Deep Learning Theory and Applications

Weight Initialization and Hyper-parameter selection

Yale

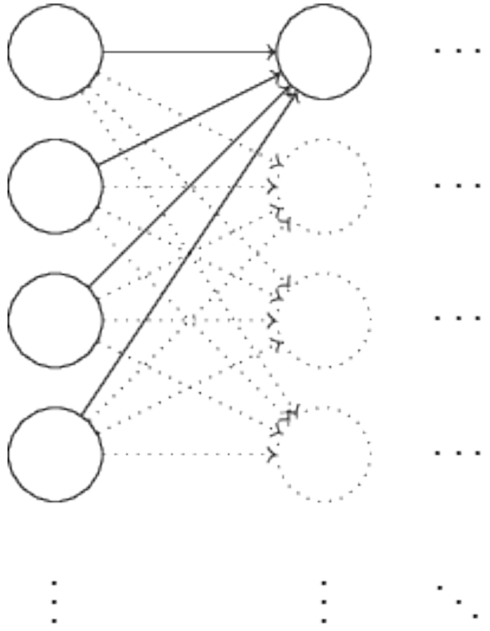
CPSC/AMTH 663

1. Weight Initialization
2. Hyper-parameter selection
    1. Broad strategy
    2. More specifics

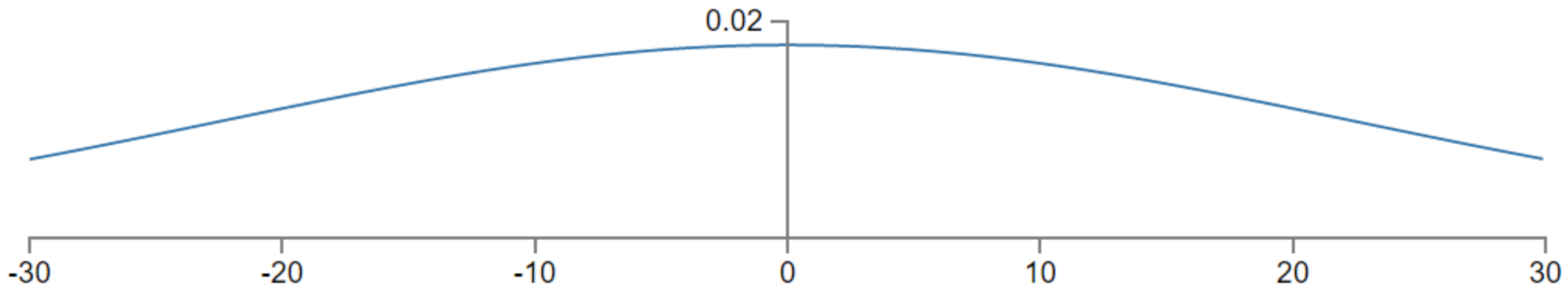
Weight initialization

- Deep learning models are quite sensitive to initialization
    - Cost functions are nonconvex and can have local minima
- Current approach: initialize weights and biases using independent Gaussian random variables
    - Mean 0 and standard deviation (std) 1
- Can we improve on this approach?
- Yes!

- Suppose we have a large network with 1,000 input neurons
  - Initialize weights connecting to the first hidden layer using standard normal random variables (RV)
  - We'll focus on the weights connecting to the first hidden neuron
- Suppose we train with an x with half the input neurons activated (i.e. set to 1)
- Input to hidden neuron: $z = \sum_j w_j x_j + b$
- z is a sum of 501 standard normal RVs
- z is a Gaussian with mean zero and std $\sqrt{501} \approx 22.4$



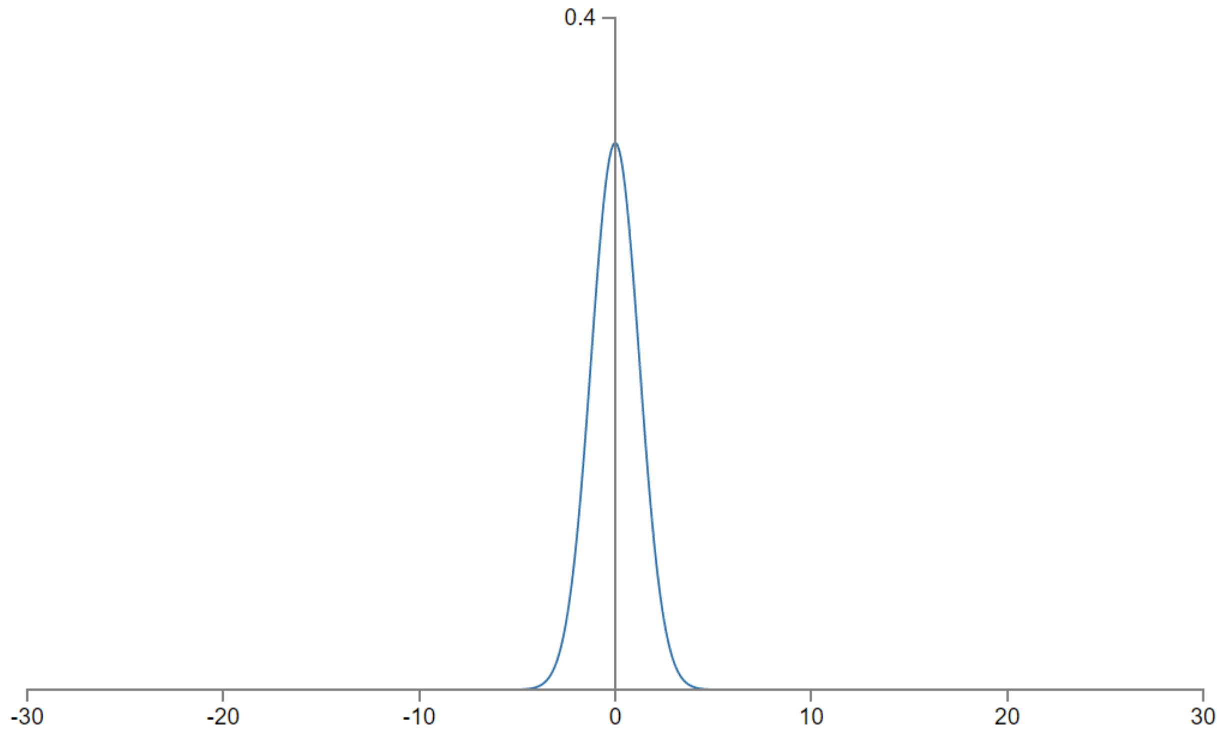
- Input to hidden neuron: $z = \sum_j w_j x_j + b$
- z is a Gaussian with mean zero and std $\sqrt{501} \approx 22.4$



