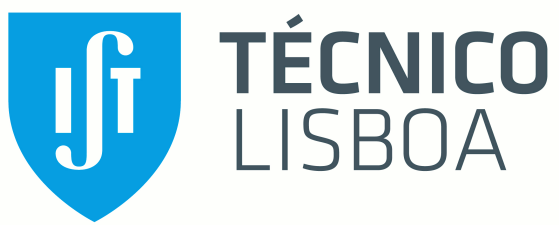
Lecture 5: Neural Networks II

André Martins, Francisco Melo, Mário Figueiredo



Deep Learning Course, Winter 2022-2023

Today's Roadmap

Last lecture was about neural networks:

- From perceptron to multi-layer perceptron
- Feed-forward neural networks
- Activation funcions: sigmoid, tanh, relu, ...
- Activation maps: softmax, sparsemax, ...
- Non-convex optimization and local minima
- Universal approximation theorem
- Gradient backpropagation

Today: autodiff, regularization, tricks of the trade.

Outline

Outline

Recap: Forward Propagation

Now assume $L \geq 1$ hidden layers:

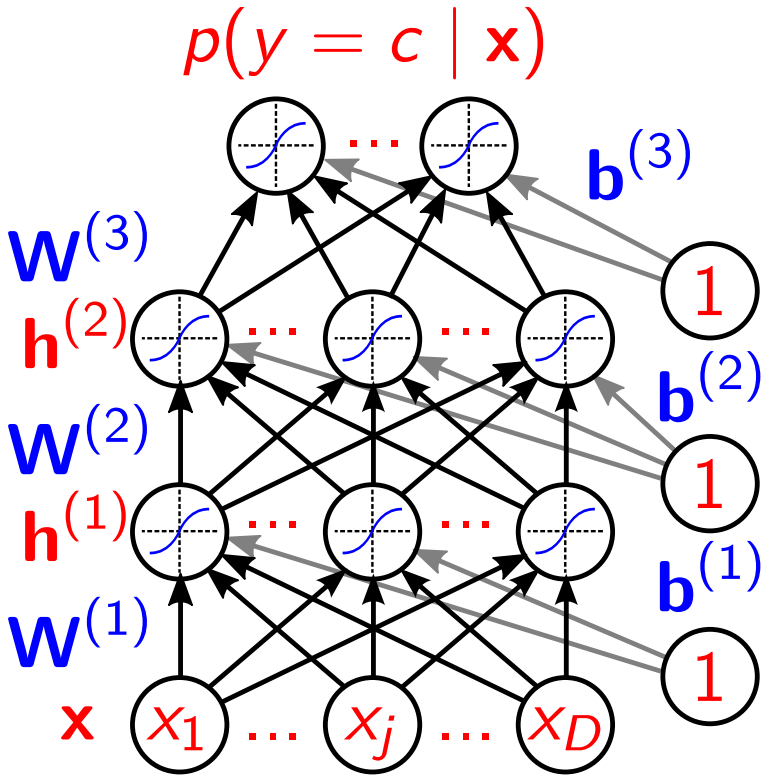
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$, for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)}(\boldsymbol{x}) + \boldsymbol{b}^{(1)},$$

  with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$ and $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$.

- **Hidden layer activation:**

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})).$$



- **Output layer activation:**

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{o}(\boldsymbol{z}^{(L+1)}(\boldsymbol{x})) = \boldsymbol{o}(\boldsymbol{W}^{(L+1)} \boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L+1)}).$$

Recap: Gradient Backpropagation

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -(1_y - \boldsymbol{f}(\boldsymbol{x}))$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\begin{aligned} \nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)\ \boldsymbol{h}^{(\ell-1)\top} \\ \nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \end{aligned}$$

Compute gradient of previous hidden layer:

$$\nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{W}^{(\ell)\top} \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

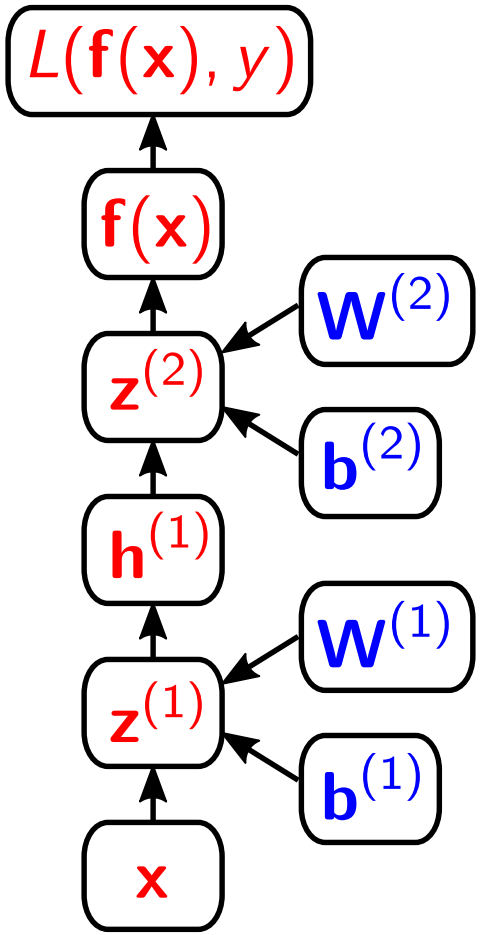
Compute gradient of previous hidden layer (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell-1)})$$

