ZZSC5836: Data Mining and Machine Learning

Week 3: (Part 1) Advances in Neural Networks

Dr Sarat Moka *Hexamester 3, 2024*

Key Topics

- Stochastic Gradient Descent
- Momentum and Nesterov momentum
- Adaptive learning rate GD methods
- Neural networks for time series data
- Dropout method
- Weight Decay (L2) regularization in neural network

Books:

(A) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Ed.) by Aurélien Géron (2019).

(B) Mathematical Engineering of Deep Learning book by Liquet, Moka, and Nazarathy (2024).

3.1 Stochastic Gradient Descent

- In the context of deep learning, the typical form of the loss function is

$$C(\theta) = \frac{1}{n} \sum_{i=1}^{n} C_i(\theta), \tag{3.1}$$

  where n is the number of data samples in a given dataset and $C_i(\theta)$ depends only on the i-th sample in the dataset. For instance, when the loss is mean squared error,

$$C_i(\theta) = (y_i - f_\theta(x_i))^2.$$

- As a consequence, the gradient loss function $C(\theta)$ is given by

$$\nabla C(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla C_i(\theta). \tag{3.2}$$

- The method of *stochastic gradient descent* exploits this structure by computing a noisy gradient using only one randomly selected training sample. That is, it evaluates the gradient for a single $C_i(\theta)$ instead of the gradient for all of $C(\theta)$.

- Computing $\nabla C_i(\theta)$ can be much faster using backpropagation than computing full gradient $\nabla C(\theta)$.

- More precisely, the algorithm operates similarly to gradient descent studied in the previous lecture, yet in the t-th iteration of stochastic gradient descent, an index variable I_t is randomly selected from the set $\{1, \ldots, n\}$ and the gradient $\nabla C_{I_t}(\theta)$ is computed only for the I_t-th data sample.

- In particular, the update rule for the decision variable is then,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \, \nabla C_{I_t}(\theta^{(t)}). \tag{3.3}$$

- This stochastic algorithm exhibits some interesting properties.

  1. Most importantly, the execution of each iteration can be much faster than that of basic gradient descent because only one sample is used at a time (the gain is of order n).
  2. Further, if the index variable I_t is selected uniformly over $\{1, \ldots, n\}$, the parameter update (3.3) is *unbiased* in the sense that the expected descent step for each iteration t is the same as that of gradient descent. That is, in each iteration, we have

$$\mathbb{E}\left[\nabla C_{I_t}(\theta)\right] = \frac{1}{n} \sum_{i=1}^{n} \nabla C_i(\theta) = \nabla C(\theta), \qquad \text{for a fixed } \theta. \tag{3.4}$$

and hence the stochastic step (3.3) is "on average correct".[1]

- While being unbiased, the stochastic nature of (3.3) introduces fluctuations in the descent direction.
- Such noisy trajectories may initially appear like a drawback, yet one advantage is that they enable the algorithm to escape flat regions, saddle points, and local minima.
- Thus, while the fluctuations make it difficult to guarantee convergence, in the context of deep learning their effect is often considered desirable in view of highly non-convex loss landscapes.
- Importantly, stochastic gradient descent is generally able to direct the parameters θ towards the region where minima of $C_i(\theta)$ are located.

Illustrative Example

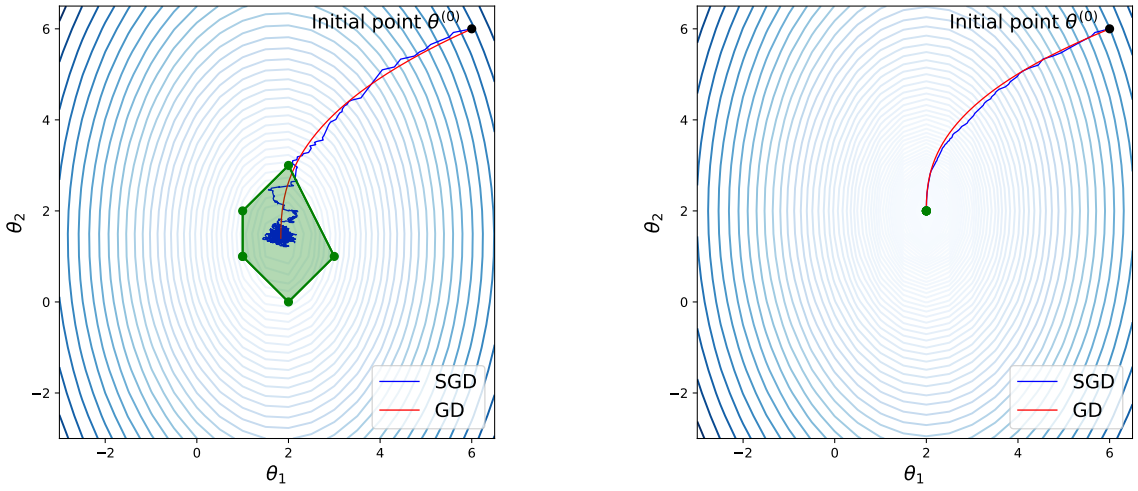
- To get a feel for the last attribute of stochastic gradient descent, we consider another hypothetical example where for $i = 1, \ldots, n$,

$$C_i(\theta) = a_i(\theta_1 - u_{i,1})^2 + (\theta_2 - u_{i,2})^2, \tag{3.5}$$

  for some constant $a_i > 0$ and $u_i = (u_{i,1}, u_{i,2}) \in \mathbb{R}^2$.
- Here u_i is the unique minimizer of $C_i(\theta)$. These individual loss functions for each observation are convex and hence the total loss function $C(\theta)$ of (3.1) is also convex.[2]
- Let $\mathcal{S}$ be the *convex hull* of $u_1, \ldots, u_n$, that is, $\mathcal{S}$ is the smallest convex set that contains $\{u_1, \ldots, u_n\}$. It is now obvious that the global minima of C is also in $\mathcal{S}$.
- In Figure 3.1 we plot the contours of such a function when $n = 5$ where in Figure 3.1 (a) the points $u_1, \ldots, u_5$ are distinct and in (b) these points are identical. The convex hull is marked by the region bounded in green.
- The figure also plots the evolution of gradient descent and stochastic gradient descent in each of the cases. As is evident, while the stochastic gradient descent trajectory is noisier, outside of the convex hull $\mathcal{S}$, its trajectory is similar to that of gradient descent.
- Interestingly, in Figure 3.1 (a), once the trajectory hits $\mathcal{S}$ it mostly stays within $\mathcal{S}$ but with a lot of fluctuations.

[1] The reader might be tempted to conclude that $\mathbb{E}[\theta^{(t)}_{\text{SGD}}] = \theta^{(t)}_{\text{GD}}$ for all time t, where the subscript SGD is for stochastic gradient descent starting at the same initial condition as GD ("Gradient Descent"). However, in general this is not correct when $\nabla C(\theta)$ is non-linear in θ.

