Backpropagation

Now we are ready to turn to the backbone of neural network training: the computation of the gradients and how they are propagated through the network. Computing gradients analytically can be straight-forward, but evaluating them numerically in an efficient way on a computer might be less so. This is where backpropagation comes into play: It computes the gradients necessary to update the weights in the network and does so efficiently. As the computation starts from the top layer, it is sometimes also called the backward pass.

Thanks to TensorFlow and Co. you usually don't have to compute the gradients yourself. Nevertheless, if you want to make headway in terms of deep learning it is important to really understand backprop and how gradients can be derived. In the following, we will not implement the backprop algorithm, but compute the gradients and use them directly for gradient descent. Later in the lecture, you will learn more about how backprop is implemented.

The final implementation of the backward pass is not more than 7 lines of code for our simple network. However, it is not always easy to see how one arrives at these lines. In the following sections, we are going to explain all the details necessary to fully comprehend the implementation of the backward pass. Doing so, we use some calculus rules that you should be familiar with, but which will be treated again in the lecture during the upcoming weeks. So, don't worry if you don't yet understand all the details. However, we do encourage you to work through this exercise already and certainly come back to it when the respective content was treated in the lecture.

To compute the gradients in a neural network, it mainly boils down to knowing the multi-variate chain rule, how to differentiate vector and matrix quantities and doing all this very carefully to avoid easy mistakes. In the following, we'll guide you through this step-by-step.

Chain Rule

Suppose that $y=g(x)$ and $z=f(g(x))=f(y)$. The chain rule then states:

$$\partial z \partial x = \partial z \partial y \partial y \partial x$$

This can be generalized to the non-scalar case. Suppose $x \in R^m$, $y \in R^n$ and $z \in R$.
If $y=g(x)$ and $z=f(y)$, the chain rule says

$$\partial z \partial x_i = \sum_j \partial z \partial y_j \partial y_j \partial x_i$$

For the multivariate version of the chain rule, it is sometimes easier to think of it in terms of the Jacobian. Assume $h: R^n \to R^m$ is a vector-valued function. It's Jacobian $J \in R^{m \times n}$ is defined as:

$$J = \begin{bmatrix} \partial h_1 \partial x_1 & \dots & \partial h_1 \partial x_n \\ \vdots & \ddots & \vdots \\ \partial h_m \partial x_1 & \dots & \partial h_m \partial x_n \end{bmatrix}, \text{ or shorter } J_{ij} = \partial h_i \partial x_j$$

Note that sometimes the Jacobian is defined as the transpose of the above, but we will stick to the above convention as it is slightly more convenient. As a special case, if $m=1$, i.e. h is a scalar function, the Jacobian reduces to a row-vector, which is equivalent to the transpose of the gradient:

$$\nabla_x h = [\partial h \partial x_1, \dots, \partial h \partial x_n]^T$$

Note that the term *gradient* (denoted by ∇) has a strict mathematical definition (it is the generalization of the derivative to scalar functions that take vectors as inputs). In Deep Learning-

speech the term is used more losely to generally refer to scalar-, vector-, matrix-, or tensor-valued derivatives. With this in mind, we can re-write the above multi-variate chain rule:

$$\partial z \partial x = \partial z \partial y \partial y \partial x = J_y(z) \cdot J_x(y) = (\nabla_x z)^T \cdot J_x(y)$$

In this definition of the chain rule, we first wrote the Jacobian as $\partial y / \partial x$. In other words, in this expression we derive a vector y by another vector x which results in a matrix $J_x(y)$. We can also think of this as computing the gradient for all combinations of elements from y and x and then storing them in a result matrix (i.e., the Jacobian). It is important to understand that matrix and vector derivatives are usually just element-wise derivatives that are specially layed out in the output. It is thus often a good idea to think about derivatives element-wise and to anticipate the resulting shape of the gradient before actually deriving it. Let's test this!