end for

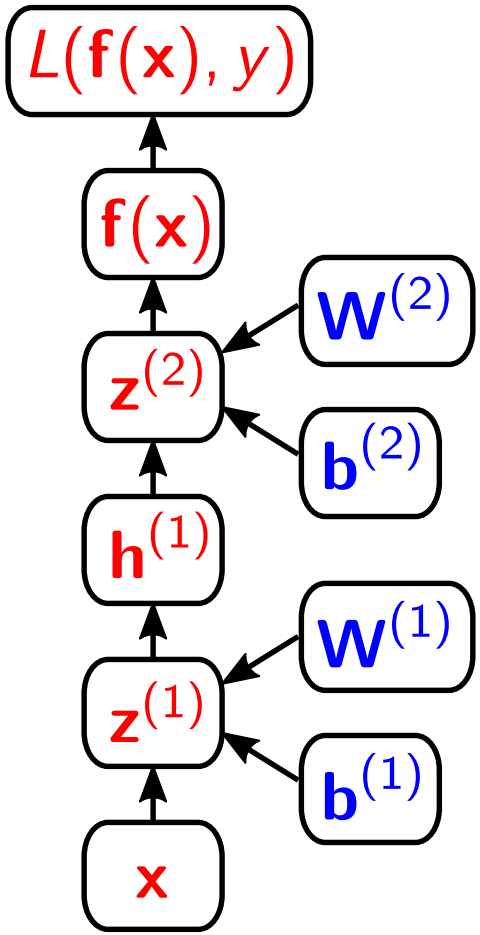
Computation Graph

- Forward propagation can be represented as a DAG (directed acyclic graph).



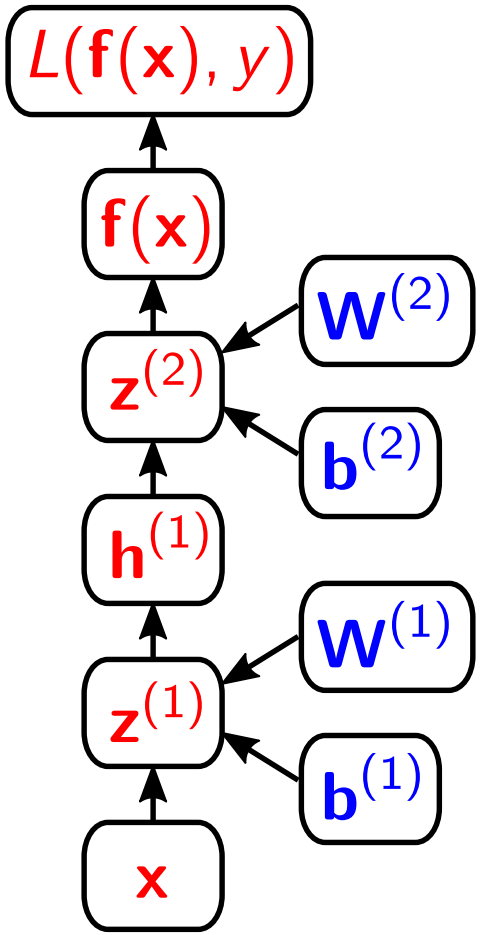
Computation Graph

- Forward propagation can be represented as a DAG (directed acyclic graph).
- Allows implementing forward propagation in a modular way.



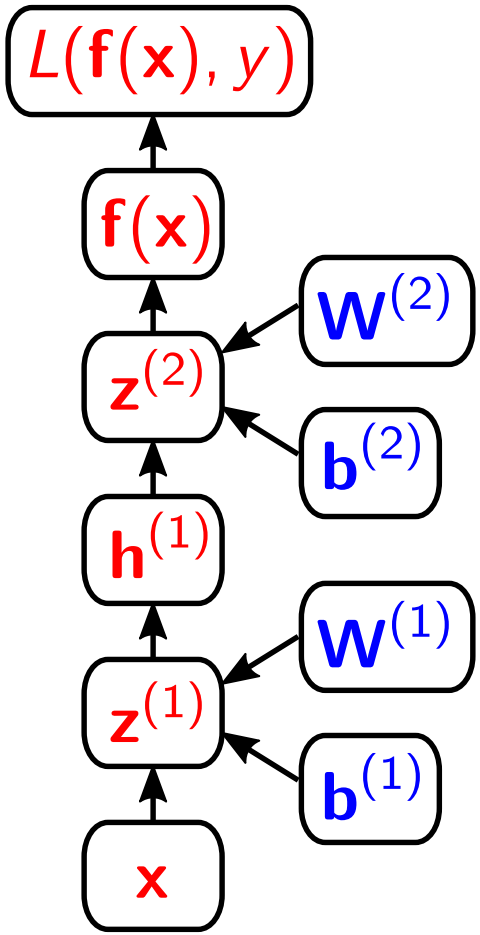
Computation Graph

- Forward propagation can be represented as a DAG (directed acyclic graph).
- Allows implementing forward propagation in a modular way.
- Each box can be an object with a `fprop` method, which computes the output of the box given its inputs.



