rnn

July 30, 2024

1 Recurrent neural network (RNN) from scratch

2 What is a recurrent neural network?

A recurrent neural network (RNN), as opposed to a regular fully connected neural network (FCNN), has layers that are connected to themselves.

The difference might be clearer by first looking at an FCNN.

In an FCNN there are no connections between nodes in a single layer. For instance, h_1^1 is not connected to h_2^1. In addition, the input and output are always of a fixed length.

In an RNN, however, this is no longer the case. Nodes in the hidden layers are connected to themselves, represented by the curved lines in the figure below.

Thus the output $\vec{h}$ from the hidden layer is fed back into the hidden layer. This recurrence makes RNNs useful when working with sequential data, as we can have input of variable length. This is more clear if we unfold the recurrent part of the network.

3 The mathematics of RNNs

3.1 The RNN architecture

Consider some sequential input X with n features. Note that X here is an array with two axes, since it contains n features at each time step in the sequence. We will denote the input at a specific time step t as

$$\vec{X}^{(t)} = \begin{pmatrix} X_1^{(t)} \\ \vdots \\ X_n^{(t)} \end{pmatrix},$$

which is then an n-dimensional vector.

Next, consider an RNN with L hidden layers, and an output layer with m features. We will denote the output of the l'th hidden layer at time step t as

$$\vec{h}_l^{(t)} = \begin{pmatrix} h_{l,1}^{(t)} \\ \vdots \\ h_{l,n_l}^{(t)} \end{pmatrix},$$

with n_l being the number of features in the l'th hidden layer. The output of the RNN at time step t is denoted

$$\hat{\vec{y}}^{(t)} = \begin{pmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_m, \end{pmatrix}$$

where the hat is there to distinguish the RNN output $\hat{\vec{y}}^{(t)}$ from the target value, which is denoted $\vec{y}^{(t)}$. The RNN will then look like this.

3.2 Forward propagation

In order to propagate forward through the network we need some weights and biases to connect the nodes. To simplify the notation going forward, we will consider the input layer to be the *zeroth layer*, and the output layer to be the $L+1$*'th layer*. We need each node to propagate to the node at the next layer (keeping the time step constant), and the next time step (keeping the layer constant), except for the input and output layers which do not connect to each other (as illustrated in the diagram above).

Let $W^{l,l+1}$ be the weight matrix and $\vec{b}^{l,l+1}$ the bias vector, both connecting nodes at the l'th layer to the $l+1$'th layer, keeping the time step constant. Next, let W^{ll} be the weight matrix and $\vec{b}^{ll}$ the bias vector, both connecting nodes at subsequent time steps in the same layer. Also, let σ_l be the activation function in the l'th layer. Lastly, define the weighted sum $\vec{z}_l^{(t)}$ at layer l and time step t such that the output of the node is the activation of that weighted sum, that is, such that $\vec{h}_l^{(t)} = \sigma_l(\vec{z}_l^{(t)})$.

Using these definitions the output from the first hidden layer at the first time step is then

$$\vec{h}_1^{(1)} = \sigma_1\left(\vec{z}_1^{(1)}\right),$$

with

$$\vec{z}_1^{(1)} = W^{01}\vec{X}^{(1)} + \vec{b}^{01}.$$

At later time steps we will also need to consider the contribution from the previous time step. Hence for $t \geq 2$ we will define

$$\left(\vec{z}_1^{(t)}\right)_{\text{layer}} = W^{01}X^{(t)} + \vec{b}^{01}$$

$$\left(\vec{z}_1^{(t)}\right)_{\text{time}} = W^{11}\vec{h}_1^{(t-1)} + \vec{b}^{11},$$

such that $\left(\vec{z}_1^{(t)}\right)_{\text{layer}}$ is the contribution from the previous layer, and $\left(\vec{z}_1^{(t)}\right)_{\text{time}}$ is the contribution from the previous time step. We then have

$$\vec{z}_1^{(t)} = \left(\vec{z}_1^{(t)}\right)_{\text{layer}} + \left(\vec{z}_1^{(t)}\right)_{\text{time}},$$

and

$$\vec{h}_1^{(t)} = \sigma_1\left(\vec{z}_1^{(t)}\right).$$

The expression is exactly the same for any hidden node, but for $l \geq 2$ we substitute $\vec{X}^{(t)}$ with $\vec{h}_{l-1}^{(t)}$. Thus for the l'th layer and t'th time step we have

$$\left(\vec{z}_l^{(t)}\right)_{layer} = W^{l-1,l}\vec{h}_{l-1}^{(t)} + \vec{b}^{l-1,l}$$

and

$$\left(\vec{z}_l^{(t)}\right)_{time} = W^{ll}\vec{h}_l^{(t-1)} + \vec{b}^{ll},$$

that combine to give

$$\vec{z}_l^{(t)} = \left(\vec{z}_l^{(t)}\right)_{layer} + \left(\vec{z}_l^{(t)}\right)_{time},$$

which in turn results in

$$\vec{h}_l^{(t)} = \sigma_l\left(\vec{z}_l^{(t)}\right).$$

This is also valid at the first time step by setting $\left(\vec{z}_l^{(1)}\right)_{\text{time}} = 0$.