- $|z|$ will likely be large (i.e. $|z| \gg 1$)
    - $\sigma(z)$ will likely be very close to 1 or 0 (i.e. saturated)
    - Thus making small changes in the weights makes tiny changes in $\sigma(z)$
    - Weights will learn very slowly via gradient descent

- Can we improve our initializations to avoid this?
- Suppose we have a neuron with n_{in} input weights
- Initialize the weights as Gaussian RVs with mean 0 and std $\frac{1}{\sqrt{n_{in}}}$
- Initialize bias with Gaussian RV with mean 0 and std 1
- $z = \sum_j w_j x_j + b$ will be a much more sharply peaked RV

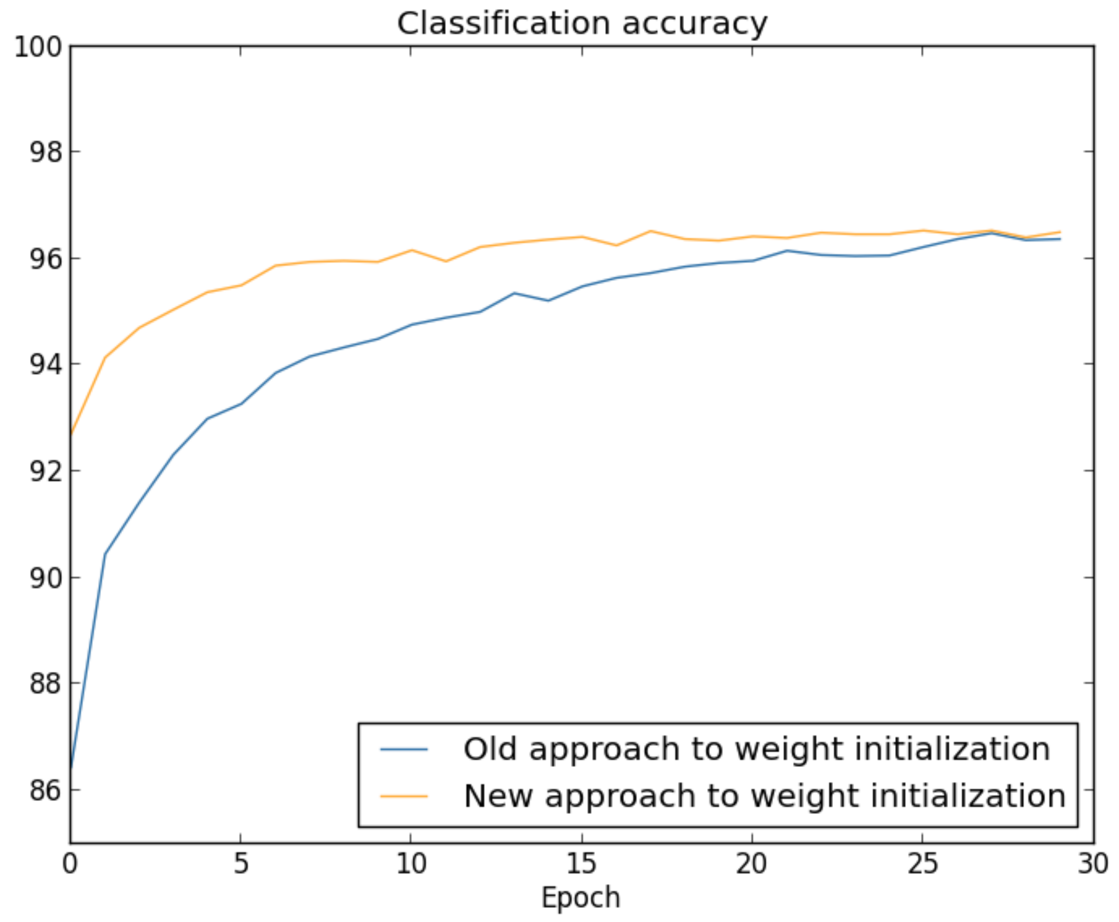
- Consider the previous case
  - 500 inputs are 0 and 500 outputs are 1
  - Then z has a Gaussian distribution with mean 0 and std $\sqrt{\frac{3}{2}} \approx 1.22$