[2] This example does not resemble a general case with multiple local minima since it is convex. Nevertheless, it is useful for understanding some of the behaviour of stochastic gradient descent.



(a) (b)

Figure 3.1: An hypothetical example where $C(\theta)$ is composed of individual $C_i(\theta)$ as in (3.5). Comparison of gradient descent (GD) and stochastic gradient descent (SGD). (a) A case where the minimizers $u_1, \ldots, u_n$ of $C_i(\theta)$ are different and are each marked by green dots in the plot. (b) A case where the minimizers $u_1, \ldots, u_n$ are the same.

- The reason for this behavior is that at any point θ not in $\mathcal{S}$, the descent directions for both gradient descent and stochastic gradient descent are towards the set $\mathcal{S}$ as it contains the minima of $C(\theta)$ and every $C_i(\theta)$.
- To make this a concrete argument, note that the descent step in the t-th iteration of the stochastic gradient descent is

$$-\alpha \nabla C_{I_t}(\theta^{(t)}) = -\alpha \sum_{i=1}^{n} \mathbf{1}\{I_t = i\} \nabla C_i(\theta^{(t)}).$$

- Since every $C_i(\theta)$ has its minimum in the set $\mathcal{S}$, if $\theta^{(t)} \notin \mathcal{S}$, then every $-\nabla C_i(\theta^{(t)})$, the steepest direction of $C_i(\theta)$ at $\theta^{(t)}$, guides the algorithm towards the set $\mathcal{S}$. Therefore, irrespective of the choice of I_t, the update moves $\theta^{(t+1)}$ towards $\mathcal{S}$.

Working with Mini-batches and the Concept of Epochs

- On one extreme, computing the full gradient for basic gradient descent is computationally challenging but the obtained gradients are noiseless. On the other extreme,

stochastic gradient descent reduces the computational complexity of each update but the updates are noisy.

- One middle ground approach is to work with mini-batches. Almost any practical deep learning training uses some variant of this approach.

- In partcular, a *mini-batch* is simply a small subset of the data points, of size n_b. Mini-batch sizes are typically quite small in comparison to the size of the dataset and are often chosen based on computational capabilities and constraints.

- In practice, a mini-batch size n_b is often selected so the computation of gradients associated with n_b observations can fit on GPU memory. This generally makes computation more efficient in comparison to stochastic gradient descent which cannot take full advantage of parallel computing or GPUs since the gradient evaluation for a single observation index I_t is not easy to parallelize.

- When using mini-batches, for each iteration t, a mini-batch with indices $\mathcal{I}_t \subseteq \{1, \ldots, n\}$ is used to approximate the gradient via,

$$\frac{1}{n_b} \sum_{i \in \mathcal{I}_t} \nabla C_i(\theta^{(t)}). \tag{3.6}$$

  This *estimated gradient* is then used as part of gradient descent or one of its variants (such as ADAM which we present in the sequel).

- One common practical approach with mini-batches is to shuffle the training data apriori and then use the shuffled indices sequentially. With this process there are n/n_b mini-batches[3] in the training dataset and each mini-batch is a distinct random subset of the indices.

- In such a case, when using some variant of gradient descent, every pass on the full dataset has n/n_b iterations, with one iteration per mini-batch. Such a pass on all the data via n/n_b iterations is typically called an *epoch*.

- The training process then involves multiple epochs. It is quite common to diagnose and track the training process in terms of epochs.

- Note that one may also randomly shuffle the data (assign new mini-batches) after every epoch to reduce the correlation between the epochs and reduce bias in the optimization process.

[3]We assume here that n_b divides n. If it is not the case then there are $\lceil n/n_b \rceil$ mini-batches with the last one having less than n_b samples.

- Similarly to (3.4) when using the mini-batch approach, the estimated gradient is unbiased in the sense that

$$\mathbb{E}\left[\frac{1}{n_b}\sum_{j=1}^{n_b}\nabla C_{I_{t,j}}(\theta)\right] = \frac{1}{n_b}\sum_{j=1}^{n_b}\frac{1}{n}\sum_{i=1}^{n}\nabla C_i(\theta) = \nabla C(\theta), \qquad \text{for a fixed } \theta.$$

  Here $I_{t,j}$ denotes the j-th element in $\mathcal{I}_t$ and the first equality holds because shuffling makes each $I_{t,j}$ to be uniform over $\{1, 2, \ldots, n\}$.

- Importantly, with min-batches, we notice that the noise of the gradients decreases as the mini-batch size n_b increases. To see this, recall that for each t, the index variables $I_{t,1}, \ldots, I_{t,n_b}$ are chosen independently. Thus, the variance of the gradient estimate is

$$\text{Var}\left(\frac{1}{n_b}\sum_{j=1}^{n_b}\nabla C_{I_{t,j}}(\theta^{(t)})\right) = \frac{1}{n_b}\text{Var}\left(\nabla C_{I_{t,1}}(\theta^{(t)})\right).$$

- Since $\text{Var}\left(\nabla C_{I_{t,1}}(\theta^{(t)})\right)$ is the variance of the gradient for an arbitrary data sample, we see that mini-batch gradient descent has n_b times smaller variance than that of stochastic gradient descent.

3.2 Momentum and Nesterov Momentum

- Recall that when $C(\theta)$ is the loss function with θ denoting all the weights and bias in the neural network, the (basic) gradient descent step at the $(t+1)$-th iteration is

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\nabla C(\theta^{(t)}),$$

  where α is the learning rate.

- For instance, for a weight $w_{i,j}$ in the network,

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} - \alpha\frac{\partial C(\theta^{(t)})}{\partial w_{i,j}}.$$

- Observe that the parameter update in the basic gradient descent depends only on the gradient at the current point.

- This implies that in flat regions of the loss landscape, gradient descent essentially stops. Further, at saddle points or local minima, gradient descent stops completely. Such drawbacks may be alleviated with the use of *momentum*.

Momentum

- To get an intuitive feel for the use of momentum in optimization, consider an analogy of rolling a ball downhill on the loss landscape. It gains momentum as it roles downhill, becoming increasingly faster until it reaches the bottom of a valley. Clearly, the ball keeps memory of the past forces in the form of acceleration. This motivates us to use exponential smoothing on the gradients to obtain an extra step called the *momentum update*.

- It is formulated as follows,

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C(\theta^{(t)}) \qquad \text{(Momentum Update}^4\text{)}, \tag{3.7}$$
$$\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t+1)} \qquad \text{(Parameter Update)}, \tag{3.8}$$

  for scalar *momentum parameter* $\beta \in [0, 1)$ and *learning rate* $\alpha > 0$, starting with $v^{(0)} = 0$.

