1. **[2pts]** *Recall that multilayer perceptrons are universal for the set of functions mapping binary-valued input vectors to binary valued outputs.*

   (a) **[1pt]** *What do we mean by universal?*

   For any function in the set, there is some network that computes it. (We also accepted "approximates" instead of "computes".)

   (b) **[1pt]** *If multilayer perceptrons are universal, why do we still consider other architectures?*

   The function might not be *compactly* representable, i.e. it might require exponentially many units. This means it may be prohibitively expensive to compute, and training such a large architecture wouldn't generalize to new data.

   Marking: Full credit for mentioning efficiency, generalization, or interpretability.

   Mean: 1.7/2

2. **[1pt]** *Give an example of a data augmentation technique that would be useful for classifying images of cats vs. dogs, but not for classifying handwritten digits. Briefly explain your answer.*

   Flipping the image horizontally; doing this to a dog image would result in a plausible dog image, but not so for an image of a digit.

   Mean: 0.85/1

3. **[1pt]** *Suppose we have a grayscale image represented as an array, where larger values denote lighter pixels. What is the effect when we convolve it with the following kernel?*

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & -4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

   It will blur the image and reverse light and dark.

   Marking: half a point for blurring, and half a point for reversing light and dark. The statement of the question was probably too ambiguous in terms of how the image was represented. We accepted a variety of alternative answers for the second half-point, such as:

   - the image values will get clipped to 0
   - it detects dark spots

- the image saturation increases

Mean: 0.4/1

4. **[2pts]** *The learning rate is an important parameter for gradient descent.*

   (a) **[1pt]** *Briefly describe something that can go wrong if we choose too high a learning rate for* full batch *gradient descent.*

   The training could get unstable, and the weights would diverge. Or the weights could oscillate in high curvature directions.

   (b) **[1pt]** *Briefly describe something that can go wrong if we choose too high a learning rate for* stochastic *gradient descent, but is not a problem in the full batch setting.*

   The sampled gradients cause the parameters to fluctuate, thereby not getting close to the optimum.

   Mean: 1.8/2

5. **[2pts]** *Suppose we are training an RNN language model using teacher forcing (the method you implemented in Assignment 3).*

   (a) **[1pt]** *What are the inputs to the network at training time?*

   The most recent word in the training sentence.

   (b) **[1pt]** *What are the inputs to the network at test time?*

   In each time step, a word is sampled from the distribution computed by the network's output units. This word is fed back as the input in the next time step.

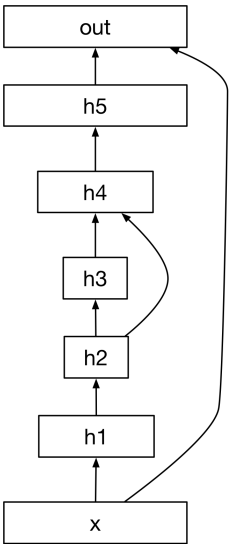
   Marking: One point off if the answer is correct except for reversing training and test time. Half a point off in either part if the answer is ambiguous (e.g. simply "ground truth" or "training data"), and/or doesn't talk about sequences, time steps, etc. Full marks were awarded in part (b) even if the answer doesn't expclicitly say the words were sampled from the softmax. (This is an important part of the answer, but there were too many borderline cases, which made it too hard to assign credit.)

   Mean: 1.5/2

6. **[1pt]** *Here is a modified version of code from Programming Assignment 2. The methods* `downconv1`, `rfconv`, *etc. implement convolution layers. Add edges to the diagram to represent the network architecture this implements. You don't need to justify your answer.*

```
class MyNet(nn.Module):
    ...

    def forward(self, x):
        self.h1 = self.downconv1(x)
        self.h2 = self.downconv2(self.h1)
        self.h3 = self.rfconv(self.h2)
        self.h4 = self.upconv1(torch.cat([self.h3, self.h2], 1))
        self.h5 = self.upconv2(self.h4)
        self.out = self.finalconv(torch.cat([self.h5, x], 1))
        return self.out
```



Marking: Half a point off for each mistake.

