UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

TECSAR
Transformative Computer Systems and Architecture Research

Introduction to ML
Lecture 19: CNN Regularization

Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Regularization

- Training a model involves two critical steps:
- **Optimization:** when we need the loss to decrease on the training set
- **Generalization:** when the model has to work not only on the training set but also on data it has not seen before, like the validation set.

- The mathematical tools aimed at easing these two steps are sometimes subsumed under the label *regularization*.

Regularization: Weight Penalties

- The first way to stabilize generalization is to add a regularization term to the loss.
- This term is crafted so that the weights of the model tend to be small on their own, limiting how much training makes them grow.
- Overall, we want to keep the value of the weights relatively small with respect to our training loss
- **In other words, it is a penalty on larger weight values.**
- This makes the loss have a smoother topography, and there's relatively less to gain from fitting individual samples.

Regularization: Weight Penalties

- The most popular regularization terms of this kind are:

    a) L2 regularization: which is the sum of squares of all weights in the model

    b) L1 regularization: which is the sum of the absolute values of all weights in the model

- Both of them are scaled by a (small) factor, which is a hyperparameter we set prior to training.
- L2 regularization is also referred to as *weight decay*.
- So, adding L2 regularization to the loss function is equivalent to decreasing each weight by an amount proportional to its current value during the optimization step (hence, the name *weight decay*).
- Note that weight decay applies to all parameters of the network, such as biases.

Regularization: Weight Penalties

```
# In[45]:
def training_loop_l2reg(n_epochs, optimizer, model, loss_fn,
                        train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            imgs = imgs.to(device=device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            l2_lambda = 0.001
            l2_norm = sum(p.pow(2.0).sum()
                          for p in model.parameters())
            loss = loss + l2_lambda * l2_norm

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            loss_train += loss.item()
        if epoch == 1 or epoch % 10 == 0:
            print('{} Epoch {}, Training loss {}'.format(
                datetime.datetime.now(), epoch,
                loss_train / len(train_loader)))
```

Replaces pow(2.0) with abs() for L1 regularization

- In PyTorch, we could implement regularization pretty easily by adding a term to the loss.
- After computing the loss, whatever the loss function is, we can iterate the parameters of the model, sum their respective square (for L2) or `abs` (for L1), and backpropagate.
- The SGD optimizer in PyTorch already has a `weight_decay` parameter, and it directly performs weight decay during the update.
- It is fully equivalent to adding the L2 norm of weights to the loss, without the need for accumulating terms in the loss.

Regularization: Dropout

- An effective strategy for combating overfitting was originally proposed in 2014 by Nitish Srivastava and coauthors from Geoff Hinton's group in Toronto, in a paper aptly entitled

  "Dropout: a Simple Way to Prevent Neural Networks from Overfitting" (http://mng.bz/nPMa)

The idea behind dropout is indeed simple: zero out a random fraction of outputs from neurons across the network, where the randomization happens at each training iteration (epoch).

- This procedure effectively generates slightly different models with different neuron topologies at each iteration.
- It gives neurons in the model less chance to coordinate in the memorization process that happens during overfitting.

Regularization: Dropout

- In PyTorch, we can implement dropout in a model by adding an `nn.Dropout` module between the nonlinear activation function and the linear or convolutional module of the subsequent layer.
- As an argument, we need to specify the probability with which inputs will be zeroed out, basically defining the rates of zeroing out during each epoch.
- In case of convolutions, we'll use the specialized `nn.Dropout2d` or `nn.Dropout3d`

```
# In[47]:
class NetDropout(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_dropout = nn.Dropout2d(p=0.4)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                               padding=1)
        self.conv2_dropout = nn.Dropout2d(p=0.4)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = self.conv1_dropout(out)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = self.conv2_dropout(out)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Regularization: Dropout

- Note that dropout is normally active during training.
- During the evaluation of a trained model in production, dropout is bypassed or, equivalently, assigned a probability equal to zero.
- This is controlled through the `train` property of the `Dropout` module. Recall that PyTorch lets us switch between the two modalities by calling

