Recurrent Neural Networks

Rohit Amarnath

February 24, 2024

1 RNNs

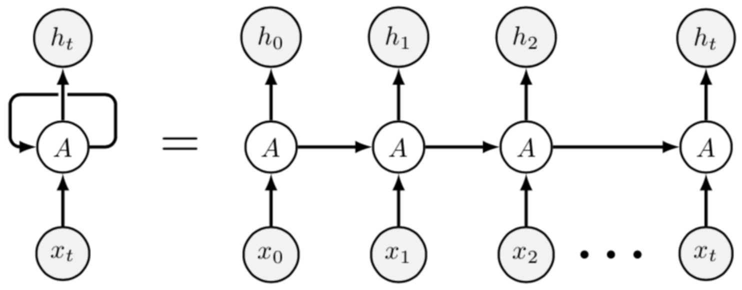

Figure 1: Unrolled and Rolled RNN

RNNs are networks that are designed to handle sequential or time-series data. At each time step, the model takes in input x_t and uses a hidden state h_{t-1} from the previous time step to generate an output and new hidden state for the current time step.

At a high level, RNNs aim to estimate the function

$$P(X_t|X_1, \dots X_{t-1})$$

1.1 Equations for a classic RNN

The classic RNN has two sets of weights: one for the previous hidden state and one for the current input. As mentioned earlier, at every time step, we generate an output and a hidden state for each output.

The equations for a vanilla RNN are as follows:

$$\begin{aligned} h_t &= f_h(x_t, h_{t-1}) = \tanh(W_h h_{t-1} + U_x x_t + b) \\ o_t &= f_o(h_t) = W_{yh} h_t + b_y \\ \hat{y_t} &= \text{softmax}(o_t) \end{aligned}$$

Note that the activation function is tanh. In practice, PyTorch and Tensorflow libraries implement only the first equation in their prototypical RNN model (it will be clear why this is in later sections). However, we generally reference these three equations together when talking about a recurrent network.

1.2 Backpropagation through time

The loss function for the RNN is

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T} \mathcal{L}_t(\hat{y_t}, y_t)$$

Now, differentiate with respect to W_{yh}, b_y, and W_h.

1.2.1 W_{yh}

1.2.2 b_y

1.2.3 W_h

$$\begin{aligned}\frac{\partial L}{\partial W_h} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial L_t(y_t,\hat{y_t})}{\partial W_h}\\ &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial L_t(y_t,\hat{y_t})}{\partial \hat{y_t}}\frac{\hat{y_t}}{\partial h_t}\frac{\partial h_t}{\partial W_h}\end{aligned}$$

Consider the transition from time step t to $t+1$. This is a little hard to understand at first, but if you visualize the computational graph, then there would be edges directed into h_{t+1} from h_i for $1 \le i \le t$. In other words, the unrolling of the RNN leads to connections between $t+1$ and every time step before it. This is why it's called back-propagation *through time.*

Therefore, we need to sum the partial derivatives with respect to each of h_i to compute the partial derivative of $\mathcal{L}_{t+1}$ with respect to W_h.

$$\frac{\partial \mathcal{L}_{t+1}}{\partial W_h} = \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}}\frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}}\sum_{i=1}^{t+1}\frac{\partial h_{t+1}}{\partial h_k}\frac{\partial h_k}{\partial W_h}$$

Since

$$\frac{\partial h_{t+1}}{\partial h_k} = \prod_{j=k}^{t}\frac{\partial h_{j+1}}{\partial h_j}$$

we have

$$\frac{\partial \mathcal{L}_{t+1}}{\partial W_h} = \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}}\frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}}\sum_{i=1}^{t+1}\left(\prod_{j=k}^{t}\frac{\partial h_{j+1}}{\partial h_j}\right)\frac{\partial h_k}{\partial W_h}$$

