DATA 586: Advanced Machine Learning

2023W2

Shan Du

Training Neural Networks

- When training neural networks, forward and backward propagation depend on each other.
- In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path.
- These are then used for backpropagation where the compute order on the graph is reversed.

Training Neural Networks

- Therefore when training neural networks, after model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation.
- Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. One of the consequences is that we need to retain the intermediate values until backpropagation is complete.
- This is also one of the reasons why training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to *out of memory* errors.

Numerical Stability and Initialization

- The choice of initialization scheme of parameters plays a significant role in neural network learning, and it can be crucial for maintaining numerical stability.
- Moreover, these choices can be tied up in interesting ways with the choice of the nonlinear activation function. Which function we choose and how we initialize parameters can determine how quickly our optimization algorithm converges.
- Poor choices here can cause us to encounter exploding or vanishing gradients while training.

Vanishing and Exploding Gradients

- Gradients of unpredictable magnitude threaten the stability of our optimization algorithms.
- We may be facing parameter updates that are either
  - (i) excessively large, destroying our model (the exploding gradient problem);
  - or (ii) excessively small (the vanishing gradient problem), rendering learning impossible as parameters hardly move on each update.

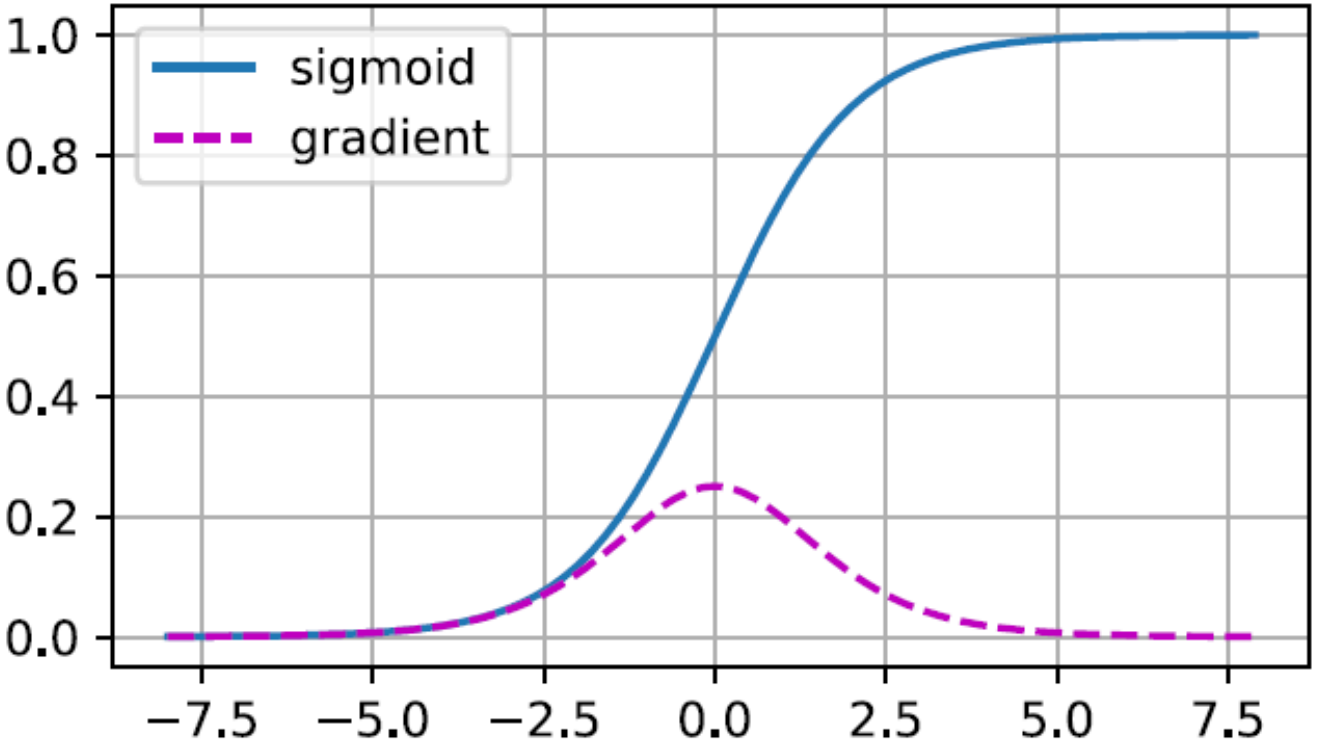
Vanishing Gradients

- One frequent reason causing the vanishing gradient problem is the choice of the activation function that is appended following each layer's linear operations. Historically, the sigmoid function was popular because it resembles a thresholding function.
- Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either *fully* or *not at all* (like biological neurons) seemed appealing.