```
model.train()
```

or

```
model.eval()
```

- The call will be automatically replicated on the submodules so that if `Dropout` is among them, it will behave accordingly in subsequent forward and backward passes.

Regularization: Batch Normalization

- Dropout was all the rage when, in 2015, another seminal paper was published by Sergey Ioffe and Christian Szegedy from Google, entitled

  > "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" (https://arxiv.org/abs/1502.03167).

- The paper described a technique that had multiple beneficial effects on training: allowing us to increase the learning rate and make training less dependent on initialization.
- It acts as a regularizer, thus representing an alternative to dropout.
- The main idea behind batch normalization is to rescale the inputs to the activations of the network so that minibatches have a certain desirable distribution.
- Recalling the mechanics of learning and the role of nonlinear activation functions, this helps avoid the inputs to activation functions being too far into the saturated portion of the function, thereby killing gradients and slowing training.

Regularization: Batch Normalization

- In practical terms, batch normalization shifts and scales an intermediate input using the mean and standard deviation collected at that intermediate location over the samples of the minibatch.
- The regularization effect is a result of the fact that an individual sample and its downstream activations are always seen by the model as shifted and scaled, depending on the statistics across the randomly extracted minibatch.
- The authors of the paper suggest that using batch normalization eliminates or at least alleviates the need for dropout

Regularization: Batch Normalization

- Batch normalization in PyTorch is provided through the `nn.BatchNorm1D`, `nn.BatchNorm2d`, and `nn.BatchNorm3d` modules, depending on the dimensionality of the input.
- Since the aim for batch normalization is to rescale the inputs of the activations, the natural location is after the linear transformation (convolution, in this case).

```
# In[49]:
class NetBatchNorm(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_batchnorm = nn.BatchNorm2d(num_features=n_chans1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                               padding=1)
        self.conv2_batchnorm = nn.BatchNorm2d(num_features=n_chans1 // 2)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.conv1_batchnorm(self.conv1(x))
        out = F.max_pool2d(torch.tanh(out), 2)
        out = self.conv2_batchnorm(self.conv2(out))
        out = F.max_pool2d(torch.tanh(out), 2)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