The expression for the output layer is exactly the same as above, but with $\left(\vec{z}_l^{(t)}\right)_{\text{time}} = 0$. Thus we have

$$\vec{z}_{L+1}^{(t)} = \left(\vec{z}_{L+1}^{(t)}\right)_{\text{layer}} = W^{L,L+1}\vec{h}_L^{(t)} + \vec{b}^{L,L+1}$$

and

$$\hat{\vec{y}}^{(t)} = \sigma_{L+1}\left(\vec{z}_{L+1}^{(t)}\right)$$

The equations given for the forward propagation can seem a bit messy, so it is nice to have a more visual aid of what is going on. Here is a diagram of the complete RNN including the weights and biases relating the different nodes.

And here is a weights and biases connected to a single arbitrary node. The green arrows represent input to the node, and the red arrows represent the output from the node.

And here is the connections resulting in $\vec{h}_l^{(t)}$ in more detail.

3.3 Backpropagation through time (BPTT)

Backpropagation in an RNN works by comparing the output of the network to some target output (just as in the regular neural network), and propagating backwards through both the layers and the *time sequence*. It is therefore commonly referred to as *backpropagation through time* (BPTT). We will now derive the necessary equations to perform BPTT.

We assume that we have propagated forward through the network, and have produced some output $\hat{\vec{y}}^{(t)}$. We want to compare this with some target output value $\vec{y}^{(t)}$, and will do so through a cost function $C\left(\hat{\vec{y}}, \vec{y}\right)$. We will denote the cost at a specific time step t by $C^{(t)} = C^{(t)}\left(\hat{\vec{y}}^{(t)}, \vec{y}^{(t)}\right)$, and the overall cost of the network as C.

From the cost function at each time step, we want to compute the gradient with respect to each weight and bias, that is, we want to compute

$$\frac{\partial C}{\partial W^{l_1 l_2}} \quad \text{and} \quad \frac{\partial C}{\partial \vec{b}^{l_1 l_2}}$$

We will do this one layer at a time, starting at the output layer, and propagating backwards through time in each layer. We assume that we know the gradient of the cost function with respect to the output $\frac{\partial C^{(t)}}{\partial \hat{\vec{y}}^{(t)}}$, and start by finding the gradient with respect to the output weights and biases $W^{L,L+1}$ and $\vec{b}^{L,L+1}$.

3.3.1 Backpropagation through the output layer

First, we want to find the gradient with respect to $\vec{z}_{L+1}^{(t)}$. The derivative of C with respect to some element $z_{L+1,i}^{(t)}$ of the weighted sum is given by

$$\begin{aligned}
\frac{\partial C}{\partial z_{L+1,i}^{(t)}} &= \frac{\partial C^{(t)}}{\partial z_{L+1,i}^{(t)}} \\
&= \sum_{j=1}^{m} \frac{\partial C^{(t)}}{\partial \hat{y}_j^{(t)}} \frac{\partial \hat{y}_j^{(t)}}{\partial z_{L+1,i}^{(t)}} \\
&= \sum_{j=1}^{m} \frac{\partial C^{(t)}}{\partial \hat{y}_j^{(t)}} \sigma'_{L+1}\left(z_{L+1,i}^{(t)}\right) \delta_{ij} \\
&= \frac{\partial C^{(t)}}{\partial \hat{y}_i^{(t)}} \sigma'_{L+1}\left(z_{L+1,i}^{(t)}\right)
\end{aligned}$$

where δ_{ij} is the Kronecker delta $\delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$, and σ'_{L+1} denotes the derivative of the activation function, which we will assume to be known. we can write this expression more compactly in vector form as

$$\frac{\partial C}{\partial \vec{z}_{L+1}^{(t)}} = \frac{\partial C^{(t)}}{\partial \hat{\vec{y}}^{(t)}} \odot \sigma'_{L+1}\left(\vec{z}_{L+1}^{(t)}\right),$$

where $\odot$ denotes the *Hadamard product*, an elementwise multiplication of two vectors/matrices of same size.