Summing across all L_t, we get

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T}\frac{\partial \mathcal{L}_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\sum_{i=1}^{t}\left(\prod_{j=k}^{t}\frac{\partial h_{j+1}}{\partial h_j}\right)\frac{\partial h_k}{\partial W_h}$$

1.2.4 U_x

There is no difference in computing the partial derivative with respect to U_x and W_h. Hence, we have a similar expression (as above):

$$\frac{\partial L}{\partial U_x} = \sum_{t=1}^{T}\frac{\partial \mathcal{L}_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\sum_{i=1}^{t}\left(\prod_{j=k}^{t}\frac{\partial h_{j+1}}{\partial h_j}\right)\frac{\partial h_k}{\partial U_x}$$

1.3 Vanishing and Exploding gradient

The following definitions and theorems are useful before reading this section.

Definition 1.1. *The spectral radius of matrix A is*

$$\rho(A) = \max(|\lambda_1|,\ldots,|\lambda_n|)$$

where $\lambda_1,\ldots,\lambda_n$ are the eigenvalues of matrix A.

Theorem 1.1. $\rho(A) \leq \|A\|_2$

Definition 1.2. *For matrices A and B, a matrix norm is said to be submultiplicative if $\|AB\| \leq \|A\|\|B\|$.*

Theorem 1.2. *The L^2 norm is submultiplicative.*

Proof

This explanation follows [4]. Consider the following term of BPTT:

$$\prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j}$$

Since

$$h_{j+1} = f_h(X_{j+1}, h_j)$$

we know that

$$\begin{aligned}\frac{\partial h_{j+1}}{\partial h_j} &= W_h^T \operatorname{diag}(f_h'(X_{j+1}, h_j)) \\ &= W_h^T \operatorname{diag}(f_h'(X_{j+1}, h_j))\end{aligned}$$

Let $\operatorname{diag}(f_h'(h_j)) < \gamma$. Then, the vanishing gradient problem occurs when $\rho(W_h^T) < \frac{1}{\gamma}$. To see this, first note that

$$\left\|\frac{\partial h_{j+1}}{\partial h_j}\right\| \leq \|W_h^T\|\|\operatorname{diag}(f_h'(h_j))\| < \frac{1}{\gamma}\gamma < 1$$

Then, let $\eta \in \mathbb{R}$ such that $\forall k, \|\frac{\partial h_{j+1}}{\partial h_j} \leq \eta < 1$. Thus, it can be shown that

$$\prod_{j=k}^{t} \left\|\frac{\partial h_{j+1}}{\partial h_j}\right\| \leq \eta^{t-k+1}$$

Since $\eta < 1$, the above term goes to 0 exponentially fast, showing that the term *vanishes*. A similar proof can be done for exploding gradients. Instead of having $\frac{1}{\gamma}$ be the upper bound for λ_1, it would be the lower bound.

2 Possible solutions for vanishing and exploding gradient

Vanishing and exploding gradients are most often seen with RNNs; however, they manifest in any deep network. Below we mention a few possible solutions to mitigate the vanishing and exploding gradient.

2.1 Weight initialization

Before we get into the relationship between weight initialization and vanishing/exploding gradients, let's understand how weight initialization affects the training of a neural network. Weight initialization is critical for a neural network to achieve a low training loss. Consider logistic regression. The weight initialization used for this model is setting all of the weight vectors to 0. In fact, the specific weight initialization in logistic regression does not matter as the model's loss function is convex, meaning that any local minimum is a global minimum.

However, neural network models have loss functions that are neither convex nor concave (the explanation for this is the subject of a set of notes on gradient descent and optimization). Thus, it is paramount that we initialize our weights correctly such that we achieve optimal training loss. Notably, initializing the weights to all 0 in a neural network will not train the network at all.

