predictions in the output layer. We will consider an example on the next slide.
In the code shown on Slide 81, the following statement calculates the prediction at the output:

\[
y_{\text{pred}} = x.\text{view}(1,-1).\text{mm}(w1).\text{clamp}(\text{min}=0).\text{mm}(w2)
\]

Do you understand what is being accomplished by the calls to \text{view()} and \text{mm()}?

**EXPLANATION:** In the above example, the input data element \(x\) is 2-dimensional. And the hidden layer in the code on Slide 81 is 10-dimensional. So \(w_1\) is \(2 \times 10\). We can calculate the pre-activation data aggregation at the hidden layer either as \(w_1^T x\) or as \(x^T w_1\). Either one of these will return a 10-dimensional entity, the first as a column vector and the second as a row vector of the same dimensionality. The implementation shown above uses the latter. We call on \text{view()} to reshape \(x\) as a \(1 \times 2\) matrix. The cool thing about calling \text{view()} with one of its argument set to -1 is that it lets PyTorch figure out on its own what that should be. In what it does, \text{torch.view()} is similar to what is accomplished by numpy’s \text{reshape()} function. However, unlike numpy’s \text{reshape()}, the new tensor returned by \text{torch.view()} shares the underlying data with the original tensor. In that sense, what \text{torch.view()} returns is just a new "view" of the old data. Said another way, numpy’s \text{reshape()} allocates fresh memory for the reshaped data object. On the other hand, \text{tensor.view()} does not make a copy of anything.

**EXPLANATION:** \text{torch.mm()} does a regular matrix multiplication of two argument matrices. So if you declare \text{mat1 = torch.randn(2,4)} and \text{mat2 = torch.randn(4,6)}, the call \text{mat1.mm(mat2)} will return a 2x6 matrix.
Outline

1. Revisiting Gradient Descent 7
2. Problems with a Vanilla Application of GD 17
3. Stochastic Gradient Descent 19
4. A Computational Graph Primer 30
5. Primer Demo 1: graph_based_dataflow.py 33
6. Primer Demo 2: one_neuron_classifier.py 45
7. Primer Demo 3: multi_neuron_classifier.py 54
8. Primer Demo 4: verify_with_torchnn.py 66
9. Autograd 73
10. Extending Autograd 77
11. Forward and Backward Calculations in an Ordinary NN 89

Purdue University
The focus of this section is exclusively the forward calculation of the gradients and the backpropagation of the prediction error in simple 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.

For simple neural networks, the gradients that need to be remembered during the forward propagation of the data are the partial derivatives of the output of the activation function with respect to its input.

Say we have a neural network with one input layer, one hidden layer, and one output layer. 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.
Additionally, let $y_{\text{pred}}$ denote the predicted output at the final layer, and $y_{\text{gt}}$ the ground-truth for a given input $x$. We can write for the loss $L$

$$L = (y_{\text{gt}} - y_{\text{pred}})^T (y_{\text{gt}} - y_{\text{pred}})$$

$$= [y_{\text{gt}} - (w_2 \cdot h_{\text{rel}})]^T [y_{\text{gt}} - (w_2 \cdot h_{\text{rel}})]$$

(21)

where $h_{\text{rel}}$ is the output of the hidden layer post the activation. Note that I am using the “column vector” representation for a vector in a matrix-vector product.

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

$$\frac{\partial L}{\partial w_2} = -2[y_{\text{gt}} - (w_2 \cdot h_{\text{rel}})] \cdot h_{\text{rel}}^T$$

$$= -2[y_{\text{gt}} - y_{\text{pred}}] \cdot h_{\text{rel}}^T$$

$$= -2y_{\text{err}} \otimes h_{\text{rel}}$$

(22)

where '$\otimes$' stands for the outer product of the two vectors.

Therefore, the link weights $w_2$ can be updated through the outer product of the prediction error vector at the output of the second layer and the post-activation vector at the output of the first layer.
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:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{\text{pred}}} \cdot \frac{\partial y_{\text{pred}}}{\partial h_{\text{rel}}} \cdot \frac{\partial h_{\text{rel}}}{\partial w_1}$$

$$= \frac{\partial L}{\partial y_{\text{pred}}} \cdot w_2 \cdot \frac{\partial h_{\text{rel}}}{\partial w_1}$$

$$= -y_{\text{err}}^T \cdot w_2 \cdot \frac{\partial h_{\text{rel}}}{\partial w_1}$$

$$= -(w_2^T \cdot y_{\text{err}})^T \cdot \frac{\partial h_{\text{rel}}}{\partial w_1}$$

$$= -(w_2^T \cdot y_{\text{err}})^T \cdot \frac{\partial g(w_1 \cdot x_{\text{input}})}{\partial w_1}$$

$$= -(y_{\text{err},1}^T \cdot \frac{\partial g(\theta)}{\partial \theta} \cdot \frac{\partial (w_1 \cdot x_{\text{input}})}{\partial w_1}$$

$$= -\left( y_{\text{err},1} \frac{\partial g(\theta)}{\partial \theta} \right) \otimes x_{\text{input}}$$

(23)

where $y_{\text{err},1}$ is the prediction error backproped to the post-activation point for the first layer and $\theta = w_1 \cdot x_{\text{input}}$ is the operating point on the activation function curve. [Note that the second equation follows from the fact that $y_{\text{pred}} = w_2 \cdot h_{\text{rel}}$, and the third equation from the fact that $L = [y_{\text{gt}} - y_{\text{pred}}]^T [y_{\text{gt}} - y_{\text{pred}}] = 2y_{\text{err}}^T y_{\text{err}}$. In the fifth equation on the right, $g()$ is the activation function for the hidden layer. In the same equation, I have used the fact that $h_{\text{rel}} = g(w_1 \cdot x_{\text{input}})$.]
Note that when you compare Eq. (23) with the form shown in Eq. (22), you can also see how the updating of the link weights and the backpropagation of the prediction errors can be extended to a neural network of any arbitrary depth.

As for the activation function $g()$ on the previous slide, if we use the ReLU activation, we have

$$h_{rel} = ReLU(w_1 \cdot x) \quad (24)$$

which is the same as saying that

$$h_{rel} = \begin{cases} w_1 \cdot x & \text{when } w_1 \cdot x > 0 \\ 0 & \text{otherwise} \end{cases} \quad (25)$$

On the other hand, for the Sigmoid activation, if $\theta$ is its input, its output will lie in the interval $(0, 1)$ and is given by

$$g(\theta) = \frac{1}{1 - e^{-\theta}} \quad (26)$$

and its derivative is given by

$$\frac{\partial g(\theta)}{\partial \theta} = g(\theta)(1 - g(\theta)) \quad (27)$$
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 $l$, and if $error_l$ is the prediction error backproped to the post-activation point for layer $l$:

   $\text{gradient of loss wrt } w_l = (error_l.g'(x)) \otimes output_{l-1}$  \hspace{1cm} (28)

   where $output_{l-1}$ is the output of the previous layer (and, therefore, the input to layer $l$) and $g'(x)$ is the partial derivative of the activation function;

2. Now backprop the prediction error $error_l$ to the post-activation point in the previous layer:

   $error_{l-1} = error_l.mm(w_l)$  \hspace{1cm} (29)

3. With regard to $g'$, if you are using the ReLU activation, all you have to do to just zero out those elements where the forward propagated values to the activation function are negative. That gives you the backpropagated error on the preactivation side of the layer $l - 1$. 