Note: Sometimes the derivatives are real numbers like $\frac{\partial C^{(t)}}{\partial z_{L+1,i}^{(t)}}$, sometimes they are vectors such as $\frac{\partial C^{(t)}}{\partial \vec{z}_{L+1}^{(t)}}$, and sometimes they are matrices. I have not included any explicit notation to explain when they are what, but will assume that this is understood implicitly. A general rule would be to look at whether the expression contains indices like $i, j, k, ...$ or not.

Another note: There are a lot of indices to keep track of, so to make the notation simpler to follow I will try to follow these rules consistently: - l = layer index (with L being the final hidden layer). If I need several layer indices I will use $l_1, l_2, ...$ - (t) = time step index. - i, j, k = vector/matrix elements. - n = number of input features (length of $\vec{X}$). - m = number of output features (length of $\hat{\vec{y}}$). - $n_1, n_2, ...$ = number of features in hidden layer number $1, 2, ...$

Third note: I will not always write the upper bound of summations explicitly, but will assume that this is understood implicitly. For instance, $\sum_j W_{ij}^{l-1,l} h_{l-1,j}$ should be understood to mean $\sum_{j=1}^{n_{l-1}} W_{ij}^{l-1,l} h_{l-1,j}$, such that it sums over all elements of $\vec{h}_{l-1}$.

The derivative with respect to the weighted sum will be used a lot during backpropagation, so we will give it its own notation

$$\vec{\delta}_{L+1}^{(t)} \equiv \frac{\partial C^{(t)}}{\partial \vec{z}_{L+1}^{(t)}} = \frac{\partial C^{(t)}}{\partial \hat{\vec{y}}^{(t)}} \odot \sigma'_{L+1}\left(\vec{z}_{L+1}^{(t)}\right).$$

$\delta_{L+1}^{(t)}$ has one index downstairs (denoting layer), and one index upstairs in parentheses (denoting time step), so don't mix it up with the Kronecker delta δ_{ij}, which I will consistently write with two indices downstairs.

From the delta we can find the cost gradient with respect to the output bias. Note that the same weights and biases occur several times in the RNN, so we have to sum over each contribution. The cost gradients with respect to the weights and biases in layer l are denoted $\frac{\partial C}{\partial W^{l-1,l}}$, $\frac{\partial C}{\partial W^{ll}}$, $\frac{\partial C}{\partial \vec{b}^{l-1,l}}$ and $\frac{\partial C}{\partial \vec{b}^{ll}}$, and we will denote the contribution at time step t as $\left(\frac{\partial C}{\partial W^{l-1,l}}\right)^{(t)}$, $\left(\frac{\partial C}{\partial W^{ll}}\right)^{(t)}$, $\left(\frac{\partial C}{\partial \vec{b}^{l-1,l}}\right)^{(t)}$ and $\left(\frac{\partial C}{\partial \vec{b}^{ll}}\right)^{(t)}$ such that $\frac{\partial C}{\partial W^{l-1,l}} = \sum_t \left(\frac{\partial C}{\partial W^{l-1,l}}\right)^{(t)}$ and so on. Using this notation, the gradient with respect to the output bias becomes

$$\begin{aligned}
\left(\frac{\partial C}{\partial b_i^{L,L+1}}\right)^{(t)} &= \sum_{j=1}^{m} \frac{\partial C}{\partial z_{L+1,j}^{(t)}} \frac{\partial z_{L+1,j}^{(t)}}{\partial b_i^{L,L+1}} \\
&= \sum_{j=1}^{m} \frac{\partial C}{\partial z_{L+1,j}^{(t)}} \frac{\partial}{\partial b_i^{L,L+1}} \left(\sum_k W_{jk}^{L,L+1} h_{L,k}^{(t)} + b_j^{L,L+1}\right) \\
&= \sum_{j=1}^{m} \frac{\partial C}{\partial z_{L+1,j}^{(t)}} \delta_{ij} \\
&= \frac{\partial C}{\partial z_{L+1,i}^{(t)}} \\
&= \delta_{L+1,i}^{(t)}.
\end{aligned}$$

Thus on vector form we have

$$\left(\frac{\partial C}{\partial \vec{b}^{L,L+1}}\right)^{(t)} = \vec{\delta}_{L+1}^{(t)},$$

and finally

$$\frac{\partial C}{\partial \vec{b}^{L,L+1}} = \sum_t \left(\frac{\partial C}{\partial \vec{b}^{L,L+1}}\right)^{(t)}$$