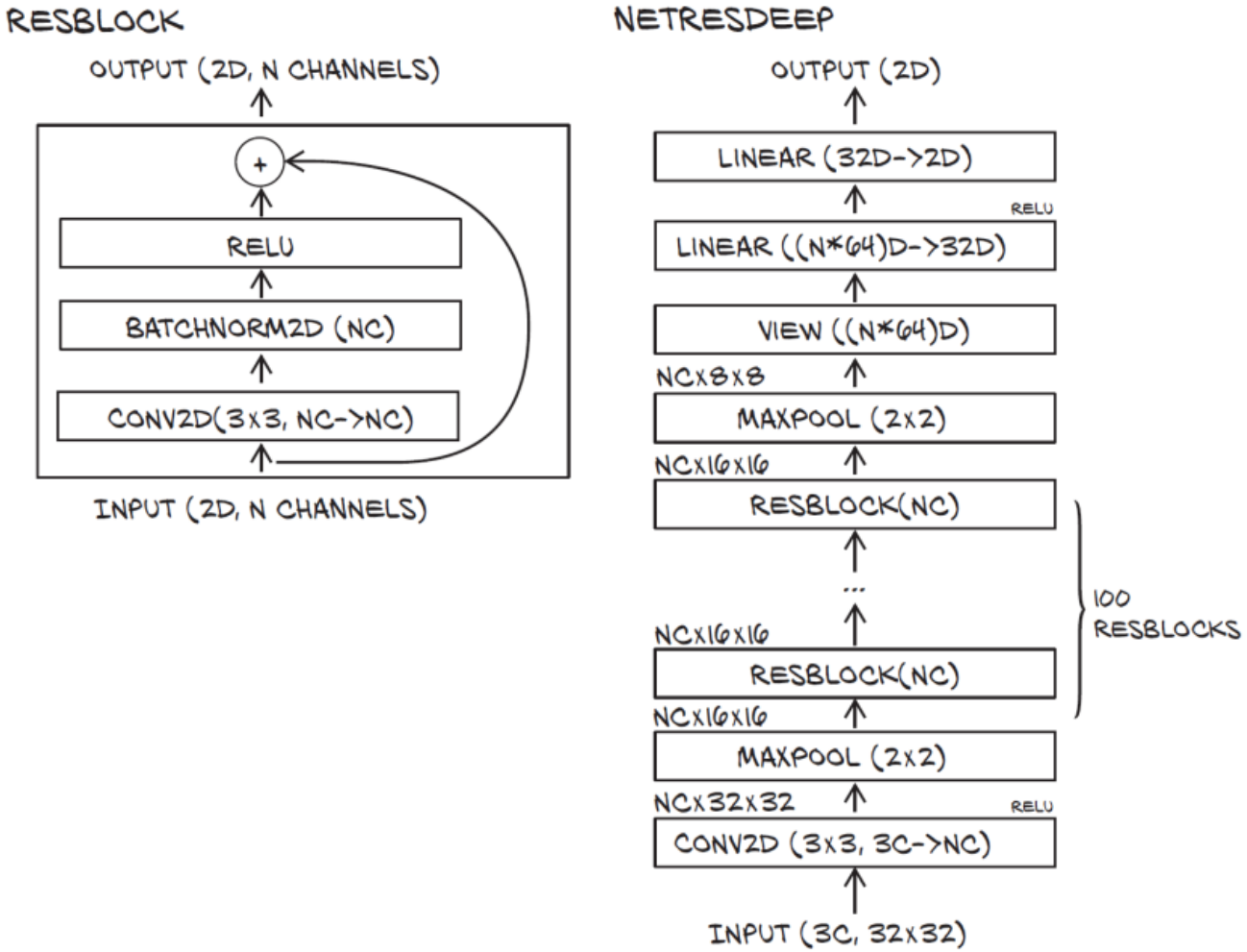
Regularization: Batch Normalization

- Just as for dropout, batch normalization needs to behave differently during training and inference.
- In fact, at inference time, we want to avoid having the output for a specific input depend on the statistics of the other inputs we're presenting to the model.
- As such, we need a way to normalize, but this time fixing the normalization parameters once and for all.
- As minibatches are processed, in addition to estimating the mean and standard deviation for the current minibatch, PyTorch also updates the running estimates for mean and standard deviation that are representative of the whole dataset, as an approximation.
- As minibatches are processed, in addition to estimating the mean and standard deviation for the current minibatch, PyTorch also updates the running estimates for mean and standard deviation that are representative of the whole dataset, as an approximation.
- This way, when the user specifies

```
model.eval()
```

- and the model contains a batch normalization module, the running estimates are frozen and used for normalization.

Building a Very Deep Network (ResNet)



Building a Very Deep Network

- We create a module subclass whose sole job is to provide the computation for one *block*—that is, one group of convolutions, activation, and skip connection:

Since we're planning to generate a deep model, we are including batch normalization in the block, since this will help prevent gradients from vanishing during training.

```
# In[55]:
class ResBlock(nn.Module):
    def __init__(self, n_chans):
        super(ResBlock, self).__init__()
        self.conv = nn.Conv2d(n_chans, n_chans, kernel_size=3,
                              padding=1, bias=False)
        self.batch_norm = nn.BatchNorm2d(num_features=n_chans)
        torch.nn.init.kaiming_normal_(self.conv.weight,
                                      nonlinearity='relu')
        torch.nn.init.constant_(self.batch_norm.weight, 0.5)
        torch.nn.init.zeros_(self.batch_norm.bias)

    def forward(self, x):
        out = self.conv(x)
        out = self.batch_norm(out)
        out = torch.relu(out)
        return out + x
```

The BatchNorm layer would cancel the effect of bias, so it is customarily left out.

Uses custom initializations . kaiming_normal_ initializes with normal random elements with standard deviation as computed in the ResNet paper. The batch norm is initialized to produce output distributions that initially have 0 mean and 0.5 variance.

Building a Very Deep Network

- We'd now like to generate a 100-block network.
- First, in *init*, we create `nn.Sequential` containing a list of `ResBlock` instances.
- Then, in `forward`, we just call the sequential to traverse the 100 blocks and generate the output