- When $\beta = 0$, we have the basic gradient descent method and for larger β, information of previous gradients plays a more significant role.

- In practice, for deep neural networks, the gradient $\nabla C(\theta^{(t)})$ is replaced with a noisy gradient based on stochastic gradient descent or mini-batches.

- The vector $v^{(t)}$ in (3.7) is called the *momentum*[5] at the t-th update. We can also write that

$$v^{(t+1)} = (1-\beta)\sum_{\tau=0}^{t} \beta^{t-\tau}\nabla C(\theta^{(\tau)}), \quad \text{for} \quad t = 0, 1, 2, \ldots. \tag{3.9}$$

  That is, the momentum accumulates all the past (weighted) gradients, providing acceleration to the parameter θ updates on downward surfaces.

- Therefore, for $\beta > 0$, the next step taken via (3.8) is not necessarily taken in the steepest descent, instead the direction is dictated by the (exponential smoothing) average of all the gradients up to the current iteration.

- Figure 3.2 compares the performance of the gradient descent method with momentum for different values of β on the Rosenbrock function presented in a previous lecture. We observe that for large β, the momentum method accelerates as it takes downward steps.

[5] In physics, momentum is defined as the product of the mass of an object and its velocity vector. The object's momentum points in the direction of an object's movement. In contrast, here the momentum vector $v^{(t)}$ points in the direction opposite to the step taken.

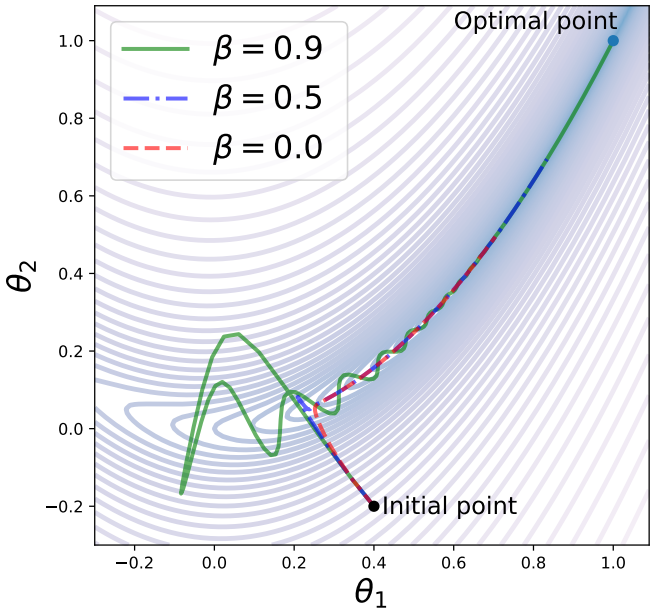


Figure 3.2: Application of momentum to the Rosenbrock function for three different values of β with learning rate $\alpha = 0.001/(1-\beta)$ and a fixed number of iterations. Note that $\beta = 0$ is basic gradient descent.

Nesterov Momentum

- We have seen that the momentum updates as in (3.7) and (3.8) accelerate like a ball rolling downhill. An issue with this method is that the steps do not slow down after reaching the bottom of a valley, and hence, there is a tendency to overshoot the valley floor. Therefore, if we know approximately where the ball will be after each step, we can slow down the ball before the hill slopes up again.
- The idea of *Nesterov momentum* tries to improve on standard momentum by using the gradient at the predicted future position instead of the gradient of the current position.
- The update equations are,

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C(\underbrace{\theta^{(t)} - \alpha\beta v^{(t)}}_{\substack{\text{predicted} \\ \text{next point}}}), \tag{3.10}$$

$$\theta^{(t+1)} = \underbrace{\theta^{(t)} - \alpha v^{(t+1)}}_{\substack{\text{actual} \\ \text{next point}}}, \tag{3.11}$$

  for constants $\beta \in [0, 1)$ and $\alpha > 0$, with $v^{(1)} = 0$. Compare (3.10) and (3.11) with (3.7) and (3.8), respectively.

- The difference here is that the gradient is computed at a predicted next point $\theta^{(t)} - \alpha\beta v^{(t)}$, which is a proxy for the (unseen) actual next point $\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t+1)}$ when $\beta \approx 1$.

- Implementing Nesterov momentum via (3.10) and (3.11) requires evaluation of the gradient at the predicted points instead of actual points. This may incur overheads, especially when incorporated as part of other algorithms.

- For example, if we also require $\nabla C(\theta^{(t)})$ at each iteration for purposes such as RMSprop, then the gradient needs to be computed twice instead of once per iteration; once for $\nabla C(\theta^{(t)})$ and once for $\nabla C(\theta^{(t)} - \alpha\beta v^{(t)})$.

- For this reason, and also for simplicity of implementing gradients only at $\theta^{(t)}$, a *look-ahead momentum* mechanism is sometimes used in place of (3.10) and (3.11).

- To see how this mechanism works, first revisit the basic momentum update equations (3.7) and (3.8) and observe that the parameter update can be represented as,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\big(\beta v^{(t)} + (1-\beta)\nabla C(\theta^{(t)})\big). \tag{3.12}$$

- Observe that $v^{(t)}$ is based on gradients at points $\theta^{(0)}, \ldots, \theta^{(t-1)}$, but not on the gradient at $\theta^{(t)}$. Hence a way to incorporate this last gradient is with look-ahead momentum where we replace $v^{(t)}$ of (3.12) by $v^{(t+1)}$. This achieves behaviour similar to Nesterov momentum.

- With such a replacement, we arrive at update equations of the form,

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C\big(\theta^{(t)}\big), \tag{3.13}$$

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\Big(\underbrace{\beta v^{(t+1)} + (1-\beta)\nabla C\big(\theta^{(t)}\big)}_{\substack{\text{look-ahead}\\ \text{momentum}}}\Big). \tag{3.14}$$

- Using (3.13) and (3.14) aims to provide similar behaviour to the Nesterov momentum equations (3.10) and (3.11), yet only requires evaluation of gradients at $\theta^{(t)}$ as opposed to a shifted point as in (3.10). While look-ahead momentum is not equivalent to Nesterov momentum, the gist of both methods is similar.

3.3 Adaptive Learning Rate Gradient Descent Methods

- Observe that the updates of the gradient descent method, with or without momentum, use the same learning rate α for all components of θ.

- However, it is often better to learn the parameters $\theta_1, \ldots, \theta_d$ with potentially a specific learning rate for each coordinate θ_i.
- A general approach for overcoming this issue is to scale each coordinate of the descent step with a different factor.
- That is, instead of considering a basic gradient descent step, $-\alpha \nabla C(\theta)$, consider descent steps of the form,

$$\text{step} = -\alpha\, r \odot \nabla C(\theta), \tag{3.15}$$

  where r is some vector of positive entries which recalibrates the descent steps and $\odot$ is the element-wise product of two vectors[6].
- However, in general, finding a fixed good vector r for (3.15) is difficult, especially when the number of parameters is huge and the actual relationship between parameters and features is not clear. Instead, we use adaptive methods that adjust the descent step during the optimization process.
- We now introduce four such adaptive approaches called *adaptive subgradient* (or *Adagrad*), *root mean square propagation* (or *RMSprop*), *Adadelta*, and *ADAM*.