We can also compute the gradient with respect to the output weights

$$\left(\frac{\partial C}{W_{ij}^{L,L+1}}\right)^{(t)} = \sum_{k_1=1}^{m} \frac{\partial C}{\partial z_{L+1,k_1}^{(t)}} \frac{\partial z_{L+1,k_1}^{(t)}}{\partial W_{ij}^{L,L+1}}$$

$$= \sum_{k_1=1}^{m} \delta_{L+1,k_1}^{(t)} \frac{\partial}{\partial W_{ij}^{L,L+1}} \left(\sum_{k_2} W_{k_1k_2}^{L,L+1} h_{L,k_2}^{(t)} + b_{k_1}^{L,L+1}\right)$$

$$= \sum_{k_1=1}^{m} \delta_{L+1,k_1}^{(t)} \sum_{k_2} h_{L,k_2}^{(t)} \delta_{ik_1} \delta_{jk_2}$$

$$= \delta_{L+1,i}^{(t)} h_{L,j}^{(t)}$$

$$= \left[\vec{\delta}_{L+1}^{(t)} \left(\vec{h}_L^{(t)}\right)^T\right]_{ij}.$$

Thus on vector form we have

$$\left(\frac{\partial C}{W^{L,L+1}}\right)^{(t)} = \vec{\delta}_{L+1}^{(t)} \left(\vec{h}_L^{(t)}\right)^T,$$

and

$$\frac{\partial C}{W^{L,L+1}} = \sum_t \left(\frac{\partial C}{W^{L,L+1}}\right)^{(t)}.$$

Note that we here have an outer product between two vectors, which results in a matrix:

$$\vec{\delta}_{L+1}^{(t)} \left(\vec{h}_L^{(t)}\right)^T = \begin{pmatrix} \delta_{L+1,1}^{(t)} \\ \vdots \\ \delta_{L+1,m}^{(t)} \end{pmatrix} \begin{pmatrix} h_{L,1}^{(t)} & \cdots & h_{L,n_L}^{(t)} \end{pmatrix} = \begin{pmatrix} \delta_{L+1,1}^{(t)} h_{L,1}^{(t)} & \cdots & \delta_{L+1,1}^{(t)} h_{L,n_L}^{(t)} \\ \vdots & \ddots & \vdots \\ \delta_{L+1,m}^{(t)} h_{L,1}^{(t)} & \cdots & \delta_{L+1,m}^{(t)} h_{L,n_L}^{(t)} \end{pmatrix}$$

Lastly, we need to compute the gradient with respect to the output from the previous layer $\frac{\partial C}{\partial \vec{h}_L^{(t)}}$, in order to continue backpropagating through previous layers. We find this in much the same way as we found the other gradients above.

$$\frac{\partial C}{\partial h_{L,i}^{(t)}} = \sum_j \frac{\partial C}{z_{L+1,j}^{(t)}} \frac{\partial z_{L+1,j}^{(t)}}{\partial h_{L,i}^{(t)}}$$

$$= \sum_j \delta_{L+1,j}^{(t)} \frac{\partial}{\partial h_{L,i}^{(t)}} \left(\sum_k W_{jk}^{L,L+1} h_{L,k}^{(t)} + b_j^{L,L+1} \right)$$

$$= \sum_j \delta_{L+1,j}^{(t)} \sum_k W_{jk}^{L,L+1} \delta_{ik}$$

$$= \sum_j \delta_{L+1,j}^{(t)} W_{ji}^{L,L+1}$$

$$= \sum_j \left[\left(W^{L,L+1} \right)^T \right]_{ij} \delta_{L+1,j}^{(t)}$$

$$= \left[\left(W^{L,L+1} \right)^T \vec{\delta}_{L+1}^{(t)} \right]_i$$

And thus on vector form we have

$$\frac{\partial C}{\partial \vec{h}_L^{(t)}} = \left(W^{L,L+1} \right)^T \vec{\delta}_{L+1}^{(t)}$$

Here is a diagram showing the backpropagation through the output layer.

3.3.2 Backpropagation through arbitrary node

Consider some arbitrary node in the RNN with output $\vec{h}_l^{(t)}$. Assume you know the total gradient of the cost with respect to this output from the two suceeding nodes

$$\frac{\partial C}{\partial \vec{h}_l^{(t)}} = \left(\frac{\partial C}{\partial \vec{h}_l^{(t)}} \right)_{\text{layer}} + \left(\frac{\partial C}{\partial \vec{h}_l^{(t)}} \right)_{\text{time}}.$$

We now want to compute the gradients with respect to the weights and biases connecting the two previous nodes to this node, so that we can update these weights and biases when training the network, as well as the gradient with respect to the two previous nodes, so that we can continue backpropagation through the other nodes. The situation is illustrated in the diagram below. The blue arrows show the input gradient from the succeeding nodes, and the red arrows show the gradients we want to compute.

