5.5.4 自动求导更新及关闭

从前面的训练循环中，我们可以了解到我们只对 train_loss 调用了 backward()，因此，误差只会在训练集上反向传播。验证集用于提供一份独立的模型评估，评估模型对未用于训练的数据的输出的准确性。

好奇的读者此时会有一个小小的疑问，那就是对模型进行了 2 次评估，一次在 train_t_u 上，一次在 val_t_u 上，然后才调用 backward()，这难道不会让自动求导“迷惑”吗？难道 backward()不会受到在验证集上传递时生成的值的影响吗？

幸运的是，情况并非如此。训练循环中的第 1 行对 train_t_u 上的模型进行评估，以生成 train_t_p，然后从 train_t_p 评估 train_loss。这将创建一个计算图，将 train_t_u、train_t_p 和 train_loss 连接起来。当模型再次在 val_t_u 上求值时，将生成 val_t_p 和 val_loss。在本例中，将创建一个单独的计算图，将 val_t_u、val_t_p 和 val_loss 连接起来。将单独的张量经过相同的函数，即 model 和 loss_fn()运算，得到单独的计算图，如图 5.15 所示。

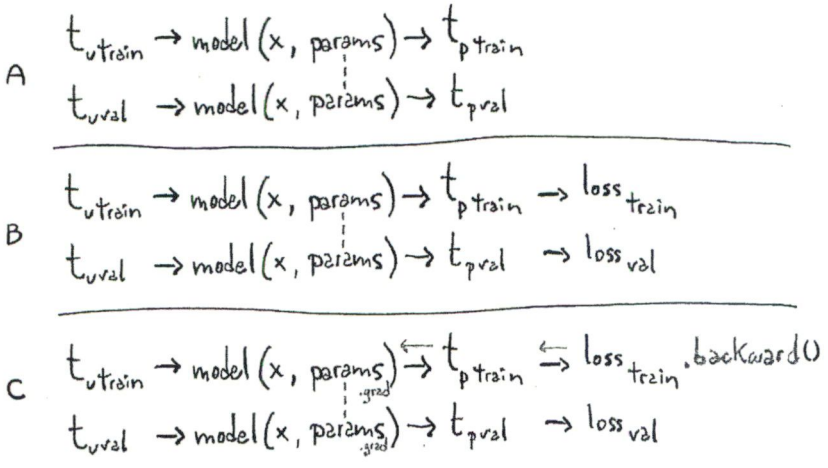

图 5.15 通过一张图显示了对于 2 个损失，其中一个损失调用了 backward()之后，梯度是如何传播的

图 5.15 中的 A、B、C 唯一相同的是张量的参数，当我们在 train_loss 上调用 backward()时，我们在第 1 张图上运行 backward()。换句话说，我们基于 train_t_u 生成的计算结果，将 train_loss 对参数的导数进行累加。

如果我们对 val_loss 误调用 backward()，则会累加 val_loss 相对同一叶节点上的参数的导数。还记得 zero_grad()吗？每次调用 backward()时，梯度都是相互累加的，除非我们显式地将梯度归零。嗯，这里会发生一些非常相似的事情：在 val_loss 上调用 backward()，在 train_loss.backward()调用生成的结果之上，将导致梯度在 params 张量中累加。在这种情况下，我们将在整个数据集上（训练集和验证集）有效地训练我们的模型，因为梯度将依赖于这二者，非常有趣。

这里还有一个需要讨论的因素，既然我们从来没有在 val_loss 上调用 backward()，那么为什