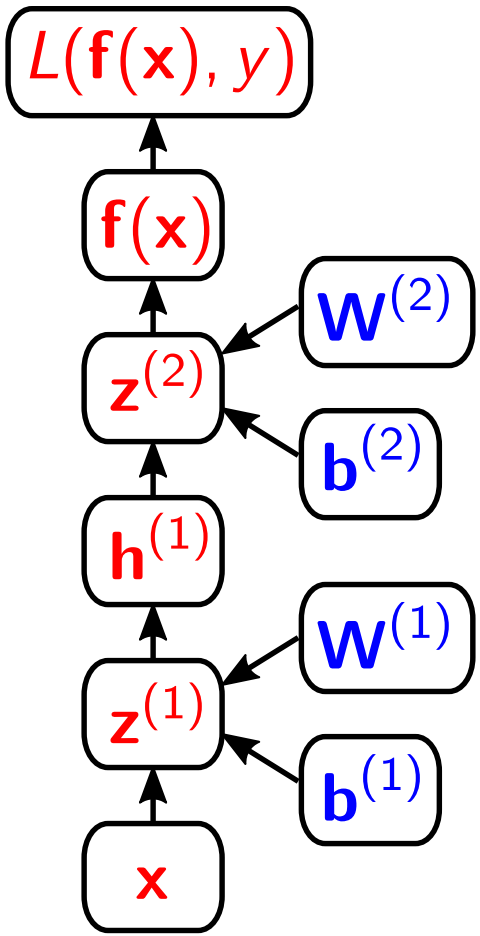
Computation Graph

- Forward propagation can be represented as a DAG (directed acyclic graph).
- Allows implementing forward propagation in a modular way.
- Each box can be an object with a `fprop` method, which computes the output of the box given its inputs.
- Calling the `fprop` method of each box in the right order yields forward propagation.



Automatic Differentiation (Autodiff)

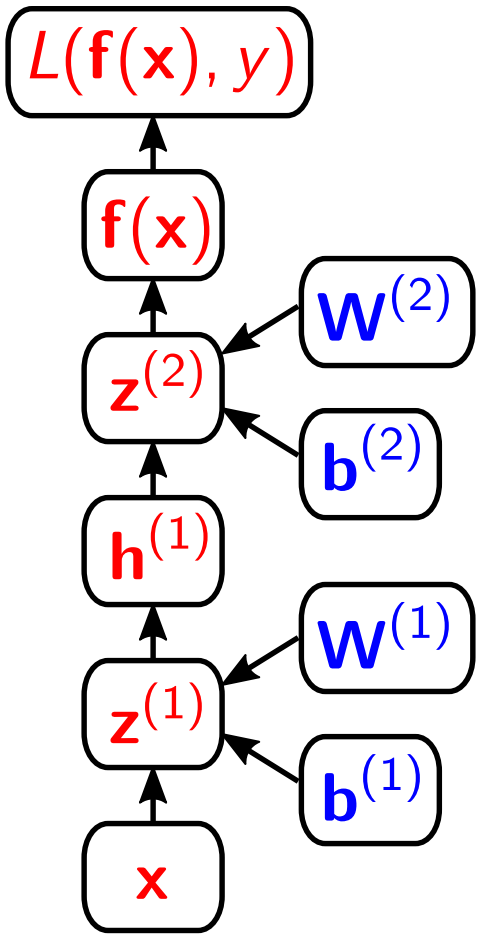
- Backpropagation is also implementable in a modular way.



Automatic Differentiation (Autodiff)

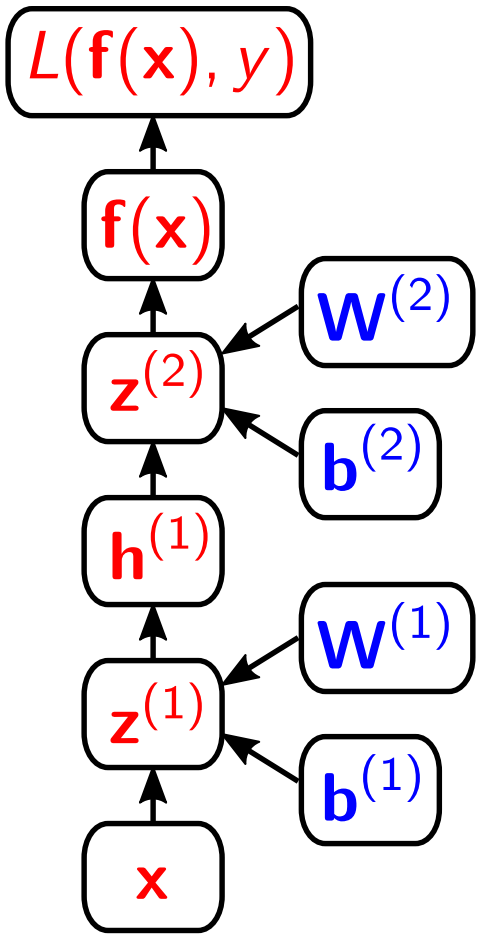
- Backpropagation is also implementable in a modular way.