Mean: 0.94/1

7. **[3pts]** *Recall the (multivariate) linear regression model:*

$$y = \mathbf{w}^\top \mathbf{x} + b$$
$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

Your job is to implement full batch gradient descent in NumPy. In particular, suppose we are given an $N \times D$ NumPy array `X` *representing all the training inputs, and an*

N dimensional NumPy vector `t` *representing the targets, where N is the number of data points, and D is the input dimension. The weights are represented with a D-dimensional NumPy vector* `w`*, and the biases are represented with a scalar* `b`*. The learning rate is given as* `alpha`*.*

Write NumPy code which implements one iteration of batch gradient descent. It should be vectorized, i.e. it should not involve a `for`-loop*. You don't need to show your work, but doing so may help you get partial credit.*

```
y = np.dot(X, w) + b
y_bar = y - t
w_bar = np.dot(X.T, y_bar) / N
w -= alpha * w_bar
b_bar = np.mean(y_bar)
b -= alpha * b_bar
```

Marking: Half a point off for each mistake. Individual mistakes made multiple times (e.g. missing transposes) are only counted once. One point off if the bias update is omitted. No penalty for not dividing by N (since we didn't specify this). 1.5 points off if given in terms of math rather than code. Half a point off for not computing y.

Mean: 2.1/3

8. **[2pts]** *Suppose we have a convolution layer which takes as input an array* $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$ *and convolves* $\mathbf{x}$ *with the kernel* $\begin{pmatrix} 2 & -1 \end{pmatrix}$*. This layer has a linear activation function. The output is an array of length 4.*

   Now let's design a fully connected layer which computes the same function. It has a linear activation function and no bias, so it computes $\mathbf{y} = \mathbf{W}\mathbf{x}$*, where the output* $\mathbf{y}$ *is a vector of length 4. Give the* 4×3 *weight matrix* $\mathbf{W}$ *which makes this fully connected layer equivalent to the convolution layer above. You don't need to justify your answer, but doing so may help you get partial credit.*

   Hint: first write the values of each output as a linear function of the inputs. To help you check your work, if $\mathbf{x} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$*, your answer should give* $\mathbf{y} = \begin{pmatrix} 2 & 3 & 4 & -3 \end{pmatrix}$*.*

$$\begin{pmatrix} 2 & 0 & 0 \\ -1 & 2 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & -1 \end{pmatrix}$$

Mean: 1.9/2

9. **[1pt]** *Briefly explain one flaw of encoder-decoder architectures for machine translation which do not use attention, and how attention can fix it.*

   An encoder-decoder architecture without attention needs to store all the necessary information about the input sentence in its code vector (final hidden state). An attention-based architecture can refer to the annotation vectors for particular words, reducing the memory pressure on its hidden state.

   Marking: We also gave credit for other answers, such as exploding/vanishing gradients or difficulty of accounting for word order, as long as there was a reasonable argument for why attention helps.

   Mean: 0.89/1

10. **[2pts]** *Recall that in order to add a new primitive operation to Autograd, you need to define a vector-Jacobian product (VJP). To refresh your memory, here is code which defines VJPs for exponentiation and multiplication.*

```
defvjp(exp,          lambda g, ans, x: ans * g)
defvjp(multiply,     lambda g, ans, x, y: y * g,
                     lambda g, ans, x, y: x * g)
```

   The arguments to ***defvjp*** *are the primitive op, followed by functions implementing the VJPs for each of the arguments. The arguments to the VJP function are: the output gradient* ***g****, the output* ***ans*** *of the op, and the arguments fed to the op.*

   (a) **[1pt]** *Write Python code that defines a vector-Jacobian product for* ***sin****.*

```
defvjp(sin, lambda g, ans, x: cos(x) * g)
```