Vanishing Gradients

- The sigmoid's gradient vanishes both when its inputs are large and when they are small. Moreover, when backpropagating through many layers, the gradients of the overall product may vanish. When our network boasts many layers, unless we are careful, the gradient will likely be cut off at some layer.
- Indeed, this problem used to plague deep network training. Consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice for practitioners.

Vanishing Gradients



Exploding Gradients

- The opposite problem, when gradients explode, can be similarly annoying. To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scale that we picked (the choice of the variance = 1), the matrix product explodes.
- When this happens due to the initialization of a deep network, we have no chance of getting a gradient descent optimizer to converge.

Exploding Gradients

```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))

print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[ 0.3777, -0.5238, -0.5041,  0.9692],
        [-0.9990,  0.6596, -0.4777,  0.3969],
        [ 0.8370,  0.3447, -0.1883, -1.1816],
        [-0.0491,  0.2414,  0.0711, -0.5202]])
after multiplying 100 matrices
 tensor([[ 9.0238e+23,  9.6501e+23,  7.8720e+23, -3.6753e+23],
        [-9.4837e+23, -1.0142e+24, -8.2732e+23,  3.8626e+23],
        [-1.0684e+24, -1.1425e+24, -9.3201e+23,  4.3514e+23],
        [-5.2101e+23, -5.5717e+23, -4.5451e+23,  2.1220e+23]])
```

Breaking the Symmetry

- Another problem in neural network design is the symmetry inherent in their parametrization.
- Assume that we have a simple MLP with one hidden layer and two units.
- Suppose that the output layer transforms the two hidden units into only one output unit, imagine what would happen if we initialized all of the parameters of the hidden layer as some constant *c.*

Breaking the Symmetry

- In this case, during forward propagation either hidden unit takes the same inputs and parameters, producing the same activation, which is fed to the output unit.
- During backpropagation, differentiating the output unit with respect to parameters gives a gradient whose elements all take the same value.
- Such iterations would never *break the symmetry* on its own and we might never be able to realize the network's expressive power. The hidden layer would behave as if it had only a single unit.

Breaking the Symmetry

- Note that while minibatch stochastic gradient descent would not break this symmetry, dropout regularization would!
- Another way of addressing—or at least mitigating—the issues raised above is through careful initialization.

Generalization in Deep Learning

- According to the “no free lunch” theorem by Wolpert et al. (1995), any learning algorithm generalizes better on data with certain distributions, and worse with other distributions.
- Our approach to training them typically consists of two phases: (i) fit the training data; and (ii) estimate the generalization error (the true error on the underlying population) by evaluating the model on holdout data.
- The difference between our fit on the training data and our fit on the test data is called the *generalization gap* and when the generalization gap is large, we say that our models overfit to the training data.

Generalization in Deep Learning

- In extreme cases of overfitting, we might exactly fit the training data, even when the test error remains significant. And in the classical view, the interpretation is that our models are too complex, requiring that we either shrink the number of features, the number of nonzero parameters learned, or the size of the parameters as quantified.

Generalization in Deep Learning

- A new line of work (Rolnick *et al.,* 2017) has revealed that in the setting of label noise, neural networks tend to fit cleanly labeled data first and only subsequently to interpolate the mislabeled data.
- Moreover, it's been established that this phenomenon translates directly into a guarantee on generalization: whenever a model has fitted the cleanly labeled data but not randomly labeled examples included in the training set, it has in fact generalized (Garg *et al*., 2021).

Generalization in Deep Learning

- Together these findings help to motivate *early stopping*, a classic technique for regularizing deep neural networks.
- Here, rather than directly constraining the values of the weights, one constrains the number of epochs of training.
- The most common way to determine the stopping criteria is to monitor validation error throughout training (typically by checking once after each epoch) and to cut off training when the validation error has not decreased by more than some small amount ϵ for some number of epochs.
- This is sometimes called a *patience criteria*.