Adagrad and RMSprop

- In both Adagrad and RMSprop, the update of coordinate i in iteration t can be represented as,

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\alpha}{\sqrt{s_i^{(t+1)}} + \varepsilon} \frac{\partial C(\theta^{(t)})}{\partial \theta_i}, \tag{3.16}$$

  where $s_i^{(0)}, s_i^{(1)}, s_i^{(2)}, \ldots$ is a sequence of non-negative values that are updated adaptively and ε taken to be a small value of order 1×10^{-8} to avoid division by zero.
- That is, with this representation, the descent step at time t represented in terms of (3.15) has $r_i = \left(\sqrt{s_i^{(t+1)}} + \varepsilon \right)^{-1}$.
- Generally we aim for s_i to be some smoothed representation of the square of the derivative $\partial C(\theta^{(t)})/\partial \theta_i$.
- Hence, r_i is roughly the inverse of the magnitude of the derivative, and r_i is low when the changes for coordinate i are steep and vise-versa.

[6]The element-wise product is also called the *Hadamard product* or *Schur product*.

- Vector-wise, the parameter update (3.16) can be represented as

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{s^{(t+1)}} + \varepsilon} \odot \nabla C(\theta^{(t)}), \tag{3.17}$$

where $s^{(t)} = (s_1^{(t)}, \ldots, s_d^{(t)})$, ε is considered a vector, and the addition, division, and square-root operations are all element-wise operations.

- Using the representation (3.16) or (3.17), let us now define how the sequence of $\{s_i^{(t)}\}$ is computed both for Adagrad and RMSprop. Specifically,

$$s_i^{(t+1)} = \begin{cases} \sum_{\tau=0}^{t} \left(\frac{\partial C(\theta^{(\tau)})}{\partial \theta_i}\right)^2, & \text{for Adagrad,} \\ \gamma s_i^{(t)} + (1-\gamma)\left(\frac{\partial C(\theta^{(t)})}{\partial \theta_i}\right)^2, & \text{for RMSprop,} \end{cases} \tag{3.18}$$

where the recursive relationship for RMSprop has the initial value at $s_i^{(0)} = 0$.

- RMSprop is also parameterized by a *decay parameter* $\gamma \in [0, 1)$ with typical values at $\gamma = 0.999$ implying that for RMSprop the sequence $\{s_i^{(t)}\}$ is a relatively slow exponential smoothing of the square of the derivative.

> **Remark**
>
> Adagrad may appear simpler than RMSprop since it does not involve any recursive step or any hyper-parameter like γ and it is simply an accumulation of the squares of the derivatives. However, the crucial drawback of Adagrad is that for each i, the sequence $\{s_i^{(t)}\}$ is strictly non-deceasing. As a consequence, the effective learning rate decreases during training, often making it infinitesimally small before convergence. RMSprop was introduced in the context of deep learning after Adagrad, and overcomes this problem via exponential smoothing of the squares of the partial derivatives.

- For RMSprop, the explicit (all-time) representation of the vector $s^{(t+1)}$ is,

$$s^{(t+1)} = (1-\gamma) \sum_{\tau=0}^{t} \gamma^{t-\tau} \left(\nabla C(\theta^{(\tau)}) \odot \nabla C(\theta^{(\tau)})\right), \tag{3.19}$$

while for Adagrad a similar representation holds without the $(1-\gamma)$ and $\gamma^{t-\tau}$ elements in the formula.

Adadelta

- Recall the computation of $s^{(t+1)}$ for the Adagrad method as in (3.18). When Adagrad's $s^{(t+1)}$ is used in (3.17), it has the problem of monotonically decreasing effective learning rates.

- This is one of the reasons that eventually RMSprop became more popular than Adagrad. However, there are other alternatives that are also very popular.
- One such alternative is the *Adadelta* method which uses exponential smoothing of the squared gradients as in RMSprop, but also uses another exponentially smoothed sequence of the descent step's squares.
- A key motivation for Adadelta is the observation that the update equation (3.17) for RMSprop or Adagrad uses a unit-less quantity as the descent step.
- Specifically, in (3.17) the only unit-full quantity in the coefficient multiplying $\nabla C(\theta^{(t)})$ is the reciprocal of $\sqrt{s^{(t+1)}}$. Hence that coefficient has units which are the inverse of the gradient and these cancel out the units of the gradient implying that the descent step is unit-less.
- With Adadelta, the update equation (3.17) is modified so that the descent step maintains the same units of the gradient, via

$$\theta^{(t+1)} = \theta^{(t)} \underbrace{- \frac{\sqrt{\Delta\theta^{(t)}} + \varepsilon}{\sqrt{s^{(t+1)}} + \varepsilon} \odot \nabla C(\theta^{(t)})}_{\text{Descent step } \widetilde{\nabla} C(\theta^{(t)})}, \tag{3.20}$$

  where $\Delta\theta^{(t)}$ is adaptively adjusted yet has the same units as $s^{(t+1)}$, making the coefficient of the gradient unit-free.
- Compare (3.20) with (3.17) to observe that $\sqrt{\Delta\theta^{(t)}} + \varepsilon$ replaces the learning rate α. This also means that Adagrad is "learning rate free". Instead, a potentially more robust parameter $\rho \in [0, 1)$ is used similarly to the γ parameter for RMSProp. This parameter specifies how to exponentially smooth squares of the descent steps.
- Using the descent step of (3.20), the update equation for $\Delta\theta^{(t)}$ is,

$$\Delta\theta^{(t)} = \rho\Delta\theta^{(t-1)} + (1-\rho)\left(\widetilde{\nabla} C(\theta^{(t-1)}) \odot \widetilde{\nabla} C(\theta^{(t-1)})\right),$$

  starting with $\Delta\theta^{(0)} = 0$.
- Then at iteration t, updating $\theta^{(t+1)}$ via (3.20), we use both $\Delta\theta^{(t)}$ and $s^{(t+1)}$, where the latter is updated via the RMSprop update equation as in (3.18).

ADAM

- Now that we understand ideas of momentum, and RMSprop, we can piece these together into a single algorithm, namely the *adaptive moment estimation* method, or simply *ADAM*.

- Metaphorically, if the execution of the momentum method is a ball rolling down a slope, the execution of ADAM can be seen as a heavy ball with friction rolling down the slope.
- The key update formula for ADAM is

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{1}{\sqrt{\hat{s}^{(t+1)}} + \varepsilon} \hat{v}^{(t+1)}, \tag{3.21}$$

  where all vector operations (division, square root, and addition of ε) are interpreted element wise. Here $\hat{v}^{(t+1)}$ and $\hat{s}^{(t+1)}$ are bias corrected exponential smoothing of the gradient and the squared gradients.