The necessary gradients are derived in the same way as for the output layer, so I will simply state the results here. We get the following set of equations for backpropagating through a general node

in the RNN.

$$\delta_l^{(t)} = \frac{\partial C}{\partial \vec{h}_l^{(t)}} \odot \sigma_l' \left(\vec{z}_l^{(t)} \right) \tag{1}$$

$$\left(\frac{\partial C}{\partial \vec{b}^{l-1,l}} \right)^{(t)} = \left(\frac{\partial C}{\partial \vec{b}^{ll}} \right)^{(t)} = \delta_l^{(t)} \tag{2}$$

$$\left(\frac{\partial C}{\partial W^{l-1,l}} \right)^{(t)} = \delta_l^{(t)} \left(\vec{h}_{l-1}^{(t)} \right)^T \tag{3}$$

$$\left(\frac{\partial C}{\partial W^{ll}} \right)^{(t)} = \delta_l^{(t)} \left(\vec{h}_l^{(t-1)} \right)^T \tag{4}$$

$$\frac{\partial C}{\partial \vec{h}_{l-1}^{(t)}} = \left[\left(W^{l-1,l} \right)^{(t)} \right]^T \delta_l^{(t)} \tag{5}$$

$$\frac{\partial C}{\partial \vec{h}_l^{(t-1)}} = \left[\left(W^{ll} \right)^{(t-1)} \right]^T \delta_l^{(t)}, \tag{6}$$

and

$$\frac{\partial C}{\partial \vec{b}^{l-1,l}} = \sum_t \left(\frac{\partial C}{\partial \vec{b}^{l-1,l}} \right)^{(t)} \tag{7}$$

$$\frac{\partial C}{\partial \vec{b}^{ll}} = \sum_t \left(\frac{\partial C}{\partial \vec{b}^{ll}} \right)^{(t)} \tag{8}$$

$$\frac{\partial C}{\partial W^{l-1,l}} = \sum_t \left(\frac{\partial C}{\partial W^{l-1,l}} \right)^{(t)} \tag{9}$$

$$\frac{\partial C}{\partial W^{ll}} = \sum_t \left(\frac{\partial C}{\partial W^{ll}} \right)^{(t)}. \tag{10}$$

With this method we can start with the nodes in the output layer, and propagate backwards. The necessary input to one node is the output from backpropagating through the previous node. Thus we can use the equations above recursively, layer by layer, to backpropagate through the entire network.

4 The RNN code

Now that we have the mathematical framework, we can develop the code for the RNN.

4.1 Functions

Before we start building a recurrent neural network, we need to define some functions. We need activation functions, cost functions and a way to differentiate these. We also need gradient descent schedulers to update our weights and biases when backpropagating. These functions are defined in this section.

4.1.1 Activation functions

We want to be able to choose which activation function to use in different layers of the RNN. Here we define some activation functions that can be used by the network. If you have developed a regular fully connected neural network in FYS-STK4155 these functions will probably look very familiar, as they are pretty much copied from those lecture notes. The main difference is that I have used JAX instead of autograd for automatic differentiation. Due to the way JAX vectorizes, I have not gotten gradients with jax to work for softmax, but have included grad_softmax() as its own function.

[ ]:
```
import numpy as np
import jax.numpy as jnp

def identity(X):
    return X


def sigmoid(X):
    try:
        return 1.0 / (1 + jnp.exp(-X))
    except FloatingPointError:
        return jnp.where(X > jnp.zeros(X.shape), jnp.ones(X.shape), jnp.zeros(X.
 ↪shape))


def softmax(X):
    X = X - np.max(X, axis=-1, keepdims=True)
    delta = 10e-10
    return np.exp(X) / (np.sum(np.exp(X), axis=-1, keepdims=True) + delta)


def grad_softmax(X):
    f = softmax(X)
    return f - f**2


def RELU(X):
    return jnp.where(X > jnp.zeros(X.shape), X, jnp.zeros(X.shape))


def LRELU(X):
```

```
    delta = 10e-4
    return jnp.where(X > jnp.zeros(X.shape), X, delta * X)


def tanh(X):
    return jnp.tanh(X)
```

4.1.2 Cost functions

Next, we need to implement the cost functions. We include three cost functions here. The ordinary least squares (OLS) is used for regression problems, and the logistic regression and cross-entropy are used for classification problems.

[ ]:
```
def CostOLS(target):

    def func(X):
        return (1.0 / target.shape[0]) * jnp.sum((target - X) ** 2)

    return func


def CostLogReg(target):

    def func(X):

        return -(1.0 / target.shape[0]) * jnp.sum(
            (target * jnp.log(X + 10e-10)) + ((1 - target) * jnp.log(1 - X +␣
 ↪10e-10))
        )

    return func


def CostCrossEntropy(target):

    def func(X):
        return -(1.0 / target.size) * jnp.sum(target * jnp.log(X + 10e-10))

    return func
```

4.1.3 Automatic differentiation

As mentioned above, we use JAX for automatic differentiation, which is done with the function *grad* in the JAX library. For *grad* to work on a function, it cannot use regular numpy, but must use *jax.numpy*, which is why we imported and used this when defining the activation and cost functions. JAX's numpy is only used for these functions, while we stick with regular numpy for everything else.