To see this, consider a vanilla neural network with L layers. For the L-th layer (the final output of the network), let $z^{(L)}$ pre-activation layer output, $W^{(L)}$ to be the weights, $\mathcal{L}$ be the loss and $h^{(L)} = f(z^{(L)})$ be the activated layer output. Backpropagation on the last layer yields

$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \frac{\partial \mathcal{L}}{\partial h^{(L)}} \frac{\partial h^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial W^{(L)}}$$
$$W^{(L)} := W^{(L)} - \alpha \frac{\partial \mathcal{L}}{\partial W^{(L)}}$$

Observe that since $z^{(L)} = W^{(L)} h^{(L-1)}$, $\frac{\partial z^{(L)}}{\partial W^{(L)}} = h^{(L-1)}$. The vector $h^{(L-1)}$ depends on the choice of activation function f. There are two cases here that will change the gradient descent update.

1. $f(0) = 0$ (think ReLU or tanh). If the weights are initialized to 0, then $h^{(L-1)} = 0$ on the first iteration of backpropagation (for that matter, every $h^{(i)} = 0$). This means that $W^{(L)}$ will not be updated in the first iteration. Since no update happens on the first iteration, the same process repeats, and the model is not trained.

2. $f(0) \neq 0$ (think sigmoid). If the weights are initialized, then vector $h^{(L-1)}$ is the same value across each element, meaning that the gradient update would also be the same. While the model *would train*, each neuron in a layer would be learning the same, which would severely inhibit the model's learning capability.

The above example shows that it doesn't make sense to initialize the weights of a layer to the same value. This is referred to as *breaking the symmetry* within a neural network, and for this reason, initialization strategies that we will explore will always involve some type of randomness in order for each neuron of the network to learn distinct features. Generally, the distributions that we use to generate random weights will be either Uniform or Gaussian. It is unclear which distribution is better (or the situations where one distribution is better than the other); usually, we just experiment either distribution and see which yields better results.

2.1.1 Xavier Initialization

Let $h^{(1)}, \dots, h^{(L)}$ be layers of a neural network. The key idea of [2] is to maintain the same variance across each layer and layer update in forward propagation and backward propagation, respectively. The intuition for this is similar to Batch Normalization: models tend to train better when each individual layer of the network train homogeneously and there is small distributional variation between each layer.

We now go through the derivation of Xavier initialization. It is fairly long! Let $z^{(l)}, h^{(l)}$ and $W^{(l)}$ be defined as how they were earlier, and additionally, define $b^{(l)}$ to be the bias and $n^{(l)}$ to be the number of neurons in layer l. For layer l, we have

$$z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}$$
$$h^{(l)} = f(z^{(l)})$$

where f is some activation function. If we want to maintain the same variance across each layer, then $\text{Var}(h^{(i)}) = \text{Var}(h^{(j)})$ for any layers i, j. Assume that f is approximately linear around the origin (such as tanh). Then, $\text{Var}(h^{(l)}) = \text{Var}(z^{(l)})$. Now, consider the k-th element of $z^{(l)}$:

$$z_k^{(l)} = \sum_{i=1}^{n^{(l-1)}} W_{k,i}^{(l)} h_i^{(l-1)} + b_k^{(l)}$$

For simplicity, ignore the bias term. Then,

$$\text{Var}(z_k^{(l)}) = \text{Var}\left(\sum_{i=1}^{n^{(l-1)}} W_{k,i}^{(l)} h_i^{(l-1)} \right)$$

Now, we make a few more assumptions: weights are i.i.d, inputs to a layer are i.i.d and weights and inputs are mutually independent. Before we proceed further, here is an important theorem relating to summing and multiplying variances of random variables.

Theorem 2.1. *For random variables X and Y,*

$$\begin{aligned}\operatorname{Var}(X+Y) &= \operatorname{Var}(X)+\operatorname{Var}(Y)+\operatorname{Cov}(X,Y)\\ \operatorname{Var}(XY) &= \mathbb{E}[X]^2\operatorname{Var}(Y)+\operatorname{Var}(X)\mathbb{E}[Y]^2+\operatorname{Var}(X)\operatorname{Var}(Y)\end{aligned}$$

Note that if X and Y are independent, then $\operatorname{Var}(X+Y)=\operatorname{Var}(X)+\operatorname{Var}(Y)$. *Furthermore, when* $\mathbb{E}[X]=\mathbb{E}[Y]=0, \operatorname{Var}(XY)=\operatorname{Var}(X)\operatorname{Var}(Y)$.