Algorithm 1: ADAM

Input: Dataset $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot)$, and
initial parameter vector $\theta^{(0)}$

Output: Approximatly *optimal* θ

$t \leftarrow 0$ (Initialize iteration counter)

$\theta \leftarrow \theta^{(0)}$, $v \leftarrow 0$, and $s \leftarrow 0$ (Initialize state vectors)

repeat

- $g \leftarrow \nabla C(\theta)$ (Compute gradient)
- $v \leftarrow \beta v + (1 - \beta)\, g$ (Momentum update)
- $s \leftarrow \gamma s + (1 - \gamma)\, (g \odot g)$ (Second moment update)
- $\hat{v} \leftarrow \dfrac{v}{1 - \beta^{t+1}}$ (Bias correction)
- $\hat{s} \leftarrow \dfrac{s}{1 - \gamma^{t+1}}$ (Bias correction)
- $\theta \leftarrow \theta - \alpha \dfrac{1}{\sqrt{\hat{s}} + \varepsilon} \hat{v}$ (Update parameters)
- $t \leftarrow t + 1$

until *termination condition* is satisfied

return θ

- With ADAM, α is still called the learning rate and is still the most important parameter which one needs to tune.
- The other parameters are $\beta \in [0, 1)$ for the momentum exponential smoothing as used in (3.7) and $\gamma \in [0, 1)$ for the RMSprop exponential smoothing as is used in (3.18).
- ADAM is presented in Algorithm 1 where Step 4 is the gradient computation, steps 5 and 6 are exponential smoothing (momentum and RMSprop), steps 7 and 8 are bias corrections, and finally Step 9 is the descent step.

Remark

Since the introduction of ADAM in 2014, this algorithm became the most widely used algorithm (or "optimizer") in deep learning frameworks. The common defaults for the hyper-parameters are $\beta = 0.9$ and $\gamma = 0.999$.

3.4 Neural Networks for Time Series Data

Neural networks are powerful tools for time series forecasting due to their ability to model complex, non-linear relationships. This section provides a mathematical overview of the steps involved in processing time series data, building neural networks for predictions, and evaluating their performance.

Forms of Sequence Data

- Denote a data sequence via $x = \left(x^{\langle 1\rangle}, \ldots, x^{\langle T\rangle}\right)$, where the superscripts $\langle t\rangle$ indicate time or position, and capture the order in the sequence.
- Each $x^{\langle t\rangle}$ is a p-dimensional numerical data point (or vector).
- The number of elements in the sequence, T, is sometimes fixed, but is also often not fixed and can be essentially unbounded.
- A classical example is a numerical univariate data sequences ($p = 1$) arising in time-series of economic, natural, or weather data.
- Similarly, multivariate time-series data ($p > 1$ but typically not huge) also arise in similar settings.
- Most common example sequence data is textual data. In this case, t is typically not the time of the text but rather the index of the word or token[7] within a text sequence.
- One way to encode text is that each $x^{\langle t\rangle}$ represents a single word using an *embedding vector*. If for example x is the text associated with the Bible then T is large,[8] whereas if x is the text associated with a movie review as per the IMDB movie dataset, then T is on average 231 words.

[7] In a complete treatment of textual data analysis or *natural language processing* (NLP) one requires to define and analyze *tokenizers* which break up text into natural "words" or parts of words known as *tokens*. These details are not our focus and we use "word" and "token" synonymously.

[8] By some counts, there are about half a million sequential words in the old testament of the Bible and more when one considers the new testament and its many variants.

- To help make the discussion concrete, assume momentarily that we encode input text in the simplest possible manner, where the embedding vector just uses a technique called *one-hot encoding*.
- With this approach we consider the number of words in the dictionary, vocabulary, or lexicon as $d_{\mathcal{V}}$ (e.g., $d_{\mathcal{V}} \approx 40,000$) and set $p = d_{\mathcal{V}}$.
- We then associate with each possible word, a unit vector $e_1, \ldots, e_p$ which uniquely identifies the word.
- At this point, an input data sequence (text) is converted into a sequence of vectors, where $x^{\langle t \rangle} = e_i$ whenever the t-th word in the sequence is the i-th word in lexicographic order in the dictionary.
- This approach is very simplistic and may appear inefficient. Yet it illustrates that textual data may be easily represented as a numerical input.

Data Processing

For simplicity, our discussion in this section assumes a single (typically long) univariate sequence x.

1. **Data Normalization** (optional): Normalize the time series data to ensure efficient training and convergence. A common normalization method is min-max scaling:

$$z^{\langle t \rangle} = \frac{x^{\langle t \rangle} - \min(x)}{\max(x) - \min(x)} \tag{3.22}$$

   where $x^{\langle t \rangle}$ is the original data point at time t, and $\{z^{\langle 1 \rangle}, z^{\langle 2 \rangle}, \ldots, z^{\langle T \rangle}\}$ is the normalized data point.

2. **Sequence Creation**: Create overlapping windows of data points to form input-output pairs for training the neural network. Given a time series $\{x_t\}$, create sequences of length p with corresponding labels:

$$\mathbf{s}_t = \{x^{\langle t \rangle}, x^{\langle t+1 \rangle}, \ldots, x^{\langle t+p-1 \rangle}\}, \quad y_t = x^{\langle t+p \rangle} \tag{3.23}$$

   where $\mathbf{s}_t$ is the input sequence and y_t is the output label.

3. **Train-Test Split**: Split the sequences into training and testing sets as

$$\text{Training Set} = \{(\mathbf{s}_t, y_t)\}_{t=1}^{T_{\text{train}}} \tag{3.24}$$

$$\text{Test Set} = \{(\mathbf{s}_t, y_t)\}_{i=T_{\text{train}}+1}^{T-p} \tag{3.25}$$

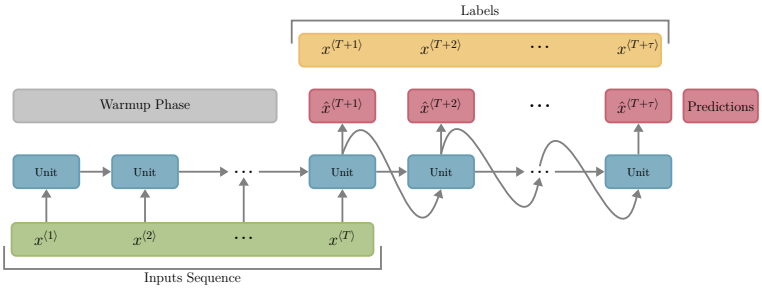
   where $T_{\text{train}} = \lfloor cT \rfloor$, for some $c \in (0, 1)$. Example, $c = 0.8$.