- Each box should have a `bprop` method, which computes the loss gradient w.r.t. its parents, given the loss gradient w.r.t. to the output.



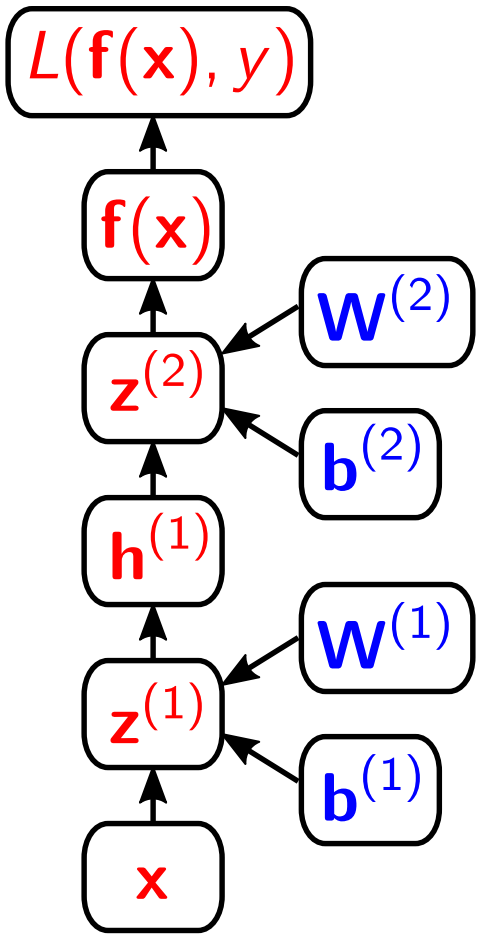
Automatic Differentiation (Autodiff)

- Backpropagation is also implementable in a modular way.
- Each box should have a `bprop` method, which computes the loss gradient w.r.t. its parents, given the loss gradient w.r.t. to the output.
- Can make use of cached computation done during the `fprop` method



Automatic Differentiation (Autodiff)

- Backpropagation is also implementable in a modular way.
- Each box should have a `bprop` method, which computes the loss gradient w.r.t. its parents, given the loss gradient w.r.t. to the output.
- Can make use of cached computation done during the `fprop` method
- Calling the `bprop` method in reverse order yields backpropagation
  (only needs to reach the parameters)



Several Autodiff Strategies

Symbol-to-number differentiation (Caffe, Torch, Pytorch, Dynet, ...)

- Take a computational graph and numerical inputs, returns a set of numerical values describing the gradient at those input values.
- Advantage: simpler to implement and debug.
- Disadvantage: only works for first-order derivatives.

Several Autodiff Strategies

Symbol-to-number differentiation (Caffe, Torch, Pytorch, Dynet, ...)

- Take a computational graph and numerical inputs, returns a set of numerical values describing the gradient at those input values.
- Advantage: simpler to implement and debug.
- Disadvantage: only works for first-order derivatives.

Symbol-to-symbol differentiation (Theano, Tensorflow, ...)

- Take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives (i.e. the derivatives are just another computational graph)
- Advantage: generalizes automatically to higher-order derivatives
- Disadvantage: harder to implement and to debug

Many Software Toolkits for Neural Networks

- Theano
- Tensorflow
- Torch, Pytorch
- MXNet
- Keras
- Caffe
- DyNet
- ...

All implement automatic differentiation.

Some Theano Code (Logistic Regression)

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400                                   # training sample size
feats = 784                               # number of input variables

# generate a dataset: D = (input_values, target_class)
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
training_steps = 10000

# Declare Theano symbolic variables
x = T.dmatrix("x")
y = T.dvector("y")

# initialize the weight vector w randomly
#
# this and the following bias variable b
# are shared so they keep their values
# between training iterations (updates)
w = theano.shared(rng.randn(feats), name="w")

# initialize the bias term
b = theano.shared(0., name="b")

print("Initial model:")
print(w.get_value())
print(b.get_value())

# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))   # Probability that target = 1
prediction = p_1 > 0.5                    # The prediction thresholded
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1) # Cross-entropy loss function
cost = xent.mean() + 0.01 * (w ** 2).sum()# The cost to minimize
gw, gb = T.grad(cost, [w, b])             # Compute the gradient of the cost
                                          # w.r.t weight vector w and
                                          # bias term b
                                          # (we shall return to this in a
                                          # following section of this tutorial)

# Compile
train = theano.function(
          inputs=[x,y],
```

Some Code in Tensorflow (Linear Regression)

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but TensorFlow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.  We will 'run' this first.
init = tf.global_variables_initializer()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))

# Learns best fit is W: [0.1], b: [0.3]
```

Some Code in Keras (Multi-Layer Perceptron)

Multilayer Perceptron (MLP) for multi-class softmax classification:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, y_train,
          nb_epoch=20,
          batch_size=16)
score = model.evaluate(X_test, y_test, batch_size=16)
```

Some Code in Pytorch (Multi-Layer Perceptron)

```
# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

model = NeuralNet(input_size, hidden_size, num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Move tensors to the configured device
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

Reminder: Key Ingredients of SGD

In sum, we need the following ingredients:

- The loss function $L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$ ✓
- A procedure for computing the gradients $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$ ✓
- The regularizer $\Omega(\boldsymbol{\theta})$ and its gradient: next!