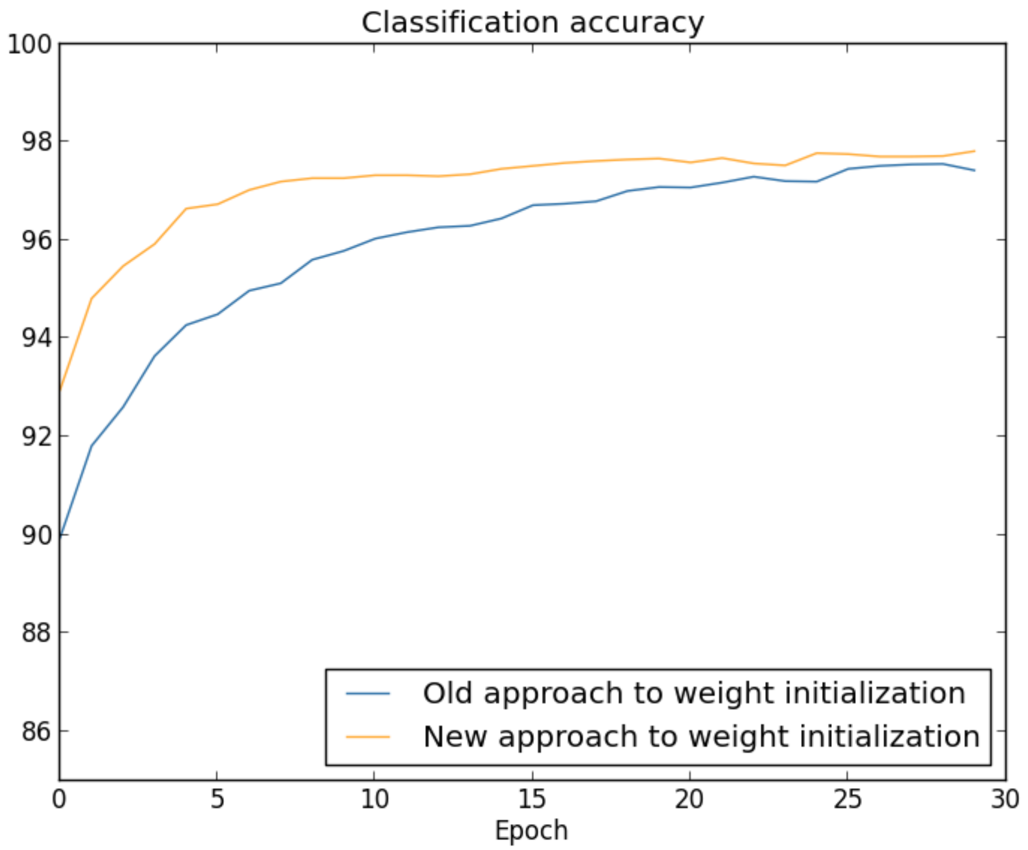
- Saturation is much less likely to occur

- What about the biases?
- Using standard normal RV is unlikely to saturate the neuron
- It doesn’t matter too much how the biases are initialized as long as saturation is avoided
  - Some people initialize biases to zero

- 30 hidden neurons, mini-batch size of 10, $\lambda = 5.0$, cross-entropy cost function, $\eta = 0.1$



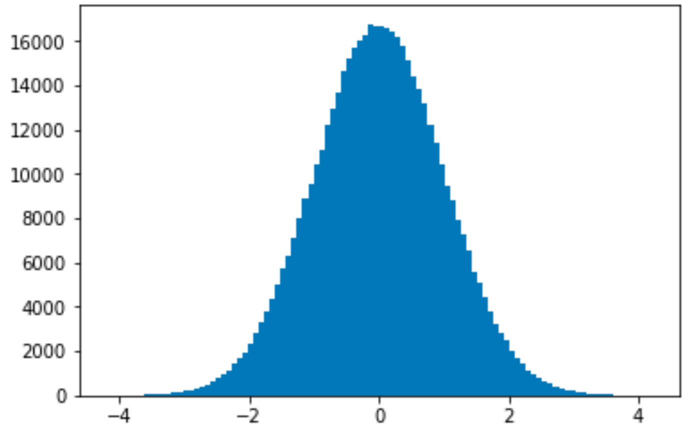
- 100 hidden neurons

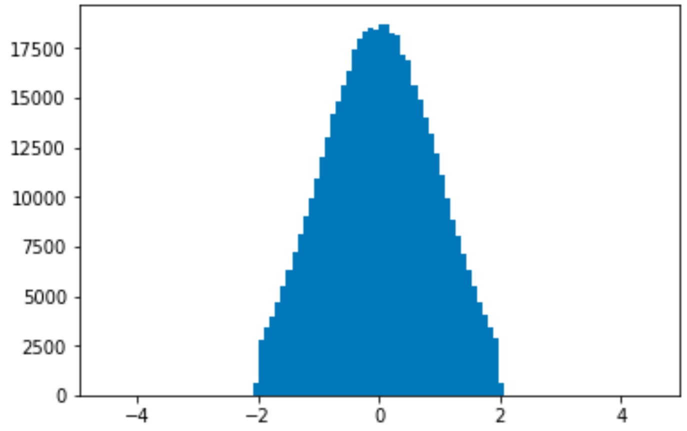


- Seems only to speed up learning
- We’ll see cases later where it improves final performance