Gradients - Output Layer

In the backward pass we want to update the trainable weights of the network, so we are interested in the quantity:

$$\nabla L(W) = \partial L \partial W$$

Here, W stands for all weights of the network, i.e. $\{W_1, W_2\}$, so we need to derive the loss with respect to all weights in the network. Let's start with the derivation of $\partial L / \partial W_2$. Recall that we defined

$$L = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} \|\hat{y}_i - y_i\|_2^2$$

and

$$\hat{y} = \sigma(XW_1)W_2 = \sigma(H)W_2 = AW_2$$

where $\hat{y} = [\hat{y}_1, ..., \hat{y}_n]^T$.

Now, let's determine $\partial L / \partial W_2$. As L only depends implicitly through $\hat{y}$ on W_2 we need to use the chain rule:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2}$$

Because $W_2 \in \mathbb{R}^{3 \times 1}$ is technically a vector, we can directly use the "Jacobian-version" of the chain rule to find these quantities.

$$\frac{\partial L}{\partial \hat{y}} = (\nabla_{\hat{y}} L)^T = \left[\frac{\partial L}{\partial \hat{y}_1}, ..., \frac{\partial L}{\partial \hat{y}_N}\right]$$

Finding the value for each element in this gradient is straight-forward:

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{1}{N}(\hat{y}_i - y_i)$$

This leaves us with $\partial \hat{y} / \partial W_2$, which is the Jacobian and should have shape 4×3. Let's look at this element-wise, i.e. let's first compute

$$J_{W_2}(\hat{y})_{ij} = \left(\frac{\partial \hat{y}}{\partial W_2}\right)_{ij} = \frac{\partial \hat{y}_i}{\partial w_{2,j}}$$

In other words, we compute the gradient of sample i w.r.t. a single weight $w_{2,j}$. So let's first recall how we compute $\hat{y}_i$:

$$\hat{y}_i = \sigma(x_i^T W_1)W_2 =: a_i^T W_2 = \sum_{j=1}^{3} a_{ij} w_{2,j}$$

Because $\sigma(x_{Ti}W_1)$ does not depend on W_2 we treat it as a constant named a_{Ti}. Having expanded the expression like this, it should now be easy to see that

$$\partial \hat{y}_i / \partial w_{2,j} = a_{ij} \Rightarrow \partial \hat{y} / \partial W_2 = A$$

Isn't that nice? The Jacobian we were looking for is simply equal to the activated units from the lower layer that we computed in the forward pass. Finally, we can now write the entire expression:

$$\partial L / \partial W_2 = \frac{1}{N}[(\hat{y}_1 - y_1), \ldots, (\hat{y}_N - y_N)] \cdot A$$

Gradients - Hidden Layer

Next, let's turn to determining $\partial L / \partial W_1$. Again, we need the chain rule:

$$\partial L / \partial W_1 = \partial L / \partial \hat{y} \, \partial \hat{y} / \partial W_1 = \partial L / \partial \hat{y} \, \partial \hat{y} / \partial A \, \partial A / \partial H \, \partial H / \partial W_1$$

At first, this might look intimidating: Although the final gradient is a 2×3 matrix, we need to determine quantities such as $\partial A / \partial H$ which would be a 4-dimensional tensor. This is possible to do, but we can also just look at each derivative element-wise. In other words, let's find the derivative for the weight at index (i,j), i.e. $\partial L / \partial w_{ij}$. Note that for clarity we will refer to W_1 simply as W in the following. Now the gradient "simplifies" to:

$$\partial L / \partial w_{ij} = \partial L / \partial \hat{y} \, \partial \hat{y} / \partial w_{ij} = \partial L / \partial \hat{y} \, \partial \hat{y} / \partial A \, \partial A / \partial H \, \partial H / \partial w_{ij}$$

This is hardly a simplification, but we can just keep looking at element-wise derivatives. Specifically, the gradient $\partial \hat{y} / \partial w_{ij}$ is a row-vector whose k-th element is equal to $\partial \hat{y}_k / \partial w_{ij}$. In other words, we can look at each sample k individually. Recall that

$$\hat{y}_k = \sigma(x_{Tk} W_1) W_2 = a_T W_2$$

To compute the gradient of $\hat{y}_k$ w.r.t. w_{ij} we now write

$$\partial \hat{y}_k / \partial w_{ij} = \partial \hat{y}_k / \partial a \, \partial a / \partial h \, \partial h / \partial w_{ij}$$

Now it actually did simplify! The three gradients on the right hand side we should know how to compute. Let's take a look at it. Note that a and h are technically dependent on sample k, so they should be indexed as a_k and h_k. For simplicity we drop this index in the following and add it back in the end.