Prediction Using Neural Networks

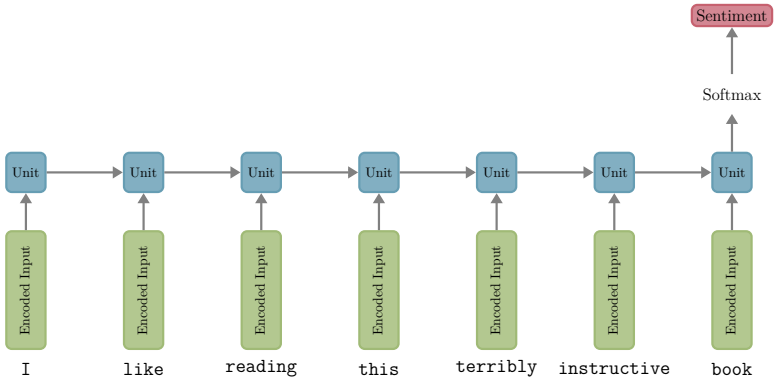
- There are plenty of tasks and applications involving sequence data. In the context of deep learning such tasks are handled by neural network models.
- The more classical forms of neural networks for sequence data are generally called *recurrent neural networks* (RNN), while more modern forms are called *transformers*.
- At this point assume that each of these models processes an input sequence x to create some output $\hat{y}$, where the creation of the output is sequential in nature.

- Figure 3.3 illustrates schematically how RNNs (or their generalizations) can be used. The building blocks of these types of models are called *units*, and they are recursively used in the computation of input to output.
- A basic task presented in (a) is *look-ahead prediction* which in the application context of text, implies predicting the next word (or collection of words) in a sequence.
- Another type of task presented in (b) is sequence regression or classification which can be used for applications such as *sentiment analysis*.
- An additional major task illustrated in (c) is *machine translation* where we translate the input sequence from one language to another (e.g., Hebrew to Arabic).
- Another type of task illustrated in (d) involves decoding an input into a sequence. One such example application is *image captioning* where text is generated to describe the input image.

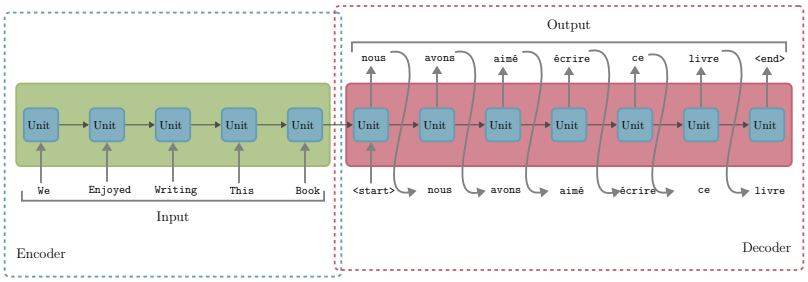
- With the tasks and applications highlighted, we see various forms of input x and output $\hat{y}$. Sometimes x and $\hat{y}$ are sequences and at other times they are not.
- It is often common to describe tasks and models as *one to many*, *many to one*, or *many to many*; see Figure 3.4.



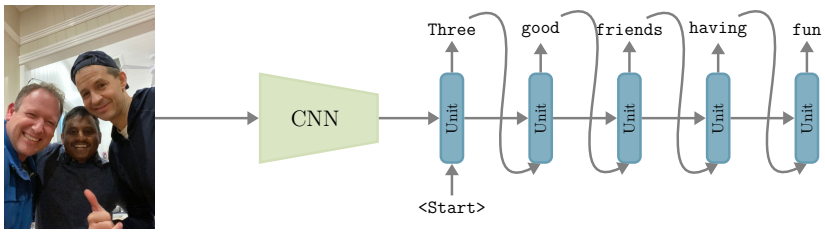
(a)



(b)

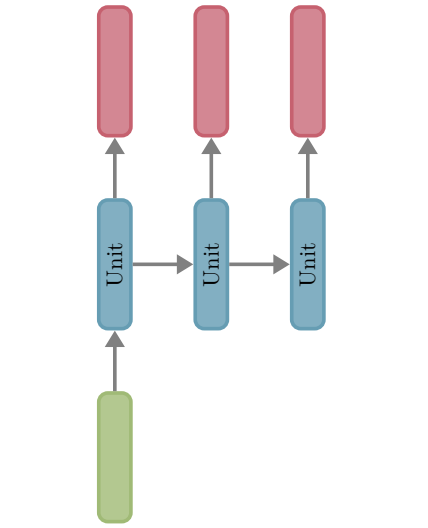


(c)

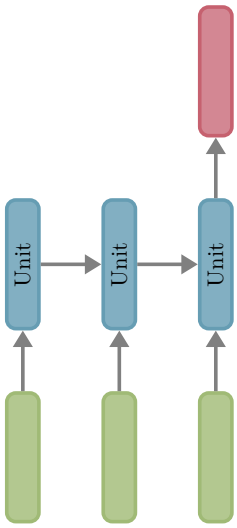


(d)

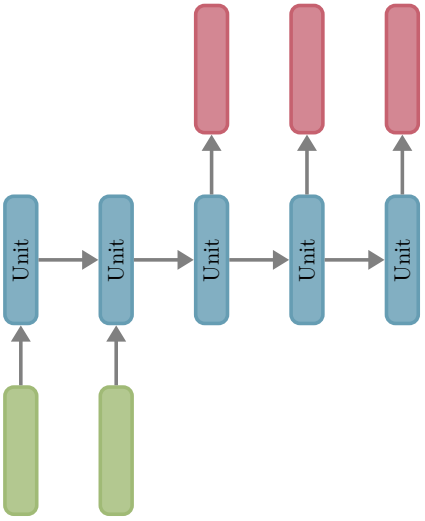
Figure 3.3: Use of recurrent neural network models (or generalizations) for various sequence data and language tasks. The basic building block, called a unit, is recursively used in the computation. (a) Lookahead prediction of the sequence. (b) Classification of a sequence or sentiment analysis. (c) Machine translation. (d) Image captioning.



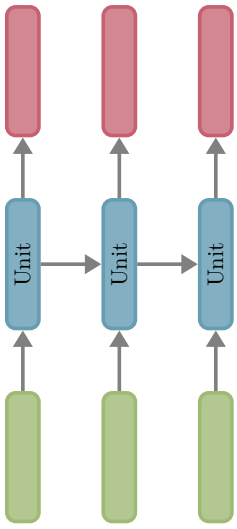
(a)



(b)



(c)



(d)

Figure 3.4: Input output paradigms of sequence models. (a) One-to-many. (b) Many-to-one. (c) Many-to-many with partial inputs and outputs. (d) Many-to-many with complete inputs and outputs.