Different networks, different initializations

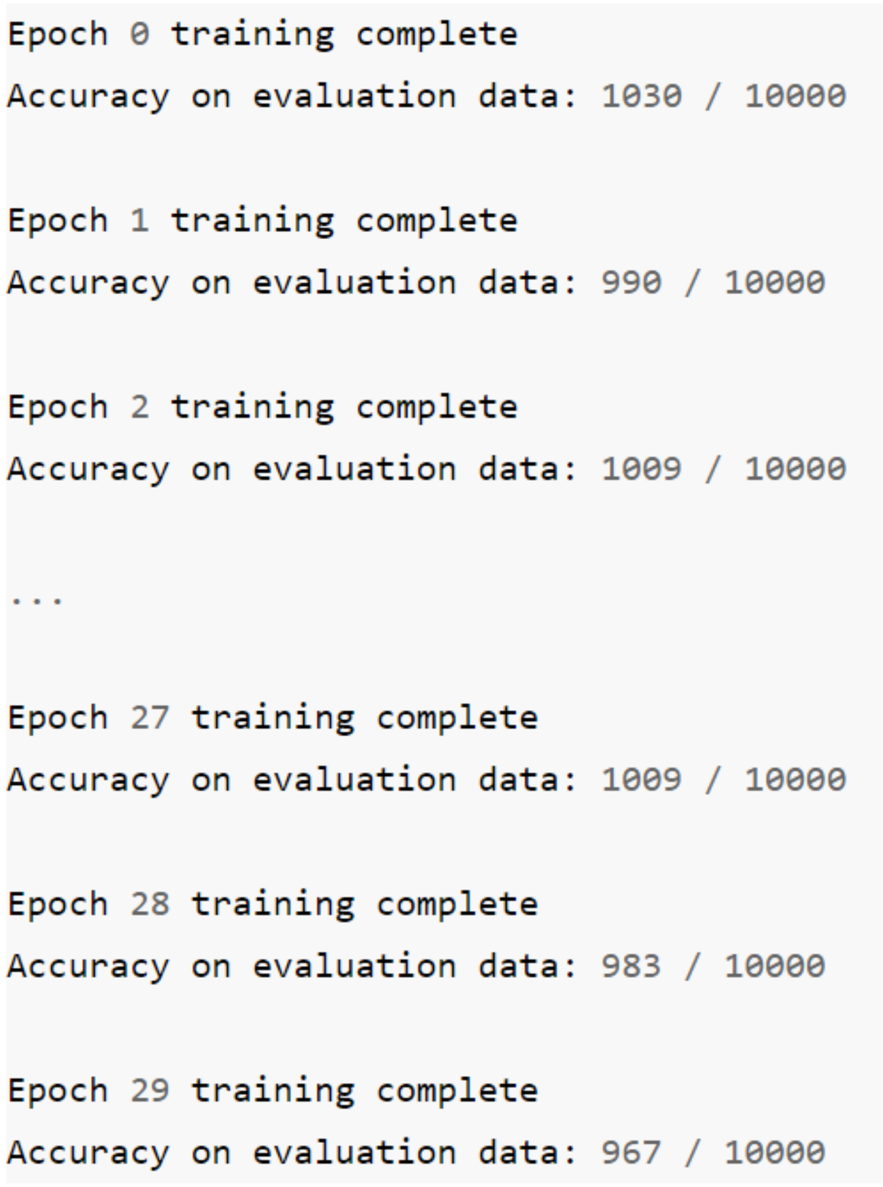
- Classification models: there is usually a sigmoid output at the end
  - Concern of saturation
  - **Truncated random normal**
- Generative models (we'll talk more about later in the semester): often no non-linear activation on the output
  - What if we needed to generate the point [5.1, 23.7]? How could we do that if our two output neurons are bounded to (0,1)?
  - No concern of saturation, don't truncate
- Using ReLU instead of LReLU?
  - Saturation possibility: Truncate





Hyper-parameter selection

- How do we choose parameters such as η and λ?
- In practice, this can be difficult
- Suppose we first started working on MNIST and chose 30 hidden neurons, mini-batch size of 10, 30 epochs, but $\eta = 10.0$ and $\lambda = 1000$
- No better than chance