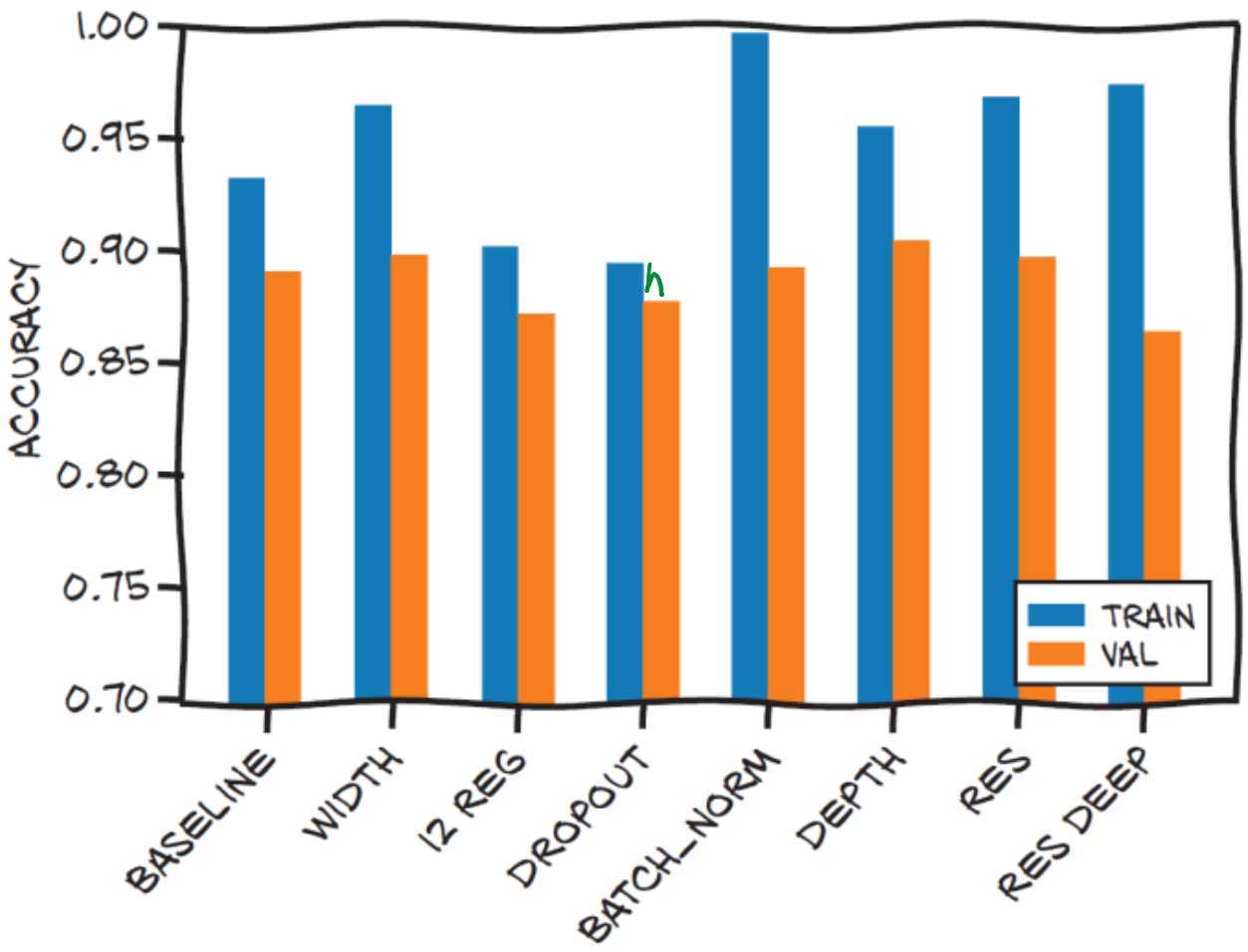
```
# In[56]:
class NetResDeep(nn.Module):
    def __init__(self, n_chans1=32, n_blocks=10):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.resblocks = nn.Sequential(
            *(n_blocks * [ResBlock(n_chans=n_chans1)]))
        self.fc1 = nn.Linear(8 * 8 * n_chans1, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = self.resblocks(out)
        out = F.max_pool2d(out, 2)
        out = out.view(-1, 8 * 8 * self.n_chans1)
        out = torch.relu(self.fc1(out))
        out = self.fc2(out)
        return out
```

Building a Very Deep Network

- In the implementation, we parameterize the actual number of layers, which is important for experimentation and reuse. Also, needless to say, backpropagation will work as expected.
- Unsurprisingly, the network is quite a bit slower to converge. It is also more fragile in convergence.
- This is why we used more-detailed initializations and trained our `NetRes` with a learning rate of 3e – 3 instead of the 1e – 2, we used for the other networks.

Comparing Design Knobs



- For this demonstration, we left all other things equal, from learning rate to number of epochs to train.
- Also, we would likely want to combine some of the additional design elements.
- The weight decay and dropout regularizations, which have a more rigorous statistical estimation interpretation as regularization than batch norm, have a much narrower gap between the two accuracies.
- Batch norm, which serves more as a convergence helper, lets us train the network to nearly 100% training accuracy, so we interpret the first two as regularization.