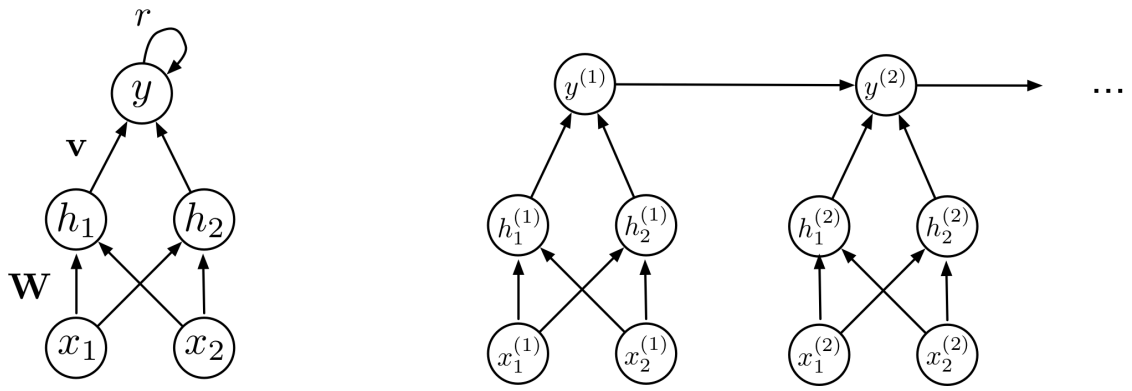
   (b) **[1pt]** *Write Python code that defines a vector-Jacobian product for* ***divide****, the function which computes the elementwise division of two arrays (i.e.* ***divide(x, y)*** *is equivalent to* ***x / y****). (This is floating point division, not integer division.)*

```
defvjp(divide, lambea g, ans, x, y: g / y,
               lambda g, ans, x, y: -g * x / y**2)
```

   Marking: One point off if missing `g`. Half a point off for most other mistakes. Each mistake is only counted once, even if it is made multiple times.

   Mean: 0.9/2

11. **[4pts]** *Suppose we receive two binary sequences* $\mathbf{x_1} = (x_1^{(1)}, \ldots, x_1^{(T)})$ *and* $\mathbf{x_2} = (x_2^{(1)}, \ldots, x_2^{(T)})$ *of equal length, and we would like to design an RNN to determine if they are identical. We will use the following (rather unusual) architecture, drawn with self-loops on the left and unrolled on the right:*



The computation in each time step is as follows:

$$\mathbf{h}^{(t)} = \phi\left(\mathbf{W}\mathbf{x}^{(t)} + \mathbf{b}\right)$$

$$y^{(t)} = \begin{cases} \phi\left(\mathbf{v}^\top \mathbf{h}^{(t)} + r y^{(t-1)} + c\right) & \text{for } t > 1 \\ \phi\left(\mathbf{v}^\top \mathbf{h}^{(t)} + c_0\right) & \text{for } t = 1, \end{cases}$$

where ϕ *denotes the hard threshold activation function*

$$\phi(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

The parameters are a 2×2 *weight matrix* $\mathbf{W}$*, a 2-dimensional bias vector* $\mathbf{b}$*, a 2-dimensional weight vector* $\mathbf{v}$*, a scalar recurrent weight* r*, a scalar bias* c *for all but the first time step, and a separate bias* c_0 *for the first time step.*

We'll use the following strategy. We'll proceed one step at a time, and at time t*, the binary-valued elements* $x_1^{(t)}$ *and* $x_2^{(t)}$ *will be fed as inputs. The output unit* $y^{(t)}$ *at time* t *will compute whether all pairs of elements have matched up to time* t*. The two hidden units* $h_1^{(t)}$ *and* $h_2^{(t)}$ *will help determine if both inputs match at a given time step. Hint: have* $h_1^{(t)}$ *determine if both inputs are 0, and* $h_2^{(t)}$ *detemine if both inputs are 1.*

$$\mathbf{W} = \begin{pmatrix} -1 & -1 \\ 1 & 1 \end{pmatrix}$$
$$\mathbf{b} = \begin{pmatrix} 0.5 \\ -1.5 \end{pmatrix}$$
$$\mathbf{v} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
$$r = 1$$
$$c = -1.5$$
$$c_0 = -0.5$$

Marking: Full marks for any correct solution, regardless of whether it follows the hint. Otherwise, two marks for the part relating to the hidden units (W and b), and two marks for the part relating to the output units ($\mathbf{v}$, r, c, and c_0). Full credit for the hidden units if it correctly implements the hint. Half credit for the hidden units if they somehow compute enough information to solve the task. Full credit for the output units if they're correct assuming the hint, even if the hidden units are wrong.