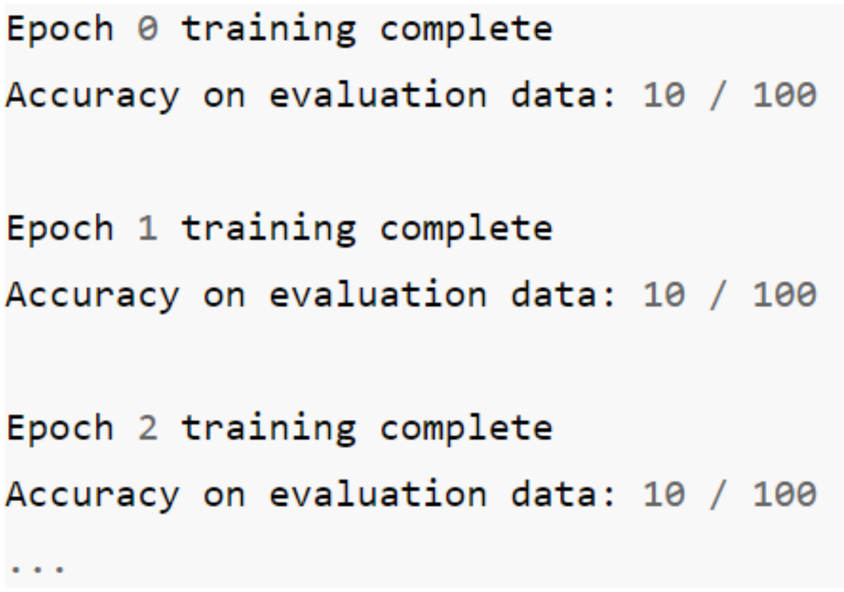
```
Epoch 0 training complete
Accuracy on evaluation data: 1030 / 10000

Epoch 1 training complete
Accuracy on evaluation data: 990 / 10000

Epoch 2 training complete
Accuracy on evaluation data: 1009 / 10000

...

Epoch 27 training complete
Accuracy on evaluation data: 1009 / 10000

Epoch 28 training complete
Accuracy on evaluation data: 983 / 10000

Epoch 29 training complete
Accuracy on evaluation data: 967 / 10000
```

- We know from our previous experiments that we should decrease the learning rate and the regularization parameter
  - But if this were the first time we saw this data, we wouldn’t know that
- Questions we could ask:
  - Do we need more or fewer hidden neurons?
  - Do we need more hidden layers?
  - Should we train for more epochs?
  - Are the mini-batches too small?
  - Should we use a different cost function?
  - Should we initialize weights differently?
  - Should we give up and take up beekeeping?

- First goal is to achieve results better than chance
  - Can be surprisingly difficult
- Suppose we’re looking at MNIST for the first time and get the results shown previously
  - Network completely fails
- Broad strategy: reduce the problem to get faster feedback

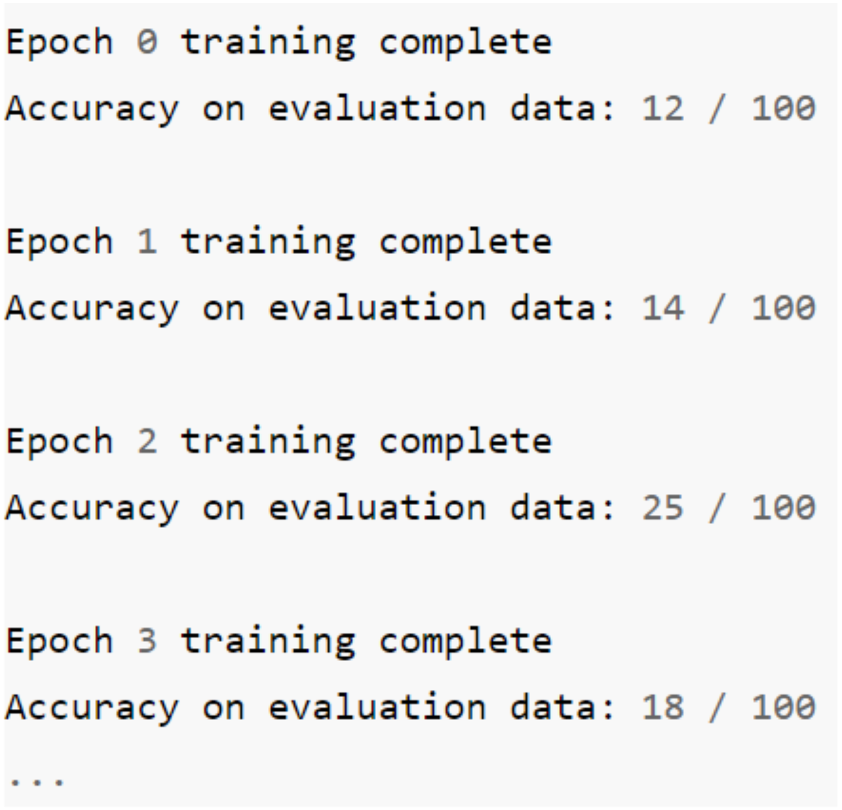
1. Focus on distinguishing 0s from 1s
   - An easier problem than all 10 digits
   - Reduces the training data, which speeds up training (enables faster experimentation)
2. Strip your network down to the simplest network that will do meaningful learning
   - E.g., start with no hidden layer if you think it will work better than chance
   - Training will be faster
3. Increase the frequency of monitoring
   - Monitor the validation accuracy more often, e.g. after every 1,000 training images instead of full 10,000

- Using full data takes ≈10 s per epoch
  - Not super long, but annoying if you want to test dozens or thousands of hyper-parameters
- Train on first 1,000 training images
- Estimate performance on 100 validation images
  - Gives a rough idea of how the network is performing
- Same parameters as before except no hidden layer
  - We still get pure noise but we get it a lot faster