In the first gradient, we can treat a as constant. To compute it, let's look at it element-wise again, i.e. let's determine $\partial \hat{y}_k / \partial a_i$. For this we rewrite

$$\hat{y}_k = a_T W_2 = \sum_i a_i w_{2,i}$$

So, if we take the gradient of $\hat{y}_k$ w.r.t. a_i we simply get $w_{2,i}$. It follows that

$$\partial \hat{y}_k / \partial a = [w_{2,1}, \ldots w_{2,H}] = W_{T2}$$

Where, H is the size of the hidden layer (in our case it's 3). Next up is $\partial a / \partial h$. From the previous section we know that this expression is actually the Jacobian.

$$(\partial a / \partial h)_{ij} = \partial a_i / \partial h_j = \partial \sigma(h_i) / \partial h_j$$

The derivative of the sigmoid will always be zero unless $i=j$ in which case it is just $\sigma'(h_i)$. We can write this as a diagonal matrix.

$$\partial a \partial h = \text{diag}[\sigma'(h_1), ..., \sigma'(h_H)]$$

Finally, let's turn to $\partial h / \partial w_{ij}$. Again, we look at single element l of h which is computed as follows:

$$h_l = (x_{Tk} W)_l = \sum_q x_{k,q} w_{ql}$$

So, the derivative $\partial h_l / \partial w_{ij}$ is exactly zero unless $l=j$ in which case it is equal to $x_{k,i}$. In other words, the entire gradient is a vector full of zeros, except for the j-th element whose value is $x_{k,i}$. We write this down as:

$$\partial h \partial w_{ij} = [0, ..., x_{k,i}, ..., 0]_T.$$

Now we can put everything together:

$$\partial y\hat{}_k \partial w_{ij} = \partial y\hat{}_k \partial a \partial a \partial h \partial h \partial w_{ij} = W_{T2} \cdot \text{diag}[\sigma'(h_1), ..., \sigma'(h_H)] \cdot [0, ..., x_{k,i}, ..., 0]_T.$$

Because these matrices and vectors are quite sparse, it actually further simplifies to

$$\partial y\hat{}_k \partial w_{ij} = W_{T2} \cdot [0, ..., \sigma'(h_j) \cdot x_{k,i}, ..., 0]_T = w_{2,j} \cdot \sigma'(h_j) \cdot x_{k,i}$$

and thus

$$\partial y\hat{} \partial w_{ij} = [w_{2,j} \cdot \sigma'(h_j) \cdot x_{1,i}, ..., w_{2,j} \cdot \sigma'(h_j) \cdot x_{N,i}]_T$$

As we furthermore know $\partial L / \partial y\hat{}$ from the previous section, we can write:

$$\partial L \partial w_{ij} = \partial L \partial y\hat{} \partial y\hat{} \partial w_{ij} = 1N [(y\hat{}_1 - y_1), ..., (y\hat{}_N - y_N)] \cdot [w_{2,j} \cdot \sigma'(h_j) \cdot x_{1,i}, ..., w_{2,j} \cdot \sigma'(h_j) \cdot x_{N,i}]_T$$

---

which further simplifies to

$$\partial L \partial w_{ij} = 1N \sum_{kN} (y\hat{}_k - y_k) \cdot w_{2,j} \cdot \sigma'(h_{k,j}) \cdot x_{k,i}$$

Note that we re-introduced index k for the hidden vector h. We repeat this for all elements w_{ij} in W_1 to get the final gradient $\partial L / \partial W_1$

Implementation

Now we are ready to transfer the math into code.