Outline

Regularization

Recall that we're minimizing the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{n=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$$

It remains to define the regularizer and its gradient

Regularization

Recall that we're minimizing the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda \Omega(\boldsymbol{\theta}) + \frac{1}{N} \sum_{n=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

It remains to define the regularizer and its gradient

We will study:

- ℓ_2 regularization
- ℓ_1 regularization
- dropout regularization

ℓ_2 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|^2$$

ℓ_2 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|^2$$

- Equivalent to a Gaussian prior on the weights

ℓ_2 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \|\boldsymbol{W}^{(\ell)}\|^2$$

- Equivalent to a Gaussian prior on the weights

- Gradient of this regularizer is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \boldsymbol{W}^{(\ell)}$

ℓ_2 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell=1}^{L+1} \| \boldsymbol{W}^{(\ell)} \|^2$$

- Equivalent to a Gaussian prior on the weights

- Gradient of this regularizer is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \boldsymbol{W}^{(\ell)}$

- Weight decay effect (as seen in the previous lecture)

$$\begin{aligned} \boldsymbol{W}^{(\ell)} &\leftarrow \boldsymbol{W}^{(\ell)} - \eta \nabla_{\boldsymbol{W}^{(\ell)}} \mathcal{L}_i(\boldsymbol{\theta}) \\ &= \boldsymbol{W}^{(\ell)} - \eta (\lambda \nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{W}^{(\ell)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)) \\ &= \underbrace{(1 - \eta\lambda)}_{<1} \boldsymbol{W}^{(\ell)} - \eta \nabla_{\boldsymbol{W}^{(\ell)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i) \end{aligned}$$

ℓ_1 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

ℓ_1 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

- Equivalent to Laplacian prior on the weights

ℓ_1 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

- Equivalent to Laplacian prior on the weights

- Gradient is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\boldsymbol{W}^{(\ell)})$

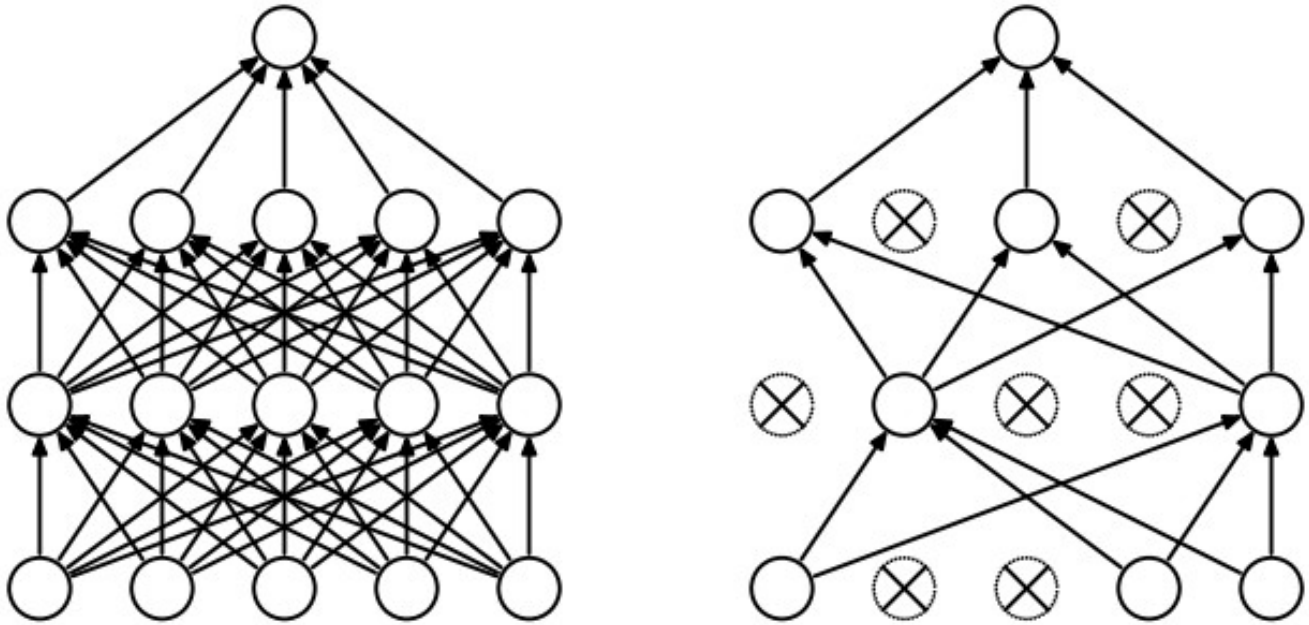
ℓ_1 Regularization

- The biases $\boldsymbol{b}^{(1)}, ..., \boldsymbol{b}^{(L+1)}$ are not regularized; only the weights:

$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\boldsymbol{W}^{(\ell)}\|_1 = \sum_{\ell} \sum_{ij} |W_{ij}^{(\ell)}|$$

- Equivalent to Laplacian prior on the weights
- Gradient is: $\nabla_{\boldsymbol{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\boldsymbol{W}^{(\ell)})$
- Promotes sparsity of the weights

Dropout Regularization (Srivastava et al., 2014)



(a) Standard Neural Net

(b) After applying dropout.