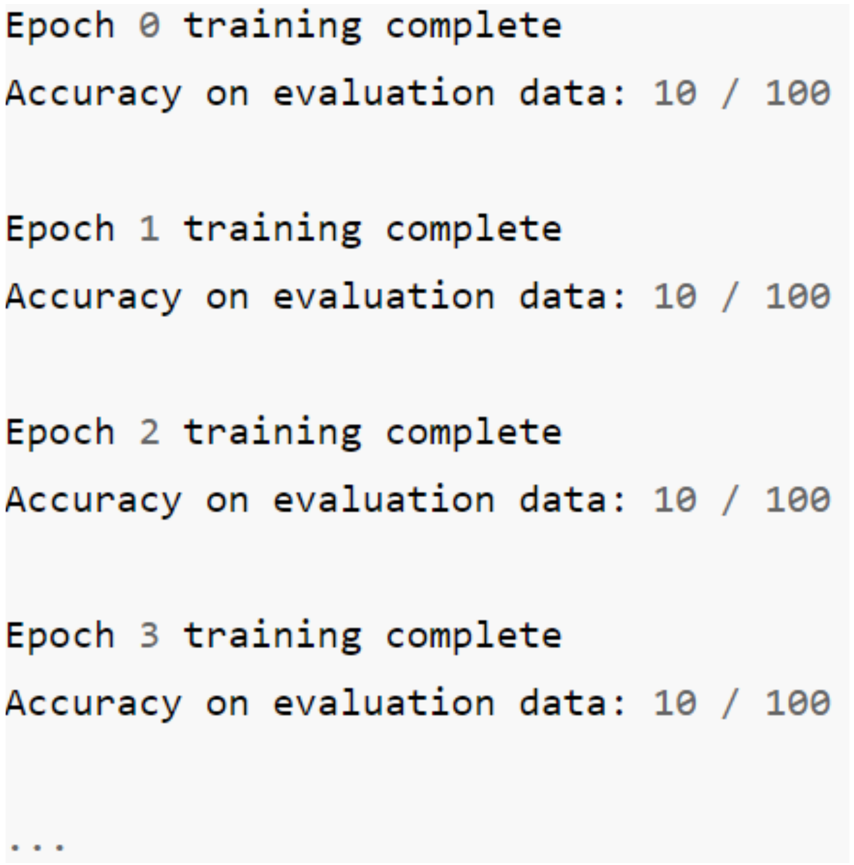
```
Epoch 0 training complete
Accuracy on evaluation data: 10 / 100

Epoch 1 training complete
Accuracy on evaluation data: 10 / 100

Epoch 2 training complete
Accuracy on evaluation data: 10 / 100
...
```

- Train on first 1,000 training images
- Estimate performance on 100 validation images
  - Gives a rough idea of how the network is performing
- Decrease λ from 1000 to 20
  - We start to see a little improvement
- Maybe the learning rate is too small?



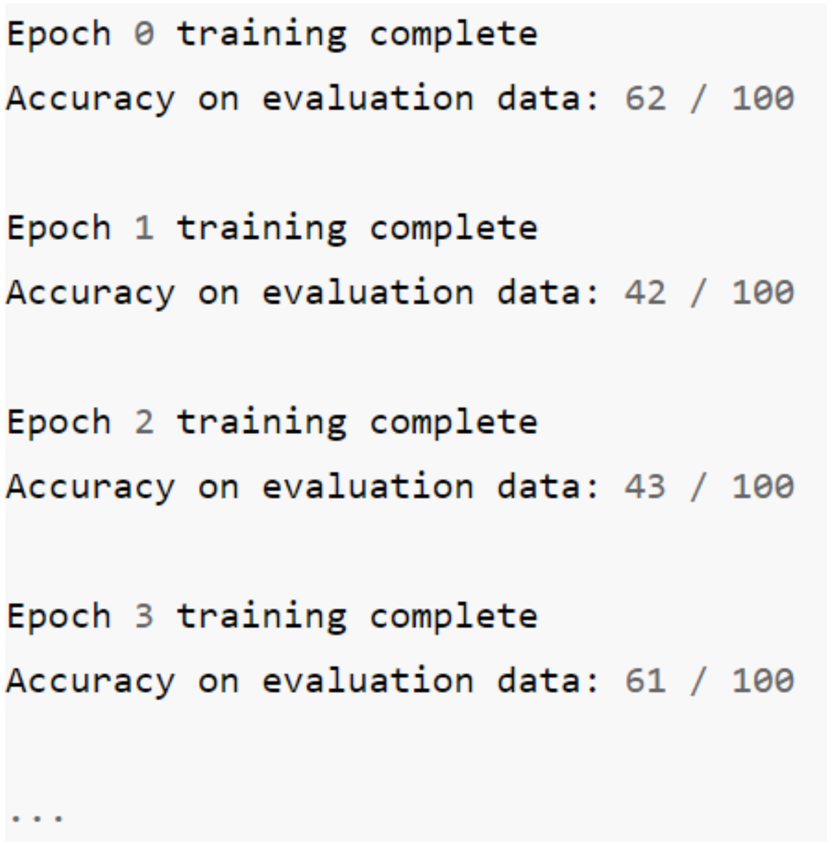
```
Epoch 0 training complete
Accuracy on evaluation data: 12 / 100

Epoch 1 training complete
Accuracy on evaluation data: 14 / 100

Epoch 2 training complete
Accuracy on evaluation data: 25 / 100

Epoch 3 training complete
Accuracy on evaluation data: 18 / 100
...
```