Mean: 3.5/4

12. **[2pts]** *Suppose we have flipped a coin multiple times, and it came up heads N_H times and tails N_T times. We would like to model the coin as a Bernoulli random variable, and fit the model using maximum likelihood.*

    (a) **[1pt]** *Give the formula for the log-likelihood $\ell(\theta)$, where θ is the probability of heads.*

    $$\ell(\theta) = N_H \log \theta + N_T \log 1 - \theta$$

    (b) **[1pt]** *Solve for the maximum likelihood estimate of θ by setting $\mathrm{d}\ell/\mathrm{d}\theta = 0$.*

    $$\frac{\partial \ell}{\partial \theta} = \frac{N_H}{\theta} + \frac{N_T}{1-\theta} = 0.$$

    The solution occurs where $\theta = N_H/(N_H + N_T)$.

    Mean: 1.7/2

13. **[2pts]** *Recall the CycleGAN architecture for style transfer.*

(a) **[1pt]** *What might go wrong if we eliminate the discriminator terms from the cost function?*

There would be nothing encouraging the generator to match the target style. It could satisfy the reconstruction objective simply by learning, e.g., the identity mapping.

(b) **[1pt]** *What might go wrong if we eliminate the reconstruction terms from the cost function?*

The generator would not be forced to preserve the structure of the original image. (Note: "preserving the structure" is all we're looking for in this question. In practice, what happens if you remove the reconstruction term is often that the generator preserves much of the structure anyway, but misses fine-grained details, such as hallucinating pedestrians in a semantic segmentation task.)

Mean: 1.5/2

14. **[2pts]** *Recall that a GAN could, in principle, be trained using the following minimax formulation, where G is the generator function, D is the probability the discriminator assigns to the sample being data, and $\mathcal{J}_D$ and $\mathcal{J}_G$ are the cost functions for the discriminator and generator, respectively.*

$$\begin{aligned}\mathcal{J}_D &= \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[-\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}}[-\log(1 - D(G(\mathbf{z})))] \\ \mathcal{J}_G &= -\mathcal{J}_D \\ &= \text{const} + \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))]\end{aligned}$$

However, in practice, the generator is usually trained with a different loss function.

(a) **[1pt]** *What cost function do we typically use for the generator?*

$$-\log D(G(\mathbf{z}))$$

Marking: Half a point for simply saying "cross-entropy". Half a point if almost correct (e.g. missing minus sign). We also accepted squared error, since this was used in Assignment 4.

(b) **[1pt]** *What is the reason to use this cost function rather than the one given above?*

In the minimax formulation, the generator's loss saturates when the generator is doing very badly (i.e. when the discriminator can easily recognize its samples as fake). The alternative objective puts a large emphasis on small differences

in $D(G(Z))$ when $D(G(z))$ is close to 0, so that the generator still has a strong gradient signal in this case.

Marking: We also accepted "convexity" as an answer. (This was part of our discussion of logistic regression. GAN training isn't actually convex even with the modified objective, but we were lenient about this.) If the answer to part (a) was squared error, we still gave full credit for talking about saturation, even though squared error loss doesn't actually help with this.

Mean: 1.5/2

15. **[2pts]** *We've covered autoregressive generative models based on both convolutional networks and RNNs.*

    (a) **[1pt]** *Give one advantage of using a convolutional network rather than an RNN.*
    The intended answer was that RNNs require a `for`-loop over time steps for both the forward and backward passes. At training time, the conv net's predictions can be made all at once. Some alternatives we accepted:

    - avoid exploding/vanishing gradients (full credit)
    - can visualize CNN features (full credit)
    - saying it "trains faster", without a correct explanation (half credit)

    (b) **[1pt]** *Give one advantage of using an RNN rather than a convolutional network.*
    The intended answer was that conv nets can only use information from a limited context (receptive field). RNNs can use information from the entire sequence that was already generated. Some alternatives we accepted:

    - RNNs can learn the distribution just as well with fewer weights/units/connections (full credit)
    - RNNs can process sequences of varying lengths (half credit — this is true of CNNs and RNNs in some contexts, but not autoregressive models)

    Mean: 1.4/2

16. **[3pts]** *Reversible architectures are based on a reversible block. Let's modify the definition of the reversible block:*

$$\mathbf{y}_1 = \mathbf{r} \circ \mathbf{x}_1 + \mathcal{F}(\mathbf{x}_2)$$
$$\mathbf{y}_2 = \mathbf{s} \circ \mathbf{x}_2,$$

where $\circ$ denotes elementwise multiplication. This modified block is identical to the ordinary reversible block, except that the inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ are multiplied elementwise by vectors $\mathbf{r}$ and $\mathbf{s}$, all of whose entries are positive.