From the above theorem, we now have

$$\operatorname{Var}(z_k^{(l)}) = \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}\left(W_{k,i}^{(l)}h_i^{(l-1)}\right)$$

Most weight initializations will have $\mathbb{E}[W_{k,i}^{(l)}]=0$, and we assume $\mathbb{E}[h_i^{(l-1)}]=0$ (this assumption is *strange* - it is relaxed in He initialization). Additionally, since weights and inputs of a layer are identically distributed, we can define $\operatorname{Var}(W_{k,i}^{(l)})=\operatorname{Var}(W^{(l)}), \operatorname{Var}(h_i^{(l-1)})=\operatorname{Var}(h^{(l-1)})$ and $\operatorname{Var}(z^{(l)})=\operatorname{Var}(z_k^{(l)})$. Then,

$$\begin{aligned}\operatorname{Var}(z^{(l)}) = \operatorname{Var}(z_k^{(l)}) &= \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}(W_{k,i}^{(l)}h_i^{(l-1)})\\ &= \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}(W_{k,i}^{(l)})\operatorname{Var}(h_i^{(l-1)})\\ &= n^{(l-1)}\operatorname{Var}(W^{(l)})\operatorname{Var}(h^{(l-1)})\end{aligned}$$

Now, since $\operatorname{Var}(z^{(l)})=\operatorname{Var}(h^{(l)})$, we have

$$\operatorname{Var}(h^{(l)}) = n^{(l-1)}\operatorname{Var}(W^{(l)})\operatorname{Var}(h^{(l-1)})$$

To maintain the same variance across each layer, we want the weight distribution to have $\operatorname{Var}(W^{(l)})=\frac{1}{n^{(l-1)}}$. Inductively, we have

$$\operatorname{Var}(h^{(l)}) = \operatorname{Var}(x)\prod_{l=1}^{l} n^{(l-1)}\operatorname{Var}(W^{(l)})$$

where x is the input to the model. Notice that setting $\operatorname{Var}(W^{(l)})=\frac{1}{n^{(l-1)}}$ makes it such that the variance of any layer is equal to the variance of the input.

This is one part of the derivation for Xavier initialization. The second part is to find the relationship between the variances of subsequent layer updates. The backpropagation updates are

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial W^{(l)}} &= \frac{\partial\mathcal{L}}{\partial h^{(l)}}\frac{\partial h^{(l)}}{\partial W^{(l)}}\\ \frac{\partial\mathcal{L}}{\partial h^{(l)}} &= \frac{\partial\mathcal{L}}{\partial h^{(l+1)}}\frac{\partial h^{(l+1)}}{\partial z^{(l+1)}}\frac{\partial z^{(l+1)}}{\partial h^{(l)}}\end{aligned}$$

The above updates indicate that we only need to find a closed expression for $\frac{\partial h^{(l)}}{\partial h^{(l)}}$ since the update with respect to the weights depends on the update with respect to the l-th layer. Note that

$$\frac{\partial h_i^{(l+1)}}{\partial z_i^{(l+1)}}\frac{\partial z_i^{(l+1)}}{\partial h_i^{(l)}} = \sum_{j=1}^{n^{(l)}} W_{j,i}f'(z^{(l+1)})_j = \sum_{j=1}^{n^{(l)}} W_{j,i}$$

since $f'(z^{(l+1)}) = 1$. Now, let's calculate the variance of the update with respect to $h^{(l)}$.