[ ]

```
                X  y_hat  act_hidden :
    # Step 1: Calculate error
            ,       = mse(      ,  )

    # Step 2: calculate gradient wrt w2
      =          [0]
          = 1.0 /   *             # shape (4, 1)
            =               # shape (4, 3)
            =    matmul(          ,          )  # shape (1, 3)

    # According to the math, `dL_dw2` is a row-vector, however, `w2` is a column-vector.
```

```
    # To prevent erroneous numpy broadcasting during the gradient update, we must make
    # sure that `dL_dw2` is also a column-vector.
              =

    # Step 3: calculate gradient wrt w1
            = sigmoid_(                )
              =     zeros_like(     )
    for   in        (              [0]):
      for   in        (              [1]):
        # Note: using `residual[:, 0]` instead of just `residual` is important here, as otherwise
        # numpy broadcasting will make `s` a 4x4 matrix, which is wrong
          =           [:, 0] *     [ , 0] *        [:,  ] *   [:,  ]
                  [ ,  ] =     mean( )

    return          ,
```

This is a naive implementation of course - the double for loop to populate `dL_dw1` can slow things down drastically. It would be better if we could implement this using only matrix and vector operations, so let's try. The expression $w_{2,j} \cdot \sigma'(h_{k,j})$ is easy to vectorize - it is just an element-wise multiplication of the weight vector W_2 with the hidden representation h_k of sample k. We can do this for all samples N at the same time if we stack W_2 row-wise into a $N \times 3$ matrix. Then we can multiply it element-wise with the residual vector $(\hat{y} - y) \in R_{N \times 1}$ by broadcasting it again to a 4×3 matrix. Finally, the sum over the number of samples is a simple matrix product with the input matrix X. In code, this looks like this:

[ ]

```
                        X  y_hat  act_hidden  :
    # Step 1: Calculate error
              ,        = mse(       ,  )

    # Step 2: calculate gradient wrt w2
      =           [0]
          = 1.0 /    *              # shape (4, 1)
            =                  # shape (4, 3)
            =     matmul(           ,           )  # shape (1, 3)

    # According to the math, `dL_dw2` is a row-vector, however, `w2` is a column-vector.
    # To prevent erroneous numpy broadcasting during the gradient update, we must make
    # sure that `dL_dw2` is also a column-vector.
              =

    # Step 3: calculate gradient wrt w1
```

```
        = sigmoid_(              ) # shape (4, 3)asting by numpy
          = 1.0 /    *     matmul(     ,           ) # shape (2, 3)

    return          ,
```

We can remove the manual tiling by realising that we can achieve the same effect with another matrix muliplication. This further simplifies the calculation of the gradient to a one-liner:

[ ]

```
                        X  y_hat  act_hidden :
    # Step 1: Calculate error
              ,        = mse(       ,  )

    # Step 2: calculate gradient wrt w2
      =           [0]
          = 1.0 /    *             # shape (4, 1)
            =                 # shape (4, 3)
            =     matmul(           ,           ) # shape (1, 3)

    # According to the math, `dL_dw2` is a row-vector, however, `w2` is a column-vector.
    # To prevent erroneous numpy broadcasting during the gradient update, we must make
    # sure that `dL_dw2` is also a column-vector.
              =

    # Step 3: calculate gradient wrt w1
              = 1.0 /    *     matmul(     ,     matmul(           ,        ) * sigmoid_(              ))

    return          ,           ,
```

As you can see, the final computations to get the gradients look pretty innocent. However, to arrive here, quite some math and implementation tricks were involved. It is easy to see, that this will quickly become more complicated as soon as we add more and different types of layers (convolutional, recurrent, etc.).

Optimization

Finally we are ready to write the last part of the pipeline: the optimizer. Here, we update the weights of the network based on the gradients that we computed in the backward pass. For this tutorial, we use the simplest version of stochastic gradient descent (SGD), i.e. the gradient update for W given gradients $\nabla L(W) = \partial L \partial W$ is done in the following way:

$$W' = W - \lambda \cdot \nabla L(W)$$

where λ is the learning rate and W' are the updated weights. In other words, we just take a step into the direction of steepest descent, scaled by λ. During training, we loop over the entire training set several times. One iteration over the entire set is called an **epoch** and each epoch

might further divide the data set into several **batches**. Because our data set is quite small, we have only one batch, so there is actually not so much stochasticity involved in the training loop, but you get the idea.