You don't need to justify your answers for this question, but doing so may help you receive partial credit.

(a) **[1pt]** *Give equations for inverting this block, i.e. computing $\mathbf{x}_1$ and $\mathbf{x}_2$ from $\mathbf{y}_1$ and $\mathbf{y}_2$. You may use / to denote elementwise division.*

$$\mathbf{x}_2 = \mathbf{y}_2/\mathbf{s}$$
$$\mathbf{x}_1 = (\mathbf{y}_1 - \mathcal{F}(\mathbf{x}_2))/\mathbf{r}$$

Marking: Half a point for each expression. Order doesn't matter.

(b) **[1pt]** *Give a formula for the Jacobian $\partial\mathbf{y}/\partial\mathbf{x}$, where $\mathbf{y}$ denotes the concatenation of $\mathbf{y}_1$ and $\mathbf{y}_2$.*

$$\begin{pmatrix} \text{diag}(\mathbf{r}) & \frac{\partial \mathcal{F}}{\partial \mathbf{x}_2} \\ \mathbf{0} & \text{diag}(\mathbf{s}) \end{pmatrix}$$

Marking: Half a point off (for the entire question) for answers that are correct apart from treating $\mathbf{r}$ and $\mathbf{s}$ as scalars. Half a point for answers that have the right idea but have a mistake. Mistakes aren't double counted between parts. No marks for just writing the definition of the Jacobian. No marks for putting all 1's on the diagonal.

(c) **[1pt]** *Give a formula for the determinant of the Jacobian from part (b).*

$$\left(\prod_{i=1}^{d} r_i\right)\left(\prod_{i=1}^{d} s_i\right)$$

Marking: Full credit for observing that the determinant is the product of the diagonal entries because of the upper triangular structure. (Though half a point off if the formula is then wrong for reasons that didn't already lose points.) Most other answers got no credit.

Mean: 1.7/3

17. **[2pts]** *Suppose we have an MDP with two time steps. It has an initial state distribution $p(\mathbf{s}_1)$, transition probabilities $p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$, and deterministic reward function $r(\mathbf{s}, \mathbf{a})$. The agent is currently following a stochastic policy $\pi_{\boldsymbol{\theta}}(\mathbf{a} \mid \mathbf{s})$ parameterized by $\boldsymbol{\theta}$.*

    (a) **[1pt]** *Give the formula for the probability $p(\tau)$ of a rollout $\tau = (\mathbf{s}_1, \mathbf{a}_1, \mathbf{s}_2, \mathbf{a}_2)$.*

    $$p(\mathbf{s}_1)\, \pi_{\boldsymbol{\theta}}(\mathbf{a}_1 \mid \mathbf{s}_1)\, p(\mathbf{s}_2 \mid \mathbf{s}_1, \mathbf{a}_1)\, \pi_{\boldsymbol{\theta}}(\mathbf{a}_2 \mid \mathbf{s}_2)$$

    (b) **[1pt]** *What is the function that REINFORCE is trying to maximize with respect to $\boldsymbol{\theta}$? (You can give your answer in terms of $p(\tau)$.)*

    $$\mathbb{E}_{p(\tau)}\left[r(\mathbf{s}_1, \mathbf{a}_1) + r(\mathbf{s}_2, \mathbf{a}_2)\right]$$

    Marking: Half a point off for each error

    Mean: 1.2/2

18. **[1pt]** *Recall that the discounted return is defined as:*

    $$G_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i},$$

    where γ is the discount factor and r_t is the reward at time t. Give the definition of the action-value function $Q^\pi(\mathbf{s}, \mathbf{a})$ for policy π, state $\mathbf{s}$, and action $\mathbf{a}$. You can either give an equation or explain it verbally.

    $$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}]$$

    This is the expected discounted return if the agent starts in state $\mathbf{s}_t$, chooses action $\mathbf{a}_t$, and then continues to follow policy π.

    Mean: 0.7/1