$$\begin{aligned}\operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l)}}\right) = \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l+1)}}\frac{\partial h^{(l+1)}}{\partial z^{(l+1)}}\frac{\partial z^{(l+1)}}{\partial h^{(l)}}\right) &= \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l+1)}}\right)\operatorname{Var}\left(\frac{\partial h^{(l+1)}}{\partial z^{(l+1)}}\frac{\partial z^{(l+1)}}{\partial h^{(l)}}\right)\\ &= \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l+1)}}\right)\operatorname{Var}\left(\sum_{j=1}^{n^{(l)}} W_{j,i}\right)\\ &= \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l+1)}}\right) n^{(l)}\operatorname{Var}(W^{(l)})\end{aligned}$$

Let L be denote the final layer of the network. Then, inductively, we have

$$\operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l)}}\right) = \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(L)}}\right)\prod_{i=l}^{L-1} n^{(i)}\operatorname{Var}(W^{(i)})$$

For the update with respect to $W^{(l)}$, note that

$$\frac{\partial h^{(l)}}{\partial W^{(l)}} = h^{(l-1)}$$

Now, we find the variance of the update with respect to the weights. We have

$$\begin{aligned}\operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial W^{(l)}}\right) &= \operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(l)}}\right)\operatorname{Var}\left(\frac{\partial h^{(l)}}{\partial W^{(l)}}\right)\\ &= \operatorname{Var}(x)\operatorname{Var}\left(\frac{\partial \mathcal{L}}{\partial h^{(L)}}\right)\prod_{i=l}^{L-1} n^{(i)}\operatorname{Var}(W^{(i)})\prod_{i=1}^{l-1} n^{(i-1)}\operatorname{Var}(W^{(i)})\end{aligned}$$

Notice that to maintain the variance across weight updates, we want both

$$\begin{aligned} n^{(i)}\operatorname{Var}(W^{(i)}) &= 1\\ n^{(i-1)}\operatorname{Var}(W^{(i)}) &= 1\end{aligned}$$

The second update aligns with the variance needed to maintain the same activations across layers. As a compromise since we need stable updates and activations, we opt for the following:

$$\operatorname{Var}(W^{(i)}) = \frac{2}{n^{(i)} + n^{(i-1)}}$$

If we consider a uniform distribution for the weight distribution, we have

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n^{(i-1)} + n^{(i)}}}, \frac{\sqrt{6}}{\sqrt{n^{(i-1)} + n^{(i)}}}\right]$$

Alternatively, a gaussian distribution for the weight distribution would be

$$W \sim \mathcal{N}\left(0, \frac{\sqrt{2}}{\sqrt{n^{(i-1)} + n^{(i)}}}\right)$$

Note that the indexing of i may be different in other places. In this calculation, we denote $n^{(i-1)}$ to be the *fan-in* of the i-th layer and $n^{(i)}$ to be the *fan-out* of the i-th layer.

2.1.2 He Initialization

He initialization takes inspiration from Xavier initialization to design a weight initialization scheme for ReLU functions. We show the derivation of He initialization below.

Start with a similar setup to Xavier initialization. For layer l, we have

$$\begin{aligned} z^{(l)} &= W^{(l)} h^{(l-1)} + b^{(l)} \\ h^{(l)} &= f(z^{(l)}) \end{aligned}$$

In Xavier initialization, we assumed that f was approximately linear at 0. In He initialization [3], we relax this assumption since it is not true for ReLU. The ReLU function $f(x) = \max(0, x)$ has a cusp at $x = 0$, and so it cannot be linear at $x = 0$. This additionally means that we cannot assume that the activated layer $h^{(l)}$ will have a mean of 0. This will make a slight difference in the derivation as we will see below.