1. **Model Architecture**: Choose a suitable neural network architecture, such as a *recurrent neural network* (Figure 3.5 (a)), a *Long Short-Term Memory* (LSTM) network (Figure 3.5 (b)), or a *gated recurrent unit* (GRU) architecture (Figure 3.5 (c)).

2. **Training the Model**: Train the model by minimizing the loss function. For example, using Mean Squared Error (MSE) as the loss function:

$$C(\theta) = \frac{1}{T_{\text{train}}} \sum_{i=1}^{T_{\text{train}}} (y_i - \hat{y}_i)^2 \tag{3.26}$$

   where $\hat{y}_i$ is the predicted value and θ represents the model parameters.

3. **Making Predictions**: Use the trained model to make predictions on the test set.

4. **Inverse Transformation**: If the data was normalized, apply the inverse transformation to convert predictions back to the original scale:

$$\hat{x}^{\langle t \rangle} = \hat{z}^{\langle t \rangle}(\max(x) - \min(x)) + \min(x) \tag{3.27}$$

   where $\hat{z}^{\langle t \rangle}$ is the denormalized predicted value.

Evaluation

1. **Performance Metrics**: Evaluate the model's performance using metrics such as Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE):

$$\text{MAE} = \frac{1}{T_{\text{test}}} \sum_{i=1}^{T_{\text{test}}} |\hat{y}_i - y_i| \tag{3.28}$$

$$\text{RMSE} = \sqrt{\frac{1}{T_{\text{test}}} \sum_{i=1}^{T_{\text{test}}} (\hat{y}_i - y_i)^2} \tag{3.29}$$

2. **Visualization**: Visualize the predictions against the true values to qualitatively assess the model's performance. Plot $\{\hat{y}_i\}$ versus $\{y_i\}$ to inspect how well the model forecasts the time series.

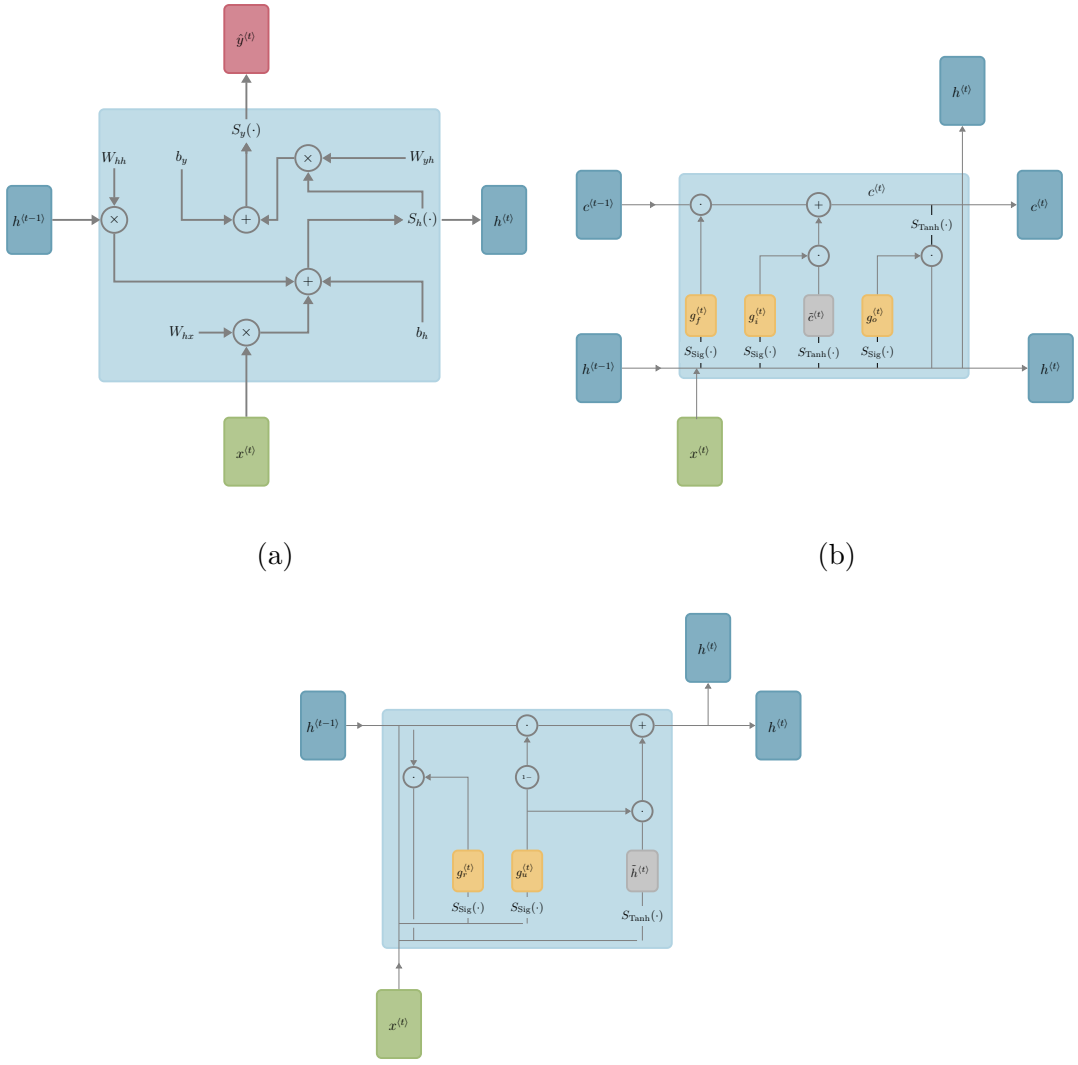


Figure 3.5: Representation of the RNN, LSTM and the GRU units. The output $\hat{y}^{\langle t \rangle}$ is not presented. (a) An RNN unit, also known as a gate, operating on input $x^{\langle t \rangle}$, and previous cell state $h^{\langle t-1 \rangle}$. The output vector of the unit is $\hat{y}^{\langle t \rangle}$. The unit also determines the cell state $h^{\langle t \rangle}$. (b) In LSTM there are three internal gates and the internal state is called the internal cell state. (b) In GRU there are two internal gates and the internal state is called the internal hidden state.

3.5 Dropout Method

- The idea of *dropout* is to randomly zero out certain neurons during the training process. This allows training to focus on multiple random subsets of the parameters and yields a form of regularization.
- With dropout, at any backpropagation iteration (forward pass and backward pass) on a mini-batch, only some random subset of the neurons is active.
- Practically neurons in layer ℓ, for $\ell = 0, \ldots, L-1$, have a specified probability $p_{\text{keep}}^{[\ell]} \in (0, 1]$ where if $p_{\text{keep}}^{[\ell]} = 1$ dropout does not affect the layer, and otherwise each neuron i of the layer is "dropped out" with probability $1 - p_{\text{keep}}^{[\ell]}$.
- This is simply a zeroing out of the neuron activation $a_i^{[\ell]}$ as we illustrate in Figure 3.6.