Dropout

- Classical generalization theory suggests that to close the gap between train and test performance, we should aim for a simple model.
- Simplicity can come in the form of a small number of dimensions.
- Another useful notion of simplicity is smoothness, i.e., that the function should not be sensitive to small changes to its inputs.

Dropout

- In 1995, Christopher Bishop formalized this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995).
- This work drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), and the requirement that it be resilient to perturbations in the input.

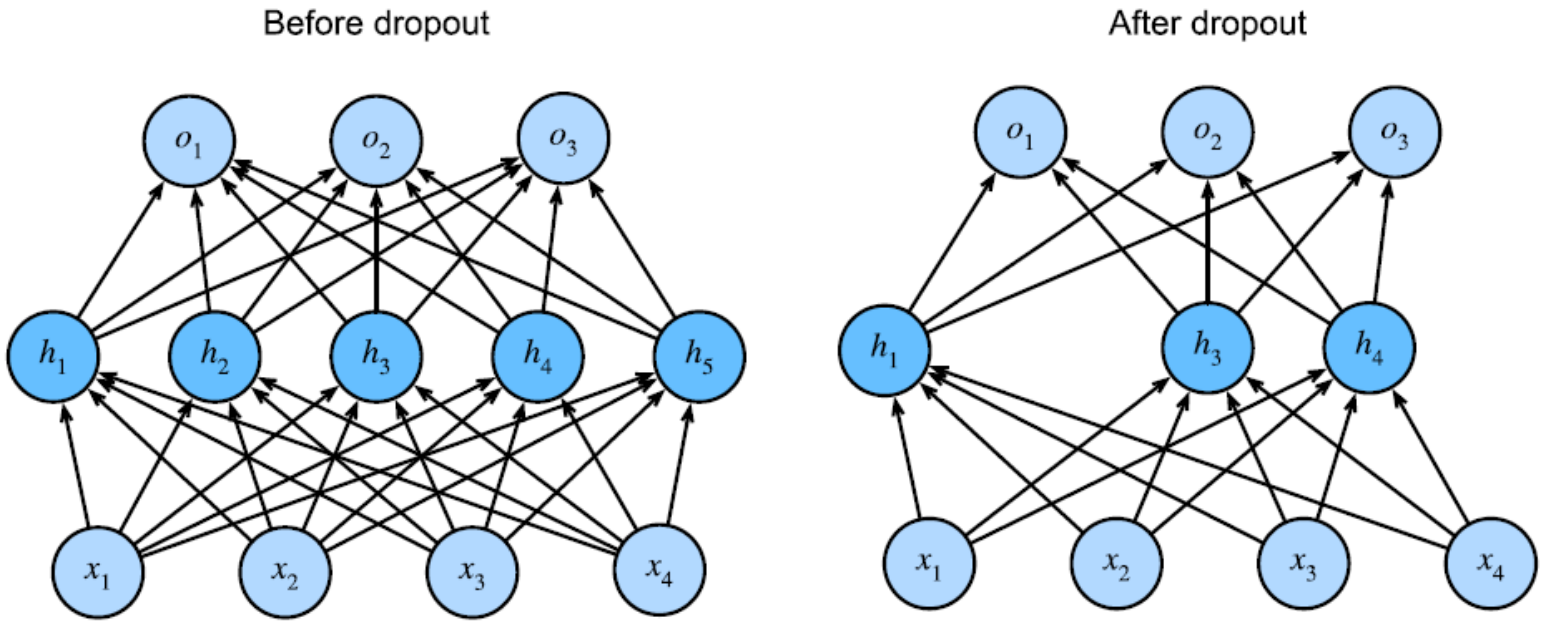
Dropout

- Then, in 2014, Srivastava et al. (2014) developed a clever idea for how to apply Bishop's idea to the internal layers of a network, too. Their idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks.
- The method is called *dropout* because we literally drop out some neurons during training. Throughout training, on each iteration, standard dropout consists of zeroing out some fraction ϕ of the nodes in each layer before calculating the subsequent layer.