- Train on first 1,000 training images
- Estimate performance on 100 validation images
  - Gives a rough idea of how the network is performing
- Increase η from 10 to 100
  - Noise again
- Maybe the learning rate was too high?



```
Epoch 0 training complete
Accuracy on evaluation data: 10 / 100

Epoch 1 training complete
Accuracy on evaluation data: 10 / 100

Epoch 2 training complete
Accuracy on evaluation data: 10 / 100

Epoch 3 training complete
Accuracy on evaluation data: 10 / 100

...
```

- Train on first 1,000 training images
- Estimate performance on 100 validation images
  - Gives a rough idea of how the network is performing
- Decrease η to 1.0
  - This gives substantial improvement



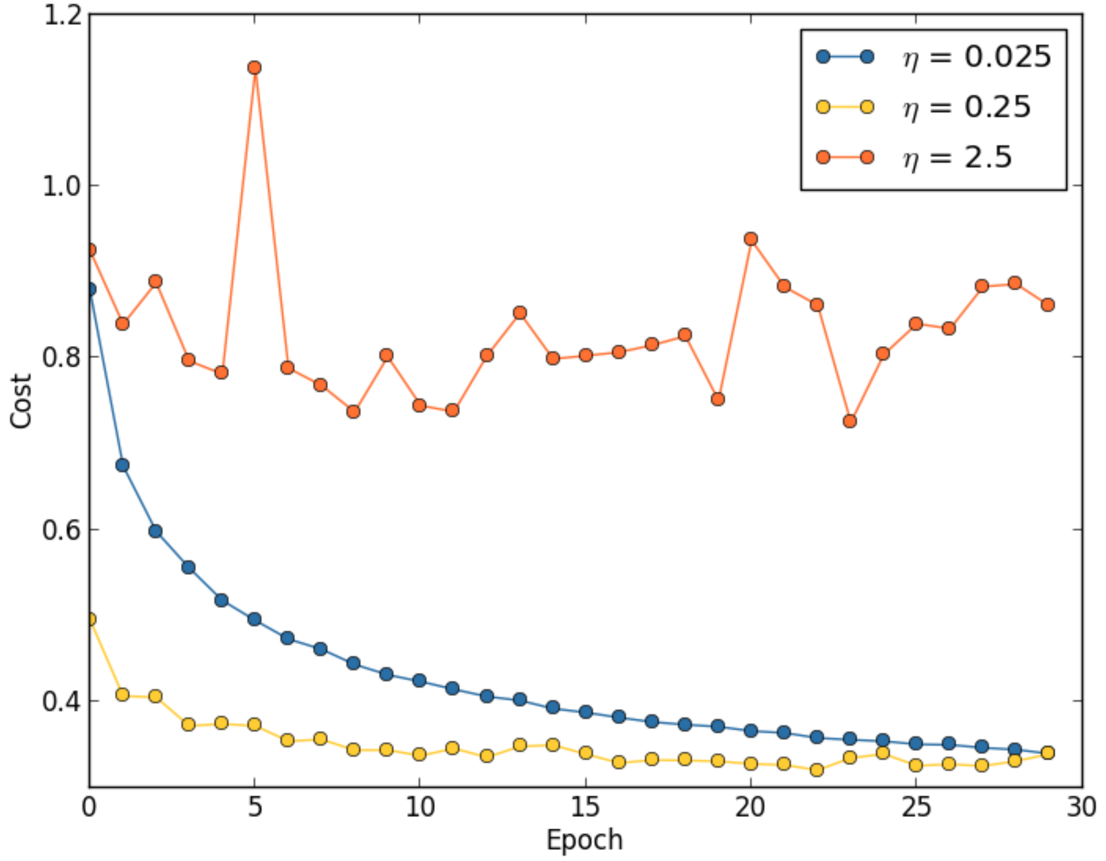
```
Epoch 0 training complete
Accuracy on evaluation data: 62 / 100

Epoch 1 training complete
Accuracy on evaluation data: 42 / 100

Epoch 2 training complete
Accuracy on evaluation data: 43 / 100

Epoch 3 training complete
Accuracy on evaluation data: 61 / 100

...
```

1. Once we're somewhat satisfied with η, move on to λ
2. Next, experiment with a more complex network
   - E.g. 10 hidden neurons
3. Adjust for η and λ again
4. Increase to 20 hidden neurons
5. And so forth