Again, ignore the bias term. In practice, the bias terms in either initialization are set to 0 in the start. Then,

$$\operatorname{Var}(z_k^{(l)}) = \operatorname{Var}\left(\sum_{i=1}^{n^{(l-1)}} W_{k,i}^{(l)} h_i^{(l-1)}\right)$$

As before, we assume that weights are i.i.d, inputs to a layer are i.i.d and weights and inputs are mutually independent. By Theorem 2.1, this gives us

$$\begin{aligned} \operatorname{Var}(z_k^{(l)}) &= \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}\left(W_{k,i}^{(l)} h_i^{(l-1)}\right) \\ &= \sum_{i=1}^{n^{(l-1)}} \left[\mathbb{E}[W_{k,i}^{(l)}]^2 \operatorname{Var}(h^{(l-1)}) + \mathbb{E}[h_i^{(l-1)}]^2 \operatorname{Var}(W_{k,i}^{(l)}) + \operatorname{Var}(W_{k,i}^{(l)}) \operatorname{Var}(h_i^{(l-1)})\right] \end{aligned}$$

We still assume that $\mathbb{E}[W_{k,i}^{(l)}] = 0$, but we cannot assume that $\mathbb{E}[h_i^{(l-1)}] = 0$ as $h_i^{(l-1)}$ is a ReLU activated value.

$$\begin{aligned} \operatorname{Var}(z^{(l)}) = \operatorname{Var}(z_k^{(l)}) &= \sum_{i=1}^{n^{(l-1)}} \left[\mathbb{E}[W_{k,i}^{(l)}]^2 \operatorname{Var}(h^{(l-1)}) + \mathbb{E}[h_i^{(l-1)}]^2 \operatorname{Var}(W_{k,i}^{(l)}) + \operatorname{Var}(W_{k,i}^{(l)}) \operatorname{Var}(h_i^{(l-1)})\right] \\ &= \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}(W_{k,i}^{(l)}) \left(\mathbb{E}[h_i^{(l-1)}]^2 + \operatorname{Var}(h_i^{(l-1)})\right) \\ &= \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}(W_{k,i}^{(l)}) \left(\mathbb{E}[h_i^{(l-1)}]^2 + \mathbb{E}[(h_i^{(l-1)})^2] - \mathbb{E}[h_i^{(l-1)}]^2\right) \\ &= \sum_{i=1}^{n^{(l-1)}} \operatorname{Var}(W_{k,i}^{(l)}) \mathbb{E}[(h_i^{(l-1)})^2] \\ &= n^{(l-1)} \operatorname{Var}(W^{(l)}) \mathbb{E}[(h^{(l-1)})^2] \end{aligned}$$

Now, from definition of expected value, we can simplify $\mathbb{E}[(h^{(l-1)})^2)]$. We will assume that $W^{(l)}$ is a symmetric distribution around 0 and the bias term is 0. This also means that $z^{(l)}$ is a symmetric distribution around 0 with mean 0 (it's worth mentioning that if the bias term wasn't initialized to 0 that this would not be true).

$$\begin{aligned}
\mathbb{E}[(h^{(l-1)})^2] &= \int_{-\infty}^{\infty} (h^{(l-1)})^2 P((h^{(l-1)})^2) dh^{(l-1)} \\
&= \int_{-\infty}^{\infty} (\max(0, z^{(l-1)}))^2 P(\max(0, z^{(l-1)})) dz^{(l-1)} \\
&= \int_{0}^{\infty} (\max(0, z^{(l-1)}))^2 P(\max(0, z^{(l-1)})) dz^{(l-1)} \\
&= \int_{0}^{\infty} (z^{(l-1)})^2 P(z^{(l-1)}) dz^{(l-1)} \\
&= \frac{1}{2} \int_{-\infty}^{\infty} (z^{(l-1)})^2 P(z^{(l-1)}) dz^{(l-1)} \\
&= \frac{1}{2} \operatorname{Var}(z^{(l-1)})
\end{aligned}$$

where the second simplification comes from $h^{(l-1)} = \max(0, z^{(l-1)})$. Substitute this back to the expression from earlier. We have

$$\operatorname{Var}(z^{(l)}) = \frac{1}{2} n^{(l-1)} \operatorname{Var}(W^{(l)}) \operatorname{Var}(z^{(l-1)})$$