[ ]

```
            = 10000
                = 0.1
                  = []

 # re-initialize the weights to be sure we start fresh
    ,      = initialize_weights()

for          in         (              ):

   # Step 1: forward pass
                ,          = forward_pass(  )

   # Step 2: backward pass
        ,       ,          = backward_pass(  ,          ,                  )

   # Step 3: apply gradients scaled by learning rate
       =      -                    *
       =      -                    *

   # Step 4: some book-keeping and print-out
   if              200 == 0:
           ('Epoch %d> Training error: %f'     (          ,          ))
                      append([          ,          ])

 # Plot training error progression over time
                    =      asarray(                    )
    plot(                     [:, 0],                    [:, 1]);
    xlabel('Epochs');
    ylabel('Training Error');
```

If everything is correct the error should be going down.

Predictions and Visualization

Now that we have a trained neural network let's try and see if it can actually correctly solve the XOR problem. Does the following look like what you would expect to see?

[ ]

```
   =      array([[0,0], [0,1], [1,0], [1,1]])
       = [              (forward_pass(  )[1]) for    in    ]
```

```
# Colors corresponding to class predictions y_hat.
        = ['green' if     == 1 else 'blue' for     in         ]

    =      figure()
    set_figwidth(6)
    set_figheight(6)
    scatter(   [:,0],   [:,1],  =200,  =          )
    xlabel('x1')
    ylabel('x2')
    show()
```

In the beginning we said that a linear model will not be able to solve the XOR problem. Hence, the decision boundary of our neural network must be non-linear. But how exactly does it look? Where does the network switch from labeling a point als "blue" to labeling it "green"? To visualize this we can sub-sample the space $[0,1]2$, get the predictions from the model for all these points and visualize them.

[ ]

```
              = 20
        ,          = 0.0, 0.0
        ,          = 1.0, 1.0
    ,     =      meshgrid(     linspace(         ,         ,                ),      linspace(         ,         ,                ))
                =      concatenate([    [  ,                ],     [  ,                ]],       =-1)
                =      reshape(                , [-1, 2])
          = [             (forward_pass(  )[1]) for    in                ]

# Colors corresponding to class predictions y_hat.
        = ['green' if     == 1 else 'blue' for     in         ]

    =      figure()
    set_figwidth(6)
    set_figheight(6)
    scatter(                [:,0],                [:,1],  =200,  =          )
    xlabel('x1')
    ylabel('x2')
    show()
```

Questions and Additional Exercises

Hopefully your network should now be able to solve the XOR problem succesfully. Here are a couple of questions to think about:

1. **No Activation**

In the beginning we mentioned that the XOR problem cannot be solved with a simple linear regression. Try removing the sigmoid function and see for yourself. Note: you will have to update the gradient computation, but you can just use the identity function as an activation function, whose gradient should be straight-forward to compute.

Implement a different activation function, such as ReLU or the hyperbolic tangent tanh and see how their choice affects training. Note: you will have to update the gradient computation.

2. **Number of Parameters**

   How many trainable parameters does our network contain?

3. **Hyper-Parameter Tuning**

   Successfull training of the neural network often depends on the choice of hyper-parameters. In our case, these are the learning rate λ, the total capacity of the network, and the number of epochs (one might also count the activation function and some other design decisions as a hyper-parameter, but we won't do that for this task). Play around with these parameters, i.e. choose high and low values for each of them and see how it affects training. Which parameter has the biggest effect?

4. **Dense Layer with Bias**

   In the lecture, dense layers had biases, i.e. $h=XW+b$. Change the `dense` function so that it can use biases. How does this affect the gradients?

5. **Decision Boundary**

   Play around with the visualization of the decision boundary, e.g. increase the resolution or visualize it for a larger space. Intuitively, what would be a "good" decision boundary? How does the choice of hyper-parameters affect the boundary?

6. **Additional Layer**

   Add an additional layer to the network. For this you have to update the forward pass, the gradient computation in the backward pass and the gradient update in the optimizer.

7. **Chain Rule**

   Verify that

   $$\partial z \partial x_i = \sum_j \partial z \partial y_j \partial y_j \partial x_i$$

   is indeed the same as

$$(\partial z \partial x)_i = ((\nabla_x z)^T \cdot J_x(y))_i$$

[ ]