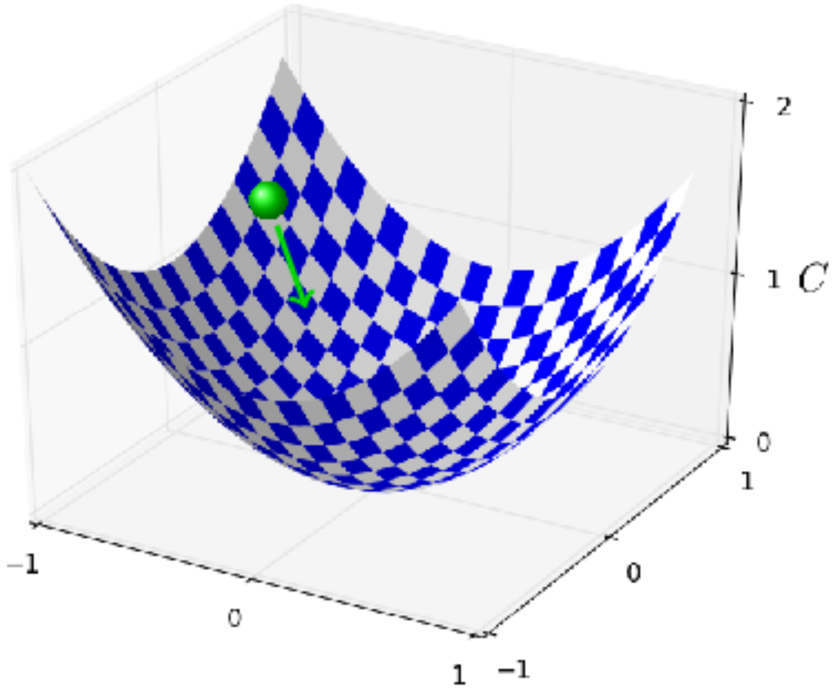
- Our initial approach to find hyper-parameters to improve learning may be a bit rosy
- It can be very frustrating to work with a network that is learning nothing
  - Hyper-parameters can be tweaked for days with no meaningful response
- So it’s very important that you get quick experimental feedback during the early stages
  - These simplifications may seem like a slowdown initially, but it’s a lot easier and faster to make improvements once you have something that’s working

- We will focus on learning rate η, regularization parameter λ, and the mini-batch size m
- However, the ideas apply to other parameters
  - Network architecture
  - Other regularization
  - Other hyper-parameters that we'll encounter

- Run three networks with different learning rates: 0.025, 0.25, and 2.5
  - 30 epochs, mini-batch size 10, $\lambda = 5.0$, 50,000 training images



- Oscillation likely due to overshoot
- But small η can take forever to converge
  - One approach: vary η (will cover later)
  - For now, we'll focus on finding a single good value for η



Estimate the threshold value for η at which the training cost begins decreasing, instead of oscillating or increasing

- Don’t need a very accurate estimate
- Start by estimating the order of magnitude of η
  - E.g., start with $\eta = 0.01$
  - If cost is decreasing for first few epochs, increase to 0.1, 1.0,… until the cost oscillates or increases during first few epochs
  - If cost is oscillating or increasing for first few epochs, decrease to 0.01, 0.001,… until the cost decreases during first few epochs
  - Can refine this but don’t go above the threshold
    - We want to train for many epochs without too much slowdown in learning
    - Good rule of thumb is to find rough value for the threshold and then divide by two

- Up to now, we’ve been using a constant learning rate
- It can be useful to vary the learning rate
  - During early training, the weights are likely very wrong
  - Using a large learning rate changes the weights quickly
  - Reducing the learning rate later allows for more fine-tuning

- How do we do this?
- One approach is to hold the learning rate constant until the validation accuracy gets worse
  - Then decrease the learning rate by some amount, e.g. a factor of two or ten
  - Repeat this many times until the learning rate is a certain factor lower than the initial value (e.g. 1024 or 1000)
- Variable learning schedules can improve performance but lead to more parameter choices
  - For first experiments, generally choosing a single value for the learning rate is best
  - Later, a learning schedule can be used to get best performance

- Nielsen suggests starting with no regularization and tuning η first
- We can then select a good value for λ
- Nielsen suggests starting with $\lambda = 1.0$ and then increasing or decreasing by factors of 10 as needed
- Once a good order of magnitude is found, we can fine tune λ
    - Once λ is selected, η should be re-optimized

- Hand-optimization of hyper-parameters gives a good feel for how neural networks behave
- But it would be nice to automate the process
- A common approach is grid search
  - Searches through a grid in hyper-parameter space
- Other approaches exist

Choosing Lambda

- Opposite of learning rate choice:
  - Start with a low regularization, gradually increase it
  - .0001 might be a good start, then .001, .01, etc…
- Cross-validation
- What to look for? Look at your output changing with different inputs
  - Is it smooth?
- There are fancy theoretical techniques like Bayesian Optimization[1] to choose coefficient, but they are almost never used in practice

[1] https://en.wikipedia.org/wiki/Bayesian_optimization

- Another challenge is that there are many papers that give (sometimes contradictory) recommendations for how to set hyper-parameters
- There are some good resources that distill this information
  - Bengio, “Practical recommendations for gradient-based training of deep architectures”
  - LeCun et al., “Efficient BackProp”
  - Montavon et al., “Neural Networks: Tricks of the Trade”
- Main takeaway is that hyper-parameter optimization has not been solved
  - There’s always another trick to try
  - Goal should be to develop a workflow that does a pretty good job at optimization while leaving some flexibility to try more detailed optimizations if necessary

- Nielsen book, chapter 3
- Goodfellow et al., section 8.4-8.5
- Bengio: “Practical Recommendations for Gradient-Based Training of Deep Architectures”
- Bergstra and Bengio, “Random search for hyper-parameter optimization”