Inductively, for layer L,

$$\operatorname{Var}(z^{(L)}) = \operatorname{Var}(z^{(1)}) \prod_{l=2}^{L} \left(\frac{1}{2} n^{(l-1)} \operatorname{Var}(W^{(l)}) \right)$$

In order to prevent the variance of the L-th layer from exploding, we set

$$\operatorname{Var}(W^{(l)}) = \frac{2}{n^{(l-1)}}$$

This is He initialization. In practice, we can also set $\operatorname{Var}(W^{(l)}) = \frac{2}{n^{(l)}}$, which would preserve the variance of backward updates. Unfortunately, we will not be deriving this expression. PyTorch and Tensorflow libraries usually give the user the option of choosing the *fan-mode* and what type of weight distribution they would want.

He initialization only works with ReLU. In addition to devising this new initialization strategy, the paper also introduced Parametric ReLU (PReLU), which is an extension on ReLU and Leaky ReLU activation. The PReLU function is defined as

$$f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$$

If α is a small constant (i.e. 0.08), then PReLU becomes Leaky ReLU. Leaky ReLU was introduced to induce a small gradient update when negative values pass through a neuron. PReLU takes this a step further and learns the scalar gradient update. Note that α is unique to a neuron in a layer of the network. Furthermore, it is worth pointing out the change that PReLU brings to the $\operatorname{Var}(z^{(l)})$ calculation we did earlier. With PReLU, we now have

$$\operatorname{Var}(z^{(l)}) = \frac{1}{2}(1 + \alpha^2) n^{(l-1)} \operatorname{Var}(W^{(l)}) \operatorname{Var}(z^{(l-1)})$$

This means that

$$\operatorname{Var}(W^{(l)}) = \frac{2}{(1 + \alpha^2) n^{(l-1)}}$$

Indeed, PReLU can make the variance of a weight layer smaller and also exert slight control over the variance as the model learns over time (I am a little amused and puzzled myself at why PReLU works well).

2.1.3 Does this work with recurrent architectures?

Unfortunately, these initialization schemes only work on feedfoward neural network architectures and **does not work on** recurrent architectures. The reason we can't use these initialization schemes with RNNs is because recurrent architectures have a third dimension within the input space: the time step. This third dimension changes many of the assumptions and equations we used in the derivation of these initialization schemes. To my knowledge, there is no specific initialization scheme for recurrent architectures that mitigate gradient issues.

In a normal feedforward neural network, it is easy to compute the fan-in and fan-out of a layer, thereby making it easy to find the weight distribution of a layer. In a CNN, the fan-in is computed by multiplying the kernel width, kernel height and number of filters, and the fan-out is computed by multiplying the kernel width, kernel height, number of filters and the reciprocal of the area of the max-pool layer (if applicable).

2.2 Gradient clipping

Gradient clipping is a technique where backpropagated gradients are restricted in order to prevent exploding gradients. There are two types of gradient clipping:

1. Threshold clipping: Let $g = \frac{\partial \mathcal{L}}{\partial x}$. Then, if $\|g\|_2 \geq 1$, set $g = 1$.
2. Norm clipping: Let $g = \frac{\partial \mathcal{L}}{\partial x}$. Then, if $\|g\|_2 \geq$ threshold, set $g \leftarrow \frac{\text{threshold}}{\|g\|_2} g$. This is the version of gradient clipping that was introduced in [4].

3 Dropout for RNN

4 Long Short-Term Memory (LSTM)

The LSTM model is an extension upon the vanilla RNN, aimed at dealing with the vanishing gradient problem. Unfortunately, LSTMs only *partially* solve the problem; however, empirically they perform substantially better than RNNs. At the time of their introduction and even in contemporary times, the LSTM model is one of the greatest machine learning discoveries as it was one of the first attempts at solving the vanishing gradient problem.