During training, remove some hidden units, chosen at random

Dropout Regularization (Srivastava et al., 2014)

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.5$)

Dropout Regularization (Srivastava et al., 2014)

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.5$)
- This prevents hidden units to co-adapt to other units, forcing them to be more generally useful

Dropout Regularization (Srivastava et al., 2014)

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.5$)
- This prevents hidden units to co-adapt to other units, forcing them to be more generally useful
- At test time, keep all units, with the outputs multiplied by $1 - p$

Dropout Regularization (Srivastava et al., 2014)

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.5$)
- This prevents hidden units to co-adapt to other units, forcing them to be more generally useful
- At test time, keep all units, with the outputs multiplied by $1 - p$
- Shown to be a form of adaptive regularization (Wager et al., 2013)

Dropout Regularization (Srivastava et al., 2014)

- Each hidden unit output is set to 0 with probability p (e.g. $p = 0.5$)
- This prevents hidden units to co-adapt to other units, forcing them to be more generally useful
- At test time, keep all units, with the outputs multiplied by $1 - p$
- Shown to be a form of adaptive regularization (Wager et al., 2013)
- Many software packages implement another variant, inverted dropout, where at training time the output of the units that were not dropped is divided by $1 - p$ and requires no change at test time

Implementation of Dropout

- Usually implemented using random binary masks

- The hidden layer activations become

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})) \odot \boldsymbol{m}^{(\ell)}$$

Implementation of Dropout

- Usually implemented using random binary masks
- The hidden layer activations become

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})) \odot \boldsymbol{m}^{(\ell)}$$

- Beats regular backpropagation on many datasets (Hinton et al., 2012)

Implementation of Dropout

- Usually implemented using random binary masks
- The hidden layer activations become

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})) \odot \boldsymbol{m}^{(\ell)}$$

- Beats regular backpropagation on many datasets (Hinton et al., 2012)
- Other variants, e.g. DropConnect (Wan et al., 2013), Stochastic Pooling (Zeiler and Fergus, 2013)

Backpropagation with Dropout

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -(1_y - \boldsymbol{f}(\boldsymbol{x}))$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \underbrace{\boldsymbol{h}^{(\ell-1)\top}}_{\text{includes } \boldsymbol{m}^{(\ell-1)}}$$

$$\nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below:

$$\nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{W}^{(\ell)\top} \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell-1)}) \odot \boldsymbol{m}^{(\ell-1)}$$

end for

Outline

Initialization

- Biases: initialize at zero

Initialization

- Biases: initialize at zero
- Weights:
  - ✓ Cannot initialize to zero with tanh activation (gradients would also be all zero and we would be at saddle point)
  - ✓ Cannot initialize the weights to the same value (need to break the symmetry)
  - ✓ Random initialization (Gaussian, uniform), sampling around 0 to break symmetry
  - ✓ For ReLU activations, the mean should be a small positive number
  - ✓ Variance cannot be too high, otherwise all neuron activations will be saturated

"Glorot Initialization"

- Recipe from Glorot and Bengio (2010):

$$\boldsymbol{W}_{i,j}^{(\ell)} \sim U[-t, t], \text{ with } t = \frac{\sqrt{6}}{\sqrt{K^{(\ell)} + K^{(\ell-1)}}}$$

- Works well in practice with tanh and sigmoid activations

Training, Validation, and Test Sets

Split datasets in training, validation, and test partitions.

- Training set: serves to train the model
- Validation set: used to tune hyperparameters (learning rate, number of hidden units, regularization coefficient, dropout probability, best epoch, etc.)
- Test set: used to estimate the generalization performance

Hyperparameter Tuning: Grid Search, Random Search

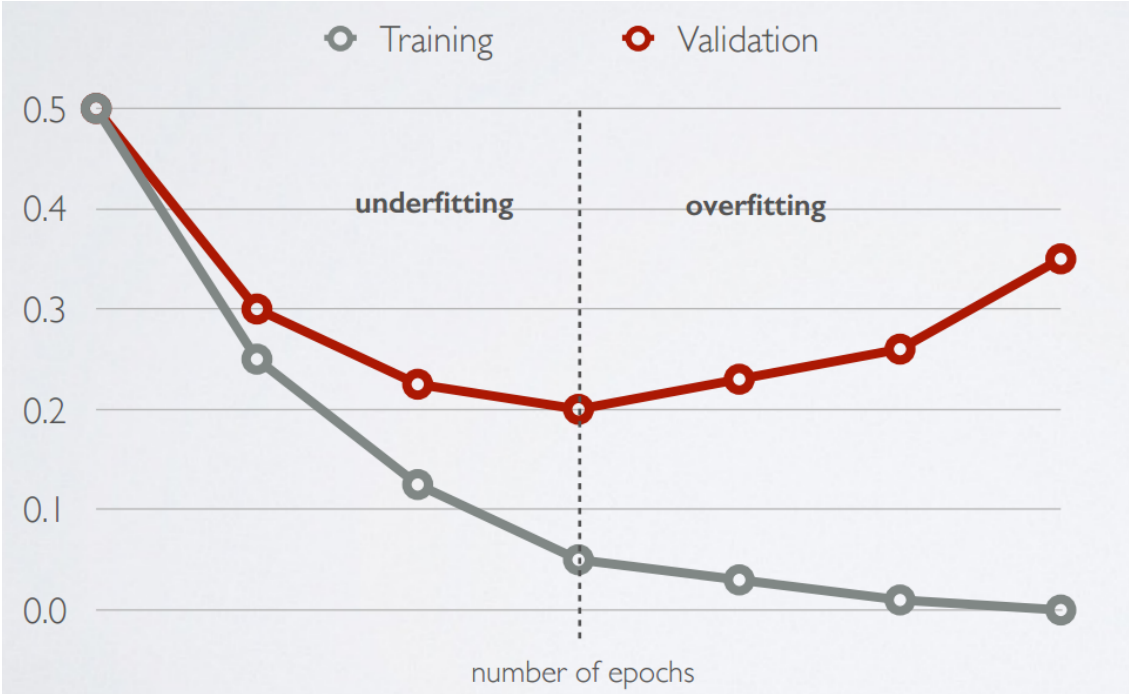
Search for the best configuration of the hyperparameters:

- Grid search: specify a set of values to test for each hyperparameter, and try all configurations of these values
- Random search: specify a distribution over the values of each hyper-parameter (e.g. uniform in some range) and sample independently each hyper-parameter to get configurations
- Bayesian optimization (Snoek et al., 2012)

We can always go back and fine-tune the grid/distributions if necessary

Early Stopping

- To select the number of epochs, stop training when validation error increases (with some look ahead)
- One common strategy (with SGD) is to halve the learning rate for every epoch where the validation error increases

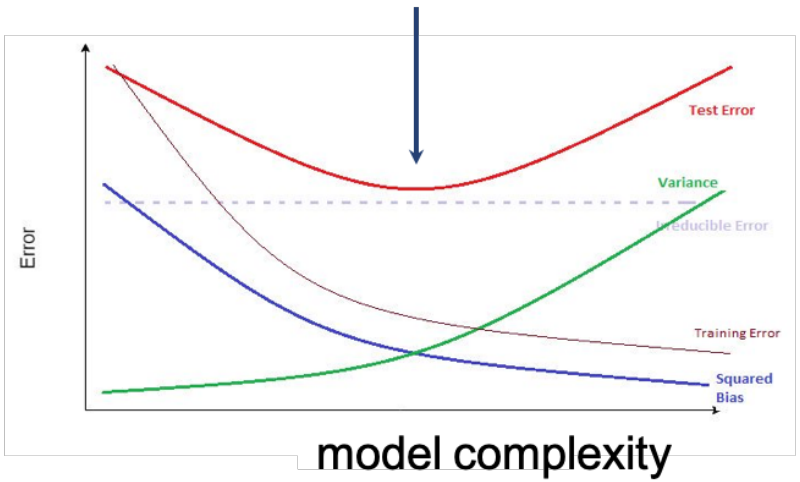


(Image credit: Hugo Larochelle)

Cross Validation

Model selection

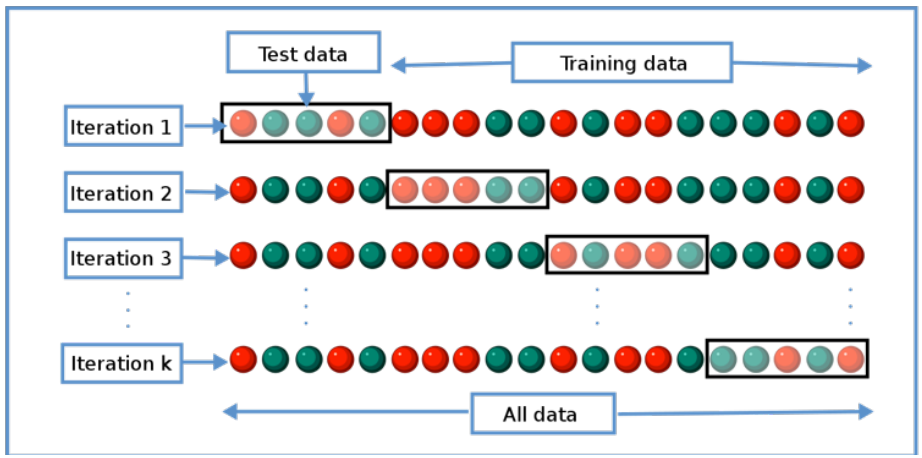
finding the sweet spot (F)



validation set:

- ✓ split data into train and validation susbsets
- ✓ train on the training subset
- ✓ test on the validation subset

cross validation (k-fold): repeat k times; average

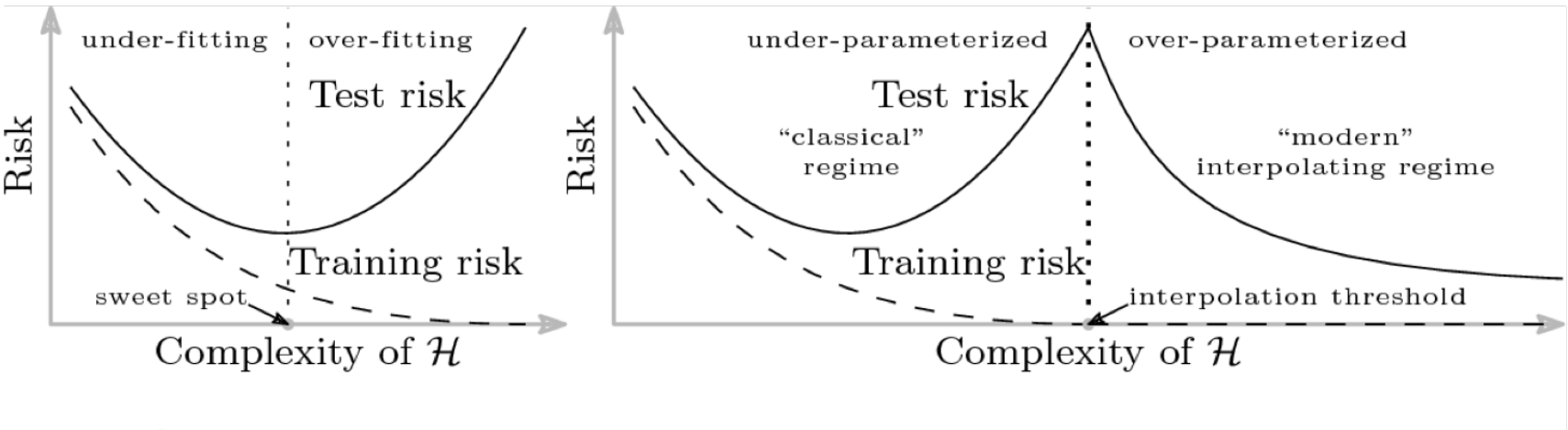


Over-parametrization

The new regime…

modern deep networks have "too many" parameters: they should overfit, …

…yet, they usually don't. Why? Ongoing research.



(illustration by Mikhail Belkin)

Tricks of the Trade

- Normalization of the data
- Decaying the learning rate
- Mini-batches
- Adaptive learning rates
- Gradient checking
- Debugging on a small dataset

Normalization of the Data

- For each input dimension: subtract the training set mean and divide by the training set standard deviation
- It makes each input dimension have zero mean, unit variance
- It can speed up training (in number of epochs)
- Doesn't work for sparse inputs (destroys sparsity)

Decaying the Learning Rate

In SGD, as we get closer to a local minimum, it makes sense to take smaller update steps (to avoid diverging)

- Start with a large learning rate (say 0.1)
- Keep it fixed while validation error keeps improving
- Divide by 2 and go back to the previous step

Mini-Batches

- Instead of updating after a single example, can aggregate a mini-batch of examples (e.g. 50–200 examples) and compute the averaged gradient for the entire mini-batch
- Less noisy than standard SGD
- Can leverage matrix-matrix computations (or tensor computations)
- Large computational speed-ups in GPUs: computation is trivially parallelizable accross the mini-batch and we can exhaust the GPU memory

Adaptive Learning Rates

Instead of using the same step size for all parameters, have one learning rate per parameter

- Adagrad (Duchi et al., 2011): learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\eta^{(t)} = \eta^{(t-1)} + (\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y))^2, \qquad \bar{\nabla}_{\boldsymbol{\theta}}^{(t)} = \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y)}{\sqrt{\eta^{(t)} + \epsilon}}$$

- RMSprop (Tieleman and Hinton, 2012): instead of cumulative sum, use exponential moving average

$$\eta^{(t)} = \beta\eta^{(t-1)} + (1-\beta)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y))^2, \qquad \bar{\nabla}_{\boldsymbol{\theta}}^{(t)} = \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y)}{\sqrt{\eta^{(t)} + \epsilon}}$$

- Adam (Kingma and Ba, 2014): combine RMSProp with momentum

Gradient Checking

- If the training loss is not decreasing even with a very small learning rate, there's likely a bug in the gradient computation

- To debug your implementation of `fprop`/`bprop`, compute the "numeric gradient," a finite difference approximation of the true gradient:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Debugging on a Small Dataset

- Extract a small subset of your training set (e.g. 50 examples)
- Monitor your training loss in this set
- You should be able to overfit in this small training set
- If not, see if some units are saturated from the very first iterations (if they are, reduce the initialization variance or properly normalize your inputs)
- If the training error is bouncing up and down, decrease the learning rate

Outline

Conclusions

- Multi-layer perceptrons are universal function approximators
- However, they need to be trained
- Stochastic gradient descent is an effective training algorithm
- This is possible with the gradient backpropagation algorithm (an application of the chain rule of derivatives)
- Most current software packages represent a computation graph and implement automatic differentiation
- Dropout regularization is effective to avoid overfitting

Thank you!

Questions?

References I

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, volume 9, pages 249–256.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In *Proc. of International Conference on Learning Representations*.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

Tieleman, T. and Hinton, G. (2012). Rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2).

Wager, S., Wang, S., and Liang, P. S. (2013). Dropout training as adaptive regularization. In *Advances in neural information processing systems*, pages 351–359.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proc. of the International Conference on Machine Learning*, pages 1058–1066.

Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*.