The RELU and leaky RELU activation functions are not continuously differentiable, so we will handle these explicitly in our code.

[ ]:
```
from jax import grad

def derivate(func):
    if func.__name__ == "RELU":

        def func(X):
            return jnp.where(X > 0, 1, 0)

        return func

    elif func.__name__ == "LRELU":

        def func(X):
            delta = 10e-4
            return jnp.where(X > 0, 1, delta)

        return func

    else:
        return grad(func)
```

Note that the *grad* function is not, in itself, vectorized. This means that if we send in an array to a function that has been differentiated, JAX will treat this as a function with an array as input, not as a function treating each element individually. This is better understood with an example.

Consider the function $f(x) = x^2$, with derivative $f'(x) = 2x$. With JAX we get this with

[ ]:
```
def f(x):
    return x**2

df = grad(f)
```

This works if we input a scalar value

[ ]:
```
x = 2.0
print(df(x))
```

```
4.0
```

But if we try to input an array of values we get an error message.

[ ]:
```
x = np.linspace(0, 3, 10)
print(df(x))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 2
```

```
      1 x = np.linspace(0, 3, 10)
----> 2 print(df(x))

    [… skipping hidden 4 frame]

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/jax/_src/api.py:715, in _check_scalar(x)
    713 if isinstance(aval, ShapedArray):
    714   if aval.shape != ():
--> 715     raise TypeError(msg(f"had shape: {aval.shape}"))
    716 else:
    717   raise TypeError(msg(f"had abstract value {aval}"))

TypeError: Gradient only defined for scalar-output functions. Output had shape:
 ↪(10,).
```

This is because JAX does not treat each element in x individually. To get around this we can use the *vmap* function in the JAX library, which vectorizes the function.

[ ]:
```
from jax import vmap
df = vmap(grad(f))
x = np.linspace(0, 3, 10)
print(df(x))
```

```
[0.        0.6666667 1.3333334 2.        2.6666667 3.3333333 4.
 4.6666665 5.3333335 6.       ]
```

Note that *vmap* only vectorizes along one dimension. So if the input array contains several axes, we have to apply *vmap* for each axis.

[ ]:
```
df = vmap(vmap(vmap(grad(f))))
x = np.linspace(0,3, 27).reshape((3,3,3))
print(df(x))
```

```
[[[0.         0.23076923 0.46153846]
  [0.6923077  0.9230769  1.1538461 ]
  [1.3846154  1.6153846  1.8461539 ]]

 [[2.0769231  2.3076923  2.5384614 ]
  [2.7692308  3.         3.2307692 ]
  [3.4615386  3.6923077  3.9230769 ]]

 [[4.1538463  4.3846154  4.6153846 ]
  [4.8461537  5.076923   5.3076925 ]
  [5.5384617  5.769231   6.        ]]]
```

This approach is tedious, yes, but as far as I know it is the only way to make JAX differentiate elementwise. If you find a simpler work-around, feel free to share with the professor or the group teachers so these notes can be updated.

4.1.4 Schedulers

We also want to be able to choose which method we want to use for gradient descent when training the RNN. This is done by using a scheduler defined below. These schedulers are identical to the ones used in FYS-STK4155.