The idea behind the LSTM model [5] is to create a system within the model to learn when to write, read, and forget. We refer to the write operation as *writing to the internal memory model* of the LSTM and reading as selectively scanning and consuming the input data for the information we need. Lastly, by learning to forget, the model isn't *confused* by having an overhead of information within it's internal memory model while analyzing the data. This structure is implemented through the input gate (writing to memory), output gate (reading selectively from data), and forget gate (forgetting unnecessary information in memory).

The equations for a LSTM are the following:

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c_t} &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c_t} \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

Here, f_t is the forget gate, i_t is the input gate, o_t is the output gate, $\tilde{c_t}$ is the candidate cell state, c_t is the cell state, and h_t is the hidden state (can be informally referred to as the *output* vector of the LSTM).

4.1 Gradient flow

We will not go over BPTT for LSTM as it is much longer than RNN. The cell state gradients are of particular importance in analyzing the differences between LSTMs and RNNs and to understand how LSTMs mitigate the vanishing gradient problem.

Note that

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t$$

Inductively, this means that

$$\frac{\partial c_j}{\partial c_i} = \prod_{t=i}^{j} f_t$$

In section 1.3, we saw that RNNs experienced gradient issues as there was a weight matrix term that either grew or decayed exponentially fast. In LSTMs, this term doesn't exist anymore, which reduces the probability of the vanishing gradient problem. This does imply that LSTMs still could experience the vanishing gradient issue; however, it is nowhere close to the degree that RNNs are affected by it. In earlier versions of LSTMs, the forget gate did not exist, and so the cell state gradient was a constant. For this reason, the cell state was initially referred to as a constant error carousel (CEC).

4.2 Peephole LSTM

Peephole LSTM is an extension on vanilla LSTM: this type of LSTM allows the read/write/forget operations to access the previous internal cell state of the LSTM.

The equations for Peephole LSTM are the following:

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + P_f c_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + P_i c_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + P_o c_{t-1} + b_o) \\
\tilde{c_t} &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c_t} \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

4.3 Gated Recurrent Unit (GRU)

GRUs are a significant extension upon the vanilla LSTM: GRUs remove the cell state and output gate of LSTMs.

The equations for a GRU are the following:

$$\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\tilde{h_t} &= \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t}
\end{aligned}$$

GRUs and LSTMs perform very similarly on most time-series related tasks, and currently, there is no concrete conclusion on which recurrent network is better [1].

5 Bidirectional RNN

As we know, due to the vanishing gradient problem, RNNs struggle to form long-term dependencies within their input sequences. A potential solution to this is to process the sequence in not only the forward direction but also the reverse direction. In this manner, we will preserve more information within the input sequence. The equations for a Bidirectional RNN are the following:

$$\begin{aligned}
\overrightarrow{h_t} &= f_h(x_t, \overrightarrow{h_{t-1}}) = \tanh(W_h^{(f)} \overrightarrow{h_{t-1}} + U_x^{(f)} x_t + b_f) \\
\overleftarrow{h_t} &= f_h(x_t, \overleftarrow{h_{t-1}}) = \tanh(W_h^{(r)} \overleftarrow{h_{t-1}} + U_x^{(r)} x_t + b_r) \\
o_t &= W_{yh}[\overrightarrow{h_t} : \overleftarrow{h_t}] + b_o
\end{aligned}$$

A bidirectional RNN can be thought of as having two RNNs, where one reads the input sequence in the forward direction and the other reads the input in the reverse direction. The output of the RNN is then found by concatenating the two hidden states and applying a linear transformation.

References

[1] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: (2014). arXiv: 1412.3555 [cs.NE].

[2] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: Proceedings of Machine Learning Research 9 (13–15 May 2010). Ed. by Yee Whye Teh and Mike Titterington, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[3] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: (2015). arXiv: 1502.01852 [cs.CV].

[4] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. "Understanding the exploding gradient problem". In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[5] Silviu Pitis. "Written Memories: Understanding, Deriving and Extending the LSTM". In: *R2RT* (2016).