Dropout

- This is done separately each time a training observation is processed. The surviving units stand in for those missing, and their weights are scaled up by a factor of $1/(1-\phi)$ to compensate.
- The key challenge is how to inject this noise. One idea is to inject the noise in an unbiased manner so that the expected value of each layer—while fixing the others—equals to the value it would have taken absent noise.

Dropout



MLP before and after dropout.

When we apply dropout to a hidden layer, zeroing out each hidden unit with probability *p*, the result can be viewed as a network containing only a subset of the original neurons.

Dropout

- Typically, we disable dropout at test time. Given a trained model and a new example, we do not drop out any nodes and thus do not need to normalize.
- However, there are some exceptions: some researchers use dropout at test time as a heuristic for estimating the uncertainty of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident.

Dropout

- This prevents nodes from becoming over-specialized, and can be seen as a form of regularization. In practice dropout is achieved by randomly setting the activations for the “dropped out” units to zero, while keeping the architecture intact.

Norms and Weight Decay

- Rather than directly manipulating the number of parameters, *weight decay*, operates by restricting the values that the parameters can take.
- More commonly called ℓ_2 regularization outside of deep learning circles when optimized by minibatch stochastic gradient descent, weight decay might be the most widely used technique for regularizing parametric machine learning models.

Norms and Weight Decay

- The complexity of a linear function $f(\mathrm{x}) = w^T \mathrm{x}$ can be measured by some norm of its weight vector, e.g., $\|w\|^2$.
- The most common method for ensuring a small weight vector is to add its norm as a penalty term to the problem of minimizing the loss.
- Thus we replace our original objective, *minimizing the prediction loss on the training labels*, with new objective, *minimizing the sum of the prediction loss and the penalty term*.

Norms and Weight Decay

- If our weight vector grows too large, our learning algorithm might focus on minimizing the weight norm $\|w\|^2$ vs. minimizing the training error. That is exactly what we want.
- Our loss was given by

$$L(w,b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(w^T \mathrm{x}^{(i)} + \mathrm{b} - y^{(i)})^2$$

$\mathrm{x}^{(i)}$ are the features, $y^{(i)}$ is the label for any data example i, and (w, b) are the weight and bias parameters, respectively.

Norms and Weight Decay

- To penalize the size of the weight vector, we must somehow add $\|w\|^2$ to the loss function, but how should the model trade off the standard loss for this new additive penalty? In practice, we characterize this tradeoff via the *regularization constant* λ, a non-negative hyperparameter that we fit using validation data:

$$L(w, b) + \frac{\lambda}{2}\|w\|^2$$

Norms and Weight Decay

- For $\lambda = 0$, we recover our original loss function. For $\lambda > 0$, we restrict the size of

$\| w \|$.

- We divide by 2 by convention: when we take the derivative of a quadratic function, the 2 and 1/2 cancel out, ensuring that the expression for the update looks nice and simple.

Norms and Weight Decay

- One reason to work with the ℓ_2 norm is that it places an outsize penalty on large components of the weight vector. This biases our learning algorithm towards models that distribute weight evenly across a larger number of features.
- In practice, this might make them more robust to measurement error in a single variable.
- By contrast, ℓ_1 penalties lead to models that concentrate weights on a small set of features by clearing the other weights to zero. This gives us an effective method for feature selection, which may be desirable for other reasons.

Norms and Weight Decay

- The minibatch stochastic gradient descent updates for ℓ_2-regularized regression follow:

$$w \leftarrow (1 - \eta\lambda)w - \frac{\eta}{|B|}\sum_{i \in B} \mathrm{x}^{(i)}(w^T \mathrm{x}^{(i)} + \mathrm{b} - y^{(i)})$$

- As before, we update w based on the amount by which our estimate differs from the observation. However, we also shrink the size of w towards zero. That is why the method is sometimes called “weight decay”: given the penalty term alone, our optimization algorithm decays the weight at each step of training.

Network Tuning

- *The number of hidden layers, and the number of units per layer.*
- *Regularization tuning parameters.*
  - These include the dropout rate λ and the strength ϕ of lasso and ridge regularization, and are typically set separately at each layer.
- *Details of stochastic gradient descent.*
  - These include the batch size, the number of epochs, and if used, details of data augmentation