[ ]:
```python
class Scheduler:
    """
    Abstract class for Schedulers
    """

    def __init__(self, eta):
        self.eta = eta

    # should be overwritten
    def update_change(self, gradient):
        raise NotImplementedError

    # overwritten if needed
    def reset(self):
        pass


class Constant(Scheduler):
    def __init__(self, eta):
        super().__init__(eta)

    def update_change(self, gradient):
        return self.eta * gradient

    def reset(self):
        pass


class Momentum(Scheduler):
    def __init__(self, eta: float, momentum: float):
        super().__init__(eta)
        self.momentum = momentum
        self.change = 0

    def update_change(self, gradient):
        self.change = self.momentum * self.change + self.eta * gradient
        return self.change

    def reset(self):
        pass
```

```
class Adagrad(Scheduler):
    def __init__(self, eta):
        super().__init__(eta)
        self.G_t = None

    def update_change(self, gradient):
        delta = 1e-8  # avoid division ny zero

        if self.G_t is None:
            self.G_t = np.zeros((gradient.shape[0], gradient.shape[0]))

        self.G_t += gradient @ gradient.T

        G_t_inverse = 1 / (
            delta + np.sqrt(np.reshape(np.diagonal(self.G_t), (self.G_t.
 ↪shape[0], 1)))
        )
        return self.eta * gradient * G_t_inverse

    def reset(self):
        self.G_t = None


class AdagradMomentum(Scheduler):
    def __init__(self, eta, momentum):
        super().__init__(eta)
        self.G_t = None
        self.momentum = momentum
        self.change = 0

    def update_change(self, gradient):
        delta = 1e-8  # avoid division ny zero

        if self.G_t is None:
            self.G_t = np.zeros((gradient.shape[0], gradient.shape[0]))

        self.G_t += gradient @ gradient.T

        G_t_inverse = 1 / (
            delta + np.sqrt(np.reshape(np.diagonal(self.G_t), (self.G_t.
 ↪shape[0], 1)))
        )
        self.change = self.change * self.momentum + self.eta * gradient *␣
 ↪G_t_inverse
        return self.change

    def reset(self):
```

```
        self.G_t = None


class RMS_prop(Scheduler):
    def __init__(self, eta, rho):
        super().__init__(eta)
        self.rho = rho
        self.second = 0.0


    def update_change(self, gradient):
        delta = 1e-8  # avoid division ny zero
        self.second = self.rho * self.second + (1 - self.rho) * gradient *␣
 ↪gradient
        return self.eta * gradient / (np.sqrt(self.second + delta))

    def reset(self):
        self.second = 0.0


class Adam(Scheduler):
    def __init__(self, eta, rho, rho2):
        super().__init__(eta)
        self.rho = rho
        self.rho2 = rho2
        self.moment = 0
        self.second = 0
        self.n_epochs = 1

    def update_change(self, gradient):
        delta = 1e-8  # avoid division ny zero
        self.moment = self.rho * self.moment + (1 - self.rho) * gradient
        self.second = self.rho2 * self.second + (1 - self.rho2) * gradient *␣
 ↪gradient

        moment_corrected = self.moment / (1 - self.rho**self.n_epochs)
        second_corrected = self.second / (1 - self.rho2**self.n_epochs)

        return self.eta * moment_corrected / (np.sqrt(second_corrected + delta))

    def reset(self):
        self.n_epochs += 1
        self.moment = 0
        self.second = 0
```

4.2 The RNN

We will now implement the code for the RNN. The network will be object-oriented, consisting of the following classes:

- *RNN*: The complete network, consisting of several *Layer* objects.
- *Layer*: Abstract class containing information that is shared across the different types of layers. It is the parent class of the following:
  - *InputLayer*: Layer containing the input to the network. Does not contain any weights and biases.
  - *RNNLayer*: The recurrent layer consisting of nodes in sequence.
  - *OutputLayer*: Recurrent layer for output. Similar to RNNLayer, but does not have any connections between the nodes, only to the nodes at the same time step in the previous layer.
  - *DenseLayer*: Fully connected layer, used to switch from a recurrent network to a regular fully connected network. Especially used for non-sequential output, for instance in classification where you want to classify the entire sequence with a single output.
- *Node*: Contains information about a single node. This is where all the math of forward- and backpropagation takes place.

Since RNN class uses the Layer classes, and the Layer classes use the Node class, we will build this from the down up, starting with the Node class.

4.2.1 The Node class

The Node class takes care of all the math discussed in

$$\delta_l^{(t)} = \frac{\partial C}{\partial \vec{h}_l^{(t)}} \odot \sigma_l' \left(\vec{z}_l^{(t)} \right) \tag{11}$$

$$\left(\frac{\partial C}{\partial \vec{b}^{l-1,l}} \right)^{(t)} = \left(\frac{\partial C}{\partial \vec{b}^{ll}} \right)^{(t)} = \delta_l^{(t)} \tag{12}$$

$$\left(\frac{\partial C}{\partial W^{l-1,l}} \right)^{(t)} = \delta_l^{(t)} \left(\vec{h}_{l-1}^{(t)} \right)^T \tag{13}$$

$$\left(\frac{\partial C}{\partial W^{ll}} \right)^{(t)} = \delta_l^{(t)} \left(\vec{h}_l^{(t-1)} \right)^T \tag{14}$$

$$\frac{\partial C}{\partial \vec{h}_{l-1}^{(t)}} = \left[\left(W^{l-1,l} \right)^{(t)} \right]^T \delta_l^{(t)} \tag{15}$$

$$\frac{\partial C}{\partial \vec{h}_l^{(t-1)}} = \left[\left(W^{ll} \right)^{(t-1)} \right]^T \delta_l^{(t)}, \tag{16}$$