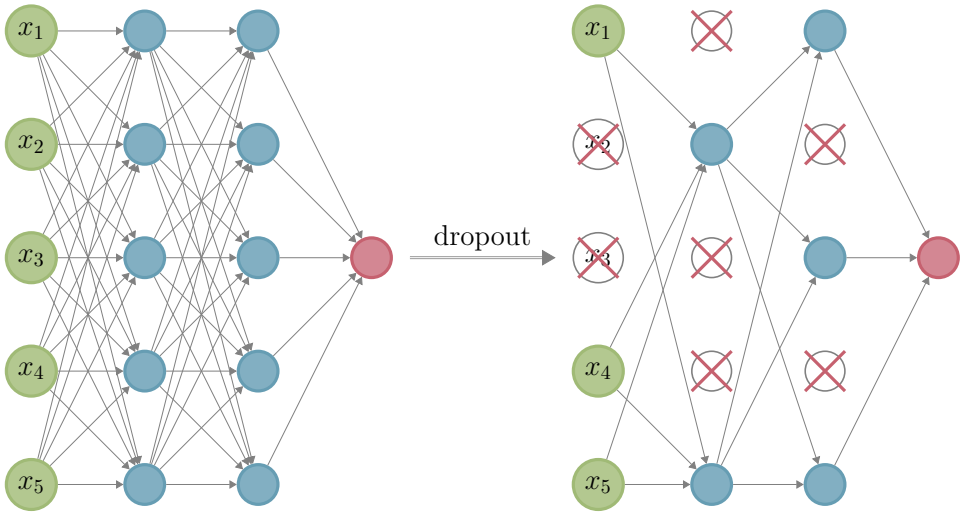


Figure 3.6: An illustration of dropout during training for a network with $L = 3$ layers, $p = 5$ input features, and $q = 1$ output. In each iteration during training, the network is transformed to the network on the right where the dropped out units are randomly selected.

- In the forward pass, when we get to the neurons of layer $\ell + 1$, all the neurons in layer ℓ that were zeroed out do not participate in the computation.
- Specifically, the update for a neuron j in the next layer, assuming a scalar activation

function $\sigma(\cdot)$, becomes,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i \text{ kept}} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big). \tag{3.30}$$

- In the backward pass, when neuron i is dropped out in layer ℓ, the weights $w_{i,j}^{[\ell+1]}$ for all neurons $j = 1, \ldots, N_{\ell+1}$ are updated based on the gradient $[g_W^{[\ell]}]_{i,j}$ which is set at 0.
- With a pure gradient descent optimizer this means that weights $w_{i,j}^{[\ell+1]}$ are not updated at all during the given iteration, whereas with a momentum based optimizer such as ADAM it means that the descent step for those weights has a smaller magnitude.
- At the end of training, even with dropout implemented, the trained model still has a complete set of weight matrices without any zeroed out elements, similar to the case in which we do not have dropout.
- Hence to account for the fact that neuron i in layer ℓ only took part in a proportion of iterations $p_{\text{keep}}^{[\ell]}$ during the training, when using the model in production (test time), we would like to use a weight matrix $\tilde{W}^{[\ell+1]} = p_{\text{keep}}^{[\ell]} W^{[\ell+1]}$ in place of $W^{[\ell+1]}$.
- The rationale here is to have the production forward pass similar to the training pass. Namely such a production forward pass has,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i=1}^{N_\ell} p_{\text{keep}}^{[\ell]} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big), \tag{3.31}$$

  and this serves as an approximation to (3.30).
- In practice, the training–production pair (3.30)–(3.31) is not typically used per-se. The more practical alternative is instead of remembering $p_{\text{keep}}^{[\ell]}$ and deploying it with the production model, the training forward pass is modified to have the reciprocal of $p_{\text{keep}}^{[\ell]}$ as a scaling factor of the weights. Namely, the training forward pass is,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i \text{ kept}} \frac{1}{p_{\text{keep}}^{[\ell]}} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big). \tag{3.32}$$

  This form allows to use the resulting model normally in production without having to take dropout into consideration at all. Namely, the production forward pass is,

$$a_j^{[\ell+1]} = \sigma\big(b_j^{[\ell+1]} + \sum_{i=1}^{N_\ell} w_{i,j}^{[\ell+1]} a_i^{[\ell]}\big), \tag{3.33}$$

- In practice, this simple and easy to implement idea of dropout has improved performance of deep neural networks in many empirically tested cases. It is now an integral part of deep learning training. We now explore the idea a bit further though the viewpoint of ensemble methods.

3.6 Weight Decay (L2) regularization

- In addition to dropout, addition of a regularization term is another key approach to prevent overfitting and improve generalization performance.
- Augmenting the loss with a regularization term $R_\lambda(\theta)$ restricts the flexibility of the model, and this restriction is sometimes needed to prevent overfitting.
- In the context of deep learning, and especially when ridge regression style regularization is applied, this practice is sometimes called *weight decay* when considering gradient based optimization.
- Take the original loss function $C(\theta)$ and augment it to be $\tilde{C}(\theta) = C(\theta) + R_\lambda(\theta)$. In our discussion here, let us focus on the ridge regression type regularization with parameter $\lambda > 0$, and,

$$R_\lambda(\theta) = \frac{\lambda}{2} R(\theta), \qquad \text{with} \qquad R(\theta) = \|\theta\|^2 = \theta_1^2 + \ldots + \theta_d^2.$$

- Further, we may even restrict regularization to certain layers and not others.
- Now assume we execute basic gradient descent steps. With a learning rate $\alpha > 0$, the update at iteration t is,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla \tilde{C}(\theta^{(t)}).$$

- In our ridge regression style penalty case we have $\nabla \tilde{C}(\theta) = \nabla C(\theta) + \lambda\theta$, and hence the gradient descent update can be represented as

$$\theta^{(t+1)} = (1 - \alpha\lambda)\theta^{(t)} - \alpha \nabla C(\theta^{(t)}). \tag{3.34}$$

- Now the interesting aspect of (3.34), assuming that $\alpha\lambda < 2$, is that it involves shrinkage or weight decay directly on the parameters in addition to gradient based learning. That is, independently of the value of the gradient $\nabla C(\theta^{(t)})$, in every iteration, (3.34) continues to decay the parameters, each time multiplying the previous parameter by a factor $1 - \alpha\lambda$.
- This weight decay phenomena can then be extended algorithmically to enforce regularization not directly via addition of a regularization term, but rather simply by augmenting the gradient descent updates to include weight decay.
- For example we may consider popular gradient based algorithms such as ADAM, and in each case add an additional step which incorporates multiplying the weights by a constant less than but close to unity.