[ ]:
```python
from collections.abc import Callable # Used for type hints of functions

class Node:
    def __init__(
            self,
            n_features: int,
            act_func: Callable[[np.ndarray], np.ndarray] = identity,
            W_layer: np.ndarray = None,
            b_layer: np.ndarray = None,
            W_time: np.ndarray = None,
            b_time: np.ndarray = None
    ):
        """
        n_features = number of features for this node
        W = weights, b = bias
        _layer = from previous layer to this one
        _time = from previous time step to this one
        output = output from feed_forward through this node
        """
        self.n_features = n_features
        self.act_func = act_func
        self.W_layer = W_layer
        self.b_layer = b_layer
        self.W_time = W_time
        self.b_time = b_time

        ## Values from feed_forward()
        self.h_layer = None # h from previous layer
        self.h_time = None # h from previous time step
        self.z_output = None # z for this node
        self.h_output = None # h from this node

        ## Values from backpropagate()
        self.grad_b_layer = None
        self.grad_b_time = None
        self.grad_W_layer = None
        self.grad_W_time = None
        self.grad_h_layer = None
        self.grad_h_time = None

    def set_Wb(
            self,
            W_layer: np.ndarray,
            b_layer: np.ndarray,
            W_time: np.ndarray,
            b_time: np.ndarray
    ):
```

```
        self.W_layer = W_layer
        self.b_layer = b_layer
        self.W_time = W_time
        self.b_time = b_time

    def get_output(self):
        return self.h_output

    def set_output(self, output: np.ndarray):
        self.h_output = output

    def feed_forward(
            self,
            h_layer: np.ndarray,
            h_time: np.ndarray = None
    ):
        """
        h_layer/h_time: Output from node at previous layer/time
        h_shape = (n_batches, n_features)
        """
        ## Save h_layer and h_time for use in backpropagation
        self.h_layer = h_layer
        self.h_time = h_time

        num_inputs = h_layer.shape[0]

        ## Compute weighted sum z for this node.
        z_layer = h_layer @ self.W_layer + self.b_layer # Dimension example:␣
↪(100,5)@(5,7) + (7) = (100,7)

        if h_time is None:
            # This node is at the first time step, thus not receiving any input␣
↪from previous time steps.
            z_time = np.zeros((num_inputs, self.n_features))
        else:
            z_time = h_time @ self.W_time + self.b_time

        self.z_output = z_layer + z_time # Save the weighted sum in the node

        ## Compute activation of the node
        h_output = self.act_func(self.z_output)

        self.h_output = h_output # Save the output in the node
        return h_output # Return output


    def backpropagate(
```

```
            self,
            dC_layer: np.ndarray = None,
            dC_time: np.ndarray = None,
            lmbd: float = 0.01
    ):
        """
        dC_layer/dC_time = Contribution of cost gradient w.r.t. this node from␣
↪node at next layer/time
        dC_shape = (n_batches, n_features), i.e., the same as h_shape in␣
↪feed_forward
        """
        n_batches = self.h_output.shape[0]

        ## Total gradient is the sum of the gradient from "next" layer and time
        if dC_time is None:
            # If this is the last node in the layer, the gradient is just the␣
↪gradient from the next layer
            dC = dC_layer
        elif dC_layer is None:
            # If the next layer has no node at this time step (because it is a␣
↪SingleOutputLayer), use only dC_time
            dC = dC_time
        else:
            dC = dC_layer + dC_time

        ## delta (gradient of cost w.r.t. z)
        if self.act_func.__name__ == "softmax":
            grad_act = grad_softmax(self.z_output)
        else:
            grad_act = vmap(vmap(derivate(self.act_func)))(self.z_output) #␣
↪vmap is necessary for jax to vectorize gradient properly

        delta = grad_act * dC # Hadamard product, i.e., elementwise␣
↪multiplication

        ## Gradients w.r.t. bias
        self.grad_b_layer = np.sum(delta, axis=0) / n_batches
        self.grad_b_time = np.sum(delta, axis=0) / n_batches

        ## Gradients w.r.t. weights
        # Need to transpose h and not delta in order for matrices to match up␣
↪correctly, since we have batches along rows, and features along columns
        self.grad_W_layer = self.h_layer.T @ delta / n_batches
        self.grad_W_layer = self.grad_W_layer + self.W_layer * lmbd #␣
↪Regularization factor
```

```
        if self.h_time is None:
            self.grad_W_time = None
        else:
            self.grad_W_time = self.h_time.T @ delta / n_batches
            self.grad_W_time = self.grad_W_time + self.W_time * lmbd #␣
↪Regularization factor

        ## Gradients w.r.t. input from previous nodes
        # Need to not transpose delta in order for matrices to match up␣
↪correctly, since we have batches along rows, and features along columns
        self.grad_h_layer = delta @ self.W_layer.T
        if self.h_time is None:
            self.grad_h_time = None
        else:
            self.grad_h_time = delta @ self.W_time.T
```

Check out this link.